Data Structures in the Visualization Toolkit

Stefan Bruckner*

Seminar Paper
The Institute of Computer Graphics and Algorithms
Vienna University of Technology, Austria

Abstract

The Visualization Toolkit (*VTK*) is an open-source toolkit for data visualization. It is based on a data-flow model and features a rich library. This paper examines *VTK*'s visualization model and the underlying data structures. Practical examples are provided.

Keywords: Visualization Toolkit, *VTK*, Visualization Networks, Data structures

1 Introduction

The Visualization Toolkit (*VTK*) is a widely used package for various visualization tasks. It offers an object-oriented API which can be integrated in many common development environments. Language bindings for popular interpreted languages (currently TCL, Python and Java) allow rapid-prototyping while C++ can be used to develop high-performance applications.

A main feature of the toolkit is its simple and well-structured nature which incorporates popular object-oriented design patterns. Since *VTK* is open-source it is easy for developers to extend its functionality and add new components [1].

This paper examines *VTK*'s object model and gives examples for taking advantage of it.

In Section 2 *VTK*'s visualization pipeline is introduced. The difference between process and data objects is discussed and *VTK*'s data structures are examined. Section 3 gives practical examples for using *VTK*. The paper is concluded in Section 4.

2 Visualization Model

2.1 Overview

The Visualization Toolkit uses a data-flow approach to transform information into graphical data. This architecture is commonly referred to as a visualization network.

A network is essentially a graph that defines the flow of data through a series of modules that process the data into a picture that can be viewed on the screen.

At the top of all networks is some kind of data input module that reads data files into the network. Next may come a series of filter modules that preprocess the data (e.g., extract a single scalar element from a vector of data values, crop or thin out the data to a more manageable size, take a single 2D plane from a 3D volume, etc.). This is followed by one or more mapper modules that turn the data into a picture or write data to a file or stream.

Similar to other data-flow based systems, such as AVS [2] and Data Explorer [3], the *VTK* visualization model consists of two basic types of objects: process objects and data objects.

Process objects: Process objects are the algorithmic portions of the visualization network. They are further characterized as source objects (e.g., a file containing geometric data), filter objects (e.g., triangulation of the input data) and mapper objects (e.g., rendering the triangulated data).

Data objects: Data objects represent the actual data that flow through the visualization network. The basic data object in *VTK* is a dataset which is composed of cells. Cells are topological organizations of points and form the atoms of the dataset.

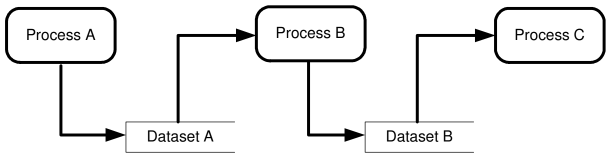


Figure 1: Process and data objects (Data-flow chart)

Building a visualization network involves setting up and connecting process and data objects. The process objects perform algorithmic operations on data as it flows through the network. The main advantage of this architecture is its flexibility and the ability to easily add new algorithms and data representations.

2.2 Process objects

Process objects can be one of three types: sources, filters, and mappers. Source objects are found at the beginning of the pipleline. They generate one or more output datasets. A source object may be a reader for a particular file type or it may even generate its own data, such as a sphere source. The output of this source object is then connected to the input of another process object. The act of connecting the input of one process object to the output of another process object is how the pipeline is built (Figure 1). For instance, to connect the output of filter A to the input of filter B, a construct of the following form is used: `B.SetInput(A.GetOutput());`

Each process object has only one output. However, fanning of the output is allowed since multiple process objects can set their input to be the output of the same process object. The pipeline terminates with mappers. A mapper "maps" its input to the screen (renders it). The mapper itself is one component of an object called a `vtkActor`. A `vtkActor` represents a geometrical object and its attributes. Other information in a `vtkActor` includes the object's appearance attributes (`vtkProperty`), and its location in space. As a result, the user will instantiate a `vtkActor` for each mapper in the *VTK* pipeline. The user will then set any attributes of each `vtkActor` and add each `vtkActor` to what is called a `vtkRenderWindow`. The `vtkRenderWindow` will then display all of its actors in a window on the screen.

Building *VTK* pipelines is simple, but one also needs to understand how a *VTK* pipeline executes. *VTK* uses a model that has an implicit control of execution. Execution only occurs when output is

*stefan.bruckner@chello.at

requested from an object (i.e., demand-driven). This scheme is implemented with two key methods: `Update()` and `Execute()`.

Each process object keeps track of its last modification time. If output is requested from the object and the object or its input has been modified since it last executed, then the process object should re-execute. The process of executing the pipeline begins with a call to a process object's `Update()` method. This typically happens when it is time to render and the mapper wants to update itself before rendering. The process object that receives the `Update()` message will recursively invoke the method on its input. This continues until a source object is encountered. The source object compares its modification time to the last time it executed. If it has been modified, then it will re-execute itself using the `Execute()` method. As mentioned previously, data objects "live" between process objects. If the source object has not been modified, then its output (a data object) is still valid. The recursion then backtracks with each successive process object comparing its input time and its own modification time with the time that it last executed. If necessary, the process object re-executes itself using the `Execute()` method. This process ends when control is returned to the object that initiated the `Update()` [4].

2.3 Data objects

2.3.1 Datasets

Data in the visualization pipeline are referred to as datasets. A dataset consists of an organizing structure and the associated attribute data. It has geometric and topological properties and consists of points and cells. Topology is the set of properties invariant under certain geometric transformations (here only rotation, translation, and non-uniform scaling are considered). Geometry is the specification of position - the instantiation of the topology. Topology is specified by cells while geometry is represented by points. Points are located where data is known and the cells are used for interpolation between these points. Datasets are characterized to whether their structure is regular or irregular. Regular means, that there is a single mathematical relationship within the composing points and cells. If a dataset's points are regular, we call the dataset regular. If the topological relationship of its cells is regular, then the topology of the dataset is regular. Regular data can be implicity represented, thus requiring less computational and memory resources than unstructured data.

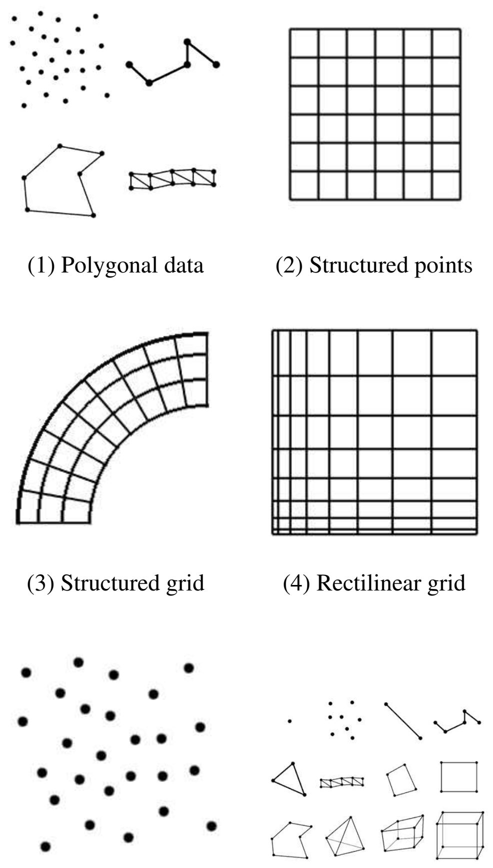
The toolkit provides the following dataset types (Figure 2):

Polygonal data: Polygonal data (1) are used frequently. Modern graphics hardware permits rendering these kind of data at very high speed, which makes them attractive for real-time visualization tasks. The topology and geometry of polygonal data is unstructured and the composing cells vary in topological dimension.

Structured points: Structured points datasets (2) are collections of points and cells arranged on a regular grid, a rectangular lattice parallel to the global coordinate system. If the points and cells are arranged on a plane they are referred to as an image or bitmap, if they are arranged as a stack of grids the dataset is called a volume.

Structured grids: Structured grid datasets (3) are regular in topology and irregular in geometry. The topology is represented implicitly by specifying a vector of dimensions. The geometry is represented explicitly by maintaining a list of point coordinates.

Rectilinear grids: A rectilinear grid dataset (4) is a collection of points and cells arranged on a regular lattice parallel to the global coordinate system. The dataset's topology is regular but its geometry is only partially regular (i.e., spacing between points may vary).


(1) Polygonal data (2) Structured points

(3) Structured grid (4) Rectilinear grid

(5) Unstructured points (6) Unstructured grid

Figure 2: Dataset types

Unstructured points: Unstructured points (5) are points irregularly located in space. Unstructured point datasets have no topology. Their geometry is completely unstructured.

Unstructured grids: An unstructured grid (6) is the most general form of a dataset. Topology and geometry are completely unstructured. Arbitrary combinations of all cell types are possible.

2.3.2 Cells

Cells can be thought of as the atoms that form a dataset. They form a topological organization of the dataset points. Cells are specified by a type in combination with a list of points (often referred to as connectivity list). The cell structure is a compact and general data structure for representing cell topology. Cell topology consists of points plus and a particular ordering of points (i.e., a cell). One or more cells may share a given point as well as other topological features such as edges and faces. The most important feature of the cell structure is that it represents adjacency, or topological neighborhood information, with minimal memory requirement. The cell structure has also been designed with access methods that support a wide variety of visualization algorithms [5].

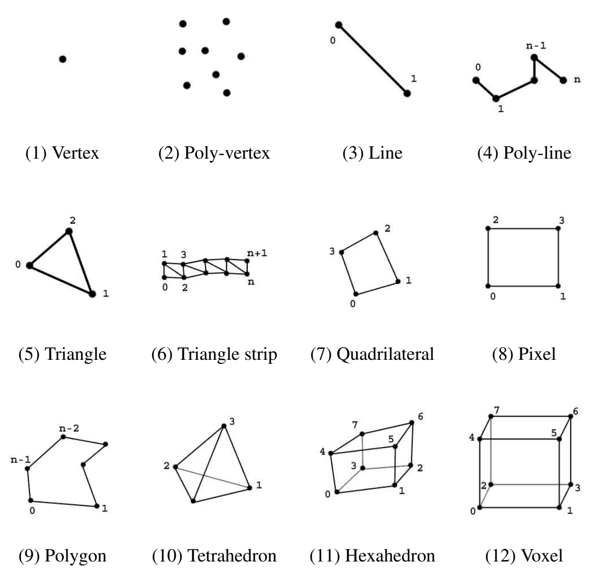


Figure 3: Cell types

The Visualization Toolkit supports the following cell types (Figure 3):

Vertex: A vertex (1) is a pimary zero-dimensional cell. It is defined by a single point.

Poly-vertex: A poly-vertex (2) is a composite zero-dimensional cell. It is defined by a list of points.

Line: A line (3) is a primary one-dimensional cell. It is defined by two points, the direction along the line is from the first to the second point.

Poly-line: A poly-line (4) is a composite one-dimensional cell. It consists of one or more connected lines. It is defined by a ordered list of points.

Triangle: A triangle (5) is a primary two-dimensional cell. A triangle is defined by a counter-clockwise ordered list of points.

Triangle strip: A triangle strip (6) is a composite two-dimensional cell consisting of one or more triangles. It is defined by an ordered list of points.

Quadrilateral: A quadrilateral (7) is a primary two-dimensional-cell. It is defined by a counter-clockwise ordered list of four points lying in a plane. It is convex and its edges must not intersect.

Pixel: A pixel (8) is a primary two-dimensional cell. It is defined by a ordered list of four points. In addition to a quadrilateral it has the geometric constraints that each edge of the pixel is perpendicular to its adjacent edges and that it lies parallel to one of the coordinate axes. Also, the ordering of the points is different from the quadrilateral cell. The points are ordered in the direction of increasing axis coordinate (starting with x, then y, then z).

Polygon: A polygon (9) is a primary two-dimensional cell. It is defined by a counter-clockwise ordered list of three or more points lying in a plane.

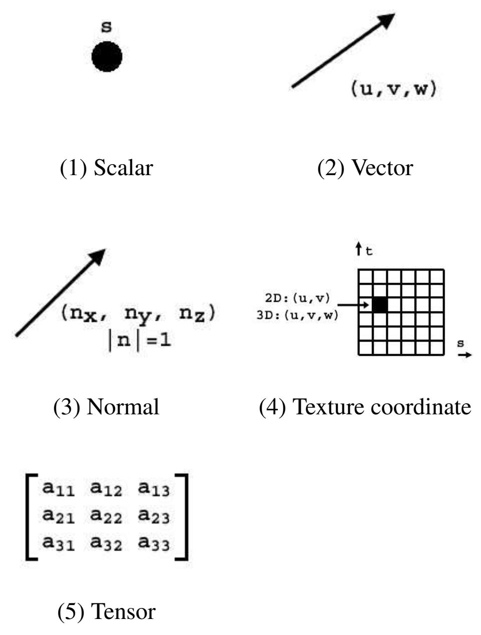


Figure 4: Attribute types

Tetrahedron: A tetrahedron (10) is a primary three-dimensional cell. It is defined by a list of four non-planar points.

Hexahedron: A hexahedron (11) is a primary three-dimensional cell. It is defined by an ordered list of eight points. It consists of six quadrilateral faces, twelve edges and eight vertices. The faces and edges must not intersect any other faces and edges, and the hexahedron must be convex.

Voxel: A voxel (12) is a primary three-dimensional cell. It is geometrically equivalent to the hexahedron with additional geometric constraints. Each face of the voxel is perpendicular to one of the coordinate axes. A voxel is defined by a list of points ordered in the direction of increasing coordinate value.

2.3.3 Attributes

In *VTK* data attributes can be associated with points, cell attributes are not supported explicitly. However, cell attributes can be represented by assigning the attribute value to each of the cell's points. Although this may result in some overhead in certain cases, this choice was made by the designers of *VTK* in order to simplify both implementation and usage.

The following attribute types are available (Figure 4):

Scalars: Scalar data (1) are single values at each location in the dataset. Scalars are used to represent temperature, density, pressure, etc.

Vectors: Vectors (2) have a magnitude and a direction. Examples of vector data include velocity and gradient function.

Normals: Normals (3) are vectors with a magnitude of 1. They are usually used to control the shading of objects.

Texture coordinates: Texture coordinates (4) are used to map a point from Cartesian space into texture space. They are regular arrays of color, intensity and/or transparency used for ren-

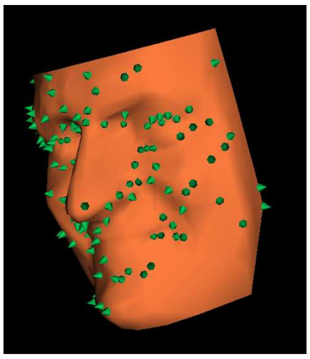


Figure 5: Resulting image for program 3.1

dering objects. Textures are used to add detail to objects without requiring a vast number of graphics primitives.

Tensors: Tensors (5) are mathematical generalizations of vectors and matrices. They are commonly used to represent stress and strain at a point in an object under load. In *VTK*, only real-valued, symmetric 3x3 tensors are suppored.

3 Examples

Program 3.1 provides an example for setting up the visualization pipeline. Polygonal input data is read from a file. Then the dataset is decimated using the `vtkDecimatePro` filter. Next, the polygonal data are smoothed using the `vtkSmoothPolyDataFilter`. Normals are then calculated with the `vtkPolyDataNormals` filter, which serves as an input for a `vtkPolyDataMapper`. A `vtkActor` is created, which uses this mapper. The output of the `vtkPolyDataNormals` also serves as an input for a `vtkGlyph3D` filter, which generates glyphs from a transformed `vtkCone` data source. The output of the `vtkGlyph3D` filter is then connected to a `vtkPolyDataMapper`, which is used by the `mySpikeActor` object. The result is an image where glyphs are used to depict surface normals (Figure 5). Figure 6 shows a data-flow chart for this example. Note that in this example, only polygonal data travels through the pipeline [6].

Program 3.2 shows a transformation between dataset types. Volumetric data is read from a file. A `vtkContourFilter` is used the extract the surface of the volume. It takes a `vtkStructuredPoints` dataset as input and outputs a `vtkPolyData` dataset. Figures 7 shows the resulting image [7], figure 8 depicts the data-flow chart for this example.

Usually, as shown in the previous examples, datasets are not accessed directly as filter outputs just serve as input for other filters. However, when implementing a filter, datasets and cells have to be manipulated. The next examples solely show how *VTK*'s dataset and cell interfaces are used and do not serve any purpose - all data values used in program 3.3 and 3.4 were chosen at random.

In program 3.3, first a structured grid dataset (`vtkStructuredGrid`) is created. Points are then inserted into the dataset. Next, attribute data is associated with the points. Scalar values are created and then assigned to the dataset's point data. The `GetPointData()` method returns a `vtkPointData` object instance for the grid. This class provides important capabilities: When a filter object executes, attribute data from its input is operated on and passed to its output. These operations typically involve copying input data from one point to an output point, interpolating input data to generate output data, or passing entire data objects from input to output. The `vtkPointData` class

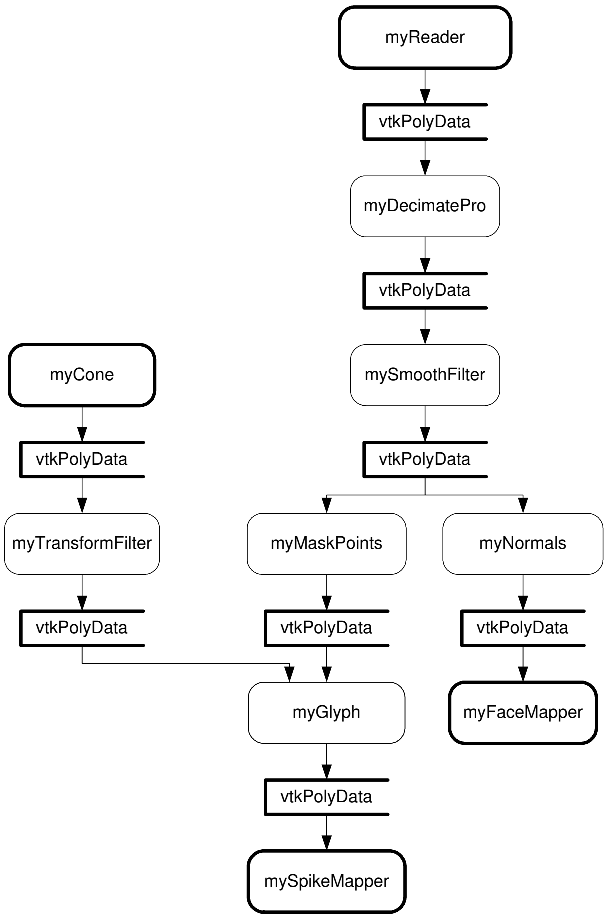


Figure 6: Data-flow chart for program 3.1

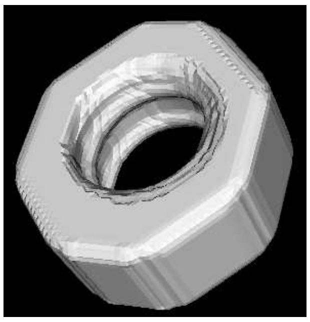


Figure 7: Resulting image for program 3.2

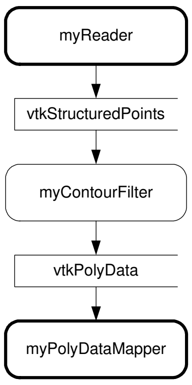


Figure 8: Data-flow chart for program 3.2

is used to prevent filters from directly accessing underlying data structures, thus making them independent of data representation.

Program 3.4 shows how to work with points and cells. We use the structured grid dataset created in the previous example. First, a cell is retrieved using its id. Each cell has a list of the points it contains. In our example, the cells are of type `vtkHexahedron` and therefore contain eight points. To retrieve all cells which contain a point, the point's id has to be looked up. This information is then used to retrieve the list of cell indices.

4 Conclusion

The design of computer systems demands careful attention to the balance between abstract and concrete systems. Visualization systems, in particular, must be carefully designed because they interface to other data systems and data models. The design of *VTK* is a well-balanced trade-off between design abstraction and simplicity, thus making it powerful enough to be used in applications, but also easy to use for prototyping of visualization methods.

References

[1] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 93–100, 1996.

[2] Craig Upson, Thomas A. Faulhaber, Jr., David Kamins, David Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The Application Visualization System: a computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.

[3] G. Abram and L. A. Treinish. An extended data-flow architecture for data analysis and visualization. In Gregory M. Nielson and Deborah E. Silver, editors, *IEEE Visualization '95*, pages 263–270, 461, 1995.

[4] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, second edition, 1998.

[5] William J. Schroeder and Boris Yamrom. A compact cell structure for scientific visualization. In *SIGGRAPH '94 Course Notes CD-ROM, Course 4: Advanced Techniques for Scientific Visualization*, pages 53–59. ACM SIGGRAPH, July 1994.

[6] William J. Schroeder, Lisa S. Avila, and William Hoffman. Visualizing with VTK: A tutorial. *IEEE Computer Graphics and Applications*, 20(5):20–27, September/October 2000.

[7] William J. Schroeder, Lisa S. Avila, Kenneth M. Martin, William A. Hoffmann, and C. Charles Law. *The Visualization Toolkit User's Guide*. Kitware, first edition, 1998.

Program 3.1 Creating a decimated, smoothed, glyphed polygonal mesh (Java code extract)

```
(...)

// Read polygonal data from file
vtkPolyDataReader myReader = new vtkPolyDataReader();
myReader.SetFileName("fran_cut.vtk");

// Create and setup the decimation filter
vtkDecimatePro myDecimatePro = new vtkDecimatePro();
myDecimatePro.SetInput(myReader.GetOutput());
myDecimatePro.SetTargetReduction(0.9);
myDecimatePro.PreserveTopologyOn();

// Create and setup the smoothing filter
vtkSmoothPolyDataFilter mySmoothFilter =
    new vtkSmoothPolyDataFilter();
mySmoothFilter.SetInput(myDecimatePro.GetOutput());

// Create and setup normal calculation
vtkPolyDataNormals myNormals =
    new vtkPolyDataNormals();
myNormals.SetInput(mySmoothFilter.GetOutput());
myNormals.SetFeatureAngle(60.0);

// Create and setup first mapper
vtkPolyDataMapper myFaceMapper = new vtkPolyDataMapper();
myFaceMapper.SetInput(myNormals.GetOutput());

// Create first actor and set its mapper and color
vtkActor myFaceActor = new vtkActor();
myFaceActor.SetMapper(myFaceMapper);
myFaceActor.GetProperty().SetColor(1.0,0.49,0.25);

// Create and setup masked points
vtkMaskPoints myMaskPoints = new vtkMaskPoints();
myMaskPoints.SetInput(myNormals.GetOutput());
myMaskPoints.SetOnRatio(5);
myMaskPoints.RandomModeOn();

// Create a cone data source
vtkConeSource myCone = new vtkConeSource();
myCone.SetResolution(6);

// Create a transformation used by the transform filter
vtkTransform myTransform = new vtkTransform();
myTransform.Translate(0.5,0.0,0.0);

// Create and setup transform filter
vtkTransformPolyDataFilter myTransformFilter =
    new vtkTransformPolyDataFilter();
myTransformFilter.SetInput(myCone.GetOutput());
myTransformFilter.SetTransform(myTransform);

// Create and setup glyph generation
vtkGlyph3D myGlyph = new vtkGlyph3D();
myGlyph.SetInput(myMaskPoints.GetOutput());
myGlyph.SetSource(myTransformFilter.GetOutput());
myGlyph.SetVectorModeToUseNormal();
myGlyph.SetScaleModeToScaleByVector();
myGlyph.SetScaleFactor(-0.005);

// Create and setup second mapper
vtkPolyDataMapper mySpikeMapper = new vtkPolyDataMapper();
mySpikeMapper.SetInput(myGlyph.GetOutput());

// Create second actor and set its mapper and color
vtkActor mySpikeActor = new vtkActor();
mySpikeActor.SetMapper(mySpikeMapper);
mySpikeActor.GetProperty().SetColor(0.0,0.79,0.34);

(...)
```

Program 3.2 Extracting a volume's surface (Java code extract)

```
(...)

// Read volume data
vtkSLCReader myReader = new vtkSLCReader();
myReader.SetFileName("nut.slc");

// Create and initialize contour filter
vtkContourFilter  myContourFilter  =
    new vtkContourFilter ();
myContourFilter.SetInput(myReader.GetOutput());
myContourFilter.SetValue(0,16.0);

// Create and initialize polygon mapper filter
vtkPolyDataMapper myPolyDataMapper = new vtkPolyDataMapper();
myPolyDataMapper.SetInput(myContourFilter.GetOutput());
myPolyDataMapper.ScalarVisibilityOff();

// Create an actor and set it's mapper and color
vtkActor myActor = new vtkActor();
myActor.SetMapper(myPolyDataMapper);
myActor.GetProperty().SetColor(1.0,1.0,1.0);

(...)
```

Program 3.3 Creating a structured grid with attribute data (Java code extract)

```
(...)

int i=0, j=0, k=0, l=0, m=0;

// Create a structured grid
vtkStructuredGrid myGrid = new vtkStructuredGrid();
myGrid.SetDimensions(20,20,20);

// Create a vtkPoints object
vtkPoints myPoints = new vtkPoints();

// Insert points
for (k = 0; k < 20; k++)
    for (j = 0; j < 20; j++)
        for (i = 0; i < 20; i++)
            myPoints.InsertNextPoint(
                (double) i,
                (double) j,
                (double) k);

// Associate points with the grid
myGrid.SetPoints(myPoints);

// Create a scalar array
vtkShortArray myScalarArray = new vtkShortArray();
myScalarArray.SetNumberOfComponents(3);
myScalarArray.SetNumberOfTuples(20*20*20);

// Insert scalar values
for (k = 0; k < 20; k++)
    for (j = 0; j < 20; j++)
        for (i = 0; i < 20; i++)
        {
            myScalarArray.InsertComponent(l,0,i);
            myScalarArray.InsertComponent(l,0,j);
            myScalarArray.InsertComponent(l,0,k);
            l++;
        }

// Associate scalar values with the points in the grid
myGrid.GetPointData().SetScalars(myScalarArray);

// Output grid data
System.out.println("myGrid:");
System.out.println(myGrid.Print());

(...)
```

Program 3.4 Accessing cells and points (Java code extract)

```
(...)

// Get a cell by id
i = 10; j = 15; k = 7;
int myCellId = k * (19 * 19) + j * 19 + i;
vtkCell myCell= myGrid.GetCell(myCellId);

// Output cell data
System.out.println("myCell:");
System.out.println(myCell.Print());

// Retrieves the cell's point list
vtkIdList myPointIds = new vtkIdList();
myGrid.GetCellPoints(myCellId, myPointIds);

// Iterate through the list of point ids
for (m = 0; m < myPointIds.GetNumberOfIds(); m++)
{
    // Retrieve a single point from list
    double[] myCellPoint = myGrid.GetPoint(
        myPointIds.GetId(m));

    // Output point data
    System.out.println("myCellPoint(" + m + ")");
    System.out.println(
        "x = " + myCellPoint[0] + ", " +
        "y = " + myCellPoint[1] + ", " +
        "z = " + myCellPoint[2]);
    System.out.println();
}

// Look up point id
double[] myPoint = new double[3];
myPoint[0] = 10.5;
myPoint[1] = 12.1;
myPoint[2] = 14.7;
int myPointId = myGrid.FindPoint(myPoint);

// Retrieve list of cells containing the point
vtkIdList myCellIds = new vtkIdList();
myGrid.GetPointCells(myPointId,myCellIds);

// Iterate through the list of cell ids
for (m = 0; m < myCellIds.GetNumberOfIds();m++)
{
    // Retrieve single cell from list
    vtkCell myPointCell = myGrid.GetCell(
        myCellIds.GetId(m));

    // Output cell data
    System.out.println("myPointCell(" + m + ")");
    System.out.println(myPointCell.Print());
}

(...)
```