VTK 用户手册

翻译整理： 陈恒

甘肃省科学院自动化研究所

文档控制

变更记录

日期	作者	版本	变更说明
20090408	陈恒		**初版**
20090408	**陈恒**	**1.0**	**发布**

审　　核（此处必须手签）

日期	审核人	单位 / 职务

批　　准（此处必须手签）

日期	批准人	单位 / 职务

安全控制

密级说明	传阅范围	修改范围
	实验室内部	

归　　档

归档日期	归档人	档案管理员
20090410	**陈恒**	

目　　录

1　系统概述 1

1.1　系统架构 1

1.1.1　图形模型 2

1.1.2　可视化模型 3

1.2　创建应用程序 8

2　VTK 使用基础 20

2.1　创建一个简单的示例 20

2.2　使用 VTK 交互功能 22

2.3　过滤器 24

2.4　控制相机 25

2.4.1　创建相机 26

2.4.2　简单的操作方法 26

2.4.3　控制观察方向 27

2.4.4　透视及正交投影 27

2.5　控制光源 27

2.6　控制场景中的物体（3D PROPS） 28

2.6.1　指定物体的空间位置 28

2.6.2　Actros 29

2.6.3　物体的层次细节 30

2.6.4　组装 30

2.6.5　使用纹理 31

2.6.6　拾取 33

2.7　VTK 中的坐标系统 35

2.8　VTKACTOR2D 36

2.9　注释 36

2.9.1　2D Annotation 36

2.9.2　3D Annotation and vtkFollower 38

2.10　特定绘图 38

2.10.1　颜色图例 39

2.10.2　绘制平面图表 39

2.10.3　显示物体的边界尺寸 40

2.10.4　标识属性数据 42

2.11　数据变换 43

3　可视化技术 44

3.1　可视化 VTKDATASET 类数据 44

3.1.1　颜色映射 46

3.1.2　轮廓提取 49

3.1.3　符号化 51

3.1.4　流线技术 53

3.1.5　流面 57

3.1.6 切割.....59
3.1.7 数据合并.....61
3.1.8 附加数据（Appending Data）.....62
3.1.9 探查(Probing).....62
3.1.10 为等值面分级着色.....65
3.1.11 提取单元数据的子集.....67
3.1.12 提取单元作为多边形数据(vtkPolyData).....69
3.2 可视化多边形数据.....72
3.2.1 手动创建 vtkPolyData.....72
3.2.2 生成表面法线.....75
3.2.3 多边形消减技术.....76
3.2.4 平滑网格 Smooth Mesh.....79
3.2.5 裁减数据.....80
3.2.6 创建纹理坐标.....83
3.3 可视化结构网格数据集.....85
3.3.1 手动创建结构化网格数据集.....85
3.3.2 提取计算平面.....87
3.3.3 对结构化网格数据二次采样.....89
3.4 可视化线性网格数据.....90
3.4.1 手动创建线性网格数据.....90
3.4.2 提取计算平面.....91
3.5 可视化非结构网格数据.....92
3.5.1 手动创建非结构化网格数据.....92
3.5.2 提取部分网格数据.....94
3.5.3 对 vtkUnstructuredGrid 提取轮廓值.....96
4 可视化图像及三维体数据.....96
4.1 VTKSTRUCTUREDPOINTS 数据的发展历史.....97
4.2 手动创建 VTKIMAGEDATA 数据.....98
4.3 图像数据的二次采样.....99
4.4 二维图像的三维显示.....102
4.5 体绘制.....103
4.5.1 一个简单的例子.....104
4.5.2 为什么有多种体绘制技术.....106
4.5.3 创建 vtkVolume 类.....107
4.5.4 使用 vtkPiecewiseFunction 类.....107
4.5.5 使用 vtkColorTransferFunction 类.....109
4.5.6 使用 vtkVolumeProperty 类设定透明度和颜色值.....109
4.5.7 使用 vtkVolumeProperty 控制阴影.....110
4.5.8 创建 vtkVolumeMapper 映射器.....112
4.5.9 切割体数据.....113
4.5.10 剪取体数据.....114
4.5.11 用射线投射法进行体绘制.....115
4.5.12 二维纹理映射法体绘制.....117

5 VTK 数据接口对象 117

5.1 数据数组 118
5.2 数据集对象 121
5.3 VTKDATASET 数据接口 122
5.4 VTKIMAGEDATA 类的数据接口 123
5.5 VTKPOINTSET 的数据接口 124
5.6 VTKPOLYDATA 的数据接口 125
5.7 VTKCELL 的数据接口 127

6 建立模型 127

6.1 隐模型 127
6.1.1 定义隐函数 127
6.1.2 对隐函数采样 129
6.2 挤压 132
6.3 表面重构 133
6.3.1 Delaunay 三角网 133
6.3.2 高斯抛雪球 139
6.3.3 杂乱点集构建表面 141

7 与视窗系统交互 142

7.1 VTKRENDERWINDOW 交互类型 142
7.2 交互方针 143
7.3 在 WINDOW 系统/VC++中使用 VTK 进行交互 144

8 VTK 对象说明 144

8.1 对象结构图 144
8.1.1 基础对象 145
8.1.2 单元对象 145
8.1.3 数据集对象 146
8.1.4 可视化流水线对象 146
8.1.5 源对象 146
8.1.6 过滤器 147
8.1.7 映射器 148
8.1.8 图形对象 149
8.1.9 体绘制 150
8.1.10 图像处理对象 151
8.1.11 OpenGL 绘制对象 152
8.1.12 拾取对象 153
8.1.13 变换对象层次图 153
8.2 过滤器 154
8.2.1 源对象 154
8.2.2 图像过滤器 155
8.2.3 可视化过滤器 158
8.2.4 映射器对象 162
8.2.5 角色对象 163

9 可视化流水线 164

9.1 概述 164
9.1.1 数据可视化示例 164
9.1.2 功能模型 165
9.1.3 可视化模型 166
9.1.4 对象模型 166
9.2 可视化流水线 166
9.2.1 数据对象 166
9.2.2 过程对象 166
9.3 流水线拓扑结构 167
9.3.1 流水线的连接 167
9.3.2 循环机制 168
9.4 流水线执行 168
9.5 数据接口 169
9.6 综合应用 170
9.6.1 隐含控制执行 170
9.6.2 多输入输出 171
9.7 可视化流水线示例 172
9.7.1 简单球体 172
9.7.2 弯曲球体 173
9.7.3 符号化处理 174
9.7.4 隐藏球体 175

10 基本数据表达 177

10.1 可视化数据的特点 177
10.2 数据对象 177
10.3 数据集 178
10.3.1 单元 178
10.3.2 属性数据 183
10.4 数据集可视化 186
10.4.1 数据集类型 186
10.4.2 数据可视化方法 189

11 功能算法 194

11.1 概述 194
11.2 标量算法 195
11.2.1 颜色映射 195
11.2.2 提取轮廓 196
11.2.3 标量数据的确定 197
11.3 矢量算法 197
11.3.1 方向线和方向符号 198
11.3.2 变形 198
11.3.3 位移绘制 199
11.3.4 时间动画 200

11.3.5 流线.....*200*
11.4 模型算法.....201
11.4.1 源对象.....*201*
11.4.2 隐函数.....*202*
11.4.3 隐式建模.....*203*
11.4.4 符号化.....*203*
11.4.5 剪切.....*204*
11.5 综合应用.....204
11.5.1 过程对象设计.....*204*
11.5.2 颜色映射.....*206*
11.5.3 隐函数.....*207*
11.5.4 提取轮廓.....*207*
11.5.5 剪切.....*208*
11.5.6 符号化.....*208*
11.5.7 流线.....*208*
11.5.8 抽象过滤器.....*209*

1 系统概述

本章的主要目的对 VTK 系统进行概括性的介绍，并且提供了一些基本的信息，让用户能用各种开发语言（C++、TCL、Java、Python）创建应用程序，在本章开始的时候，主要介绍 VTK 系统的基本概念和对象模型，在本章的最后，用一些示例程序说明如何用 VTK 构建应用系统。

1.1 系统架构

VTK 系统主要由 C++类库、解释包装层（对 C++类库进行包装，便于 TCL、Java 等语言使用）两个基本子系统构成，见图 3-1。

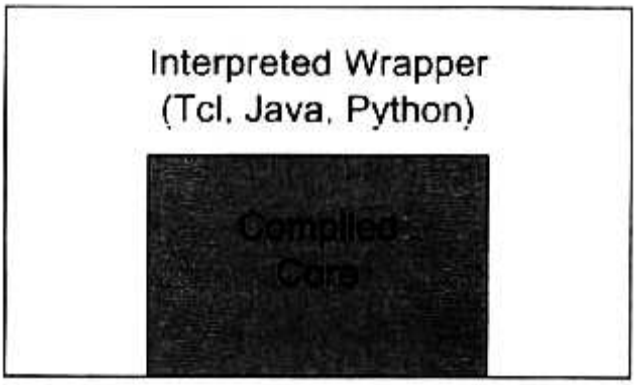


图 1-1 VTK 组成

采用这种架构的优势是我们能使用 C++语言建立高效的算法，用其他的脚本语言（如 TCL、Python）可以进行快速的开发，当然，如果精通 C++，也可以用 C++语言建立应用程序。

VTK 是一个基于面向对象的系统，提高 VTK 开发效率的关键因素是建立一个好的、易于理解的、优化的对象模型，我们根据对象模型，很容易把各种对象组合起来，构建应用程序，在本手册中我们尽量提供一些对用户有帮助的示例程序，在示例程序中介绍如何应用对象构建可视化程序，最好的办法是用户可以通过示例代码或网站获取更多的 VTK 对象功能描述信息。

在本章的剩下部分，介绍构成 VTK 系统的两种对象模型：图形模型和可视化模型，这部分的内容相对来说，是比较高级的内容，建议你在看这部分内容时，多看示例程序的代码。

1.1.1 图形模型

VTK 图形模型由以下核心类组成。

- vtkActor、vtkActor2D、vtkVolume
- vtkLight
- vtkCamera
- vtkProperty,vtkProperty2D
- vtkMaper,vtkMapper2D - vtkAbstractMapper 的子类
- vtkTransform
- vtkLookupTable,vtkColorTransferFunction- vtkScalarsToColors 的子类
- vtkRender
- vtkRenderWindow
- vtkRenderWindowInteractor

注意：这里并不是一个完整的包含所有类的列表，只是列出了最常用的类。

Props 表示在场景中能“看到”的东西(如场景中的地形、山脉、人物、花草等)，如果 Props 是 vtkProp3D 类型，它能操作 3D，如果 Props 是 vtkProp2D 类型，它能表现 2D 数据，Props 不能直接表现几何数据，在 VTK 中用 Mapper 表现几何数据，Props 能够引用 Property（属性)对象，Property 对象能够控制 Pros 的外观(如颜色、灯光、显示模式等)，在 Actors、Volumes 类中也有一个控制 Props 位置、方向的内部变换对象（vtkTransform)，vtkActor 是 vtkProper3D 的一个子类。

Lights(vtkLight)是用于表现和处理光线的，Lights 只用于三维，在二维中我们不使用光线。

相机类（vtkCamera）在绘制过程中，用相机类控制被绘制到场景中的物体的数量，相机类提供了一些方法控制相机的位置，同时相机类也控制透视投影和立体观察。

映射器类（vtkMapper）经常和查找表类（vtkLookupTable）一起对几何体进行变换和绘制，该类也是图形模型和可视化流水线之间的接口。

查找表类（vtkLookupTable）和颜色变换类（vtkColorTransferFunction）是 vtkScalarsToColors 类的子类，主要为映射的数据赋予不同的颜色。

绘制类（vtkRenderer）和绘制窗口类（vtkRenderWindow）管理图形引擎和计算机窗口系统之间的接口，绘制窗口是绘制图形的显示区域，多个绘制图形也许显示在一个绘制窗口

中，但是，用户可以同时创建多个绘制窗口，绘制数据在绘制窗口显示的区域被称为视口，一个绘制窗口可以同时有多个视口。

一旦在绘制窗口中绘制了图形对象，你就有了与图形对象交互的机会，VTK 提供了多个方法来与图形对象交互，其中的一个类是 vtkRenderWindowInteractor，这是一个操作相机对象和拾取工具的类，调用用户定义方法，进入/退出立体视角，并且修改 Actor 的一些属性。

上述的许多对象均有子类，例如，vtkAssembly、vtkFollower 和 vtkLODActor 都是 vtkActor 的子类，vtkAssembly 类用于将多个角色对象相互组合，构成一个复杂的对象实体，vtkFollower 是一个用于总是面对一个特定 camera(对布告板和文字非常有用) 的演示者，vtkLODActor 类用于在不同的层次表现物体细节的程度。

1.1.2 可视化模型

图形模型的主要作用是用图形描述几何体构成的场景，可视化流水线的主要作用是把几何数据（如立方体的顶点坐标）转换成图形数据和负责构建几何体，VTK 使用数据流的方式把几何体数据转换成图形数据，主要有两个基本类和数据转换相关，它们是：

- vtkDataObject

数据对象表达各种类型的数据，vtkDataObject 可以被看作是一个二进制大块（blob）数据，结构化的数据可以被认为是一个数据集(dataset) (vtkDataSet 类)。

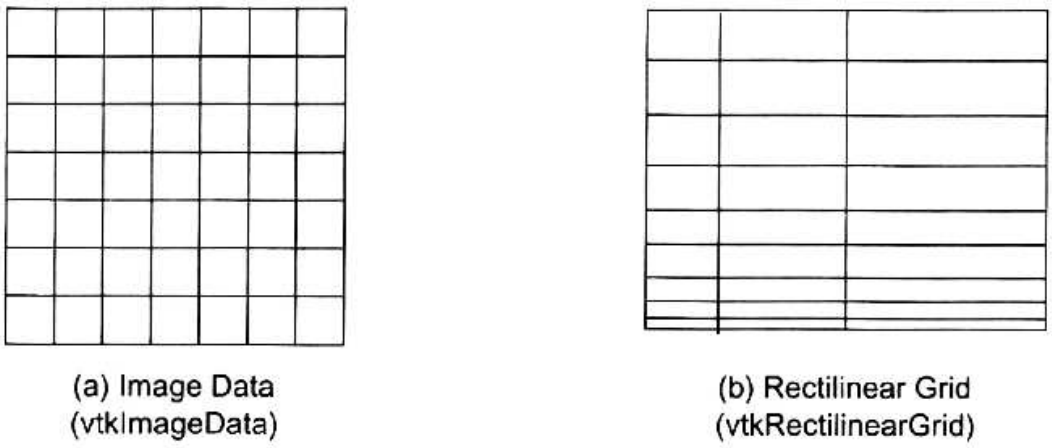


图 1-2 VTK 支持的数据对象

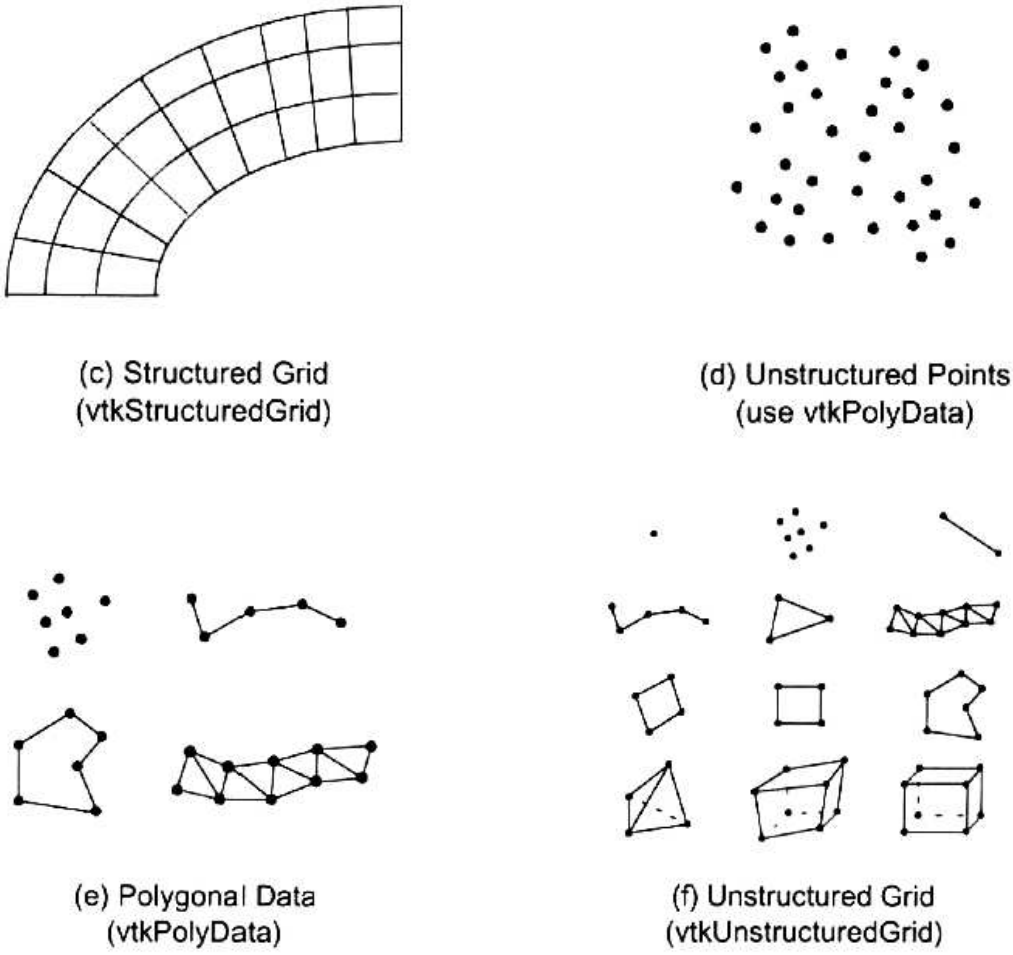


图 1-2　VTK 中的数据对象类型

a 均匀网格　b 线性网格　c 结构网格　d 离散点　e 多边形　f 非结构网格

数据对象由几何和拓扑结构组成（点集和单元集），同时数据对象与属性数据（例如缩放比例和向量）相关，这些属性数据用于描述数据对象中的点集和单元集的属性（对于一个办公桌而言，描述桌面的颜色、描述桌腿的颜色），单元集是数据对象的基本组成单位（如一个立方体对象由若干个三角形构成,这些三角形就是单元集,也可以称为图元),图 1-3 显示了 VTK 支持的各种属性数据。

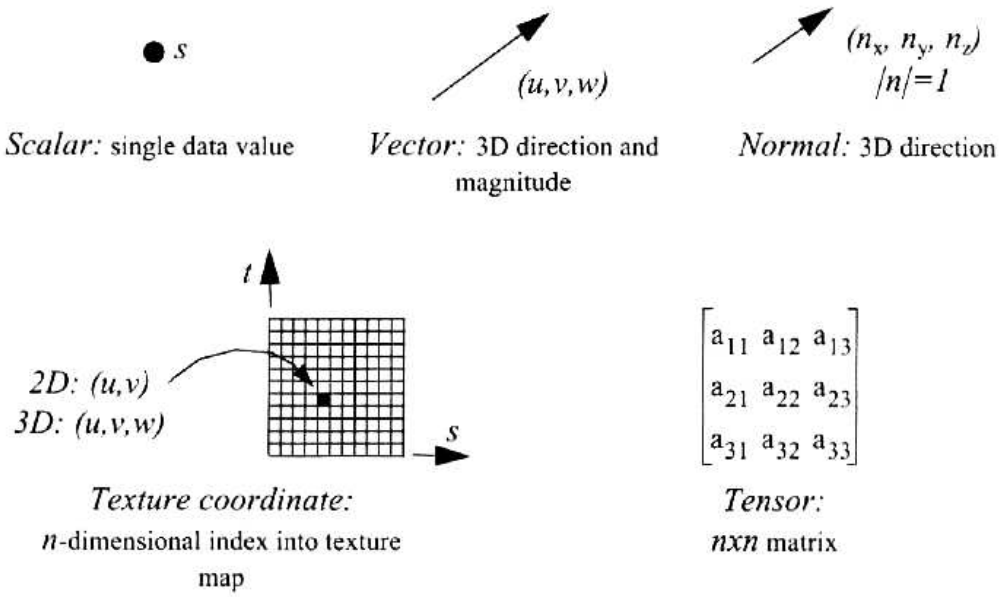


图 1-3　VTK 中用于描述单元集和点集的属性数据

- vtkProcessObject

过程对象一般也称为过滤器，按照某种运算法则对数据对象进行处理，对数据对象的数据进行优化，过程对象表现系统中的几何形状，数据对象和过程对象连接在一起形成可视化流水线(例如，数据流网络)。图 1-4 是一种可视化流程的描述。

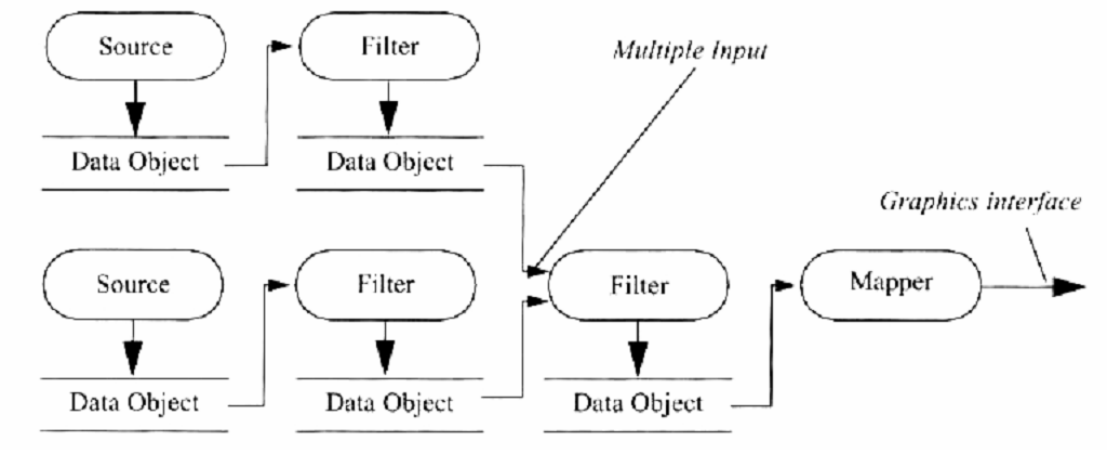


图 1-4 数据对象和过程对象连接创建可视化流水线

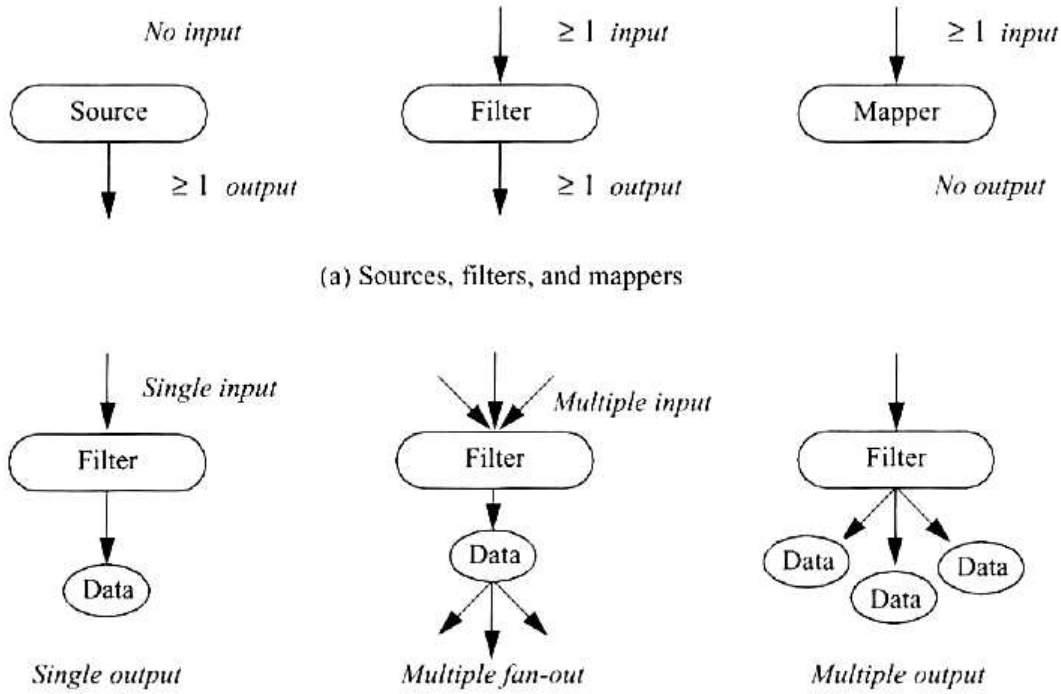


图 1-5　不同类型的过程对象和输入、输出之间的关系

图 1-4 与图 1-5 一起说明了一些重要的可视化概念，在可视化流水线中，主要包括源对象、过滤器对象、映射器对象三种对象。

源对象是可视化流水线的起点，源对象依据数据生成的方式，分为如下两种类型：

- 过程源对象

可以通过读数据文件的方式（读对象）产生数据。

- 程序源对象

也可以用数学表达式或其它的数学方法产生数据。

过滤器对象接收一个或多个数据对象作为输入，对数据对象处理之后生成一个或多个数据对象作为输出。

映射器（Mapper）对象是可视化流水线的终点，是图形模型和可视化模型之间的接口，其主要作用是将数据对象转换成图形对象，然后由图形引擎绘制出来，复写器是映射器的一种类型，作用是将数据写入文件或流。

在创建可视化流水线时，有几个重要的问题需要引起我们的注意。

首先，流水线的拓扑可以使用多种方法生成，在这里：

aFilter->SetInput(anotherFilter->GetOutput());这里将 anotherFilter 过滤器的

输出设置为 aFilter 过滤器的输入(input),(有多个输入和输出系统的过滤器也有类似的输入输出设置方法)。

第二，我们必须有一种可以控制流水线执行的机制，及时更新所需的数据到流水线中进行处理，VTK 使用基于每个对象的内部修改时间的惰性评估模式（只在数据需要时运行）对数据更新。

第三，在流水线中只有输入、输出数据类型兼容的对象才可以用 SetInuput()和 GetOutput()方法组合，在 VTK 中，C++编译时对类型的兼容性进行强制检测（解释器如 Tcl 在运行时报错）。

最后，我们必须决定在数据在流水线中执行时，哪些数据对象需要进行缓存，哪些需要保留，因为图形（图像）处理（可视化）的数据量非常大，对图形(图像)的绘制效率有非常大的影响，VTK 提供了启用或停用缓存的方法，对于那些不能一次性读入内存的数据集，采用数据分片读取的方法将数据读入内存。

1. 可视化流水线执行

在前面的章节我们讨论了数据在流水线中执行时，需要对流水线的执行过程进行控制，在这一节中我们要对关于控制流水线执行过程的一些关键概念进行详细的说明。

在前面的章节中指出的，VTK 可视化流水线只有数据需要被执行时（惰性赋值）才执行，如果你实例化一个阅读器对象，通过调用 GetNumberOfPoints 方法获取这个阅读器中点的数量，阅读器会返回“0”，尽管这个数据文件中包含了数千个点。

下面显示的代码示例使用的语言是 Tcl：

```
vtkPLOT3Dreader reader
reader SetXYZFileName  “$VTK_DATA_ROOT/Data/combxyz.bin
[reader GetOutput] GetNumberOfPoints
```

然而，如果你加入 Update()方法

```
Reader->Update
[reader GetOutput] GetNumberOfPoints
```

现在阅读器对象会返回正确的数字，出现这样的原因是在第一个例子中 GetNumberOfPoints()方法没有被要求执行，对象简单的返回了当前为“0”的点数，在第二个例子中，这个 Update()方法强制了流水线的执行，因此强制了阅读器执行数据。

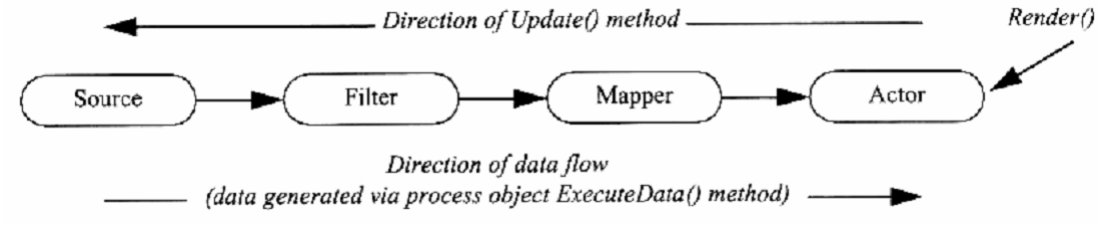


图 1-6 流水线执行概念总览

通常，你不必手动调用 Update(),因为过滤器连接到了一个流水线后，当 Actor 对象(角色）收到请求要对自己进行绘制时，角色对象将这个方法转发给和它相关的映射器，这时 Update()方法通过流水线自动被调用，流水线执行如图 1-6 所示，图中说明了 Render()方法发起对数据的请求后，数据通过流水线向左传递，一直传递到源对象(流水线的起点)，然后从源对象开始直到映射器对象,流水线中的每个对象依次检查需要处理的那部分数据是否已经过期，如果过期调用 Update()方法更新数据，这样保证流水线末端的数据总是最新的数据，接着 Actor 对象(角色)将绘制这些数据。

2. **图像处理**

VTK 提供了图像处理和体可视化的能力，在 VTK 中二维的图像数据和三维的体数据都表现为 vtkImageData 对象，VTK 中的图像数据是有序的，轴象对齐的数组数据，图像、像素映射和位图都是二维图像数据的例子，体数据是一种三维的图像数据对象。

有一些过程对象（过滤器）在图像流水线中总是输入和输出图像数据对象，由于图像数据的结构是规则排列的，使得图像流水线有另外一些重要的特性，体绘制用于可视化三维的 vtkImageData 数据，特殊的图像浏览器用于查看二维的 vtkImageData，图像流水线中几乎所有的过程对象是多线程的，并具有分片处理流数据的能力（为适应用户内存大小的限制）。过滤器能自动感知系统中有效的 CPU 数量，然后在运行期创建相应数量的线程，同时自动的通过流水线将流数据分片。

到这我们结束了 VTK 系统架构的概述，例子程序是学习 VTK 有效的途径，后面的章节用许多例子程序来演示 VTK 的各种各样的功能，当然，VTK 的例子程序的源码可以是可以下载得到的，源码里包含有数百个例子程序可供学习。

下面，我们介绍通过 C++,Tcl,Java 和 Python 创建应用程序的方法。

1.2 创建应用程序

本章包含了使用 4 种编程语言 Tcl,C++,Java 和 Python 来开发 VTK 应用程序所需的基本

知识及如何创建和运行简单的应用程序的方法，通过阅读这些介绍，你应该跳到你感兴趣使用的语言的子章节（在本节我们只讨论 VTK 在 windows 系统下的应用）。

1. 用户方法，观察者和命令集

回调（或用户方法）在 VTK 中使用 Subject/Observer 和 Command 设计模式来实现，这意味着每个 VTK 类（每个 vtkObject 的子类）都提供一个 AddObserver()方法建立回调机制，观察者监控一个对象所有被调用的事件，如果正在监控的一个事件被触发，一个与之相应的回调函数就会被调用。

举例说明：所有的 VTK 过滤器在开始运行前都要调用 StartEvent，如果你添加一个 observer 来监控 StartEvent，那么它就会在过滤器开始运行时被调用。

看看下面的 Tcl 脚本，创建了一个 vtkElevationFilter 实例，再增加一个对 StartEvent 的 observer 来调用过程 PrintStatus。

```
Proc PrintStatus {} {
Puts “Starting to execute the elevation filter “
}
vtkElevationFilter foo
foo AddObserver StartEvnt PrintStatus
```

回调功能对于 VTK 支持的所有开发语言都有效的。

为了创建你自己的应用程序，我们建议从一个 VTK 提供的例子开始，这个例子可以从源码的 VTK/Examples 的目录结构中找到，在子目录中将有可以用于在 PC 上执行的发布包，在源码的发布包中，例子是按照先主题然后语言的顺序来组织的，在 VTK/Examples 中你能找到不同主题的目录和在这些目录下你能找到为不同的开发语言如 Tcl 的子目录。

在 Windows 视窗系统中，你可以通过双击文件名，如 test1.tcl 运行 Tcl 脚本，如果什么也没发生，你的脚本中可能有错误，为了检测错误，你需要先运行 vtk.exe，vtk.exe 在你的开始菜单的 VTK 下面能够找到，一旦运行开始，一个包含输入的视窗控制台窗口就会显示出来，在输入部分键入一个 cd 的命令来改变到 test1.tcl 所在的目录。下面给出了 2 个例子：

```
% cd “c:/Program Files/Visualization Toolkit/examples”
% cd “c:/VTK/Examples/Tutorial/Tcl”
```

然后你就要使用下述命令指出你的例子脚本源程序：

```
%source Cone.tcl
```

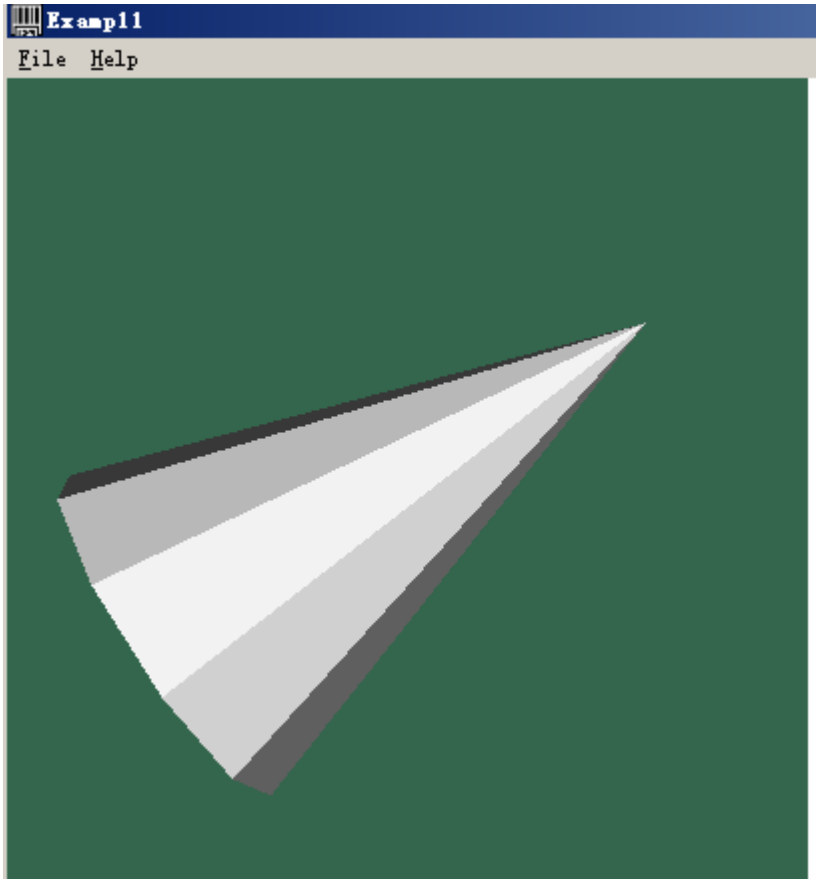
这样 Tcl 就会试着运行 test1.tcl，你就能看到那些原来不能被显示出来的错误或警告信息了。

2. Microsoft Visual C++

一旦你为 Cone 这个例子运行了 CMake 你就可以准备启动你的 Microsoft Visual C++软件来加载 cone.dsw 项目，你可以选择一种创建类型（例如 Release 或者 Debug）然后创建你的应用。

现在来看一个这个真正的视窗应用例子，这个例子要生成一个视窗的应用程序而非一个终端的应用程序，很多代码都是标准的视窗代码，任一个 Windows 的开发者都非常熟悉。

示例程序的运行结果如下图所示：



示例程序代码(Examp11)：

```
#include “vtkConeSource.h”
#include “vtkPolyDataMapper.h”
#include “vtkRenderWindow.h”
#include “vtkRenderWindowInteractor.h”
static HANDLE hinst;
```

```
long FAR PASCAL WndProc(HWND ,UINT,UINT,LONG);
//define the vtk part as a simple c++ class
Class myVTKApp
{
Public:
myVTKApp(HWND parent);
~myVTKApp();
Private:
vtkRenderWindow *renWin;
vtkRenderer *renderer;
vtkRenderWindowInteractor *iren;
vtkConeSource *cone;
vtkPolyDataMapper *coneMapper;
vtkActor *coneActor;
};
```

我们从包含必要的 VTK 包含头文件开始，我们不必包含标准的 windows 头文件，因为 VTK 的头文件中已经包含了，下一步我们有 2 个 windows 标准的原型，跟着一个叫作 myVTKApp 的类定义，当我们使用 C++来开发时，你应该试着使用面象对象来实现而非使用在很多 Tcl 例子中一样的脚本编程风格，这里我们将这个应用中使用到 VTK 的组件集中到 myVTKApp 类中来。

```
myVTKApp::myVTKApp(HWND hwnd)
{
    //Similar to Examples/Tutorial/Step1/Cxx/Cone.cx
    //We create the basic parts of a pipeline and conne
    this->renderer = vtkRenderer::New();
    this->renWin = vtkRenderWindow::New();
    this->renWin->AddRenderer(this->renderer);
    // setup the parent window
    this->renWin->SetParentId(hwnd);
    this->iren = vtkRenderWindowInteractor::New()
```

```
    this->iren->SetRenderWindow(this->renWin);
    this->cone = vtkConeSource::New();
    this->cone->SetHeight( 3.0 );
    this->cone->SetRadius( 1.0 );
    this->cone->SetResolution( 10 );
    this->coneMapper = vtkPolyDataMapper::New();
    this->coneMapper->SetInputConnection(this->cone->GetOutputPort());
    this->coneActor = vtkActor::New();
    this->coneActor->SetMapper(this->coneMapper);

    this->renderer->AddActor(this->coneActor);
    this->renderer->SetBackground(0.2,0.4,0.3);
    this->renWin->SetSize(400,400);
    // Finally we start the interactor so that event will be handled
    this->renWin->Render();
}
```

这是 myVTKApp 的构造器。你可以看到他生成了必要的 VTK 对象，设置了他们实例化的变量，然后将他们连接组成代了图像处理的流程。除了 vtdRenderWindow 外，大部分都是直接调用 VTK 的代码。这个构建器使用到你窗口 HWND 句柄来包含 VTK 图像处理的窗口，然后使用 vtkRenderWindow 的 SetParentId()方法，这样一来，它会将他的窗口创建成一个子窗口传递给构建器。

```
myVTKApp::~myVTKApp()
{
        renWin->Delete();
        renderer->Delete();
        iren->Delete();
        cone->Delete();
        coneMapper->Delete();
        coneActor->Delete();
}
```

这个析构函数只是简单的将在构造函数中生成的 VTK 对象释放内存。

```
int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                        LPSTR lpszCmdParam, int nCmdShow)
{
    static char szAppName[] = "Win32Cone";
    HWND                hwnd ;
    MSG                  msg ;
    WNDCLASS        wndclass ;
    if (!hPrevInstance)
        {
        wndclass.style         = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
        wndclass.lpfnWndProc   = WndProc ;
        wndclass.cbClsExtra    = 0 ;
        wndclass.cbWndExtra    = 0 ;
        wndclass.hInstance     = hInstance;
        wndclass.hIcon         = LoadIcon(NULL,IDI_APPLICATION);
        wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW);
        wndclass.lpszMenuName    = NULL;
        wndclass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
        wndclass.lpszClassName = szAppName;
        RegisterClass (&wndclass);
        }

    hinst = hInstance;
    hwnd = CreateWindow ( szAppName,
                          "Draw Window",
                           WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT,
                           CW_USEDEFAULT,
                           400,
```

```
                                480,
                                NULL,
                                NULL,
                                hInstance,
                                NULL);
    ShowWindow (hwnd, nCmdShow);
    UpdateWindow (hwnd);
while (GetMessage (&msg, NULL, 0, 0))
        {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
        }
    return msg.wParam;
}
```

这里的 WinMain 代码全部是一些标准的 windows 代码,并没有相关的 VTK 的东西在里边。你可以看到应用处理了事件的循环，事件被如下描述的 WndProc 处理了。

```
long FAR PASCAL WndProc (HWND hwnd, UINT message,
                                                    UINT wParam, LONG lParam)
{
    static HWND ewin;
    static myVTKApp *theVTKApp;
    switch (message)
        {
        case WM_CREATE:
            {
            ewin = CreateWindow("button","Exit",
                                WS_CHILD | WS_VISIBLE | SS_CENTER,
                                0,400,400,60,
                                hwnd,(HMENU)2,
    (HINSTANCE)GetWindowLong(hwnd,GWL_HINSTANCE),
```

```
                                NULL);
            theVTKApp = new myVTKApp(hwnd);
            return 0;
            }

        case WM_COMMAND
switch (wParam)
                {
                case 2:
                    PostQuitMessage (0);
                    if (theVTKApp)
                        {
                        delete theVTKApp;
                        theVTKApp = NULL;
                        }
                    break;
                }
return 0;

        case WM_DESTROY:
            PostQuitMessage (0);
            if (theVTKApp)
                {
                delete theVTKApp;
                theVTKApp = NULL;
                }
            return 0;
        }
    return DefWindowProc (hwnd, message, wParam, lParam);
}
```

这个例子中，WndProc 是一个非常简单的事件处理器，在这个函数的顶端我们定义了一个静态 myVTKApp 类的实例变量，当处理 WM_CREATE 方法时我们生成一个退出按钮然后构建它，并在 myVTKApp 中实例化并将指针传向当前窗口，vtkRenderWindowInteractor 将处理针对 vtkRenderWindow 的所有事件，你可能需要增加处理调整大小的事件，以便 VTK 的绘制窗口能与整体的用户窗口的大小一并变化，如果你不设置 vtkRenderWindow 的 ParentId，它就会顶层一个无关的窗口中显示出来。

3. **C++中的用户方法**

你可以使用 C++通过创建 vtkCommand 的子类并重载其中的 Execute()方法来定义用户方法（使用 observer/command 设计模式）。

查看下面这个来自于 Examples/Tutorial/Step2/cxx 的例子：

```
//派生于 vtkCommand 类
myCallback:public vtkCommand {
static myCallback *New() { return new myCallback;}
virtual void Execute(vtkObject *caller,unsigned long ,void *callData)
{
    cerr <<" Starting to Render \n" ; }
};
```

Execute（）方法总是传递调用的对象，如果你确实需要使用调用者时，你要执行 SafeDownCast()来获得实际的类型，下面是一个例子：

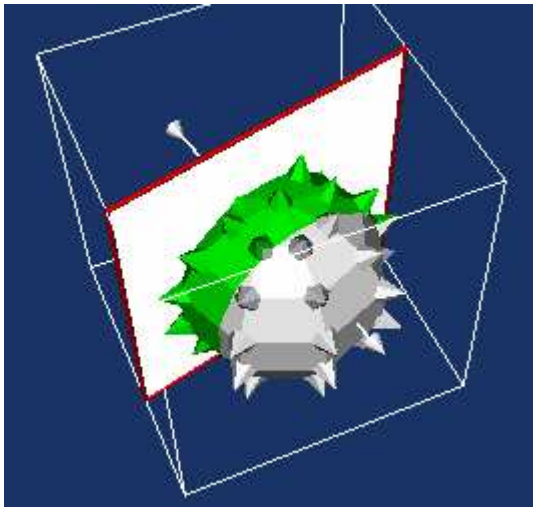
```
virtual void Execute(vtkObject *caller,unsigned long ,void *callData) {
  vtkRenderer *ren = vtkRenderer::SafeDownCast(caller);
  if (ren) { ren->SetBackground(0.2,0.3,0.4);}
}
```

一旦你创建了 vtkCommand 的子类，你就可以增加一个观察者在并设置相应的事件，当事件发生时调用你的命令，可以按如下例子这样来实现：

```
//Here is where we setup the observer,
//We do a new and ren1 will eventually free the observer
myCallback *mo1 = myCallback::New();
ren1->AddObserver(vtkCommand::StartEvent,mo1);
```

上面的代码创建了一个 myCallback 实例，然后在 ren1 上为 StartEvent 事件增加了

一个 observer，只要 ren1 开始运行 Execute()方法 myCallback 就会被调用，当 ren1 被删除后，这个回调也会被删除，示例程序的运行结果如下图:



示例程序代码（Examp12）:

```
class vtkTIPWCallback : public vtkCommand
{
public:
    static vtkTIPWCallback *New()
    { return new vtkTIPWCallback; }
    virtual void Execute(vtkObject *caller, unsigned long, void*)
    {
        vtkImplicitPlaneWidget *planeWidget =
            reinterpret_cast<vtkImplicitPlaneWidget*>(caller);
        //得到隐函数
        planeWidget->GetPlane(this->Plane);
        this->Actor->VisibilityOn();
    }
    vtkTIPWCallback():Plane(0),Actor(0) {}
    vtkPlane *Plane;
    vtkActor *Actor;

};
```

```
vtkSphereSource *sphere = vtkSphereSource::New();
vtkConeSource *cone = vtkConeSource::New();
//符号化过滤器
vtkGlyph3D *glyph = vtkGlyph3D::New();
//设定需要被符号化的数据，这里为球体
glyph->SetInputConnection(sphere->GetOutputPort());
//设定符号的类型，这里为一个锥体
glyph->SetSource(cone->GetOutput());
//设定使用法向量缩放符号
glyph->SetVectorModeToUseNormal();
//设定使用标量值或矢量大小缩放符号
glyph->SetScaleModeToScaleByVector();
//设定符号的缩放系数，按符号原始大小缩放
glyph->SetScaleFactor(0.2);
//球体和符号组合在一起，形成一个新的多边形数据集
vtkAppendPolyData *apd = vtkAppendPolyData::New();
apd->AddInput(glyph->GetOutput());
apd->AddInput(sphere->GetOutput());

vtkPolyDataMapper *maceMapper = vtkPolyDataMapper::New();
maceMapper->SetInputConnection(apd->GetOutputPort());

vtkLODActor *maceActor = vtkLODActor::New();
maceActor->SetMapper(maceMapper);
maceActor->VisibilityOn();
//定义一个平面方程隐函数
vtkPlane *plane = vtkPlane::New();
vtkClipPolyData *clipper = vtkClipPolyData::New();
//设定被剪切的数据集
clipper->SetInputConnection(apd->GetOutputPort());
```

```
//设定剪切函数
clipper->SetClipFunction(plane);
//设定隐函数剪切的值
clipper->SetValue(0.3);
//被剪切掉部分的顶点值如果小于或等于 0.3，显示这些顶点
clipper->InsideOutOn();

vtkPolyDataMapper *selectMapper = vtkPolyDataMapper::New();
selectMapper->SetInputConnection(clipper->GetOutputPort());

vtkLODActor *selectActor = vtkLODActor::New();
selectActor->SetMapper(selectMapper);
selectActor->GetProperty()->SetColor(0,1,0);
selectActor->VisibilityOff();
selectActor->SetScale(1.01, 1.01, 1.01);

// Create the RenderWindow, Renderer and both Actors
//
vtkRenderer *ren1 = vtkRenderer::New();
vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer(ren1);

vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
iren->SetRenderWindow(renWin);
//定义一个回调类
vtkTIPWCallback *myCallback = vtkTIPWCallback::New();
myCallback->Plane = plane;
myCallback->Actor = selectActor;
//设定剪切平面
vtkImplicitPlaneWidget *planeWidget = vtkImplicitPlaneWidget::New();
```

```
//设定交互器
planeWidget->SetInteractor(iren);
//设定平面的缩放系数
planeWidget->SetPlaceFactor(1.25);
planeWidget->SetInput(glyph->GetOutput());
//设定平面的初始位置
planeWidget->PlaceWidget();
planeWidget->AddObserver(vtkCommand::InteractionEvent,myCallback);
//显示平面
planeWidget->On();
```

2 VTK 使用基础

本章主要通过示例程序绍 VTK 所具有的一些的功能及一些通用的对象和方法，同时也对一些重要的概念和有用的应用程序进行着重说明。

2.1 创建一个简单的示例

VTK 建立应用程序的基本过程如下：

1) 读取/生成数据

2) 过滤数据

3) 绘制图形

4) 交互操作

在这一部分，我们将要说明如何读取/生成数据，VTK 提供了两种获取数据的方法，一种方法是读取存在的数据文件，另一种方法是通过算法或数学表达式生成数据，在可视化流水线中起始节点对象被称为源对象，源对象又分为程序源对象、读源对象，通过数学方法生成数据的对象被称为程序源对象，从数据文件中读取数据的对象被称为读源对象。

◆ 程序源对象

我们通过 vtkCylinderSource 对象创建一个程序源对象（圆柱体），并绘制它，这个源对象通过创建多边形来表现一个圆柱体，圆柱体的输出作为映射器的输入，

示例代码如下：

```
vtkCylinderSource *cylinder = vtkCylinderSource::New();//创建圆柱体
cylinder->SetResolution(8);//生成圆柱体多边形的数目
vtkPolyDataMapper *myMapper = vtkPolyDataMapper::New();//创建影射器对象
myMapper->SetInput(cylinder->GetOutput());//圆柱体的输出作为影射器的输入
vtkLODActor *myActor=vtkLODActor::New();//创建演员对象
myActor->SetMapper(myMapper);//将可视化流水线数据转入图形系统
myActor->GetProperty()->SetDiffuseColor(1.0,1.0,1.0);//设定颜色
vtkRenderer *ren1 = vtkRenderer::New();//创建绘制者
ren1->AddActor(myActor);
ren1->SetBackground(0.1, 0.2, 0.4);
vtkRenderWindow *renWin = vtkRenderWindow::New();//绘制窗口
renWin->AddRenderer(ren1);
vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
//iren->SetDesiredUpdateRate(5.0);
iren->SetRenderWindow(renWin);
renWin->SetSize(200, 200);
renWin->Render();//绘制
iren->Start();
//删除对象
cylinder ->Delete();
myMapper->Delete();
myActor->Delete();
ren1->Delete();
renWin->Delete();
iren->Delete();
```

在这个示例中，映射器对象是流水线和图形对象的接口，流水线主要用于对数据的处理，图形对象主要用于绘制数据。

◆ 读源对象

这个示例是上面的示例相似，只不过数据的获取是从数据文件中读取，而不是由程序生成，代码如下：

```
vtkSTLReader *myReader= vtkSTLReader::New();//创建读源对象
myReader->SetFileName("abc.stl");//读取数据文件 abc.stl
```

使用该对象时需要注意的是：当数据文件发生变化后，VTK 读源对象不能够感觉数据文件已经变化，所以必须调用 Modified 方法，保证数据文件变化时，让流水线重新执行。

2.2 使用 VTK 交互功能

一旦可视化了一个数据后，用户一般都需要和它交互，VTK 提供了一些交互的方法，第一种方法是使用内建的交互类 vtkRenderWindowInteractor,另外一种方法是根据事件邦定创建交互方法。

- 使用vtkRenderWindowInteractor类交互

最简单的和数据交互的方法是创建该类的对象实例,该类提供了一些预定义的事件和行为，该类允许我们控制相机和 Actors,并且提供了操纵杆模式和跟踪球模式两种交互类型。该类对绘制窗口的如下事件作出响应：

压下键盘 j/t 键，在操纵杆和跟踪球交互类型间切换。

压下键盘 c/a 键，在相机和 Actors 模式间切换。

压下鼠标左键，在相机模式下，围绕相机焦点旋转，在 Actors 模式下，围绕 Actors 的原点旋转。

压下鼠标中键，在相机模式下，扫视相机，在 Actors 模式下，平移 Actors，对于 2 键鼠标，使用 shift+鼠标左键。

压下鼠标右键，在相机模式下，推拉相机，在 Actors 模式下，缩放 Actors。

压下键盘 3 键，进入立体模式。

压下键盘 e 键，退出程序。

压下键盘 p 键，进行拾取。

压下键盘 r 键，沿着当前的视方向重新设置相机。

压下键盘 u 键，调用用户设定的方法。

压下键盘 s/w 键，Actors 的显示方式在线框/表面模式间切换。

缺省的交互类型是跟踪球模式,在这种模式下,只要鼠标键按下,就可以对相机、Actors、Renders 进行持续的操作。

vtkRenderWindowInteractor 类还有其他一些特性，如调用 LightFollowCameraOn()方

法，可以让相机和光源同步，使用 AddObserver(UserEvent)方法，可以添加用户定义的方法(u 键按下时),也可以设定一些和拾取相关的方法,如使用 AddObserver(StartPickEvent)可以定义一个当拾取事件发生时，调用用户定义的方法，SetPicker()方法可以设定 vtkAbstractPicker 类子类的对象实例，SetDesiredUpdae（）方法可以设定细节层次的帧频。

如何使用该类，在示例代码中的代码如下：

```
vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
//iren->SetDesiredUpdateRate(5.0);
iren->SetRenderWindow(renWin);
```

- 交互类型

VTK 提供了两种交互方法，第一种是使用 vtkInteractorStyle 类的子类，第二种是使用系统提供的交互方法（如 windows 系统的事件）或者用户自己定义交互类型，如果使用第二种方法，用户必须直接管理事件循环。

1) 第一种交互方法

vtkRenderWindowInteractor 类支持不同的交互类型，它的工作方式是：vtkRenderWindowInteractor 将它接收的事件转送给它的交互类型，由些交互类型回应这些事件，为了设定交互类型，使用 vtkRenderWindowInteractor:: SetInteractorStyle（）方法(在 windows 系统下,提供了 vtkWin32RenderWindowInteractor 交互类,有关与 windows 系统的交互，参阅第七章)。

交互示例代码如下：

```
//定义飞行类型
vtkInteractorStyleFlight *myFlight= vtkInteractorStyleFlight::New();
vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
iren->SetInteractorStyle(myFlight);//设置交互类型
```

2) 第二种交互方法

这种交互方法是直接对事件循环进行管理，用户可以创建自己的事件邦定，这些邦定可以被 VTK 所支持的语言（C++、Tcl、Java）管理。

2.3 过滤器

过滤器（Filter） 是一种数据处理机制,有一个或多个输入,有一个输出，其目的是对图形图像数据进行处理,以得到我们期望的数据。

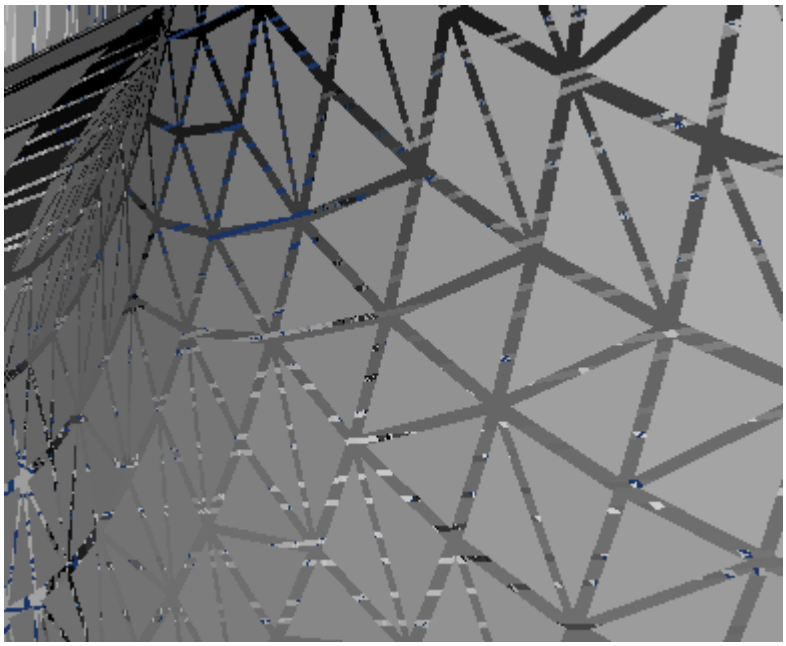
如专门对图像数据进行处理的类有:

```
#include "vtkImageSobel2D.h"
#include "vtkImageLaplacian.h"
#include "vtkImageHybridMedian2D.h"
#include "vtkImageFFT.h"
#include "vtkImageRFFT.h"
#include "vtkImageButterworthHighPass.h"
#include "vtkImageButterworthLowPass.h"
#include "vtkImageIdealLowPass.h"
#include "vtkImageIdealHighPass.h"
#include "vtkImageAnisotropicDiffusion2D.h"
#include "vtkImageAccumulate.h"
#include "vtkImageConvolve.h"
#include "vtkImageExtractComponents.h"
#include "vtkImageGaussianSmooth.h"
#include "vtkImageShiftScale.h"
#include "vtkImageShrink3D.h"
#include "vtkImageFlip.h"
#include "vtkImagePermute.h"
#include "vtkImageResample.h"
#include "vtkImageConstantPad.h"
#include "vtkImageMirrorPad.h"
#include "vtkImageMagnify.h"
```

以上的类大都可以用作对二维图像的处理，也可以用做处理三维的体数据。vtkShrinkPolyData 的作用是将图形中的所有三角面片缩小一定的比例,但整个图形的大小没有变,这就使得图中出现龟裂。

在前面示例的代码中，可视化流水线由源对象和映射器对象两部分组成，这部分我们将说明如何在流水线中添加过滤器对象。

过滤器对象之间使用 SetInput()和 GetOutput()方法相互建立连接，下面的示例程序用过滤器对多边形数据进行收缩处理，运行结果如下：



示例代码如下(Examp21)：

```
vtkSTLReader *myReader= vtkSTLReader::New();
myReader->SetFileName("42400-IDGH.stl");
//创建过滤器
vtkShrinkPolyData *myshirkData=vtkShrinkPolyData::New();
myshirkData->SetShrinkFactor(0.9);
//建立连接
myshirkData->SetInput(myReader->GetOutput());
vtkPolyDataMapper *myMapper = vtkPolyDataMapper::New();
myMapper->SetInput(myshirkData->GetOutput());
```

2.4 控制相机

绘制三维场景时，相机和灯光是不可缺少的，在 VTK 中建立场景时，绘制器已经创建了缺省的相机和灯光，所以相机和灯光不需要自己创建。

2.4.1 创建相机

Camera（相机）在 VTK 中可以理解成视点，即观察图形的位置，下面的代码示例了如何将创建的相机和绘制器关联。

使用 vktCamera 类创建相机，并设定相机的各种属性，示例代码如下：

```
//创建相机
vtkCamera *myCamer=vtkCamera::New();
myCamer->SetClippingRange(0.01,1000);//设定远、近裁减平面
myCamer->SetFocalPoint(0.0,0.0,0.0);//设定焦点位置
myCamer->SetPosition(0.0,1.0,0.0);//设定相机位置
myCamer->SetViewUp(0.0,0.0,1.0);//设定相机向上方向
myCamer->ComputeViewPlaneNormal();//设定视平面法线
myCamer->SetEyeAngle(30);//设定视角
vtkRenderer *ren1 = vtkRenderer::New();
ren1->SetActiveCamera(myCamer);//将相机设为活动相机
```

我们也可以通过 ren1->GetActiveCamera()方法获得当前的相机。

让我们回顾一下刚才介绍的相机对象类的一些方法，SetClippingPlane()方法有俩两个参数，分别表示沿着视平面法线近、远裁减平面的距离，所有位于近、远裁减平面的之外的物体在绘制时被裁减掉，所以我们必须确保被绘制的物体位于近、远裁减平面内，相机在世界坐标系内的放置位置和焦点位置控制着相机的位置和方向，ComputeViewPlaneNormal()，方法根据相机的位置和焦点重新设定视平面的法矢量，Zoom()方法通过改变视角放大对象，也可以使用 Dolly()方法，沿着视平面法矢量移动相机，放大或缩小物体。

2.4.2 简单的操作方法

以上描述的方法在控制相机方面还有许多的不便之处，我们可以使用 Azmuch()、Elevation()方法围绕焦点旋转相机。

示例代码：

```
myCamer->Azmuch(120);
myCamer->Elevation(60);
```

这些方法以焦点（视点）为中心，在球面坐标上移动相机，Azmuch()方法沿着经线方向

移动相机，Elevation()方法沿着纬线方向移动相机，注意当相机移动到南极或北极时，观察的向上方向和观察方向平行，会产生异常，为了避免这种情况，我们可以使用，OrthogonalizeViewUP()方法强迫观察方向和观察的向上方向正交。

2.4.3 控制观察方向

我们可以通过调用 SetFocalPoint()、SetPosition()、ComputeViewPlaneNormal()方法生成相机特定的观察方向，并调用ResetCamera()方法使观察方向生效。

示例代码：

```
myCamer->SetFocalPoint(0,0,0);//观察点在原点
myCamer->SetPosition(1,1,1);//相机的位置在(1,1,1)处
myCamer->ComputeViewPlaneNormal();//设定视平面的法矢量
myCamer->SetViewUp(1,0,0);//设置观察向上方向
vtkRenderer *ren1 = vtkRenderer::New();
ren1->SetActiveCamera(myCamer);//将相机设为活动相机
ren1->ReSetCamera();
```

2.4.4 透视及正交投影

VTK 提供了透视投影和正交投影，默认的情况下，使用透视投影，透视投影保持了人眼近大远小的实际观察效果，具有空间感，但有可能带来失真现象，对于正交投影而言，由于视线是平行的，物体的在各个方向和距离都保持不变，不符合人眼自然的观察习惯，但没有失真现象，使用 vtkCamera:: ParallelProjectionOn()方法设定相机使用正交投影。

我们可以使用 vtkCamera 类的相关属性保存相机的状态（如裁减平面、视角等）。

2.5 控制光源

光源比相机更加容易控制，控制光源经常使用的方法是：SetPosition（）、SetFocalPoint()、SetColor(),光源的位置点和照射点（焦点）控制光源的方向（由光源的位置点指向照射点形成的方向矢量是光源的方向），SwitchOn()和 SwitchOff()方法控制光源的开和关，SetIntensity()方法控制光源的亮度,光源分为点光源、方向光、聚光灯，默认的光源为方向光。

光源和绘制器关联后才可生效，关联的代码如下：

```
vtkLight *myLight=vtkLight::New();
myLight->SetPosition(0,0,10);//设定光源位置点
myLight->SetFocalPoint(0,0,0);//设定光源照射点
myLight->SetDiffuseColor(1.0,1.0,0.0);//漫反射光颜色
myLight->SetAmbientColor(1.0,1.0,1.0);//环境光颜色
vtkRenderer *ren1 = vtkRenderer::New();
ren1->AddLight(myLight);
```

定位光源指点光源与聚光灯，使用 SetPositionalOn()方法设置点光源，这个方法和 SetConeAngle()方法联合起来可以设置聚光灯，180 度的圆锥角对聚光灯无效。

2.6 控制场景中的物体（3D Props）

被绘制到场景中的物体被称为“Props”，在 VTK 中，vtkProp3D 类是所有物体的抽象基类，该类有一个变换矩阵，提供缩放、平移、旋转等变换。

2.6.1 指定物体的空间位置

我们既可以保持物体位置固定，让相机围绕物体旋转，也可以保持相机位置固定，移动物体，让物体围绕相机旋转，VTK 提供了如下的方法移动物体的位置。

SetPosition(x,y,z)：指定物体在世界坐标系中的位置。

AddPosition（deltax,deltay, deltaz）按照指定的平移量在 x 轴、y 轴、z 轴平移物体。

RotateX(angle)、RotateY(angle)、RotateZ(angle)：绕 x、y、z 轴旋转物体。

SetOrientation(x,y,z)：设定物体的方位，物体绕 z、x、y 轴的顺序旋转。

RotateWXYZ(theat,x,y,z)：物体绕用户定义的轴(x,y,z)旋转 theat 角度。

其它相关方法参考 vtkProp3D 类。

在对物体实施缩放、平移、旋转变换时，要注意变换的顺序，不同的变换顺序会产生不同的效果，在 VTK 中，对物体实施变换的过程如下，假设物体在初始位置：

将物体移动到坐标原点、缩放变换、绕 Y 轴旋转、绕 X 轴旋转、绕 Z 轴旋转、将物体移回到初始位置。

在平移变换、旋转变换和缩放变换中，最容易混淆的变换是旋转变换，旋转的顺序不同，

产生的结构不同，假设一个物体先绕 X 轴旋转，然后再绕 Y 轴旋转和先绕 Y 轴旋转再绕 X 周旋转，产生的变换结果是不同的。

2.6.2 Actros

Actor（角色）用来在一场景中表现一个可视化实体，主要用作对图形的绘制，如一些简单的球形、锥体等，Actor 是一个应用比较多的 vtkProp3D 类的子类，Actor 提供了一组相关的绘制属性，如物体的表面属性（反射光、漫反射光、颜色等）、物体的显示方式（实体显示、线框显示）、物体的纹理映射、物体的几何体定义等。

- 几何体定义

Actor 的几何体通过 SetMapper()方法指定。

示例代码如下：

```
vtkCylinderSource *cylinder = vtkCylinderSource::New();//创建圆锥体
cylinder->SetResolution(8);//生成圆锥体多边形的数目
vtkPolyDataMapper *myMapper = vtkPolyDataMapper::New();//创建映射器对象
myMapper->SetInput(cylinder->GetOutput());//圆锥体的输出作为映射器的输入
vtkLODActor *myActor=vtkLODActor::New();//创建演员对象
myActor->SetMapper(myMapper);//将可视化流水线数据转入图形系统
```

在这个示例中，使用 vtkPolyDataMapper 类定义映射器，该映射器使用基本的图元（如：点、线、多边形、三角带）绘制几何体，映射器是可视化流水线和图形系统的接口，其将流水线数据映射到图形系统。

- Actor 属性

被绘制的物体包含了一些重要的属性，如物体的显示方式（点、线框、表面显示）、物体材质颜色，透明度等，这些属性均可通过 Actor 设定，Actor 包含了一个 vtkProperty 类的实例，如:：

```
Actor->GetProperty()->SetColor(1,1,1);     //(1,1,1)白色
```

表示将绘制对象的着色设为白色.。

- Actor 颜色

物体的材质、物体的环境光、漫反射光、镜面反射光等都可以通过 Actor 设定，需要注意的是，当被绘制的几何体带有属性数据时，通过上述方法设置的颜色将无效，VTK 将要

使用属性数据和默认的颜色查找表为物体绘制颜色，为了使用物体的材质、物体的环境光、漫反射光、镜面反射光为物体绘制颜色，可以使用 ScalarVisibilityOff（）方法忽略属性数据。示例代码如下：

```
vtkPolyDataMapper *pPolyMap= vtkPolyDataMapper::New();
pPoly->SetInput(pPlane->GetOutput());
//忽略属性数据
pPoly-> ScalarVisibilityOff();
vtkActor *myActor=vtkActor::New();//创建演员对象
myActor->SetMapper(pPoly);//将可视化流水线数据转入图形系统
myActor->GetProperty()->SetColor(1,1,1);     //设置绘制物体的颜色
```

- 物体的透明度

物体的透明度可以显示物体内部的情况，通过 vtkProperty::SetOpacity()方法设定物体的透明度。

注意：有多个物体同时绘制时，要保证被透明的物体最后绘制，可以使用过滤器（vtkDepthSortPolyData）对被绘制的物体沿着观察方向按深度排序。

- 其它属性

Actor 也提供了其它的物体属性设定方法，如物体是否在场景中可见（使用 VisibilityOff/On()方法设定）、物体是否可被拾取、得到物体的边界框（GetBoundes()方法）等。

2.6.3 物体的层次细节

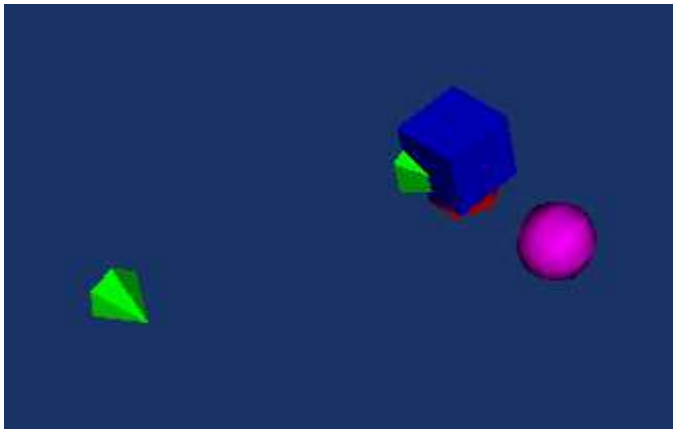
VTK 使用物体层次细节提高物体的绘制效率，最简单的方法是使用 vtkLODActor 代替 vtkActor，控制物体的细节显示，vtkLODActor 可以用 AddLODMapper（）方法添加不同细节层次的映射物体，物体在绘制期间，为了控制物体的层次细节，我们可以在绘制窗口中设定期望的帧率。

2.6.4 组装

在现实生活中,通常的物体都是复杂图形,我们可以用简单图形来组成复杂图形,在 VTK 中将一些描绘简单图形的 actor 组装起来,合并成一个 actor,在绘制过程中,只要添加到

一个 assembily 类中就可以显示多个图形。

场景中的物体通常由多个部分按一定的层次组装而成，如一个机器人的手臂由前臂、上臂、手腕和手组装而成，这些部分通过节点连接在一起，当上臂围绕肩部旋转的时候，我们期望其余的部分也随之转动，我们使用组装实现这种行为，下面的示例程序，显示了组装的使用，显示的结果如下：

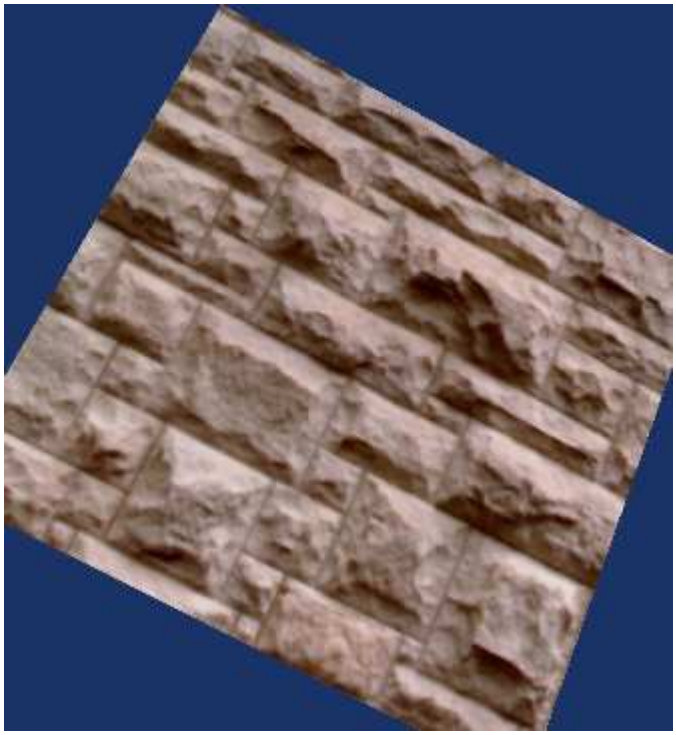


程序示例代码如下(Examp22)：

```
//将物体组装在一起
vtkAssembly *assembly = vtkAssembly::New();
assembly->AddPart(sphereActor);
assembly->AddPart(cubeActor);
assembly->AddPart(coneActor);
assembly->AddPart(cylinderActor);
assembly->SetOrigin(5 , 10 , 15);
assembly->SetPosition(0 , 0 , 0);
```

2.6.5 使用纹理

在 VTK 中可以将一个二维图像文件贴在一个三维图形上做为纹理，在 VTK 中使用纹理，需要一张纹理图片和纹理坐标（控制纹理图片粘贴到物体表面的位置），下面的示例程序说明如何使用纹理图片，运行的结果如下图：



要对物体进行纹理影射，必须要有纹理图（图片）和纹理坐标数据（控制图片在物体表面的位置），下面的示例显示了怎么样使用纹理。

示例程序代码（Examp23）：

```
vtkBYUReader *model_reader=vtkBYUReader::New();
model_reader->SetGeometryFileName("teapot.g");//载入三维图形数据
vtkPolyDataNormals *model_normals=vtkPolyDataNormals::New();
model_normals->SetInput(model_reader->GetOutput());//法线设置
vtkTextureMapToCylinder *tmapper=vtkTextureMapToCylinder::New();
tmapper->SetInput(model_normals->GetOutput());
tmapper->PreventSeamOn();
vtkTransformTextureCoords*
transform_texture=vtkTransformTextureCoords::New();
transform_texture->SetInput(tmapper->GetOutput());//纹理处理
vtkDataSetMapper *mapper=vtkDataSetMapper::New();
mapper->SetInput(transform_texture->GetOutput());
vtkJPEGReader *texture_reader=vtkJPEGReader::New();//载入图像文件数据
texture_reader->SetFileName("beach.jpg");
vtkTexture *texture=vtkTexture::New();
```

```
texture->SetInputConnection(texture_reader->GetOutputPort());
texture->InterpolateOn();
vtkActor *actor=vtkActor::New();
actor->SetMapper(mapper);//图形数据显示
actor->SetTexture(texture);//将图像映射到图形数据上
```

2.6.6 拾取

拾取常用来选择场景中的物体，获取物体的信息，通常调用vtkAbstractPicker::Pick()方法拾取物体，根据使用的拾取类不同，从拾取物体中返回的信息也不相同，如返回的信息可能是被拾取物体的全局坐标、单元 ID 或者点的 ID，Pick()方法的语法格式如下：

Pick（SelectX,SelectY,SelectZ,Renderer）；

我们注意到在这个方法中需要使用 Renderer（绘制器），凡是和 Renderer 相关联的角色都是被拾取对象，该方法不能直接调用，而是使用 vtkRenderWindowInteractor 类管理拾取对象，用户将拾取对象的实例赋给 vtkRenderWindowInteractor 类控制拾取的过程。

为了满足不同的拾取要求，VTK 提供了几种不同类型的拾取类，vtkAbstractPicker 是拾取类的基类，它定义了了一些基本的拾取方法，它的派生类包括：

- vtkWorldPointPicker

用于快速的对角色进行拾取，并获取拾取点的世界坐标值(x,y,z)。

- vtkAbstractPropPicker

用于拾取场景中的对象，并且返回拾取对象的数据类型。

- vtkPropPicker

是 vtkAbstractPropPicker 类的派生类，用于硬件支持的拾取操作，在有些硬件图形系统上，该类不被支持，在这种情况下，使用软件版本的 vtkPicker 类。

- vtkPicker

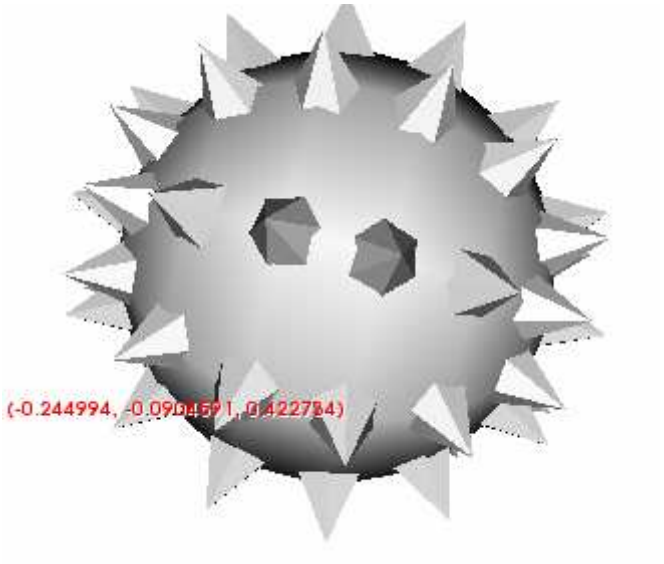
是 vtkAbstractPropPicker 类的派生类，用软件的方式实现边界拾取，由于该类执行边界拾取，所以不能实现单一拾取功能，该类还有两个派生类，用于获取更多的拾取信息，如单元 ID、点 ID 和拾取点坐标等，vtkPointPicker 用于获取点的信息（点 ID、坐标），vtkCellPicker 用于拾取单元并且返回单元的信息（单元 ID、单元参数坐标）。

- vtkAssemblyPath

当场景对象由 vtkAssembly 对象组装在一起时，vtkAssemblyPath 包含了 vtkAssembly 中每个对象的结点和变换矩阵，通过 vtkAssemblyPath 类可以访问 vtkAssembly 中的每个对象。

使用拾取可以定义一些交互操作，StartPickEvent 事件在执行拾取之前被调用，EndPickEvent 事件在拾取执行完毕之后被调用，每次对象被拾取时 PickEvent 事件(vtkActor 类也有该事件)被调用。

在通常情况下，vtkRenderWindowInteractor 类自动对拾取操作进行管理，如当我们按下 p 键时，该类会调用 vtkPropPicker 类对对象进行拾取操作，下面例举拾取的应用，该示例程序的运行结果如下图所示：



示例程序代码(Examp24)：

```
//创建拾取器
vtkCellPicker *picker = vtkCellPicker::New();
 picker->SetTolerance(0.025);
 picker->SetPickFromList(0);
 //当拾取结束时，调用回到方法
 picker->AddObserver(vtkCommand::EndPickEvent,myCall);
 vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
 iren->SetRenderWindow(renWin);
 iren->SetLightFollowCamera(1);
 //设定我们定义的拾取器
```

```
iren->SetPicker(picker);
ren1->AddActor( textActor );
ren1->AddActor( sphereActor );
ren1->AddActor( spikeActor );
```

2.7 VTK 中的坐标系统

VTK 支持多种不同类型的坐标系统,类 vtkCoordinate 可以在不同的坐标系统进行变换,VTK 支持的坐标系统如下:

```
DISPLAY                    x-y pixel values in window
NORMALIZED DISPLAY         x-y (0,1) normalized values
VIEWPORT                   x-y pixel values in viewport
NORMALIZED VIEWPORT        x-y (0,1) normalized value in viewport
VIEW              x-y-z (-1,1) values in camera coordinates. (z is depth)
WORLD             x-y-z global coordinate values
USERDEFINED       x-y-z in User defined space
```

坐标系统变换示例代码如下:

```
vtkCoordinate *normCoords= vtkCoordinate::New();
normCoords->SetCoordinateSystemToNormalizedViewport();
vtkPolyDataMapper2D *mapper= vtkPolyDataMapper2D::New();
mapper->SetInput(anthor);
mappe->SetTransformCoordinate(normCoords);
```

设置坐标系统代码如下:

```
void vtkCoordinate::SetCoordinateSystemToDisplay();
void vtkCoordinate::SetCoordinateSystemToNormalizedDisplay();
void vtkCoordinate::SetCoordinateSystemToViewport();
void vtkCoordinate::SetCoordinateSystemToNormalizedViewport();
void vtkCoordinate::SetCoordinateSystemToView();
void vtkCoordinate::SetCoordinateSystemToWorld()
```

2.8 vtkActor2D

vtkActor2D 和 vtkActor 很相似,只不过它是对二维图形的操作.

示例代码如下:

```
vtkPolyData *selectRect= vtkPolyData::New();
selectRect->SetPoints (pts);
selectRect ->SetLines (rect);
vtkPolyDataMapper2D *rectMapper= vtkPolyDataMapper2D::New();
rectMapper ->SetInput (selectRect);
vtkActor2D *rectActor= vtkActor2D::New();
rectActor ->SetMapper (rectMapper);
```

2.9 注释

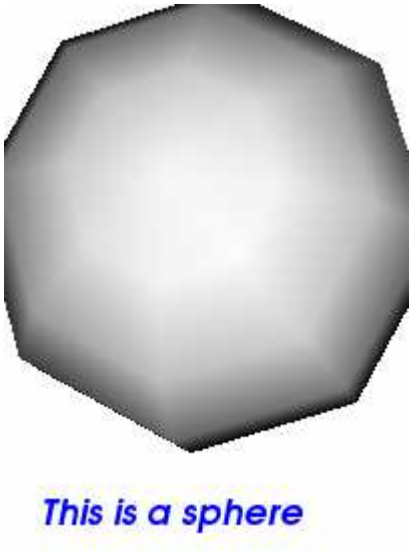
在 VTK 中有两种方法为所生成的图形添加注释:

(1)2D 注释，将文字放在二维的图形窗口上，文字不会因为用户的操作而变化。

(2)3D 注释，将文字添加在所生成的三维图形上，或者将文字直接以三维图形的方式显示。

2.9.1 2D Annotation

平面注释使用 vtkActor2D、vtkMapper2D 类将文字显示在一个平面上，下面的示例程序说明如何使用平面注释，程序运行结果如下：



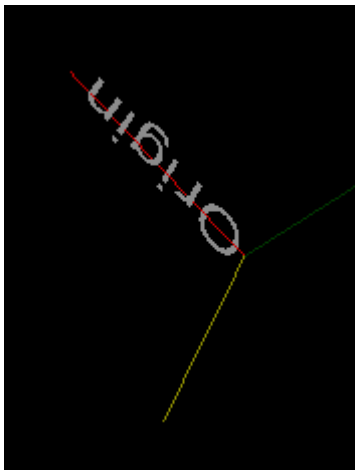
程序代码如下(Examp25):

```
//建立文本属性对象
vtkTextProperty *pTexProper=vtkTextProperty::New();
pTexProper->SetColor(0 , 0 , 1);
pTexProper->SetFontSize(18);
pTexProper->SetFontSize(18);
pTexProper->SetFontFamily(0);
pTexProper->SetJustification(1);
pTexProper->SetBold(1);
pTexProper->SetItalic(1);
pTexProper->SetShadow(1);
//建立文本映射器对象
vtkTextMapper *pTexMap=vtkTextMapper::New();
pTexMap->SetInput("This is a sphere");
pTexMap->SetTextProperty(pTexProper);
vtkTextActor *textActor = vtkTextActor::New();
textActor->SetMapper(pTexMap);
textActor->SetHeight(0.1);
//设置字体显示的位置
 textActor->SetPosition(90 , 50);
```

在 vtkTextProperty 类中,我们可以对文字的字体、大小、内容等属性做出更改,在 vtkScaledTextActor 中,我们可以控制注释的位置,以及颜色。

2.9.2 3D Annotation and vtkFollower

三维注释是用 vtkVectorText 来创建注释内容文字，vtkFollower 是绘制三维注释的一种方式，下面的示例程序说明了三维注释的使用方式，程序运行结果如下：



程序代码(Examp26):

```
//建立 3D 文本，为多边形数据
vtkVectorText *atext = vtkVectorText::New();
atext->SetText("Origin");
//映射文本，注意映射器的类型
vtkPolyDataMapper *textMapper = vtkPolyDataMapper::New();
textMapper->SetInput((vtkPolyData *) atext->GetOutput());
vtkFollower *textActor = vtkFollower::New();
textActor->SetMapper(textMapper);
textActor->SetPosition(0 , -0.1 , 0);
textActor->SetScale(0.2 , 0.2 , 0.2);
ren1->AddActor( textActor );
```

2.10 特定绘图

VTK 提供了了一些特定的绘图类，用于绘制颜色图例、X-Y 平面图表和在三维空间中绘

制坐标轴。

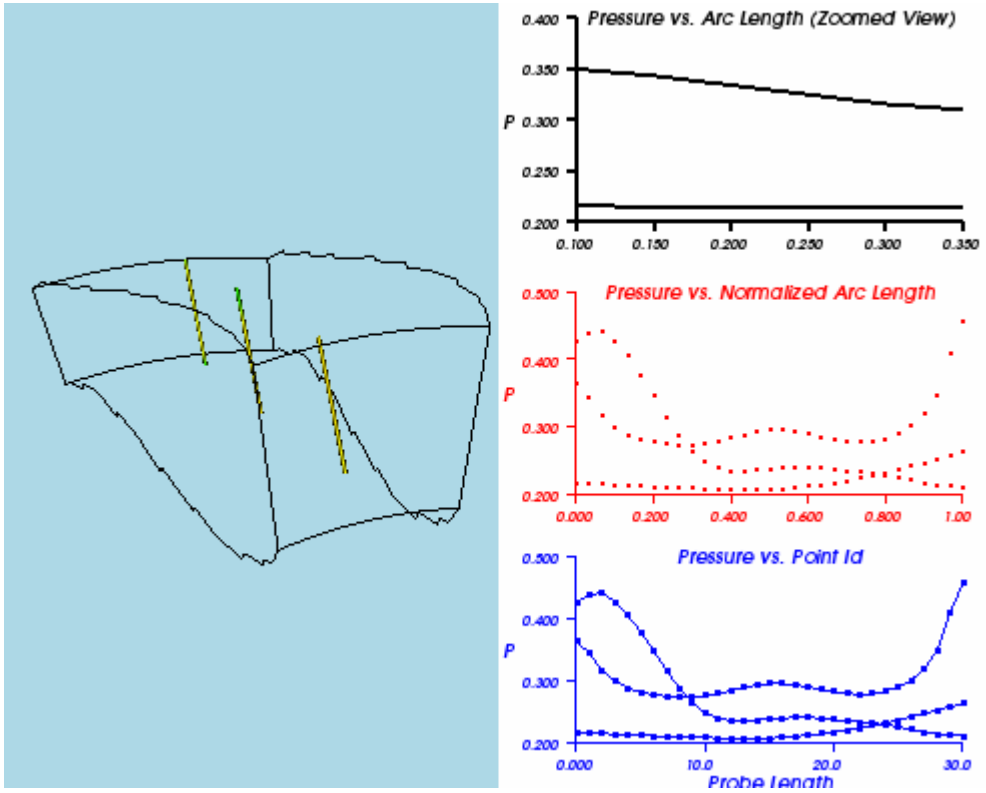
2.10.1 颜色图例

vtkScalarBar 类用于绘制一个分级的颜色图例，不同的颜色对应于不同的标量值，颜色图例主要由三部分组成：用于显示分级颜色的矩形框、注记和标题，一般 vtkScalarBar 类和 vtkLookupTable 类一同使用，下面的示例程序说明了颜色图例的使用方式，程序运行结果如下：

示例程序代码如下（Examp27）：

2.10.2 绘制平面图表

类 vtkXYPlotActor 在 X-Y 平面上绘制图表，如各种折线图、曲线图等，在绘制图表时炫耀输入绘制的数据，建立坐标轴，下面的示例程序说明如何绘制图表，程序的运行结果如下图所示：



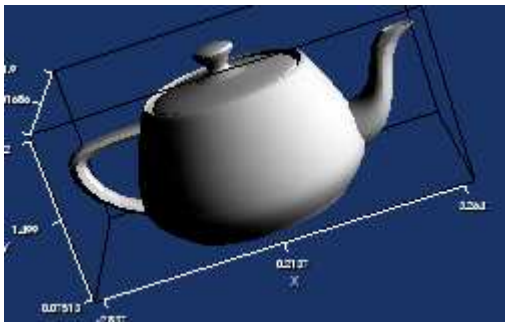
程序代码如下（Examp28）：

//创建图表 1，显示 3 个探查线的结果,x 轴显示弧段长度

```
vtkXYPlotActor *xyplot = vtkXYPlotActor::New();
xyplot->AddInput(probe->GetOutput());
xyplot->AddInput(probe2->GetOutput());
xyplot->AddInput(probe3->GetOutput());
xyplot->GetAxisLabelTextProperty()->SetColor(0 , 0 , 0);
xyplot->GetAxisLabelTextProperty()->SetFontSize(12);
//设置标题文字颜色
xyplot->GetAxisTitleTextProperty()->SetColor(0 , 0 , 0);
//设置线的颜色
xyplot->GetProperty()->SetColor(0 , 0 , 0);
xyplot->GetProperty()->SetLineWidth(2);
xyplot->SetTitle("Pressure vs. Arc Length (Zoomed View)");
xyplot->SetXTitle("");
xyplot->SetYTitle("p");
xyplot->SetXRange(0.1,0.35);
xyplot->SetYRange(0.2,0.4);
xyplot->GetPositionCoordinate()->SetValue(0.0,0.67,0.0);
xyplot->GetPosition2Coordinate()->SetValue(1.0,0.33,0.0);
xyplot->SetNumberOfXLabels(6);
xyplot->SetXValuesToArcLength();
```

2.10.3 显示物体的边界尺寸

类 vtkCubeAxesActor2D 用于标识视口的空间位置，该类在数据集边界框周围绘制坐标轴，并标识坐标值，当视口放大时，坐标轴也随之变化，使之和视口相匹配，并且坐标值也随之更新，下面的示例程序显示了该类的用法，程序运行结果如下：



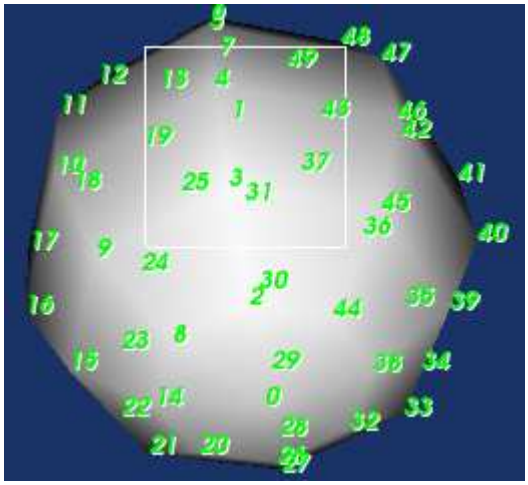
程序示例代码如下（Examp29）：

```
//读取模型文件
vtkBYUReader *fohe = vtkBYUReader::New();
fohe->SetFileName("teapot.g");
fohe->SetPartNumber(0);
fohe->SetReadDisplacement(1);
fohe->SetReadScalar(1);
fohe->SetReadTexture(1);
//计算生成的多边形每个单元的法向量
vtkPolyDataNormals *normals = vtkPolyDataNormals::New();
normals->SetInput((vtkPolyData *) fohe->GetOutput());
//设置两个邻边的特征角
normals->SetFeatureAngle(100);
normals->SetComputeCellNormals(0);
//计算单元的法向量
normals->SetComputePointNormals(1);
normals->SetFlipNormals(0);
normals->SetSplitting(1);
//创建坐标轴
vtkCubeAxesActor2D *axes = vtkCubeAxesActor2D::New();
axes->SetCamera(camera);
axes->SetInput((vtkDataSet *) normals->GetOutput());
axes->SetFontFactor(0.8);
axes->SetLabelFormat("%6.4g");
```

```
axes->SetFlyMode(0);
axes->SetScaling(1);
axes->SetVisibility(1);
```

2.10.4 标识属性数据

在有些应用中，可能要标识数据集点集所对应的属性数据，VTK 提供了 vtkLabeledDataMapper 类用于标识属性数据（包括标量、矢量、张量、纹理坐标、点的 ID 等属性数据），并在一个平面上显示属性数据，下面的示例程序说明了该类的用法，程序运行结果如下图所示：



示例程序代码（Examp210）：

```
//创建一个球体
vtkSphereSource *psphere=vtkSphereSource::New();
vtkPolyDataMapper *sphMap=vtkPolyDataMapper::New();
sphMap->SetInputConnection(psphere->GetOutputPort());
vtkActor *psphereActor=vtkActor::New();
psphereActor->SetMapper(sphMap);
//依据所有点和单元的 ID，生成标量数据
vtkIdFilter *pids=vtkIdFilter::New();
pids->SetInputConnection(psphere->GetOutputPort());
pids->PointIdsOn();
pids->CellIdsOn();
```

```
pids->FieldDataOn();
//创建一个映射器，显示点的 ID
vtkLabeledDataMapper *pcellMapper=vtkLabeledDataMapper::New();
pcellMapper->SetInputConnection(pids->GetOutputPort());
pcellMapper->SetLabelFormat("%g");
pcellMapper->SetLabelModeToLabelFieldData();
pcellMapper->GetLabelTextProperty()->SetColor(0 ,1 ,0);
vtkActor2D *pcellLabels=vtkActor2D::New();
pcellLabels->SetMapper(pcellMapper);
```

2.11 数据变换

当一个 3D 物体在场景中显示时，有时我们只想看物体的某一部分，我们必须重新在场景中放置 3D 物体并且改变物体的位置和方向，使之满足我们的观察要求，在 VTK 中，有些对象提供了和位置、方向相关的属性，我们只需在可视化流水线中改变物体的这些相关属性数据，就可满足我们的观察要求，如，对于一个切割物体的平面来说，我们只需改变它的位置属性数据值，就可以将其放置在流水线的内部，对物体进行切割，对于没有提供位置、方向相关属性的对象来说，我们必须使用 vtkTransformFilter、vtkTransformPolyDataFilter 对对象进行变换，才能使对象移动到新的位置。

vtkTransformFilter 将 vtkPointSet 类型的数据对象作为输入，vtkPointSet 常用来表达离散点的信息，一般用于存储离散点的坐标，vtkTransformFilter 对离散点数据对象进行变换后，生成离散点的变换数组，但离散点数据对象的结构和属性不发生改变，vtkTransformPolyDataFilter 的作用和 vtkTransformFilter 相似，只是它还能对可视化流水线中的多边形对象进行变换。

使用范例如下:

```
vtkTransform *transL1= vtkTransform::New();
transL1->Translate(3.7, 0.0 ,28.37);
transL1->Scale( 5 ,5 ,5);
transL1->RotateY( 90);
vtkTransformPolyDataFilter *tf= vtkTransformPolyDataFilter::New();
```

```
tf->SetInputConnection (line->GetOutputPort());
tf-> SetTransform(transL1);
```

3 可视化技术

在本章将要介绍各种可视化技术，这些技术按照它们所操作的数据类型不同被分类介绍，在 VTK 中有些过滤器能够处理任何类型的数据，如接收 vtkDataSet 类型（或其任何子类）的数据做为其输入，还有一些过滤器只能接收特定类型的数据作为其输入(如：vtkPolyData)，还有一类过滤器只能接收的输入类型为 vtkImageData。

当你阅读这一章时请记住以下两点：

- 过滤器产生各种输出类型，并且输出类型没有必要和输入类型一致；
- 不同的过滤器可以被组合起来创建各种复杂的数据处理流水线。

VTK 提供了一些固定的使用模式或常见的过滤器组合方式，在后面的这些例子中你将会看到这些组合方式。

3.1 可视化 vtkDataSet 类数据

在这一节中， 我们将介绍一些常见的对 vtkDataSet 数据对象（及其子类对象）可视化的方法，因为 vtkDataSet 是所有数据集类的父类，是一个抽象的不能实例化的类，所以，这里所描述的方法对所有的数据集类都适用，换句话说，所有将 vtkDataSet 做为输入的过滤器也可以接收 vtkPolyData、vtkImageData、vtkStructuredGrid、vtkRectilinearGri 和 vtkUnstructuredGrid 等类型做为输入。

属性数据用于描述数据集的点或单元属性信息，VTK 中主要包括点属性数据和单元属性数据，在 VTK 中，属性数据和数据集的结构可以被许多过滤器处理，并且产生新的结构和属性，对属性数据的详细的介绍不属于本节的内容。

在 VTK 中可以使用 vtkDataArrays 类型存储属性数据，属性数据可以被表示为标量、张量、矢量、纹理坐标、法向量等形式，每个属性数据分别与 vtkDataSet 中的点和单元相关联，可以用 vtkDataArray 的子类存储不同类型的属性数据，比如 vtkFloatArray 或 vtkIntArray 类存储浮点型、整型数据，存储属性数据的数组的内存是连续、线性分配的，并且数组由子数组或组元（常常表示成员数目确定，每个成员类型也确定的结构）组成，创建属性数据数组意味着实例化一个所需类型的数据数组，指定组元的大小，插入数据，将属

性数据与数据集关联在一起，用于描述数据集。

创建属性数据由下面几个步骤组成：

１．按数据类型创建一个数组

２．指定数组大小

３．将数组与数据集相关联

如下面的代码所示：

```
//存储标量属性数据
vtkFloatArray *scalar=vtkFloatArray::New();
scalar->insertTuple1(0,1.0);
scalar->insertTuple1(1, 1.0);
```

上述代码设定一个标量数组，数组组元的大小为 1，使用 insertTuple1 方法把数据插入到组元的末尾。

```
//存储矢量属性数据
vtkFloatArray *vector=vtkFloatArray::New();
//设置组元的大小
vector->SetNumberOfComponents(3);
vector->insertTuple3(0, 1.0 ,2.0 ,3.0);
vector->insertTuple3(1, 1.0,  4.0 ,5.0);
```

上述代码设定一个矢量数组，因为矢量一般是由 x,y,z 三个分量组成，所以矢量数组组元的大小设置为 3，分别对应矢量的三个分量，使用 insertTuple3 方法把数据插入到组元的末尾。

```
vtkIntArry *justArry=vtkIntArry::New();
justArry->SetNumberOfComponents(2);
//分配 100 个数据存储空间
justArry->SetNumberOfTuples(100);
justArry->setTuple2(0,1,2);
justArry->setTuple2(1,3,4);
```

上述代码创建了一个组元大小为 2 的数组，用 SetNumberOfTuples 方法分配内存，并用 setTuple2 方法添加数据，该方法用于已经预先分配内存了，所以添加数据的速度要比 insertTuple2 块。

```
vtkPolyDataSet *polyData=vtkPolyDataSet::New();
//标量属性数据和点数据对象关联
polyData->GetPointData()->SetScalars(scalar);
//矢量属性数据和单元数据对象关联
polyData->GetCellData()->SetVectors(vector);
```

上述代码将属性数据数组和数据对象关联。

当我们将属性数据数组和数据对象相关联时，要注意属性数据标注的是标量还是矢量，如果是标量使用方法 SetScalars()方法，如果是矢量使用 SetVectors()方法。

需要注意的是属性数据数目和数据集中点集数目或单元数目的关系，与点相关联的属性数据的数目一定要等于数据集中点的数目，同理，与单元相关联的属性数据数目一定要等于数据集中单元的数目。

类似的，我们可以使用下述方法访问属性数据：

```
//得到标量
Scalars=polyData->GetPointData()->GetScalars();
//得到矢量
Vectors=polyData->GetCellData()->GetVectors();
```

在 VTK 中，许多过滤器只接收特定形式的属性数据作为输入，例如：vtkElevationFilter 过滤器接收在指定方向上的高程值作为输入并且生成新的标量值，也有一些过滤器即接收结构化数据集类型也接收属性数据作为其输入，但是其对属性数据不进行处理，而是直接将它们输出，还有一些过滤器不仅接收属性数据同时也接收结构化数据集类型作为其输入，并产生新的输出，vtkMarchingCubes 是一个典型的例子，它把标量属性数据和结构化的数据集对象同时作为输入，并输出等值线形成的图形（如点、线和三角形）。

另一个有关于属性数据的问题是，有些过滤器在只能处理某种类型的属性数据（点数据与单元数据），对于其不能处理的数据类型，将其忽略或将其直接输出，不予处理。

vtkPointDataToCellData 和 vtkCellDataToPointData 这两个类为我们提供了在不同的属性数据之间进行互相转换的方法（在点属性数据和单元属性数据之间相互转换）。

3.1.1 颜色映射

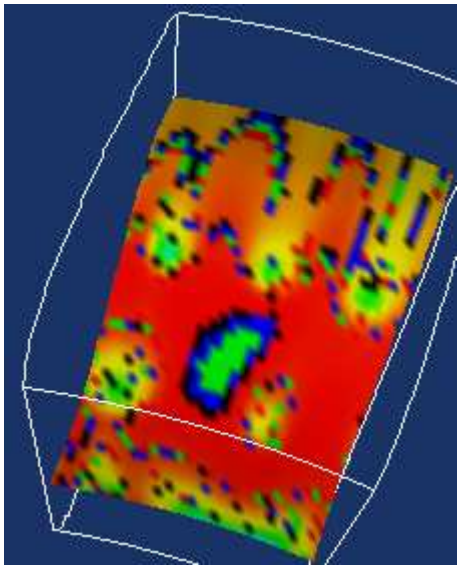
最常用的可视化技术是根据属性值（标量值）对绘制的对象着色或是彩色绘图，这种可

视化技术称为颜色映射算法，颜色映射的思想比较简单：

将颜色查找表（lookup table）中的颜色映射到点或单元的属性值(标量)值上，在绘制阶段使用这些颜色值绘制相应的点或者单元。

在学习本节以前，你必须知道怎么样使用 Actors 控制绘制物体的颜色。

在 VTK 中，颜色映射典型的是由标量值和颜色查找表（被 vtkMapper 实例使用来执行颜色映射）来控制，使用 ColorByArrayComoponent()方法还可以使用数据数组来进行颜色映射，我们可以创建自己的颜色查找表并设定表中的颜色数目和颜色值，如果没有指定颜色查找表，映射器将要创建一个默认的颜色查找表，下面的示例程序说明颜色映射的使用，程序运行结果如下：



示例代码（Examp31）：

```
//读取结构化网格数据文件
vtkPLOT3DReader *pl3d = vtkPLOT3DReader::New();
pl3d->SetFileName("combxyz.bin");
pl3d->SetQFileName("combq.bin");
pl3d->SetScalarFunctionNumber(100);
pl3d->SetVectorFunctionNumber(202);
//从结构化数据集中提取一个几何体
vtkStructuredGridGeometryFilter *plane =
vtkStructuredGridGeometryFilter::New();
plane->SetInput((vtkStructuredGrid *) pl3d->GetOutput());
```

```
//设定平面的范围
plane->SetExtent(1 , 100 , 1 , 100 , 0 , 0);
//设定颜色查找表
vtkLookupTable *lut = vtkLookupTable::New();
//设定透明度范围 0 透明 1 不透明
lut->SetAlphaRange(1 , 1);
 //设定颜色范围
 lut->SetHueRange(0 , 1);
 //设定表中颜色数目
 lut->SetNumberOfTableValues(256);
  vtkPolyDataMapper *planeMapper = vtkPolyDataMapper::New();
 planeMapper->SetInput((vtkPolyData *) plane->GetOutput());
 planeMapper->SetLookupTable(lut);
 planeMapper->SetNumberOfPieces(1);
 //设定被映射的标量值的范围
 planeMapper->SetScalarRange(0 , 0.45);
```

颜色查找表被以两种不同的方式创建，就像这个例子所说明的那样，首先你可以指定一个 HSVA（Hue-Saturation-Value-Alpha transparency）颜色范围，使用线性插值来生成表中的颜色，其次设定好表中颜色的数量后，使用 SetTabValue()方法手动将设定的颜色插入到表中指定的位置。

映射器的 SetScalarRange()方法控制标量值怎样映射到表中，标量值大于最大值的将被归为最大值，标量值小于最小值的将被归为最小值，对于 unsigned char 类型的标量数据其本身就可以作为颜色值直接进行绘制，不需要通过查找表来进行颜色映射。

映射器提供了几种方法来控制映射行为。

SetColorModeToDefault()调用默认的映射器行为，默认的行为是将点或单元的 unsigned char 类型的标量值做为颜色值直接绘制，其它类型的标量值都将通过颜色查找表进行映射。

SetColorModeToMapScalars()将通过查找表映射所有的标量值，如果标量的组元列数(组元分量)大于 1，那么组元的第 0 列分量将要被用于执行映射。

vtkMapper 的另一个重要特征是控制哪一种属性数据(例如点属性数据、单元属性数据、

属性数据数组）被用于颜色映射，VTK 提供了下面的方法执行这些行为，注意这些方法产生明显不同的结果：点属性数据在图元绘制过程中将要被插值处理，而单元属性数据则保持不变。

SetScalarModeToDefault()方法调用默认的映射器行为，默认的行为使用点的属性数据值进行颜色映射，如果点属性数据不存在，而单元属性数据存在，将使用单元属性数据值进行颜色映色。

SetScalarModeToUsePointData()方法总是使用点属性数据来进行颜色映射，如果点属性数据不存在，不进行映射。

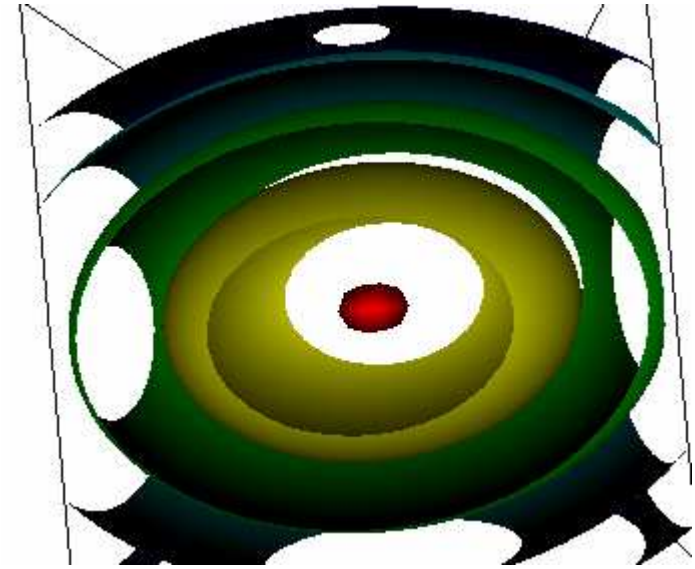
SetScalarModeToUseCellData()方法总是使用单元属性数据来进行映射，如果单元属性数据不存在，不进行映射。

SetScalarModeToUsePointFieldData()方法表明既不使用点也不使用单元属性数据，而是使用点属性数据中的一个数据数组进行颜色映射，这个方法应该结合 ColorByArrayComponent()方法来指定数据数组和用作标量值的部分。

SetScalarModeToUseCellFieldData()方法表明既不使用点也不使用单元属性数据，而是使用单元属性数据中的一个数据数组进行颜色映射。这个方法应该结合 ColorByArrayComponent()方法来指定数据数组和用作标量值的部分。

3.1.2 轮廓提取

另一个常见的可视化技术生成轮廓（contours），轮廓是由确定的标量值生成的线或面，在 VTK 中，过滤器 vtkContourFilter 被用于提取轮廓，下面的示例程序显示了如何提取轮廓，程序的运行结果如下：



示例程序代码（Examp32）：

```
//创建一个二次曲面
vtkQuadric *quadric = vtkQuadric::New();
//设置二次曲面系数
quadric->SetCoefficients(0.5 , 1 , 0.2 , 0 , 0.1 , 0 , 0 , 0.2 , 0 , 0);
//对隐函数进行采样，生成结构化数据集
vtkSampleFunction *sample = vtkSampleFunction::New();
sample->SetImplicitFunction(quadric);
sample->SetCapValue(1e+038);
//设定采样边界范围
sample->SetModelBounds(-1 , 1 , -1 , 1 , -1 , 1);
//设定维度大小
sample->SetSampleDimensions(300 , 300 , 300);
sample->SetOutputScalarType(10);
sample->SetCapping(0);
sample->SetComputeNormals(1);
//提取轮廓表面
vtkContourFilter *contours = vtkContourFilter::New();
contours->SetInput((vtkDataSet *) sample->GetOutput());
//提取标量值为 0 的轮廓面
```

```
contours->SetValue(0 , 0);
//提取标量值为 0.3 的轮廓面
contours->SetValue(1 , 0.3);
contours->SetValue(2 , 0.6);
contours->SetValue(3 , 0.9);
contours->SetValue(4 , 1.2);
contours->SetComputeNormals(1);
contours->SetComputeScalars(1);
```

可以有两种方式来指定标量值，最简单的方式是使用 SetValue()方法来指定轮廓数和它的值（同时可以设定多个值），如：

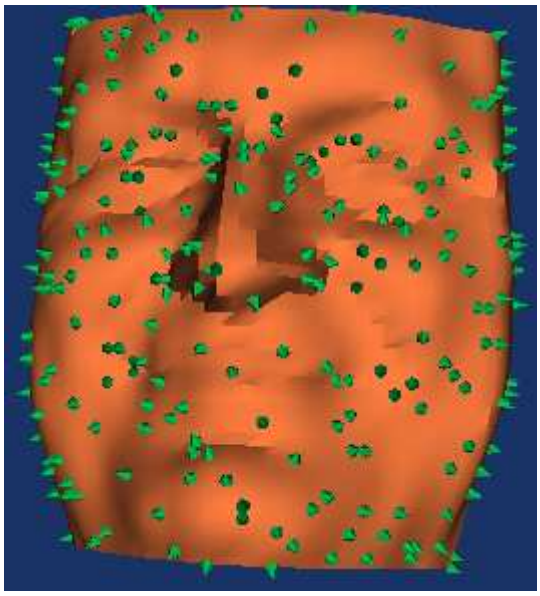
```
contours SetValue 0 0.5
```

第二种方式：使用方法 GenerateValues()，你可以指定标量值的范围和在这个范围内将被生成的轮廓的数量。

在 VTK 中有几种对象来执行特殊数据集类型的轮廓提取,例如 vtkMarchingCubes，vtkMarchingSquares 等。

3.1.3 符号化

符号化是使用符号表示数据的一种可视化技术，符号可以简单也可以非常复杂， 在 VTK 中 vtkGlyph3D 类可以创建一个具有大小、方向和用颜色表示的符号，输入数据对象中的每一个点都可以用符号来可视化，下面的示例程序说明符号可视化的用法，程序的运行结果如下图所示：



示例程序代码（Examp33）：

```
//读取模型文件
vtkPolyDataReader *fran = vtkPolyDataReader::New();
fran->SetFileName("fran_cut.vtk");
//计算模型每个面的法矢量
vtkPolyDataNormals *normals = vtkPolyDataNormals::New();
normals->SetInput((vtkPolyData *) fran->GetOutput());
normals->SetFeatureAngle(30);
normals->SetComputeCellNormals(0);
normals->SetComputePointNormals(1);
normals->SetFlipNormals(1);
normals->SetSplitting(1);
vtkPolyDataMapper *franMapper = vtkPolyDataMapper::New();
franMapper->SetInput((vtkPolyData *) normals->GetOutput());
//过滤数据集中的点和属性，生成可显示的点
vtkMaskPoints *ptMask = vtkMaskPoints::New();
ptMask->SetInput((vtkDataSet *) normals->GetOutput());
//创建符号
vtkConeSource *cone = vtkConeSource::New();
cone->SetAngle(26.5651);
```

```
cone->SetHeight(1);
cone->SetRadius(0.5);
cone->SetResolution(6);
cone->SetCapping(1);
vtkGlyph3D *glyph = vtkGlyph3D::New();
//设置被标识的点
glyph->SetInput((vtkDataSet *) ptMask->GetOutput());
//设置符号
glyph->SetSource(cone->GetOutput());
```

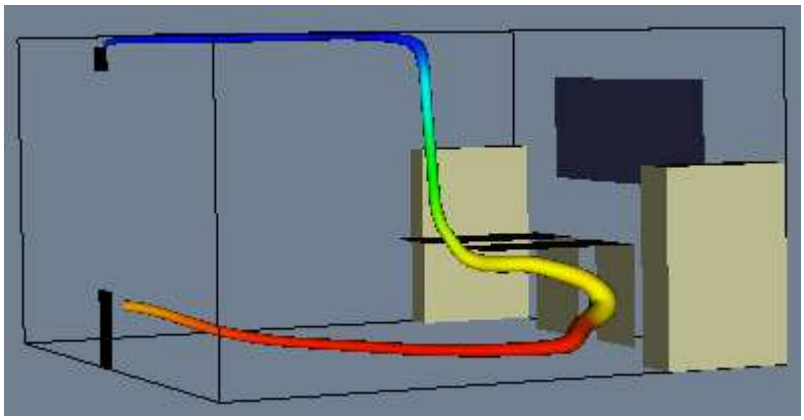
这个示例程序首先读取模型文件，作为输入数据，接着 vtkMaskPoints 过滤器对输入数据进行二次采样，并将采样结果作为 vtkGlyph3D 对象的输入，vtkConeSource 对象创建一个圆锥符号，表示曲面的法线方向。

vtkGlyph3D 对象符号被配置使用点属性法矢量作为方向向量，（另外也可以使用 SetVectorModeToUseVector（）方法用矢量数据代替法矢量），并根据矢量值的大小，按比例绘制符号的大小。

3.1.4 流线技术

一条流线可以被认为是质点在矢量场中的运动路径，流线被常用来描述矢量场的结构，通常用多条流线来研究矢量场中感兴趣的特征要素，流线通常用数值积分的方式被计算(对速率次数的积分)，因此得到流线是实际流线的近似表达。

创建一个流线需要指定一个开始点(如果多条流线需要指定一个点集)、一个流线积分的方向(流程方向、流程的反方向，或者正反两个方向)和一个控制它的传播的参数，下面示例程序将显示出如何创造一单个流线，程序运行结果如下图所示：



示例程序代码（Examp34）：

```
vtkStructuredGridReader *reader = vtkStructuredGridReader::New();
reader->SetFileName("office.binary.vtk");
//用龙格—库塔方法求解微分方程
vtkRungeKutta4 *integ = vtkRungeKutta4::New();
//创建流线对象，该对象为过滤器对象，接受任何类型数据作为输入
vtkStreamLine *streamer = vtkStreamLine::New();
 streamer->SetInput((vtkDataSet *) reader->GetOutput());
 //设定积分类型
 streamer->SetIntegrator(integ);
 //设定积分步长
 streamer->SetIntegrationStepLength(0.15);
 //设定传播的最长时间
 streamer->SetMaximumPropagationTime(500);
 streamer->SetNumberOfThreads(2);
 //设置积分的初始位置点
 streamer->SetStartPosition(0.1 , 2.1 , 0.5);
 //设置每段流线的长度
 streamer->SetStepLength(2.5);
 streamer->SetTerminalSpeed(0);
 streamer->SetIntegrationDirection(2);
 streamer->SetOrientationScalars(0);
 streamer->SetSpeedScalars(0);
 streamer->SetVorticity(0);
  vtkTubeFilter *streamTube = vtkTubeFilter::New();
 streamTube->SetInput((vtkPolyData *) streamer->GetOutput());
 streamTube->SetNumberOfSides(12);
 streamTube->SetRadius(0.02);
 streamTube->SetUseDefaultNormal(0);
```

```
vtkPolyDataMapper *mapStreamTube = vtkPolyDataMapper::New();
mapStreamTube->SetInput((vtkPolyData *) streamTube->GetOutput());
vtkActor *streamTubeActor = vtkActor::New();
streamTubeActor->SetMapper(mapStreamTube);
```

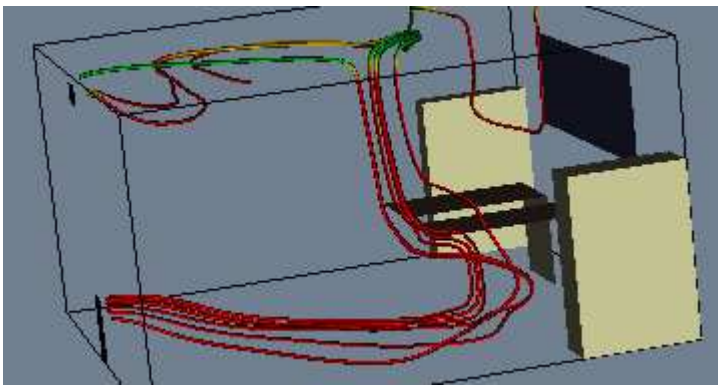
在这个例子中，我们选择世界坐标系的（0.1，2.1，0.5）作为开始点，当然，我们也可选择单元 ID、单元子 ID 和参量的坐标指定开始位置，MaximuinPropagationTime（）方法控制着最大流线的长度(单位时间测量的长度)，StepLength（）方法 控制输出流线线段的尺度， IntegrationStepLength（）方法可以设定的数值范围在 0 到 1 之间，用于表明步长（单元长度/单元对角线长度），如果想要提高流线输出的精度(要花费较多计算时间)设定 IntegrationStepLength 步长为一个比较小的值，选择 vtkInitialValueProblemSolve 类的不同子类也可以实现输出流线精度的提升，如选择 vlkRungcKulta4 可以使流线输出提升的更高。(在缺省的情况下，streamer 类使用 vtkRungcKutta2 执行数值积分)。

使用下面的方法，可以控制流线的积分方向：

- SetIntegrationDirectionToIntegrateForward（）
- SetIntegrationDirectionToIntegrateBack ward（）
- SetIntegrationDirectionToIntegrateBothDirections（）

流线在日常生活中是不可见的，为了直观的表示流线，在这个例子中，使用一个管线来代表流线，管径和流线流动的速率成反比，流速快的地方，管径变粗，流动慢的地方，管径变细。

有时我们希望能同时产生多条流线,这时可以用SetSource ()方法设定一个vtkDataSet 数据对象作为其输入参数，使用数据对象中包含的点集数据作为流线的起点，下面给出一个示例程序，该程序随机生成 6 个数据点，作为流线的起点生成 6 条流线，程序运行结果如下：

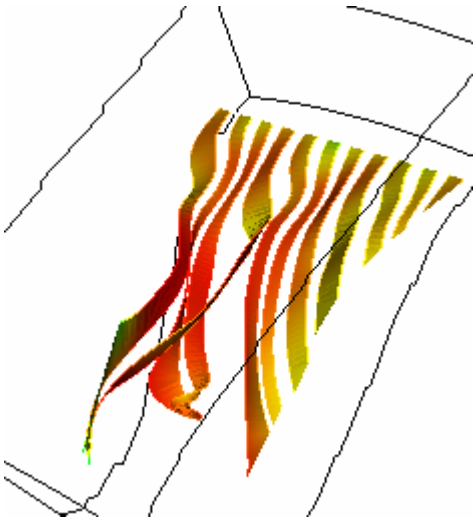


示例代码如下(Examp35)：

```
//随机生成云点
vtkPointSource *seeds = vtkPointSource::New();
seeds->SetCenter(0.1 , 2.1 , 0.5);
//设定生成的点的数目
seeds->SetNumberOfPoints(6);
seeds->SetRadius(0.15);
seeds->SetDistribution(1);
vtkStreamLine *streamer = vtkStreamLine::New();
streamer->SetInput((vtkDataSet *) reader->GetOutput());
streamer->SetIntegrator(integ);
//随机生成的 6 个点作为起始点
streamer->SetSource(seeds->GetOutput());
streamer->SetIntegrationStepLength(0.05);
streamer->SetMaximumPropagationTime(500);
streamer->SetNumberOfThreads(2);
streamer->SetStartPosition(0 , 0 , 0);
streamer->SetStepLength(0.5);
streamer->SetTerminalSpeed(0);
streamer->SetIntegrationDirection(2);
streamer->SetOrientationScalars(0);
streamer->SetSpeedScalars(0);
streamer->SetVorticity(0);
vtkTubeFilter *streamTube = vtkTubeFilter::New();
streamTube->SetInput((vtkPolyData *) streamer->GetOutput());
streamTube->SetRadius(0.02);
vtkPolyDataMapper *mapStreamTube = vtkPolyDataMapper::New();
mapStreamTube->SetInput((vtkPolyData *) streamTube->GetOutput());
vtkActor *streamTubeActor = vtkActor::New();
streamTubeActor->SetMapper(mapStreamTube);
```

3.1.5 流面

高级的用户想要使用 VTK 的流面功能，流面的产生需要经过以下两个步骤：首先用一系列有序的点生成一系列流线，然后 vtkRuledSurfaceFilter 过滤器用一系列流线创建一个表面，点的排序对流面的生成有非常大的影响，不同的排序方式，生成的流面也不相同，如果排序不正确，可能得不到预期的结果，所以一定要重视点的排序，下面的示例程序说明如何生成流面，程序运行结果如下：



示例程序代码(Examp36)：

```
//创建一条多义线，沿线生成流线的起始点
vtkLineSource *rake = vtkLineSource::New();
//设置起始点
rake->SetPoint1(15 , -5 , 32);
//设置终点
rake->SetPoint2(15 , 5 , 32);
//设置直线由多少小线段组成，每个小线段可作为流线的起始点
rake->SetResolution(10);
vtkPolyDataMapper *rakeMapper = vtkPolyDataMapper::New();
rakeMapper->SetInput((vtkPolyData *) rake->GetOutput());
vtkActor *rakeActor = vtkActor::New();
rakeActor->SetMapper(rakeMapper);
rakeActor->GetProperty()->SetColor(1.0,0.0,0.0);
```

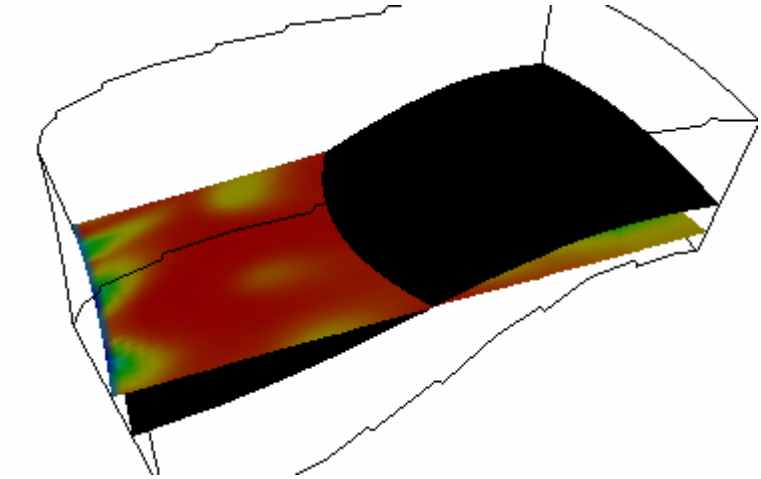
```
//积分方程
vtkRungeKutta4 *integ = vtkRungeKutta4::New();
//创建流线
vtkStreamLine *sl = vtkStreamLine::New();
sl->SetInput((vtkDataSet *) pl3d->GetOutput());
sl->SetIntegrator(integ);
//起始点由多义线生成，多义线的线段数目，
//决定流线的数目
sl->SetSource(rake->GetOutput());
sl->SetIntegrationStepLength(0.1);
sl->SetMaximumPropagationTime(0.1);
sl->SetNumberOfThreads(2);
sl->SetStartPosition(0 , 0 , 0);
sl->SetStepLength(0.001);
sl->SetTerminalSpeed(0);
sl->SetIntegrationDirection(1);
sl->SetOrientationScalars(0);
sl->SetSpeedScalars(0);
sl->SetVorticity(0);
//根据一系列的线，生成表面
vtkRuledSurfaceFilter *scalarSurface = vtkRuledSurfaceFilter::New();
scalarSurface->SetInput((vtkPolyData *) sl->GetOutput());
//设定相邻两条线间连接的最大距离
scalarSurface->SetDistanceFactor(30);
scalarSurface->SetOffset(0);
//控制是否生成规则条带，如果大于 1,条带之间有偏移
scalarSurface->SetOnRatio(0);
scalarSurface->SetResolution(1 , 1);
scalarSurface->SetRuledMode(1);
scalarSurface->SetCloseSurface(0);
```

```
scalarSurface->SetPassLines(1);
 vtkPolyDataMapper *mapper = vtkPolyDataMapper::New();
mapper->SetInput((vtkPolyData *) scalarSurface->GetOutput());
 vtkActor *actor = vtkActor::New();
actor->SetMapper(mapper);
```

如果有多重的线作为 vtkRuledSurfaceFilter 过滤器的输入，过滤器能过滤掉重复的线，这有助于我们更好的理解流面结构。

3.1.6 切割

在ＶＴＫ中对对象进行切割或者切片首先需要先创建一个切面，切面我们可以通过隐函数（如平面方程）创建，创建切面后，我们可以在任意方向上利用切面切割物体，当物体被切割后，物体的切割表面使用插值的方式重新构建，切割后的数据一般都是 vtkPolyData 类型， 对一个 n 维单元的数据切割后会生成一个(n-1）维图元输出（如一个四面体，其单元为三维四面体结构，对其切割后，切割面是三角形或四边形图元），下面的示例程序说明了切割的用法，程序运行结果如下：



程序示例代码如下（Examp37）：

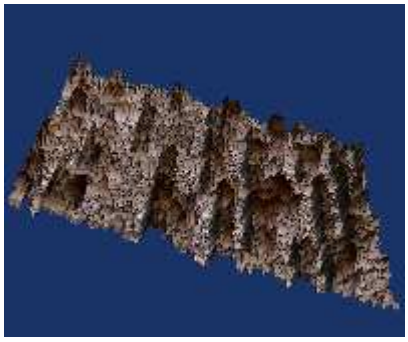
```
//创建剪切平面
vtkPlane *plane = vtkPlane::New();
//设定剪切平面在被剪切的数据中心位置
pStrData=pl3d->GetOutput();
plane->SetOrigin(pStrData->GetCenter());
```

```
plane->SetNormal(-0.287, 0, 0.9579);
//过滤器类，利用隐函数对数据进行剪切
vtkCutter *planeCut = vtkCutter::New();
planeCut->SetInputConnection(pl3d->GetOutputPort());
//设定隐函数
planeCut->SetCutFunction(plane);
//计算数据集在指定点的位置的属性
vtkProbeFilter *probe = vtkProbeFilter::New();
//计算平面在数据集通过点的属性
probe->SetInputConnection(planeCut->GetOutputPort());
//被计算的数据集
probe->SetSourceConnection(pl3d->GetOutputPort());
vtkDataSetMapper *cutMapper = vtkDataSetMapper::New();
cutMapper->SetInputConnection(probe->GetOutputPort());
//得到数据集属性值的范围
vtkPointData *pData=vtkPointData::New();
pStrData=pl3d->GetOutput();
pData=pStrData->GetPointData();
//pData->GetScalars()->GetRange();
//设定属性值范围
cutMapper->SetScalarRange(pData->GetScalars()->GetRange());
cutMapper->ScalarVisibilityOn();
//cutMapper->AddClippingPlane(plane);
vtkActor *cutActor=vtkActor::New();
cutActor->SetMapper(cutMapper);
```

vtkCutter 需要指定一个隐函数用于建立切割平面，另外，如果想要设定一个或多个切割值，可以使用 vtkCutter 的 SetValue()或 GenerateValues()方法，这些值设定用来切割的隐函数的值（典型的切割值是 0，意味着切割平面恰恰在隐函数所定义的平面上，值小于 0 或大于 0 代表切割面在隐函数平面之下或之上，切割值可以被看作是到切割平面的距离）。

3.1.7 数据合并

本节将要介绍如何用 VTK 提供的 vtkMergeFilter 过滤器构建其它数据类型的方法，vtkMergeFilter 过滤器可以从几个不同的数据集对象中提取数据并将这些数据合并到一个新的数据集对象中，例如，你可以从第一个数据集对象中提取结构（拓扑或者几何），从第二个数据集对象中提取标量数据，从第三个数据集中提取矢量数据，最终将他们合并到一个新的数据集中，下面的示例程序给出该过滤器的用法。程序的运行结果如下：



示例代码如下（Examp38）：

```
//读取图像数据，并计算亮度值
vtkBMPReader *reader = vtkBMPReader::New();
reader->SetFileName("masonry.bmp");
//计算亮度值，输出 vtkImageData 数据集类型
vtkImageLuminance *luminance = vtkImageLuminance::New();
luminance->SetInputConnection(reader->GetOutputPort());
//从结构化数据集中提取几何体，输出 vtkPolyData 数据集类型
vtkImageDataGeometryFilter *geometry = vtkImageDataGeometryFilter::New();
geometry->SetInputConnection(luminance->GetOutputPort());
//使用标量数据对提取的几何体变形
vtkWarpScalar *warp = vtkWarpScalar::New();
warp->SetInputConnection(geometry->GetOutputPort());
//设定缩放的位移值
warp->SetScaleFactor(-0.1);
//合并初始的图像和变形的几何体，输出和输入的类型相同
vtkMergeFilter *merge = vtkMergeFilter::New();
//指定提取结合信息的对象
```

```
merge->SetGeometryConnection(warp->GetOutputPort());
//指定提取标量信息的对象
merge->SetScalarsConnection(reader->GetOutputPort());
vtkDataSetMapper *mapper = vtkDataSetMapper::New();
mapper->SetInputConnection(merge->GetOutputPort());
mapper->SetScalarRange(0, 255);
vtkActor *actor = vtkActor::New();
actor->SetMapper(mapper);
```

在这个示例程序中，来自于 vtkWarpScalar 对象的几何体和来自于 vtkBMPReader 的标量数据合并，因为几何体要单独处理，所以流水线先分开处理几何体，然后在合并到一起。

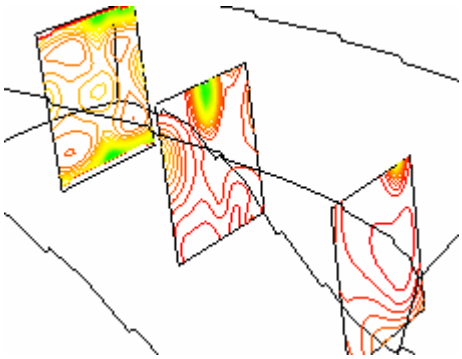
需要注意的是在合并数据时，属性数组元组（数组的列数）的数目必须和点（或单元）的数目相等。

3.1.8 附加数据（Appending Data）

与 vtkMergeFilter 过滤器相似， vtkAppendFilter 过滤器通过附加数据生成一个新的数据集，vtkAppendFilter 过滤器接收多个数据类型相同的数据集对象作为输入，在附加操作过程中，只有所有数据集共有的属性数据才会被添加到一起，示例程序请参考 Examp39。

3.1.9 探查(Probing)

探查是一个数据集对象对另一个数据集对象进行采样的过程，在 VTK 中，你可以使用任何数据集作为一个探测体（如探测平面）进入被探测的数据集对象内部，并计算当探测体通过被探测数据集对象时，被探测对象所对应点的属性值，下面的示例程序说明了探查的使用，程序运行结果如下：



示例程序代码(Examp39)：

```
//读取结构化数据网格
vtkPLOT3DReader *pl3d = vtkPLOT3DReader::New();
pl3d->SetFileName("combxyz.bin");
pl3d->SetQFileName("combq.bin");
pl3d->SetScalarFunctionNumber(100);
 pl3d->SetVectorFunctionNumber(202);
//提取结构化网格外轮廓
vtkStructuredGridOutlineFilter                    *outline                    =
vtkStructuredGridOutlineFilter::New();
outline->SetInput((vtkStructuredGrid *) pl3d->GetOutput());
//数据映射
vtkPolyDataMapper *outlineMapper = vtkPolyDataMapper::New();
outlineMapper->SetInput((vtkPolyData *) outline->GetOutput());
vtkActor *outlineActor = vtkActor::New();
outlineActor->SetMapper(outlineMapper);
outlineActor->GetProperty()->SetColor(0,0,0);
//建立探测平面
vtkPlaneSource *plane=vtkPlaneSource::New();
plane->SetCenter(0 , 0 , 0);
plane->SetNormal(0 , 0 , 1);
plane->SetOrigin(-0.5 , -0.5 , 0);
plane->SetPoint1(0.5 , -0.5 , 0);
plane->SetPoint2(-0.5 , 0.5 , 0);
plane->SetXResolution(50);
plane->SetYResolution(50);
//对探测平面进行变换，生成三个探测平面
vtkTransform *transP1 = vtkTransform::New();
transP1->Translate(3.7,0.0,28.37);
transP1->Scale(5,5,5);
```

```
transP1->RotateY(90);
vtkTransformPolyDataFilter *tpd1 = vtkTransformPolyDataFilter::New();
tpd1->SetInput((vtkPolyData *) plane->GetOutput());
tpd1->SetTransform(transP1);
//变换 1
vtkOutlineFilter *outTpd1 = vtkOutlineFilter::New();
outTpd1->SetInput((vtkDataSet *) tpd1->GetOutput());
vtkPolyDataMapper *mapTpd1 = vtkPolyDataMapper::New();
mapTpd1->SetInput((vtkPolyData *) outTpd1->GetOutput());
vtkActor *tpd1Actor = vtkActor::New();
tpd1Actor->SetMapper(mapTpd1);
tpd1Actor->GetProperty()->SetColor(0 , 0 , 0);
//变换 2
vtkTransform *transP2 = vtkTransform::New();
transP2->Translate(9.2,0.0,31.2);
transP2->Scale(5,5,5);
transP2->RotateY(90);
vtkTransformPolyDataFilter *tpd2 = vtkTransformPolyDataFilter::New();
tpd2->SetInput((vtkPolyData *) plane->GetOutput());
tpd2->SetTransform(transP2);
vtkOutlineFilter *outTpd2 = vtkOutlineFilter::New();
outTpd2->SetInput((vtkDataSet *) tpd2->GetOutput());
vtkPolyDataMapper *mapTpd2 = vtkPolyDataMapper::New();
mapTpd2->SetInput((vtkPolyData *) outTpd2->GetOutput());
vtkActor *tpd2Actor = vtkActor::New();
tpd2Actor->SetMapper(mapTpd2);
tpd2Actor->GetProperty()->SetColor(0 , 0 , 0);
…
//附加数据
vtkAppendPolyData *appendF=vtkAppendPolyData::New();
```

```
appendF->AddInput(tpd1->GetOutput());
appendF->AddInput(tpd2->GetOutput());
appendF->AddInput(tpd3->GetOutput());
//对读取的数据进行探测，计算通过探测平面对应点的属性值
vtkProbeFilter *probe = vtkProbeFilter::New();
//输入探测平面
probe->SetInput((vtkDataSet *) appendF->GetOutput());
//输入被探测的数据
probe->SetSource(pl3d->GetOutput());
probe->SetSpatialMatch(0);
```

上面的程序创建了三个平面作为探查平面，用这三个平面对一个结构化的网格数据集进行采样，其后这三个平面被 vtkContourFilter 过滤器处理生成了轮廓线。

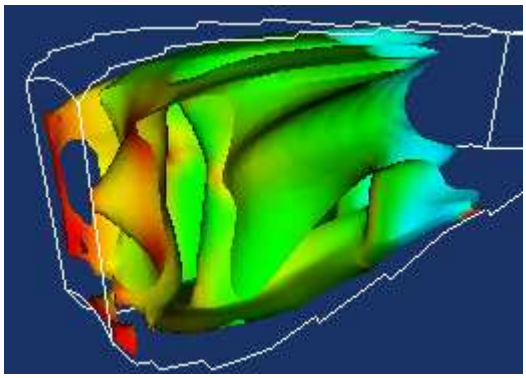
需要注意的是：探查体被设定为 vtkProbeFilter 过滤器的输入，被探查的数据对象为 vtkProbeFilter 过滤器的源。

探查的主要应用是对数据重采样，例如，有一个非结构化网格数据，如果想用 vtkImageData 对象可视化这个数据，可以使用 vtkProbeFilte 对非结构化网格数据重新采样，生成体数据，然后对生成的体数据进行可视化，需要说明的是使用直线或曲线探查数据集对象也是可行的，不仅仅限于点。

最需要注意的是：切割和探查可以得到类似的结果，但它们的含义不同，切割创建的切割表面的清晰度取决于输入数据的质量，而探查创建的表面的清晰度和输入的数据无关，探测数据时应避免过多采样，因为过多采样可能会导致错误的可视化，并且消耗过多的计算时间。

3.1.10 为等值面分级着色

一个常用的可视化方法是产生一个等值面，然后根据等值面的数值不同，为其分级着色，用探查技术来做这些是更加有效的方法，这是因为 vtkConcourFilter 过滤器在生成等值面过程中，对所有的数据进行插值计算，插值的数据可以作为标量数据用来映射等值面的颜色，下面的示例程序说明了相关的用法，程序运行结果如下：



示例程序代码（Examp310）：

```
//读取结构化网格数据
vtkPLOT3DReader *pl3d = vtkPLOT3DReader::New();
pl3d->SetFileName("combxyz.bin");
pl3d->SetQFileName("combq.bin");
pl3d->SetScalarFunctionNumber(100);
pl3d->SetVectorFunctionNumber(202);
//用速度的大小作为标量值
pl3d->AddFunction(153);
//提取等值面
vtkContourFilter *iso = vtkContourFilter::New();
iso->SetInput((vtkDataSet *) pl3d->GetOutput());
iso->SetValue(0, 0.24);
iso->SetValue(1, 0.34);
iso->SetComputeNormals(1);
iso->SetComputeScalars(1);
//计算多边形的法矢量
vtkPolyDataNormals *normals = vtkPolyDataNormals::New();
normals->SetInput((vtkPolyData *) iso->GetOutput());
normals->SetFeatureAngle(45);
normals->SetComputeCellNormals(0);
normals->SetComputePointNormals(1);
normals->SetFlipNormals(0);
normals->SetSplitting(1);
```

```
vtkPolyDataMapper *isoMapper = vtkPolyDataMapper::New();
isoMapper->SetInput((vtkPolyData *) normals->GetOutput());
isoMapper->SetScalarRange(0,1500);
//设定使用标量值的形式
isoMapper->SetScalarModeToUsePointFieldData();
//使用速度作为标量值
isoMapper->ColorByArrayComponent("VelocityMagnitude",0);
vtkLODActor *isoActor = vtkLODActor::New();
isoActor->SetMapper(isoMapper);
```

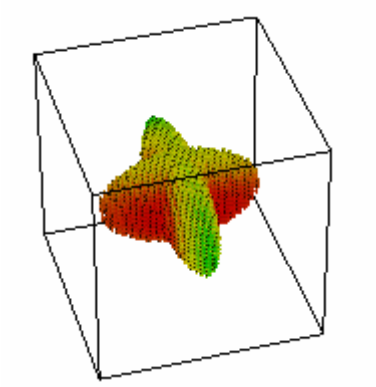
首先，数据对象被 vtkPLOT3DReader 读入，我们添加一个函数来读（功能号为 153），称为“Velocity Magnitude”的数据，等值面生成过程中，将所有的输入数据进行插值，包括“Velocity Magnitude”数据，然后调用 SetScalarModeToUsePointFieldData()方法说明使用“Velocity Magnitude”数据作为标量值映射等值面的颜色，用 ColorByArrayComponent ()方法设定使用“Velocity Magnitude”数据的那个组分作为标量。

3.1.11 提取单元数据的子集

被可视化的数据量通常非常大，处理这些数据花费的代价非常大，很多时候我们需要处理的只是整个数据中的一部分数据，因此，VTK 具有从数据集中提取部分数据的功能是非常重要的，在 VTK 中提供了一些用于提取部分数据或对数据进行采样的工具，前面我们已经看到了如何用 vtkProbeFilter 过滤器对数据进行二次采样，在 VTK 中对数据进行提取的工具主要包括：

- 二次采样工具
- 在空间矩形区域内提取单元数据工具

在本节中，我们将描述如何在一个区域空间中从数据集对象中提取部分数据的方法，类 vtkExtractGeometry 用于提取隐函数确定的数据集对象中的所有单元，下面的示例程序说明了该过滤器的用法，程序运行结果如下：



程序示例代码（Examp311）：

```
//二次曲面隐函数定义二次曲面
vtkQuadric *quadric = vtkQuadric::New();
quadric->SetCoefficients(0.5 , 1 , 0.2 , 0 , 0.1 , 0 , 0 , 0.2 , 0 , 0);
//对隐函数进行二次采样，生成结构化数据集，并计算每个点的值
vtkSampleFunction *sample = vtkSampleFunction::New();
sample->SetImplicitFunction(quadric);
//设定结构化数据集的大小
sample->SetSampleDimensions(50,50,50);
sample->SetComputeNormals(0);
vtkTransform *trans = vtkTransform::New();
trans->Scale(1,0.5,0.333);
//球体 1 隐函数
vtkSphere *sphere = vtkSphere::New();
sphere->SetTransform(trans);
sphere->SetRadius(0.25);
vtkTransform *trans2 = vtkTransform::New();
trans2->Scale(0.25,0.5,1.0);
//球体 2 隐函数
vtkSphere *sphere2 = vtkSphere::New();
sphere2->SetTransform(trans2);
sphere2->SetRadius(0.25);
//对球体 1 和球体 2 进行并布尔运算
```

```
vtkImplicitBoolean *xunion = vtkImplicitBoolean::New();
xunion->SetOperationType(0);
xunion->AddFunction(sphere);
xunion->AddFunction(sphere2);
//从结构化数据集中提取单元，在隐函数内部的单元
vtkExtractGeometry *extract = vtkExtractGeometry::New();
extract->SetInput((vtkDataSet *) sample->GetOutput());
//设定隐函数
extract->SetImplicitFunction(xunion);
//对生成的非结构化数据集的每个单元进行收缩
vtkShrinkFilter *shrink = vtkShrinkFilter::New();
shrink->SetInput((vtkDataSet *) extract->GetOutput());
shrink->SetShrinkFactor(0.6);
vtkDataSetMapper *dataMapper = vtkDataSetMapper::New();
dataMapper->SetInput(shrink->GetOutput());
vtkActor *dataActor = vtkActor::New();
dataActor->SetMapper(dataMapper);
```

vtkExtractGeomery 的输出始终是 vtkUnstructuredGrid 类型，这是因为在数据提取过程中会破坏数据对象的拓扑结构，并且 vtkUnstructuredGrid 类型也是一种通用的数据对象输出格式。

需要注意的是也能使用 vtkTransform 对隐函数进行变换，该变换将要修改隐函数的值。

3.1.12 提取单元作为多边形数据(vtkPolyData)

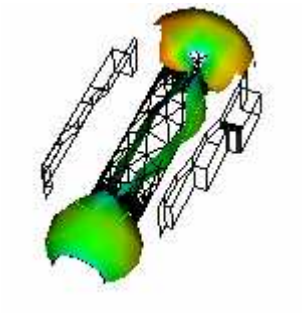
大部分的数据集类型不能直接被图形硬件或图形库绘制，图形库只能绘制多边形数据类型（vtkPolyData）、结构点集、图像和体数据，其它的数据集类型如果要绘制的话，必需经过特殊的处理，在 VTK 中，如果要绘制非多边形数据对象，需要使用 vtkGeometryFilter 过滤器将非多边形数据对象转化为多边形数据对象。

vtkGeometryFilter 过滤器接收任何类型的数据集对象作为输入，并且输出 vtkPolyData 类型数据，数据的转换遵循下述规则：

1) 所有二维、一维数或者无维数据(如：线、点、多边形)被直接输出。

2) 三维数据场的边界数据（指仅被一个网格共享的数据）也直接输出。

另外，vtkGeometryFilter 过滤器还提供了一些方法，这些方法可以由开发者自己根据点 ID 号或单元 ID 号设定提取范围提取该范围内的单元,或者开发者自己可以设定一个的矩形空间区域提取该区域内的单元，下面的示例程序说明了该过滤器的使用方法，程序运行结果如下：



示例程序代码（Examp312）：

```
//读取非结构化数据网格
vtkUnstructuredGridReader *reader = vtkUnstructuredGridReader::New();
reader->SetFileName("blow.vtk");
//设定被提取的标量数据名称，在 vtk 数据文件中
reader->SetScalarsName("thickness9");
reader->SetVectorsName("displacement9");
//显示非结构化网格数据
vtkDataSetMapper *moldMapper = vtkDataSetMapper::New();
moldMapper->SetInput(reader->GetOutput());
vtkActor *moldActor = vtkActor::New();
moldActor->SetMapper(moldMapper);
moldActor->GetProperty()->SetColor(0.2 , 0.2 , 0.2);
//moldActor->GetProperty()->SetRepresentationToWireframe();
//将点数据转换成单元数据
vtkPointDataToCellData *p2c = vtkPointDataToCellData::New();
p2c->SetInput((vtkDataSet *) reader->GetOutput());
p2c->SetPassPointData(1);
```

```
//用矢量数据对几何体变形
vtkWarpVector *warp = vtkWarpVector::New();
//将单元数据以非结构化网格的方式输出
warp->SetInput(p2c->GetUnstructuredGridOutput());
//设定位移系数
warp->SetScaleFactor(1.0);
//从数据集中提取满足阈值条件的单元
vtkThreshold *thresh = vtkThreshold::New();
thresh->SetInput(warp->GetUnstructuredGridOutput());
//设定阈值范围
thresh->ThresholdBetween(0.0,1);
//根据几何的连通性提取数据
vtkConnectivityFilter *connect2 = vtkConnectivityFilter::New();
connect2->SetInput((vtkDataSet *) thresh->GetOutput());
vtkGeometryFilter *parison = vtkGeometryFilter::New();
parison->SetInput((vtkDataSet *) connect2->GetOutput());
vtkPolyDataNormals *normals2 = vtkPolyDataNormals::New();
normals2->SetInput((vtkPolyData *) parison->GetOutput());
normals2->SetFeatureAngle(60);
normals2->SetComputeCellNormals(0);
normals2->SetComputePointNormals(1);
normals2->SetFlipNormals(0);
normals2->SetSplitting(1);
vtkLookupTable *lut = vtkLookupTable::New();
lut->SetHueRange(0 , 0.66667);
vtkPolyDataMapper *parisonMapper = vtkPolyDataMapper::New();
parisonMapper->SetInput((vtkPolyData *) normals2->GetOutput());
parisonMapper->SetLookupTable(lut);
parisonMapper->SetScalarRange(0.12 , 1);
vtkActor *parisonActor = vtkActor::New();
```

parisonActor->SetMapper(parisonMapper);

事实上是 vtkDataSetMapper 映射器使用 vtkGeometryFilter 过滤器转换任何类型的数据集为多边形数据(这个过滤器如果输入是 vtkPolyData 数据类型则直接将其输出不进行转换)，vtkGeometryFilter 过滤器通过使用 PointClippingOn()、SetPointMinimum()、SetPointMaximum()、CellClippintOn()、SetCellMinimum() 、SetCellMaximum()这些方法基于点或单元的 ID 号来提取数据集，另外，你还可以使用矩形区域来限制所提取的数据，如：使用 ExtentClippingOn()方法可以设定裁减范围，使用 SetExtent()方法可以设定提取范围（ 范围由六个值组成， 这六个值（ Xmin,Xmax,Ymin,Ymax,Zmin,Zmax）定义了一个空间中的 bounding box)。

3.2 可视化多边形数据

多边形数据（vtkPolyData）是可视化数据的一种重要的形式，它的重要性归因于它被用作图形硬件/绘制引擎的几何接口，其它的数据类型必须被转换为多边形数据才能够被绘制（图像和体数据除外），对于如何将其它类型数据集转换成多边形数据，可以参考 3.1.12 节“提取单元作为多边形数据”相关的内容。

多边形数据（vtkPolyData）由点、线、多义线、三角形、四边形、多边形、三角带等基本单元构成，大多数过滤器（输入类型为 vtkPolyData）都可以处理这种类型的数据，然而有些过滤器（比如 vtkDecimatePro 和 vtkTubeFilter）仅仅处理这种类型数据的一部分（三角网格和线）。

3.2.1 手动创建 vtkPolyData

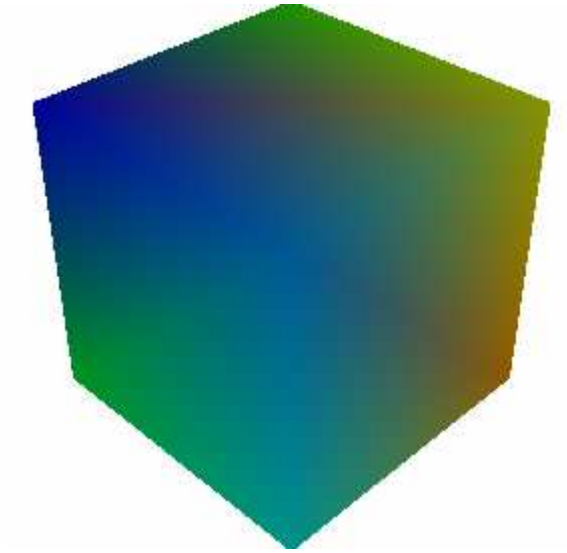
有几种不同的方式来构造多边形数据，一般的情况下首先创建一个 vtkPoints 类型的对象存储点，然后创建 vtkCellArrays 类型的对象存储单元，下面的示例程序用一个三角带创建了一个 vtkPolyData，程序运行结果如下：

程序示例代码（Examp313）：

```
//定义定点
vtkPoints *points = vtkPoints::New();
points->InsertPoint(0, 0.0, 0.0, 0.0);
points->InsertPoint(1, 0.0, 1.0, 0.0);
points->InsertPoint(2, 1.0, 0.0, 0.0);
points->InsertPoint(3, 1.0, 1.0, 0.0);
points->InsertPoint(4, 2.0, 0.0, 0.0);
points->InsertPoint(5, 2.0, 1.0, 0.0);
points->InsertPoint(6, 3.0, 0.0, 0.0);
points->InsertPoint(7, 3.0, 1.0, 0.0);
//构建组成多边形数据的单元
vtkCellArray *strips = vtkCellArray::New();
//插入一个单元，该单元由 8 个顶点组成
strips->InsertNextCell(8);
strips->InsertCellPoint(0);
strips->InsertCellPoint(1);
strips->InsertCellPoint(2);
strips->InsertCellPoint(3);
strips->InsertCellPoint(4);
strips->InsertCellPoint(5);
strips->InsertCellPoint(6);
strips->InsertCellPoint(7);
//构建多边形数据
vtkPolyData *profile = vtkPolyData::New();
//设定组成多边形数据的点
profile->SetPoints(points);
//设定单元的组成方式，三角条带
profile->SetStrips(strips);
//数据映射
vtkPolyDataMapper *map = vtkPolyDataMapper::New();
```

```
map->SetInput(profile);
//创建角色
vtkActor *strip = vtkActor::New();
strip->SetMapper(map);
```

下面给出一个创建立方体的例子，这次我们创建六个四边形并且用立方体顶点处的标量值作为颜色映色绘制立方体，示例程序运行结果如下：



示例代码（Examp314）：

```
//定义立方体的顶点坐标
static float x[8][3]={{0,0,0}, {1,0,0}, {1,1,0}, {0,1,0},
    {0,0,1}, {1,0,1}, {1,1,1}, {0,1,1}};
//定义单元，每 4 个顶点建立一个四边形单元，共计 6 个单元
static vtkIdType pts[6][4]={{0,1,2,3}, {4,5,6,7}, {0,1,5,4},
    {1,2,6,5}, {2,3,7,6}, {3,0,4,7}};
//创建对象
vtkPolyData *cube = vtkPolyData::New();
vtkPoints *points = vtkPoints::New();
vtkCellArray *polys = vtkCellArray::New();
//存储标量值
vtkFloatArray *scalars = vtkFloatArray::New();
//存储顶点
```

```
for(i=0;i<8;i++)points->InsertPoint(i,x[i]);
//设定单元
for(i=0;i<6;i++)polys->InsertNextCell(4,pts[i]);
//设定每个顶点的标量值
for(i=0;i<8;i++)scalars->InsertTuple1(i,i);
//创建多边形数据
cube->SetPoints(points);
//设定单元类型为多边形
cube->SetPolys(polys);
//设定每个顶点的标量值
cube->GetPointData()->SetScalars(scalars);
points->Delete();
polys->Delete();
scalars->Delete();
//数据映射，使用默认颜色映射表
vtkPolyDataMapper *cubeMapper = vtkPolyDataMapper::New();
cubeMapper->SetInput(cube);
cubeMapper->SetScalarRange(0,7);
vtkActor *cubeActor = vtkActor::New();
cubeActor->SetMapper(cubeMapper);
```

vtkPolyData 可以由顶点、线、多边形、三角带等基本单元任意组合构造。

3.2.2 生成表面法线

在绘制一个多边形网格的时候，可能会发现绘制的图形有许多明显的网格小平面，如下图所示，产生这种现象的原因是多边形表面的法线没有被使用，光照使得绘制的图形缺乏平滑过渡感，虽然使用 Gouraud 着色的方式可以改善图形的显示质量，然而如果对网格中的每一个点都能计算出其法线，将会产生更好的显示质量，在 VTK 中使用vtkPolyDataNormals过滤器可以计算网格上每个点的法线。

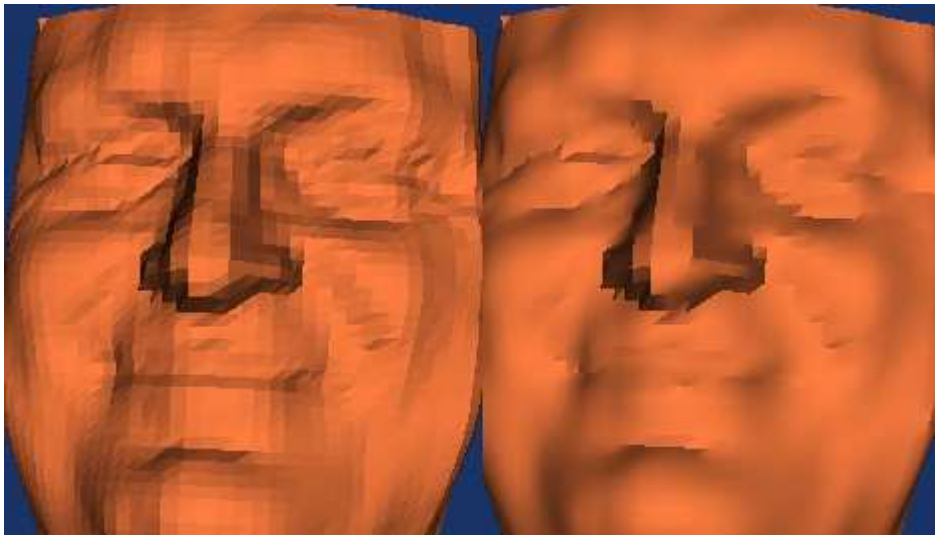


图 1 没有使用法线的效果　　图 2 使用法线后的效果

在“符号化” 和“为等值面分级着色”等章节的示例程序都使用了 vtkPolyDataNormals 过滤器，可以参考这些程序看这个过滤器是如何使用的。

vtkPolyDataNormals 过滤器两个重要的变量是 Splitting 和 FeatureAngle,如果 splitting 的值被设为 ON ，特征边（是这样的边，边的两侧多边形法线形成的角度大于等于特征角）是“split”的 ，也就是说点被沿着边复制，并且网格被分开在特征边的两侧,使得和特征边相邻的两个多边的连接处在绘制时缺少平滑感,另一个重要的变量是 FlipNormals,调用 FlipNormalsOn()会导致过滤器反转法线的方向 (构成多边形各个顶点的连接顺序也被改变)。

3.2.3 多边形消减技术

多边形数据尤其是三角片是表示图形数据常见的形式,有些过滤器输出就是用三角片表示的多边形数据类型，比如 vtkContourFilter 过滤器产生三角片多边形数据输出，通常这些过滤器输出的多边形数据由大量的三角片构成，不能够被快速的绘制和处理，尤其是对于交互的应用程序来说，对于多边形数据的快速绘制有更高的要求，Decimation 技术用来解决这个问题，Decimation 技术也被称之为多边形削减或网格简化，该技术的主要目的是减多边形数据的三角片数量，而且图形的绘制质量不能因为三角片数量的减少受到太大的影响。

VTK 支持四种 decimation 对象：

vtkDecimate

vtkDecimatePro

vtkQuadricClustering

vtkQuardricDecimation

这些对象在使用上都是非常相似的，只是它们分别适用于不同的场合，其中：

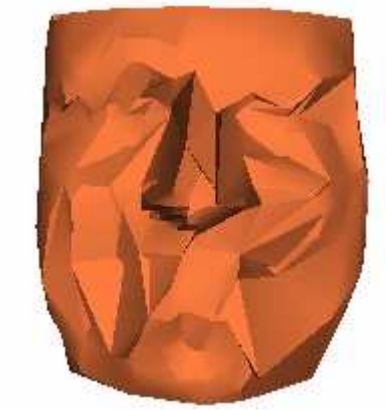
vtkDecimatePro 执行相对较快，并且在削减过程中能够修改拓扑结构，它使用边折叠处理来消除顶点和三角形，它的错误度量方法使用基于到平面或边的距离方法，vtkDecimatePro 的一个优点是你能够实现被要求任意的削减程度，因为这个算法将网格分成小片来实现（假如拓扑修改被允许）。

vtkDecimate 是取得专利权的(假如你将它用于商业用途，就得小心了)，并且它有一个小优点就是它在处理过程中使用了更好的三角化技术,它的主要特点是它属于第一种 decimation 算法，并且相对容易理解和修改。

vtkQuadricDecimation 使用二次错误度量方法，由 Garland 和 Heckbert 两人提出,它使用边折叠来消除顶点和三角形，二次错误度量通常被认为是较好的错误度量方法。

vtkQuadricClustering 是最快的算法，它能够快速的削减大量的网格，并且这个类还支持处理网格碎片的能力(使用 StartAppend()，Append()和 EndAppend()方法）,这使得用户可以避免将整个网格一次读入到内存,这个算法对于大量的网格能够较好的处理，但当网格数据量较小的时候，三角化处理过程不是很好（将这个算法和其它的算法结合起来是一个较好的方式）。

下面的示例程序说明了 vtkDecimatePro 过滤器的使用方法，程序运行结果如下图：



示例程序代码（Examp315）：

```
//读取多边形数据
vtkPolyDataReader *fran = vtkPolyDataReader::New();
fran->SetOutput(fran->GetOutput());
fran->SetFileName("fran_cut.vtk");
```

```
//消减多边形数据三角片的数量
vtkDecimatePro *deci = vtkDecimatePro::New();
deci->SetInput((vtkPolyData *) fran->GetOutput());
//设定缩减的数目
deci->SetTargetReduction(0.5);
deci->SetAccumulateError(0);
deci->SetBoundaryVertexDeletion(1);
deci->SetPreserveTopology(1);
vtkPolyDataNormals *normals = vtkPolyDataNormals::New();
normals->SetInput((vtkPolyData *) deci->GetOutput());
normals->SetFeatureAngle(30);
normals->SetComputeCellNormals(0);
normals->SetComputePointNormals(1);
normals->SetFlipNormals(1);
normals->SetSplitting(1);
vtkPolyDataMapper *franMapper = vtkPolyDataMapper::New();
franMapper->SetInput((vtkPolyData *) normals->GetOutput());
vtkActor *franActor = vtkActor::New();
franActor->SetMapper(franMapper);
```

vtkDecimatePro 过滤器有两个重要的变量是TargetReduction 和 PreserveTopology，TargetReduction 是被要求减少的数量（例如，值 0.9 意味着我们希望减少网格中 90%的三角形），根据是否允许改变拓扑结构（用 PreserveTopologyOn/Off()方法），可能实现或者也可能不能够实现所要求的削减量，如果使用 PreserveTopologyOff()方法，那么 vtkDecimatePro 过滤器将完成所要求的任意削减量（但不保证削减后的拓扑结构不改变），需要注意的是上述提到的三角形消减过滤器只接收三角片组成的多边形数据类型作为输入，如果是由多边形网格构成的多边形数据，应该使用vtkTriangleFilter将多边形转换为三角形。

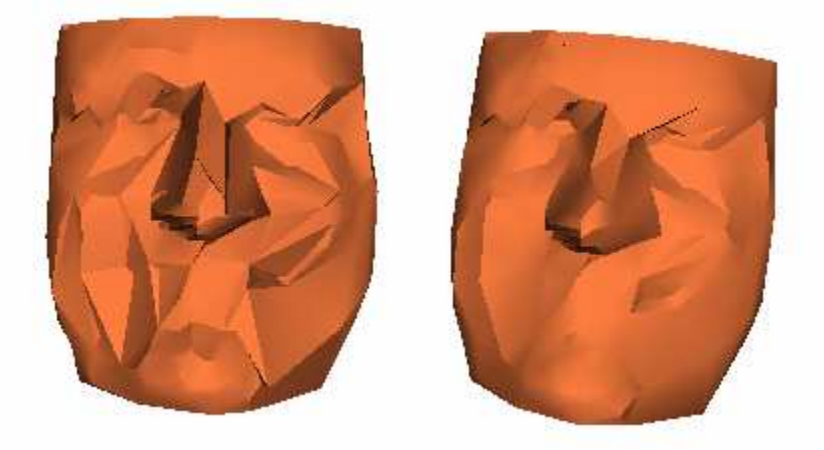
3.2.4 平滑网格 Smooth Mesh

多边形网格通常包含噪声或者光洁度太差从而对绘制图形的质量造成影响，例如，低分辨率数据生成的轮廓面会表现出锯齿效应或分级效应，解决这个问题的一种方式是使用平滑网格技术，该技术是通过调整点的位置来减少表面上的噪声的过程。

VTK 提供了两种平滑过滤器：

- vtkSmoothPolyDataFilter
- vtkWindowedSincPolyDataFilter

在这两种过滤器中 vtkWindowedSincPolyDataFilter 的平滑效果较好、平滑速度较快，下面的示例程序说明了怎样使用平滑过滤器，这个例子和上一节所给的例子是一样的，除了增加了平滑过滤器，程序运行的结果如下：



未平滑处理　　平滑处理

示例程序（Examp316）：

```
//读取多边形数据
vtkPolyDataReader *fran = vtkPolyDataReader::New();
fran->SetOutput(fran->GetOutput());
fran->SetFileName("fran_cut.vtk");
//消减多边形数据三角片的数量
vtkDecimatePro *deci = vtkDecimatePro::New();
deci->SetInput((vtkPolyData *) fran->GetOutput());
//设定缩减的数目
deci->SetTargetReduction(0.9);
deci->SetAccumulateError(0);
```

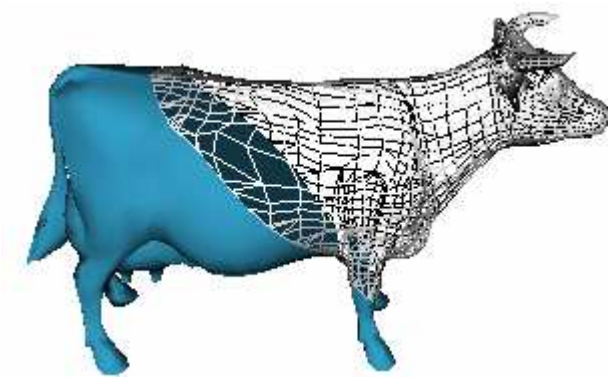
```
deci->SetBoundaryVertexDeletion(1);
deci->PreserveTopologyOff();
//对网格平滑处理
vtkSmoothPolyDataFilter *smooth=vtkSmoothPolyDataFilter::New();
smooth->SetInput(deci->GetOutput());
//设定拉普拉斯算子迭代数目控制平滑效果，数目越大，效果越好
smooth->SetNumberOfIterations(100);
vtkPolyDataNormals *normals = vtkPolyDataNormals::New();
 normals->SetInput((vtkPolyData *) smooth->GetOutput());
normals->SetFeatureAngle(30);
normals->SetComputeCellNormals(0);
normals->SetComputePointNormals(1);
normals->SetFlipNormals(1);
normals->SetSplitting(1);
vtkPolyDataMapper *franMapper = vtkPolyDataMapper::New();
franMapper->SetInput((vtkPolyData *) normals->GetOutput());
vtkActor *franActor = vtkActor::New();
franActor->SetMapper(franMapper);
```

两种平滑过滤器的使用的方式是相似的，该过滤器有一个可选的方法来控制沿特征边和边界上的平滑效果。

3.2.5 裁减数据

裁减（clipping）和切割一样使用一个隐函数来定义一个裁减面，裁减即可以将多边形网格分散成片也可以将多边形网格处于裁减面上的同一单元分成两部分，和切割一样裁减也可以设置一个裁减值来定义隐函数的值。

下面的示例程序使用一个平面来裁剪一个多边形网格数据（牛的模型数据），裁剪值被用来沿着法线方向移动裁剪平面，以便于可以在不同的位置裁剪多边形网格数据，程序的运行结果如下：



程序示例代码（Examp317）：

```
//读取模型数据
vtkBYUReader *cow = vtkBYUReader::New();
cow->SetGeometryFileName("cow.g");
vtkPolyDataNormals *cowNormals = vtkPolyDataNormals::New();
cowNormals->SetInput((vtkPolyData *) cow->GetOutput());
//定义裁剪平面，位置靠近数据集的中心，初始裁减平面
vtkPlane *plane = vtkPlane::New();
plane->SetOrigin(0.05 , 0 , 0);
plane->SetNormal(-1 , -1 , 0);
//使用隐函数裁减多边形数据
vtkClipPolyData *clipper = vtkClipPolyData::New();
//设定隐函数
clipper->SetClipFunction(plane);
clipper->SetInput((vtkPolyData *)cowNormals->GetOutput());
//将裁减平面移到新的位置
clipper->SetValue(0.5);
clipper->SetGenerateClipScalars(1);
//设定输出被裁减后剩余的部分
clipper->GenerateClippedOutputOn();
vtkPolyDataMapper *clipMapper = vtkPolyDataMapper::New();
clipMapper->SetInput((vtkPolyData *) clipper->GetOutput());
clipMapper->SetScalarVisibility(0);
vtkProperty *backProp = vtkProperty::New();
```

```
backProp->SetDiffuseColor(1, 0, 0);
vtkActor *clipActor = vtkActor::New();
clipActor->SetBackfaceProperty(backProp);
clipActor->SetMapper(clipMapper);
clipActor->GetProperty()->SetColor(0 , 1 , 0);
//切割处理,生成切割线
vtkCutter *cutEdges = vtkCutter::New();
cutEdges->SetCutFunction(plane);
cutEdges->SetInput((vtkDataSet *) cowNormals->GetOutput());
cutEdges->SetValue(0 , 0.5);
cutEdges->GenerateCutScalarsOn();
//将输入数据生成线，在此将切割的部分生成切割线
vtkStripper *cutStrips = vtkStripper::New();
cutStrips->SetInput((vtkPolyData *) cutEdges->GetOutput());
//cutStrips->Update();
vtkPolyData *cutPoly = vtkPolyData::New();
cutPoly->SetPoints((cutStrips->GetOutput())->GetPoints());
cutPoly->SetPolys((cutStrips->GetOutput())->GetLines());
vtkTriangleFilter *cutTriangles = vtkTriangleFilter::New();
cutTriangles->SetInput(cutPoly);
cutTriangles->SetPassLines(1);
cutTriangles->SetPassVerts(1);
vtkPolyDataMapper *cutMapper = vtkPolyDataMapper::New();
cutMapper->SetInput((vtkPolyData *) cutTriangles->GetOutput());
//cutMapper->SetInput(cutPoly);
vtkActor *cutActor = vtkActor::New();
cutActor->SetMapper(cutMapper);
//cutActor->GetProperty()->SetColor(0 , 0 , 0);
cutActor->GetProperty()->SetLineWidth(30);
//被裁减剩余的部分用线框绘制
```

```
vtkPolyDataMapper *restMapper = vtkPolyDataMapper::New();
 restMapper->SetInput(clipper->GetClippedOutput());
 restMapper->ScalarVisibilityOff();
vtkActor *restActor = vtkActor::New();
 restActor->SetMapper(restMapper);
```

示例程序中设定 GenerateClippedOutputOn()方法输出被裁减掉的部分，并用线框模式显示，用 SetValue()方法改变裁减值就等于修改了裁减平面的位置，使新裁减平面沿初始裁减平面（隐函数定义）向上或向下平移。

3.2.6 创建纹理坐标

在 VTK 中提供了以下几种过滤器产生纹理坐标：

vtkTextureMapToPlane

vtkTextureMapToCylinder

vtkTextureMapToSphere

这些对象分别是基于平面、圆柱和球形坐标系统来生成纹理坐标，而且类 vtkTransformTextureCoordinates 提供了平移和缩放纹理坐标的功能，把纹理贴图映射到表面上，下面的示例程序使用 vtkTextureMapToCylinder 来为产生自 vtkDelaunay3D 对象的非结构化网格创建一个纹理坐标，程序运行结果如下：

示例程序代码（Examp318）：

```
//在指定位置和半径范围内随机生成点
vtkPointSource *sphere = vtkPointSource::New();
sphere->SetCenter(0 , 0 , 0);
//生成 25 个随机点
sphere->SetNumberOfPoints(125);
```

```
//用 3D 三角网创建表面
vtkDelaunay3D *del = vtkDelaunay3D::New();
del->SetInput((vtkPointSet *) sphere->GetOutput());
del->SetTolerance(0.01);
//生成柱面纹理坐标
vtkTextureMapToCylinder *tmapper = vtkTextureMapToCylinder::New();
tmapper->SetInput((vtkDataSet *) del->GetOutput());
//控制纹理坐标的生成方式
tmapper->SetPreventSeam(0);
//对纹理坐标缩放，在每个三角面上纹理坐标的重复次数，设为 5 次
vtkTransformTextureCoords *xform = vtkTransformTextureCoords::New();
xform->SetInput(tmapper->GetUnstructuredGridOutput());
xform->SetScale(5 , 5 , 5);
vtkDataSetMapper *mapper = vtkDataSetMapper::New();
 mapper->SetInput(xform->GetUnstructuredGridOutput());
//读取纹理图片
vtkBMPReader *bmpReader = vtkBMPReader::New();
bmpReader->SetFileName("masonry.bmp");
//设定纹理
vtkTexture *atext = vtkTexture::New();
atext->SetInput((vtkImageData *) bmpReader->GetOutput());
vtkActor *triangulation = vtkActor::New();
triangulation->SetMapper(mapper);
triangulation->SetTexture(atext);
```

在这个例子中，一系列随机的点集构成小三角片，小三角片构成组成球体，每个小三角片都被分配纹理坐标，为了重复纹理，纹理坐标在 i-j 的方向上进行缩放处理，最后，读取纹理图片并把纹理图片并交由 Actor（角色）对象处理。

3.3 可视化结构网格数据集

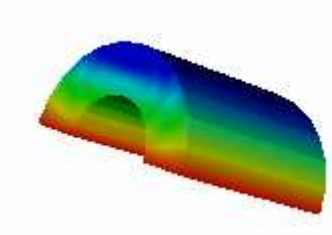
结构化网格数据集的内部拓扑结构(单元)是规则的，外部的几何形状是不规则的，结构化网格数据集经常用于数值分析（如：计算流体力学），在 VTK 中用 vtkStructuredGrid 类表示结构化网格数据集，结构化网格数据集由六面体或者四面体单元组成。

3.3.1 手动创建结构化网格数据集

结构化网格数据集的创建步骤包括：

- 定义网格的尺寸。
- 定义 vtkPoint 对象，存储网格数据点。

下面的示例程序说明了如何建一个结构化网格数据，程序运行结果如下：



程序示例代码（Examp319）：

```
//创建结构化网格数据
vtkStructuredGrid *sgrid = vtkStructuredGrid::New();
//定义网格的尺寸
sgrid->SetDimensions(dims);
//定义网格数据点及属性存储对象
vtkFloatArray *vectors = vtkFloatArray::New();
vectors->SetNumberOfComponents(1);
vectors->SetNumberOfTuples(dims[0]*dims[1]*dims[2]);
vtkPoints *points = vtkPoints::New();
points->Allocate(dims[0]*dims[1]*dims[2]);
//依据半圆柱的方式，生成点坐标
deltaZ=2.0/(dims[2]-1);
deltaRad=(rMax-rMin)/(dims[1]-1);
v[2]=0.0;
```

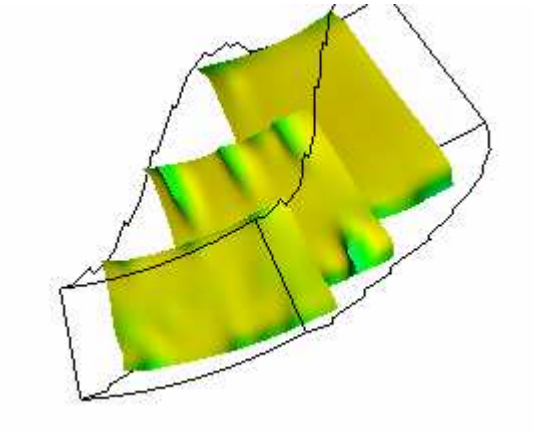
```
for(k=0; k<dims[2]; k++){
  x[2]=-1.0+k*deltaZ;
  kOffset=k*dims[0]*dims[1];
  for(j=0;j<dims[1];j++){
    radius = rMin + j*deltaRad;
    jOffset = j * dims[0];
    for(i=0;i<dims[0]; i++) {
        theta = i * 15.0 * vtkMath::DegreesToRadians();
        x[0] = radius * cos(theta);
        x[1] = radius * sin(theta);
        v[0] = -x[1];
        v[1] = x[0];
        offset = i + jOffset + kOffset;
        points->InsertPoint(offset,x);
        vectors->InsertTuple1(offset,x[1]);
      }
    }
  }
  sgrid->SetPoints(points);
  points->Delete();
  //设定每个点的标量值
  sgrid->GetPointData()->SetScalars(vectors);
  vectors->Delete();
  vtkDataSetMapper *pDataSetMapper=vtkDataSetMapper::New();
  pDataSetMapper->SetInput(sgrid);
  pDataSetMapper->SetScalarRange(0,0.8);
  vtkActor *sgridActor = vtkActor::New();
  sgridActor->SetMapper(pDataSetMapper);
```

创建结构化网格时，需要注意的是在 vtkpoints 对象里点的数量与设定的结构化网格在三个坐标方向的尺寸要相等，如结构化网格在三个坐标方向的尺寸都为 3，那么点集对象中

包含的点的数目必须为 27。

3.3.2 提取计算平面

在大部分情况下，接收 vtkDataSet 类型作为输入的过滤器也能处理结构化网格数据，VTK 提供的直接处理结构化网格数据的过滤器是 vtkStructuredGridGeometryFilter，这个过滤器用来从网格中提取部分数据，例如点，线或者多边形，在下面的示例程序中，我们读入一个结构网格数据，然后从中提取了三个平面，并用相关的矢量值将这三个平面弯曲，程序运行的结果如下：



示例程序代码如下（Examp320）：

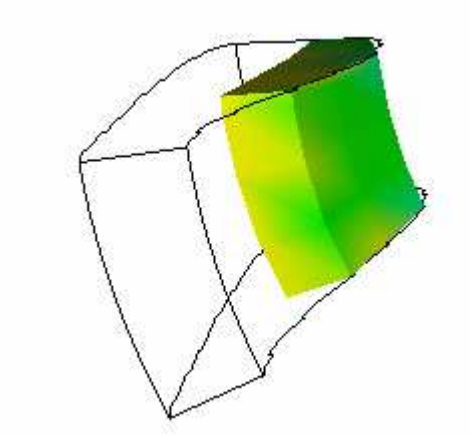
```
//读取结构化网格数据
vtkPLOT3DReader *pl3d=vtkPLOT3DReader::New();
pl3d->SetXYZFileName("combxyz.bin");
pl3d->SetQFileName("combq.bin");
//定义过滤器提取几何体
vtkStructuredGridGeometryFilter *plane=vtkStructuredGridGeometryFilter::New();
plane->SetInputConnection(pl3d->GetOutputPort());
//定义提取的几何体，本程序为提取平面，平面垂直于 x 轴
plane->SetExtent(10, 10, 1, 100, 1, 100);
//定义第 2 到第 3 个提取平面
vtkStructuredGridGeometryFilter *plane2=vtkStructuredGridGeometryFilter::New();
plane2->SetInputConnection(pl3d->GetOutputPort());
plane2->SetExtent(30, 30, 1, 100, 1, 100);
```

```
vtkStructuredGridGeometryFilter *plane3=vtkStructuredGridGeometryFilter::New();
    plane3->SetInputConnection(pl3d->GetOutputPort());
    plane3->SetExtent(45, 45, 1, 100, 1, 100);
    //把三个平面合并到一起
    vtkAppendPolyData *appendF = vtkAppendPolyData::New();
    appendF->AddInputConnection(plane->GetOutputPort());
    appendF->AddInputConnection(plane2->GetOutputPort());
    appendF->AddInputConnection(plane3->GetOutputPort());
    //变形平面
    vtkWarpScalar *warp = vtkWarpScalar::New();
    warp->SetInputConnection(appendF->GetOutputPort());
    warp->UseNormalOn();
    warp->SetNormal(1.0, 0.0, 0.0);
    warp->SetScaleFactor(2.5);
    //计算平面的法向量
    vtkPolyDataNormals *normals = vtkPolyDataNormals::New();
    normals->SetInputConnection(warp->GetOutputPort());
    normals->SetFeatureAngle(60);
    //数据映射
    vtkPolyDataMapper *planeMapper = vtkPolyDataMapper::New();
    planeMapper->SetInputConnection(normals->GetOutputPort());
    planeMapper->SetScalarRange((pl3d->GetOutput())->GetScalarRange());
    vtkActor *planeActor = vtkActor::New();
    planeActor->SetMapper(planeMapper);
```

示例程序通过使用 SetExtent（）方法中参数确定提取几何体的类型，如果提取的是点、线或面，使用 vtkPolyDataMapper 映射器进行数据映射，如果提取的是结构化网格数据，则使用 vtkDataSetMapper 映射器进行映射。

3.3.3 对结构化网格数据二次采样

结构化网格数据的采样过程和结构化点集的采样过程相类似，在 VTK 中用 vtkExtractGrid 过滤器用来执行此项工作，下面给出示例程序，程序运行结果如下：



示例程序代码（Examp321）：

```
//读取结构化网格数据
vtkPLOT3DReader *pl3d=vtkPLOT3DReader::New();
pl3d->SetXYZFileName("combxyz.bin");
pl3d->SetQFileName("combq.bin");
//对读取的结构化网格数据采样，提取子集
vtkExtractGrid *extract= vtkExtractGrid::New();
extract->SetInput(pl3d->GetOutput());
//设置提取的区域范围
extract->SetVOI(-30,30,-1000,1000,-10,10);
//设置在 x,y,z 三个方向采样的间隔点数
extract->SetSampleRate(1,2,3);
extract->IncludeBoundaryOn();
//数据映射
vtkDataSetMapper *DataSetMapper = vtkDataSetMapper::New();
DataSetMapper->SetInput(extract->GetOutput());
DataSetMapper->SetScalarRange((pl3d->GetOutput())->GetScalarRange());
vtkActor *DataSetActor = vtkActor::New();
DataSetActor->SetMapper(DataSetMapper);
```

在这个示例程序中读取的结构化网格数据集的尺寸为 57*33*25，在三个方向的采样率为(1,2,3)，经过采样后，生成一个在三个方向尺寸为 29*17*9 的结构化网格数据，IncludeBoundaryOn（）方法确保边界的数据能被采样。

3.4 可视化线性网格数据

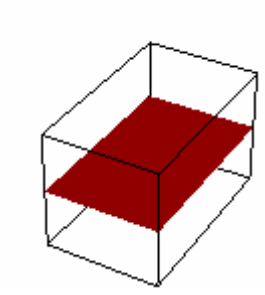
线性网格数据集的内部拓扑结构(单元)是规则的，外部的几何形状具有半规则性（如图 1-2b 所示），线性网格数据一般用于数值分析中，线形网格数据集类型的单元由像素单元或者体素单元构成。

3.4.1 手动创建线性网格数据

线性网格数据集的创建步骤包括：

- 定义网格的尺寸。
- 定义 vtkFloatArray 对象，存储网格数据点。

需要注意的是，要确保标量数组中标量数据的数目要等于网格在三个方向尺寸点的总数，下面的示例程序说明如何创建线性网格数据集，程序的运行结果如下：



示例程序代码（Examp322）：

```
//存储坐标点
vtkFloatArray *xCoords = vtkFloatArray::New();
for (i=0; i<47; i++) xCoords->InsertNextValue(x[i]);
vtkFloatArray *yCoords = vtkFloatArray::New();
for (i=0; i<33; i++) yCoords->InsertNextValue(y[i]);
vtkFloatArray *zCoords = vtkFloatArray::New();
for (i=0; i<44; i++) zCoords->InsertNextValue(z[i]);
//定义线性网格
```

```
vtkRectilinearGrid *rgrid = vtkRectilinearGrid::New();
//设定网格的尺寸
rgrid->SetDimensions(47,33,44);
//设定坐标
 rgrid->SetXCoordinates(xCoords);
rgrid->SetYCoordinates(yCoords);
rgrid->SetZCoordinates(zCoords);
//提取一个平面
vtkRectilinearGridGeometryFilter\                                    、
     *plane=vtkRectilinearGridGeometryFilter::New();
plane->SetInput(rgrid);
plane->SetExtent(0,46, 16,16, 0,43);
vtkPolyDataMapper *rgridMapper = vtkPolyDataMapper::New();
rgridMapper->SetInputConnection(plane->GetOutputPort());
vtkActor *wireActor = vtkActor::New();
wireActor->SetMapper(rgridMapper);
//wireActor->GetProperty()->SetRepresentationToWireframe();
wireActor->GetProperty()->SetColor(1,0,0);
//提取线性网格的外轮廓
vtkOutlineFilter *pGridOut=vtkOutlineFilter::New();
pGridOut->SetInput(rgrid);
vtkPolyDataMapper *outMapper = vtkPolyDataMapper::New();
outMapper->SetInputConnection(pGridOut->GetOutputPort());
vtkActor *outActor = vtkActor::New();
outActor->SetMapper(outMapper);
```

3.4.2 提取计算平面

接收 vtkDataSet 类型作为输入的过滤器也能处理线性网格数据， VTK 提供的能直接处理线性网格数据的过滤器是 vtkRectilinearGridGeometryFilter， 该过滤器用来从网格中

提取部分数据，例如点、线或者多边形面，有关该过滤器的用法请参考示例程序 Examp322。

3.5 可视化非结构网格数据

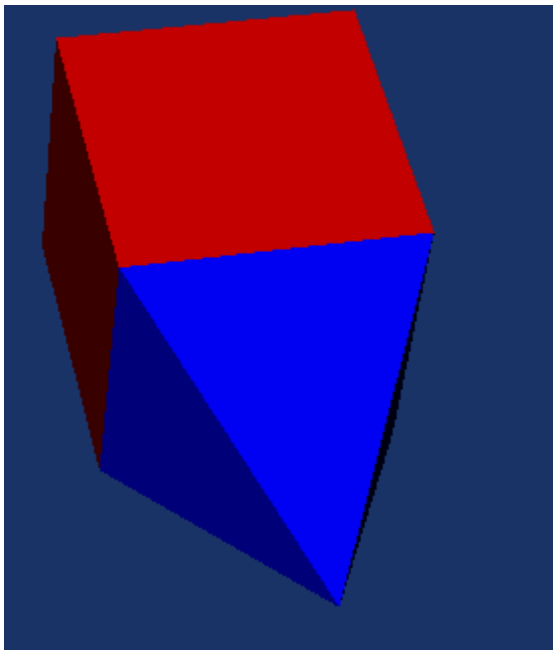
非结构网格数据集无论在拓扑结构还是在外部几何形状上都是无规则的，非结构化网格数据经常用于数值分析（例如有限元分析），几乎所有的网格类型都可以被非结构化网格所表示。

3.5.1 手动创建非结构化网格数据

非结构化网格数据集的创建包括：

- 通过 vtkPoint 对象储存数据点
- 定义数据集的组成单元
- 将单元添加到非结构化数据集中

下面的示例程序说明了非结构化数据集的创建方法，程序的运行结果如下：



示例程序代码（Examp323）：

```
//定义体元数据点
vtkPoints *voxelPoints = vtkPoints::New();
voxelPoints->SetNumberOfPoints(8);
voxelPoints->InsertPoint(0, 0, 0, 0);
voxelPoints->InsertPoint(1, 1, 0, 0);
```

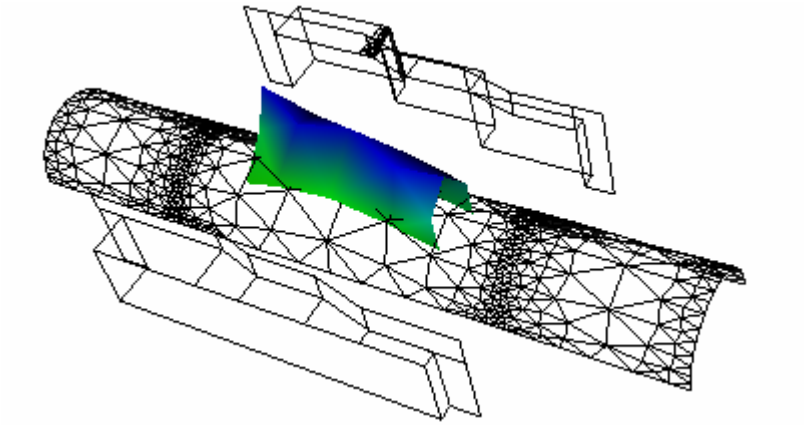
```
voxelPoints->InsertPoint(2, 0, 1, 0);
voxelPoints->InsertPoint(3, 1, 1, 0);
voxelPoints->InsertPoint(4, 0, 0, 1);
voxelPoints->InsertPoint(5, 1, 0, 1);
voxelPoints->InsertPoint(6, 0, 1, 1);
voxelPoints->InsertPoint(7, 1, 1, 1);
voxelPoints->InsertPoint(8, 0.5, 0.5, 2);
//定义体元单元
vtkVoxel *aVoxel = vtkVoxel::New();
aVoxel->GetPointIds()->SetId(0,0);
aVoxel->GetPointIds()->SetId(1,1);
aVoxel->GetPointIds()->SetId(2,2);
aVoxel->GetPointIds()->SetId(3,3);
aVoxel->GetPointIds()->SetId(4,4);
aVoxel->GetPointIds()->SetId(5,5);
aVoxel->GetPointIds()->SetId(6,6);
aVoxel->GetPointIds()->SetId(7,7);
//定义金字塔单元
vtkPyramid *aPyramid=vtkPyramid::New();
aPyramid->GetPointIds()->SetId(0,4);
aPyramid->GetPointIds()->SetId(1,5);
aPyramid->GetPointIds()->SetId(2,6);
aPyramid->GetPointIds()->SetId(3,7);
aPyramid->GetPointIds()->SetId(4,8);
//存储标量值
vtkFloatArray *scalars = vtkFloatArray::New();
for(int i=0;i<2;i++)scalars->InsertTuple1(i,i);
//定义非结构化数据集
vtkUnstructuredGrid *aVoxelGrid = vtkUnstructuredGrid::New();
aVoxelGrid->Allocate();
```

```
aVoxelGrid->SetPoints(voxelPoints);
//插入单元
aVoxelGrid->InsertNextCell(aVoxel->GetCellType(),aVoxel->GetPointIds());
aVoxelGrid->InsertNextCell(aPyramid->GetCellType(),aPyramid->GetPointIds());
//设定单元的标量值，2 个单元
aVoxelGrid->GetCellData()->SetScalars(scalars);
vtkDataSetMapper *aVoxelMapper = vtkDataSetMapper::New();
aVoxelMapper->SetInput(aVoxelGrid);
aVoxelMapper->SetScalarRange(0 , 1);
vtkActor *aVoxelActor = vtkActor::New();
aVoxelActor->SetMapper(aVoxelMapper);
```

该示例程序创建了由两个不同单元构成的非结构化网格数据集，当把单元插入到 vtkUnstructuredGrid 中前，必须先调用 Allocate()方法分配足够的内存空间存储单元。

3.5.2 提取部分网格数据

接收 vtkDataSet 类型作为输入的过滤器也能处理非结构化网格数据， VTK 提供的能直接处理非结构化网格数据的过滤器是 vtkExtractUnstructuredGrid，该过滤器用于提取一定范围内的部分网格，下面的示例程序给出了该过滤器的用法，程序运行结果如下：



示例程序代码（Examp324）：

```
//读取非结构化数据网格
vtkUnstructuredGridReader *reader = vtkUnstructuredGridReader::New();
reader->SetFileName("blow.vtk");
```

```
//设定被提取的标量数据名称，在 vtk 数据文件中
reader->SetScalarsName("thickness9");
reader->SetVectorsName("displacement9");
//对数据类型进行检查
vtkCastToConcrete *castToUnstructuredGrid=vtkCastToConcrete::New();
castToUnstructuredGrid->SetInput(reader->GetOutput());
//显示非结构化网格数据
vtkDataSetMapper *moldMapper = vtkDataSetMapper::New();
moldMapper->SetInput(castToUnstructuredGrid->GetOutput());
vtkActor *moldActor = vtkActor::New();
moldActor->SetMapper(moldMapper);
moldActor->GetProperty()->SetColor(0.2 , 0.2 , 0.2);
moldActor->GetProperty()->SetRepresentationToWireframe();
//用矢量数据对几何体变形，点在法向量方向按位移系数产生位移
vtkWarpVector *warp = vtkWarpVector::New();
//将单元数据以非结构化网格的方式输出
warp->SetInput(castToUnstructuredGrid->GetUnstructuredGridOutput());
//设定位移系数
warp->SetScaleFactor(0.2);
//根据几何的连通性提取数据
vtkConnectivityFilter *connect2 = vtkConnectivityFilter::New();
connect2->SetInput((vtkDataSet *) warp->GetOutput());
//提取非结构化数据集中的数据
vtkExtractUnstructuredGrid *extractGrid=vtkExtractUnstructuredGrid::New();
extractGrid->SetInputConnection(connect2->GetOutputPort());
extractGrid->CellClippingOn();
//设定提取单元的最大、最小值范围
extractGrid->SetCellMinimum(0);
extractGrid->SetCellMaximum(123);
vtkGeometryFilter *parison = vtkGeometryFilter::New();
```

```
parison->SetInput((vtkDataSet *)extractGrid->GetOutput());
```

3.5.3 对 vtkUnstructuredGrid 提取轮廓值

VTK 在内部专门提供了 vtkContourGrid 过滤器用于在非结构化数据集中提取轮廓线或轮廓面， vtkContourGrid 过滤器比 vtkContourFilter 过滤器有着更好的性能，由于这个过滤器是一个内部过滤器，所以不能直接使用，当使用 vtkContourFilter 过滤器提取轮廓线或轮廓面时，过滤器判断输入的数据集是否是非结构化数据集，如果是非结构化数据集，过滤器会自动创建一个 vtkContourGrid 过滤器的实例，使用 vtkContourGrid 过滤器处理非结构化数据集。

4 可视化图像及三维体数据

Image（图像）数据具有规则的拓扑结构和几何形状，在 VTK 中用类 vtkImageData 表示这种类型的数据，由于这种数据的结构是规则的，所以数据集中各个数据点的位置由 origin(起始位置),spacing(空间间距),dimensions(三维范围)这几个参数就可以确定，医学和科学扫描设备（CT，MRI，超声波扫描仪，以及共焦显微镜等）常产生这种形式的数据，vtkImageData 数据集由体元（vtkVoxel ）或者像元（vtkPixel）单元组成，由于这种数据集的结构是规则的，所以我们用一个简单的数组就可以存储数据值，而不用创造 vtkVoxel 或 vtkPixel 单元，这种数据结构如图 4-1 所示：

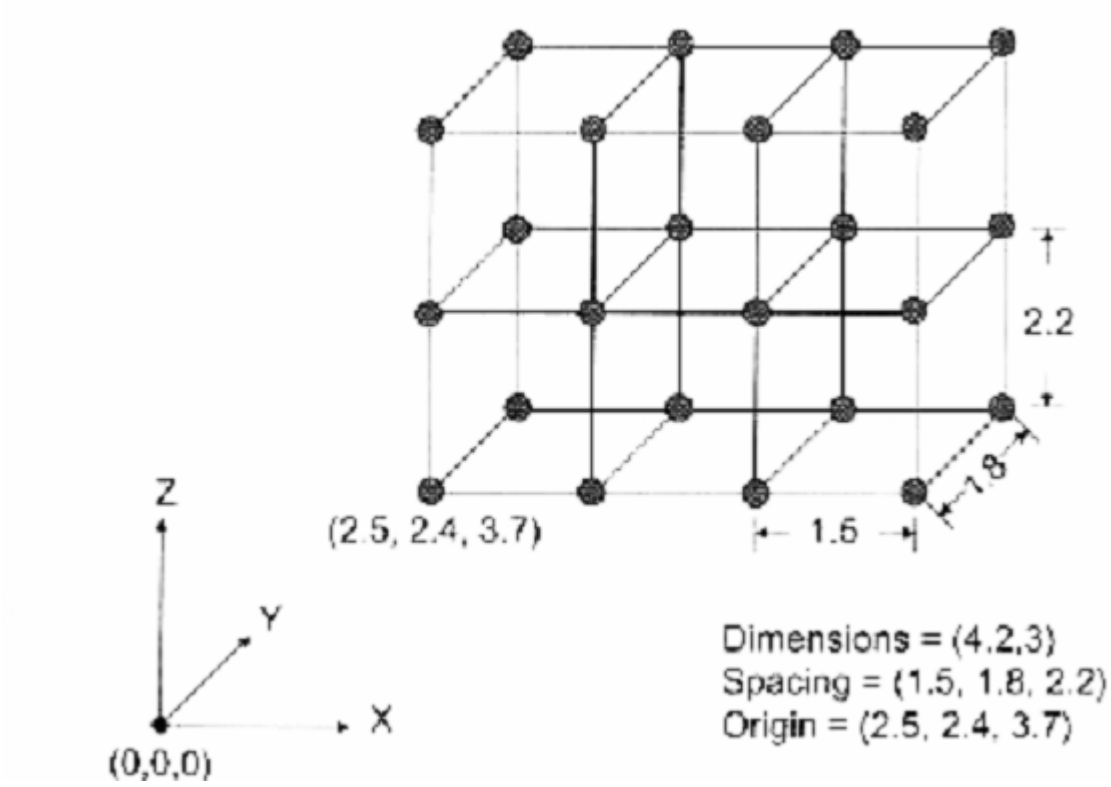


图 4-1　体数据单元的构成

在 VTK 中图像数据是一种特殊的数据类型，所以有多种不同的处理和显示方式，对图像数据的处理主要包括：

- 图像处理
- 多边形提取
- 直接绘制

在 VTK 中有许多能够处理图像数据集的过滤器，这些过滤器把 vtkImageData 作为输入，产生 vtkImageData 作为输出，还有能够把 vtkImageData 转换为 vtkPolyData 类型的过滤器，例如，vtkContoutFilter 过滤器能够从图像数据集里以三角形网的形式提取等值线，最后还有一些用于绘制 vtkImageData 数据的映射器和特定的 Actors（角色）对象，这些对象可以完成简单的 2D 图像绘制、体绘制。

在这一节中，我们将介绍几种重要的图像处理技术，包括图像显示、图像处理、多边形提取以及体绘制。

4.1　vtkStructuredPoints 数据的发展历史

结构化点集是 VTK 中表示图像数据的唯一方式，通常 VTK 读入一幅图像时，首先会将图像文件转换为结构化点集的数据格式，然后再对这种结构化点集数据进行处理，随着 VTK

对图像数据处理的逐步发展和完善，在以后的 VTK 版本中发布了 vtkImageData 类，在这个类比 vtkStructuredPoints 更灵活，它允许并行处理数据流，现在 vtkStructuredPoints 类已经逐渐被淘汰掉了。

4.2 手动创建 vtkImageData 数据

创建 vtkImageData 类型的数据集,我们要对设定如下数据：

- origin(起始位置)
- spacing(空间间距，表示体素与其相邻体素之间的长宽高这三个方向上的间距)
- dimensions(三维尺度，表示在三个主轴方向体元或像元的数目)

在下面给出的创建 vtkImageData 数据集代码中,定义了一个无符号字符数组存储数据，可以使用 VTK 过滤器和绘制操作用存储的数据创建 vtkImageData 数据集，下面给出创建 vtkImageData 数据集的步骤：

1. 要做的第一件事是创建一个用来储存数据的无符号数组，我们使用方法 SetVoidArray()指定指向数据的指针,函数的最后一个参数指出 VTK 不应当释放这个内存,data 为指向存储数据内存区的指针，数组 size[3]存储数据集在 x,y,z 三个方向上的数据数量。

   ```
   vtkUnsignedCharArray *array= vtkUnsignedCharArray::New();
   array->SetVoidArray(data,size[0]*size[1]*size[2],1);
   ```

2. 第二步是创建图像数据，我们必须确保所有值必须匹配，图像数据的标量类型必须是无符号类型，而且图像数据的维数必须与数据点数的数量值相等（如对于三个方向维数全部为 3 的图像数据来说，其数据点数的数据值必须为 27)。

   ```
   imageData=vtkImageData::New();
   //设定标量值
   imageData->GetPointData()->SetScalars(arry);
   //设定三维大小
   imageData->SetSimensions(size);
   imageData->SetScalarType（VTK_UNSIGNED_CHAR）;
   imageData->SetSpacing(1.0,1.0,1.0);
   imageData->SetOrigin(0.0,0.0,0.0);
   ```

在下面的例子中，我们将使用 C++来创建图像数据，使用 vtkImageData 对象为我们创建标量数据(而不是像上述步骤那样手工创建标量数据数组并且把它们和图像数据关联在一起)，使用这种方法将会消除标量的大小和图像数据的维数不匹配的可能性。

```
//创建图像数据
vtkImageData *id=vtkImageData::New();
id->SetSimensions(10,25,100);
id->SetScalarTypeToUnsignedShort();
id->SetNumberOfScalarComponents(1);
id->AllocateScalars();
//设定标量值
Unsigned short *ptr=(unsigned short *)id->GetScalarPointer();
For(int i=0;i<10*25*100;i++){
   *ptr++=i;
}
```

在这个例子中，方法 AllocateScalars（）被用来为图像数据分配存储空间，需要注意的是：这个方法必须在标量类型和标量组分(列数)被设定后才能调用，然后调用 GetScalarPointer（）方法，该方法实际上返回结果的类型为无符号短整数。

以下程序代码为读取 BMP 文件系列,并对其数据进行设定::

```
vtkBMPReader *reader = vtkBMPReader::New();
reader->SetDataExtent(0,512,0,512,1,57);
reader->SetFilePrefix("E://CT/headbmpskin/CThead");
reader->FileLowerLeftOn();
reader->SetDataSpacing (1, 1, 6.25);//像素间的间隔
reader->SetDataOrigin(0,0,0);
```

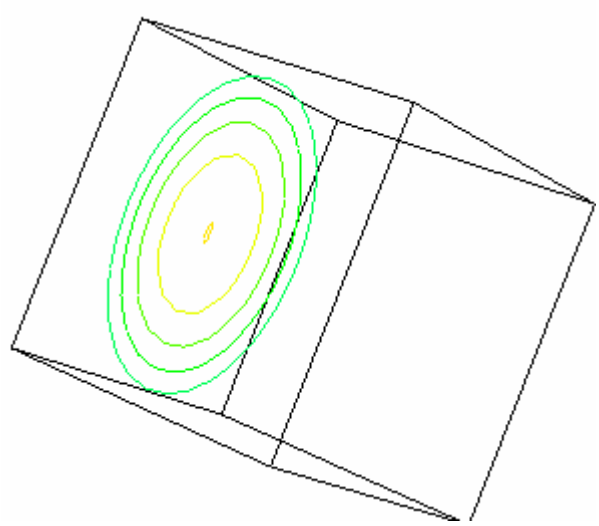
4.3 图像数据的二次采样

vtkExtractVOI 过滤器能够对输入图像数据集的部分数据进行提取并进行二次采样，该过滤器的输出是 vtkImageData 类型。

在 VTK 中有两个过滤器 vtkExtractVOI 和 vtkImageClip 都可以对图像数据进行剪切，最初VTK中图像流水线和图形流水线是相互分隔的,导致vtkImageClip过滤器仅仅只能在图

像流水线中剪切 vtkImageData 数据对象,而 vtkExtractVOI 则只能在图形流水线中剪切 vtkStructuredPoints 数据对象，目前虽然两种流水线已合并到了一起，但是这两个过滤器之间还是有一些不同之处，所以一直还在使用，vtkExtractVOI 过滤器提取一个体的子区域并且产生一个包含准确信息的 vtkImageData 数据对象,另外 vtkExtractVOI 能被用来在 VOI 中重新采样，而 vtkImageClip 过滤器默认方式是将输入的数据无变化的输出（除了一些扩展信息除外），当区域被复制到输出为 vtkImageData 数据对象时，可以在 vtkImageClip 过滤器中设置一个标志强制它产生精确的数据量， vtkImageClip 过滤器不能重新对体数据进行采样。

下面的示例程序说明了怎样使用 vtkExtractVOI 过滤器,这个示例程序提取了体数据的一部分,然后对它重新采样，输出则传给 vtkContourFilter 过滤器(你可以试着移除 vtkExtractVOI 并且比较它们的结果)，程序运行结果如下：



示例程序代码如下（Examp41）：

```
//定义二次曲面
vtkQuadric *quadric = vtkQuadric::New();
quadric->SetCoefficients(0.5 , 1 , 0.2 , 0 , 0.1 , 0 , 0 , 0.2 , 0 , 0);
//对二次曲面进行采样，生成体数据
vtkSampleFunction *sample = vtkSampleFunction::New();
sample->SetImplicitFunction(quadric);
sample->SetCapValue(1e+038);
sample->SetModelBounds(-1 , 1 , -1 , 1 , -1 , 1);
//设定体数据的三维大小
```

```
sample->SetSampleDimensions(30 , 30 , 30);
sample->SetOutputScalarType(10);
sample->SetCapping(0);
sample->SetComputeNormals(0);
//提取感兴趣区域
vtkExtractVOI *extract = vtkExtractVOI::New();
extract->SetInput((vtkImageData *) sample->GetOutput());
extract->SetSampleRate(1 , 2 , 3);
extract->SetVOI(0 , 29 , 0 , 29 , 0 , 0);
vtkContourFilter *contours = vtkContourFilter::New();
contours->SetInput((vtkDataSet *) extract->GetOutput());
contours->SetValue(0 , 0);
contours->SetValue(1 , 0.1);
   …
vtkPolyDataMapper *contMapper = vtkPolyDataMapper::New();
contMapper->SetInput((vtkPolyData *) contours->GetOutput());
vtkActor *contActor = vtkActor::New();
contActor->SetMapper(contMapper);
```

注意：示例程序从原始数据里通过指定感兴趣区域 (0,29,0,29,15,15)（在三个坐标轴上的最大最小值）来提取一个平面,而且在每个坐标轴上的采样率不同，通过更改感兴趣区域（VOI）的值，也可以提取一个子体、一条线或一个点。

对图像数据二次采样其实就是从图像中选取部分数据进行处理，有下述两种处理方式：

- 图像读取的类中有专门的方法设置要读取的数据范围,如:

```
vtkBMPReader *reader = vtkBMPReader::New();
reader->SetDataVOI(200,400,200,400,1,1);
```

- 对所读入的图像数据设定提取的数据范围，其使用方式如下:

```
vtkExtractVOI *voi=vtkExtractVOI::New();
voi->SetInputConnection(readerImageCast->GetOutputPort());
voi->SetVOI(200,400,200,400,5,25);//确定数据范围
voi-> SetSampleRate( 1, 2, 3);
```

4.4 二维图像的三维显示

有些图像存储的数据是范围值和高度值，这些图像被称为区域图或者高度，图像中的每一个像素的标量值代表一个高度或者一个范围值，对这些图像的高度或范围数据进行处理，可以生成三维的高度或范围图形，如图 4-2 所示，左图为原始图像，而右图为可视化处理后产生 3D 的表面图形。

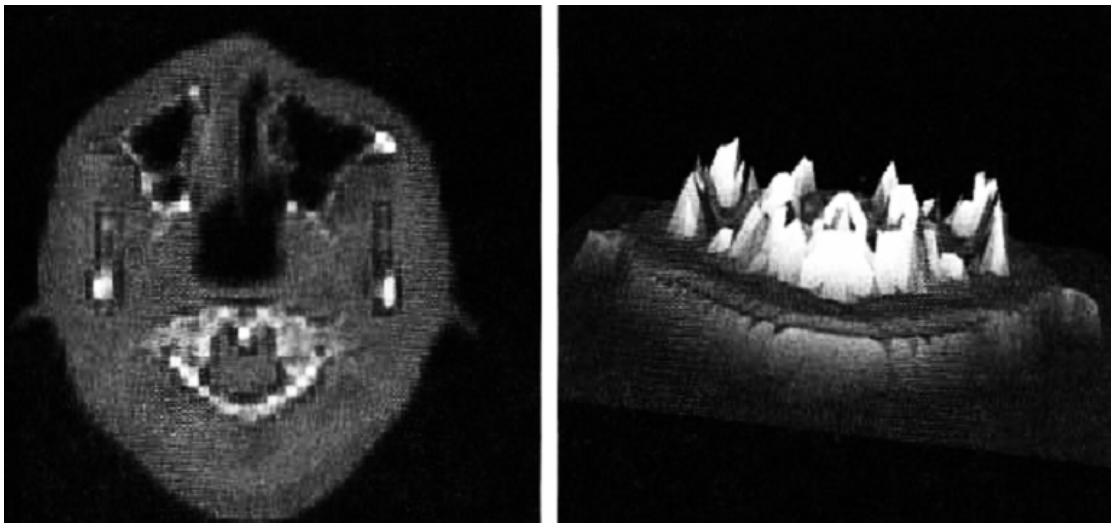
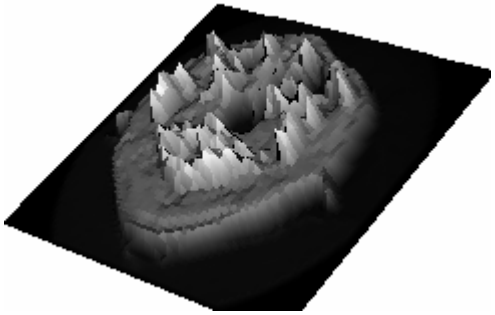


图 4-2 高度图像可视化处理

在 VTK 中用可视化流水线对高度图进行可视化处理相当简单，原始的图像数据有明确几何形状和拓扑结构，对图像进行处理后会导致 3D 表面的几何体形状发生改变，为了支持这种改变，首先使用 vtkImageDataGeometryFilter 过滤器将图像数据转换为 vtkPloyData 对象类型，然后对图像进行处理，并将处理的结果输出到一个映射器中，下面的示例程序说明了 vtkImageDataGeometryFilter 过滤器的使用方式，并利用 vtkWindowLevelLookupTable 来提供一个灰度标尺查询表来替代默认的红蓝查询表，程序的运行结果如下：



示例程序代码如下（Examp42）：

```
vtkImageReader *reader=vtkImageReader::New();
```

```
reader->SetDataByteOrderToLittleEndian();
reader->SetDataExtent(0,63,0,63,40,40);
reader->SetFilePrefix("headsq\\quarter");
reader->SetDataMask(0x7fff);
vtkImageDataGeometryFilter *filter=vtkImageDataGeometryFilter::New();
filter->SetInput(reader->GetOutput());
vtkWarpScalar *warp=vtkWarpScalar::New();
warp->SetInput(filter->GetOutput());
warp->SetScaleFactor(0.005);
//生成颜色表
vtkWindowLevelLookupTable *wl=vtkWindowLevelLookupTable::New();
//wl->SetMinimumTableValue(15,0,0);
//wl->SetNumberOfTableValues(5);
 vtkPolyDataMapper *mapper=vtkPolyDataMapper::New();
mapper->SetInput(warp->GetPolyDataOutput());
mapper->SetScalarRange(0,2000);
mapper->SetLookupTable(wl);
vtkActor *pActor=vtkActor::New();
pActor->SetMapper(mapper);
```

在这个例子中，使用了多种可视化技术，如果想给图像的不同高度值绘制不同的颜色，用 vtkMergeFilter 过滤器，如果想减少 3D 表面多边形的数量，可以通过使用 vtkDecimatePro 来减少它们的数量，同时你应当考虑使用 vtkTriangleFilter 来将多面体转换为三角形，它们往往执行的更快且占用较少的存储空间。

4.5 体绘制

体绘制是一个术语，用于描述 3D 数据的绘制过程，3D 数据信息存在于整个 3D 空间而不是简单的 2D 表面，在体绘制和几何绘制技术之间并没有一个明确的划分，两种不同的绘制方式可以产生相似的结果，在某些情况下，某些绘制方式可以被认为是体绘制和几何绘制的结合，例如，你可以使用等值线技术在结构化的点集中抽取三角形表示一个等值面，并且

使用几何绘制技术来显示这些三角形，或者你可以在图像数据集上使用光线投射技术，这两种不同的方式产生相似的结果，另一个例子是应用纹理映射硬件技术来执行合成体绘制，因为它作用于图像数据，这种方法可以被认为是体绘制技术，又因为它使用几何元素和标准的图形硬件，所以也可以被认为是几何绘制技术。

在 VTK 中已经提供了体绘制技术，vtkVolumeMapper 映射器的 SetInput()方法接收 vtkImageData 类型的数据对象作为输入，非结构化的数据集不能被直接地进行体绘制，然而可以重新对非结构化的网格进行采样，输出规则的图像数据格式以便于利用体绘制技术进行绘制。

在 VTK 中实现了三种主要的体绘制技术：

- 光线投射法
- 2D 纹理映射法
- 基于硬件的 VolumePro 体绘制

我们将从一个简单的示例程序开始介绍体绘制的内容，这个示例程序使用了三种不同的体绘制技术，接下来更详细地讨论这三种技术，然后分别对这三种技术的执行效率进行讨论，最后是对每一种体绘制方法的特点进行总结。

4.5.1 一个简单的例子

下面给出一个简单的体绘制示例程序，这个示例程序使用光线投射技术进行体绘制，但程序中仅被加黑显示的部分代码是这种体绘制技术特有的，学习这个例子后，你将发现只要改变用黑体显示的部分代码就可以实现其它的体绘制技术，比如 2D 纹理映射法或基于硬件的 VolumePro 体绘制，示例程序运行的结果如下：

示例程序代码（Examp43）：

```
//读取结构化数据集
```

```
vtkStructuredPointsReader *reader = vtkStructuredPointsReader::New();
reader->SetFileName("ironProt.vtk");
//设定体数据中不同标量值的透明度
vtkPiecewiseFunction *opacityTransferFunction = vtkPiecewiseFunction::New();
//标量值为 20 的点透明
opacityTransferFunction->AddPoint(20,0.0);
opacityTransferFunction->AddPoint(255,1.0);
//设定不同标量值的颜色
vtkColorTransferFunction *colorTransferFunction=
                              vtkColorTransferFunction::New();
colorTransferFunction->AddRGBPoint(0,0,0,0);
colorTransferFunction->AddRGBPoint(64,1.0,0,0);
colorTransferFunction->AddRGBPoint(128,0.0,0,1.0);
colorTransferFunction->AddRGBPoint(192,0.0,1.0,0.0);
colorTransferFunction->AddRGBPoint(255,0.0,0.2,0.0);
//设定体数据的属性
vtkVolumeProperty *volumeProperty = vtkVolumeProperty::New();
volumeProperty->SetColor(colorTransferFunction);
volumeProperty->SetScalarOpacity(opacityTransferFunction);
//显示阴影
volumeProperty->ShadeOn();
//设定插值类型为线性插值
volumeProperty->SetInterpolationTypeToLinear();
vtkVolumeRayCastCompositeFunction *compositeFunction=
          vtkVolumeRayCastCompositeFunction::New();
//设定体数据映射器
vtkVolumeRayCastMapper *volumeMapper = vtkVolumeRayCastMapper::New();
volumeMapper->SetInput((vtkImageData *) reader->GetOutput());
volumeMapper->SetVolumeRayCastFunction(compositeFunction);
vtkVolume *volume = vtkVolume::New();
```

```
volume->SetMapper(volumeMapper);
volume->SetProperty(volumeProperty);
```

这个示例程序从磁盘读取结构化数据，然后设定标量值的不透明性和颜色值，使用 vtkVolumeProperty 对象的方法按设定的不透明性和颜色值映射标量值，接下来创建特定的体射线投射对象 vtkVolumeRayCastCompositeFunction 对象，它执行沿着射线方向的采样合成，vtkVolulmeRayCastMapper 映射器执行一些基本的射线投射操作比如变换和裁剪，设置映射器的输入数据为读取的结构化数据，并创建一个 vtkVolume（vtkProp3D 的子类，类似于 vtkActor）对象来处理被映射的体数据和体属性数据，最后创建标准的图形显示对象对体数据进行绘制。

如果想用 2D 纹理映射法代替光线投射法，那么上述代码的黑体部分应该被替换为：

```
vtkVolumeTextureMapper2D *volMap= vtkVolumeTextureMapper2D::New();
```

如果系统的硬件支持 VolumePro 体绘制，可以使用下面的代码来替换黑体部分的代码，实现基于硬件的 VolumePro 体绘制方法。

```
vtkVolumeProMapper *volMap= vtkVolumeProMapper::New();
```

4.5.2 为什么有多种体绘制技术

正如上个示例程序所示，在不同的体绘制方法之间进行转换所做的工作就是改变体映射器的类型，这可能会产生下面的一些疑问：为什么 VTK 中要有不同的体绘制方法？为什么 VTK 不能简单的提供一种工作最好的体绘制方法？，下面将要详细讨论不同的体绘制方法之间的优缺点。

首先，上述三种体绘制方法都有不同的优点，由于受到体数据量的大小、图形硬件设备、处理器等条件的影响，选择哪种体绘制方法要根据实际的应用选取，所以并不能说哪一种体绘制方法是最好的。

第二由于计算的复杂性，大多数的体绘制方法仅仅产生期望结果的近似值，例如，对体数据进行采样和使用 alpha 混合函数进行合成绘制的体数据，仅仅是对实际情况的近似表达，在不同的情况下不同的体绘制方法在质量和速度上是不同的。

另外有些体绘制方法仅在特定的条件下使用，例如，基于硬件的 VolumePro 体绘制仅支持平行投影，而光线投射和纹理映射方法对平行和透视投影都支持。

下面给出了这些体绘制技术的性能和局限的总结（见表 1） 。

体绘制算法		图像品质	绘制速度	算法特点	
空间域	光线投射法	最高	慢	可利用不透明度得到整体结构	占用内存大
	足迹法	高	中等		占用内存小
	剪切-曲变法	中等	最快		占用内存小
	纹理映射法	中等	最快		硬件加速，图像品质依赖于存储器位分辨率
变换域	频域体绘法	较高	快	X 光片效果，算法简洁	
	光线投射法	高	慢	占用内存大，	
	足迹法	较高	较快	X 光片效果，可逐渐显示，局部细节添加	

表 1　算法主要性能评价

4.5.3　创建 vtkVolume 类

vtkVolume 类是 vtkProp3D 的子类，它被用于体绘制中，类似于 vtkActor（它被用于几何绘制中）vtkVolume 类也提供了一些方法用于设置被绘制的体的一些信息，比如位置、方向、大小和体属性等，vtkVolume 类提供了 SetMapper()方法接收体映射器映射（vtkVolulmeMapper）的数据，提供了 SetProperty（）方法设定体属性数据，vtkActor、vtkVolume 是两个应用于不同绘制环境的角色对象，例如 vtkProperty 的 SetRepresentationToWireframe()方法在体绘制中是没有意义的，而 vtkVolumeProperty 的 SetInterpolationTypeToNearest()方法在几何绘制中则没有值。

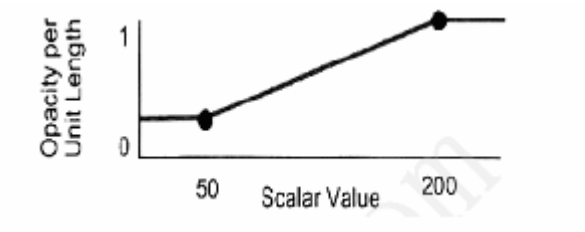
4.5.4　使用 vtkPiecewiseFunction 类

为了正确的对体数据进行绘制，在绘制前必须先设定体数据标量属性，第一个设定的属性是标量值的透明度，它将不同的标量值映射成不同的透明度，　第二个设定的属性是标量值的颜色，它将标量值映射为不同的颜色，第三个设定的属性是梯度不透明度，它将标量值梯度的大小映射为不透明度乘数，这些映射中的任何一个都可以被定义为单个值到单个值的映射，使用 vtkPiecewiseTransferFunction 来设定标量值的透明度属性，使用 vtkColorTransferFunction 类设定标量值的颜色属性，从用户的观点来看，vtkPiecewiseFunction 类提供了线性插值透明度映射方法：

当标量值和透明度被设定后，采用线性插值的方法映射这些设定的标量值之间的透明度，看下面的示例代码：

```
vtkPicewiseFunction *tFun=vtkPicewiseFunction::New();
tFun->AddPoint(50,0.2);
tFun->AddPoint(200,1.0);
```

示例代码将产生如下图所示的线性映射：

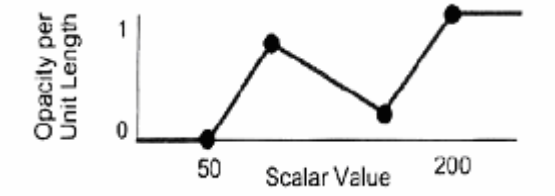


标量值为 50 和 200 的控制点的透明度分别设定为 0.2 和 1.0，其它的标量值（在 50 和 200 之间）的透明度可以利用这两个值线性插值得到，如果 Clamping 被设定为有效，那么任何低于 50 的标量值的透明度将被映射为 0.2，任何大于 200 的标量值的透明度被映射为 1.0，如果 Clamping 被设定为无效，那么任何此范围之外的透明度值都被映射为 0.0，在任何时候都可以添加新的控制点重新进行映射，除了可以添加单个控制点外，还可以增加一个段，段定义了两个控制点并且清除在这两个点之间的其它控制点，考虑下面修改（对前面给出的代码的部分片段）：

第一段代码：

```
vtkPicewiseFunction *tFun=vtkPicewiseFunction::New();
tFun->RemovePoint(50);
tFun->AddPoint(50,0.0);
tFun->AddSegment(100,0.8,150,0.2);
tFun->AddPoint(200,1.0)
```

示例代码将产生如下图所示的线性映射：

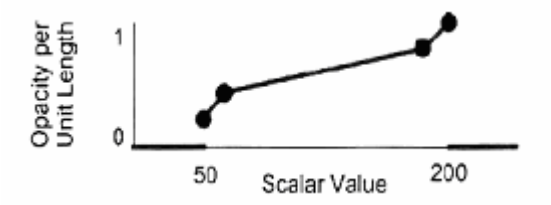


第二段代码：

```
vtkPicewiseFunction *tFun=vtkPicewiseFunction::New();
```

```
tFun->AddPoint(50,0.2);
tFun->AddSegment(60,0.4,190,0.8);
tFun->AddPoint(200,1.0)
tFun->ClampingOff();
```

示例代码将产生如下图所示的线性映射:



在第一段代码中，我们改变标量值 50 的映射是通过先移除这个点，然后再次增加，并且还增加了一个段，在第二段代码中我们改变标量值 50 的映射是通过直接增加一个新的映射，而没有移除旧的映射（如果一个映射被重新定义，那它将取代现有的映射），还增加了一个新的段，它清除原有的 100 和 150 的映射，并设置 clamping 无效。

4.5.5 使用 vtkColorTransferFunction 类

vtkColorTransferFunction 类被用来指定标量值到颜色的映射，使用 RGB 或 HSV 颜色空间，它的方法类似于 vtkPiecewiseFunction 类所提供的方法，但却有两种不同的颜色选择方案（RGB 或 HSV），例如，AddRGBPoin（）方法和 AddHSVPoint（）方法都可添加映射的控制点，前者使用的是 RGB 颜色值，后者使用的 HSV 颜色值。

下面代码显示了怎样设定颜色映射，使用 RGB 插值进行映射。

```
vtkColorTransferFunction
        *colorTransferFunction=vtkColorTransferFunction::New();
colorTransferFunction->AddRGBPoint(0,0,0,0);
colorTransferFunction->AddRGBPoint(64,1.0,0,0);
colorTransferFunction->AddRGBPoint(128,0.0,0,1.0);
colorTransferFunction->AddRGBPoint(192,0.0,1.0,0.0);
colorTransferFunction->AddRGBPoint(255,0.0,0.2,0.0);
```

4.5.6 使用 vtkVolumeProperty 类设定透明度和颜色值

前面已经详细的讨论了标量值的透明度映射和颜色映射，但是还没有描述如何为体数据

设定透明度和颜色值，一般的情况下，定义一个好的标量值的透明度和颜色值映射是非常重要的，因为它对体数据可视化的绘制效果有很大的影响，所以必须要准确理解体数据标量值潜在的含义。

标量值的透明度和颜色映射一般情况下用于对标量数据值所表达的含义进行分类，对于表示背景的标量值、表示噪声的标量值、与被表达的内容无关的标量值一般被映射为 0.0，表示透明，以消除这些无关的值对可视化结果的影响，其余的标量值应该按他们所表达的含义不同为它们定义不同的透明度和颜色值，例如，CT 扫描所获得的数据通常归类为空气、软组织、骨头，表示空气的标量值被映射为 0.0，表示软组织的标量值被映射为浅棕色，表示骨头的标量值可以被映射为白颜色，通过改变这些组织的透明度，就能够对皮肤表面或骨头表面进行可视化，如穿过半透明的皮肤看见骨头。

在 vtkVolumeProperty 中有一些方法涉及到颜色和不透明度转换函数，SetColor()方法接收 vtkPiecewiseFunction（如果你的颜色函数定义的仅仅是灰度值）或 vtkColorTransferFunction()作为输入。可以使用 GetColorChannels()查询颜色通道数，如果 vtkPiecewiseFunction 被设置为颜色，它将会返回 1，如果 vtkColorTransferFunction()被用来指定颜色，它将返回 3，一旦你知道有多少种颜色通道被使用，你就可以调用 GetGrayTransferFunction()方法或 GetRGBTransferFunction()方法来得到适合的转换函数。SetScalarOpacity()方法接收 vtkPiecewiseFunction 类作为输入，定义一个不透明度转换函数，使用相应的 GetScalarOpacity()方法来返回这个函数。类似的，梯度不透明度转换函数也有两种方法：SetGradientOpacity()和 GetGradientOpacity()方法。如果一个纹理被定义在 volume mapper 中，如果系统支持 3D 纹理绘制，那么使用 SetRGBTextrueCofficient()方法也可以控制体的颜色，这个方法设定的系数按照：

$$\text{标量值颜色}\times(1-\text{设定系数})+(\text{纹理颜色值}\times\text{设定系数})$$

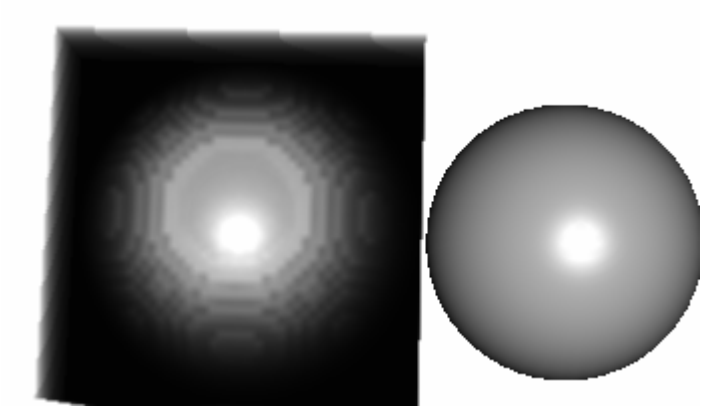
公式计算采样点的颜色，从上式可以看出，如果系数设置为 0.0 将忽略纹理，并且仅使用 ScalarColor 转换函数设定的标量值颜色，如果系数为 1.0 就将仅使用纹理颜色，忽略标量值颜色。

4.5.7 使用 vtkVolumeProperty 控制阴影

使用 vtkVolumeProperty 来控制体的阴影类似于使用 vtkActor 来控制一个几何体的阴影。在 VTK 中阴影由一个标志（允许/禁止）和四个基本参数（环境光系数，散射光系数，镜面光系数和镜面光强度）来控制。一般的情况下前三个系数之和为 1.0，有时为了增强被

绘制体的亮度，在体会制中这个值必须要大于 1，这些参数设定后体绘制的效果依赖于体绘制技术所使用的照度因素，总的说来，如果环境光占优势，那么绘制的体看起来没有色泽差别，不具有立体感，如果散射光占优势，那么绘制的体看起来具有很强的立体感，如果镜面光占优势，那么 绘制的体看起来将是平滑的，同时镜面光强度也被用来控制外表的光滑度。

默认情况下体绘制不使用阴影，必须调用 ShadeOn()允许阴影显示，设置阴影标志为禁止状态通常和将环境光系数设置为 1.0，散射光系数设置为 0.0，镜面光系数设置为 0.0 等同。某些体绘制技术，比如体射线投射技术中的最大密度投影法不会考虑阴影系数，无论阴影标志的状态是禁止或是允许的。一个体的阴影的外观不仅仅依赖于阴影系数的值，而且还与包含在绘制者和它们的属性中的光源有关，下面给出在体绘制时如何使用阴影的示例程序，程序运行结果如下：



示例程序代码（Examp44）：

```
//读取体数据
vtkSLCReader *reader=vtkSLCReader::New();
reader->SetFileName("sphere.slc");
//设定体数据中不同标量值的透明度
vtkPiecewiseFunction *tfun = vtkPiecewiseFunction::New();
//tfun->AddPoint(0,0.01);
tfun->AddPoint(50,0.2);
tfun->AddPoint(200,1.0);
//设定灰度
vtkPiecewiseFunction  *opacityTransferFunction
                = vtkPiecewiseFunction::New();
opacityTransferFunction->AddSegment(0,1.0,255,1.0);
```

```
//设定体数据的属性
vtkVolumeProperty *volumeProperty = vtkVolumeProperty::New();
volumeProperty->SetColor(opacityTransferFunction);
volumeProperty->SetScalarOpacity(tfun);
//显示阴影
volumeProperty->ShadeOn();
//设定插值类型为线性插值
volumeProperty->SetInterpolationTypeToLinear();
volumeProperty->SetDiffuse(0.7);
volumeProperty->SetAmbient(0.01);
volumeProperty->SetSpecular(0.5);
volumeProperty->SetSpecularPower(70.0);
//使用光线投射算法
vtkVolumeRayCastCompositeFunction *compositeFunction=
            vtkVolumeRayCastCompositeFunction::New();
vtkVolumeRayCastMapper *volumeMapper=vtkVolumeRayCastMapper::New();
volumeMapper->SetVolumeRayCastFunction(compositeFunction);
volumeMapper->SetInput((vtkImageData *) reader->GetOutput());
vtkVolume *volume = vtkVolume::New();
volume->SetMapper(volumeMapper);
```

在示例程序所绘制的图像中，左边的球是使用体射线投射所绘制的一个体，右面的球是使用 OpenGL 绘制的一个几何形，被以同样的环境光、散射光、镜面光及强度设置，所以两个球的颜色都是白色的，具有相似的外观。

4.5.8 创建 vtkVolumeMapper 映射器

vtkVolumeMapper 是一个抽象的父类，不能用它来直接的创建对象，然而，你可以创建一个具体的所需类型的体映射器的子类，在 VTK 中提供了以下三个子类：

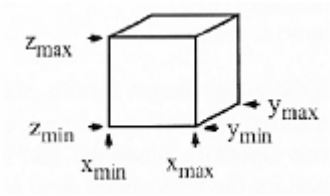
- vtkVolumeRayCastMapper
- vtkVolulmeTextureMapper2D

- vtkVolumeProMapper

有些体绘制技术仅仅支持特定类型的输入数据，例如：vtkVolumeRayCastMapper 类和 vtkVolumeTextureMapper2D类都仅支持 VTK_UNSIGNED_CHAR和VTK_UNSIGNED_SHORT类型数据。

4.5.9 切割体数据

对于数据量比较大的体数据，如果全部进行绘制那么绘制的效率非常低，由于经常只观察体的一部分数据，所以一般都是将感兴趣的部分数据裁减下来进行绘制，VTK 提供了剪切和裁剪两种技术来限制所绘制的数据量，剪切使用六个剪切面来定义体的可见区域（沿着每个主坐标轴方向有两个面），剪切面的位置使用数据坐标确定，一般的情况下使用这六个剪切面定义一个感兴趣的子体，如下图所示：



如果要对体数据剪切，必须将剪切标志设定为允许状态，并且在体映射器中设定剪切平面，下面给出示例代码：

```
float xmin=20.0,xmax=50.0;
float ymin=0.0,ymax=33.0;
float zmin=21.0,zmax=47.0;
vtkVolumeRayCastMapper *Mapper=vtkVolumeRayCastMapper::New();
//设置剪切标志为允许状态
Mapper->CroppingOn();
//设置剪切区域
Mapper->SetCroppingRegionPlane(xmin,xmax,ymin,ymax,zmin,zmax);
Mapper->SetCroppingRegionFlagsToSubVolume()
```

除了上面的例子使用vtkVolumeRayCastMapper映射器外，也可以使用vtkVolumeMapper体映射器的子类，因为所有的剪切方法都是被定义在vtkVolumeMapper类中。

通过设置 xmin、xmax、ymin、ymax、zmin、zmax 的值定义六个剪切面，将体数据分割成 27 个区域（每个区域大小 3×3 网格），属性 CroppingRegionFlags 是一个 27 位数，它的

每一位代表一个区域，若这个位值为 1 则表明在这个区域中的数据是可见的，值为 0 表明这个区域中的数据将被剪切，小于 xmin、ymin、zmin 值的体的区域由第一个位来代表。区域的顺序按照 x 轴向、y 轴向、z 轴向来定义，使用 SetCroppingRegionFlagsToSubVolume()方法设置 falgs 为 0x0002000，表示仅中间区域可见，尽管任意 27 位数都能被用于定义 cropping 区域， 但在实际中仅有较少的一些被使用。

VTK 还提供了另外有四种方便的方法来设置 CroppingRegionFlags：

SetCroppingRegionFlagsToFence()

SetCroppingRegionFlagsToInvertedFence()

SetCroppingRegionFlagsToCross()

SetCroppingRegionFlagsToInverted Cross (),

具体描述见图 4-3 所示

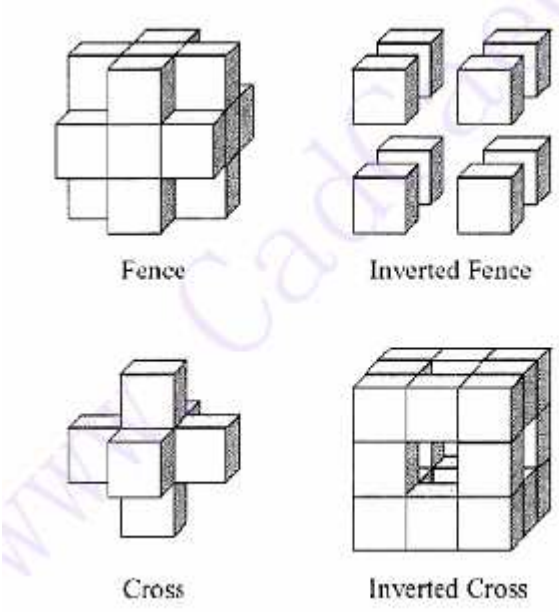


图 4-3　不同类型的剪切区域标志

4.5.10 剪取体数据

vtkAbstractMapper3D 映射器中提供了基于硬件的任意剪取平面（clipping planes）功能，该映射器剪取的平面不能多于 6 个，vtkVolumeRayCastMapper 映射器提供了基于软件的任意剪取平面功能，该映射器剪取的平面不受数量限制。vtkVolumeProMappe 映射器虽然提供了设定剪取平面的方法，但是该映射器不支持剪取平面功能。

剪取平面使用 vtkPlane 类创建，使用这个类定义平面参数，然后使用 AddClippingPlane()方法将这个平面增加到映射器中。

4.5.11 用射线投射法进行体绘制

vtkVolumeRayCastMapper 映射器使用基于软件方式的射线投射技术来进行体绘制，和其它的体绘制技术相比，该技术绘制的精确度比较高，但是速度也最慢，在前面示例程序中，有关射线投射技术一些特有的参数还没有被讨论，首先，射线投射函数必须被设置在体映射器中，该函数决定了沿着射线对数据采样后采样数据最终的 RGBA 值。在 VTK 中射线投射技术提供了三种射线函数，适用于不同的场合，这三个函数说明如下：

- vtkVolulmeRayCastIsosurfaceFunction

使用等值面法进行体绘制。

- vtkVolulmeRayCastMIPFunction

使用最大密度投影法进行体绘制。

- vtkVolulmeRayCastCompositeFunction

使用 alpha 合成法进进行体绘制。

可以根据需要使用以上三种不同的射线函数进行体绘制，但是需要注意的是当设定的透明度线性变化较大时，alpha 合成法绘制所产生的图像和等值面法绘制所产生的图像不容易区分。

每一种射线投射函数中都有一些参数，这些参数可以影响体数据的绘制过程。

对于 vtkVolulmeRayCastIsosurfaceFunction 类，使用 SetIsoValue()方法可以设置被绘制的等值面的值。

对于 vtkVolulmeRayCastMIPFunction 类，调用 SetMaximizeMethodToScalarValue()方法时，沿着射线的每一个采样点的标量值被考虑，具有最大标量值的采样点被选择，然后这个标量值被传递给颜色和不透明度转换函数来生成最后的射线值，如果调用 SetMaximizeMethodToScalarValue()方法，沿着射线的每一步采样的不透明度将被计算，具有最大不透明度的采样点将被选择。

对于 vtkVolumeRayCastCompositeFunction 类，调用不同的方法可以改变插值优先或分类优先的顺序，调用 SetCompositeMethodToInterPolateFirst()方法，首先使用插值计算采样点的标量值，然后这个值将被用于分类（应用颜色和不透明度转换函数），调用 SetCompositeMethodToClassifyFirst()方法，首先计算采样区域区域单元 8 个顶点的 RGBA 值，然后根据 8 个顶点的 RGBA 值来计算采样点的值，只有三线性插值被使用的时候，系统

才会考虑考虑插值优先或分类优先的顺序。通常插值优先的三维重建效果要明显好于分类优先的，但是插值优先的算法会出现一定程序上的失真，比如:在体绘制中，骨头是像素值最大的数，空气是像素最小的数，中间是软组织数据，在插值优级先中，骨头外边是肌肉等软组织数据，但是如果要体绘牙齿的话，插值优级算法会自动为牙齿外部插值出一层软件组织数据，这明显不全实际要求。插值类型实例变量的值在 vtkVolumeProperty 中是非常重要的，对于射线投射体绘制来说，有两种选择：

- 使用默认的 SetInterpolationTypeToNearest()方法

当沿着射线采样时，它使用最近邻近似值。

- 使用 SetInterpolationTypeToLinear()方法

当沿着射线采样时，使用三线性插值。

使用三线性插值产生较平滑的图像，但是通常要花费较长的时间。使用这两种方法所产生的图像质量不同，如图 4-4 所示：一个球被体素化（voxelized）为 50×50×50 体素的体，并且被使用最近邻插值的 alpha 合成法绘制(图 4-4 的左边)和三线性插值的 alpha 合成法绘制（图 4-4 的右边） 。在原始的图像中可能很难区分出这两种插值法， 但是通过放大图像到仅看见球的一部分时，在左图中很容易看见个别的体素。

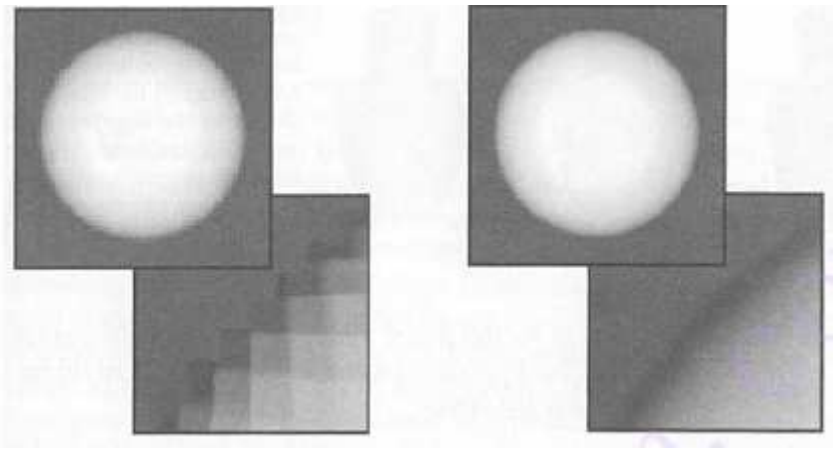


图 4-4　不同插值方法生成的图像

vtkVolumeRayCastMapper 的另外一个影响图像的参数是 SampleDistance（采样间隔），表示在世界坐标系中相邻采样点之间的距离，设置一个小的采样间距或称插值间距可以提高三维重建的质量,但其运算的数据量将会大量的增大，但这将花费较多的绘制时间。最大密度射线函数也执行采样来定位最大值。等值面射线函数不执行采样，但它根据当前的插值函数计算交叉点（intersection）的准确位置。

4.5.12 二维纹理映射法体绘制

该方法也称为三维纹理映射法，三维纹理映射方法是基于硬件来提高体绘制的速度，所谓基于硬件的三维纹理映射指的是，在纹理空间中实现重采样的插值运算及具有不透明度值的图象合成等操作均由硬件完成，从而大大提高了运算速度。这种方法首先将体数据作为三维纹理图装入纹理内存，然后在体数据内部定义一系列采样多边形去采样物体的纹理，再通过查找表将采样得到的数据转换为相应的颜色值及不透明度值，这样就可以按照从后向前的顺序进行图象合成，投影于视平面而形成最后的图象。

3D 纹理映射的实现需要昂贵的专用图形硬件，限制 3D 纹理影射绘制图像质量的主要因素归结于其画面帧存储器有限的位分辨率（8 至 12 位），这远低于软件算法中使用的浮点数的精度。特别是有限的位分辨率严重制约了不透明度加权颜色（或亮度）方法的使用，因为低不透明度的体素经不透明度加权后，其体素的颜色（或亮度）值会降低到帧存储器分辨率以下，这就限制了具有低不透明度、低密度体素区域的合成绘制。一种解决方法是按比例地放大体素颜色（亮度）值，但这样有时会造成其它区域值的饱和。使用纹理映射法的示例代码如下：

```
//纹理映射法
vtkOpenGLVolumeTextureMapper3D *volumeMapper=
vtkOpenGLVolumeTextureMapper3D::New();
volumeMapper->SetInputConnection(readerImageCast->GetOutputPort());
//表示透示图中的一组三维数据
vtkVolume *volume = vtkVolume::New();
volume->SetMapper(volumeMapper);
```

纹理映射法的体绘制质量不如光线投影法，但绘制速度快。

5 VTK 数据接口对象

在本章中将会详细介绍关于 VTK 中各种数据接口的信息，在 VTK 中数据是我们所处理的对象，而 VTK 程序对数据对象的处理机制是基于流水线的（pipelines），通过流水线的过滤器（filter）来处理数据，以达到我们所期望的结果。

为了便于了解 VTK 中的数据对象关系，请看下图：

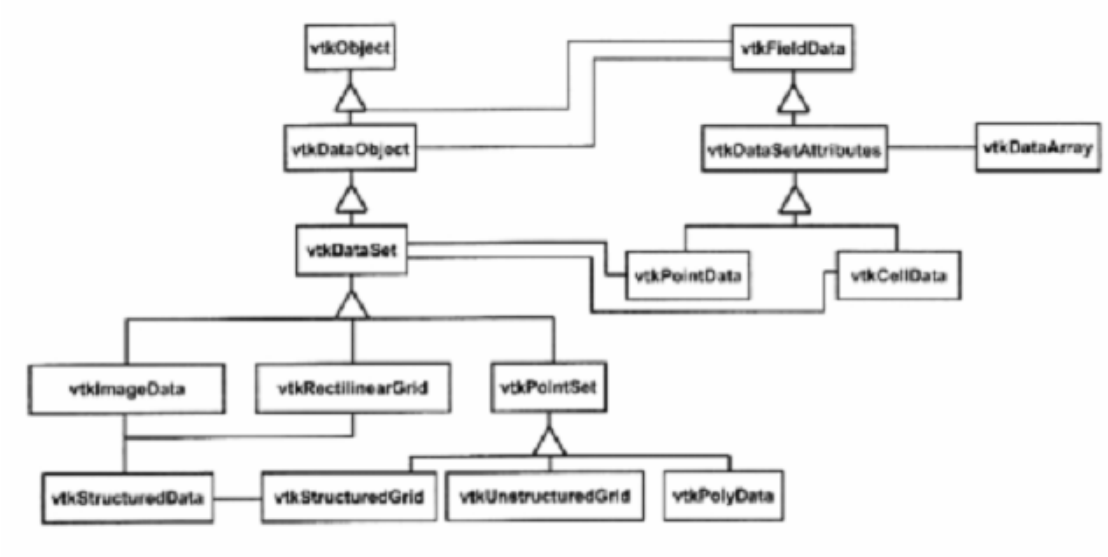


图 5-1 数据对象模型图

5.1 数据数组

vtkDataArray 类是所有数据数组对象的抽象超类，该类自定义了很多应用程序接口，可以说 vtkDataArray 是 VTK 中数据对象建立的基础，数据数组用来存储连续的同一类型的数据，如（char、int、float 等），在定义数据数组的过程中必需动态的为其分配内存空间，此外，VTK 还为我们提供了专门的方法来对数据数组进行操作，如返回数组指针、插入新的数据等。

数据数组中的元素是由组元(tuple)组成的，而每一个组元也是一种结构体，由 N 个数据类型相同的分量组成，数据数组可以用来存储的数据包括：几何数据、属性数据（矢量，向量，张量，法线向量）等。

数据数组的构成如下图所示：

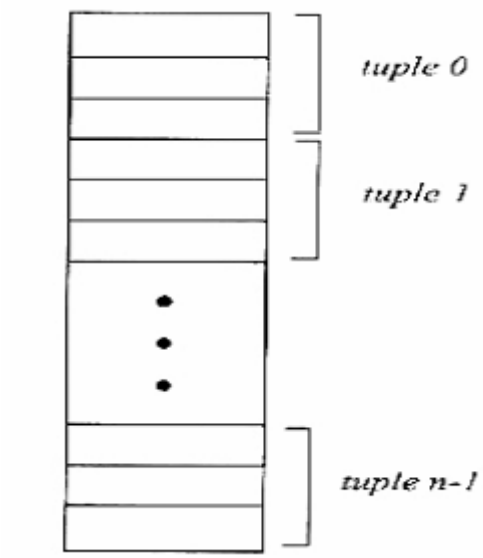


图 5-2　数据数组结构

在图 5-2 中，数据数组可以存储多个组元，每个组元由三个分量组成，。

对于 vtkDataArray 的派生类以及关系由下图表示：

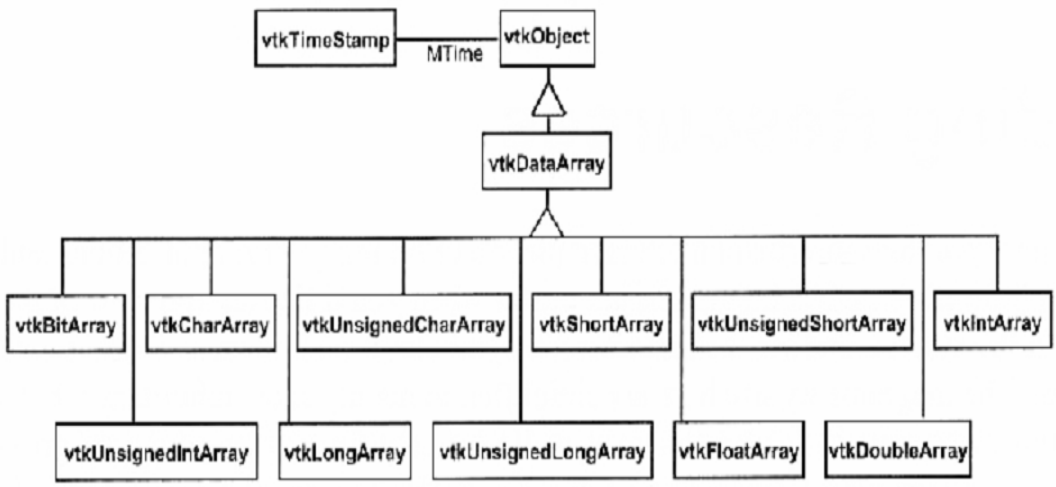


图 5-2　　数据数组类层次图

数据数组提供的方法：

在 vtkDataArray 类中，为我们提供了大量的方法来对 vtkDataArray 对象进行操作，这些方法都可以直接应用于 vtkDataArray 的继承类的对象中，当然做为 vtkDataArray 的子类，也会有其自己特殊的方法仅供该子类对象使用。

MakeObject（）

用来创建一个与当前 vtkDataArray 实例对象相同类型的数据队列，使用时通常这样写：

dataArray=MakeObject();

virtual int vtkPoints::GetDataType () ;

返回一个整数来表示数据队列中的数据类型，返回的类型可能是：

VTK_BIT

VTK_CHAR

VTK_UNSIGNED_CHAR

VTK_SHORT

VTK_UNSIGNED_SHORT

VTK_INT, VTK_UNSIGNED_INT

VTK_LONG,

VTK_UNSIGNED_LONG

VTK_FLOAT, VTK_DOUBLE

VTK_ID_TYPE

virtual void vtkDataArray::SetNumberOfComponents(int) [virtual]

确定组元分量的个数。

int vtkDataArray::GetNumberOfComponents() [inline]

得到组元分量的个数。

virtual void vtkDataArray::SetNumberOfTuples(const vtkIdType number)

设定数据数组中存储组元的个数。

vtkIdType vtkDataArray::GetNumberOfTuples()[inline]

返回数据数组中组元的个数。

virtual float* vtkDataArray::GetTuple(const vtkIdType i)

返回数据数组中第 i 个组元分量的指针。

virtual void vtkDataArray::InsertTuple (const vtkIdType i, const float * tuple)

在数据数组中的第 i 个元素位置插入一个新的组元。

vtkIdType vtkDataArray::InsertNextTuple(const float * tuple)

在数据数组中最后的位置插入一个新的组元。

virtual float vtkDataArray::GetComponent (const vtkIdType i, const int j)

返回数据数组中第 i 个组元中的第 j 个分量所存储的数值。

virtual void vtkDataArray::SetComponent (const vtkIdType i, const int j, float c)

设置数据数组中第 i 个组元中的第 j 个分量中所存放的数值为 c。

virtual void vtkDataArray::GetData(vtkIdType tupleMin, vtkIdType tupleMax, int compMin, intcompMax, vtkFloatArray * data)

数据获取方法，在 data 中存放数据数组组元范围(tupleMin,tupleMax)，且组元分量值的范围为(compMin, compMax)之间数据。

virtual void vtkDataArray::DeepCopy (vtkDataArray * da) [virtual]

深度复制意味着将所有数据全都复制下来，而不是仅仅拷贝指针，比如有的数组中的数据是无符号整形，有的是无符号字符型，体现在对不同类型的数据均可实现拷贝上,deep 的含义应该就是如此。

virtual void vtkDataArray::Squeeze()[pure virtual]

压缩数据，释放不需要的内存空间。

virtual void vtkDataArray::Resize (vtkIdType numTuples) [pure virtual]

重新设置数据数组的大小

void vtkDataArray::Reset () [inline]

将数据数组中的数据清空。

vtkIdType vtkDataArray::GetSize () [inline]

返回数据数组的大小。

void vtkDataArray::CreateDefaultLookupTable ( )

默认的生成数据数组的伪彩映射表。

void vtkDataArray::SetLookupTable(vtkLookupTable * lut)

为数据数组设置伪彩映射表。

5.2 数据集对象

在编写过滤器对象时最难处理的是过滤器和 VTK 数据对象的连接，为了编写一个高效的过滤器，不仅要熟悉算法还要理解数据模型，理解 VTK 数据模型的一个重要方面是着重理解数据集结构和数据集属性。

数据集是 vtkDataObject 数据对象的一个子类，这两个类之间的区别是数据集有拓扑结构和几何结构，而数据对象则表现一般的数据域，数据集主要由点和点之间或单元和单元之间的几何和拓扑信息相互关联组成，数据集的属性数据用于描述数据集的几何结构和拓扑结

构含义。

VTK 中的数据集有两个非常重要的特性：结构和单元，如果想要对 VTK 中的数据模型有了很深入的认识，那么必需要知道如何从数据集中得到你想要的那部分数据、如何创建一个数据集、如何为已有的数据集添加新的数据，对于不同的数据集类型有不同的接口方式，数据集根据结构分为如下几种类型，类层次图如图 5-3 所示：

1) structured points(结构化点集)
2) vtkRectilinearGrid（线性网格）
3) structured grid(结构化网格)
4) unstructured grid(非结构化网格)
5) Polygonal Data(多边形数据)

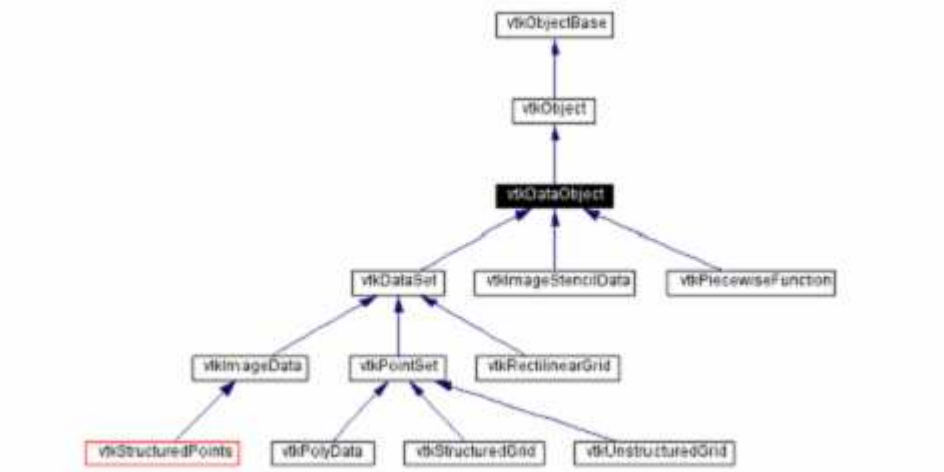


图 5-3 数据集类层次图

如果你对 VTK 数据模型已经有了一个透彻的理解，那么你需要知道怎样从数据集得到数据、怎样创建数据集和把数据放进数据集中，不同的数据集类型供了不同的实现方式，例如，为了得到 vtkPolyData 的点坐标，可以调用父类的方法 vtkDataSet::GetPoint（）或者用 pts=vtkPolyData::GetPoints()方法在 vtkPolyData 中获得点数据集中的点集数组，接下来用 pts->GetPoint()方法获取点坐标。

5.3 vtkDataSet 数据接口

vtkDataSet 类是数据集的抽象类，抽象类是不能被直接实例化的类，在 vtkDataSet 类中同样提供了大量的方法，这些方法都可以直接应用于 vtkDataSet 子类的对象中，vtkDataSet 的方法与 vtkDataArray 相类似，方法的含义可以参考帮助文档。

5.4 vtkImageData 类的数据接口

vtkImageData 是一种规则的结构化数据，vtkImageData 数据在使用前必需要定义好其维数、空间间距、起始点的空间位置，如果 vtkImageData 对象的维度为 2D 则表示为一幅图像，其组成单元为像素点(vtkPixel)，如果 vtkImageData 对象的维度为 3D 则表示为体数据，其内部组成单元为体素（vtkVoxel）。

以下程序是从 vtkExtractVOI 类中选取的部分原程序，vtkExtractVOI 的功能是对输入数据进行重新采样操作，从图像图形数据中选取一小块做其它处理。

程序实例：

```
//如果要选取的范围与数据原始大小相同则报错处理
if ( outDims[0] == dims[0] && outDims[1] == dims[1] && outDims[2] == dims[2] &&
        rate[0] == 1 && rate[1] == 1 && rate[2] == 1 ) {
        output->GetPointData()->PassData(input->GetPointData());
        output->GetCellData()->PassData(input->GetCellData());
        vtkDebugMacro(<<"Passed data through because input and output are the same");
         return 1;
}
outPD->CopyAllocate(pd,outSize,outSize);//点数据
outCD->CopyAllocate(cd,outSize,outSize);//Cell 数据
//为输出分配必要空间
sliceSize = dims[0]*dims[1];//X 轴*Y 轴，表每一个切片的大小
// Traverse input data and copy point attributes to output
// 将所选取的区间内的点数据属性拷贝到新的存储区间中
newIdx = 0;//点计数器
/ /Z 轴，voi[6]是表示选取的三个轴上的数据区域
for ( k=voi[4]; k <= voi[5]; k += rate[2] ) {
      kOffset = (k-inExt[4]) * sliceSize;
for ( j=voi[2]; j <= voi[3]; j += rate[1] )//Y 轴
{
     jOffset = (j-inExt[2]) * dims[0];
```

```
        for ( i=voi[0]; i <= voi[1]; i += rate[0] )//X 轴
        {
            idx = (i-inExt[0]) + jOffset + kOffset;
            outPD->CopyData(pd, idx, newIdx++);//单点的数据信息拷贝
        }
    }
}
```

5.5 vtkPointSet 的数据接口

vtkPointSet 是一个抽象类，具体使用时使用它的子类存储点的坐标，而 vtkPolyData 和 vtkUnstructuredGrid 需要使用 vtkPointSet 子类设定点的坐标位置。

vtkPointSet 重要的方法有：

```
//返回数据集中点的数目
numPoints=GetNumberOfPoints();
//返回一个 vtkPoints 类型的指针，指向数据集中的数据点
Points=GetPoints();
//为数据集设定点
SetPointes（Points）;
```

以下程序是从 vtkWarpVector 类的选取的部分原代码。

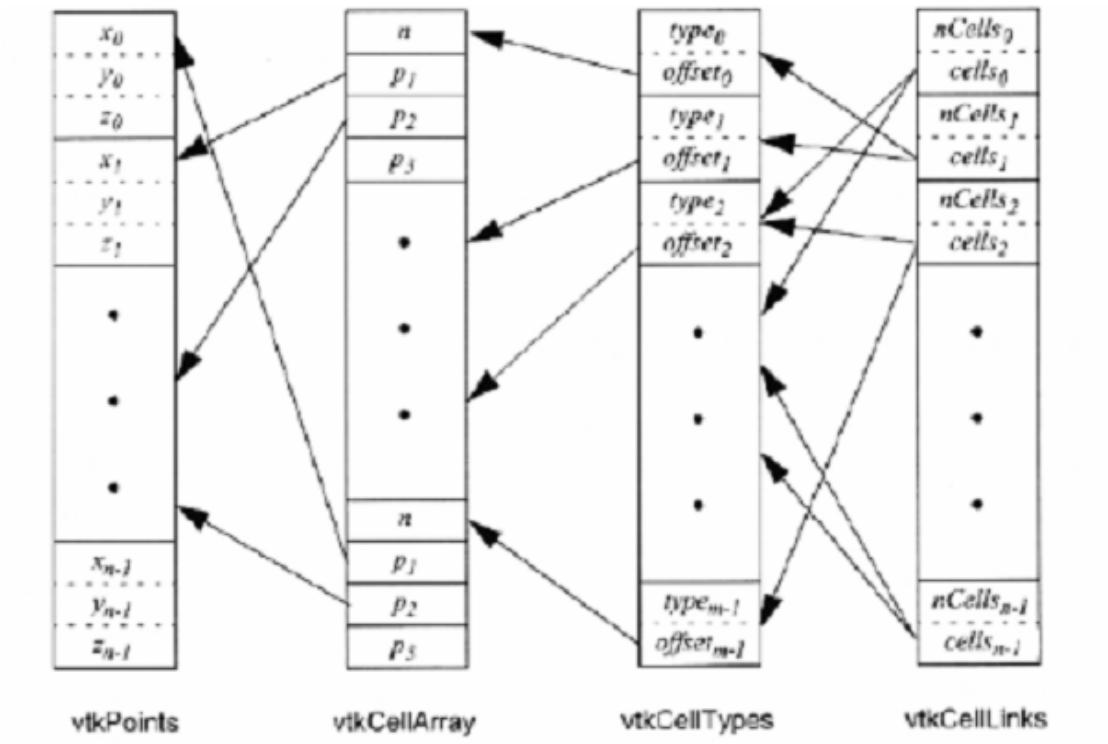
```
vtkPointSet *output = vtkPointSet::SafeDownCast(
            outInfo->Get(vtkDataObject::DATA_OBJECT()));//定义数据输出对象
vtkPoints *points;
vtkIdType numPts;
// 拷贝 input 中的点到 output 中
output->CopyStructure( input );
//输入中点个数
numPts = input->GetNumberOfPoints();
points = input->GetPoints()->NewInstance();
points->SetDataType(input->GetPoints()->GetDataType());
```

```
points->Allocate(numPts);//分配内存空间
points->SetNumberOfPoints(numPts);
output->SetPoints(points);
//遍历所有点对象，重新调节其位置
for (ptId=0; ptId < max; ptId++)
{
        if (!(ptId & 0xfff))
        {
        self->UpdateProgress ((double)ptId/(max+1));
        if (self->GetAbortExecute())
           {
           break;
           }
        }

      *outPts = *inPts + scaleFactor * (T1)(*inVec);//为输出重新调节其 x 轴位置
      outPts++; inPts++; inVec++;
      *outPts = *inPts + scaleFactor * (T1)(*inVec); //为输出重新调节其 y 轴位置
      outPts++; inPts++; inVec++;
      *outPts = *inPts + scaleFactor * (T1)(*inVec); //为输出重新调节其 z 轴位置
      outPts++; inPts++; inVec++;
}
```

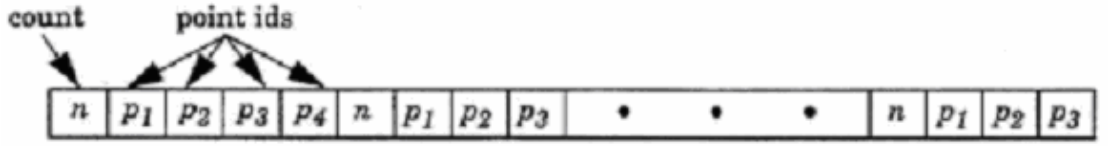
5.6 vtkPolyData 的数据接口

vtkPolyData 数据类型是由一些基本图元（如：顶点、线、多边形、三角面片等）组成的复杂图形，这种数据是完全无结构的，它含有点集和单元，点集属性继承自 vtkPointSet，单元属性则来自 vtkCellArrays，对于无结构数据 vtkPolyData 中的点与单元必需要定义其位置。



上图为无结构数据的结构关系图，这种关系不仅可以表示 vtkPolyData 数据格式，还可以表示 vtkUnstructuredGrid 数据格式。

在 vtkPolyData 数据中用 vtkCellArray 来确定数据单元的拓扑属性，vtkCellArray 类是一个关联所有单元的列表，每一个单元都是由多个点所组成，在这个列表的第一个数据是组成这个单元的的点的个数，如下图：



vtkCellTypes 是对 vtkPolyData 等非结构数据的补充说明，对于每一个单元都用一个整形变量来表示该单元的类型，由于 vtkCellArray 不能记录一个单元的类型，所以用 vtkCellTypes 来指向 vtkCellArray 中的数据项，以表示该单元是何种类型的。

vtkPoints 是 vtkDataArray 的继承类，vtkPoints 用于存储 x,y,z 坐标点的信息,通常我们要明确地定义每一个顶点的位置,如下程序所示：

```
static float x[8][3]={{0,0,0}, {1,0,0}, {1,1,0}, {0,1,0},
                      {0,0,1}, {1,0,1}, {1,1,1}, {0,1,1}};
vtkPoints *points = vtkPoints::New();
//存储 8 个顶点
```

for (i=0; i<8; i++) points->InsertPoint(i,x[i]);

定义点坐标位置的数据可以是整数也可以是实数。

5.7 vtkCell 的数据接口

单元是 VTK 中数据类型的基础，通常对图形数据的操作都是对单元的操作，如进行插值、坐标变换、坐标搜索、图形的几何变换等，从一个图形数据中提取单元可以用方法 GetCell(int cellID)来实现。

单元是由一种确定类型的一系列顶点组成的基本图元，具有二维（对于面）或三维（对于体）的维度，它是组成其他复杂图形的基础。

6 建立模型

我们已经知道怎样使用源对象（读源对象和程序源对象）来创建几何模型，VTK 还提供了其它技术用于生成更加复杂的几何模型，本章将介绍用挤压、隐模型、表面重构三种技术结合散乱点集来建立几何模型。

对于缺少拓扑结构和几何形状的离散点数据，VTK 采用场可视化的方式对这些数据进行可视化，当然，我们也可以使用本章介绍的技术将这些数据处理成 VTK 提供的数据集类型，然后进行可视化。

6.1 隐模型

隐模型是一种借用 3D 等直线（isosurface 生成）来产生多边形表面网格的技术，等直线生成一般被应用在包含属性数据的 vtkImageData 数据集类型中（一种规则的体数据），vtkImageData 数据中每个数据点的属性数据可以通过采样或卷积的方法生成。

6.1.1 定义隐函数

下面的例子说明如何用线段生成复杂的多边形表面，这个例子将线段排列成字母“HELLO”，结果如图所示：

例子代码段如下（Examp61）：

```
//创建多边形数据对象，用线画出 hello 这个单词
vtkPolyData *pPolyData=vtkPolyData::New();
pPolyData->SetPoints(myPoints);//存储离散点集
pPolyData->SetLines(pLineCell);//设置线单元,15 单元，拼出 hello 单词
//创建映射器对象
vtkPolyDataMapper *lineMapper = vtkPolyDataMapper::New();
lineMapper->SetInput(pPolyData);
//创建线角色对象
vtkActor *lineActor = vtkActor::New();
lineActor->SetMapper(lineMapper);
lineActor->GetProperty()->SetAmbientColor(1 , 0 , 0);
lineActor->GetProperty()->SetColor(1 , 0 , 0);//设置线的颜色为红色
vtkImplicitModeller *imp = vtkImplicitModeller::New();
imp->SetInput(pPolyData);
imp->SetOutput(imp->GetOutput());
imp->SetMaximumDistance(12.55);//设置计算距离值的最大范围
imp->SetSampleDimensions(110 , 40 , 20);//设置输出的结构化点集的三个维度的大小
imp->SetModelBounds(-1.0,10.0,-1.0,3.0,-1.0,1.0);
VtkContourFilter *contour = vtkContourFilter::New();
contour->SetInput((vtkDataSet *) imp->GetOutput());
contour->SetValue(0 , 0.35);
vtkPolyDataMapper *impMapper = vtkPolyDataMapper::New();
impMapper->SetInput((vtkPolyData *) contour->GetOutput());
```

```
vtkActor *impActor = vtkActor::New();
impActor->SetMapper(impMapper);
```

在这个例子中，用 15 条线构成多边形数据，15 条线划出“HELLO”这个单词，每条线作为基本图元，vtkImplicitModeller 类创建爱的对象计算出离这些线最近的距离的点，挑选出满足设定的距离范围内的点构成结构化点集数据集，并将构成的结构化点集输出，由 VtkContourFilter 对象处理后，生成多边形等值面，等值面的数据值为计算的距离值。

vtkImplicitModeller 类提供了两个重要的方法，SetMaximumDistance（）用于控制计算距离的范围，如果这个值设定的比较小的话，计算速度比较快，但是设定的太小的话，生成的等值面将要发生变形，达不到所希望的效果，SetSampleDimensions（）方法控制输出的结构化数据集在 x、y、z 三个方向的维度的大小，设定的值越大，图形显示的分辨率越高，但是计算越慢，SetModelBounds（）方法控制输出数据集在空间的位置。

6.1.2 对隐函数采样

另一种强大的建模技术，是使用隐函数，隐函数有如下的形式：$F(x,y,z)=0$，球体、锥体、椭球、平面以及其他许多有用的几何实体可以用隐函数来描述，例如，一个半径为 R 的球体 S 可由方程 $F(x,y,z)=R^2-X^2-Y^2-Z^2$ 描述，当 $F(x,y,z)=0$ 时 ，方程可以准确的描述出这个球体。

除了建模，对隐函数也可以进行布尔运算，这些运算允许我们建立更加复杂的模型，下面的例子描述如何创建一个冰淇淋锥体，锥和两个平面交叉（创建一个有限程度的锥体）并用另一个球体的模拟冰淇淋被咬了一口的效果，例子结果如图所示：

例子代码如下（Examp62）：

```
//使用隐函数创建一个锥体
vtkCone *cone = vtkCone::New();
//设置锥角为 20 度
cone->SetAngle(20);
//使用隐函数创建 2 个平面
vtkPlane *vertPlane = vtkPlane::New();
vertPlane->SetNormal(-1 , 0 , 0);
vertPlane->SetOrigin(0.2 , 0.0 , 0);
vtkPlane *basePlane = vtkPlane::New();
basePlane->SetNormal(1 , 0 , 0);
basePlane->SetOrigin(1.2 , 0 , 0);
vtkSphere *iceCream = vtkSphere::New();
iceCream->SetCenter(1.333 , 0 , 0);
iceCream->SetRadius(0.5);
//建立一个球体
vtkSphere *bite = vtkSphere::New();
bite->SetCenter(1.5 , 0 , 0.5);
bite->SetRadius(0.25);
```

```
//对隐函数进行布尔操作
vtkImplicitBoolean *theCone = vtkImplicitBoolean::New();
//求平面和椎体交操作，构建新的隐函数
theCone->SetOperationType(1);
theCone->AddFunction(cone);
theCone->AddFunction(vertPlane);
theCone->AddFunction(basePlane);
//创建冰激淋被咬得效果，球体 iceCream 减去 bite，构建新的隐函数
vtkImplicitBoolean *theCream = vtkImplicitBoolean::New();
//求差运算
theCream->SetOperationType(2);
theCream->AddFunction(iceCream);
theCream->AddFunction(bite);
//对 theCone 隐函数进行采样
vtkSampleFunction *theConeSample = vtkSampleFunction::New();
theConeSample->SetImplicitFunction(theCone);
theConeSample->SetModelBounds(-1 , 1.5 , -1.25 , 1.25 , -1.25 , 1.25);
theConeSample->SetSampleDimensions(60 , 60 , 60);
theConeSample->SetComputeNormals(0);
//生成几何体
vtkContourFilter *theConeSurface = vtkContourFilter::New();
theConeSurface->SetInput((vtkDataSet *) theConeSample->GetOutput());
theConeSurface->SetValue(0 , 0);
vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();
coneMapper->SetInput((vtkPolyData *) theConeSurface->GetOutput());
vtkActor *coneActor = vtkActor::New();
coneActor->SetMapper(coneMapper);
coneActor->GetProperty()->SetColor(1 , 0 , 0);
coneActor->SetVisibility(1);
//对 theCream 隐函数进行采样
```

```
vtkSampleFunction *theCreamSample = vtkSampleFunction::New();
theCreamSample->SetImplicitFunction(theCream);
theCreamSample->SetModelBounds(0 , 2.5 , -1.25 , 1.25 , -1.25 , 1.25);
theCreamSample->SetSampleDimensions(60 , 60 , 60);
theCreamSample->SetComputeNormals(0);
//生成几何体
vtkContourFilter *theCreamSurface = vtkContourFilter::New();
theCreamSurface->SetInput((vtkDataSet *) theCreamSample->GetOutput());
theCreamSurface->SetValue(0 , 0.0);
vtkPolyDataMapper*creamMapper=vtkPolyDataMapper::New();
creamMapper->SetInput((vtkPolyData *) theCreamSurface->GetOutput());
creamMapper->SetScalarVisibility(0);
vtkActor *creamActor = vtkActor::New();
creamActor->SetMapper(creamMapper);
creamActor->GetProperty()->SetColor(0.74 , 0.99 , 0.79);
creamActor->SetVisibility(1);
```

在这个例子中，vtkSampleFunction 对布尔运算后的隐函数进行采样，用采样点建立 vtkImageData 数据对象输出，输出数据的精度由 SetSampleDimensions（）方法设定，vtkContourFilter 类用于生成逼近采样函数的轮廓面。

6.2 挤压

挤压是对存在的对象沿着一条路径进行扫描并创建表面的建模技术，例如，我们可以沿垂直的方向扫描一条线，创造一个平面。

VTK 提供了两种挤压的方法：线性挤压和旋转挤压，在 VTK 中，被扫描的对象是 vtkPolyData 数据对象，这些对象被用于生成挤压表面，vtkLinearExtrusionFilter 被用于沿着直线路径扫描对象，生成挤压表面，vtkRotationalExtrusionFilter 被用于沿着旋转路径扫描对象，生成挤压表面。

6.3 表面重构

我们经常希望通过一组离散的点或其数据它去构建一个表面，这些点可能来自激光数字化系统、测绘系统等，本节主要讲述如何通过这些离散点重新构建表面模型。

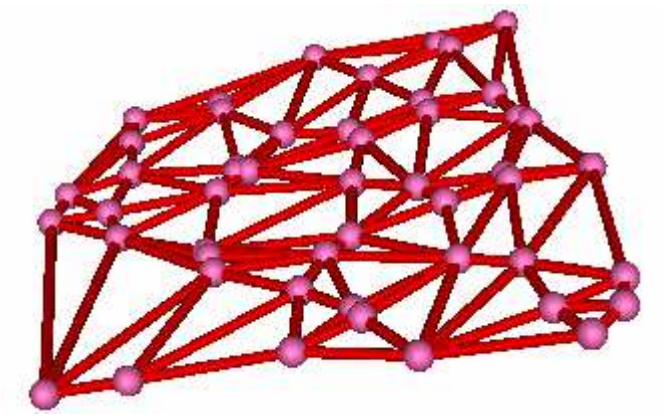
6.3.1 Delaunay 三角网

Delaunay 三角网在计算几何中被广泛使用，其最基本的一个应用是使用离散点数据构建网格数据，网格数据能被应用于多个方面，如利用网格数据进行地形可视化，在 VTK 中提供了二维、三维的 Delaunay 三角网创建方法。

- 二维 Delaunay 三角网

vtkDelauany2D 对象将 vtkPointSet(或其子类)数据类型作为输入，并且生成 vtkPolyData 数据类型作为输出，数据一般都是以三角网格的形式输出，如果将 Alpha 值设为非零值，生成的三角网格可能由三角形、线或顶点组成（Alpha 控制三角形输出的数量，只有在 Alpha 值设定范围内的三角形才被输出。

下面的示例程序，结果如下图所示：



该程序随机在 X-Y 平面上生成离散点，然后利用 vtkTubeFilter 过滤器根据生成的三角网建立管线（如图中红色部分），用 vtkGlyph3D 创建三角网中网格节点处的球体，如图中球体部分），示例代码（Examp64）：

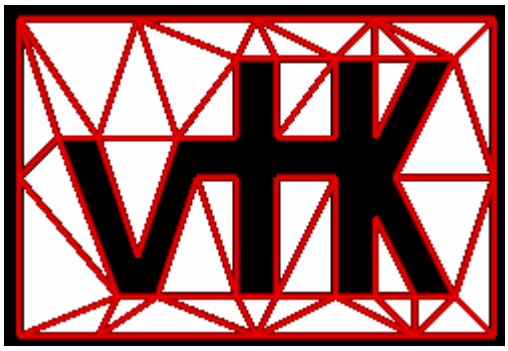
```
vtkMath *math = vtkMath::New();
//创建点数据集
vtkPoints *points = vtkPoints::New();
 points->SetNumberOfPoints(50);
   //设置存储的数据类型为 float
```

```
 points->SetDataType(10);
 float x,y;
 for(int i=0;i<50;i++){
     x=math->Random(0,1);//随机生成 x 坐标
     y=math->Random(0,1);//随机生成 y 坐标
     points->InsertPoint(i,x,y,0.0);
 }
//创建多边形对象
vtkPolyData *profile = vtkPolyData::New();
 profile->SetPoints(points);
//建立 Delaunay 三角网
vtkDelaunay2D *del = vtkDelaunay2D::New();
 del->SetInput(profile);
 del->SetAlpha(0);
 //设置空间点闭合的公差,公差值越小，闭合的越精确
 del->SetTolerance(0.0000001);
  vtkPolyDataMapper *mapMesh = vtkPolyDataMapper::New();
 mapMesh->SetInput((vtkPolyData *) del->GetOutput());
  vtkActor *meshActor = vtkActor::New();
 meshActor->SetMapper(mapMesh);
 meshActor->GetProperty()->SetColor(1.0 , 0.0 , 0.0);
//提取三角网的边，作为线
vtkExtractEdges *extract = vtkExtractEdges::New();
extract->SetInput((vtkDataSet *) del->GetOutput());
//生成管线
vtkTubeFilter *tubes = vtkTubeFilter::New();
tubes->SetInput((vtkPolyData *) extract->GetOutput());
//设定管线用多少个面来逼近
tubes->SetNumberOfSides(6);
tubes->SetRadius(0.01);
```

```
vtkPolyDataMapper *mapEdges = vtkPolyDataMapper::New();
mapEdges->SetInput((vtkPolyData *) tubes->GetOutput());
vtkActor *edgeActor = vtkActor::New();
edgeActor->SetMapper(mapEdges);
edgeActor->GetProperty()->SetColor(1 , 0 , 0);
//创建球体
vtkSphereSource *ball = vtkSphereSource::New();
 ball->SetCenter(0 , 0 , 0);
 ball->SetPhiResolution(12);
 ball->SetRadius(0.025);
 ball->SetThetaResolution(12);
//符号化处理，注意有两个输入
vtkGlyph3D *balls = vtkGlyph3D::New();
//输入需要符号化的数据
balls->SetInput((vtkDataSet *) del->GetOutput());
//设定符号形状
balls->SetSource(ball->GetOutput());
vtkPolyDataMapper *mapBalls = vtkPolyDataMapper::New();
mapBalls->SetInput((vtkPolyData *) balls->GetOutput());
vtkActor *ballActor = vtkActor::New();
ballActor->SetMapper(mapBalls);
```

公差用于确定两个点是否闭合，如果两个点之间的距离小于设定的公差值，则认为这两个点是闭合的，公差可以表示为输入点集边界形成四边形区域的对角线的一小部分。

通常情况下，vtkDelaunay2D 依据输入的离散点和圆原则构建 Delaunay 三角网，但是 vtkDelaunay2D 通过指定约束边和封闭多边形环也可以构建约束 Delaunay 三角网，下面的示例程序定义了一个约束 Delaunay 三角网，示例结构如下：



代码如下（Examp65）：

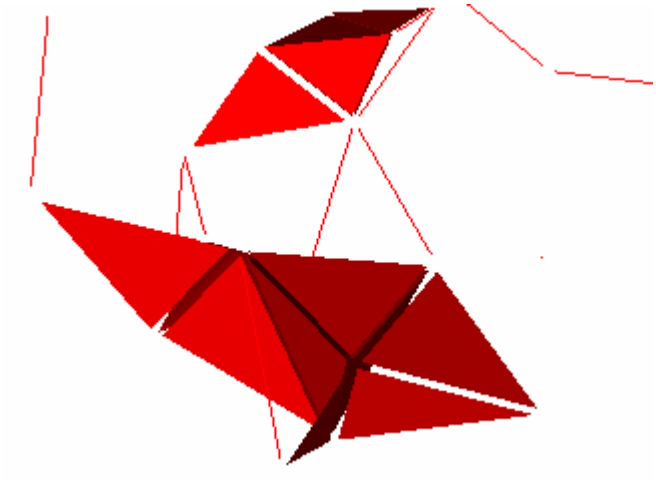
```
//存储离散点
vtkPoints *points = vtkPoints::New();
points->InsertPoint(0, 1, 4, 0);
points->InsertPoint(1, 3, 4, 0);
…
//构建约束的多边形封闭环
vtkCellArray *polys = vtkCellArray::New();
//第一个单元
polys->InsertNextCell(12);
…
//第二个单元
polys->InsertNextCell(28);
…
//设定多边形数据
vtkPolyData *polyData = vtkPolyData::New();
polyData->SetPoints(points);
polyData->SetPolys(polys);
//构建 Delaunay 三角网
vtkDelaunay2D *del = vtkDelaunay2D::New();
//设定离散点集构建三角网
del->SetInput(polyData);
//输入约束的封闭多边形环
del->SetSource(polyData);
```

```
//多边形数据映射器
vtkPolyDataMapper *mapMesh = vtkPolyDataMapper::New();
mapMesh->SetInputConnection(del->GetOutputPort());
//创建角色
vtkActor *meshActor = vtkActor::New();
meshActor->SetMapper(mapMesh);
//提取三角网的边，作为线
vtkExtractEdges *extract = vtkExtractEdges::New();
extract->SetInputConnection(del->GetOutputPort());
//建立管线
vtkTubeFilter *tubes = vtkTubeFilter::New();
tubes->SetInput(extract->GetOutput());
tubes->SetRadius(0.1);
tubes->SetNumberOfSides(6);
vtkPolyDataMapper *mapEdges = vtkPolyDataMapper::New();
mapEdges->SetInputConnection(tubes->GetOutputPort());
vtkActor *edgeActor = vtkActor::New();
edgeActor->SetMapper(mapEdges);
edgeActor->GetProperty()->SetColor(1.0,0.0,0.0);
```

在这个例子中，使用 vtkDelaunay2D 的 SetSource（）方法设定约束封闭多边形环，可以使用相似的方法，设定约束边。

- 三维 Delaunay 三角网

三维 Delaunay 三角网和二维的使用方式相似，唯一的区别是 vtkDelaunay3D 输出的数据类型是非结构化网格数据，示例程序结果如下：



程序代码如下（Examp66）：

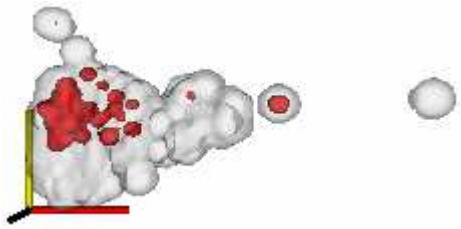
```
vtkMath *math = vtkMath::New();
    vtkPoints *points = vtkPoints::New();
    points->SetNumberOfPoints(25);
    points->SetDataType(10);
    //随机生成坐标点
    for(i=0;i<25;i++){
        x=math->Random(0,1);
        y=math->Random(0,1);
        z=math->Random(0,1);
        points->InsertPoint(i,x,y,z);
    }
    vtkPolyData *profile = vtkPolyData::New();
    profile->SetPoints(points);
    vtkDelaunay3D *del = vtkDelaunay3D::New();
    del->SetInput(profile);
    del->SetAlpha(0.2);
    del->SetOffset(2.5);
    del->SetTolerance(0.0001);
    del->SetBoundingTriangulation(0);
    vtkShrinkFilter *shrink = vtkShrinkFilter::New();
    shrink->SetInput((vtkDataSet *) del->GetOutput());
```

```
shrink->SetShrinkFactor(0.9);
vtkDataSetMapper *map = vtkDataSetMapper::New();
map->SetInput(shrink->GetOutput());
vtkActor *triangulation = vtkActor::New();
triangulation->SetMapper(map);
triangulation->GetProperty()->SetColor(1 , 0 , 0);
```

6.3.2 高斯抛雪球

对于无层次结构数据或者高维度数据，如空间任意一点的温度、多维财务数据等形式的数据，对于这样一些数据，最简单的可视化方法是对这些数据进行重新采样，生成体数据（vtkImageData），然后对体数据进行可视化。

下面的示例程序显示了如何可视化一个财务数据，该数据以文本文件的形式存储，共有 3188 条记录，每条记录包含着以下的信息：付贷款的时间（TIME_LATE）、每个月支付的贷款（MONTHLY_PAYMENT）、每月还的贷款数目（NONTHLY_PRINCIAL）、剩余的贷款（UNPAID_PRINCIPAL）、初始的贷款总额(LOAN_AMOUNT)、贷款利息（INTEREST_RATE）、每月贷款的收入（MONTHLY_INCOME），在这个例子中显示的是迟交贷款的人在总贷款人数中的比例，选择 MONTHLY_PAYMENT 做为 X 轴，INTEREST_RATE 做为 Y 轴，还有 LOAN_AMONT 做为 Z 轴，并且选择 TIME_LATE 做为参变量（也可以说，我们选取三个变量做为研究对象而忽略其它的因素），vtkGaussianSplatter 类被用来减少数据，用高斯分布函数将这些数据变为 vtkImageData 数据集，用 vtkContourFilter 类生成等面体，结果如下图所示：



程序代码（Examp67）：

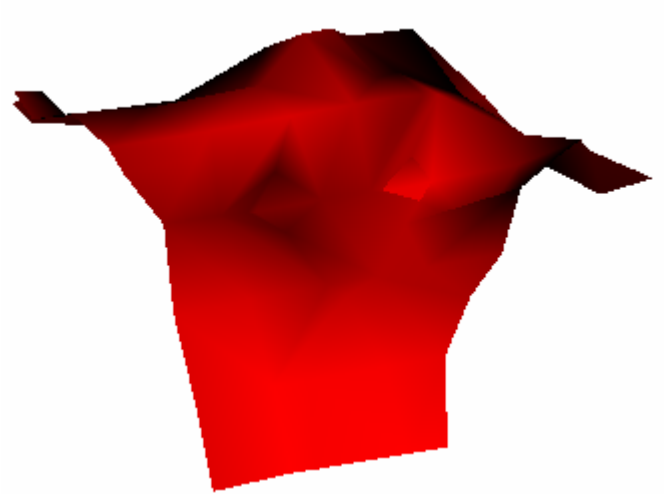
```
//读取数据，构建非结构化网格数据
vtkDataSet*dataSet=ReadFinancialData("financial.txt",
"MONTHLY_PAYMENT","INTEREST_RATE","LOAN_AMOUNT","TIME_LATE");
    //构建可视化流水线，非结构化数据集变换成体数据
    vtkGaussianSplatter *popSplatter=vtkGaussianSplatter::New();
```

```
popSplatter->SetInput(dataSet);
popSplatter->SetSampleDimensions(150,150,150);
popSplatter->SetRadius(0.05);
popSplatter->ScalarWarpingOff();
vtkContourFilter *popSurface = vtkContourFilter::New();
popSurface->SetInputConnection(popSplatter->GetOutputPort());
popSurface->SetValue(0,0.01);
vtkPolyDataMapper *popMapper = vtkPolyDataMapper::New();
popMapper->SetInputConnection(popSurface->GetOutputPort());
popMapper->ScalarVisibilityOff();
vtkActor *popActor = vtkActor::New();
popActor->SetMapper(popMapper);
popActor->GetProperty()->SetOpacity(0.3);
popActor->GetProperty()->SetColor(.9,.9,.9);
// construct pipeline for delinquent population
vtkGaussianSplatter *lateSplatter = vtkGaussianSplatter::New();
lateSplatter->SetInput(dataSet);
lateSplatter->SetSampleDimensions(50,50,50);
lateSplatter->SetRadius(0.01);
//设置一个系数，使用数据集的标量值，乘以高斯函数，计算体素的值
lateSplatter->SetScaleFactor(100);
vtkContourFilter *lateSurface = vtkContourFilter::New();
lateSurface->SetInputConnection(lateSplatter->GetOutputPort());
lateSurface->SetValue(0,0.02);
vtkPolyDataMapper *lateMapper = vtkPolyDataMapper::New();
lateMapper->SetInputConnection(lateSurface->GetOutputPort());
lateMapper->ScalarVisibilityOff();
vtkActor *lateActor = vtkActor::New();
lateActor->SetMapper(lateMapper);
lateActor->GetProperty()->SetColor(1.0,0.0,0.0);
```

对一个体进行重采样还有另外一个过滤器 vtkShepardMethod 类，可以使用这个类来修改上面的例子，看一下显示的结果。

6.3.3 杂乱点集构建表面

在计算机图形应用中，表面经常用三维无序的点表示，用这些点重构表面在计算效率和算法设计上都具有挑战性，先前描述的 Delaunay 三角网和高斯抛雪球方法提供了不同级别重构表面的方法，为了提高表面重构的效率，VTK 为杂乱点集构建表面设计了专门的类 vtkSurfaceReconStructionFilter，示例程序结果如下图：



示例程序代码（Examp68）：

```
 vtkMath *math = vtkMath::New();
vtkPoints *points = vtkPoints::New();
points->SetNumberOfPoints(25);
points->SetDataType(10);
//随机生成坐标点
for(i=0;i<25;i++){
    x=math->Random(0,1);
    y=math->Random(0,1);
    z=math->Random(0,1);
    points->InsertPoint(i,x,y,z);
 }
 vtkPolyData *profile = vtkPolyData::New();
```

```
profile->SetPoints(points);
//重构三维表面
vtkSurfaceReconstructionFilter *del=vtkSurfaceReconstructionFilter::New();
del->SetInput(profile);
//提取表面
vtkContourFilter *popSurface = vtkContourFilter::New();
popSurface->SetInputConnection(del->GetOutputPort());
popSurface->SetValue(0,0.0);
vtkPolyDataMapper *map=vtkPolyDataMapper::New();
map->SetInput(popSurface->GetOutput());
vtkActor *triangulation = vtkActor::New();
triangulation->SetMapper(map);
triangulation->GetProperty()->SetColor(1 , 0 , 0);
```

该程序随机生成杂乱的离散点集，构建多边形数据，然后由 vtkSurfaceReconStructionFilter 类对数据处理后生成 vtkImageData 类型的数据，提取设定值的轮廓面，完成表面重构。

7 与视窗系统交互

在使用 VTK 的过程中，你可能要修改 VTK 提供的缺省的交互方式，或者为了更好的和系统进行交互，你可能需要在 VTK 应用程序中建立用户界面（GUI），本章主要介绍如何利用 VTK 提供的交互接口，建立交互应用程序。

7.1 vtkRenderWindow 交互类型

vtkRenderWindowInteractor 类捕获绘制窗口的鼠标事件和键盘事件，紧接着分发这些事件到其它的类，因此，要在 VTK 添加新的的交互方式，需要从 vtkInteractorStyle 类中派生新的类，例如，vtkInteractorStyleTrackball 类提供了轨迹球的交互方式。vtkInteractorStyleJoystickActor 或者 vtkInteractorStyleJoystickCamera 实现了操纵杆的交互方式。

事件截取的工作过程如下，调用 vtkRenderWindowInteractor::Start()方法，允许截

取绘制窗口发生的事件，当截取到绘制窗口的事件后，将事件转交给 vtkRenderWindowInteractor::InteractorStyle 实例处理。

下面列出了 VTK 提供的主要的交互类型类：

- vtkInteractorStyleJoystickActor

实现了操纵杆交互方式，对角色进行操纵。

- vtkInteractorStyleJoystickCamera

实现了操纵杆交互方式，对相机进行操纵。

- vtkInteractorStyleTrackballActor

实现了跟踪球交互方式，对角色进行操纵。

- vtkInteractorStyleTrackballCamera

实现了跟踪球交互方式，对相机进行操纵。

- vtkInteractorStyleSwitch

在操纵杆模式和跟踪球模式之间切换。

- vtkInteractorStyleTrackball

一个先前版本的跟踪球操作模式的类，对相机和角色进行操纵。

- vtkInteractorStyleFlight

实现飞行交互模式。

如果 VTK 提供的交互方式不能满足我们的要求，想要创建新的交互方式，必须从 vtkInteractorStyle 类派生新的类，并且重载 vtkInteractorStyle 类的虚函数，然后将新建的交互类用 vtkRenderWindowInteractor::SetInteractorStyle() 方法和 vtkRenderWindowInteractor 类相关联。

7.2 交互方针

VTK 从灵活性和便捷性考虑，在设计时将功能实现和用户接口相互分离，为了更加容易的使用用户接口，提高应用程序交互的灵活性，VTK 提供了用户方法接口，用户可以设定自定义的方法，当事件发生时，调用用户自定义的方法（有关内容可参考第一章创建应用程序的内容）。

下面列出了一些经常用到的事件：

所有的过滤器（vtkProcessObject 的子类）都调用的事件包括：StartEvent、EndEvent、

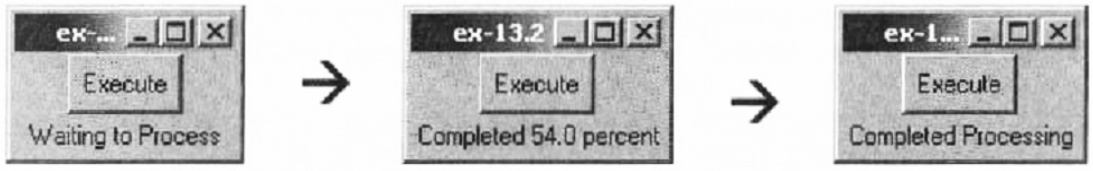
ProgressEvent。

vtkRenderWindow 类在绘制是调用的事件包括：AbortCheckEvent。

vtkActor、vtkVolume、vtkPicker 类在拾取时，调用的事件包括：PickEvent、StartPickEvent（仅对 vtkPicker 类适用）、EndPickEvent（仅对 vtkPicker 类适用）。

vtkRenderWindowInteractor 类调用的事件包括：StartPickEvent、EndPickEvent、UserEvent、ExitEvent。

StartEvent、EndEvent、ProgressEvent 事件常被用来将应用程序的工作状态反馈给用户，所有的过滤器都可以调用已经被定义的开始事件和结束事件，但是，过程事件只能被图像过滤器、部分阅读器和部分可视化过滤器调用，调用 AbortCheckEvent 事件可以中断当前的窗口绘制，PickEvent 事件是个虚方法，当我们自己定义交互方式时，可以重载该方法，为了帮助你理解如何使用事件，下面列举两个示例程序，第一个示例程序定义了一个方法，用于捕获 ProgressEvent 事件，显示 vtkImageShrink3D 过滤器执行的情况，然后捕获 EndEvent 事件，在用户界面上更新显示执行的结果。



第二个示例使用 AbortCheckEvent 事件终止窗口的长时间绘制。

7.3 在 Window 系统/VC++中使用 VTK 进行交互

在第一章中，我们已经介绍了 VTK 在 Windows 系统中的一些基本交互功能（参考第一章创建应用程序小节），我们也可以使用以下两种方法开发基于 MFC 的应用程序，第一种方法是在 MFC 应用程序内部使用 VTK，第二种方法是使用 VTK 提供的 vtkMFCView、vtkMFCDocument 等类创建基于 MFC 应用程序，相关的示例程序请参考示例程序 Examp71 和 Examp72。

8 VTK 对象说明

8.1 对象结构图

本章主要描述 VTK 对象结构图。

8.1.1 基础对象

基础对象如图 8-1 所示，这些对象在 VTK 中实核心对象，提供了基础服务。

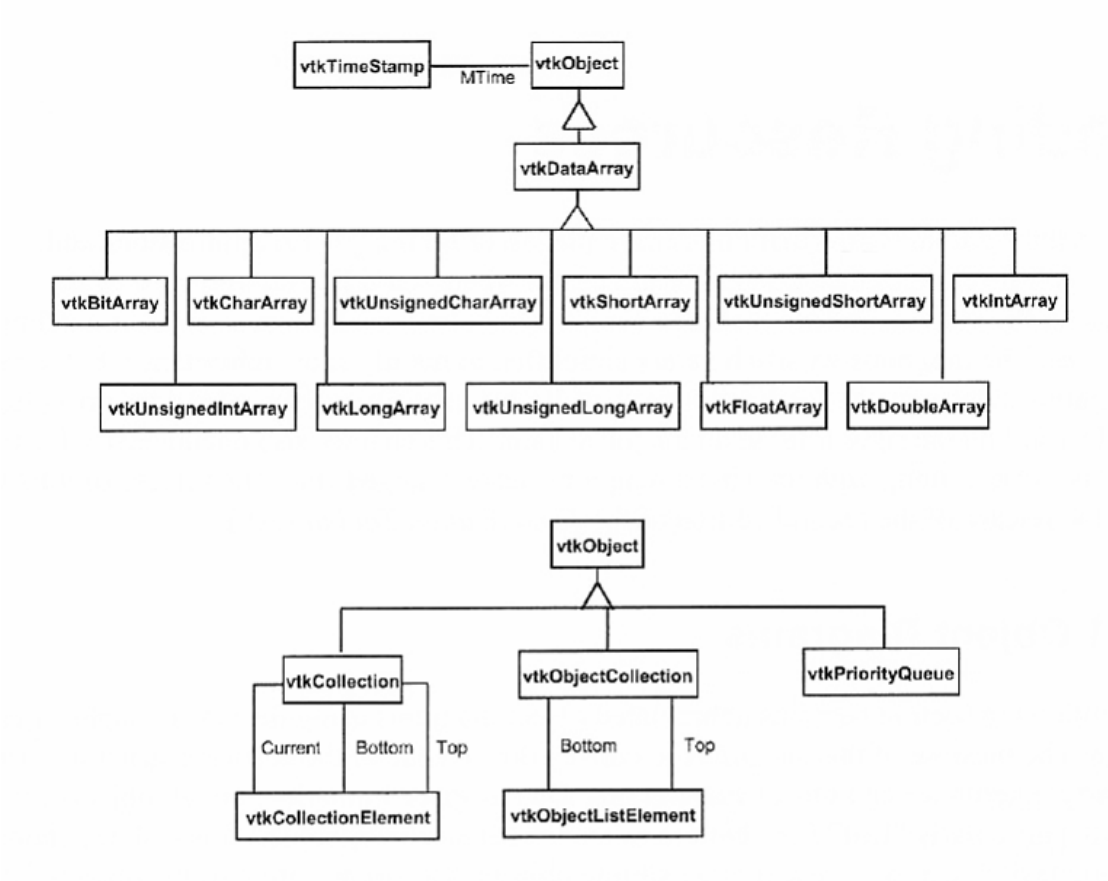


图 8-1 基础对象结构图

8.1.2 单元对象

在 VTK 中提供了 14 个具体的单元对象，对象图如 8-2 所示，其中 vtkGenericCell 类可以表达任何的单元类型，vtkEmptyCell 类用于表达当前存在的对象被删除或者为空。

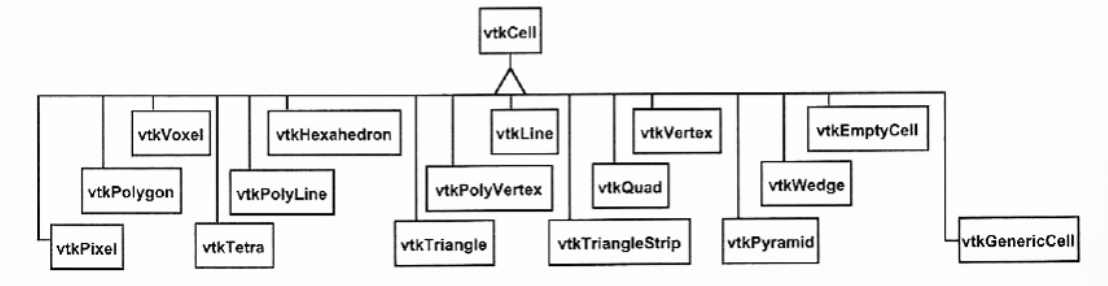


图 8-2 单元对象结构图

8.1.3 数据集对象

当前提供了 5 个具体的数据集对象，如图 8-3 所示，离散的点数据集可以用 vtkPointSet 的任一子类表达，vtkStructuredGrid 类表达结构化网格数据，vtkImageData 类表达二维图像和三维体数据。

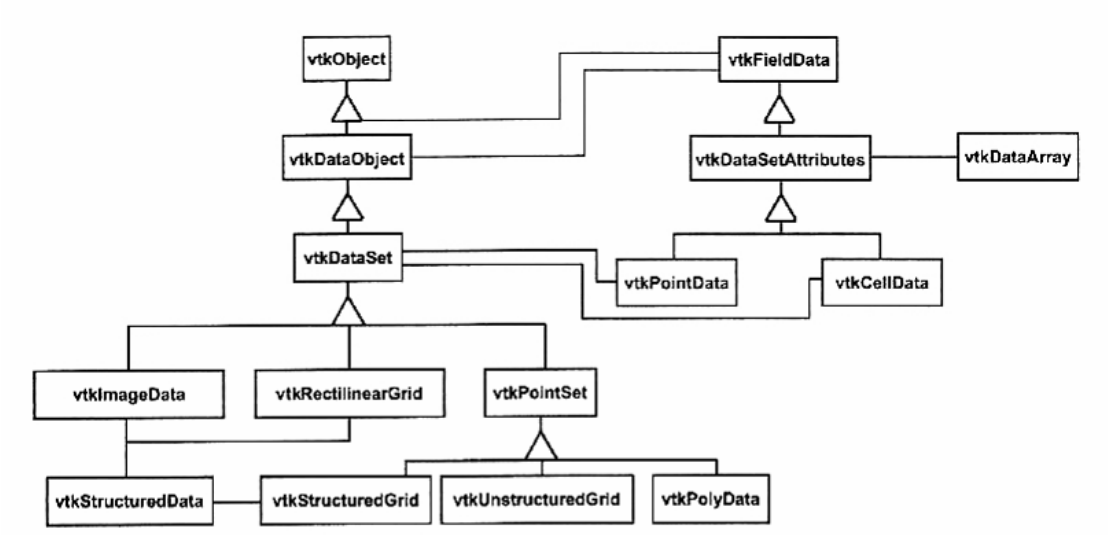


图 8-3 数据集对象结构图

8.1.4 可视化流水线对象

可视化流水线对象如图 8-4 所示，是 VTK 的核心对象。

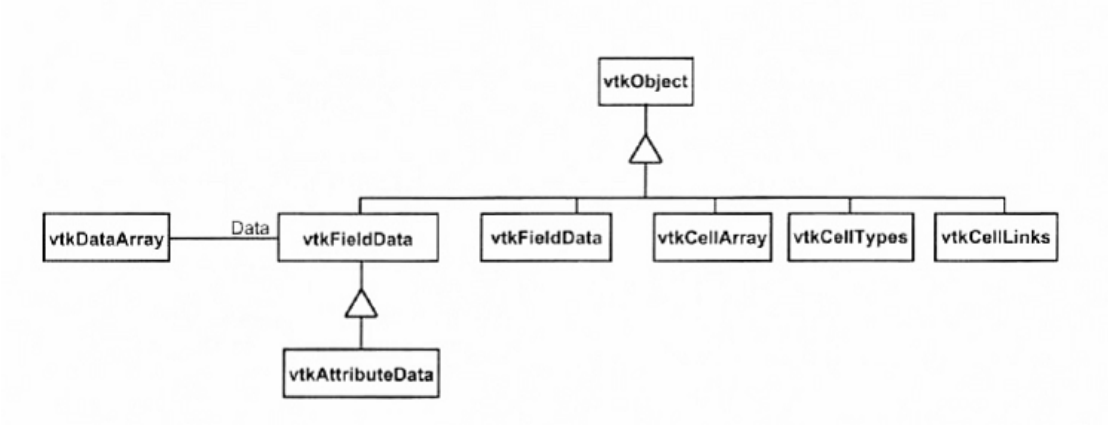


图 8-4 可视化流水线对象结构图

8.1.5 源对象

如下图：

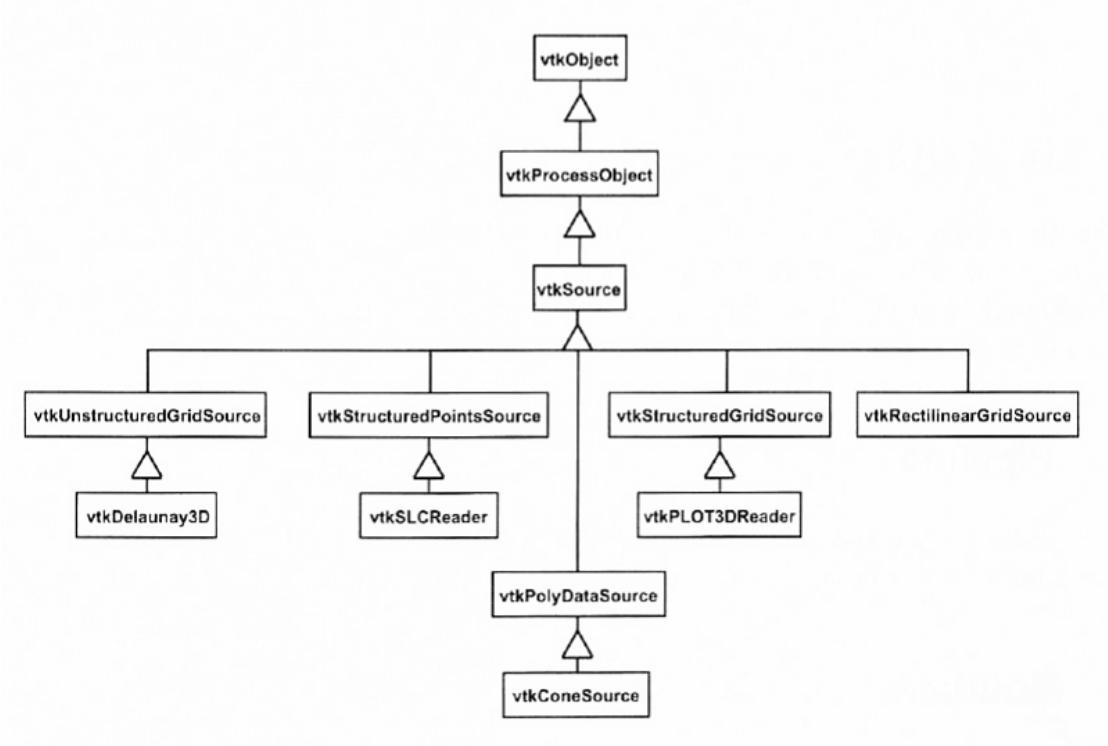


图 8-5 源对象结构图

8.1.6　过滤器

对象图如 8-6 所示：

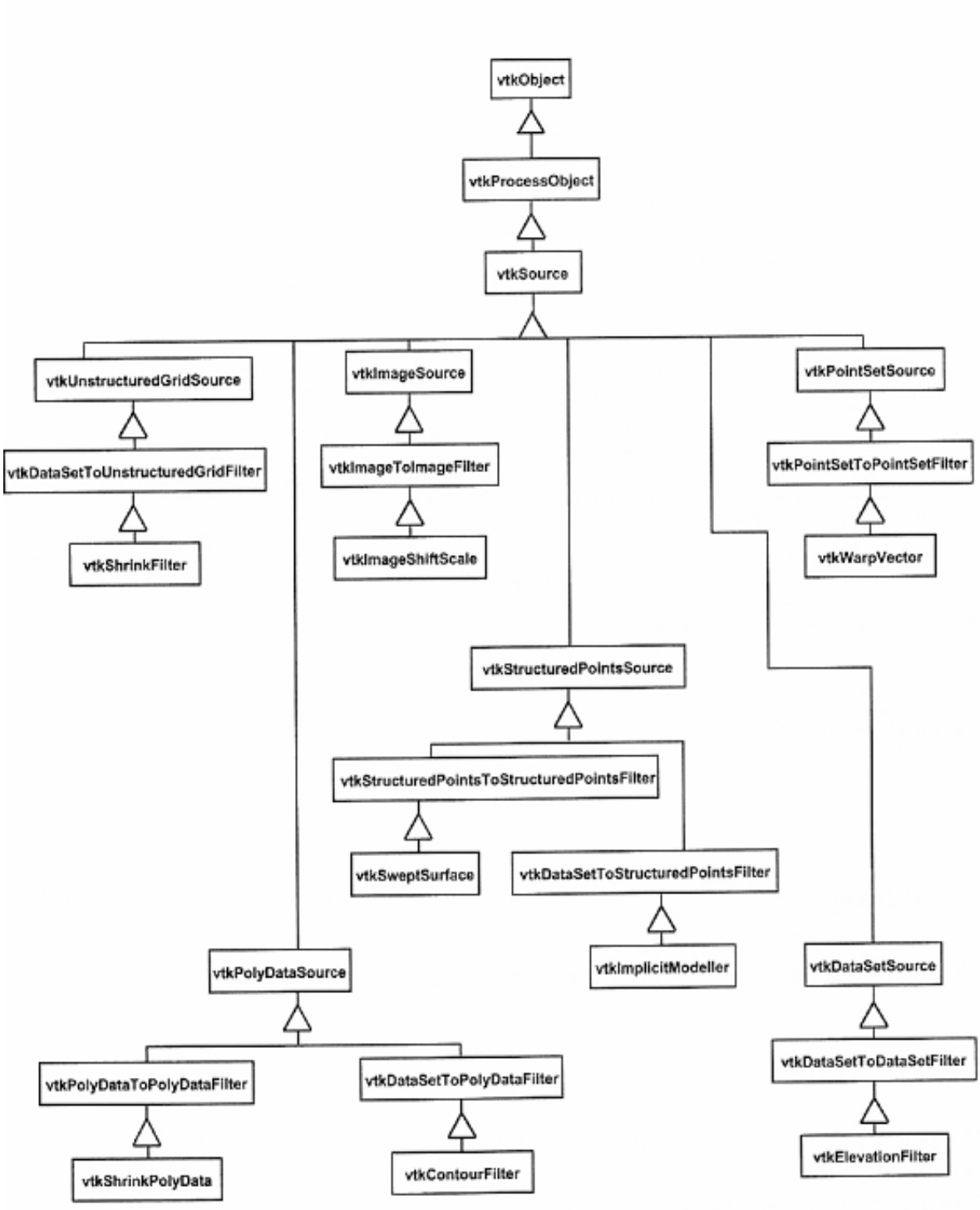


图 8-6 过滤器对象结构图

8.1.7 映射器

有两种基本类型，一种是图形映射器，将可视化数据映射到图形系统，另外一种是复写器，将可视化数据写成 VTK 数据文件形式，对象如图 8-7 所示：

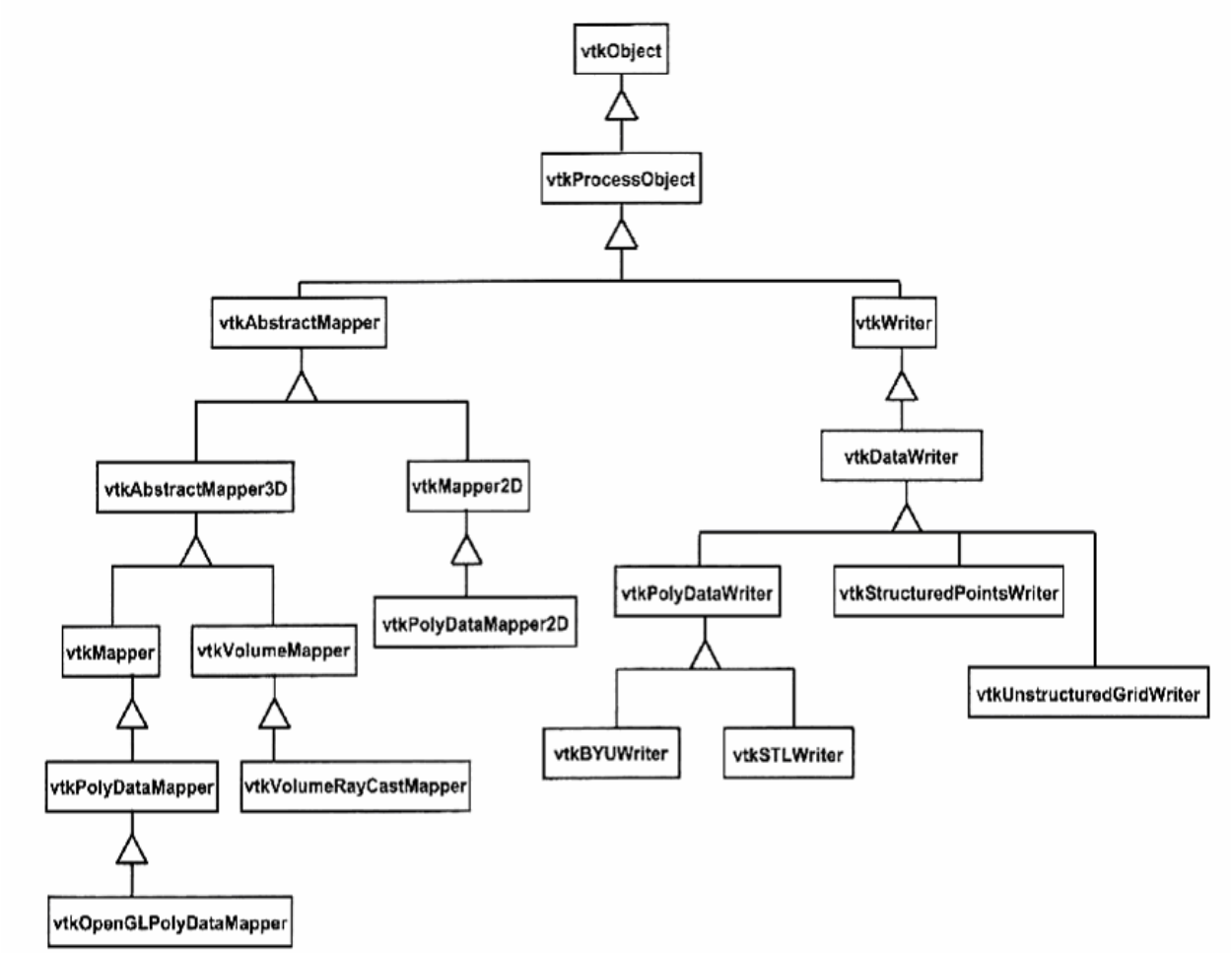


图 8-7 映射器对象结构图

8.1.8 图形对象

图形对象对可视化数据进行绘制，它和 VTK 中的其它对象关联，对象图如图 8-8 所示：

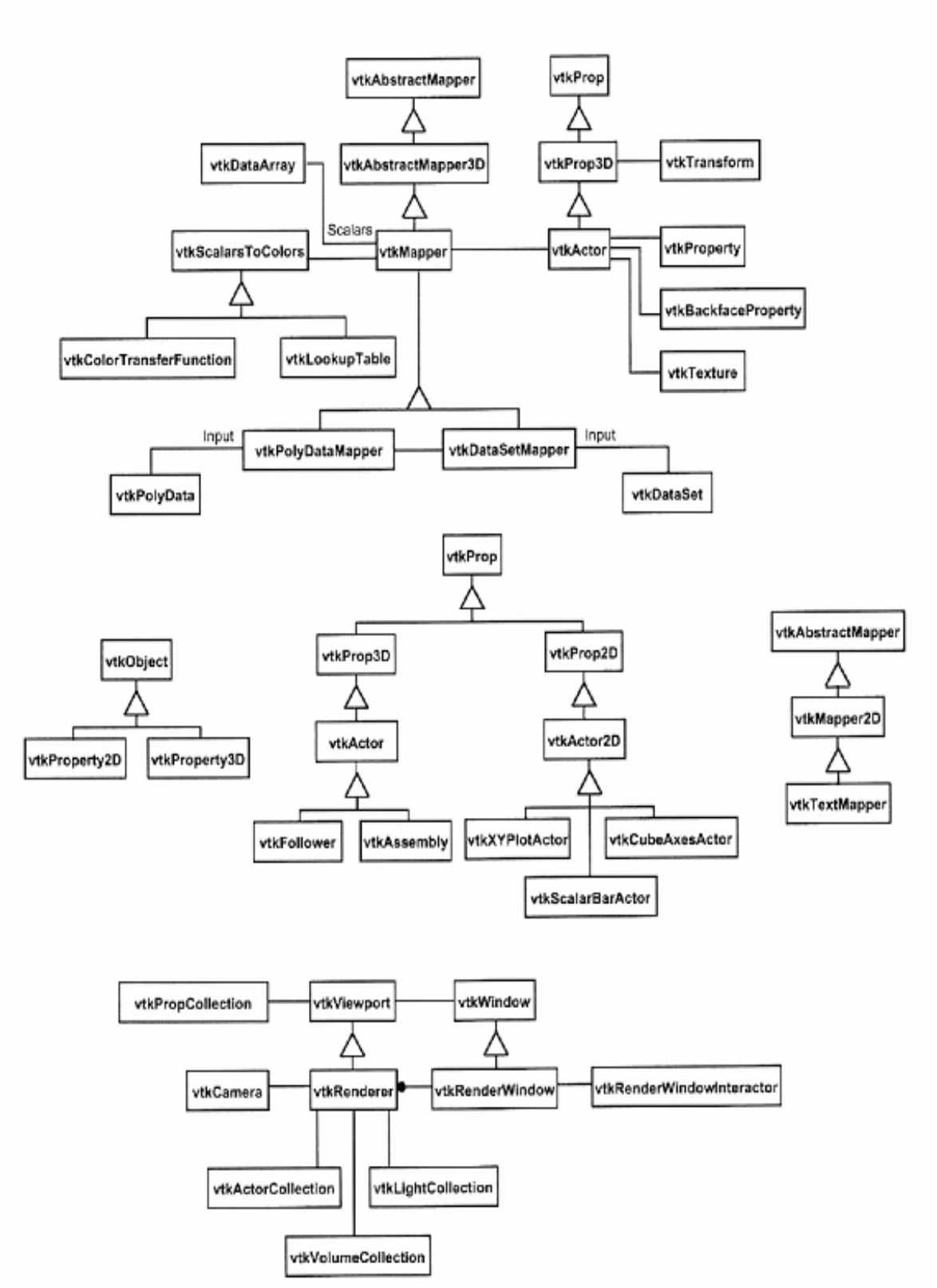


图 8-8 图形对象结构图

8.1.9 体绘制

体绘制对象的类层次图如图 8-9 所示，需要注意的是：体绘制和表面绘制都使用 VTK

的图形系统进行绘制。

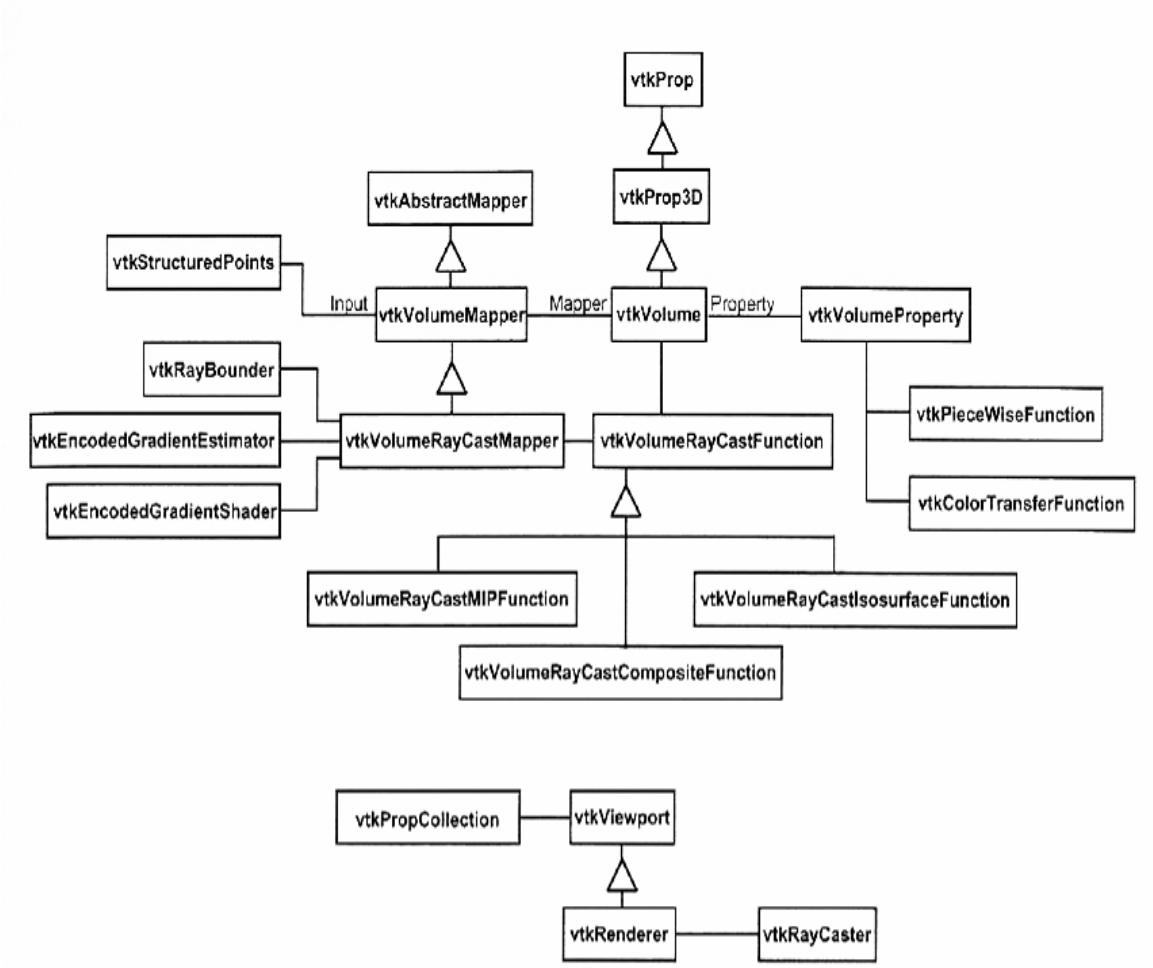


图 8-9 体绘制对象结构图

8.1.10 图像处理对象

图像处理对象与图形管线通过结构化点数据集相结合，也有可能由绘制器通过 vtkRendererSource 对象获取图像，然后送到图像流水线，图像处理对象如图 8-10 所示：

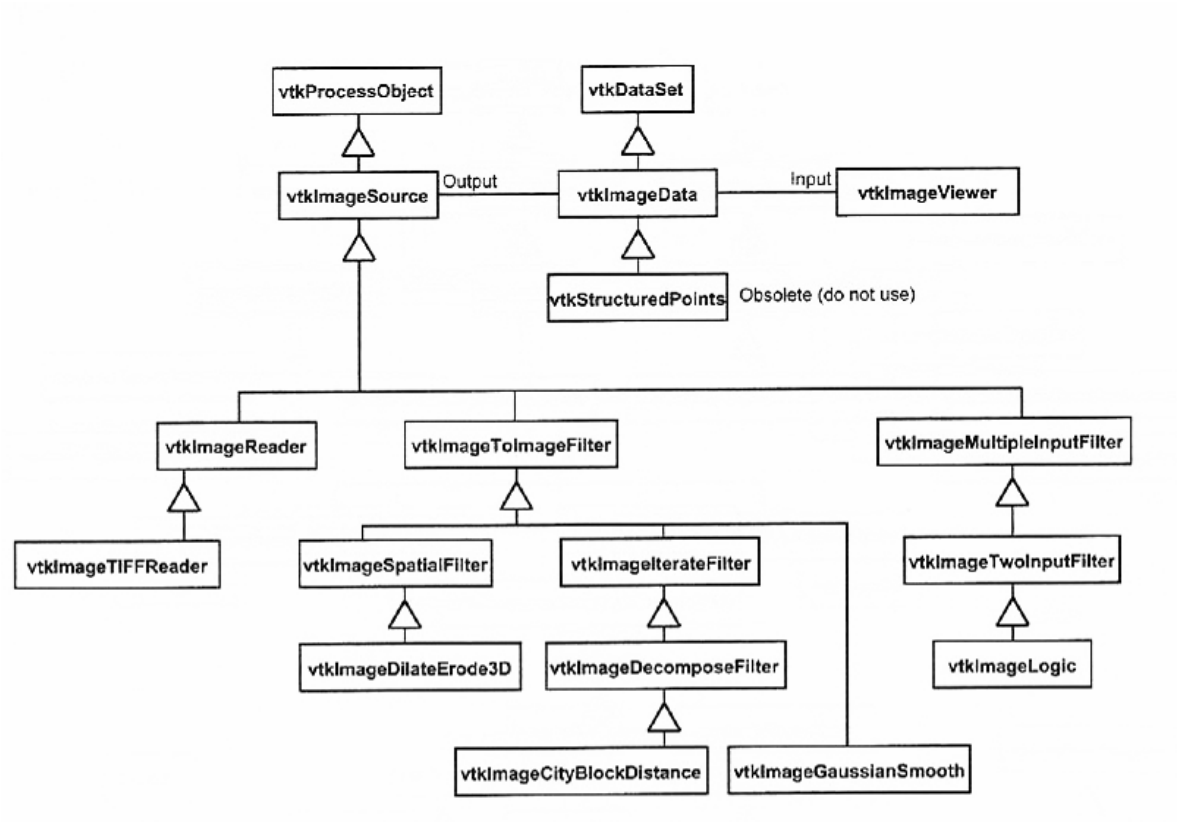


图 8-10 图像处理对象结构图

8.1.11 OpenGL 绘制对象

VTK 中有多个绘制库，OpenGL 对象结构图显示了这些库，如图 8-11 所示：

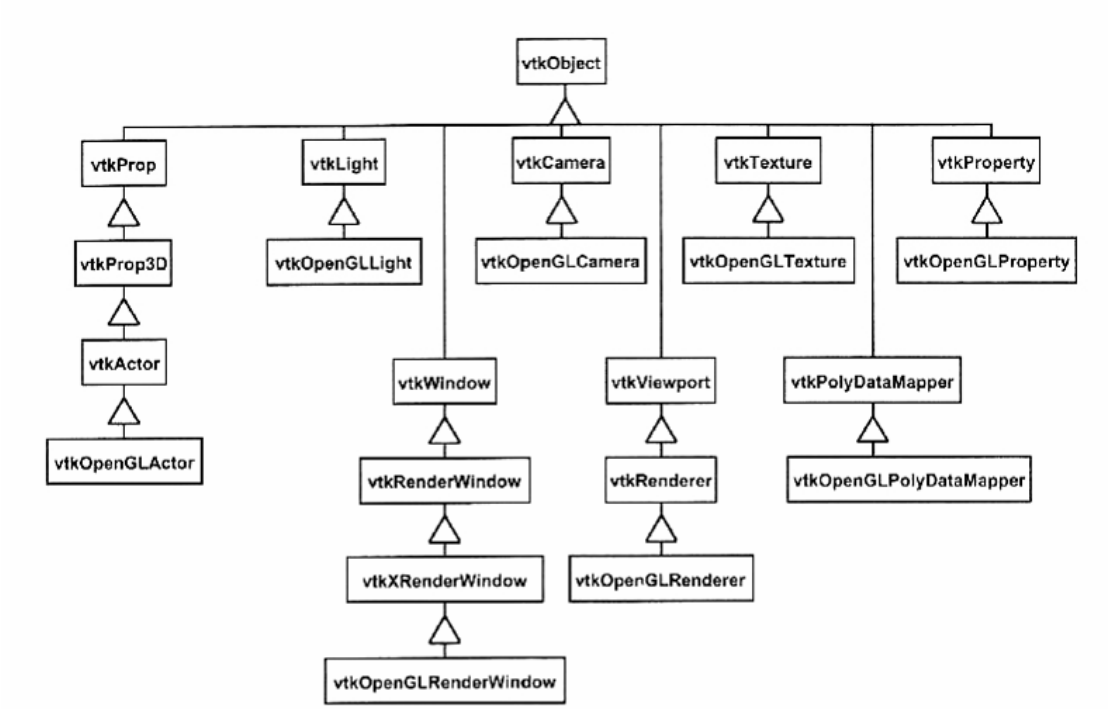


图 14-11OpenGL 绘制对象结构图

8.1.12 拾取对象

拾取对象结构如图 8-12 所示，vtkPropPicker 和 vtkWorldPointPicker 是最快的拾取器，所有拾取器都会为在绘制窗口选中的点返回一个全局的 x,y,z 值，vtkCellPicker 使用光线投射方法获取单元信息，vtkPointPicker 返回点的 ID，vtkPropPicker 标识被拾取的物体同时返回物体的坐标。

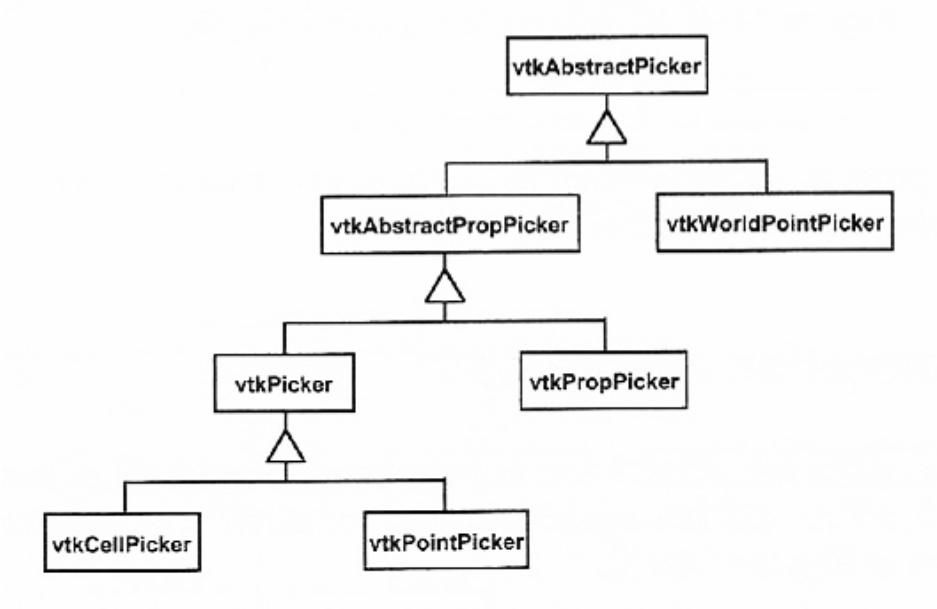


图 8-12 拾取对象结构图

8.1.13 变换对象层次图

VTK 提供了强大的变换对象，它支持线性的、非线性的、仿射的和同源的变换，变换对象层次图如图 8-13 所示：

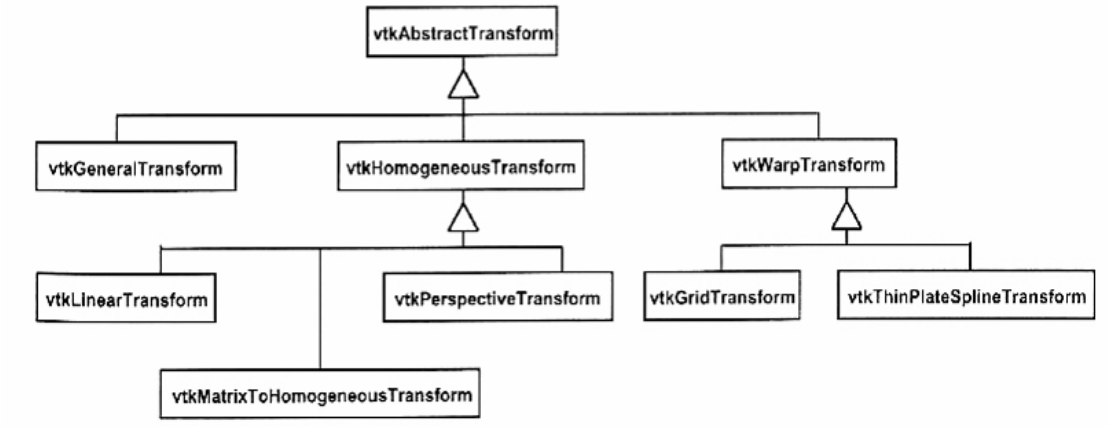


图 8-13 变换对象类层次图

8.2 过滤器

本小节主要描述 VTK 提供的可视化过滤器、图像过滤器、源对象。

8.2.1 源对象

源对象是可视化流水线的起始节点，下面列出了除了 Reader 源对象以外的所有源对象描述。

1) vtkBooleanTexture：根据隐函数创建二维纹理地图。
2) vtkConeSource：生成圆锥体的多边形表达。
3) vtkCubeSource：生成立方体的多边形表达。
4) vtkCursor3D：为给定的边界盒和焦点生成一个三维指针，这个指针是用线表示的。
5) vtkCylinderSource：生成圆柱体的多边形表达。
6) vtkDiskSource：生成圆锥体的多边形表达。
7) vtkEarthSource：生成地球的多边形表达。
8) vtkImageCanvasSource2D：用现有图形绘制图像。
9) vtkImageEllipsoidSource：创建一个椭圆分布的图像。
10) vtkImageGaussianSource：创建一个正态分布的图像。
11) vtkImageMandelbrotSource：创建一个曼德布洛集的图像。
12) vtkNoiseSource：创建一个干扰图像。
13) vtkImageSinusoidSource：创建一个正弦值图像。
14) vtkLineSource：生成圆锥体的多边形表达。
15) vtkOutlineSource：生成圆锥体的多边形表达。
16) vtkPlaneSource：生成圆锥体的多边形表达。
17) vtkPointLoad：
18) vtkPointSetSource：创建一个包含正弦图像值的图像。
19) vtkPointSource：生成圆锥体的多边形表达。
20) vtkProgrammableDataObjectSource：在运行中可以读取或生成一个 vtkDataObject 对象的过滤器。

21) vtkProgrammableSource：在运行中可以读取或生成任意类型数据的过滤器。

22) vtkRendererSource：一个图像过滤器，可以将绘制器或者绘制窗口放入图像管线。

23) vtkSampleFunction：为一个体抽取隐函数。

24) vtkSphereSource：生成一个圆锥体的多边形表达。

25) vtkSuperquadricSource：生成超二次曲面的多边形表达。

26) vtkTextSource：创建一个多边形表达文本。

27) vtkTexturedSphereSource：创建一个关联了纹理坐标的球体的多边形表达。

28) vtkTriangularTexture：生成一个二维的三角形纹理地图。

29) vtkVectorText：创建文本的多边形表达。

30) vtkVedioSource：抓取视频信号作为一个图像。

8.2.2 图像过滤器

在这里所描述的过滤器,将 vtkImageData 数据集类型(或者 vtkStructuredPoints)作为输入，并且生成和输入类型相同的数据集类型作为输出。

1) vtkClipVolume：用隐函数建立裁减平面，对体进行裁减，生成四面体网格。

2) vtkCompositeFilter：将结构化点集组合成单一的数据集。

3) vtkDividingCubes：生成一个云点等值面。

4) vtkExtractVOI：提取体中感兴趣的区域或提取子体。

5) vtkImageAccumulate：生成输入图像的柱状图。

6) vtkImageAnisotropicDiffusion2D：迭代地运用 2D 扩散过滤器。

7) vtkImageAnisotropicDiffusion3D：迭代地运用 3D 扩散过滤器。

8) vtkImageAppend：将多个输入图像合并成一个输出图像。

9) vtkImageAppendComponents：合并两个输入图像的组件。

10) vtkImageBlend：根据每个输入的未知值和（或）不透明场景混和若干图像。

11) vtkImageButterworthHighPass：运用高频率域通过过滤器。

12) tkImageButterworthLowPass：运用低频率域通过过滤器。

13) vtkImageCachFilter：为避免管线重复执行，将图像进行隐藏起来备用。

14) vtkImageCanvasSource2D：基础图像显示，原始的绘图功能。

15) vtkImageCast：将输入图像换算成特定的输入类型。

16) vtkImageCityBlockDistance：根据城市街区之间的比例绘制一个相应比例的地图。
17) vtkImageClip：缩小输入图像的尺寸。
18) vtkImageComposite：使用像素数据或 Z 轴缓冲器数据合成复合图像。
19) vtkImageConstantPad：为输入图像添加一个常量。
20) vtkImageContinuousDilate3D：计算椭圆邻域的最大值。
21) vtkImageContinuousErode3D：计算椭圆邻域的最小值。
22) vtkImageCorrelation：创建与输入的两个图像相互关联的图像。
23) vtkImageCursor3D：为输入图像添加指针。
24) vtkImageDataStreamer：开启图像数据流。
25) vtkImageDataToPolyDataFilter：将图像数据转换成多边形数据。
26) vtkImageDifference：为两个图像生成不同的图像或误差值。
27) vtkImageDilateErode3D：在边界线上完成扩大或是缩减。
28) vtkImageDivergence：创建一个标量场来表达输入矢量场的变化速率。
29) vtkImageDotProduct：由两个矢量图创建一个点乘图像。
30) vtkImageEuclideanToPolar：将二维欧几里得坐标转换成极坐标。
31) vtkImageExtractComponents：提取输入图像组件的子集。
32) vtkImageFFT：完成快速傅氏变换。
33) vtkImageFlip：将图像沿指定的轴进行旋转。
34) vtkImageFourierCenter：从源点到中心变换零位频率。
35) vtkImageGaussionSmooth：完成一维、二维、三维的高斯旋转。
36) vtkImageGradient：计算图像的梯度向量。
37) vtkImageGradientMagnitude：计算梯度向量的大小。
38) vtkImageHSVToRGB：将 HSV 色彩模式转换成 RGB 模式。
39) vtkImageHybridMedian2D：保存线或角时执行中值过滤器。
40) vtkImageIdealHighPass：执行一个简单的频域高行过滤器。
41) vtkImgaeIdealLowPass：执行一个简单的频域低行过滤器。
42) vtkImageIslandRemoval2D：从图像中移除小的群集器。
43) vtkImageLaplacian：计算拉普拉斯算子。
44) vtkImageLogrithmicScale：每个像素点执行日志函数。
45) vtkImageLogic：完成逻辑运算：与，或,异或，与非，或非，非。

46) vtkImageLuminance：计算 RGB 图像的亮度。

47) vtkImageMagnify：按整数比例放大图像。

48) vtkImageMagnitude：计算图像组件的量级。

49) vtkImageMapToColors：通过检索表绘制图像。

50) vtkImageMarchingCubes：描述立方体的移动。

51) vtkImageMask：为图像提供掩码。

52) vtkImageMaskBits：为图像的组件提供位掩码。

53) vtkImageMathematics：提供一或两个图像的基本数学运算。

54) vtkImageMedian3D：计算矩形领域的中值过滤器。

55) vtkImageMirrorPad：用一个镜像图像衬托输入的图像。

56) vtkImageNonMaximumSuppression：执行非最大限度压制。

57) vtkImageNormalize：将图像的标量组件标准化。

58) vtkImageOpenClose3D：执行扩大或缩减操作。

59) vtkImagePermute：改变图像的轴线的序列。

60) vtkImageQuantizeRGBToIndex：将 RGB 图像转换为一个索引图像和一个检索表。

61) vtkImageRange3D：计算椭圆领域的范围。

62) vtkImageResample：放大或缩小重采样的图像。

63) vtkImageRFFT：执行反向快速傅里叶变换。

64) vtkImageRGBToHSV：将 RGB 模式的组件转换为 HSV 模式。

65) vtkImageReslice：体沿指定的轴做纯旋转变换。

66) vtkImageSeedConnectivity：评价与用户提供种子的连通性。

67) vtkImageShiftScale：在输入图像上执行变换和按比例绘制。

68) vtkImageShrink3D：通过在均匀网格上的二次抽样缩小图像。

69) vtkImageSkeleton2D：执行二维的构架操作。

70) vtkImageSobel2D：使用边缘算子函数计算图像的矢量场。

71) vtkImageSobel3D：使用边缘算子函数计算体矢量场。

72) vtkImageThreshold：执行二元或连续的阀值转换。

73) vtkImageVariance3D：计算椭圆领域内的差异。

74) vtkImageWrapPad：用象素索引的模运算修饰图像。

75) vtkLinkEdgels：连接边缘线形成数字曲线。

76) vtkMarchingCubes：高性能等值线绘制算法。

77) vtkMarchingSquares：二维高效等值线绘制算法

78) vtkRecursiveDividingCubes：生成云点等值线。

79) vtkStructuredPointsGeometryFilter：提取几何数据作为 vtkPolyData 数据。

80) vtkSweptSurface：生成移动部分扫描曲面。

81) vtkSynchronizedTemplates2D：二维高效等值线绘制算法。

82) vtkSynchronizedTemplates3D：三维高效等值线绘制算法。

8.2.3 可视化过滤器

下面按输入的数据类型来分类描述可视化过滤器。

- 处理 vtkDataSet 数据集的过滤器

1) vtkAppendFilter：将一个或多个数据集附加到非结构化网格数据中。

2) vtkAsynchronousBuffer：允许可视化流水线异步执行。

3) vtkAttributeDataToFieldDataFilter：将属性数据变换成场数据。

4) vtkBrownianPoints：给点集赋予一个随机的矢量值。

5) vtkCastToConcrete：将一个抽象的数据类型转换成一个具体的类型。

6) vtkCellCenters：生成一个标识单元核心的点。

7) vtkCellDataToPointData：将单元数据转换成点数据。

8) vtkCellDerivatives：计算标量或矢量的导数。

9) vtkClipDataSet：使用隐函数建立的平面或曲面对数据集进行裁减。

10) vtkConnectivityFilter：提取相互连通的单元构成非结构化网格数据。

11) vtkContourFilter：生成等值面。

12) vtkCutter：对 n 维的数据集进行剪切，生成 n-1 维的数据集。

13) vtkDashedStreamLine：生成一条流线，用虚线表示时间的变化。

14) vtkDataSetToDataObjectFilter：将一个子类数据集对象转换成父类对象。

15) vtkDicer：根据空间的间隔生成一个数值。

16) vtkEdgePoints：沿着等值面断面的边缘生成一个点。

17) vtkElevationFilter：沿着投影的方向生成一个标量值。

18) vtkExtractEdges：提取数据集的轮廓，生成一条轮廓线。

19) vtkExtractGeometry：提取完全位于指定的隐函数内部或外部的数据集单元。
20) vtkExtractTensorComponents：提取张量、矢量、法线或纹理坐标的分量作为标量。
21) vtkFieldDataToAttributeDataFilter：将场数据转换为数据集的点或单元的属性数据。
22) vtkGaussianSplatter：用高斯正态分布建立标量常场体数据。
23) vtkGeometryFilter：从数据集中提取几何表面或将数据集转换成多边形数据对象(vtkPolyData)。
24) vtkGlyph2D：对数据对象的变换（平移、旋转、缩放）被限定在 X-Y 平面上。
25) vtkGlyph3D：将多边形数据赋予输入数据的每个点。
26) vtkGraphLayout：将无序的曲线网进行排列。
27) vtkHedgeHog：将矢量场中的每个都用方向线表示。
28) vtkHyperStreamline：利用张量数据生成流管，根据特征向量进行弯曲。
29) vtkIdFilter：生成一个整型 ID 值(适用于图形显示)。
30) vtkImplicitModeller：生成一个距离输入几何体距离的距离场。
31) vtkImplicitTextureCoords：用隐函数创建纹理坐标。
32) vtkInterPolateDataSetAttributes：在两个数据集之间对属性数据进行插值。
33) vtkMaskPoints：选择输入点集的子集。
34) vtkMergeDataObjectFilter：将数据集合并生成新的数据集。
35) vtkMergeFilter：将不同数据集的不同部分合并到一起，生成一个新的数据集。
36) vtkOBBDicer：将数据集划分为若干部分。
37) vtkOutlineFilter：创建数据集的外形。
38) vtkPointDataToCellData：把点属性数据转换为单元属性数据。
39) vtkProbeFilter：对数据集进行探查或重采样。
40) vtkProgrammableAttributeDataFilter：序运行时，可动态处理属性数据。
41) vtkProgrammableFilter：可编程过滤器。
42) vtkProgrammableGlyphFilter：可对任意数数据值符号化的可编程过滤器。
43) vtkProjectedTexture：生成一个可投影到任意表面的纹理坐标。
44) vtkSelectVisiblePoints：设置选择的点集是否可见。
45) vtkShepardMethod：对点集重新采样，生成体数据。

46) vtkShrinkFilter：压缩数据集的单元。

47) vtkSimpleElevationFilter：将坐标点的 Z 坐标值生成能被可视化的标量值。

48) vtkSpatialRepresentationFilter：将查询的空间对象用多边形数据对象表示。

49) vtkStreamer：抽象基类，描述矢量场中质点的运动。

50) vtkStreamLine：一个具体的类，描述矢量场中质点的运动方向，并连成运动轨迹线。

51) vtkStreamPoints：描述矢量场中质点的运动轨迹，并用点标识质点的运动轨迹。

52) vtkSurfaceReconstructionFilter：用一系列杂乱的数据点，构造一个表面。

53) vtkTensorGlyph：对张量值进行符号化处理。

54) vtkTextureMapToBox：生成三维纹理坐标。

55) vtkTextureMapToCylinder：生成二维纹理柱面坐标。

56) vtkTextureMapToPlane：生成平面二维纹理坐标。

57) vtkTextureMapToSphere：生成二维纹理球面坐标。

58) vtkThreshold：提取在指定阈值范围内的单元。

59) vtkThresholdPoints：提取在指定阈值范围内的点。

60) vtkTresholdTextureCoords：计算满足阀值标准的纹理坐标。

61) vtkTransformTextureCoords：变换纹理坐标。

62) vtkVectorDot：矢量和法线点乘，生成标量值。

63) vtkVectorNorm：计算标量值。

64) vtkVectorTopology：标记矢量场消失的点。

65) vtkVoxelModeller：把任意类型的数据集转换成体元。

- 处理 vtkPointSet 类型数据集的过滤器

1) vtkDelunay2D：构建狄洛尼三角网。

2) vtkDelaunay3D：创建三维狄洛尼三角网。

3) vtkTransformFilter：使用变换矩阵变换点集。

4) vtkWarpLens：根据光学畸变，对点集变换。

5) vtkWarpScalar：根据缩放比例，缩放点集的坐标。

6) vtkWarpTo：根据点的扭曲方向修改点的坐标。

7) vtkWarpVector：根据点向量方向的缩放比例修改点的坐标。

- 处理 vtkPolyData 类型数据集过滤器

同时接收 vtkDataSet、vtkPointSet 数据集类型作为输入的过滤器也可以处理 vtkPolyData 类型的数据集。

1) vtkAppendPolyData：将一个或多个 vtkPolyData 数据集添加到其它数据集中。
2) vtkApproximatingSubdivisionFilter：使用逼近的方法生成表面的细部。
3) vtkArcPlotter：沿着多义线绘制数据。
4) vtkButterflySubdivisionFilter：使用碟形细部方案，细分三角型或多边形表面。
5) vtkCleanPolyData：合并符合条件的点，删除退化的图元。
6) vtkClipPolyData：用隐函数或标量值裁减多边形数据集。
7) vtkDecimate：缩减三角形的数目。
8) vtkDecimatePro：缩减三角形的数目。
9) vtkDepthSortPolyData：按照深度值对绘制的物体进行排序，便于透明绘制。
10) vtkExtractPolyDataGeometry：根据隐函数设定的值，提取多边形单元。
11) vtkFeatureEdges：提取满足条件的边。
12) vtkHull：用多个自由面生成凸壳。
13) vtkLinearExtrusionFilter：挤压其它的多边形数据集，生成新的数据集。
14) vtkLinearSubdivisionFilter：使用线性细部方法，细分三角形或多边形表面。
15) vtkLoopSubdivisionFilter：使用循环细部方法，细分三角形或多边形表面。
16) vtkMaskPolyData：选择多边形数据的部分数据。
17) vtkPolyDataConnectivityFilter：提取连通区域。
18) vtkPolyDataNormals：生成表面法矢量。
19) vtkQuadricClustering：大型数据集的抽选算法。
20) vtkQuadricDecimation：使用二次误差测量的抽选算法。
21) vtkReverseSense：颠倒联通次序或表面法向量的方向。
22) vtkRibbonFilter：根据线创建有向带。
23) vtkRotationalExtrusionFilter：通过旋转挤压其它 vtkPolyData 数据对象生成多边形数据。
24) vtkRuledSrufaceFilter：根据平行的线构造面。
25) vtkSelectPolyData：根据回路选择多边形数据。
26) vtkShrinkPolyData：向单元的质心收缩单元。
27) vtkSmoothPolyDataFilter：使用拉普拉斯算子对网格光滑处理。

28) vtkStripper：根据输入三角网生成三角带。

29) vtkSubPixelPositionEdgels：根据梯度，调整边缘线的位置。

30) vtkThinPlateSplineMeshWarp：使用界标弯曲多边形网格。

31) vtkTransformPolyDataFilter：根据 4*4 的变换矩阵变换多边形数据。

32) vtkTriangleFilter：根据多边形或三角带生成三角形。

33) vtkTriangularTCoords：生成二维三角形纹理地图。

34) vtkTubeFilter：用管覆盖线。

35) vtkVoxelContoursToSurfaceFilter：将线轮廓变换成表面。

36) vtkWindowedSincPolyDataFilter：用窗口化的 sinc 函数对网格进行光滑处理。

- 处理 vtkStructuredGrid 类型数据集的过滤器

同时接收 vtkDataSet、vtkPointSet 数据集作为输入的过滤器也可以处理 vtkStructuredGrid 数据集。

1) vtkExtractGrid：对 vtkStructuredGrid 抽取有用区域或进行二次抽样。

2) vtkGridSynchronizedTemplates3D：高效等值线算法。

3) vtkStructuredGridGeometryFilter：提取一部分栅格作为多边形数据。

4) vtkStructuredGridOutlineFilter：生成结构化网格的边界轮廓线。

- 以下过滤器处理 vtkUnstructuredGrid 类型数据集

接收 vtkDataSet 数据集为输入的过滤器也接收 vtkUnstructuredGrid 数据集为输入。

1) vtkExtractUnstructuredGrid：从非结构化网格中提取点或单元的区域。

2) vtkSubdivideTetra：把每一个原始的四面体网格细分成 12 个四面体。

- 处理 vtkRectilinearGrid 数据集的过滤器。

接收 vtkDataSet 数据集的过滤器也会接收 vtkRctilinearGrid 数据集。

1) vtkRectilinearGridGeometryFilter：提取一部分栅格作为多边形数据。

8.2.4 映射器对象

本小节主要描述映射器对象，映射器对象是可视化流水线的终点，以下对映射器对象及其输入类型作简单的描述。

1) vtkDataSetMapper：将输入的任意类型数据集在图形系统中绘制。

2) vtkImageMapper：二维图像显示。

3) vtkLabeledDataMapper：生成数据的三维文字标签。

4) vtkPolyDataMapper：在图形系统中对多边形数据进行绘制。

5) vtkPolyDataMapper2D：在覆盖面上绘制 vtkPolyData。

6) vtkTextMapper：生成二维的注释文本。

7) vtkVolumeProMapper：经过 VolumePro 硬件绘制板将体绘制成图像。

8) vtkVolumeRayCastMapper：用光线投射算法将体绘制成图像。

9) vtkVolumeTextureMapper2D：通过二维纹理算法将体绘制成图像。

8.2.5 角色对象

本小节主要描述在系统中可用的几种角色对象。

1) vtkActor2D：支持在覆盖面绘图的类型。

2) vtkAssembly：vtkProp3D 的拥有变换矩阵的规则分组。

3) vtkAxisActor2D：在覆盖面绘制的单个带标记的轴线。

4) vtkCaptionActor2D：为对象添加文字说明。

5) vtkCubeAxesActor2D：为 vtkProp 绘制 x-y-z 轴。

6) vtkFollower：面向摄像机的 vtkProp3D 类型。

7) vtkImageActor：在单个多边形之上绘制纹理地图。

8) vtk Legend Box Actor：被 vtkXYPlotActor 用来绘制曲线图例，将文本，符号，线组合到曲线图例中。

9) vtkLODActor：三维几何体表达的简单细节层次设计。

10) vtkLODProp3D：vtkProp3D 的细节层次方法。

11) vtk Parallel Coordinates Actor：多元可视化技术。

12) vtkPropAssembly：vtkProp 的集合。

13) vtkProp3D：vtkProp 的变化类型。

14) vtkScalarBar Actor：用带标记的色块来直观地表达颜色和标量值的关系。

15) vtkScaledTextActor：视口大小改变的比例。

16) vtkVolume：用于体描绘。

17) vtkXYPlotActor：绘制数据的 x-y 图。

9 可视化流水线

9.1 概述

数据可视化是以图形的形式描述数据信息，数据可视化主要包括两方面的内容：变换和表现。

变换过程主要包括两个步骤：

- 将数据转换成基本图形元素
- 转换成可被计算机显示的图形

例如，按时间序列提取股票价格数据，并且建立时间和价格之间的函数关系是一个变换过程。

9.1.1 数据可视化示例

二次曲面可视化示例，二次曲面方程如 1-1 式：

$$F(x,y,z)=a_0x^2+a_1y^2+a_2z^2+a_3xy+a_4yz+a_5xz+a_6x+a_7y+a_8z+a_9 \quad \text{式 1-1}$$

曲面的取值范围：-1<=x,y,z<=1，在取值范围内进行数据点采样，各个方向统一采样 50 个数据点，形成 50*50*50 的格网数据，使用三种不同的可视化技术，对二次曲面进行可视化，结果如图 9-1(a)左图生成一个 3D 表面，对应方程 $F(x,y,z)=c$，图中对数据剪切生成不同的平面，右图生成了三个平面的等值线轮廓线。

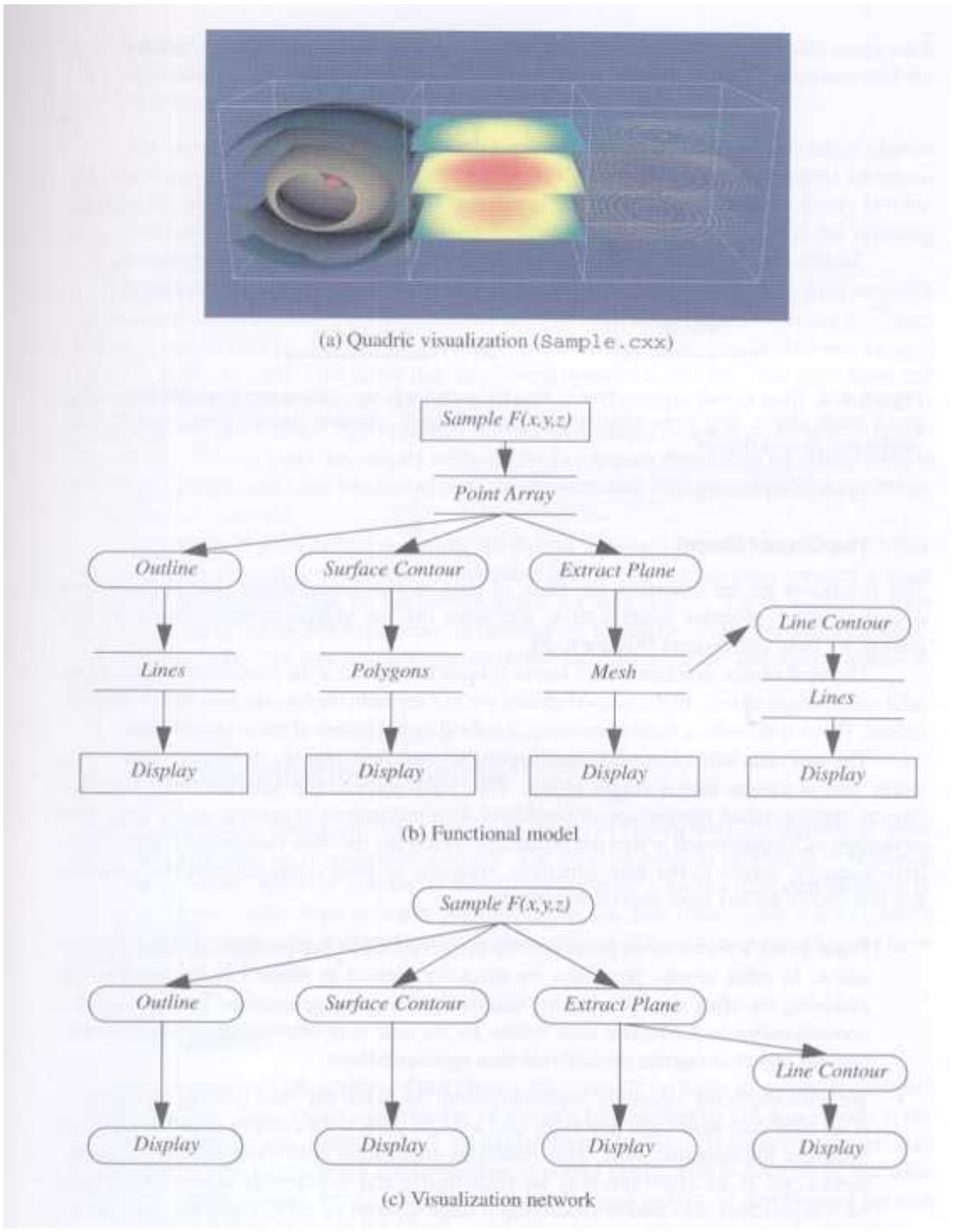


图 9-1 二次曲面可视化

9.1.2 功能模型

图 9-1(b)描述了一个功能模型，该功能模型描述了可视化的步骤，在功能模型中，用椭圆形块表示对数据的处理过程，用矩形块表现数据的存储或读取过程，箭头表示数据的流向。在 VTK 中，功能模型包含三种对象，没有输入并且能创建数据的对象被称为源对象（如在

VTK 中用于创建圆锥体的对象)，接收数据输入并对数据处理，但不输出数据的对象称为映射器对象，即有输入又有输出的对象成为过滤器。

9.1.3 可视化模型

图 9-1(c)表现了一个可视化模型，在这个模型中根据图中输入、输出来确定源对象、过滤器对象、映射器对象，在图中用椭圆形表示需要表现的数据对象。

9.1.4 对象模型

功能模型描述了数据在可视化流水线中的处理过程，对象模型描述了数据被哪些对象处理，在 VTK 系统中什么是对象呢？有如下两种选择：

第一种选择：将数据存储和数据的处理整合在一起，构成一个对象；第二种选择：将数据的存储和数据的处理分开，分别构建不同的对象,从面向对象的观点出发，第一种选择符合面向对象的特性，但是从数据管理的角度出发，VTK 选择了第二种构建对象方式，即 VTK 中的对象包含数据存储（如源对象）、数据处理两种类型的对象（如过滤器对象）。

9.2 可视化流水线

可视化流水线描述了数据的流向，功能模型也被称为可视化流水线，从图 1-1 的功能模型可以看出，流水线主要由数据对象、处理对象和表示数据流向的箭头线组成，在以后，经常使用可视化流水线描述数据可视化的实现方式。

9.2.1 数据对象

数据对象主要提供了数据的创建、存取功能以及提取数据一些特征的功能(如计算最大、最小值等)。

9.2.2 过程对象

过程对象主要对输入的数据进行处理，并将处理后的数据输出，过程对象又分为源对象、过滤器对象、映射器对象。

源对象用于创建或生成数据，如式 1-1 用二次曲面方程生成数据，这些对象成为源对象，该对象没有输入，只有输出，其主要功能是生成数据并输出。

过滤器对象有一个或多个输入和输出，该对象的主要功能是对输入的数据进行处理，并生成新的输出数据。

映射器对象只有输入没有输出，其主要功能是将数据映射到图形系统或以文件的形式存储数据，该对象是可视化流水线中的终端节点对象。

9.3 流水线拓扑结构

这一部分主要介绍如何将不同类型的过程对象连接在一起，形成可视化流水线。

9.3.1 流水线的连接

过程对象能以多种方式相互连接，构成不同类型的可视化流水线，在构建可视化流水线时，有两点需要注意：类型和多重性。

类型指过程对象所能处理的数据类型，各种类型的过程对象所接收和处理并输出的数据类型是不相同的，所以，在建立连接时，要明确所连接的过程对象所能接收和输出的数据类型，在 VTK 中对输入数据类型的控制提供了两种方法，一种是建立单一数据类型处理的过程对象，只能处理单一的数据类型，另一种是建立数据类型检查机制，对数据类型进行检查，只有处理数据类型相互兼容的过程对象才能相互连接。如图 9-2 所示。

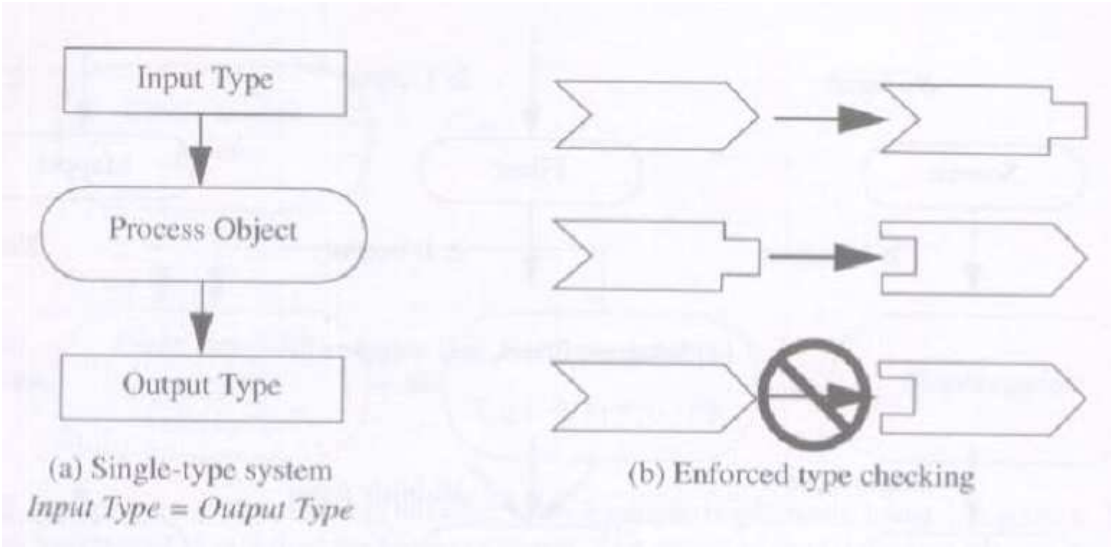


图 9-2 保持兼容的数据类型

多重性指过程对象能接收输入数据或者被允许输出数据的个数，如过滤器和映射器必须至少有一个输入，有些过滤器指定了输入数据的个数，如布尔运算过滤器，必须指定两个输入，我们需要区别多重输出的含义，一般的情况下，源对象和过滤器对象只有一个输出，当

一个输入同时被多个过程对象使用，会同时产生多个输出，如一个输入数据同时被两个过程对象使用，一个用于生成线框图形，一个用于生成轮廓线，则同时产生两个输出。

9.3.2 循环机制

在可视化流水线中引入反馈循环机制，用于将过程对象的输出数据变成输入数据反馈给过程对象，作为过程对象的输入数据，如下图所示，Integrate 过滤器对初始输入的数据处理后，再将输出的数据反馈给 Probe 过滤器。

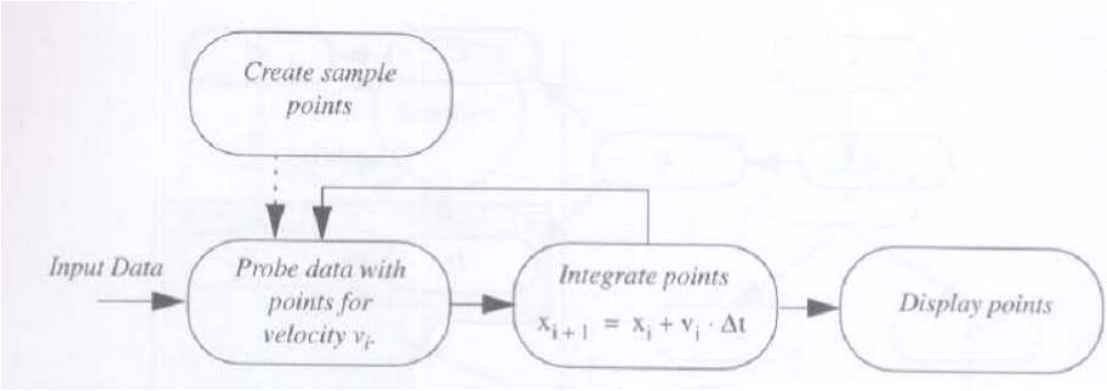


图 9-3 流水线的循环机制

9.4 流水线执行

可视化流水线中的过程对象必须对输入的数据进行处理，才能得到期望的结果，流水线中每个对象的处理数据的过程称为流水线执行。

流水线中的过程对象或输入数据发生改变时，流水线都要重新执行，以便于取的最新的执行结果，所以流水线的执行是反复进行的。

为了保持流水线执行的效率，可以设计多条并行流水线，处理不同的任务，每条流水线在执行过程中，必须保持过程对象的同步，在 VTK 中提供了如下两种同步方式：

- 显式执行：直接对流水线的改变进行跟踪，对改变的情况进行分析，依据分析结果，控制流水线过程对象的执行（如图 9-4（a))。
- 隐式执行：仅仅当流水线中某个过程对象的参数或输入数据改变时，执行这个过程对象(如图 9-4（b))。

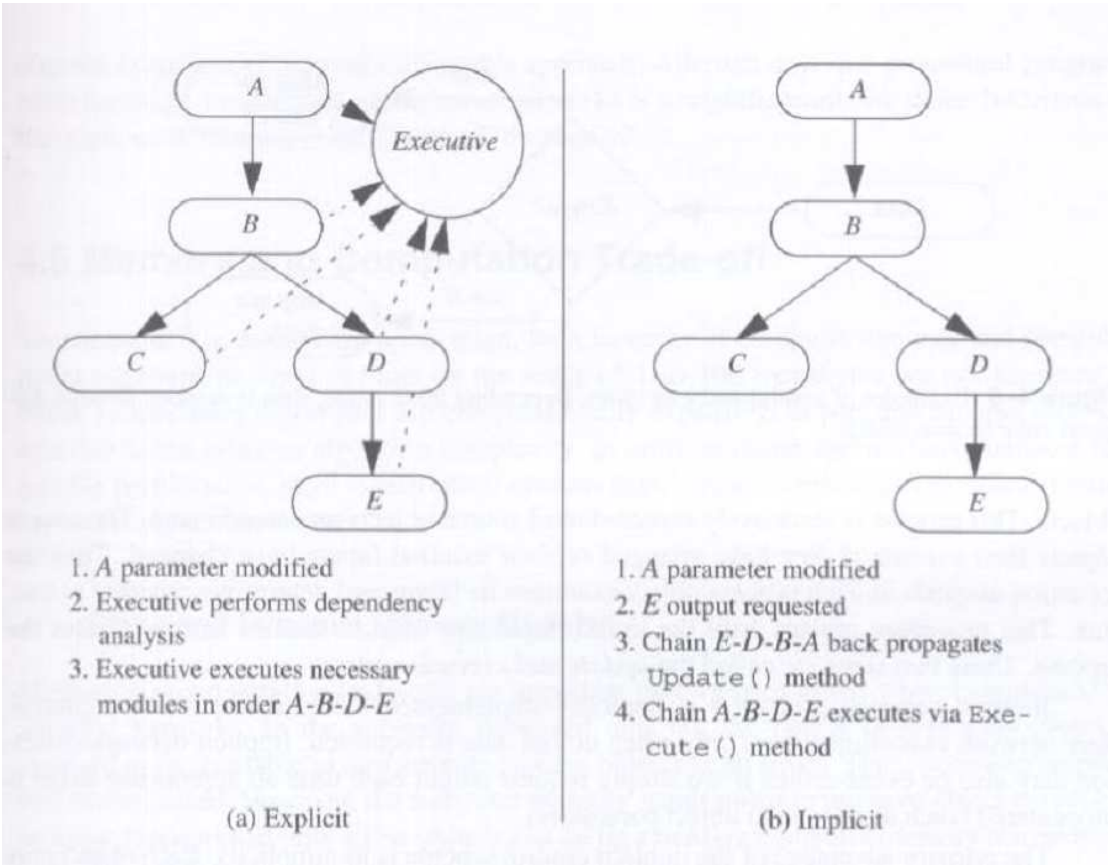


图 9-4 显式执行和隐式执行

可视化流水线还有一个重要的功能是条件执行，根据数据的要求，执行流水线中不同的过程对象，如图 9-5 所示。

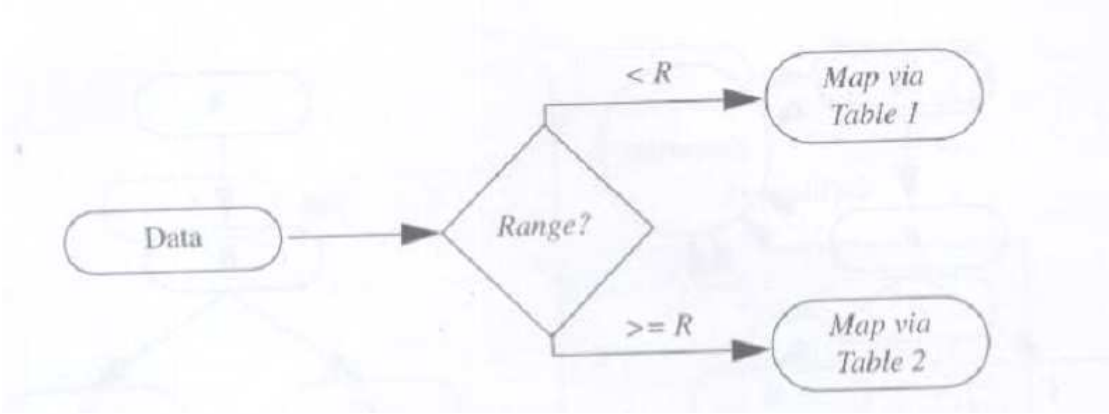


图 9-5 条件执行

9.5 数据接口

VTK 提供了三种方法将数据应用到可视化流水线中。

- 程序接口

最灵活和不受任何限制的方式，在编写应用程序时，直接读取、存储数据和对数据进行

处理。

- 文件接口

从数据文件中读取数据和存储数据到数据文件，在可视化流水线中的源对象，如阅读器对象，映射器对象，如复写器对象，都可实现文件的读取及存储，创建相应的数据对象，并将数据对象应用到可视化流水线中。

- 系统接口

读取或存储三维场景，如 VTK 提供的导入器、导出器对象，如读取 3ds 或 vrml 文件，创建场景的绘制窗口、灯光等。

9.6 综合应用

本节主要描述实现细节。

9.6.1 隐含控制执行

在可视化流水线中实现了隐式控制流水线的执行，这种方式是最简单和透明的，实现隐式执行的关键是两个方法：Update（）和 Execute()。

当请求系统绘制场景时，Update()这个方法被调用，当流水线中对象的创建时间发生改变时，执行 Excute（）方法。

假设一个可视化流水线由源对象、过滤器对象、映射器对象组成，其隐含执行过程如下：

1、用户发出绘制场景的请求。

2、图形系统的角色（Actor）对象给映射器对象发送场景将要绘制的信息，流水线开始执行。

3、映射器对象调用 Update()方法，然后过滤器对象、源对象依次调用 Update()方法。

4、源对象调用 Update()方法后，开始比较当前被修改的时间和最后执行的时间，如果当前被修改的时间比最后被执行的时间更新，说明源对象最近被修改过，还未执行，于是调用 Execute()方法开始执行流水线。

5、过滤器对象及映射器对象开始比较它们的修改时间和最后的执行时间，根据比较的结果决定是否调用 Execute()方法。

9.6.2 多输入输出

VTK 可视化流水线中的过程对象被设计成可支持多个输入和输出，实际上，因为很多的数学算法只需要提供一组输入数据，生成一组输出数据，所以，我们发现很多的过程对象实际上只提供了单个的输入、输出，当然，也有一些过程对象提供了多个输入、输出。

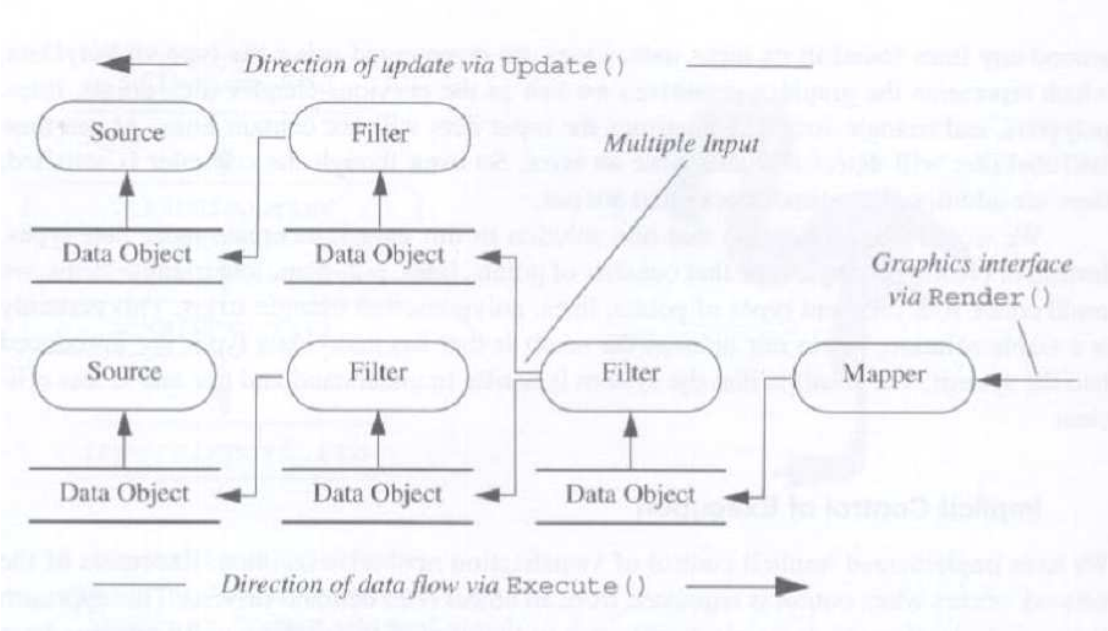


图 9-5 隐式执行过程的实现

图 9-5 显示了流水线架构的设计，图中描述了数据对象和过滤器对象是如何连接在一起，构成了一个可视化流水线，在流水线中，过程对象之间的数据传输通过 SetInput()、GetOutput（）方法，其中 SetInput()方法用于设置输入数据，GetOutput()方法用于输出数据。用于连接过滤器的 C++示例代码如下：

```
filter2->SetInput(filter1->GetOutput());
```

在这里，filter1 和 filter2 是类型相互兼容的过滤器对象。

通过以上流水线执行过程的介绍，我们可以看出流水线中过程对象多输入、输出是可以实现的，和单输入、输出的过程对象相比，多输入、输出的过程对象在执行 Update()和 Excute()时，要对所有的输入、输出对象都要执行上述操作。

在 VTK 中，典型的具有多输入、输出的过程对象包括：

- vtkGlyph3D
- vtkExtractVectorComponents

vtkGlyph3D 过滤器是一个具有多个输入一个输出过滤器，SetInput()输入的是用于被符

号化的多边形数据对象，SetSource()方法输入的是表示符号的源对象，用该符号表示多边形数据对象，如下面的示例代码所示：

```
glphy=vtkGlphy3D::New（）；
//输入被符号化的多边形数据对象
glphy->SetInput(foo->GetOutput());
//输入源对象，表示符号对象
glphy->SetSource(bar->GetOutput());
```

vtkExtractVectorConpoments 过滤器是一个具有一个输入多个输出的过滤器，该过滤器主要提取向量值的三个分量,输出方法为:VxCompomnet()、VyCompomnet()、VzCompomnet()分别表示向量的三个分量 x,y,z。

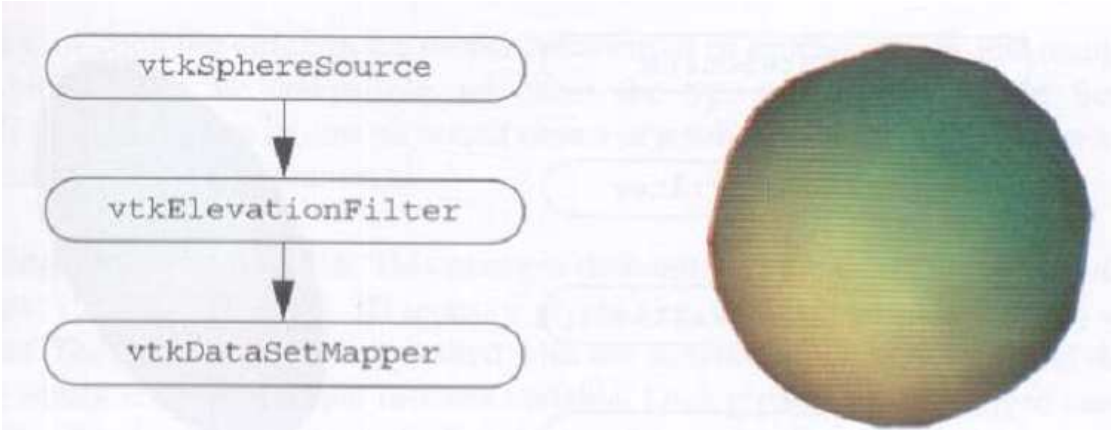
示例代码：

```
Vz=vtkExtractVectorConpoments::New();
Foo=vtkDataSetMapper::New();
Foo->SetInput(Vz->VzCompomnet());
```

9.7 可视化流水线示例

9.7.1 简单球体

该示例程序建立一个简单的可视化流水线，用源对象建立一个球体，然后用 Z 轴和球体相交，过滤器计算和 Z 轴相交的球体点的 y 值，并把高度值映射成颜色，显示球体，处理过程如下图所示：

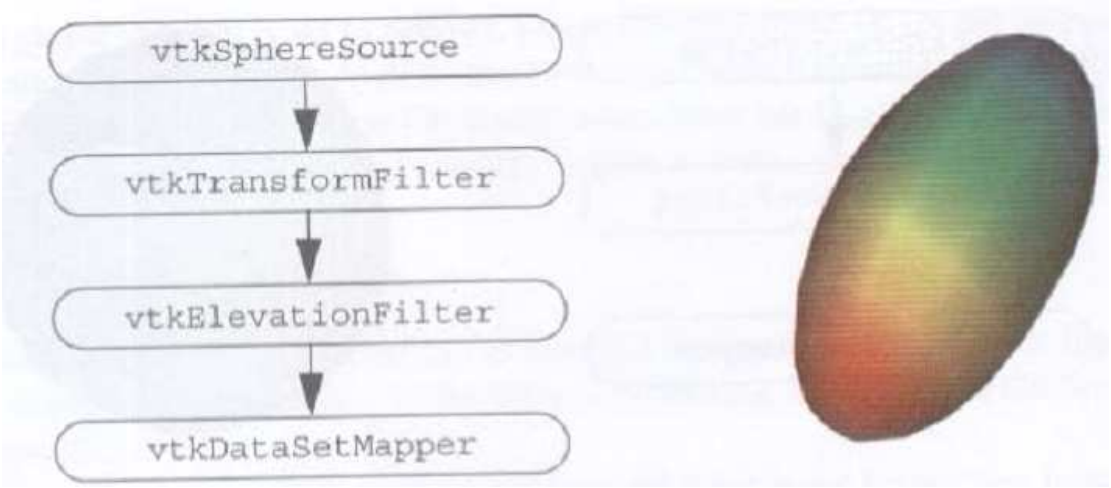


示例代码如下：

```
//用源对象创建球体
vtkSphereSource *Sphers=vtkSphereSource::New();
sphere->SetThetaResolution(8);
sphere->SetPhiResolution(8);
//提取球体点的 y 值（高度值）
vtkElevationFilter *colorIt= vtkElevationFilter::New();
//设定高度范围
colorIt->SetLowPoint(0,0,-0.5);
colorIt->SetHighPoint(0,0,0.5);
colorIt->SetInput(Sphers->GetOutput());
//数据映射
vtkDataSetMapper *map=vtkDataSetMapper::New();
map->SetInput(colorIt->GetOutput());
```

9.7.2 弯曲球体

该示例在上个示例的基础上，在 x,y,z 三个方向对球体进行不均匀变换导致球体边形，处理过程如下图所示：



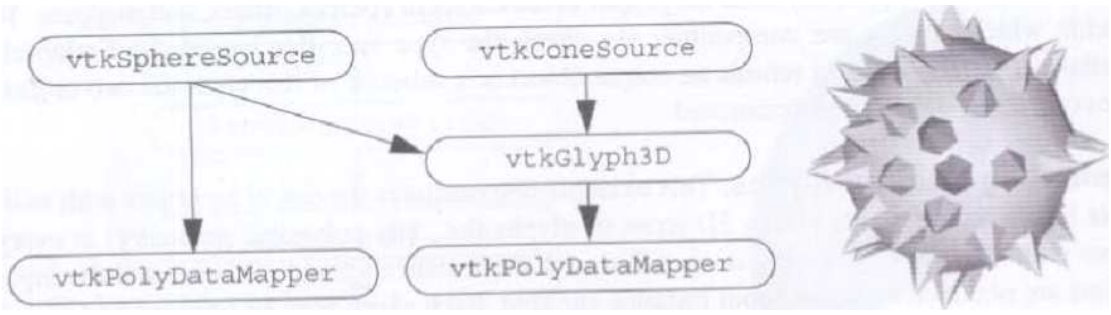
示例代码如下：

```
//用源对象创建球体
vtkSphereSource *Sphers=vtkSphereSource::New();
sphere->SetThetaResolution(8);
```

```
sphere->SetPhiResolution(8);
//创建变换对象，缩放变换
vtkTransForm *aTransForm=vtkTransForm::New();
aTransForm->Scale(1.0,1.5,2.0);
//建立变换过滤器，执行变换
vtkTransformFilter *transFilter=vtkTransformFilter::New();
//设定变换对象
transFilter->SetInput(Sphers->GetOutput());
//执行变换
transFilter->SetTransForm(aTransForm);
//提取球体点的 y 值（高度值）
vtkElevationFilter *colorIt= vtkElevationFilter::New();
//设定高度范围
colorIt->SetLowPoint(0,0,-0.5);
colorIt->SetHighPoint(0,0,0.5);
colorIt->SetInput(transFilter->GetOutput());
//数据映射
vtkDataSetMapper *map=vtkDataSetMapper::New();
map->SetInput(colorIt->GetOutput());
```

9.7.3 符号化处理

该示例主要说明如何在可视化流水线中用多输入、输出过滤器，用 vtkGlyph3D 过滤器对输入的数据点符号化处理，处理过程如下图所示：

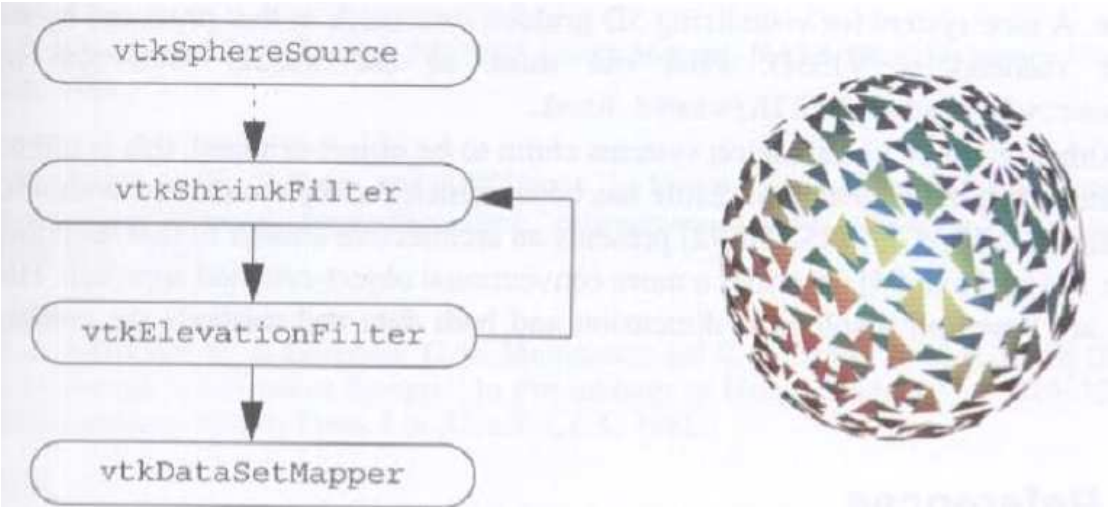


示例代码如下：

```
//创建球体
vtkSphereSource *sphere = vtkSphereSource::New();
sphere->SetThetaResolution(8);
sphere->SetPhiResolution(8);
vtkPolyDataMapper *sphereMapper = vtkPolyDataMapper::New();
sphereMapper->SetInputConnection(sphere->GetOutputPort());
vtkActor *sphereActor = vtkActor::New();
sphereActor->SetMapper(sphereMapper);
//创建圆锥源对象，表示符号
vtkConeSource *cone = vtkConeSource::New();
cone->SetResolution(6);
vtkGlyph3D *glyph = vtkGlyph3D::New();
//第一个输入，需要符号化的数据对象
glyph->SetInputConnection(sphere->GetOutputPort());
//第二个输入，圆锥符号
glyph->SetSourceConnection(cone->GetOutputPort());
glyph->SetVectorModeToUseNormal();
glyph->SetScaleModeToScaleByVector();
glyph->SetScaleFactor(0.25);
//映射
vtkPolyDataMapper *spikeMapper = vtkPolyDataMapper::New();
spikeMapper->SetInputConnection(glyph->GetOutputPort());
```

9.7.4 隐藏球体

该示例用反馈机制构建可视化流水线，并且说明如何改变网络拓扑，处理过程如下图所示：



示例代码如下：

```
vtkSphereSource *sphere = vtkSphereSource::New();
sphere->SetThetaResolution(12); sphere->SetPhiResolution(12);
vtkShrinkFilter *shrink = vtkShrinkFilter::New();
shrink->SetInput((vtkDataSet *)sphere->GetOutput());
shrink->SetShrinkFactor(0.9);
vtkElevationFilter *colorIt = vtkElevationFilter::New();
colorIt->SetInput((vtkDataSet *)shrink->GetOutput());
colorIt->SetLowPoint(0,0,-0.5);
colorIt->SetHighPoint(0,0,0.5);
vtkDataSetMapper *mapper = vtkDataSetMapper::New();
mapper->SetInput(colorIt->GetOutput());
vtkActor *actor = vtkActor::New();
actor->SetMapper(mapper);
//第一次绘制
renWin->Render();
//创建循环
shrink->SetInput(colorIt->GetOutput());
// 开始执行循环
renWin->Render();
```

10 基本数据表达

10.1 可视化数据的特点

为了更好的采用可视化的手段表达数据，必须要了解可视化数据的一些特性，以便于设计有效的数据结构和存取方法，可视化数据具有以下三个特点：

- 离散性

为了让计算机能够获取、处理和分析数据，必许对无限、连续的空间体进行采样，生成有限的采样数据点，这些数据以离散点的形式存储，采样的过程是一个离散化的过程，为了求得连续空间体的任意一点的值，必须依据离散点进行插值计算。

- 数据具有规则的或不规则的结构

规则结构数据点之间有固定的关联关系，可以通过这些关联确定每个点的坐标，非规则结构数据之间没有固定的关联关系，优点是可以更加细致、灵活的表现数据。

- 数据具有维度

可视化数据具有零维、一维、二维、三维等任意维度，数据的维度决定了数据可视化的方法，如：零维的数据表现为点，一维数据表现为曲线，二维数据表现为曲面，三维数据表现为体等。

10.2 数据对象

在 VTK 中，数据一般以数据对象的形式表现，数据对象是数据的集合，数据对象表现的数据是被可视化流水线处理过的数据，只有数据对象被组织成一种结构后，才能被 VTK 提供的可视化算法处理，数据对象组成如下图所示：

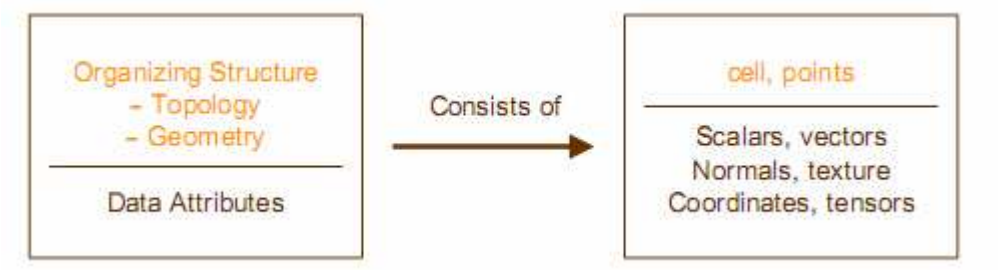


图 10-1　数据对象组成

数据主要由结构和属性数据组成，其中结构包括拓扑结构和几何结构（几何形状），几何结构和拓扑结构在实际应用中用点集（points）和单元（cell）表达，属性数据包括：标量、矢量、张量、纹理坐标四种类型。

10.3 数据集

数据对象被组织成一种结构并且被赋予属性值时形成数据集，数据集是一种抽象的类，一般情况下用具体的子类来表达数据集，在 VTK 中有许多算法可以处理数据集，数据集的结构由拓扑结构和几何结构两部分组成，拓扑结构描述了物体的构成形式，几何结构描述了物体的空间位置关系，如对于一个立方体而言，用 8 个离散的顶点描述立方体的空间位置，这 8 个顶点构成了立方体的几何结构，立方体由 6 个面组成，6 个面之描述了立方体的构成形式。

在 VTK 中数据集的结构用单元(cell)和点集(points)来表达，单元表达数据集的构成形式，点集存储几何结构的空间位置信息。

属性数据描述几何结构或拓扑结构的特性，一般包括标量、矢量、纹理坐标等，如描述立方体顶点的颜色或立方体每个面的颜色等。

10.3.1 单元

数据集由一个或多个单元组成，单元是建立数据可视化的基础，如下图所示为一个单元示例：

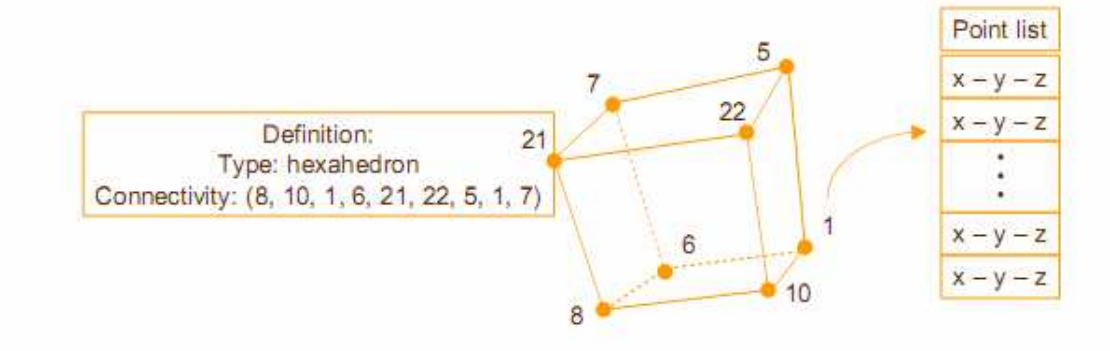


图 10-2　六面体单元构成

在这个示例中，可以看出单元由两部分构成：

- 单元的类型
- 构成单元的顶点列表

在这个示例中，单元的类型为 6 面体，顶点列表的顶点决定了单元的类型，并且隐含的定义了单元的拓扑组成形式，如顶点 8 和 0 定义了一条边，顶点 8、10、1、6 定义了一个面，顶点的空间坐标(x,y,z)定义单元的几何形状。

单元具有零维、一维、二维和三维等拓扑维度空间，如顶点、线、三角形和四面体分别代表零维、一维、二维和三维拓扑维度，对于不同维度空间的单元，又进一步可分为基本单

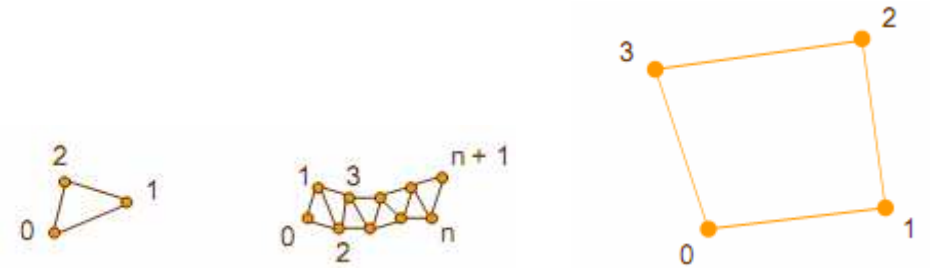
元和组合单元，基本单元是不可再分的单元，组合单元由基本单元组合而成，如二维的三角条带是一个组合单元，其由若干个三角形基本单元组合而成。

依据构成单元顶定列表的变化，单元的类型是多种多样的，应根据应用的需求选择合适的单元类型，在 VTK 中将单元分为线性单元和非线性单元。

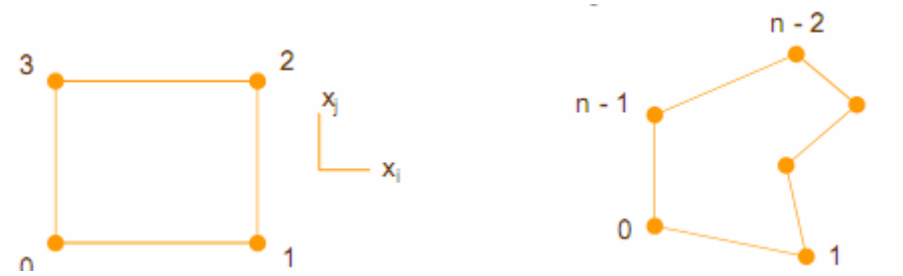
10.3.1.1线性单元

线性单元的特征是构成单元的每条边都为由两个点确定的直线，如图 10-1 所示的六面体单元，由顶点 8 和 10 采用线性插值方法确定一条边， VTK 中包含的线性单元如图 10-2 所示：

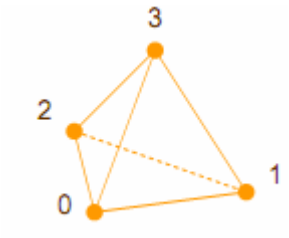
a（顶点） b（多顶点） c（线段） d（折线）



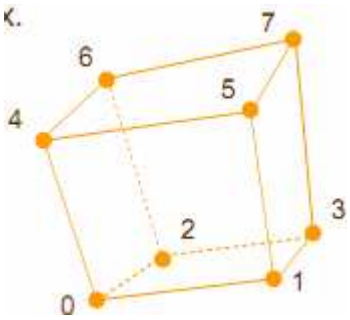
e（三角形） f（三角条带） g（四边形）



h（像素） i（多边形）

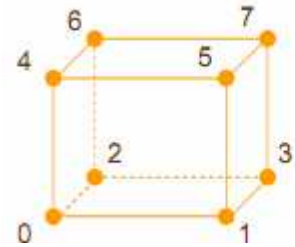


j（四面体）



k（六面体）

n（金字塔）



l（体素）

m（锲形）

图 10-2　线性单元

- 顶点单元

顶点单元是零维的基本单元，由一个点定义，如图 10-2(a)。

- 多顶点单元

是零维的组合单元，其定义不受顶点顺序的限制，如图 10-2(b)。

- 直线单元

是一维的基本单元，由两个点来定义，方向从第一个点指向第二个点，如图 10-2(c)。

- 折线单元

是一维的复合单元，有多个首尾相连的直线单元组成，如图 10-2(d)。

- 三角形单元

是二维的基本单元，由按逆时针排列的三个顶点构成，如图 10-2(e)。

- 三角条带单元

是二维的组合单元，由多个三角形单元构成，如图 10-2(f)。

- 四边形单元

是二维的基本单元，由位于一个平面内的 4 个顶点构成，如图 10-2(g)。

- 像素单元

是二维的基本单元，由位于一个平面内的 4 个顶点构成，但是每个边都相互正交，如图 10-2(h)。

- 多边形单元

是二维的基本单元，由位于一个平面内的多个顶点构成，如图 10-2(i)。

- 四面体单元

是三维的基本单元，由不在同一平面上的四个顶点构成，有 6 条边和 4 个三角形面，如图 10-2(j)。

- 六面体单元

是三维的基本单元，有 6 个四边形面，12 条边和 8 个顶点，如图 10-2(k)。

- 体素单元

是三维的基本单元，组成单元的 6 个面相互正交，如图 10-2(l)。

- 锲形单元

是三维的基本单元，由三个四边形面组成，如图 10-2(m)。

- 金字塔单元

是三维的基本单元，如图 10-2(n)。

10.3.1.2 非线性单元

在数值分析方面，更加流行的方法是使用非线性单元，非线性单元提供了更加精确的插值函数和更加好的模型曲线，VTK 仅仅对非线性单元提供了二次插值函数，线性单元和非线性单元的不同点是非线性单元在绘制和数据处理方法方面存在不同，线性单元可以很容易的转换成线性图元被图形库处理，而非线性单元不被图形库直接支持，因此非线性单元必须被转换成线性单元以后，才能被图形库所支持。

VTK 提供的非线性单元如下图所示：

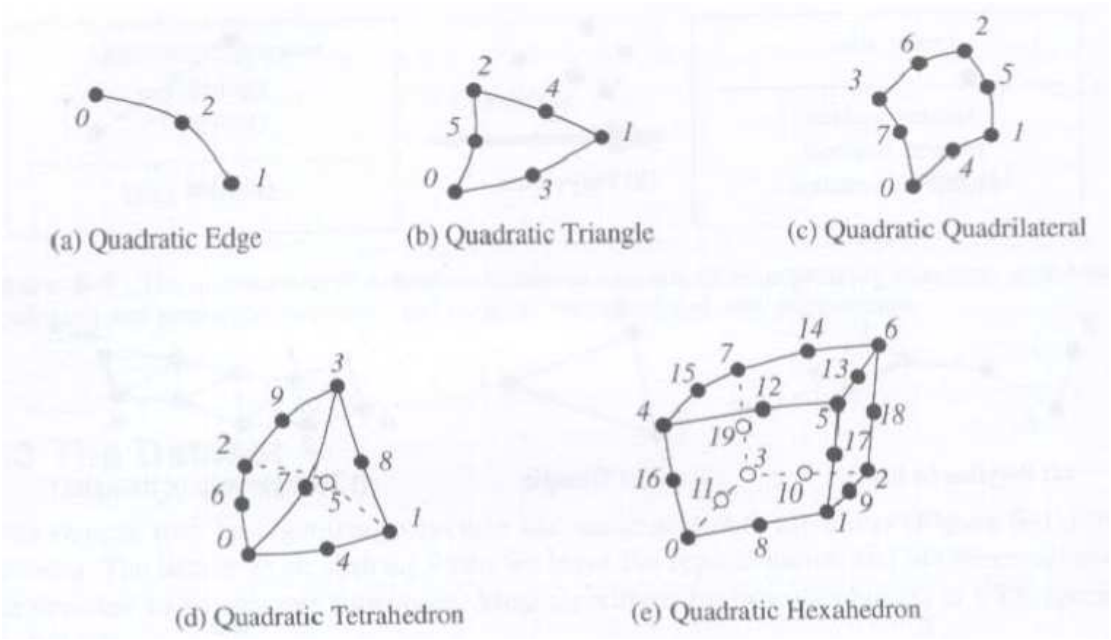


图 10-3　非线性单元

10.3.1.3 单元表达

在 VTK 中每种类型的单元是通过创建具体类型单元类的实例来实现的，每个单元是抽象类 vtkCell 的一个子类，单元的拓扑结构由顶点的排列顺序决定，顶点的空间坐标(x,y,z)定义单元的几何形状，如图 10-1 所示，单元的对象层次图如图 10-4 所示：

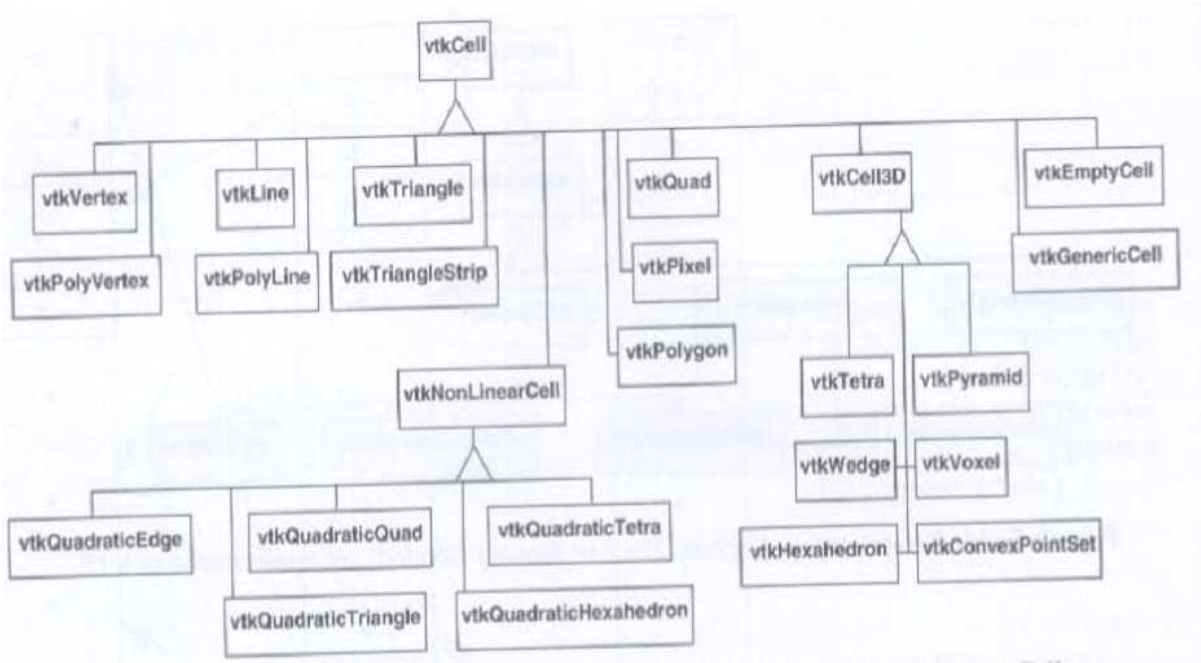


图 10-4 单元对象层次图

10.3.2 属性数据

10.3.2.1属性数据

属性数据用于描述数据集中的点或单元，有时也描述单元的一个面或一条边，属性数据按照数据类型进行分类，可视化算法也根据所处理的属性数据类型进行分类， VTK 提供的属性数据的类型如图 10-5 所示：

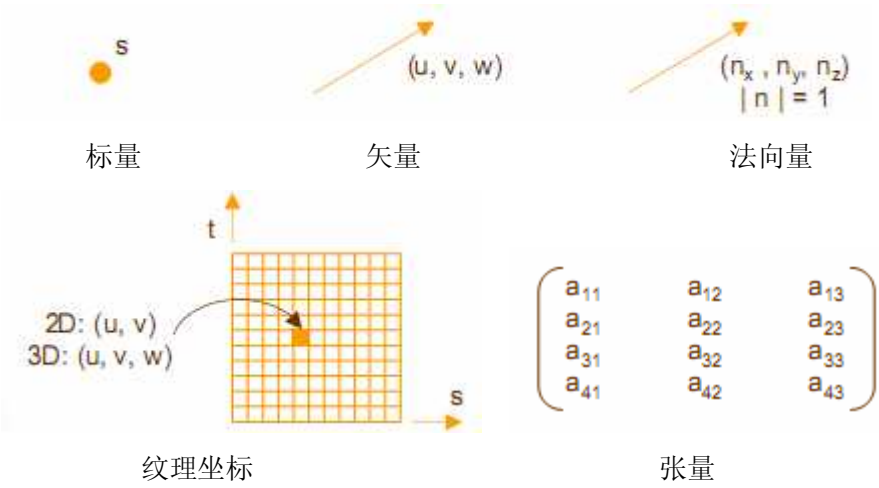


图 10-5 属性数据

- 标量（Scalars）

标量数据是没有大小和方向的数值，例如温度、压力、密度、海拔等，标量是最简单、最常用的可视化数据。

- 矢量（Vectors）

是拥有大小和方向的数据，在三维空间中用一个三元组（u,v,w）来描述。

- 法线（Normals）

法线是方向矢量，即模|n|=1 的矢量，通常法线在图形系统中用来控制对象的阴影。

- 纹理坐标(Texture Coordinates)

用来从卡笛尔空间向一维、二维或三维纹理空间绘制点，纹理空间通常称作纹理图。

- 张量(Tensors)

张量是矢量和矩阵通过复杂的数学算法得到的，一个 k 阶的张量可当作一个 k 维的表格。零阶的张量是标量，一阶的张量是矢量，二阶的张量是纹理坐标，三阶的张量是一个三维阵列。

属性数据只能和数据集中的点及单元关联，对于构成单元的基本组成要素，如边和面等

不能与数据属性关联，我们称与点关联的数据属性为点属性，与单元关联的数据属性为单元属性。

我们用 vtkPointData 类和 vtkCellData 类表达数据集属性，它们是类 vtkFieldData 的子类，如下图所示：

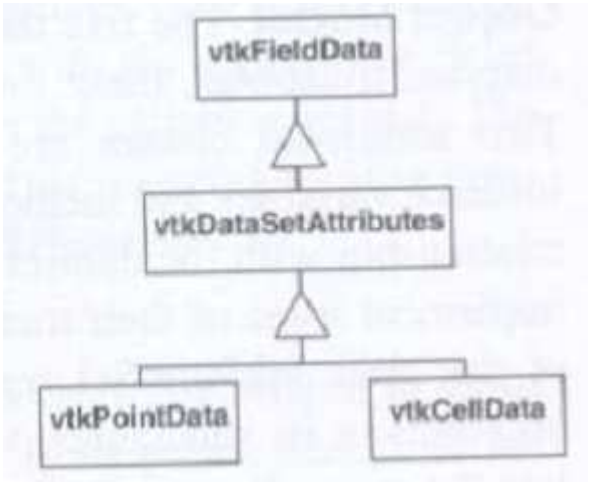


图 10-6 点属性和单元属性类图

构成数据集的每个点和属性数据之间存在一对一的关系，如一个数据集由 N 个点构成，那么必须有 N 个属性数据和这 N 个点一一对应，通过点的 ID 号就可以对该点的属性数据进行访问，例如在数据集 aDataSet 中访问 ID 号为 129 的点的标量值时（假设标量数据已被定义且不为空）使用如下方法：

```
aDataSet->GetPointData()->GetScalars()->GetScalar(129);
```

10.3.2.2数据存储

在 VTK 中用连续的数据数组作为数据结构的基础，因为它可被快速地创建、删除和遍历。VTK 中称其为数据数组，并用 vtkDataArray 类实现，如下图所示：

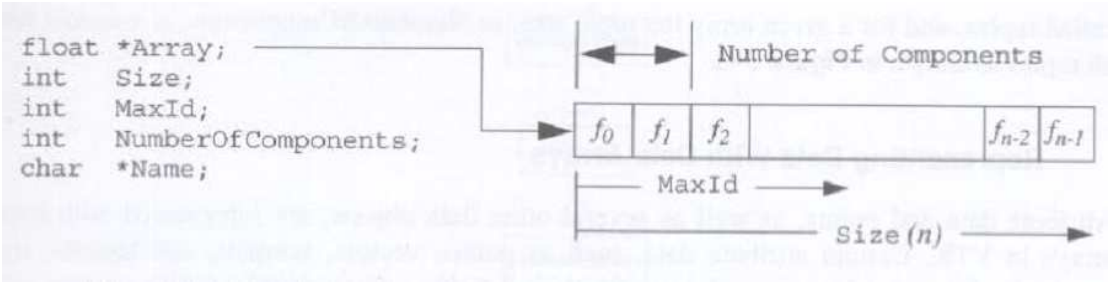


图 10-7 连续数组的实现

连续数组可以在网络上方便快速地传输，尤其是独立于计算机内存地址的数组数据，在 VTK 中通过数据数组索引去访问相关信息，和 C++数组一样数据组的下标也是从 0 开始的，

就是说，数组的大小是 N，那么系统将从索引 0 开始连续地存取它们直到 N-1。

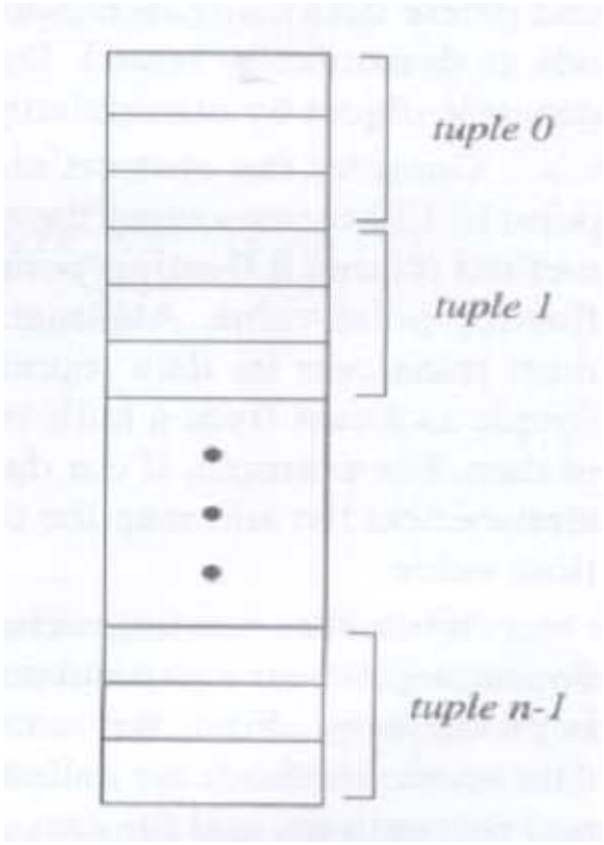


图 10-8 数据数组组成

我们将用 vtkFloatArray 类来描述如何在 VTK 中实现连续数组，图 10-7 所示的是一个指向实型的指针数组，数组的长度是由 Size 指定的，由于数组的长度是动态的增加的，所以当数组存储数据的长度超出指定的长度时，会自动触发 Resize()操作调整数组的长度，使数组的长度变成原来的两倍，MaxId 是一个整型的偏移量，用来定义最后一个被插入的数据，如果没有数据插入 MaxId 等于-1，否则的话 MaxId 的值介于 0 到 Size 之间，许多可视化数据是由多个数据分量组成的，如颜色数据由红、绿、蓝三个分量组成，为了在连续数据数组中表达这类数据，引入了元组的概念，组元是数据数组的子数组，用于存储数据类型相同的分量数据，元组的大小在给定后不会改变，如图 10-8 所示的数据数组由 N 个组元组成，每个组元存储三个分量数据，在 VTK 中数据都是用数据数组来表达的，有的数据如点、矢量、法线和张量等在定义时需要指定元组的大小。

可视化数据有多种类型，如简单的浮点型、整型、字节型和双精度型，还有复杂的类型如特征字符串和多维的标识符，既然有这么多种数据类型，那么数据数组是如何操作和表达这些数据的呢？VTK 通过对数据对象的抽象提供运行时解决方案和使用 C++编译时动态邦定的方法来解决这个问题的。

抽象数据对象就是通过动态绑定的方式使用统一的方法来创建、操作和删除数据，在C++中用 virtual 关键字来声明动态方法,VTK 对数据进行抽象抽象出数据数组的抽象类,(如vtkDataArry 类)并提供通用的数据操作功能，并对每种类型的数据都建立了相应的子类，VTK 提供的数据数组类型类如图 2-7 所示：

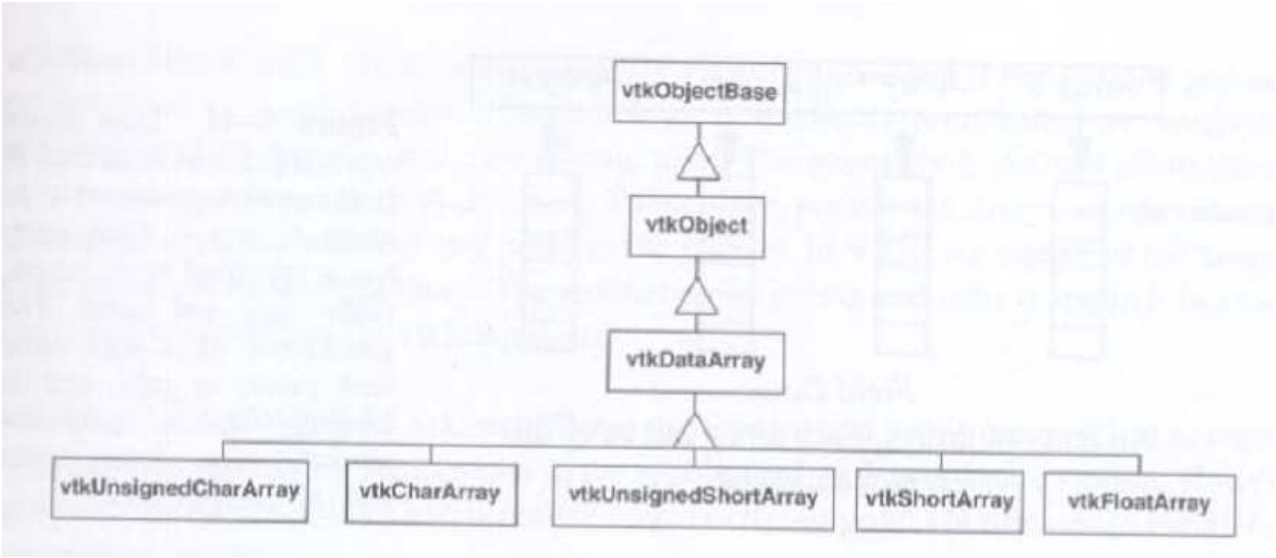


图 10-9 数据数组类层次图

10.4 数据集可视化

10.4.1 数据集类型

一个数据集由一个组织结构和关联的属性数据构组成,组织结构包括拓扑结构和几何结构，数据集的类型是由它的组织结构决定的，同时数据集的类型指定了点和单元之间的相互关系，如图 10-10 列出了常用的数据集类型。

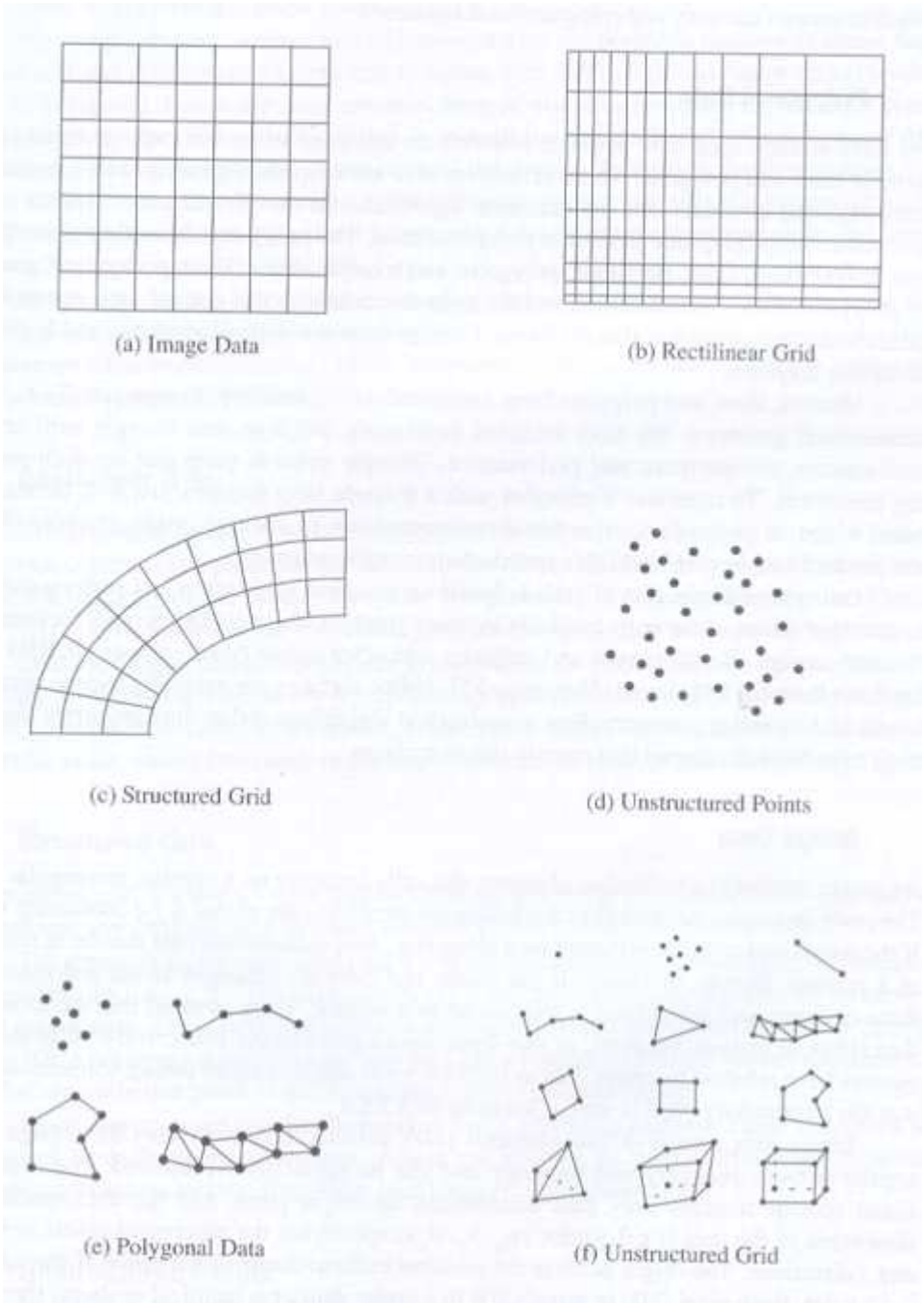


图 10-10 数据集类型

数据集依据其结构特征分为规则的或不规则的，规则数据集的点和单元都是规则排列的，每个点的位置都可以依据相互之间的关系得到，规规则结构数据集没有固定的模式，不能用简洁的方式描述，因此，对其维护和管理需要更加全面，非规则数据的使用更广泛，同时它也需要更多的内存和计算资源。

10.4.1.1均网格数据（Image Data）

是按规则排列在矩形方格中的点和单元的集合，如图 10-10（a）所示，如果数据集的点和单元被排列在平面（二维）上，称为此数据集为象素映射、位图或图象，如果被排列在层叠面（三维）上，则称为体，均匀网格数据是由线元素、象素或三维象素组成，由于均匀网格数据的点是规则排列的，所以每个点的位置可被隐式地表达，只需要知道数据的维数、起始点的位置和相邻点之间的间隔，就可以计算出每个点的空间位置，数据维数用一个三元组（nx,ny,nz）来表示，分别表示在 x,y,z 方向上点的个数。

10.4.1.2线性网格（Rectilinear Grid）

是排列在矩形方格中的点和单元的集合，如图 10-10(b)所示，线性网格的拓扑结构是规则的，但其几何结构只有一部分是规则的，也就是说，它的点是沿着坐标轴排列的，但是两点间的间隔可能不同，和均匀栅格数据相似，线性网格是由象素和三维象素组成的，它的拓扑结构通过指定网格的维数来隐式地表达，几何结构通过 x,y,z 坐标来表达。

10.4.1.3结构化网格（Structured Grid）

结构化网格具有规则的拓扑结构和不规则的几何结构，但是单元没有重叠或交叉，如图 10-9（c）所示，结构化网格的单元是由四边形或六面体组成，结构化网格通常用于有限差分析。

10.4.1.4非结构化点集（Unstructured Points）

是指不规则地分布在空间的点集，非结构化点集具有不规则的几何结构，但不具有拓扑结构，非结构化点集用离散点来表达，如图 10-9（d）所示。

通常，这类数据没有固定的结构，是由一些可视化程序识别和创建的，非结构化点集适合表现非结构化数据，为了实现数据的可视化，可将这种数据形式转换成其它一些结构化的数据形式。

10.4.1.5多边形数据(Polygonal Data)

在 VTK 中多边形数据集是由顶点、多顶点、线、多线、多边形各三角带构成，多边形数

据是非结构化的，并且多边形数据集的单元在拓扑维度上有多种变化，如图 10-10（e）所示。

顶点、线和多边形构成了用来表达 0、1 和 2 维几何图形的基本要素的最小集合，同时用多顶点、多线和三角形带单元来提高效率和性能,特别是三角形带用一个三角形带表达 N 个三角形只需要用 N+2 个点，但是用传统的表达方法需要用 3N 个点。

10.4.1.6非结构化网格（Unstructured Grid）

非结构化网格集是最常见的数据集类型，它的拓扑结构的几何结构都是非结构化的，在此数据集中所有单元类型都可以组成任意组合，所以单元的拓扑结构从零维延伸至三维，如图 10-10（f）所示。

在 VTK 中任一类型的数据集都可用非结构化网格来表达,非结构化网格被用于有限元分析、计算几何和图形表示等领域。

10.4.2 数据可视化方法

VTK 中提供了 5 种类型的数据集：

- vtkPolyData
- vtkImageData
- vtkStructureGrid
- vtkRectilinearGridt
- vtkUnstructuredGrid

非结构化点集可用 vtkPolyData 或 vtkUnstructuredGrid 表达,数据集的创建分为如下两步：

- 首先根据数据集的类型定义数据集的几何和拓扑结构
- 创建点属性数据或单元属性数据并与数据集关联

10.4.2.1均匀网格数据（Image Data）可视化

因为均匀网格数据集数据点是规则排列的，所以它的拓扑结构是由数据集的维数定义的，几何结构则由数据集的起始点位置和相邻两点之间的间隔距离定义，间隔距离是指每个相邻点的长度，宽度和高度，起点是指数据集在 3D 空间中左下角点的坐标，在 vtkImageData

中单元和点是根据 x,y,z 依次递增的顺序编码的，总的点数为 nx*ny*nz，单元的总数为(nx-1)*(ny-1)*(nz-1)（其中 nx,ny,nz 是 vtkImageData 的维数）。

下面的示例程序设定了数据集的起始点和间隔距离，并且用球体方程生成数据集每个点的标量数据，球体方程的表达式为：$F(x,y,z)=x^2+y^2+z^2-r^2$

在这里球体的半径设定为 0.4，生成的属性数据用 vtkFloatArray 类的实例存储属性数据，并将它作为数据集的点属性，并且在数据集中提取标量值 $F(x,y,z)=0$ 的表面显示，下图给出了处理可视化流水线的处理过程。

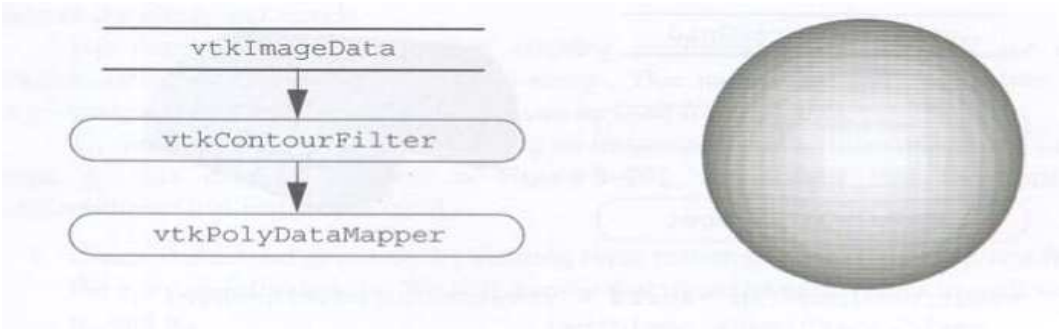


图 10-11 结构化网格数据集对象模型

示例代码请参考 4.5 节体绘制的内容

均匀网格数集的几何和拓扑结构被隐含的定义，所以它的创建过程非常简单，假设定义：vtkImageData *vol= vtkImageData::New()，创建步骤如下：

- 使用 vol->SetDimensions()方法定义拓扑结构；
- 使用 vol->SetOrigin()方法和 vol->SetSpacing()方法定义几何结构；
- 属性数据和数据集相关联

10.4.2.2 多边形数据集（Polygonal Data）可视化

vtkPolyData 数据集中的点数据用 vtkPoints 类表达，单元的拓扑结构由类 vtkCellArray 显式地表达，这个类用来保持单元间的连通性，如图 10-12 所示：

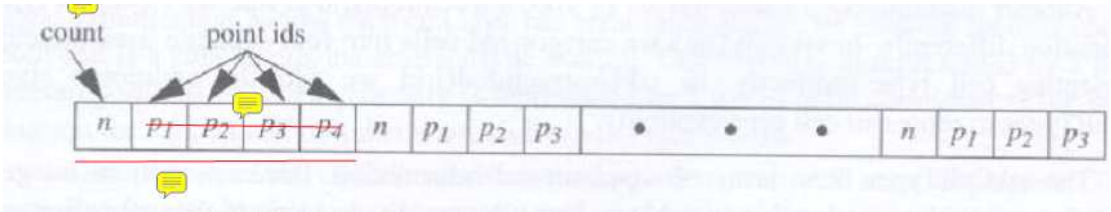


图 10-12 单元的拓扑结构

图中的 n 表示一个单元由多少个点构成，p1、p2…pn 表示构成该单元的顶点 ID 号（在点集中存储的孙顺序号），如图中第一个单元由 4 个顶点构成，第二个单元由 3 个顶点构成，

上图表达的结构中没有直接指出单元类型，vtkPolyData 用了四张独立的表来维护顶点、线、多边形和三角带，顶点表表达 vtkVertex 和 vtkPolyVertex 类型的单元，线表表达 vtkLine 和 vtkPolyLine 类型的单元，多边形表表达 vtkTriangle, vtkQuad, vtkPolygon 类型的单元。

多边形数据是由许多单元构成的，再将这些单元组合形成完整的数据集，下面的示例程序说明了多边形数据集的创建过程，如图 2-13 所示：

图 10-13　多边形数据集的对象模型

示例代码请参考 3.2 节可视化多边形数据的内容。

假设定义 vtkPolyData *cube=vtkPolyData::New()，多边形数据集的创建步骤如下：

- 创建 vtkPoints 点集对象存储数据点，定义几何体，使用 cube->SetPoints()方法将点集对象和数据集关联。
- 使用 vtkCellArray 类的具体子类定义单元的类型，然后根据单元类型分别使用 cube->SetVerts()、cube->SetLines()方法、cube->SetPolys()方法、cube->SetStrips()方法将单元和数据集关联。
- 创建点或单元的属性数据，然后分别使用 pd=cube->GetPointData()方法得到点属性数据的指针 pd，pd=cube->GetCellData()方法得到单元属性数据的指针 pd，使用 pd->SetScalars()方法将标量属性和数据集关联，使用 pd->SetVectors()方法将矢量属性和数据集关联，使用 pd->SetNormals()方法将法线属性和数据集关联。

10.4.2.3线性网格(Rectilinear Grid)可视化

vtkRectilinearGrid 的拓扑结构是规则，几何结构是半规则的，单元在指定数据集的维数时隐式地表达，而几何结构则由顶点来定义，点和单元的编码方式和 vtkImageData 中相同，线性网格数据集的单元类型只能是体素或像素，线性网格数据的处理过程如下图所示：

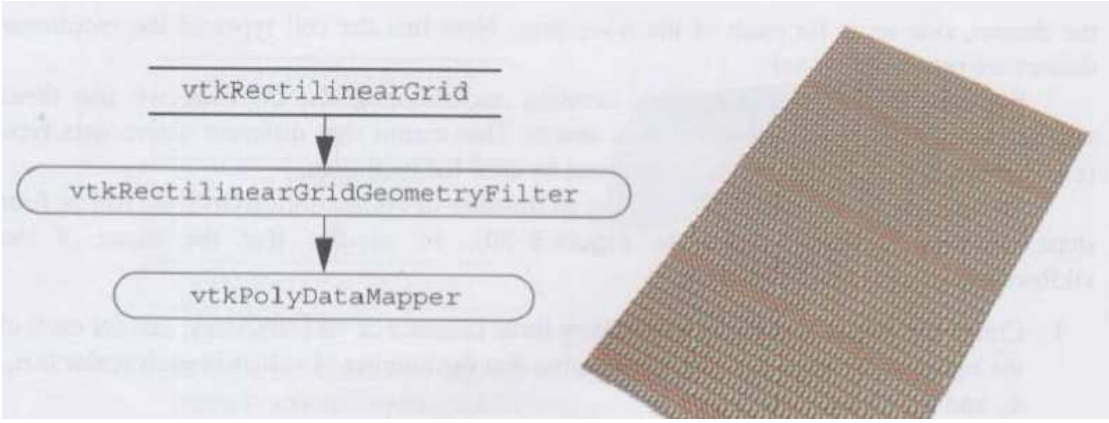


图 10-14　线性网格数据集对象模型图

程序代码示例参考 3.4 节可视化线性网格数据的内容。

假设 vtkRectilinearGrid *rgrid = vtkRectilinearGrid::New()，线性网格数据集的创建步骤如下：

- 创建三个 vtkDataArray 类的子类分别存储数据点的 x,y,z 坐标定义几何体，假设数据点为 n 个，则 x,y,z 的数据分别为 nx,ny,nz。
- 使用数据集对象的 SetXCoordinates()、SetYCoordinates()、SetXCoordinates()方法设定数据集每个点的坐标。
- 使用数据集对象的 SetDimensions()方法设定数据集的拓扑，单元被隐含的表达。
- 创建点或单元的属性数据并和数据集关联。

10.4.2.4结构化网格（Stuctured Grid）可视化

有关结构化网格数据集的描述及示例代码参考 3.3 节可视化结构化网格数据的内容，结构化网格数据的创建过程如下图所示：

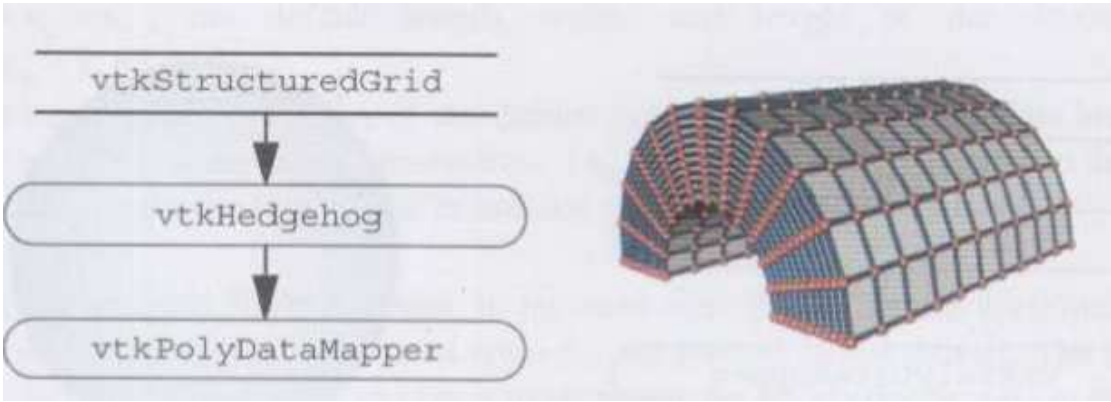


图 10-15　结构化网格数据集对象模型图

结构化网格数据集的长建步骤如下：

- 使用 vtkPoints 类的实例存储数据点的坐标定义几何体，并使用数据集对象的 SetPoints()方法将数据点和数据集关联。
- 使用数据集对象 SetDimensions()方法设定数据集的拓扑，单元被隐含的表达。
- 创建点或单元的属性数据并和数据集关联。

10.4.2.5非结构化网格（Unstructured Grid）可视化

vtkUnstructuredGrid 和 vtkPolyData 相似，不同的是前者可以表达所有的单元类型，在 vtkUnstructuredGrid 中增加了类 vtkCellTypes 来显式地表达单元的类型，如图 10-16 所示：

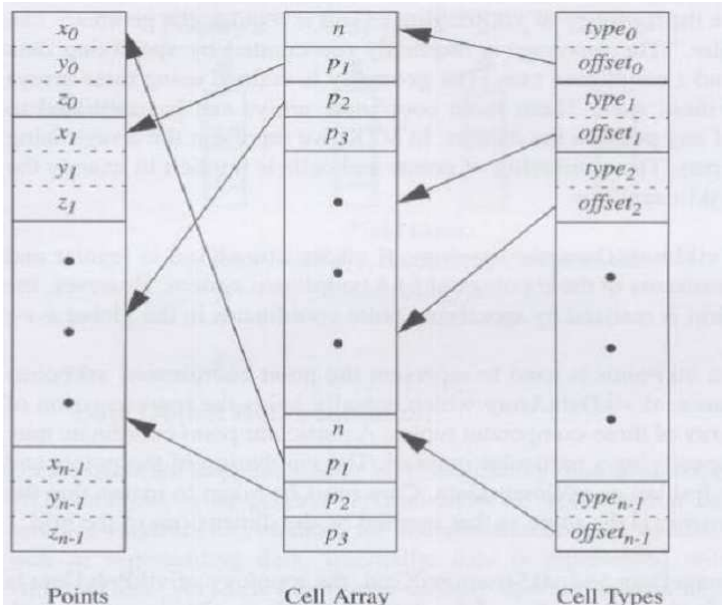


图 10-16 vtkUnstructuredGrid 类的数据结构

非结构化网格数据集可视化示例参考 3.5 节可视化非结构网格数据的内容。

非结构化网格数据集的处理过程如下图所示：

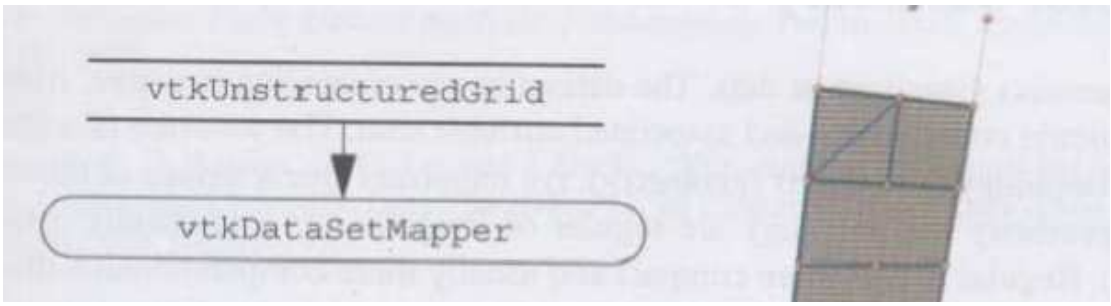


图 10-17 非结构化网格数据集对象模型图

创建非结构化网格数据集步骤如下：

- 使用数据集对象的 Allocate()方法分配数据集内存。
- 创建 vtkPoints 对象存储数据点的坐标定义几何体，并使用 SetPoints()方法将数据点和数据集关联。
- 创建数据集的单元，由于非结花数据集的单元可以有多种类型、多种维度，所以每个单元都要单独创建，使用 InsertNextCell()方法将创建的单元插入到数据集中。
- 创建点或单元的属性数据并和数据集关联。
- 执行 Squeeze()方法完成完整的创建过程。

11 功能算法

本章重点讲述算法，算法是对数据进行处理后，生成能够被绘制的图形基本要素的数据处理方法，不同的可视化领域有不同的算法实现，算法是一个动词，允许我们用可视化的形式表现数据，适当的利用算法我们能将复杂的数据简单化处理，可以很容易的了解数据可视化的含义。

11.1 概述

算法是数据可视化的核心，依据数据集结构和类型的变换对算法进行分类，分为结构变换和类型变换，结构变换是指数据集几何结构和拓扑结构的变换，类型变换是指数据集类型的变换。

根据对数据集几何结构、拓扑结构和属性的影响可将结构变换分为四类：

1) 几何变换只改变几何体本身，不改变几何体的拓扑结构。几何变换只改变物体的空间位置，常用的几何变换包括：平移、旋转、缩放等。
2) 拓扑变换改变了拓扑结构，不改变物体的几何形状，其实质上是一种数据集类型的变换。
3) 属性数据变换将属性数据从一种形式变换到另一种形式，或者利用输入的数据生成新的属性数据，属性数据变换对数据集的结构没有影响。计算矢量的大小和依据高程数据生成标量数据都是属性数据变换。
4) 数据集结构和属性数据都改变的变换为组合变换，如：计算等值线、三维表面等都是组合变换。

依据所要处理的属性数据的类型，对算法进行分类，分为如下几类：

1） 标量算法运算标量数据，如：依据气象资料生成数值等值线等。

2） 矢量算法运算矢量数据，如生成气体流动方向的有向线段，是矢量可视化的例子。

3） 张量算法对张量矩阵进行运算。

4) 模型算法生成数据集的几何和拓扑结构、表面的法矢量或者纹理数据。除去以上算法的所有算法都可归为模型算法。

在下面的论述中，我们主要以属性数据的类型对算法进行分类，分别介绍各种算法。

11.2 标量算法

标量数据用于描述数据集的点和单元属性的值，因为标量数据在现实世界中应用的非常广泛，而且非常容易使用，如和天气相关的温度值、气压值都是标量值，所以对标量值的可视化有多种算法。

11.2.1 颜色映射

颜色映射是一种最常用的标量可视化技术，其将标量数值和某种颜色相对应，在计算机系统中用颜色标识这些数值，主要实现方式是建立颜色映射表，将每个标量值和颜色映射表中的颜色一一对应，通过索引的方式检索指定标量值所对应的颜色。

颜色映射是一种常用的对标量数据进行可视化的技术，映射的实现过程如下：

1) 建立一个颜色表，在该表中存储需要映射的颜色。

2) 将颜色表和标量范围值（最大、最小值）相互对应。

通过以上两个步骤，实现映射，假设颜色表中的颜色数量为 n，标量值的最大、最小值为 max、min，则对任意一个标量值 si 其在颜色表中的索引值 i 可由下式计算：

si<min,i=0

si>max,i=N-1

i=n(si-min)/(max-min)

颜色映射只能映射一种类型的数据信息，但是可以对一维、二维、三维空间物体的某种数据信息进行映射并显示，如映射三维空间体气压数据是一个对三维空间进行颜色映射的例子。

颜色映射的关键问题是要确定好颜色映射表中的颜色条目，设计一个颜色映射表是科学的艺术，在进行颜色映射时，如何设计颜色表示非常重要的，一方面，颜色表的设计不仅要突出需要表达的内容，另一方面表达的形式具有艺术、品质的内涵，将艺术、美学和品质相

结合，将表达内容和形式相互结合，实现和谐的可视化表现，下图给出了流体密度颜色映射图。

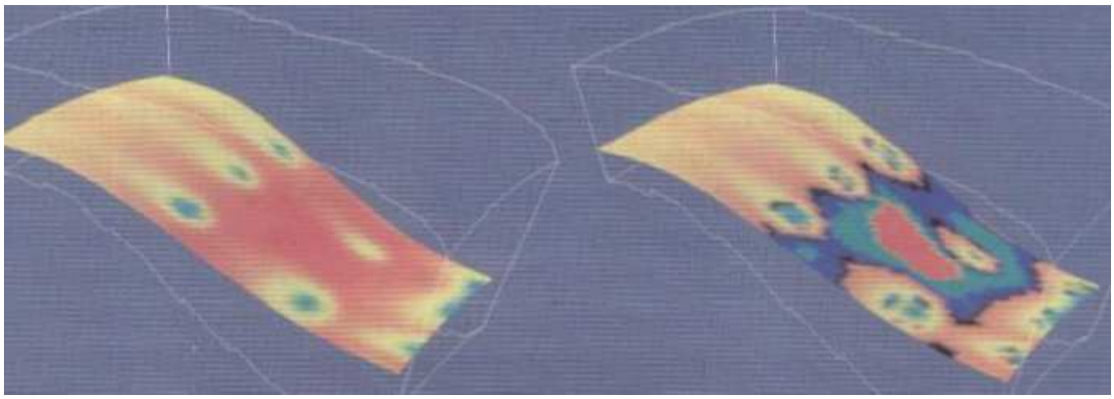


图 11-1　流体密度颜色映射

11.2.2 提取轮廓

提取轮廓是对颜色映射技术的扩展，在一个表面上用不同的颜色标识不同的区域的时候，用相似颜色标识的区域，用人眼很难区分区域的轮廓，指定一个标量值作为等值线常量，利用等值线生成算法可以很容易区分这些区域之间的边界线，划分出区域的轮廓，下图所示的是在二维结构化网格数据中划分标量值为 5 的区域。

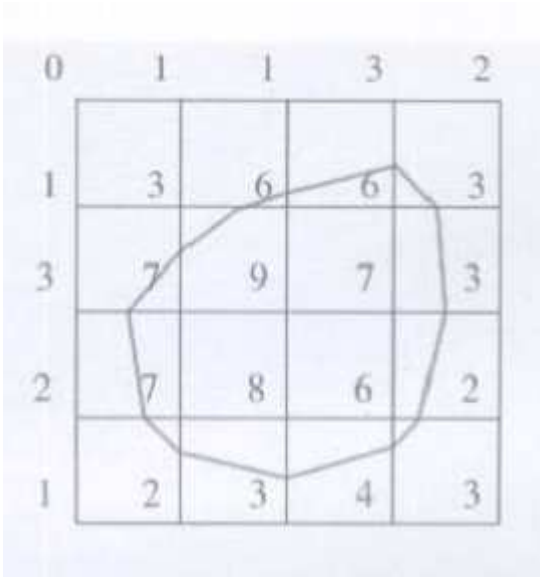


图 11-2　等值线划分区域

三维等值线被称为等值面，等值面也可以用许多多边形图元逼近，流体的温度或压力区域也可以用等值面的算法进行重建。

提取轮廓通常首先选择一个能代表所要划分区域特征的标量值作为等高值，然后再利用

插值算法生成等值线，因为数据集是由有限个点组成的，所以通常采用线性插值技术计算某个点的值，如一个边界在两个端点之间的标量值为 10 和 0，并且我们要生成的等值线的值为 5，那么等值线将经过边缘的中点。

提取轮廓算法主要用于二维、三维的规则网格数据，对于二维的网格数据采用移动四边形算法，确定区域边界线的范围，对于三维的网格数据，采用移动立方体算法确定区域边界线的范围，下图给出了轮廓提取的示例图：

左边的图采用移动四边形法生成等值线，有边的图采用移动立方体法，从三维体数据中提取等值面。

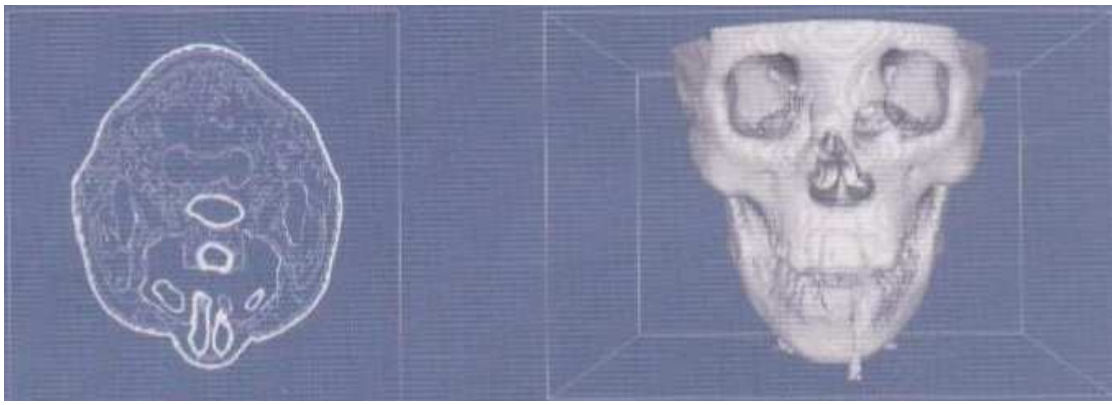


图 11-3　轮廓提取

11.2.3 标量数据的确定

颜色映射和提取轮廓是当前对标量数据进行可视化最简单、有效的两种可视化算法。当对数据可视化时，首先需要考虑使用这些算法，对于有些数据不能直接用这直接用这两种技术进行可视化时，必须将这些数据转化为能可视化的标量数据形式，例如，对于地形数据，每个数据点由 x、y、z 坐标构成，假设 Z 坐标表示高程值，我们如果用不同的颜色标识高程，那么我们就提取 z 值作为可视化的标量值。

在通常情况下，使用标量可视化算法，仅仅需要建立产生惟一标量值的关系式，总之，颜色映射和轮廓提取是可视化标量数据最简单、有效的技术。

11.3 矢量算法

矢量是有大小和方向的量，矢量数据在我们日常生活中经常见到：如水的流动、运动的速度等，在 VTK 中提供了如下几种矢量数据可视化的方法。

11.3.1 方向线和方向符号

一种矢量可视化技术是绘制矢量方向线，如图 11-4（a）所示，因为这种技术绘制的方向线和刺猬身上的刺很相似，所以又被称为刺猬。

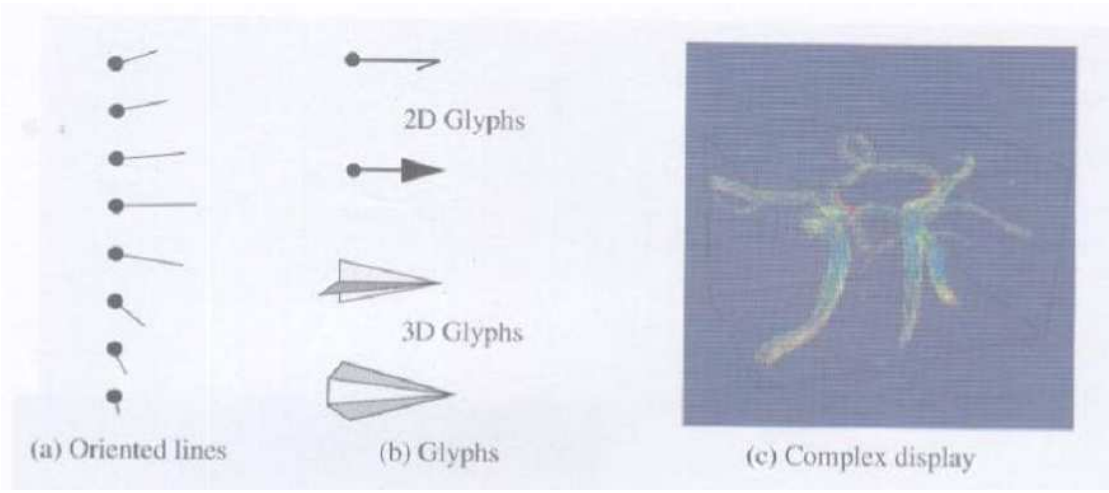


图 11-4 矢量可视化技术

这种矢量可视化技术有许多的变化，如添加一个箭头标识线的方向，或根据矢量的大小（及其它标量值，如温度、压力）把矢量线绘制成不同的颜色，也可以用方向符号代替方向线，用 2D 或 3D 几何体，如：三角形、圆锥体表示符号，如图 11-4（b）所示。

因为三维矢量被投影成二维图像，所以它的位置和方向很难辨别，使用大量的矢量会使点的表达变得混乱，从而使可视化变得毫无意义。图 11-4（c）显示了表达的人体劲动脉区域的 167，000 个三维矢量，很明显，矢量场的细节是无法辨别的。

11.3.2 变形

矢量数据总是与“运动”相关联，运动是速度或位移的形式表达，使用可视化的手段表达矢量数据的有效方法是弯曲矢量场或变形几何体，例如要将流体的流动进行可视化，那么就要对插入到流体中的垂直线按流动方向进行扭曲创建流体基线。

图 11-5 显示了两个矢量变形的例子，（a）显示了一个震荡梁，（b）显示了一个结构化网格数据集中的变形平面，这些平面根据流动动量而变形，流体向前和向后的流动关系在平面的变形中明显得看出来。

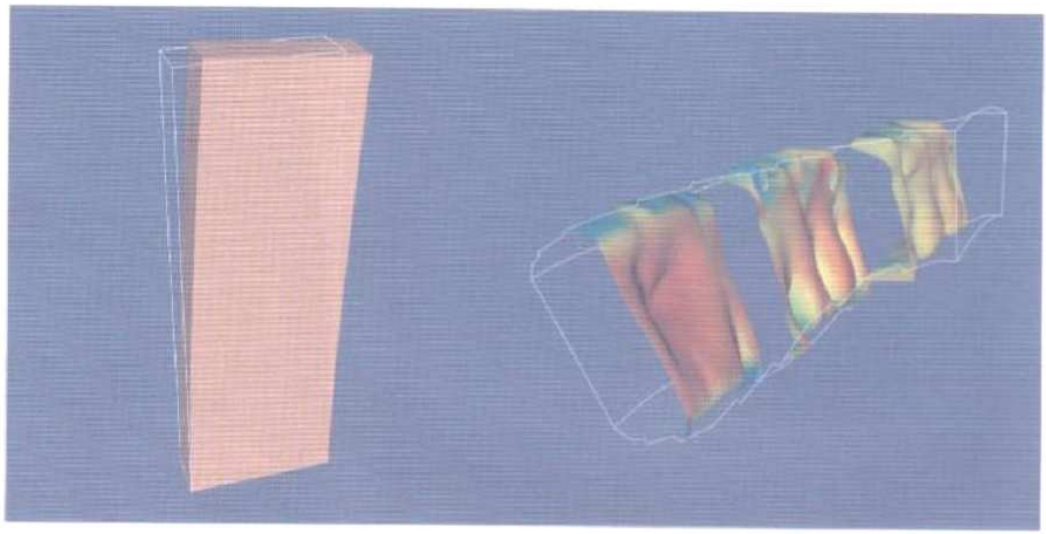


图 11-5 矢量场变形几何体

矢量场必须按比例缩放，控制扭曲程度，过于小的变形可能不可见，但同时太大的扭曲可能导致结构反转或自身交叉。

11.3.3 位移绘制

把各个时间的运动变化位置连成线，逼近运动的轨迹，对象表面的矢量位移可通过位移绘制可视化，位移绘制表达一个对象在其表面垂直正交方向的运动，对象的运动是由的矢量场引起的，在应用中矢量场就是位移或应变场。

矢量位移绘制一般把矢量数据转换成标量数据，把每个矢量和其表面的法向量点乘生成标量数据，然后采用标量可视化的手段进行绘制，若点乘结果为正，则点的运动方向与法向量一致，若为负，则点的运动方向背离法向量，图 11-6（a）所示为标量计算的方式。

例如，在震荡分析中，比较注重结构的特征值和特征向量，为了更好的理解震荡模型的外表变化，用位移绘制的手段来表明运动区域，当震荡从正位移变化到负位移时，经过位移为零的区域，这一区域称为震荡线，图 11-6 显示了一个震荡梁的位移变化情况。

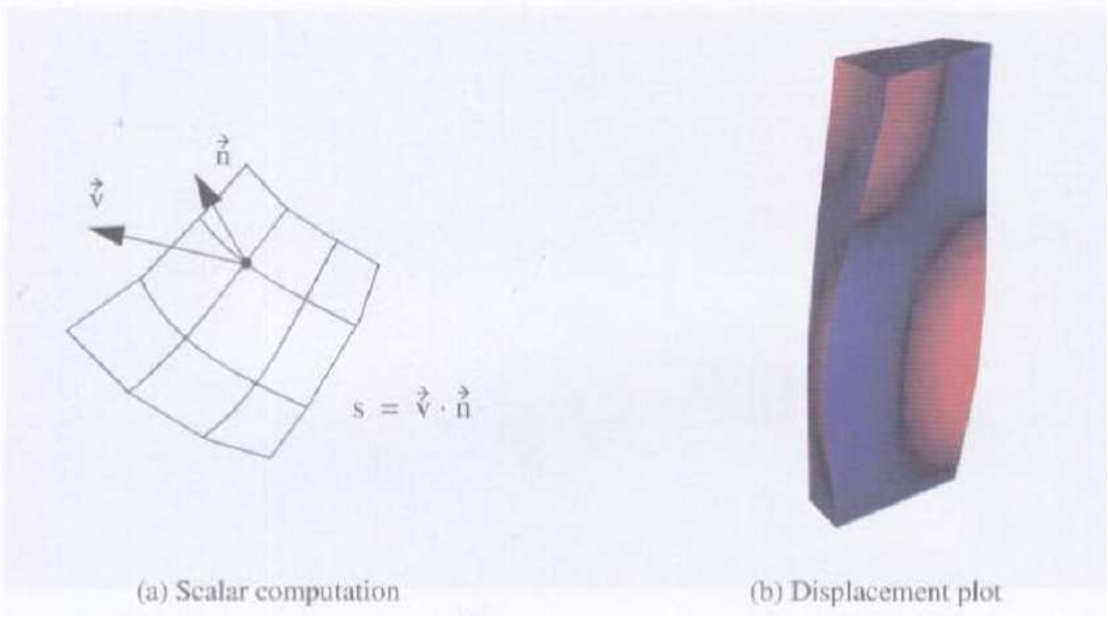


图 11-6 矢量位移绘制

图 11-6（b）显示一个矩形振荡梁的震荡线，这是一个二级扭振形式模型，图中黑色的部分是震荡线，蓝色的部分是位移最大的区域。

11.3.4 时间动画

到目前为止所描述的算法可以看作是在一个小时间步长内点或对象的运动，方向线是点在某一时刻的运动的近似值，例如，如果知道运动的速度，点的位移可由式 11-1 计算：

$$dx = \vec{v}dt \qquad \text{式（11-1）}$$

因此可以计算出点在多个步长时间段内的位移，进而描绘出点随时间变化的规律，如图 11-6 所示。

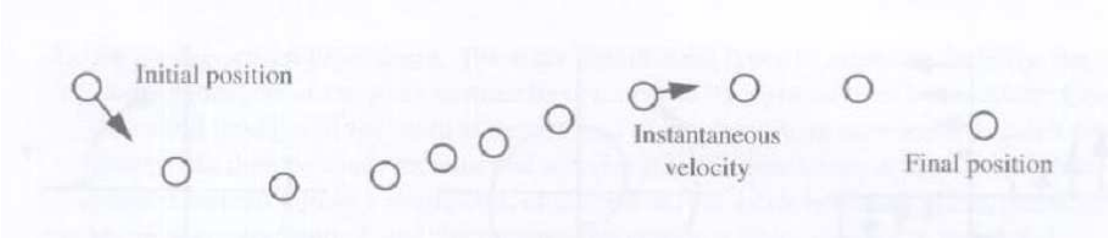


图 11-6　点 C 的时间动画（点的间隔不同，但是时间的增量是个常量）

11.3.5 流线

把图 11-6 中各个时间段质点的运动位置连成线，形成质点的运动轨迹。在流体流量中，为矢量场提供三个与线表达相关的方案。

- 轨迹流体随时间运动的路径。
- 纹线是某一时刻经过某点的轨迹的集合。
- 流线是满足表达式 11-2 所示方程的曲线积分。

$$s = \int \vec{v} ds \qquad s = s(x,i) \qquad \text{式 11-2}$$

如果流量一定，流线、纹线及轨迹三者是等价的。流量随时间变化时，流线只是瞬时存在的。可视化系统一般都提供计算轨迹的工具。但是，如果时间是固定的，这些工具可以用来计算流线。因些，通常在矢量场中追踪运动轨迹的方法用流线实现。

图 11-6 显示的是一个小厨房内的四十条流线。房间有两个窗户一个漏风的门和一个热炉。漏进的风的温度变化一起引起空气的对流。流线的开始位置通过创建一个倾斜或曲线来定义。倾斜是一条直线，这些流线浅析地表现了流体场的特征。

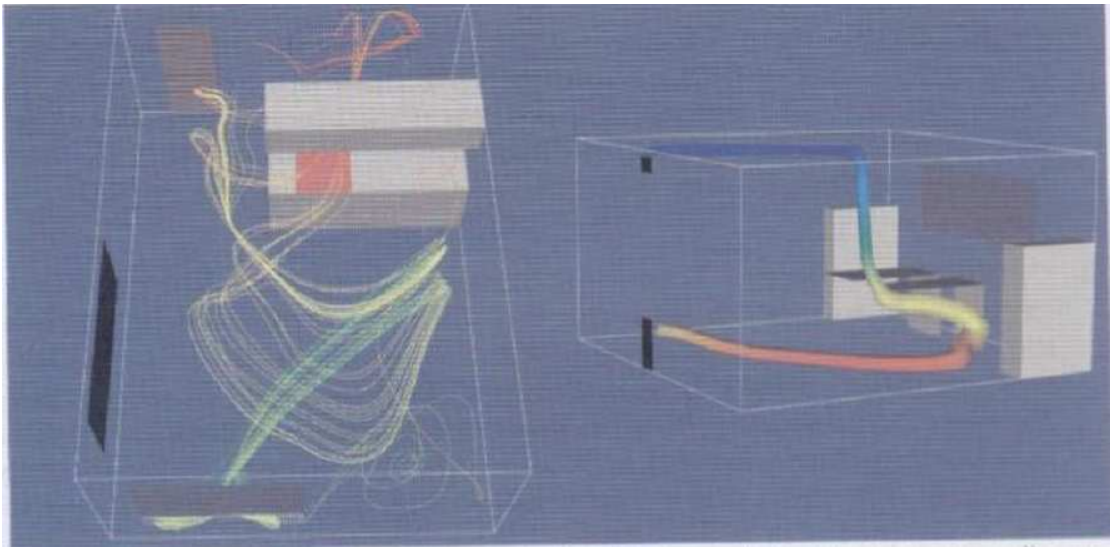


图 11-6　厨房中的流线

（这四十条流线从窗户下方倾斜处开始，一部分经过热炉并且向上对流）

11.4 模型算法

11.4.1 源对象

源对象是可视化流水线的开始，源对象用于生成可视化的几何体（如圆柱体、圆锥体、立方体等），或者用于从数据文件读取数据，源对象也可用于创建属性对象。

11.4.2 隐函数

隐函数的形式为：$F(x,y,z)=c$，其中 c 是任意常量。

隐函数的三个重要特性：

- 简单几何体描述：隐函数是用于描述几何形状的工具，描述的几何形状包括：面、球体、柱体、二次曲面等。
- 区域分离：根据隐函数值的大小，将区域分为内、外及在区域上，判定空间点所属的区域。
- 标量生成：将一个空间位置点转换成一个标量值。

下面给出半径为 R 的球体方程的隐函数：$F(x,y,z)=x^2+y^2+z^2-r^2$

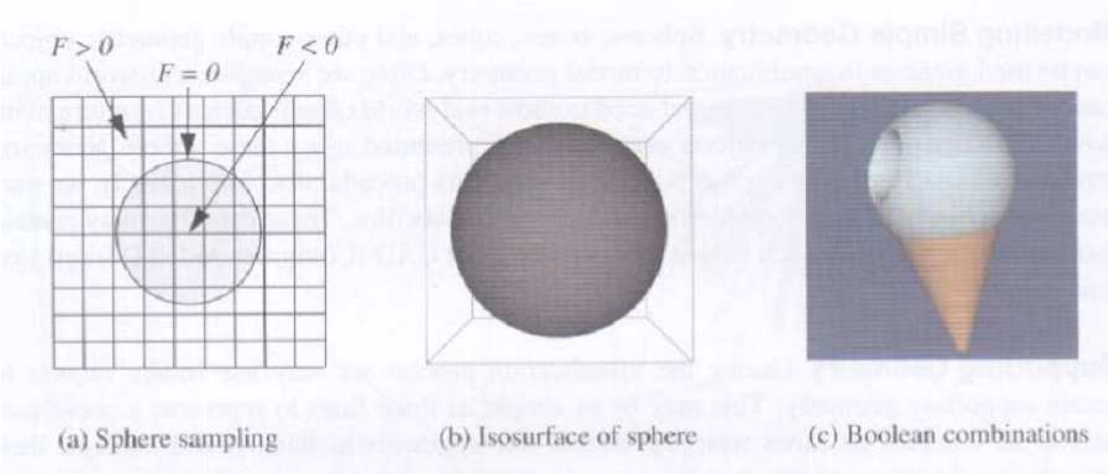


图 11-7 球体方程

该球体方程定义了三个区域：$F(x,y,z)=0$ 表示球体表面，$F(x,y,z)<0$ 表示球体内部，$F(x,y,z)>0$ 表示球体外部。

隐函数有多种用法，包括几何建模、数据选择、复杂数学描述可视化。

11.4.2.1 对象建模

隐函数可以用来模拟几何对象。例如，用隐函数模拟等值面，在数据集中抽取样例 F，生成等高值为 Ci 的等值面，最后得到函数的多边形表达。

隐函数与布尔运算并、交、差联合可创建复杂的对象，函数 F(x,y,z)与 G(x,y,z)在点（x0,y0,z0）处的布尔运算如下所示：

$$F \cup G = \min(F(x_0, y_0, z_0), G(x_0, y_0, z_0))$$

并

$$F \cap G = \max(F(x_0, y_0, z_0), G(x_0, y_0, z_0))$$

交

$$F - G = \max(F(x_0, y_0, z_0), -G(x_0, y_0, z_0))$$

差

11.4.2.2数据选择

可以用隐函数选择处剪切数据，用区域分离特性来选择数据，用隐函数选择或提取数据即选择位于函数的特殊区域内的单元或点，通过求点的值和检查结果的符号，可以确定点是否在区域内，如果一个单元在区域内，那么单元内的点全部在区域内。

11.4.3 隐式建模

隐模型与隐函数的区别是标量值由距离函数生成，图 11-8 显示了到点、线、三角形的距离，距离计算主要计算到指定图元的欧式距离（如到某个点、线或面的距离），由于距离函数是一个单调函数，我们可以指定不同的距离值，构建一系列偏移表面来逼近真实的表面。

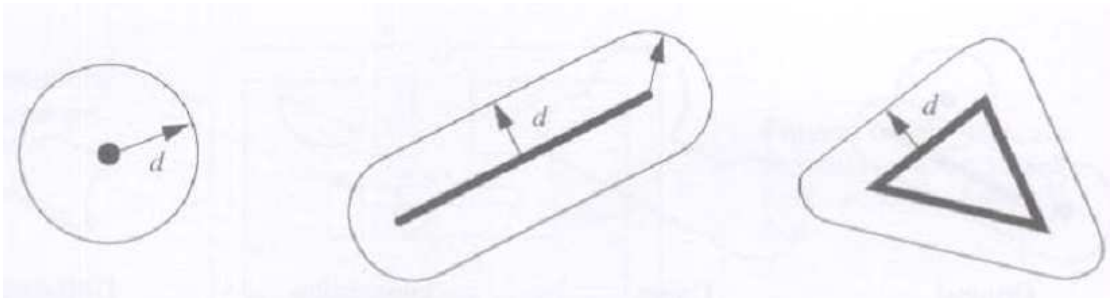


图 11-8　到点、线、三角形的距离

11.4.4 符号化

符号化，是对可对各种数据类型进行可视化的通用技术，一个符号就是一个由输入数据定义的对象，这个对象可能是几何体，数据集或是图像，有关符号化的过程请参考 3.1.3 节

的内容。

11.4.5 剪切

通常使用一个表面剪切一个数据集，并显示剪切后的结果值，我们称这种技术为数据剪切，数据剪切需要两方面的信息：被剪切的数据集和剪切面，剪切面一般是使用隐函数定义，一般的应用使用一个平面对数据集进行切片，然后使用颜色映射或矢量可视化的方法显示剪切结果。

隐函数的一个特性是可以将一个空间位置点转换成一个标量值，我们可以将这个特性与等值面算法相结合，生成剪切面，其基本思想是：对数据集的每个点或单元生成一个标量值，然后用等值面算法提取剪切范围值，生成剪切面。

剪切算法的过程如下：对于组成单元的每个点，用隐函数 $F(x,y,z)$ 计算其值，如果所有的点都为正值或负值，则剪切面不通过该单元，反之，剪切面通过该单元，利用隐函数 $F(x,y,z)=0$ 计算函数值为零的点所组成的面生成等值面，同时利用线性插值算法计算面的属性值。

11.5 综合应用

11.5.1 过程对象设计

以上所介绍的算法，在 VTK 中，由处理对象提供，处理器对象或许是源对象、过滤器对象、映射器对象，这一部分主要介绍这些对象的实现方式。

11.5.1.1源对象

源对象是没有输入并且有一个或多个输出的对象，功能模型用于指定源对象被处理后输出的数据集类型。

11.5.1.2过滤器

过滤器对象有一个或多个输入及输出，依据类层次图可以确定过滤器输入输出的数据类型，类层次图在开发时是非常用价值的，它是可视化流水线架构的详细描述。从 3-8 类功能模型图可以看出，vtkContourFileter 接收一般的数据集对象作为输入，并且生成多边形数据

作为输出。

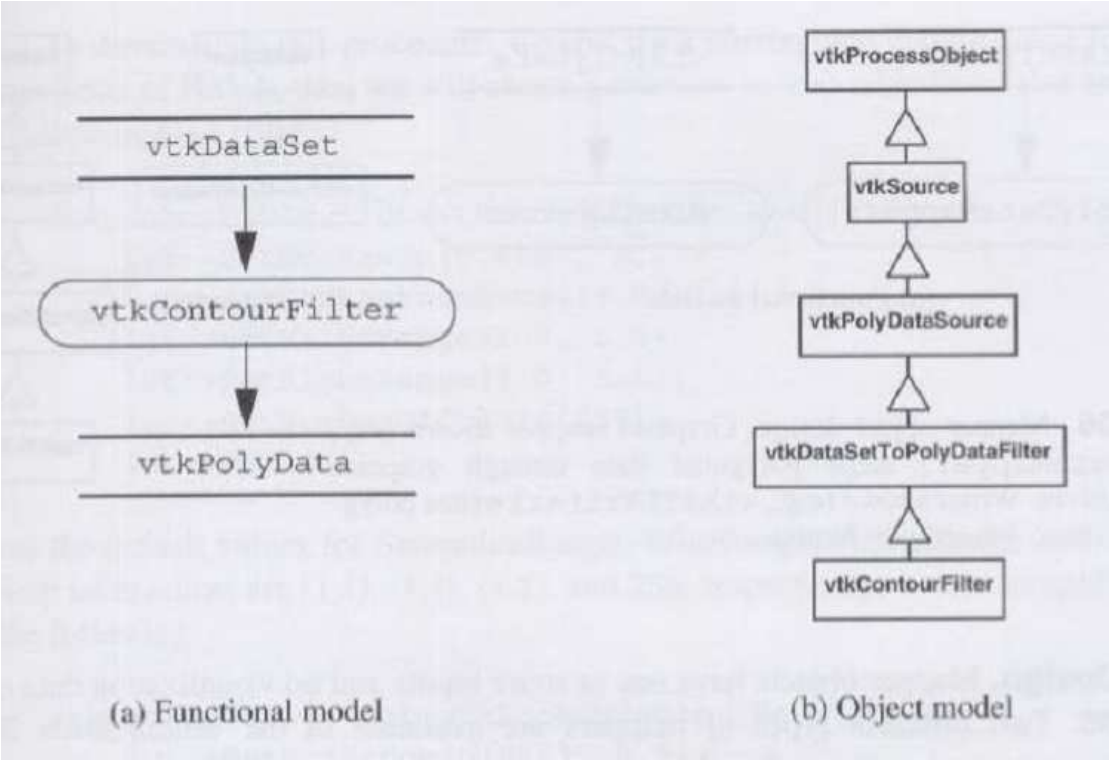


图 11-9 过滤器对象设计

从 11-9 的类层次图可以看出，vtkContourFilter 类是一个源对象，其中 vtkPolyDatasource 对象说明 vtkContourFilter 将输出多边形数据对象， vtkDataSetToPolyFilter 说明 vtkContourFilter 将数据集对象作为输入。

在 C++中使用过滤器时，一般在编译时对输入、输出的数据类型进行检查，对于不同的过滤器对象，派生于不同的类，如：接收多边形数据作为输入数据的过滤器，一般派生于 vtkPolyDataToPolyDataFilter 类，接收非结构化网格数据作为输入数据的过滤器，一般派生于 vtkUnstructuredGridToPolyDataFilter,由此看出，从类层次图上，可以很容易的知道每个过滤器能够处理的数据类型和输出的数据类型。

11.5.1.3 映射器

映射器对象有一个或多个输入，没有可视化数据输出，在 VTK 中有两种不同类型的映射器：图形映射器和复写器，图形映射器将数据对象的几何结构和属性数据映射到图形系统进行绘制，复写器将数据对象成文件的形式存储。

每个映射器的子类都必须实现 Render()方法，这个方法的作用是将输入的数据映射到图形系统，复写器对象必须实现 WriteData()方法，这个方法用于将输入数据存储到数据文件中。

11.5.2 颜色映射

在 VTK 中，颜色索引表使用 vtkLookTable 类创建，该类允许我们创建 HSVA 颜色模式的索引表（色调、饱和度、值和透明度），有两种创建颜色索引表的方式：

- 第一种方式：设定表中颜色数目，定义一对 HVAS 颜色值，当调用 Builde()方法时，将在这对颜色值之间建立线性颜色色阶，色阶的数目和设定的颜色数目相等。如定义一对红、绿颜色，设定颜色数目为 5，将在红到绿颜色之间，生成 5 种颜色数目的颜色。
- 第二种方式：在有些情况下，自己定义颜色表中颜色的数目，建立颜色表，然后再颜色表中插入指定的颜色，使用这种方式不能创建颜色色阶，必须自己定义颜色表中的全部颜色数目。

下面的代码说明了用第一种方式创建颜色索引表的过程：

```
vtkLookupTable *lut=vtkLookupTable::New();
lut->SetHueRange(0.667, 0.0);     //设置调色范围
lut->SetSaturationRange(1.0,1.0);
lut->SetValueRange(1.0,1.0);
lut->SetAlphaRange(1.0,1.0);
lut->SetNumberOfColors(256);     //设置索引表中颜色数为 256
lut->Build();     //从红到蓝生成 256 种颜色
```

下面的示例代码说明了用第二种方式创建颜色索引表：

```
vtkLookupTable *lut=vtkLookupTable::New();
lut->SetNumberOfColors(3);
lut->Build();
lut->SetTableValue(0,1.0,0.0,0.0,1.0);
lut->SetTableValue(0,0.0,1.0,0.0,1.0);
lut->SetTableValue(0,0.0,0.0,1.0,1.0);
```

在 VTK 中，颜色索引表和映射器相关联，如果没有指定颜色索引表，因射器将自动创建一个从红到蓝过渡颜色的颜色索引表，如果我们创建了自己的颜色索引表，那么使用映射器的 SetLookupTable()方法，设定颜色索引表和映射器相关联。

使用颜色索引表需要注意以下几点：

- 映射器用于将颜色索引表和数据集的属性数据相关联，如果没有指定数据集的属性数据，那么颜色索引表将不起作用，在这种情况下，如果要设定数据集的颜色，只能使用 vtkProperty 类的 SetColor()方法。
- 如果想要禁用颜色映射，使用映射器的 ScalarVisbilityOff()方法，禁用颜色映射。
- 使用映射器的 SetScalarRange()方法设定将要进行颜色映射标量值的范围。

11.5.3 隐函数

隐函数在可视化中主要用于创建几何体、或者在数据集种选择、剪切数据，VTK 包含了一些隐函数，如平面(vtkPlane)、球体（vtkSphere)等，利用 vtkImplicitBoolean 类可以对几何体进行布尔操作。

隐函数的类图如 11-10 所示，隐函数的所有子类都实现了 Evaluate()和 Gradient（）方法，Evaluate()返回点（x,y,z)处的函数值，Gradient()返回点(x,y,z)处的矢量梯度值。

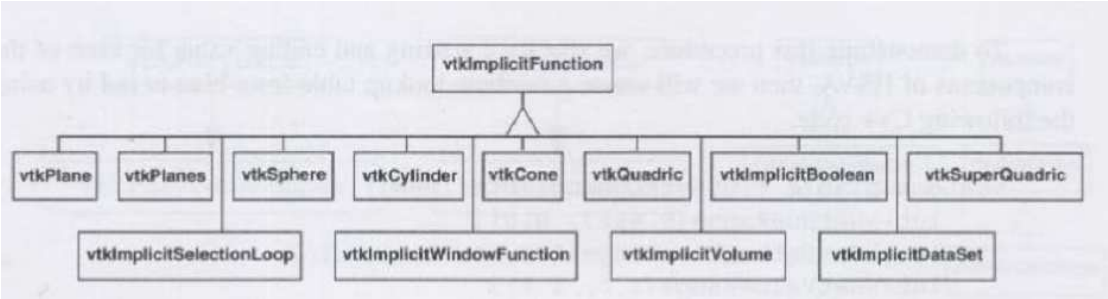


图 11-10 隐函数类图

11.5.4 提取轮廓

在 VTK 中，用 vtkContourFilter 类实现轮廓提取，该类接收任何的数据集类型作为输入，因为该类处理每个单元，所以每个单元必须包含提取自身轮廓的方法，该类提取的轮廓类型包括：点、线和等值面，在 VTK 中还存在另外一个轮廓提取过滤器类 vtkMarchingCubes，该类只接收 vtkImageData 数据集类型作为输入数据，这两种过滤器在数据处理的时间消耗上是不相同的。

有关该类的使用请参考 3.1.2 节轮廓提取得相关内容。

11.5.5 剪切

在 vtk 中使用 vtkCutter 执行对数据集的剪切操作，SetValue()、GenerateValues()方法用于设定剪切数据值，该类还需要一个隐函数用于对数据集中的点进行评估，对于单元的剪切，使用单元的 Contour 方法。

11.5.6 符号化

在 VTK 中，vtkGlyph3D 类对符号化处理提供了有力的支持。

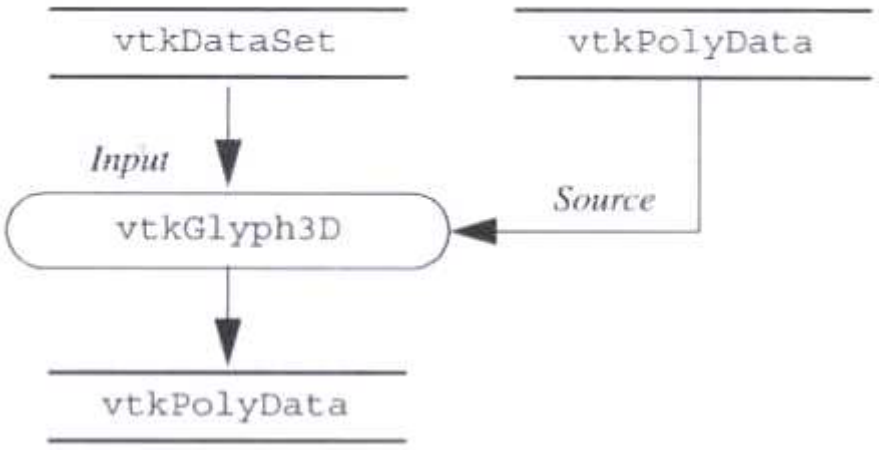


图 11-11　vtkGlyph3D 类的数据流

vtkGlyph3D 类是一个对象有多个输入的一个例子，如图 11-11 所示。一个输入是由 SetInput()方法指定，表示被符号化的点和相关的属性数据。第二个输入由 SetSource()方法指定，定义了一个几何体表示符号。

11.5.7 流线

在 VTK 中，对质点的运动用 vtkStreamer 类描述，该类是一个基类，提供了基本的质点运动路径描述功能，对于质点在矢量场中的运动，该类的子类提供更强的功能，如：vtkStreamLine 类能显示质点的运动轨迹，如果用 vtkGlyph3D 类对运动轨迹进行符号化处理，还可以显示质点的运动方向和路径类层次图如 11-12 所示。

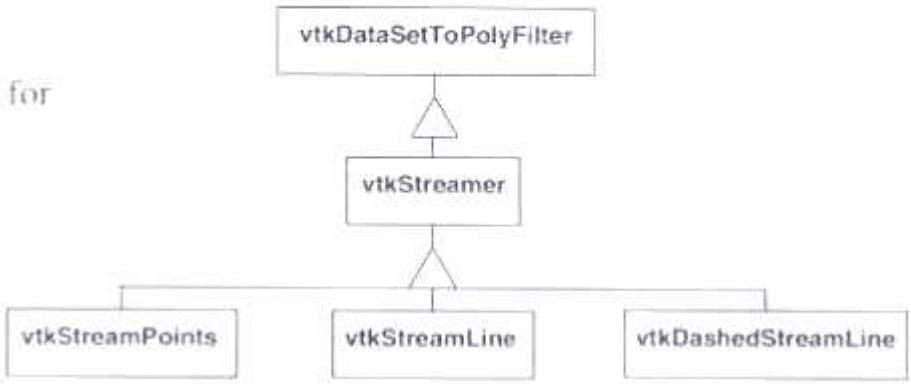


图 11-12　　vtkStreamLine 类层次图

vtkStreamer 类的积分方法是一个虚函数，它可以被派生类按需求重载。

在 VTK 中，矢量积分是对数据集的单元进行的，所以，对于任何类型、任何维数的单元都可以进行矢量积分运算，如果数据集单元维数是 2 维的或是 1 维的，那么质点运动的路径将在 2 维的面或 1 维的线上。

11.5.8 抽象过滤器

属性变换只能对数据集的属性数据进行操作，和数据集的几何和拓扑结构无关，因此，理论上可以认为实现属性变换的过滤器可以接收任何类型的数据集作为输入，并且也可以输出任何类型的数据集，不幸的是，因为 vtkDataSet 是一个不能实例化的抽象类，因此，在 VTK 中每个过滤器都有指定数据集的类型作为输出。

幸运的是，在 VTK 中提供了虚构造函数的形式，解决这个问题，在数据集类和单元类中都提供了虚构造函数的实现方法。

使用虚构造函数，我们可以实现过滤器输出抽象的数据集类型，如 vtkDataSet 类，使用 NewInstance()方法，可以生成一个抽象过滤器，这个过滤器可以接收任何类型数据集作为输入，并且可以将抽象的数据集作为输出。