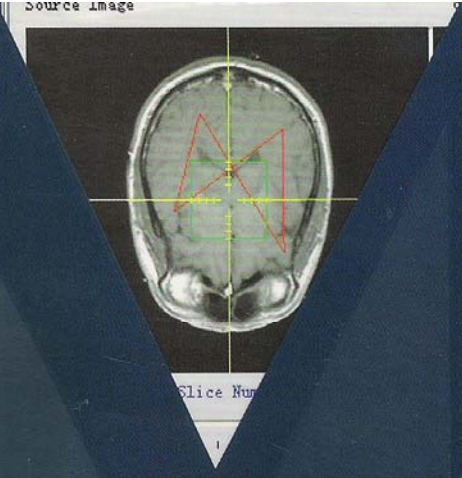
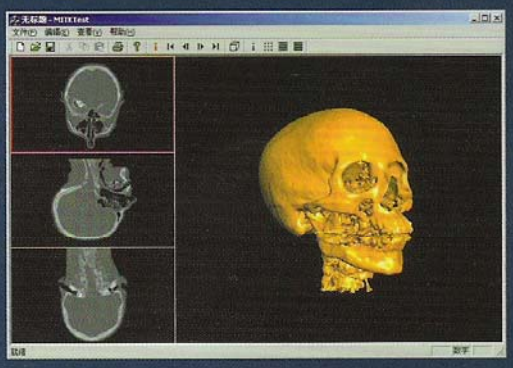



田　捷　赵明昌　何晖光◎编著

# 集成化医学影像算法平台
# 理论与实践

DEVELOPMENT AND IMPLEMENTATION OF MEDICAL IMAGING TOOLKIT



清华大学出版社

# 目 录

# 1 绪论

## 1.1 医学影像算法平台研究的背景及意义

自从德国科学家伦琴在 1895 年发明X射线以来，CT（计算机断层成像）、MRI（核磁共振成像）、CR（计算机X线成像）、B超、电子内窥镜等现代医学影像设备先后出现，使得传统的医学诊断方式发生了革命性的变化。使用计算机对医学影像设备采集到的影像进行处理这一技术被称为医学影像处理与分析，它可以辅助医生进行更好、更准确的诊断。随着现代计算机科学技术的发展，医学影像处理与分析越来越多地受到人们的重视，现在已经成为一门新兴的发展迅速的交叉科学领域[1]。

21 世纪是以人为中心的世纪，如何更全面地掌握和更有效地利用人的信息将成为许多高新技术产业的关键因素，而跟全社会人民的医疗保健和健康事业息息相关的医学影像处理与分析学科也将在 21 世纪得到快速的发展。医学影像处理与分析是计算机信息学、物理学和医学等相结合的产物，在 20 世纪内经历了学科形成、发展和快速发展的过程，如果说 20 世纪是医学影像形成和快速发展的世纪，在 21 世纪就将是医学影像广泛应用的世纪。

目前随着改革开放和国力不断提高，我国从国外进口了越来越多的高精密的医疗设备并在医疗临床上广泛使用。然而，由于国内缺乏配套的开发队伍，也没有形成开展跨学科开发研究的机制，一般使用的都是国外公司随机器附带的软件，使得我国对这些高精密医疗设备利用及研发的速度缓慢。考虑到目前以硬件设备为主的医疗器械都必须配套相应的计算机软件，而软件的成本和开支将逐步超过硬件，目前我国已经逐步具备了医疗器械的生产能力，但是高质量的配套软件仍然相当缺乏，因为软件的编制依赖于对科学问题的数学描述和计算方法。21 世纪的产业，医学影像的计算模型和计算方法将成为制约医学软件业的首要因素，抓住这一契机将成为我国医疗信息高质量稳定发展和参与国际竞争的重中之重。而积累发展具有我国自主知识产权的高质量的医学影像软件平台，尤其是底层的算法研发平台，对促进我国医疗仪器设备的应用、尤其是医学影像软件业的持续发展，并直接造福于人民的医疗保健和健康事业是非常重要的。

由于医学影像领域的研究涉及的面非常宽，研究本身需要多学科的交叉，这就导致开发医学影像领域的高质量软件，尤其是算法研发平台非常困难。但是一旦研究成功之后，受益面将会非常大，今后的发展前途非常广阔。中国是世界上人口最多的国家，国际间的竞争，国家的安全本质上是国民平均素质之间的竞争。利用现代信息学已取得的科学成果，对人体医学影像信息进行挖掘，建立相应的理论、方法并开发相应的软件开发平台，必将为人类对自身的认知、疾病的诊断和治疗、国民教育及商业软件的开发提供极好的机会，使得医学影像信息学科在中国的发展奠定良好基础。可以预计：医学影像将会扩展到医院的每个科室，每个病人，以至全社会每个家庭成员，成为每个人健康状况记录的基本信息源之一。

在发达国家，尤其是美国对开发高质量的医学影像软件和算法研发平台非常重视。美国国家卫生院下属的国立医学图书馆近年投入巨资支持三家科研机构（包括University of North Carolina、University of Utah、University of Pennsylvania）开发医学影像分割与配准算法的研发平台ITK（Insight Segmentation and Registration Toolkit）[2]，现在已经开发出初步的版本。在医学影像领域的主流国际会议SPIE Medical Imaging 在 2004 年的年会上有一个专门的Session，叫做Visualization Toolkits，探讨医学影像领域内算法研发平台的研究；而在Medical Image Computing & Computer Assisted Intervention（MICCAI）2003 会议上有一个专门的Workshop，叫做Software Development Issues for Medical Imaging Computing & Computer Assisted Interventions，也是探讨未来在医学影像领域内高质量软件，尤其是算法研发平台（Toolkits）的研究问题。

考虑到医学影像技术在未来世纪的重要性，以及国内外相关研究人员对一个集成化的医学影像算法研发平台的需求，本书的工作是希望通过 MITK 的研究与开发以及免费发行，能够推广医学影像软件在国内的广泛使用，为我国民族软件事业和医疗事业做出自己微薄的贡献。

## 1.2 医学影像算法平台研究的内容

对于一个实用的医学影像算法平台来说，有两个大的方面需要研究：第一个方面是整体框架的研究；第二个方面是医学影像算法方面的研究，又包括医学影像分割、医学影像配准、三维可视化这三大研究领域。

### 1.2.1 整体框架的研究

医学影像处理与分析的目的是从数据到知识,其中在算法层面包含医学影像分割、医学影像配准（包括图像信息融合）、三维可视化（包括表面绘制、体绘制和数字几何处理）等；在数据表达层面，医学影像数据具有多源（CT、MRI、PET 等）、多维（二维、三维、四维及更高维）、多模态（形态、生理、心理、病理）、异构信息（像素矩阵表示的图像、像素集合表示的目标、符号表示的知识）等特点。如何对这么多的算法以及这么复杂的数据进行抽象，得到统一的数据表达和高效处理方法，以便能够在一个统一的软件框架中高效、精确地来处理是一个比较困难的问题。另外，目前研究人员在各个算法分支已经提出了多种算法，并且现在对于各种算法性能的评价的研究也越来越受到重视，比如对分割算法准确性的评价已经成为一个非常前沿的问题。如果将这些算法置入一个统一的计算框架，组成一条管道式的流水线，在统一的数据表达、统一的算法框架以及统一的参数设置下，去研究各个算法的性能、前一算法对后续算法性能的影响等等，不仅能更好地在局部上去理解单个算法的性能指标、不同算法的性能差异，还有助于在整体上去理解和把握不同算法之间的影响、整体的瓶颈所在、不同信息的整合等。

### 1.2.2 医学影像算法的研究

目前医学影像算法方面的研究主要分为医学影像分割、医学影像配准、三维可视化这三大研究领域，其中每一类下面又可根据一定的原则分为不同的子类，并且每一个子类都有非常多不同的算法。

(1) 医学影像分割算法

医学影像分割的主要目的是将医学影像中感兴趣的物体（一般是病灶区）提取出来，以前医生往往是通过自己的经验手工分割病灶区，而分割的目的就是尽量自动、准确地将这些病灶区提取出来。图像分割是图像处理、图像分析和计算机视觉等领域最经典的研究课题之一，也是最大的难点之一，其理论和方法至今尚未获得圆满的解决。在医学领域，图像分割也一直是一个非常活跃的研究课题，吸引着很多的研究人员去探索这个问题。

医学影像分割可以大致分为以下几种类型：基于区域的分割方法，包括阈值分割[3][4]、区域生长等[5]；基于边缘的分割方法，包括梯度算子、Roberts算子、

Sobel算子、Prewitt算子、Laplacian算子和Kirsch算子等[1]；基于形变模型的方法，包括Active Contour[6]、Level Set[7]、Fast Marching[8]等；基于人机交互的方法，包括手工交互式分割、Live Wire分割[9]等。

(2) 医学影像配准算法

医学影像配准的主要目的是将两幅医学影像上的对应点达到空间上的一致，也就是找出两幅图像中对应于人体同一位置的点的对应关系，从而可以让医生把多幅不同模态的图像融合起来获得更多的信息。目前对于多模态医学影像的配准，以及对具有弹性形变的图像之间的配准都是非常热门的研究领域。

医学影像配准算法也有很多不同的类型，根据空间维数的数目和时间是否为附加维这两点可以分为 2D/2D、2D/3D、3D/3D图像配准；根据配准所依据的特征可分为基于外部特征和基于内部特征两大类；根据变换的性质可分为刚性变换、仿射变换、投影变换和曲线变换四种；根据用户交互性的多少，可分为交互的，半自动的和自动的三种；根据配准的医学图像模态可以分为单模图像之间的配准、多模图像之间的配准和患者和模态之间的配准这四种；根据配准过程中变换参数确定的方式可以分为两种，一种是通过直接计算公式来得到，另一种是通过在参数空间中寻求某个函数的最优解来得到[10]。

(3) 三维可视化算法

三维可视化的主要目的是将医学影像设备得到的一系列的二维的切片，使用计算机图形学的技术，构造出一个器官的三维模型，并且非常逼真地显示给医生，它是科学可视化研究的一个重要的分支。目前随着医疗设备的进展，医学影像数据集越来越大，海量的数据给传统的三维重建与绘制技术带来了非常大的挑战，所以目前大数据量的三维医学影像的快速重建和绘制技术也是一个非常有挑战性的问题。

三维可视化可以分为三个大类：表面绘制（Surface Rendering）、体绘制（Volume Rendering）和数字几何处理。表面绘制包括经典的Marching Cubes算法[11][12]，Cuberille[13]算法等；体绘制包括经典的Ray Casting[14]算法、Splatting[15]算法、Shear Warp[16]算法，以及新近出现的基于图形硬件GPU的加速算法[17][18]等；而数字几何处理[19][20]是在SigGraph 2001 上才提出的一个新概念，用于对表面绘制产生的网格进行处理，包括网格化简、网格细分、网

格平滑等算法。

## 1.3 医学影像算法平台的国内外研究现状

如上所述,医学影像处理算法的三个主要研究领域目前都已经有非常多的成熟的算法，并且新的算法还在不断出现。除了在算法研究方面的努力外，一些研究组织为了更好地利用现有的算法，避免重复的劳动，开发了许多算法平台（Algorithms Toolkit），这些算法平台不仅封装了他们自己的算法，还封装了很多已经成熟的相关算法。这些算法平台极大地便利了医学影像领域的研究者，他们可以在这些平台上来创建自己的实验环境，验证自己的算法，而不用从头再写一些已经成熟的算法。可以作个比喻，这些平台的作用就相当于 Matlab 对于研究人员的作用一样，不过它们是专门针对医学影像领域的。目前在医学影像研究人员中使用最广泛的两个算法平台是 VTK(Visualization Toolkit)和 ITK，虽然还有其它一些平台，但是它们都不系统，只是针对某一个特定的领域的，另外，VTK 和 ITK 也是对 MITK 的设计影响最大，最相关的两个算法平台，所以下面分别介绍一下它们的发展历程和发展现状，并给出它们的局限性。

### 1.3.1 VTK 简介

VTK（Visualization ToolKit）[21]是一套进行数据可视化的开发工具包，最早在 1993 年 12 月由美国GE公司研发部门的 Will Schroeder和 Ken Martin首次发布，当时是作为“The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics”这本书的配套软件赠送,后来在 1998 年时，此书出版第二版，并且在此五年期间使用VTK的人数不断增加，形成了一个社区，VTK也以开放源码（Open Source）的形式开发。现在这本书已经出到第三版[22]，同时VTK也由美国Kitware公司负责维护，全世界的开发人员都可以贡献自己的力量。

VTK完全采用面向对象的设计思想来设计与开发[23]，其提供了非常强大的功能，提供了超过 300 个C++类，并且可以支持跨平台开发，支持Windows、Unix、Linux等多种平台[24]。值得一提的是，VTK并不是专门针对医学影像领域开发的平台，它的主要目标是通用可视化领域，但是它仍然在医学影像领域得到了比较广泛的应用，这一方面与其几个主力研发人员在GE公司的背景分不开（他们是表面绘制经典算法Marching Cubes算法的发明人），另外一个方面是因为VTK里面提供了表面绘制、体绘制、一部分数字几何处理算法，这些都对医学

影像领域内的研究有所帮助。发展到现在，VTK的稳定版本已经发行到 4.2 版本，并且新的 5.0 版本也在持续地开发中，已经成为通用可视化领域内最负盛名的软件开发包，也在医学影像领域内赢得了尊敬。

### 1.3.2 ITK 简介

ITK（Insight Segmentation and Registration Toolkit）[2] 的主要目的是提供一个医学影像分割与配准的算法平台，它的起源还是基于美国的可视人体项目[25]，当可视人体的数据采集完成以后，对这些数据进行配准并分割就形成了迫切的需求，因此在 1999 年，由美国NIH（国家卫生院）下属的NLM（国立医学图书馆）发起了一个投标活动，要出资资助开发一个分割与配准的算法研发平台，作为可视人体项目的一个工具，对可视人体项目得到的数据进行处理与分析。最终选中六家单位合作开发，包括University of North Carolina，University of Utah，University of Pennsylvania三个大学，以及Kitware、GE、Insightful三个商业公司。从 1999 年 10 月开始，到 2002 年 10 月，已经完成了第一个三年计划，并成功发行了ITK 1.0。

与VTK相同的是，ITK的框架设计也是由Kitware公司来完成的；但是与VTK严重不同的是，它们的设计风格截然不同。ITK大量使用了 1998 年以后ANSI C++标准里面的新特性，尤其是Template（模板），并且ITK整个就是基于范型编程（Generic Programming）[26]这种设计思想来设计与实现的。ITK也可以支持跨平台开发，支持Windows、Unix、Linux等多种平台，目前也是采用Open Source的形式发行，最大限度地推广它。经过四年多的开发，目前ITK的稳定版本已经发行到 1.6，提供了几乎所有主流的医学影像分割与配准算法[27]，并且现在还一直在持续地进化，它已经并且将继续为医学影像领域内的研究人员提供一个分割与配准算法的仓库和基础。

### 1.3.3 VTK 和 ITK 的局限性

VTK 和 ITK 目前已经成为国际上非常知名的可视化与医学影像分割与配准的算法研发平台，已经并且正在为研究人员提供着非常多的便利。但是由于一些原因，导致 VTK 或者 ITK 或者 VTK+ITK 有一些自己的缺陷，影响了其在更大范围的广泛使用。

首先，因为 ITK 并不提供可视化的能力，所以一般要与 VTK 联合起来使用

才能构成一个比较完整的医学影像的处理与分析系统。首先使用 ITK 进行影像的配准、分割，然后再使用 VTK 进行三维可视化，观看结果，这样就极大地增加了复杂度。尤其是 VTK 和 ITK 所使用的编程风格完全不同，VTK 开发的比较早，在 1998 年 ANSI C++ 标准制定之前就已经比较成型，所以使用的是传统的 Object-Oriented（面向对象）的设计和开发方法；而 ITK 是在 1999 年才开始开发的，所以运用了许多新的 C++语言的特性，以及 Generic Programming（范型编程）的设计和开发方法。这种编程风格上的不一致，导致了同时使用 VTK 和 ITK 时，必须学习两套规模都相当庞大的开发平台，对使用者造成了一定的学习难度。

其次，VTK 是一个面向通用可视化领域的一个开发平台，并不是专门针对医学领域的。VTK 的规模相当庞大，里面的算法也很多，这样一方面使得它的适用面非常宽，有很多领域的研究人员都可以使用它，但是另外一方面也使得它的复杂度大大增加。同时因为要照顾到各个领域，在设计 VTK 的时候，主要目标是一个通用的、灵活的框架，并没有对某一个特定的算法进行优化，因此 VTK 的速度并不是太令人满意。

最后，ITK 是专门针对医学影像的分割和配准而实现的一个开发平台，因为历史性的原因，ITK 并没有重新实现可视化的功能，而是利用 VTK 的可视化的功能。对 ITK 来说，设计的时候利用了非常多的现代 C++语言的新特性，并且大量使用了模板。这本来是一件非常好的事情，但是因为 C++的新标准刚颁布没有几年，导致现在有很多编译器不能完全编译 ITK；并且因为 ITK 里面大量依赖于 STL 和模板，导致最终只能编译成一个静态的函数库，而不能实现动态的加载；同时最重要的一点是，有很多医学影像领域的研究人员对 C++的新标准以及模板编程技术了解的不是很深入，导致他们在使用 ITK 时感觉就象在学习一门新的编程语言。由于上面的这些原因，现在 ITK 还只是在一个比较小的范围内被使用。

## 1.4 本书的主要内容

考虑到上面这些因素，本书介绍的主要是我们实验室研发的集成化的医学影像处理与分析算法研发平台——MITK（Medical Imaging ToolKit）。其目的很简单，所谓集成化，就是指在一个统一的框架里面来实现医学影像分割、配准、

三维可视化等算法，弥补 VTK+ITK 的缺憾。MITK 并不是要取代 VTK 和 ITK，而只是给医学影像领域内的研究人员和开发人员提供另外的一个可用选择，用来丰富国际上的医学影像算法平台。另外，为了验证使用 MITK 开发复杂医学影像实用系统的能力，我们还基于 MITK 设计并实现了三维医学影像处理与分析系统 3DMed。Kitware 公司也曾经使用 VTK 来开发了一个医学影像可视化系统 VolView，但可惜的是这是一个商业软件，即使用于科研目的也是需要购买 License 的。MITK 和 3DMed 均作为免费软件（Freeware）在 Internet 上发布 (www.mitk.net，www.3dmed.net)。本书工作的另外一层意义就是希望能够推广医学影像软件在国内的广泛使用。

具体来讲，本书介绍的内容主要集中在两个方面：第一个方面是算法研究方面，在表面绘制的算法尤其是大规模医学数据的可视化算法的研究方面作了一些探讨，这个部分主要在第三章中介绍；第二个方面是平台的框架设计以及分割、配准、可视化算法在 MITK 中的实现方面，主要是设计并实现集成化的医学影像处理与分析算法平台 MITK，这个部分的内容分散在本书的各个章节。本书的主要内容包括两个方面：

一是研究并设计实现了一个集成化的医学影像处理与分析算法平台——MITK（Medical Imaging ToolKit）。MITK 在一个统一的框架里面实现了医学影像处理与分析的三大研究领域的算法，包括医学影像分割、医学影像配准以及三维可视化，而国际上相同类型的平台如 VTK 和 ITK 均只实现了一部分功能，并且它们还具有完全不同的框架和风格。另外，MITK 是专门针对医学影像这一特定领域的，遵循“小而精”的原则，对一些重要算法作了特定的优化，支持新的 CPU 指令集 SSE 的优化以及新的显卡中的 GPU 的利用。为了最大限度地使 MITK 得到使用，其在 Internet 上面向国际免费发行，相关科研人员可以免费地下载并在自己的科研工作中进行二次开发。

二是基于 MITK 设计并开发了三维医学影像处理与分析系统 3DMed，一方面证明了 MITK 可以胜任现实世界当中医学影像领域复杂软件系统对算法平台的需求，另一方面的目的是为相关科研人员和普通用户提供一个易于使用的软件工具。目前 3DMed 也作为免费软件发行，如果 MITK 和 3DMed 能够推广医学影像软件在国内的广泛使用，那么也就体现出本书工作的意义所在了。

本书以MITK的研究与设计为主线贯穿始终，在软件设计框图的绘制上，始

终遵循UML（统一建模语言）[28]的规范。内容安排如下：第二章介绍MITK的总体框架的设计；第三章侧重介绍在面绘制算法研究方面的工作，包括基于单层表面跟踪的重建算法和基于点的重建算法，并且给出了面绘制算法在MITK中的实现；第四章介绍MITK中的一个重要部分，即体绘制的框架设计及具体的算法实现；第五章给出了在三维人机交互方面的一些探索，并给出了其在MITK中的框架和实现；第六、七章分别给出了不同的分割算法和配准算法在MITK中的实现；第八章介绍了DICOM标准以及其在MITK中的实现，第九，十两章分别介绍应用MITK开发项目和MITK的扩展功能，以便读者可以方便的使用MITK；第十一章介绍基于MITK算法平台的三维医学影像处理与分析实用软件系统3DMed的设计与实现；第十二章介绍了 3DMed中的Plugin，这样读者也可以将自己的算法用plugin开发出来，集成到 3dMed中。附录A介绍一些医学影像数据集的资源，方便读者用来测试自己算法。附录B介绍了MITK论坛的情况。

参考文献

1. 田捷，包尚联，周明全. 医学影像处理与分析. 北京：电子工业出版社，2003.
2. Insight Segmentation and Registration Toolkit, http://www.itk.org.
3. Y. J. Zhang, J. J. Gerbrands. Transition region determination based thresholding. Pattern Recognition Letter, 1991, 12:13-23.
4. P.Sahoo, C.Wilkins and J.Yeager, Threshold selection using Renyi's entropy. Pattern Recognition, 1997, 30(1):71-84.
5. I.N. Manousakas, P.E. Undrill, G.G. Cameron, T.W. Redpath, Split-and-merge segmentation of magnetic resonance medical images: performance evaluation and extension to three dimensions. Computers and Biomedical Research, 1998, 31:393–412.
6. M. Kass, A. Witkin, and D. Terzopoulos, Snakes - Active Contour Models， International Journal of Computer Vision, 1(4): 321-331, 1987.
7. S. Osher and J. Sethian, Fronts propagating with curvature dependent speed， J. Comput. Phys., Vol.79, pp.12-49, 1988.
8. J.A.Sethian, Fast marching methods， SIAM Rev., 41(1999), pp.199-235.
9. A. X. Falcao, J. K. Udupa, S. Samarasekera, Shoba Sharma, User-steered Image Segmentation Paradigms : Live Wire and Live Lane, Graphic models and Image Processing, 1998, 60:233-260.
10. J. B. A. Maintz and M. A. Viergever, A survey of medical image registration, Medical Image Analysis, 2(1):1-36, 1998.

11. W. Lorensen and H. Cline, Marching cubes: a high resolution 3D surface construction algorithm. ACM Computer Graphics, 21(4): pp. 163 –170, 1987.
12. Gregory M. Nielson, On Marching Cubes, IEEE Transaction on Visualization and Computer Graphics, Vol. 9, No. 3, pp. 283-297, 2003.
13. G. T. Herman, H. K. Liu, Three-Dimensional Display of Human Organs form Computed Tomography, Computer Graphics & Image Processing, 1979, Vol. 9, pp. 1-29.
14. M. Levoy, Display of surfaces from volume data, IEEE Transaction on Computer Graphics and Applications, 1988, 8(3): 29-37.
15. D. Laur and P. Hanrahan, Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering, ACM Computer Graphics, Proc. SIGGRAPH '93, 25(4):285–288, July 1991.
16. P. Lacroute and M. Levoy, Fast volume rendering using a shear-warp factorization of the viewing transformation, Proc. SIGGRAPH '94, pp. 451- 458, 1994.
17. K. Engel, M. Kraus, and T. Ertl, High-quality pre-integrated volume rendering using hardware accelerated pixel shading, in Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware 2001.
18. C. Resk-Salama, K. Engel, M. Bauer, G. Greiner, T. Ertl， Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization，in Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware 2000.
19. Wim Sweldens, Peter Schr oder, Digital Geometry Processing, Course Notes for SigGraph'2001, Los Angeles , California, Aug. 12 , 2001.
20. 何晖光，数字几何的研究及其在医学可视化中的应用，[博士论文]，北京：中科院自动化所，2002.
21. Visualization Toolkit, http://www.vtk.org.
22. Will Schroeder, Ken Martin, Bill Lorensen Schroeder, The Visualization Toolkit: An Object Oriented Approach to 3D Graphics 3rd Edition, Kitware, Inc. Publisher, 2003.
23. William J. Schroeder, Kenneth M. Martin, William E. Lorensen, "The Design and Implementation Of An Object-Oriented Toolkit For 3D Graphics And Visualization", Proc. Of IEEE Visualization '96.
24. William J. Schroeder, Lisa S. Avila, William Hoffman, "Visualizing with VTK: A Tutorial", IEEE Transaction on Computer Graphics and Applications, Vol. 20, No. 5, pp. 20-27, 2000.
25. Ackerman, M. J., The Visible Human Project, Medicine Meets Virtual Reality II: Interactive Technology and Healthcare, pp. 5-7
26. Matthew H. Austern, Generic Programming and the Stl : Using and Extending the C++

Standard Template Library, Addison-Wesley Professional Computing Series, 1998.

27. Luis Ibanez, Will Schroeder, Lydia Ng, Josh Cates. The ITK Software Guide: The Insight Segmentation and Registration Toolkit (version 1.4), Kitware, Inc. Publisher, 2003.
28. Fowler M., Scott K. UML Distilled Second Edition: A Brief Guide to the Standard Object Modeling Language, Addison Wesley, 2000.

# 2 MITK 的总体设计

为了在一个统一的框架里面实现一个具有一致接口的、可复用的、包括可视化、分割、配准功能的集成化的医学影像算法平台，MITK使用了基于数据流的模型[1] ，把算法对象和数据对象分开来考虑，减少它们之间的藕合性，同时形成一个优雅的算法的计算框架。

另外，为了增加其易用性，MITK并没有象ITK那样使用范型编程和模板，而是采用的传统的面向对象的设计方法[2]。另外，MITK的设计还吸收了近几年流行起来的基于Design Pattern[3]的设计方法。所谓Design Pattern，就是软件工业界在长期的编写大型、复杂软件的过程中总结出来的一些经典的解决问题的方法，其中每个都是行之有效的，经过实践的检验。Design Pattern结合了面向对象设计与分析，面向特定问题的分析等设计方法，可以很大程度地提高软件开发的质量和效率。虽然已经有研究者总结了流行最广泛的Design Pattern，但是这些都是面向通用领域的比较成功的设计软件的方法，面向医学影像处理与分析这一特定领域的软件设计的Design Pattern还没有得到充分的重视。本书通过对MITK的研究与设计，同时还探讨了Design Pattern在医学影像领域内软件开发的应用。

本章首先给出了 MITK 的总体设计目标，然后介绍它的总体框架，从数据模型和算法模型两个方面给出说明，最后给出了 MITK 整体框架里面的一些基础设施，便于后续章节的描述。

## 2.1 MITK 的设计目标

对于软件设计，尤其是特定领域内的复杂软件设计，必须事先有一个非常明确的设计目标。MITK 从一开始设计，就始终追求以下几个高层的设计目标：

### 2.1.1 统一的风格

VTK和ITK由于历史性的原因，使用了不同的编程风格。VTK在 1998 年ANSI C++ 标准制定之前就已经比较成型，所以使用的是传统的Object-Oriented（面向对象）的设计和开发方法；而ITK是在 1999 年才开始开发的，所以运用了许多新的C++标准规定的语言特性，以及Generic Programming（范型编程）的设计和开发方法。这种编程风格上的不一致，给VTK+ITK的使用者带来了很大的不方

便。而MITK使用统一的面向对象的设计方法，再加上一些Design Patterns（设计模式）[3]的使用，提供一个统一的编程风格和整体框架。

### 2.1.2 有限目标

MITK 是专门面向医学影像领域的，只关注于这一特定领域内的算法，不追求大而全，只追求少而精。例如 MITK 中可视化算法只包括规则数据场（医学影像设备得到的数据场即为此类）的支持，分割算法的输出也只限于是一个二值数据场。这样的设计准则简化了整个 MITK，使得其保持在一个中等的规模，但同时提供了必须的功能，包括主流的可视化、分割和配准算法的实现。

### 2.1.3 可移植性

为了使 MITK 能够得到最广泛的应用，可移植性是非常重要的一个环节。整个 MITK 的代码全部使用 ANSI C++编写，没有使用任何编译器提供的特殊关键字或者特殊函数，并且尽量降低平台相关的代码量。在整个 MITK 中，与平台相关的部分就是与窗口系统打交道的部分，此处针对不同的操作系统写了不同的代码，目前支持 Windows 系列操作系统、Unix、Linux 等。而 MITK 目前可以在多数主流的 C++编译器下编译通过，包括对模板支持不完善的编译器。

### 2.1.4 代码优化

因为医学影像处理与分析算法中很多算法计算量大，尤其是可视化算法，对实时性要求很高，这些就需要对代码进行优化。因为MITK的规模保持在中等，这就使得对一些关键算法进行优化成为可能。MITK支持对CPU的扩展指令集的使用，比如Intel的MMX、SSE指令集，为了不至于违背可移植性目标，MITK中在使用SSE等指令集时，并没有直接使用汇编语言，而是使用了编译器提供的Intrinsics指令，目前MITK当中实现了SSE加速的矩阵和矢量运算、双线性和三线性插值计算等；另外MITK还支持对目前主流显卡中GPU的编程，实现了使用纹理映射进行Volume Rendering（体绘制）的加速算法[4][5]。

## 2.2 MITK 的整体计算框架

医学影像处理与分析技术包含可视化、分割、配准三大类算法，每一类下面又都有很多不同的方法，医学影像数据本身也包含多维（一维、二维、三维）、多源（CT、MRI 等）、多态（以像素矩阵形式表达的图像信息、以像素集合形式

表达的目标信息以及以几何形状形式表达的知识信息）的特点，这些都决定了将其整合进一个统一的框架之内是一个比较复杂的过程，必须经过抽象、建模，才能得到一个比较灵活、可用的整体计算框架。有了计算框架以后，还可以在相同的条件之下比较同一种类不同算法的性能、效果，进行算法的比较；另外还可以进行算法的评价，比如对分割算法、配准算法的准确性进行评价。

### 2.2.1 基于数据流模型的整体框架

正如前面提到的一样，MITK 的计算框架采用基于数据流的模型，以数据处理为中心，将算法和数据对象分开考虑，这种模型很适合那些牵涉大量不同的算法、需要处理不同类型数据的领域的应用。

在MITK的数据流模型中，数据和算法均使用对象表示；一个算法被抽象成一个滤波器（Filter），它接受一个输入，生成一个输出，其中输入和输出均为数据对象（Data），一个算法的输出可以作为另外一个算法的输入。通过这样的模型，一连串的算法可以被串成一个流水线，组成统一的计算框架，如 图 2-1 所示。

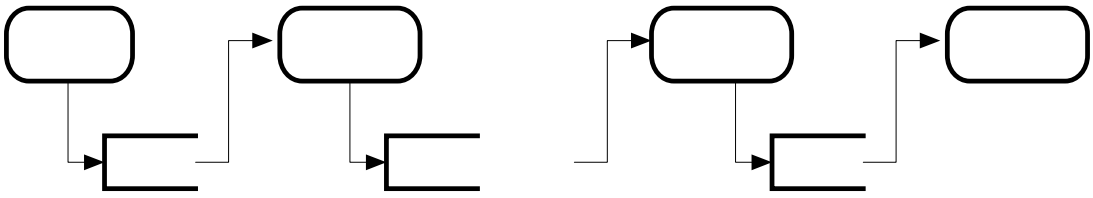


图 2-1 MITK 的计算框架

在图中，Data 表示抽象的数据对象，Source、Filter 和 Target 分别表示三种不同的抽象算法对象，它们的意义分别如下所示：

Data: Data 是对医学影像领域内要处理的数据所进行的抽象。对于医学影像处理与分析系统来说，要能够处理多种不同的数据，在 MITK 中通过 Data 的各个具体的子类来描述不同的数据对象，代表了数据流模型中最重要的数据对象。在下一小节将会具体讲述 MITK 中的数据模型。

Source: Source 是算法类的一种，但是它只有输出，没有输入，代表一个流水线的起源。Source 的作用是负责生成整个流水线的起始数据，比如从磁盘上

读文件，或者用某些算法生成数据。

Filter: Filter是算法类的一种，它负责对数据对象进行处理，代表各种各样的算法。Filter有一个输入，一个输出，为了概念上简单，MITK中不支持多输入、多输出的Filter，而是通过Filter的具体子类提供的辅助函数来实现。在 2.2.3 小节中将具体讲述MITK中的算法模型。

Target：Target 是算法类的一种，顾名思义，它代表整个流水线数据的终点。Target 只有输入，没有输出，它的作用是将最后的数据放在一个合适的位置，终止流水线的执行。比如将得到的结果数据保存至磁盘，或者将得到的结果在屏幕上显示出来。

有了这些高层的概念以后，就可以很容易地将这些组件联系在一块，组成一个实用的系统。如 图 2-1 所示，整个流水线从一个Source开始，经过中间n个Filter的处理，最终终结于Target。这种抽象关系足以描述医学影像处理与分析这类以数据处理为中心的应用程序的需求，并且在概念上非常简洁、清晰，可以提供一个统一的模型来设计MITK的整个计算框架。

### 2.2.2 数据模型

数据表达是数据流模型中的一个核心的内容，数据对象起着连接算法流水线的重要作用。考虑到MITK的设计目标之一是有限目标，只关注于医学影像这个特定的领域，此处就简化了数据模型的建立。通过对医学影像数据进行仔细的分析以后，MITK中的数据对象（Data）可以被具体化为Volume和Mesh两种不同的具体数据对象，Data的类继承关系如 图 2-2 所示：

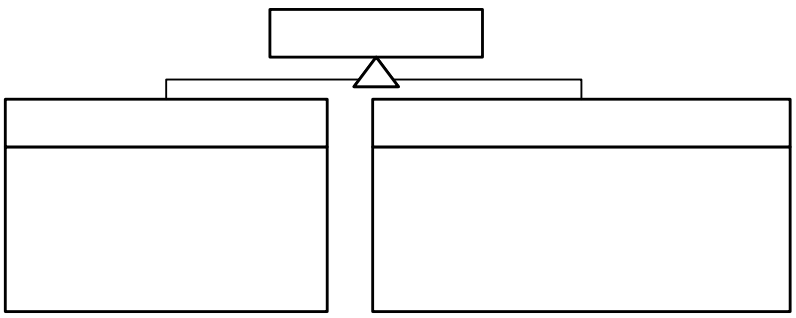


图 2-2 MITK 的数据模型

在图中，Volume 用来表达一个医学影像数据集，提供一个多维（包括一、

二、三维）、多模态（CT、MRI 等）的规则数据场的抽象。Volume 提供了丰富的接口来给算法对象使用，图中给出了一部分最常用的函数接口，这些接口可以使得其它的对象可以很方便地访问其内部的实际数据。Volume 是 MITK 中的核心对象之一。

Mesh用来表达一个几何数据，尤其是以三角面片网格形式表达的几何模型。Mesh内部的结构基于半边数据结构[6]，它提供了对一维线段、二维平面图形、三维三角网格以及通用多边形的支持，提供了丰富的接口来给相关的算法对象使用，上图中给出了一部分最常用的函数接口。不同于Volume，Mesh并不对应于医学影像设备所直接采集得到的数据，而是对医学影像数据集处理后得到的结果所依赖的表达形式，它是可视化算法中面绘制和某些分割算法所处理的对象，是MITK中的核心对象之一。

### 2.2.3 算法模型

有了一个设计良好的数据模型，下一步就是算法模型的抽象了。因为Source和Target只是特殊目的的算法对象，真正的算法是由Filter对象来代表的。按照一个Filter的输入和输出数据对象的类型，它可以被具体化为四种不同的算法种类，Filter的类继承关系如 图 2-3 所示：

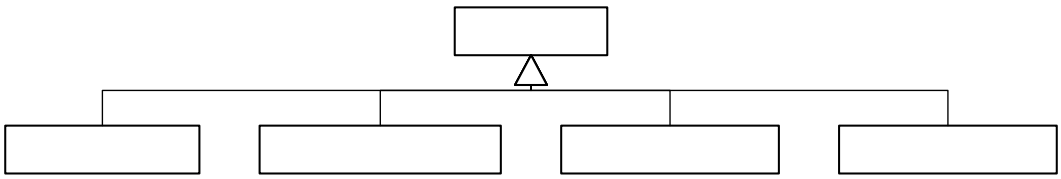


图 2-3 MITK 的算法模型

从图中可以看出 Filter 被具体化为四种不同的种类：VolumeToVolumeFilter 指的是输入数据和输出数据都为 Volume 数据对象的算法，在其下可以更细分出图像处理算法、图像配准算法、一部分图像分割算法等，在第六、七章中实现的算法就全部是从 VolumeToVolumeFilter 继承下来的；VolumeToMeshFilter 指的是输入数据为 Volume 数据对象，而输出数据为 Mesh 数据对象的算法，包括三维可视化中的面重建算法、一部分图像分割算法，在第三章中实现的算法就是

从 VolumeToMeshFilter 继承下来的；MeshToMeshFilter 指的是输入数据为 Mesh 数据对象，而输出数据为 Mesh 数据对象的算法，包括三维可视化中的面片网格化简、平滑、细分等数字几何处理算法；MeshToVolumeFilter 指的是输入数据为 Mesh 数据对象，而输出数据为 Volume 数据对象的算法，包括三维可视化中的基于距离场的算法、隐式曲面算法等。这四种 Filter 也只是一种高层的抽象，其中每一个种类下面都包含很多具体的算法，但是这些具体的算法的实现将只能在高层的模型规定的框架下进行。通过这样的算法模型，MITK 不仅为可视化、分割、配准等算法提供了一个框架，而且可以很方便地往里面扩充新的算法，而不需要改变上层的接口。

除了 Filter 所规定的四种不同的算法大类以外，在 MITK 里面还有一种特殊的算法，就是体绘制（Volume Rendering）算法，它的输入是一个 Volume，通过算法直接将整个 Volume 绘制到最终的图像中，在屏幕的窗口（属于一个 Target）里面显示。在第四章中将详细介绍 MITK 中的体绘制算法的框架。

## 2.3 MITK 的基础设施搭建

遵循面向对象设计的准则，整个 MITK 的类层次结构从 mitkObject 开始开枝散叶，另外，MITK 框架里面的类的命名遵循一定的原则，每个类都以小写的 mitk 为前缀，后面再跟上自己这个类的名字，每个单层的第一个字母大写，例如 mitkVolumeToVolumeFilter。在下面正文里面的描述中，为了方便起见，在不引起混淆的情况下，通常把类名字前面的 mitk 前缀省掉。

### 2.3.1 Object 提供的服务

为了给整个 MITK 框架中的类提供基础服务，在 Object 里面实现了五大基础设施，分别是 RTTI（Run-Time Type Information，运行时信息）、调试信息、对象引用计数（Reference Counting）、智能指针（Smart Pointer）、观察者（Observer），下面分别给予介绍。

(1) RTTI（运行时信息）

为了提供更高的效率，MITK中没有使用ANSI C++语言所提供的RTTI特性，而是自己实现了一套RTTI的机制，提供对象在运行时查询自己的类型信息。Object里面与RTTI有关的函数如 图 2-4 所示，其中GetClassname函数以字符串的

形式返回当前类的名字，比如mitkObject类返回的就是“mitkObject”；IsTypeOf和IsA函数判断当前的类是否是与typeName同种类型的类或者是其子类，两者的差异在于IsTypeOf是静态函数，可以在没有对象生成的时候调用，而IsA是个成员函数，只能在对象存在的时候被调用；SafeDownCast将object安全地转换为自身的类型。

为了实现这些基础服务，Object 要求它的每个子类都必须实现 GetClassname 和 IsA 这两个虚函数，以及 IsTypeOf 和 SafeDownCast 这两个静态函数。为了减少子类的工作量，MITK 里面用 C++的宏作为代码生成器来自动为子类生成这些函数，这个宏的代码如下：

```
#define MITK_TYPE(thisClass,superclass) \
  virtual const char *GetClassname() const {return #thisClass;} \
  static int IsTypeOf(const char *type) \
  { \
    if ( !strcmp(#thisClass,type) ) \
      { \
      return 1; \
      } \
    return superclass::IsTypeOf(type); \
  } \
  virtual int IsA(const char *type) \
  { \
    return this->thisClass::IsTypeOf(type); \
  } \
  static thisClass* SafeDownCast(mitkObject *o) \
  { \
    if ( o && o->IsA(#thisClass) ) \
      { \
      return static_cast<thisClass *>(o); \
      } \
    return NULL;\
  }
```

在这个宏中，thisClass 是类自身的类名，而 superclass 是父类的类名。有了这个宏，在 Object 的子类里面就只需写下这个宏的名字就行了，省去很多烦琐的工作。比如 DataObject 是 Object 的一个子类，那么在 DataObject 的声明里面

就只用写下下面的代码即可：

```
MITK_TYPE(mitkDataObject,mitkObject)
```

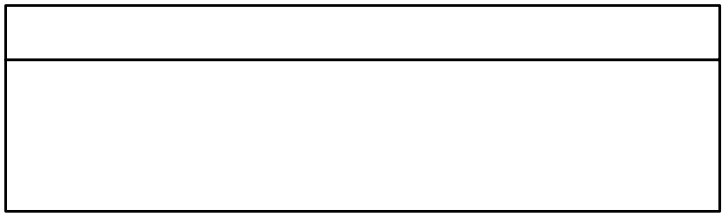


图 2-4 Object 提供的 RTTI 服务

(2) 调试信息

Object里面还为算法提供了调试信息的支持，与调试信息有关的函数如 图2-5 所示，其中SetDebug函数用来设置是否打开调试信息支持；而GetDebug函数用来获得当前调试信息支持是否打开；DebugOn函数用来打开调试信息支持；而DebugOff函数用来关闭调试信息支持；Print函数用来将对象内部的状态输送到os指定的设备上显示，其内部要调用虚函数PrintSelf，将实际调试信息的输出工作委托给PrintSelf函数来实现，所以Object的每个子类必须实现虚函数PrintSelf。

Object
+GetClassname() : char*
+IsTypeOf(in const char *typ
+IsA(in const char *typeNam
+SafeDownCast(in object : C

除了提供这些接口函数以外，MITK 还定义了一些宏来输出调试信息或者出错信息，这些宏的定义如下：

```
#define mitkGenericMessage(x) \
{     char *mitkmsgbuff; \
      ostrstream mitkmsg; \
      mitkmsg << x << "\n" << ends; \
      mitkmsgbuff = mitkmsg.str(); \
      mitkDisplayMessage(mitkmsgbuff);\
      mitkmsg.rdbuf()->freeze(0);}

#define mitkDebugMessage(x) \
{ if (this->m_Debug ) \
    { char *mitkmsgbuff; \
      ostrstream mitkmsg; \
```

```
      mitkmsg << "Debug: In " __FILE__ ", line " << __LINE__ << "\n" <<
this->GetClassname() << " (" << this << "): " << x  << "\n\n" << ends;
\
      mitkmsgbuff = mitkmsg.str(); \
      mitkDisplayMessage(mitkmsgbuff);\
      mitkmsg.rdbuf()->freeze(0);}}

#define mitkWarningMessage(x) \
{     char *mitkmsgbuff; \
      ostrstream mitkmsg; \
      mitkmsg << "Warning: In " __FILE__ ", line " << __LINE__ << "\n"
<< this->GetClassname() << " (" << this << "): " << x << "\n\n" << ends;
\
      mitkmsgbuff = mitkmsg.str(); \
      mitkDisplayMessage(mitkmsgbuff);\
      mitkmsg.rdbuf()->freeze(0);}

#define mitkErrorMessage(x) \
{     char *mitkmsgbuff; \
      ostrstream mitkmsg; \
      mitkmsg << "ERROR: In " __FILE__ ", line " << __LINE__ << "\n" <<
this->GetClassname() << " (" << this << "): " << x << "\n\n" << ends;
\
      mitkmsgbuff = mitkmsg.str(); \
      mitkDisplayMessage(mitkmsgbuff);\
      mitkmsg.rdbuf()->freeze(0); mitkObject::BreakOnError();}
```

其中 mitkGenericMessage 是用来输出通用的消息，其可以在任意位置被调用；而 mitkDebugMessage，mitkWarningMessage 和 mitkErrorMessage 都只能在 Object 及其子类的成员函数里面被调用，分别用来输出调试信息，警告信息和严重错误信息。它们和 Object 里面的相关函数一起，提供了 MITK 里面提供调试信息的基础服务。

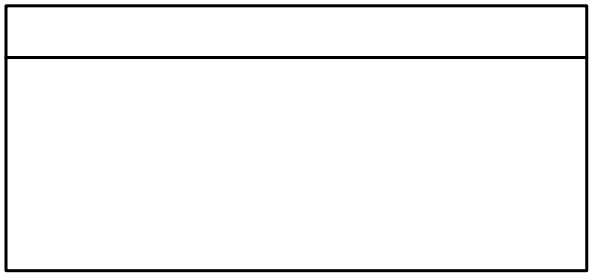


图 2-5 Object 提供的调试信息服务

(3) Reference Counting（引用计数）

引用计数是实现内存管理的一种手段，它的目的是将内存中不再被其它对象引用的对象给自动释放掉，从而达到节省内存并保证内存不泄漏。Object里面与引用计数有关的函数如 图 2-6 所示，其中AddReference函数将本身被其它对象引用的次数加一；RemoveReference函数将本身被其它对象引用的次数减一；GetReferenceCount函数返回本身被其它对象引用的次数；Delete函数将本身从内存中删除，但是前提是本身被其它对象引用的次数小于或者等于零。

为了实现引用计数，MITK 还使用了设计模式来限制所有从 Object 继承下来的子类都必须在堆（Heap）上来分配内存，而不能从栈（Stack）上来分配内存。为了达到这个目的，Object 和所有从 Object 继承下来的类的析构函数都必须是 Protected 的。在删除一个从 Object 继承下来的对象时，必须调用其 Delete 函数，而不能直接使用 delete 操作符。

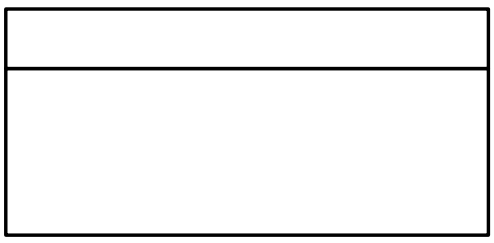


图 2-6 Object 提供的引用计数服务

(4) Smart Pointer（**智能指针**）

智能指针和引用计数合起来构成了MITK中内存管理的基础，所谓智能指针，就是其行为模仿普通的指针，但是可以通过运算符重载来进行一些必要的操作，从而可以减少不必要的代码。在MITK中提供了一个模板类SmartPointer用来实现智能指针，它的内部维护着一个成员变量realPtr，可以被具体的模板参数T来具体化，这里要求类型T必须是Object的子类，其提供的主要功能请参看 图 2-7。为了实现普通的指针行为，它重载了 -> 和 * 运算符，分别返回实际指针的地址和引用；为了实现其智能的行为，它重载了 = 运算符，用于在把一个Object的子类的指针赋值给智能指针时，来自动地进行引用计数的工作，其实现代码如下：

```
template<class T>
inline mitkSmartPointer<T>& mitkSmartPointer<T>::operator=(T* realPtr)
{
    if(realPtr == NULL)    return *this;
    if(realPtr == m_Pointee) return *this;
    if(m_Pointee) m_Pointee->RemoveReference();
    m_Pointee = realPtr;
    m_Pointee->AddReference();
    return *this;
}
```

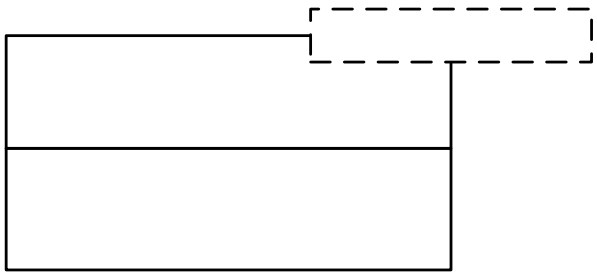


图 2-7 MITK 中的智能指针

(5) Observer（**观察者**）

MITK是一个底层的算法库，但是在基于MITK开发实际应用程序时，有时

必须知道内部算法的状态，比如算法执行的进度等信息。为了达到MITK底层算法库和上层的应用程序之间的通信，MITK里面实现了Observer这一设计模式，其在Object里面的函数如 图 2-8 所示，AddObserver将一个Observer插入到内部维护的一个队列中；RemoveObserver将一个指定的Observer从内部的队列中删除；RemoveAllObservers将内部的队列清空。而MITK中也单独定义了Observer这个类层次，所有从Observer继承的子类必须实现其虚函数Update，从而更新界面元素。一个Observer就是一个单独的对象，在外面观察一个MITK对象的内部状态，并且在必要的时候被更新。

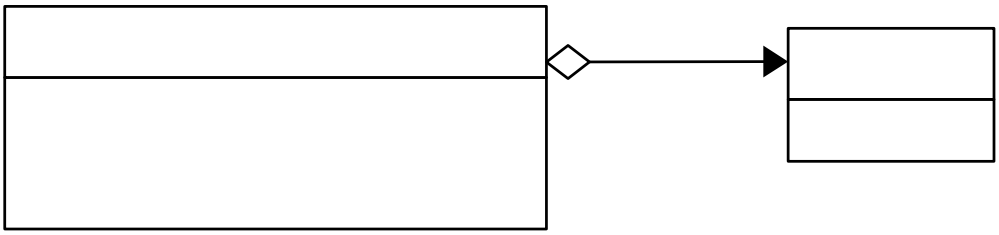


图 2-8 Object 中提供的 Observer 设计模式实现

### 2.3.2 内存管理

因为MITK的结构比较复杂，所以实现一个有效的内存管理方案是非常必要的。上面介绍的引用计数和智能指针的联合使用，形成了MITK中的第一层内存管理，用来保证不出现内存泄漏。另外，MITK中实现了一个简单的垃圾回收（Garbage Collection）机制，用于在程序退出之前将所有可能泄漏的内存释放掉，此功能是在GarbageCollection类里面实现的，它的接口函数如 图 2-9 所示，AddObject函数将一个MITK的对象的指针加入到内部维护的一张列表中，而RemoveObject函数将指定的对象从内部的列表中删除。由于GarbageCollection在整个系统中只能存在一份实例，所以MITK中使用了设计模式中的Singleton来达到这一目的。

有了 GarbageCollection 类提供的基本功能以后，Object 里面也必须提供相应的配合才能实现垃圾回收机制。在 Object 的构造函数里面，通过 GetGarbageCollector 函数来得到系统里面唯一的 GarbageCollection 实例，并且将自身加入到 GarbageCollection 的列表里面，代码如下：

Object(Observer)
+AddObserver(in observer : Observer)
+RemoveObserver(in observer : Observe
+RemoveAllObservers()

```
GetGarbageCollector()->AddObject(this);
```

在 Object 的析构函数里面，也同样通过 GetGarbageCollector 函数来得到系统里面唯一的 GarbageCollection 实例，并且通过 RemoveObject 将自身从 GarbageCollection 的列表里面删除，代码如下：

```
GetGarbageCollector()->RemoveObject(this);
```

在整个系统退出的时候，系统里面唯一的 GarbageCollection 实例的析构函数被调用，检查自己维护的列表中是否为空，如果不为空，则说明有内存泄漏，同时考虑到引用计数这一服务，按照一定的规则将泄漏的内存清空，代码如下所示：

```
if(m_ObjectCollection->Count() <= 0)
{
    m_ObjectCollection->Delete();
    return;
}

mitkObject *aObject;

while(m_ObjectCollection->Count() > 0)
{
    for(m_ObjectCollection->InitTraversal(); aObject =
    m_ObjectCollection->GetNextItem(); )
    {
        if(aObject->GetReferenceCount() <= 0)
        {
            aObject->Delete();
            break;
        }
    }
}
```

```
m_ObjectCollection->Clear();
m_ObjectCollection->Delete();
```

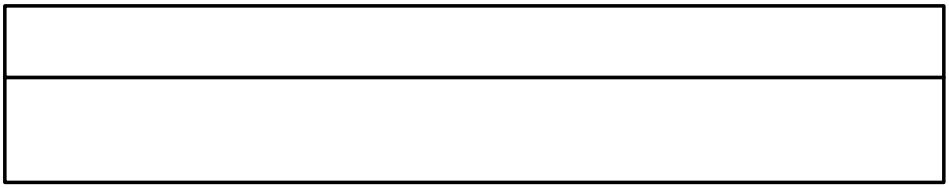



图 2-9 MITK 中的垃圾回收接口

### 2.3.3 跨平台的实现

MITK 的所有代码均使用符合 ANSI C++标准的特性，这保证了代码的可移植性。但是对一些操作系统相关的代码，比如窗口的事件处理、虚拟内存管理等，必须对每一个操作系统写一套代码。为了能够透明地封装这些操作系统相关的代码，得到一个优雅的解决方案，MITK 中再次使用了一种设计模式，名字为桥（Bridge），下面以 View 这一对象为例进行介绍。

在MITK中，View这一对象是唯一跟图形用户界面相关的部分，给用户提供了一个用来显示图像或者三维图形的窗口，这也就必然导致它跟具体的操作系统相关。为了使操作系统相关的代码相对独立，并且添加对新的操作系统的支持的时候也不影响到客户端的代码，View这一部分的框架结构图如（图 2-10）所示。

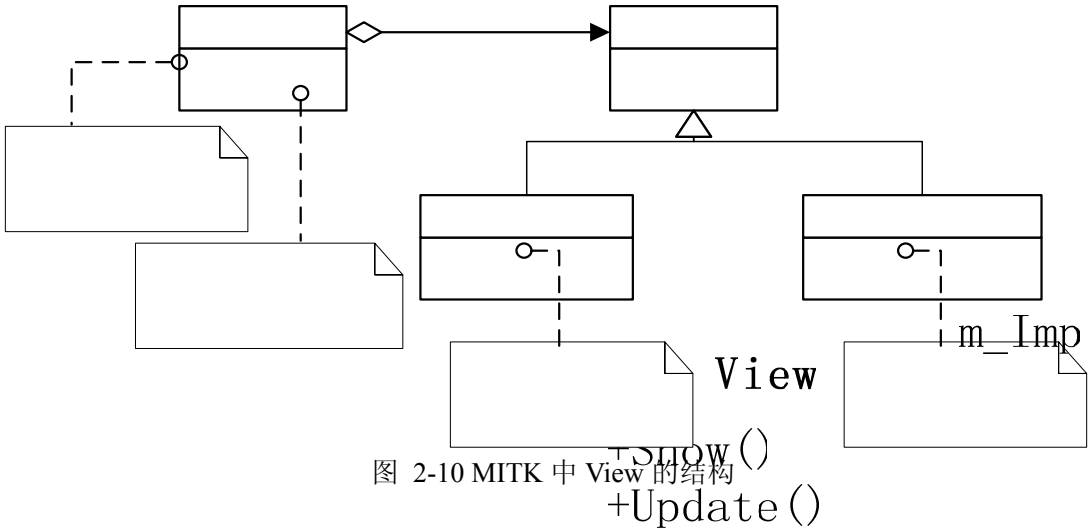



图 2-10 MITK 中 View 的结构

从图中可以看出，View 里面维护着 Implementor 基类的指针，并且 View 把所有与操作系统相关的代码都委托给 Implementor 来实现，Implementor 通过其子类来实现具体的代码。通过这种结构，客户端只知道有 View，而不知道有 Implementor 的存在，因此降低了藕合；并且要添加对一个新的操作系统的支持的时候，只需要在 Implementor 分支下面再增加对应的子类即可，无需更改客户端的代码。

### 2.3.4 SSE 加速的实现

Intel 在其奔腾 3 以后的 CPU 中增加了 SSE 扩展指令集，用以提高三维程序以及多媒体程序的性能。因为 MITK 中三维可视化是一个比较重要的部分，对性能要求也比较高，因此内部提供了对 SSE 的支持，并且已经实现了一部分算法的 SSE 优化加速。实现 SSE 指令集的支持有一些问题必须考虑，因为传统上是使用汇编语言对 SSE 指令集进行编程，这显然将会违背 MITK 的可移植性这一设计目标；另外，并不是所有的 CPU 都支持 SSE 指令集，因此必须有一种手段能在运行时检测当前 CPU 信息，并动态决定使用 SSE 加速版本还是使用普通版本。在 MITK 中，对第一个问题的解决方案是使用编译器提供的 Intrinsics 去对 SSE 指令集进行编程，这样虽然没有使用汇编语言的效率高，但是目前的主流编译器（包括 Visual C++ 6.0 + Processor Pack，Visual C++ 7.0，gcc，Intel Compiler 等）都提供了对 Intrinsics 的支持，因此可移植性得到了保证；对第二个问题的解决方案是提供了两套分离的动态链接库，一套是包含 SSE 加速算法

的，一套是非 SSE 加速算法的，由客户端在使用时进行判断并动态加载。

## 2.4 小结

本章的第一部分介绍了 MITK 的设计目标，其目的是为了给医学影像领域的研究者提供一套具有一致接口的，可复用的，结合了 VTK 和 ITK 中可视化、分割、配准等功能的开发工具包。本章的第二部分介绍了 MITK 的整体计算框架，其中包括基于数据流模型的整体框架，数据模型和算法模型。第三部分介绍的是 MITK 的基础设施搭建，包括基类 Object 所提供的服务以及相关信息，内存管理，跨平台实现和 SSE 加速的实现。

目前 MITK 的整体框架已经基本稳定，在保持 MITK 的整体框架的稳定性的基础上，下一步可以不断地完善和添加 MITK 的各种算法，使其越来越丰富，越来越实用。接下来的几章将会介绍具体的算法在 MITK 中的实现。

参考文献

1. J. Rumbaugh, M. Blaha, W. Premerlani, et al. Object-Oriented Modeling and Design. Prentice-Hall, 1991.
2. Meyer, B. Object-Oriented Software Construction. Prentice Hall, 1997.
3. Erich Gamma, Richard Helm, Ralph Johnson, et al. Design Patterns, Elements of Reusable Object-Oriented Software. Pearson Education Publisher, 1994.
4. K. Engel, M. Kraus, and T. Ertl, High-quality pre-integrated volume rendering using hardware accelerated pixel shading, in Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware 2001.
5. C. Resk-Salama, K. Engel, M. Bauer, G. Greiner, T. Ertl， Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization，in Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware 2000.
6. S. Campagna, L. Kobbelt, H. P. Seidel. Directed Edges - A Scalable Representation For Triangle Meshes. ACM Journal of Graphics Tools 3 (4), 1998, pp. 1-12

# 3 面绘制（Surface Rendering）的框架与实现

面绘制是医学图像三维可视化的重要手段之一，它通过对一系列的二维图像进行边界识别等分割处理，重新还原出被检物体的三维模型，并以表面的方式显示出来，从而为用户提供具有较强真实感的三维医学图像，便于医生从多角度、多层次进行观察和分析，并且能够使医生有效地参与数据的处理分析过程，在辅助医生诊断、手术仿真、引导治疗等方面都可以发挥重要的作用。

本章主要介绍 MITK 中的面绘制框架的设计和实现，包括两大部分：表面重建和表面绘制，在下面两节中将分别予以详细介绍。

## 3.1 表面重建算法及其在 MITK 中的实现

在 MITK 中，表面重建算法被抽象成一个 VolumeToMeshFilter，如图 3-1 所示。

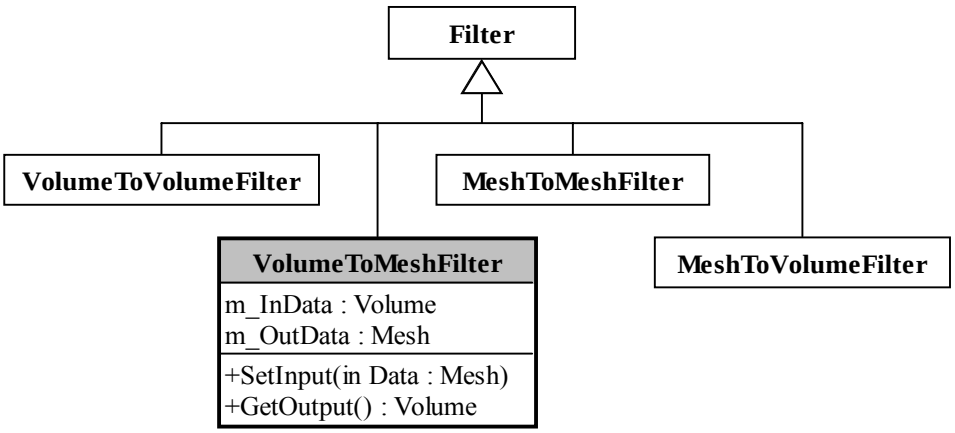



图 3-2 VolumeToMeshFilter

其输入数据是N张两维的切片生数据（如 图 3-3 所示），表示为一个mitkVolume；经处理后的输出数据是一个以三角网格来表示的三维表面模型（如图 3-4 所示），表示为mitkMesh。

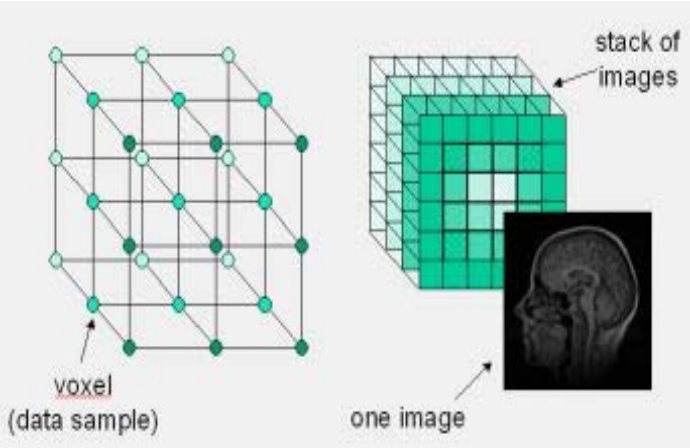



图 3-3 体数据示意图

图 3-4 以三角网格表示的表面模型

mitkMesh 中的三角面片数据以点表和面表来组织。点表是一个 float 型数组，每 6 个数保存一个顶点的信息，包括顶点坐标和法向量，按(***x, y, z; nx, ny, nz***)存放；面表是一个 unsigned int 型数组，每 3 个数保存一个三角片的信息，分别是组成该三角片的 3 个顶点在点表中的索引。

MITK 中的表面重建主要采用了目前得到广泛应用的 Marching Cubes 算法，下面几节就对该算法的原理和实现做一个介绍。

### 3.1.1 传统的 Marching Cubes 算法

Marching Cubes 算法是 W. Lorensen 等人于 1987 年提出来的一种三维重建方法[1]，因为它的原理简单，并且很容易实现，因此得到了广泛的应用，并且此算法在美国已经申请专利，它被认为是至今为止最流行的面显示算法之一。

Marching Cubes 算法是面显示算法中的一种，因为它的本质是从一个三维的

数据场中抽取出一个等值面，所以也被称为“等值面提取”（Isosurface Extraction）算法。

对于一个标准的医学图像的体数据集，它往往是由一系列的二维切片数据组成的，而每张切片都有空间上的分辨率。假设有一个体数据集，包含 58 张切片，每张切片的分辨率是 512×512，那么它可以被认为是一个连续函数 f(x,y,z) 在 x、y、z 三个方向上按一定的间隔分别采样了 512、512、58 次所得到的。而所谓的等值面，实际上是指空间中的一张曲面，在该曲面上函数 f(x,y,z)的值等于某一给定值。等值面提取算法的核心就是要从给定的采样点中找出等值面来，这时最容易想到的方法就是首先由采样点恢复出连续函数 f(x,y,z) 来，然后由 f(x,y,z) 和某一给定的值（通常叫域值）来得出等值面，这种方法一般被称为显式的等值面提取算法，它的计算复杂度比较高，并且由于重构和重采样所带来的误差比较大，所以精度也得不到保证。

与此相反，Marching Cubes 算法采用了隐式的等值面提取方法，它不直接计算 f(x,y,z) ，而是直接从体数据中获取等值面的信息。 算法需要用户提供一个域值，也就是所希望提取出来的物质的密度值，比如要提取出骨骼，域值就要相对大一些，然后根据体数据的信息，就可以提取出来等值面的三角网格表达。

Marching Cubes 算法的过程可以描述如下：

1．每次读出两张切片，形成一层（Layer）；

2．两张切片上下相对应的四个点构成一个立方体（Cube），也叫 Cell , Voxel 等，如 图 3-5 所示；

3．从左至右，从前到后顺序处理一层中的 Cubes(抽取每个 Cube 中的等值面)，然后从下到上顺序处理（n－1）层，算法就结束，故名为 Marching Cubes。

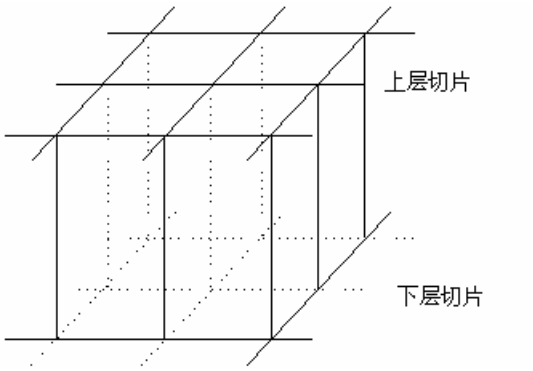



图 3-5 Cubes 示意图

对于每一个 Cube 而言，它的八个顶点的灰度值可以直接从输入数据中得到，要抽取的等值面的域值也已经知道。如果一个顶点的灰度值大于域值，则将它标记为黑色（Marked Vertex），而小于的不标（Unmarked Vertex），如**错误！未找到引用源。**所示。

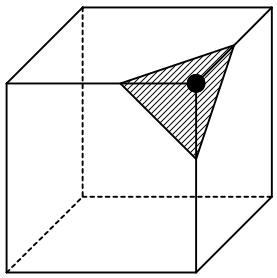


图 3-6 Marked Vertex 与 Unmarked Vertex

因为一个Cube有 8 个顶点，每个顶点有Marked、Unmarked两种状态，所以等值面的分布总共可能有 28＝256 种。但是考虑到Cube有旋转（Rotation）对称性，旋转不影响等值面的拓扑结构，另外，所有的Marked Vertex变为Unmarked Vertex，或者反过来(Invertion 对称)，等值面的连接方式也不会改变。考虑了Rotation和Invertion两种情况后，原作者总结了 15 种Basic Cube，它们覆盖了所有 256 种可能的情况，如 图 3-7 所示。

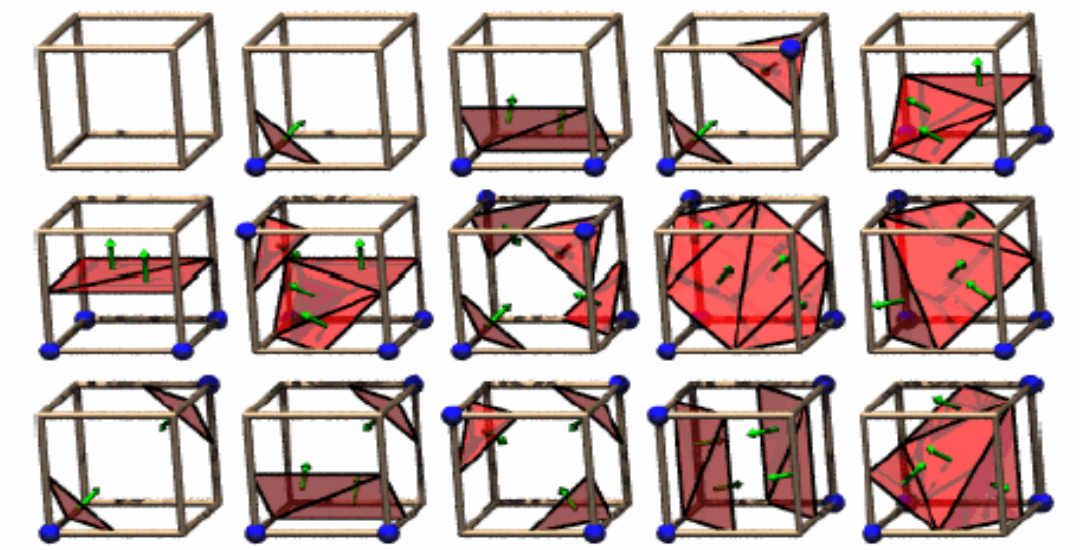


图 3-7 15 种基本 Cubes 的拓扑形状

根据这 15 种基本的 Cubes，可以造出一个查找表（Look-up Table）。表的长度为 256，记录了所有情况下的等值面连接方式。所以此时只需分别比较一个 Cube 的八个顶点与域值之间的大小关系，即可得出一个 0-255 之间的索引值（CubeIndex），然后直接查表就可得到此 Cube 在那条边上有等值点，并且还能得到等值点的连接方式等信息，这时候就可以将等值点连接起来以形成等值面。

要想用真实感图形学技术将等值面显示出来，除了要知道每个等值点的坐标外，还必须知道每个等值点的法向量。在计算 Cube 某条边上的等值点坐标与法向量时，有两种方法，一种是线性插值，另外一种是中点选择（MidPoint Selection）。

线性插值的公式如下所示：

$$\mathbf{P} = \mathbf{P_1} + (\text{isovalue} - V_1)(\mathbf{P_2} - \mathbf{P_1}) / (V_2 - V_1) \qquad (3\text{-}1)$$

$$\mathbf{N} = \mathbf{N_1} + (\text{isovalue} - V_1)(\mathbf{N_2} - \mathbf{N_1}) / (V_2 - V_1) \qquad (3\text{-}2)$$

其中 **P** 代表等值点坐标，$\mathbf{P_1}$、$\mathbf{P_2}$ 代表两个端点的坐标，$V_1$、$V_2$ 代表两个端点的灰度值， isovalue 代表阈值；N 代表等值点法向量，$\mathbf{N_1}$、$\mathbf{N_2}$ 代表两个端点的法向量。

中点选择的公式如下所示：

$$\mathbf{P} = (\mathbf{P_1} + \mathbf{P_2}) / 2 \quad (3\text{-}3)$$

$$\mathbf{N} = (\mathbf{N_1} + \mathbf{N_2}) / 2 \quad (3\text{-}4)$$

其中 **P** 和 **N**、$\mathbf{P_1}$、$\mathbf{P_2}$、$\mathbf{N_1}$、$\mathbf{N_2}$ 所代表的意义同上。

中点选择所具有的优点是：

1. 引起的误差低于 1/2 Cube 边长，这在医学图像的解析度越来越高的情况下，所重建出来的图像与线性插值得到的图像并没有明显的视觉上的差异
2. 如果先放大 10 倍再进行运算，就可以完全采用整数运算，避免浮点运算
3. 可以使得局部表面更平坦，有利于后续的化简过程，如 图 3-8 所示：

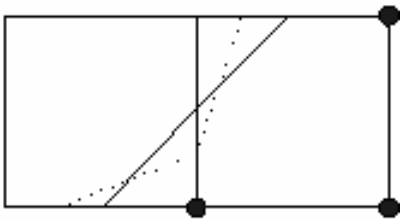


图 3-8 局部表面的平坦化

MITK 中的 mitkMarchingCubes 类实现的表面重建算法基于传统的 Marching Cubes 算法，它可以接受高、低两个阈值，把灰度在这两个阈值之间的组织分界面提取出来，从而达到物体表面重建的目的。

该算法使用了如下所示的查找表：

```
short g_EdgeTable[256] =
{
    0x0  , 0x109, 0x203, 0x30a, 0x406, 0x50f, 0x605, 0x70c,
    0x80c, 0x905, 0xa0f, 0xb06, 0xc0a, 0xd03, 0xe09, 0xf00,
    0x190, 0x99 , 0x393, 0x29a, 0x596, 0x49f, 0x795, 0x69c,
    0x99c, 0x895, 0xb9f, 0xa96, 0xd9a, 0xc93, 0xf99, 0xe90,
    0x230, 0x339, 0x33 , 0x13a, 0x636, 0x73f, 0x435, 0x53c,
    0xa3c, 0xb35, 0x83f, 0x936, 0xe3a, 0xf33, 0xc39, 0xd30,
    0x3a0, 0x2a9, 0x1a3, 0xaa , 0x7a6, 0x6af, 0x5a5, 0x4ac,
    0xbac, 0xaa5, 0x9af, 0x8a6, 0xfaa, 0xea3, 0xda9, 0xca0,
```

```
    0x460, 0x569, 0x663, 0x76a, 0x66 , 0x16f, 0x265, 0x36c,
    0xc6c, 0xd65, 0xe6f, 0xf66, 0x86a, 0x963, 0xa69, 0xb60,
    0x5f0, 0x4f9, 0x7f3, 0x6fa, 0x1f6, 0xff , 0x3f5, 0x2fc,
    0xdfc, 0xcf5, 0xfff, 0xef6, 0x9fa, 0x8f3, 0xbf9, 0xaf0,
    0x650, 0x759, 0x453, 0x55a, 0x256, 0x35f, 0x55 , 0x15c,
    0xe5c, 0xf55, 0xc5f, 0xd56, 0xa5a, 0xb53, 0x859, 0x950,
    0x7c0, 0x6c9, 0x5c3, 0x4ca, 0x3c6, 0x2cf, 0x1c5, 0xcc ,
    0xfcc, 0xec5, 0xdcf, 0xcc6, 0xbca, 0xac3, 0x9c9, 0x8c0,
    0x8c0, 0x9c9, 0xac3, 0xbca, 0xcc6, 0xdcf, 0xec5, 0xfcc,
    0xcc , 0x1c5, 0x2cf, 0x3c6, 0x4ca, 0x5c3, 0x6c9, 0x7c0,
    0x950, 0x859, 0xb53, 0xa5a, 0xd56, 0xc5f, 0xf55, 0xe5c,
    0x15c, 0x55 , 0x35f, 0x256, 0x55a, 0x453, 0x759, 0x650,
    0xaf0, 0xbf9, 0x8f3, 0x9fa, 0xef6, 0xfff, 0xcf5, 0xdfc,
    0x2fc, 0x3f5, 0xff , 0x1f6, 0x6fa, 0x7f3, 0x4f9, 0x5f0,
    0xb60, 0xa69, 0x963, 0x86a, 0xf66, 0xe6f, 0xd65, 0xc6c,
    0x36c, 0x265, 0x16f, 0x66 , 0x76a, 0x663, 0x569, 0x460,
    0xca0, 0xda9, 0xea3, 0xfaa, 0x8a6, 0x9af, 0xaa5, 0xbac,
    0x4ac, 0x5a5, 0x6af, 0x7a6, 0xaa , 0x1a3, 0x2a9, 0x3a0,
    0xd30, 0xc39, 0xf33, 0xe3a, 0x936, 0x83f, 0xb35, 0xa3c,
    0x53c, 0x435, 0x73f, 0x636, 0x13a, 0x33 , 0x339, 0x230,
    0xe90, 0xf99, 0xc93, 0xd9a, 0xa96, 0xb9f, 0x895, 0x99c,
    0x69c, 0x795, 0x49f, 0x596, 0x29a, 0x393, 0x99 , 0x190,
    0xf00, 0xe09, 0xd03, 0xc0a, 0xb06, 0xa0f, 0x905, 0x80c,
    0x70c, 0x605, 0x50f, 0x406, 0x30a, 0x203, 0x109, 0x0
};

char g_TriTable[256][16] =
{
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    { 0,  8,  3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    { 0,  1,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    { 1,  8,  3,  9,  8,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    { 1,  2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    { 0,  8,  3,  1,  2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    { 9,  2, 10,  0,  2,  9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    { 2,  8,  3,  2, 10,  8, 10,  9,  8, -1, -1, -1, -1, -1, -1, -1},
    { 3, 11,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    { 0, 11,  2,  8, 11,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},

    ......
```

```
    { 1, 10,  2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    { 1,  3,  8,  9,  1,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    { 0,  9,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    { 0,  3,  8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
};
```

其中，g_TriTable比较大，篇幅所限只能列出部分数据。与这两个查找表对应的Cube顶点和边的位置关系如 图 3-9 所示，按该图，可以很容易的生成这两个表。

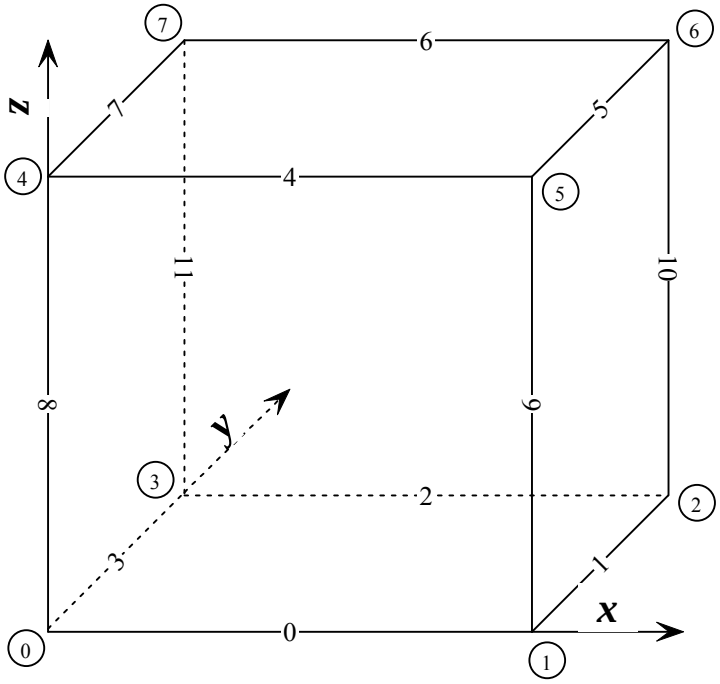



图 3-9 与查找表对应的 Cube 各顶点和边的位置关系及索引编号

首先，每个顶点以一位二进位表示，将 0~7 八个顶点从低到高排成一个 8 bits 的整数，若第 i 个顶点灰度在低阈值和高阈值之间，则第 i 位置 1，否则置 0，这样就形成了一个索引值，范围从 0 到 255。

然后，根据这 256 种不同情况写出上面两表中的对应项。对于g_EdgeTable来说，其每一项是一个 12 bits的整数，从第 0~11 bit分别代表 图 3-9 中的第 0~11 条边，根据索引值所对应的顶点灰度情况判断，若第i条边与等值面存在交点（一个端点灰度值在高、低阈值之间，另一个端点灰度值在高、低阈值之外），则第

i位置 1，否则置 0。对于g_TriTable，其每一项是一个由 16 个 8 bits整数组成的数组，记录了与索引值对应的三角面片分割情况。比如，对于索引 9（$00001001_2$），顶点 0 和 3 的灰度值在低阈值和高阈值之间，则第 0、2、8 和 11 条边与等值面有交点，g_EdgeTable[9]的值就为 $100100000101_2=905_{16}$，而g_TriTable[9]则以顶点所在边的编号记录了这 4 个交点所构成的两个三角片：0-11-2 和 8-11-0，如 图 3-10 所示。注意，生成的三角片正向须保持一致。

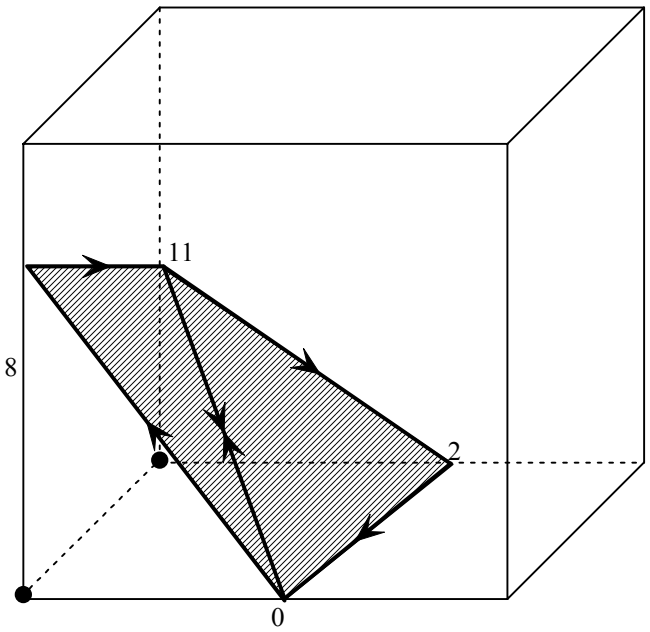



图 3-10 索引值为 9 时对应表项示意

算法在计算等值点坐标和法向量时使用线性插值的方法。算法的具体实现如下：

```
template <class VT>
int t_ExecuteMarchingCubes(mitkVolume *input, mitkMesh *output,
                           mitkMarchingCubes *self, VT *vData)
{
    // 记录生成的顶点数和面数
    index_type vNumber = 0;
    index_type fNumber = 0;

    // 当前 Cube 中生成的顶点和面数
    int vertPos, facePos;
```

```
// 包围盒的尺寸
float min[3];
float max[3];
min[0] = min[1] = min[2] =  30000;
max[0] = max[1] = max[2] = -30000;

// 当前扫描层的切片数据
// 因为要计算法向量，前后共设计 4 层切片数据
VT *pSliceA;
VT *pSliceB;
VT *pSliceC;
VT *pSliceD;
VT *tempSlice;

// 得到所需的有关 Volume 的信息
int imageWidth = input->GetWidth();
int imageHeight = input->GetHeight();
int imageSize = imageWidth * imageHeight;
int sliceNumber = input->GetImageNum();

//two adjacent pixels's distance from 3 axes
float XSpace = input->GetSpacingX();
float YSpace = input->GetSpacingY();
float ZSpace = input->GetSpacingZ();

// 得到两个灰度阈值
float lowThreshold = self->GetLowThreshold();
float highThreshold = self->GetHighThreshold();

if (XSpace == 0)
{
    mitkGenericMessage("Marching Cubes : Sorry,your data are wrong!
                        XSpace = " << XSpace);
    return 0;
}
if (YSpace == 0)
{
    mitkGenericMessage("Marching Cubes : Sorry,your data are wrong!
                        YSpace = " << YSpace);
    return 0;
```

```
}
if (ZSpace == 0)
{
    mitkGenericMessage("Marching Cubes : Sorry,your data are wrong!
                        ZSpace = " << ZSpace);
    return 0;
}

pSliceA = NULL;
pSliceB = NULL;
pSliceC = NULL;
pSliceD = NULL;

// 用于记录上层扫描生成的顶点，避免生成重复的顶点
index_type *bottomXEdge = new index_type[imageSize];
index_type *bottomYEdge = new index_type[imageSize];
index_type *topXEdge = new index_type[imageSize];
index_type *topYEdge = new index_type[imageSize];
index_type *zEdge = new index_type[imageSize];


tempSlice = new VT[imageSize];

if ( bottomXEdge == NULL || bottomYEdge == NULL
  || topXEdge == NULL || topYEdge == NULL
  || zEdge == NULL || tempSlice == NULL)
{
    mitkGenericMessage("Marching Cubes : sorry,the assistant data
                        structure's memory are not right allocated");
    return 0;
}

//initialize the assistant data structure
memset(bottomXEdge, -1, sizeof(index_type) * imageSize);
memset(bottomYEdge, -1, sizeof(index_type) * imageSize);
memset(topXEdge, -1, sizeof(index_type) * imageSize);
memset(topYEdge, -1, sizeof(index_type) * imageSize);
memset(zEdge, -1, sizeof(index_type) * imageSize);

memset(tempSlice, 0, sizeof(VT) * imageSize);
```

```
    // 计算某一层顶点和三角片时所需的一些变量
    // 一些循环变量
    short i, j, k, w, r;

    // Cube 类型
    unsigned char cubeType(0);

    // 用于计算法向量
    float dx[8];
    float dy[8];
    float dz[8];
    float squaroot;

    // 记录某个 Cube 生成的顶点坐标和相应顶点的法向量，
    // 对应一个 Cube 的 12 条边
//  float vertPoint[6];
    float vertPoint[12][6];
    index_type cellVerts[12];
    index_type triIndex[5][3];

    // 用于记录已生成顶点索引的临时变量
    index_type offset;

    // 当前 Cube 八个顶点的灰度值
    VT cubegrid[8];

    index_type *edgeGroup;

    //get memory data
    pSliceD = vData; // HERE
    pSliceB = tempSlice;
    pSliceA = tempSlice;

    //allocate memory for output data
    float *pVertex;
    unsigned int *pFace;

    output->SetVertexNumber(100000);
    output->SetFaceNumber(100000);
```

```
pVertex = output->GetVertexData();
pFace = output->GetFaceData();

// 扫描时 4 层切片的排列顺序
/*-------------------D
  -------------------B   |
  -------------------A   |
  -------------------C   |
                        V */
for (i=0; i<=sliceNumber ; ++i)
{
    pSliceC = pSliceA;
    pSliceA = pSliceB;
    pSliceB = pSliceD;

    if (i >= sliceNumber - 1)
    {
        pSliceD = tempSlice;
    }
    else
    {
        pSliceD += imageSize;
    }

    for (j=0; j<imageHeight - 1; ++j)
        for (k=0; k<imageWidth - 1; ++k)
        {
            //calculate the cube's eight point's grey value
            //得到当前立方体 8 顶点灰度
            cubegrid[0] = pSliceA[j * imageWidth + k];
            cubegrid[1] = pSliceA[j * imageWidth + k + 1];
            cubegrid[2] = pSliceA[(j + 1) * imageWidth + k + 1];
            cubegrid[3] = pSliceA[(j + 1) * imageWidth + k];
            cubegrid[4] = pSliceB[j * imageWidth + k];
            cubegrid[5] = pSliceB[j * imageWidth + k + 1];
            cubegrid[6] = pSliceB[(j + 1) * imageWidth + k + 1];
            cubegrid[7] = pSliceB[(j + 1) * imageWidth + k];

            // 计算 Cube 的类型
            cubeType = 0;
```

```
for (w=0; w<8; ++w)
{
    if ((cubegrid[w] > lowThreshold)
        && (cubegrid[w] < highThreshold))
    {
        cubeType |= (1 << w);
    }
}

if ((cubeType == 0) || (cubeType == 255))
{
    continue;
}

// initialize the cellVerts table
// and make it turn into zero table
for (w=0; w<12; w++)
{
    cellVerts[w] = -1;
}

// 计算 6 个方向相邻点的像素差值（用于计算法向量）
if (k == 0)
{
    dx[0] = pSliceA[j * imageWidth + 1];
    dx[3] = pSliceA[(j + 1) * imageWidth + 1];
    dx[4] = pSliceB[j * imageWidth + 1];
    dx[7] = pSliceB[(j + 1) * imageWidth + 1];
}
else
{
    dx[0] = pSliceA[j * imageWidth + k + 1]
          - pSliceA[j * imageWidth + k - 1];
    dx[3] = pSliceA[(j + 1) * imageWidth + k + 1]
          - pSliceA[(j + 1) * imageWidth + k - 1];
    dx[4] = pSliceB[j * imageWidth + k + 1]
          - pSliceB[j * imageWidth + k - 1];
    dx[7] = pSliceB[(j + 1) * imageWidth + k + 1]
          - pSliceB[(j + 1) * imageWidth + k - 1];
}
```

```
if (k == imageWidth - 2)
{
    dx[1] = - pSliceA[j * imageWidth + imageWidth - 2];
    dx[2] = - pSliceA[(j+1) * imageWidth + imageWidth - 2];
    dx[5] = - pSliceB[j * imageWidth + imageWidth - 2];
    dx[6] = - pSliceB[(j+1) * imageWidth + imageWidth - 2];
}
else
{
    dx[1] = pSliceA[j * imageWidth + k + 2]
          - pSliceA[j * imageWidth + k];
    dx[2] = pSliceA[(j + 1) * imageWidth + k + 2]
          - pSliceA[(j + 1) * imageWidth + k];
    dx[5] = pSliceB[j * imageWidth + k + 2]
          - pSliceB[j * imageWidth + k];
    dx[6] = pSliceB[(j + 1) * imageWidth + k + 2]
          - pSliceB[(j + 1) * imageWidth + k];
}

if (j == 0)
{
    dy[0] = pSliceA[imageWidth + k];
    dy[1] = pSliceA[imageWidth + k + 1];
    dy[4] = pSliceB[imageWidth + k];
    dy[5] = pSliceB[imageWidth + k + 1];
}
else
{
    dy[0] = pSliceA[(j + 1) * imageWidth + k]
          - pSliceA[(j - 1) * imageWidth + k];
    dy[1] = pSliceA[(j + 1) * imageWidth + k + 1]
          - pSliceA[(j - 1) * imageWidth + k + 1];
    dy[4] = pSliceB[(j + 1) * imageWidth + k]
          - pSliceB[(j - 1) * imageWidth + k];
    dy[5] = pSliceB[(j + 1) * imageWidth + k + 1]
          - pSliceB[(j - 1) * imageWidth + k + 1];
}

if (j == imageHeight - 2)
```

```
{
    dy[2] = - pSliceA[(imageHeight-2) * imageWidth + k+1];
    dy[3] = - pSliceA[(imageHeight-2) * imageWidth + k];
    dy[6] = - pSliceB[(imageHeight-2) * imageWidth + k+1];
    dy[7] = - pSliceB[(imageHeight-2) * imageWidth + k];
}
else
{
    dy[2] = pSliceA[(j + 2) * imageWidth + k + 1]
          - pSliceA[j * imageWidth + k + 1];
    dy[3] = pSliceA[(j + 2) * imageWidth + k]
          - pSliceA[j * imageWidth + k];
    dy[6] = pSliceB[(j + 2) * imageWidth + k + 1]
          - pSliceB[j * imageWidth + k + 1];
    dy[7] = pSliceB[(j + 2) * imageWidth + k]
          - pSliceB[j * imageWidth + k];
}

dz[0] = pSliceB[j * imageWidth + k]
      - pSliceC[j * imageWidth + k];

dz[1] = pSliceB[j * imageWidth + k + 1]
      - pSliceC[j * imageWidth + k + 1];

dz[2] = pSliceB[(j + 1) * imageWidth + k + 1]
      - pSliceC[(j + 1) * imageWidth + k + 1];

dz[3] = pSliceB[(j + 1) * imageWidth + k]
      - pSliceC[(j + 1) * imageWidth + k];

dz[4] = pSliceD[j * imageWidth + k]
      - pSliceA[j *imageWidth + k];

dz[5] = pSliceD[j * imageWidth + k + 1]
      - pSliceA[j * imageWidth + k + 1];

dz[6] = pSliceD[(j + 1) * imageWidth + k + 1]
      - pSliceA[(j + 1) * imageWidth + k + 1];

dz[7] = pSliceD[(j + 1) * imageWidth + k]
```

```
        - pSliceA[(j + 1) * imageWidth + k];

    //calculate triangle vertex's coordinate value
    //and gradient
    vertPos = 0;
    facePos = 0;

    for (w=0; w<12; w++)
    {
        // 根据 g_EdgeTable 对应值判断 Cube 的那一条边与等值面有交点
        if (g_EdgeTable[cubeType] & (1 << w))
        {
            switch (w)
            {
            case 0:
                offset = j * imageWidth + k;
                edgeGroup = bottomXEdge;
                break;

            case 1:
                offset = j * imageWidth + k + 1;
                edgeGroup = bottomYEdge;
                break;

            case 2:
                offset = (j+1) * imageWidth + k;
                edgeGroup = bottomXEdge;
                break;

            case 3:
                offset = j * imageWidth + k;
                edgeGroup = bottomYEdge;
                break;

            case 4:
                offset = j * imageWidth + k;
                edgeGroup = topXEdge;
                break;

            case 5:
```

```
            offset = j * imageWidth + k + 1;
            edgeGroup = topYEdge;
            break;

        case 6:
            offset = (j+1) * imageWidth + k;
            edgeGroup = topXEdge;
            break;

        case 7:
            offset = j * imageWidth + k;
            edgeGroup = topYEdge;
            break;

        case 8:
            offset = j * imageWidth + k;
            edgeGroup = zEdge;
            break;

        case 9:
            offset = j * imageWidth + k + 1;
            edgeGroup = zEdge;
            break;

        case 10:
            offset = (j+1)*imageWidth + k+1;
            edgeGroup = zEdge;
            break;

        case 11:
            offset = (j+1)*imageWidth + k;
            edgeGroup = zEdge;
            break;
        }

        // 该边上的顶点是否已经在上一层中生成
        if (edgeGroup[offset] == -1)
        {
            int index1;
            int index2;
```

```
VT s1, s2, s;
float x1, y1, z1, nx1, ny1, nz1;
float x2, y2, z2, nx2, ny2, nz2;

// 得到该边两端点的索引进而得到两点的灰度值
index1 = g_CoordTable[w][3];
index2 = g_CoordTable[w][4];
s1 = cubegrid[index1];
s2 = cubegrid[index2];

if (s1 < highThreshold && s1 > lowThreshold)
{
    if (s2 >= highThreshold)
    {
        s = highThreshold;
    }
    else if (s2 <= lowThreshold)
    {
        s = lowThreshold;
    }
}
else if (s2 < highThreshold && s2 > lowThreshold)
{
    if (s1 >= highThreshold)
    {
        s = highThreshold;
    }
    else if (s1 <= lowThreshold)
    {
        s = lowThreshold;
    }
}

// 计算两端点的实际坐标
x1 = (k + g_CoordVertex[index1][0]) * XSpace;
y1 = (j + g_CoordVertex[index1][1]) * YSpace;
z1 = (i + g_CoordVertex[index1][2]) * ZSpace;
x2 = (k + g_CoordVertex[index2][0]) * XSpace;
y2 = (j + g_CoordVertex[index2][1]) * YSpace;
z2 = (i + g_CoordVertex[index2][2]) * ZSpace;
```

```
// 计算两端点处的法向量
nx1 = dx[index1] / XSpace;
ny1 = dy[index1] / YSpace;
nz1 = dz[index1] / ZSpace;
nx2 = dx[index2] / XSpace;
ny2 = dy[index2] / YSpace;
nz2 = dz[index2] / ZSpace;

float factor = ((float)(s - s1))
               / ((float)(s2 - s1));

// 插值计算交点坐标
vertPoint[vertPos][0] = factor * (x2 - x1) + x1;
vertPoint[vertPos][1] = factor * (y2 - y1) + y1;
vertPoint[vertPos][2] = factor * (z2 - z1) + z1;

// 插值计算交点处法向量
vertPoint[vertPos][3] = factor*(nx1-nx2)-nx1;
vertPoint[vertPos][4] = factor*(ny1-ny2)-ny1;
vertPoint[vertPos][5] = factor*(nz1-nz2)-nz1;
// 法向量归一化
squaroot=sqrt(vertPoint[vertPos][3]
            * vertPoint[vertPos][3]
            + vertPoint[vertPos][4]
            * vertPoint[vertPos][4]
            + vertPoint[vertPos][5]
            * vertPoint[vertPos][5]);
if (squaroot <= 0) squaroot = 1.0;
vertPoint[vertPos][3] /= squaroot;
vertPoint[vertPos][4] /= squaroot;
vertPoint[vertPos][5] /= squaroot;

// 更新包围盒数据
if(min[0] > vertPoint[vertPos][0])
    min[0] = vertPoint[vertPos][0];
if(max[0] < vertPoint[vertPos][0])
    max[0] = vertPoint[vertPos][0];
if(min[1] > vertPoint[vertPos][1])
    min[1] = vertPoint[vertPos][1];
```

```
            if(max[1] < vertPoint[vertPos][1])
                max[1] = vertPoint[vertPos][1];
            if(min[2] > vertPoint[vertPos][2])
                min[2] = vertPoint[vertPos][2];
            if(max[2] < vertPoint[vertPos][2])
                max[2] = vertPoint[vertPos][2];

            // 记录新生成的顶点索引
            cellVerts[w] = vNumber;
            edgeGroup[offset] = cellVerts[w];
            vNumber ++;
            vertPos ++;
        }
        else
        {
            // 若该顶点已经在上一层生成，则直接得到其索引
            cellVerts[w] = edgeGroup[offset];
        }
    }
}

// 存储当前 Cube 新生成的顶点及其法向量数据
index_type m = output->GetVertexNumber();
if (vNumber > m)
{
    output->SetVertexNumber(m+100000);
    pVertex = output->GetVertexData();
}
memcpy(pVertex + 6 * (vNumber - vertPos), vertPoint,
       sizeof(float) * 6 * vertPos);

// 记录新生成的三角面片数据
w = 0;
while (g_TriTable[cubeType][w] != -1)
{
    for (r=0; r<3; r++)
    {
        triIndex[facePos][r] =
            cellVerts[g_TriTable[cubeType][w++]];
    }
```

```
                facePos ++;
                fNumber ++;
            }

            m = output->GetFaceNumber();
            if (fNumber > m)
            {
                output->SetFaceNumber(m+100000);
                pFace = output->GetFaceData();
            }
            memcpy(pFace + 3 * (fNumber - facePos), triIndex,
                   sizeof(unsigned int) * 3 * facePos);
        }

        memcpy(bottomXEdge, topXEdge, sizeof(index_type) * imageSize);
        memcpy(bottomYEdge, topYEdge, sizeof(index_type) * imageSize);
        memset(topXEdge, -1, sizeof(index_type) * imageSize);
        memset(topYEdge, -1, sizeof(index_type) * imageSize);
        memset(zEdge, -1, sizeof(index_type) * imageSize);
    }

    delete []tempSlice;

    delete []bottomXEdge;
    delete []bottomYEdge;
    delete []topXEdge;
    delete []topYEdge;
    delete []zEdge;

    // 设置输出 Mesh 的相关参数
    output->SetVertexNumber(vNumber);
    output->SetFaceNumber(fNumber);
    output->SetBoundingBox(min[0], max[0], min[1], max[1],
                           min[2], max[2]);

    return 1;
}
```

### 3.1.2 基于分割的Marching Cubes方法[1]

众所周知，医学图像具有模糊性。首先，医学图像具有灰度上的模糊性。在同一种组织中密度值会出现大幅度的变化如骨骼中股骨，鼻窦骨骼和牙齿的密度就有很大差别；在同一个物体中密度值也不均匀如股骨外表面和内部的骨髓的密度。其次，医学图像具有几何上的模糊性。在一个边界上的大体素中常常同时包含边界和物体两种物质；图像中物体的边缘，拐角及区域间的关系都难以精确的加以描述。一些病变组织由于侵袭周围组织其边缘无法明确界定。然后，还有不确定性知识的影响。通常正常组织或部位没有的结构在病变情况下出现，如脏器表面的肿物，骨骼表面的骨刺，它的出现给建造模型带来困难。医学图像的这些特点为它的三维重建带来很多困难。

基于以上考虑，我们采用了基于分割的等值面生成算法（SEGMC），该算法将分割结果作为 MC 的输入，这样可以根据图像特征选择最恰当的分割方法，利用分割结果构造等值面。

图像分割是进行表面重建的基础,分割的效果直接影响到表面重建的速度和重建后模型的视觉效果。通过分割可以帮助医生将感兴趣的物体（病变组织等）提取出来，减少了三维体数据的数据量，为表面重建和显示提供了方便，并使得医生能够对病变组织进行定性及定量的分析，从而提高医学诊断的准确性和科学性。

这里所介绍的算法SEGMC由于将分割与MC相结合，它摆脱了SMC只能用阈值分割的局限性，该算法的模块性和可扩充性好，可以将各种分割算法集成到算法中。其算法的简易流程如 图 3-11 所示。

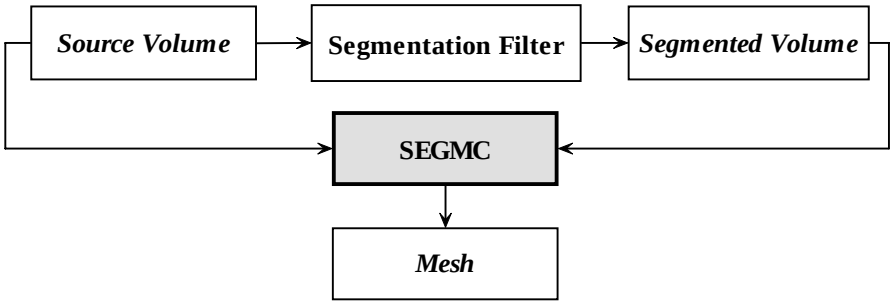



图 3-11 SEGMC 简易流程示意

该算法在 MITK 中由 mitkBinMarchingCubes 实现。mitkBinMarchingCubes

类接受两个输入：一个是原始的 Volume，用于计算法向量；另一个是原始 Volume 经分割算法所产生的二值数据，用于计算等值点坐标，只要当前 Cube 的某边两端点灰度值不同，则认为该边与等值面相交。交点坐标与法向量均采用中点选择的方法计算。

算法的具体实现与 3.1.1 所述类似，只是在判断有无交点及交点坐标与法向量的计算上有所区别，这里就不再详细说明，感兴趣的读者可以自己修改 3.1.1 中的代码实现该算法。

## 3.2 MITK 中的表面绘制框架

### 3.2.1 表面绘制框架的设计

在介绍面绘制框架之前，首先介绍一下 MITK 中的绘制模型。

MITK中的绘制模型如 图 3-12 所示，其中，View是Target的一种，用于将计算所得的结果显示在计算机屏幕上。它本身并不做实际的绘制工作，而只是提供一个显示环境。所有计算结果的绘制实际上由MITK中的不同Model来实现，Model在概念上代表场景中的一个待画的物体，可以是一幅图像，也可以是三维图形，其实际的绘制由相应的Model来实现。View中维护了一个Model的列表，可以通过View提供的接口向这个列表中添加或删除Model，而在View的绘制函数OnDraw()里遍历列表中的每一个Model，调用其虚函数Render()将其最终绘制出来。

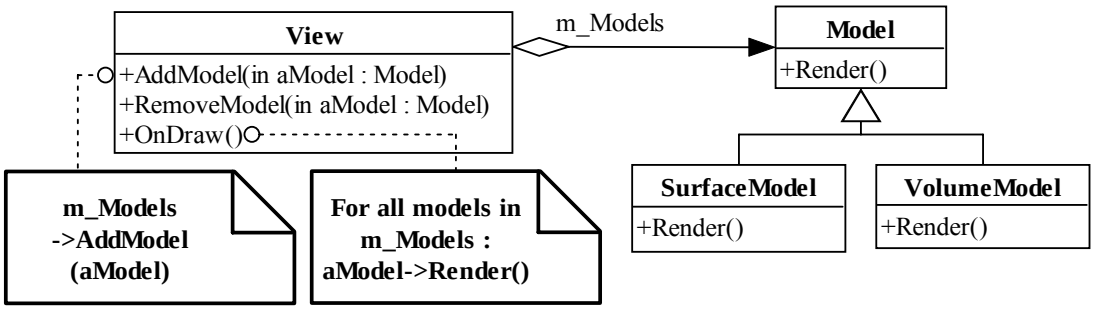



图 3-12 MITK 的绘制模型

MITK中跟面绘制相关的Model是SurfaceModel，其主要任务是实现父类里规定的接口Render()来绘制表面重建算法生成的三角网格数据。但是，绘制三角片

网格的方法多种多样，而且加光照模型时设置表面材质属性的参数众多，特别是针对不同级别的硬件配置有不同的硬件加速算法，而这些算法往往不是通用的。为了最大程度的提升面绘制的整体性能并且保证最大限度的灵活性，我们设计了如 图 3-13 所示的面绘制框架。

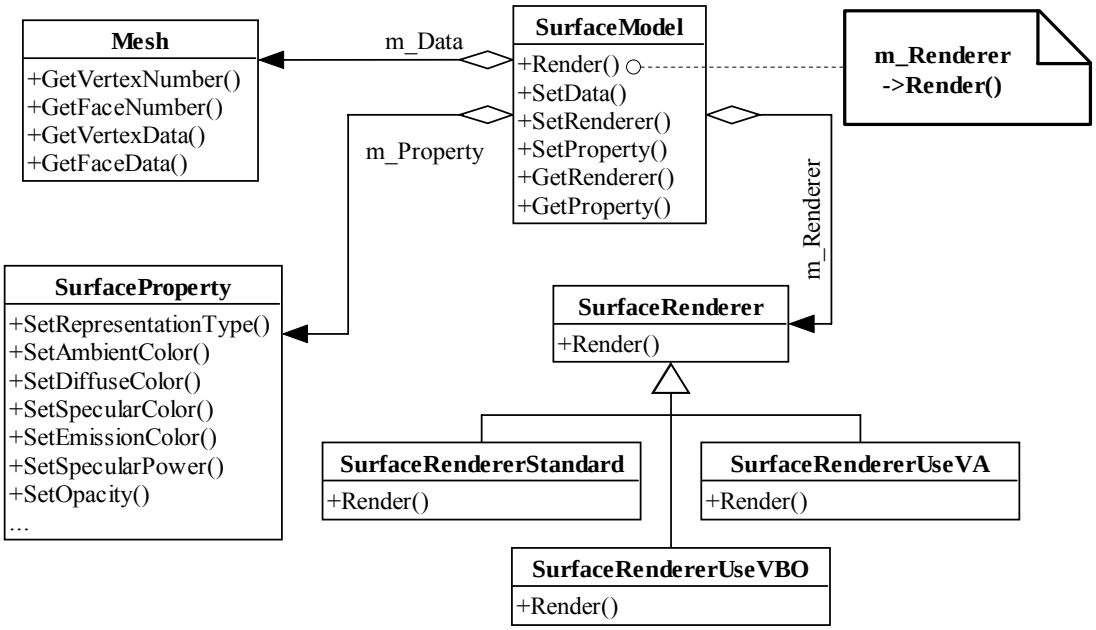



图 3-13 MITK 的面绘制框架

从图中可以看出，SurfaceModel 拥有三个类成员：Mesh、SurfaceProperty 和 SurfaceRenderer。Mesh 提供对生成的三角面片数据的访问；SurfaceProperty 维护表面模型的材质属性，并且提供给用户修改这些属性参数的接口；而 SurfaceRenderer 则负责最终实际的绘制工作。SurfaceModel 的 Render()函数并未做任何实际的绘制，而是通过 m_Renderer 成员调用 SurfaceRenderer 的 Render()函数来实现对本 Model 的绘制，具体的绘制方法又由 SurfaceRenderer 的子类实现。SurfaceRenderer 的各个子类代表了不同的面绘制方法，目前 MITK 中提供了三种面绘制的 Renderer：标准的 Renderer（SurfaceRendererStandard）、采用 OpenGL 的顶点数组加速绘制的 Renderer（SurfaceRendererUseVA）和采用 OpenGL 的 VBO 扩展加速绘制的 Renderer（SurfaceRendererUseVBO），这些具体的方法将在下一节中做详细的介绍。

这种模块化的面绘制框架使得往里面添加新的绘制方法变得非常简单，只要

从 SurfaceModel 再派生出相应的子类就可以了，而通过 SurfaceModel 的 SetRenderer()接口可以随意更换当前 SurfaceModel 的绘制方法。

### 3.2.2 表面绘制框架的实现

(1) SurfaceModel 的实现

SurfaceModel 中包含三个成员变量：

```
mitkRCPtr<mitkMesh> m_Data;
mitkRCPtr<mitkSurfaceProperty> m_Property;
mitkRCPtr<mitkSurfaceRenderer> m_Renderer;
```

它们均采用 Smart Pointer 包装，为的是在 SurfaceModel 存在的过程中始终保持这些指针指向有效的对象。通过如下接口可以方便地对它们进行访问：

```
void SetRenderer(mitkSurfaceRenderer *renderer);
mitkSurfaceRenderer* GetRenderer();
void SetProperty(mitkSurfaceProperty *prop);
mitkSurfaceProperty* GetProperty();
void SetData(mitkMesh *data);
mitkMesh* GetData();
```

其中，SetRenderer()、SetProperty()只是简单的赋值操作，GetData()直接返回 m_Data，没什么好说的。余下的 GetRenderer()、GetProperty()和 SetData()则需要一些额外的处理，其代码如下所示：

```
mitkSurfaceRenderer* mitkSurfaceModel::GetRenderer()
{
    if (m_Renderer == NULL)
    {
        // 要用 Vertex Array 或 Vertex Buffer Object，
        // 必须保证 OpenGL 版本在 1.1 以上
        #ifdef GL_VERSION_1_1
            if (g_IsExtensionSupported("GL_ARB_vertex_buffer_object"))
            {
                // 如果支持 VBO 扩展，
                // 则选用 mitkSurfaceRendererUseVBO 作为 Renderer
                m_Renderer = new mitkSurfaceRendererUseVBO;
            }
            else
            {
                // 否则，选用 mitkSurfaceRendererUseVA 作为 Renderer
```

```
                m_Renderer = new mitkSurfaceRendererUseVA;
            }
        #else
            // 最坏的情况，只能用标准的也是速度最慢的 Renderer
            m_Renderer = new mitkSurfaceRendererStandard;
        #endif
    }
    return m_Renderer;
}
//---------------------------------------------------------------------
mitkSurfaceProperty* mitkSurfaceModel::GetProperty()
{
    if (m_Property == NULL)
    {
        // 若无 SurfaceProperty 对象则立即产生一个
        m_Property = new mitkSurfaceProperty;
    }
    return m_Property;
}
//---------------------------------------------------------------------
void mitkSurfaceModel::SetData(mitkMesh *data)
{
    m_Data = data;
    if (data == NULL) return;

    // 得到包围盒
    m_Data->GetBoundingBox(m_Bounds);

    // 重新计算模型中心位置
    m_Origin[0] = (m_Bounds[1] + m_Bounds[0]) / 2.0f;
    m_Origin[1] = (m_Bounds[3] + m_Bounds[2]) / 2.0f;
    m_Origin[2] = (m_Bounds[5] + m_Bounds[4]) / 2.0f;

    // 设置更改标志，让 View 重新计算相机位置
    m_DataChanged = true;

    // 设置更改标志，以更新 Model 所维护的模型变换矩阵
    m_MatrixModified = 1;
}
//---------------------------------------------------------------------
```

可以看到，在 GetRenderer()中，根据当前硬件的配置选用了最合适的 Renderer 来绘制这个 SurfaceModel，以求得到比较好的面绘制性能。当然，用户也可以在自己程序的任何地方通过 SetRenderer()函数直接更换 MITK 配置的 Renderer。其中的 g_IsExtensionSupported()用于检测 OpenGL 的某项扩展是否被当前硬件支持，其原型在 mitkOpenGLExam.h 中。

在 SurfaceModel 的 Render()函数中，只是调用其 SurfaceRenderer 成员 m_Renderer 的 Render()函数来实现 Model 的绘制：

```
int mitkSurfaceModel::Render(mitkView *view)
{
    if(m_Data == NULL)
    {
        return 0;
    }

    this->GetRenderer()->Render(view, this);

    return 1;
}
```

(2) SurfaceProperty 的实现

SurfaceProperty 中包含一组数据成员，记录了表面材质属性的参数值：

```
    float m_Color[4];           //表面颜色
    float m_AmbientColor[4];    //环境光颜色
    float m_DiffuseColor[4];    //散射光颜色
    float m_SpecularColor[4];   //反射光颜色
    float m_EmissionColor[4];   //发射光颜色
    float m_EdgeColor[4];       //采用线框方式绘制时线框的颜色
    float m_Ambient;            //环境光系数
    float m_Diffuse;            //散射光系数
    float m_Specular;           //反射光系数
    float m_Emission;           //发射光系数
    float m_SpecularPower;      //发射光强度
    float m_Opacity;            //不透明度
    float m_PointSize;          //采用顶点方式绘制时点的大小
    float m_LineWidth;          //采用线框方式绘制时线的宽度
```

```
    int m_RepresentationType;    //绘制方式
    int m_InterpolationType;     //插值方式
    int m_LineStippleRepeatFactor;          //点划线重复因子
    unsigned short m_LineStipplePattern;    //点划线模板
    unsigned short m_Modified;              //参数是否被更改
```

每种颜色值按 RGBA 四个分量存储，其中，m_Color 的值由如下算法得到：

```
float* mitkSurfaceProperty::GetColor()
{
    float norm, total = m_Ambient + m_Diffuse + m_Specular + m_Emission;

    if ( total > 0 )
    {
        norm = 1.0f / total;
    }
    else
    {
        norm = 0.0f;
    }

    for (int i=0; i<4; ++i)
    {
        m_Color[i] = m_AmbientColor[i] * m_Ambient * norm
                   + m_DiffuseColor[i] * m_Diffuse * norm
                   + m_SpecularColor[i] * m_Specular * norm
                   + m_EmissionColor[i] * m_Emission * norm;
    }

    return m_Color;
}
```

m_Opacity 只是为方便而提供，实际绘制中其含意与 Diffuse Color 的 Alpha 分量（即 m_DiffuseColor[3]）相同，均代表物体表面的不透明程度，其值也始终保持一致，值为 0 表示完全透明，1 表示完全不透明。

m_RepresentationType 记录三角面片数据的绘制方式，其值为如下三个值之一：MITK_MESH_POINTS（仅绘制顶点）、MITK_MESH_WIREFRAME（绘制线框模型）和 MITK_MESH_SURFACE（绘制加光照的表面模型）。

m_InterpolationType 记录绘制表面模型时的顶点着色的插值方式，其值为如下三个值之一：MITK_SURFACE_FLAT（平面着色方式，计算最简便但效果最差），MITK_SURFACE_GOURAUD（Gouraud 着色，计算复杂但效果较好），MITK_SURFACE_PHONG（Phong 着色，计算非常耗时，但可以得到更加平滑的效果）。其中，Phong 着色并未在目前 MITK 提供的 Renderer 中实现，而只是采取与 Gouraud 着色相同的方式进行处理，所以实际上只有两种插值方式在工作。

m_LineStippleRepeatFactor 和 m_LineStipplePattern 指定当采用线框方式绘制模型并采用点划线时点划线的线型。其中，m_LineStipplePattern 为一个 16 位整型数，它的二进制位模板决定在绘制线段时哪个片断将被绘出（二进制位为 1 的绘出，为 0 的则不绘出），m_LineStippleRepeatFactor 则指定绘制模板中每个二进制位的重复次数，比如其值为 3 时，模板中每个二进制位将重复使用 3 次后才使用下一位。

SurfaceProperty 提供了丰富的 Set/Get 接口来访问上述参数，接口函数请参考 mitkSurfaceProperty.h，这里就不再多说了。

具体的 Renderer 将根据上述参数来决定如何绘制其所在的 SurfaceModel。

在实际的使用过程中，当要调整某一个 SurfaceModel 的部分绘制参数时，一般情况下我们并不提倡额外保留一个指向 SurfaceProperty 的指针来对其中的参数进行设置，而是直接通过 SurfaceModel 的 GetProperty()接口实现，如下所示，以免出现不一致的情况而发生错误：

```
aSurfModel->GetProperty()->SetAmbientColor(0.75f, 0.75f, 0.75f, 1.0f);
aSurfModel->GetProperty()->SetDiffuseColor(1.0f, 0.57f, 0.04f, 1.0f);
aSurfModel->GetProperty()->SetSpecularColor(1.0f, 1.0f, 1.0f, 1.0f);
aSurfModel->GetProperty()->SetSpecularPower(100.0f);
aSurfModel->GetProperty()->SetEmissionColor(0.0f, 0.0f, 0.0f, 0.0f);
```

当然也可以预先生成多个不同情况下绘制所对应的 SurfaceProperty，然后在需要的时候通过 SetProperty()直接替换 SurfaceModel 的 Property。

(3) **实现具体的** SurfaceRenderer

SurfaceRenderer 是面绘制的关键，所有的绘制工作都在 SurfaceRenderer 的子类中完成。

SurfaceRenderer 继承自 Renderer，Renderer 中维护了一个裁剪平面的列表，如果额外的裁剪平面被激活，则其子类中将根据其中记录的裁剪平面实施具体的裁剪操作。

SurfaceRenderer 的子类则通过实现父类中的虚函数 Render()完成最终的绘制。下面通过 MITK 中已经实现的三个 SurfaceRenderer 子类来看一下如何实现一个具体的 SurfaceRenderer。

**SurfaceRendererStandard**

这是一个标准的 SurfaceRenderer，采用传统的 OpenGL 绘制方式，没有做任何加速的尝试，但是其兼容性是最好的，可以在任何支持 OpenGL 的计算机上运行。其 Render()函数如下：

```
void mitkSurfaceRendererStandard::Render(mitkView *view,
                                         mitkSurfaceModel *surf)
{
    // 得到需要绘制的 Mesh 数据
    mitkMesh *mesh = surf->GetData();
    if (mesh == NULL)
    {
        return;
    }

    // 取得指向点表和面表的指针
    float *vertexBuf = mesh->GetVertexData();
    unsigned int *faceBuf = mesh->GetFaceData();
    if ((vertexBuf == NULL) || (faceBuf == NULL))
    {
        return;
    }

    // 得到顶点和三角片的数量
    unsigned int vertexNum = mesh->GetVertexNumber();
    unsigned int faceNum = mesh->GetFaceNumber();

    // 得到 SurfaceModel 的 Property
    mitkSurfaceProperty *prop = surf->GetProperty();
```

```
// 是否需要 alpha 融合（半透明显示）
bool blend = ( prop->GetOpacity() < 1.0f );

// 线框显示时是否采用点划方式
bool stipple = ( prop->GetLineStipplePattern() != 0xFFFF );

if (prop->IsModified())
{
    // 若材质属性被修改，则重新设置材质属性
    this->_setMaterial(prop);
    prop->SetUnmodified();
}

glDisable(GL_TEXTURE_2D);
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
{
    // 模型的几何变换
    glMultMatrixf((GLfloat *)surf->GetModelMatrix());

    // 如果附加裁剪平面被激活则设定裁剪平面
    if (this->GetClipping())
    {
        mitkPlane *aPlane = NULL;
        int count = 0;
        mitkList *planes = this->GetClippingPlanes();
        double equation[4];
        float nx, ny, nz, ox, oy, oz;

        for (planes->InitTraversal();
               (aPlane = (mitkPlane *)planes->GetNextItem())
                   && (count<6);)
        {
            aPlane->GetNormal(nx, ny, nz);
            aPlane->GetOrigin(ox, oy, oz);
            equation[0] = (double)nx;
            equation[1] = (double)ny;
            equation[2] = (double)nz;
            equation[3] = - (double)(nx*ox + ny*oy + nz*oz);
            glClipPlane(GL_CLIP_PLANE0+count,equation);
```

```
            glEnable(GL_CLIP_PLANE0+count);
            count ++;
        }
    }

    // 根据不同的显示方式绘制 SurfaceModel
    switch(prop->GetRepresentationType())
    {
    case MITK_MESH_POINTS: //点显示
        glDisable(GL_LIGHTING);
        this->_drawPoints(vertexNum, vertexBuf);
        break;

    case MITK_MESH_WIREFRAME: //线框显示
        if (stipple) glEnable(GL_LINE_STIPPLE);
        else glDisable(GL_LINE_STIPPLE);
        glDisable(GL_LIGHTING);
        glColor4fv(prop->GetEdgeColor());
        this->_drawWireFrame(faceNum, vertexBuf, faceBuf);
        if (stipple) glDisable(GL_LINE_STIPPLE);
        break;

    case MITK_MESH_SURFACE: //面显示
        glEnable(GL_DEPTH_TEST);
        if (blend)
        {
            glEnable(GL_BLEND);
            glDepthMask(GL_FALSE);
        }
        glEnable(GL_LIGHTING);
        glEnable(GL_NORMALIZE);
        if (mesh->IsClockwise()) glFrontFace(GL_CW);
        this->_drawSurface(faceNum, vertexBuf, faceBuf);
        if (blend) glDisable(GL_BLEND);
        if (mesh->IsClockwise()) glFrontFace(GL_CCW);
        if (blend)
        {
            glDisable(GL_BLEND);
            glDepthMask(GL_TRUE);
        }
```

```
            glDisable(GL_NORMALIZE);
            glDisable(GL_LIGHTING);
            glDisable(GL_DEPTH_TEST);
            break;
        }
    }
    glPopMatrix();
}
```

其中，集中设置表面材质属性的函数_setMaterial()如下所示：

```
void mitkSurfaceRendererStandard::_setMaterial(mitkSurfaceProperty
*prop)
{
    GLenum method, face;

    // 设置插值方式
    switch (prop->GetInterpolationType())
    {
    case MITK_SURFACE_FLAT:
        method = GL_FLAT;
        break;

    case MITK_SURFACE_GOURAUD:
    case MITK_SURFACE_PHONG:
        method = GL_SMOOTH;
        break;

    default:
        method = GL_SMOOTH;
    }
    glShadeModel(method);

    // 点和线段的绘制属性设置
    glPointSize(prop->GetPointSize());
    glLineWidth(prop->GetLineWidth());
    glLineStipple(prop->GetLineStippleRepeatFactor(),
                  prop->GetLineStipplePattern());

    // 表面材质属性设置
    face = GL_FRONT_AND_BACK;
```

```
    glMaterialfv(face, GL_AMBIENT, prop->GetAmbientColor());
    glMaterialfv(face, GL_DIFFUSE, prop->GetDiffuseColor());
    glMaterialfv(face, GL_SPECULAR, prop->GetSpecularColor());
    glMaterialfv(face, GL_EMISSION, prop->GetEmissionColor());
    glMaterialf(face, GL_SHININESS, prop->GetSpecularPower());

    // 设置 Alpha 融合运算方式（用于表面的半透明显示）
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
}
```

MITK 为三维模型的绘制环境维护了三个变换矩阵及其逆矩阵，它们是模型变换矩阵及其逆矩阵、视图变换矩阵及其逆矩阵和投影变换矩阵及其逆矩阵，这里对模型的几何变换（包括平移、旋转、缩放）即通过直接乘上模型变换矩阵来实现，而没有通过调用 glTranslatef()、glRotatef()等函数实现，这样不仅简化代码书写，而且在一定程度上提高了绘制的效率。

三种显示方式的绘制实际是在_drawPoints()、_drawWireFrame()和_drawSurface()三个函数中实现的，其代码如下：

```
void mitkSurfaceRendererStandard::_drawPoints(unsigned int vertexNum,
float *vertexData)
{
    glBegin(GL_POINTS);
        for (unsigned int i=0; i<vertexNum; ++i)
            glVertex3fv(&vertexData[6*i]);
    glEnd();
}
//----------------------------------------------------------------------
void mitkSurfaceRendererStandard::_drawWireFrame(unsigned int faceNum,
                                                 float *vertexData, unsigned
int *faceData)
{
    glBegin(GL_LINES);
        for (unsigned int i=0; i<faceNum; ++i)
        {
            glVertex3fv(&vertexData[6*faceData[3*i]]);
            glVertex3fv(&vertexData[6*faceData[3*i+1]]);
            glVertex3fv(&vertexData[6*faceData[3*i+1]]);
            glVertex3fv(&vertexData[6*faceData[3*i+2]]);
            glVertex3fv(&vertexData[6*faceData[3*i+2]]);
```

```
            glVertex3fv(&vertexData[6*faceData[3*i]]);
        }
    glEnd();
}
//---------------------------------------------------------------------
void mitkSurfaceRendererStandard::_drawSurface(unsigned int faceNum,
                                               float *vertexData, unsigned
int *faceData)
{
    glBegin(GL_TRIANGLES);
        for (unsigned int i=0; i<faceNum; ++i)
        {
            glNormal3fv(vertexData + 6*faceData[3*i] + 3);
            glVertex3fv(vertexData + 6*faceData[3*i]);

            glNormal3fv(vertexData + 6*faceData[3*i+1] + 3);
            glVertex3fv(vertexData + 6*faceData[3*i+1]);

            glNormal3fv(vertexData + 6*faceData[3*i+2] + 3);
            glVertex3fv(vertexData + 6*faceData[3*i+2]);
        }
    glEnd();
}
//---------------------------------------------------------------------
```

可以看出，所有的绘制都是采用最基本的 OpenGL 代码，虽然慢，但是可以适应绝大多数的绘制环境。

**SurfaceRendererUseVA**[7][8]

这个 SurfaceRenderer 使用了 OpenGL 中的顶点数组（Vertex Array）来加速三角面片的绘制。采用顶点数组，只需将指向顶点数据的指针传给 OpenGL，并在绘制时告诉 OpenGL 顶点数据的排列方式，由 OpenGL 决定按何种顺序以及如何绘制这些顶点，这样就可以避免绘制每个顶点时 glVertex*()、glNormal*() 等函数调用的开销，从而大大提高绘制速度。

该类包含了指向顶点数组的指针 m_Vertices 以及指向边表和面表的指针 m_Edges 和 m_Faces，其中边表和面表均存放构成边和面的顶点在顶点数组中的索引。顶点数组和面表可以直接从 Mesh 中得来，而边表还需创建。理想情况下，

边表所存储的边应该是无重复的，但是考虑到要创建这样的边表，算法复杂度比较高，故采用直接从面表创建的方法，这样，被两个面共用的边会出现两次，占用的存储空间最多是理想情况的两倍，但创建的速度要快的多。这部分的工作在_buildArrays()中完成，如下所示：

```
bool mitkSurfaceRendererUseVA::_buildArrays(mitkMesh *mesh)
{
    // 清除旧的数据
    this->_clearArrays();

    // 由 Mesh 直接得到顶点数组和面表数据
    m_VertNum = mesh->GetVertexNumber();
    m_FaceNum = mesh->GetFaceNumber();
    m_Vertices = mesh->GetVertexData();
    m_Faces = mesh->GetFaceData();

    if (m_Vertices==NULL || m_Faces==NULL)
    {
        mitkErrorMessage("Something wrong with mesh data!");
        return false;
    }

    // 创建边表
    m_EdgeNum = m_FaceNum * 3;
    m_Edges = new unsigned int[m_EdgeNum*2];
    if (m_Edges == NULL)
    {
        mitkErrorMessage("Not enough memory for building edge table using
                          vertex array!");
        return false;
    }
    unsigned int *curFace = m_Faces;
    unsigned int *curEdge = m_Edges;
    for (unsigned long i=0; i<m_FaceNum; ++i)
    {
        curEdge[0] = curFace[0];
        curEdge[1] = curFace[1];

        curEdge[2] = curFace[0];
```

```
        curEdge[3] = curFace[2];

        curEdge[4] = curFace[1];
        curEdge[5] = curFace[2];

        curFace += 3;
        curEdge += 6;
    }

    return true;
}
```

最后，该类的 Render()函数如下所示，框架基本与 SurfaceRendererStandard 同，因此只在不同之处加以注释：

```
void mitkSurfaceRendererUseVA::Render(mitkView *view,
                                      mitkSurfaceModel *surf)
{
    mitkMesh *mesh = surf->GetData();
    if (mesh == NULL)
    {
        return;
    }

    if (surf->GetDataModifyStatus())
    {
        // 若 SurfaceModel 包含的 Mesh 数据有所改变，则重新创建相关数组
        if (!this->_buildArrays(mesh)) return;
    }

    mitkSurfaceProperty *prop = surf->GetProperty();

    bool blend = ( prop->GetOpacity() < 1.0f );
    bool stipple = ( prop->GetLineStipplePattern() != 0xFFFF );

    if (prop->IsModified())
    {
        this->_setMaterial(prop);
        prop->SetUnmodified();
    }
```

```
glDisable(GL_TEXTURE_2D);
glEnableClientState(GL_VERTEX_ARRAY); //启用顶点数组
glEnableClientState(GL_NORMAL_ARRAY); //启用法向量数组
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
{
    glMultMatrixf((GLfloat *)surf->GetModelMatrix());

    if (this->GetClipping())
    {
        mitkPlane *aPlane = NULL;
        int count = 0;
        mitkList *planes = this->GetClippingPlanes();
        double equation[4];
        float nx, ny, nz, ox, oy, oz;

        for (planes->InitTraversal();
               (aPlane = (mitkPlane *)planes->GetNextItem())
                   && (count<6);)
        {
            aPlane->GetNormal(nx, ny, nz);
            aPlane->GetOrigin(ox, oy, oz);
            equation[0] = (double)nx;
            equation[1] = (double)ny;
            equation[2] = (double)nz;
            equation[3] = - (double)(nx*ox + ny*oy + nz*oz);
            glClipPlane(GL_CLIP_PLANE0+count,equation);
            glEnable(GL_CLIP_PLANE0+count);
            count ++;
        }
    }

    switch(prop->GetRepresentationType())
    {
    case MITK_MESH_POINTS:
        glDisable(GL_LIGHTING);

        // 指定顶点数组并按顶点方式绘制
        glVertexPointer(3, GL_FLOAT, 6*sizeof(float), m_Vertices);
        glDrawArrays(GL_POINTS, 0, m_VertNum);
```

```
        break;

    case MITK_MESH_WIREFRAME:
        if (stipple) glEnable(GL_LINE_STIPPLE);
        else glDisable(GL_LINE_STIPPLE);
        glDisable(GL_LIGHTING);
        glColor4fv(prop->GetEdgeColor());

        // 指定顶点数组并按索引方式绘制所有的边
        glVertexPointer(3, GL_FLOAT, 6*sizeof(float), m_Vertices);
        glDrawElements(GL_LINES, m_EdgeNum * 2,
                       GL_UNSIGNED_INT, m_Edges);

        if (stipple) glDisable(GL_LINE_STIPPLE);
        break;

    case MITK_MESH_SURFACE:
        glEnable(GL_DEPTH_TEST);
        if (blend)
        {
            glEnable(GL_BLEND);
            glDepthMask(GL_FALSE);
        }
        glEnable(GL_LIGHTING);
        glEnable(GL_NORMALIZE);
        if (mesh->IsClockwise()) glFrontFace(GL_CW);

        // 指定顶点数组及法向量数组并按索引方式绘制所有三角片
        glVertexPointer(3, GL_FLOAT, 6*sizeof(float), m_Vertices);
        glNormalPointer(GL_FLOAT, 6*sizeof(float), m_Vertices+3);
        glDrawElements(GL_TRIANGLES, m_FaceNum * 3,
                       GL_UNSIGNED_INT, m_Faces);

        if (blend) glDisable(GL_BLEND);
        if (mesh->IsClockwise()) glFrontFace(GL_CCW);
        if (blend)
        {
            glDisable(GL_BLEND);
            glDepthMask(GL_TRUE);
```

```
            }
            glDisable(GL_NORMALIZE);
            glDisable(GL_LIGHTING);
            glDisable(GL_DEPTH_TEST);
            break;
        }
    }
    glDisableClientState(GL_VERTEX_ARRAY); //关闭顶点数组
    glDisableClientState(GL_NORMAL_ARRAY); //关闭法向量数组
    glPopMatrix();
}
```

该绘制方法仅被 OpenGL 1.1 及以上版本支持。

**SurfaceRendererUseVBO**[9]

这个 SurfaceRenderer 使用了 OpenGL 的 VBO（Vertex Buffer Object）扩展来加速三角面片的绘制。

VBO 扩展的工作方式和顶点数组很像，唯一的区别就是 VBO 将数据直接加载到显卡的高性能显存里，省去了系统内存与显存之间频繁的数据传输，因此大大提高了渲染速度。

该扩展是依赖于较新的硬件的，所以我们在使用它之前必须确保当前所使用的显卡支持这一扩展，方法是使用 glGetString(GL_EXTENSIONS)得到所有当前被支持的 OpenGL 扩展的字符串，然后查询其中是否包含“GL_ARB_vertex_buffer_object”。这一功能已被封装进全局函数 g_IsExtensionSupported()中，其原型为：

```
bool g_IsExtensionSupported(const char *extension);
```

它接受扩展的名称字符串作为参数，返回一个 bool 值表示该扩展是否被支持。

由于 Microsoft 在 Windows 系列操作系统中提供的 OpenGL 库文件到现在还停留在 1.1 版本，所以如果显卡支持 VBO 扩展，我们还需要用 wglGetProcAddress() 直接从由显卡驱动提供的最新的 DLL 文件中得到使用 VBO 扩展所需的 4 个函数的指针：

```
glGenBuffersARB =
(PFNGLGENBUFFERSARBPROC)wglGetProcAddress("glGenBuffersARB");
```

```
    glBindBufferARB =
(PFNGLBINDBUFFERARBPROC)wglGetProcAddress("glBindBufferARB");
    glBufferDataARB =
(PFNGLBUFFERDATAARBPROC)wglGetProcAddress("glBufferDataARB");
    glDeleteBuffersARB =
(PFNGLDELETEBUFFERSARBPROC)wglGetProcAddress("glDeleteBuffersARB");
```

关于PFNGLGENBUFFERSARBPROC、PFNGLBINDBUFFERARBPROC、PFNGLBUFFERDATAARBPROC 和 PFNGLDELETEBUFFERSARBPROC 的声明参见 OpenGL 扩展的头文件 glext.h，该文件的最新版本可以在 http://oss.sgi.com/projects/ogl-sample/sdk.html得到。这些初始化工作在构造函数中完成：

```
mitkSurfaceRendererUseVBO::mitkSurfaceRendererUseVBO()
{
    glGenBuffersARB = NULL;
    glBindBufferARB = NULL;
    glBufferDataARB = NULL;
    glDeleteBuffersARB = NULL;

    m_VertVBO = m_FaceVBO = m_EdgeVBO = 0;
    m_Vertices = NULL;
    m_Faces = NULL;
    m_Edges = NULL;
    m_IsVBOSupported =
           g_IsExtensionSupported("GL_ARB_vertex_buffer_object");
    m_VBOBuilt = false;
    m_NormalReversed = false;

    if (m_IsVBOSupported)
    {
        glGenBuffersARB =
(PFNGLGENBUFFERSARBPROC)wglGetProcAddress("glGenBuffersARB");
       glBindBufferARB =
(PFNGLBINDBUFFERARBPROC)wglGetProcAddress("glBindBufferARB");
       glBufferDataARB =
(PFNGLBUFFERDATAARBPROC)wglGetProcAddress("glBufferDataARB");
       glDeleteBuffersARB =
(PFNGLDELETEBUFFERSARBPROC)wglGetProcAddress("glDeleteBuffersARB");
    }
```

```
}
```

在 SurfaceRendererUseVBO 中，我们为顶点数组、面表和边表分别创建了一个 VBO 对象：m_VertVBO、m_FaceVBO 和 m_EdgeVBO，其创建工作在_buildVBOs()中完成：

```
bool mitkSurfaceRendererUseVBO::_buildVBOs(mitkMesh *mesh)
{
    if (!this->_buildArrays(mesh)) return false;

    if (!m_VBOBuilt)
    {
        // 创建 VBO
        glGenBuffersARB(1, &m_VertVBO);
        glGenBuffersARB(1, &m_FaceVBO);
        glGenBuffersARB(1, &m_EdgeVBO);
        m_VBOBuilt = true;
    }

    // 绑定顶点 VBO 对象并向显卡传送顶点数组数据
    glBindBufferARB(GL_ARRAY_BUFFER_ARB, m_VertVBO);
    glBufferDataARB(GL_ARRAY_BUFFER_ARB, m_VertNum*6*sizeof(float),
                    m_Vertices, GL_STATIC_DRAW_ARB);
    m_NormalReversed = mesh->IsNormalReversed();

    // 绑定面表 VBO 对象并向显卡传送面表索引数据
    glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, m_FaceVBO);
    glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER_ARB,
                    m_FaceNum*3*sizeof(unsigned int),
                    m_Faces, GL_STATIC_DRAW_ARB);

    // 绑定边表 VBO 对象并向显卡传送边表索引数据
    glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, m_EdgeVBO);
    glBufferDataARB(GL_ELEMENT_ARRAY_BUFFER_ARB,
                    m_EdgeNum*2*sizeof(unsigned int),
                    m_Edges, GL_STATIC_DRAW_ARB);

    // 因为数据均已传送到显卡显存，
    // 所以临时申请的存放边表的系统内存可以释放了
    this->_clearArrays();
```

```
    return true;
}
```

其中的_buildArrays()与 SurfaceRendererUseVA 中的相同，不再赘述。

最后，来看一下 Render()函数中的绘制过程，同样的，只在与前面不同之处加以注释：

```
void mitkSurfaceRendererUseVBO::Render(mitkView *view,
                                       mitkSurfaceModel *surf)
{
    if (!m_IsVBOSupported) return;

    mitkMesh *mesh = surf->GetData();
    if (mesh == NULL)
    {
        return;
    }

    if (surf->GetDataModifyStatus())
    {
        // 创建 VBO 对象
        if (!this->_buildVBOs(mesh)) return;
    }

    if (m_NormalReversed != mesh->IsNormalReversed())
    {
        // 如果法线方向改变，则重新传送顶点数组数据
        m_Vertices = mesh->GetVertexData();
        glBindBufferARB(GL_ARRAY_BUFFER_ARB, m_VertVBO);
        glBufferDataARB(GL_ARRAY_BUFFER_ARB,
                        m_VertNum*6*sizeof(float),
                        m_Vertices, GL_STATIC_DRAW_ARB);
        m_Vertices = NULL;
        m_NormalReversed = mesh->IsNormalReversed();
    }

    mitkSurfaceProperty *prop = surf->GetProperty();

    bool blend = ( prop->GetOpacity() < 1.0f );
```

```
bool stipple = ( prop->GetLineStipplePattern() != 0xFFFF );

if (prop->IsModified())
{
    this->_setMaterial(prop);
    prop->SetUnmodified();
}


glDisable(GL_TEXTURE_2D);
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
{
    glMultMatrixf((GLfloat *)surf->GetModelMatrix());

    if (this->GetClipping())
    {
        mitkPlane *aPlane = NULL;
        int count = 0;
        mitkList *planes = this->GetClippingPlanes();
        double equation[4];
        float nx, ny, nz, ox, oy, oz;

        for (planes->InitTraversal();
               (aPlane = (mitkPlane *)planes->GetNextItem())
               && (count<6);)
        {
            aPlane->GetNormal(nx, ny, nz);
            aPlane->GetOrigin(ox, oy, oz);
            equation[0] = (double)nx;
            equation[1] = (double)ny;
            equation[2] = (double)nz;
            equation[3] = - (double)(nx*ox + ny*oy + nz*oz);
            glClipPlane(GL_CLIP_PLANE0+count,equation);
            glEnable(GL_CLIP_PLANE0+count);
            count ++;
        }
    }
```

```
switch(prop->GetRepresentationType())
{
case MITK_MESH_POINTS:
    glDisable(GL_LIGHTING);

    // 首先要绑定所需使用的 VBO
    glBindBufferARB(GL_ARRAY_BUFFER_ARB, m_VertVBO);
    glVertexPointer(3, GL_FLOAT, 6*sizeof(float), NULL);
    glDrawArrays(GL_POINTS, 0, m_VertNum);

    break;

case MITK_MESH_WIREFRAME:
    if (stipple) glEnable(GL_LINE_STIPPLE);
    else glDisable(GL_LINE_STIPPLE);
    glDisable(GL_LIGHTING);
    glColor4fv(prop->GetEdgeColor());

    // 首先要绑定所需使用的 VBO
    // 在指定顶点数组数据头指针时因为数据均已传送进显卡，
    // 所以只使用 NULL（实际上表示了相对位移 0）
    glBindBufferARB(GL_ARRAY_BUFFER_ARB, m_VertVBO);
    glVertexPointer(3, GL_FLOAT, 6*sizeof(float), NULL);
    glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, m_EdgeVBO);
    glDrawElements(GL_LINES, m_EdgeNum * 2,
                   GL_UNSIGNED_INT, NULL);

    if (stipple) glDisable(GL_LINE_STIPPLE);
    break;

case MITK_MESH_SURFACE:
    glEnable(GL_DEPTH_TEST);
    if (blend)
    {
        glEnable(GL_BLEND);
        glDepthMask(GL_FALSE);
    }
    glEnable(GL_LIGHTING);
    glEnable(GL_NORMALIZE);
```

```
            if (mesh->IsClockwise()) glFrontFace(GL_CW);

            // 首先要绑定所需使用的 VBO
            // 因为顶点和法向量是放在一块存储的，
            // 所以要指定两顶点数据之间的间隔 6*sizeof(float)，
            // 并且在指定法向量数组头指针时，相对的从
            // (const void *)(3*sizeof(float))开始
            glBindBufferARB(GL_ARRAY_BUFFER_ARB, m_VertVBO);
            glVertexPointer(3, GL_FLOAT, 6*sizeof(float), NULL);
            glNormalPointer(GL_FLOAT, 6*sizeof(float),
                            (const void *)(3*sizeof(float)));
            glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, m_FaceVBO);
            glDrawElements(GL_TRIANGLES, m_FaceNum * 3,
                            GL_UNSIGNED_INT, NULL);

            if (blend) glDisable(GL_BLEND);
            if (mesh->IsClockwise()) glFrontFace(GL_CCW);
            if (blend)
            {
                glDisable(GL_BLEND);
                glDepthMask(GL_TRUE);
            }
            glDisable(GL_NORMALIZE);
            glDisable(GL_LIGHTING);
            glDisable(GL_DEPTH_TEST);
            break;
        }
    }
    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
    glPopMatrix();
}
```

另外，在析构函数中归还 VBO 对象所占用的资源：

```
mitkSurfaceRendererUseVBO::~mitkSurfaceRendererUseVBO()
{
    this->_clearArrays();

    if (m_VBOBuilt)
    {
```

```
        glDeleteBuffersARB(1, &m_VertVBO);
        glDeleteBuffersARB(1, &m_FaceVBO);
        glDeleteBuffersARB(1, &m_EdgeVBO);
    }
}
```

## 3.3 小结

MITK所实现的面绘制功能采用了改进后的Marching Cubes算法进行表面重建，并提供了一个灵活的、易于扩充的绘制框架来显示重建所得的表面。图 3-14 就是运用MITK所提供的面绘制功能做出的一些多层表面重建及显示的实例。

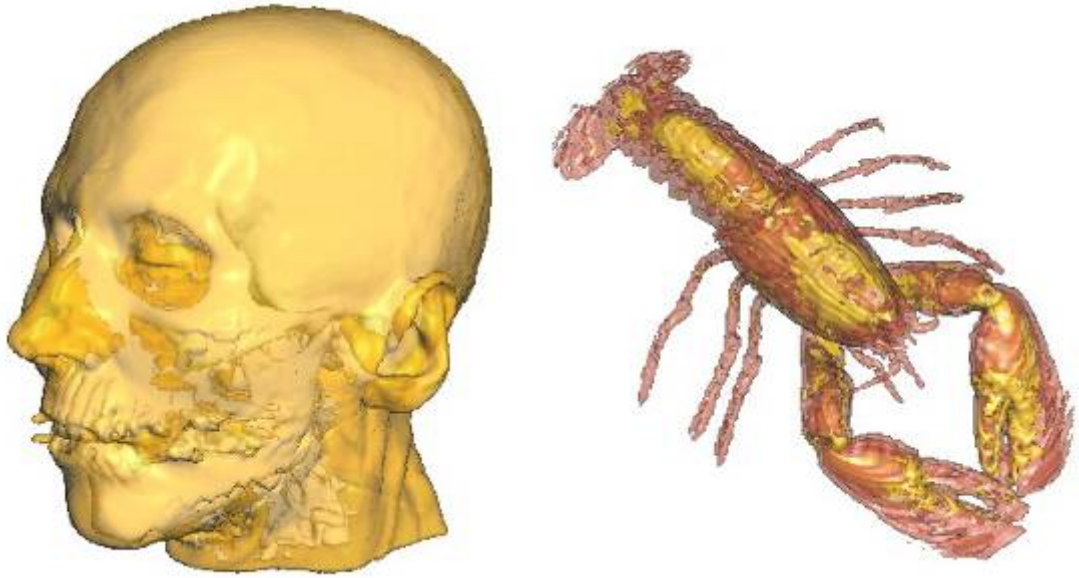


图 3-14 多层表面重建及绘制实例

### 参考文献


1. 田捷，包尚联，周明全. 医学影像处理与分析. 北京：电子工业出版社，2003.
2. W. Lorensen and H. Cline. Marching cubes: a high resolution 3D surface construction algorithm. ACM Computer Graphics, 21(4): pp. 163 –170,1987.
3. M. J. Durst. Letters: Additional reference to “Marching Cubes”. ACM Computer Graphics, 22(4): pp. 72 –73,1988.

4. Nielson G. M. ,Hamann B. The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes. IEEE Proceedings of Visualization' 91, pp. 83-91.
5. J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. ACM Computer Graphics, 24(5): pp. 57 –62,Nov.1990.
6. Mingchang Zhao, Jie Tian , Xun Zhu, Jian Xue, Zhanglin Cheng, Hua Zhao, The Design and Implementation of a C++ Toolkit for Integrated Medical Image Processing and Analyzing, Proc. of SPIE Medical Imaging 2004
7. Mason Woo, Jackie Neider, Tom Davis, et al. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2 (3rd Edition). Addison-Wesley Professional, 1999.
8. Dave Shreiner, OpenGL Architecture Review. OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2 (3rd Edition). Addison-Wesley Professional, 2000.
9. NVIDIA Corporation. White Paper: Using Vertex Buffer Objects (VBOs). NVIDIA Corporation, 2003

# 4 体绘制（Volume Rendering）的框架与实现

体绘制算法是可视化算法中非常重要的一种方法，体绘制的研究出现于上个世纪 80 年代末，传统的算法可以分为三个大类：图像空间（Image Space）的绘制算法、物体空间（Object Space）的绘制算法以及图像和物体空间混合（Hybrid）绘制算法。

## 4.1 体绘制算法综述

Ray Casting[1][2]是图像空间的经典绘制算法，它是从投影平面的每一个像素点发射出一条光线，穿过三维体数据场，并计算光的传输方程，得到最后的图像，这种方法得到的绘制效果比较好，并且可以很方便地实现一些插值算法和光线的提取终止，但是算法的速度比较慢，目前还不能达到实时的绘制目的。

Splatting[3][4]是物体空间的经典绘制算法，它的思路与Ray Casting完全不同，它按照物体顺序扫描每个体素，并将其投影到图像平面上，由于每个体素要影响到图像平面上的好几个像素，所以算法的名字被称为Splatting。Splatting算法尽管是按照物体顺序来访问三维数据集，可以充分利用现代CPU的Cache机制，但仍要牵涉到非常复杂的投影核心的计算，所以计算量也非常大，传统的算法仍然达不到实时绘制的目的。在文献[5]里面提出了一种优化的算法，可以将算法速度提高三到五倍，但是在目前的硬件水平上尚无法实现实时绘制。

Shear Warp[6][7]是综合了图像空间和物体空间优点的混合绘制算法，是目前为止基于软件实现的最快的体绘制算法。它首先将三维体数据集进行错切（Shear）变换，使其和图像平面平行，然后再进行投影计算，由于此时图像平面和数据集的特殊位置关系，使得投影计算量大大降低，最后通过一个两维的图像变形，得到最终的结果。由于此算法既可以很好地利用CPU的Cache来获得内存访问性能，又能够利用提前射线终止等图像空间算法的优点，并且所有的计算都被降到两维来完成，因此可以实现非常快的绘制速度。虽然在绘制速度上非常快，但这是建立在牺牲图像质量的前提基础之上的，在文献[8] 中提出了一些提高绘制质量的方法，同时尽可能减少算法在速度上的损失，在文献[9]中将Pre-Integration集成进原始的Shear Warp算法框架，进一步提高算法的绘制质量。

上面所述的三大类算法都是基于纯软件的算法，而近年随着普通显卡里面三维加速功能的逐渐强大，基于图形硬件的体绘制算法正逐渐称为主流。自从 1994 年Brian Cabral等人提出使用纹理映射[10]来加速体绘制，直至 1998 年，由于此算法只能运行在昂贵的图形工作站的显卡上，这期间研究人员一直没有认识到这个算法的重要性。随着三维游戏等娱乐市场的不断需求，普通PC机上装配的显卡的三维加速能力越来越强，同时纹理映射所需要的二维光栅的处理能力也越来越强，使用图形硬件来作体绘制也变得更为可行。在 1998 年的SigGraph文章[11]中，Westermann等人基于[10]的思想，并结合当前的显卡所提供的功能，实现了一个比较实用的基于硬件的体绘制算法，并且进一步完善[12]，加入了基于硬件加速的分类和光照计算。另外值得一提的是在 2000 年的Graphics Hardware年会的最佳论文[13]中，作者总结了前人所作过的工作，使用显卡中新提供的多纹理功能，在普通的PC机上实现了实时体绘制；在第二年（2001 年）的Graphics Hardware年会的最佳论文仍然是基于硬件的体绘制[14]，作者将Pre-Integration集成到基于硬件的体绘制算法中，从而显著地提高了绘制质量；同样，在 2003 年的Visualization年会上，最佳论文又是基于硬件的体绘制算法[15]，作者基于图形硬件实现了多维的传递函数的调节，从而可以更好地探索三维体数据的内部结构，尤其是边界信息。这三篇最佳论文带来了体绘制算法研究的春天，给原本比较沉闷的这一研究领域带来了新的活力。随着现在NVidia和ATI对三维显卡的不断更新换代，目前显卡已经具备了编程能力（被称为GPU），并且具有很强的计算能力。现在最活跃的研究领域是彻底在GPU上实现Ray Casting算法[16][17][18]，从而实现基于硬件的体绘制算法可以达到和基于软件的算法同样的绘制效果。

尽管基于硬件的体绘制算法有很多优越性，但是目前还是受到很多因素的限制，最大的一个问题就是有限的纹理存储容量，因此对于比较大的三维数据集来说，不能被加载到纹理内存中进行处理；另外一个问题就是计算精度，由于显卡的绘制引擎一般在内部使用定点运算，因此最终的精度会受到影响，不过随着更新显卡的出现，在不远的将来应该能实现全 IEEE 32 位浮点的绘制引擎，那样计算精度的问题就会得到缓解。

## 4.2 MITK 中的体绘制算法框架

由上面给出的综述，可以看出体绘制算法的多样性，另外，体绘制算法本身

的实现也是比较复杂的，里面还牵涉到各种光学参数和绘制参数的调节，因此要实现一个非常实用、灵活、可扩充的算法框架非常困难。MITK 中的体绘制算法框架是建立在 VTK 的体绘制算法框架之上，并进行了扩充和完善而得到的，它的目标是在一个统一的框架里面能够集成不同的体绘制算法，集成不同的参数调节算法，从而使得以后扩充新的算法非常容易。

在介绍体绘制的算法框架之前，先介绍MITK中的绘制模型，如 图 4-1 所示。View是一个Target的子类，用来将计算得到的图像或者三维图形显示在屏幕上，其维护着一个Model的数组m_Models，可以往里面添加Model。在View的绘制函数OnDraw里面，遍历每一个Model，并且调用每个Model的虚函数Render。Model在概念上代表着场景中一个待画的物体，可以是一个图像，也可以是三维图形，具体由Model的子类来实现，其中跟体绘制关系密切的就是VolumeModel。

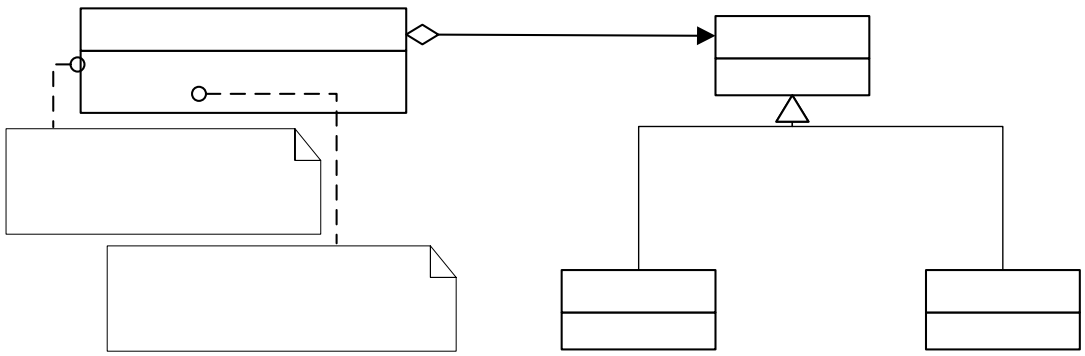


图 4-1 MITK 的绘制模型

VolumeModel的主要职责是实现父类里面规定的接口Render函数，但是体绘制有很多种不同的算法，又有很多参数需要调节，尤其是阻光度传递函数（Transfer Function），其选择的好坏直接影响着体绘制的结果。因此为了实现一个灵活的框架，VolumeModel的结构图如 图 4-2 所示。



m_Models

View

+AddModel(in aModel : Model)
+OnDraw()

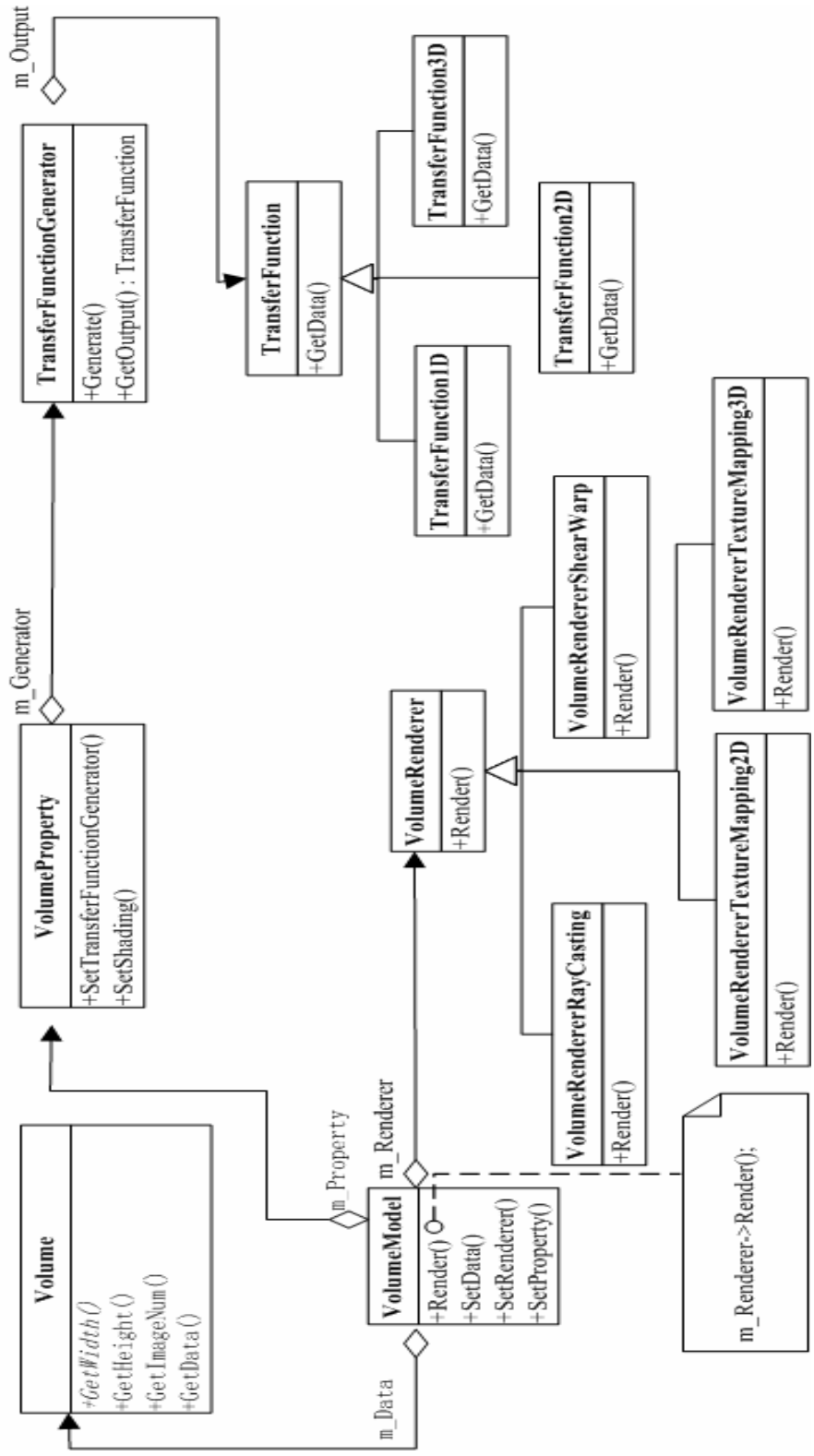



图 4-2 MITK 的体绘制框架

从图中可见，VolumeModel 拥有三个类成员：Volume、VolumeProperty 和 VolumeRenderer。Volume 主要是提供对体数据的访问，VolumeProperty 主要是提供对体绘制算法的一些参数的调整，而 VolumeRenderer 则负责实际的绘制工作。在 VolumeModel 的 Render 函数里面，绘制的工作被委托给了 VolumeRenderer 的 Render 函数，又由其子类来负责具体的实现。

VolumeRenderer 的各个子类代表了各种不同的体绘制算法，目前在 MITK 里面实现了上一节中介绍的 Ray Casting 算法，以及 Shear Warp 算法，并且还实现了硬件加速的基于纹理映射的体绘制算法。由于 MITK 中采用了这样灵活的绘制架构，以后往里面添加新的算法也非常方便，只用再增加一个 VolumeRenderer 的子类即可。

VolumeProperty除了提供体绘制算法所必须的光照计算的参数以外，最重要的一个目的就是提供阻光度的传递函数，为了提供足够的灵活性，VolumeProperty里面拥有一个TransferFunctionGenerator基类对象的指针，用来生成传递函数。而TransferFunctionGenerator的子类则负责实现各种不同的传递函数的生成算法，包括手工的和半自动的方法。TransferFunctionGenerator的输出为TransferFunction，而考虑到现在国际上已经开始研究多维的传递函数[19]，TransferFunction通过子类来实现一、二、三维的传递函数（TransferFunction1D，TransferFunction2D，TransferFunction3D）的支持，并且要求所有的Renderer也支持多维的传递函数。

## 4.3 体绘制算法在 MITK 中的实现

有了整个体绘制的框架以后，理解整个体绘制算法在 MITK 中的实现就比较容易了。这里为了完整起见，也涉及到 View 中绘制时的操作，但侧重点在整个体绘制流程的实现以及具体体绘制算法的实现。

### 4.3.1 View 中绘制操作的实现

因为整个绘制过程是由 View 得到重绘消息以后发起的，所以为了更好地理解整个绘制过程，这里简单给出 View 中绘制操作的实现。

由 图 4-1 可以得知，在View中有一个Model的列表，View负责维护这个列表（添加、删除等），并且在绘制的时候也是绘制这些Model，因此Model的角色

就相当于被绘制的物体加上一些属性。在View中跟维护Model和绘制相关的变量和函数声明如下面代码所示：

```
class mitkView : public mitkTarget
{
public:
    //维护模型列表的相关函数
    void AddModel(mitkModel *model);
    void RemoveModel(mitkModel *model);
    void RemoveAllModel(void);
    mitkModel* GetModel(int i);
    mitkList*  GetModels();
    int GetModelCount();

    //绘制操作
    virtual void OnDraw();

protected:
    mitkList *m_Models;  //存储模型列表
    mitkModel **m_VisibleModels;
    int m_NumberOfPropsRendered;
    unsigned char m_VisibleModelCount;
};
```

AddModel()函数往模型列表中添加一个模型，RemoveModel()函数将一个指定的模型从模型列表中删除，RemoveAllModel()函数清空模型列表，GetModel()函数得到指定索引位置的模型指针，GetModels()函数直接得到整个模型列表的指针，GetModelCount()函数得到模型列表中模型的个数，这些函数的实现如下所示：

```
void mitkView::AddModel(mitkModel *model)
{
    if(model)
    {
        model->AddReference();
```

```
        m_Models->Add(model);
    }
}

void mitkView::RemoveModel(mitkModel *model)
{
    if(model)
    {
        int indInModels = m_Models->Find(model);
        if(indInModels >= 0)
        {
            m_Models->Remove(indInModels);
            model->RemoveReference();
        }
    }
}

void mitkView::RemoveAllModel(void)
{
    mitkModel *aModel = NULL;
    for(m_Models->InitTraversal(); (aModel = (mitkModel*)
    m_Models->GetNextItem()); )
    {
        aModel->RemoveReference();
    }
    m_Models->Clear();
}

mitkModel* mitkView::GetModel(int i)
{
    return (mitkModel*) m_Models->Item(i);
}

mitkList*  mitkView::GetModels()
{
    return m_Models;
}

int mitkView::GetModelCount()
{
```

```
    return m_Models->Count();
}
```

OnDraw()函数是 View 在接收到重绘消息以后调用的，其循环遍历每个 Model，并且调用它们的虚函数 Render，来使其子类，也就是具体的模型来绘制自己。其实现代码如下所示：

```
void mitkView::OnDraw()
{
    // 初始化屏幕颜色
    glClearColor(m_BackColor[0], m_BackColor[1], m_BackColor[2],
    m_BackColor[3]);
    glClearDepth(1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    if(m_Models->Count() <= 0)    return;

    int     i;
    mitkModel  *aModel;

    m_VisibleModelCount = 0;
    m_VisibleModels = new mitkModel* [m_Models->Count()];
    // 循环遍历模型列表，找出可见模型，并将其存储在数组中
    for (m_Models->InitTraversal(); (aModel = (mitkModel*)
    m_Models->GetNextItem()); )
    {
        if(aModel->GetVisibility())
        {
            m_VisibleModels[m_VisibleModelCount++] = aModel;
        }
    }
    if(m_VisibleModelCount == 0)    return;

    // 循环遍历模型列表，先画不透明物体
    m_NumberOfPropsRendered = 0;
    for(i = 0; i < m_VisibleModelCount; i++)
    {
```

```
        if(m_VisibleModels[i]->IsOpaque())
        m_NumberOfPropsRendered += m_VisibleModels[i]->Render(this);
    }

    // 循环遍历模型列表，再画透明物体
    for(i = 0; i < m_VisibleModelCount; i++)
    {
        if(!m_VisibleModels[i]->IsOpaque())
        m_NumberOfPropsRendered += m_VisibleModels[i]->Render(this);
    }

    delete []m_VisibleModels;
    m_VisibleModels = NULL;
}
```

从上面的代码可以看出，实际的绘制工作被 View 委托给了各个 Model 的 Render 函数。并且为了实现表面绘制和体绘制、透明物体和不透明物体的同时显示，这里先循环遍历所有模型，画出不透明的物体，然后再循环遍历一次，画出透明的物体。VolumeModel 由于体绘制算法的本质特点，决定了它是一个透明的物体。

### 4.3.2 VolumeModel 的实现

VolumeModel 是 Model 的一个子类，它的最重要任务是实现 Model 所规定的 Render 接口，从而将自身绘制出来。为了能达到这一目的，VolumeModel 里面存储有三个对象指针 m_Data, m_Property, m_Renderer，分别指向 Volume、VolumeProperty、VolumeRenderer 的具体实例。VolumeModel 的声明如下所示：

```
class mitkVolumeModel : public mitkDataModel
{
public:
    // 设置新的体绘制算法
    void SetRenderer(mitkVolumeRenderer *renderer);

    // 得到当前的体绘制算法
    mitkVolumeRenderer* GetRenderer(void);
```

```
    // 设置一个新的体数据属性
    void SetProperty(mitkVolumeProperty *prop);

    // 得到当前的体数据属性
    mitkVolumeProperty* GetProperty(void);

    // 设置新的体数据
    void SetData(mitkVolume *data);

    // 得到当前的体数据
    mitkVolume* GetData(void);

    // 必须实现的接口，绘制自身
    virtual int Render(mitkView *view);

    // 得到当前模型的边界盒
    virtual float* GetBounds();

    // 得到当前模型是否为不透明。
// 对于体绘制来说，始终是透明的
    virtual bool IsOpaque() {return false;}

protected:
    mitkRCPtr<mitkVolume> m_Data;  //数据指针
    mitkRCPtr<mitkVolumeProperty> m_Property;  //属性指针
    mitkRCPtr<mitkVolumeRenderer> m_Renderer; //算法指针
};
```

m_Data, m_Property, m_Renderer 这三个成员变量都是通过智能指针 RCPtr 来实现的，它们可以被当做普通的指针使用，这里无需去理会 RCPtr 的实现细节。

通过一系列的 Get 和 Set 函数，VolumeModel 可以很方便地在运行时切换不同的数据，属性甚至绘制算法，这就使得整个框架非常灵活。这些 Get 和 Set 函数的实现都非常简单，如下面代码所示：

```
void mitkVolumeModel::SetRenderer(mitkVolumeRenderer *renderer)
```

```
{
    m_Renderer = renderer;
}

mitkVolumeRenderer* mitkVolumeModel::GetRenderer(void)
{
    if(m_Renderer == NULL)
    {
        m_Renderer = new mitkVolumeRendererRayCasting;
    }
    return m_Renderer;
}

void mitkVolumeModel::SetProperty(mitkVolumeProperty *prop)
{
    m_Property = prop;
}

mitkVolumeProperty* mitkVolumeModel::GetProperty(void)
{
    if(m_Property == NULL)
    {
        m_Property = new mitkVolumeProperty;
    }
    return m_Property;
}

void mitkVolumeModel::SetData(mitkVolume *data)
{
    m_Data = data;
    if(m_Data == NULL) return;

    m_Bounds[0] = 0.0f;
    m_Bounds[1] = m_Data->GetWidth() - 1.0f;
    m_Bounds[2] = 0.0f;
    m_Bounds[3] = m_Data->GetHeight() - 1.0f;
    m_Bounds[4] = 0.0f;
    m_Bounds[5] = m_Data->GetImageNum() - 1.0f;

    m_Origin[0] = (m_Bounds[0] + m_Bounds[1]) / 2.0f;
```

```
    m_Origin[1] = (m_Bounds[2] + m_Bounds[3]) / 2.0f;
    m_Origin[2] = (m_Bounds[4] + m_Bounds[5]) / 2.0f;
}

mitkVolume* mitkVolumeModel::GetData(void)
{
    return m_Data;
}
```

其中在 GetRenderer()函数中，如果当前还没有合适的绘制算法，则缺省地生成 Ray Casting 绘制算法的对象。在 SetData()函数中，还要设置好合适的边界和原点等信息，总之，上面的这些代码比较容易理解，这里就不过多解释了。

接下来比较重要的一个辅助函数是 GetBounds()函数，它负责返回当前模型的边界盒信息，并且是在世界坐标系中的边界盒。因此它需要调用父类中的函数 ModelToWorld()函数将模型坐标系中的坐标转换为世界坐标系中的坐标，最后返回世界坐标系中的边界盒，其实现代码如下：

```
float* mitkVolumeModel::GetBounds()
{
    if(m_Data == NULL || m_MatrixModified == 0)
        return m_Bounds;

    float minx, miny, minz, maxx, maxy, maxz;
    minx = 0.0f;
    maxx = m_Data->GetWidth() - 1.0f;
    miny = 0.0f;
    maxy = m_Data->GetHeight() - 1.0f;
    minz = 0.0f;
    maxz = m_Data->GetImageNum() - 1.0f;

    float modelPoint[4];
    float worldPoint[4];

    //First point
    modelPoint[0] = minx;
    modelPoint[1] = miny;
```

```
    modelPoint[2] = minz;
    modelPoint[3] = 1.0f;
    this->ModelToWorld(modelPoint, worldPoint);
    m_Bounds[0] = m_Bounds[1] = worldPoint[0];
    m_Bounds[2] = m_Bounds[3] = worldPoint[1];
    m_Bounds[4] = m_Bounds[5] = worldPoint[2];

    //Second point
    modelPoint[0] = maxx;
    modelPoint[1] = miny;
    modelPoint[2] = minz;
    modelPoint[3] = 1.0f;
    this->ModelToWorld(modelPoint, worldPoint);
    if(worldPoint[0] < m_Bounds[0])   m_Bounds[0] = worldPoint[0];
    if(worldPoint[0] > m_Bounds[1])   m_Bounds[1] = worldPoint[0];
    if(worldPoint[1] < m_Bounds[2])   m_Bounds[2] = worldPoint[1];
    if(worldPoint[1] > m_Bounds[3])   m_Bounds[3] = worldPoint[1];
    if(worldPoint[2] < m_Bounds[4])   m_Bounds[4] = worldPoint[2];
    if(worldPoint[2] > m_Bounds[5])   m_Bounds[5] = worldPoint[2];

    //Third point
    modelPoint[0] = maxx;
    modelPoint[1] = maxy;
    modelPoint[2] = minz;
    modelPoint[3] = 1.0f;
    this->ModelToWorld(modelPoint, worldPoint);
    if(worldPoint[0] < m_Bounds[0])   m_Bounds[0] = worldPoint[0];
    if(worldPoint[0] > m_Bounds[1])   m_Bounds[1] = worldPoint[0];
    if(worldPoint[1] < m_Bounds[2])   m_Bounds[2] = worldPoint[1];
    if(worldPoint[1] > m_Bounds[3])   m_Bounds[3] = worldPoint[1];
    if(worldPoint[2] < m_Bounds[4])   m_Bounds[4] = worldPoint[2];
    if(worldPoint[2] > m_Bounds[5])   m_Bounds[5] = worldPoint[2];

    //Fourth point
    modelPoint[0] = minx;
    modelPoint[1] = maxy;
    modelPoint[2] = minz;
    modelPoint[3] = 1.0f;
    this->ModelToWorld(modelPoint, worldPoint);
    if(worldPoint[0] < m_Bounds[0])   m_Bounds[0] = worldPoint[0];
```

```
    if(worldPoint[0] > m_Bounds[1])    m_Bounds[1] = worldPoint[0];
    if(worldPoint[1] < m_Bounds[2])    m_Bounds[2] = worldPoint[1];
    if(worldPoint[1] > m_Bounds[3])    m_Bounds[3] = worldPoint[1];
    if(worldPoint[2] < m_Bounds[4])    m_Bounds[4] = worldPoint[2];
    if(worldPoint[2] > m_Bounds[5])    m_Bounds[5] = worldPoint[2];

    //Fifth point
    modelPoint[0] = minx;
    modelPoint[1] = miny;
    modelPoint[2] = maxz;
    modelPoint[3] = 1.0f;
    this->ModelToWorld(modelPoint, worldPoint);
    if(worldPoint[0] < m_Bounds[0])    m_Bounds[0] = worldPoint[0];
    if(worldPoint[0] > m_Bounds[1])    m_Bounds[1] = worldPoint[0];
    if(worldPoint[1] < m_Bounds[2])    m_Bounds[2] = worldPoint[1];
    if(worldPoint[1] > m_Bounds[3])    m_Bounds[3] = worldPoint[1];
    if(worldPoint[2] < m_Bounds[4])    m_Bounds[4] = worldPoint[2];
    if(worldPoint[2] > m_Bounds[5])    m_Bounds[5] = worldPoint[2];

    //Sixth point
    modelPoint[0] = maxx;
    modelPoint[1] = miny;
    modelPoint[2] = maxz;
    modelPoint[3] = 1.0f;
    this->ModelToWorld(modelPoint, worldPoint);
    if(worldPoint[0] < m_Bounds[0])    m_Bounds[0] = worldPoint[0];
    if(worldPoint[0] > m_Bounds[1])    m_Bounds[1] = worldPoint[0];
    if(worldPoint[1] < m_Bounds[2])    m_Bounds[2] = worldPoint[1];
    if(worldPoint[1] > m_Bounds[3])    m_Bounds[3] = worldPoint[1];
    if(worldPoint[2] < m_Bounds[4])    m_Bounds[4] = worldPoint[2];
    if(worldPoint[2] > m_Bounds[5])    m_Bounds[5] = worldPoint[2];

    //Seventh point
    modelPoint[0] = maxx;
    modelPoint[1] = maxy;
    modelPoint[2] = maxz;
    modelPoint[3] = 1.0f;
    this->ModelToWorld(modelPoint, worldPoint);
    if(worldPoint[0] < m_Bounds[0])    m_Bounds[0] = worldPoint[0];
    if(worldPoint[0] > m_Bounds[1])    m_Bounds[1] = worldPoint[0];
```

```
    if(worldPoint[1] < m_Bounds[2])    m_Bounds[2] = worldPoint[1];
    if(worldPoint[1] > m_Bounds[3])    m_Bounds[3] = worldPoint[1];
    if(worldPoint[2] < m_Bounds[4])    m_Bounds[4] = worldPoint[2];
    if(worldPoint[2] > m_Bounds[5])    m_Bounds[5] = worldPoint[2];

    //Eighth point
    modelPoint[0] = minx;
    modelPoint[1] = maxy;
    modelPoint[2] = maxz;
    modelPoint[3] = 1.0f;
    this->ModelToWorld(modelPoint, worldPoint);
    if(worldPoint[0] < m_Bounds[0])    m_Bounds[0] = worldPoint[0];
    if(worldPoint[0] > m_Bounds[1])    m_Bounds[1] = worldPoint[0];
    if(worldPoint[1] < m_Bounds[2])    m_Bounds[2] = worldPoint[1];
    if(worldPoint[1] > m_Bounds[3])    m_Bounds[3] = worldPoint[1];
    if(worldPoint[2] < m_Bounds[4])    m_Bounds[4] = worldPoint[2];
    if(worldPoint[2] > m_Bounds[5])    m_Bounds[5] = worldPoint[2];

    return m_Bounds;
}
```

最后，该到最重要的 Render()函数了，也许你以为这是一个最复杂的函数了，可是这里考虑到框架的需求，并没有将复杂的逻辑都交给 VolumeModel 去处理，而是委托给 VolumeRenderer 去作绘制工作，这样，Render()的代码就比较简单了，如下所示：

```
int mitkVolumeModel::Render(mitkView *view)
{
    if(m_Data == NULL)
    {
        return 0;
    }

    this->GetRenderer()->Render(view, this);

    return 1;
}
```

### 4.3.3 VolumeProperty 的实现

上面已经提到过，VolumeModel 并没有大包大揽地将所有复杂的事务都自己处理，而是分散给 Volume、VolumeProperty 和 VolumeRenderer 来分别处理，它们各司其职，共同协作来完成复杂的任务。VolumeProperty 的主要任务就是提供体绘制各种参数的调节，其中分为两大类：传递函数的调节和光照效果参数的调节。对于传递函数来说，VolumeProperty 里面支持两种风格的传递函数：Classical 和 MultiDimensional，对于 Classical 风格的传递函数，又分为灰度值－颜色传递函数、灰度值－阻光度传递函数、梯度值－阻光度传递函数，而对于 MultiDimensional 风格的传递函数，只使用一个通用的类型 mitkTransferFunction 来表示即可，它可以表示一维、两维和三维的传递函数。对于光照效果参数来说，VolumeProperty 可以控制是否打开光照计算，以及分别控制各个光照分量的多少。VolumeProperty 的相关声明如下所示：

```
class mitkVolumeProperty : public mitkObject
{
public:
    // 设置灰度值－颜色传递函数
    void SetScalarColor(mitkColorTransferFunction *scFunction);

    // 设置灰度值－阻光度传递函数
    void SetScalarOpacity(mitkTransferFunction1D *soFunction);

    // 设置梯度值－阻光度传递函数
    void SetGradientOpacity(mitkTransferFunction1D *goFunction);

    // 得到灰度值－颜色传递函数
    mitkColorTransferFunction* GetScalarColor();

    // 得到灰度值－阻光度传递函数
    mitkTransferFunction1D* GetScalarOpacity();

    // 得到梯度值－阻光度传递函数
    mitkTransferFunction1D* GetGradientOpacity();

    // 设置梯度值－阻光度传递函数是否打开
```

```
void SetGradientOpacityCalculation(bool goEnable) {m_GOEnable =
goEnable;}

// 得到梯度值－阻光度传递函数是否打开
int  GetGradientOpacityCalculation(void) {return m_GOEnable;}

// 设置通用传递函数（可以是高维的）
void SetOpacityTransferFunction(mitkTransferFunction *moFunction);

// 得到通用传递函数（可以是高维的）
mitkTransferFunction* GetOpacityTransferFunction();

// TransferFunctionStyle 指定传递函数的类型
enum TransferFunctionStyle
{
    Classical,
    MultiDimensional
};

// 设置传递函数的类型
void SetTransferFunctionStyle(TransferFunctionStyle tfStyle)
{
    m_TransferFunctionStyle = tfStyle;
}

// 得到传递函数的类型
TransferFunctionStyle GetTransferFunctionStyle(void)
{
    return m_TransferFunctionStyle;
}

// 设置是否打开 Shading（光照效果）
void SetShade(bool shade)    {m_Shade = shade;}

// 得到当前的 Shading 效果是否打开
int  GetShade(void)         {return m_Shade;}

// 设置环境光系数
void SetAmbient(float value) {m_Ambient = value;}
```

```
    // 得到环境光系数
    float GetAmbient() {return m_Ambient;}

    // 设置散射光系数
    void SetDiffuse(float value) {m_Diffuse = value;}

    // 得到散射光系数
    float GetDiffuse() {return m_Diffuse;}

    // 设置高光系数
    void SetSpecular(float value) {m_Specular = value;}

    // 得到高光系数
    float GetSpecular() {return m_Specular;}

    // 设置高光指数
    void SetSpecularPower(float value) {m_SpecularPower = value;}

    // 得到高光指数
    float GetSpecularPower() {return m_SpecularPower;}

protected:
    mitkRCPtr<mitkColorTransferFunction>  m_ScalarColor;
    mitkRCPtr<mitkTransferFunction1D>     m_ScalarOpacity;
    mitkRCPtr<mitkTransferFunction1D>     m_GradientOpacity;

    mitkRCPtr<mitkTransferFunction> m_Opacity;

    bool  m_Shade;
    bool  m_GOEnable;
    float m_Ambient;
    float m_Diffuse;
    float m_Specular;
    float m_SpecularPower;
    TransferFunctionStyle m_TransferFunctionStyle;
};
```

这些函数都是设置一些状态，因此实现相对比较简单，有一些函数已经内联

实现在声明里面，其它的函数的实现如下所示：

```
void mitkVolumeProperty::SetScalarColor(mitkColorTransferFunction
*scFunction)
{
    m_ScalarColor = scFunction;
}

void mitkVolumeProperty::SetScalarOpacity(mitkTransferFunction1D
*soFunction)
{
    m_ScalarOpacity = soFunction;
}

void mitkVolumeProperty::SetGradientOpacity(mitkTransferFunction1D
*goFunction)
{
    m_GradientOpacity = goFunction;
}

mitkColorTransferFunction* mitkVolumeProperty::GetScalarColor()
{
    if(m_ScalarColor == NULL)
    {
        m_ScalarColor = new mitkColorTransferFunction;
    }
    return m_ScalarColor;
}

mitkTransferFunction1D* mitkVolumeProperty::GetScalarOpacity(void)
{
    if(m_ScalarOpacity == NULL)
    {
        m_ScalarOpacity = new mitkTransferFunction1D;
    }
    return m_ScalarOpacity;
}
```

```
mitkTransferFunction1D* mitkVolumeProperty::GetGradientOpacity(void)
{
    if(m_GradientOpacity == NULL)
    {
        m_GradientOpacity = new mitkTransferFunction1D;
    }
    return m_GradientOpacity;
}

void
mitkVolumeProperty::SetOpacityTransferFunction(mitkTransferFunction
*moFunction)
{
    m_Opacity = moFunction;
}

mitkTransferFunction*
mitkVolumeProperty::GetOpacityTransferFunction()
{
    if(m_Opacity == NULL)
    {
        m_Opacity = new mitkTransferFunction1D;
    }
    return m_Opacity;
}
```

### 4.3.4 VolumeRenderer 的实现

VolumeRenderer 的职责是实现 VolumeModel 委托给它的 Render 接口，也就是将模型绘制出来，为了允许其它算法的无缝集成，所以 VolumeRenderer 仍然只是个虚基类，各个具体的算法通过继承 VolumeRenderer 来实现 Render 接口。VolumeRenderer 为其子类提供了两个方面的能力：设置不同的分类方法，设置裁剪平面。在体绘制算法中，如果分类方法是 PreClassification，那么就是先使用阻光度传递函数进行分类，然后再插值计算；如果分类方法是 PostClassification，那么就是先进行插值计算，然后再根据阻光度传递函数进行分类。另外，VolumeRenderer 也支持平面裁剪，用一系列的平面去裁剪 Volume，从而可以得到切割效果，为了和 OpenGL 保持兼容，这里规定裁剪平面最多不能超过六个。因为分类方法和裁剪平面是所有体绘制算法都公用的东西，因此

将它们提在基类 VolumeRenderer 中实现，减少子类的负担，从而使得子类只用关注于实现自己的 Render 接口就行了。VolumeRenderer 中这两个部分的相关的函数和变量声明如下所示：

```
class mitkVolumeRenderer : public mitkRenderer
{
public:
    // 设置体绘制算法的分类方法
    void SetClassifyMethod(int classifyMethod)
    {
        m_ClassifyMethod = classifyMethod;
    }

    // 得到体绘制算法的分类方法
    int GetClassifyMethod()    {return m_ClassifyMethod;}

    // 设置体绘制算法的分类方法为 PreClassification
    void SetClassifyMethodToPreClassification()
    {
        m_ClassifyMethod = MITK_PRE_CLASSIFICATION;
    }

    // 设置体绘制算法的分类方法为 PostClassification
    void SetClassifyMethodToPostClassification()
    {
        m_ClassifyMethod = MITK_POST_CLASSIFICATION;
    }

    // 添加一个裁剪平面
    void AddClippingPlane(mitkPlane *plane);

    // 删除一个裁剪平面
    void RemoveClippingPlane(mitkPlane *plane);

    // 删除所有的裁剪平面，清空裁剪平面列表
    void RemoveAllClippingPlanes();

    // 得到裁剪平面列表
```

```
    mitkList* GetClippingPlanes(void)  {return m_ClippingPlanes;}

    // 得到指定位置的裁剪平面
    mitkPlane* GetClippingPlane(int planeIndex);

    // 得到裁剪平面的个数
    int  GetClippingPlaneCount(void)   {return
m_ClippingPlanes->Count();}

    // 允许裁剪
    void ClippingOn() {m_EnableClipping = true;}

    // 关闭裁剪
    void ClippingOff() {m_EnableClipping = false;}
    // 设置是否打开裁剪
    void SetClipping(bool enableClipping) {m_EnableClipping =
enableClipping;}

    // 得到当前裁剪是否打开
    bool GetClipping() {return m_EnableClipping;}

    // 纯虚函数，规定 Render 接口
    virtual void Render(mitkView *view, mitkVolumeModel *vol) = 0;


protected:
    int   m_ClassifyMethod;

    mitkList *m_ClippingPlanes;
    bool m_EnableClipping;
};
```

设置分类方法的相关函数都已经作为内联函数实现了，设置裁剪平面的函数的实现代码如下所示，因为代码相对简单，这里就不过多解释了。

```
void mitkVolumeRenderer::AddClippingPlane(mitkPlane *plane)
{
    m_ClippingPlanes->Add(plane);
```

```
}

void mitkVolumeRenderer::RemoveClippingPlane(mitkPlane *plane)
{
    m_ClippingPlanes->Remove(plane);
}

void mitkVolumeRenderer::RemoveAllClippingPlanes()
{
    m_ClippingPlanes->RemoveAll();
}

mitkPlane* mitkVolumeRenderer::GetClippingPlane(int planeIndex)
{
    return (mitkPlane*) m_ClippingPlanes->GetItem(planeIndex);
}
```

### 4.3.5 Ray Casting 算法的实现

VolumeRenderer 只是规定了一个抽象的 Render 接口，具体的体绘制算法由具体的子类从 VolumeRenderer 继承，并实现 Render()函数来实现算法。在 Render()函数中，综合利用 VolumeModel 中的各项信息，并充分利用 VolumeRenderer 父类提供的公用函数，来将体数据绘制出来。在 MITK 中目前实现了几种不同的体绘制算法，但是考虑到篇幅，这里只给出最常用也是最经典的 Ray Casting 算法的实现。

MITK 中 Ray Casting 算法在类 mitkVolumeRendererRayCasting 中实现，其相关类声明如下所示：

```
class mitkVolumeRendererRayCasting : public mitkVolumeRenderer
{
public:
    // 必须实现的接口
    virtual void Render(mitkView *view, mitkVolumeModel *vol);

    // 设置沿着射线方向的采样距离
    void  SetSampleDistance(float fVal) {m_SampleDistance = fVal;}
```

```
    // 得到沿着射线方向的采样距离
    float GetSampleDistance(){return m_SampleDistance;}

    // 设置在图像平面上的采样距离
    void SetImageSampleDistance(float fVal)
    {
        m_ImageSampleDistance = fVal;
    }

    // 得到在图像平面上的采样距离
    float GetImageSampleDistance() {return m_ImageSampleDistance;}

protected:
    mitkVolumeModel **m_RenderVolumeTable;
    mitkView       **m_RenderViewTable;

    // 一些变换矩阵
    mitkMatrix *m_PerspectiveMatrix;
    mitkMatrix *m_ViewToWorldMatrix;
    mitkMatrix *m_ViewToVoxelsMatrix;
    mitkMatrix *m_VoxelsToViewMatrix;
    mitkMatrix *m_WorldToVoxelsMatrix;
    mitkMatrix *m_VoxelsToWorldMatrix;

    int m_ImageViewportSize[2];
    int m_ImageMemorySize[2];
    int m_ImageInUseSize[2];
    int m_ImageOrigin[2];
    unsigned char *m_Image;

    int *m_RowBounds;
    int *m_OldRowBounds;

    int m_RenderTableSize;
    int m_RenderTableEntries;

    float m_SampleDistance;
    float m_ImageSampleDistance;
```

```
    float m_WorldSampleDistance;
    int   m_ScalarDataType;
    void  *m_ScalarDataPointer;
    mitkRay *m_Ray;

    void _updateShadingTables(mitkView *view, mitkVolumeModel *vol);

    void _renderTexture(mitkVolumeModel *vol, mitkView *view);
    int _computeRowBounds(mitkVolumeModel *vol, mitkView *view);
    int _clipRayAgainstVolume(mitkRay *rayInfo, float bounds[6] );
    int _clipRayAgainstClippingPlanes(int planeCount, float *planeEqus,
mitkRay *rayInfo);

    void _castRaysClipOff(mitkView *view, mitkVolumeModel *vol);
    void _castRaysClipOn(mitkView *view, mitkVolumeModel *vol);

    void _castRay(mitkRay *rayInfo);
};
```

其中，Render()函数是必须实现的接口，而 Ray Casting 算法自身也有一些参数可以设置，比如 SetSampleDistance()函数和 SetImageSampleDistance()函数可以用来控制最后图像的质量和绘制速度的折衷，采样距离设置的越大，最后图像的质量就越差，但是绘制速度会大大加快。另外，在 mitkVolumeRendererRayCasting 中还用到了很多变换矩阵，mitkMatrix 提供了对 4×4 矩阵的封装，可以简化很多操作。为了实现复杂的算法，mitkVolumeRendererRayCasting 中还有很多受保护的函数，来实现内部的结构化，每个都实现其具体的功能，它们将具体在下面被描述。

为了理清 Ray Casting 算法，还是让我们从 Render()函数来开始入手，其代码如下所示：

```
void mitkVolumeRendererRayCasting::Render(mitkView *view,
mitkVolumeModel *vol)
{
    mitkVolume *input = vol->GetData();
    if (input == NULL || input->GetData() == NULL)
```

```
{
    mitkErrorMessage("No Input!");
    return;
}

//得到体数据的属性
mitkVolumeProperty *volProperty = vol->GetProperty();

//得到相机
mitkCamera *cam = view->GetCamera();

//初始化 Ray 的一些属性
m_Ray->m_ScalarDataPointer = input->GetData();
m_Ray->m_ScalarDataType = input->GetDataType();
input->GetIncrements(m_Ray->m_DataIncrement);
input->GetDimensions(m_Ray->m_DataSize);
input->GetSpacings(m_Ray->m_DataSpacing);
vol->GetOrigin(m_Ray->m_DataOrigin);

m_Ray->m_ClassificationMethod = this->m_ClassifyMethod;

m_Ray->m_Color[0] = 0.0f;
m_Ray->m_Color[1] = 0.0f;
m_Ray->m_Color[2] = 0.0f;
m_Ray->m_Color[3] = 0.0f;

m_Ray->m_MaximizeOpacity = 1;

if(volProperty->GetTransferFunctionStyle() ==
mitkVolumeProperty::Classical)
{
    m_Ray->m_TF_Dimension = 1;
    m_Ray->m_TF_ScalarOpacityMaxX =
    volProperty->GetScalarOpacity()->GetMaxX();
    m_Ray->m_TF_ScalarOpacity =
    volProperty->GetScalarOpacity()->GetData();
    m_Ray->m_TF_ScalarColorRed =
    volProperty->GetScalarColor()->GetRData();
    m_Ray->m_TF_ScalarColorGreen =
    volProperty->GetScalarColor()->GetGData();
```

```
        m_Ray->m_TF_ScalarColorBlue =
        volProperty->GetScalarColor()->GetBData();
        if(volProperty->GetGradientOpacityCalculation())
        {
            m_Ray->m_TF_GradientOpacity =
            volProperty->GetGradientOpacity()->GetData();
        }
        else
        {
            m_Ray->m_TF_GradientOpacity = NULL;
        }
    }
    else
    {
        m_Ray->m_TF_Dimension =
        volProperty->GetOpacityTransferFunction()->GetDimension();
    }


    //计算 Shading Table
    this->_updateShadingTables(view, vol);


    //Voxel(Logical)----->Volume(Physical)---->World---->Camera---->
    //View

    //Voxel to Volume transformation =
    // translate(orgin)*scale(sapcings)*translate(-orgin)
    mitkMatrix tempMatrix;
    tempMatrix.ele[0]  = m_Ray->m_DataSpacing[0];
    tempMatrix.ele[1] = tempMatrix.ele[2] = tempMatrix.ele[3] =
    tempMatrix.ele[4] = 0.0f;
    tempMatrix.ele[5]  = m_Ray->m_DataSpacing[1];
    tempMatrix.ele[6] = tempMatrix.ele[7] = tempMatrix.ele[8] =
    tempMatrix.ele[9] = 0.0f;
    tempMatrix.ele[10] = m_Ray->m_DataSpacing[2];
    tempMatrix.ele[11] = 0.0f;
    tempMatrix.ele[12]  = m_Ray->m_DataOrigin[0]*(1.0f -
    m_Ray->m_DataSpacing[0]);
```

```
tempMatrix.ele[13]  = m_Ray->m_DataOrigin[1]*(1.0f -
m_Ray->m_DataSpacing[1]);
tempMatrix.ele[14]  = m_Ray->m_DataOrigin[2]*(1.0f -
m_Ray->m_DataSpacing[2]);
tempMatrix.ele[15] = 1.0f;

vol->GetModelMatrix(m_VoxelsToWorldMatrix);
*m_VoxelsToWorldMatrix *= tempMatrix;

//Volume to Voxel transformation =
//translate(orgin)*scale(1/sapcings)*translate(-orgin)
float invsx = 1.0f / m_Ray->m_DataSpacing[0];
float invsy = 1.0f / m_Ray->m_DataSpacing[1];
float invsz = 1.0f / m_Ray->m_DataSpacing[2];
tempMatrix.ele[0]  = invsx;
tempMatrix.ele[1] = tempMatrix.ele[2] = tempMatrix.ele[3] =
tempMatrix.ele[4] = 0.0f;
tempMatrix.ele[5]  = invsy;
tempMatrix.ele[6] = tempMatrix.ele[7] = tempMatrix.ele[8] =
tempMatrix.ele[9] = 0.0f;
tempMatrix.ele[10] = invsz;
tempMatrix.ele[11] = 0.0f;
tempMatrix.ele[12]  = m_Ray->m_DataOrigin[0]*(1.0f - invsx);
tempMatrix.ele[13]  = m_Ray->m_DataOrigin[1]*(1.0f - invsy);
tempMatrix.ele[14]  = m_Ray->m_DataOrigin[2]*(1.0f - invsz);
tempMatrix.ele[15] = 1.0f;

vol->GetInverseOfModelMatrix(m_WorldToVoxelsMatrix);
*m_WorldToVoxelsMatrix = tempMatrix * *m_WorldToVoxelsMatrix;

//World to View(Clipping Spacing) Transformation = Projection * View
cam->GetProjectionMatrix(m_PerspectiveMatrix);
cam->GetViewMatrix(&tempMatrix);
*m_PerspectiveMatrix *= tempMatrix;

//View(Clipping Spacing) to World Transformation = InverseOfView *
//InverseOfProjection
cam->GetInverseOfProjectionMatrix(&tempMatrix);
cam->GetInverseOfViewMatrix(m_ViewToWorldMatrix);
*m_ViewToWorldMatrix *= tempMatrix;
```

```
    //Voxels to View = World to View * Voxels to World
    *m_VoxelsToViewMatrix = *m_PerspectiveMatrix *
    *m_VoxelsToWorldMatrix;

    //View to Voxels = World to Voxels * View to World
    *m_ViewToVoxelsMatrix = *m_WorldToVoxelsMatrix *
    *m_ViewToWorldMatrix;

    //根据图像采样距离，计算最终图像的大小
    m_ImageViewportSize[0] = (int) ((float) view->GetWidth()  /
    m_ImageSampleDistance);
    m_ImageViewportSize[1] = (int) ((float) view->GetHeight() /
    m_ImageSampleDistance);

    if (this->_computeRowBounds(vol, view))
    {
        bool enableClipping = m_EnableClipping &
        (this->GetClippingPlaneCount() > 0);
        if(!enableClipping)
            _castRaysClipOff(view, vol);
        else
            _castRaysClipOn(view, vol);

        this->_renderTexture(vol, view);
    }
}
```

Render()函数的一开始，从 VolumeModel 取得体数据（Volume），数据属性（VolumeProperty），从 View 取得当前的相机，之后就要初始化射线 m_Ray 的一些公有的属性（也就是不随每条射线变化的属性）。这里必须提到的是，m_Ray 是一个类型为 mitkRay 的结构，记录了投射一条射线所必须的信息，其声明如下所示：

```
struct mitkRay
{
    float m_Color[4];
```

```
//射线起点
mitkVector *m_RayStart;
//射线终点
mitkVector *m_RayEnd;
//射线方向
mitkVector *m_RayDirection;
//射线在三个方向的增量
mitkVector *m_RayIncrement;

int m_NumberOfStepsToTake;
int m_NumberOfStepsTaken;

//辅助信息
int   m_ScalarDataType;
void  *m_ScalarDataPointer;
int   m_DataIncrement[3];
int   m_DataSize[3];
float m_DataSpacing[3];
float m_DataOrigin[3];

//所用的分类方法
int   m_ClassificationMethod;

int   m_MaximizeOpacity;
int   m_ScalarValue;

//计算光照效果需要用到的信息
int   m_Shading;
float *m_RedDiffuseShadingTable;
float *m_GreenDiffuseShadingTable;
float *m_BlueDiffuseShadingTable;
float *m_RedSpecularShadingTable;
float *m_GreenSpecularShadingTable;
float *m_BlueSpecularShadingTable;

unsigned short *m_EncodedNormals;
unsigned char  *m_GradientMagnitudes;

//传递函数相关信息
```

```
    int m_TF_Dimension;
    int m_TF_ScalarOpacityMaxX;
    int m_TF_ScalarOpacityMaxY;
    float *m_TF_ScalarOpacity;
    float *m_TF_GradientOpacity;
    float *m_TF_ScalarColorRed;
    float *m_TF_ScalarColorGreen;
    float *m_TF_ScalarColorBlue;
};
```

之后，计算跟光照相关的 Shading 表，然后是一些矩阵变换，这部分的代码虽然看起来很长，但是本身并不复杂，它们的功能也只是完成每一帧绘制的前期准备工作。在进行矩阵变换时，这里遵循的是 OpenGL 的坐标空间命名规则。_computeRowBounds()这个保护的成员函数的任务是计算在当前的变换矩阵下，体数据投影到图像平面所占的区域，这个区域的表示方式是使用一条一条的扫描线来记录的，每条扫描线记录它的起点和终点 x 坐标。_computeRowBounds()函数的实现代码如下，其每一步都给出了比较详细的注释：

```
int mitkVolumeRendererRayCasting::_computeRowBounds(mitkVolumeModel
*vol, mitkView *view)
{
    mitkVector voxelPoint;
    mitkVector viewPoint[8];
    unsigned char *ucptr;
    //需要注意的是，这里遵循 OpenGL 的标准，经过投影变换后，
    //x、y、z 的坐标范围都在-1.0~1.0 之间
    float minX, minY, maxX, maxY, minZ, maxZ;
    float bounds[6];
    int dim[3];
    int i, j;

    vol->GetData()->GetDimensions(dim);
    //逻辑坐标系（voxel）里面的包围盒
    bounds[0] = bounds[2] = bounds[4] = 0.0;
    bounds[1] = dim[0] - 1.0f;
    bounds[3] = dim[1] - 1.0f;
```

```
    bounds[5] = dim[2] - 1.0f;

    float camPos[3];
    float worldBounds[6];
    int insideFlag = 0;

    //得到世界坐标系（world）里面的包围盒
    vol->mitkModel::GetBounds(worldBounds);

    view->GetCamera()->GetPosition(camPos);
    if (camPos[0] >= worldBounds[0] &&
        camPos[0] <= worldBounds[1] &&
        camPos[1] >= worldBounds[2] &&
        camPos[1] <= worldBounds[3] &&
        camPos[2] >= worldBounds[4] &&
        camPos[2] <= worldBounds[5])
    {
        //相机在 Volume 内部
        insideFlag = 1;
    }

    //将包围盒投影到图像平面上，找出投影后所占的空间范围以及图像的大小
    //分别计算包围盒的八个顶点

    //如果相机在 Volume 内部，则认为 Volume 充满整个图像平面
    if (insideFlag)
    {
        minX = -1.0;
        maxX =  1.0;
        minY = -1.0;
        maxY =  1.0;
        minZ =  -1.0f;
        maxZ =  1.0f;
    }
    else
    {
        //First point---0 2 4
        voxelPoint.ele[0] = bounds[0];
        voxelPoint.ele[1] = bounds[2];
        voxelPoint.ele[2] = bounds[4];
```

```
voxelPoint.ele[3] = 1.0f;
viewPoint[0] = *m_VoxelsToViewMatrix * voxelPoint;
viewPoint[0] *= (1.0f/viewPoint[0].ele[3]);
minX = maxX = viewPoint[0].ele[0];
minY = maxY = viewPoint[0].ele[1];
minZ = maxZ = viewPoint[0].ele[2];

//Second point---1 2 4
voxelPoint.ele[0] = bounds[1];
viewPoint[1] = *m_VoxelsToViewMatrix * voxelPoint;
viewPoint[1] *= (1.0f/viewPoint[1].ele[3]);
minX = (viewPoint[1].ele[0] < minX) ? (viewPoint[1].ele[0]) :
(minX);
minY = (viewPoint[1].ele[1] < minY) ? (viewPoint[1].ele[1]) :
(minY);
maxX = (viewPoint[1].ele[0] > maxX) ? (viewPoint[1].ele[0]) :
(maxX);
maxY = (viewPoint[1].ele[1] > maxY) ? (viewPoint[1].ele[1]) :
(maxY);
minZ = (viewPoint[1].ele[2] < minZ) ? (viewPoint[1].ele[2]) :
(minZ);
maxZ = (viewPoint[1].ele[2] > maxZ) ? (viewPoint[1].ele[2]) :
(maxZ);

//Third point---1 3 4
voxelPoint.ele[1] = bounds[3];
viewPoint[2] = *m_VoxelsToViewMatrix * voxelPoint;
viewPoint[2] *= (1.0f/viewPoint[2].ele[3]);
minX = (viewPoint[2].ele[0] < minX) ? (viewPoint[2].ele[0]) :
(minX);
minY = (viewPoint[2].ele[1] < minY) ? (viewPoint[2].ele[1]) :
(minY);
maxX = (viewPoint[2].ele[0] > maxX) ? (viewPoint[2].ele[0]) :
(maxX);
maxY = (viewPoint[2].ele[1] > maxY) ? (viewPoint[2].ele[1]) :
(maxY);
minZ = (viewPoint[2].ele[2] < minZ) ? (viewPoint[2].ele[2]) :
(minZ);
maxZ = (viewPoint[2].ele[2] > maxZ) ? (viewPoint[2].ele[2]) :
(maxZ);
```

```
//Fourth point---0 3 4
voxelPoint.ele[0] = bounds[0];
viewPoint[3] = *m_VoxelsToViewMatrix * voxelPoint;
viewPoint[3] *= (1.0f/viewPoint[3].ele[3]);
minX = (viewPoint[3].ele[0] < minX) ? (viewPoint[3].ele[0]) :
(minX);
minY = (viewPoint[3].ele[1] < minY) ? (viewPoint[3].ele[1]) :
(minY);
maxX = (viewPoint[3].ele[0] > maxX) ? (viewPoint[3].ele[0]) :
(maxX);
maxY = (viewPoint[3].ele[1] > maxY) ? (viewPoint[3].ele[1]) :
(maxY);
minZ = (viewPoint[3].ele[2] < minZ) ? (viewPoint[3].ele[2]) :
(minZ);
maxZ = (viewPoint[3].ele[2] > maxZ) ? (viewPoint[3].ele[2]) :
(maxZ);

//Fifth point---0 3 5
voxelPoint.ele[2] = bounds[5];
viewPoint[4] = *m_VoxelsToViewMatrix * voxelPoint;
viewPoint[4] *= (1.0f/viewPoint[4].ele[3]);
minX = (viewPoint[4].ele[0] < minX) ? (viewPoint[4].ele[0]) :
(minX);
minY = (viewPoint[4].ele[1] < minY) ? (viewPoint[4].ele[1]) :
(minY);
maxX = (viewPoint[4].ele[0] > maxX) ? (viewPoint[4].ele[0]) :
(maxX);
maxY = (viewPoint[4].ele[1] > maxY) ? (viewPoint[4].ele[1]) :
(maxY);
minZ = (viewPoint[4].ele[2] < minZ) ? (viewPoint[4].ele[2]) :
(minZ);
maxZ = (viewPoint[4].ele[2] > maxZ) ? (viewPoint[4].ele[2]) :
(maxZ);

//Sixth point---1 3 5
voxelPoint.ele[0] = bounds[1];
viewPoint[5] = *m_VoxelsToViewMatrix * voxelPoint;
viewPoint[5] *= (1.0f/viewPoint[5].ele[3]);
```

```
minX = (viewPoint[5].ele[0] < minX) ? (viewPoint[5].ele[0]) :
(minX);
minY = (viewPoint[5].ele[1] < minY) ? (viewPoint[5].ele[1]) :
(minY);
maxX = (viewPoint[5].ele[0] > maxX) ? (viewPoint[5].ele[0]) :
(maxX);
maxY = (viewPoint[5].ele[1] > maxY) ? (viewPoint[5].ele[1]) :
(maxY);
minZ = (viewPoint[5].ele[2] < minZ) ? (viewPoint[5].ele[2]) :
(minZ);
maxZ = (viewPoint[5].ele[2] > maxZ) ? (viewPoint[5].ele[2]) :
(maxZ);

//Seventh point---1 2 5
voxelPoint.ele[1] = bounds[2];
viewPoint[6] = *m_VoxelsToViewMatrix * voxelPoint;
viewPoint[6] *= (1.0f/viewPoint[6].ele[3]);
minX = (viewPoint[6].ele[0] < minX) ? (viewPoint[6].ele[0]) :
(minX);
minY = (viewPoint[6].ele[1] < minY) ? (viewPoint[6].ele[1]) :
(minY);
maxX = (viewPoint[6].ele[0] > maxX) ? (viewPoint[6].ele[0]) :
(maxX);
maxY = (viewPoint[6].ele[1] > maxY) ? (viewPoint[6].ele[1]) :
(maxY);
minZ = (viewPoint[6].ele[2] < minZ) ? (viewPoint[6].ele[2]) :
(minZ);
maxZ = (viewPoint[6].ele[2] > maxZ) ? (viewPoint[6].ele[2]) :
(maxZ);

//Eighth point---0 2 5
voxelPoint.ele[0] = bounds[0];
viewPoint[7] = *m_VoxelsToViewMatrix * voxelPoint;
viewPoint[7] *= (1.0f/viewPoint[7].ele[3]);
minX = (viewPoint[7].ele[0] < minX) ? (viewPoint[7].ele[0]) :
(minX);
minY = (viewPoint[7].ele[1] < minY) ? (viewPoint[7].ele[1]) :
(minY);
maxX = (viewPoint[7].ele[0] > maxX) ? (viewPoint[7].ele[0]) :
(maxX);
```

```
        maxY = (viewPoint[7].ele[1] > maxY) ? (viewPoint[7].ele[1]) :
        (maxY);
        minZ = (viewPoint[7].ele[2] < minZ) ? (viewPoint[7].ele[2]) :
        (minZ);
        maxZ = (viewPoint[7].ele[2] > maxZ) ? (viewPoint[7].ele[2]) :
        (maxZ);
    }

    if (minZ < -0.9999f || maxZ > 0.9999f )
    {
        minX = -1.0f;
        maxX =  1.0f;
        minY = -1.0f;
        maxY =  1.0f;
        insideFlag = 1;
    }

    //转换成屏幕坐标，留有一定的裕量
    minX = (minX + 1.0f) * 0.5f * m_ImageViewportSize[0] - 2.0f;
    minY = (minY + 1.0f) * 0.5f * m_ImageViewportSize[1] - 2.0f;
    maxX = (maxX + 1.0f) * 0.5f * m_ImageViewportSize[0] + 2.0f;
    maxY = (maxY + 1.0f) * 0.5f * m_ImageViewportSize[1] + 2.0f;
    minZ = (minZ + 1.0f) * 0.5f;
    maxZ = (maxZ + 1.0f) * 0.5f;

    m_MinimumViewDistance = (minZ < 0.001f) ? (0.001f) : ((minZ > 0.999f) ?
    (0.999f) : (minZ));

    // 进行裁剪
    if ((minX < 0 && maxX < 0) ||
        (minY < 0 && maxY < 0) ||
        (minX > (m_ImageViewportSize[0] - 1) &&   maxX >
        (m_ImageViewportSize[0] - 1)) ||
        (minY > (m_ImageViewportSize[1] - 1) &&   maxY >
        (m_ImageViewportSize[1] - 1)))
    {
        return 0;
    }

    int oldImageMemorySize[2];
```

```
oldImageMemorySize[0] = m_ImageMemorySize[0];
oldImageMemorySize[1] = m_ImageMemorySize[1];

// Swap the row bounds
//如果是第一次进来，则 m_RowBounds 和 m_OldRowBounds 都为 NULL,要被重新分配
//如果前一帧比本帧小，则 m_RowBounds 和 m_OldRowBounds 也需要被重新分配
//如果前一帧比本帧大，但是又不是太大，则需要重用 m_Image，也仅在这种情况
//下，m_RowBounds 和 m_OldRowBounds 的前后帧关联才被用上
int *tmpptr;
tmpptr = m_RowBounds;
m_RowBounds = m_OldRowBounds;
m_OldRowBounds = tmpptr;

//  进行必要的裁剪
minX = (minX < 0.0f) ? (0.0f) : (minX);
minY = (minY < 0.0f) ? (0.0f) : (minY);
maxX = (maxX > m_ImageViewportSize[0] - 1.0f) ?
(m_ImageViewportSize[0] - 1.0f) : (maxX);
maxY = (maxY > m_ImageViewportSize[1] - 1.0f) ?
(m_ImageViewportSize[1] - 1.0f) : (maxY);

// 得到新图像的实际大小
m_ImageInUseSize[0] = (int) (maxX - minX + 1.0f);
m_ImageInUseSize[1] = (int) (maxY - minY + 1.0f);

// 计算新图像在内存里面的大小，需要是 2 的倍数
// 主要是为了后续使用 OpenGL 纹理绘制方便
m_ImageMemorySize[0] = 32;
m_ImageMemorySize[1] = 32;
while(m_ImageMemorySize[0] < m_ImageInUseSize[0])
{
    m_ImageMemorySize[0] *= 2;
}
while(m_ImageMemorySize[1] < m_ImageInUseSize[1])
{
    m_ImageMemorySize[1] *= 2;
}

m_ImageOrigin[0] = (int) minX;
m_ImageOrigin[1] = (int) minY;
```

```
// 如果旧的图像大小太大，则不复用前一帧的结果
if (oldImageMemorySize[0] > 2*m_ImageMemorySize[0] ||
    oldImageMemorySize[1] > 2*m_ImageMemorySize[1])
{
    oldImageMemorySize[0] = 0;
}

if (oldImageMemorySize[0] >= m_ImageMemorySize[0] &&
    oldImageMemorySize[1] >= m_ImageMemorySize[1])
{
    m_ImageMemorySize[0] = oldImageMemorySize[0];
    m_ImageMemorySize[1] = oldImageMemorySize[1];
}

// 清除前一帧的
if (!m_Image ||
    m_ImageMemorySize[0] > oldImageMemorySize[0] ||
    m_ImageMemorySize[1] > oldImageMemorySize[1])
{
    if (m_Image)
    {
        delete [] m_Image;
        delete [] m_RowBounds;
        delete [] m_OldRowBounds;
    }

    m_Image = new unsigned
    char[m_ImageMemorySize[0]*m_ImageMemorySize[1]*4];
    m_RowBounds    = new int [2*m_ImageMemorySize[1]];
    m_OldRowBounds = new int [2*m_ImageMemorySize[1]];


    for(i = 0; i < m_ImageMemorySize[1]; i++)
    {
        m_RowBounds[i*2]       = m_ImageMemorySize[0];
        m_RowBounds[i*2 + 1]    = -1;
        m_OldRowBounds[i*2]     = m_ImageMemorySize[0];
        m_OldRowBounds[i*2 + 1] = -1;
    }
```

```
        ucptr = m_Image;
        memset(ucptr, 0, m_ImageMemorySize[0]*m_ImageMemorySize[1]*4);

    }

    if (insideFlag)
    {
        for (i = 0; i < m_ImageInUseSize[1]; i++)
        {
            m_RowBounds[i*2] = 0;
            m_RowBounds[i*2 + 1] = m_ImageInUseSize[0] - 1;
        }
    }
    else
    {
        // 计算每条扫描线的起点和终点
        float lines[12][4];
        float x1, y1, x2, y2;
        int xlow, xhigh;
        int lineIndex[12][2] = {{0,1}, {2,3}, {4,5}, {6,7},
                                {0,3}, {2,1} ,{6,5}, {4,7},
                                {0,7}, {6,1}, {2,5}, {4,3}};

        for(i = 0; i < 8; i++)
        {
            viewPoint[i].ele[0] = (viewPoint[i].ele[0] +
            1.0f)*0.5f*m_ImageViewportSize[0] - m_ImageOrigin[0];
            viewPoint[i].ele[1] = (viewPoint[i].ele[1] +
            1.0f)*0.5f*m_ImageViewportSize[1] - m_ImageOrigin[1];
        }
        for (i = 0; i < 12; i++)
        {
            x1 = viewPoint[lineIndex[i][0]].ele[0];
            y1 = viewPoint[lineIndex[i][0]].ele[1];
            x2 = viewPoint[lineIndex[i][1]].ele[0];
            y2 = viewPoint[lineIndex[i][1]].ele[1];

            if ( y1 < y2 )
            {
```

```
            lines[i][0] = x1;
            lines[i][1] = y1;
            lines[i][2] = x2;
            lines[i][3] = y2;
        }
        else
        {
            lines[i][0] = x2;
            lines[i][1] = y2;
            lines[i][2] = x1;
            lines[i][3] = y1;
        }
    }

    for(j = 0; j < m_ImageInUseSize[1]; j++)
    {
        m_RowBounds[j*2]    = m_ImageMemorySize[0];
        m_RowBounds[j*2 + 1] = -1;

        for(i = 0; i < 12; i++)
        {
            if(j >= lines[i][1] && j <= lines[i][3] &&
               (lines[i][1] != lines[i][3]))
            {
                x1 = lines[i][0] + ((float)(j) - lines[i][1]) /
                (lines[i][3] - lines[i][1]) * (lines[i][2] -
                lines[i][0]);

                xlow  = (int) (x1 + 1.5f);
                xhigh = (int) (x1 - 1.0f);

                xlow = (xlow < 0.0f) ? (0.0f) : (xlow);
                xlow = (xlow > m_ImageInUseSize[0] - 1.0f) ?
                (m_ImageInUseSize[0] - 1.0f) : (xlow);

                xhigh = (xhigh < 0.0f) ? (0.0f) : (xhigh);
                xhigh = (xhigh > m_ImageInUseSize[0] - 1.0f) ?
                (m_ImageInUseSize[0] - 1.0f) : (xhigh);

                if (xlow < m_RowBounds[j*2])
```

```
                {
                    m_RowBounds[j*2] = xlow;
                }
                if (xhigh > m_RowBounds[j*2 + 1])
                {
                    m_RowBounds[j*2 + 1] = xhigh;
                }
            }
        }

        if(m_RowBounds[j*2] == m_RowBounds[j*2 + 1])
        {
            m_RowBounds[j*2]    = m_ImageMemorySize[0];
            m_RowBounds[j*2 + 1] = -1;
        }
    }
}

// 区域之外的填充成－1
for (j = m_ImageInUseSize[1]; j < m_ImageMemorySize[1]; j++)
{
    m_RowBounds[j*2]    = m_ImageMemorySize[0];
    m_RowBounds[j*2 + 1] = -1;
}

for(j = 0; j < m_ImageMemorySize[1]; j++)
{
    if(m_RowBounds[j*2+1] < m_OldRowBounds[j*2] ||
       m_RowBounds[j*2]   > m_OldRowBounds[j*2+1])
    {
        ucptr = m_Image + 4*(j*m_ImageMemorySize[0] +
        m_OldRowBounds[j*2]);
        if((m_OldRowBounds[j*2 + 1] - m_OldRowBounds[j*2]) >= 0)
            memset(ucptr, 0, 4*(m_OldRowBounds[j*2 + 1] -
            m_OldRowBounds[j*2] + 1));
    }
    else
    {
        if((m_RowBounds[j*2] - m_OldRowBounds[j*2]) > 0)
        {
```

```
                ucptr = m_Image + 4*(j*m_ImageMemorySize[0] +
                m_OldRowBounds[j*2]);
                memset(ucptr, 0, 4*(m_RowBounds[j*2] -
                m_OldRowBounds[j*2]));
            }

            if((m_OldRowBounds[j*2+1] - m_RowBounds[j*2+1]) > 0)
            {
                ucptr = m_Image + 4*(j*m_ImageMemorySize[0] +
                m_RowBounds[j*2 + 1] + 1);
                memset(ucptr, 0, 4*(m_OldRowBounds[j*2+1] -
                m_RowBounds[j*2+1]));
            }
        }
    }

    return 1;
}
```

在计算完投影区域以后，该完成的准备工作已经都差不多了，需要注意的是，到目前为止算法最复杂的多层循环还没有开始，前面的都是一些循环外层的初始工作。在进入主要循环之前，程序还分两种情况：支持平面裁剪和不支持平面裁剪，如果设置了支持平面裁剪并且当前存在裁剪平面，那么程序调用_castRaysClipOn()函数来进行实际的光线投射计算工作，如果没有打开平面裁剪，那么程序调用_castRaysClipOff()函数来进行实际的光线投射计算工作。因为_castRaysClipOn()比_castRaysClipOff()更复杂，这里为了避免大量的代码段出现，只给出_castRaysClipOn()的实现，如下所示：

```
void mitkVolumeRendererRayCasting::_castRaysClipOn(mitkView *view,
mitkVolumeModel *vol)
{
    int i, j;
    unsigned char *ucptr;
    float cameraThickness = view->GetCamera()->GetThickness();

    //得到射线的必要信息
```

```
mitkVector *rayStart     = m_Ray->m_RayStart;
mitkVector *rayEnd       = m_Ray->m_RayEnd;
mitkVector *rayDirection = m_Ray->m_RayDirection;
mitkVector *rayStep      = m_Ray->m_RayIncrement;

//初始化裁剪平面（Clipping Plane）
//得到裁剪平面（Clipping Plane） 的 Ax + By + Cz + D = 0 的表达式
mitkPlane *onePlane;
int     count;
float    *clippingPlane, *tempPlanePointer;
float    *planeOrigin;
count = this->GetClippingPlaneCount();
clippingPlane = new float[4*count];    //必须在用完时释放
tempPlanePointer = clippingPlane;
for(m_ClippingPlanes->InitTraversal(); (onePlane = (mitkPlane*)
m_ClippingPlanes->GetNextItem()); )
{
    //此处的 Clipping Plane 是在 Voxel Space 中指定的，所以无需转换
    onePlane->GetNormal(tempPlanePointer[0], tempPlanePointer[1],
    tempPlanePointer[2]);
    planeOrigin = (float*) onePlane->GetOrigin()->ele;
    tempPlanePointer[3] = -(tempPlanePointer[0]*planeOrigin[0] +
    tempPlanePointer[1]*planeOrigin[1] +
    tempPlanePointer[2]*planeOrigin[2]);
    tempPlanePointer += 4;
}

float norm;
mitkVector viewRay(0.0f, 0.0f, -1.0f);
mitkVector rayCenter;
float absStep[3];
mitkVector voxelPoint;

// 在 View（NDC）坐标系中的近平面的中心点
// 转换到 Voxel（Logical)坐标系中
*rayStart = *m_ViewToVoxelsMatrix * viewRay;
*rayStart *= (1.0f / rayStart->ele[3]);

// 在 View（NDC）坐标系中的远平面的中心点
// 转换到 Voxel（Logical)坐标系中
```

```
    viewRay.ele[2] = 1.0f;
    voxelPoint = *m_ViewToVoxelsMatrix * viewRay;
    voxelPoint *= (1.0f / voxelPoint.ele[3]);

    // 得到中心射线的方向向量
    rayCenter = voxelPoint - *rayStart;

    // 标准化中心射线的方向向量
    rayCenter *= (1.0f / cameraThickness);

    // centerScale = Len(rayCenter) / (|far| - |near|)
    float centerScale = rayCenter.Length();
    rayCenter.Normalize();

    float bounds[6];
    int dim[3];
    float val;

    vol->GetData()->GetDimensions(dim);
    bounds[0] = bounds[2] = bounds[4] = 0.0;
    bounds[1] = dim[0]-1;
    bounds[3] = dim[1]-1;
    bounds[5] = dim[2]-1;

    float offsetX = 1.0f / (float) (m_ImageViewportSize[0]);
    float offsetY = 1.0f / (float) (m_ImageViewportSize[1]);

    //主要循环开始，遍历投影图像的每一条扫描线
    for (j = 0; j < m_ImageInUseSize[1]; j++)
    {
        ucptr = m_Image + 4*(j*m_ImageMemorySize[0] + m_RowBounds[j*2]);

        //计算 NDC 空间里面的 y 坐标
        viewRay.ele[1] = (((float)(j + m_ImageOrigin[1])) / (float)
        m_ImageViewportSize[1]) * 2.0f - 1.0f + offsetY;

        //内层循环开始，遍历扫描线上起点和终点之间的每一个点
        for(i = m_RowBounds[j*2]; i <= m_RowBounds[j*2+1]; i++)
        {
            ucptr[0] = 0;
```

```
ucptr[1] = 0;
ucptr[2] = 0;
ucptr[3] = 0;
//计算 NDC 空间里面的 x 坐标
viewRay.ele[0] = (((float)(i + m_ImageOrigin[0])) / (float)
m_ImageViewportSize[0]) * 2.0f - 1.0f + offsetX;

//在 NDC 空间中，每条射线都是平行的，并且近平面就是图像平面。
//射线起始于图像平面，终止于远平面，或者终止于最近的不透明
//物体前（由 zbuffer 决定）

//先算起点坐标，将其转换至 Voxel 空间中
viewRay.ele[2] = -1.0f;
*rayStart = *m_ViewToVoxelsMatrix * viewRay;
*rayStart *= (1.0f / rayStart->ele[3]);

//再算终点坐标，将其转换至 Voxel 空间中
viewRay.ele[2] = (m_ZBuffer) ? (this->_getZBufferValue(i,j)) :
(1.0f);
*rayEnd = *m_ViewToVoxelsMatrix * viewRay;

*rayEnd *= (1.0f / rayEnd->ele[3]);

//得到射线的方向
*rayDirection = *rayEnd - *rayStart;

//本射线相对于 Volume 和 Clipping Plane 进行裁剪
if(this->_clipRayAgainstVolume(m_Ray, bounds) &&
this->_clipRayAgainstClippingPlanes(count, clippingPlane,
m_Ray))
{
    //重新计算射线方向
    *rayDirection = *rayEnd - *rayStart;


    //归一化
    *rayStep = *rayDirection;
    rayStep->Normalize();
    val =  *rayStep * rayCenter;  //Cos(theta)
    norm = (val != 0.0f) ? (1.0f / val) : (1.0f);
```

```
//此处应该注意的是，sampleDistance / (|far| - |near|) 是 NDC
//空间的采样距离，它保证在 NDC 空间是等间距采样的，但是在 Voxel
//空间，每条射线的采样距离就不相等了。在 Voxel 空间的采样距离
//为:0.5 * Len(rayCenter) * sampleDistance / (|far| -
//|near|),因为 centerScale = Len(rayCenter) / (|far| -
//|near|)，所以上式为：0.5 * sampleDistance * centerScale.
//这是在中心线上的采样距离，本算法中采用平行于图像平面的平面去采
//样，而没有用半球面上的采样，所以最终的采样距离为：
//0.5 * sampleDistance * centerScale / cos(theta)，也即
//0.5 * sampleDistance * centerScale * norm
*rayStep *= 0.5f * norm * m_SampleDistance * centerScale;

absStep[0] = (rayStep->ele[0] < 0.0f) ? (-rayStep->ele[0]) :
(rayStep->ele[0]);
absStep[1] = (rayStep->ele[1] < 0.0f) ? (-rayStep->ele[1]) :
(rayStep->ele[1]);
absStep[2] = (rayStep->ele[2] < 0.0f) ? (-rayStep->ele[2]) :
(rayStep->ele[2]);

if ( absStep[0] >= absStep[1] && absStep[0] >= absStep[2] )
{
    m_Ray->m_NumberOfStepsToTake = (int) ((rayEnd->ele[0]
    - rayStart->ele[0]) / rayStep->ele[0]);
}
else if ( absStep[1] >= absStep[2] && absStep[1] >=
absStep[0] )
{
    m_Ray->m_NumberOfStepsToTake = (int) ((rayEnd->ele[1]
    - rayStart->ele[1]) / rayStep->ele[1]);
}
else
{
    m_Ray->m_NumberOfStepsToTake = (int) ((rayEnd->ele[2]
    - rayStart->ele[2]) / rayStep->ele[2]);
}

//最内层循环在此函数内完成
this->_castRay(m_Ray);
```

```
                //得到这条射线最后的颜色
                if(m_Ray->m_Color[3] > 0.0f)
                {
                    val = (m_Ray->m_Color[0] / m_Ray->m_Color[3]) *
                    255.0f;
                    val = (val > 255.0f) ? (255.0f) : (val);
                    val = (val <  0.0f) ? (  0.0f) : (val);
                    ucptr[0] = (unsigned char) (val);

                    val = (m_Ray->m_Color[1] / m_Ray->m_Color[3]) *
                    255.0f;
                    val = (val > 255.0f) ? (255.0f) : (val);
                    val = (val <  0.0f) ? (  0.0f) : (val);
                    ucptr[1] = (unsigned char) (val);

                    val = (m_Ray->m_Color[2] / m_Ray->m_Color[3]) *
                    255.0f;
                    val = (val > 255.0f) ? (255.0f) : (val);
                    val = (val <  0.0f) ? (  0.0f) : (val);
                    ucptr[2] = (unsigned char) (val);

                    val = m_Ray->m_Color[3] * 255.0f;
                    val = (val > 255.0f) ? (255.0f) : (val);
                    val = (val <  0.0f) ? (  0.0f) : (val);
                    ucptr[3] = (unsigned char) (val);
                }
            }
            ucptr += 4;
        }
    }
    delete []clippingPlane;
}
```

可以看到，在上面的代码中主要是一个双层循环，外层循环遍历我们在_computeRowBounds()函数中计算出来的投影区域的每一条扫描线，而内层循环则遍历这条扫描线上从起点到终点的每一个像素点，对于每一个像素点，计算出从其投射的光线的起始点和终止点坐标、射线方向等信息（在 Voxel 空间），然后再调用_castRay()函数来完成最后的计算工作。

_castRay()函数内部实际上也是一个循环，它对传进来的射线按照一定的步长采样，这个循环就是要遍历每一个采样点，所以实际上 Ray Casting 算法的本质是个三重循环。_castRay()函数处于最内层循环，它接收一条射线信息，计算出这条射线穿过 Volume 后累积得到的颜色值，其代码实现如下所示：

```
void mitkVolumeRendererRayCasting::_castRay(mitkRay *rayInfo)
{
    void *data_ptr = rayInfo->m_ScalarDataPointer;

    if(rayInfo->m_Shading == 0)  //没有光照效果（Shading）
    {
        // 判断数据类型，并分发给模板函数去处理
        switch(rayInfo->m_ScalarDataType)
        {
            case MITK_UNSIGNED_CHAR:
                t_CastRayUnshaded((unsigned char *)data_ptr, rayInfo);
            break;
            case MITK_UNSIGNED_SHORT:
                t_CastRayUnshaded((unsigned short *)data_ptr, rayInfo);
            break;
        }
    }
    else  //有光照效果（Shading）
    {
        // 判断数据类型，并分发给模板函数去处理
        switch(rayInfo->m_ScalarDataType)
        {
            case MITK_UNSIGNED_CHAR:
                t_CastRayShaded((unsigned char *)data_ptr, rayInfo);
            break;
            case MITK_UNSIGNED_SHORT:
                t_CastRayShaded((unsigned short *)data_ptr, rayInfo);
            break;
        }
    }
}
```

在_castRay()函数中，要处理有光照效果和无光照效果两种情况，并且还要处理不同的数据类型。这里使用模板函数来处理不同的数据类型，避免代码的重复：模板函数 t_CastRayUnshaded()用来计算无光照效果下的结果，而模板函数 t_CastRayShaded()用来计算有光照效果下的结果，它们都接收一个参数化的数据指针，和一个 mitkRay 类型的指针（提供射线信息）。为了避免重复，这里只给出比较复杂的有光照效果计算的函数 t_CastRayShaded()的实现，其代码如下所示：

```
//要计算光照效果（Shading），必须付出的额外的内存代价就是 Normal 的存储空间
//这里将 Normal 压缩编码成 16bits
//并且 Shading_Table 可以直接接收一个 16bits 编码的 Normal 作为输入，
//输出对应的 Diffuse 和 Specular 光照颜色。
//这里最终颜色的计算公式如下：
//FinalColor = Diffuse * Color + Specular
template <class T>
static void t_CastRayShaded(T *data_ptr, mitkRay *rayInfo)

{
    //局部变量定义
    int             value = 0;
    unsigned char   *grad_mag_ptr = NULL;
    float           accum_red_intensity;
    float           accum_green_intensity;
    float           accum_blue_intensity;
    float           remaining_opacity;
    float           opacity = 0.0f;
    float           opacity_mul;
    float           *diff_shade_red;
    float           *diff_shade_green;
    float           *diff_shade_blue;
    float           *spec_shade_red;
    float           *spec_shade_green;
    float           *spec_shade_blue;
    unsigned short  *encoded_normals;
    int             encoded_normal;
    int             loop;
    int             xinc, yinc, zinc;
```

```
int           voxel[3];
float         ray_position[3];
int           tf_dim;
float         *SOTF;
float         *SCTF_R;
float         *SCTF_G;
float         *SCTF_B;
float         *GOTF;
int           offset = 0;
int           steps_this_ray = 0;
int           num_steps;
float         *ray_start, *ray_increment;

//将必须的信息拷贝到局部变量中，为编译器优化提供条件
num_steps = rayInfo->m_NumberOfStepsToTake;
ray_start = rayInfo->m_RayStart->ele;
ray_increment = rayInfo->m_RayIncrement->ele;

SOTF =  rayInfo->m_TF_ScalarOpacity;
GOTF =  rayInfo->m_TF_GradientOpacity;
SCTF_R = rayInfo->m_TF_ScalarColorRed;
SCTF_G = rayInfo->m_TF_ScalarColorGreen;
SCTF_B = rayInfo->m_TF_ScalarColorBlue;
tf_dim = rayInfo->m_TF_Dimension;

diff_shade_red   = rayInfo->m_RedDiffuseShadingTable;
diff_shade_green = rayInfo->m_GreenDiffuseShadingTable;
diff_shade_blue  = rayInfo->m_BlueDiffuseShadingTable;
spec_shade_red   = rayInfo->m_RedSpecularShadingTable;
spec_shade_green = rayInfo->m_GreenSpecularShadingTable;
spec_shade_blue  = rayInfo->m_BlueSpecularShadingTable;

encoded_normals = rayInfo->m_EncodedNormals;

xinc = rayInfo->m_DataIncrement[0];
yinc = rayInfo->m_DataIncrement[1];
zinc = rayInfo->m_DataIncrement[2];

ray_position[0] = ray_start[0];
ray_position[1] = ray_start[1];
```

```
    ray_position[2] = ray_start[2];
    voxel[0] = mitkRoundFuncMacro( ray_position[0] );
    voxel[1] = mitkRoundFuncMacro( ray_position[1] );
    voxel[2] = mitkRoundFuncMacro( ray_position[2] );

    accum_red_intensity    = 0.0f;
    accum_green_intensity   = 0.0f;
    accum_blue_intensity    = 0.0f;
    remaining_opacity      = 1.0f;

    //如果需要计算 Gradient-Opacity，则得到 Volume 的梯度场
    if(GOTF)
    {
        grad_mag_ptr = rayInfo->m_GradientMagnitudes;
    }


    // 循环遍历本条射线，最内层循环开始
    for(loop = 0; loop < num_steps && remaining_opacity >
    MITK_REMAINING_OPACITY; loop++ )
    {
        steps_this_ray++;

        //得到 Scalar Value
        offset = voxel[2] * zinc + voxel[1] * yinc + voxel[0] * xinc;
        value = data_ptr[offset];

        //计算 Opacity
        if(tf_dim == 1) //一维的 TF
        {
            opacity = SOTF[value];

            if(opacity && grad_mag_ptr)
                opacity *= GOTF[grad_mag_ptr[offset]];
        }
        else if(tf_dim == 2) //二维的 TF
        {

        }
        else //三维的 TF
```

```
        {

        }

        //累加颜色值，由前向后
        opacity_mul = opacity * remaining_opacity;
        encoded_normal = encoded_normals[offset];
        accum_red_intensity   += (opacity_mul *
        (diff_shade_red[encoded_normal] * SCTF_R[value] +
        spec_shade_red[encoded_normal]));
        accum_green_intensity   += (opacity_mul *
        (diff_shade_green[encoded_normal] * SCTF_G[value] +
        spec_shade_green[encoded_normal]));
        accum_blue_intensity   += (opacity_mul *
        (diff_shade_blue[encoded_normal] * SCTF_B[value] +
        spec_shade_blue[encoded_normal]));
        remaining_opacity *= (1.0 - opacity);

        //计算下一个 Voxel 的位置
        ray_position[0] += ray_increment[0];
        ray_position[1] += ray_increment[1];
        ray_position[2] += ray_increment[2];
        voxel[0] = mitkRoundFuncMacro( ray_position[0] );
        voxel[1] = mitkRoundFuncMacro( ray_position[1] );
        voxel[2] = mitkRoundFuncMacro( ray_position[2] );
    }

    if(accum_red_intensity > 1.0f)
    {
        accum_red_intensity = 1.0f;
    }
    if(accum_green_intensity > 1.0f)
    {
        accum_green_intensity = 1.0f;
    }
    if(accum_blue_intensity > 1.0f)
    {
        accum_blue_intensity = 1.0f;
    }
    if(remaining_opacity < MITK_REMAINING_OPACITY)
```

```
    {
        remaining_opacity = 0.0f;
    }

    //储存颜色信息
    rayInfo->m_Color[0] = accum_red_intensity;
    rayInfo->m_Color[1] = accum_green_intensity;
    rayInfo->m_Color[2] = accum_blue_intensity;
    rayInfo->m_Color[3] = 1.0 - remaining_opacity;
    rayInfo->m_NumberOfStepsTaken = steps_this_ray;
}
```

至此为止，整个 Ray Casting 绘制算法的主要步骤的实现细节都已经给出了，还有一些其它的辅助函数的实现这里没有详细给出，不过都是一些枝节性的东西。深入理解了 Ray Casting 算法的实现以后，对于其它类型的体绘制算法也可以很快地理解并给出自己的实现。

## 4.4 小结

本节给出了 MITK 中体绘制算法的框架和实现，在第一部分，简要介绍了体绘制算法的发展历史，以及体绘制算法的分类；在第二部分，给出了 MITK 中体绘制算法的整体框架，目的是为了得到灵活性、易于扩充性；在第三部分，以实例代码的形式给出了 MITK 中体绘制算法框架各个部分的详细实现，并且用 Ray Casting 算法为例，给出了此算法在 MITK 框架里面的详细实现。

读者在阅读完本章以后，应该对体绘制的整个过程有一个比较深入的理解，并且对算法的实现细节也有比较好的掌握。为了使复杂的体绘制算法能够被容易地把握，本章不吝笔墨，给出了大量的实现细节，包括实际的代码，因为我们相信最终代码最能说明问题。希望读者能够反复地阅读这些代码，彻底理解体绘制的工作流程，这样可以在自己的工作中，举一反三，设计出自己的体绘制算法，丰富可用的算法仓库。

参考文献

1. M. Levoy, Display of surfaces from volume data, IEEE Transaction on Computer Graphics and Applications, 1988, 8(3): 29-37.

2. R. Drebin, L. Carpenter, P. Hanrahan. Volume Rendering. Computer Graphics, Vol. 22, No. 4, pp. 65-74, Aug. 1988.
3. L. Westover. Footprint evaluation for volume rendering. Computer Graphics (SIGGRAPH '90 Proceedings), 24(4):367-376, August 1990.
4. D. Laur and P. Hanrahan, Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering, ACM Computer Graphics, Proc. SIGGRAPH ’93, 25(4):285–288, July 1991.
5. J. Huang, K. Mueller, N. Shareef, et al. FastSplats: Optimized Splatting on Rectilinear Grids. In Proceedings of IEEE Visualization 2000, pp. 219-226, October 2000.
6. P. Lacroute and M. Levoy, Fast volume rendering using a shear-warp factorization of the viewing transformation, Proc. SIGGRAPH ‘94, pp. 451- 458, 1994.
7. P. Lacroute, M. Levoy. Real-time volume rendering on shared-memory multiprocessors using the shear-warp factorization. In Proc. of Parallel Rendering Symposium’95, pp. 15-22, 1995.
8. J. Sweeney, K. Mueller, Shear-Warp Deluxe: The shear-warp algorithm revisited. In Proceeding of Joint Eurographics - IEEE TCVG Symposium on Visualization 2002, Barcelona, Spain, May 2002, pp. 95-104.
9. J. P. Schulze, M. Kraus, U. Lang, et al. Integrating Pre-Integration into the Shear-Warp Algorithm. In Proceedings of the Third International Workshop on Volume Graphics, Tokyo, July 7-8, 2003, pp. 109-118.
10. Brian Cabral, Nancy Cam, Jim Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In Proc. Symposium on Volume Visualization ’94, pages 91–98. ACM SIGGRAPH, 1994.
11. R. Westermann, T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In Computer Graphics, SigGraph Annual Conference Series, pages 169–177, 1998.
12. M. Meißner, U. Hoffmann, W Straßer. Enabling Classification and Shading for 3D Texture Based Volume Rendering Using OpenGL and Extensions. In Proc. of Visualization ’99, 1999.
13. C. Resk-Salama, K. Engel, M. Bauer, G. Greiner, T. Ertl， Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization，in Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware 2000.
14. K. Engel, M. Kraus, and T. Ertl, High-quality pre-integrated volume rendering using hardware accelerated pixel shading, in Proc. Eurographics/SIGGRAPH Workshop on

Graphics Hardware 2001.

15. Joe Kniss, Gordon Kindlmann, Charles Hansen. Multidimensional Transfer Functions for Interactive Volume Rendering. IEEE Transactions on Visualization and Computer Graphics, Volume 8, Issue 3, pp. 270 – 285, 2002
16. Roettger S., Guthe S., Weiskopf D., et al. Smart hardware accelerated volume rendering. In EUROGRAPHICS/IEEE Symposium on Visualization (2003), pp. 231-238.
17. J. Krüger, R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In Proc. of IEEE Visualization 2003, pp. 287–292.
18. Eric B. Lum, Brett Wilson, Kwan-Liu Ma. High-Quality Lighting and Efficient Pre-Integration for Volume Rendering. In Proc. of Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2004).
19. Joe Kniss, Gordon Kindlmann, Charles Hansen. Multidimensional Transfer Functions for Interactive Volume Rendering. IEEE Transactions on Visualization and Computer Graphics, Volume 8, Issue 3, pp. 270 – 285, 2002

# 5 三维人机交互的设计与实现

随着计算机硬件和软件的飞速发展，传统的二维用户接口（User Interfaces）已经无法适应人机交互的需求，这一现象突出的反应在虚拟现实技术中，在虚拟的三维场景中操纵三维物体，传统的二维交互模式显然不能胜任，为此，人们开始将更多的注意力转移到三维交互技术上来。

## 5.1 背景介绍

对三维交互的研究由来已久，从设备角度讲，这方面的研究主要有两个方向，一是使用三维的输入输出设备模拟三维交互，比如三维跟踪球、数据手套、头盔显示器等，但由于此类设备价格昂贵、应用面窄且操作复杂而无法普及；二是用传统的二维输入输出设备（鼠标、键盘、普通显示器等）模拟三维交互，虽然在很多的情况下难以得到十分精确的控制，但由于其设备简单且普及面广而得到广泛应用。在这方面比较突出的是 Brookshire D. Conner 等在 1992 年首先提出的 3D Widgets[1]，MITK 所包含的三维交互设计框架中最核心的元素也即来源于此。

在[1]中，作者给出了 3D Widgets的定义，即“一种封装了三维几何形状及行为的实体”，其中的“三维几何形状”即指 3D Widgets本身的外观，“行为”包括了 3D Widgets对三维场景中其他物体的控制和对其他物体信息的显示。此后，围绕 3D Widgets展开了一系列的研究，如[2]提出了一套称为“导轨（racks）”的 3D Widgets可用于控制三维物体的形变；[3]提出了一种称为“阴影（shadows）”的 3D Widgets，可以清楚地展示三维空间中物体间的相对位置并可对物体进行直接的定位；[4]中更是在Widgets中增加了时间的维度，使其可以控制三维物体在空间中的运动。随着研究的深入，3D Widgets也越来越显示出其在三维交互中的重要价值，比如Joe Kniss等在[5]中将 3D Widgets用于交互式体绘制中对多维传递函数的调节，极大地降低了手工调节传递函数的难度；[6]和[7]分别将 3D Widgets用于体数据的浏览和医学数据虚拟现实的应用。同时，各种针对 3D Widgets的开发包也在不断涌现，如[8][9]等。

尽管如此，从总体上来说，目前在三维人机交互的设计上并没有一个得到普遍公认的框架或标准，这主要是由三维交互本身的复杂性所决定的。而从应用

的角度来讲，三维人机交互在可视化领域具有非常重要的价值，尤其在医学影像可视化中三维交互对于更方便、准确地使用可视化的结果辅助医生进行诊断和手术模拟具有十分重要的意义。因此在 MITK 中设计并实现一个实用的、模块化的、可扩展的三维交互框架是十分必要的。

## 5.2 以 3D Widgets 为核心的三维人机交互的框架设计

### 5.2.1 3D Widgets 的设计准则

Scott S. Snibbe等在[2]中提出了 3D Widgets设计中应遵循的一些基本原则：

（1） 行为自明。即 Widget 的外观应该能够直观的反映出它的行为；

（2） 去除不必要的自由度。即对 Widget 的行为方式做尽可能多的限定，这样不仅能极大地降低实现难度，而且便于用户理解和使用该 Widget。

此外，对于不同应用，Widget的设计要求也是不同的，比如[2]中提到对于一般的用于展示或设计用途的造型工具，Widget的设计只要能达到让需展示的三维物体“看起来对”就可以了；而在机械或工业制造领域，Widget必须能够精确地调节零件的参数值，并且对任何参数的更改必须是“可再现的”。而医学影像可视化中 3D Widgets的设计要求更接近于后者，一方面，让受控于Widget的三维物体正确地显示是基本要求；另一方面，更重要的是Widget所反映或调节的三维物体的物理参量必须达到足够的精度。

### 5.2.2 以 3D Widgets 为核心的三维交互框架总体结构

MITK中所实现的三维交互框架以 3D Widgets为核心，其总体结构如 图 5-1 所示。其中，组成这个框架的所有模块均继承自同一个根Object。View用于控制整个三维场景的最终显示，所有在三维场景（View）中显示的物体都称为Model，处于核心地位的WidgetModel从概念上来说也是属于Model的一种，因为它本身也提供了可显示的外观。而除了WidgetModel，在三维场景中还存在另外一种Model，即观察对象，称为DataModel，是WidgetModel的操纵对象。但本身就处于三维场景中的WidgetModel无法自行触发对DataModel的控制，在它和View之间需要一个沟通的桥梁，这个桥梁就是Manipulator模块，它就像是一个驱动器，接受View截获的命令，然后驱动WidgetModel对DataModel实施控制。此外，还

有一个Observer模块对Model进行观察，随时反映Model状态的改变，比如用于测量的WidgetModel就通过Observer将测量结果返回给用户。

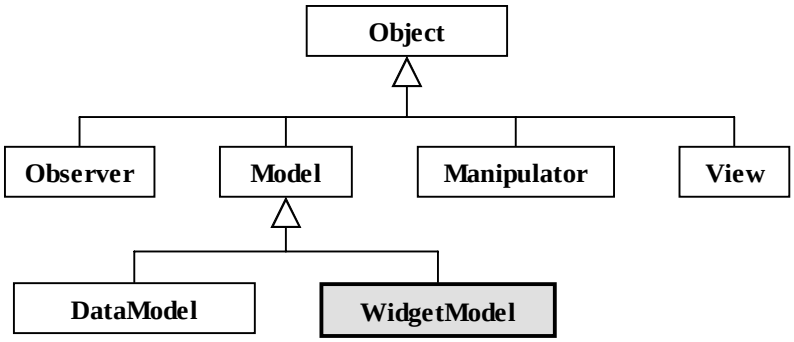



图 5-1 以 3D Widgets 为核心的三维交互框架结构

### 5.2.3 以 3D Widgets 为核心的三维交互框架设计

整个三维交互框架以 3D Widgets 为核心，但要顺利完成整个交互过程，其他模块的支持是必不可少的。

首先，是 widgets 所操纵的对象 DataModel，没有它，widgets 就没有存在的必要。具体来说，WidgetModel 和 DataModel 之间存在一种依赖关系，一般情况下一个 WidgetModel 总是与唯一的一个 DataModel 相关联，这由其 DataModel 成员 m_SourceModel 指出，而一个 DataModel 可以同时与一组 WidgetModel 产生关系，这由 DataModel 中的一个 WidgetModel 数组成员 m_WidgetModels 维护。当然，这些关系并非是必然存在的，比如一个 WidgetModel 完全可以不与任何一个 DataModel 关联。

其次，需要由 Manipulator 来驱动 WidgetModel 实现对 DataModel 的操纵，为此，该 Manipulator 必须具备从 View 中拣选出当前鼠标所指 WidgetModel 的能力，并能够将控制权移交给该 WidgetModel，而 WidgetModel 也必须提供给 Manipulator 一个 Select()接口实现拣选操作，以及相应的 OnMouseDown()、OnMouseUp()和 OnMouseMove()接口来接受控制权的转移并实现具体的控制行为。此外，WidgetModel 还需提供 Pick ()和 Release()接口给 Manipulator 以控制选中或释放时自身状态的更新。

第三，需要一个开放的 Observer 提供给用户以各种合适的方式显示 WidgetModel 所提取的相关 DataModel 的一些几何或物理参量。Observer 是一个高度抽象的类，它只提供一个 Update()接口而没有任何具体内容，每一个自 Object 继承的类都具有添加多个 Observer 的能力（Observer 本身除外），WidgetModel 当然也不例外，当 WidgetModel 参数更新之后即调用所有与之关联的 Observer 的 Update()接口通知 Observer 更新所要显示的数据，而具体的显示方式由用户在其具体实现的 Update()接口中给出。

最终，由 View 提供给 DataModel 和 WidgetModel 展示的舞台，显示最后的结果，同时，View 还肩负着与操作系统打交道的任务，负责截获窗口的鼠标和键盘消息，根据不同的消息组合调用 Manipulator 相应的接口，启动三维交互的整个过程。

上述这些模块及它们之间的相互作用构成了整个三维人机交互的框架，如图 5-2 所示。

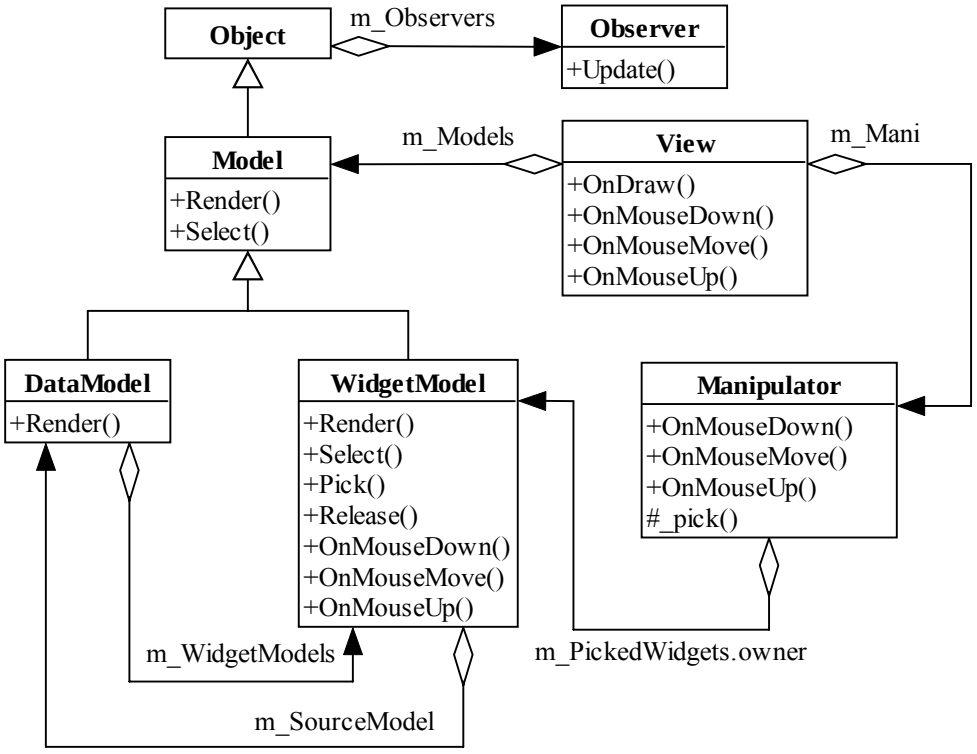



图 5-2 三维人机交互框架中各模块之间的关系

从图中可以清晰地描绘出整个三维交互的过程：

（1） View 显示三维场景中的各个 Model，接收鼠标和键盘消息，根据不同消息组合驱动 Manipulator；

（2） Manipulator 在合适的时机（比如 OnMouseDown()被触发时）根据当前鼠标位置在场景中进行拣选操作，若有 WidgetModel 被选中，则将控制权移交到选中的 WidgetModel，否则进行一般的处理；

（3） 掌握了控制权的 WidgetModel 将实现其预先定义的行为，其行为的影响最终反映在 View 及 Observer 中。

整个三维交互的过程就是（1）（2）（3）不断重复的过程。

## 5.3 以 3D Widgets 为核心的三维人机交互的实现

在上述的设计框架中，与交互功能的实现密切相关的核心部分是 Manipulator 和继承自 WidgetModel 的具体的 Widgets，下面对交互框架具体实现的介绍将主要围绕这两方面展开。而 View 和 Obserser 在整个交互过程中并不起决定性的作用，并且设计相对简单，这里就不再赘述。

### 5.3.1 Manipulator 的实现

Manipulator 的关键是拣选功能，由于整个交互框架采用 OpenGL 绘制 Model，因此很自然的拣选功能最终也将基于 OpenGL 提供的选择和反馈机制。

交互操作绝大部分情况下采用鼠标作为输入设备，对各种鼠标事件，Manipulator 将做如下处理：

- 当 OnMouseDown()被触发时（鼠标按键按下）：首先判断当前是否有 WidgetModel 处于选中状态，若有则调用其 Release()接口释放它；然后在 OpenGL 选择模式下绘制整个场景，即遍历场景中的每个 Model，依次调用其 Select()接口，Model 基类中缺省的 Select()什么也不做，只有可被选择的 Model 才实现它自己的 Select()，在这里就是直接调用 Render()；如果选择模式的绘制过程返回了某个 WidgetModel，则调用其 Pick()接口选中它，然后调用其 OnMouseDown()接口转移控制，否则按常规处理。
- 当 OnMouseMove()被触发时（移动鼠标）：若当前有 WidgetModel 处于选

中状态则调用其 OnMouseMove()接口转移控制，否则按常规处理。

- 当 OnMouseUp()被触发时（鼠标按键松开）：若当前有 WidgetModel 处于选中状态则调用其 OnMouseUp()接口转移控制，否则按常规处理。

上述实现是将鼠标按键按下的事件作为触发拣选操作的时机，这并不是唯一的实现方案，这个三维交互框架为用户提供了最大程度的自由，用户完全可以采用自己的方式来实现三维交互的 Manipulator，只要其具体实现符合整个三维交互框架的接口规范即可，具体的说就是从 mitkManipulator 派生出自己的类，在里面实现几个需要实现的虚函数就可以了，主要是 OnMouseDown()、OnMouseUp()和 OnMouseMove()三个函数。至于拣选操作，你可以按照自己的想法来实现它，MITK 对此并未作出任何限定。但是，如果你觉得这样很麻烦（编程实现拣选操作确实也比较复杂）或者对 OpenGL 的选择和反馈机制并不熟悉，没有关系，MITK 已经为你提供了一个实现了拣选操作的基类：mitkPickManipulator，它提供了一个保护成员_pick()来实现拣选操作，其函数原型为：

```
bool _pick(int xPos, int yPos);
```

该函数接受当前鼠标在 View 内的坐标为输入参数，返回一个 bool 值以示有无 WidgetModel 被选中。若返回 true，则选中物体的相关信息可以在如下的一个结构中得到：

```
typedef struct _widget_names
{
    mitkWidgetModel *owner;
    GLuint name1;
    GLuint name2;
    GLuint name3;

} WidgetNames;
```

该结构的定义在 mitkWidgetModel.h 中。其中 owner 指向选中的 WidgetModel，这是我们在编写自己的三维交互 Manipulator 时所需要的；下面 name1 到 name3 标明了被选中的是该 WidgetModel 的哪一个组成部件，这是被选中的 WidgetModel 在实施具体的控制时所需要的信息，目前只用到了 name3，name1 和 name2 作为将来扩展功能用。在 mitkPickManipulator 有一个此结构的成员变量：m_PickedWidgets，若有 WidgetModel 被选中，被选中的 WidgetModel

的信息就被填入到此结构中，我们只需将其作为参数，通过调用m_PickedWidgets.owner->Pick(m_PickedWidgets)就可以将必要的信息传给被选中的 WidgetModel。此后，只要通过 owner 这个指针调用 WidgetModel 的OnMouseDown()、OnMouseUp()及 OnMouseMove()，即可将控制权转移到具体的 WidgetModel。MITK 中已经提供了一个标准的带三维交互功能的 Manipulator：mitkWidgetsViewManipulator，该类即继承自 mitkPickManipulator，下面是其三个主要虚函数的实现，可以体会一下如何使用 mitkPickManipulator 提供的拣选功能：

```
void mitkWidgetsViewManipulator::OnMouseDown(int mouseButton,
                                             bool ctrlDown,
                                             bool shiftDown,
                                             int xPos,
                                             int yPos)
{
    if (m_View == NULL) return;
    switch(mouseButton)
    {
        case MITK_LEFTBUTTON:
            m_ButtonDown[MITK_LEFTBUTTON] = true;
            m_OldX[MITK_LEFTBUTTON] = xPos;
            m_OldY[MITK_LEFTBUTTON] = yPos;
            if (this->_pick(xPos, yPos)) //有 Widgets 被选中
            {
                // 通知 WidgetModel 被选中并传递必要信息
                m_PickedWidgets.owner->Pick(m_PickedWidgets);

                // 将控制权转移到选中的 WidgetModel
                m_PickedWidgets.owner->OnMouseDown(mouseButton,
                                                   ctrlDown,
                                                   shiftDown,
                                                   xPos,
                                                   yPos);
            }
            m_View->Update();
        break;

        case MITK_MIDDLEBUTTON:
```

```
            m_ButtonDown[MITK_MIDDLEBUTTON] = true;
            m_OldX[MITK_MIDDLEBUTTON] = xPos;
            m_OldY[MITK_MIDDLEBUTTON] = yPos;
        break;

        case MITK_RIGHTBUTTON:
            m_ButtonDown[MITK_RIGHTBUTTON] = true;
            m_OldX[MITK_RIGHTBUTTON] = xPos;
            m_OldY[MITK_RIGHTBUTTON] = yPos;
        break;
    }
}
//----------------------------------------------------------------------
void mitkWidgetsViewManipulator::OnMouseUp(int mouseButton,
                                           bool ctrlDown,
                                           bool shiftDown,
                                           int xPos,
                                           int yPos)
{
    if (m_View == NULL) return;
    switch(mouseButton)
    {
        case MITK_LEFTBUTTON:
            m_ButtonDown[MITK_LEFTBUTTON] = false;
        break;
        case MITK_MIDDLEBUTTON:
            m_ButtonDown[MITK_MIDDLEBUTTON] = false;
        break;
        case MITK_RIGHTBUTTON:
            m_ButtonDown[MITK_RIGHTBUTTON] = false;
        break;
    }

    // 如果当前有选中的 WidgetModel 则转移控制权
    if (m_PickedWidgets.owner)
    {
        m_PickedWidgets.owner->OnMouseUp(mouseButton, ctrlDown,
                                         shiftDown, xPos, yPos);
    }
}
```

```
//----------------------------------------------------------------------
void mitkWidgetsViewManipulator::OnMouseMove(bool ctrlDown,
                                             bool shiftDown,
                                             int xPos,
                                             int yPos)
{
    if(m_View == NULL)    return;
    if(m_ButtonDown[MITK_LEFTBUTTON] == true)
    {
        if (m_PickedWidgets.owner != NULL) // 有选中则转移控制
            m_PickedWidgets.owner->OnMouseMove(ctrlDown, shiftDown,
                                    xPos, yPos,
                                    xPos - m_OldX[MITK_LEFTBUTTON],
                                    m_OldY[MITK_LEFTBUTTON] - yPos);
        else // 否则按常规处理
            m_View->Rotate(m_OldY[MITK_LEFTBUTTON] - yPos,
                           xPos - m_OldX[MITK_LEFTBUTTON], 0.0f);
        m_OldX[MITK_LEFTBUTTON] = xPos;
        m_OldY[MITK_LEFTBUTTON] = yPos;
    }
    else if(m_ButtonDown[MITK_MIDDLEBUTTON] == true)
    {
        m_View->Translate(xPos - m_OldX[MITK_MIDDLEBUTTON],
                          m_OldY[MITK_MIDDLEBUTTON] - yPos, 0.0f);
        m_OldX[MITK_MIDDLEBUTTON] = xPos;
        m_OldY[MITK_MIDDLEBUTTON] = yPos;
    }
    else if(m_ButtonDown[MITK_RIGHTBUTTON] == true)
    {
        m_View->Scale(m_OldY[MITK_RIGHTBUTTON] - yPos);
        m_OldY[MITK_RIGHTBUTTON] = yPos;
    }
    else return;

    m_View->Update();
}
//----------------------------------------------------------------------
```

### 5.3.2 实现具体的 WidgetModel

在整个三维交互框架中，WidgetModel部分的继承结构如 图 5-3 所示。考虑到整个框架的弹性，有必要使其向下兼容二维的医学图像，而二维的View和三维的View存在一定差异，如果让同一个WidgetModel同时支持二维和三维的View，单个WidgetModel的实现难度和复杂度将大大增加，因此从框架的弹性和扩展的难易程度考虑，将WidgetModel分为二维和三维两类。并且这样做还有利于从WidgetModel扩展出更高维的Widget，很容易想到的是将时间的维度加进去，实现动态的Widget[4]。当然，下面的重点是讨论跟三维交互直接相关的三维WidgetModel的实现。

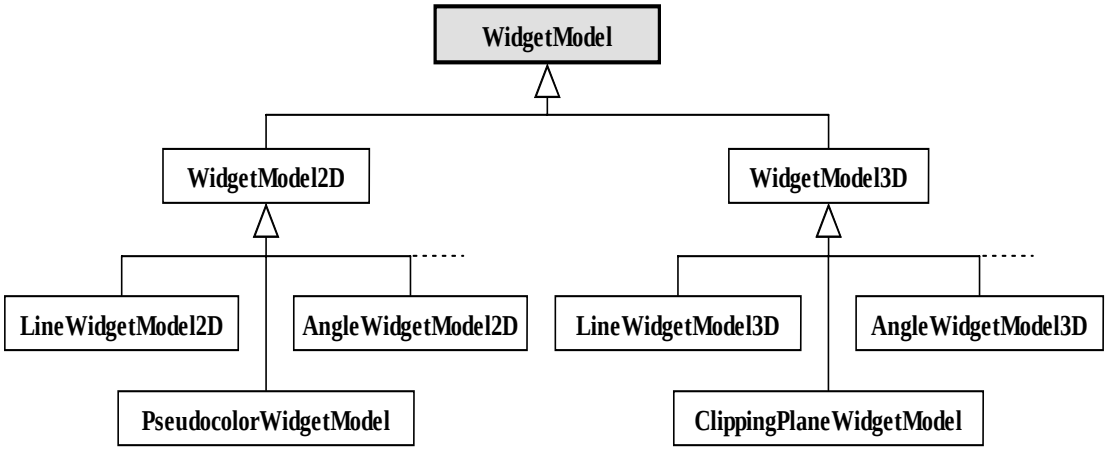



图 5-3 WidgetModel 的继承结构

所有 Widget 的基类是 mitkWidgetModel，这个类规定了一些如下一些统一的接口：

```
virtual void Pick(const WidgetNames &names) = 0;
virtual void Release() = 0;
virtual void Select(mitkView *view);
virtual void OnMouseDown(int mouseButton, bool ctrlDown,
                         bool shiftDown, int xPos, int yPos) = 0;
virtual void OnMouseUp(int mouseButton, bool ctrlDown,
                       bool shiftDown, int xPos, int yPos) = 0;
virtual void OnMouseMove(bool ctrlDown, bool shiftDown, int xPos,
                         int yPos, int deltaX, int deltaY) = 0;
virtual void SetSourceModel(mitkDataModel *model);
```

```
    mitkDataModel* GetSourceModel() { return m_SourceModel; }
    virtual void SetView(mitkView *view) = 0;
    virtual bool IsOpaque() { return m_IsOpaque; }
    virtual bool IsActive() { return m_IsActive; }
    void UpdateObservers() { this->_updateObservers(); }
```

其中 Pick()的作用不用多说了，Release()的作用与 Pick()相对，这一对操作在子类中必须实现，其实现的范例如下：

```
void mitkLineWidgetModel3D::Pick(const WidgetNames &names)
{
    // 得到选中部件的 id
    // 一个 Widget 通常由几个基本的部件构成，每个部件对应一个无符号整型 id，
    // 其含意由具体的 WidgetModel 定义
    m_PickedObj = names.name3;

    // 一般在选中后将其设为不透明显示
    // 该参数实际上与你的 WidgetModel 实际绘制是不是透明没有必然的联系，
    // 它的作用是给 View 提供一个绘制顺序的参考，在 View 绘制它所包含的所有 Model
    // 时将根据 IsOpaque()的返回值先绘制所有不透明的物体，再绘制所有透明物体，
    // 这样，整个场景看上去才比较正常，如果你需要确保自己实现的 WidgetModel 的
    // 绘制顺序比较靠后，那完全可以将其设为 false，而不必管是不是真的要将这个
    // WidgetModel 绘制成透明的
    m_IsOpaque = true;

    // 标志本 WidgetModel 进入活动状态
    m_IsActive = true;

    // 通知 Observer 状态已更新
    this->_updateObservers();
}
//----------------------------------------------------------------------
void mitkLineWidgetModel3D::Release()
{
    // 将选中部件置为“unknown”，表示无部件被选中
    m_PickedObj = unknown;

    // 可以在释放后设为透明显示
    m_IsOpaque = false;

    // 标志本 WidgetModel 离开活动状态
```

```
    m_IsActive = false;

    // 通知 Observer 状态已更新
    this->_updateObservers();
}
//----------------------------------------------------------------------
```

以上是在 Pick()和 Release()里面需要完成的一些基本工作，你也可以根据自己的需要在里面更新一些自定义的状态变量。其中涉及的 m_PickedObj 是以无符号整型数表示的当前选中的组成这个 Widget 的某一个部件的 id，各部件的 id 一般是在具体的 WidgetModel 中定义的。比如在 MITK 中的 mitkLineWidgetModel3D 就定义了这样一组 id：

```
    enum
    {
        unknown,
        arrow0,
        arrow1,
        line
    };
```

可以看出该 WidgetModel 是由两个以箭头表示的端点和一条线段组成的，这三个组件可以分别被选中，在 WidgetModel 中根据选中的部件做相应的操作。

Select()功能是检测本 WidgetModel 是否被选中，一般是在选择操作中遍历地调用所有 WidgetModel 的 Select()接口，以判断哪一个被选中了，这个在基类中已经有缺省的实现，如果你不想自己实现选择功能就不需要动这个函数。

OnMouseDown()、OnMouseUp()和 OnMouseMove()都是在子类中必须实现的，WidgetModel 所有的鼠标交互功能尽在这三个函数中体现（同时还可以搭配键盘上 Shift、Ctrl 键作为组合键使用）。在实现一个 WidgetModel 时，根据具体的鼠标和键盘操作，在这三个函数调整 WidgetModel 本身及其所控制的 Source Model 的显示参数以达到交互功能。以实现测量三维 Model 内任意两点间距离的 Widget 为例，我们将首先在 OnMouseDown()触发时记录当前线段两个端点的坐标，然后鼠标移动时在 OnMouseMove()中根据当时的鼠标位置即时更新鼠标所选中的端点部件在三维空间中的坐标，最终表现为该端点跟随鼠标指针移动，同时在 GetLineLength()函数中根据端点坐标以及该 Widget 控制对象（Source Model）的一些几何信息计算出当前线段的实际长度返回给用户。

余下的几个函数在基类中均已实现，一般情况下子类也不需要再动它们了。

最后还有一个很重要的函数 Render()，这是继承自 mitkModel 的函数，WidgetModel 所有的绘制工作均在这里面完成。以下就是 Render()函数在 OpenGL 绘制框架下的一个范例：

```
int MyWidgetModel::Render(mitkView *view)
{
    // 开启深度检测和光照
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);

    // 关闭所有裁剪平面（如果你不想让你的 WidgetModel 也被裁剪的话）
    this->_disableClippingPlanes();

    // 采用缺省的材质设定（当然可以替换成自定义的设定）
    this->_defaultMaterial();

    // MODELVIEW 矩阵压栈，以免对其他 Model 的绘制产生影响
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    {
        // 一般是直接从 SourceModel 得到模型变换矩阵，
        // 以使 WidgetModel 能跟随 SourceModel 做全局的几何变换
        if (m_SourceModel != NULL)
        {
            glMultMatrixf(*m_SourceModel->GetModelMatrix());
        }

        // 每个部件实际的绘制
        // 注意：如果你想采用 MITK 提供的选择功能的话，
        // 在绘制每个部件之前，调用 glLoadName(部件 id)来登记所绘部件的 id，
        // 选择程序将把选中部件的 id 返回，而 WidgetModel 将根据返回的 id 判断是
        // 哪个部件被选中
        glLoadName(id1);
        ......
        glLoadName(id2);
        ......
        glLoadName(id3);
        ......
```

```
    }
    // 对应于上面的 glPushMatrix()
    glPopMatrix();

    // 返回 1 表示绘制成功
    return 1;
}
```

值得一提的是 MITK 为三维模型的绘制环境维护了三个变换矩阵及其逆矩阵，它们是模型变换矩阵及其逆矩阵、视图变换矩阵及其逆矩阵和投影变换矩阵及其逆矩阵。模型变换矩阵及其逆矩阵可以通过 mitkModel 类提供的 GetModelMatrix()及 GetInverseOfModelMatrix()得到，视图变换矩阵和投影变换矩阵及其逆矩阵可以通过 mitkCamera 类提供的 GetViewMatrix()、GetInverseOfViewMatrix()、GetProjectionMatrix()及 GetInverseOfProjectionMatrix()得到，而指向 mitkCamera 的指针可以通过 mitkView 的 GetCamera()得到。前面两个矩阵的乘积就相当于 OpenGL 中的 MODELVIEW MATRIX，最后一个矩阵和 OpenGL 中的投影矩阵意义相同。有了这三个矩阵和它们的逆矩阵，不仅可以很方便地得到屏幕坐标和模型空间三维坐标之间的对应关系（这一点在 5.3.2 的第一条中有更具体的体现），而且在绘制时不需要再繁琐地调用 glTranslate*()、glRotate*()等函数来对模型进行几何变换，而只要将所需变换的矩阵作为参数调用一次 glMultMatrixf()就可以了，正如上面给出的例子所示。

另外，在 mitkWidgetModel2D 及 mitkWidgetModel3D 中均已实现了一些标准部件的绘制，比如 mitkWidgetModel3D 中的圆锥（_drawCone()）、圆柱（_drawCylinder()）、球（_drawSphere()）等，如果直接用这些部件构成 Widget，在绘制每一个部件时就不需要再显式地调用 glLoadName(id)了，而只要把 id 当作参数传给标准部件的绘制函数就可以了。

由上述介绍可见，与 Manipulator 相比，实现具体的 WidgetModel 要显得更自由一些，在遵循三维交互框架的接口规范的前提下，其行为的定义完全是开放的。实现具体的 WidgetModel，其关键在于在现有的框架下，如何使 WidgetModel 的行为效果与其预定义的一致，下面通过 MITK 中几个已经实现的 3D Widgets 来说明这一点。

(1) 测量距离的 Widget（mitkLineWidgetModel3D）

该 Widget 的功能是取得模型空间（坐标未经模型、视图及投影变换）中任意两点之间的距离，外观即为两端带有箭头的线段，箭头是可用鼠标控制的端点。该 Widget 在对重建的三维物体进行测量时经常用到。为达到测量距离的目的，必须能在模型空间任意移动线段的两个端点以及整条线段，而通过鼠标所获得的屏幕坐标是二维的，不可能直接用它去控制线段端点在三维空间中的位置，也不可能光靠鼠标实现具有三个自由度的端点（仅看作一个几何意义上的点的话）在空间的自由移动，这时通常需要其他设备（如键盘）的辅助来完成端点的自由移动，但这样做无疑会使操作复杂化。权衡利弊，我们采用另一种方式来达到目的，即限制在鼠标控制下端点移动的自由度，合理的做法是当鼠标选中某个端点时，让它只能在其当前所在的与屏幕平行的平面上移动，这样只用两个坐标就可以确定端点的当前位置（第三个坐标固定），只用鼠标就能进行操作，而且在视觉反馈上也是可以接受的，然后辅助以整个模型空间的旋转（改变第三个坐标），就可以达到端点自由移动的目的。

要实现如上描述的鼠标控制下端点的移动，关键是如何求得与当前鼠标在屏幕上的位置相对应的原模型空间中的三维点坐标，为此 Widget 必须知道以下一些信息：

- 选中端点未移动前在模型空间中的坐标（$x_{objold}$，$y_{objold}$，$z_{objold}$）；
- 模型变换矩阵 **M**、视图变换矩阵 **V** 和投影变换矩阵 **P**；
- 当前鼠标在屏幕上的位置（$x_{snew}$，$y_{snew}$）；
- 当前视区的位置和大小（$o_x$，$o_y$，$w$，$h$）。

其中 **M**、**V** 和 **P** 均为 4×4 矩阵，在 OpenGL 中 **M** 和 **V** 合并为一个矩阵，但在我们的三维交互框架中可以通过 SourceModel（或 WidgetModel 本身）和 View 分别得到这两个矩阵。

根据上述信息即可推算出移动后的端点在模型空间中的坐标（$x_{objnew}$，$y_{objnew}$，$z_{objnew}$）：

首先，由

$$\begin{bmatrix} x_{sold} \\ y_{sold} \\ z_{sold} \\ w \end{bmatrix} = \mathbf{PVM} \begin{bmatrix} x_{objold} \\ y_{objold} \\ z_{objold} \\ 1.0 \end{bmatrix}. \tag{5-1}$$

可以得到原坐标经变换后在屏幕坐标空间中的 $z$ 坐标 $z_{sold}$，要限制端点只能在与屏幕平行的平面移动，只需保持屏幕空间中 $z$ 坐标不变即可，因此只需将 $x_{sold}$ 和 $y_{sold}$ 分别用 $x_{snew}$ 和 $y_{snew}$ 替换然后反推回去就可以了，但有一点需要注意，通过上式计算所得到的屏幕坐标并非真正意义上的屏幕坐标，这些坐标的值均被归一化到[-1.0，1.0)，根据当前视区的位置和大小可以将其化到正常的屏幕坐标，但是这一步计算实际上是没有必要的，只需在反推前将新的屏幕坐标根据当前视区的位置和大小归一化到[-1.0，1.0)就可以了。（$x_{objnew}$，$y_{objnew}$，$z_{objnew}$）的计算如下所示：

$$\begin{bmatrix} x_{objnew} \\ y_{objnew} \\ z_{objnew} \\ w \end{bmatrix} = \mathbf{M}^{-1}\mathbf{V}^{-1}\mathbf{P}^{-1} \begin{bmatrix} \frac{2.0 \cdot (x_{snew} - o_x)}{w} - 1.0 \\ \frac{2.0 \cdot (y_{snew} - o_y)}{h} - 1.0 \\ z_{sold} \\ 1.0 \end{bmatrix} \tag{5-2}$$

上述 **M** 及其逆可通过 SourceModel（或 WidgetModel 本身）直接得到，**V** 和 **P** 及其逆可通过 View 中的 Camera 直接得到，当然也可以用 OpenGL 的 API 函数计算。但采用前者效率更高，并且有利于提高整个模块与底层平台的无关性。

对于整条线段的移动，两个端点的坐标要同时更新，而所有计算基于偏移量进行，最后将原坐标加上偏移量即为新坐标。

当 mitkLineWidgetModel3D 的 OnMouseMove()被触发时，通过上述计算即可即时更新端点坐标。采用上述方法可以保证端点紧随鼠标指针移动，从而达到精确控制端点位置的目的。同时，Observer 可以通过 GetLineLength()接口得到当前线段的长度。

由于坐标均为原始模型空间中的坐标，只要给予足够准确的原始数据的信

息，该 Widget 便能根据坐标以及各坐标轴方向上的单位长度（由 SourceModel 提供）给出在三维重建的模型空间中对应于原始物体的比较准确的测量长度，保证了测量的精度。

(2) 测量角度的 Widget（mitkAngleWidgetModel3D）

该 Widget 的功能是取得模型空间中任意三点组成的角的角度值，其外观即为由两条线段和三个控制点（两个箭头和一个球状的端点）构成的一个张开的角。采用与 mitkLineWidgetModel3D 相同的方式可以自由地控制三个端点的位置从而准确地作出所需的角，同时 Observer 可以通过 GetAngleInDegree()或 GetAngleInRadian()得到以角度或弧度表示的角度值。

(3) 裁剪平面 Widget（mitkClippingPlaneWidgetModel）

该 Widget 的功能是对当前场景中显示的三维物体在任意位置和任意方向上进行裁剪，其外观包括一个由 4 个圆柱体和 4 个球体围成的边框和边框内一个半透明的平面。该 Widget 对于观察三维物体的内部构造比较有用。

圆柱体和球体都是控制点，四个球体分别实现四种对裁剪平面的控制，包括：绕中心的任意旋转，外框相对中心的的缩放，垂直于平面的平移以及沿平面的平移，圆柱体也是实现沿平面的平移。其中缩放比较简单，而限制裁剪平面的平移路径（垂直于裁剪平面或平行于裁剪平面）则要费一番功夫，这里采用的方式是先反推出鼠标在屏幕上的平移向量所对应的在原模型空间中的平移向量，计算方法基本上与 mitkLineWidgetModel 计算端点坐标的方法相同，然后将平移向量在移动路径方向（裁剪平面的法线方向或裁剪平面本身）上进行投影，所得结果即为最终的平移向量，将裁剪平面的中心坐标加上这个平移向量即可。此外，为保证旋转基本上与鼠标的移动保持一致，这里采用了一个虚拟跟踪球来跟踪和控制裁剪平面的旋转[10]。

对物体进行裁剪的功能是通过激活 OpenGL 中的附加裁剪平面实现的。激活后将该 Widget 的平面法线方向和中心点坐标作为 OpenGL 中裁剪平面的参数，就可以达到裁剪的目的。

## 5.4 三维交互的应用实例

### 5.4.1 mitkLineWidgetModel3D 的应用实例

图 5-4 给出了一组使用LineWidgetModel3D对用等值面提取算法进行表面重建的结果进行测量的实例，原始数据是部分头部的CT扫描切片图像，其中Observer采用类似Tool Tip的方式实时返回测量的结果。

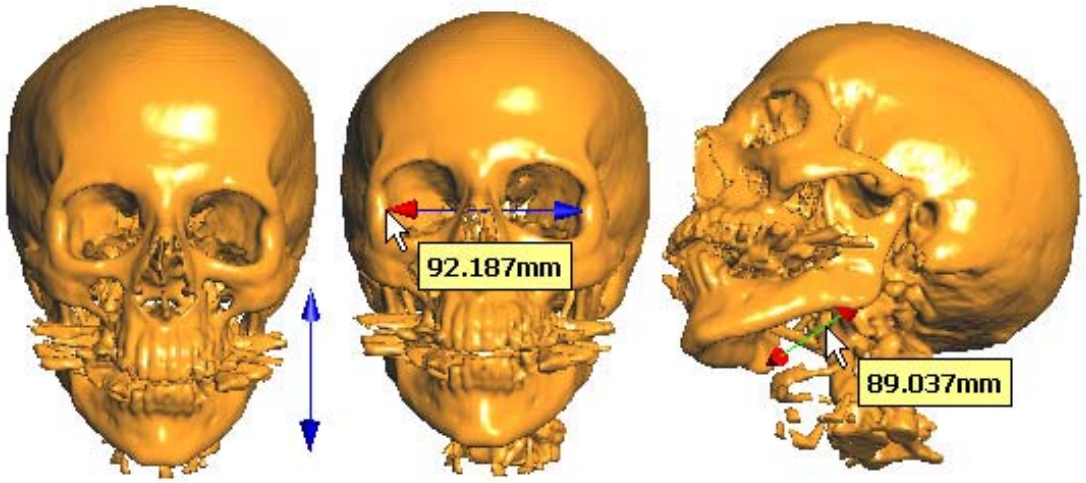



图 5-4 mitkLineWidgetModel3D 的使用实例

### 5.4.2 mitkAngleWidgetModel3D 的应用实例

图 5-5 给出了一组使用AngleWidgetModel3D对用等值面提取算法进行表面重建的结果进行测量的实例，原始数据与 4.1 中的相同，Observer也使用相同的方式返回结果。

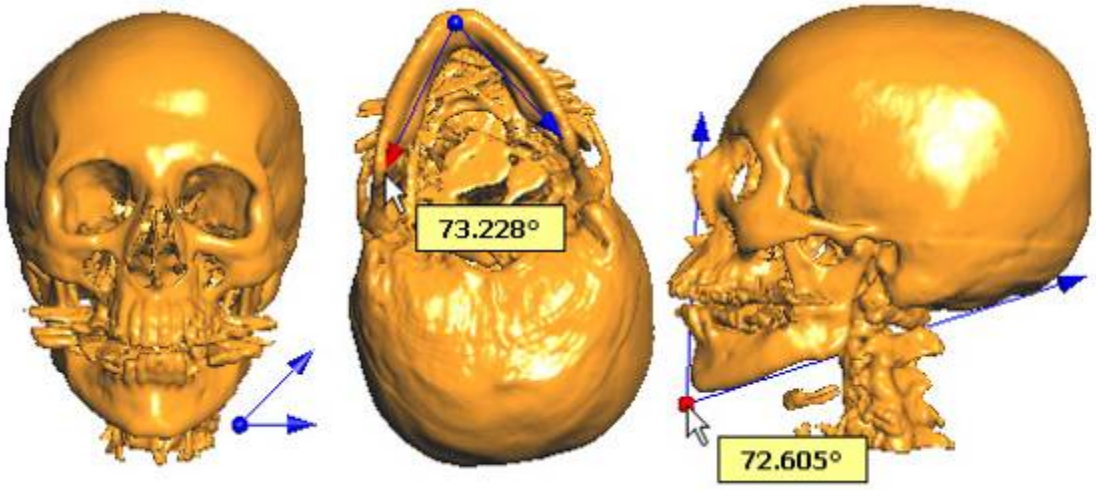



图 5-5 mitkAngleWidgetModel3D 的使用实例

### 5.4.3 mitkClippingPlaneWidget 的应用实例

图 5-6 给出了一组使用ClippingPlaneWidget对用等值面提取算法进行表面重建的结果进行裁剪的实例，原始数据与 4.1 中所用的相同。裁剪平面采用半透明显示，这样可以使物体之间的前后位置关系显得更清楚一些，当需要更清楚地观察被裁剪物体的内部时可以将其透明度调至最大（即完全透明）。

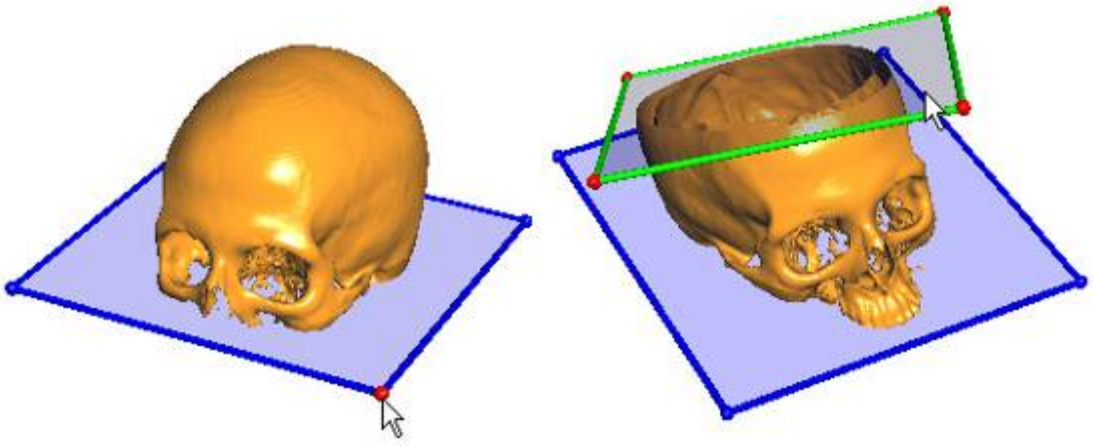


图 5-6 mitkClippingPlaneWidget 的使用实例

## 5.5 小结

本章主要介绍了 MITK 中基于 3D Widgets 的三维人机交互的设计与实现，集中讨论了以 3D Widgets 为核心的三维人机交互框架的设计思路及其具体实现。整个三维交互的框架采用模块化的设计思路，将功能分解到各个模块中，使整个框架具有很强的弹性和可维护性。同时，整个框架开放式的结构给用户提供了最大限度的自由，在遵循框架接口规范的前提下，用户可以根据自己的需求通过具现化 WidgetModel 来实现各种不同的三维交互功能。

MITK 的 3D Widgets 目前仍处于不断的完善与发展之中，以其为核心的三维交互框架也有待于进一步的完善和增强，特别是向框架中添加各种实用的 3D Widgets，从而增强整个框架的三维交互功能，最终使其成为一个简单易用、灵活可靠并且可扩展的三维人机交互平台。

### 参考文献

1. Brookshire D. Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, Andries van Dam. Three-dimensional widgets. Proceedings of Interactive 3D graphics Symposium, 1992, pp. 183-188.
2. Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Brookshire D. Conner, Andries van Dam. Using deformations to explore 3D widget design. Proceedings of SIGGRAPH'92, 1992, ACM, pp. 351-352.
3. Kenneth P. Herndon, Robert C. Zeleznik, Daniel C. Robbins, D. Brookshire Conner, Scott S. Snibbe, Andries van Dam. Interactive shadows. Proceedings of UIST'92, 1992, ACM, pp. 1-6.
4. J. Döllner, K. Hinrichs. Interactive, Animated 3D Widgets. Proceedings of Computer Graphics International'98, 1998, IEEE, pp. 278-286.
5. Joe Kniss, Gordon Kindlmann, Charles Hansen. Multidimensional Transfer Functions for Interactive Volume Rendering. IEEE Transactions on Visualization and Computer Graphics, Volume 8, Issue 3, pp. 270 – 285, 2002
6. Michael J. McGuffin, Liviu Tancau, Ravin Balakrishnan. Using Deformations for Browsing Volumetric Data. Proceedings of the IEEE Visualization Conference, 2003, pp. 401-408.
7. Fred Dech, Jonathan C. Silverstein. Rigorous Exploration of Medical Data in

Collaborative Virtual Reality Applications. Sixth International Conference on Information Visualisation, 2002, pp. 32-38

8. Marc P. Stevens, Robert C. Zeleznik, John F. Hughes. An architecture for an extensible 3D interface toolkit. Proceedings of UIST'94, 1994, ACM, pp. 59-67.
9. Robert C. Zeleznik, Kenneth P. Herndon, Daniel C. Robbins, Nate Huang, Tom Meyer, Noah Parker, John F. Hughes. An interactive 3D toolkit for constructing 3D widgets. Proceedings of SIGGRAPH'93, 1993, ACM, New York, NY, USA, pp. 81-84.
10. Michael Chen, S. Joy Mountford, Abigail Sellen. A study in interactive 3-D rotation using 2-D control devices. Proceedings of SIGGRAPH'88, 1988, ACM, pp. 121-129
11. Mingchang Zhao, Jie Tian, Xun Zhu, Jian Xue, Zhanglin Cheng, Hua Zhao. The Design and Implementation of a C++ Toolkit for Integrated Medical Image Processing and Analyzing. Proceedings of SPIE Medical Imaging 2004
12. Kenneth P. Herndon, Tom Meyer. 3D widgets for exploratory scientific visualization. Proceedings of UIST'94. 1994, ACM, pp. 69-70.
13. Mark R. Mine. Virtual environment interaction techniques. UNC Chapel Hill CS Dept.: Technical Report TR95-018. 1995.
14. Doug A. Bowman, Ernst Kruijff, Joseph J. LaViola, Jr., Ivan Poupyrev. An Introduction to 3-D User Interface Design. Presence, Vol. 10, No. 1, 2001, pp. 96-108.
15. Paul S. Strauss, Rikk Carey. An object-oriented 3D graphics toolkit. Proceedings of SIGGRAPH'92, 1992, ACM, pp. 341-347.
16. Kim MH, Choi SM, Rhee SM, Kwon DY, Kim HS. A Guided Interaction Approach for Architectural Design in a Table-Type VR Environment. 3rd IEEE Pacific Rim Conference on Multimedia (PCM 2002), 2002
17. Robert W. Lindeman, John L. Sibert, James N. Templeman. The Effect of 3D Widget Representation and Simulated Surface Constraints on Interaction in Virtual Environments. Proceedings of Virtual Reality Annual International Symposium, IEEE , 2001, pp. 141-148.

# 6 分割算法的设计与实现

MITK 的设计初衷是为了给医学影像领域的研究者提供一套具有一致接口的、可复用的、包括可视化、分割、配准功能的集成化的医学影像开发包，弥补 VTK 和 ITK 的一些缺憾，并引入一些新的特性，使得 MITK 能够成为除了 VTK＋ITK 以外的另外一个选择。

医学影像分割是医学影像处理与分析中的一个重点课题和难点，分割的结果是三维可视化和定量分析等后续处理的基础。近几年来虽然仍然有很多研究人员致力于图像分割的研究，发表了很多的研究成果，但由于问题本身的困难性，与八十年代相比并没有取得多少实质性的进展。本章旨在将分割算法集成在一个统一的框架内，力图设计一个具有良好可扩充性和可重用性的算法包，使得算法开发人员能利用算法开发包深入研究并改进各种算法，工程技术人员能利用开发包方便的生成自己的应用程序。

目前 MITK 中主要提供了一些主流算法，如 level set 算法、fast marching 算法、live wire 算法、区域增长算法、交互式分割算法和阈值分割算法。下面详细介绍各个算法的设计与实现，对于每种算法，将从如下三方面加以阐述：

- 原理概述

  该部分将简要介绍该算法的理论依据、数学方法。

- 该算法开发包的设计与实现

  该部分主要介绍该算法的算法流程、分割算法开发包的设计和实现方法，重点在于如何按照面向对象的软件设计方法来组织各个类，以使算法模块具有最大的灵活性、可重用性和可扩充性。

- 实验结果

  该部分以系统界面的形式给出每个算法的分割结果。

## 6.1 MITK 中的分割算法框架

分割算法开发包既可以作为 MITK 的一部分，也可以独立出来，作为一个

单独的开发包使用。它是一个基于面向对象方法的 ANSI C++开发包，目标是为用户提供一个具有一致接口的算法包。我们采用了基于设计模式的软件开发方法，这种方法强调代码模块化和对象间相互独立性，使得软件具有很高的灵活性。同时，由于开发包使用标准 C++来书写，它还大量使用了 C++标准模板库 STL 来简化一些繁琐的编程工作。

算法包主要分为四大模块：数据模块、数据获取模块、数据输出模块和数据处理模块。他们分别负责数据表示、文件读取、文件输出和数据处理工作。定义类 mitkObject 作为开发包中所有类的基类，在 mitkObject 中定义了开发包中所有类的一些共同属性、行为和接口，如有关类自身的一些信息、调试信息和内存管理接口等。

### 6.1.1 数据模块

数据模块主要用于数据的表示。分割算法处理的数据主要是一些有多张切片组成的体数据，在分割算法包中用类 mitkVolume 为他们提供一个统一的表示方法。

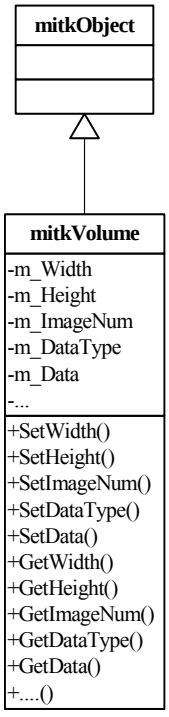



图 6-1

如图 6-1 所示，mitkVolume继承于mitkObject，它封装了体数据的一些特性，如体数据的宽度(m_Width)、体数据的高度(m_Height)、体数据的切片数(m_ImageNum)、体数据的数据类型(m_DataType)以及体数据数值(m_Data)等。用户通过mitkVolume提供的一些公用接口来设置和获取体数据的属性。如：用SetWidth()方法来设置体数据的宽度，用GetWidth()方法来获取体数据的宽度。

分割算法包处理的数据类型可以是二维的也可以是三维的。当 mitkVolume 的 m_ImageNum 值为 2 时，代表二维体数据即图像数据；当 mitkVolume 的 m_ImageNum 值为 3 时，代表三维体数据。

### 6.1.2 数据获取模块

数据获取模块的作用是从磁盘上读取输入文件，将其写入内存，并转化为统一的表示模式——mitkVolume。图像文件有很多种存贮格式，医学图像中比较常用的文件格式有 Dicom 文件、Im0 文件、Tiff 文件和生数据文件等。本章的分割算法平台除支持以上文件格式的读取外，还支持一些常见的文件格式如：jpeg、bmp 文件的读取。

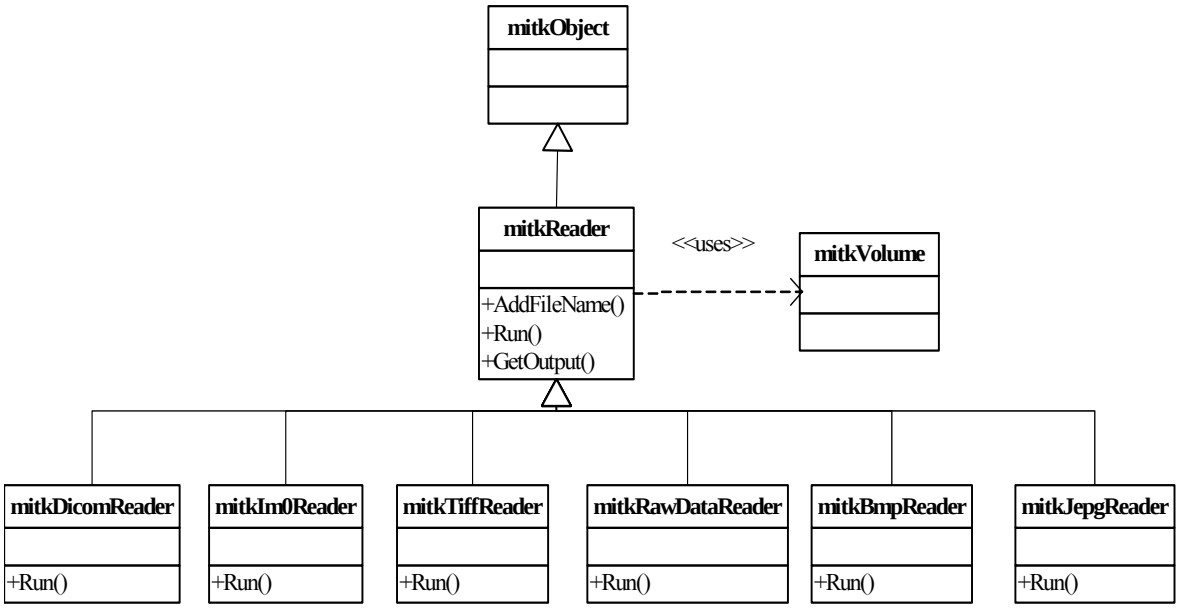



图 6-2

如图 6-2 所示，mitkReader 为数据获取模块定义了一个统一的接口。用户通过 AddFileName()指定将要读取的文件在磁盘上的路径；Run()函数为虚函数，mitkReader 的子类通过重载 Run()函数实现不同文件类型的读取；用户通过

GetOutput()获取 mitkReader 的输出结果——一个 mitkVolume 类型的数据。

例如, 欲将路径为“e:\ttt.im0” 的 Im0 类型的文件读取到 volume 中，可用如下代码：

```
mitkIm0Reader *reader = new mitkIm0Reader;
reader->AddFile(“e:\\ttt.im0”);
reader->Run();
mitkVolume *volume = reader->GetOutput();
```

### 6.1.3 数据输出模块

数据输出模块的作用是将内存中的 mitkVolume 数据写入磁盘的指定路径。同数据获取模块一样，数据输出模块支持 Dicom 文件、Im0 文件、Tiff 文件、生数据文件、jpeg 文件和 bmp 文件的输出。

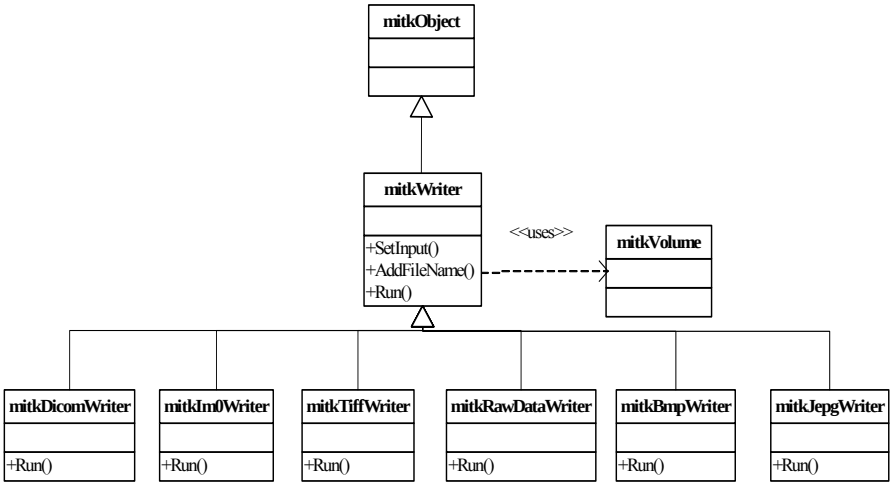



图 6-3

如 图 6-3 所示，mitkWriter为数据输出模块定义了一个统一的接口。用户通过SetInput 输入要保存的数据；通过AddFileName()指定将要输出的文件在磁盘上的路径；Run()函数为虚函数，mitkWriter的子类通过重载Run()函数实现不同文件类型的输出。

例如, 欲将内存中的 volume 数据写入路径为“e:\ttt.im0” 的文件中，可用

如下代码：

```
mitkIm0Writer *writer = new mitkIm0Writer;
writer ->AddFile("e:\\ttt.im0");
writer->SetInput(volume);
writer ->Run();
```

### 6.1.4 数据处理模块

数据处理模块是分割算法包的核心部分。它的输入是一个 mitkVolume 型的数据，输出也是一个 mitkVolume 型的数据，因此可以称它为一个过滤器(Filter)，它的作用是对输入的数据进行处理，将结果写入输出的数据。

如图 6-4 所示，因为数据处理模块的输入和输出数据都是mitkVolume类型，因此定义mitkVolumeToVolumeFilter为所有Filter的基类，在其中将函数Run()定义为虚函数，它的子类可以通过重载Run()函数实现不同的Filter功能。mitVolumeToVolumeFilter可以包含实现各种功能的子类，如：实现滤波功能的Filter，实现距离函数功能的Filter，和实现各种分割功能的Filter等。

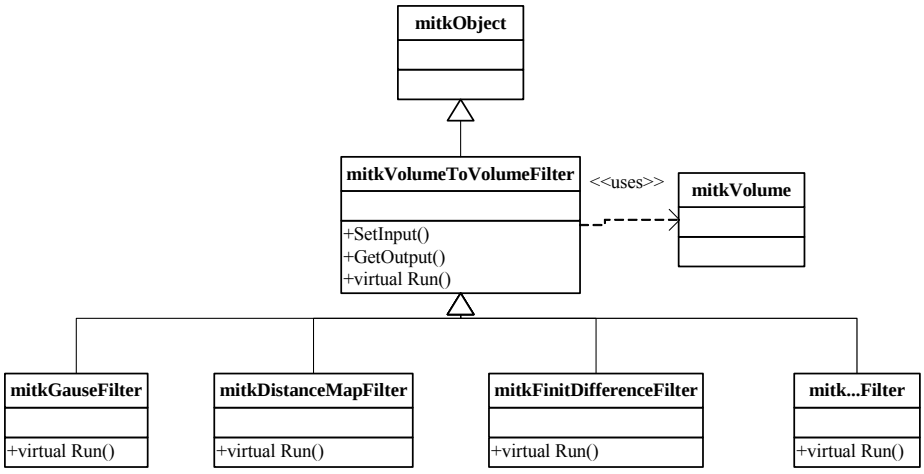



图 6-4

例如，要对 InputVolume 数据做阈值分割，高阈值和低阈值分别为 vh 和 vl，并将分割后的结果写入 OutputVolume 数据，可用如下代码：

```
mitkThresholdSegmentFilter *filter = new mitkThresholdSegmentFilter;
filter->SetInput(InputVolume);
filter->SetLowThreshold(vl);
```

```
filter->SetHighThreshold(vh);
filter->Run();
OutputVolume = filter->GetOutput();
```

如 图 6-5 所示，一个Filter的输出可以作为另一个Filter的输入，N个Filter可以组合成一个大的Filter。

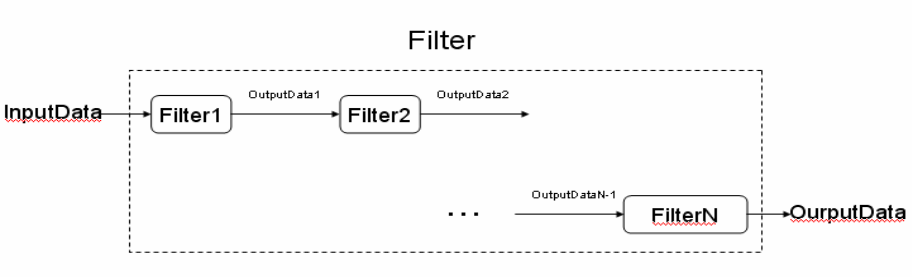



图 6-5

## 6.2 基于阈值的分割算法在 MITK 中的实现

### 6.2.1 原理概述

阈值分割是最常见的并行的直接检测区域的分割方法[1]。如果只用选取一个阈值称为单阈值分割，它将图像分为目标和背景两大类；如果用多个阈值分割称为多阈值方法，图像将被分割为多个目标区域和背景，为区分目标，还需要对各个区域进行标记。阈值分割方法基于对灰度图像的一种假设：目标或背景内的相邻象素间的灰度值是相似的，但不同目 标或背景的象素在灰度上有差异，反映在图像直方图上，不同目标和背景则对应不同的峰。选取的阈值应位于两个峰之间的谷，从而将各个峰分开（见 图 6-6）。

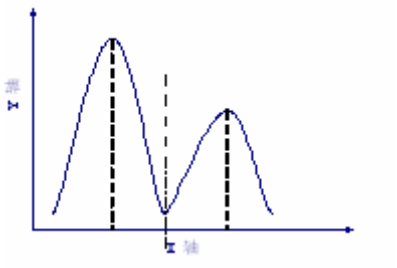


图 6-6 阈值分割示意图

阈值分割的优点是简单，同时对于不同类的物体灰度值或其他特征值相差很大时，它能很有效的对图像进行分割。阈值分割通常作为预处理，在其后应用其他一系列分割方法进行处理，它常被用于 CT 图像中皮肤、骨骼的分割。

阈值分割的缺点是不适用于多通道图像和特征值相差不大的图像，对于图像中不存在明显的灰度差异或各物体的灰度值范围有较大重叠的图像分割问题难以得到准确的结果。另外，由于它仅仅考虑了图像的灰度信息而不考虑图像的空间信息，阈值分割对噪声和灰度不均匀很敏感。针对阈值分割方法的缺点，不少学者提出了许多改进方法。在噪声图像的分割中，一些阈值分割方法还利用了一些象素邻域的局部信息，如基于过渡区的方法[2]，还有利用像素点空间位置信息的变化阈值法[3]，结合局部灰度[4]和连通信息[5]的阈值方法。

### 6.2.2 阈值分割算法开发包设计与实现


图 6-7 阈值分割类示意图

mitkThresholdImageFilter()用来实现阈值分割。

*函数 SetLowThreshold()*：设置低阈值；

*函数 SetHighThreshold()*：设置高阈值；

*函数 Run()*：启动与之分割算法。像素值在低阈值和高阈值之间的保留原值，其他的赋零值。

### 6.2.3 阈值分割结果示意图

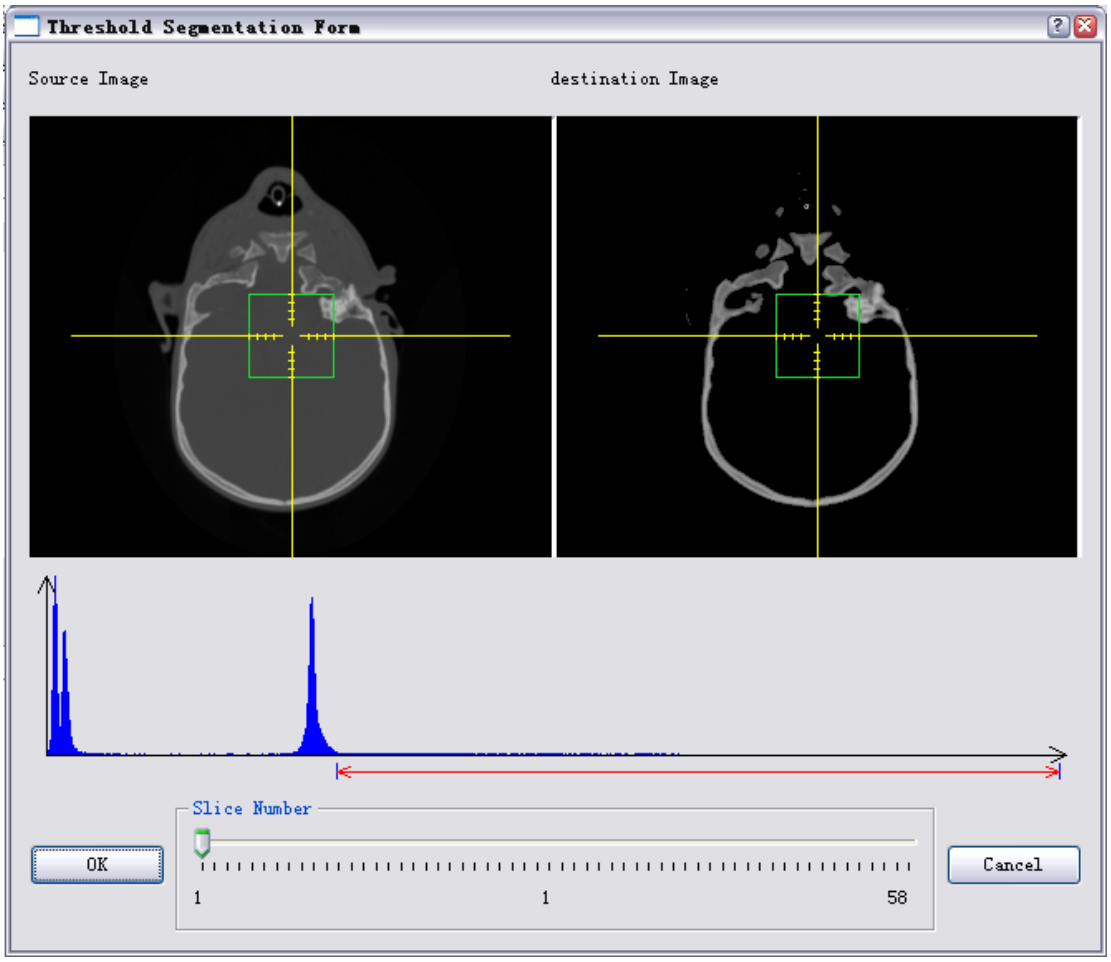



图 6-8 阈值分割结果实验结果

如 图 6-8 以系统界面的形式给出了与之分割的结果

## 6.3 区域增长算法在 MITK 中的实现

### 6.3.1 原理概述

区域生长是典型的串行区域分割方法，其特点是将分割过程分解为多个顺序的步骤，其中后续步骤要根据前面步骤的结果进行判断而确定。

区域生长的基本思想是将具有相似性质的像素集中起来构成区域，该方法需要先选取一个种子点，然后依次将种子像素周围的相似像素合并到种子像素所

在的区域中。区域生长算法的研究重点一是特征度量和区域增长规则的设计，二是算法的高效性和准确性。区域生长算法的优点是计算简单，特别适用于分割小的结构如肿瘤和伤疤[6]。与阈值分割类似，区域生长也很少单独使用，往往是与其他分割方法一起使用。

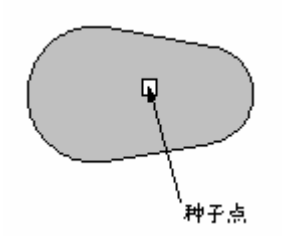



图 6-9 区域生长算法示意图

区域生长的缺点是它需要人工交互以获得种子点，这样使用者必须在每个需要抽取出的区域中植入一个种子点。同时，区域生长方法也对噪声敏感，导致抽取出的区域有空洞或者在局部体效应的情况下将原本分开的区域连接起来。为了解决这些缺点，J.F. Mangin 等提出了一种同伦的（homotopic）区域生长方法[7]，以保证初始区域和最终抽取出的区域的拓扑结构相同。另外，模糊连接度理论与区域生长相结合也是一个发展方向[8]。

在区域合并方法中，输入图像往往先被分为多个相似的区域，然后类似的相邻区域根据某种判断准则迭代地进行合并。在区域分裂技术中，整个图像先被看成一个区域，然后区域不断被分裂为四个矩形区域，直到每个区域内部都是相似的。在区域的分裂合并方法中[9]，先从整幅图像进行分裂，然后将相邻的区域进行合并。分裂合并方法不需要预先指定种子点，它的研究重点是分裂和合并规则的设计。但是，分裂可能会使分割区域的边界被破坏。

### 6.3.2 区域生长算法开发包的设计与实现

(1) 算法流程

如 图 6-10 给出了区域生长算法的算法流程，主要分为两个模块：

初始化部分：该部分主要工作有

1．由用户选取初始种子点；

2．初始化堆栈，将种子点压入队列；

循环部分：该部分主要工作有

1．循环结束条件：队列为空时循环结束；

2．从队列中取出队顶元素，获取它的邻域点，对于二维图像，取八邻域，对于三维图像，取六邻域；

3．相似度条件：有很多种定义方法，这里采取比较简单的定义方法，设当前队列顶端元素的灰度值为 gc，当前邻点灰度值为 gn，种子点的灰度值为 gs，nv、cv 为用户设定的值，定义当$|gc-gn|<nv$且$|gs-gn|<cv$时，满足相似度条件；

4．如果邻点满足相似度条件，则将其压入堆栈继续循环，如果不满足则跳出循环，结束程序；

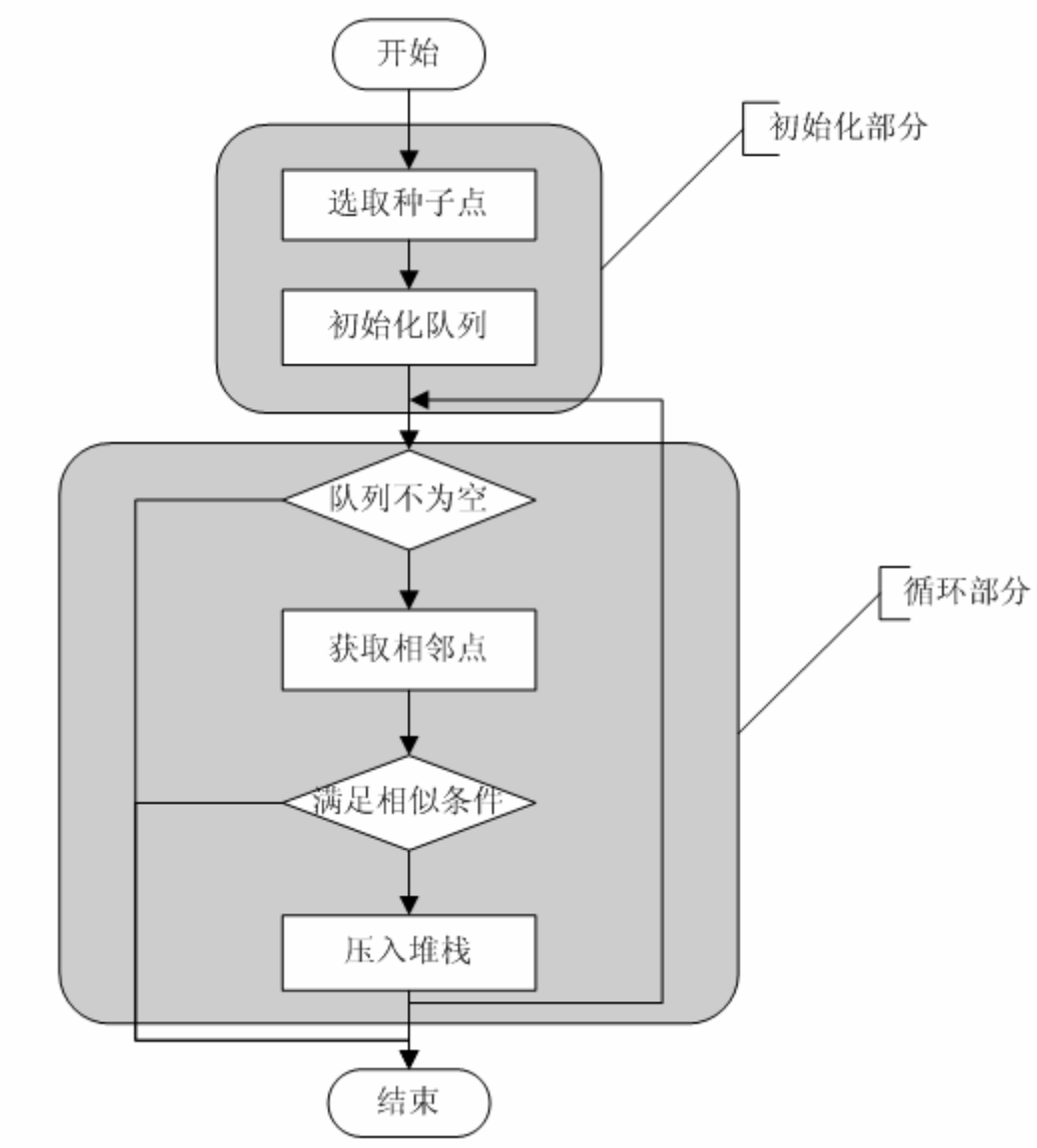



图 6-10 区域生长算法流程

(2) 类协作图

如图 6-11 给出了区域生长算法的类协作图。mitkRegionGrowImageFilter定义了按照图 6-10 定义了区域生长算法的主框架流程；mitk2DRegionGrowImageFilter 和 mitk3DRegionGrowImageFilter 是mitkRegionGrowImageFilter的子类，分别定义了二维和三维区域生长算法的框架；mitkRegionGrowTemplateFunction为所有区域生长算法中的相似度function定义了一个基类；mitkRegionGrowFunction是mitkRegionGrowTemplateFunction的一

个子类，它定义了一种相似度函数的具体实现。

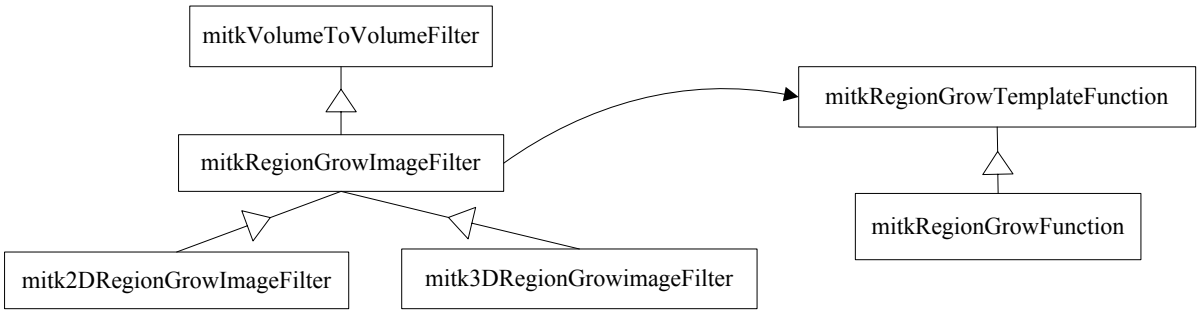



图 6-11 区域生长算法类协作图

(3) 类组成结构详解

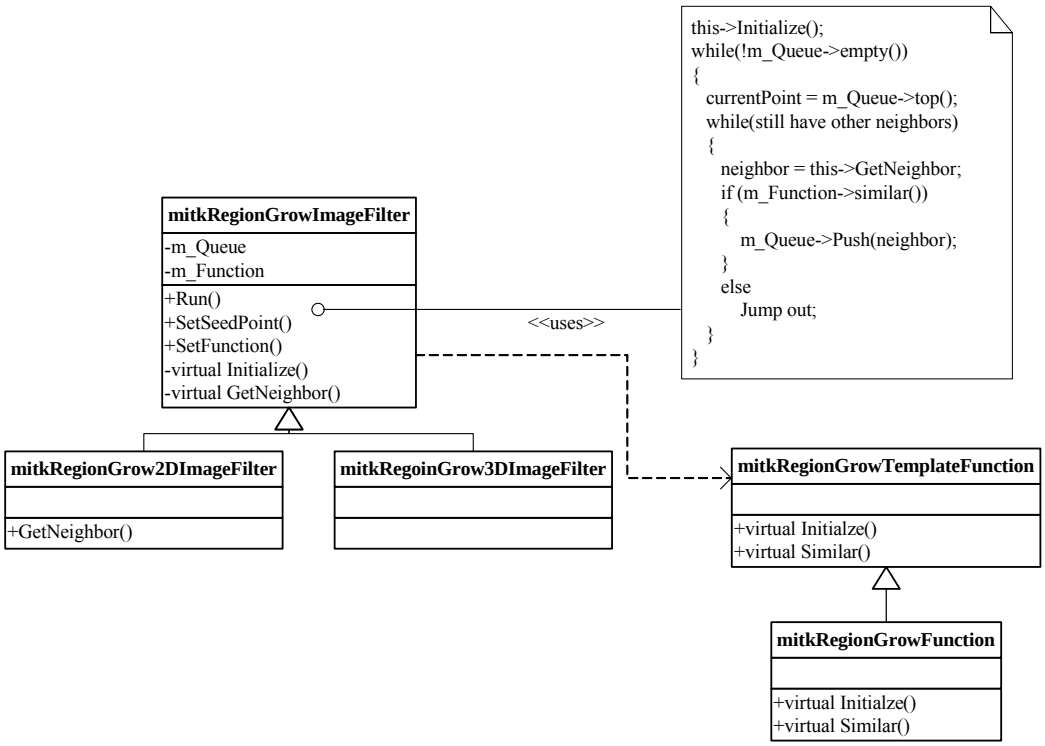



图 6-12 Region Grow 算法开发包内部结构图

如 图 6-12 给出了Region Grow算法开发包的内部结构，算法的输入数据为原始图像，输出数据为分割后的图像。

mitkRegionGrowImageFilter按照 图 6-10 定义了区域生长算法的主框架流程

m_Queue：指向队列的指针，m_Queue 保存了当前活动点；

m_Function：它是一个 mitkRegionGrowTemplateFunctino 类型的指针，用来进行相似度判断，程序运行时，用户需将一个具体的 mitkRegionGrowTemplateFunction 的子类用 SetFunction()函数指定；

*函数 SetSeedPoint()*：指定种子点的位置；

*函数 Initialize()*：主要是做一些初始化工作，初始化队列，将种子点压入堆栈；

*函数 GetNeighbor()*：获取当前点的邻域点，可以由子类重载，以实现不同的获取方式；

*函数Run()*：由用户调用，以启动区域生长算法，如 图 6-12 所示，他起重要调用m_Function以进行相似度判断；

mitkRegionGrow2DImageFilter 实现两维图像的区域生长算法

*函数 GetNeighbor()*：由基类重载而来，获得当前点的八邻域点；

mitkRegionGrow3DImageFilter 实现三维图像的区域生长算法

*函数 GetNeighbor()*：由基类重载而来，获得当前点的四邻域；

mitkRegionGrowTemplateFunction 为区域生长算法的相似度函数定义了一个基类

*函数 Initialize()*：做一些初始化函数，由子类重载；

*函数 Similar()*：判断相似条件是否满足，满足则返回值为真，不满足返回值为假；

mitkRegionGrowFunction：定义了一种具体的判断相似度的方法

*函数 similar()*：详见上一小节，$|gc-gn|<nv$ 且 $|gs-gn|<cv$ 时返回值为真，否则返回值为假；

### 6.3.3 区域生长分割结果

下面以系统界面的形式给出分割结果：

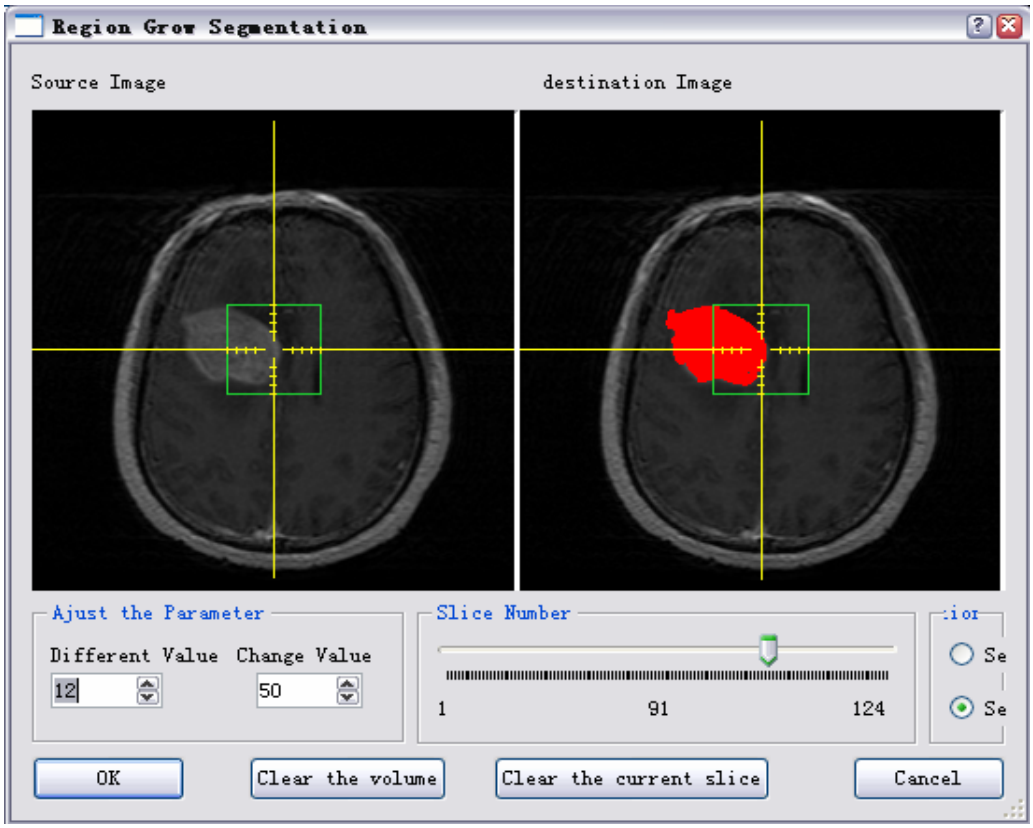



图 6-13 脑肿瘤的二维分割结果

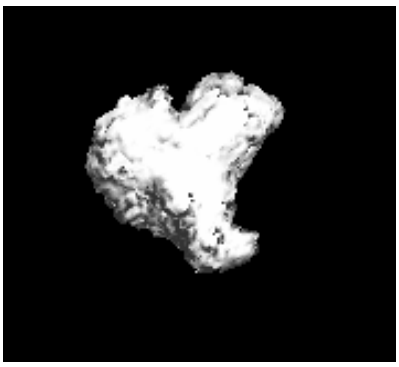


图 6-14 脑肿瘤三维分割重建后的结果

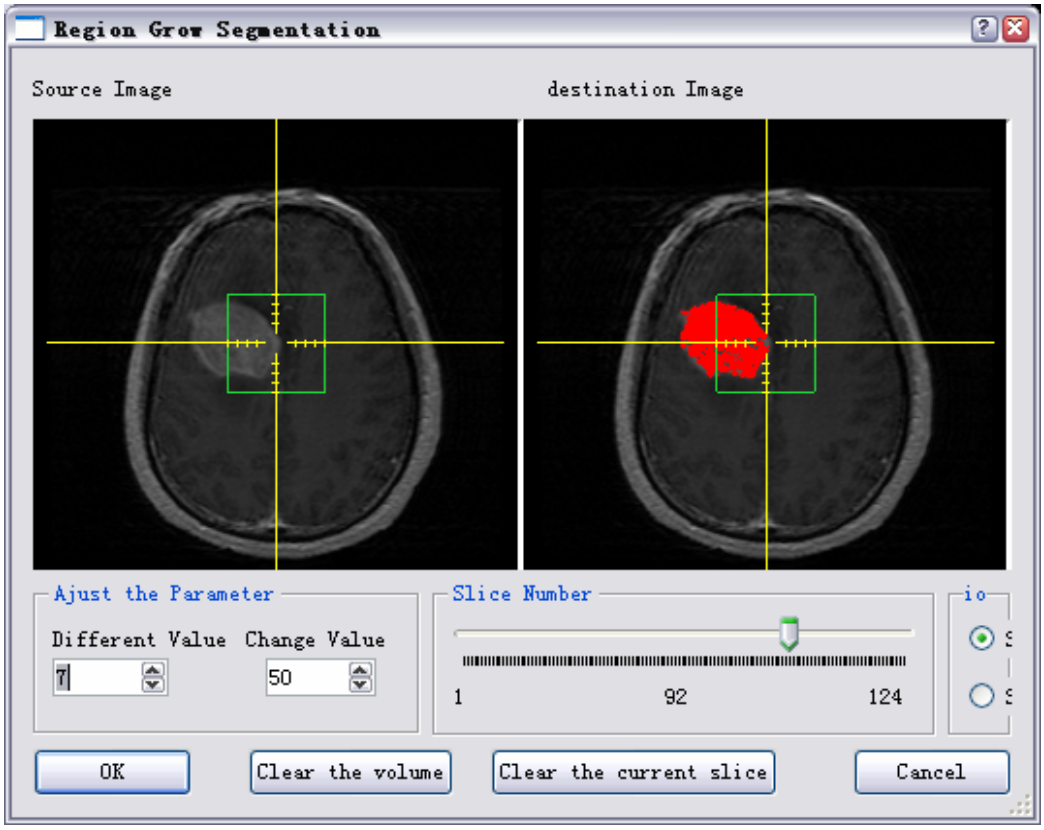



图 6-15 脑肿瘤三维分割中的一张切片

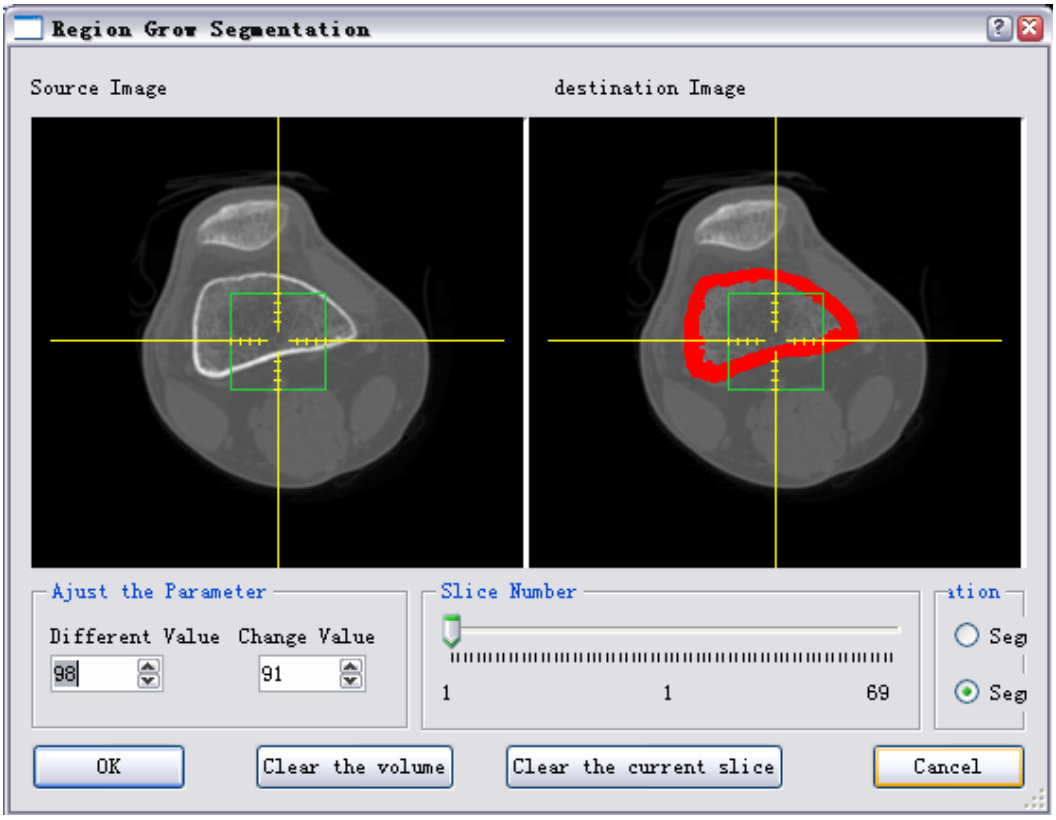



图 6-16 膝盖骨质的二维分割结果

## 6.4 交互式分割在 MITK 中的实现

### 6.4.1 原理概述

本节所指的交互式分割指：由用户指定一些作为多边形的顶点，系统自动将多边形内部填充作为分割结果。填充多边形时要用到图形学中的边填充算法，下面做以简要介绍。

边填充算法的基本思想是：对于每一条扫描线和每条多边形边的交点 $(x_1, y_1)$,将该扫描线上交点右方的所有像素取补。对多边行的每条边做此处理，多边行的顺序随意。如 图 6-17 所示，为应用最简单的边填充算法填充一个多边形的示意图。

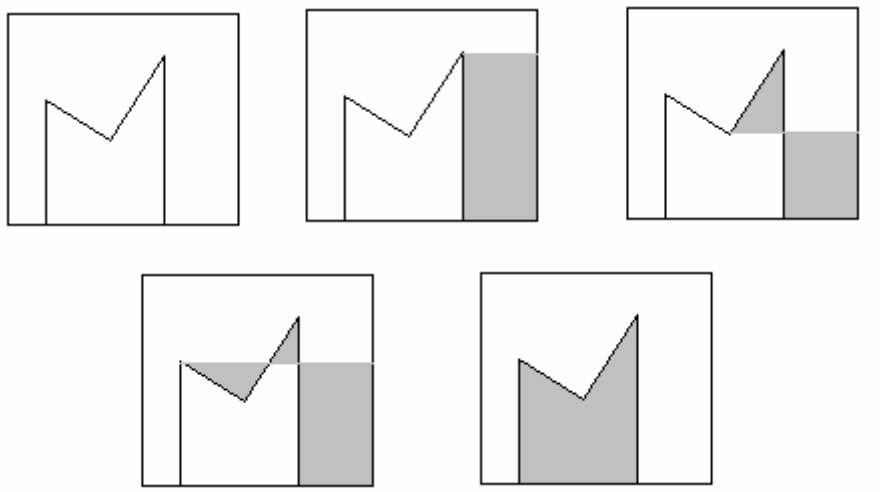


图 6-17 边填充算法示意图

### 6.4.2 交互式分割算法开发包的设计与实现

(1) 算法流程

如 图 6-18 给出了交互式分割算法的算法流程图，算法的输入是原始图像和一组多边形的顶点，输出是分割好的图像。

算法分为三大模块：

初始化部分，该部分主要进行如下操作：

1．初始化图像：将标记图像的所有点初始化为零点；

2．离散化每条边：设扫描线的方向为水平方向，相邻两条扫描线间的间距为图像垂直方向的单位距离。如 图 6-19 所示，求出每条扫描线与多边形各边的交点P1 、P2 、P3、P4…并将他们标值为非零值；

多边形填充部分，该部分按照如下为代码工作：

```
Inside = FALSE;
For (扫描线上的每个像素)
{
    if (该像素值为非零)
        Inside = !Inside;
```

If (Inside == true)

将该像素赋值非零；

}

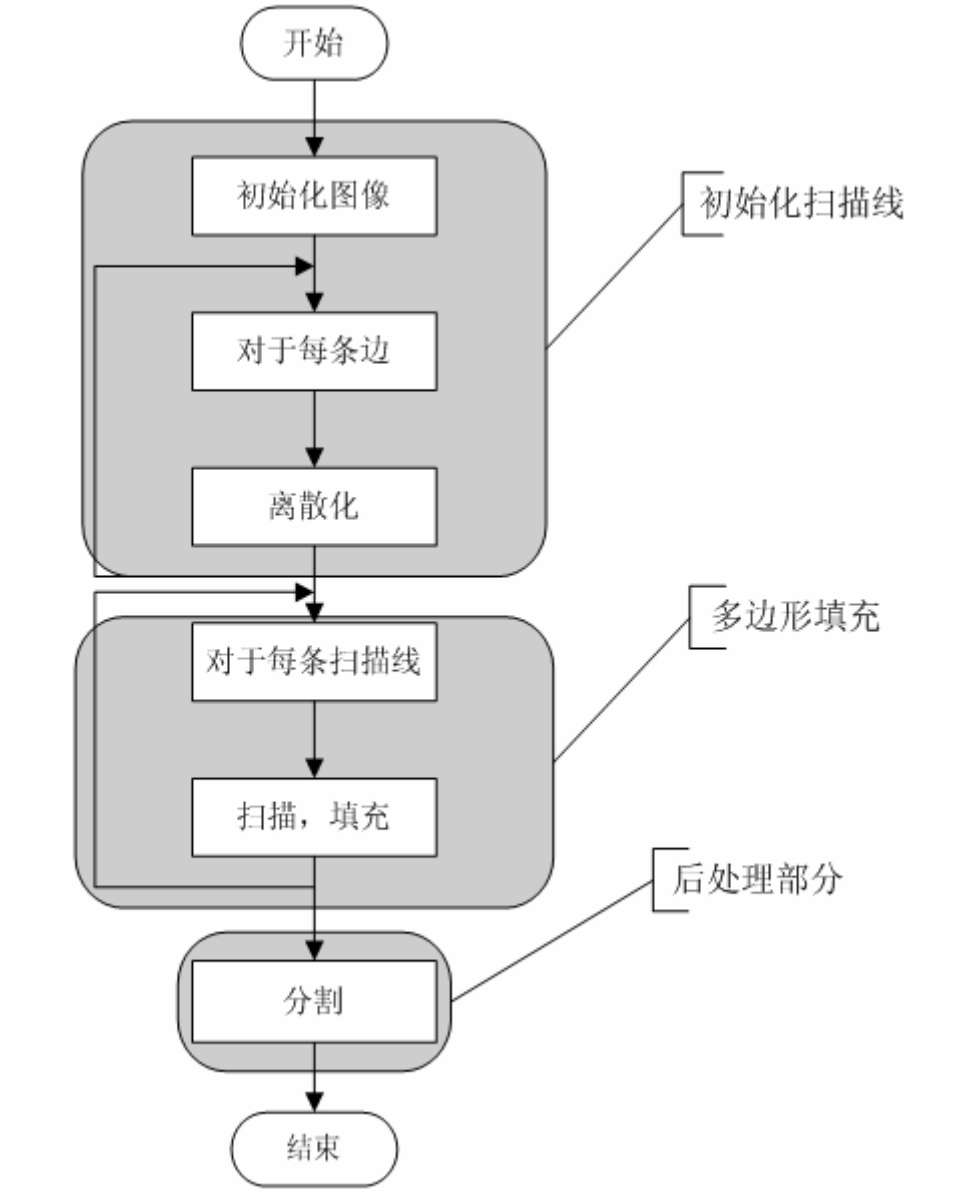



图 6-18 边填充算法流程

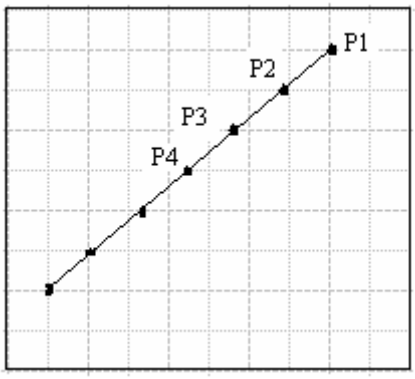



图 6-19 边的离散化

后处理部分，生成分割结果。对于标记图像上的非零点，输出图像的像素点等于输入图像上对应点的值，对于标记图像上的零点，输出图像上的对应点赋值为零；

(2) 交互分割算法开发包的设计与实现

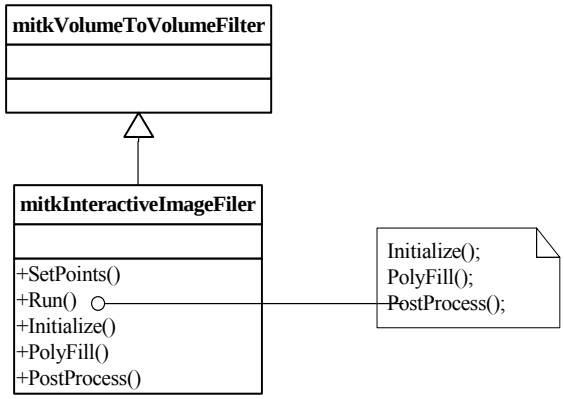



图 6-20 交互分割算法开发包内部结构

如 图 6-20 给出了区域生长算法开发包内部结构图：

mitkInteractiveImageFilter 实现了一种交互式分割算法

*函数 SetPoints()*：用于指定多边形的各个定点；

*函数 Run()*：由用户调用，用于启动交互式分割算法，开始分割。在其中依次调用函数 Initialize()、PolyFill()和 PostProcess()以具体实现图 4.46 内列出的各个步骤；

### 6.4.3 交互式分割算法的分割结果

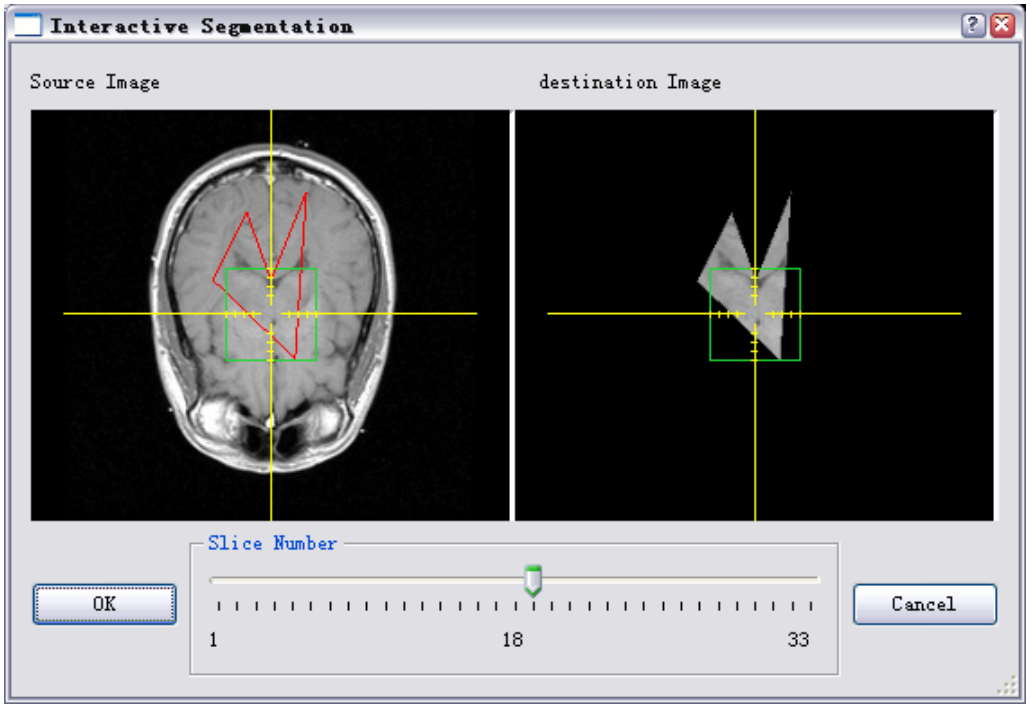



图 6-21 交互式分割结果一

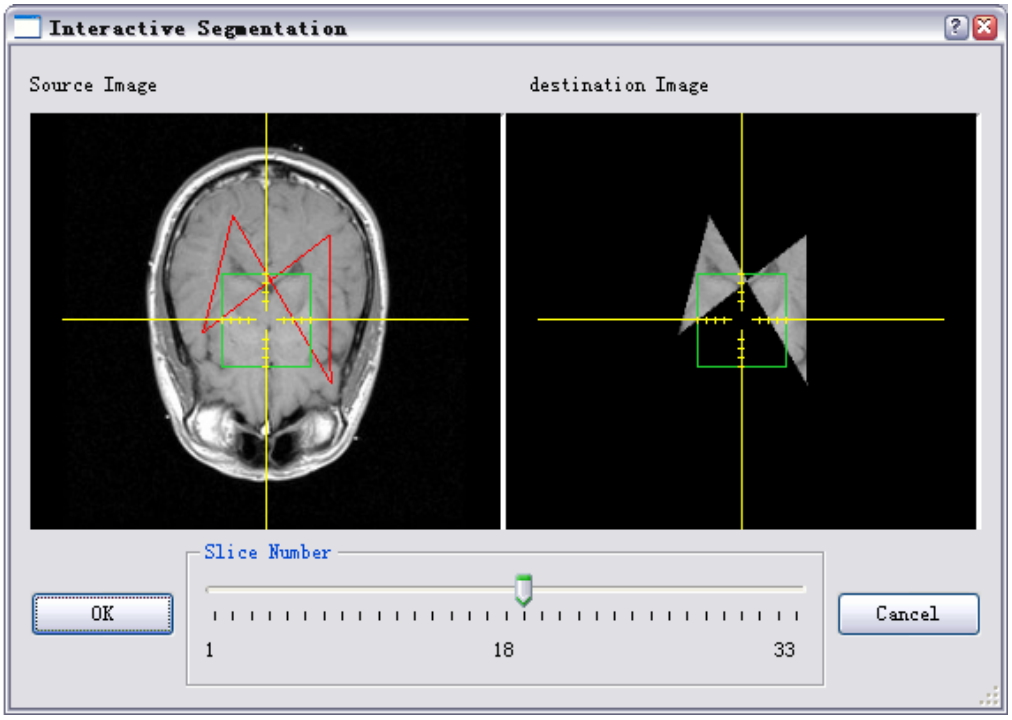



图 6-22 交互式分割结果二

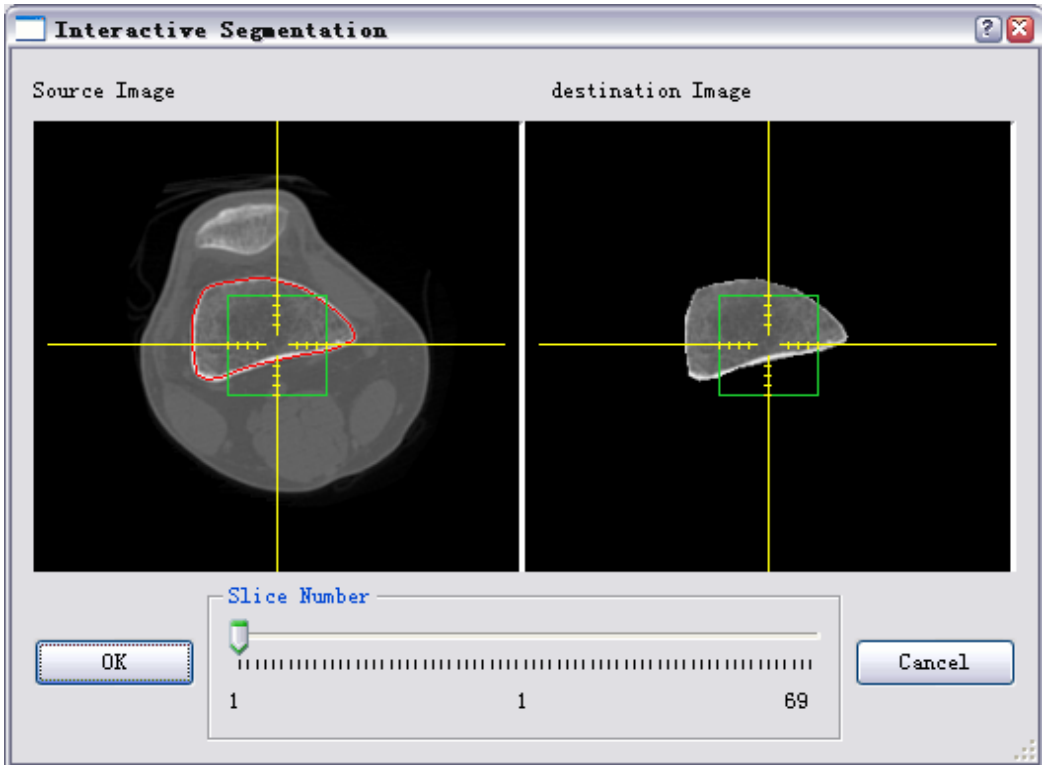



图 6-23 交互式分割结果三

## 6.5 Live Wire 算法在 MITK 中的实现

live wire 算法是用来确定图像中轮廓线的方法，如果想把轮廓线外或轮廓线内的部分分割出来，还要用到种子点填充算法。因此在下面的原理概述和算法包的设计部分，将对他们分别予以阐述。

### 6.5.1 原理概述

(1) live wire 基本原理

live wire 方法属于交互式分割。广义上讲，图像分割可分为两大类：自动分割和交互式分割。自动分割方法可以避免用户的交互，但很难保证他们总是有效的。在交互式分割方法中，有的完全需要有由用户来完成，如那些由用户来话轮廓线的分割方式，有的只需要很少的用户交互，如本节要介绍的 live wire 分割方法。自动分割目前已被广泛采用，但当一些新的图像出现时，他们往往不能顺利工作，这方面还有大量的研究工作要做。在这种情况下，往往应用交

互式分割，它能够使用户完全控制分割过程，而且在任何情况下他都可以正常工作。

在live wire算法中，将图像看成是一个连通图，图像中的像素当作连通图中的节点，相邻像素间的边当作连接节点的边。在边上定义一个代价函数，使强边缘具有较小的代价值，非边缘具有较大的代价值，两个节点间的距离可用代价值表示。然后通过图搜索来找物体的边界，把用户指定的物体边界上的两点之间的最短路径（即该条路径上所有边的代价值总和最小）当作物体的边界。一般用动态规划来查找连通图中两点之间的最短路径，如 图 6-24 所示。

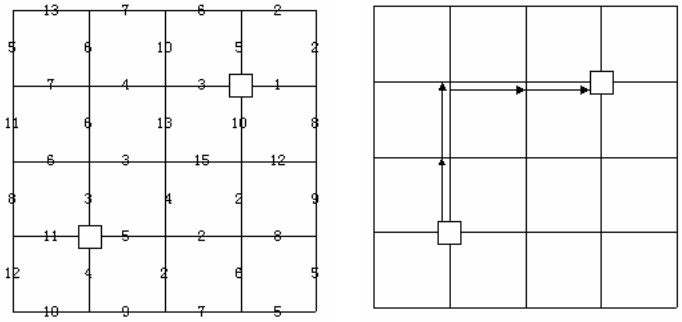


图 6-24 连接两个点之间的最短路径

由上面的分析可知,图像中目标物体的边缘跟踪问题可以被转换成赋权图的最优路径搜索问题。为此，需对一幅 n×n 的灰度图像进行如下处理。

一幅 n×n 的图像被描述成具有 4 邻域象素的象素阵列，每个象素被描述成一个正方形，相邻象素有一条公共边，称为元边。对于 G 中的每一条元边，根据一定的规则赋予其相应的特征值，用以描述该元边属于物体边缘的可能性，元边的特征值经过特征转换函数转变成一定的代价值。我们定义图中任意两个节点间的最优路径，由两个节点间累积代价和最小的连续元边组成。此时，该图像确定了一个赋权图：

G＝（V，E）

其中 V 为图像的象素点集合，E 为元边的集合。

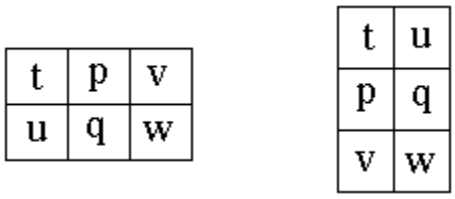



图 6-25 相邻像素

如 图 6-25 所示，相邻象素p与q之间的公共边b，b的特征值根据图像的边缘信息确定，边缘信息越强，特征值越大。利用特征转换函数可以将特征值转换为边的代价值。

文献[9]用如下几式来确定边的特征值，和特征转换函数。

特征值为：

$$f_1 = |g(p) - g(q)| \tag{6-1}$$

$$f_2 = \frac{1}{3}|g(p) + g(t) + g(v) - g(q) - g(u) - g(w)| \tag{6-2}$$

$$f_3 = \frac{1}{2}\left|g(p) + \frac{1}{2}g(t) + \frac{1}{2}g(v) - g(q) - \frac{1}{2}g(u) - \frac{1}{2}g(w)\right| \tag{6-3}$$

$$f_4 = \frac{1}{4}\left(|g(p) - g(u)| + |g(t) - g(q)| + |g(p) - g(w)| + |g(v) - g(q)|\right) \tag{6-4}$$

特征转换函数为：

$$c(f) = \begin{cases} 1, & f \le l_1 + \dfrac{a^2}{2} \\ \dfrac{a^2}{2(f - l_1)}, & l_1 + \dfrac{a^2}{2} < f \le l_2 \\ 0, & f > l_2 \end{cases} \tag{6-5}$$

其中 $a$ 、 $l_1$ 和 $l_2$ 为自由参数，他们分别取如下值：

$$l_1 = \min(f) \tag{6-6}$$

$$l_2 = \max(f) \tag{6-7}$$

$$a = \frac{[l_2 - l_1]}{100} \tag{6-8}$$

目标物体的边缘并不总是强边缘，而且在目标物体边缘的周围可能存在具有强边缘的物体，或存在具有强特征的噪音干扰，采用上述特征转换函数会将某物体的强边缘或噪音误判为目标物体的边缘。因此考虑到高斯函数的特性，采用高斯函数的一种变体作为特征转换函数：

$$c_2 = 1 - e^{-\frac{(f-mean)^2}{2\sigma^2}} \tag{6-9}$$

其中 *mean* 表示赋权图中期望的元边的均值，σ 表示赋权图中期望的元边的方差 。这样，对于具有不同特征值的期望边缘，其代价值都可取得较小值。

特征转换函数$c_1$ 、$c_2$ 的粗略函数曲线图如 图 6-26 所示：

由 图 6-26 中$c_2$的粗略函数曲线图可知，当元边特征值为均值*mean*时其代价值最小。

因此在分割前用户通过鼠标交互式的选定特征较弱的物体边缘区域作为训练区域，并以该训练区域的样本梯度均值和样本梯度方差分别作为特征转换函数 $c_2$的 *mean* 值和 σ 值，这样即使目标物体边缘存在具有强特征的噪音，但因为期望的物体边缘代价值较小，分割不再会受噪音的干扰而导致错误的结果。

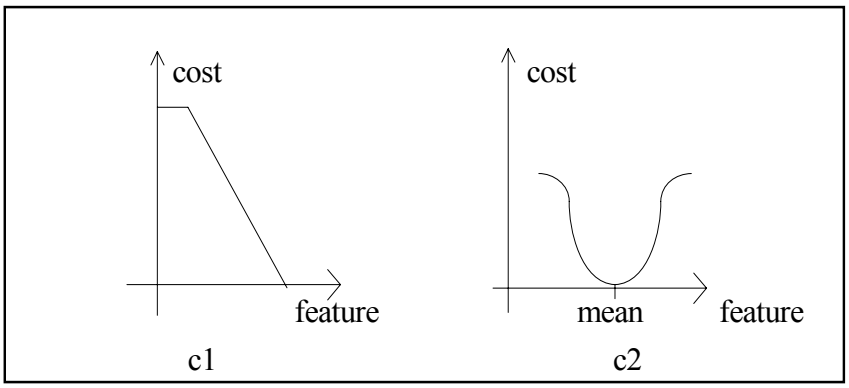



图 6-26 代价函数

(2) 种子点填充基本原理

种子点填充是图形学中的算法，是轮廓提取算法的逆过程。它首先假定封闭轮廓线内某点是已知的，然后算法开始搜索与种字点相邻且位于轮廓线内的点。如果相邻点不再轮廓线内，那么就到达轮廓线的边界；如果相邻点位于轮廓线之内，那么这一点就成为新的种子点，然后继续搜索下去。种子点填充区域的连通情况又有四连通和八连通之分。

### 6.5.2 live Wire 算法包的设计与实现

(1) 算法流程

**live wire 算法流程**

由上面的介绍可知，live wire 算法的关键在于如何确定边的特征值和特征转换函数。在 live wire 算法包中，采用式(6-1)-(6-4)来确定边的特征值，采用式(6-5)来确定边的特征转换函数。

如 图 6-27 给出了live wire算法的框架图。他的输入是原始图像，输出是一幅二值图，从初始点到终止点的最优路径上的点为非零值，其他的点为零值。有图中可见，该算法主要分为两大模块，初始化部分和动态规划部分：

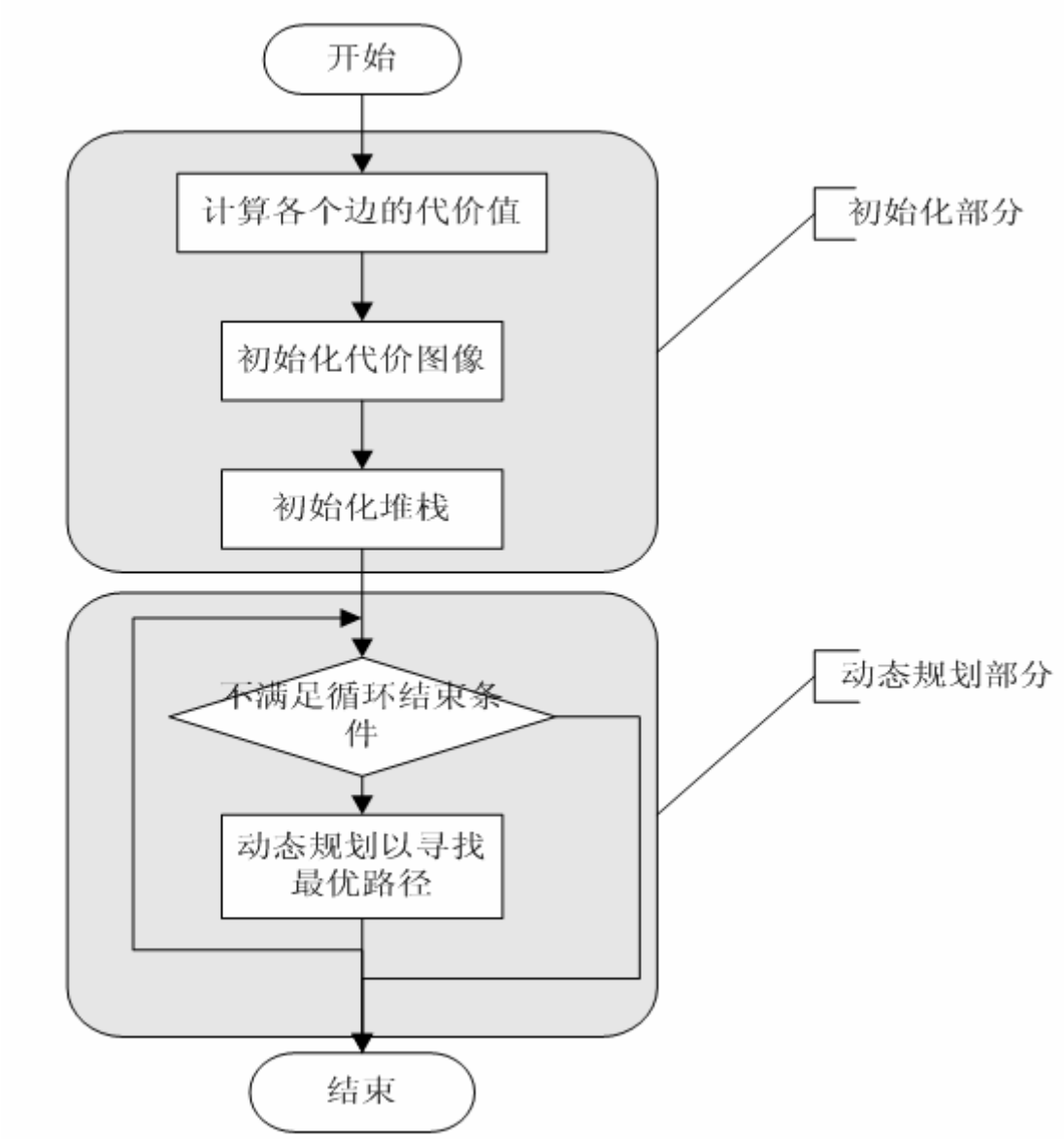



图 6-27 live wire 算法框架图

初始化部分：该部分主要完成如下工作

计算由图 6-26 确定的每条边的代价值$c(b)$；

用与输入图像大小相同的一幅图像记录图像中各个点的累积代价值。成这幅图像为代价图像。在代价图像中，将初始节点 $v_s$ 的累积代价值 $cc$ ($v_s$)设为 0，其余节点的累积代价值为∞；

初始化队列 Q，把初始节点 $v_s$ 置入待处理的节点队列 Q 中；

动态规划部分：该部分主要完成如下工作

设置循环终止条件：当堆栈为空时循环终止

从 Q 中移出一个累积代价值最小的节点 v，并将 v 置入队列 L；

对于 v 的每一个 4 邻节点$v',v'\notin L$，计算$cc(v)+c(b)$ 其中$b'$为节点$v'$到节点 v 的元边

如果 $cc(v')>cc(v)+c(b')$ 且 $T_{CC} > cc(v) + c(b')$ 则使 $cc(v')=cc(v)+c(b')$，并将节点 v 的方向信息保存在$dir(v')$中

如果$v'\notin Q$ 则将$v'$插入到 Q 中

如果$v'$点为终止点则跳出循环

从终止点开始，根据$dir(v')$的纪录，找到最优路径。

**种子点填充算法流程**

对于种子点填充算法，我们可以用堆栈的方法，对边界定义的区域进行填充。基本流程是：

种子点像素压入堆栈；

当堆栈非空时，从堆栈中推出一个像素，并将该像素设置成所要的值；

对每个于当前像素邻接的得四连通或八连通像素，进行上述两部分内容的测试；

若所测试的像素在区域内没有被填充过，则将该像素压入堆栈；

(2) **类协作图**

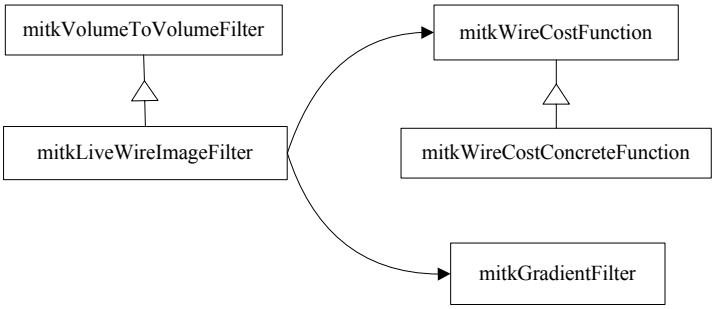



图 6-28 live wire 算法包类协作图

如 图 6-28 给出了live wire算法包的类协作图。mitkLiveWireImageFilter定义了live wire算法的主框架流程，也就是一个动态规划过程；mitkWireCostFunction为所有计算边代价值的方法提供了一个基类；mitkLiveWireCostConcreteFunction提供了一种计算边代价值的方法；mitkGradientFilter用来计算图像的梯度场。mitkLiveWireImageFilter拥有指向mitkWireCostFunction和mitkGradientFilter的指针。

(3) 类内部组成结构详解

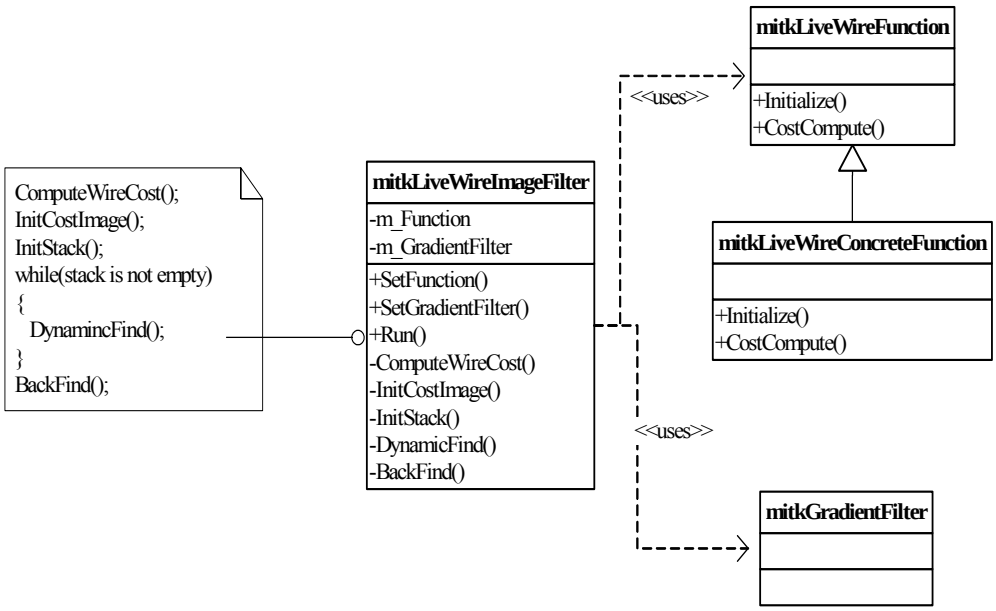



图 6-29 类内部组成结构详解

如 图 6-29 给出了live wire算法包的内部组成结构。

mitkLiveWireFunction：他为所有计算边代价的数学方法提供了一个基类。用户通过为他添加不同的子类以实现不同的计算边代价的方法。

*函数 Initialize()*：主要做一些初始化工作

*函数 CostCompute()*：计算每条边的代价值

这两个函数都为虚函数，由子类重载。

mitkLiveWireConcreteFunction：由 mitkLiveWireFunction 继承而来，根据式 6-18—6-19 实现了一种计算边代价的方法

mitkLiveWireImageFilter：依据 图 6-27 定义了live wire主框架流程，主要是定义了动态规划的实现过程。它包含了指向 mitkLiveWireFunction 和 mitkGradientFilter的指针m_Function和m_GradientFilter，用以计算边代价值和求图像的梯度场。

*函数 SetFunction()* 用来制定所采用的 function，这些 function 都是 mitkLiveWireFunction 的子类。

*函数 SetGradientFilter()*用来指定所采用的求梯度的方法。

*函数Run()*为由用户调用，用来启动mitkLiveWireImageFilter，在这个函数中，一次调用与 图 6-27 对应的各个虚函数，以实现动态规划过程。

*函数 ComputeWireCost()*用来计算边代价值：首先用 m_GradientFilter 计算图像的梯度场，然后用 m_Function 计算各个边的代价值。

*函数 InitCostImage()*用来初始化代价图像。

*函数 InitStack()*用来初始化堆栈。

*函数 DynamicFind()*用来实现动态规划的过程。

*函数 BackFind()*从终止点向前回溯，找到最优路径。

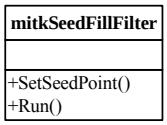



图 6-30

如图对于种子点填充算法，用类 mitkSeedFill 来实现。输入图像是一幅二值图，轮廓线上的点非零，其他点为零。点进行填充，输出图像也为二值图，被填充的部分值为非零，其他部分为零值。

### 6.5.3 live wire 分割结果

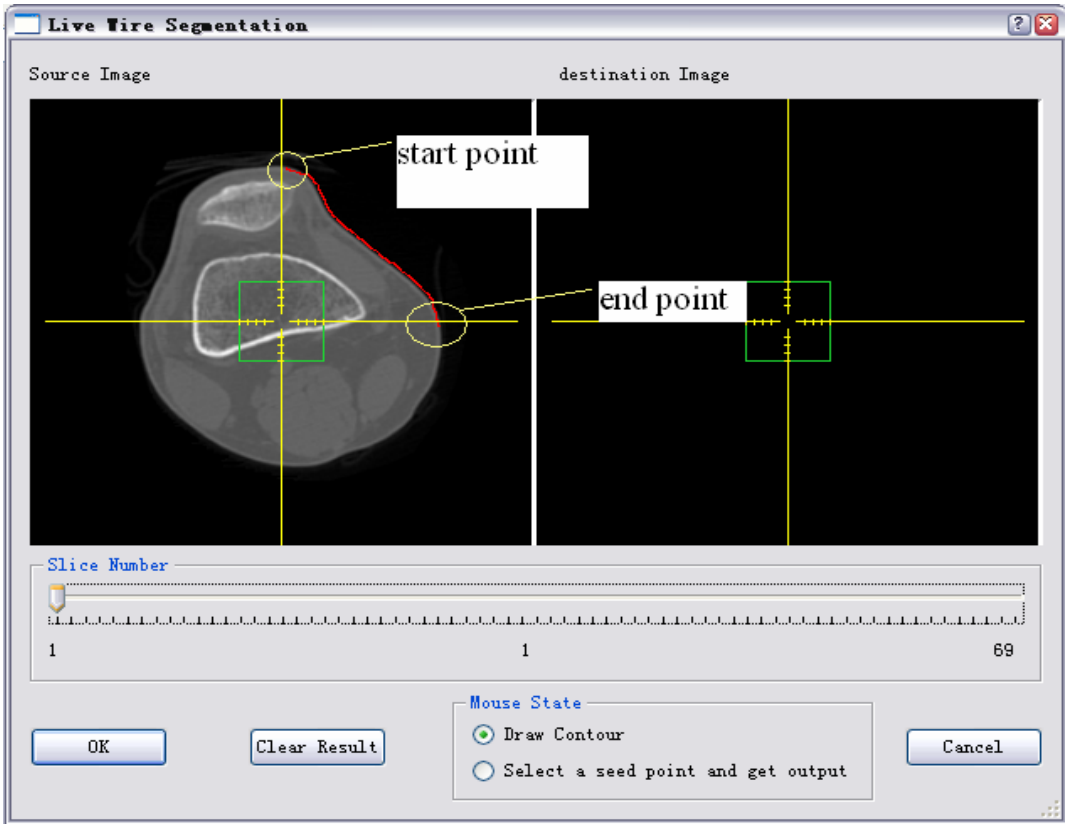



图 6-31 给定起始点和终止点下找出的轮廓线

图 6-31 给出了在如图所示的起始点和终止点下，live wire算法找出的轮廓线；

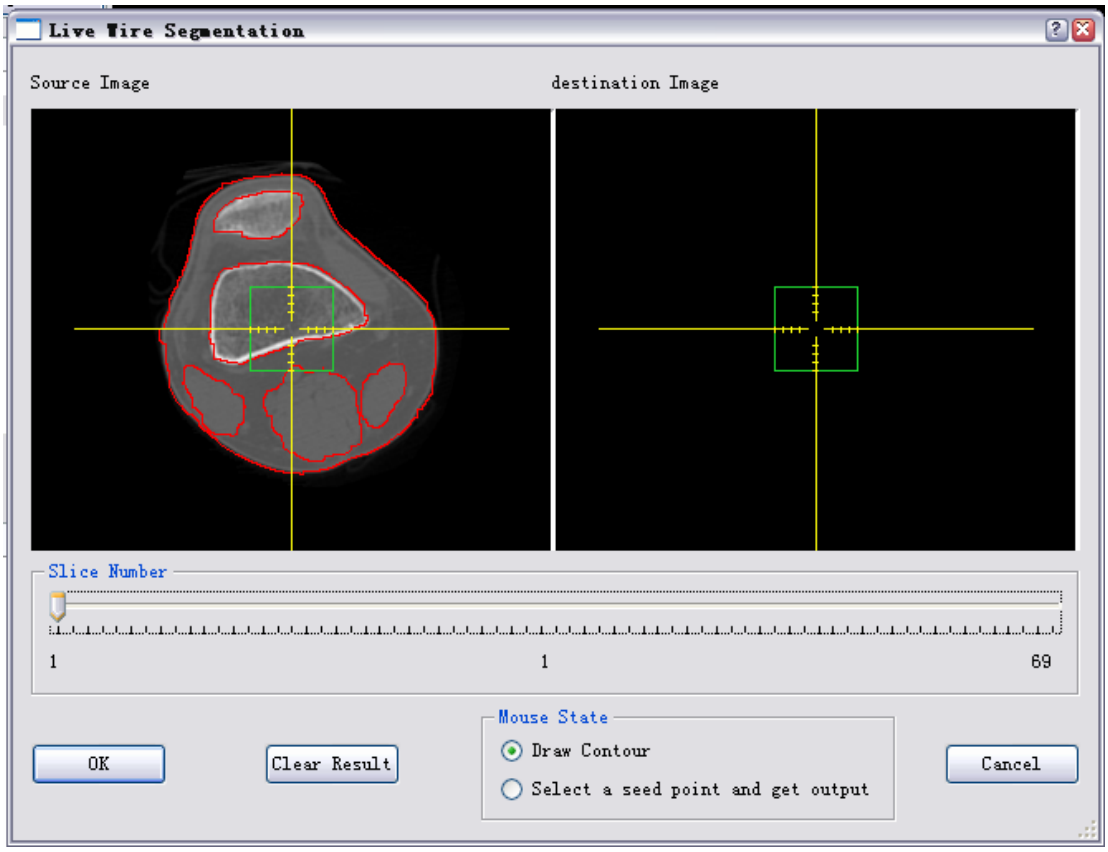



图 6-32 用 live wire 算法生成的几条轮廓线

图 6-32 给出了几条用live wire算法找出的轮廓线；

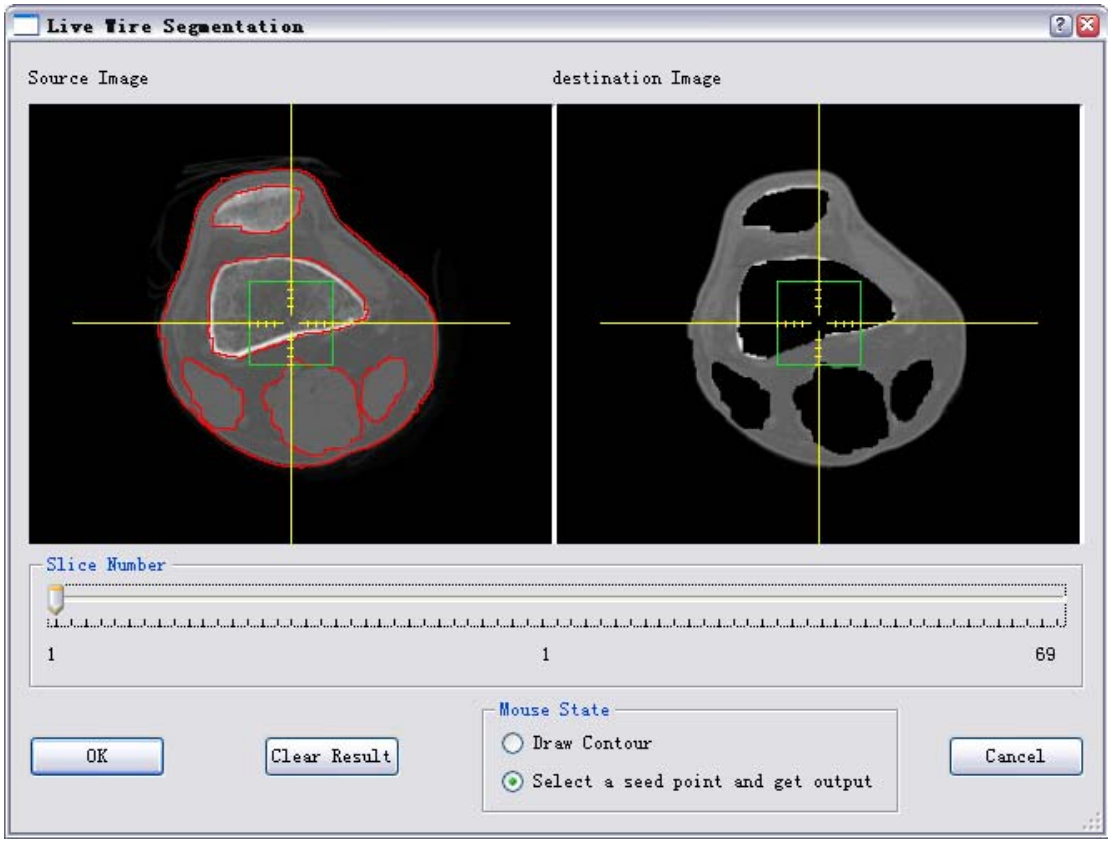



图 6-33 对轮廓线进行种子点填充后的分割结果

图 6-33 给出了种子点选在各轮廓线间时的种子点填充结果。

## 6.6 Fast Marching 算法在 MITK 中的实现

### 6.6.1 原理概述

Fast Marching算法是一种基于几何形变模型的医学影像分割方法。几何形变模型是由Irasel的Caselles 等 [10]和 Florida的Malladi等[11]提出的，这些模型的理论基础是曲线演化（curve evolution）理论 [12] [13] [14] [15]和水平集（level set）方法[16] [17]。几何形变模型的基本思想是将曲线的形状变化用曲线演化理论来描述，即用曲率或法向量等几何度量表示曲线或曲面演化的速度函数，并将速度函数与图像数据关联起来，从而使曲线在对象边缘处停止演化。由于曲线的演化与参数无关，几何形变模型能被自动处理对象拓扑的变化，演化过程中的

曲线和曲面只能被隐含表示为一个更高维函数的一个水平集，因此曲线演化过程采用了水平集方法加以实现。另一种跟踪运动的曲线或曲面的方法是固定曲线或者曲面的演化方向，也就是说，曲线或者曲面只能收缩或者扩张，这就是Fast Marching方法。

(1) **曲线进化理论**

曲线演化理论的研究目的是利用几何度量描述曲线的形状随着时间的变化，几何度量如单位法向量和曲率，而不是一些与参数有关的数量（如任意参数曲线的导数），如 图 6-34 所示。试想一条运动中的曲线 $X(s,t)=[X(s,t),Y(s,t)]$，其中$s$是任意参数，$t$是时间变量，$N$是向内的单位法向量，$k$是曲率。曲线沿着法线方向的形变可用以下偏微分方程描述：

$$\frac{\partial X}{\partial t}=V(k)N \tag{6-10}$$

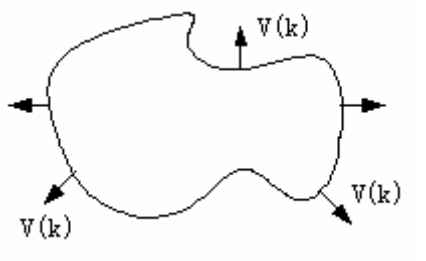



图 6-34

曲线演化其中 $V(k)$ 被称为速度函数（*speed functio*n），因为它决定了曲线演化的速度。注意，曲线沿任意方向的运动都可以重新参数化后再用公式(6-10)表示[15]。此事实的直观解释就是，切线方向的形变只影响曲线的参数，而不改变形状和几何特征。在曲线形变理论中研究得最多的是曲率形变（*curvature deformation*）和常数形变（*constant deformatio*n）。

曲率形变用一个几何热方程（geometric heat equation）表示：

$$\frac{\partial X}{\partial t}=\alpha kN \tag{6-11}$$

其中 $\alpha$ 是一个正常数。此方程将使一条曲线平滑并最终收缩为一个圆点。使用曲率形变的效果类似于在参数形变模型中使用弹性内力。

常数形变表示为：

$$\frac{\partial X}{\partial t} = V_0 N \tag{6-12}$$

其中$V_0$是决定形变速度和方向的系数。常数形变所起的作用与参数形变模型中的压力相同。

曲率形变和常数形变的性质是互补的，曲率形变通过平滑曲线除去了奇异点，而常数形变可以在初始平滑曲线上创造奇异点。

(2) Fast Marching 方法

考虑一种界面运动的特殊情况，那就是界面的运动速度$F>0$或者$F<0$，也就是说运动的界面或者只能进行扩张，或者只能进行收缩。如 图 6-35 所示，图中的$F>0$，运动的界面—蓝色的圆圈只能向外扩张，而不能收缩。

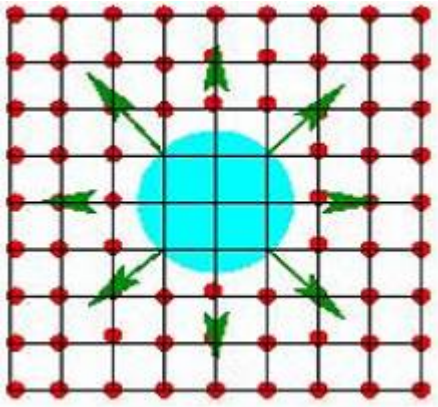


图 6-35 运动的界面

假定 $T$ 是界面经过一个指定点$(x, y)$的时间，这样 $T$ 就满足如下的方程：

$$\left|\nabla T\right| F = 1 \tag{6-13}$$

这个公式简单的说明了到达时间的梯度和界面的运动速度成反比。从广义上说，有两种方法可以用来近似运动的界面随时间变化的位置：一种是通过迭代和数字近似公式微商来解决；另一种是构建公式(6-13)中到达时间 $T$ 的解决方案。而 Fast Marching 方法依赖于后一种方法。

### 6.6.2 Fast Marching 算法开发包的设计与实现

(1) Fast Marching 的算法流程

公式(6-13)是著名的Eikonal方程的一种形式，Sethian在文[19]中指出，要得

到公式 6-14)中的到达时间$T$，等价于求解下面的二次方程，有关它的具体的近似解决方案请参见文[20] [21]。

$$\left[\begin{array}{l}\max(D_{ij}^{-x}T,0)^2+\min(D_{ij}^{+x}T,0)^2+\\ \qquad\qquad\max(D_{ij}^{-y}T,0)^2+\min(D_{ij}^{+y}T,0)^2\end{array}\right]^{1/2}=1/F_{i,j} \quad (6\text{-}14)$$

这里$D^-$和$D^+$分别是后向差分和前向差分算子。

$$\begin{cases}D^{+x}T=\dfrac{T(x,y+1)-T(x,y)}{2} \ \&\ D^{-x}T=\dfrac{T(x,y)-T(x,y-1)}{2}\\ D^{+y}T=\dfrac{T(x+1,y)-T(x,y)}{2} \ \&\ D^{-y}T=\dfrac{T(x,y)-T(x-1,y)}{2}\end{cases} \quad (6\text{-}15)$$

下面给出分割算法包中该算法的主框架流程

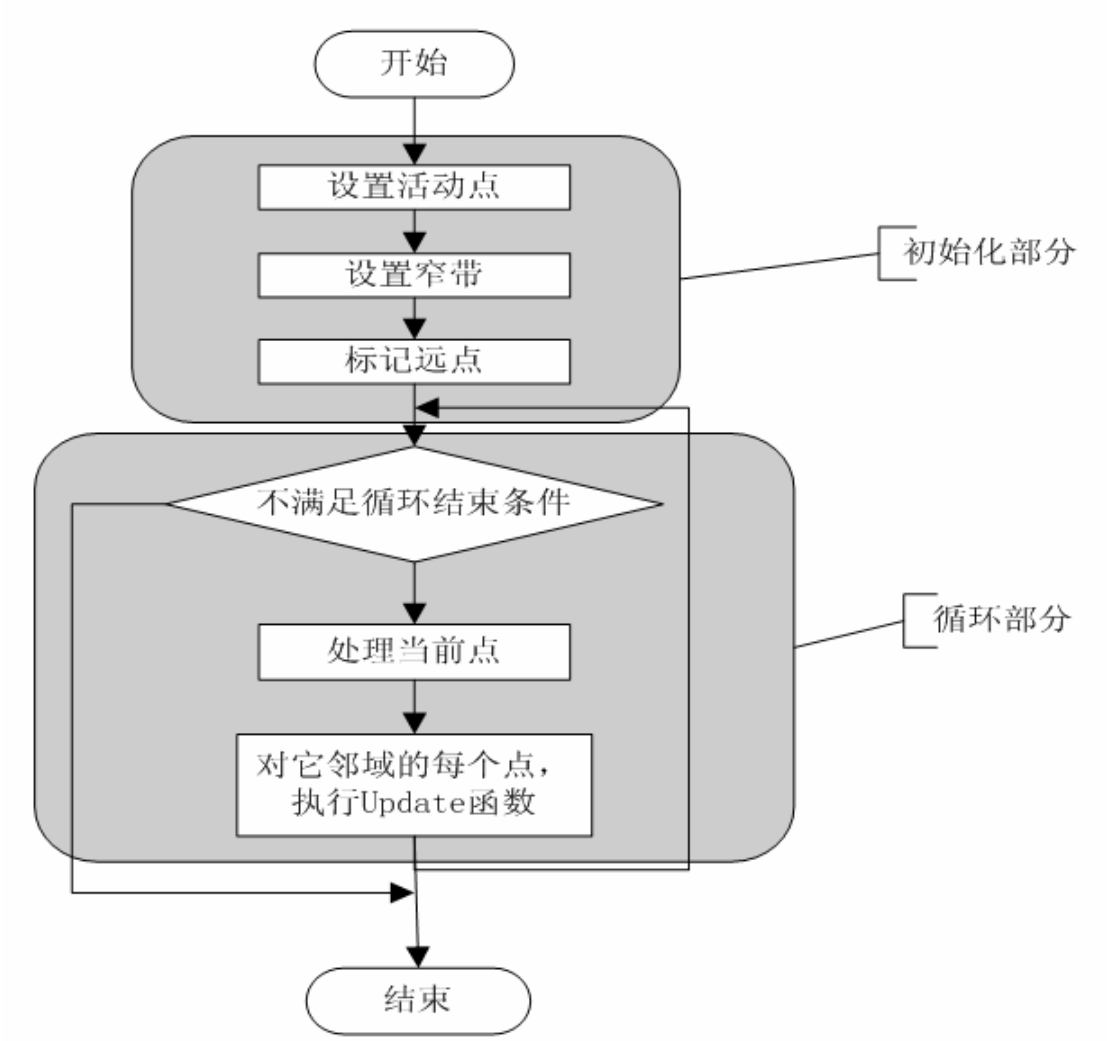



图 6-36 Fast Marching 算法主框架流程

该算法的输入为一幅速度图像，该图像上每个像素点的值代表轮廓线在该点的扩散速度；输出为一幅时间图，该图像上每个像素点的值代表轮廓线扩散到该点所需的时间。如图 6-36 所示，算法主要分为两大模块：初始化部分和循环部分。

初始化部分：

设置活动点：活动点就是所有网格点中时间 $T$ 固定的点。可以有多种指定方法，在分割算法包目前提供的算法中，也就是用户指定的种子点，时间 $T(x, y)=0$。

设置窄带：窄带是指活动点附近的点。在运算时只更新窄带中的点，以提高计算速度。在分割算法包目前提供的算法中，也就是所有种子点的邻接点，时间 $T(x, y)= 1/F(x, y)$。

标记远点：除了活动点和窄带点外，所有其它的网格点为远离点，$T(x, y)= TIME\_MAX$

循环部分：

循环结束条件：可以有多种结束条件，由具体的子类指定。此处采用的结束条件为，当当前像素的时间值大于给定的停止时间值时停止循环；

处理当前点：具体处理方法由子类定义。本节中该步骤的主要工作是，从最小化堆栈中取出栈顶元素，检查它的合法性，并获得它的每个邻点。

处理邻域点：更新当前点的每个邻点的值，具体更新方法由子类指定。

(2) **类协作图**

分割算法开发包中 Fast Marching 算法部分主要类的协作图如下：

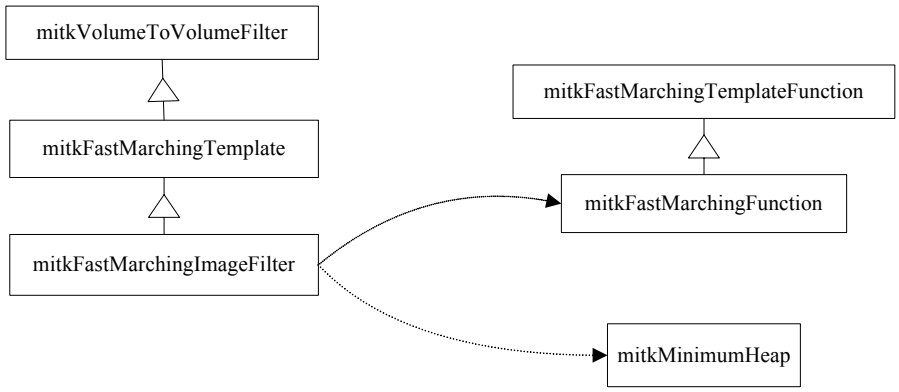



图 6-37 Fast Marching 算法部分主要类协作图

mitkFastMarchingTemplate 定义了图 6-37 所示的 Fast Marching 算法的主要流程框架；mitkFastMarchingImageFilter 是 mitkFastMarchingTemplate 的一个子类，它定义了 Fast Marching 主框架的一种具体实现方法；mitkFastMarchingTemplateFunction 为解方程 6-13 的所有方法提供了一个基类；mitkFastMarchingFunction 定义了一个解方 6-13 的一种具体方法；mitkMinimumHeap 是这部分中很重要的一个类，它定义了一个最小堆栈，它自

动对栈内的所有元素排序，以保证每次弹出的元素都是栈内最小的元素。

(3) 类内部组成结构详解

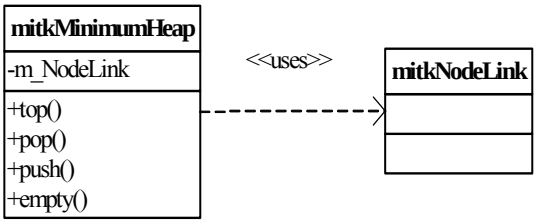



图 6-38 mitkMinimumHeap 主要接口

图 6-38 给出了类mitkMinimumHeapde 的内部结构。

mitkNodeLink：定义了一个链表，它的每个节点包含了该点的坐标值和时间值

mitkMinimumHeap：它定义了一个最小化堆栈，自动对栈内元素排序，保证栈顶元素为站内最小的元素，堆栈内部的排序过程对用户来说是不可见的：

*m_NodeLink* 是一个 mitkNodeLink 类型的成员变量，对栈中的所有元素都是以链表形式存贮的。

*函数 push()*：将一个节点压入堆栈中；

*函数 top()*：用来得到堆栈中值最小的节点即栈顶元素；

*函数 pop()*：将栈顶元素弹出栈；

*函数 empty()*：用来判断堆栈是否为空。

图 4.6 给出了 fast marching 算法包中其他类的内部结构。

mitkFastMarchingTemplate 定义了算法的主要流程，它的成员函数分别对应于图 4.3 中的每一个步骤

*函数 SetActivePoints()*：用来为算法设置活动点；

*函数 SetTrialPoints()*：用来为算法设置窄带；

*函数Run()*：由用户调用，用于启动fast marchig算法。如 图 6-39 所示，在Run()函数中，他分别调用虚函数ProcessActivePoints()、ProcessTrialPoints()、

ProcessFarpoints()、Halt()、GetCurrentNode()、ProcessCurrentNode()和Update()。这些虚函数在mitkFastMarchingTemplate中没有定义，子类通过重载这些函数来来完成改算法的一种实现；

mitkFastMarchignTemplateFunction 为所有的 function 提供了一个基类

函数 Update()：用来更新当前点的时间值，为虚函数，由子类重载，以实现一种数学方法；

mitkFastMarchingFunction 为 mitkFastMarchingTemplateFunction 的一个子类

函数 Update()：定义了一种解方程 6-13 的方法

mitkFastMarchingImageFilter 是 mitkFastMarchingTemplate 的一个子类，重载了它的一部分虚函数：

m_NodeHeap：是一个 mitkMinimumHeap 类型的指针

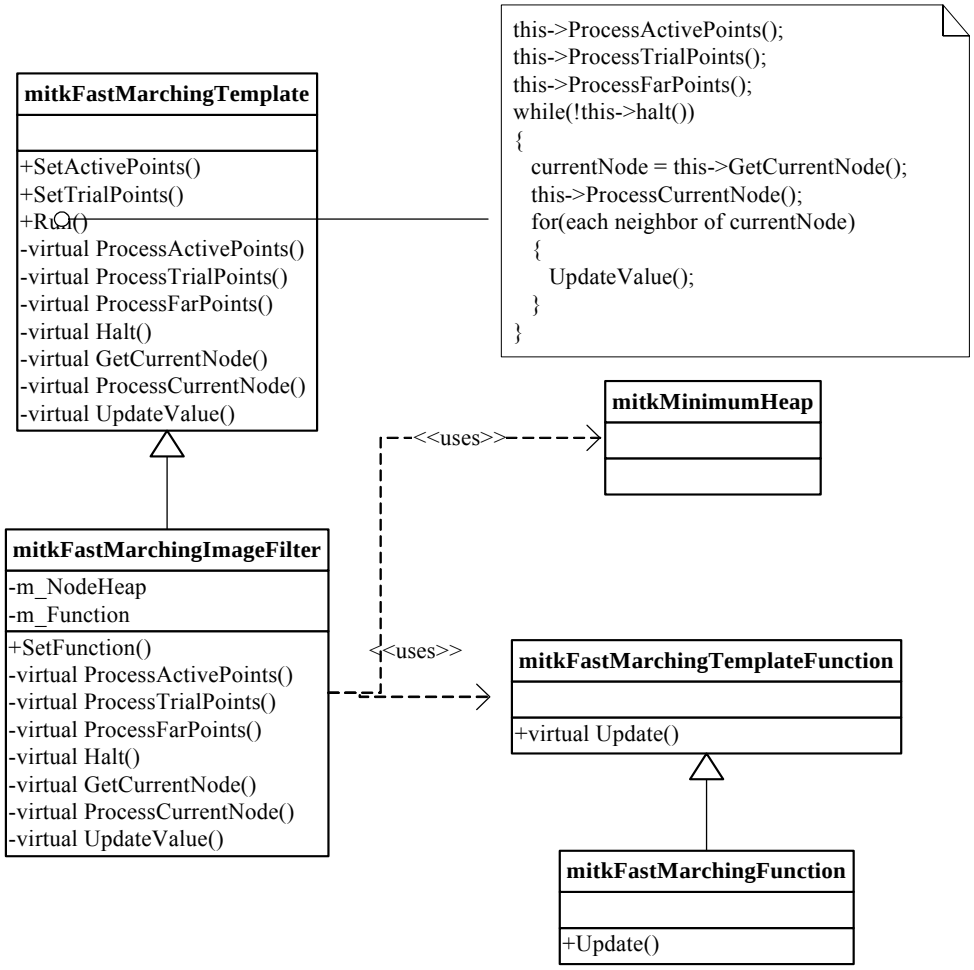



图 6-39

*m_Function*：是一个 mitkFastMarchingTemplateFunction 类型的指针

*SetFunction()*：用来为 m_Function 指定一个 function 类型，它必须是 mitkFastMarchingTemplateFunction 的子类；

*ProcessActivePoints()*： 将活动点的 $T(x, y)$值设为零

*ProcessTrialPoints()*：将窄带内点的 $T(x, y)$ 值设为 $1/F(x, y)$

*ProcessFarPoints()*： 将既非活动点也非窄带内的点的 $T(x, y)$值设为 *TIME_MAX*

*Halt()*：当 m_NodeHeap->empty() == true 即最小堆栈为空时，循环结束。

*GetCurrentNode()*： 返回 m_NodeHeap 的栈顶元素

*ProcessCurrentNode()*： 更新当前点，如果当前点的值大于已经设定的停止值，跳出循环。

*UpdateValue()*：调用 m_Function，计算各邻域点的值。

如图 6-40 所示，分割算法开发包最终提供给用户的类是 mitkFastMarchingPacketFilter，它里面封装了三个类：mitkSpeedFilter 将用户输入的原始图像根据它们的梯度图像转化为速度图像；mitkFastMarchingImageFilter 对速度图像执行 FastMarching 算法；mitkTimeTransferFilter 将时间图像转化为要输出的分割结果。

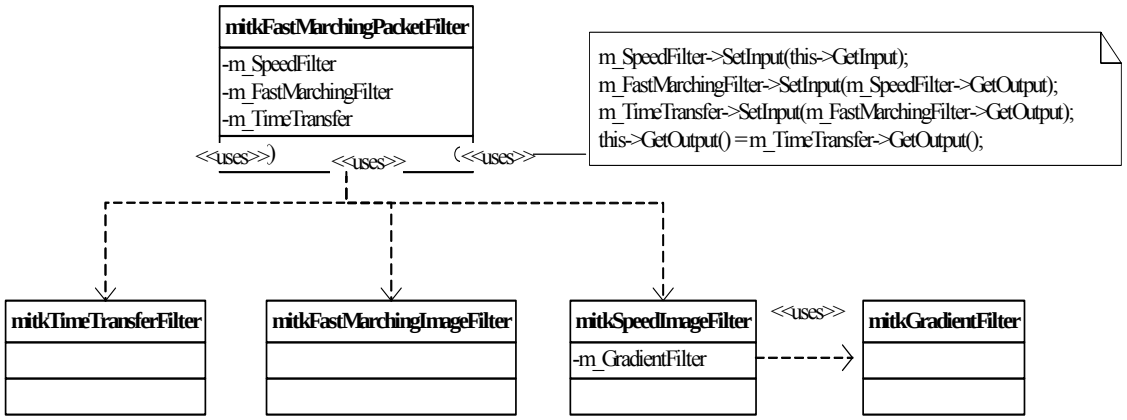



图 6-40 封装后的 Fast Marching Filter

### 6.6.3 Fast Marching 分割结果

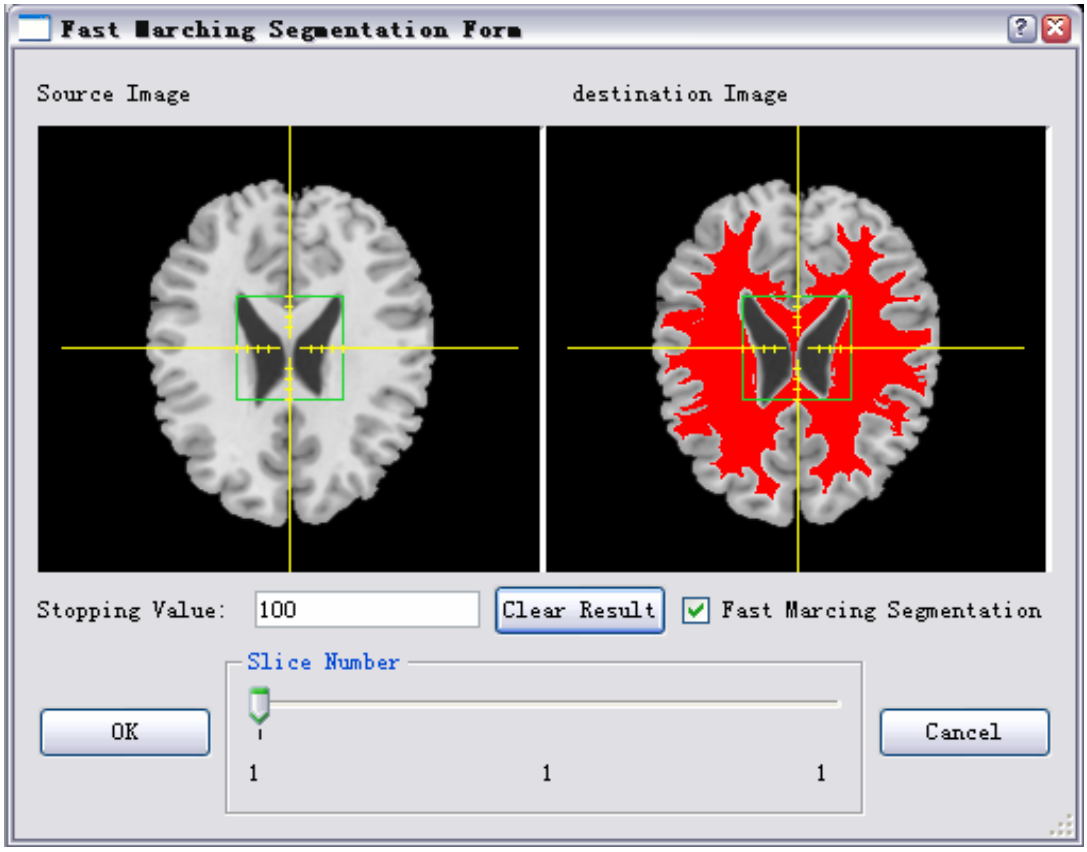



图 6-41 Fast Marching 分割结果一

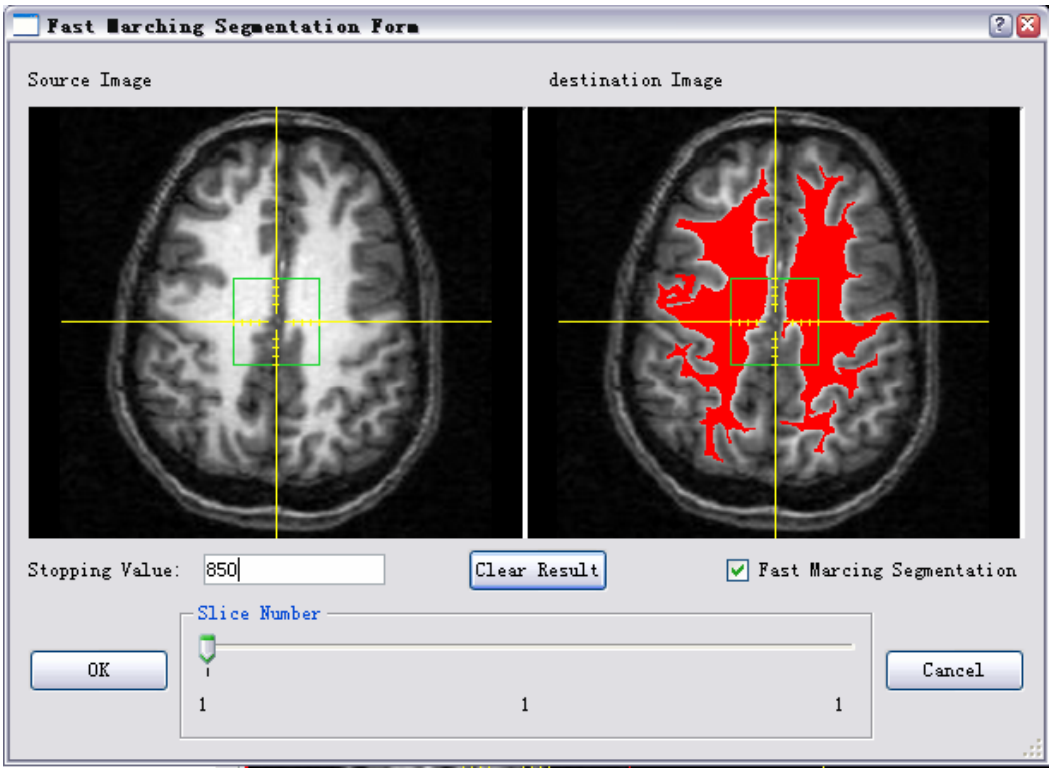



图 6-42 Fast Marching 插件分割结果

## 6.7 Level Set 算法在 MITK 中的实现

### 6.7.1 原理概述

下面介绍曲线演化的水平集实现方法。水平集方法用于解决拓扑的自动变化，它也提供了几何形变模型的数学实现基础。Osher and Sethian最先将水平集方法用于实现曲线的演化[16] [21] [22]。

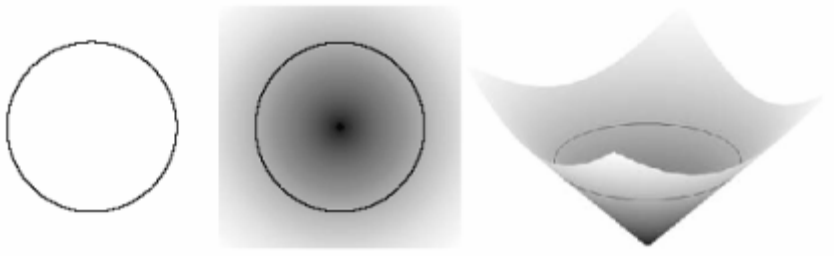


图 6-43 将一条曲线嵌入为一个水平集

在水平集方法中，曲线隐含表示为一个更高维曲面函数的一个水平集（level set），该高维函数称为水平集函数（level set function），其定义域通常为图像空间。水平集是由那些水平集函数值相等的点组成的集合。图 6-43 所示为一条嵌入为零水平集的曲线。

与参数形变模型不同，水平集方法没有跟踪不同时刻曲线的运动情况，而是在固定坐标系中更新不同时刻下的水平集函数来模拟曲线的演化。图 6-43 中，一条圆形曲线被嵌入为一个水平集函数的零水平集，当圆扩大时，水平集函数相应地发生了变化，但新曲线仍对应为新水平集函数的零水平集。在嵌入的曲线改变其拓扑时，水平集函数依然保持是一个有效的函数，见 图 6-44

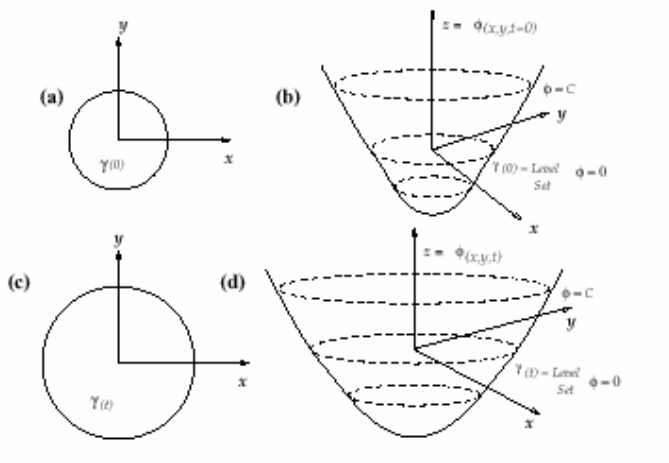



图 6-44 曲线的运动对应着水平集函数Φ的变化

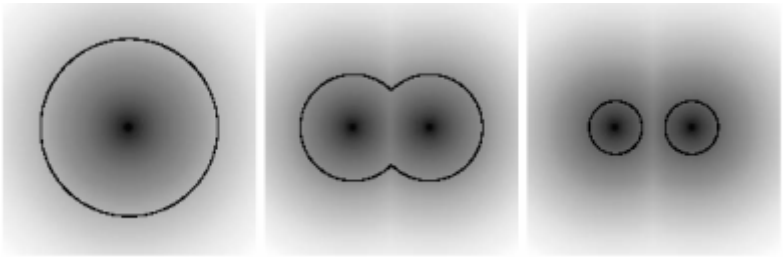


图 6-45 水平集分裂为两条曲线时水平集函数仍保持有效

给定一个水平集函数$\Phi(x,y,t)$，其零水平集对应为轮廓曲线$X(s,t)$，则有：

$$\Phi(X(s,t),t)=0 \tag{6-16}$$

使用链规则令此方程对 $t$ 求导，可得到：

$$\frac{\partial \phi}{\partial t}+\nabla \phi \cdot \frac{\partial X}{\partial t}=0 \tag{6-17}$$

其中$\nabla \phi$代表$\phi$的梯度。不妨假设在零水平集内$\phi$是负的，在外是正的。

因此，水平集曲线的向内的单位法向量可以表示为：

$$N=-\frac{\nabla \phi}{|\nabla \phi|} \tag{6-18}$$

因此有：

$$\frac{\partial \phi}{\partial t}=V(k)|\nabla \phi| \tag{6-19}$$

其中零水平集对应曲线的曲率 $k$ 为：

$$k=\nabla \cdot \frac{\nabla \phi}{|\nabla \phi|}=\frac{\phi_{xx}\phi_y^2-2\phi_x\phi_y\phi_{xy}+\phi_{yy}\phi_x^2}{(\phi_x^2+\phi_y^2)^{3/2}} \tag{6-20}$$

为实现几何形变模型需要解决三个问题：

(1) 初始化函数$\phi(x,y,t=0)$的构造，必须使其零水平集对应于初始轮廓的位置。通常做法是设置$\phi(x,y,0)=D(x,y)$，其中 $D(x,y)$是从每个网格点到零水平集的符号距离。

(2) 速度函数的设计：Caselles 等[Caselles 1993] 和 Malladi 等 [Malladi 1995]提出的几何形变轮廓采用如下公式：

$$\frac{\partial \phi}{\partial t}=c(k+V_0)|\nabla \phi|, \qquad c=\frac{1}{1+|\nabla(G_\sigma * I)|} \tag{6-21}$$

正 $V_0$ 使曲线收缩，负 $V_0$ 使曲线扩展。曲线演化通过一个乘数停止项 c 与图像数据关联。

此方法对于分割对比度高的对象效果不错。但是，当对象的边缘不清晰或者有狭窄缺口(gap)时，几何形变模型可能出错，而且一旦曲线跨过边缘，它不会被拉回到正确的边缘位置。

为解决后一个问题，Caselles 等[23] [24]和 Kichenassamy 等[25] [26]使用一个能量最小化公式设计速度函数，通过在速度函数中加入额外的停止项，能使模型即使跨过边缘也能被拉回来，而且也不会跨过边缘上的小缺口。

(3) 常数形变通常被用于解决大尺度的形变和发现窄边缘锯齿（indentation）和突起（protrusion）。但是，常数形变使曲线可能从初始光滑的零水平集形变成锋利的角点（corner）。一旦出现角点，由于其法向量方向有二义性，如何继续形变就不明确了。解决二义性的方法有 Sethian 提出的熵条件（*entropy condition*）[Sethian 1982]和由 Osher and Sethian 提出的 *entropy satisfying* 数值方法[Osher 1988]。

Level Set方法自提出以来，已在图像处理和计算机视觉等领域得到广泛的应用：如Sethian和Osher [16]等用Level Set去除图像噪声；Malladi [19]将其应用于图像分割，特别是医学图像的分割和重建中；Bertalmio [27]等将Level Set应用于图像变形和破损图像修复中；Masouri [28]将Level Set运用于运动目标跟踪领域；Parogios和Deriche [29]用Level Set方法进行纹理分割以及运动目标分割和跟踪；Samson [30]等人用Level Set方法实现图像分类等。

### 6.7.2 level set 算法开发包的设计与实现

(1) Level Set 算法流程

Level Set 方法的基本思想是将平面闭合曲线隐含的表达为二维曲面函数的水平集，即具有相同函数值的点集，通过 Level Set 函数曲面的进化隐含的求解曲线的运动。尽管这种转化使得问题在形式上变得复杂，但在问题的求解上带来很多优点，其最大的优点在于曲线的拓扑变化能够得到很自然的处理，而且可以获得唯一的满足熵条件的弱解；另外，界面的几何属性(法向量、曲率等)可以很容易的得到；最后，这种思想可以不做任何改变推广到高维空间。

Level Set 函数的演化满足如下的基本方程：

$$\Phi_t + F\left|\nabla\Phi\right| = 0 \tag{6-22}$$

其中 $\Phi$ 为 Level Set 函数，其零水平集表示目标轮廓曲线，即：

$$\Gamma(t) = \left\{x \mid \Phi(x,t) = 0\right\} \tag{6-23}$$

$\left|\nabla\Phi\right|$ 表示 Level Set 函数的梯度范数；$F$ 为曲面法线方向上的速度函数，控制曲线的运动，一般 $F$ 包括三项：与图像有关的项（如梯度信息等），与曲线的几何形状有关的项（如曲线的曲率等）以及附加的演化项(Additional propagation terms)。

$$\frac{\partial \Phi}{\partial t} \approx \frac{\Phi_{i,j}^{t+1} - \Phi_{i,j}^{t}}{\Delta t} \tag{6-24}$$

$$\left(\left|\nabla \Phi\right|\right)^2 \approx \left(\frac{\partial \Phi}{\partial x}\right)^2 + \left(\frac{\partial \Phi}{\partial y}\right)^2 \approx \left(\frac{\Phi_{i+1,j}^{t} - \Phi_{i-1,j}^{t}}{2\Delta x}\right)^2 + \left(\frac{\Phi_{i,j+1}^{t} - \Phi_{i,j-1}^{t}}{2\Delta y}\right)^2 \tag{6-25}$$

我们可以得到下面的离散方程：

$$\Phi_{i,j}^{t+1} = \Phi_{i,j}^{t} + F(K)\left|\left(\frac{\Phi_{i+1,j}^{t} - \Phi_{i-1,j}^{t}}{2\Delta x}\right)^2 + \left(\frac{\Phi_{i,j+1}^{t} - \Phi_{i,j-1}^{t}}{2\Delta y}\right)^2\right|\Delta t \tag{6-26}$$

水平集方法虽然在处理拓扑变化方面具有参数形变模型所无可比拟的优势，但是由于它是将问题转化到高一维空间来处理，必然会带来计算量方面的大大增加。窄带(Narrow Band)算法是水平集方法实现中常用的算法，其主要思想是只更新零水平集附近点的水平集函数值，而无须计算整个图像空间中每一点的水平集函数值，以此来提高水平集方法的计算效率。如 图 6-46 所示，左边的图表示设置的窄带，在这个窄带以外的像素点，就不需要再进行符号距离的计算，从而大大的减少了计算量。随着时间演化，当我们跟踪的前端演化到 图 6-46 右边的情况时，也就是前端的部分像素点已经或者非常接近窄带的边缘时，要停止演化，要进行重新初始化操作和重新设置窄带宽度。对于窄带算法而言，如何选取窄带宽度成为提高窄带算法效率的关键，过宽必然加大计算量，过窄必然影响水平集函数值的差分迭代。

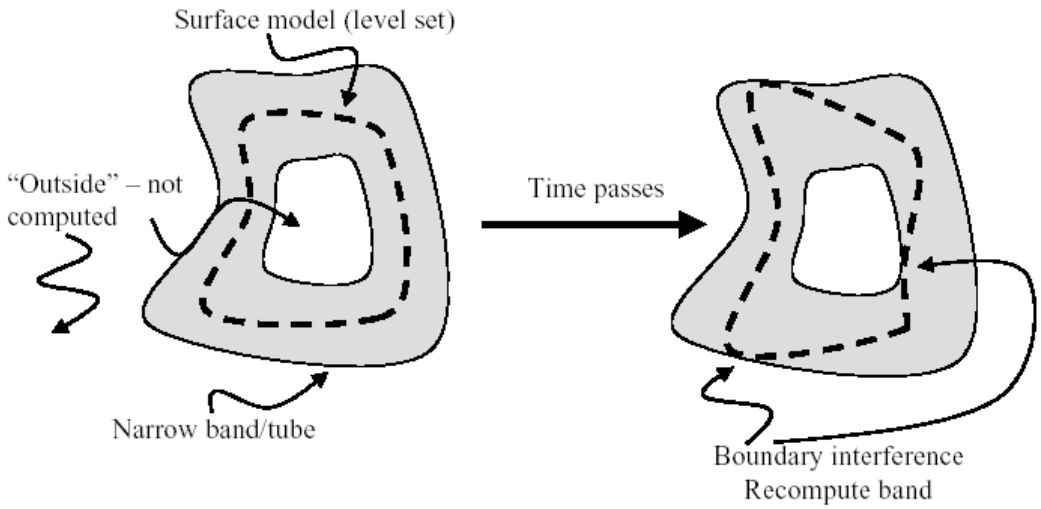



图 6-46 窄带算法示意图

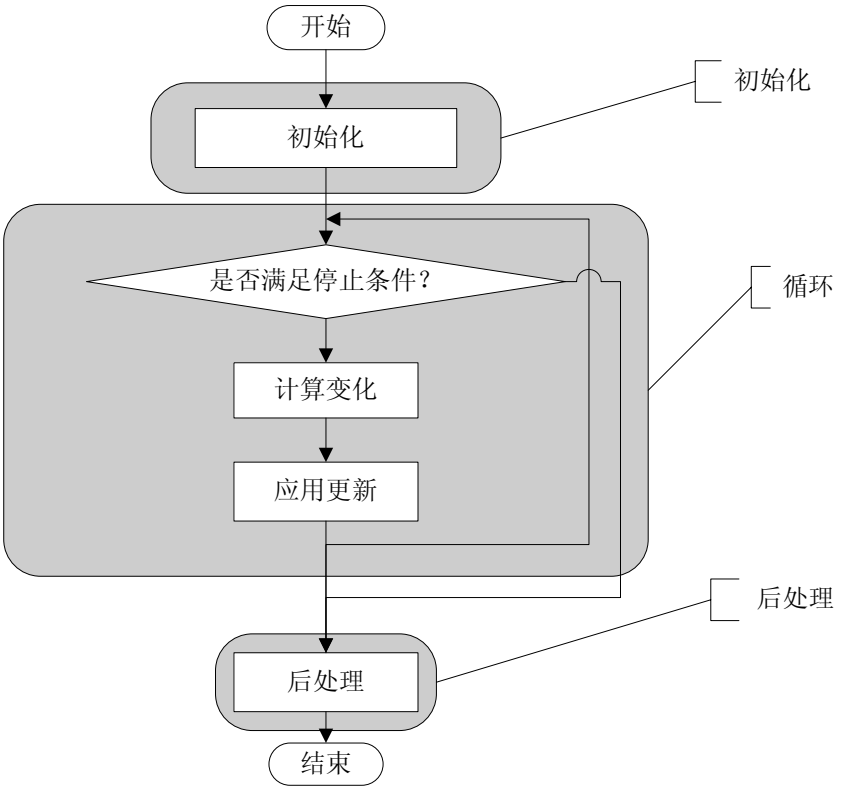



图 6-47 Level Set 算法主框架

图 6-47 Level Set算法主框架给出了level set算法的主框架流程。该算法的输

入数据为距离图，距离图是这样定义的，给定一条轮廓线，图像上每个像素的值表示该点到轮廓线上最近一点的距离，如果该点在轮廓线外部，其值为正，如果在内部，其值为负。输出数据仍为一幅距离图，只是零水平线已经被更新过。该算法主要分为三个模块：

初始化部分：一般在这个部分主要对输入图像进行处理，找出零水平线和窄带，初始化的方法有很多种，可以通过不同的子类来定义。

循环部分：这部分一般包含三块

循环终止条件：一般用循环次数来定义。当循环次数小于指定循环次数时继续循环，大于时终止循环。这个条件由子类具体定义。

计算窄带内的点的更新值：应用数学公式，更新窄带内每一点的值

更新窄带：根据上一步计算出来的更新值，重新计算零水平线和窄带，具体方法由子类定义。

后处理部分：这一部分可有可无，如果需要的话，可以在子类中定义。

(2) **类协作图**

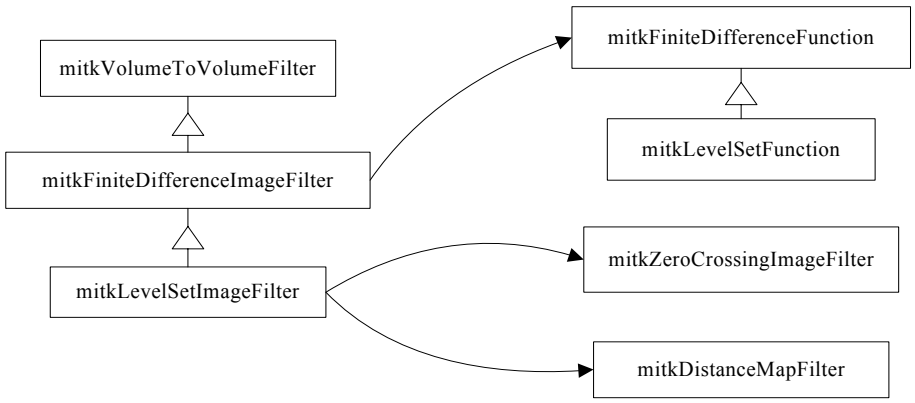



图 6-48 level set 算法包类协作图

如图 6-48 给出了level set算法包中各类的协作图。mitkFiniteDifferenceImageFilter依据图 6-48 定义了level set算法的主框架流程。事实上，它可作为所有有限元差分算法的基类，为他们提供一个统一的框架。类mitkFiniteDifferenceFunction是一个纯虚类，他为实现level set算法的各种数学方法提供了一个基类。mitkFiniteDifferenceImageFilter中保存了一个指向

mitkFiniteDifferenceFunction的指针。类mitkLevelSetImageFilter定义了一个level set算法的具体实现，他拥有指向mitkZeroCrossingImageFilter和mitkDistanceMapFilter的指针，用以实现零水平集的查找和距离变换。mitkLevelSetFunction定义了一种依据公式4.17给出了level set算法中所用到的数学方法的一种实现。

(3) 类内部组成结构详解

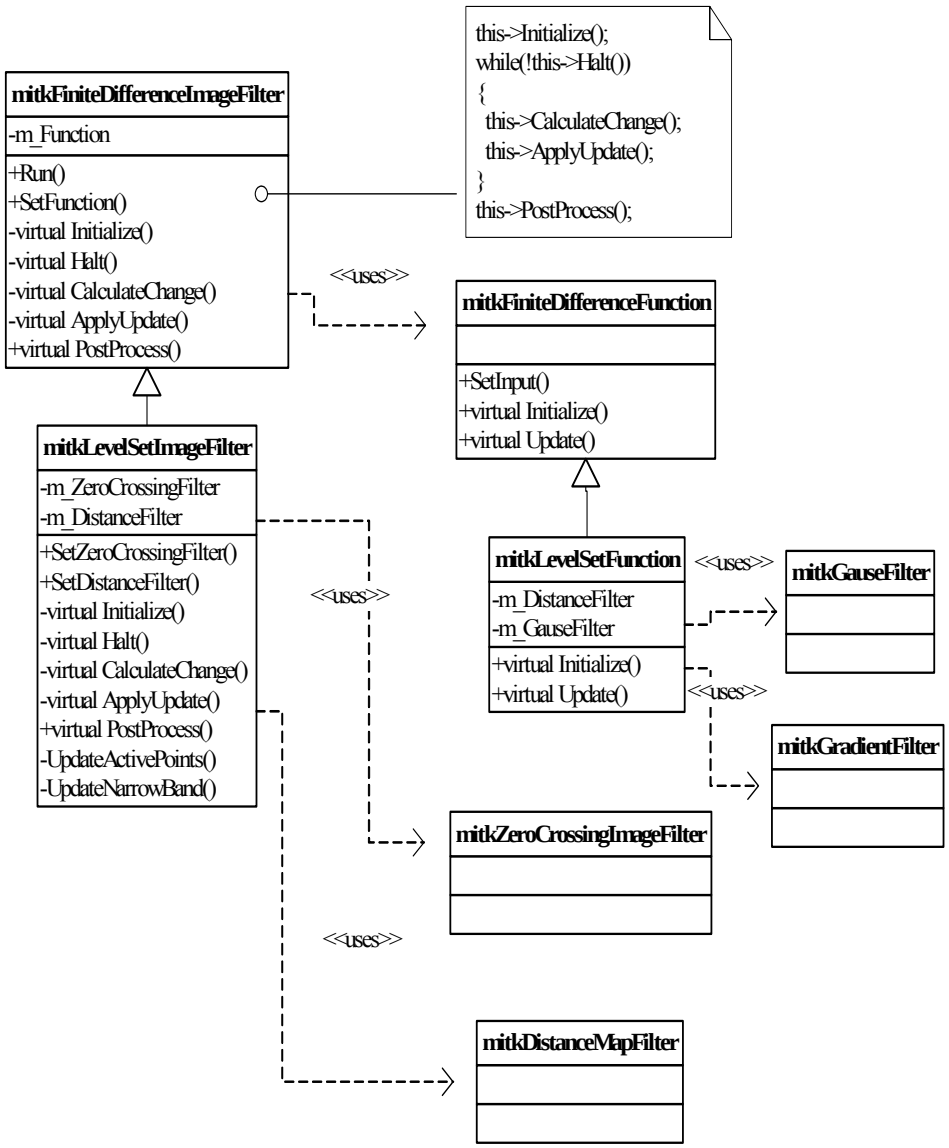



图 6-49 level set 算法包内部组成结构

如 图 6-49 给出了level set算法包的内部组成结构：

mitkFiniteDifferenceImageFilter 给出了 level set 算法的主框架：

*函数Run()*：为外部可见的，用来启动level set算法。在Run函数中依次调用虚函数Initialize()、Halt()、CalculateChange()、ApplyUpdate()和PostProcess()，这些函数对应于 图 6-47 中的各个步骤，在mitkFinitDifferenceImageFilter中没有定义他们，子类通过重载各个虚函数，可以完成不同的实现方法。

*函数 SetFunction()*：指定所采用的 function 的类型，他们必须是 mitkFiniteDifferenceFunction 的子类；

类 mitkFiniteDifferenceFunction 为 level set 算法所采用的数学方法提供了一个基类，他根据输入数据，计算每个像素的更新值。虚函数 Initialize()和 Update()由子类具体定义。

*函数 SetInput()*：mitkFinitDifferenceFunction 的输入数据是图像的原始数据，由函数 SetInput()指定；

mitkZeroCrossingImageFilter 定义了一种求零水平线的方法。他的输入是距离图，其中包含了一条零水平线，但这条水平线是隐含的，mitkZeroCrossingImageFilter 的作用就是找到这条零水平线，将水平线上所有的点赋零值，并写到输出的距离图像中。

mitkDistanceMapFilter 的作用是将的二值图转化为距离图，他的输入是一幅二值图，二值图的边界为给定的轮廓线，轮廓线内部的值为零，轮廓线外部的值为非零。输出是一幅距离图，轮廓线上的距离值为零，轮廓线内部的距离值为负，轮廓线外部的距离值为正。

mitkLevelSetImageFilter 定义了一种 level set 算法的具体实现。

*m_ZeroCrossingFilter*：是一个 mitkZeroCrossingImageFilter 类型的指针，用来查找令水平线；

*m_DistanceFilter*：是一个 mitkDistanceMapFilter 类型的指针，用来进行距离变换；

*函数 SetZeroCrossingFilter()*：用来指定 m_ZeroCrossingFilter 的类型；

*函数 SetDistanceFilter()*：用来指定 m_DistanceFilter 的类型

*函数 Initialize()*：做一些初始化工作：调用 m_ZeroCrossingFilter 在输入的距离图中找到隐含的水平线；调用 UpDateActivePoints()函数处理活动点；调用 UpdateNarrowBand()函数处理窄带中的点；

*函数 Halt()*为循环终止条件，在本节的定义为，当循环次数小于用户指定的循环次数时继续循环，大于时终止；

*函数 CalculateChange()*调用 m_Function 来计算窄带内每个点的更新值，对于窄带内的点，可以在该函数内更新多次，更新的次数由用户指定；

函数 ApplyUpdate()利用 CalculateChange()中更新后的窄带，定义一幅二值图，将值为负的点定义为零点，将非负的点定义为非零点，然后调用 m_ZeroCrossingFilter 找出新的水平线，调用 UpdateActivePoints()更新活动点，调用 UpdateNarrowBand()更新窄带点；

*函数 PostProcess()*主要做一些后处理工作，将更新后的距离图按照给定的格式写入输出数据；

mitkLevelSetFunction 根据式 6-25 给出了 level set 算法所采用的一种数学方法。

*m_GauseFilter*：一个 mitkGauseFilter 类型的指针，用来对图像滤波；

m_GradientFilter：一个 GradientFilter 类型的指针，用来计算图像的梯度值；

*函数 Initialize()*：首先调用 m_GauseFilter 对输入的图像进行高斯滤波，再调用 m_GradientFilter 求出滤波后的图像的梯度图像；

*函数 Update()*应用式 6-20 求出每个像素点更新后的值；

同上一节介绍的 fast marching 算法一样，level set 算法开发包同样为那些对该算法不太了解的人提供了一个封装，使得输入数据为原始图像数据，输出数据为分割好的图像。

如图 6-50，mitkLevelSetPacket是封装好的level set算法包，它包含了指向 mitkDistanceFilter、mitkLevelSetImageFilter 和 mitkDistanceConvertFilter 的指针 m_DistanceFilter、m_LevelSetFilter 和 m_DistanceConverFilter。m_DistanceFilter

将输入数据根据给定的轮廓线转化为距离图像，作为m_LevelSetImageFilter的输入，m_DistanceConvertFilter将m_LevelSetImageFilter输出的距离图像转化为分割后的图像。

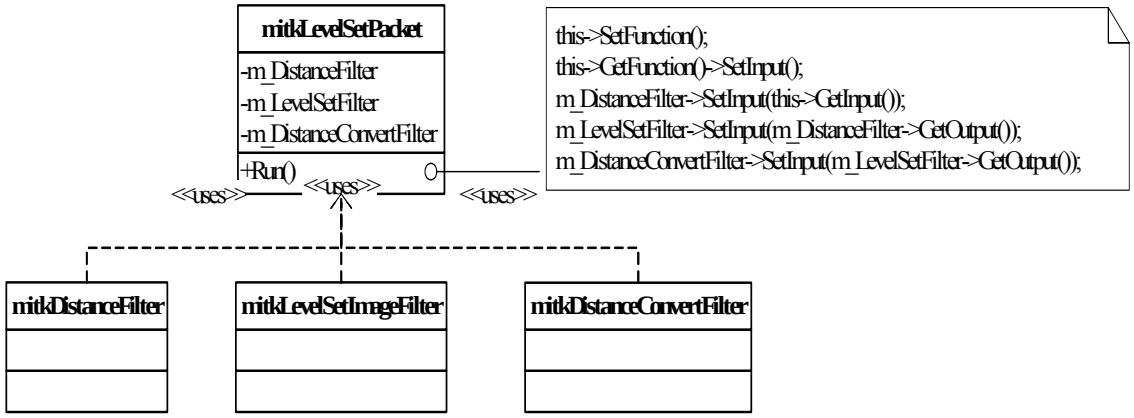



图 6-50　图 4.19 封装后的 level set 开发包

### 6.7.3 level set 分割结果

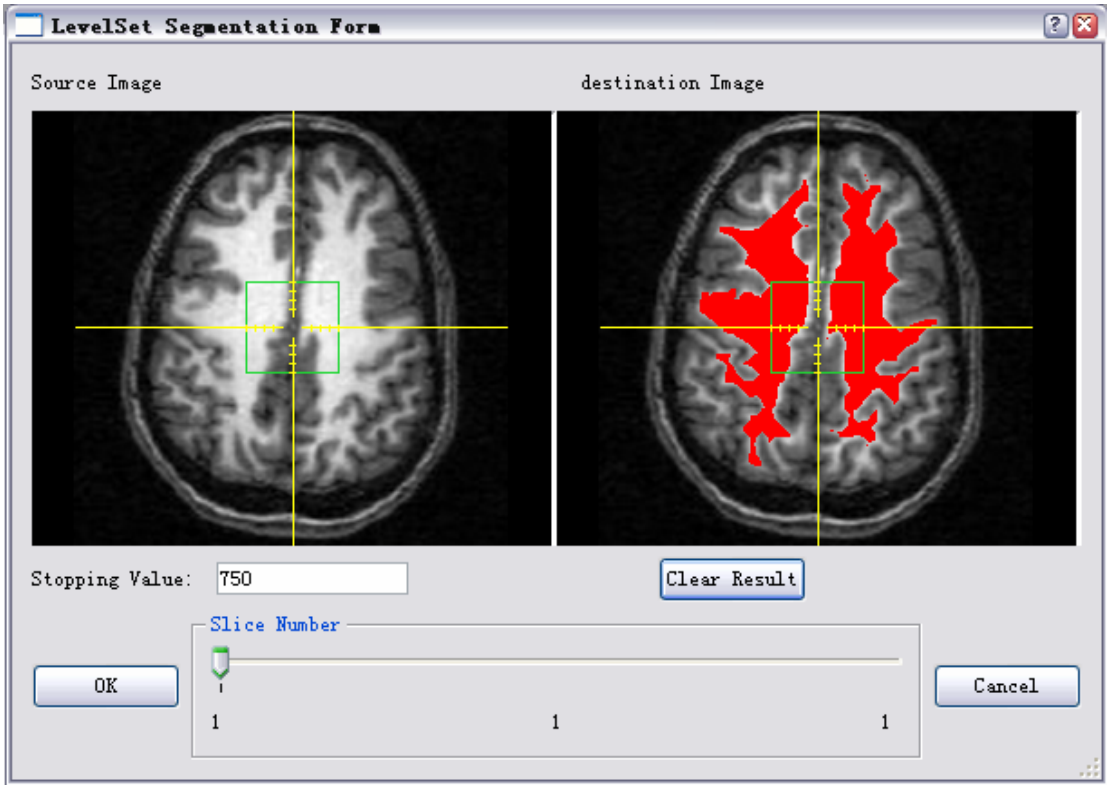



图 6-51 level set 插件分割结果

图 6-51 以系统界面的形式给出了level set插件的分割结果，其中fast marching算法的阈值设为 750, level set内循环的次数为 3，外循环的次数为 3，窄带的宽度为 5。

## 6.8 小结

医学图像分割是医学影像处理与分析中的一个重点课题和难点，分割的结果是三维可视化和定量分析等后续处理的基础。本章针对医学图像中的一些主流分割算法，对其原理进行了介绍，并给出了类的结构框图说明其是如何在MITK 的框架中实现的。本章介绍的几个算法各有特点，就其实用性来说，阈值分割与区域增长最简单易用，交互式分割最准确，但是耗时长。就发展方向来说，Level Set 得到了越来越多的重视，通过它可以直接将三维模型提取出来。更进一步，人机交互的分割策略将会成为分割算法的主流。关键是如何将人的知识转化为模型，让计算机能够理解，从而引导分割策略。我们应当认识到，尽管对医学图像分割方法的研究已有三十余年的历史，但是到目前为止尚不存在一个通用的解决方法，现有的任何一种单独的图像分割算法都难以对一般图像取得令人满意的分割结果，因而人们在继续致力于将新的概念，新的方法引入图像分割领域的同时，更加重视多种分割算法的有效结合，近几年来提出的方法大多数是结合了多种算法的。采取什么样的结合方式才能体现各种方法的优点，取得好的效果成为人们关注的问题。

参考文献

1. P.K.Sahoo, S.Soltani, A.K.C.Wang, and Y.C.Chen. A survey of thresholding techniques. *Computer Vision, Graphics, and Image Processing*, 1988, 41:233-260.
2. Y. J. Zhang, J. J. Gerbrands. Transition region determination based thresholding. *Pattern Recognition Letter*, 1991, 12:13-23.
3. Y. Nakagawa, A. Rosenfeld. Some experiments on variable thresholding. *Pattern Recognition*, 1979, 11:191~204.
4. H.D. Li, M. Kallergi, L.P. Clarke, V.K. Jain, R.A. Clark. Markov random field for tumor detection in digital mammagraphy. *IEEE Trans On Medical Imaging*, 1995, 14:565–576.
5. C. Lee, S. Hun, T.A. Ketter, and M. Unser. Unsupervised connectivity-based thresholding

segmentation of midsaggital brain MR images. *Comput. Biol. Med.*, 1998, 28:309–338.

6. S. Pohlman, K.A. Powell, N.A. Obuchowski, W.A. Chilcote, and S. Grundfest-Broniatowski. Quantitative classification of breast tumors in digitzed mamograms. *Medical Physics*, 1996, 23:1337–1345.
7. J.F. Mangin, V. Frouin, I. Bloch, J. Regis, J. Lopez-Krahe. From 3D magnetic resonance images to structural representations of the cortex topography using topology preserving deformations. *J. Math. Imag. Vis.* , 1995, 5:297–318.
8. J. K. Udupa, and S. Samarasekera. Fuzzy connectedness and object definition: theory, algorithms, and applications in image segmentation. *Graphical Model and Image Processing*, 1995, 58(3): 246-261.
9. I.N. Manousakas, P.E. Undrill, G.G. Cameron, T.W. Redpath. Split-and-merge segmentation of magnetic resonance medical images: performance evaluation and extension to three dimensions. *Computers and Biomedical Research*, 1998, 31:393–412.
10. V. Caselles, F. Catte, T. Coll, and F. Dibos. A geometric model for active contours. *Numerische Mathematik*, 1993, 66:1–31.
11. R. Malladi, J.Sethian, and B.Vemuri. Shape modeling with front propagation: A level set approach. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 1995, 17(2):158-175.
12. L. Alvarez, F. Guichard, P. L. Lions, and J. M. Morel. Axioms and fundamental equations of image processing. Archive for Rational Mechanics and Analysis, 1993, 123(3):199–257.
13. G. Sapiro and A. Tannenbaum. Affine invariant scale-space. *Proc. Intl. Conf. on Computer Vision*, 1993, 11(1):25–44.
14. R. Kimmel, A. Amir, and A. M. Bruckstein. Finding shortest paths on surfaces using level sets propagation. *IEEE Transaction On Pattern Analysis and Machine Intelligence*, 1995, 17(6):635–640.
15. B. B. Kimia, A. R. Tannenbaum, and S. W. Zucker. Shapes, shocks, and deformations I: the components of two-dimensional shape and the reaction-diffusion space. Int'l J. Comp. Vis., 1995, 15:189–224.
16. S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 1988, 79:12–49.
17. J. A. Sethian, Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Material Science. Cambridge, UK: Cambridge University Press, 2nd ed., 1999.
18. J.A.Sethian. A fast marching level set method for monotonically advancing fronts. Proc. Natl. Acad. Sci. USA, 93(1996): pp.1591-1595.

19. R.Malladi and J.A.Sethian. An O(N log(N)) Algorithm for Shape Modeling. In Proceedings of National Academy of Sciences, USA, Sept. 1996,Vol. 93: pp. 9389-9392.
20. J. A. Sethian. Level Set Methods. Cambridge University Press, 1996.
21. J. A. Sethian. Curvature and evolution of fronts. Communication: Mathematics & Physics, 1985, 101:487–499.
22. J. A. Sethian. A review of recent numerical algorithms for hypersurfaces moving with curvature dependent speed. Journal of Differential Geometry, 1989, 31:131–161.
23. V. Caselles, R. Kimmel, and G. Sapiro. Geodesic active contours. in *Proc. 5th Int'l Conf. Computer Vision*, 1995, 694–699.
24. V. Caselles, R.Kimmel, and G.Sapiro. Geodesic active contours. International Journal of Computer Vision, 1997, 22(1): 61-79.
25. A. Kichenassamy, A.Kumar, P.Olver, A. Tannenbaum and A. Yezzi. Gradient flows and geometric active contour models. *Proceedings of IEEE International Conference on Computer Vision*, 1995, 810-815.
26. A. Yezzi, S. Kichenassamy, A. Kumar, P. Olver, and A. Tennenbaum. A geometric snake model for segmentation of medical imagery. IEEE Trans. On Medical Imaging, 1997, 16:199–209.
27. Bertalmio M, Sapiro G, Randall G. Region tracking on level-set methods. IEEE Transactions on Medical Imaging, 1999, 18(5): 448-451.
28. Masouri A-R, Sirivong B, Konrad J. Multiple motion segmentation with level sets. Proceedings of SPIE, 2000, Vol.3974, pp.584-595.
29. Paragios N, Deriche R. Geodesic active contours and level sets for the detection and tracking of moving objects. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2000, 22(3): 266-280.
30. Samon C, Blanc-Feraud L, Aubert G, Josiane Z. Level set model for image classification. International Journal of computer Vision, 2000, 40(3): 187-197.

# 7 配准算法的设计与实现

## 7.1 配准算法简介

图像配准(Image Registration)技术现已广泛应用应用于模式识别、计算机视觉、医学影像处理、遥感数据处理等诸多领域[1]。本章中我们主要考虑其在医学领域的应用。

20 世纪以来医学成像技术经历了一个从静态到动态、从形态到功能、从平面到立体的发展过程，尤其在计算机技术高度发达之后，医学成像技术的发展给临床医学提供了从 X 线，超声，计算机断层成像(CT)，数字减影血管造影(DSA)，单光子发射断层成像(SPECT)，磁共振成像(MRI)，数字荧光造影(DF)，正电子发射断层成像(PET)等形态和功能的影像信息。根据医学图像所提供的信息内涵，可将这些信息分为两大类：解剖结构图像（CT、MRI、B 超等）和功能图像（SPECT、PET 等）。这两类图像各有其优缺点：功能图像分辨率较差，但它提供的脏器功能代谢信息是解剖图像所不能替代的；解剖图像以较高的分辨率提供了脏器的解剖形态信息（功能图像无法提供脏器或病灶的解剖细节），但无法反映脏器的功能情况。目前这两类成像设备的研究都已取得了很大的进步，图像的空间分辨率和图像质量有很大的提高，但由成像原理不同所造成的图像信息局限性，使得单独使用某一类图像的效果并不理想，而多种图像的利用又必须借助医生的空间构想和推测去综合判定他们所要的信息，其准确性受到主观影响，更主要的是一些信息将可能被忽视。解决这个问题的最有效方法，就是以医学图像配准技术为基础，利用信息融合技术，将这两种图像结合起来，利用各自的信息优势，在一幅图像上同时表达来自人体的多方面信息。使人体内部的结构、功能等多方面的状况通过影像反映出来，从而更加直观地提供了人体解剖、生理及病理等信息。其中图像配准技术是图像融合的关键和难点[2]。

20 世纪 80 年代以来，医学图像配准(Medical Image Registration)技术研究取得了显著的进展。从基于外部特征(Extrinsic)到基于内部特征(Intrinsic)，从二维领域到三维领域，从刚性配准到非刚性配准，从单模态(Monomodal)图像到多模态 (Multimodal) 图像，研究领域不断扩大，配准算法日益丰富，特别是计算机硬件的飞速发展更进一步推动了图像配准技术的研究，产生了许多成功的临床

应用。目前，医学图像配准已发展成为图像处理领域里一个比较活跃的分支，每年相关的期刊、会议上发表了大量关于医学图像配准的论文。

图像配准(Image Registration)实际上是指在两幅图像相应点之间建立一一映射的过程，也就是说，将两幅图像中对应于空间同一位置的点联系起来，这里的映射一般称为变换(Transform)。

图 7-1 简单说明了一个二维图像配准的概念。(a)和 (b)是对应于同一物体同一位置的两幅图像。这两幅图有两点明显的不同，第一是方向上有差异（图 (a)已经被旋转了一个角度以便更好的观察与(b)在方向上的差别），第二是形状上(b)少了一部分。我们可以认为它们由不同的成像设备得到（(b)中少的那部分代表两种成像设备成像模式的差异），也可以认为它们来自同一个成像设备（如手术前和手术后， (b)中少的那部分代表病变组织）。(c)和(d)给出了两个图像空间中点对点的映射过程。(a)中的每一个点 $(x_1, y_1)$ 都被映射到(b)中唯一的一个点 $(x_2, y_2)$，(c)表示前向过程，(d)表示逆向过程。如果这种映射是一对一的，即一个图像空间中的每一个点在另外一个图像空间中都有对应点（如(a)和(b)中的两个黑点），或者至少在医疗诊断上感兴趣的那些点能够准确或近似准确的对应起来，我们就称为配准。

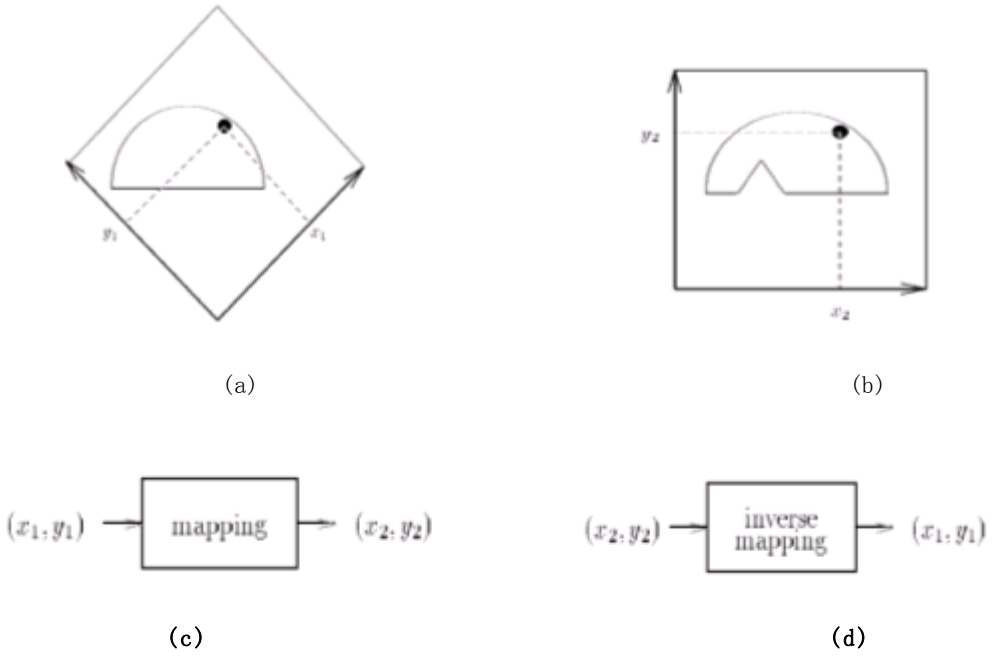



图 7-1 图像配准概念的图示表示

下面我们首先介绍 MITK 中采用的配准算法框架，然后具体讨论配准的分模块实现，最后是应用实例说明。

## 7.2 MITK 中的配准算法框架

很多实际应用中一个刚性变换就可以描述图像间的空间对应关系，例如同一主体（Intrasubject）的脑部图像配准。目前，刚性配准技术已经发展得比较成熟，并且进入临床应用。因此 MITK 目前版本中主要是考虑这些技术的实现。当然也有很多情况下需要用到各种比较复杂的非刚性变换。相比较而言，非刚性配准还处于很活跃的研究阶段[3]。实际配准过程中，可以根据不同的特点和要求采用简单的刚性变换(Regid Transform)、仿射变换(Affine Transform)，或较复杂的弹性形变(Elastic Deform)等。

由于医学图像配准的算法比较复杂，而且方法种类很多，新的方法也层出不穷，为了给使用者提供一个开放的、易于扩充的架构，便于以后添加新的配准算法，我们在MITK中根据配准算法的一般步骤，分几何变换（Transform）、图像插值（Image Interpolator）、相似性测度（Similarity Metric）、优化（Optimizer）四个相对独立的模块来实现整个配准算法，各模块间的相互关系如 图 7-2 所示。

对于两幅图像 $F$ 和 $M$，分别用函数 $f(X)$ 和 $m(Y)$ 表示，其中 $X$、$Y$ 为各自图像的定义域（解剖结构空间）；这里图像配准定义为寻找一种几何变换 $T_t$（$t$ 为该变换的控制参数），使 $S(f(X),m(T_t(X))$ 取得最大值：

$$T_t^* = \arg\max_{T_t} S(f(X),m(T_t(X))) \tag{7-1}$$

其中 $S$ 为对任意两幅图像定义的一目标函数，用来衡量两图像的匹配效果，一般用相似性测度表示。

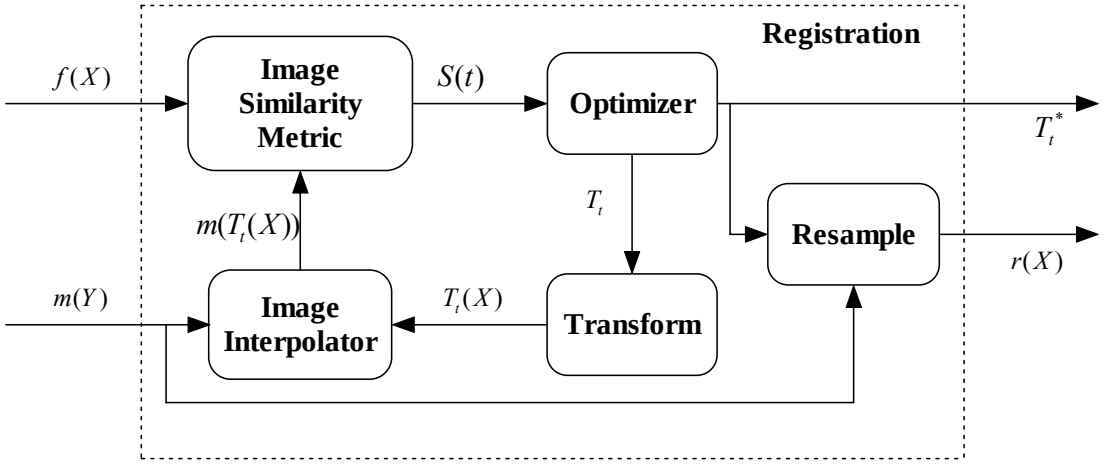



图 7-2 配准算法框架图

从算法框图中可以看出整个配准算法的流程：

1. 输入待配准的两幅图像，分别记为参考图 $f(X)$（Fixed Volume）和浮动图 $m(Y)$（Moving Volume）；
2. 对参考图指定区域 $X$ 进行几何坐标变换(Transform)得到新的区域 $T_t(X)$ 坐标，其中 $t$ 表示变换参数；
3. 通过一定的插值方法(Image Interpolator)得到浮动图在区域 $T_t(X)$ 的取值 $m(T_t(X))$；
4. 在相似性测度模块计算参考图 $f(X)$ 和插值图 $m(T_t(X))$ 的相似度，它是一个关于几何变换参数的函数 $S(t)$；
5. 相似度函数 $S(t)$ 输入优化模块中进行最优化计算得到最终变换参数，这个过程在计算中一般通过迭代来实现，即重复步骤 2~4 直到取得最大相似度时终止迭代循环。
6. 整个配准算法模块输出配准时所采用几何变换的最优变换参数以及浮动图在最优变换下的插值图像。重采样模块 Resample 由 Transform 和 Image Interpolator 组成。

可以看出在此过程中，配准实际上作为一种优化问题来考虑：寻找一种变换 $T_t$，使相似度函数 $S(t)$ 取得最大值。下面分别对以上各算法模块作简要介绍。

几何变换（Transform）将参考图空间 $X$（Fixed Volume Space）中像素点映射到浮动图空间 $Y$（Moving Volume Space）中去。这和配准中图像变换的直观理解有点不一样，刚好是它的逆过程。因为正变换中从浮动图空间变换到参考图空间后可能会产生“空洞”，不利于后续的插值处理，逆变换则可以避免这一点。几何变换的类型一般有刚性变换、仿射变换、投影变换、曲线变换等。各种变换都是通过一组参数 $t$ 来表示，例如刚性变换可用 3 个方向上的平移参数和旋转角度共 6 个参数来表示。几何变换模块在 MITK 中用 Transform 抽象类表示，具体的变换算法由 Transform 派生类实现。

图像插值（Image Interpolator）是为了估计浮动图中非网格点处的像素值。由于数字图像是对模拟图像的网格采样，只有格点处有像素值，而参考图空间中的格点映射到浮动图空间中去后可能不在格点上，为了得到这些浮动图中非格点处的像素值就有必要进行图像插值。一般常采用的插值方法有最近邻插值、线性插值、B 样条插值等。图像插值模块有两个输入一个输出：浮动图像 $m(Y)$（Moving Volume）和几何变换结果 $T_t(X)$（Transform 的输出）作为输入；插值结果 $I(X)=m(T_t(X))$ 作为输出。插值结果也是一个 Volume，因此 MITK 中将该抽象类（Interpolator）作为 VolumeToVolumeFilter 的派生类。

相似性测度（Similarity Metric Measure）作为一种准则用来评价参考图和插值后得到的图像匹配的效果，可以说这是整个配准框架中最关键的部分，它直接影响配准效果的好坏[4]。相似性测度是一个以几何变换参数为自变量的单值函数：$S(t)=S(f(X),m(T_t(X))$。该模块以参考图 $f(X)$ 和浮动图插值后得到的图象 $m(T_t(X)$ 作为输入，输出一个表示两图像相似（相关）性的标量值，可以看作一个以变换参数为自变量的函数，该函数输入到优化模块中作为代价函数（Cost Function）求取最优变换。一些非刚性配准算法中，除相似性测度外，还要考虑形变约束，两者之和作为优化的代价函数，即

$$C=-C_{similarity}+\lambda C_{deformation}。\tag{7-2}$$

函数优化（Cost Function Optimizer）如同图像处理与分析领域中的很多问题一样，图像配准也被归结成一个多参数优化问题。如前所述，给定一种配准准则，就可以定义一个以几何变换参数为自变量的多元目标函数，通过对该目标函数的最优化搜索得到配准时的几何变换参数。由于优化问题是一个比较经典的数学问题，有很多解决方法，这里关键的就是根据目标函数的特点选择合适

的优化算法和策略，准确、高效、快速地得到结果。

MITK中上述算法模块分别用Transform、InterpolateFilter、Metric、Optimizer四个抽象类表示。根据 图 7-2 算法框图，Transform得到几何变换参数后对参考图定义域进行几何变换，输出变换后的坐标；InterpolateFilter输入Transform变换后的坐标及浮动图进行插值运算，输出新的图像；Metric输入参考图和插值图，输出相似度函数；Optimizer对输入的相似度函数进行优化，输出最优变换参数。这些模块在配准类RegistrationFilter中通过相应的接口组合到一起，完成整个配准算法流程，最后输出参考图变换后的重采样图像。显然RegistrationFilter、InterpolateFilter输入输出均为三维体数据Volume，可以作为VolumeToVolumeFilter的派生类继承其数据成员和成员函数。Transform、Metric、Optimizer作为ProcesssObject的派生类。各算法模块类的继承结构及相互组合关系如 图 7-3、图 7-4 所示。

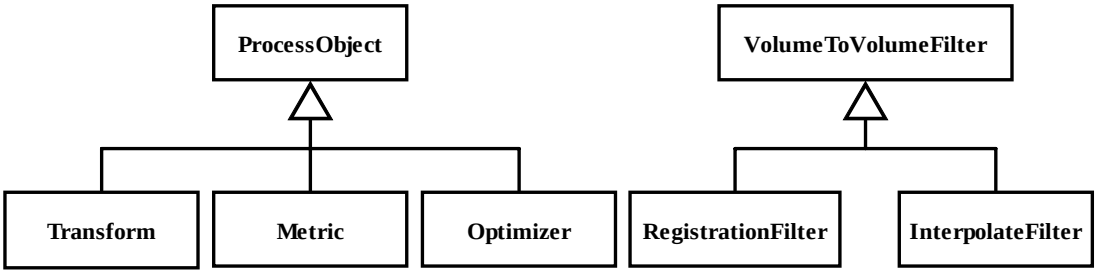



图 7-3 配准算法各模块类继承结构

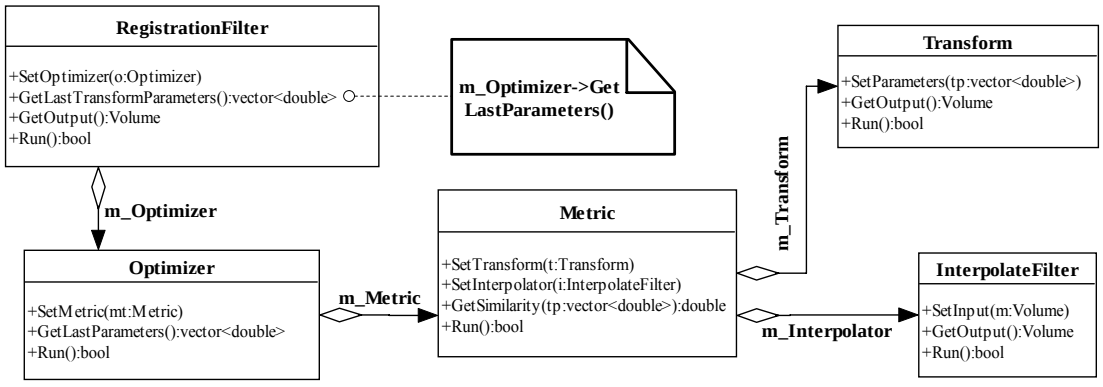



图 7-4 MITK 中配准算法各模块关系图

以上 Transform、InterpolateFilter、Metric、Optimizer 四模块是从高层概念上实现了相应的算法，提供基本的数据成员和成员函数接口。不同类型的各种具体算法通过它们的派生类在虚函数中实现。

用户通过设置具体的几何变换类型、插值方法、相似性度量准则、优化策略就能实现不同的配准算法。下面几节详细介绍 MITK 中采用的具体几何变换类型、插值方法、相似性度量准则和优化策略。

## 7.3 几何变换

空间映射 T 描述了一幅图像中的位置与另一幅图像中的相应位置之间的关系。这里的图像可以是二维的（2D），也可以是三维的（3D），所以这种映射可能是从二维空间到二维空间、从三维空间到三维空间、或者是在三维和二维之间的变换。然而在所有上述情况中，象源体——人体的部分或全部——都是三维的。所以，对大多数情况而言，二维空间内部的变换无法满足配准的要求。目前最广泛的配准应用中就包含三维图像对之间的配准。另外，一个更加重要的应用就是二维图像与三维图像之间的配准（2D-3D 配准）。在 2D-3D 的配准过程中，T 包含了从 3D 物体到 2D 平面的投影，以及 3D-3D 的变换。

### 7.3.1 刚性变换算法

如果用以配准的图像包含相同的内容，只是位置有所不同，那么就可以用旋转和平移来描述配准变换——这就是刚性变换。在三维情况下，刚性变换包含 6 个自由度，它们分别是：沿着三个坐标轴的平移 x、y、z，以及围绕三个坐标轴的旋转 α 、β 、γ 。通过这些未知量，我们可以构造一个刚性变换矩阵 $T_{rigid}$。它可以将一幅图像中的任意点映射到另一幅图像中，成为与之对应的变换点。这种变换可以通过旋转变换 R 和平移变换 $t=(t_x, t_y, t_z)^T$ 来表示：

$$T_{rigid}(x) = Rx + t \tag{7-3}$$

其中，旋转矩阵 R 的构造如下：

$$R=\begin{pmatrix} \cos\beta\cos\gamma & \cos\alpha\sin\gamma+\sin\alpha\sin\beta\cos\gamma & \sin\alpha\sin\gamma-\cos\alpha\sin\beta\cos\gamma \\ -\cos\beta\sin\gamma & \cos\alpha\cos\gamma-\sin\alpha\sin\beta\sin\gamma & \sin\alpha\cos\gamma+\cos\alpha\sin\beta\sin\gamma \\ \sin\beta & -\sin\alpha\cos\beta & \cos\alpha\cos\beta \end{pmatrix} \tag{7-4}$$

对于 2D－3D 刚性配准，我们需要同时考虑刚性变换和 3D 物体在平面上的投影。我们可以将刚性变换、平移变换和投影变换在各向同性坐标系中统一成

一个 4x4 的矩阵：

$$T_{rigid}=\begin{pmatrix}\cos\beta\cos\gamma & \cos\alpha\sin\gamma+\sin\alpha\sin\beta\cos\gamma & \sin\alpha\sin\gamma-\cos\alpha\sin\beta\cos\gamma & t_x\\ \cos\beta\sin\gamma & \cos\alpha\cos\gamma-\sin\alpha\sin\beta\sin\gamma & \sin\alpha\cos\gamma+\cos\alpha\sin\beta\sin\gamma & t_y\\ \sin\beta & -\sin\alpha\cos\beta & \cos\alpha\cos\beta & t_z\\ 0 & 0 & 0 & 1\end{pmatrix} \tag{7-5}$$

通常认为投影是（x、y、z）沿着 z 轴方向投影到 u、v 平面。这种投影变换可以通过图像系统的内在参数（$u_0$、$v_0$、$k_u$、$k_v$）来表示。对于 x 光投影而言，我们如下解释这些参数：$u_0$ 和 $v_0$ 定义了光线穿透点（在（u、v）平面上，该点的法矢指向 x 光光源），$k_v$ 和 $k_u$ 分别表示象素点在水平方向（u）和垂直方向（v）的大小。另外，它们也可以作为未知量，这样就会给配准算法增加四个自由度。投影变换矩阵 $T_{projection}$ 可以表示成 4x3 的矩阵，它将沿着 z 轴方向投影：

$$T_{projection}=\begin{pmatrix}k_u & 0 & u_0 & 0\\ 0 & k_v & v_0 & 0\\ 0 & 0 & 1 & 0\end{pmatrix} \tag{7-6}$$

在各向同性坐标系（x、y、z、1）$^T$ 中的 3D 坐标同这个矩阵相乘，得到向量（$\lambda_u$，$\lambda_v$，$\lambda$）$^T$。其中 $\lambda$ 代表缩放因子，投影到 2D 平面的结果可以由该向量中的前两项分别除以缩放因子 $\lambda$ 得到。

2D-3D 刚性配准所需要的变换 $T_{2D-3D}$ 可以由投影变换和刚体变换组合而成：

$$T_{2D-3D}=T_{projection}T_{rigid} \tag{7-7}$$

当我们考虑骨质或者与之接近的结构时，这种刚性变换可以正确的将反映相同内容的图像对应起来。这种假设对在处理脑部图像的时候非常适用，因为脑颅骨限制了大脑的形变。

然而对身体内大多数组织而言，刚性变换远远满足不了配准的需要，我们需要更多的自由度来足够精确地描述组织的形变。

### 7.3.2 线性变换与一对一变换

**线性变换**：许多作者认为仿射（affine）变换是线性的。从严格意义上讲，这是不正确的。线性变换是一种满足以下条件的特殊变换：

$$L(\alpha x_A+\beta x'_A)=\alpha L(x_A)+\beta L(x'_A)\quad \forall x_A, x'_A\in R \tag{7-8}$$

而仿射变换的平移部分不符合这一条件。确切的说，仿射变化是一种线性变换与平移变换的复合变换。

更进一步，反射（reflection）变换也是一种线性变换，但是它们在图像配准中很少用到。举例而言，如果在图像指导的神经外科中所使用的配准算法包含反射变换，那么就有可能导致在穿颅手术中定位到错误的一边。如果怀疑到算法中可能包括反射变换，那么就一定要在应用之前得出确证，以免造成事故。

**一对一变换**：在对同一目标的配准中，病人通过不同的设备进行成像。这种配准中所要求的变换 T 似乎是一对一的。这意味着图像 A 中的点经过变换后与图像 B 中唯一确定的点对应，反之亦然。在有些情况下，这种规则并不适用。首先，如果配准图像的维数并不相同，例如 x 光照片和 CT 成像之间的配准，一对一变换就是不可能的。其次，在一幅图像中的采样数据并没有反映在另一幅图像的采样数据中。

对于各种非仿射变换配准，一对一的变换都是不适用的。举例而言，在对不同目标的配准中，或者在对同一目标手术前和手术后的配准中，A 图像中就可能存在 B 图像中所不包含的结构，反之亦然。

### 7.3.3 变换算法在 MITK 中的实现

从 图 7-2 配准算法框架图中可以看出，变换模块Transform接受优化模块Optimizer的输出作为输入，即输入是一个变换参数集合。而Transform的输出是一个数据集，它记载着变换后点的坐标位置，其组织形式与volume相同。这个输出进一步作为插值模块Interpolator的输入参数之一。抽象类Transform的框架图如 图 7-5 示：

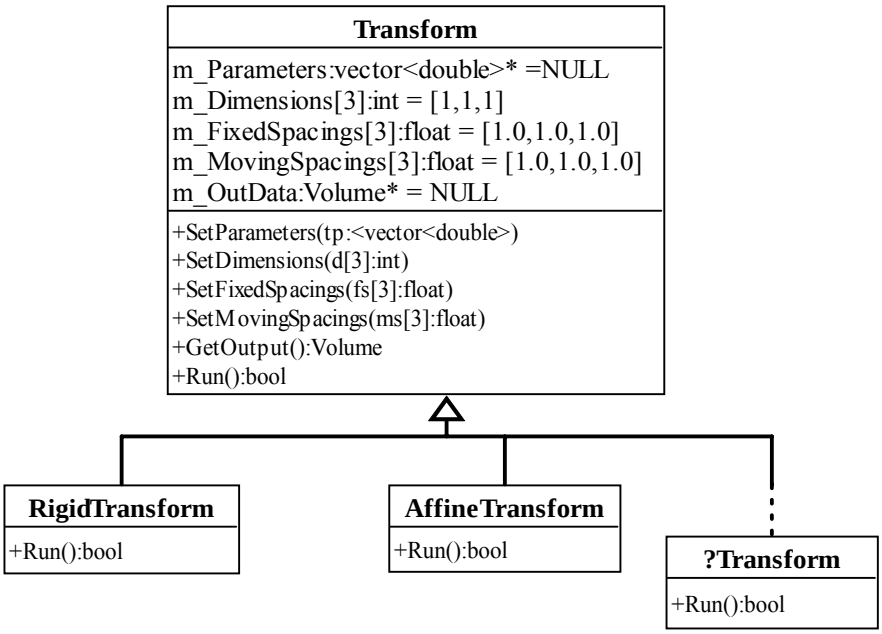



图 7-5 几何变换类图

对于各种变换算法，其用户接口都是一致的。通过函数 void SetTransParameter(float * p, int n)设置变换的参数。通过 `void SetVolumeSize(int w, int h, int n=1)` 设置变换图像数据的大小。具体的变换算法实现将在子类中，由 `virtual bool Excute()` 的重载函数实现。在变换参数和图像数据大小设置完成之后，就可以通过 `Run()` 函数调用 `Exctute()`，从而完成变换工作。最终，变换的结果可以通过函数：`Point * GetTransRslt( )` 得到。

其它函数介绍：

`int GetNumOfPoint( )` 获取参与变换的点的个数。

`int GetNumberOfParameters( )` 获取变换参数的个数。

## 7.4 图像插值

在配准过程中，相似性测度（Metric）通常是比较固定图像（fixed image）和变换图像（moving image）之间对应点的灰度值。当一个点通过某种变换，从

一个空间映射到另一个空间时，目标点的坐标通常不在网格点上。在这种情况下，插值算法就需要用来估计目标点的灰度值。变换示意图见 图 7-6：

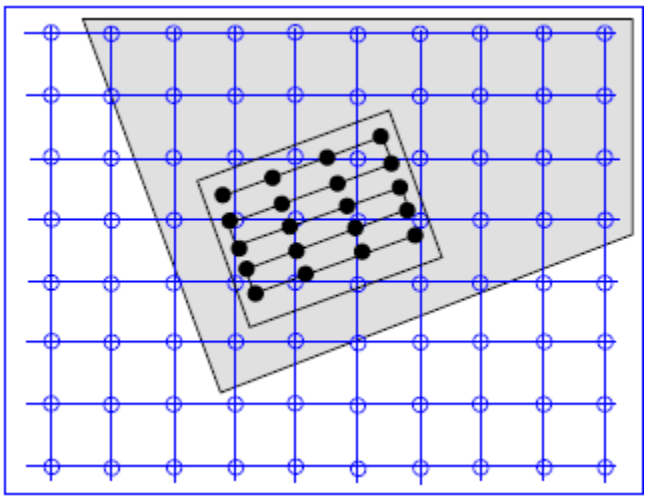


图 7-6 固定图像网格点与变换后的浮动图像非网格点

插值方法影像着图像的平滑性，优化的搜索空间，以及总体的计算时间。因为在一个优化周期之中，插值算法将被执行成千上万次。所以，在指定插值方案时，我们需要在计算复杂性和图像平滑性之间做一个权衡。

### 7.4.1 最近邻插值

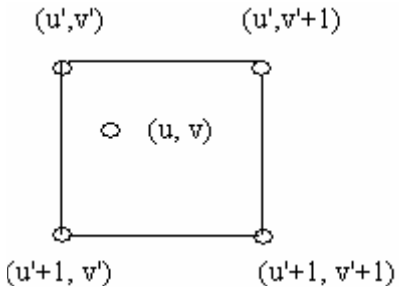



图 7-7 最近邻域法插值

最近邻域法是指把距离非相网点（u，v）最近的u-- v坐标系中的格网点的灰度值设为（u,v）点灰度值的算法。如 图 7-7 所示，其不足是会使细线状目标边界产生锯齿。

### 7.4.2 线性插值

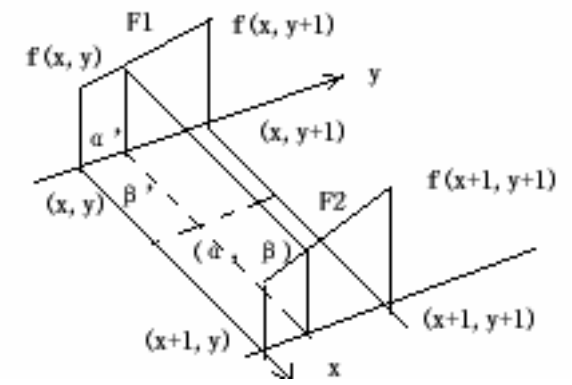



图 7-8 线性内插法

线性内插法是指如 图 7-8 所示，采用在（u,v）周围四个格网点的灰度值进行内插，其关系式为：

$$f(u,v)=(1-\alpha')(1-\beta')f(x,y)+\beta'(1-\alpha')f(x,y+1)$$
$$+\alpha'(1-\beta')f(x+1,y)+\alpha'\beta'f(x+1,y+1)$$

### 7.4.3 PV 插值

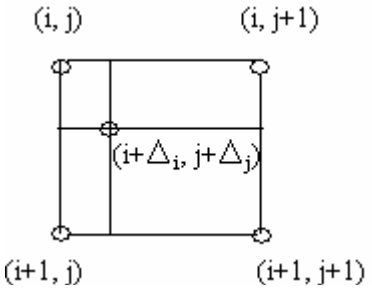



图 7-9 PV 插值

PV（Partial Volume）插值是指如 图 7-9 所示，通过周围 4 个点距对应点的距离分配分数权值，以使它们贡献于联合灰度分布统计。假设两幅待配准的图像为F（浮动图像）和R（参考图像）。令$T_\alpha$是由参数α确定的变换矩阵，它将被作用于F。假设$T_\alpha$将F中的点（a，b）映射到R中的坐标（i＋$\triangle_i$，j＋$\triangle_j$），其中（i，j）是R中的网格点，并且 0≤$\triangle_i$，$\triangle_j$<1。

如果 $f$ 是实函数，满足：

1. $f$（x）≥0，x 是实数

2. $\sum_{n=-\infty}^{\infty} f(n+\square)=1, n$是整数$, 0 \leq \square \leq 1$

那么，对于 F 中的任何点（a，b），联合灰度分布为：

h（F（a，b），R（i＋p，j＋q））＝h（F（a，b），R（i+p，j＋q））

＋$f$（p－△$_i$）* $f$（q－△$_j$）(7-9)

其中 p，q，r 是整数，$f$ 是核函数。

### 7.4.4 插值算法在 MITK 中的实现

从 图 7-2 中可以看出, 插值模块Interpolator接受变换模块Transform的输出作为其输入。另外，它还有一个mitkVolume的输入参数。经过插值运算后，Interpolator输出mitkVolume类型的参数，并将其作为metric模块的输入。

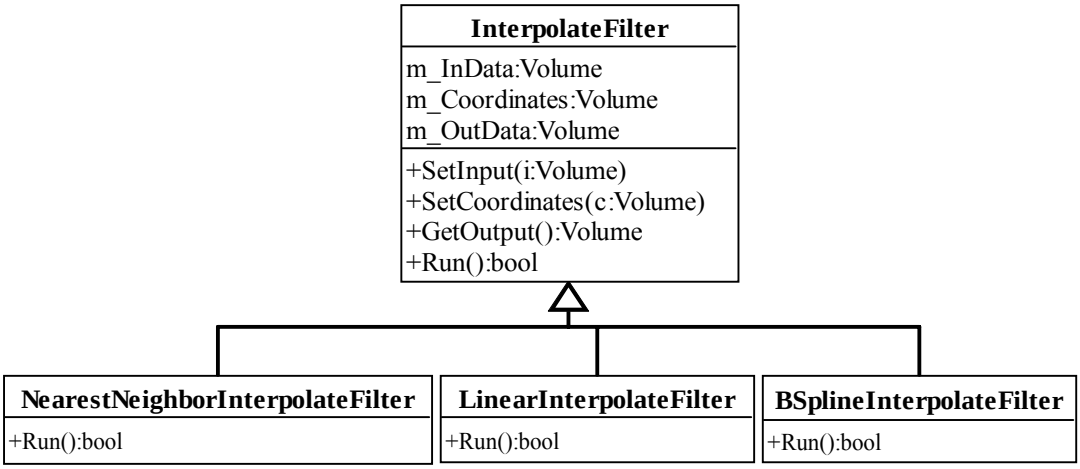



图 7-10 图像插值类图

Interoplator 模块函数介绍：

`void SetInput(mitkVolume *inData);` 设置图像数据

`void SetTransRslt(Point * p, int w, int h, int n);` 设置变换结果

在 Interoplator 中设置完图像数据和变换结果之后，就可以通过 Run（）调用 Excute（）执行具体的插值操作。

## 7.5 相似性测度

相似性测度（Similarity Measure）定量化地衡量了两幅图像匹配的效果，它是图像配准过程中十分重要的一部分[4]。一般情况下待配准的图像是在不同时间、不同条件、甚至不同成像技术下获取的，图像描述的信息可能存在本质的差别，这种情况下就没有绝对的配准问题，那么我们的任务就是寻找一种准则，使两幅图像在这种准则下达到最佳的匹配效果。这里的准则称之为相似性测度，在一些非刚性配准中还加上形变约束。准则的选择和配准目的、具体的图像形态、几何变换类型等有关。例如，有些准则允许很大的几何变换搜索范围，而有些准则要求初始位置和最优的配准结果比较接近才能得到正确结果；有些准则仅仅适用于同一模态图像间的配准，而有些准则能处理不同模态的图像配准。遗憾的是现在还没有一个明确的准则来指导在各种情况下如何选择配准的相似度量准则，更不存在各种情况下都通用的相似度准则。

既然配准只是在某种准则下取得相对最优,准则的选择直接影响着配准的效果。因此，如何选择合适的相似性度量准则就成为图像配准中一个十分关键的研究问题，大量的研究论文也表明了这一点。从发表的论文来看，主要有两种相似性度量准则：基于特征（feature-based）和基于体素（voxel-based）。基于特征的准则一般是最小化两图像相应特征间的距离，常用的特征有对应解剖结构中的控制点、二维边缘线、三维表面等。这种准则下，通常先要提取特征，利用这些局部的特征信息进行配准。特征提取的准确性直接影响着配准的精度。基于体素的方法是目前的研究热点。从理论上来讲，这种方法应该是最灵活的，因为它利用了图像中的所有信息。从总体上来看，这类准则中较常见的有：1、相关性测度，包括相关系数、傅立叶域的互相关和相位相关等；2、总体平均差最小化；3、灰度比的方差最小化；4、互信息最大化。其中基于互信息最大化的方法获得了很大的成功[5]，大量文献表明，该方法不仅适用于单模态图像配准，对多模态图像配准问题也能取得不错的结果。

MITK 中目前实现了下列基于体素的相似性度量准则：

- 灰度平均差（Mean Squares Metric）

- 归一化相关系数（Normalized Correlation Metric）
- Pattern Intensity
- 互信息（Mutual Information Metric）

下面我们先简要介绍这几种准则，然后看看其在 MITK 中的实现。为了书写方便，我们记参考图 $f(X)$ 和变换插值后的浮动图 $m(T_t(X)$ 为 $A$ 和 $B$ 。$A_i, B_i$ 分别是图像 $A,B$ 第 $i$ 个像素的灰度值， $N$ 是计算区域的像素个数。

### 7.5.1 灰度平均差测度

MitkMeanSquaresMetric 类计算图像 $A,B$ 在给定区域的灰度平均差：

$$MS(A,B)=\frac{1}{N}\sum_{i}^{N}(A_i-B_i)^2 \tag{7-10}$$

理想情况下的最优测度是 0，这是基于以下假设：两图像对应像素点灰度值相同。因此该准则只适用于同模态图像配准。该准则计算简单，相对来说可以在一个比较大的范围内搜索匹配。但该准则对图像灰度值的线性变化比较敏感。

### 7.5.2 归一化相关系数

MitkNormalizedCorrelationMetric 计算图像 $A,B$ 的归一化互相关系数

$$NC(A,B)=\frac{\sum_{i}^{N}(A_i\cdot B_i)}{\sqrt{\sum_{i}^{N}A_i^2\cdot\sum_{i}^{N}B_i^2}} \tag{7-11}$$

理想情况下该准则的最优值是 1。该准则也仅限于单模态图像配准，并且产生尖峰状极值，搜索范围较小。

### 7.5.3 Pattern Intensity

MitkPatternIntensityMetric 计算灰度差，并代入钟形函数 $\frac{1}{1+x^2}$ 求和。

$$PI(A,B)=\sum_{1}^{N}\frac{1}{1+\lambda(A_i-B_i)^2} \tag{7-12}$$

其中 $\lambda$ 是比例系数，控制搜索范围。该准则的具体性质可参考Penney[6]和Holden[4]的论文。该准则也仅限于单模态图像配准，对图像灰度值的线性变化比较敏感。

### 7.5.4 互信息

互信息(Mutual Information)是信息论中的一个概念，通常用于描述两个系统

间的统计相关性，或一个系统中包含的另一个系统中信息的多少，一般用熵(Entropy)来表示。

随机变量 $A$ 的熵定义为

$$H(A) = -\int p_A(a)\log p_A(a)da \tag{7-13}$$

两个随机变量 $A$、$B$ 的联合熵定义为

$$H(A,B) = -\int p_{AB}(a,b)\log p_{A,B}(a,b)dadb \tag{7-14}$$

如果 $A$、$B$ 互相独立，则

$$p_{AB}(a,b) = p_A(a)p_B(b) \tag{7-15}$$

$$H(A,B) = H(A) + H(B) \tag{7-16}$$

如果 $A$、$B$ 不互相独立，则

$$H(A,B) < H(A) + H(B) \tag{7-17}$$

其差值称为 $A$、$B$ 的互信息 $I(A,B)$

$$I(A,B) = H(A) + H(B) - H(A,B) \tag{7-18}$$

在医学图像配准中，待配准的两幅图像可能来自于不同的时间或不同的成像设备，但它们都基于共同的人体解剖信息，它们被看作两个随机变量时，二者显然不会相互独立，并且我们假设当它们的空间位置达到一致时，其互信息应为最大。这就是用互信息最大化作为相似性测度的原理所在。

将待配准的两幅图像看作二维随机变量 $(A,B)$ 计算其互信息之前，首先需要估计该二维随机变量的边缘概率分布密度函数 $p_A(a)$、$p_B(b)$ 和联合概率分布密度函数 $p_{AB}(a,b)$。一个最直观的想法就是用图像的灰度级（联合）直方图来估计，即各灰度级出现的相对频率作为该灰度级像素的概率；常用的另一种估计方法是采用著名的 Parzen 窗概率密度估计（Parzen Window density estimate）$P^*(z)$：

$$p(z) \approx P^*(z) = \frac{1}{N_A}\sum_{z_i \in A} R(z - z_i)\,, \tag{7-19}$$

其中，$N_A$ 是图像 $A$ 的像素个数（或者说随机变量 $A$ 的样本数），$R$ 是窗函数，其定义域内积分等于 1。窗函数常采用高斯函数 $G_\Psi(z)$：

$$G_\Psi(z) \equiv (2\pi)^{-n/2}\left|\Psi\right|^{-1/2}\exp(-\frac{1}{2}z^T\Psi^{-1}z)\,。 \tag{7-20}$$

估计出边缘概率密度和联合概率密度函数后，可以很容易求出两图像的互信息。有文献指出待配准两图像重叠区的大小会影响互信息[5]，用规整化互信息NMI（normalized mesure of mutual information）或熵相关系数ECC作为相似性测度函数能克服重叠区的影响，其中NMI和ECC计算公式如下：

$$NMI(A,B)=\frac{H(A)+H(B)}{H(A,B)}, \tag{7-21}$$

$$ECC(A,B)=\frac{2I(A,B)}{H(A)+H(B)}, \tag{7-22}$$

显然NMI和ECC有如下关系：

$$ECC(A,B)=2-2/NMI(A,B)。 \tag{7-23}$$

采用基于体素方法计算相似度时，对于高分辨率的三维体数据集，其包含的数据量极大，为减少计算量，常只采用部分而不是全部数据点，计算前先对原图像进行重采样。

### 7.5.5 相似性测度在 MITK 中的实现

从 图 7-2 可以看出，算法上相似性测度模块Metric接受Fixed Volume和Image Interpolator的输出作为输入，即输入两个Volume体数据，输出的是计算得到的相似度量值。在MITK具体实现中，我们用Metic这一抽象基类来实现相似性测度，其框架如 图 7-11 所示。

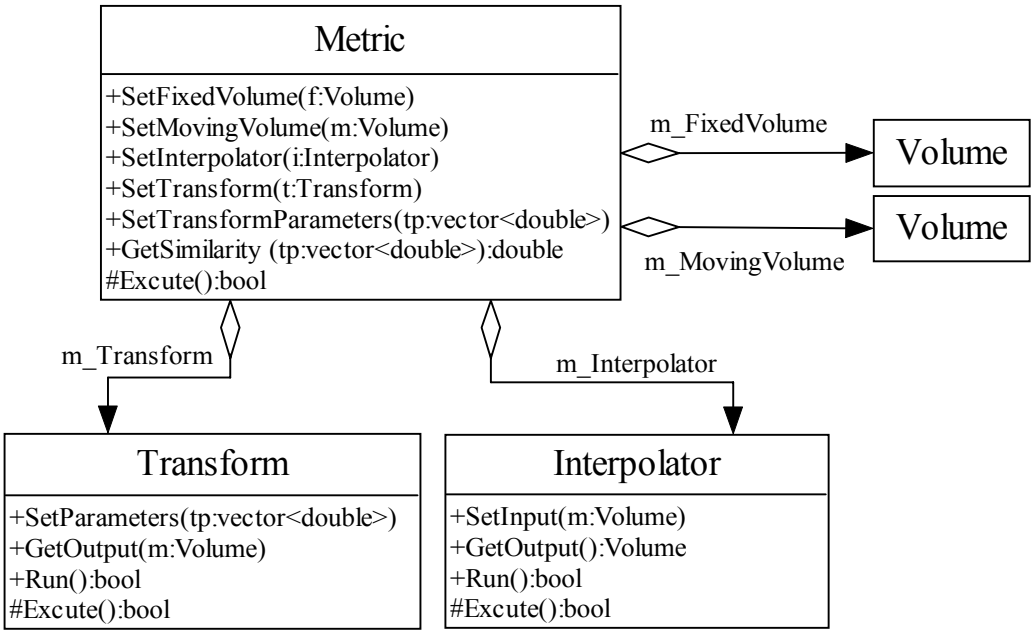



图 7-11 相似性测度模块实现

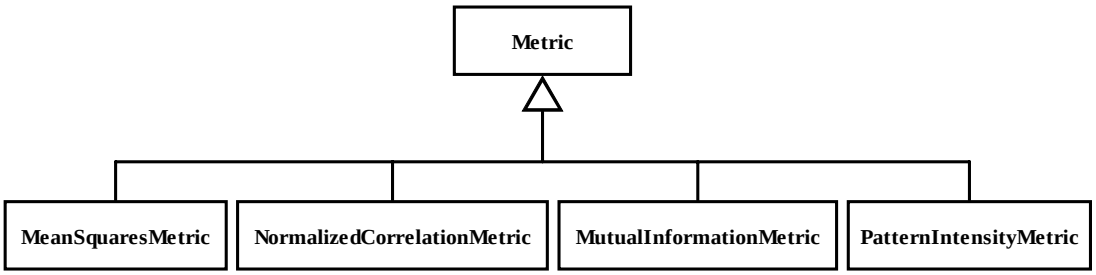



图 7-12 相似性测度 Metric 层次结构

我们将Transform和Interpolator作为Metric的成员变量封装起来，在bool Execute()函数中完成一次性完成几何变换、图像插值以及求相似度的工作。各种不同的相似性测度都通过Metric派生出来，并在虚函数Execute()中具体实现。如图 7-12 所示。

对于各种相似性测度，用户接口都是一致的，即通过成员函数 void SetFixedVolume(mitkVolume *fixedVolume)输入参考图数据信息，通过函数 void SetMovingVolume(mitkVolume *movingVolume)输入浮动图数据信息，通过 void SetTransform(mitkTransform *transform) 和 void SetTransformParameters(vector<double> *parameters)两函数指定几何变换的类型

及其相应的参数，通过函数 void SetInterpolator(mitkInterpolator *interpolator)指定图像插值的类型。算法计算结果通过 double GetSimilarity(const vector<double> *parameters)得到。

## 7.6 函数优化

根据相似性测度选择的不同，配准变换的参数求解方式可分成两类，一是从获得的数据用联立方程组直接计算得到的，二是以对定义在参数空间的能量函数最优化搜索得到的。前者完全限制在基于特征(feature-based)的配准应用中。在后者中，所有的配准问题都变成一个能量函数的极植求解问题，能量函数是由需要被优化的变换参数表示的，一般是拟凸的，能用标准的优化算法求解极值。图像配准问题本质上是多参数优化问题，所以优化算法的选择至关重要。常用的优化算法有：Powell 法、下山单纯形法、Arent 法、Levenberg-Marquadrt 法、Newton-Raphson 迭代法、随机搜索法、梯度下降法、遗传算法、模拟退火法、几何 hash 法、半穷尽搜索法。在实际应用中，经常使用附加的多分辨率和多尺度方法加速收敛、降低需要求解的变换参数数目、避免局部最小值，并且多种优化算法混合使用，即开始时使用粗略的快速算法，然后使用精确的慢速算法。

图像配准中的优化函数就是相似性测度函数 $S(t)$，其中 $t$ 是 $n$ 维向量，表示几何变换中的 $n$ 个参数。这里的优化函数 $S(t)$ 有几个显著特点：

1. 一般来说，$S(t)$ 不够平滑，存在很多局部极值点。这给优化算法的选择提出了很高的要求，由于这些局部极值点的存在，优化结果会对初始值（初始变换）比较敏感，导致配准结果不够鲁棒。

2. $S(t)$ 中各参数的变化对函数结果的影响因子差别较大。比如在刚性变换下，旋转比平移引起的图像变化要大。因此，最好给每个参数分配一个缩放因子，使各参数以合适的步长进行迭代搜索，减少搜索时间、提高计算精度。另外还要优化各参数的搜索顺序，比如考虑到成像过程中，病人在 $xy$ 平面的旋转和平移比其它方向的转转和平移都要大，故可以先搜索 $xy$ 平面的旋转和平移参数。

3. 在有些情况下，$S(t)$ 的全局最优并不带来最好的配准效果，由于相似度函数并不包括配准图像的所有有效信息，两幅图像大面积不匹配时反而比正确匹配时具有更大的相似度。因此要合理选择优化参数的取值范围。

从已有的文献来看，比较多的采用 Powell 程序进行优化，MITK 中 PowellOptimizer 实现了这种算法。

MITK 中函数优化模块用 Optimizer 表示，继承自 ObiectProcess 类，其最关键的输入就是优化函数（相似性测度），这通过函数 void SetMetric(mitkMetric *metric)设定，在 Metric 类的成员函数 double GetSimilarity(const vector<double> *parameters)中得到具体体现。除此之外比较重要的接口函数有：

void SetScales(const vector<double> *scales)设定各几何变换参数的尺度因子；

void SetInitialPosition( const vector<double> *param )设置初始变换参数；

各种具体的优化算法也同Metric模块类似，通过Optimizer的派生类在bool Execute()函数中实现。如 图 7-13 所示。

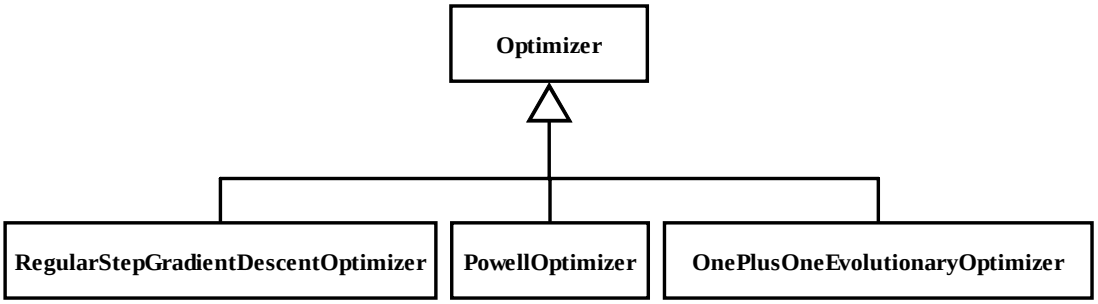



图 7-13 优化模块 Optimizer 层次结构

## 7.7 配准算法实现

以上 Transform、InterpolateFilter、Metric、Optimizer 四模块从高层概念上实

现了相应的算法，提供基本的数据成员和成员函数接口。不同类型的各种具体算法通过它们的派生类在虚函数 Run()中实现，配准类 RegistrationFilter 的 Run()函数则组合这四个模块，形成一个完整的算法流程，配准结果通过相应的 Get 函数得到。

为了提高算法的易用性和灵活性，我们在RegistrtionFileter中也提供了各常用数据和参数的输入、输出接口，如 图 7-14 所示。用户通过设置具体的几何变换类型、插值方法、相似性度量准则、优化策略就能完成各种不同的配准算法。

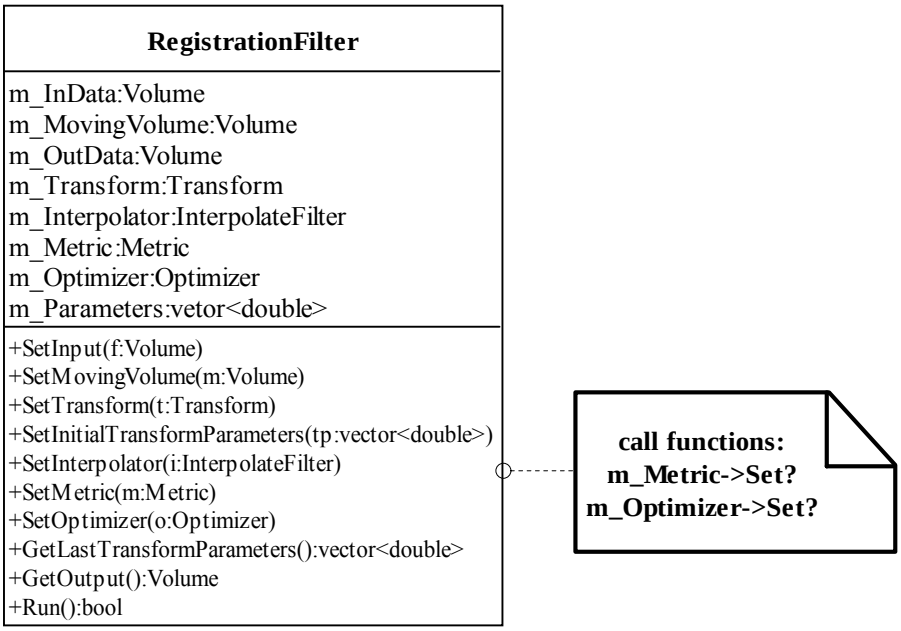



图 7-14 配准类图

## 7.8 应用实例

下面用一个简单的例子来说明如何使用 MITK 中的配准算法来实现两幅图像的配准。初始化参数均采用默认值。

```
//需包含以下头文件
#include "mitkRegistrationFilter.h"
#include "mitkRigidTransform.h"
#include "mitkMeanSquareMetric.h"
#include "mitkNearestNeighborInterpolateFilter.h"
#include "mitkPowellOptimizer.h"
#include "mitkVolume.h"
//初始化配准算法
```

```
mitkVolume* fixedVolume = new mitkVolume;
mitkVolume* movingVolume = new mitkVolume;
mitkRegistrationFilter* registration = new mitkRegistrationFilter;
mitkRigidTransform* transform = new mitkRigidTransform;
mitkMeanSquareMetric* metric = new mitkMeanSquareMetric;
mitkNearestNeighborInterpolateFilter* interpolator = new
mitkNearestNeighborInterpolateFilter;
mitkPowellOptimizer* optimizer = new mitkPowellOptimizer;
registration->SetInput(fixedVolume);
registration->SetMovingVolume(movingVolume);
registration->SetTransform(transform);
registration->SetInterpolator(interpolator);
registration->SetMetric(metric);
registration->SetOptimizer(optimizer);
//开始配准
registration->Run();
//输出结果
vector<double>* parameters;
mitkVolume* outVolume;
parameters = registration->GetLastTransformParameters();
outVolume = registration->GetOutput();
```

## 7.9 小结

本章介绍了不同的图像配准算法在 MITK 中的实现策略，通过算法分析与抽象，在高层概念上整个配准算法被分成四个相互独立的模块：Transform、Interpolate、Metric、Optimizer，每个模块下再派生出各种不同类型的具体实现，最后将这四个模块按照固定的流程搭配在一起就可以组合出多种多样的配准算法。这样一种框架结构保证了配准算法的扩充性，新的算法可以通过相应模块下的派生类实现，而且各种算法的性能也能够在这一致的框架结构中方便地进行比较。

MITK 中目前的配准算法还主要是考虑刚性配准算法的实现，并且还在不断改善的过程中。 随着各种多模态非刚性配准算法的出现和完善[7]，比如基于流体模型、基于有限元的方法等，这些新的算法也将逐渐加入 MITK 中。

参考文献

1. Brown L, A Survey of Image Registration Techniques, ACM Computing Surveys (CSUR), 1992，24(4):325-376.
2. 田捷，包尚联，周明全. 医学影像处理与分析. 北京：电子工业出版社，2003.
3. J. Schnabel, et al. A Generic Framework for Non-rigid Registration Based on Non-uniform Multi-level Free-form Deformations, Medical Image Computing and Computer-Assisted Intervention (MICCAI 2001), Utrecht, NL, 2001.
4. G. P. Penney, J. Weese, J. A. Little, P. Desmedt, D. L. G. Hill, and D. J. Hawkes. A comparision of similarity measures for use in 2d-3d medical image registration. *IEEE Transactions on Medical Imaging*, 17(4):586–595, August 1998.
5. J. Pluim, J. Maintz, M. Viergever, Mutual-Information-Based Registration of Medical Images: A Survey, IEEE Transaction on Medical Image, 22(8):986-1004, 2003.
6. M. Holden, D. L. G. Hill, E. R. E. Denton, J. M. Jarosz, T. C. S. Cox, and D. J. Hawkes. Voxel similarity measures for 3d serial mr brain image registration. In A. Kuba, M. Samal, and A. Todd-Pkropek, editors, *Information Processing in Medical Imaging 1999 (IPMI'99)*, pages 472–477. Springer, 1999.
7. H. Lester, S. Arridge, A Survey of Hierarchical Non-linear Medical Image Registration, Pattern Recognition, 32:129-149, 1999.

# 8 DICOM 标准的实现

## 8.1 DICOM 标准简介

计算机技术的发展，大容量存储介质和图像压缩技术的应用，使医学图像可以大量存储；计算机运行速度的提高，使得对图像的实时分析成为可能；计算机显示技术和虚拟现实技术的发展，使得医生不用开刀就可以看到病人体内逼真的三维图像。在这些医学图像诊断的技术当中，现在的热点和关键问题之一是医学图像及诊断数据的存储和传输问题。

八十年代以来，为了利用网络在不同的设备和医疗诊断系统之间交换图像数据和诊断信息，国外已经开始着手制订专门针对医学信息通信的协议。1983 年，美国放射学会（American College of Radiology——ACR）和美国电器制造商协会（National Electrical Manufacturers Association——NEMA）成立了一个联合委员会开发相关标准，并于 1985 年发布了 ACR-NEMA 标准。经过多年的增补和修改，ACR-NEMA 标准最终演变成为新一代的 DICOM3.0（Digital Imaging and Communications in Medicine）标准[1]。在这个标准中，增强了对网络的支持，成为医学影像设备的国际标准通信协议。现在，各个厂家的医疗仪器和医学诊断系统都已开始使用国际化的通信和数据格式标准 DICOM3.0。

### 8.1.1 DICOM 标准的产生和演化

DICOM 标准的演变历史大致可以划分为如下三个阶段：

（1） ACR-NEMA 标准 1.0 版（1985 年）

美国放射学会（ACR）和美国电器制造商协会（NEMA）组成的联合委员会经过两年开发，于 1985 年发布了 ACR-NEMA1.0。1986 年 10 月和 1988 年 1 月又分别颁布了两个修改版本。在这些标准当中，解决了以下问题：

- 统一数据格式和传输标准，实现了不同厂家不同设备间数字化医疗数据的通信问题；
- 提供了PACS（Picture Archiving and Communication System）系统与其它医学信息系统（Hospital Information System——HIS）的接口[4]；
- 通过将医生诊断信息数据随同病人图像数据按统一格式提供给不同的

医疗设备和医疗工作站，并建立诊疗数据库，使得不同科室的医生可以方便地查询相关病人的完整信息。

（2） ACR-NEMA 标准 2.0 版（1988 年）

1988 年 ACR-NEMA 标准颁布了 2.0 版。在这个版本中，除包含 ACR-NEMA1.0 的内容之外，还包括了以下的修改和补充：

- 提供了对显示设备的命令支持；
- 引入新的层次结构模型，用以更清晰地标识医疗图像；
- 增加新的数据元素以描述医学图像的相关信息；
- 指定硬件接口标准，以及相关的基本命令集和数据编码格式集。

（3） DICOM 3.0 标准（1995 年）

为了更有效地支持医学信息系统 HIS/RIS（Hospital Information System/Radiology Information System）对网络通信的要求，从 1989 年开始，ACR-NEMA 着手制订新一代的医疗数据通信标准。为了有别于早期的 ACR-NEMA 标准，将其命名为医学数字图像数据通信标准 DICOM 3.0（Digital Imaging and Communications in Medicine）。DICOM 3.0 标准的制订工作直到 1995 年才基本完成。在此期间，主要经历了如下过程：

- 1991 年，发布 DICOM 标准的 1—8 章，规定了 DICOM 通信和数据编码的主要内容；
- 1992 年，北美放射年会（Radiology Sociaty of North America——RSNA）发布对标准 1—8 章的测试结果；
- 1993 年，完成对 DICOM 标准的 1—8 章的修订，增加了第 9 章（点到点通信支持）；
- 1994 年，增加第 10 章（媒体存储和文件格式）；
- 1995 年，增加第 11，12，13 章，分别规定了有关建立媒体存储方式存档，物理媒体数据格式以及点到点打印管理的有关内容；
- 与早期的 ACR-NEMA 标准相比，DICOM 3.0 体现了许多方面的改进和增强。表 8-1 是对 ACR-NEMA 2.0 标准和 DICOM 3.0 标准主要特点的

对比。

表 8-1 ACR-NEMA 2.0 和 DICOM 3.0 的主要特点对比

| ACR-NEMA 2.0标准 | DICOM 3.0 |
|---|---|
| 基于直观的信息模型 | 基于明确的信息模型，使用E-R模型描述信息对象之间的关系 |
| 提供的服务在命令中组成，只能以命令的形式完成数据的交换 | 支持的服务根据其用途描述，引入服务类的概念描述命令及与其相联系数据的语义 |
| 定义了符合标准的最低要求，但没有描述标准符合声明的规定 | 包含描述标准符合声明的规定，允许存在不同级别的标准符合声明以供用户选择 |
| 只支持点对点通信，在网络环境中需要特定的网络接口设备才能支持网络协议 | 支持网络通信，提供对OSI七层协议和TCP/IP协议等工业标准的全面支持 |
| 一个消息中最多包含一幅图像 | 支持文件夹功能，一个消息中可以包含多个图像 |
| 定义了ACR-NEMA独有的命令 | 使用现存的其他标准定义命令，规定了符合本标准的设备对命令的反馈和数据交换方式 |

医学信息数据通信领域正在经历一个飞速发展的阶段，大量新的医疗设备、计算机诊断系统和诊疗理念不断丰富着这一领域。因此，DICOM 标准从诞生之日起就是一个开放的标准，它不断地进行着自我完善、扩充和演化的过程。为了适应这种内容不断扩充和更新的需要，DICOM 标准采用了广义的信息对象定义 IOD（Information Object Definition）的概念，不仅包括医学图形和图像，也包括大量相关的检查、报告等广义的信息对象，并且通过唯一标识 UID（Unique Identifier）的方式在网络环境下唯一地确定这些信息对象。在此基础上，DICOM

标准定义了大量不同的服务类（Service Class），用以完成不同的服务功能。因此，对于不断更新的医疗设备及其各种类型的数据，可以通过修改或定义新的 IOD 使得标准与之相适应并得以扩充；而对于新增的功能，则可以通过定义新的服务类来完成。这样就达到了不需对 DICOM 标准的整体架构进行修改的情况下，完成标准本身不断扩充和更新的目的。也就是说，DICOM 标准可以在不断吸收新特性的同时，仍能很好地保持与原先版本的兼容性[1]。

实际上，ACR-NEMA 每年都会公布当年新修订的 DICOM 版本。MITK 中部分 DICOM 标准的实现正是基于 DICOM 3.0 的 2003 年版。

### 8.1.2 DICOM 标准的主要特点

作为医学信息通信领域的国际标准，DICOM 具有以下一些突出特点：

（1） DICOM 协议是一种上层网络协议[1]

DICOM 协议处于 OSI 开放系统互联七层协议的上三层，即会话层、表示层和应用层的位置，而在七层协议的下层主要使用 TCP/IP 协议所提供的服务。DICOM 协议要求在数据的编码、传输之前，必须先进行连接协商以确认双方“同意”某些特定的条件，可以完成特定的通信功能。DICOM 称连接协商的成功为建立了一个“关联”（association）。只有在建立“关联”之后，才能进行 DICOM 命令和数据的发送和接收。

（2） DICOM 数据编码的特点

- 标准定义了 26 种内部数据类型；
- 像素数据的编码支持 JPEG 图像压缩；
- 图像可以包含缩略图和正常图像，也可以有多帧格式；
- DICOM 标准支持多个字符集；
- DICOM 具有自己独特的数据模型；
- DICOM 通过“信息对象定义”IOD（Information Object Definition）的形式来完整地建立和定义医院环境下的数据模型；
- DICOM 使用“全局唯一标识”UID（Unique Identifier）在网络环境

下唯一地标识各种 IOD 信息对象，使之不致混淆。

（3） 拥有完整、庞大的数据字典

DICOM 标准拥有一个庞大的数据字典，其内容包含了几乎所有医疗环境下的常用数据，可以完整地描述各种医学设备、图像格式数据以及病人相关信息。

数据字典的条目以“数据元素”（Data Element）为单位，每个数据元素描述一项数据内容，如病人姓名、检查日期、一幅图像的像素数据等都可以是一个数据元素。

数据字典具有可扩充性。DICOM 标准预留出数据字典的一部分，允许各厂家按照标准的格式自定义新的数据元素。

（4） 通过“服务类”（Service Class）概念实现应用层功能

为了完成某个特定的应用功能（如图像管理、打印管理等），DICOM 定义了“服务类”（Service Class）的概念。服务类描述了可以对信息对象 IOD 所做的操作。服务类和信息对象结合起来构成了 DICOM 的基本单元，称为服务-对象对（Service-Object Pair——SOP）[1]。表 8-2 列举了一些典型的服务类及其描述：

表 8-3 DICOM 服务类举例

| 服务类 | 描述 |
|---|---|
| Image storage | 提供数据集的存储服务 |
| Image query | 支持数据集的查询 |
| Image retrieval | 支持从存储设备检索图像 |
| Image print | 提供硬拷贝图像生成服务 |
| Examination | 支持检查管理 |
| Storage resource | 支持网络数据存储资源管理 |

（5） 离线媒体支持

DICOM 定义了自己的文件夹结构，用以形成“文件集合”（File-set）。此外，允许以“媒体存储特征”（media profiles）的形式定义对数据的不同媒体存储策略。

（6） 不同级别的一致性声明（Conformance Statement）

DICOM 标准要求一个实际的通信系统应该有一个一致性声明（Conformance Statement），用以说明此系统对 DICOM 协议的支持程度，以及支持哪些类型的数据和服务。

### 8.1.3 DICOM 标准的总体结构和主要内容

2003 版的 DICOM 标准分为以下 16 个相关而又相对独立的部分[1]：

（1） DICOM 标准概要（Introduction and Overview）；

（2） 一致性声明（Conformance）；

（3） 信息对象定义（Information Object Definitions）；

（4） 服务类规范（Service Class Specifications）；

（5） 数据结构和语义（Data Structure and Semantics）；

（6） 数据字典（Data Dictionary）；

（7） 消息交换（Message Exchange）；

（8） 消息交换的网络通信支持（Network Communication Support for Message Exchange）；

（9） 消息交换的点到点通信模式（Point-to-Point Communication Support for Message Exchange）；

（10） 媒体存储和文件格式（Media Storage and File Format）；

（11） 媒体存储策略（Media Storage Application Profiles）；

（12） 数据交换的存储功能和媒体格式（Storage Functions and Media Formats for Data Interchange）；

（13） 点到点打印管理（Print Management Point-to-Point Communication Support）；

（14） 标准灰度显示功能（Grayscale Standard Display Function）；

（15） 安全策略（Security Profiles）；

（16） 内容映射资源（Content Mapping Resource）。

由于点到点通信已逐渐被网络通信所代替，因此其中与“点到点通信”有关的第（9）和（13）部分已被标记为 RETIRED，不再更新。

上述 16 个部分大致可以归结为以下几个主要的方面：

（1） 一致性声明（Conformance）

这部分提出了医疗设备或诊断系统为满足 DICOM 标准的要求，所必须遵循的规范。它详细地规定了规范声明的层次结构，以及声明中各部分所必须包含的内容。实际上，DICOM 标准的各个部分都有自己的规范声明。此处的“一致性声明”是各部分规范声明的汇总。

一致性声明可以分为几个主要部分：

- 本医疗设备或诊断系统（即应用实体）所支持的 DICOM 信息对象；
- 应用实体系统支持的服务类；
- 应用实体系统支持的通信协议，如 TCP/IP 协议等；
- 所支持的表示上下文信息；
- 系统配置信息。

一致性声明的意义在于：

由于 DICOM 协议十分庞大，以至于至今仍没有一个公司或团体已经将其完全实现。事实上，往往只需要实现其中一部分功能即可满足实际需要。因此，允许实现者根据需求选择支持哪些 DICOM 组件，也允许进行扩展。不同的系统会有不同的支持部分，所以也会有不同的一致性声明[1]；

用户或系统设计人员通过对比两种不同实现的一致性声明，就能够判断出两个系统是否可以进行互操作和通信。

（2） 信息对象定义（Information Object Definitions）

这一部分具体介绍了 DICOM 标准面向对象的信息结构模型，即用信息对象定义 IOD 来描述现实世界中的各种医疗信息实体。这里详细定义了多种 IOD，并规定了它们的内部结构。

DICOM 用 IOD 定义服务类所作用的对象的数据结构和属性，IOD 是对现实世界中具有共同属性实体的面向对象的抽象。为了方便标准的扩展并保持与以前版本的兼容性，DICOM 定义了两种 IOD，即正规 IOD（Normalized IOD）和复合 IOD（Composite IOD）。每个 IOD 由用途说明和大量相关的“属性”组成，但 IOD 本身并不包括各个“属性”的值，而是只包含其定义。这些属性描述的内容虽然千差万别，却拥有几乎相同的结构。属性又按照一定规则分成组，以利于被不同的 IOD 复用。

当需要表示现实世界的某个实体时，就要将相应的 IOD 实例化，这时 IOD 属性的值被填充进来。这些属性值在下文介绍的服务类作用下可以不断发生变化。

（3） 服务类规范（Service Class Specifications）

标准的这一部分指定作用于信息对象实例上的操作，即所提供的服务，如图像的存储、查询、检索、打印等，并称之为服务类。一个特定的服务类可通过一至多个命令作用于一至多个 IOD 实例。这里具体阐明了每个服务类对其命令元素的规范要求，以及通信服务的提供者和使用者应完成的功能。这一部分还提供了以下一些服务类的实例：

- 存储服务类（Storage Service Class）
- 查询服务类（Query Service Class）
- 检索服务类（Retrieval Service Class）
- 检查管理服务类（Study Management Service Class）
- 数据结构和编码（Data Structure and Encoding）

这一部分规定了服务类为完成特定操作而交换的数据的构造过程和编码结构。服务类的命令和 IOD 数据都要经过编码成指定结构的数据流，才能形成“消息”并完成发送过程；同样，要接收网络上的命令和数据，必须以逆过程进行

解码。数据可以分为“命令集”（Command-set）和“数据集”（Data-set），数据集允许嵌套。

（4） 数据字典（Data Dictionary）

数据字典标明了已注册的 IOD 属性数据的类型、标识和含义。一般来说，应包含每个属性的如下特征：

- 属性的唯一标签（tag）：包括一个组号和一个元素号，用户可用这两个号码检索这个属性；
- 属性的名称：用于了解其含义；
- 属性的数据类型（如 character string, integer 等）。

数据字典被用于在通信过程中辅助建造数据集。

（5） 消息交换（Message Exchange）

DICOM 标准的这一部分指定了为达到特定医疗信息交换的目的而引入的操作和协议。这些操作用来完成服务类所定义的相应服务。一个典型的 DICOM 消息由一个命令流和紧随其后的数据流（可选）组成。这里定义了各服务类所发送和接收的消息，并阐述了以下规则：

- 建立和终止“关联”（association）的规则；
- 控制交换网络命令请求和响应的规则；
- 用于建造命令流数据和消息的编码规则；
- 消息交换的网络通信模式（Network Communication Support for Message Exchange）。

这部分是 DICOM 通信协议的核心，它详细制订了医学图像和相关信息在网络上通信所需使用的服务和协议的具体内容。这些服务和协议内容要协调并保证网络上的 DICOM 应用实体间通信的效率。

DICOM 通信协议是 OSI 七层协议的一个子集，它主要包括一些 OSI 上层服务，包括表示层服务（ISO8822）和 OSI 规定的连接控制服务单元（Association Control Service Element——ACSE，ISO8649）的协议服务内容。在此基础上，对等应用实体间能够建立关联，传送消息和终止关联[1]。

DICOM 协议广泛地支持现有的网络环境和技术，如 ISO8802-3 CSMA/CD（Ethernet），FDDI，ISDN，X.25 等；DICOM 协议与 TCP/IP 协议相结合可以很好地实现其功能。从 TCP/IP 网络环境向其它 OSI 环境移植也很方便。

## 8.2 MITK 中 DICOM 标准的实现

MITK 中 DICOM 标准的实现基于 NEMA 发布的 2003 年版 DICOM 3.0 标准。但是由于 DICOM 标准十分庞大，MITK 不可能也没有必要将其完全实现。MITK 所需要的只是从存储设备上读取 DICOM 格式的文件以及将处理过的图像数据存储为简单格式的 DICOM 文件，所以 MITK 中所实现的部分实际上是 DICOM 标准中关于文件读写的一个子集，主要涉及的部分有：信息对象定义、数据结构和语义、数据字典以及媒体存储和文件格式等。

MITK 中完成 DICOM 文件读写功能的类是 mitkDICOMReader 和 mitkDICOMWriter。

考虑到 DICOM 标准本身的规模，我们将实现 DICOM 文件读写功能的基本数据结构及算法提取出来成为一个相对独立的部分作为 MITK 的一个 Utility，而 mitkDICOMReader/Writer 只是对这个 Utility 的一个封装，面向 MITK 用户提供 DICOM 文件的读写功能，如图 8-1 所示。

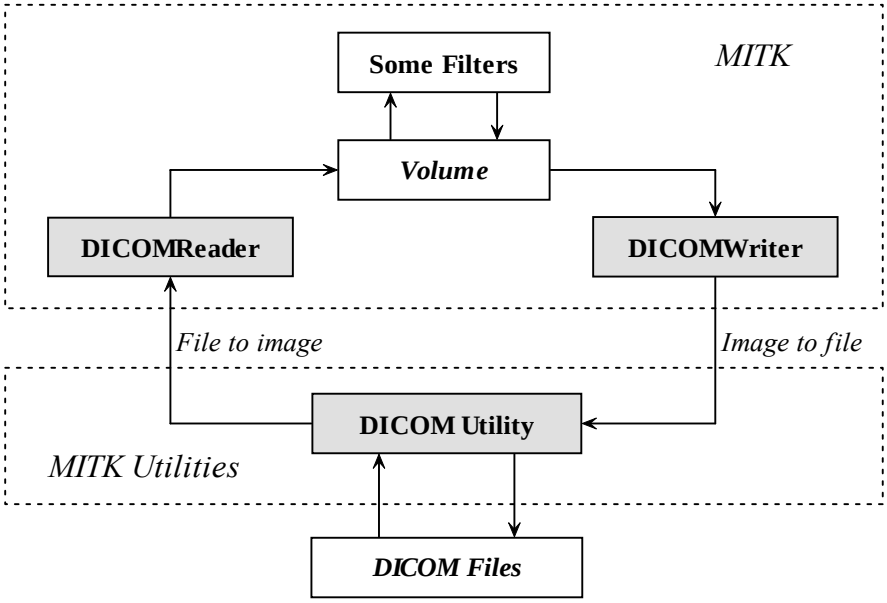



图 8-1 MITK 中 DICOM 文件读写部分的基本框架

### 8.2.1 DICOM 数据编码方式和文件结构[1][1]

(1) DICOM 数据编码规则

**1.** DICOM 数据流的结构

在 DICOM 标准中，真实世界中的某个信息对象（Information Object）被编码为一个数据集（Data Set），从而对其进行存储或传输。数据集以数据元素（Data Element）为编码单元，每一个数据元素存储了该数据集某一个属性的值，其编码格式如图 8-2 所示。

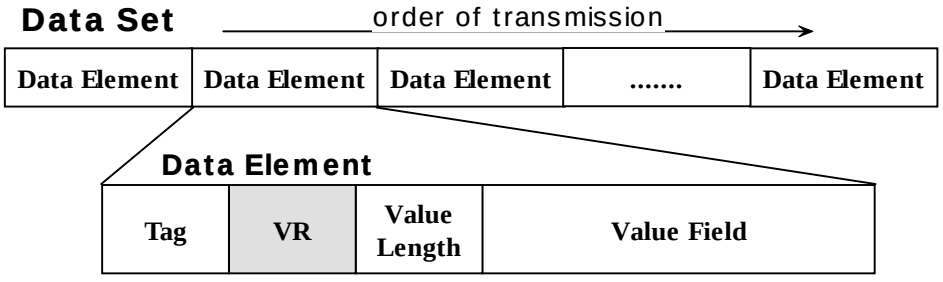



图 8-2 Data Set 编码格式

其中，每个数据元素包括一下一些字段的内容：

（1） 标签（Tag）

标签是一对 16 位无符号整数（占 4 字节），其格式用 16 进制表示为（gggg，eeee），其中，gggg 是组号（Group Number），eeee 是元素号（Element Number）。组号表示该数据元素属于哪一组属性（性质相近或相关的属性被编入同一组中），比如有关病人的信息（姓名、性别之类）都被编入 0010 组；而元素号则用于区分同一组中的不同属性的数据元素。由组号和元素号组成的标签唯一标识了某个数据元素的属性，数据字典即通过标签值来检索特定的数据元素，比如标签值为（0010，0010）的数据元素所存储的数据为病人姓名。

DICOM 标准定义了两大类的数据元素：

- 标准数据元素：除（0000，eeee）、（0002，eeee）、（0004，eeee）和（0006，eeee）之外组号为偶数的数据元素。这些都是已经由 DICOM 标准规定了具体属性的数据元素，其含义可以通过数据字典检索得到。组号为 0000、0002、0004 和 0006 的数据元素为 DICOM 标准所保留，主要用于 DIMSE（DICOM Message Service Element）命令以及 DICOM 文件格式；
- 私有数据元素：除（0001，eeee）、（0003，eeee）、（0005，eeee）、（0007，eeee）和（FFFF，eeee）之外组号为奇数的数据元素。这类数据元素可以由用户定义其属性含义，属于扩展部分，不编入数据字典。通用的 DICOM 数据解析程序一般不处理这一部分数据元素。组号为 0001、0003、0005、0007 和 FFFF 的数据元素同样为 DICOM 标准所保留。

另外，没一组的第一个元素，即标签为（gggg，0000）的数据元素是一个特殊的数据元素，它记录了某个数据集中所有属于该组的数据元素的总长度。

（2） 数据类型（Value Representation，VR）

数据类型是一个 2 字节的字符串，表示该数据元素的数据域（Value Field）所存储数据的类型。

该字段是可选字段。某个数据集中的数据元素是否包含这个字段，由该数据集的传输语法（Transfer Syntax）所决定，即若传输语法规定为隐式数据类型

（Implicit VR），则该字段被省略，在解析过程中可以根据数据元素的标签值，通过检索数据字典找到该数据元素的数据类型；而当传输语法规定为显式数据类型（Explicit VR）时，每个数据元素则必须包含该字段。

DICOM 标准所规定的数据类型如表 8-3 所示。

表 8-3 数据类型描述

| VR 名称 | 对应数据域的格式描述 | 值长度（字节） | 对应C++类型 |
|---|---|---|---|
| AE | Application Entity，字符串 | ≤16 | char* |
| AS | Age String，格式为nnnD、nnnW、nnnM或nnnY的字符串，nnn是一个3位十进制数表示天数（D）、周数（W）、月数（M）或年数（Y） | 4 | char* |
| AT | Attribute Tag，一对16位无符号整型数 | 4 | unsigned short[2] |
| CS | Code String，字符串 | ≤16 | char* |
| DA | Date，格式为yyyymmdd的日期字符串 | 8 | char[8] |
| DS | Decimal String，用字符串表示的实数，如12.3、-1.234E5等 | ≤16 | char* (to double) |
| DT | Date Time，格式为yyyymmdd hhmmss.ffffff &zzzz的日期时间字符串，其中从左到右依次为：yyyy=Year，mm=Month，dd=Day，hh=Hour，mm=Minute，ss=Second，ffffff=Fractional Second，&=’+’/’-’，zzzz=Hours和Minutes的偏移量 | ≤26 | char* (to time_t) |
| FL | Floating Point Single，32位单精度浮点数 | 4 | float |
| FD | Floating Point Double，64位双精度浮点数 | 8 | double |
| IS | Integer String，以字符串表示的十进制整数， | ≤12 | char* (to int) |

| | | | |
|---|---|---|---|
| | 如123、-321等，其取值范围为[$-2^{31}$, $2^{31}$-1] | | |
| LO | Long String，长字符串 | ≤64 | char* |
| LT | Long Text，长文本数据 | ≤10240 | char* |
| OB | Other Byte String，编码方式由传输语法所规定的字节流 | 自定义 | char* |
| OF | Other Float String，32位浮点数组 | ≤$2^{32}$-4 | float* |
| OW | Other Word String，编码方式由传输语法所规定的双字节（16位）流 | 自定义 | short* |
| PN | Person Name，由5个部分组成的人名字符串，依次是：Family Name Complex、Given Name Complex、Middle Name、Name Prefix、Name Suffix，任一部分都可为空，每部分间用'^'隔开 | 每个部分≤64 | char* |
| SH | Short String，短字符串 | ≤16 | char* |
| SL | Singed Long，带符号32位整型数 | 4 | int (long) |
| SQ | Sequence of Items，包含0个或多个Item，用于存储嵌套的数据集 | N/A | N/A |
| SS | Signed Short，带符号16位整型数 | 2 | short |
| ST | Short Text，短文本数据 | ≤1024 | char* |
| TM | Time，格式为hhmmss.frac的时间字符串，hh范围为"00"-"23"，mm和ss范围均为"00"-"59"，frac范围为"000000"-"999999" | ≤16 | char* (to time_t) |
| UI | Unique Identifier，唯一标识符，数字字符串，中间以'.'分隔，如1.2.840.10008.1.1 | ≤64 | char* |
| UL | Unsigned Long，无符号32位整型数 | 4 | unsigned int |

| | | | |
|---|---|---|---|
| | | | (unsigned long) |
| UN | Unknown，未知类型 | 任意 | N/A |
| US | Unsigned Short，无符号16位整型数 | 2 | unsigned short |
| UT | Unlimited Text，无限制文本 | $\leqslant 2^{32}-2$ | char* |

（3） 数据长度（Value Length）

一个 16 位或 32 位整型数（根据 VR 种类和 VR 是显式还是隐式决定），指定数据域（Value Field）的长度（以字节为单位）。DICOM 标准规定数据域长度必须是偶数，不足时要用“填充字节”补齐。

若该字段编码为 FFFFFFFFH，则表示长度不定，适用于 VR 为 SQ 和 UN 的数据元素。对于 VR 为 OB 和 OW 的数据元素，在特定的传输语法下也可能将此字段编码为 FFFFFFFFH。

（4） 数据域（Value Field）

存放真正的数据，必须为偶数个字节，不足时由“填充字节”补齐。其数据类型由该数据元素的 VR 指出，可以存放多个值，由‘\’分隔。

根据不同的传输语法（在文件头中或通信协商阶段确定），数据元素的格式可为表 8-4、表 8-5 或表 8-6 所示的三者之一。

表 8-4 显式 VR 为 OB、OW、OF、SQ、UT 或 UN 时的数据元素格式

| 标签 | | 数据类型 | | 数据长度 | 数据域 |
|---|---|---|---|---|---|
| 组号（16位无符号整数） | 元素号（16位无符号整数） | VR（“OB”、“OW”、“OF”、“SQ”、“UT”或“UN”） | 2字节保留，置为0000H | 32位无符号整数 | 偶数字节的数据，其编码格式由VR和传输语法确定 |

| | | | | | |
|---|---|---|---|---|---|
| 2字节 | 2字节 | 2字节 | 2字节 | 4字节 | 由数据长度确定或不定 |

表 8-5 显式 VR 除 OB、OW、OF、SQ、UT 和 UN 以外的数据元素格式

| 标签 | | 数据类型 | 数据长度 | 数据域 |
|---|---|---|---|---|
| 组号（16位无符号整数） | 元素号（16位无符号整数） | VR，2字节字符串 | 16位无符号整数 | 偶数字节的数据，其编码格式由VR和传输语法确定 |
| 2字节 | 2字节 | 2字节 | 2字节 | 由数据长度确定 |

表 8-6 隐式 VR 数据元素格式

| 标签 | | 数据长度 | 数据域 |
|---|---|---|---|
| 组号（16位无符号整数） | 元素号（16位无符号整数） | 32位无符号整数 | 偶数字节的数据，其编码格式由VR和传输语法确定 |
| 2字节 | 2字节 | 4字节 | 由数据长度确定或不定 |

**2. 字节序**（Byte Ordering）

任何存储单位大于 1 字节的数据（比如 32 位整型数以 4 字节为一个存储单位）都存在字节排序的问题，即每一个数据单元从起始地址开始所有字节按从低到高排序还是从高到低排序，前者称为小端序（Little Endian），后者称为大端序（Big Endian）。比如一个 32 位整数 987654321，用 16 进制表示是 3ADE68B1，则在小端序的方式下存储为 B1 68 DE 3A（存储单元地址从左往右递增），在大端序下存储为 3A DE 68 B1。

不同体系结构的计算机可能采用不同的字节序来存储数据，比如 IBM PC 系列兼容计算机采用小端序，而 Apple 的 Power PC 系列则采用大端序。DICOM 标准同时支持两种字节序，具体采用哪种字节序由传输语法确定，缺省采用小端

序。

因此，当读取与本地计算机硬件要求的字节序不同的数据时，对于那些多字节的数据单元必须先交换前后字节（Byte Swapping）才能进行进一步的处理。

**3. 数据集的嵌套**（Nesting of Data Sets）

当数据元素的类型（VR）为“SQ”时，其数据域包含了一系列的项（Item），每一项又是一个数据集（Data Set），这就构成了数据集的嵌套。

与 SQ 有关的数据元素有：项元素（Item），其标签为（FFFE，E000）；项定界符（Item Delimitation），标签为（FFFE，E00D）；序列定界符（Sequence Delimitation），标签为（FFFE，E0DD）。这三个数据元素比较特殊，它们的编码格式并不受传输语法的限制，并且统一采用隐式 VR 的编码方式。但是，它们的数据域所嵌套包含的数据集则还是由传输语法规定其编码格式。

VR 为“SQ”的数据元素所嵌套的每一个数据集被封装进一个项元素中，由于采用隐式 VR 编码，其格式如表 8-6 所示。而根据数据长度字段的不同，项元素的编码方式有两种：

- 显式长度（Explicit Length）：项元素的数据长度字段包含数据域的长度（以字节为单位）；
- 未定义长度（Undefined Length）：项元素的数据长度字段编码为 FFFFFFFFH，因为未指明后面数据域的长度，所以必须紧跟一个标签为（FFFE，E00D）的项定界符标志该项的结束，项定界符不包含任何数据，其数据长度字段必须为 00000000H。

整个 VR 为“SQ”的数据元素编码根据数据长度字段的不同也有相应的两种编码方式：

- 显式长度（Explicit Length）：项元素的数据长度字段包含数据域的长度（以字节为单位），该长度为数据域中所有项的总长度；
- 未定义长度（Undefined Length）：项元素的数据长度字段编码为 FFFFFFFFH，因为未指明后面数据域的长度，所以必须紧跟一个标签为（FFFE，E0DD）的序列定界符标志该序列的结束，序列定界符不包含任何数据，其数据长度字段必须为 00000000H。

上述情况下的实例如表 8-7、表 8-8 和表 8-9 所示。

表 8-7 带三个显式长度项的隐式 VR 为“SQ”的数据元素例子

| 标签 | 数据长度 | 数据域 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| (gggg,eeee) VR 为 ”SQ” | 00000F00H | 第一项 | | | 第二项 | | | 第三项 | | |
| | | 项标签(FFFE,E000) | 项长度 00000 4F8H | 项值(数据集) | 项标签(FFFE,E000) | 项长度 00000 4F8H | 项值(数据集) | 项标签(FFFE,E000) | 项长度 00000 4F8H | 项值(数据集) |
| 4 字节 | 4 字节 | 4 字节 | 4 字节 | 04F8H 字节 | 4 字节 | 4 字节 | 04F8H 字节 | 4 字节 | 4 字节 | 04F8H 字节 |

表 8-8 带两个显式长度项的显式 VR 为“SQ”未定义长度的数据元素例子

| 标签 | 数据类型 | | 数据长度 | 数据域 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| (gggg,eeee) VR 为 ”SQ” | SQ | 0000H 保留 | FFFFFFFFH 未定义长度 | 第一项 | | | 第二项 | | | 序列定界符 | |
| | | | | (FFFE,E000) | 项长度 98A52C68H | 项值(数据集) | (FFFE,E000) | 项长度 B321762CH | 项值(数据集) | (FFFE,E0DD) | 00000000H |
| 4 字节 | 2 字节 | 2 字节 | 4 字节 | 4 字节 | 4 字节 | 98A52C68H 字节 | 4 字节 | 4 字节 | B321762CH 字节 | 4 字节 | 4 字节 |

表 8-4 带未定义长度项的隐式 VR 为“SQ”且未定义长度的数据元素例子

| 标签 | 数据长度 | 数据域 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| (gggg,eeee) VR 为 ”SQ” | FFFFFFFFH 未定义长度 | 第一项 | | | 第二项 | | | | | 序列定界符 | |
| | | (FFFE,E000) | 项长度 00000017B6H | 项值(数据集) | (FFFE,E000) | FFFFFFFFH 未指定长度 | 项值(数据集) | 项定界符 (FFFE,E00D) | 00000000H | (FFFE,E0DD) | 00000000H |
| 4 字节 | 4 字节 | 4 字节 | 4 字节 | 17B6 字节 | 4 字节 | 4 字节 | 未定义长度 | 4 字节 | 4 字节 | 4 字节 | 4 字节 |

**4. 像素数据的编码和压缩**

像素数据往往是 DICOM 数据流中数据量最大，同时也是最重要的数据。记录像素数据的数据元素标签为（7FE0，0010），其所含的像素数据由大量的“像素单元（Pixel Cell）”构成，每个像素单元的编码方式由下列三个数据元素的值所决定：

- 分配位数（Bits Allocated），标签为（0028，0100），表示每个像素单元占用多少位（Bits）；
- 存储位数（Bits Stored），标签为（0028，0101），表示在为每个像素单元分配的所有存储位中，有多少位存储了像素采样值（Pixel Sample Value）；
- 高位（High Bit），标签为（0028，0102），表示存储像素采样值的最高位在分配的存储位中在第几位（从 0 开始）。

例如，分配位数为 24，存储位数为 18，高位为 19，表示每个像素单元占用 24 位存储空间，其中的 18 位存储了实际的像素采样值，而这 18 位的最高位对应于 24 位的第 19 位，如 8-3 所示。除被像素采样值占用的存储位以外未使用的存储位可以另作它用。

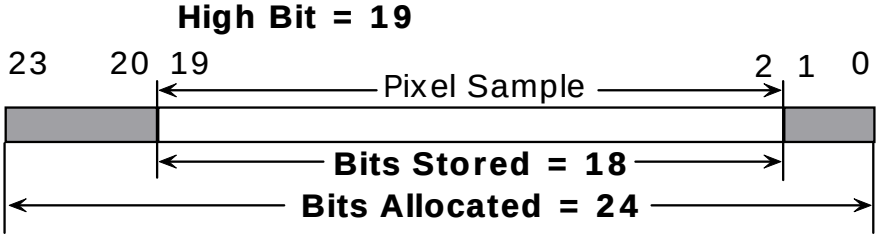



图 8-4 像素单元编码实例

像素数据可以采用自然格式（Native Format）或者封装格式（Encapsulated Format）进行存储或传输。

在自然格式下，所有像素单元不经任何形式的转换编码（如压缩等）而直接按序存储在（7FE0，0010）数据元素的数据域，应用程序通过给定的分配位数、存储位数及高位值对其进行解读，通常 VR 为“OW”，如果分配位数小于等于 8，也可采用“OB”。

在封装格式下，像素数据的所有像素单元采用 DICOM 标准以外的某种编码方式进行编码，编码方式由传输语法确定，通常是某种压缩格式，比如 JPEG 压缩格式。在此方式下，（7FE0，0010）数据元素的 VR 必须为“OB”，其数据域存储的是编码后的字节流，应用程序必须对此字节流用相同的编码方式进行解码，然后才能得到有意义的像素单元数据。

封装格式采用分段的方式存储编码后的字节流，每一段（Fragment）被编码进一个显式长度的项（Item）中，整个项的序列以一个基本偏移表项（Basic Offset Table Item）开始，对于多帧（Multi-frame）的图像，其中记录了每一帧的起始段（项）相对于第一项开头的位移量，以此来区分多帧图像的多个帧，而对于单帧或只有一帧的多帧图像，该偏移表的数据域可为空，数据长度字段相应编码为 00000000H。而且在封装格式下，（7FE0，0010）数据元素的数据长度必须是未定义的，因此整个项的序列以一个序列定界符作为结尾（参考表 8-8 的例子）。

此外，大多数的编码方式会将图像或像素格式的相关信息编码进最终的字节流中，比如图像的宽、高，上面讲到的每个像素单元的分配位数、存储位数等，这时候 DICOM 标准要求相关数据元素的值必须与编码进像素数据字节流的相关信息保持一致。

DICOM 标准接受 JPEG、JPEG 2000、JPEG-LS 及 RLE 压缩格式。

**5. 唯一标识符**（Unique Identifier，UID）

为了在网络环境下唯一地标识各种信息，DICOM 采用了 UID 的方式。UID 的定义基于 ISO8824 标准，并使用 ISO9834-3 中所注册的值来保证全局唯一性。一个 UID 唯一标识符可以用公式表示为：

UID=<org root>.<suffix>

其中 org root 代表组织编号（如制造商、研究单位等），而 suffix 部分则是在此组织范围内的唯一编号。这两部分均由一串点号隔开的数字组成，如<org root>=.2.840.10008 代表“美国电器制造商协会”。

**6. 传输语法**（Transfer Syntax）

前面已经多次提到“传输语法”，所谓传输语法就是一组编码规则，用于无歧义地解读传输中的或存储于存储设备上的 DICOM 数据。

不同的传输语法用一组 UID 来区分，常用的传输语法如表 8-10 所示。

表 8-10 常用传输语法

| UID值 | 说明 |
| --- | --- |
| 1.2.840.10008.1.2 | 隐式VR，小端序（缺省传输语法） |
| 1.2.840.10008.1.2.1 | 显式VR，小端序 |
| 1.2.840.10008.1.2.1.99 | 显式VR，小端序，“Deflate”压缩 |
| 1.2.840.10008.1.2.2 | 显式VR，大端序 |
| 1.2.840.10008.1.2.4.50 | JPEG Baseline压缩（有损JPEG 8-bit压缩的缺省传输语法） |

| | |
|---|---|
| 1.2.840.10008.1.2.4.51 | JPEG Extended压缩（有损JPEG 12-bit压缩的缺省传输语法） |
| 1.2.840.10008.1.2.4.57 | JPEG无损压缩（Non-Hierarchical） |
| 1.2.840.10008.1.2.4.70 | JPEG 无损压缩（Non-Hierarchical，First-Order Prediction，JPEG无损压缩的缺省传输语法) |
| 1.2.840.10008.1.2.4.80 | JPEG-LS无损压缩 |
| 1.2.840.10008.1.2.4.81 | JPEG-LS有损压缩 |
| 1.2.840.10008.1.2.4.90 | JPEG 2000压缩（仅有损方式） |
| 1.2.840.10008.1.2.4.91 | JPEG 2000压缩 |
| 1.2.840.10008.1.2.5 | RLE无损压缩 |

其中，所有压缩编码方式均采用显式 VR 和小端序编码。1.2.840.10008.1.2.4.51 ～ 1.2.840.10008.1.2.4.56 、 1.2.840.10008.1.2.4.58 ～ 1.2.840.10008.1.2.4.66 均为 JPEG 的各种压缩格式，已标记为 RETIRED，故不再详细说明

(2) DICOM 文件结构

典型的 DICOM 文件由两部分组成：

- 文件元信息（File Meta Information）：包含此文件所存储的数据集的识别信息，包括文件识别前缀、数据集标识及存储格式等信息；
- DICOM 数据集（DICOM Data Set）：文件所存储的 DICOM 数据，是文件的主题。

其中，文件元信息有可以分为以下一些部分：

- 文件前同步码（File Preamble）：长 128 字节，可以存放一些生成该 DICOM 文件的应用程序的某些指定信息，如不使用则全部置为 00H；

- DICOM 前缀（DICOM Prefix）：4 字节字符串，规定为“DICM”；
- 文件元数据元素（File Meta Elements）：一组组号为 0002H 的 DICOM 数据元素，包含文件数据集的标识、格式等信息，其中标签为（0002，0010）的数据元素即存储了传输语法 UID。这组数据元素均采用显式 VR 和小端序格式存储。

DICOM 文件结构如图 8-4 所示。

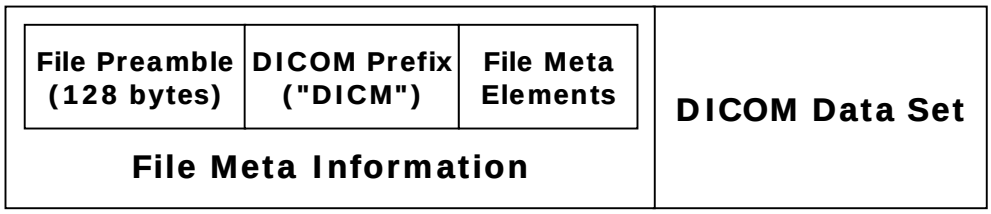



图 8-4 DICOM 文件结构

### 8.2.2 DICOM 文件读写模块（DICOM Utility）的实现

如前所述，MITK 中关于 DICOM 文件的读写功能是作为一个独立的 Utility 模块实现的，该模块的框架如图 8-5 所示。

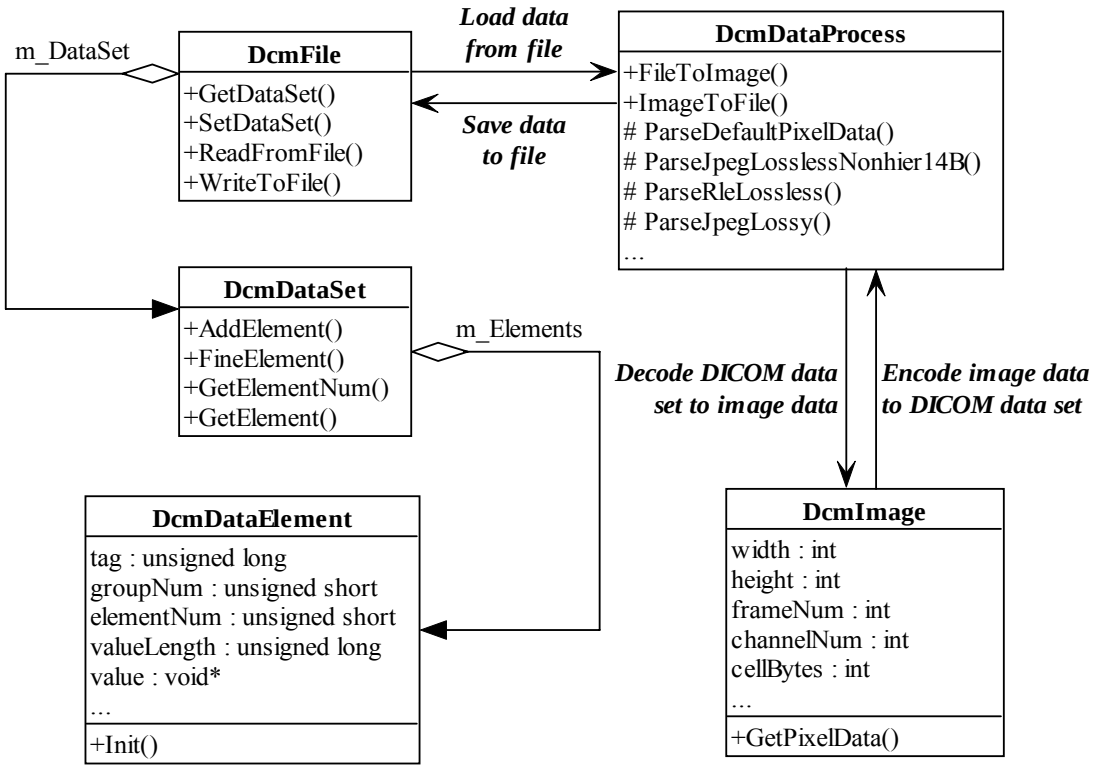



图 8-5 DICOM 文件读写模块框架

从图中可以看出，该模块的核心部分是 DcmDataProcess 类，其中包含了对 DICOM 数据流进行解码的全部过程。但是该类提供给外部使用的接口却相对简单而且直观：FileToImage()对从 DICOM 文件中读取的像素进行解码并产生出用户可用的图像数据，封装在 DcmImage 对象中；ImageToFile()则将用户给定的图像数据编码为 DICOM 数据流通过 DcmFile 类提供的接口写入 DICOM 文件中。即 DcmFile 只负责数据结构的组织和 DICOM 文件的读写操作，而具体的编码和解码工作则由 DcmDataProcess 完成。

下面分别介绍其中所用到的基本数据结构及编、解码方法。

(1) **基本数据结构**

DcmDataElement

```
class DcmDataElement
{
public:
    DcmDataElement();
    void Init(unsigned short group = 0,  unsigned short element = 0,
```

```
            unsigned short vr = VR_UN, unsigned long  vl = 0,
            void        *value = NULL, bool isSquence = false);
    ~DcmDataElement();

    unsigned long     tag;              // 标签
    unsigned short    groupNum;         // 组号
    unsigned short    elementNum;       // 元素号
    unsigned short    VR;               // 数据类型
    unsigned long     VL;               // 数据长度
    bool              isSequence;       // 是否为嵌套序列
    void              *value;           // 指向该元素数据的指针
};
```

可以看出，该结构完全对应于 DICOM 标准中对数据元素的编码格式。其中，tag 实际上就是 groupNum 和 elementNum 的组合，之所以重复存放，主要是为了方便解码。isSequence 标明了该数据元素是否包含了嵌套序列，如果值为 true，则 value 应理解为一个指向 DcmDataSet 的指针，而指向的这个 DcmDataSet 所包含的均为 Item 类型的数据元素，每个 Item 均包含一个子数据集，即其 value 均为指向一个 DcmDataSet 的指针（参考图 8-6）。

DcmDataSet

```
class DcmDataSet
{
private:
    // 一个 DcmDataElement 的列表
    vector<DcmDataElement *> m_Elements;

public:
    DcmDataSet();
    ~DcmDataSet();

    // 添加数据元素.
    void AddElement(DcmDataElement *element);

    // 在该数据集中寻找标签值为 tag 的数据元素.
    // 若找到，则返回 true，且 element 中为指向该数据元素的指针;
    // 否则返回 false.
    bool Find(const unsigned long tag, DcmDataElement * &element);
```

```
    // 该数据集是否为空.
    bool IsEmpty();

    // 返回该数据集中所包含数据元素的个数.
    int GetElementNum();

    // 取得该数据集中第 n 个数据元素.
    DcmDataElement* GetElement(int n);
};
```

该结构完全对应于 DICOM 标准中的数据集定义。它维护了一个数据元素的列表，并提供一些接口对其中的数据元素进行方便的存取。

对于嵌套结构的数据集，其逻辑结构如图 8-6 所示。

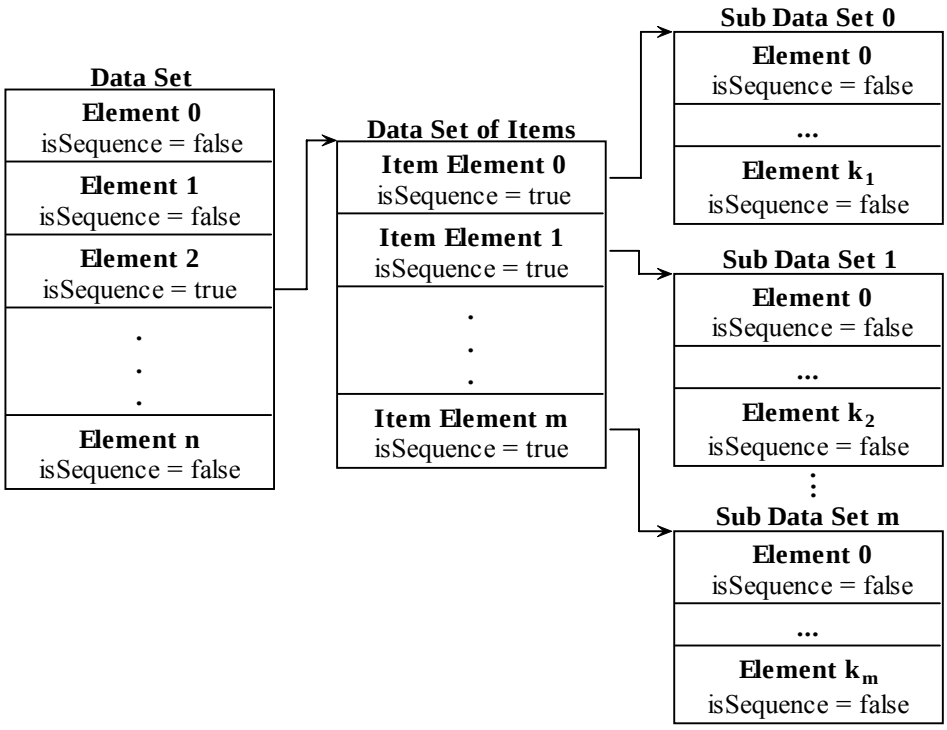



图 8-6 嵌套数据集的结构示意图

DcmImage

```
class DcmImage
{
public:
```

```
DcmImage()
{
    width = height = 0;
    frameNum = channelNum = cellBytes =1;
    spacingX = spacingY = thickness = 1.0f;
    location = windowCenter = windowWidth = 0.0f;
    isUnsigned = isColorByPxl = true;
    pixelData = NULL;
}

~DcmImage()
{
    delete []pixelData;
}

void Clear()
{
    width = height = 0;
    frameNum = channelNum = cellBytes =1;
    spacingX = spacingY = 1.0f;
    location = windowCenter = windowWidth = 0.0f;
    isUnsigned = isColorByPxl = true;
    delete []pixelData;
    pixelData = NULL;
}

// 取得指向像素数据的指针.
// 注意：调用此函数后，像素数据所占内存由用户释放.
void* GetPixelData()
{
    void *data = pixelData;
    pixelData = NULL;
    return data;
}

void SetPixelData(void *data) { pixelData = data; }

bool IsPixelDataOk() { return (pixelData!=NULL); }

int   frameNum;
```

```
    int   width;
    int   height;
    int   channelNum;
    int   cellBytes;
    float spacingX;
    float spacingY;
    float thickness;
    float location;
    float windowCenter;
    float windowWidth;
    bool  isUnsigned;
    bool  isColorByPxl;

private:
    void *pixelData;
};
```

这是一个辅助类，用于存放解码后的图像数据，包括与图像相关的信息和解码后的像素数据（Pixel Data）。为尽量避免对像素数据所占内存区的操作失误，对指向像素数据的指针采取了适当的保护，通过提供 GetPixelData()、SetPixelData()以及 IsPixelDataOk()来提供对像素数据的存取以及数据有效性的判别。

(2) DICOM 文件的读写——DcmFile

DcmFile 类封装了对 DICOM 文件的读写操作。该类包含一个 DcmDataSet，通过 ReadFromFile()将 DICOM 文件中的数据元素提取出来，生成一个个 DcmDataElement 对象，加入到 DcmDataSet 对象中，组织成如图 8-6 所示的树状逻辑结构，而 DcmFile 本身并不对解读数据集的含意做任何尝试，这是后面要讲到的 DcmDataProcess 类所完成的工作。

DcmFile 首先解读 DICOM 文件头信息，即本章第二节中所提到的 File Meta Information，得到存储该文件所使用的传输语法，从而取得该文件所包含数据集的编码格式，为读取 DICOM 数据集做准备。

DICOM 数据集的嵌套结构给文件的读取带来了一定的困难，这里我们采用递归的方法来读取文件所包含的 DICOM 数据集并构造出树状结构的

DcmDataSet 对象，该方法通过下面 5 个子过程的相互协作而实现：

```
int ReadELDataSet(DcmDataSet *dataSet, unsigned long dataLength);
int ReadULDataSet(DcmDataSet *dataSet, unsigned long delimitTag);
int ReadELSQ(DcmDataSet *dataSet, unsigned long dataLength);
int ReadULSQ(DcmDataSet *dataSet, unsigned long delimitTag);
int ReadOBSQ(DcmDataSet *dataSet, unsigned long delimitTag);
```

其中，ReadELDataSet() 用于读取显式长度的 DICOM 数据集；ReadULDataSet()用于读取未定义长度的 DICOM 数据集；ReadELSQ()用于读取显式长度的 Item 序列；ReadULSQ()用于读取未定义长度的 Item 序列；ReadOBSQ 专门用于读取封装格式下像素数据的 Item 序列。这些子过程在将文件中的数据元素读入内存时统一将其格式转换为小端序及显式 VR，对于隐式 VR 的数据，通过检索 DICOM 数据字典得到其 VR 并填入 DcmDataElement 对象的 VR 域中。其伪代码如下所示：

```
int ReadELDataSet(DcmDataSet *dataSet, unsigned long dataLength)
  if dataLength == 0 return 0;
  DcmDataElement *element;
  DcmDataSet *subDataSet;
  int count = 0;
  do
      element = new DcmDataElement;
      从文件中读取当前数据元素的相关信息填入 element(tag、VR、VL 等，不包括数据域);
      count += 读入的字节数;
      dataSet->AddElement(element);
      if element->VL==0xFFFFFFFF
          subDataSet = new DcmDataSet;
          if element->VR=="SQ" or element->VR=="UN"
              count += ReadULSQ(subDataSet, 0xFFFEE0DD);
          else if element->VR=="OB"
              count += ReadOBSQ(subDataSet, 0xFFFEE0DD);
          end if
          element->isSequence = true;
          element->value = subDataSet;
      else if element->VR=="SQ"
          subDataSet = new DcmDataSet;
          count += ReadELSQ(subDataSet, element->VL);
          element->isSequence = true;
```

```
            element->value = subDataSet;
        else
            从文件读出 element->VL 字节的内容写入 element->value 域
            (必要时做 Byte Swapping);
            count += 读入的字节数;
        end if
    while count<dataLength;
    return count.


int ReadULDataSet(DcmDataSet *dataSet, unsigned long delimitTag)
    DcmDataElement *element;
    DcmDataSet *subDataSet;
    int count = 0;
    do
        element = new DcmDataElement;
        从文件中读取当前数据元素的相关信息填入 element(tag、VR、VL 等，不包括数据
        域);
        count += 读入的字节数;
        if element->tag==delimitTag break; //出口
        dataSet->AddElement(element);
        if element->VL==0xFFFFFFFF
            subDataSet = new DcmDataSet;
            if element->VR==”SQ” or element->VR==”UN”
                count += ReadULSQ(subDataSet, 0xFFFEE0DD);
            else if element->VR==”OB”
                count += ReadOBSQ(subDataSet, 0xFFFEE0DD);
            end if
            element->isSequence = true;
            element->value = subDataSet;
        else if element->VR==”SQ”
            subDataSet = new DcmDataSet;
            count += ReadELSQ(subDataSet, element->VL);
            element->isSequence = true;
            element->value = subDataSet;
        else
            从文件读出 element->VL 字节的内容写入 element->value 域
            (必要时做 Byte Swapping);
        end if
    while 文件未结束;
```

```
  return count.


int ReadELSQ(DcmDataSet *dataSet, unsigned long dataLength)
  if dataLength == 0 return 0;
  DcmDataElement *item;
  DcmDataSet *subDataSet;
  int count = 0;
  do
      item = new DcmDataElement;
      从文件中读取当前数据元素的相关信息填入 item(tag、VR、VL 等，不包括数据
      域);
      count += 读入的字节数;
      dataSet->AddElement(item);
      if item->VL==0xFFFFFFFF
          subDataSet = new DcmDataSet;
          count += ReadULDataSet(subDataSet, 0xFFFEE00D);
      else
          subDataSet = new DcmDataSet;
          count += ReadELDataSet(subDataSet, item->VL);
      end if
      item->isSequence = true;
      item->value = subDataSet;
  while count<dataLength;
  return count.


int ReadULSQ(DcmDataSet *dataSet, unsigned long delimitTag)
  DcmDataElement *item;
  DcmDataSet *subDataSet;
  int count = 0;
  do
      item = new DcmDataElement;
      从文件中读取当前数据元素的相关信息填入 item(tag、VR、VL 等，不包括数据
      域);
      count += 读入的字节数;
      if item->tag==delimitTag break; //出口
      dataSet->AddElement(item);
      if item->VL==0xFFFFFFFF
          subDataSet = new DcmDataSet;
```

```
            count += ReadULDataSet(subDataSet, 0xFFFEE00D);
        else
            subDataSet = new DcmDataSet;
            count += ReadELDataSet(subDataSet, item->VL);
        end if
        item->isSequence = true;
        item->value = subDataSet;
    while 文件未结束;
    return count.


int ReadOBSQ(DcmDataSet *dataSet, unsigned long delimitTag)
    DcmDataElement *item;
    DcmDataSet *subDataSet;
    int count = 0;
    do
        item = new DcmDataElement;
        从文件中读取当前数据元素的相关信息填入 item(tag、VR、VL 等，不包括数据
        域);
        count += 读入的字节数;
        if item->tag==delimitTag break; //出口
        dataSet->AddElement(item);
        if item->VL!=0
            从文件读取 item->VL 个字节的数据放入 item->value 域;
            count += item->VL;
        end if
    while 文件未结束;
    return count.
```

相对于将 DICOM 数据集从文件中读入 DcmDataSet 对象来说，将 DcmDataSet 对象中树状结构的数据写入指定的 DICOM 文件要简单一些，直接将 DcmDataSet 中数据按传输语法所规定的格式写入文件，对于嵌套的数据结构也采用递归的方式处理。这一功能由 WriteToFile()接口实现。在 WriteToFile()中首先调用 WriteMetaInfo()将 File Meta Information 写入文件头部，然后调用 WriteDataSet()函数将 DcmDataSet 对象中的数据写入文件。WriteDataSet()是一个递归函数，用伪代码描述如下：

```
void WriteDataSet(DcmDataSet *dataSet, FILE *fp)
```

```
DcmDataElement *element;
int i;
for i=0 to dataSet->GetElementNum()-1 do
    element = dataSet->GetElement(i);
    将 element->tag、element->VR、element->VL 按传输语法要求的格式写入文
    件 fp(若 element->VL 为奇数,则相应写入数值应为 element->VL+1);
    if element->isSequence
        WriteDataSet((DcmDataSet *)element->value, fp);  //递归调用
        if element->VL==0xFFFFFFFF  //未定义长度,需补上项或序列的结束标志
            if element->tag==0xFFFEE000  //项结束
                将一个 tag 为 0xFFFEE00D,VL 为 0 的项定界符写入文件 fp;
            else  //序列结束
                将一个 tag 为 0xFFFEE0DD,VL 为 0 的序列定界符写入文件 fp;
            end if
        end if
    else  //一般数据元素
        将 element->value 中的 element->VL 个字节的数据写入文件 fp
        (若 element->VL 为奇数，则在末尾加上一个填充字节);
    end if
end for.
```

(3) DICOM 数据流的编码和解码——DcmDataProcess

DcmDataProcess 类封装了 DICOM 数据流的编码和解码功能。

解码时，它直接从一个 DcmFile 对象得到其中包含的 DcmDataSet 对象，其中包含了与实际 DICOM 文件完全对应的但经过结构化之后的数据，解码程序用 DICOM 数据元素的标签作为关键字，可以很方便地从 DcmDataSet 对象中取得对应的数据元素。对于普通的数据元素，直接可从其 value 域得到所需数据，而对于封装结构的像素数据，必须提供相应的解码过程，比如对于采用 RLE 无损压缩的像素数据，DcmDataProcess 类提供了一个 ParseRleLossless()函数对其进行解码。解码后所得的图像数据写入一个 DcmImage 对象提供给用户。

编码时，它从用户得到一个待编码的 DcmImage，根据用户指定的传输语法构造一个 DcmDataSet 对象，然后交给 DcmFile 将其写入文件。

### 8.2.3 DICOM Utility 在 MITK 中的封装

在 MITK 中，通过 mitkDICOMReader 和 mitkDICOMWriter 两个类封装了 DICOM Utility 的功能，提供给用户使用。

mitkDICOMReader 是 mitkVolumeReader 的一种，它负责读取一系列的 DICOM文件中所包含的图像数据，将其构造成一个mitkVolume（参见 2.2.2）提供给后续处理的算法作为输入数据。为了提供一个比较规整的Volume数据，mitkDICOMReader对于用DcmDataProcess读入的一系列切片还要作一些处理，主要是按切片的位置排序，切片的位置信息在DICOM文件中由SLICE LOCATION 数据元素提供。如果切片间距不一致，则以最小间距为标准，通过插值补上间距比较大的切片之间所缺的切片，以利于后续算法的处理。作为一个读取单一切片图像的例子，下面的代码演示了在mitkDICOMReader中如何使用 DcmDataProcess提供的功能读取DICOM文件中的图像数据：

```
int mitkDICOMReader::_readSingleFile(VOLUMEINFO &info)
{
    DcmDataProcess dcmp;  //DcmDataProcess 对象
    DcmImage image;       //用于存放解码后的图像数据

    // 调用 DcmDataProcess 类的 FileToImage()接口读取 DICOM 文件中的图像数据，
    // 并对出现的错误进行相应的处理
    switch (dcmp.FileToImage(image,this->_getFileName(0)))
    {
    case DcmDataProcess::RT_ERROR:
        mitkErrorMessage(this->_getFileName(0) << ": " <<
                         dcmp.GetErrorMessage());
        return -1;

    case DcmDataProcess::RT_WARNING:
        mitkWarningMessage(this->_getFileName(0) << ": " <<
                           dcmp.GetWarningMessage());
        break;
    }

    // 将图像信息传给 Volume
    info.width = image.width;
    info.height = image.height;
```

```
    info.channelNum = image.channelNum;
    info.cellBytes = image.cellBytes;
    info.isUnsigned = image.isUnsigned;
    info.spacingX = image.spacingX;
    info.spacingY = image.spacingY;
    info.spacingZ = image.thickness;
    info.windowCenter = image.windowCenter;
    info.windowWidth = image.windowWidth;

    // 处理像素数据
    void *data = image.GetPixelData();
    char *bufPtr = (char *)data;
    unsigned long imageBytes = image.width * image.height
                             * image.channelNum * image.cellBytes;

    // 处理多帧的情况
    if (image.frameNum > 1) m_IsMultiFrame = true;
    for (int i=0; i<image.frameNum; i++)
    {
        SLICE *slice = new SLICE;
        slice->isColorByPxl = image.isColorByPxl;
        slice->location = i;
        slice->value = bufPtr;
        bufPtr += imageBytes;
        m_Slices->push_back(slice);
    }

    info.imageNum = image.frameNum;

    // 调整切片顺序和间距
    if (this->_adjustSlices(info.width,info.height,info.channelNum,
                            info.cellBytes,info.isUnsigned,false) != 0)
    {
        return -1;
    }

    return 0;
}
```

mitkDICOMWriter 是 mitkVolumeWriter 的一种，负责将一个 mitkVolume 对

象中的所有切片的图像数据分别存储到一个个 DICOM 文件中。该类的处理要比 mitkDICOMReader 简单的多，只需循环地调用 DcmDataProcess 的 ImageToFile() 就可以输出所有切片图像到文件中。目前 MITK 中的 mitkDICOMWriter 采用显式 VR、小端序的缺省 DICOM 数据格式生成 DICOM 文件，即传输语法 UID 为 1.2.840.10008.1.2.1。下面的代码演示了在 mitkDICOMWriter 中如何使用 DcmDataProcess 提供的接口将 Volume 中的切片图像存储到 DICOM 文件中：

```
bool mitkDICOMWriter::Execute()
{
    // 要写的文件个数
    int fileNum = this->_getFileCount();
    if (fileNum <= 0)
    {
        mitkErrorMessage("Please input file names first.");
        return false;
    }

    // 要写入 DICOM 文件的 Volume
    mitkVolume *InputVolume = this->GetInput();

    // Volume 中包含的切片张数
    int imageNum = InputVolume->GetImageNum();

    if (fileNum > imageNum)
    {
        mitkWarningMessage("File names are more than images, the last "
                           << fileNum - imageNum
                            << " file names will be ignored.");
    }
    else if (fileNum < imageNum)
    {
        mitkErrorMessage("Images are more than file names, cannot save
                            all images.");
        return false;
    }

    // 填充相关图像信息
    DcmImage image;
```

```
image.width       = InputVolume->GetWidth();
image.height      = InputVolume->GetHeight();
image.channelNum  = InputVolume->GetNumberOfChannel();
image.cellBytes   = InputVolume->GetDataTypeSize();
image.spacingX    = InputVolume->GetSpacingX();
image.spacingY    = InputVolume->GetSpacingY();
image.windowCenter = InputVolume->GetWindowCenter();
image.windowWidth  = InputVolume->GetWindowWidth();

switch (InputVolume->GetDataType())
{
case MITK_CHAR:
case MITK_SHORT:
case MITK_INT:
case MITK_LONG:
    image.isUnsigned = false;
    break;

default:
    image.isUnsigned = true;
}

// 设定文件头信息
DCMFILEMETAINFO metaInfo;
metaInfo.mediaStorageSOPClassUID = "";
metaInfo.mediaStorageSOPInstanceUID = "";
metaInfo.transferSyntaxUID = UID_EXPLICIT_VR_LITTLE_ENDIAN;
metaInfo.implementationClassUID = "";
metaInfo.implementationVersionName = "";
metaInfo.sourceApplicationEntityTitle = "";

DcmDataProcess dcmp; //DcmDataProcess 对象

void *dataPtr = InputVolume->GetData();  //指向像素数据的指针
unsigned long imageBytes = image.width * image.height
                         * image.channelNum * image.cellBytes;

string str;
str.assign(m_StudyUID);
dcmp.SetStudyUID(str);
```

```
    str.assign(m_SeriesUID);
    dcmp.SetSeriesUID(str);

    for (int fileIdx=0; fileIdx<imageNum; fileIdx++)
    {
        image.location    = InputVolume->GetSpacingZ() * fileIdx;
        image.SetPixelData(dataPtr);

        // 通过调用 DcmDataProcess 类的 ImageToFile 接口将图像数据写入文件，
        // 并对出现的错误作相应处理
        switch (dcmp.ImageToFile(metaInfo,image,
                                 this->_getFileName(fileIdx),
                                 true,fileIdx))
        {
        case DcmDataProcess::RT_ERROR:
            mitkErrorMessage(dcmp.GetErrorMessage());
            return false;

        case DcmDataProcess::RT_WARNING:
            mitkWarningMessage(dcmp.GetWarningMessage());
            continue;
        }

        dataPtr = (char *)dataPtr + imageBytes;
    }

    return true;
}
```

## 8.3 小结

本章介绍了 DICOM 标准在 MITK 中的实现。

DICOM 标准本身所包含的内容庞大，涵盖面十分广泛，但 MITK 作为专注于医学影像处理与分析的算法开发平台，不可能也没有必要完全实现 DICOM 标准，所以我们只是将精力集中在对 DICOM 文件读写功能的实现上，通过一个相对独立的 DICOM Utility 来提供 MITK 所需的 DICOM 文件读写功能。

DICOM Utility 严格按照 DICOM 标准中关于数据结构和编码规范的规定进行设计，它的基础结构，包括 DcmFile、DcmDataSet 和 DcmDataElement，其适应性是相当好的，基本上能兼容 DICOM 标准中的所有不同的编码方式，这为以后的扩展带来了很大的方便。可以预见，随着 MITK 的不断发展，DICOM Utility 的功能也将不断得到完善和增强，从而为 MITK 提供强有力的 DICOM 数据解析支持。

## 参考文献

1. National Electrical Manufacturers Association. Digital Imaging and Communications in Medicine (DICOM)，2003
2. 刘景春，田捷，常红星，曹勇，邱峰．PACS 的结构与实现．中国医学影像技术，2000，第 16 卷第 1 期
3. 田捷，包尚联，周明全. 医学影像处理与分析. 北京：电子工业出版社，2003.
4. 邱峰，田捷，曹勇等．PACS 系统综述．中国医学影像技术，2000，第 16 卷第 1 期
5. Bidgood W.D., Horii S. Introduction to the ACR-NEMA DICOM Standard. RadioGraphics，1992，12(2):345-355

# 9 应用 MITK 开发实际项目

在介绍了 MITK 的设计与具体实现之后，本章将以几个实例来说明如何应用 MITK 来开发实际的项目。受篇幅所限，不宜介绍过于复杂的大型项目的实现细节，只能以几个短小精悍的例子来说明 MITK 的使用规范以及在使用过程中需要注意的一些问题。读者可以将其作为使用 MITK 开发实际项目的入门教程。

本章将以目前使用比较广泛的VC++6.0 作为集成开发环境。MITK开发包可以在 http://www.3dmed.net/mitk的Download页面下载，下载的压缩包mitk.zip解压后得到一个MITK目录，其中包含 4 个子目录，其中Include子目录下的所有头文件和Lib子目录下的Mitk_dll.lib文件是编译使用MITK的应用程序时所需要的，Lib子目录下的Mitk_dll.dll是编译生成的应用程序在执行时所要用到的动态链接库，请保证你的应用程序在运行时能找到这个dll文件，通常的做法是将其拷贝到应用程序相同的目录下，更省事的做法是将其拷贝到\%Windir%\System32\目录下，在下面的章节中我们假设你已经将Mitk_dll.dll放置到合适的位置而不再做额外的说明。

## 9.1 开发环境的设置

首先，用VC++6.0 新建一个MFC工程，我们将其命名为MITKTest，如 图 9-1 所示。选择Single document，然后Finish，这样就建立了一个单文档结构的MFC工程。

接下来要对这个工程做一些额外的设置，以使其能使用 MITK 类库。

选择Project→Settings菜单，在弹出的对话框中选择“C/C++”属性页，在“Category”中选择“Preprocessor”项，在“Additional include directories”下面的编辑框中输入你放置MITK头文件的位置，我们将工程直接建立在mitk.zip解压后得到的MITK\Examples目录下，因此这里我们输入“../../Include”，如 图 9-2 所示。

仍然是“Project settings”对话框，选择“Link”属性页，在“Category”中选择“Input”项，在“Additional library path”下面的编辑框中输入你放置Mitk_dll.lib的位置，我们在这儿填入“../../Lib”，在“Object/library modules”下

面的编辑框中输入“Mitk_dll.lib”，如 图 9-3 所示。

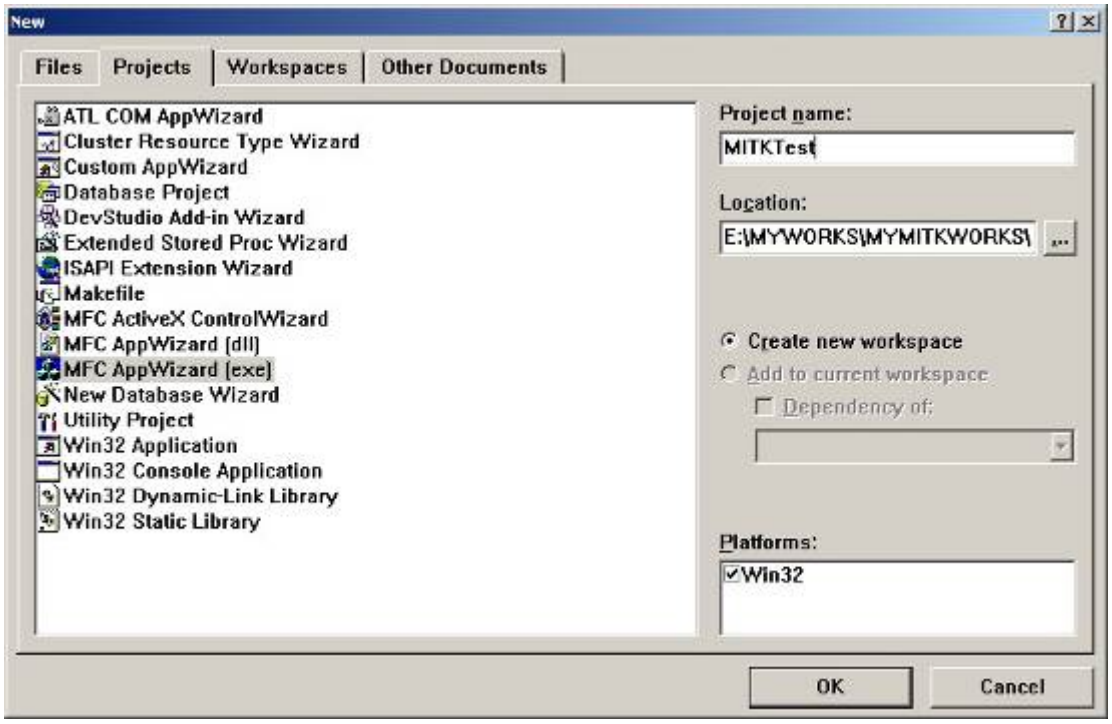



图 9-1 新建 MFC 工程

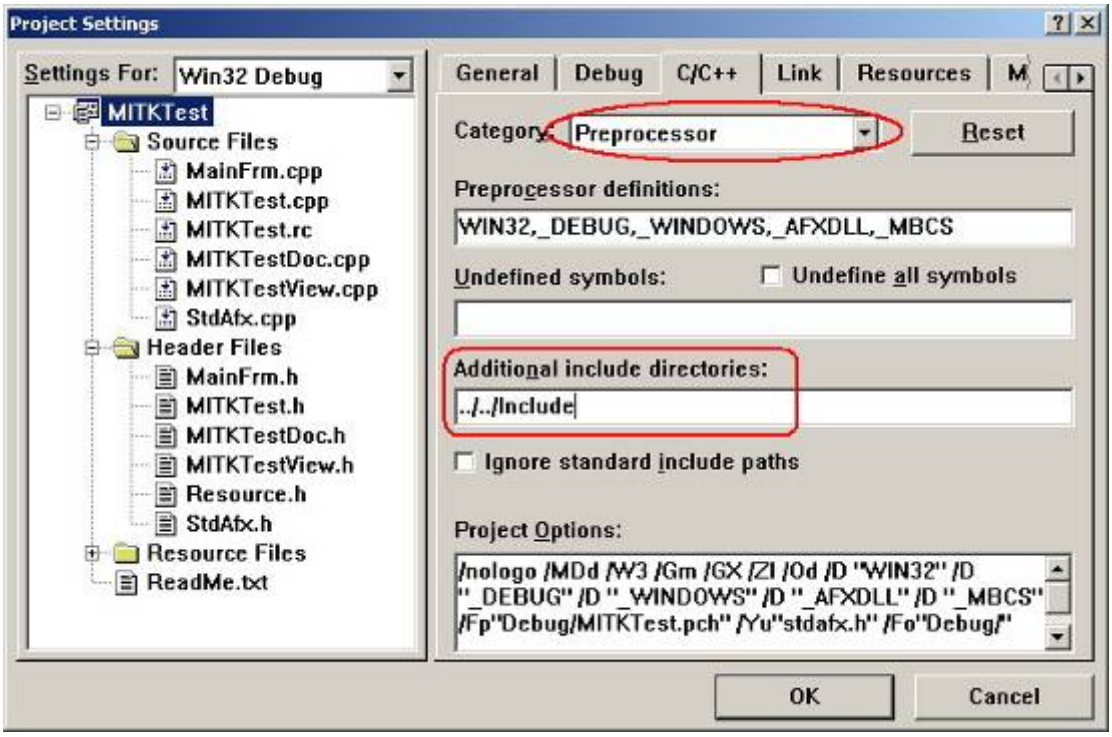



图 9-2 设置 MITK 头文件路径

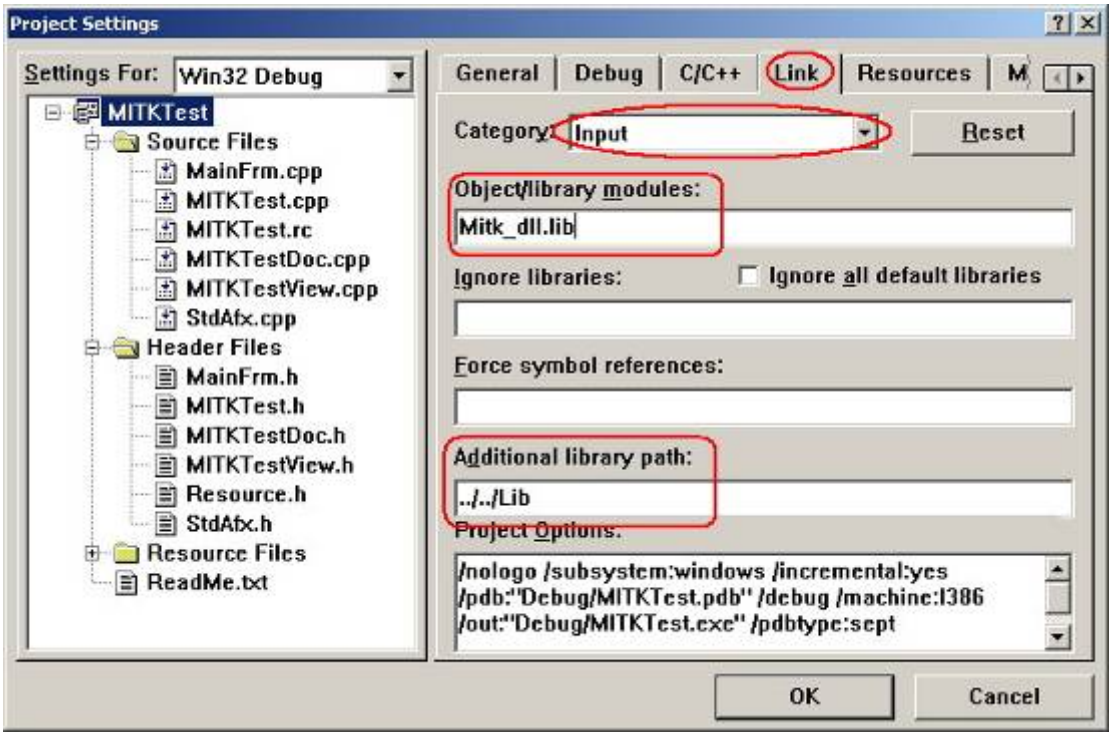



图 9-3 设置 MITK 库文件路径

以上设置请读者根据自己的实际情况自行调整，前提是让 VC 在编译时能够找到 MITK 的头文件及库文件。

接下来就要写一些代码，建立 MITK 的显示环境，这在下面各节的例子中都是要用到的。

首先，在 MITKTestView.h 中给 CMITKTestView 类添加一个 mitkView *类型的成员变量提供显示环境：

```
class mitkView;       //mitkView 类的前向声明
class CMITKTestView : public CView
{
......
protected:
    mitkView *m_View;     //添加一个指向 mitkView 的指针
......
};
```

接着，在 ClassWidzard 里面给 CMITKTestView 类添加 WM_CREATE、WM_SIZE和WM_DESTROY的消息处理函数，如图 9-4 和图 9-5 和图 9-6 所示：

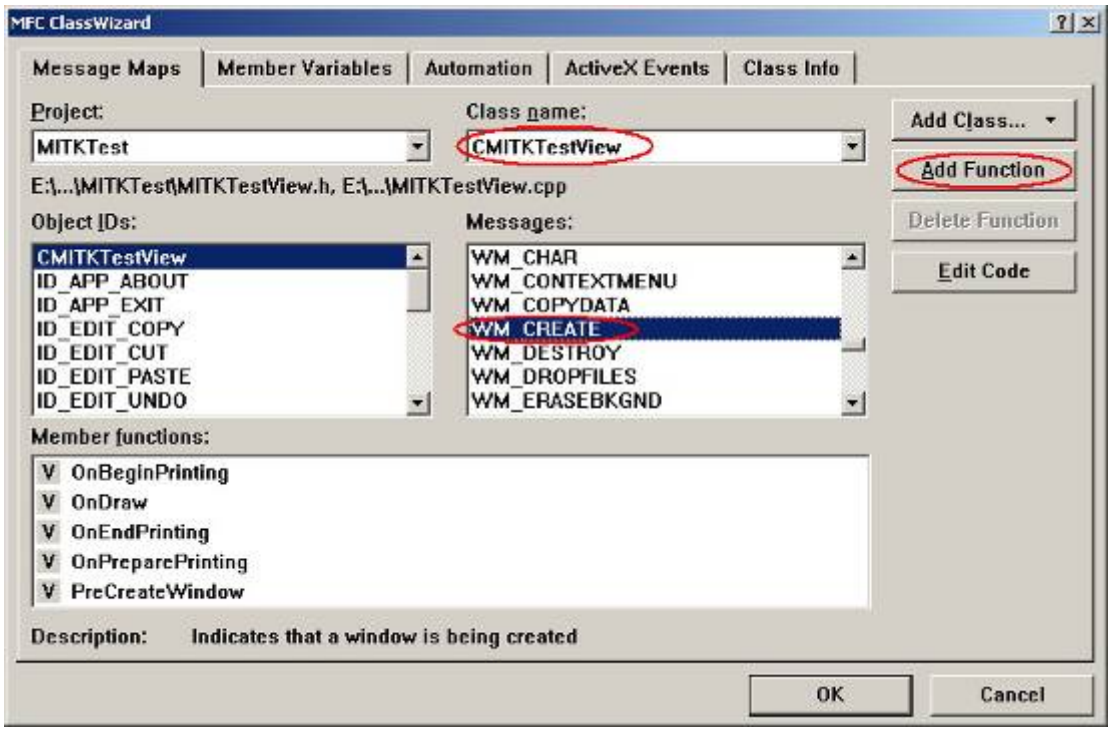



图 9-4 添加 WM_CREATE 的消息处理函数

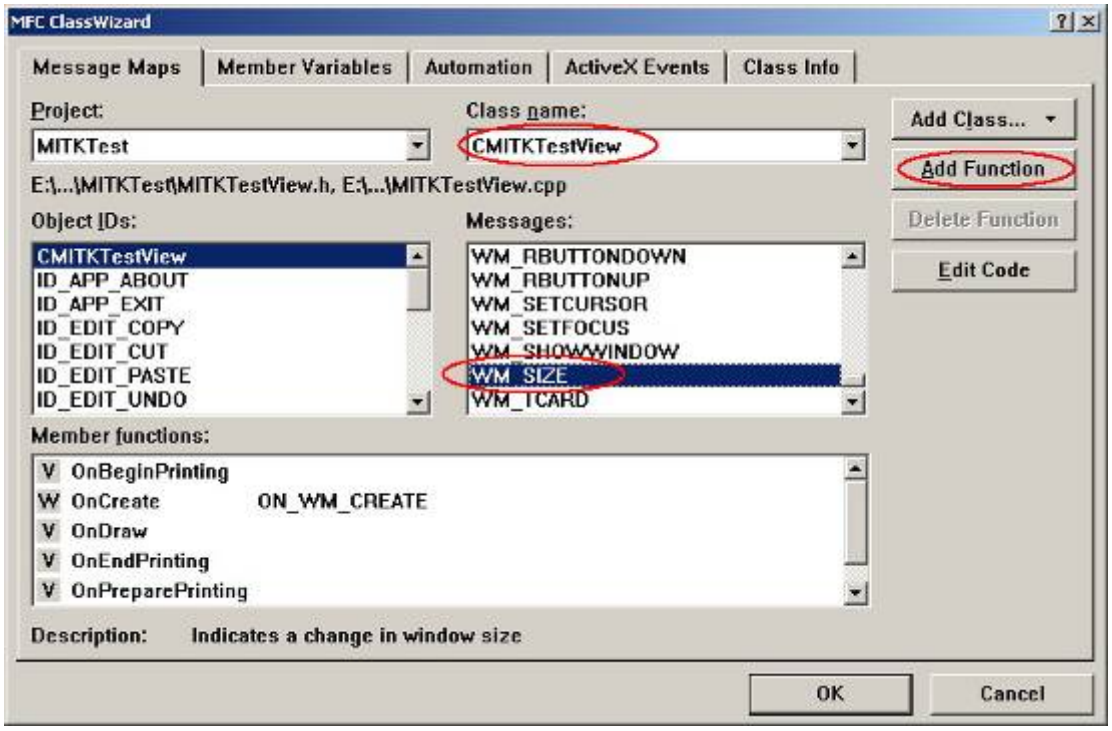



图 9-5 添加 WM_SIZE 的消息处理函数

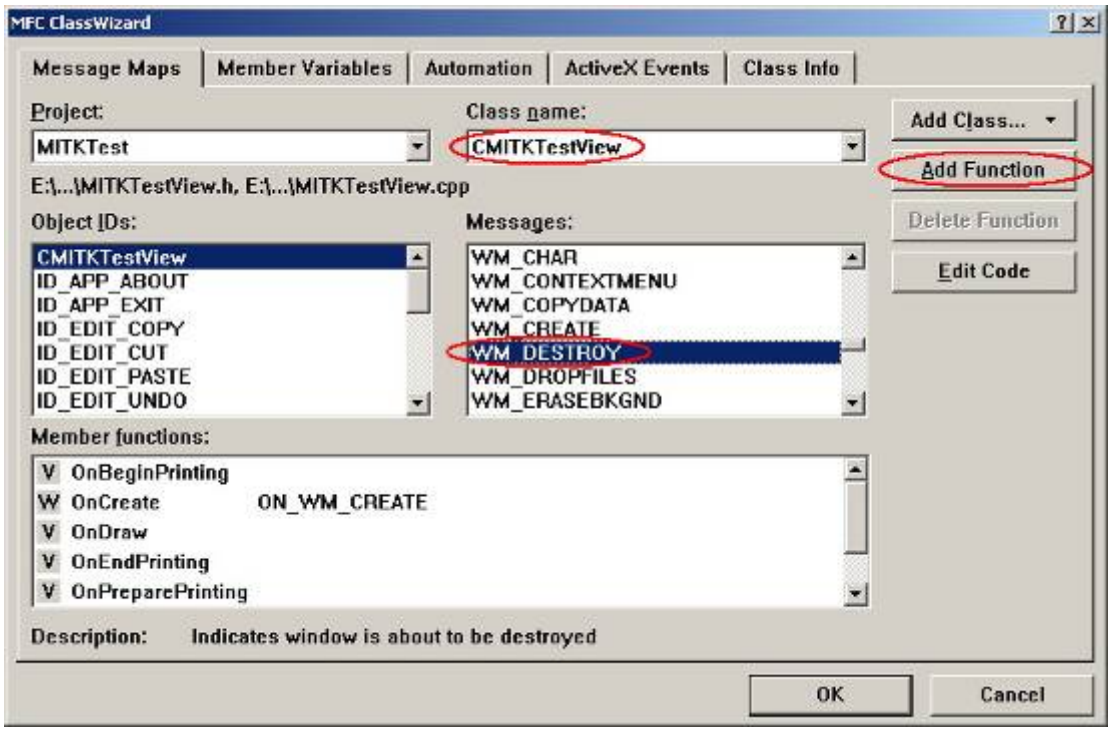



图 9-6 添加 WM_DESTROY 的消息处理函数

在 MITKTestView.cpp 开头添加#include “mitkView.h”：

```
#include "stdafx.h"
#include "MITKTest.h"

#include "MITKTestDoc.h"
#include "MITKTestView.h"

#include "mitkView.h"   //添加 mitkView 的头文件

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

......
```

在 OnCreate 函数中添加代码生成并初始化一个 mitkView：

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

    // TODO: Add your specialized creation code here
    // 得到当前客户区大小
    RECT clientRect;
    this->GetClientRect(&clientRect);
    int wWidth = clientRect.right - clientRect.left;
    int wHeight = clientRect.bottom - clientRect.top;

    // 产生 mitkView 对象
    m_View = new mitkView;

    // 设置父窗口句柄
    m_View->SetParent(GetSafeHwnd());

    // 设置 mitkView 在父窗口中显示的位置和大小
    m_View->SetLeft(0);
    m_View->SetTop(0);
    m_View->SetWidth(wWidth);
    m_View->SetHeight(wHeight);

    // 设置 mitkView 的背景颜色（这里将其设置为黑色）
    m_View->SetBackColor(0, 0, 0);

    // 显示 mitkView
    m_View->Show();

    return 0;
}
```

在 OnSize 函数中添加代码以便在父窗口改变大小时，mitkView 也能相应地调整自身大小：

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

    // TODO: Add your message handler code here
```

```
    // 防止对 NULL 指针操作
    if (!m_View) return;

    // 更新 mitkView 在父窗口中的位置和尺寸
    m_View->SetLeft(0);
    m_View->SetTop(0);
    m_View->SetWidth(cx);
    m_View->SetHeight(cy);

    // 刷新 mitkView
    m_View->Update();
}
```

在 OnDestroy()函数中添加代码以便在父窗口销毁时即时删除包含的mitkView。注意：不要将这步工作放到 CMITKTestView 的析构函数中去做，虽然在这里这么做不会出现任何问题，但是因为 MITK 允许在同一个父窗口中容纳多个 mitkView，当 mitkView 的个数多于一个时，在析构函数删除这些 mitkView 会导致程序异常，正确的做法时在父窗口响应 WM_DESTROY 消息时就即时删除这些 mitkView。代码如下：

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

    // TODO: Add your message handler code here
    if (m_View)
    {
        m_View->Delete();
        m_View = NULL;
    }
}
```

最后别忘了在 CMITKTestView 类的构造函数里初始化 m_View 指针为 NULL 以防出现野指针：

```
CMITKTestView::CMITKTestView()
{
    // TODO: add construction code here
    m_View = NULL;  //初始化指针变量
}
```

开发环境的设置到此基本完成，编译一下整个工程，如果没有错误的话程序运行结果是下面这个样子：

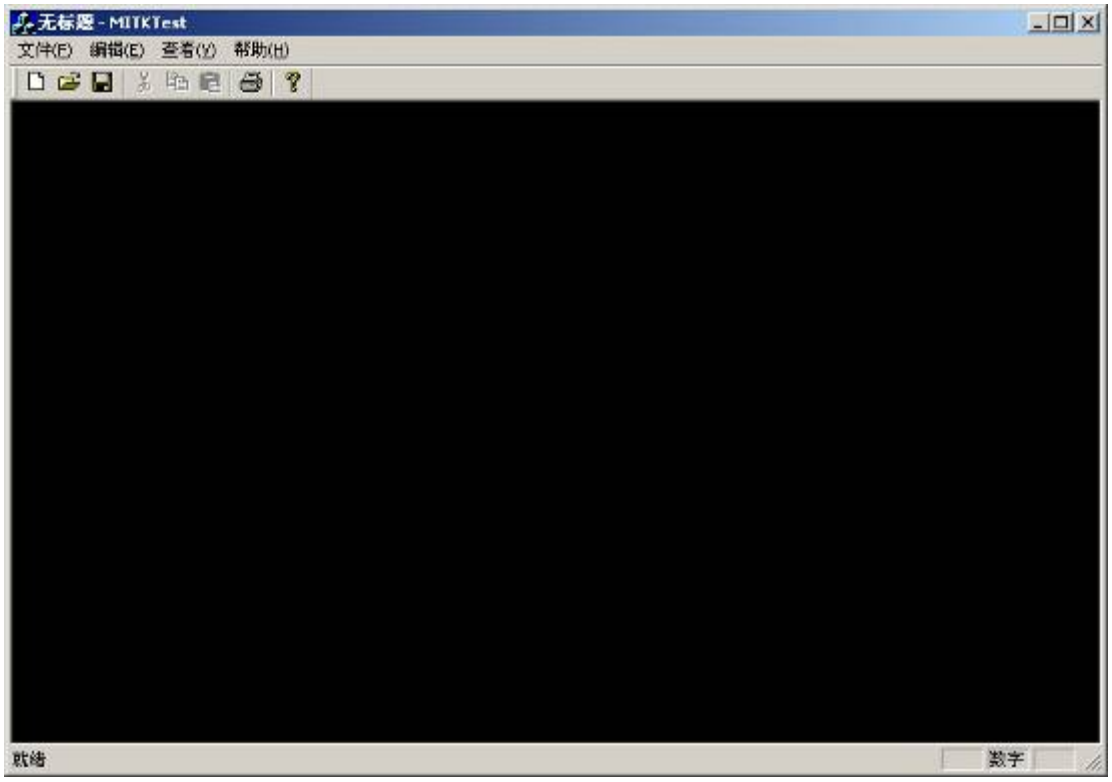



图 9-7 运行结果

## 9.2 一个简单的图像浏览器

MITK的I/O部分本身支持了包括DICOM在内的多种格式医学图像文件的读写，在这一节我们就在 9.1 的基础上添砖加瓦，编写一个简单的医学图像浏览工具。

在开始之前有一点需要说明，在MITK中对图像的显示环境由一个从mitkView派生而来的类mitkImageView负责，所以对 9.1 完成的程序必须做一定的修改，主要是把CMITKTestView中的成员变量m_View类型由mitkView *变为mitkImageView *，在OnCreate()中生成的对象也变为mitkImageView。

在 MITKTestView.h 中的改动如下：

```
class mitkImageView;       //mitkImageView 类的前向声明
class CMITKTestView : public CView
```

```
{
......
protected:
    mitkImageView *m_View;     //添加一个指向 mitkImageView 的指针
......
};
```

在 MITKTestView.cpp 中的改动如下：

```
......
#include "mitkImageView.h"   //添加 mitkImageView 的头文件
......
int CMITKTestView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
......
    // 产生 mitkImageView 对象
    m_View = new mitkImageView;
......
}
```

编译后运行结果应该与 图 9-7 没什么区别。

接下来就进入本节的正题，让我们一步一步地做出这个图像浏览器。

第一步：添加打开文件的菜单项。

MITK 的 I/O 部分目前支持 6 种数据格式：DICOM、JPEG、BMP、TIFF、IM0、Raw，第一种是医学影像的专用格式，MITK 的实现基于 DICOM 标准 3.0。接下来 3 种是常见的图像文件格式，最后两种是体数据文件格式。IM0 后面还会遇到（MITK 网站上提供的测试文件就是这种格式）， Raw 是生数据格式，其本身只保存图像数据，不包含任何与图像相关的信息（如图像长、宽等）。后面会讲怎么用 MITK 读这些数据。先让我们把读取这些文件的菜单项加上。

首先，将“文件”菜单下的“打开”项改为“Pop-up”方式的菜单项，如 图 9-8 所示。

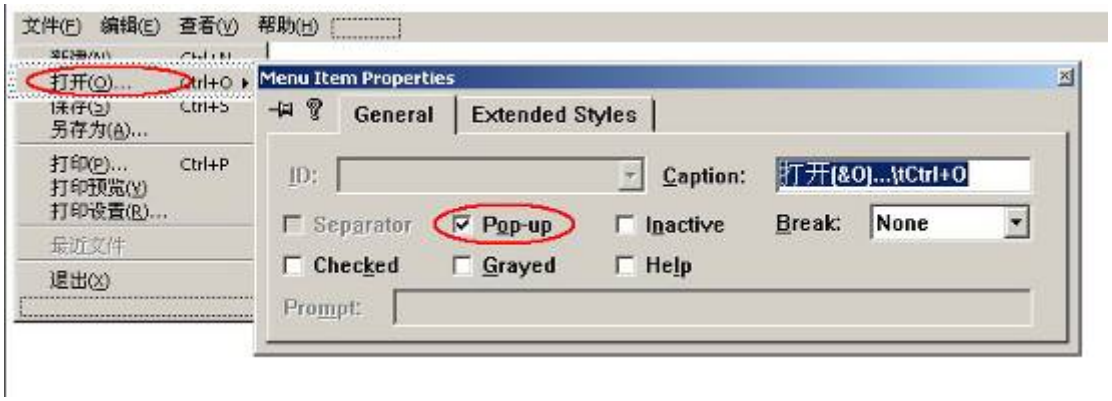



图 9-8 修改“打开”菜单项

接下来在后面弹出的菜单栏中依次添加相应的打开文件菜单项，如 图 9-9 所示是添加“打开DICOM文件”的菜单项。各菜单项的ID依次取：

```
ID_FILE_OPEN_DICOM
ID_FILE_OPEN_BMP
ID_FILE_OPEN_JPEG
ID_FILE_OPEN_TIFF
ID_FILE_OPEN_IM0
ID_FILE_OPEN_RAW
```

最后完成的菜单如 图 9-10 所示。

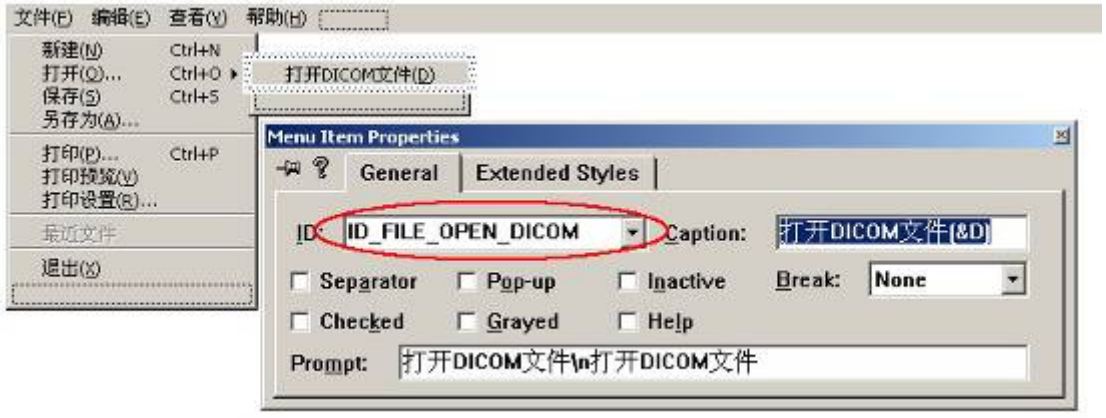



图 9-9 添加“打开 DICOM 文件”的菜单项


图 9-10 完成后的菜单

第二步：在 CMITKTestDoc 类里面添加一个 mitkVolume *成员变量，用来存放读入的数据。同时添加一个保护成员函数 clearVolume()在读入新数据时清理原先的数据；添加一个公共成员函数 GetVolume()以使 CMITKTestView 中能得到指向 mitkVolume 数据的指针。

mitkVolume 是 MITK 中最基础的数据对象之一，代表一个三维断层图像数据集，简单的说，就是一系列切片图像（slices）堆积在一起组成的，所有的输入图像数据都要表达成 Volume 才能进行后续处理。

在 MITKTestDoc.h 中的相关代码如下：

```
class mitkVolume;       //mitkVolume 类的前向声明
class CMITKTestDoc : public CDocument
{
......
// Attributes
public:
    // 提供取得指向 mitkVolume 指针的接口
    mitkVolume* GetVolume() { return m_Volume; }
......
protected:
    // 清理 Volume
    void clearVolume();

    // 指向 mitkVolume 的指针
    mitkVolume *m_Volume;
```

```
......
};
```

在大多数情况下，临时生成的对象如果以某种方式 Add/Set 进 MITK 中去之后， MITK 就替你接管了这个对象，什么时候该从内存中删除它将由 MITK 自行决定。比如将一个 mitkVolume 对象作为输入数据通过某个算法对象的 SetInput()函数加入到该算法中，则当算法对象被删除时其中的输入数据也一并被删除，也就是说算法对象向外输出结果但不保留原始数据。大多数时候这是符合实际的，但是也有特殊情况，比如此处的 mitkVolume 对象，我们希望能够控制它的生命周期而不让 MITK 过早地删除这个对象，这时就要通过调用 mitkVolume 对象的 AddReference()函数向 MITK 表明“我也在引用这个对象，请不要删除它”。而当你需要要删除它的时候，调用 RemoveReference()函数解除引用，该函数会判断当前此对象的被引用数，如果值为 0 则删除这个对象。（注意，不要再调用 Delete()了，在 RemoveReference()之前调用 Delete()将不起任何作用，而在 RemoveReference()之后调用 Delete()可能会引起不可预期的错误）。

经过这些解释再看 CMITKTestDoc.cpp 里面的相关代码应该不难理解了：

```
#include "stdafx.h"
#include "MITKTest.h"

#include "MITKTestDoc.h"

#include "mitkVolume.h"   //mitkVolume 的头文件
......
/////////////////////////////////////////////////////////////////////////
// CMITKTestDoc construction/destruction

CMITKTestDoc::CMITKTestDoc()
{
    // TODO: add one-time construction code here
    m_Volume = NULL;    //初始化指针变量

}

CMITKTestDoc::~CMITKTestDoc()
{
    this->clearVolume();    //清理 Volume 数据
```

```
}
......
////////////////////////////////////////////////////////////////////
// CMITKTestDoc commands
void CMITKTestDoc::clearVolume()
{
    if (m_Volume)
    {
        // 解除引用
        m_Volume->RemoveReference();

        // 这时候的 m_Volume 是无意义的指针，应将其置为 NULL
        m_Volume = NULL;
    }
}
```

第三步：给 CMITKTestDoc 类添加代码实现读各种图像文件的功能。

首先，在ClassWizard中添加与上述 6 个打开文件菜单项ID对应的消息处理函数，图 9-11 为ID_FILE_OPEN_IM0 添加了消息处理函数，其他的消息处理函数如法炮制。

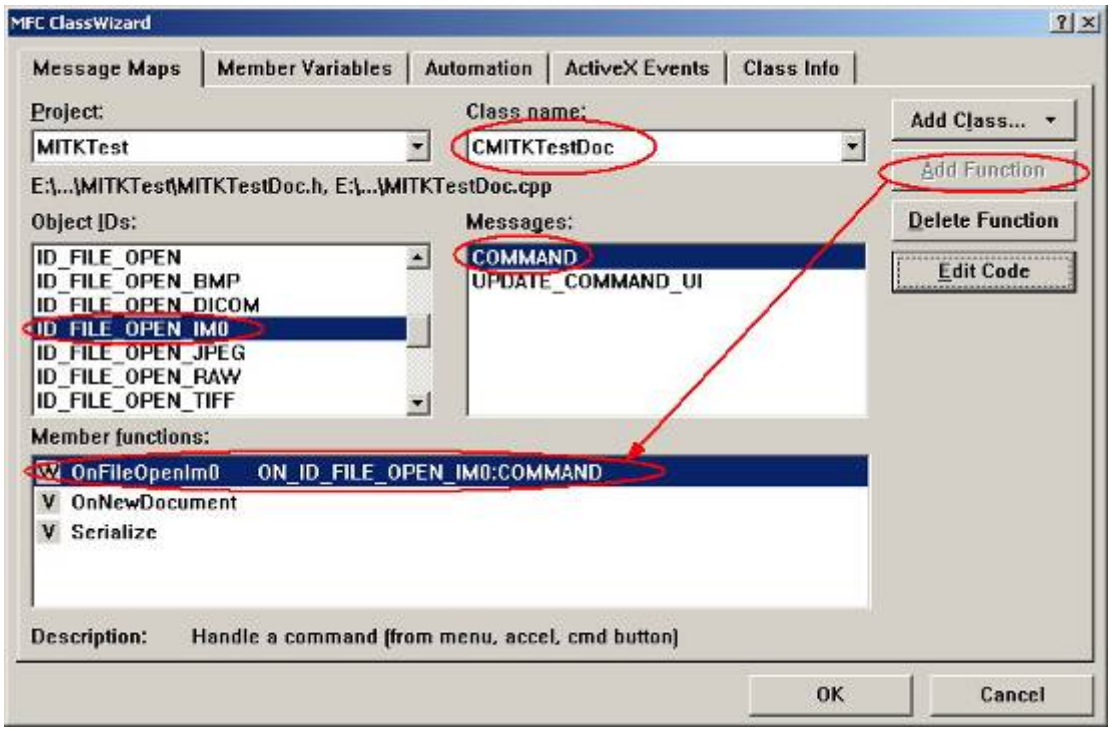



图 9-11 为 ID_FILE_OPEN_IM0 添加消息处理函数

在继续下去之前，先简要解释一下 mitk*Reader 的用法。本节所用到的 Reader 均属于 mitkVolumeReader，它是 mitkSource 的一种，是一种特殊的“算法”，它没有输入数据只有输出数据。mitkVolumeReader 顾名思义输出的就是一个 mitkVolume。

使用 mitk*Reader，首先要告诉它要读取的文件名，一个 Reader 可以处理一系列的文件，所以可以通过 AddFileName()添加多个文件名；然后要设置一些读取参数，比如对于 JPEG、TIFF、BMP，还需要设定像素间距和切片间距，因为这些信息文件本身并不提供，但它们对以后的处理是很重要的。对于 Raw，要设定的就更多了，还包括图像长宽等基本信息，而对于 IM0 和 DICOM 文件，由于其文件本身包含了所有与图像相关的信息，包括切片及像素之间的间距等等，所以不用什么额外的设定，另外 Raw 和 IM0 由于一个文件就包含了一个 Volume，所以只用设一个文件名，实际上，你加入的所有文件名对于这两个 Reader 来说，只有第一个是有用的，其他的都被忽略了；接着，就可以调用 Run() 来运行这个 Reader，读取文件里面的数据，注意，该函数返回一个 bool 型的变量，可以根据返回值来判断读文件过程中是否发生错误。如果没有错误发生，

通过 GetOutput()得到输出的 mitkVolume。

由上面的解释可知，对于BMP、JPEG、TIFF这样的通用图像存储格式，由于其文件本身并不能存储医学图像处理所需的一些信息，所以需要添加额外的对话框在打开文件时手动输入这些信息，对于Raw文件更是如此。图 9-12 是输入三个方向像素间距的对话框，在读取BMP、JPEG、TIFF格式的文件时需要用到。图 9-13 是为输入读取Raw文件时所需信息而设计的对话框，其中标题字节数指的是Raw文件开头一段自定义信息的长度（以字节为单位），MITK在读取Raw文件时会跳过这一段；大端序是为兼容Mac系列的计算机存储方式准备的；非交错存储方式是针对多通道图像而言，比如RGB三通道彩色图像，一般是采用“RGBRGB...”这种各通道像素值交错存储的方式，也有的是采用“RRR...GGG....BBB...”这种各通道分开存储的方式（按颜色平面存储）。MITK在读入图像数据时会根据这些信息对数据进行一些必要的预处理，以便统一格式。由于界面设计并非本书的主题，所以关于这两个对话框的设计请读者自行参考VC界面编程的相关书籍，这里就不再多说了。


图 9-12 设置像素间距的对话框


图 9-13 读取 Raw 文件时设置参数的对话框

下面是 MITKTestDoc.cpp 中的代码：

```
// MITKTestDoc.cpp : implementation of the CMITKTestDoc class
//

#include "stdafx.h"
#include "MITKTest.h"

#include "MITKTestDoc.h"

// 两个参数设置对话框的头文件
#include "SpacingSetDlg.h"
#include "RawSetDlg.h"

// 用到的 MITK 相关类的头文件
#include "mitkVolume.h"
#include "mitkIM0Reader.h"
#include "mitkJPEGReader.h"
#include "mitkDICOMReader.h"
#include "mitkTIFFReader.h"
#include "mitkBMPReader.h"
#include "mitkRawReader.h"

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

```
/////////////////////////////////////////////////////////////////////////
/////////
// CMITKTestDoc

IMPLEMENT_DYNCREATE(CMITKTestDoc, CDocument)

BEGIN_MESSAGE_MAP(CMITKTestDoc, CDocument)
    //{{AFX_MSG_MAP(CMITKTestDoc)
    ON_COMMAND(ID_FILE_OPEN_BMP, OnFileOpenBmp)
    ON_COMMAND(ID_FILE_OPEN_DICOM, OnFileOpenDicom)
    ON_COMMAND(ID_FILE_OPEN_IM0, OnFileOpenIm0)
    ON_COMMAND(ID_FILE_OPEN_JPEG, OnFileOpenJpeg)
    ON_COMMAND(ID_FILE_OPEN_RAW, OnFileOpenRaw)
    ON_COMMAND(ID_FILE_OPEN_TIFF, OnFileOpenTiff)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////
/////////
// CMITKTestDoc construction/destruction

CMITKTestDoc::CMITKTestDoc()
{
    // TODO: add one-time construction code here
    m_Volume = NULL;    //初始化指针变量

}

CMITKTestDoc::~CMITKTestDoc()
{
    this->clearVolume();    //清理 Volume 数据
}

BOOL CMITKTestDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add reinitialization code here
```

```
    // (SDI documents will reuse this document)

    return TRUE;
}



/////////////////////////////////////////////////////////////////////////////
// CMITKTestDoc serialization

void CMITKTestDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}

/////////////////////////////////////////////////////////////////////////////
// CMITKTestDoc diagnostics

#ifdef _DEBUG
void CMITKTestDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CMITKTestDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMITKTestDoc commands
void CMITKTestDoc::clearVolume()
```

```
{
    if (m_Volume)
    {
        // 解除引用
        m_Volume->RemoveReference();

        // 这时候的 m_Volume 是无意义的指针，应将其置为 NULL
        m_Volume = NULL;
    }
}

void CMITKTestDoc::OnFileOpenBmp()
{
    // TODO: Add your command handler code here
    // 生成打开文件对话框（可以选择多个文件）
    CFileDialog dlg(TRUE,
                    ".bmp",
                     NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT |
OFN_ALLOWMULTISELECT,
                    "BMP 文件(*.bmp)|*.bmp||",
                     NULL);

    if (dlg.DoModal() == IDOK)
    {
        // 弹出设置像素间距的对话框
        CSpacingSetDlg setDlg;
        if (setDlg.DoModal() == IDCANCEL) return;

        // 生成一个 mitkBMPReader
        mitkBMPReader *reader = new mitkBMPReader;

        // 根据对话框中的数据设置像素间距
        reader->SetSpacingX(setDlg.m_SpacingX);
        reader->SetSpacingY(setDlg.m_SpacingY);
        reader->SetSpacingZ(setDlg.m_SpacingZ);

        // 得到所有选中文件的文件名，加入到 reader 中。
        // 各个切片在 Volume 中是有排列次序的，
        // 对于 BMP 文件来说，次序信息只能从文件名得到，
        // 但为简便起见，这儿省去了对文件名进行排序的过程
```

```
        POSITION pos = dlg.GetStartPosition();
        while (pos)   reader->AddFileName(dlg.GetNextPathName(pos));

        // 运行 reader
        if (reader->Run())
        {
            // 若无错误发生，则清除旧的数据，加载新的数据
            this->clearVolume();
            m_Volume = reader->GetOutput();
            m_Volume->AddReference(); //重要!

            // 更新 View 中的显示
            this->UpdateAllViews(NULL);
        }

        // 删除 reader
        reader->Delete();
    }
}

void CMITKTestDoc::OnFileOpenDicom()
{
    // TODO: Add your command handler code here
    // 生成打开文件对话框（可以选择多个文件）
    CFileDialog dlg(TRUE,
                    NULL,
                     NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT |
OFN_ALLOWMULTISELECT,
                    "DICOM 文件(*.*)|*.*||",
                     NULL);

    if (dlg.DoModal() == IDOK)
    {
        // 生成一个 mitkDICOMReader
        mitkDICOMReader *reader = new mitkDICOMReader;

        // 得到所有选中文件的文件名，加入到 reader 中。
        // DICOM 文件本身包含了切片的位置信息，mitkDICOMReader 会根据相关信息
        // 对切片进行重排，所以这里只需要把文件名直接加入进去就可以了。
        POSITION pos = dlg.GetStartPosition();
```

```
        while (pos)   reader->AddFileName(dlg.GetNextPathName(pos));

        // 运行 reader
        if (reader->Run())
        {
            // 若无错误发生，则清除旧的数据，加载新的数据
            this->clearVolume();
            m_Volume = reader->GetOutput();
            m_Volume->AddReference(); //重要!

            // 更新 View 中的显示
            this->UpdateAllViews(NULL);
        }

        // 删除 reader
        reader->Delete();
    }
}

void CMITKTestDoc::OnFileOpenIm0()
{
    // TODO: Add your command handler code here
    // 生成打开文件对话框
    CFileDialog dlg(TRUE,
                    ".im0",
                     NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
                    "IM0 文件(*.im0)|*.im0||",
                     NULL);

    if (dlg.DoModal() == IDOK)
    {
        // 生成一个 mitkIM0Reader
        mitkIM0Reader *reader = new mitkIM0Reader;

        // 一个 IM0 文件就包含一个 Volume，只需加一个文件名
        reader->AddFileName(dlg.GetPathName());

        // 运行 reader
        if (reader->Run())
        {
```

```
            // 若无错误发生，则清除旧的数据，加载新的数据
            this->clearVolume();
            m_Volume = reader->GetOutput();
            m_Volume->AddReference(); //重要!

            // 更新 View 中的显示
            this->UpdateAllViews(NULL);
        }

        // 删除 reader
        reader->Delete();
    }
}

void CMITKTestDoc::OnFileOpenJpeg()
{
    // TODO: Add your command handler code here
    // 生成打开文件对话框（可以选择多个文件）
    CFileDialog dlg(TRUE,
                    ".jpg",
                     NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT
                          | OFN_ALLOWMULTISELECT,
                    "JPEG 文件(*.jpeg;*.jpg)|*.jpeg;*.jpg||",
                     NULL);

    if (dlg.DoModal() == IDOK)
    {
        // 弹出设置像素间距的对话框
        CSpacingSetDlg setDlg;
        if (setDlg.DoModal() == IDCANCEL) return;

        // 生成一个 mitkJPEGReader
        mitkJPEGReader *reader = new mitkJPEGReader;

        // 根据对话框中的数据设置像素间距
        reader->SetSpacingX(setDlg.m_SpacingX);
        reader->SetSpacingY(setDlg.m_SpacingY);
        reader->SetSpacingZ(setDlg.m_SpacingZ);

        // 得到所有选中文件的文件名，加入到 reader 中。
```

```
        // 各个切片在 Volume 中是有排列次序的，
        // 对于 JPEG 文件来说，次序信息只能从文件名得到，
        // 但为简便起见，这儿省去了对文件名进行排序的过程
        POSITION pos = dlg.GetStartPosition();
        while (pos)   reader->AddFileName(dlg.GetNextPathName(pos));

        // 运行 reader
        if (reader->Run())
        {
            // 若无错误发生，则清除旧的数据，加载新的数据
            this->clearVolume();
            m_Volume = reader->GetOutput();
            m_Volume->AddReference(); //重要!

            // 更新 View 中的显示
            this->UpdateAllViews(NULL);
        }

        // 删除 reader
        reader->Delete();
    }
}

void CMITKTestDoc::OnFileOpenRaw()
{
    // TODO: Add your command handler code here
    // 生成打开文件对话框
    CFileDialog dlg(TRUE,
                    ".raw",
                     NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
                    "Raw 文件(*.raw;*.*)|*.raw;*.*||",
                     NULL);

    if (dlg.DoModal() == IDOK)
    {
        // 弹出参数设置对话框
        CRawSetDlg setDlg;
        if (setDlg.DoModal() == IDCANCEL) return;

        // 生成一个 mitkRawReader
```

```
mitkRawReader *reader = new mitkRawReader;

// 根据对话框中的数据设置读取图像的参数

// 图像的尺寸以及切片数
reader->SetWidth(setDlg.m_Width);
reader->SetHeight(setDlg.m_Height);
reader->SetImageNum(setDlg.m_ImageNum);

// 三个方向的像素间距
reader->SetSpacingX(setDlg.m_SpacingX);
reader->SetSpacingY(setDlg.m_SpacingY);
reader->SetSpacingZ(setDlg.m_SpacingZ);

// 通道数
reader->SetChannelNum(setDlg.m_ChannelNum);

// 数据类型
// 其值为 MITK_CHAR、MITK_SHORT ... 中的一个，
// 这些数据类型的定义参见 mitkGlobal.h
reader->SetDataType(setDlg.m_DataType);

// 是否是大端序
reader->SetEndian((bool)(setDlg.m_IsBigEndian!=0));

// 各通道是否分开存储（按颜色平面存储）
reader->SetPlanarCfg((bool)(setDlg.m_IsColorByPlane!=0));

reader->AddFileName(dlg.GetPathName());

if (reader->Run())
{
    // 若无错误发生，则清除旧的数据，加载新的数据
    this->clearVolume();
    m_Volume = reader->GetOutput();
    m_Volume->AddReference(); //重要!

    // 更新 View 中的显示
    this->UpdateAllViews(NULL);
}
```

```
        // 删除 reader
        reader->Delete();
    }
}

void CMITKTestDoc::OnFileOpenTiff()
{
    // TODO: Add your command handler code here
    // 生成打开文件对话框（可以选择多个文件）
    CFileDialog dlg(TRUE,
                    ".tif",
                     NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT |
OFN_ALLOWMULTISELECT,
                    "TIFF 文件(*.tif;*.tiff)|*.tif;*.tiff||",
                     NULL);

    if (dlg.DoModal() == IDOK)
    {
        // 弹出设置像素间距的对话框
        CSpacingSetDlg setDlg;
        if (setDlg.DoModal() == IDCANCEL) return;

        // 生成一个 mitkTIFFReader
        mitkTIFFReader *reader = new mitkTIFFReader;

        // 根据对话框中的数据设置像素间距
        reader->SetSpacingX(setDlg.m_SpacingX);
        reader->SetSpacingY(setDlg.m_SpacingY);
        reader->SetSpacingZ(setDlg.m_SpacingZ);

        // 得到所有选中文件的文件名，加入到 reader 中。
        // 各个切片在 Volume 中是有排列次序的，
        // 对于 TIFF 文件来说，次序信息只能从文件名得到，
        // 但为简便起见，这儿省去了对文件名进行排序的过程
        POSITION pos = dlg.GetStartPosition();
        while (pos)   reader->AddFileName(dlg.GetNextPathName(pos));
        if (reader->Run())
        {
            // 若无错误发生，则清除旧的数据，加载新的数据
```

```
            this->clearVolume();
            m_Volume = reader->GetOutput();
            m_Volume->AddReference(); //重要!

            // 更新 View 中的显示
            this->UpdateAllViews(NULL);
        }

        // 删除 reader
        reader->Delete();
    }
}
```

第四步：在 CMITKTestView 里面添加一个 mitkImageModel*成员，用来显示 Volume 里面的图像。

MITK 的 View 并不直接绘制场景中的对象，它只是提供一个显示环境，同时对场景中的对象进行管理。场景中可显示的对象被抽象成 Model，一个 View 可以带多个 Model，通过 AddModel()添加新的 Model，RemoveModel()移除场景中已有的 Model。View 遍历它所包含的所有 Model，调用它的 Render()函数来绘制 Model，所以实际对象的绘制是由其对应的 Model 来完成的。

在本节中用到了 mitkImageModel，它是专门用来显示 Volume 中断层图像的 Model，它有三种显示方式：显示 X-Y、Y-Z 或 Z-X 平面的断层图像，我们现在只用了缺省的方式来显示 X-Y 平面的断层图像，其实就是读入的图像。

在 CMITKTestView 里添加一个 mitkImageModel*类型的成员变量：

```
class mitkImageView;       //mitkImageView 类的前向声明
class mitkImageModel;      //mitkImageModel 类的前向声明
class CMITKTestView : public CView
{
......
protected:
    mitkImageView *m_View;          //添加一个指向 mitkImageView 的指针
    mitkImageModel *m_ImageModel;   //添加一个指向 mitkImageModel 的指针
......
};
```

在 MITKTestView.cpp 中添加代码，显示读入的图像：

```
// MITKTestView.cpp : implementation of the CMITKTestView class
//

#include "stdafx.h"
#include "MITKTest.h"

#include "MITKTestDoc.h"
#include "MITKTestView.h"

// 相关头文件
#include "mitkImageView.h"
#include "mitkImageModel.h"

......

/////////////////////////////////////////////////////////////////////////
// CMITKTestView construction/destruction

CMITKTestView::CMITKTestView()
{
    // TODO: add construction code here
    // 初始化指针变量为 NULL
    m_View = NULL;
    m_ImageModel = NULL;
}

......

/////////////////////////////////////////////////////////////////////////
// CMITKTestView drawing

void CMITKTestView::OnDraw(CDC* pDC)
{
    CMITKTestDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    // Volume 数据更新，m_ImageModel 也相应更新
    if (m_ImageModel->GetData() != pDoc->GetVolume())
        m_ImageModel->SetData(pDoc->GetVolume());
}
```

```
......

/////////////////////////////////////////////////////////////////////////
// CMITKTestView message handlers

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

    // TODO: Add your specialized creation code here
    // 得到当前客户区大小
    RECT clientRect;
    this->GetClientRect(&clientRect);
    int wWidth = clientRect.right - clientRect.left;
    int wHeight = clientRect.bottom - clientRect.top;

    // 产生 mitkView 对象
    m_View = new mitkImageView;

    // 设置父窗口句柄
    m_View->SetParent(GetSafeHwnd());

    // 设置 mitkView 在父窗口中显示的位置和大小
    m_View->SetLeft(0);
    m_View->SetTop(0);
    m_View->SetWidth(wWidth);
    m_View->SetHeight(wHeight);

    // 设置 mitkView 的背景颜色（这里将其设置为黑色）
    m_View->SetBackColor(0, 0, 0);

    // 生成一个 mitkImageModel 并将其加入到 View 中
    m_ImageModel = new mitkImageModel;
    m_View->AddModel(m_ImageModel);

    // 显示 mitkView
    m_View->Show();
```

```
    return 0;
}

......

void CMITKTestView::OnDestroy()
{
    CView::OnDestroy();

    // TODO: Add your message handler code here
    // 删除 m_View
    // 注意：不需要删除 m_ImageModel，
    // 自 m_ImageModel 通过 AddModel()加入到 m_View 中起，
    // 其生命周期由 m_View 来管理，
    // 当删除 m_View 时，m_View 会负责删除它所带的所有 Model
    if (m_View)
    {
        m_View->Delete();
        m_View = NULL;
    }
}

......
```

至此，编译运行，应该可以打开图像文件了。mitkImageView在缺省情况下会在图像上加上一个十字箭头，如 图 9-14 所示。可以在CMITKTestView中通过调用m_View->SetCrossArrow(false)去掉这个箭头，我们将这句加在OnCreate()函数中：

```
int CMITKTestView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
......
    // 设置 mitkView 的背景颜色（这里将其设置为黑色）
    m_View->SetBackColor(0, 0, 0);

    // 生成一个 mitkImageModel 并将其加入到 View 中
    m_ImageModel = new mitkImageModel;
    m_View->AddModel(m_ImageModel);
```

```
    // 将缺省显示的十字箭头去掉
    m_View->SetCrossArrow(false);

    // 显示 mitkView
    m_View->Show();

    return 0;
}
```

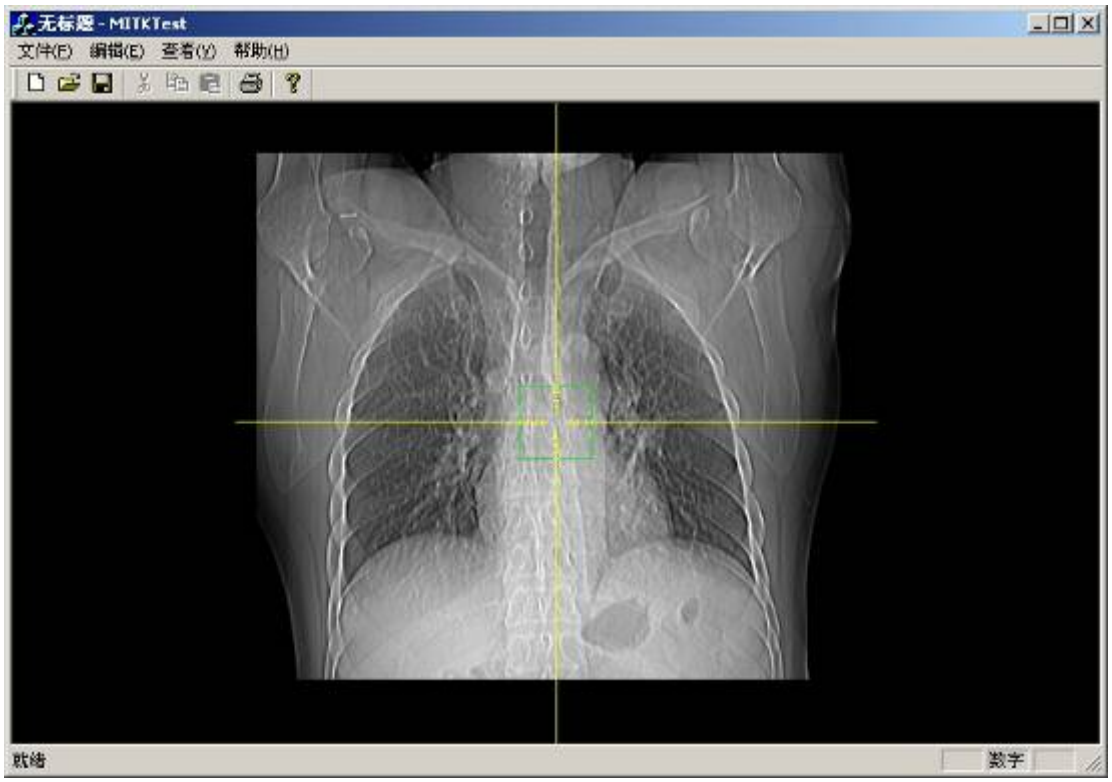



图 9-14 读入并显示一个 DICOM 文件

到这里为止，基本功能就都已经完成了。但是一般情况下，一个 Volume 总是由多张切片组成的，现在的程序还没有在同一个 Volume 中浏览各张切片图像的功能，下面就加上这一功能。

在工具栏添加一组按钮，如 图 9-15 所示。ID分别取为ID_SLICE_FIRST、ID_SLICE_PREV、ID_SLICE_NEXT、ID_SLICE_LAST。

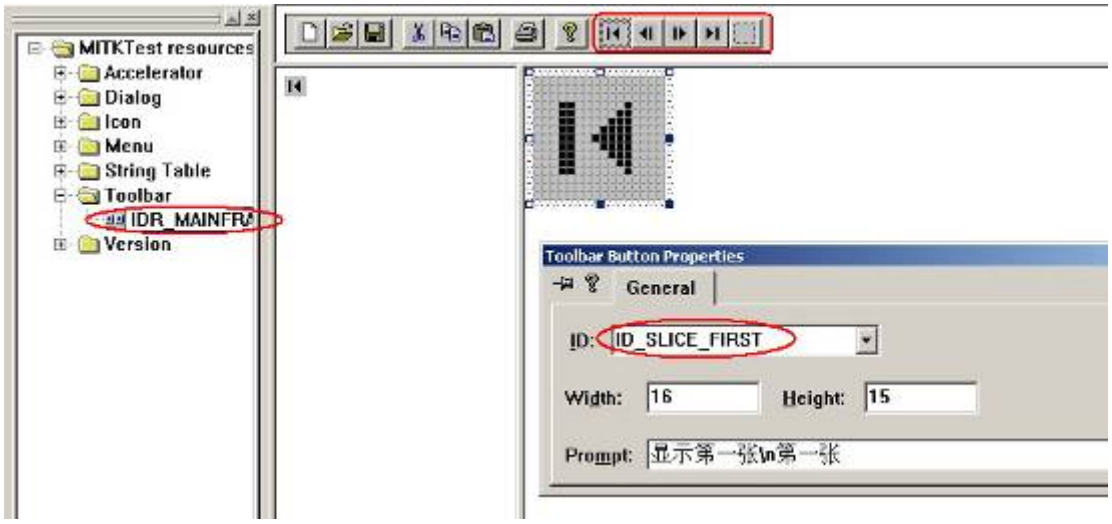



图 9-15 在工具栏添加浏览切片的按钮

在ClassWizard里给上述ID添加相应的消息处理函数，如 图 9-16 所示。

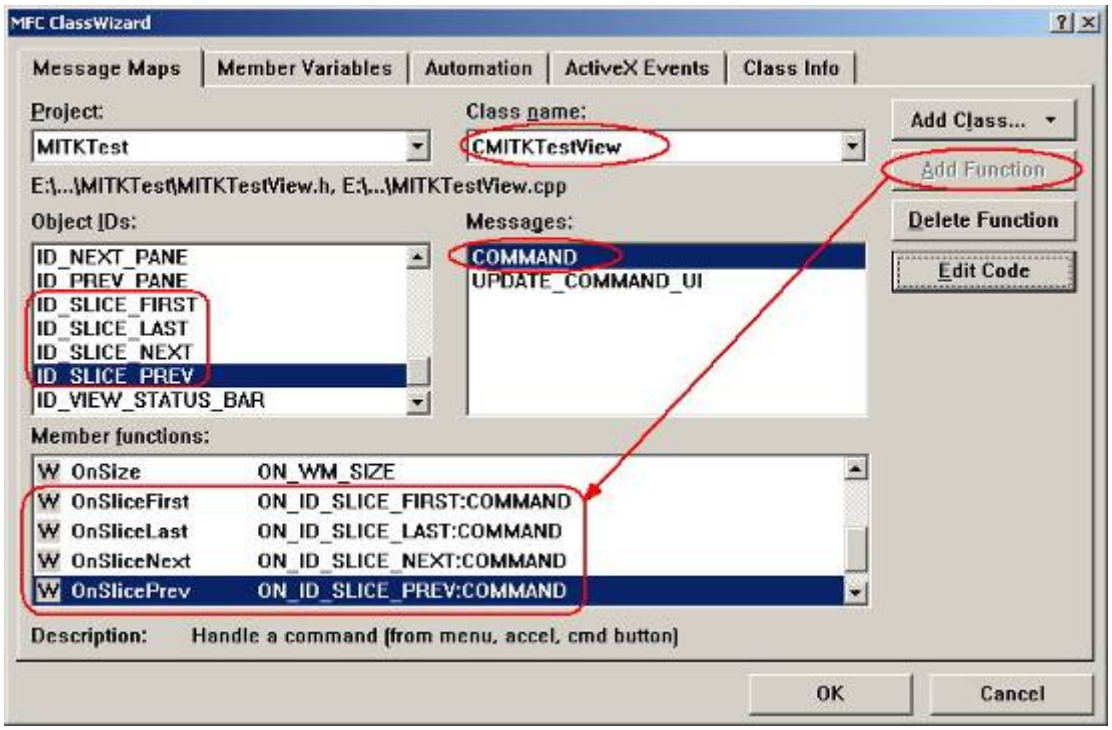



图 9-16 给按钮 ID 添加消息处理函数

然后，在 MITKTestView.cpp 中添加相应代码实现浏览功能：

```
void CMITKTestView::OnSliceFirst()
```

```
{
    // TODO: Add your command handler code here
    if (m_ImageModel)
    {
        // 定位在第 1 张切片
        m_ImageModel->SetCurrentSliceNumber(0);

        // 更新 View 中的显示
        m_View->Update();
    }
}

void CMITKTestView::OnSliceLast()
{
    // TODO: Add your command handler code here
    if (m_ImageModel)
    {
        // 定位到最后一张切片
        m_ImageModel->SetCurrentSliceNumber(
                    m_ImageModel->GetTotalSliceNumber()-1);

        // 更新 View 中的显示
        m_View->Update();
    }
}

void CMITKTestView::OnSliceNext()
{
    // TODO: Add your command handler code here
    if (m_ImageModel)
    {
        // 定位到下一张切片
        m_ImageModel->NextSlice();

        // 更新 View 中的显示
        m_View->Update();
    }
}

void CMITKTestView::OnSlicePrev()
```

```
{
    // TODO: Add your command handler code here
    if (m_ImageModel)
    {
        // 定位到前一张切片
        m_ImageModel->PrevSlice();

        // 更新 View 中的显示
        m_View->Update();
    }
}
```

好了，编译运行，最后结果如 图 9-17 所示。

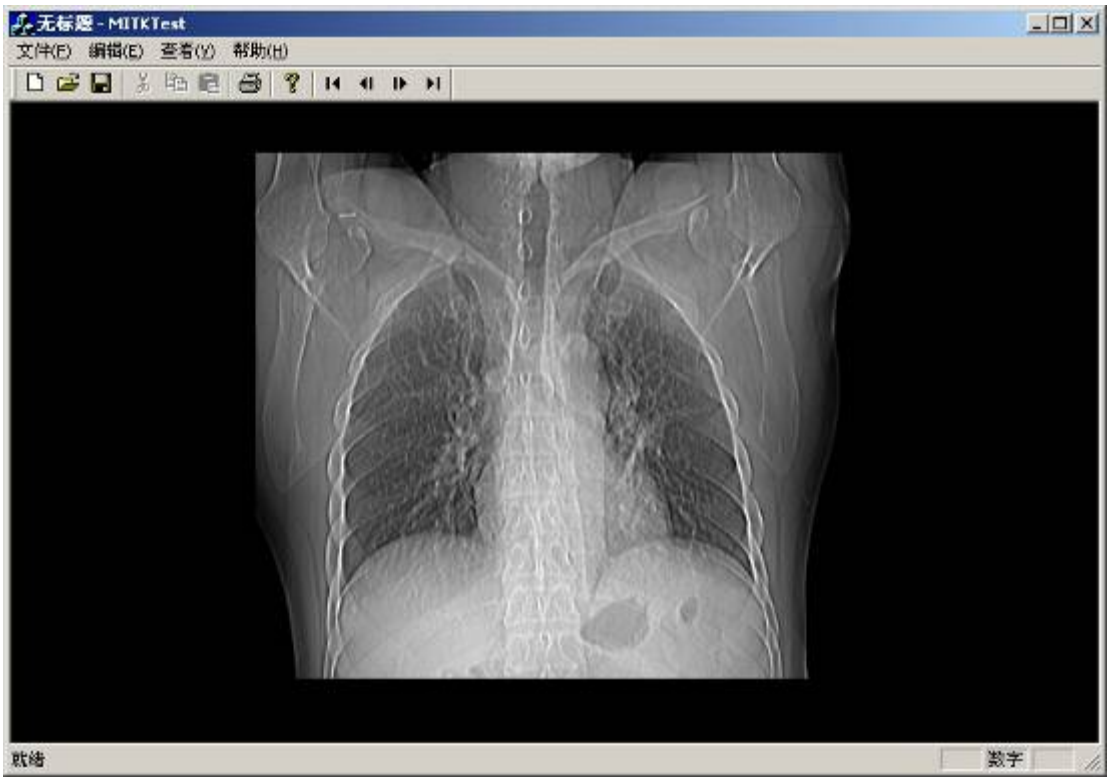



图 9-17 运行结果

mitkImageView 所带的缺省 Manipulator 已经实现了常用的鼠标操作，所以不用写一行代码，你的程序就拥有了对图像进行平移、缩放和窗宽/窗位调整的功能，其中，按住鼠标左键拖动鼠标是平移图像；按住鼠标右键拖动鼠标是对图像进行缩放；按住鼠标中键拖动鼠标是调整图像的窗宽/窗位，水平移动调整的是窗宽，垂直移动调整的是窗位。

## 9.3 用 MITK 进行表面重建

利用 MITK 提供的等值面提取算法可以对 Volume 数据进行表面重建，重建的结果可以通过 Surface Model 显示，下面的例子就展示了如何用 MITK 来实现这些功能。

回到 9.1 建立的工程，按 9.2 的方法加入读取Volume数据的部分。然后给CMITKTestDoc类添加一个mitkMesh*类型的成员变量，mitkMesh在MITK中是除mitkVolume之外的另一大数据类型，代表一个三维的几何数据对象，具体的说就是用三角片来表示的物体表面模型，表面重建输出的就是这种类型的数据。MITKTestDoc.h及MITKTestDoc.cpp中添加的代码如下，基本上和mitkVolume成员相关代码是对称的：

```
// MITKTestDoc.h : interface of the CMITKTestDoc class
//
/////////////////////////////////////////////////////////////////////////////

......

class mitkVolume;      //mitkVolume 类的前向声明
class mitkMesh;        //mitkMesh 类的前向声明
class CMITKTestDoc : public CDocument
{
protected: // create from serialization only
    CMITKTestDoc();
    DECLARE_DYNCREATE(CMITKTestDoc)

// Attributes
public:
    // 提供取得指向 mitkVolume 指针的接口
    mitkVolume* GetVolume() { return m_Volume; }

    // 提供取得指向 mitkMesh 指针的接口
    mitkMesh* GetMesh() { return m_Mesh; }

......

protected:
```

```
    void clearVolume();     //清理 Volume
    void clearMesh();       //清理 Mesh

    mitkVolume *m_Volume;   //指向 mitkVolume 的指针
    mitkMesh   *m_Mesh;     //指向 mitkMesh 的指针

......
};
......


// MITKTestDoc.cpp : implementation of the CMITKTestDoc class
//

#include "stdafx.h"
#include "MITKTest.h"

#include "MITKTestDoc.h"
#include "SpacingSetDlg.h"
#include "RawSetDlg.h"
#include "ThresholdDlg.h"

#include "mitkVolume.h"
#include "mitkMesh.h"   // 添加 mitkMesh 类的头文件
#include "mitkIM0Reader.h"
#include "mitkJPEGReader.h"
#include "mitkDICOMReader.h"
#include "mitkTIFFReader.h"
#include "mitkBMPReader.h"
#include "mitkRawReader.h"

......

/////////////////////////////////////////////////////////////////////////
// CMITKTestDoc construction/destruction

CMITKTestDoc::CMITKTestDoc()
{
    // TODO: add one-time construction code here
    // 初始化指针变量
```

```
    m_Volume = NULL;
    m_Mesh = NULL;
}

CMITKTestDoc::~CMITKTestDoc()
{
    this->clearVolume();    //清理 Volume 数据
    this->clearMesh();      //清理 Mesh 数据
}

......

////////////////////////////////////////////////////////////////////
// CMITKTestDoc commands

void CMITKTestDoc::clearVolume()
{
    if (m_Volume)
    {
        // 解除引用
        m_Volume->RemoveReference();

        // 这时候的 m_Volume 是无意义的指针，应将其置为 NULL
        m_Volume = NULL;
    }
}

void CMITKTestDoc::clearMesh()
{
    if (m_Mesh)
    {
        // 解除引用
        m_Mesh->RemoveReference();

        // 这时候的 m_Mesh 是无意义的指针，应将其置为 NULL
        m_Mesh = NULL;
    }
}
......
```

接着在工具栏添加一个按钮，其功能就是对读入的Volume数据进行表面重建，如 图 9-18 所示。

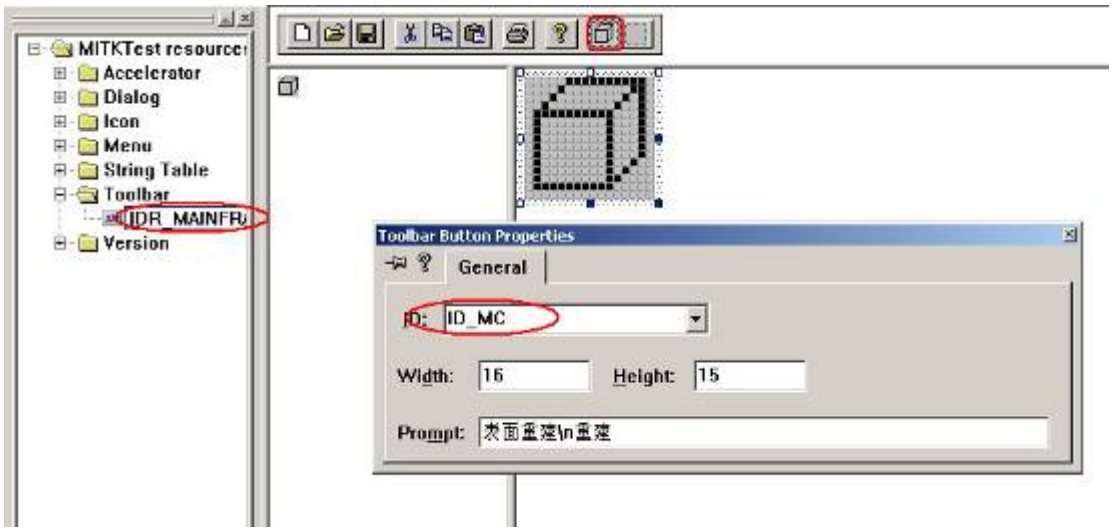



图 9-18 添加表面重建按钮

为改按钮ID添加消息处理函数，注意，要添加在CMITKTestDoc类中，如 图 9-19 所示。

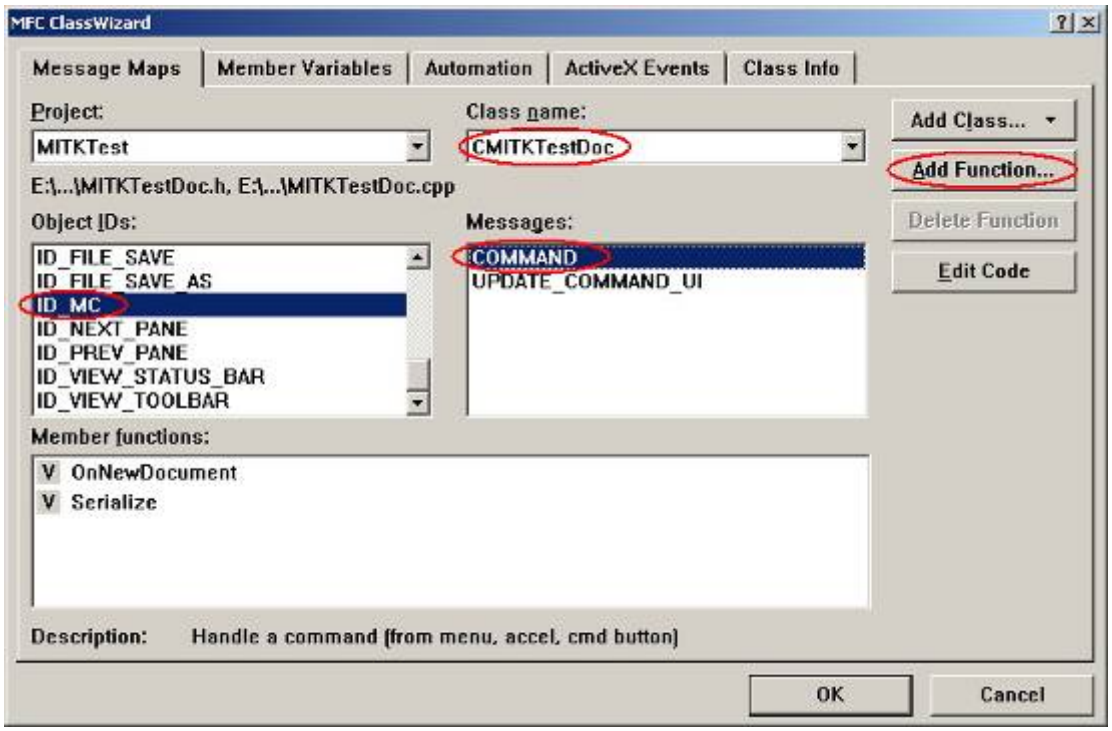



图 9-19 给 ID_MC 添加消息处理函数

另外，还需加一个对话框，接受用户输入的阈值作为重建算法的输入参数，如 图 9-20 所示。


图 9-20 设置阈值的对话框

现在可以来写表面重建部分的代码了。

这里，我们使用标准的 Marching Cubes 算法来进行表面重建，在 MITK 中，该算法被封装在 mitkMarchingCubes 类中，该类是一个 Filter，代表 Marching Cubes 算法对象，它接收一个 mitkVolume 输入，经过处理，输出一个 mitkMesh 对象，具体代码如下：

```
// MITKTestDoc.cpp : implementation of the CMITKTestDoc class
//
```

```
......

#include "mitkMarchingCubes.h"  //包含 mitkMarchingCubes 头文件
......

void CMITKTestDoc::OnMc()
{
    // TODO: Add your command handler code here
    // 产生阈值设置对话框
    CThresholdDlg dlg;

    // 显示阈值设置对话框
    if (dlg.DoModal() == IDOK)
    {
        // 生成一个 mitkMarchingCubes 对象
        mitkMarchingCubes *mc = new mitkMarchingCubes;

        // 将从对话框中得到的阈值设置给 Marching Cubes 算法
        mc->SetThreshold(dlg.m_LowValue, dlg.m_HighValue);

        // 设置输入数据
        mc->SetInput(m_Volume);

        // 运行该算法，如果算法正常结束则更新 Mesh 数据并显示
        if (mc->Run())
        {
            // 清除旧的 Mesh 数据
            this->clearMesh();

            // 从 mitkMarchingCubes 算法得到输出结果
            m_Mesh = mc->GetOutput();
            m_Mesh->AddReference(); //重要

            // 更新 View
            UpdateAllViews(NULL);
        }

        // 删除 mitkMarchingCubes 对象
        mc->Delete();
    }
```

```
}
```

接下来在 CMITKTestView 类中添加代码将生成的 mitkMesh 对象显示出来。

首先给 CMITKTestView 类添加一个 mitkSurfaceModel*类型的成员变量。在 MITK 中，mitkSurfaceModel 属于一种 Data Model，表示某数据对象在三维场景中的显示模型，mitkSurfaceModel 即是对应于 mitkMesh 对象的显示模型，通过三维重建得到的 mitkMesh 对象将由 mitkSurfaceModel 来负责绘制。代码如下：

```
// MITKTestView.h : interface of the CMITKTestView class
//
/////////////////////////////////////////////////////////////////////////////

......

class mitkView;           //mitkView 类的前向声明
class mitkSurfaceModel;    //mitkSurfaceModel 类的前向声明
class CMITKTestView : public CView
{

......

protected:
    //添加一个指向 mitkView 的指针
    mitkView *m_View;

    //添加一个指向 mitkSurfaceModel 的指针
    mitkSurfaceModel *m_SurfaceModel;

......

};

......
```

在 MITKTestView.cpp 开头添加“#include “mitkSurfaceModel.h””，修改 OnCreate()函数，添加 mitkSurfaceModel 的初始化代码：

```
int CMITKTestView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
```

```
        return -1;

    // TODO: Add your specialized creation code here
    // 得到当前客户区大小
    RECT clientRect;
    this->GetClientRect(&clientRect);
    int wWidth = clientRect.right - clientRect.left;
    int wHeight = clientRect.bottom - clientRect.top;

    // 产生 mitkView 对象
    m_View = new mitkView;

    // 设置父窗口句柄
    m_View->SetParent(GetSafeHwnd());

    // 设置 mitkView 在父窗口中显示的位置和大小
    m_View->SetLeft(0);
    m_View->SetTop(0);
    m_View->SetWidth(wWidth);
    m_View->SetHeight(wHeight);

    // 设置 mitkView 的背景颜色（这里将其设置为黑色）
    m_View->SetBackColor(0, 0, 0);

    // 显示 mitkView
    m_View->Show();

    // 生成一个 mitkSurfaceModel
    m_SurfaceModel = new mitkSurfaceModel;

    // 设置表面材质属性（这些属性可以随时调整）
   m_SurfaceModel->GetProperty()->SetAmbientColor(0.75f, 0.75f, 0.75f,
1.0f);
    m_SurfaceModel->GetProperty()->SetDiffuseColor(1.0f, 0.57f, 0.04f,
1.0f);
    m_SurfaceModel->GetProperty()->SetSpecularColor(1.0f, 1.0f, 1.0f,
1.0f);
    m_SurfaceModel->GetProperty()->SetSpecularPower(100.0f);
    m_SurfaceModel->GetProperty()->SetEmissionColor(0.0f, 0.0f, 0.0f,
0.0f);
```

```
    // 将 Model 加入到 View 中
    m_View->AddModel(m_SurfaceModel);

    return 0;
}
```

修改 OnDraw()函数以便在绘制之前保证 mitkMesh 数据的有效性：

```
void CMITKTestView::OnDraw(CDC* pDC)
{
    CMITKTestDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    // Mesh 数据更新，Surface Model 也随之更新
    if (m_SurfaceModel->GetData() != pDoc->GetMesh())
    {
        m_SurfaceModel->SetData(pDoc->GetMesh());
    }
}
```

完成之后编译运行，在“打开”菜单的子菜单中选择一种格式的文件打开，读入一个Volume，然后按下  ，在弹出的“设置阈值”对话框中输入阈值进行重建。阈值的选择取决于要重建表面的物质的性质，需要一些先验知识，主要是该物质的密度（在图像中表现为灰度值）在哪个范围之内。当输入合适的阈值后点“确定”按钮即开始表面重建，稍候片刻其结果就会显示出来，在你可以用鼠标对结果进行一些简单的操作：按住左键拖动鼠标是旋转，中键是平移，邮件是缩放。图 9-21 是一个重建结果的示例。

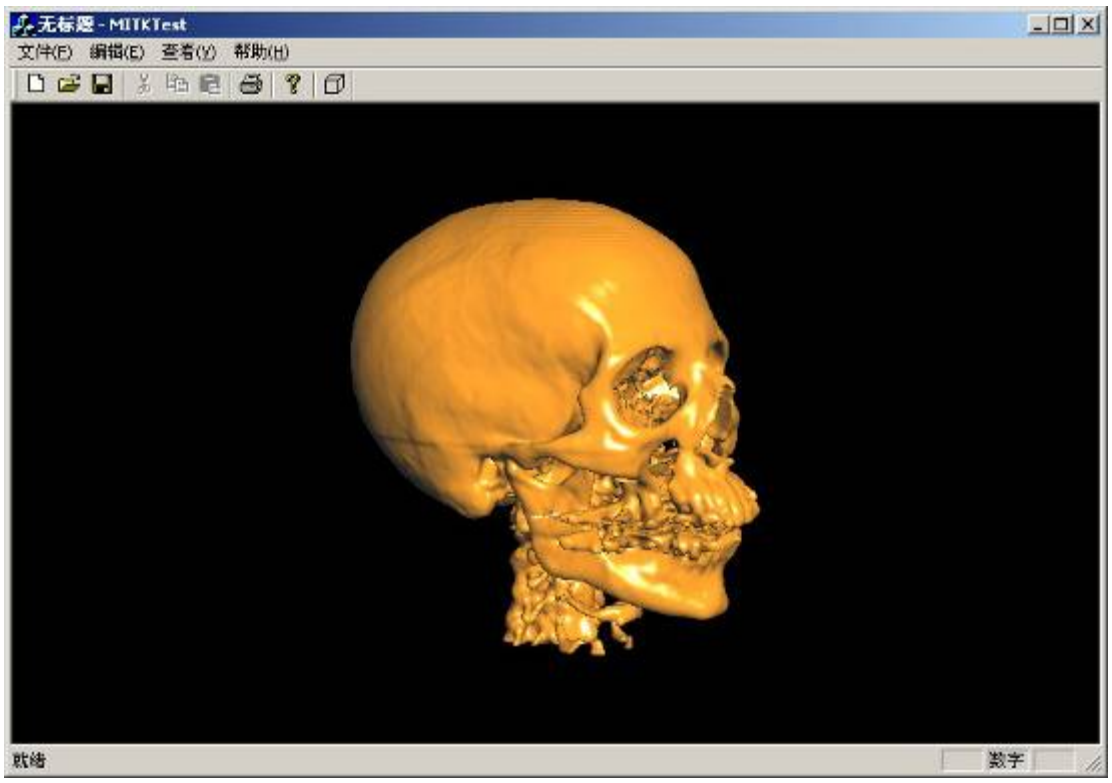



图 9-21 重建结果示例

## 9.4 一个比较完善的例子

9.2 和 9.3 都是比较简单的例子，功能比较单一。在这一节，我们将综合上面两节所完成的功能，作出一个比较完善的应用程序。

先来看一下完成后的整个应用程序的运行状况。

所 图 9-22 示是正在进行表面重建时的显示界面，在这个例子中增加了重建进度的显示，如所示。图 9-23 所示是重建结束后的程序界面。主窗口的整个客户区被划分为左右两部分，右边比较大的显示区用来显示经过表面重建生成的三维模型，左边又被划分为三个小的显示区，分别显示读入的Volume数据X-Y、Y-Z及Z-X平面的断层图像。

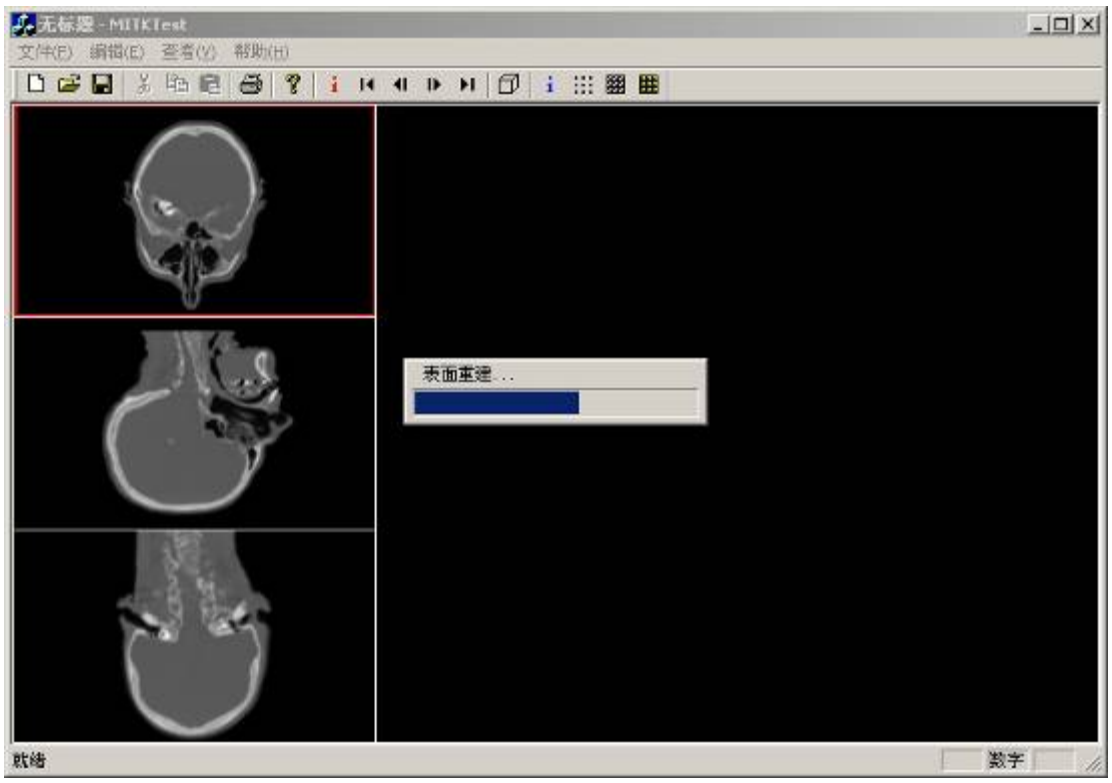



图 9-22 表面重建时显示进度

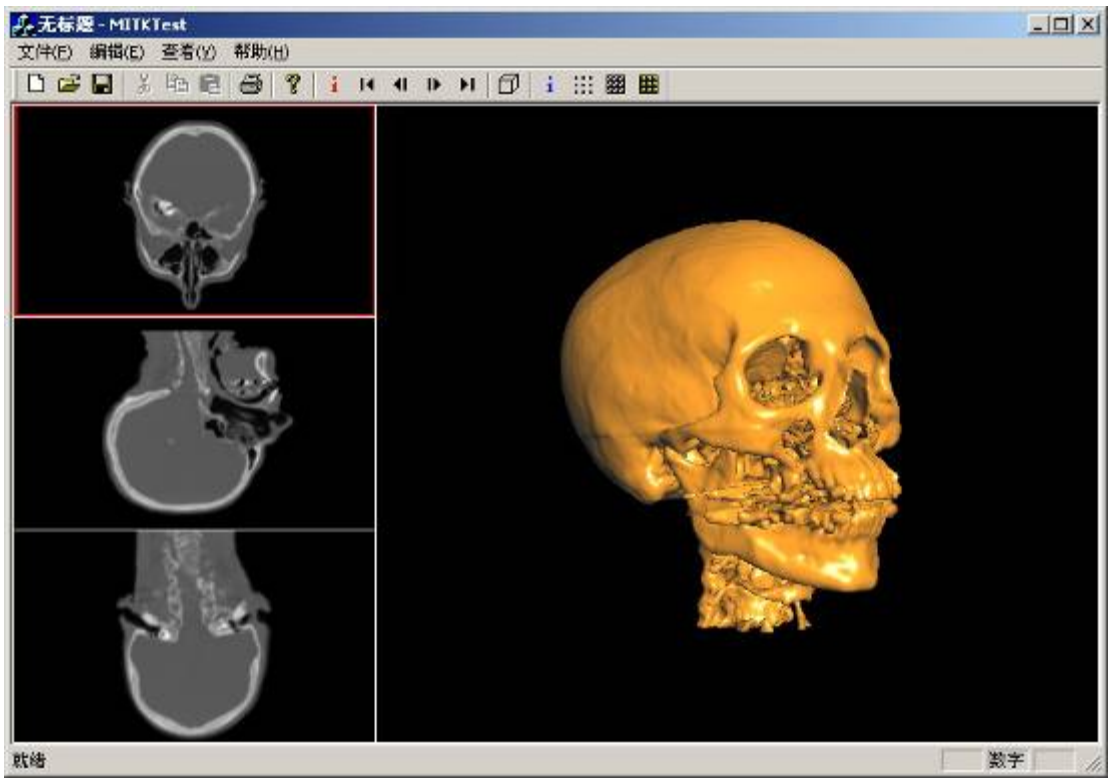



图 9-23 重建完成后的界面

工具栏中的按钮大部分都在 9.2 和 9.3 中介绍过了，其功能大致相同。新增按钮的功能如下：

 ：显示读入的 Volume 的相关信息；

 ：显示生成的 Mesh 的相关信息；

 ：显示生成的Mesh的所有顶点，如 图 9-24 所示；

 ：显示Mesh的线框模型，如 图 9-25 所示；

 ：显示Mesh的表面模型，如 图 9-23 界面中所示。

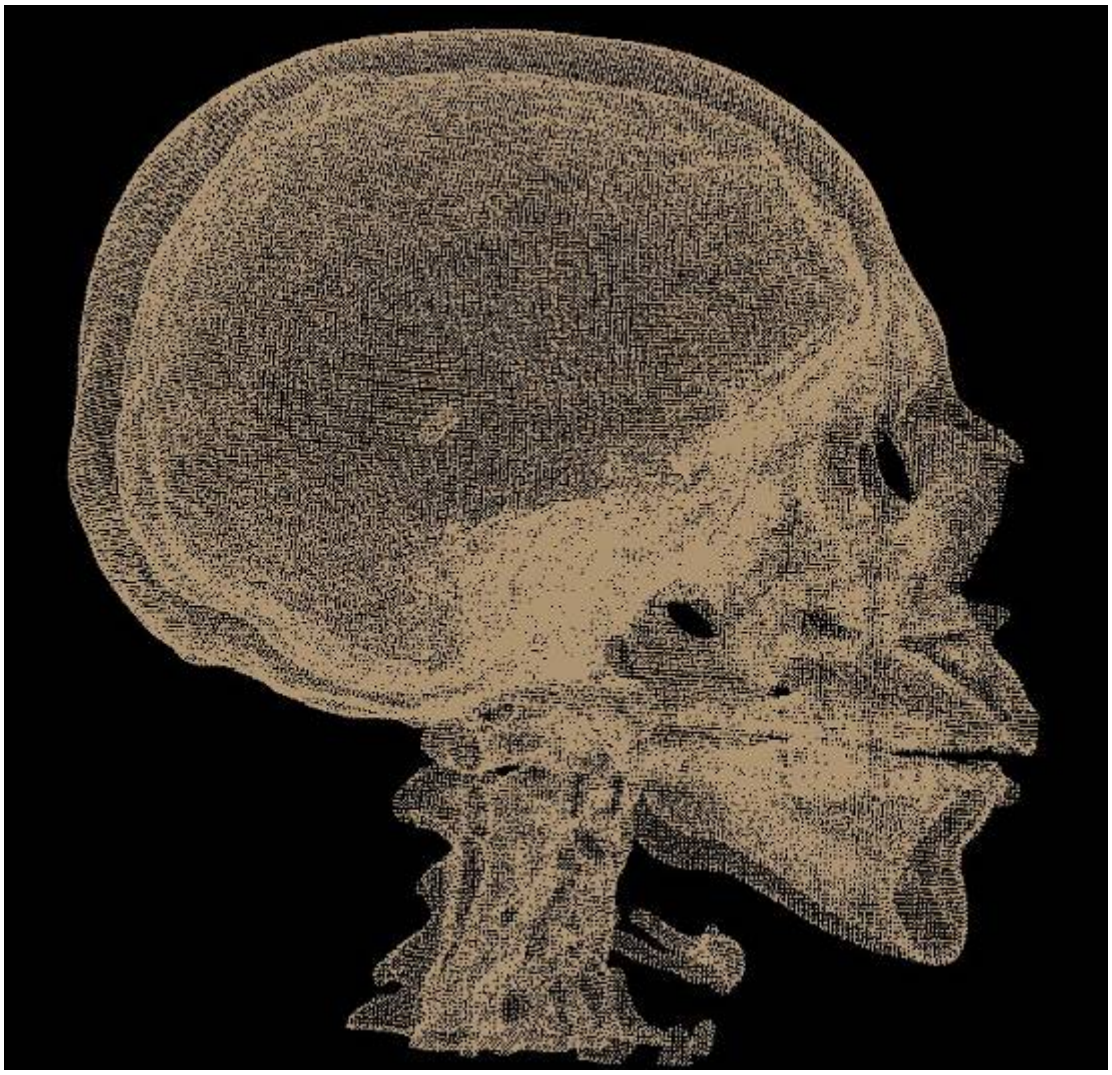


图 9-24 点显示的三维模型

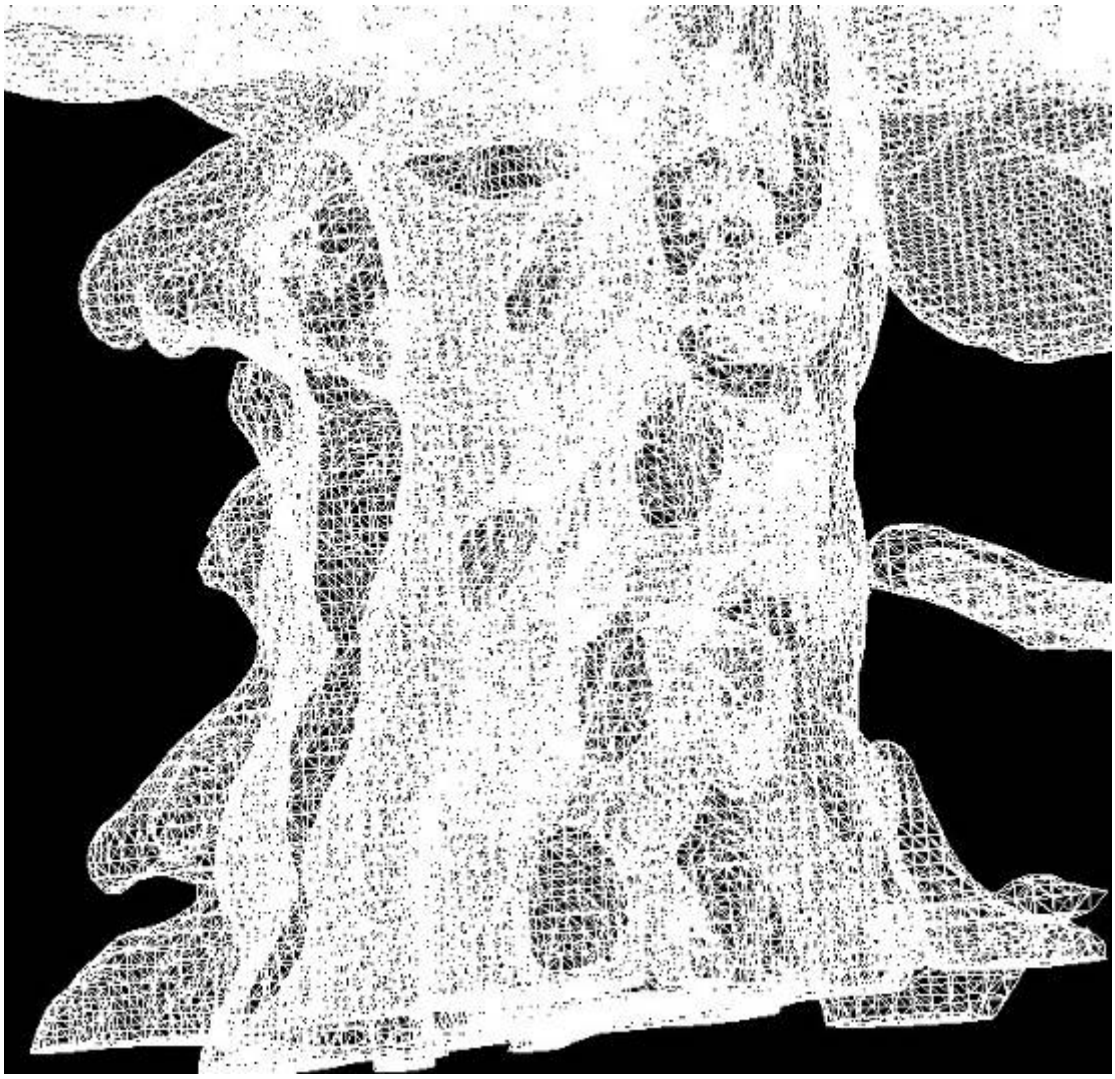


图 9-25 线框显示的三维模型

有了上面两节的基础，本节将不再拘泥于编程细节的介绍，而把主要精力放在如何实现新增的功能上。

首先是主窗口的划分。在一个父窗口下，可以同时安放多个 mitkView，通过 mitkView 的 SetLeft()、SetTop()、SetWidth()和 SetHeight()确定每一个 mitkView 在父窗口中的位置（以父窗口左上角为坐标原点）和尺寸。在本节的例子中，一共在主窗口的客户区放了 4 个 mitkView，左边三个是 mitkImageView（mitkView 的子类），用来显示三个方向的断层图像，右边一个 mitkView 用来显示表面重建生成的三维模型。完成这部分功能的代码主要集中在 MITKTestView.h 和 MITKTestView.cpp 中，如下所示：

```
// MITKTestView.h : interface of the CMITKTestView class
//
/////////////////////////////////////////////////////////////////////////////

......

class CMITKTestDoc;

class mitkImageView;      //mitkImageView 类的前向声明
class mitkImageModel;     //mitkImageModel 类的前向声明
class mitkView;           //mitkView 类的前向声明
class mitkSurfaceModel;   //mitkSurfaceModel 类的前向声明
class CMITKTestView : public CView
{
......

public:
    // 改变选中的 mitkImageView
    void ChangeSelectedImageView(mitkImageView *selView);

......

protected:
    // 以下为显示切片图像而设
    mitkImageView *m_ImageView[3];     //三个指向 mitkImageView 的指针
    mitkImageModel *m_ImageModel[3];   //三个指向 mitkImageModel 的指针
    mitkImageView *m_CurImageView;     //指向当前选中 mitkImageView 的指针
    mitkImageModel *m_CurImageModel;   //指向当前选中 mitkImageModel 的指
针

    // 以下为显示重建的表面模型而设
    mitkView *m_SceneView;            //一个指向 mitkView 的指针
    mitkSurfaceModel *m_SurfaceModel;  //一个指向 mitkSurfaceModel 的指针


// Generated message map functions
protected:
    //{{AFX_MSG(CMITKTestView)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnSize(UINT nType, int cx, int cy);
```

```
    afx_msg void OnSliceFirst();
    afx_msg void OnSliceLast();
    afx_msg void OnSliceNext();
    afx_msg void OnSlicePrev();
    afx_msg void OnDestroy();
    afx_msg void OnPoints();
    afx_msg void OnWireframe();
    afx_msg void OnSurface();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
......

// MITKTestView.cpp : implementation of the CMITKTestView class
//

#include "stdafx.h"
#include "MITKTest.h"

#include "MITKTestDoc.h"
#include "MITKTestView.h"

// 需要的头文件
#include "mitkImageView.h"
#include "mitkImageModel.h"
#include "mitkSurfaceModel.h"

#include "ChooseViewManipulator.h"

// 各个 View 之间的空隙宽度
const int VIEW_MARGIN = 1;

......

/////////////////////////////////////////////////////////////////////////////
// CMITKTestView construction/destruction

CMITKTestView::CMITKTestView()
{
    // TODO: add construction code here
```

```
    // 初始化指针变量为 NULL
    for (int i=0; i<3; ++i)
    {
        m_ImageView[i] = NULL;
        m_ImageModel[i] = NULL;
    }

    m_CurImageView = NULL;
    m_CurImageModel = NULL;

    m_SceneView = NULL;
    m_SurfaceModel = NULL;
}

CMITKTestView::~CMITKTestView()
{
}

......

/////////////////////////////////////////////////////////////////////////
// CMITKTestView drawing

void CMITKTestView::OnDraw(CDC* pDC)
{
    CMITKTestDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    // Volume 数据更新，m_ImageModel 也相应更新，且将显示的切片定位在中间
    if (m_ImageModel[0]->GetData() != pDoc->GetVolume())
    {
        m_ImageModel[0]->SetData(pDoc->GetVolume());

    m_ImageModel[0]->SetCurrentSliceNumber(m_ImageModel[0]->GetTotalS
liceNumber()/2);

        m_ImageModel[1]->SetData(pDoc->GetVolume());

    m_ImageModel[1]->SetCurrentSliceNumber(m_ImageModel[1]->GetTotalS
liceNumber()/2);
```

```
        m_ImageModel[2]->SetData(pDoc->GetVolume());

    m_ImageModel[2]->SetCurrentSliceNumber(m_ImageModel[2]->GetTotalS
liceNumber()/2);
    }

    // Mesh 数据更新，m_SurfaceModel 也相应更新
    if (m_SurfaceModel->GetData() != pDoc->GetMesh())
    {
        m_SurfaceModel->SetData(pDoc->GetMesh());
    }
}

......

/////////////////////////////////////////////////////////////////////////
// CMITKTestView message handlers

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

    // TODO: Add your specialized creation code here
    // 得到当前客户区大小
    RECT clientRect;
    this->GetClientRect(&clientRect);
    int wWidth = clientRect.right - clientRect.left + 1;
    int wHeight = clientRect.bottom - clientRect.top + 1;

    // 三个 mitkImageView 的尺寸
    int imageWidth = (int)((wWidth - VIEW_MARGIN * 3) / 3.0);
    int imageHeight = (int)((wHeight - VIEW_MARGIN * 4) / 3.0);
    int imageHeight1 = wHeight - imageHeight * 2 - VIEW_MARGIN * 4;

    // 下面产生三个 mitkImageView 对象安排在父窗口中
    // 并且给每个 mitkImageView 生成一个可显示的 mitkImageModel
    // mitkImageModel 的 SetViewMode()函数就是设定显示断面的方向的
    mitkImageView *tempView;
```

```
    mitkImageModel *tempModel;

    tempView = new mitkImageView;
    tempModel = new mitkImageModel;
    tempModel->SetViewMode(mitkImageModel::VIEW_XY);
    tempView->SetParent(GetSafeHwnd());
    tempView->SetLeft(VIEW_MARGIN);
    tempView->SetTop(VIEW_MARGIN);
    tempView->SetWidth(imageWidth);
    tempView->SetHeight(imageHeight1);
    tempView->SetBackColor(0,0,0);
    tempView->SetCrossArrow(false);
    tempView->SetManipulator(new ChooseViewManipulator(this));
    tempView->AddModel(tempModel);
    tempView->Show();
    m_ImageView[0] = tempView;
    m_ImageModel[0] = tempModel;

    tempView = new mitkImageView;
    tempModel = new mitkImageModel;
    tempModel->SetViewMode(mitkImageModel::VIEW_YZ);
    tempView->SetParent(GetSafeHwnd());
    tempView->SetLeft(VIEW_MARGIN);
    tempView->SetTop(imageHeight1 + VIEW_MARGIN * 2);
    tempView->SetWidth(imageWidth);
    tempView->SetHeight(imageHeight);
    tempView->SetBackColor(0,0,0);
    tempView->SetCrossArrow(false);
    tempView->SetManipulator(new ChooseViewManipulator(this));
    tempView->AddModel(tempModel);
    tempView->Show();
    m_ImageView[1] = tempView;
    m_ImageModel[1] = tempModel;

    tempView = new mitkImageView;
    tempModel = new mitkImageModel;
    tempModel->SetViewMode(mitkImageModel::VIEW_ZX);
    tempView->SetParent(GetSafeHwnd());
    tempView->SetLeft(VIEW_MARGIN);
    tempView->SetTop(imageHeight1 + imageHeight + VIEW_MARGIN * 3);
```

```
    tempView->SetWidth(imageWidth);
    tempView->SetHeight(imageHeight);
    tempView->SetBackColor(0,0,0);
    tempView->SetCrossArrow(false);
    tempView->SetManipulator(new ChooseViewManipulator(this));
    tempView->AddModel(tempModel);
    tempView->Show();
    m_ImageView[2] = tempView;
    m_ImageModel[2] = tempModel;

    // 初始情况第一个 mitkImageView 为选中状态
    this->ChangeSelectedImageView(m_ImageView[0]);

    // 下面生成左边显示三维模型的 mitkView
    m_SceneView = new mitkView;

    m_SceneView->SetParent(GetSafeHwnd());

    m_SceneView->SetLeft(imageWidth + VIEW_MARGIN * 2);
    m_SceneView->SetTop(VIEW_MARGIN);
    m_SceneView->SetWidth(wWidth - imageWidth - VIEW_MARGIN * 3);
    m_SceneView->SetHeight(wHeight - VIEW_MARGIN * 2);

    m_SceneView->SetBackColor(0, 0, 0);

    m_SceneView->Show();

    // 生成一个 mitkSurfaceModel
    m_SurfaceModel = new mitkSurfaceModel;

    // 设置表面材质属性（这些属性可以随时调整）
   m_SurfaceModel->GetProperty()->SetAmbientColor(0.75f, 0.75f, 0.75f,
1.0f);
    m_SurfaceModel->GetProperty()->SetDiffuseColor(1.0f, 0.57f, 0.04f,
1.0f);
    m_SurfaceModel->GetProperty()->SetSpecularColor(1.0f, 1.0f, 1.0f,
1.0f);
    m_SurfaceModel->GetProperty()->SetSpecularPower(100.0f);
    m_SurfaceModel->GetProperty()->SetEmissionColor(0.0f, 0.0f, 0.0f,
0.0f);
```

```
    // 将 Model 加入到 View 中
    m_SceneView->AddModel(m_SurfaceModel);

    return 0;
}

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

    // TODO: Add your message handler code here
    // 更新 mitkView 在父窗口中的位置和尺寸
    int imageWidth = (int)((cx - VIEW_MARGIN * 3) / 3.0);
    int imageHeight = (int)((cy - VIEW_MARGIN * 4) / 3.0);
    int imageHeight1 = cy - imageHeight * 2 - VIEW_MARGIN * 4;

    // 调整三个 mitkImageView 的位置和尺寸
    m_ImageView[0]->SetLeft(VIEW_MARGIN);
    m_ImageView[0]->SetTop(VIEW_MARGIN);
    m_ImageView[0]->SetWidth(imageWidth);
    m_ImageView[0]->SetHeight(imageHeight1);

    m_ImageView[1]->SetLeft(VIEW_MARGIN);
    m_ImageView[1]->SetTop(imageHeight1 + VIEW_MARGIN * 2);
    m_ImageView[1]->SetWidth(imageWidth);
    m_ImageView[1]->SetHeight(imageHeight);

    m_ImageView[2]->SetLeft(VIEW_MARGIN);
    m_ImageView[2]->SetTop(imageHeight1 + imageHeight + VIEW_MARGIN *
3);
    m_ImageView[2]->SetWidth(imageWidth);
    m_ImageView[2]->SetHeight(imageHeight);

    // 调整左边的 mitkView 的位置和尺寸
    m_SceneView->SetLeft(imageWidth + VIEW_MARGIN * 2);
    m_SceneView->SetTop(VIEW_MARGIN);
    m_SceneView->SetWidth(cx - imageWidth - VIEW_MARGIN * 3);
    m_SceneView->SetHeight(cy - VIEW_MARGIN * 2);
}
```

```
void CMITKTestView::OnSliceFirst()
{
    // TODO: Add your command handler code here
    if (m_CurImageModel)
    {
        // 定位在第 1 张切片
        m_CurImageModel->SetCurrentSliceNumber(0);

        // 更新 View 中的显示
        m_CurImageView->Update();
    }
}

void CMITKTestView::OnSliceLast()
{
    // TODO: Add your command handler code here
    if (m_CurImageModel)
    {
        // 定位到最后一张切片

    m_CurImageModel->SetCurrentSliceNumber(m_ImageModel[0]->GetTotalS
liceNumber()-1);

        // 更新 View 中的显示
        m_CurImageView->Update();
    }
}

void CMITKTestView::OnSliceNext()
{
    // TODO: Add your command handler code here
    if (m_CurImageModel)
    {
        // 定位到下一张切片
        m_CurImageModel->NextSlice();

        // 更新 View 中的显示
        m_CurImageView->Update();
    }
```

```
}

void CMITKTestView::OnSlicePrev()
{
    // TODO: Add your command handler code here
    if (m_CurImageModel)
    {
        // 定位到前一张切片
        m_CurImageModel->PrevSlice();

        // 更新 View 中的显示
        m_CurImageView->Update();
    }
}

void CMITKTestView::OnDestroy()
{
    CView::OnDestroy();

    // TODO: Add your message handler code here
    for (int i=0; i<3; ++i)
    {
        if (m_ImageView[i])
        {
            m_ImageView[i]->Delete();
            m_ImageView[i] = NULL;
        }
    }

    if (m_SceneView)
    {
        m_SceneView->Delete();
        m_SceneView = NULL;
    }
}

void CMITKTestView::OnPoints()
{
    // TODO: Add your command handler code here
    m_SurfaceModel->GetProperty()->SetRepresentationTypeToPoints();
```

```
    m_SceneView->Update();
}

void CMITKTestView::OnWireframe()
{
    // TODO: Add your command handler code here
    m_SurfaceModel->GetProperty()->SetRepresentationTypeToWireframe()
;
    m_SceneView->Update();
}

void CMITKTestView::OnSurface()
{
    // TODO: Add your command handler code here
    m_SurfaceModel->GetProperty()->SetRepresentationTypeToSurface();
    m_SceneView->Update();
}

void CMITKTestView::ChangeSelectedImageView(mitkImageView *selView)
{
    if (m_CurImageView == selView) return;

    if (m_CurImageView)
    {
        m_CurImageView->SetSelected(false);
        m_CurImageView->Update();
    }
    m_CurImageView = selView;

    if (m_CurImageView)
    {

        // 设置 m_CurImageView 为选中状态并更新 View
        m_CurImageView->SetSelected(true);
        m_CurImageView->Update();

        // 从当前选中的 mitkImageView 中得到当前的 mitkImageModel
        m_CurImageModel =
mitkImageModel::SafeDownCast(m_CurImageView->GetModel(0));
```

```
    }
    else m_CurImageModel = NULL;
}
```

注意到在上面的代码中还包含了其他一些功能，首先是以不同的方式显示（点显示、线框显示和面显示）三维模型的功能在OnPoints()、OnWireframe()和OnSurface()三个函数中，都是通过直接得到Surface Model的Property然后将其设置成所需要的显示方式而实现的，三维模型的表面材质也是通过这种方式设置。其次是对三个mitkImageView的选择功能，在上面的代码中切片的浏览都是针对m_CurImageView和m_CurImageModel即当前选中的mitkImageView进行的，从前面的图中也可以看到当前处于选中状态下的mitkImageView是有一个红色的方框包围的。要实现用鼠标对这三个mitkImageView进行选择的功能，还牵扯到另外一个类：ChooseViewManipulator，默认情况下，mitkImageView包含的Manipulator是mitkImageViewManipulatorStandard，它实现了三种基本的鼠标操作，这个在 9.2 中已经有所介绍，在这个例子中，我们不仅想保留这些基本功能，还要再提供鼠标任意键按下是对所在View的选择，所以ChooseViewManipulator从mitkImageViewManipulatorStandard继承，并重载它的OnMouseDown函数，加入新的功能，主要是通过调用CMITKTestView的ChangeSelectedImageView()函数进行具体的切换操作。 ChooseViewManipulator的具体代码如下：

```
// ChooseViewManipulator.h : interface of the ChooseViewManipulator class
//
/////////////////////////////////////////////////////////////////////////////

#ifndef __ChooseViewManipulator_h
#define __ChooseViewManipulator_h

#include "mitkImageViewManipulatorStandard.h"

class CMITKTestView;

class ChooseViewManipulator : public mitkImageViewManipulatorStandard
{
public:
    // 如果不需要 RTTI，这句可以省去
    MITK_TYPE(ChooseViewManipulator, mitkImageViewManipulatorStandard)
```

```
    ChooseViewManipulator(CMITKTestView *parentView);

    // 重载父类的 OnMouseDown()函数，加入选择所在 mitkImageView
    // 为当前选中 mitkImageView 的功能
    virtual void OnMouseDown(int mouseButton, bool ctrlDown, bool
shiftDown, int xPos, int yPos);

protected:
    virtual ~ChooseViewManipulator();

    // 保存指向父窗口 View 的指针，以便在 OnMouseDown()触发时
    // 调用其 ChangeSelectedImageView()函数改变当前选中的
    // mitkImageView
    CMITKTestView *m_PView;

private:
};

#endif


// ChooseViewManipulator.cpp : implementation of the
// ChooseViewManipulator class
//

#include "stdafx.h"
#include "MITKTest.h"

#include "MITKTestView.h"
#include "ChooseViewManipulator.h"
#include "mitkImageView.h"

ChooseViewManipulator::ChooseViewManipulator(CMITKTestView
*parentView)
: m_PView(parentView)
{
}

ChooseViewManipulator::~ChooseViewManipulator()
```

```
{
}

void ChooseViewManipulator::OnMouseDown(int mouseButton, bool ctrlDown,
bool shiftDown, int xPos, int yPos)
{
    // 调用父类的函数保留原始功能
    mitkImageViewManipulatorStandard::OnMouseDown(mouseButton,
ctrlDown, shiftDown, xPos, yPos);

    // 添加选择所在 mitkImageView 为当前选定 mitkImageView 的功能
    // m_View 是其父类中的成员，保存了指向该 Manipulator 所在的 View 的指针
    if (m_PView)
    m_PView->ChangeSelectedImageView(mitkImageView::SafeDownCast(m_Vi
    ew));
}
```

有了这个类，在生成每一个 mitkImageView 的时候通过如上面的代码中那样调用 tempView->SetManipulator(new ChooseViewManipulator(this)) 将缺省的 Manipulator 换掉，这样就在保留标准 Manipulator 功能的基础上加入了自定义的功能。

接下来，我们再来看一下如何在表面重建时显示其进度。

这一功能的实现主要是依靠 MITK 提供的 Observer。mitkOberver 类是一个高度抽象的类，它只提供了一个 Update()接口，MITK 中所有自 mitkObject 基础下来的类均有添加一组 mitkObserver 的功能，通过 AddObserver()、RemoveObserver()管理其所包含的 mitkObserver 队列。在程序运行过程中，通过调用 mitkObserver 提供的 Update()接口通知具体的 Observer 更新状态。我们所要做的就是从 mitkObserver 派生出具体的 Observer，通过实现 Update()接口，在界面上更新所观察对象的状态。在本节的例子中，我们为 MITK 中的 mitkFilter 做了一个 Observer，mitkFilter 继承自 mitkProcessObject，它已经实现了一些记录当前程序的运行进度的基本功能，在我们的 Observer 中通过调用 GetProgressRate()就可以得到当前程序的运行进度。需要注意的是在开始之前，必须通过 SetProgressRateMax()设定用来度量进度的整数的最大值，在程序运行的整个过程中，进度值将从 0 增大到这个设定的最大值。以下是 FilterObserver 的具体实

现：

```
// FilterObserver.h : interface of the FilterObserver class
//
/////////////////////////////////////////////////////////////////////////////

#ifndef __FilterObserver_h
#define __FilterObserver_h

#include "mitkObserver.h"

class mitkFilter;
class CProgressDlg;
class FilterObserver : public mitkObserver
{
public:
    FilterObserver(mitkFilter *obj);

    // 从父类继承的必须实现的虚函数
    // MITK 中的对象就通过这个接口通知 Observer 更新状态
    virtual void Update();

    // 在这个函数中完成一些必要的初始化工作
    void StartShowProgress(const char *title);

    // 结束进度显示，隐藏对话框
    void FinishShowProgress();

protected:
    virtual ~FilterObserver();

    mitkFilter *m_Object; // 指向观察对象的指针
    CProgressDlg *m_ProgressDlg; // 指向显示进度的对话框的指针

private:
};

#endif
```

```
// FilterObserver.cpp : implementation of the FilterObserver class
//

#include "stdafx.h"
#include "MITKTest.h"

#include "FilterObserver.h"
#include "mitkFilter.h"
#include "ProgressDlg.h"

FilterObserver::FilterObserver(mitkFilter *obj) : m_Object(obj)
{
    m_ProgressDlg = NULL;
}

FilterObserver::~FilterObserver()
{
    if (m_ProgressDlg)
    {
        delete m_ProgressDlg;
    }
}

void FilterObserver::StartShowProgress(const char *title)
{
    if (m_Object == NULL) return;
    m_Object->SetProgressRateMax(1000);

    // 产生显示进度的对话框
    if (m_ProgressDlg == NULL)
    {
        m_ProgressDlg = new CProgressDlg;
        m_ProgressDlg->Create(CProgressDlg::IDD);
    }

    // 以下代码将显示进度的对话框定位在主窗口的中心
    CRect mfRect;
    CRect dlgRect;

    ::AfxGetApp()->GetMainWnd()->GetWindowRect(&mfRect);
```

```
    m_ProgressDlg->GetWindowRect(dlgRect);

    int newLeft = (mfRect.left + mfRect.right) / 2 - dlgRect.Width() /
2;
    int newTop = (mfRect.top + mfRect.bottom) / 2 - dlgRect.Height() /
2;

    m_ProgressDlg->MoveWindow(newLeft, newTop, dlgRect.Width(),
dlgRect.Height());

    m_ProgressDlg->ShowWindow(SW_SHOW);

    // 一些相关参数的初始化
    // 注意：进度条的最大值跟 m_Object->SetProgressRateMax()的设置需保持一致
    m_ProgressDlg->SetMaxProgress(1000);
    m_ProgressDlg->SetProgressStep(1);
    m_ProgressDlg->UpdateProgress(0);
    m_ProgressDlg->SetLabelText(title);
}

void FilterObserver::FinishShowProgress()
{
    if (m_ProgressDlg)
    {
        m_ProgressDlg->ShowWindow(SW_HIDE);
    }
}

void FilterObserver::Update()
{
    if (m_Object && m_ProgressDlg)
    {
        // 更新进度条的显示
        m_ProgressDlg->UpdateProgress(m_Object->GetProgressRate());
    }
}
```

FilterObserver 将显示进度的任务交给对话框 CProgressDlg 去做，该对话框的样子如所示。具体怎么制作这个对话框就不细讲了，上面用到的几个函数如

下（均为 inline 函数）：

```
#if !defined(AFX_PROGRESSDLG_H__02135212_7541_402A_9E3D_F62AF713A692__INCLUDED_)
#define AFX_PROGRESSDLG_H__02135212_7541_402A_9E3D_F62AF713A692__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// ProgressDlg.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CProgressDlg dialog

class CProgressDlg : public CDialog
{
// Construction
public:
    CProgressDlg(CWnd* pParent = NULL);   // standard constructor

    // 更新标签中显示的文本
    void SetLabelText(LPCSTR labelText)
    { m_InfoLabel.SetWindowText(labelText); }

    // 设置进度条的最大值
    void SetMaxProgress(int maxProgress)
    { m_ProgressCtrl.SetRange32(0, maxProgress); }

    // 设置进度条的步进值
    void SetProgressStep(int step)
    { m_ProgressCtrl.SetStep(step); }

    // 更新进度条位置
    void UpdateProgress(int pos)
    { m_ProgressCtrl.SetPos(pos); }

// Dialog Data
    //{{AFX_DATA(CProgressDlg)
```

```
    enum { IDD = IDD_PROGRESS_DIALOG };
    CProgressCtrl m_ProgressCtrl;
    CStatic    m_InfoLabel;
    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CProgressDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CProgressDlg)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

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

#endif
// !defined(AFX_PROGRESSDLG_H__02135212_7541_402A_9E3D_F62AF713A692_
_INCLUDED_)
```

在 MITKTestDoc.cpp 中添加代码，为重建算法附加一个显示进度的 Observer：

```
void CMITKTestDoc::OnMc()
{
    // TODO: Add your command handler code here
    // 产生阈值设置对话框
    CThresholdDlg dlg;
```

```
// 显示阈值设置对话框
if (dlg.DoModal() == IDOK)
{
    // 生成一个 mitkMarchingCubes 对象
    mitkMarchingCubes *mc = new mitkMarchingCubes;

    // 生成一个以 mc 为观察对象的 FilterObserver
    FilterObserver *observer = new FilterObserver(mc);

    // 将生成的 FilterObserver 加入到 mc 中
    mc->AddObserver(observer);

    // 将从对话框中得到的阈值设置给 Marching Cubes 算法
    mc->SetThreshold(dlg.m_LowValue, dlg.m_HighValue);

    // 设置输入数据
    mc->SetInput(m_Volume);

    // 初始化 Observer
    observer->StartShowProgress("表面重建...");

    // 运行算法
    bool r = mc->Run();

    // 结束进度的显示
    observer->FinishShowProgress();

    // 如果算法正常结束则更新 Mesh 数据并显示
    if (r)
    {
        this->clearMesh();

        m_Mesh = mc->GetOutput();
        m_Mesh->AddReference();

        UpdateAllViews(NULL);
    }

    // 删除 mitkMarchingCubes 对象
    // 注意：不用删除 observer，它将由 mc 来删除
```

```
        mc->Delete();
    }
}
```

显示 Volume 和 Mesh 相关信息的代码也在 MITKTestDoc.cpp 中：

```
void CMITKTestDoc::OnVolumeinfo()
{
    // TODO: Add your command handler code here
    if (!m_Volume) return;

    CString msg;
   msg.Format("Width:\t%d\nHeight:\t%d\nSlices:\t%d\n\nChannel Number:
%d\nBits per Channel: %d\n\nX Spacing: %.3fmm\nY Spacing: %.3fmm\nZ
Spacing: %.3fmm\t",
              m_Volume->GetWidth(),
              m_Volume->GetHeight(),
              m_Volume->GetImageNum(),
              m_Volume->GetNumberOfChannel(),
              m_Volume->GetDataTypeSize()*8,
              m_Volume->GetSpacingX(),
              m_Volume->GetSpacingY(),
              m_Volume->GetSpacingZ());

    ::AfxMessageBox(msg, MB_OK|MB_ICONINFORMATION);
}

void CMITKTestDoc::OnSurfaceinfo()
{
    // TODO: Add your command handler code here
    if (!m_Mesh) return;

    CString msg;
    msg.Format("Vertices:\t%d\nTriangles:\t%d\t",
              m_Mesh->GetVertexNumber(),
              m_Mesh->GetFaceNumber());

    ::AfxMessageBox(msg, MB_OK|MB_ICONINFORMATION);
}
```

至此，几个主要的功能均已介绍完，实际上，本节中的例子还有很多地方可

以改进，比如可以增加几个对话框来调节三维模型的表面材质、在 View 中添加一些 MITK 提供的 Widgets 来丰富交互功能等等，甚至可以像上面提到的 ChooseViewManipulator 和 FilterObserver 那样扩展 MITK 实现自定义的功能，限于篇幅，这些内容不能一一介绍了。读者可以自己动手实践一下，用 MITK 来开发自己的软件项目。另外，第十一章所介绍的 3DMed 就是基于 MITK 开发的一套规模比较大的软件，读者可以以此作为参考。

## 9.5 小结

本章介绍了 MITK 开发实际项目的例子，通过开发环境的设置，二维功能的图像浏览器，再到用 MITK 进行表面重建，最后给出了一个完善的例子，其目的是让读者能够按照本章的引导，一步步熟悉 MITK，使得 MITK 能够帮助读者更好的理解 MITK，更好的使用 MITK 开发自己的应用程序。

# 10 扩充 MITK 功能

MITK 本身是一个开放的架构，允许用户自己扩充其功能，以满足项目开发中的需求。在 MITK 中，可以扩充的部分包括 Filter、Reader 和 Writer，通过扩充 Filter，可以添加自己的处理算法；通过扩充 Reader，可以添加自己的文件格式的导入功能；通过扩充 Writer，可以添加自己的文件格式的保存功能。

要扩充这几个部分，必须在实现自己的算法的时候遵循一些 MITK 规定的接口。本章首先介绍这些接口，然后以两个实际的例子来讲述如何扩充 MITK 的功能，这两个例子一个是如何扩充 Reader 的功能，另外一个是如何扩充 Filter 的功能，读者在阅读这两个例子之后，可以对扩充 MITK 功能有个充分的了解，并可以在自己的项目中来使用。和上一章的风格一样，这两个例子也使用 Microsoft Visual C++ 6.0 来作为编程环境，并且一步一步地加以说明。

## 10.1 扩充 MITK 功能的预备知识

在 MITK 中，是通过类的继承层次来实现不同的功能的，其中在一些上层的父类里面规定了一些子类必须实现的接口，子类来实现这些接口从而完成具体的功能，这是整个的大的原则。

在第二章中介绍了MITK的整体框架，整个MITK是由数据和算法来构成的，而算法又是由三个不同的种类：Source、Filter和Target来构成的，其在MITK的整个类层次结构中的位置如 图 10-1 所示。

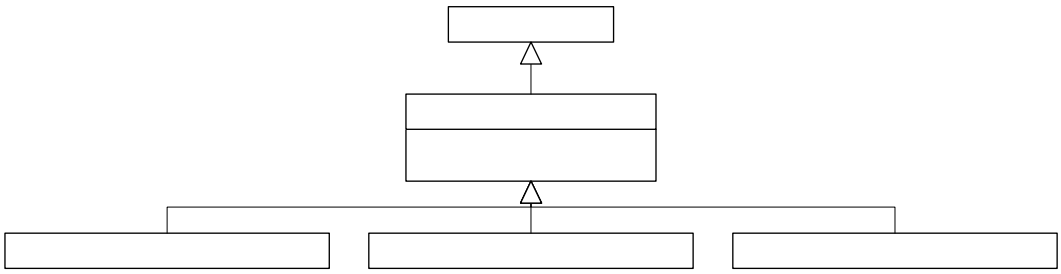


图 10-1 MITK 中算法类的上层结构

其中，在它们共同的父类 ProcessObject 中，规定了一个公有的接口 bool Run

（），这就意味着所有的 MITK 的算法类都因之而继承了这一接口，所以在最终用户使用某个算法时，可以直接调用其 Run 成员函数来进行算法的计算。ProcessObject 为了能使其子类实现不同的 Run 的行为，从而实现不同的算法，在 Run 函数里面采用了 Template Method 的设计模式，调用一个保护的虚函数 Execute，使其子类来重载 Execute 函数来完成实际的工作，Run 函数的一部分代码如下所示：

```
bool mitkProcessObject::Run()
{
    … …;
    return (this->Execute());
}
```

对于Reader来说，它属于Source的一种，主要任务是完成各种格式文件的读入，并转换为MITK的内部的统一的格式。它是一个比较上层的抽象基类，其在整个MITK的类层次中的位置如 图 10-2 所示

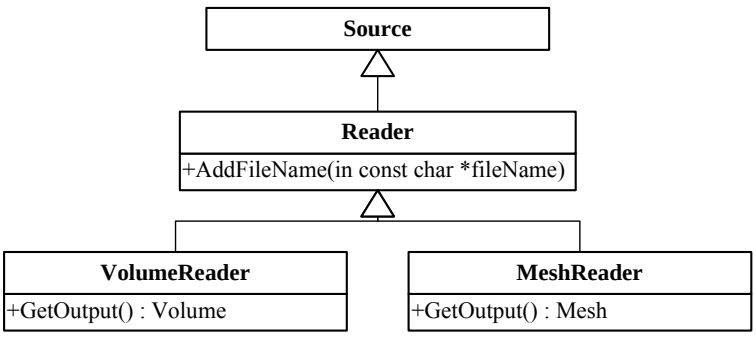



图 10-2 Reader 的类层次结构

Reader 内部维护着一个文件名的列表（因为有的格式的文件只需要一个，而有的格式的文件需要多个），可以使用其 AddFilename 成员函数来将要读入的文件的文件名传进来。考虑到 MITK 的数据分为两种：Volume 和 Mesh，所以 Reader 也分为两种：VolumeReader 和 MeshReader。VolumeReader 主要读入一个

三维的数据集，不同格式的支持由其子类来完成，目前 MITK 里面提供了对 DICOM、IM0、RAW、BMP、JPEG、TIFF 文件格式的支持，分别由 mitkDICOMReader、mitkIM0Reader、mitkRAWReader、mitkBMPReader、mitkJPEGReader 和 mitkTIFFReader 来完成；而 MeshReader 主要读入三维网格数据，不同格式的支持由其子类来完成，目前 MITK 里面提供了对 Wavefront OBJ、快速成型机 STL 文件格式的支持，分别由 mitkOBJReader、mitkSTLReader 来完成。

对于Writer来说，它属于Target的一种，主要任务是完成各种格式文件的磁盘写入，它是一个比较上层的抽象基类，其在整个MITK的类层次中的位置如 图 10-3 所示：

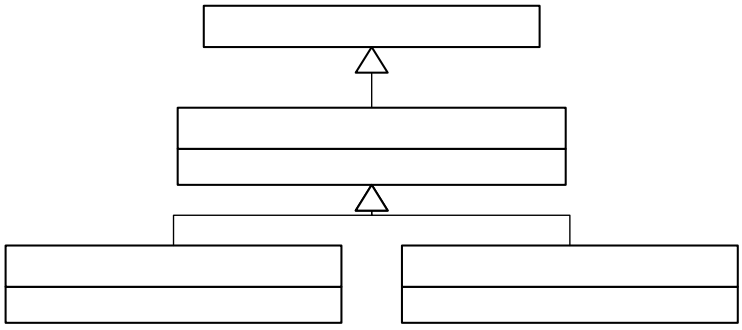


图 10-3 Writer 的类层次结构

如 Reader 一样，Writer 内部也维护着一个文件名的列表，可以使用其 AddFilename 成员函数来将要读入的文件的文件名传进来。考虑到 MITK 的数据分为两种：Volume 和 Mesh，所以 Writer 也分为两种：VolumeWriter 和 MeshWriter。VolumeWriter 主要将内存中的一个 Volume 写入磁盘，不同格式的支持由其子类来完成，目前 MITK 里面提供了对 DICOM、IM0、RAW、BMP、JPEG、TIFF 文件格式的支持，分别由 mitkDICOMWriter、mitkIM0Writer、mitkRAWWriter、mitkBMPWriter、mitkJPEGWriter 和 mitkTIFFWriter 来完成；而 MeshWriter 主要是将内存中的一个 Mesh 写入磁盘，不同格式的支持由其子类来完成，目前 MITK 里面提供了对 Wavefront OBJ、快速成型机 STL 文件格式的支持，分别由

mitkOBJWriter、mitkSTLWriter 来完成。

至于Filter，在 2.2.3 节已经有比较详细的介绍，这里就不再赘述了。

## 10.2 实例之一：扩充 Reader 功能

好了，了解了上面的预备知识之后，下面本节以一个实际的例子来演示如何在自己的项目里面来扩充 MITK 的 Reader 所支持的格式，风格上和第九章的例子一样，通过一步一步的深入浅出的讲解，加上程序代码的辅助，来力图把概念讲解清楚。

### 10.2.1 扩充 Reader 功能的一般步骤

要读取一种 MITK 不支持的文件格式，可以通过扩充 Reader 的类层次结构来实现。扩充 Reader 功能的一般步骤如下：

第一步，先判断自己要读取的文件的内容是一个三维的数据集还是三维的面片网格，比如一系列的 BMP 文件就是一个三维的数据集，而 OBJ 和 STL 等文件就是三维的面片网格；

第二步，写自己的文件格式的 Reader，如果要读取的文件是三维数据集，那么选择从 VolumeReader 继承，如果要读取的文件是三维网格，那么选择从 MeshReader 继承；

第三步，实现虚函数 bool Execute()，在这个函数里面完成文件的实际读入工作，并将数据填入 Volume 或者 Mesh 中。

### 10.2.2 实例程序的功能

下面我们要完成的实例程序，将要读入一种我们自己定义的格式的文件，其数据内容为三维数据集，这种文件有一个文件头，记录了一些文件大小、像素尺寸等信息，然后是实际数据。其具体文件格式如 图 10-4 所示。

<table>
<tr><td rowspan="6">文件头</td><td>图像宽度</td><td>4 字节，整型</td></tr>
<tr><td>图像高度</td><td>4 字节，整型</td></tr>
<tr><td>图像张数</td><td>4 字节，整型</td></tr>
<tr><td>像素间距（x 方向）</td><td>4 字节，浮点型</td></tr>
<tr><td>像素间距（y 方向）</td><td>4 字节，浮点型</td></tr>
<tr><td>像素间距（z 方向）</td><td>4 字节，浮点型</td></tr>
<tr><td colspan="3">实际数据，每个元素为<br>单精度浮点型，顺序存放</td></tr>
</table>

图 10-4 文件格式

例子程序的功能很简单，就是打开上面格式的文件并将其读入内存中，其“文件”菜单里面有一项“打开自定义文件”，点击此菜单项后将会弹出一个“打开自定义格式文件”对话框，如 图 10-5 所示。选中要打开的文件以后，例子程序会调用我们自己扩充的Reader，解析文件并将其读入内存，为了简化、清晰起见，这个例子并没有显示读进来的数据，侧重点在如何扩充Reader的功能上，请参考 8 的例子去实现数据的显示等功能。


图 10-5 打开文件对话框

### 10.2.3 实例程序的制作

还是和第九章一样，我们通过具体的例子一步一步地展现如何扩充 MITK 中 Reader 的功能。

第一步，是在 Microsoft Visual C++ 6.0 的 IDE 开发环境中新建一个 MFC 工程，单文档界面的，工程名字叫做 MitkEnhancement，这一步的操作过程这里就不再赘述了。

第二步，修改菜单，删除不必要的菜单项，只留下“文件”、“查看”、“帮助”三个菜单项，在“文件”菜单里面删除原先的子菜单，加入“打开自定义文件”子菜单，修改后的菜单如 图 10-6 所示。

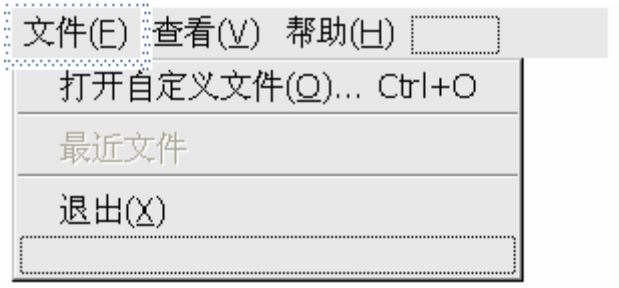



图 10-6 修改后的菜单

第三步，到了添加自己的Reader的时候了，用“New Class”生成一个我们自己的类，名字叫做CMyMitkReader，如 图 10-7 所示。然后遵循 10.2.1 节中所述的一般规则，因为我们要打开的文件是一个三维数据文件，所以应该从VolumeReader继承，之后实现Execute接口。

CMyMitkReader 的声明在 MyMitkReader.h 文件中，其代码如下所示：

```
class CMyMitkReader  : public mitkVolumeReader
{
public:
    CMyMitkReader();
    virtual ~CMyMitkReader();

protected:
    virtual bool Execute();

};
```

可以从代码中看出，CMyMitkReader 公有继承了 mitkVolumeReader，并重载了其 Protected 的虚函数 Execute，当然在前面应该加上包含 mitkVolumeReader 的预处理指令：

```
#include "mitkVolumeReader.h"
```

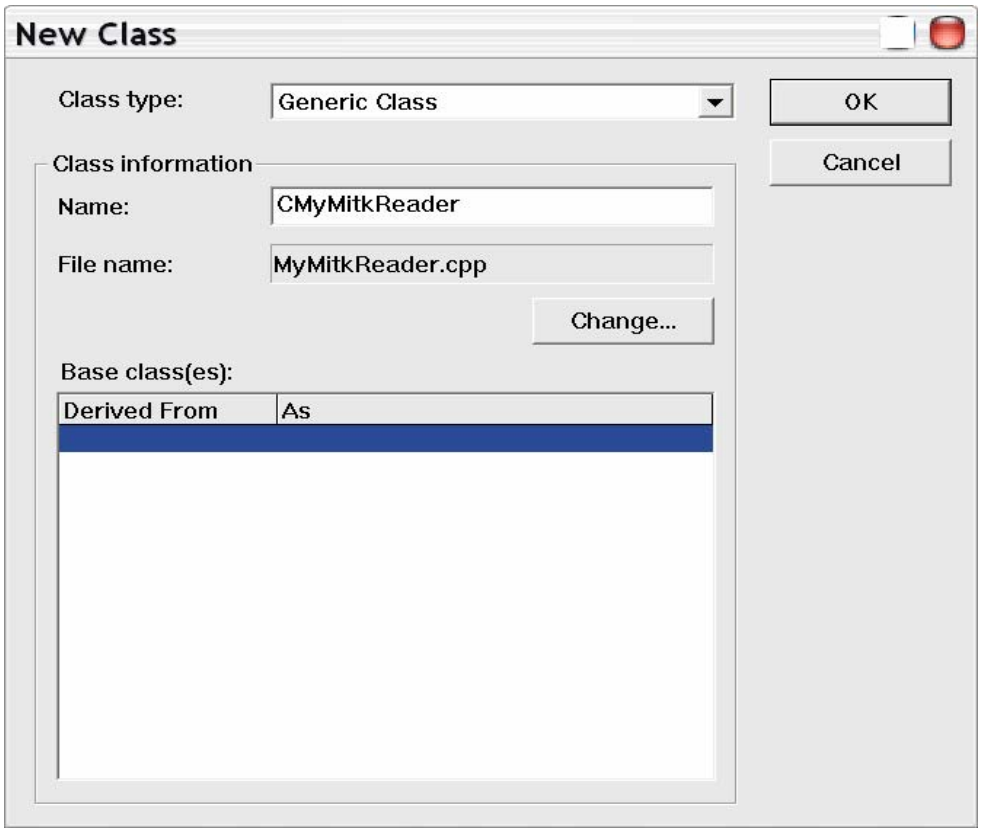



图 10-7 新建 CMyMitkReader 类

CMyMitkReader 的实现代码在 MyMitkReader.cpp 文件中，对于它来说，最重要的就是函数 Execute 的实现，在这个函数内必须实现解析文件格式、读数据至 Volume 中等任务。为了解析文件格式，首先我们定义一个结构来容纳文件头，其代码如下所示：

```
struct FILEHEADER
{
    int m_ImgWidth;
    int m_ImgHeight;
    int m_ImgNum;
    float m_SpacingX;
    float m_SpacingY;
    float m_SpacingZ;
};
```

这个结构对应着 图 10-4 文件格式中的文件头信息，有了它以后，就可以实现Execute函数了，其代码如下所示：

```
bool CMyMitkReader::Execute()
{
    int fileCount = this->_getFileCount();
    if (fileCount <= 0)
    {
           AfxMessageBox("尚没有添加文件名");
           return false;
    }

    //得到 Reader 的输出 Volume 的指针，注意在 GetOutput
    //里面已经创建了 Volume.
    mitkVolume *outputVolume = this->GetOutput();

    //得到文件名，因为要读入的是三维文件，故只需一个文件名即可.
    const char *fileName = this->_getFileName(0);

    //打开文件，并读取文件头信息.
    FILE *inputFile = fopen(fileName, "rb");
    if(inputFile == NULL)
    {
        AfxMessageBox("不能打开文件");
        return false;
    }
    FILEHEADER fileHeader;
    fread(&fileHeader, sizeof(FILEHEADER), 1, inputFile);

    //写输出 Volume 的必要信息.
    outputVolume->SetWidth(fileHeader.m_ImgWidth);
    outputVolume->SetHeight(fileHeader.m_ImgHeight);
    outputVolume->SetImageNum(fileHeader.m_ImgNum);
    outputVolume->SetSpacingX(fileHeader.m_SpacingX);
    outputVolume->SetSpacingY(fileHeader.m_SpacingY);
    outputVolume->SetSpacingZ(fileHeader.m_SpacingZ);
    outputVolume->SetDataTypeToFloat();
    outputVolume->SetNumberOfChannel(1);
```

```
    //分配必要的内存.
    void *volMemory = outputVolume->Allocate();
    if(volMemory == NULL)
    {
        AfxMessageBox("内存不足");
        return false;
    }

    //得到 Volume 所占用的内存的大小
    int volSize = outputVolume->GetActualMemorySize();

    //将实际数据读入 Volume 中.
    fread(volMemory, 1, volSize, inputFile);

    //关闭文件并返回.
    fclose(inputFile);
    return true;
}
```

上面的代码首先得到此 Reader 的文件名，然后打开文件并读入文件头信息，之后设置 Volume 的各项属性，分配 Volume 的内部数据缓冲，最后将数据读入 Volume。大部分代码都加了注释，整个过程也并不是很复杂，这里要解释的就是 Reader 本身的一些 API，它提供了两个保护的函数 _getFileCount 和 _getFileName 给派生类使用，来访问 Reader 内部的文件名列表。对于从 VolumeReader 派生下来的子类，还继承了 GetOutput 函数，其内部在第一次被访问时生成一个 Volume 的指针，并将 Volume 的各项属性设置为初始值，实际数据缓冲也没有分配，在其后的调用中将直接返回这个 Volume 的指针。因此对于我们的 CMyMitkReader 来说，可以直接通过 GetOutput 函数拿到输出的 Volume，然后填充其各项属性，分配实际数据内存并填充数据，这样就完成了一个 Reader 的功能。

第四步，完成了 CMyMitkReader 以后，剩下的工作就很容易了，在文档类 CMitkEnhancementDoc 里面添加成员变量

```
mitkVolume *m_Volume;
```

以记录实际数据，并添加处理函数 OnFileOpenCustom，响应菜单“打开自定义文件”的单击事件，其代码如下所示：

```
void CMitkEnhancementDoc::OnFileOpenCustom()
{
    // 显示打开文件对话框.
    COpenFileDialog fileDialog;
    fileDialog.SetTitle("打开自定义格式文件");
    fileDialog.SetFilter("自定义格式(*.*)\0*.*\0\0");

    // 如果用户选择了一个文件并点“确定”按钮.
    if(fileDialog.Run())
    {
        int nFilterIndex = fileDialog.GetFilterIndex();
        CString szFileName = fileDialog.GetPathName();

        // 清除掉旧的 Volume.
        this->clearVolume();

        //生成一个我们自己的 Reader 对象.
        CMyMitkReader *myReader = new CMyMitkReader;

        //设置文件名.
        myReader->AddFileName(fileDialog.GetFileName());

        //运行我们自己的 Reader，完成读入过程.
        myReader->Run();

        //得到 Reader 读出来的 Volume.
        m_Volume = myReader->GetOutput();
        m_Volume->AddReference();

        //删除 Reader.
        myReader->Delete();
    }
}
```

在这一部分代码里面，我们首先弹出一个打开文件对话框，让用户选择要打开的文件名，然后生成我们扩充的 CMyMitkReader 的对象，设置文件名，并调用其 Run 函数以实际执行我们的 Reader 的功能，最后删除掉此 Reader。在这里需要说明的是 clearVolume 函数，它的功能是将上一次读入的 Volume 释放掉，其代码如下所示：

```
void CMitkEnhancementDoc::clearVolume()
{
    if (m_Volume)
    {
        m_Volume->RemoveReference();
        m_Volume = NULL;
    }
}
```

剩下的事情就是修改 CMitkEnhancementDoc 的构造和析构函数，完成必要的初始化和善后工作了，其代码如下所示：

```
CMitkEnhancementDoc::CMitkEnhancementDoc()
{
    m_Volume = NULL;
}

CMitkEnhancementDoc::~CMitkEnhancementDoc()
{
    this->clearVolume();
}
```

至此，扩充 Reader 功能的实例程序已经基本完成，你可以结合第九章的例子来增添这个实例程序的功能，对读入的数据进行显示、处理等操作。对于 Writer 的扩充，在概念上和步骤上与扩充 Reader 是相同的，这里不再赘述，读者可以根据本节的内容，自己进行实验。

## 10.3实例之二：扩充 Filter 功能

本节接着上一节的例子，在里面增加格式转换的功能，来演示如何在自己的项目里面来扩充 MITK 的 Filter 的功能，也就是如何扩充新的算法。掌握了这一节的内容以后，你就可以将自己的算法集成到 MITK 中去，并且可以和 MITK 中已有的同类算法作比较，甚至可以进行算法的性能评价等工作。

### 10.3.1 扩充 Filter 功能的一般步骤

扩充 Filter 功能的步骤和扩充 Reader 的步骤相似，也是通过扩充 Filter 的类层次结构来实现的。扩充 Filter 功能的一般步骤如下：

第一步，先判断自己要实现的算法的种类，这个可以从算法的输入与输出数据类型来判断：如果输入和输出都是 Volume 类型的，那么此算法属于 VolumeToVolumeFilter，如果输入是 Volume 类型的，而输出是 Mesh 类型的，那么此算法属于 VolumeToMeshFilter，其余的依此类推；

第二步，写自己的算法的 Filter，如果自己的算法是属于 VolumeToVolumeFilter，则从 VolumeToVolumeFilter 公有继承，如果自己的算法是属于 VolumeToMeshFilter，则从 VolumeToMeshFilter 公有继承；

第三步，实现虚函数 bool Execute()，在这个函数里面完成算法的实际过程，也就是对输入数据的处理，生成输出数据。

### 10.3.2 实例程序的功能

下面我们接着 10.2 节所完成的实例程序，继续在其上增加功能，可以把读进来的数据的格式进行转换。当运行例子程序时，将会发现在菜单栏上多了“格式转换”菜单及子菜单，首先通过“文件”菜单项下面的 “打开自定义文件”打开一个文件后，我们可以点击“格式转换”下面的“转换成Double”将读入的数据的格式转换成双精度浮点（Double）型，而点击“转换成Short”将会把读入的数据的格式转换成短整型（Short）。同样，为了简化、清晰起见，这个例子并没有显示格式转换后的数据，也没有对其进行进一步的处理，这里的侧重点放在如何扩充Filter的功能上，请参考第九章的例子去实现数据的显示、处理等功能。

### 10.3.3 实例程序的制作

和上一个例子一样，我们通过具体的例子一步一步地展现如何扩充 MITK 中 Filter 的功能。

第一步，利用上一个例子程序的工程文件 MitkEnhancement，直接在 Microsoft Visual C++集成环境中打开它。

第二步，修改菜单，在菜单栏上增加“格式转换”菜单项，并且在其下增加两个子菜单：“转换成Double”和“转换成Short”，修改后的菜单如 图 10-8 所示。

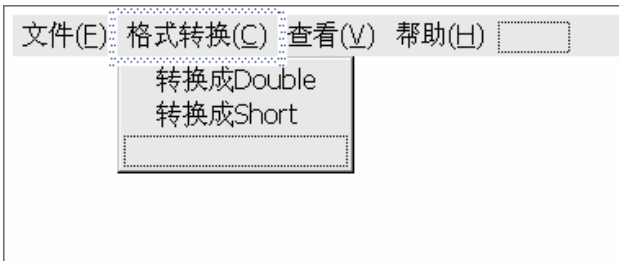



图 10-8 格式转换菜单

第三步，到了添加自己的Filter的时候了，用“New Class”生成一个我们自己的类，名字叫做CMyMitkFilter，如 图 10-9 所示。然后遵循 10.3.1 节中所述的一般规则，因为我们要实现的算法输入的数据是一个Volume，输出的数据是另外一个数据类型不同的Volume，所以应该从VolumeToVolumeFilter继承，之后实现Execute接口。

CMyMitkFilter 的声明在 MyMitkFilter.h 文件中，其代码如下所示：

```
class CMyMitkFilter : public mitkVolumeToVolumeFilter
{
public:
    CMyMitkFilter();
    virtual ~CMyMitkFilter();
```

```
    void ConvertToUnsignedChar();
    void ConvertToChar();
    void ConvertToToUnsignedShort();
    void ConvertToShort();
    void ConvertToUnsignedInt();
    void ConvertToInt();
    void ConvertToUnsignedLong();
    void ConvertToLong();
    void ConvertToFloat();
    void ConvertToDouble();


protected:
    virtual bool Execute();

private:
    int m_DataType;

};
```

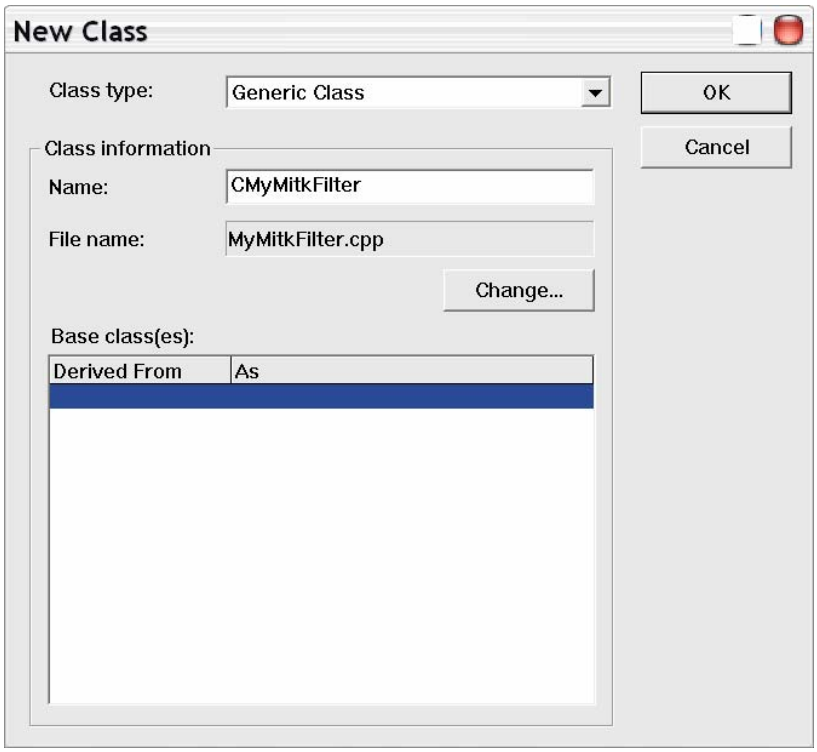



图 10-9 新建 CMyMitkFilter 类

可以从代码中看出，CMyMitkFilter 公有继承了 mitkVolumeToVolumeFilter，并重载了其 Protected 的虚函数 Execute，当然在前面应该加上包含 mitkVolumeToVolumeFilter 的预处理指令：

```
#include "mitkVolumeToVolumeFilter.h"
```

另外，在 CMyMitkFilter 中定义的一系列的 ConvertTo*** 函数，是用来设置转换后的 Volume 的数据类型的；并且 CMyMitkFilter 类还有一个私有的数据成员 m_DataType，用来记录转换后的 Volume 的数据类型，实际上 ConvertTo*** 函数就是操纵的 m_DataType，用其来反应当前的状态。

CMyMitkFilter 的实现代码在 MyMitkFilter.cpp 文件中，让我们先来看一下最简单的 ConvertTo*** 函数的实现，它们只是简单地将 m_DataType 变量设置成目标格式，其代码如下所示：

```
void CMyMitkFilter::ConvertToUnsignedChar()
{
    m_DataType = MITK_UNSIGNED_CHAR;
}

void CMyMitkFilter::ConvertToChar()
{
    m_DataType = MITK_CHAR;
}

void CMyMitkFilter::ConvertToToUnsignedShort()
{
    m_DataType = MITK_UNSIGNED_SHORT;
}

void CMyMitkFilter::ConvertToShort()
{
    m_DataType = MITK_SHORT;
}

void CMyMitkFilter::ConvertToUnsignedInt()
```

```
{
    m_DataType = MITK_UNSIGNED_INT;
}

void CMyMitkFilter::ConvertToInt()
{
    m_DataType = MITK_INT;
}

void CMyMitkFilter::ConvertToUnsignedLong()
{
    m_DataType = MITK_UNSIGNED_LONG;
}

void CMyMitkFilter::ConvertToLong()
{
    m_DataType = MITK_LONG;
}

void CMyMitkFilter::ConvertToFloat()
{
    m_DataType = MITK_FLOAT;
}

void CMyMitkFilter::ConvertToDouble()
{
    m_DataType = MITK_DOUBLE;
}
```

在这些代码中， MITK_*** 这些常量是在 MITK 里面预先定义好的，代表不同的数据类型，它们直接对应着 C 或者 C++ 语言里面的标准数据类型。

对 CMyMitkFilter 来说，最重要的就是函数 Execute 的实现，在这个函数内必须实现处理输入 Volume，将其格式转换为 m_DataType 所指示的目标格式，最后写至输出 Volume 中等一系列任务。为了完成例子程序的功能，我们本来只需限制 m_DataType 为 MITK_DOUBLE 或者 MITK_SHORT 两者之一，但是为了演示 MITK 的功能，这里允许 m_DataType 可以为任何数据类型，这样就

导致了比较复杂的程序实现，不过从这个例子中可以学到很多 MITK 内部的细节，也可以加深对 MITK 的了解。Execute 函数的整个代码如下所示：

```
bool CMyMitkFilter::Execute()
{
    mitkVolume *inVolume = this->GetInput();

    //判断输入数据是否为空.
    if(inVolume == NULL)
    {
        AfxMessageBox("没有设置输入数据");
        return false;
    }
    unsigned char *inData = (unsigned char *)inVolume->GetData();
    if(inData == NULL)
    {
        AfxMessageBox("输入 Volume 数据为空");
        return false;
    }

    //得到输出 Volume 的指针.
    mitkVolume *outVolume = this->GetOutput();

    //设置输出 Volume 的必要的属性,
    //大部分与输入 Volume 的相同.
    outVolume->SetWidth(inVolume->GetWidth());
    outVolume->SetHeight(inVolume->GetHeight());
    outVolume->SetImageNum(inVolume->GetImageNum());
    outVolume->SetNumberOfChannel(inVolume->GetNumberOfChannel());
    outVolume->SetSpacingX(inVolume->GetSpacingX());
    outVolume->SetSpacingY(inVolume->GetSpacingY());
    outVolume->SetSpacingZ(inVolume->GetSpacingZ());

    //设置输出 Volume 的数据类型.
    outVolume->SetDataType(m_DataType);

    //为实际的数据分配内存.
    unsigned char *outData = NULL;
```

```
if((outData = (unsigned char *)outVolume->Allocate()) == NULL)
{
    AfxMessageBox("内存不足");
    return false;
}

//根据输入 Volume 的数据类型,
//调用不同的模板函数来完成格式转换.
switch(inVolume->GetDataType())
{
   case MITK_FLOAT:
       t_ExecuteConvertion(inVolume, (float *)inData, outData,
       m_DataType);
      break;

   case MITK_DOUBLE:
       t_ExecuteConvertion(inVolume, (double *)inData, outData,
       m_DataType);
       break;

   case MITK_INT:
       t_ExecuteConvertion(inVolume, (int *)inData, outData,
       m_DataType);
       break;

   case MITK_UNSIGNED_INT:
       t_ExecuteConvertion(inVolume, (unsigned int *)inData,
       outData, m_DataType);
       break;

   case MITK_LONG:
       t_ExecuteConvertion(inVolume, (long *)inData, outData,
       m_DataType);
       break;

   case MITK_UNSIGNED_LONG:
       t_ExecuteConvertion(inVolume, (unsigned long *)inData,
       outData, m_DataType);
       break;
```

```
        case MITK_SHORT:
            t_ExecuteConvertion(inVolume, (short *)inData, outData,
            m_DataType);
            break;

        case MITK_UNSIGNED_SHORT:
            t_ExecuteConvertion(inVolume, (unsigned short *)inData,
            outData, m_DataType);
            break;

        case MITK_UNSIGNED_CHAR:
            t_ExecuteConvertion(inVolume, (unsigned char *)inData,
            outData, m_DataType);
            break;

        case MITK_CHAR:
            t_ExecuteConvertion(inVolume, (char *)inData, outData,
            m_DataType);
            break;

        default:
           AfxMessageBox("未知数据类型");
            return false;
    }

    return true;
}
```

在这段代码中，最核心的地方在于那个大的 Switch 语句，它判断输入 Volume 的数据类型，然后将实际数据指针 inData 强制类型转换成此种类型的指针。为了减少重复的代码，这里还使用了模板函数 t_ExecuteConvertion 来处理不同的数据类型，将输入的数据指针类型参数化，实际的格式转换就是在此函数里面完成的，其代码如下所示：

```
template <typename T>
void t_ExecuteConvertion(mitkVolume *inVolume, T *inData,
                         unsigned char *outData, int dataType)
```

```
{
    int i, j, k, l;
    int imageWidth = inVolume->GetWidth();
    int imageHeight = inVolume->GetHeight();
    int imageNum = inVolume->GetImageNum();
    int channelNum = inVolume->GetNumberOfChannel();
    float *fOutData = (float *) outData;
    double *dOutData = (double *) outData;
    int *iOutData = (int *) outData;
    unsigned int *uiOutData = (unsigned int *) outData;
    long *lOutData = (long *) outData;
    unsigned long *ulOutData = (unsigned long *) outData;
    short *sOutData = (short *) outData;
    unsigned short *usOutData = (unsigned short *) outData;
    char *cOutData = (char *) outData;
    unsigned char *ucOutData = (unsigned char *) outData;

    for(k = 0; k < imageNum; k++)
    {
        for(j = 0; j < imageHeight; j++)
        {
            for(i = 0; i < imageWidth; i++)
            {
                for(l = 0; l < channelNum; l++)
                {
                    switch(dataType)
                    {
                        case MITK_FLOAT:
                            fOutData[0] = (float) inData[0];
                            fOutData++;
                            break;

                        case MITK_DOUBLE:
                            dOutData[0] = (double) inData[0];
                            dOutData++;
                            break;

                        case MITK_INT:
                            iOutData[0] = (int) inData[0];
                            iOutData++;
```

```
                    break;

                case MITK_UNSIGNED_INT:
                    uiOutData[0] = (unsigned int) inData[0];
                    uiOutData++;
                    break;

                case MITK_LONG:
                    lOutData[0] = (long) inData[0];
                    lOutData++;
                    break;

                case MITK_UNSIGNED_LONG:
                    ulOutData[0] = (unsigned long) inData[0];
                    ulOutData++;
                    break;

                case MITK_SHORT:
                    sOutData[0] = (short) inData[0];
                    sOutData++;
                    break;

                case MITK_UNSIGNED_SHORT:
                    usOutData[0] = (unsigned short) inData[0];
                    usOutData++;
                    break;

                case MITK_UNSIGNED_CHAR:
                    ucOutData[0] = (unsigned char) inData[0];
                    ucOutData++;
                    break;

                case MITK_CHAR:
                    cOutData[0] = (char) inData[0];
                    cOutData++;
                    break;
            }

            inData++;
        }
```

```
            }
        }
    }
}
```

在模板函数 t_ExecuteConvertion 中，需要四个参数，第一个 inVolume 是 CMyMitkFilter 的输入 Volume，用来从中得到图像的宽、高、切片张数等信息；第二个 inData 是被参数化的数据指针，它可以指向任意类型的数据；第三个参数 outData 是 CMyMitkFilter 的输出 Volume 的实际数据指针；第四个参数 dataType 用来指示 outData 的实际数据类型。

由于 dataType 可以取任意的数据类型，所以这个函数里面最复杂的地方还是在于那个大的 Switch 语句，它判断 outData 的数据类型，并用合适的指针（fOutData、dOutData 等）来进行运算，将输入数据 inData 里面的数据强制类型转换成所需要的目标格式，并写进 outData 中。整个程序并不复杂，只是由于要处理的数据类型很多，所以比较烦琐。考虑到输入数据的类型可以在 10 种不同的类型中选择，而输出的数据类型也可以在 10 种不同的类型中选择，所以整个可能的组合有 10×10＝100 种，而程序必须处理这 100 种不同的组合，因此代码逻辑就显得比较重要了。

第四步，完成了 CMyMitkFilter 以后，剩下的工作就很容易了，在文档类 CMitkEnhancementDoc 里面添加处理函数 OnConvertToDouble 和 OnConvertToShort，分别响应菜单“转换成 Double”和“转换成 Short”的单击事件，其代码如下所示：

```
void CMitkEnhancementDoc::OnConvertToDouble()
{
    if(m_Volume == NULL)
        return;

    CMyMitkFilter *aFilter = new CMyMitkFilter;

    //设置输入数据.
    aFilter->SetInput(m_Volume);
```

```
    //设置转换后的数据格式.
    aFilter->ConvertToDouble();

    //运行此算法.
    aFilter->Run();

    //得到输出数据.
    mitkVolume *outVolume = aFilter->GetOutput();

    //使用 outVolume.
    //... ...

    //删除算法对象.
    aFilter->Delete();
}

void CMitkEnhancementDoc::OnConvertToShort()
{
    if(m_Volume == NULL)
        return;

    CMyMitkFilter *aFilter = new CMyMitkFilter;

    //设置输入数据.
    aFilter->SetInput(m_Volume);

    //设置转换后的数据格式.
    aFilter->ConvertToShort();

    //运行此算法.
    aFilter->Run();

    //得到输出数据.
    mitkVolume *outVolume = aFilter->GetOutput();

    //使用 outVolume.
    //... ...

    //删除算法对象.
    aFilter->Delete();
```

```
}
```

这两个函数的实现大致相同，都是先生成一个 CMyMitkFilter 的对象，然后设置输入数据，设置转换后的数据类型，运行算法，得到输出数据并使用，最后删除自己的 Filter 对象。这里为了简化起见，并没有详细给出对输出 Volume 的显示和处理，读者有兴趣可以自行加上。

至此，扩充 Filter 功能的实例程序已经基本完成，以这个程序为蓝本，读者可以开发自己的算法，将其集成到 MITK 层次结构中。

## 10.4 小结

本章以实例来描述如何扩充 MITK 的功能，正如我们一开始就追求的目标，MITK 的架构是开放式的，用户可以方便地往里面扩充自己的算法，从而不断地增强 MITK 的功能。

本章首先以一个读入自定义格式的文件为例子，来讲解如何扩充 MITK 中的 Reader 功能，从而支持更多的数据格式，并且这个例子的概念可以完全应用到对 Writer 的扩充中去。

本章的第二个例子以对数据进行格式转换为例，来讲解如何扩充 MITK 中的 Filter 功能，也是最核心的功能，因为所有的算法都要通过 Filter 来实现。扩充 Filter 功能也可以允许不同的算法在同一个框架下实现，从而进行算法的评价工作。

# 11 基于 MITK 的三维医学影像处理与分析系统 3DMed 的设计与实现

## 11.1背景介绍

前面的章节介绍了关于 MITK 的设计框架以及各个子模块的功能以及实现细节。我们已经知道，MITK 是一个软件开发包，它的功能类似于 VTK 和 ITK，可以用来进行二次开发。而本章要介绍的则是一个医学影像处理与分析系统，主要面对医生，为其提供直观易用的辅助工具来进行更准确的医疗诊断，同时也可以应用于远程医疗以及医疗教学中。

本章主要介绍我们自主开发的三维医学影像处理与分析系统 3DMed（3D Medical Image Processing and Analyzing System），它的设计思想、主要功能以及一些实现细节。

## 11.2相关工作

在国际上有很多医学影像处理与分析的应用系统，包括商业软件和科研软件，这里的科研软件指的是软件的目的是为了提供给科研人员或者医疗人员进行研究和开发使用，而不是为了纯粹的商业目的，但它不一定是免费软件，这将在下面介绍。由于 3DMed 的目的是形成一个科研软件，所以这里只介绍相关的国际上的工作，对于纯粹的商业软件并不涉及。

### 11.2.1 3DVIEWNIX 系统简介

3DVIEWNIX 系统是由美国宾州大学放射系医学影像处理小组开发的，提供了医学影像预处理、二维和三维可视化、图像分析等功能，它是使用 C 语言在 Unix 下开发的，利用 X-Window 提供用户界面 [5]。3DVIEWNIX 系统的特色之处就是提供了很多图像分割工具，包括域值分割、基于模糊连接度的分割、Livewire 分割等等，这些工具简化了用户对图像分割的工作量，非常有价值。由于 3DVIEWNIX 开发的比较早，在上世纪 80 年代就已经推出，所以是国际上相当知名的一个系统。但是该软件系统并不是一个免费软件，甚至对科研目的和教学目的也不免费提供，另外加上其只能在 Unix 环境下运行，用户界面比较复杂，所以应用范围收到限制。另外，它现在公开发行的最新版本是 1.4，更新比较缓慢，有很多最近国际上同类系统比较流行的一些新的特性都没有吸收进去，

显得稍微有一点过时。

### 11.2.2 VolView 系统简介

VolView 系统是由美国 Kitware 公司开发的，其主要目的是提供一个易于使用的、交互式的可视化工具。VolView 虽然是一个商业软件，但是 Kitware 公司负责开发并维护着两个非常著名的开放源码的医学影像处理开发包 VTK 和 ITK，其科研背景非常浓厚。VolView 也是基于 VTK 和 ITK 开发出来的，并且提供免费的试用下载。正如其名字所示，VolView 主要的强项在于体绘制（Volume Rendering），其提供了非常好的界面来辅助用户完成复杂的调节参数的过程。在其 2.0 版本之前，VolView 并不提供分割功能，其面绘制（Surface Rendering）功能也是通过脚本语言来支持的，不是非常完善。但是在其最新推出的 2.0 版本以后，通过 ITK 这一强大的分割与配准开发包，提供了比较多的分割算法，也提供了一些等值面生成（Isosurface Generation）的算法来支持面绘制。VolView 系统现在主要是针对 Windows 操作系统提供，在 Linux 系统下也有提供，不过版本比较低。另外，其新提供的分割功能集中在一个相对比较小的面板上，需要调节的参数又比较多，界面不够直观。

## 11.3 3DMed 的整体设计

此处所讲的 3DMed 是我们在自己以前老版本的 3DMed[6] [7]的基础之上，基于我们新开发的一套集成化的医学影像处理与分析开发包 MITK[8]而完全重新编制的新版本的 3DMed。其提供了更灵活的框架、更强大的易扩充性以及更友好的界面，最重要的是，现在 3DMed 更改了发行方式，已经成为免费软件（Freeware）。

### 11.3.1 3DMed 的设计目标

对于软件设计，尤其是医学影像这一特定领域内的复杂软件设计，必须事先有一个非常明确的设计目标。3DMed 从一开始设计，就始终追求以下几个高层的设计目标：

(1) 支持跨平台

这一目标是考虑到 3DMed的潜在用户分为两类，一类是研究和开发人员，另外一类是普通用户。他们在对操作系统的选择上有着不同的倾向，为了使

3DMed能够得到最广泛的应用，支持跨平台是非常重要的一个环节。为了实现这一目标，3DMed的设计是完全分成模块来进行的，它的整个的结构如 图 11-1 所示。其中 3DMed是建立在两个开发包的基础之上，一个是我们自己开发的医学影像处理与分析开发包MITK，它负责提供所有的图像处理、可视化、分割、配准等核心算法，并且我们在设计MITK的时候，目标之一就是实现跨平台的支持；另外一个是用户界面开发包，为了能够让 3DMed跨平台运行，这个用户界面开发包也必须能够跨平台，幸运的是，现在有很多这样的支持Windows、Linux等多个操作系统的用户界面开发包可以免费得到。建立在这两个跨平台的开发包的基础之上，整个 3DMed的代码也是全部使用ANSI C++编写，没有使用任何编译器提供的特殊关键字或者特殊函数，因此 3DMed可以很自然地在多个操作系统下运行。

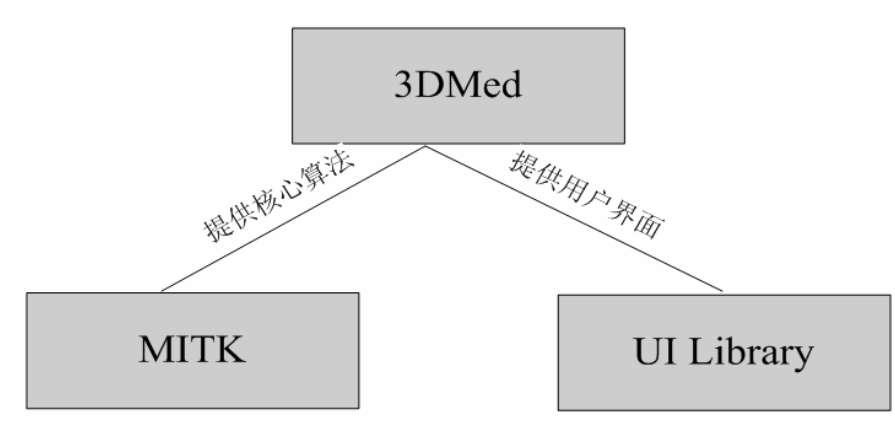



图 11-1 3DMed 的结构图

(2) 强大的可扩充性

前面已经提到过，3DMed 是属于科研软件性质的，所以其一个最重要的设计目标就是可扩充性，允许第三方的科研机构或者开发人员将自己的功能集成到 3DMed 中去。为此 3DMed 里面提供了一个灵活的、基于 Plugin 的框架，所有 3DMed 的主要功能，包括分割、可视化等，都是通过一个一个的 Plugin 来实现的，这些 Plugin 在运行时被动态加载，因此用户可以按照 3DMed 事先定义好的 Plguin 的规范，来开发自己的 Plugin，并放入特定的目录，这样 3DMed 就能

动态地将其加载进来。

(3) 易于获取

根据我们对先前版本的 3DMed 的一定范围的免费发行所取得的经验，为了使 3DMed 能够得到最广泛的应用，必须使其能够很容易地被用户获取。现在我们已经将 3DMed 作为免费软件（Freeware）发行，并且直接放在 Internet 网络上供国内外相关人员免费下载，除了不能用于商业目的和必须保留版权信息等一些条件外，用户可以完全免费地在自己的科研工作中使用 3DMed。

### 11.3.2 3DMed 提供的功能简介

图 11-2 给出了 3DMed所提供的基本的功能，其中二维操作、虚拟切割和三维测量等功能是在 3DMed的核心中实现的，相对固定，同时为了形成一个相对完整的版本，核心里面也实现了基本的表面绘制和体绘制功能；而医学影像数据I/O、医学影像分割、医学影像配准、表面绘制和体绘制等功能是由Plugins动态加载进来的，也就意味着这一部分的功能是可以根据需要来添加的，是动态部分。下面分别对每一个功能作简单的介绍。

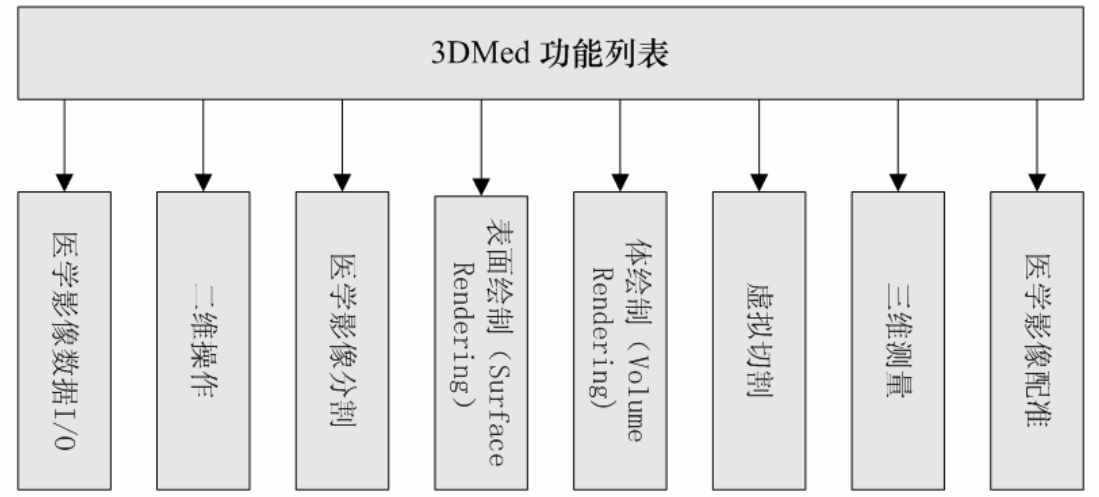



图 11-2 3DMed 提供的功能列表

(1) 医学影像数据 I/O

这一部分的功能主要是提供对多种医学影像数据格式的支持，为了能够处理

新的格式以及方便地加入其它格式的支持，这一部分采用了 Plugin 机制，每一种格式的读和写都分别是一个独立的 Plugin，可以随时被加载。现在 3DMed 中提供了对 DICOM 3.0 标准（部分支持）规定的数据的读取和存储、序列 BMP 图像的读取和存储、序列 JPEG 图像的读取和存储、序列 TIFF 图像的读取和存储、ACR-NEMA 标准规定的数据的读取和存储、生数据的读取和存储。另外，这一部分的功能可以当做一个格式转换器，在 3DMed 所支持的这些数据格式中进行转换。

(2) **二维操作**

这一部分的功能主要是提供二维的阅片功能，主要包括：三个断面上的图像的同时显示、图像浏览、动画播放、窗宽窗位调整、几何变换、伪彩显示、测量和标注等等。使用者可以选择各种方式阅片，并能够通过滤波处理来消除噪声，提高图像质量。还能方便的得到 CT 值，对图像的像素点进行分析、计算、处理，得出相关的完整数据，为医学诊断提供从定性到定量、更客观的信息。

(3) **医学影像分割**

分割是整个医学影像处理与分析的核心所在，因为它的结果直接影响着后续的三维显示或者配准等操作的质量。因为医学影像的模态多种多样，也各有各的特点，所以存在多种分割算法。为了能够随时加载新的分割算法，在 3DMed 里面，这一部分也采用了 Plugin 机制，每一种分割算法都是一个独立的 Plugin，可以在运行时被加载。当前 3DMed 里面已经提供了手工交互分割、阈值分割、种子生长分割、Fast Marching 分割、Live Wire 分割、Level Set 分割等算法，并且以后新的算法会不断地以 Plugin 的形式加进去。

(4) **表面绘制**（Surface Rendering）

表面绘制提供了一种手段来以真实感的三维图形来显示人体内部器官，它首先要经过分割这个步骤，将感兴趣的器官提取出来，然后用三维重建算法生成其表面（IsoSurface），再使用图形学的方法将其绘制出来。3DMed 的核心层提供了一种基于分割的增强的 Marching Cubes 算法[9]来实现表面绘制功能，同时为了易于扩充，同样也提供了对表面绘制 Plugin 机制的支持。

(5) **体绘制**（Volume Rendering）

体绘制也是一种三维显示的手段，但是与表面绘制不同，它不需要经过分割

这一步骤，直接将数据集里面所有的器官同时显示出来，用传递函数（Transfer Function）来控制显示的效果，它可以很直观地反映出整个数据场的全貌。3DMed 的核心层提供了基于 Ray Casting 的体绘制算法，也提供了很多易于使用的界面来调节传递函数，不过考虑到体绘制算法的多样性，同时为了易于扩充，这里同样也提供了对体绘制 Plugin 机制的支持。

(6) 虚拟切割

虚拟切割可以使用户看到被外部器官遮挡住的组织和器官，它通过将数据场的一部分切割掉，从而更清晰地反映内部的信息。在 3DMed 里面，支持对表面绘制和体绘制的结果实行虚拟切割，其中对于表面绘制，支持任意平面的切割；而对于体绘制，除了平面切割以外，还支持用立方体进行切割。

(7) 三维测量

三维测量是对二维图像的测量与标注功能的扩充，它可以允许使用者进行空间任意两个点之间的距离测量、空间角度的测量等等。为了在两维的输入设备和显示设备之下能够提供直观的、直接对三维物体的操作，3DMed 里面使用了 3D Widgets 来实现三维的人机交互界面。

(8) 医学影像配准

配准和分割、可视化算法一起，组成了医学影像处理与分析的理论基础。有了配准，就可以实现多个不同模态的医学影像的信息融合和可视化，可以为临床诊断或者科学研究提供更多的有用信息。考虑到配准算法的多样性，在 3DMed 里面，这一部分也采用了 Plugin 机制，每一种配准算法都是一个独立的 Plugin，可以随时被加载。当前 3DMed 里面只提供了少量的刚性配准算法，随着 3DMed 功能的不断完善，多种刚性和非刚性的配准算法将会很快以 Plugin 的形式加进去。

## 11.43DMed 的 Plugin 整体框架的实现

3DMed 是医学影像这一特定领域内的一个复杂的软件系统，它的核心算法由 MITK 来提供，而它自己是处在应用层上面，正如图 1 所示。下面从软件设计的角度比较宏观地介绍 3DMed 的整体框架，更偏重于算法角度的文献请参考[8]。

因为 3DMed 是一个大而复杂的系统，牵涉到医学影像分割、配准和可视化等多个方面，而这三个方面中的每一个小的方面，本身都是一个非常复杂的系统，有很多种不同的算法，有很多不同的参数需要调节。为了降低 3DMed 各个模块之间的藕合性，同时更重要的目的是为了给使用者提供一个开放的、易扩充的架构，整个 3DMed 使用了 Plugin 机制。

所谓 Plugin，就是一个动态链接库[10]，在 Windows 操作系统下的表现形式就是一个 DLL 文件，它按照一定的规范来编写，所以可以被 3DMed 的核心所动态加载并调用。要实现一个 Plugin 机制，必须有三个部分相互协作来共同完成，如图 3 所示。第一个部分 3DMed Kernel 是系统核心，它负责加载、管理并调用各个 Plugin；第二个部分 Plugin SDK 是编写 Plugin 的规范，所有 Plugin 的编写者都必须遵循这个规范，生成的 Plugin 才能被系统核心所识别并加载；第三个部分 Plugins 就是各个具体的 Plugin 实现了，它们一方面遵循 Plugin SDK 所制定的接口，另外一方面要负责实现具体的功能。下面将分别讲述这三个部分在 3DMed 里面的实现。

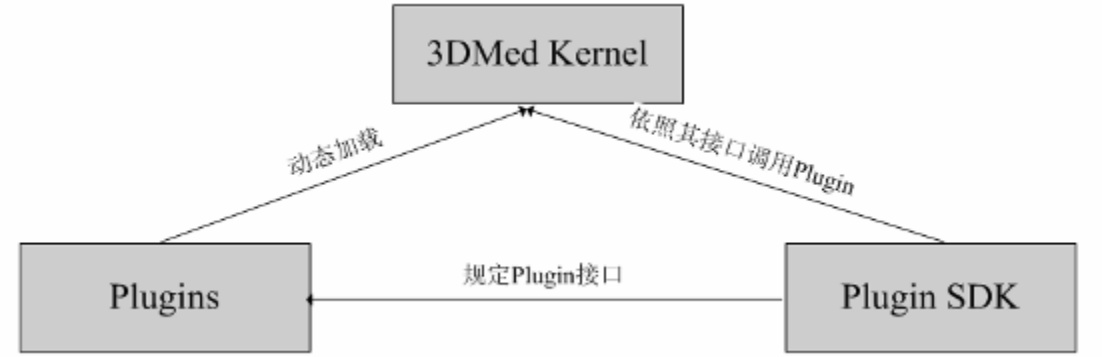



图 11-3 Plugin 框架的三个组成部分

### 11.4.1 Plugin SDK 的实现

Plugin SDK 在整个 Plugin 框架中起着至关重要的作用，它不仅为系统核心提供 Plugin 的调用接口，同时也为 Plugin 的开发者（可以是使用者）规定了撰写 Plugin 所必须遵循的接口。要想使第三方开发者能够写出自己的 Plugin，那么必须把 3DMed 的内部数据通过一个接口暴露给外部。在 3DMed 中，使用的数

据有两个大类：medVolume 和 medMesh，它们分别代表由医学影像设备采集得到的三维数据集和表面重建算法生成的几何面片网格。它们分别通过一些 Get 函数将自己的属性以及实际数据暴露给 Plugin 的开发者，如图 11-4 所示。

另外，整个 3DMed 的 Plugins 被分为五个大类，分别是 I/O Plugins、Filter Plugins、Registration Plugins、Segmentation Plugins 和 Visualization Plugins，请参看图 11-4。它们都从一个共同的抽象基类 medPlugin 继承下来，并且都继承了纯虚函数 Show，其子类必须实现这一纯虚函数以完成具体功能。I/O Plugins 提供对不同数据格式的读入和保存，由于数据对象分为两种，所以这一部分总共有四种 Plugins，分别是对 Volume 的读和写、对 Mesh 的读和写。对于读数据的 Plugin 来说，它只需通过其 GetOutput 函数将读进来的数据以 Volume 或者 Mesh 的形式返回就行了；对于写数据的 Plugin 来说，它需要用 SetInput 函数从系统核心那里得到一个 Volume 或者 Mesh，然后将其写成自己的格式。对于 Filter Plugins 来说，也因为两种不同的数据而分为两种 Plugin：medVolumeFilterPlugin 以 SetInput 函数从系统核心那里得到一个 Volume，然后经过自己在 Show 函数里面的处理，生成一个滤波后的 Volume，通过 GetOutput 函数返回给系统核心，大部分的图像处理算法都可以通过这种 Plugin 加载进 3DMed；medMeshFilterPlugin 以 SetInput 函数从系统核心那里得到一个 Mesh，然后经过自己在 Show 函数里面的处理，生成一个滤波后的 Mesh，通过 GetOutput 函数返回给系统核心，大部分的数字几何处理算法，包括面片化简、面片平滑、面片细分等都可以通过这种 Plugin 加载进 3DMed。Registration Plugins 是直接对应各种配准算法的，它需要两个 Volume，生成一个配准后的 Volume。Segmentation Plugins 是直接对应分割算法的，它实际上可以归为 medVolumeFilter 的一种，但是考虑到分割算法的重要性，这里还是将其单独作为一个种类，它接收一个原始的未分割的 Volume，生成一个分割后的 Volume。Visualization Plugins 是直接对应各种可视化算法的，除了系统核心里面提供的标准的算法以外，其它新的算法可以通过这种手段加载进来。这种 Plugin 只需要接收一个 Volume，然后直接生成显示后的结果图像，可以让使用者交互操作。

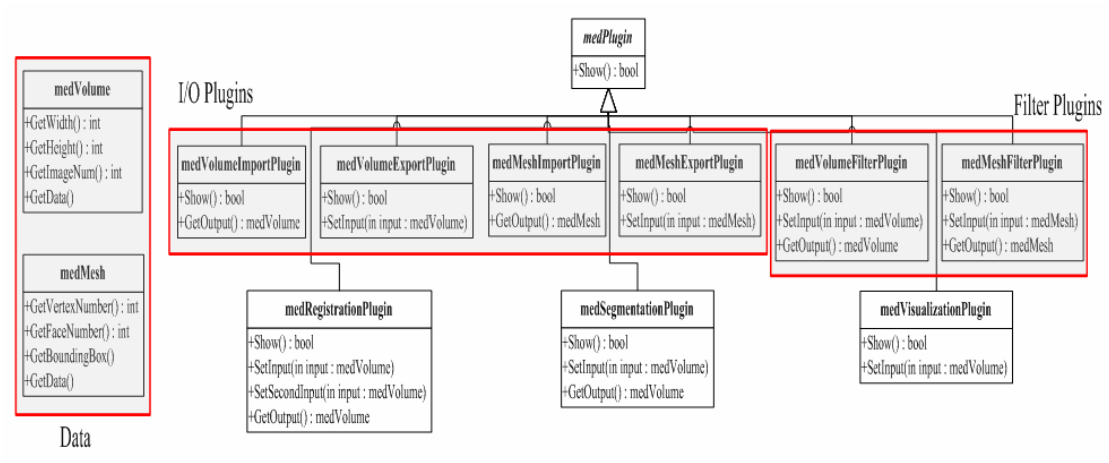



图 11-4 Plugin SDK 的类层次结构图

### 11.4.2 Plugins 的实现

前面已经提到过，一个 Plugin 实际上就是一个动态链接库，在功能上对应着某一种具体的算法。要实现一个具体的 Plugin，首先要确定其属于哪一类的 Plugin，比如是要实现读入一系列 BMP 这一功能的 Plugin，那么它应该是属于 I/O Plugins 下面的 VolumeImport 这一种类，所以其必须从 medVolumeImportPlugin 继承，并实现父类的纯虚函数 Show ，在其中完成 BMP 文件的读取工作。

另外，实现一个 Plugin 时还有一些技术难题需要解决，因为动态加载一个动态链接库时，只能从中得到 C 函数的指针，而 C++由于命名转换的问题，不能直接拿到 C++成员函数的指针。为了解决这一难题，3DMed 中使用了设计模式中的 Abstract Factory[11]，它要求每个 Plugin 必须提供一个 C 函数 MakePlugin，动态生成一个自己的实例，但是将其转换为父类类型的指针并返回，下面依旧以 BMP 的读入这一 Plugin 为例，它的 MakePlugin 函数的声明和实现如下：

```
medVolumeImportPlugin* MakePlugin() {return new medBMPImportPlugin;}
```

### 11.4.3 3DMed Kernel 的实现

系统核心的主要作用是在运行时动态加载各个类型的 Plugins，并根据类型，动态生成菜单项，在使用者点击菜单项时调用对应的 Plugin 功能，并且在系统

退出时清除掉加载进来的 Plugins。

要动态加载 Plugins，那么每个 Plugin 必须提供一些必要的信息，以便系统核心能够识别它们、为其分类并且生成对应的菜单项。在 3DMed 的实现里，要求每个 Plugin 必须提供如下的几个 C 函数，以便系统核心能够得到它们的指针并调用它们：

```
const char* GetTypeName();
const char* GetClassname();
const char* GetMenuDescription();
```

其中，GetTypeName 给出这个 Plugin 的类型，就是图 11-4 所示的九种类型中之一；GetClassname 给出这个 Plugin 的名字；GetMenuDescription 给出这个 Plugin 在界面菜单上显示的字符串。为了简化 Plugin 开发者的工作，Plugin SDK 使用 C/C++语言里面的宏作为代码生成器，自动生成包括 MakePlugin 在内的这四个必须的函数。下面给出 medVolumeImportPlugin 这种类型的 Plugin 所对应的宏代码：

```
#define IMPLEMENT_VOLUME_IMPORT(ClassName, MenuDescription) \
extern "C" { \
MEDEXPORT const char* GetTypeName() {return "VolumeImportPlugin"; } \
MEDEXPORT const char* GetClassname() {return #ClassName;} \
MEDEXPORT const char* GetMenuDescription() {return MenuDescription;} \
MEDEXPORT medVolumeImportPlugin* MakePlugin() {return new ClassName;}
\
}
```

加载完 Plugins 以后，还必须对其进行管理。3DMed 里面仍然使用了 Abstract Factory 这一设计模式来进行 Plugins 的管理，从而保证框架的灵活性和优雅性。如图 11-5 所示，medFactory 是一个模板类，T 是待参数化的类型，它可以是九种 Plugin 类型中的一种。medFactory 内部使用一个 map 来管理 Plugins，并且使用 Plugin 的名字作为键值来索引 Plugin 的指针。Add 函数将一个新的 Plugin 登

记到 Factory 里面，而 Create 函数从 Plugin 的名字得到其实际指针，Clear 函数销毁掉所有的 Plugins。

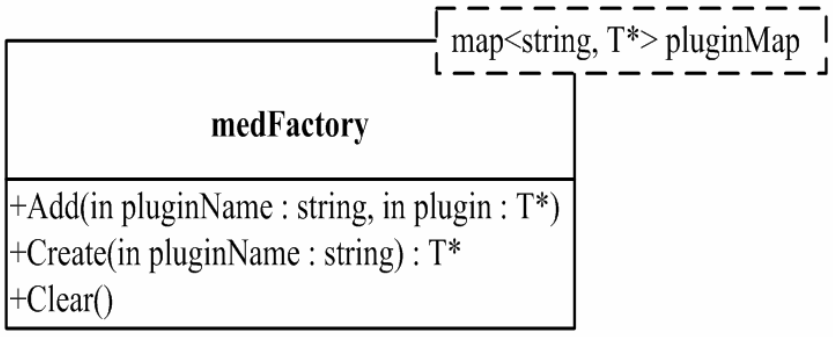



图 11-5 Plugins Factory 架构

下面的伪代码给出了 3DMed 核心在运行时加载并管理各种 Plugins 的流程图：

```
对于 Plugin 目录下面的每个文件：
判断是否有 GetTypeName 等四个必须的函数；
如果有，则表明是一个有效的 Plugin；
typeNameStr = GetTypeName();
classNameStr = GetClassName();
menuDescriptionStr = GetMenuDescription();
由 typeNameStr 来判断此 Plugin 的类型；
typeNameStr newPlugin = MakePlugin();
medFactory<typeNameStr> :: Add(classNameStr, newPlugin);
生成动态菜单，菜单标题是 menuDescriptionStr；
设置菜单的消息响应事件为此 Plugin 的 Show 函数；
```

## 11.5 应用实例

图 11-6 给出了 3DMed 的主界面，左边是参数调节区域，右边上半部分是三维显示区域，下半部分是二维显示区域。三维显示给出了人体大脑的表面绘制，二维显示给出了三个断面上的图像，同时一部分给以伪彩显示。

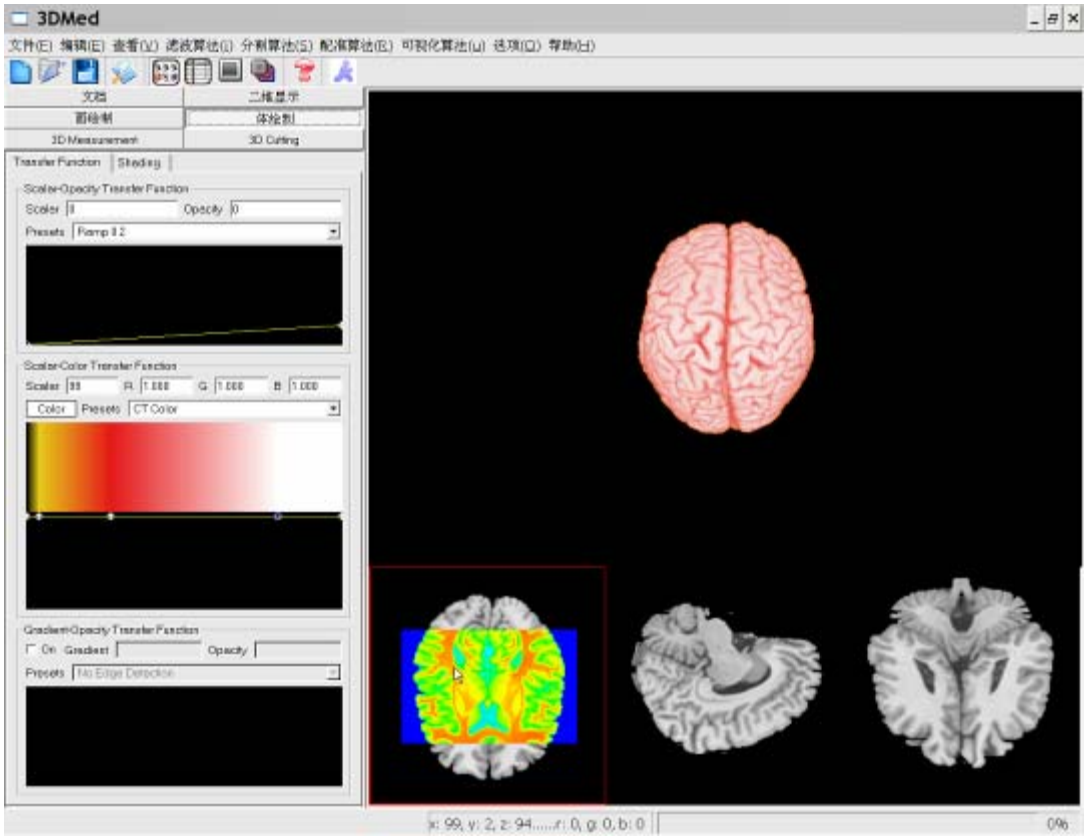



图 11-6 3DMed 的主界面

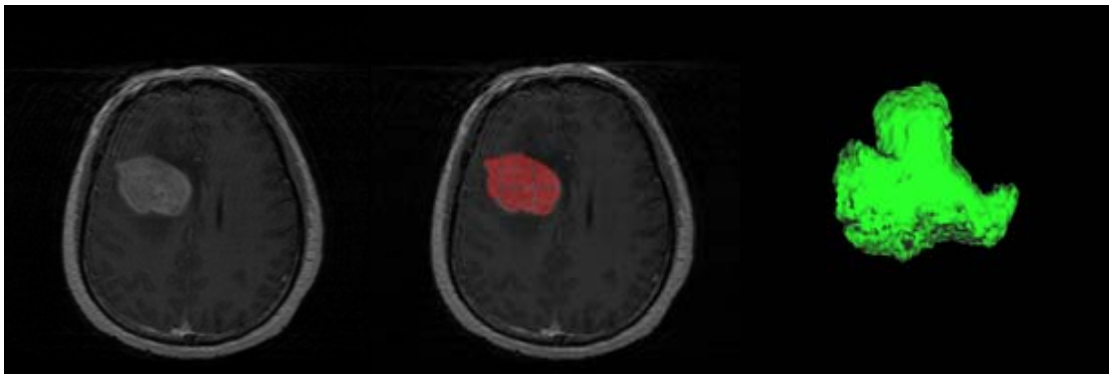


图 11-7 肿瘤分割的应用实例

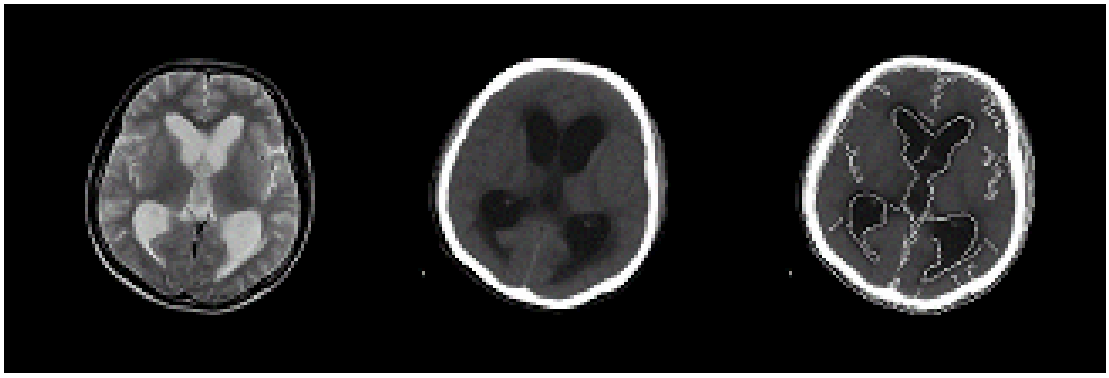


图 11-8 配准的应用实例

图 11-9 可视化的应用实例

图 11-7 给出了一个肿瘤的分割实例，左边的图像是原始的切片，中间的图像是分割后的结果，肿瘤以红色部分显示，右边的图像是三维重建后的显示结果。

图 11-8 给出了CT图像和MR图像配准的实例，左边的图像是原始的CT图像，中间的图像是原始的MR图像，右边的图像是配准并叠加在一起的图像。

图 11-9 给出了三个可视化的例子，左边的图像是对骨骼进行面绘制得到的结果，中间的图像是对骨骼和皮肤进行多层面绘制得出的结果，右边的图像是体绘制并经过切割后得到的结果。

## 11.6 小结

本书介绍了我们在前期工作积累的基础上开发的新版的三维医学影像处理与分析系统 3DMed，目前我们已经完成一个初步的版本，为了更大限度地普及3DMed的用户，现在 3DMed已经作为免费软件发行。任何人都可以在http://www.3dmed.net/mitk免费下载并使用 3DMed，同时在这个网址还可以免费下载我们开发的底层算法包MITK。

在将来的工作中，我们将不断地收集用户的反馈信息，持续地改善 3DMed。另外，我们还将不断地往 3DMed 里面加入更多的 Plugins，如果第三方开发的Plugins 达到一定的标准以后，也可以加进 3DMed 的发行版本当中，通过这一手段来不断地补充和丰富它的功能。

# 12 开发 3DMed 的 Plugin

上一章介绍了 3DMed 的整体功能和 Plugin 框架，从中我们可以看到 3DMed 也是一个开放的架构，允许用户自己通过撰写 Plugin 来扩充其功能，以满足实际使用中的需求。本章介绍如何开发 3DMed 的 Plugin，首先给出总体介绍，然后通过两个实际的例子，以 Microsoft Visual C++ 6.0 来作为编程环境，一步一步地演示如何撰写自己的 Plugin。

## 12.1总体介绍

在 3DMed 中，所有的 Plugins 分为五个大类（I/O Plugin、Filter Plugin、Registration Plugin、Segmentation Plugin 和 Visualization Plugin），其中 I/O Plugin 又分为 medVolumeImportPlugin、medVolumeExportPlugin、medMeshImportPlugin 和 medMeshExportPlugin 四个小类，Filter Plugin 又分为 medVolumeFilterPlugin 和 medMeshFilterPlugin 两个小类。这五大类九小类 Plugin 定义了编写 3DMed Plugin 时必须遵循的接口，是整个 Plugin 机制的基础。

要编写一个 3DMed 的 Plugin，一般的步骤如下所示：

第一步，先判断自己要完成的 Plugin 在功能上属于上面所讲的五大类中哪一类的，然后再找出从属的具体的某个小类；

第二步，写自己的 Plugin，从第一步中选择的那个基类中公有继承；

第三步，实现虚函数 bool Show ()，在这个函数里面完成 Plugin 的具体功能。注意的是，在这个函数里面你可以作出完整的 GUI 图形界面来完成某项功能，也可以不要图形界面，只是在后台完成功能。这个接口规定的相当宽松，所以实现 GUI 图形界面时所用的编程工具也并没有限制，本章中的例子使用 Microsoft 的 MFC 来作为 GUI 开发工具，而 3DMed 里面自带的所有的 Plugin 都是使用跨平台的 GUI 类库 Qt 来开发的，你甚至可以使用 Windows API，或者 Borland 的 C++ Builder，所有这些工具作出来的 Plugin 都可以很好地被 3DMed 的主程序所管理和调用。

另外，3DMed 通过两个数据类：medVolume 和 medMesh 把自己内部的数据暴露给 Plugin 的开发者，所以要开发一个 3DMed 的 Plugin，还必须对这两个数据类所提供的 API 函数有所了解，medVolume 所提供的主要的 API 函数如下：

```
void SetWidth(int w);  //设置 Volume 的宽度（Pixel 为单位）
int  GetWidth();  //得到 Volume 的宽度（Pixel 为单位）

void SetHeight(int h);  //设置 Volume 的高度（Pixel 为单位）
int  GetHeight();  //得到 Volume 的高度（Pixel 为单位）

void SetImageNum(int s);  //设置 Volume 的切片张数
int  GetImageNum();  //得到 Volume 的切片张数

void  SetSpacingX(float px);  //设置像素间的水平间距（mm 为单位）
float GetSpacingX();  //得到像素间的水平间距（mm 为单位）

void  SetSpacingY(float py); //设置像素间的竖直间距（mm 为单位）
float GetSpacingY();  //得到像素间的竖直间距（mm 为单位）

void  SetSpacingZ(float pz); //设置切片间的间距（mm 为单位）
float GetSpacingZ();  //得到切片间的间距（mm 为单位）

//设置 Volume 的通道数（1 为灰度图像，3 为 RGB 彩色图像）
void SetNumberOfChannel( int n );
//得到 Volume 的通道数（1 为灰度图像，3 为 RGB 彩色图像）
int  GetNumberOfChannel();

//得到实际的数据指针，必须依据 GetDataType 返回的数据类型，
//将其强制转换成相应的指针类型才能使用
void* GetRawData();
//得到某一张切片的数据指针，必须依据 GetDataType 返回的数据类型，
//将其强制转换成相应的指针类型才能使用
void* GetSliceData(int sliceNum);

//得到此 Volume 的数据类型，返回的值及其代表的意义如下：
// MED_CHAR ——c 语言中 char 类型
// MED_UNSIGNED_CHAR ——c 语言中 unsigned char 类型
// MED_SHORT ——c 语言中 short 类型
// MED_UNSIGNED_SHOR ——c 语言中 unsigned short 类型
// MED_INT ——c 语言中 int 类型
// MED_UNSIGNED_INT ——c 语言中 unsigned int 类型
// MED_LONG ——c 语言中 long 类型
// MED_UNSIGNED_LONG ——c 语言中 unsigned long 类型
// MED_FLOAT ——c 语言中 float 类型
```

```
// MED_DOUBLE ——c 语言中 double 类型
int  GetDataType();

//设置 Volume 的数据类型,变量 type 的取值及意义参见 GetDataType
void SetDataType(int type);
//设置 Volume 的数据类型为 c 语言中的 float 类型
void SetDataTypeToFloat();
//设置 Volume 的数据类型为 c 语言中的 double 类型
void SetDataTypeToDouble();
//设置 Volume 的数据类型为 c 语言中的 int 类型
void SetDataTypeToInt();
//设置 Volume 的数据类型为 c 语言中的 unsigned int 类型
void SetDataTypeToUnsignedInt();
//设置 Volume 的数据类型为 c 语言中的 long 类型
void SetDataTypeToLong();
//设置 Volume 的数据类型为 c 语言中的 unsigned long 类型
void SetDataTypeToUnsignedLong();
//设置 Volume 的数据类型为 c 语言中的 short 类型
void SetDataTypeToShort();
//设置 Volume 的数据类型为 c 语言中的 unsigned short 类型
void SetDataTypeToUnsignedShort();
//设置 Volume 的数据类型为 c 语言中的 char 类型
void SetDataTypeToUnsignedChar();
//设置 Volume 的数据类型为 c 语言中的 unsigned char 类型
void SetDataTypeToChar();

//根据预先设置好的 Width、Height、SliceNumber、Datatype 和 ChannelNumber
//来计算实际数据所需的内存大小，并分配内存，最后返回首地址。需要注意的是，在调
//用此函数之前，必须确保已经调用了 SetWidth、SetHeight、SetImageNum、
//SetNumberOfChannel、SetDataType 设置好了相关的信息。
void* Allocate();
//返回数据所占用的内存的大小，单位是字节。
unsigned long GetActualMemorySize();
```

medMesh 提供的主要的 API 函数如下：

```
//设置此 Mesh 的顶点个数
void SetVertexNumber(int number);
```

```
//得到此 Mesh 的顶点个数
int GetVertexNumber();

//设置此 Mesh 的三角片个数
void SetFaceNumber(int number);
//得到此 Mesh 的三角片个数
int GetFaceNumber();

//得到此 Mesh 的顶点数据指针
float* GetVertexData();

//得到此 Mesh 的三角面片数据指针
unsigned int* GetFaceData();

//得到此 Mesh 的包围盒，返回一个指向六个浮点数的指针.
//六个浮点数的顺序及意义为：最小 x，最小 y，最小 z，
//最大 x，最大 y，最大 z。
float* GetBoundingBox(void);
//设置此 Mesh 的包围盒，参数意义见 GetBoundingBox
void   SetBoundingBox(float minX, float minY, float minZ, float maxX,
float maxY, float maxZ);

//得到顶点数据和三角面片数据所占的内存大小，单位是字节。
unsigned long GetActualMemorySize();
```

有了 medVolume 和 medMesh 提供的 API 函数以后，Plugin 的开发者就可以通过它们访问 3DMed 的内部数据，并且他们不需知道 MITK 的存在，只需要和 medVolume 和 medMesh 打交道就行了。这也是 3DMed 的 Plugin 框架的灵活性的体现之一，可以允许 3DMed 的使用者在没有 MITK 的情况下，照样可以扩充 3DMed 的功能。

上面给出了一些比较抽象的总体介绍，下面将会提供一些撰写 Plugin 的更具体的内容。首先一个 Plugin 是一个动态链接库，所以在一开始创建工程的时候就要选择正确的工程类型；其次所有 Plugin 的开发都要用到 PluginsSDK，它里面提供了 medVolume、medMesh 以及九个小类的 Plugin 的定义，PluginsSDK 是随着 3DMed 一起分发的，提供了必要的头文件和库文件；最后，Plugin 的开发可以分为两种，一种使用 MITK 作为底层算法库，一种可以完全不使用 MITK，

纯粹加入自己的算法，如果使用 MITK 的话，那么开发 Plugin 时还需要 MITK 的头文件以及相应的库文件。

下面将用两个具体的实例来一步一步地演示 3DMed Plugin 的开发，其中第一个实例使用 MITK，而第二个实例不使用 MITK，力图将不同的情况以及一些基本的概念阐明清楚。

## 12.2 Plugin 实例：使用 MITK

本实例使用 Microsoft Visual C++ 6.0 作为工具，从如何建立工程，到修改工程的设置，以及具体的编程实现，一直到最后集成进 3DMed，给出了一个完整的例子，演示如何在使用 MITK 的情况下开发 3DMed 的 Plugin。为了清晰和简化起见，这个 Plugin 的功能比较简单，只是读入一系列 BMP 格式的文件，并送给 3DMed 处理。

### 12.2.1 工程的建立及设置

在Microsoft Visual C++ 6.0 的IDE环境下，新建一个工程IOPlugin，工程的类型选择 “MFC AppWizard(dll)” （如 图 12-1 所示），因为我们要使用MFC作GUI图形界面，并且目标是生成一个动态链接库。在接下来的一步中，选择DLL类型为“Regular DLL using shared MFC DLL”，如 图 12-2 所示。此时可以选择“Finish”按钮完成工程的创建。

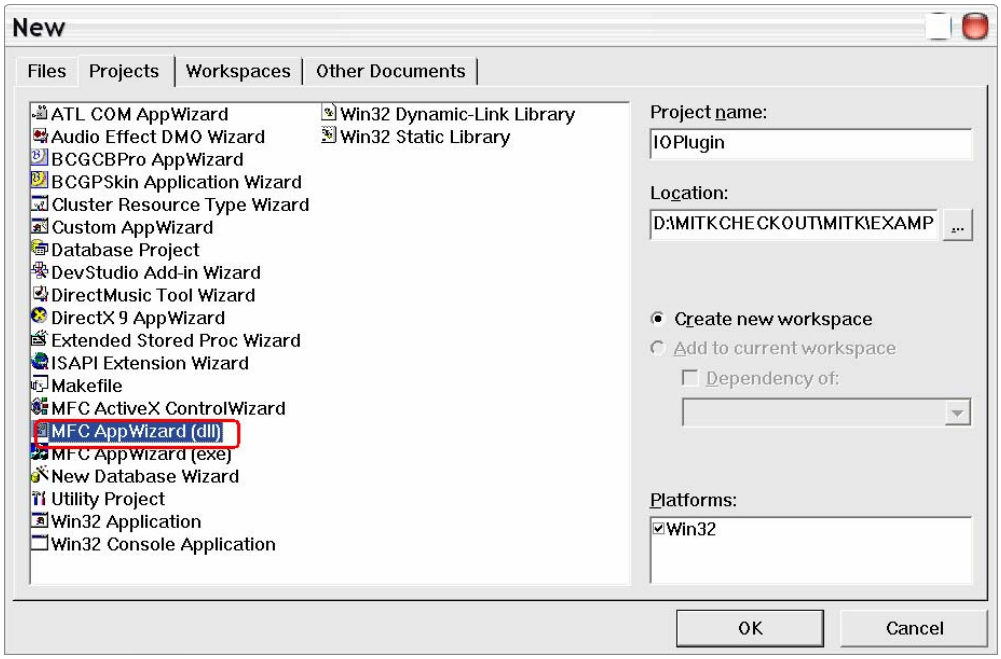



图 12-1 新建 IOPlugin 工程

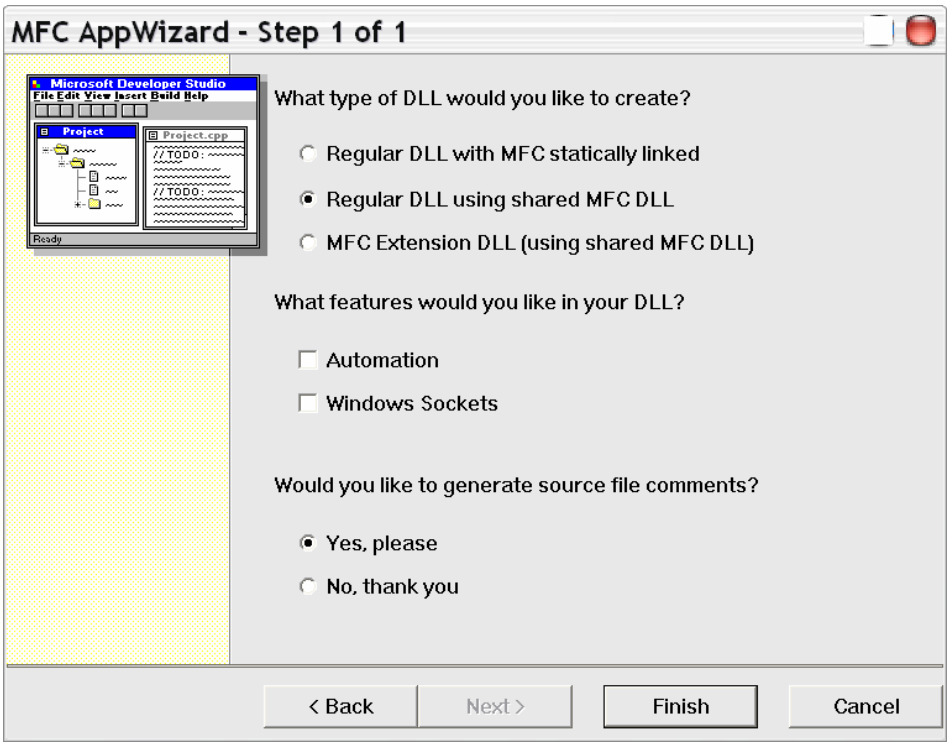



图 12-2 Dll 类型选择

创建完工程以后，选“Projcet”—》“Settings”菜单，在弹出来的“Project Settings”对话框中，选择“C/C++”标签页，在“Category”列表框里面选择“Preprocessor”，然后在“Additional Include directories”编辑框中输入PluginsSDK和MITK的头文件路径，如 图 12-3 所示。需要注意的是这里实际的路径依赖于你的机器上的PluginsSDK和MITK的放置路径，请根据实际情况设置。

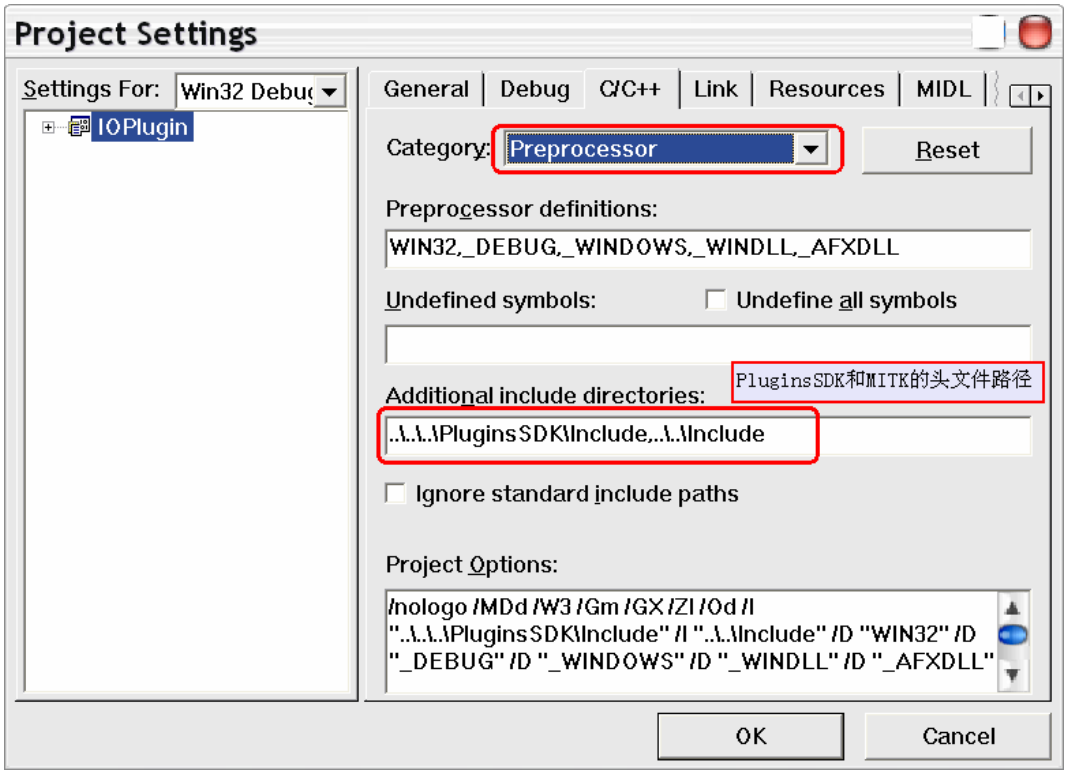



图 12-3 设置必要的头文件路径

设置完头文件路径以后，接下来要设置的是必要的导入库文件的路径。保持“Projcet Settings”对话框打开，选择“Link”标签页，在“Category”列表框中选择“Input”，然后在“Object/library modules”编辑框中输入“PluginsSDK.lib Mitk_dll.lib”，表示要使用这两个导入库，并且还需要在“Additional library path”编辑框中输入这两个导入库文件所在的路径，如 图 12-4 所示。需要注意的是这里实际的路径依赖于你的机器上的PluginsSDK和MITK的库文件的放置路径，请根据实际情况设置。

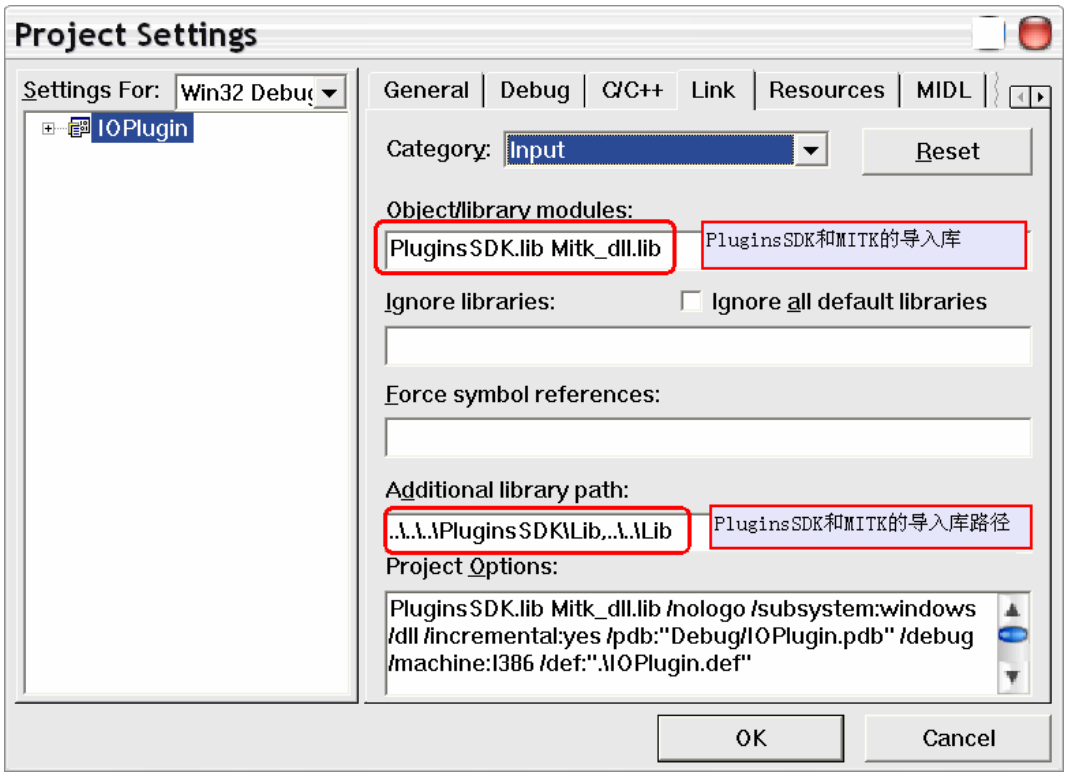



图 12-4 设置必要的库文件路径

### 12.2.2 实例制作

好了，设置好烦琐的工程选项以后，就可以进入实质部分了。下面创建我们的Plugin的类，用“New Class”创建一个新类，并将其命名为CMyIOPlugin，如图 12-5 所示。因为我们的Plugin的功能是读入一系列BMP格式的文件，所以很显然是属于I/O Plugin中的VolumeImportPlugin，故CMyIOPlugin应该从VolumeImportPlugin公有继承，并应该重载 Show 函数，其类声明如下所示：

```
class CMyIOPlugin : public medVolumeImportPlugin
{
public:
    CMyIOPlugin();
    virtual ~CMyIOPlugin();

    virtual bool Show(void);
```

```
};
```

当然，在其前面应该包含 PluginsSDK 中的 medPlugin.h 文件，里面定义了 medVolumeImportPlugin，包含语句如下：

```
#include "medPlugin.h"
```

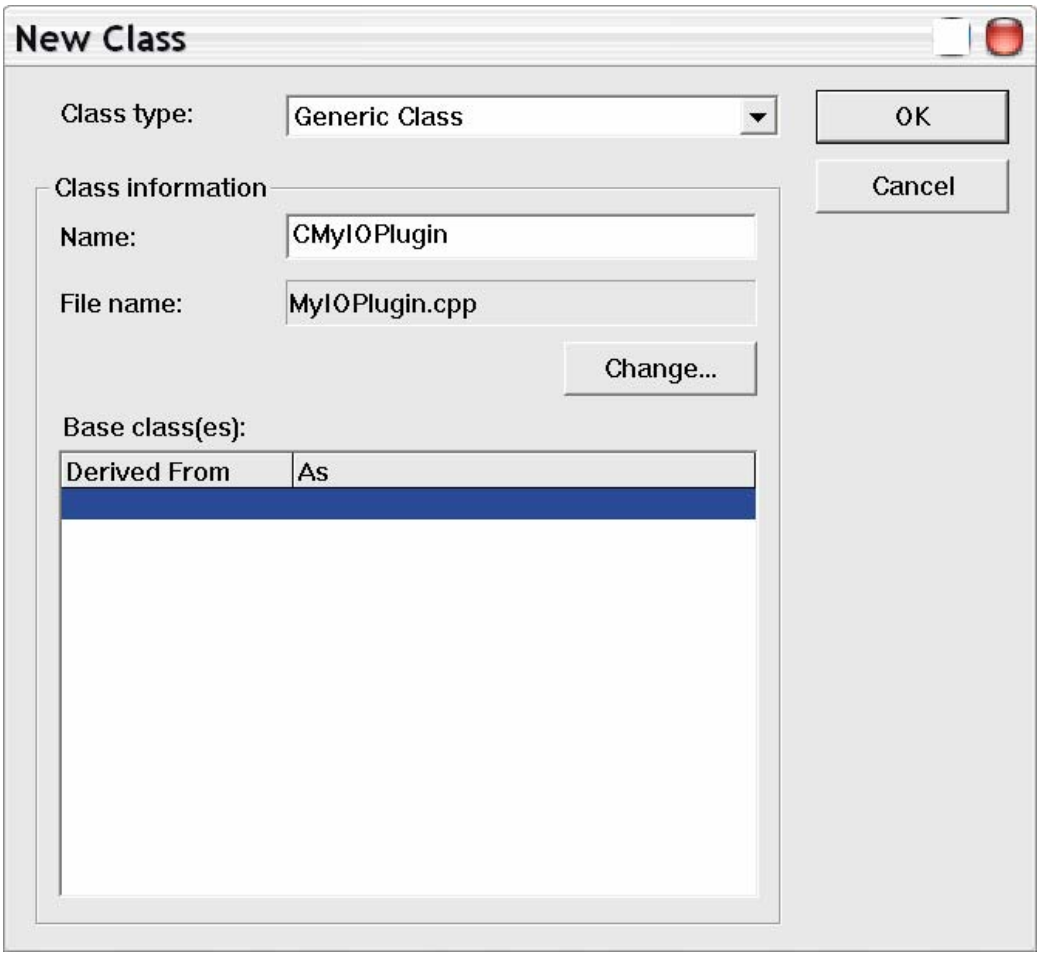



图 12-5 创建 CMyIOPlugin 类

整个 Plugin 的功能都在最重要的 Show 函数里面完成，其提供一个打开文件

对话框，供用户选择多个 BMP 文件，并打开这些文件，形成一个 Volume 传给 3DMed 去处理。因为此例子使用了 MITK，所以这里的代码很简单，直接使用了 MITK 中提供的 mitkBMPReader，整个 Show 函数的代码如下所示：

```
bool CMyIOPlugin::Show(void)
{
    //MFC 的规定，必须先调用这个宏.
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    COpenFileDialog fileDialog;
    fileDialog.SetTitle("打开序列 BMP 文件");
    fileDialog.SetFilter("BMP 文件(*.bmp)\0*.bmp\0\0");
    fileDialog.EnableMultiSelect();

    // 如果用户选择了文件并点“确定”按钮.
    if(fileDialog.Run())
    {
        mitkBMPReader *aReader = new mitkBMPReader;

        //循环得到用户选中的每个文件名，
        //并将其加入到 Reader 中.
        POSITION pos = fileDialog.GetStartPosition();
        CString szFileName;
        while(pos != NULL)
        {
            szFileName = fileDialog.GetNextPathName(pos);
            aReader->AddFileName(szFileName);
        }

        //设置一下其它必要的信息.
        aReader->SetSpacingX(1.0f);
        aReader->SetSpacingY(1.0f);
        aReader->SetSpacingZ(1.0f);

        //调用 Reader 的实际读入程序.
        aReader->Run();

        //生成输出的数据.
```

```
        m_Data = new medVolume;
        m_Data->SetData(aReader->GetOutput());
        m_Data->SetFileName(szFileName);
        m_Data->SetName("My BMP Files");

        aReader->Delete();
        return true;
    }

    return false;
}
```

上面的代码加了大量的注释，这里就不再赘述了，重载完了 Show 函数以后，还剩下最后一件事情，就是加入一些导出函数，这一切只需调用下面的宏即可实现，其中第一个参数是我们所写的 Plugin 的类名，这里当然是 CMyIOPlugin 了，第二个参数是我们所希望在 3DMed 菜单里面所看到的此 Plugin 对应的菜单文字。

```
IMPLEMENT_VOLUME_IMPORT(CMyIOPlugin, "My BMP Plugin")
```

最后不要忘了在前面加上必要的头文件，如下所示：

```
//PluginsSDK 头文件
#include "medVolume.h"
//MITK 头文件
#include "mitkBMPReader.h"
#include "mitkVolume.h"
```

### 12.2.3 插入到 3DMed

经过了上面的步骤以后，现在可以编译整个工程，得到IOPlugin.dll文件，并将其拷贝到 3DMed的安装目录下的Plugins子目录里面，这时运行 3DMed主程序，3DMed加载完所有的插件以后，我们点击“文件”菜单下的“加载体数据”菜

单时，就会发现我们的“My BMP Plugin”也在其中，如 图 12-6 所示。当点击“My BMP Plugin”后，就会有打开对话框弹出来，让用户选择BMP文件，如所示。选择完以后这些数据就会在 3DMed中打开并显示，一切都如我们想像的一样。

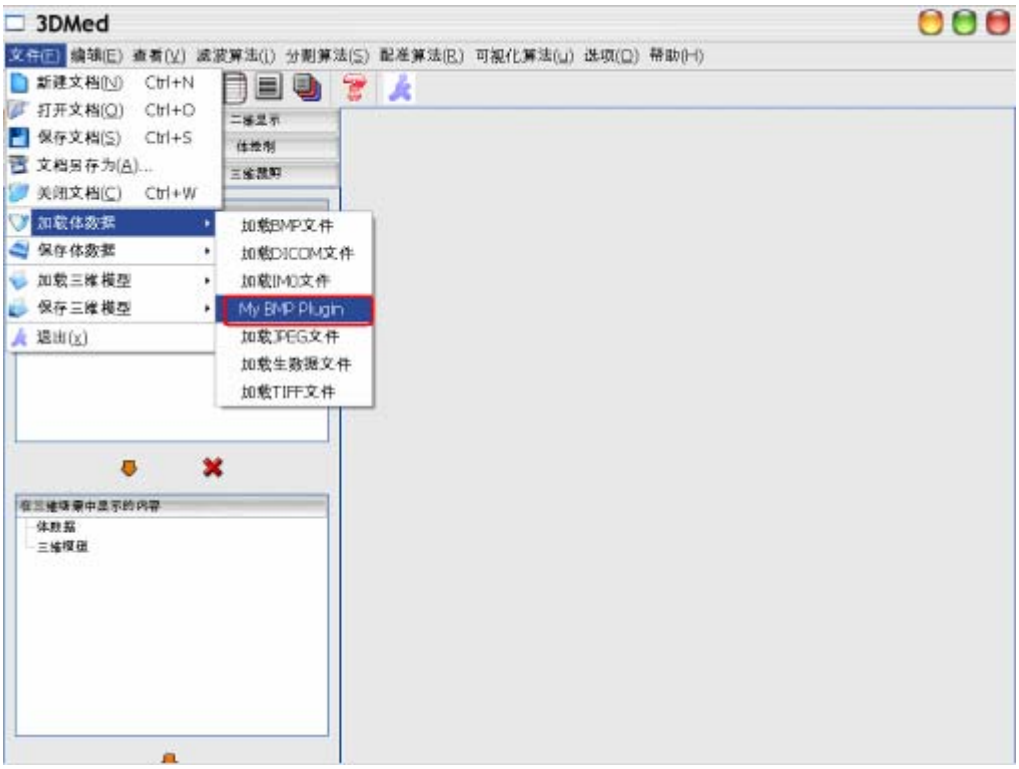



图 12-6 成功加载的 IOPlugin

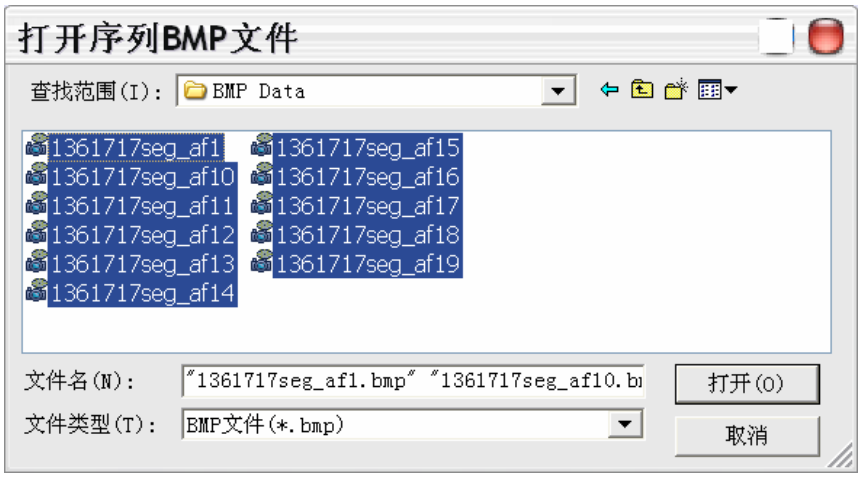



图 12-7 IOPlugin 运行界面

## 12.3Plugin 实例：不使用 MITK

在有些情况下，用户只对 3DMed 感兴趣，而并不想学习并使用 MITK，那么照样可以撰写 3DMed 的 Plugin。本节的实例就是演示如何在没有 MITK 的情况下，来编写有效的 Plugin 并集成到 3DMed 中去。本实例仍然使用 Microsoft Visual C++ 6.0 作为工具，从如何建立工程，到修改工程的设置，以及具体的编程实现，一直到最后集成进 3DMed，给出了完整的过程。为了清晰和简化起见，这个 Plugin 的功能比较简单，实现了对任意数据类型的 Volume 数据的域值分割算法，支持设置高域值和低域值，提供了一个简单的对话框来设置参数。

### 12.3.1 工程的建立及设置

在Microsoft Visual C++ 6.0 的IDE环境下，新建一个工程SegPlugin，工程的类型选择 “MFC AppWizard(dll)” （这里不再贴图，请参看 12.2.1 节中的相关截图），因为我们要使用MFC作GUI图形界面，并且目标是生成一个动态链接库。在接下来的一步中，选择DLL类型为“Regular DLL using shared MFC DLL”，此时可以选择“Finish”按钮完成工程的创建。

创建完工程以后，选“Projcet”－》“Settings”菜单，在弹出来的“Project Settings”对话框中，选择“C/C++”标签页，在“Category”列表框里面选择“Preprocessor”，然后在“Additional Include directories”编辑框中输入PluginsSDK头文件路径，如 图 12-8 所示。需要注意的是这里实际的路径依赖于你的机器上的PluginsSDK的放置路径，请根据实际情况设置。

设置完头文件路径以后，接下来要设置的是必要的导入库文件的路径。保持“Projcet Settings”对话框打开，选择“Link”标签页，在“Category”列表框中选择“Input”，然后在“Object/library modules”编辑框中输入“PluginsSDK.lib”，表示要使用这个导入库，并且还需要在“Additional library path”编辑框中输入PluginsSDK.lib所在的路径，如 图 12-9 所示。需要注意的是这里实际的路径依赖于你的机器上的PluginsSDK.lib的放置路径，请根据实际情况设置。

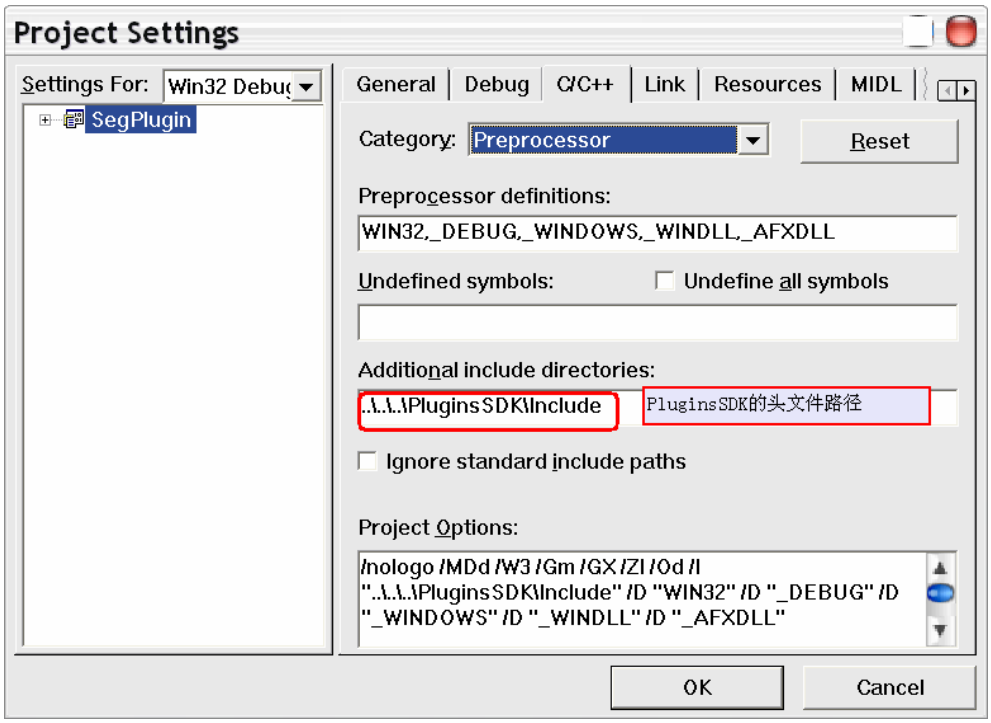



图 12-8 设置必要的头文件路径

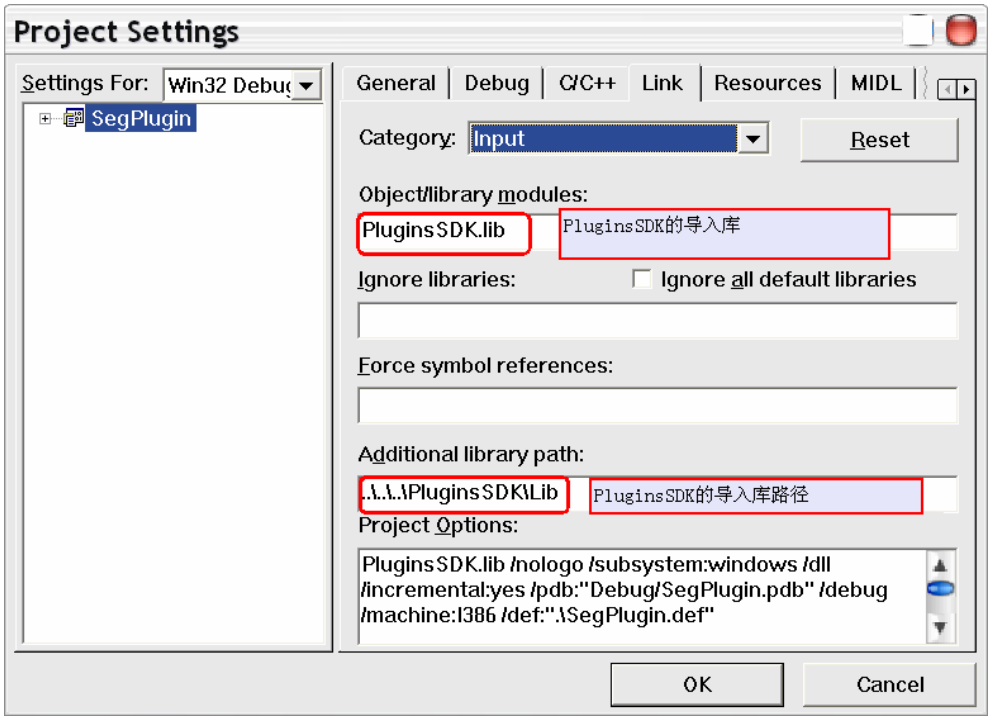



图 12-9 设置必要的库文件路径

### 12.3.2 实例制作

设置好烦琐的工程选项以后，就可以进入实质部分了。下面创建我们的Plugin的类，用“New Class”创建一个新类，并将其命名为CMySegPlugin，如 图 12-10 所示。因为我们的Plugin的功能是进行域值分割，所以很显然是属于SegmentationPlugin，故CMySegPlugin应该从SegmentationPlugin公有继承，并应该重载 Show 函数，其类声明如下所示：

```
class CMySegPlugin : public medSegmentationPlugin
{
public:
    CMySegPlugin();
    virtual ~CMySegPlugin();

    virtual bool Show(void);
private:
    bool doSegmentation(float lowThre, float highThre);
};
```

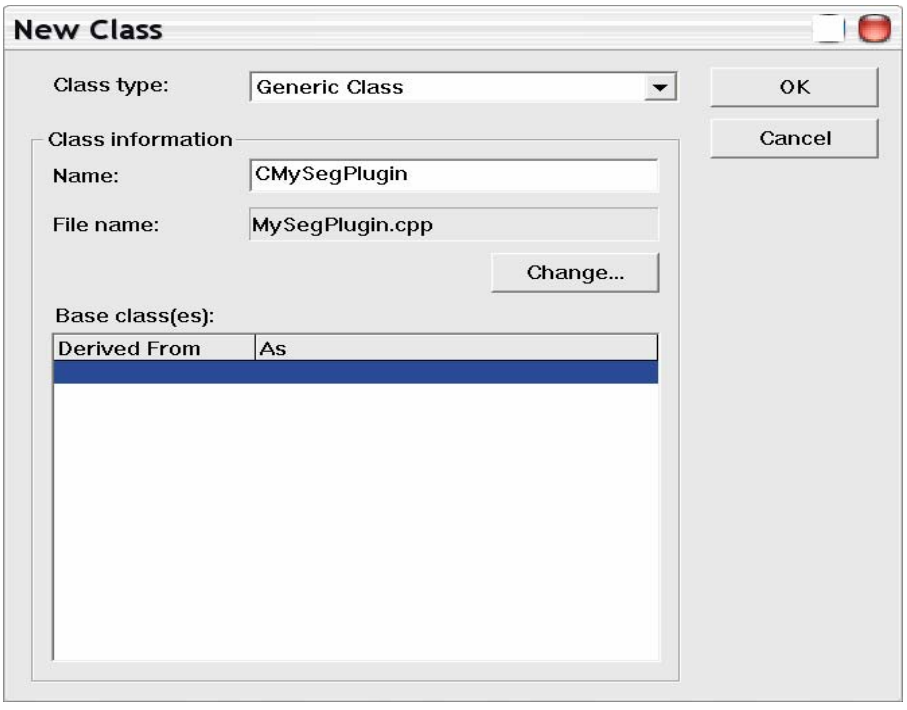



图 12-10 创建 CMySegPlugin 类

其中的私有成员函数 doSegmentation 用来完成实际的分割工作，其在 Show 函数里面被调用。

下面需要作的是GUI 的图形界面工作了，我们需要制作一个对话框，在“ResourceView”面板上点右键，插入一个新的对话框资源，在“Dialog Properties”对话框中将其ID设置为“IDD_DIALOG_PARAMETER”，Caption设置为“设置高低域值”，如 图 12-11 所示。然后在对话框上放置两个静态文本控件、两个编辑框控件，和“确定”、“取消”按钮，其界面如 图 12-12 所示。


图 12-11 对话框属性设置


图 12-12 对话框界面

界面创建完以后，使用ClassWizard为这个对话框创建一个新类，类的名字叫CDialogParameter，如 图 12-13 所示。然后再使用ClassWizard对话框中的“Member Variables”标签页，将对话框中的两个编辑框设置成CDialogPapameter类的成员变量，类型均为float型，分别如 图 12-14 和 图 12-15 所示。

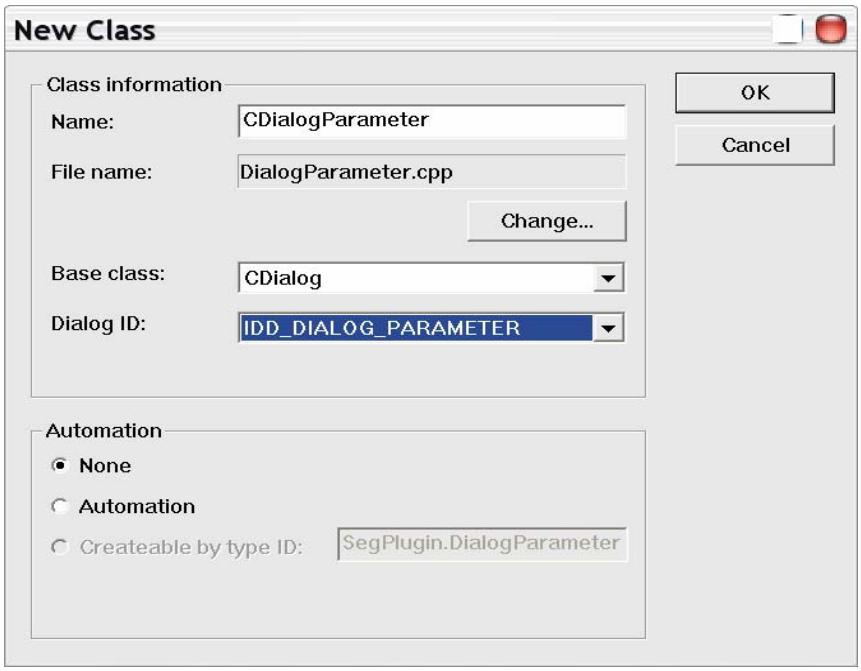



图 12-13 创建 CDialogParameter 类

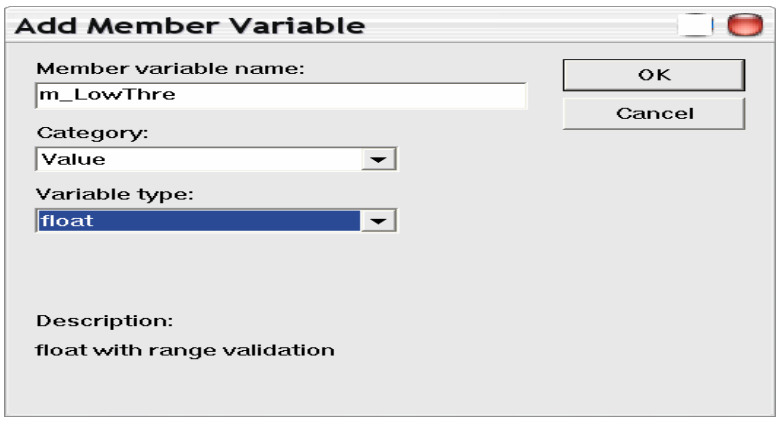



图 12-14 添加低域值成员变量

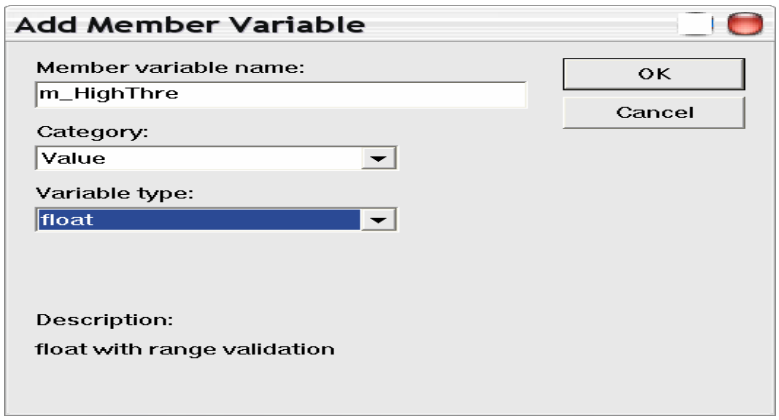



图 12-15 添加高域值成员变量

最后剩下的是添加成员函数来读取和设置高、低域值这两个成员变量了，分别是 SetLowThre、SetHighThre、GetLowThre 和 GetHighThre 函数，整个 CDialogParameter 的程序代码如下所示：

```
class CDialogParameter : public CDialog
{
// Construction
public:
    CDialogParameter(CWnd* pParent = NULL);   // standard constructor

    void SetLowThre(float lowThre);
    void SetHighThre(float highThre);

    float GetLowThre();
    float GetHighThre();

// Dialog Data
    //{{AFX_DATA(CDialogParameter)
    enum { IDD = IDD_DIALOG_PARAMETER };
    float  m_LowThre;
    float  m_HighThre;
    //}}AFX_DATA
```

```
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDialogParameter)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CDialogParameter)
        // NOTE: the ClassWizard will add member functions here
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

终于作完了烦琐的 GUI 图形界面工作，也有了 CDialogParameter 的辅助，下一步就是实现最重要的 CMySegPlugin 类中的 Show 函数了，其实现代码如下所示：

```
bool CMySegPlugin::Show(void)
{
    //MFC 的规定，必须先调用这个宏.
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    //弹出域值选择对话框.
    CDialogParameter paraDialog;
    paraDialog.SetLowThre(0.0f);
    paraDialog.SetHighThre(255.0f);

    if(paraDialog.DoModal() == IDOK)
    {
        //作实际的域值分割工作.
        return this->doSegmentation(paraDialog.GetLowThre(),
                                    paraDialog.GetHighThre());
```

```
    }

    return false;
}
```

里面用到了我们刚完成的 CDialogParameter 类，弹出一个域值设置对话框，如果用户设置完高、低域值并选择了“确定”以后，Show 函数内部调用自己的私有成员函数 doSegmentation，把用户设置的高、低域值传进来，具体的工作由 doSegmentation 函数来完成，其实现代码如下所示：

```
bool CMySegPlugin::doSegmentation(float lowThre, float highThre)
{
    if(lowThre >= highThre)
        return false;

    medVolume *inVolume = this->GetInput();

    if(inVolume == NULL)
    {
        AfxMessageBox("输入数据为空");
        return false;
    }

    if(inVolume->GetNumberOfChannel() != 1)
    {
        AfxMessageBox("对不起，只支持单通道的数据");
        return false;
    }

    //生成输出数据(即分割后数据).
    m_OutData = new medVolume;


    //设置分割后数据的属性，
    //和原始数据大部分一致.
    medVolume *outVolume = this->GetOutput();
    outVolume->SetWidth(inVolume->GetWidth());
```

```
    outVolume->SetHeight(inVolume->GetHeight());
    outVolume->SetImageNum(inVolume->GetImageNum());
    outVolume->SetSpacingX(inVolume->GetSpacingX());
    outVolume->SetSpacingY(inVolume->GetSpacingY());
    outVolume->SetSpacingZ(inVolume->GetSpacingZ());
    outVolume->SetNumberOfChannel(inVolume->GetNumberOfChannel());

    //分割后的数据为二值数据,
    //因此这里将数据类型设置为 unsigned char（8bit）.
    outVolume->SetDataType(MED_UNSIGNED_CHAR);

    //为分割后的数据分配实际内存.
    unsigned char *outData = (unsigned char*) outVolume->Allocate();

    //得到输入数据的内存指针.
    void *inData = inVolume->GetRawData();

    //根据输入数据的类型,
    //使用模板函数来完成实际分割过程.
    switch(inVolume->GetDataType())
    {
        case MED_CHAR:
            t_ExecuteSegmentation((char*) inData, outData, inVolume,
                                  lowThre, highThre);
            break;

        case MED_UNSIGNED_CHAR:
            t_ExecuteSegmentation((unsigned char*) inData, outData,
            inVolume, lowThre, highThre);
            break;

        case MED_SHORT:
            t_ExecuteSegmentation((short*) inData, outData, inVolume,
                                  lowThre, highThre);
            break;

        case MED_UNSIGNED_SHORT:
            t_ExecuteSegmentation((unsigned short*) inData, outData,
            inVolume, lowThre, highThre);
            break;
```

```
        case MED_INT:
            t_ExecuteSegmentation((int*) inData, outData, inVolume,
                                  lowThre, highThre);
            break;

        case MED_UNSIGNED_INT:
            t_ExecuteSegmentation((unsigned int*) inData, outData,
            inVolume, lowThre, highThre);
            break;

        case MED_LONG:
            t_ExecuteSegmentation((long*) inData, outData, inVolume,
                                  lowThre, highThre);
            break;

        case MED_UNSIGNED_LONG:
            t_ExecuteSegmentation((unsigned long*) inData, outData,
            inVolume, lowThre, highThre);
            break;

        case MED_FLOAT:
            t_ExecuteSegmentation((float*) inData, outData, inVolume,
                                  lowThre, highThre);
            break;

        case MED_DOUBLE:
            t_ExecuteSegmentation((double*) inData, outData, inVolume,
                                  lowThre, highThre);
            break;

        default:
            AfxMessageBox("不支持的数据类型!");
            return false;
    }

    return true;

}
```

由于这个例子并不使用MITK，因此许多底层细节必须处理，所以程序稍微烦琐。为简化起见，这个例子只处理单通道的数据，如果是多通道的数据，可以先经过预处理，转换成单通道的数据。另外，这个地方大量地使用了medVolume所提供的API函数，所以请参考 12.1 节的介绍。在整个代码中，因为要处理不同数据类型的输入数据，所以最烦琐的地方在那个大的Switch语句中，判断输入数据的类型，并据此将输入数据的内存指针强制转换成对应的指针类型，交给模板函数t_ExecuteSegmentation来处理，而t_ExecuteSegmentation的第一个参数是模板参数，可以被实例化成各种不同的指针类型，这也大大简化了编程的工作量。注意的是t_ExecuteSegmentation并不是CMySegPlugin的成员函数，而只是一个普通函数，所以其必须在doSegmentation函数前面被声明，其整个实现代码如下所示：

```
template <class T>
void t_ExecuteSegmentation(T *inData, unsigned char *outData,
                           medVolume *inVolume,
                           float lowThresh, float highThresh)
{
    //得到图像的一些属性信息.
    int imageWidth = inVolume->GetWidth();
    int imageHeight = inVolume->GetHeight();
    int imageNum = inVolume->GetImageNum();
    int i, j, k;

    //循环遍历整个数据.
    for(k = 0; k < imageNum; k++)
    {
        for(j = 0; j < imageHeight; j++)
        {
            for(i = 0; i < imageWidth; i++)
            {
                //如果数据值在指定的域值范围内，
                //则输出的数据二值化为 255.
                if(inData[0] >= lowThresh && inData[0] <= highThresh)
                {
                    outData[0] = 255;
                }
```

```
                //否则为 0.
                else
                {
                    outData[0] = 0;
                }

                //输入数据和输出数据指针前移.
                inData++;
                outData++;
            }
        }
    }
}
```

至此为止，已经完成了整个程序中最艰苦的部分，下面就是一些例行工作了，首先是加入必要的导出函数，这一切只需调用下面的宏即可实现，其中第一个参数是我们所写的 Plugin 的类名，这里当然是 CMySegPlugin 了，第二个参数是我们所希望在 3DMed 菜单里面所看到的此 Plugin 对应的菜单文字。

```
IMPLEMENT_SEGMENTATION (CMySegPlugin, "My Segmentation Plugin")
```

接着是在前面加上必要的头文件，如下所示：

```
//PluginsSDK 头文件
#include "medVolume.h"
//对话框头文件
#include "DialogParameter.h"
```

### 12.3.3 插入到 3DMed

经过了上面的步骤以后，现在可以编译整个工程，得到SegPlugin.dll文件，并将其拷贝到 3DMed的安装目录下的Plugins子目录里面，这时运行 3DMed主程序，3DMed加载完所有的插件以后，我们点击“分割算法”菜单时，就会发现

我们的“My Segmentation Plugin”也在其中，如 图 12-16 所示。当点击“My Segmentation Plugin”后，就会弹出“设置高低域值”对话框，让用户设置参数，如 图 12-17 所示，设置完并点“确定”按钮以后，3DMed将利用分割后的数据进行三维重建并显示。

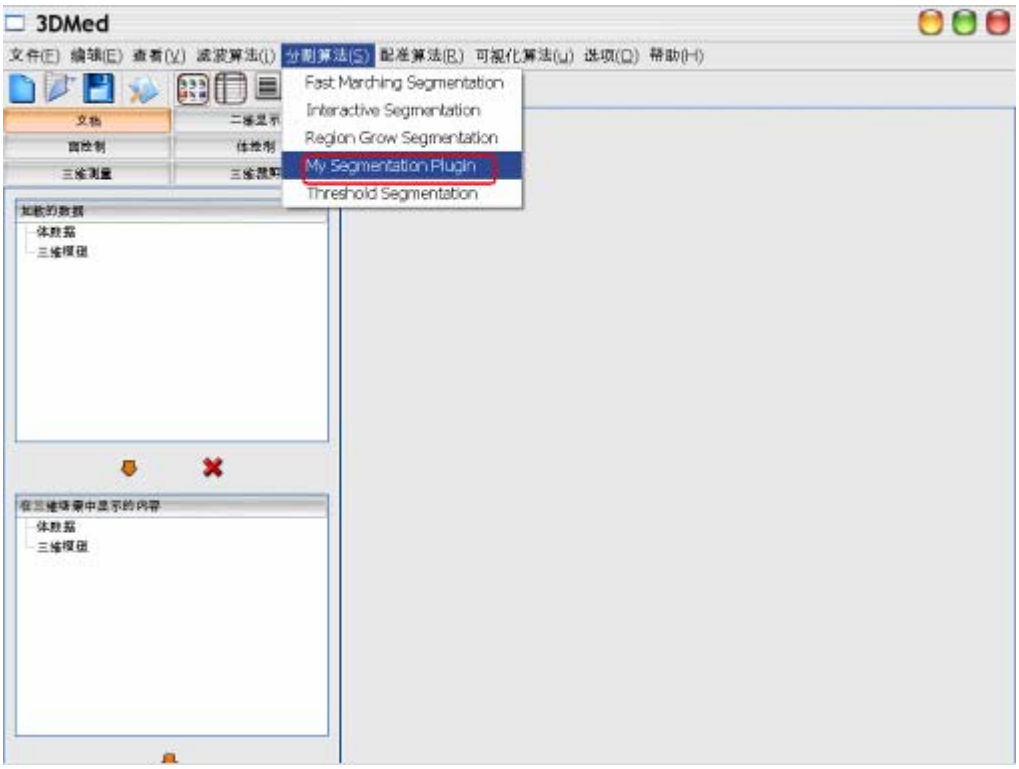



图 12-16 成功加载的 SegPlugin


图 12-17 SegPlugin 的运行界面

## 12.4 小结

本章给出了如何开发 3DMed 的 Plugin，从而扩充 3DMed 的功能以满足自己的实际使用需求。开发 3DMed Plugin 有两种途径，一种是使用 MITK 作为底层算法库，这种方式可以充分利用 MITK 提供的功能和各种算法，比较简单；另外一种不使用 MITK，直接从底层开始写自己的算法，这种适用于在用户手上没有 MITK 的情况下扩充 3DMed 的功能，这种方式要处理一些底层细节，因此相对于第一种方法来说，稍微复杂一点。

本章的两个例子分别演示了在这两种情况下如何撰写自己的 3DMed Plugin 并将其集成到 3DMed 中，这两个例子也很好地说明了 3DMed 对这两种方法都支持的很好。用户可以根据自己的实际情况，根据本章提供的例子，来编写 3DMed 的 Plugin，从而不断地提升 3DMed 的功能。

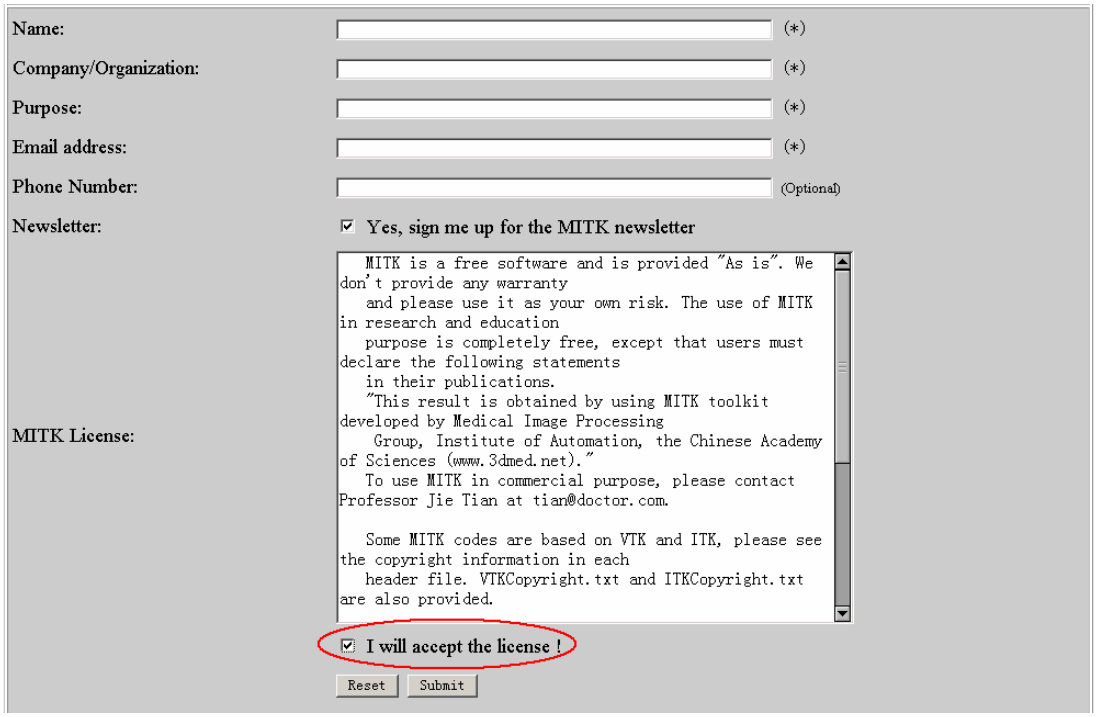
# 附录A　医学影像数据集

1. http://www.volren.org/ ，提供很多断层成像数据集
2. http://radiology.uiowa.edu/downloads/ ，爱荷华大学放射系提供的多套三维医学影像数据集
3. http://graphics.stanford.edu/data/voldata/ ，斯坦福大学图形学实验室提供的一些三维数据集
4. http://www.nlm.nih.gov/research/visible/visible_human.html ，美国虚拟人体数据集
5. http://www.psychology.nottingham.ac.uk/staff/cr1/ct.zip ，头部CT数据
6. http://www.psychology.nottingham.ac.uk/staff/cr1/mricro.html ，MRICro软件自带的标准的脑部数据

# 附录B　MITK 网站介绍

一、如何获取 MITK 和 3DMed？

登录 http://www.mitk.net，在Download页面（http://www.mitk.net/download.htm）下，点击你所需下载的项目，然后会弹出一个页面（如下图所示），要求输入一些必要的用户信息，包括姓名（Name）、单位（Company/Organization）、使用目的（Purpose）、电子邮件地址（Email address）、电话号码（Phone Number），其中前 4 项是必填项。下面是一个询问是否加入邮件通讯列表的选择项，如果选择加入，就可以在第一时间收到我们的更新信息。最后在接受许可协议（“I will accept the license!”）前打勾并按“Submit”按钮提交注册单就可以正常下载了。必须提醒一句，目前MITK和 3DMed只是对教育和研究目的免费发放，切勿在未经许可的情况下用于商业目的，违者必究！




二、如何得到技术支持？

由于 MITK 和 3DMed 均为免费软件，所以我们通过 Web 方式提供技术支持。我们设置了一个 MITK 论坛，对用户的疑问、意见以及建议作出回应，并不定期地发布一些教程和更新信息，以满足用户需求。同时 MITK 论坛还设置了许多技术交流的版块，为 MITK 和 3DMed 的用户以及医学影像处理领域的研究人员提供一个理论和技术交流的场所，并希望以此来推动 MITK 和 3DMed 的进一步发展。

目前 MITK 论坛对普通用户也是开放的，不用注册即可在论坛提问或发表看法，但是注册会员比普通用户享有更多的权利，比如即时从论坛得到一些重要信息的邮件通知和获得以附件方式提供的更新程序等。

论坛的网址是 http://www.mitk.net/forum/。

三、如何成为 MITK 论坛的注册会员？

登录 http://www.mitk.net/forum/，进入“注册”页面，填写一些必要信息（如下图所示），其中，必填项中“会员名”最长为 25 个字符（中文每字按 2 字符记），“密码”最长为 32 个字符。填完后按注册信息表下方的“提交”按钮，若“会员名”和“电子邮件地址”未与已注册会员重复，即可完成注册，否则，请更换“会员名”或“电子邮件地址”重新注册。


四、如何在论坛上获得帮助？

在使用 MITK 和 3DMed 软件的过程中遇到任何问题都可以到 MITK 论坛上寻求帮助，如何快速而有效的获得答案或帮助，与提问的方式有很大关系。

首先，我们建议您在提问之前先尝试在我们提供的手册和帮助文档中寻找答案，这些资源包含在MITK网站的“Documentation”页面（http://www.mitk.net/document.htm）和您所下载的软件包中。如果您所提的问题可以很容易的在这些文档中找到答案，您多半也只会得到“参见××手册（或文档）”之类的回答；

其次，在所提的问题中应尽可能将您所遇到的问题描述清楚，比如当您运行程序时遇到异常退出的情况（通常由程序的 bug 引起）而自己又无法解决时，您在所提的问题中应当尽量包括如下一些内容：

（1） 出错情况的描述，包括出错的提示信息，当时程序的运行参数等；

（2） 出错程序所读取的外部数据的信息，包括数据来源、格式、基本参数等，最好能提供原始数据文件；

（3） 出错程序的运行环境，主要包括操作系统环境及计算机的硬件配置等。

另外，如果是 3DMed 运行时的错误，提供一张出错界面的截图也是一个不错的选择；

第三，寻找合适的版面提出自己的问题，比如将上述关于程序出错的问题发到“Bug Report”版，而将关于 MITK 中某个类使用方法的疑问发到“MITK”版。我们在论坛设立了许多主题版面，包括一些技术交流版面，找到合适的板面提出你的疑问可以确保您在尽量短的时间内得到回应。

同时，我们也欢迎您在 MITK 论坛发表您的意见建议、心得体会以及交流一些与医学影像处理相关的技术问题等，希望这个论坛能成为一个我们共同拥有的技术交流园地。