ParaView Scripting Changes

From VizWiki

Contents

Introduction

The purpose of this Wikipage is to capture the ideas and observations concerning the design and implementation of a revised scripting language for ParaView. The interested parties are Sandia National Laboratories and Kitware Incorporated.

Current State of ParaView Scripting

ParaView currently uses Python as its scripting language. A majority of ParaView implementation classes are "wrapped", making them accessible from a Python interpreter. Users can build VTK data pipelines using Python to drive ParaView from both inside and outside of the ParaView GUI client.

The current ParaView Python scripting language is flexible and powerful in that it allows almost all classes defined in VTK and ParaView to be accessed from Python. This power and flexibility also greatly limits the number of ParaView users who can spend the time to master and apply ParaView Python scripting for their domain specific problem. In addition, the Python scripting interface is not well documented, which forces users to consult the C++ implementation details of the underlying implementation classes to fully use ParaView Python scripting.

ParaView uses XML files to record and load the state of the ParaView GUI client. The XML output for ParaView state files is not easily read or edited by users.

Desired Changes and Additions to ParaView Scripting

Python based

The modified ParaView scripting language is command based and in Python. The scripting commands are

simple, uniform, and documented. The documentation is available in varying levels of detail from the Python interpreter, the ParaView program help system, and from the ParaView webpage. Use of these scripting commands does not require a deep understanding of Python programming or the implementation specifics of ParaView and VTK. The scripting commands are modular and extensible. Old scripts can be appended to new scripts or used like subroutines.

- Users can still access the more featured and flexible underlying Python interface that exists currently in ParaView. The new command language is similar to a macro language that encapsulates a large number of low level Python commands.
- The scripting command language supports calculator type operations in addition to visualization operations. Examples of such calculations are the average values of results variables for defined regions of the mesh, and integration of results variables over individual elements of the mesh. The quantities should be created by a simple function interface, and would then exist as variables in the Python interpreter for further calculation.
- Create a library of pre-recorded scripts that is made available for common visualization and data post-processing activities. Users can make small edits to these scripts for custom applications.
- Create a scripting record and playback capability in the ParaView client GUI. A simple key press in the GUI should initiate the recording of a user readable and editable script based on the data visualization constructed in the GUI. Previously recorded GUI scripts can be loaded and executed with a similar key press. The record and playback capability is distinct from the XML based client GUI load and save state feature currently implemented in ParaView.

ParaView scripting can drive run-time visualization (in-situ)

This use case involves coupling ParaView as a library within the simulation code and the visualization being created using results in the simulation codes memory space, as they are generated. This scripting language is used to tell the ParaView library how to render the simulation code data. There is also interest in translating other scripting languages into the ParaView scripting language for run-time visualization (for example CTH Slang scripts).

Command simplification

Hide/automate repetitive tasks and technical details. The ParaView scripting still contains a dizzying amount of overhead for doing simple things. For example, creating any reader or filter, have it registered in the pipeline browser, and be ready to view should take only a simple command. The following is a running list of script behaviors that need cleaning up.

- When you create a reader or filter, it should automatically be registered with a default name with the proxy manager so that it shows up in the pipeline browser.
- Representation objects should be created automatically.
- The method for selecting input arrays (the 5 argument call to SetInputArrayToProcess) should be cleaned up. There should be some syntactic sugar to simply specify the input array with the array name.
- Sub proxies, like the ClipFunction for the Clip filter, should be created (or set) automatically. The user should only need to give the same name as in the GUI and then be able to access the proxy.

Scripts from the GUI

There should be a mechanism to easily build scripts that mirror (or closely mirror) interactions that the user does with the GUI. This probably includes either the ability to turn on a "macro recording" feature that writes out a script of actions the user performs or the ability to write out the current state as a script that will set up the state.

Hotly contested changes

The following are changes that are still being contested. When resolution is achieved, they will be moved above or simply removed.

Active pipeline objects

Interaction with pipeline objects through the GUI happens by having an actively selected pipeline object. Although this is not necessary in a scripting environment (objects can be referenced by variable identifiers), it could be helpful. It would work in a similar manner as the GUI (in fact, the active pipeline object in each should probably follow the other). When new pipeline objects are created, they become the active one. When filters are created, their input is set to the active object. Properties of the active object can be manipulated by simply referencing the active object.

Arguments for:

- Consistency between GUI and scripting. By following the same mechanism of the GUI, the scripting becomes easier by becoming an extension of the GUI actions and vice versa.
- The active object with some syntactic sugar can give the scripts a more imperative feel. If a user wants to add a sequence of operations to data, they simply create the objects in the order they want them applied.
- Simply referring to the active object reduces the need to create identifiers for each pipeline object; they are generally only needed at pipeline branches. Thus, when writing a script the user does not have to continuously come up with new variable names. Furthermore, GUI generated scripts do not need to create a bunch of impossible-to-read identifiers (as we have in the past), or if it does then they will be used much less frequently.
- Referring to active objects rather than object identifiers can make it easier to move around scripts and script fragments. For example, if I have a section of code that adds some combination of filters to an existing pipeline object, I can just cut and paste that code anywhere without having to resolve variable names and the like.

It should be noticed that both EnSight and VisIt scripts have the concept of active objects or data, and that feature has contributed greatly to the popularity of each. EnSight scripts are complicated, but you are able to easily write out scripts from the GUI and then cut and paste scripts together. VisIt scripts are popular because they are very "clean" and are easy to read and write. Because new objects act on an active set of databases or plots, VisIt scripts require almost no bookkeeping.

Arguments against:

- In short, making all the scripting commands work of an active selection will add more confusion and complexity in the long run. Using an active pipeline object is a feeble attempt to hide the nature of the pipeline concept. However, the pipeline is a fundamental concept when working with ParaView. Although it may be counterintuitive to users at first, the sooner they embrace the idea, the easier things will get.

- Within python, not storing the proxies for pipeline objects leads to headaches trying to retrieve them from the servermanager.
    - There could be many ways to retrieve pipeline objects if the script did not save the reference in a local variable (which it already has the option to as mentioned below). You could get the object by name from the servermanager as mentioned. This can be problematic though because, for instance, two objects could have the same name. Another approach is to select by index into a list of existing pipeline objects (both EnSight and VisIt select objects this way and I believe that ParaView can do this at least internally). In a way, you get the best of all worlds.--Ken 09:31, 1 October 2008 (MDT)

- Forcing someone to use a name will lead to them actually naming items within the pipeline to make the scripts easier to understand.
    - Just because someone is forced to make a name doesn't mean that they are going to come up

with one of any meaning. Often filters produce intermediate results which have no specific meaning anyway. Besides, if a user names an object, it might as well be the one that shows up in the pipeline browser, not one that is hidden in the script.--Ken 09:35, 1 October 2008 (MDT)

- Global variables of this nature are usually a bad idea in modern programming languages. They undermine the natural scoping of function and method calls by providing side effects outside of the obvious parameters of the call.

- (Dave K) I would contend that EnSight's active object and selection concepts are a detriment to the scripting language. EnSight defaults to using EnSight part number as the selection id. There's an option to use part names as well. The default names are about as useful as ParaView's. The methods for selecting an active object in EnSight are pretty bad. First choice is to use the object ID, an arbitrary number based on the block ordering (not numbering) within an exodus file and the number of items before it. Second choice is basically a regular expression on the name with no notification of errors (last time I used it). Furthermore, within EnSight, cut and paste fails when the part ID's are different or were created in a different order. [What does VisIt do -- how does it handle its "active" objects?]
  - VisIt identifiers are more-or-less the same. Plots and operators are referenced by integer. However, because there are usually a small number of plots, because databases can be referenced by filename, and because operators can usually be referenced by type, the identification is usually less of an issue.
  - I would contend that the detriment you speak of is the poor identifiers for objects, not the fact that one is active. This is a problem no matter what when scripts are automatically generated; the identifiers never make sense. EnSight scripts alleviate the problem significantly by working against an active object. Imagine if every line in the EnSight script had to contain the part number on which they acted upon. Consider how much harder that would be to read. It would be even harder to copy/paste; you would have to change every line to reference the appropriate object rather than just one selection line. --Ken 10:09, 1 October 2008 (MDT)

- EnSight actually doesn't have a default selection, a part is explicitly selected after it is created.

```
clip: begin
<clip parameters elided>
clip: end
clip: create
part: select_begin
 7
part: select_end
```

- Python is an object oriented language. Users who know python will not expect "hidden" objects to be acted on. How will changes in scope be handled? For instance, if I call a subroutine that creates a new dataset, will it be active when the routine returns? If not, how can I refer to it?
  - Based on the discussions so far, I am becoming convinced that the "active" object has a name (Dave below refers to it as *data* but perhaps a more descriptive name like *activedata* or *activesource* makes sense) that is used like any other reference to a pipeline object with parameters. --Ken 10:36, 1 October 2008 (MDT)
  - The side effects of a method would be pretty limited. In this description, apart from creating the objects it was clearly designed to do, it might change the value of the active object to something it created. (How the method returns references to the objects it created is orthogonal; it should probably be solved the same way regardless of the existence of the active selection.) This does bring up the point of what should change the active pipeline object. Should the active pipeline object be changed to a new object when it is created (as it is in the GUI) or should the active object be changed more explicitly (i.e. something like `activesource = CreateSphere(...)`)? --Ken 10:36, 1 October 2008 (MDT)
    - The active pipeline object should probably be set by the code generated by the wrapper. As long as it's a normal Python variable, users could assign new values in between object constructors (and there might be some sugary Python function like `SetActiveSource()` that simply assigns a value to *activeSource*) but the wrappers should force new values when new filters/sources are constructed. It might make life easier on script writers if `setActiveSource` set not only *activeSource* but also fetched the output from the source and set *activeData* so that it's easy to fetch summary

data about the output dataset quickly (like the names of fields, their ranges, etc.). --Dcthomp 11:33, 1 October 2008 (MDT)

- We are moving off topic, but since active data and source are so tightly coupled, one should depend on the other to avoid the possibility of a mismatch. Thus, there would be no *activeData* but rather *activeSource.data()*. --Ken 12:48, 1 October 2008 (MDT)

- What objects exactly will have an active default? Filters? Datasets? Views? If more than one of these, it might get confusing to users. For example, if there is both an active dataset and an active view, then it probably won't be clear to users whether

```
color_by( 'EQPS' )
```

is applied to the view or the dataset (or, in this case, the combination of the two).

- One interesting compromise might be to have active objects but force users to refer to them by (a fixed) name. For instance, the active dataset might be *data* and the active view would be *view*. This would still allow easy cutting and pasting of scripts but be more explicit about what is being acted upon. Instead of

```
open( 'can.ex2' )
clip( plane( origin=(0,4,-5), normal=(1,0,0) ) )
add_to_view()
color_data_by( 'EQPS' )
camera( aim=(0,0,0), eye=(10,0,0) )
save_image( 'can.png' )
```

you would have

```
open( 'can.ex2' )
data.clip( plane( origin=(0,4,-5), normal=(1,0,0) ) )
view.add( data )
view.color_by( data, 'EQPS' )
view.camera( aim=(0,0,0), eye=(10,0,0) )
view.save_image( 'can.png' )
```

but you would also have the opportunity to name objects to perform more complex operations:

```
can = open( 'can.ex2' )
clipped = can.clip( plane( origin=(0,4,-5), normal=(1,0,0) ) )
contour = can.contour( 'EQPS', (20000,30000,40000) )
view1 = create_view( data=clipped, aim=(0,0,0), eye=(10,0,0) )
view1.add( contour )
view1.color_by( clipped, 'EQPS' )
view1.color_solid( contour, color( red=1, green=1, blue=1 ) )
view1.save_image( 'can.png' )
```

High-level functions

While access to low-level proxies that directly correspond to pipeline objects is useful, the higher-level scripting language should be more focused on typical user tasks than on maintaining a one-to-one correspondence between pipeline objects and script commands. For instance, with large datasets -- rather than clipping the entire dataset and letting the mapper extract a surface to draw -- it is frequently useful to extract a boundary, clip the boundary, cut the source with the clipping plane, and then draw the clipped boundary plus the cutting plane. This would be a good place for a higher-level scripting language to provide a function like extract_clipped_surface().

> I'm not sure there is any feature request to be derived from this. Python already allows you to make a function to do exactly what you described. What further feature or binding for the scripting needs to be specified? --Ken 11:10, 1 October 2008 (MDT)

Another example where there should not necessarily be a one-to-one mapping between pipeline objects and script commands is where there are multiple filters that do nearly the same thing. This is important because it is hard for users to browse the hugemongous list of VTK filters and know which one will do what they want. For instance, "Clean to grid" and "Clean" provide very similar behavior. They should really present the same script interface with the unstructured grid version ignoring some parameters. Another case where it *might* be useful to present multiple filters under a single interface is the Calculator and Programmable Python filters.

> I don't think it is appropriate to address these at the scripting level. These issues, particularly the one concerning the clean vs. clean to grid, effect GUI users as much as scripting users. Lets fix the problem at the filter level. For example, we could make a filter that cleans poly data to poly data, unstructured data to unstructured data, and accepts nothing else, and then have a second filter that converts structured data to unstructured data for a cleaner conceptual difference. We can deliberate more on the specifics of these proposed filters, but the scripting page is not the place for it. --Ken 11:10, 1 October 2008 (MDT)

Finally, there is some awkwardness with the wrapped versions of some routines and it would be nice for higher-level scripting functions to work with more Python-esque bindings. For example, in the can-clipping examples above, having planes specified using a pair of tuples for the origin and normal is much more convenient than

```
x = vtkPlane()
x.SetOrigin( 0, 0, 0 )
x.SetNormal( 1, 0, 0 )
```

> I totally agree that the interface to subproxies on things like clipping functions needs some serious retooling to make it easier to set. I propose that the clip filter should be assigned a default clip function and have one or more methods to change the clip function to a specified default. This would be either something like `SetClipFunctionToPlane()` or `SetClipFunction("plane")`. The subproxy should also be referenced just like any other proxy. So, you would get something like the following (which I think is very readable).

```
clip.SetClipFunctionToPlane()
clip.ClipFunction.Origin = [0, 0, 0]
clip.ClipFunction.Normal = [1, 0, 0]
```

> I think this feature could be automatically added to filters of this nature with subproxies by looking at tags like ProxyListDomain in the server manager XML. --Ken 11:10, 1 October 2008 (MDT)

Similar methods for constructing colors, transfer functions, and other geometric primitives that can be used to prepare pipeline objects or set display parameters would be useful.

Backward filter application

In the current version of scripting and in all previous versions I have seen, filters are added by creating the filter object and setting the input on it appropriately. Basically, you get something like this:

```
clip = filters.Clip()
clip.Input = can
```

Above, Dave freehanded some potential code that added a filter by calling a method on the source.

```
clip = can.Clip()
```

This struck me as a natural way to apply filters. I want to clip something, so I call clip. I want to isosurface something, so I call isosurface on it. I don't know if it's practical to add a method to every source for every possible filter, but it should be possible to add a single method that creates a filter with the object as input.

```
clip = can.Filter("Clip")
```

Acknowledgments

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.

Retrieved from "http://vizrd.srn.sandia.gov/vizwiki/ParaView_Scripting_Changes"

- This page was last modified 18:38, 2 October 2008.