The Basics

The purpose of this chapter is to introduce you to some of VTK's capabilities by way of a selected set of examples. Our focus will be on commonly used methods and objects, and combinations of objects. We will also introduce important concepts and useful applications. By no means are all of VTK's features covered; this chapter is meant to give you a broad overview of what's possible. You'll want to refer to man pages or class `.h` files to learn about other options each class might have.

Most of the examples included here are implemented in the Tcl programming language. They could just as easily be implemented in C++, Java, and Python—the conversion process between the languages is straightforward (see "Conversion Between Languages" on page 38). C++ does offer some advantages, mainly access and manipulation of data structures and pointers, and some examples reflect this by being implemented in the C++ language.

Each example presented here includes sample code and often a supplemental image. We indicate the name of the source code file (when one exists in the accompanying distribution CD), so you will not have to enter it manually. We recommend that you run and understand the example and then experiment with object methods and parameters. You may also wish to try suggested alternative methods and/or classes. Often, the *Visualization Toolkit* offers several approaches to achieve similar results. Note also that the scripts are often modified from what's found in the source code distribution. This is done to simplify concepts or remove extraneous code.

Learning an object-oriented system like VTK first requires understanding the programming abstraction, and then becoming familiar with the library of objects and their methods. We recommend that you review "System Architecture" on page 19 for information about the programming abstraction. The examples in this chapter will then provide you with a good overview of the many VTK objects.

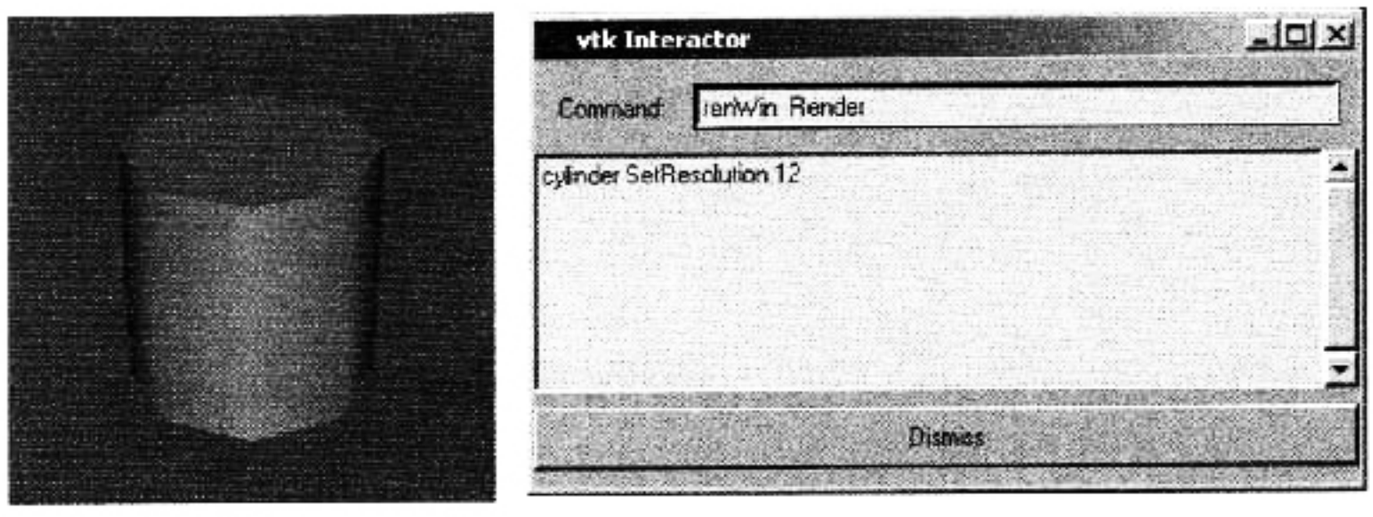

Figure 4–1 Using Tcl and Tk to build an interpreted application.

4.1 Creating Simple Models

The use of the *Visualization Toolkit* typically goes as follows: read/generate some data, filter it, render it, and interact with it. In this section, we'll start by looking at ways to read and generate data.

There are two basic ways to obtain data. The data may exist in a file (or files, streams, etc.) that is read into the system; or, the data may be procedurally generated (via an algorithm or mathematical expression). Recall that objects that initiate the processing of data in the visualization pipeline are called source objects (see **Figure 3–5**). Objects that generate data are called procedural (source) objects, and objects that read data are called reader (source) objects.

Procedural Source Object

We'll start off by rendering a simple cylinder. The example code shown below (`VTK/Examples/Rendering/Tcl/Cylinder.tcl`) demonstrates many basic concepts in the visualization and graphics systems. Refer to **Figure 4–1** to see the results of running the script.

We begin the script by invoking a Tcl command to load the VTK package (`package require vtk`) and create a GUI interpreter (`package require vtkinteractor`) that lets you type commands at run-time. Also, we load `vtktesting` which defines a set of colors, one of which (tomato) is used later in the script.

Historical note: In previous versions of VTK, the shared VTK libraries were loaded by calling `catch {load vtktcl}` instead of `package require vtk`.

```
package require vtk
package require vtkinteraction
package require vtktesting
```

We then create a procedural source object: vtkCylinderSource. This source creates a polygonal representation of a cylinder. The output of the cylinder is set as the input to the vtkPolyDataMapper via the method SetInput(). We create an actor (the object that is rendered) that refers to the mapper as its defining geometry. Notice the way objects are constructed in Tcl: we use the class name followed by the desired instance name:

```
vtkCylinderSource cylinder
    cylinder SetResolution 8
vtkPolyDataMapper cylinderMapper
    cylinderMapper SetInput [cylinder GetOutput]
vtkActor cylinderActor
    cylinderActor SetMapper cylinderMapper
    eval [cylinderActor GetProperty] SetColor $tomato
    cylinderActor RotateX  30.0
    cylinderActor RotateY -45.0
```

(As a reminder of how similar a C++ implementation is to a Tcl (or other interpreted languages) implementation, the same code implemented in C++ is shown below, and can be found in `VTK/Examples/Rendering/Cxx/Cylinder.cxx`.)

```
vtkCylinderSource *cylinder = vtkCylinderSource::New();
    cylinder->SetResolution(8);
vtkPolyDataMapper *cylinderMapper = vtkPolyDataMapper::New();
    cylinderMapper->SetInput(cylinder->GetOutput());
vtkActor *cylinderActor = vtkActor::New();
    cylinderActor->SetMapper(cylinderMapper);
    cylinderActor->GetProperty()->SetColor(1.0000, 0.3882, 0.2784);
    cylinderActor->RotateX(30.0);
    cylinderActor->RotateY(-45.0);
```

Recall that source objects initiate the visualization pipeline, and mapper objects terminate the pipeline, so in this example we have a pipeline consisting of two process objects (i.e., a source and mapper). The VTK pipeline uses a lazy evaluation scheme, so even though the pipeline is connected, no generation or processing of data has yet occurred (since we have not yet requested the data).

Next we create graphics objects which will allow us to render the actor. The vtkRenderer instance `ren1` coordinates the rendering process for a viewport of the render window

renWin. The render window interactor iren is a 3D widget that allows us to manipulate the camera.

```
#Create the graphics stuff
#
vtkRenderer ren1
vtkRenderWindow renWin
    renWin AddRenderer ren1
vtkRenderWindowInteractor iren
    iren SetRenderWindow renWin
```

Notice that we've associated the renderer with the render window via the AddRenderer() method. We must also associate the actor with the renderer using the AddActor() method.

```
# 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
```

The SetBackground() method specifies the background color of the rendering window using RGB (red, green, blue) values between (0,1), and SetSize() specifies the window size in pixels. Finally, we conclude this example by associating the GUI interactor with the render window interactor's user-defined method. (The user-defined method is invoked by pressing the u key when the mouse focus is in the rendering window. See "Using VTK Interactors" on page 48.) The Initialize() method begins the event loop, and the Tcl/Tk command wm withdraw . makes sure that the interpreter widget .Interact is not visible when the application starts.

```
# 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 .
```

When the script is run, the visualization pipeline will execute because the rendering process will request data. (The window expose event will force the render window to render itself.) Only after the pipeline executes are the filters up-to-date with respect to the input data. If you desire, you can manually cause execution of the pipeline by invoking renWin Render.

After you get this example running, you might try a couple of things. First, use the interactor by mousing in the rendering window. Next, change the resolution of the cylinder object by invoking the `cylinder SetResolution 12`. You can do this by editing the example file and re-executing it, or by pressing `u` in the rendering window to bring up the interpreter GUI and typing the command there. Remember, if you are using the Tcl interactor popup, the changes you make are visible only after data is requested, so follow changes with a `renWin Render` command, or by mousing in the rendering window.

Reader Source Object

This example is similar to the previous example except that we read a data file rather than procedurally generating the data. A stereo-lithography file is read (suffix `.stl`) that represents polygonal data using the binary STL data format. (Refer to **Figure 4–2** and the Tcl script `VTK/Examples/Rendering/Tcl/CADPart.tcl`.)

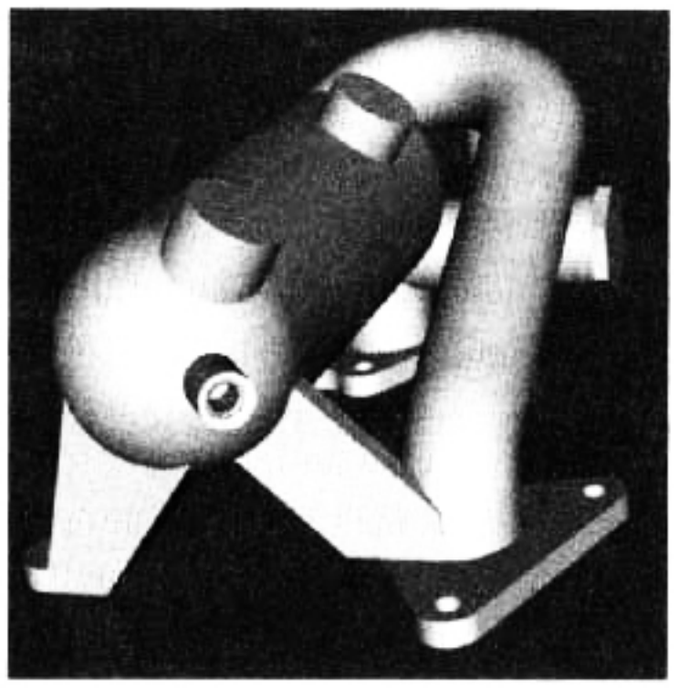

Figure 4–2 Reader source object.

```
vtkSTLReader part
    part SetFileName $VTK.-DATA-ROOT/Data/42400-IDGH.stl
vtkPolyDataMapper   partMapper
    partMapper SetInput [part GetOutput]
vtkLODActor partActor
    partActor SetMapper partMapper
```

Notice the use of the vtkLODActor. This actor changes its representation to maintain interactive performance. Its default behavior is to create a point cloud and wireframe, bounding-box outline to represent the intermediate and low-level representations. (See "Level-Of-Detail Actors" on page 61 for more information.) You may want to try replacing vtkLODActor with an instance of vtkActor to see the difference.

Many of the readers do not sense when the input file(s) change and re-execute. For example, if the file `42400-IDGH.stl` changes, the pipeline will not re-execute. You can manually modify objects by invoking the Modified() method on them. This will cause the filter to re-execute, as well as all filters downstream of it.

The *Visualization Toolkit* has limited, built-in modeling capabilities. If you want to use VTK to edit and manipulate complex models (e.g., those created by a solid modeler or modeling tool), you'll typically use a reader (see "Readers" on page 185) to interface to the data. (Another option is importers, which are used to ingest entire scenes. See "Importers" on page 189 for more information.)

4.2 Using VTK Interactors

Once you've visualized your data, you typically want to interact with it. The *Visualization Toolkit* offers several approaches to do this. The first approach is to use the built in class vtkRenderWindowInteractor. The second approach is to create your own interactor by specifying event bindings. And don't forget (if you are using an interpreted language) that you can type commands at run-time. You may also wish to refer to "Picking" on page 65 to see how to select data from the screen. (Note: Developers can also interface to a windowing system of their choice. See "Integrating With The Windowing System" on page 303.)

vtkRenderWindowInteractor

The simplest way to interact with your data is to instantiate vtkRenderWindowInteractor. This class responds to pre-defined set of events and actions, and provides a way to override the default actions. vtkRenderWindowInteractor allows you to control the camera and actors, as well as offering two interaction styles: position sensitive (i.e., joystick mode); and motion sensitive (i.e., trackball mode). (More about interactor styles shortly.)

vtkRenderWindowInteractor responds to the following events in the render window. (Remember that multiple renderers can draw into a rendering window, and that the renderer draws into a viewport within the render window. Interactors support multiple renderers in a render window.)

- `Keypress j` / `Keypress t` — toggle between **joystick** (position sensitive) and **trackball** (motion sensitive) **styles**. In joystick style, motion occurs continuously as long as a mouse button is pressed. In trackball style, motion occurs when the mouse button is pressed and the mouse pointer moves.
- `Keypress c` / `Keypress a` — toggle between **camera** and **actor** (object) **modes**. In camera mode, mouse events affect the camera position and focal point. In object mode, mouse events affect the actor that is under the mouse pointer.

- `Button 1` — **rotate** the camera around its focal point (if camera mode) or rotate the actor around its origin (if actor mode). The rotation is in the direction defined from the center of the renderer's viewport towards the mouse position. In joystick mode, the magnitude of the rotation is determined by the distance the mouse is from the center of the render window.
- `Button 2` — **pan** the camera (if camera mode) or **translate** the actor (if object mode). In joystick mode, the direction of pan or translation is from the center of the viewport towards the mouse position. In trackball mode, the direction of motion is the direction the mouse moves. (Note: with a 2-button mouse, pan is defined as `<Shift>-Button 1`.)
- `Button 3` — **zoom** the camera (if camera mode) or **scale** the actor (if object mode). Zoom in/increase scale if the mouse position is in the top half of the viewport; zoom out/decrease scale if the mouse position is in the bottom half. In joystick mode, the amount of zoom is controlled by the distance of the mouse pointer from the horizontal centerline of the window.
- `Keypress 3` — **toggle** the render window into and out of **stereo mode**. By default, red-blue stereo pairs are created. Some systems support Crystal Eyes LCD stereo glasses; you have to invoke `SetStereoTypeToCrystalEyes()` on the rendering window.
- `Keypress e` — **exit** the application.
- `Keypress f` — **fly-to** the point under the cursor. This sets the focal point and allows rotations around that point.
- `Keypress p` — perform a **pick** operation. The render window interactor has an internal instance of `vtkPropPicker` that it uses to pick. See "Picking" on page 65 for more information about picking.
- `Keypress r` — **reset** the camera view along the current view direction. Centers the actors and moves the camera so that all actors are visible.
- `Keypress s` — modify the representation of all actors so that they are **surfaces**.
- `Keypress u` — invoke the **user-defined method**. Typically, this keypress will bring up an interactor that you can type commands into.
- `Keypress w` — modify the representation of all actors so that they are **wireframe**.

The default interaction style is position sensitive (i.e., joystick style)—that is, it manipulates the camera or actor and renders continuously as long as a mouse button is pressed. If

you don't like the default behavior, you can change it, or write your own. (See "vtkRenderWindow Interaction Style" on page 303 for information about writing your own style.)

vtkRenderWindowInteractor has other useful features. Invoking LightFollowCameraOn() (the default behavior) causes the light position and focal point to be synchronized with the camera position and focal point (i.e., a "headlight" is created). Of course, this can be turned off with LightFollowCameraOff(). A callback that responds to the "u" keypress can be added with "AddObserver(UserEvent) method. It is also possible to set several pick-related methods. AddObserver(StartPickEvent) defines a method to be called prior to picking, and AddObserver(EndPickEvent) defines a method after the pick has been performed. (Please see "User Methods, Observers, and Commands" on page 28 for more information on defining user methods.) You can also specify an instance of a subclass of vtkAbstractPicker to use via the SetPicker() method (see "Picking" on page 65).

If you are using vtkLODActor, you may wish to set the desired frame rate via SetDesiredUpdateRate() in the interactor. Normally, this is handled automatically (when the mouse buttons are activated, the desired update rate is increased, when the mouse button is released, the desired update rate is set back down). Refer to "Level-Of-Detail Actors" on page 61 for more information.

We've seen how to use vtkRenderWindowInteractor previously, here's a recapitulation.

```
vtkRenderWindowInteractor iren
   iren SetRenderWindow renWin
   iren AddObserver UserEvent {wm deiconify .vtkInteract}
```

Interactor Styles

There are two distinctly different ways to control interaction style in VTK. The first is to use a subclass of vtkInteractorStyle, either one supplied with the system or one that you write. The second method is to manage the event loop directly.

vtkInteractorStyle. The class vtkRenderWindowInteractor can support different interaction styles. When you type "t" or "j" in the interactor (see the previous section) you are changing between trackball and joystick interaction styles. The way this works is that vtkRenderWindowInteractor forwards any events it receives (e.g., mouse button press, mouse motion, keyboard events, etc.) to its style. The style is then responsible for handling the events and performing the correct actions. To set the style, use the vtkRenderWindowInteractor::SetInteractorStyle() method. For example:

```
vtkInteractorStyleFlight flightStyle
vtkRenderWindowInteractor iren
    iren SetInteractorStyle flightStyle
```

(Note: When vtkRenderWindowInteractor is instantiated, a window-system specific render window interactor is actually instantiated. For example, on Unix systems the class vtkXRenderWindowInteractor is actually created, and returned as an instance of vtkRenderWindowInteractor. On Windows, the class vtkWin32RenderWindowInteractor is instantiated.)

Managing The Event Loop. You may wish to create your own event bindings. The bindings can be managed in any language that VTK supports, including C++, Tcl, Python, and Java. (Normally you'll do this in the system with which you're building the GUI. For more information about interfacing VTK to the windowing system, see "Integrating With The Windowing System" on page 303.) One example to look at is `Wrapping/Tcl/TkInteractor.tcl`, which defines bindings for Tcl/Tk. Here's a portion of that example to give you an idea of what's going on. The `bind` command is a Tcl/Tk command that associates events in a widget with a callback or function invocation.

```
proc BindTkRenderWidget {widget} {
    bind $widget <Any-ButtonPress> {StartMotion %W %x %y}
    bind $widget <Any-ButtonRelease> {EndMotion %W %x %y}
    bind $widget <B1-Motion> {Rotate %W %x %y}
    bind $widget <B2-Motion> {Pan %W %x %y}
    bind $widget <B3-Motion> {Zoom %W %x %y}
    bind $widget <Shift-B1-Motion> {Pan %W %x %y}
    bind $widget <KeyPress-r> {Reset %W %x %y}
    bind $widget <KeyPress-u> {wm deiconify .vtkInteract}
    bind $widget <KeyPress-w> {Wireframe %W}
    bind $widget <KeyPress-s> {Surface %W}
    bind $widget <KeyPress-p> {PickActor %W %x %y}
    bind $widget <Enter> {Enter %W %x %y}
    bind $widget <Leave> {focus $oldFocus}
    bind $widget <Expose> {Expose %W}
}
```

Notice that the bindings are quite similar to vtkRenderWindowInteractor. These bindings are like the trackball interactor style; that is, the camera does not change unless the mouse is moved.

4.3 Filtering Data

The previous examples consisted of a source and mapper object; the pipeline had no filters. In this section we show how to add a filter into the pipeline.

Filters are connected by using the SetInput() and GetOutput() methods. For example, we can modify the script in "Reader Source Object" on page 46 to shrink the polygons that make up the model. The script is shown below. (Only the pipeline and other pertinent objects are shown.) The complete script can be found at `VTK/Examples/Rendering/Tcl/FilterCADPart.tcl`.

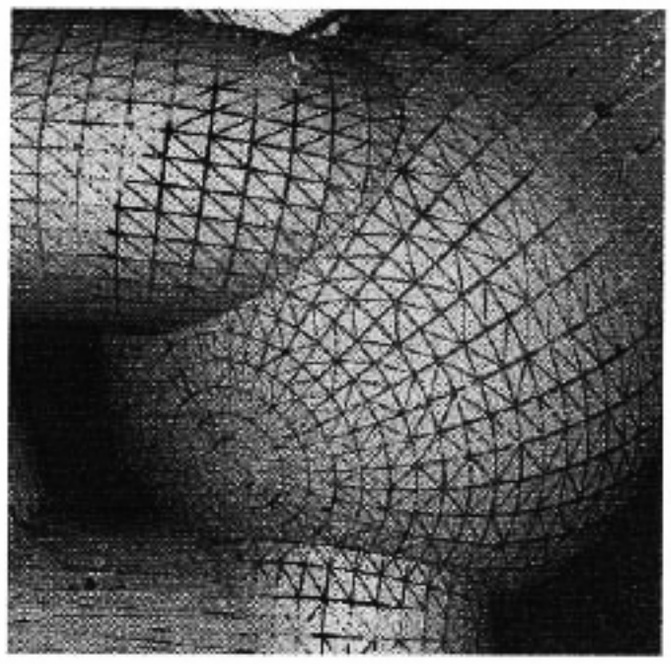

Figure 4–3 Filtering data. Here we use a filter to shrink the polygons forming the model towards their centroid.

```
vtkSTLReader part
    part SetFileName "$VTK_DATA_ROOT/Data/
42400-IDGH.stl"
vtkShrinkPolyData shrink
    shrink SetInput [part GetOutput]
    shrink SetShrinkFactor 0.85
vtkPolyDataMapper partMapper
    partMapper SetInput [shrink GetOutput]
vtkLODActor partActor
    partActor SetMapper partMapper
```

As you can see, creating a visualization pipeline is simple. You need to select the right classes for the task at hand, make sure that the input and output type of connected filters are compatible, and set the necessary instance variables. (Input and output types are compatible when the output dataset type is of the same type as the input will accept, or a subclass of the input type.) Visualization pipelines can contain loops, although the output of a filter cannot be directly connected to its input.

4.4 Controlling The Camera

You may have noticed that in the proceeding scripts no cameras or lights were instantiated. If you're familiar with 3D graphics, you know that lights and cameras are necessary to render objects. In VTK, if lights and cameras are not directly created, the renderer automatically instantiates them.

Instantiating The Camera

The following Tcl script shows how to instantiate and associate a camera with a renderer.

```
vtkCamera cam1
    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
```

Alternatively, if you wish to access a camera that already exists (for example, a camera that the renderer has automatically instantiated), in Tcl you would use

```
set cam1 [ren1 GetActiveCamera]
$cam1 Zoom 1.4
```

Let's review some of the camera methods that we've just introduced. SetClippingPlane() takes two arguments, the distance to the near and far clipping planes along the view plane normal. Recall that all graphics primitives not between these planes are eliminated during rendering, so you need to make sure the objects you want to see lie between the clipping planes. The FocalPoint and Position (in world coordinates) instance variables control the direction and position of the camera. ComputeViewPlaneNormal() resets the normal to the view plane based on the current position and focal point. (If the view plane normal is not perpendicular to the view plane you can get some interesting shearing effects.) Setting the ViewUp controls the "up" direction for the camera. Finally, the Zoom() method magnifies objects by changing the view angle (e.g., SetViewAngle()). You can also use the Dolly() method to move the camera in and out along the view plane normal and either enlarge or shrink the visible actors.

Simple Manipulation Methods

The methods described above are not always the most convenient ones for controlling the camera. If the camera is "looking" the point you want (i.e., the focal point is set), you can use the Azimuth() and Elevation() methods to move the camera about the focal point.

```
cam1 Azimuth 150
cam1 Elevation 60
```

These methods move the camera in a spherical coordinate system centered at the focal point by moving in the longitude direction (azimuth) and the latitude direction (elevation) by the angle (in degrees) given. These methods depend on the view-up vector remaining constant, and do not modify the view-up vector. Note that there are singularities at the north and south pole—the view-up vector becomes parallel with the view plane normal. To avoid this, you can force the view-up vector to be orthogonal to the view vector by using OrthogonalizeViewUp(). However, this changes the camera coordinate system, so if you're flying around an object with a natural horizon or view-up vector (such as terrain), camera manipulation is no longer natural with respect to the data.

Controlling The View Direction

A common function of the camera is to generate a view from a particular direction. You can do this by invoking SetFocalPoint(), SetPosition(), and ComputeViewPlaneNormal() followed by invoking ResetCamera() on the renderer associated with the camera.

```
vtkCamera cam1
   cam1 SetFocalPoint 0 0 0
   cam1 SetPosition 1 1 1
   cam1 ComputeViewPlaneNormal
   cam1 SetViewUp 1 0 0
   cam1 OrthogonalizeViewUp
ren1 SetActiveCamera cam1
ren1 ResetCamera
```

The initial direction (view vector or view plane normal) is computed from the focal point and position of the camera, which, together with ComputeViewPlaneNormal(), defines the initial view vector. Optionally, you can specify an initial view-up vector and orthogonalize it with respect to the view vector. The ResetCamera() method then moves the camera along the view vector so that the renderer's actors are all visible to the camera.

Perspective Versus Orthogonal Views

In the examples shown thus far, we have assumed that the camera is a perspective camera; that is, a view angle controls the projection of the actors onto the view plane during the rendering process. Perspective projection, while generating more natural looking images, introduces distortion that can be undesirable in some applications. Orthogonal (or parallel)

projection is an alternative projection method. In orthogonal projection, view rays are parallel, and objects are rendered without distance effects.

To set the camera to use orthogonal projection, use the vtkCamera::ParallelProjectionOn() method. In parallel projection mode, the camera view angle is no longer effective for controlling zoom. Instead, use the SetParallelScale() method to control the magnification of the actors.

Saving/Restoring Camera State

Another common requirement of applications is the capability to save and restore camera state (i.e., recover a view). To save camera state, you'll need to save (at a minimum) the clipping range, the focal point and position, and the view-up vector. You'll also want to compute the view plane normal (as shown in the example in "Instantiating The Camera" on page 53). Then, to recover camera state, simply instantiate a camera with the saved information and assign it to the appropriate renderer (i.e., SetActiveCamera()).

In some cases you may need to store additional information. For example, if the camera view angle (or parallel scale) is set, you'll need to save these. Or, if you are using the camera for stereo viewing, the EyeAngle and Stereo flag are required.

4.5 Controlling Lights

Lights are easier to control than cameras. The most frequently used methods are SetPosition(), SetFocalPoint(), and SetColor(). The position and focal point of the light control the direction in which the light points. The color of the light is expressed as a RGB vector. Also, lights can be turned on and off via the SwitchOn() and SwitchOff() methods, and the brightness of the light can be set with the SetIntensity() method.

By default, instances of vtkLight are directional lights. That is, the position and focal point define a vector parallel to which light rays travel, and the light source is assumed to be located at the infinity point. This means that the lighting on an object does not change if the focal point and position are translated identically.

Lights are associated with renderers as follows.

```
vtkLight light
   light SetColor 1 0 0
```

```
    light SetFocalPoint [cam1 GetFocalPoint]
    light SetPosition [cam1 GetPosition]

ren1 AddLight light
```

Here we've created a red headlight: a light located at the camera's (`cam1`'s) position and pointing towards the camera's focal point. This is a useful trick, and is used by the interactive renderer to position the light as the camera moves (see "Using VTK Interactors" on page 48).

Positional Lights

It is possible to create positional (i.e., spot lights) by using the PositionalOn() method. This method is used in conjunction with the SetConeAngle() method to control the spread of the spot. A cone angle of 180 degrees indicates that no spot light effects will be applied (i.e., truncated light cone); only the effects of position.

4.6 Controlling 3D Props

Objects in VTK that are to be drawn in the render window are generically known as "props." (The word prop comes from the vocabulary of theatre—a prop is something that appears on stage.) There are several different types of props including vtkProp3D and vtkActor. vtkProp3D is an abstract superclass for those types of props existing in 3D space. (vtkProp3D has a 4x4 transformation matrix that supports scaling, translating, rotating, and geometric projection in 3D space.) The class vtkActor is a type of vtkProp3D whose geometry is defined by analytic primitives such as polygons and lines. We will examine vtkActor and other types of vtkProp3D's later in this section.

Specifying the Position of a vtkProp3D

We have already seen how to use cameras to move around an object; alternatively, we can also hold the camera steady and transform the props. The following methods can be used to define the position of a vtkProp3D (and its subclasses).

- `SetPosition(x,y,z)` — Specify the position of the vtkProp3D in world coordinates.

- `AddPosition(deltaX,deltaY,deltaZ)` — Translate the prop by the specified amount along each of the x, y, and z axes.
- `RotateX(theta)`, `RotateY(theta)`, `RotateZ(theta)` — Rotate the prop by theta degrees around the *x*, *y*, *z* coordinate axes, respectively.
- `SetOrientation(x,y,z)` — Set the orientation of the prop by rotating about the *z* axis, them about the *x* axis, and then about the *y* axis.
- `AddOrientation(a1,a2,a3)` — Add to the current orientation of the prop.
- `RotateWXYZ(theta,x,y,z)` — Rotate the prop by theta degress around the *x*-*y*-*z* vector defined.
- `Scale(sx,sy,sz)` — Scale the prop in the *x*, *y*, *z* axes coordinate directions.
- `SetOrigin(x,y,z)` — Specify the origin of the prop. The origin is the point around which rotations and scaling occurs.

These methods work together in complex ways to control the resulting transformation matrix. The most important thing to remember is that the operations listed above are applied in a particular order, and the order of applications dramatically affects the resulting actor position. The order used in VTK to apply these transformations is as follows:

Shift to Origin →*Scale* → *Rotate Y* → *Rotate X* → *Rotate Z* →
Shift from Origin → *Translate*

The shift to and from the origin is a negative and positive translation of the Origin value, respectively. The net translation is given by the Position value of the vtkProp3D. The most confusing part of these transformations are the rotations. For example, performing an *x* rotation followed by a *y* rotation gives very different results than the operations applied in reverse order (see **Figure 4–4**). For more information about actor transformation, please refer to page 74 of the *Visualization Toolkit* text.

In the next section we describe a variety of vtkProp3D's—of which the most widely used class in VTK is called vtkActor. Later on (see "Controlling vtkActor2D" on page 70) we will examine 2D props (i.e., vtkActor2D) which tend to be used for annotation and other 2D operations.

Figure 4–4 The effects of applying rotation in different order. On the left, first a x rotation followed by a y rotation; on the right, first a y rotation followed by a x rotation.

Actors

An actor is the most common type of vtkProp3D. Like other concrete subclasses of vtkProp3D, vtkActor serves to group rendering attributes such as surface properties (e.g., ambient, diffuse, specular color), representation (e.g., surface or wireframe), texture maps, and/or a geometric definition (a mapper).

Defining Geometry. As we have seen in previous examples, the geometry of an actor is specified with the SetMapper() method:

```
vtkPolyDataMapper mapper
   mapper SetInput [aFilter GetOutput]
vtkActor anActor
   anActor SetMapper mapper
```

In this case `mapper` is of type vtkPolyDataMapper, which renders geometry using analytic primitives such as points, lines, polygons, and triangle strips. The mapper terminates the visualization pipeline, and serves as the bridge between the visualization subsystem and the graphics subsystem.

Actor Properties. Actors refer to an instance of vtkProperty, which in turn controls the appearance of the actor. Probably the most used property is actor color, which we will describe in the next section. Other important features of the property are its representation (points, wireframe, or surface), its shading method (either flat or Gouraud shaded), the actor's opacity (relative transparency), and the ambient, diffuse, and specular color

and related coefficients. The following script shows how to set some of these instance variables.

```
vtkActor anActor
   anActor SetMapper mapper
   [anActor GetProperty] SetOpacity 0.25
   [anActor GetProperty] SetAmbient 0.5
   [anActor GetProperty] SetDiffuse 0.6
   [anActor GetProperty] SetSpecular 1.0
   [anActor GetProperty] SetSpecularPower 10.0
```

Notice how we dereference the actor's property via the GetProperty() method. Alternatively, we can create a property and assign it to the actor:

```
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
```

The advantage of the latter method is that we control the properties of several actors by assigning each the same property.

Actor Color. Color is perhaps the most important property applied to an actor. The simplest procedure is to use the SetColor() method to set the red, green, and blue (RGB) values of the actor. Each value ranges from zero to one.

```
   anActor SetColor 0.1 0.2 0.4
```

Alternatively, you can set the ambient, diffuse, and specular colors separately.

```
vtkActor anActor
   anActor SetMapper mapper
   [anActor GetProperty] SetAmbientColor .1 .1 .1
   [anActor GetProperty] SetDiffuseColor .1 .2 .4
   [anActor GetProperty] SetSpecularColor 1 1 1
```

In this example we've set the ambient color to a dark gray, the diffuse color to a shade of blue, and the specular color to white. (Note: The SetColor() method sets the ambient, diffuse, and specular colors to the color specified.)

Important: The color set in the actor's property only takes effect if there is no scalar data available to the actor's mapper. By default, the mapper's input scalar data colors the actor, and the actor's color is ignored. To ignore the scalar data, use the method ScalarVisibilityOff() as shown in the Tcl script below.

```
vtkPolyDataMapper planeMapper
    planeMapper SetInput [CompPlane GetOutput]
    planeMapper ScalarVisibilityOff
vtkActor planeActor
    planeActor SetMapper planeMapper
    [planeActor GetProperty] SetRepresentationToWireframe
    [planeActor GetProperty] SetColor 0 0 0
```

Actor Transparency. Many times it is useful to adjust transparency (or opacity) of an actor. For example, if you wish to show internal organs surrounded by the skin of a patient, adjusting the transparency of the skin allows the user to see the organs in relation to the skin. Use the vtkProperty::SetOpacity() method as follows:

```
vtkActor popActor
    popActor SetMapper popMapper
    [popActor GetProperty] SetOpacity 0.3
    [popActor GetProperty] SetColor .9 .9 .9
```

(Please note that transparency is implemented in the rendering library using an α-blending process. This process requires that polygons are rendered in the correct order. In practice, this is very difficult to achieve, especially if you have multiple transparent actors. To order polygons, you should add transparent actors to the end of renderer's list of actors (i.e., add them last). Also, you can use the filter vtkDepthSortPolyData to sort polygons along the view vector. Please see `VTK/Examples/VisualizationAlgorithms/Tcl/DepthSort.tcl` for an example using this filter.)

Miscellaneous Features. Actors have several other important features. You can control whether an actor is visible with the VisibilityOn() and VisibilityOff() methods. If you don't want to pick an actor during a picking operation, use the PickableOff() method (see "Picking" on page 65 for more information about picking). Actors also have a pick event that can be invoked when they are picked. You can also get the axis-aligned bounding box of actor with the GetBounds() method.

Level-Of-Detail Actors

One major problem with graphics systems is that they often become too slow for interactive use. To handle this problem, VTK uses level-of-detail actors to achieve acceptable rendering performance at the cost of lower-resolution representations.

In "Reader Source Object" on page 47 we saw how to use a vtkLODActor. Basically, the simplest way to use vtkLODActor is to replace instances of vtkActor with instances of vtkLODActor. In addition, you can control the representation of the levels of detail. The default behavior of vtkLODActor is to create two additional, lower-resolution models from the original mapper. The first is a point cloud, sampled from the points defining the mapper's input. You can control the number of points in the cloud as follows. (The default is 150 points.)

```
vtkLODActor dotActor
   dotActor SetMapper dotMapper
   dotActor SetNumberOfCloudPoints 1000
```

The lowest resolution model is a bounding box of the actor. Additional levels of detail can be added using the AddLODMapper() method. They do not have to be added in order of complexity.

To control the level-of-detail selected by the actor during rendering, you can set the desired frame rate in the rendering window:

```
vtkRenderWindow renWin
   renWin SetDesiredUpdateRate 5.0
```

which translates into five frames per second. The vtkLODActor will automatically select the appropriate level-of-detail to yield the requested rate. (Note: The interactor widgets such as vtkRenderWindowInteractor automatically control the desired update rate. They typically set the frame rate very low when a mouse button is released, and increase the rate when a mouse button is pressed. This gives the pleasing effect of low-resolution/high frame rate models with camera motion, and high-resolution/low frame rate when the camera stops. If you would like more control over the levels-of-detail, see "vtkLODProp3D" on page 64. vtkLODProp3D allow you to specifically set each level.)

Assemblies

Actors are often grouped in hierarchal assemblies so that the motion of one actor affects the position of other actors. For example, a robot arm might consist of an upper arm, forearm, wrist, and end effector, all connected via joints. When the upper arm rotates around the shoulder joint, we expect the rest of the arm to move with it. This behavior is implemented using assemblies, which are a type of (subclass of) vtkActor. The following script shows how it's done (from `VTK/Examples/Rendering/Tcl/assembly.tcl`).

```
# create four parts: a top level assembly and three primitives
vtkSphereSource sphere
vtkPolyDataMapper sphereMapper
    sphereMapper SetInput [sphere GetOutput]
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 SetInput [cube GetOutput]
vtkActor cubeActor
    cubeActor SetMapper cubeMapper
    cubeActor SetPosition 0.0 .25 0
    [cubeActor GetProperty] SetColor 0 0 1

vtkConeSource cone
vtkPolyDataMapper coneMapper
    coneMapper SetInput [cone GetOutput]
vtkActor coneActor
    coneActor SetMapper coneMapper
    coneActor SetPosition 0 0 .25
    [coneActor GetProperty] SetColor 0 1 0

vtkCylinderSource cylinder
vtkPolyDataMapper cylinderMapper
    CylinderMapper SetInput [cylinder GetOutput]
vtkActor cylinderActor
    cylinderActor SetMapper cylinderMapper
    [cylinderActor GetProperty] SetColor 1 0 0
```

```
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

# Add the actors to the renderer, set the background and size
ren1 AddActor assembly
ren1 AddActor coneActor
```

Notice how we use vtkAssembly's AddPart() method to build the hierarchies. Assemblies can be nested arbitrarily deeply as long as there are not any self-referencing cycles. Note that vtkAssembly is a subclass of vtkProp3D, so it has no notion of properties or of an associated mapper. Therefore, the leaf nodes of the vtkAssembly hierarchy must carry information about material properties (color, etc.) and any associated geometry. Actors may also be used by more than one assembly (notice how `coneActor` is used in the assembly and as an actor). Also, the renderer's AddActor() method is used to associate the top level of the assembly with the renderer; those actors at lower levels in the assembly hierarchy do not need to be added to the renderer since they are recursively rendered.

You may be wondering how to distinguish the use of an actor relative to its context if an actor is used in more than one assembly, or is mixed with an assembly as in the example above. (This is particularly important in activities like picking, where the user may need to know which vtkProp was picked as well as the context in which it was picked.) We address this issue along with the introduction of the class vtkAssemblyPath, which is an ordered list of vtkProps with associated transformation matrices (if any), in detail in "Picking" on page 65.

Volumes

The class vtkVolume is used for volume rendering. It is analogous to the class vtkActor. Like vtkActor, vtkVolume inherits methods from vtkProp3D to position and orient the volume. vtkVolume has an associated property object, in this case a vtkVolumeProperty. Please see "Volume Rendering" on page 136 for a thorough description of the use of vtkVolume and a description of volume rendering.

vtkLODProp3D

The vtkLODProp3D class is similar to vtkLODActor (see "Level-Of-Detail Actors" on page 61) in that it uses different representations of itself in order to achieve interactive frame rates. Unlike vtkLODActor, vtkLODProp3D supports both volume rendering and surface rendering. This means that you can use vtkLODProp3D in volume rendering applications to achieve interactive frame rates. The following example shows how to use the class.

```
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]
```

Basically, you create different mappers each corresponding to a different rendering complexity, and add the mappers to the vtkLODProp3D. The AddLOD() method accepts either volume or geometric mappers, and optionally a texture map and property object. (There are different signatures for this method depending on what information you wish to provide.) The last value in the field is an estimated time to render. Typically you set it to zero to indicate that there is no initial estimate. The method returns an integer id that can be used to access the appropriate LOD (i.e., to select a level or delete it).

vtkLODProp3D measures the time it takes to render each LOD and sorts them appropriately. Then, depending on the render window's desired update rate, vtkLODProp3D selects the appropriate level to render. See "Using a vtkLODProp3D to Improve Performance" on page 162 for more information.

4.7 Using Texture

Texture mapping is a powerful graphics tool for creating realistic and compelling visualizations. The basic idea behind 2D texture mapping is that images can be "pasted" onto a surface during the rendering process, thereby creating richer and more detailed images. Texture mapping requires two pieces of information: 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 a surface. (Note: 3D textures are also possible, but not yet widely supported by most rendering hardware.)

The following example (**Figure 4–5**) demonstrates the use of texture mapping (see `VTK/Examples/Rendering/Tcl/TPlane.tcl`). Notice that the texture map (of class vtkTexture) is associated with the actor, and the texture coordinates come from the plane (the texture coordinates are generated by vtkPlaneSource when the plane is created).

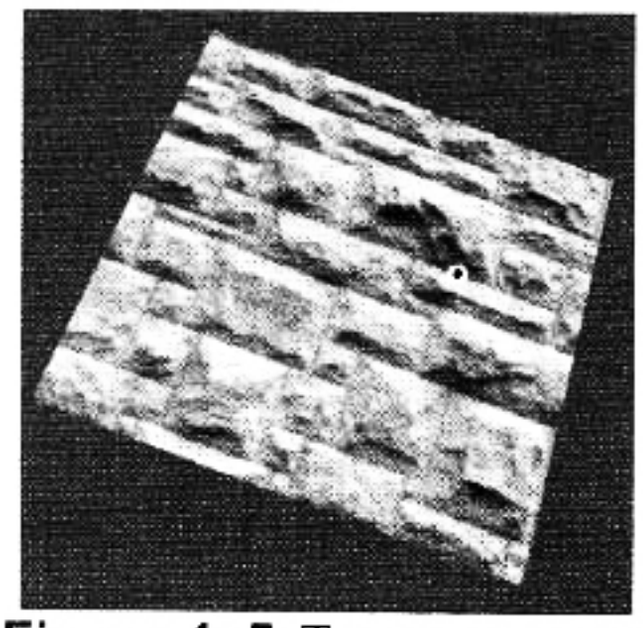

Figure 4–5 Texture map on plane

```
# load in the texture map
vtkBMPReader bmpReader
   bmpReader SetFileName "$VTK_DATA_ROOT/Data/
masonry.bmp"
vtkTexture atext
   atext SetInput [bmpReader GetOutput]
   atext InterpolateOn

# create a plane source and actor
vtkPlaneSource plane
vtkPolyDataMapper  planeMapper
   planeMapper SetInput [plane GetOutput]
vtkActor planeActor
   planeActor SetMapper planeMapper
   planeActor SetTexture atext
```

Often times texture coordinates are not available, usually because they are not generated in the pipeline. If you need to generate texture coordinates, refer to "Generate Texture Coordinates" on page 112. Also, you should note that while system graphics hardware/libraries (e.g., OpenGL) only accept texture maps that are powers of two in dimensions (e.g., 128 x 256 and so on), in VTK non-power of two textures are automatically resampled onto a power of two, which may impact performance in some applications.

4.8 Picking

Picking is a common visualization task. Picking is used to select data and actors, or interrogate underlying data values. A pick is made when a display position (i.e., pixel coordinate) is selected and used to invoke vtkAbstractPicker's Pick() method. Depending on the type of picking class, the information returned from the pick may be as simple as an *x*-*y*-*z* global coordinate, or may include cell ids, point ids, cell parametric coordinates, the instance of vtkProp that was picked, and/or assembly paths. The syntax of the pick method is as follows.

```
Pick(selectionX, selectionY, selectionZ, Renderer)
```

Notice that the pick method requires a renderer. The actors associated with the renderer are the candidates for pick selection. Also, `selectionZ` is typically set to 0.0—it relates to depth in the *z*-buffer. (In typical usage, this method is not invoked directly. Rather the user interacts with the class vtkRenderWindowInteractor which manages the pick. In this case, the user would control the picking process by assigning an instance of a picking class to the vtkRenderWindowInteractor, as we will see in the example following later.)

The *Visualization Toolkit* supports several types of pickers of varying functionality and performance. (Please see **Figure 14–12** which is an illustration of the picking class hierarchy.) The class vtkAbstractPicker serves as the base class for all pickers. It defines a minimal API which allows the user to retrieve the pick position (in global coordinates) using the GetPickPosition() method.

Three direct subclasses of vtkAbstractPicker exist. The first, vtkWorldPointPicker, is a fast (usually in hardware) picking class that uses the *z*-buffer to return the *x*-*y*-*z* global pick position. However, no other information (about the vtkProp that was picked, etc.) is returned. The class vtkAbstractPropPicker is another direct subclass of vtkAbstractPicker. It defines an API for pickers that can pick an instance of vtkProp. There are several convenience methods in this class to allow query for the return type of a pick. The functionality of these methods can be obtained by calling GetPath() on the picker and using the IsA() method to determine the type.

- `GetProp()` — Return the instance of vtkProp that was picked. If anything at all was picked, then this method will return a pointer to the instance of vtkProp, otherwise `NULL` is returned.
- `GetProp3D()` — If an instance of vtkProp3D was picked, return a pointer to the instance of vtkProp3D.
- `GetActor2D()` — If an instance of vtkActor2D was picked, return a pointer to the instance of vtkActor2D.
- `GetActor()` — If an instance of vtkActor was picked, return a pointer to the instance of vtkActor.
- `GetVolume()` — If an instance of vtkVolume was picked, return a pointer to the instance of vtkVolume.
- `GetAssembly()` — If an instance of vtkAssembly was picked, return a pointer to the instance of vtkAssembly.

- `GetPropAssembly()` — If an instance of vtkPropAssembly was picked, return a pointer to the instance of vtkPropAssembly.

A word of caution about these methods. The class (and its subclass) return information about the *top level of the assembly path* that was picked. So if you have an assembly whose top level is of type vtkAssembly, and whose leaf node is of type vtkActor, the method GetAssembly() will return a pointer to the instance of vtkAssembly, while the GetActor() method will return a `NULL` pointer (i.e., no vtkActor). If you have a complex scene that includes assemblies, actors, and other types of props, the safest course to take is to use the GetProp() method to determine whether anything at all was picked, and then use GetPath().

There is one direct subclass of vtkAbstractPropPicker. vtkPropPicker uses hardware picking to determine the instance of vtkProp that was picked, as well as the pick position (in global coordinates). It is generally faster than vtkPicker and its subclasses, but cannot return information about what cell was picked, etc. Warning: In some graphics hardware (especially lower-cost PC boards) the pick operation is not implemented properly. In this case, you will have to use a software version of vtkAbstractPicker (one of the three classes described next.)

The third subclass of vtkAbstractPicker is vtkPicker, a software-based picker that selects vtkProp's based on their bounding box. Its pick method fires a ray from the camera position through the selection point and intersects the bounding box of each prop 3D; of course, more than one prop 3D may be picked. The "closest" prop 3D in terms of its bounding box intersection point along the ray is returned. (The GetProp3Ds() method can be used to get all prop 3D's whose bounding box was intersected.) vtkPicker is fairly fast but cannot generate a single unique pick.

vtkPicker has two subclasses that can be used to retrieve more detailed information about what was picked (e.g., point ids, cell ids, etc.) vtkPointPicker selects a point and returns the point id and coordinates. It operates by firing a ray from the camera position through the selection point, and projecting those points that lie within Tolerance onto the ray. The projected point closest to the camera position is selected, along with its associated actor. (Note: The instance variable Tolerance is expressed as a fraction of the renderer window's diagonal length.) vtkPointPicker is slower than vtkPicker but faster than vtkCellPicker. It cannot always return a unique pick because of the tolerances involved.

vtkCellPicker selects a cell and returns information about the intersection point (cell id, global coordinates, and parametric cell coordinates). It operates by firing a ray and inter-

secting all cells in each actor's underlying geometry, determining if each intersects this ray, within a certain specified tolerance. The cell closest to the camera position along the specified ray is selected, along with its associated actor. (Note: The instance variable Tolerance is used during intersection calculation, and you may need to experiment with its value to get satisfactory behavior.) vtkCellPicker is the slowest of all the pickers, but provides the most information. It will generate a unique pick within the tolerance specified.

Several events are defined to interact with the pick operation. The picker invokes StartPickEvent prior to executing the pick operation. EndPickEvent is invoked after the pick operation is complete. The PickEvent and the actor's PickEvent are invoked each time an actor is picked.

vtkAssemblyPath

An understanding of the class vtkAssemblyPath is essential if you are to perform picking in a scene with different types of vtkProp's, especially if the scene contains instances of vtkAssembly. vtkAssemblyPath is simply an ordered list of vtkAssemblyNode's, where each node contains a pointer to a vtkProp, as well as an optional vtkMatrix4x4. The order of the list is important: the start of the list represents the root, or top level node in an assembly hierarchy, while the end of the list represents a leaf node in an assembly hierarchy. The ordering of the nodes also affects the associated matrix. Each matrix is a concatenation of the node's vtkProp's matrix with the previous matrix in the list. Thus, for a given vtkAssemblyNode, the associated vtkMatrix4x4 represents the position and orientation of the vtkProp (assuming that the vtkProp is initially untransformed).

Example

Typically, picking is automatically managed by vtkRenderWindowInteractor ("Using VTK Interactors" on page 48 for more information about interactors). For example, when pressing the `p` key, vtkRenderWindowInteractor invokes a pick with its internal instance of vtkPropPicker. You can then ask the vtkRenderWindowInteractor for its picker, and gather the information you need. You can also specify a particular vtkAbstractPicker instance for vtkRenderWindowInteractor to use, as the following script illustrates. The results on a sample data set are shown in **Figure 4–6**. The script for this example can be found in `VTK/Examples/Annotation/Tcl/annotatePick.tcl`.

```
vtkCellPicker picker
   picker AddObserver EndPickEvent annotatePick
vtkTextMapper textMapper
    textMapper SetFontFamilyToArial
    textMapper SetFontSize 10
    textMapper BoldOn
    textMapper ShadowOn
vtkActor2D textActor
    textActor VisibilityOff
    textActor SetMapper textMapper
    [textActor GetProperty] SetColor 1 0 0
vtkRenderWindowInteractor iren
    iren SetRenderWindow renWin
    iren SetPicker picker

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
}
picker Pick 85 126 0 ren1
```

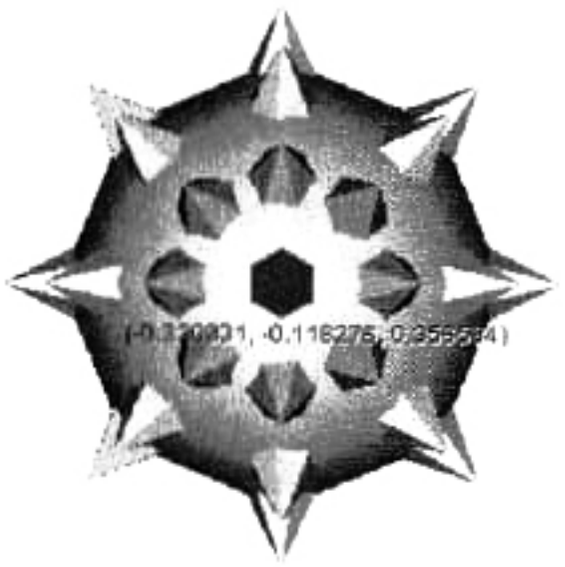

Figure 4–6 Annotating a pick operation.

This example uses a vtkTextMapper to draw the world coordinate of the pick on the screen (see "Annotation" on page 71 for more information). Notice that we register the EndPick-Event to perform setup after the pick occurs. The method is configured to invoke the `annotatePick()` procedure when picking is complete.

4.9 vtkCoordinate and Coordinate Systems

The *Visualization Toolkit* supports several different coordinate systems, and the class vtkCoordinate manages transformations between them. The supported coordinate systems are as follows.

- DISPLAY — *x-y* pixel values in the (rendering) window. (Note that vtkRenderWindow is a subclass of vtkWindow). The origin is the lower-left corner (which is true for all 2D coordinate systems described below).
- NORMALIZED DISPLAY — *x-y* (0,1) normalized values in the window.
- VIEWPORT — *x-y* pixel values in the viewport (or renderer — a subclass of vtkViewport)
- NORMALIZED VIEWPORT — *x-y* (0,1) normalized values in viewport
- VIEW — *x-y-z* (-1,1) values in camera coordinates (*z* is depth)
- WORLD — *x-y-z* global coordinate value
- USERDEFINED - *x-y-z* in user-defined space. The user must provide a transformation method for user defined coordinate systems. See vtkCoordinate for more information.

The class vtkCoordinate can be used to transform between coordinate systems, and linked together to form "relative" or "offset" coordinate values. Refer to the next section for an example of using vtkCoordinate in an application.

4.10 Controlling vtkActor2D

vtkActor2D is analogous to vtkActor, except that it draws on the overlay plane, and does not have a 4x4 transformation matrix associated with it. Like vtkActor, vtkActor2D refers to a mapper (vtkMapper2D) and a property object (vtkProperty2D). The most difficult part when working with vtkActor2D is positioning it. To do that, the class vtkCoordinate is used (See previous section, "vtkCoordinate and Coordinate Systems"). The following script shows how to use the vtkCoordinate object.

```
vtkActor2D bannerActor
    bannerActor SetMapper banner
    [bannerActor GetProperty] SetColor 0 1 0
    [bannerActor GetPositionCoordinate]
```

Figure 4–7 2D (left) and 3D (right) annotation.

```
                        SetCoordinateSystemToNormalizedDisplay
[bannerActor GetPositionCoordinate] SetValue 0.5 0.5
```

What's done in this script is to access the coordinate object and define it's coordinate system. Then the appropriate value is set for that coordinate system. In this script a normalized display coordinate system is used, so display coordinates range from zero to one, and the values (0.5,0.5) are set to position the vtkActor2D in the middle of the rendering window. vtkActor2D also provides a convenience method, SetDisplayPosition(), that sets the coordinate system to DISPLAY, and accepts the values as pixel offsets in the render window. The example in the following section shows how the method is used.

4.11 Annotation

The *Visualization Toolkit* offers two ways to annotate images. First, text (and graphics) can be rendered on top of the underlying 3D graphics window (often referred to as the overlay plane). Second, text can be created as 3D polygonal data, and transformed and displayed as any other 3D graphics object. We refer to this as 2D and 3D annotation, respectively. See **Figure 4–7** to see the difference.

2D Annotation

To use 2D annotation, we employ 2D actors (vtkActor2D and its subclasses such as vtkScaledTextActor) and mappers (vtkMapper2D and subclasses such as vtkTextMapper). 2D actors and mappers are similar to their 3D counterparts, except that they render in the

overlay plane on top of underlying graphics or images. Here's an example Tcl script found in `VTK/Examples/Annotation/Tcl/TestText.tcl`; the results are shown on the left side of **Figure 4–7**.

```
vtkSphereSource sphere
vtkPolyDataMapper   sphereMapper
    sphereMapper SetInput [sphere GetOutput]
    sphereMapper GlobalImmediateModeRenderingOn
vtkLODActor sphereActor
    sphereActor SetMapper sphereMapper

vtkTextMapper textMapper
    textMapper SetInput "This is a sphere"
    textMapper SetFontSize 18
    textMapper SetFontFamilyToArial
    textMapper BoldOn
    textMapper ItalicOn
    textMapper ShadowOn
vtkScaledTextActor text
    text SetMapper textMapper
    text SetDisplayPosition 90 50
    [text GetProperty] SetColor 0 0 1

# Create the RenderWindow, Renderer and both Actors
vtkRenderer ren1
vtkRenderWindow renWin
    renWin AddRenderer ren1
vtkRenderWindowInteractor iren
    iren SetRenderWindow renWin

# Add the actors to the renderer, set the background and size
ren1 AddActor2D text
ren1 AddActor sphereActor
```

Instances of the class vtkTextMapper allow you to control font family (Arial, Courier, or Times), turn bolding and italics on and off, and support font shadowing (shadowing is used to make the font more readable when placed on top of complex background images). The position and color of the text is controlled by the associated vtkActor2D (in this example, the position is set using display or pixel coordinates). Notice how the AddActor2D() method is used to associate 2D actors with the renderer.

vtkTextMapper also supports justification (vertical and horizontal) and multi-line text. Use the methods SetJustificationToLeft(), SetJustificationToCentered(), and SetJustifica-

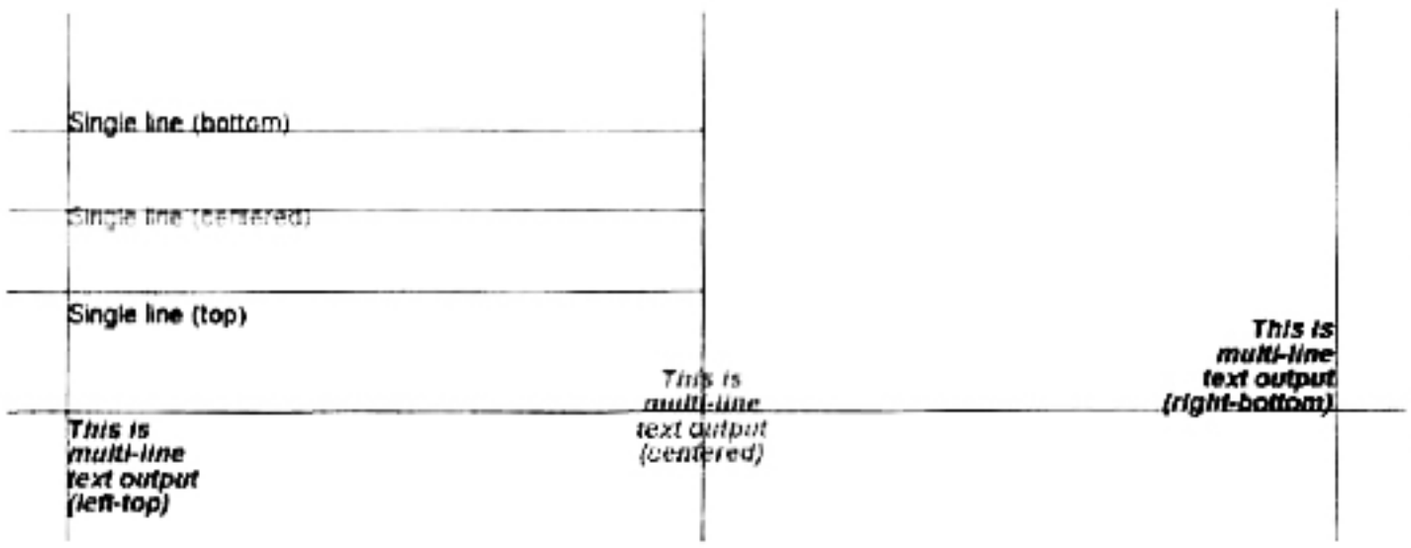

Figure 4–8 Justification and use of multi-line text. Use the \n character embedded in the text string to generate line breaks. Both vertical and horizontal justification is supported.

tionToRight() to control the horizontal justification. Use the methods SetVerticalJustificationToBottom(), SetVerticalJustificationToCentered(), and SetVerticalJustificationToTop() to control vertical justification. By default, text is left-bottom justified. To insert multi-line text, use the `\n` character embedded in the text. The example in **Figure 4–8** demonstrates justification and multi-line text (taken from `VTK/Examples/Annotation/Tcl/multiLineText.tcl`)

The essence of the example is shown below.

```
vtkTextMapper textMapperL
    textMapperL SetInput "This is\nmulti-line\n
                                text output\n(left-top)"
    textMapperL SetFontSize 14
    textMapperL SetFontFamilyToArial
    textMapperL BoldOn
    textMapperL ItalicOn
    textMapperL ShadowOn
    textMapperL SetJustificationToLeft
    textMapperL SetVerticalJustificationToTop
    textMapperL SetLineSpacing 0.8
vtkActor2D textActorL
    textActorL SetMapper textMapperL
    [textActorL GetPositionCoordinate]
                    SetCoordinateSystemToNormalizedDisplay
    [textActorL GetPositionCoordinate] SetValue 0.05 0.5
    [textActorL GetProperty] SetColor 1 0 0
```

Note the use of the vtkCoordinate object (obtained by invoking the GetPositionCoordinate() method) to control the position of the actor in the normalized display coordinate system. See the previous section "vtkCoordinate and Coordinate Systems" on page 70 for more information about placing annotation.

3D Annotation and vtkFollower

3D annotation is implemented using vtkVectorText to create a polygonal representation of a text string, and then appropriately positioned in the scene. One useful class for positioning 3D text is vtkFollower. This class is a type of actor that always faces the renderer's active camera, thereby insuring that the text is readable. This Tcl script found in `VTK/Examples/Annotation/Tcl/textOrigin.tcl` shows how to do this (**Figure 4–7**). The example creates an axes and labels the origin using an instance of vtkVectorText in combination with a vtkFollower.

```
vtkAxes axes
    axes SetOrigin 0 0 0
vtkPolyDataMapper axesMapper
    axesMapper SetInput [axes GetOutput]
vtkActor axesActor
    axesActor SetMapper axesMapper

vtkVectorText atext
    atext SetText "Origin"
vtkPolyDataMapper textMapper
    textMapper SetInput [atext GetOutput]
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]
```

As the camera moves around the axes, the follower will orient itself to face the camera. (Try this by mousing in the rendering window to move the camera.)

4.12 Special Plotting Classes

The *Visualization Toolkit* provides several composite classes that perform supplemental plotting operations. These include the ability to plot scalar bars, perform simple x-y plotting, and place flying axes for 3D spatial context.

Scalar Bar

The class vtkScalarBar is used to create a color-coded key that relates color values to numerical data values as shown in **Figure 4–9**. There are three parts to the scalar bar: a rectangular bar with colored segments, labels, and a title. To use vtkScalarBar, you must reference an instance of vtkLookup-Table (defines colors and the range of data values), position and orient the scalar bar on the overlay plane, and optionally specify attributes such as color, number of labels, and text string for the title. The following example shows typical usage.

Figure 4–9 vtkScalarBarActor used to create color legends.

```
vtkScalarBarActor scalarBar
   scalarBar SetLookupTable [mapper
GetLookupTable]
    scalarBar SetTitle "Temperature"
    [scalarBar GetPositionCoordinate] \
                    SetCoordinateSystemToNormalizedViewport
    [scalarBar GetPositionCoordinate] SetValue 0.1 0.01
    scalarBar SetOrientationToHorizontal
    scalarBar SetWidth 0.8
    scalarBar SetHeight 0.17
```

The orientation of the scalar bar is controlled by the methods SetOrientationToVertical() and vtkSetOrientationToHorizontal(). To control the position of the scalar bar (i.e., its lower-left corner), set the position coordinate (in whatever coordinate system you desire—see "vtkCoordinate and Coordinate Systems" on page 70), and then specify the width and height using normalized viewport values (or alternatively, specify the Position2 instance variable to set the upper-right corner).

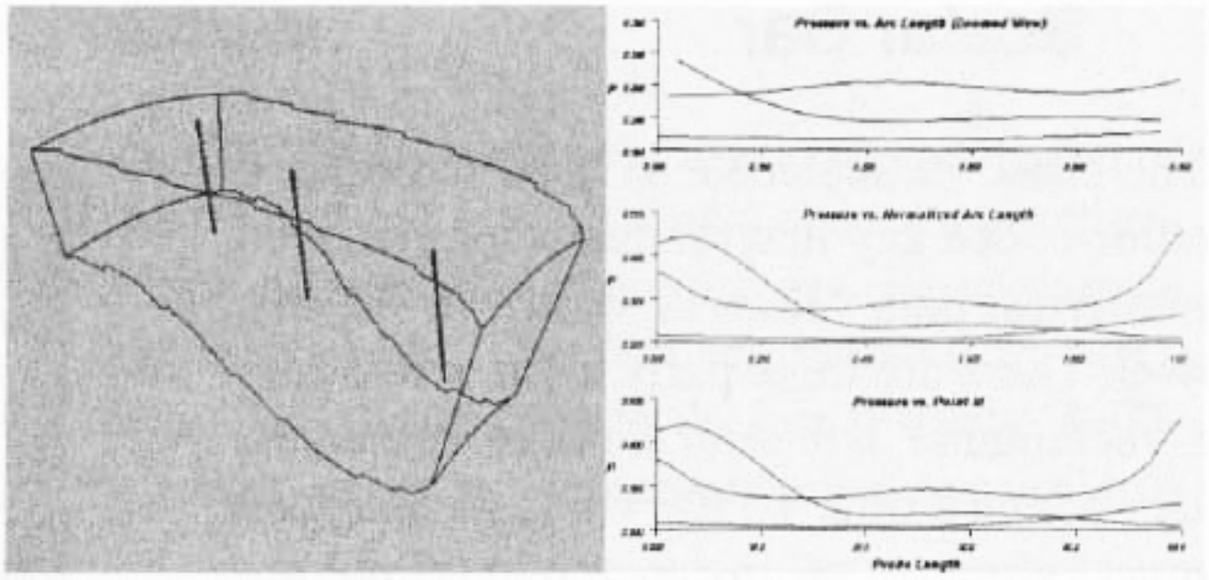

Figure 4–10 Example of using the vtkXYPlotActor2D class to display three probe lines using three different techniques (see Hybrid/Testing/Tcl/xyPlot.tcl).

X-Y Plots

The class vtkXYPlotActor generates *x-y* plots from one or more input datasets, as shown in **Figure 4–10**. This class is particularly useful for showing the variation of data across a sequence of points such as a line probe or a boundary edge.

To use vtkXYPlotActor2D, you must specify one or more input datasets, axes, and the plot title and position the composite actor on the overlay plane. The PositionCoordinate instance variable defines the lower-left location of the x-y plot (specified in normalized viewport coordinates) and the Position2Coordinate instance variable defines the upper-right corner. (Note: The Position2Coordinate is relative to PositionCoordinate, so you can move the vtkXYPlotActor around the viewport by setting just the PositionCoordinate.) The combination of the two position coordinates specifies a rectangle in which the plot will lie. The following example (from `VTK/Examples/Annotation/Tcl/xyPlot.tcl`) shows how the class is used.

```
vtkXYPlotActor xyplot
    xyplot AddInput [probe GetOutput]
    xyplot AddInput [probe2 GetOutput]
    xyplot AddInput [probe3 GetOutput]
    [xyplot GetPositionCoordinate] SetValue 0.0 0.67 0
    [xyplot GetPosition2Coordinate] SetValue 1.0 0.33 0
    xyplot SetXValuesToArcLength
    xyplot SetNumberOfXLabels 6
    xyplot SetTitle "Pressure vs. Arc Length (Zoomed View)"
    xyplot SetXTitle ""
    xyplot SetYTitle "P"
    xyplot SetXRange .1 .35
    xyplot SetYRange .2 .4
    [xyplot GetProperty] SetColor 0 0 0
```

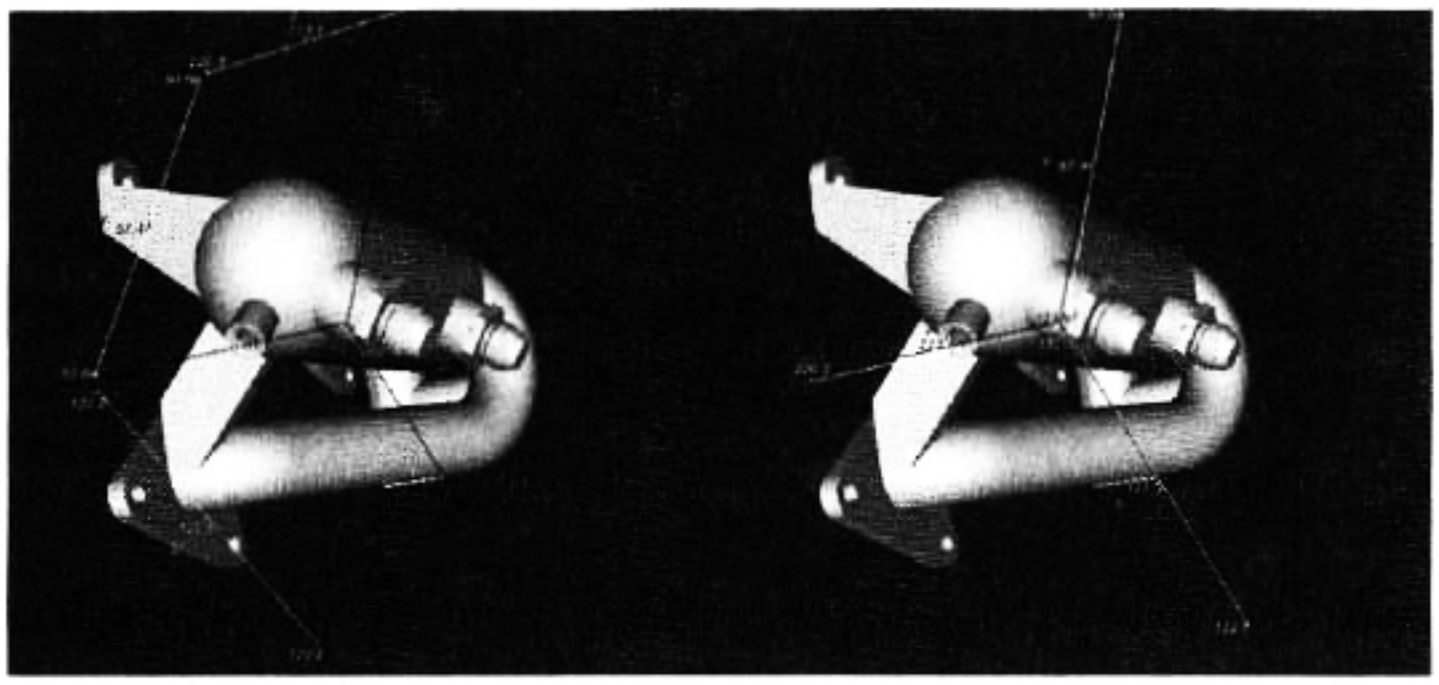

Figure 4–11 Use of vtkCubeAxisActor2D. On the left, outer edges of the cube are used to draw the axes. On the right, the closest vertex to the camera is used.

Note the *x* axis definition. By default, the *x* coordinate is set as the point index in the input datasets. Alternatively, you can use arc length and normalized arc length of lines used as input to vtkXYPlotActor to generate the *x* values.

Bounding Box Axes (vtkCubeAxesActor2D)

Another composite actor class is vtkCubeAxesActor2D. This class can be used to indicate the position in space that the camera is viewing, as shown in **Figure 4–11**. The class draws axes around the bounding box of the input dataset labeled with *x*-*y*-*z* coordinate values. As the camera zooms in, the axes are scaled to fit within the cameras viewport, and the label values are updated. The user can control various font attributes as well as the relative font size (The font size is selected automatically—the method SetFontFactor() can be used to affect the size of the selected font.) The following script demonstrates how to use the class (taken from `VTK/Examples/Annotation/Tcl/cubeAxes.tcl`).

```
vtkCubeAxesActor2D axes
    axes SetInput [normals GetOutput]
    axes SetCamera [ren1 GetActiveCamera]
    axes SetLabelFormat "%6.4g"
    axes ShadowOn
    axes SetFlyModeToOuterEdges
    axes SetFontFactor 0.8
    [axes GetProperty] SetColor 1 1 1
```

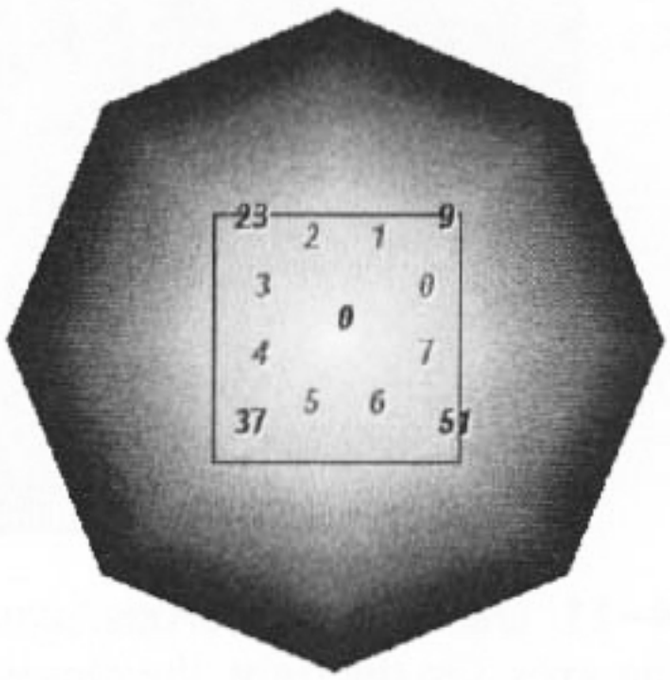

Figure 4–12 Labelling point and cell ids on a sphere within a window.

Note that there are two ways that the axes can be drawn. By default, the outer edges of the bounding box are used (SetFlyModeToOuterEdges()). You can also place the axes at the vertex closest to the camera position (SetFlyModeToClosestTriad())

Labeling Data

In some applications, you may wish to display numerical values from an underlying data set. The class vtkLabeledDataMapper allows you to label the data associated with the points of a dataset. This includes scalars, vectors, tensors, normals, texture coordinates, and field data, as well as the point ids of the dataset. The text labels are placed on the overlay plane of the rendered image as shown in **Figure 4–12**. The figure was generated from the Tcl script `VTK/Examples/Annotation/Tcl/labeledMesh.tcl` and included in part below. The script uses three new classes, vtkCellCenters (to generate points at the parametric centers of cells), vtkIdFilter (to generate ids as scalar or field data from dataset ids), and vtkSelectVisiblePoints (to select those points currently visible), to label the cell and point ids of the sphere. In addition, vtkSelectVisiblePoints has the ability to define a "window" in display (pixel) coordinates in which it operates—all points outside of the window are discarded.

```
# Create a sphere
vtkSphereSource sphere
vtkPolyDataMapper   sphereMapper
    sphereMapper SetInput [sphere GetOutput]
    sphereMapper GlobalImmediateModeRenderingOn
vtkActor sphereActor
    sphereActor SetMapper sphereMapper
```

```
# Generate ids for labeling
vtkIdFilter ids
    ids SetInput [sphere GetOutput]
    ids PointIdsOn
    ids CellIdsOn
    ids FieldDataOn

# Create labels for points
vtkSelectVisiblePoints visPts
    visPts SetInput [ids GetOutput]
    visPts SetRenderer ren1
    visPts SelectionWindowOn
    visPts SetSelection $xmin [expr $xmin + $xLength] \
    $ymin [expr $ymin + $yLength]
vtkLabeledDataMapper ldm
    ldm SetInput [visPts GetOutput]
    ldm SetLabelFormat "%g"
    ldm SetLabelModeToLabelFieldData
vtkActor2D pointLabels
    pointLabels SetMapper ldm

# Create labels for cells
vtkCellCenters cc
    cc SetInput [ids GetOutput]
vtkSelectVisiblePoints visCells
    visCells SetInput [cc GetOutput]
    visCells SetRenderer ren1
    visCells SelectionWindowOn
    visCells SetSelection $xmin [expr $xmin + $xLength] \
    $ymin [expr $ymin + $yLength]
vtkLabeledDataMapper cellMapper
    cellMapper SetInput [visCells GetOutput]
    cellMapper SetLabelFormat "%g"
    cellMapper SetLabelModeToLabelFieldData
vtkActor2D cellLabels
    cellLabels SetMapper cellMapper
    [cellLabels GetProperty] SetColor 0 1 0

# Add the actors to the renderer, set the background and size
ren1 AddActor sphereActor
ren1 AddActor2D pointLabels
ren1 AddActor2D cellLabels
```

4.13 Transforming Data

As we saw in the section "Specifying the Position of a vtkProp3D" on page 56, it is possible to position and orient vtkProp3D's in world space. However, in many applications we wish to transform the data prior to using it in the visualization pipeline. For example, to use a plane to cut ("Cutting" on page 96) or clip ("Clip Data" on page 111) an object, the plane must be positioned within the pipeline, not via the actor transformation matrix. Some objects (especially procedural source objects) can be created at a specific position and orientation in space. For example, vtkSphereSource has a Center and Radius instance variable, and vtkPlaneSource has Origin, Point1, and Point2 instance variables that allow you to position the plane using three points. However, many classes do not provide this capability, without moving data into a new position. In this case, you must transform the data using vtkTransformFilter or vtkTransformPolyDataFilter.

vtkTransformFilter is a filter that takes vtkPointSet dataset objects as input. Datasets that are subclasses of the abstract class vtkPointSet represent points explicitly, that is, an instance of vtkPoints is used to store coordinate information. vtkTransformFilter applies a transformation matrix to the points and create a transformed points array; the rest of the dataset structure (i.e., cell topology) and attribute data (e.g., scalars, vectors, etc.) remains unchanged. vtkTransformPolyDataFilter does the same thing as vtkTransformFilter except that it is more convenient to use in a visualization pipeline containing polygonal data.

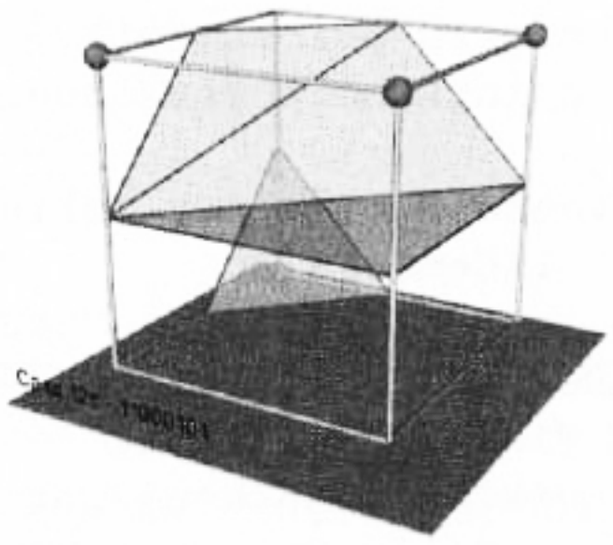

Figure 4–13 Transforming data within the pipeline.

The following example (taken from `VTK/Examples/DataManipulation/Tcl/marching.tcl` with results shown in **Figure 4–13**) uses a vtkTransformPolyDataFilter to reposition a 3D text string (See "3D Annotation and vtkFollower" on page 74) for more information about 3D text)

```
#define the text for the labels
vtkVectorText caseLabel
  caseLabel SetText "Case 12c - 11000101"
vtkTransform aLabelTransform
  aLabelTransform Identity
  aLabelTransform Translate  -.2 0 1.25
  aLabelTransform Scale .05 .05 .05
vtkTransformPolyDataFilter labelTransform
```

```
  labelTransform SetTransform aLabelTransform
  labelTransform SetInput [caseLabel GetOutput]
vtkPolyDataMapper labelMapper
  labelMapper SetInput [labelTransform GetOutput];
vtkActor labelActor
  labelActor SetMapper labelMapper
```

Notice that vtkTransformPolyDataFilter requires that you supply it with an instance of vtkTransform. Recall that vtkTransform is used by actors to control their position and orientation in space. Instances of vtkTransform supports many methods, some of the most commonly used are shown here.

- `RotateX(angle)` — apply rotation (angle in degrees) around the *x* axis
- `RotateY(angle)` — apply rotation around the *y* axis
- `RotateZ(angle)` — apply rotation around the *z* axis
- `RotateWXYZ(angle,x,y,z)` — apply rotation around a vector defined by *x*-*y*-*z* components
- `Scale(x,y,z)` — apply scale in the *x*, *y*, and *z* directions
- `Translate(x,y,z)` — apply translation
- `Inverse()` — invert the transformation matrix
- `SetMatrix(m)` — specify the 4x4 transformation matrix directly
- `GetMatrix(m)` — get the 4x4 transformation matrix
- `PostMultiply()` — control the order of multiplication of transformation matrices. If PostMultiply() is invoked, matrix operations are applied on the right hand side of the current matrix.
- `PreMultiply()` — matrix multiplications are applied on the left hand side of the current transformation matrix

The last two methods described above remind us that the order in which transformations are applied dramatically affects the resulting transformation matrix (see "Specifying the Position of a vtkProp3D" on page 56). We recommend that you spend some time experimenting with these methods, and the order of application, to fully understand vtkTransform.

Advanced Transformation

Advanced users may wish to use VTK's extensive transformation hierarchy. (Much of this work was done by David Gobbi.) The hierarchy, of which the class hierarchy is shown in **Figure 14–13**, supports a variety of linear and non-linear transformations.

A wonderful feature of the VTK transformation hierarchy is that different types of transformation can be used in a filter to give very different results. For example, the vtkTransformPolyDataFilter accepts any transform of type vtkAbstractTransform (or a subclass). This includes transformation types ranging from the linear, affine vtkTransform (represented by a 4x4 matrix) to the non-linear, warp vtkThinPlateSplineTransform, which is a complex function representing a correlation between a set of source and target landmarks.