Scientific data formats and visualization of large datasets

Compute Ontario Summer School, May 2013

Alex Razoumov
razoumov@sharcnet.ca

SHARCNET/UOIT

- copy of these slides in `http://razoumov.sharcnet.ca/paraview.pdf`
- data and sample C++, Fortran, Python codes in `http://razoumov.sharcnet.ca/visualization.tar.gz` (two directories inside: `code/` and `data/`)

Visualization in different fields

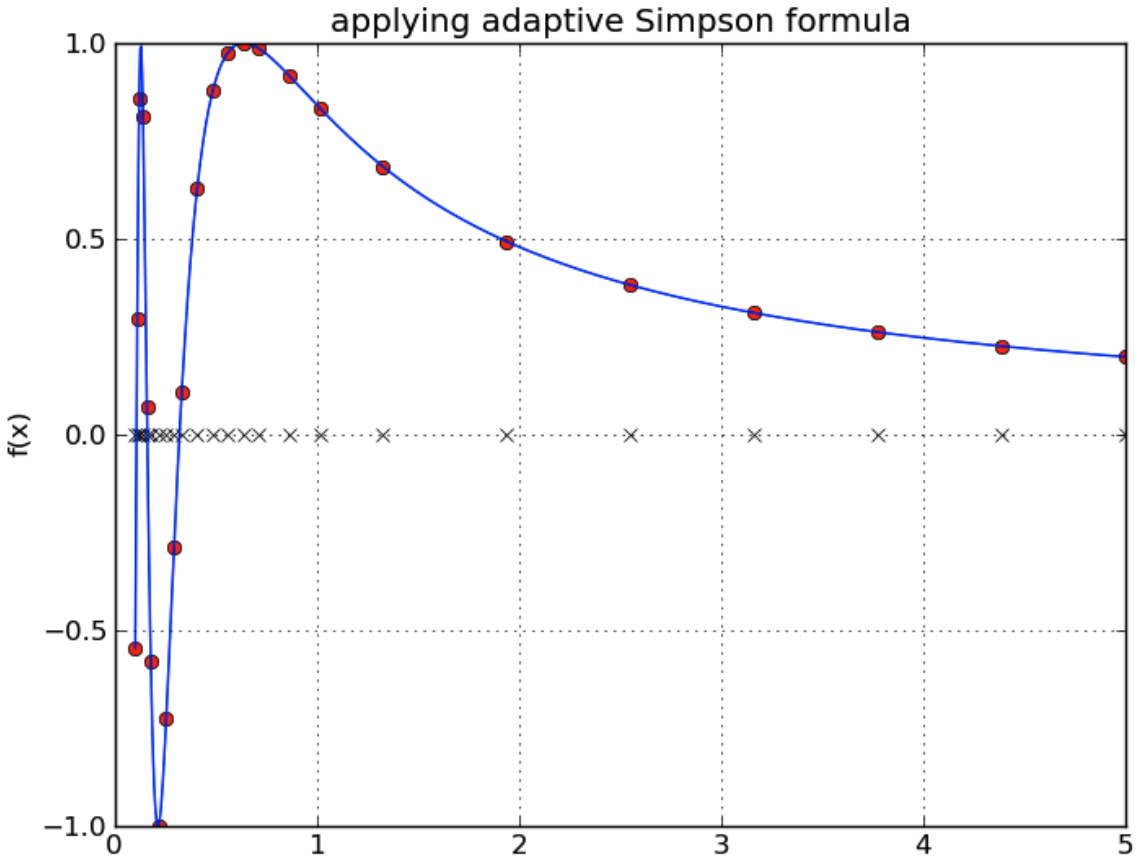
FIELD	VISUALIZATION TYPE
computational fluid dynamics	2D/3D flows, density, temperature, tracers
climate, meteorology, oceanography	fluid dynamics, clouds, chemistry, etc.
quantum chemistry	wave functions
molecular dynamics (phys, chem, bio)	particle/molecular data
bio-informatics	networks, trees, sequences
astrophysics	gravitational fields, 2D/3D fluids, $\leq$6D radiation field, magnetic fields, particle data
geographic information systems	digital elevation, rivers, etc.
medical imaging	MRI, CT scans, ultrasound
info-vis	abstract data

1D plotting vs. 2D/3D visualization

- **1D plotting**: plotting functions of one variable, 1D tabulated data
  - something as simple as gnuplot or pgplot
  - highly recommend: Python's Matplotlib library, other Python libraries
- **2D/3D visualization**: displaying multidimensional datasets, typically data on 2D/3D structured grids or on unstructured meshes (that have some topology in 2D/3D)
- Whatever you do, try to avoid proprietary tools, unless those tools provide a clear advantage (most likely not)
  - large $$
  - limitations on where you can run them, which machines/platforms, etc.
  - cannot get help from open-source community, user base usually smaller than for open-source tools
  - once you start accumulating scripts, you lock yourself into using these tools forever, and consequently paying $$ on a regular basis
  - there is nothing you cannot do with open-source tools
  - examples of closed proprietary tools: IDL, Matlab

Matplotlib example

Adaptive Simpson integration




```
from pylab import *
from adaptive import basicSimpson, partition, simpsonAdaptive
def f(x):
    return sin(1./x)

simpsonAdaptive(f,0.1,5.,1.e-5)
```

Adaptive Simpson integration (cont.)

- Driver script that calls `partition` (next page) and plots the function and the grid points used in integration

```
def simpsonAdaptive(f,a,b,maxError):
    global k, x, integralArray, errorArray
    k = 0
    x = zeros(1000)
    integralArray = zeros(1000)
    errorArray = zeros(1000)
    partition(f,a,b,maxError)
    x[k] = b
    print 'number of intervals =', k
    print 'total integral =', sum(integralArray[0:k+1])
    print 'total error <=', sum(errorArray[0:k+1])
    xmesh = x[0:k+1]
    ymesh = f(xmesh)
    plot(xmesh,ymesh,'ro')
    plot(xmesh,ymesh-ymesh,'kx')
    xx = linspace(a,b,1000)
    yy = f(xx)
    plot(xx,yy,'b-')
    return
```

Adaptive Simpson integration (cont.)

```
def partition(f,a,b,maxError):
    global k, x, integralArray, errorArray
    integral,error = basicSimpson(f,a,b)
    if abs(error) > maxError:
        midpoint = 0.5*(a+b)
        partition(f,a,midpoint,maxError)
        partition(f,midpoint,b,maxError)
    else:
        print 'interval =', a, b, ' --> ', integral, error
        x[k] = a
        integralArray[k] = integral
        errorArray[k] = error
        k += 1
    return
```

```
def basicSimpson(f,a,b):
    quarter = 0.25*(b-a)
    x1 = a
    x2 = x1 + quarter
    x3 = x2 + quarter
    x4 = x3 + quarter
    x5 = b
    f1 = f(x1); f2 = f(x2); f3 = f(x3); f4 = f(x4); f5 = f(x5)
    s = (b-a)*(f1+4.*f3+f5)/6.
    error = 4./45. * (b-a) * abs(f1+f5-4.*(f2+f4)+6.*f3)
    return s, error
```

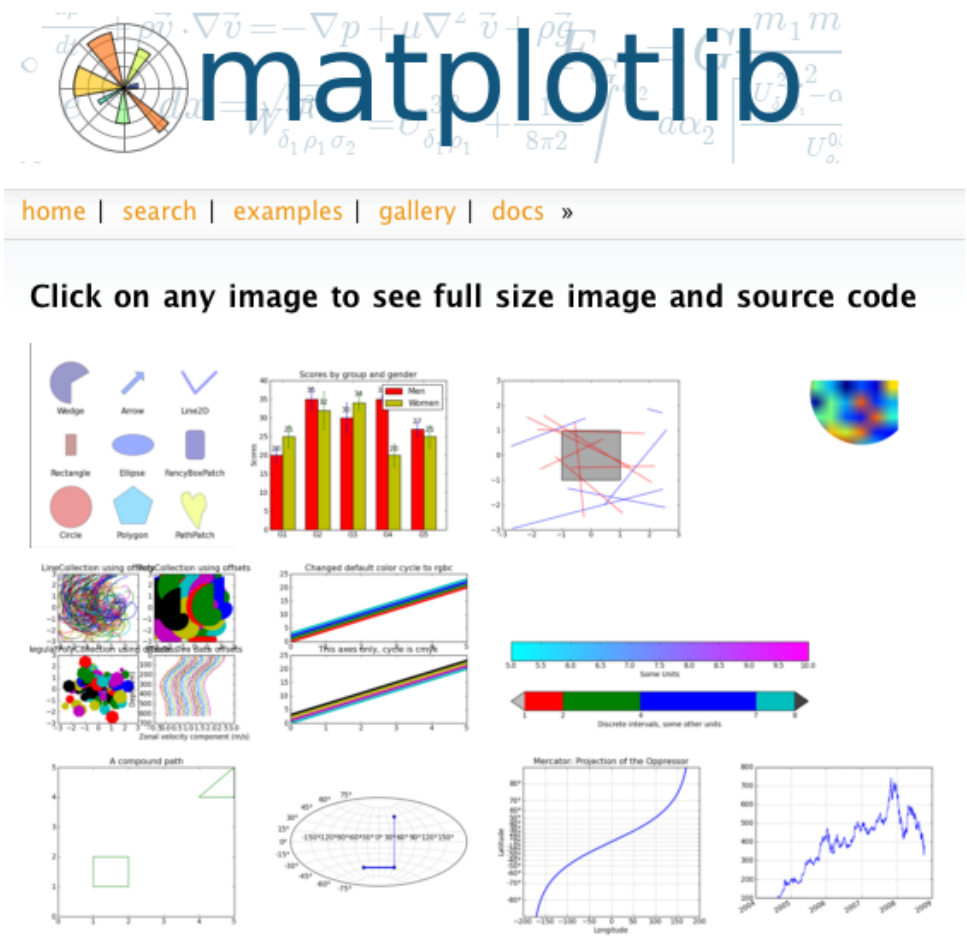
Python Imaging Library (PIL) example

Edge detection using numerical differentiation

Edge detection using numerical differentiation (cont.)

```
import Image as im
a = im.open("figures/fuji1.png")     # load image data from file
from pylab import *
nx = shape(a)[0]
ny = shape(a)[1]

c = a.load()
b_array = zeros((nx,ny),dtype=int)
for i in range(0,nx):
    for j in range(0,ny):
        b_array[i,j] = c[i,j][2]    # store blue pixels in a 2D matrix

gr = gradient(b_array)   # use only the blue band for edge detection
print shape(gr)
print shape(gr[0]), shape(gr[1])
gradientNormSquared = gr[0]**2 + gr[1]**2

for i in range(0,nx):
    for j in range(0,ny):
        value = int(gradientNormSquared[i,j])
        value = min(value,255)
        c[i,j] = (255-value,255-value,255-value)

a.show()     # display image
```

Matplotlib gallery contains hundreds of examples



`http://matplotlib.sourceforge.net/gallery.html`

click on any plot to get its source code

Other Python graphics and visualization libraries

- For more examples, see `http://python.org`, search for *"visualization"*
- Few other notable examples:

PACKAGE	DESCRIPTION
MayaVi2	scientific data 3D visualizer (Python + VTK)
yt	analysis and visualization toolkit for astrophysical simulations, focusing on AMR data from Enzo, Orion, FLASH, etc.
neuronvisio	GUI for NEURON simulator enviroment
VPython	3D graphics library
PyVisfile	storing data in a variety of scientific visualization file formats
PyVTK	tools for manipulating VTK files in Python
ScientificPython	various Python modules for scientific computing and visualization
chaco	interactive 2D plotting

2D/3D visualization packages

Visualization	
GNUPLOT	graphing utility
GRAPHVIZ	Represent structural information as diagrams of abstract graphs and networks
HYPERMESH	Altair Hyperworks Suite (commercial, hyperworks group, limited access)
ICEMCFD	ANSYS ICEM CFD Meshing Software (commercial, fluent group)
Paraview	Parallel Visualization Application
Scilab	Open Source Platform for Numerical Computation
VisIt	VisIT Visualization Tool
VMD	Visual Molecular Dynamics
XCrySDen	Crystalline and Molecular Structure Visualisation

`http://www.sharcnet.ca/my/software`

- Open-source, multi-platform, and general-purpose:
  - visualize scalar and vector fields
  - structured and unstructured meshes in 2D and 3D, particle data, polygonal data, irregular topologies
  - ability to handle very large datasets (GB-TB)
  - support for scripting, common data formats, parallel I/O (optional)
  - iteractive manipulation

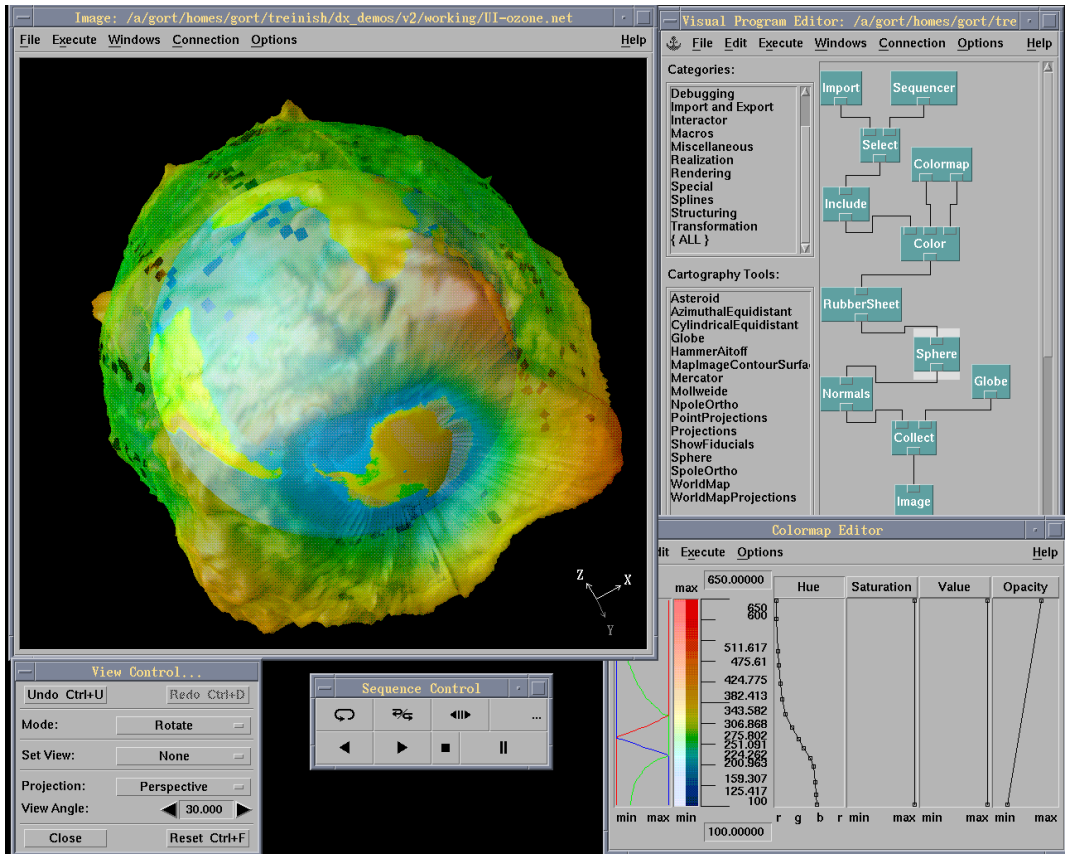
(1) OpenDX 4.4.4 – used to be installed on vizN-site (until few months ago)

(2) VisIT 2.2.2 – installed on vizN-site

(3) ParaView 3.6 - 3.8 (latest is 3.10.1) – installed on rainbow, vizN-site

OpenDX = Open Visualization Data Explorer

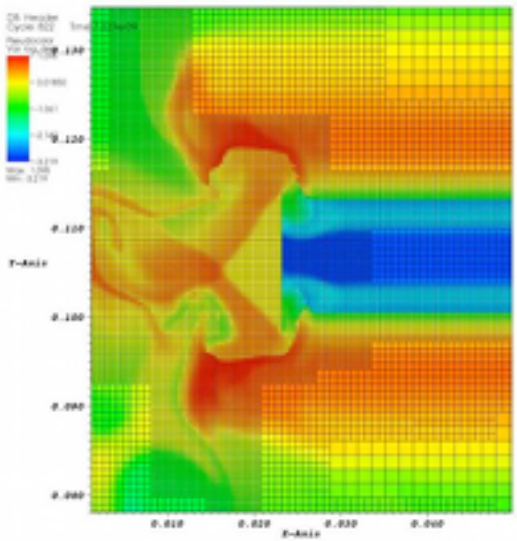
- `http://www.opendx.org`
- Started in 1991 at IBM, released as open source in 1999, latest v4.4.4 from 2006-Aug ...
- Windows/Linux binaries free, need to compile on Mac (available in darwinports/fink)
- Based on the Motif widget toolkit on top of X11
- Interactive Visual Program Editor

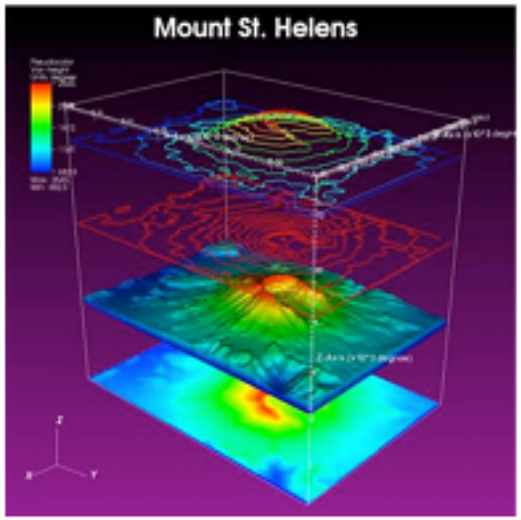


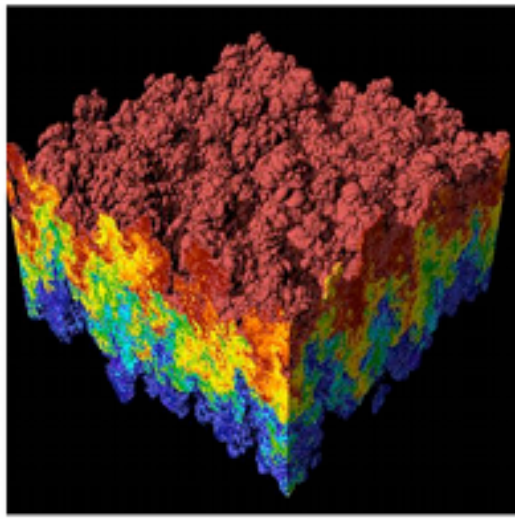
`http://www.research.ibm.com`

VisIT

- `http://wci.llnl.gov/codes/visit`
- Developed by the Department of Energy (DOE) Advanced Simulation and Computing Initiative (ASCI) to visualize results of terascale simulations
- v2.6.2 available as source and binary for Linux/Mac/Windows
- Over 60 visualization features (contour, mesh, slice, volume, molecule, ...)
- Reads over 60 different file formats
- Interfaces with C++, Python, and Java



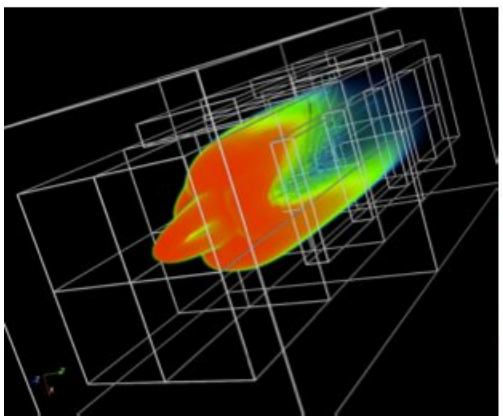
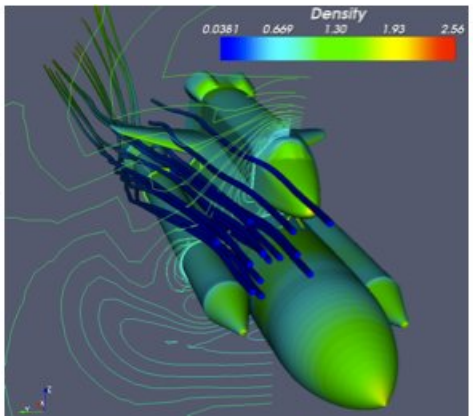


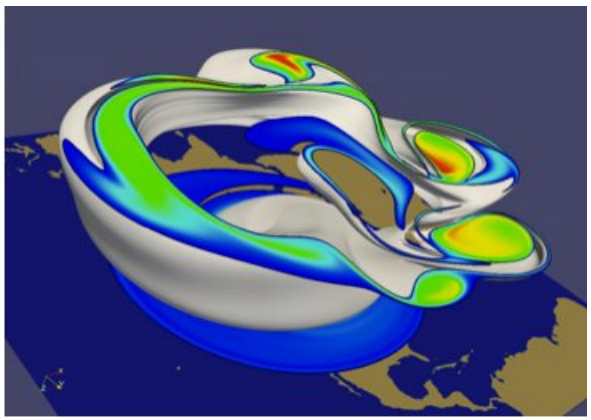


Lawrence Livermore National Laboratory

ParaView

- `http://www.paraview.org`
- Developed jointly by Sandia National Labs + Los Alamos National Lab + Kitware Inc.
- Latest binary release 3.98.1 (2013-Feb), available for Linux/Mac/Windows
- To visualize extremely large datasets on distributed memory machines
- Both interactive and Python scripting
- ParaView is based on VTK (Visualization Toolkit); not the only VTK-based open-source scientific visualizer, e.g. also see MayaVi (written in Python + numpy + scipy + VTK); note that VTK can be used from C++, Tcl, Java, Python as a standalone renderer



Why ParaView for this course?

- A lot of interest in ParaView among HPC users
- In my experience, ParaView is less buggy and more feature-rich than VisIT
- Wide binary availability, active development
- Tight integration with VTK (developed by the same folks)
- Support for over 130 input file formats
- Comes with many filters and plugins, including a Mobile Remote plugin to control ParaView from an iOS device (KiwiViewer)

- Not that I discourage you from using VisIT or other open-source packages

Online resources

ParaView on SHARCNET
`http://www.sharcnet.ca/my/software/show/67`
`https://www.sharcnet.ca/help/index.php/ParaView`

ParaView official documentation
`http://www.paraview.org/OnlineHelpCurrent`

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

ParaView/Python batch scripting
`http://www.paraview.org/Wiki/ParaView/Python_Scripting`

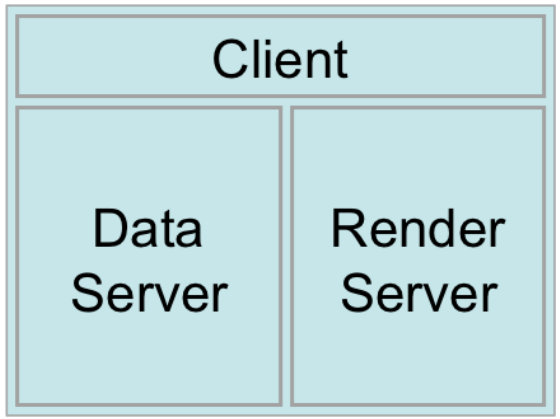
VTK tutorials `http://www.itk.org/Wiki/VTK/Tutorials`

ParaView architecture

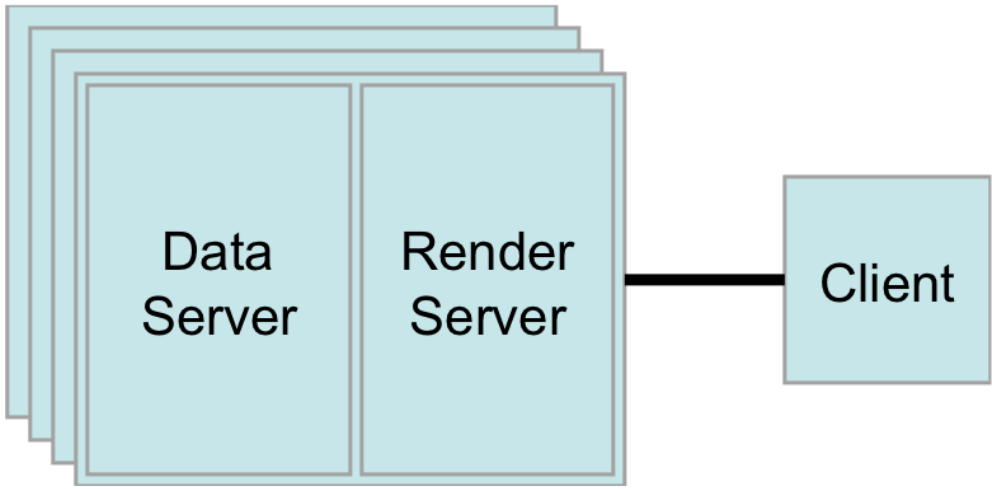
Three logical units of ParaView – these units can be embedded in the same application on the same computer, but can also run on different machines:

- **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 controls the object creation, execution, and destruction in the servers, but does not contain any of the data, 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.

Two major workflow models

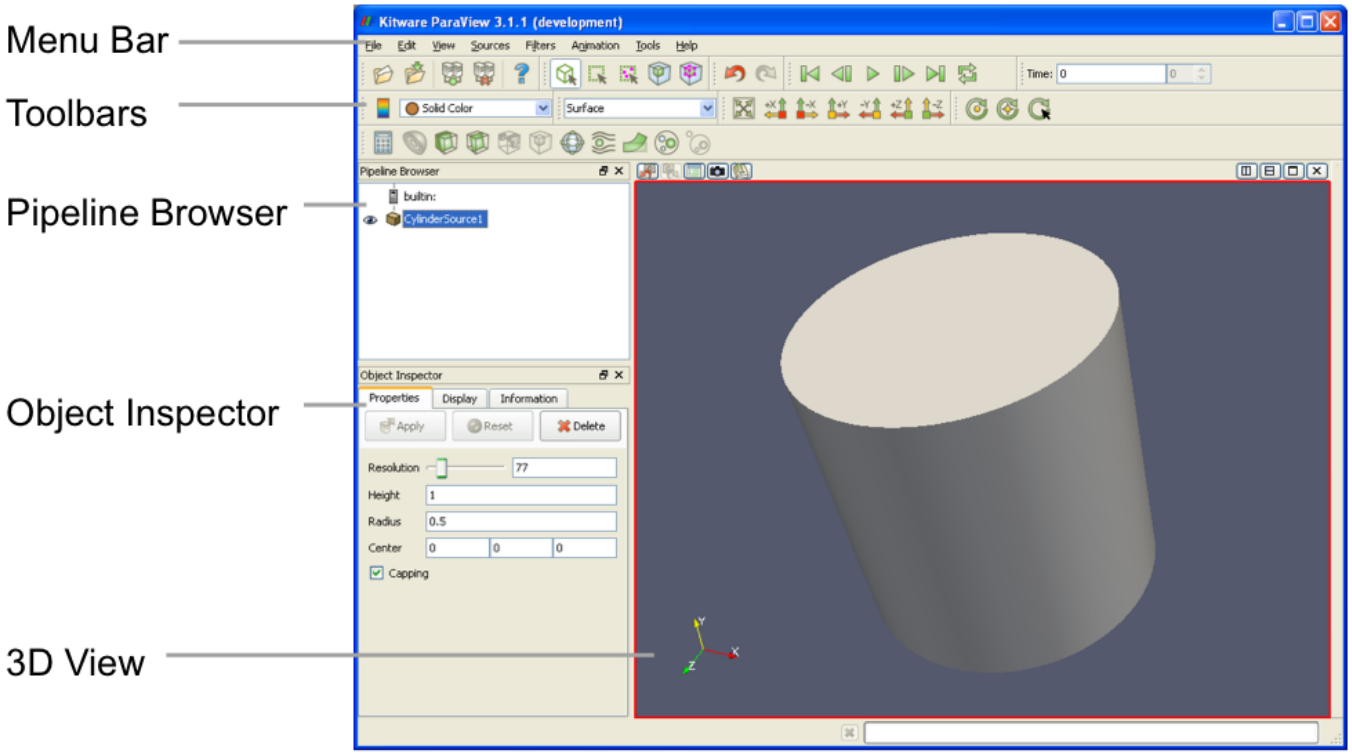


standalone mode



client-server mode: pvserver on a parallel machine

User interface

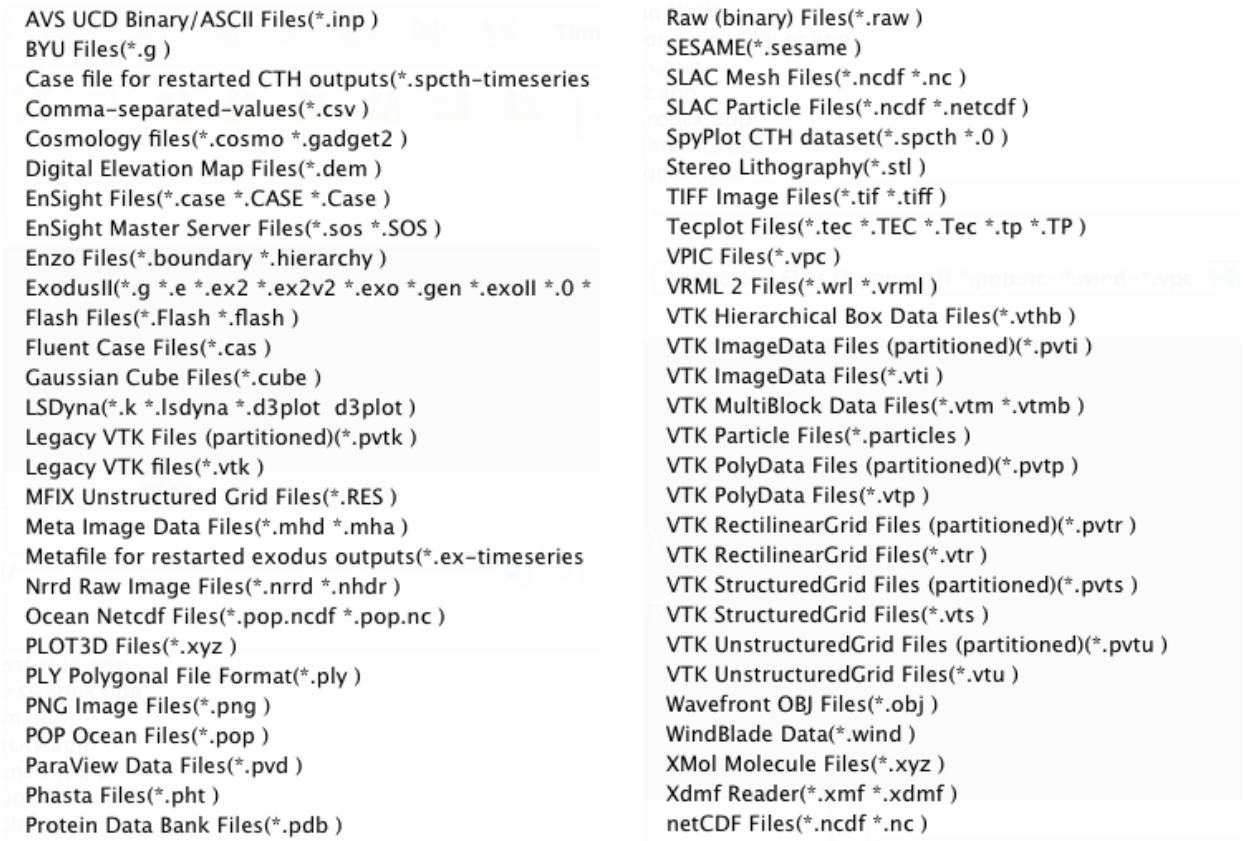


- Pipeline Browser: reading and filtering of data
- Object Inspector: view and change parameters of the current pipeline object (properties - display - information)
- View window

Find the following in the toolbar: “Connect”, “Disconnect”, “Toggle Color Legend Visibility”, “Edit Colour Map”, “Rescale to Data Range”

Data sources

- Generate data with a *Source* object
- Read data from a file



Somewhat incomplete list of file readers:

`http://paraview.org/OnlineHelpCurrent/ParaViewReaders.html`

Example: reading raw (binary) data

Show $f(x,y,z) = (1-z)\left[(1-y)\sin(\pi x) + y\sin(2\pi x)^2\right]$
$+z\left[(1-x)\sin(\pi y) + x\sin(2\pi y)^2\right]$ in $x, y, z \in [0,1]$ sampled at 16^3

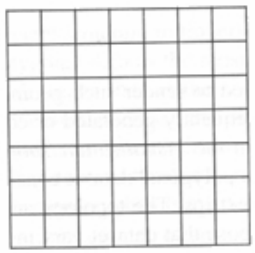
1. File: `data/raw/simpleData.raw` – load it as RAW BINARY
2. Describe the dataset in properties:
   - Data Scalar Type = float
   - Data Byte Order = Little Endian
   - File Dimensionality = 3
   - Data Extent: 1 to 16 in each dimension
   - Scalar Array Name = density
3. Try different views: outline, points, wireframe, ...
4. Depending on the view, can set:
   - Rescale to Data Range
   - Edit Color Map
5. Try saving data as paraview data type (*.pvd), deleting the object, and reading back from *.pvd – file now contains full description of dataset

VTK = Visualization Toolkit

- Open-source software system for 3D computer graphics, image processing and visualization
- Bindings to C++, Tcl, Java, Python
- ParaView is based on VTK $\Rightarrow$ supports all standard VTK file formats
- VTK file formats
  `http://www.vtk.org/VTK/img/file-formats.pdf`
  - legacy serial format (*.vtk): ASCII header lines + ASCII/binary data
  - XML formats (newer, much preferred, supports parallel file I/O, extension depends on data type): XML tags + ASCII/binary/compressed data

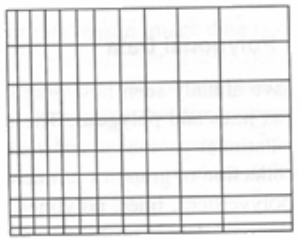
VTK 3D data: 6 major dataset (discretization) types

- **Image Data/Structured Points**: *.vti, points on a regular rectangular lattice, scalars or vectors at each point



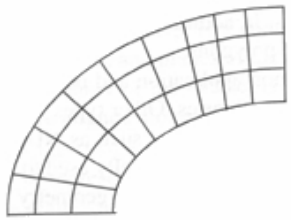
(a) Image Data

- **Rectilinear Grid**: *.vtr, same as Image Data, but spacing between points may vary, need to provide steps along the coordinate axes, not coordinates of each point



(b) Rectilinear Grid

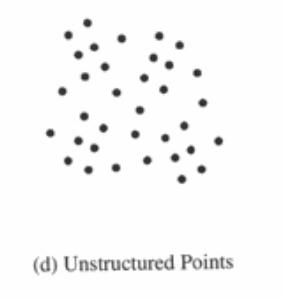
- **Structured Grid**: *.vts, regular topology and irregular geometry, need to indicate coordinates of each point

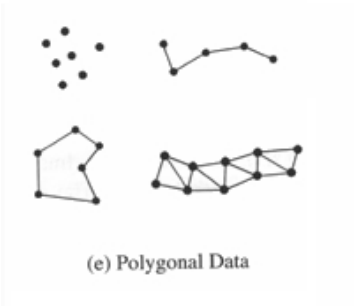


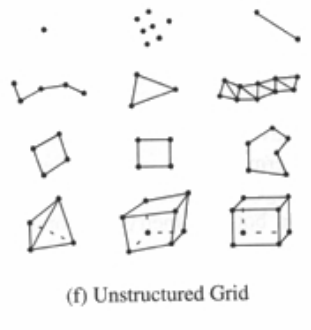
(c) Structured Grid

VTK 3D data: 6 major dataset (discretization) types

- **Particles/Unstructured Points**: *.particles
- **Polygonal Data**: *.vtp, unstructured topology and geometry, point coordinates, 2D cells only (i.e. no polyhedra), suited for maps
- **Unstructured Grid**: *.vtu, irregular in both topology and geometry, point coordinates, 2D/3D cells, suited for finite element analysis, structural design


(d) Unstructured Points


(e) Polygonal Data


(f) Unstructured Grid

VTK 3D data: dataset attributes

- Each VTK file can store a number of datasets, each with one of the following attributes
  - Scalars: single valued, e.g. density, temperature, pressure
  - Vectors: magnitude and direction, e.g. velocity
  - Normals: direction vectors ($|\mathbf{n}| = 1$) used for shading
  - LookupTable: each entry in the lookup table is a red-green-blue-alpha array (alpha is opacity: alpha=0 is transparent); if the file format is ASCII, the lookup table values must be float values in the range [0,1]
  - TextureCoordinates: used for texture mapping
  - Tensors: 3×3 real-valued symmetric tensors, e.g. stress tensor
  - FieldData: array of data arrays

Example: reading legacy VTK

1. File: `data/vtk/legacy/volume.vtk`
   - simple example (Structured Points): $3 \times 4 \times 6$ dataset, one scalar field, one vector field
2. File: `data/vtk/legacy/density.vtk`
   - another simple example (Structured Grid): $2 \times 2 \times 2$ dataset, one scalar field
3. File: `data/vtk/legacy/cube.vtk`
   - more complex example (Polygonal Data): cube represented by six polygonal faces. A single-component scalar, normals, and field data are defined on all six faces (CELL_DATA). There are scalar data associated with the eight vertices (POINT_DATA). A lookup table of eight colors, associated with the point scalars, is also defined.

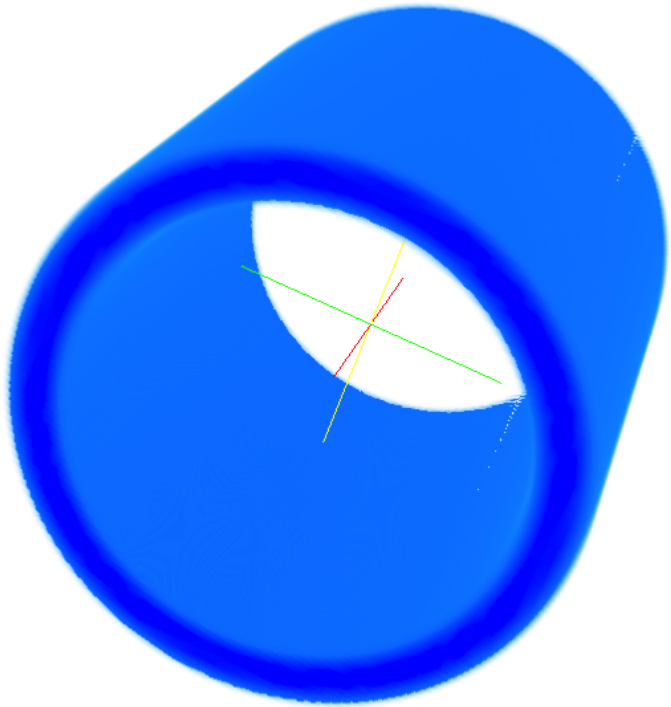
Exercise: visualizing 3D data with legacy VTK

- If you have your own 3D scalar data $\Rightarrow$ try writing it from C/C++/Fortran/etc as a VTK ASCII legacy file (*.vtk) in structured points format using *volume.vtk* as a template

- If not $\Rightarrow$ try visualizing a 3D "cylinder" function

$$f(x, y, z) = e^{-[(r-0.4)^2]^{0.5}}$$

where $r = \sqrt{(x-0.5)^2 + (y-0.5)^2}$,
or some other function of your choice,
inside the unit cube



- In either case, try sampling it at some moderate resolution, e.g. 30^3, since we are dealing with ASCII

Writing XML VTK from C++

- You can write directly in legacy VTK using VTK libraries for C/C++
- Here is an example: `code/SGrid.C` and `code/Makefile`, generates the file `data/vtk/xml/halfCylinder.vts`

  This example shows how to manually create a structured grid, set grid coordinates, fill the grid with a scalar and a vector, and write it in XML VTK to a *.vts file.
- To run it, you need an installed VTK library (either standalone or pulled from ParaView) – check `code/Makefile` to see which library files are needed

```
export LD_LIBRARY_PATH=/path/to/vtk/lib:$LD_LIBRARY_PATH
cd code
make SGrid
./SGrid
```

NetCDF and HDF5

- VTK is incredibly versatile format, can describe many different data types
- More often than not, in science one needs to simply store and visualize multidimensional arrays
- Problem: how do you store a 2000^3 array of real numbers (30GB of data)?
  - ASCII – forget about it
  - raw binary – possible, but many problems
  - VTK – probably an overkill, lacks some of the features below
- Scientific data formats come to rescue, two popular scientific data formats are NetCDF and HDF5
  - binary (of course!)
  - self-descriptive (include metadata)
  - portable (cross-platform): universal datatypes, bit order in a byte (little vs. big endian), etc.
  - support parallel I/O
  - optionally support compression

NetCDF and HDF5 support in ParaView

- NetCDF supported natively in ParaView (more about it in a minute)
- No native support for HDF5, however, ParaView supports a container format XDMF which uses HDF5 for actual data
- Also support for a number of file formats generated by third-party software that in turn use HDF5 underneath

XDMF = eXtensible Data Model and Format

- only briefly mention it, details at `http://www.xdmf.org`
- XDMF = XML for **light** data + HDF5 for **heavy** data
  - data type (float, integer, etc.), precision, rank, and dimensions completely described in the XML layer (as well as in HDF5)
  - the actual values in HDF5, potentially can be enormous
- single XML wrapper can reference multiple HDF5 files (e.g. written by each node on a cluster)
- don't need HDF5 libraries to perform simple operations
- C++ API is provided to read/write XDMF data
- also available from Python, Tcl, Java, Fortran through C++ calls
- in Fortran can generate XDMF files with HDF5 calls + plain text for the XML wrapper `http://www.xdmf.org/index.php/Write_from_Fortran`

NetCDF

- `code/writeNetCDF.c` (Fortran version `code/writeNetCDF.f90`) writes a 30^3 volume with a doughnut shape at the centre in NetCDF

C example

```
module load netcdf/intel/4.2
icc writeNetCDF.c -o writeNetCDF $CPPFLAGS $LDFLAGS -lnetcdf
./writeNetCDF
```

f90 example

```
module load netcdf/intel/4.2
ifort writeNetCDF.f90 -o writeNetCDF $CPPFLAGS $LDFLAGS -lnetcdff -lnetcdf
./writeNetCDF
```

Recap of input file formats

Now you know how to import data from your code:

- Raw binary data
- VTK legacy format (*.vtk) with ASCII data, looked at:
  - Structured Points
  - Structured Grid
  - Polygonal Data
- VTK XML formats from C++ writing binary data with VTK libraries, looked at:
  - Structured Grid (*.vts)
  - other formats can be written using the respective class, e.g. vtkPolyData, vtkRectilinearGrid, vtkStructuredGrid, vtkUnstructuredGrid
- HDF5 files via XDMF, **native NetCDF**

Filters

Many interesting features about a dataset cannot be determined by simply looking at its surface – a lot of useful information is on the inside, or can be extracted from a combination of variables

Volumetric view - available only for Structured Points (regularly spaced grid) among all VTK datasets.

Filters are functional units that process the data to generate, extract, or derive additional features. The filter connections form a **visualization pipeline**.

Over 80 filters are currently available.

Check out “Filters” in the menu; some are found in the toolbar.

Toolbar filters

- **Calculator** evaluates a user-defined expression on a per-point or per-cell basis.
- **Contour** extracts user-defined points, isocontours, or isosurfaces from a scalar field.
- **Clip** removes all 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 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 By 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 dataset.
- **Extract Level** extracts one or more items from a multi-block dataset.

Calculator

- Load one of the datasets, e.g. `data/other/disk_out_ref.ex2` (load temperature, velocity, pressure), and try to visualize individual variables: Pres, Temp, V
- In "Object Inspector" > "Display" use "Rescale to Data Range" and "Edit Color Map ..." to see the data range
- Now try to apply **Calculator** filter to display the following variables: Pres/Temp, log10(Temp), mag(V) - pay attention to the data range
- Dropdown menus "Scalars" and "Vectors" will help you enter variables
- "?" button is surprisingly useful
- You can change visibility of each object in the pipeline browser by clicking on the eyeball icon next to it

Contour

- Delete **Calculator** from the pipeline browser, load **Contour**
- Create an isosurface where the temperature is 400 K
- Try different views (Surface, Wireframe, ...)

Creating a visualization pipeline

You can apply one filter to the data generated by another filter

Delete all previous filters, start with the original data from `data/other/disk_out_ref.ex2`, or just press "Disconnect" and reload the data

1. apply **Clip** filter to the data: rotate, move the clipping plane, select variables to display, make sure there are data points inside the object (easy to see with points/wireframe, uncheck "Show Plane")
2. delete **Clip**, now apply Filters $\rightarrow$ Alphabetical $\rightarrow$ **Extract Surface**, and then add **Clip** to the result of **Extract Surface** $\Rightarrow$ the dataset is now hollow (use wireframe/surface)

Multiview: several variables side by side

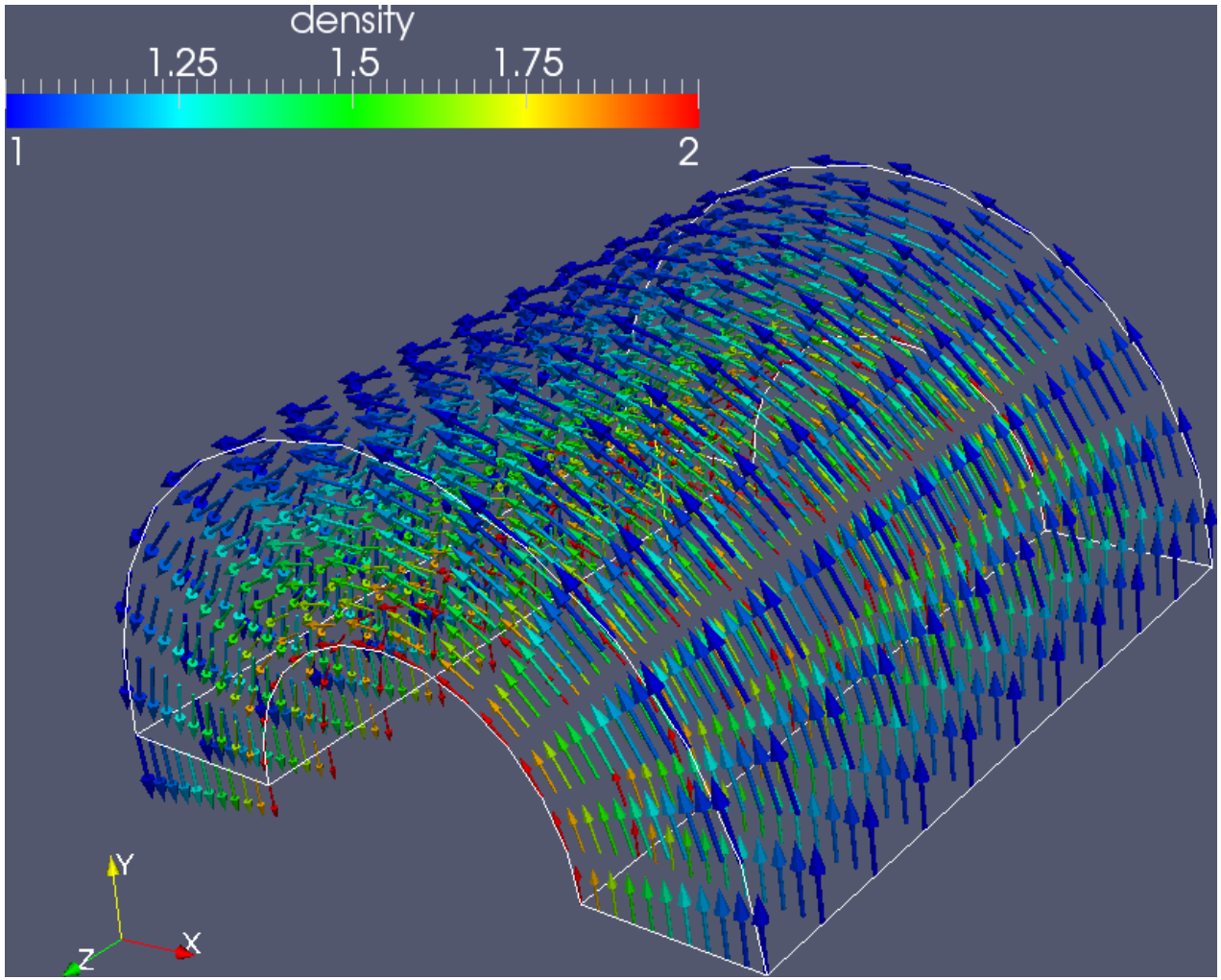
- Start with original data (`data/other/disk_out_ref.ex2`), load all variables
- Add the **Clip** filter, uncheck "Show Plane" in the object inspector, click "Apply"
- Color the surface by pressure by changing the variable chooser in the toolbar from "Solid Color" to "Pres"
- Press "Split horizontal", make sure the view in the right is active (has a blue border around it)
- Turn on the visibility of the clipped data by clicking the eyeball next to Clip in the pipeline browser
- Color the surface by temperature by changing the toolbar variable chooser from "Solid Color" to "Temp"
- Can reset (fit/reposition) the view in either column by clicking "Reset"
- To link the two views, right click on one of the views and select "Link Camera...", click in a second view, and try moving the object in each view
- Can add colourbars to either view by clicking "Toggle Color Legend Visibility", try moving colourbars around

Vector visualization: streamlines and glyphs

- Start with the original data from `data/other/disk_out_ref.ex2`, load velocity, Temp
- Add the **Stream Tracer** filter, play with Seed Type ("Point Source", "Line Source"), other parameters
- Add shading and depth cues to streamlines: Filters → Alphabetical → **Tube** (could be also called Generate Tubes)
- Add glyphs to streamlines to show the orientation and magnitude:
  - select StreamTracer in the pipeline browser
  - add the **Glyph** filter to StreamTracer
  - in the object inspector, change the Vectors option (second from the top) to "V"
  - in the object inspector, change the Glyph Type option (third from the top) to "Cone"
  - hit "Apply"
  - color the glyphs with the "Temp" variable
- Now try displaying "V" glyphs directly from data, can colour them using different variables ("Temp", "V")

Exercise: vectors

Load `data/vtk/xml/halfCylinder.vts` and display the velocity field as arrows, colouring them by density – try to reproduce the view below

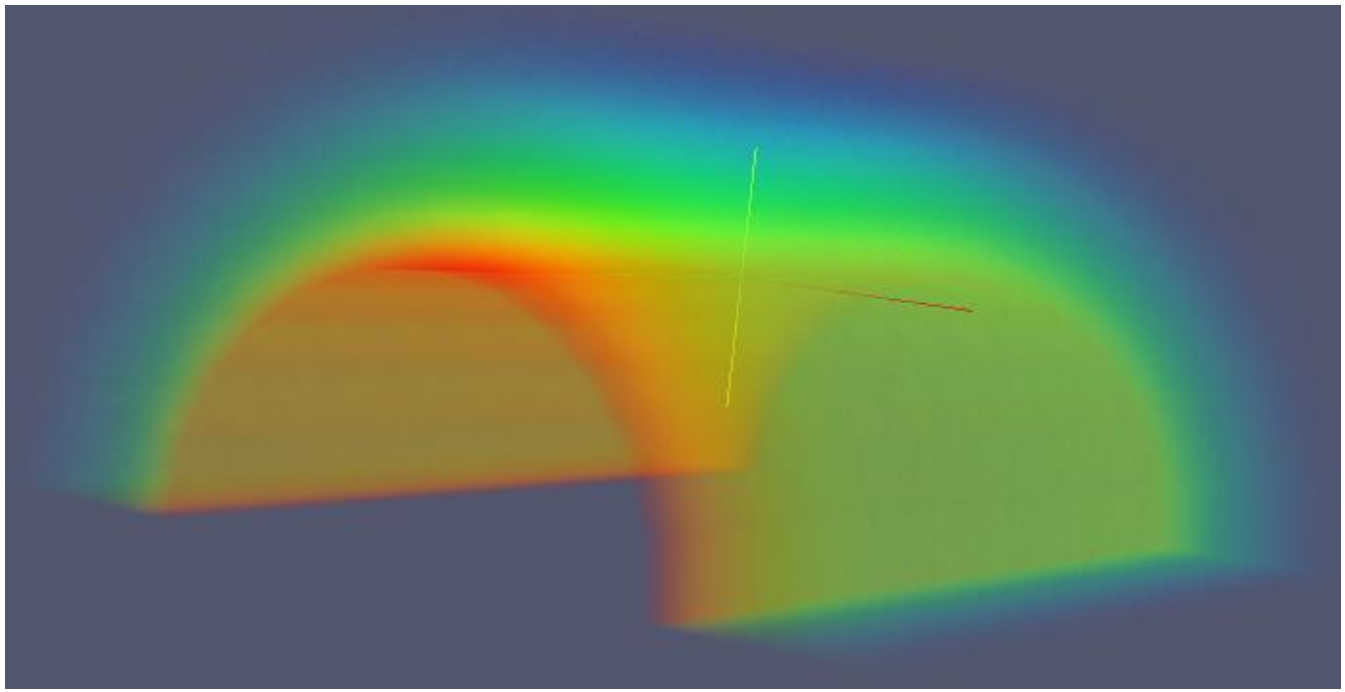


Word of caution

- Many visualization filters transform stuctured grid data into unstructured data (e.g. Clip, Slice)
- Memory footprint and CPU load can grow very quickly, e.g. clipping 400^3 to 150 million cells can take $\sim$ 1 hour on a single CPU $\Rightarrow$ might want to run in distributed mode

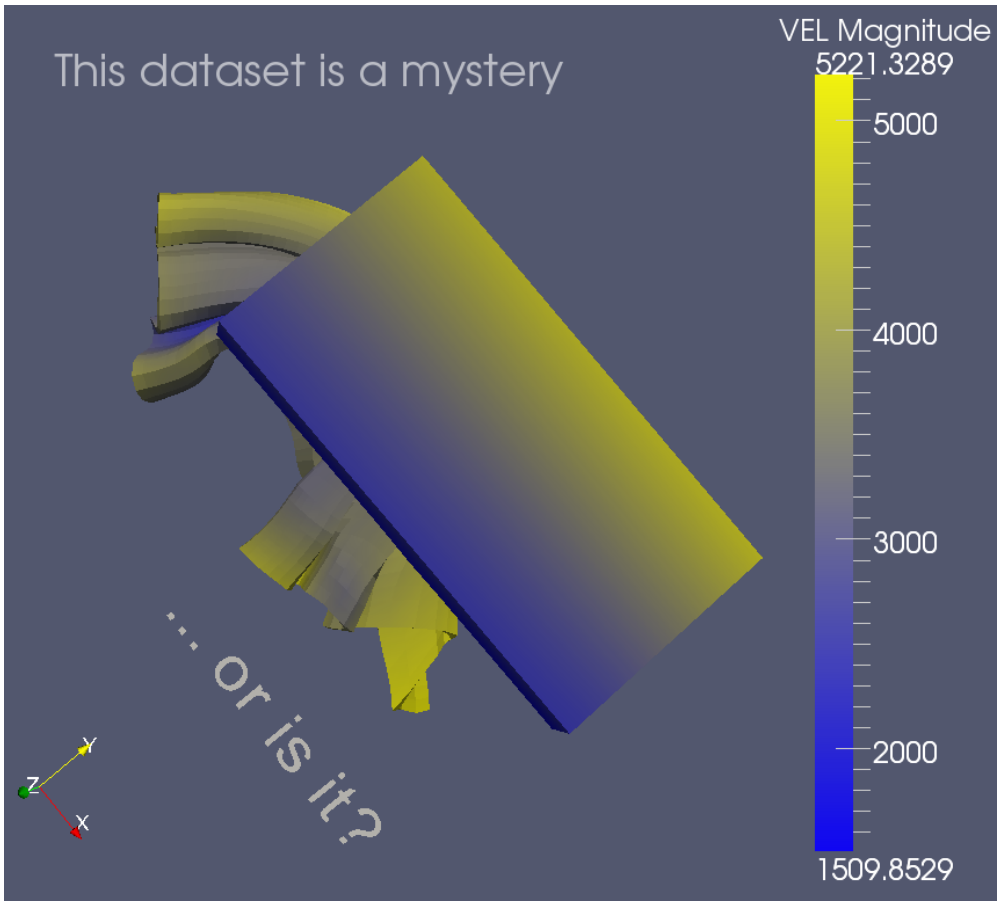
Volume Rendering

- How can we do volumetric rendering of datasets where it's not available?
  - "Volume" view is available only for Structured Points (regularly spaced grid) and Unstructured Grid (3D "polygons")
- What about Structured Grid – try loading `halfCylinder.vts` and doing volumetric rendering of density



Text Annotation

- Select Sources → Text, type in the text edit box of the object inspector, hit "Apply", edit Display properties
- Sources → 3D Text



Batch scripting for automating visualization

- One can automate mundane or repetitive tasks or use ParaView without GUI, complete documentation at `http://www.paraview.org/Wiki/ParaView/Python_Scripting`
- Tools → Python Shell
- *[/usr/bin/ /usr/local/bin/ /Applications/paraview.app/Contents/bin/] pvpython* will give you a Python shell connected to a ParaView server (local or remote) without the GUI
- *[/usr/bin/ /usr/local/bin/ /Applications/paraview.app/Contents/bin/] pvbatch pythonScript.py* is a serial (on some machines parallel) application using local server
  - great for making movies!

First script

- Bring up Tools → Python Shell
- "Run Script" `code/displaySphere.py`

displaySphere.py

```
from paraview.simple import *

sphere = Sphere() # create a sphere pipeline object

print sphere.ThetaResolution # print one of the attributes of the sphere
sphere.ThetaResolution = 16

Show() # turn on visibility of the object in the view
Render()
```

- Can always get help from the command line

```
help(paraview.simple)
help(sphere)
help(Sphere)
```

Using filters

- "Run Script" `code/displayWireframe.py`

displayWireframe.py

```
from paraview.simple import *

sphere = Sphere(ThetaResolution=36, PhiResolution=18)

wireframe = ExtractEdges(Input=sphere) # apply Extract Edges to sphere

Show() # turn on visibility of the last object in the view
Render()
```

- Now try replacing Show() with Show(sphere)

Reading from files

- "Run Script" `code/readDiskOutRef.py` (change the path!)

readDiskOutRef.py

```
from paraview.simple import *
path = '/Users/razoumov/Dropbox/visualization/data/other/'
reader = ExodusIIReader(FileName=path+'disk_out_ref.ex2')
Show()
Render()
```

- With VTK file formats can use something like:

```
reader = XMLStructuredGridReader(FileName='/Users/.../vtk/xml/halfCylinder.vts')
```

- Starting with ParaView 3.8, can load correct reader automatically using file extension:

```
reader = OpenDataFile('/Users/.../vtk/xml/halfCylinder.vts')
```

Querying field attributes: readStructuredGrid.py

```
from paraview.simple import *

path = '/Users/razoumov/Dropbox/visualization/data/vtk/xml/' # edit the path accordi
reader = OpenDataFile(path+'halfCylinder.vts')

Show()
Render()

print 'print all variables'
print reader.PointData[:]

print 'get a handle to PointData and print all point fields'
pd = reader.PointData
print pd.keys()

print 'get some info about individual fields'
print pd['density'].GetNumberOfComponents()
print pd['density'].GetRange()
print pd['velocity'].GetNumberOfComponents()

print 'run through all arrays and print the ranges of all components'
for ai in pd.values():
    print ai.GetName(), ai.GetNumberOfComponents(),
    for i in xrange(ai.GetNumberOfComponents()):
        print ai.GetRange(i),
    print
```

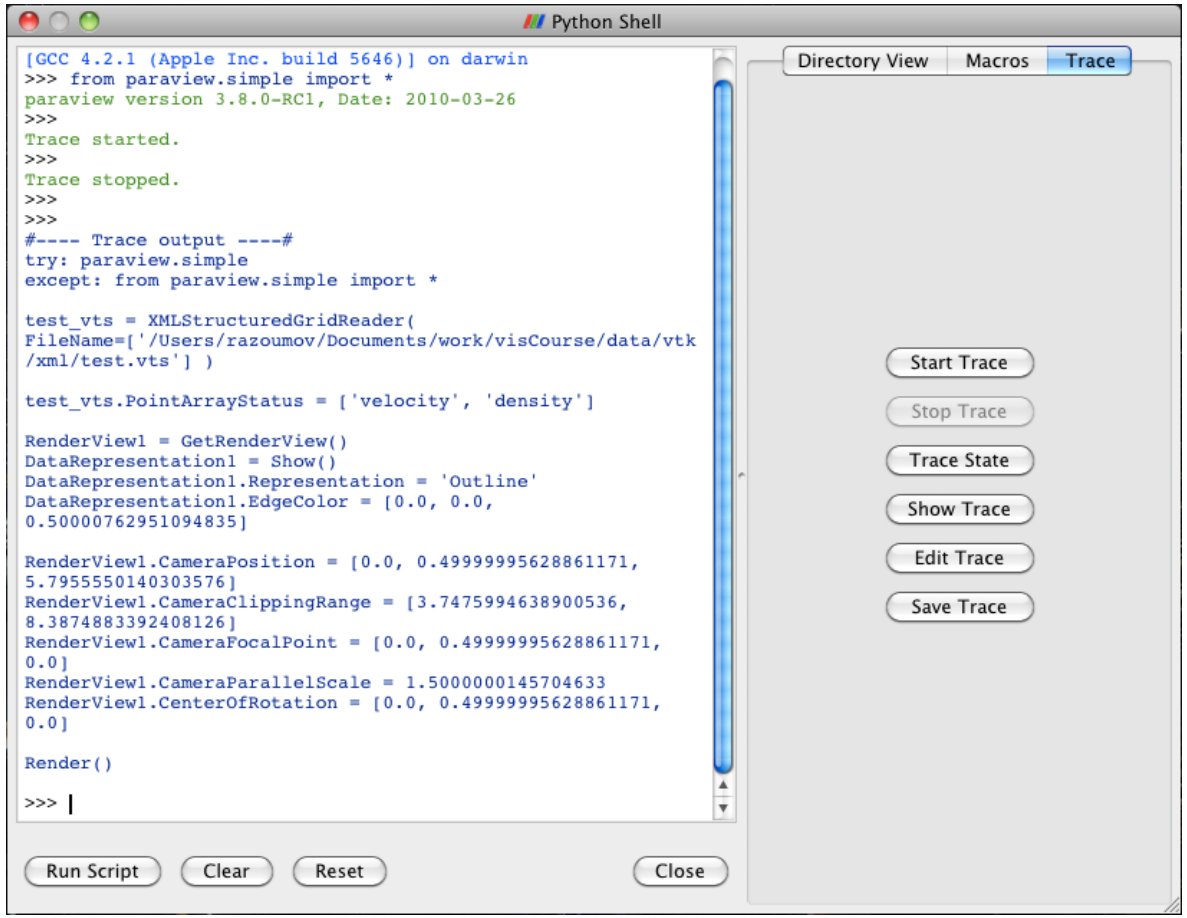
Trace tool

Generate Python code from GUI operations

Start/stop trace at any time

Older ParaView: Tools → Python Shell → Trace → [Start | Stop | Show Trace]

Newer ParaView: Tools → [Start Trace | Stop Trace]



More complex example generated via trace

"Run Script" `code/writeImage.py` – draws vector field in half-cylinder

```
from paraview.simple import *

path = '/Users/razoumov/Dropbox/visualization/data/vtk/xml/' # edit the path accordingly
test_vts = XMLStructuredGridReader(FileName=[path+'halfCylinder.vts'])

DataRepresentation1 = Show() # turn on outline
DataRepresentation1.Representation = 'Outline'
DataRepresentation1.EdgeColor = [0.0, 0.0, 0.5]

# set camera position
RenderView = GetRenderView()
RenderView.CameraViewUp = [-0.25, 0.82, -0.51]
RenderView.CameraFocalPoint = [0., 0.5, 0.]
RenderView.CameraClippingRange = [2.91, 9.55]
RenderView.CameraPosition = [1.85, 3.79, 4.40]

Glyph2 = Glyph(GlyphType="Arrow" )
Glyph2.Scalars = ['POINTS', 'density']
Glyph2.SetScaleFactor = 0.2
Glyph2.Vectors = ['POINTS', 'velocity']
Glyph2.SetScaleFactor = 0.2

DataRepresentation2 = Show() # turn on vectors
DataRepresentation2.EdgeColor = [0.0, 0.0, 0.5]
DataRepresentation2.ColorAttributeType = 'POINT_DATA'
DataRepresentation2.ColorArrayName = 'density'

# set colour table
a1_density_PiecewiseFunction = CreatePiecewiseFunction( Points=[-15.70, 0.0, -5.7, 0.0, -4.23, 0.0, -4.07, 0.1, -3.21,
a1_density_PVLookupTable = GetLookupTableForArray( "density", 1, RGBPoints=[1.0, 0.0, 0.0, 0.0, 1.64, 0.90, 0.0, 0.0,
DataRepresentation2.LookupTable = a1_density_PVLookupTable

WriteImage('/Users/razoumov/Desktop/output.png')
Render()
```

Working with pipeline objects

`GetSources()` - get a list of objects
`GetActiveSource()` - get the active object
`SetActiveSource` - change the active object
`GetRepresentation()` - return the *view representation* for the active pipeline object and the active view

the following two scripts produce identical results
(see `getRepresentation.py`):

from paraview.simple import *	from paraview.simple import *
test_vts = XMLStructuredGridReader(FileName=['halfCylinder.vts'])	test_vts = XMLStructuredGridReader(FileName=['halfCylinder.vts'])
DataRepresentation = Show()	Show()
	handle = GetRepresentation()
DataRepresentation.Representation = 'Surface'	handle.Representation = 'Surface'
DataRepresentation.DiffuseColor = [0, 0, 1]	handle.DiffuseColor = [0, 0, 1]
DataRepresentation.SpecularColor = [1, 1, 1]	handle.SpecularColor = [1, 1, 1]
DataRepresentation.SpecularPower = 200	handle.SpecularPower = 200
DataRepresentation.Specular = 1	handle.Specular = 1
Render()	Render()

Animation

1. Use ParaView's built-in animation capabilities
   - animate any property of any pipeline object
   - in Animation View: select object, select property, create a new track with "+", double-click the track to edit it, press "play"
   - lets you create snazzy animations, but very limited in what you can do
2. Script your animation in Python
   - steep learning curve, but very powerful
   - typical usage scenario: generate one frame per input file
   - we'll try a simpler exercise without input files: write a script that
     - imports Sources $\rightarrow$ Cone
     - sets the cone's resolution to $6 + i$, where $i = 0, \ldots, 20$
     - saves each frame to a file called "cone"+str(i)+".png"
     - (optionally we can make a movie from these frames)

Visualizing remote dataset

- What we've covered so far: working with standalone ParaView on your desktop
- Let's say, your dataset is on cluster.consortium.ca
  $\Rightarrow$ fundamentally there are three options:
  1. download data to your desktop and visualize it locally (limited by dataset size and your desktop's CPU/memory)
  2. in SHARCNET can work through a visualization workstation
     yourDesktop $\xrightarrow{\text{ssh/NX/VNC}}$ vizN-site running ParaView mounting /work/user
  3. work in a **client-server mode** connecting to cluster.consortium.ca directly
     ParaView client on yourDesktop $\rightleftharpoons$ ParaView server on remote cluster
     - currently not used in SHARCNET but possible to set up if necessary
     - setup details depend on the consortium

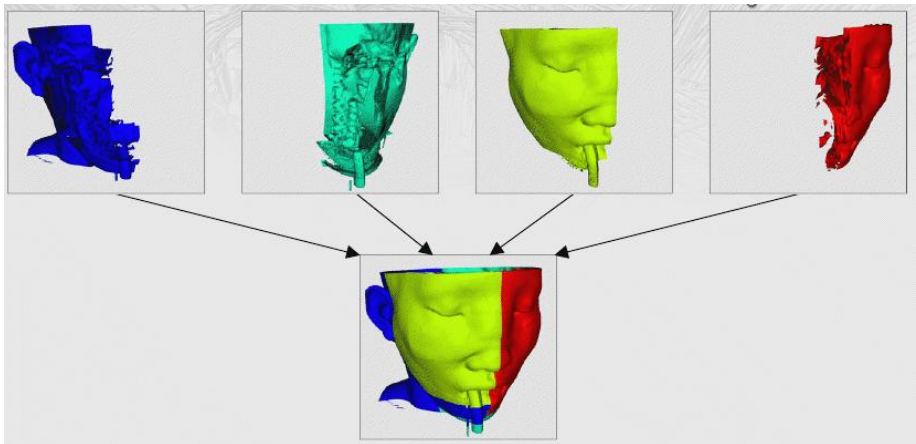
Data partitioning

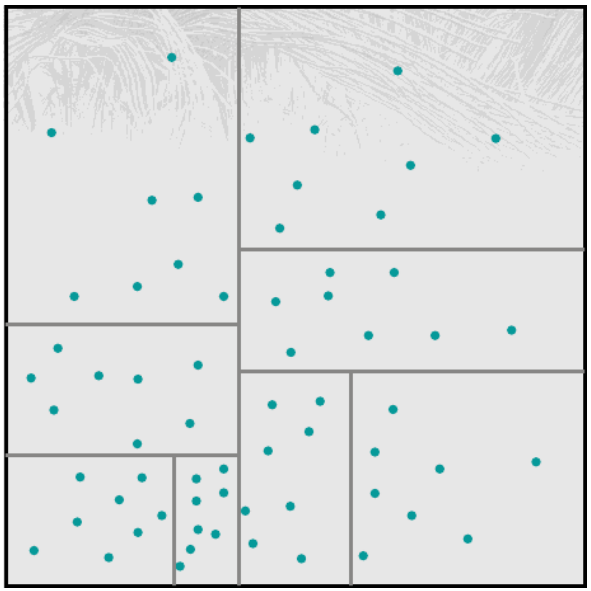
Scalable parallel distributed rendering – load balancing is handled automatically by ParaView for structured data:

- Structured Points
- Rectilinear Grid
- Structured Grid

Unstructured data must be passed through D3 (Distributed Data Decomposition) filter for better load balancing:

- Particles/Unstructured Points
- Polygonal Data
- Unstructured Grid





Best strategies for large datasets

- Working with structured data (Structured Points, Rectilinear Grid, Structured Grid): one processor core per 10-20 million cells ← *according to ParaView documentation*
- Unstructured data (Unstructured Points, Polygonal Data, Unstructured Grid): one processor core per 0.5-1 million cells ← *according to ParaView documentation*
- In practice, memory is the main issue, e.g. with structured data can do:
  - $\sim 1000^3$ on a notebook with 4 GB memory, single/dual core
  - $\sim 2000^3$ on a viz workstation with 50 GB memory, dual/quad core
- Rainbow: 20 compute nodes, 4 cores / 8 GB memory per node
- Always do a scaling study before attempting to visualize large datasets
- It is important to understand memory requirements of filters

Working with large datasets

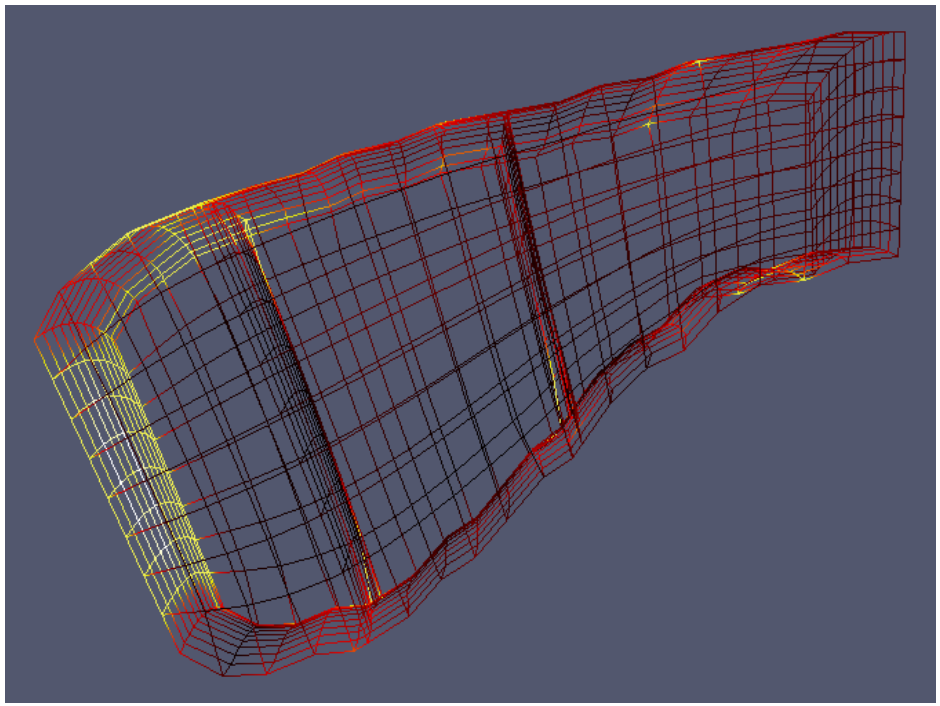
Some filters **should not be used with structured data**:
they write unstructured data, can be heavy on memory usage

- 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

use these with caution: Clip, Decimate, Extract Cells by Region, Extract Selection, Quadric Clustering, Threshold

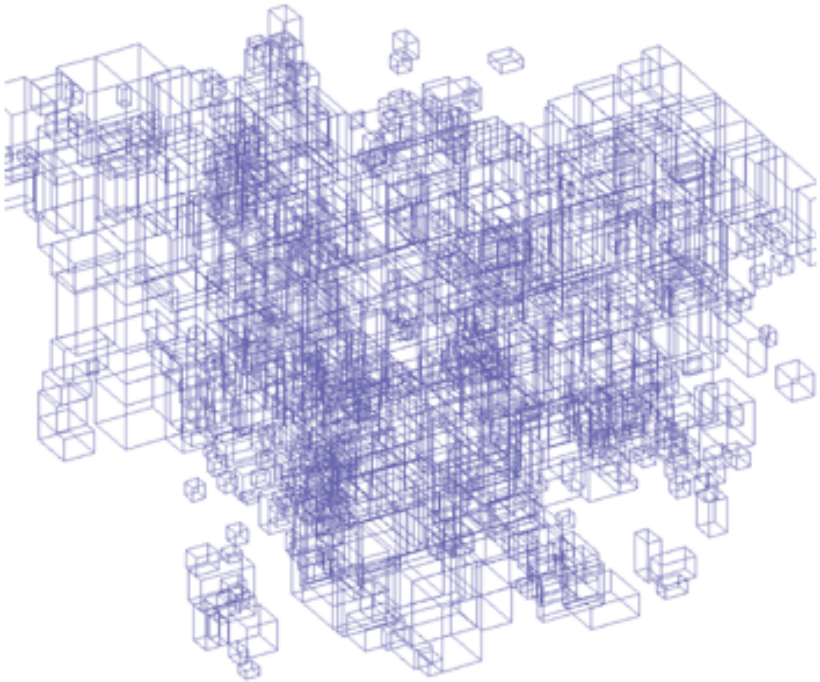
VTK composite datasets: vtkMultiBlockDataSet

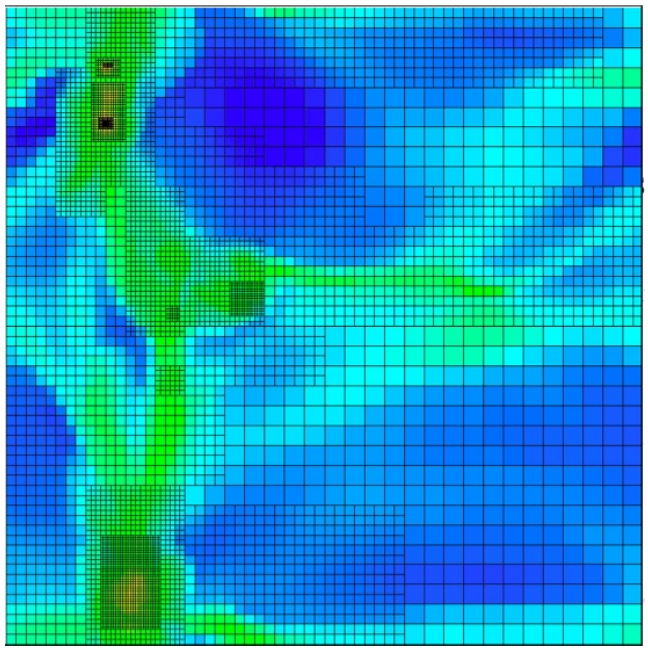
- **vtkMultiBlockDataSet** is a dataset comprising of blocks. Each block can be either a leaf (non-composite), or an instance of vtkMultiBlockDataSet itself – this makes is possible to build trees
- Study MultiBlock.C (adapted from from VTK/Examples/MultiBlock): loads three separate structured grid datasets, each from its own file, and writes them as a single multi-block *.vtm dataset (XML-based file format)



VTK composite datasets: vtkHierarchicalBoxDataSet

- **vtkHierarchicalBoxDataSet** is used for AMR datasets, comprises of refinement levels and uniform grid datasets at each refinement level
- prototype code hierarchicalBoxDataWriter.C (does not assign scalars yet, need to sort out cell centers vs. cell edges) – writes multiple grids as a single hierarchical *.vtm dataset





- more on composite datasets
  `http://www.itk.org/Wiki/Composite_Datasets_in_VTK`
  `http://www.itk.org/Wiki/VTK/Composite_Data_Redesign`

Further resources

extended ParaView tutorial and sample data in many different formats
`http://www.cmake.org/Wiki/The_ParaView_Tutorial`

ParaView F.A.Q. `http://www.itk.org/Wiki/ParaView:FAQ`

VTK for C++/Python/etc. code examples
`http://www.itk.org/Wiki/VTK/Examples`

VTK file formats `http://www.vtk.org/VTK/img/file-formats.pdf`

generate XDMF files in Fortran with
(1) either HDF5 calls + plain text for the XML,
(2) or Fortran calling C functions with XDMF
`http://www.xdmf.org/index.php/Write_from_Fortran`

Exercises

- Which file format would work best for your dataset?
  - VTK: versatile, support for many different data types (arrays, curvilinear grids, polygons, irregular topologies, particles)
  - NetCDF: best for multidimensional arrays (data on regular grids)
  - HDF5:
    - if your code already outputs HDF5 $\Rightarrow$ easy to add XML metadata `http://www.xdmf.org`
    - ParaView understands HDF5 output of several third-party packages
- More complex animation: loading a sequence of data files