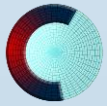
Kitware

In-Situ Visualization with ParaView Catalyst

For ParaView Catalyst 1.0

Agenda

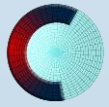
- Introduction to ParaView Catalyst
- Catalyst for Users
- Catalyst for Developers



Kitware

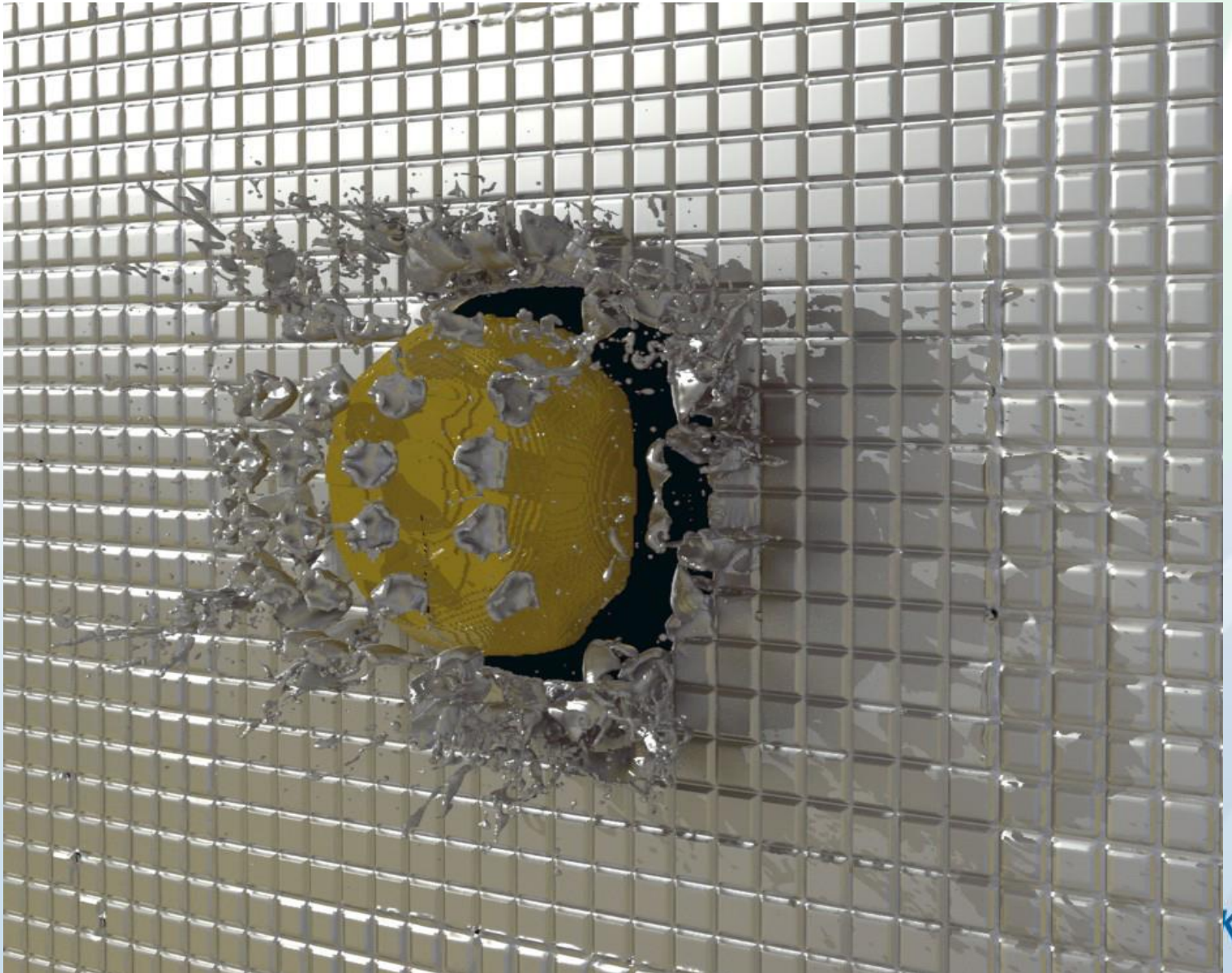
Online Help

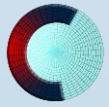
- Email list: paraview@paraview.org
- Doxygen:
  - http://www.vtk.org/doc/nightly/html/classes.html
  - http://www.paraview.org/ParaView3/Doc/Nightly/html/classes.html
- Sphinx:
  - http://www.paraview.org/ParaView3/Doc/Nightly/www/py-doc/index.html
- Websites:
  - http://www.paraview.org
  - http://catalyst.paraview.org
- Examples:
  - https://github.com/acbauer/CatalystExampleCode



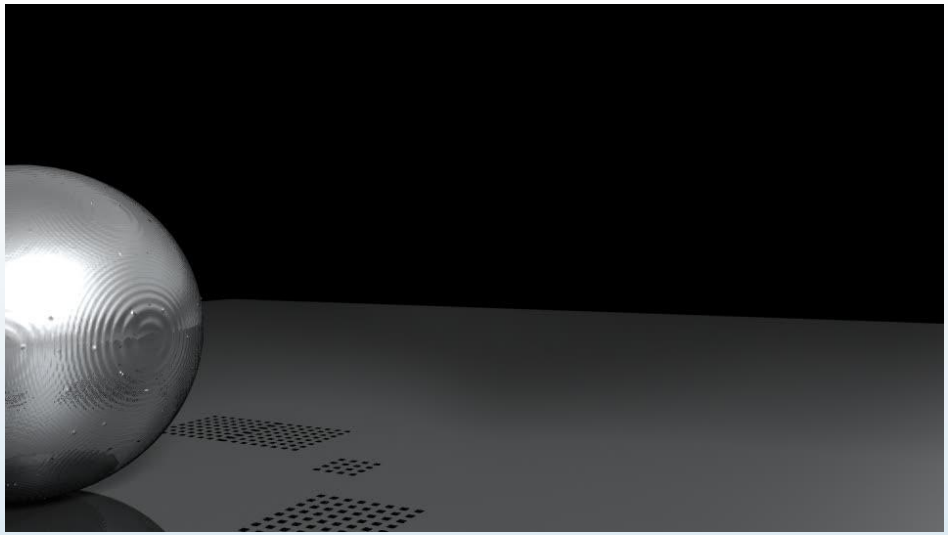
Kitware

What is Catalyst

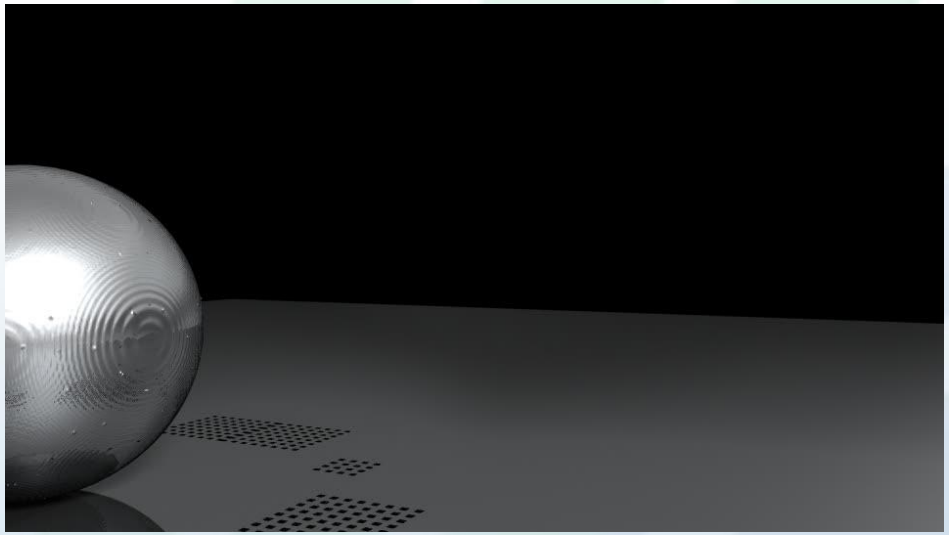




Access to More Data



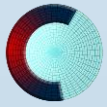
Post-processing



In situ processing

Roughly equal data stored at simulation time

Reflections and shadows added in post-processing for both examples

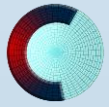


Kitware

Why *In Situ*?

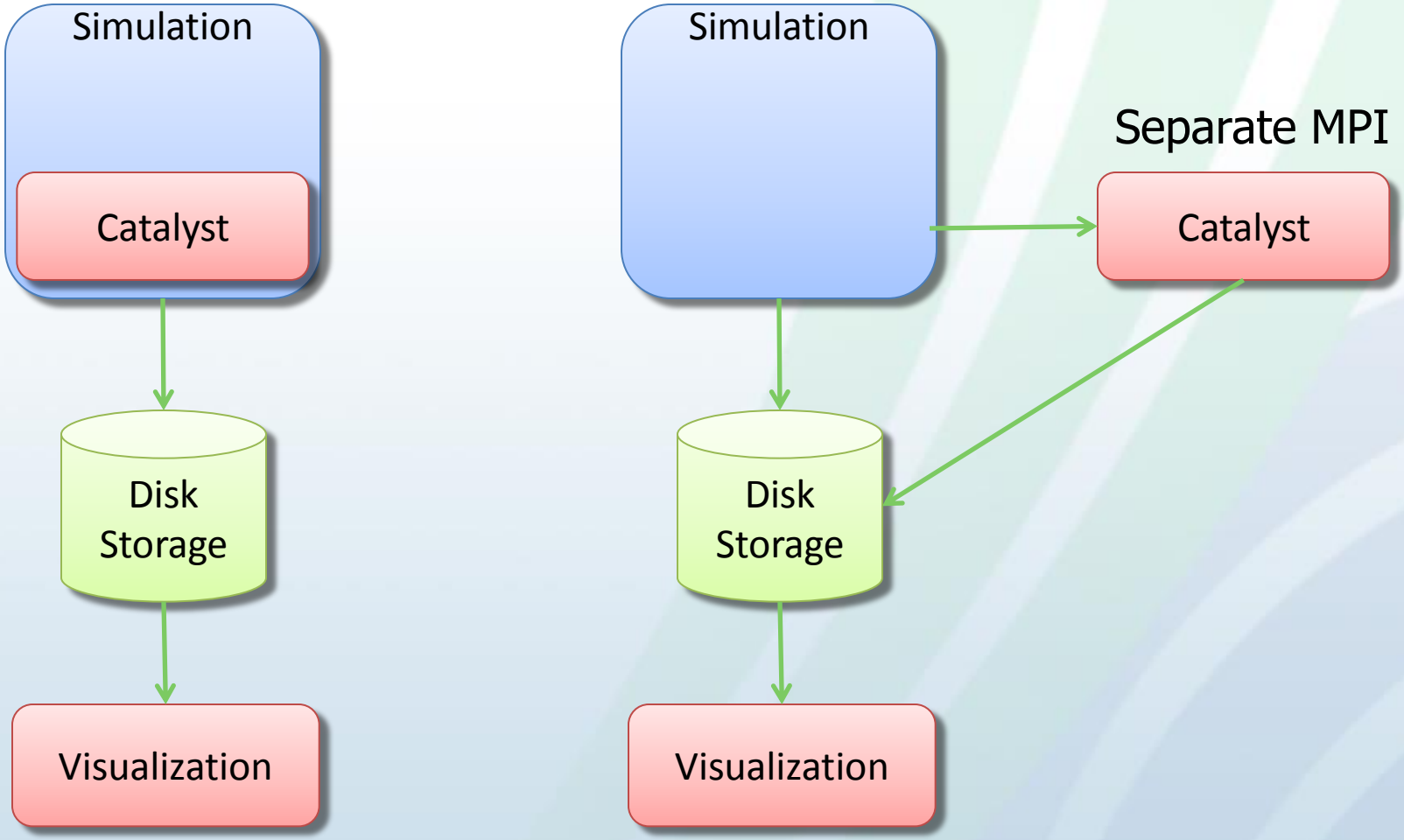
	2010	2018	Factor Change
System peak	2 Pf/s	1 Ef/s	500
Power	6 MW	20 MW	3
System Memory	0.3 PB	10 PB	33
Node Performance	0.125 Gf/s	10 Tf/s	80
Node Memory BW	25 GB/s	400 GB/s	16
Node Concurrency	12 cpus	1,000 cpus	83
Interconnect BW	1.5 GB/s	50 GB/s	33
System size (nodes)	20 K nodes	1 M nodes	50
Total Concurrency	225 K	1 B	4,444
Storage	15 PB	300 PB	20
Input/Output Bandwidth	0.2 TB/s	20 TB/s	100

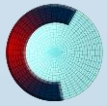
DOE Exascale Initiative Roadmap, Architecture and Technology Workshop, San Diego, December, 2009.



Kitware

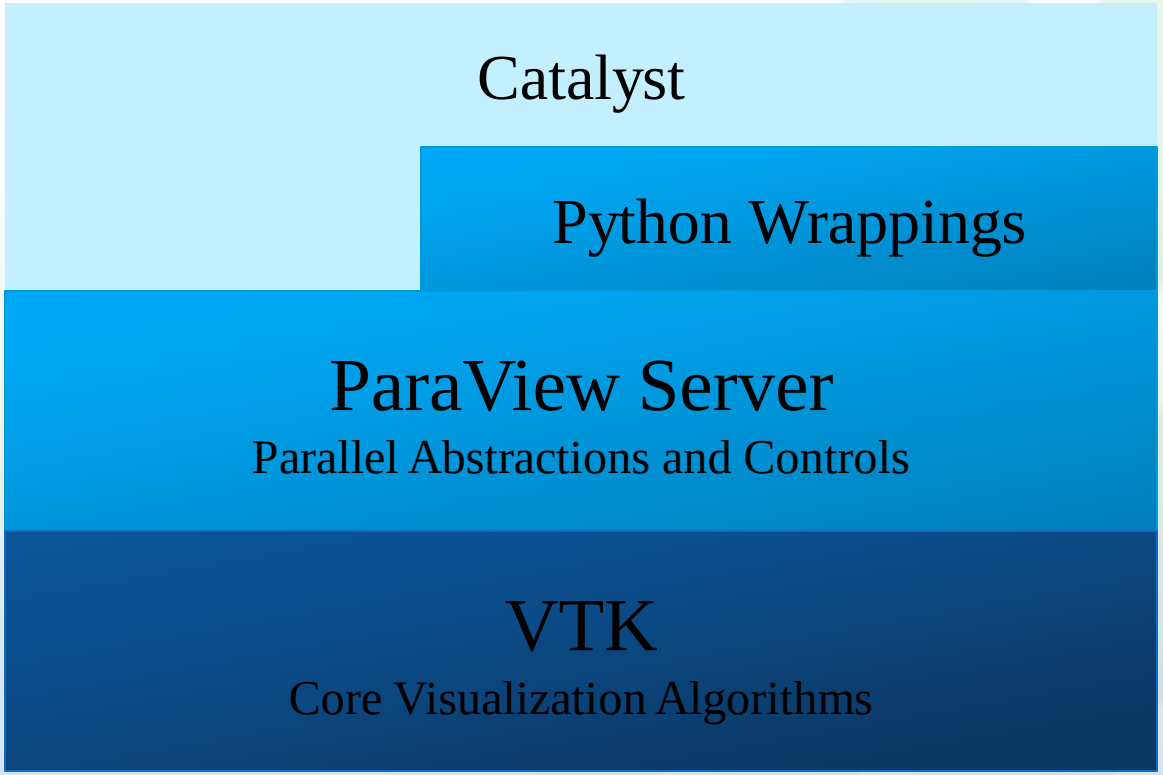
Two ways to run

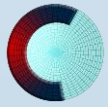




Kitware

Catalyst Architecture





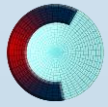
High Level View

Simulation Users

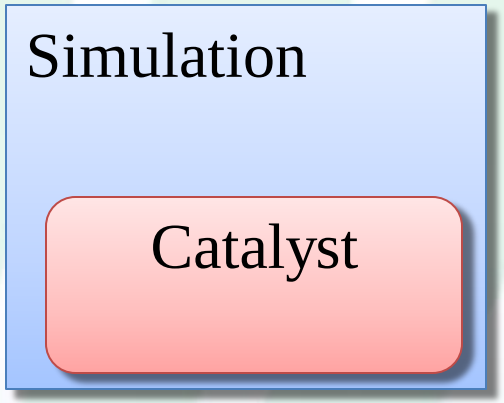
- Knowledge of ParaView as a post-processing/analysis tool
  - Basic interaction with GUI co-processing script generator plugin
  - Incremental knowledge increase to use the co-processing tools from basic ParaView use
- Programming knowledge can be useful to extend the tools

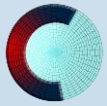
Simulation Developers

- Pass necessary simulation data to ParaView
- Need sufficient knowledge of both codes
  - VTK for grids and field data
  - ParaView Catalyst libraries
- Transparent to simulation users
- Extensible



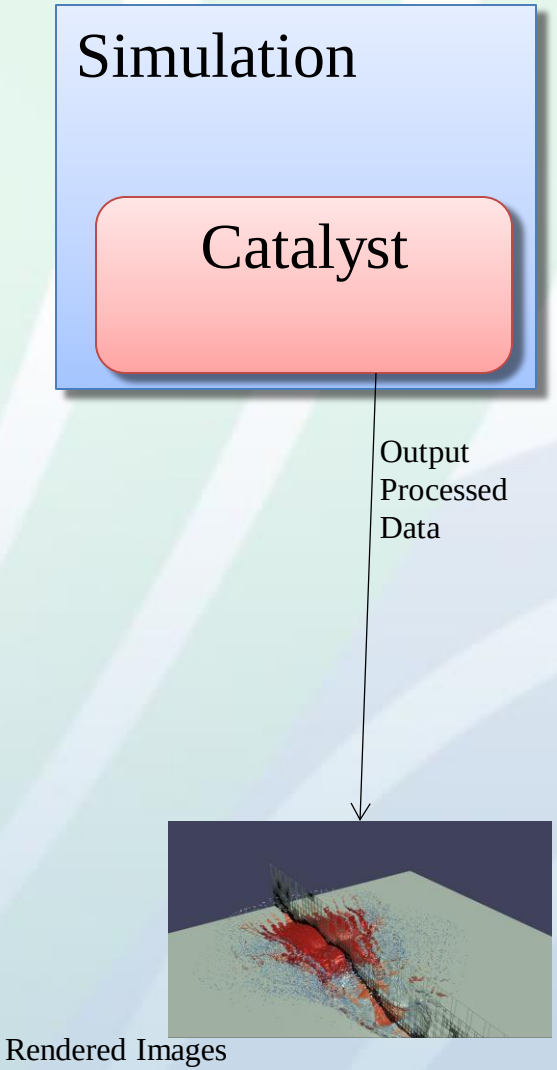
User Perspective

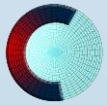




Kitware

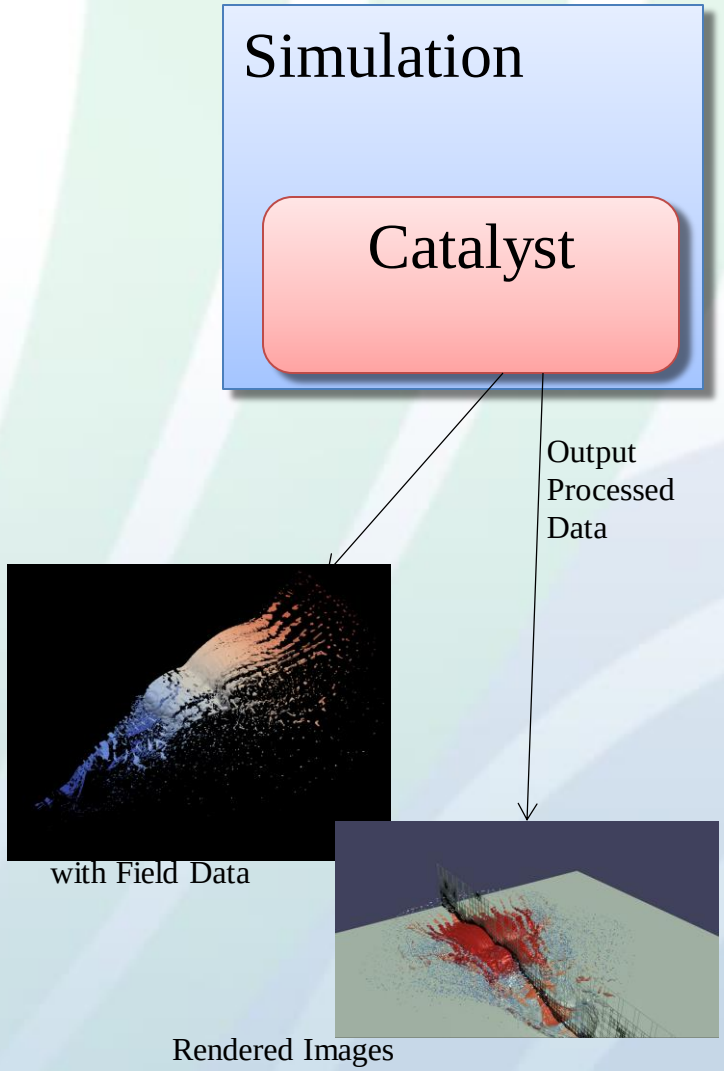
User Perspective

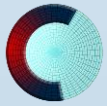




Kitware

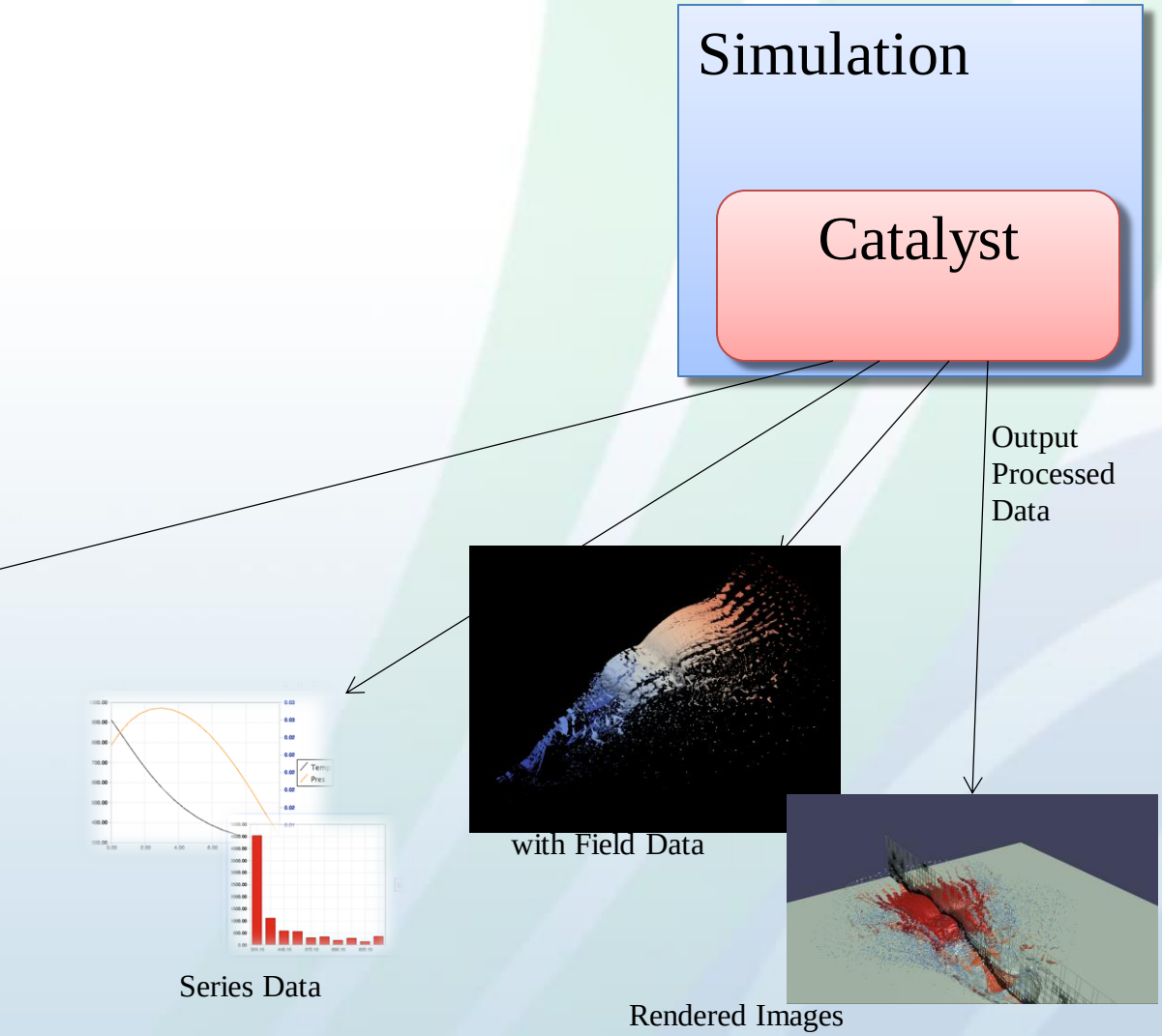
User Perspective

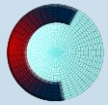




Kitware

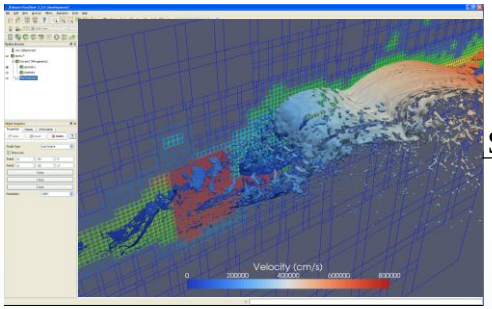
User Perspective





Kitware

User Perspective



Script Export

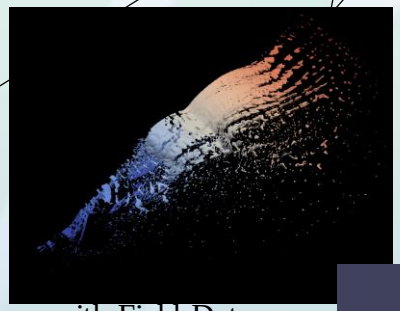
```
# Create the reader and set the filename.
reader = servermanager.sources.Reader(FileNames=path)
view = servermanager.CreateRenderView()
repr = servermanager.CreateRepresentation(reader,  view)
reader.UpdatePipeline()
dataInfo = reader.GetDataInformation()
pDinfo = dataInfo.GetPointDataInformation()
arrayInfo = pDInfo.GetArrayInformation("displacement9")
if arrayInfo:
 # get the range for the magnitude of displacement9
 range = arrayInfo.GetComponentRange(-1)
 lut = servermanager.rendering.PVLookupTable()
 lut.RGBPoints  = [range[0], 0.0, 0.0, 1.0,
            range[1], 1.0, 0.0, 0.0]
 lut.VectorMode = "Magnitude"
 repr.LookupTable = lut
 repr.ColorArrayName = "displacement9"
 repr.ColorAttributeType = "POINT_DATA"
```

Augmented script in input deck.

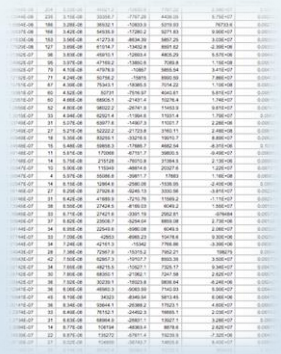
Simulation

Catalyst

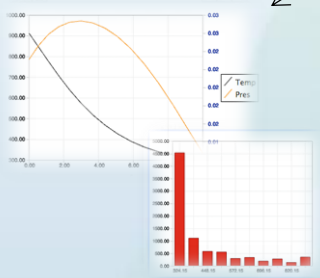
Output Processed Data



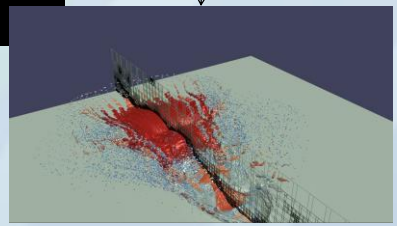
with Field Data



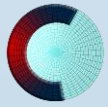
Statistics



Series Data

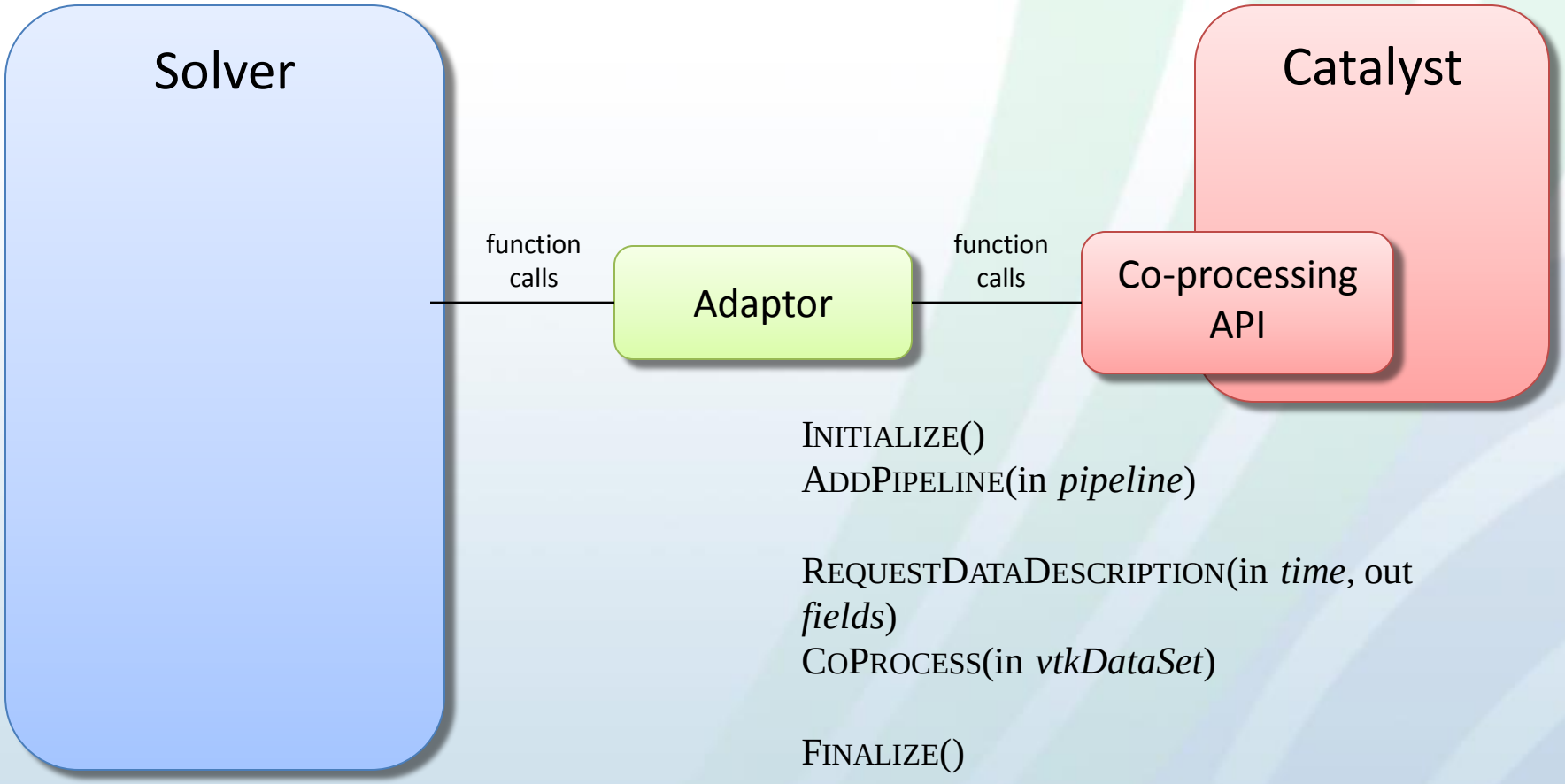


Rendered Images



Kitware

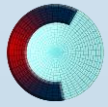
Developer Perspective



INITIALIZE()
ADDPIPELINE(in *pipeline*)

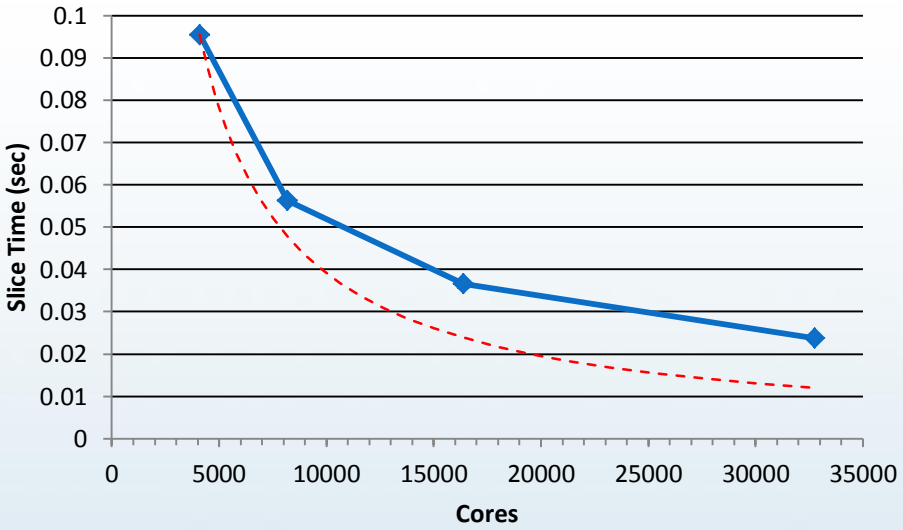
REQUESTDATADESCRIPTION(in *time*, out *fields*)
COPROCESS(in *vtkDataSet*)

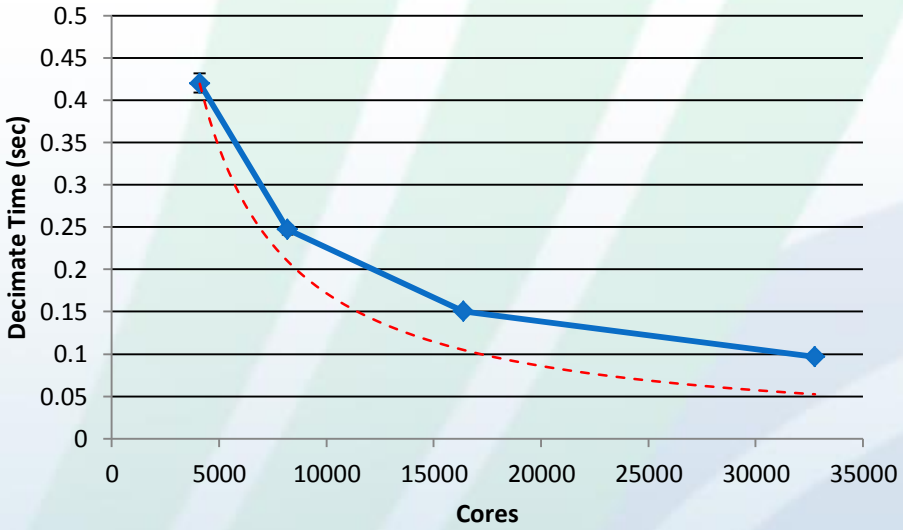
FINALIZE()

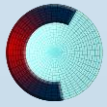


Kitware

Developer Perspective







Kitware

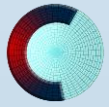
Kitware

ParaView Catalyst for Simulation Users

Creating Catalyst Output

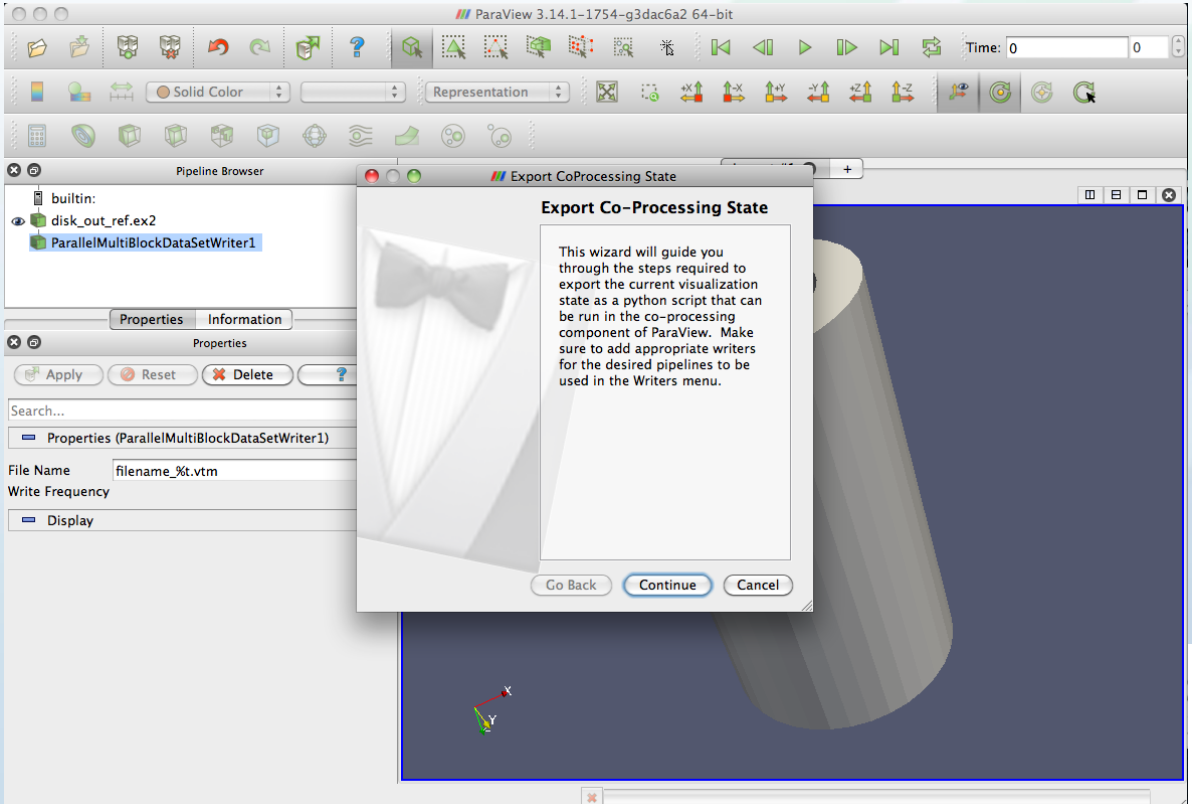
Two main ways:

- Create and/or modify Python scripts
  - ParaView GUI plugin to create Python scripts
  - Modification with knowledge of ParaView Python API
- Developer generated “canned” scripts
  - User provides parameters for already created Catalyst pipelines
  - User may not even need to know ParaView
  - See ParaView Catalyst User’s Guide



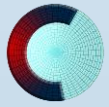
Create Python Scripts from ParaView

- Interact with ParaView normally
- Export a script that mimics that interaction
- Queries during each co-processing step
  - (one frame at a time)



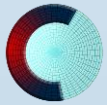
ParaView GUI Plugin

- Similar to using ParaView interactively
  - Setup desired pipelines
  - Ideally, start with a representative data set from the simulation
- Extra pipeline information to tell what to output during simulation run
  - Add in data extract writers
  - Create screenshots to output
  - Both require file name and write frequency



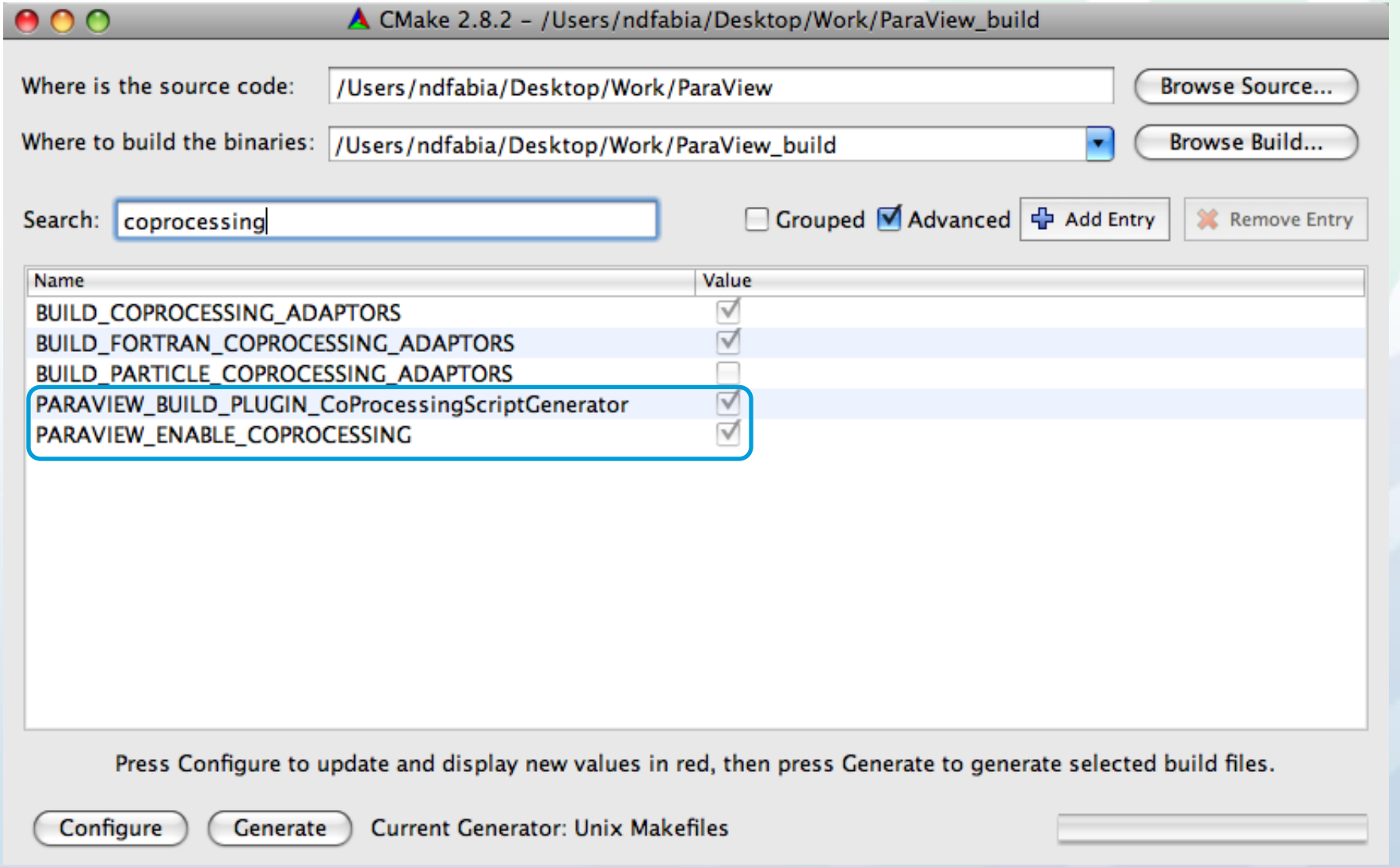
In Situ Demo

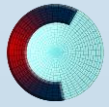
- Create a ParaView Catalyst Python pipeline script
  - Specify desired outputs from run
  - Export the script
- Run the script with a fictitious input
  - Time dependent grid and field data come from a file instead of from an actual simulation
- Examine results



Kitware

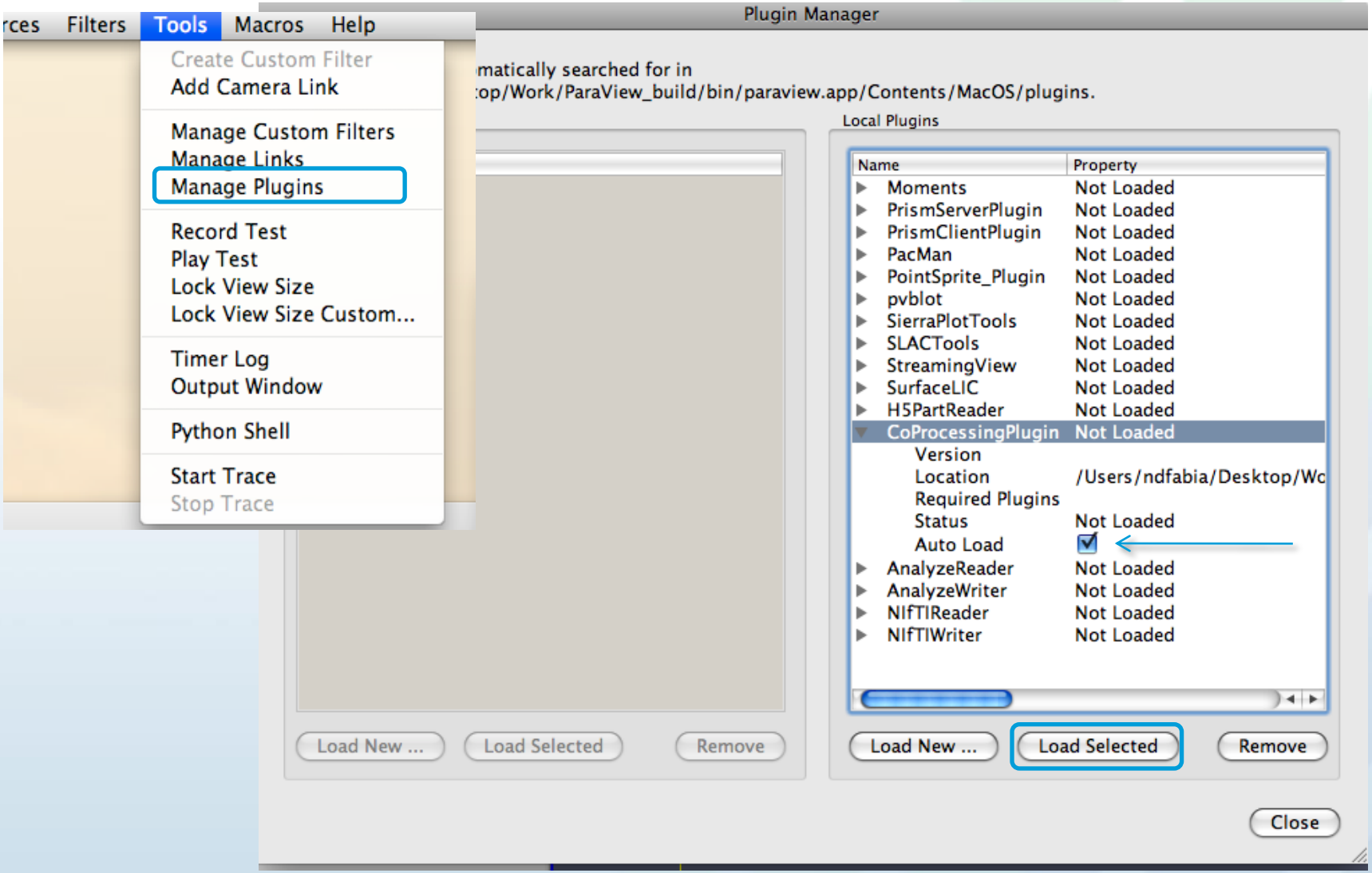
In Situ Demo – Build Step

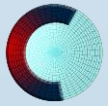




Kitware

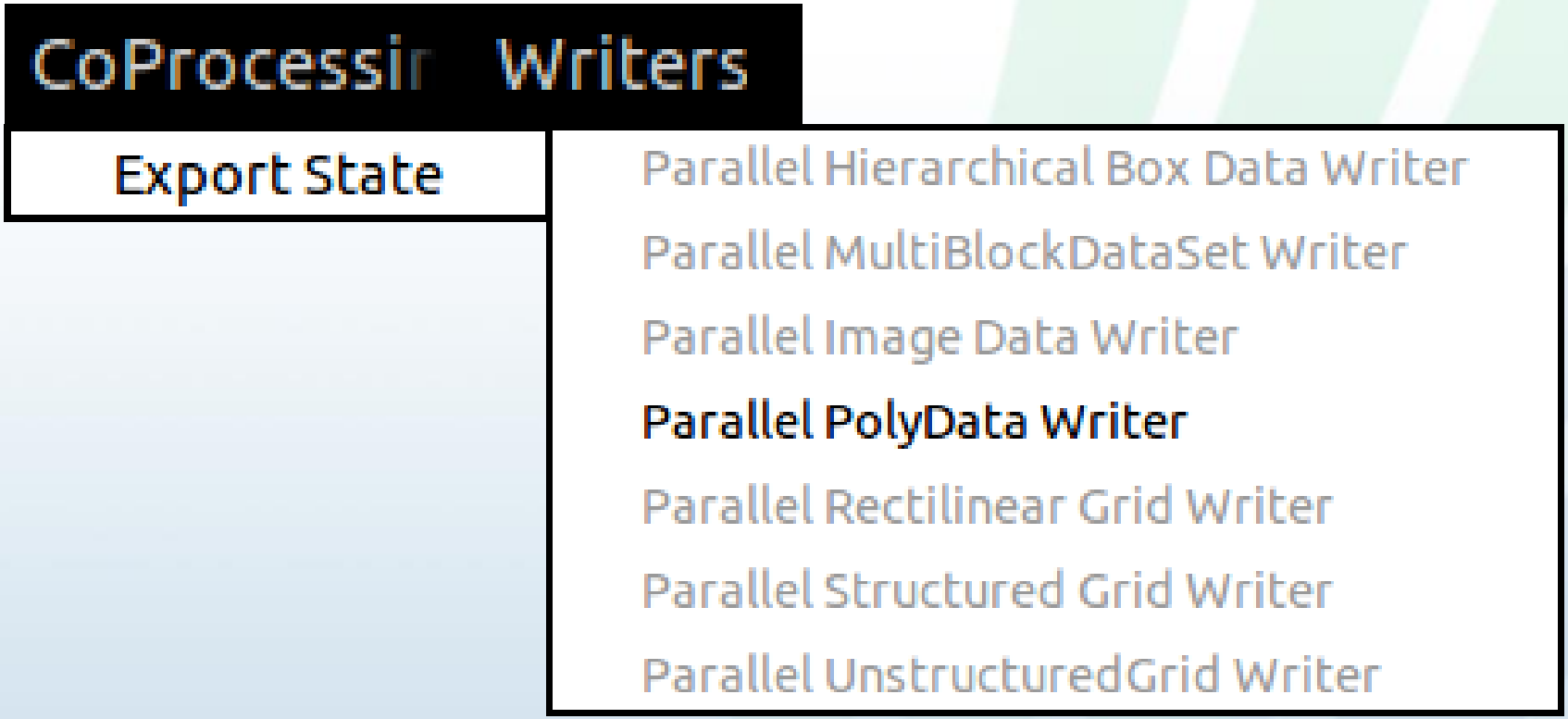
In Situ Demo – Load Plugin Step

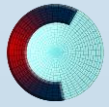




Kitware

In Situ Demo – New Plugin Menus

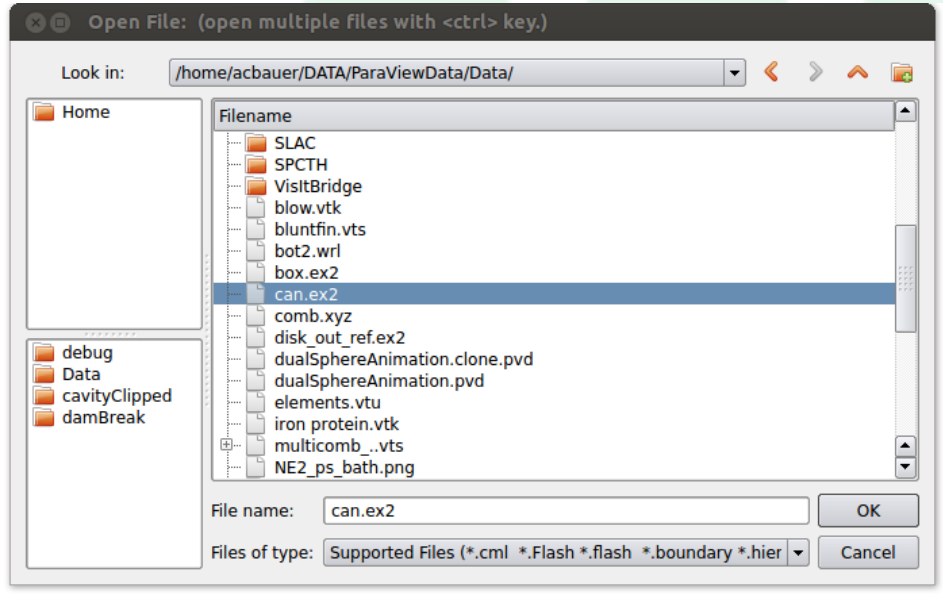




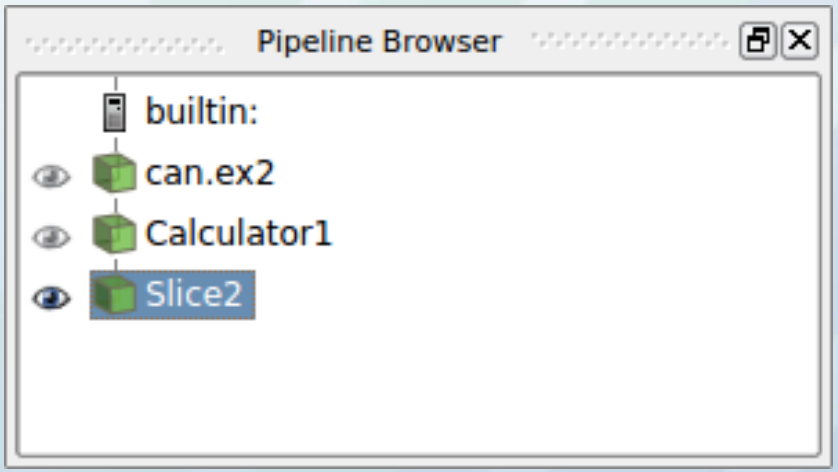
Kitware

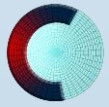
In Situ Demo – Creating a Catalyst Python Script

- Load can.ex2
  - Note that there are 44 time steps



- Create desired pipeline

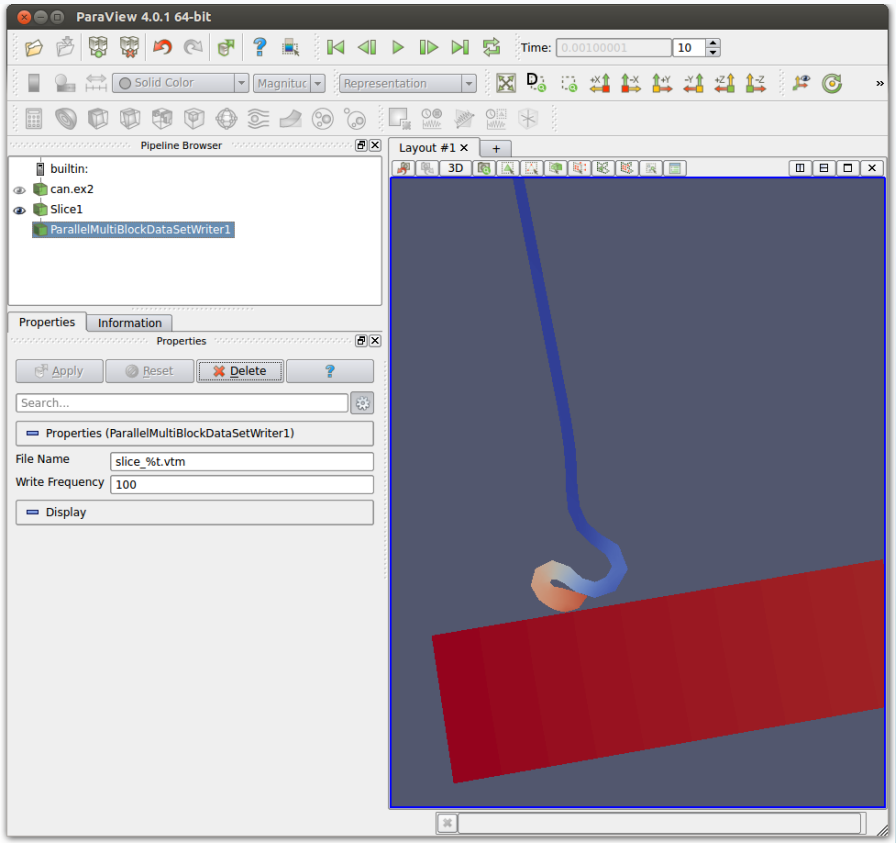


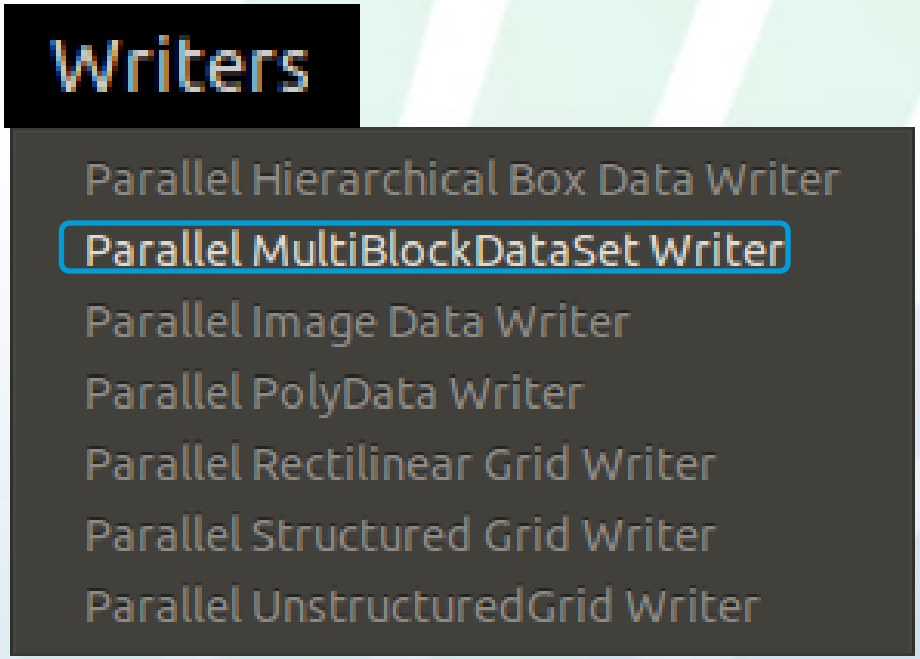


Kitware

In Situ Demo – Adding in Writers

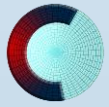
- Parameters:
  - File Name – %t gets replaced with time step
  - Write Frequency





Kitware

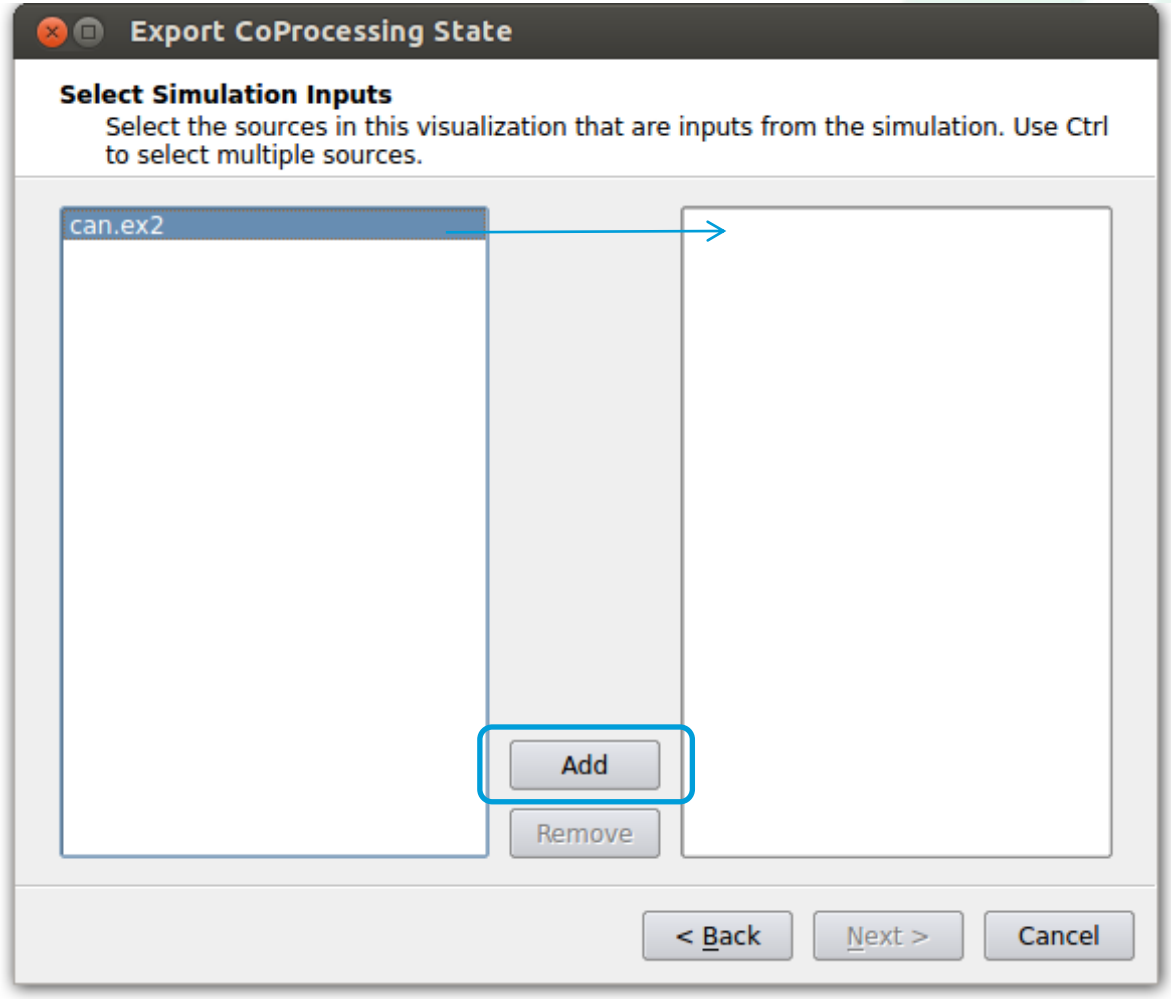
In Situ Demo – Exporting the Script

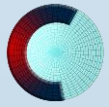


Kitware

In Situ Demo – Select Inputs

- Usually only a single input but can have multiple inputs

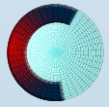




Kitware

In Situ Demo – Select Inputs

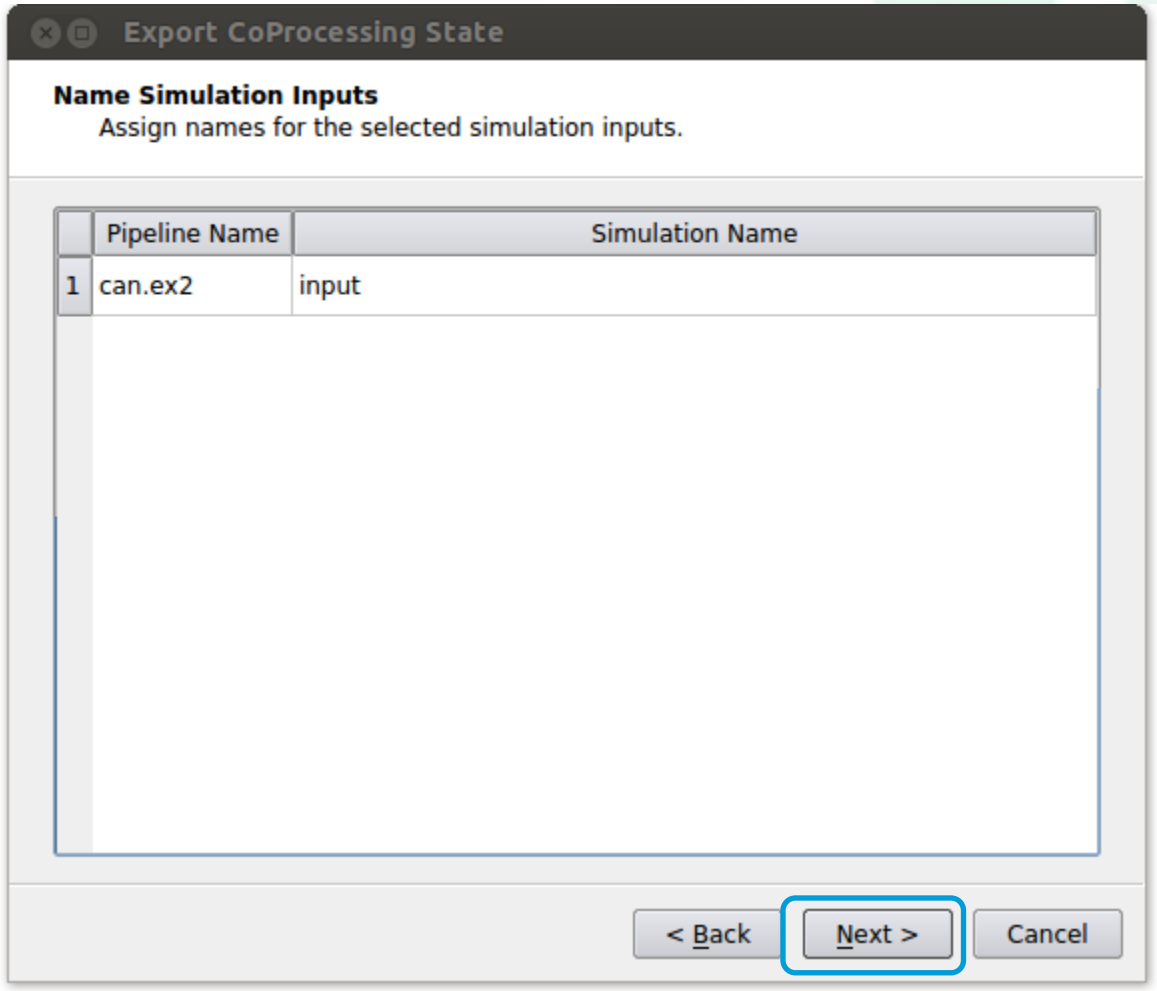
- Each pipeline source is a potential input

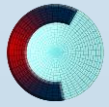


Kitware

In Situ Demo – Match Up Inputs

- Source name (e.g. “can.ex2”) needs to be matched with string key in adaptor (e.g. “input”)

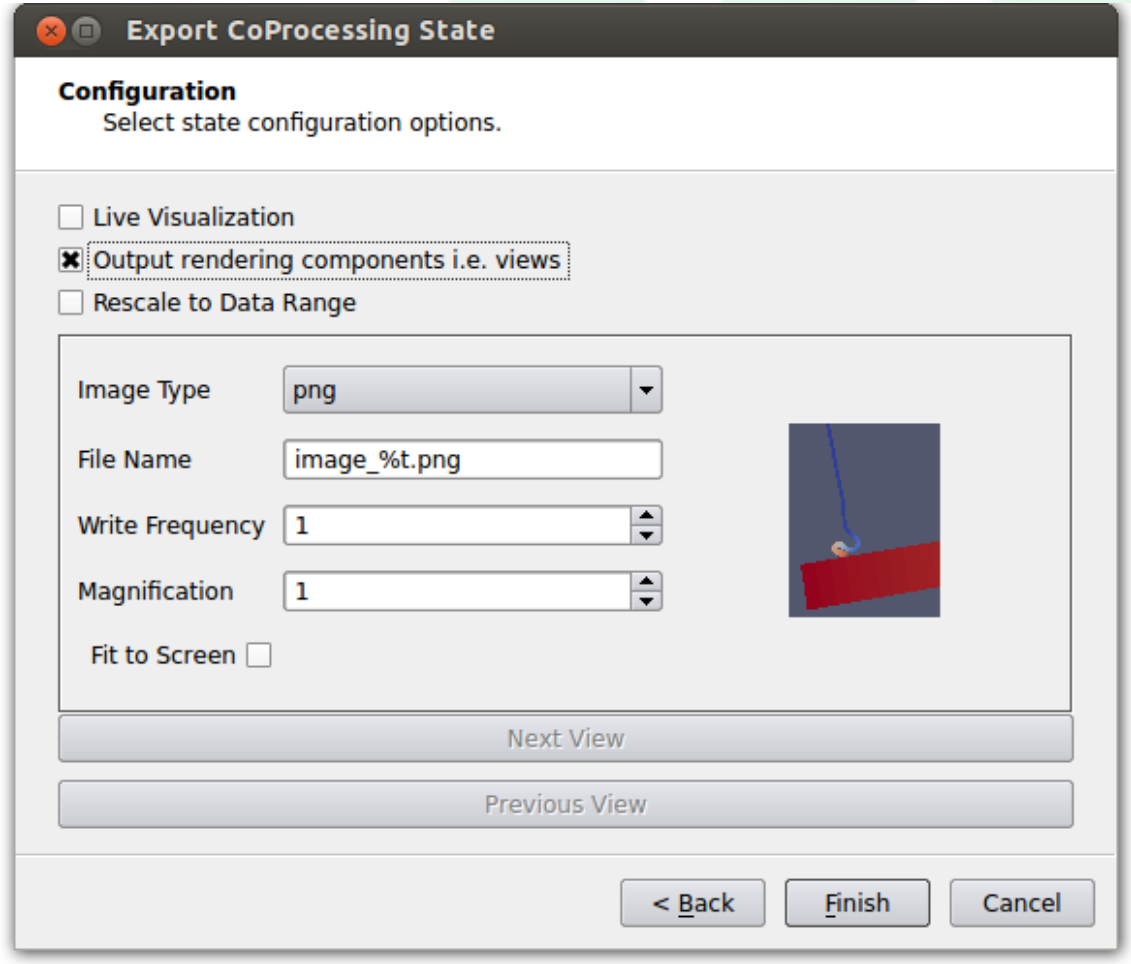


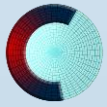


Kitware

In Situ Demo – Generating an Image

- Parameters/Options:
  - Live visualization
  - Rescale to Data Range (all images)
  - Individual images
    - Image Type
    - File Name
    - Write Frequency
    - Magnification
    - Fit to Screen
- %t gets replaced with time step

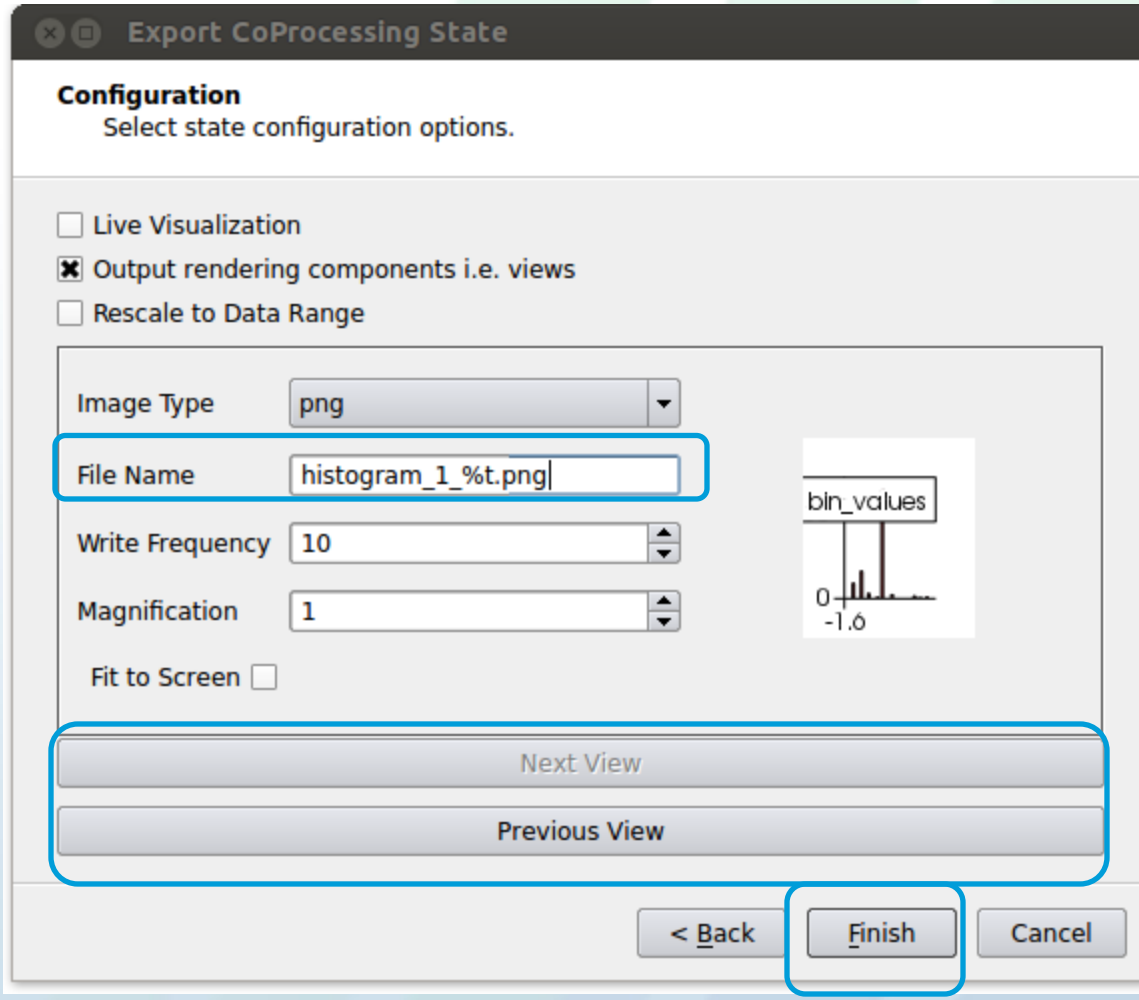


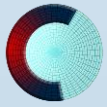


Kitware

In Situ Demo – Generating Two Images

- Parameters/Options:
  - Live visualization
  - Rescale to Data Range (all images)
  - Individual images
    - Image Type
    - File Name
    - Write Frequency
    - Magnification
    - Fit to Screen





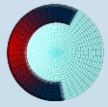
Kitware

In Situ Demo – Write Out the Script

- Generated script will look something like this

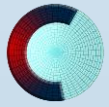
```
def DoCoProcessing(datadescription):

  input = CreateProducer( datadescription, "input" )

  ParallelMultiBlockDataSetWriter1 = CreateWriter( XMLMultiBlockDataWriter, "filename_%t.vtm", 1 )
```

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

cp_writers = []

def RequestDataDescription(datadescription):
    "Callback to populate the request for current timestep"
    timestep = datadescription.GetTimeStep()


    input_name = 'input'
    if (timestep % 1 == 0) :
        datadescription.GetInputDescriptionByName(input_name).AllFieldsOn()
        datadescription.GetInputDescriptionByName(input_name).GenerateMeshOn()
    else:
        datadescription.GetInputDescriptionByName(input_name).AllFieldsOff()
        datadescription.GetInputDescriptionByName(input_name).GenerateMeshOff()


def DoCoProcessing(datadescription):
    "Callback to do co-processing for current timestep"
    global cp_writers
    cp_writers = []
    timestep = datadescription.GetTimeStep()

    input = CreateProducer( datadescription, "input" )

    ParallelMultiBlockDataSetWriter1 = CreateWriter( XMLMultiBlockDataWriter, "filename_%t.vtm", 1 )



    for writer in cp_writers:
        if timestep % writer.cpFrequency == 0:
            writer.FileName = writer.cpFileName.replace("%t", str(timestep))
                                ine()


                                nager.GetRenderViews()

                                renderviews)):
                                ame.replace("%v", str(view))
                                ace("%t", str(timestep))
                                renderviews[view])

                                oxies -- we do it this way to avoid problems with prototypes
                                Delete()
                                :

                                esToDelete()


    iter = servermanager.vtkSMProxyIterator()
    iter.Begin()
    tobedeleted = []
    while not iter.IsAtEnd():
      if iter.GetGroup().find("prototypes") != -1:
         iter.Next()
         continue
      proxy = servermanager._getPyProxy(iter.GetProxy())
      proxygroup = iter.GetGroup()
      iter.Next()
      if proxygroup != 'timekeeper' and proxy != None and proxygroup.find("pq_helper_proxies") == -1 :
          tobedeleted.append(proxy)

    return tobedeleted

def CreateProducer(datadescription, gridname):
  "Creates a producer proxy for the grid"
  if not datadescription.GetInputDescriptionByName(gridname):
    raise RuntimeError, "Simulation input name '%s' does not exist" % gridname
  grid = datadescription.GetInputDescriptionByName(gridname).GetGrid()
  producer = TrivialProducer()
  producer.GetClientSideObject().SetOutput(grid)
  producer.UpdatePipeline()
  return producer


def CreateWriter(proxy_ctor, filename, freq):
    global cp_writers
    writer = proxy_ctor()
    writer.FileName = filename
    writer.add_attribute("cpFrequency", freq)
    writer.add_attribute("cpFileName", filename)
    cp_writers.append(writer)
    return writer
```



In Situ Demo – Run the Script

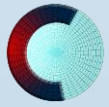
- Put candriver.py and the generated Python script in the same directory
- Linux and Mac from a terminal
  - <path>/pvpython candriver.py <generated script> <path>/can.ex2
- Windows from a command prompt
  - start “simple example” <path>/pvpython.exe candriver.py <generated script> <path>/can.ex2



Kitware

Live *In Situ* Analysis and Visualization

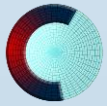
- Everything before this was “batch”
  - Preset information with possible logic
- “Beta” functionality for interacting with simulation data during simulation run
  - When exporting a Python script, select “Live Visualization”
  - During simulation run choose the “Tools->Connect to Catalyst” GUI menu item



Kitware

Live *In Situ* Example

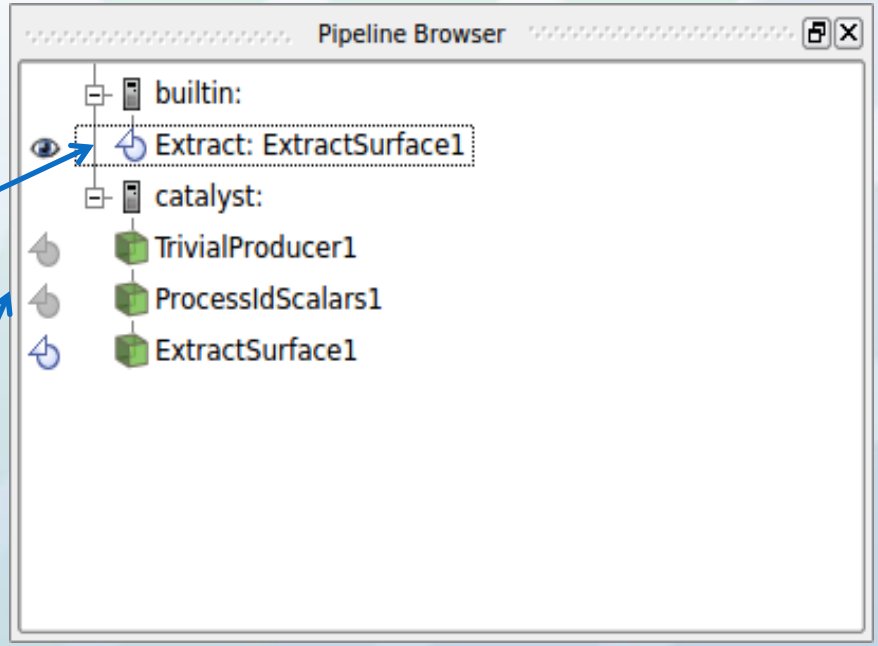
- Linux and Mac from a terminal
  - <path>/pvpython livecoprocessing.py <path>/can.ex2
- Windows from a command prompt
  - start “simple example” <path>/pvpython.exe livecoprocessing.py <path>/can.ex2
- Start ParaView and select Tools→Connect to Catalyst
  - Select port (22222 is default)

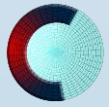


Live *In Situ* Example

- Only transfer requested data from server (simulation run) to client
  - ExtractSurface1 is already getting extracted
- Use 

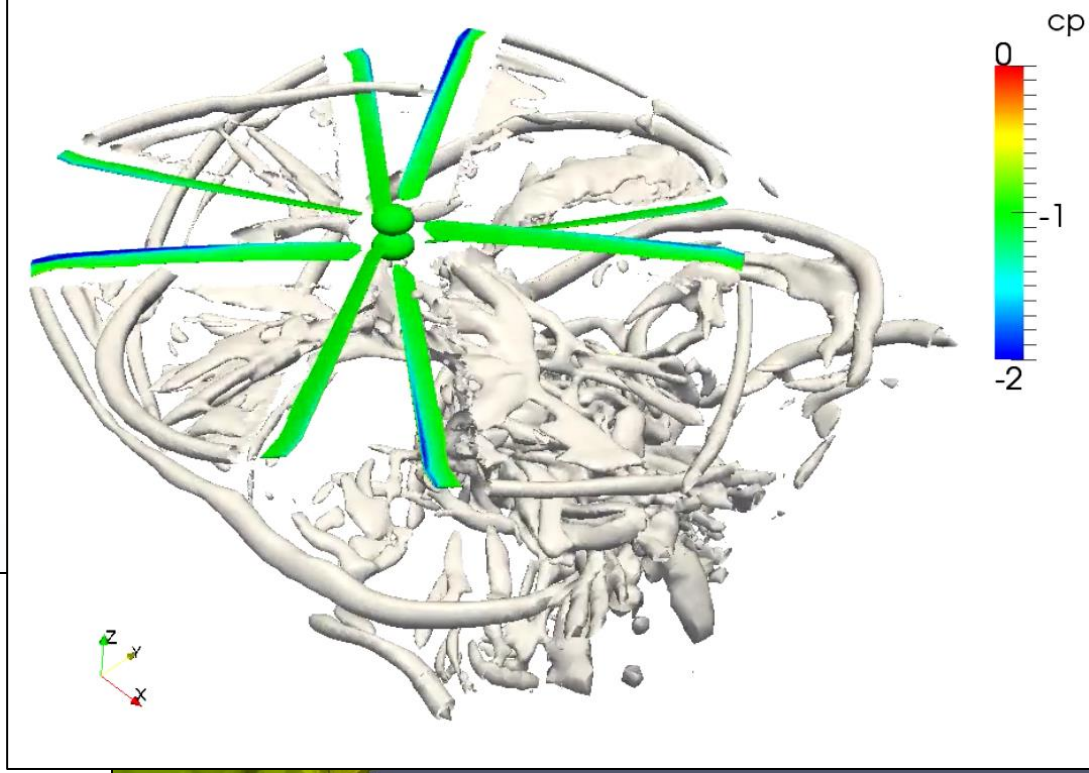
on client to stop transferring to client
- Click on  to transfer to client from Catalyst

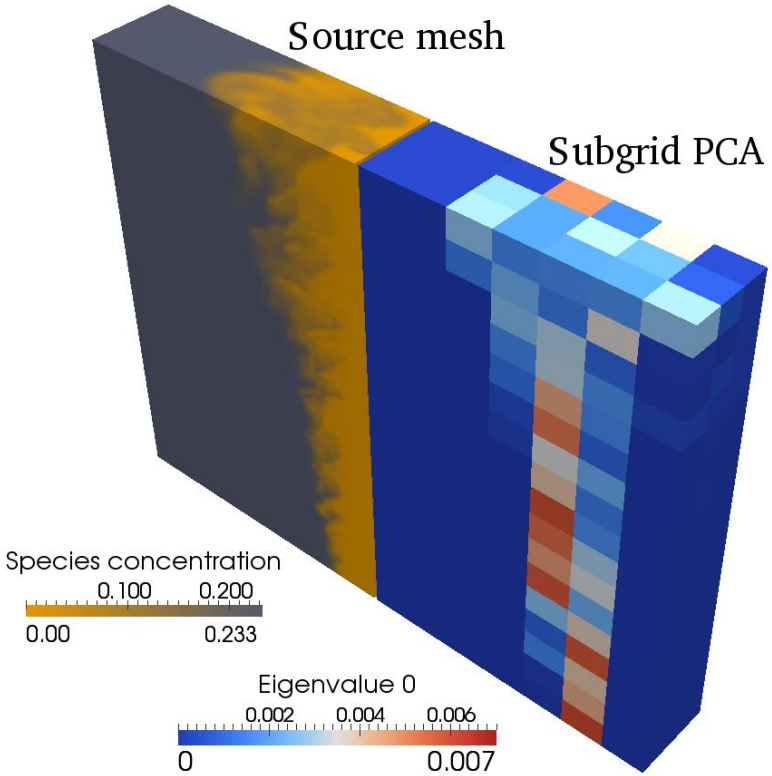


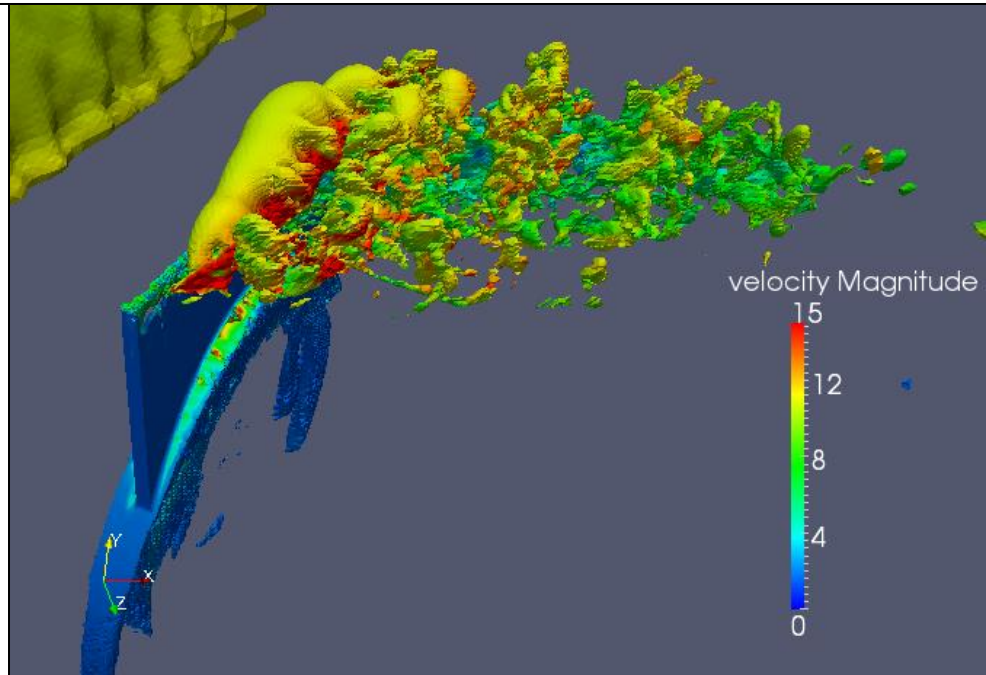


Kitware

Gratuitous Catalyst Images





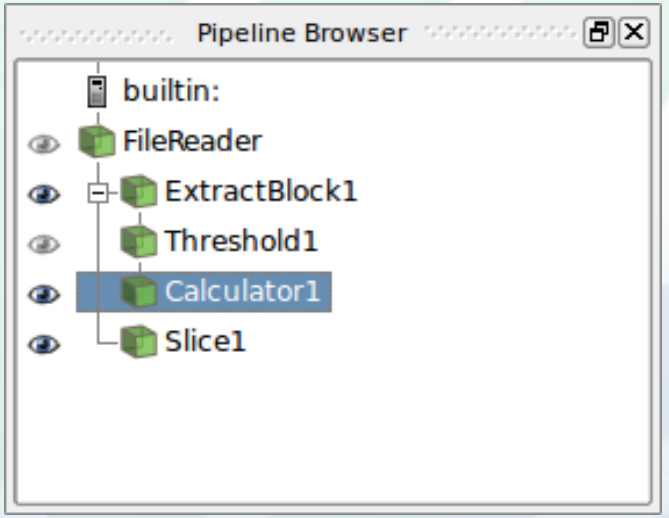


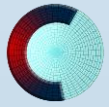
Kitware

ParaView Catalyst for Developers

ParaView Pipeline Concept

- Fundamental concept in ParaView
- Directed acyclic graph specifying how to process information
- Filters are nodes in the graph
  - Perform a certain action on a data set (grid and fields)
    - Contours, streamlines, file IO, etc.
  - Do not modify input data set
- Catalyst executes user pipelines at specified times

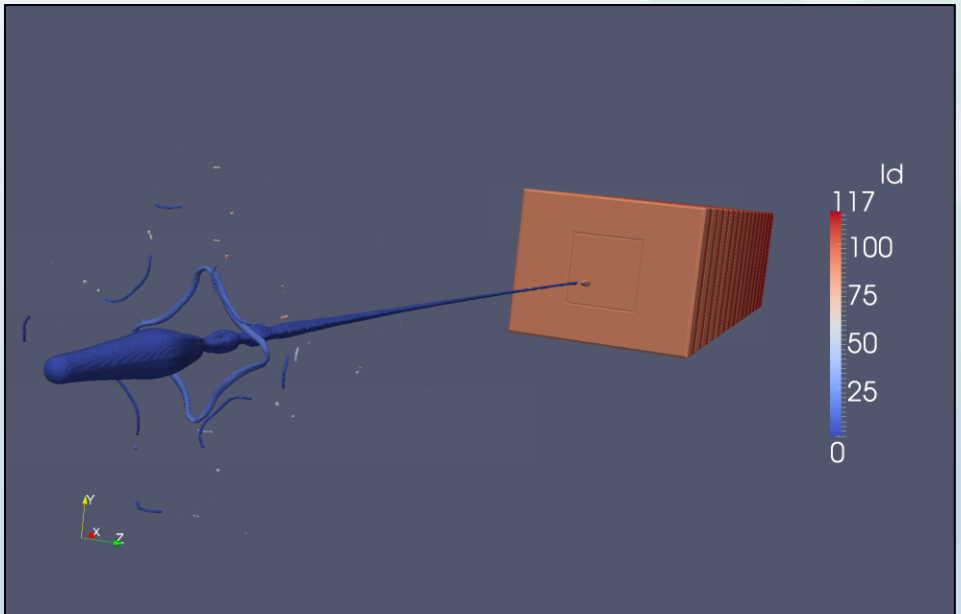


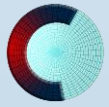


Kitware

Catalyst Pipelines

- User generated Python scripts from the ParaView plugin
  - Executed with vtkCPPythonScriptPipeline
- Hard-coded pipelines (“canned” output)
  - Executed with a class that derives from vtkCPPipeline



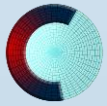


Kitware

vtkCPPythonScriptPipeline

- Typically from ParaView script generator plugin
- Initialize from a path to a ParaView Catalyst Python script
  - vtkCPPythonScriptPipeline::Initialize(const char* fileName)

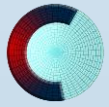
http://www.paraview.org/ParaView3/Doc/Nightly/html/classvtkCPPythonScriptPipeline.html



Kitware

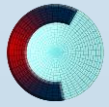
vtkCPPythonScriptPipeline (2)

- Advantages
  - Easily created through ParaView plugin
    - Output should be moderately readable
  - Encourage advanced users to modify
    - Can use ParaView's Python trace utility to see options
  - View/screenshot settings can be difficult to set
    - Camera angle, zoom, lighting, data representations, etc.
  - Takes care of parallel image compositing
  - Can modify without recompiling



vtkCPPythonScriptPipeline (3)

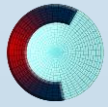
- Disadvantages
  - Slight overhead compared to C++ hard-coded pipeline
    - Roughly 10^{-5} seconds per time step
  - Simulation code must be linked with Python
    - Static build issues
    - More complex to minimize executable size



Hard-coded Pipelines

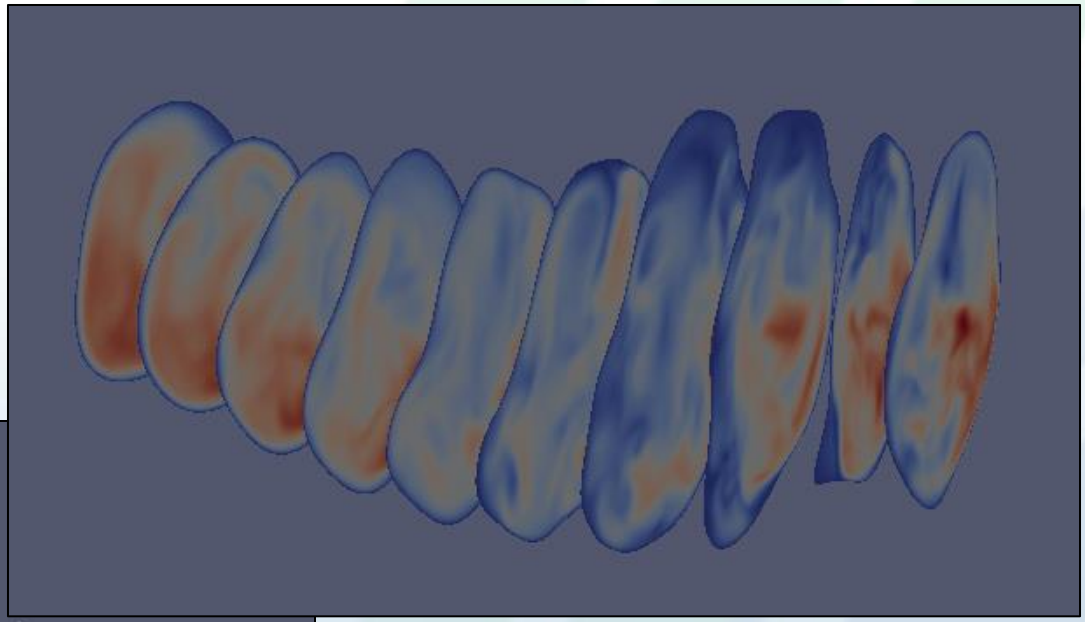
- Derives from vtkCPPipeline
- Generally C++ but could be Python code
- Most are done directly creating VTK filters and connecting them together
  - Creating screen shots from rendering pipeline with compositing can be daunting
- Possible to do using ParaView C++ proxies
  - Low level access is more complex
  - Simpler set up for rendering pipeline
- Less dependencies for compiling and linking

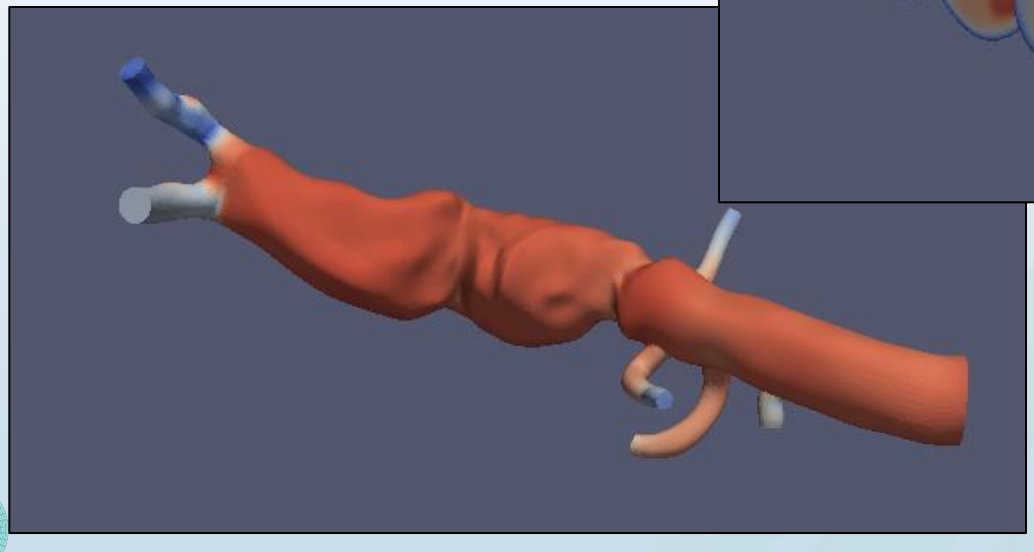
http://www.paraview.org/ParaView3/Doc/Nightly/html/classvtkCPPipeline.html

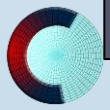


Catalyst

- Catalyst's job is to create and execute pipelines



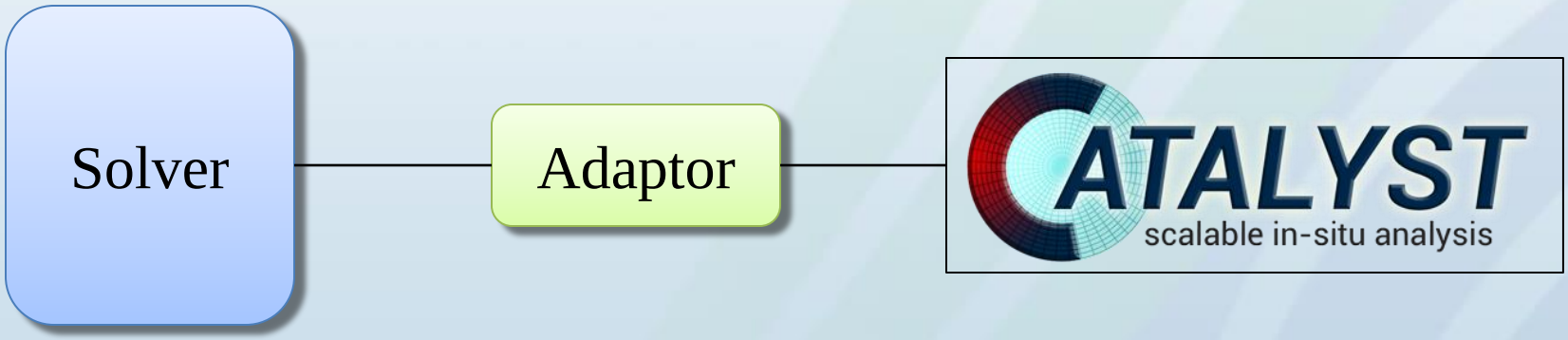


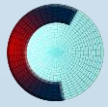


Kitware

Data Structures

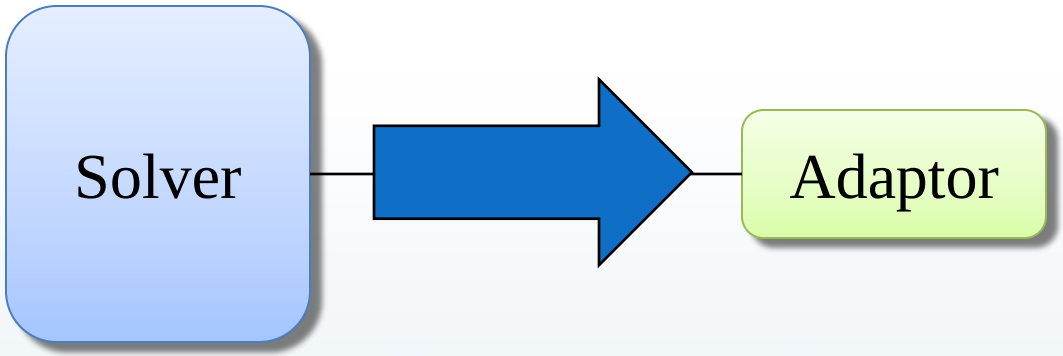
- Simulation has separate data structures from VTK data structures
- User an adaptor to bridge the gap
  - Try to reuse existing memory
  - Also responsible for other interactions between simulation code and Catalyst



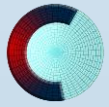


Kitware

Information Flow

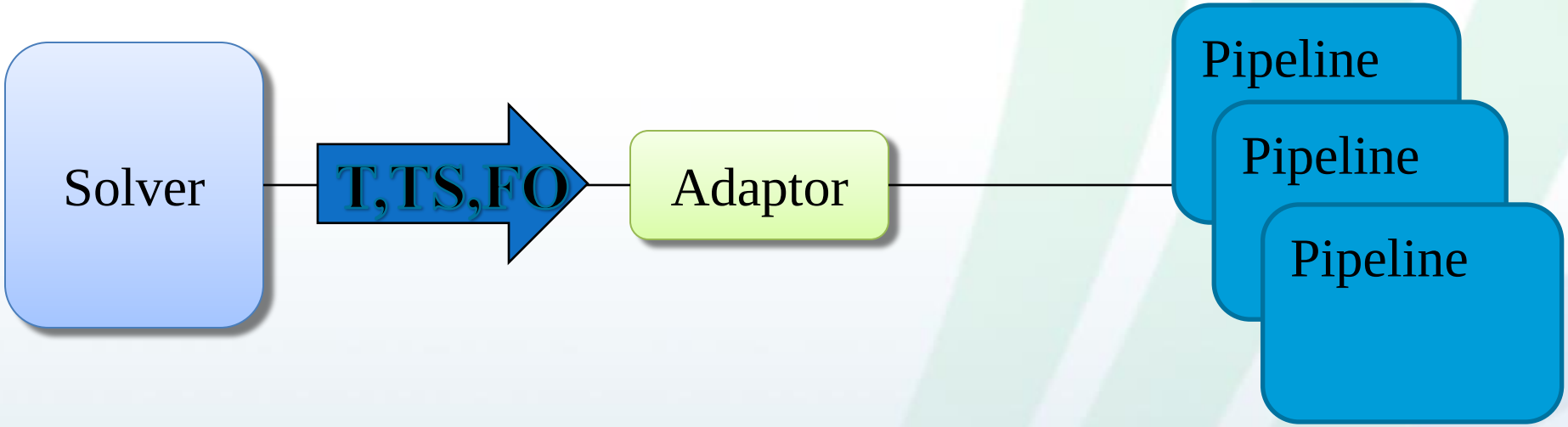


- Initialization
  - Information for creating pipelines

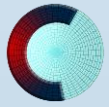


Kitware

Information Flow

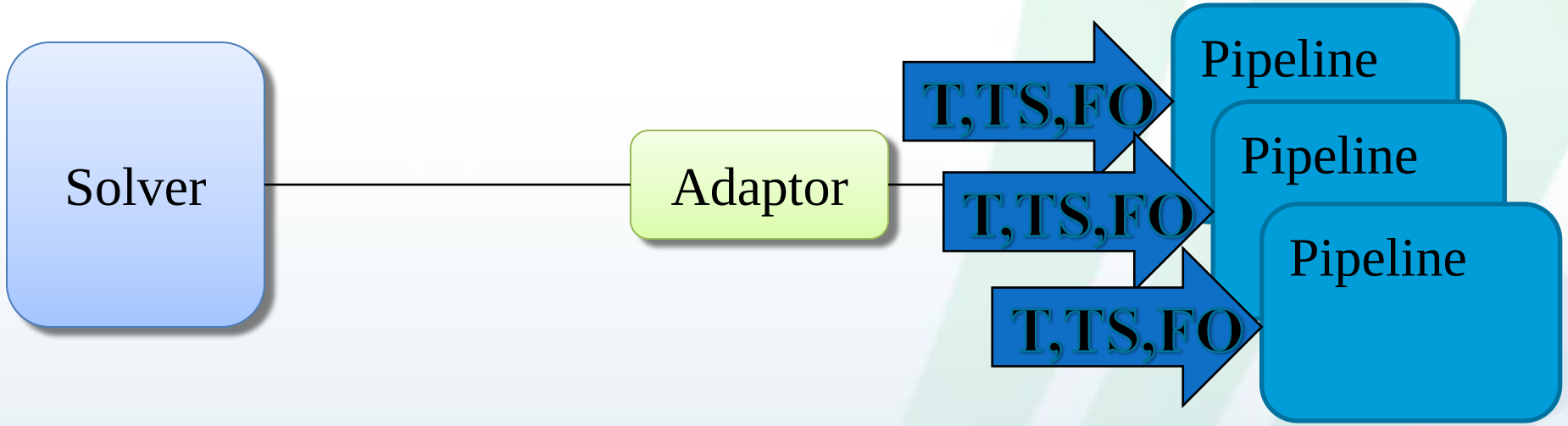


- After simulation completes time step update
  - Time, time step, force output flag
  - Information for creating grid and field information



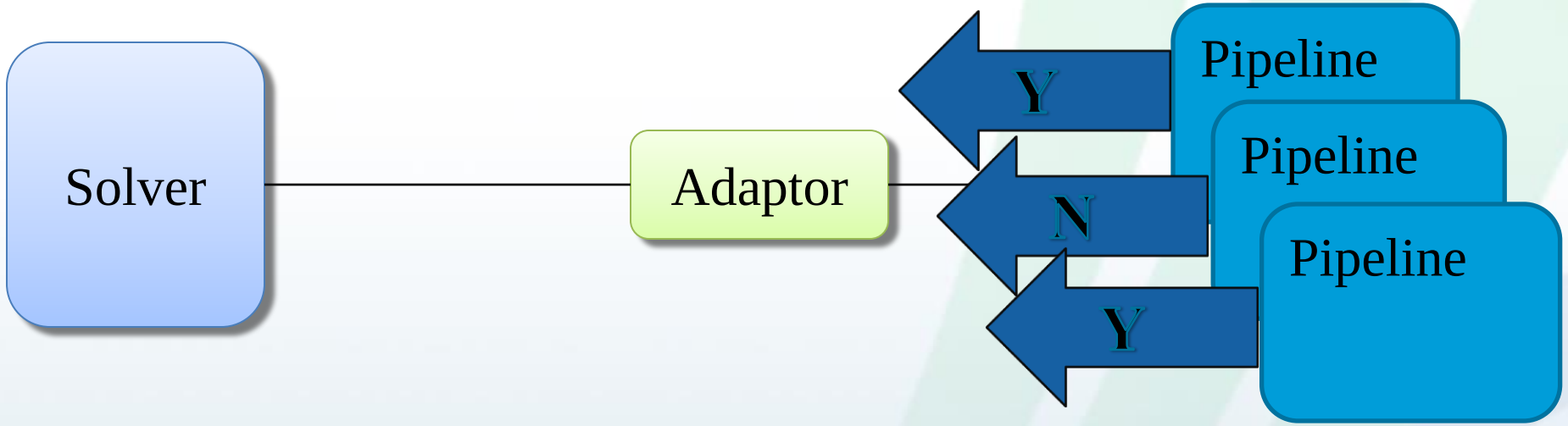
Kitware

Information Flow

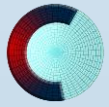


- After simulation completes time step update
  - Time, time step, force output flag passed to each pipeline

Information Flow

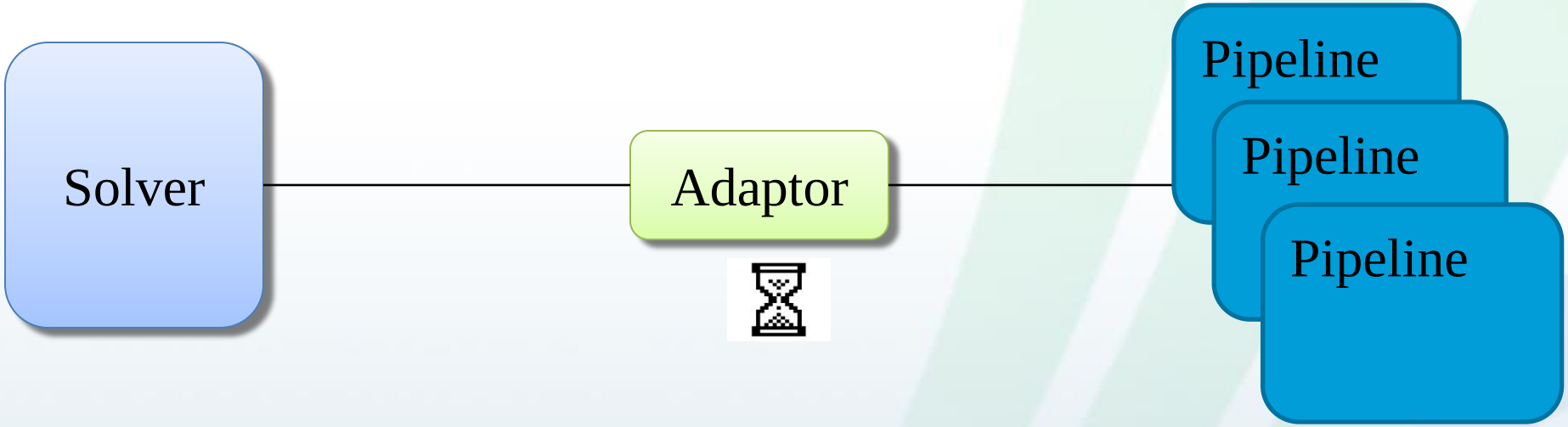


- After simulation completes time step update
  - Flag indicating which pipelines need to be executed/updated

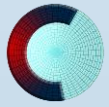


Kitware

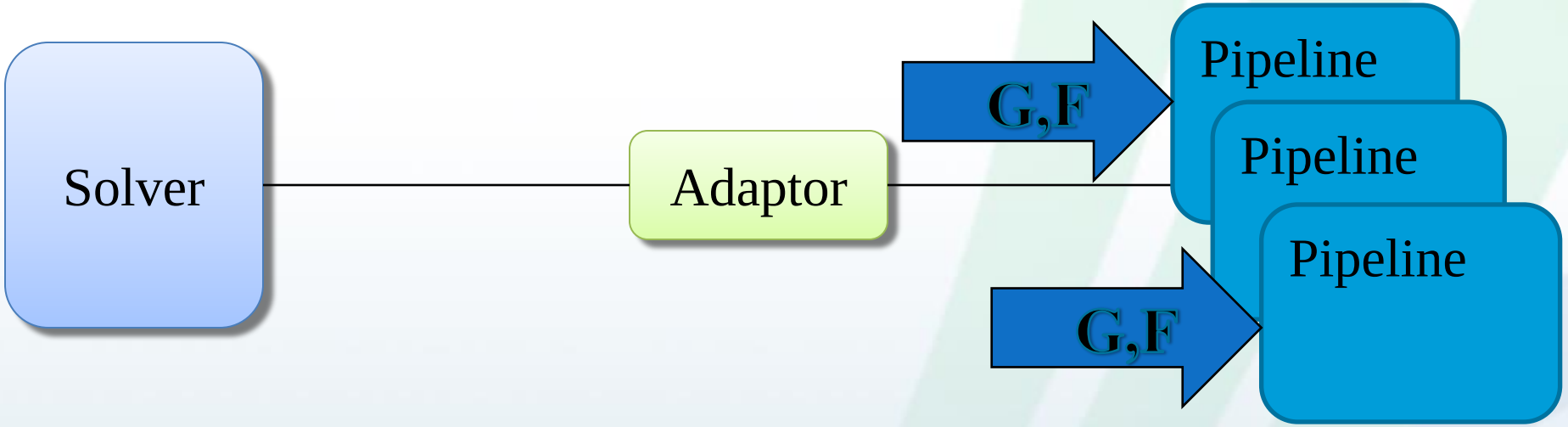
Information Flow



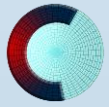
- After simulation completes time step update
  - If any pipeline needs to be executed
    - Adaptor creates VTK objects to represent grids and fields



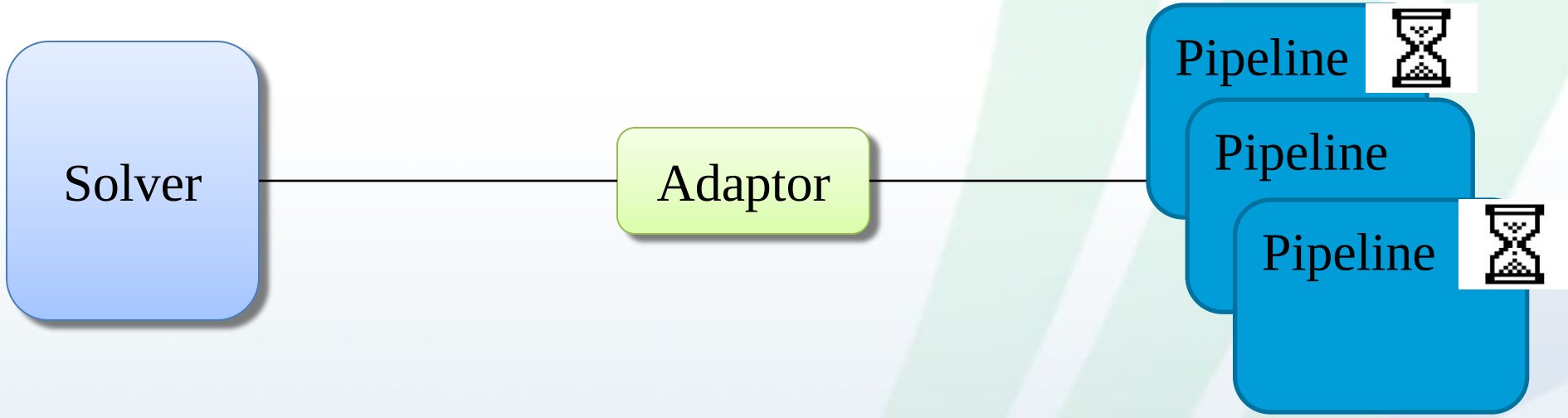
Information Flow



- After simulation completes time step update
  - Pass VTK data object representing grids and fields to pipelines that need to execute/update

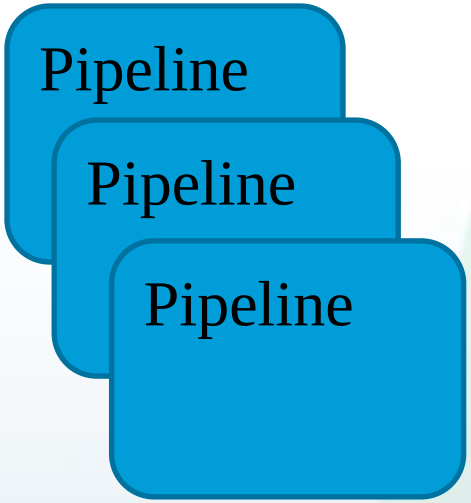


Information Flow

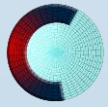


- After simulation completes time step update
  - Pipelines execute and output desired information

What is Catalyst?



- Communication/synchronization routines
- Pipeline mechanics
- Data processing through filters
- Writers
- Compositing and rendering

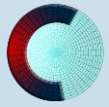


Kitware

Interfacing with Catalyst

Adaptor

- Catalyst calls should have minimal footprint in simulation code
  - Initialization call
  - Co-processing call
  - Finalize call
- VTK data structures usually not appropriate for simulation code data structures
- Adaptor's job is to provide interface between simulation code and Catalyst

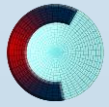


Kitware

Adaptor Architectural View

```
Adaptor:
vtkCPProcessor* Processor;
vtkDataObject* CreateGrid();
vtkCPPipeline* CreatePipeline();
```

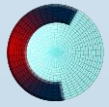
- Initialization
  - Creates vtkCPProcessor and initializes it
    - vtkCPProcessor manages pipelines
    - vtkCPProcessor::Initialize()
  - Creates vtkCPPipeline objects and adds them to vtkCPProcessor
    - vtkCPProcessor::AddPipeline(vtkCPPipeline*)



Adaptor Architectural View

```
Adaptor:
vtkCPProcessor* Processor;
vtkDataObject* CreateGrid();
vtkCPPipeline* CreatePipeline();
```

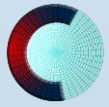
- When called after update time step is completed
  - Creates a vtkCPDataDescription
    - Information needed by Catalyst to determine what pipelines need to execute
    - vtkCPDataDescription::SetTimeData(double time, vtkIdType timeStep)
    - vtkCPDataDescription::SetForceOutput(bool) *optional*
  - Queries pipelines to determine if co-processing is necessary
    - Processor->RequestDataDescription(vtkCPDataDescription*)
    - Returns 0 if no co-processing is needed that time step



Adaptor Architectural View

Adaptor:
vtkCPProcessor* Processor;
vtkDataObject* CreateGrid();
vtkCPPipeline* CreatePipeline();

- If Processor->RequestDataDescription() != 0
  - CreateGrid()
    - Creates grids and fields
    - Biggest part of the adaptor
  - vtkCPDataDescription::AddInput(const char* name)
    - Convention is that name := “input” for a single input
  - vtkCPDataDescription::GetInputDataDescriptionByName(const char* name)->SetGrid(CreateGrid())
  - Processor->CoProcess(vtkCPDataDescription*)

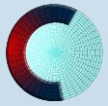


Kitware

Adaptor Architectural View

```
Adaptor:
vtkCPProcessor* Processor;
vtkDataObject* CreateGrid();
vtkCPPipeline* CreatePipeline();
```

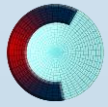
- Finalize
  - Processor->Finalize()



Kitware

Main Co-Processing Parts

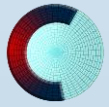
- Initialize
- Check with each Catalyst pipeline if execution is necessary
  - Input is simulation time and time step
  - Output is if co-processing is needed along with needed grids and fields
- If co-processing is needed at specified time and time step:
  - Input the grid and field information
  - Execute the proper Catalyst pipelines
- Finalize



Kitware

C++ Example

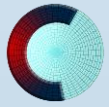
```
int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);
  std::string cpPythonFile = argv[1];
  int nSteps = atoi(argv[2]);
  vtkCPProcessor* processor = vtkCPProcessor::New();
  processor->Initialize();
  vtkCPPythonScriptPipeline* pipeline =
    vtkCPPythonScriptPipeline::New();

  // read the coprocessing python file
  if(pipeline->Initialize(cpPythonFile.c_str()) == 0) {
    cout << "Problem reading the python script.\n";
    return 1;
    }
  processor->AddPipeline(pipeline);
  pipeline->Delete();

  if (nSteps == 0) {
    return 0;
    }
...
```



Kitware

C++ Example (2)

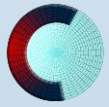
```
...
  double tStart = 0.0;
  double tEnd = 1.0;
  double stepSize = (tEnd - tStart)/nSteps;

  vtkCPDataDescription* dataDesc = vtkCPDataDescription::New();
  dataDesc->AddInput("input");

  for (int i = 0; i < nSteps; ++i)
    {
    double currentTime = tStart + stepSize*i;
    // set the current time and time step
    dataDesc->SetTimeData(currentTime, i);

    // check if the script says we should do coprocessing now
    if(processor->RequestDataDescription(dataDesc) != 0)
      {
      // create our vtkDataObject with the grids and fields
      vtkDataObject* dataObject =  <generate grid>;
      dataDesc->GetInputDescriptionByName("input")->SetGrid(dataObject);
      processor->CoProcess(dataDesc);
      }
    }

  dataDesc->Delete();
  processor->Finalize();
  processor->Delete();

  MPI_Finalize();

  return 0;
}
```



Creating VTK Objects

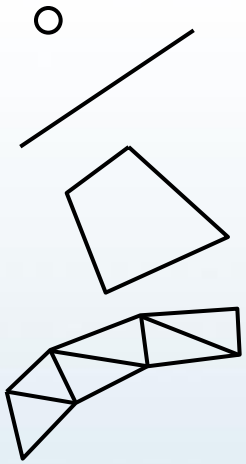
Getting Data Into Catalyst

- Main object will derive from vtkDataObject
  - Grids that derive from vtkDataSet
  - Multiblock grids that contain multiple vtkDataSets
- Field (aka attribute) information
  - Point data – information specified for each point in a grid
  - Cell data – information specified for each cell in a grid
  - Field data – meta-data not associated with either points or cells
- All object groups are 0-based/C/C++ indexing

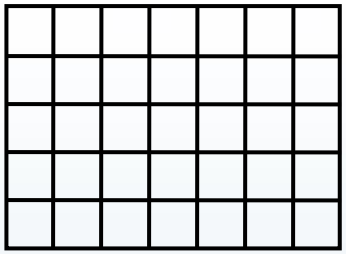


vtkDataSet Subclasses

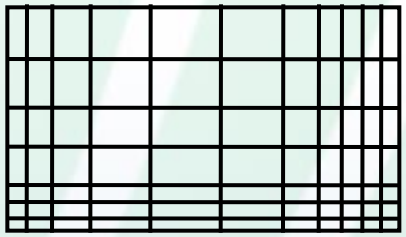
vtkPolyData



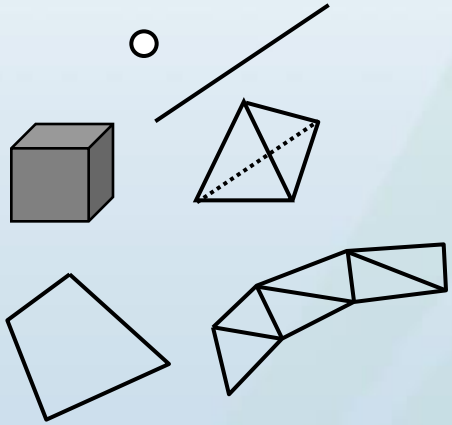
vtkImageData
vtkUniformGrid



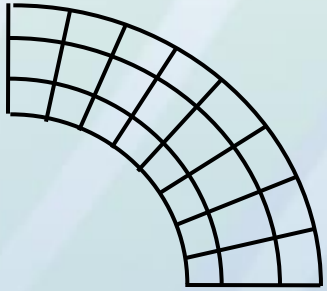
vtkRectilinearGrid



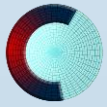
vtkUnstructuredGrid



vtkStructuredGrid

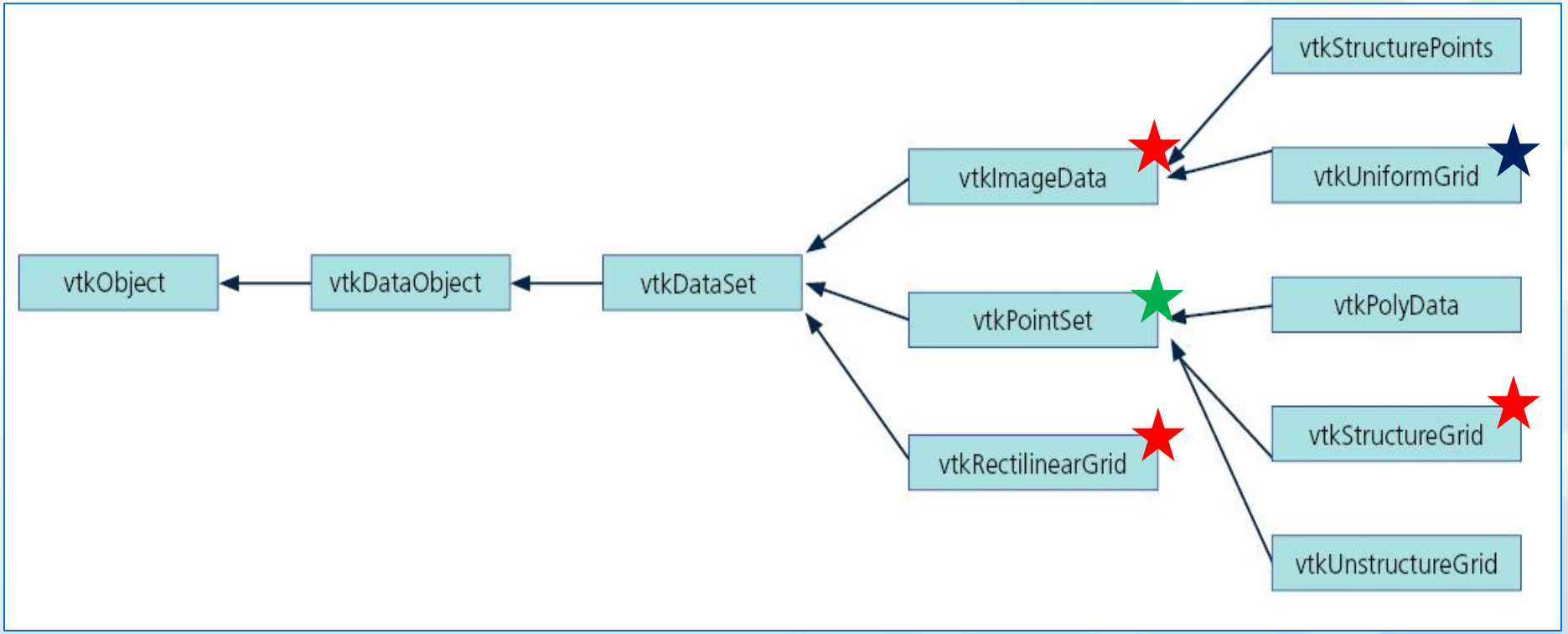


http://www.vtk.org/doc/nightly/html/classvtkDataSet.html



Kitware

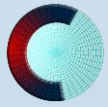
vtkDataSet Class Hierarchy



★ Topologically regular grid

★ Irregular geometry

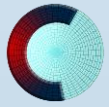
★ Supports blanking



Kitware

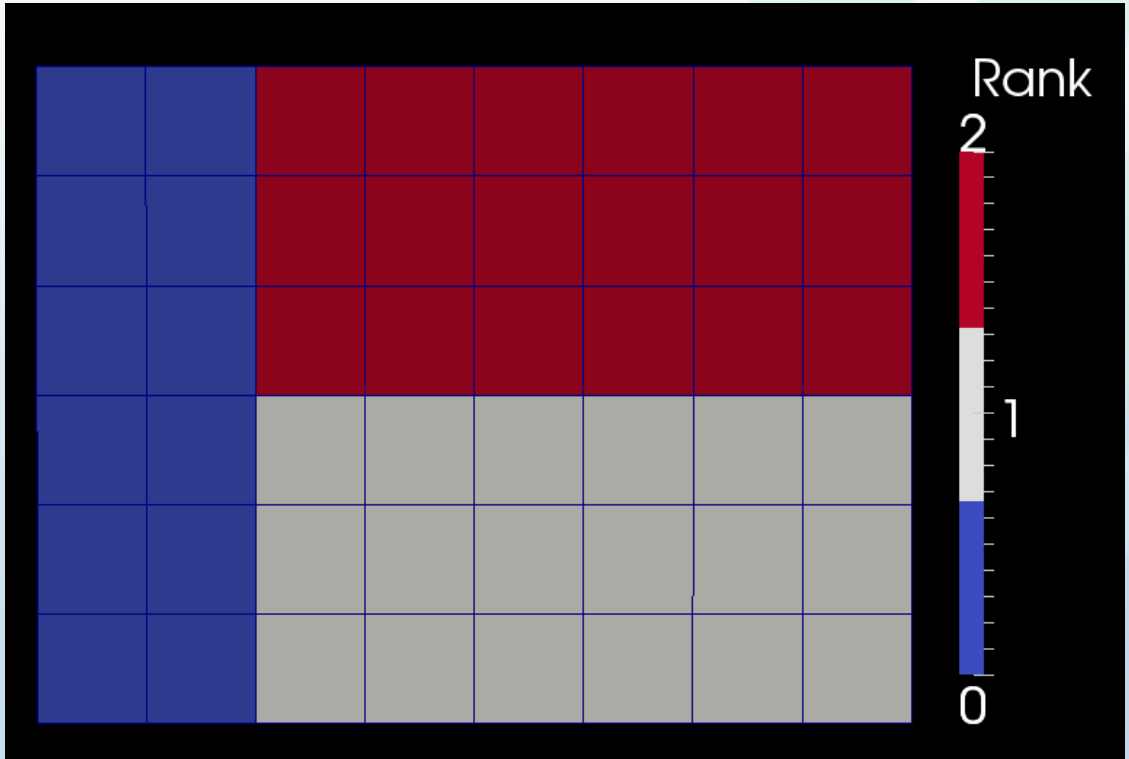
Topologically Regular Grids

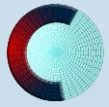
- vtkImageData/vtkUnstructuredGrid, vtkRectilinearGrid and vtkStructuredGrid
- Topological structure defined by whole extent
  - Gives first and last point in each logical direction
  - Not required to start at 0
- Partitioning defined by extents
  - First and last point in each logical direction of part of the entire grid
- Ordering of points and cells is fastest in logical x-direction, then y-direction and slowest in z-direction



Extents

- Whole extent for all processes (0, 8, 0, 6, 0, 0)
- Extents different for each process
  - Rank 0: (0, 2, 0, 6, 0, 0), 21 points, 12 cells
  - Rank 1: (2, 8, 0, 3, 0, 0), 28 points, 18 cells
  - Rank 2: (2, 8, 3, 6, 0, 0), 28 points, 18 cells

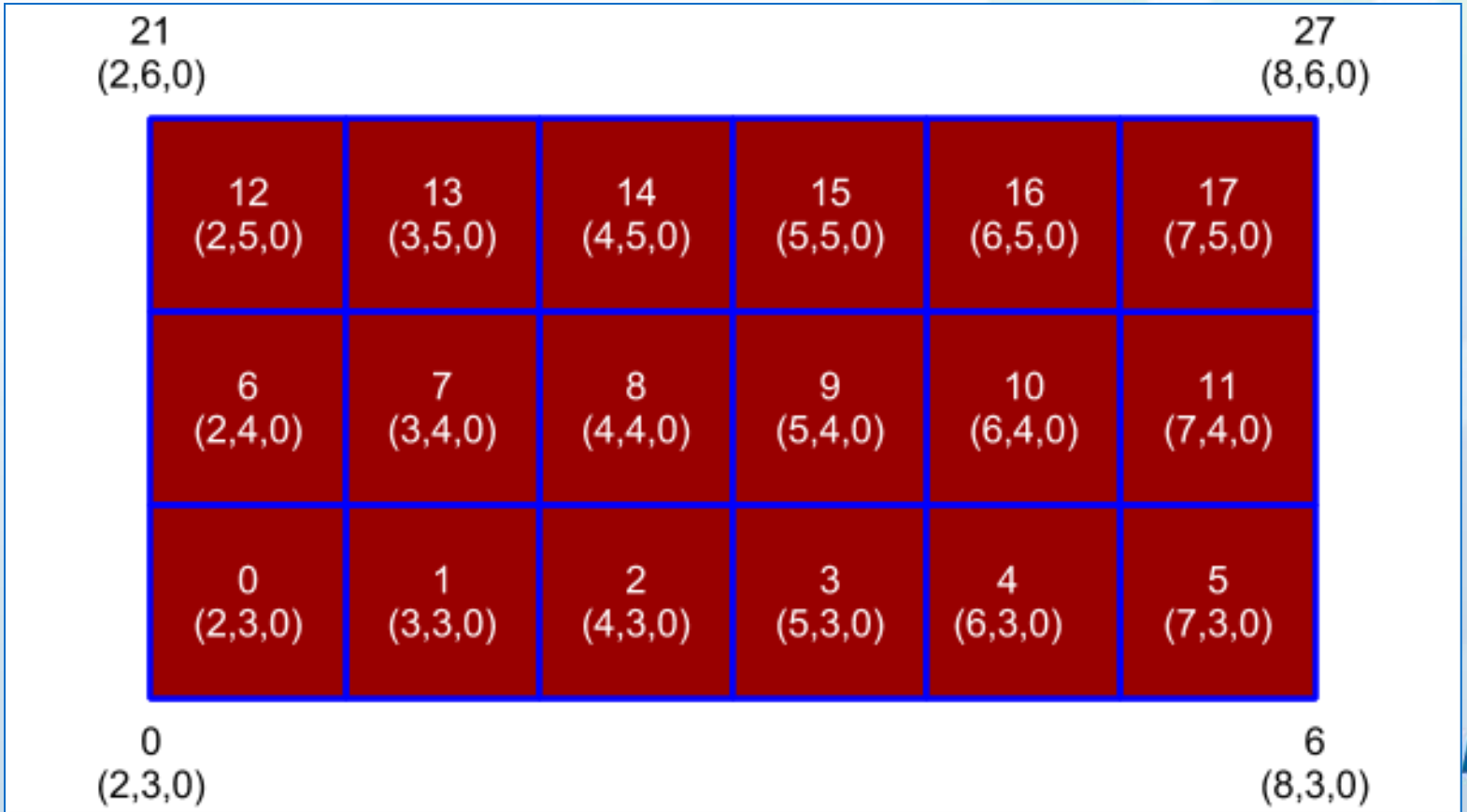


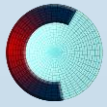


Kitware

Extents

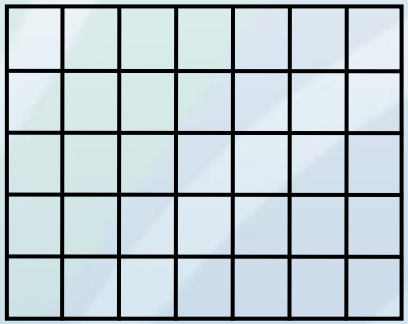
- Point and cell indices for extent of (3, 8, 3, 6, 0, 0)
  - From rank 2 in previous slide
  - 28 points and 18 cells



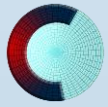


vtkImageData/vtkUniformGrid

- vtkCPInputDataDescription::SetWholeExtent() – total number of points in each direction
- SetExtent() – a process's part of the whole extent
- SetSpacing() – cell lengths in each direction
- SetOrigin() – location of point 0 (i=0, j=0, k=0)



- vtkUniformGrid
  - Supports cell blanking
  - Currently written to file as vtkImageData

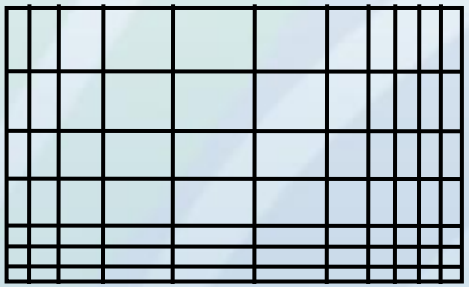


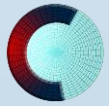
http://www.vtk.org/doc/nightly/html/classvtkUniformGrid.html

Kitware

vtkRectilinearGrid

- vtkCPInputDataDescription::SetWholeExtent() – total number of points in each direction
- SetExtents() – a process's part of the whole extent
- Set<X,Y,Z>Coordinates() – point coordinates in each direction
  - Only values for process's extents
  - Index starting at 0



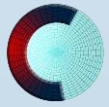


http://www.vtk.org/doc/nightly/html/classvtkRectilinearGrid.html

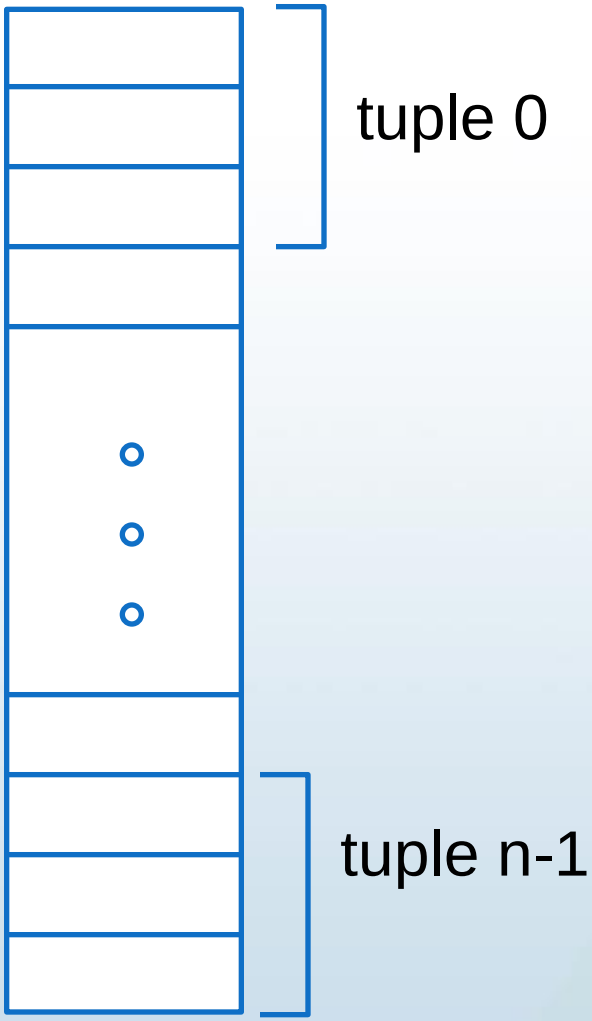
Kitware

Irregular Geometry

- Data sets that derive from vtkPointSet
  - vtkStructuredGrid – still topologically regular
  - vtkPolyData – topologically irregular with 0D, 1D and 2D cell types
  - vtkUnstructuredGrid – topologically irregular with 0D, 1D, 2D and 3D cell types
- Uses vtkPoints to specify grid's point locations
  - SetPoints(vtkPoints*) – set the number and coordinates of a process's points



vtkDataArray – Basis for vtkDataObject Contents



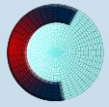
- An array of *n* tuples
- Each tuple has *m* components which are logically grouped together
- Internal implementation is a pointer to an n x m block of memory
- Data type determined by class

- Two APIs : generic and data type specific

http://www.vtk.org/doc/nightly/html/classvtkDataArray.html

vtkDataArray

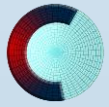
- SetNumberOfComponents() – call first
- SetNumberOfTuples()
  - For point data must be set to number of points
  - For cell data must be set to number of cells
- SetArray()
  - If flat array has proper component & tuple ordering use existing simulation memory – most efficient
  - Can specify who should delete
  - VTK uses pipeline architecture so Catalyst libraries will NOT modify array values
- SetTupleValue() – uses native data type
- SetValue() – uses native data type
- SetName() – array descriptor, e.g. velocity, density



http://www.vtk.org/doc/nightly/html/classvtkDataArray.html

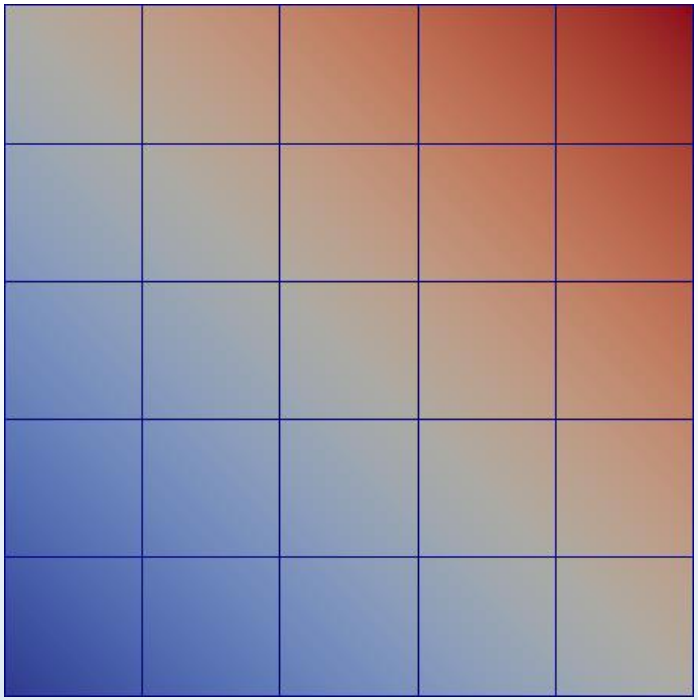
vtkFieldData

- Object for storing vtkDataArrays
- vtkDataSet::GetFieldData() – non-grid associated arrays
- Derived classes
  - vtkPointData – vtkDataSet::GetPointData()
  - vtkCellData – vtkDataSet::GetCellData()
- vtkFieldData::AddArray(vtkDataArray*)
- Specific arrays are normally retrieved by name from vtkFieldData
  - Uniqueness is not required but can cause unexpected results

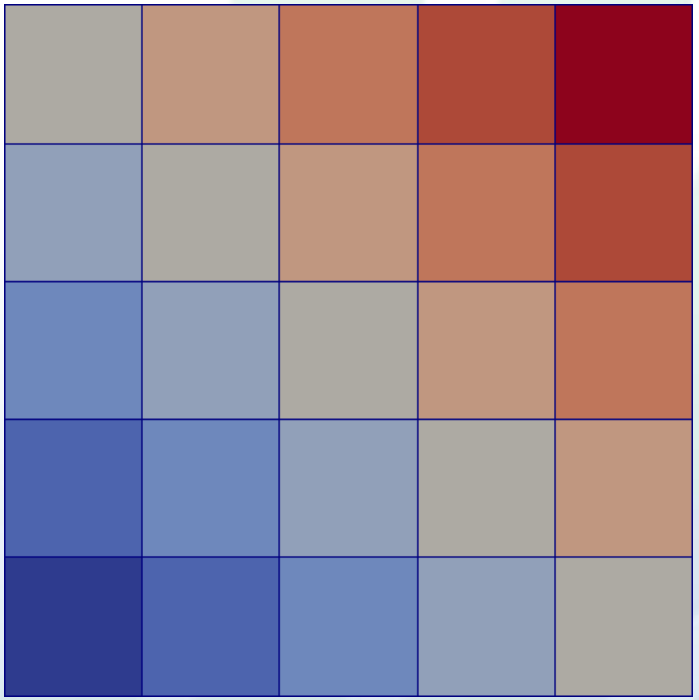


http://www.vtk.org/doc/nightly/html/classvtkFieldData.html

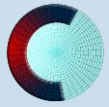
Point Data and Cell Data



Point data – 36 values

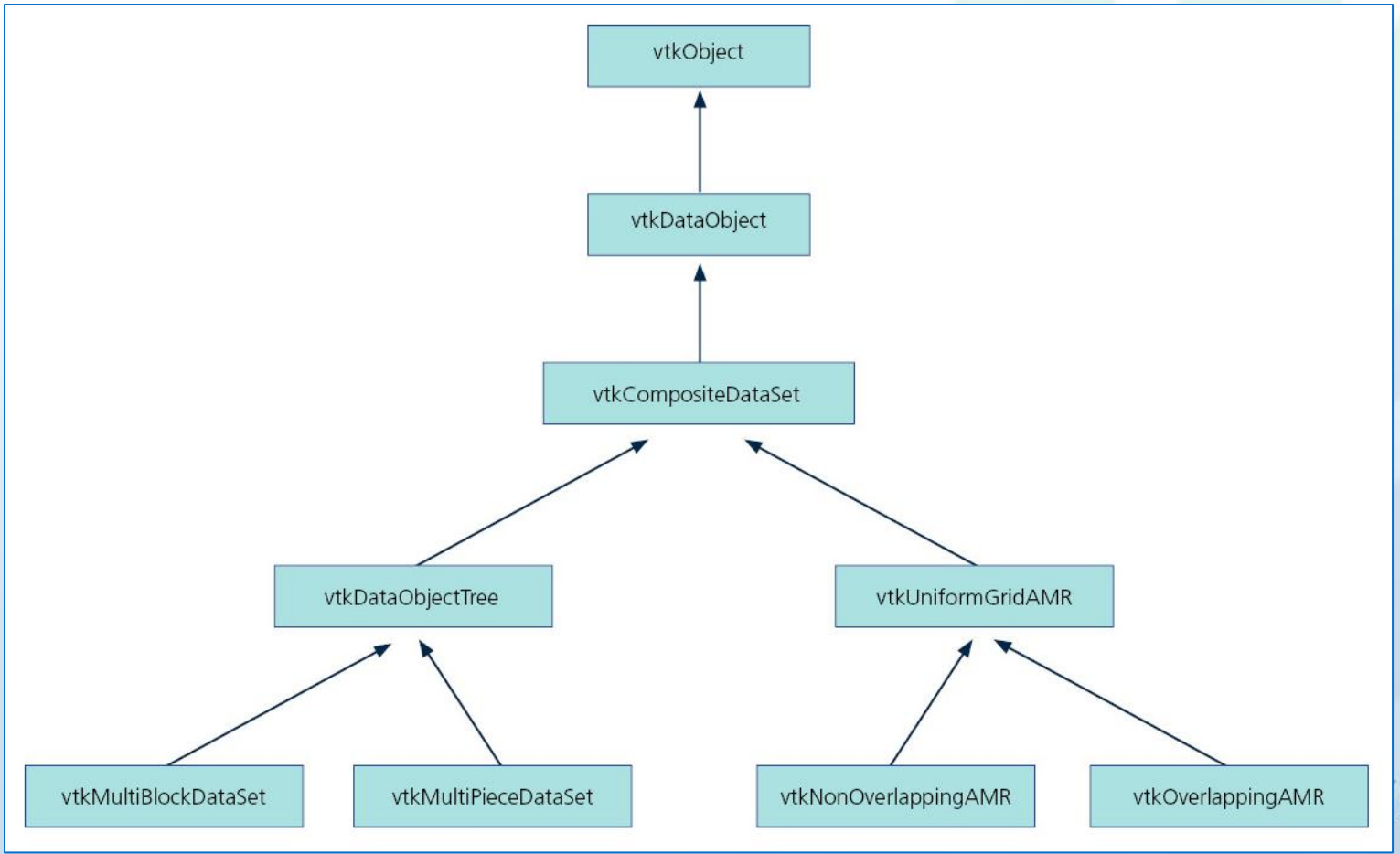


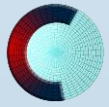
Cell data – 25 values



Multiple Grids

- Use something that derives from vtkCompositeDataSet to group multiple vtkDataSets together

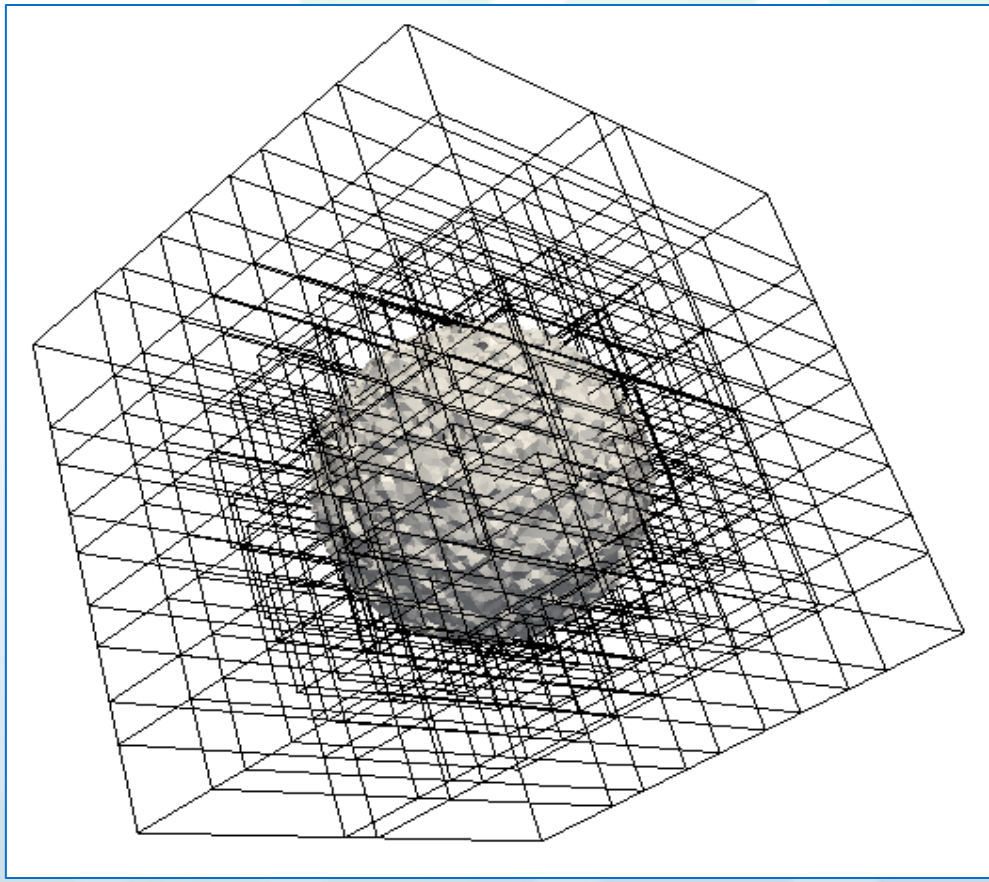


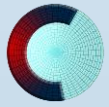


Kitware

vtkMultiBlockDataSet

- Most general way of storing vtkDataSets and other vtkCompositeDataSets
- Block structure must be the same on each process
- Block is null if data set is on a different process
- SetNumberOfBlocks()
- SetBlock()



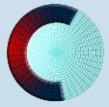


http://www.vtk.org/doc/nightly/html/classvtkMultiBlockDataSet.html

Kitware

vtkMultiPieceDataSet

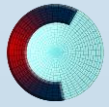
- Way to combine multiple vtkDataSets of the same type into a single logical object
  - Useful when the amount of pieces is unknown *a priori*
- Must be contained in a vtkMultiBlockDataSet
- SetNumberOfPieces()
- SetPiece()



http://www.vtk.org/doc/nightly/html/classvtkMultiPieceDataSet.html

AMR Data Sets

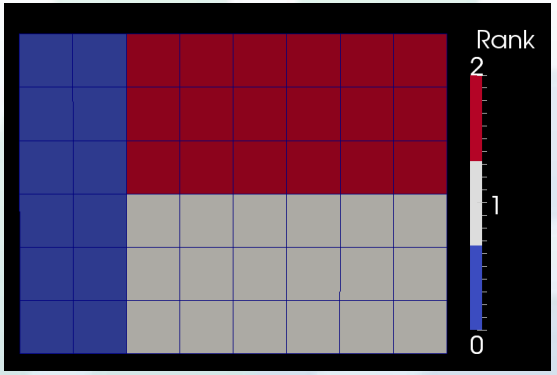
- Classes derive from vtkUniformGridAMR
- Concept of grid levels for changing cell size locally
- Uses only vtkUniformGrids
  - vtkNonOverlappingAMR – no blanking used since no data sets overlap
  - vtkOverlappingAMR – vtkUniformGrids overlap and use blanking



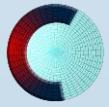
http://www.vtk.org/doc/nightly/html/classvtkUniformGridAMR.html

Grid Partitioning

- For unstructured grids and polydatas
  - Single data set per process – use as is
  - Multiple data sets per process – choice of combining or not depends on memory layout



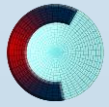
- For topologically structured grids
  - Must be partitioned into logical blocks
  - SetExtent()
    - Index of first and last point in each direction
    - Will be different on each process
  - vtkCPInputDataDescription::SetWholeExtent() – same on each process



Kitware

vtkUnstructuredGrid Example

```
void CreateGrid(vtkCPDataDescription* dataDescription, int
numPoints, double* coordsArray, int numCells, int*
cellConnectivity, double* dofArray)
{
  vtkUnstructuredGrid* grid = vtkUnstructuredGrid::New();
  vtkPoints* nodePoints = vtkPoints::New();
  vtkDoubleArray* coords = vtkDoubleArray::New();
  coords->SetNumberOfComponents(3);
  coords->SetNumberOfTuples(numPoints);
  for(int i=0;i<*numPoints;i++)
    {
    double tuple[3] = {coordsArray[i],
                       coordsArray[i+numPoints],
                       coordsArray[i+numPoints*2]};
    coords->SetTupleValue(i, tuple);
    }
  nodePoints->SetData(coords);
  coords->Delete();
  grid->SetPoints(nodePoints);
  nodePoints->Delete();
…
```

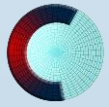


Kitware

vtkUnstructuredGrid Example (2)

- Only tetrahedra in this example
- Canonical ordering of tetrahedra is same in simulation data structures and VTK

```
…
  vtkIdType     [4];
  for(int iCell=0;iCell<numCells;iCell++)
    {
    for(int i=0;i<4;i++)
      {
      pts[i] = cellConnectivity[iCell+i*numCells];
      }
    grid->InsertNextCell(VTK_TETRA, 4, pts);
    }
   dataDescription->GetInputDescriptionByName("input")
      ->SetGrid(grid);
  grid->Delete();
…
```



Kitware

vtkUnstructuredGrid Example (3)

```
…
  vtkCPInputDataDescription* idd =
    dataDescription->GetInputDescriptionByName("input");
  if(idd->IsFieldNeeded("velocity")) {
    vtkDoubleArray* velocity = vtkDoubleArray::New();
    velocity->SetName("velocity");
    velocity->SetNumberOfComponents(3);
    velocity->SetNumberOfTuples(numPoints);
    for (vtkIdType idx=0; idx<numPoints; idx++) {
      velocity->SetTuple3(idx, dofArray[idx],
        dofArray[idx+ numPoints], dofArray[idx+ 2*numPoints]);
      }
    grid->GetPointData()->AddArray(velocity);
    velocity->Delete();
    }
  if(idd->IsFieldNeeded("pressure")) {
    vtkDoubleArray* pressure = vtkDoubleArray::New();
    pressure->SetName("pressure");
    pressure->SetArray(dofArray+3*numPoints, numPoints, 1);
    grid->GetPointData()->AddArray(pressure);
    pressure->Delete();
    }
}
```