Kitware SOURCE

SOFTWARE DEVELOPER'S QUARTERLY

Issue 3 • Jan 2007

TABLE OF CONTENTS

EDITOR'S NOTE

This issue of the Kitware Software Developer's Quarterly newsletter contains three diverse articles related to Kitware's open source projects. Will Schroeder provides an overview of interactive elements known as 3D Widgets that can be embedded into a VTK scene to intuitively control various visualization parameters. Andy Cedilnik describes a cross-platform software packaging tool known as CPack that is distributed as part of CMake. Amy Squillacote contributes a detailed tutorial describing the steps required to add customized functionality, including algorithms and user interface, to ParaView 2.6.

Kitware would like to encourage contributions to this newletter from our active developer community by offering a free five-volume set of Kitware books for any accepted article. Perhaps you have contributed to one of the open source projects and would like to write a technical article describing your enhancement. Or perhaps you are developing a product that is built upon one or more of Kitware's open source projects, and would like to document your success or lessons learned. Please send your ideas to kitware@kitware.com.

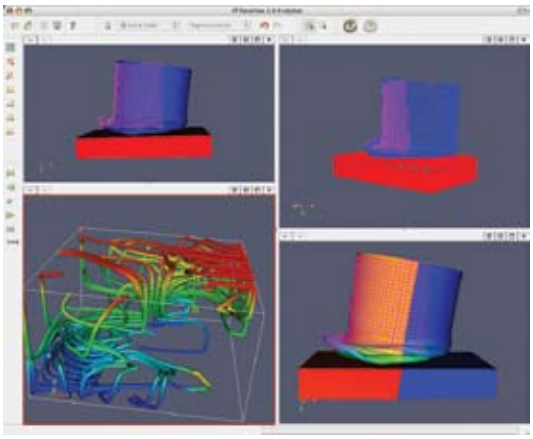
This newsletter is just one of a suite of products and services that Kitware offers to assist developers in getting the most out of our open source products. Each project web site contains links to free resources including mailing lists, documentation, FAQs and Wikis. In addition, Kitware offers technical books and user's guides, consulting services, support contracts, and training courses. For more information on Kitware's products and services, please visit our web site at www.kitware.com.

RECENT RELEASES

PARAVIEW III ALPHA RELEASE

In January 2007, a new snapshot (2.9.7) of the alpha release of ParaView III was created. This is the eighth monthly snapshot; the first was created in June 2006. It includes binaries for Windows, Linux (32 and 64 bit) and Mac OS X. To download the snapshot, visit http://paraview.org/Wiki/ParaView_III_snapshots. The following new features have been added over the past three months:

- Improved python support. We added a programmable python filter. The python source can be entered on the object inspector.
- Better default values for the object inspector. These are computed based on the input. For example, the default maximum length of the streamlines is now computed based on the input size.
- Pipeline browser improvements. It is now possible to change inputs as well as add multiple inputs to filters that support it.
- First pass at annotation. We added a text source. This allows creation of an arbitrary number of labels on the display. The text can be moved by clicking and dragging. There are some glitches with the interaction. We will fix this soon.
- Reading multiple exodus files.
- Improved chart code.
- Improved testing framework, which now includes python.
- Improved pipeline browser.
- Play/Pause in VCR controls.
- Light Kit controls added.
- Auto-generated and custom panel improvements.
- More robust server connection.
- File dialog improvements.



This screen shot highlights ParaView III's support for multiple viewing windows.

CMAKE RELEASE

CMake 2.4.6 was released in January 2007. It is available for download at http://www.cmake.org/HTML/Download.html. Improvements since version 2.4.3 include the following.

- Support for Windows dll version numbers.
- Improved Find/Use wxWidgets.
- Improved FindJava and FindJNI.
- Added FILE_IS_NEWER to IF command.
- Added OPTIONAL to INSTALL command.
- Added SORT and REVERSE to LIST command.
- Added SYMBOLIC as a source file property.
- Much faster dependency scanning.
- Improved support for Visual Studio 8.
- Support for QNX.
- Added APPEND option to ADD_CUSTOM_COMMAND.
- Added VERBATIM option to ADD_CUSTOM_COMMAND and ADD_CUSTOM_TARGET.
- Added EXCLUDE_FROM_ALL option for ADD_LIBRARY and ADD_EXECUTABLE.
- Improved FindKDE3 and FindKDE4.
- Improved FindRuby.
- Improved FindQt3 and FindQt4.
- Improved FindPNG.

ITK RELEASE

ITK 3.0 was released on November 21 2006.

The main change in this release is the addition of an alternative wrapping system called WrapITK. This system was contributed by Gaetan Lehmann, Zachary Pincus and Benoit Regrain in a paper submitted to the Insight Journal. The paper can be found at the following Insight Journal handle: http://hdl.handle.net/1926/188.

WrapITK uses CMake macros in order to generate the instantiations of ITK templated classes. The instantiations are then processed using CableSwig in order to generate wrappings for Tcl, Python, and Java. The previous wrapping mechanism is still available. Users can select what wrapping to use by setting CMake variables at configuration time.

A review of copyrights and licenses of different subcomponents of the Insight Toolkit was also performed. Several components that were not compatible with the BSD license were removed.

The full list of changes in this release can be found at http://www.itk.org/Wiki/ITK_Release_3.0.

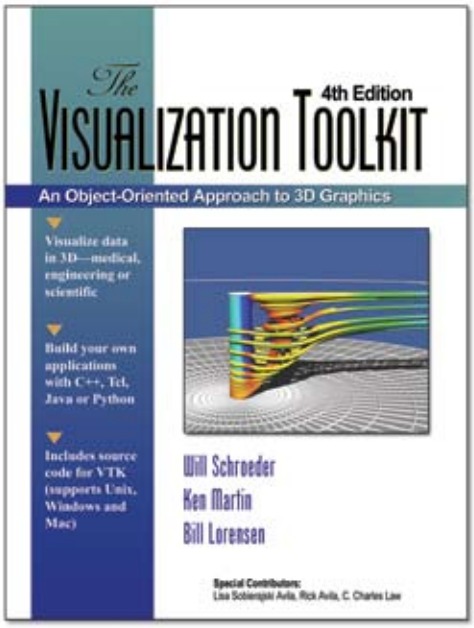
KWWIDGETS UPDATE

There has not been an official release of KWWidgets. However, the following features have been added to the project recently. You can download KWWidgets from http://www.kwwidgets.org/Wiki/KWWidgets#Download.

- State machine framework with workflow wizard. (See vtkKWStateMachine, vtkKWWizardWorkflow.) This is currently used by the EMSegment module in NAMIC Slicer3. The vtkKWStateMachineDOTWriter can use its output to create figures/diagrams automatically using graphviz's dot, or dynamically into a Wiki that supports it.
- VTK's Error/Warning macros can be redirected to a vtkKWLogWidget. This is currently used by the main application of NAMIC Slicer3. This widget can be used to display various types of records/events in the form of a multi-column log. Each record is time-stamped automatically, and the interface allows the user to sort the list by time, type, or description. This widget can be inserted in any widget hierarchy, or used as a standalone dialog through the vtkKWLogDialog class.
- Many new widgets have been added including,
  - vtkKWListBoxWithScrollbarsWithLabel
  - vtkKWMatrixWidget
  - vtkKWMatrixWidgetWithLabel
  - vtkKWMultiColumnListWithLabel
  - vtkKWScaleWithLabelSet

TECHNICAL BOOKS

The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, 4th edition (ISBN 1-930934-19-X) has been recently updated for VTK 5. This new hardcover version is 528 pages long and is printed in full color. It is available from Kitware's on-line store (http://www.kitware.com/products/vtktextbook.html) and from Amazon.com.



RECENT PUBLICATIONS

The January issue of *IEEE Software* contains an article by Ken Martin and Bill Hoffman describing Kitware's software process. The article focuses on the software engineering tools and processes used by Kitware for software development. The process includes communication and documentation, revision control, build management with CMake, automated testing with dashboards, and the process for release creation. If your organization is interested in applying Kitware's software process, please contact Kitware for information about setting up a course.

IEEE Software, January/February 2007 (Vol. 24, No. 1) , ISSN: 0740-7459.

VTK'S 3D WIDGETS

INTRODUCTION

Visualization is most effective when users can interact with their data. In the earliest incarnations of visualization systems, this capability was typically supported by providing interactive scripting interfaces such as Python or Tcl, or building applications with conventional 2D GUI's that drove the underlying visualization pipeline. While effective at times, such approaches did not realize the full potential of interactive visualization. Users often want to reach into the 3D scene and directly manipulate the underlying data (for example, by orientating a cut plane or seeding streamlines).

Over the last couple of years, the VTK community has responded to the desire for interactive, 3D visualization by developing a set of widgets. Simply speaking, in VTK a widget is an object in a scene that responds to user events (e.g., mouse clicks) and data changes by corresponding changes in its appearance or behavior. For example, Figure 1 shows a simple vtkLineWidget. In this widget, the end points of the line (which have sphere handles) as well as the line itself can be directly positioned with the mouse. As the figure illustrates, this widget can be used to seed streamlines, as well as for many other purposes, since the widgets are designed to be flexible in application.

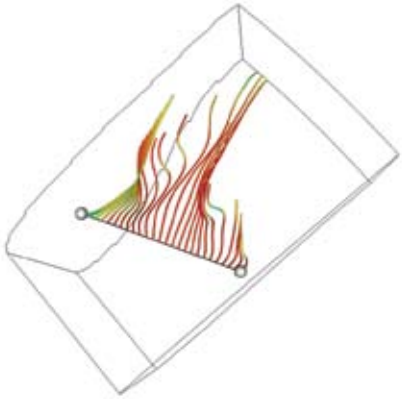


Figure 1: vtkLineWidget

OVERVIEW

VTK's widget design is evolving. In large part this is because no definitive widget sets for 3D visualization have been created. While researchers at Brown University (http://graphics.cs.brown.edu/research/widgetlib/), at the SCI Institute in Utah (http://www.sci.utah.edu/stories/2001/fall_haptics.html) and others have performed leading edge work in this area, because of the complexity of the underlying data and diversity of applications, the full potential of widget capabilities has been mostly unrealized. Thus, while the VTK community is learning from previous work, in many cases we are creating new technology without established precedent.

The earliest attempts at widgets in VTK created monolithic objects derived from the vtkInteractorObserver class. This class simply observes events emitted from the vtkRenderWindowInteractor class, which is responsible for managing VTK's render window event loop. (Note that the interactor styles, which translate mouse and keyboard events into camera motion, are also subclasses of vtkInteractorObserver. Interactor styles differ from widgets in that they do not provide a geometric representation in the 3D scene.) These early widgets, while demonstrating the utility of the widget concept, suffered from some design deficiencies. First, they provided no mechanism to change the event bindings to customize behavior, and second, the monolithic design combined both the event processing and geometric representation, meaning that changing the appearance of the widget was difficult, requiring a complete rewrite. As a result, a second generation widget design, which is the one described in this article, has evolved.

Before getting into details about the widget implementation, you may wish to orient yourself by checking out the latest CVS version of VTK, and studying the VTK/Widgets subdirectory. Also, for those wishing more details than this article can provide, the wiki page http://www.vtk.org/Wiki/VTKWidgets provides detailed information suitable for the developer. Figure 2 below illustrates just a few of the many widgets that are currently extant in VTK.

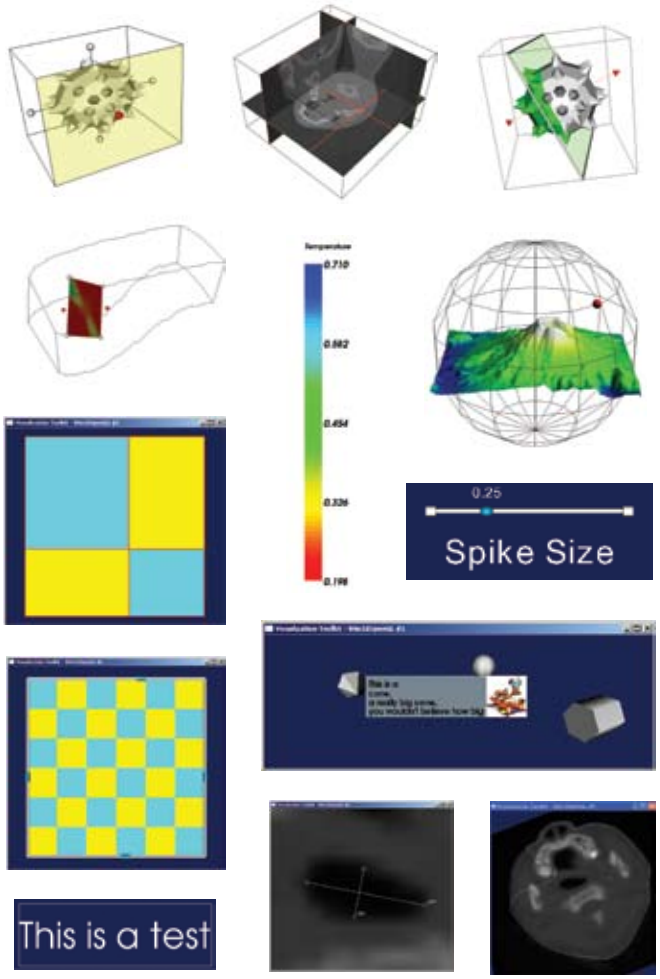


Figure 2: Some of the many widgets found in VTK.

A brief description of these widgets follows. Note that this is a subset chosen to exemplify the breadth of VTK's current capabilities.

- vtkBoxWidget – a 3D box that can be translated, scaled, and/or oriented, producing an output transformation matrix.
- vtkImagePlaneWidget – control planes embedded in a volume.

- vtkImplicitPlaneWidget – orient a plane using its normal and origin.
- vtkPlaneWidget – manipulate the extent and orientation of a finite plane.
- vtkScalarBarWidget – position a vtkScalarBar (i.e., data legend).
- vtkSphereWidget – orient a point in a spherical coordinate system.
- vtkContourWidget – define and edit contours; typically used for segmentation.
- vtkRectilinearWipeWidget – display two widgets simultaneously in a window pane (used to compare images, the panes can be interactively positioned).
- vtkSliderWidget – control a scalar value with a 2D or 3D slider.
- vtkBalloonWidget – pop up text and/or images when the mouse hovers over an object.
- vtkTextWidget – control the position and size of text.
- vtkDistanceWidget – measure the distance between two points.
- vtkBiDimensionalWidget. – measure in two orthogonal directions across an image.

Many more widgets exist, but this provides an overview of the capabilities that can be found in VTK.

IMPLEMENTATION

VTK's current widget design supports the ability to customize event bindings and decouples the widget representation (i.e., its geometry) from event processing. Thus it is possible to reprogram event bindings and create a new appearance for a widget. In this section we provide a brief overview of this capability.

Decoupling Event Processing from Representation. Second generation VTK widgets are implemented in two parts. The first, the part responsible for event processing, inherits from vtkAbstractWidget. The second, the part responsible for geometric representation, inherits from the abstract class vtkWidgetRepresentation. This relationship is depicted in Figure 3.

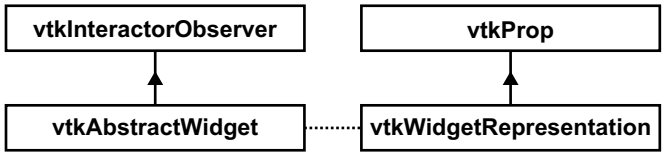


Figure 3: Overview of VTK widget design.

Of special note is that widget representations are a subclass of vtkProp. This means that, like all actors in VTK, they can be added to vtkRenderer and appear in the scene. However, they have the added distinction that they can cooperate with appropriate subclasses of vtkAbstractWidget to create a 3D widget. This separation, besides providing a relatively easy way to change the appearance of a widget (by swapping out its representation), is important in parallel processing or client server applications. In such situations, a single client widget may drive multiple representations (e.g., imagine a power wall display where a single sequence of events on the client may control multiple representations distributed across multiple processors).

Customized Event Bindings. The first generation VTK widgets were implemented with hard-wired event bindings; e.g., a "LeftButtonPressEvent" was directly implemented to execute a particular action. Changing the action/event binding could not be changed without re-coding the widget. This is a problem when users desire different bindings due to personal preference, use of different interaction devices, etc. To address this problem, the second generation widgets create a separation between VTK events and widget events. An event translator is used to translate a VTK event into a widget event. In turn, the widget event is mapped into a method invocation. Figure 4 provides a logical overview of this process.

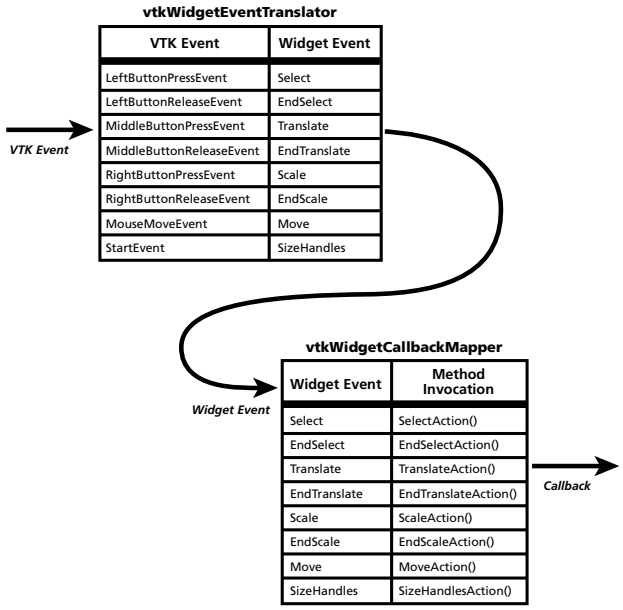


Figure 4: Widget event translation.

Other Considerations. VTK's widget design allows multiple widgets to be active simultaneously and in conjunction with camera manipulators (i.e., instances of vtkInteractorStyle or any other subclass of vtkInteractorObserver). This works because most widgets are spatially sensitive and/or beacuse event processing can be ordered. When events are emitted by the interactor, active widgets may receive the event if they have registered to receive it, and if so, they further evaluate the event to determine whether it lies within the spatial domain encompassed by the widget. If so, the widget will process the event, and usually abort the processing of the event so that other widgets do not also react to it. If not processed, the event is processed by the next registered observer. VTK's event processing system enables observers to prioritize themselves through a combination of setting their priority and tracking the order in which they register for events. Thus users can manage multiple widgets and control situations in which contention for events or other resources is a possibility.

Users will also notice that carefully implemented widgets invoke cursor change requests to provide hints to the user that a widget is active and what the behavior is likely to be. For example, when passing the mouse cursor over a corner of the text widget, the cursor changes to indicate that a resize operation may take place.

EXAMPLES

With this brief introduction behind us, it is time to examine how this plays out in practice. Creating a widget requires instantiating the widget and its representation, and then associating them with each other as follows.

```
vtkLogoRepresentation *rep =
   vtkLogoRepresentation::New();
rep->SetImage(image1->GetOutput());
rep->GetImageProperty()->SetOpacity(0.67);

vtkLogoWidget *widget = vtkLogoWidget::New();
widget->SetInteractor(iren);
widget->SetRepresentation(rep);
```

Here we create a logo widget and its representation. This widget simply provides a method to position and resize an image on the renderer's overlay plane (the image may be translucent). The representation typically requires specific data members to be specified. (Here the image is provided and its opacity specified.) Then the representation is associated with the widget. Finally, widgets must know which vtkRenderWindowInteractor they are observing; this is specified via the SetInteractor(vtkRenderWindowInteractor*) method.

(Note: often widgets are created in the "off" state, that is, they are not listening to any events except the activation event which is typically defined as "keypress i." Alternatively, invoking the widget->On() method will activate the widget.)

The next example is more complex because we position the widget in the scene, set up its appearance, and override some event bindings as follows.

```
vtkSliderRepresentation3D *sliderRep =
   vtkSliderRepresentation3D::New();
sliderRep->SetValue(0.25);
sliderRep->SetTitleText("Spike Size");
sliderRep->GetPoint1Coordinate()->
   SetCoordinateSystemToWorld();
sliderRep->GetPoint1Coordinate()->SetValue(0,0,0);
sliderRep->GetPoint2Coordinate()->
   SetCoordinateSystemToWorld();
sliderRep->GetPoint2Coordinate()->SetValue(2,0,0);
sliderRep->SetSliderLength(0.075);
sliderRep->SetSliderWidth(0.05);
sliderRep->SetEndCapLength(0.05);

vtkSliderWidget *sliderWidget =
   vtkSliderWidget::New();
sliderWidget->GetEventTranslator()->SetTranslation(
   vtkCommand::RightButtonPressEvent,
   vtkWidgetEvent::Select);
sliderWidget->GetEventTranslator()->SetTranslation(
   vtkCommand::RightButtonReleaseEvent,
   vtkWidgetEvent::EndSelect);
sliderWidget->SetInteractor(iren);
sliderWidget->SetRepresentation(sliderRep);
```

Typically the widget representations provide special methods that couple the widget to data. For example, the vtkCheckerboardRepresentation provides methods to associate the widget with vtkImageData. Thus interacting with the widget can directly modify the image and any data processing pipeline dependent upon it. Further, in some cases more than one representation is available; for example the representations vtkSliderRepresentation2D and vtkSliderRepresentation3D provide slider behavior in a 2D (overlay) and 3D implementation, respectively.

WRITING YOUR OWN WIDGET

Writing your own widget is not for the novice. It requires knowledge of VTK's rendering process, event processing, transformations, and geometry engine. It also requires dogged determination to polish the widget's appearance and behavior. Ask any widget implementer, and they will likely tell you they are never completely satisfied with the behavior or appearance of the widget in certain conditions. As is typical in many implementation tasks, the best place to start is to study existing implementations, focusing closely on those implementations that are similar to the new one. For example, if you are creating a widget that renders in the 2D overlay plane, study the various widgets that do the same. Also a subtle but critically important issue regards the coordinate system in which widget interaction and display occurs. User events naturally occur in the renderer's x-y display space; however 3D widgets are defined in the x-y-z world space. Managing the events and transforming between coordinate systems can be difficult to implement correctly without careful consideration.

If you are programming a new representation to plug into an existing widget, the API between the representation and its associated widget is defined. Thus the developer's attention should be focused on the appearance and behavior of the representation based on a known set of method invocations. Designing the geometry requires an eye towards creating a simple, intuitive interface that is easy to interact with. Further, the widget author must carefully consider the potential widget states and the resulting appearance, including highlighting handles, changing the cursor, and enabling/disabling annotation. It is also important to anticipate extreme interaction behavior. For example, what happens when a widget, such as vtkBoxWidget, is turned inside out?

If you are designing an entirely new widget, then the event processing must be taken into account as well as its representation. This requires defining an API between the two classes. The starting point is to subclass the necessary methods from vtkWidgetRepresentation, and to make the API consistent with other, pre-existing widgets. (vtkWidgetRepresentation cannot fully specify the entire API for all its subclasses because widget behavior is so complex. Rather it defines a framework which subclasses should utilize.) Further, an initial mapping between VTK events and widget behavior must be defined. This requires defining a sequence of callbacks that correspond to events. Of course, you will subclass from vtkAbstractWidget and implement the virtual methods as appropriate.

Finally, before implementing your widget, look for superclasses from which to derive widget behavior. The vtkBorderWidget and its associated representation supports rectangular-box resizing and expects subclasses to place their representation inside the border (e.g., vtkTextWidget). The vtkHoverWidget sets a timer when the mouse stops moving; subclasses can intercept the timer to take action (e.g., displaying text and/or an image next to the cursor, which the subclass vtkBalloonWidget does). The ContourWidget (and associated classes including vtkPointPlacer) defines a framework for creating contours on objects.

WHAT'S NEXT

VTK's widget design and implementation is an ongoing effort. At Kitware, our customers have expressed strong interest in this technology. Due to generous support from the US National Library of Medicine, the National Centers for Biomedical Computing (see na-mic.org), the National Science Foundation, and several commercial customers, we are adding new widgets and addressing several design issues.

Most of the new widgets currently in development are aimed at applications in biomedical computing. For example,

widgets to segment or register image data are becoming increasingly important. Another emerging capability in VTK, that of information visualization, is placing demands on tools for interacting with meta-data.

Many of the current design issues address resource contention. When scenes are populated with many widgets, it is necessary to coordinate events and rendering so that widgets are rendered appropriately, including displaying cursor shape requests correctly. For example, if widgets are prioritized in a particular order, the rendering process must render the widget geometry in consistent fashion.

CONCLUSION

VTK's 3D widgets have shown themselves to be powerful, useful additions to the toolkit. While the current suite of possible widgets is relatively small compared to what is possible, and active development continues, VTK's widget technology is mature enough to benefit most visualization applications. Many users find that once they begin using widgets, their applications become inherently more interactive and powerful. We encourage you to incorporate VTK's widgets into your own work, and provide the community with the necessary feedback to make this technology even better.

Dr. William J. Schroeder *is President and co-founder of Kitware, Inc. His current interests include human-computer interaction, open source software systems, and visualization technology.*

PACKAGING SOFTWARE WITH CPACK

Software development and deployment typically consists of several stages including design, coding, testing, and distribution. Each stage imposes some cost on the project since it requires time and effort to complete it. This is why even a small amount of automation of these stages can be extremely helpful.

A big part of distribution of software is a good packaging strategy. However, most developers opt for a manual packaging technique. This may include manually adding a list of files to be distributed into an external packaging tool, or even developing their own packaging tool. This strategy may work on static projects, but on today's highly agile software projects, the packaging can quickly become out of date with the actual software project. This introduces bugs that software developers generally do not see. Furthermore, when developing software in a cross-platform fashion, packaging bugs can be introduced because developers do not have access to all the platforms.

CPack is a simple packaging tool that is attempting to solve this problem. Its first technology preview release is distributed with CMake 2.4. CPack uses a familiar notion of generators from CMake to abstract platform specific packaging requirements. It then internally invokes the native system packaging tool. Using this approach, the developers do not actually have to have access to all the platforms. They are only required to ensure their platform works properly, and then CPack will perform the work on other platforms.

Currently, there are several generators available. The first group of generators consists of compress-only generators. They include ZIP, TZ (Tar compress), TGZ (Tar GZip), and TBZ2 (Tar BZip2). The second group contains generators that actually perform installation on the system. They include self-extracting Tar GZip for Unix systems (STGZ), Mac OSX Package Maker (PackageMaker), and MS Windows Null Soft Installer System (NSIS). TBZ2, ZIP, and NSIS rely on system-installed programs in order to create the packages. For example the NSIS generator requires the Null Soft Installer System to be installed.

GENERATING INPUT FOR CPACK

To use CPack, the following line must be added to the CMake list file.

```
include(CPack)
```

This line will include the CPack module that will examine the project and pick defaults for the package. CMake will internally create CPackConfig.cmake and CPackSourceConfig.cmake files in the build directory. These files include all the necessary settings that CPack uses to install the project.

The CPack module sets a list of parameters based on certain defaults. However, the user can overwrite all these defaults. For example, by default CPACK_PACKAGE_NAME (used as a base for the package name) is set to the current CMake project name. The user can overwrite this by setting the appropriate CMake variables before including the CPack module.

```
project(MyProject)
...
set(CPACK_PACKAGE_NAME "MySpecialProject")
...
include(CPack)
```

In addition to including the CPack module, one or more INSTALL commands must be used to specify the actual files to be installed. The INSTALL command can be used to install files, libraries, and executables, and even to invoke custom CMake scripts.

CPack can be also used without CMake. When using CPack with CMake, CMake can report to CPack all the necessary configuration options. However, when using CPack without CMake, the developer will have to provide all the needed information. The developer also has to make sure that there is an appropriate install script available. This install script is then used by CPack to populate the directories that will be compressed in the package.

RUNNING CPACK

Once the input files for CPack are provided, the CPack executable can be invoked to generate the actual package. When using CMake to generate CPack input files, CMake will also generate packaging targets in the Make, Visual Studio, or Xcode files. This way the user can simply invoke the packaging target as shown below.

```
make package
```

Alternatively, the user may invoke the CPack executable directly. For example on Mac OSX, the user may invoke the following command.

```
cpack --config CPackConfig.cmake -G PackageMaker
```

To get other command line option, the --help command line option can be used with the CPack executable. This option will also show the list of packaging generators.

CPACK SETTINGS

CPack uses several variables for packaging the project. These variables can be generator-independent or generator-dependent. Generator-independent ones include the following.

```
CPACK_GENERATOR
      - Generator used to create package
CPACK_INSTALL_CMAKE_PROJECTS
      - For each project (path, name, component)
CPACK_CMAKE_GENERATOR
      - CMake Generator used for the projects
CPACK_INSTALL_COMMANDS
      - Extra commands to install components
CPACK_INSTALL_DIRECTORIES
      - Extra directories to install
CPACK_PACKAGE_DESCRIPTION_FILE
      - Description file for the package
CPACK_PACKAGE_DESCRIPTION_SUMMARY
      - Summary of the package
CPACK_PACKAGE_EXECUTABLES
      - List of pairs of executables and labels
CPACK_PACKAGE_FILE_NAME
      - Name of the package generated
CPACK_PACKAGE_ICON
      - Icon used for the package
CPACK_PACKAGE_INSTALL_DIRECTORY
      - Name of directory for the installer
CPACK_PACKAGE_NAME
      - Package project name
CPACK_PACKAGE_VENDOR
      - Package project vendor
CPACK_PACKAGE_VERSION
      - Package project version
CPACK_PACKAGE_VERSION_MAJOR
      - Package project version (major)
CPACK_PACKAGE_VERSION_MINOR
      - Package project version (minor)
CPACK_PACKAGE_VERSION_PATCH
      - Package project version (patch)
```

Some NSIS generator specific variables follow.

```
CPACK_PACKAGE_INSTALL_REGISTRY_KEY
      - Name of the registry key for the installer
CPACK_NSIS_EXTRA_UNINSTALL_COMMANDS
      - Extra commands used during uninstall
CPACK_NSIS_EXTRA_INSTALL_COMMANDS
      - Extra commands used during install
```

FULL EXAMPLE

For this example, we assume we have a shared library, an executable, some header files, and documentation. The CMake list file would look like this.

```
project(WetFox)

add_library(wetfoxlib ...)
add_executable(WetFox ...)
target_link_libraries(WetFox wetfoxlib ...)

install(TARGETS wetfoxlib WetFox
        # .exe, .dll
        RUNTIME DESTINATION bin COMPONENT Runtime
        # .so, .sl, ...
        LIBRARY DESTINATION lib COMPONENT Runtime
        # .a, .lib
        ARCHIVE DESTINATION lib COMPONENT Develop
)
install(FILES
  ${WetFox_SOURCE_DIR}/wetfox.h
  DESTINATION include
  COMPONENT Develop)

set(wf_version_major "0")
set(wf_version_minor "3")
set(wf_version_patch "19")

set(CPACK_PACKAGE_DESCRIPTION_SUMMARY
    "WetFox is an alternative to FireFox")
set(CPACK_PACKAGE_VENDOR "WetWare")
set(CPACK_PACKAGE_DESCRIPTION_FILE
    "${CMAKE_CURRENT_SOURCE_DIR}/ReadMe.txt")
set(CPACK_RESOURCE_FILE_LICENSE
    "${CMAKE_CURRENT_SOURCE_DIR}/Copyright.txt")
set(CPACK_PACKAGE_VERSION_MAJOR
    "${wf_version_major}")
set(CPACK_PACKAGE_VERSION_MINOR
    "${wf_version_minor}")
set(CPACK_PACKAGE_VERSION_PATCH
    "${wf_version_patch}")
set(CPACK_PACKAGE_INSTALL_DIRECTORY
    "WetFox
    ${wf_version_major}.${wf_version_minor}")
set(CPACK_SOURCE_PACKAGE_FILE_NAME
    "wetfox-${wf_version_major}.
     ${wf_version_minor}.${wf_version_patch}")
if(WIN32 AND NOT UNIX)
  # There is a bug in NSIS that does not handle full
  # unix paths properly. Make sure there is at least
  # one set of four (4) backslashes.
  set(CPACK_PACKAGE_ICON
      "${WetFox_SOURCE_DIR}/
       Utilities/Release\\\\WetFox.bmp")
  set(CPACK_NSIS_INSTALLED_ICON_NAME
      "bin\\\\WetFox.exe")
  set(CPACK_NSIS_DISPLAY_NAME
      "${CPACK_PACKAGE_INSTALL_DIRECTORY} a FireFox
       alternative")
  set(CPACK_NSIS_HELP_LINK
      "http:\\\\\\\\www.wetware.com/WetFox")
  set(CPACK_NSIS_URL_INFO_ABOUT
      "http:\\\\\\\\www.wetware.com")
  set(CPACK_NSIS_CONTACT "fox@wetware.com")
else(WIN32 AND NOT UNIX)
  set(CPACK_STRIP_FILES "bin/WetFox")
  set(CPACK_SOURCE_STRIP_FILES "")
endif(WIN32 AND NOT UNIX)
set(CPACK_PACKAGE_EXECUTABLES "WetFox" "Wet Fox")
```

CONCLUSION

CPack is a work in progress, and new features are added on a daily basis. Some improvements for the future include more generators, such as RPM for RedHat and Fedora Linux, as well as Cygwin support. Additionally a scripting mode will

be added to allow automatic packaging of projects. With better uploading and downloading capabilities of CMake, CPack will be able to automatically package and upload the packages to the download directory.

__Andy Cedilnik__ is a project lead in Kitware's Clifton Park, NY, office. Mr. Cedilnik is the principle developer behind CTest and CPack, as well as one of the principle developers behind CMake. He is also a project manager for ParaView Enterprise Edition

ADDING A NEW FILTER TO PARAVIEW 2.6

One of the most powerful features of ParaView is its extensibility. This allows ParaView users and developers to add new functionality to ParaView in the form of readers, sources, and filters. We will demonstrate the process of adding new algorithmic functionality to ParaView as well as the user interface (GUI) to support it. There are three different methods you may use to create a new filter and make it available within ParaView 2.6. You may directly modify the ParaView source code; you may create your new filter outside of ParaView, but then link it into ParaView at compile time; or you may create your new filter outside of ParaView and import it into the application at run-time. This final option is the one we will discuss in this article. We will demonstrate this step-by-step process using a very simple filter, but more complex algorithms can be added to ParaView in the same manner.

The eight steps outlined in this article may be applied for any new filter you wish to add to this application. If the filter you wish to enable in ParaView already exists in VTK, you may skip steps 1, 4, and 5. To demonstrate the required steps when the filter does not already exist in ParaView, we will create a filter to flip the coordinates of a VTK dataset by negating the X-, Y-, or Z-coordinate of each point in the dataset. After applying this filter, the dataset will use left-handed coordinates rather than right-handed ones.

1. WRITE A VTK FILTER

Before we can add a filter to ParaView, we must first write the new VTK filter, a subclass of vtkAlgorithm. For this example, the filter, named vtkFlipCoordsFilter, will operate on and produces as output a dataset with explicit points (i.e., subclasses of vtkPointSet: vtkPolyData, vtkUnstructuredGrid, and vtkStructuredGrid), so the filter will be a subclass of vtkPointSetAlgorithm. The header file for this class is shown below.

```
#ifndef __vtkFlipCoordsFilter_h
#define __vtkFlipCoordsFilter_h

#include "vtkPointSetAlgorithm.h"

class VTK_EXPORT vtkFlipCoordsFilter :
public vtkPointSetAlgorithm
{
public:
  static vtkFlipCoordsFilter* New();
  vtkTypeRevisionMacro(vtkFlipCoordsFilter,
                       vtkPointSetAlgorithm);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Description:
  // Set/get which axis' coordinates should
  // be negated.
  // X = 0, Y = 1, Z = 2.
  vtkSetClampMacro(Axis, int, 0, 2);
  vtkGetMacro(Axis, int);

protected:
  vtkFlipCoordsFilter();
  ~vtkFlipCoordsFilter() {}

  int RequestData(vtkInformation *,
                  vtkInformationVector **,
                  vtkInformationVector *);

  int Axis;

private:
  vtkFlipCoordsFilter(const vtkFlipCoordsFilter&);
  void operator=(const vtkFlipCoordsFilter&);
};

#endif
```

The Axis variable will determine which coordinate of the points will be negated. It can have any of the integer values 0, 1, or 2. For this reason, we use a vtkSetClampMacro to create the SetAxis method. The last two parameters passed to this macro specify the range of the Axis variable. If you try to set the variable to a value outside this range, it will be clamped to this range.

Notice that in the header file for this filter, the constructor and destructor are protected rather than public. VTK objects are created and deleted using New() and Delete() methods rather than calling new and delete directly. This allows VTK to perform certain functions on every vtkObject that is created or deleted.

Also notice the RequestData method. This is the method where most of the work of this filter is done. The VTK pipeline calls this method to request that the filter perform its operation on the data. Shown below is the implementation of the RequestData method.

```
int vtkFlipCoordsFilter::RequestData(
  vtkInformation* vtkNotUsed(request),
  vtkInformationVector** inputVector,
  vtkInformationVector* outputVector)
{
  // get the information objects
  vtkInformation *inInfo =
    inputVector[0]->GetInformationObject(0);
  vtkInformation *outInfo =
    outputVector->GetInformationObject(0);

  // get the input and output
  vtkPointSet *input = vtkPointSet::SafeDownCast(
    inInfo->Get(vtkDataObject::DATA_OBJECT()));
  vtkPointSet *output = vtkPointSet::SafeDownCast(
    outInfo->Get(vtkDataObject::DATA_OBJECT()));

  // Copy the structure of the input dataset
  // to the output.
  output->CopyStructure(input);

  int numPts = input->GetNumberOfPoints();
  vtkPoints *points = input->GetPoints();
  vtkPoints *newPts = vtkPoints::New();
  newPts->SetNumberOfPoints(numPts);

  double pt[3];
  int i;
```

```
  for (i = 0; i < numPts; i++)
    {
    // Negate the specified coordinate.
    points->GetPoint(i, pt);
    pt[this->Axis] *= -1;
    newPts->SetPoint(i, pt);
    }

  output->GetPointData()->CopyNormalsOff();
  output->GetPointData()->PassData(
    input->GetPointData());
  output->GetCellData()->CopyNormalsOff();
  output->GetCellData()->PassData(
    input->GetCellData());

  output->SetPoints(newPts);
  newPts->Delete();

  return 1;
}
```

Early in this method we get the input and output datasets. In VTK, filters operate on input datasets to produce output datasets; the input dataset is not modified. Next we set the number of points in the output dataset to be the same as the number in the input. We also copy the structure of the input dataset since only the point coordinates will be changing. We then loop through the points, negating the specified component of each coordinate. Then we pass any point-centered or cell-centered data from the input dataset to the output dataset, with the exception of normals, since they would now point in the wrong direction.

Once we have completed the required operations for this method, we return a value of 1, indicating that the method completed successfully. A value of 0 may be returned if you need to exit from this method without completing the desired operation. Please see the source code available from http://www.kitware.com/products/newsletter.html for the rest of the implementation details of this filter.

2. WRITE MYCUSTOM.XML.IN

Once the new filter has been written, we must write two XML files in order to make it accessible from ParaView. The first of these files specifies the user interface elements ParaView should use when presenting this filter to a user. In the case of the vtkFlipCoordsFilter, a ParaView user must be able to choose the input to the filter and which portion of the point coordinates to negate to change the handedness of the dataset. We will use an InputMenu to specify the input dataset and a SelectionList for choosing whether to negate the X-, Y-, or Z-coordinate. The contents of the XML file (myCustom.xml.in) for specifying the user interface follow.

```
<ModuleInterface>
 <Library name="@MODULE_NAME@"
          directory="@LIBRARY_OUTPUT_PATH@"/>
 <ServerManagerFile name="@MODULE_NAME@.pvsm"/>

  <!-- Start of ParaView plugin interface
       specification. -->
  <Module name="MyCustom"
          menu_name="Flip Coords"
          root_name="FlipCoords"
          module_type="Filter"
          long_help="Custom filter to demonstrate
                     importing filters to ParaView."
          short_help="Custom imported filter.">
    <Filter class="vtkFlipCoordsFilter">
      <Input name="Input"
             class="vtkPointSet"/>
    </Filter>
    <InputMenu trace_name="Input" label="Input"
          property="Input"
          help="Set the input to this filter."
          input_name="Input"/>
    <SelectionList property="Axis"
          trace_name="Axis"
          label="Axis"
          help="Select which coordinate to negate.">
      <Item name="X"
          value="0"/>
      <Item name="Y"
          value="1"/>
      <Item name="Z"
          value="2"/>
    </SelectionList>
    <Documentation>
      This filter is here to demonstrate how to
      Import filters to ParaView. It can be imported
      during run time or compiled into the code.
    </Documentation>
  </Module>
  <!-- End of ParaView plugin interface
       specification. -->
</ModuleInterface>
```

Each filter added to ParaView contains a starting and ending <Module> </Module> XML tag. In this example, the name of the Module is MyCustom; this is the name that will be referenced by the XML file for the ParaView server; see the next section for details. The menu_name lists the name that that will be used for this filter in ParaView's Filter menu. The root_name is used by ParaView internally to determine a unique name for each instance of this filter in a ParaView session. The module_type specifies that this is a filter, not a source or a reader, etc.

Inside the Module tag is a Filter tag. It specifies the name of the VTK class that implements this filter. It also specifies the type(s) of datasets on which this filter operates.

Following this is an XML tag for each user interface element needed by this filter. Each user interface element has an associated property in the server XML file; the name of that property is listed in the XML tag for that user interface element. Also listed will be a trace_name, used by ParaView to identify the user interface element in its trace files. The label is the text that is shown by that particular element in the user interface for this filter. Additionally, for the SelectionList (a drop-down menu listing options from which the user may choose) we must specify what value to associate with each option in the list. Our SelectionList will show the values "X", "Y", and "Z", but the values 0, 1, and 2 should be passed to the filter.

At the beginning of the user interface XML file are two XML tags after the opening ModuleInterface tag: Library and ServerManagerFile. They contain a series of variables enclosed in "@" symbols. These variables will be filled in during the configuration and compilation process described later in this article. In the Library XML tag, the directory element will point to the location of the library we will build containing the new filter. In the ServerManagerFile tag, the name element lists the name of the server-side XML file.

3. WRITE MYCUSTOM.PVSM.IN

In addition to the user interface XML file, an associated server-side XML file must also be created. The XML tags contained in this file describe proxies (one per source, filter, reader, etc.), properties (one per user interface element), and domains (to specify what values are acceptable for a given

property). The server-side XML file (myCustom.pvsm.in) for the example filter is shown below.

```
<ServerManagerConfiguration>
  <ProxyGroup name="filters">
   <SourceProxy name="MyCustom"
                class="vtkFlipCoordsFilter">
     <InputProperty
        name="Input"
        command="SetInputConnection">
          <ProxyGroupDomain name="groups">
            <Group name="sources"/>
            <Group name="filters"/>
          </ProxyGroupDomain>
          <DataTypeDomain name="input_type">
            <DataType value="vtkPointSet"/>
          </DataTypeDomain>
     </InputProperty>
     <IntVectorProperty
        name="Axis"
        command="SetAxis"
        number_of_elements="1"
        animateable="1"
        default_values="1" >
     </IntVectorProperty>
   <!-- End MyCustom-->
   </SourceProxy>
  </ProxyGroup>
</ServerManagerConfiguration>
```

The SourceProxy in the server-side XML file corresponds to the Module in the user interface XML file. Within the SourceProxy tag, the "name" element must match the "name" element of the Module tag in the user interface XML file; this is how ParaView determines which user interface description matches which server-side description.

Each Property element in the above XML file (InputProperty and IntVectorProperty) specify a name for the property and a command. The property name must match the "property" entry of the corresponding user interface element in the user interface XML file. The command is the name of the VTK method to call to pass values to the VTK filter we have written. The method SetInputConnection (in the InputProperty XML tag) is defined in the vtkAlgorithm class (the direct superclass of vtkPointSetAlgorithm from which vtkFlipCoordsFilter is derived). The SetAxis method mentioned in the IntVectorProperty tag is in the filter we wrote; it is defined by the vtkSetClampMacro.

The InputProperty specifies the input to the filter. It has two domains to specify acceptable values for it. The ProxyGroupDomain says that the input dataset may be the result of any sources (including file readers) or filters in ParaView. The DataTypeDomain limits the type of dataset to vtkPointSet and its subclasses.

The IntVectorProperty sets the value of the Axis variable in vtkFlipCoordsFilter. It is not using any domains. Its default_values element determines the default value ParaView will use when this filter is loaded. (In this case, the default value is 1, the Y-coordinate.) The number_of_elements entry determines the number of parameters that should be passed to the SetAxis method; this method only takes one parameter. The animateable element determines whether this property of this filter may be used in creating a ParaView animation; the "1" value specifies that it may be animated.

Please see *The ParaView Guide* for a more complete description of the XML tags that may be used in ParaView's user interface and server-side XML files.

4. WRITE CMAKELISTS.TXT

In order to compile the source code and configure the XML files we have written, we must write a CMakeLists.txt file that CMake can use to create appropriate Makefiles or project files (depending on the compiler to be used). The CMakeLists.txt file will allow us to create a library containing the new filter. It will also fill in the MODULE_NAME and LIBRARY_OUTPUT_PATH variables in the user interface XML file. The CMakeLists.txt file used is shown below.

```
PROJECT(ParaViewCustomFilter)

FIND_PACKAGE(ParaView REQUIRED)
INCLUDE(${PARAVIEW_USE_FILE})

# Configure output directories.
SET (LIBRARY_OUTPUT_PATH "${PROJECT_BINARY_DIR}"
     CACHE INTERNAL "For libraries.")
SET (EXECUTABLE_OUTPUT_PATH "${PROJECT_BINARY_DIR}"
     CACHE INTERNAL "For executables.")
INCLUDE_DIRECTORIES("${CMAKE_CURRENT_SOURCE_DIR}"
                    "${CMAKE_CURRENT_BINARY_DIR}")

# specify the sources
SET(mySrcs "vtkFlipCoordsFilter.cxx")

# specify the name of the module
SET(MODULE_NAME myCustom)

# Create vtk client/server wrappers for the classes.
VTK_WRAP_ClientServer("${MODULE_NAME}" wrappedSrcs
                      "${mySrcs}")

# Build the package as a plugin for ParaView.
ADD_LIBRARY("${MODULE_NAME}" MODULE ${wrappedSrcs}
            ${mySrcs})
TARGET_LINK_LIBRARIES("${MODULE_NAME}"
                      vtkClientServer
                      vtkPVServerManager)

# Place the package configuration file into the build
tree.
CONFIGURE_FILE(
  ${CMAKE_CURRENT_SOURCE_DIR}/myCustom.xml.in
  ${CMAKE_CURRENT_BINARY_DIR}/myCustom.xml
  @ONLY IMMEDIATE)

# Place the package configuration file into
# the build tree.
CONFIGURE_FILE(
  ${CMAKE_CURRENT_SOURCE_DIR}/myCustom.pvsm.in
  ${CMAKE_CURRENT_BINARY_DIR}/myCustom.pvsm
  @ONLY IMMEDIATE)
```

The first line of the CMakeLists.txt file specifies a name for this project. The following two lines specify that we need to find a version of ParaView on this computer. We then set two path variables and list the "include" directories.

The following line (`SET(mySrcs "vtkFlipCoordsFilter.cxx")`) allows you to specify a space-separated list of C++ files to compile. The ADD_LIBRARY line specifies that these files will be used for creating a library. The TARGET_LINK_LIBRARIES line allows you to specify additional libraries to which the new library needs to link.

The line, `SET(MODULE_NAME myCustom)`, lists the MODULE_NAME that will replace the variables enclosed in @'s in the user interface XML file. The CONFIGURE_FILE lines configure the XML files appropriately and specify that versions of them without the ".in" should be placed in the bin directory for this project when it is compiled.

The VTK_WRAP_ClientServer line expresses that the new C++ classes should be wrapped in ParaView's client/server

language. ParaView uses this language for message passing; a fuller description of this language is outside the scope of this article.

5. COMPILE

Once you have written the CMakeLists.txt file, you can run CMake, and specify the source and binary directories you wish to use. If CMake cannot find ParaView on your system, you must specify it in CMake's interface (ccmake on unix platforms or CMakeSetup on Windows). Once CMake has completed the configuration process, it will have (as described earlier) created appropriate Makefiles or project files for the selected compiler.

Then you can compile the new module as you would any other project. When the compilation finishes, a library file will have been created, and the files myCustom.xml and myCustom.pvsm will be located in the top level of the binary directory.

6. LOAD THE MODULE INTO PARAVIEW

Now that you have compiled the new module, it must be loaded into ParaView in order to be used. There are two methods for accomplishing this. The most straightforward is to first launch ParaView. Then select "Import Package" from the file menu. While this method is the easier of the two options, it must be performed each time ParaView is launched if you wish to use the new filter in that ParaView session.

If you wish to use the new filter often, then a better option is to set the PV_INTERFACE_PATH environment variable. This variable must point to the location of the user interface and serve-side XML files. When ParaView starts, it will attempt to load any modules specified by the XML files found in the specified location.

7. LOAD DATA

In order to apply the filter that has now been imported into ParaView, we must first load some data to which this filter may be applied. For the purposes of this demonstration, we will use ParaView's cone source, but any structured grid, unstructured grid, or polygonal dataset could be used. To use the cone source, select Cone from ParaView's Source menu, and click the Accept button on the Parameters tab.

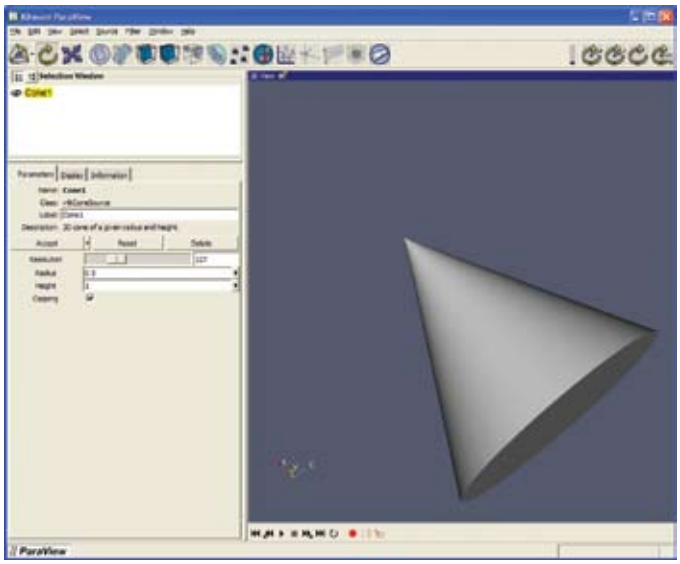


Figure 1: A cone loaded in ParaView 2.6

8. APPLY FILTER

Now that data has been loaded into ParaView, select Flip Coords from the Filter menu. The interface (as specified by the user interface XML file) appears in ParaView's left panel.

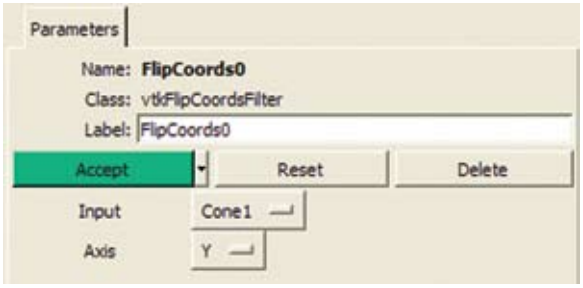


Figure 2: User interface for the vtkFlipCoordsFilter

The Input and selection menus appear as expected. The default value in the Input menu is the Cone1 dataset from the previous step. By default ParaView uses the dataset that was selected when a filter is applied as the input to that filter. If more vtkPointSet datasets were loaded in ParaView when this filter was applied, they would also be listed in the Input menu. The Axis selection menu contains the values X, Y, and Z, as we specified in the user interface XML file. Notice that the default value is Y, as we specified in the server-side XML file. Select X from the Axis menu (so the results will be more readily visible). Click the Accept button to apply the filter to the cone dataset. It will be flipped across the X axis because the X coordinates were negated.

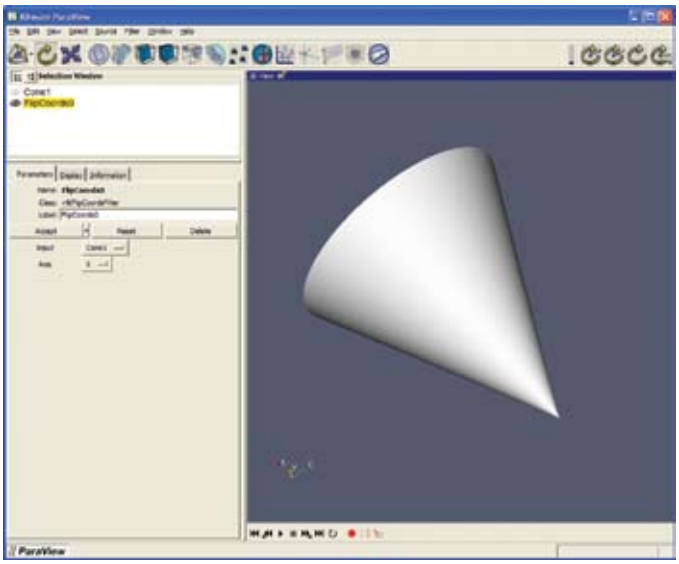


Figure 3: The cone dataset after the vtkFlipCoordsFilter has been applied

***Amy Squillacote** is a technical developer in Kitware's Clifton Park, NY, office. She is a contributor to both VTK and ParaView. She is also the primary author of "The ParaView Guide."*

IN PROGRESS

INSIGHT JOURNAL CONTRIBUTIONS TO ITK

ITK is adopting code from contributions submitted to the *Insight Journal*. In preparation for the release of ITK 3.2, a group of papers from the *Insight Journal* has been selected to be integrated with the toolkit. The source code from these papers is currently being reorganized and refactored in the Code/Review directory of the Insight Toolkit CVS repository. The new code will stay in this directory for one release, and it will be moved to its final location in the toolkit by the following release. The purpose of this mechanism is to perfect the API and coding style of the classes before they officially become part of the toolkit. Once the classes are moved to their final destination, the backward compatibility policy of ITK will prevent developers from making significant API changes to the classes.

The new contributions include a variation of the Mesh class that is intended to represent 2D manifolds with consistent orientation, a filter for mapping 2D manifolds to spheres and planes, a family of projection filters, and an implementation of the marching squares algorithm.

The *Insight Journal* is an Open Access online publication that enforces the verification of reproducibility in papers. Submissions to the *Insight Journal* are expected to be accompanied by the full source code and data that will allow any reader to replicate the work reported by the authors. The journal also supports open, public peer-review and continuous online dialog between authors, reviewers and readers.

UPCOMING PARAVIEW FEATURES

It has been over a year since ParaView 2.4 was released. We have been working hard on the next generation of ParaView (ParaView 3) that has a completely overhauled user interface. Meanwhile, we added several important features to ParaView 2. These will be featured in the upcoming 2.6 release. The highlights from 2.6 include support for parallel volume rendering of uniform rectilinear datasets (vtkImageData), new techniques for volume rendering unstructured grid datasets, support for hardware accelerated offscreen rendering on all supported platforms (using OpenGL framebuffers), improved opacity support (depth-peeling algorithm), several new readers (including Fluent, OpenFOAM, LSDyna, and AcuSolve), ffmpeg support, improved multi-block and AMR support, and a Python client. The beta for 2.6 will be released in January 2007.

We also continue to work on ParaView 3. The monthly development snapshots are posted at http://paraview.org/Wiki/ParaView_III_snapshots. We hope to release a beta version in the second quarter of 2007. Important new features of ParaView 3 include a completely new user interface based on Trolltech's Qt toolkit, support for multiple views, improved support for quantitative data analysis, first class Python support, and undo/redo support.

KITWARE NEWS

ITK PARALLELIZATION

Brigham and Women's Hospital, Surgical Planning Lab, has funded a subcontract to Kitware to optimize ITK for multi-core and multi-processor, shared memory systems. Initially the work will focus on the optimization of ITK's image resampling and non-rigid registration methods, particularly those involving the b-spline transform. The optimized methods and various ITK performance studies will be published in an *Insight Journal* article.

AUTOMATIC SEGMENTATION

The DOD Air Force Research Lab has awarded Kitware a two-year, Phase II SBIR for automated image segmentation and volumetric model editing. The image segmentation methods to be developed will include image-driven and atlas-driven segmentation methods, and combinations thereof. The volumetric model editing work will feature vtk3DWidget technology. A unifying theme of these development efforts will be an emphasis on workflow, i.e., encoding image segmentation and model editing domain knowledge into an intuitive sequence of operations that can be quickly and consistently applied by novice users.

Kitware's Software Developer's Quarterly is published by Kitware, Inc., Clifton Park, New York.

Contributors: Stephen Aylward, Sebastien Barre, Andy Cedilnik, David Cole, Berk Geveci, Bill Hoffman, Luis Ibanez, Will Schroeder, Amy Squillacote and Yumin Yuan.

Design: Melissa Kingman, www.elevationda.com

Editor: Lisa Avila

To contribute to future editions of this publication, please contact the editor at kitware@kitware.com.