ParaView-based Applications

From ParaQ Wiki

Applications based on ParaView have grown since the release of ParaView 3.0. In spite of best of our intentions we soon realized that it's not very easy to create applications that use ParaView core without subscribing to an user interface identical to ParaView's. The infamous pqMainWindowCore ended up being copy-pasted for every application and then tweaked and modified breaking every possible rule for good programming practices. Also it is hard to create ParaView-clone with limited user interface components, since various components have cross dependencies among each other and it becomes hard to ensure that all signals/slots are connected correctly for them to work.

To address these issues, we've been working on a re-factoring inspired by OverView's branding mechanism. The main goals we set out to address are:

- Facilitate creation of applications radically different from ParaView in work-flow, such as OverView.
- Facilitate creation of ParaView-variants that use most of ParaView functionality with minor behavioral or user-interface changes.

I've started working on a design based on OverView and other applications to address these goals. The latest state of this development can be checked out from git-hub repository (branch: Branding):

```
> git clone git://github.com/utkarshayachit/ParaView.git
```

The new architecture pivots around two new concepts:

- Reactions -- these are action handlers. They implement logic to handle the triggering of an action. They also implement logic to keep the enable state for the action update-to-date based on current application state. eg. pqLoadStateReaction -- reaction of load state action, which encapsulates ParaView's response for load state action. Reactions require the QAction to which they are reacting to. Any custom user interface that wants a QAction to be behave exactly like Paraview's, simply instantiates the reaction for it and that's it! The rest it managed by the reaction.

- Behaviors -- these are abstract application behaviors eg. ParaView always remains connected to a server (builtin by default). This gets encapsulated into pqAlwaysConnectedBehavior. If any custom application wants to a particular behavior, simply instantiates the corresponding behavior! We'll

see examples and it should hopefully become clearer.

Now let's look at some example applications to see how all this helps us.

Contents

Radically different Application based on ParaView core

First consider an application which is a totally new client based on ParaView core i.e. ServerManager. It has it's own custom everything. Here's how such an application's main.cxx will be:

```
#include <QApplication>
#include "pqApplicationCore.h"
#include "myCustomMainWindow.h"

int main(int argc, char** argc)
{
 QApplication app(argc, argv);
 pqApplicationCore appCore(argc, argv);
 myCustomMainWindow window;
 window.show();
 return app.exec();
}
```

As you can see, this is exactly like creating a Qt application, except that after creating a QApplication, we are creating pqApplicationCore. pqApplicationCore enables access to the pqObjectBuilder, pqServerManagerModel among other things, which is the thin Qt-layer over ServerManager. No more ProcessModuleGUIHelper, or PVMain or any such whimsical, hard to understand devices. Just create the pqApplicationCore and you are all set.

ParaView-based Applications

Now lets consider applications that are variants of ParaView that use elements for the pqComponents library such as the pipeline browser, object inspector, selection inspector etc. Such applications fall under the purview of branding. There's a new Cmake macro "build_paraview_client" that can be used for this purpose. This macro will be of the following form. Note all parts aren't implemented yet namely the plugins etc. but it's getting there.

```
build_paraview_client(
 # The name for this client. This is the name used for the executable created.
 paraview_revamped

 # This is the title bar text. If none is provided the name will be used.
 TITLE "Kitware ParaView"

 # This is the organization name.
 ORGANIZATION "Kitware Inc."

 # PNG Image to be used for the Splash screen. If none is provided, default
 # ParaView splash screen will be shown.
 SPLASH_IMAGE "${CMAKE_CURRENT_SOURCE_DIR}/Splash.png"

 # Not sure how useful this is, but since OverView was using it, we are
 # providing an option to change the text color used when showing the splash
 # screen. Optional, of course.
 SPLASH_TEXT_COLOR "black"

 # Provide version information for the client.
 VERSION_MAJOR ${PARAVIEW_VERSION_MAJOR}
 VERSION_MINOR ${PARAVIEW_VERSION_MINOR}
 VERSION_PATCH ${PARAVIEW_VERSION_PATCH}

 # Icon to be used for the Mac bundle.
 BUNDLE_ICON   "${CMAKE_CURRENT_SOURCE_DIR}/Icon.icns"

 # Icon to be used for the Windows application.
 APPLICATION_ICON "${CMAKE_CURRENT_SOURCE_DIR}/Icon.ico"

 # Name of the class to use for the main window. If none is specified,
 # default QMainWindow will be used.
 PVMAIN_WINDOW QMainWindow
 PVMAIN_WINDOW_INCLUDE QMainWindow

 # Next specify the plugins that are needed to be built and loaded on startup
 # for this client to work. These must be specified in the order that they
 # should be loaded.
 # Currently, only client-based plugins are supported. i.e. no effort is made
 # to load the plugins on the server side when a new server connection is made.
 # That may be added in future, if deemed necessary.
 REQUIRED_PLUGINS PointSpritePlugin

 # Next specify the plugin that are not required, but if enabled, should be
 # loaded on startup.
 # These must be specified in the order that they
 # should be loaded.
 # Currently, only client-based plugins are supported. i.e. no effort is made
 # to load the plugins on the server side when a new server connection is made.
 # That may be added in future, if deemed necessary.
 OPTIONAL_PLUGINS ClientGraphView ClientTreeView

 # Extra targets that this executable depends on.
 EXTRA_DEPENDENCIES blah1 blah2

 # GUI Configuration XMLs that are used to configure the client eg. readers,
 # writers, sources menu, filters menu etc.
 GUI_CONFIGURATION_XMLS <list of xml files>
 )
```

So for the new ParaView client which is a branded -paraview application itself, this looks like (the code is in Applications/Client2):

```
#------------------------------------------------------------------------
# Build the client
build_paraview_client(paraview_revamped
  TITLE "ParaView (Revamped)"
  ORGANIZATION  "Kitware Inc."
  VERSION_MAJOR 3
  VERSION_MINOR 7
  VERSION_PATCH 1
  SPLASH_IMAGE "${CMAKE_CURRENT_SOURCE_DIR}/PVSplashScreen.png"
  PVMAIN_WINDOW pqClient2MainWindow
  PVMAIN_WINDOW_INCLUDE pqClient2MainWindow.h
  EXTRA_DEPENDENCIES pqClient2
  GUI_CONFIGURATION_XMLS
    ${CMAKE_CURRENT_SOURCE_DIR}/ParaViewSources.xml
    ${CMAKE_CURRENT_SOURCE_DIR}/ParaViewFilters.xml
    ${CMAKE_CURRENT_SOURCE_DIR}/ParaViewReaders.xml
    ${CMAKE_CURRENT_SOURCE_DIR}/ParaViewWriters.xml
)
```

Here the pqClient2MainWindow is the main window which uses a ui file for the GUI design. It has empty menus as place holders for the File/Sources/Filters and other menus. Now since this client needs a pipeline browser, in the designer, we simply create a QWidget and promote it to pqPipelineBrowser and place it however we want and that's it. No more signal/slot connections or any such nonsense. pqPipelineBrowser is now a first class QWidget subclass that is autonomous in the sense that it relies on pqApplicationCore and works on it's own on simple instantiation. Exactly same is the case with pqProxyTabWidget (which is the object inspector) or pqSelectionInspector etc. etc. All these are QWidget subclasses that you merely need to instantiate in the ui file and you will get those in your application. These components have no cross dependencies. So pipeline browser doesn't depend on object inspector and vice-versa. Thus making it easier for custom clients to include only those components from the ParaView client that they are interested in.

Now lets move on to the menu/toolbars in general QActions. If my custom client has it's own File menu with just once action "Open Data", and I want this action to behave like paraview' open data where it prompts the user for the file to load based on supported readers, I do the following in my MainWindow subclass (or an auto-start plugin's initialization where the UI is being initialized)

```
new pqLoadDataReaction(ui.actionLoadData);
```

This will automatically enable/disable the action based on whether paraview is connected to a server; add action handlers to popup the file dialog listing all readers supported etc. Now how to define the supported readers? The GUI_CONFIGURATION_XMLS !!! Just list the readers under a <ParaViewReaders /> xml element and those file formats will be listed in this dialog.

Now if my application wants a File menu exactly like ParaView's, then we do:

```
pqParaViewMenuBuilders::buildFileMenu(*ui.menu_File);
```

pqParaViewMenuBuilders has helper methods to create the actions and reactions for those actions for all standard paraview menus. Looking at the implementation of those, custom-app writers can pick and choose the reactions for their custom menus.

Finally, behaviors. If our client needs to stay connected to a server always, like ParaView, simply instantiate the pqAlwaysConnectedBehavior in the main window constructor or an auto-start plugin.

The MainWindow implementation for the new ParaView-client currently looks as follows:

```
#include "pqClient2MainWindow.h"
#include "ui_pqClient2MainWindow.h"

#include "pqParaViewMenuBuilders.h"
#include "pqParaViewBehaviours.h"

class pqClient2MainWindow::pqInternals : public Ui::pqClient2MainWindow
{
};

//-----------------------------------------------------------------------------
pqClient2MainWindow::pqClient2MainWindow()
{
  this->Internals = new pqInternals();
  this->Internals->setupUi(this);

  // enable automatic creation of representation on accept.
  this->Internals->proxyTabWidget->setShowOnAccept(true);

  // Populate application menus with actions.
  pqParaViewMenuBuilders::buildFileMenu(*this->Internals->menu_File);

  // Populate sources menu.
  pqParaViewMenuBuilders::buildSourcesMenu(*this->Internals->menu_Sources);

  // Populate filters menu.
  pqParaViewMenuBuilders::buildFiltersMenu(*this->Internals->menu_Filters);

  // Define application behaviours.
  // pqParaViewBehaviours simply creates all the behaviour instances used by ParaView by default.
  // Currently equivalent to following, but will change as new behaviors are added.:
  // new pqDefaultViewBehaviour(this);
  // new pqAlwaysConnectedBehavior(this);
  // new pqNewSourceActiveBehaviour(this);
  new pqParaViewBehaviours(this);
}

//-----------------------------------------------------------------------------
pqClient2MainWindow::~pqClient2MainWindow()
{
  delete this->Internals;
}
```

Using behaviors and reactions makes it possible to avoid ending up with a huge monolith as pqMainWindowCore where all application logic gets concentrated. It also makes it possible to pick-and-choose when writing custom apps, avoiding duplication whenever possible.

Please take a look at the ui file and C++ code in ParaView3/Application/Client2 in the git repository to understand how this works.

Interesting Side-effects

As I am implementing more and more stuff, I am realizing neat (and not so neat) advantages of this restructuring. Here's a list of those:

- Often we have two disconnected sections of the gui wanting to the same

thing eg. the pipeline browser's context menu has a "Change Input" action as well as the "Edit" menu. In such cases we either end up duplicating the code or hacking to have a cross reference (in current ParaView, the Edit menu calls the pqPipelineBrowser::changeInput(). Hence now edit menu requires the pqPipelineBrowser to work! A nice thing with implementing reactions is that they provide a logical place to put such logic that can be reused by whoever is interested. So now I have a pqChangePipelineInputReaction which handles change of input, including when change input action is enabled etc. and I instantiate the reaction for both the Edit menu as well as the PIpeline browser's context menu. As a result both behave exactly the same with no code duplication and less bug prone since there's only one place to fix how an input changes! And because reactions even manages the enable/disable state it's just works nicely together -- I am eulogizing I know.

- Reactions make it easier to avoid undo-redo related issues. As a rule of thumb, every reaction does work within an undo-block. So just use the helper functions BEGIN_UNDO_SET("name for undo-set") and END_UNDO_SET() and the start and end of your reaction crux and you are golden!

Resource Space

The Resource space has some reserved directories/files which are used to load brand specific configurations.

Location	Role
:/<app-name>/SplashImage.img	Splash Image used for splash screen and About dialog
:/<app-name>/Configuration/.xml*	GUI configuration XML files which includes readers, writers, filters menu, sources menu etc. If muliple xml files are present, then all are loaded.
:/<app-name>/Documentation/.qch*	Application documentation. If multiple Qt compressed help files are detected, then all are loaded.

Configuration XML Formats

Some components of the application can be configured using configuration xmls. These xmls must either have the root element as the tag required for configuring the component or that tag must a first-level child element under the root element i.e. for configuring the reader-factory, your configuration xml can

be:

```
<ParaViewReaders>
</ParaViewReaders>
```

OR

```
<SomeRoot>
  ...
  <ParaViewReaders>
    ...
  </ParaViewReaders>
  ..
</SomeRoot>.
```

ParaViewReaders: Reader Factory Configuration

Reader Factory is used by FileOpen dialog/recent files and the like. Configure the reader factory to specify the support readers and file-formats. The identification tag for this configuration is <ParaViewReaders>. The format of this xml is as follows:

```
  <ParaViewReaders>
     <Proxy group="[sm-proxy-group]" name="[sm-proxy-name]" />
     ....
  </ParaViewReaders>
```

ParaViewWriters:Writer Factory Configuration

Writer factory is used when writing datasets. Configure the writer factory to specify the supported writers. The identification tag for this configuration is <ParaViewWriters> and the format is same as that for the reader-factory.

```
  <ParaViewWriters>
     <Proxy group="[sm-proxy-group]" name="[sm-proxy-name]" />
     ....
  </ParaViewWriters>
```

ParaViewFilters : Filters Menu Configuration

This is useful only if you are using the standard filter's menu provided by ParaView. The identification tag for this configuration is <ParaViewFilters />.

```
<ParaViewFilters>
  <Category name="[category name]" menu_label="[label for category sub-menu"
    preserve_order="[optional, when 1, the filters are not sorted alphabetically in this sub-menu]"
    show_in_toolbal="[optional, when 1, a toolbar is created for this category]">
    ...
    <Proxy group="sm-group" name="sm-name" icon="optional, icon resource name" />
    ...
  </Category>
  ....

  <Proxy group="sm-group" name="sm-name" icon="optional, icon resource name" />

  ....
</ParaViewFilters>
```

ParaViewSources : Sources Menu Configuration

This is useful only when you are using the standard sources menu provided by ParaView. The identification tag is <ParaViewSources /> and the format is same as that for the ParaViewFilters.

Application Initialization Sequence

When an ParaView-based application is created using the build_paraview_client() mechanism described here, following are sequence in which different main operations are performed.

- The applicationName, applicationVersion and organizationNAme as specified in the macro, are set using the static QCoreApplication API. This happens before any objects are instantiated.
- QApplication instance is created. This is required for any Qt-based application.
- pqPVApplicationCore instance in instantiated.
  - This first initializes the server-manager application i.e. the process-module is set up, the proxy-manager is set up.
  - This results in creation of the various managers such as the pqPluginManager, pqPQLookupTableManager, pqAnimationManager, pqSelectionManager etc.
- The QMainWindow subclass specified in the macro or QMainWindow if none is specified, is instantiated. Once the core components are initialized, the main window is created. So if you write your own QMainWindow subclass, you are free to use any of the server-manager or pqPVApplicationCore components as needed in your initialization code.
- Next, we try to load the required and optional plugins are listed in the macro. If a required plugin could not be located or loaded, then the application quits with an error. If an optional plugin could not be located or loaded, then they are quietly skipped. Note this is happening after the mainWindow has been created. So do not use any components that will be

brought in by the plugins in your mainWindow initialization code. The locations where these plugins are searched are as follows in the given order:

- executable-dir (for Mac *.app, it's the app dir)
- executable-dir/plugins/pluginname
- *.app/Contents/Plugins/ (for Mac)

This is bound to change. Please refer to the documentation of Qt/Core /pqBrandPluginsLoader.h for a complete and updated list.

- Once the plugins are loaded, the next step is to load the configuration xmls specified in the macro. All these xmls get compiled into a qt-resource that is then processed one after the other by calling pqApplicationCore::loadConfiguration(). Any GUI components that processes such configuration files listen to the pqApplicationCore::loadXML() signal and process the configuration xml as and when it is loaded. Since the configuration xmls are loaded after the plugins are loaded, your plugins can rely on configuration xmls.
- Finally, the mainWindow's window-title is updated to match that specified in the macro and then the mainWindow is shown and the Qt event loop is begun.

TODO List

- Undoing apply should re-enable Apply button -- fixed.
- The buttons on the top of the View in the pqViewManager need to be customizable -- fixed.
- Pipeline browser-eyeball and renaming doesn't work yet -- fixed.
- pqStandardViewModules is instantiated in the pqApplicationCore, that's wrong. It must become a behavior that applications can employ -- fixed.
- Missing Toolbars: Common Filters toolbar, Axes toolbar -- fixed
- Python shell reactions --fixed
- Help doesn't seem to work on Mac, can't seem to locate the assistant -- fixed
- paraview_rebased -dr comes up totally crappy -- fixed
- About dialog -- maybe build_paraview_client macro should provide mechanism to specify custom about dialog -- fixed.
- toggling eye visibility is broken when trying to toggle the visibility of item not currently selected (fixed).
- redo broken for redoing of creation of a source -- apply button doesn't get enabled. (fixed)
- Test playback with surface selection not working -- fixed.

- Make ParaView's standard GUI configuration xmls easily available to custom apps.

- Fix install rules for custom applications as well as the new libraries added in the process (and also for the assistant stuff -- qt.conf etc.)
- Progress Widget - need a new progress widget that directly listens to the progress events from the vtkProcessModule. Also capability for disabling the full GUI when playing animation etc.
- Add a ton of examples for custom application using different styles for doing different things including changing display policies, changing how representations are created etc.
- Help from plugins -- register *.qch files at run-time with the application main help collection file (*.qhc).
- This has nothing to do with branding: paraview application should have some mechanism (maybe a config file) that can be used to list the plugins that should be loaded by default) thus making it possible to create variants of standard paraview by simply changing the set of plugins loaded. This will be suitable for cases where the application wants all the standard paraview stuff, just with additions that be made using plugins.
