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. You should be familiar with the basic design of the old pipeline before delving into this. 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: provides the flexibility to grow. Most of the methods and meta information storage make use of this class. vtkInformation is a map-based data structure that supports heterogeneous key-value operations with compile time type checking. There is also a vtkInformationVector class for storing vectors of information objects. When passing information up or down the pipeline (or from the executive to the algorithm) this is the class to use.

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 stored 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 port characteristics. The main method of an algorithm is ProcessRequest.

vtkExecutive: contains the logic of how to connect and execute a pipeline. This class is the superclass of all executives. Executives are distributed (as opposed to centralized) and each filter/algorithm has its own executive that communicates with other executives. vtkExecutive 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 )
)
```

Using this terminology a port is like a pin on an integrated circuit. An algorithm has some input ports and some output ports. A connection is a "connection" between two ports. So to connect two algorithms you make a connection between the output port of one

algorithm and the input port of another algorithm. 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. While ports can be repeatable there is still a need for multiple ports. Your algorithm should have multiple ports when the concept, data type, or semantics of a port are different. So AppendFilter only needs one repeatable input port because it treats all of its inputs the same. Glyph in contrast should have two ports, one for the input points and one for the glyph model because these are two distinct concepts. 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)

Typical Pipeline Execution

Let us take a quick look at the typical execution of a pipeline in VTK. First the use instantiates an Algorithm such as vtkImageGradient, when the algorithm is instantiate it will automatically create a default executive and connect the algorithm and executive together. The algorithm will also call FillInputPortInformation and FillOutputPortInformation on each input and output port respectively. These two methods should setup the static characteristics of the input and output ports such as data type requirements and whether the port is optional or repeatable. For example, by default all subclasses of vtkImageAlgorithm are assumed to take an vtkImageData as input:

```
int vtkImageAlgorithm::FillInputPortInformation(int vtkNotUsed(port), vtkInformation*
info)
{
 info->Set(vtkAlgorithm::INPUT_REQUIRED_DATA_TYPE(), "vtkImageData");
 return 1;
}
```

This can be overridden by subclasses as required. Once the algorithm and executive are instantiated they will be connected to other algorithm executive pairs. If the new SetInputConnection signature is used (and it should be used) then this just stores the connectivity information. The old VTK pipeline used the data objects to store connectivity and thus required that the data objects be instantiated prior to calling GetOutput. Once the entire pipeline is instantiated and connected it will be executed (typically as the result of a Render() call) Typically the first request an algorithm will see is upon execution is REQUEST_DATA_OBJECT (see vtkExecutive, vtkDataObject and their subclasses for all the possible keys and requests.) This request asks the algorithm to create an instance of vtkDataObject (or appropriate subclass) for all of its output ports. If you handle this request you can set the output ports output data to be whatever type you want (see vtkDataSetAlgorithm for an example), if the algorithm does not handle the request then by default the executive will look at the output port information to see what type the output port has been set to (from FillOutputPortInformation) If this is a concrete

type then it will instantiate an instance of that type and use it. If it isn't concrete then (I can't remember if it just fails with an error or has another fallback position)

Request Information

This point the pipeline is instantiated, connected, and the data objects have been instantiated to store the data. The next request an algorithm will typically see is REQUEST_INFORMATION. This request asks the Algorithm to provide as much information as it can about what the output data will look like once the algorithm has generated it. Typically an algorithm will look at the information provided about its inputs and try to specify what it can about its outputs. In many image processing filters quite a bit can be specified such as the WHOLE_EXTENT, SCALAR_TYPE, SCALAR_NUMBER_OF_COMPONENTS, ORIGIN, SPACING, etc. The rule here is to provide or compute as much information as you can without actually executing (or reading in the entire data file) and without taking up significant CPU time. For example an image reader should read the header information from the file to get what information it can out of it, but it should not read in the entire image so that it can compute the scalar range of the data. When providing information about an output, an algorithm is not limited to the current information keys (such as WHOLE_EXTENT) that are provided by VTK. Part of the new pipeline design is that it can be easily extended. You can define your own keys and then in the REQUEST_INFORMATION request you can add those keys and their values to the output information objects. You can also specify that you want those keys to be passed down (or up) the pipeline by adding them to the KEYS_TO_COPY. That way you could have a specialized reader that populates the information with special keys and then have a writer or mapper downstream that uses those keys. By default executives will first copy the input's information to the output's information. You only need to handle the cases where it is different.

Request Update Extent

The next request is typically REQUEST_UPDATE_EXTENT. To fulfill this request an algorithm should take the update extents in its output's information and then set the correct update extents in its input's information. As with REQUEST_INFROMATION there is a default behavior by the executive and you only need to handle cases where it changes. (see vtkImageGradient for an example)

Request Data

Finally REQUEST_DATA will be called and the algorithm should fill in the output ports data objects.

Converting an Existing Filter to the New Pipeline

The best approach at this point is to find a VTK filter similar to your filter and then copy that. An alternate approach is to follow the instructions below. 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 SetNumberOfInputPorts(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 RequestInformation. They also typically provide default implementations of RequestData that call the older style ExecuteData functions to make converting your old filters easier.