Integrating With The Windowing System

At some point in your use of VTK you will probably want to modify the default interaction behavior (in vtkRenderWindowInteractor) or add a graphical user interface (GUI) to your VTK based application. This section will explain how to do this for many common user interface toolkits. Up-to-date and new examples might be found in the `VTK/Examples/GUI` source directory. Begin by reading the next section on changing interaction style. Then (to embed VTK into a GUI) look at "General Guidelines for GUI Interaction" on page 305 and then only the subsection appropriate to your user interface. If you are using a GUI other than the ones covered here, read the subsections that are similar to your GUI.

13.1 vtkRenderWindow Interaction Style

The class vtkRenderWindowInteractor is used to capture mouse and keyboard events in the render window, and then dispatches these events to another class—the interactor style. Therefore, to add a new style of interaction to VTK, you need to derive a new class from vtkInteractorStyle. For example, the class vtkInteractorStyleTrackball implements the trackball style interaction described in "vtkRenderWindowInteractor" on page 48. vtkInteractorStyleJoystickActor or vtkInteractorStyleJoystickCamera implements the joystick interaction style described in the same section. Another option is to use the class vtkInteractorStyleUser. This class allows users to define a new interactor style without subclassing.

Basically, the way this works as follows. vtkRenderWindowInteractor intercepts events occurring in the vtkRenderWindow with which it is associated. Recall that on instantiation, vtkRenderWindowInteractor actually instantiates a device/windowing-specific implementation—either vtkXRenderWindowInteractor (Unix) or vtkWin32RenderWindowInteractor (Windows). The event intercepts are enabled when the vtkRenderWindowInteractor::Start() method is called. These events are in turn forwarded to the vtkRenderWindowInteractor::InteractorStyle instance. The interactor style processes the events as appropriate.

Here is a list of the available interactor styles with a brief description of what each one does.

- vtkInteractorStyleJoystickActor — implements joystick style for actor manipulation.
- vtkInteractorStyleJoystickCamera — implements joystick style for camera manipulation.
- vtkInteractorStyleTrackballActor — implements trackball style for actor manipulation.
- vtkInteractorStyleTrackballCamera — implements trackball style for camera manipulation.
- vtkInteractorStyleSwitch — manages the switching between trackball and joystick mode, and camera and object (actor) mode. It does this by intersection keystrokes and internally switching to one of the modes listed above.
- vtkInteractorStyleTrackball — a legacy class, this interactor style combines both the actor and camera trackball modes.
- vtkInteractorStyleFlight — a special class that allows the user to "fly-through" complex scenes.
- vtkInteractorStyleImage — a specially designed interactor style for images. This class performs window and level adjustment via mouse motion, as well as pan and dolly constrained to the *x*-*y* plane.
- vtkInteractorStyleUnicam — single button camera manipulation. Rotation, zoom, and pan can all be performed with the one mouse button.

If one of these interactor style does not suit your needs, you can create your own interactor style. To create your own, subclass from vtkInteractorStyle. This means overriding the following methods (as described in vtkInteractorStyle.h):

```
// Description:
// Generic event bindings must be overridden in subclasses
virtual void OnMouseMove  (int ctrl, int shift, int X, int Y);
virtual void OnLeftButtonDown(int ctrl, int shift, int X, int Y);
virtual void OnLeftButtonUp  (int ctrl, int shift, int X, int Y);
virtual void OnMiddleButtonDown(int ctrl, int shift, int X, int Y);
virtual void OnMiddleButtonUp  (int ctrl, int shift, int X, int Y);
virtual void OnRightButtonDown(int ctrl, int shift, int X, int Y);
virtual void OnRightButtonUp  (int ctrl, int shift, int X, int Y);
```

```
// Description:
// OnChar implements keyboard functions, but subclasses can override this
// behavior
virtual void OnChar(int ctrl, int shift, char keycode, int repeatcount);
virtual void OnKeyDown(int ctrl, int shift, char keycode,
                       int repeatcount);
virtual void OnKeyUp(int ctrl, int shift, char keycode, int repeatcount);
virtual void OnKeyPress(int ctrl, int shift, char keycode,
                        char *keysym, int repeatcount);
virtual void OnKeyRelease(int ctrl, int shift, char keycode,
                          char *keysym,int repeatcount);

// Description:
// These are more esoteric events, but are useful in some cases.
virtual void OnConfigure(int width, int height);
virtual void OnEnter(int ctrl, int shift, int x, int y);
virtual void OnLeave(int ctrl, int shift, int x, int y);
```

Then to use the interactor style, associate it with vtkRenderWindowInteractor via the SetInteractorStyle() method.

13.2 General Guidelines for GUI Interaction

For the most part VTK has been designed to isolate the functional objects from the user interface. This has been done for portability and flexibility. But sooner or later you will need to create a user interface, so we have provided a number of hooks to help in this process. These hooks are called *user methods* and they are discussed in Chapter 3 (see "User Methods, Observers, and Commands" on page 28). Recall that the essence of user methods in VTK is that any class can invoke an event. If an observer is registered with the class that invokes the event, then an instance of vtkCommand is executed which is the implementation of the callback. There are a variety of events invoked by different VTK classes that come in handy when developing a user interface. A partial list of the more useful events is provided below.

All filters (subclasses of vtkProcessObject) invoke these events:

- StartEvent
- EndEvent
- ProgressEvent

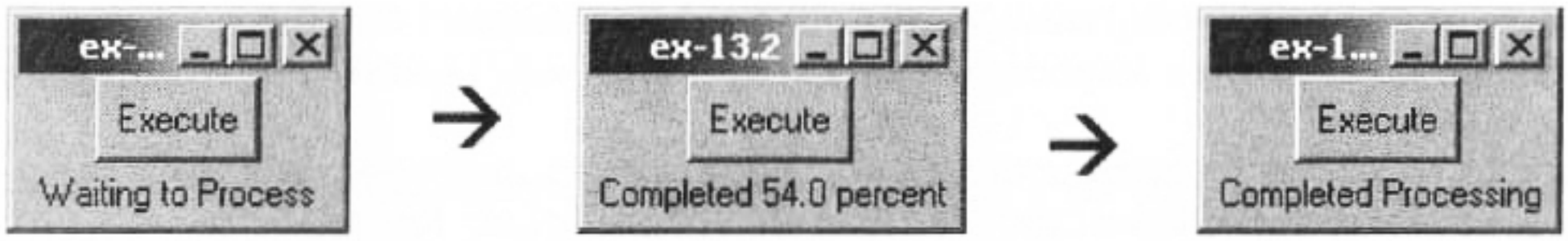

Figure 13–1 GUI feedback as a result of invoking the StartEvent, ProgressEvent, and EndEvent as a filter executes.

The class vtkRenderWindow invokes these events (while rendering):

- AbortCheckEvent

The classes (and subclasses of) vtkActor, vtkVolume, and vtkPicker invoke these events while picking:

- PickEvent
- StartPickEvent (available to vtkPicker only)
- EndPickEvent (available to vtkPicker only)

The class vtkRenderWindowInteractor invokes these events:

- StartPickEvent — while picking
- EndPickEvent — while picking
- UserEvent — in response to "u" keypress in the render window
- ExitEvent — in response to the "e" keypress in the render window

And don't forget that you can define your own vtkInteractorStyle with its own set of special callbacks.

The StartEvent, EndEvent, and ProgressEvent invocations can be used to provide feedback to the user on what the application is doing and how much longer it will take. All filters invoke the StartEvent and EndEvent events if they have been defined. ProgressEvents are supported by imaging filters, some readers, and many (but not all) of the visualization filters. The AbortCheckEvent can be used to allow the user to interrupt a render that is taking too long (requires the use of vtkLODActors). The PickEvents and virtual methods can be used to override the default VTK interactor behavior so that you can create your own custom interaction style.

To help you get started, consider the following two examples that incorporate user methods. Both are written in Tcl but can be easily converted to other languages. The first defines a proc that catches the ProgressEvent to display the progress of the vtkImageShrink3D filter. It then catches the EndEvent to update the display to indicate the processing is complete (**Figure 13–1**).

```
# Halves the size of the image in the x, y and z dimensions.
package require vtk

# Image pipeline
vtkImageReader reader
    reader SetDataByteOrderToLittleEndian
    reader SetDataExtent 0 255 0 255 1 93
    reader SetFilePrefix $env(VTK DATA ROOT)/Data/headsq/quarter
    reader SetDataMask 0x7fff

vtkImageShrink3D shrink
    shrink SetInput [reader GetOutput]
    shrink SetShrinkFactors 2 2 2
    shrink AddObserver ProgressEvent {
      .text configure -text \
                "Completed [expr [shrink GetProgress]*100.0] percent"
      update
    }
    shrink AddObserver EndEvent {
     .text configure -text "Completed Processing"
     update
    }

button .run -text "Execute" -command{
    shrink Modified
    shrink Update
}
label .text -text "Waiting to Process"
pack .run .text
```

For pipelines consisting of multiple filters, each filter could provide an indication of its progress. You can also create generic Tcl procs (rather than define them in-line as here) and assign them to different filter.

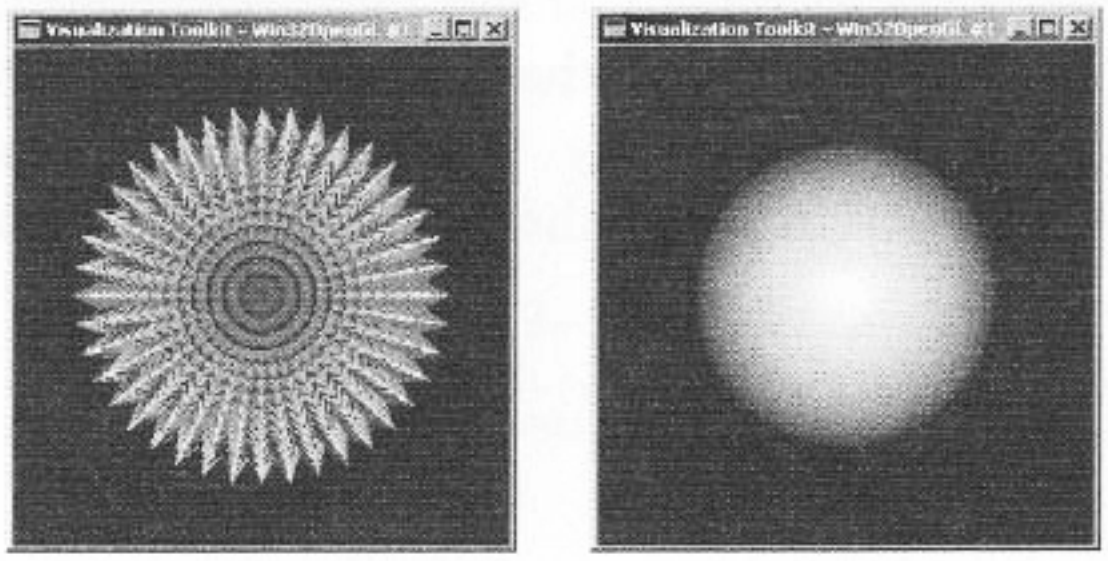

Figure 13–2 Aborting the rendering process. This is used to improve overall interaction with VTK. Rendering can be aborted whenever an event is pending. Make sure that you are using immediate mode rendering.

The second example makes use of the AbortCheckEvent to interrupt a long render if a mouse event is pending. Most of the code is typical VTK code. The critical changes are that you must use instances of vtkLODActor, which is currently only supported under OpenGL, it is best if you turn on GlobalImmediateModeRendering() since the abort method cannot be invoked in the middle of display list processing, and finally you must add a few lines of code to process the abort check. In this example we define a simple procedure called TkCheckAbort which invokes the GetEventPending() method of vtkRenderWindow and then sets the AbortRender instance variable to 1 if an event is pending. The resolution of the mace model has been dramatically increased (**Figure 13–2**(left)) so that you can see the effects of using the AbortRender logic. Feel free to adjust the resolution of the sphere to suit your system. If everything is working properly then you should be able to quickly rotate and then zoom without waiting for the full resolution sphere to render in between the two actions (**Figure 13–2**(right)).

```
package require vtk
package require vtkinteraction
# this is a tcl version of the Mace example

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

# create a sphere source and actor
vtkSphereSource sphere
    sphere SetThetaResolution 40
    sphere SetPhiResolution 40
vtkPolyDataMapper   sphereMapper
    sphereMapper SetInput [sphere GetOutput]
```

```
    sphereMapper GlobalImmediateModeRenderingOn
vtkLODActor sphereActor
    sphereActor SetMapper sphereMapper

# create the spikes using a cone source and the sphere source
#
  vtkConeSource cone
vtkGlyph3D glyph
    glyph SetInput [sphere GetOutput]
    glyph SetSource [cone GetOutput]
    glyph SetVectorModeToUseNormal
    glyph SetScaleModeToScaleByVector
    glyph SetScaleFactor 0.25
vtkPolyDataMapper spikeMapper
    spikeMapper SetInput [glyph GetOutput]
vtkLODActor spikeActor
    spikeActor SetMapper spikeMapper

# Add the actors to the renderer, set the background and size ren1 AddActor
sphereActor
ren1 AddActor spikeActor
ren1 SetBackground 0.1 0.2 0.4
renWin SetSize 300 300

# render the image
iren AddObserver UserEvent {wm deiconify .vtkInteract}

set cam1 [ren1 GetActiveCamera]
$cam1 Zoom 1.4
iren Initialize

proc TkCheckAbort {} {
  if {[renWin GetEventPending] != 0} {renWin SetAbortRender 1}
}
renWin AddObserver AbortCheckMethod TkCheckAbort

# prevent the tk window from appearing; start the event loop
wm withdraw .
```

13.3 X Windows, Xt, and Motif

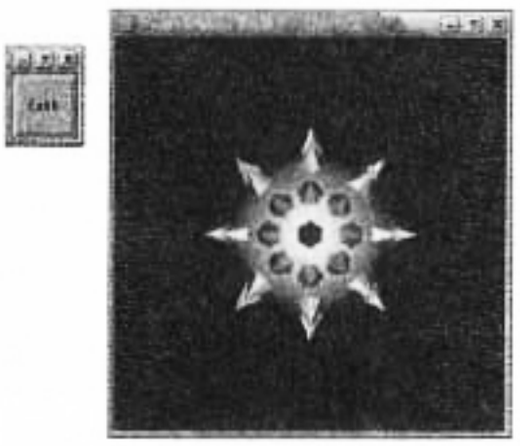

Figure 13–3 Simple Motif application using VTK.

Most traditional UNIX based applications use either Xt or Motif as their widget set. Many of those that don't directly use Xt or Motif end up using Xt at a lower level. There are two common ways to integrate VTK into your Xt (or Motif) based application. These examples might be found in the `VTK/Examples/GUI/Motif` source directory. First we will look at an example where the VTK rendering window and the application UI are in separate windows (**Figure 13–3**). This helps avoid some problems that can occur if VTK and the UI do not use the same X visual. Both windows will use the same X event loop and the UI can own that loop as is typical. Consider the following example application. It draws a mace into a VTK render window and then creates a Motif push button and associated callback in a separate window.

```
// include OS specific include file to mix in X code
#include "vtkRenderWindow.h"
#include "vtkXRenderWindowInteractor.h"
#include "vtkSphereSource.h"
#include "vtkConeSource.h"
#include "vtkGlyph3D.h"
#include "vtkPolyDataMapper.h"
#include <Xm/PushB.h>
void quit_cb(Widget,XtPointer,XtPointer);
main (int argc, char *argv[])
{
  // X window stuff
  XtAppContext app;
  Widget toplevel, button;
  Display *display;
  // VTK stuff
  vtkRenderWindow *renWin;
  vtkRenderer *ren1;
  vtkActor *sphereActor1, *spikeActor1;
  vtkSphereSource *sphere;
  vtkConeSource *cone;
  vtkGlyph3D *glyph;
  vtkPolyDataMapper *sphereMapper, *spikeMapper;
  vtkXRenderWindowInteractor *iren;
```

The first section of code simply includes the required header files and prototypes a simple callback called `quit_cb`. Then we enter the main function and declare some standard X/Motif variables. Then we declare the VTK objects we will need as before. The only significant change here is the use of the vtkXRenderWindowInteractor subclass instead of the typical vtkRenderWindowInteractor. This subclass allows us to access some additional methods specifically designed for X windows.

```
renWin = vtkXRenderWindow::New();
ren1 = vtkRenderer::New();
renWin->AddRenderer(ren1);
sphere = vtkSphereSource::New();
sphereMapper = vtkPolyDataMapper::New();
sphereMapper->SetInput(sphere->GetOutput());
sphereActor1 = vtkActor::New();
sphereActor1->SetMapper(sphereMapper);
cone = vtkConeSource::New();
glyph = vtkGlyph3D::New();
glyph->SetInput(sphere->GetOutput());
glyph->SetSource(cone->GetOutput());
glyph->SetVectorModeToUseNormal();
glyph->SetScaleModeToScaleByVector();
glyph->SetScaleFactor(0.25);
spikeMapper = vtkPolyDataMapper::New();
spikeMapper->SetInput(glyph->GetOutput());
spikeActor1 = vtkActor::New();
spikeActor1->SetMapper(spikeMapper);
ren1->AddActor(sphereActor1);
ren1->AddActor(spikeActor1);
ren1->SetBackground(0.4,0.1,0.2);
```

The above code is standard VTK code to create a mace.

```
// do the xwindow ui stuff
XtSetLanguageProc(NULL,NULL,NULL);
toplevel = XtVaAppInitialize(&app,"Sample",NULL,0,
                             &argc,argv,NULL, NULL);

// get the display connection and give it to the renderer
display = XtDisplay(toplevel);
renWin->SetDisplayId(display);

// we typecast to an X specific interactor
// Since we have decided to make this an X program
```

```
  iren = vtkXRenderWindowInteractor::New();
  iren->SetRenderWindow(renWin);
  iren->Initialize(app);

  button = XtVaCreateManagedWidget("Exit",
                                   xmPushButtonWidgetClass,
                                   toplevel,
                                   XmNwidth, 50,
                                   XmNheight, 50,NULL);

  XtRealizeWidget(toplevel);
  XtAddCallback(button,XmNactivateCallback,quit_cb,NULL);
  XtAppMainLoop(app);
}
// simple quit callback
void quit_cb(Widget w,XtPointer client_data, XtPointer call_data)
{
  exit(0);
}
```

Finally we perform the standard Xt initialization and create our toplevel shell. The next few lines are very important. We obtain the X display id from the toplevel shell and tell the render window to use the same display id. Next we create the vtkXRenderWindowInteractor, set its render window and finally initialize it using the X application context from our earlier XtVaAppInitialize() call. Then we use standard Xt/Motif calls to create a push button, realize the toplevel shell, and assign a callback to the pushbutton. The last step is to start the XtAppMainLoop. The `quit_cb` is a simple callback that just exits the application. It is critical in this type of approach that the VTK render window interactor is initialized prior to creating the rest of your user interface. Otherwise some event may not be handled correctly.

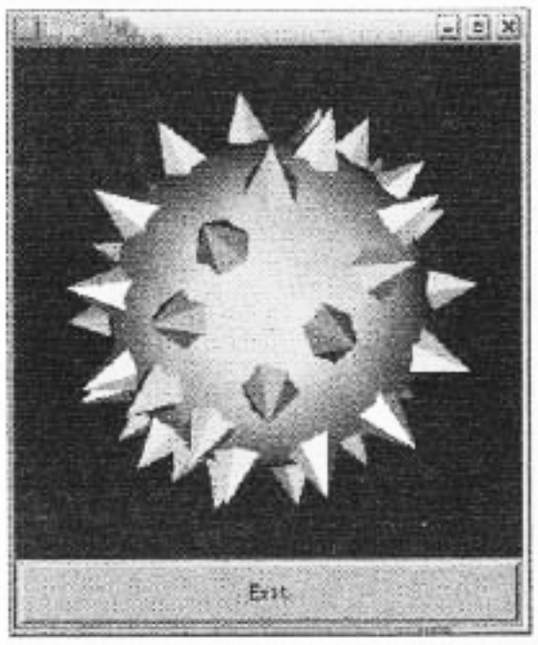

Figure 13–4 Simple Motif application with integrated VTK render window.

Now we will modify the proceeding example so that the rendering window is part of the user interface (**Figure 13–4**). This will require that we create a toplevel shell with a visual that VTK can use for rendering. Fortunately vtkXOpenGLRenderWindow includes some methods for helping you create an appropriate toplevel shell. Much of the code in the following example is the same as the previous example. The differences will be discussed shortly.

```
// include OS specific include file to mix in X code
#include "vtkXOpenGLRenderWindow.h"
#include "vtkXRenderWindowInteractor.h"
#include "vtkSphereSource.h"
#include "vtkConeSource.h"
#include "vtkGlyph3D.h"
#include "vtkPolyDataMapper.h"
#include <Xm/PushB.h>
#include <Xm/Form.h>
void quit_cb(Widget,XtPointer,XtPointer);
main (int argc, char *argv[])
{
  // X window stuff
  XtAppContext app;
  Widget toplevel, form, toplevel2, vtk;
  Widget button;
  int depth;
  Visual *vis;
  Display *display;
  Colormap col;

  // VTK stuff
  vtkXOpenGLRenderWindow *renWin;
  vtkRenderer *ren1;
  vtkActor *sphereActor1, *spikeActor1;
  vtkSphereSource *sphere;
  vtkConeSource *cone;
  vtkGlyph3D *glyph;
  vtkPolyDataMapper *sphereMapper, *spikeMapper;
  vtkXRenderWindowInteractor *iren;

  renWin = vtkXOpenGLRenderWindow::New();
  ren1 = vtkRenderer::New();
  renWin->AddRenderer(ren1);

  sphere = vtkSphereSource::New();
  sphereMapper = vtkPolyDataMapper::New();
  sphereMapper->SetInput(sphere->GetOutput());
  sphereActor1 = vtkActor::New();
  sphereActor1->SetMapper(sphereMapper);
  cone = vtkConeSource::New();
  glyph = vtkGlyph3D::New();
  glyph->SetInput(sphere->GetOutput());
  glyph->SetSource(cone->GetOutput());
  glyph->SetVectorModeToUseNormal();
  glyph->SetScaleModeToScaleByVector();
```

```
glyph->SetScaleFactor(0.25);
spikeMapper = vtkPolyDataMapper::New();
spikeMapper->SetInput(glyph->GetOutput());
spikeActor1 = vtkActor::New();
spikeActor1->SetMapper(spikeMapper);
ren1->AddActor(sphereActor1);
ren1->AddActor(spikeActor1);
ren1->SetBackground(0.4,0.1,0.2);
// do the xwindow ui stuff
XtSetLanguageProc(NULL,NULL,NULL);
toplevel1 = XtVaAppInitialize(&app,"Sample",NULL,0,
                              &argc,argv,NULL, NULL);
```

The initial code is relatively unchanged. In the beginning we have included an additional Motif header file to support the Motif form widget. In the main function we have added some additional variables to store some additional X properties.

```
// get the display connection and give it to the renderer
display = XtDisplay(toplevel);
renWin->SetDisplayId(display);
depth = renWin->GetDesiredDepth();
vis = renWin->GetDesiredVisual();
col = renWin->GetDesiredColormap();

toplevel2 = XtVaCreateWidget("top2",
     topLevelShellWidgetClass,toplevel,
     XmNdepth, depth,
     XmNvisual, vis,
     XmNcolormap, col,
     NULL);
```

Here is where the significant changes begin. We use the first toplevel shell widget to get an X display connection. We then set the render window to use that display connection and then query what X depth, visual, and colormap would be best for it to use. Then we create another toplevel shell widget this time explicitly specifying the depth, colormap, and visual. That way the second toplevel shell will be suitable for VTK rendering. All of the child widgets of this toplevel shell will have the same depth, colormap, and visual as `toplevel2`.

```
form      = XtVaCreateWidget("form",xmFormWidgetClass, toplevel2, NULL);
vtk       = XtVaCreateManagedWidget("vtk",
   xmPrimitiveWidgetClass, form,
   XmNwidth, 300, XmNheight, 300,
   XmNleftAttachment, XmATTACH_FORM,
```

```
      XmNrightAttachment, XmATTACH_FORM,
      XmNtopAttachment, XmATTACH_FORM,
      NULL);
  button     = XtVaCreateManagedWidget("Exit",
       xmPushButtonWidgetClass, form,
       XmNheight, 40,
       XmNbottomAttachment, XmATTACH_FORM,
       XmNtopAttachment, XmATTACH_WIDGET,
       XmNtopWidget, vtk,
       XmNleftAttachment, XmATTACH_FORM,
       XmNrightAttachment, XmATTACH_FORM,
       NULL);

  XtAddCallback(button,XmNactivateCallback,quit_cb,NULL);
  XtManageChild(form);
  XtRealizeWidget(toplevel2);
  XtMapWidget(toplevel2);

  // we typecast to an X specific interactor
  // Since we have decided to make this an X program
  iren = vtkXRenderWindowInteractor::New();
  iren->SetRenderWindow(renWin);
  iren->SetWidget(vtk);
  iren->Initialize(app);
  XtAppMainLoop(app);
}
/* quit when the arrow */
void quit_cb(Widget w,XtPointer client_data, XtPointer call_data)
{
  exit(0);
}
```

Finally we create a few Motif widgets including a xmPrimitiveWidgetClass which is what VTK will render into. The form widget has been added simply to handle layout of the button and the rendering window. The SetWidget() call is used in this example to tell the interactor (and hence the render window) what widget to use for rendering.

13.4 MS Windows / Microsoft Foundation Classes

The basics of integration of VTK within the Windows environment has been shown previously (see "Create An Application" on page 28). You can also develop MFC-based appli-

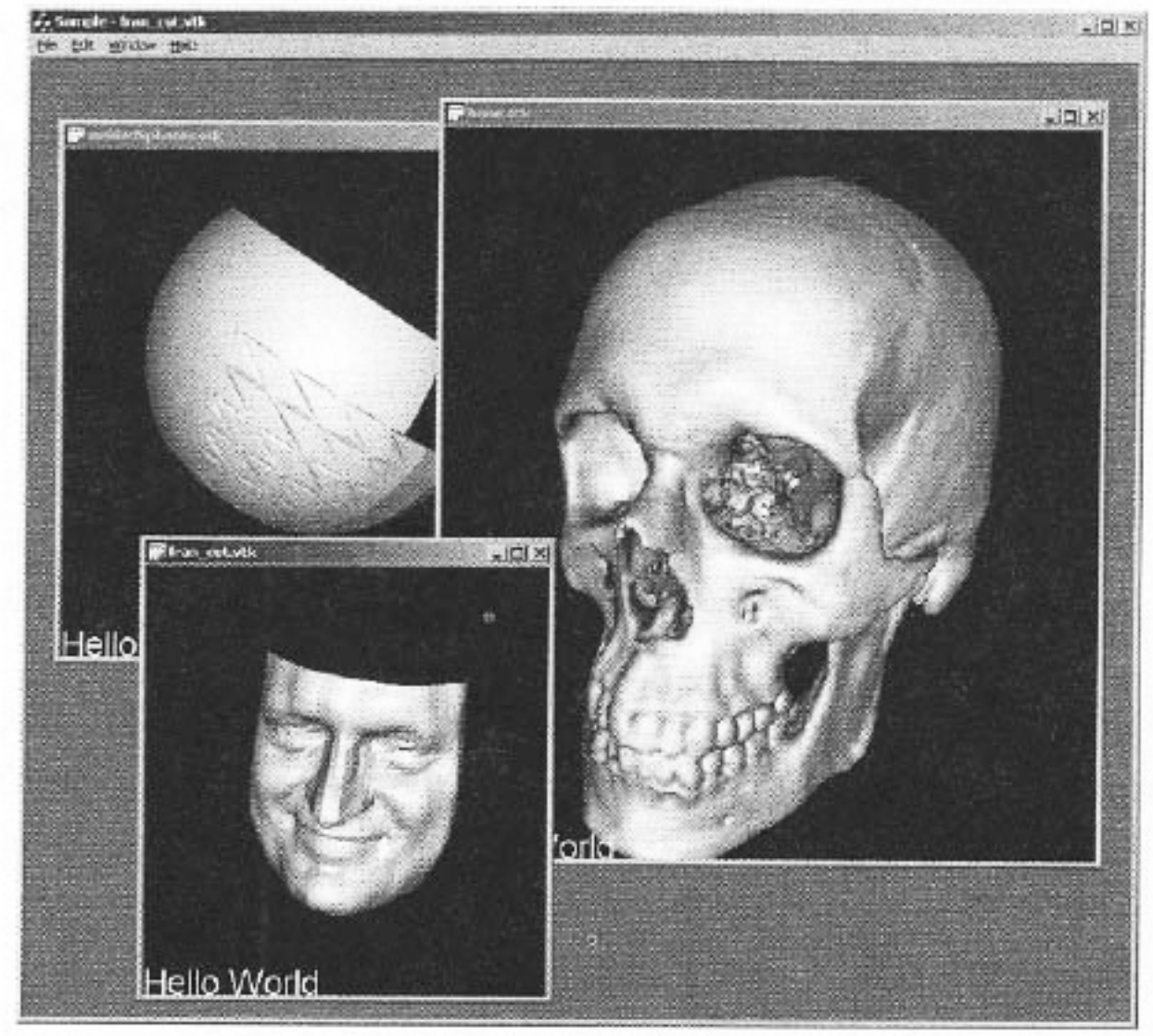

Figure 13–5 A sample application built with the Microsoft multi-document interface.

cations that make use of VTK in two different ways. The first way to use VTK within a MFC based application. Here, following the code from `VTK/Examples/GUI/Win32/SimpleCxx/Win32Cone.cxx`, we create a vtkRenderWindow in the MFC application and if desired, parent it with a MFC-based window. The second way is to make use of the vtkMFCView, vtkMFCRenderView and vtkMFCDocument classes that are provided in the `Examples/GUI/Win32/SampleMFC` subdirectory. In fact, the `Sample.exe` application is a sample MFC-based application that demonstrates the use of these classes. This MDI application (Multi-Document Interface) shows how to open several VTK data files and interact with them through the GUI (**Figure 13–5**). You should be able to copy these classes into new MFC applications you develop. Refer to Chapter 3 to check how the Microsoft Visual Studio project file should be modified to compile this example.

13.5 Tcl/Tk

Integrating VTK with Tcl/Tk user interfaces is typically a fairly easy process thanks to classes such as vtkTkRenderWidget, vtkTkImageViewerWidget, and vtkTkImageWindowWidget. These classes can be used just like you would use any other Tk widget. Up-to-date informations and new examples might be found both in the `VTK/Examples/GUI/Tcl` and `Wrapping/Tcl` source directories. Consider the following example (**Figure 13–6**):

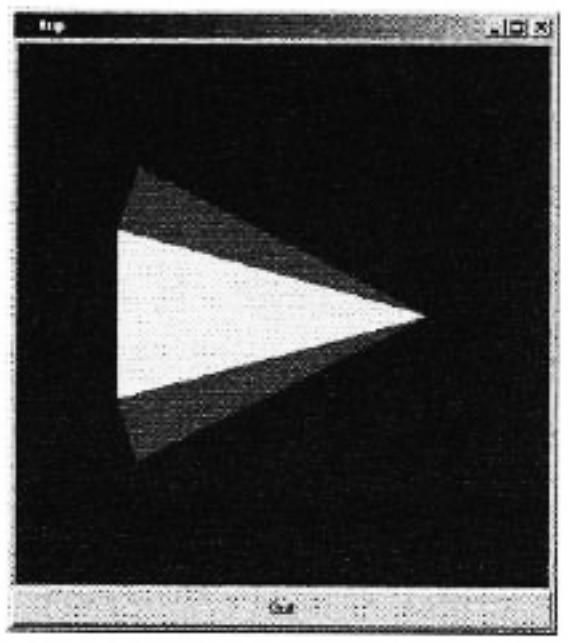

Figure 13–6 Tcl/Tk example.

```
package require vtk
package require vtkinteraction
# This script uses a vtkTkRenderWidget to create a
# Tk widget that is associated with a vtkRenderWindow.

# Create the GUI: two renderer widgets and a quit button
wm withdraw .
toplevel .top
frame .top.f1
vtkTkRenderWidget .top.f1.r1 -width 400 -height 400
button .top.btn  -text Quit -command exit
pack .top.f1.r1 -side left -padx 3 -pady 3 -fill both \
    -expand t
pack .top.f1  -fill both -expand t
pack .top.btn -fill x
# Get the render window associated with the widget.
set renWin [.top.f1.r1 GetRenderWindow]
vtkRenderer ren1
$renWin AddRenderer ren1

# bind the mouse events
BindTkRenderWidget .top.f1.r1

# create a Cone source and actor
vtkConeSource cone
vtkPolyDataMapper   coneMapper
    coneMapper SetInput [cone GetOutput]
    coneMapper GlobalImmediateModeRenderingOn
vtkLODActor coneActor
```

```
    coneActor SetMapper coneMapper
# Add the actors to the renderer, set the background and size
#
ren1 AddActor coneActor
ren1 SetBackground 0.1 0.2 0.4
```

The first line is the standard `package require vtk` command that is used to load the VTK Tcl package. It may also be used on UNIX if the VTK target has been built with shared libraries (but this is not required). The `vtkinteraction` package contains default bindings for handling mouse and keyboard events for a render widget. Specifically it defines the `BindTkRenderWidget` proc which sets up those bindings for a particular vtkTkRenderWidget. Next we withdraw the default toplevel widget and create a new one called `.top`. On some systems you may need to create `.top` with the following line instead of the one given above:

```
toplevel .top -visual best
```

Next we create and pack the frame, vtkTkRenderWidget, and a button in the traditional Tk manner. The next line queries the vtkTkRenderWidget for the underlying render window that it is using. We store this in a variable called `renWin`. We then create a renderer, associate it with the render window, and then bind the mouse events to the vtkRenderWidget using the `BindTkRenderWidget proc`. Finally we create a cone and actor in the normal manner. If you wish the render window can be provided as an argument on the creation of the vtkTkRenderWidget as follows:

```
vtkRenderWindow renWin
vtkTkRenderWidget .top.f1.r1 \
    -width 400 -height 400 -rw renWin
```

Then simply use `renWin` instead of `$renWin` since it is now an instance, not a variable reference.

For your application development you will probably want to customize the event handling. The best way to do this is to make a copy of `TkInteractor.tcl` located in `VTK/Wrapping/Tcl` and then edit it to suit your preferences. The format of this file is fairly straight forward. The beginning of the file defines the `BindTkRenderWidget` proc which associates events with specific Tcl procedures. The remainder of the file defines these procedures. The same techniques used with vtkTkRenderWidget can be used with vtkTkImageViewerWidget and vtkTkImageWindowWidget for image processing. Instead of having a `-rw` option and GetRenderWindow() method, vtkTkImageViewerWidget supports `-iv` and GetImageViewer().

vtkTkImageWindowWidget supports `-iw` and GetImageWindow(). Likewise, there is a `TkImageViewerInteractor.tcl` file that provides some default event handling for the image viewer, which are loaded by the `vtkinteraction` package.

When using the vtkTkWidget classes you should not use the interactor classes such as vtkRenderWindowInteractor. While it may work, the behavior may be unstable. Normally you should use either an interactor or a vtkTkWidget but never both for a given window.

13.6 Java

The *Visualization Toolkit* includes a class specially designed to help you integrate VTK into your Java based application. This is a fairly tricky procedure since Java does not provide any "public" classes to support native code integration. It is made more difficult by the fact that Java is a multithreaded language and yet windowing systems such as X11R5 do not support multithreaded user interfaces. To help overcome these difficulties, we have provided a Java class called `vtkPanel`. This class works with Java to make a vtkRenderWindow appear like a normal Java AWT Canvas. The `App2.java` example which can be found in the your `java` subdirectory makes use of the `vtkPanel` class. `vtkPanel.java` can be found in `vtk.jar` for executable distributions and in `vtk/java/vtk` for source code distributions.

Be aware that the `vtkPanel` class comes in two different varieties: Windows and UNIX. The code to support both operating systems is in the same file. If you experience difficulty you should take a look at your copy of `vtkPanel.java` and make sure the appropriate lines are commented out. Be aware that `vtkPanel` uses some classes that are not officially supported by Sun and may not be portable to other vendors' Java development environments. We of course don't like using these classes, but Sun makes the rules and they have made it fairly difficult to integrate user interface code into their language. But as long as you use the JDK you should be okay.