3

The Visualization Pipeline

The visualization pipeline refers to the steps that data undergoes as it is transformed from raw formats into final visualizations. The term has both an abstract meaning, in the sense of a high level characterization of the main processing steps; and a concrete one, where the high level representation is mapped to a set of computational steps, or procedures. Often in visualization, these computation steps are organized in terms of a dataflow program. We will cover each of these concepts below.

3.1 Dataflow Programming and Scripting

Visualization systems (as other large-scale software projects) are often built in multiple layers. The idea is to create abstractions that make them more understandable, flexible, and easier to use. One of the most common and useful abstractions for visualization systems is the notion of a dataflow computation.

A dataflow is a directed graph where nodes represent computations and edges represent streams of data: each node or module corresponds to a procedure that is applied on the input data and generates some output data as a result. The flow of data in the graph determines the order in which the processing nodes are executed. In a dataflow computing model, a change in one variable should force the recalculation of other connected variables (for instance, cells in a spreadsheet). In visualization, it is common to refer to a dataflow network as a "visualization pipeline".

The terms dataflow, workflow, pipeline, and network as pertains to visualization are often used interchangeably.

Workflows are more general than dataflows. In traditional dataflows, the flow of the data is fixed by the graph, and the firing (execution) of the different modules is data driven. Workflows, on the other hand, specify the movement of information through a work process. Here, it is possible to specify how tasks are structured, who performs them, what their relative order is, how they are synchronized, how information flows to support the tasks, and how tasks are being tracked. Workflows are commonly used in the scientific community to interconnect tools, handle multiple data formats and large quanitities of data. Although, technically speaking, the term "workflow" represents a different concept, for our purposes, dataflow, workflow, pipeline, and network as pertains to visualization

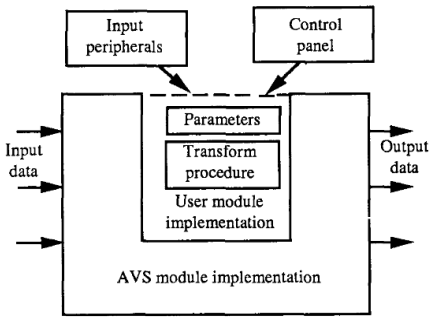


Figure 3.1. The conceptual model of a module.

are used interchangeably.

The actual computation is performed by the modules by operating on their data. It is sometimes useful to think of the modules as doing different parts of the computation. Common types include: modules that generate data (sources), consume data (sinks), and operate on data by applying algorithms to the data (processes). In their most general form, each module has a set of inputs, set of outputs, a set of parameters, and, in certain systems, some form of GUI that can be used for interacting with the modules (see Figure 3.1). In general, these inputs and outputs are called "ports". It is useful to enforce "types" in these systems, only allowing connections between ports of compatible types.

The order of execution of modules in a dataflow network is also very important, and system dependent. There are many tricky issues involved, in particular for very large datasets where it may be necessary to perform computations under a "streaming model", that is, instead of assuming that all the data is available at once to a particular model, it might be necessary to change the algorithms to be able to work on small chunks of data, which are streamed piece by piece through the processing pipeline. We will limit our discussion of the execution strategies to a minimum, since these are out of the scope of our text.

As in other areas of scientific computing, over the last ten years or so, traditional programming of dataflow networks has been replaced by scripting. We use the term scripting to mean programming in higher level dynamically typed languages (e.g., Python or Tcl). Scripting has shown to be very useful in "glueing" many pieces of software, in particular in the realm of scientific computing. It simplifies writing user interfaces, providing modern interfaces to old codes, rapid prototyping, *etc.* Because scripting languages are often compiled "on the fly", they can also be used by advanced users for extending system functionality at runtime, without the need for expensive recompilation steps. Scripting in Python or Tcl are the defacto standards for programming with the popular Visualization Toolkit (VTK) that will be described below.

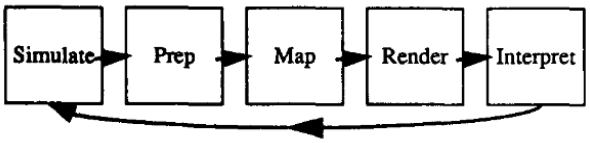


Figure 3.2. The visualization pipeline.

3.2 The Visualization Pipeline as a Dataflow Network

Now let us go back to the notion of a visualization pipeline as a means to create visual representations of raw data. It is fairly common to break up (at least, conceptually) the visualization pipeline into three main phases: *filter*, *map*, and *render*, that each data item will pass through to be visualized (see Figure 3.2). In practice, there might be steps that need to happen before (*e.g.,* reading the data from disk and uncompressing it) or after (*e.g.,* image deliveray). Also, depending on the exact visualization algorithm being applied, one can think of "skipping" a step. This "3-stage pipeline" is a bit too simplistic, but despite its potential faults, it helps to provide a general framework for setting up visualization pipelines.

The goal of the filtering phase is to perform data preparation. It can take the raw simulated or sensored data and transform it into another form that is more informative or less voluminous. It might also be generating auxiliary information that is not readily available, but is necessary for later processing steps. For instance, it might be doing data resampling, interpolation, or gradient calculation.

The mapping phase will turn the filtered data into geometric primitives that can be rendered. For instance, it might generate OpenGL display lists from triangulated models or create polygonalized versions of glyphs.

The rendering phase takes the geometric phase and creates pictures from them, taking advantage of graphics hardware where possible. The goal here is to generate actual images from the descriptions generated in the previous steps.

The relationship between the conceptual visualization pipeline, and its implementation as a dataflow network can be a bit fuzzy. To make things more concrete, in Figure 3.3 we show a dataflow network that has been segmented into filter, map, and render phases.

3.3 Dataflow Programming with the Visualization Toolkit

Kitware's Visualization Toolkit (VTK) [Kitware b] is an open source, object-oriented toolkit with based on the dataflow programming model. This model allows complex applications to be built upon smaller pieces. For efficiency, the core components of the system are written in a compiled language (C++). For flexibility and extensibility, an interpreted language can be used for higher level applications (Python, Tcl, or Java). This scripting capability and large core of

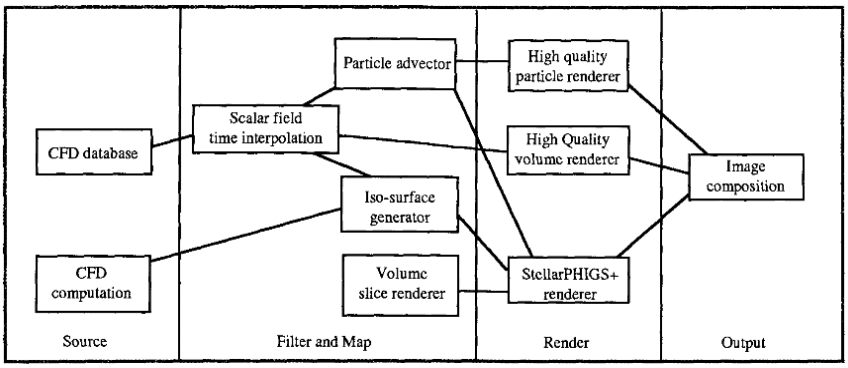


Figure 3.3. An example dataflow categorized within the visualization pipeline.

algorithms has promoted VTK to its current status as a one of the most popular visualization packages for researchers. In addition, larger systems such as ParaView [Kitware a] leverage the toolkit to provide specific applications with an interface that is much simpler for an end user than scripting.

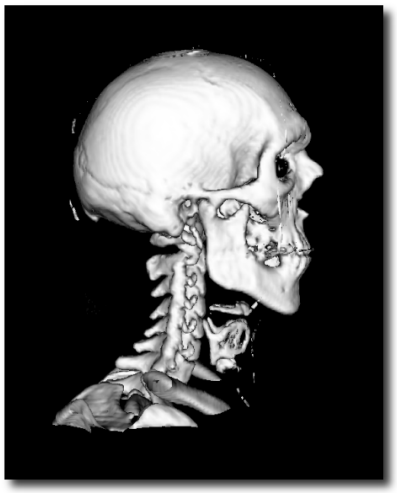
VTK uses two models for creating a visualization [Schroeder et al. 96,Schroeder et al. 06, Kit 06]. The visualization model takes information and produces a geometric representation that the graphical model displays to the screen. The graphical model consists mainly of the following basic objects:

- Render Windows: The object which manages a window on the display device.
- Renderers: The object which coordinates the lights, cameras, and actors of the scene and draws them into the render window.
- Props: The objects added to the renderers to create a scene. The props are the things that you see in the scene.
- Mappers: The object that refer to an input data object and knows how to transform and render it.
- Properties: The object that contains rendering parameters such as color and material properties.

These objects are abstract base classes in VTK, and their derived objects are nodes that can be connected together to form a scene graph, an acyclic, directed graph that defines the rendering process.

The visualization model of VTK uses the graphical model in a dataflow paradigm to create visualization pipelines. There are two object types in the visualization model:

```
import vtk
data = vtk.vtkStructuredPointsReader()
data.SetFileName("..examples/data/head.120.vtk")

contour = vtk.vtkContourFilter()
contour.SetInput(0,data.GetOutput())
contour.SetValue(0, 67)

mapper = vtk.vtkPolyDataMapper()
mapper.SetInput(contour.GetOutput())
mapper.ScalarVisibilityOff()

actor = vtk.vtkActor()
actor.SetMapper(mapper)

ren = vtk.vtkRenderer()
ren.AddActor(actor)
renwin = vtk.vtkRenderWindow()
renwin.AddRenderer(ren)

style = vtk.vtkInteractorStyleTrackballCamera()
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renwin)
iren.SetInteractorStyle(style)
iren.Initialize()
iren.Start()
```



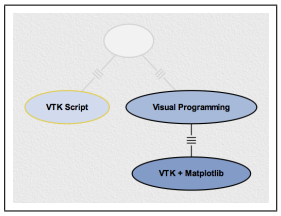


Figure 3.1: The Python script for a simple VTK example the resulting visualization.

- Process Objects: The sources, filters, and mapper algorithms that manipulate the data.

- Data Objects: The datasets that define the dataflow through the network.

The process objects are the modules that are connected into a dataflow network using SetInput() and GetOutput() methods. The execution of the dataflow within VTK is controlled in response to demands for the data (demand-driven) or in response to user input (event-driven).

As a simple example, consider the Python script in Figure 3.1 that describes a simple VTK pipeline for extracting and displaying an isosurface (or contour) from a 3D structured dataset. First, the `vtkStructuredPointsReader` source module reads the data object from a file. Next, a `vtkCountourFilter` filter module takes the data object and produces another different data object. The `vtkPolyDataMapper` mapper module then transforms the data object into geometric primitives which are managed by a `vtkActor` module. Finally, the `vtkRenderer` module is responsible for drawing an image from the geometric primitives into a window that is managed by the `vtkRenderWindow` module.

3.4 Dataflow Programming with VisTrails

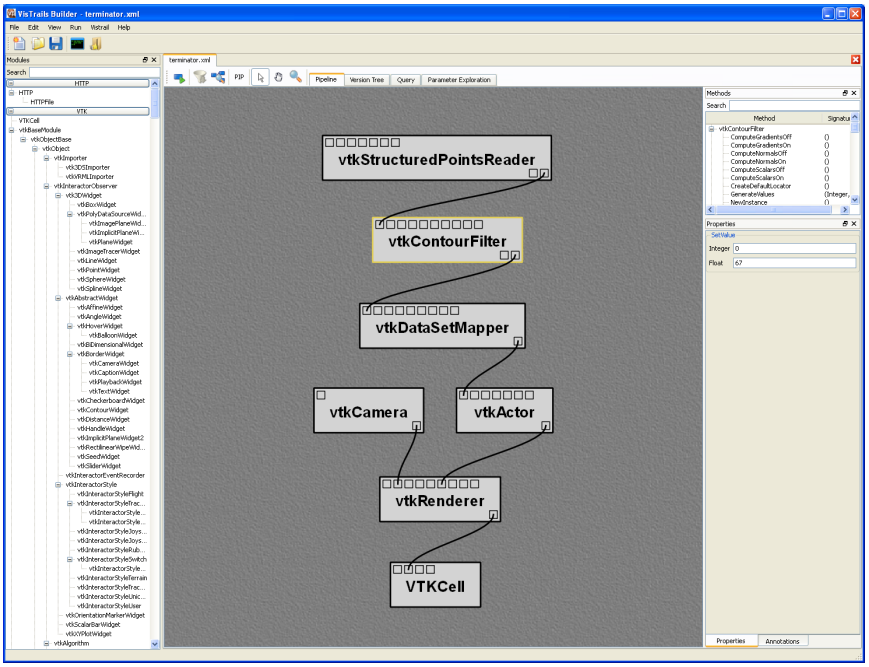
Scripting provides the power to rapidly create visualization pipelines. However, there are limitations to using scripts for anything but small tasks. In particular, the user needs to expend substantial effort managing the data (*e.g.,* scripts, raw data, data products, images, and notes) and maintaining complex scripts is difficult for anyone but the creator. Visual programming interfaces have been developed to simplify the task of workflow creation, modification, and reuse. They have the following advantages over scripting: they allow modular reuse and application interoperability, debugging and monitoring of the workflow execution, automated data managemement (*e.g.,* provenance), and immediate validation (*e.g.,* data, structural, and semantic typing) [Gertz and Ludäsher 06]. Some examples of systems for visual programming workflows are VisTrails [Callahan et al. 06b], MayaVi [Ramachandran 01], SCIRun [Parker and Johnson 95], AVS [Upson et al 89], and OpenDX [IBM].

Unlike other visualization systems, VisTrails captures detailed provenance of the exploratory process. VisTrails uses an action-based provenance model that uniformly captures both changes to parameter values and to pipeline definitions by unobtrusively tracking all changes that users make to pipelines in an exploration task [Callahan et al. 06a]. We refer to this detailed provenance of the pipeline evolution as a visualization trail, or a *vistrail*. The stored provenance ensures reproducibility of the visualizations, and it also allows scientists to easily navigate through the space of pipelines created for a given exploration task. The VisTrails interface gives users the ability to query, interact with, and understand the history of the visualization process. In particular, they can return to previous versions of a pipeline and change the specification or parameters to generate a new visualization without losing previous changes. Another important feature of the action-based provenance model is that it enables a series of operations that greatly simplify the exploration process and have the potential to reduce the time to insight. In particular, it allows the flexible re-use of pipelines and it provides a scalable mechanism for creating and comparing a large number of visualizations as well as their corresponding pipelines.

The VisTrails system consists of three major components. The Builder is used to visually create and maintain the visualization pipelines. The History Tree provides an interface for accessing the complete provenance of the exploration process. Finally, the Spreadsheet is used for comparative visualization of the pipeline executions. More detail on creating and managing workflows with VisTrails is available in the user's guide [**?**] on the project web page.

VisTrails software and documentation is available at http://www.vistrails.org.

Dataflow programming in VisTrails Builder is straightforward. Figure 3.2 shows the VTK script described in the previous section as a pipeline within VisTrails. Modules are shown as gray rectangles with input ports on top, and output ports on bottom. Connections defining the dataflow between modules are shown with curved black lines. Labels on each module indicate the corresponding VTK class. In this figure, it is natural to think of data flowing from top to bottom,



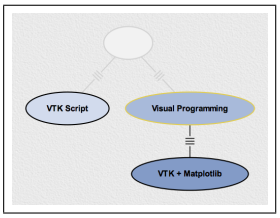


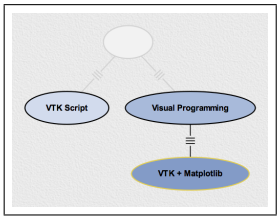
Figure 3.2: The dataflow builder interface of VisTrails for the simple VTK script shown previously.

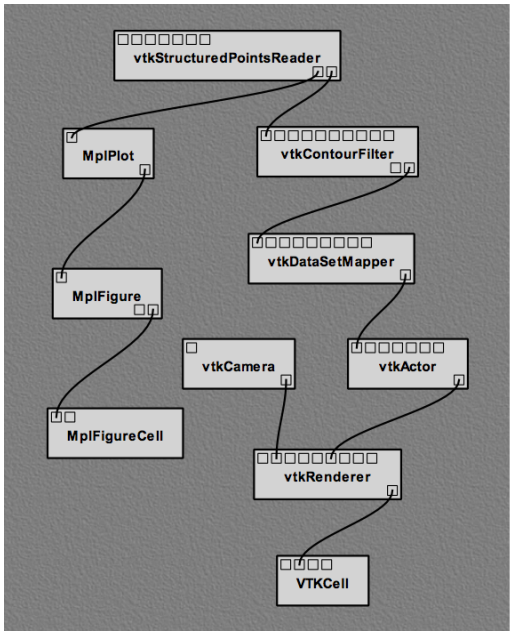
eventually being rendered, and presented for display.

Regardless of the specific mechanism used to define a pipeline, the end goal of the visualization process is to gain insight from the data. And often, to obtain insight, users must generate and compare multiple visualizations. Going back to our example, there are several alternatives to render structured volume data. The option we used above is the commonly used technique of isosurfacing. Given a function $f : \mathbf{R}^n \rightarrow \mathbf{R}$ and a value a, an *isosurface* consists of the set of points in the domain that map to a, *i.e.*, $\mathscr{S}_a = \{x \in \mathbf{R}^n : f(x) = a\}$.

The range of values of a determines all possible isosurfaces that can be generated. To identify "good" values of a that represent important features of a data set, it is useful to look at the range of values taken by a, and their frequency, in the form of a histogram. Using VisTrails, it is straightforward to extend the isosurface pipeline to also display a histogram of the data. VisTrails provides a very simple plugin functionality that can be used to add packages and libraries, including your own! For our example, we used the 2-D plotting functionality of matplotlib (http://matplotlib.sourceforge.net) to generate the histogram shown in Figure 3.3.

This histogram helps in the data exploration process by suggesting regions





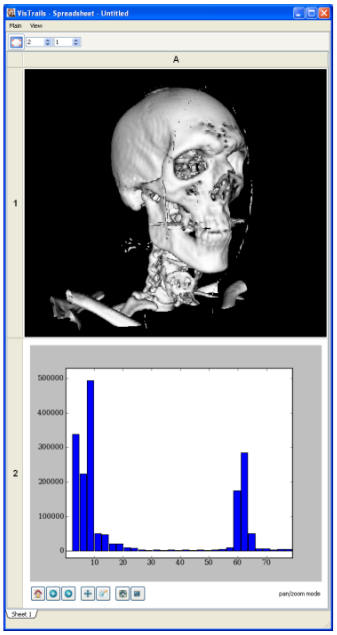
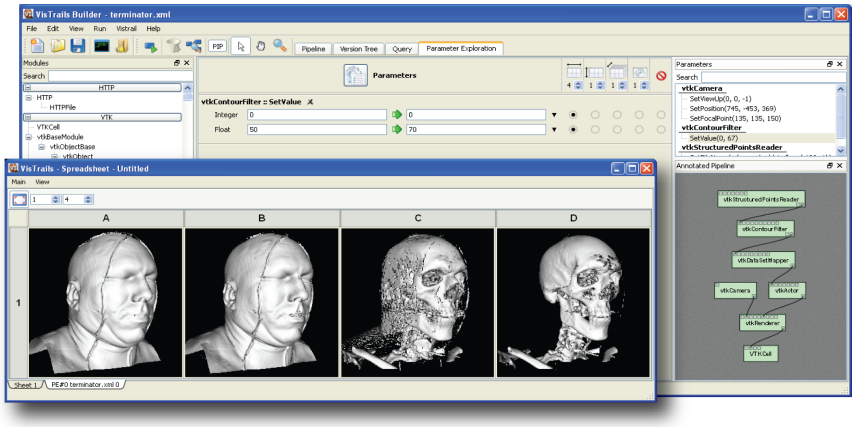


Figure 3.3: Multiple libraries are combined to create two seperate visualizations of the same data. The histogram uses Matplotlib.

of interest in the volume. The plot shows that the highest frequency features lie between the ranges [0,25] and [58,68]. To identify the features the correspond to these ranges, we need to explore these regions directly through visualization.

VisTrails provides an interface for parameter exploration. The interface allows the user to specify a set of parameters to explore, as well as how to explore, group, and display them. As a simple two dimensional example, we show an exploration of the isosurface value as four steps between 50 and 70 and display them horizontally in the VisTrails Spreadsheet. We also show an exploration of a volume simplification method vertically in the same Spreadsheet. Figure 3.4 shows the interface as well as the results of this exploration. The VisTrails Spreadsheet provides the means to compare visualizations in different dimensions (row, column, sheet, time). The cells in the Spreadsheet can be linked to synchronize the interactions between visualizations. We note that VisTrails leverages the dataflow specifications to identify and avoid redundant operations. By using the same cache for different cells in the spreadsheet, VisTrails allows the efficient exploration of a large number of related visualizations.



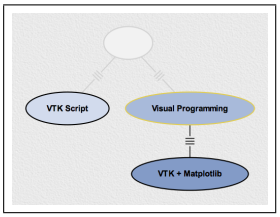


Figure 3.4: Parameter exploration is performed in VisTrails using a simple interface. The results are computed efficiently by avoiding redundant computation and displayed in the spreadsheet for interactive comparative visualization.