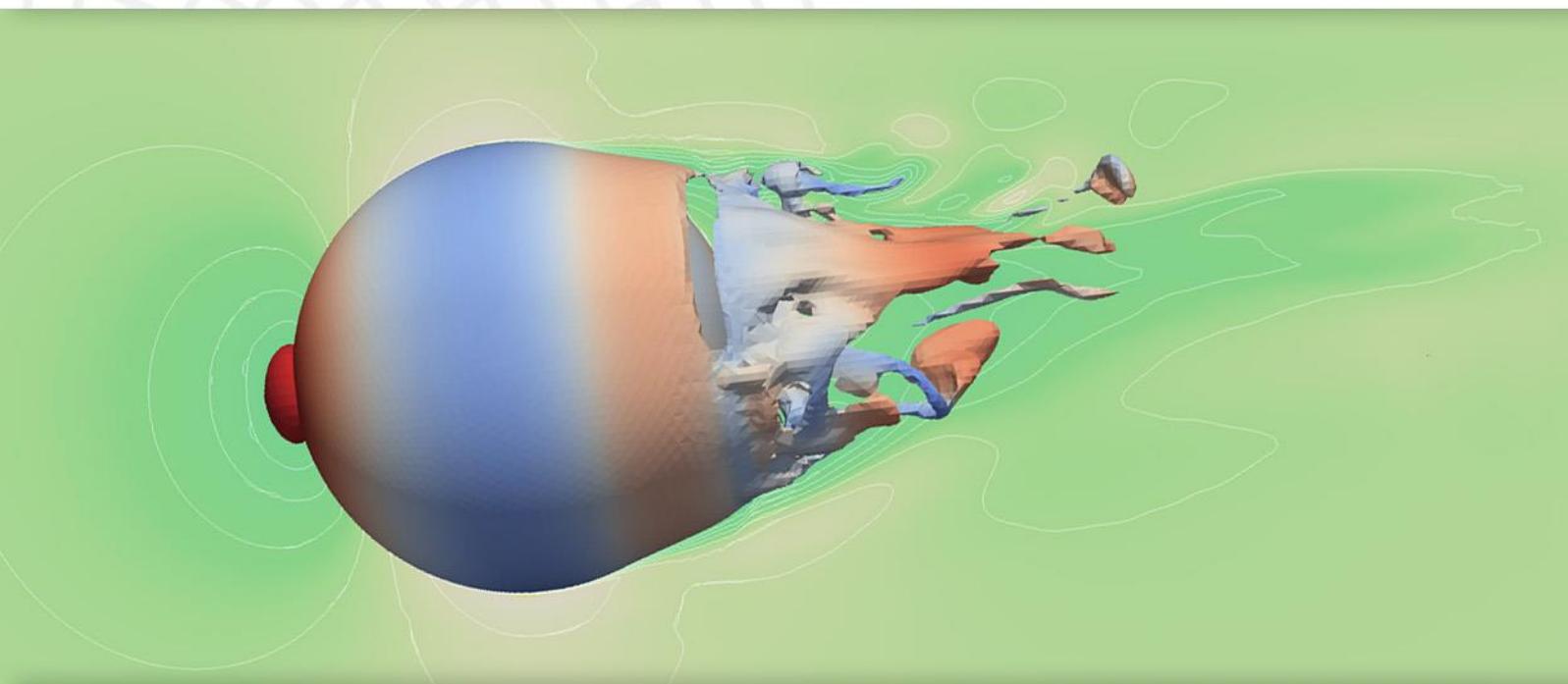


The ParaView Catalyst User's Guide v1.0



Andrew C. Bauer, Berk Geveci, Will Schroeder
Kitware Inc.

The ParaView Catalyst User's Guide is available under a [Creative Commons Attribution license](#) (CC by 3.0). © 2013, Kitware Inc.

<http://www.kitware.com>

Cover image generated from a Helios simulation of compressible flow past a sphere. Helios is developed by the U.S. Army's Aeroflightdynamics Directorate.

We would like to acknowledge the support from:



Ken Moreland is the project lead for Sandia. Sandia has contributed significantly to the project both in development and vision. Sandia developers included Nathan Fabian and Ken Moreland.



Los Alamos National Lab - Jim Ahrens is the project lead at LANL. The LANL team has been integrating Catalyst with various LANL simulation codes and has contributed to the development of the library.



Army SBIR - Mark Potsdam, from Aeroflightdynamics Directorate, was the main technical point of contact for Army SBIRs and has contributed significantly to the vision of Catalyst.

Section 1: Introduction to ParaView Catalyst

Computer simulations are growing in sophistication and producing results of ever greater fidelity. This trend has been enabled by advances in numerical methods and increasing computing power. Yet these advances come with several costs including massive increases in data size, difficulties examining output data, challenges in configuring simulation runs, and difficulty debugging running codes. For computer simulation to provide the many significant benefits that have been exhaustively documented (“The Opportunities and Challenges of Exascale Computing: Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee,” http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf, Fall 2010), it’s imperative to address these issues.

The *Catalyst* library is a system that addresses such challenges. It is designed to be easily integrated directly into large-scale numerical codes. Built on and designed to interoperate with the standard visualization toolkit VTK and the scalable ParaView application, it enables simulations to intelligently perform analysis, generate relevant output data, and visualize results concurrent with a running simulation. This ability to visualize and analyze data from simulations is referred to synonymously as *in situ* processing, co-processing, co-analysis, and co-visualization. Thus Catalyst is often referred to as a co-processing, or *in situ*, library for high-performance computing (HPC).

In the remainder of this *Introduction* section, we will motivate the use of Catalyst, and describe an example workflow to demonstrate just how easy Catalyst is to use in practice.

Motivation

Computing systems have been increasing in speed and capacity for many years now. Yet not all of the various subsystems which make up a computing environment have been advancing equally as fast. This has led to many changes in the way large-scale computing is performed. For example, simulations have long been scaling towards hundreds of thousands of parallel computing cores in recognition that serial processing is inherently limited by the bottleneck of a single processor. As a result, parallel computing methods and systems are now central to modern computer simulation. Similarly, with the number of computing cores increasing to address bigger problems, IO is now becoming a limiting factor as the table below indicates. While the increase in FLOPS between FY2010 and 2018 is expected to be on the order of 500, IO bandwidth is increasing on the order of 100 times.

	2010	2018	Factor Change
Peak FLOP Rate	2 Pf/s	1 Ef/s	500
Input/Output Bandwidth	0.2 TB/s	20 TB/s	100

Table 1.1: Potential exascale computer performance. Source: DOE Exascale Initiative Roadmap, Architecture and Technology Workshop, San Diego, December, 2009.

This divergence between computational power and IO bandwidth has profound implications on the way future simulation is performed. In the past it was typical to break the simulation process into three pieces: pre-processing (preparing input); simulation (execution); and post-processing (analyzing and visualizing results). This workflow is fairly simple and treats these three tasks independently, simplifying the development of new computational tools by relying on a loose coupling via data file exchange between each of the pieces. For example, preprocessing system are typically used to discretize the domain and specify material properties and boundary conditions, finally writing this information out into one or more input files to the simulation code. Similarly, the simulation process typically writes output files which are read in by the postprocessing system. However limitations in IO bandwidth throw a monkey wrench into this process, as the time to read and write data on systems with relatively large computational power is becoming a severe bottleneck to the simulation workflow. Savings can be obtained even for desktop systems with a small amount of parallel processes. This is shown in the figure below for a 6 process run on a desktop machine performing different analysis operations as well as IO. It is clear that the abundance of computational resources (cores) results in relatively rapid analysis, which even when taken together are faster than the time it takes for the simulation code to save a full dataset.

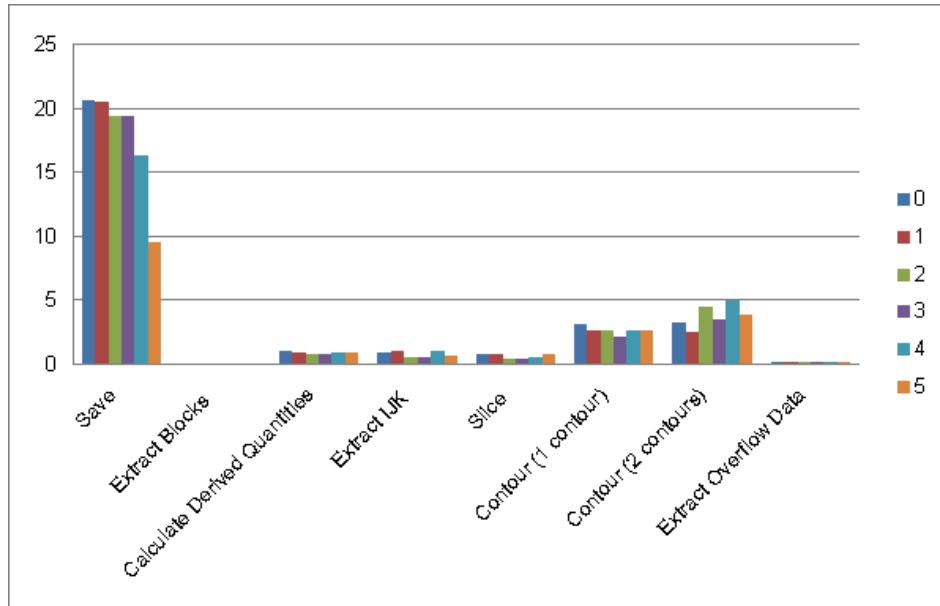


Figure 1.1: Comparison of compute time in seconds for certain analysis operations vs. saving the full data set for a 6 process run on a desktop machine.

The root problem is that due to the ever increasing computational power available on the top HPC machines, analysts are now able to run simulations with increasing fidelity. Unfortunately this increase in fidelity corresponds to an increase in the data generated by the simulation. Due to the divergence between IO and computational capabilities, the resulting data bottleneck is now negatively impacting the simulation workflow. For large problems, gone are the days when

data could be routinely written to disk and/or copied to a local workstation or visualization cluster. The cost of IO is becoming prohibitive, and large-scale simulation in the era of cheap FLOPS and expensive IO requires new approaches.

One popular, but crude approach relies on configuring the simulation process to save results less frequently (for example, in a time-varying analysis, every tenth time step may be saved, meaning that 90% of the results are simply discarded). However even this strategy is problematic: It is possible that a full save of the simulation data for even a single time step may exceed the capacity of the IO system, or require too much time to be practical.

A better approach, and the approach that Catalyst takes, is to change the traditional three-step simulation workflow of pre-processing, simulation, and post-processing (shown below)



to one that integrates post-processing directly into the simulation process as shown below.



This integration of simulation with post-processing provides several key advantages. First, it avoids the need to save out intermediate results for the purpose of post-processing; instead post-processing can be performed *in situ* as the simulation is running. This saves considerable time as illustrated below.

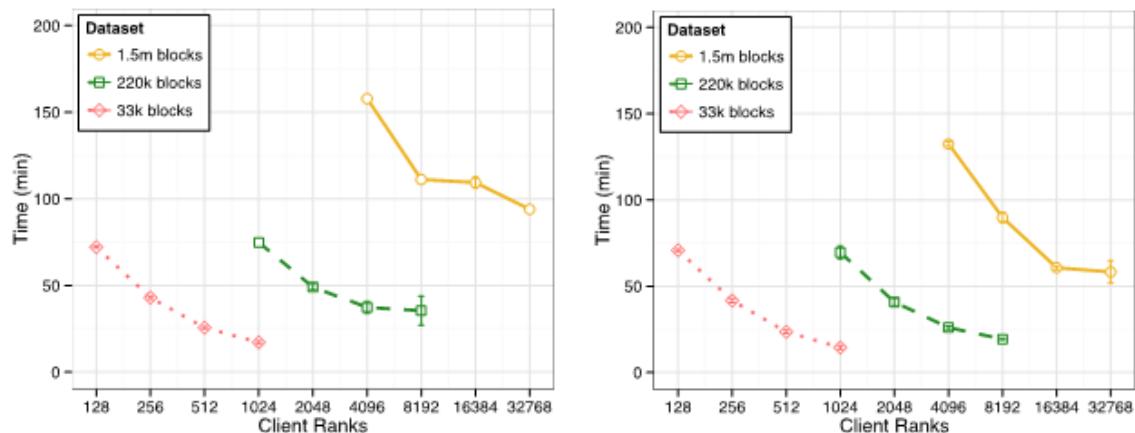


Figure 1.2: Comparison of full workflow for CTH with post-processing results (left plot) vs. full workflow with *in situ* processing with Catalyst. Results courtesy of Sandia National Laboratories.

Further, instead of saving full datasets to disk, IO can be reduced by extracting only relevant information. Data extracts like isocontours, data slices, or streamlines are generally orders of magnitude smaller than the full dataset (see figure below). Thus writing out extracts significantly reduces the total IO cost.

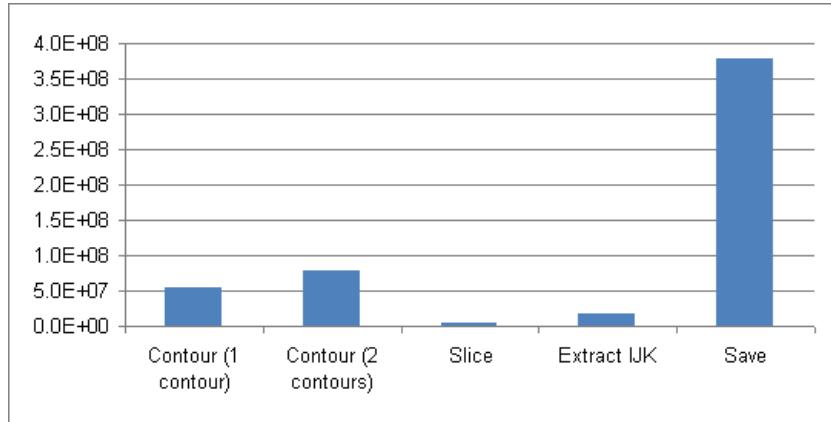


Figure 1.3: Comparison of file size in bytes for saving full data set vs. saving specific analysis outputs.

Finally, unlike blind subsampling of results, using an integrated approach it becomes possible to analyze the current state of the simulation and save only information pertinent to the scientific query at hand. For example, it is possible to identify the signs of a forming shock and then save only that information in the neighborhood of the shock.

There are other important applications that address the complexity of the simulation process. Using co-processing it is possible to monitor the progress of the simulation, and ensure that it is progressing in a valid way. It is not uncommon for a long running simulation (maybe days or longer in duration) to be tossed out because initial boundary conditions or solution parameters were specified incorrectly. By checking intermediate results it's possible to catch mistakes like these and terminate such runs before they incur excessive costs. Similarly, co-processing enables debugging of simulation codes. Visualization can be used to great effect to identify regions of instability or numerical breakdown.

Catalyst was created as a library to achieve the integration of simulation and post-processing. It has been designed to be easy to use and introduces minimal disruption into numerical codes. It leverages standard systems such as VTK and ParaView (for post-processing) and utilizes modern scientific tools like Python for control and analysis. Overall Catalyst has been shown to dramatically increase the effectiveness of the simulation workflow by reducing the amount of IO, thereby reducing the time to gain insight into a given problem, and more efficiently utilizing modern HPC environments with abundant FLOPS and restricted IO bandwidth.

Example Workflow

The figure below demonstrates a typical workflow using Catalyst for *in situ* processing. In this figure it is assumed that Catalyst has already been integrated into the simulation code (see Section 3 for details on how to integrate Catalyst). The workflow is initiated by creating a Python script using ParaView's GUI which specifies the desired output from the simulation. Next, when the simulation starts it loads this script; then during execution any analysis and visualization output is generated in synchronous fashion (i.e., while the simulation is running). Catalyst can produce images/screenshots, compute statistical quantities, generate plots, and extract derived information such as polygonal data or iso-surfaces to visualize geometry and/or data.

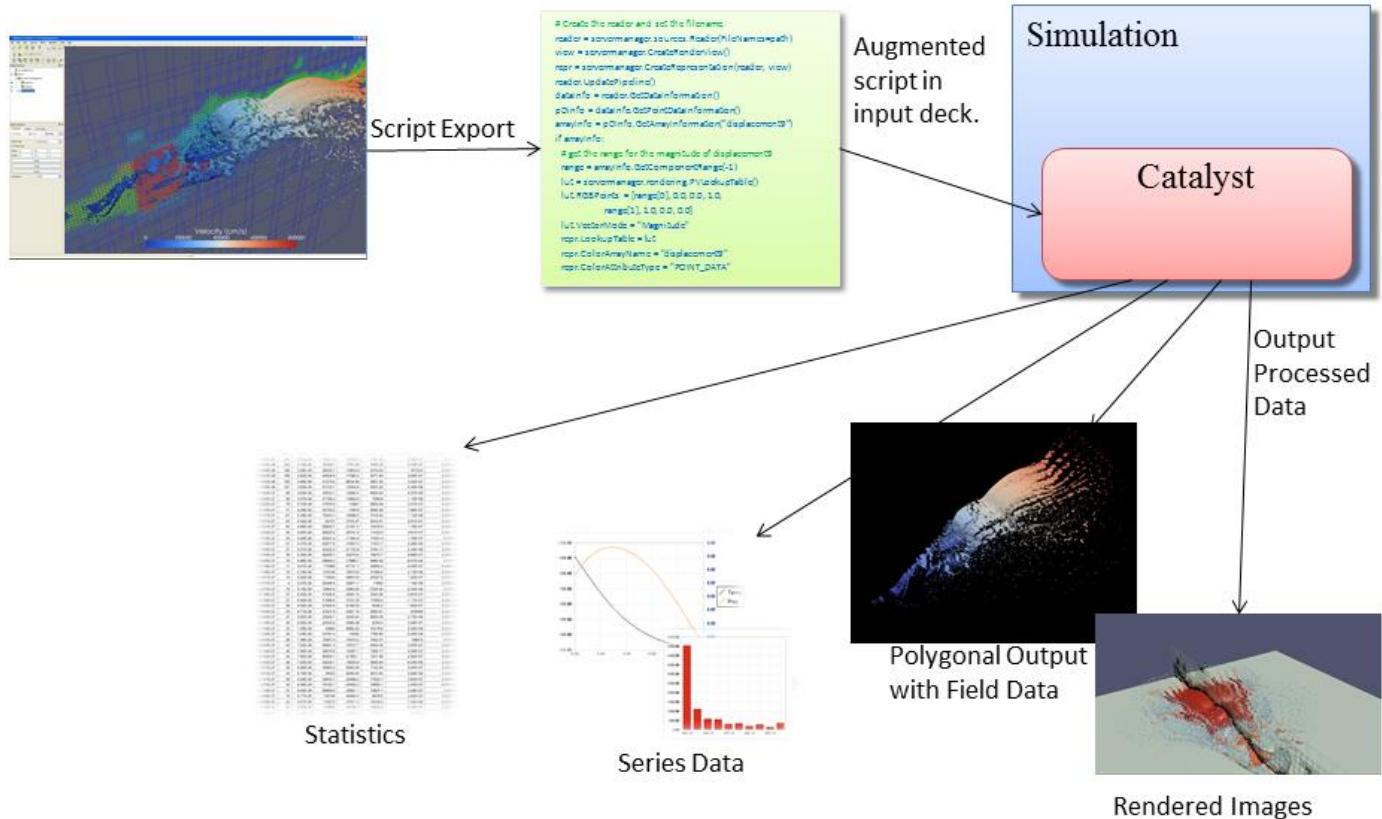


Figure 1.4: *In situ* workflow with various Catalyst outputs.

Catalyst has been used by a variety of simulation codes. CTH, a shock physics code from Sandia, has been instrumented to use Catalyst. Additionally, Phasta from UC Boulder <insert ref>, NPIC and VPIC from LANL, Helios from the Army's Aeroflightdynamics Directorate, S3D and the Sierra simulation framework from Sandia and H3D from UCSD have all been instrumented to use Catalyst. Some example outputs are shown below.

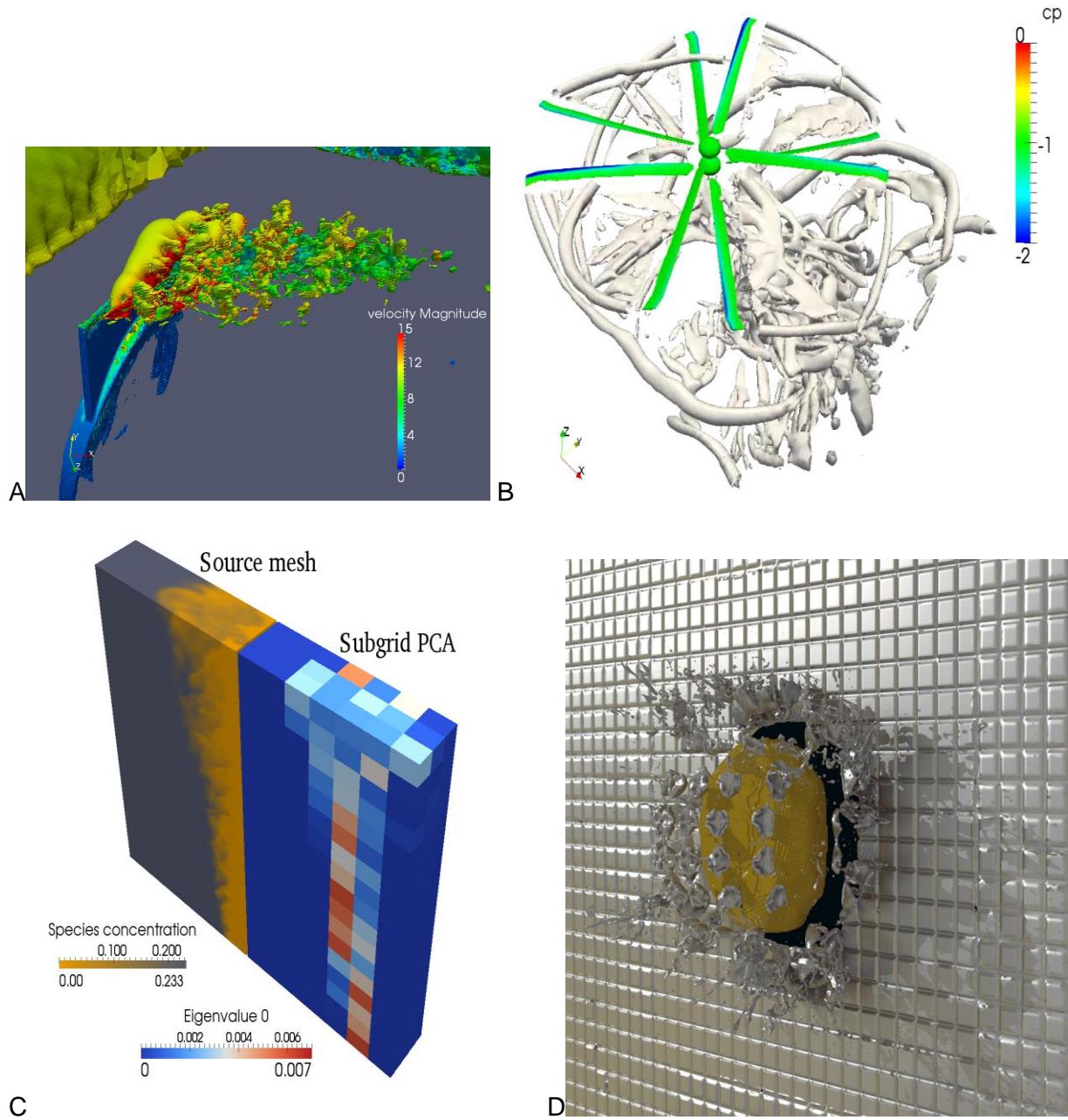


Figure 1.5: Various results from simulation codes linked with Catalyst (A: Phasta; B: Helios; C: S3D; D: CTH). Note that post-processing with different packages was performed with B and D.

Of course Catalyst is not necessarily applicable in all situations. First, if significant reductions in IO are important, then it's important that the specified analysis and visualization pipelines invoked by Catalyst actually produce reduced data size. Another important consideration is whether these pipelines scale appropriately. If they do not, then a large-scale simulation may bog down during co-processing, detrimentally impacting total analysis cycle time. However, both the underlying ParaView and VTK systems have been developed with parallel scaling in mind, and generally perform well in most applications. The figure below shows two scale plots for two

popular algorithms: slicing through a dataset, and decimating large meshes (e.g., reducing the size of an output isocontour).

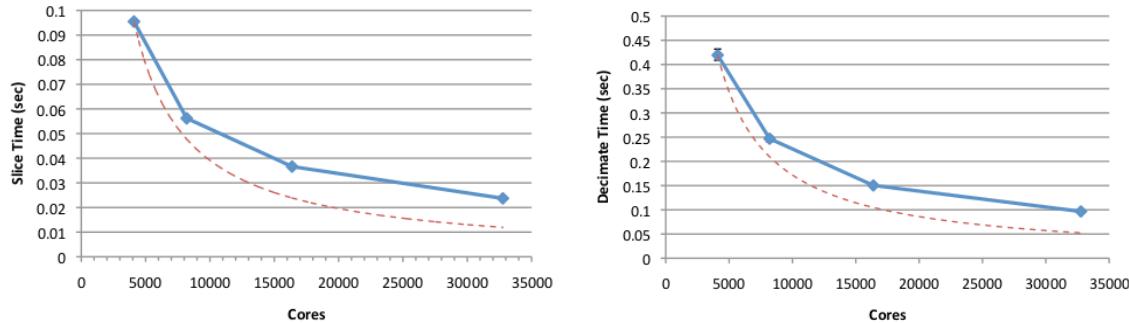


Figure 1.6: Scaling performance of the slice filter (left) and decimate filter (right).

Such stellar performance is typical of VTK algorithms, but we recommend that you confirm this behavior for your particular analysis pipeline(s).

Further Information

The following are various links of interest related to Catalyst:

- www.paraview.org - the main ParaView page with links to wikis, code, documentation, etc.
- www.paraview.org/paraview/resources/software.php - the main ParaView download page -- useful for installing ParaView on local machines for creating Catalyst scripts and viewing Catalyst output.
- catalyst.paraview.org - the main page for Catalyst
- paraview@paraview.org - the mailing list for general ParaView and Catalyst support

The remainder of this guide is broken up into three main sections. Section 2 addresses users that wish to use simulation codes that have already been instrumented with Catalyst. Section 3 is for developers who wish to instrument their simulation code. Section 4 focuses on those users who wish to install and maintain Catalyst on their computing systems.

Section 2: Catalyst for Users

This section describes Catalyst from the perspective of the simulation user. As described in the previous section, Catalyst changes the workflow with the goal of efficiently extracting useful insights from the numerical simulation process.

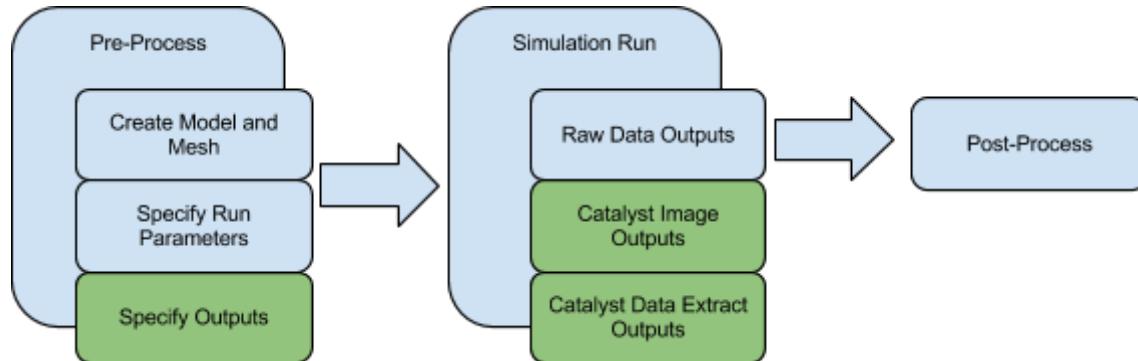


Figure 2.1: Traditional workflow (blue) and Catalyst enhanced workflow (green).

With the Catalyst enhanced workflow, the user specifies visualization and analysis output during the pre-processing step. These output data are then generated during the simulation run and later analyzed by the user. The Catalyst output can be produced in a variety of formats such as rendered images; pseudo-coloring of variables; plots (e.g. bar graphs, line plots, etc.); data extracts (e.g. iso-surfaces, slices, streamlines, etc.); and computed quantities (e.g. lift on a wing, maximum stress, flow rate, etc.). The goal of the enhanced workflow is to reduce the time to gain insight into a given physical problem by performing some of the traditional post-processing work *in situ*. While the enhanced workflow uses ParaView Catalyst to produce *in situ* outputs, the user does not need to be familiar with ParaView to use this functionality. Configuration of the pre-processing step can be based on generic information to produce desired outputs (e.g. an iso-surface value and the variable to iso-surface) and the output can be written in either image file or other formats with which the user has experience.

There are two major ways in which the user can utilize Catalyst for *in situ* analysis and visualization. The first is to specify a set of parameters that are passed into a pre-configured Catalyst pipeline. The second is to create a Catalyst pipeline script using ParaView's GUI.

Pre-Configured Catalyst Pipelines

Creating pre-configured Catalyst pipelines places more responsibility on the simulation developer but can simplify matters for the user. Using pre-configured pipelines can lower the barrier to using Catalyst with a simulation code. The concept is that for most filters there is a limited set of parameters that need to be set. For example, for a slice filter the user only needs to specify a point and a normal defining the slice plane. Another example is for the threshold filter where only the variable and range needs to be specified. For each pipeline though, the parameters should also include a file name to output to and an output frequency. These

parameters can be presented for the user to set in their normal workflow for creating their simulation inputs.

Creating Catalyst Scripts in ParaView

The downside to using pre-configured scripts is that they are only as useful as the simulation developer makes them. These scripts can cover a large amount of use cases of interest to the user but inevitably the user will want more functionality or better control. This is where it is useful for the simulation user to create their own Catalyst Python scripts pipeline using the ParaView GUI.

There are two main prerequisites for creating Catalyst Python scripts in the ParaView GUI. The first is that ParaView is built with the CoProcessing Script Generator plugin enabled (it is enabled by default when building ParaView from source as well as for versions of ParaView installed from the available installers). Note that the version of ParaView used to generate the script should also correspond to the version of Catalyst that the simulation code runs with. The second prerequisite is that the user has a representative data set to start from. What we mean by this is that when reading the data set from disk into ParaView that it is the same data set type (e.g. vtkUnstructuredGrid, vtkImageData, etc.) and has the same attributes defined over the grids as the simulation adaptor code will provide to Catalyst during simulation runs. Ideally, the geometry and the attribute ranges will be similar to what is provided by the simulation run's configuration. The steps to create a Catalyst Python pipeline in the ParaView GUI are:

1. First load the ParaView plugin for creating the scripts. Do this by going to the Tools menu and selecting “Manage Plugins...”. In the window that pops up, select CoProcessingPlugin and press the “Load Selected” button. After this, press the Close button to close the window. This will create two new top-level menu items, Writers and CoProcessing. Note that you can have the plugin automatically loaded when ParaView starts up by expanding the CoProcessingPlugin information by clicking on the + sign in the box to the left of it and then by checking the box to the right of Auto Load.
2. Next, load in a representative data set and create a pipeline. In this case though, instead of actually writing the desired output to a file we need to specify when and where the files will be created when running the simulation. For data extracts we specify at this point that information by choosing an appropriate writer under the Writers menu. The user should specify a descriptive file name as well as a write frequency in the Properties panel as shown in the image below. The file name must contain a %t in it as this gets replaced by the time step when creating the file. Note that the step to specify screenshot outputs for Catalyst is done later.

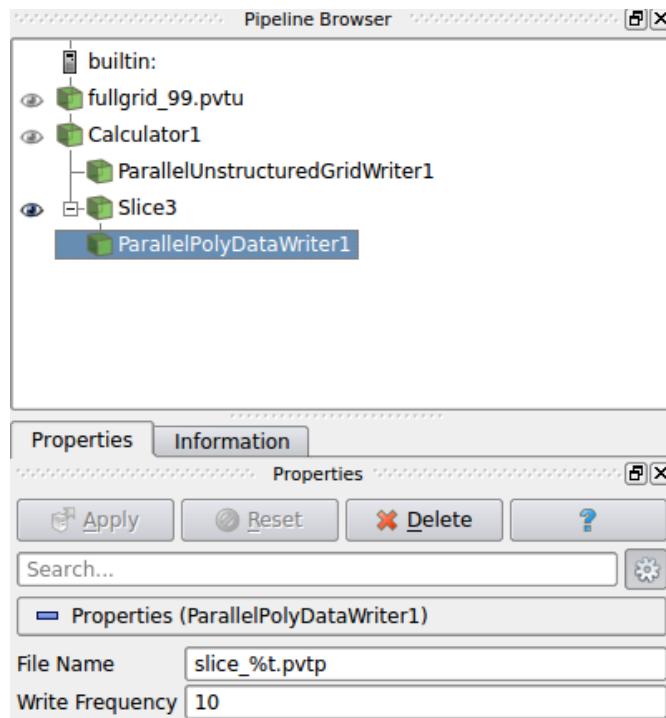


Figure 2.2: Example of a pipeline with two writers included. The first writes output from the Calculator filter and the second write output from the Slice filter. The highlighted polydata writer also shows the file name and the write frequency for Catalyst output.

3. Once the full Catalyst pipeline has been created, the Python script must be exported from ParaView. This is done by choosing the Export State wizard under the CoProcessing menu. The user can click on the Next button in the initial window that pops up.
4. After that, the user must select the sources (i.e. pipeline objects without any input connections) that the adaptor will create and add them to the output. In this case it is the fullgrid_99.pvtu source from the above figure that is analogous to the input that the simulation code's adaptor will provide. The user can either double click on the desired sources in the left box to add them to the right box or select the desired sources in the left box and click Add. This is shown in the Figure below. After all of the proper sources have been selected, click on Next.

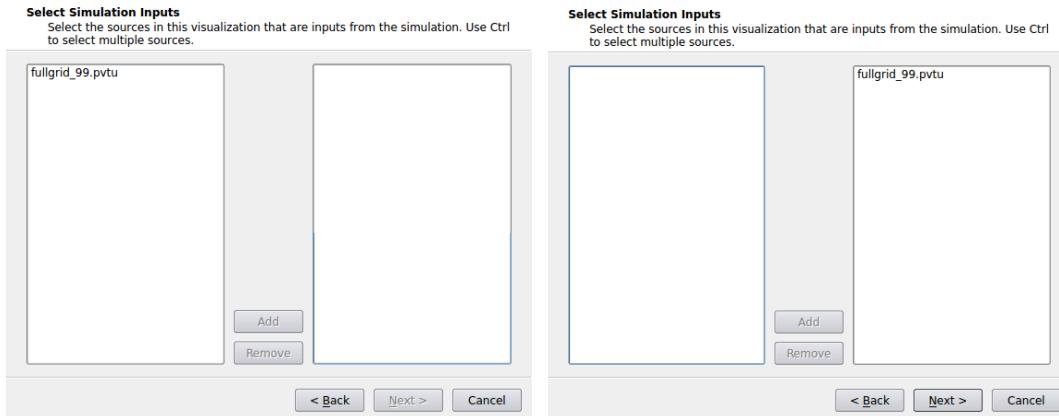


Figure 2.3: Selecting fullgrid_99.vtu as an input for the Catalyst pipeline. The left image shows the source not being selected and the right shows it being selected.

- The next step is labeling the inputs. The most common case is a single input in which case we use the convention that it should be named “input”, the default value. For situations where the adaptor can provide multiple sources (e.g. fluid-structure interaction codes where a separate input exists for the fluid domain and the solid domain), the user will need to label which input corresponds to which label. This is shown in the figure below. After this is done, click Next.

Name Simulation Inputs
Assign names for the selected simulation inputs.

	Pipeline Name	Simulation Name
1	fullgrid_99.pvtu	input

< Back Next > Cancel

Figure 2.4: Providing identifier strings for Catalyst inputs.

- The next page in the wizard gives the user the option to allow Catalyst to check for a Live Visualization connection and to output screenshots from different views. Check the box next to Live Visualization to enable it. For screenshots, there are a variety of options. The first is a global option which will rescale the lookup table for pseudo-coloring to the current data range for all views. The other options are per view and are:
 - Image type -- choice of image format to output the screenshot in.

- File Name -- the name of the file to create. It must contain a %t in it so that the actual simulation time step value will replace it.
- Write Frequency -- how often the screenshot should be created.
- Magnification -- the user can create an image with a higher resolution than the resolution shown in the current ParaView GUI view.
- Fit to Screen -- specify whether to fit the data in the screenshot. This gives the same results in Catalyst as clicking on the  button in the ParaView GUI.

If there are multiple views, the user should toggle through each one with the Next View and Previous View buttons in the window. After everything has been set, click on the Finish button to create the Python script.

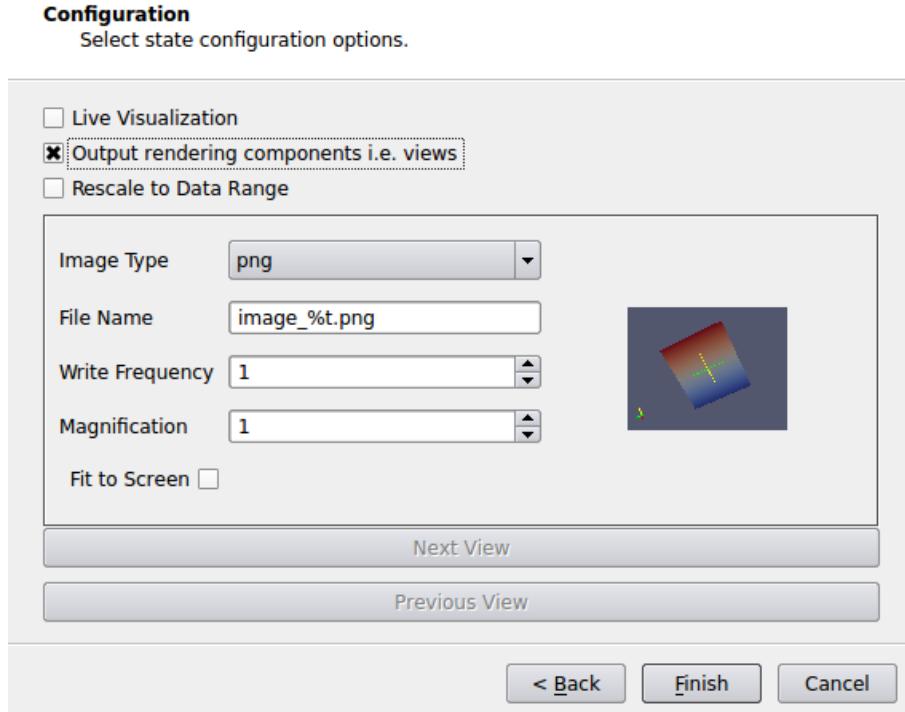


Figure 2.5: Setting the parameters for outputting screenshots.

7. The final step is specifying the name of the generated Python script. Specify a directory and a name to save the script at and click OK when finished.

Creating a Representative Data Set

A question that often arises is how to create a representative data set. There are two ways to do this. The first is by using the sources and filter in ParaView and the second one is to run the simulation with Catalyst with a script that outputs the full grid with all attribute information. The easiest grids to create within the GUI are image data grids (i.e. uniform rectilinear grids), polydata and unstructured grids. For those knowledgeable enough about VTK, the programmable source can also be used to create all grid types. If a multi-block grid is needed, the group datasets filter can be used to group together multiple data sets into a single output.

The next step is to create the attribute information (i.e. point and/or cell data). This can be easily done with the calculator filter as it can create data with one or three components, name the array to match the name of the array provided by the adaptor, and set an appropriate range of values for the data. Once this is done, the user should save this out and then read the file back in to have the reader act as the source for the pipeline. The second method involves running the simulation with Catalyst with a general Python pipeline script that outputs the data set in its proper file format.

Manipulating Python scripts

For users that are comfortable programming in Python, we encourage them to modify the given scripts as desired. The following information can be helpful for doing this:

- Sphinx generated ParaView Python API documentation at www.paraview.org/ParaView3/Doc/Nightly/www/py-doc/index.html.
- Using the ParaView GUI trace functionality to determine how to create desired filters and set their parameters. This is done with Start Trace and Stop Trace under the Tools menu.
- Using the ParaView GUI Python shell with tab completion. This is done with Python Shell under the Tools menu.

Avoiding Data Explosion

A key point to keep in mind when creating Catalyst pipelines is that the choice and order of filters can make a dramatic difference in the performance of Catalyst. Often, the source of performance degradation is when dealing with very large amounts of data. For memory-limited machines like today's supercomputers, poor decisions when creating a pipeline can cause the executable to crash due to insufficient memory. The worst case scenario is creating an unstructured grid from a topologically regular grid. This is because the filter will change from using a compact grid data structure to a more general grid data structure.

We classify the filters into several categories, ordered from most memory efficient to least memory efficient:

1. Total shallow copy or output independent of input -- negligible memory used in creating a filter's output.
2. Add field data -- the same grid is used but an extra variable is stored.
3. Topology changing, dimension reduction -- the output is a polygonal data set but the output cells are one or more dimensions less than the input cell dimensions.
4. Topology changing, moderate reduction -- reduces the total number of cells in the data set but outputs in either a polygonal or unstructured grid format.
5. Topology changing, no reduction -- does not reduce the number of cells in the data set while changing the topology of the data set and outputs in either a polygonal or unstructured grid format.

When creating a pipeline, the filters should generally be ordered in this same fashion to limit data explosion. For example, pipelines should be organized to reduce dimensionality early. Additionally, reduction is preferred over extraction (e.g. the Slice filter is preferred over the Clip

filter). Extracting should only be done when reducing by an order of magnitude or more. When outputting data extracts, subsampling (e.g. the Extract Subset filter or the Decimate filter) can be used to reduce file size but caution should be used to make sure that the data reduction doesn't hide any fine features. Below we categorize the common filters in ParaView.

Total Shallow Copy or Output Independent of Input

Annotate Time, Append Attributes, Extract Block, Extract Datasets, Extract Level, Glyph, Group Datasets, Histogram, Integrate Variables, Normal Glyphs, Outline, Outline Corners, Plot Over Line, Probe Location

Add Field Data

Block Scalars, Calculator, Cell Data to Point Data, Compute Derivatives, Curvature, Elevation, Generate Ids, Generate Surface Normals, Gradient, Level Scalars, Median, Mesh Quality, Octree Depth Limit, Octree Depth Scalars, Point Data to Cell Data, Process Id Scalars, Random Vectors, Resample with Dataset, Surface Flow, Surface Vectors, Transform, Warp (scalar), Warp (vector)

Topology Changing, Dimension Reduction

Cell Centers, Contour, Extract CTH Fragments, Extract CTH Parts, Extract Surface, Feature Edges, Mask Points, Outline (curvilinear), Slice, Stream Tracer

Topology Changing, Moderate Reduction

Clip, Decimate, Extract Cells by Region, Extract Selection, Quadric Clustering, Threshold

Topology Changing, No Reduction

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

Section 3: Catalyst for Developers

In this section we describe how developers can interface a simulation code with the Catalyst libraries. The interface to the simulation code is called an *adaptor*. Its primary function is to adapt the internal structures of the simulation code information and transform these structures into forms that Catalyst can process. This process is depicted in the figure below.



A developer creating an adaptor needs to have knowledge of the simulation code data structures, relevant understanding of the appropriate VTK data model, and the Catalyst API. Examples of adaptors are available online at <https://github.com/acbauer/CatalystExampleCode>.

High-Level View

While interfacing Catalyst with a simulation code may require significant effort, the impact on the code base is minimal. In most situations, there are only three functions that need to be called from the existing simulation code:

1. Initialize -- Catalyst needs to be initialized in order to be put in the proper state. For codes that depend on MPI, this is normally done after `MPI_Init()` is called. The initialize method is often implemented in the adaptor.
2. CoProcess -- This function calls the adaptor code to check on any computations that Catalyst may need to do. This call needs to provide the grid and field data structures to the adaptor as well as time and time step information. It may also provide additional control information but that is not required. This is normally called at the end of every time step update in the simulation code (i.e. after the fields have been updated to the new time step and/or the grid has been modified).
3. Finalize -- On completion the simulation code must call Catalyst to finalize any state and properly clean up after itself. For codes that depend on MPI, this is normally done before `MPI_Finalize()` is called. The finalize method is often implemented in the adaptor.

This is demonstrated in the code below:

```
MPI_Init(argc, argv);
#ifndef CATALYST
CatalystInit(argc, argv);
#endif
for(int timeStep=0;timeStep<numberOfTimeSteps;timeStep++)
{
    <update grids and fields to timeStep>
#ifndef CATALYST
CatalystCoProcess(timeStep, time, <grid info>, <field info>);
#endif
}
```

```
#ifdef CATALYST
CatalystFinalize();
#endif
MPI_Finalize();
```

The adaptor code should be implemented in a separate source file. The reason for this is that it simplifies the simulation code build process. The fact that there are only three calls to the adaptor from the simulation code also helps in this matter.

As shown above, the adaptor code is responsible for the interface between the simulation code and Catalyst. Besides being responsible for initializing and finalizing Catalyst, the other responsibilities of the adaptor are:

- Querying Catalyst to see if any co-processing needs to be performed.
- Providing VTK data objects representing the grids and fields for co-processing.

The pseudo-code shown below gives an idea of what this would look like in the adaptor:

```
void CatalystCoProcess(int timeStep, double time, <grid info>,
                      <field_info>
)
{
    1. Specify current timeStep and time for Catalyst
    2. Check with Catalyst if anything needs to be done this call
    3. If nothing needs to be done this call, return
    4. Create VTK grid
    5. Create VTK fields and associate with VTK grid
    6. Specify VTK grid for Catalyst
    7. Call Catalyst to perform co-processing (i.e. execute VTK
       pipelines)
}
```

A complete example of a simple adaptor is shown below. Following this section we'll discuss the details of the API to help solidify the understanding of the flow of information.

```
// static data
vtkCPPProcessor* Processor = NULL;

void CatalystInit(int numScripts, char* scripts[])
{
    if(Processor == NULL)
    {
        Processor = vtkCPPProcessor::New();
        Processor->Initialize();
    }
    // scripts are passed in as command line arguments
    for(int i=0;i<numScripts;i++)
    {
        vtkCPPythonScriptPipeline* pipeline =
```

```

    vtkCPPythonScriptPipeline::New();
    pipeline->Initialize(scripts[i]);
    Processor->AddPipeline(pipeline);
    pipeline->Delete();
}
}

void CatalystFinalize()
{
    if(Processor)
    {
        Processor->Delete();
        Processor = NULL;
    }
}

// The grid is a uniform, rectilinear grid that can be specified
// with the number of points in each direction and the uniform
// spacing between points. There is only one field called
// temperature which is specified over the points/nodes of the
// grid.
void CatalystCoProcess(
    int timeStep, double time, unsigned int numPoints[3],
    double spacing[3], double* field)
{
    vtkCPDataDescription* dataDescription =
        vtkCPDataDescription::New();
    dataDescription->AddInput("input");
    dataDescription->SetTimeData(time, timeStep);
    if(Processor->RequestDataDescription(
        dataDescription) != 0)
    {
        // Catalyst needs to output data
        // Create a uniform grid
        vtkImageData* grid = vtkImageData::New();
        grid->SetExtents(0, numPoints[0]-1, 0, numPoints[1]-1,
                          0, numPoints[2]-1);
        dataDescription->GetInputDescriptionByName("input")
            ->SetGrid(grid);
        grid->Delete();
        // Create a field associated with points
        vtkDoubleArray* array = vtkDoubleArray::New();
        array->SetName("temperature");
        array->SetArray(field, grid->GetNumberOfPoints(), 1);
        grid->GetPointData()->AddArray(array);
    }
}

```

```

        array->Delete() ;
        Processor->CoProcess (dataDescription) ;
    }
    dataDescription->New () ;
}

```

Overview

Before we go into the details of the VTK and Catalyst API required for writing the adaptor code, we would like to highlight some general details to keep in mind:

- VTK does indexing starting at 0.
- A vtkIdType is an integer type that is set during Catalyst configuration. It can either be a 32 bit or a 64 bit integer and by default is based on a native data type. A user may decide to manually configure Catalyst to use either size. The advantage here could be reusing existing data array memory instead of allocating extra memory to store essentially the same information in a different data type.
- The most up-to-date Doxygen generated VTK documentation can be found at www.vtk.org/doc/nightly/html/classes.html.
- The most up-to-date Doxygen generated ParaView documentation can be found at www.paraview.org/ParaView3/Doc/Nightly/html/classes.html

The most in-depth knowledge of VTK that is required for writing the adaptor code is how to create the VTK objects that are used to represent the grid and the field information.

Since VTK is a general toolkit, it has a variety of ways of representing grids. The reason for this is that it needs the generality of being able to handle topologically unstructured grids while also having the constraint that it can handle simpler grids (e.g. topologically uniform, axis-aligned grids) efficiently as well. The following figure shows the types of grids that are supported in VTK.

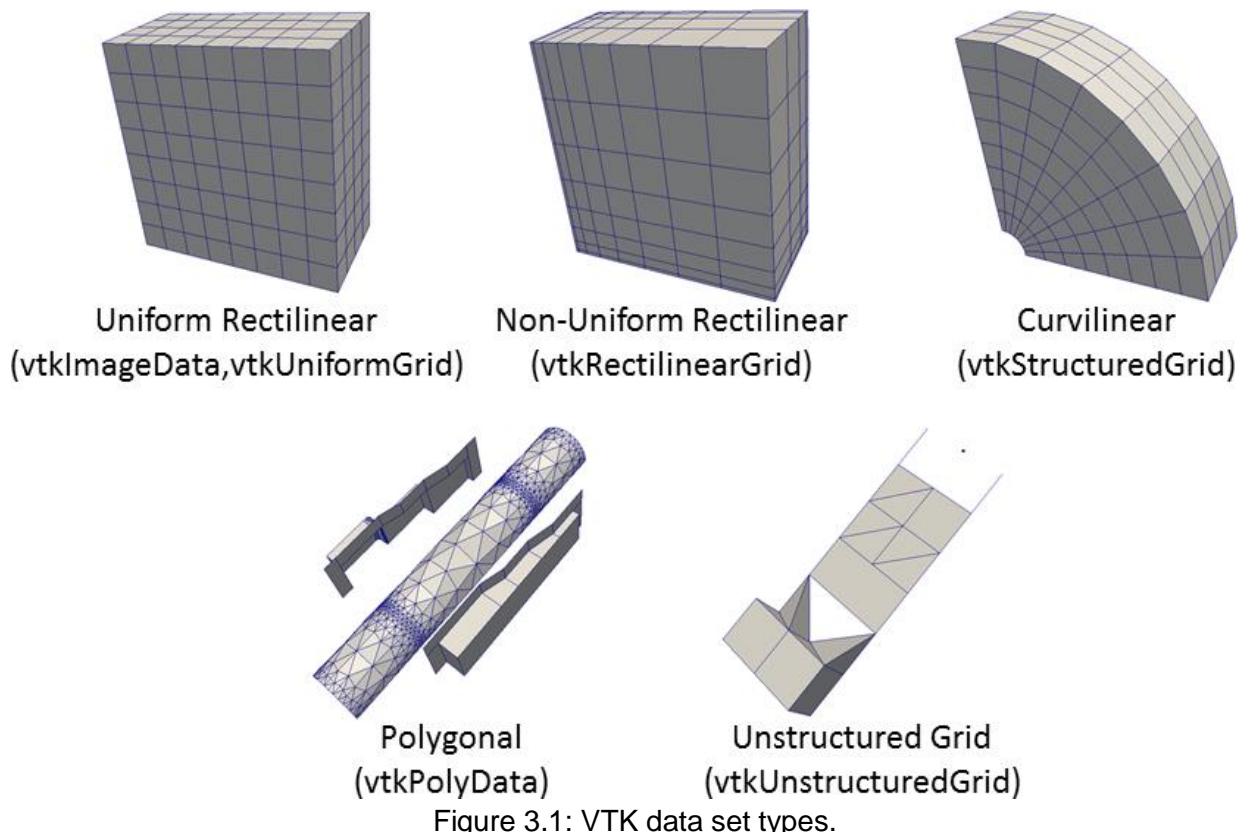


Figure 3.1: VTK data set types.

In addition to these data set types, VTK also supports a wide variety of cell types as well. These include all of the normal 2D and 3D linear cell types such as triangles, quadrilaterals, tetrahedron, pyramids, prisms/wedges and hexahedron. VTK also supports associating field information with each point or cell in the data sets. In VTK this is called attribute data in general and point data and cell data when it is with respect to points or cells in the data set, respectively. Figure 3.10 below shows the difference between point data and cell data.

The overall structure of a VTK data set is that it has grid information, arrays for information associated with each point in the grid and arrays for information associated with each cell in the grid. This is shown in the figure below.

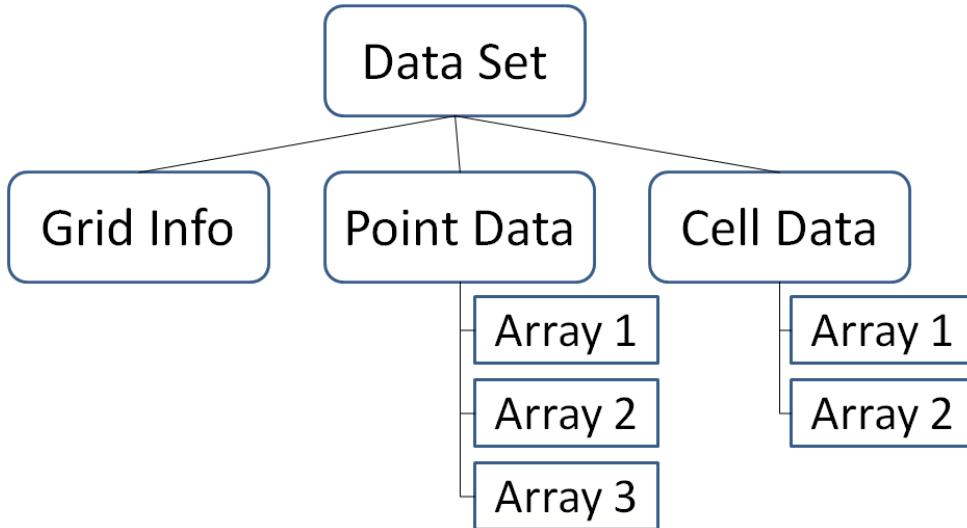


Figure 3.2: High level view of a VTK data set.

VTK Data Object API

It is important to know that VTK uses a pipeline architecture to process data. See the Pipeline section on page 46 of the *ParaView User's Guide*

(<http://www.paraview.org/paraview/help/download/ParaView%20User%27s%20Guide%20v3.10.pdf>). This pipeline architecture has some consequences for writing the adaptor. The first is that the objects that process the data, filters in VTK parlance, are not allowed to modify any input data objects. The second is since VTK is a visualization system, once data objects are created they are typically not incrementally modified (e.g. removing a cell from a grid). Hence many of the data objects are stored in flat arrays in memory to preserve computational efficiency.

In the sections that follow, we do not discuss the full public API of each object since just a few methods are used when creating data objects from scratch. In addition, most of the methods described below are “setter” methods with corresponding “getter” methods that are not described here. For a full description of these classes we refer the reader to the online Doxygen documentation and the VTK and ParaView User’s Guides.

vtkObject

Almost all VTK classes derive from `vtkObject`. This class provides many basic capabilities including reference counting (to handle the creation, sharing and deletion of objects). Reference counting enables the VTK user to track how many places an instantiated object is used as well as when it can be deleted (i.e. when its reference count goes to zero). VTK doesn’t allow `vtkObjects` to be created directly through their constructors. Instead all objects that derive from `vtkObject` use the static `New()` method to create a new instance (this method is referred to as an object factory). Because of reference counting, users are also not allowed to directly delete an object. Instead, the reference count is reduced on an instance by invoking the `Delete()` method on it when it is no longer needed within a particular scope. Thus the `vtkObject` (and its

subclasses) will automatically be deleted when the reference count goes to zero. The following code snippet shows an example of how VTK objects are created, referenced and deleted.

```
vtkDoubleArray* a =
    vtkDoubleArray::New();      // a's ref count = 1
vtkPointData* pd =
    vtkPointData::New();      // pd's ref count = 1
pd->AddArray(a);          // a's ref count = 2
a->Delete();              // a's ref count = 1
a->SetName("an array");   // valid as a hasn't been deleted
pd->Delete();             // deletes both pd and a
```

Some key points here, dereferencing a or pd after pd has been deleted is a bug. It is valid though to dereference a pointer to a VTK object after Delete() has been called on it as long as its reference count is one or greater. To simplify the management of objects that derive from vtkObject, vtkWeakPointer, vtkSmartPointer and vtkNew can be used. These are covered in the appendix.

vtkdataArray

The first major VTK data object we will discuss is vtkdataArray and its concrete implementations (e.g. vtkDoubleArray, vtkIntArray, vtkFloatArray, vtkIdTypeArray, etc.). Concrete classes that derive from vtkdataArray typically store numerical data and are always homogeneous. They also store their data in a contiguous block of memory and assume that the data is not sparse. Since there can be many data arrays associated with a grid, we identify them with a string name and use the const char* GetName() and void SetName(const char* name) methods to get and set the name of the array, respectively. vtkdataArray uses the concept of *tuples* and *components*. A component is a single data value of a tuple. A tuple is a set of pieces of information representing a single concept. For example, for representing pressure there would be a single component in each tuple. For velocity in a 3D space there would be 3 components in a tuple. The number of tuples in a vtkdataArray corresponds to the number of these objects to be represented. For example, if the array was being used to store values at nodes, or points, of the grid, the number of tuples for the array would be equal to the number of nodes in the grid it is defined over. This is shown in the figure below where we have a tuple of size 3 and 6 nodes in the grid, resulting in an array of size 18.

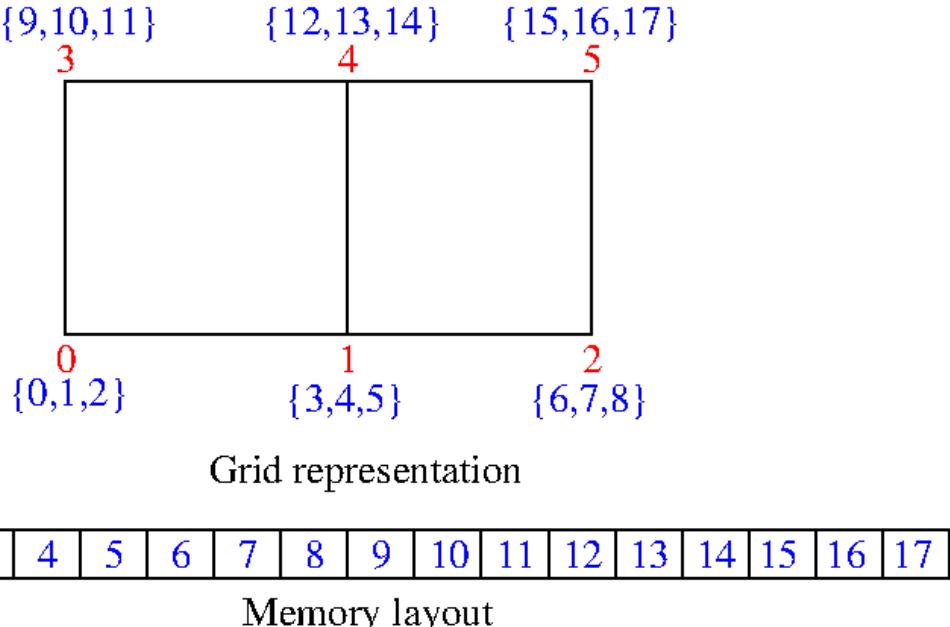


Figure 3.3: Grid representation of a vtkDataArray specified at nodes of the grid. The node index is in red and the array index of each tuple component is shown in blue.

vtkdataArray can either use existing memory or allocate its own space to store the data. The preferred way is to use existing memory if the layout matches what VTK is expecting. If the memory layout matches, this is the recommended way since no extra memory is needed to store the information in VTK format and no memory copy operation needs to be performed. The methods to do this for a vtkFloatArray are:

- void SetArray(float* array, vtkIdType size, int save)
- void SetArray(float* array, vtkIdType size, int save, int deleteMethod)

The parameters are:

- array -- the pointer to the existing chunk of memory to be used.
- size -- the length of the array which needs to be at least the number of tuples multiplied by the number of components in the vtkdataArray.
- save -- set to 1 to keep the class from deleting the array when it is deleted or set to 0 to have the array deleted when the object is deleted. By default the memory will be freed using free().
- deleteMethod -- set to VTK_DATA_ARRAY_FREE to use free() or set to VTK_DATA_ARRAY_DELETE to use delete[] to free the memory.

As VTK filters don't modify their input, it is guaranteed that Catalyst will not modify any of the values in the passed in array. An example of creating a vtkFloatArray from existing memory is shown below:

```

vtkFloatArray* arr = vtkFloatArray::New();
arr->SetName("an array");
float* values = new float[300];
arr->SetArray(values, 300, 0,

```

```

    vtkDoubleArray::VTK_DATA_ARRAY_DELETE) ;
arr->SetNumberOfComponents(3) ;

```

In this example, values will be deleted when the array arr gets deleted. It will have 100 tuples and 3 components. The component values still need to be specified however.

If the memory layout doesn't match what VTK expects, the adaptor will have to allocate additional memory in order to pass the data to Catalyst. There are multiple ways to set the size of the array. For adaptors the length of the array is usually known before it is constructed. In this case the user should call SetNumberOfComponents(int) first and then SetNumberOfTuples(vtkIdType) to set the proper length. The values of the array should be set using one of the following, assuming we're using a vtkFloatArray object:

- void SetValue(vtkIdType id, float value) -- set a single value at location id in the array.
- void SetTupleValue(vtkIdType i, float* tuple) -- set all components of the i'th tuple in the array.

It is important to note that the above methods do not perform range checking. This enables faster execution time but at the expense of potential memory corruption. Some sample code is shown below.

```

vtkIntArray* arr = vtkIntArray::New() ;
arr->SetNumberOfComponents(3) ;
arr->SetNumberOfTuples(100) ;
arr->SetName("an array") ;
int tuple[3] ;
for(vtkIdType i=0;i<100;i++)
{
    tuple[0] = <value> ;
    tuple[1] = <value> ;
    tuple[2] = <value> ;
    arr->SetTupleValue(i, tuple) ;
}

```

If the array length isn't known ahead of time then the following methods, which perform range checking and allocate memory as necessary, should be used, again assuming that the object is a vtkFloatArray:

- vtkIdType InsertNextValue(float value) -- set a single value in the next location in the array and return the newly created array index.
- vtkIdType InsertNextTupleValue(const float* tuple) -- set the next tuple of values in the array and return the newly created tuple index.
- void InsertValue(vtkIdType id, float value) -- set the value at location id in the array to value.

- void InsertTupleValue(vtkIdType i, const float* tuple) -- set the tuple values for the array at tuple location i.

Note that all of these methods will allocate memory as needed. Similar to C++'s stl vector, memory is not allocated for every call to these methods though but doubled when inserting beyond its capacity. The Squeeze() method can be used to regain all of the unused capacity. For the last two functions, the user needs to be careful since using them can result in uninitialized values to be contained in the array.

Grid Types

VTK has a variety of grid types to choose from. They all derive from vtkDataSet and are inherently spatial structures. vtkDataSet also allows the subcomponents, i.e. the points and cells in VTK parlance, to have attributes stored in vtkDataArrays set on them. The types of grids available in VTK are polygonal mesh/polydata, unstructured grid, structured (curvilinear) grid, rectilinear grid and image data/uniform rectilinear grid. In VTK these grid types correspond to the following classes respectively: vtkPolyData, vtkUnstructuredGrid, vtkStructuredGrid, vtkRectilinearGrid and vtkImageData/vtkUniformGrid. Examples of each are shown in the Figure 3.1. The class hierarchy is shown below:

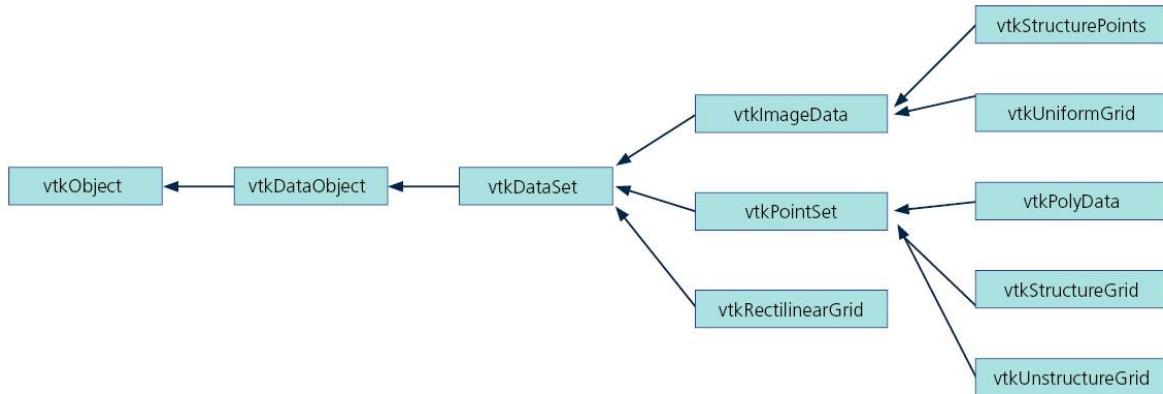


Figure 3.4: Class hierarchy for VTK data sets.

The most efficient grids for storage take advantage of a predefined topology and geometry. They are also the least general.

Topologically Structured Grids

vtkImageData, vtkUniformData, vtkRectilinearGrid and vtkStructuredGrid all assume a regular grid topology. When iterating over points or cells, the order is fastest in the logical i direction, next in the logical j direction and finally in the logical k direction. For axis-aligned grids these correspond to the x-, y-, and z-directions, respectively. For these regular grids, we use what are called extents for describing their topology as well as how they are partitioned over multiple processes. Extents are arrays of 6 integers which specify the start and end indices of the points in each of the three logical directions. The whole extent is the extent for the entire grid and sub-extent, often referred to just as the extent, is one portion of the whole extent that is accessed at a time. For adaptors the sub-extent will usually correspond to an individual process's part of the

grid. This is shown in the figure below. Note that due to using extents, the partitioning is forced to be logically blocked into contiguous pieces. While extents exist for each logical direction, these grids are not required to have more than a single point in any logical direction. This allows the creation of 1D, 2D or 3D structured grids.

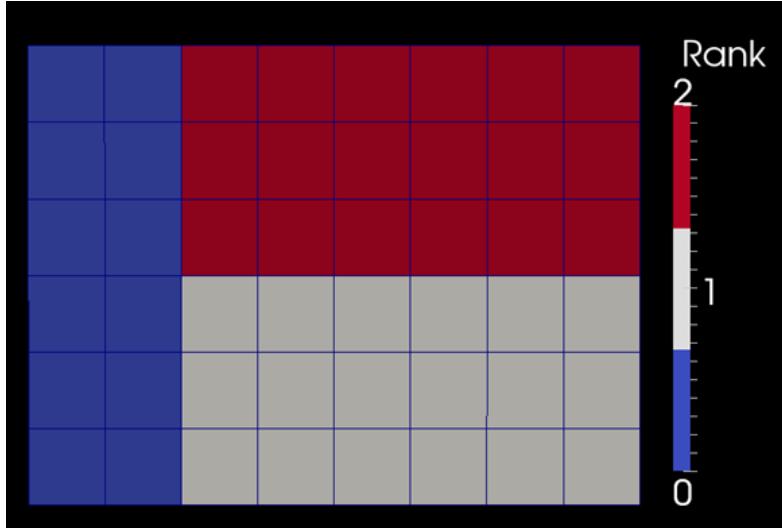


Figure 3.5: Three process partition of a grid. The whole extent for the points is (0, 8, 0, 6, 0, 0).

Process 0 (blue) has an extent of (0, 3, 0, 6, 0, 0) which results in 21 points and 18 cells, process 1 (grey) has an extent of (3, 8, 0, 3, 0, 0) which results in 28 points and 18 cells, and process 2 (red) has an extent of (3, 8, 3, 6, 0, 0) which results in 28 points and 18 cells.

For each of VTK's topologically structured grid types, the user must set the extent for each process. This can be done with either of the two following methods:

- `void SetExtent (int extent[6])`
- `void SetExtent (int x1, int x2, int y1, int y2, int z1, int z2)`

Negative values for extents can be used as long as the second extent in a direction is greater than or equal to the first. The user should not use the `SetDimensions()` methods of any of these classes as this will cause problems with partitioning the structured grids in parallel. Additionally, in the adaptor the user must call either of the following two methods in the `vtkCPInputDataDescription` object for setting the whole extent of the grid:

- `void SetWholeExtent (int x1, int x2, int y1, int y2, int z1, int z2)`
- `void SetWholeExtent (int extent[6])`

We will go into the details of this later but it is worth mentioning here as this step is often forgotten.

As we mentioned earlier, iterating over points and cells is fastest in the logical i direction, then the logical j direction, and slowest in the logical k direction. Indexing of points and cells is done

independent of its whole extent but the logical coordinates are with respect to the whole extent. For example, the flat indexing and logical indexing of the points and cells are shown in the figure below for process 2's partition in the above figure.

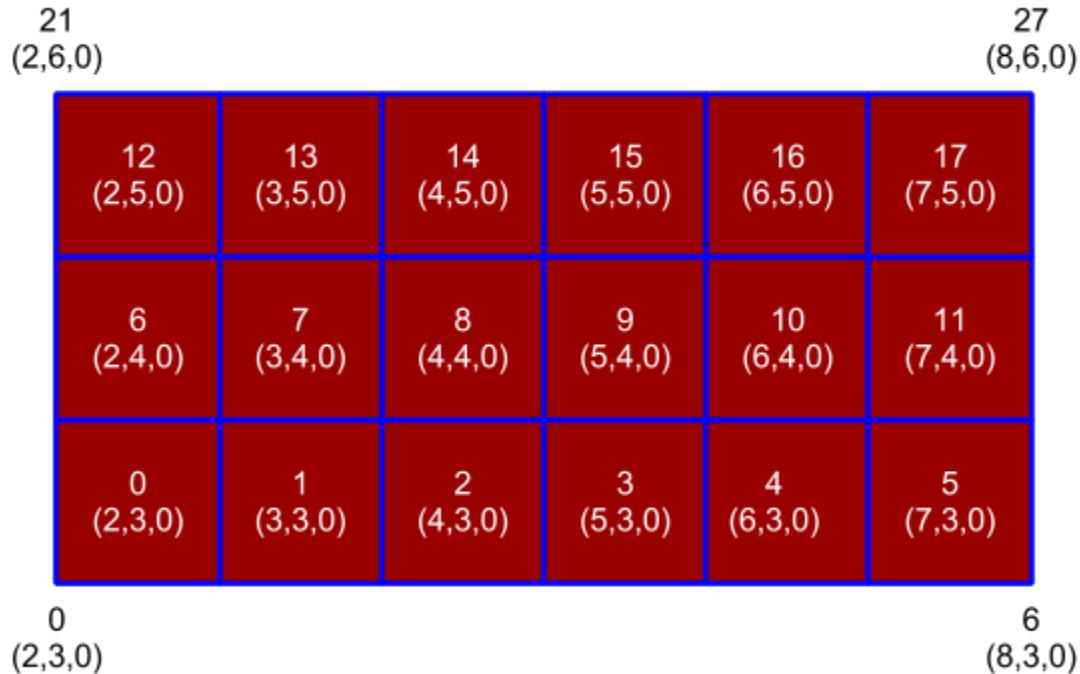


Figure 3.6: Showing the cell numbering in white and the point numbering for the corners for process 2's extents in the above figure. The first number is its index and the set of numbers in parentheses are its logical global coordinates.

vtkImageData and vtkUniformGrid

vtkImageData and *vtkUniformGrid*, which derives from *vtkImageData*, are axis-aligned grids with constant spacing between the points. *vtkUniformGrid* adds in blanking to *vtkImageData* which can be useful for overset grid types (covered later). Beyond setting the extents of the grid, the spacing between points in each direction must be specified as well as the geometric location of logical point (0,0,0), i.e. the origin of the grid. The spacing can be set as:

- void SetSpacing(double x, double y, double z)
- void SetSpacing(double x[3])

The origin of the grid can be set as:

- void SetOrigin(double x, double y, double z)
- void SetOrigin(double x[3])

This is for logical coordinate (0,0,0), even if the whole extent does not contain (0,0,0). This is all that is required to set the geometry and topology of a *vtkImageData* object. The example below shows how to create a *vtkImageData*:

```
vtkImageData* grid = vtkImageData::New();
```

```

grid->SetExtent(-10, 10, -20, 20, -10, 10);
grid->SetSpacing(2.0, 1.0, 2.0);
grid->SetOrigin(100., 100., 100.);

```

In this example, the grid will have 18,081 points and 16,000 cells. The bounds of grid will be $80 \leq x,y,z \leq 120$.

If blanking is needed then the following vtkUniformGrid methods can be used for blanking points and/or cells:

- void BlankPoint(vtkIdType ptId)
- void BlankPoint(const int i, const int j, const int k)
- void BlankCell(vtkIdType cellId)
- void BlankCell(const int i, const int j, const int k)

The above methods operate on individual grid entities. The user can create a vtkUnsignedCharArray to manually specify grid entity blanking as well. Array values of 1 indicate the grid entity is visible and values of 0 indicate the grid entity is blanked. The vtkUniformGrid methods to set the point and cell blanking arrays with a vtkUnsignedCharArray are:

- void SetPointVisibilityArray (vtkUnsignedCharArray *pointVisibility)
- void SetCellVisibilityArray (vtkUnsignedCharArray *cellVisibility)

Note that a cell is considered visible (i.e. not blanked) if both it is visible and all of its points are visible. Point visibility though is independent of any blanked cells.

vtkRectilinearGrid

The vtkRectilinearGrid class is the next more general grid representation in VTK. It is still topologically structured but geometrically it is a semi-regular array of points. The cells are still axis-aligned but the spacing between the points in each direction is specified with a vtkDataArray. This is done with the following methods:

- void SetXCoordinates(vtkDataArray *xCoordinates)
- void SetYCoordinates(vtkDataArray *yCoordinates)
- void SetZCoordinates(vtkDataArray *zCoordinates)

Note that the number of components in each of these arrays should be 1 and the length should be equal to the local process's extent in that direction, not the whole extent. An example of how to construct a rectilinear grid is:

```

vtkRectilinearGrid* grid = vtkRectilinearGrid::New();
grid->SetExtent(0, 10, 0, 20, 0, 0);
vtkFloatArray* xCoords = vtkFloatArray::New();

```

```

xCoords->SetNumberOfTuples(11);
for(vtkIdType i=0;i<11;i++)
    xCoords->SetValue(i, i*i);
vtkFloatArray* yCoords = vtkFloatArray::New();
yCoords->SetNumberOfTuples(21);
for(vtkIdType i=0;i<21;i++)
    yCoords->SetValue(i, i*i);
grid->SetXCoordinates(xCoords);
xCoords->Delete();
grid->SetYCoordinates(yCoords);
yCoords->Delete();

```

In this example, the grid has 231 points and 200 2D cells. The points are irregularly spaced in both the X and Y directions.

vtkPointSet

The remaining grids, `vtkStructuredGrid`, `vtkPolyData` and `vtkUnstructuredGrid`, are all geometrically irregular grids. Subsequently, they all derive from `vtkPointSet` which explicitly stores the point locations in a `vtkPoints` object which has a `vtkdataArray` as a data member. The first way to set the point coordinates of the grid is to create a `vtkdataArray` and use `vtkPoints'` `void SetData(vtkdataArray* coords)` method. The `vtkdataArray` object must have three components (i.e. tuple size of 3) in order to be used as the coordinates of a `vtkPointSet`. The other option for creating the points is to build up the points array directly in `vtkPoints`. The first method to call is to set the proper data precision for the coordinate representation using `void SetDataTypeToFloat()` or `void SetDataType.ToDouble()`. If the number of points are known *a priori*, the next call should be setting the number of points with `void SetNumberOfPoints(vtkIdType numberOfPoints)`. After that, the coordinates can be set with the following methods:

- `void SetPoint(vtkIdType id, float x[3])`
- `void SetPoint(vtkIdType id, double x[3])`
- `void SetPoint(vtkIdType id, double x, double y, double z)`

It is important to remember that the `Set` methods are the fastest but the reason for that is that they don't do range checking (i.e. they can overwrite memory not allocated by the array). If the number of points is not known *a priori*, then the user should allocate an estimated size with the `int Allocate (const vtkIdType size, const vtkIdType ext=1000)` method. `size` is the estimated size. When a value is inserted which exceeds the `vtkPoint` object's capacity, the capacity of the object is doubled. This used to be what the `ext` parameter was used for but that is no longer used. As with `vtkdataArray`, there are `Insert` methods to add in coordinate values to `vtkPoints` and allocate memory as needed. They are:

- `void InsertPoint (vtkIdType id, const float x[3])`
- `void InsertPoint (vtkIdType id, const double x[3])`

- void InsertPoint (vtkIdType id, double x, double y, double z)
- vtkIdType InsertNextPoint (const float x[3])
- vtkIdType InsertNextPoint (const double x[3])
- vtkIdType InsertNextPoint (double x, double y, double z)

We reiterate the warning that using InsertPoint() improperly may lead to having uninitialized data in the array. Use void Squeeze() to reclaim unused memory.

The final step is to define the vtkPoints in the vtkPointSet via the void SetPoints(vtkPoints* points) method.

vtkStructuredGrid

vtkStructuredGrid is still a topologically regular grid but is geometrically irregular. All of the major functions for creating a vtkStructuredGrid have been discussed already. The only thing left to mention is that the ordering of the coordinates in vtkPoints must match the ordering that they are iterated through. This was shown in the figure above. An example of creating a structured grid is:

```
vtkStructuredGrid* grid = vtkStructuredGrid::New();
grid->SetExtent(0, 10, 0, 20, 0, 0);
vtkPoints* points = vtkPoints::New();
points->SetNumberOfPoints(11*21);
for(int j=0;j<21;j++)
    for(int i=0;i<11;i++)
        points->SetPoint(i+j*11, i, j, 0);
grid->SetPoints(points);
points->Delete();
```

Cell Types

The two grid types left, vtkPolyData and vtkUnstructuredGrid, are the only grids that are not topologically structured and typically are also not geometrically irregular. Beside point definitions, they consist of a collection of cells of different types. Before describing into these two grid types, we discuss the types of cells that are available in VTK.

VTK supports a wide variety of cell types. As cells are implicitly stored as a type and a list of point ids in VTK's unstructured grids as a vtkCellArray (discussed later), we don't need to go into the API of vtkCell or any of its subclasses. Instead, we provide the information required for adding cells to vtkPolyData and vtkUnstructuredGrid. All that is needed is the cell type value and the canonical ordering of the points that define the cell's geometry. The two figures below give this information. Note that the cell definitions are in the vtkCellType.h header file. For a more in-depth guide to all of the cell types we refer the reader to the *VTK User's Guide* (<http://www.kitware.com/products/books/vtkguide.html>).

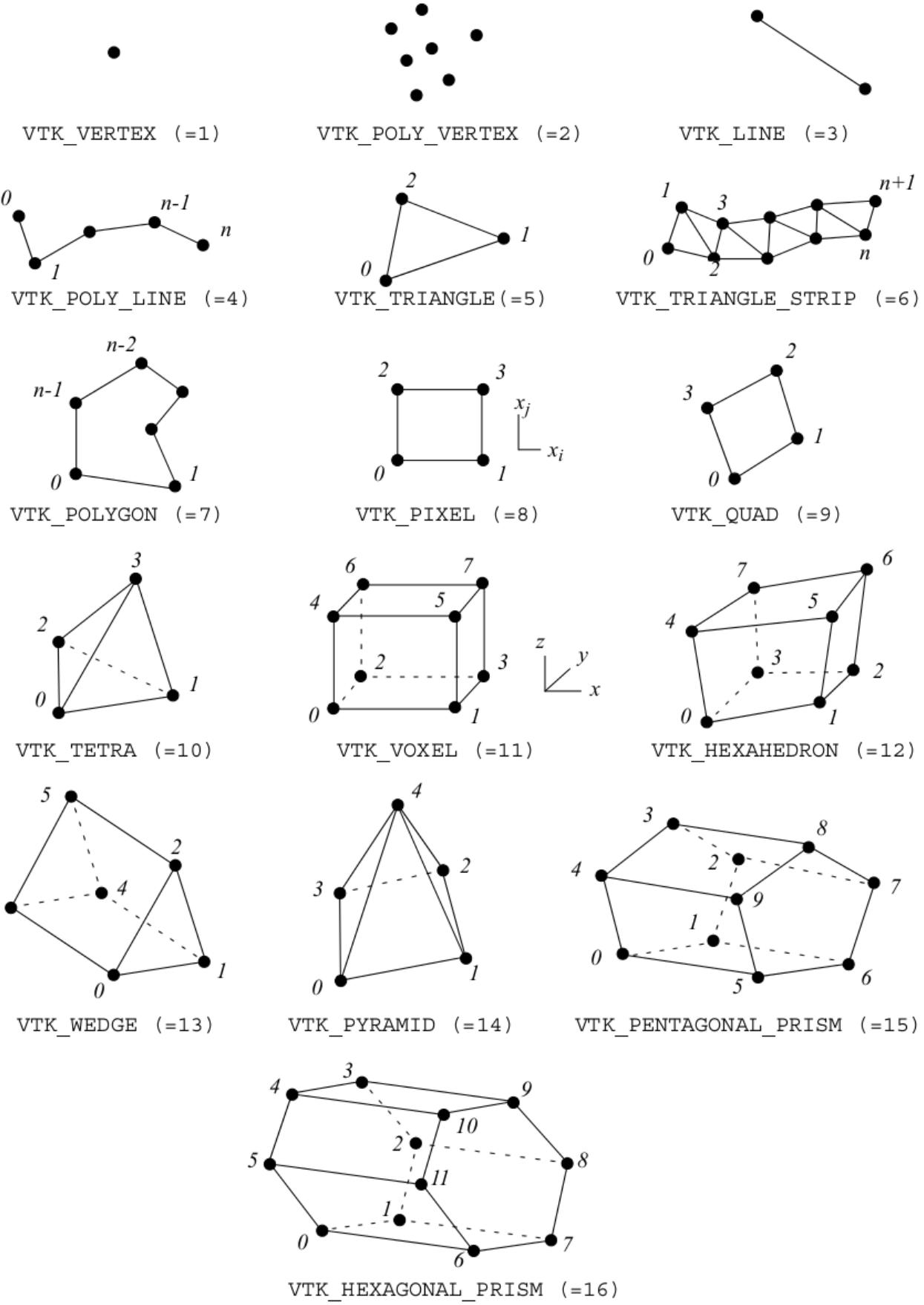


Figure 3.7: Straight-edge cells.

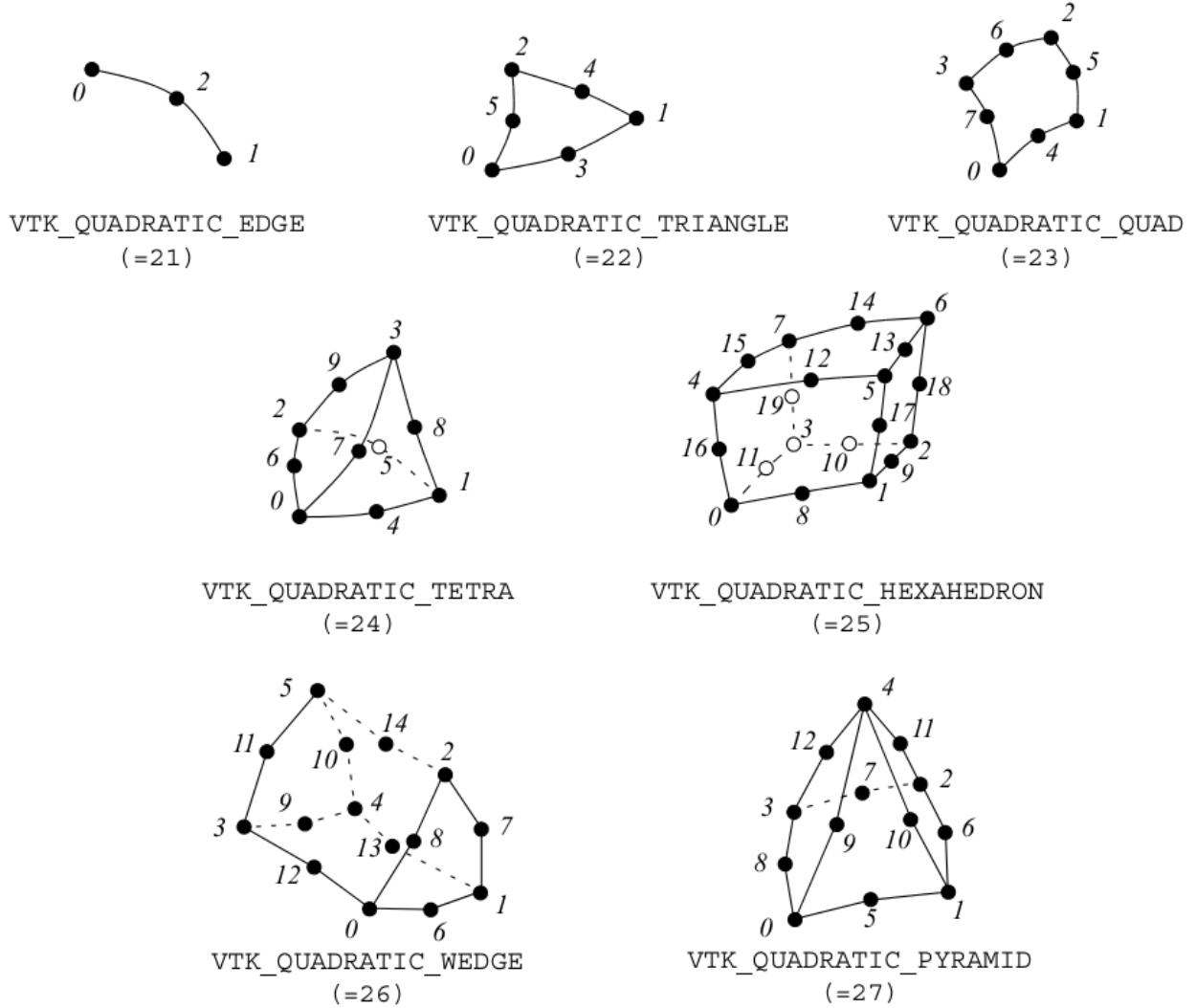


Figure 3.8: Curvi-linear edge cells.

vtkIdList

vtkIdList is an object that stores an ordered list of ids of type *vtkIdType*. They are stored in a flat array of memory. *vtkIdList* is commonly used for storing ids of points or cells. The methods of interest here are:

- void SetNumberOfIds(const *vtkIdType* number) -- set the amount of ids the object will store. The list will contain uninitialized values that must be set.
- *vtkIdType* GetNumberOfIds() -- return the number of ids stored in the list.
- void SetId(const *vtkIdType* i, const *vtkIdType* vtkid) -- set the *i*'th index of the list to value *vtkid*. Note that for efficiency this doesn't do range checking and can cause memory corruption if the proper amount of memory is not already allocated.

- void Allocate(const vtkIdType size, const int strategy) -- allocate a capacity of size for the list and set the number of ids in the list to 0. strategy is not used.
- void Reset() -- keep the current capacity of the list but mark that no ids are stored in the list.
- void InsertId(const vtkIdType i, const vtkIdType vtkid) -- set the i'th value to vtkid. This does range checking and will allocate space as needed. Using this is potentially dangerous in that it can cause uninitialized values to exist in the list.
- vtkIdType InsertNextId(const vtkIdType vtkid) -- insert vtkid at the end of the list. This method allocates space as needed. The return value is the index location where vtkid was inserted.
- void Squeeze() -- reclaim any capacity not used by the list.

The following is some code demonstrating the use of vtkIdList:

```
vtkIdList* idList = vtkIdList::New();
idList->SetNumberOfIds(1);
idList->SetId(0, 5);           // first Id is 5
idList->InsertId(1, 3);       // second Id is 3
```

vtkPolyData

vtkPolyData supports all 0D, 1D and 2D VTK cell types. It derives from *vtkPointSet* for storing point information but has its own internal data structures for storing cell information. For efficiency it stores 0D cells, 1D cells, 2D cells and triangle strips separately in *vtkCellArrays* (covered below). Even though there are separate objects for storing cells, the cells are still iterated in the order that they are inserted in. With the coordinates already set using the methods discussed previously, the next step is adding in the cells. First, memory should be allocated using void *Allocate(vtkIdType numCells, int extSize=1000)* where *numCells* is a good estimate of the number of cells that the *vtkPolyData* object will hold. Note that the actual amount of memory allocated is *4*sizeof(vtkIdType)* as each *vtkCellArray* gets allocated that amount of memory. Thus, it is important to use *Squeeze()* to reclaim the unused memory in the *vtkCellDataArrays*. Note that *extSize* is no longer used and that the capacity of each array is doubled when more memory is needed. Once this is done, the cells are inserted using:

```
int InsertNextCell(int type, int numPoints, vtkIdType* pts)
```

The *type* is the value shown in the above cell figures, *numPoints* is the number of points in the cell, and *pts* is an array of the cell's points. The return value is the cell's index in the *vtkPolyData* object. The points need to be in the proper canonical ordering shown in the above cell figures. Alternatively, the following method can be used to add cells to the grid, where now *pts* stores the cell's point ids:

```
int InsertNextCell(int type, vtkIdList* pts)
```

An example of creating a vtkPolyData with all quadrilateral cells is:

```
vtkPolyData* grid = vtkPolyData::New();
vtkPoints* points = vtkPoints::New();
points->SetNumberOfPoints(11*21);
for(int j=0;j<21;j++)
    for(int i=0;i<11;i++)
        points->SetPoint(i+j*11, i, j, 0);
grid->SetPoints(points);
points->Delete();
for(int i=0;i<10;i++)
    for(int j=0;j<20;j++)
    {
        vtkIdType ids[4] = {i+1+j*11, i+j*11, i+(j+1)*11,
                            i+1+(j+1)*11};
        grid->InsertNextCell(9, 4, ids);
    }
```

vtkUnstructuredGrid

vtkUnstructuredGrid supports all VTK cell types. It also derives from vtkPointSet for storing point information. It uses a single vtkCellArray to store all of the cells. As for vtkPolyData, we recommend using void Allocate(vtkIdType size) to pre-allocate memory for storing cells. In this case though we recommend a value of numCells*(numPointsPerCell+1) for the size. Similarly, for inserting cells, either the following methods should be used:

- vtkIdType InsertNextCell(int type, vtkIdType numPoints, vtkIdType* pts)
- vtkIdType InsertNextCell(int type, vtkIdList* pts)

These are the same as for vtkPolyData. Similarly, the example for creating points and cells for a vtkUnstructuredGrid is the same as for vtkPolyData.

vtkCellArray

The functions listed above for adding cells to either vtkUnstructuredGrid or vtkPolyData are the simplest to use. The problem with this approach is that it won't reuse existing memory for storing the cell connectivity arrays. Internally in vtkUnstructuredGrid and vtkPolyData, this information is stored in vtkCellArray objects. vtkCellArray is a supporting object that explicitly represents cell connectivity using a vtkIdTypeArray. The data in the array is stored in the form: (n,id1,id2,...,idn, n,id1,id2,...,idn, ...) where n is the number of points in the cell, and id is a zero-offset index into the vtkDataArray in vtkPoints. This is shown in the figure below. Advantages of this data structure are its compactness, simplicity, and easy interface to external data. However, it is totally inadequate for random access. We include this information for completeness but unless a user's native simulation data structure matches the vtkCellArray form, we suggest that users add cells to vtkPolyData and vtkUnstructuredGrid through the interfaces of those classes and not by creating a vtkCellArray and directly populating it with data. Note that if vtkCellArray is

directly used with existing allocated memory, the user can configure Catalyst to have vtkIdType match the native type that the simulation code uses to store ids.

The directions for using an existing array with vtkCellArray is to first create a vtkIdTypeArray and use the SetArray() method to reuse existing memory. Next, use void SetCells(vtkIdType numberOfRowsCells, vtkIdTypeArray* cells) to use the cells array in vtkCellArray. The next steps depend on which grid type is being used.

For a vtkPolyData, as we mentioned above, it actually has four vtkCellArrays to store its cells, one for vertex and polyvertex cells, one for line and polyline cells, one for triangle, quadrilateral, polygonal and pixel cells, and one for triangle strips. To set the cell arrays, the following methods should be used in the given order:

1. void SetVerts(vtkCellArray* v)
2. void SetLines(vtkCellArray* l)
3. void SetPolys(vtkCellArray* p)
4. void SetStrips(vtkCellArray* s)
5. void BuildCells()

Any of the above Set methods can be skipped if there are no corresponding cells of the proper type. This order should be followed to ensure that the ordering of the cells matches any cell data attributes that exist. The last method builds up the full cell information that enables random access to a vtkPolyData's cells.

For vtkUnstructuredGrids, we assume that there aren't any polyhedral cells. In this case the following methods can be used:

- void SetCells(int type, vtkCellArray* cells)
- void SetCells(int* types, vtkCellArray* cells)
- void SetCells(vtkUnsignedCharArray *cellTypes, vtkIdTypeArray *cellLocations, vtkCellArray *cells)
- void SetCells(vtkUnsignedCharArray *cellTypes, vtkIdTypeArray *cellLocations, vtkCellArray *cells, vtkIdTypeArray *faceLocations, vtkIdTypeArray *faces)

The first method is when all of the cell types are the same and the second is for heterogeneous cell type grids where types is an array length equal to the number of cells. These two methods will still build up the information needed for random access to the cells. The third method contains the cell types in the cellTypes array and cellLocations contains the array index of the vtkCellArray to find a cell's point ids. The fourth method contains the cell types in the cellTypes array and cellLocations contains the array index of the vtkCellArray to find a cell's point ids and can pass in NULL for the last 2 arguments when no polyhedra cell types are used. For all four SetCell() methods above, the cells argument corresponds to the vtkCellArray shown in the figure below. The cellTypes argument in the last two methods corresponds to the

`vtkUnsignedCharArray` and the `cellLocations` argument corresponds to the `vtkIdTypeArray` in the figure below.

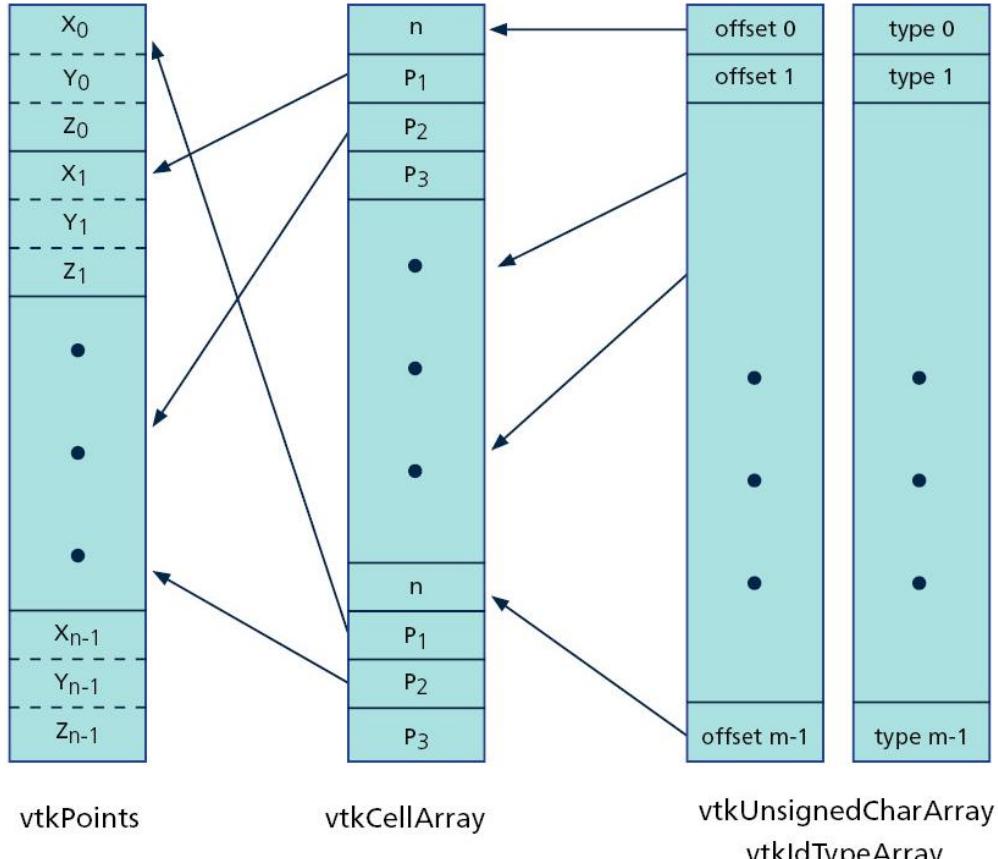


Figure 3.9: Internal `vtkUnstructuredGrid` data structures for storing cells' connectivities.

An example is shown below for creating the cell topology data structures for an unstructured grid. Note that it assumes the points have already been added to the grid.

```
// create the cell data structures
vtkCellArray* cellArray = vtkCellArray::New();
vtkIdTypeArray* offsets = vtkIdTypeArray::New();
vtkUnsignedCharArray* types = vtkUnsignedCharArray::New();
vtkIdType ids[8];
// create a triangle
ids[0] = 0; ids[1] = 1; ids[2] = 2;
cellArray->InsertNextCell(3, ids);
offsets->InsertNextValue(0);
types->InsertNextValue(VTK_TRIANGLE);
// create a quad
ids[0] = 0; ids[1] = 1; ids[2] = 2; ids[3] = 3;
```

```

cellArray->InsertNextCell(4, ids);
offsets->InsertNextValue(4);
types->InsertNextValue(VTK_QUAD);
// create a tet
ids[0] = 0; ids[1] = 1; ids[2] = 2; ids[3] = 4;
cellArray->InsertNextCell(4, ids);
offsets->InsertNextValue(9);
types->InsertNextValue(VTK_TETRA);
// create a hex
ids[0] = 0; ids[1] = 1; ids[2] = 2; ids[3] = 3;
ids[4] = 4; ids[5] = 5; ids[6] = 6; ids[7] = 7;
cellArray->InsertNextCell(8, ids);
offsets->InsertNextValue(14);
types->InsertNextValue(VTK_HEXAHEDRON);
// add the cell data to the unstructured grid
grid->SetCells(types, offsets, cellArray);
types->Delete();
offsets->Delete();
cellArray->Delete();

```

Field Data

Once the grids are created, the next step is to associate attributes with the points and/or cells of the grid. The following figure shows a pseudo-coloring of a point data array and a cell data array. Note that the point data will have continuous values as long as the points are properly associated with the cells. In general, cell data will be discontinuous unless there is a constant value for the field.

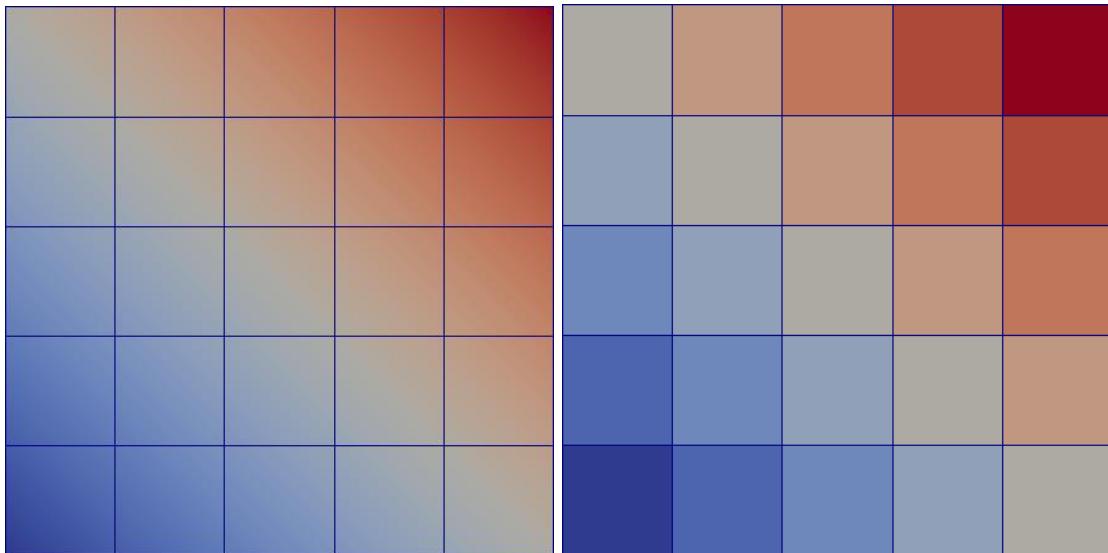


Figure 3.10: Pseudo-coloring of point data (left image) and cell data (right image).

The main class for field data is `vtkFieldData` which is a container to store `vtkDataArrays`. The arrays are stored in a flat array of memory and accessed either by their index location or the name of the `vtkdataArray`. The main methods here is `int AddArray(vtkAbstractArray* array)`. This method appends an array to the `vtkFieldData` object's list of arrays unless an array with that name already exists. If an array with that name already exists then it replaces that with the passed in `vtkdataArray`. The return value is the index in the list where the array was inserted. Note that every `vtkDataObject` has a `vtkFieldData` member object which can be accessed through the `vtkFieldData* GetFieldData()` method. This can be used for storing meta-data about the `vtkDataObject` and the arrays stored in the `vtkFieldData` do not have to have the same number of tuples.

If we want to store arrays where the tuples are associated with either points or cells, we use `vtkPointData` and `vtkCellData`, respectively. Both of these derive from `vtkFieldData`. Every `vtkDataSet` has both a `vtkPointData` and a `vtkCellData` object and they are accessed with `vtkPointData* GetPointData()` and `vtkCellData* GetCellData()`. Note that the arrays in either of these objects should have the number of tuples matching the number of grid entities of the corresponding type. There is no explicit check when inserting arrays into either of these but many filters will give warnings and/or fail if this condition isn't met.

The following snippet of code demonstrates how arrays are added to point data and cell data.

```

vtkDoubleArray* pressure = vtkDoubleArray::New();
pressure->SetNumberOfTuples(grid->GetNumberOfPoints());
pressure->SetName("pressure");
<set values for pressure>
grid->GetPointData()->AddArray(pressure);
pressure->Delete();
vtkFloatArray* temperature = vtkFloatArray::New();
temperature->SetName("temperature");
temperature->SetNumberOfTuples(grid->GetNumberOfCells());
grid->GetCellData()->AddArray(temperature);
temperature->Delete();

```

Multi-Block Data Sets

So far we've covered all of the main data sets and how to define attributes over them (i.e. the point and cell data). For many situations though we will want to use multiple data sets to represent our simulation data structures. Examples include overlapping grids (e.g. AMR) or when a single data set type isn't appropriate for storing the cell topology (e.g. using a `vtkUniformGrid` and a `vtkUnstructuredGrid`). The main class for this is the `vtkCompositeDataSet` class. This is an abstract class that is intended to simplify the way to iterate through the `vtkDataSets` stored in the different concrete derived classes. There are two main types of composite data sets. The first type is for AMR type grids where only `vtkUniformGrid` data sets are used to discretize the domain. These types of composite data sets have support for automatically stitching the grids together through blanking. The two classes for AMR grids are

`vtkOverlappingAMR` and `vtkNonOverlappingAMR` which both derive from `vtkUniformGridAMR`. The second composite data set type supports all grids that derive from `vtkDataSet` but require any blanking needed for overlapping grids to be taken care of explicitly. The two classes for these are `vtkMultiBlockDataSet` and `vtkMultiPieceDataSet` and both derive from `vtkDataObjectTree`. Because `vtkCompositeDataSet` derives from `vtkDataObject` it has a `vtkFieldData` object that can be accessed by the `vtkFieldData* GetFieldData()` method. This can be useful for storing meta-data.

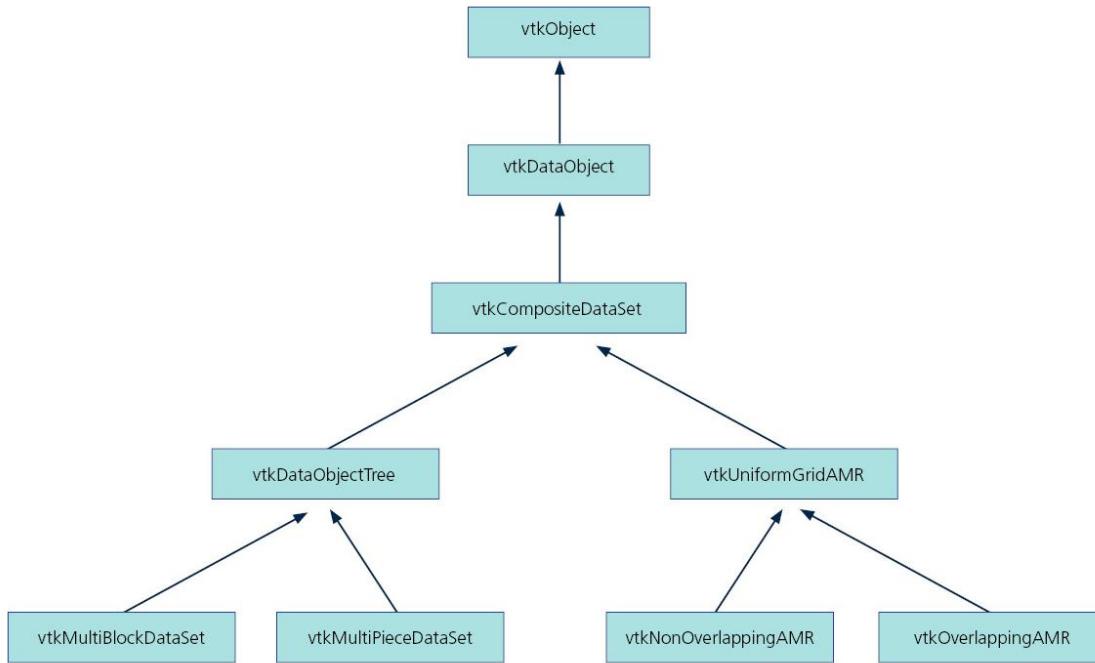


Figure 3.11: Class hierarchy for VTK composite data sets.

`vtkMultiBlockDataSet`

`vtkMultiBlockDataSet` is the most general of the concrete implementations of `vtkCompositeDataSet`. Each block can contain either any `vtkDataSet` type or any `vtkCompositeDataSet` type. This leads to a hierarchy of data sets that can be stored in a `vtkMultiBlockDataSet`. An example of this is shown below. The `vtkMultiBlockDataSet` can be used recursively to store blocks at different levels. For each level that a `vtkMultiBlockDataSet` is used, the adaptor should set the amount of blocks at that level using the void `SetNumberOfBlocks(unsigned int numBlocks)` method. In parallel, the tree hierarchy must match on each process but leaves of the tree are only required to be non-empty on at least one process. If the leaf is a `vtkDataSet` then it should be non-empty on exactly one process. To set a sub-block of a `vtkMultiBlockDataSet`, use the void `SetBlock(unsigned int blockNumber, vtkDataObject* dataObject)` method. This method assigns `dataObject` into the `blockNumber` location of its direct children.

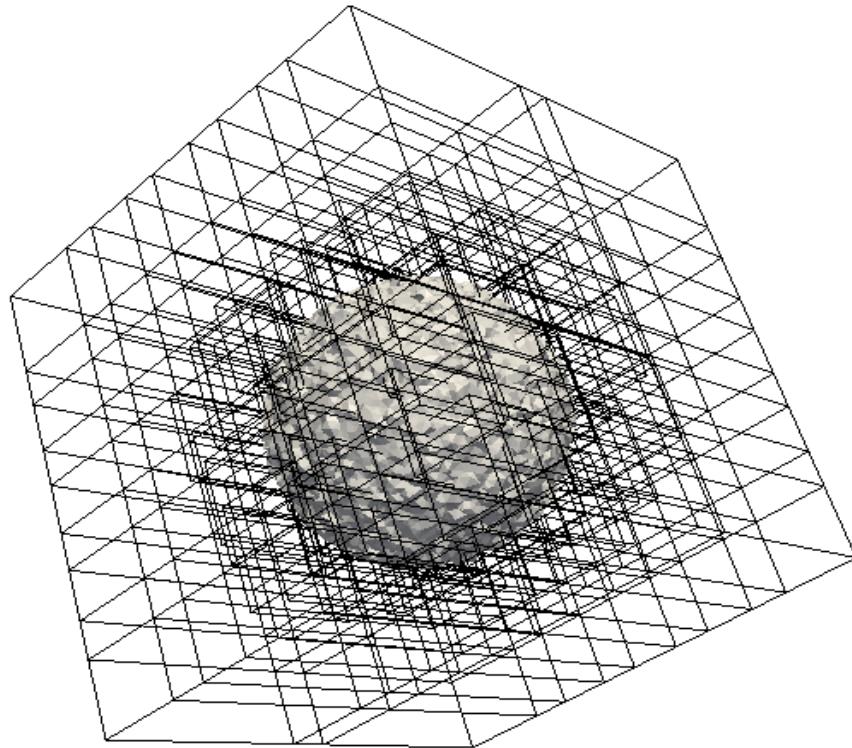


Figure 3.12: Multi-block data set where the outlines are for blocks with `vtkUniformGrids` and the interior surface is a `vtkUnstructuredGrid`.

vtkMultiPieceDataSet

A `vtkMultiPieceDataSet` class groups data sets that span multiple processes together into a single logical sub-block of a `vtkCompositeDataSet`. The purpose is to help avoid some of the rigidity of concrete instances of `vtkDataSets` while maintaining their logical grouping together. One example of this is the rigidity of partitioning topologically regular grids into logically rectangular blocks of cells. A process may have multiple sub-blocks of the topologically regular grid such that when trying to combine them would cause the combined blocks to not be able to be stored in a topologically convex sub-block. Another use for the `vtkMultiPieceDataSet` is for a sub-block that is a partitioned `vtkDataSet`. In this case they are logically grouped together but can't be stored in the same sub-block of a `vtkMultiBlockDataSet` since each process will think that it contains the entire `vtkDataSet`. The `vtkMultiPieceDataSet` is a flat structure with all of its children being the same grid type. The methods that are used to set the pieces of the `vtkMultiPieceDataSet` are:

- `void SetNumberOfPieces(unsigned int numPieces)` -- sets the number of pieces to be contained in the `vtkMultiPieceDataSet`. This should be the same value on each process. When there is a single piece per process the value of `numPieces` will be the number of processes.
- `void SetPiece(unsigned int pieceNumber, vtkDataObject* piece)` -- sets piece for the `pieceNumber` location. Note that piece must be a `vtkDataSet` even though the method

signature allows a vtkDataObject to be passed in.

Note that vtkMultiPieceDataSet is intended to be included in other composite data sets e.g. vtkMultiBlockDataSet or vtkOverlappingAMR. There is no writer in ParaView that can handle a vtkMultiPieceDataSet as the main input so adaptors should nest any multi-piece data sets in a separate composite data set. An example of creating a multi-piece data set where we only partition the grid in the x-direction is shown below:

```
vtkImageData* imageData = vtkImageData::New();
imageData->SetSpacing(1, 1, 1);
imageData->SetExtent(0, 50, 0, 100, 0, 100);
int mpiSize = 1;
int mpiRank = 0;
MPI_Comm_rank(MPI_COMM_WORLD, &mpiRank);
MPI_Comm_size(MPI_COMM_WORLD, &mpiSize);
vtkMultiPieceDataSet* multiPiece = vtkMultiPieceDataSet::New();
multiPiece->SetNumberOfPieces(mpiSize);
imageData->SetOrigin(50*mpiRank, 0, 0);
multiPiece->SetPiece(mpiRank, imageData);
imageData->Delete();
vtkMultiBlockDataSet* multiBlock = vtkMultiBlockDataSet::New();
multiBlock->SetNumberOfBlocks(1);
multiBlock->SetBlock(0, multiPiece);
multiPiece->Delete();
```

vtkUniformGridAMR

The vtkUniformGridAMR class is used to deal with AMR grids and to automate the process of nesting the grids and blanking the appropriate points and cells, if blanking is needed. The first call to use in constructing a vtkUniformGridAMR or any of its derived classes is the void Initialize(int numLevels, const int* blocksPerLevel) method. This specifies how many levels there will be in the AMR data object and how many blocks in each level. Note that blocksPerLevel needs to have at least numLevels values. The values passed into Initialize() need to match on every process. Other class methods which should be used in the order listed are:

- void SetDataSet(unsigned int level, unsigned int idx, vtkUniformGrid *grid) -- Once the uniform grid has been created it can be added to the vtkUniformGridAMR with this method. Note that coarsest level is 0 and that idx is the index that grid is to be inserted at for the specified level (i.e. $0 \leq idx < blocksPerLevel[level]$, where blocksPerLevel was passed in the Initialize() method).
- void SetGridDescription(int gridDescription) -- The values of gridDescription specify what geometric coordinates the uniform grids are meant to discretize. For example, VTK_XYZ_GRID indicates that the vtkUniformGridAMR discretizes a volume (the default

value) and VTK_XZ_PLANE indicates that the vtkUniformGridAMR discretizes an area in the XZ plane. The definitions of appropriate values for gridDescription are in vtkStructuredData.h.

vtkOverlappingAMR

The vtkOverlappingAMR grid is for when the set of uniform grids overlap in space and require blanking in order to determine which grid is used to discretize the domain and for specifying attributes over. This is the appropriate composite data set for Berger-Colella type AMR grids. Because of this hierarchy there is the notion of a global origin of the composite data set. This is set with the void SetOrigin(double* origin) method. A key point for vtkOverlappingAMR composite data sets is that the spacing is unique to each level and must be maintained by the vtkUniformGrids that are used to discretize the domain at that level of the composite data set. This is done with the following method:

```
void SetSpacing(unsigned int level, const double spacing[3])
```

This needs to be called for each level with level 0 being the coarsest level. spacing is the distance between consecutive points in each Cartesian direction. In addition to the spacing of each level, the nested hierarchy must also be built up. vtkAMRBox is a helper class used to determine the hierarchy of the uniform grids and their respective blanking. The main thing to keep in mind is that there is both a global origin value as well as an origin value for each block. Additionally, vtkAMRBox needs the dimensions of each block and the spacing for each level. The dimensions are the number of points in each direction. The main function of interest for vtkAMRBox is the constructor which passes in all of the necessary values:

- vtkAMRBox (const double *origin, const int *dimensions, const double *spacing, const double *globalOrigin, int gridDescription=VTK_XYZ_GRID) -- Here, origin is the minimum bounds of the vtkUniformGrid that the box represents, dimensions is the number of points in each of the grid's logical directions, spacing is the distance between points in each logical direction, globalOrigin is the minimum bounds of the entire composite data set and gridDescription specifies the logical coordinates that the grid discretizes.

Once the vtkAMRBox is created for a vtkUniformGrid, the following methods should be used:

- void SetAMRBox (unsigned int level, unsigned int id, const vtkAMRBox &box) -- level is the hierarchical level that box belongs to and id is the index at that level for the box. Note that similar to the SetDataSet() method, valid values of id are between 0 and up to but not including the global number of vtkUniformGrids at that level.
- void SetAMRBlockSourceIndex (unsigned int level, unsigned int id, int sourceId) -- This method is very similar to the SetDataSet() method but instead of specifying the data set for a given level and index at that level, it specifies the sourceId in the global composite data set hierarchy. This is the overall composite index of the data set and can be set as the total number of data sets that exist at coarser levels plus the number of data sets

that have a lower index but are at the same level as the given data set.

After this has been done for each data set, the void GenerateParentChildInformation() method needs to be called. This method generates the proper relations between the blocks and the blanking inside of each block. After this has been done, the uniform grids should be added with SetDataSet(). An example of creating a vtkOverlappingAMR composite data set is included below to help elucidate how all of this comes together.

```
int numberOfLevels = 3;
int blocksPerLevel[3] = {1, 1, 1};
vtkOverlappingAMR* amrGrid = vtkOverlappingAMR::New();
amrGrid->Initialize(numberOfLevels, blocksPerLevel);
amrGrid->SetGridDescription(VTK_XYZ_GRID);
double origin[] = {0,0,0};
double level0Spacing[] = {4, 4, 4};
double level1Spacing[] = {2, 2, 2};
double level2Spacing[] = {1, 1, 1};
amrGrid->SetOrigin(origin);
int level0Dims[] = {25, 25, 25};
vtkAMRBox level0Box(origin, level0Dims, level0Spacing, origin,
                     VTK_XYZ_GRID);
int level1Dims[] = {20, 20, 20};
vtkAMRBox level1Box(origin, level1Dims, level1Spacing, origin,
                     VTK_XYZ_GRID);
int level2Dims[] = {10, 10, 10};
vtkAMRBox level2Box(origin, level2Dims, level2Spacing, origin,
                     VTK_XYZ_GRID);
amrGrid->SetSpacing(0, level0Spacing);
amrGrid->SetAMRBox(0, 0, level0Box);
amrGrid->SetSpacing(1, level1Spacing);
amrGrid->SetAMRBox(1, 0, level1Box);
amrGrid->SetSpacing(2, level2Spacing);
amrGrid->SetAMRBox(2, 0, level2Box);
amrGrid->GenerateParentChildInformation();

// the highest level grid
vtkUniformGrid* level0Grid = vtkUniformGrid::New();
level0Grid->SetSpacing(level0Spacing);
level0Grid->SetOrigin(0, 0, 0);
level0Grid->SetExtent(0, 25, 0, 25, 0, 25);
amrGrid->SetDataSet(0, 0, level0Grid);
level0Grid->Delete();
// the mid-level grid
vtkUniformGrid* level1Grid = vtkUniformGrid::New();
level1Grid->SetSpacing(level1Spacing);
```

```

level1Grid->SetExtent(0, 20, 0, 20, 0, 20);
amrGrid->SetDataSet(1, 0, level1Grid);
level1Grid->Delete();
// the lowest level grid
vtkUniformGrid* level2Grid = vtkUniformGrid::New();
level2Grid->SetSpacing(level2Spacing);
level2Grid->SetExtent(0, 10, 0, 10, 0, 10);
amrGrid->SetDataSet(2, 0, level2Grid);
level2Grid->Delete();

```

vtkNonOverlappingAMR

The vtkNonOverlappingAMR grid is for the case of groups of vtkUniformGrids that do not overlap but can have grids that are associated with different levels of the hierarchy. Note that the adaptor could arbitrarily assign all vtkUniformGrids to be at the coarsest level but this would remove any hierarchical information that may be useful by storing the grids at different levels. The methods of interested for constructing non-overlapping composite data sets are all in its vtkUniformGridAMR superclass. An example is included below which demonstrates the construction of a vtkNonOverlappingAMR grid.

```

int numberOfLevels = 3;
int blocksPerLevel[3] = {1, 2, 1};
vtkNonOverlappingAMR* amrGrid = vtkNonOverlappingAMR::New();
amrGrid->Initialize(numberOfLevels, blocksPerLevel);
// the highest level grid
vtkUniformGrid* level0Grid = vtkUniformGrid::New();
level0Grid->SetSpacing(4, 4, 4);
level0Grid->SetOrigin(0, 0, 0);
level0Grid->SetExtent(0, 10, 0, 20, 0, 20);
amrGrid->SetDataSet(0, 0, level0Grid);
level0Grid->Delete();
// the first mid-level grid
vtkUniformGrid* level1Grid0 = vtkUniformGrid::New();
level1Grid0->SetSpacing(2, 2, 2);
level1Grid0->SetOrigin(40, 0, 0);
level1Grid0->SetExtent(0, 8, 0, 20, 0, 40);
amrGrid->SetDataSet(1, 0, level1Grid0);
level1Grid0->Delete();
// the second mid-level grid
vtkUniformGrid* level1Grid1 = vtkUniformGrid::New();
level1Grid1->SetSpacing(2, 2, 2);
level1Grid1->SetOrigin(40, 40, 0);
level1Grid1->SetExtent(0, 40, 0, 20, 0, 40);
amrGrid->SetDataSet(1, 1, level1Grid1);
level1Grid1->Delete();

```

```

// the lowest level grid
vtkUniformGrid* level2Grid = vtkUniformGrid::New();
level2Grid->SetSpacing(1, 1, 2);
level2Grid->SetOrigin(0, 0, 0);
level2Grid->SetExtent(56, 120, 0, 40, 0, 40);
amrGrid->SetDataSet(2, 0, level2Grid);
level2Grid->Delete();

```

Grid Partitioning

We have briefly covered partitioning the grid for parallel computing already but it is an important enough of a topic that it deserves to be discussed in a complete manner. The driving motivation here is to use the existing partitioning of the simulation grid. We assume that most filters will scale well with the existing grid partitioning supplied by the simulation. VTK's data sets and composite data sets cover a wide enough range of use cases that it should be rare that interprocess communication will be necessary to migrate simulation grid data in order to properly create partitioned VTK grid data. VTK does assume a cell-based partitioning of the grid where a cell is uniquely represented on a single process.

For topologically structured grids partitioning is done via extents as discussed above. For topologically regular grids the developer has two choices for partitioning the grid. The first is using the grid that derives from `vtkDataSet` and the second is using a `vtkMultiBlockDataSet` for each partition of the grid. Due to the rigidity of the local extents and how they interact with the VTK pipeline, we recommend using the `vtkMultiBlockDataSet` approach where each process's partition of the data set is inserted as a block in the composite data set. This allows the extents to be independent for each block. For situations where a process's partitioning of a topologically regular grid is not convex in the logical coordinates, a process can contribute multiple blocks to a `vtkMultiBlockDataSet` such that each block is a convex logical subset of the total grid. This is also useful for situations where the simulation data is chunked into smaller blocks to decrease cache misses during runs. An example of a multi-block data set with one block per process is shown below where `numberOfProcesses` is the number of MPI processes and `rank` is the MPI rank of a process:

```

vtkMultiBlockDataSet multiBlock = vtkMultiBlockDataSet::New();
multiBlock->SetNumberOfBlocks(numberOfProcesses);
multiBlock->SetBlock(rank, dataSet);

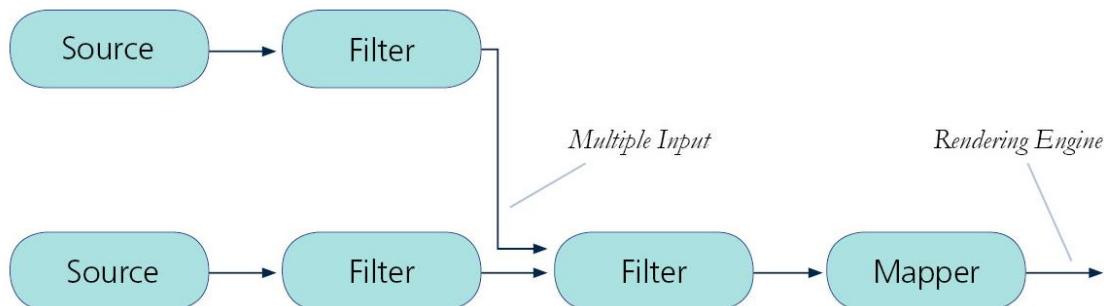
```

For `vtkPolyData` and `vtkUnstructuredGrid`, partitioning is straightforward when using cell-based partitionings of the simulation's grid. Ghost cells should not be added to the VTK grids. For point-based partitionings of the grid, there is typically an overlap of a single layer of cells that exists on multiple process. These cells need to be assigned uniquely to a single process for VTK grids and partitionings. Similar to topologically regular grids, `vtkPolyData` and `vtkUnstructuredGrid` data sets can also be inserted into a `vtkMultiBlockDataSet` to allow for multiple blocks per process.

The VTK Pipeline

For a full description of VTK's pipeline architecture, we refer the reader to the *VTK User's Guide*. We include a summary description of this architecture since it is key to understanding how Catalyst outputs are generated. From a high level, Catalyst simply defines and configures VTK pipelines that are executed at defined points in a simulation run.

VTK uses a data flow approach to transform information into desired forms. The desired form may be derived quantities, subsetted quantities and/or graphical information. The transformations are performed by filters in VTK. These filters take in data and perform operations based on a set of input parameters to the filter. Most VTK filters do a very specific operation but by chaining multiple filters together a wide variety of operations can be done to transform the data. A filter that doesn't have any input from a separate filter is called a source and a filter that doesn't send its output to any other filters is called a sink. An example of a source filter would be a file reader and an example of a sink filter would be a file writer. We call this set of connected filters the pipeline. For Catalyst, the adaptor acts as the source filter for all pipelines. An example of this is shown below.



The pipeline's task is to configure, execute, and pass `vtkDataObjects` between the filters. The pipeline can be viewed as a directed, acyclic graph. Some key features of VTK's pipeline are:

- Filters are not allowed to modify their input data objects.
- It is demand driven meaning that filters only execute when something downstream requests that they execute.
- A filter will only re-execute if a request changed or something upstream changed.
- Filters can have multiple inputs or outputs.
- Filters can send their output to multiple separate filters.

This affects Catalyst in several key ways. The first being that the adaptor can use existing memory when building the VTK data structures. The reason for this is that the VTK filter which operates on that data will either create a new copy if the data needs to change or reuse the existing data through reference counting if the data won't be modified. The second key way is that the pipeline will only be re-executed when it is specifically requested

The Catalyst API

In this section we discuss how the adaptor passes information back and forth between the simulation code and Catalyst. This information can be broken up into three areas:

1. VTK data objects
2. Pipelines
3. Control information

The VTK data objects are the information containing the input to the pipelines. The pipelines specify what operations to perform on the data and how to output the results. The control information specifies when each pipeline should execute and what information is needed in the VTK data objects in order for the pipelines to be able to execute properly.

For most simulation codes, the interface between the main simulation code and the adaptor will only involve three function calls. The first call initializes Catalyst and the pipelines; the second call performs any requested co-processing; and the third call finalizes Catalyst. This was shown in the first code snippet near the beginning of Section 3. The rest of the interface between the simulation code and Catalyst is all contained in the adaptor code. This allows a small footprint in the main code base which makes it simple to build the simulation code both with and without linking to Catalyst.

High-Level View

Before diving into the details of the API, we want to describe the flow of information and its purpose to help give a higher level of understanding of how the pieces work together. The first step is initialization which sets up the Catalyst environment and creates the pipelines that will be executed later on. This is typically called near the beginning of the simulation shortly after MPI is initialized. The next step is to execute the pipelines if needed. This is usually done at the end of each time step update. The final step is finalizing Catalyst and is usually done right before MPI is finalized.

The first and last steps are pretty simple but the middle step has a lot happening underneath the covers. Essentially, the middle step queries the pipelines to see if any of them need to be executed. If they don't then it immediately returns control back to the simulation code. In our experience, this is nearly instantaneous. This must be fast since we expect many calls here and don't want to waste valuable compute cycles. If one or more pipelines need to re-execute, then the adaptor needs to update the VTK data objects representing the grid and attribute information and then execute the desired pipelines. Depending on the amount of work that needs to be done by the filters in the pipeline, this can take a wide range of time. Once all of the pipelines that need to be re-executed finish, control is returned back to the simulation code.

Class API

The main classes of interest for the Catalyst API are `vtkCPPProcessor`, `vtkCPDataDescription`, `vtkCPIInputDataDescription`, `vtkCPPipeline` and the derived classes that are specialized for Python. When Catalyst is built with Python support, all of these classes are Python wrapped as well.

vtkCPPProcessor

vtkCPPProcessor is responsible for managing the pipelines. This includes storing them, querying them to see if they need to be executed and executing them. The methods of interest for *vtkCPPProcessor* are:

- void Initialize() -- initializes the object and sets up Catalyst. This should be done after MPI_Init() is called.
- void Finalize() -- releases all resources used by Catalyst. This should be done before MPI_Finalize() is called.
- int AddPipeline(vtkCPPipeline* pipeline) -- add in a pipeline to be executed at requested times.
- int RequestDataDescription(vtkCPDataDescription* description) -- determine if for a given description if any data pipelines should be executed. The return value is 1 if a pipeline needs to be executed and 0 otherwise. For this call, description should have the time and time step set and the identifier for the inputs that are available (i.e. vtkCPDataDescription::AddInput(const char*)).
- int CoProcess(vtkCPDataDescription* description) -- executes the proper pipelines based on information in description. At this call the vtkDataObject representing the grids and fields should have updated to the current time step and added to description, in addition to the values set before the call to RequestDataDescription().

The above methods are usually sufficient for basic functionality. If the adaptor code wants to manually remove pipelines during the simulation runs, the following methods can be used:

- int GetNumberOfPipelines() -- get the number of existing pipelines.
- vtkCPPipeline* GetPipeline(int i) -- get the ith pipeline.
- void RemoveAllPipelines() -- remove all of the existing pipelines from the co-processor and prevents them from being executed.
- void RemovePipeline(vtkCPPipeline* pipeline) -- removes a pipeline and prevents it from being executed.

vtkCPPProcessor doesn't add any extra functions to the public API. Its main purpose is to initialize the use of Python for co-processing. It is required to be used if the user wants to run a Python co-processing pipeline. If the pipelines are all implemented in C++ then *vtkCPPProcessor* should be used.

vtkCPPipeline and vtkCPPythonScriptPipeline

vtkCPPipeline is the abstract base class for storing VTK pipelines that will be executed by *vtkCPPProcessor*. *vtkCPPythonScriptPipeline* is a concrete derived class that takes in a Python script that stores the VTK pipeline. The methods of interest are:

- int RequestDataDescription(vtkCPDataDescription *description) -- given description, return 1 if this pipeline needs to execute and 0 otherwise.

- int CoProcess(vtkCPDataDescription *description) -- execute the pipeline stored by this object and return 1 for success and 0 for failure.

Note that both of these methods are abstract in vtkCPPipeline. For pipelines coded in C++, we expect the user to create them in a class that derives from vtkCPPipeline. They are implemented in vtkCPPythonScriptPipeline for Python pipelines. For vtkCPPythonScriptPipeline, the only other method of interest is:

- int Initialize (const char *fileName) -- initialize the pipeline with the file name of a Python script.

vtkCPDataDescription

The vtkCPDataDescription class is meant to store information that gets passed between the adaptor and the pipelines. Some of the information is provided by the adaptor and is used by the pipeline and other parts of the information is provided by the pipeline and used by the adaptor. The methods for the information provided by the adaptor to the pipelines are:

- void SetTimeData(double time, vtkIdType timeStep) -- sets the time step and current simulation time. This needs to be called before each call to vtkCPProcessor::RequestDataDescription().
- void AddInput (const char *gridName) -- add name keys for input grids produced by the simulation. For most use cases, the adaptor will provide a single input grid to Catalyst and the convention is that it is called "input". Naming the inputs is needed for situations where the adaptor provides multiple input grids and each grid should be treated independently. An example of this is fluid-structure interaction simulations where separate grids may discretize each domain and each grid contains different field attributes. This is demonstrated in the figure below. This needs to be called before vtkCPProcessor::RequestDataDescription() is called but only needs to be called once per simulation time step.
- void SetForceOutput (bool on) -- this allows the adaptor to force all of the pipelines to execute by calling this method with on set to true. By default it is false and it is reset after each call to vtkCPProcessor::CoProcess(). In general, the adaptor won't know when a pipeline will want to execute but in certain situations the adaptor may realize that some noteworthy event has occurred. An example of this may be some key simulation feature occurs or the last time step of the simulation. In this situation the adaptor can use this method to make sure that all pipelines execute. Note that user implemented classes that derive from vtkCPPipeline should include logic for this. If this is used it should be called before calling vtkCPProcessor::RequestDataDescription().

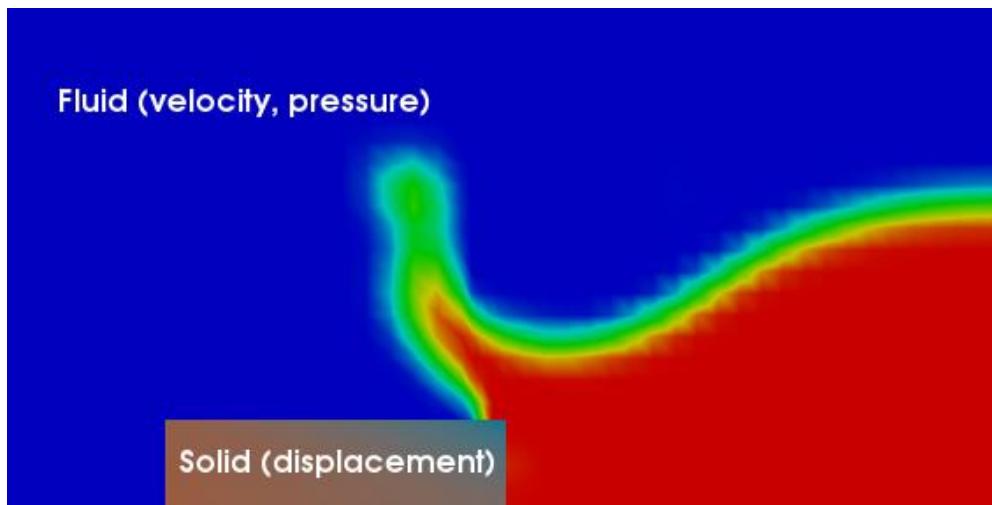


Figure 3.13: An example of a fluid-structure interaction simulation with separate grids and fields used for the solid and the fluid domains.

After `vtkCPPProcessor::RequestDataDescription()` has been called, if the method returned 1 then the adaptor needs to get the information set in the `vtkCPPipeline` objects. This information is used to determine what data to provide to the pipelines for the following `vtkCPPProcessor::CoProcess()` call. The following methods can be used to get the `vtkCPInputDataDescription` object which is used to pass the grid to the pipelines:

- `vtkCPInputDataDescription* GetInputDescription(unsigned int)`
- `vtkCPInputDataDescription* GetInputDescriptionByName(const char* name)`

For adaptors that provide a single pipeline input (i.e. `AddInput()` has only been called once), the conventional arguments for the above two methods are 0 and “input”, respectively. If multiple grid inputs are provided by the adaptor, it’s possible that not all of them are needed. To determine which ones are needed to update the required pipelines the following method can be used:

- `bool GetIfGridIsNecessary(const char* name)`

While `vtkCPDataDescription` is intended to pass the above information back and forth between the adaptor and the pipelines, for user-developed pipelines there may be more information necessary to pass back and forth. In this case, there is a user data object that can be used for this purpose. Currently we use a `vtkFieldData` object for this functionality. The reasons for this are that it is Python wrapped and it can hold a variety of data types through its intended use of aggregating classes that derive from `vtkAbstractArray`. The classes that derive from `vtkAbstractArray` are Python wrapped as well. The methods for this are:

- `void SetUserData(vtkFieldData* data)`
- `vtkFieldData* GetUserData()`

vtkCPInputDataDescription

The `vtkCPInputDataDescription` class is similar to `vtkCPDataDescription` in that it passes information between the adaptor and the pipelines. The difference though is that `vtkCPInputDataDescription` is meant to pass information about the grids and fields. As mentioned above, there should be a `vtkCPInputDataDescription` object in `vtkCPDataDescription` for each separate input VTK data object provided by the adaptor. The main methods of interest are:

- `void SetGrid (vtkDataObject *grid)` -- set the input data object representing the grids and their attributes for the pipelines.
- `void SetWholeExtent (int, int, int, int, int, int)` or `void SetWholeExtent (int[6])` -- for topologically regular grids, set the whole extent for the entire grid.

There are a variety of other methods that are intended to increase the efficiency of the adaptor. The purpose of them is to inform the adaptor code which attributes are needed for the pipelines. It is potential future work for Catalyst and so for now the above methods are the proper ones to be used for this class.

Adaptors: Putting it All Together

Here we provide details through a summary example. Note that there are more examples available at <https://github.com/acbauer/CatalystExampleCode>.

- Initialization steps
 - Create a `vtkCPPProcessor` object and call `Initialize()`
 - Create `vtkCPPipeline` objects and add them to `vtkCPPProcessor`
- Calling the co-processing routines
 - Create a `vtkCPDataDescription` object (*)
 - call `SetTimeData()`
 - For each input data object, call `AddInput()` with the key identifier string (*)
 - Optionally, call `SetForceOutput()`
 - Call `vtkCPPProcessor::requestDataDescription()` with created `vtkCPDataDescription` object
 - If `requestDataDescription()` returns 0, return control to simulation code
 - If `requestDataDescription()` returns 1:
 - For each `vtkCPInputDataDescription` create the `vtkDataObject` and attributes and add them with `vtkCPDataDescription::GetInputDataDescriptionByName(const char* name)->SetGrid()`
 - Call `vtkCPPProcessor::CoProcess()`
- Finalization steps
 - Call `vtkCPPProcessor::Finalize()` and delete the `vtkCPPProcessor` object.

Note that items with an asterisk only need to be done the first time the co-processing routines are executed as long as they remain persistent data structures. In the code below we walk the reader through a full example of a simplified adaptor.

```
// declare some static variables
vtkCPPProcessor* Processor = NULL;
vtkUnstructuredGrid* VTKGrid;

// Initialize Catalyst and pass in some file names
// for Python scripts.
void Initialize(int numScripts, char* scripts[])
{
    if(Processor == NULL)
    {
        Processor = vtkCPPProcessor::New();
        Processor->Initialize();
    }
    else
    {
        Processor->RemoveAllPipelines();
    }
    // Add in the Python script
    for(int i=1;i<numScripts;i++)
    {
        vtkCPPythonScriptPipeline* pipeline =
            vtkCPPythonScriptPipeline::New();
        pipeline->Initialize(scripts[i]);
        Processor->AddPipeline(pipeline);
        pipeline->Delete();
    }
}

// clean up at the end
void Finalize()
{
    if(Processor)
    {
        Processor->Delete();
        Processor = NULL;
    }
    if(VTKGrid)
    {
        VTKGrid->Delete();
        VTKGrid = NULL;
    }
}
```

```

}

// The simulation calls this method at the end of every time
// step. grid and attributes are the simulation data structures.
// lastTimeStep is a flag indicating whether this will be
// the last time CoProcess is called. It will force all of
// the pipelines to execute.
void CoProcess(Grid& grid, Attributes& attributes, double time,
               unsigned int timeStep, bool lastTimeStep)
{
    vtkCPDataDescription* dataDescription =
        vtkCPDataDescription::New();
    // specify the simulation time and time step for Catalyst
    dataDescription->AddInput("input");
    dataDescription->SetTimeData(time, timeStep);
    if(lastTimeStep == true)
    {
        // assume that we want to all the pipelines to execute if it
        // is the last time step.
        dataDescription->ForceOutputOn();
    }
    if(Processor->RequestDataDescription(dataDescription) != 0)
    {
        // Catalyst wants to perform co-processing. We need to build
        // the VTK grid and set the attribute information on it now.
        BuildVTKDataStructures(grid, attributes);
        // Make a map from "input" to our VTK grid so that
        // Catalyst gets the proper input data set for the pipeline.
        dataDescription->GetInputDescriptionByName("input")-
            SetGrid(VTKGrid);
        // Call Catalyst to execute the desired pipelines.
        Processor->CoProcess(dataDescription);
    }
    dataDescription->Delete();
}

```

Linking with C and Fortran Simulation Codes

Catalyst is implemented as a C++ library with the addition of Python wrapping for many methods. This makes it simple to natively link Catalyst with simulation codes developed in either C++ or Python. However, many simulation codes are written in C or Fortran and require the addition of C++ code to create VTK data objects. This is a common enough situation that we have added methods to Catalyst to simplify this. Removing name mangling of C++ functions is necessary so that they may be called by Fortran or C code. This is done by adding in ‘extern “C”’

'to the beginning of the C++ function declaration. For header files that are to be used with C and C++ code, the following can be done:

```
#ifdef __cplusplus
extern "C"
{
#endif
    void CatalystInitialize(int numScripts, char* scripts[]);
    void CatalystFinalize();
#endif
```

The __cplusplus macro is only defined for C++ compilers which then support extern “C” to remove C++ mangling of the function names without affecting the C compilers use of the header file. Another key to inter-language calls is that generally only built-in types and pointers to arrays of built-in types should be used. For Fortran, all data objects are passed as pointers. For simulation codes written in C, the proper header file to include is CAaptorAPI.h if Python isn’t used or needed. The main functions of interest defined here are:

- void coprocessorinitialize() -- initialize Catalyst.
- void coprocessorfinalize() -- finalize Catalyst.
- void requestdatadescription(int* timeStep, double* time, int* coprocessThisTimeStep) -- check the current pipelines to see if any of them need to execute for the given time and time step. The return value is in coprocessThisTimeStep and is 1 if co-processing needs to be performed and 0 otherwise.
- void coprocess() -- execute the Catalyst pipelines for the timeStep and time specified in requestdatadescription(). Note that the adaptor must update the grid and attribute information and set them in the proper vtkCPInputDataDescription object obtained through vtkCPAdaptorAPI::GetCoProcessorData() method.

If Python is used in Catalyst, then the proper header file to include in C code is CPythonAdaptorAPI.h. The two functions defined in this header file are:

- void coprocessorinitializewithpython(char* pythonFileName, int* pythonFileNameLength) -- initialize Catalyst with the ability to use Python. If pythonFileName is not null and pythonFileNameLength is greater than zero it also creates a vtkCPPythonScriptPipeline object and adds it to the vtkCPPProcessor object. Note that this method should be used instead of coprocessorinitialize().
- void coprocessoraddpythonscript(char* pythonFileName, int* pythonFileNameLength) -- creates a vtkCPPythonScriptPipeline object and adds it to group of pipelines to be executed by the vtkCPPProcessor object.

Note that these are just convenience methods and are not required to be used. A C example is included in the git examples repository (<https://github.com/acbauer/CatalystExampleCode>) which does not use these methods.

Compiling and Linking Source Code

The final step to integrating Catalyst with the simulation code is compiling all code and linking the resulting objects together. The simplest way to do this is to use CMake (www.cmake.org) as that will take care of all of the dependencies (i.e. header files as well as libraries). An example CMake file, CMakeLists.txt, is shown below.

```
cmake_minimum_required(VERSION 2.8.8)
project(CatalystCxxFullExample)

set(USE_CATALYST ON CACHE BOOL
    "Link the simulator with Catalyst")
if(USE_CATALYST)
    find_package(ParaView 3.98 REQUIRED COMPONENTS
        vtkPVPythonCatalyst)
    include("${PARAVIEW_USE_FILE}")
    set(Adapter_SRCS FEAdaptor.cxx)
    add_library(Adapter ${Adapter_SRCS})
    target_link_libraries(Adapter vtkPVPythonCatalyst)
    add_definitions("-DUSE_CATALYST")
else()
    find_package(MPI REQUIRED)
    include_directories(${MPI_CXX_INCLUDE_PATH})
endif()

add_executable(FEDriver FEDriver.cxx FEDataStructures.cxx)
if(USE_CATALYST)
    target_link_libraries(FEDriver Adapter)
else()
    target_link_libraries(FEDriver ${MPI_LIBRARIES})
endif()
```

This gives the option of building the simulation code with or without linking to Catalyst by allowing the user at configure time to enable or disable using Catalyst with the USE_CATALYST CMake option. If Catalyst is enabled then the USE_CATALYST macro is defined and can be used in the driver code to include header files and function calls to the adaptor code.

Additionally, this example CMake file adds a dependency on the Adaptor for the FEDriver simulation code example. If the simulation code doesn't require the Python interface to Catalyst, the user can avoid the Python dependency by changing the required ParaView components

from vtkPVPythonCatalyst to vtkPVCPythonCatalyst. Either of these components also brings in the rest of the Catalyst components and header files for compiling and linking.

For simulation codes that do not require CMake to build, we suggest using an example to determine the required header file locations for compiling and required libraries for linking. Due to system specific configurations, any attempt to list all the dependencies and locations here would be incomplete.

Linking with Python Simulation Codes

Python code doesn't need to compile and link with Catalyst as the needed parts of Catalyst are Python wrapped and available by importing the proper modules. The typical Catalyst modules that need to be imported are paraview, vtkPVCPythonCatalystPython, vtkPVPythonCatalystPython, paraview.simple, paraview.vtk, paraview.numpy_support and vtkParallelMPIPython. However it is necessary to set up the proper system paths so that the Catalyst modules can be properly loaded. Assuming that the build tree is available, the following system variables need to be set for a Linux machine:

- LD_LIBRARY_PATH needs to include the lib subdirectory of the build directory.
- PYTHONPATH needs to include the lib and lib/site-packages subdirectories of the build directory.

ParaView is built with NumPy (<http://www.numpy.org/>) support and optionally mpi4py (<http://mpi4py.scipy.org/>). mpi4py can be used for interprocess communication in the adaptor. NumPy has more utility in the adaptor in that if it is used to store data, a vtkdataArray object can easily be created from it. A snippet of code demonstrating this is shown below:

```
import paraview
import numpy
from paraview import numpy_support
scal = numpy.zeros(50)
vtkscal = numpy_support.numpy_to_vtk(scal)
vec = numpy.zeros(50, 3)
vtkvec = numpy_support.numpy_to_vtk(vec)
```

This creates two vtkDataArrays from NumPy arrays. The vtkscal will have 50 tuples and 1 component while the vtkvec array will have 50 tuples and 3 components.

Creating Specialized Catalyst Pipelines

If Catalyst is built without Python support, all pipelines will need to be hard-coded in C++. Even in cases when Catalyst is built with Python, simulation code developers may wish to create hard-coded C++ pipelines for their users. The main reason for this approach is to create a

simplified interface for the simulation user. The user does not have to use ParaView to create any Catalyst pipelines and may not even need to use ParaView for post-processing *in situ* extracts. In this section, we go through three different ways to do this. The first is directly creating VTK pipelines. The second is creating VTK pipelines through ParaView's C++ server-manager interface. The third is creating VTK pipelines through ParaView wrapped Python scripts. In the table below we list the main advantages and disadvantages of each.

Pipeline	Advantages	Disadvantages
VTK C++	Good documentation, not dependent on Python, many examples	Complex to create output images in parallel, changes require recompilation
ParaView C++	Automatically sets up compositing and render passes easily, not dependent on Python	Sparse documentation, few examples, changes require recompilation
ParaView Python	Can be modified without requiring recompilation, can use existing scripts created in GUI and/or using ParaView's trace functionality	Requires linking with Python

We recommend reviewing the *Avoiding Data Explosion* subsection of the Catalyst for Users section before creating specialized pipelines. The reason for this is that while many filters are very memory efficient, others can dramatically increase the amount of needed. This is a major factor to consider when running on memory limited HPC machines where no virtual memory is available.

VTK C++ Pipeline

Creating a custom VTK C++ pipeline is fairly straightforward for those that are familiar with VTK. This is done in a class that derives from `vtkCPPipeline`. The two methods that need to be implemented are `RequestDataDescription()` and `CoProcess()`. Optionally, `Finalize()` can be implemented if there are operations that the class needs to do before being deleted.

`RequestDataDescription()` will contain code to determine if the VTK C++ pipeline needs to be executed and return 1 if it does and 0 otherwise. It should also check that the proper information is set (e.g. output file name information) for the pipeline to output the desired data. `CoProcess()` is the method in which the actual VTK C++ pipeline is executed. In the example in the git repository we create the pipeline every time it is needed but that is not necessary. Note that pipelines are not limited to using only filters specified in the VTK code base. They can also use filters specified in the ParaView code base as well. For example, we recommend using ParaView's `vtkCompleteArrays` filter prior to using any of the parallel XML writers available in VTK. The reason for this is that the parallel XML writers can give bad output if process 0 has no points or cells due to not having the needed attribute information to include in the meta-file of the format.

It is beyond the scope of this *Catalyst Users Guide* to give a complete description of all of the filters in VTK and ParaView. In addition to the *VTK User's Guide*, wiki and doxygen documentation web pages, we also recommend looking at the examples at <http://vtk.org/Wiki/VTK/Examples/Cxx> for help in creating VTK pipelines. A fully functioning example with a hard-coded VTK C++ pipeline is available from the Catalyst examples git repository.

ParaView C++ Pipeline

As background, ParaView's server-manager is the code that controls the flow of information and maintains state in ParaView's client-server architecture. It is used by the client to set up the pipeline on the server, to set the parameters for the filters and execute the pipeline, among other duties. Besides these duties, it will automatically do things like add in the `vtkCompleteArrays` filter prior to any parallel writers that are added in the pipeline. The reason for this is mentioned in the previous section. Additionally, it properly sets up the parallel image compositing that can be difficult in pure VTK code.

Similar to creating a VTK C++ pipeline, we won't go into the full details of creating a ParaView server-manager pipeline due to the extent of the information. Most classes that will be used derive from `vtkSMProxy`, `vtkSMProperty` or `vtkSMDomain`. `vtkSMProxy` is used for creating VTK objects such as filters and maintaining references and state of the VTK objects. `vtkSMProperty` is used for calling methods on the VTK objects with given passed in parameters (e.g. setting the file name of a writer or setting the isosurface values of the contour filter). `vtkSMDomain` represents the possible values properties can have (e.g. the radius of a sphere must be positive). The XML files under the `ParaViewCore/ServerManager/SMAplication/Resources` subdirectory of the ParaView source directory lists all of the proxy information. The key XML files are:

- `filters.xml` -- contains the descriptions of all of the filters that may be available in Catalyst.
- `sources.xml` -- contains pipeline sources such as spheres, planes, etc. They may be useful for setting inputs such as seed points for streamlines.
- `writers.xml` -- contains descriptions of the writers that may be available in Catalyst.
- `utilities.xml` -- contains utility proxies such as functions and point locators that may be needed by certain filters.
- `rendering.xml` -- contains proxies for setting rendering options such as cameras, mappers, textures, etc.
- `views_and_representations.xml` -- contains proxies for setting view information such as 3d render views, charts, etc. and representations such as surface, wireframe, etc.

Note that due to configuration of Catalyst, some proxies listed in the XML files may not be available. An example of a ParaView server-manager created Catalyst pipeline is included in the git examples repository.

Custom Python Script Pipeline

For creating custom Catalyst Python pipelines, the simplest way is to start with something that is fairly similar already. The easiest way to do that is to create a similar pipeline in ParaView and export it using the co-processing script generator plugin (discussed in Section 4). For the most part, these generated Python scripts are very readable, especially if no screenshots are being output. Other useful ParaView tools for creating and/or modifying Catalyst Python scripts include:

- the ParaView GUI's Python interpreter (available by going to Tools->Python Shell in the main menu) supports tab completion to help see available methods for each object.
- using ParaView's trace functionality that can record the Python commands that mimic a user's interaction with the GUI. This is available by using Tools->Start Trace and Tools->Stop Trace to start and stop the trace, respectively.

Additionally, most of the ParaView wrapped objects have at least a minimal built-in documentation. There is also sphinx generated documentation available at http://www.paraview.org/ParaView3/Doc/Nightly/html_pages/index.html. We assume that through the tools above users will be able to create the proper objects and set the proper parameters for them. The other information that is useful for creating custom Catalyst scripts is being able to query for information about the output from filters. This includes information like bounds of the output data set, the ranges of attributes, etc. These are the typical pieces of information that will be used to add logic into a custom Python pipeline. For example, when creating iso-surfaces through the contour filter it is necessary to know the range of the data array that is to be isosurfaced with respect to. From a ParaView Python wrapped filter proxy, the following methods are the most useful for querying filter output:

- `UpdatePipeline()` -- executes the pipeline such that the filter output is current. This should be called before any information is requested from the following methods.
- `GetDataInformation()` -- get information about the filter's output data object. Members of interest are:
 - `GetDataSetNameAsString()` -- return the VTK class name of the data set (e.g. `vtkPolyData`)
 - `DataInformation` -- the Python wrapped `vtkPVDataInformation` object. Methods of interest include:
 - `GetBounds()` -- return the geometric bounds of the data object. Values are in a list with an ordering of {minimum X, maximum X, minimum Y, maximum Y, minimum Z, maximum Z}
 - `GetNumberOfPoints()` -- return the number of points in the data object
 - `GetNumberOfCells()` -- return the number of cells in the data object
- `GetPointData()/GetCellData()` -- an object that provides information about the point data or cell data arrays, respectively. The main members of interest for this are:
 - `GetNumberOfArrays()` -- give the number of point data or cell data arrays available

- `GetArray()` -- return an array information object. The single argument to this method can either be an integer index or a string name. The main members of this are:
 - `Name` -- the name of the array
 - `GetRange()` -- the range of the values

An example of how of querying a filter's output is given below:

```
from paraview.simple import *
s = Sphere()
e = Elevation()
e.UpdatePipeline() # updates e so that output is generated
di = e.GetDataInformation()
# get the bounds of the output of e
bounds = di.DataInformation.GetBounds()
pd = e.GetPointData()
# get the range of the Elevation point data array
datarange = pd.GetArray("Elevation").GetRange()
```

Note that these scripts can be added to `vtkCPPProcessor` as long as they implement the `RequestDataDescription()` and `DoCoProcessing()` methods.

Section 4: Building Catalyst

This section is targeted towards those users or developers responsible for building ParaView Catalyst. As far as installation is concerned, Catalyst is a subset of the ParaView code base. Thus, all of the functionality available in Catalyst is also available in ParaView. The difference is that ParaView will by default have many more dependencies and thus will have a larger executable size. Catalyst is the flexible and specialized configuration of ParaView that is used to reduce the executable size by reducing dependencies. For example, if no output images are to be produced from a Catalyst-instrumented simulation run then all of the ParaView and VTK code related to rendering and the OpenGL libraries need not be linked in. This can result in significant memory savings, especially when considering the number of processes utilized when running a simulation at scale. In one simple example, the executable size was reduced from 75 MB when linking with ParaView to less than 20 MB when linking with Catalyst.

The main steps for configuring Catalyst are:

- 1) Setting up an “edition”
- 2) Extract the desired code from ParaView source tree into a separate Catalyst source tree
- 3) Build Catalyst

Most of the work is in the first step which is described below. A Catalyst edition is a customization of ParaView to support a desired subset of functionality from ParaView and VTK. There can be many editions of Catalyst and these editions can be combined to create several

customized Catalyst builds. Assuming that the desired editions have already been created, the second step is automated and is done by invoking the following command from the <ParaView_source_dir>/Catalyst directory:

```
python catalyze.py -i <edition_dir> -o <Catalyst_source_dir>
```

Note that more editions can be added with the `-i <edition_dir>` and that these are processed in the order they are given, first to last. For the minimal base edition included with ParaView, this would be `-i Editions/Base`. The generated Catalyst source tree will be put in `<Catalyst_source_dir>`. For configuring Catalyst from the desired build directory, do the following:

```
<Catalyst_source_dir>/cmake.sh <Catalyst_source_dir>
```

The next step is to build Catalyst (e.g. using make on Linux systems).

Creating a Catalyst Edition

The main operations for creating an edition of Catalyst are:

- 1) Set CMake build parameters (e.g. static or shared library build).
- 2) Specify files from the ParaView source tree to be copied into the created Catalyst source tree.
- 3) Specify files from the edition to be copied into the Catalyst source tree.

The information describing which files are in the generated Catalyst source tree is all stored in a JSON file called `manifest.json` in the main directory of the edition. The user processes this information with a Python script called `catalyze.py` that is located in the <ParaView_source_dir>/Catalyst directory.

Setting CMake Build Parameters

By default, Catalyst will be built with the default ParaView build parameters (e.g. build with shared libraries) unless one of the Catalyst editions changes that in its `manifest.json` file. An example of this is shown below:

```
"cmake": {  
    "cache": [  
        {  
            "name": "BUILD_SHARED_LIBS",  
            "type": "BOOL",  
            "value": "OFF"  
        }  
    ]  
}
```

Here, ParaView's CMake option of building shared libraries will be set to OFF. It should be noted that users can still change the build configuration from these settings but it should be done after Catalyst is configured with the cmake.sh script.

Copying Files from the ParaView Source Tree into the Created Catalyst Source Tree

By default, very little source code from the ParaView source tree will be copied to the generated Catalyst source tree. Each edition will likely want to add in several source code files to the Catalyst source tree. Most of these files will be filters but there may also be several helper classes that are needed to be copied over as well. In the following JSON snippet we demonstrate how to copy the vtkPVArrayCalculator class into the generated Catalyst source tree.

```
"modules": [
    {
        "name": "vtkPVVTKExtensionsDefault",
        "path": "ParaViewCore/VTKExtensions/Default",
        "include": [
            {
                "path": "vtkPVArrayCalculator.cxx"
            },
            {
                "path": "vtkPVArrayCalculator.h"
            }
        ],
        "cswrap": true
    }
]
```

A description of the pertinent information follows:

- "name": "vtkPVVTKExtensionsDefault" – the name of the VTK or ParaView module. In this case it is vtkPVVTKExtensionsDefault. The name of the module can be found in the modules.cmake file in the corresponding directory. It is the first argument to the vtk_module() function.
- "path": "ParaViewCore/VTKExtensions/Default" – the subdirectory location of the module relative to the main ParaView source tree directory (e.g. <ParaView_source_dir>/ParaViewCore/VTKExtensions/Default in this case)
- "path": "vtkPVArrayCalculator.cxx" – the name of the file to copy from the ParaView source tree to the generated Catalyst source tree.
- "cswrap": true – if the source code needs to be client-server wrapped such that it is available through ParaView's server-manager. For filters that are used through

ParaView's Python interface or through a server-manager hard-coded C++ pipeline this should be tree. For helper classes this should be false.

The difficult part here is determining which files need to be included in Catalyst. In the example above, the actual name of the ParaView proxy for the vtkPVArrayCalculator is Calculator. Thus, to construct a ParaView client proxy for vtkPVArrayCalculator on the server, the user would need to call Calculator() in the Python script. The best way to determine this connection between the name of the ParaView proxy and the actual source code is in the XML files in the ParaViewCore/ServerManager/SMAplication/Resources. In this case the proxy definition is in the filters.xml file. The proxy label XML element will be converted into the Python constructor for the proxy and the class name is stored in the proxy class XML element. The conversion of the proxy label is done by removing spaces in the XML attribute. This is sufficient for many situations but for some cases there will be additional classes needed to be included in order to properly compile Catalyst. This can occur when the included source code derives from a class not already included in Catalyst or uses helper classes not already included in Catalyst. For the vtkPVArrayCalculator class we will also need to include the vtkArrayCalculator class that it derives from.

Copying files from the edition into the Catalyst source tree

Some of the files that need to be in the generated Catalyst source tree cannot be directly copied over from the ParaView source tree. For example, CMakeLists.txt files need to be modified in the Catalyst source tree when multiple editions need to be added into a specialized CMakeLists.txt file in the same directory. This is done with the "replace" keyword. An example of this is shown below for the vtkFiltersCore module. Here, the vtkArrayCalculator source code is added to the Catalyst source tree and so the CMakeLists.txt file in that directory needs to be modified in order to include that class to be added to the build.

```
"modules": [
  {
    "name": "vtkFiltersCore",
    "path": "VTK/Filters/Core",
    "include": [
      {
        "path": "vtkArrayCalculator.cxx"
      },
      {
        "path": "vtkArrayCalculator.h"
      }
    ],
    "replace": [
      {
        "path": "VTK/Filters/Core/CMakeLists.txt"
      }
    ],
  }
],
```

```

    "cswrap":true
}
]

```

In this case, the CMakeLists.txt file that needs to be copied to the Catalyst source tree exists in the <edition_dir>/VTK/Filters/Core directory, where edition_dir is the location of this custom edition of Catalyst. Since the Base edition already includes some files from this directory, we want to make sure that the CMakeLists.txt file from this edition also includes those from the Base edition. This CMakeLists.txt file is shown below:

```

set(Module_SRCS
vtkArrayCalculator.cxx
vtkCellDataToPointData.cxx
vtkContourFilter.cxx
vtkContourGrid.cxx
vtkContourHelper.cxx
vtkCutter.cxx
vtkExecutionTimer.cxx
vtkFeatureEdges.cxx
vtkGridSynchronizedTemplates3D.cxx
vtkMarchingCubes.cxx
vtkMarchingSquares.cxx
vtkPointDataToCellData.cxx
vtkPolyDataNormals.cxx
vtkProbeFilter.cxx
vtkQuadricClustering.cxx
vtkRectilinearSynchronizedTemplates.cxx
vtkSynchronizedTemplates2D.cxx
vtkSynchronizedTemplates3D.cxx
vtkSynchronizedTemplatesCutter3D.cxx
vtkThreshold.cxx
vtkAppendCompositeDataLeaves.cxx
vtkAppendFilter.cxx
vtkAppendPolyData.cxx
vtkImageAppend.cxx
)

set_source_files_properties(
    vtkContourHelper
    WRAP_EXCLUDE
)

vtk_module_library(vtkFiltersCore ${Module_SRCS})

```

Note that this CMakeLists.txt file does two things. Firstly it specifies which files to be compiled in the source directory. Next, it specifies properties of the source files. In the above example, vtkContourHelper is given a property specifying that it should not be wrapped. Another property which is commonly set indicates that a class is an abstract class (i.e. it has pure virtual functions). An example of how to do this is shown below.

```
set_source_files_properties(
    vtkXMLPStructuredDataWriter
    vtkXMLStructuredDataWriter
    ABSTRACT
)
```

References

- "Data Co-Processing for Extreme Scale Analysis Level II ASC Milestone". David Rogers, Kenneth Moreland, Ron Oldfield, and Nathan Fabian. Tech Report SAND 2013-1122, Sandia National Laboratories, March 2013.
- "The ParaView Coprocessing Library: A Scalable, General Purpose *In Situ* Visualization Library." Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C. Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth E. Jansen. In *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, October 2011, pp. 89–96. DOI 10.1109/LDAV.2011.6092322.
- "The Visualization Toolkit: An Object Oriented Approach to 3D Graphics". Will Schroeder, Ken Martin, and Bill Lorensen. Kitware Inc., fourth edition, 2004. ISBN 1-930934-19-X.
- "The ParaView Guide: A Parallel Visualization Application". Utkarsh Ayachit et al. Kitware Inc., 4th edition, 2012. ISBN 978-1-930934-24-5.

Examples

There are a wide variety of VTK examples at <http://www.vtk.org/Wiki/VTK/Examples>. This site includes both C++ and Python examples but is targeted for general VTK development. Examples specific to Catalyst can be found at <https://github.com/acbauer/CatalystExampleCode>.

Appendix

vtkWeakPointer, vtkSmartPointer and vtkNew

To simplify reference counting, vtkWeakPointer, vtkSmartPointer and vtkNew can be used. vtkWeakPointer stores a pointer to an object but doesn't change the reference count. When the object gets deleted vtkWeakPointer will get initialized to NULL avoiding any dangling

references. The latter two classes keep track of other vtkObjects by managing the object's reference count. When these objects are created, they increment the reference count of the object they are referring to and when they go out of scope, they decrement the reference count of the object they are referring to. The following example demonstrates this.

```
{  
vtkNew<vtkDoubleArray> a; // a's ref count = 1  
a->SetName("an array");  
vtkSmartPointer<vtkPointData> pd =  
    vtkSmartPointer<vtkPointData>::New(); // pd's ref count = 1  
pd->AddArray(a.GetPointer()); // a's ref count = 2  
vtkSmartPointer<vtkDoubleArray> a2 =  
    vtkSmartPointer<vtkDoubleArray>::New(); // a2's ref count = 1  
pd->AddArray(a2); // a2's ref count = 2  
vtkWeakPointer<vtkPointData> pd2;  
pd2 = pd; // pd's ref count = 1  
vtkPointData* pd3 = vtkPointData::New();  
pd2 = pd3;  
pd3->Delete(); // pd3 is deleted  
pd2->GetClassName(); // bug!  
} // don't need to call Delete on any object
```

Note that when passing a pointer returned from vtkNew as a parameter to a method that the GetPointer() method must be used. Other than this caveat, vtkSmartPointer and vtkNew objects can be treated as pointers.