Upgrading to and Understanding the New VTK Pipeline

This document will first introduce you to the classes and concepts used in the new VTK pipeline and then provide some instructions and tips for converting your existing classes to work within the new pipeline. The new pipeline was designed to reduce complexity while at the same time provide more flexibility. In the old pipeline the pipeline functionality and mechanics were contained in the data object and filters. In the new pipeline this functionality is contained in a new class (and its subclasses) called vtkExecutive. There are four key classes that make up the new pipeline. They are:

> **vtkInformation:** to provide the flexibility to grow most of the methods and meta information storage makes use of a new class called vtkInformation. vtkInformation is a map based data structure that supports heterogeneous key-value operations with compile time type checking.
>
> **vtkDataObject:** in the past this class both stored data and handled some of the pipeline logic. In the new pipeline this class is only supposed to store data. In practice there are some pipeline methods in the class for backwards compatibility so that all of VTK doesn't break, but the goal is that vtkDataObject should only be about storing data. vtkDataObject has an instance of vtkInformation that can be used to store key-value pairs in. For example the current extent of the data object is store in there but the whole extent is not, because that is a pipeline attribute containing information about a specific pipeline topology .
>
> **vtkAlgorithm:** an algorithm is the new superclass for all filters/sources/sinks in VTK. It is basically the replacement for vtkSource. Like vtkDataObject, vtkAlgorithm should know nothing about the pipeline and should only be an algorithm/function/filter. Call it with the correct arguments and it will produce results. It also has a vtkInformation instance that describes the properties of the algorithm and it has information objects that describe its input and output ports. The main method of an algorithm is ProcessRequest.
>
> **vtkExecutive:** contains the logic of how to connect and execute a pipeline. This class has a direct subclass called vtkDistributedExecutive which is the superclass of all executives that are distributed as opposed to centralized (e.g. each filter/algorithm has its own executive that communicates with other executives). vtkDistributedExecutive has a subclass called vtkDemandDrivenPipeline which in turn has a subclass called vtkStreamingDemandDrivenPipeline. vtkStreamingDemandDrivenPipeline provides most of the functionality that was found in the old VTK pipeline and is the default executive for all algorithms if you do not specify one.

Let us first look at the vtkAlgorithm class. It may seem odd that a class with no notion of a pipeline has one key method called ProcessRequest. At its simplest, an algorithm has a basic function to take input data and produce output data. This is a down-stream request (specifically REQUEST_DATA) that all algorithms should implement. But algorithms can do more than just produce data; they also have characteristics or metadata that they can provide. For example, an algorithm can provide information about what type of output it will produce when you execute it. An imaging algorithm might only be capable of producing *double* results. The algorithm can specify this by responding to another down-stream request called REQUEST_INFORMATION. Consider the following code fragment:

```
int vtkMyAlgorithm::ProcessRequest(
  vtkInformation *request,
  vtkInformationVector *inputVector,
  vtkInformationVector *outputVector)
{
  // generate the data
  if(request->Has(vtkDemandDrivenPipeline::REQUEST_INFORMATION()))
    {
    // specify that the output (only one for this filter) will be
double
    vtkInformation*        outInfo        =        outputVector-
>GetInformationObject(0);
    outInfo->Set(vtkDataObject::SCALAR_TYPE(),VTK_DOUBLE);
    return 1;
    }
  return  this->Superclass::ProcessRequest(request,  inputVector,
outputVector);
}
```

This method takes three information objects as input. The first is the request which specifies what you are asking the algorithm to do. Typically this is just one key such as REQUEST_INFORMATION. The next two arguments are information vectors one for the inputs to this algorithm and one for the outputs of this algorithm. In the above example no input information was used. The output information vector was used to get the information object associated with the first output of this algorithm. Into that information was placed a key-value pair specifying that it would produce results of type *double*. Any requests that the algorithm doesn't handle should be forwarded to the superclass.

The pipeline topology in the new pipeline is a little different from the old one. In the new pipeline you connect the output port of one algorithm to the input port of another algorithm. For example,

```
alg1->SetInputConnection( inPort, alg2->GetOutputPort( outPort ) )
```

Any input port in the new pipeline can be specified as repeatable and or optional. Repeatable means that more than one connection can be made to this input port (such as for append filters). Optional means that the input is not required for execution. Of course the old style of SetInput, GetOutput connections will work with existing algorithms as well. So in the new pipeline outputs are referred to by port number while inputs are referred to by both their port number and connection number (because a single input port can have more than one connection)

Converting an Existing Filter to the New Pipeline

The new pipeline implementation does include a backwards compatibility layer for old filters. Specifically vtkProcessObject, vtkSource, and their related subclasses are still present and working. Most filters should work with the new pipeline without any changes. The most common problems with the backwards compatibility layer involve filters that manipulate the pipeline. If your filter overrides UpdateData or UpdateInformation you will probably have to make some changes. If your filter uses an internal pipeline then you might need to make some changes, otherwise you should be OK.

Now ideally you would convert your filter to the new pipeline. There is a script that you can run that will help you to convert your filter. The script doesn't do everything but it will help get you going in the right direction. You can run the script on an existing class as follows:

```
cd VTK/MYClasses
cmake            -D            CLASS=vtkMyClass             -P
../Utilities/Upgrading/NewPipeConvert.cmake
```

One of the effects this script might have is to change the superclass of your class. There are some convenience superclasses to make writing algorithms a little easier. In the old pipeline there were classes such as vtkImageToImageFilter. Some of the classes designed for the new pipeline include:

vtkPolyDataAlgorithm: for algorithms that produce vtkPolyData
vtkImageAlgorithm: for algorithms that take and or produce vtkImageData
vtkThreadedImageAlgorithm: a subclass of vtkImageAlgorithm that implements multithreading

These classes have some defaults that can be easily changed in your subclass. The first default is that the subclass will take on input and produce one output. This is typically specified in the constructor using SetNumberOfInputPorts(1) and SetNumberOfOutputPorts(1). If your subclass doesn't take an input then in its constructor just call SetNumberOfOutputPorts(0). Another assumption that is made is that all the input port and output ports take vtkPolyData (for vtkPolyDataAlgorithm, vtkImageData for vtkImageAlgorithm etc). Again your subclass can override this by providing its own implementation of FillInputPortInformation or FillOutputPortInformation.

These superclasses typically provide an implementation of ProcessRequest that handles REQUEST_INFORMATION and REQUEST_DATA request by invoking virtual functions called RequestData and ExecuteInformation. They also typically provide default implementations of RequestData that call the older style ExecuteData functions to make converting your old filters easier.