Summary

One of the main goals of the pipeline re-architecture between VTK 4 and 5 was to 1) move the execution logic from data and process (algorithm) objects to a new set of classes called executives. This was to enable easy modification and extension of the pipeline (execution) code as well as to 2) separate data and execution "models" in order to simplify the code for both. The changes between VTK 4 and 5 achieved the first goal fairly well but did not tackle the second goal in order to preserve backwards compatibility with VTK 4. The work described herein has two major goals:

1. To remove the VTK 4 backwards compatibility layer introduced in VTK 5 in order to simplify the toolkit.
2. To carry the work started in VTK 5 to its logical conclusion by completely separating data and executions models

This work comes at a cost: many of the changes described here are not backwards compatible with VTK 4 and some of them are not backwards compatible with VTK 5. In this document, we summarize the changes introduced as part of this work as well as provide guidance in migrating existing code forward.

Overview of Changes

The changes introduced as part of this work can be classified under three major categories:

1. Removal of VTK 4 backwards compatibility superclasses: This includes vtkProcessObject, vtkSource and all of their subclasses. These classes were changed in VTK 5 in order to provide backwards compatibility. In VTK 6, all pipeline modules should subclass from vtkAlgorithm or one of its subclasses.
2. Removal of the pipeline meta-data API from vtkDataObject: As of VTK 5, the main containers of all pipeline meta-data and heavy-data are vtkInformation objects that represent input and output information for algorithms. Data objects should no longer be used to store meta-data (e.g. Whole Extent, Update Extent etc.). The vtkDataObject API for pipeline meta-data was kept for backwards compatibility to VTK 4 and was removed as part of this work.
3. Removal of data objects' dependency on the pipeline code (algorithms and executives): The aims of this change are to simplify data object classes and to allow for the separation of data and execution model libraries as part of the ongoing modularization effort. This change causes a few incompatibilities explained in detail below.

Changes in Detail

Removal of VTK 4 Backwards Compatibility Superclasses

The changes to the pipeline and pipeline modules introduced in VTK are explained in detail in a PDF document in VTK's source code distribution: VTK/Utilities/Upgrading/TheNewVTKPipeline.pdf. If you are not familiar with these changes and find that your code stopped compiling due to changes described in this section, we recommend that you review that document first. It described in more detail the changes to pipeline modules and how to migrate to VTK 5.

In order to the transition to the VTK 5 pipeline implementation as smoothly as possible, VTK 5 introduced a set of backwards compatibility classes. These included modified versions of previous algorithm superclasses including vtkProcessObject, vtkSource, vtkXXXSource (where XXX represents various data types) and vtkXXXToYYYFilter (where XXX and YYY are various data types). VTK 6 removes all of these classes and requires that all pipeline modules subclass from vtkAlgorithm or one of its subclasses and use the RequestInformation / RequestUpdateExtent / RequestData API rather than ExecutionInformation / PropagateUpdateExtent / Execute API. Below is a full list of backwards compatibility classes that were removed in VTK 6. If your code fails to compile because it refers to any of these classes, it is time to migrate to the VTK 5 API.

Filtering/vtkDataObjectSource
Filtering/vtkDataSetSource
Filtering/vtkDataSetToDataSetFilter
Filtering/vtkDataSetToImageFilter
Filtering/vtkDataSetToPolyDataFilter
Filtering/vtkDataSetToStructuredGridFilter
Filtering/vtkDataSetToStructuredPointsFilter
Filtering/vtkDataSetToUnstructuredGridFilter
Filtering/vtkImageInPlaceFilter
Filtering/vtkImageMultipleInputFilter
Filtering/vtkImageMultipleInputOutputFilter
Filtering/vtkImageSource
Filtering/vtkImageToImageFilter
Filtering/vtkImageTwoInputFilter
Filtering/vtkPointSetSource
Filtering/vtkPointSetToPointSetFilter
Filtering/vtkPolyDataSource
Filtering/vtkPolyDataToPolyDataFilter
Filtering/vtkProcessObject
Filtering/vtkRectilinearGridSource
Filtering/vtkRectilinearGridToPolyDataFilter
Filtering/vtkSource
Filtering/vtkStructuredGridSource

Filtering/vtkStructuredGridToPolyDataFilter
Filtering/vtkStructuredGridToStructuredGridFilter
Filtering/vtkStructuredPointsSource
Filtering/vtkStructuredPointsToPolyDataFilter
Filtering/vtkStructuredPointsToStructuredPointsFilter
Filtering/vtkStructuredPointsToUnstructuredGridFilter
Filtering/vtkUnstructuredGridSource
Filtering/vtkUnstructuredGridToPolyDataFilter
Filtering/vtkUnstructuredGridToUnstructuredGridFilter
FilteringvtkStructuredPointsToUnstructuredGridFilter
FilteringvtkUnstructuredGridToUnstructuredGridFilter
Imaging/vtkImageSpatialFilter

Removal of All Pipeline Meta-Data API from vtkDataObject

Examples of pipeline meta-data include WholeExtent, UpdateExtent, MaximumNumberOfPieces, UpdateNumberOfPieces, UpdatePiece, UpdateGhostLevel, ScalarType and NumberOfScalarComponents. In VTK 5, this meta-data is represented by vtkInformation keys such as vtkStreamingDemandDriven::WHOLE_EXTENT() and vtkStreamingDemandDrivenPipeline::UPDATE_EXTENT() and flow up and down the pipeline through input and output information objects passed to ProcessRequest (hence to RequestInformation, RequestUpdateExtent and RequestData). In VTK 4, this was achieved using vtkDataObjects and the vtkDataObject API (methods such as Set/GetWholeExtent, Set/GetUpdateExtent etc.). VTK 5 maintained this API as well as the associated vtkDataObject data members. The Executives had the responsibility of synching vtkDataObject data members with data in input and output information objects. The code to achieve this was hard to maintain and fragile. Furthermore, developers were often confused by the API to obtain pipeline specific meta-data (such as WholeExtent – the extent of the entire image available from a source or a reader) and data specific meta-data (such as Extent – the extent of the image data currently in memory). For example, we encountered the following in several places in VTK code as well as in other applications that used VTK:

```
vtkImageData* image = vtkImageData::New();
int extent[6] = {0, 10, 0, 10, 0, 10};
image->SetExtent(extent);
image->SetUpdateExtent(extent);
image->SetWholeExtent(extent);
image->AllocateScalars();
```

The correct implementation (which only uses SetExtent) was not clear to many developers and they chose the “safe” route of setting everything related to Extent to the same value – which is not really safe since it is a bug.

VTK 6 removes all pipeline meta-data API from data object and requires the use of the appropriate information keys to create, modify and read pipeline meta-data. Refer to the Appendix for a full list of methods removed from vtkDataObject.

Removal of Data Objects' Dependency on the Pipeline

The final set of changes introduced as part of this work remove the dependency of vtkDataObject and its subclasses on the pipeline objects (executives and algorithms) completely decoupling the VTK "data model" from the VTK "execution model". This has two major advantages:

- Support for modularization: With these changes, it will be possible to create a small, self-contained library that only includes support for core classes and data model classes. This will enable developers to leverage VTK's data model in their applications without significant code bloat.
- Simplification of data object and pipeline execution APIs: These changes eliminate a significant amount of API duplication between algorithms, executives and executives simplifying the VTK API. For example, to update (i.e. to force the execution of an) algorithm, the developer now has access to one methods: vtkAlgorithm:Update() rather than several distributed among data objects and algorithms.

Some of the changes introduced to remove this dependency are transparent to all but advanced users/developers of VTK. Others impact many developers that use VTK. The first of these is that all pipeline execution API was removed from vtkDataObject. Therefore, code similar to the following will not compile:

```
vtkDataObject* dobj = someAlgorithm->GetOutput();
dobj->Update();
```

This needs to be replaced with the following

```
someAlgorithm->Update();
```

The second backwards-incompatible change is probably the one that will impact the most code. In VTK 4, pipeline objects were connected as follows:

```
someFilter->SetInput(someReader->GetOutput());
```

In VTK 5, this was changed to:

```
someFilter->SetInputConnection(someReader->GetOutputPort());
```

However, the SetInput() and related methods were preserved for backwards compatibility. This method and all related functions, such as SetSource, were removed in VTK 6. The main reason behind this is that it is impossible to preserve this method while decoupling data objects from pipeline objects. In VTK 5,

SetInput() was implemented under the hood using SetInputConnection() but this required access to the algorithm and its output port given a data object. In VTK 6, since the data object no longer has a reference to the algorithm that produced it, it is not possible to establish a pipeline connection given only a data object.

In order to make it easy to assign a stand-along data object as an input to an algorithm, we introduced a set of convenience functions in VTK 6. Below is an example:

```
someFilter->SetInputData(aDataObject);
```

Note that even though the following will compile, it will NOT create a pipeline connection and should not be used in place of SetInputConnection().

```
someFilter->SetInputData(someReader->GetOutput());
```

Another advantage of decoupling data objects from pipeline objects is that developers no longer need to create shallow copies of inputs in order to use internal filters. Previously, this was required for an algorithm to function properly:

```
void MyFilter::RequestData(…)
{
  vtkDataObject* input = inInfo->Get(vtkDataObject::DATA_OBJECT());
  vtkDataObject* copy = input->NewInstance();
  copy->ShallowCopy(input);
  this->InternalFilter->SetInput(copy);
  this->InternalFilter->Update();
  …
}
```

This can now be replaced with the following without any unexpected side effects:

```
void MyFilter::RequestData(…)
{
  vtkDataObject* input = inInfo->Get(vtkDataObject::DATA_OBJECT());
  this->InternalFilter->SetInputData(copy);
  this->InternalFilter->Update();
  …
}
```

Another advantage is that this change removes some circular references from the pipeline making it unnecessary to use the garbage collector. This should have a noticeable impact on the performance of VTK when using large pipelines.

Miscellaneous Changes

Certain classes in VTK 5 provided only a SetInput(vtkDataObject*) or similar method and not a SetInputConnection(vtkAlgorithmOutput*) (or similar) method. In all of these classes SetInput() was implemented by storing a reference to a data object. None of them were subclasses of vtkAlgorithm. We replaced all of these methods with SetInputData() with the caveat that these classes no longer update any pipeline that may be associated with their input. When using these classes, the developer has to explicitly call Update() on any associated pipeline objects. These classes are:

vtkParallelCoordinatesActor
vtkTkRenderWidget
vtkEncodedGradientEstimator
vtkGPUVolumeRayCastMapper (no longer updated “TransformedInput”)
vtkPKdTree
vtkBarChartActor
vtkCaptionActor2D
vtkCubeAxesActor2D
vtkGridTransform
vtkLegendBoxActor
vtkPieChartActor
vtkSpiderPlotActor
vtkXYPlotActor
vtk3DWidget
vtkBalloonRepresentation
vtkCheckerboardRepresentation
vtkLogoRepresentation
vtkPolyDataSourceWidget
vtkSCurveSpline
vtkKdTree
vtkImplicitDataSet
vtkImplicitVolume
vtkKochanekSpline
vtkCardinalSpline

In addition, we changed or removed a few algorithms that produced variable number of outputs. This is not supported in VTK 5 without the backwards compatibility later. Such algorithms need to produced vtkMultiBlockDataSet instead. We removed vtkPLOT3DReader. Use vtkMultiBlockPLOT3DReader instead. We changed vtkProcrustesAlignmentFilter to produce a multi-block dataset as well as have one input port that accepts multiple connections.

Specific Description of Changes and Recipes for Updating Existing Code

Removal of SetInput()

This change will probably have the biggest impact on the VTK community. At the same time, it is the easiest to fix. Here is a rule of thumb:

- If you want to establish a pipeline connection, use SetInputConnection
- If you want to process a stand-alone dataset, use SetInputData

Here are some specific examples. Note that even though we use SetInput() in all of these examples, similar methods such as SetSource() should follow the same pattern. Look for SetSourceConnection() or SetSourceData() for example. Furthermore, we ignore the port number in these examples but most should work with a port number.

Example 1

```
anotherFilter->SetInput(aFilter->GetOutput());
```

should become

```
anotherFilter->SetInputConnection(aFilter->GetOutputPort());
```

Example 2

```
vtkDataObject* output = aFilter->GetOutput();
anotherFilter->SetInput(output);
```

should become

```
anotherFilter->SetInputConnection(aFilter->GetOutputPort());
```

Example 3

```
vtkPolyData *pd = vtkPolyData::New();
aFilter->SetInput(pd);
```

should become

```
vtkPolyData *pd = vtkPolyData::New();
aFilter->SetInputData(pd);
```

Example 4

```
vtkDataObject* output = aFilter->GetOutput();
```

```
aFilter->Update();
anotherFilter->SetInput(output);
```

This is ambiguous. If aFilter or any other source/filter upstream in never modified after this point, the following is valid:

```
vtkDataObject* output = aFilter->GetOutput();
aFilter->Update();
anotherFilter->SetInputData(output);
```

Otherwise, SetInputConnection() should be used. Remember that SetInputData() does not establish a pipeline connection and therefore updating anotherFilter can never cause any other filter to update. Here, anotherFilter will always process output after the manual update of aFilter.

Example 5

```
void myfunction(vtkDataObject* dobj)
{
 vtkAFilter* aFilter = vtkAFilter::New();
 aFilter->SetInput(dobj);
 aFilter->Update();
 // …
}
```

This example is also ambiguous. You need to trace it up to find the origin of dobj. If this is the common use:

```
myfunction(aFilter->GetOutput());
```

you will have to refactor myfunction to take an algorithm and an output port. For example:

```
void myfunction(vtkAlgorithm* alg, int port)
{
 vtkAFilter* aFilter = vtkAFilter::New();
 aFilter->SetInputConnection(alg->GetOutputPort(port));
 aFilter->Update();
 // …
}

myfunction(aFilter, 0);
```

If this is the common use:

```
vtkPolyData* pd = vtkPolyData::New();
```

```
// fill pd
myfunction(pd);
```

then replacing SetInput() with SetInputData() would work.

Example 6

```
class foo
{
 vtkDataObject* DataObject;
 void Process()
  {
  vtkAFilter* aFilter = vtkAFilter::New();
  aFilter->SetInput(this->DataObject);
  aFilter->Update();
  // …
  }
};
```

This is almost identical to the previous example. Refer to that one for the suggested solution.

Removal of vtkDataObject Pipeline Methods

Many of the pipeline methods that were removed from vtkDataObject can be easily replaced by calling them directly on the algorithm instead. See below for some examples.

Example 1

Replace:

```
vtkDataObject* dobj = aFilter->GetOutput(1);
dobj->Update();
```

with:

```
aFilter->Update(1);
```

Example 2

```
vtkDataObject* dobj = aFilter->GetOutput();
dobj->UpdateInformation();
dobj->SetUpdateExtent(0 /*piece*/,  2 /*number of pieces*/);
dobj->Update();
```

This is a bit more involved but still pretty easy:

```
aFilter->UpdateInformation();
vtkStreamingDemandDrivenPipeline::SetUpdateExtent(
      aFilter->GetOutputInformation(0 /*port number*/),
      0 /*piece*/,
      2 /*number of pieces*/,
      0 /*number of ghost levels);
aFilter->Update();
```

Example 3

```
vtkDataObject* dobj = aFilter->GetOutput();
dobj->UpdateInformation();
int* wExt = dobj->GetWholeExtent();
```

can be replaced with:

```
aFilter->UpdateInformation();
int* wExt = aFilter->GetOutputInformation(0)->Get(
  vtkStreamingDemandDrivenPipeline::WHOLE_EXTENT());
```

Example 4

```
void myfunction(vtkDataObject* dobj)
{
  dobj->Update();
  // Do something with dobj
}
```

It is possible that the Update() call here is unnecessary – if dobj was not produced by a filter or the filter that produced it was already updated. In this case, Update() can be removed. However, in all other case, you need to refactor this code. For example:

```
vtkAReader* aReader = vtkAReader::New();
aReader->SetFileName("something.foo");
myfunction(aReader->GetOutput());
```

would be refactored as:

```
void myfunction(vtkAlgorithm* alg)
{
 alg->Update();
 vtkDataObject* dobj = alg->GetOutputDataObject(0);
  // Do something with dobj
}
```

```
vtkAReader* aReader = vtkAReader::New();
aReader->SetFileName("something.foo");
myfunction(aReader);
```

Example 5

```
class foo
{
 vtkDataObject* Input; void SetInput(vtkDataObject* dobj)
  {
  this->Input = dobj;
  }

void Process()
   {
   vtkAFilter* aFilter = vtkAFilter::New();
   aFilter->SetInput(this->Input);
   aFilter->Update();
   // …
   }
};
```

There are two ways of refactoring this class:

1. Backwards compatible (behavior-wise)

```
class Foo
{
 vtkAlgorithm* Input;
 int Port;

 void SetInputConnection(vtkAlgorithmOutput* output)
  {
  this->Input = output->GetProducer();
  this->Port = output->GetIndex();
  }

void Process()
   {
   vtkAFilter* aFilter = vtkAFilter::New();
   aFilter->SetInputConnection(this->Input->GetOutputPort(
     this->Index));
   aFilter->Update();
   // …
   }
```

};

2. Backwards incompatible (behavior-wise)

The code for this version would be identical to the original code. However, the behavior would be such that this would fail:

```
vtkAReader* aReader = vtkAReader::New();
aReader->SetFileName("something.foo");

Foo afoo;
afoo.SetInput(aReader->GetOutput());
afoo.Process();
```

This could be fixed by adding an Update() call as follows:

```
vtkAReader* aReader = vtkAReader::New();
aReader->SetFileName("something.foo");
aReader->Update();

Foo afoo;
afoo.SetInput(aReader->GetOutput());
afoo.Process();
```

In situations like this, we recommend renaming the SetInput() function to SetInputData() so that the change in behavior is also followed by a change in API, telling users of Foo that they need to change their code at compile time.

Changes to Algorithms

Overall the changes introduced as part of this work do not change the behavior or backwards compatibility of algorithms. However, some of the algorithms continued to use the data object API to create, copy and read pipeline meta-data. This is specially true if you have implemented algorithms that used ExecuteInformation() for VTK 4 and then used scripts provided in VTK 5 to migrate your classes. These classes will no longer compile.

Example 1

```
void vtkMyReader::ExecuteInformation()
{
  vtkImageData* output = this->GetOutput();
  output->SetWholeExtent(...);
  output->SetScalarType(VTK_UNSIGNED_CHAR);
}
```

This needs to be replaced with:

```
int vtkMyReader::RequestInformation(vtkInformation*, vtkInformationVector**,
      vtkInformationVector* outInfoVec)
{
  vtkInformation* outInfo = outInfoVec->GetInformationObject(0);
  outInfo->Set(vtkStreamingDemandDrivenPipeline::WHOLE_EXTENT(), ...);
   vtkDataObject::SetPointDataActiveScalarInfo(
     outInfo, VTK_UNSIGNED_CHAR, 1);
  return 1;
}
```

Changes to Image Data and Image Data Algorithms

Some minor changes were made to vtkImageData and vtkImageAlgorithm due to the removal of the pipeline meta-data from data objects. If you subclass from vtkImageAlgorithm or heavily use vtkImageData in your code, there is a good chance that you will have to make some minor adjustments.

The major change to vtkImageData is the removal of the pipeline meta-data API. This includes methods like Set/GetScalarType, Set/GetNumberOfScalarComponents.

Example 1

In a subclass of vtkImageAlgorithm, the following

```
int vtkMyAlg::RequestInformation(vtkInformation*, vtkInformationVector**,
      vtkInformationVector* outInfoVec)
{
  vtkImageData* output = this->GetOutput();
  output->SetWholeExtent(...);
  output->SetScalarType(VTK_UNSIGNED_CHAR);
}
```

needs to be replaced with:

```
int vtkMyAlg::RequestInformation(vtkInformation*, vtkInformationVector**,
      vtkInformationVector* outInfoVec)
{
  vtkInformation* outInfo = outInfoVec->GetInformationObject(0);
  outInfo->Set(vtkStreamingDemandDrivenPipeline::WHOLE_EXTENT(), ...);
   vtkDataObject::SetPointDataActiveScalarInfo(
     outInfo, VTK_UNSIGNED_CHAR, 1);
  return 1;
}
```

Example 2

Replace:

```
vtkImageData* imageData = vtkImageData::New();
imageData->SetExtent(…);
imageData->SetScalarType(VTK_FLOAT);
imageData->SetNumberOfScalarComponents(3);
imageData->AllocateScalars();
```

with:

```
vtkImageData* imageData = vtkImageData::New();
imageData->SetExtent(…);
imageData->AllocateScalars(VTK_FLOAT, 3);
```

Example 3

Replace:

```
void vtkImageGridSource::ExecuteData(vtkDataObject *output)
{
 vtkImageData *data = this->AllocateOutputData(output);
```

with

```
void vtkImageGridSource::ExecuteData(vtkDataObject *output,
                                     vtkInformation* outInfo)
{
 vtkImageData *data = this->AllocateOutputData(output, outInfo);
```

Note that the reason behind this change is that the output image data no longer stores information about the output extent, scalar type and number of scalar components.

Miscellaneous

The following are examples of some miscellaneous changes introduced by VTK 6, and recipes for updating code.

Example 1

Replace

```
vtkPLOT3DReader pl3d
   pl3d SetXYZFileName "$VTK_DATA_ROOT/Data/combxyz.bin"
   pl3d SetQFileName "$VTK_DATA_ROOT/Data/combq.bin"
   pl3d Update
vtkPlane plane
```

```
   eval plane SetOrigin [[pl3d GetOutput] GetCenter]
   plane SetNormal -0.287 0 0.9579
```

with

```
vtkMultiBlockPLOT3DReader pl3d
   pl3d SetXYZFileName "$VTK_DATA_ROOT/Data/combxyz.bin"
   pl3d SetQFileName "$VTK_DATA_ROOT/Data/combq.bin"
   pl3d Update
   set output [[pl3d GetOutput] GetBlock 0]
vtkPlane plane
   eval plane SetOrigin [$output GetCenter]
```

Note the use of [[pl3d GetOutput] GetBlock 0]. It is also possible to create a pipeline that works on one block (or more) by using the composite data pipeline and vtkExtractBlock filter.

Example 2

Replace:

```
vtkProcrustesAlignmentFilter procrustes
   procrustes SetNumberOfInputs 3
   procrustes SetInput 0 [sphere GetOutput]
   procrustes SetInput 1 [transformer1 GetOutput]
   procrustes SetInput 2 [transformer2 GetOutput]
   [procrustes GetLandmarkTransform] SetModeToRigidBody

vtkPolyDataMapper map2a
   map2a SetInput [procrustes GetOutput 0]
```

with:

```
vtkMultiBlockDataGroupFilter group
   group AddInputConnection [sphere GetOutputPort]
   group AddInputConnection [transformer1 GetOutputPort]
   group AddInputConnection [transformer2 GetOutputPort]

vtkProcrustesAlignmentFilter procrustes
   procrustes SetInputConnection [group GetOutputPort]
   [procrustes GetLandmarkTransform] SetModeToRigidBody
   procrustes Update

vtkPolyDataMapper map2a
   map2a SetInputData [[procrustes GetOutput] GetBlock 0]
```

Appendix

In the appendix, we list the more salient API changes to some of the core classes. These are from diffs generated using Git. Lines that start by – are the lines that were removed whereas lines that start with + are lines that were added. Other lines are for context only and were not changed.

Changes to vtkDataObject

Below is a list of changes to the vtkDataObject API obtained from Git diff of vtkDataObject.h

```
  // Description:
-  // Get/Set the pipeline information object that owns this data
-  // object.
-  vtkGetObjectMacro(PipelineInformation, vtkInformation);
-  virtual void SetPipelineInformation(vtkInformation*);
-
-  // Description:
-  // Get the port currently producing this object.
-  virtual vtkAlgorithmOutput* GetProducerPort();
-
-  // Description:
   // Data objects are composite objects and need to check each part for MTime.
   // The information object also needs to be considered.
   unsigned long int GetMTime();

   // Description:
-  // Return flag indicating whether data should be released after use
-  // by a filter.
-  int ShouldIReleaseData();
-
-  // Description:
-  // Turn on/off flag to control whether this object's data is released
-  // after being used by a filter.
-  void SetReleaseDataFlag(int);
-  int GetReleaseDataFlag();
-  vtkBooleanMacro(ReleaseDataFlag,int);

   // Description:
-  // Provides opportunity for the data object to insure internal
-  // consistency before access. Also causes owning source/filter
-  // (if any) to update itself. The Update() method is composed of
-  // UpdateInformation(), PropagateUpdateExtent(),
-  // TriggerAsynchronousUpdate(), and UpdateData().
-  virtual void Update();
```

-
- // Description:
- // WARNING: INTERNAL METHOD - NOT FOR GENERAL USE.
- // THIS METHOD IS PART OF THE PIPELINE UPDATE FUNCTIONALITY.
- // Update all the "easy to update" information about the object such
- // as the extent which will be used to control the update.
- // This propagates all the way up then back down the pipeline.
- // As a by-product the PipelineMTime is updated.
- virtual void UpdateInformation();
-
- // Description:
- // WARNING: INTERNAL METHOD - NOT FOR GENERAL USE.
- // THIS METHOD IS PART OF THE PIPELINE UPDATE FUNCTIONALITY.
- // The update extent for this object is propagated up the pipeline.
- // This propagation may early terminate based on the PipelineMTime.
- virtual void PropagateUpdateExtent();
-
- // Description:
- // WARNING: INTERNAL METHOD - NOT FOR GENERAL USE.
- // THIS METHOD IS PART OF THE PIPELINE UPDATE FUNCTIONALITY.
- // Propagate back up the pipeline for ports and trigger the update on the
- // other side of the port to allow for asynchronous parallel processing in
- // the pipeline.
- // This propagation may early terminate based on the PipelineMTime.
- virtual void TriggerAsynchronousUpdate();
-
- // Description:
- // WARNING: INTERNAL METHOD - NOT FOR GENERAL USE.
- // THIS METHOD IS PART OF THE PIPELINE UPDATE FUNCTIONALITY.
- // Propagate the update back up the pipeline, and perform the actual
- // work of updating on the way down. When the propagate arrives at a
- // port, block and wait for the asynchronous update to finish on the
- // other side.
- // This propagation may early terminate based on the PipelineMTime.
- virtual void UpdateData();
-
- // Description:
- // Get the estimated size of this data object itself. Should be called
- // after UpdateInformation() and PropagateUpdateExtent() have both been
- // called. Should be overridden in a subclass - otherwise the default
- // is to assume that this data object requires no memory.
- // The size is returned in kilobytes.
- virtual unsigned long GetEstimatedMemorySize();
-
- // Description:
- // A generic way of specifying an update extent. Subclasses

```
- // must decide what a piece is.  When the NumberOfPieces is zero, then
- // no data is requested, and the source will not execute.
- virtual void SetUpdateExtent(int piece,int numPieces, int ghostLevel);
- void SetUpdateExtent(int piece, int numPieces)
-   {this->SetUpdateExtent(piece, numPieces, 0);}
-
- // Description:
- // Set the update extent for data objects that use 3D extents. Using this
- // method on data objects that set extents as pieces (such as vtkPolyData or
- // vtkUnstructuredGrid) has no real effect.
- // Don't use the set macro to set the update extent
- // since we don't want this object to be modified just due to
- // a change in update extent. When the volume of the extent is zero (0, -1,..),
- // then no data is requested, and the source will not execute.
- virtual void SetUpdateExtent(int x0, int x1, int y0, int y1, int z0, int z1);
- virtual void SetUpdateExtent(int extent[6]);
- virtual int* GetUpdateExtent();
- virtual void GetUpdateExtent(int& x0, int& x1, int& y0, int& y1,
-                              int& z0, int& z1);
- virtual void GetUpdateExtent(int extent[6]);
-
  // Description:
- // If the whole input extent is required to generate the requested output
- // extent, this method can be called to set the input update extent to the
- // whole input extent. This method assumes that the whole extent is known
- // (that UpdateInformation has been called)
- void SetUpdateExtentToWholeExtent();
-
- // Description:
- // Get the cumulative modified time of everything upstream.  Does
- // not include the MTime of this object.
- unsigned long GetPipelineMTime();
-
  // Description:
- // Copy the generic information (WholeExtent ...)
- void CopyInformation( vtkDataObject *data );
-
- // Description:
  // By default, there is no type specific information
- virtual void CopyTypeSpecificInformation( vtkDataObject *data )
-   {this->CopyInformation( data );};
-
- // Description:
- // Set / Get the update piece and the update number of pieces. Similar
- // to update extent in 3D.
- void SetUpdatePiece(int piece);
```

```
- void SetUpdateNumberOfPieces(int num);
- virtual int GetUpdatePiece();
- virtual int GetUpdateNumberOfPieces();
-
- // Description:
- // Set / Get the update ghost level and the update number of ghost levels.
- // Similar to update extent in 3D.
- void SetUpdateGhostLevel(int level);
- virtual int GetUpdateGhostLevel();
-
- // Description:
- // This request flag indicates whether the requester can handle
- // more data than requested.  Right now it is used in vtkImageData.
- // Image filters can return more data than requested.  The the
- // consumer cannot handle this (i.e. DataSetToDataSetFitler)
- // the image will crop itself.  This functionality used to be in
- // ImageToStructuredPoints.
- virtual void SetRequestExactExtent(int flag);
- virtual int GetRequestExactExtent();
- vtkBooleanMacro(RequestExactExtent, int);
-
- // Description:
- // Set/Get the whole extent of this data object.
- // The whole extent is meta data for structured data sets.
- // It gets set by the source during the update information call.
- virtual void SetWholeExtent(int x0, int x1, int y0, int y1, int z0, int z1);
- virtual void SetWholeExtent(int extent[6]);
- virtual int* GetWholeExtent();
- virtual void GetWholeExtent(int& x0, int& x1, int& y0, int& y1,
-                             int& z0, int& z1);
- virtual void GetWholeExtent(int extent[6]);
-
- // Description:
- // Set/Get the whole bounding box of this data object.
- // The whole whole bounding box is meta data for data sets
- // It gets set by the source during the update information call.
- virtual void SetWholeBoundingBox(double x0, double x1, double y0,
-                                  double y1, double z0, double z1);
- virtual void SetWholeBoundingBox(double bb[6]);
- virtual double* GetWholeBoundingBox();
- virtual void GetWholeBoundingBox(double& x0, double& x1, double& y0,
-                                  double& y1, double& z0, double& z1);
- virtual void GetWholeBoundingBox(double extent[6]);
-
- // Description:
- // Set/Get the maximum number of pieces that can be requested.
```

```
- // The maximum number of pieces is meta data for unstructured data sets.
- // It gets set by the source during the update information call.
- // A value of -1 indicates that there is no maximum.
- virtual void SetMaximumNumberOfPieces(int);
- virtual int GetMaximumNumberOfPieces();

- // Description:
- // Copy information about this data object to the output
- // information from its own Information for the given
- // request.  If the second argument is not NULL then it is the
- // pipeline information object for the input to this data object's
- // producer. If forceCopy is true, information is copied even
- // if it exists in the output.
- virtual void CopyInformationToPipeline(vtkInformation* request,
-                                        vtkInformation* input,
-                                        vtkInformation* output,
-                                        int forceCopy);
-
- // Description:
- // Calls CopyInformationToPipeline(request, input, this->PipelineInformation, 0).
- // Subclasses should not override this method (not virtual)
- void CopyInformationToPipeline(vtkInformation* request,
-                                vtkInformation* input)
-   {
-     this->CopyInformationToPipeline(request, input, this->PipelineInformation, 0);
-   }
-
- // Description:
- // Copy information about this data object from the
- // PipelineInformation to its own Information for the given request.
- virtual void CopyInformationFromPipeline(vtkInformation* request);

+   // Description:
+   // Copy information about this data object from the
+   // pipeline information to its own Information.
+  // Called right before the main execution pass..
+  virtual void CopyInformationFromPipeline(vtkInformation* vtkNotUsed(info))
+  {}

   // Description:
- // An object that will translate pieces into structured extents.
- void SetExtentTranslator(vtkExtentTranslator* translator);
- vtkExtentTranslator* GetExtentTranslator();
-
   // Description:
   // This method crops the data object (if necesary) so that the extent
```

```
  // matches the update extent.
-  virtual void Crop();
+  virtual void Crop(const int* updateExtent);

-  // Get the executive that manages this data object.
-  vtkExecutive* GetExecutive();
-
-  // Get the port number producing this data object.
-  int GetPortNumber();
-
-  // Reference the pipeline information object that owns this data
-  // object.
-  vtkInformation* PipelineInformation;
-
```

Changes to vtkImageData

Below is a list of changes to the vtkImageData API obtained from Git diff of vtkImageData.h

```
  // Description:
  // Set / Get the extent on just one axis
-  virtual void SetAxisUpdateExtent(int axis, int min, int max);
-  virtual void GetAxisUpdateExtent(int axis, int &min, int &max);
-
-  // Description:
-  // Override to copy information from pipeline information to data
-  // information for backward compatibility.  See
-  // vtkDataObject::UpdateInformation for details.
-  virtual void UpdateInformation();
+  virtual void SetAxisUpdateExtent(int axis, int min, int max,
+                                   const int* updateExtent,
+                                   int* axisUpdateExtent);
+  virtual void GetAxisUpdateExtent(int axis, int &min, int &max, const int*
updateExtent);

  // Description:
  // Set/Get the extent. On each axis, the extent is defined by the index
@@ -170,20 +166,16 @@ public:
  vtkGetVector6Macro(Extent, int);

  // Description:
-  // Get the estimated size of this data object itself. Should be called
-  // after UpdateInformation() and PropagateUpdateExtent() have both been
-  // called. This estimate should be fairly accurate since this is structured
-  // data.
-  virtual unsigned long GetEstimatedMemorySize();
```

```
-
-  // Description:
   // These returns the minimum and maximum values the ScalarType can hold
   // without overflowing.
+  virtual double GetScalarTypeMin(vtkInformation* meta_data);
   virtual double GetScalarTypeMin();
+  virtual double GetScalarTypeMax(vtkInformation* meta_data);
   virtual double GetScalarTypeMax();

   // Description:
   // Get the size of the scalar type in bytes.
+  virtual int GetScalarSize(vtkInformation* meta_data);
   virtual int GetScalarSize();

   // Description:
@@ -224,8 +216,17 @@ public:
    int x, int y, int z, int component, double v);

   // Description:
-  // Allocate the vtkScalars object associated with this object.
-  virtual void AllocateScalars();
+  // Allocate the point scalars for this dataset. The data type determines
+  // the type of the array (VTK_FLOAT, VTK_INT etc.) where as numComponents
+  // determines its number of components.
+  virtual void AllocateScalars(int dataType, int numComponents);
+
+  // Description:
+  // Allocate the point scalars for this dataset. The data type and the
+  // number of components of the array is determined by the meta-data in
+  // the pipeline information. This is usually produced by a reader/filter
+  // upstream in the pipeline.
+  virtual void AllocateScalars(vtkInformation* pipeline_info);

   // Description:
   // This method is passed a input and output region, and executes the filter
@@ -239,10 +240,10 @@ public:
    this->CopyAndCastFrom(inData, e);}

   // Description:
-  // Reallocates and copies to set the Extent to the UpdateExtent.
+  // Reallocates and copies to set the Extent to updateExtent.
   // This is used internally when the exact extent is requested,
   // and the source generated more than the update extent.
-  virtual void Crop();
+  virtual void Crop(const int* updateExtent);
```

```
  // Description:
  // Return the actual size of the data in kilobytes. This number
@@ -269,30 +270,9 @@ public:
  vtkSetVector3Macro(Origin,double);
  vtkGetVector3Macro(Origin,double);

-  // Description:
-  // Set/Get the data scalar type (i.e VTK_DOUBLE). Note that these methods
-  // are setting and getting the pipeline scalar type. i.e. they are setting
-  // the type that the image data will be once it has executed. Until the
-  // REQUEST_DATA pass the actual scalars may be of some other type. This is
-  // for backwards compatibility
-  void SetScalarTypeToFloat(){this->SetScalarType(VTK_FLOAT);};
-  void SetScalarTypeToDouble(){this->SetScalarType(VTK_DOUBLE);};
-  void SetScalarTypeToInt(){this->SetScalarType(VTK_INT);};
-  void SetScalarTypeToUnsignedInt()
-    {this->SetScalarType(VTK_UNSIGNED_INT);};
-  void SetScalarTypeToLong(){this->SetScalarType(VTK_LONG);};
-  void SetScalarTypeToUnsignedLong()
-    {this->SetScalarType(VTK_UNSIGNED_LONG);};
-  void SetScalarTypeToShort(){this->SetScalarType(VTK_SHORT);};
-  void SetScalarTypeToUnsignedShort()
-    {this->SetScalarType(VTK_UNSIGNED_SHORT);};
-  void SetScalarTypeToUnsignedChar()
-    {this->SetScalarType(VTK_UNSIGNED_CHAR);};
-  void SetScalarTypeToSignedChar()
-    {this->SetScalarType(VTK_SIGNED_CHAR);};
-  void SetScalarTypeToChar()
-    {this->SetScalarType(VTK_CHAR);};
-  void SetScalarType(int);
+  static void SetScalarType(int, vtkInformation* meta_data);
+  static int GetScalarType(vtkInformation* meta_data);
+  static bool HasScalarType(vtkInformation* meta_data);
  int GetScalarType();
  const char* GetScalarTypeAsString()
    { return vtkImageScalarTypeNameMacro ( this->GetScalarType() ); };
@@ -300,7 +280,9 @@ public:
  // Description:
  // Set/Get the number of scalar components for points. As with the
  // SetScalarType method this is setting pipeline info.
-  void SetNumberOfScalarComponents( int n );
+  static void SetNumberOfScalarComponents( int n, vtkInformation* meta_data);
+  static int GetNumberOfScalarComponents(vtkInformation* meta_data);
+  static bool HasNumberOfScalarComponents(vtkInformation* meta_data);
  int GetNumberOfScalarComponents();
```

```
  // Must only be called with vtkImageData (or subclass) as input
@@ -309,11 +291,7 @@ public:
  // Description:
  // Override these to handle origin, spacing, scalar type, and scalar
  // number of components.  See vtkDataObject for details.
-  virtual void CopyInformationToPipeline(vtkInformation* request,
-                                         vtkInformation* input,
-                                         vtkInformation* output,
-                                         int forceCopy);
-  virtual void CopyInformationFromPipeline(vtkInformation* request);
+  virtual void CopyInformationFromPipeline(vtkInformation* information);

  // Description:
  // make the output data ready for new data to be inserted. For most
@@ -378,7 +356,7 @@ protected:

  void ComputeIncrements();
  void ComputeIncrements(vtkIdType inc[3]);
-  void CopyOriginAndSpacingFromPipeline();
+  void CopyOriginAndSpacingFromPipeline(vtkInformation* info);
```

Changes to vtkAlgorithm

Below is a list of changes to the vtkAlgorithm API obtained from Git diff of vtkAlgorithm.h

```
  // Description:
+  // Remove a connection given by index idx.
+  virtual void RemoveInputConnection(int port, int idx);
+
+  // Description:
+  // Removes all input connections.
+  virtual void RemoveAllInputConnections(int port);
+
+  // Description:
  // Get a proxy object corresponding to the given output port of this
  // algorithm.  The proxy object can be passed to another algorithm's
  // SetInputConnection(), AddInputConnection(), and
@@ -320,13 +328,66 @@ public:
  vtkAlgorithmOutput* GetInputConnection(int port, int index);

  // Description:
+  // Returns the algorithm connected to a port-index pair.
+  vtkAlgorithm* GetInputAlgorithm(int port, int index);
+
+  // Description:
```

```
+  // Equivalent to GetInputAlgorithm(0, 0).
+  vtkAlgorithm* GetInputAlgorithm()
+  {
+    return this->GetInputAlgorithm(0, 0);
+  }
+
+  // Description:
+  // Returns the executive associated with a particular input
+  // connection.
+  vtkExecutive* GetInputExecutive(int port, int index);
+
+  // Description:
+  // Equivalent to GetInputExecutive(0, 0)
+  vtkExecutive* GetInputExecutive()
+  {
+    return this->GetInputExecutive(0, 0);
+  }
+
+  // Description:
+  // Return the information object that is associated with
+  // a particular input connection. This can be used to get
+  // meta-data coming from the REQUEST_INFORMATION pass and set
+  // requests for the REQUEST_UPDATE_EXTENT pass. NOTE:
+  // Do not use this in any of the pipeline passes. Use
+  // the information objects passed as arguments instead.
+  vtkInformation* GetInputInformation(int port, int index);
+
+  // Description:
+  // Equivalent to GetInputInformation(0, 0)
+  vtkInformation* GetInputInformation()
+  {
+    return this->GetInputInformation(0, 0);
+  }
+
+  // Description:
+  // Return the information object that is associated with
+  // a particular output port. This can be used to set
+  // meta-data coming during the REQUEST_INFORMATION. NOTE:
+  // Do not use this in any of the pipeline passes. Use
+  // the information objects passed as arguments instead.
+  vtkInformation* GetOutputInformation(int port);
+
+  // Description:
   // Bring this algorithm's outputs up-to-date.
+  virtual void Update(int port);
   virtual void Update();
```

```
+
   // Description:
-  // Backward compatibility method to invoke UpdateInformation on executive.
+  // Bring the algorithm's information up-to-date.
   virtual void UpdateInformation();

+  // Description::
+  // Propagate meta-data upstream.
+  virtual void PropagateUpdateExtent();
+
   // Description:
   // Bring this algorithm's outputs up-to-date.
   virtual void UpdateWholeExtent();
@@ -356,7 +417,7 @@ public:
   // This condition is satisfied when the UpdateExtent has
   // zero volume (0,-1,...) or the UpdateNumberOfPieces is 0.
   // The source uses this call to determine whether to call Execute.
-  int UpdateExtentIsEmpty(vtkDataObject *output);
+  int UpdateExtentIsEmpty(vtkInformation *pinfo, vtkDataObject *output);
   int UpdateExtentIsEmpty(vtkInformation *pinfo, int extentType);

   // Description:
@@ -379,6 +440,48 @@ public:
   static vtkInformationIntegerKey* PRESERVES_TOPOLOGY();
   static vtkInformationIntegerKey* PRESERVES_ATTRIBUTES();
   static vtkInformationIntegerKey* PRESERVES_RANGES();
+  static vtkInformationIntegerKey* MANAGES_METAINFORMATION();
+
+  // Description:
+  // If the whole input extent is required to generate the requested output
+  // extent, this method can be called to set the input update extent to the
+  // whole input extent. This method assumes that the whole extent is known
+  // (that UpdateInformation has been called).
+  // This function has no effect is input connection is not established.
+  int SetUpdateExtentToWholeExtent(int port, int connection);
+
+  // Description:
+  // Convenience function equivalent to SetUpdateExtentToWholeExtent(0, 0)
+  // This function has no effect is input connection is not established.
+  int SetUpdateExtentToWholeExtent();
+
+  // Description:
+  // Set the update extent in terms of piece and ghost levels.
+  // This function has no effect is input connection is not established.
+  void SetUpdateExtent(int port, int connection,
```

```
+                   int piece,int numPieces, int ghostLevel);
+
+  // Description:
+  // Convenience function equivalent to SetUpdateExtent(0, 0, piece,
+  // numPieces, ghostLevel)
+  // This function has no effect is input connection is not established.
+  void SetUpdateExtent(int piece,int numPieces, int ghostLevel)
+  {
+    this->SetUpdateExtent(0, 0, piece, numPieces, ghostLevel);
+  }
+
+  // Description:
+  // Set the update extent for data objects that use 3D extents
+  // This function has no effect is input connection is not established.
+  void SetUpdateExtent(int port, int connection, int extent[6]);
+
+  // Description:
+  // Convenience function equivalent to SetUpdateExtent(0, 0, extent)
+  // This function has no effect is input connection is not established.
+  void SetUpdateExtent(int extent[6])
+  {
+    this->SetUpdateExtent(0, 0, extent);
+  }

 protected:
   vtkAlgorithm();
@@ -539,6 +642,14 @@ protected:

   static vtkExecutive* DefaultExecutivePrototype;

+  // Description:
+  // These methods are used by subclasses to implement methods to
+  // set data objects directly as input. Internally, they create
+  // a vtkTrivialProducer that has the data object as output and
+  // connect it to the algorithm.
+  void SetInputDataInternal(int port, vtkDataObject *input);
+  void AddInputDataInternal(int port, vtkDataObject *input);
+
```