How To Write A Process Object

This section describes how to create your own process object in VTK. Process objects are objects that ingest data (sources), operate on data to generate new data (filters), and interface the data to graphics systems and/or other systems (mappers). (Process objects are also referred to generically as filters.) You may want to review "The Visualization Model" on page 23 for information about the graphics pipeline.

12.1 Overview

If you have an algorithm or process that you wish to implement as a VTK filter, there are several issues to consider.

1. Is the object to be a source, filter, or mapper? Does it ingest data and/or generate data?
2. What type (i.e., dataset type) does it accept as input? What type of dataset does it generate (if any)? This information will help you decide what superclass to derive your filter from.
3. How many inputs and outputs does it have? Multiple inputs and/or outputs often require special treatment since convenience superclasses often do not exist. For example, you may have to write methods similar to SetInput() or GetOutput() to accept extra inputs or retrieve the extra outputs.
4. Is the process object to be written for generality or for performance? High-performing filters are often much faster but are usually limited in the type of data they can accept. General filters treat a variety of data and are therefore of wider applicability.
5. Is there a similar object in the system that can serve as a superclass or be expanded to accommodate the new functionality? It's best to try and consolidate similar functionality, if appropriate.

There are several other basic concepts that we address in the following sections.

Never Modify Input Data

One of the most important guidelines for any filter writer is to never modify the input to the filter. The reason for this is simple: the proper execution of the pipeline requires that filters create and modify their own output. Remember, other filters may be using the input data as well; if you modify a filters input, you can be potentially corrupt the data with respect to other filters that use it, or the filter that created it.

Reference Count Data

If the input data to a filter is sent to the output of the filter unchanged, make sure that you use reference counting. Do not copy data, the memory cost can be quite high for large visualization data. Typically, if you use Get__() and Set__() methods to get and set data, reference counting is taken care of for you. You also may wish to use specialized methods for passing data through a filter, see "Interface To Field and Attribute Data" on page 270 for more information.

Use Debug Macros

Filters should provide debugging information when the object's Debug flag is set. This is conveniently done using VTK's debug macros defined in `VTK/Common/vtkSetGet.h`. At a minimum, a filter should report the initiation of execution similar to the following (from `VTK/Graphics/vtkContourFilter.cxx`):

```
vtkDebugMacro(<< "Executing contour filter");
```

You may also wish to provide other information as the filter executes, for example, a summary of execution (again from `vtkContourFilter.cxx`):

```
vtkDebugMacro(<<"Created: "
              << newPts->GetNumberOfPoints() << " points, "
              << newVerts->GetNumberOfCells() << " verts, "
              << newLines->GetNumberOfCells() << " lines, "
              << newPolys->GetNumberOfCells() << " triangles");
```

Do not place debugging macros in the inner portions of a loop, since the macro invokes an `if` check that may affect performance. Also, if debugging is turned on in an inner loop, too much information will be output to be meaningfully interpreted.

Reclaim/Delete Allocated Memory

One common mistake that filter writers make is introducing memory leaks or using excessive memory. Memory leaks can be avoided by pairing New() and Delete() methods for all VTK objects, and `new` and `delete` methods for all native or non-VTK objects. (See "Standard Methods: Creating and Deleting Objects" on page 205.)

Another way to reduce memory usage is to use the Squeeze() methods that all data objects support. This method reclaims excess memory that an object may be using. Use the Squeeze() method whenever the size of the data object can only be estimated.

Modified Time

One of the trickiest parts of writing a filter is making sure that its modified time is properly managed. As you may recall, modified time is an internal time stamp that each object maintains in response to changes in its internal state. Typically, modified time changes when a Set__() method is invoked. For example, the method vtkTubeFilter::SetNumberOfSides(num) causes the vtkTubeFilter's modified time to change when this method is invoked, as long as `num` is different than the tube filter's original instance variable value.

Normally, the modified time of a filter is maintained without requiring intervention. For example, if you use the vtkSet/Get macros defined in `VTK/Common/vtkSetGet.h` to get and set instance variable values, modified time is properly managed, and the inherited method vtkObject::GetMTime() returns the correct value. However, if you define your own Set__() methods, or include methods that modify the internal state of the object, you will have to invoke Modified() (i.e., change the internal modified time) on the filter as appropriate. And, if your object definition includes references to other objects, the correct modified time of the filter includes both the filter and objects it depends on. This requires overloading the GetMTime() method.

vtkCutter is an example of a filter demonstrating this behavior. This filter makes reference to another object, (i.e., the CutFunction), an instance of vtkImplicitFunction. When the implicit function definition changes, we expect the cutter to reexecute. Thus, the GetMTime() method of the filter must reflect this by considering the modified time of the implicit function, and the GetMTime() method inherited from its superclass vtkObject must be overloaded. The implementation of vtkCutter::GetMTime() method is as follows. It's actually more complicated than suggested here because vtkCutter depends on two other objects: vtkLocator and vtkContourValues as shown.

```
unsigned long vtkCutter::GetMTime()
{
  unsigned long mTime=this->vtkDataSetFilter::GetMTime();
  unsigned long
      contourValuesMTime=this->ContourValues->GetMTime();
  unsigned long time;

  mTime = ( contourValuesMTime > mTime ?
                                contourValuesMTime : mTime );
  if ( this->CutFunction != NULL )
    {
    time = this->CutFunction->GetMTime();
    mTime = ( time > mTime ? time : mTime );
    }
  if ( this->Locator != NULL )
    {
    time = this->Locator->GetMTime();
    mTime = ( time > mTime ? time : mTime );
    }
  return mTime;
}
```

The class vtkLocator is used to merge coincident points as the filter executes; while vtkContourValues is a helper class used to generate contour values for those functions using isosurfacing as part of their execution process. The GetMTime() method must check all objects that vtkCutter depends on, returning the largest modified time it finds.

Although the use of modified time and the implicit execution model of the VTK visualization pipeline is simple, there is one common source of confusion. That is, you must distinguish between a filter's modified time and its dependency on the data stream. Remember, filters will execute either when they are modified (the modified time changes), or the input to the filter changes (dependency on data stream). Modified time only reflects changes to a filter or dependencies on other objects independent of the data stream.

ProgressEvent and AbortExecute

The ProgressEvent is invoked at regular intervals during the execution of a source, filter, or mapper object (i.e., any process object). Progress user methods typically perform such functions as updating an application user interface (e.g., imagine manipulating a progress bar in the GUI). (See "General Guidelines for GUI Interaction" on page 305 and "User Methods, Observers, and Commands" on page 28 for more information.)

A progress user method is created by invoking AddObserver() with an event type of vtkCommand::ProgressEvent. When the progress method is invoked, the filter also sets the current progress value (a decimal fraction between (0,1) retrieved with the GetProgress() method). The progress method can be used in combination with the vtkProcessObject's invocation of StartEvent and EndEvent. For example, `Examples/Tutorial/Step2` shows how to use these methods. Here's a code fragment to show how it works.

```
vtkDecimatePro deci
   deci AddObserver StartEvent {StartProgress deci "Decimating..."}
   deci AddObserver ProgressEvent {ShowProgress deci "Decimating..."}
   deci AddObserver EndEvent EndProgress
```

Not all filters invoke ProgressEvents or invoke them infrequently (depending on the filter implementation). Also, the progress value may not be a true measure of the actual work done by the filter. It is difficult to measure progress in some filters and/or the filter implementor will often choose to update the filter at key points in the algorithm (e.g., after building internal data structures, reading in a piece of data, etc.).

Related to progress methods is the concept of a flag that is used to stop execution of a filter. This flag, defined in vtkProcessObject, is called the AbortExecute flag. When set, some filters will terminate their execution, sending to their output some portion of the result of their execution (maybe even nothing). Typically, the flag is set during invocation of a ProgressEvent and is used to prematurely terminate execution of a filter when it is taking too long or the application is attempting to keep up with user input events. Not all filters support the AbortExecute flag. Check the source code to be sure which ones support the flag. (Most do, and those that don't will in the near future.)

An example use of progress user methods and the abort flag is shown in the following code fragment taken from `VTK/Graphics/vtkDecimatePro.cxx`:

```
for ( ptId=0; ptId < npts && !abort ; ptId++ )
  {
  if ( ! (ptId % 10000) )
    {
    vtkDebugMacro(<<"Inserting vertex #" << ptId);
    this->UpdateProgress (0.25*ptId/npts);//25% inserting
    if (this->GetAbortExecute())
      {
      abort = 1;
      break;
      }
    }
```

```
    this->Insert(ptId);
    }
```

Note that the filter implementor made some arbitrary decisions: the progress methods are invoked every 10,000 points; and the portion of the Execute() method shown is assumed to take approximately 25% of the total execution time. Also, some debug output is combined with the execution of the progress method, as well as a status check on the abort flag. This points out an important guideline: as a filter implementor you do not want to invoke progress methods too often because they can affect overall performance.

12.2 How To Write A Graphics Filter

In "Interfacing To VTK Data Objects" on page 235 we dealt at length with the methods used to create, manipulate, and access VTK datasets and other data objects. With this background, the actual procedure to write a filter is relatively straightforward. The idea is simple: we access information from the input dataset, configure the output dataset, and implement in the body of the filter a method that generates data and sends it to the output. However, many complications can creep in due to multiple inputs and/or outputs, instance variables referenced by the filter, unusual combinations of input and output data, and the desire to stream data.

The instructions written here assume that you'll be implementing the filter in C++. However, you may want to investigate writing a programmable filter, which lets you implement filters in other languages and at run-time (i.e., no compiling, linking, or rebuilding libraries required). See "Programmable Filters" on page 292 for more information.

Overview

Besides the general considerations outlined in the "Overview" on page 275, you must make several design decisions when writing a filter in the graphics pipeline.

1. which superclass to derive from,
2. how to properly manage modified time including overloading the GetMTime() method and invoking Modified(), as appropriate,
3. whether to overload methods defined by the pipeline execution process,
4. creation of an Execute() method,

5. whether to support Progress() and Abort() methods,
6. and any instance variables, structures, or methods required by the filter implementation.

The proper derivation from a superclass is a function of whether the filter is a source, filter, or mapper object; how many inputs and outputs the class has; and the type(s) of input and output data (see "The Visualization Model" on page 23 for more information about sources, filters, and mappers). Many objects require only a single input and output, these can derive much of their implementation from the convenience superclasses provided by VTK, and do not need to overload the pipeline execution or GetMTime() methods. These classes we call simple filters and are described in "Simple Filters" on page 281. Filters that are not simple filters are termed complex filters and are described in "Complex Filters and Pipeline Execution" on page 286.

Simple Filters

Simple filters are filters that manifest typical behavior. This means that they have one input and/or output, support common combinations of input and output data types, and do not require overloading the GetMTime() method. If you are writing a simple filter, generally all you'll need to do is follow the coding conventions outlined in Chapter 9 "Contributing Code" on page 201, create an Execute() method and any other supporting methods or functions internal to the filter. The most difficult task you'll face (for simple filters) is finding the correct superclass to inherit from. Of course, the algorithm may be difficult to implement as well.

Here are some candidate abstract superclasses to inherit from. These classes provide control of input and output, as well as the appropriate Update() method. Typically the choice of superclass is made from the type of input/output data, and the type of process object (source, filter, mapper).

If you are writing a source object, consider deriving from these classes. The abstract superclasses are used by both reader and procedural source objects.

- vtkPolyDataSource — read data from a file or procedurally generate vtkPolyData
- vtkStructuredPointsSource — read data from a file and or procedurally generate vtkStructuredPoints

- vtkRectilinearGridSource — read data from a file and or procedurally generate vtkRectilinearGrid
- vtkStructuredGridSource — read data from a file and or procedurally generate vtk-StructuredGrid
- vtkUnstructuredGridSource — read data from a file and or procedurally generate vtkUnstructuredGrid

If you are writing a filter object, consider deriving from the classes listed in the following. The names of these abstract classes clearly identify the input and output data types of the filter.

- vtkDataSetToPolyDataFilter
- vtkDataSetToStructuredGridFilter
- vtkDataSetToStructuredPointsFilter
- vtkDataSetToUnstructuredGridFilter
- vtkPolyDataToPolyDataFilter
- vtkRectilinearGridToPolyDataFilter
- vtkStructuredGridToPolyDataFilter
- vtkStructuredGridToStructuredGridFilter
- vtkStructuredPointsToPolyDataFilter
- vtkStructuredPointsToUnstructuredGridFilter

If you are writing a mapper object, consider deriving from the classes listed in the following. Mappers generally are not simple process objects, with the exceptions of writers. And even these require writing a WriteData() method as well as a SetInput() method. You'll also have to provide instance variables to handle output file names, and methods to manipulate them.

- vtkWriter — specify abstract interface for writer objects

Once you've chosen the correct superclass, you'll want to implement the Execute() method. The execute method should access the appropriate input data, perform error checking if necessary, implement an algorithm (i.e., generate data objects), and send the data objects to the output.

Here's a full example from vtkShrinkFilter. vtkShrinkFilter shrinks cells towards their geometric center. It takes generic vtkDataSet for input and generate vtkUnstructuredGrid as output. Thus we can subclass from the abstract superclass vtkDataSetToUnstructuredGridFilter as shown in the following (from `VTK/Graphics/vtkShrinkFilter.h`):

```
#ifndef __vtkShrinkFilter_h
#define __vtkShrinkFilter_h

#include "vtkDataSetToUnstructuredGridFilter.h"

class VTK_GRAPHICS_EXPORT vtkShrinkFilter : public
                          vtkDataSetToUnstructuredGridFilter
{
public:
  static vtkShrinkFilter *New();
  vtkTypeMacro(vtkShrinkFilter,
                          vtkDataSetToUnstructuredGridFilter);
  void PrintSelf(ostream& os, vtkIndent indent);
  // Description:
  // Set the fraction of shrink for each cell.
  vtkSetClampMacro(ShrinkFactor,float,0.0,1.0);
  // Description:
  // Get the fraction of shrink for each cell.
  vtkGetMacro(ShrinkFactor,float);
protected:
  vtkShrinkFilter();
  ~vtkShrinkFilter() {};

  void Execute();
  float ShrinkFactor;

private:
  vtkShrinkFilter(const vtkShrinkFilter&);
  void operator=(const vtkShrinkFilter&);
};
#endif
```

To complete the definition of the class, we need only to implement the PrintSelf() and Execute() methods. The Execute() method (excerpted from the `VTK/Graphics/vtkShrinkFilter.cxx` file) is shown in the following. (Note: `VTK_GRAPHICS_EXPORT` is a `#define` macro that is used by some compilers to export symbols.)

```
void vtkShrinkFilter::Execute()
```

```
{
vtkPoints *newPts;
  int i, j, cellId, numCells, numPts;
  int oldId, newId, numIds;
  float center[3], *p, pt[3];
  vtkPointData *pd, *outPD;;
  vtkIdList *ptIds, *newPtIds;
  vtkDataSet *input= this->GetInput();
  vtkUnstructuredGrid *output = this->GetOutput();
```

We continue by accessing the input data and performing error checking. Note the use of the debug macro:

```
  vtkDebugMacro(<<"Shrinking cells");
  numCells=input->GetNumberOfCells();
  numPts = input->GetNumberOfPoints();
  if (numCells < 1 || numPts < 1)
    {
    vtkErrorMacro(<<"No data to shrink!");
    return;
    }
```

The input and output to the filter is guaranteed to be non-NULL in the Execute() method. Next, we instantiate and allocate some data objects.

```
  ptIds = vtkIdList::New();
  ptIds->Allocate(VTK_CELL_SIZE);
  newPtIds = vtkIdList::New();
  newPtIds->Allocate(VTK_CELL_SIZE);

  output->Allocate(numCells);
  newPts = vtkPoints::New();
  newPts->Allocate(numPts*8,numPts);
  pd = input->GetPointData();
  outPD = output->GetPointData();
  outPD->CopyAllocate(pd,numPts*8,numPts);
```

We've estimated the size of the allocated object (i.e., `numPts*8`) because we don't know the exact size of the resulting output data (or at least we don't want to do the work necessary to find the exact size). Next, we implement the algorithm proper (shrink cells towards their center). If any of these data object methods are unfamiliar to you, see "Interfacing To VTK Data Objects" on page 235.

```
  for (cellId=0; cellId < numCells; cellId++)
```

```
{
input->GetCellPoints(cellId,ptIds);
numIds = ptIds->GetNumberOfIds();
// get the center of the cell
center[0] = center[1] = center[2] = 0.0;
for (i=0; i < numIds; i++)
  {
  p = input->GetPoint(ptIds->GetId(i));
  for (j=0; j < 3; j++) center[j] += p[j];
  }
for (j=0; j<3; j++) center[j] /= numIds;
// Create new points and cells
newPtIds->Reset();
for (i=0; i < numIds; i++)
  {
  p = input->GetPoint(ptIds->GetId(i));
  for (j=0; j < 3; j++)
    pt[j] = center[j] + this->ShrinkFactor*(p[j] -
                                            center[j]);
  oldId = ptIds->GetId(i);
  newId = newPts->InsertNextPoint(pt);
  newPtIds->InsertId(i,newId);
  outPD->CopyData(pd, oldId, newId);
  }
output->InsertNextCell(input->GetCellType(cellId),
                       newPtIds);
}
```

Finally, we pass through any cell attribute data (because the cells didn't change, only their defining points), set the point coordinates data, and use the Squeeze() method to reclaim memory. Also, we use the Delete() method on the points. (In this case the points don't actually get deleted because their reference count is non-zero. The output data references them.)

```
  output->GetCellData()->PassData(input->GetCellData());
  output->SetPoints(newPts);
  output->Squeeze();
  newPts->Delete();
  ptIds->Delete();
  newPtIds->Delete();
  newPts->Delete();
}
```

Complex Filters and Pipeline Execution

Complex filters are non-simple filters that cannot derive their implementation from one of the convenience classes listed in "Simple Filters" on page 281. Typically these classes have more than one input and/or output, have unusual combinations of input and output data, and/or need to overload methods defined in the pipeline execution process. Creating a complex filter means that you'll have to create methods to set the input, define internal data objects (and methods to access them) to represent the output, and/or you'll have to overload one or more methods defined in "The Execution Process" on page 219.

The best place to begin this discussion is to look at what a simple filter does. A simple filter like vtkShrinkPolyData derives from the superclass vtkPolyDataToPolyDataFilter. This class in turn derives from the superclasses vtkPolyDataSource, vtkSource, and vtkProcessObject. These classes perform the following functions.

- vtkProcessObject are expected to invoke the StartEvent, ProgressEvent, and EndEvent events; so it defines and interfaces with the AbortExecute and Progress instance variables. It also defines the Inputs instance variable, which is a list of inputs to the filter. There are also special methods for setting and managing the inputs.
- vtkSource defines the API for and implements the pipeline execution process; and interfaces with the vtkDataObject ReleaseData flags. It also creates the protected instance variable Outputs, which is a list of outputs of the filter. The execution time of the filter is tracked with the InformationTime instance variable.
- vtkPolyDataSource defines the GetOutput() method as a method that returns a pointer to a vtkPolyData.
- vtkPolyDataToPolyDataFilter defines the SetInput() method to accept a pointer to vtkPolyData.
- And finally, vtkShrinkPolyData overloads the Execute() method to actually do something useful.

Hopefully it's a little clearer why filters with multiple inputs and/or outputs, or with unusual combinations of input and output types, require special treatment. As you can see, there are several assumptions about the number of inputs and the combination of input and output. If you do wish to write a complex filter, what you need to do is provide the functionality contained in this combination of objects. The best way to see how it's done is to look at an example. We'll look at the class vtkExtractVectorComponents.

The functionality of the class vtkExtractVectorComponents is to extract the *x*-*y*-*z* vector components of the input vector field into three separate outputs, one for each component. The structure of each output is the same, but the scalar data is changed. The class header file looks like the following (`VTK/Graphics/vtkExtractVectorComponents.h`).

```
#ifndef __vtkExtractVectorComponents_h
#define __vtkExtractVectorComponents_h
#include "vtkSource.h"
#include "vtkDataSet.h"

class VTK_GRAPHICS_EXPORT vtkExtractVectorComponents :
                    public vtkSource
{
public:
  static vtkExtractVectorComponents *New();
  vtkTypeMacro(vtkExtractVectorComponents,vtkSource);
  void PrintSelf(ostream& os, vtkIndent indent);

  virtual void SetInput(vtkDataSet *input);
  vtkDataSet *GetInput();

  vtkDataSet *GetVxComponent();
  vtkDataSet *GetVyComponent();
  vtkDataSet *GetVzComponent();
  vtkDataSet *GetOutput(int i=0);

  vtkSetMacro(ExtractToFieldData, int);
  vtkGetMacro(ExtractToFieldData, int);
  vtkBooleanMacro(ExtractToFieldData, int);

protected:
  vtkExtractVectorComponents();
  ~vtkExtractVectorComponents();

  void Execute();
  int ExtractToFieldData;

private:
  vtkExtractVectorComponents(const vtkExtractVectorComponents&);
  void operator=(const vtkExtractVectorComponents&);

};
#endif
```

This example demonstrates several important concepts. First, notice that we're inheriting from vtkSource, since no convenience superclass is available. And second, we have to define SetInput() and several methods to get the output of the filter (i.e., GetVxComponent(), GetVyComponent(), GetVzComponent(), and GetOutput()).

The SetInput() method is most instructive:

```
void vtkExtractVectorComponents::SetInput(vtkDataSet *input)
{
  if (this->NumberOfInputs > 0 && this->Inputs[0] == input )
    {
    return;
    }
  this->vtkProcessObject::SetNthInput(0, input);
  if ( input == NULL )
    {
    return;
    }
  if (this->NumberOfOutputs < 3)
    {
    this->SetNthOutput(0,input->MakeObject());
    this->Outputs[0]->Delete();
    this->SetNthOutput(1,input->MakeObject());
    this->Outputs[1]->Delete();
    this->SetNthOutput(2,input->MakeObject());
    this->Outputs[2]->Delete();
    return;
    }
  // since the input has changed we might need to create a new output
  if (strcmp(this->Outputs[0]->GetClassName(),
                                        input->GetClassName()))
    {
    this->SetNthOutput(0,input->MakeObject());
    this->Outputs[0]->Delete();
    this->SetNthOutput(1,input->MakeObject());
    this->Outputs[1]->Delete();
    this->SetNthOutput(2,input->MakeObject());
    this->Outputs[2]->Delete();
   vtkWarningMacro(<<" a new output had to be created since the input type
changed.");
    }
}
```

As you can see, the filter takes advantage of superclass methods to set the inputs and outputs. One of the unusual behaviors of this filter is that setting the input causes the outputs (there are three) to be generated. The MakeObject() method is used, which is called a virtual constructor because it creates an object as the same type as the invoking object.

Abstract Graphics Filters

There are a number of VTK filters that generate abstract dataset output types such as vtkDataSet or vtkPointSet. Since abstract dataset types cannot be instantiated (due to the presence of pure virtual functions), you may be wondering how this is possible, or why such filters are even useful. The answers are that 1) we make a reference-counted copy of the input dataset's structure and place it in the output, and 2) many filters manipulate attribute data (e.g., scalars or vectors) and do not modify the dataset structure. Thus, filters that can pass the structure through (independent of input type) and modify or generate attribute data are valuable additions to the toolkit.

There are two superclasses to use if you want to create abstract graphics filters. These are

- vtkDataSetToDataSetFilter — input any dataset type and output the same dataset type, and
- vtkPointSetToPointSetFilter — input any dataset of type vtkPointSet (i.e., datasets that explicitly represent their points) and output a vtkPointSet.

Writing filters of this type are quite simple as shown in the following example. Here we'll show how vtkElevationFilter is implemented, a subclass of vtkDataSetToDataSetFilter. The header `.h` file is as follows (and is found in `Graphics/vtkElevationFilter.h`).

```
class VTK_GRAPHICS_EXPORT vtkElevationFilter : public
                                    vtkDataSetToDataSetFilter
{
public:
  static vtkElevationFilter *New();
  vtkTypeMacro(vtkElevationFilter,vtkDataSetToDataSetFilter);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Description:
  // Define one end of the line (small scalar values).
  vtkSetVector3Macro(LowPoint,float);
  vtkGetVectorMacro(LowPoint,float,3);
```

```
  // Description:
  // Define other end of the line (large scalar values).
  vtkSetVector3Macro(HighPoint,float);
  vtkGetVectorMacro(HighPoint,float,3);

  // Description:
  // Specify range to map scalars into.
  vtkSetVector2Macro(ScalarRange,float);
  vtkGetVectorMacro(ScalarRange,float,2);

protected:
  vtkElevationFilter();
  ~vtkElevationFilter() {};

  void Execute();
  float LowPoint[3];
  float HighPoint[3];
  float ScalarRange[2];

private:
  vtkElevationFilter(const vtkElevationFilter&); //not implemented
  void operator=(const vtkElevationFilter&); //not implemented
};
```

Note that we only have to implement the usual class methods, as well as the Execute() method. The body of the Execute() method appears as follows.

```
void vtkElevationFilter::Execute()
{
int i, j, numPts;
vtkFloatArray *newScalars;
float l, *x, s, v[3];
float diffVector[3], diffScalar;
vtkDataSet *input = this->GetInput();

// Initialize
vtkDebugMacro(<<"Generating elevation scalars!");

// First, copy the input to the output as a starting point
this->GetOutput()->CopyStructure( input );
if ( ((numPts=input->GetNumberOfPoints()) < 1) )
  {
  vtkErrorMacro(<< "No input!");
  return;
  }
```

```
// Allocate
newScalars = vtkFloatArray::New();
newScalars->SetNumberOfTuples(numPts);

// Set up 1D parametric system
bounds = input->GetBounds();
for (i=0; i<3; i++)
    {diffVector[i] = this->HighPoint[i] - this->LowPoint[i];}
if ( (l = vtkMath::Dot(diffVector,diffVector)) == 0.0)
  {
  vtkErrorMacro(<< this << ": Bad vector, using (0,0,1)\n");
  diffVector[0] = diffVector[1] = 0.0; diffVector[2] = 1.0;
  l = 1.0;
  }

// Compute parametric coordinate and map into scalar range
diffScalar = this->ScalarRange[1] - this->ScalarRange[0];
for (i=0; i<numPts; i++)
  {
  if ( ! (i % 10000) )
    {
    this->UpdateProgress ((float)i/numPts);
    if (this->GetAbortExecute())
      {
      break;
      }
    }
  x = input->GetPoint(i);
  for (j=0; j<3; j++)
    {
    v[j] = x[j] - this->LowPoint[j];
    }
  s = vtkMath::Dot(v,diffVector) / l;
  s = (s < 0.0 ? 0.0 : s > 1.0 ? 1.0 : s);
  newScalars->SetValue(i,this->ScalarRange[0]+s*diffScalar);
  }

// Update output
  this->GetOutput()->GetPointData()->CopyScalarsOff();
  this->GetOutput()->GetPointData()->PassData(input->GetPointData());
  this->GetOutput()->GetCellData()->PassData(input->GetCellData());
  newScalars->SetName("Elevation");
  this->GetOutput()->GetPointData()->SetScalars(newScalars);
  newScalars->Delete();
}
```

Note that the filter only computes scalar data and then passes it to the output. The actual generation of the output structure is done using the CopyStructure() method. This method makes a reference-counted copy of the input data topological/geometric structure.

vtkPointSetToPointSetFilter works in a similar fashion, except that the point coordinates are modified or generated and sent to the output. See vtkTransformFilter if you'd like to see a concrete example of vtkPointSetToPointSetFilter.

If you desire to write a filter that modifies attribute data, or modifies point positions without changing the number of points or cells, these abstract filters should be used.

Programmable Filters

An alternative to developing a filter in C++ is to use programmable process objects. These objects allow you to create a function that is invoked during the execution of the process object (i.e., during the Execute() method). The advantage of programmable filters is that you do not have to rebuild the VTK libraries, or even use C++. In fact, you can use the supported interpreted languages Tcl, Java, and Python to create a filter!

Programmable sources and filters are process objects that enable you to create new filters at run-time. There is no need to create a C++ class or rebuild object libraries. Programmable objects take care of the overhead of hooking into the visualization pipeline (i.e., managing modified time checking, properly invoking the Update() and Execute() methods, and providing methods for setting input and/or getting filter output), requiring only that you write the body of the filter's Execute() method.

The new objects are vtkProgrammableSource, vtkProgrammableFilter, and vtkProgrammableAttributeDataFilter. vtkProgrammableSource is a source object that supports and can generate an output of any of the VTK dataset types. vtkProgrammableFilter allows you to set input and retrieve output of any dataset type (e.g., GetPolyDataOutput()). The filter vtkProgrammableAttributeDataFilter allows one or more inputs of the same or different types, and can generate an output of any dataset type.

An example will clarify the application of these filters. This excerpted code is from `VTK/Examples/Modelling/Tcl/expCos.tcl`.

```
vtkProgrammableFilter besselF
   besselF SetInput [transF GetOutput]
   besselF SetExecuteMethod bessel
```

```
proc bessel {} {
    set input [besselF GetPolyDataInput]
    set numPts [$input GetNumberOfPoints]
     vtkPoints newPts
     vtkFloatArray derivs
     for {set i 0} {$i < $numPts} {incr i} {
         set x [$input GetPoint $i]
         set x0 [lindex $x 0]
         set x1 [lindex $x 1]
         set r [expr sqrt($x0*$x0 + $x1*$x1)]
         set x2 [expr exp(-$r) * cos(10.0*$r)]
         set deriv [expr -exp(-$r) * (cos(10.0*$r) + 10.0*sin(10.0*$r))]
         newPts InsertPoint $i $x0 $x1 $x2
         eval derivs InsertValue $i $deriv
     }
     [besselF GetPolyDataOutput] CopyStructure $input
     [besselF GetPolyDataOutput] SetPoints newPts
     [[besselF GetPolyDataOutput] GetPointData] SetScalars derivs
    newPts Delete; #reference counting - it's ok
    derivs Delete
}

# warp plane
vtkWarpScalar warp
    warp SetInput [besselF GetPolyDataOutput]
    warp XYPlaneOn
    warp SetScaleFactor 0.5
```

This example instantiates a vtkProgrammableFilter and then the Tcl `proc bessel()` serves as the function to compute the Bessel functions and derivatives. Note that `bessel()` works directly with the output of the filter obtained with the method GetPolyDataOutput(). This is because the output of `besselF` can be of any VTK supported dataset type, and we have to indicate to objects working with the output which type to use.

Overloading Pipeline Execution Methods

Probably the most difficult task you can undertake when writing a new filter is to overload one or more of the pipeline execution methods described in "The Execution Process" on page 219. The difficulty of this task lies in understanding how the pipeline execution process works. Do this only if you need your filter to stream.

There are relatively few methods that you need to consider overloading. These methods are described here, with a general description of what you must do and how they affect the execution process.

ExecuteInformation(). This method gives the subclass a chance to configure the output. For example, the WholeExtent, spacing, origin, scalar type, and number of components can be modified. By default, vtkSource simply copies the basic extent information from the first input to all its outputs. If this is not the correct behavior for your filter, then you must overload this method.

EnlargeOutputUpdateExtents(). This method gives the subclass a chance to indicate that it will provide more data than required for the output. This can happen, for example, when the filter can only produce the whole output. Although this is being called for a specific output which is passed in as a parameter, the source may need to enlarge all outputs. The default implementation is to do nothing (do not enlarge the output). If your filter requires the output update extent to be larger than requested (for example, your filter can only produce the whole output), then you must overload this method.

ComputeInputUpdateExtents(). This method gives the subclass a chance to request a larger extent on the inputs. This is necessary when, for example, a filter requires more data at the "internal" boundaries to produce the boundary values—such as an image filter that derives a new pixel value by applying some operation to a kernel of surrounding input values. The default implementation is to require the entire input in order to generate the output. This is overridden, for example, in vtkImageToImageFilter to request the same input update extent as the output update extent. It is redefined again in vtkImageSpatialFilter in order to request a slightly larger input update extent due to the additional pixels required according to the kernel size of the filter. You should carefully look up the hierarchy for your filter too see if the default implementation is the correct one.

Execute(). The Execute() method almost always has to be overloaded. The major exception to this rule are imaging filters (see "How To Write An Imaging Filter" on page 295). Most imaging filters are derived from vtkImageToImageFilter that implements an Execute() method that in turn invokes ThreadedExecute(). ThreadedExecute() is a special version of Execute() that supports multithreading.

12.3 How To Write An Imaging Filter

The graphics pipeline and the imaging pipeline share a unified execution process with the release of version VTK 3.1 and later. If you understand the pipeline execution process described previously, you understand it for imaging filters as well. However, there remains several important difference between imaging filters and other types of filters.

1. Imaging algorithms are typically local in nature, and the input and output type is the same.
2. Most imaging filters only process scalar data attributes (of one to four components). Other information, such as vectors, normals, etc., is not processed.
3. Imaging filters use templated functions to implement their Execute() method.
4. Almost all imaging filters have been written to stream data.
5. Almost all imaging filters are multi-threaded.

These difference gives rise to code different than we have seen previously. The code is typically more complex (e.g., multithreaded and streaming) but better performing (localized algorithms and templated functions).

Implementing An Imaging Filter

To demonstrate creation of an imaging filter, let's consider an image filter that performs a simple scale and shift operation on each input pixel. The filter—vtkImageShiftScale—is a subclass of vtkImageToImageFilter, as are most imaging filters.

```
#include "vtkImageToImageFilter.h"
class VTK_IMAGING_EXPORT vtkImageShiftScale : public vtkImageToImageFilter
{
public:
  static vtkImageShiftScale *New();
  vtkTypeMacro(vtkImageShiftScale,vtkImageToImageFilter);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Description:
  // Set/Get the shift value.
  vtkSetMacro(Shift,float);
  vtkGetMacro(Shift,float);

  // Description:
```

```
  // Set/Get the scale value.
  vtkSetMacro(Scale,float);
  vtkGetMacro(Scale,float);

  // Description:
  // Set the desired output scalar type. The result of the shift
  // and scale operations is cast to the type specified.
  vtkSetMacro(OutputScalarType, int);
  vtkGetMacro(OutputScalarType, int);
  void SetOutputScalarTypeToDouble()
    {this->SetOutputScalarType(VTK_DOUBLE);}
  void SetOutputScalarTypeToFloat()
    {this->SetOutputScalarType(VTK_FLOAT);}
  void SetOutputScalarTypeToLong()
    {this->SetOutputScalarType(VTK_LONG);}
  void SetOutputScalarTypeToUnsignedLong()
    {this->SetOutputScalarType(VTK_UNSIGNED_LONG);};
  void SetOutputScalarTypeToInt()
    {this->SetOutputScalarType(VTK_INT);}
  void SetOutputScalarTypeToUnsignedInt()
    {this->SetOutputScalarType(VTK_UNSIGNED_INT);}
  void SetOutputScalarTypeToShort()
    {this->SetOutputScalarType(VTK_SHORT);}
  void SetOutputScalarTypeToUnsignedShort()
    {this->SetOutputScalarType(VTK_UNSIGNED_SHORT);}
  void SetOutputScalarTypeToChar()
    {this->SetOutputScalarType(VTK_CHAR);}
  void SetOutputScalarTypeToUnsignedChar()
    {this->SetOutputScalarType(VTK_UNSIGNED_CHAR);}

    // Description:
    // When the ClampOverflow flag is on, the data is thresholded so
    // that the output value does not exceed the max or min of the
    // data type. By defualt, ClampOverflow is off.
    vtkSetMacro(ClampOverflow, int);
    vtkGetMacro(ClampOverflow, int);
    vtkBooleanMacro(ClampOverflow, int);

protected:
    vtkImageShiftScale();
    ~vtkImageShiftScale() {};

    float Shift;
    float Scale;
    int OutputScalarType;
    int ClampOverflow;
```

```
void ExecuteInformation(vtkImageData *inData,
                        vtkImageData *outData);
void ExecuteInformation(){
     this->vtkImageToImageFilter::ExecuteInformation();};
void ThreadedExecute(vtkImageData *inData, vtkImageData *outData,
     int extent[6], int id);

private:
  vtkImageShiftScale(const vtkImageShiftScale&);
  void operator=(const vtkImageShiftScale&);
};
```

As you can see from the header file, this class is implemented similar to other VTK filter classes. The major differences are the extra protected methods ExecuteInformation() and ThreadedExecute(). These methods are required to support streaming and multithreading, as we'll see shortly.

Inside the C++ file `VTK/Imaging/vtkImageShiftScale.cxx`, there are some other important differences from what we've seen before. First, the ThreadedExecute() method:

```
void vtkImageShiftScale::ThreadedExecute(vtkImageData *inData,
 vtkImageData *outData,
 int outExt[6], int id)
{
  void *inPtr = inData->GetScalarPointerForExtent(outExt);

  switch (inData->GetScalarType())
    {
    vtkTemplateMacro6(vtkImageShiftScaleExecute1, this,
                      inData, (VTK_TT*) (inPtr), outData, outExt, id);
    default:
      vtkErrorMacro(<< "Execute: Unknown ScalarType");
      return;
    }
}
```

The ThreadedExecute() method is actually invoked by the superclass vtkImageToImage-Filter in its Execute() method. ThreadedExecute() is meant to process the filter in multiple threads (i.e., a shared memory implementation).

An important feature of this code is the `switch` statement. This code switches on the input data type and invokes a templated function. This templated function,

vtkImageShiftScaleExecute1(), in turn invokes another templated function vtkImage-ShiftScaleExecute() (switched on output data type). The double template is necessary because the input and output types may be different.

```
template <class T>
static void vtkImageShiftScaleExecute1(vtkImageShiftScale *self,
      vtkImageData *inData, T *inPtr,
      vtkImageData *outData,
      int outExt[6], int id)
{
  void *outPtr = outData->GetScalarPointerForExtent(outExt);

  switch (outData->GetScalarType())
    {
vtkTemplateMacro7(vtkImageShiftScaleExecute, self, inData, inPtr,
       outData, (VTK_TT*)(outPtr),outExt, id);
      default:
      vtkGenericWarningMacro("Execute: Unknown input ScalarType");
      return;
    }
}
```

The templated functions used here are not declared in the header file since it will only be used by this one class. It is not a class method either, just a simple C++ templated function. It should always appear in the source code before it is used. So typically the templated execute function comes right before the Execute() (or ThreadedExecute()) method discussed previously. The template keyword before the function definition indicates that it is a templated function, and that it is templated over the type `T`. You can think of `T` as a string that gets replaced by `int`, `short`, `float`, etc. at compile time. Notice that `T` is used as the data type for `inPtr` and `outPtr`. Since this is a function instead of a method, it doesn't have access to the `this` pointer and its associated instance variables. So we pass in the `this` pointer as a variable called `self`. `Self` can then be used to access the values of the class as is done on the first line of the function. Here we obtain the value of `Scale` from the instance and place it into a local variable called scale. The remaining lines declare some local variables that will be used to move through the data.

Finally, the actual work of the algorithm is performed in the vtkImageShiftScaleExecute() method: Note the use of AbortExecute instance variable and the UpdateProgress() method as we saw in previous implementations of filters.

```
template <class IT, class OT>
static void vtkImageShiftScaleExecute(vtkImageShiftScale *self,
```

```
        vtkImageData *inData, IT *inPtr,
        vtkImageData *outData, OT *outPtr,
        int outExt[6], int id)
{
  float typeMin, typeMax, val;
  int clamp;
  float shift = self->GetShift();
  float scale = self->GetScale();
  int idxR, idxY, idxZ;
  int maxY, maxZ;
  int inIncX, inIncY, inIncZ;
  int outIncX, outIncY, outIncZ;
  int rowLength;
  unsigned long count = 0;
  unsigned long target;
  // for preventing overflow
  typeMin = outData->GetScalarTypeMin();
  typeMax = outData->GetScalarTypeMax();
  clamp = self->GetClampOverflow();

  // find the region to loop over
  rowLength = (outExt[1] - outExt[0]+1)*
              inData->GetNumberOfScalarComponents();
  maxY = outExt[3] - outExt[2];
  maxZ = outExt[5] - outExt[4];
  target = (unsigned long)((maxZ+1)*(maxY+1)/50.0);
  target++;

  // Get increments to march through data
  inData->GetContinuousIncrements(outExt, inIncX, inIncY, inIncZ);
  outData->GetContinuousIncrements(outExt, outIncX,
                                   outIncY, outIncZ);
  // Loop through ouput pixels
  for (idxZ = 0; idxZ <= maxZ; idxZ++)
    {
    for (idxY = 0; !self->AbortExecute && idxY <= maxY; idxY++)
      {
      if (!id)
        {
        if (!(count%target))
          {
          self->UpdateProgress(count/(50.0*target));
          }
        count++;
        }
      if (clamp) // put the test for clamp to avoid the innermost loop
```

```
{
for (idxR = 0; idxR < rowLength; idxR++)
  {
  // Pixel operation
  val = ((float)(*inPtr) + shift) * scale;
  if (val > typeMax)
    {
    val = typeMax;
    }
  if (val < typeMin)
    {
    val = typeMin;
    }
  *outPtr = (OT)(val);
  outPtr++;
  inPtr++;
  }

else
  {
  for (idxR = 0; idxR < rowLength; idxR++)
    {
    // Pixel operation
    *outPtr = (OT)(((float)(*inPtr) + shift) * scale);
    outPtr++;
    inPtr++;
    }
}
outPtr += outIncY;
inPtr += inIncY;
  }
outPtr += outIncZ;
inPtr += inIncZ;
  }
}
```

Initially we computes the length of a row of pixels. This is just the number of pixels multiplied by the number of components per pixel (e.g. three for RGB, one for grayscale). The second and third lines compute the *y* and *z* dimensions. The next two lines get the continuous increments for the `inData` and `outData`. There are two types of increments that can be used to step through the data: regular and continuous. The regular increments specify how many units in memory you need to move in order to navigate though the image. The regular *x* increment specifies how many units to move to go from one pixel to the next. The regular *y* and *z* increments do the same for rows and slices, respectively. For example,

if `foo` was a pointer to a pixel in the image and you wanted to get the value of the pixel one row above `foo`, it would be `*(foo + yInc)`.

Continuous increments are retrieved using the following method invocations:

```
inData->GetContinuousIncrements(outExt, inIncX, inIncY, inIncZ);
outData->GetContinuousIncrements(outExt, outIncX,
                                 outIncY, outIncZ);
```

Continuous increments are a convenience that makes it easier to loop through the data in the current extent. The y and z continuous increments are the only values that get used. The y continuous increment indicates how many units in memory must be moved past to get from the end of one row to the beginning of the next. If the UpdateExtent matches the data's Extent that value will be zero. But when working on a sub-portion of the image, it may have a non zero value. The same idea applies for the z continuous increment. It indicates how many units in memory need to be skipped to get from the end of one slice to the beginning of the next.

Another important feature of this filter is that the ExecuteInformation() is overloaded.

```
void vtkImageShiftScale::ExecuteInformation(vtkImageData *inData,
    vtkImageData *outData)
{
  this->vtkImageToImageFilter::ExecuteInformation(inData,
                                                  outData );
  if (this->OutputScalarType != -1)
    {
    outData->SetScalarType(this->OutputScalarType);
    }
}
```

The purpose of this method is to overload the default superclass behavior that sets the output type to the same as the input type. In this particular filter, the user may optionally set the output type to a different type, hence the need for the method overload.

We hope that this chapter helps you write your own filters in VTK. You may wish to build on the information given here by studying the source code in other filters. It's particularly helpful if you can find an algorithm that you understand, and then look to see how VTK implements it.