



VisWeek 09
VIS • INFOVIS • VAST

ParaView

In Situ Post-Processing and Visualization

Nathan Fabian

David Thompson



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-AC04-94-AL85000



Outline

- Motivation & Teaser
 - Example
 - Part I: Creating a script and source
 - Part II: Gluing it all together
 - Conclusion
-
- ↑ David
- ↓ Nathan



Motivation & Teaser



Motivation

- Want to allow direct access to simulation state
 - To avoid writing to disk when possible
 - To perform analysis not otherwise possible
- Avoid rewriting glue code
- Provide but don't require parallel rendering
- Allow quick reconfiguration for exploration
- Hmm... why not take advantage of ParaView Scripting from this morning's tutorial?



Solution

- A utility library available to simulations via a simple interface:

Initialize Start up pvpython interp, provide script, provide information about state variables

Update Evaluate Python function to update pipeline

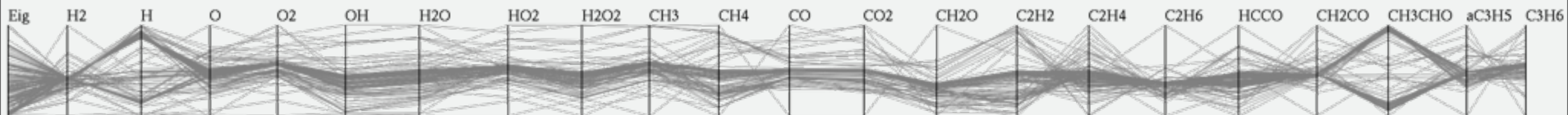
Finalize Shut down Python interpreter

- Minimal work on analysis side:
 - Write a data source
 - Create/modify a ParaView Python script



Results

PCA run on subgrids of a 2025x1600x400 combustion simulation:



Thanks to Chunsang Yoo, Jacqueline H. Chen, and Ray Grout for providing combustion data, compute time, and expertise.



Overview of Interface

- View from simulation side:
 - Calls to initialize/finalize a utility library
 - One call per timestep for updates
- View from analysis script side:
 - Simulation is source of a pipeline set up by script.
 - One script function called each timestep to possibly modify and definitely re-run the pipeline



Source Class

- Inherits `vtkAlgorithm` subclass based on type of data (e.g. `vtkPolyDataAlgorithm`)
- Will usually own a data object to shallow-copy into `RequestData()` output
- Provides methods for simulation to prepare data object with simulation state. You must decide:
 - whether to copy or grant direct access
 - whether to use barriers/fences and threads or not
 - whether to provide derived state or direct state



Example: Simulation

```
CALL in_situ_init( rank, comm, &  
  num_species, species, &  
  ni, nj, nk, mi, mj, mk, neighbors )
```

```
10 CALL advance_simulation( time, dt )  
  CALL in_situ_step( time, dt )  
  IF ( .NOT. evaluate_exit_condition() ) GOTO 10  
  
  CALL in_situ_fini( rank, comm )
```



Example: Simulation

```
CALL in_situ_init( rank, comm, &  
  num_species, species, &  
  ni, nj,  mi, mj, mk, neighbors )
```

```
10 CALL  
CALL  
IF (   
  
CALL
```

- Prepare VTK source with simulation data
- Initialize ParaView interpreter
- Read and execute Python script
 - Script creates VTK source instance
 - Script defines update function

10



Example: Simulation

```
CALL in_situ_init( rank, comm, &  
  num_species, species, &  
  ni, nj, mi, mj, mk, neighbors )
```

```
10 CALL  
CALL  
IF (   
  
CALL
```

- Prepare VTK source with simulation data
- Initialize ParaView interpreter
- Read and execute Python script
 - Script creates VTK source instance
 - Script defines update function



Example: Simulation

```
CALL in_situ_init( rank, comm, &  
  num_species, species, &  
  ni, nj, nk, mi, mj, mk, neighbors )
```

```
10 CALL advance_simulation( time, dt )  
  CALL in_situ_step( time, dt )  
  IF ( .NOT. evaluate_exit_condition() ) GOTO 10  
  
  CALL in_situ_fini( rank, comm )
```



Example: Script

```
from paraview import *  
s3dData = S3DSource()  
pca = GridPCA()  
pca.Input = s3dData  
km = PKMeans()  
km.Input = pca  
wr = TableWriter()  
wr.Input = km  
def update( p, ti, time, dt ):  
    wr.SetFileName( 'kmeans-%d-%d.vtk' % (p,ti) )  
    wr.Write()
```



Example, Part I

Bootstrap



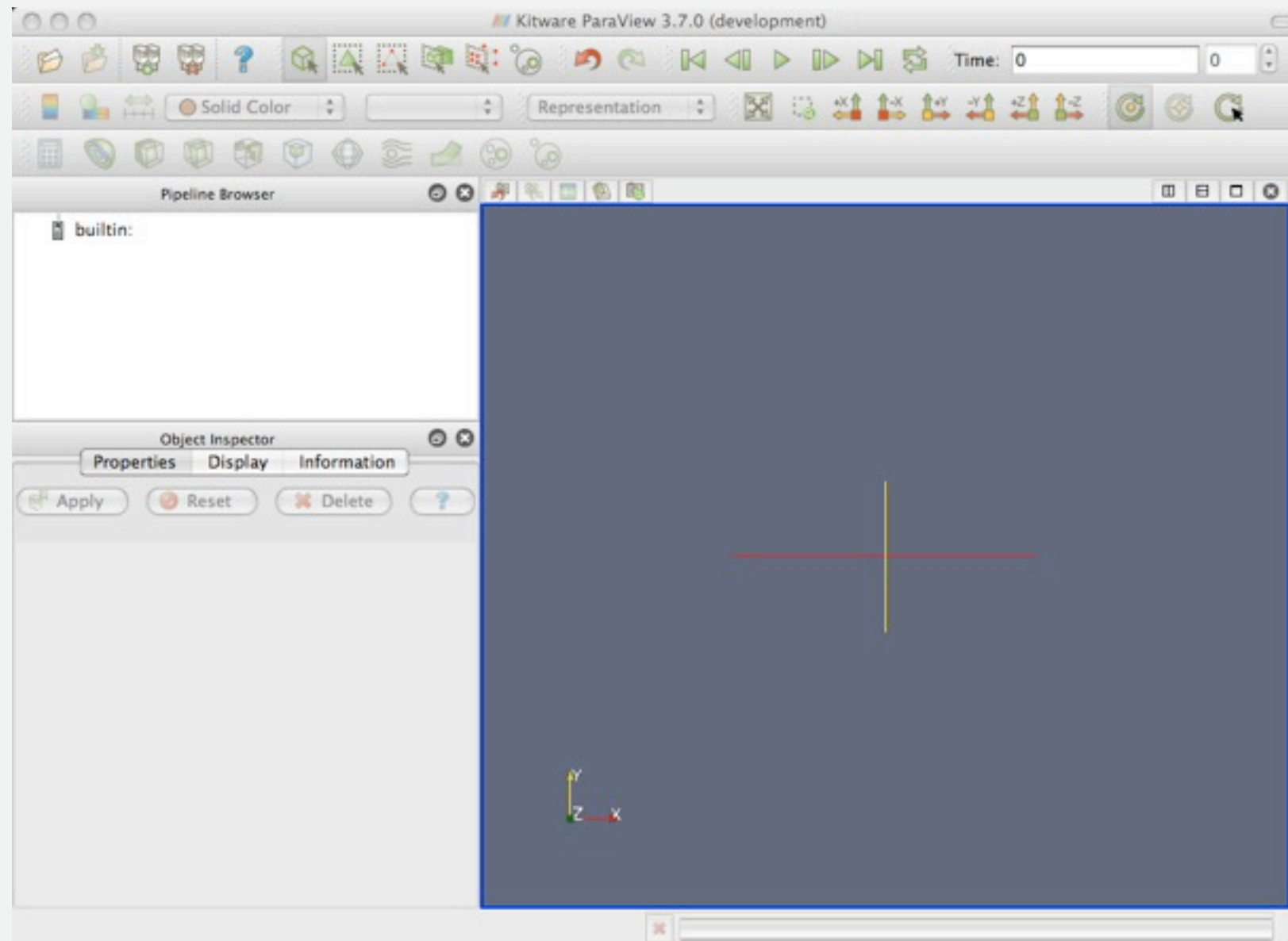
Bootstrapping

- Get some data similar to your simulation.
(Write VTK files from your simulation **or** just modify some dummy data to look like your simulation.)
- Use ParaView to prototype.
- Prepare ParaView trace for use *in situ*.
- Write a “source” class.
- Create and update pvbatch engine from your simulation.



Creating a Script

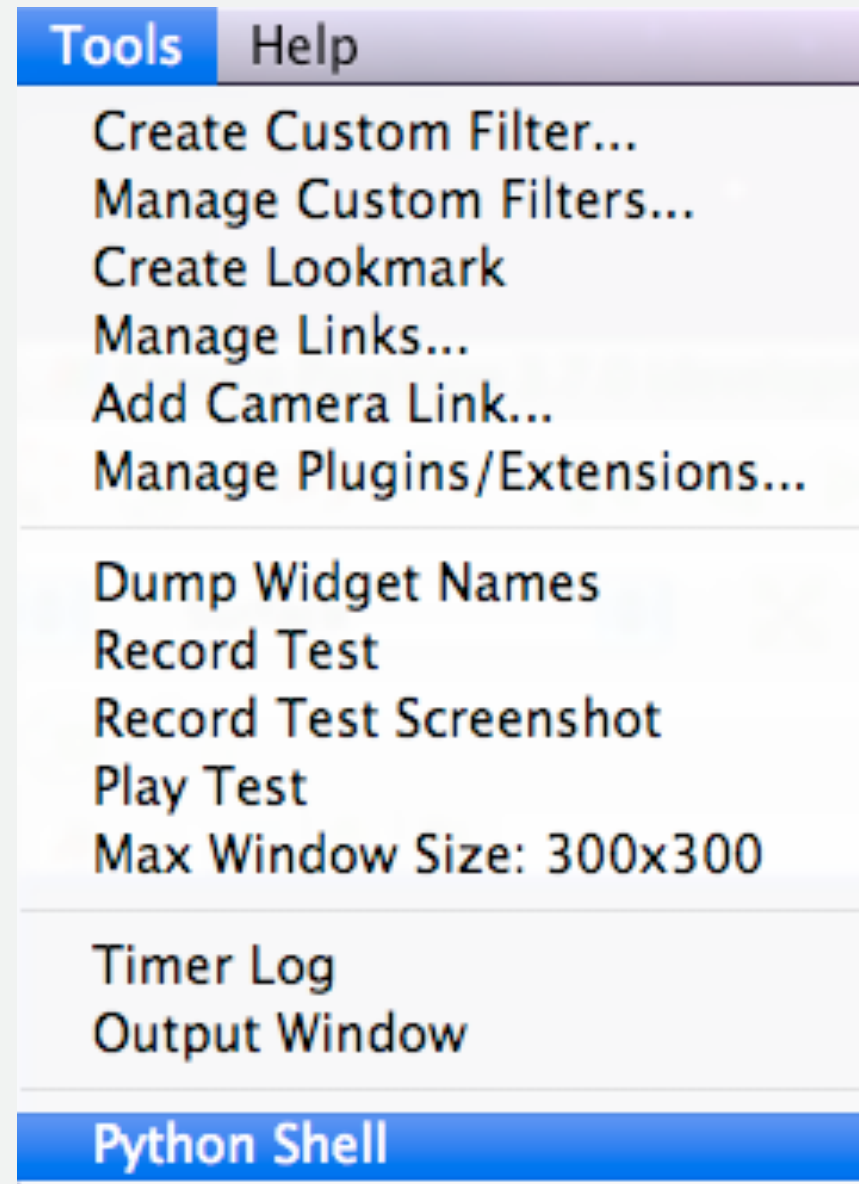
- No need to write script from scratch...
- ... use ParaView to prototype.





Creating a Script

- No need to write script from scratch...
- ... use ParaView to prototype.





Creating a Script

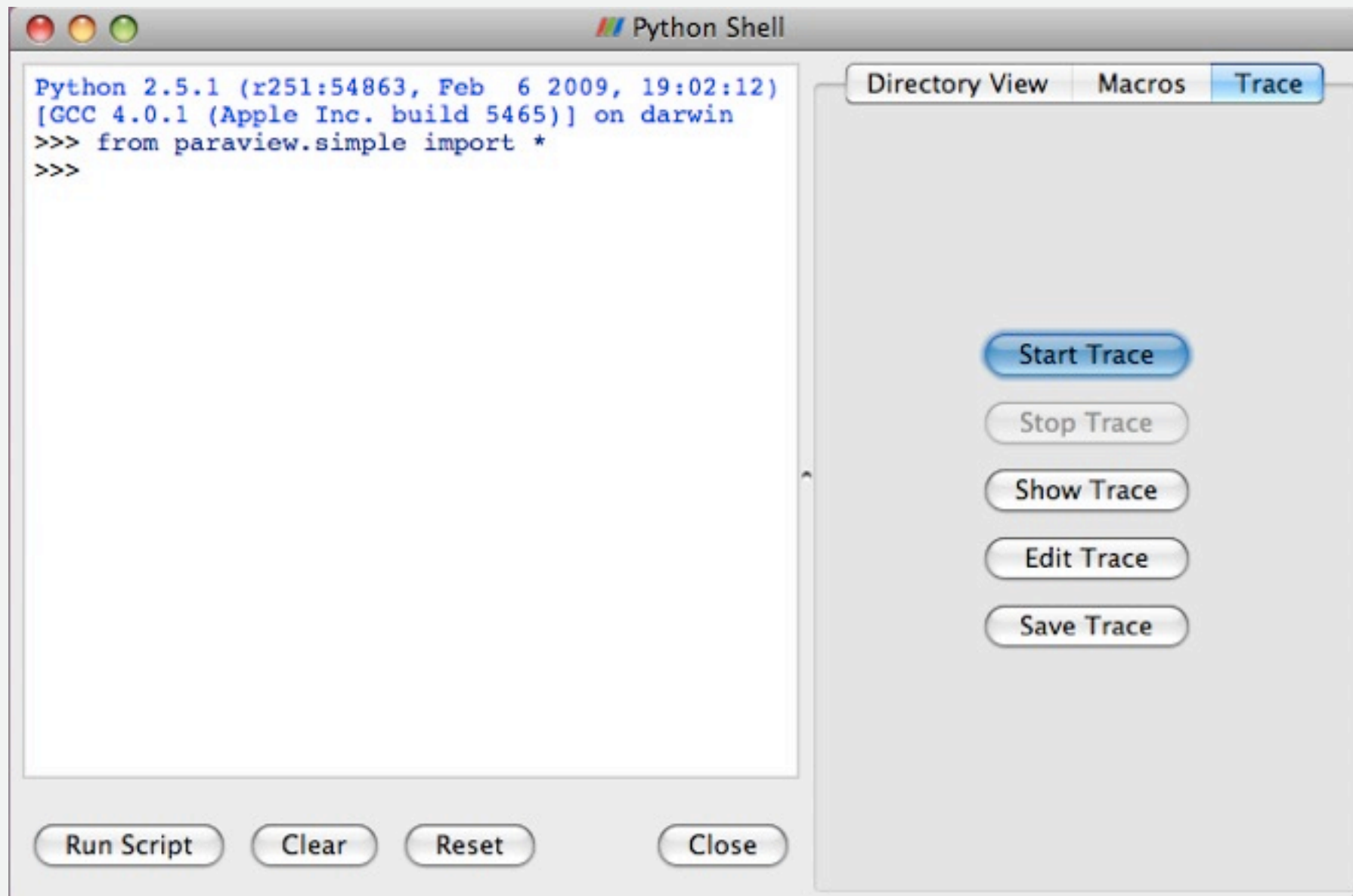
- No need to write script from scratch...
- ... use ParaView to prototype.





Creating a Script

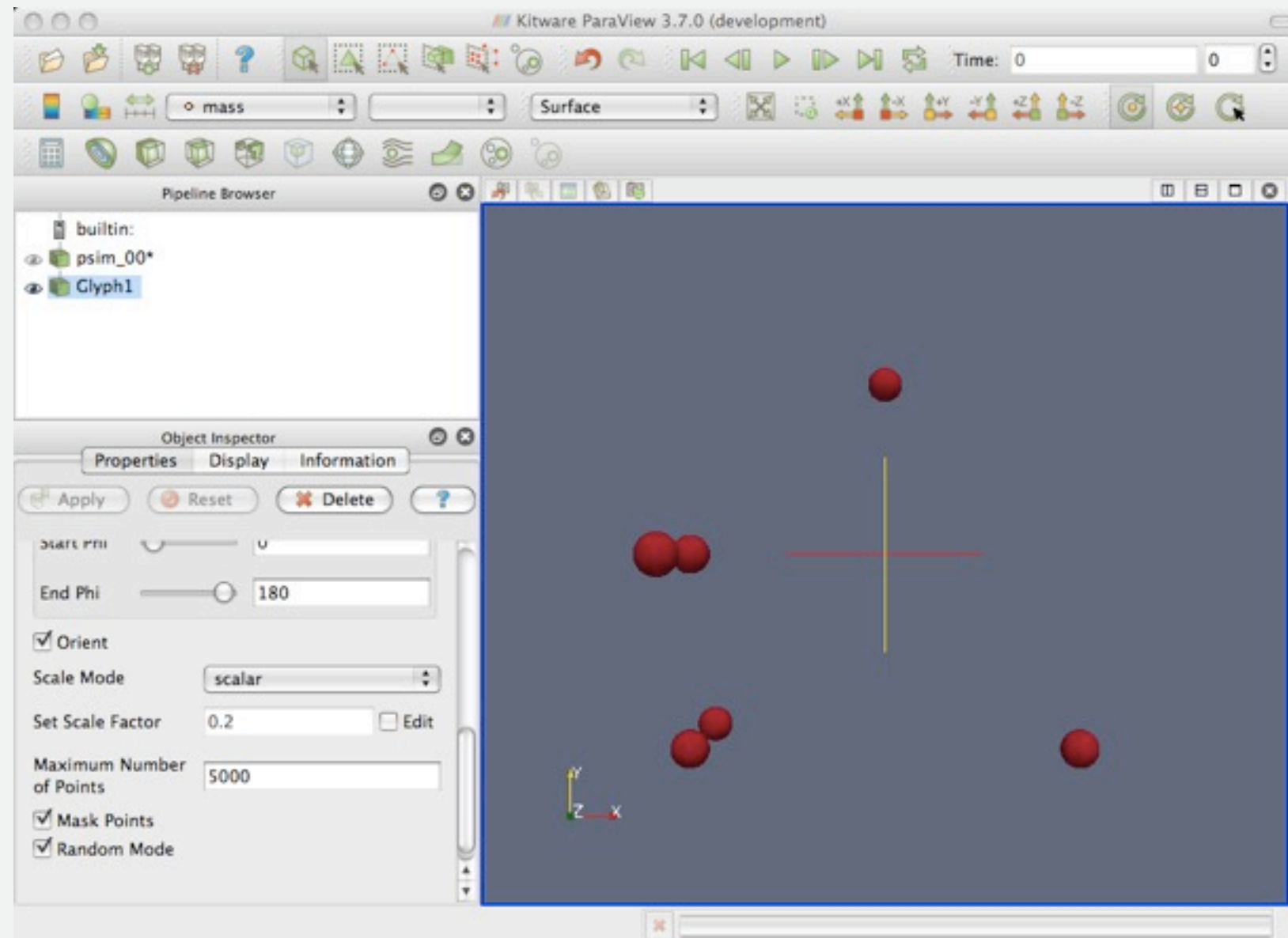
- No need to write script from scratch...
- ... use ParaView to prototype.





Creating a Script

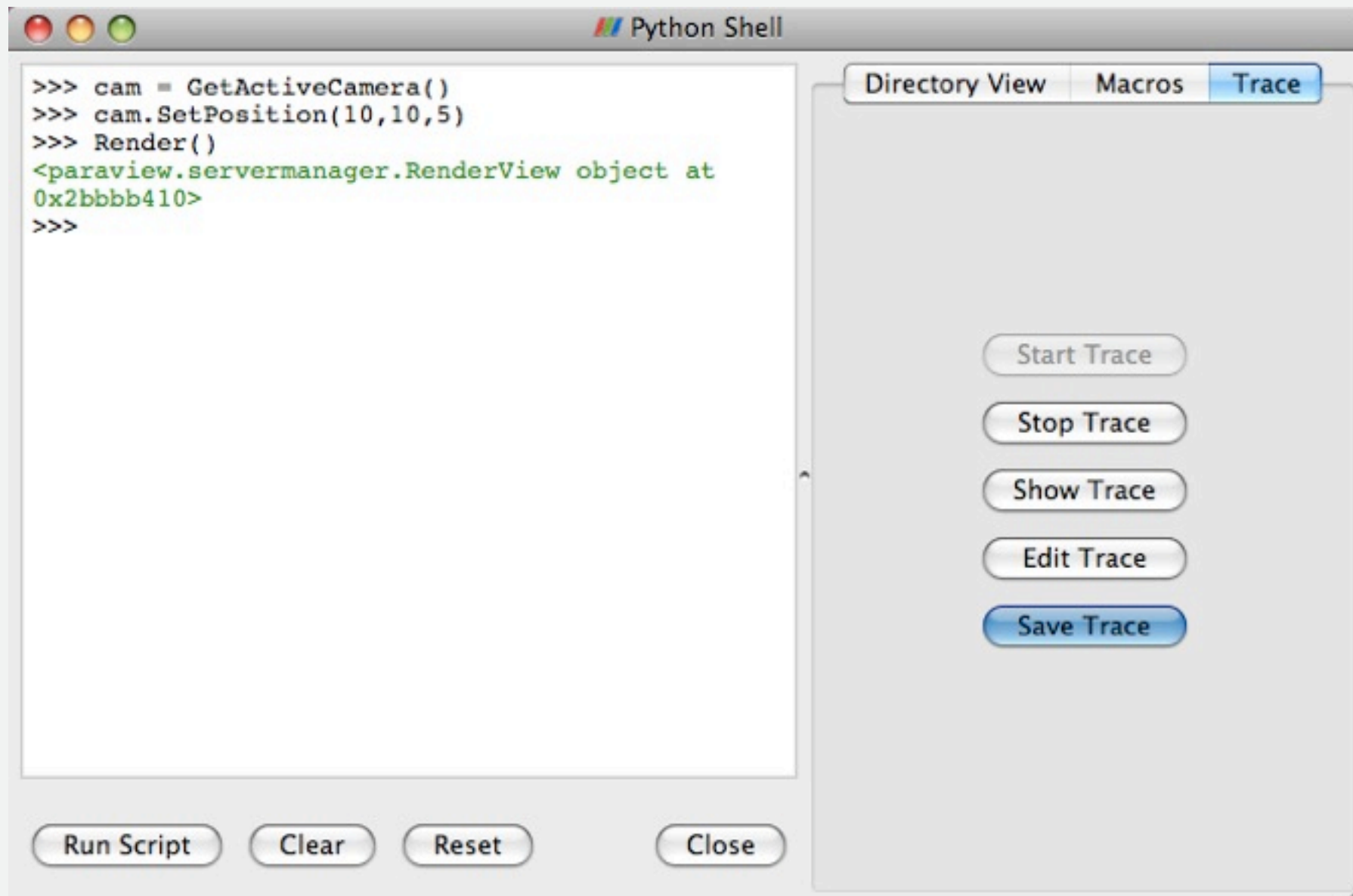
- No need to write script from scratch...
- ... use ParaView to prototype.





Creating a Script

- No need to write script from scratch...
- ... use ParaView to prototype.





Creating a Script

Now remove the reader and add an *in situ* source...

```
try: paraview.simple
except: from paraview.simple import *

psim_0001 = XMLPolyDataReader( FileName=['psim_0001.vtp'] )
psim_0001.PointArrayStatus = ['pmom', 'pfrc', 'mass']

RenderView1 = GetRenderView()
...
```



```
try: paraview.simple
except: from paraview.simple import *

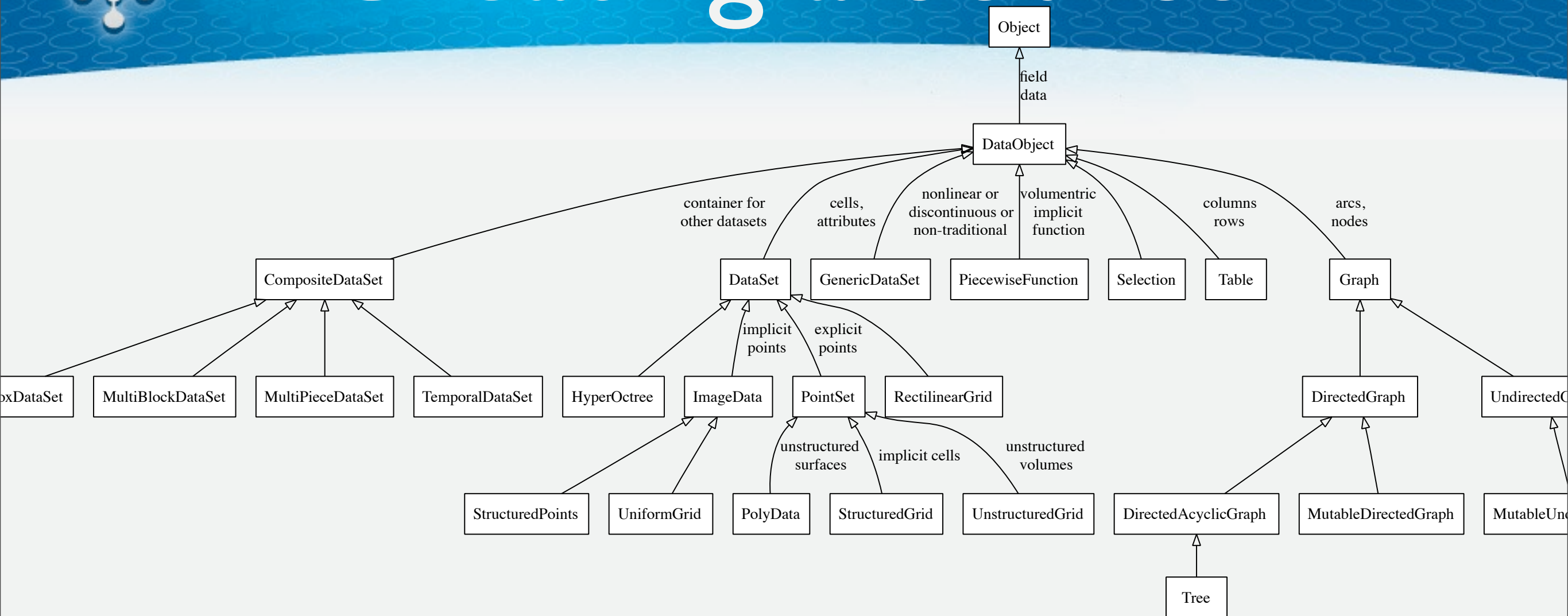
psim_0001 = servermanager.server.DotSource()

RenderView1 = GetRenderView()
...
```

... which we'll write next.



Creating a Source



- Find state you wish to expose in simulation.
- Determine appropriate subclass of `vtkDataObject` and thus `vtkAlgorithm` for your source.
- Add methods to populate dataset from simulation.



Creating a Source

From the Fortran module `simulation/particle_m.f90`
we identify some state to expose:

```
! -----  
! PARTICLE STATE  
! -----  
module particle_m  
  use parallel_m  
  
  ! -----  
  integer num_particles ! number of particles in the simulation  
  integer num_neighbors ! number of nearest neighbors to track  
  real terminal_force ! the smallest inter-particle force we'll allow.  
  real terminal_velocity ! the smallest particle velocity we'll allow.  
  integer min_steps ! the smallest number of time steps we'll allow.  
  integer simstep ! the number of steps we've simulated  
  real, allocatable, target, public :: pxyz(:, :) ! particle coordinates  
  real, allocatable, target, public :: pmom(:, :) ! particle momenta  
  real, allocatable, target, public :: pfrc(:, :) ! particle forces  
  real, allocatable, target, public :: mass(:) ! particle masses
```



Creating a Source

Now create a `vtkAlgorithm` subclass and add an initialization method to `insitu/vtkDotSource.h`:

```
#include "vtkPolyDataAlgorithm.h"
class VTK_EXPORT vtkDotSource : public vtkPolyDataAlgorithm
{
public:
    static vtkDotSource* New();
    ...
    /// Prepare SimulationData using arrays from the simulation.
    virtual void Initialize( int num_particles, double* pxyz, double* mass,
                           double* pmom, double* pfrc, int local_id, int num_procs );
protected:
    virtual int RequestData( vtkInformation*, vtkInformationVector**, ...
    vtkPolyData* SimulationData; // References arrays owned by simulation.
};
```



Creating a Source

Implementing the source simply involves a shallow copy of the initialized object in `insitu/vtkDotSource.cxx`:

```
int vtkDotSource::RequestData( vtkInformation*,
    vtkInformationVector** inInfoVec, vtkInformationVector* ouInfoVec )
{
    if ( this->SimulationData ) {
        vtkInformation* ouInfo0 = ouInfoVec->GetInformationObject( 0 );
        if ( ! ouInfo0 ) return 0;
        vtkDataObject* ouData0 = ouInfo0->Get(vtkDataObject::DATA_OBJECT());
        if ( ouData0 ) {
            ouData0->ShallowCopy( this->SimulationData );
            ouData0->GetInformation()->Set( vtkDataObject::DATA_TIME_STEPS(),
                &this->SimulationTime, 1 );
        }
    }
    return 1;
}
```




Example, Part II

Finish & Polish