Welcome to the first issue of the Kitware Software Developer’s Quarterly newsletter. This publication was created to deliver basic information on recent releases and upcoming changes, as well as detailed technical articles related to Kitware’s open source products. This includes

- The Visualization Toolkit (www.vtk.org)
- ParaView (www.paraview.org)
- The Insight Segmentation and Registration Toolkit (www.itk.org)
- CMake (www.cmake.org)
- KWWidgets (www.kwwidgets.org)

Kitware welcomes contributions to the newsletter from our active developer community. 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.

**TABLE OF CONTENTS**

Welcome .................................................................................. 1
Recent Releases ................................................................. 1
Using Ghost Cells in Parallel Filters ..................................... 3
VTK 5.0 Pipeline Architecture ........................................... 4
An Introduction to KWWidgets ........................................... 7
CTest ..................................................................................... 9
In Progress ........................................................................... 10
Kitware News ........................................................................ 12

**EDITOR’S NOTE**

The past few months have been an active time for open source projects at Kitware, with a release in nearly every project including a major release for VTK and a change in the release model for ITK.

**VTK RELEASE**

VTK 5.0 was released in January 2006. This is a major release with significant changes from the previous version. Highlights include the following:

- The execution pipeline was overhauled, as described in the “VTK 5.0 Pipeline Architecture” article included in this newsletter.
- A higher-order adaptor framework was developed.
- Support for hierarchical data sets was added, including AMR and multi-block data sets.
- With the expiration of the Marching Cubes patent, the Patented kit has been removed and the Marching Cubes classes have been integrated in the Graphics kit.
- New interpolators have been added (including quaternions).
- Volume rendering performance for vtkImageData has been improved with the addition of space leaping.
- Unstructured grid volume rendering has been added.
- Support for animation was added.

**PARAVIEW RELEASE**

ParaView 2.4 was released in November 2005. The latest patch to this release (2.4.4) was in June 2006. New features in ParaView 2.4 include:

- Improved support for multi-block and AMR data sets.
- Movies and screen captures are now rendered off-screen, allowing you to cover or close the application without destroying the captured image.
- Improved handling of empty data.
- Support has been added for ParaView internals scripts.
- The Spy Plot CTH reader has been improved.
- Added support for selecting functions from Plot3D files.
- Preliminary Python wrapping support has been added.

**CMAKE RELEASE**

The latest release of CMake is the 2.4 release. The 2.4 release branch has many new features to support KDE (www.kde.org), which has chosen CMake as its build system. In addition to improvements in the CMake build system, a new tool called CPack has been added to the CMake family. CPack is a tool to create release packages automatically. It is currently in beta and supports the Nullsoft installer on Windows, the Mac OSX installer, and several forms of compressed tar files. In future releases it will support RPMs and other Linux/UNIX distribution systems.
The changes for the CMake 2.4 branch are as follows.

**Changes in CMake 2.4.2:**
- Run symlink command from correct directory for executable versions
- Fix for universal binaries and Xcode depend problem
- Changes to LIST command; see --help-command LIST
- Fix FindQT to be able to use full paths to source files
- Fix CPack ZIP on Windows command line problem
- Find executables with no extension on Windows mingw
- Fix FindQt3 to use QTDIR over path
- Significant speedup in try-compile for nmake
- CPack improvements including tar bzip2
- FindQt4 Windows path fix
- Sunos cc optimize flags are correct
- Fix crash with $() empty variable
- Increase depend speed on Mac OS.
- Install command CONFIGURATIONS option.
- Fix MSVC60, MSVC70, MSVC71, MSVC80 definitions for IDE builds
- Fix for C++ compiler being used for c code in VS IDE

**Changes in CMake 2.4.1:**
- Several ctest and cpack bug fixes
- Many updates and fixes for FindQt4 cmake
- Fix CMAKE_REQUIRED_FLAGS in CheckCXXSourceCompiles. cmake
- Handle running make from a symlinked build tree
- Automatic color output detection for shells building with make
- Kdevelop generator handles CMakeFiles directory better
- Add correct depend information for fluid
- Allow the cache to be saved even if a fatal error occurs
- Fix bug in relative path subdir and add_subdirectory commands
- Support in Visual Studio for two object files with the same name
- Short file names used for library paths in Visual Studio
- Package target only shows up when you have cpack config files
- Use dl and not -ldl for adding in the dynamic library
- Fix check c/cxx source compiles macos to not clobber log files
- Fix nmake version detection of cl and create correct pdb files
- Fix msys bootstrap
- Change color output to be more readable
- Fix Visual Studio 6 library naming

**Changes in CMake 2.4.0:**
- CPack beta
- Visual Studio 2005 win64 support
- Improved install support
- Improved FIND_PROGRAM, FIND_LIBRARY, FIND_PATH, FIND_FILE
- Improved support for finding/using OSX Frameworks
- Multiple output support for custom commands
- Color output in make with vt100 terminals CMAKE_COLOR_MAKEFILE
- Better variables for MSVC MSVC80
- Library path order is preserved
- Fix for text file busy in xcodebuild runs
- Better bundle support on OSX
- ctest -S scripts can run in new process with new environment
- OSX universal binary support

**ITK RELEASE**

ITK 2.6 was released on March 1, 2006. ITK has adopted a Quarterly release model in order to more rapidly deliver new features to the community. This release schedule began with the ITK 2.4 release. Highlights of the 2.6 release include:

- For the first time the Open Access Insight Journal was used as the official mechanism for new classes and algorithms. See the ITK Wiki (http://www.itk.org/Wiki) for a description of this new procedure.
- Four contributions from papers submitted and reviewed in the Insight Journal were added to the toolkit. These contributions are as follows:
  - The Invert Intensity Image Filter is a filter based on a Functor that inverts the intensities in one image. Although this filter is relatively simple, it is significant because it was contributed with a clean coding style by a user that has not participated before as a developer.
  - The Modulus Image Filter is also a filter based on a Functor and computes modulus on a pixel-by-pixel basis.
  - The Morphological Gradient, also called Beucher gradient, is the difference between the dilation and erosion of an image.
  - Threshold Maximum Connected Components is a filter that computes the threshold value that will result in a maximum number of connected components with a size larger than a limit specified by the user. This filter was demonstrated to be useful for cellular images in microscopy.
  - Support for Bayesian Classification was improved.
  - The DICOM libraries were updated to JPEG 2000 and the DICOM reader (GDCM) was updated in order to use this new version of JPEG.

**KWWIDGETS STATUS**

Although KWWidgets has been under development for quite some time, and some of the classes had been released as part of the ParaView distribution, the KWWidget project only recently became its own toolkit with an associated web site (www.kwwidgets.org), mailing list, FAQ, and Wiki. Currently, the development code is available via CVS. Kitware intends to release the first official version of KWWidgets in conjunction with the release of VTK 5.2. Thereafter, KWWidgets releases will coincide with VTK releases.

**TECHNICAL BOOKS UPDATE**

Three of Kitware's technical books have been recently revised. The *ITK Software Guide* (ISBN 1-930934-15-7) has been updated for ITK version 2.4, and is 832 pages. The *ParaView Guide* (ISBN 1-930934-17-3) has been updated for ParaView 2.4, and is 384 pages. *Mastering CMake* (ISBN 1-930934-16-5) has been updated for CMake 2.4, and is 256 pages. These books can be ordered directly through Kitware at www.kitware.com, or from Amazon.com. Kitware is currently revising *The Visualization Toolkit*, and *The VTK User's Guide* for VTK 5.0. These titles will be available Summer 2006.
Using Ghost Cells in Parallel Filters

Ghost cells are an extremely important concept when processing data with VTK or ParaView in parallel. In this article we will discuss what ghost cells are, how they are used, and why they are important.

Parallel Filters

To understand the function of ghost cells, we must first understand how most VTK filters operate in parallel. We do this by example. Consider the simple example mesh below that we have split into three partitions and distributed amongst three processes of a parallel job.

Although they process data in parallel on any machine, including shared memory machines, VTK and ParaView are designed to perform efficiently on distributed memory machines. This means that filters do not share data and take great care to minimize communication between processes.

Ideally, each process could perform a filtering operation only on the local data without any need for communication or global information. For some filters, this is possible. Take for instance the clip filter (vtkClipDataSet). Below we show our example data set clipped by a plane. The filter is replicated on each process and given the same parameters for the clipping plane. We see that even though each clip took into account only local information, when we bring the pieces back together the result is equivalent to a global clipping operation.

Unfortunately, this naïve approach does not work with most filters. For a simple example, consider an extract external faces filter (vtkDataSetSurfaceFilter).

We see that all of the external faces are correctly identified. However, many faces in the interior of the data are incorrectly extracted. The false positives occur whenever a cell has a neighbor that happens to be located on a remote process. Because no information about the neighbor is known, the filter has no way of knowing that the adjoining face is not external.

Ghost Cells

We can solve the aforementioned problem with the extract external faces filter (and similar problems with many other filters) by introducing ghost cells. Ghost cells are cells that belong to one process but are replicated on other processes. On all but the owning process, a ghost cell is marked with a special ghost cell value that alerts filters that, although the data is available, the location of the cell is considered to be on another process. If we judiciously add ghost cells to our example, we solve our problems with only a small amount of data duplication and without introducing any interprocess communication.

Although the external faces filter still incorrectly identifies some faces as external, all of these false positives are attached to ghost cells. They can therefore easily be culled from the final result, leaving us with the correct set of external faces.

Ghost Cell Levels

Ghost cells are defined by levels. Each process has its own definition of ghost cell levels. Any cell on ghost level 1 or higher is a ghost cell that is owned by some other processes, whereas any cell on ghost level 0 belongs to the process’s partition of cells.

Ghost level 0 is defined implicitly by the cells in the process’s partition of cells. Ghost level 1 comprises all cells neighboring a cell in ghost level 0 but not belonging to ghost level 0. Further levels are defined recursively. Ghost level n + 1 comprises all cells neighboring a cell in ghost level n but not belonging to any cell in levels 0 through n.

Ghost levels provide a convenient way of specifying the cells to replicate as ghost cells. The ghost level gives a rudimentary distance of a cell to the partition. Filtering operations are usually only concerned with cells nearby the local ones and can get a near minimal set of ghost cells simply by specifying the maximum level they need. In the external faces example, the set of ghost cells needed are those at level 1.

Creating Ghost Cells

Whenever possible, VTK takes care of creating ghost cells for you. A filter that requires ghost cells can pass this information up the pipeline. Upstream filters use this information to create and label ghost cells in their outputs.

Many sources and filters will generate ghost cells upon request. However, determining ghost cells may be a global operation and can require much computation and communication. This is particularly true for unstructured data types (vtkPolyData and vtkUnstructuredGrid), and ghost cells may be missing for these types of data.

As a VTK or ParaView user, your primary concern with ghost cells is making sure that they can be created when needed. Ghost cells may be reconstructed in poly data and unstructured grids with the D3 filter. You can use D3 in VTK...
by adding a `vtkDistributedDataFilter` to your pipeline. You can use D3 in ParaView by selecting the D3 filter.

The images below demonstrate how D3 can be used to correct the operation of a filter. The left image shows a parallel external face extraction without ghost cells whereas the right image uses D3 to create ghost cells.

**ACKNOWLEDGEMENTS**

This work was done in part at Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

*Kenneth Moreland* performs research and development at Sandia National Laboratories in Albuquerque, NM. Dr. Moreland contributes to many of ParaView’s parallel processing algorithms including mesh distribution, parallel rendering, and tile display. He also designs his own head gear.

**VTK 5.0 PIPELINE ARCHITECTURE**

VTK 5.0 introduces a generalized pipeline architecture designed to reduce complexity while increasing flexibility. Pipeline mechanics are separated from algorithm implementation and data representation. The design offers three main features. First, algorithm authors may focus on implementing isolated data processing tasks without worrying about the surrounding pipeline structure. A filter provides an interface specifying requirements for input data and will not be executed unless these requirements are met. Second, pipeline authors may choose the pipeline execution model and the set of algorithms independently. Arbitrary information may be passed transparently up and down the pipeline without knowledge of intermediate filters. This enables the use of alternative pipeline execution models without modifying core classes. Finally, pipelines may be connected before data objects are allocated allowing use of sources that do not know their output data type before executing.

**PIPELINE LAYOUT**

VTK pipelines are constructed from instances of four basic classes. A `vtkAlgorithm` is the basic worker unit which processes data. Each algorithm is paired with a `vtkExecutive` which is responsible for managing pipeline execution. A `vtkInformation` object is a heterogeneous key-value map that holds arbitrary information associated with the inputs or outputs of an algorithm. Data produced by an algorithm are represented by a `vtkDataObject` stored as part of the algorithm’s output information.

The following figure depicts the objects involved in representing a portion of a pipeline. Two algorithms adjacent in the pipeline are shown with a connection between them. A producer on the left provides input to a consumer on the right. In a real pipeline the left algorithm may also consume input from its left and the right algorithm may also produce input to its right.

A `vtkAlgorithm` defines zero or more input ports and zero or more output ports through which it obtains input and sends output. These ports are logical concepts defining the interface of an algorithm and have several objects associated with them. Each input port corresponds to a specific algorithmic purpose and each output port holds one output data object produced by the algorithm. For example, a glyph filter may define one input port from which it obtains geometry describing glyph placement, one input port from which it obtains the glyph itself, and one output port in which it stores the generated glyphs.

Pipelines are formed by connecting the output ports of some algorithms to the input ports of others. This is done using `vtkAlgorithm`’s `SetInputConnection` and `GetOutputPort` methods. For example, the code

```cpp
vtkSmartPointer<vtkImageGridSource> grid = vtkSmartPointer<vtkImageGridSource>::New();
vtkSmartPointer<vtkContourFilter> contour = vtkSmartPointer<vtkContourFilter>::New();
contour->SetInputConnection(0, grid->GetOutputPort(0));
```

creates an image source whose output port 0 is connected to a contour filter on input port 0. Note that the connection is formed before either filter executes or any data are allocated.

A `vtkAlgorithm` owns PORT information consisting of one `vtkInformation` object for each input or output port. Port information is set by the `vtkAlgorithm` when it is created and usually remains unchanged for the lifetime of the object. Input port information specifies the algorithm’s requirements for data provided by that input port. Most input ports require exactly one data object in order for the algorithm to function. Some algorithms may allow optional inputs or multiple data objects on a single input. For example, an append filter defines one input port allowing an arbitrary number of data objects that are all treated equally by the algorithm.
Output port information describes the output generated by the algorithm on that output port.

The port information of a vtkAlgorithm may be obtained using the GetInputPortInformation and GetOutputPortInformation methods. For example, the code

```cpp
contour->GetInputPortInformation(0)->Print(cout);
contour->GetOutputPortInformation(0)->Print(cout);
```

produces the output

```cpp
vtkInformation (0x8052d50)
  Debug: Off
  Modified Time: 103
  Reference Count: 1
  Registered Events: (none)
  INPUT_REQUIRED_DATA_TYPE: vtkDataSet
  PORT_REQUIREDMENTS_FILLED: 1

vtkInformation (0x8052e90)
  Debug: Off
  Modified Time: 105
  Reference Count: 1
  Registered Events: (none)
  DATA_TYPE_NAME: vtkPolyData
  PORT_REQUIREDMENTS_FILLED: 1
```

A vtkExecutive owns PIPELINE information consisting of a vtkInformationVector with one vtkInformation object for each output port of the executive's algorithm. Pipeline information is updated by both executives and algorithms while fulfilling pipeline requests. The output pipeline information may be obtained using the GetOutputInformation method.

For example, the code

```cpp
grid->GetExecutive()
    ->GetOutputInformation(0)->Print(cout);
```

produces the output

```cpp
vtkInformation (0x8058870)
  Debug: Off
  Modified Time: 133
  Reference Count: 2
  Registered Events: (none)
  CONSUMERS:
    vtkStreamingDemandDrivenPipeline(0x80584d0) port 0
  PRODUCER:
    vtkStreamingDemandDrivenPipeline(0x8052c28) port 0
```

Input pipeline information on an input port is actually the output pipeline information from the output ports to which it is connected. An executive has one vtkInformationVector for each input port of its algorithm. The entries of this vector are the output pipeline information objects connected to the input port.

The pipeline information objects themselves store references to their producers, consumers, and data objects. The vtkExecutive::PRODUCER key maps to a pair consisting of a strong pointer to the vtkExecutive instance producing the information and the producer's corresponding output port number. The vtkExecutive::CONSUMERS key maps to a vector of pairs consisting of weak pointers to the vtkExecutive instances consuming the information and the corresponding input port numbers. The vtkDataObject::DATA_OBJECT key maps to a strong pointer to the vtkDataObject instance holding data.

Finally a vtkDataObject stores a strong pointer to the vtkInformation object holding its pipeline information. This is accessible through the GetPipelineInformation method. A vtkDataObject also stores DATA information which is a vtkInformation object holding arbitrary information about the data currently stored in the object. This is accessible through the GetInformation method.

**PIPELINE EXECUTION**

The execution of a VTK pipeline usually begins when a method such as Write or Update is called on a consumer. The algorithm asks its executive to first make sure its inputs are up-to-date and then to run the algorithm. Then the execution model implemented by the executive determines how the pipeline actually executes.

Specific pipeline execution models are defined by subclasses of vtkExecutive. An execution model consists of a set of requests defined by the executives. A "request" is the basic pipeline operation (or "pipeline pass") which generally asks for certain information to be propagated through the pipeline. It is represented by a vtkInformation object that holds a specific key defined by the executive and corresponding to the request. Requests are typically generated by an executive and sent to the executive through vtkExecutive's ProcessRequest method.

The ProcessRequest method of a vtkExecutive is responsible for fulfilling a request or reporting failure. It may pass the request to other executives through connections on its input or output ports. It may also pass the request to its algorithm through vtkAlgorithm's ProcessRequest method. This can be done in any order as necessary to satisfy the particular request given.

The executive vtkDemandDrivenPipeline defines the timestamp-based on-demand execution model. Its subclass vtkStreamingDemandDrivenPipeline adds streaming support and is the default executive given to vtkAlgorithm objects. This defines the traditional VTK pipeline execution model. The following requests are defined by the executive:

- **REQUEST_DATA_OBJECT** asks for the DATA_OBJECT to be created, but not populated, and stored in the output pipeline information.
- **REQUEST_INFORMATION** asks for output pipeline information such as the ORIGIN, SPACING, and WHOLE_EXTENT.
- **REQUEST_UPDATE_EXTENT** asks for the input update extent necessary to produce a given output update extent to be stored in the input pipeline information.
- **REQUEST_DATA** asks for the output DATA_OBJECT to be populated with the actual output data.

All the information the executives need to implement this execution model is stored in the pipeline information objects. For example, the code

```cpp
contour->Update();
grid->GetExecutive()
    ->GetOutputInformation(0)->Print(cout);
```

updates the pipeline by issuing all four requests in order. The output from the code shows all the pipeline information stored:

```cpp
vtkInformation (0x8058870)
  Debug: Off
  Modified Time: 349
  Reference Count: 3
  Registered Events: (none)
  UPDATE_EXTENT_INITIALIZED: 1
  CONSUMERS:
    vtkStreamingDemandDrivenPipeline(0x80584d0) port 0
  WHOLE_EXTENT: 0 255 0 255 0 0
```
The DATA_OBJECT key references a vtkImageData instance which holds the real data. This data object holds only its geometry and data arrays and knows nothing about the pipeline structure used to produce it. The ORIGIN and SPACING keys are left from the REQUEST_INFORMATION pass. During the REQUEST_DATA pass they were copied to the data object's data information.

### IMPLEMENTING AN ALGORITHM

Algorithms in VTK are implemented as direct or indirect subclasses of vtkAlgorithm. An algorithm may implement several methods to define its interface and respond to requests. Typically the constructor may set the number of input and output ports defined in its interface:

```cpp
tvtkMyAlgorithm::vtkMyAlgorithm()
{
    this->SetNumberOfInputPorts(1);
    this->SetNumberOfOutputPorts(1);
}
```

The input and output port information is defined by implementing the FillInputPortInformation and FillOutputPortInformation methods respectively:

```cpp
int vtkMyAlgorithm::FillInputPortInformation(  
    int /*port*/,
    vtkInformation* info)
{
    // All input ports consume polygonal data.
    info->Set(  
        (vtkAlgorithm::INPUT_REQUIRED_DATA_TYPE()),
        "vtkPolyData");
    return 1;
}
```

```cpp
int vtkMyAlgorithm::FillOutputPortInformation(  
    int /*port*/,
    vtkInformation* info)
{
    // All output ports produce polygonal data.
    info->Set(vtkDataObject::DATA_TYPE_NAME(),
              "vtkPolyData");
    return 1;
}
```

The entry point for algorithm execution is the ProcessRequest method:

```cpp
int vtkMyAlgorithm::ProcessRequest(  
    vtkInformation* request,
    vtkInformationVector** inputVector,
    vtkInformationVector* outputVector)
{
    // Look for requests we implement.
    if(request->Has(  
        vtkDemandDrivenPipeline::REQUEST_DATA()))
    {
        // Found request for data to be processed.
        // Get pointers to the input and output data
        // objects from the first connection on the
        // first input port and from the first
        // output port.
        vtkInformation* inInfo =...
```
ACKNOWLEDGEMENTS
This work was funded by the US Department of Energy ASCI Views program as part of a multi-year contract awarded to Kitware, Inc. by a consortium of three National Labs - Los Alamos, Sandia, and Lawrence Livermore.

Brad King is a technical developer in Kitware’s Clifton Park, NY office. He led the final design and implementation of the VTK 5.0 pipeline architecture.

AN INTRODUCTION TO KWWidgets

KWWidgets is a free, cross-platform, open-license GUI Toolkit. Over one hundred C++ classes have been developed for more than 7 years by Kitware, Inc. to create open-source and commercial end-user applications like ParaView, VolView, DataFusion, and VisualJournal. The KWWidgets toolkit is also used by the National Alliance for Medical Image Computing (NAMIC) for the 3D Slicer project.

KWWidgets is an object-oriented C++ layer on top of the Tcl/Tk UI toolkit. It uses the same coding framework and guidelines as VTK. It can interact and co-exist with Tcl/Tk directly from C++, allowing you to load or execute Tcl modules directly from a C++ application. As VTK, it is also wrapped automatically into a Tcl package, and therefore can be used directly from Tcl/Tk, allowing you to fast-prototype applications in Tcl, create small KWWidgets modules that can be used from a Tcl/Tk application, or quickly write testing or demo scripts.

CORE WIDGETS

Like many other GUI toolkits, KWWidgets provides low-level core widgets like buttons, entries, scales, menus, combo-boxes, thumbwheels, spin-boxes, trees, notebooks and multi-column lists to name a few. Each widget is a C++ class, with parameters that can be set to control the basic properties of the widget such as the label on a button or the options in a menu. Each widget has a set of callback commands that an application can set to be notified of state changes. For example, a scale widget will invoke one callback command when the user starts interaction, another while the user is moving the slider, and a third when the user releases the widget. This allows an application to, for example, switch from high quality rendering to interactive rendering when the scale value is linked directly to the visualization result.

The widgets also provide callback functionality using VTK’s event/observer model. For example, the scale widget will emit a ScaleValueChangedEvent. This allows multiple objects to listen to the same scale widget, whereas the callback methods are a one-to-one relationship between the scale widget and the object to be called by the callback command. For this reason, many people prefer to use the event/observer mechanism. See the on-line documentation for the vtkCommand class at http://www.vtk.org/doc/nightly/html/classvtkCommand.html for more information about using this mechanism.

COMPOSITE WIDGETS

Like many other GUI toolkits, KWWidgets also provides advanced composite widgets like toolbars, tooltips, progress gauges, split-frames, splash-screens, 2D/3D extents, color pickers, histograms, windows and dialogs. These are higher-level building blocks that can be used by a developer to quickly create a cross-platform application. Although these composite widgets are more complex internally to the KWWidgets toolkit, from the point of view of a software developer they are as easy to use as the basic core widgets.

VISUALIZATION WIDGETS

The true power of this GUI toolkit lies in the visualization widgets. KWWidgets builds upon the set of core and composite widgets to interface to the VTK visualization library and offer high-level visualization-oriented widgets out-of-the-box. Some examples of these visualization widgets include surface material editors, simple animation generators, transfer function editors, annotation editors, window/level and volume property preset editors, text property editors, and rendering widgets. Note that some of these visualization widgets are complex interfaces for setting visualization parameters such as material properties and transfer functions, while others integrate VTK to display data. This is actually the very same code-base that we use and expand daily to create open-source and commercial applications at Kitware.
The code below shows how to create a simple application, a toplevel window, and a checkbutton inside that window. A complex 3D render widget could be created just as easily instead of a button:

```cpp
tvtkKWApplication *app= vtkKWApplication::New();
tvtkKWTopLevel *top = vtkKWTopLevel::New();
top->SetApplication(app);
top->Create();
tvtkKWCheckButton *cb = vtkKWCheckButton::New();
cb->SetParent(top);
cb->Create();
cb->SetText("A checkbutton");
cb->Deselect();
```

**C++ Example**

```tcl
package require kwwidgets
vtkKWApplication app
vtkKWTopLevel top
top SetApplication app
top Create
tvtkKWCheckButton cb
cb SetParent top
cb Create
cb SetText "A checkbutton"
cb Deselect
```

**Tcl Example**

Kitware utilizes KWWidgets in both our open-source projects such as ParaView, as well as our commercial products such as VolView and vjEditor. Although these three applications were derived from the same code base, they each have a unique look that is the result of tailoring a visualization application to a specific market. However, with a common code base it is easy to share newly developed widgets across different applications. For example, the transfer function editor developed for VolView to allow radiologists to adjust the viewing parameters of medical images is also used in the ParaView application to allow scientists to control properties of their tetrahedral meshes produced by fluid dynamics simulations.

KWWidgets can be compiled on a large number of platforms, including Windows, Mac, and many flavors of Unix/Linux systems. It is open-source and uses the same BSD-style open-license as VTK. It uses CTest (part of CMake) to build and test the source tree automatically on a nightly basis and to display its status on Kitware’s public dashboard, ensuring the same quality standards as all other Kitware products.
ACKNOWLEDGEMENTS

This work is part of the National Alliance for Medical Image Computing (http://na-mic.org) (NA-MIC), funded by the National Institutes of Health through the NIH Roadmap for Medical Research, Grant U54 EB005149. This work was also funded by Los Alamos National Laboratory and Sandia National Laboratories, and by the National Science Foundation under SBIR Phase II Award DMI-0339345.

Sebastien Barré is a technical developer in Kitware’s Clifton Park, NY office. Dr. Barré is the main architect of the KWWidgets toolkit, which is used extensively in many of Kitware’s products including ParaView and VolView.

TESTING SOFTWARE WITH CTEST AND DART

Even when adequate time is allocated for the design and implementation phases, most software will contain “bugs” or “flaws”. Sometimes the problem will lie in the design; sometimes the bug is in the implementation. This issue is even more pronounced in large, distributed software projects that develop using a rapid cycle of design / implement / revise, as is the case in many open-source projects.

To mitigate the problem of software quality, software engineers typically use one of many approaches. Examples of these approaches are: waiting for a bug to show-up and fixing it, regression testing, and rigorous testing of subroutines before they are added to the rest of the code.

Each approach has its good points and bad points. They also have costs associated with them. In Kitware’s software projects, such as the Insight Toolkit and ParaView, the approach is a combination of the previously mentioned approaches. It is based on nightly regression testing with results being reported to a centralized location for everybody to examine. In order to achieve this, two tools are used. The systems performing the testing use a tool called CTest, which is distributed with Kitware’s build tool called CMake. The server that is gathering results is called Dart and is being developed and maintained by GE. Client systems running CTest then run tests every night and submit the results to the Dart testing dashboard server. The Dart server will store the history and present results to the users.

CTest performs several tasks in the process of testing a project. It retrieves the latest copy of the source code from the CVS or Subversion revision control system. Once the latest copy is retrieved, it configures the code based on the user-specified properties. The next stage is to build the project. In this stage the appropriate build tool is invoked to perform the actual build. At this point, there are several optional stages that can be performed. The most common one is the stage in which a set of tests is executed to determine if the project does what it should. The second one is to perform the same tests, running them through a memory checking program, such as Valgrind or Purify. This stage will evaluate if there are any memory-related problems that could cause the program to crash. Another stage is coverage checking. This stage examines the code to determine how much of it was actually executed during testing. The final stage is to package the testing results of all other stages and send them to the Dart server.

CTest currently supports three types of tests: pass/fail tests, valid output tests, and image difference tests. All three types can be used at the same time and even within the same test. The only real difference is how CTest determines whether a test passes or fails. The simplest one is the pass/fail test, in which the return code of the test is examined; if the code is 0, then the test passes; otherwise the test fails. The valid output test is similar, but instead of a return code, the output of the test is examined. Here, there are two options. One is to require certain output being present in order for the test to pass, while the other option is to prevent certain output from being present. In an image difference test, the test will produce an image as a part of testing and compare it with the valid image(s). If the images are similar within some tolerance level, then the test will pass. If the images are different, then the test will fail, producing a report with the difference image as seen in Figure 2.

Figure 1: Sample dashboard

CTest currently supports three types of tests: pass/fail tests, valid output tests, and image difference tests. All three types can be used at the same time and even within the same test. The only real difference is how CTest determines whether a test passes or fails. The simplest one is the pass/fail test, in which the return code of the test is examined; if the code is 0, then the test passes; otherwise the test fails. The valid output test is similar, but instead of a return code, the output of the test is examined. Here, there are two options. One is to require certain output being present in order for the test to pass, while the other option is to prevent certain output from being present. In an image difference test, the test will produce an image as a part of testing and compare it with the valid image(s). If the images are similar within some tolerance level, then the test will pass. If the images are different, then the test will fail, producing a report with the difference image as seen in Figure 2.

Figure 2: Image difference test output.
the optional build lines that are or are not considered as warnings or errors, optional commands that are run before or after testing, tests that are excluded from the testing or memory checking, files that are excluded from coverage testing, etc. There are also several customizations that can be performed on the client system that will run CTest. These customizations include the name of the build system and the build description (used for identifying on the dashboard), the type of dashboard it is submitting (Nightly, Experimental, or Continuous), and any other system-specific information.

Unfortunately there is no way to remove all the problems with any given software. We can only try to solve enough of them to make sure the software can be stable and correct within a given environment. Using tools like CTest and Dart, we can automate this process, have a documented history of problems and flaws, and make sure all the problems are solved and stay solved.

FURTHER INFORMATION
Additional information on CTest can be found on the CMake Wiki. The CTest FAQ, including information on obtaining CTest, reporting bugs, and requesting new features, can be found at: http://www.cmake.org/Wiki/CTest:FAQ

Two short tutorials for CTest are also provided on the Wiki. These tutorials will help you enable testing within CMake and add, build, and run new tests. You will also learn how to prepare a nightly, continuous, and experimental dashboard for submission, and you will find details on advanced CTest features including dynamic memory analysis and customization. These tutorials can be found at: http://www.cmake.org/Wiki/CMake_Testing_With_CTest http://www.cmake.org/Wiki/CMake_Scripting_Of_CTest


ACKNOWLEDGEMENTS
This work has been funded in part by the National Library of Medicine, National Institutes of Health contract number N01-LM-9-3531, N01-LM-9-3532, N01-LM-0-3501, N01-LM-0-3502, N01-LM-0-3503, and N01-LM-0-3504. This work has also been funded by the Department of Energy SBIR Phase II Award DE-FG02-03ER83692, by the National Institutes of Health through the NIH Roadmap for Medical Research Grant U54 EB005149, and by the US Department of Energy ASCI Views program through Los Alamos National Laboratory, Sandia National Laboratories, and Lawrence Livermore National Laboratory.

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.

IN PROGRESS
3D WIDGETS IN VTK
The 3D interactive widgets in VTK are undergoing a major redesign to separate the interaction portion of the widget from the underlying representation. This redesign will facilitate customized widget subclasses that tailor the appearance and behavior of the widget for a particular application. In addition, this redesign will benefit the ParaView development effort and other client/server applications where the mouse interaction occurs on the client, but a server is responsible for the rendering of the widget.

The 3D widgets are graphical representations that can be placed in a 2D or 3D scene and provide methods for user interaction. The 3D widgets are subclasses of vtkInteractorObserver, similar to vtkInteractorStyle, and therefore map mouse and keyboard events into VTK actions. In the redesign, the concept of a “widget” was separated from the concept of a “representation”, where the widget is responsible for mapping the interaction events into actions, and the representation is responsible for processing the actions, maintaining the state of the widget geometry, and rendering the geometry. All representations are subclasses of vtkWidgetRepresentation, which itself is a subclass of vtkProp.

As an example, consider vtkContourWidget. The vtkContourWidget can be used by an application to allow the user to define a contour in a 3D scene. The contour widget turns the interaction events such as mouse clicks into actions such as AddNode, which are passed to the contour representation along with the 2D (X,Y) mouse location. The specific contour representation is responsible for converting the 2D mouse location into a 3D node location or perhaps determining that it would be invalid to add a node at this location. Currently, two specific vtkContourRepresentation subclasses exist: one that will add the node on the focal plane, and a second that will add the node on a user-defined bounded plane. It is easy to imagine creating a subclass that will add nodes on the surface of any arbitrary geometric shape or on the first non-transparent voxel encountered in a volumetric object.

Another improvement to the widget design includes event translation, which enables applications to override the default mouse and keyboard events. For example, applications can be readily programmed to use “right-mouse-down” rather than “left-mouse-down” to trigger specified widget behavior. For more information, you may wish to visit the VTK Wiki at http://www.vtk.org/wiki/VTKWidgets.

Besides the vtkContourWidget described above, many other new widgets have been added including widgets for:
• measuring (vtkDistanceWidget, vtkAngleWidget, vtkBiDimensionalWidget),
• managing text (vtkTextWidget, vtkCaptionWidget, vtkBalloonWidget),
• creating keyframe sequences for the camera (vtkCameraWidget),
• controlling playback of animations (vtkPlaybackWidget),
• placing seeds in segmentation (vtkSeedWidget),
• comparing two images (vtkCheckerboardWidget, vtkRectilinearWipeWidget),
• setting parameter values (vtkSliderWidget),
• controlling point position (vtkHandleWidget), and
• performing 2D transformations (vtkAffineWidget).
Many of these widgets have 2D and 3D versions of their representations, so there is even more capability than what might first appear from the list above.

The new 3D widgets are in the VTK source code available through cvs and will be released with VTK 5.2. Please realize that these widgets are under active development and will stabilize once VTK 5.2 is released.

**VTK SHADERS**

In recent years there has been a growing interest in GPU programming. High-level languages such as Cg and GLSL are routinely used to write shaders that harness the GPU capabilities. The VTK rendering process is being modified in order to allow VTK programmers to use vertex and fragment shaders. VTK supports GLSL as well as Cg shaders.

Shaders can be used to improve the appearance of the rendered objects. Techniques such as multi-texturing and complex illumination models can be used to make the visualizations appear realistic. Alternatively, shaders can be used to offset CPU intensive tasks such as volume rendering and image processing to the GPU. Due to the parallelism provided by modern GPUs, it is often possible to perform compute-intensive tasks such as image processing more efficiently on the GPU.

The shader functionality is available now using the cvs version of VTK and is scheduled to be released with VTK 5.2. This functionality is also available in ParaView 2.4 and in the cvs version. To access this feature in ParaView, you must build ParaView from source and turn on the VTK_USE_CG_SHADERS and/or the VTK_USE_GLSL_SHADERS CMake variables.

**CPACK**

Software distribution and dissemination is an important aspect of the software development cycle. It involves packaging the software, distributing it, and updating it. Each operating system provides some form of package management scheme that solves the distribution issues in some operating system dependent way. The end result of this platform-specific way of generating packages is that the project maintainers have to consider every distribution method while designing the project. CPack is a tool developed as a part of the CMake cross-platform development suite that includes CMake and CTest. The goal of CPack is to abstract the platform-specific package management into a set of simple rules that would generate the package on any system.

One example of platform-specific package management is application management on a RedHat Linux system. RedHat Linux systems use the RedHat Package Management (RPM) tool which packages all the necessary files to be installed as well as the scripts, programs, and auxiliary data used for installation itself. During installation the files are unpacked to the appropriate location to be run properly once installed.

Unfortunately on a similar system, such as Debian GNU/Linux, even though the files installed are relatively similar and the locations are almost the same, the package management tool on Debian GNU/Linux is Deb, which is entirely different and non-compatible with RPM. This problem becomes even more apparent when looking into an entirely different platform, such as Microsoft Windows or Apple's Mac OSX.

CPack will solve the problem of cross-platform package generation by building on top of the highly successful CMake build tool. It will use platform-specific generators to abstract the operating system's way of generating packages. To facilitate day-to-day software dissemination needs, it will provide some rudimentary tools such as downloading of files. Furthermore, it will provide a scripting mode similar to the one CTest uses to provide automatic periodic software packaging. Using these tools, the software maintainers will be able to build, test, and package their software on a daily basis without the need for human interaction.

The first beta version of CPack is included with CMake 2.4; the 2.4.2 patch release of CMake was made in May 2006.

**PARAVIEW 2.6**

ParaView (http://paraview.org) is an open-source, cross-platform scientific visualization application developed specifically to handle very large data. ParaView runs on standard desktops and workstations, as well as shared-memory and distributed systems. It was created by Kitware in conjunction with Los Alamos National Laboratory (LANL), Sandia National Laboratories (SNL) and the Army Research Laboratory (ARL). The development started in 1999, and the current version (2.4) can be downloaded from http://www.paraview.org/HTML/Download.html.

It has been over 6 months since the last release of ParaView. Although ParaView is usually released quarterly, the next release (2.6) will not arrive until Q4 of 2006. Kitware is working with SNL on ParaView 3.0. ParaView 3.0 will have a myriad of new features including a brand new interface. This interface will be based on the Qt library (http://www.trolltech.com/products/qt) and is designed to be elegant and easy to use. Paraview 3.0 will also have a greatly improved user interface with more intuitive controls.
expanded emphasis on quantitative analysis, with lots of plotting, graphing, and linked views.

Release 2.6 of ParaView will focus on parallel volume rendering and will contain many improvements in this area. In release 2.2, we introduced parallel unstructured volume rendering. Since then, volume rendering support has matured, and the user interface has been improved. New algorithms have been introduced, and performance has improved. Release 2.6 will further improve the unstructured volume rendering support as well as introduce parallel structured (uniformly sampled volume, i.e., vtkImageData) volume rendering.

Kitware has experienced substantial growth over the past two years, leading to the expansion of the Clifton Park office and the addition of an office in Chapel Hill, North Carolina.

CLIFTON PARK EXPANSION
Kitware, Inc. has expanded into an additional 5,000 square feet of office space in the 28 Corporate Drive office building located in Clifton Park, New York. This expansion will allow Kitware to grow to approximately 50 employees in their Clifton Park offices. Currently, Kitware has 32 employees in Clifton Park and is actively hiring motivated individuals with expertise in software development, computer vision, and scientific visualization.

NORTH CAROLINA OFFICE
Kitware, Inc. is pleased to announce that it has established a Chapel Hill, North Carolina office. The Chapel Hill office currently has three employees and is led by Dr. Stephen Aylward, Chief Medical Scientist. The role of the Chapel Hill office will be to extend the company’s ties to the medical research community and to support current efforts in medical image processing.

Prior to establishing Kitware’s Chapel Hill office, Dr. Stephen Aylward was director of the CADDLab as a tenured Associate Professor in the Department of Radiology at UNC. The CADDLab is the Image Processing Core Facility for the School of Medicine and a component of the Small Animal Imaging Core for UNC’s Lineberger Comprehensive Cancer Research Center. Additionally, Dr. Aylward serves as President of the Insight Software Consortium and as an Associate Editor of IEEE Transactions on Medical Imaging.

Under Dr. Aylward’s direction, the CADDLab has a history of successful collaborations with Kitware. The CADDLab has worked with Kitware on the development of the NLM’s Insight Toolkit (ITK), a histological atlas application, and Georgetown University’s Image Guided Surgery Toolkit (IGSTK). The CADDLab also worked closely with Kitware to co-develop the Insight Journal.

EMPLOYMENT OPPORTUNITIES
Kitware is seeking talented software professionals with experience in medical image analysis, image processing, 3D graphics, graphical user interface, visualization, computer vision, and/or software engineering. Candidates must show initiative, flexibility, and the focus necessary to deliver quality software (both open-source and proprietary code). Applicants must have experience in C++ and demonstrated software development skills.

Qualified candidates will work with leaders in the field on cutting edge problems. Kitware offers significant growth opportunities, an annual bonus, six weeks total paid time off, health and dental benefits, and a 401(k) plan with company contributions. Kitware is an equal opportunity employer.

Send your resume to kitware@kitware.com with “Kitware Employment” as the subject line. Please include a plain text cover letter in the body of the email.