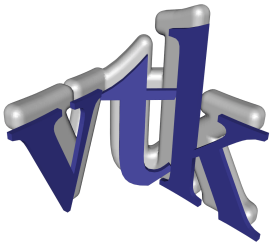


File Formats

for VTK Version 4.2

(Taken from The VTK User's Guide
Contact Kitware www.kitware.com to purchase)

VTK File Formats

The *Visualization Toolkit* provides a number of source and writer objects to read and write popular data file formats. The *Visualization Toolkit* also provides some of its own file formats. The main reason for creating yet another data file format is to offer a consistent data representation scheme for a variety of dataset types, and to provide a simple method to communicate data between software. Whenever possible, we recommend that you use formats that are more widely used. But if this is not possible, the *Visualization Toolkit* formats described here can be used instead. Note that these formats may not be supported by many other tools.

There are two different styles of file formats available in VTK. The simplest are the legacy, serial formats that are easy to read and write either by hand or programmatically. However, these formats are less flexible than the XML based file formats described later in this section. The XML formats support random access, parallel I/O, and portable data compression and are preferred to the serial VTK file formats whenever possible.

Simple Legacy Formats

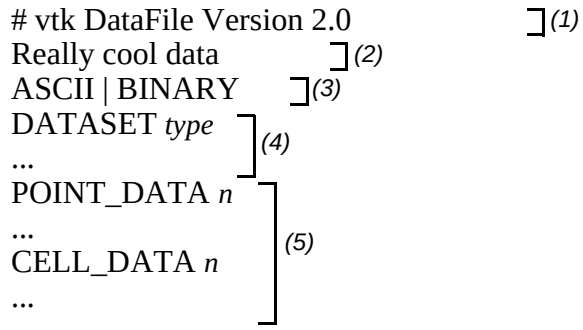
The legacy VTK file formats consist of five basic parts.

1. The first part is the file version and identifier. This part contains the single line: `# vtk DataFile Version x.x`. This line must be exactly as shown with the exception of the version number `x.x`, which will vary with different releases of VTK. (Note: the current version number is 3.0. Version 1.0 and 2.0 files are compatible with version 3.0 files.)
2. The second part is the header. The header consists of a character string terminated by end-of-line character `\n`. The header is 256 characters maximum. The header can be used to describe the data and include any other pertinent information.
3. The next part is the file format. The file format describes the type of file, either ASCII or binary. On this line the single word `ASCII` or `BINARY` must appear.
4. The fourth part is the dataset structure. The geometry part describes the geometry and topology of the dataset. This part begins with a line containing the keyword `DATASET` followed by a keyword describing the type of dataset. Then, depending upon the type of dataset, other keyword/data combinations define the actual data.
5. The final part describes the dataset attributes. This part begins with the keywords `POINT_DATA` or `CELL_DATA`, followed by an integer number specifying the number of points or cells, respectively. (It doesn't matter whether `POINT_DATA` or `CELL_DATA` comes first.) Other keyword/data combinations then define the actual dataset attribute values (i.e., scalars, vectors, tensors, normals, texture coordinates, or field data).

An overview of the file format is shown in **Figure 1**. The first three parts are mandatory, but the other two are optional. Thus you have the flexibility of mixing and matching dataset attributes and geometry, either by operating system file manipulation or using VTK filters to merge data. Keywords are case insensitive, and may be separated by whitespace.

Before describing the data file formats please note the following.

- *dataType* is one of the types `bit`, `unsigned_char`, `char`, `unsigned_short`, `short`, `unsigned_int`, `int`, `unsigned_long`, `long`, `float`, or `double`. These keywords are used to describe the form of the data, both for reading from file, as well as constructing the appropriate internal objects. Not all data types are supported for all classes.



Part 1: Header

Part 2: Title (256 characters maximum, terminated with newline `\n` character)

Part 3: Data type, either ASCII or BINARY

Part 4: Geometry/topology. *Type* is one of:

```
STRUCTURED_POINTS
STRUCTURED_GRID
UNSTRUCTURED_GRID
POLYDATA
RECTILINEAR_GRID
FIELD
```

Part 5: Dataset attributes. The number of data items *n* of each type must match the number of points or cells in the dataset. (If *type* is `FIELD`, point and cell data should be omitted.

Figure 1 Overview of five parts of VTK data file format.

- All keyword phrases are written in ASCII form whether the file is binary or ASCII. The binary section of the file (if in binary form) is the data proper; i.e., the numbers that define points coordinates, scalars, cell indices, and so forth.
- Indices are 0-offset. Thus the first point is point id 0.
- If both the data attribute and geometry/topology part are present in the file, then the number of data values defined in the data attribute part must exactly match the number of points or cells defined in the geometry/topology part.
- Cell types and indices are of type `int`.
- Binary data must be placed into the file immediately after the "newline" (`\n`) character from the previous ASCII keyword and parameter sequence.
- The geometry/topology description must occur prior to the data attribute description.

Binary Files. Binary files in VTK are portable across different computer systems as long as you observe two conditions. First, make sure that the byte ordering of the data is correct, and second, make sure that the length of each data type is consistent.

Most of the time VTK manages the byte ordering of binary files for you. When you write a binary file on one computer and read it in from another computer, the bytes representing the data will be automatically swapped as necessary. For example, binary files written on a Sun are stored in big endian order, while those on a PC are stored in little endian order. As a result, files written on a Sun workstation require byte swapping when read on a PC. (See the class vtkByteSwap for implementation details.) The VTK data files described here are written in big endian form.

Some file formats, however, do not explicitly define a byte ordering form. You will find that data read or written by external programs, or the classes vtkVolume16Reader, vtkMCubesReader, and vtkMCubesWriter may have a different byte order depending on the system of origin. In such cases, VTK allows you to specify the byte order by using the methods

```
SetDataByteOrderToBigEndian()
SetDataByteOrderToLittleEndian()
```

Another problem with binary files is that systems may use a different number of bytes to represent an integer or other native type. For example, some 64-bit systems will represent an integer with 8-bytes, while others represent an integer with 4-bytes. Currently, the *Visualization Toolkit* cannot handle transporting binary files across systems with incompatible data length. In this case, use ASCII file formats instead.

Dataset Format. The *Visualization Toolkit* supports five different dataset formats: structured points, structured grid, rectilinear grid, unstructured grid, and polygonal data. Data with implicit topology (structured data such as vtkImageData and vtkStructuredGrid) are ordered with *x* increasing fastest, then *y*, then *z*. These formats are as follows.

- Structured Points
  The file format supports 1D, 2D, and 3D structured point datasets. The dimensions n_x, n_y, n_z must be greater than or equal to 1. The data spacing s_x, s_y, s_z must be greater than 0. (Note: in the version 1.0 data file, spacing was referred to as "aspect ratio". `ASPECT_RATIO` can still be used in version 2.0 data files, but is discouraged.)

  `DATASET STRUCTURED_POINTS`
  `DIMENSIONS` n_x n_y n_z
  `ORIGIN` *x y z*
  `SPACING` s_x s_y s_z

- Structured Grid
  The file format supports 1D, 2D, and 3D structured grid datasets. The dimensions n_x, n_y, n_z must be greater than or equal to 1. The point coordinates are defined by the data in the `POINTS` section. This consists of *x-y-z* data values for each point.

  `DATASET STRUCTURED_GRID`
  `DIMENSIONS` n_x n_y n_z
  `POINTS` *n dataType*
  p_{0x} p_{0y} p_{0z}
  p_{1x} p_{1y} p_{1z}
  ...
  $p_{(n-1)x}$ $p_{(n-1)y}$ $p_{(n-1)z}$

- Rectilinear Grid
  A rectilinear grid defines a dataset with regular topology, and semi-regular geometry aligned along the *x-y-z* coordinate axes. The geometry is defined by three lists of monotonically increasing coordinate values, one list for each of the *x-y-z* coordinate axes. The topology is defined by specifying the grid dimensions, which must be greater than or equal to 1.

  `DATASET RECTILINEAR_GRID`
  `DIMENSIONS` n_x n_y n_z
  `X_COORDINATES` n_x *dataType*
  x_0 x_1 ... $x_{(nx-1)}$
  `Y_COORDINATES` n_y *dataType*
  y_0 y_1 ... $y_{(ny-1)}$
  `Z_COORDINATES` n_z *dataType*
  z_0 z_1 ... $z_{(nz-1)}$

- Polygonal Data
  The polygonal dataset consists of arbitrary combinations of surface graphics primitives vertices (and polyvertices), lines (and polylines), polygons (of various types), and triangle strips. Polygonal data is defined by the `POINTS` `VERTICES`, `LINES`, `POLYGONS`, or `TRIANGLE_STRIPS` sections. The `POINTS` definition is the same as we saw for structured grid datasets. The `VERTICES`, `LINES`, `POLYGONS`, or `TRIANGLE_STRIPS` keywords define the polygonal dataset topology. Each of these keywords requires two parameters: the number of cells *n* and the size of the cell list *size*. The cell list size is the total number of integer values required to represent the list (i.e., sum of *numPoints* and

connectivity indices over each cell). None of the keywords `VERTICES`, `LINES`, `POLYGONS`, or `TRIANGLE_STRIPS` is required.

`DATASET POLYDATA`
`POINTS` *n dataType*
$p_{0x}\ p_{0y}\ p_{0z}$
$p_{1x}\ p_{1y}\ p_{1z}$
...
$p_{(n-1)x}\ p_{(n-1)y}\ p_{(n-1)z}$

`VERTICES` *n size*
$numPoints_0, i_0, j_0, k_0, ...$
$numPoints_1, i_1, j_1, k_1, ...$
...
$numPoints_{n-1}, i_{n-1}, j_{n-1}, k_{n-1}, ...$

`LINES` *n size*
$numPoints_0, i_0, j_0, k_0, ...$
$numPoints_1, i_1, j_1, k_1, ...$
...
$numPoints_{n-1}, i_{n-1}, j_{n-1}, k_{n-1}, ...$

`POLYGONS` *n size*
$numPoints_0, i_0, j_0, k_0, ...$
$numPoints_1, i_1, j_1, k_1, ...$
...
$numPoints_{n-1}, i_{n-1}, j_{n-1}, k_{n-1}, ...$

`TRIANGLE_STRIPS` *n size*
$numPoints_0, i_0, j_0, k_0, ...$
$numPoints_1, i_1, j_1, k_1, ...$
...
$numPoints_{n-1}, i_{n-1}, j_{n-1}, k_{n-1}, ...$

- Unstructured Grid
  The unstructured grid dataset consists of arbitrary combinations of any possible cell type. Unstructured grids are defined by points, cells, and cell types. The `CELLS` keyword requires two parameters: the number of cells *n* and the size of the cell list *size*. The cell list size is the total number of integer values required to represent the list (i.e., sum of *numPoints* and connectivity indices over each cell). The `CELL_TYPES` keyword requires a single parameter: the number of cells *n*. This value should match the value specified by the `CELLS` keyword. The cell types data is a single integer value per cell that specified cell type (see `vtkCell.h` or **Figure 2**).

  `DATASET UNSTRUCTURED_GRID`
  `POINTS` *n dataType*
  $p_{0x}\ p_{0y}\ p_{0z}$
  $p_{1x}\ p_{1y}\ p_{1z}$
  ...
  $p_{(n-1)x}\ p_{(n-1)y}\ p_{(n-1)z}$

  `CELLS` *n size*
  $numPoints_0, i, j, k, l, ...$
  $numPoints_1, i, j, k, l, ...$
  $numPoints_2, i, j, k, l, ...$
  ...
  $numPoints_{n-1}, i, j, k, l, ...$

`CELL_TYPES` *n*
type$_0$
type$_1$
type$_2$
...
type$_{n-1}$

- Field
  Field data is a general format without topological and geometric structure, and without a particular dimensionality. Typically field data is associated with the points or cells of a dataset. However, if the `FIELD` *type* is specified as the dataset type (see **Figure 1**), then a general VTK data object is defined. Use the format described in the next section to define a field. Also see "Working With Field Data" on page 158 and the fourth example in this chapter "Examples" on page 7.

Dataset Attribute Format. The *Visualization Toolkit* supports the following dataset attributes: scalars (one to four components), vectors, normals, texture coordinates (1D, 2D, and 3D), 3×3 tensors, and field data. In addition, a lookup table using the RGBA color specification, associated with the scalar data, can be defined as well. Dataset attributes are supported for both points and cells.

Each type of attribute data has a *dataName* associated with it. This is a character string (without embedded whitespace) used to identify a particular data. The *dataName* is used by the VTK readers to extract data. As a result, more than one attribute data of the same type can be included in a file. For example, two different scalar fields defined on the dataset points, pressure and temperature, can be contained in the same file. (If the appropriate *dataName* is not specified in the VTK reader, then the first data of that type is extracted from the file.)

- Scalars
  Scalar definition includes specification of a lookup table. The definition of a lookup table is optional. If not specified, the default VTK table will be used (and *tableName* should be "`default`"). Also note that the *numComp* variable is optional—by default the number of components is equal to one. (The parameter *numComp* must range between (1,4) inclusive; in versions of VTK prior to vtk2.3 this parameter was not supported.)

  `SCALARS` *dataName dataType numComp*
  `LOOKUP_TABLE` *tableName*
  s$_0$
  s$_1$
  ...
  s$_{n-1}$

  The definition of color scalars (i.e., `unsigned char` values directly mapped to color) varies depending upon the number of values (*nValues*) per scalar. If the file format is `ASCII`, the color scalars are defined using *nValues* `float` values between (0,1). If the file format is `BINARY`, the stream of data consists of *nValues* `unsigned char` values per scalar value.

  `COLOR_SCALARS` *dataName nValues*
  $c_{00}\ c_{01} \cdots c_{0(nValues-1)}$
  $c_{10}\ c_{11} \cdots c_{1(nValues-1)}$
  ...
  $c_{(n-1)0}\ c_{(n-1)1} \cdots c_{(n-1)(nValues-1)}$

- Lookup Table
  The *tableName* field is a character string (without imbedded white space) used to identify the lookup table. This label is used by the VTK reader to extract a specific table.

  Each entry in the lookup table is a `rgba[4]` (*red-green-blue-alpha*) array (*alpha* is opacity where *alpha=0* is transparent). If the file format is `ASCII`, the lookup table values must be `float` values between (0,1). If the file format is

BINARY, the stream of data must be four unsigned char values per table entry.

LOOKUP_TABLE *tableName size*
$r_0\ g_0\ b_0\ a_0$
$r_1\ g_1\ b_1\ a_1$
...
$r_{size-1}\ g_{size-1}\ b_{size-1}\ a_{size-1}$

- Vectors

VECTORS *dataName dataType*
$v_{0x}\ v_{0y}\ v_{0z}$
$v_{1x}\ v_{1y}\ v_{1z}$
...
$v_{(n-1)x}\ v_{(n-1)y}\ v_{(n-1)z}$

- Normals
Normals are assumed normalized $|n| = 1$.

NORMALS *dataName dataType*
$n_{0x}\ n_{0y}\ n_{0z}$
$n_{1x}\ n_{1y}\ n_{1z}$
...
$n_{(n-1)x}\ n_{(n-1)y}\ n_{(n-1)z}$

- Texture Coordinates
Texture coordinates of 1, 2, and 3 dimensions are supported.

TEXTURE_COORDINATES *dataName dim dataType*
$t_{00}\ t_{01}\ \ldots\ t_{0(dim-1)}$
$t_{10}\ t_{11}\ \ldots\ t_{1(dim-1)}$
...
$t_{(n-1)0}\ t_{(n-1)1}\ \ldots\ t_{(n-1)(dim-1)}$

- Tensors
Currently only 3×3 real-valued, symmetric tensors are supported.

TENSORS *dataName dataType*
$t^0_{00}\ t^0_{01}\ t^0_{02}$
$t^0_{10}\ t^0_{11}\ t^0_{12}$
$t^0_{20}\ t^0_{21}\ t^0_{22}$

$t^1_{00}\ t^1_{01}\ t^1_{02}$
$t^1_{10}\ t^1_{11}\ t^1_{12}$
$t^1_{20}\ t^1_{21}\ t^1_{22}$

...
$t^{n-1}_{00}\ t^{n-1}_{01}\ t^{n-1}_{02}$
$t^{n-1}_{10}\ t^{n-1}_{11}\ t^{n-1}_{12}$
$t^{n-1}_{20}\ t^{n-1}_{21}\ t^{n-1}_{22}$

- Field Data
Field data is essentially an array of data arrays. Defining field data means giving a name to the field and specifying the number of arrays it contains. Then, for each array, the name of the array *arrayName(i)*, the number of components of the array, *numComponents*, the number of tuples in the array, *numTuples*, and the data type, *dataType*, are

defined.

FIELD *dataName numArrays*
arrayName0 numComponents numTuples dataType
$f_{00}\ f_{01} \cdots f_{0(numComponents-1)}$
$f_{10}\ f_{11} \cdots f_{1(numComponents-1)}$
...
$f_{(numTuples-1)0}\ f_{(numTuples-1)1} \cdots f_{(numTuples-1)(numComponents-1)}$

arrayName1 numComponents numTuples dataType
$f_{00}\ f_{01} \cdots f_{0(numComponents-1)}$
$f_{10}\ f_{11} \cdots f_{1(numComponents-1)}$
...
$f_{(numTuples-1)0}\ f_{(numTuples-1)1} \cdots f_{(numTuples-1)(numComponents-1)}$

...
arrayName(numArrays-1) numComponents numTuples dataType
$f_{00}\ f_{01} \cdots f_{0(numComponents-1)}$
$f_{10}\ f_{11} \cdots f_{1(numComponents-1)}$
...
$f_{(numTuples-1)0}\ f_{(numTuples-1)1} \cdots f_{(numTuples-1)(numComponents-1)}$

Examples. The first example is a cube represented by six polygonal faces. We define a single-component scalar, normals, and field data on the six faces. There are scalar data associated with the eight vertices. A lookup table of eight colors, associated with the point scalars, is also defined.

```
# vtk DataFile Version 2.0
Cube example
ASCII
DATASET POLYDATA
POINTS 8 float
0.0 0.0 0.0
1.0 0.0 0.0
1.0 1.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
1.0 0.0 1.0
1.0 1.0 1.0
0.0 1.0 1.0
POLYGONS 6 30
4 0 1 2 3
4 4 5 6 7
4 0 1 5 4
4 2 3 7 6
4 0 4 7 3
4 1 2 6 5

CELL_DATA 6
SCALARS cell_scalars int 1
LOOKUP_TABLE default
0
1
2
3
4
5
NORMALS cell_normals float
```

```
0 0 -1
0 0 1
0 -1 0
0 1 0
-1 0 0
1 0 0
FIELD FieldData 2
cellIds 1 6 int
0 1 2 3 4 5
faceAttributes 2 6 float
0.0 1.0 1.0 2.0 2.0 3.0 3.0 4.0 4.0 5.0 5.0 6.0

POINT_DATA 8
SCALARS sample_scalars float 1
LOOKUP_TABLE my_table
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
LOOKUP_TABLE my_table 8
0.0 0.0 0.0 1.0
1.0 0.0 0.0 1.0
0.0 1.0 0.0 1.0
1.0 1.0 0.0 1.0
0.0 0.0 1.0 1.0
1.0 0.0 1.0 1.0
0.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0
```

The next example is a volume of dimension $3 \times 4 \times 5$. Since no lookup table is defined, either the user must create one in VTK, or the default lookup table will be used.

```
# vtk DataFile Version 2.0
Volume example
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 3 4 6
ASPECT_RATIO 1 1 1
ORIGIN 0 0 0
POINT_DATA 72
SCALARS volume_scalars char 1
LOOKUP_TABLE default
0 0 0 0 0 0 0 0 0 0 0 0
0 5 10 15 20 25 25 20 15 10 5 0
0 10 20 30 40 50 50 40 30 20 10 0
0 10 20 30 40 50 50 40 30 20 10 0
0 5 10 15 20 25 25 20 15 10 5 0
0 0 0 0 0 0 0 0 0 0 0 0
```

The third example is an unstructured grid containing twelve of the nineteen VTK cell types (see **Figure 2** and **Figure 3**). The file contains scalar and vector data.

```
# vtk DataFile Version 2.0
Unstructured Grid Example
ASCII
```

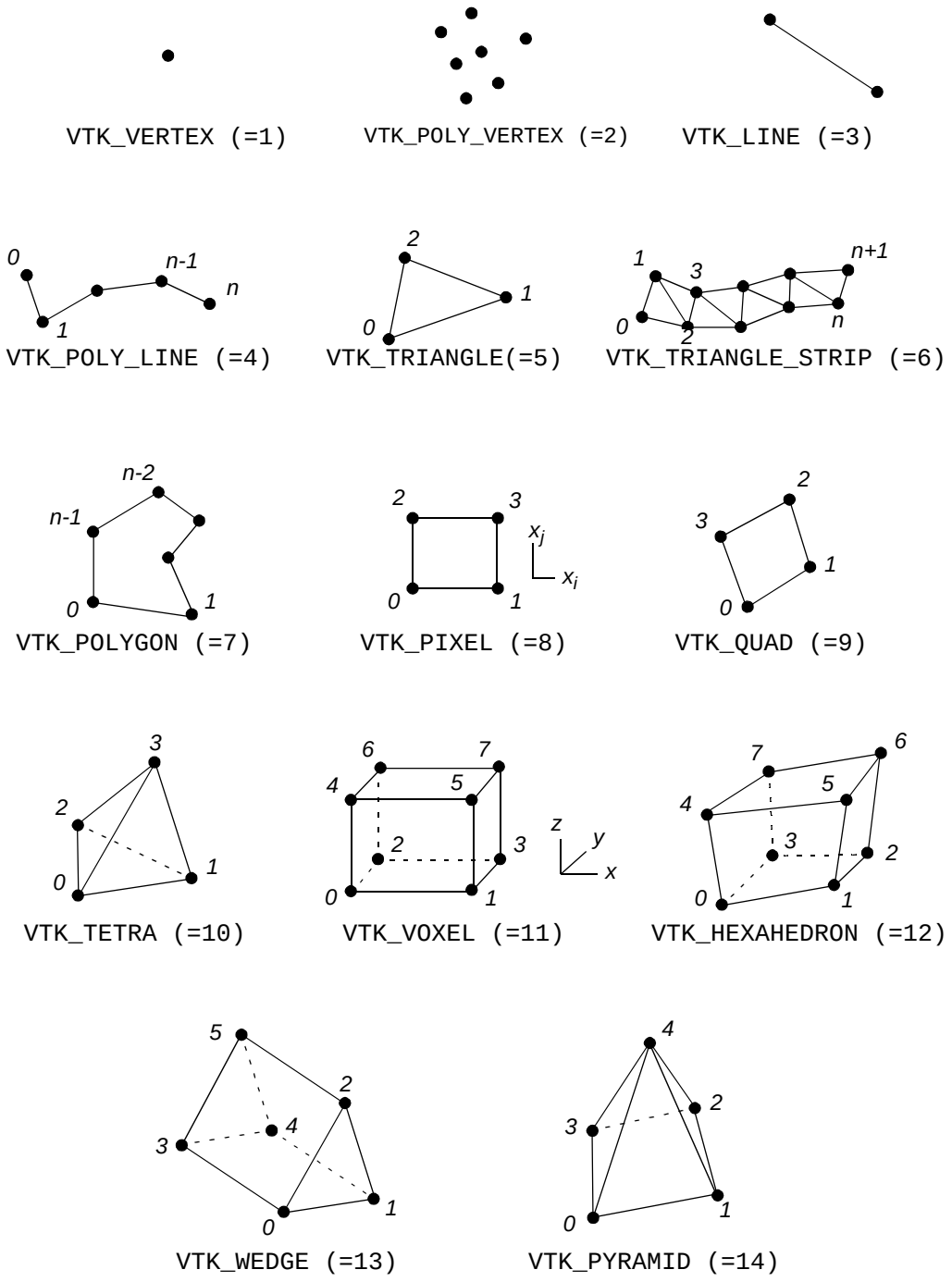


Figure 2 Linear cell types found in VTK. Use the include file CellType.h to manipulate cell types.

```
DATASET UNSTRUCTURED_GRID
POINTS 27 float
0 0 0   1 0 0   2 0 0   0 1 0   1 1 0   2 1 0
0 0 1   1 0 1   2 0 1   0 1 1   1 1 1   2 1 1
0 1 2   1 1 2   2 1 2   0 1 3   1 1 3   2 1 3
0 1 4   1 1 4   2 1 4   0 1 5   1 1 5   2 1 5
0 1 6   1 1 6   2 1 6

CELLS 11 60
```

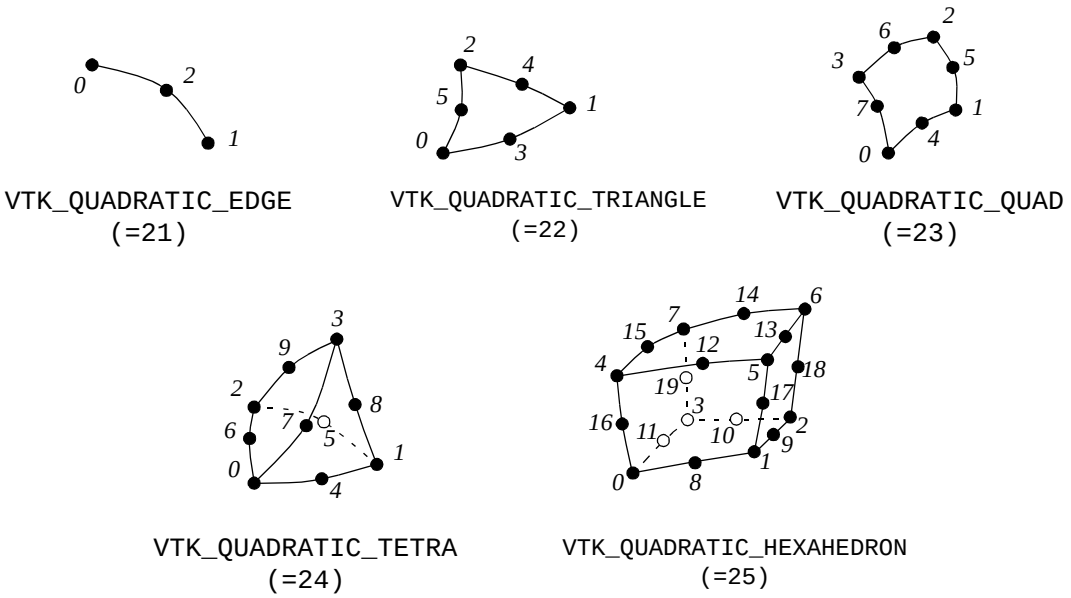


Figure 3 Non-linear cell types found in VTK.

```
8 0 1 4 3 6 7 10 9
8 1 2 5 4 7 8 11 10
4 6 10 9 12
4 5 11 10 14
6 15 16 17 14 13 12
6 18 15 19 16 20 17
4 22 23 20 19
3 21 22 18
3 22 19 18
2 26 25
1 24

CELL_TYPES 11
12
12
10
10
7
6
9
5
5
3
1


POINT_DATA 27
SCALARS scalars float 1
LOOKUP_TABLE default
0.0   1.0   2.0   3.0   4.0   5.0
6.0   7.0   8.0   9.0   10.0   11.0
12.0   13.0   14.0   15.0   16.0   17.0
18.0   19.0   20.0   21.0   22.0   23.0
24.0   25.0   26.0
VECTORS vectors float
1 0 0   1 1 0   0 2 0   1 0 0   1 1 0   0 2 0
```

```
1 0 0   1 1 0   0 2 0   1 0 0   1 1 0   0 2 0
0 0 1   0 0 1   0 0 1   0 0 1   0 0 1   0 0 1
0 0 1   0 0 1   0 0 1   0 0 1   0 0 1   0 0 1
0 0 1   0 0 1   0 0 1
```

The fourth and final example is data represented as a field. You may also wish to see “Working With Field Data” on page 158 to see how to manipulate this data. (The data file shown below can be found in its entirety in `$VTK_DATA_ROOT/Data/financial.vtk`.)

```
# vtk DataFile Version 2.0
Financial data in vtk field format
ASCII
FIELD financialData 6
TIME_LATE 1 3188 float
 29.14   0.00   0.00  11.71   0.00   0.00   0.00   0.00
...(more stuff – 3188 total values)...

MONTHLY_PAYMENT 1 3188 float
  7.26   5.27   8.01  16.84   8.21  15.75  10.62  15.47
  ...(more stuff)...

UNPAID_PRINCIPLE 1 3188 float
  430.70   380.88   516.22  1351.23   629.66  1181.97   888.91  1437.83
  ...(more stuff)...

LOAN_AMOUNT 1 3188 float
  441.50   391.00   530.00  1400.00   650.00  1224.00   920.00  1496.00
  ...(more stuff)...

INTEREST_RATE 1 3188 float
  13.875 13.875 13.750 11.250 11.875 12.875 10.625 10.500
  ...(more stuff)...

MONTHLY_INCOME 1 3188 unsigned_short
 39  51  51  38  35  49  45  56
  ...(more stuff)...
```

In this example, a field is represented using six arrays. Each array has a single component and 3,188 tuples. Five of the six arrays are of type `float`, while the last array is of type `unsigned_short`.
Additional examples are available in the data directory.

XML File Formats

VTK provides another set of data formats using XML syntax. While these formats are much more complicated than the original VTK format described previously (see “Simple Legacy Formats” on page 1), they support many more features. The major motivation for their development was to facilitate data streaming and parallel I/O. Some features of the format include support for compression, portable binary encoding, random access, big endian and little endian byte order, multiple file representation of piece data, and new file extensions for different VTK dataset types. XML provides many features as well, especially the ability to extend a file format with application specific tags.
There are two types of VTK XML data files: parallel and serial as described in the following.

- **Serial.** File types designed for reading and writing by applications of only a single process. All of the data are contained within a single file.
- **Parallel.** File types designed for reading and writing by applications with multiple processes executing in parallel. The dataset is broken into pieces. Each process is assigned a piece or set of pieces to read or write. An individual piece is stored in a corresponding serial file type. The parallel file type does not actually contain any data, but instead describes structural information and then references other serial files containing the data for each piece.

In the XML format, VTK datasets are classified into one of two categories.

- **Structured.** The dataset is a topologically regular array of cells such as pixels and voxels (e.g., image data) or quadrilaterals and hexahedra (e.g., structured grid) (see “The Visualization Model” on page 19 for more information). Rectangular subsets of the data are described through extents. The structured dataset types are vtkImageData, vtkRectilinearGrid, and vtkStructuredGrid.
- **Unstructured.** The dataset forms a topologically irregular set of points and cells. Subsets of the data are described using pieces. The unstructured dataset types are vtkPolyData and vtkUnstructuredGrid (see “The Visualization Model” on page 19 for more information).

By convention, each data type and file type is paired with a particular file extension. The types and corresponding extensions are

- ImageData (`.vti`) — Serial vtkImageData (structured).
- PolyData (`.vtp`) — Serial vtkPolyData (unstructured).
- RectilinearGrid (`.vtr`) — Serial vtkRectilinearGrid (structured).
- StructuredGrid (`.vts`) — Serial vtkStructuredGrid (structured).
- UnstructuredGrid (`.vtu`) — Serial vtkUnstructuredGrid (unstructured).
- PImageData (`.pvti`) — Parallel vtkImageData (structured).
- PPolyData (`.pvtp`) — Parallel vtkPolyData (unstructured).
- PRectilinearGrid (`.pvtr`) — Parallel vtkRectilinearGrid (structured).
- PStructuredGrid (`.pvts`) — Parallel vtkStructuredGrid (structured).
- PUnstructuredGrid (`.pvtu`) — Parallel vtkUnstructuredGrid (unstructured).

All of the VTK XML file types are valid XML documents.* The document-level element is `VTKFile`:

```
<VTKFile type="ImageData" version="0.1" byte_order="LittleEndian">
  ...
</VTKFile>
```

The attributes of the element are:

`type` — The type of the file (the bulleted items in the previous list)..

`version` — File version number in “major.minor” format.

`byte_order` — Machine byte order in which data are stored. This is either “BigEndian” or “LittleEndian”.

`compressor` — Some data in the file may be compressed. This specifies the subclass of vtkDataCompressor that was used to compress the data.

Nested inside the `VTKFile` element is an element whose name corresponds to the type of the data format (i.e., the `type` attribute). This element describes the topology the dataset, and is different for the serial and parallel formats, which are described as follows.

Serial XML File Formats. The `VTKFile` element contains one element whose name corresponds to the type of dataset the file describes. We refer to this as the dataset element, which is one of `ImageData`, `RectilinearGrid`, `StructuredGrid`, `PolyData`, or `UnstructuredGrid`. The dataset element contains one or more `Piece` elements, each describing a portion of the dataset. Together, the dataset element and `Piece` elements specify the entire dataset.

Each piece of a dataset must specify the geometry (points and cells) of that piece along with the data associated with each point or cell. Geometry is specified differently for each dataset type, but every piece of every dataset contains `PointData` and `CellData` elements specifying the data for each point and cell in the piece.

The general structure for each serial dataset format is as follows:

- **ImageData** — Each `ImageData` piece specifies its extent within the dataset’s whole extent. The points and cells

* There is one case in which the file is not a valid XML document. When the AppendedData section is not encoded as base64, raw binary data is present that may violate the XML specification. This is not default behavior, and must be explicitly enabled by the user.

are described implicitly by the extent, origin, and spacing. Note that the origin and spacing are constant across all pieces, so they are specified as attributes of the `ImageData` XML element as follows.

```
<VTKFile type="ImageData" ...>
  <ImageData WholeExtent="x1 x2 y1 y2 z1 z2"
   Origin="x0 y0 z0" Spacing="dx dy dz">
    <Piece Extent="x1 x2 y1 y2 z1 z2">
      <PointData>...</PointData>
      <CellData>...</CellData>
    </Piece>
  </ImageData>
</VTKFile>
```

- **RectilinearGrid** — Each `RectilinearGrid` piece specifies its extent within the dataset's whole extent. The points are described by the `Coordinates` element. The cells are described implicitly by the extent.

```
<VTKFile type="RectilinearGrid" ...>
  <RectilinearGrid WholeExtent="x1 x2 y1 y2 z1 z2">
    <Piece Extent="x1 x2 y1 y2 z1 z2">
      <PointData>...</PointData>
      <CellData>...</CellData>
      <Coordinates>...</Coordinates>
    </Piece>
  </RectilinearGrid>
</VTKFile>
```

- **StructuredGrid** — Each `StructuredGrid` piece specifies its extent within the dataset's whole extent. The points are described explicitly by the `Points` element. The cells are described implicitly by the extent.

```
<VTKFile type="StructuredGrid" ...>
  <StructuredGrid WholeExtent="x1 x2 y1 y2 z1 z2">
    <Piece Extent="x1 x2 y1 y2 z1 z2">
      <PointData>...</PointData>
      <CellData>...</CellData>
      <Points>...</Points>
    </Piece>
  </StructuredGrid>
</VTKFile>
```

- **PolyData** — Each `PolyData` piece specifies a set of points and cells independently from the other pieces. The points are described explicitly by the `Points` element. The cells are described explicitly by the `Verts`, `Lines`, `Strips`, and `Polys` elements.

```
<VTKFile type="PolyData" ...>
  <PolyData>
    <Piece       NumberOfPoints="#"       NumberOfVerts="#"       NumberOfLines="#"
     NumberOfStrips="#" NumberOfPolys="#">
      <PointData>...</PointData>
      <CellData>...</CellData>
      <Points>...</Points>
      <Verts>...</Verts>
      <Lines>...</Lines>
      <Strips>...</Strips>
      <Polys>...</Polys>
    </Piece>
```

```
  </PolyData>
</VTKFile>
```

- **UnstructuredGrid** — Each `UnstructuredGrid` piece specifies a set of points and cells independently from the other pieces. The points are described explicitly by the `Points` element. The cells are described explicitly by the `Cells` element.

```
<VTKFile type="UnstructuredGrid" ...>
  <UnstructuredGrid>
    <Piece NumberOfPoints="#" NumberOfCells="#">
      <PointData>...</PointData>
      <CellData>...</CellData>
      <Points>...</Points>
      <Cells>...</Cells>
    </Piece>
  </UnstructuredGrid>
</VTKFile>
```

Every dataset describes the data associated with its points and cells with `PointData` and `CellData` XML elements as follows:

```
  <PointData Scalars="Temperature" Vectors="Velocity">
    <DataArray Name="Velocity" .../>
    <DataArray Name="Temperature" .../>
    <DataArray Name="Pressure" .../>
  </PointData>
```

VTK allows an arbitrary number of data arrays to be associated with the points and cells of a dataset. Each data array is described by a `DataArray` element which, among other things, gives each array a name. The following attributes of `PointData` and `CellData` are used to specify the active arrays by name:

`Scalars` — The name of the active scalars array, if any.

`Vectors` — The name of the active vectors array, if any.

`Normals` — The name of the active normals array, if any.

`Tensors` — The name of the active tensors array, if any.

`TCoords` — The name of the active texture coordinates array, if any.

Some datasets describe their points and cells using different combinations of the following common elements:

- **Points** — The `Points` element explicitly defines coordinates for each point individually. It contains one `DataArray` element describing an array with three components per value, each specifying the coordinates of one point.

```
<Points>
  <DataArray NumberOfComponents="3" .../>
</Points>
```

`Coordinates` — The `Coordinates` element defines point coordinates for an extent by specifying the ordinate along each axis for each integer value in the extent's range. It contains three `DataArray` elements describing the ordinates along the *x*-*y*-*z* axes, respectively.

```
<Coordinates>
  <DataArray .../>
  <DataArray .../>
  <DataArray .../>
```

```
</Coordinates>
```

- **Verts**, **Lines**, **Strips**, and **Polys** — The `Verts`, `Lines`, `Strips`, and `Polys` elements define cells explicitly by specifying point connectivity. Cell types are implicitly known by the type of element in which they are specified. Each element contains two `DataArray` elements. The first array specifies the point connectivity. All the cells' point lists are concatenated together. The second array specifies the offset into the connectivity array for the end of each cell.

```
<Verts>
  <DataArray type="Int32" Name="connectivity" .../>
  <DataArray type="Int32" Name="offsets" .../>
</Verts>
```

- **Cells** — The `Cells` element defines cells explicitly by specifying point connectivity and cell types. It contains three `DataArray` elements. The first array specifies the point connectivity. All the cells' point lists are concatenated together. The second array specifies the offset into the connectivity array for the end of each cell. The third array specifies the type of each cell. (Note: the cell types are defined in **Figure 2** and **Figure 3**.)

```
<Cells>
  <DataArray type="Int32" Name="connectivity" .../>
  <DataArray type="Int32" Name="offsets" .../>
  <DataArray type="UInt8" Name="types" .../>
</Cells>
```

All of the data and geometry specifications use `DataArray` elements to describe their actual content as follows:

- **DataArray** — The `DataArray` element stores a sequence of values of one type. There may be one or more components per value.

```
<DataArray type="Float32" Name="vectors" NumberOfComponents="3"
           format="appended" offset="0"/>
<DataArray type="Float32" Name="scalars" format="binary">
           bAAAAAAAAAAAAIA/AAAAQAAAQEAAAIBA... </DataArray>
<DataArray type="Int32" Name="offsets" format="ascii">
           10 20 30 ... </DataArray>
```

The attributes of the `DataArray` elements are described as follows:

`type` — The data type of a single component of the array. This is one of `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Float32`, `Float64`. Note: the 64-bit integer types are only supported if `VTK_USE_64BIT_IDS` is on (a CMake variable—see "CMake" on page 8) or the platform is 64-bit.

`Name` — The name of the array. This is usually a brief description of the data stored in the array.

`NumberOfComponents` — The number of components per value in the array.

`format` — The means by which the data values themselves are stored in the file. This is "ascii", "binary", or "appended".

`offset` — If the format attribute is "appended", this specifies the offset from the beginning of the appended data section to the beginning of this array's data.

The `format` attribute chooses among the three ways in which data values can be stored:

`format="ascii"` — The data are listed in ASCII directly inside the `DataArray` element. Whitespace is used for separation.

`format="binary"` — The data are encoded in base64 and listed contiguously inside the `DataArray` element. Data may also be compressed before encoding in base64. The byte-order of the data matches that specified by

the `byte_order` attribute of the `VTKFile` element.

`format="appended"` — The data are stored in the appended data section. Since many `DataArray` elements may store their data in this section, the offset attribute is used to specify where each `DataArray's` data begins. This format is the default used by VTK's writers.

The appended data section is stored in an `AppendedData` element that is nested inside `VTKFile` after the dataset element:

```
<VTKFile ...>
  ...
  <AppendedData encoding="base64">
    _QMwEAAAAAAAAA...
  </AppendedData>
</VTKFile>
```

The appended data section begins with the first character after the underscore inside the `AppendedData` element. The underscore is not part of the data, but is always present. Data in this section is always in binary form, but can be compressed and/or base64 encoded. The byte-order of the data matches that specified by the `byte_order` attribute of the `VTKFile` element. Each `DataArray`'s data are stored contiguously and appended immediately after the previous `DataArray`'s data without a seperator. The `DataArray`'s `offset` attribute indicates the file position offset from the first character after the underscore to the beginning its data.

Parallel File Formats. The parallel file formats do not actually store any data in the file. Instead, the data are broken into pieces, each of which is stored in a serial file of the same dataset type.

The `VTKFile` element contains one element whose name corresponds to the type of dataset the file describes, but with a "P" prefix. We refer to this as the parallel dataset element, which is one of PImageData, PRectilinearGrid, PStructuredGrid, PPolyData, or PUnstructuredGrid.

The parallel dataset element and those nested inside specify the types of the data arrays used to store points, point data, and cell data (the type of arrays used to store cells is fixed by VTK). The element does not actually contain any data, but instead includes a list of `Piece` elements that specify the source from which to read each piece. Individual pieces are stored in the corresponding serial file format. The parallel file needs to specify the type and structural information so that readers can update pipeline information without actually reading the pieces' files.

The general structure for each parallel dataset format is as follows:

- **PImageData** — The `PImageData` element specifies the whole extent of the dataset and the number of ghost-levels by which the extents in the individual pieces overlap. The `Origin` and `Spacing` attributes implicitly specify the point locations. Each `Piece` element describes the extent of one piece and the file in which it is stored.

```
<VTKFile type="PImageData" ...>
  <PImageData WholeExtent="x1 x2 y1 y2 z1 z2"
              GhostLevel="#" Origin="x0 y0 z0" Spacing="dx dy dz">
    <PPointData>...</PPointData>
    <PCellData>...</PCellData>
    <Piece Extent="x1 x2 y1 y2 z1 z2" Source="imageData0.vti"/>
    ...
  </PImageData>
</VTKFile>
```

- **PRectilinearGrid** — The `PRectilinearGrid` element specifies the whole extent of the dataset and the number of ghost-levels by which the extents in the individual pieces overlap. The `PCoordinates` element describes the type of arrays used to specify the point ordinates along each axis, but does not actually contain the data. Each `Piece` element describes the extent of one piece and the file in which it is stored.

```
<VTKFile type="PRectilinearGrid" ...>
  <PRectilinearGrid WholeExtent="x1 x2 y1 y2 z1 z2"
                    GhostLevel="#">
```

```
    <PPointData>...</PPointData>
    <PCellData>...</PCellData>
    <PCoordinates>...</PCoordinates>
    <Piece Extent="x1 x2 y1 y2 z1 z2"
           Source="rectilinearGrid0.vtr"/>
    ...
  </PRectilinearGrid>
</VTKFile>
```

- **PStructuredGrid** — The `PStructuredGrid` element specifies the whole extent of the dataset and the number of ghost-levels by which the extents in the individual pieces overlap. The `PPoints` element describes the type of array used to specify the point locations, but does not actually contain the data. Each `Piece` element describes the extent of one piece and the file in which it is stored.

```
<VTKFile type="PStructuredGrid" ...>
  <PStructuredGrid WholeExtent="x1 x2 y1 y2 z1 z2"
                   GhostLevel="#">
    <PPointData>...</PPointData>
    <PCellData>...</PCellData>
    <PPoints>...</PPoints>
    <Piece Extent="x1 x2 y1 y2 z1 z2"
           Source="structuredGrid0.vts"/>
    ...
  </PStructuredGrid>
</VTKFile>
```

- **PPolyData** — The `PPolyData` element specifies the number of ghost-levels by which the individual pieces overlap. The `PPoints` element describes the type of array used to specify the point locations, but does not actually contain the data. Each `Piece` element specifies the file in which the piece is stored.

```
<VTKFile type="PPolyData" ...>
  <PPolyData GhostLevel="#">
    <PPointData>...</PPointData>
    <PCellData>...</PCellData>
    <PPoints>...</PPoints>
    <Piece Source="polyData0.vtp"/>
    ...
  </PPolyData>
</VTKFile>
```

- **PUnstructuredGrid** — The `PUnstructuredGrid` element specifies the number of ghost-levels by which the individual pieces overlap. The `PPoints` element describes the type of array used to specify the point locations, but does not actually contain the data. Each `Piece` element specifies the file in which the piece is stored.

```
<VTKFile type="PUnstructuredGrid" ...>
  <PUnstructuredGrid GhostLevel="0">
    <PPointData>...</PPointData>
    <PCellData>...</PCellData>
    <PPoints>...</PPoints>
    <Piece Source="unstructuredGrid0.vtu"/>
    ...
  </PUnstructuredGrid>
</VTKFile>
```

Every dataset uses PPointData and PCellData elements to describe the types of data arrays associated with its points and cells.

- **PPointData** and **PCellData** — These elements simply mirror the `PointData` and `CellData` elements from the serial file formats. They contain `PDataArray` elements describing the data arrays, but without any actual data.

```
<PPointData Scalars=”Temperature” Vectors=”Velocity”>
  <PDataArray Name=”Velocity” .../>
  <PDataArray Name=”Temperature” .../>
  <PDataArray Name=”Pressure” .../>
</PPointData>
```

For datasets that need specification of points, the following elements mirror their counterparts from the serial file format:

- **PPoints** — The `PPoints` element contains one `PDataArray` element describing an array with three components. The data array does not actually contain any data.

```
<PPoints>
  <PDataArray NumberOfComponents=”3” .../>
</PPoints>
```

- **PCoordinates** — The `PCoordinates` element contains three `PDataArray` elements describing the arrays used to specify ordinates along each axis. The data arrays do not actually contain any data.

```
<PCoordinates>
  <PDataArray .../>
  <PDataArray .../>
  <PDataArray .../>
</PCoordinates>
```

All of the data and geometry specifications use `PDataArray` elements to describe the data array types:

- **PDataArray** — The `PDataArray` element specifies the `type`, `Name`, and optionally the `NumberOfComponents` attributes from the `DataArray` element. It does not contain the actual data. This can be used by readers to create the data array in their output without needing to read any real data, which is necessary for efficient pipeline updates in some cases.

```
<PDataArray type=”Float32” Name=”vectors” NumberOfComponents=”3”/>
```

Example. The following is a complete example specifying a `vtkPolyData` representing a cube with some scalar data on its points and faces.

```
<?xml version="1.0"?>
<VTKFile type="PPolyData" version="0.1" byte_order="LittleEndian">
  <PPolyData GhostLevel="0">
    <PPointData Scalars="my_scalars">
      <PDataArray type="Float32" Name="my_scalars"/>
    </PPointData>
    <PCellData Scalars="cell_scalars" Normals="cell_normals">
      <PDataArray type="Int32" Name="cell_scalars"/>
      <PDataArray type="Float32" Name="cell_normals" NumberOfComponents="3"/>
    </PCellData>
    <PPoints>
      <PDataArray type="Float32" NumberOfComponents="3"/>
    </PPoints>
    <Piece Source="polyEx0.vtp"/>
  </PPolyData>
```

```
</VTKFile>


<?xml version="1.0"?>
<VTKFile type="PolyData" version="0.1" byte_order="LittleEndian">
  <PolyData>
    <Piece NumberOfPoints="8" NumberOfVerts="0" NumberOfLines="0"
           NumberOfStrips="0" NumberOfPolys="6">
      <Points>
        <DataArray type="Float32" NumberOfComponents="3" format="ascii">
          0 0 0 1 0 0 1 1 0 0 1 0 0 0 1 1 0 1 1 1 1 0 1 1
        </DataArray>
      </Points>
      <PointData Scalars="my_scalars">
        <DataArray type="Float32" Name="my_scalars" format="ascii">
          0 1 2 3 4 5 6 7
        </DataArray>
      </PointData>
      <CellData Scalars="cell_scalars" Normals="cell_normals">
        <DataArray type="Int32" Name="cell_scalars" format="ascii">
          0 1 2 3 4 5
        </DataArray>
        <DataArray type="Float32" Name="cell_normals"
                   NumberOfComponents="3" format="ascii">
          0 0 -1 0 0 1 0 -1 0 0 1 0 -1 0 0 1 0 0
        </DataArray>
      </CellData>
      <Polys>
        <DataArray type="Int32" Name="connectivity" format="ascii">
          0 1 2 3 4 5 6 7 0 1 5 4 2 3 7 6 0 4 7 3 1 2 6 5
        </DataArray>
        <DataArray type="Int32" Name="offsets" format="ascii">
          4 8 12 16 20 24
        </DataArray>
      </Polys>
    </Piece>
  </PolyData>
</VTKFile>
```