Chapter 3

System Overview

The purpose of this chapter is to provide you with an overview of the *Visualization Toolkit* system, and to show you the basic information you'll need to create applications in C++, Java, Tcl, and Python. We begin by introducing basic system concepts and object model abstractions. We close the chapter by demonstrating these concepts, and describing what you'll need to know to build applications.

3.1 System Architecture

The *Visualization Toolkit* consists of two basic subsystems: a compiled C++ class library, and an "interpreted" wrapper layer that lets you manipulate the compiled classes using the languages Java, Tcl, and Python. See **Figure 3–1**.

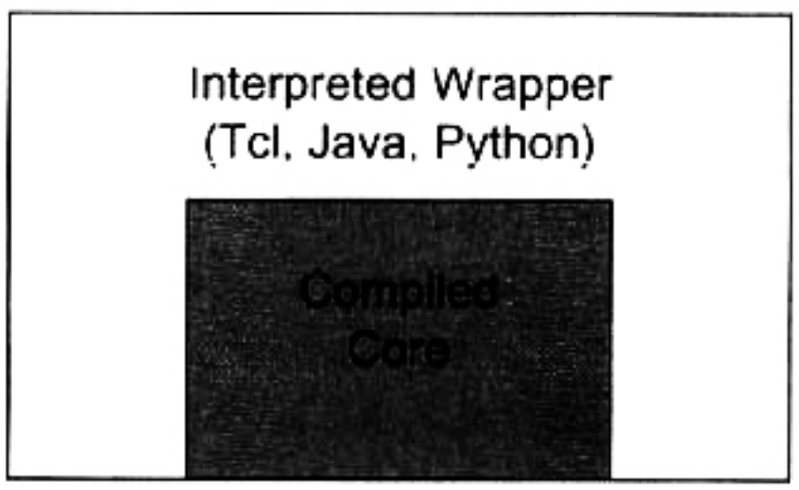


Figure 3–1 The Visualization Toolkit consists of a compiled (C++) core wrapped with various interpreted languages (Java, Tcl, Python).

The advantage of this architecture is that you can build efficient (in both CPU and memory) algorithms in the compiled language C++, and retain the rapid code development features of interpreted languages (avoidance of compile/link cycle,
simple but powerful tools, and access to GUI tools). Of course, for those proficient in C++ and who have the tools to do so, applications can be built entirely in C++.

The *Visualization Toolkit* is an object-oriented system. The key to using VTK effectively is to develop a good understanding of the underlying object models. Doing so will remove much of the mystery surrounding the use of the hundreds of objects in the system. With this understanding in place it's much easier to combine objects to build applications. You'll also need to know something about the capabilities of the many objects in the system; this only comes with reviewing code examples and man pages. In this User's Guide, we've tried to provide you with useful combinations of VTK objects that you can adapt to your own applications.

In the remainder of this section, we will review the two object models that make up the *Visualization Toolkit*: the graphics model and the visualization model. These sections are fairly high-level—we recommend that you augment this reading with a review of the examples, either those in this chapter, in the next chapter, or by executing the hundreds of examples available in the VTK source distribution.

The Graphics Model

The VTK graphics model consists of the following core objects. (Note: this is not an exhaustive list of all objects, just the ones you'll use most often.)

- vtkActor, vtkActor2D, vtkVolume—subclasses of vtkProp and/or vtkProp3D.
- vtkLight
- vtkCamera
- vtkProperty, vtkProperty2D
- vtkMapper, vtkMapper2D—subclasses of vtkAbstractMapper
- vtkTransform
- vtkLookupTable, vtkColorTransferFunction—subclasses of vtkScalarsToColors
- vtkRenderer
- vtkRenderWindow
- vtkRenderWindowInteractor

When we combine these objects together we create a scene. (Refer to the object diagram of **Figure 14–8** to see how these objects are related.)

Props represent the things that we "see" in the scene. Props that are positioned and manipulated in 3D (i.e., have a general 4x4 transformation matrix) are of type vtkProp3D (for example, vtkActor is a concrete subclass of vtkProp3D; or if we are volume rendering, vtkVolume). Props that are positioned and represent 2D data (i.e., image data) are of type vtkActor2D. Props don't directly represent their geometry; instead they refer to mappers, which are responsible for representing data (among other things). Props also refer to a property object. The property object controls the appearance of the prop (e.g., color, diffuse, ambient specular lighting effects, rendering representation: wireframe versus surface, and so on). Actors and volumes (via their superclass vtkProp3D) also have an

internal transformation object (vtkTransform). This object encapsulates a 4x4 transformation matrix that in turn controls the prop's position, orientation, and scale.

Lights (vtkLight) are used to represent and manipulate the lighting of the scene. Lights are used only in 3D. In 2D, we do not need lights.

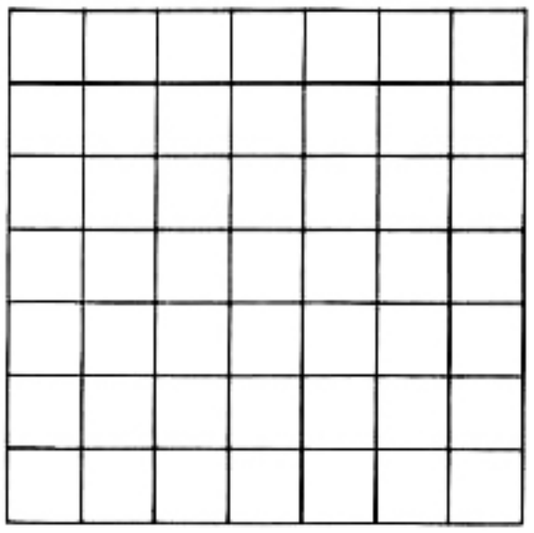
The camera object (vtkCamera) controls how 3D geometry is projected into the 2D image during the rendering process. The camera has several methods for positioning, pointing, and orienting it. In addition, the camera controls perspective projection and stereo viewing (if enabled). Cameras are not needed for 2D imaging data.

The mapper (vtkMapper), in conjunction with a lookup table, (vtkLookupTable), is used to transform and render geometry. The mapper provides the interface between the visualization pipeline (described in the next section) and the graphics model. vtkLookupTable is a subclass of vtkScalarsToColors, as is vtkColorTransferFunction. (Typically vtkColorTransferFunction is used for volume rendering—see "Volume Rendering" on page 136.) Subclasses of vtkScalarsToColors are responsible for mapping data values to color, one of the most important visualization techniques.

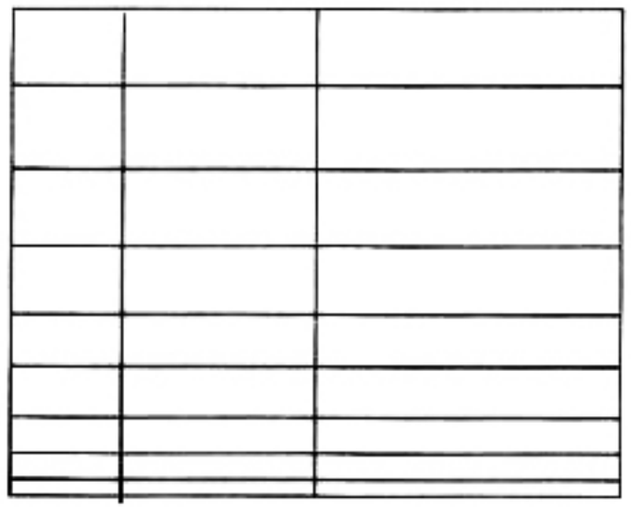
Renderers (vtkRenderer) and render windows (vtkRenderWindow) are used to manage the interface between the graphics engine and you computer's windowing system. The render window is the window on your computer that the renderer draws into. More than one renderer may draw into a single render window; and you may create multiple render windows. The region that a renderer draws into is called the viewport, of which several may exist in a rendering window.

Once you draw objects into the render window, chances are that you will interact with the data. The *Visualization Toolkit* has several methods of interaction into the scene. One of them is the object vtkRenderWindowInteractor, which is a simple tool for manipulating the camera, picking objects, invoking user-defined methods, entering/exiting stereo viewing, and changing some of the properties of actors.

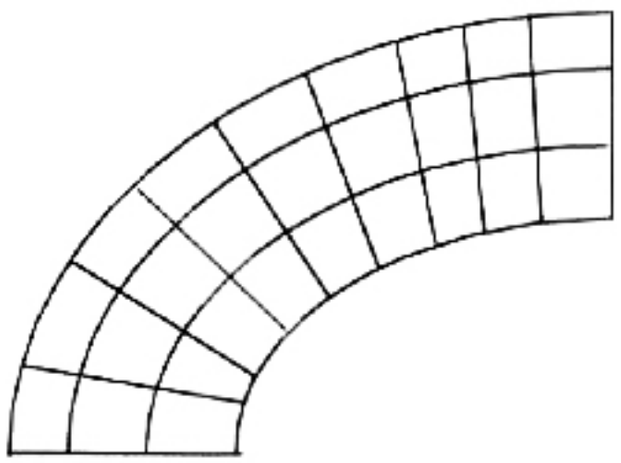
Many of the objects described above have subclasses that specialize the object's behavior. For example, vtkAssembly, vtkFollower, and vtkLODActor are all subclasses of vtkActor. vtkAssembly allows hierarchies of actors, properly managing the transformations when the hierarchy is translated, rotated, or scaled. vtkFollower is an actor that always faces a specified camera (useful for billboards or text). vtkLODActor (LOD means level-of-detail) is an actor that changes its geometric representation to maintain interactive frame rates.



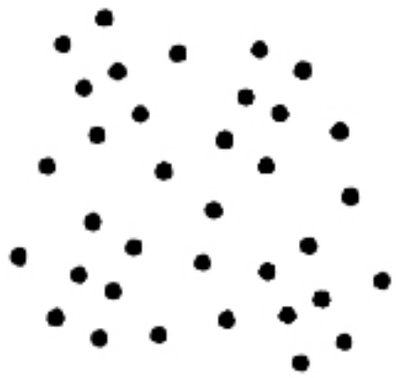
(a) Image Data
(vtkImageData)



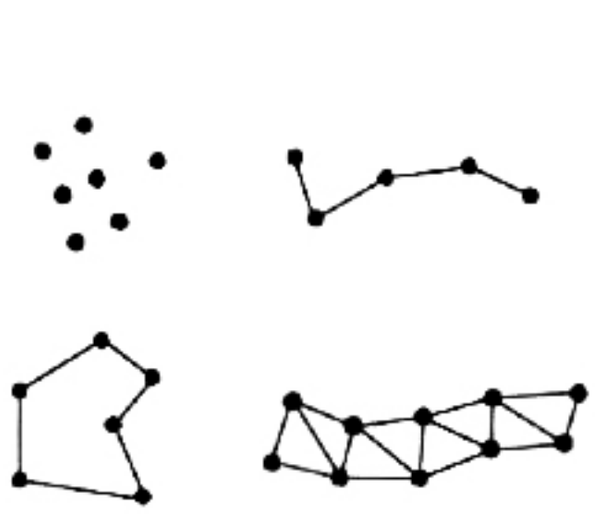
(b) Rectilinear Grid
(vtkRectilinearGrid)



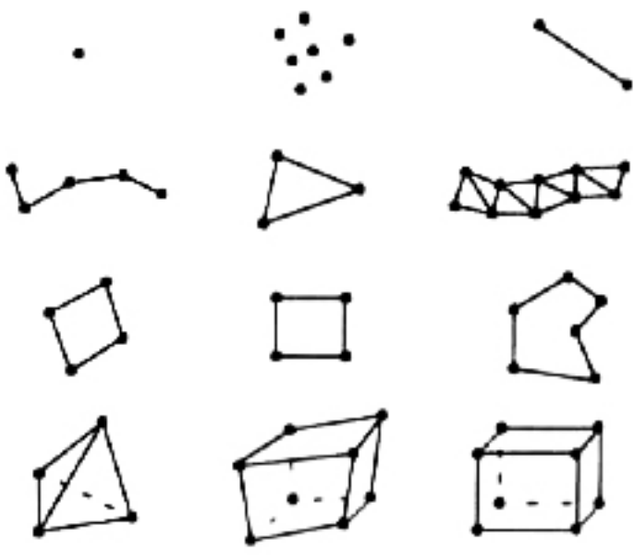
(c) Structured Grid
(vtkStructuredGrid)



(d) Unstructured Points
(use vtkPolyData)



(e) Polygonal Data
(vtkPolyData)



(f) Unstructured Grid
(vtkUnstructuredGrid)

Figure 3–2 Dataset types found in **VTK**. Note that unstructured points can be represented by either polygonal data or unstructured grids, so are not explicitly represented in the system.

The Visualization Model

The role of the graphics pipeline is to transform graphical data into pictures. The role of the visualization pipeline is to transform information into graphical data. Another way of looking at this is that the visualization pipeline is responsible for constructing the geometric representation that is then rendered by the graphics pipeline.

The *Visualization Toolkit* uses a data flow approach to transform information into graphical data. There are two basic types of objects involved in this approach.

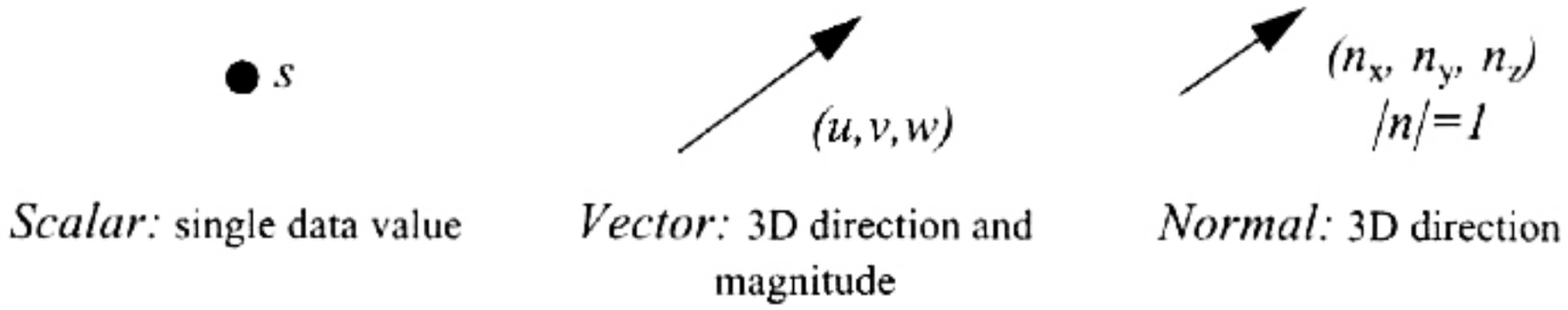


Figure 3–3 Data attributes associated with the points and cells of a dataset.

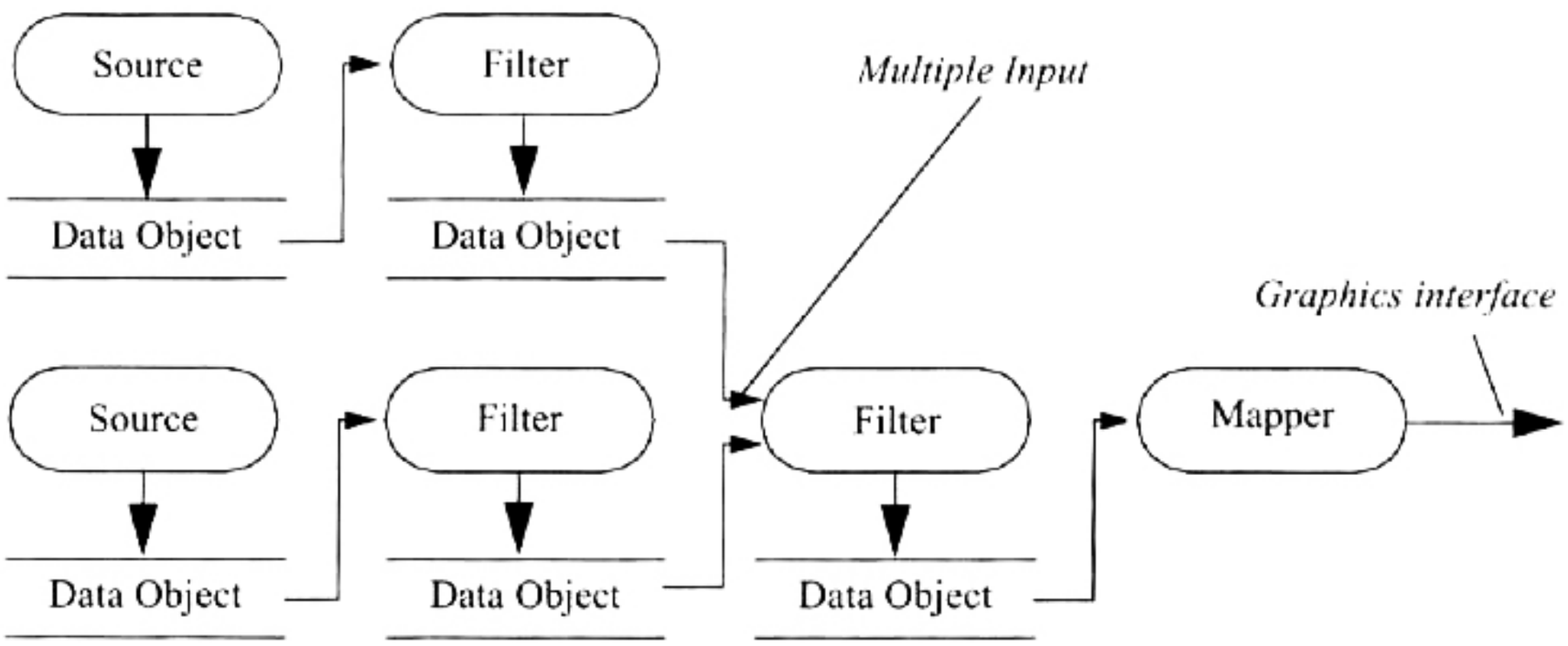


Figure 3–4 Data objects are connected with process objects to create the visualization pipeline. The arrows point in the direction of data flow.

- vtkDataObject
- vtkProcessObject

Data objects represent data of various types. The class vtkDataObject can be viewed as a generic "blob" of data. Data that has a formal structure is referred to as a dataset (class vtkDataSet). **Figure 3–2** shows the dataset objects supported in VTK. Data objects consist of a geometric and topological structure (points and cells) as illustrated by the figure, as well as associated attribute data such as scalars or vectors. The attribute data can be associated with the points or cells of the dataset. Cells are topological organizations of points; cells form the atoms of the dataset and are used to interpolate information between points **Figure 14–15** shows the various cell types supported by VTK. **Figure 3–3** shows the attribute data supported by VTK.

Process objects, also referred to generically as filters, operate on data objects to produce new data objects. Process objects represent the algorithms of the system. Process and data objects are connected together to form visualization pipelines (i.e., data-flow networks). **Figure 3–4** is a depiction of a visualization pipeline.

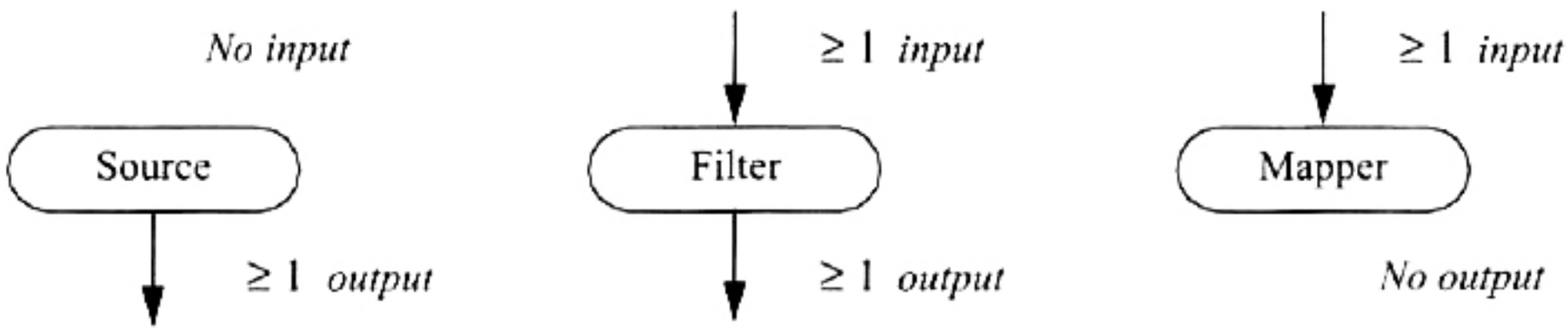
This figure together with **Figure 3–5** illustrate some important visualization concepts. Source process objects are objects that produce data by reading (reader objects) or constructing one or more data objects (procedural source objects). Filters ingest one or more

data objects, and generate one or more data objects on output. Mappers, which we have seen earlier in the graphics model, transform data objects into graphics data, which is then rendered by the graphics engine. Writers are a type of mapper that write data to a file or stream.

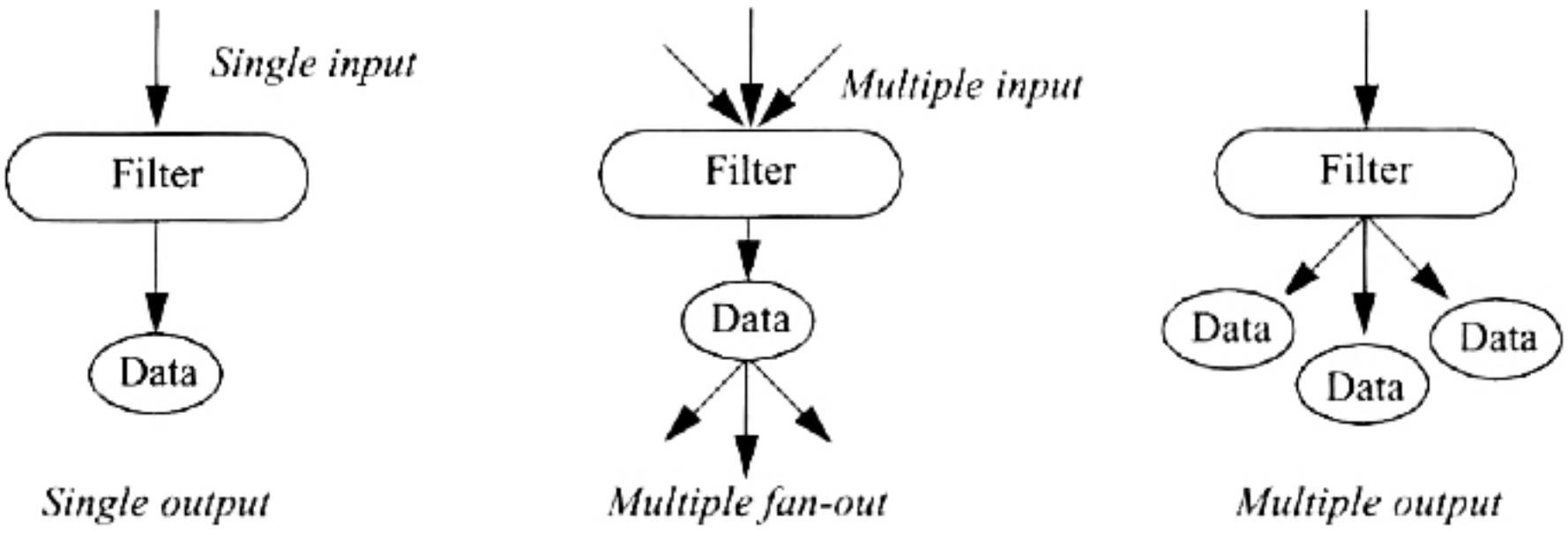
There are several important issues regarding the construction of the visualization pipeline that we will briefly introduce here. First, pipeline topology is constructed using variations of the methods

```
aFilter->SetInput( anotherFilter->GetOutput() );
```

which sets the input to the filter `aFilter` to the output of the filter `anotherFilter`. (Filters with multiple input and output have similar methods for setting input and output.) Second, we must have a mechanism for controlling the execution of the pipeline. We only want to execute those portions of the pipeline necessary to bring the output up to date. The *Visualization Toolkit* uses a lazy evaluation scheme (executes only when the data is



(a) Sources, filters, and mappers



(b) Multiplicity of input and output

Figure 3–5 Different types of process objects. Filters ingest one or more inputs and produce one or more output data objects.

requested) based on an internal modification time of each object. Third, the assembly of the pipeline requires that only those objects compatible with one another can fit together with the `SetInput()` and `GetOutput()` methods. In VTK, C++ compile-time type checking enforces this (interpreters such as Tcl report errors at run-time). Finally, we must decide whether to cache, or retain, the data objects once the pipeline has executed. Since visualization datasets are typically quite large, this is important to the successful application of visualization tools. VTK offers methods to turn data caching on and off, use of reference counting to avoid copying data, and methods to stream data in pieces if an entire dataset cannot be held in memory. (We recommend that you review Chapter 4 of The Visualization Toolkit text for more information.)

Please note that there are many varieties of both process and data objects. **Figure 11–2** shows six different data object types supported by the current version of VTK. Process objects vary in the type(s) of input data and output data, and of course the particular algorithm they implement.

Pipeline Execution. In the previous section we discussed the need to control the execution of the visualization pipeline. In this section we will expand our understanding of some key concepts regarding pipeline execution.

As indicated in the previous section, the VTK visualization pipelines only executes when data is required for computation (lazy evaluation). For example, if you instantiate a reader object and ask for the number of points (the language shown here is Tcl):

```
vtkPLOT3DReader reader
    reader SetXYZFileName "$VTK_DATA_ROOT/Data/combxyz.bin
    [reader GetOutput] GetNumberOfPoints
```

the `reader` object will return "0" from the GetNumberOfPoints() method call, despite the fact that the data file contains thousands of points. However, if you add the Update() method

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

then the `reader` object will return the correct number. The reason for this is that—in the first example—the GetNumberOfPoints() methods does not require computation, and the object simply returns the current number of points, which is "0". In the second example, the Update() method forces execution of the pipeline, thereby forcing the reader to execute and read the data from the file indicated.

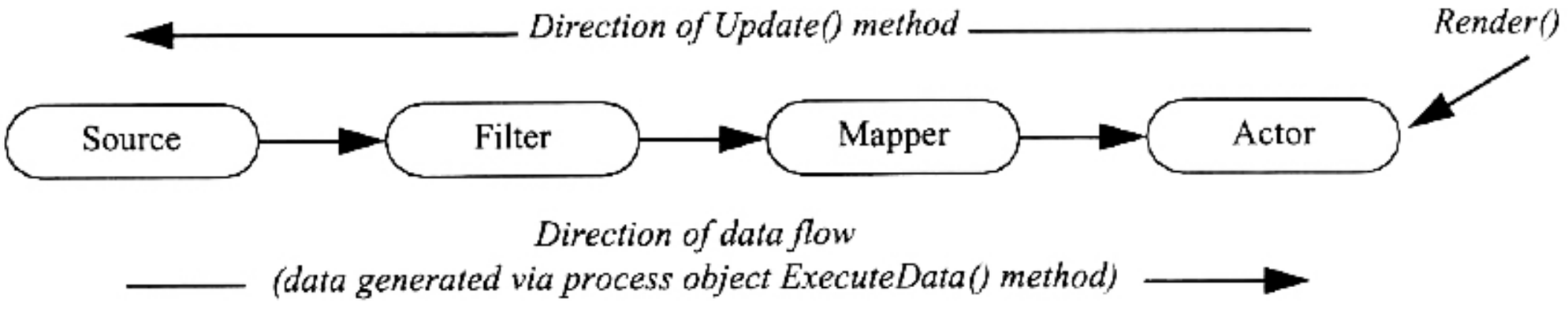


Figure 3–6 Conceptual overview of pipeline execution.

Normally, you do not need to manually invoke Update() because the filters are connected into a visualization pipeline. In this case, when the actor receives a request to render itself, it forwards the method to its mapper, and the Update() method is automatically sent through the visualization pipeline. From a high level, the execution of the pipeline appears shown in **Figure 3–6**. As this figure illustrates, the Render() method often initiates the request for data, which is then delivered down through the pipeline. Depending on which portions of the pipeline are out of date, the filters in the pipeline may re-execute, thereby bringing the data at the end of the pipeline up-to-date, which is then rendered by the actor. (For more information about the execution process, see Chapter 10 "Managing Pipeline Execution" on page 219.)

Image Processing. VTK supports an extensive set of image processing and volume rendering functionality. In VTK, both 2D (image) and 3D (volume) data are referred to as vtkImageData. An image dataset in VTK is one in which the data is arranged in a regular, axis-aligned array. Images, pixmaps, and bitmaps are examples of 2D image datasets; volumes (a stack of 2D images) is a 3D image dataset.

Process objects in the imaging pipeline always input and output image data objects. Because of the regular and simple nature of the data, the imaging pipeline has other important features. Volume rendering is used to visualize 3D vtkImageData (see "Volume Rendering" on page 136), and special image viewers are used to view 2D vtkImageData. Almost all process objects in the imaging pipeline are multithreaded, and are capable of streaming data in pieces (to satisfy a user-specified memory limit). Filters automatically sense the number of processors available on the system and create that number of threads during execution, as well as automatically separating data into pieces that are streamed through the pipeline. (See "Using Streaming" on page 232 for more information.)

This concludes our brief overview of the *Visualization Toolkit* system architecture. We recommend the *Visualization Toolkit* text for more details on many of the algorithms found in VTK. Learning by example is another helpful approach. Chapters 4 through 8 contain many annotated examples demonstrating various capabilities of VTK. Also, since source code is available, and there are hundreds of examples, you may wish to study these as well.

With this abbreviated introduction behind us, let's look at ways to create applications in C++, Tcl, Java, and Python.

3.2 Create An Application

This section covers the basic information you need to develop VTK applications in the four programming languages Tcl, C++, Java, and Python. (Kitware offers commercial products to build VTK applications with Visual Basic or ActiveX/COM. (See "Visual Basic / COM / ActiveX" on page 38) After reading this introduction, you should jump to the subsection(s) that discuss the language(s) you are interested in using. In addition to providing you with instructions on how to create and run a simple application, each section will show you how to take advantage of callbacks in that language.

User Methods, Observers, and Commands

Callbacks (or *user methods*) are implemented in VTK using the Subject/Observer and Command design pattern. This means that every class in VTK (every subclass of vtkObject) has an AddObserver() method that can be used to setup callbacks from VTK. The observer looks at every event invoked on an object, and if it matches one of the events that the observer is watching for, then an associated command is invoked (i.e., the callback). For example, all VTK filters invoke a `StartEvent` right before they start to execute. If you add an observer that watches for a `StartEvent` then it will get called every time that filter starts to execute. Consider the following Tcl script that creates an instance of vtkElevationFilter, and adds an observer for the `StartEvent` to call the procedure `PrintStatus`.

```
proc PrintStatus {} {
puts "Starting to execute the elevation filter"
}
vtkElevationFilter foo
foo AddObserver StartEvent PrintStatus
```

This type of functionality (i.e., callback) is available to all the languages VTK supports. Each section that follows will show a brief example of how to use it. Further discussion on user methods is provided in "Integrating With The Windowing System" on page 303. (This section also discusses user interface integration issues.)

To create your own application, we suggest starting with one of the examples that come with VTK. They can be found in `VTK/Examples` in the source distribution, and in subdirectories with the PC executable distribution. In the source distribution the examples are organized first by topic and then by language. Under `VTK/Examples` you will find directories for different topics and under the directories there will be subdirectories for different languages such as Tcl.

Tcl

Tcl is one of the easiest languages with which to start creating VTK applications. Once you have installed VTK, you should be able to run the Tcl examples that come with the distribution. Under UNIX you have to compile VTK with Tcl support as mentioned in "Installing VTK On Unix Systems" on page 14. Under Windows you can just install the self extracting archive as described in "Installing VTK on Windows 9x/NT/ME/2000/XP" on page 8, and you are ready to go.

Windows. Under Windows, you can run a Tcl script just by double clicking on the file (`test1.tcl` in this example). If nothing happens you might have an error in your script. To detect this you need to run `vtk.exe` first. `vtk.exe` can be found on your start menu under VTK. Once execution begins, a console window should appear with a prompt in it. At this prompt type in a cd command to change directory to where `test1.tcl` is located. Two examples are given below:

```
% cd "c:/Program Files/Visualization Toolkit/examples"

% cd "c:/VTK/Examples/Tutorial/Tcl"
```

Then you will need to source in the example script using the following command:

```
% source Cone.tcl
```

Then Tcl will try to execute `test1.tcl` and you will be able to see additional errors or warning messages that would otherwise not show up.

Unix. Under UNIX, Tcl development can be done by running the VTK executable (after you have compiled the source code) that can be found in your binary directory such as `VTK/bin/vtk`, or `VTK-Solaris/bin/vtk`, etc., and then providing the Tcl script as the first argument. For example:

```
unix machine> cd VTK/Examples/Tutorial/Step1/Tcl
unix machine> /home/VTK-Solaris/bin/vtk Cone.tcl
```

User methods can be setup as shown in the introduction of this chapter. An example can be found in `Examples/Tutorial/Step2/Tcl`. The key changes are shown below:

```
proc myCallback {} {
    puts "Starting to render"
}

vtkRenderer ren1
ren1 AddObserver StartEvent myCallback
```

or by simply providing the body of the proc directly to AddObserver() as shown below:

```
vtkRenderer ren1
ren1 AddObserver StartEvent {puts "Starting to render"}
```

C++

Using C++ as your development language will typically result in smaller, faster, and more easily deployed applications than most any other language. C++ development also has the advantage that you do not need to compile any additional support for Tcl, Java, or Python. This section will show you how to create a simple VTK C++ application for the PC with Microsoft Visual C++, and also for UNIX using the appropriate compiler. We will start with a simple example called `Cone.cxx` which can be found in `Examples/Tutorial/Step1/Cxx`. For both windows and UNIX you can use a source code installation of VTK or installed binaries. These examples will work with both.

The first step in building your C++ program is to use CMake to generate a Makefile (Unix) or Microsoft Workspace (Windows MSVC++). The `CMakeList.txt` file that comes with `Cone.cxx` (shown below) makes use of the FindVTK and UseVTK CMake modules. These modules attempt to locate VTK and then setup your include paths and link

lines for building C++ programs. If they do not successfully find VTK, you will have to manually specify the appropriate CMake parameters, and rerun CMake as necessary.

```
PROJECT (Step1)

INCLUDE (${CMAKE_ROOT}/Modules/FindVTK.cmake)
IF (USE_VTK_FILE)
  INCLUDE(${USE_VTK_FILE})
ENDIF (USE_VTK_FILE)

LINK_LIBRARIES(
  vtkRendering
  vtkGraphics
  vtkImaging
  vtkFiltering
  vtkCommon
)

ADD_EXECUTABLE(Cone Cone.cxx)
```

Microsoft Visual C++. Once you have run CMake for the Cone example you are ready to start up Microsoft Visual C++ and load the `Cone.dsw` workspace. You can now select a build type (such as Release or Debug) and build your application. If you want to integrate VTK into an existing project that does not use CMake, you can copy the settings from this simple example into your existing workspaces.

Now consider an example of a true Windows application. The process is very similar to what we did above, except that we create a windows application instead of a console application, as shown in the following. Much of the code is standard Windows code and will be familiar to any Windows developer. This example can be found in `Examples/GUI/Win32/SimpleCxx`. Notice that the only significant change to the `CMakeLists.txt` file was the addition of the WIN32 parameter in the `ADD_EXECUTABLE` command.

```
#include "vtkConeSource.h"
#include "vtkPolyDataMapper.h"
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"

static HANDLE hinst;
long FAR PASCAL WndProc(HWND, UINT, UINT, LONG);

// define the vtk part as a simple c++ class
```

```
class myVTKApp
{
public:
  myVTKApp(HWND parent);
  ~myVTKApp();
private:
  vtkRenderWindow *renWin;
  vtkRenderer *renderer;
  vtkRenderWindowInteractor *iren;
  vtkConeSource *cone;
  vtkPolyDataMapper *coneMapper;
  vtkActor *coneActor;
};
```

We start by including the required VTK include files. We do not need to include the standard windows header files because the VTK header files include them. Next we have two standard windows prototypes followed by a small class definition called myVTKApp. When developing in C++, you should try to use object-oriented approaches instead of the scripting programming style found in many of the Tcl examples. Here we are encapsulating the VTK components of the application into a small class.

```
myVTKApp::myVTKApp(HWND hwnd)
{
  // Similar to Examples/Tutorial/Step1/Cxx/Cone.cxx
  // We create the basic parts of a pipeline and connect them
  this->renderer = vtkRenderer::New();
  this->renWin = vtkRenderWindow::New();
  this->renWin->AddRenderer(this->renderer);

  // setup the parent window
  this->renWin->SetParentId(hwnd);
  this->iren = vtkRenderWindowInteractor::New();
  this->iren->SetRenderWindow(this->renWin);

  this->cone = vtkConeSource::New();
  this->cone->SetHeight( 3.0 );
  this->cone->SetRadius( 1.0 );
  this->cone->SetResolution( 10 );
  this->coneMapper = vtkPolyDataMapper::New();
  this->coneMapper->SetInput(this->cone->GetOutput());
  this->coneActor = vtkActor::New();
  this->coneActor->SetMapper(this->coneMapper);

  this->renderer->AddActor(this->coneActor);
```

```
  this->renderer->SetBackground(0.2,0.4,0.3);
  this->renWin->SetSize(400,400);

  // Finally we start the interactor so that event will be handled
  this->renWin->Render();
}
```

This is the constructor for myVTKApp. As you can see it allocates the required VTK objects, sets their instance variables, and then connects them to form a visualization pipeline. Most of this is straightforward VTK code except for the vtkRenderWindow. This constructor takes a HWND handle to the parent window that should contain the VTK rendering window. We then use this in the SetParentId() method of vtkRenderWindow so that it will create its window as a child of the window passed to the constructor.

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

The destructor simply frees all of the VTK objects that were allocated in the constructor.

```
int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
 LPSTR lpszCmdParam, int nCmdShow)
{
  static char szAppName[] = "Win32Cone";
  HWND hwnd ;
  MSG msg ;
  WNDCLASS wndclass ;

  if (!hPrevInstance)
    {
    wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra = 0 ;
    wndclass.cbWndExtra = 0 ;
    wndclass.hInstance = hInstance;
    wndclass.hIcon = LoadIcon(NULL,IDI_APPLICATION);
    wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
```

```
    wndclass.lpszMenuName = NULL;
    wndclass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
    wndclass.lpszClassName = szAppName;
    RegisterClass (&wndclass);
    }
  hinst = hInstance;

  hwnd = CreateWindow ( szAppName,
                        "Draw Window",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        400,
                        480,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);
  ShowWindow (hwnd, nCmdShow);
  UpdateWindow (hwnd);
  while (GetMessage (&msg, NULL, 0, 0))
    {
    TranslateMessage (&msg);
    DispatchMessage (&msg);
  }
  return msg.wParam;
}
```

The WinMain code here is all standard windows code and has no VTK references in it. As you can see the application has control of the event loop. Events are handled by the WndProc described below.

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

```
                                (HINSTANCE)GetWindowLong(hwnd,GWL_HINSTANCE),
                                NULL);
    theVTKApp = new myVTKApp(hwnd);
    return 0;
    }
  case WM_COMMAND:
    switch (wParam)
      {
      case 2:
        PostQuitMessage (0);
        if (theVTKApp)
          {
          delete theVTKApp;
          theVTKApp = NULL;
          }
        break;
      }
    return 0;

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

This WndProc is a very simple event handler. For a full application it would be significantly more complicated but the key integration issues are the same. At the top of this function we declare a static reference to a myVTKApp instance. When handling the `WM_CREATE` method we create an Exit button and then construct and instance of myVTKApp passing in the handle to the current window. The vtkRenderWindowInteractor will handle all of the events for the vtkRenderWindow so that you do not need to handle them here. You probably will want to add code to handle resizing events so that the render window resizes appropriate to your overall user interface. If you do not set the `ParentId` of the vtkRenderWindow, it will show up as a top-level independent window. Everything else should behave the same as before.

UNIX. Creating a C++ application on UNIX is done by running CMake and then make. CMake creates a Makefile that specified the include paths, link lines, and dependencies. make then uses this makefile to compile the application. This should result in a Cone executable that you can run. If `Cone.cxx` does not compile then check the make errors and correct them. Make sure that the values in the top of `CMakeCache.txt` are valid. If it does compile, but you receive errors when you try running it, you might need to set your `LD_LIBRARY_PATH` as described in Chapter 2.

User Methods in C++. You can add user methods (using the observer/command design pattern) in C++ by creating a subclass of vtkCommand that overrides the Execute() method. Consider the following example taken from `Examples/Tutorial/Step2/Cxx`:

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

While the Execute() method is always passed the calling object (caller) you do not need to use it. If you do use the caller you will typically want to perform a SafeDownCast() to the actual type. For example:

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

Once you have created your subclass of vtkCommand you are ready to add an observer that will call your command on certain events. This can be done as follows:

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

The above code creates an instance of myCallback and then adds an observer on `ren1` for the `StartEvent`. Whenever ren1 starts to render the Execute() method of myCallback will get called. When ren1 is deleted then the callback will be deleted as well.

Java

To create Java applications you must first have a working Java development environment. This section provides instructions for using Sun's JDK 1.3 or later on either Windows or UNIX. Once your JDK has been installed and you have installed VTK, you need to set your `CLASSPATH` environment variable to include the VTK classes. Under Microsoft Windows this can be set in your `autoexec.bat` file using the sysedit program or on Windows 2000 by right clicking on the My Computer icon, selecting the properties option, and then selecting the Advanced tab. Then add a `CLASSPATH` environment variable and set it to include your VTK/java directory. For a Windows executable installation this will be something similar to `C:\Program Files\Visualization Toolkit\java\vtk.jar`. For a Windows build it will be something like `C:\vtkbin\java`. Under UNIX you should set your `CLASSPATH` environment variable to something similar to `/yourdisk/vtk/java` or `/yourdisk/vtk-solaris/java`.

The next step is to byte compile your Java program. For starters try byte compiling (with javac) the `Cone.java` example that comes with VTK under `Examples/Tutorial/Step1/Java`. Then you should be able to run the resulting application using the java command. It should display a cone which rotates 360 degrees and then exits. The next step is to create your own applications using the examples provided as a starting point.

One note of caution: Java is undergoing some growing pains and you will undoubtedly run into some problems. For example, running VTK applets inside of a web browser can be a difficult process. We recommend that if you use VTK, you plan your development for running within the AppletViewer or as a standard Java program. Callbacks in Java are essentially public void class methods such as:

```
public void myCallback()
{
  System.out.println("Starting a render");
}
```

You setup a callback by passing three arguments. The first is the name of the event you are interested in, the second is an instance of a class, the third is the name of the method you want to invoke. In this example we setup the `StartEvent` to invoke the myCallback method on me (which is an instance of Cone2). The myCallback method must of course be a valid method of Cone2 to avoid an error.

```
Cone2 me = new Cone2();
ren1.AddObserver("StartEvent",me,"myCallback");
```

Python

Before running Python you will need to set up your `PYTHONPATH` environment variable. This environment variable is used by Python to find additional modules and libraries. This will typically be something like `VTK/Wrapping/Python` or `vtkbin/bin`. Once this is set you should be able to run `Examples/Tutorial/Step1/Python/Cone.py` as follows

```
python Cone.py
```

Creating your own Python scripts is a simple matter of using some of our example scripts as a starting point. User methods can be setup by defining a function and then passing it as the argument to the AddObserver as shown below.

```
def myCallback(obj,event):
print "Starting to render"
ren1.AddObserver("StartEvent",myCallback)
```

You can look at `Examples/Tutorial/Step2/Python/Cone2.py` for the source code shown above.

Visual Basic / COM / ActiveX

If you are interested in creating applications in Microsoft Visual Basic or ActiveX/COM, Kitware offers commercial products to support this. These products are known as the ActiViz component library. See Kitware's Web site at `http://www.kitware.com/ActiViz.htm` for more information.

3.3 Conversion Between Languages

As we have seen, VTK's core is implemented in C++ and then wrapped with the Tcl, Java, and Python programming languages. This means that you have a language choice when developing applications. Your choice will depend on which language you are most comfortable with, the nature of the application, and whether you need access to internal data structures and/or have special performance requirements. C++ offers several advantages over the other languages when you need to access internal data structure or require the highest-performing applications possible. However, using C++ means the extra burden of the compile/link cycle, which often slows the software development process.

You may find yourself developing prototypes in an interpreted language such as Tcl, and then converting them to C++. Or, you may discover example code (in the VTK distribution or from other users) that you wish to convert to your implementation language.

Converting VTK code from one language to another is fairly straightforward. Class names and method names remain the same across languages, what changes are the implementation details and GUI interface, if any. For example, the C++ statement

```
anActor->GetProperty()->SetColor(red,green,blue);
```

in Tcl becomes

```
[anActor GetProperty] SetColor $red $green $blue
```

in Java becomes

```
anActor.GetProperty().SetColor(red,green,blue);
```

and in Python becomes

```
anActor.GetProperty().SetColor(red,green,blue)
```

One major limitation you'll find is that some C++ applications cannot be converted to the other three languages because of pointer manipulation.