The ParaView Tutorial

Version 5.6

Kenneth Moreland
Sandia National Laboratories
kmorel@sandia.gov

Sandia National Laboratories

U.S. DEPARTMENT OF
ENERGY

Contents

List of Exercises

Chapter 1

Introduction

ParaView is an open-source application for visualizing two- and three-dimensional data sets. The size of the data sets ParaView can handle varies widely depending on the architecture on which the application is run. The platforms supported by ParaView range from single-processor workstations to multiple-processor distributed-memory supercomputers or workstation clusters. Using a parallel machine, ParaView can process very large data sets in parallel and later collect the results. To date, ParaView has been demonstrated to process billions of unstructured cells and to process over a trillion structured cells. ParaView's parallel framework has run on over 100,000 processing cores.

ParaView's design contains many conceptual features that make it stand apart from other scientific visualization solutions.

- An open-source, scalable, multi-platform visualization application.

- Support for distributed computation models to process large data sets.

- An open, flexible, and intuitive user interface.

- An extensible, modular architecture based on open standards.

- A flexible BSD 3-clause license.

- Commercial maintenance and support.

ParaView is used by many academic, government, and commercial institutions all over the world. Para-View's open license makes it impossible to track exactly how many users ParaView has, but it is thought to be many thousands large based on indirect evidence. For example, ParaView is downloaded roughly 100,000 times every year. ParaView also won the HPCwire Readers' Choice Award in 2010 and 2012 and HPCwire Editors' Choice Award in 2010 for Best HPC Visualization Product or Technology.
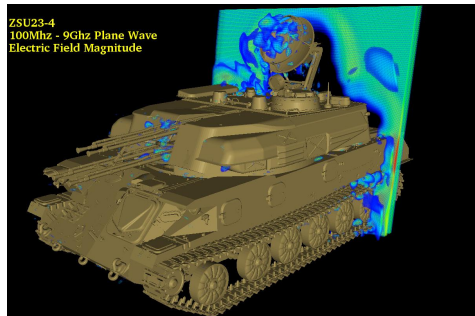
ZSU23-4 Russian Anti-Aircraft vehicle being hit by a planar wave. Image courtesy of Jerry Clarke, US Army Research Laboratory.
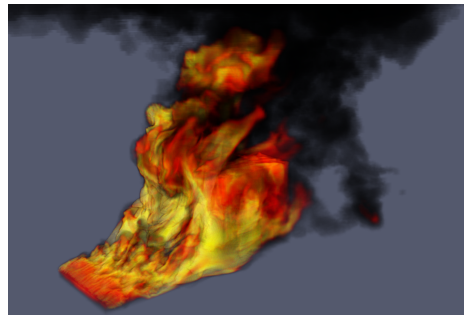
A loosely coupled SIERRA-Fuego-Syrinx-Calore simulation with 10 million unstructured hexahedra cells of objects-in-crosswind fire.
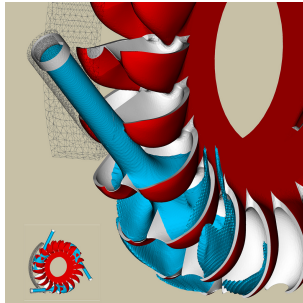
Simulation of a Pelton turbine. Image courtesy of the Swiss National Supercomputing Centre
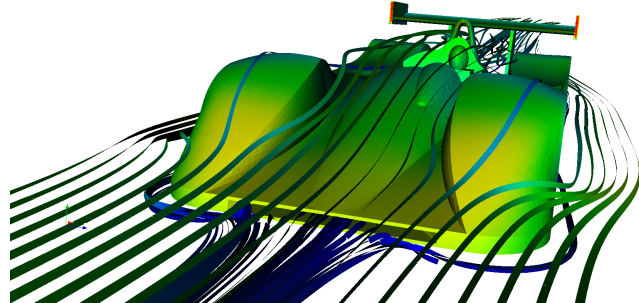
Airflow around a Le Mans Race car. Image courtesy of Renato N. Elias, NACAD/COPPE/UFRJ, Rio de Janerio, Brazil

As demonstrated in these visualizations, ParaView is a general-purpose tool with a wide breadth of applications. In addition to scaling from small to large data, ParaView provides many general-purpose visualization algorithms

as well as some specific to particular scientific disciplines. Furthermore, the ParaView system can be extended with custom visualization algorithms.
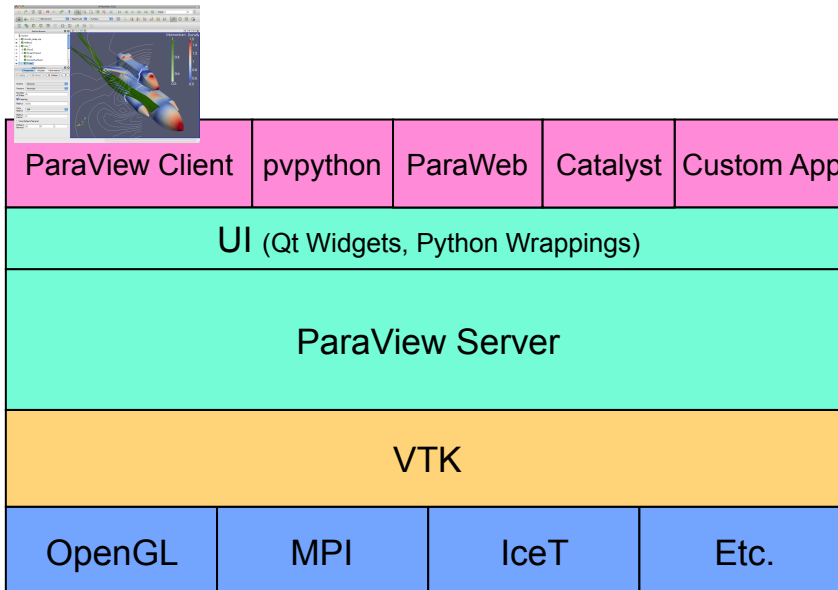
The application most people associate with ParaView is really just a small client application built on top of a tall stack of libraries that provide ParaView with its functionality. Because the vast majority of ParaView features are implemented in libraries, it is possible to completely replace the ParaView GUI with your own custom application, as demonstrated in the following figure. Furthermore, ParaView comes with a **pvpython** application that allows you to automate the visualization and post-processing with Python scripting.

Available to each ParaView application is a library of user interface components to maximize code sharing between them. A **ParaView Server** library provides the abstraction layer necessary for running parallel, interactive visualization. It relieves the client application from most of the issues concerning if and how ParaView is running in parallel. The **Visualization Toolkit** (**VTK**) provides the basic visualization and rendering algorithms. VTK incorporates several other libraries to provide basic functionalities such as rendering, parallel processing, file I/O, and parallel rendering. Although this tutorial demonstrates using ParaView through the ParaView client application, be aware that the modular design of ParaView allows for a great deal of flexibility and customization.

1.1 Development and Funding

The ParaView project started in 2000 as a collaborative effort between Kitware Inc. and Los Alamos National Laboratory. The initial funding was provided by a three year contract with the US Department of Energy ASCI Views program. The first public release, ParaView 0.6, was announced in October 2002. Development of ParaView continued through collaboration of Kitware Inc. with Sandia National Laboratories, Los Alamos National Laboratories, the Army Research Laboratory, and various other academic and government institutions.

In September 2005, Kitware, Sandia National Labs and CSimSoft started the development of ParaView 3.0. This was a major effort focused on rewriting the user interface to be more user friendly and on developing a quantitative analysis framework. ParaView 3.0 was released in May 2007.

Since this time, ParaView development continues. ParaView 4.0 was released in June 2013 and introduced more cohesive GUI controls and better multiblock interaction. Subsequent releases also include the Catalyst library for in situ integration into simulation and other applications. ParaView 5.0 was released in January 2016 and provided a major update to the rendering system. The new rendering takes advantage of OpenGL 3.2 features to provide huge performance improvements. Subsequent releases also added support for ray cast rendering with the OSPRay library.

Development of ParaView continues today. Sandia National Laboratories continues to fund ParaView development through the ASC project. ParaView is part of the SciDAC Scalable Data Management, Analysis, and Visualization (SDAV) Institute Toolkit (sdav-scidac.org). The US Department of Energy also funds ParaView through Los Alamos National Laboratories and various SBIR projects and other contracts. The US National Science Foundation also often funds ParaView through SBIR projects. Other institutions also have ParaView support contracts: Electricity de France, Mirarco, and oil industry customers. Also, because ParaView is an open source project, other institutions such as the Swiss National Supercomputing Centre contribute back their own development.

1.2 Basics of Visualization

```
0265640 132304 133732 032051 037334 024721 015013 052226 001662
0265660 025537 064663 054606 043244 074076 124153 135216 126614
0265700 144210 056426 044700 042650 165230 137037 003655 006254
0265720 134453 124327 176005 027034 107614 170774 073702 067274
0265740 072451 007735 147620 061064 157435 113057 155356 114603
0265760 107204 102316 171451 046040 120223 001774 030477 046673
0266000 171317 116055 155117 134444 167210 041405 147127 050505
0266020 004137 046472 124015 134360 173550 053517 044635 021135
0266040 070176 047705 113754 175477 105532 076515 177366 056333
0266060 041023 074017 127113 003214 037026 037640 066171 123424
0266100 067701 037406 140000 165341 072410 100032 125455 056646
0266120 006716 071402 055672 132571 105645 170073 050376 072117
0266140 024451 007424 114200 077733 024434 012546 172404 102345
0266160 040223 050170 055164 164634 047154 126525 112514 032315
0266200 016041 176055 042766 025015 176314 017234 110060 014515
0266220 117156 030746 154234 125001 151144 163706 136237 164376
0266240 137055 062276 161755 115466 005322 132567 073216 002655
0266260 171466 126161 117155 065763 016177 014460 112765 055527
0266300 003767 175367 104754 036436 172172 150750 043643 145410
0266320 072074 000007 040627 070652 173011 002151 125132 140214
0266340 060115 014356 015164 067027 120206 070242 033065 131334
0266360 170601 170106 040437 127277 124446 136631 041462 116321
0266400 020243 005602 004146 121574 124651 006634 071331 102070
0266420 157504 160307 166330 074251 024520 114433 167273 030635
0266440 133614 106171 144160 010652 007365 026416 160716 100413
0266460 026630 007210 000630 121224 076033 140764 000737 003276
0266500 114060 042647 104475 110537 066716 104754 075447 112254
0266520 030374 144251 077734 015157 002513 173526 035531 150003
0266540 146207 015135 024446 130101 072457 040764 165513 156412
0266560 166410 067251 156160 106406 136770 030516 064740 022032
0266600 142166 123707 175121 071170 076357 037233 031136 015232
0266620 075074 016744 044055 102230 110063 033350 052765 172463
```

Momentum Density
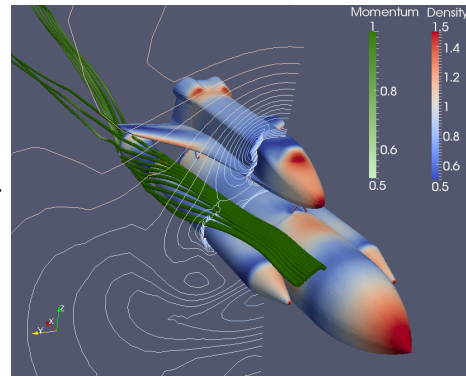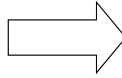
Put simply, the process of visualization is taking raw data and converting it to a form that is viewable and understandable to humans. This allows us to get a better cognitive understanding of our data. Scientific visualization is specifically concerned with the type of data that has a well defined representation in 2D or 3D space. Data that comes from simulation meshes and scanner data is well suited for this type of analysis.

There are three basic steps to visualizing your data: reading, filtering, and rendering. First, your data must be read into ParaView. Next, you may apply any number of **filters** that process the data to generate, extract, or derive features from the data. Finally, a viewable image is rendered from the data.

ParaView was designed primarily to handle data with spatial representation. Thus the primary **data types** used in ParaView are meshes.
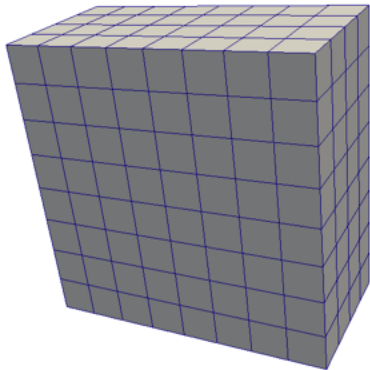
Uniform Rectilinear (Image Data)
A uniform rectilinear grid is a one- two- or three- dimensional array of data. The points are orthonormal to each other and are spaced regularly along each direction.
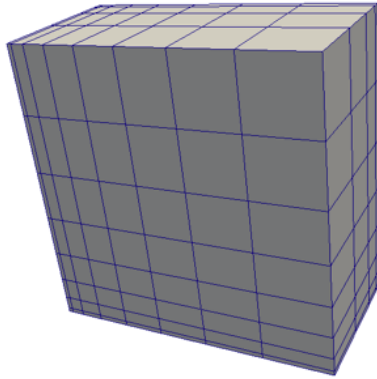
Non-uniform Rectilinear (Rectilinear Grid)

Similar to the uniform rectilinear grid except that the spacing between points may vary along each axis.
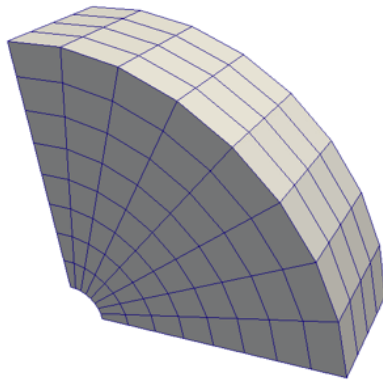
Curvilinear (Structured Grid)

Curvilinear grids have the same topology as rectilinear grids. However, each point in a curvilinear grid can be placed at an arbitrary coordinate (provided that it does not result in cells that overlap or self intersect). Curvilinear grids provide the more compact memory footprint and implicit topology of the rectilinear grids, but also allow for much more variation in the shape of the mesh.
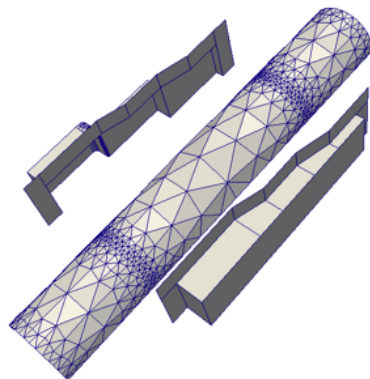
Polygonal (Poly Data)

Polygonal data sets are composed of points, lines, and 2D polygons. Connections between cells can be arbitrary or non-existent. Polygonal data represents the basic rendering primitives. Any data must be converted to polygonal data before being rendered (unless volume rendering is employed), although ParaView will automatically make this conversion.
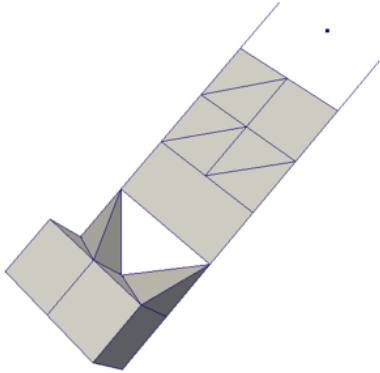
Unstructured Grid
Unstructured data sets are composed of points, lines, 2D polygons, 3D tetrahedra, and nonlinear cells. They are similar to polygonal data except that they can also represent 3D tetrahedra and nonlinear cells, which cannot be directly rendered.

In addition to these basic data types, ParaView also supports **multi-block** data. A basic multi-block data set is created whenever data sets are grouped together or whenever a file containing multiple blocks is read. ParaView also has some special data types for representing **Hierarchical Adaptive Mesh Refinement** (**AMR**), **Hierarchical Uniform AMR**, **Octree**, **Tablular**, and **Graph** type data sets.

1.3 More Information

There are many places to find more information about ParaView. The manual, titled *The ParaView Guide*, is available for purchase as a hard copy or can be downloaded for free from http://www.paraview.org/paraview-guide.

The ParaView web page, www.paraview.org, is also an excellent place to find more information about ParaView. From there you can find helpful links to mailing lists, Wiki pages, and frequently asked questions as well as information about professional support services.
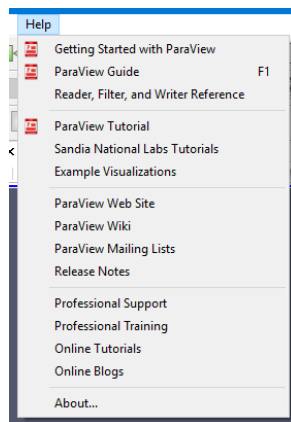
ParaView's Help menu is a good place to start to find useful information and contains many links to documentation and training materials. The Help menu has entries that directly bring up the aforementioned *ParaView Guide* and web pages. The Help menu also provides several training materials including a quick *Getting Started with ParaView* guide, multiple tutorials (including this one), and some example visualizations.

Chapter 2

Basic Usage

Let us get started using ParaView. In order to follow along, you will need your own installation of ParaView. Specifically, this document is based off of ParaView version 5.6. If you do not already have ParaView 5.6, you can download a copy from www.paraview.org (click on the download link). ParaView launches like most other applications. On Windows, the launcher is located in the start menu. On Macintosh, open the application bundle that you installed. On Linux, execute `paraview` from a command prompt (you may need to set your path).

The examples in this tutorial also rely on some data that is available at http://www.paraview.org/Wiki/The_ParaView_Tutorial. You may install this data into any directory that you like, but make sure that you can find that directory easily. Any time the tutorial asks you to load a file it will be from the directory you installed this data in.

2.1   User Interface

Menu Bar

Toolbars

Pipeline Browser

3D View

Properties Panel

Advanced Toggle
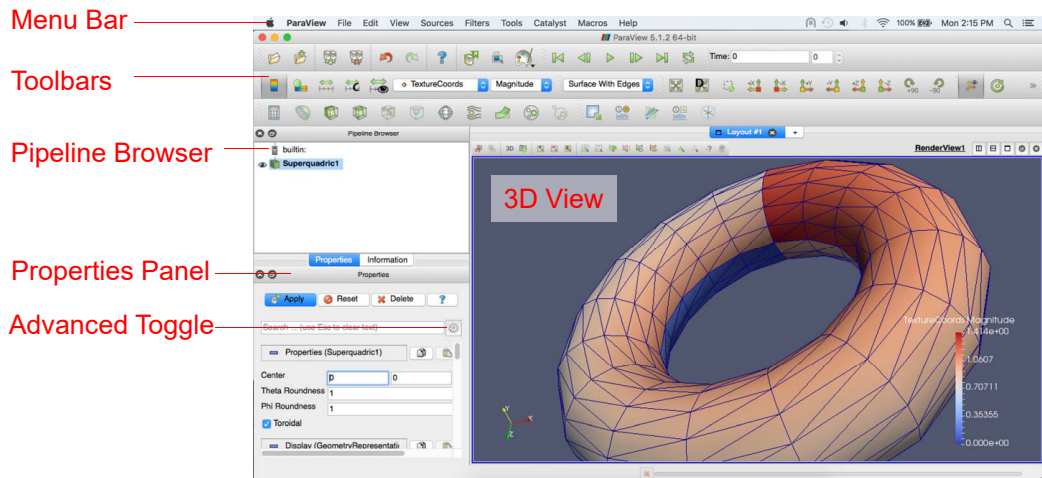
The ParaView GUI conforms to the platform on which it is running, but on all platforms it behaves basically the same. The layout shown here is the default layout given when ParaView is first started. The GUI comprises the following components.

Menu Bar As with just about any other program, the menu bar allows you to access the majority of features.

Toolbars The toolbars provide quick access to the most commonly used features within ParaView.

Pipeline Browser ParaView manages the reading and filtering of data with a pipeline. The pipeline browser allows you to view the pipeline structure and select pipeline objects. The pipeline browser provides a convenient list of pipeline objects with an indentation style that shows the pipeline structure.

Properties Panel The properties panel allows you to view and change the parameters of the current pipeline object. On the properties panel is an advanced properties toggle that shows and hides advanced controls. The properties are by default coupled with an **Information** tab that shows a basic summary of the data produced by the pipeline object.

3D View The remainder of the GUI is used to present data so that you may view, interact with, and explore your data. This area is initially

populated with a 3D view that will provide a geometric representation of the data.

Note that the GUI layout is highly configurable, so that it is easy to change the look of the window. The toolbars can be moved around and even hidden from view. To toggle the use of a toolbar, use the View → Toolbars submenu. The pipeline browser and properties panel are both **dockable** windows. This means that these components can be moved around in the GUI, torn off as their own floating windows, or hidden altogether. These two windows are important to the operation of ParaView, so if you hide them and then need them again, you can get them back with the View menu.

2.2 Sources

There are two ways to get data into ParaView: read data from a file or generate data with a **source** object. All sources are located in the Sources menu. Sources can be used to add annotation to a view, but they are also very handy when exploring ParaView's features.

Exercise 2.1: Creating a Source

Let us start with a simple one. Go to the Sources menu, open the Geometric Shapes submenu, and select Cylinder. Once you select the Cylinder item you will notice that an item named Cylinder1 is added to and selected in the pipeline browser. You will also notice that the properties panel is filled with the properties for the cylinder source. Click the Apply button ⟨ Apply ⟩ to accept the default parameters.

Once you click Apply, the cylinder object will be displayed in the 3D view window on the right. ♦

2.3 Basic 3D Interaction

Now that we have created our first simple visualization, we want to interact with it. There are many ways to interact with a visualization in ParaView. We start by exploring the data in the 3D view.

Exercise 2.2: Interacting with a 3D View

This exercise is a continuation of Exercise 2.1. You will need to finish that
exercise before beginning this one.

You can manipulate the cylinder in the 3D view by dragging the mouse
over the 3D view. Experiment with dragging different mouse buttons—left,
middle, and right—to perform different rotate, pan, and zoom operations.
Also try using the buttons in conjunction with the shift and ctrl modifier
keys. Additionally you can hold down the x, y, or z key while you drag the
mouse to constrain movement along the x, y, or z axis.

ParaView contains a couple of toolbars to help with camera manipula-
tions. The first toolbar, the Camera Controls toolbar, shown here, provides
quick access to particular camera views. The leftmost button performs a
reset camera such that it maintains the same view direction but repositions
the camera such that the entire object can be seen. The second button
performs a **zoom to data**. It behaves very much like reset camera except
that instead of positioning the camera to see all data, the camera is placed to
look specifically at the data currently selected in the pipeline browser. You
currently only have one object in the pipeline browser, so right now reset
camera and zoom to data will perform the same operation.

The next button in the camera controls toolbar allows you to select
a rectangular region of the screen to zoom to (a **rubber-band zoom**). The
following six buttons, starting with , reposition the camera to view the
scene straight down one of the global coordinate's axes in either the positive
or negative direction. The rightmost two buttons rotate the view
either clockwise or counterclockwise. Try playing with these controls now.

The second toolbar controls the location of the center of rotation and the
visibility of the orientation axes. The rightmost button allows you to pick
the **center of rotation**. Try clicking that button then clicking somewhere
on the cylinder. If you then drag the left button in the 3D view, you will
notice that the cylinder now rotates around this new point. The next button
to the left replaces the center of rotation to the center of the object.

The next button to the left ⟳ shows or hides axes drawn at the center of rotation. (You probably will not notice the effects when the center of rotation is at the center of the cylinder because the axes will be hidden by the cylinder. Use the pick center of rotation ⟳ again and you should be able to see the effects.) The final leftmost button ⚘ toggles showing the **orientation axes**, the always-viewable axes in the lower left corner of the 3D view. ♦

2.4 Modifying Visualization Parameters

Although interactive 3D controls are a vital part of visualization, an equally important ability is to modify the parameters of the data processing and display. ParaView contains many GUI components for modifying visualization parameters, which we will begin to explore in the next exercise.

Exercise 2.3: Modifying Visualization Parameters

This exercise is a continuation of Exercise 2.2. You will need to finish that exercise before beginning this one.

You surely noticed that ParaView creates not a real cylinder but rather an approximation of a cylinder using polygonal **facets**. The default parameters for the cylinder source provide a very coarse approximation of only six facets. (In fact, this object looks more like a prism than a cylinder.) If we want a better representation of a cylinder, we can create one by increasing the Resolution parameter. The Resolution parameters, like all other parameters for the cylinder object, are located in the properties panel under the Apply button when the cylinder object is selected in the pipeline browser.
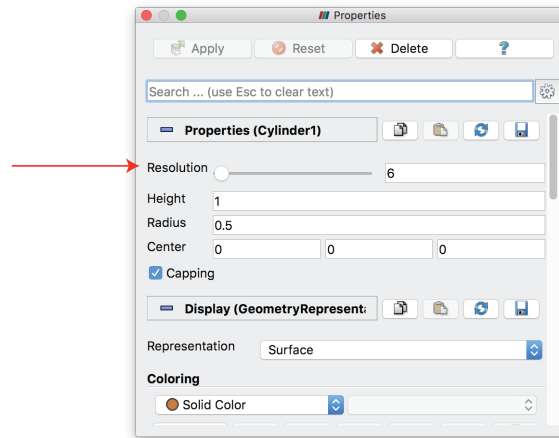
Using either the slider or text edit, increase the resolution to 50 or more. Notice that the Apply button [Apply] become colored again. This is because changes you make to the object properties are not immediately enacted. The highlighted button is a reminder that the parameters of one or more pipeline objects are "out of sync" with the data that you are viewing. Hitting the Apply button will accept these changes whereas hitting the Reset button [Reset] will revert the options back to the last time they were applied. Hit the Apply button now. The resolution is changed so that it is virtually indistinguishable from a true cylinder.

If your work has you creating cylinder sources frequently and you find yourself modifying Resolution or other parameters to some value other than the default each time, you can save your preferred default parameters by hitting the save parameters [💾] button. Once you hit the [💾] button, ParaView will remember your preferences for objects of that type and use those parameters when you create future objects. Conversely, if you have changed the parameters and want to reset them to the "factory default," you can click the restore parameters [🔄] button. As we will see in future exercises, we can have multiple visualization objects open at once. To copy parameters from one object to another, use the copy [📋] and paste [📋] parameters buttons.
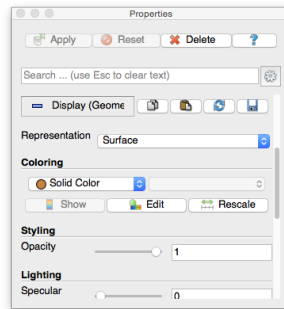
If you scroll down the properties panel, you will notice a set of Display properties. Try these options now by clicking on the Edit 🔲 button under Coloring to select a new color for the cylinder. (This button is also replicated in the toolbar.) You may notice that you do not need to hit Apply for display properties.
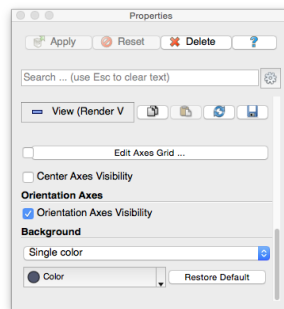
If you scroll down further yet to the bottom of the properties panel, you will notice a set of View properties. Use the view properties to turn on the Axes Grid.

By default many of the lesser used display properties are hidden. The **advanced properties** toggle 🔧 can be used to show or hide these extra parameters. There is also a search box at the top of the properties panel that can be used to quickly find a property. Try typing specular into this search box now. Under the display properties you should see an option named Specular. This controls the intensity of the specular highlight seen on shiny objects. Set this parameter to 1 to make the cylinder shiny.

Most objects have similar display and view properties. Here are some other common tricks you can do with most objects using parameters available in the properties panel and that you can try now.

- Show 3D axes at the borders of the object containing rulers showing the physical distance in each direction by clicking the Axes Grid checkbox under the View options.

- Make objects transparent by changing their Opacity parameter. An opacity parameter of 1 is completely opaque, a parameter of 0 is completely invisible, and values in between are varying degrees of see through.

♦

From the previous exercises you have noted that some visualization operations (but not all) require pressing the Apply button before seeing the effect of the change. This apply button serves an important function. When visualizing large data, which ParaView is designed to do, simple actions like creating an object or changing a parameter can take a long time. Thus this two phased approach allows you to establish all the visualization parameters for a particular action before enacting an operation (by hitting Apply). However, when dealing with small data, operations complete near instantaneously, so the process of hitting Apply becomes redundant. In these cases, you may wish to turn on auto apply.

Exercise 2.4: Toggle Auto Apply

Find the auto apply button 🔲 in the top toolbar. This is a toggle button. Click it now and note that it stays depressed.

While auto apply is on, it is no longer necessary to hit the Apply button. Try changing the Resolution of the cylinder source as you did in Exercise 2.3 (or create a new source if your cylinder is no longer available). Note that as soon as you make the change, the visualization is updated.

You can turn off auto apply by clicking the toolbar button 🔲 again. You can complete the rest of these exercises with auto apply either on or off. The instructions will assume that auto apply is off and prompt you to hit the Apply button. If you have auto apply on, ignore these instructions.   ♦

As you would expect, ParaView allows you to control the color of many elements. In many cases the changing the color of one element necessitates the changing of another. For example, if changing the background to a light color, it is important to change text on that background to a dark color. Otherwise the text will be unreadable. To help manage sets of interdependent

colors, ParaView supports the idea of color palettes. You can easily change the view's color palette using the load color palette button 🎨 in the toolbar.

Exercise 2.5: Changing the Color Palette

Make sure the orientation axes is shown in the lower left corner. This is toggled with the 🧭 button as described in Exercise 2.2. Note that the orientation axis has the labels "X," "Y," and "Z."

Find the load color palette button 🎨 in the top toolbar. Click that button to get a pull down menu of available palettes. Experiment with different palettes. Observe that both the background color and the labels in the orientation axes change. ♦

The colors used for the color palettes are part of ParaView's settings. You can see and set all of these colors in the Edit → Settings (ParaView → Preferences on the Mac) under the Color Palette tab. You can also get to the color palette settings by clicking on the color palette button 🎨 and selecting the Edit Current Palette... button.
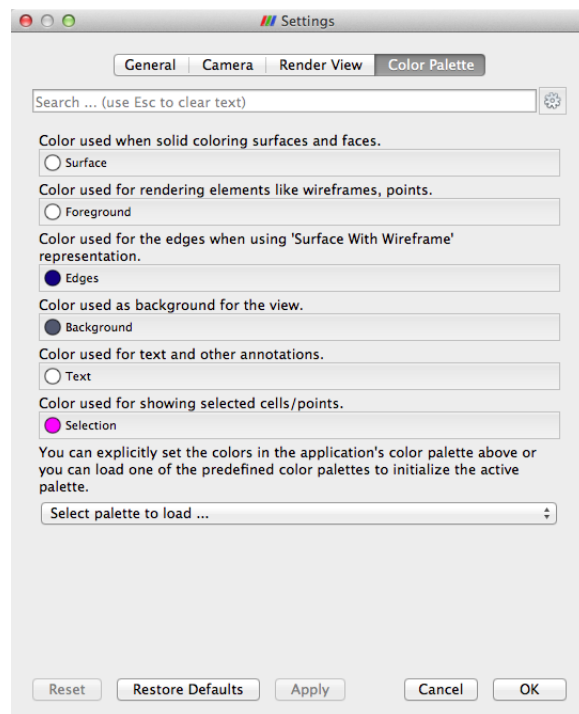
Now is a good time to note the undo ↶ and redo ↷ buttons in the toolbar. Visualizing your data is often an exploratory process, and it is often helpful to revert back to a previous state. You can even undo back to the point before your data were created and redo again.

Exercise 2.6: Undo and Redo

Experiment with the undo ↶ and redo ↷ buttons. If you have not done so, create and modify a pipeline object like what is done in Exercise 2.1. Watch how parameter changes can be reverted and restored. Also notice how whole pipeline objects can be destroyed and recreated.

There are also undo camera and redo camera buttons. These allow you to go back and forth between camera angles that you have made so that you do not have to worry about errant mouse movements ruining that perfect view. Move the camera around and then use these buttons to revert and restore the camera angle.   ♦

We are done with the cylinder source now. We can delete the pipeline object by making sure the cylinder is selected in the pipeline browser and hitting delete ✖ Delete in the properties panel.

2.5   Loading Data

Now that we have had some practice using the ParaView GUI, let us load in some real data. As you would expect, the Open command is the first one off of the File menu, and there is also toolbar button 📂 for opening a file. ParaView currently supports about 220 distinct file formats, and the list grows as more types get added. To see the current list of supported files, invoke the Open command and look at the list of files in the Files of type chooser box.
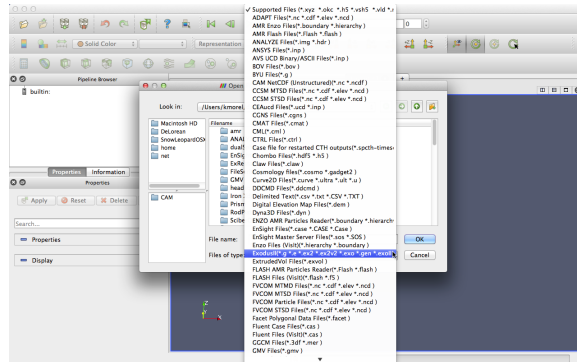
ParaView's modular design allows for easy integration of new VTK readers into ParaView. Thus, check back often for new file formats. If you are looking for a file reader that does not seem to be included with ParaView, check in with the ParaView mailing list (paraview@paraview.org). There are many file readers included with VTK but not exposed within ParaView that could easily be added. There are also many readers created that can plug into the VTK framework but have not been committed back to VTK; someone may have a reader readily available that you can use.

Exercise 2.7: Opening a File

Let us open our first file now. Click the **Open** toolbar button (or menu item) 📂. Note that ParaView uses a custom file browser, which provides several convenience features. On the left side of the file browser dialog are a pair of boxes containing lists of directories, which provide quick access to files in common directories. The top left list contains a list of common data directories on your system. The bottom left list, which is initially empty, is filled with directories from which you have recently loaded files. Double click on the **Examples** directory listed in the top left box. This is a directory created by the ParaView installation that contains the files we use in this tutorial.
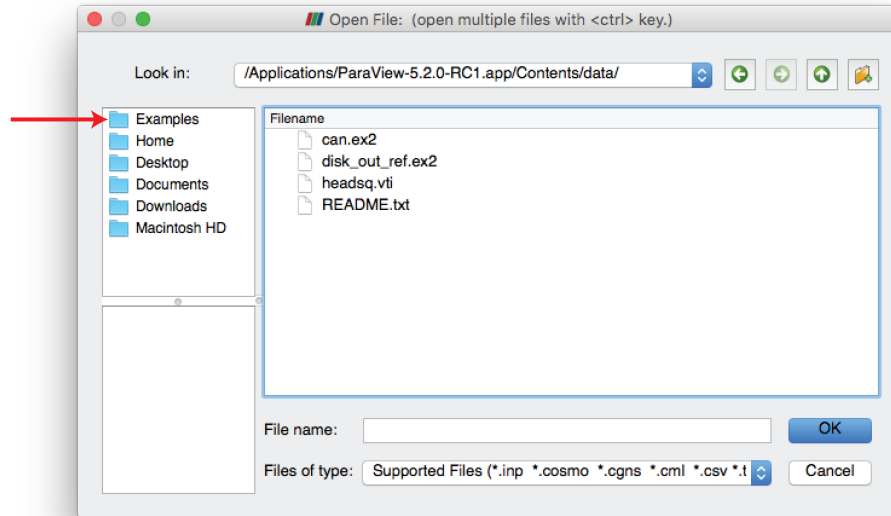
Open the file disk_out_ref.ex2. Note that opening a file is a two step process, so you do not see any data yet. Instead, you see that the properties panel is populated with several options about how we want to read the data.
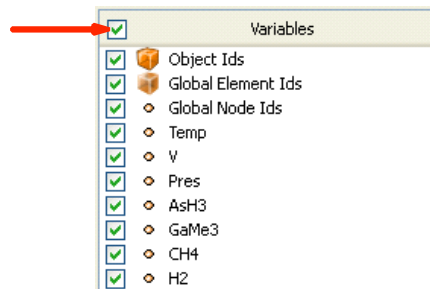
Click the checkbox in the header of the variable list to turn on the loading of all the variables. This is a small data set, so we do not have to worry about loading too much into memory. Once all of the variables are selected, click Apply to load all of the data. When the data are loaded you will see that the geometry looks like a cylinder with a hollowed out portion in one end. These data are the output of a simulation for the flow of air around a heated and spinning disk. The mesh you are seeing is the air around the disk (with the cylinder shape being the boundary of the simulation). The hollow

area in the middle is where the heated disk would be were it meshed for the simulation.   ♦

Most of the time ParaView will be able to determine the appropriate method to read your file based on the file extension and underlying data, as was the case in Exercise 2.7. However, with so many file formats supported by ParaView there are some files that cannot be fully determined. In this case, ParaView will present a dialog box asking what type of file is being loaded. The following image is an example from opening a netCDF file, which is a generic file format for which ParaView has many readers for different conventions.
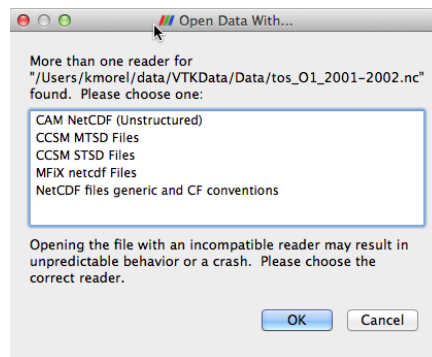
Before we continue on to filtering the data, let us take a quick look at some of the ways to represent the data. The most common parameters for representing data are located in a pair of toolbars. (They can also be found in the Display group of the properties panel.)
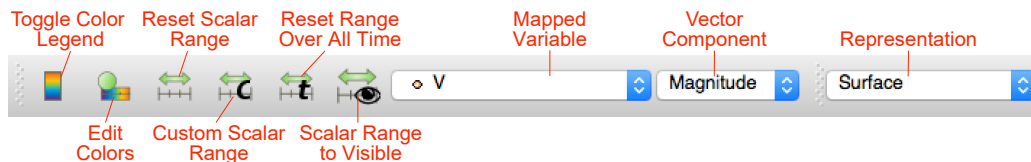
Exercise 2.8: Representation and Field Coloring

Play with the data representation a bit. Make sure disk_out_ref.ex2 is selected in the pipeline browser. (If you do not have the data loaded, repeat Exercise 2.7.) Use the variable chooser to color the surface by the Pres variable. To see the structure of the mesh, change the representation to Surface With

Edges. You can view both the cell structure and the interior of the mesh
with the Wireframe representation.

♦
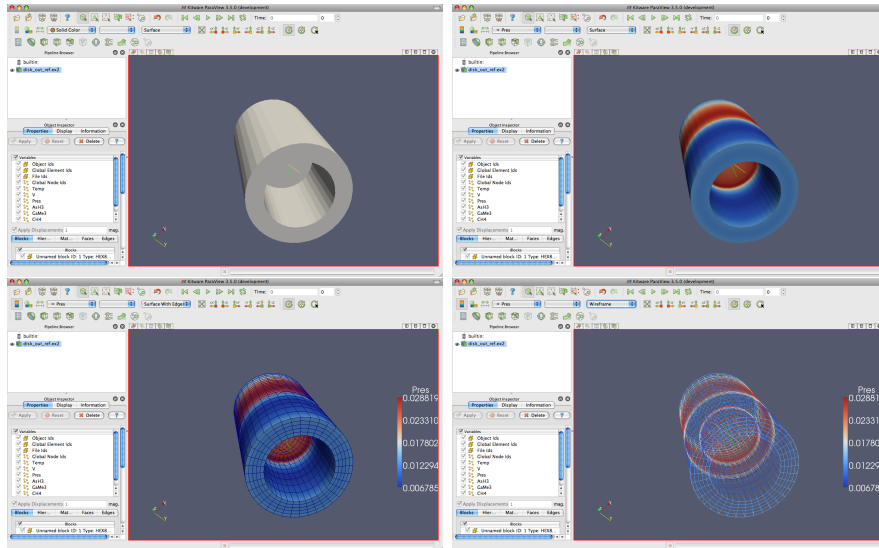
2.6   Filters

We have now successfully read in some data and gleaned some information
about it. We can see the basic structure of the mesh and map some data
onto the surface of the mesh. However, as we will soon see, there are many
interesting features about these data that we cannot determine by simply
looking at the surface of these data. There are many variables associated
with the mesh of different types (scalars and vectors). And remember that
the mesh is a solid model. Most of the interesting information is on the
inside.

We can discover much more about our data by applying **filters**. Filters
are functional units that process the data to generate, extract, or derive
features from the data. Filters are attached to readers, sources, or other filters
to modify its data in some way. These filter connections form a **visualization
pipeline**. There are a great many filters available in ParaView. Here are
the most common, which are all available by clicking on the respective icon
in the filters toolbar.

Calculator Evaluates a user-defined expression on a per-point or per-cell basis.

Contour Extracts the points, curves, or surfaces where a scalar field is equal to a user-defined value. This surface is often also called an **isosurface**.

Clip Intersects the geometry with a half space. The effect is to remove all the geometry on one side of a user-defined plane.

Slice Intersects the geometry with a plane. The effect is similar to clipping except that all that remains is the geometry where the plane is located.

Threshold Extracts cells that lie within a specified range of a scalar field.

Extract Subset Extracts a subset of a grid by defining either a volume of interest or a sampling rate.

Glyph Places a **glyph**, a simple shape, on each point in a mesh. The glyphs may be oriented by a vector and scaled by a vector or scalar.

Stream Tracer Seeds a vector field with points and then traces those seed points through the (steady state) vector field.

Warp (vector) Displaces each point in a mesh by a given vector field.

Group Datasets Combines the output of several pipeline objects into a single multi block data set.

Extract Level Extract one or more items from a multi block data set.

These eleven filters are a small sampling of what is available in ParaView. In the Filters menu are a great many more filters that you can use to process your data. ParaView currently exposes more than one hundred filters, so to make them easier to find the Filters menu is organized into submenus.
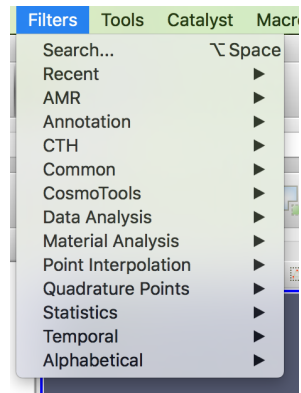
These submenus are organized as follows.

Recent The list of most recently used filters sorted with the most recently used filters on top.

AMR A set of filters designed specifically for data in an adaptive mesh refinement (AMR) structure.

Annotation Filters that add annotation (such as text information) to the visualization.

CTH Filters used to process results from a CTH simulation.

Common The most common filters. This is the same list of filters available in the filters toolbar and listed previously.

Data Analysis The filters designed to retrieve quantitative values from the data. These filters compute data on the mesh, extract elements from the mesh, or plot data.

Material Analysis Filters for processing data from volume fractions of materials.

Point Interpolation Filters that take an unstructured collection of points in space without cells connecting them and estimate the field interpolation between them.

Quadrature Points Filters to support simulation data given as integration points that can be used for numerical integration with Gaussian quadrature.

Statistics This contains filters that provide descriptive statistics of data, primarily in tabular form.

Temporal Filters that analyze or modify data that changes over time. All filters can work on data that changes over time because they are executed on each time snapshot. However, filters in this category will retrieve the available time extents and examine how data changes over time.

Alphabetical An alphabetical listing of all the filters available. If you are not sure where to find a particular filter, this list is guaranteed to have it. There are also many filters that are not listed anywhere but in this list.

Searching through these lists of filters, particularly the full alphabetical list, can be cumbersome. To speed up the selection of filters, you should use the **quick launch** dialog. Pressing the ctrl and space keys together on Windows or Linux or the alt and space keys together on Macintosh, ParaView brings up a small, lightweight dialog box like the one shown here.
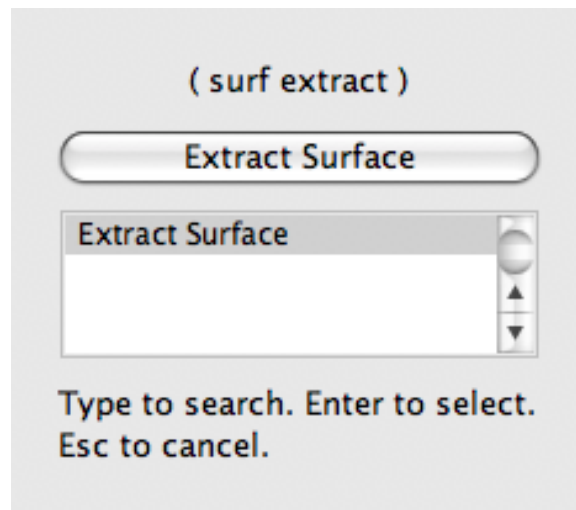
Type in words or word fragments that are contained in the filter name, and the box will list only those sources and filters that match the terms. Hit enter to add the object to the pipeline browser. Press Esc a couple of times to cancel the dialog.

You have probably noticed that some of the filters are grayed out. Many filters only work on a specific types of data and therefore cannot always be used. ParaView disables these filters from the menu and toolbars to indicate (and enforce) that you cannot use these filters.

Throughout this tutorial we will explore many filters. However, we cannot explore all the filters in this forum. Consult the Filters Menu chapter of ParaView's on-line or built-in help for more information on each filter.

Exercise 2.9: Apply a Filter

Let us apply our first filter. If you do not have the disk_out_ref.ex2 data loaded, do so know (Exercise 2.7). Make sure that **disk_out_ref.ex2** is selected in the pipeline browser and then select the contour filter 🍃 from the filter toolbar or **Filters** menu. Notice that a new item is added to the pipeline filter underneath the reader and that the properties panel updates to the parameters of the new filter. As with reading a file, applying a filter is a two step process (unless auto apply is enabled). After creating the filter you get a chance to modify the parameters (which you will almost always do) before applying the filter.
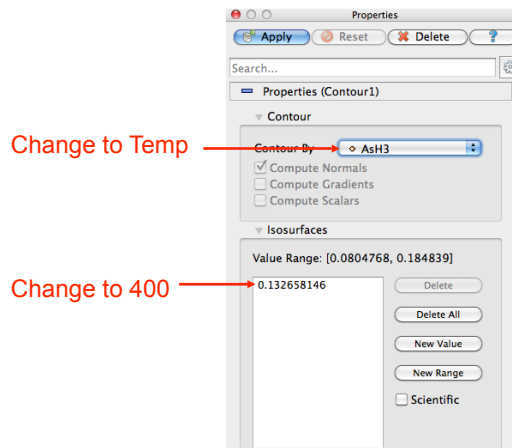
We will use the contour filter to create an isosurface where the temperature is equal to 400 K. First, change the **Contour By** parameter to the **Temp** variable. Then, change the isosurface value to **400**. Finally, hit [Apply]. You will see the isosurface appear inside of the volume. If **disk_out_ref.ex2** was still colored by pressure from Exercise 2.8, then the surface is colored by pressure to match.
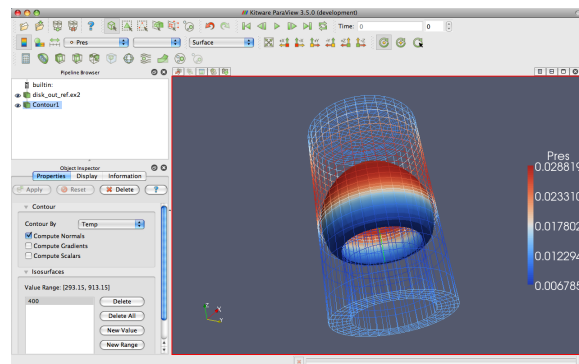
In the preceding exercise, we applied a filter that processed the data and gave us the results we needed. For most common operations, a single filter operation is sufficient to get the information we need. However, filters are of the same class as readers. That is, the general operations we apply to readers can also be applied to filters. Thus, you can apply one filter to the data that is generated by another filter. These readers and filters connected together form what we call a **visualization pipeline**. The ability to form visualization pipelines provides a powerful mechanism for customizing the visualization to your needs.

Let us play with some more filters. Rather than show the mesh surface in wireframe, which often interferes with the view of what is inside it, we will replace it with a cutaway of the surface. We need two filters to perform this task. The first filter will extract the surface, and the second filter will cut some away.

Exercise 2.10: Creating a Visualization Pipeline

The images and some of the discussion in this exercise assume you are starting with the state right after finishing Exercise 2.9. Finish that exercise before beginning this one.

Start by adding a filter that will extract the surfaces. We do that with the following steps.

1. Select disk_out_ref.ex2 in the pipeline browser.

2. From the menu bar, select Filters → Alphabetical → Extract Surface. Or bring up the quick launch (ctrl+space Win/Linux, alt+space Mac), type extract surface, and select that filter.

3. Hit the [⚙ Apply] button.
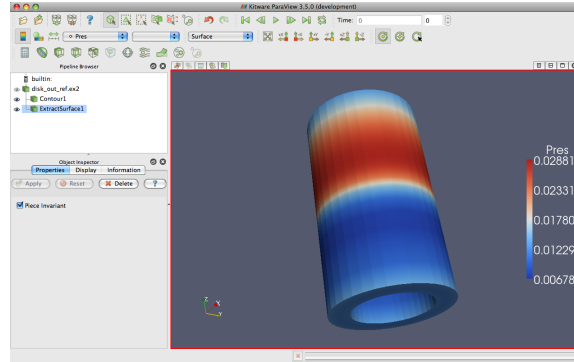
When you apply the Extract Surface filter, you will once again see the surface of the mesh. Although it looks like the original mesh, it is different in that this mesh is hollow whereas the original mesh was solid throughout.

If you were showing the results of the contour filter, you cannot see the contour anymore, but do not worry. It is still in there hidden by the surface. If you are showing the contour but you did not see any effect after applying the filter, you may have forgotten step one and applied the filter to the wrong object. If the ExtractSurface1 object is not connected directly to the disk_-out_ref.ex2, then this is what went wrong. If so, you can delete the filter and try again.

Now we will cut away the external surface to expose the internal structure and isosurface underneath (if you have one).

4. Verify that ExtractSurface1 is selected in the pipeline browser.

5. Create a clip filter 🔲 from the toolbar or Filters menu.

6. Uncheck the Show Plane checkbox [✓ Show Plane] in the properties panel.

7. Click the [⚙ Apply] button.
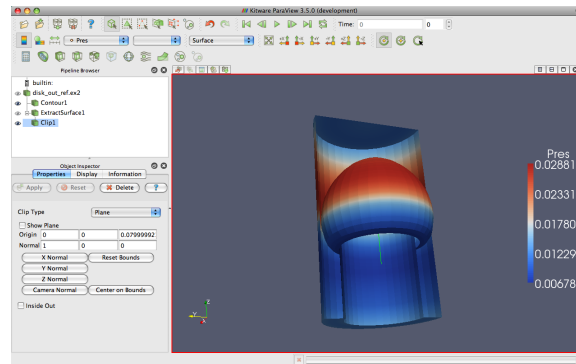
If you have a contour, you should now see the isosurface contour within a cutaway of the mesh surface. You will probably have to rotate the mesh to see the contour clearly.   ♦
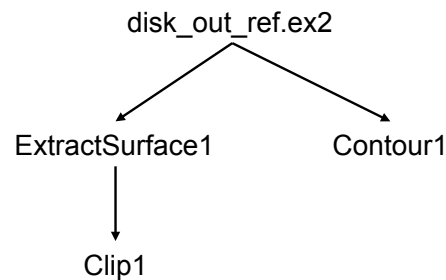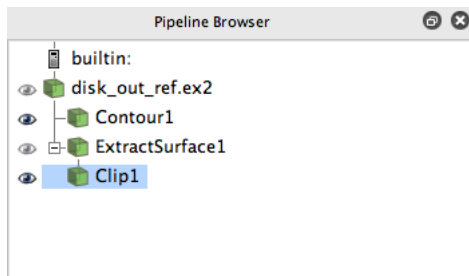
Now that we have added several filters to our pipeline, let us take a look at the layout of these filters in the pipeline browser. The pipeline browser provides a convenient list of pipeline objects that we have created. This makes it easy to select pipeline objects and change their **visibility** by clicking on the eyeball icons ⊙ next to them. But also notice the indentation of the entries in the list and the connecting lines toward the left. These features reveal the **connectivity** of the pipeline. It shows the same information as the traditional graph layout on the right, but in a much more compact space. The trouble with the traditional layout of pipeline objects is that it takes a lot of space, and even moderately sized pipelines require a significant portion of the GUI to see fully. The pipeline browser, however, is complete and compact.

2.7   Multiview

Occasionally in the pursuit of science we can narrow our focus down to one variable. However, most interesting physical phenomena rely on not one but many variables interacting in certain ways. It can be very challenging to present many variables in the same view. To help you explore complicated visualization data, ParaView contains the ability to present multiple views of data and correlate them together.

So far in our visualization we are looking at two variables: We are coloring with pressure and have extracted an isosurface with temperature. Although we are starting to get the feel for the layout of these variables, it is still difficult to make correlations between them. To make this correlation easier, we can use multiple views. Each view can show an independent aspect of the data and together they may yield a more complete understanding.

On top of each view is a small toolbar, and the buttons controlling the creating and deletion of views are located on the right side of this tool bar. There are four buttons in all. You can create a new view by splitting an existing view horizontally or vertically with the ⊞ and ⊟ buttons, respectively. The ✗ button deletes a view, whose space is consumed by an adjacent view. The ▢ temporarily fills view space with the selected view until ⊟ is pressed.

Exercise 2.11: Using Multiple Views

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to select Edit → Reset Session from the menu or hit ⟳ on the toolbar. This option deletes all of your current work and resets ParaView back to its initial state. It is roughly the equivalent of restarting ParaView.

First, we will look at one variable. We need to see the variable through the middle of the mesh, so we are going to clip the mesh in half.

1. Open the file disk_out_ref.ex2, load all variables, [Apply] (see Exercise 2.7).

2. Add the Clip filter 🧊 to disk_out_ref.ex2.

3. Uncheck the Show Plane checkbox ☑ Show Plane in the properties panel.

4. Click the [Apply] button.

5. Color the surface by pressure by changing the variable chooser (in the toolbar) from vtkBlockColors to Pres.

Now we can see the pressure in a plane through the middle of the mesh. We want to compare that to the temperature on the same plane. To do that, we create a new view to build another visualization.

6. Press the ▥ button.
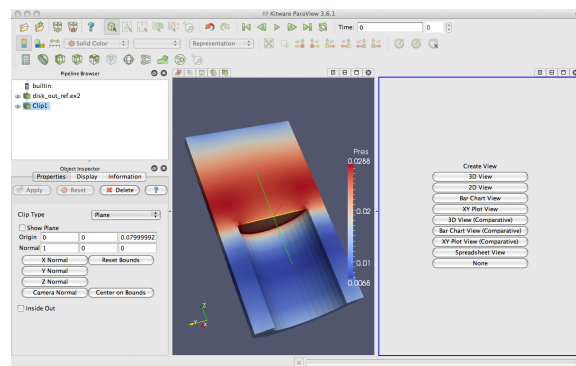
The current view is split in half and the right side is blank, ready to be filled with a new visualization. Notice that the view in the right has a blue border around it. This means that it is the **active view**. Widgets that give information about and controls for a single view, including the pipeline browser and properties panel, follow the active view. In this new view we will visualize the temperature of the mesh.

7. Make sure the blue border is still around the new, blank view (to the right). You can make any view the active view by simply clicking on it.

8. Turn on the visibility of the clipped data by clicking the eyeball 👁 next to Clip1 in the pipeline browser.

9. Color the surface by temperature by selecting Clip1 in the pipeline browser and changing the variable chooser (in the toolbar) from Solid Color to Temp.
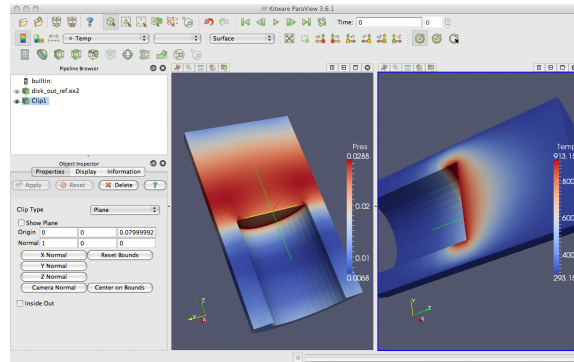
We now have two views: one showing information about pressure and the
other information about temperature. We would like to compare these, but it
is difficult to do because the orientations are different. How are we to know
how a location in one correlates to a location in the other? We can solve
this problem by adding a **camera link** so that the two views will always be
drawn from the same viewpoint. Linking cameras is easy.

10. Right click on one of the views and select Link Camera... from the pop
    up menu. (If you are on a Mac with no right mouse button, you can
    perform the same operation with the menu option Tools → Add Camera
    Link....)

11. Click in the second view.

12. Try moving the camera in each view.

Voilà! The two cameras are linked; each will follow the other. With the
cameras linked, we can make some comparisons between the two views. Click
the button button to get a straight-on view of the cross section and zoom in a
bit.
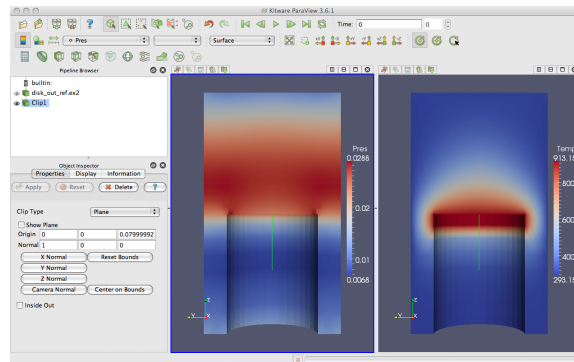
Notice that the temperature is highest at the interface with the heated disk. That alone is not surprising. We expect the air temperature to be greatest near the heat source and drop off away from it. But notice that at the same position the pressure is not maximal. The air pressure is maximal at a position above the disk. Based on this information we can draw some interesting hypotheses about the physical phenomenon. We can expect that there are two forces contributing to air pressure. The first force is that of gravity causing the upper air to press down on the lower air. The second force is that of the heated air becoming less dense and therefore rising. We can see based on the maximal pressure where these two forces are equal. Such an observation cannot be drawn without looking at both the temperature and pressure in this way. ♦

Multiview in ParaView is of course not limited to simply two windows. Note that each of the views has its own set of multiview buttons. You can create more views by using the split view buttons ⊞ ⊟ to arbitrarily divide up the working space. And you can delete views ✕ at any time.

The location of each view is also not fixed. You are also able to swap two views by clicking on one of the view toolbars (somewhere outside of where the buttons are), holding down the mouse button, and dragging onto one of the other view toolbars. This will immediately swap the two views.
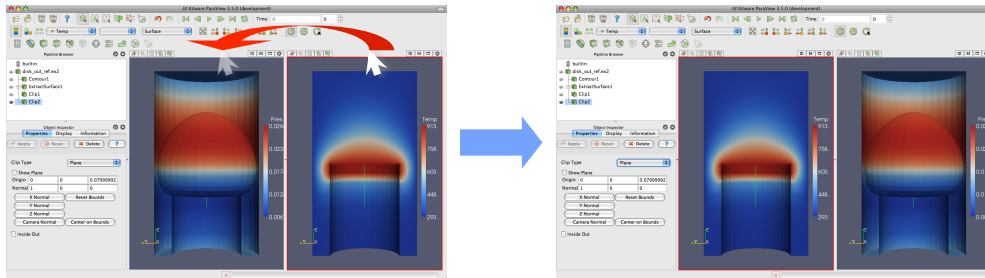
You can also change the size of the views by clicking on the space in between views, holding down the mouse button, and dragging in the direction of either one of the views. The divider will follow the mouse and adjust the size of the views as it moves.
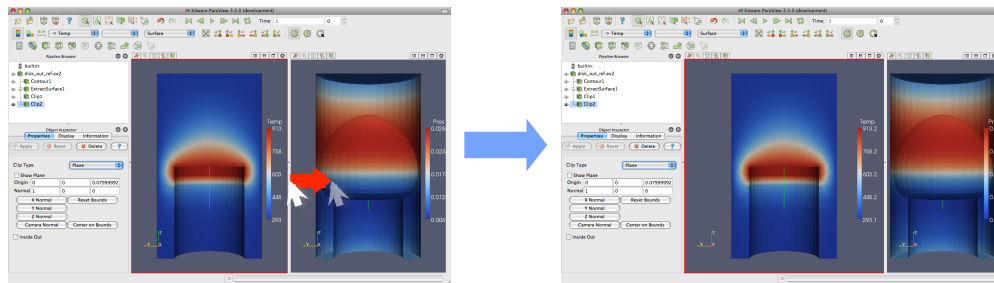
2.8   Vector Visualization

Let us see what else we can learn about this simulation. The simulation has also outputted a velocity field describing the movement of the air over the heated rotating disk. We will use ParaView to determine the currents in the air.

A common and effective way to characterize a vector field is with **streamlines**. A streamline is a curve through space that at every point is tangent to the vector field. It represents the path a weightless particle will take through the vector field (assuming steady-state flow). Streamlines are generated by providing a set of **seed points**.

Exercise 2.12: Streamlines

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to select reset session ⟳ from the toolbar.

1. Open the file disk_out_ref.ex2, load all variables, [⟳ Apply] (see Exercise 2.7).

2. Add the stream tracer filter ≋ to disk_out_ref.ex2.

3. Change the Seed Type parameter in the properties panel to Point Source.

4. Uncheck the Show Sphere checkbox [☑ Show Sphere] in the properties panel under the seed type.

5. Click the [⟳ Apply] button to accept these parameters.
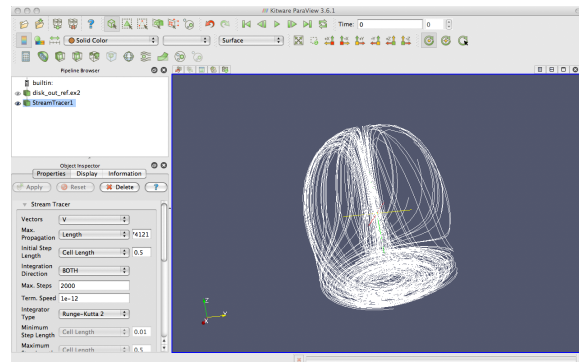
The surface of the mesh is replaced with some swirling lines. These lines represent the flow through the volume. Notice that there is a spinning motion around the center line of the cylinder. There is also a vertical motion in the center and near the edges.

The new geometry is off-center from the previous geometry. We can quickly center the view on the new geometry with the **reset camera** ⤢ command. This command centers and fits the visible geometry within the current view and also resets the center of rotation to the middle of the visible geometry.   ♦

One issue with the streamlines as they stand now is that the lines are difficult to distinguish because there are many close together and they have no shading. Lines are a 1D structure and shading requires a 2D surface. Another issue with the streamlines is that we cannot be sure in which direction the flow is.

In the next exercise, we will modify the streamlines we created in Exercise 2.12 to correct these problems. We can create a 2D surface around our stream traces with the tube filter. This surface adds shading and depth cues to the lines. We can also add glyphs to the lines that point in the direction of the flow.

Exercise 2.13: Making Streamlines Fancy

This exercise is a continuation of Exercise 2.12. You will need to finish that exercise before beginning this one.

1. Use the quick launch (ctrl+space Win/Linux, alt+space Mac) to add the Tube filter to the streamlines.

2. Hit the [Apply] button.
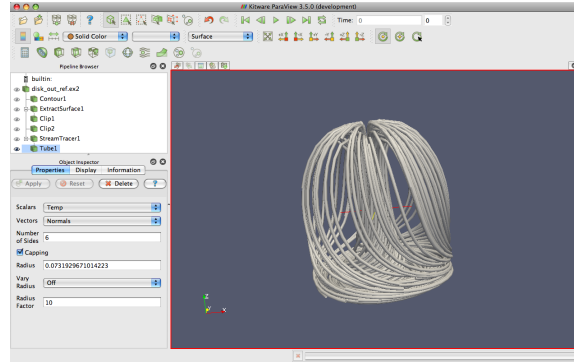
You can now see the streamlines much more clearly. As you look at the streamlines from the side, you should be able to see circular convection as air heats, rises, cools, and falls. If you rotate the streams to look down the Z axis at the bottom near where the heated plate should be, you will also see that the air is moving in a circular pattern due to the friction of the rotating disk.

(Note that as an alternative to adding the Tube filter, you can instead directly render lines as tubes. This applies a special rendering mode that renders wide lines with tube-like shading. To do this, *instead* of creating the Tube filter set Line Width display parameter to some value larger than 1 (say 5) and click on the Render Lines As Tubes display parameter.)

Now we can get a little fancier. We can add glyphs to the streamlines to show the orientation and magnitude.

3. Select StreamTracer1 in the pipeline browser.

4. Add the glyph filter 🌐 to StreamTracer1.

5. In the properties panel, change the Glyph Type option to Cone.

6. In the properties panel, change the Orientation Array to V.

7. In the properties panel, change the Scale Array to V.

8. Click the reset 🔄 button under Glyph Data Range.

9. Hit the [Apply] button.

10. Color the glyphs with the Temp variable.
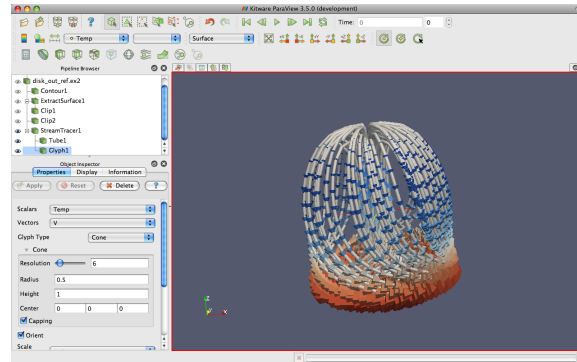
Now the streamlines are augmented with little pointers. The pointers face in the direction of the velocity, and their size is proportional to the magnitude of the velocity. Try using this new information to answer the following questions.

- Where is the air moving the fastest? Near the disk or away from it? At the center of the disk or near its edges?

- Which way is the plate spinning?

- At the surface of the disk, is air moving toward the center or away from it?

♦

2.9 Plotting

ParaView's plotting capabilities provide a mechanism to drill down into your data to allow quantitative analysis. Plots are usually created with filters, and all of the plotting filters can be found in the Data Analysis submenu of Filters. There is also a data analysis toolbar containing the most common data analysis filters, some of which are used to generate plots.

Extract Selection Extracts any data selected into its own object. Selections are described in Section 2.14.

⚙ **Plot Global Variables Over Time** Data sets sometimes capture information in "global" variables that apply to an entire dataset rather than a single point or cell. This filter plots the global information over time. ParaView's handling of time is described in Section 2.11.

🗺 **Plot Over Line** Allows you to define a line segment in 3D space and then plot field information over this line.

⚙ **Plot Selection Over Time** Takes the fields in selected points or cells and plots their values over time. Selections are described in Section 2.14 and time is described in Section 2.11.

❉ **Probe** Provides the field values in a particular location in space.

In the next exercise, we create a filter that will plot the values of the mesh's fields over a line in space.

Exercise 2.14: Plot Over a Line in Space

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to select ⟳ from the toolbar.

1. Open the file disk_out_ref.ex2, load all variables, [⚙ Apply] (see Exercise 2.7).

2. Add the Clip filter 🔲 to disk_out_ref.ex2, Uncheck the Show Plane checkbox [✓ Show Plane] in the properties panel, and click [⚙ Apply] (like in Exercise 2.11). This will make it easier to see and manipulate the line we are plotting over.

3. Click on disk_out_ref.ex2 in the pipeline browser to make that the active object.

4. From the toolbars, select the plot over line 🗺 filter.
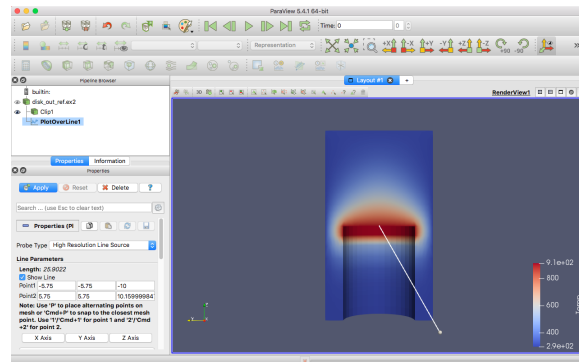
In the active view you will see a line through your data with a ball at each end. If you move your mouse over either of these balls, you can drag the balls through the 3D view to place them. If you hold down the x, y, or z key while you drag one of these balls, the movement will be constrained to that axis. Notice that each time you move the balls some of the fields in the properties panel also change. You can also place the balls by hovering your mouse over the target location and hitting the 1, 2, or p key. The 1 key will place the first ball at the surface underneath the mouse cursor. The 2 key will likewise place the second ball. The p key will alternate between placing the first and second balls. If you hold down the Ctrl modifier while hitting any of these keys, then the ball will be placed at the nearest point of the underlying mesh rather than directly under the mouse. This was the purpose of adding the clip filter: It allows us to easily add the endpoints to this plane. Note that placing the endpoints in this manner only works when rendering solid surfaces. It will not work with a volume rendered image or transparent surfaces.

This representation is called a **3D widget** because it is a GUI component that is manipulated in 3D space. There are many examples of 3D widgets in ParaView. This particular widget, the line widget, allows you to specify a line segment in space. Other widgets allow you to specify points or planes.

5. Adjust the line so that it goes from the base of the disk straight up to the top of the mesh using the 3D widget manipulators, the p key shortcut, or the properties panel parameters. The plot works best with Point 1 around $(0, 0, 0)$ and Point 2 around $(0, 0, 10)$.

6. Once you have your line satisfactorily located, click the [Apply] button.
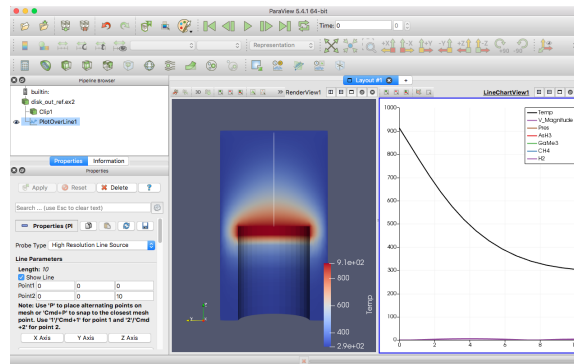
There are several interactions you can do with the plot. Roll the mouse wheel up and down to zoom in and out. Drag with the middle button to do a rubber band zoom. Drag with the left button to scroll the plot around. You can also use the reset camera command ⤢ to restore the view to the full domain and range of the plot.   ♦

Plots, like 3D renderings, are considered views. Both provide a representation for your data; they just do it in different ways. Because plots are views, you interact with them in much the same ways as with a 3D view. If you look in the Display section of the properties panel, you will see many options on the representation for each line of the plot including colors, line styles, vector components, and legend names.
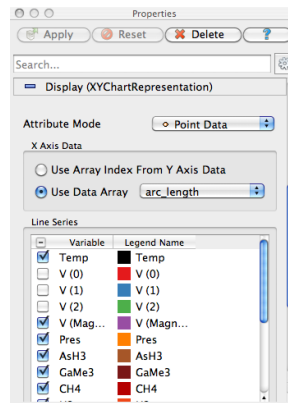
If you scroll down further to the View section of the properties panel, you to change plot-wide options such as labels, legends, and axes ranges.
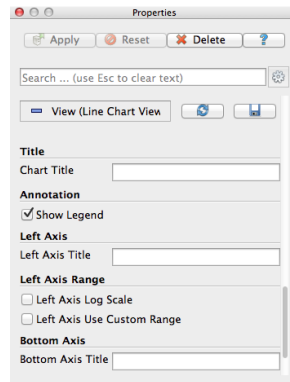
Like any other views, you can capture the plot with the File → 📷 Save Screenshot. Additionally, if you choose File → Export Scene... you can export a file with vector graphics that will scale properly for paper-quality images. We will discuss these image capture features later in Section 2.13. You can also resize and swap plots in the GUI like you can other views.

In the next exercise, we modify the display to get more information out of our plot. Specifically, we use the plot to compare the pressure and temperature variables.

Exercise 2.15: Plot Series Display Options

This exercise is a continuation of Exercise 2.14. You will need to finish that exercise before beginning this one.

1. Choose a place in your GUI that you would like the plot to go and try using the split, delete, resize, and swap view features to move it there.

2. Make the plot view active, go to the Display section of the properties panel, and turn off all variables except Temp and Pres.

The Temp and Pres variables have different units. Putting them on the same scale is not useful. We can still compare them in the same plot by placing each variable on its own scale. The line plot in ParaView allows for a different scale on the left and right axis, and you can scale each variable individually on each axis.

3. Select the Pres variable in the Display options.

4. Change the Chart Axis to Bottom - Right
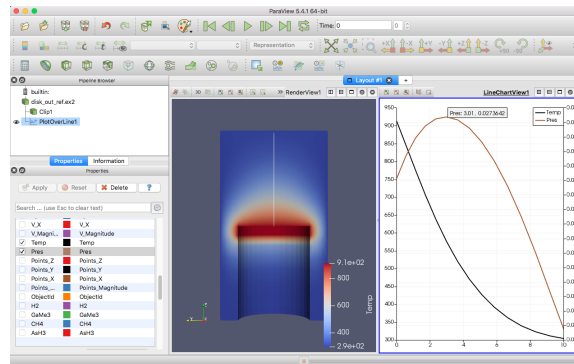
From this plot we can verify some of the observations we made in Section 2.7. We can see that the temperature is maximal at the plate surface and falls as we move away from the plate, but the pressure goes up and then back down. In addition, we can observe that the maximal pressure (and hence the location where the forces on the air are equalized) is about 3 units away from the disk.   ♦

The ParaView framework is designed to accommodate any number of different types of views. This is to provide researchers and developers a way to deliver new ways of looking at data. To see another example of view, select disk_out_ref.ex2 in the pipeline browser, and then select Filters → Data Analysis → Histogram . Make the histogram for the Temp variable, and then hit the Apply button.
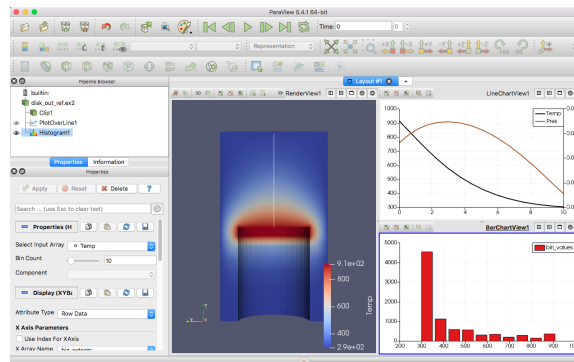
2.10 Volume Rendering

ParaView has several ways to represent data. We have already seen some examples: surfaces, wireframe, and a combination of both. ParaView can also render the points on the surface or simply draw a bounding box of the data.

| Points | Wireframe | Surface | Surface with Edges | Volume |
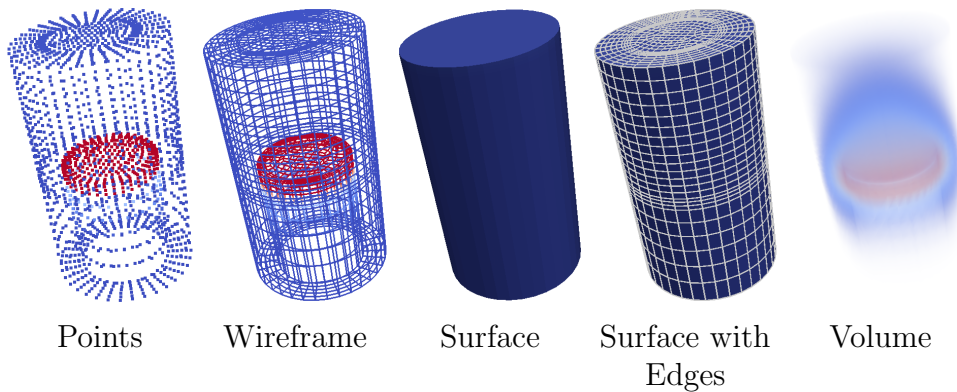
A powerful way that ParaView lets you represent your data is with a technique called **volume rendering**. With volume rendering, a solid mesh is rendered as a translucent cloud with the scalar field determining the color and density at every point in the cloud. Unlike with surface rendering, volume rendering allows you to see features all the way through a volume.

Volume rendering is enabled by simply changing the representation of the object. Let us try an example of that now.

Exercise 2.16: Turning On Volume Rendering

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to select ⟳ from the toolbar.

1. Open the file disk_out_ref.ex2, load all variables, Apply (see Exercise 2.7).

2. Make sure disk_out_ref.ex2 is selected in the pipeline browser. Change the variable viewed to Temp and change the representation to Volume.

3. If you get an Are you sure? dialog box warning you about the change to the volume representation, click Yes to enact the change.
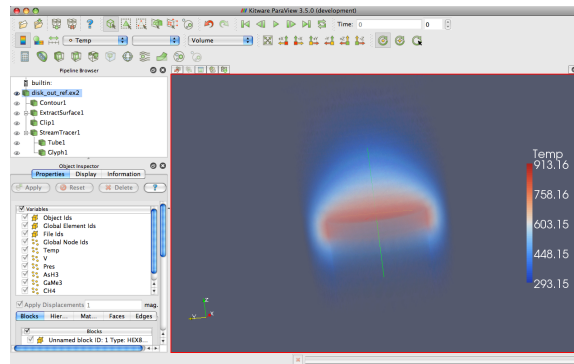
The solid opaque mesh is replaced with a translucent volume. You may notice that when rotating your object that the rendering is temporarily replaced with a simpler transparent surface for performance reasons. We discuss this behavior in more detail later in Chapter 4.   ♦

A useful feature of ParaView's volume rendering is that it can be mixed with the surface rendering of other objects. This allows you to add context to the volume rendering or to mix visualizations for a more information-rich view. For example, we can combine this volume rendering with a streamline vector visualization like we did in Exercise 2.12.

Exercise 2.17: Combining Volume Rendering and Surface-Based Visualization

This exercise is a continuation of Exercise 2.16. You will need to finish that exercise before beginning this one.

1. Add the stream tracer filter 🖿 to disk_out_ref.ex2.

2. Change the Seed Type parameter in the properties panel to Point Source.

3. Uncheck the Show Sphere checkbox ☑ Show Sphere in the properties panel under the seed type.

4. Click the 🖿 Apply button to accept these parameters.

You should now be seeing the streamlines embedded within the volume rendering. The following additional steps add geometry to make the streamlines easier to see much like in Exercise 2.13. They are optional, so you can skip them if you wish.

5. Use the quick launch (ctrl+space Win/Linux, alt+space Mac) to apply the Tube filter and hit [Apply].

6. If the streamlines are colored by Temp, change that to Solid Color.

7. Select StreamTracer1 in the pipeline browser.

8. Add the glyph filter to StreamTracer1.

9. In the properties panel, change the Glyph Type option to Cone.

10. In the properties panel, change the Orientation Array to V.

11. In the properties panel, change the Scale Array to V.

12. Click the reset button under Glyph Data Range.

13. Hit the [Apply] button.

14. Color the glyphs with the Temp variable.
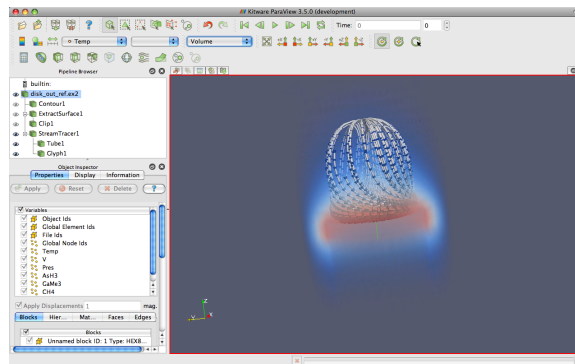
The streamlines are now shown in context with the temperature throughout the volume. ♦

By default, ParaView will render the volume with the same colors as used on the surface with the transparency set to 0 for the low end of the range and 1 for the high end of the range. ParaView also provides an easy way to change the **transfer function**, how scalar values are mapped to color and transparency. You can access the transfer function editor by selecting the volume rendered-pipeline object (in this case disk_out_ref.ex2) and clicking on the edit color map button.
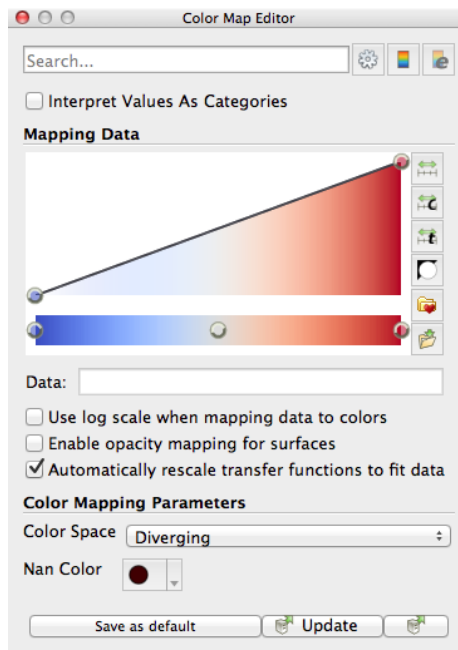
The first time you bring up the color map editor, it should appear at the right side of the ParaView GUI window. Like most of the panels in ParaView, this is a dockable window that you can move around the GUI or pull off and place elsewhere on your desktop. Like the properties panel, some of the advanced options are hidden to simplify the interface. To access these hidden features, toggle the ⚙ button in the upper right or type a search string.

The two colorful boxes at the top represent the transfer function. The first box with a function plot with colors underneath represents the transparency whereas the long box at the bottom represents the colors.[1] The dots on the transfer functions represent the **control points**. The control points are the specific color and opacity you set at particular scalar values, and the colors and transparency are interpolated between them. Clicking on a blank spot in either bar will create a new control point. Clicking on an existing control point will select it. The selected control point can be dragged throughout the box to change its scalar value and transparency (if applicable). Double clicking on a color control point will allow you to change the color. The

---

[1] For surface rendering, the transparency controls have no effect unless "Enable opacity mapping for surfaces" is enabled.

selected control point will be deleted when you hit the backspace or delete key.

Directly below the color and transparency bars is a text entry widget to numerically specify the Data Value of the selected control point. Below this are checkbox options to Use log scale when mapping data to colors, to Enable opacity mapping for surfaces, and to Automatically rescale transfer functions to fit data. (Note that this last option causes the data range to be resized under most operations that change data, but not when the time value changes. See Section 2.11 for more details.)

The following Color Space parameter changes how colors are interpolated. This parameter has no effect on the color at the control points, but can drastically affect the colors between the control points. Finally, the Nan Color allows you to select a color for "invalid" values. A **NaN** is a special floating point value used to represent something that is not a number (such as the result of 0/0).

Setting up a transfer function can be tedious, so you can save it by clicking the Save to preset 🗁 button. The Choose preset 💝 button brings up a dialog that allows you to manage and apply the color maps that you have created as well as many provided by ParaView.

Exercise 2.18: Modifying Volume Rendering Transfer Functions

This exercise is a continuation of Exercise 2.17. You will need to finish that exercise (or minimally Exercise 2.16) before beginning this one.

1. Click on disk_out_ref.ex2 in the pipeline browser to make that the active object.

2. Click on the edit color map 🎨 button.

3. Change the volume rendering to be more representative of heat. Press Choose preset 💝, select Black-Body Radiation in the dialog box, and then click Apply followed by Close.

4. Try adding and changing control points and observe their effect on the volume rendering. By default as you make changes the render views update. If this interactive update is too slow, you can turn off this feature by toggling the 🗔 button. When automatic updates are off,

transfer function changes are not applied until the 🗃 Render Views is clicked.
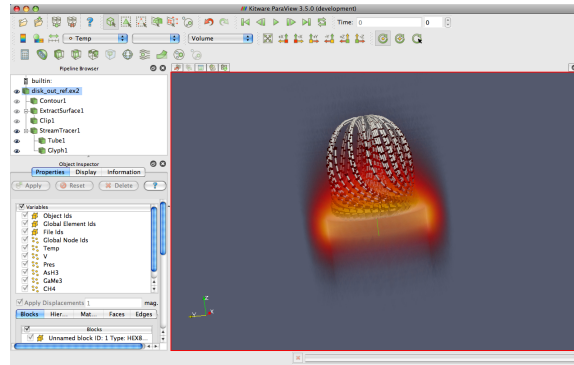
Notice that not only did the color mapping in the volume rendering change, but all the color mapping for Temp changed including the cone glyphs if you created them. This ensures consistency between the views and avoids any confusion from mapping the same variable with different colors or different ranges.   ♦

While looking through the color map presents 🗀, you probably noticed one or more entries with **Rainbow** as part of the title that incorporate the colors of the a rainbow into the color map. You may also recognize this set of colors from other visualizations you have seen in the past.

Rainbow colors are certainly a popular choice. However, we recommend that you **never use rainbow color maps** in your visualizations.

The problem with rainbow colors is that they have numerous perceptual properties that serve to obfuscate the data. Although we will not go into detail into the many perceptual studies that provide evidence that rainbow colors are bad for data display, we provide a brief synopsis of the problems.

The first problem is that the colors do not follow any natural perceived ordering. Some color groups lead to a natural perception of order (with relative brightness being the strongest perceptual cue). The hues of the rainbow, however, have no real ordered meaning in our visual system. Rather, we have to learn an ordering, which can lead to visual confusion. Consider the following two example images. In the left image, the rainbow hues make

it difficult to ascertain the relative high and low values that are more clear in the right image.
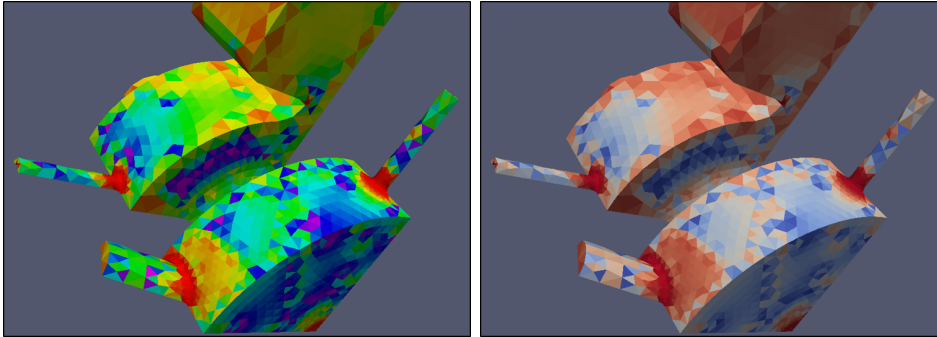
The second problem with the perception of rainbow colors is that the perceptual changes in the colors are not uniform. The colors appear to change faster in the cyan and yellow regions, which can introduce artifacts in those regions that do not exist in the data. The colors appear to change more slowly in the blue, green, and red regions, which creates larger bands of color that hide artifacts in the data. We can see this effect in the following two images of a spatial contrast sensitivity function. The grayscale on the left faithfully reproduces the function. However, the rainbow colors on the right hides the variation in low contrast regions and appears less smooth in the high-contrast regions.
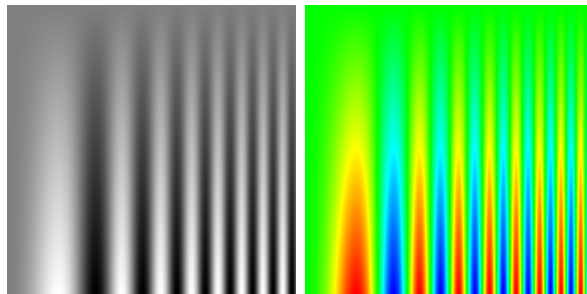
A third problem with the rainbow color map is that it is sensitive to deficiencies in vision. Roughly 5% of the population cannot distinguish between the red and green colors. Viewers with color deficiencies cannot distinguish many colors considered "far apart" in the rainbow color map.

We provide this description of the problems with rainbow colors in the hopes that you do not use these color maps. Despite the well know problems

with rainbow colors, they remain a popular choice. Relying on rainbow hues obfuscates data, which circumvents the entire process of data analysis that ParaView provides. Multiple perceptual studies have shown that subjects overestimate the effectiveness of rainbow colors. That is, people think they do better with rainbow colors when in fact they do worse. Don't fall into this trap.

2.11   Time

Now that we have thoroughly analyzed the disk_out_ref simulation, we will move to a new simulation to see how ParaView handles time. In this section we will use a new data set from another simple simulation, this time with data that changes over time.

Exercise 2.19: Loading Temporal Data

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to select ⟳ from the toolbar.

1. Open the file can.ex2.
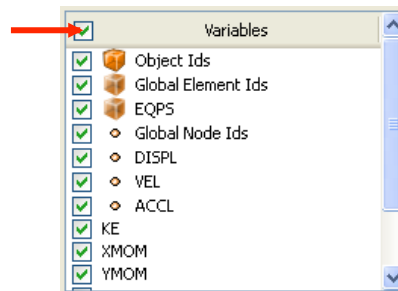
2. As before, click the checkbox in the header of the variable list to turn on the loading of all the variables and hit the ⟦ Apply ⟧ button.

3. Press the ⬆+Y button to orient the camera to the mesh.

4. Press the play button ▷ in the toolbars and watch ParaView animate the mesh to crush the can with the falling brick.

♦
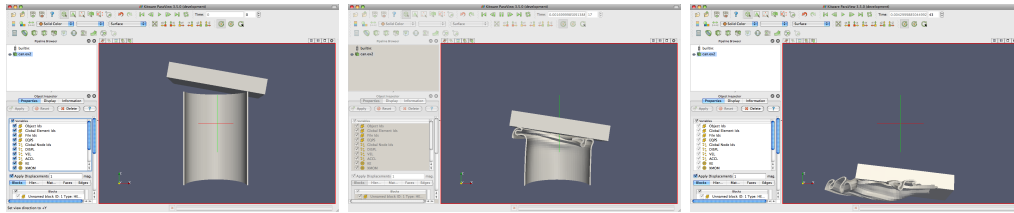
That is really all there is to dealing with data that is defined over time. ParaView has an internal concept of time and automatically links in the time defined by your data. Become familiar with the toolbars that can be used to control time.
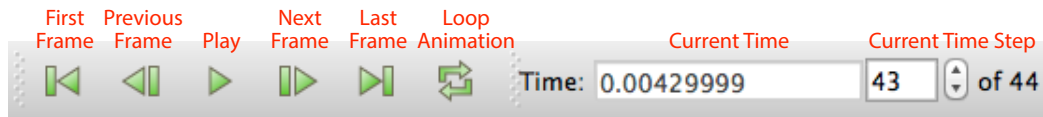
Saving an animation is equally as easy. From the menu, select File → Save Animation. ParaView provides dialogs specifying how you want to save the animation, and then automatically iterates and saves the animation.

Exercise 2.20: Temporal Data Pitfall

The biggest pitfall users run into is that with mapping a set of colors whose range changes over time. To demonstrate this, do the following.

1. If you are not continuing from Exercise 2.19, open the file can.ex2, load all variables, Apply .

2. Go to the first time step .

3. Color by the EQPS variable.

4. Play ▷ through the animation (or skip to the last time step).

The coloring is not very useful. To quickly fix the problem:

5. While at the last time step, click the Rescale to Data Range button.

6. Play ▷ the animation again.

The colors are more useful now.   ◆

Although this behavior seems like a bug, it is not. It is the consequence
of two unavoidable behaviors. First, when you turn on the visibility of a
scalar field, the range of the field is set to the range of values in the current
time step. Ideally, the range would be set to the max and min over all time
steps in the data.

However, this requires ParaView to load in all of the data on the initial
read, and that is prohibitively slow for large data. Second, when you animate
over time, it is important to hold the color range fixed even if the range in the
data changes. Changing the scale of the data as an animation plays causes
a misrepresentation of the data. It is far better to let the scalars go out of
the original color map's range than to imply that they have not. There are
several workarounds to this problem:

- If for whatever reason your animation gets stuck on a poor color range,
  simply go to a representative time step and hit ⇌. This is what we
  did in the previous exercise.

- Open the settings dialog box accessed in the menu from Edit → Settings
  (ParaView → Preferences on the Mac). Under the General tab, find the
  option labeled Default Time Step and change it to Go to last timestep.
  (If you have trouble finding this option, try typing timestep into the
  setting's search box.) When this is selected, ParaView will automati-
  cally go to the last time step when loading any data set with time. For
  many data (such as in can), the field ranges are more representative at
  the last time step than at the beginning. Thus, as long as you color by
  a field before changing the time, the color range will be adequate.

- Click the Rescale to Custom Data Range ⇌ toolbar button. This is a
  good choice if you cannot find, or do not know, a "representative" time
  step or if you already know a good range to use.

- If you are willing to wait or have small data, you can use the Rescale
  to data range over all timesteps ⇌ toolbar button and ParaView will
  compute this overall temporal range automatically. Keep in mind that
  this option will require ParaView to load your entire data set over all
  time steps. Although ParaView will not hold more than one time step
  in memory at a time, it will take a long time to pull all that memory
  off of disk for large data sets.

ParaView has many powerful options for controlling time and animation. The majority of these are accessed through the **animation view**. From the menu, click on View → Animation View.
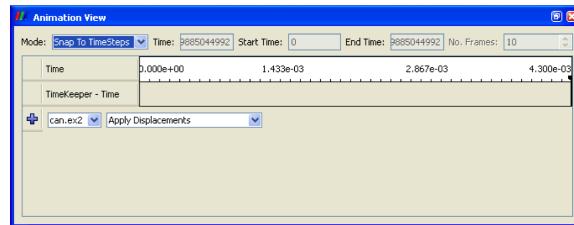
For now we will examine the controls at the top of the animation view. (We will examine the use of the tracks in the rest of the animation view later in Section 2.15.) The **animation mode** parameter determines how ParaView will step through time during playback. There are three modes available.

Sequence Given a start and end time, break the animation into a specified number of frames spaced equally apart.

Real Time ParaView will play back the animation such that it lasts the specified number of seconds. The actual number of frames created depends on the update time between frames.

Snap To TimeSteps ParaView will play back exactly those time steps that are defined by your data.

Whenever you load a file that contains time, ParaView will automatically change the animation mode to Snap To TimeSteps. Thus, by default you can load in your data, hit play ▷, and see each time step as defined in your data. This is by far the most common use case.

A counter use case can occur when a simulation writes data at variable time intervals. Perhaps you would like the animation to play back relative to the simulation time rather than the time index. No problem. We can switch to one of the other two animation modes. Another use case is the desire to change the playback rate. Perhaps you would like to speed up or slow down the animation. The other two animation modes allow us to do that.

Exercise 2.21: Slowing Down an Animation with the Animation Mode

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to select ⟳ from the toolbar.

1. Open the file can.ex2, load all variables, 𝗔𝗽𝗽𝗹𝘆 (see Exercise 2.19).

2. Press the ⬆+Y button to orient the camera to the mesh.

3. Press the play button ▷ in the toolbars.

   During this animation, ParaView is visiting each time step in the original data file exactly once. Note the speed at which the animation plays.

4. If you have not done so yet, make the animation view visible: View → Animation View.

5. Change the animation mode to Real Time. By default the animation is set up with the time range specified by the data and a duration of 10 seconds.

6. Play ▷ the animation again.

   The result looks similar to the previous Snap To TimeSteps animation, but the animation is now a linear scaling of the simulation time and will complete in 10 seconds.

7. Change the Duration to 60 seconds.

8. Play ▷ the animation again.

   The animation is clearly playing back more slowly. Unless your computer is updating slowly, you will also notice that the animation appears jerkier than before. This is because we have exceeded the temporal resolution of the data set.   ♦

   Often showing the jerky time steps from the original data is the desired behavior; it is showing you exactly what is present in the data. However, if you wanted to make an animation for a presentation, you may want a smoother animation.

There is a filter in ParaView designed for this purpose. It is called the **temporal interpolator**. This filter will interpolate the positional and field data in between the time steps defined in the original data set.

Exercise 2.22: Temporal Interpolation

This exercise is a continuation of Exercise 2.21. You will need to finish that exercise before beginning this one.

1. Make sure can.ex2 is highlighted in the pipeline browser.

2. Select Filters → Temporal → Temporal Interpolator or apply the Temporal Interpolator filter using the quick launch (ctrl+space Win/Linux, alt+space Mac).

3. Apply .

4. Split the view , show the TemporalInterpolator1 in one, show can.ex2 in the other, and link the cameras.

5. Play ▷ the animation.

You should notice that the output from the temporal interpolator animates much more smoothly than the original data.   ♦

It is worth noting that the temporal interpolator can (and often does) introduce artifacts in the data. It is because of this that ParaView will never apply this type of interpolation automatically; you will have to explicitly add the Temporal Interpolator. In general, mesh deformations often interpolate well but moving fields through a static mesh do not. Also be aware that the Temporal Interpolator only works if the topology remains consistent. If you have an adaptive mesh that changes from one time step to the next, the Temporal Interpolator will give errors.

2.12   Text Annotation

When using ParaView as a communication tool it is often helpful to annotate the images you create with text. With ParaView it is very easy to create

text annotation wherever you want in a 3D view. There is a special **text source** that simply places some text in the view.

Exercise 2.23: Adding Text Annotation

If you are continuing this exercise after finishing Exercise 2.22, feel free to simply continue. If you have had to restart ParaView since or your state does not match up well enough, it is also fine to start with a fresh state.

1. Use the quick launch (ctrl+space Win/Linux, alt+space Mac) to create the Text source (or from the menu bar Sources → Annotation → Text) and hit Apply.

2. In the text edit box of the properties panel, type a message.

3. Hit the Apply button.
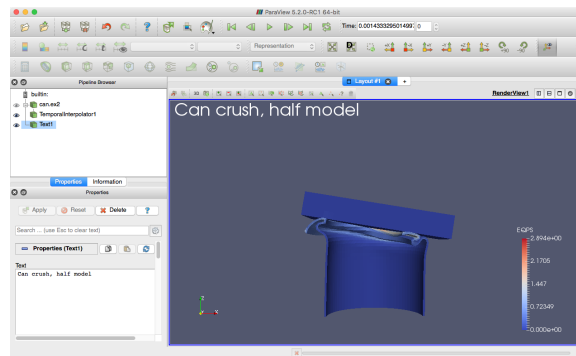
The text you entered appears in the 3D view. If you scroll down to the Display options in the properties panel, you will see six buttons that allow you to quickly place the text in each of the four corners of the view as well as centered at the top and bottom.
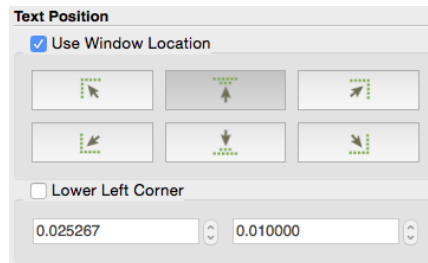
You can place this text at an arbitrary position by clicking the Lower Left Corner checkbox. With the Lower Left Corner option checked, you can use the mouse to drag the text to any position within the view.   ♦

Often times you will need to put the current time value into the text annotation. Typing the correct time value can be tedious and error prone with the standard text source and impossible when making an animation. Therefore, there is a special **annotate time source** that will insert the current animation time into the string.

Exercise 2.24: Adding Time Annotation

1. If you do not already have it loaded from a previous exercise, open the file can.ex2, [Apply].

2. Add an Annotate Time source (Sources → Annotate → Annotate Time or use the quick launch: ctrl+space Win/Linux, alt+space Mac). [Apply].

3. Move the annotation around as necessary.

4. Play ▷ and observe how the time annotation changes.
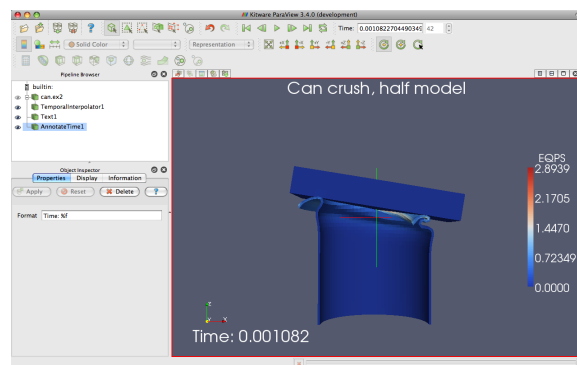
There are instances when the current animation time is not the same as the time step read from a data file. Often it is important to know what the time stored in the data file is, and there is a special version of annotate time that acts as a filter.

5. Select can.ex2 in the pipeline browser.

6. Use the quick launch (ctrl+space Win/Linux, alt+space Mac) to apply the Annotate Time Filter.

7. ⬚ Apply .

8. Move the annotation around as necessary.

9. Check the animation mode in the Animation View. If it is set to Snap to TimeSteps, change it to Real Time.

10. Play ▷ and observe how the time annotation changes.

♦
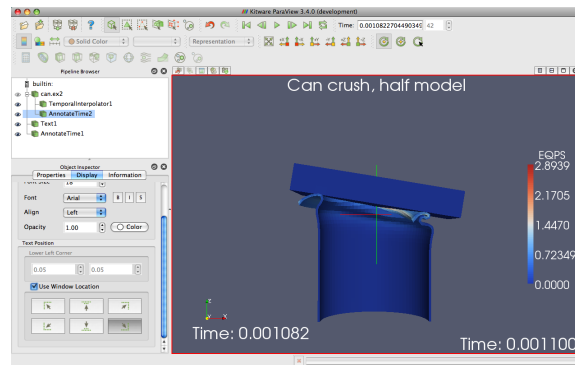
You can close the animation view. We are done with it for now, but we will revisit it again in Section 2.15.

2.13   Save Screenshot and Save Animation

One of the most important products of any visualization is screenshots and movies that can be used in presentations and reports. In this section we save a screenshot (picture) and animation (movie). Once again, we will use the can.ex2 dataset.

Exercise 2.25: Save Screenshot

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to select ⏻ from the toolbar.

1. Open the file can.ex2, load all variables, ⎾ Apply ⏋ (see Exercise 2.19).

2. Press the ⬆+Y button to orient the camera to the mesh.

3. Color by GlobalNodeId. We use GlobalNodeId so that the 3D object has some color.

4. Select File → Save Screenshot ⎾◉⏋.

   This brings us to the file selection screen. If you pull down the menu Files of type: at the bottom of the dialog box, you will see several file types supported including portable network graphics (PNG), and joint photographic experts group (JPEG).

   Select a File name for your file, and place it somewhere you can later find and delete. We usually recommend saving images as PNG files. The lossy compression of JPEG often creates noticeable artifacts in the images generated by ParaView, and the compression of PNG is better than most other raster formats.

5. Press the OK button.

   After you select a file, you will be presented with a dialog box for more screenshot options.
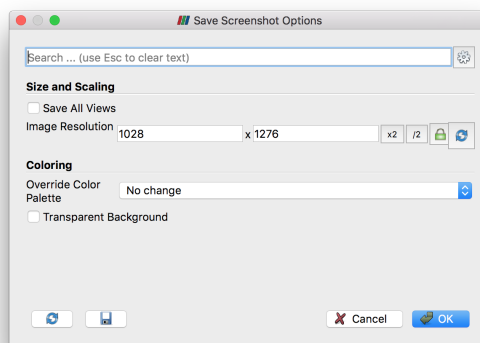
The Save Screenshot Options dialog box includes numerous important controls.

At the top of the dialog box are Size and Scaling options. The Image Resolution options allow you to create an image that is larger (or smaller) than the current size of the 3D view. You can directly enter a resolution or use the x2 and /2 buttons to double or halve the resolution. The 🔒 button toggles locking the aspect ratio when selecting an image resolution. The 🔄 button restores the resolution to that of the view in the GUI. If you have multiple views open, there is also a Save All Views checkbox that toggles between saving only the active view and saving all views in one image.

The Coloring controls modify how some elements, such as the background, get colored. The Override Color Palette pulldown menu allows a user to use the default color scheme, one with a white color motif for printing, or other defined palettes. These are the same palettes described in Exercise 2.5 on page 17. There is also a Transparent Background checkbox that adds an alpha channel to the written image (in formats that support it).

There are also many advanced options that, as always are accessible via the ⚙ button or the search bar.

6. Press the OK button.

Using your favorite image viewer, find and load the image you created. If you have no image viewer, ParaView itself is capable of loading PNG files.

<div align="right">◆</div>

Next, we will save an animation.

Exercise 2.26: Save Animation

1. If you do not already have it loaded from the previous exercise, open the file can.ex2, load all variables, and 🗗 Apply (see Exercise 2.19).

2. Select File → Save Animation 📸.

This brings us to the file selection screen. If you pull down the menu Files of type: at the bottom of the dialog box, you will see the several file types including Ogg/Theora, AVI, JPEG, and PNG.

Select a File name for your file, and place it somewhere you can later find and delete. AVI will create a movie format that can be used on windows, and with some open source viewers. Ogg/Theora is used in many open source viewers. Otherwise, you can create a **flipbook**, or series of images. These images can be stitched together to form a movie using numerous open source tools. For now, try creating an AVI.

3. Press the OK button.

The Save Animation Options dialog box looks essentially the same as the Save Screenshot Options dialog for screenshots. If you look at the advanced options ⚙, you may note that there are some added Animation Options that allow you to control the frame rate and narrow the time steps that will be animated.

4. Press the OK button.

Using your favorite movie viewer, find and load the image you created.

♦

2.14   Selection

The goal of visualization is often to find the important details within a large body of information. ParaView's selection abstraction is an important simplification of this process. Selection is the act of identifying a subset of some dataset. There are a variety of ways that this selection can be made, most of which are intuitive to end users, and a variety of ways to display and process the specific qualities of the subset once it is identified.

More specifically the subset identifies particular select points, cells, or blocks within any single data set. There are multiple ways of specifying which elements to include in the selection including Id lists of multiple varieties, spatial locations, and scalar values and scalar ranges.

In ParaView, selection can take place at any time, and the program maintains a current selected set that is linked between all views. That is, if you select something in one view, that selection is also shown in all other views that display the same object.

The most direct means to create a selection is via the Find Data 🖼 dialog. Launch this dialog from the toolbar or the Edit menu. From this dialog you

can enter characteristics of the data that you are seraching for. For example, you could look for points whose velocity magnitude is near terminal velocity. Or you could look for cells whose strain exceeds the failure of the material. The following exercise provides a quick example of using the Find Data 🖥 dialog box.

Exercise 2.27: Performing Query-Based Selections

In this exercise we will find all cells with a large equivalent plastic strain (EQPS).

   We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to select ⟳ from the toolbar.

1. Open the file can.ex2, load all variables, 📷 Apply (see Exercise 2.19).

2. Go to the last time step ▶|.

3. Open the find data dialog 🖥.

4. From the top combo box, choose to find Cells.

5. In the next row of widgets, choose EQPS from the first combo box, is >= from the second combo box, and enter 1.5 in the final text box.

6. Click the Run Selection Query button.
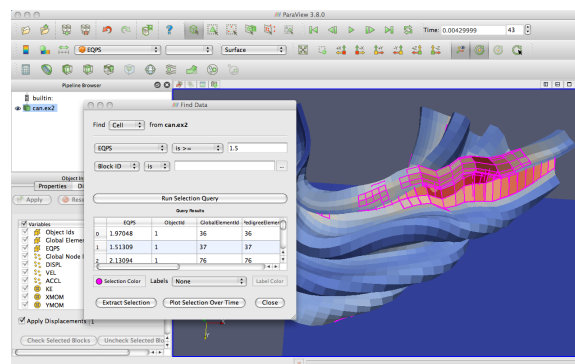
   Observe the spreadsheet below the Run Selection Query button that gets populated with the results of your query. Each row represents a cell and each column represents a field value or property (such as an identifier).

You may also notice that several cells are highlighted in the 3D view of the main ParaView window. These highlights represent the selection that your query created. Close the Find Data dialog and note that the selection remains. ♦

Another way of creating a selection is to pick elements right inside the 3D view. Most of the 3D view selections are performed with a **rubber-band selection**. That is, by clicking and dragging the mouse in the 3D view, you will create a boxed region that will select elements underneath it. There are also some 3D view selections that allow you to select within a polygonal region drawn on the screen. There are several types of interactive selections that can be performed, and you initiate one by selecting one of the icons in the small toolbar over the 3D view or using one of the shortcut keys. The following 3D selections are possible.

Select Cells On (Surface) Selects cells that are visible in the view under a rubber band. (Shortcut: s)

Select Points On (Surface) Selects points that are visible in the view under a rubber band. (Shortcut: d)

Select Cells Through (Frustum) Selects all cells that exist under a rubber band. (Shortcut: f)

Select Points Through (Frustum) Selects all points that exist under a rubber band. (Shortcut: g)

Select Cells With Polygon Like Select Cells On except that you draw a polygon by dragging the mouse rather than making a rubber-band selection.

Select Points With Polygon Like Select Points On except that you draw a polygon by dragging the mouse rather than making a rubber-band selection.

Select Blocks Selects blocks in a multiblock data set. (Shortcut: b)

Interactively Select Cells Enter an interactive selection mode where cells are highlighted as the mouse hovers over them and selected when clicked.

⚲ **Interactively Select Points** Enter an interactive selection mode where points are highlighted as the mouse hovers near them and selected when clicked.

In addition to these selection modes, there are a hover point ·❓ query mode and a hover cell ▲❓ query mode that allow you to quickly inspect the field values at a visible point by hovering the mouse over it. There is also a clear selection button that will remove the current selection.

Several of the selection modes have shortcut keys that allow you to make selections more quickly. Use them by placing the mouse cursor somewhere in the currently selected 3D view and hitting the appropriate key. Then click on the cell or block you want selected (or drag a rubber band over multiple elements).

Feel free to experiment with the selections now.

You can manage your selection with the Find Data ▦ dialog even if the selection was created with one of these 3D interactions rather than directly with a find data query. The find data dialog allows you to view all the points and cells in the selection as well as perform simple operations on the selection. These include inverting the selection (a checkbox just over the spreadsheet), adding labels (Exercise 2.29), freezing selections (Exercise 2.28), and shortcuts for the Plot Selection Over Time ▦ and Extract Selection ▦ filters (Exercises 2.30 and 2.31, respectively).

Experiment with selections in Find Data ▦ a bit. Open the Find Data ▦ dialog box. Then make selections using the rubber-band selection and see the results in the Find Data dialog box. Also experiment with altering the selection by inverting selections with the Invert selection checkbox.

It should be noted that selections can be internally represented in different ways. Some are recorded as a list of data element ids. Others are specified as a region in space or by query parameters. Although the selections all look the same, they can behave differently, especially with respect to changes in time. The following exercise demonstrates how these different selection mechanisms can behave differently.

Exercise 2.28: Data Element Selections vs. Spatial Selections

1. If you do not already have it loaded from the previous exercise, open

the file can.ex2, load all variables, [🔲 Apply] (see Exercise 2.19).

2. Make a selection using the **Select Cells Through** 🔲 tool.

3. If it is not already visible, show the **Find Data** 🔲 dialog box.

4. Click on the **Show Frustum** 🔲 checkbox in the **Find Data** dialog and rotate the 3D view. (Yes, the **Show Frustum** button has the same icon as the **Select Cells Through** button.)
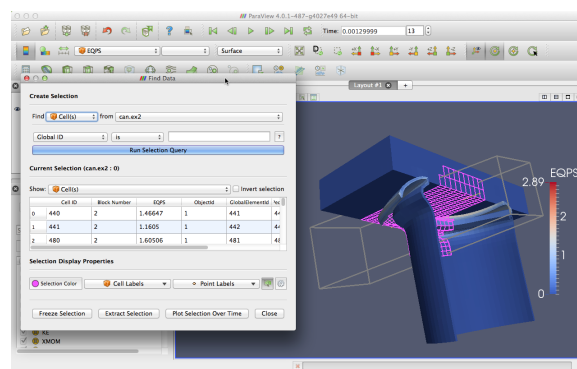
5. Play ▷ the animation a bit. Notice that the region remains fixed and the selection changes based on what cells move in or out of the region.

6. Go to a timestep where some data are selected. In the **Find Data** dialog box, click the **Freeze Selection** button.

7. Play ▷ again. Notice that the cells selected are fixed regardless of position.

In summary, a spatial selection (created with one of the select through tools) will re-perform the selection at each time step as elements move in and out of the selected region. Likewise, other queries such as field range queries will also re-execute as the data changes. However, when you select the **Freeze Selection** button, ParaView captures the identifiers of the currently selected elements so that they will remain the same throughout the animation.   ♦

The spreadsheet in the find dialog provides a readable way to inspect field data. However, sometimes it is helpful to place the field data directly in the 3D view. The next exercise describes how we can do that.

Exercise 2.29: Labeling Selections

1. If you do not already have it loaded from the previous exercise, open the file can.ex2, load all variables, [⚙ Apply] (see Exercise 2.19).

2. Go to the last time step ▶|.

3. If it is not already visible, show the Find Data 🖳 dialog box.

4. Using the controls at the top of the find data dialog box, create a selection where Global ID is min. Click Run Selection Query.

5. In the Cell Labels chooser, select EQPS.
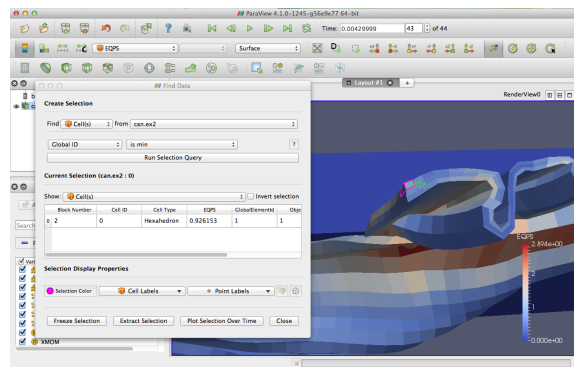
ParaView places the values for the EQPS field near the selected cell that contains that value. You should be able to see it in the 3D view. It is also possible to change the look of the font with respect to type, size, and color by clicking the ⚙ button to the right of the label choosers.

6. When you are done, turn off the labels by unchecking the entry in the Cell Labels chooser.

♦

ParaView provides the ability to plot field data over time. These plots can work on a selection, and to make this easier the Find Data 🖳 dialog box contains convenience controls to create them.

Exercise 2.30: Plot Over Time

1. If you do not already have it loaded from the previous exercise, open the file can.ex2, load all variables, ⟨ Apply ⟩ (see Exercise 2.19).

2. If it is not already visible, show the Find Data dialog box.

3. Using the controls at the top of the find data dialog box, create a selection where EQPS is max. Click Run Selection Query.

4. Click the Plot Selection Over Time button at the bottom of the find data dialog box to add that filter. This filter is also easily accessible by the toolbar button in the main ParaView window.

5. In the properties panel in the main ParaView window, click ⟨ Apply ⟩.
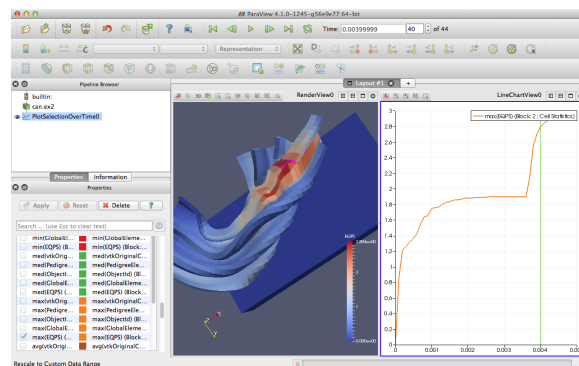
Note that the selection you had was automatically added as the selection to use in the Properties panel when the Plot Selection Over Time filter was created. If you want to change the selection, simply make a new one and click Copy Active Selection in the Properties panel.

Also note that the plot was for the maximum EQPS value at each timestep, which could potentially be from a different cell at every timestep. If the desire is to identify a cell with a maximum value at some timestep and then plot that cell's value at every timestep, then use the Freeze Selection feature demonstrated in Exercise 2.28. ◆

You can also extract a selection in order to view the selected points or cells separately or perform some independent processing on them. This is done through the Extract Selection ⬚ filter.

Exercise 2.31: Extracting a Selection

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to select ⟳ from the toolbar.

1. Open the file can.ex2, load all variables, ⬚ Apply (see Exercise 2.19).

2. Make a sizable cell selection for example, with Select Cells Through ⬚.

3. Create an Extract Selection ⬚ filter (available on the toolbar and from the find data dialog box).

4. ⬚ Apply.
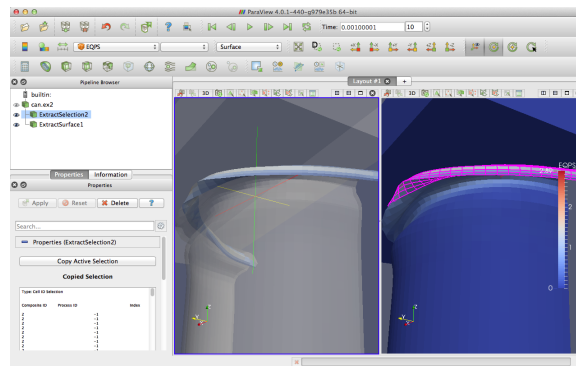
The object in the view is replaced with the cells that you just selected. (Note that in this image I added a translucent surface and a second view with the original selection to show the extracted cells in relation to the full data.) You can perform computations on the extracted cells by simply adding filters to the extract selection pipeline object.   ♦

2.15   Animations

We have already seen how to animate a data set with time in it (hit ▷) and other ways to manipulate temporal data in Section 2.11. However, Para-

View's animation capabilities go far beyond that. With ParaView you can animate nearly any property of any pipeline object.

Exercise 2.32: Animating Properties

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to select ⟳ from the toolbar.

1. Use the quick launch (ctrl+space Win/Linux, alt+space Mac) to create the Sphere source and Apply it.

2. If the animation view is not visible, make it so: View → Animation View.

3. Change the No. Frames option to 50 (10 will go far too quickly).

4. Find the property selection widgets at the bottom of the animation view and select Sphere1 in the first box and Start Theta in the second box.

Hit the ✚ button.
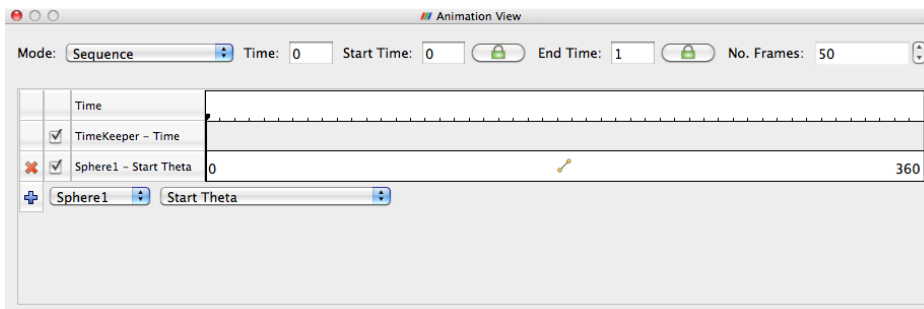
If you play ▷ the animation, you will see the sphere open up then eventually wrap around itself and disappear.

♦
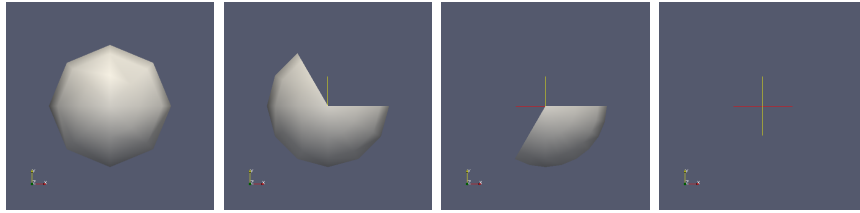
What you have done is created a **track** for the Start Theta property of the Sphere1 object. A track is represented as horizontal bars in the animation view. They hold **key frames** that specify values for the property at a specific time instance. The value for the property is interpolated between the key frames. When you created a track two key frames were created automatically: a key frame at the start time with the minimal value and a key frame at the end time with the maximal value. The property you set here defines the start range of the sphere.

You can modify a track by double clicking on it. That will bring up a dialog box that you can use to add, delete, and modify key frames.
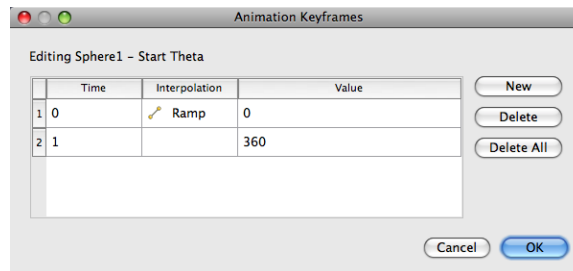
We use this feature to create a new key frame in the animation in the next exercise.

Exercise 2.33: Modifying Animation Track Keyframes

This exercise is a continuation of Exercise 2.32. You will need to finish that exercise before beginning this one.

1. Double-click on the Sphere1 – Start Theta track.

2. In the Animation Keyframes dialog, click the New button. This will create a new key frame.

3. Modify the first key frame value to be 360.

4. Modify the second key frame time to be 0.5 and value to be 0.

5. Click OK.
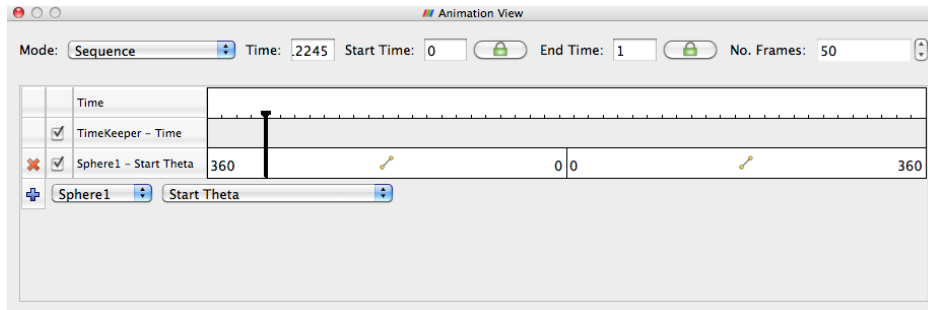
When you play the animation, the sphere will first get bigger and then get smaller again.   ♦

You are not limited to animating just one property. You can animate any number of properties you wish. Now we will create an animation that depends on modifying two properties.

Exercise 2.34: Multiple Animation Tracks

This exercise is a continuation of Exercises 2.32 and 2.33. You will need to finish those exercises before beginning this one.

1. Double-click on the Sphere1 – Start Theta track.

2. In the Animation Keyframes dialog, Delete the first track (at time step 0).

3. Click OK.

4. In the animation view, create a track for the Sphere1 object, End Theta property.

5. Double-click on the Sphere1 – End Theta track.

6. Change the time for the second key frame to be 0.5.
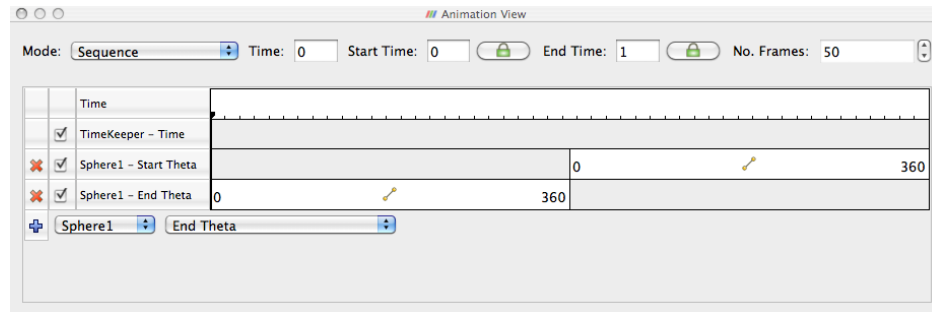
The animation will show the sphere creating and destroying itself, but this time the range front rotates in the same direction. It makes for a very satisfying animation when you loop 🔁 the animation.   ♦

In addition to animating properties for pipeline objects, you can animate the camera. ParaView provides methods for animating the camera along curves that you specify. The most common animation is to rotate the camera around an object, always facing the object, and ParaView provides a means to automatically generate such an animation.

Exercise 2.35: Camera Orbit Animations

For this exercise, we will orbit the camera around whatever data you have loaded. If you are continuing from the previous exercise, you are set up. If not, simply load or create some data. To see the effects, it is best to have asymmetry in the geometry you load. can.ex2 is a good data set to load for this exercise.

1. Place the camera where you want the orbit to start. The camera will move to the right around the viewpoint.

2. Make sure the animation view panel is visible (View → Animation View if it is not).

3. In the property selection widgets, select Camera in the first box and Orbit in the second box.
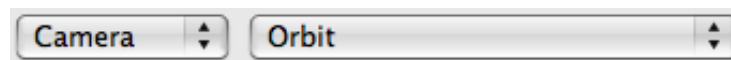
Hit the ➕ button.
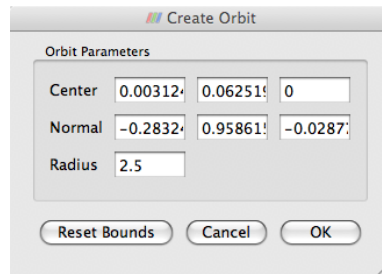
Before the new track is created, you will be presented with a dialog box that specifies the parameters of the orbit. The default values come from the current camera position, which is usually what you want.

4. Click OK.

5. Play ▷.

The camera will now animate around the object.   ◆

Another common camera annotation is to follow an object as it moves through space. Imagine a simulation of a traveling bullet or vehicle. If we hold the camera steady, then the object will quickly move out of view. To help with this situation, ParaView provides a special track that allows the camera to follow the data in the scene.

Exercise 2.36: Following Data in an Animation

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to select ⟳ from the toolbar.

1. Open the file can.ex2, load all variables, 🔄 Apply (see Exercise 2.19).

2. Press the ⬆+Y button to orient the camera to the mesh.

3. Make sure the animation view panel is visible (View → Animation View if it is not).

4. In the property selection widgets, select Camera in the first box and Follow Data in the second box.

Camera ⬍   Follow Data   ⬍
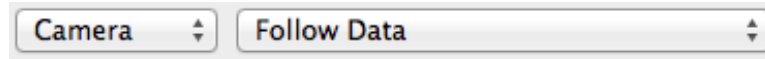
Hit the ✚ button.

5. Play ▷.

Note that instead of the can crushing to the bottom of the view, the animation shows the can lifted up to be continually centered in the image. This is because the camera is following the can down as it is crushed.   ◆

Chapter 3

Batch Python Scripting

Python scripting can be leveraged in two ways within ParaView. First, Python scripts can automate the setup and execution of visualizations by performing the same actions as a user at the GUI. Second, Python scripts can be run inside pipeline objects, thereby performing parallel visualization algorithms. This chapter describes the first mode, batch scripting for automating the visualization.

Batch scripting is a good way to automate mundane or repetitive tasks, but it is also a critical component when using ParaView in situations where the GUI is undesired or unavailable. The automation of Python scripts allows you to leverage ParaView as a scalable parallel post-processing framework. We are also leveraging Python scripting to establish *in situ* computation within simulation code. (ParaView supports an *in situ* library called **Catalyst**, which is documented in Section 4.10 of this tutorial and more fully online at http://www.paraview.org/in-situ/.

This tutorial gives only a brief introduction to Python scripting. More comprehensive documentation on scripting is given in the *ParaView User's Guide*. There are also further links on ParaView's documentation web page (http://www.paraview.org/documentation) including a complete reference to the ParaView Python API.

3.1   Starting the Python Interpreter

There are many ways to invoke the Python interpreter. The method you use depends on how you are using the scripting. The easiest way to get a

python interpreter, and the method we use in this tutorial, is to select View → Python Shell from the menu. This will bring up a dialog box containing controls for ParaView's Python shell. This is the Python interpreter, where you directly control ParaView via the commands described later. For convenience in typing, ParaView's Python shell supports tab completion and history browsing with the up and down keys.
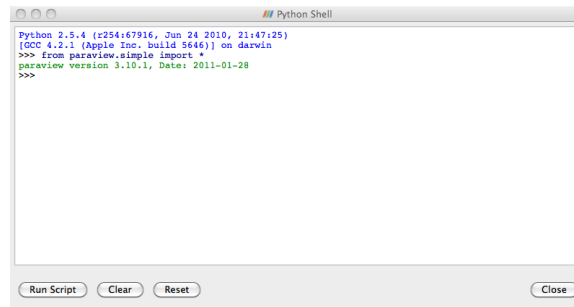
If you are most interested in getting started on writing scripts, feel free to skip to the next section past the discussion of the other ways to invoke scripting.

ParaView comes with two command line programs that execute Python scripts: `pvpython` and `pvbatch`. They are similar to the `python` executable that comes with Python distributions in that they accept Python scripts either from the command line or from a file and they feed the scripts to the Python interpreter.

The difference between `pvpython` and `pvbatch` is subtle and has to do with the way they establish the visualization service. `pvpython` is roughly equivalent to the `paraview` client GUI with the GUI replaced with the Python interpreter. It is a serial application that connects to a ParaView server (which can be either builtin or remote). `pvbatch` is roughly equivalent to `pvserver` except that commands are taken from a Python script rather than from a socket connection to a ParaView client. It is a parallel application that can be launched with `mpirun` (assuming it was compiled with MPI), but it cannot connect to another server; it is its own server. In general, you should use `pvpython` if you will be using the interpreter interactively and `pvbatch` if you are running in parallel.

It is also possible to use the ParaView Python modules from programs outside of ParaView. This can be done by pointing the `PYTHONPATH` environment variable to the location of the ParaView libraries and Python modules

and pointing the `LD_LIBRARY_PATH` (on Unix/Linux), `DYLD_LIBRARY_PATH` (on Mac), or `PATH` (on Windows) environment variable to the ParaView libraries. Running the Python script this way allows you to take advantage of third-party applications such as IDLE. For more information on setting up your environment, consult the ParaView Wiki.

3.2   Tracing ParaView State

Before diving into the depths of the Python scripting features, let us take a moment to explore the automated facilities for creating Python scripts. The ParaView GUI's Python Trace feature allows one to very easily create Python scripts for many common tasks. To use Trace, one simply begins a trace recording via Start Trace, found in the Tools menu, and ends a trace recording via Stop Trace, also found in the Tools menu. This produces a Python script that reconstructs the actions taken in the GUI. That script contains the same set of operations that we are about to describe. As such, Trace recordings are a good resource when you are trying to figure out how to do some action via the Python interface, and conversely the following descriptions will help in understanding the contents of any given Trace script.

Exercise 3.1: Creating a Python Script Trace

If you have been following an exercise in a previous section, now is a good time to reset ParaView. The easiest way to do this is to select Edit → Reset Session from the menu.

1. Click the Start Trace option in the Tools menu.

2. A dialog box with options for the trace is presented. We will discuss the meaning of these options later. For now, just click OK.

3. Build a simple pipeline in the main ParaView GUI. For example, create a sphere source and then clip it.

4. Click Stop Trace in the Tools menu.

5. An editing window will open populated with a Python script that replicates the operations you just made.

Even if you have not been exposed to ParaView's Python bindings, the commands being performed in the traced script should be familiar. Once saved to your hard drive, you can of course edit the script with your favorite editor. The final script can be interpreted by the `pvpython` or `pvbatch` program for totally automated visualization. It is also possible to run this script in the GUI. The Python Shell dialog has a Run Script button that invokes a saved script.   ♦

It should be noted that there is also a way to capture the current Para-View state as a Python script without tracing actions. Simply select Save State... from the ParaView File menu and choose to save as a Python .py state file (as opposed to a ParaView .pvsm state file). We will not have an exercise on state Python scripts, but suffice it to say they can be used in much the same way as traced Python scripts. You are welcome to experiment with this feature as you like.
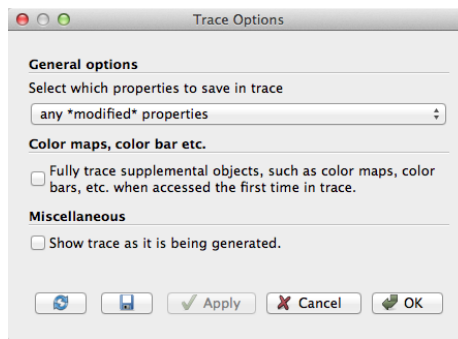
As noted earlier in the exercise, Python tracing has some options that are presented in a dialog box before the tracing starts. The first option selections what properties are saved to the trace. Some properties you explicitly set through the GUI, such as a value entered in a GUI widget. Some properties are set internally by the ParaView application, such as the initial position of a clip plane based on the bounds of the object it is being applied to. Many other properties are left at some default value. You can choose one of the following classes of properties to save:

all properties Traces the values of all properties even if they remain at the default. This can be helpful to introspect all possible properties or to ensure a consistent state regardless of the settings for other users. This also yields a very verbose output that can be hard to read.

any *modified* properties Ignores any properties that do not change from
their defaults. This is a good option for most use cases.

only *user-modified* properties Ignores any properties that are not explic-
itly set by the user. Traces of this nature rely on any internally set
properties being reapplied when the script is run.

The next option has to do with supplemental objects that are managed
by the ParaView GUI (or client) rather than in the server's state. Check this
box to capture all of the state associated with these objects, which includes
color maps, color bars, and other annotation.

Finally, ParaView provides the option to show the trace file as it is being
generated. This can be a helpful option to use when learning what Python
commands can be used to replicate particular actions in the ParaView GUI.

3.3   Macros

A simple but powerful way to customize the behavior of ParaView is to add
your Python script as a **macro**. A macro is simply an automated script that
can be invoked through its button in a toolbar or its entry in the menu bar.
Any Python script can be assigned to a macro.

Exercise 3.2: Adding a Macro

This exercise is a continuation of Exercise 3.1. You will need to finish that
exercise before beginning this one. You should have the editing window
containing the Python script created in Exercise 3.1 open.

1. In the menu bar (of the editing window), select File → Save As Macro....

2. Choose a descriptive name for the macro file and save it in the default
directory provided by the browser. You should now see your macro on
the Macro toolbar at the top of the ParaView GUI.

At this point, you should now see your macro added to the toolbars. By
default, macro toolbar buttons are placed in the middle row all the way to
the right. If you are short on space in your GUI, you may need to move
toolbars around to see it. You will also see that your macro has been added
to the Macros menu.

3. Close the Python editor window.

4. Delete the pipeline you have created by either selecting Edit → Delete All from the menu or selecting ⟳ from the toolbar.

5. Activate your macro by clicking on the toolbar button or selecting it in the Macros menu.

In this example our macro created something from scratch. This is helpful if you often load some data in the same way every time. You can also trace the creation of filters that are applied to existing data. A macro from a trace of this nature allows you to automate the same visualization on different data. ♦

3.4   Creating a Pipeline

As described in the previous two sections, the ParaView GUI's Python Trace feature provides a simple mechanism to create scripts. In this section we will begin to describe the basic bindings for ParaView scripting. This is important information in building Python scripts, but you can always fall back on producing traces with the GUI.

The first thing any ParaView Python script must do is load the `paraview.simple` module. This is done by invoking

```
from paraview.simple import *
```

In general, this command needs to be invoked at the beginning of any ParaView batch Python script. This command is automatically invoked for you when you bring up the scripting dialog in ParaView, but you must add it yourself when using the Python interpreter in other programs (including `pvpython` and `pvbatch`).

The `paraview.simple` module defines a function for every source, reader, filter, and writer defined in ParaView. The function will be the same name as shown in the GUI menus with spaces and special characters removed. For example, the `Sphere` function corresponds to Sources → Sphere in the GUI and the `PlotOverLine` function corresponds to Filters → Data Analysis → Plot Over Line. Each function creates a pipeline object, which will show up in the pipeline browser (with the exception of writers), and returns an object

that is a **proxy** that can be used to query and manipulate the properties of that pipeline object.

There are also several other functions in the `paraview.simple` module that perform other manipulations. For example, the pair of functions `Show` and `Hide` turn on and off, respectively, the visibility of a pipeline object in a view. The `Render` function causes a view to be redrawn.

To obtain a concise list of the functions available in `paraview.simple`, invoke `dir(paraview.simple)`. Alternatively, as explained in Section 3.6 you can get a verbose listing via `help(paraview.simple)`.

Exercise 3.3: Creating and Showing a Source

If you have been following an exercise in a previous section, now is a good time to reset ParaView. The easiest way to do this is to select ⟳ from the toolbar.

If you have not already done so, open the Python shell in the ParaView GUI by selecting View → Python Shell from the menu. You will notice that

```
from paraview.simple import *
```

has been added for you.

Create and show a Sphere source by typing the following in the Python shell.

```
sphere = Sphere()
Show()
Render()
ResetCamera()
```

The `Sphere` command creates a sphere pipeline object. Once it is executed you will see an item in the pipeline browser created. We save a proxy to the pipeline object in the variable `sphere`. We are not using this variable (yet), but it is good practice to save references to your pipeline objects.

The subsequent `Show` command turns on visibility of this object in the view, and the subsequent `Render` causes the results to be seen. Finally, although the ParaView GUI automatically adjusts the camera to data shown for the first time, the Python scripting does not. The call to `ResetCamera` performs this automatic camera adjustment if necessary.

At this point you can interact directly with the GUI again. Try changing the camera angle in the view with the mouse. ◆

Exercise 3.4: Creating and Showing a Filter

Creating filters is almost identical to creating sources. By default, the last created pipeline object will be set as the input to the newly created filter, much like when creating filters in the GUI.

This exercise is a continuation of Exercise 3.3. You will need to finish that exercise before beginning this one.

Type in the following script in the Python shell that hides the sphere and then adds the shrink filter to the sphere and shows that.

```
Hide()
shrink = Shrink()
Show()
Render()
```

The sphere should be replaced with the output of the `Shrink` filter, which makes all of the polygons smaller to give the mesh an exploded type of appearance. ◆

So far as we have built pipelines we have accepted the default parameters for the pipeline objects. As we have seen in the exercises of Chapter 2, it is common to have to modify the parameters of the objects using the properties panel.

In Python scripting, we use the **proxy** returned from the creation functions to manipulate the pipeline objects. These proxies are in fact Python objects with class attributes that correspond to the same properties you set in the properties panel. They have the same names as those in the properties panel with spaces and other illegal characters removed. Use `dir(variable)` or `help(variable)` to get a list of all attributes on any variable that you have access to. In most cases, simply assign values to an object's attributes in order to change them.

Exercise 3.5: Changing Pipeline Object Properties

This exercise is a continuation of Exercises 3.3 and 3.4. You will need to finish those exercises before beginning this one.

Recall that we have so far created two Python variables, `sphere` and `shrink`, that are proxies to the corresponding pipeline objects. First, enter the following command into the Python shell to get a concise listing of all attributes of the sphere.

```
dir(sphere)
```

Next, enter the following command into the Python shell to get the current value of the Theta Resolution property of the sphere.

```
print sphere.ThetaResolution
```

The Python interpreter should respond with the result 8. (Note that using the `print` keyword, which instructs Python to output the arguments to standard out, is superfluous here as the Python shell will output the result of any command anyway.) Let us double the number of polygons around the equator of the sphere by changing this property.

```
sphere.ThetaResolution = 16
Render()
```

The shrink filter has only one property, Shrink Factor. We can adjust this factor to make the size of the polygons larger or smaller. Let us change the factor to make the polygons smaller.

```
shrink.ShrinkFactor = 0.25
Render()
```

You may have noticed that as you type in Python commands to change the pipeline object properties, the GUI in the properties panel updates accordingly.   ♦

So far we have created only non-branching pipelines. This is a simple and common case and, like many other things in the `paraview.simple` module, is designed to minimize the amount of work for the simple and common case but also provide a clear path to the more complicated cases. As we have built the non-branching pipeline, ParaView has automatically connected the filter input to the previously created object so that the script reads like the sequence of operations it is. However, if the pipeline has branching, we need to be more specific about the filter inputs.

Exercise 3.6: Branching Pipelines

This exercise is a continuation of Exercises 3.3 through 3.5. You will need to finish Exercises 3.3 and 3.4 before beginning this one (Exercise 3.5 is optional).

Recall that we have so far created two Python variables, `sphere` and `shrink`, that are proxies to the corresponding pipeline objects. We will now add a second filter to the sphere source that will extract the wireframe from it. Enter the following in the Python shell.

```
wireframe = ExtractEdges(Input=sphere)
Show()
Render()
```

An **Extract Edges** filter is added to the sphere source. You should now see both the wireframe of the original sphere and the shrunken polygons.

Notice that we explicitly set the input for the **Extract Edges** filter by providing `Input=sphere` as an argument to the `ExtractEdges` function. What we are really doing is setting the `Input` property upon construction of the object. Although it would be possible to create the object with the default input, and then set the input later, it is not recommended. The problem is that not all filters accept all input. If you initially create a filter with the wrong input, you could get error messages before you get a chance to change the `Input` property to the correct input.

The sphere source having two filters connected to its output is an example of **fan out** in the pipeline. It is always possible to have multiple filters attached to a single output. Some filters, but not all, also support having multiple filters connected to their input. Multiple filters are attached to an input is known as **fan in**. In ParaView's Python scripting, fan in is handled much like fan out, by explicitly defining a filter's inputs. When setting multiple inputs (on a single port[1]), simply set the input to a list of pipeline objects rather than a single one. For example, let us group the results of the shrink and extract edges filters using the **Group Datasets** filter. Type the following line in the Python shell.

```
group = GroupDatasets(Input=[shrink,wireframe])
Show()
```

---

[1] Filters that have multiple input ports, like `ResampleWithDataset`, use different names to distinguish amongst the input properties instead. The ports are typically called "Input" and "Source" but consult `Trace` or `help` to be sure.

There is now no longer any reason for showing the shrink and extract edges filters, so let us hide them. By default, the `Show` and `Hide` functions operate on the last pipeline object created (much like the default input when creating a filter), but you can explicitly choose the object by giving it as an argument. To hide the shrink and extract edges filters, type the following in the Python shell.

```
Hide(shrink)
Hide(wireframe)
Render()
```

<div align="right">♦</div>

In the previous exercise, we saw that we could set the `Input` property by placing `Input=⟨input object⟩` in the arguments of the creator function. In general we can set any of the properties at object construction by specifying ⟨*property name*⟩=⟨*property value*⟩. For example, we can set both the Theta Resolution and Phi Resolution when we create a sphere with a line like this.

```
sphere = Sphere(ThetaResolution=360, PhiResolution=180)
```

3.5   Active Objects

If you have any experience with the ParaView GUI, then you should already be familiar with the concept of an active object. As you build and manipulate visualizations within the GUI, you first have to select an object in the pipeline browser. Other GUI panels such as the properties panel will change based on what the active object is. The active object is also used as the default object to use for some operations such as adding a filter.

The batch Python scripting also understands the concept of the active object. In fact, when running together, the GUI and the Python interpreter share the same active object. When you created filters in the previous section, the default input they were given was actually the active object. When you created a new pipeline object, that new object became the active one (just like when you create an object in the GUI).

You can get and set the active object with the `GetActiveSource` and `SetActiveSource` functions, respectively. You can also get a list of all pipeline objects with the `GetSources` function. When you click on a new object in

the GUI pipeline browser, the active object in Python will change. Likewise, if you call `SetActiveSource` in python, you will see the corresponding entry become highlighted in the pipeline browser.

Exercise 3.7: Experiment with Active Pipeline Objects

This exercise is a continuation of the exercises in the previous section. However, if you prefer you can create any pipeline you want and follow along.

Play with active objects by trying the following.

- Get a list of objects by calling `GetSources()`. Find the sources and filters you created in that list.

- Get the active object by calling `GetActiveSource()`. Compare that to what is selected in the pipeline browser.

- Select something new in the pipeline browser and call `GetActiveSource()` again.

- Change the active object with the `SetActiveSource()` function. You can use one of the proxy objects you created earlier as an argument to `SetActiveSource`. Observe the change in the pipeline browser.

$$\blacklozenge$$

In addition to maintaining an active pipeline object, ParaView Python scripting also maintains an active view. As a ParaView user, you should also already be familiar with multiple views and the active view. The active view is marked in the GUI with a blue border. The Python functions `GetActiveView` and `SetActiveView` allow you to query and change the active view. As with pipeline objects, the active view is synchronized between the GUI and the Python interpreter.

3.6   Online Help

This tutorial, as well as similar instructions in the ParaView book and Wiki, is designed to give the key concepts necessary to understand and create batch Python scripts. The detailed documentation including complete lists of functions, classes, and properties available is maintained by the ParaView build

process and provided as online help from within the ParaView application. In this way we can ensure that the documentation is up to date for whatever version of ParaView you are using and that it is easily accessible.

The ParaView Python bindings make use of the `help` built-in function. This function takes as an argument any Python object and returns some documentation on it. For example, typing

```
help(paraview.simple)
```

returns a brief description of the module and then a list of all the functions included in the module with a brief synopsis of what each one does. For example

```
help(Sphere)
sphere = Sphere()
help(sphere)
```

will first give help on the `Sphere` function, then use it to create an object, and then give help on the object that was returned (including a list of all the properties for the proxy).

Most of the widgets displayed in the properties panel's **Properties** group are automatically generated from the same introspection that builds the Python classes. (There are a small number of exceptions where a custom panel was created for better usability.) Thus, if you see a labeled widget in the properties panel, there is a good chance that there is a corresponding property in the Python object with the same name.

Regardless of whether the GUI contains a custom panel for a pipeline object, you can still get information about that object's properties from the GUI's online help. As always, bring up the help with the ![help icon] toolbar button. You can find documentation for all the available pipeline objects under the **Sources**, **Filters**, **Readers**, and **Writers** entries in the help **Contents**. Each entry gives a list of objects of that type. Clicking on any one of the objects gives a list of the properties you can set from within Python.
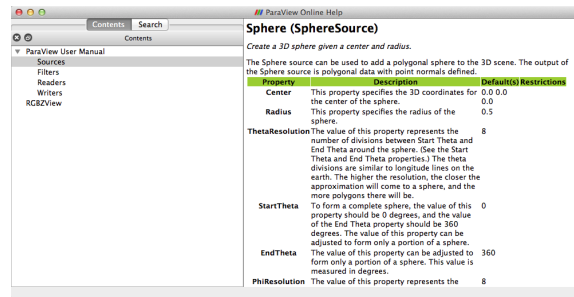
3.7 Reading from Files

The equivalent to opening a file in the ParaView GUI is to create a reader in Python scripting. Reader objects are created in much the same way as sources and filters; `paraview.simple` has a function for each reader type that creates the pipeline object and returns a proxy object. One can instantiate any given reader directly as described below, or more simply call `reader = OpenDataFile(`⟨*filename*⟩`)`

All reader objects have at least one property (hidden in the GUI) that specifies the file name. This property is conventionally called either `FileName` or `FileNames`. You should always specify a valid file name when creating a reader by placing something like `FileName=`⟨*full path*⟩ in the arguments of the construction object. Readers often do not initialize correctly if not given a valid file name.

Exercise 3.8: Creating a Reader

We are going to start a fresh visualization, so if you have been following along with the exercises so far, now is a good time to reset ParaView. The easiest way to do this is to select ⏻ from the toolbar. You will also need the Python shell. If you have not already done so, open it with View → Python Shell from the menu.

In this exercise we are loading the disk_out_ref.ex2 file from the Python shell. Locate this file on your computer and be ready to type or copy it into the Python shell. We will reference it as ⟨*path*⟩/`disk_out_ref.ex2`. You can use the file browser to help you locate this file. (Click on the Examples quick access directory and observe where the file browser takes you.) Common paths for the file are as follows:

Mac /Applications/ParaView-5.6.0.app/Contents/examples

Windows C:/Program Files/ParaView 5.6.0/examples

Linux /usr/local/lib/paraview-5.6/examples

Create the reader while specifying the file name by entering the following in the Python shell.

```
reader = OpenDataFile('⟨path⟩/disk_out_ref.ex2')
Show()
Render()
ResetCamera()
```

♦

3.8 Querying Field Attributes

In addition to having properties specific to the class, all proxies for pipeline objects share a set of common properties and methods. Two very important such properties are the `PointData` and `CellData` properties. These properties act like **dictionaries**, an associative array type in Python, that maps variable names (in strings) to `ArrayInformation` objects that hold some characteristics of the fields. Of particular note are the `ArrayInformation` methods `GetName`, which returns the name of the field, `GetNumberOfComponents`, which returns the size of each field value (1 for scalars, more for vectors), and `GetRange`, which returns the minimum and maximum values for a particular component.

Exercise 3.9: Getting Field Information

This exercise is a continuation of Exercise 3.8. You will need to finish that exercise before beginning this one.

To start with, get a handle to the point data and print out all of the point fields available.

```
pd = reader.PointData
print pd.keys()
```

Get some information about the "Pres" and "V" fields.

```
print pd['Pres'].GetNumberOfComponents()
print pd['Pres'].GetRange()
print pd['V'].GetNumberOfComponents()
```

Now let us get fancy. Use the Python `for` construct to iterate over all of the arrays and print the ranges for all the components.

```
for ai in pd.values():
    print ai.GetName(), ai.GetNumberOfComponents(),
    for i in xrange(ai.GetNumberOfComponents()):
        print ai.GetRange(i),
    print
```

<div align="right">♦</div>

3.9   Representations

Representations are the "glue" between the data in a pipeline object and a view. The representation is responsible for managing how a data set is drawn in the view. The representation defines and manages the underlying rendering objects used to draw the data as well as other rendering properties such as coloring and lighting. Parameters made available in the Display group of the GUI are managed by representations. There is a separate representation object instance for every pipeline-object–view pair. This is so that each view can display the data differently.

Representations are created automatically by the GUI. In python scripting they are created with the `Show` function instead. In fact `Show` returns a proxy to the representation. Therefore you can save `Show`'s return value in a variable as we've done above for sources, filters and readers. If you neglect to save it, you can always get it back with the `GetRepresentation` function. With no arguments, this function will return the representation for the active pipeline object and the active view. You can also specify a pipeline object or view or both.

Exercise 3.10: Coloring Data

This exercise is a continuation of Exercise 3.8 (and optionally Exercise 3.9). If you do not have the exodus file open, you will need to finish Exercise 3.8 before beginning this one.

Let us change the color of the geometry to blue and give it a very pronounced specular highlight (that is, make it really shiny). Type in the following into the Python shell to get the representation and change the material properties.

```
readerRep = GetRepresentation()
readerRep.DiffuseColor = [0, 0, 1]
readerRep.SpecularColor = [1, 1, 1]
readerRep.SpecularPower = 128
readerRep.Specular = 1
Render()
```

Now rotate the camera with the mouse in the GUI to see the effect of the specular highlighting.

We can also use the representation to color by a field variable. There is actually quite a bit of state that must be set to change field variables. The `ColorBy` function provides a simple interface to color a mesh by a field value. A subsequent call to `UpdateScalarBars` also updates the color bar annotation. Enter the following into the Python shell to color the mesh by the "Pres" field variable.

```
ColorBy(readerRep, ('POINTS', 'Pres'))
UpdateScalarBars()
Render()
```

♦

3.10 Views

Drawing areas or windows are called Views in ParaView. As with readers, sources, filters, and representations, views are wrapped into python objects and these can be created, obtained and controlled via scripts.

Views are usually created for you by the GUI, but in python you have to create views more intentionally. The most convenient way to do so is to rely on the way that `Render` returns a view, creating one first if necessary. If you prefer, you can create specific view types via `CreateView('`⟨*viewname*⟩`')` or `CreateRenderView`, `CreateXYPlotView` and the like. However you make them, call `GetRenderViews` to get a list of all Views, or `GetActiveView` get access to the currently active view. There is also a function named `GetRenderView` (no 's' on the end) that that gets the active view if there is one or creates a new one if there is no active view.

Once you have a view you have access to all of the properties that you see on the View group of the GUI. For instance you can easily turn on and off the orientation widget, change the background color, alter the lighting and more. Besides these first level properties, the view also gives you access to other scene wide controls such as the camera, animation time, and when not running alongside the GUI, the view's size.

Exercise 3.11: Controlling the View

This exercise is a continuation of Exercise 3.8 (and optionally Exercises 3.9 and 3.10). If you do not have the exodus file open, you will need to finish Exercise 3.8 before beginning this one.

Let us change the background color of the scene from ParaView's default gray to a nice gradient instead. Type the following into the Python shell to get a hold of the View and change it.

```
view = GetActiveView()
view.Background = [0, 0, 0]
view.Background2 = [0, 0, 0.6]
view.UseGradientBackground = True
Render()
```

Next, let us ask the view what position the camera is sitting at, and then move it within a for loop to create a short animation.

```
x,y,z = view.CameraPosition
print x,y,z
for iter in xrange(0,10):
  x = x + 1
  y = y + 1
```

```
z = z + 1
view.CameraPosition = [x,y,z]
print x,y,z
Render()
```

♦

3.11 Saving Results

Within a script it is easy to save out results, and by saving your data and your scripts it becomes easy to create reproducible visualization with ParaView.

As within the GUI, there are several products that you might like to save out when you are working with ParaView.

- To save out the data produced by a filter, add a writer to the filter with the desired filename using the `CreateWriter` function and then update the writer proxy with the `UpdatePipeline` method. This is analogous to clicking on a pipeline element and selecting File → Save Data.

- Saving images is as simple as typing `SaveScreen-shot('`⟨*path*⟩`/filename.still_extension')`.

- Assuming your ParaView is linked to an encoder and codecs, saving compressed animations is as simple as typing `WriteAnima-tion('`⟨*path*⟩`/filename.animation_extension')`.

In all cases ParaView uses the file name extension to determine the specific file type to create.

Exercise 3.12: Save Results

This exercise is a continuation of Exercise 3.8 (and optionally Exercises 3.9 through 3.11). If you do not have the exodus file open, you will need to finish Exercise 3.8 before beginning this one.

Let us first probe the data to get something compact out of it. Then we will save out the result of the probe in the form of a comma separated values file so that we can look at it in a text editor and import it into any other tool we choose.

```
plot = PlotOverLine()
plot.Source.Point1 = [0,0,0]
plot.Source.Point2 = [0,0,10]
writer = CreateWriter('⟨path⟩/plot.csv')
writer.UpdatePipeline()
```

Next, lets create a LineChartView to show the plot in and then save out a screenshot of our results.

```
plotView = CreateView('XYChartView')
Show(plot)
Render()
SaveScreenshot('⟨path⟩/plot.png')
```

♦

As you can see, ParaView's scripting interface is quite powerful, and once you know the fundamentals and are familiar with Python's syntax, it is fairly easy to get up and running with it. We have just touched on the higher level aspects of ParaView scriptability in this tutorial. More details, including how to run python scripted filters, how to work with numpy and other tools, and how to package your scripts for execution under batch schedulers can be found online.

Chapter 4

Visualizing Large Models

ParaView is used frequently at Sandia National Laboratories and other institutions for visualizing data from large-scale simulations run on the world's largest supercomputers including the examples shown here.
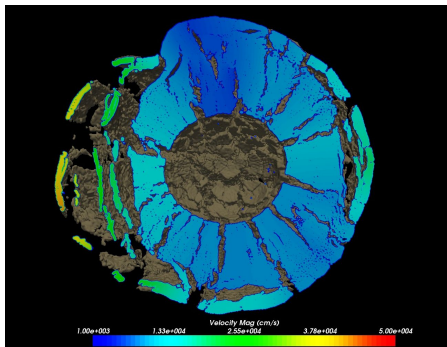
CTH shock physics simulation with over 1 billion cells of a 10 megaton explosion detonated at the center of the Golevka asteroid.
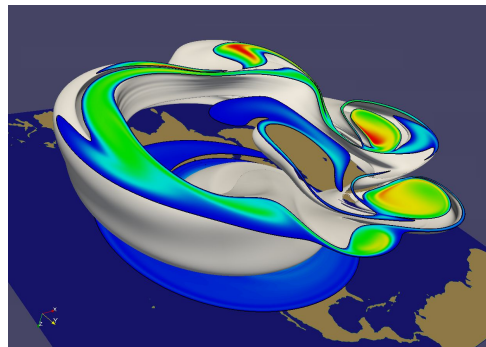
SEAM Climate Modeling simulation with 1 billion cells modeling the breakdown of the polar vortex, a circumpolar jet that traps polar air at high latitudes.
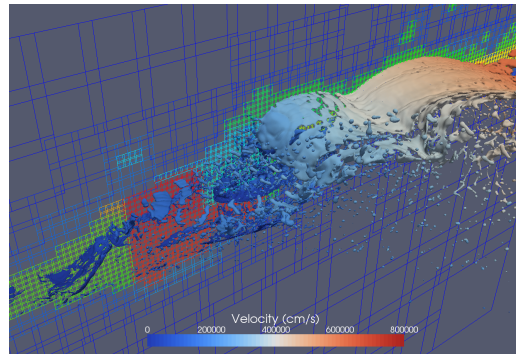
A CTH simulation that generates AMR data. ParaView has been used to visualize CTH simulation AMR data comprising billions of cells, 100's of thousands of blocks, and eleven levels of hierarchy (not shown).
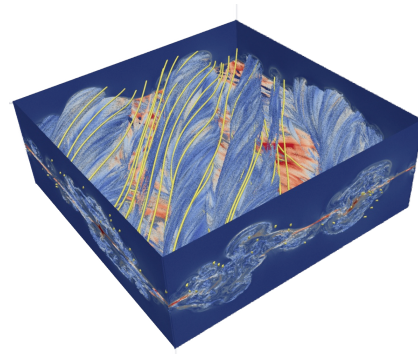
A VPIC simulation of magnetic reconnection with 3.3 billion structured cells. Image courtesy of Bill Daughton, Los Alamos National Laboratory.
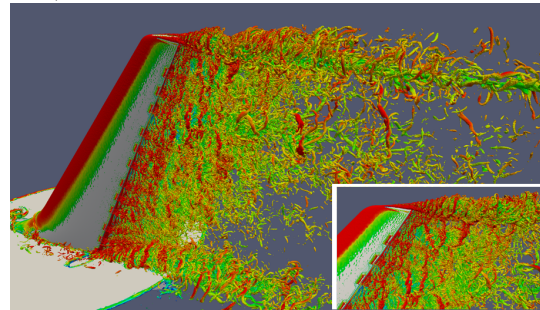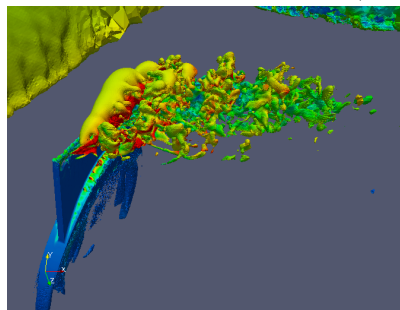
ParaView visualizations run in situ with large scale PHASTA simulations. On the left is a 3.3 billion tetrahedral mesh simulating the flow over a full wing where a synthetic jet issues an unsteady crossflow jet (run on 160 thousand MPI processes). On the right is a 1.3 billion element mesh simulating the wake of a deflected wing flap (run on 256 thousand MPI processes). Images courtesy of Michel Rasquin, Argonne National Laboratory.

In this section we discuss visualizing large meshes like these using the parallel visualization capabilities of ParaView. This section has some exercises, but is less "hands-on" than the previous section. Primarily you will learn the conceptual knowledge needed to perform large parallel visualization. The exercises demonstrate the basic techniques needed to run ParaView on parallel machines.

The most fundamental idea to grasp is that when run on a large machine every node processes different regions of the entire dataset simultaneously. Thus the workable data resolution is limited by the aggregate memory space of the machine. We now present the basic ParaView architecture and parallel algorithms and demonstrate how to apply this knowledge.

4.1 Parallel Visualization Algorithms

We are fortunate in visualization in that many operations are straightoward to parallelize. The data we deal with is contained in a mesh, which means the data is already broken into little pieces by the cells. We can do visualization on a distributed parallel machine by first dividing the cells among the processes. For demonstrative purposes, consider this very simplified mesh.
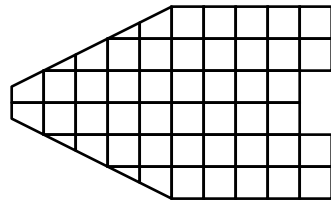
Now let us say we want to perform visualizations on this mesh using three processes. We can divide the cells of the mesh as shown below with the blue, yellow, and pink regions.
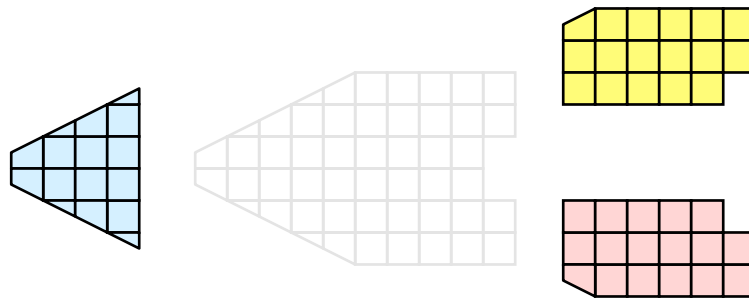
Once partitioned, many visualization algorithms will work by simply allowing each process to independently run the algorithm on its local collection of cells. For example, take clipping (which is demonstrated in multiple exercises including 2.11). Let us say that we define a clipping plane and give that same plane to each of the processes.
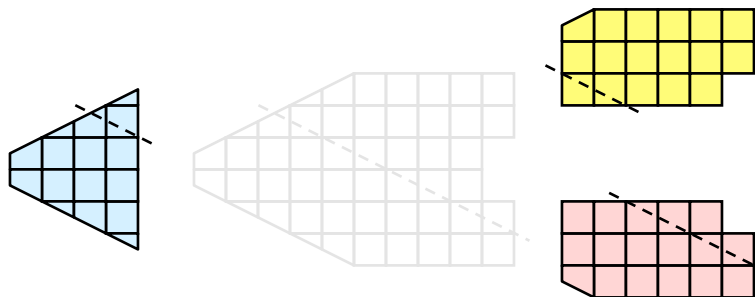
Each process can independently clip its cells with this plane. The end result is the same as if we had done the clipping serially. If we were to bring the cells together (which we would never actually do for large data for obvious reasons) we would see that the clipping operation took place correctly.
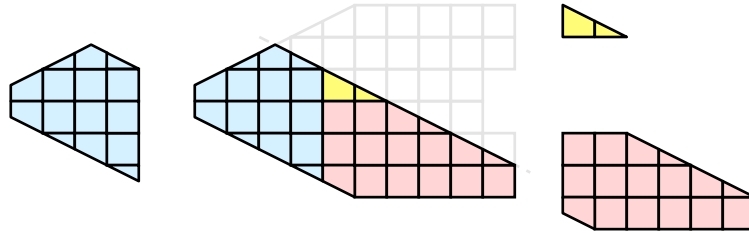
Many, but not all operations are thus trivially parallizeable. Other operations are straightforward to parallelize if ghost layers as described in Section 4.7.2 are available. Other operations require more extensive data sharing among nodes. In these filters ParaView resorts to MPI to communicate amongst the nodes of the machine.

4.2 Basic Parallel Rendering

When performing parallel visualization, we are careful to ensure that the data remains partitioned among all of the processes up to and including the rendering processes. ParaView uses a parallel rendering library called **IceT**. IceT uses a **sort-last** algorithm for parallel rendering. This parallel rendering algorithm has each process independently render its partition of the geometry and then **composites** the partial images together to form the final image.
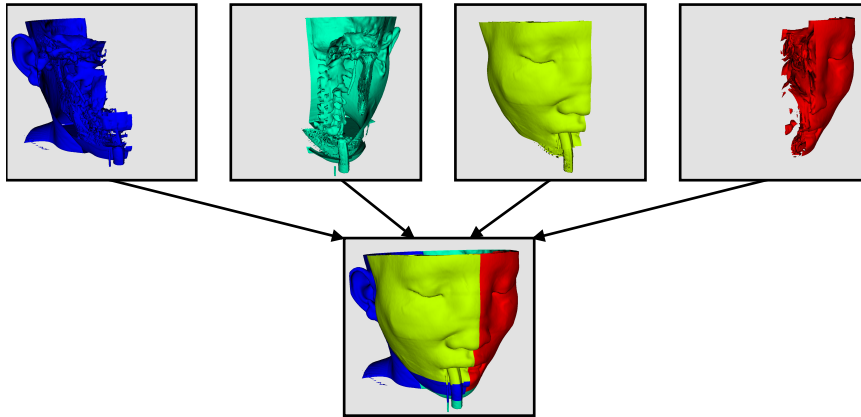
The preceding diagram is an oversimplification. IceT contains multiple parallel image compositing algorithms such as **binary tree**, **binary swap**, and **radix-k** that efficiently divide work among processes using multiple phases.
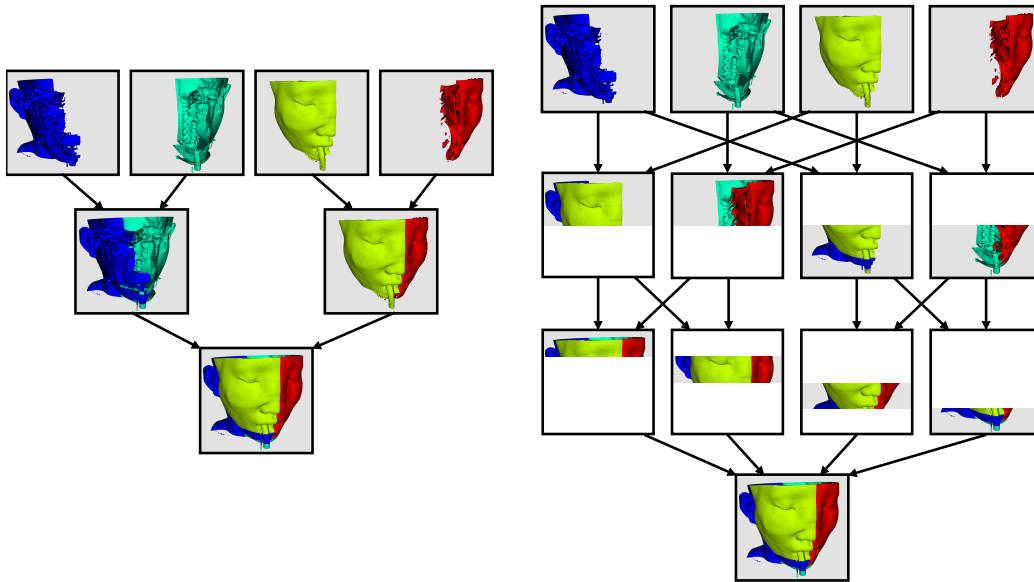
The wonderful thing about sort-last parallel rendering is that its efficiency is completely insensitive to the amount of data being rendered. This makes it a very scalable algorithm and well suited to large data. However, the parallel rendering overhead does increase linearly with the number of pixels in the image. This can be a problem for example when driving tile display walls

with ParaView. Consequently, some of the rendering parameters described
later in this chapter deal with limiting image size.

Because there is an overhead associated with parallel rendering, ParaView
can also render serially and will do so automatically when the visible data
is small enough. When the visible meshes are smaller than a user defined
threshold preference, or when parallel rendering is turned off or unavailable,
the geometry is shipped to the display node, which then rasterizes it locally.
Obviously, this should only happen when the data being rendered is small or
the rendering process will be overwhelmed.

4.3   ParaView Architecture

With this introduction to parallel visualization, it is useful to know something
about how ParaView is structured and how it orchestrates the parallel tasks
described above.

ParaView is designed as a three-tier client-server architecture. The three
logical units of ParaView are as follows.

Data Server The unit responsible for data reading, filtering, and writing.
All of the pipeline objects seen in the pipeline browser are contained
in the data server. The data server can be parallel.

Render Server The unit responsible for rendering. The render server can
also be parallel, in which case built in parallel rendering is also enabled.

Client The unit responsible for establishing visualization. The client con-
trols the object creation, execution, and destruction in the servers, but
does not contain any of the data (thus allowing the servers to scale
without bottlenecking on the client). If there is a GUI, that is also in
the client. The client is always a serial application.

These logical units need not be physically separated. Logical units are
often embedded in the same application, removing the need for any com-
munication between them. There are three modes in which you can run
ParaView. Note that no matter what mode ParaView runs in the user in-
terface and scripting API that you learned in Chapters 2 and 3 undergoes
very little change.
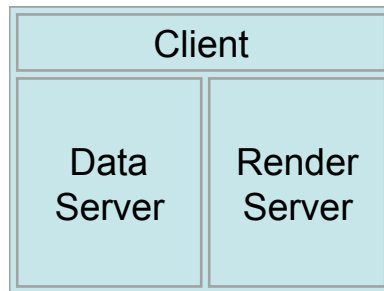
The first mode, which you are already familiar with, is **standalone** mode. In standalone mode, the client, data server, and render server are all combined into a single serial application. When you run the `paraview` application, you are automatically connected to a **builtin** server so that you are ready to use the full features of ParaView.
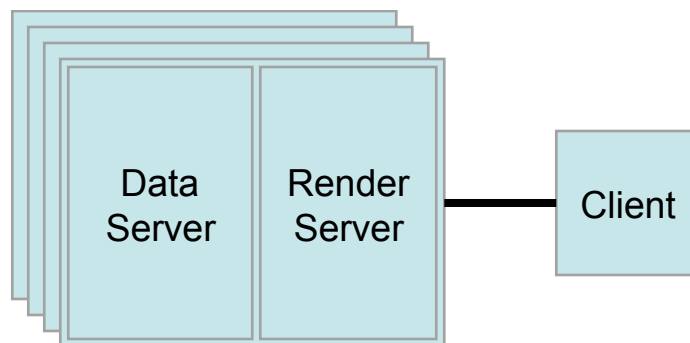
The second mode is **client-server** mode. In client-server mode, you execute the `pvserver` program on a parallel machine and connect to it with the `paraview` client application. The `pvserver` program has both the data server and render server embedded in it, so both data processing and rendering take place there. The client and server are connected via a socket, which is assumed to be a relatively slow mode of communication, so data transfer over this socket is minimized.
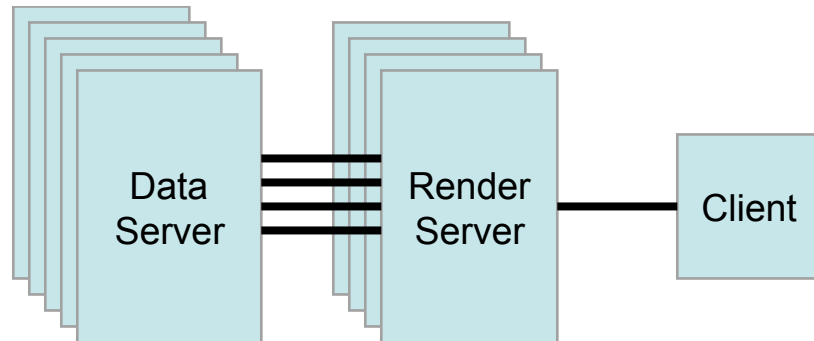
The third mode is **client–render-server–data-server** mode. In this mode, all three logical units are running in separate programs. As before, the client is connected to the render server via a single socket connection. The render server and data server are connected by many socket connections, one for each process in the render server. Data transfer over the sockets is minimized.

Although the client-render server-data server mode is supported, we almost never recommend using it. The original intent of this mode was to take advantage of heterogeneous environments where one might have a large, powerful computational platform and a second smaller parallel machine with graphics hardware in it. However, in practice we find any benefit is almost always outstripped by the time it takes to move geometry from the data server to the render server. If the computational platform is much bigger than the graphics cluster, then use software rendering on the large computational platform. If the two platforms are about the same size just perform all the computation on the graphics cluster.

4.4 Accessing a Parallel ParaView Server

Accessing a standalone installation of ParaView is trivial. Download and install a pre-compiled binary or get it from your package manager, and go. To visualize extreme scale results, you need to access an installation of ParaView, built with the MPI components enabled, on a machine with sufficient aggregate memory to hold the entire data set and derived data products. Accessing a parallel enabled installation of ParaView's server is intrinsically harder than acessing a standalone version.

Recent binary distributions of ParaView provided by Kitware do include MPI for Mac, Linux and optionally Windows. The Windows MPI enabled bi-

naries depend upon Microsoft's MPI, which must be installed separately. We will not cover those in the rest of this tutorial, but feel free to ask questions on the ParaView mailing list.

Note that the specific MPI library version bundled into the binaries were chosen for widest possible compatibility. For production use ParaView will be much more effective when compiled with an MPI version that is tuned for your machine's networking fabric. Your system administrators can help you decide what MPI version to use.

To compile ParaView on a parallel machine you or your system administrators will need the following.

- CMake cross-platform build setup tool (www.cmake.org)

- MPI

- OpenGL, either using a GPU in on- (X11) or off- screen (EGL) modes, or in software via Mesa.

- Python +NumPy +Matplotlib (all optional but strongly recommended)

- Qt $\geq$ 4.7 (optional)

Compiling without one of the optional libraries means a feature will not be available. Compiling without Qt means that you will not have the GUI application and compiling without Python means that you will not have scripting available. NumPy is almost as important as Python itself, and Matplotlib is helpful for numeric text and some types of views and color lookup tables.

To compile ParaView, you first run CMake, which will allow you to set up compilation parameters and point to libraries on your system. This will create the make files that you then use to build ParaView. For more details on building a ParaView server, see the ParaView Wiki.

http://www.paraview.org/Wiki/Setting_up_a_ParaView_Server#Compiling

Compiling ParaView on exotic machines, for example HPC class machines that require cross compilation and some cloud-based on demand systems, can be an arduous task. You are welcome to seek free advice on the ParaView mailing list or contracted assistance from Kitware.

Fortunately there are a number of large scale systems on which ParaView has already been installed. If you have are fortunate enough to have an

account on one of these systems, you shouldn't need to compile anything. The ParaView community maintains an opt in list of these with pointers to system specific documentation on the ParaView Wiki. We invite system maintainers to add to the list if they are permitted to.

http://www.paraview.org/Wiki/HPC_Installations

Running ParaView in parallel is also intrinsically more difficult than running the standalone client. It typically involves a number of steps that change depending on the hardware you are running on: logging in to remote computers, allocating parallel nodes, launching a parallel program, and sometimes tunneling through firewalls to establish interactive connections to the compute nodes. These steps are discussed in more detail in the next two sections in which we finally return from theory to practical exercises.

4.5   Batch Processing

ParaView's Python scripted interface, introduced in Chapter 3 has the important qualities of being runnable offline, with the human removed from the loop, and being inherently reproducible.

On large scale systems there are two steps that you need to master. The first is to run ParaView in parallel through MPI. The second is to submit a job through the system's queuing system. For both steps the syntax varies from machine to machine and you should consult your system's documentation for the details.

We demonstrate how to run ParaView in parallel with the simplest case of running Kitware's binary in parallel on the PC class system that you have at hand.

Exercise 4.1: Running a visualization script in parallel

You will need three things, the `mpiexec` program to spawn MPI parallel programs, the `pvbatch` executable and a Python script.

As for the Python script, let us use an example in which each node draws a portion of a simple polygonal sphere with a color map that varies over the number of processes in the job.

```
from paraview.simple import *
sphere = Sphere()
```

```
rep = Show()
ColorBy(rep, ("POINTS", "vtkProcessId"))
Render()
rep.RescaleTransferFunctionToDataRange(True)
Render()
WriteImage("parasphere.png")
```

Use an editor to type this script into a file called parasphere.py and save it somewhere. Note this is a stripped down version of a script recorded from a parallel ParaView session in which we created a Sources → Sphere and chose the vtkProcessId array to color by.

As for the executables, let us use the ones that come with ParaView. After installing Kitware's ParaView binaries, you will find `mpiexec` and `pvbatch` at:

- On Mac:

  ```
  /Applications/ParaView-5.6.app/Contents/MacOS/mpiexec
  ```

  and

  ```
  /Applications/ParaView-5.6.app/Contents/bin/pvbatch
  ```

- On Linux and assuming you have extracted the binary into /usr/local:

  ```
  /usr/local/lib/paraview-5.6/mpiexec
  ```

  and

  ```
  /usr/local/bin/pvbatch
  ```

  Note that you will need to add ParaView's lib directory (/usr/local/lib/paraview-5.6) to your LD_LIBRARY_PATH to use the mpi that comes with ParaView.

- On Windows and assuming you installed the "MPI" version of ParaView as well as the MS-MPI software (available separately from Microsoft for free):

  ```
  C:/Program Files/Microsoft MPI/Bin/mpiexec
  ```

  and

```
C:/Program Files/ParaView 5.6/bin/pvbatch
```

Now we run the script in parallel by issuing this from the command line.

- On Mac:

```
/Applications/ParaView-5.6.app/Contents/MacOS/mpiexec -np 4 \
   /Applications/ParaView-5.6.app/Contents/bin/pvbatch \
   parasphere.py
```

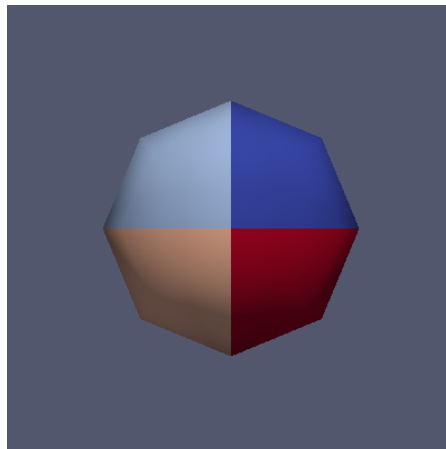- On Linux:

```
/usr/local/lib/paraview-5.6/mpiexec -np 4 \
   /usr/local/bin/pvbatch \
   parasphere.py
```

- On Windows:

```
mpiexec -np 4 ^
   "C:/Program Files/ParaView 5.6/bin/pvbatch" ^
   parasphere.py
```

ParaView will run momentarily and it may or may not briefly display a window on your desktop. In any case you will now find a file named parasphere.png that looks like the following:

♦

Submitting a job through your system's queuing system typically involves issuing a command like the following on the command line.

```
qsub -A <project name to charge to compute time to> \
 -N <number of nodes>  \
 -n <number of processors on each node> \
 mpiexec -np <N*n> \
   pvbatch <arguments for pvbatch> \
   script_to_execute.py <arguments for Python script>
```

The command reserves the requested number of compute nodes on the machine and, at some time in the future when the nodes become ready for your use, spawns an MPI job. The MPI job runs ParaView's Python scripted server in parallel, telling it to process the supplied script.

There are many queuing systems and mpi implementations and site specific policies. Thus it it impossible to provide an exact syntax that will work on every system here. Ask your system administrators for guidance. Once you have the syntax, you should be able to run the sphere example above to determine if you have a working system.

4.6   Interactive Parallel Processing

For day to day work with large datasets it is feasible to do interactive visualization in parallel too. In this mode you can dynamically inspect the data and modify the visualization pipeline while you work with the ParaView GUI on a workstation far from from the HPC resource where the data is being processed.

In interactive configurations the data processing and rendering portions work in parallel like above but they are controled from the ParaView GUI instead of a Python script.

Starting the server process is similar to the above, but instead of using the `pvbatch` executable we use the `pvserver` executable. The former is limited to taking commands from Python scripts, the later is built to take commands from a remote ParaView GUI program.

Client-server connections are established through the `paraview` client application. You connect to servers and disconnect from servers with the 🗒

and 🗔 buttons. When ParaView starts, it automatically connects to the
builtin server. It also connects to builtin whenever it disconnects 🗔 from a
server.

When you hit the 🗔 button, ParaView presents you with a dialog box
containing a list of known servers you may connect to. This list of servers
can be both site- and user-specific.

You can specify how to connect to a server either through the GUI by
pressing the Add Server button or through an XML definition file. There
are several options for specifying server connections, but ultimately you are
giving ParaView a command to launch the server and a host to connect to
after it is launched.

Once more we will demonstrate using the Kitware binaries and then out-
line the steps required on large scale systems where the syntax varies widely.

Exercise 4.2: Interactive Parallel Visualization

The `pvserver` executable can be found in the same directory as the `pvbatch`
exectuable. Let us begin by running it in parallel.

- On Mac:

```
/Applications/ParaView-5.6.app/Contents/MacOS/mpiexec -np 4 \
   /Applications/ParaView-5.6.app/Contents/bin/pvserver &
```

- On Linux:

```
/usr/local/lib/paraview-5.6/mpiexec -np 4 \
   /usr/local/bin/pvserver &
```

- On Windows:

```
mpiexec -np 4 ^
   "C:/Program Files/ParaView 5.6/bin/pvserver"
```

Now start up the ParaView GUI and click the 🗔 button. This will bring
the Choose Server Configuration dialog box where you can download or create
server connection paths, or connect to predefined paths.

Now click the **Add Server** button to create a new path, which we will construct to connect to the waiting `pvserver` running on your local machine.
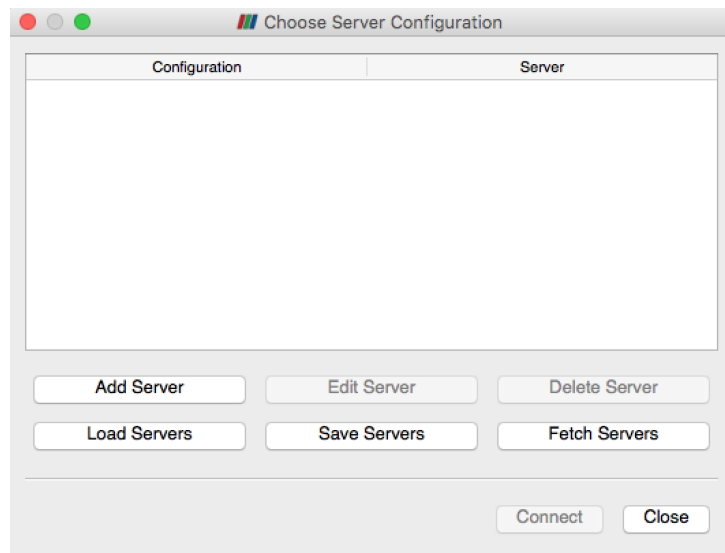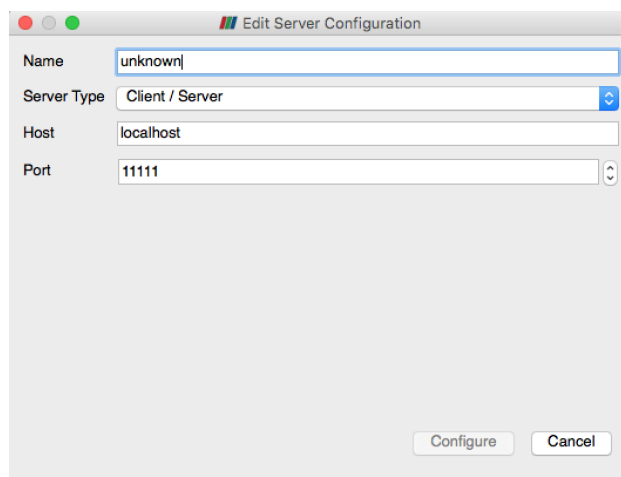

In the **Edit Server Configuration Dialog**, change the **Name** of the connection path from "unknown" to "my computer". In actual use one would enter in a nickname for and the IP address of the remote machine we want to connect to. Now hit the **Configure** button.

On the resulting **Edit Server Launch Configuration** dialog, leave the **Startup Type** as **Manual** since we have already started a waiting `pvserver`. In actual

use one would typically use **Command** and in the dialog enter in a script that starts up the server on the remote machine. Click **Save** on the dialog to finish defining the connection.
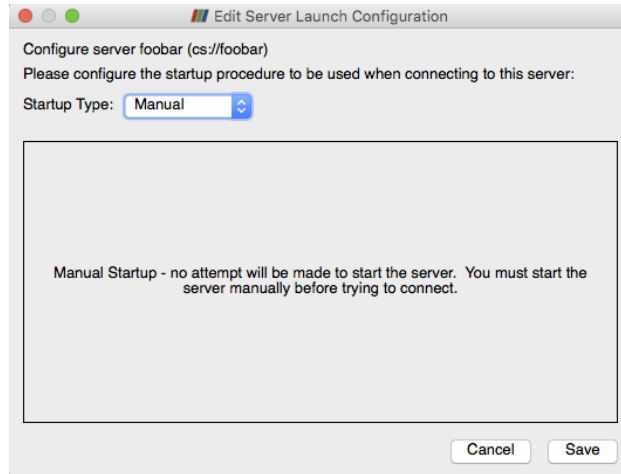
Now that we have defined a connection, it will show up under the **Choose Server Configuration** dialog whenever we hit ▤. Click on the name of the connection, "my computer" in this case, and click **Connect** to establish the connection between the GUI and `pvserver`.

Once connected, simply **File** → **Open** files on the remote system and work with them as before. To complete the exercise, open or create a data set, and choose the **vtkProcessId** array to examine how it is partioned across the distributed memory of the server.

<div align="right">♦</div>

In practice, the syntax of starting up a parallel server on a remote machine and connecting to it will be more involved. Typically the steps include ssh'ing to a remote machine's login node, submitting a job that runs a script that requests some number of nodes and runs `pvserver` under MPI on them with `pvserver` made to connect back to the waiting client (instead of the reverse as above) over at least one ssh tunnel hops. System administrators and other adventurous folk can find details online here:

http://www.paraview.org/Wiki/Setting_up_a_ParaView_Server#Running_the_Server

http://www.paraview.org/Wiki/Reverse_connection_and_port_forwarding

On the local machine you may need helper utiliites as well. XQuartz (for X11) on Mac and Putty (for ssh) on Windows are useful for starting up remote connections. These are required to complete the next exercise.

Once a connection is established the steps should be saved in scripts and configuration files. System administrators can publish these so that authorized users can simply download and use them. Several of these configuration files are hosted on the ParaView website where the ParaView client can download them from.

Exercise 4.3: Fetching and using Connections

Once again, click on to bring up the Choose Server Configuration dialog. This time, hit the Fetch Servers button. This will bring up the Fetch Server Configurations dialog box populated with a list of server definitions from the webite. Choose one and click the Import Selected button to download it into your ParaView preferences. The new connection is now available just as the connection to "my computer" is.

To use the remote machine, one more click . This time choose the new machine instead of "my machine" and click Connect. Doing so will bring up a Connection Options for ... dialog box that lets where you enter in the parameters for your parallel session.
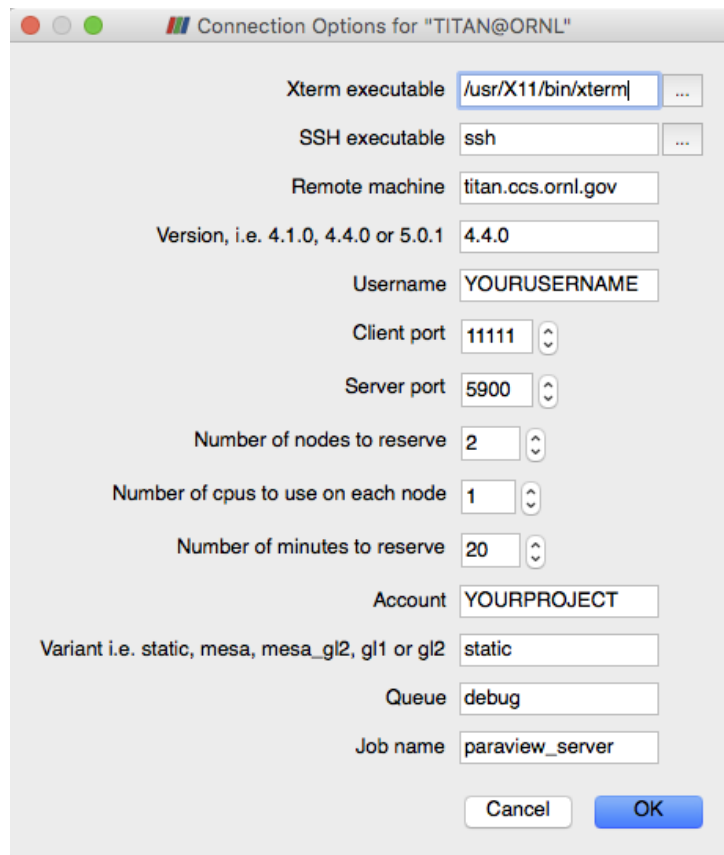
The parameters you will want to set include your username on the remote machine, the number of nodes and processes to reserve and the amount of time you want to reserve them for. Each site may have additional choices that are passed into the launch scripts on the server. Once set, click OK to try to establish a connection.

When you click OK ParaView typically spawns a terminal window (xterm on Mac and Linux, cmd.exe on Windows). Here you will have to enter in your credentials to actually access the remote machine. Once logged on, the script to reserve the nodes automatically runs and in most cases the terminal session has a menu with options for looking at the remote system's queue. When your job reaches the top of the queue, the ParaView 3D View window will return and you can begin doing remote visualization, this time with the capacity of the remote machine at your fingertips.

♦

4.7   Parallel Data Processing Practicalities

4.7.1   Keeping Track of Memory

When working with very large models, it is important to keep track of memory usage on the computer. One of the most common and frustrating problems encountered with large models is running out of memory. This in turn will lead to thrashing in the virtual memory system or an outright program fault.

Sections 4.8.2 and 4.8.3 provide suggestions to reduce your memory usage. Even so, it is wise to keep an eye on the memory available in your system. ParaView provides a tool called the **memory inspector** designed to do just that.
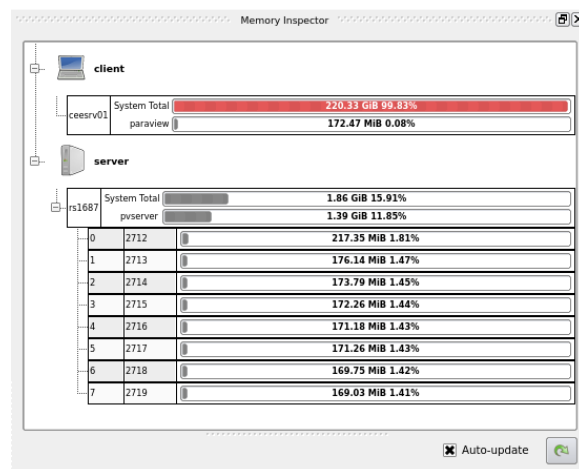
To access the memory inspector, select in the menu bar View → Memory Inspector. The memory inspector provides information for both the client you are running on and any server you might be connected to. It will tell you the total amount of memory used on the system and the amount of memory ParaView is using. For servers containing multiple nodes, information both for the conglomerate job and for each individual node are given. Note that a memory issue in any single node can cause a problem for the entire ParaView job.

4.7.2   Ghost Levels

Unfortunately, blindly running visualization algorithms on partitions of cells does not always result in the correct answer. As a simple example, consider the **external faces** algorithm. The external faces algorithm finds all cell faces that belong to only one cell, thereby identifying the boundaries of the mesh. What happens when we run external faces independently on our partitions?
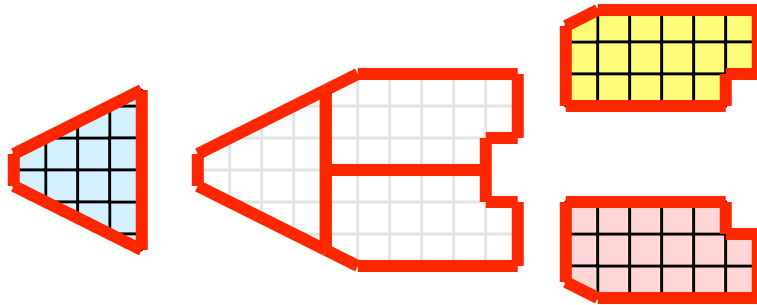
Oops. We see that when all the processes ran the external faces algorithm independently, many internal faces were incorrectly identified as being external. This happens where a cell in one partition has a neighbor in another partition. A process has no access to cells in other partitions, so there is no way of knowing that these neighboring cells exist.

The solution employed by ParaView and other parallel visualization systems is to use **ghost cells** (sometimes also called **halo regions**). Ghost cells are cells that are held in one process but actually belong to another. To use ghost cells, we first have to identify all the neighboring cells in each partition. We then copy these neighboring cells to the partition and mark them as ghost cells, as indicated with the gray colored cells in the following example.
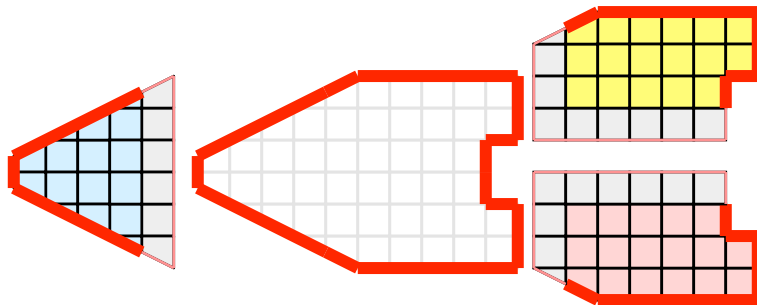
When we run the external faces algorithm with the ghost cells, we see that we are still incorrectly identifying some internal faces as external. However, all of these misclassified faces are on ghost cells, and the faces inherit the ghost status of the cell it came from. ParaView then strips off the ghost faces and we are left with the correct answer.

In this example we have shown one layer of ghost cells: only those cells that are direct neighbors of the partition's cells. ParaView also has the ability to retrieve multiple layers of ghost cells, where each layer contains the neighbors of the previous layer not already contained in a lower ghost layer or the original data itself. This is useful when we have cascading filters that each require their own layer of ghost cells. They each request an additional layer of ghost cells from upstream, and then remove a layer from the data before sending it downstream.

4.7.3   Data Partitioning

For the most part, ParaView leaves the task of breaking up or partitioning the data to the simulation code that produced the data. It is then the responsibility of ParaView's specific reader module for the file format in question to work efficiently, typically reading different files from different nodes independently but simultaneously. In ideal cases the rest of the parallel pipeline also operates independently and simultaneously.

Still, in the general case, since we are breaking up and distributing our data, it is prudent to address the ramifications of how we partition the data. The data shown in Section 4.1 has a **spatially coherent** partitioning. That is, all the cells of each partition are located in a compact region of space. There are other ways to partition data. For example, you could have a random partitioning.
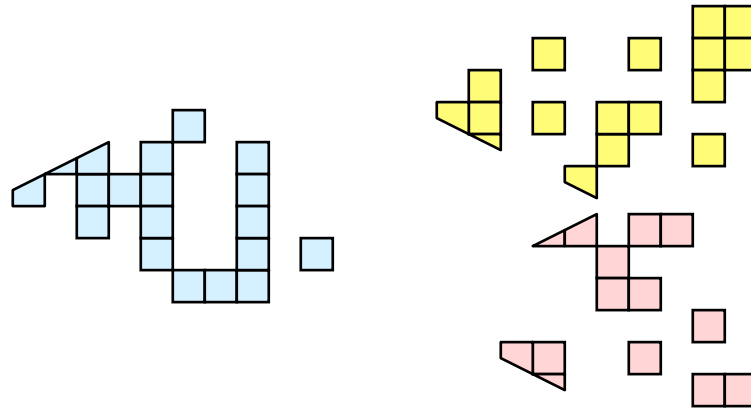
Random partitioning has some nice features. It is easy to create and is friendly to load balancing. However, a serious problem exists with respect to ghost cells.
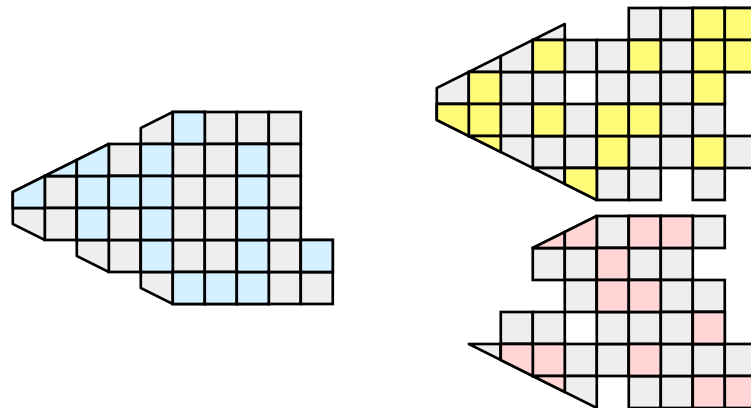
In this example, we see that a single level of ghost cells nearly replicates the entire data set on all processes. We have thus removed any advantage we had with parallel processing. Because ghost cells are used so frequently, random partitioning is not used in ParaView.

4.7.4   D3 Filter

The previous sections described the importance of load balancing and ghost levels for parallel visualization. This section describes how to achieve that.

Load balancing and ghost cells are handled automatically by ParaView when you are reading structured data (image data, rectilinear grid, and structured grid). The implicit topology makes it easy to break the data into spatially coherent chunks and identify where neighboring cells are located.

It is an entirely different matter when you are reading in unstructured data (poly data and unstructured grid). There is no implicit topology and no neighborhood information available. ParaView is at the mercy of how the data was written to disk. Thus, when you read in unstructured data there is no guarantee about how well load balanced your data will be. It is also unlikely that the data will have ghost cells available, which means that the output of some filters may be incorrect.

Fortunately, ParaView has a filter that will both balance your unstructured data and create ghost cells. This filter is called D3, which is short for distributed data decomposition. Using D3 is easy; simply attach the filter (located in Filters → Alphabetical → D3) to whatever data you wish to repartition.
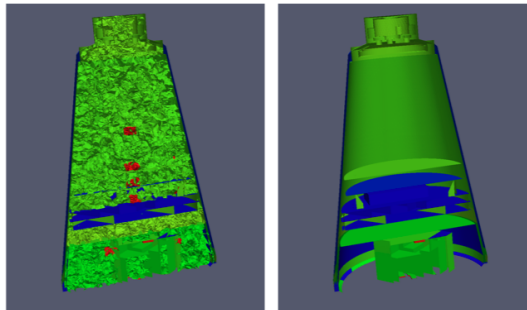
The most common use case for D3 is to attach it directly to your unstructured grid reader. Regardless of how well load balanced the incoming data might be, it is important to be able to retrieve ghost cells so that subsequent filters will generate the correct data. The example above shows a cutaway of the extract surface filter on an unstructured grid. On the left we see that there are many faces improperly extracted because we are missing ghost cells. On the right the problem is fixed by first using the D3 filter.

4.7.5   Ghost Cells Generator Filter

In many cases a better alternative to D3 filter is the Ghost Level Generator. This filter is more efficient than the D3 filter because it makes the assumption

that the input unstructured grid data is already partitioned into spatially coherent regions. This is generally a safe assumption as simulations benefit from coherency too. Because of this, it concerns itself only with cells at and near the external shell of the mesh, and does not consider or transfer the bulk of the data at the interior.

4.8   Advice

4.8.1   Matching Job Size to Data Size

How many cores should I have in my ParaView server? This is a common question with many important ramifications. It is also an enormously difficult question. The answer depends on a wide variety of factors including what hardware each processor has, how much data is being processed, what type of data is being processed, what type of visualization operations are being done, and your own patience.

Consequently, we have no hard answer. We do however have several rules of thumb.

If you are loading structured data (image data, rectilinear grid, structured grid), try to have a minimum of one core per 20 million cells. If you can spare the cores, one core for every 5 to 10 million cells is usually plenty.

If you are loading unstructured data (poly data, unstructured grid), try to have a minimum of one core per 1 million cells. If you can spare the cores, one core for every 250 to 500 thousand cells is usually plenty.

As stated before, these are just rules of thumb, not absolutes. You should always try to experiment to gage what your core to data size should be. And, of course, there will always be times when the data you want to load will stretch the limit of the resources you have available. When this happens, you will want to make sure that you avoid data explosion and that you cull your data quickly.

4.8.2   Avoiding Data Explosion

The pipeline model that ParaView presents is very convenient for exploratory visualization. The loose coupling between components provides a very flexible framework for building unique visualizations, and the pipeline structure allows you to tweak parameters quickly and easily.

The downside of this coupling is that it can have a larger memory footprint. Each stage of this pipeline maintains its own copy of the data. Whenever possible, ParaView performs **shallow copies** of the data so that different stages of the pipeline point to the same block of data in memory. However, any filter that creates new data or changes the values or topology of the data must allocate new memory for the result. If ParaView is filtering a very large mesh, inappropriate use of filters can quickly deplete all available memory. Therefore, when visualizing large data sets, it is important to understand the memory requirements of filters.

Please keep in mind that the following advice is intended *only for when dealing with very large amounts of data and the remaining available memory is low*. When you are not in danger of running out of memory, ignore all of the following advice.

When dealing with structured data, it is absolutely important to know what filters will change the data to unstructured. Unstructured data has a much higher memory footprint, per cell, than structured data because the topology must be explicitly written out. There are many filters in ParaView that will change the topology in some way, and these filters will write out the data as an unstructured grid, because that is the only data set that will handle any type of topology that is generated. The following list of filters will write out a new unstructured topology in its output that is roughly equivalent to the input. These filters should *never* be used with structured data and should be used with caution on unstructured data.

- Append Datasets
- Append Geometry
- Clean
- Clean to Grid
- Connectivity
- D3
- Delaunay 2D/3D
- Extract Edges
- Linear Extrusion
- Loop Subdivision

- Reflect
- Rotational Extrusion
- Shrink
- Smooth
- Subdivide
- Tessellate
- Tetrahedralize
- Triangle Strips
- Triangulate

Technically, the Ribbon and Tube filters should fall into this list. However, as they only work on 1D cells in poly data, the input data is usually small and of little concern.

This similar set of filters also output unstructured grids, but they also tend to reduce some of this data. Be aware though that this data reduction is often smaller than the overhead of converting to unstructured data. Also note that the reduction is often not well balanced. It is possible (often likely) that a single process may not lose any cells. Thus, these filters should be used with caution on unstructured data and extreme caution on structured data.

- Clip
- Decimate
- Extract Cells by Region

- Extract Selection
- Quadric Clustering
- Threshold

Similar to the items in the preceding list, Extract Subset performs data reduction on a structured data set, but also outputs a structured data set. So the warning about creating new data still applies, but you do not have to worry about converting to an unstructured grid.

This next set of filters also outputs unstructured data, but it also performs a reduction on the dimension of the data (for example 3D to 2D), which results in a much smaller output. Thus, these filters are usually safe to use with unstructured data and require only mild caution with structured data.

- Cell Centers
- Contour
- Extract CTH Parts
- Extract Surface
- Feature Edges

- Mask Points
- Outline (curvilinear)
- Slice
- Stream Tracer

These filters do not change the connectivity of the data at all. Instead, they only add field arrays to the data. All the existing data is shallow copied. These filters are usually safe to use on all data.

- Block Scalars
- Calculator 🖩
- Cell Data to Point Data
- Curvature
- Elevation
- Generate Surface Normals
- Gradient
- Level Scalars
- Median
- Mesh Quality
- Octree Depth Limit
- Octree Depth Scalars

- Point Data to Cell Data
- Process Id Scalars
- Python Calculator
- Random Vectors
- Resample with dataset
- Surface Flow
- Surface Vectors
- Texture Map to...
- Transform
- Warp (scalar)
- Warp (vector) 🍃

This final set of filters are those that either add no data to the output (all data of consequence is shallow copied) or the data they add is generally independent of the size of the input. These are almost always safe to add under any circumstances (although they may take a lot of time).

- Annotate Time
- Append Attributes
- Extract Block
- Extract Level
- Glyph
- Group Datasets
- Histogram
- Integrate Variables
- Normal Glyphs

- Outline
- Outline Corners
- Plot Global Variables Over Time
- Plot Over Line
- Plot Selection Over Time
- Probe Location
- Temporal Shift Scale
- Temporal Snap-to-Time-Steps
- Temporal Statistics

There are a few special case filters that do not fit well into any of the pre-
vious classes. Some of the filters, currently Temporal Interpolator and Particle
Tracer, perform calculations based on how data changes over time. Thus,
these filters may need to load data for two or more instances of time, which
can double or more the amount of data needed in memory. The Temporal
Cache filter will also hold data for multiple instances of time. Also keep in
mind that some of the temporal filters such as the temporal statistics and
the filters that plot over time may need to iteratively load all data from disk.
Thus, it may take an impractically long amount of time even though it does
not require any extra memory.

The Programmable Filter ⟨...⟩ is also a special case that is impossible to
classify. Since this filter does whatever it is programmed to do, it can fall
into any one of these categories.

4.8.3   Culling Data

When dealing with large data, it is clearly best to cull out data whenever
possible, and the earlier the better. Most large data starts as 3D geometry
and the desired geometry is often a surface. As surfaces usually have a much
smaller memory footprint than the volumes that they are derived from, it is
best to convert to a surface soon. Once you do that, you can apply other
filters in relative safety.

A very common visualization operation is to extract isosurfaces from a
volume using the Contour 🔵 filter. The Contour filter usually outputs geom-
etry much smaller than its input. Thus, the Contour filter should be applied
early if it is to be used at all. Be careful when setting up the parameters
to the Contour filter because it still is possible for it to generate a lot of
data. This obviously can happen if you specify many isosurface values. High
frequencies such as noise around an isosurface value can also cause a large,
irregular surface to form.

Another way to peer inside of a volume is to perform a Slice 🧊 on it.
The Slice 🧊 filter will intersect a volume with a plane and allow you to see
the data in the volume where the plane intersects. If you know the relative
location of an interesting feature in your large data set, slicing is a good way
to view it.

If you have little *a-priori* knowledge of your data and would like to explore
the data without paying the memory and processing time for the full data
set, you can use the Extract Subset 🧊 filter to subsample the data. The

subsampled data can be dramatically smaller than the original data and should still be well load balanced. Of course, be aware that you may miss small features if the subsampling steps over them and that once you find a feature you should go back and visualize it with the full data set.

There are also several features that can pull out a subset of a volume: Clip , Threshold , Extract Selection, and Extract Subset can all extract cells based on some criterion. Be aware, however, that the extracted cells are almost never well balanced; expect some processes to have no cells removed. Also, all of these filters with the exception of Extract Subset will convert structured data types to unstructured grids. Therefore, they should not be used unless the extracted cells are of at least an order of magnitude less than the source data.

When possible, replace the use of a filter that extracts 3D data with one that will extract 2D surfaces. For example, if you are interested in a plane through the data, use the Slice filter rather than the Clip filter. If you are interested in knowing the location of a region of cells containing a particular range of values, consider using the Contour filter to generate surfaces at the ends of the range rather than extract all of the cells with the Threshold filter. Be aware that substituting filters can have an effect on downstream filters. For example, running the Histogram filter after Threshold will have an entirely different effect than running it after the roughly equivalent Contour filter.

4.8.4   Downsamplng

Downsampling is also fequently helpful for very large data. For example, the Extract Subset filter reduce the size of Structured DataSets by simpling taking strided samples along the i, j and k axes.

For unstructured grids the Quadric Clustering filter downsamples unstructured data into a smaller dataset with similar appearance by averaging vertices within cells of a grid. ParaView injects this algorithm into the display pipeline while you interact with the camera as described in the next section.

In some cases though, it can be more effective to work with structured data than unstructured data. In particular structured data volume rendering algorithms are usually much faster. In general structured data representations are very compact, adding very little overhead to the raw data values. Unstructured data types require 12 bytes of memory storage for every every vertex and at least 8 bytes of storage to define each cell beyond the normal

storage for cell and point aligned values. Structured types use a total of 36 bytes to represent the entire set of vertices and cells in comparison.

Fortunately you can convert from unstructured to structured easily with the Resample To Image filter. When the sizes of the cells in the input do not very widely, and can thus be approximated well by constant sized voxels, this can be very effective. This filter internally makes use of the DIY2 block-parallel communication and computation library to communicate and transfer data among the parallel nodes.
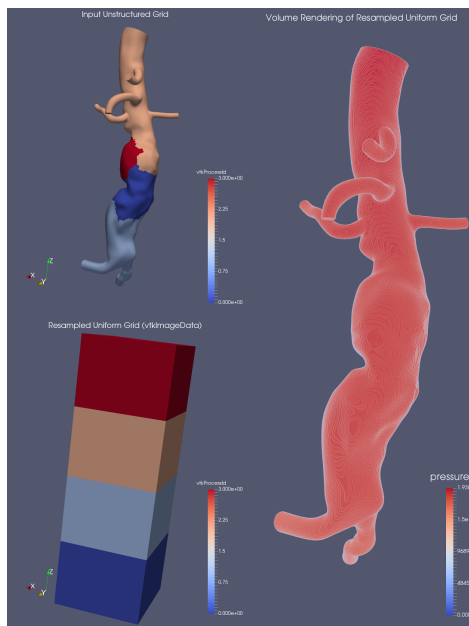
There is a companion filter Resample With DataSet which takes in a source and an input object and resamples values from the source onto the input, which does not have to be a regular grid. This also uses DIY2. This is used to assigning new values onto a particular shaped object.

4.9   Parallel Rendering Details

Rendering is the process of synthesizing the images that you see based on your data. The ability to effectively interact with your data depends highly on the speed of the rendering. Thanks to advances in 3D hardware acceleration, fueled by the computer gaming market, we have the ability to render 3D

quickly even on moderately priced computers. But, of course, the speed of rendering is proportional to the amount of data being rendered. As data gets bigger, the rendering process naturally gets slower.

To ensure that your visualization session remains interactive, ParaView supports two modes of rendering that are automatically flipped as necessary. In the first mode, **still render**, the data is rendered at the highest level of detail. This rendering mode ensures that all of the data is represented accurately. In the second mode, **interactive render**, speed takes precedence over accuracy. This rendering mode endeavors to provide a quick rendering rate regardless of data size.

While you are interacting with a 3D view, for example rotating, panning, or zooming with the mouse, ParaView uses an interactive render. This is because during the interaction a high frame rate is necessary to make these features usable and because each frame is immediately replaced with a new rendering while the interaction is occurring so that fine details are less important during this mode. At any time when interaction of the 3D view is not taking place, ParaView uses a still render so that the full detail of the data is available as you study it. As you drag your mouse in a 3D view to move the data, you may see an approximate rendering while you are moving the mouse, but the full detail will be presented as soon as you release the mouse button.

The interactive render is a compromise between speed and accuracy. As such, many of the rendering parameters concern when and how lower levels of detail are used.

4.9.1   Basic Rendering Settings

Some of the most important rendering options are the LOD parameters. During interactive rendering, the geometry may be replaced with a lower **level of detail** (**LOD**), an approximate geometry with fewer polygons.
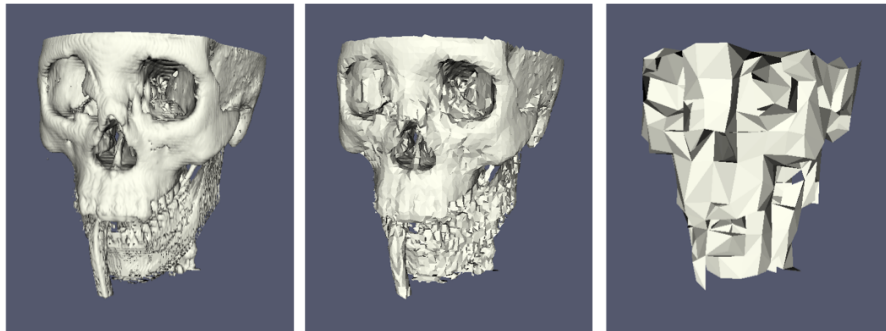
The resolution of the geometric approximation can be controlled. In the proceeding images, the left image is the full resolution; the middle image is the default decimation for interactive rendering, and the right image is ParaView's maximum decimation setting.

The 3D rendering parameters are located in the settings dialog box which is accessed in the menu from Edit → Settings (ParaView → Preferences on the Mac). The rendering options in the dialog are in the Render View tab.
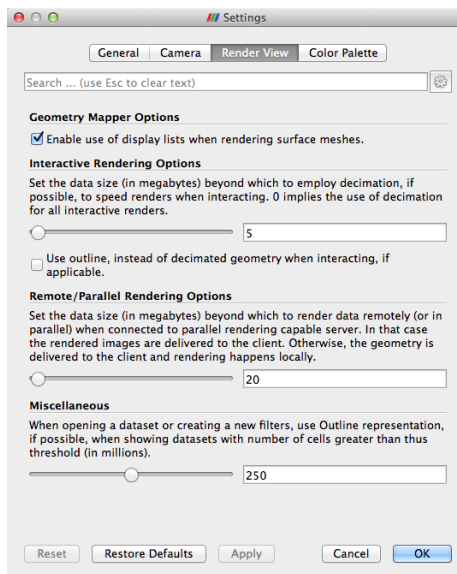
The options pertaining to the geometric decimation for interactive rendering are located in a section labeled Interactive Rendering Options. Some of these options are considered advanced, so to access them you have to either toggle on the advanced options with the ⚙ button or search for the option

using the edit box at the top of the dialog. The interactive rendering options include the following.

- Set the data size at which to use a decimated geometry in interactive rendering. If the geometry size is under this threshold, ParaView always renders the full geometry. Increase this value if you have a decent graphics card that can handle larger data. Try decreasing this value if your interactive renders are too slow.

- Set the factor that controls how large the decimated geometry should be. This control is set to a value between 0 and 1. 0 produces a very small number of triangles but possibly with a lot of distortion. 1 produces more detailed surfaces but with larger geometry. ⚙

- Add a delay between an interactive render and a still render. ParaView usually performs a still render immediately after an interactive motion is finished (for example, releasing the mouse button after a rotation). This option can add a delay that can give you time to start a second interaction before the still render starts, which is helpful if the still render takes a long time to complete. ⚙

- Use an outline in place of decimated geometry. The outline is an alternative for when the geometry decimation takes too long or still produces too much geometry. However, it is more difficult to interact with just an outline.

ParaView contains many more rendering settings. Here is a summary of some other settings that can effect the rendering performance regardless of whether ParaView is run in client-server mode or not. These options are spread among several categories, and several are considered advanced.

Geometry Mapper Options

- Enable or disable the use of display lists. Display lists are internal structures built by graphics systems. They can potentially speed up rendering but can also take up memory.

Translucent Rendering Options

- Enable or disable depth peeling. Depth peeling is a technique
  ParaView uses to properly render translucent surfaces. With it,
  the top surface is rendered and then "peeled away" so that the
  next lower surface can be rendered and so on. If you find that
  making surfaces transparent really slows things down or renders
  completely incorrectly, then your graphics hardware may not be
  implementing the depth peeling extensions well; try shutting off
  depth peeling. ⚙

- Set the maximum number of peels to use with depth peeling. Us-
  ing more peels allows more depth complexity but allowing less
  peels runs faster. You can try adjusting this parameter if translu-
  cent geometry renders too slow or translucent images do not look
 correct. ⚙

Miscellaneous

- When creating very large datasets, default to the outline repre-
  sentation. Surface representations usually require ParaView to
  extract geometry of the surface, which takes time and memory.
  For data with size above this threshold, use the outline represen-
  tation, which has very little overhead, by default instead.

- Show or hide annotation providing rendering performance infor-
  mation. This information is handy when diagnosing performance
  problems. ⚙

Note that this is not a complete list of ParaView rendering settings. We
have left out settings that do not significantly effect rendering performance.
We have also left out settings that are only valid for parallel client-server
rendering, which are discussed in Section 4.9.3.

4.9.2   Image Level of Detail

The overhead incurred by the parallel rendering algorithms is proportional
to the size of the images being generated. Also, images generated on a server
must be transfered to the client, a cost that is also proportional to the image
size. To help increase the frame rate during interaction, ParaView introduces
a new LOD parameter that controls the size of the images.

During interaction while parallel rendering, ParaView can optionally subsample the image. That is, ParaView will reduce the resolution of the image in each dimension by a factor during interaction. Reduced images will be rendered, composited, and transfered. On the client, the image is inflated to the size of the available space in the GUI.
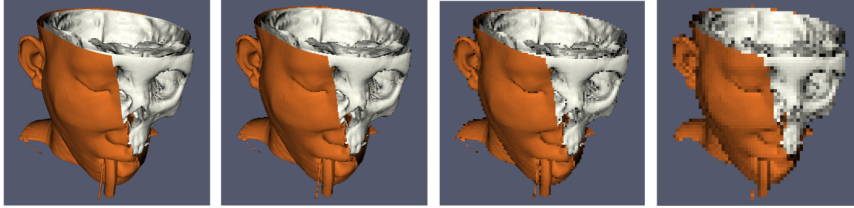

The resolution of the reduced images is controlled by the factor with which the dimensions are divided. In the preceeding images, the left image has the full resolution. The following images were rendered with the resolution reduced by a factor of 2, 4, and 8, respectively.

ParaView also has the ability to compress images before transferring them from server to client. Compression, of course, reduces the amount of data transferred and therefore makes the most of the available bandwidth. However, the time it takes to compress and decompress the images adds to the latency.

ParaView contains three different image compression algorithms for client-server rendering. The oldest is a custom algorithm called **Squirt**, which stands for Sequential Unified Image Run Transfer. Squirt is a run-length encoding compression that reduces color depth to increase run lengths. The second algorithm uses the **Zlib** compression library, which implements a variation of the Lempel-Ziv algorithm. Zlib typically provides better compression than Squirt, but takes longer to perform and hence adds to the latency. The most recent addition is the **LZ4** algorithm which is tuned for fast compression and decompression.

4.9.3   Parallel Render Parameters
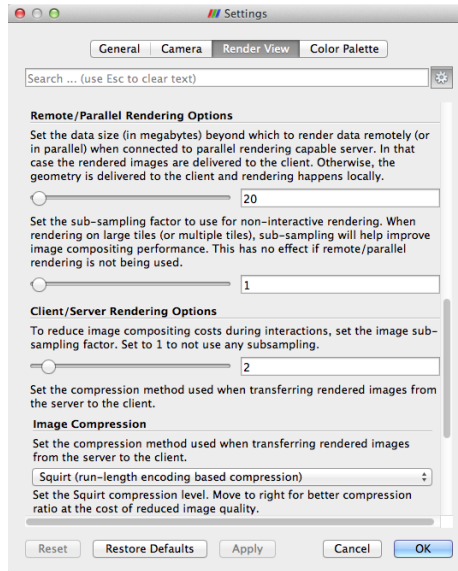
Like the other 3D rendering parameters, the parallel rendering parameters are located in the settings dialog box, which is accessed in the menu from Edit → Settings (ParaView → Preferences on the Mac). The parallel rendering options in the dialog are in the Render View tab (intermixed with several other rendering options such as those described in Section 4.9.1). The parallel and client-server options are divided among several categories, and several are considered advanced.

Remote/Parallel Rendering Options

- Set the data size at which to render remotely in parallel or to render locally. If the geometry is over this threshold (and ParaView is connected to a remote server), the data is rendered in parallel remotely and images are sent back to the client. If the geometry is under this threshold, the geometry is sent back to the client and images are rendered locally on the client.

- Set the sub-sampling factor for still (non-interactive) rendering. Some large displays have more resolution than is really necessary, so this sub-sampling reduces the resolution of all images displayed.

Client/Server Rendering Options

- Set the interactive subsampling factor. The overhead of parallel rendering is proportional to the size of the images generated. Thus, you can speed up interactive rendering by specifying an image subsampling rate. When this box is checked, interactive renders will create smaller images, which are then magnified when displayed. This parameter is only used during interactive renders. ⚙

Image Compression

- Before images are shipped from server to client, they optionally can be compressed using one of the three compression algorithms: Squirt, Zlib or LZ4. To make the compression more effective, each algorithm has one or more tunable parameters that let you customize the behavior and target level of compression. ⚙

- Suggested image compression presets are provided for several common network types. When attempting to select the best image compression options, try starting with the presets that best match your connection. ⚙

4.9.4   Parameters for Large Data

The default rendering parameters are suitable for most users. However, when dealing with very large data, it can help to tweak the rendering parameters. The optimal parameters depend on your data and the hardware ParaView is running on, but here are several pieces of advice that you should follow.

- Try turning off display lists. Turning this option off will prevent the graphics system from building special rendering structures. If you have graphics hardware, these rendering structures are important for feeding the GPUs fast enough. However, if you do not have GPUs, these rendering structures do not help much.

- If there is a long pause before the first interactive render of a particular data set, it might be the creation of the decimated geometry. Try using an outline instead of decimated geometry for interaction. You could

also try lowering the factor of the decimation to 0 to create smaller geometry.

- Avoid shipping large geometry back to the client. The remote rendering will use the power of entire server to render and ship images to the client. If remote rendering is off, geometry is shipped back to the client. When you have large data, it is always faster to ship images than to ship data (although if your network has a high latency, this could become problematic for interactive frame rates).

- Adjust the interactive image sub-sampling for client-server rendering as needed. If image compositing is slow, if the connection between client and server has low bandwidth, or if you are rendering very large images, then a higher subsample rate can greatly improve your interactive rendering performance.

- Make sure Image Compression is on. It has a tremendous effect on desktop delivery performance, and the artifacts it introduces, which are only there during interactive rendering, are minimal. Lower bandwidth connections can try using Zlib instead of Squirt compression. Zlib will create smaller images at the cost of longer compression/decompression times.

- If the network connection has a high latency, adjust the parameters to avoid remote rendering during interaction. In this case, you can try turning up the remote rendering threshold a bit, and this is a place where using the outline for interactive rendering is effective.

- If the still (non-interactive) render is slow, try turning on the delay between interactive and still rendering to avoid unnecessary renders.

4.10   Catalyst

For small scale data, running the ParaView GUI in serial mode is likely acceptable. For large scale data, interactive parallel visualization is quite useful. For very large scale data, batch mode parallel processing is more effective because of practical concerns like batch system queue wait time

before a parallel job is launched and system policies that limit the size and duration of interactive jobs.

For extreme scale visualization even batch mode is beginning to be impractical though as comparatively slow disks limit the size of data that can be saved off by the simulation at runtime. In this domain ParaView's Catalyst configuration is recommended as an *in situ* analysis and visualization library.

Catalyst is a visualization framework that packages either portions or the entirety of ParaView's parallel server framework so that it can be linked into and called from arbitrary simulation codes. Slow IO systems are largely bypassed via an adaptor mechanism whereby the simulation code's data structures are translated or ideally reused directly by ParaView while still in RAM.

With Catalyst, full resolution simulation products stay in memory instead of being saved to and later loaded back from disk. As the simulation progresses, it periodically calls into ParaView. In this way it is possible to immediately generate smaller derived data sets, images, plots, etc. that would otherwise be generated during post processing after a much longer, human in the loop, delay.
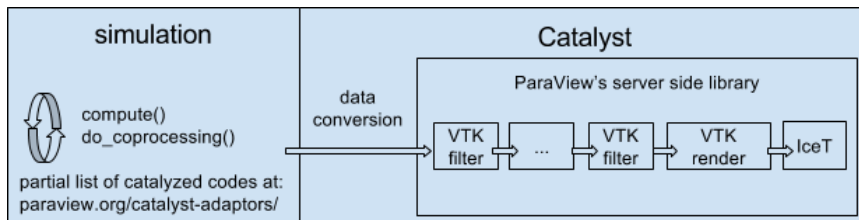
As explained more fully in the ParaView Catalyst User's Guide there are two components to making use of Catalyst. The first is to Catalyze the simulation. This is where a software developer compiles Catalyst into the simulation, adds a handful of (typically three) function calls to it, and fleshes out an adaptor template with working code that produces VTK data sets from the simulation's own internal data structures.

Once the simulation has been Catalyzed, day to day users need to define what ParaView should do with the data it is given. Catalyst can produce data extracts, images, statistical quantities, etc. Any of these can be made with Python or even lower level Fortran, C or C++ coding to maximize runtime efficiency, but it is more flexible and convenient to do so with recorded Catalyst scripts which are demonstrated in the next exercise.

With Catalyst scripts the simulation input deck references a Python file
that defines the visualization pipeline. The script can do nearly everything
that ParaView batch scripts do and is generally created with help from the
ParaView GUI. To create the script one loads a representative data set,
creates a pipeline, designates the set of inputs and outputs to and output
from the pipeline, and then simply saves out the Python file. Catalyst scripts
are very similar to recorded Python traces.

The user interface for defining the inputs and outputs is found inside
the the Catalyst Script Generator plugin. Our final exercise demonstrates
with a small example.

Exercise 4.4: Catalyst

We demonstrate a Catalyst workflow using the ParaView binaries and the
PythonFullExample code from the ParaView source tree. PythonFullExam-
ple is a toy simulation that uses numpy and mpi4py to update a regular grid
as time evolves. The example consists of four files. Open and read these to
begin.

The computational kernel is in fedatastructures.py. Inspection of the code
shows that it has no particular dependency on ParaView, and simply makes
up a structured grid in parallel with numpy and mpi4py.

From fedatastructures.py:

```
 def Update(self, time):
     self.Velocity = numpy.zeros((self.Grid.GetNumberOfPoints(), 3))
     self.Velocity = self.Velocity + time
     self.Pressure = numpy.zeros(self.Grid.GetNumberOfCells())
```

The main loop is found in the fedriver.py. If the doCoprocessing variable
is false, this falls back to a simple loop which runs the simulation to updates
the in-memory array over time. When the variable is true the Catalyst library
is exercised via three function calls: initialize(), coprocess() and finalize().

From fedriver.py:

```
import numpy
import sys
from mpi4py import MPI

comm = MPI.COMM_WORLD
```

```
rank = comm.Get_rank()

import fedatastructures

grid = fedatastructures.GridClass([10, 12, 10], [2, 2, 2])
attributes = fedatastructures.AttributesClass(grid)
doCoprocessing = True

if doCoprocessing:
    import coprocessor
    coprocessor.initialize()
    coprocessor.addscript("cpscript.py")

for i in range(100):
    attributes.Update(i)
    if doCoprocessing:
        import coprocessor
        coprocessor.coprocess(i, i, grid, attributes)

if doCoprocessing:
    import coprocessor
    coprocessor.finalize()
```

fedriver.py and fedatastructures.py represent the simulation code. The
Catalyst parts consist of the general purpose adaptor code, found within in
the coprocessor.py, and the pipeline definition, found in cpscript.py. In co-
processor.py note in particular where the coprocess function populates the
dataDescription data structure to give fedatastructure's content over to Para-
View. The coprocessor also takes in, with the addscript() call the pipeline
definition file.

From coprocessing.py:

```
if coProcessor.RequestDataDescription(dataDescription):
    import fedatastructures
    imageData = vtk.vtkImageData()
    imageData.SetExtent(\
      grid.XStartPoint, grid.XEndPoint, \
      0, grid.NumberOfYPoints-1, \
      0, grid.NumberOfZPoints-1)
```

```
    imageData.SetSpacing(grid.Spacing)

    velocity = paraview.numpy_support.numpy_to_vtk(attributes.Velocity)
    velocity.SetName("velocity")
    imageData.GetPointData().AddArray(velocity)

    pressure = numpy_support.numpy_to_vtk(attributes.Pressure)
    pressure.SetName("pressure")
    imageData.GetCellData().AddArray(pressure)
```

The starting script for this example is called cpscript.py. This simply
saves out the input it is given in a format that the ParaView GUI can readily
understand. The internals of the PipelineClass is essentially a ParaView
batch script that takes as input not a file but the datadescription class from
the adaptor.

From cpscript.py:

```
 class Pipeline:
   filename_6_pvti = \
       coprocessor.CreateProducer( datadescription, "input" )

   # create a new 'Parallel ImageData Writer'
   imageDataWriter1 = \
       servermanager.writers.XMLPImageDataWriter(Input=filename_6_pvti)

   # register the writer with coprocessor
   # and provide it with information such as the filename to use,
   # how frequently to write the data, etc.
   coprocessor.RegisterWriter(imageDataWriter1, \
       filename="fullgrid_%t.pvti", freq=100)

   Slice1 = Slice( \
       Input=filename_6_pvti, guiName="Slice1", \
       Crinkleslice=0, SliceOffsetValues=[0.0], \
       Triangulatetheslice=1, SliceType="Plane" )
   Slice1.SliceType.Offset = 0.0
   Slice1.SliceType.Origin = [9.0, 11.0, 9.0]
   Slice1.SliceType.Normal = [1.0, 0.0, 0.0]
```

```
    # create a new 'Parallel PolyData Writer'
    parallelPolyDataWriter1 = \
        servermanager.writers.XMLPPolyDataWriter(Input=Slice1)

    # register the writer with coprocessor
    # and provide it with information such as the filename to use,
    # how frequently to write the data, etc.
    coprocessor.RegisterWriter(\
        parallelPolyDataWriter1, filename='slice_%t.pvtp', freq=10)

 return Pipeline()
```

Now let us run the simulation code to produce some data.

- On Mac:

```
/Applications/ParaView-5.6.app/Contents/MacOS/mpiexec \
  -np 2 \
  /Applications/ParaView-5.6.app/Contents/bin/pvbatch  --symmetric \
  fedriver.py
```

- On Linux:

```
/usr/local/lib/paraview-5.6/mpiexec -np 2 \
  /usr/local/bin/pvbatch --symmetric \
  fedriver.py
```

- On Windows:

```
mpiexec -np 2 ^
  "C:/Program Files/ParaView 5.6/bin/pvbatch" --symmetric ^
  fedriver.py
```

The example will run and produce several files, among them fullgrid_0.pvti which is a raw dump of the simulation's full output. Let us use that to customize Catalyst's output.

First open the file in the ParaView GUI and inspect it. You will find that this is a very simple structured grid with two variables, **pressure** and **velocity**. Now insert the File → Extract Subset filter into the pipeline.

Now load the Catalyst Script Generator Plugin to get access to the GUI for defining new Catalyst scripts. Navigate to Tools → Manage Plugins...,

click on the CatalystScriptGeneratorPlugin, click Load Selected, and click Close. The only difference you will see in the GUI immediately is that there are new Writers and CoProcessing menus.

The Writers menu allows you to insert file output filters into the pipeline for Catalyst. As we will see in a moment, the ParaView GUI is able to generate Catalyst scripts by capturing the state of the visualization constructed in ParaView. However, this state does not include writing data, which is done through the action of File → Save Data. Thus, the Writers allow you to insert stubs in the visualization pipeline that indicate Catalyst should record that result. It also lets you set an independent simulation output filename and frequency for each writer.

Now use the Writers menu and insert a writer into the pipeline after the Extract Subset filter.

The CoProcessing menu is where you go to save the pipeline. It also lets you designate which rendered views Catalyst should export and their output frequency because like before, Catalyst does not otherwise have File → Save Screenshot ....

Now that we have a pipeline, Click on CoProcessing → Export State to set up the Catalyst script.

The first pane is informational. Hit Continue to go on to the next pane.

The second pane lets us choose any of the available pipeline roots (readers or sources) as the one that is representative of the the data that the simulation will produce. Double click on fullgrid_0.pvti and again click Continue.

The third pane lets us customize the name of the input within the coprocessing script. Simply click Continue to go on as input is appropriate in this, and in fact most cases with non-complex adaptors.

The fourth pane lets us choose which views should be exported, how often, and a variety of other aspects of the coprocessing run. Click on Output rendering components i.e. views to have the script save off images and click Done. Next choose a name for the Catalyst script. Choose cpscript.py to overwrite the original script with your new script, or choose a new name and edit fedriver.py to call it instead of cpscript.py.

Finally quit ParaView and run the simulation again. The files produced will now include png files for screen captures at each time step, and vti files corresponding to the output of the Extract Subset filter at each timestep. Simply repeat the process, adding filters and views for example, to change the extracts that Catalyst will produce in the simulation.

♦

Note this is but one interface to using Catalyst. If simulation users are uncomfortable with ParaView and/or Python, it is entirely possible for simulation developers to instrument the simulation code with predefined and parameterized pipelines with or without resorting to Python. It is also possible to create minimized Catalyst editions that consist of just a small portion of the larger ParaView code base.

As is the case with ParaView itself, Catalyst is evolving rapidly. It already has more advanced offshoots than described here. One, is a live data capability in which one can connect a ParaView client to a running Catalyzed simulation in order to check in on its progress from time to time and do a limited amount of computational steering and debugging. Another is Cinema, which is an image based framework for deferred visualization of automatically generated simulation results organized in a database. For information on these we refer the reader to the ParaView mailing list and websites.

Chapter 5

Further Reading

Thank you for participating in this tutorial. Hopefully you have learned enough to get you started visualizing large data with ParaView. Here are some sources for further reading.

The documentation page on ParaView's web site contains a list of resources available for further learning and questions.

http://www.paraview.org/documentation

The ParaView Guide is a good resource to have with ParaView. It provides many other instructions and more detailed descriptions on many features. The ParaView guide can be accessed from the ParaView documentation page.

The ParaView Wiki is full of information that you can use to help you set up and use ParaView.

http://www.paraview.org/Wiki/ParaView

In particular, those of you who wish to install a parallel ParaView server should consult the appropriate build and install pages.

http://www.paraview.org/Wiki/Setting_up_a_ParaView_Server

If you are interested in learning more about visualization or more specifics about the filters available in ParaView, consider picking up the following visualization textbook.

Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit.* Kitware, Inc., fourth edition, 2006. ISBN 1-930934-19-X.

If you plan on customizing ParaView, the previous books and web pages have lots of information. For more information about using VTK, the underlying visualization library, and Qt, the GUI library, consider the following books have more information.

Kitware Inc. *The VTK User's Guide.* Kitware, Inc., 2006.

Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4.* Prentice Hall, 2006. ISBN 0-13-187249-4.

If you are interested about the design of parallel visualization and other features of the VTK pipeline, there are several technical papers available.

Kenneth Moreland. "A Survey of Visualization Pipelines." *IEEE Transactions on Visualization and Computer Graphics*, 19(3), March 2013. DOI 10.1109/TVCG.2012.133.

James Ahrens, Charles Law, Will Schroeder, Ken Martin, and Michael Papka. "A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets." Technical Report #LAUR-00-1620, Los Alamos National Laboratory, 2000.

James Ahrens, Kristi Brislawn, Ken Martin, Berk Geveci, C. Charles Law, and Michael Papka. "Large-Scale Data Visualization Using Parallel Data Streaming." *IEEE Computer Graphics and Applications*, 21(4): 34–41, July/August 2001.

Andy Cedilnik, Berk Geveci, Kenneth Moreland, James Ahrens, and Jean Farve. "Remote Large Data Visualization in the ParaView Framework." *Eurographics Parallel Graphics and Visualization 2006*, pg. 163–170, May 2006.

James P. Ahrens, Nehal Desai, Patrick S. McCormic, Ken Martin, and Jonathan Woodring. "A Modular, Extensible Visualization System Architecture for Culled, Prioritized Data Streaming." *Visualization and Data Analysis 2007, Proceedings of SPIE-IS&T Electronic Imaging*, pg 64950I1-1–12, January 2007.

John Biddiscombe, Berk Geveci, Ken Martin, Kenneth Moreland, and David Thompson. "Time Dependent Processing in a Parallel Pipeline Architecture." *IEEE Visualization 2007*. October 2007.

If you are interested in the algorithms and architecture for ParaView's parallel rendering, there are also many technical articles on this as well.

Kenneth Moreland, Brian Wylie, and Constantine Pavlakos. "Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays." *Proceedings of IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pg. 85–92, October 2001.

Kenneth Moreland and David Thompson. "From Cluster to Wall with VTK." *Proceddings of IEEE 2003 Symposium on Parallel and Large-Data Visualization and Graphics*, pg. 25–31, October 2003.

Kenneth Moreland, Lisa Avila, and Lee Ann Fisk. "Parallel Unstructured Volume Rendering in ParaView." *Visualization and Data Analysis 2007, Proceedings of SPIE-IS&T Electronic Imaging*, pg. 64950F-1-12, January 2007.

Acknowledgements

Index