Customizing ParaView

Utkarsh A. Ayachit* David E. DeMarle†

Kitware Inc.

ABSTRACT

ParaView is an Open Source Visualization Application that scales, via data parallel processing, to massive problem sizes. The nature of Open Source software means that ParaView has always been well suited to customization. However having access to the source code does not imply that it is trivial to extend or reuse the application. An ongoing goal of the ParaView project is to make the code simple to extend, customize, and change arbitrarily.

In this paper we present the ways by which the application can be modified to date. We then describe in more detail the latest efforts to simplify the task of reusing the most visible part of ParaView, the client GUI application. These efforts include a new CMake configuration macro that makes it simple to assemble the major functional components of a new application, and then describe the Reaction and Behavior abstractions which help to make the Qt level application code modular enough so that the existing widgets can be easily reused.

Index Terms: D.2.13 [Software Engineering]: Reusable Software—Reusable libraries I.3.8 [Computing Methodologies]: Computer Graphics—Applications

1 INTRODUCTION

ParaView [7] is an open source application which is built upon VTK, the visualization toolkit [6]. ParaView has three fundamental purposes. One purpose is to use parallel processing to scale in terms of data size so that if a given problem is too large to be analyzed on a single workstation, it will be possible to analyze it directly by connecting to a larger parallel computer. The second purpose is to be a front end to VTK. This means that users of ParaView do not need to br programmers or even to understand scientific visualization techniques in order to perform useful analysis of complex scientific data. The third goal is to function as a reusable library so that, like VTK, it may be used to create or become a part of larger works that the initial designers could not anticipate. The first two goals both increase the size and complexity of the application's software, and therefore are counterproductive to the third.

2 ARCHITECTURE

ParaView's solution to the problem of parallel processing is to make use of the Proxy pattern [5]. That is, a family of proxy classes, each of which is populated by parsing XML files, are instantiated at run time to control instances of VTK classes. The objects controlled through the proxies may live in the same process as the proxies themselves or may be instantiated on remote machines. The objects may exist as single instances or may be replicated on many machines to form a data parallel unit. Proxies provide a means to create parallel configuration insensitive VTK pipelines.

ParaView's extensive use of proxies adds a level of indirection to the software. This level of indirection is referred to as the ServerManager. The ServerManager is the initial layer of complexity that VTK developers must overcome in order to extend ParaView. Fortunately the ServerManager includes both Module [2] and server side Plugin [3] facilities. The two facilities differ in that Modules are statically linked into ParaView at compile time, whereas Plugins are dynamically linked into ParaView at run time. Either facility standardizes, through CMake[6] configuration macros, the process by which new types of VTK objects can be added to ParaView. Plugin and Module macros turn the build environment definition into a simple procedure call. A standard list of arguments is populated with information such as the names of the C++ files for the new routines, and the macro creates the correct platform independent build environment to make a library that is compatible with the application. An elementary Plugin macro, which adds a new VTK class to ParaView's ServerManager follows.

```
ADD_PARAVIEW_PLUGIN(
  ANewFilter  1.0 #<-- Name and version
  #C++ source for the VTK class
  SERVER_MANAGER_SOURCES vtkNewFilter.cxx
  #file that tells ParaView how to use it
  SERVER_MANAGER_XML NewFilter.xml
)
```

It is possible to create new applications by working directly at the ServerManager layer. Unfortunately, developing a full featured application of the same scale as the ParaView client is a non trivial task, requiring several man years of effort.

The ParaView client is an application that provides interfaces (either Qt GUI based or python scripted) that an end user can use to create VTK pipelines and perform visualization. The existing client is meant to be general. For example, every intermediate output generated along the pipeline is controllable and can be displayed simultaneously. There are many cases where a general application is not beneficial. For example, novice users may benefit from a more limited application with fewer choices and more automatic guidance. Domain specific users may need only a fraction of the filters available in ParaView, may require other special purpose ones, and may require the application to use the terminology (jargon) of the domain.

Fortunately Plugins can also be used to change the standard ParaView client application. There are thirteen (at last count) "client side" Plugin types. As with server side plugins, CMake macros are invoked to simplify compilation of client plugins. Each macro standardizes the definition of the source code implementation for a specific type of change. Examples include adding custom filter property editing and display controlling panels, adding new menus and toolbars, adding startup and shutdown routines, and adding arbitrary dockable control panels. A non trivial example Plugin is that developed for VisTrails [1].

Unfortunately, there are important customizations tasks for which there is no Plugin type. Please note that all of the existing Plugins provide ways to add functionality to ParaView and none provide ways to take functionality away[1]. In order to make a custom client which exposes only a subset of the standard application's features, one is forced to directly modify the standard client's code, or copy from it heavily. This has been done so far in only a handful

*e-mail:utkarsh.ayachit@kitware.com

†e-mail:dave.demarle@kitware.com

[1] this can be done given an intimate knowledge of the client side Qt code

of cases. The problem with doing so is that the standard application is monolithic and too complex for all but experienced ParaView developers to tackle. What is neede, is a code restructuring that breaks apart internal class dependencies, thereby making it possible to build a customized ParaView application from the bottom up. Ideally, this rearchitecture will make it possible to use Qt Designer to visually assemble the application from existing components. The rest of this paper describes our ongoing work towards this aim.

3 APPLICATION DESIGN

3.1 Top Down Application Design

Developers who undertake the task of deriving a Qt application from ParaView are immediately faced with a design choice. That is, should the new application build up a ParaView like application, starting from a minimal set and adding only the desired features, or should it start from the entire existing application and remove the pieces that are not needed.

Despite the obvious lack of elegance implied by copying the more than three thousand lines of code in the Applications/Client directory, to date everyone who has faced the choice has chosen the top down approach. The code in that directory defines the top level of the standard application client, and contains a number of potentially useful utilities. These include features like an application level testing framework to orchestrate parallel regression tests and binary release bundling scripts for example. However, none of the utilities is readily reusable in other applications. For instance, the application startup and shutdown process is actually implemented in pqClientProcessModuleGUIHelper, a class that lies within the ParaView Qt level library. When one wants to change some of these features, one has to not only copy the top level calls, but also to delve into ParaView's Qt level library code to make tailored changes in behavior. The OverView and StreamingParaView experimental applications serve as examples of derivative applications that were written in a top down manner.

3.2 Bottom Up Application Design

When only a handful of the existing features are needed, or when substantial changes are to be made, the obvious approach is to build from the bottom up. One begins by trying to find the minimal set of Qt components that are required to build an application that lets the user contro ParaView's servermanager API A View window, which provides an area to see the result of the visualization pipeline, is certainly necessary. A file browser, that can show and let the user open files on the server's file system is also essential. An Object Inspector panel, which one uses to enable and then control filter parameters also seems necessary. However an investigation of the code shows that the Object Inspector panel will not function without signals that tell it what filter it is supposed to control. Those signals originate in the the Pipeline Browser, in which ParaView displays the layout of the visualization pipeline. The Pipeline Browser itself has dependencies. The dependencies form a knot such that the minimal set of components needed to make a ParaView like application are the complete set of components in the ParaView application.

Furthermore, as these dependencies are untangled, one finds that the slot handling code that responds to the signaling events, for example disabling the selection buttons when a non 3D view are made active, lie deep within the internals of the four thousand line pqMainWindowCore file.

4 SOLUTION

4.1 Branding Macros

To make it easier to put together the set of libraries and c++ files that make up a new application, one will soon use Branding macros. A Branding macro is similar to a Plugin macro. It consists of a callable CMake function that one supplies arguments to which define the set of components that make up an application. Internally the macro glues those components together and creates a build environment that is able to link them all together. For example, supply the filename of an image file to the SPLASH_IMAGE argument and the resulting application will show that image at startup while it parses the required proxy definining XML files. The XML files are themselves defined with the GUI_CONFIGURATION_XML arguments. In the current code, these same choices are made by hardcoded rules buried within the application, library and CMake configuration code.

The complete branding macro for a new application, one that mimics the standard client, follows. Note that this new application will become the standard client and the current monolithic one deprecated or removed by the ParaView 3.8 release.

```
build_paraview_client(paraview_revamped
  TITLE "ParaView (Revamped)"
  ORGANIZATION  "Kitware Inc."
  VERSION_MAJOR 3
  VERSION_MINOR 7
  VERSION_PATCH 1
  SPLASH_IMAGE
    \${CC_SOURCE_DIR}/PVSplashScreen.png
  PVMAIN_WINDOW pqClient2MainWindow
  PVMAIN_WINDOW_INCLUDE
    pqClient2MainWindow.h
  EXTRADEPENDENCIES pqClient2
  GUI_CONFIGURATION_XMLS
    \${CC_SOURCE_DIR}/ParaViewSources.xml
    \${CC_SOURCE_DIR}/ParaViewFilters.xml
    \${CC_SOURCE_DIR}/ParaViewReaders.xml
    \${CC_SOURCE_DIR}/ParaViewWriters.xml
)
```

4.2 Reactions

With the availability of a application specifying macro, what remains is to make sure that the individual components of the UI function in isolation, and yet are able to work in concert, in a predictable manner, when put together. It is also desireable, in order to make more than cosmetic modifications, to be able to change what the coordinated behaviors are.

In Qt, QActions typically correspond to actions that the user can perform on the user interface such a clicking buttons, toolbars etc. Every QAction has an handler connected to it which performs the tasks that need to be done when the user has triggered the action. Additionally, there is logic that determines when the action can be enabled based on the current state of the application. In the current ParaView client, thie action handler and enable state code are hardcoded in various places within the GUI panels.

In the new design we encapsulate the QAction logic within a new ParaView Reaction type. Reactions are independent objects that use ParaView's core functionality to execute users commands. Reaction range widely, from showing the about dialog, to opening datasets, to establishing connections with a remote server, etc.

Reactions are autonomous and complete i.e. they do not have any cross dependencies and make use of the core ParaView functionality. This makes it easier to for custom applications to bring in components from ParaView eg. if a custom application wants to provide support for connecting to a remote server, they simply need to connect the pqServerConnectReaction to the appropriate QAction in the GUI and they get the logic to connect with servers, manage server configuration etc.

```
new pqServerConnectReaction(
      ui.actionConnectToServer);
```

4.3 Behaviors

Every application typically has behaviors ingrained into them e.g. for word processors, when a file is opened, the act of showing the

first page initially is a behavior. An alternative behavior is to show the page that was open when the file was last saved. In ParaView, behaviors include things like creating a 3D visualization window as soon as a connection is established to a server. Another example is the way that the application prompts the user to save the current state when a connection is about to be terminated. Typically, such behavior is coded into the internal logic of the application, which makes it hard to override standard behaviors. Hence, we formalized behaviors in ParaView. ParaView Behavoirs are classes that use the core to implement such application level logic. Any custom application can now pick and choose the behaviors it wants by simply instantiating the appropriate behavior classes. Furthermore custom applications can subclass from thee provided classes to fine tune the way the user interacts with the application.

With Reactions and Behaviors, we have been able to remove the direct signal/slot interdependencies between the major GUI components, such as the Object Inspector, Pipeline Browser, and Selection Manager. Now all of these talk to a global pqApplicationCore singleton, from which Reactions and Behaviors observe and coordinate without direct interaction. The code in the singleton is abstract, and relies on swappable Reaction and Behavior classes for enforcement.

5 Trying It

The code being developed today is publicly available. To try it, install the git [8] source code control system on your development machine. Then clone the repository at git://github.com/utkarshayachit/ParaView.git to obtain the modified ParaView source. The branding code is on the Branding branch, and a new reference application, called paraview_revamed will be found in Applications/Client2. Once downloaded, the process for building is the same as for building normal paraview [4].

5.1 Legal Considerations

ParaView has a BSD license and, from ParaView release 2.8 on, the Qt library that ParaView's GUI is built upon has an LGPL license. In practice this means that ParaView can be freely used in commercial and non-commercial settings, provided only that the ParaView copyright be provided with the derived work.

6 Conclusion

ParaView 3.6, with its Plugin and Module facilities made it feasible for people to add to the standard ParaView client application. Due to the monolithic code structure of ParaView's Qt code layer, it requires a great deal of knowledge of the ParaView source code to create derived works, especially when those works reuse only a portion of the ParaView User Interface. A new restructuring of the Qt Code layer, specifically the introduction and implementation of Reaction and Behavior abstractions, and the introduction of a new application building CMake macro for ParaView, will rectify this in the next ParaView release. These changes should significantly lower the experience barrier needed to develop new ParaView based applications. It is our hope that these changes will catalyze the introduction of a wide assortment of new end user visualization tools.

Acknowledgements

The authors wish to thank Ken Moreland of Sandia National Labs and Stephane Ploix of EDF for having a need for and for providing funding for this work.

References

[1] 2009. VisTrails: A scientific workflow management system. Scientific Computing and Imaging Institute (SCI), Download from: http://www.vistrails.org.

[2] Extending paraview at compile time. http://www.paraview.org/Wiki/Extending_ParaView_at_Compile_Time, 2009.

[3] Extending paraview using plugins. http://www.paraview.org/Wiki/Plugin_HowTo, 2009.

[4] Paraview: Building and installation instructions. http://www.paraview.org/Wiki/ParaView:Build_And_Install, 2009.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[6] Kitware, Inc. *The Visualization Toolkit User's Guide*, 2006.

[7] Kitware, Inc. *The ParaView Guide : A Parallel Visualization Application*, 2007.

[8] J. Loeliger, B. Collins-Sussman, and B. W. Fitzpatrick. *Version control with Git*.