Managing Pipeline Execution

This chapter provides details of the VTK execution process. Understanding this process will help you to write new filters, and work with VTK's streaming functionality. The chapter begins by describing the pipeline execution process, and then follows that with a description of how to use streaming in an application.

10.1 The Execution Process

The *Visualization Toolkit* uses an implicit execution mechanism. That is, there is no central executive controlling the execution of the visualization pipeline. While this is unusual as compared to many visualization systems, it has several advantages. First, the method is relatively simple in principle—time stamps are used to determine when objects are out of date. Such objects execute as necessary to bring themselves, and the system, up to date. Second, implicit execution is natural for parallel processing because there is no centralized bottleneck preventing distribution across processors. And finally, without a central executive it is relatively straightforward to change the behavior of the execution process by controlling it in your own application. For example, you can make VTK look like a functional system (versus data flow) simply by invoking Update() on a single filter in response to a user input (e.g., a menu choice), as well as dynamically changing the input and output of each filter as necessary.

As we saw in the overview section (see "Pipeline Execution" on page 26), pipeline execution is initiated by a request for data. Typically the request is initiated when a subclass of vtkProp is rendered. The render request results in an Update() method being invoked on a mapper, which in turn forwards the method to its data object. The Update() method in vtkDataObject consists of four separate steps as illustrated by the following.

```
void vtkDataObject::Update()
{
  this->UpdateInformation();
  this->PropagateUpdateExtent();
  this->TriggerAsynchronousUpdate();
  this->UpdateData();
}
```

Each of these methods is typically propagated recursively up the pipeline. In the first pass, the UpdateInformation() method travels up the pipeline (opposite the direction of data flow) and at least two important pieces of information come back down the pipeline—the WholeExtent (or MaximumNumberOfPieces) for each data object, and the PipelineMTime. The whole extent refers to the entire data that is available for processing, and is expressed as a 6-vector (x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}) for structured data. The PipelineMTime is the maximum modified time (time stamp) found in the pipeline.

In the PropagateUpdateExtent() pass, each process object in the pipeline determines how much data it will produce (which will be at least as big as the requested update extent of the outputs) and how much input data it requires for execution.

In the TriggerAsynchronousUpdate() pass, ports are given an opportunity to begin a non-blocking update in another process. (A port is an object that transmits data across the network, or between processes. Ports are used in distributed, parallel applications.) If no ports exist in the pipeline then TriggerAsynchronousUpdate() propagates up and then back down the pipeline doing nothing.

The final stage is the UpdateData() pass where the actual execution of each source occurs on the way back up the pipeline via the process object Execute() method.

The reason for these methods, and the information they return, is probably somewhat of a mystery to you at this point. Besides the basic need to control the execution of filters, the execution process supports two other important features. First, it allows data to be *streamed*—that is, data can be divided into pieces (and processed piece-by-piece) to reduce memory consumption (see "Using Streaming" on page 232). Applications can also request just a piece of data. This can be useful when developing applications requiring high interactivity or when the user is only interested in a portion of the dataset. Second, the execution process supports asynchronous, distributed, parallel processing. Such a capability is necessary when visualizing large datasets and the problem must be distributed across many processors.

The next few sections will cover the execution process in more detail. This information is important if you plan to write your own process object. If you simply plan to make use of existing sources and filters, then these internal details are not necessary but they may help you better understand the behavior and capabilities of the pipeline execution model.

Overview and Terminology

As mentioned previously, each of the four stages of the Update() process may propagate recursively back up the pipeline. Of course, we do not want to execute process objects if it is not necessary, so we would like to terminate propagation when appropriate. In the first stage of the Update() process, the UpdateInformation() pass traverses up the entire pipeline since this is necessary for computing the PipelineMTime. In the other three stages, the data objects consult their PipelineMTime, their UpdateTime and whether or not their data has been released to determine whether they will need to be regenerated. The propagation continues through the data object only if regeneration of the data is necessary.

It is assumed that when the Update() method is called, the desired update extent has been set to something valid. If the update extent has not been set before the first call to UpdateInformation(), then the update extent is set to the whole extent of the data.

Data objects have two possible ways of storing extents. Which way the information is stored is indicated by the return value of GetExtentType() which may be `VTK_PIECES_EXTENT` or `VTK_3D_EXTENT`. Data objects that use pieces to define their extent store the MaximumNumberOfPieces that the data object can be broken into, the Piece out of how many NumberOfPieces that was created last time this data object was updated, and the UpdatePiece out of how many UpdateNumberOfPieces is being requested currently. If the UpdatePiece is -1 and the UpdateNumberOfPieces is 0, then no data is requested and the data object will remain empty after the update.

As mentioned previously, data objects that use 3D extents store all their extent information as six integers defining the $(x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max})$ bounds of some structured data region. These data objects keep track of their WholeExtent which is the maximum size of the data, their Extent which is the size of the data last generated by an update, and their UpdateExtent which is the currently requested data size. The UpdateExtent must lie completely within the WholeExtent or must be a special invalid extent with $(x_{max} < x_{min})$ or $(y_{max} < y_{min})$ or $(z_{max} < z_{min})$. If this special extent is encountered during the UpdateInformation pass, then no data requested and the data object will remain empty after the update.

If Update() is called on a vtkProcessObject (or vtkSource) subclass instead of a vtkDataObject subclass, the source simply calls the Update() method of its first output (as shown in the code following this paragraph). This way, the update extent of the first output can be considered. Please note that we will use the generic term "update extent" to refer to

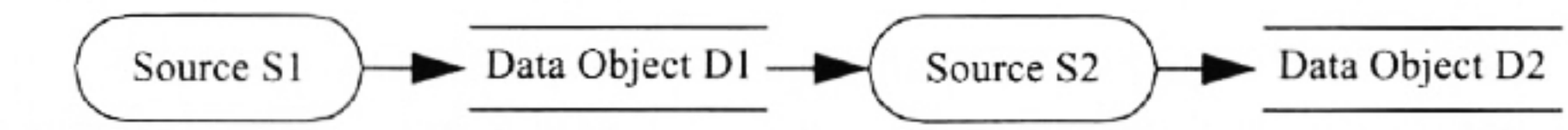

Figure 10–1 Two example pipeline configurations used to illustrate the ideas covered in this chapter. Here we are referring to process objects as "sources" of data.

either the UpdateExtent or the UpdatePiece depending on the actual data type of the data object.

```
void vtkSource::Update()
{
  if (this->GetOutput(0))
    {
    this->GetOutput(0)->Update();
    }
}
```

The two pipelines shown in **Figure 10–1** will be used to illustrate issues with the update mechanism. The first represents a simple case such as a reader providing data to a single filter. The second represents a slightly more complex case of a reader that provides data to two filters, which in turn provide their data to a two-input filter that produces one resulting output.

Looking at the example pipeline at the top of **Figure 10–1**, and assuming that it has not executed before (so all calls will propagate all the way up the pipeline), the high-level calls that would be made are illustrated by the following pseudo-code.

```
D2->Update()
  D2->UpdateInformation()
    S2->UpdateInformation();
      D1->UpdateInformation();
```

```
        S1->UpdateInformation();
  D2->PropagateUpdateExtent(D2);
    S2->PropagateUpdateExtent();
      D1->PropagateUpdateExtent(D1);
        S1->PropagateUpdateExtent();
  D2->TriggerAsynchronousUpdate();
    S2->TriggerAsynchronousUpdate();
      D1->TriggerAsynchronousUpdate();
        S1->TriggerAsynchronousUpdate();
  D2->UpdateData(D2);
    S2->UpdateData();
      D1->UpdateData(D1);
        S1->UpdateData();
```

Here we simply propagate each stage of the update independent of all the others. Looking at example pipeline B, we can see that this simple approach will not work in this case. The reason is that S4 might require a different input update extent than S5. If these stages are propagated independently, then when S4 goes to execute (within the UpdateData method), D3 will have the update extent that was requested by S5. Therefore, if a source has multiple inputs (as S6 does), then the update extent must be propagated again before calling either UpdateData() or TriggerAsynchronousUpdate().

The UpdateInformation() Pass

The UpdateInformation() method for vtkDataObject is shown in the following code. This method first calls UpdateInformation() on its source. After this, it checks if the update extent is invalid. If it is, then it sets the UpdatePiece to 0 and the UpdateNumberOfPieces to 1 in the case of data objects that use pieces for extent, or it copies the WholeExtent to the UpdateExtent in the case of data objects that use 3D extents.

```
void vtkDataObject::UpdateInformation()
{
  if (this->Source)
    {
    this->Source->UpdateInformation();
    }
  else
    {
    //If we don't have a source, then let's make our whole
    //extent equal to our extent.
    memcpy( this->WholeExtent, this->Extent, 6*sizeof(int) );
    //We also need to set the PipelineMTime to our MTime.
```

```
    this->PipelineMTime = this->GetMTime();
    }

  // Now we should know what our whole extent is. If our update extent
  // was not set yet, then set it to the whole extent.
  if(! this->UpdateExtenInitialized)
    {
    this->SetUpdateExtentToWholeExtent();
    this->UpdateExtentInitialized = 1;
    }
}
```

The UpdateInformation() method for vtkSource is shown in the code following this paragraph. To avoid infinite recursion, the first thing that is done for all update stages in vtkSource is to check whether we are already updating. This would occur if there were a loop in the pipeline. If so, we set the PipelineMTime of our outputs to ensure we will execute, and then break the loop. Otherwise, we proceed to propagate the UpdateInformation() call to each of the inputs of this source. In addition, we compare the MTime of this vtkSource with the PipelineMTime of each input to find the largest value. This will be used to set the PipelineMTime of each of our outputs. If the PipelineMTime is larger than the InformationTime, the pipeline was modified since the last UpdateInformation pass, and the ExecuteInformation() method is called. The default implementation of ExecuteInformation() in vtkSource simply copies information from the first input to the outputs. Filters such as vtkImageClip that modify the WholeExtent, must implement their own version of ExecuteInformation(). A variable called locality is also computed in UpdateInformation(). This variable is used during the UpdateData() call to avoid serial execution of parallel pipelines.

```
void vtkSource::UpdateInformation()
{
  unsigned long t1, t2;
  int           idx;
  vtkDataObject *input;
  vtkDataObject *output;
  float         maxLocality = 0.0;
  float         locality;

  // Watch out for loops in the pipeline
  if ( this->Updating )
    {
    // Since we are in a loop, execute on every call to update.
    // We set the pipline mtimes of our outputs
    // to ensure the pipeline executes again.
```

```
this->Modified();
for (idx = 0; idx < this->NumberOfOutputs; ++idx)
  {
  output = this->GetOutput(idx);
  if (output)
    {
    output->SetPipelineMTime(this->GetMTime());
    }
  }
return;
}

// The MTime of this source will be used in determine the PipelineMTime
// for the outputs
t1 = this->GetMTime();

// Loop through the inputs
for (idx = 0; idx < this->NumberOfInputs; ++idx)
  {
  if (this->Inputs[idx] != NULL)
    {
    input = this->Inputs[idx];

    // Propagate the UpdateInformation call
    this->Updating = 1;
    input->UpdateInformation();
    this->Updating = 0;

    // Compute the max locality of the inputs.
    locality = input->GetLocality();
    if (locality > maxLocality)
      {
      maxLocality = locality;
      }

    // What is the PipelineMTime of this input? Compare this against
    // our current computation to find the largest one.
    t2 = input->GetPipelineMTime();

    if (t2 > t1)
      {
      t1 = t2;
      }
    }
  }
locality = maxLocality * 0.5;
```

```
// Call ExecuteInformation for subclass specific information.
// Since UpdateInformation propagates all the way up the pipeline,
// we need to call ExecuteInformation only if necessary.
if (t1 > this->InformationTime.GetMTime())
  {
  for (idx = 0; idx < this->NumberOfOutputs; ++idx)
    {
    output = this->GetOutput(idx);
    if (output)
      {
      output->SetPipelineMTime(t1);
      output->SetLocality(locality);
      }
    }

  this->ExecuteInformation();
  }
}
```

The PropagateUpdateExtent() Pass

The PropagateUpdateExtent() method for vtkDataObject is shown in the code following this paragraph. The first thing that happens is a check for an empty request. The call returns immediately if no data is requested. The next condition checks to see if the data object is already up to date. The UpdateTime is compared against the PipelineMTime, we check whether the data has been released, and we see if the update extent is already in the data object.

```
void vtkDataObject::PropagateUpdateExtent()
{
  if (this->UpdateExtentIsEmpty())
  {
  return;
  }

// If we need to update due to PipelineMTime, or the fact that our
// data was released, then propagate the update extent to the source
// if there is one.
if ( this->UpdateTime < this->PipelineMTime || this->DataReleased ||
     this->UpdateExtentIsOutsideOfTheExtent())
  {
  if (this->Source)
    {
```

```
      this->Source->PropagateUpdateExtent(this);
      }
    }

  // Check that the update extent lies within the whole extent
  this->VerifyUpdateExtent();
}
```

The next step in the PropagateUpdateExtent() method of vtkDataObject is to actually propagate the call to the source. This is done only if necessary according to the PipelineMTime, or if the data has been released. After the propagation, we do some quick error checking to make sure the update extent has not been set to something that cannot be satisfied. PropagateUpdateExtent() method for vtkSource is shown in the following code. As usual, the first step is to return if we detect that we are in a loop. The next step is to set the default value for RequestExactExtent. This flag is used as a part of structured data filters update extent. It indicates whether the filter is capable of handling more input than requested. In the next step, the subclass is asked how much input data is necessary to produce the output data. The default implementation is to ask for everything, although this is overridden in many subclasses. For example, vtkImageToImageFilter assumes that it needs the same input update extent as the output update extent. In the vtkImageSpatialFilter this is overridden again to add in the padding needed for execution. The final step in the PropagateUpdateExtent() method of vtkSource is to actually propagate the update extent.

```
void vtkSource::PropagateUpdateExtent(vtkDataObject *output)
{
  // Check flag to avoid executing forever if there is a loop.
  if (this->Updating)
    {
    return;
    }

  // Set the default value of RequestExactExtent.
  // This indicates that filters can handle more data than requested.
  for (idx = 0; idx < this->NumberOfInputs; ++idx)
    {
    if (this->Inputs[idx] != NULL)
      {
      this->Inputs[idx]->RequestExactExtentOff();
      }
    }

  // Give 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
  // neighborhood of surrounding original values.
  this->ComputeInputUpdateExtents( output );

  // Now that we know the input update extent, propagate this
  // through all the inputs.
  this->Updating = 1;
  for (idx = 0; idx < this->NumberOfInputs; ++idx)
    {
    if (this->Inputs[idx] != NULL)
      {
      this->Inputs[idx]->PropagateUpdateExtent();
      }
    }
  this->Updating = 0;
}
```

The TriggerAsynchronousUpdate() Pass

The TriggerAsynchronousUpdate() method of vtkDataObject is shown in the following code. This method simply propagates the trigger method to its source depending on its PipelineMTime or if the data was released.

```
void vtkDataObject::TriggerAsynchronousUpdate()
{
  // If we need to update due to PipelineMTime, or the fact that our
  // data was released, then propagate the trigger to the source
  // if there is one.
  if ( this->UpdateTime < this->PipelineMTime || this->DataReleased )
    {
    if (this->Source)
      {
      this->Source->TriggerAsynchronousUpdate();
      }
    }
}
```

The TriggerAsynchronousUpdate() method of vtkSource is shown in the following code. After checking that we are not in a pipeline loop, the trigger is propagated to all inputs.

The TriggerAsynchronousUpdate() method will be overridden in the port objects to allow updates to begin in other processes without blocking until the data is requested.

```
void vtkSource::TriggerAsynchronousUpdate()
{
  // check flag to avoid executing forever if there is a loop
  if (this->Updating)
    {
    return;
    }
  // Propagate the trigger to all the inputs
  this->Updating = 1;
  for (int idx = 0; idx < this->NumberOfInputs; ++idx)
    {
    if (this->Inputs[idx] != NULL)
      {
      this->Inputs[idx]->TriggerAsynchronousUpdate();
      }
    }
  this->Updating = 0;
}
```

The UpdateData() Pass

The UpdateData() method of vtkDataObject is shown in the code following this paragraph. This method simply propagates the UpdateData() to the source if it is necessary according to the PipelineMTime or if the data was released.

```
void vtkDataObject::UpdateData()
{
  // If we need to update due to PipelineMTime, or the fact that our
  // data was released, then propagate the UpdateData to the source
  // if there is one.
  if ( this->UpdateTime < this->PipelineMTime || this->DataReleased )
    {
    if (this->Source)
      {
      this->Source->UpdateData(this);
      }
    }
}
```

The UpdateData() method of vtkSource is shown in the following code. This method begins by verifying that we are not in a pipeline loop. Then the UpdateData() method is propagated to all the inputs to make sure everything is up-to-date before this source executes. If there is only one input, then we simply need to propagate the UpdateData() call, but if we have more than one we must propagate the PropagateUpdateExtent() method before the UpdateData() method to ensure that all upstream data objects have the right update extent. The next step is to initialize all the outputs in preparation for execution. If there is a start method it is called, then execute is called, then the end method if there is one defined. The outputs are marked as up-to-date, and we release any inputs that require it. Finally, we update the InformationTime to indicate that we have updated this filter.

```
void vtkSource::UpdateData(vtkDataObject *output)
{
  int idx;
  // prevent chasing our tail
  if (this->Updating)
    {
    return;
    }
  // Propagate the update call - make sure everything we might rely on is
  // up-to-date
  // Must call PropagateUpdateExtent before UpdateData if multiple inputs
  // since they may lead back to the same data object.
  this->Updating = 1;
  if ( this->NumberOfInputs == 1 )
    {
    if (this->Inputs[0] != NULL)
      {
      this->Inputs[0]->UpdateData();
      }
    }
  else
    {
    for (idx = 0; idx < this->NumberOfInputs; ++idx)
      {
      if (this->Inputs[idx] != NULL)
       {
        this->Inputs[idx]->PropagateUpdateExtent();
        this->Inputs[idx]->UpdateData();
       }
      }
    }
  this->Updating = 0;
```

```
// Initialize all the outputs
for (idx = 0; idx < this->NumberOfOutputs; idx++)
  {
  if (this->Outputs[idx])
    {
    this->Outputs[idx]->Initialize();
    }
  }

// If there is a start method, call it
this->InvokeEvent(vtkCommand::StartEvent,NULL);
// Execute this object - we have not aborted yet, and our progress before
// we start to execute is 0.0.
this->AbortExecute = 0; this->Progress = 0.0;
this->Execute();
// If we ended due to aborting, push the progress up to 1.0 (since it
// probably didn't end there)
if ( !this->AbortExecute )
  {
  this->UpdateProgress(1.0);
  }
// Call the end method, if there is one
this->InvokeEvent(vtkCommand::EndEvent,NULL);
// Now we have to mark the data as up to data.
for (idx = 0; idx < this->NumberOfOutputs; ++idx)
  {
  if (this->Outputs[idx])
    {
    this->Outputs[idx]->DataHasBeenGenerated();
    }
  }

// Release any inputs if marked for release
for (idx = 0; idx < this->NumberOfInputs; ++idx)
  {
  if (this->Inputs[idx] != NULL)
    {
    if ( this->Inputs[idx]->ShouldIReleaseData() )
     {
      this->Inputs[idx]->ReleaseData();
     }
    }
  }

// Information gets invalidated as soon as Update is called,
// so validate it again here.
```

```
  this->InformationTime.Modified();
}
```

This description of the pipeline execution process may be useful when you wish to add a new process object or data object to the system. Refer to "How To Write A Graphics Filter" on page 280 for information about adding a filter to VTK.

10.2 Using Streaming

Visualization often involves working with large data. When data size becomes large, applications often become slow (due to excessive paging) or even fail, as virtual memory limits are exceeded. The *Visualization Toolkit* offers a technique to address this problem. This technique is referred to as *data streaming*. (Also see "The Execution Process" on page 219 for more details.)

In VTK, streaming is the process of separating large data into pieces, and then processing the dataset piece-by-piece. The advantage of streaming is that applications can avoid swapping or system failure due to memory constraints. On the other hand, streaming requires the ability to break data into pieces, adding complexity to the pipeline execution process as well as algorithms that process the data.

There are several other benefits to streaming. Streaming naturally lends itself to parallel processing, since streaming supports a parallel data model (i.e., process piece *n* of *m* total pieces). Streaming is also useful when supporting applications requiring high levels of interactivity since just a subset of the total data is required. It is easy to imagine an application that processes pieces ranked in order of importance, displays highest-ranked results first, and then allows the user to interact with the results, possibly interrupting data processing according to what is being visualized.

Streaming is a complex piece of functionality. The streaming mechanism has been reworked several times to insure that it is robust and easy to use. While this goal has been generally realized for certain types of data (e.g., vtkImageData—images and volumes), this is an ongoing research and implementation challenge in VTK. In particular, streaming implementations for unstructured data are currently under development, and you will likely see future changes.

The following C++ example demonstrates how to use streaming in the imaging pipeline. The example reads a volume of 256^2 x 93, magnifies the input data 3x3x1, computes the

gradient, and then shrinks the result (the peak memory size is over 750 MByte without streaming).

The key step is the addition of the streamer object vtkImageDataStreamer. The streamer is told to divide its processing into 20 pieces. Its output update extent is separated into 20 non-overlapping pieces, and the input is updated on each piece separately. Exactly how the extent is divided can be controlled by using the vtkExtentTranslator class contained in the streamer. The vtkImageDataStreamer manages the pipeline execution so that the peak memory size does not exceed 40 megabytes.

```
#include "vtkImageViewer.h"
#include "vtkImageReader.h"
#include "vtkImageMagnify.h"
#include "vtkImageShrink3D.h"
#include "vtkImageGradient.h"
#include "vtkImageDataStreamer.h"
main ()
{
  int i, j;

  vtkImageReader *reader = vtkImageReader::New();
  reader->SetFilePrefix("$VTK_DATA_ROOT/Data/headsq/quarter");
  reader->SetDataByteOrderToLittleEndian();
  reader->SetDataExtent(0,255,0,255,1,93);
  reader->SetDataMask( 0x7fff);

  vtkImageMagnify *mag = vtkImageMagnify::New();
  mag->SetInput(reader->GetOutput());
  mag->SetMagnificationFactors(3,3,1);

vtkImageGradient *grad = vtkImageGradient::New();
grad->SetInput(mag->GetOutput());
grad->SetDimensionality(3);

vtkImageShrink3D *shrink = vtkImageShrink3D::New();
shrink->SetInput(grad->GetOutput());
shrink->SetShrinkFactors(3,3,1);

vtkImageDataStreamer *ids = vtkImageDataStreamer::New();
ids->SetInput(shrink->GetOutput());
ids->SetNumberofStreamDivisions(20);
ids->UpdateWholeExtent();

vtkImageViewer *viewer = vtkImageViewer::New();
```

```
  viewer->SetInput(ids->GetOutput());
  viewer->SetColorLevel(0);
  viewer->SetColorWindow(200);

  // interact with data
  for (j = 0; j < 10; j++)
    {
    for (i = 1; i < 93; i++)
      {
      viewer->SetZSlice(i);
      viewer->Render();
      }
    }

  // Clean up
  viewer->Delete();
  reader->Delete();
  grad->Delete();
  shrink->Delete();
  mag->Delete();
}
```

There are limits to what can be accomplished with streaming. As mentioned earlier, support for streaming unstructured data is minimal. Many algorithms, such as vtkImageAccumulate, cannot be streamed effectively because every output pixel requires a visit to every input pixel. (Of course multipass algorithms are possible.) Some algorithms are written assuming that the entire input dataset is available. i.e., they have not been written with streaming in mind (e.g., such as volume rendering in Chapter 6).