# 第四章 VTK基础

这一章通过一系列典型的例子介绍VTK的功能。重点在于常用的方法和对象，以及对象间的结合。同时，介绍了VTK中的重要概念和应用。本章的目的是让读者对VTK有个大概的了解，并没有涵盖VTK的所有特性。读者可以参考在线文档或者类.h文件来学习每个类的其他功能。

本书的大多数例子是用Tcl程序语言实现的（注：我们在本书的翻译过程中，把大部分的Tcl代码转成了C++代码，你可以在VTK的源码目录里找到相应的Tcl文件）。这些例子可以用C++，Java和Python实现，这些编程语言之间可以直接地进行转换。（参考“语言之间的转换”一节）。由于C++在数据结构和指针操作方面具有显著的优势，有关这方面的例子会采用C++语言实现。

这里列出的每个例子都包括了相关的代码和补充说明的图像。如果该例子在VTK源文件目录中存在的话，我们会列出该例子文件的名字，所以你不需要手动去输入这些代码。建议你自己去运行这些例子，理解里面的代码，用不同的参数进行实验。你也可以用不同的方法或者类去实现同样的功能。通常，VTK都会提供几种不同的方法来获得类似的结果。注意，脚本文件通常会因发布的源码的改变而作修改，这是为了简化概念或者删除多余的代码。

学习像VTK这样的面向对象的系统首先需要理解抽象编程，然后要熟悉对象库及其方法。因此，建议读者先复习一下第三章“系统架构”中有关抽象编程的概念。这一章的例子可以让读者对VTK对象有一个很好的认识。

## 4.1 创建简单模型

使用VTK的步骤一般为：读入/生成数据，对数据进行Filter，数据的渲染和交互。这部分的主要内容就是数据的读取和生成。

获取数据有两种方法，数据可能存在于读入系统内存的文件（或数据流）中，也可能由程序生成（通过算法或者数学表达式等形式）。在可视化管道中初始化数据处理的对象称为源对象（参考图3-5）。生成数据的对象称为程序（源）对象，读入数据的对象称为读（源）对象。

### 程序源对象 (Procedural Source Object)

我们从绘制简单的圆柱开始。以下例子代码（VTK/Examples/Rendering/Tcl/Cylinder.tcl）展示了可视化管道和渲染引擎的一些基本概念。图4-1显示了该Tcl脚本的运行结果。

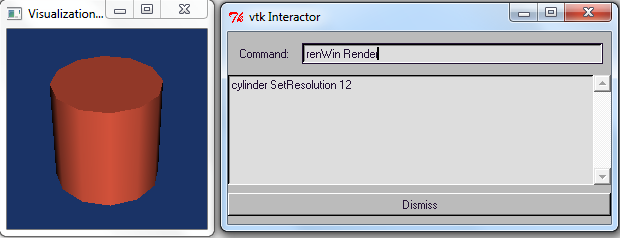


图4-1 使用Tcl/Tk编写的解释性应用程序

首先我们在该脚本程序的最开始调用了Tcl命令来加载VTK工具包（package required vtk），并且创建了带用户交互界面的解释器（package required vtkinteraction），通过该交互界面可以让你在程序运行时输入命令。另外，脚本也加载了vtktesting模块，该模块定义了一系列的颜色，其中“番茄色”在该脚本中会使用到。

Package required vtk

Package required vtkinteraction

Package required vtktesting

接着，创建一个程序源对象：vtkCylinderSource，该类创建一个横截面为多边形的柱体，柱体的输出通过方法SetInputConnection()设置为vtkPolyDataMapper对象的输入。接着创建一个actor对象（要渲染的对象），设置定义几何信息的mapper到这个actor里。注意类对象在Tcl中的构造方式：类名后面紧跟对象的名字。

vtkCylinderSource cylinder

cylinder SetResolution 8

vtkPolyDataMapper cylinderMapper

cylinderMapper SetInputConnection [ cylinder GetOutputPort ]

vtkActor cylinderActor

cylinderActor SetMapper cylinderMapper

eval [ cyliderActor GetProperty ] SetColor Stomato

cylinderActor RotateX 30.0

cylinderActor RotateY -45.0

为了证明C++代码的实现与Tcl（及其他解释性语言）的相似性，与以上例子相同的C++实现代码也在下面列出，可以在VTK/Examples/Rendering/Cxx/Cylinder.cxx中找到该例子的C++代码。

vtkCylinderSource \*cylinder = vtkCylinderSource::New();

cylinder->SetResolution(8);

vtkPolyDataMapper \*cylinderMapper = vtkPolyDataMapper::New();

cylinderMapper->SetInputConnection( cylinder->GetOutputPort() );

vtkActor \*cylinderActor = vtkActor::New();

cylinderActor->SetMapper( cylinderMapper );

cylinderActor->GetProperty()->SetColor ( 1.0000, 0.3882, 0.2784 );

cylinderActor->RotateX( 30.0 );

cylinderActor->RotateY( -45.0 );

回顾之前的内容可知，源对象位于可视化管线的开始，mapper对象（或者具有mapper功能的prop对象）处于管线的终端。因此，该例子的管线包含两种算法（即源source和mapper）。VTK的管线使用惰性赋值机制，即使管线已经连接，如果没有请求获取数据，程序也不会生成数据及对数据进行处理。

接下来，为了渲染actor需要创建图形对象。vtkRenderer的实例ren1协调渲染窗口renWin的视口（viewport）的渲染过程。渲染窗口交互器实例iren是一个3D的widget，可以控制三维渲染场景的相机。

#Create the graphics structure

vtkRenderer ren1

vtkRenderWindow renWin

renWin AddRenderer ren1

vtkRenderWindowInteractor iren

iren SetRenderWindow renWin

注意，我们通过渲染窗口类的方法AddRenderer()把renderer和渲染窗口关联起来，接着使用renderer的方法AddActor()把要渲染的actor加入到renderer中去。

# Add the actors to the renderer, set the background and size

ren1 AddActor cylinderActor

ren1 SetBackground 0.1 0.2 0.4

renWin SetSize 200 200

SetBackground()方法用取值范围为(0, 1)的RGB（红，绿，蓝）值设置渲染窗口的背景色，SetSize()以像素为单位来确定窗口的大小。最后，将GUI交互器与用户自定义的渲染交互窗口交互器的方法关联起来。（当鼠标的焦点处在渲染窗口时，用户自定义的方法可以通过按击键盘u键激活。可参考本章的“使用VTK交互器”一节，或者参考第三章的“用户方法、观察者和命令”一节）。调用Initialize()方法进入事件循环，Tcl/Tk的命令wm withdraw .可以确保在程序开始运行时interacter widget .vtkInteract不可见。

# Associate the “u” keypress with a UserEvent and start the event loop

iren AddObserver UserEvent ( wm deiconify , vtkInteract )

iren Initialize

# suppress the tk window

wm withdraw

当以上脚本程序运行时，因为渲染引擎会请求数据，所以可视化管线就会执行。（窗口的expose事件会迫使渲染窗口自行渲染。）只有输入数据改变才能令管线执行的数据更新。读者可根据需要调用renWin Render手动控制管线的执行。

例子运行后，你可以尝试：首先，用鼠标与渲染窗口进行交互。然后，可以调用cylinder SetResolution 12改变柱体的分辨率（即柱体横截面的多边形的边数）。可以编辑脚本然后重新执行这个例子；也可以按击键盘u键，就会弹出图4-1所示的基于GUI的命令解释器，可以在上面输入命令。这些改变只有在你重新请求数据时才会生效，所以在输入cylinder SetResolution 12命令以后，还必须输入renWin Renderer命令，或者用鼠标点击一下渲染窗口，所做的改变才会生效。

### 读取源对象Reader Source Object

这个例子与前面的例子类似，除了用读取数据文件来生成数据替代由程序生成数据。立体成型(stereo-lithography)数据文件(后缀为.stl)是使用二进制STL数据格式来表达多边形数据。（参考图4-2和Tcl脚本VTK/Examples/Rendering/Tcl/CADPart.tcl）

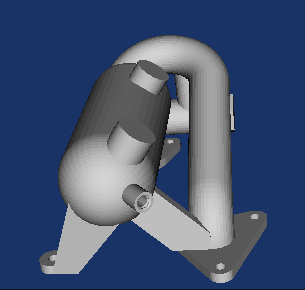


图4-2 读取源对象

vtkSTLReader part

part SetFileName $VTK\_DATA\_ROOT/Data/42400-IDGH.stl

vtkPolyDataMapper partMapper

partMapper SetInputConnecteion [ part GetOutputPort ]

vtkLODActor patrActor

partActor SetMapper partMapper

注意类vtkLODActor的使用。为保证交互时的顺畅性，这种类型的actor会更改自身的显示形式，缺省情况下会创建一个带线框的点云替代交互时actor的显示(Its default behavior is to create a point cloud and wireframe, bounding-box outline to representation the intermediate and low-level representations)。（参考本章“Level-Of-Detail Actors”一节。）本例中所用到的模型数据量较小，在现在的大多数机器上运行的话，还是显示模型完整的形式（图4-2所示）。

文件读取类不会监听输入文件是否发生变化以及是否要重新执行管线。例如，如果文件42400-IDGH.stl作了改变，管线不会重新执行。可以手动调用方法Modified()以使这些更改生效。这个方法会使得该filter重新执行以及更新它后续的数据。

VTK限制了部分建模功能，如果你想用VTK编辑或者操作复杂的模型（例如，用solid modeler或其他建模工具建立的模型），一般使用文件读取（参考第十二章关于数据读取方面的内容）来导入数据。

## 4.2 使用VTK交互器

对数据可视化以后，通常都希望与这些数据进行交互，VTK提供多种数据交互的方法。第一种方法是使用vtkRenderWindowInteractor类，第二种方法是通过绑定特定的事件定制自己的交互类。如果是使用解释性语言的话，还可以在程序运行输入命令与之交互。可以参考本章“Picking”一节，了解如何从屏幕上选择数据。（注意：开发人员也可以选择窗口系统接口，详细内容请参考第十八章。）

### vtkRenderWindowInteractor

与数据进行交互最简单的方法就是实例化一个vtkRenderWindowInterator对象。该类会响应一系列预先设置的事件和动作，而且也提供了重载缺省动作的方法。通过这个类可以控制渲染场景的相机和Actor，包括两种交互风格：位置敏感型(position sensitive)(即joystick模式)和动作敏感型(motion sensitive)(即trackball模式)。（后面的章节会详细介绍交互风格方面的内容。）

vtkRenderWindowInterator可以响应以下的渲染窗口事件。（一个渲染窗口(Rendering window)可以关联多个渲染器(Renderer)，其中渲染器绘制在渲染窗口的视口里，交互器支持一个渲染窗口拥有多个渲染器。）

* 按键j和按键t —— 在joystick（位置敏感型）和trackball（动作敏感型）模式之间的切换。joystick模式下，只要鼠标按下动作就持续发生；trackball模式下，鼠标按下且移动动作才产生。
* 按键c和按键a ——相机和演员（对象）模式之间的切换。相机模式下，鼠标事件对相机的位置和焦点起作用；对象模式下，鼠标事件作用于鼠标指针下的actor。
* 鼠标左键 ——相机模式下绕焦点旋转相机；对象模式下绕原点旋转actor。旋转方向是从渲染器视口的中心指向鼠标所在的位置。joystick模式下，旋转速度取决于鼠标的位置与视口中心的距离大小。
* 鼠标中键 —— 相机模式下滚动中键可以实现相机镜头的拉伸(Pan)；对象模式下中键可以平移actor。joystick模式下，镜头拉伸或平移的方由从视口中心指向鼠标所在的位置；trackball模式下，actor运动的方向与鼠标运动的方向一致。（注意：如果是两键鼠标，镜头的拉伸/对象的平移操作可以用按键<shift>+鼠标左键完成。）
* 鼠标右键 —— 相机模式下可以实现相机镜头的拉伸(Zoom)；对象模式下可以实现actor的缩放。按下鼠标右键且朝视口上半部分移动鼠标时，就是镜头的拉近/actor的放大；按下鼠标右键且朝视口的下半部分移动鼠标时，就是镜头的拉远/actor的缩小。joystick模式下，镜头拉伸的快慢或actor缩放速度取决于鼠标所在的位置与视口水平中心线的距离。
* 按键3 —— 激活或关闭渲染窗口的立体模式(stereo mode)。缺省情况下，会创建红-蓝立体对(red-blue stereo pairs)。对一些支持Crystal Eyes LCD立体眼镜的，必须调用方法SetStereoToCrystalEyes()实现该功能。
* 按键e/q —— 退出应用程序。
* 按键f —— 移动actor到光标的当前的位置。设置当前光标所在的位置为焦点，而且会以该点为旋转中心。
* 按键p —— 执行拾取（pick）操作。渲染窗口交互器内部有一个vtkPropPicker实例，可实现拾取功能。详细内容参考本章“拾取”一节。
* 按键r —— 沿当前视场的方向重置相机。移动相机和actor使所有的对象都可见。
* 按键s —— 改变所有actor的显示形式为表面模型。
* ·按键u —— 调用用户自定义方法。（在Tcl/Tk脚本程序下）通常会弹出交互窗口，可以输入命令进行交互。
* 按键w —— 改变所有actor的显示形式为线框模型。

默认的交互风格是位置敏感型(即joystick模式)，也就是说，只要鼠标按下就可以持续控制相机或actor以及渲染器。如果不喜欢这种缺省风格，你可以替换或者创建自己的交互风格。（参考第十八章了解有关编写交互风格的内容。）

vtkRenderWindowInteraction还有其他一些有用的特性。调用LightFollowCameraOn()方法(默认的行为)可以使相机位置和焦点与光源的位置和焦点同步（即创建“headlight”）。当然，也可以调用LightFollowCameraOff()关闭该功能。响应“u”键的回调函数可以通过AddObserver(UserEvent)方法进行设置。也可以设置一些与拾取有关的方法。AddObserver(StartPickEvent)定义了拾取之前调用的方法，而AddObserver(EndPickEvent)定义的是拾取完成后调用的方法。（详细内容可参考第三章“User Methods, Observers, and Commands”一节。）同样，你也可以实例化一个vtkAbstractPicker子类的对象，通过vtkRenderWindowInteractor里的方法SetPicker()设置拾取类型。（参考本章“拾取”一节。）

针对某一个Prop如果你想在交互速度与渲染质量方面取得平衡，可以通过交互器里的方法SetDesiredUpdateRate()设置期望的帧更新速率(Desired frame rate)。正常情况下，这些会自动进行处理的。（当鼠标处于活动状态时，期望更新速率(Desired update rate)会提高；当鼠标松开时，期望更新速率会降回原来的值。）参考本章的“Level-Of-Detail Actors”，“vtkLODProp3D”以及第七章“体绘制”方面的内容，了解更多的关于Prop以及与之相关联的mapper是如何调整渲染风格以达到期望的帧更新速率的。

前面已经介绍了vtkRenderWindowInteractor的使用方法，现摘要如下：

vtkRenderWindowInteractor iren

iren SetRenderWindow renWin

iren AddObserver UserEvent {wm deiconify .vtkInteract}

### 交互风格 Interactor Styles

VTK有两种不同的方法可以控制交互风格。第一种是使用vtkInteractorStyle的子类，可以是VTK自带的或者是你自行编写的类。第二种是添加Observer监听vtkRenderWindowInteractor里的事件，定义一系列回调函数（或命令）来实现交互。（注意：3D Widget是另外一种与渲染场景中的数据进行交互的更加复杂的方式，可以参考本章的“3D Widget”一节了解更多的信息。）

**vtkInteractorStyle.** 类vtkRenderWindowInteractor可以支持不同的交互风格。交互时键入“t”或者“j”可以在trackball和joystick两种交互模式之间切换（见前面的内容）。vtkRenderWindowInteractor可以将特定的窗口系统事件（如鼠标按下，鼠标移动，键盘事件等）转换成MouseMoveEvent, StartEvent等VTK事件。（参考第三章“User Methods, Observers, and Commands”一节。）不同的交互风格可以监听特定的事件，然后执行与之相关联的动作。通过vtkRenderWindowInteractor::SetInteractorStyle()方法可以设置不同的交互风格，比如：

vtkInteractorStyleFlight fightStyle

vtkRenderWindowInteractor iren

iren SetInteractorStyle flightStyle

注意：实例化vtkRenderWindowInteractor后，实际上就是实例化一个与特定的窗口系统相关的交互器实例，例如， Unix系统下，当建立一个vtkRenderWindowInteractor实例时，实际上建立的是vtkXRenderWindowInteractor实例，而Windows系统下就是建立一个vtkWin32RenderWindowInteractor实例。

**添加vtkRenderWindowInteractor Observers.** 虽然VTK里已经有大量的交互风格可供使用，但你也可根据应用程序的需要自行定制其他的交互风格。C++里做法一般是从vtkInteractorStyle派生出子类（参考第十八章“vtkRenderWindow Interaction Style”的内容），但对于解释性语言（如Tcl, Python或Java）来说，这样做就比较困难。对于这些解释性语言，最简单的方法就是使用Observer来绑定特定的事件进行交互（参考第三章“User Methods, Observers, and Commands”一节）。这种绑定适用于VTK支持的任何语言，包括C++, Tcl, Python和Java，VTK/Examples/GUI/Tcl/CustomInteraction.tcl这个Tcl例子看到这种用法。下面的代码就是从这个例子摘抄出来的，看看这种绑定是如何进行的。

vtkRenderWindowInteractor iren

iren SetInteractorStyle ""

iren SetRenderWindow renWin

# Add the observers to watch for particular events. These invoke

# Tcl procedures.

set Rotating 0

set Panning 0

set Zooming 0

iren AddObserver LeftButtonPressEvent {global Rotating; set Rotating 1}

iren AddObserver LeftButtonReleaseEvent {global Rotating; set Rotating 0}

iren AddObserver MiddleButtonPressEvent {global Panning; set Panning 1}

iren AddObserver MiddleButtonReleaseEvent {global Panning; set Panning 0}

iren AddObserver RightButtonPressEvent {global Zooming; set Zooming 1}

iren AddObserver RightButtonReleaseEvent {global Zooming; set Zooming 0}

iren AddObserver MouseMoveEvent MouseMove

iren AddObserver KeyPressEvent Keypress

proc MouseMove {} {

…

set xypos [iren GetEventPosition]

set x [lindex $xypos 0]

set y [lindex $xypos 1]

...

}

proc Keypress {} {

set key [iren GetKeySym]

if { $key == "e" } {

vtkCommand DeleteAllObjects

exit

}

…

}

这个例子关键的一步是调用SetInteractorStyle(“”)方法关闭缺省的交互风格，然后加入关联适当的Tcl过程的Observer来监听特定的事件。

这个例子是一个简单的介绍如何加入事件绑定的Tcl脚本程序，如果你想使用Tcl/Tk创建完整的包含用户图形界面的应用程序，可以使用类vtkTkRenderWidget，详细内容请参考第十八章。

## 4.3 Filtering Data

4.1节例子里的管线只由一个Source和Mapper对象组成，管线里没有Filter。这一部分我们会演示如何在管线里增加一个Filter。

Filter的连接可以通过方法SetInputConnection()和GetOutputPort()。例如，我们可以在4.1节例子的基础上，对组成模型的多边形数据(Polygon)做收缩(Shrink)处理，下面列出了部分代码，完整的代码可以在VTK/Examples/Rendering/Tcl/FilterCADPart.tcl里找到。

vtkSTLReader part

part SetFileName "$VTK\_DATA\_ROOT/Data/42400-IDGH.stl"

vtkShrinkPolyData shrink

shrink SetInputConnection [part GetOutputPort]

shrink SetShrinkFactor 0.85

vtkPolyDataMapper partMapper

partMapper SetInputConnection [shrink GetOutputPort]

vtkLODActor partActor

partActor SetMapper partMapper

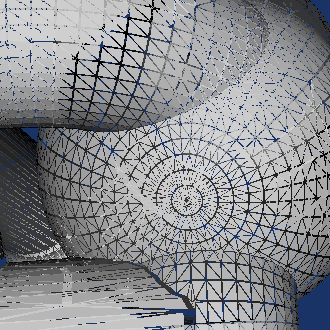


图4-3 Filtering Data, 这个例子我们使用了收缩Filter (Shrink Filter)，将多边形数据沿着模型的中心进行收缩。

正如你所见，创建可视化管线是比较简单的。只要选对正确的类，保证相互连接的Filter的输入和输出的类型匹配，以及设置必要的变量值即可。当管线里的Source或者Filter输出的数据类型与下一个Filter或Mapper的输入类型一致，就说输入和输出的类型是匹配的。输出的数据类型除了要与输入的匹配，还必须与输入的子类的类型匹配。可视化管线可以包含循环，但是一个Filter的输出不能直接地作为它本身的输入。

## 4.4 控制相机

你可能已经注意到，以上所列的例子里，并没有实例化相机或光源对象。如果你对3D图形学比较熟悉的话，就应该知道相机和光源对于要渲染的对象来说是必不可少的。VTK里，如果没有直接地创建相机和光源对象的话，渲染器会自动地创建默认的相机和光源实例。

### 实例化相机

下面的Tcl脚本演示了如何实例化一个相机对象以及如何把相机与渲染器关联起来。

vtkCamera caml

cam1 SetClippingRange 0.0475572 2.37786

cam1 SetFocalPoint 0.052665 -0.129454 -0.0573973

cam1 SetPosition 0.327637 -0.116299 -0.256418

cam1 ComputeViewPlaneNormal

cam1 SetViewUp -0.0225386 0.999137 0.034901

ren1 SetActiveCamera cam1

如果你想访问一个已经存在的相机（比如，渲染器自动创建的相机实例），用Tcl脚本可以写为：

set cam1 [ren1 GetActiveCamera]

$cam1 Zoom 1.4

一起来看一下上面列出的有关相机的方法。SetClippingRange()函数接收两个参数，分别表示沿着视平面法向量的方向，相机与近剪裁平面(Clipping Plane)和远剪裁平面的距离。渲染时，所有位于这两个剪裁平面之外的图形元将会被剪切掉，所以，必须保证所有的你想观察的对象位于这两个剪裁平面之间。FocalPoint和Position变量（在世界坐标系里定义）分别控制相机的方向和位置。ComputeViewPlaneNormal()方法会以当前设置的相机位置和焦点重置视平面的法向量。如果视平面的法向量与视平面不垂直的话，可以得到其他的视觉效果，如错切(Shearing)等。ViewUp变量控置相机“向上”的方向。最后，Zoom()方法是通过改变视角(View Angle)(即SetViewAngle()方法)使得物体放大显示。同样，你也可以使用Dolly()方法沿着视平面的法向方向移动相机，或者放大、收缩渲染场景中可见的actor。

### 简单操作方法

以上介绍的方法并不是常用的控制相机的简便方法。如果相机已经“看着”你想要的那个点，即已经设置了相机的焦点(Focal Point)，你可以使用Azimuth()和Elevation()方法绕着焦点旋转相机。

cam1 Azimuth 150

cam1 Elevation 60

Azimuth()方法会在球坐标下以焦点为中心沿着经度方向(Longitude Direction)旋转指定的角度；Elevation()方法则是沿着纬度方向(Latitude Direction)旋转。这两个方法都不会改变相机的View-up向量。但要注意一点，在球坐标的北极和南极时，View-up向量变成了与视平面向量平行。如果要避免这种情况，可以调用方法OrthogonalizeViewUp()强制View-up向量与视向量正交。However, this changes the camera coordinate system, so if you’re flying around an object with a horizon or view-up vector (such as terrain), camera manipulation is no longer natural with respect to the data.

### 控制视方向(Controlling The View Direction)

相机的一个常用的功能是在一个特定的方向生成一个视方向。可以调用SetFocalPoint()，SetPosition()和ComputeViewPlanNormal()方法，紧接着调用渲染器的ResetCamera()方法把该相机设置到渲染器中。

vtkCamera cam1

cam1 SetFocalPoint 0 0 0

cam1 SetPosition 1 1 1

cam1 ComputeViewPlaneNormal

cam1 OrthogonalizeViewUp

ren1 SetActiveCamera cam1

ren1 ResetCamera

相机的初始方向(视向量或视平面向量)是通过焦点、相机位置及方法ComputeViewPlanNormal()来计算的。你也可以指定一个初始的View-up向量，然后将它与视向量正交而得到初始的方向。ResetCamera()方法可以沿着视向量移动相机，这样，渲染器里所有的Actor都会可见。

### 透视和正交投影(Perspective Versus Orthogonal Views)

前面的例子，我们都假定相机是透视投影的，即通过相机的视角(View Angle)来控制渲染时Actor在视平面上的投影。透视投影产生更加自然的影像的同时，会引入形变的效果，这在某些应用程序中是不希望的。正交(或平行)投影是另外一种投影模式。正交投影模式下，视线都是平行的，物体的渲染就没有距离差别的效果。

相机的正交投影模式可以使用方法vtkCamera::ParallelProjectionOn()设置。正交(平行)投影模式下，相机的视角(View Angle)就不再控制对象的缩放了，可以用方法SetParallelScale()来代替控制对象的放大与缩小。

### 保存/恢复相机状态

另外一个应用程序普遍需要的功能是保存和恢复相机状态(也就是重置视图)。要保存相机状态，最起码需要保存剪裁范围、相机焦点、位置和View-up向量，计算视平面向量。然后可以用这些保存的信息实例化一个相机对象，恢复相机的状态，并把相机设置到适当的渲染器中(即SetActiveCamera())。

某些情况下，可能还需要保存其他的信息。比如，如果设置了相机的视角(View Angle) (或Parallel Scale)，你就需要保存这些信息。又或者你把相机用于立体视图，就需要设置及保存EyeAngle和Stereo标志。

## 4.5 控制光源

相对于相机来说，光源的控制要简单一些。使用较多的方法主要有SetPosition()，SetFocalPoint()以及SetColor()。Position和FocalPoint分别控制光源的位置和方向。光源的Color是用RGB值来表示。光源可以通过方法SwitchOn()和SwitchOff()打开和关闭，光源的强度则是用方法SetIntensity()来设置。

缺省的vtkLight实例是方向光源(Direction Light)，也就是说光源的位置和焦点所确定的向量与光源的光线是平行的，光源的位置位于无限远处。这意味着如果光源的焦点和位置做平移变换的话，照射在物体上的光照不会发生变化。

光源与渲染器的关联如下面代码所示。

vtkLight light

light SetColor 1 0 0

light SetFocalPoint [cam1 GetFocalPoint]

light SetPosition [cam1 GetPosition]

ren1 AddLight light

以上代码创建的是一个红色的前灯光源(Headlight)，即光源的位置和焦点与相机的一致。这也是一个非常有用的特技，主要应用于与渲染器交互时，当相机发生移动的时候可以重置光源的位置。

### 位置光源 (Positional Lights)

用PositionnalOn()方法可以创建位置光源(即聚光灯)，这个方法通常与SetConeAngle()访求协同使用，用来控制聚光光照的照射范围。聚光锥角(Cone Angle)等于180度时意味着聚光光源不起作用。

## 4.6 控制3D Props

VTK中的渲染窗口渲染的对象通常称之为“Prop”(“Prop”这个词来源于戏剧，指的是出现在舞台上的东西)。VTK里有几种不同类型的Prop，包括vtkProp3D和vtkActor。其中vtkProp3D是一个抽象父类，表示的是三维场景中的对象；vtkActor是vtkProp3D的子类，用类似多边形(Polygon)和线(Lines)等基本数据来定义它的几何。

### 指定vtkProp3D的位置

我们已经知道了如何绕着一个对象来移动相机；反过来，也可以保持相机不动，而对Prop进行变换。下面的方法可以用于定义一个vtkProp3D(及其子类)对象的位置。

* SetPosition(x, y, z) —— 指定vtkProp3D对象在世界坐标系中的位置。
* AddPosition(deltaX, deltaY, deltaZ) —— 用指定的X、Y、Z三个方向的增量来平移Prop。
* RotateX(theta), RotateY(theta), RotateZ(theta) —— 分别用指定的角度绕X、Y、Z轴旋转Prop。
* SetOrientation(x, y, z) —— 通过先绕Z轴，然后绕X轴，最后绕Y轴旋转，从而来确定Prop的方向。
* AddOrientation(a1, a2, a3) —— 在当前Prop方向增加a1, a2, a3增量。
* RotateWXYZ(theta, x, y, z) —— 绕x, y, z指定的向量旋转theta角度。
* SetScale(sx, sy, sz) —— 分别沿X、Y、Z三个方向缩放sx, sy, sz比例。
* SetOrigin(x, y, z) —— 指定Prop的原点，Prop的原点指的是Prop绕转或缩放时的基准点。

以上这些方法联合使用，可以产生复杂的变换矩阵。最重要的一点是，以上方法使用时要注意它们的调用顺序，不同的顺序对Actor的位置有不同的影响。VTK里是用以下的顺序来应用这些变换的：

1. 移动Prop到原点。

2. 缩放。

3. 绕Y轴旋转。

4. 绕X轴旋转。

5. 绕Z轴旋转。

6. 从原点中移动回原来的位置。

7. 平移。

第1步和第6步平移的大小分别是Origin的负值和正值。单纯的平移是由vtkProp3D的Position值来确定的。这些变换中最容易混淆的旋转操作。例如，将一个Prop先绕X旋转，再绕Y轴旋转，它的效果与先绕Y轴旋转，再绕X轴旋转的效果是完全不一样的(图4-4)。要了解更多关于Actor变换内容可以参考《Visualization Toolkit》一书。



图4-4 不同旋转顺序的效果。左边是先绕X轴旋转，再绕Y轴旋转的效果；右边是先绕Y轴旋转，再绕X轴旋转的效果

接下来，我们会介绍各种类型的vtkProp3D，其中VTK里最常用的类是vtkActor。在“控制vtkActor2D”一节里会介绍一下2D Prop (也就是vtkActor2D)，这个类主要应用于标注(Annotation)及其他的二维操作。

### Actors

Actor是最常的vtkProp3D类型，像其他的vtkProp3D子类一样，vtkActor提供了一组渲染属性，如表面属性(如环境、散射和镜面颜色)，显示形式(Representation)(如表面模型或线框模型)，纹理映射以及几何定义(mapper)。

**定义几何**。前面的例子我们已经知道，一个Actor的几何是通过SetMapper()方法指定的：

vtkPolyDataMapper mapper

mapper SetInputConnection [aFilter GetOutputPort]

vtkActor anActor

anActor SetMapper mapper

在这个例子里，mapper的类型是vtkPolyDataMapper，也就是用类似点、线、多边形(Polygons)和三角形带(Triangle Strips)等几何图元进行渲染的。Mapper会结束可视化管线，在可视化子系统和图形子系统之间起到桥梁的作用。

**Actor的属性**。Actor里有一个类型为vtkProperty的实例，主要是用来控制Actor的显示属性。常用的属性是Actor的颜色，我们会在后面的内容详细描述。其他的重要属性有Representation(点、线框模型或表面模型)、着色方法(Shading Method)(平面着色或Gouraud着色)、Actor的不透明度(相对于透明度)以及环境光、散射光和镜面光颜色等相关参数。下面的脚本程序演示了如何设置这些变量。

vtkActor anActor

anActor SetMapper mapper

[anActor GetProperty] SetOpacity 0.25

[anActor GetProperty] SetAmbient 0.25

[anActor GetProperty] SetDiffuse 0.6

[anActor GetProperty] SetSpecular 1.0

[anActor GetProperty] SetSpecularPower 10.0

注意，我们通过方法GetProperty()间接引用Actor的属性。或者，我们也可以先实例化一个vtkProperty对象，然后把它设置到Actor中。

vtkProperty prop

prop SetOpacity 0.25

prop SetAmbient 0.5

prop SetDiffuse 0.6

prop SetSpecular 1.0

prop SetSpecularPower 10.0

vtkActor anActor

anActor SetMapper mapper

anActor SetProperty prop

后一种方法的好处是，我们可以把多个Actor设置成同一种属性。

**Actor的颜色**。颜色可能是Actor里最重要的属性了。设置Actor的颜色最简单的方法莫过于调用SetColor()方法，该方法用RGB值来设置一个Actor的红、绿、蓝分量的颜色，每个分量的取值范围从0到1。

[anActor GetProperty] SetColor 0.1 0.2 0.4

或者，我们也可以通过设置环境光颜色、散射光颜色和镜面光颜色来控制Actor的颜色。

vtkActor anActor

anActor SetMapper mapper

[anActor GetProperty] SetAmbientColor .1 .1 .1

[anActor GetProperty] SetDiffuseColor .1 .2 .4

[anActor GetProperty] SetSpecularColor 1 1 1

以上代码把环境光颜色设置成深灰色，散射光颜色设置成蓝色的阴影，镜面光颜色设置成白色。(提示：SetColor()方法就是用指定的RGB值来设置环境光颜色、散射光颜色和镜面光颜色。)

**重要**：Actor的属性关于颜色的设置只有当Actor的Mapper没有标量数据(Scalar Data)时才有效。缺省情况下，Mapper输入的标量数据会对Actor进行着色，而Actor的颜色设置会被忽略。如果要忽略这些标量数据，可以使用方法ScalarVisibilityOff()，如下面的Tcl脚本所示：

vtkPolyDataMapper planeMapper

planeMapper SetInputConnection [CompPlane GetOutputPort]

planeMapper ScalarVisibilityOff

vtkActor planeActor

planeActor SetMapper planeMapper

[planeActor GetProperty] SetRepresentationToWireframe

[planeActor GetProperty] SetColor 0 0 0

**Actor的透明度**。很多情况下，调整Actor的透明度(或者不透明度)是很有用的。比如，如果你想显示一个病人图像的内部器官，而器官的外面包含着皮肤，这时你可以调整皮肤的透明度，使得内部器官可见。可以使用vtkProperty::SetOpacity()方法，如下：

vtkActor popActor

popActor SetMapper popMapper

[popActor GetProperty] SetOpacity 0.3

[popActor GetProperty] SetColor .9 .9 .9

要注意透明度的实现是使用渲染库里的α-Blending处理技术。这种处理技术要求以正确的顺序来渲染多边形(Polygons)。实际上，这是很难做到的，特别是当你有多个透明的Actor需要渲染时。要对多边形排序，应该把透明的Actor加到待渲染的Actor列表的最后。你也可以用vtkDepthSortPolyData这个Filter沿着视向量方向对多边形排序。关于这个Filter用法，可以参考VTK/Examples/VisualizationAlgorithm/Tcl/DepthSort.tcl这个例子。更多关于这方面的内容可以参考本章的“半透明多边形几何”(Translucent polygonal geometry)一节。

**其他的属性**。Actor还有其他一些重要的属性。你可以用方法VisibilityOn()/VisibilityOff()来控制Actor的可见与不可见。如果有拾取过程中，你不想某个Actor被拾取，可以使用方法PickableOff()关闭拾取属性（关于拾取方面的内容，可以参考本章的“拾取”一节）。Actor有一个拾取事件(Pick Event)，当它们被拾取时就会调用这个事件。另外，方法GetBounds()可以获取与坐标轴对齐的Actor的包围盒(Bounding Box)。

### Level-Of-Detail Actors

图形系统一个主要的问题是，交互时有时会变得很慢。为了解决这个问题，VTK使用Level-of-detail技术，在与数据交互时，以低分辨率的Representation来表示Actor以求达到更快的渲染速度。

在本章的“读取源对象”(Reader Source Object)一节，我们已经使用了vtkLODActor类。基本上，最简单的方法就是用vtkLODActor实例来替代vtkActor实例。另外，你也可以控制Level-of-detail的Representation。vtkLODActor缺省的做法是利用原始的Mapper创建另外两个低分辨率的模型。第一个是从定义Mapper的输入的点采样得到的点云。你可以控制点云里点的个数(缺省是150个点)，如下所示。

vtkLODActor dotActor

dotActor SetMapper dotMapper

dotActor SetNumberOfCloudPoints 1000

Actor最低分辨率的模型是就是一个包围盒。其他的level-of-detail也可以通过方法AddLODMapper()加入，由于复杂性问题，一般都没有必要加入。

为了控制渲染时Actor所选的level-of-detail，你可以设置渲染窗口的期望渲染帧率：

vtkRenderWindow renWin

renWin SetDesiredUpdateRate 5.0

这样，就会以每秒5帧的速率进行渲染。vtkLODActor会自动地选择合适的Level-of-detail来达到请求的渲染速率。（注意：类似vtkRenderWindowInteractor等交互器，会自动地控制期望更新速度(Desired Update Rate)，一般的做法是，当鼠标松开时，会把帧率设置得很低，而鼠标按下时，则会提高相应的帧率。这样就能保证相机的运动时产生低分辨率/高帧率，而相机停止时产生高分辨率/低帧率的理想效果。如果你想了解更多关于Level-of-detail的控制方面的内容，可以参考本章的“vtkLODProp3D”一节，通过这个类，可以指定不同的Level。）

### Assemblies

Actors有时也会组合在一起形成层次结构，当其中的某个Actor运动时，会影响到其他Actor的位置。例如，一个机械手臂可能由上臂、前臂、手腕和末端等部分通过关节连接起来。当上臂绕着肩关节旋转时，我们希望的是其他部分也会跟着运动。这种行为的实现就要用到Assembly，vtkAssembly是vtkActor的子类。下面的程序演示了如何使用vtkAssembly(摘自VTK/Examples/Rendering/Tcl/assembly.tcl)。

vtkSphereSource sphere

vtkPolyDataMapper sphereMapper

sphereMapper SetInputConnection [sphere GetOutputPort]

vtkActor sphereActor

sphereActor SetMapper sphereMapper

sphereActor SetOrigin 2 1 3

sphereActor RotateY 6

sphereActor SetPosition 2.25 0 0

[sphereActor GetProperty] SetColor 1 0 1

vtkCubeSource cube

vtkPolyDataMapper cubeMapper

cubeMapper SetInputConnection [cube GetOutputPort]

vtkActor cubeActor

cubeActor SetMapper cubeMapper

cubeActor SetPosition 0.0 .25 0

[cubeActor GetProperty] SetColor 0 0 1

vtkConeSource cone

vtkPolyDataMapper coneMapper

coneMapper SetInputConnection [cone GetOutputPort]

vtkActor coneActor

coneActor SetMapper coneMapper

coneActor SetPosition 0 0 .25

[coneActor GetProperty] SetColor 0 1 0

# top part of the assembly

vtkCylinderSource cylinder;

vtkPolyDataMapper cylinderMapper

cylinderMapper SetInputConnection [cylinder GetOutputPort]

cylinderMapper SetResolveCoincidentTopologyToPolygonOffset

vtkActor cylinderActor

cylinderActor SetMapper cylinderMapper

[cylinderActor GetProperty] SetColor 1 0 0

# Create the assembly and add the 4 parts to it. Also set the origin, position

# and orientation in space.

vtkAssembly assembly

assembly AddPart cylinderActor

assembly AddPart sphereActor

assembly AddPart cubeActor

assembly AddPart coneActor

assembly SetOrigin 5 10 15

assembly AddPosition 5 0 0

assembly RotateX 15

# Create the Renderer, RenderWindow, and RenderWindowInteractor

#

vtkRenderer ren1

vtkRenderWindow renWin

renWin AddRenderer ren1

vtkRenderWindowInteractor iren

iren SetRenderWindow renWin

# Add the actors to the renderer, set the background and size

#

ren1 AddActor assembly

ren1 AddActor coneActor

注意是如何使用vtkAssembly里的方法AddPart()来建立层次结构的。只要不是自我嵌套，Assembly可以组合成任意层次深度的结构。vtkAssembly是vtkProp3D的子类，但没有与之相关联的Property以及Mapper。因此，vtkAssembly层次结构里的结点必须要包含关于材料的属性(如颜色等)及其他相关的几何信息。一个Actor可以用于多个Assembly(注意以上例子中的coneActor是如何作为一个单独的Actor以及作为Assembly里的一个节点的)。通过渲染器里的方法AddActor()只要加入最顶层的Assembly，而低层次的Assembly不用加入，因为它们会递归加入到渲染器中。

如果一个Actor加入到多个Assembly时(就像上面的例子)，你可以会想到如何去区分不同的Actor。（关于这一点，在类似“拾取”操作时是非常重要的，因为你必须要区分一下到底哪个vtkProp对象是处于选中的姿态。）我们会在后面的“拾取”一节，详细讨论这个问题，以及介绍类vtkAssemblyPath的用法。

### Volumes

类vtkVolume主要用于体绘制，这个类与vtkActor非常类似。vtkVolume从vtkProp3D派生了对Volume进行定位和定向的方法。vtkVolume内部也有一个与它本身相关联的Property对象，即vtkVolumeProperty。请参考第七章“体绘制”了解更多的关于vtkVolume的应用以及体绘制方面的内容。

### vtkLODProp3D

类vtkLODProp3D与vtkLODActor(参考本章“Level-Of-Detail Actors”一节)类似，也是使用不同的Representation来表示它本身，以求达到更佳的渲染速率。与vtkLODActor不同的是，vtkLODProp3D只支持体绘制和面绘制。也就是说，你只能够在体绘制应用程序中使用vtkLODProp3D类来获取理想的交互帧率。以下的例子演示了这个类的使用。

vtkLODProp3D lod

set level1 [lod AddLOD volumeMapper volumeProperty2 0.0]

set level2 [lod AddLOD volumeMapper volumeProperty 0.0]

set level3 [lod AddLOD probeMapper\_hres probeProperty 0.0]

set level4 [lod AddLOD probeMapper\_lres probeProperty 0.0]

set level5 [lod AddLOD outlineMapper outlineProperty 0.0]

基本上，根据不同的渲染复杂度，你会创建不同的Mapper，并把它设置到vtkLODProp3D里。AddLOD()方法可以接收Volume或几何类型的Mapper作为参数，可选的参数包括纹理映射、属性对象等。（根据你提供的信息不同，该方法会有不同的函数签名(Signatures)。）AddLOD()方法的最后一个参数是渲染的估计时间。一般情况下都设置为0，即没有针对渲染的初始估计值。该方法返回一个整型的ID值，利用这个值就可以访问对应的LOD（可以用来选择某一Level或者删除某一Level）。

## 4.7 使用纹理

纹理映射是生成逼真的可视化效果的强大的图形工具。二维纹理映射的基本思想是在渲染过程中，图像可以“贴”到渲染对象的表面上，因此可以创建出细节更加丰富的渲染效果。纹理映射时需要提供三类信息：待贴纹理图的面、纹理映射(在VTK里，其实就是vtkImageData类型的数据，即2D图像)以及纹理坐标(控制纹理图在面上的位置)。(Texture mapping requires three pieces of information: a surface to apply the texture to; a texture map, which in VTK is a vtkImageData dataset (i.e., a 2D image); and texture coordinates, which control the positioning of the texture on the surface.)

下面的例子演示了如何使用纹理映射(完整的程序代码见VTK/Examples/Rendering/Tcl/TPlane.tcl)。要注意纹理映射(类vtkTexture)是与Actor相联的，而纹理坐标则是由平面来定义(纹理坐标是由类vtkPlaneSource创建的)。

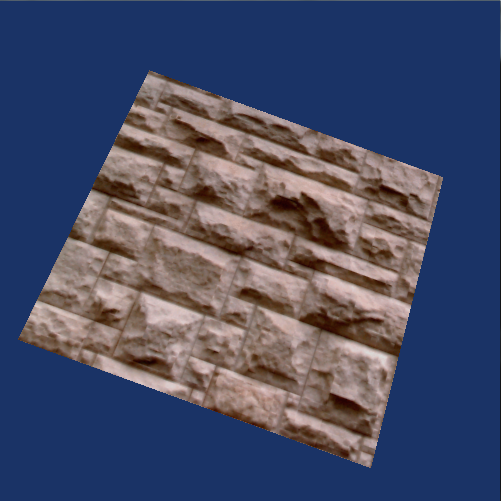


图4-5 平面纹理映射

# Load in the texture map.

vtkBMPReader bmpReader

bmpReader SetFileName "$VTK\_DATA\_ROOT/Data/masonry.bmp"

vtkTexture atext

atext SetInputConnection [bmpReader GetOutputPort]

atext InterpolateOn

# Create a plane source and actor.

vtkPlaneSource plane

vtkPolyDataMapper planeMapper

planeMapper SetInputConnection [plane GetOutputPort]

vtkActor planeActor

planeActor SetMapper planeMapper

planeActor SetTexture atext

很多时候，纹理坐标是获取不到的，因为这些纹理坐标不能在管线中生成。如果你需要生成纹理坐标，可以参考第五章“生成纹理坐标”一节。尽管一些老的图形卡在纹理贴图时会有所限制（比如要求所贴的纹理图必须是二维的，而且每维的大小必须小于1024），但VTK支持任意尺寸的纹理图。程序运行时，VTK会检索图形系统，确定这些图形系统的性能，然后会自动地对你所设置的纹理图做采样，以求达到特定的图形卡的要求。

## 4.8 拾取Picking

拾取操作是可视化应用程序中常见的一种功能。拾取主要是用于选择数据和Actor或者获取底层的数据值。在显示位置上(以像素为坐标值)中拾取时，就会调用vtkAbstractPicker的Pick()方法。依赖于所用的拾取类不同，拾取时返回的信息也不同，最简单的是返回一个x-y-z的全局坐标值，或者是cell的ID值，点的ID值，cell参数坐标(Cell Parametric Coordinates)，所拾取的vtkProp实例，以及Assembly path。拾取方法的原型是：

Pick(selectionX, selectionY, selectionZ, Renderer)

注意Pick()方法需要一个渲染器作为参数。与渲染器相关联的Actor都是拾取的候选对象。另外，selectionZ通常都设置为0.0，它是与Z-buffer相关的值。（一般，Pick()这个方法都不会直接去调用它，用户使用vtkRenderWindowInteractor进行交互时，由这个类来管理拾取操作。这种情况下，用户只要选择一个拾取实例，让这个实例来控制拾取过程即可，后面的例子会演示如何使用。）

VTK支持多种不同功能的拾取类型（请参考图19-16，列出了与拾取相关的类的继承图）。类vtkAbstractPicker是所有拾取类的基类，它定义了一些公用的API，允许用户通过方法GetPickPosition()来获取拾取位置(全局坐标下)。

vtkAbstractPicker有两个直接子类。第一个是vtkWorldPointPicker，这是一种使用Z-buffer快速返回所拾取的位置的x-y-z全局坐标的类(基于硬件的)，但不会返回其他的信息(比如到底拾取了哪一个vtkProp实例等)。类vtkAbstractPropPicker是从vtkAbstractPicker中直接派生的另外一个子类。它定义了可以用于拾取某个vtkProp实例的API。下面列出一些比较方便的用于获取vtkProp实例的方法。

GetProp() —— 返回拾取的vtkProp实例指针。如果拾取了某个对象，就返回指向该vtkProp对象的指针，否则返回NULL。

GetProp3D() —— 如果拾取的是vtkProp3D对象，则返回该vtkProp3D对象的指针。

GetActor2D() —— 如果拾取的是vtkActor2D对象，则返回该vtkActor2D对象的指针。

GetActor() —— 如果拾取的是vtkActor对象，则返回该vtkActor对象的指针。

GetVolume() —— 如果拾取的是vtkVolume对象，则返回该vtkVolume对象的指针。

GetAssembly() —— 如果拾取的是vtkAssembly对象，则返回该vtkAssembly对象的指针。

GetPropAssembly() —— 如果拾取的是vtkPropAssembly对象，则返回该vtkPropAssembly对象的指针。

### vtkAssemblyPath

### 例子

通常，拾取是由vtkRenderWindowInteractor自动管理的(见“使用VTK交互器”一节了解更多关于交互器的内容)。比如，当按下P键时，vtkRenderWindowInteractor会调用内部的vtkPropPicker实例执行拾取操作。接着，可以通过vtkRenderWindowInteractor访问拾取器(Picker)，或者其他信息。也可以给vtkRenderWindowInteractor指定一个从vtkAbstractPicker派生的拾取器。图4-6显示了对数据集拾取的结果，程序代码摘自VTK/Examples/Annotation/Tcl/annotationPick.tcl。



图4-6 带标注信息的拾取操作

vtkCellPicker picker

picker AddObserver EndPickEvent annotatePick

# Create a text mapper and actor to display the results of picking.

vtkTextMapper textMapper

set tprop [textMapper GetTextProperty]

$tprop SetFontFamilyToArial

$tprop SetFontSize 10

$tprop BoldOn

$tprop ShadowOn

$tprop SetColor 1 0 0

vtkActor2D textActor

textActor VisibilityOff

textActor SetMapper textMapper

# Create the Renderer, RenderWindow, and RenderWindowInteractor

#

vtkRenderer ren1

vtkRenderWindow renWin

renWin AddRenderer ren1

vtkRenderWindowInteractor iren

iren SetRenderWindow renWin

iren SetPicker picker

# Create a Tcl procedure to create the text for the text mapper used to

# display the results of picking.

proc annotatePick {} {

if { [picker GetCellId] < 0 } {

textActor VisibilityOff

} else {

set selPt [picker GetSelectionPoint]

set x [lindex $selPt 0]

set y [lindex $selPt 1]

set pickPos [picker GetPickPosition]

set xp [lindex $pickPos 0]

set yp [lindex $pickPos 1]

set zp [lindex $pickPos 2]

textMapper SetInput "($xp, $yp, $zp)"

textActor SetPosition $x $y

textActor VisibilityOn

}

renWin Render

}

# Pick the cell at this location.

picker Pick 85 126 0 ren1

这个例子使用vtkTextMapper在屏幕上绘制拾取点的世界坐标值。(参考“文本标注”一节了解更多信息)。注意，在这个例子中，我们注册了EndPickEvent事件，拾取操作完成后，就会调用annotatePick()过程。

## 4.9 vtkCoordinate和坐标系统

VTK支持多种不同类型的坐标系统，类vtkCoordinate管理这些坐标系统之间的变换。支持的坐标系统有：

* DISPLAY ——X-Y坐标值定义在渲染窗口(Rendering Window)中，以像素为单位(注意vtkRenderWindow是vtkWindow的子类)。原点在窗口的左下角(这一点对于下面的二维坐标系统都是如此)。
* NORMALIZED DISPLAY —— 窗口的X、Y坐标取值归一化，即(0, 1)。
* VIEWPORT —— X-Y坐标值定义在视口(Viewport)或者渲染器(Renderer，vtkRenderer是vtkViewport的子类)里。
* NORMALIZED VIEWPORT —— 视口里的X-Y坐标取值归一化，即(0, 1)。
* VIEW —— X-Y-Z坐标值(取值范围为(-1, 1))定义在相机坐标系统，Z表示深度。
* WORLD —— X-Y-Z为全局坐标值。
* USERDEFINED —— X-Y-Z定义在用户自定义的空间里。用户必须为自定义的坐标系统提供空间变换方法。请参考类vtkCoordinate了解更多信息。

类vtkCoordinate可以用于坐标系统之间的变换，也可以用于连接各个坐标系统以形成“相对”或者“偏移”等坐标值。参考下部分内容了解vtkCoordinate的用法。

## 4.10 控制vtkActor2D

vtkActor2D与vtkActor很多功能都类似，除了vtkActor2D是在覆盖平面上绘制的以及没有与之相关联的4×4的变换矩阵。与vtkActor类似，vtkActor2D涉及到一个mapper(即vtkMapper2D)和属性对象(即vtkProperty2D)。使用vtkActor2D时，比较困难的部分是如何定位它的对象。定位vtkActor2D对象时会用到类vtkCoordinate(请参考上一部分的“vtkCoordinate和坐标系统”相关内容)。下面的例子演示了如何使用vtkCoordinate对象。

vtkActor2D bannerActor

bannerActor SetMapper banner

[bannerActor GetProperty] SetColor 0 1 0

[bannerActor GetPositionCoordinate] SetCoordinateSystemToNormalizedDisplay

[bannerActor GetPositionCoordinate] SetValue 0.5 0.5

在这个例子中，访问了坐标对象以及定义了它的坐标系统，然后设置了该坐标系统下的合适的坐标值。这个例子中，使用了归一化显示坐标系统(Normalized Display Coordinate System)，因此坐标范围定义为0到1，坐标值(0.5, 0.5)就设置vtkActor2D对象在渲染窗口的中间位置。vtkActor2D也提供了一个方便的接口，SetDisplayPosition()，可以设置坐标系统为DISPLAY，并且利用传入的参数(以像素为单位)设置vtkActor2D对象在渲染窗口中的位置。下面一节的内容将会演示如何使用这个方法。

## 4.11 文本标注(Text Annotation)

VTK提供了两种方法用于标注图像。第一种是可以在三维图形窗口的顶层上绘制的文本(和图形)，通常涉及的是在覆盖平面上绘制。第二种是创建三维的多边形数据的文本，可以像其他三维图形对象一样进行变换及显示。这两种类型的标注分别称为二维和三维标注，从图4-7中可以看出它们的区别。

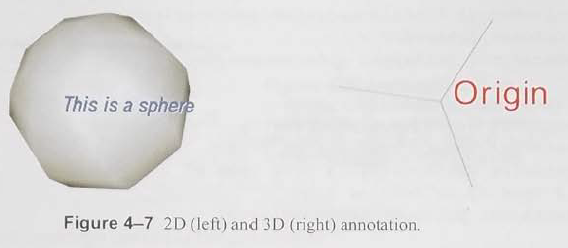


图4-7 二维(左)和三维(右)标注

### 二维文本标注(2D Text Annotation)

使用二维文本标注时，需要用到二维的Actor(vtkActor2D或它的子类，如vtkScaledTextActor)和Mapper(vtkMapper2D或其子类vtkTextMapper)。二维的Actor和Mapper跟三维的类似，不同的是前者是在图形或图像的顶层覆盖平面(overlay plane)上进行渲染的。下面的例子摘自VTK/Examples/Annotation/Tcl/TestText.tcl，结果如图4-7所示。

vtkSphereSource sphere

vtkPolyDataMapper sphereMapper

sphereMapper SetInputConnection [sphere GetOutputPort]

sphereMapper GlobalImmediateModeRenderingOn

vtkLODActor sphereActor

sphereActor SetMapper sphereMapper

# Create a scaled text actor.

# Set the text, font, justification, and properties (bold, italics, etc.).

vtkTextActor textActor

textActor SetTextScaleModeToProp

textActor SetDisplayPosition 90 50

textActor SetInput "This is a sphere"

# Set coordinates to match the old vtkScaledTextActor default value

[textActor GetPosition2Coordinate] SetCoordinateSystemToNormalizedViewport

[textActor GetPosition2Coordinate] SetValue 0.6 0.1

set tprop [textActor GetTextProperty]

$tprop SetFontSize 18

$tprop SetFontFamilyToArial

$tprop SetJustificationToCentered

$tprop BoldOn

$tprop ItalicOn

$tprop ShadowOn

$tprop SetColor 0 0 1

# Create the Renderer, RenderWindow, RenderWindowInteractor

#

vtkRenderer ren1

vtkRenderWindow renWin

renWin AddRenderer ren1

vtkRenderWindowInteractor iren

iren SetRenderWindow renWin

# Add the actors to the renderer; set the background and size; zoom in;

# and render.

#

ren1 AddViewProp textActor

ren1 AddViewProp sphereActor

vtkTextProperty实例可以设置字体(Arial, Courier或者Times等)，文本颜色，粗体、斜体开关以及字体的阴影效果等。字体的阴影效果可以使标注的文本在复杂背景下可读性更强。标注文本的位置和颜色则通过关联的vtkActor2D控制的。在这个例子中，文本的位置是用显示坐标或者像素坐标进行设置的。

vtkTextProperty也支持对齐(垂直和水平)及多行文本。方法SetJustificationToLeft()，SetJustificationToCentered()和SetJustificationToRight()可以控制水平对齐方向。SetVerticalJustificationToBottom()，SetVerticalJustificationToCentered()和SetVerticalJustificationToTop()等方法可以控制垂直对齐方向，缺省的对齐方向是左下角对齐。文本中嵌入“\n”字符可以支持多行文本。图4-8是对齐和多行文本的效果，该例子摘自VTK/Examples/Annotation/Tcl/multiLineText.tcl。以下是例子的主要代码：

vtkTextMapper textMapperL

textMapperL SetInput "This is\nmulti-line\ntext output\n(left-top)"

set tprop [textMapperL GetTextProperty]

$tprop ShallowCopy multiLineTextProp

$tprop SetJustificationToLeft

$tprop SetVerticalJustificationToTop

$tprop SetColor 1 0 0

vtkActor2D textActorL

textActorL SetMapper textMapperL

[textActorL GetPositionCoordinate] SetCoordinateSystemToNormalizedDisplay

[textActorL GetPositionCoordinate] SetValue 0.05 0.5

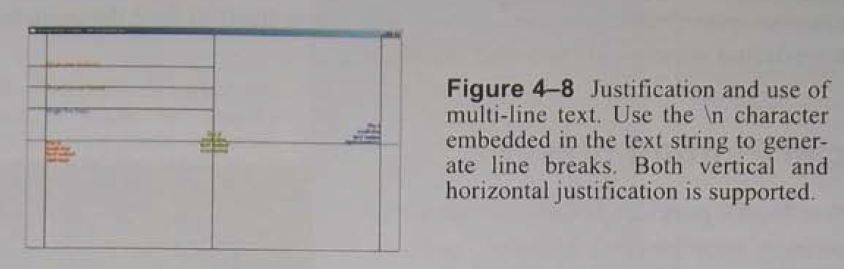


图4-8 文本对齐和多行文本。使用文本里嵌入\n字符可以生成多行文本，支持水平和垂直对齐

注意，以上代码使用vtkCoordinate对象(调用方法GetPositionCoordinate()获得)控制Actor在Normalized Display坐标系统下位置。参考“vtkCoordinate和坐标系统”一节，了解更多关于放置文本标注的信息。

### 三维文本标注与vtkFollower

三维文本标注的实现是使用vtkVectorText创建文本字符串的多边形表达形式(Polygonal Representation)，然后把它放置在渲染场景中。放置三维文本标注有用的类是vtkFollower。该类是vtkActor的子类，vtkFollower的对象总是面向渲染器的活动相机，因此可以保证这种文本标注都是可读、可见的。以下的代码演示了如何使用这个类，代码摘自VTK/Examples/Annotation/Tcl/textOrigin.tcl，运行结果如图4-7所示。这个例子创建了一个坐标轴和一个vtkVectorText实例，结合vtkFollower对象标注该坐标轴的原点。

vtkAxes axes

axes SetOrigin 0 0 0

vtkPolyDataMapper axesMapper

axesMapper SetInputConnection [axes GetOutputPort]

vtkActor axesActor

axesActor SetMapper axesMapper

# Create the 3D text and the associated mapper and follower (a type of

# actor). Position the text so it is displayed over the origin of the axes.

vtkVectorText atext

atext SetText "Origin"

vtkPolyDataMapper textMapper

textMapper SetInputConnection [atext GetOutputPort]

vtkFollower textActor

textActor SetMapper textMapper

textActor SetScale 0.2 0.2 0.2

textActor AddPosition 0 -0.1 0

…etc…after rendering…

textActor SetCamera [ren1 GetActiveCamera]

当相机绕着坐标轴旋转时，vtkFollower对象会调整自己的方向，使其朝向相机。你可以试着在渲染窗口里，用鼠标移动相机，观察这种变化。

## 4.12 专用的绘图类(Special Plotting Classes)

### Scalar Bar

### X-Y Plots