Generalized VTK Data Processing Framework

September 2003

Kitware, Inc.

1 Introduction

This document was written as a part of a contract between Kitware, Inc. and Trilabs (LANL, LLNL, and Sandia). It proposes a new generalized data processing framework for VTK that is powerful enough to encompass all current VTK pipeline functionality while allowing more advanced functionality to be added in a minimally-intrusive way.

2 Overview

The generalized VTK data processing framework consists of two classes of objects: processors and executives. A *processor* object is a basic worker unit that loads, manipulates, or stores, but does not own data. All processors are implemented as subclasses of `vtkProcessor`. An *executive* owns a processor and manages its execution. One or more executive objects and their corresponding processors may be used to implement a pipeline. The executives also own the data objects with which they instruct the processors to execute. All executives are implemented as subclasses of `vtkExecutive`.

Figure 1 shows the basic design for a `vtkProcessor`. Each processor defines zero or more input ports and output ports with certain restrictions on the kind of data and operations supported. The ports are read-only and do not hold references to data objects or information associated with the actual execution of the processor. All data and information for execution are passed to the processor by its executive when a request is made.

The processor's executive reads information from the ports and uses it to construct valid requests. Requests for information or data flowing from input to output are called *downstream* requests. Requests for information or data flowing from output to input are called *upstream* requests.

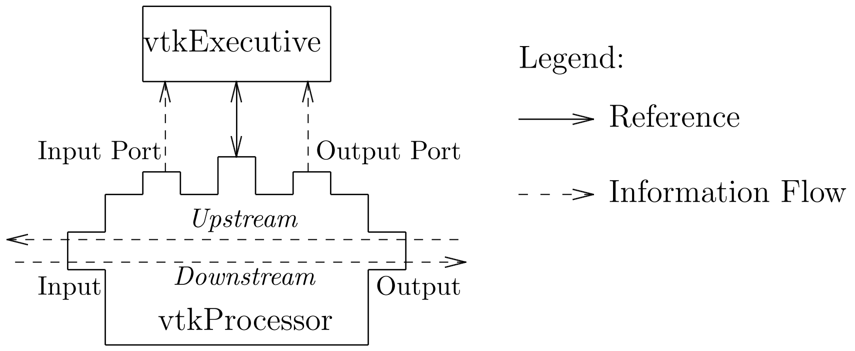

Figure 1: A vtkProcessor Object

3 The User's Point of View

Very little will change from the point of view of an application developer from the introduction of the proposed framework. Pipelines may still be created in the usual way:

```
vtkXMLUnstructuredGridReader* reader = vtkXMLUnstructuredGridReader::New();
vtkElevation* elevation = vtkElevation::New();
elevation->SetInput(reader->GetOutput());
elevation->Update();
```

Most existing code should still work without modification (see section 6). In fact, new code should be written using the same approach for creating pipelines. However, new capabilities will be available to replace the executive controlling a set of processors:

```
vtkPrioritizedStreamingPipeline* e = vtkPrioritizedStreamingPipeline::New();
vtkXMLUnstructuredGridReader* reader = vtkXMLUnstructuredGridReader::New();
vtkElevation* elevation = vtkElevation::New();
elevation->SetExecutive(e);
reader->SetExecutive(e);
elevation->SetInput(reader->GetOutput());
elevation->Update();
```

or alternatively:

```
vtkPrioritizedStreamingPipeline::SetAsDefaultExecutive();
vtkXMLUnstructuredGridReader* reader = vtkXMLUnstructuredGridReader::New();
vtkElevation* elevation = vtkElevation::New();
elevation->SetInput(reader->GetOutput());
elevation->Update();
```

4 The Developer's Point of View

The proposed framework will change VTK from the point of view of an internal developer or third-party filter writer. While filters written for the current VTK will still work (see section 6), new filters should be written under the proposed framework. We expect filter writing to be easier with the proposed design than it was previously.

The current VTK pipeline requires filters to implement a certain set of requests a specific way, and has no support for taking advantage of additional functionality provided by a filter. Filters in the existing architecture will not function properly if not written to support all requests defined by the current pipeline.

The proposed framework does not require a specific set of requests from a `vtkProcessor` or place restrictions on how the requests are implemented. It only requires a processor to specify the kinds of requests it supports. The executives can take advantage of the functionality a processor provides while working around the lack of support for other requests.

A `vtkProcessor` will report its input requirements and the downstream requests it supports through the input port information vector. It will implement a `FillInputPortInformation` method similar to this example:

```
  void FillInputPortInformation(vtkInformationVector* portInfo)
  {
    portInfo->SetNumberOfInformationObjects(1);
    vtkInformation* inputInfo = portInfo->GetInformationObject(0);
    vtkInformationVector* fieldInfoVector = vtkInformationVector::New();

    // Input requirements.
    inputInfo->Set(inputInfo->DATA_TYPE(), VTK_IMAGE_DATA);
    inputInfo->Set(inputInfo->INPUT_REQUIRED_FIELDS(), fieldInfoVector);
    fieldInfoVector->SetNumberOfInformationObjects(1);
    vtkInformation* fieldInfo = fieldInfoVector->GetInformationObject(0);
    fieldInfo->Set(fieldInfo->FIELD_ATTRIBUTE_TYPE(),
                   vtkDataSetAttributes::SCALARS);
    fieldInfo->Set(fieldInfo->FIELD_OPERATION(),
                   vtkInformation::FIELD_OPERATION_MODIFIED);
    fieldInfoVector->Delete();

    // Downstream requests supported.
    int downstream[] = { vtkInformation::KEY_DATA_OBJECT };
    inputInfo->Set(inputInfo->SUPPORTED_DOWNSTREAM_REQUESTS(),
                   downstream, sizeof(downstream)/sizeof(int));
  }
```

The example processor requires as input a vtkImageData instance with scalars data. It supports only execution as a downstream request because it will handle only a data object as input to such a request.

Upstream request support will be reported similarly through a `FillOutputPortInformation` method. The same processor would implement `ProcessDownstreamRequest` to handle the requests it reports in its input port information vector and similarly for `ProcessUpstreamRequest`.

5 Implementation

The following give prototype class declarations for the main classes involved in implementing the proposed design. Some internal implementation details unrelated to the design have been removed.

5.1 vtkProcessor

This is the superclass for all sources, filters, and sinks in VTK. It defines a generalized interface for executing data processing algorithms.

Instances may be used independently or within pipelines with a variety of architectures and update mechanisms. Pipelines are controlled by instances of `vtkExecutive`. Every `vtkProcessor` instance has an associated `vtkExecutive` when it is used in a pipeline. The executive is responsible for pipeline connectivity and data flow.

```
class vtkProcessor : public vtkObject
{
public:
  static vtkProcessor *New();
  vtkTypeRevisionMacro(vtkProcessor,vtkObject);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Description:
  // Check whether this processor has an assigned executive.  This
  // will NOT create a default executive.
  int HasExecutive();

  // Description:
  // Get this processor's executive.  If it has none, a default
  // executive will be created.
  vtkExecutive* GetExecutive();

  // Description:
  // Set this processor's executive.  This processor is removed from
  // any executive to which it has previously been assigned and then
  // assigned to the given executive.  A processor's executive should
  // not be changed after pipeline connectivity has been established.
  void SetExecutive(vtkExecutive* executive);

  // Description:
  // Upstream/Downstream requests form the generalized interface
  // through which executives invoke a processor's functionality.
  // Upstream requests correspond to information flow from the
  // processor's outputs to its inputs.
  //
  // An upstream request is defined by the contents of the output
  // information vector passed to ProcessUpstreamRequest.  The results
  // of an upstream request are stored in the input information vector
  // passed to ProcessUpstreamRequest.
  virtual int ProcessUpstreamRequest(vtkInformationVector* inInfo,
                                     vtkInformationVector* outInfo);

  // Description:
  // Upstream/Downstream requests form the generalized interface
  // through which executives invoke a processor's functionality.
  // Downstream requests correspond to information flow from the
  // processor's inputs to its outputs.
  //
  // An downstream request is defined by the contents of the input
  // information vector passed to ProcessDownstreamRequest.  The
  // results of an downstream request are stored in the output
  // information vector passed to ProcessDownstreamRequest.
  virtual int ProcessDownstreamRequest(vtkInformationVector* inInfo,
                                       vtkInformationVector* outInfo);

  // Description:
  // Get the information objects associated with this processor's
  // input ports.  There is one input port per input to the processor.
  // Each input port tells executives what kind of data and downstream
  // requests this processor can handle for that input.
  vtkInformationVector* GetInputPortInformation();

  // Description:
  // Get the information objects associated with this processor's
  // output ports.  There is one output port per output from the
  // processor.  Each output port tells executives what kind of
  // upstream requests this processor can handle for that output.
  vtkInformationVector* GetOutputPortInformation();

  // Description:
  // Set an input of this processor.  The input must correspond to the
  // output of another processor.
  //
  // TODO: Accept an actual vtkDataObject instance as input and use it
  // to establish the pipeline connection to maintain backward
  // compatibility.  See vtkExecutive::SetInput().
  void SetInput(int index, vtkProcessorOutput* input);

  // Description:
  // Get a proxy object corresponding to the given output of this
  // processor.  The proxy object can be passed to another processor's
  // SetInput method to establish a pipeline connection.
  //
  // TODO: Use an actual vtkDataObject instance as the proxy object to
  // maintain backward compatibility.  See vtkExecutive::GetOutput().
  vtkProcessorOutput* GetOutput(int index);

  // Description:
  // Bring this processor's outputs up-to-date.
  virtual void Update();

  // Description:
  // Decrement the count of references to this object.  This will
  // detect executive/processor reference loops and break them when
  // necessary.
```

```
  virtual void UnRegister(vtkObjectBase* o);

protected:
  vtkProcessor();
  ~vtkProcessor();

  // Description:
  // Fill the input port information objects for this processor.  This
  // is invoked by the first call to GetInputPortInformation so
  // subclasses can specify what they can handle.
  virtual void FillInputPortInformation(vtkInformationVector* portInfo);

  // Description:
  // Fill the output port information objects for this processor.
  // This is invoked by the first call to GetOutputPortInformation so
  // subclasses can specify what they can handle.
  virtual void FillOutputPortInformation(vtkInformationVector* portInfo);
};
```

5.2 vtkExecutive

This is the superclass for all pipeline executives in VTK. A VTK executive is responsible for controlling one or more instances of `vtkProcessor`. A pipeline consists of one or more executives that maintain the connectivity and control data flow. Every reader, source, writer, or data processing algorithm in the pipeline is implemented in an instance of `vtkProcessor`.

```
class vtkExecutive : public vtkObject
{
public:
  vtkTypeRevisionMacro(vtkExecutive,vtkObject);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Description:
  // Set an input of the given vtkProcessor instance.  The input must
  // correspond to the output of another processor.
  //
  // TODO: Accept an actual vtkDataObject instance as input and use it
  // to establish the pipeline connection to maintain backward
  // compatibility.
  virtual void SetInput(vtkProcessor* processor, int inIndex,
                        vtkProcessorOutput* input)=0;

  // Description:
  // Get a proxy object corresponding to an output of the given
  // vtkProcessor instance.  The processor must already be managed by
  // this executive.  The proxy object can be passed to this or
  // another executives's SetInput method to establish a pipeline
  // connection.
  //
  // TODO: Use an actual vtkDataObject instance as the proxy object to
  // maintain backward compatibility.  The instance should be used by
  // this executive as the processor's real output when it is
  // executed.
  virtual vtkProcessorOutput* GetOutput(vtkProcessor* processor,
                                        int outIndex)=0;

  // Description:
  // Bring the given processor's outputs up-to-date.  The processor
  // must already be managed by this executive.
  virtual void Update(vtkProcessor* processor)=0;

  // Description:
  // Decrement the count of references to this object.  This will
  // detect executive/processor reference loops and break them when
  // necessary.
  virtual void UnRegister(vtkObjectBase* o);

protected:
  vtkExecutive();
  ~vtkExecutive();

  // Description:
  // Called from vtkExecutive::UnRegister and vtkProcessor::UnRegister
  // to detect a connected component in the reference graph among
  // processors and executives and delete it if it has a net reference
  // count of 0.
  void CheckReferenceLoops();

  // Description:
  // Report to the given root executive references this executive
  // makes to other executives and processors.  This is used by
  // CheckReferenceLoops to detect isolated connected components in
  // the reference graph for arbitrary executive implementations.
  virtual void ReportReferences(vtkExecutive* root)=0;

  // Description:
  // Remove this executive's references to other executives and
  // processors.  This is used by CheckReferenceLoops to delete a
  // connected component in the reference graph that has a net
  // reference count of 0.
  virtual void RemoveReferences()=0;

  // Description:
  // Used by the ReportReferences method implementations in subclasses
  // to report a reference to an executive or processor.
  static void ReportReference(vtkExecutive* root, vtkExecutive* executive);
  static void ReportReference(vtkExecutive* root, vtkProcessor* processor);

  // Description:
  // Add/Remove a processor from the control of this executive.  Some
  // executives support more than one processor while others do not.
  // These methods are called by vtkProcessor::SetExecutive and should
  // not be called from elsewhere.
  virtual void AddProcessor(vtkProcessor* processor)=0;
  virtual void RemoveProcessor(vtkProcessor* processor)=0;
};
```

5.3 vtkDistributedExecutive

This is the superclass for VTK's distributed executives. Some pipeline architectures are more easily maintained with an executive instance per `vtkProcessor` instance. This class maintains pipeline connectivity in a distributed manner so subclasses can focus on their pipeline update designs.

```
class vtkDistributedExecutive : public vtkExecutive
{
public:
  static vtkDistributedExecutive* New();
  vtkTypeRevisionMacro(vtkDistributedExecutive,vtkObject);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Description:
  // See vtkExecutive::SetInput().
  virtual void SetInput(vtkProcessor* processor, int inIndex,
                        vtkProcessorOutput* input);

  // Description:
  // See vtkExecutive::GetOutput().
  virtual vtkProcessorOutput* GetOutput(vtkProcessor* processor,
                                        int outIndex);

  // Description:
  // Distributed executives have a one-to-one correspondence with
  // their processors.  Get the processor to which this executive has
  // been assigned.
  vtkProcessor* GetProcessor();

  // Description:
  // Bring the given processor's outputs up-to-date.  The processor
  // must already be managed by this executive.
  virtual void Update(vtkProcessor* processor);
protected:
  vtkDistributedExecutive();
  ~vtkDistributedExecutive();

  // Implement methods required by superclass.
  virtual void AddProcessor(vtkProcessor* processor);
  virtual void RemoveProcessor(vtkProcessor* processor);
  virtual void ReportReferences(vtkExecutive* root);
  virtual void RemoveReferences();
};
```

There will be several subclasses of `vtkDistributedExecutive` that support the old-style pipeline functionality. A `vtkDemandDrivenPipeline` class will derive directly from `vtkDistributedExecutive` to provide simple pipeline updates without streaming support. The full old-style pipeline will be provided by `vtkStreamingDemandDrivenPipeline`, which will derive from `vtkDemandDrivenPipeline`. New features such as prioritized streaming can be added by providing another executive.

5.4 vtkCentralizedExecutive

This is the superclass for VTK's centralized executives. Some pipeline architectures are more easily maintained with a single executive managing the entire pipeline. This class maintains pipeline connectivity in a centralized manner so subclasses can focus on their pipeline update designs. This class is present in this document only for completeness, but may not be implemented in practice until a need for it is found in the future.

```
class vtkCentralizedExecutive : public vtkExecutive
{
public:
  static vtkCentralizedExecutive* New();
  vtkTypeRevisionMacro(vtkCentralizedExecutive,vtkObject);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Description:
  // See vtkExecutive::SetInput().
  virtual void SetInput(vtkProcessor* processor, int inIndex,
                        vtkProcessorOutput* input);

  // Description:
```

```
  // See vtkExecutive::GetOutput().
  virtual vtkProcessorOutput* GetOutput(vtkProcessor* processor,
                                        int outIndex);

  // Description:
  // Bring the given processor's outputs up-to-date.  The processor
  // must already be managed by this executive.
  virtual void Update(vtkProcessor* processor);
protected:
  vtkCentralizedExecutive();
  ~vtkCentralizedExecutive();

  // Implement methods required by superclass.
  virtual void AddProcessor(vtkProcessor* processor);
  virtual void RemoveProcessor(vtkProcessor* processor);
  virtual void ReportReferences(vtkExecutive* root);
  virtual void RemoveReferences();
};
```

5.5 vtkInformation

This represents information and/or data for one input or one output of a `vtkProcessor`. It maps from keys to values of several data types. Instances of this class are collected in `vtkInformationVector` instances and passed to `vtkProcessor::ProcessUpstreamRequest` and `vtkProcessor::ProcessDownstreamRequest` calls. The information and data referenced by the instance on a particular input or output define the request made to the `vtkProcessor` instance.

```
class VTK_EXPORT vtkInformation : public vtkObject
{
public:
  static vtkInformation *New();
  vtkTypeRevisionMacro(vtkInformation,vtkObject);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Description:
  // Copy all information entries from the given vtkInformation
  // instance.  Any previously existing entries are removed.
  void Copy(vtkInformation* from);

  // Description:
  // Get/Set an integer-valued entry.
  void Set(vtkInformationIntegerKey* key, int value);
  int Get(vtkInformationIntegerKey* key);
  void Remove(vtkInformationIntegerKey* key);
```

```
int Has(vtkInformationIntegerKey* key);

// Description:
// Get/Set an integer-vector-valued entry.
void Set(vtkInformationIntegerVectorKey* key, int* value, int length);
int* Get(vtkInformationIntegerVectorKey* key);
void Get(vtkInformationIntegerVectorKey* key, int* value);
int Length(vtkInformationIntegerVectorKey* key);

// Description:
// Get/Set a string-valued entry.
void Set(vtkInformationStringKey* key, const char*);
const char* Get(vtkInformationStringKey* key);

// Description:
// Get/Set an entry storing another vtkInformation instance.
void Set(vtkInformationInformationKey* key, vtkInformation*);
vtkInformation* Get(vtkInformationInformationKey* key);

// Description:
// Get/Set an entry storing a vtkInformationVector instance.
void Set(vtkInformationInformationVectorKey* key, vtkInformationVector*);
vtkInformationVector* Get(vtkInformationInformationVectorKey* key);

// Description:
// Get/Set an entry storing a vtkDataSet instance.
void Set(vtkInformationDataSetKey* key, vtkDataSet*);
vtkDataSet* Get(vtkInformationDataSetKey* key);

//BTX
// Enumeration of standard information entry keys.  These should not
// be used directly.  Instead, call the method with the
// corresponding name to get an instance of the key.
enum Keys
{
  KEY_DATA_OBJECT,
  KEY_DATA_TYPE,
  KEY_INPUT_REQUIRED_FIELDS,
  KEY_FIELD_ATTRIBUTE_TYPE,
  KEY_FIELD_OPERATION,
  KEY_FIELD_NAME,
  KEY_UPDATE_EXTENT,
  KEY_SUPPORTED_UPSTREAM_REQUESTS,
  KEY_SUPPORTED_DOWNSTREAM_REQUESTS,
  KEY_USER_BEGIN=10000
  };

  // Description:
  // Possible values for the FIELD_OPERATION information entry.
  enum FieldOperations
  {
    FIELD_OPERATION_PRESERVED,
    FIELD_OPERATION_REINTERPOLATED,
    FIELD_OPERATION_MODIFIED,
    FIELD_OPERATION_REMOVED
  };
  //ETX

  // Description:
  // Get a key instance for the information entry specified by the
  // method name.
  vtkInformationIntegerKey* DATA_TYPE();
  vtkInformationInformationVectorKey* INPUT_REQUIRED_FIELDS();
  vtkInformationDataSetKey* DATA_OBJECT();
  vtkInformationStringKey* FIELD_NAME();
  vtkInformationIntegerKey* FIELD_ATTRIBUTE_TYPE();
  vtkInformationIntegerKey* FIELD_OPERATION();
  vtkInformationIntegerVectorKey* UPDATE_EXTENT();
  vtkInformationIntegerVectorKey* SUPPORTED_UPSTREAM_REQUESTS();
  vtkInformationIntegerVectorKey* SUPPORTED_DOWNSTREAM_REQUESTS();

  // Description:
  // Get the enumeration value associated with the given key instance.
  int GetKey(vtkInformationDataSetKey* key);
  int GetKey(vtkInformationInformationKey* key);
  int GetKey(vtkInformationInformationVectorKey* key);
  int GetKey(vtkInformationIntegerKey* key);
  int GetKey(vtkInformationIntegerVectorKey* key);
  int GetKey(vtkInformationKey* key);
  int GetKey(vtkInformationStringKey* key);

protected:
  vtkInformation();
  ~vtkInformation();

  // Get/Set a map entry directly through the vtkObjectBase instance
  // representing the value.  Used internally to manage the map.
  void SetAsObjectBase(vtkInformationKey* key, vtkObjectBase* value);
  vtkObjectBase* GetAsObjectBase(vtkInformationKey* key);
};
```

5.6 vtkInformationVector

vtkInformationVector stores a vector of zero or more vtkInformation objects corresponding to the input or output information for a vtkProcessor. An instance of this class is passed to vtkProcessor::ProcessUpstreamRequest and vtkProcessor::ProcessDownstreamRequest calls.

```
class VTK_EXPORT vtkInformationVector : public vtkObject
{
public:
  static vtkInformationVector *New();
  vtkTypeRevisionMacro(vtkInformationVector,vtkObject);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Description:
  // Get/Set the number of information objects in the vector.  Setting
  // the number to larger than the current number will create empty
  // vtkInformation instances.  Setting the number to smaller than the
  // current number will remove entries from higher indices.
  int GetNumberOfInformationObjects();
  void SetNumberOfInformationObjects(int n);

  // Description:
  // Get/Set the vtkInformation instance stored at the given index in
  // the vector.  The vector will automatically expand to include the
  // index given if necessary.  Missing entries in-between will be
  // filled with empty vtkInformation instances.
  void SetInformationObject(int index, vtkInformation* info);
  vtkInformation* GetInformationObject(int index);

  // Description:
  // Copy the set of vtkInformation objects from the given vector.
  // All data in the information objects will be duplicated and stored
  // in new instances.  Any existing vector contents are removed.
  void DeepCopy(vtkInformationVector* from);

  // Description:
  // Copy the set of vtkInformation objects from the given vector.
  // The same instances of vtkInformation will be referenced.  Any
  // existing vector contents are removed.
  void ShallowCopy(vtkInformationVector* from);

protected:
  vtkInformationVector();
  ~vtkInformationVector();
};
```

6 Backward Compatibility

Integrating the proposed framework into the current VTK will be very intrusive, but it is possible to provide some backward compatibility. Our goal is to maintain backward compatibility in two forms. First, applications using VTK filters to create pipelines should work with few or no changes. Second, existing VTK filters implemented within VTK or by outside projects should compile and run with as little modification as possible.

Support for existing application code will be straightforward because pipeline connectivity will still be specified using the SetInput/GetOutput idiom already in place. Although connectivity is no longer stored in the filter or data instances, enough information is available to generate it or request it from an executive as needed.

We will support existing filters by taking advantage of the existing class hierarchy of filters and modifying the top-most classes. The new `vtkProcessObject` will derive from `vtkProcessor`, and `vtkSource` will still derive from `vtkProcessObject`. These classes will override `ProcessUpstreamRequest` and `ProcessDownstreamRequest` to support a streaming, demand driven pipeline by translating requests to old-style pipeline update calls.

While rather strong backward compatibility can be maintained, it will not be possible to be fully compatibile. Filters or applications that abuse the current pipeline implementation by taking advantage of unspecified but known implementation details will not work without modification. We expect most of these cases to be situations in which the old-style pipeline was barely adequate but that will be supported by the proposed framework.

6.1 vtkProcessObject

This is the superclass for all old-style VTK readers, writers, and filters. Along with `vtkSource`, it translates upstream/downstream requests into old-style pipeline calls to subclasses.

```
class vtkProcessObject : public vtkProcessor
{
public:
  static vtkProcessObject *New();
  vtkTypeRevisionMacro(vtkProcessObject,vtkProcessor);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Description:
  // Get this processor's executive.  If it has none, a
  // vtkStreamingDemandDrivenPipeline will be created.
  vtkExecutive* GetExecutive();

  // ... other old vtkProcessObject methods ...

protected:
  vtkProcessObject();
  ~vtkProcessObject();
  // Description:
  // Provide subclasses with access to filter's input
  // as it was previously stored.  This will not actually own
  // any references to the input, but will be set correctly
  // before calls to subclasses are made.
  vtkDataObject** Inputs;

  // Description:
  // Fill the input port information objects for this processor.
  // Port requirements and capabilities will match those of the old
  // VTK vtkProcessObject inputs.
  virtual void FillInputPortInformation(vtkInformationVector* portInfo);

  // Description:
  // Fill the output port information objects for this processor.
  // There are no output ports because vtkProcessObject did not support
  // output.
  virtual void FillOutputPortInformation(vtkInformationVector* portInfo);
};
```

6.2 vtkSource

This is the superclass for all old-style VTK readers and filters. Along with `vtkProcessObject`, it translates upstream/downstream requests into old-style pipeline calls to subclasses.

```
class vtkSource : public vtkProcessObject
{
public:
  static vtkSource *New();
  vtkTypeRevisionMacro(vtkSource,vtkProcessObject);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Description:
  // Translate upstream requests into PropagateUpdateExtent
  // and TriggerAsynchronousUpdate calls.
  virtual int ProcessUpstreamRequest(vtkInformationVector* inInfo,
                                     vtkInformationVector* outInfo);

  // Description:
  // Translate downstream requests into Execute and ExecuteInformation calls.
  virtual int ProcessDownstreamRequest(vtkInformationVector* inInfo,
                                       vtkInformationVector* outInfo);

  // Old pipeline update methods:
  virtual void Execute();
  virtual void ExecuteInformation();
  virtual void PropagateUpdateExtent();
  virtual void TriggerAsynchronousUpdate();
  // ... other old vtkSource methods ...
protected:
  vtkSource();
  ~vtkSource();

  // Description:
  // Provide subclasses with access to filter's output
  // as it was previously stored.  This will not actually own
  // any references to the output, but will be set correctly
  // before calls to subclasses are made.
  vtkDataObject** Outputs;

  // Description:
  // Fill the output port information objects for this processor.
  // Port capabilities will match those of the old vtkSource outputs.
  virtual void FillOutputPortInformation(vtkInformationVector* portInfo);
};
```

6.3 Old-Style Pipeline Execution

The current VTK pipeline will be represented in the proposed framework by an exectuve class called `vtkStreamingDemandDrivenPipeline`. This will be the default executive created for `vtkProcessor` instances.

Figure 2 shows the three basic stages of the old-style VTK pipeline update process. First, the `UpdateInformation` phase propagates information about what data are available from the sources through all the filters to the sinks. Second, the `PropagateUpdateExtent` phase sends information about what data are desired from the sinks back through all the filters to the sources. Finally, the `Update` phase executes sources and then filters as necessary to produce the requested data. `vtkStreamingDemandDrivenPipeline` will imple-

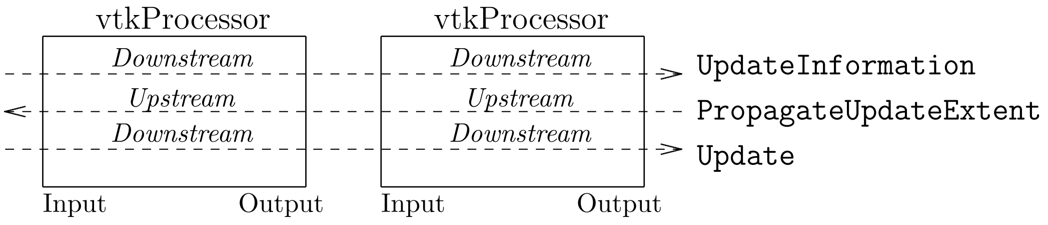

Figure 2: Streaming Demand Driven Pipeline Update

ment the old-style execution by making the proper calls to `ProcessDownstreamRequest` and `ProcessUpstreamRequest` on each `vtkProcessor` in the proper order.