Data Interface & Miscellaneous

In this chapter we briefly describe various ways to read, write, import, and export data. Readers ingest a single dataset, while importers create an entire scene, which may include one or more datasets, actors, lights, cameras, and so on. Writers output a single dataset to disk (or stream), and exporters output an entire scene. In some cases, you may want to interface to data that is not in standard VTK format, or in any other common format that VTK supports. In such circumstances, you may wish to treat data as field data, and convert it in the visualization pipeline into datasets that the standard visualization techniques can properly handle.

8.1 Readers

We saw in "Reader Source Object" on page 47 how to use a reader to bring data into the visualization pipeline. Using a similar approach, we can read many other types of data. Here's a list of the available readers:

Polygonal Data Readers

These readers produce a vtkPolyData as output.

- vtkBYUReader — read MOVIE.BYU files
- vtkMCubesReader — read binary marching cubes files
- vtkOBJReader — read Wavefront .obj files
- vtkPolyDataReader — read a VTK polygonal data files. See "VTK File Formats" on page 344 for information on VTK file formats.
- vtkPLYReader — read Stanford University PLY polygonal data files
- vtkSTLReader — read stereo-lithography files
- vtkUGFacetReader — read EDS Unigraphics facet files

Image and Volume Readers

These readers produce vtkImageData (or vtkStructuredPoints, now obsolete) as output.

- vtkBMPReader — read PC bitmap files
- vtkDEMReader — read digital elevation model files
- vtkJPEGReader — read JPEG files
- vtkImageReader — read various image files in various formats
- vtkPNMReader — read PNM (ppm, pgm, pbm) files
- vtkPNGReader — read Portable Network Graphics files
- vtkStructuredPointsReader — read VTK structured points data files. See "VTK File Formats" on page 344 for information on VTK file formats.
- vtkSLCReader — read SLC structured points files
- vtkTIFFReader — read files in TIFF format
- vtkVolumeReader — read image (volume) files
- vtkVolume16Reader — read 16-bit image (volume) files

Data Set Readers

These readers produce generic vtkDataSet as output. Typically, the reader requires an Update() invocation to determine what kind of concrete vtkDataSet subclass is created.

- vtkDataSetReader — read any VTK dataset file. See "VTK File Formats" on page 344 for information on VTK file formats.
- vtkGenericEnsightReader (and subclasses) — read Ensight files

Structured Grid Readers

These readers produce vtkStructuredGrid as output.

- vtkPLOT3DReader — read structured grid PLOT3D files
- vtkStructuredGridReader — read VTK structured grid data files. See "VTK File For-

mats" on page 344 for information on VTK file formats.

Rectilinear Grid Readers

These readers produce vtkRectilinearGrid as output.

- vtkRectilinearGridReader — read VTK rectilinear grid data files. See "VTK File Formats" on page 344 for information on VTK file formats.

Unstructured Data Readers

These readers produce vtkUnstructuredGrid as output.

- vtkUnstructuredGridReader — read VTK unstructured grid data files. See "VTK File Formats" on page 344 for information on VTK file formats.

8.2 Writers

Writers output datasets of various types to disk. Typically, a writer requires setting an input and specifying an output file name (sometimes multiple files) as shown in the following.

```
vtkPolyDataWriter writer
writer SetInput [aFilter GetOutput]
writer SetFileName "outFile.vtk"
writer SetFileTypeToBinary
writer Write
```

The VTK writers offer you the option of writing binary (SetFileTypeToBinary()) or ASCII (SetFileTypeToASCII()) files. (Note: binary files may not be transportable across computers. VTK takes care of swapping bytes, but does not handle transport between 64-bit and 32-bit computers.)

The following is a list of available writers.

Polygonal Data Writers

These writers require vtkPolyData as input.

- vtkBYUWriter — write MOVIE.BYU polygonal files
- vtkCGMWriter — write 2D polygonal data as a CGM file
- vtkIVWriter — write Inventor files
- vtkMCubesWriter — write triangles (polygonal data) in marching cubes format
- vtkPolyDataWriter — write a VTK vtkPolyData file. See "VTK File Formats" on page 344 for information on VTK file formats.
- vtkPLYWriter — write Stanford University PLY files
- vtkSTLWriter — write stereo-lithography files

Image and Volume Writers

These writers require vtkImageData as input.

- vtkBMPWriter — write PC bitmap files
- vtkJPEGWriter — write images in JPEG format
- vtkPNGWriter — write images in Portable Network Graphics format
- vtkPNMWriter — write images in PNM (ppm, pgm, pbm) format
- vtkPostScriptWriter — write images in PostScript format
- vtkStructuredPointsWriter — write a VTK vtkStructuredPoints file. See "VTK File Formats" on page 344 for information on VTK file formats.
- vtkTIFFWriter — write files in TIFF format

Structured Grid Writers

These writers require vtkStructuredGrid as input.

- vtkStructuredGridWriter — write a VTK vtkStructuredGrid file. See "VTK File Formats" on page 344 for information on VTK file formats.

Rectilinear Grid Writers

These writers require vtkRectilinearGrid as input.

- vtkRectilinearGridWriter — write a VTK vtkRectilinearGrid file. See "VTK File Formats" on page 344 for information on VTK file formats.

Unstructured Grid Writers

These writers require a vtkUnstructuredGrid as input.

- vtkUnstructuredGridWriter — write a VTK vtkUnstructuredGrid file. See "VTK File Formats" on page 344 for information on VTK file formats.

8.3 Importers

Importers accept data files that contain multiple datasets and/or the objects that compose a scene (i.e., lights, cameras, actors, properties, transformation matrices, etc.). Importers will either generate an instance of vtkRenderWindow and/or vtkRenderer, or you can specify them. If specified, the importer will create lights, cameras, actors, and so on, and place them into the specified instance(s). Otherwise, it will create instances of vtkRenderer and vtkRenderWindow, as necessary. The following example show how to use an instance of vtkImporter (in this case a vtk3DSImporter—imports 3D Studio files). This Tcl script was excerpted from `VTK/Examples/IO/Tcl/flamingo.tcl` (please see **Figure 8–1**).

Figure 8–1 Importing a file.

```
vtk3DSImporter importer
  importer ComputeNormalsOn
  importer SetFileName "$VTK_DATA_ROOT/Data/iflamigm.3ds"
  importer Read

set renWin [importer GetRenderWindow]
```

```
vtkRenderWindowInteractor iren
  iren SetRenderWindow $renWin
```

The *Visualization Toolkit* supports the following importers. (Note that the superclass vtkImporter is available for developing new subclasses.)

- vtk3DSImporter — import 3D Studio files
- vtkVRMLExporter — import VRML version 2.0 files

8.4 Exporters

Exporters output scenes in various formats. Instances of vtkExporter accept an instance of vtkRenderWindow, and write out the graphics objects supported by the exported format.

```
vtkRIBExporter exporter
    exporter SetRenderWindow renWin
    exporter SetFilePrefix "anExportedFile"
    exporter Write
```

The vtkRIBExporter shown above writes out multiple files in RenderMan format. The FilePrefix instance variable is used to write one or more files (geometry and texture map(s), if any).

The *Visualization Toolkit* supports the following exporters.

- vtkRIBExporter — export RenderMan files
- vtkIVExporter — export Inventor scene graph
- vtkOBJExporter — export a Wavefront `.obj` files
- vtkVRMLExporter — export VRML version 2.0 files

8.5 Creating Hardcopy

Creating informative pictures is a primary objective of VTK. And to document what you've done, saving pictures and series of pictures (i.e., animations) are important. This section describes various ways to create graphical output.

Saving Images

The simplest way to save images is to use the vtkWindowToImageFilter which grabs the output buffer of the render window and converts it into vtkImageData. This image can then be saved using one of the image writers (see "Writers" on page "Writers" on page 187 for more information). Here is an example

```
vtkWindowToImageFilter w2i
  w2i SetInput renWin

vtkJPEGWriter writer
  writer SetInput [w2i GetOutput]
  writer SetFileName "DelMesh.jpg"
  writer Write
```

Note that it is possible to use the off-screen mode of the render window when saving an image. The off-screen mode can be turned on by setting OffScreenRenderingOn() for the render window. Currently, this is supported on Windows and Mesa compiled with off-screen rendering support.

Saving Large (High-Resolution) Images

The images saved via screen capture or by saving the render window vary greatly in quality depending on the graphics hardware and screen resolution supported on your computer. To improve the quality of your images, there are two approaches that you can try. The first approach allows you to use the imaging pipeline to render pieces of your image and then combine them into a very high-resolution final image. We'll refer to this as piecemeal imaging. The second approach requires external software to perform high resolution rendering. We'll refer to this as the RenderMan solution.

Piecemeal Rendering. Often we want to save an image of resolution greater than the resolution of the computer hardware. For example, generating an image of 4000 x 4000 pixels is not possible (without tricks) on a computer system displaying 1280 x 1024 pixels. The *Visualization Toolkit* solves this problem with the class vtkRenderLargeImage. This class breaks up the rendering process into separate pieces, each piece containing just a portion of the final image. The pieces are assembled into a final image, which can be saved to file using one of the VTK image writers. Here's how it works (Tcl script taken from `VTK/Examples/Rendering/Tcl/RenderLargeImage.tcl`).

```
vtkRenderLargeImage renderLarge
  renderLarge SetInput ren
  renderLarge SetMagnification 4

vtkTIFFWriter writer
  writer SetInput [renderLarge GetOutput]
  writer SetFileName largeImage.tif
  writer Write
```

The Magnification instance variable (an integer value) controls how much to magnify the input renderer's current image. If the renderer's image size is (400,400) and the magnification factor is 5, the final image will be of resolution (2000,2000). In this example, the resulting image is written to a file with an instance of vtkPNMWriter. Of course, other writer types could be used.

RenderMan. RenderMan is a high-quality software rendering system currently sold by Pixar, the graphics animation house that created the famous *Toy Story* movie. RenderMan is a commercial package (retailing for about $5,000 at the time of this writing). Fortunately, there is at least one modestly priced (or free system if you're non-commercial) RenderMan compatible system that you can download and use: BMRT (Blue Moon Ray Tracer). BMRT is slower than RenderMan, but it also offers several features that RenderMan does not.

In an earlier section ("Exporters" on page 190) we saw how to export a RenderMan `.rib` file (and associated textures). You can adjust the size of the image RenderMan produces using the SetSize() method in the RIBExporter. This method adds a line to the rib file that causes RenderMan (or RenderMan compatible system such as BMRT) to create an output TIFF image of size (`xres`, `yres`) pixels.

8.6 Creating Animations (Using Splines)

Animations are simple to create in principle. A sequence of images is saved, using any of the methods described previously (see "Saving Images" on page 191), and the sequences are assembled into a final animation. The challenge of creating a good animation is controlling the motion or change in properties of objects and images to generate smooth, well-timed, and synchronized sequences. This is particularly hard when you combine images with sound.

Figure 8–2 A spline of a set of random points in 3D.

There are a couple of things you can do to simplify your task. First, if using sound, start by developing the sound track, and then plan and synchronize the animation around the track. You'll want a script detailing the start and duration of each sequence along with its relationship to the sound. Work hard on the timing of each sequence, make sure it's not too long or short, and that events don't happen too quickly or slowly. You might consider making sequences multiples of 30 frames (30 frames per second on video), or some multiple of your music beat. This greatly facilitates final editing.

VTK offers limited support for creating animations. Besides the image saving tools discussed previously, VTK supports splines for interpolating motion and other parameters. There are two types of interpolating 1D splines: vtkCardinalSpline and vtkKochanekSpline, both of subclass vtkSpline. (These splines differ in their basis functions. vtkKochenekSpline offers more control over the shape of the spline.) To define a spline, you create a sequence of independent/dependent variable pairs, and then optionally set end conditions and/or other spline parameters. To interpolate between points, the Evaluate() method is invoked as shown in the following example (taken from `VTK/Examples/Rendering/Tcl/CSpline.tcl` — see **Figure 8–2**).

```
set numberOfInputPoints 10

vtkCardinalSpline aSplineX
vtkCardinalSpline aSplineY
vtkCardinalSpline aSplineZ

vtkPoints inputPoints
for {set i 0} {$i < $numberOfInputPoints} {incr i 1} {
```

```
    set x  [math Random 0 1]
    set y  [math Random 0 1]
    set z  [math Random 0 1]
    aSplineX AddPoint $i $x
    aSplineY AddPoint $i $y
    aSplineZ AddPoint $i $z
    inputPoints InsertPoint $i $x $y $z
}
...more stuff...
# Interpolate x, y and z by using the three spline filters and
# create new points
for {set i 0} {$i< $numberOfOutputPoints} {incr i 1} {
    set t [expr ( $numberOfInputPoints - 1.0 ) / ( $numberOfOutputPoints -
1.0 ) * $i]
    points InsertPoint $i [aSplineX Evaluate $t] [aSplineY Evaluate $t]
[aSplineZ Evaluate $t]
}
```

Three separate splines are created, one each for the *x*, *y*, and *z* axes. Using these splines, a polyline is created by evaluating them at points along the range of the independent variable (which is the point id).

If you are using Tcl and would like to see an applications that employs splines to create animations, the example in `VTK/Examples/Rendering/Tcl/keyBottle.tcl` uses the Tcl script `VTK/Examples/Rendering/Tcl/KeyFrame.tcl` to create an animation sequence of camera motion (variations in camera azimuth).

8.7 Working With Field Data

Many times data is organized in a form different from that found in VTK. For example, your data may be tabular, or possibly even higher-dimensional. And sometimes you'd like to be able to rearrange your data, assigning some data as scalars, some as point coordinates, and some as other attribute data. In such situations VTK's field data, and the filters that allow you to manipulate field data, are essential.

To introduce this topic a concrete example is useful. In the previous chapter (see "Gaussian Splatting" on page 176) we saw an example that required writing custom code to read a tabular data file, extracting out specified data to form points and scalars (look at the function `ReadFinancialData()` found in `VTK/Examples/Modelling/Cxx/`

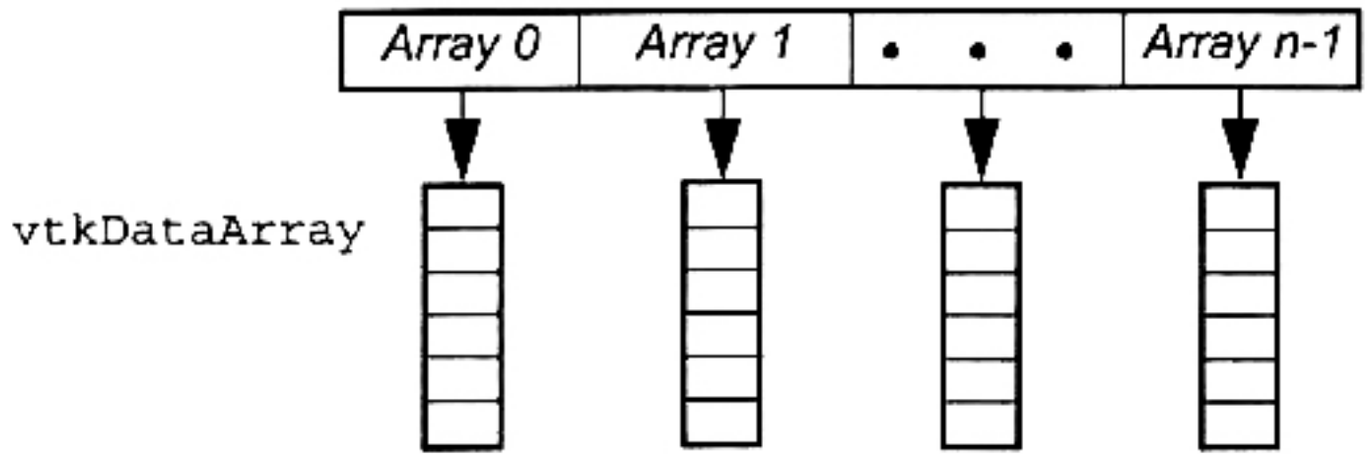

Figure 8–3 Structure of field data—an array of arrays. Each array may be of a different native data type and may have one or more components.

`finance.cxx`). While this works fine for this example, it does require a lot of work and is not very flexible. In the following example, we'll do the same thing using field data.

The data is in the following tabular format:

```
NUMBER_POINTS 3188
TIME_LATE
 29.14    0.00    0.00   11.71    0.00    0.00    0.00    0.00
  0.00   29.14    0.00    0.00    0.00    0.00    0.00    0.00
....
MONTHLY_PAYMENT
  7.26    5.27    8.01   16.84    8.21   15.75   10.62   15.47
  5.63    9.50   15.29   15.65   11.51   11.21   10.33   10.78
....
```

which repeats for the fields the time late in paying the loan (`TIME_LATE`); the monthly payment of the loan (`MONTHLY_PAYMENT`); the principal left on the loan (`UNPAID_PRINCIPAL`); the original amount of the loan (`LOAN_AMOUNT`); the interest rate on the loan (`INTEREST_RATE`); and the monthly income of the loanee (`MONTHLY_INCOME`). These six fields form a matrix of 3188 rows and 6 columns.

We start by parsing the data file. The class vtkProgrammableDataObjectSource is useful for defining special input methods without having to modify VTK. All we need to do is to define a function that parses the file and stuffs them into a VTK data object (recall that vtkDataObject is the most general form of data representation). Reading the data is the most challenging part of this example, which can be found in `VTK/Examples/DataManipulation/Tcl/FinancialField.tcl`.

```
set xAxis INTEREST_RATE
set yAxis MONTHLY_PAYMENT
```

```
set zAxis MONTHLY_INCOME
set scalar TIME_LATE

# Parse an ascii file and manually create a field. Then construct a
# dataset from the field.
vtkProgrammableDataObjectSource dos
    dos SetExecuteMethod parseFile

proc parseFile {} {
   global VTK_DATA_ROOT

   # Use Tcl to read an ascii file
   set file [open "$VTK_DATA_ROOT/Data/financial.txt" r]
   set line [gets $file]
   scan $line "%*s %d" numPts
   set numLines [expr (($numPts - 1) / 8) + 1 ]

   # Get the data object's field data and allocate
   # room for 4 fields
   set fieldData [[dos GetOutput] GetFieldData]
   $fieldData AllocateArrays 4

   # read TIME_LATE - dependent variable
   # search the file until an array called TIME_LATE is found
   while { [gets $file arrayName] == 0 } {}
   # Create the corresponding float array
   vtkFloatArray timeLate
   timeLate SetName TIME_LATE
   # Read the values
   for {set i 0} {$i < $numLines} {incr i} {
      set line [gets $file]
      set m [scan $line "%f %f %f %f %f %f %f %f" \
    v(0) v(1) v(2) v(3) v(4) v(5) v(6) v(7)]
      for {set j 0} {$j < $m} {incr j} {timeLate InsertNextValue $v($j)}
   }
   # Add the array
   $fieldData AddArray timeLate

   # MONTHLY_PAYMENT - independent variable
   while { [gets $file arrayName] == 0 } {}
   vtkFloatArray monthlyPayment
   monthlyPayment SetName MONTHLY_PAYMENT
   for {set i 0} {$i < $numLines} {incr i} {
      set line [gets $file]
      set m [scan $line "%f %f %f %f %f %f %f %f" \
    v(0) v(1) v(2) v(3) v(4) v(5) v(6) v(7)]
```

```
      for {set j 0} {$j < $m} {incr j} {monthlyPayment InsertNextValue
$v($j)}
   }
   $fieldData AddArray monthlyPayment

   # UNPAID_PRINCIPLE - skip
   while { [gets $file arrayName] == 0 } {}
   for {set i 0} {$i < $numLines} {incr i} {
      set line [gets $file]
   }

   # LOAN_AMOUNT - skip
   while { [gets $file arrayName] == 0 } {}
   for {set i 0} {$i < $numLines} {incr i} {
      set line [gets $file]
   }

   # INTEREST_RATE - independnet variable
   while { [gets $file arrayName] == 0 } {}    .
   vtkFloatArray interestRate
   interestRate SetName INTEREST_RATE
   for {set i 0} {$i < $numLines} {incr i} {
      set line [gets $file]
      set m [scan $line "%f %f %f %f %f %f %f %f" \
    v(0) v(1) v(2) v(3) v(4) v(5) v(6) v(7)]
     for {set j 0} {$j < $m} {incr j} {interestRate InsertNextValue $v($j)}
   }
   $fieldData AddArray interestRate

   # MONTHLY_INCOME - independent variable
   while { [gets $file arrayName] == 0 } {}
   vtkIntArray monthlyIncome
   monthlyIncome SetName MONTHLY_INCOME
   for {set i 0} {$i < $numLines} {incr i} {
      set line [gets $file]
      set m [scan $line "%d %d %d %d %d %d %d %d" \
    v(0) v(1) v(2) v(3) v(4) v(5) v(6) v(7)]
     for {set j 0} {$j < $m} {incr j} {monthlyIncome InsertNextValue $v($j)}
   }
   $fieldData AddArray  monthlyIncome
}
```

Now that we've read the data, we have to rearrange the field data contained by the output vtkDataObject into a form suitable for processing by the visualization pipeline (i.e., the vtkGaussianSplatter). This means creating a subclass of vtkDataSet, since vtkGaussian-

Splatter takes an instance of vtkDataSet as input. There are two steps required. First, the filter vtkDataObjectToDataSetFilter is used to convert the vtkDataObject to type vtk-DataSet. Then, vtkRearrangeFields and vtkAssignAttribute are used to move a field from the vtkDataObject to the vtkPointData of the newly created vtkDataSet and label it as the active scalar field.

```
vtkDataObjectToDataSetFilter do2ds
    do2ds SetInput [dos GetOutput]
    do2ds SetDataSetTypeToPolyData
    do2ds DefaultNormalizeOn
    do2ds SetPointComponent 0 $xAxis 0
    do2ds SetPointComponent 1 $yAxis 0
    do2ds SetPointComponent 2 $zAxis 0

vtkRearrangeFields rf
    rf SetInput [do2ds GetOutput]
    rf AddOperation MOVE $scalar DATA_OBJECT POINT_DATA

vtkAssignAttribute aa
    aa SetInput [rf GetOutput]
    aa Assign $scalar SCALARS POINT_DATA
    aa Update
```

There are several important notes here.

1. All filters pass their input vtkDataObject through to their output unless instructed otherwise (or unless they modify vtkDataObject). We will take advantage of this in the downstream filters.
2. vtkDataObjectToDataSetFilter is set up to create an instance of vtkPolyData as its output, with the three components of the field data serving as x, y, and z coordinates. In this case we use vtkPolyData because the data is unstructured and consists only of points.
3. The field values are normalized between (0,1) because the axes' ranges are different enough that we create a better visualization by filling the entire space with data.
4. The filter vtkRearrangeFields copies/moves fields between vtkDataObject, vtkPoint-Data and vtkCellData. In this example, an operation to move the field called $scalar from the data object of the input to the point data of the output is assigned.
5. The filter vtkAssignAttribute labels fields as attributes. In this example, the field called $scalar (that is in the point data) is labeled as the active scalar field.

The Set___Component() methods are the key methods of vtkDataObjectToDataSetFilter. These methods refer to the data arrays in the field data by name and by component number. (Recall that a data array may have more than one component.) It is also possible to indicate a (min,max) tuple range from the data array, and to perform normalization. However, make sure that the number of tuples extracted matches the number of items in the dataset structure (e.g., the number of points or cells).

There are several related classes that do similar operations. These classes can be used to rearrange data arbitrarily to and from field data, into datasets, and into attribute data. These filters include:

- vtkDataObjectToDataSetFilter — Transform field data contained in a vtkDataObject to a subclass of vtkDataSet.
- vtkDataSetToDataObjectFilter — Transform vtkDataSet into vtkFieldData contained in a vtkDataObject.
- vtkRearrangeFields — Move/copy fields between field data, point data and cell data.
- vtkAssignAttribute — Labels a field as an attribute.
- vtkMergeFields — Merge multiple fields into one.
- vtkSplitField — Split a field into multiple single component fields.
- vtkDataObjectReader — Read a VTK formatted field data file. (See "Dataset Attribute Format" on page 351 for a description of the field data format.)
- vtkDataObjectWriter — Write a VTK formatted field data file. (See "Dataset Attribute Format" on page 351 for a description of the field data format.)
- vtkProgrammableDataObjectSource — Define a method to read data of arbitrary form and represent it as field data (i.e., place it in a vtkDataObject).