Chapter 5

Visualization Techniques

Some basic tools to render and interact with data were presented in the previous chapter. In this chapter we'll show you a variety of visualization techniques. These techniques (implemented as filters) are organized according to the type of data they operate on. Some filters are general and can be applied to any type of data—those filters that accept input of class vtkDataSet (or any subclass). Many filters are more specialized to the type of input they accept (e.g., vtkPolyData). There is one class of filters—those that accept input of type vtkImageData (or its obsolete subclass vtkStructuredPoints)—that are not addressed in this chapter. Instead, filters of this type are described in the next chapter ("Visualization Techniques" on page 83).

Please keep two things to keep in mind while you read this chapter. First, filters generate a variety of output types, and the output type is not necessarily the same as the input type. Second, filters are used in combination to create complex data processing pipelines. Often there are patterns of usage, or common combinations of filters, that are used. In the following examples you may wish to note these combinations.

5.1 Visualizing vtkDataSet (and Subclasses)

In this section, we'll show you how to perform some common visualization operations on data objects of type vtkDataSet. Recall that vtkDataSet is the superclass for all concrete types of visualization data (see **Figure 3–2**). Therefore, the methods described here are applicable to all of the various data types. (In other words, all filters taking vtkDataSet as input will also accept vtkPolyData, vtkImageData, vtkStructuredGrid, vtkRectilinearGrid, and vtkUnstructuredGrid.)

Working With Data Attributes

Data attributes are information associated with the structure of the dataset (as described in "The Visualization Model" on page 23). In VTK, attribute data is associated with points (point attribute data) and cells (cell attribute data). Attribute data, along with the dataset structure, are processed by the many VTK filters to generate new structures and attributes.

A general introduction to attribute data is beyond the scope of this section, but a simple example will demonstrate the basic ideas. (You may wish to refer to "Interface To Field and Attribute Data" on page 270 for more information and **Figure 11–1**.)

Data attributes are simply vtkDataArrays which may be labeled as being one of scalars, vectors, tensors, normals, or texture coordinates. The data attributes may be associated with the points or cells of a vtkDataSet. Every vtkDataArray associated with a vtkDataSet is a concrete subclass of vtkDataArray, such as vtkFloatArray or vtkIntArray. These data arrays can be thought of as contiguous, linear blocks of memory of the named native type. Within this linear block, the data array is thought to consist of subarrays or "tuples." Creating attribute data means instantiating a data array of desired type, specifying the tuple size, inserting data, and associating it with a dataset, as shown in the following Tcl script. The association may have the side effect of labeling the data as scalars, vectors, tensors, texture coordinates, or normals. For example:

```
vtkFloatArray scalars
    scalars InsertTuple1 0 1.0
    scalars InsertTuple1 1 1.2
    ...etc...

vtkDoubleArray vectors
    vectors SetNumberOfComponents 3
    vectors InsertTuple3 0 0.0 0.0 1.0
    vectors InsertTuple3 1 1.2 0.3 1.1
    ...etc...

vtkIntArray justAnArray
    justAnArray SetNumberOfComponents 2
    justAnArray SetNumberOfTuples $numberOfPoints
    justAnArray SetName "Solution Attributes"
    justAnArray SetTuple2 0 1 2
    justAnArray SetTuple2 1 3 4
    ...etc...

vtkPolyData polyData;#A concrete type of vtkDataSet
    [polyData GetPointData] SetScalars scalars
    [polyData GetCellData] SetVectors vectors
    [polyData GetPointData] AddArray justAnArray
```

Here we create three arrays of types `float`, `double`, and `int`. The first array (`scalars`) is instantiated and by default has a tuple size of one. The method InsertTuple1() is used to place data into the array (all methods named Insert___() allocate memory as necessary to

hold data). The next data array (`vectors`) is created with a tuple size of three, because vectors are defined as having three components, and InsertTuple3 is used to add data to the array. Finally, we create a general array of tuple size two, and allocate memory using SetNumberOfTuples(). We then use SetTuple2() to add data; this method assumes that memory has been allocated and is therefore faster than the similar Insert__() methods. Notice that the labelling of what is a scalar, vector, etc. occurs when we associate the data arrays with the point data or cell data of the dataset (using the methods SetScalars() and SetVectors()). Please remember that the number of point attributes (e.g., number of scalars in this example) must equal the number of points in the dataset, and the number of cell attributes (e.g., number of vectors) must match the number of cells in the dataset.

Similarly, to access attribute data, use these methods

```
set scalars [[polyData GetPointData] GetScalars]
set vectors [[polyData GetCellData] GetVectors]
```

You'll find that many of the filters work with attribute data specifically. For example, vtkElevationFilter generates scalar values based on their elevation in a specified direction. Other filters work with the structure of the dataset, and generally ignore or pass the attribute data through the filter (e.g., vtkDecimatePro). And finally, some filters work with (portions of) the attribute data and the structure to generate their output. vtkMarchingCubes is one example. It uses the input scalars in combination with the dataset structure to generate contour primitives (i.e., triangles, lines or points). Other types of attribute data, such as vectors, are interpolated during the contouring process and sent to the output of the filter.

Another important issue regarding attribute data is that some filters will process only one type of attribute (point data versus cell data), ignoring or passing to their output the other attribute data type. You may find that your input data is of one attribute type and you want to process it with a filter that will not handle that type, or you simply want to convert from one attribute type to another. There are two filters that can help you with this: vtkPointDataToCellData and vtkCellDataToPointData, which convert to and from point and cell data attributes. Here's an example of their use (from the Tcl scrip `VTK/Examples/DataManipulation/Tcl/pointToCellData.tcl`).

```
vtkUnstructuredGridReader reader
    reader SetFileName "$VTK_DATA_ROOT/Data/blow.vtk"
    reader SetScalarsName "thickness9"
    reader SetVectorsName "displacement9"
```

```
vtkPointDataToCellData p2c
    p2c SetInput [reader GetOutput]
    p2c PassPointDataOn
vtkWarpVector warp
    warp SetInput [p2c GetUnstructuredGridOutput]
vtkThreshold thresh
    thresh SetInput [warp GetOutput]
    thresh ThresholdBetween 0.25 0.75
    thresh SetAttributeModeToUseCellData
```

This example is interesting because it demonstrates the conversion between attribute data types (vtkPointDataToCellData), and the use of a filter that can process either cell data or point data (vtkThreshold). The method PassPointDataOn() indicates to vtkPointDataToCellData to create cell data and also pass to its output the input point data. The method SetAttributeModeToUseCellData() configures the vtkThreshold filter to use the cell data to perform the thresholding operation.

The conversion between point and cell data and vice versa is performed using an averaging algorithm. Point data is converted to cell data by averaging the values of the point data associated with the points used by a given cell. Cell data is converted to point data by averaging the cell data associated with the cells that use a given point.

Color Mapping

Probably the single most used visualization technique is coloring objects via scalar value, or color mapping. The ideas behind this technique is simple: scalar values are mapped through a lookup table to obtain a color, and the color is applied during rendering to modify the appearance of points or cells. Before proceeding with this section, make sure that you understand how to control the color of an actor (see "Actor Color" on page 59).

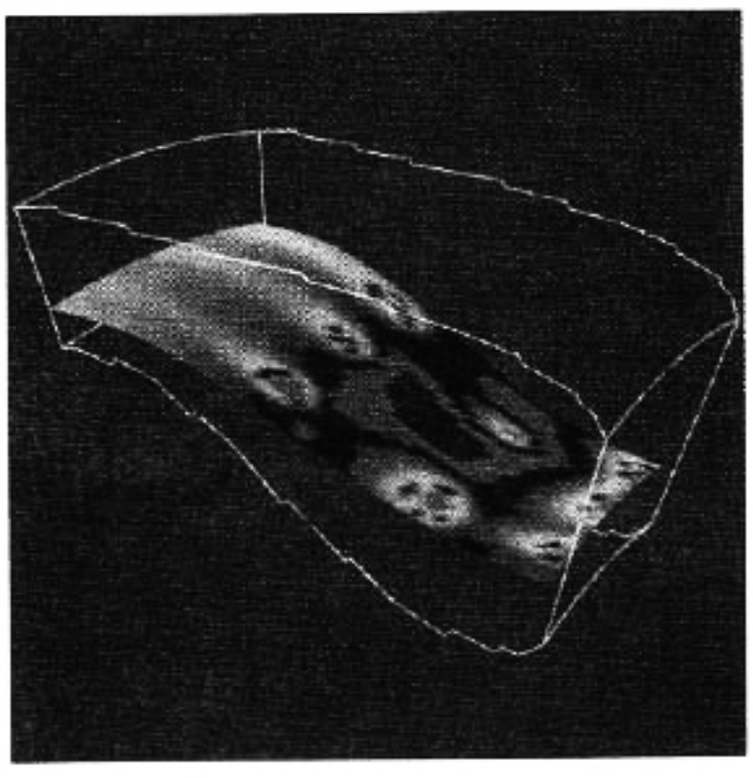

Figure 5–1 Color mapping.

In VTK, color mapping is typically controlled by scalars, which we assume you've created or read from a data file, and the lookup table, which is used by instances of vtkMapper to perform color mapping. It is also possible to use any data array to perform the coloring, just use the method

ColorByArrayComponent(). If not specified, a default lookup table is created by the mapper, but you can create your own (taken from `VTK/Examples/Rendering/Tcl/rainbow.tcl`—see **Figure 5–1**).

```
vtkLookupTable lut
   lut SetNumberOfColors 64
   lut SetHueRange 0.0 0.667
   lut Build
    for {set i 0} {$i<16} {incr i 1} {
        eval lut SetTableValue [expr $i*16] $red 1
        eval lut SetTableValue [expr $i*16+1] $green 1
        eval lut SetTableValue [expr $i*16+2] $blue 1
        eval lut SetTableValue [expr $i*16+3] $black 1
    }
vtkPolyDataMapper planeMapper
    planeMapper SetLookupTable lut
    planeMapper SetInput [plane GetOutput]
    planeMapper SetScalarRange 0.197813 0.710419
vtkActor planeActor
    planeActor SetMapper planeMapper
```

Lookup tables can be manipulated in two different ways, as this example illustrates. First, you can specify a HSVA (Hue-Saturation-Value-Alpha transparency) ramp that is used to generate the colors in the table using linear interpolation in HSVA space (the Build() method actually generates the table). Second, you can manually insert colors at specific locations in the table. Note that the number of colors in the table can be set. Also, you can generate the table with the HSVA ramp, and then replace colors in the table with the SetTableValue() method.

The mapper's SetScalarRange() method controls how scalars are mapped into the table. Scalar values greater than the maximum value are clamped to the maximum value. Scalar values less than the minimum value are clamped to the minimum value. Using the scalar range let's you "expand" a region of the scalar data by mapping more colors to it.

Sometimes the scalar data is actually color, and does not need to be mapped through a lookup table. The mapper provides several methods to control the mapping behavior.

- SetColorModeToDefault() invokes the default mapper behavior. The default behavior treats scalars of data type `unsigned char` as colors and performs no mapping; all other types of scalars are mapped through the lookup table.
- SetColorModeToMapScalars() maps all scalars through the lookup table, regardless

of type. If the scalar has more than one component per tuple, then the scalars zeroth component is used perform the mapping.

Another important feature of vtkMapper is controlling which attribute data (i.e., point or cell scalars, or a general data array) is used to color objects. The following methods let you control this behavior. Note that these methods give strikingly different results: point attribute data is interpolated across rendering primitives during the rendering process, whereas cell attribute data colors the cell a constant value.

- SetScalarModeToDefault() invokes the default mapper behavior. The default behavior uses point scalars to color objects unless they are not available, in which case cell scalars are used, if they are available.
- SetScalarModeToUsePointData() always uses point data to color objects. If no point scalar data is available, then the object color is not affected by scalar data.
- SetScalarModeToUseCellData() always uses cell data to color objects. If no cell scalar data is available, then the object color is not affected by scalar data.
- SetScalarModeToUsePointFieldData() indicates that neither the point or cell scalars are to be used, but rather a data array found in the point attribute data. This method should be used in conjunction with ColorByArrayComponent() to specify the data array and component to use as the scalar.
- SetScalarModeToUseCellFieldData() indicates that neither the point or cell scalars are to be used, but rather a data array found in the cell field data. This method should be used in conjunction with ColorByArrayComponent() to specify the data array and component to use as the scalar.

Normally the default behavior works well, unless both cell and point scalar data is available. In this case, you will probably want to explicitly indicate whether to use point scalars or cell scalars to color your object.

Contouring

Another common visualization technique is generating contours. Contours are lines or surfaces of constant scalar value. In VTK, the filter vtkContourFilter is used to perform contouring as shown in the following Tcl example from `VTK/Examples/VisualizationAlgorithms/VisQuad.tcl`—refer to **Figure 5–2**.

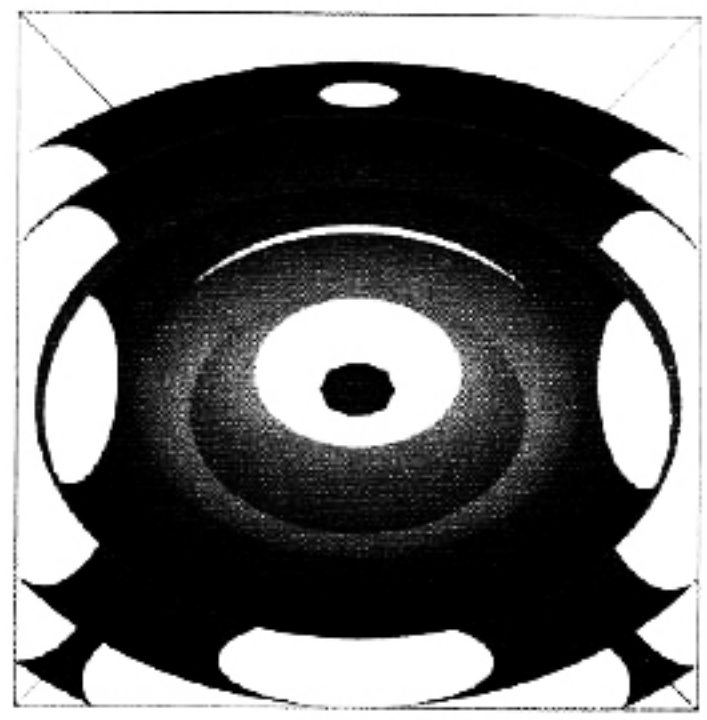

Figure 5–2 Generating contours.

```
# Create 5 surfaces in range specified
vtkContourFilter contours
  contours SetInput [sample GetOutput]
  contours GenerateValues 5 0.0 1.2
vtkPolyDataMapper contMapper
  contMapper SetInput [contours GetOutput]
  contMapper SetScalarRange 0.0 1.2
vtkActor contActor
  contActor SetMapper contMapper
```

You can specify contour values in two ways. The simplest way is to use the SetValue() method to specify the contour number and its value (multiple values can be specified)

```
contours SetValue 0 0.5
```

The earlier example demonstrated the second way: the GenerateValues(). With this method, you specify the scalar range and the number of contours to be generated in the range (end values inclusive).

Note that there are several objects in VTK that perform contouring specialized to a particular dataset type (and are faster). Examples include vtkMarchingCubes, vtkMarchingSquares, and so on. You do not need to instantiate these directly if you use vtkContourFilter; the filter will select the best contouring function for your dataset type automatically.

Glyphing

Glyphing is a visualization technique that represents data by using symbols, or glyphs (**Figure 5–3**). The symbols can be simple or complex, ranging from oriented cones to show vector data, to complex, multivariate glyphs such as Chernoff faces (symbolic representations of the human face whose expression is controlled by data values). In VTK, the vtkGlyph3D class allows you to create glyphs that can be scaled, colored, and oriented along a direction. The glyphs are copied at each point of the input dataset. The glyph itself is defined by the second input to the filter (which is of type vtkPolyData), the Source instance variable. The following script demonstrates the use of vtkGlyph3D (the Tcl script is taken from `VTK/Examples/VisualizationAlgorithms/Tcl/spikeF.tcl`).

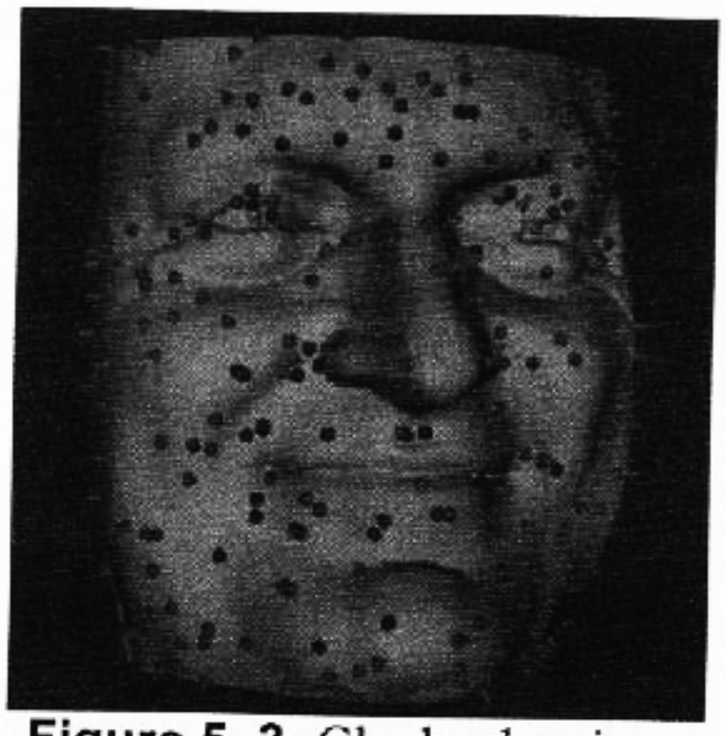

Figure 5–3 Glyphs showing surface normals.

```
vtkPolyDataReader fran
    fran SetFileName "$VTK_DATA_ROOT/Data/fran_cut.vtk"
vtkPolyDataNormals normals
    normals SetInput [fran GetOutput]
    normals FlipNormalsOn
vtkPolyDataMapper franMapper
    franMapper SetInput [normals GetOutput]
vtkActor franActor
    franActor SetMapper franMapper
    eval [franActor GetProperty] SetColor 1.0 0.49 0.25

vtkMaskPoints ptMask
    ptMask SetInput [normals GetOutput]
    ptMask SetOnRatio 10
    ptMask RandomModeOn

# In this case we are using a cone as a glyph. We transform the cone so
# its base is at 0,0,0. This is the point where glyph rotation occurs.
vtkConeSource cone
    cone SetResolution 6
vtkTransform transform
    transform Translate 0.5 0.0 0.0
vtkTransformPolyDataFilter transformF
    transformF SetInput [cone GetOutput]
```

```
    transformF SetTransform transform

vtkGlyph3D glyph
    glyph SetInput [ptMask GetOutput]
    glyph SetSource [transformF GetOutput]
    glyph SetVectorModeToUseNormal
    glyph SetScaleModeToScaleByVector
    glyph SetScaleFactor 0.004
vtkPolyDataMapper spikeMapper
    spikeMapper SetInput [glyph GetOutput]
vtkActor spikeActor
    spikeActor SetMapper spikeMapper
    eval [spikeActor GetProperty] SetColor 0.0 0.79 0.34
```

The purpose of the script is to indicate the direction of surface normals using small, oriented cones. An input dataset (from a Cyberware laser digitizing system) is read and displayed. Next, the filter vtkMaskPoints is used to subsample the points (and associated point attribute data) from the Cyberware data. This serves as the input to the vtkGlyph3D instance. A vtkConeSource is used as the Source for the glyph instance. Notice that the cone is translated (with vtkTransformPolyDataFilter) so that its base is on the origin (0,0,0) (since vtkGlyph3D rotates the source object around the origin).

The vtkGlyph3D object `glyph` is configured to use the point attribute normals as the orientation vector. (Alternatively, use SetVectorModeToUseVector() to use the vector data instead of the normals.) It also scales the cones by the magnitude of the vector value there, with the given scale factor. (You can scale the glyphs by scalar data or turn data scaling off with the SetScaleModeToScaleByScalar() and SetScaleModeToDataScalingOff().)

It is also possible to color the glyphs with scalar or vector data, or by the scale factor. You can also create a table of glyphs, and use scalar or vector data to index into the table. Refer to the man pages for more information.

Streamlines

A streamline can be thought of as the path that a massless particle takes in a vector field (e.g., velocity field). Streamlines are used to convey the structure of a vector field. Usually multiple streamlines are created to explore interesting features in the field (**Figure 5–4**). Streamlines are computed via numerical integration (integrating the product of velocity times Δt), and are therefore only approximations to the actual streamlines.

Creating a streamlines requires specifying a starting point (or points, if multiple streamlines), an integration direction (along the flow, or opposite the flow direction, or in both directions), and other parameters to control its propagation. The following script shows how to create a single streamline. The streamline is wrapped with a tube whose radius is proportional to the inverse of velocity magnitude. This indicates where the flow is slow (fat tube) and where it is fast (thin tube). This Tcl script is extracted from `VTK/Examples/VisualizationAlgorithms/Tcl/officeTube.tcl`.

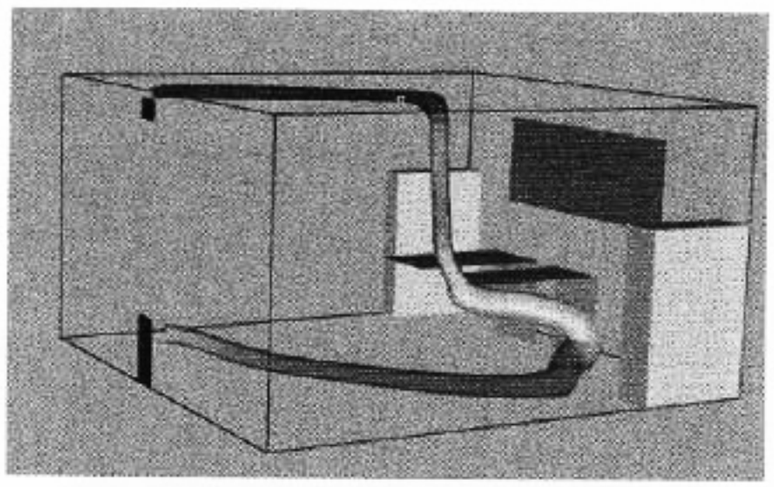

Figure 5–4 Streamline wrapped with a tube.

```
# Read structured grid data
vtkStructuredGridReader reader
    reader SetFileName "$VTK_DATA_ROOT/Data/office.binary.vtk"
    reader Update;#force a read to occur

# Create source for streamtubes
vtkRungeKutta4 integ
vtkStreamLine streamer
    streamer SetInput [reader GetOutput]
    streamer SetStartPosition 0.1 2.1 0.5
    streamer SetMaximumPropagationTime 500
    streamer SetStepLength 0.5
    streamer SetIntegrationDirectionToIntegrateBothDirections
    streamer SetIntegrator integ
vtkTubeFilter streamTube
    streamTube SetInput [streamer GetOutput]
    streamTube SetRadius 0.05
    streamTube SetNumberOfSides 6
    streamTube SetVaryRadiusToVaryRadiusByVector
vtkPolyDataMapper mapStreamTube
    mapStreamTube SetInput [streamTube GetOutput]
    eval mapStreamTube SetScalarRange \
       [[[[reader GetOutput] GetPointData] GetScalars] \
                    GetRange];#this is why we did an Update
vtkActor streamTubeActor
    streamTubeActor SetMapper mapStreamTube
```

In this example we've selected a starting point by specifying the world coordinate (0.1,2.1,0.5). It's also possible to specify a starting location by using cellId, cell subId, and parametric coordinates. The MaximumPropagationTime instance variable controls the

maximum length of the streamline (measured in units of time), and StepLength controls the size of the output line segments that make up the streamline (which is represented as a polyline). If you want greater accuracy (at the cost of more computation time) set the IntegrationStepLength instance variable to a smaller value. IntegrationStepLength is a number between (0,1) that indicates the step length as a fraction of the current cell size based on the diagonal length of the cells bounding box. Accuracy improvements can also be realized by choosing a different subclass of vtkInitialValueProblemSolver such as vtkRungeKutta4. (By default, the streamer classes use vtkRungeKutta2 to perform the numerical integration.)

You can also control the direction of integration with the methods:

- SetIntegrationDirectionToIntegrateForward()
- SetIntegrationDirectionToIntegrateBackward()
- SetIntegrationDirectionToIntegrateBothDirections()

Lines are often difficult to see and create useful images from. In this example we wrap the lines with a tube filter. The tube filter is configured to vary the radius of the tube inversely proportional to the velocity magnitude (i.e., a flux preserving relationship if the flow field is incompressible). The SetVaryRadiusToVaryRadiusByVector() enables this. You can also vary the radius by scalar value (SetVaryRadiusToVaryRadiusByScalar()) or turn off variable radius (SetVaryRadiusToVaryRadiusOff()).

As suggested earlier, we often wish to generate many streamlines simultaneously. One way to do this is to use the SetSource() method to specify an instance of vtkDataSet whose points are used to seed streamlines. Here's an example of its use (from `VTK/Examples/VisualizationAlgorithms/Tcl/officeTubes.tcl`).

```
vtkPointSource seeds
    seeds SetRadius 0.15
    eval seeds SetCenter 0.1 2.1 0.5
    seeds SetNumberOfPoints 6
vtkRungeKutta4 integ
vtkStreamLine streamer
    streamer SetInput [reader GetOutput]
    streamer SetSource [seeds GetOutput]
    streamer SetMaximumPropagationTime 500
    streamer SetStepLength 0.5
    streamer SetIntegrationStepLength 0.05
    streamer SetIntegrationDirectionToIntegrateBothDirections
```

```
    streamer SetIntegrator integ

vtkTubeFilter streamTube
    streamTube SetInput [streamer GetOutput]
    streamTube SetRadius 0.02
    streamTube SetNumberOfSides 12
    streamTube SetVaryRadiusToVaryRadiusByVector
vtkPolyDataMapper mapStreamTube
    mapStreamTube SetInput [streamTube GetOutput]
    eval mapStreamTube SetScalarRange \
       [[[[reader GetOutput] GetPointData] GetScalars] GetRange]
vtkActor streamTubeActor
    streamTubeActor SetMapper mapStreamTube
    [streamTubeActor GetProperty] BackfaceCullingOn
```

Notice that the example uses the source object vtkPointSource to create a spherical cloud of points, which are then set as the source to `streamers`. For every point (inside the input dataset) a streamline will be computed.

Stream Surfaces

Advanced users may want to use VTK's stream surface capability. Stream surfaces are generated in two parts. First, a rake or series of ordered points are used to generate a series of streamlines. Then, vtkRuledSurfaceFilter is used to create a surface from the streamlines. It is very important that the points (and hence streamlines) are ordered carefully because the vtkRuledSurfaceFilter assumes that the lines lie next to one another, and are within a specified distance (DistanceFactor) of the neighbor to the left and right.
Otherwise, the surface tears or you can obtain poor results. The following script (taken from `VTK/Examples/VisualizationAlgorithms/Tcl/streamSurface.tcl` and shown in **Figure 5–5**.

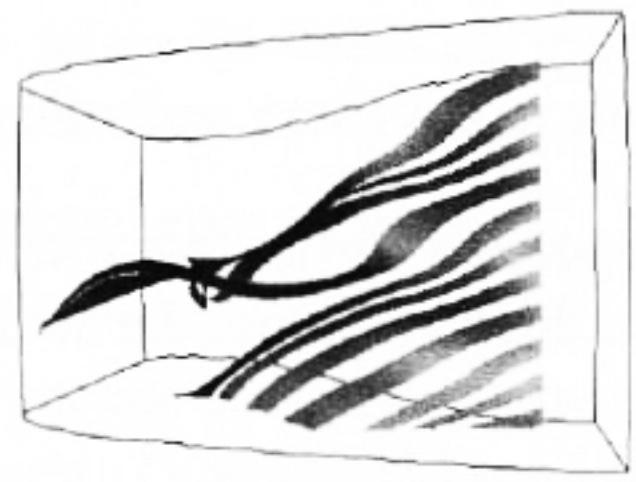

Figure 5–5 Stream surface.

```
vtkLineSource rake
  rake SetPoint1 15 -5 32
  rake SetPoint2 15 5 32
  rake SetResolution 21
vtkPolyDataMapper rakeMapper
  rakeMapper SetInput [rake GetOutput]
```

```
vtkActor rakeActor
  rakeActor SetMapper rakeMapper

vtkRungeKutta4 integ
vtkStreamLine sl
  sl SetInput [pl3d GetOutput]
  sl SetSource [rake GetOutput]
  sl SetIntegrator integ
  sl SetMaximumPropagationTime 0.1
  sl SetIntegrationStepLength 0.1
  sl SetIntegrationDirectionToBackward
  sl SetStepLength 0.001

vtkRuledSurfaceFilter scalarSurface
  scalarSurface SetInput [sl GetOutput]
  scalarSurface SetOffset 0
  scalarSurface SetOnRatio 2
  scalarSurface PassLinesOn
  scalarSurface SetRuledModeToPointWalk
  scalarSurface SetDistanceFactor 30
vtkPolyDataMapper mapper
  mapper SetInput [scalarSurface GetOutput]
  eval mapper SetScalarRange [[pl3d GetOutput] GetScalarRange]
vtkActor actor
  actor SetMapper mapper
```

A nice feature of the vtkRuledSurfaceFilter is the ability to turn off strips, if multiple lines are provided as input to the filter (the method SetOnRatio()). This helps understand the structure of the surface.

Cutting

Cutting, or slicing, a dataset in VTK entails creating a "cross-section" through the dataset using any type of implicit function. For example, we can slice through a dataset with a plane to create a planar cut. The cutting surface interpolates the data as it cuts, which can then be visualized using any standard visualization technique. The result of cutting is always of type vtkPolyData. (Cutting a *n*-dimensional cell results in a *(n-1)*-dimensional output primitive. For example, cutting a tetrahedron creates either a triangle or quadrilateral.)

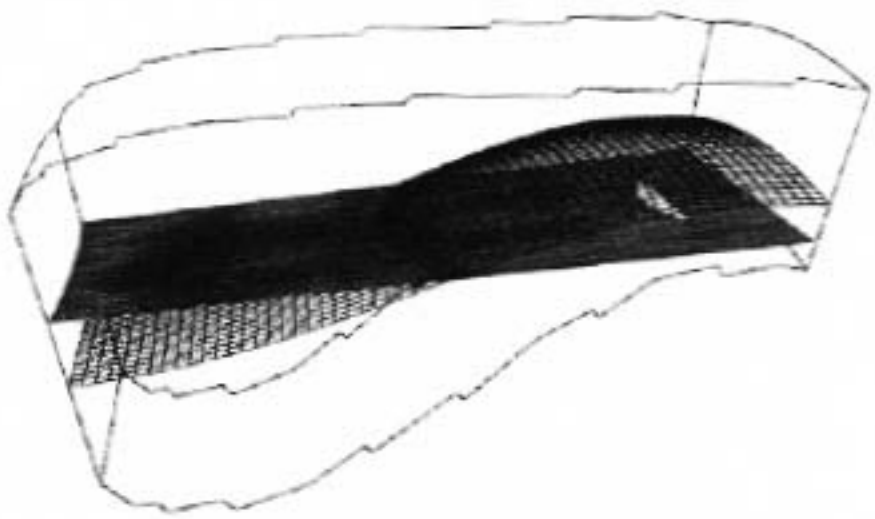

Figure 5–6 Cutting a combustor.

In the following Tcl example, a combustor (structured grid) is cut with a plane as shown in **Figure 5–6**. The example is taken from `VTK/Examples/VisualizationAlgorithms/Tcl/probe.tcl`.

```
vtkPlane plane
    eval plane SetOrigin [[pl3d GetOutput] GetCenter]
    plane SetNormal -0.287 0 0.9579
vtkCutter planeCut
    planeCut SetInput [pl3d GetOutput]
    planeCut SetCutFunction plane
vtkPolyDataMapper cutMapper
    cutMapper SetInput [planeCut GetOutput]
    eval cutMapper SetScalarRange \
      [[[[pl3d GetOutput] GetPointData] GetScalars] GetRange]
vtkActor cutActor
    cutActor SetMapper cutMapper
```

vtkCutter requires that you specify an implicit function with which to cut. Also, you may wish to specify one or more cut values using the SetValue() or GenerateValues() methods. These values specify the value of the implicit function used to perform the cutting. (Typically the cutting value is zero, meaning that the cut surface is precisely on the implicit function. Values less than or greater than zero are implicit surfaces below and above the implicit surface. The cut value can also be thought of as a "distance" to the implicit surface, which is only strictly true for vtkPlane.)

Merging Data

Up to this point we have seen simple, linear visualization pipelines. However, it is possible for pipelines to have loops, and for pieces of data to move from one leg of the pipeline to another. In this section and the following, we introduce two filters that allow you to build datasets from other datasets. We'll start with vtkMergeFilter.

vtkMergeFilter merges pieces of data from several datasets into a new dataset. For example, you can take the structure (topology and geometry) from one dataset, the scalars from a second, and the vectors from a third dataset, and combine them into a single dataset. Here's an example of its use (From the Tcl script `VTK/Examples/VisualizationAlgorithms/Tcl/imageWarp.tcl`). (Please ignore those filters that you don't recognize, focus on the use of vtkMergeFilter. We'll describe more fully the details of the script in "Warp Based On Scalar Values" on page 125.)

```
vtkBMPReader reader
  reader SetFileName $VTK_DATA_ROOT/Data/masonry.bmp
vtkImageLuminance luminance
  luminance SetInput [reader GetOutput]
vtkImageDataGeometryFilter geometry
  geometry SetInput [luminance GetOutput]
vtkWarpScalar warp
  warp SetInput [geometry GetOutput]
  warp SetScaleFactor -0.1

# use merge to put back scalars from image file
vtkMergeFilter merge
  merge SetGeometry [warp GetOutput]
  merge SetScalars  [reader GetOutput]
vtkDataSetMapper mapper
  mapper SetInput [merge GetOutput]
  mapper SetScalarRange 0 255
  mapper ImmediateModeRenderingOff
vtkActor actor
  actor SetMapper mapper
```

What's happening here is that the dataset (or geometry) from vtkWarpScalar (which happens to be of type vtkPolyData) is combined with the scalar data from the vtkPNMReader. The pipeline has split and rejoined because the geometry had to be processed separately (in the imaging pipeline) from the scalar data.

When merging data, the number of tuples found in the data arrays that make up the point attribute data must equal the number of points. This is also true for the cell data.

Appending Data

Like vtkMergeFilter, vtkAppendFilter (and its specialized cousin vtkAppendPolyData) builds a new dataset by appending datasets. The append filters take a list of inputs, each of which must be the same type. During the append operation, only those data attributes that are common to all input datasets are appended together. A great example of its application is shown in the example in the following section ("Probing" on page 98).

Probing

Probing is a process of sampling one dataset with another dataset. In VTK, you can use any dataset as a probe geometry onto which point data attributes are mapped from another dataset. For example, the following Tcl script (taken from `VTK/Examples/VisualizationAlgorithms/Tcl/probeComb.tcl`) creates three planes (which serve as the probe geometry) used to sample a structured grid dataset. The planes are then processed with vtkContourFilter to generate contour lines.

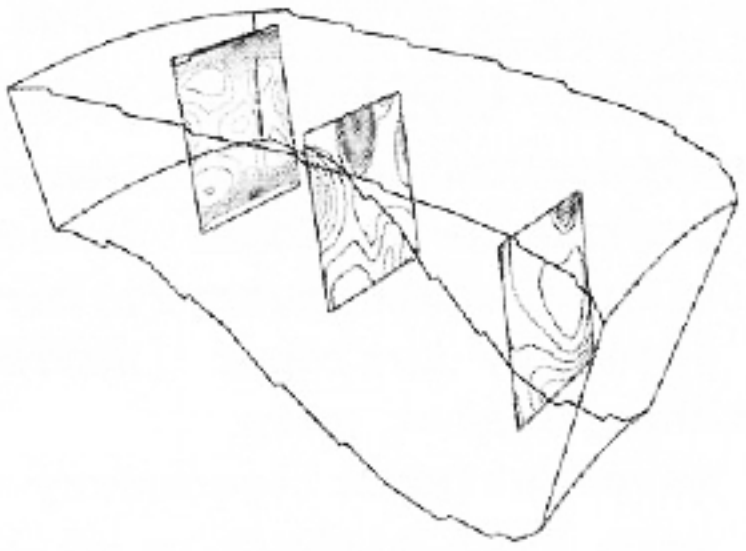

Figure 5–7 Probing data.

```
# Create pipeline
vtkPLOT3DReader pl3d
    pl3d SetXYZFileName "$VTK_DATA_ROOT/Data/combxyz.bin"
    pl3d SetQFileName "$VTK_DATA_ROOT/Data/combq.bin"
    pl3d SetScalarFunctionNumber 100
    pl3d SetVectorFunctionNumber 202
    pl3d Update;#force data read

# Create the probes. Transform them into right place.
vtkPlaneSource plane
    plane SetResolution 50 50
vtkTransform transP1
    transP1 Translate 3.7 0.0 28.37
    transP1 Scale 5 5 5
    transP1 RotateY 90
```

```
vtkTransformPolyDataFilter tpd1
    tpd1 SetInput [plane GetOutput]
    tpd1 SetTransform transP1
vtkOutlineFilter outTpd1
    outTpd1 SetInput [tpd1 GetOutput]
vtkPolyDataMapper mapTpd1
    mapTpd1 SetInput [outTpd1 GetOutput]
vtkActor tpd1Actor
    tpd1Actor SetMapper mapTpd1
    [tpd1Actor GetProperty] SetColor 0 0 0

vtkTransform transP2
    transP2 Translate 9.2 0.0 31.20
    transP2 Scale 5 5 5
    transP2 RotateY 90
vtkTransformPolyDataFilter tpd2
    tpd2 SetInput [plane GetOutput]
    tpd2 SetTransform transP2
vtkOutlineFilter outTpd2
    outTpd2 SetInput [tpd2 GetOutput]
vtkPolyDataMapper mapTpd2
    mapTpd2 SetInput [outTpd2 GetOutput]
vtkActor tpd2Actor
    tpd2Actor SetMapper mapTpd2
    [tpd2Actor GetProperty] SetColor 0 0 0

vtkTransform transP3
    transP3 Translate 13.27 0.0 33.30
    transP3 Scale 5 5 5
    transP3 RotateY 90
vtkTransformPolyDataFilter tpd3
    tpd3 SetInput [plane GetOutput]
    tpd3 SetTransform transP3
vtkOutlineFilter outTpd3
    outTpd3 SetInput [tpd3 GetOutput]
vtkPolyDataMapper mapTpd3
    mapTpd3 SetInput [outTpd3 GetOutput]
vtkActor tpd3Actor
    tpd3Actor SetMapper mapTpd3
    [tpd3Actor GetProperty] SetColor 0 0 0

vtkAppendPolyData appendF
    appendF AddInput [tpd1 GetOutput]
    appendF AddInput [tpd2 GetOutput]
    appendF AddInput [tpd3 GetOutput]
vtkProbeFilter probe
```

```
    probe SetInput [appendF GetOutput]
    probe SetSource [pl3d GetOutput]
vtkContourFilter contour
    contour SetInput [probe GetOutput]
    eval contour GenerateValues 50 [[pl3d GetOutput]\
                                    GetScalarRange]
vtkPolyDataMapper contourMapper
    contourMapper SetInput [contour GetOutput]
    eval contourMapper SetScalarRange [[pl3d GetOutput]\
                                       GetScalarRange]
vtkActor planeActor
    planeActor SetMapper contourMapper
```

Notice that the probe is set as the input to vtkProbeFilter, and the dataset to probe is set to the Source of vtkProbeFilter.

Another useful application of probing is resampling data. For example, if you have an unstructured grid and wish to visualize it with tools specific to vtkImageData (such as volume rendering—see "Volume Rendering" on page 136), you can use vtkProbeFilter to sample the unstructured grid with a volume, and then visualize the volume. It is also possible to probe data with lines (or curves) and use the output to perform *x*-*y* plotting.

One final note: cutting and probing can give similar results, although there is a difference in resolution. Similar to the example described in "Cutting" on page 96, vtkProbeFilter could be used with a vtkPlaneSource to generate a plane with data attributes from the structured grid. However, cutting creates surfaces with a resolution dependent on the resolution of the input data. Probing creates surfaces (and other geometries) with a resolution independent of the input data. Care must be taken when probing data to avoid under- or oversampling. Undersampling can result in errors in visualization, and oversampling can consume excessive computation time.

Color An Isosurface With Another Scalar

A common visualization task is to generate an isosurface and then color it with another scalar. While you might do this with a probe, their is a much more efficient way when the dataset that you isosurface contains the data you wish to color the isosurface with. This is because the vtkContourFilter (which generates the isosurface) interpolates all data to the isosurface during the generation process. The interpolated data can then be used during the mapping process to color the isosurface. Here's an example from the Tcl script `VTK/Examples/VisualizationAlgorithms/Tcl/ColorIsosurface.tcl`.

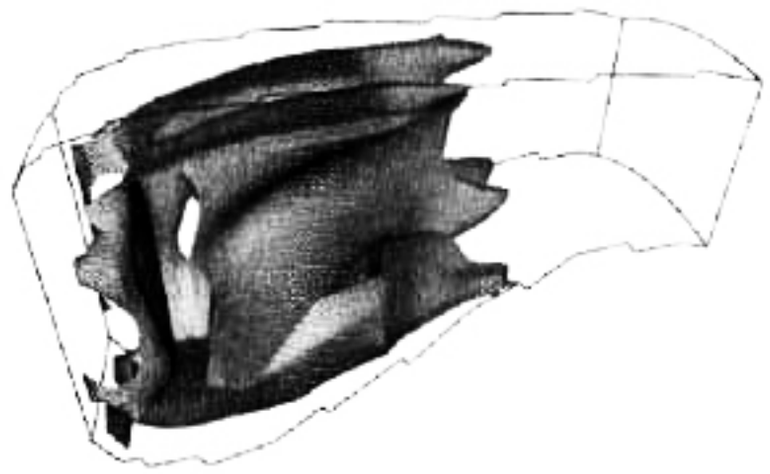

Figure 5–8 Coloring an isosurface with another scalar.

```
vtkPLOT3DReader pl3d
    pl3d SetXYZFileName "$VTK_DATA_ROOT/Data/combxyz.bin"
    pl3d SetQFileName "$VTK_DATA_ROOT/Data/combq.bin"
    pl3d SetScalarFunctionNumber 100
    pl3d SetVectorFunctionNumber 202
    pl3d AddFunction 153
    pl3d Update
vtkContourFilter iso
    iso SetInput [pl3d GetOutput]
    iso SetValue 0 .24
vtkPolyDataNormals normals
    normals SetInput [iso GetOutput]
    normals SetFeatureAngle 45
vtkPolyDataMapper isoMapper
    isoMapper SetInput [normals GetOutput]
    isoMapper ScalarVisibilityOn
    isoMapper SetScalarRange 0 1500
    isoMapper SetScalarModeToUsePointFieldData
    isoMapper ColorByArrayComponent "Velocity Magnitude" 0
vtkLODActor isoActor
    isoActor SetMapper isoMapper
    isoActor SetNumberOfCloudPoints 1000
```

First, the dataset is read with a vtkPLOT3DReader. Here we add a function to be read (function number 153) which we know to be named "Velocity Magnitude." An isosurface is generated which also interpolates all its input data arrays including the velocity magnitude data. We then use the velocity magnitude to color the contour by invoking the method

SetScalarModeToUsePointFieldData() and specifying the data array to use to color with the ColorByArrayComponent() method.

Extract Subset of Cells

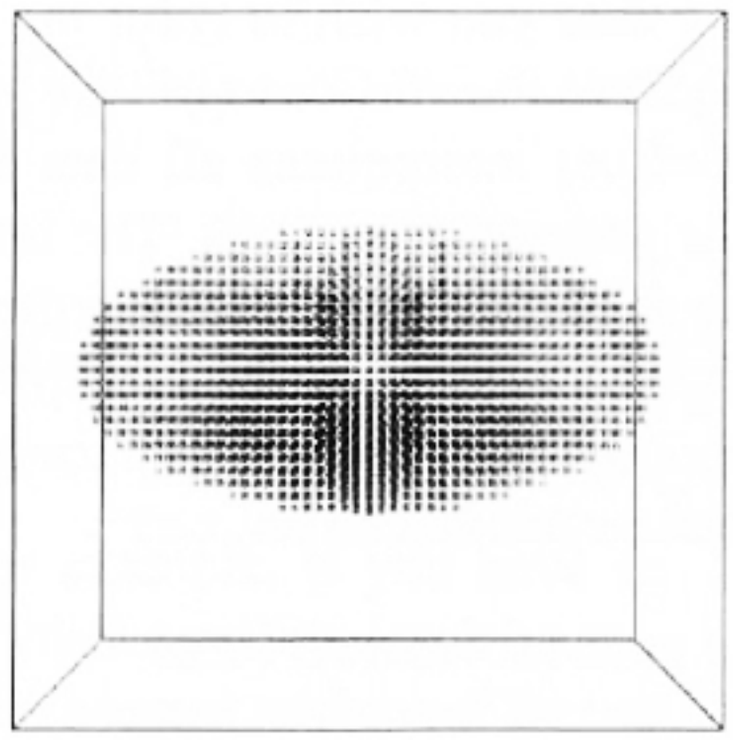

Figure 5–9 Extracting cells.

Visualization data is often large and processing such data can be quite costly in execution time and memory requirements. As a result, the ability to extract pieces of data is important. Many times only a subset of the data contains meaningful information, or the resolution of the data can be reduced without significant loss of accuracy.

The *Visualization Toolkit* offers several tools to extract portions of, or subsample data. We've already seen how vtkProbeFilter can be used to subsample data (see "Probing" on page 98). Other tools include classes to subsample data, and tools to extract cells within a region in space. (Subsampling tools are specific to a type of dataset. See "Subsampling Image Data" on page 124 for information about subsampling structured points datasets, and "Subsampling Structured Grids" on page 115 for information about subsampling structured grids.) In this section, we describe how to extract pieces of a dataset contained within a region in space.

The class vtkExtractGeometry extracts all cells in a dataset that lie either inside or outside of a vtkImplicitFunction (remember, implicit functions can consist of boolean combinations of other implicit functions). The following script creates a boolean combination of two ellipsoids that is used as the extraction region. A vtkShrinkFilter is also used to shrink the cells so you can see what's been extracted. (The Tcl script is from `VTK/Examples/VisualizationAlgorithms/Tcl/ExtractGeometry.tcl`.)

```
vtkQuadric quadric
    quadric SetCoefficients .5 1 .2 0 .1 0 0 .2 0 0
vtkSampleFunction sample
    sample SetSampleDimensions 50 50 50
    sample SetImplicitFunction quadric
    sample ComputeNormalsOff
vtkTransform trans
    trans Scale 1 .5 .333
vtkSphere sphere
```

```
    sphere SetRadius 0.25
    sphere SetTransform trans
vtkTransform trans2
    trans2 Scale .25 .5 1.0
vtkSphere sphere2
    sphere2 SetRadius 0.25
    sphere2 SetTransform trans2
vtkImplicitBoolean union
    union AddFunction sphere
    union AddFunction sphere2
    union SetOperationType 0;#union

vtkExtractGeometry extract
    extract SetInput [sample GetOutput]
    extract SetImplicitFunction union
vtkShrinkFilter shrink
    shrink SetInput [extract GetOutput]
    shrink SetShrinkFactor 0.5
vtkDataSetMapper dataMapper
    dataMapper SetInput [shrink GetOutput]
vtkActor dataActor
    dataActor SetMapper dataMapper
```

The output of vtkExtractGeometry is always a vtkUnstructuredGrid. This is because the extraction process generally disrupts the topological structure of the dataset, and the most general dataset form (i.e., vtkUnstructuredGrid) must be used to represent the output.

As a side note: implicit functions can be transformed by assigning them a vtkTransform. If specified, the vtkTransform is used to modify the evaluation of the implicit function. You may wish to experiment with this capability.

Extract Cells As Polygonal Data

Most dataset types cannot be directly rendered by graphics hardware or libraries. Only polygonal data (vtkPolyData) is commonly supported by rendering systems. Structured points datasets, especially images and sometimes volumes, are also supported by graphics systems. All other datasets require special processing if they are to be rendered. In VTK, one approach to rendering non-polygonal datasets is to convert them to polygonal data. This is the function of vtkGeometryFilter.

vtkGeometryFilter accepts as input any type of vtkDataSet, and generates vtkPolyData on output. It performs the conversion using the following rules. All input cells of topological

dimension 2 or less (e.g., polygons, lines, vertices) are passed to the output. The faces of cells of dimension 3 are sent to the output if they are on the boundary of the dataset. (A face is on the boundary if it is used by only one cell.) In addition, vtkGeometryFilter has methods that allows you to extract cells based on a range of point ids, cell ids, or whether the cells lie in a particular rectangular region in space.

The principal use of vtkGeometryFilter is as a conversion filter. The following example from `VTK/Examples/DataManipulation/Tcl/pointToCellData.tcl` uses vtkGeometryFilter to convert a 2D unstructured grid into polygonal data for later processing by filters that accept vtkPolyData as input. Here, the vtkConnectivityFilter extracts data as vtkUnstructuredGrid which is then converted into polygons using vtkGeometryFilter.

```
vtkConnectivityFilter connect2
    connect2 SetInput [thresh GetOutput]
vtkGeometryFilter parison
    parison SetInput [connect2 GetOutput]
vtkPolyDataNormals normals2
    normals2 SetInput [parison GetOutput]
    normals2 SetFeatureAngle 60
vtkLookupTable lut
    lut SetHueRange 0.0 0.66667
vtkPolyDataMapper parisonMapper
    parisonMapper SetInput [normals2 GetOutput]
    parisonMapper SetLookupTable lut
    parisonMapper SetScalarRange 0.12 1.0
vtkActor parisonActor
    parisonActor SetMapper parisonMapper
```

In fact, the vtkDataSetMapper mapper uses vtkGeometryFilter internally to convert datasets of any type into polygonal data. (The filter is smart enough to pass input vtkPolyData straight to its output without processing.)

vtkGeometryFilter extracts pieces of datasets based on point and cell ids using the methods PointClippingOn(), SetPointMinimum(), SetPointMaximum() and CellClippingOn(), SetCellMinimum(), SetCellMaximum(). The minimum and maximum values specify a range of ids which are extracted. Also, you can use a rectangular region in space to limit what's extracted. Use the ExtentClippingOn() and SetExtent() to enable extent clipping and specify the extent. (The extent consists of six values defining a bounding box in space—(x_{min},x_{max}, y_{min},y_{max}, z_{min},z_{max}).) You can use point, cell, and extent clipping in any combination. This is a useful feature when debugging data, or when you only want to look at a portion of it.

5.2 Visualizing Polygonal Data

Polygonal data (vtkPolyData) is an important form of visualization data. Its importance is due to it use as the geometry interface into the graphics hardware/rendering engine. Other data types must be converted into polygonal data in order to be rendered (with the exception of vtkImageData (images and volumes) which uses special imaging or volume rendering techniques). You may wish to refer to "Extract Cells As Polygonal Data" on page 103 to see how this conversion is performed.

Polygonal data (vtkPolyData) consists of combinations of vertices and polyvertices; lines and polylines, triangles, quadrilaterals, and polygons; and triangle strips. Most filters (that input vtkPolyData) will process any combination of this data; however, some filters (like vtkDecimatePro and vtkTubeFilter) will only process portions of the data (triangle meshes and lines).

Manually Create vtkPolyData

Polygonal data can be constructed several different ways. Typically, you'll create a vtkPoints to represent the points, and then one to four vtkCellArrays to represent vertex, line, polygon, and triangle strip connectivity. Here's an example taken from `VTK/Examples/DataManipulation/Tcl/CreateStrip.tcl`. It creates a vtkPolyData with a single triangle strip.

```
vtkPoints points
    points InsertPoint 0 0.0 0.0 0.0
    points InsertPoint 1 0.0 1.0 0.0
    points InsertPoint 2 1.0 0.0 0.0
    points InsertPoint 3 1.0 1.0 0.0
    points InsertPoint 4 2.0 0.0 0.0
    points InsertPoint 5 2.0 1.0 0.0
    points InsertPoint 6 3.0 0.0 0.0
    points InsertPoint 7 3.0 1.0 0.0
vtkCellArray strips
    strips InsertNextCell 8;#number of points
    strips InsertCellPoint 0
    strips InsertCellPoint 1
    strips InsertCellPoint 2
    strips InsertCellPoint 3
    strips InsertCellPoint 4
    strips InsertCellPoint 5
```

```
    strips InsertCellPoint 6
    strips InsertCellPoint 7
vtkPolyData profile
    profile SetPoints points
    profile SetStrips strips
vtkPolyDataMapper map
    map SetInput profile
vtkActor strip
    strip SetMapper map
    [strip GetProperty] SetColor 0.3800 0.7000 0.1600
```

In C++, here's another example showing how to create a cube (`Cube.cxx`). This time we create six quadrilateral polygons, as well as scalar values at the vertices of the cube.

```
int i;
static float x[8][3]={{0,0,0}, {1,0,0}, {1,1,0}, {0,1,0},
                      {0,0,1}, {1,0,1}, {1,1,1}, {0,1,1}};
static vtkIdType pts[6][4]={{0,1,2,3}, {4,5,6,7}, {0,1,5,4},
                      {1,2,6,5}, {2,3,7,6}, {3,0,4,7}};

// We'll create the building blocks of polydata including data attributes.
vtkPolyData *cube = vtkPolyData::New();
vtkPoints *points = vtkPoints::New();
vtkCellArray *polys = vtkCellArray::New();
vtkFloatArray *scalars = vtkFloatArray::New();

// Load the point, cell, and data attributes.
for (i=0; i<8; i++) points->InsertPoint(i,x[i]);
for (i=0; i<6; i++) polys->InsertNextCell(4,pts[i]);
for (i=0; i<8; i++) scalars->InsertTuple1(i,i);

// We now assign the pieces to the vtkPolyData.
cube->SetPoints(points);
points->Delete();
cube->SetPolys(polys);
polys->Delete();
cube->GetPointData()->SetScalars(scalars);
scalars->Delete();
```

vtkPolyData can be constructed with any combination of vertices, lines, polygons, and triangle strips. Also, vtkPolyData supports an extensive set of operators that allows you to edit and modify the underlying structure. Refer to "Interface To vtkPolyData" on page 253 for more information.

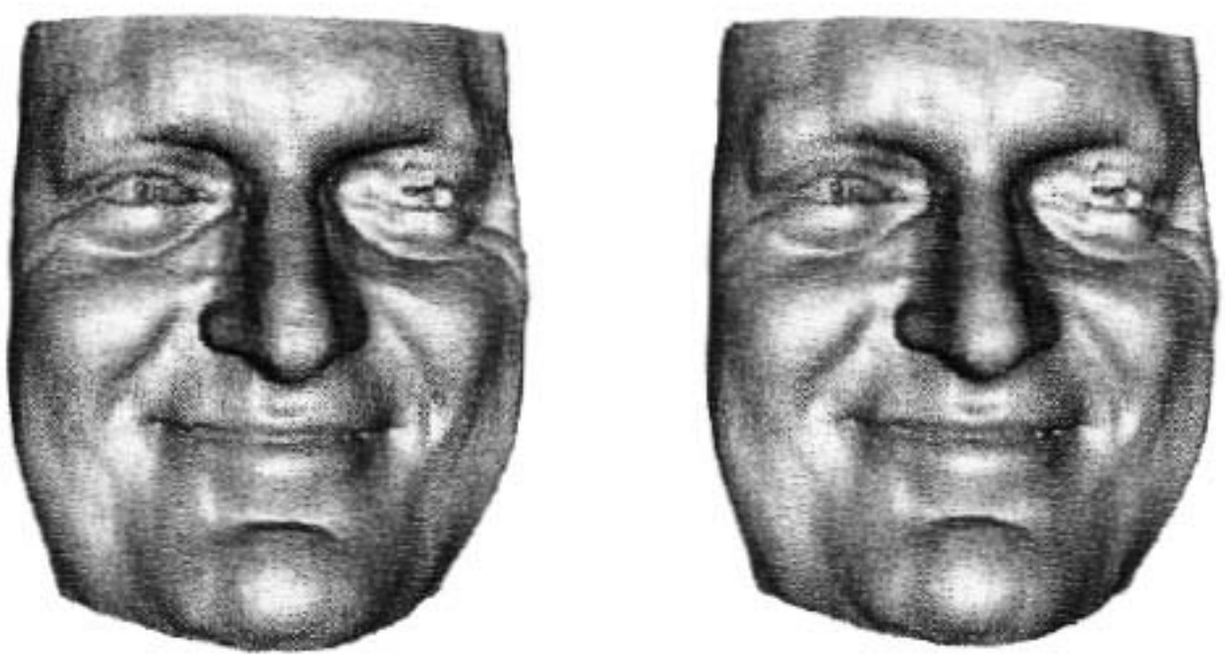

Figure 5–10 Comparing a mesh with and without surface normals.

Generate Surface Normals

When you render a polygonal mesh, you may find that the image clearly shows the faceted nature of the mesh (**Figure 5–10**). The image can be improved by using Gouraud shading (see "Actor Properties" on page 58). However, Gouraud shading depends on the existence of normals at each point in the mesh.The vtkPolyDataNormals filter can be used to generate normals on the mesh. The scripts in "Sampling Implicit Functions" on page 167, "Glyphing" on page 90, and "Color An Isosurface With Another Scalar" on page 101 all use vtkPolyDataNormals, refer to them to see how the filter is used.

Two important instance variables are Splitting and FeatureAngle. If splitting is on, feature edges (defined as edges where the polygonal normals on either side of the edge make an angle greater than or equal to the feature angle) are "split," that is, points are duplicated along the edge, and the mesh is separated on either side of the feature edge (see *The Visualization Toolkit* text). This creates new points, but allows sharp corners to be rendered crisply. Another important instance variable is FlipNormals. Invoking FlipNormalsOn() causes the filter to reverse the direction of the normals (and the ordering of the polygon connectivity list).

Decimation

Polygonal data, especially triangle meshes, are a common form of graphics data. Filters such as vtkContourFilter generate triangle meshes. Often, these meshes are quite large and cannot be rendered or processed quickly enough for interactive application. Decimation techniques have been developed to address this problem. Decimation, also referred to as polygonal reduction, mesh simplification, or multiresolution modeling, is a process to reduce the number of triangles in a triangle mesh, while maintaining a faithful approximation to the original mesh.

VTK supports four decimation objects: vtkDecimate, vtkDecimatePro, vtkQuadricClustering, and vtkQuadricDecimation. All are similar in usage and application, although they each offer advantages and disadvantages as follows:

- vtkDecimatePro is relatively fast and has the ability to modify topology during the reduction process. It uses an edge collapse process to eliminate vertices and triangles. It's error metric is based on distance to plane/distance to edge. A nice feature of vtkDecimatePro is that you can achieve any level of reduction requested, since the algorithm will begin tearing the mesh into pieces to achieve this (if topology modification is allowed).
- vtkDecimate is patented (be careful if you use it in commercial application), and it has the slight advantage that it uses a better triangulation technique during processing (i.e., results might be slightly better) than some of the other methods. It's major feature is that it is one of the first decimation algorithms, and is relatively easy to understand and modify.
- vtkQuadricDecimation uses the quadric error measure proposed by Garland and Heckbert in Siggraph '97 *Surface Simplification Using Quadric Error Metrics*. It uses an edge collapse to eliminate vertices and triangles. The quadric error metric is generally accepted as one of the better error metrics.
- vtkQuadricClustering is the fastest algorithm. It is based on the algorithm presented by Peter Lindstrom in his Siggraph 2000 paper *Out-of-Core Simplification of Large Polygonal Models*. It is capable of quickly reducing huge meshes, and the class supports the ability to process pieces of a mesh (using the StartAppend(), Append(), and EndAppend() methods). This enables the user to avoid reading an entire mesh into memory. This algorithm works well with large meshes; the triangulation process does not work well as meshes are become smaller. (Combining this algorithm with

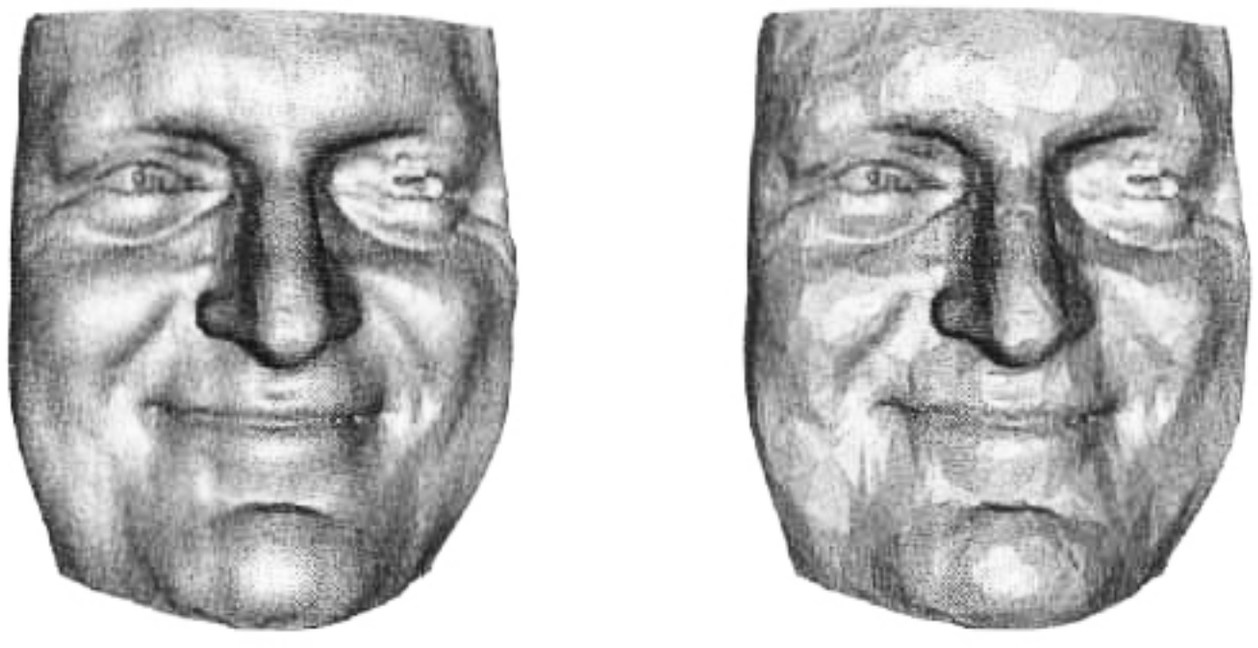

Figure 5–11 Triangle mesh before (left) and after (right) 90% decimation.

one of the other algorithms is a good approach.)

Here's an example using vtkDecimatePro. It's been adapted from the Tcl script `VTK/Examples/VisualizationAlgorithms/Tcl/deciFran.tcl` (**Figure 5–11**).

```
vtkDecimatePro deci
    deci SetInput [fran GetOutput]
    deci SetTargetReduction 0.9
    deci PreserveTopologyOn
vtkPolyDataNormals normals
    normals SetInput [fran GetOutput]
    normals FlipNormalsOn
vtkPolyDataMapper franMapper
    franMapper SetInput [normals GetOutput]
vtkActor franActor
    franActor SetMapper franMapper
    eval [franActor GetProperty] SetColor 1.0 0.49 0.25
```

Two important instance variables of vtkDecimatePro are TargetReduction and PreserveTopology. The TargetReduction is the requested amount of reduction (e.g., a value of 0.9 means that we wish to reduce the number of triangles in the mesh by 90%). Depending on whether you allow topology to change or not (PreserveTopologyOn/Off()), you may or may not achieve the requested reduction. If PreserveTopology is off, then vtkDecimatePro will give you the requested reduction.

A final note: the decimation filters take triangle data as input. If you have a polygonal mesh you can convert the polygons to triangles with vtkTriangleFilter.

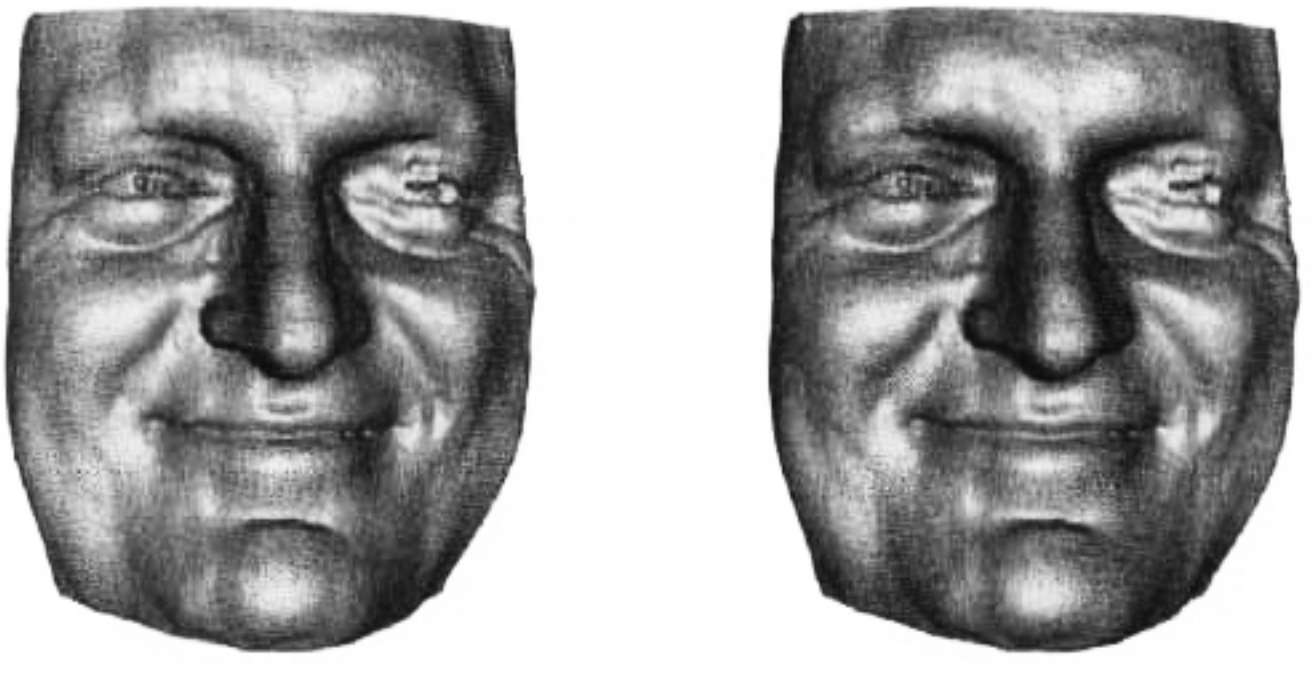

Figure 5–12 Smoothing a polygonal mesh. Right image shows the effect of smoothing.

Smooth Mesh

Polygonal meshes often contain noise or excessive roughness that affect the quality of the rendered image. For example, isosurfacing low resolution data can show aliasing, or stepping effects. One way to treat this problem is to use smoothing. Smoothing is a process that adjusts the positions of points to reduce the noise content in the surface.

VTK offers two smoothing objects: vtkSmoothPolyDataFilter and vtkWindowedSincPolyDataFilter. Of the two, the vtkWindowedSincPolyDataFilter gives the best results and is slightly faster. The following example shows how to use the smoothing filter. The example is the same as the one in the previous section, except that a smoothing filter has been added. **Figure 5–12** shows the effects of smoothing on the decimated mesh.

```
# decimate and smooth data
vtkDecimatePro deci
    deci SetInput [fran GetOutput]
    deci SetTargetReduction 0.9
    deci PreserveTopologyOn
vtkSmoothPolyDataFilter smoother
    smoother SetInput [deci GetOutput]
    smoother SetNumberOfIterations 50
vtkPolyDataNormals normals
    normals SetInput [smoother GetOutput]
    normals FlipNormalsOn
vtkPolyDataMapper franMapper
    franMapper SetInput [normals GetOutput]
vtkActor franActor
```

```
franActor SetMapper franMapper
eval [franActor GetProperty] SetColor 1.0 0.49 0.25
```

Both smoothing filters are used similarly. There are optional methods for controlling the effects of smoothing along feature edges and on boundaries. Check the man pages and/or `.h` files for more information.

Clip Data

Clipping, like cutting (see "Cutting" on page 96), uses an implicit function to define a surface with which to clip. Clipping separates a polygonal mesh into pieces, as shown in **Figure 5–13**. Clipping will break polygonal primitives into separate parts on either side of the clipping surface. Like cutting, clipping allows you to set a clip value defining the value of the implicit clipping function.

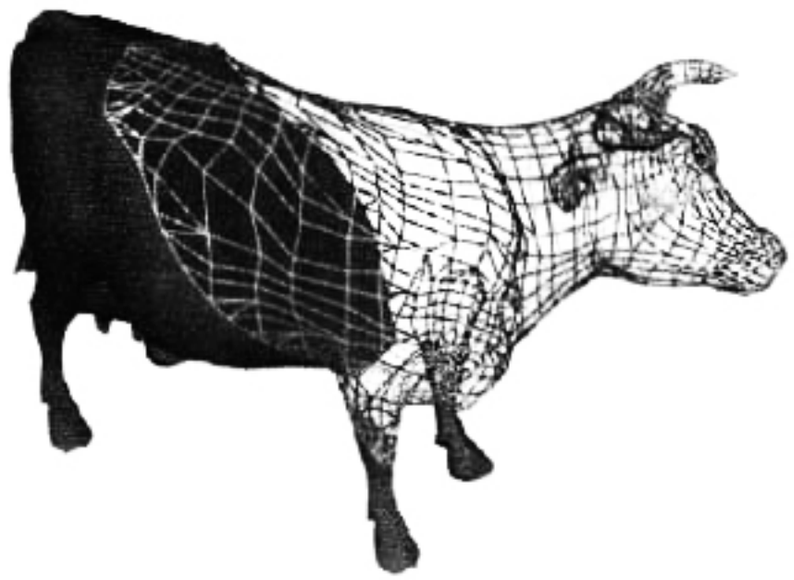

Figure 5–13 Clipping a model.

The following example uses a plane to clip a polygonal model of a cow. The clip value is used to move the plane along its normal so that the model can be clipped at different locations. The example Tcl script shown below is taken from `VTK/Examples/Visualization-Algorithms/Tcl/ClipCow.tcl`.

```
# Read the polygonal data and generate vertex normals
vtkBYUReader cow
  cow SetGeometryFileName "$VTK_DATA_ROOT/Data/Viewpoint/cow.g"
vtkPolyDataNormals cowNormals
  cowNormals SetInput [cow GetOutput]

# Define a clip plane to clip the cow in half
vtkPlane plane
    plane SetOrigin 0.25 0 0
    plane SetNormal -1 -1 0
vtkClipPolyData clipper
    clipper SetInput [cowNormals GetOutput]
    clipper SetClipFunction plane
    clipper GenerateClippedOutputOn
    clipper SetValue 0.5
vtkPolyDataMapper clipMapper
    clipMapper SetInput [clipper GetOutput]
```

```
vtkActor clipActor
    clipActor SetMapper clipMapper
    eval [clipActor GetProperty] SetColor $peacock

# Create the rest of the cow in wireframe
vtkPolyDataMapper restMapper
  restMapper SetInput [clipper GetClippedOutput]
vtkActor restActor
  restActor SetMapper restMapper
  [restActor GetProperty] SetRepresentationToWireframe
```

The GenerateClippedOutputOn() method causes the filter to create a second output: the data that was clipped away. This output is shown in wireframe in the figure. If the SetValue() method is used to change the clip value, the implicit function will cut at a point parallel to the original plane, but above or below it. (You could also change the definition of vtkPlane to achieve the same result.)

Generate Texture Coordinates

Several filters are available to generate texture coordinates: vtkTextureMapToPlane, vtkTextureMapToCylinder, and vtkTextureMapToSphere. These objects generated texture coordinates based on a planar, cylindrical, and spherical coordinate system, respectively. Also, the class vtkTransformTextureCoordinates allows you to position the texture map on the surface by translating and scaling the texture coordinates. The following example shows using vtkTextureMapToCylinder to create texture coordinates for an unstructured grid generated from the vtkDelaunay3D object (see "Delaunay Triangulation" on page 171 for more information). The full example can be found at `VTK/Examples/VisualizationAlgorithms/Tcl/GenerateTextureCoordinates.tcl`.

```
vtkPointSource sphere
  sphere SetNumberOfPoints 25

vtkDelaunay3D del
  del SetInput [sphere GetOutput]
  del SetTolerance 0.01

vtkTextureMapToCylinder tmapper
  tmapper SetInput [del GetOutput]
  tmapper PreventSeamOn

vtkTransformTextureCoords xform
  xform SetInput [tmapper GetOutput]
```

```
  xform SetScale 4 4 1

vtkDataSetMapper mapper
  mapper SetInput [xform GetOutput]

vtkBMPReader bmpReader
  bmpReader SetFileName "$VTK_DATA_ROOT/Data/masonry.bmp"
vtkTexture atext
  atext SetInput [bmpReader GetOutput]
  atext InterpolateOn
vtkActor triangulation
  triangulation SetMapper mapper
  triangulation SetTexture atext
```

In this example a random set of points in the unit sphere is triangulated. The triangulation then has texture coordinates generated over it. These texture coordinates are then scaled in the *i-j* texture coordinate directions in order to cause texture repeats. Finally, a texture map is read in and assigned to the actor.

As a side note: instances of vtkDataSetMapper are mappers that accept any type of data as input. They use an internal instance of vtkGeometryFilter followed by vtkPolyDataMapper to convert the data into polygonal primitives that can then be passed to the rendering engine. See "Extract Cells As Polygonal Data" on page 103 for further information.

5.3 Visualizing Structured Grids

Structured grids are regular in topology, and irregular in geometry (see **Figure 3–2**(c)). Structured grids are often used in numerical analysis (e.g., computational fluid dynamics). The vtkStructuredGrid dataset is composed of hexahedral (vtkHexahedron) or quadrilateral (vtkQuad) cells.

Manually Create vtkStructuredGrid

Structured grids are created by specifying grid dimensions (to define topology) along with a vtkPoints object defining the *x-y-z* point coordinates (to define geometry). This code was derived from `VTK/Examples/DataManipulation/Cxx/SGrid.cxx`.

```
vtkPoints points
   points InsertPoint 0 0.0 0.0 0.0
```

```
   ...etc...

vtkStructuredGrid sgrid
   sgrid SetDimensions 13 11 11
   sgrid SetPoints points
```

Make sure that the number of points in the vtkPoints object is consistent with the number of points defined by the product of the three dimension values in the *i*, *j*, and *k* topological directions.

Extract Computational Plane

In most cases, structured grids are processed by filters that accept vtkDataSet as input (see "Visualization Techniques" on page 83). One filter that directly accepts vtkStructuredGrid as input is the vtkStructuredGridGeometryFilter. This filter is used to extract pieces of the grid as points, lines, or polygonal "planes", depending on the specification of the Extent instance variable. (Extent is a 6-vector that describes a $(i_{min}, i_{max}, j_{min}, j_{max}, k_{min}, k_{max})$ topological region.)

In the following example, we read a structured grid, extract three planes, and warp the planes with the associated vector data (from `VTK/Examples/VisualizationALgorithms/Tcl/warpComb.tcl`).

```
vtkPLOT3DReader pl3d
    pl3d SetXYZFileName "$VTK_DATA_ROOT/Data/combxyz.bin"
    pl3d SetQFileName "$VTK_DATA_ROOT/Data/combq.bin"
    pl3d SetScalarFunctionNumber 100
    pl3d SetVectorFunctionNumber 202
    pl3d Update
vtkStructuredGridGeometryFilter plane
    plane SetInput [pl3d GetOutput]
    plane SetExtent 10 10 1 100 1 100
vtkStructuredGridGeometryFilter plane2
    plane2 SetInput [pl3d GetOutput]
    plane2 SetExtent 30 30 1 100 1 100
vtkStructuredGridGeometryFilter plane3
    plane3 SetInput [pl3d GetOutput]
    plane3 SetExtent 45 45 1 100 1 100
vtkAppendPolyData appendF
    appendF AddInput [plane GetOutput]
    appendF AddInput [plane2 GetOutput]
    appendF AddInput [plane3 GetOutput]
```

```
vtkWarpScalar warp
    warp SetInput [appendF GetOutput]
    warp UseNormalOn
    warp SetNormal 1.0 0.0 0.0
    warp SetScaleFactor 2.5
vtkPolyDataNormals normals
    normals SetInput [warp GetPolyDataOutput]
    normals SetFeatureAngle 60
vtkPolyDataMapper planeMapper
    planeMapper SetInput [normals GetOutput]
    eval planeMapper SetScalarRange [[pl3d GetOutput] GetScalarRange]
vtkActor planeActor
    planeActor SetMapper planeMapper
```

Subsampling Structured Grids

Structured grids can be subsampled similar to structured points (see "Subsampling Image Data" on page 124). The vtkExtractGrid performs the subsampling and data extraction.

```
vtkPLOT3DReader pl3d
    pl3d SetXYZFileName "$VTK_DATA_ROOT/Data/combxyz.bin"
    pl3d SetQFileName "$VTK_DATA_ROOT/Data/combq.bin"
    pl3d SetScalarFunctionNumber 100
    pl3d SetVectorFunctionNumber 202
    pl3d Update
vtkExtractGrid extract
    extract SetInput [pl3d GetOutput]
    extract SetVOI 30 30 -1000 1000 -1000 1000
    extract SetSampleRate 1 2 3
    extract IncludeBoundaryOn
```

In this example, a subset of the original structured grid (which has dimensions 57x33x25) is extracted with a sampling rate of (1,2,3) resulting in a structured grid with dimensions (1,17,9). The IncludeBoundaryOn method makes sure that the boundary is extracted even if the sampling rate does not pick up the boundary.

5.4 Visualizing Rectilinear Grids

Rectilinear grids are regular in topology, and semi-regular in geometry (see **Figure 3–2**(b)). Rectilinear grids are often used in numerical analysis. The vtkRectilinearGrid dataset is composed of voxel (vtkVoxel) or pixel (vtkPixel) cells.

Manually Create vtkRectilinearGrid

Rectilinear grids are created by specifying grid dimensions (to define topology) along with three scalar arrays to define point coordinates along the *x*-*y*-*z* axes (to define geometry). This code was modified from `VTK/Examples/DataManipulation/Cxx/RGrid.cxx`.

```
vtkFloatArray *xCoords = vtkFloatArray::New();
  for (i=0; i<47; i++) xCoords->InsertNextValue(x[i]);

  vtkFloatArray *yCoords = vtkFloatArray::New();
  for (i=0; i<33; i++) yCoords->InsertNextValue(y[i]);

  vtkFloatArray *zCoords = vtkFloatArray::New();
  for (i=0; i<44; i++) zCoords->InsertNextValue(z[i]);

  vtkRectilinearGrid *rgrid = vtkRectilinearGrid::New();
    rgrid->SetDimensions(47,33,44);
    rgrid->SetXCoordinates(xCoords);
    rgrid->SetYCoordinates(yCoords);
    rgrid->SetZCoordinates(zCoords);
```

Make sure that the number of scalars in the *x*, *y*, and *z* directions equals the three dimension values in the *i*, *j*, and *k* topological directions.

Extract Computational Plane

In most cases, rectilinear grids are processed by filters that accept vtkDataSet as input (see "Visualization Techniques" on page 83). One filter that directly accepts vtkRectilinearGrid as input is the vtkRectilinearGridGeometryFilter. This filter is used to extract pieces of the grid as points, lines, or polygonal "planes", depending on the specification of the Extent instance variable. (Extent is a 6-vector that describes a $(i_{min},i_{max}, j_{min},j_{max}, k_{min},k_{max})$ topological region.)

The following example, which we continue from the previous found in `VTK/Examples/DataManipulation/Cxx/RGrid.cxx` we extract a plane as follows

```
vtkRectilinearGridGeometryFilter *plane =
                                  vtkRectilinearGridGeometryFilter::New();
    plane->SetInput(rgrid);
    plane->SetExtent(0,46, 16,16, 0,43);
```

5.5 Visualizing Unstructured Grids

Unstructured grids are irregular in both topology and geometry (see **Figure 3–2**(f)). Unstructured grids are often used in numerical analysis (e.g., finite element analysis). Any and all cell types can be represented in an unstructured grid.

Manually Create vtkUnstructuredGrid

Unstructured grids are created by defining geometry via a vtkPoints instance, and defining topology by inserting cells. (This script was derived from the example `VTK/Examples/DataManipulation/Tcl/BuildUGrid.tcl`.)

```
vtkPoints tetraPoints
  tetraPoints SetNumberOfPoints 4
  tetraPoints InsertPoint 0 0 0 0
  tetraPoints InsertPoint 1 1 0 0
  tetraPoints InsertPoint 2 .5 1 0
  tetraPoints InsertPoint 3 .5 .5 1
vtkTetra aTetra
  [aTetra GetPointIds] SetId 0 0
  [aTetra GetPointIds] SetId 1 1
  [aTetra GetPointIds] SetId 2 2
  [aTetra GetPointIds] SetId 3 3
vtkUnstructuredGrid aTetraGrid
  aTetraGrid Allocate 1 1
  aTetraGrid InsertNextCell [aTetra GetCellType] [aTetra GetPointIds]
  aTetraGrid SetPoints tetraPoints
  ...insert other cells if any...
```

It is mandatory that you invoke the Allocate() method prior to inserting cells into an instance of vtkUnstructuredGrid. The values supplied to this method are the initial size of the data, and the size to extend the allocation by when additional memory is required. Larger values generally give better performance (since fewer memory reallocations are required).

Extract Portions of the Mesh

In most cases, unstructured grids are processed by filters that accept vtkDataSet as input (see "Visualization Techniques" on page 83). One filter that directly accepts vtkUnstruc-

turedGrid as input is the vtkExtractUnstructuredGrid. This filter is used to extract portions of the grid using a range of point ids, cell ids, or geometric bounds (the Extent instance variable which defines a bounding box). This script was derived from `VTK/Examples/VisualizationAlgorithms/Tcl/ExtractUGrid.tcl`.

```
vtkDataSetReader reader
    reader SetFileName "$VTK_DATA_ROOT/Data/blow.vtk"
    reader SetScalarsName "thickness9"
    reader SetVectorsName "displacement9"
vtkCastToConcrete castToUnstructuredGrid
    castToUnstructuredGrid SetInput [reader GetOutput]
vtkWarpVector warp
    warp SetInput [castToUnstructuredGrid GetUnstructuredGridOutput]

vtkConnectivityFilter connect
    connect SetInput [warp GetOutput]
    connect SetExtractionModeToSpecifiedRegions
    connect AddSpecifiedRegion 0
    connect AddSpecifiedRegion 1
vtkDataSetMapper moldMapper
    moldMapper SetInput [reader GetOutput]
    moldMapper ScalarVisibilityOff
vtkActor moldActor
    moldActor SetMapper moldMapper
    [moldActor GetProperty] SetColor .2 .2 .2
    [moldActor GetProperty] SetRepresentationToWireframe

vtkConnectivityFilter connect2
    connect2 SetInput [warp GetOutput]
    connect2 SetExtractionModeToSpecifiedRegions
    connect2 AddSpecifiedRegion 2
vtkExtractUnstructuredGrid extractGrid
    extractGrid SetInput [connect2 GetOutput]
    extractGrid CellClippingOn
    extractGrid SetCellMinimum 0
    extractGrid SetCellMaximum 23
vtkGeometryFilter parison
    parison SetInput [extractGrid GetOutput]
vtkPolyDataNormals normals2
    normals2 SetInput [parison GetOutput]
    normals2 SetFeatureAngle 60
vtkLookupTable lut
    lut SetHueRange 0.0 0.66667
vtkPolyDataMapper parisonMapper
    parisonMapper SetInput [normals2 GetOutput]
```

```
    parisonMapper SetLookupTable lut
    parisonMapper SetScalarRange 0.12 1.0
vtkActor parisonActor
    parisonActor SetMapper parisonMapper
```

In this example, we are using cell clipping (i.e., using cell ids) in combination with a connectivity filter to extract portions of the mesh. Similarly, we could use point ids and a geometric extent to extract portions of the mesh. The vtkConnectivityFilter (and a related class vtkPolyDataConnectivityFilter) are used to extract connected portions of a dataset. (Cells are connected when they share points.) The SetExtractionModeToSpecifiedRegions() method indicates to the filter which connected region to extract. By default, the connectivity filters extract the largest connected regions encountered. However, it is also possible to specify a particular region as this example does, which of course requires some experimentation to determine which region is which.

Contour Unstructured Grids

A special contouring class is available to generate isocontours for unstructured grids. The class vtkContourGrid is a higher-performing version than the generic vtkContourFilter isocontouring filter. Normally you do not need to instantiate this class directly since vtkContourFilter will automatically create an internal instance of vtkContourGrid if it senses that its input is of type vtkUnstructuredGrid.

This concludes our overview of visualization techniques. You may also wish to refer to the next chapter which describes image processing and volume rendering. Also, see "Summary Of Filters" on page 332 for a summary of the filters in VTK.