Coding Resources

This chapter provides information to make the job of building VTK applications and classes a little easier. Object diagrams are useful when you'd like an overview of the objects in the system, and are included here in symbolic form known as object modeling diagrams. The diagrams we use here are simplified to mainly show inheritance, but some associations between classes are shown as well. Succinct filter descriptions are provided to help you find the right filter to do the job at hand. This chapter also documents VTK file formats, which, along with the object diagrams, have changed since the release of vtk2.0 (and the release of the second edition of the *Visualization Toolkit* text.)

14.1 Object Diagrams

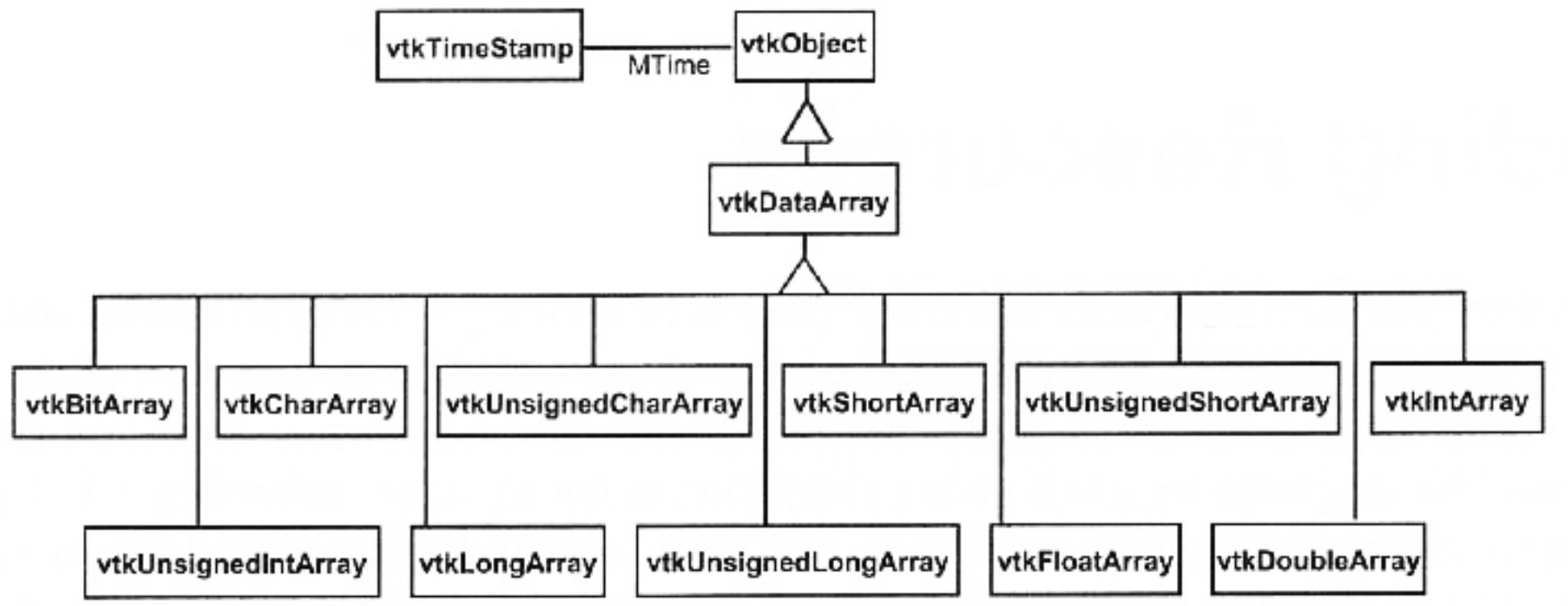
The following section contains abbreviated object diagrams using the OMT graphical language. The purpose of this section is to convey the essence of the software structure, particularly inheritance and object associations. Due to space limitation, not all objects are shown, particularly "leaf" (i.e., bottom of the inheritance tree) objects. Instead, we choose a single leaf object to represent other sibling objects. The organization of the objects follows that of the synopsis.

Foundation

The foundation object diagram is shown in **Figure 14–1**. These represent the core data objects, as well as other object manipulation classes.

Cells

The cell object diagram is shown in **Figure 14–2**. Currently, 14 concrete cell types are supported in VTK. The special class vtkGenericCell is used to represent any type of cell (i.e., supports the thread-safe vtkDataSet::GetCell() method). The class vtkEmptyCell is used to indicate the presence of a deleted or NULL cell.



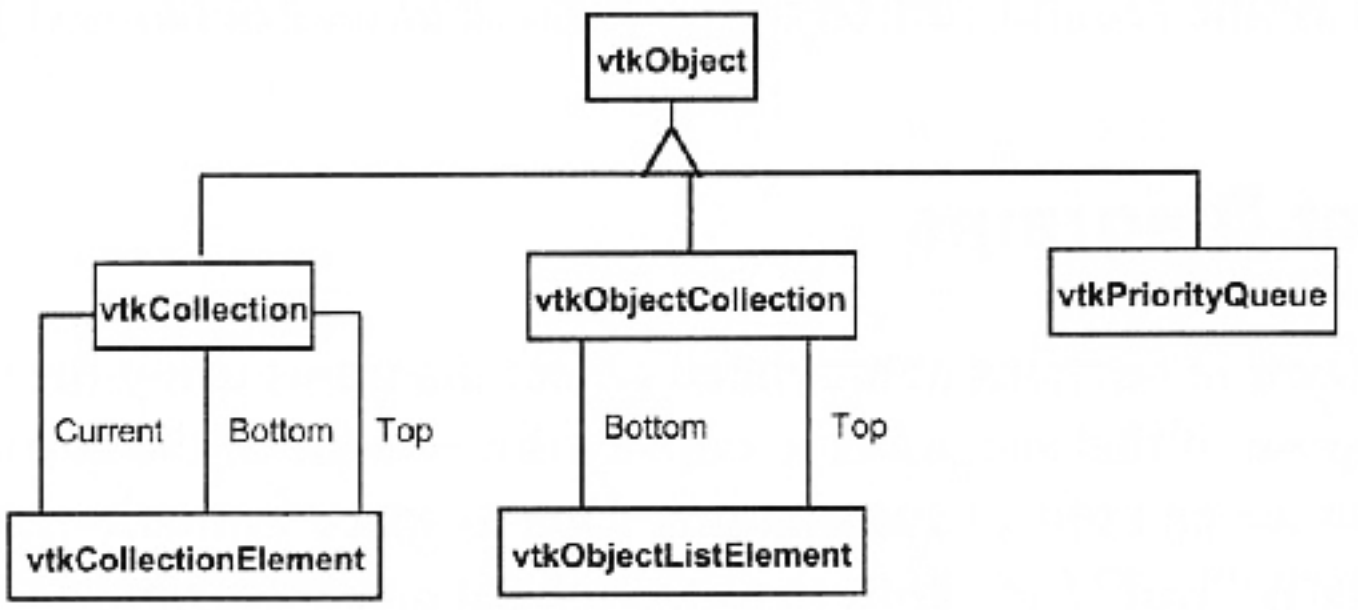


Figure 14–1 Foundation object diagram.

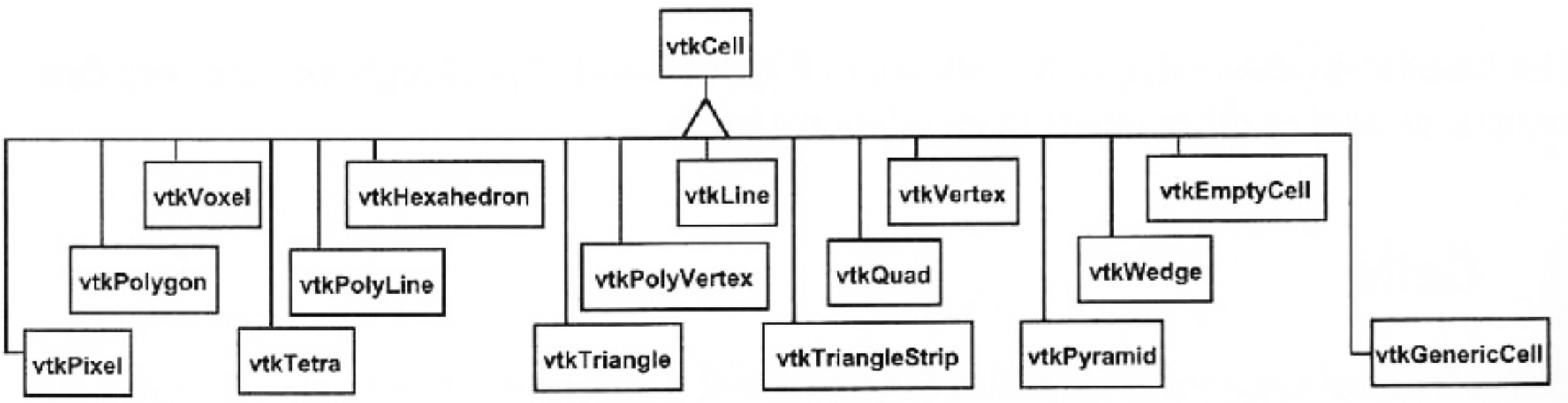


Figure 14–2 Cell object diagram.

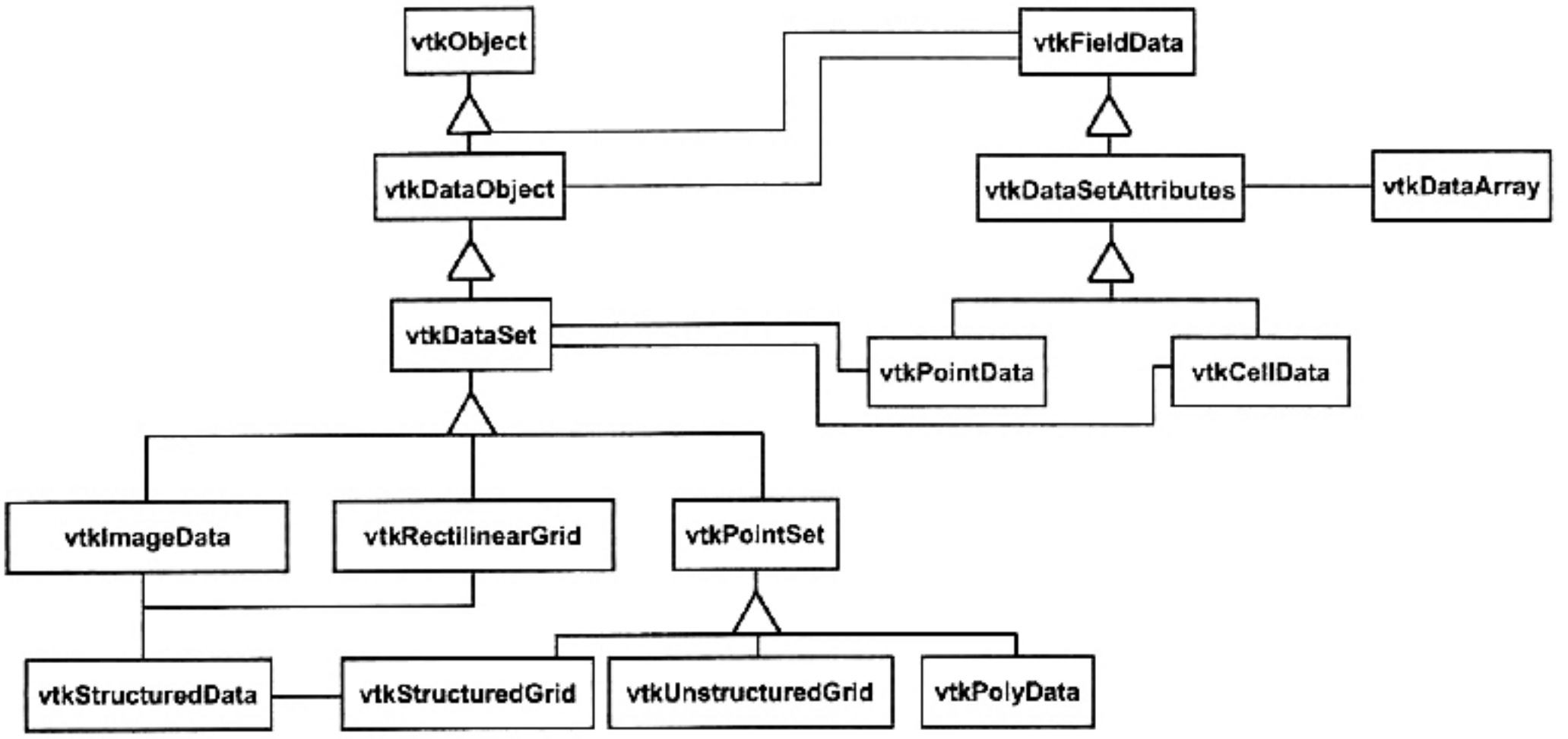


Figure 14–3 Dataset object diagram.

Datasets

The dataset object diagram is shown in **Figure 14–3**. Currently, five concrete dataset types are supported. Unstructured point data can be represented by any of the subclasses of vtkPointSet. Rectilinear grids are represented using vtkStructuredGrid. vtkImageData used to be vtkStructuredPoints, and represents 2D image and 3D volume data.

Pipeline

The pipeline object diagram is shown in **Figure 14–4**. These are the core objects to represent data.

Sources

The source object diagram is shown in **Figure 14–5**.

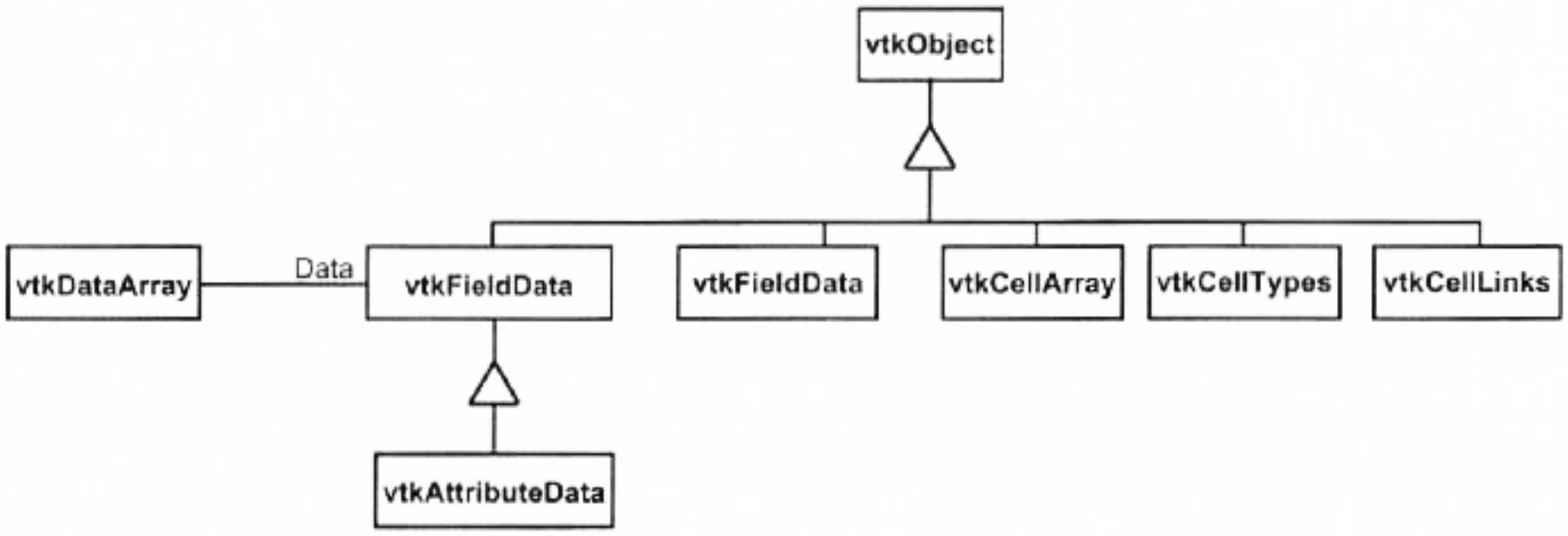


Figure 14–4 Pipeline object diagram.

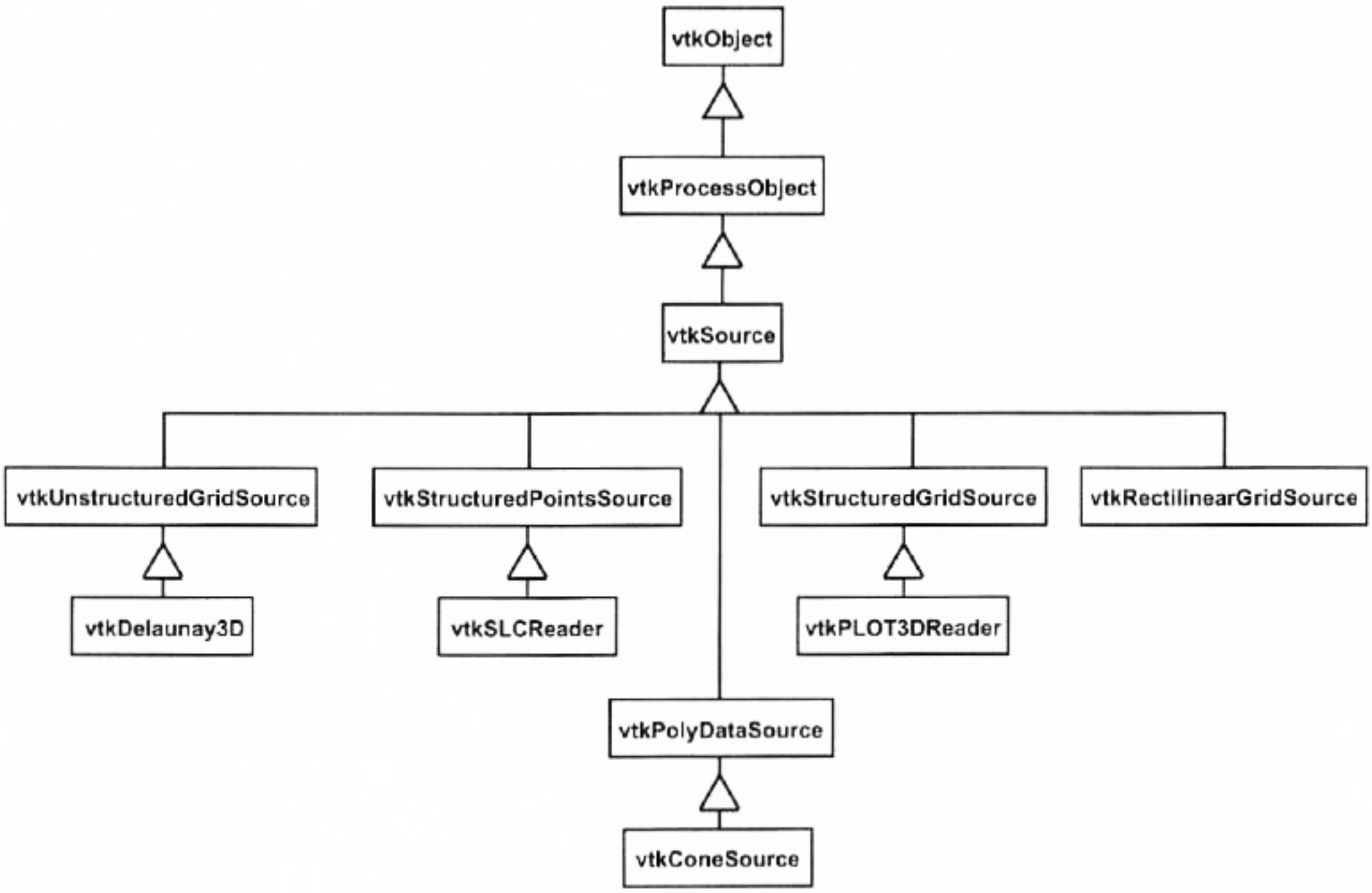


Figure 14–5 Source object diagram.

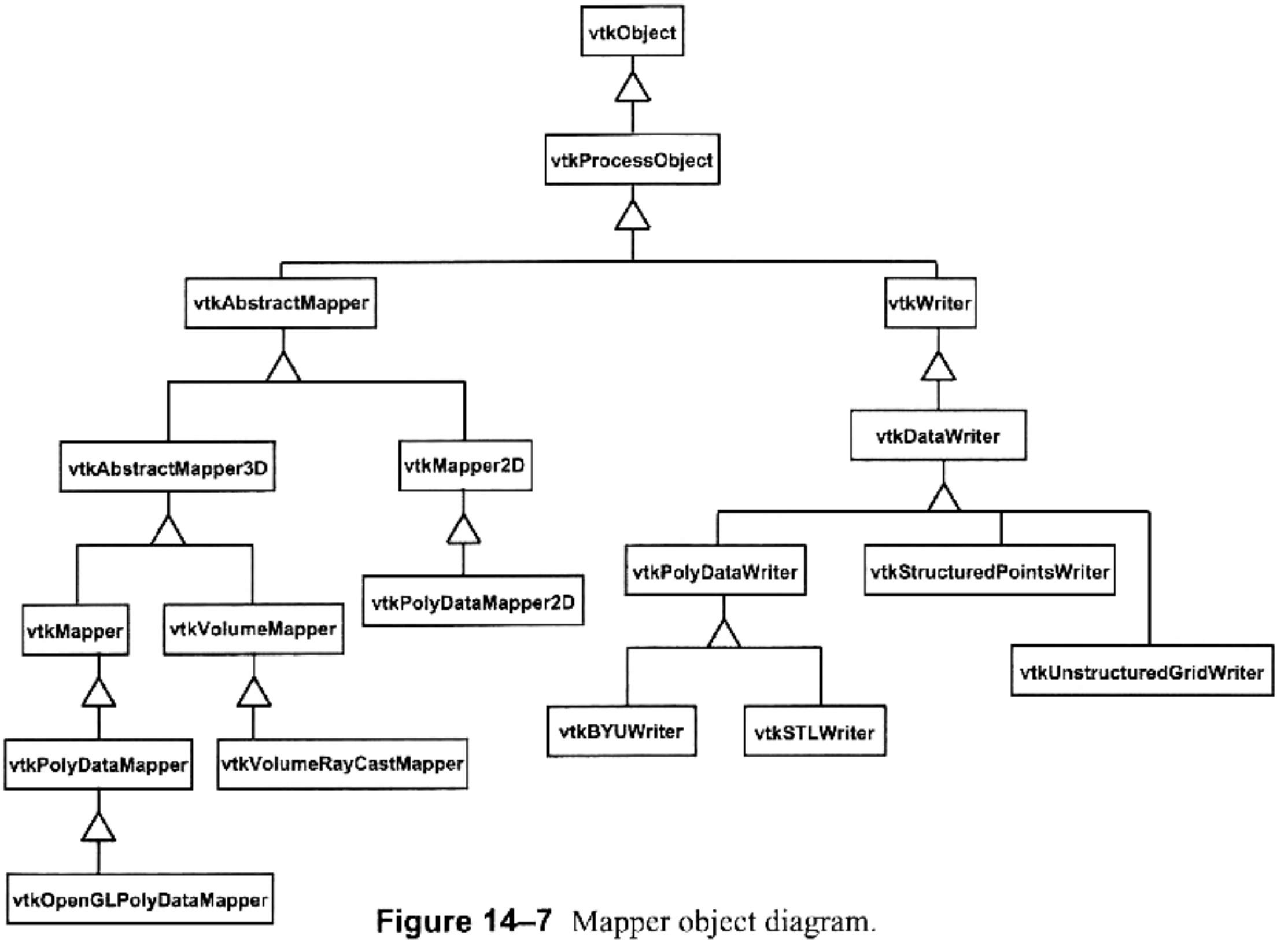


Figure 14–7 Mapper object diagram.

Filters

The filter object diagram is shown in **Figure 14–6**.

Mappers

The mapper object diagram is shown in **Figure 14–7**. There are basically two types: graphics mappers that map visualization data to the graphics system and writers that write data to output file (or other I/O device).

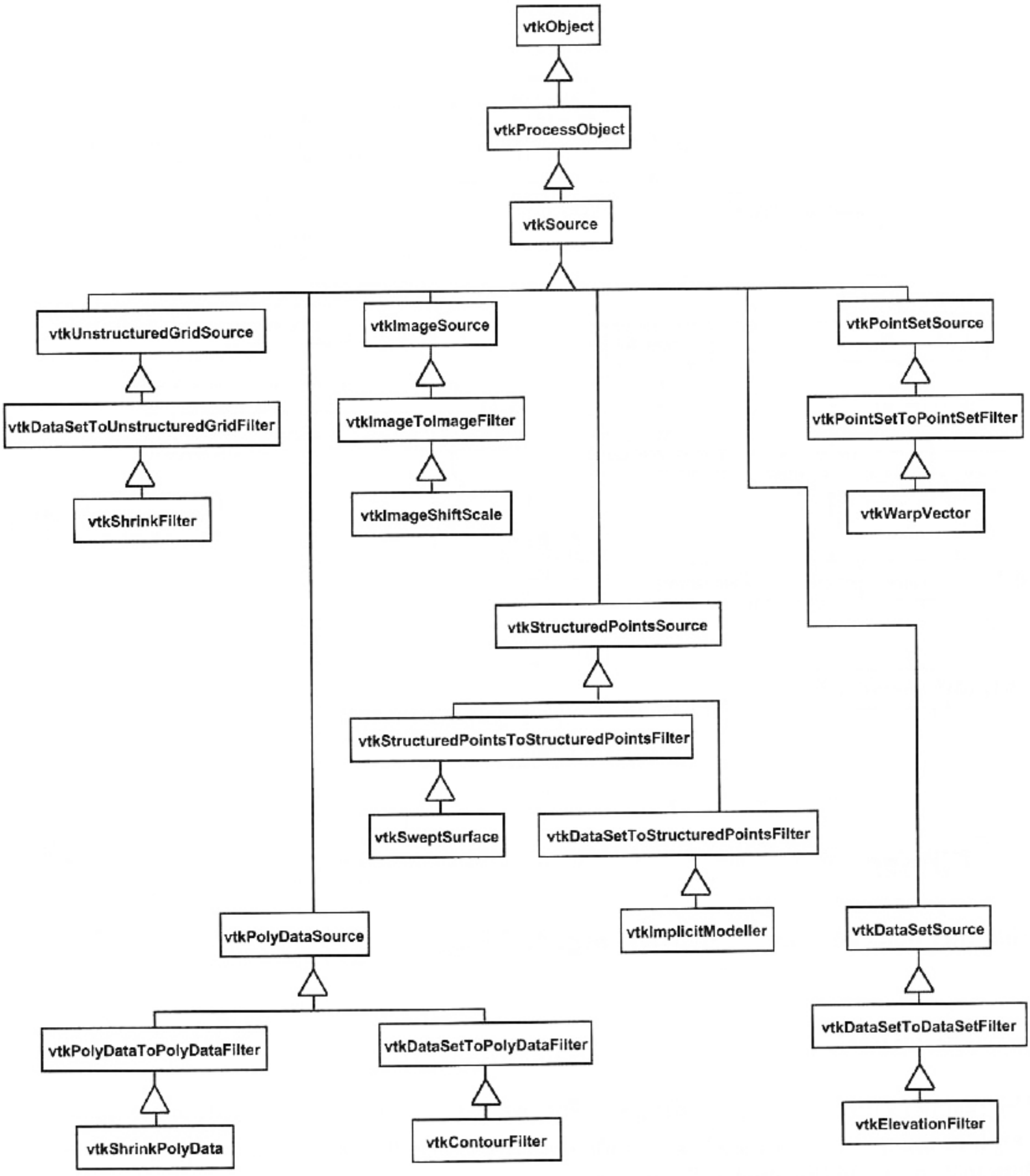


Figure 14–6 Filter object diagram.

Graphics

The graphics object diagram is shown in **Figure 14–8**. The diagram has been extended to include some associations with objects in the system. If you are unfamiliar with the object-oriented graphics notation see Rumbaugh et al., *Object-Oriented Modeling and Design.*

Volume Rendering

The volume rendering class hierarchy is shown in **Figure 14–9**. Note that volume rendering and surface rendering are integrated together via the renderer. The renderer maintains a list of both actors and volumes (in the vtkPropCollection) and handles compositing the image.

Imaging

The imaging object diagram is shown in **Figure 14–10**. Imaging integrates with the graphics pipeline via the structured points dataset. Also, it is possible to capture an image from the renderer via the vtkRendererSource object, and then feed the image into the imaging pipeline.

OpenGL Renderer

The OpenGL renderer object diagram is shown in **Figure 14–11**. Note that there are other rendering libraries in VTK. The OpenGL object diagram is representative of these other libraries.

Picking

The picking class hierarchy is shown in **Figure 14–12**. vtkPropPicker and vtkWorld-PointPicker are the fastest (use hardware) pickers. All pickers can return a global *x-y-z* from a selection point in the render window. vtkCellPicker uses software ray casting to return information about cells (cell id, parametric coordinate of intersection). vtkPoint-

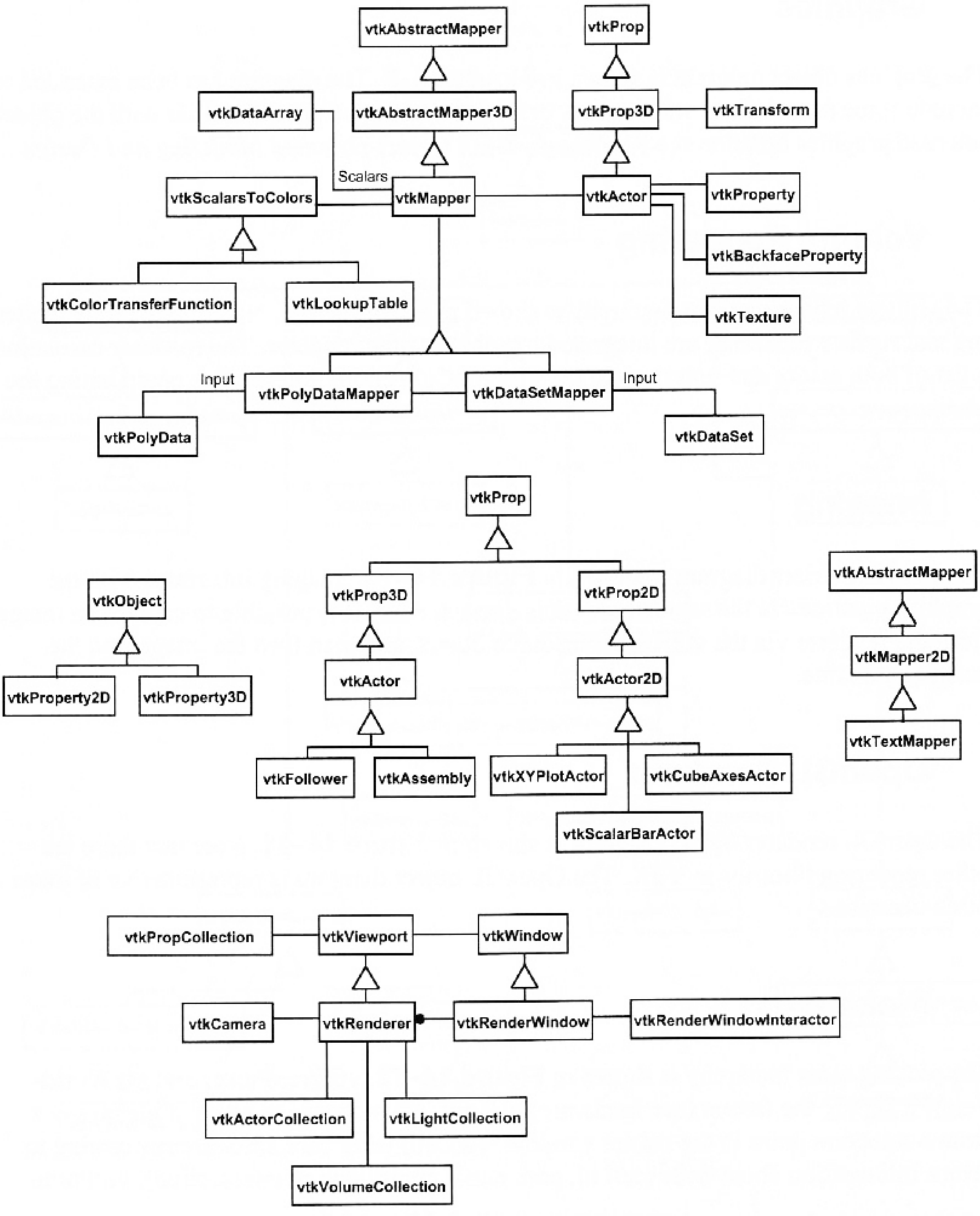


Figure 14–8 Graphics object diagram.

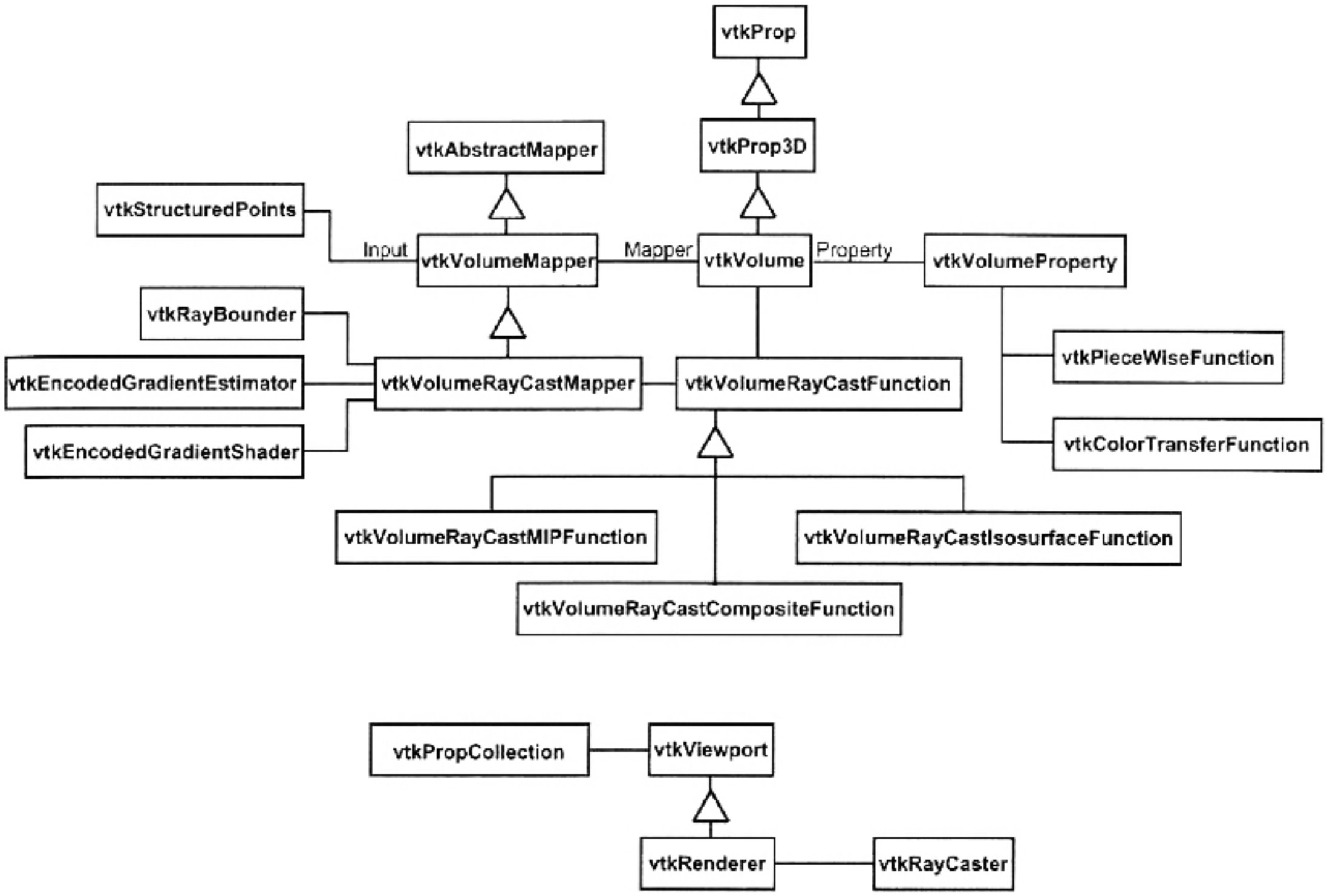


Figure 14–9 Volume rendering object diagram.

Picker returns a point id. vtkPropPicker indicates which instance of vtkProp was picked, as well as returning the pick coordinates.

Transformation Hierarchy

VTK provides an extensive, powerful transformation hierarchy. This hierarchy supports linear, non-linear, affine, and homogeneous transformations. The transformation object diagram is shown in **Figure 14–13**.

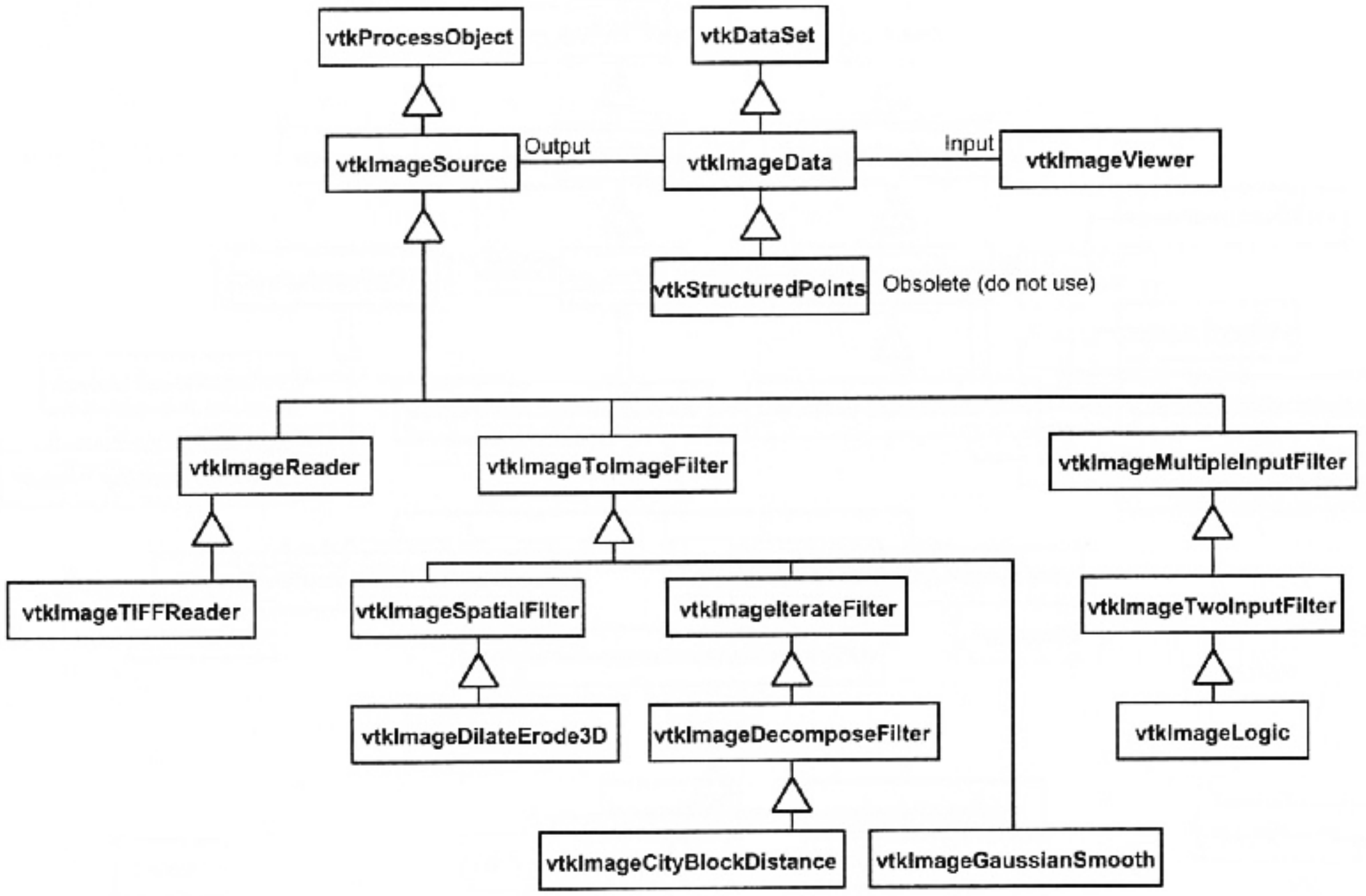


Figure 14–10 Imaging object diagram.

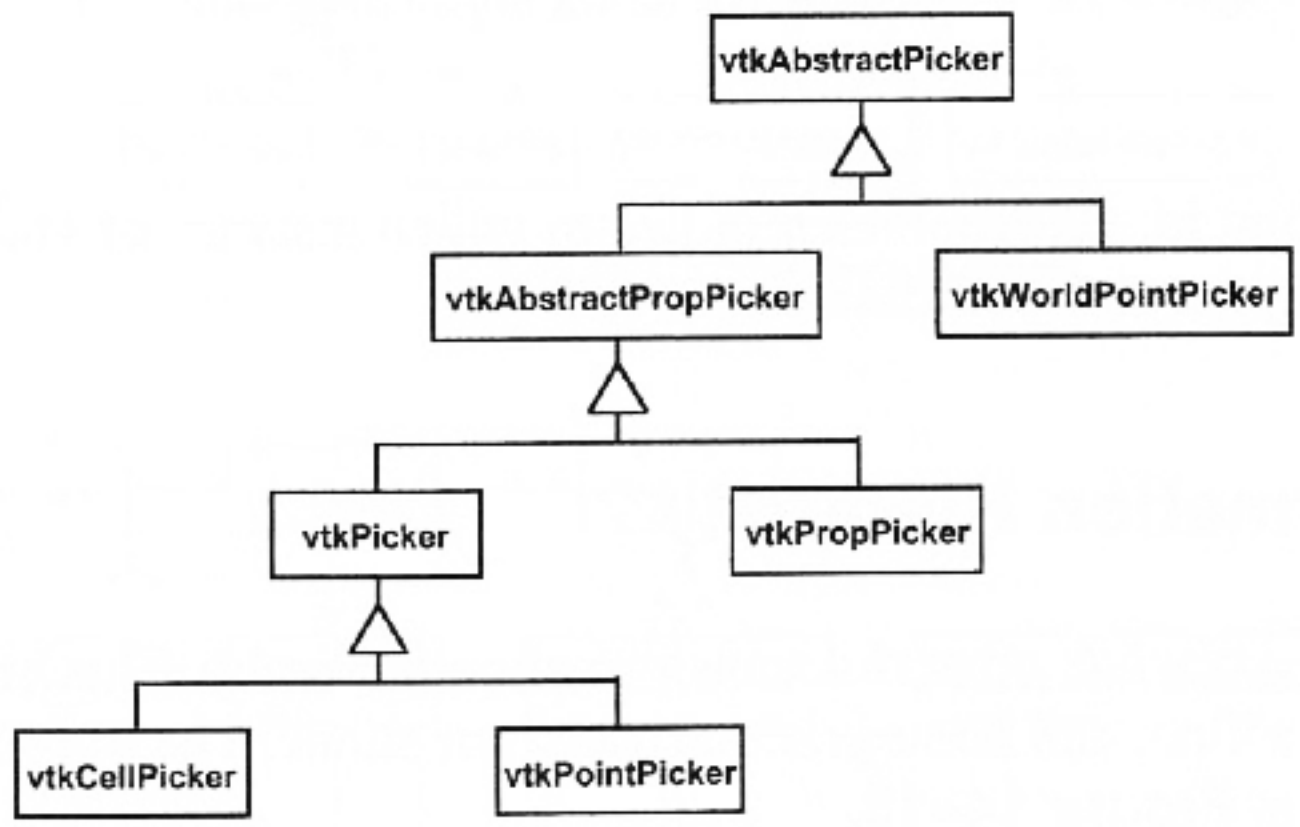


Figure 14–12 Picking class hierarchy

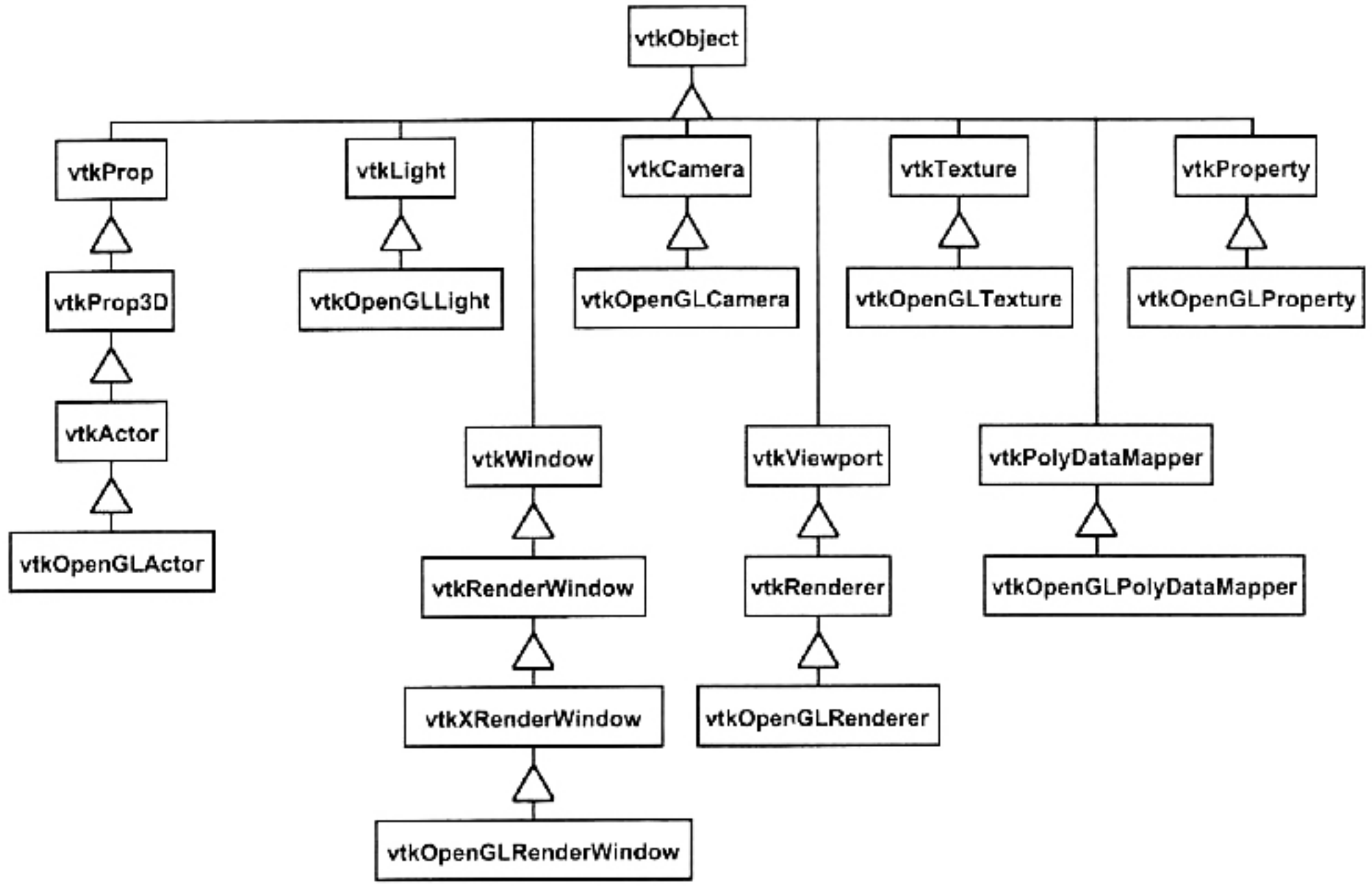


Figure 14–11 OpenGL / graphics interface object diagram.

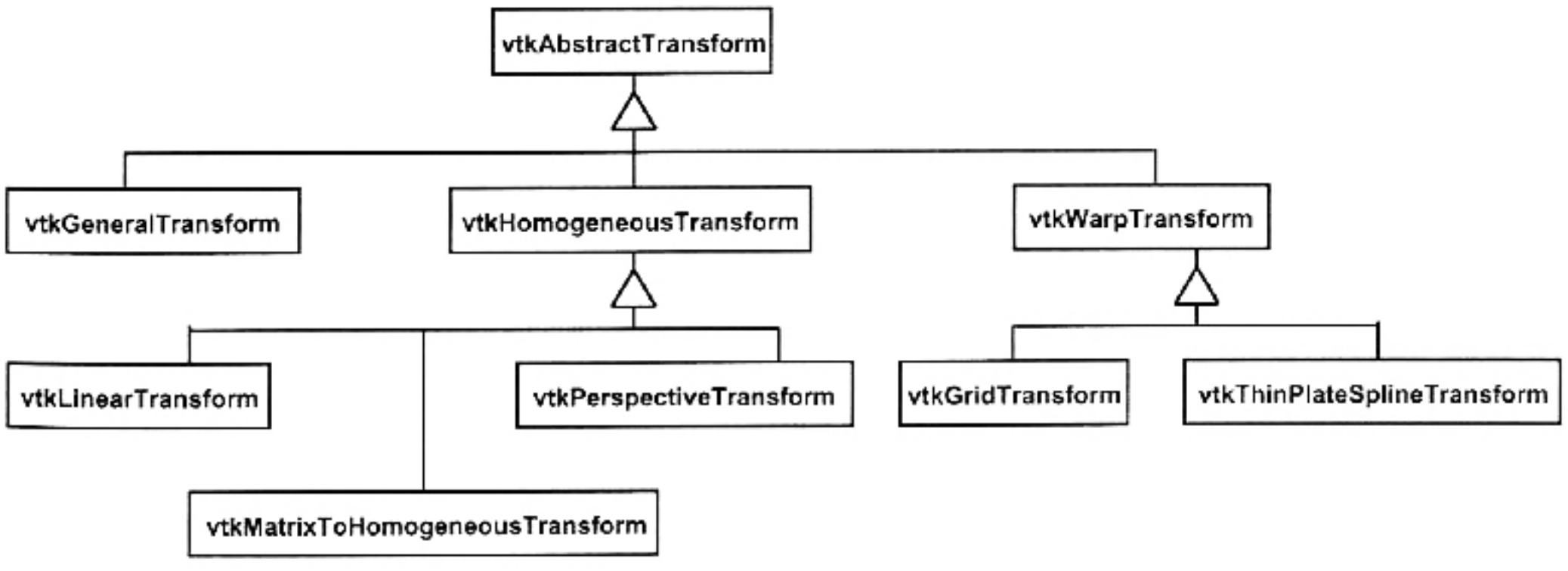


Figure 14–13 Transformation class hierarchy

14.2 Summary Of Filters

In this section we provide a brief summary of VTK filters. The section is divided into three parts: an overview of source objects, a description of visualization filters, and a list of imaging filters. Classes used to interface with data (i.e., readers, writers, importers, and exporters) are described in Chapter 8 "Data Interface & Miscellaneous" on page 185.

Source Objects

In this section we provide a brief description of source objects. Source objects initiate the visualization pipeline. Note that readers (source objects that read files) are not listed here. Instead, find them in Chapter 8 "Data Interface & Miscellaneous" on page 185. Each entry includes a brief description including the type of output they generate.

- vtkBooleanTexture — create a 2D texture map (structured points) based on combinations of inside, outside, and on implicit functions.
- vtkConeSource — generate a polygonal representation of a cone.
- vtkCubeSource — generate a polygonal representation of a cube.
- vtkCursor3D — generate a 3D cursor (represented as lines) given a bounding box and focal point.
- vtkCylinderSource — generate a polygonal representation of a cylinder.
- vtkDiskSource — generate a polygonal representation of a cone.
- vtkEarthSource — generate a polygonal representation of the earth.
- vtkImageCanvasSource2D — create an image by drawing into it with primitive shapes.
- vtkImageEllipsoidSource — create an image of a ellipsoid distribution.
- vtkImageGaussianSource — create an image of a Gaussian distribution.
- vtkImageMandelbrotSource — create an image of the Mandelbrot set.
- vtkImageNoiseSource — create an image filled with noise.
- vtkImageSinusoidSource — create an image of sinusoidal values.
- vtkLineSource — generate a polygonal representation of a cone.
- vtkOutlineSource — generate a polygonal representation of a cone.

- vtkPlaneSource — generate a polygonal representation of a cone.
- vtkPointLoad — generate a tensor field from a point load on a semi-infinite domain.
- vtkPointSetSource — create an image with sinusoidal image values.
- vtkPointSource — generate a polygonal representation of a cone.
- vtkProgrammableDataObjectSource — a filter that can be programmed at run-time to read or generate a vtkDataObject (i.e., a field).
- vtkProgrammableSource — a filter that can be programmed at run-time to read or generate any type of data.
- vtkRendererSource — an imaging filter that takes the renderer or render window into the imaging pipeline (great for screen capture).
- vtkSampleFunction — sample an implicit function unto a volume.
- vtkSphereSource — generate a polygonal representation of a cone.
- vtkSuperquadricSource — generates a polygonal representation of a superquadric.
- vtkTextSource — create text as a polygonal representation.
- vtkTexturedSphereSource — create a polygonal representation of a sphere with associated texture coordinates.
- vtkTriangularTexture — generate a triangular 2D texture map.
- vtkVectorText — create a polygonal representation of text.
- vtkVideoSource — grabs video signals as an image.

Imaging Filters

In this section we provide a brief summary of imaging filters. Note that descriptions of other visualization filters are found in "Visualization Filters" on page 337. Classes used to interface with data are described in Chapter 8 "Data Interface & Miscellaneous" on page 185.

All the filters described here take vtkImageData (or obsolete vtkStructuredPoints) as input, and produce the same type of output.

- vtkClipVolume — clip a volume with an implicit function to generate a tetrahedral mesh.

- vtkCompositeFilter — combine (composite) structured points into a single dataset.
- vtkDividingCubes — generate an isosurface as a cloud of points.
- vtkExtractVOI — extract a volume of interest and/or subsample the volume.
- vtkImageAccumulate — generate a histogram of the input image.
- vtkImageAnisotropicDiffusion2D — iteratively apply a 2D diffusion filter.
- vtkImageAnisotropicDiffusion3D — iteratively apply a 3D diffusion filter.
- vtkImageAppend — merge multiple input images into one output image.
- vtkImageAppendComponents — merge the components from two input images.
- vtkImageBlend — blend multiple images according to the alpha values and/or the opacity setting for each input.
- vtkImageButterworthHighPass — apply a frequency-domain high pass filter.
- vtkImageButterworthLowPass — apply a frequency-domain low pass filter.
- vtkImageCacheFilter — cache images for future use to avoid pipeline re-execution.
- vtkImageCanvasSource2D — basic image display / primitive drawing functionality.
- vtkImageCast — cast the input image to a specified output type.
- vtkImageCityBlockDistance — create a distance map using the city block metric.
- vtkImageClip — reduce the size of the input image.
- vtkImageComposite — composite multiple images using pixel and *Z*-buffer data.
- vtkImageConstantPad — pad the input image with a constant value.
- vtkImageContinuousDilate3D — evaluate the maximum value in an elipsoidal neighborhood.
- vtkImageContinuousErode3D — evaluate the minimum value in an elipsoidal neighborhood.
- vtkImageCorrelation — create a correlation image for two input images.
- vtkImageCursor3D — add a cursor to the input image.
- vtkImageDataStreamer — initiate streaming for image data.
- vtkImageDataToPolyDataFilter — convert an image to polygons.
- vtkImageDifference — generate a difference image / error value for two images.
- vtkImageDilateErode3D — perform dilation / erosion on a boundary.

- vtkImageDivergence — create a scalar field that represents the rate of change of the input vector field.
- vtkImageDotProduct — create a dot product image from two vector images.
- vtkImageEuclideanToPolar — convert 2D Euclidean coordinates to polar coordinates.
- vtkImageExtractComponents — extract a subset of the components of the input image.
- vtkImageFFT — perform a Fast Fourier Transform.
- vtkImageFlip — flip an image about a specified axis.
- vtkImageFourierCenter — shift the zero frequency from the origin to the center.
- vtkImageGaussianSmooth — perform 1D, 2D, or 3D Gaussian convolution.
- vtkImageGradient — compute the gradient vector of an image.
- vtkImageGradientMagnitude — compute the magnitude of the gradient vector.
- vtkImageHSVToRGB — convert HSV components to RGB.
- vtkImageHybridMedian2D — perform a median filter while preserving lines / corners.
- vtkImageIdealHighPass — perform a simple frequency domain high pass filter.
- vtkImageIdealLowPass — perform a simple frequency domain low pass filter.
- vtkImageIslandRemoval2D — remove small clusters from the image.
- vtkImageLaplacian — compute the Laplacian (divergence of the gradient).
- vtkImageLogarithmicScale — perform a log function on each pixel.
- vtkImageLogic — perform a logic operation: AND, OR, XOR, NAND, NOR, NOT.
- vtkImageLuminance — calculate luminance of an RGB image.
- vtkImageMagnify — magnify the image by an integer scale factor.
- vtkImageMagnitude — compute the magnitude of the components of an image.
- vtkImageMapToColors — map an image through a lookup table.
- vtkImageMarchingCubes — a streaming version of marching cubes.
- vtkImageMask — apply a mask to an image.
- vtkImageMaskBits — apply a bit-mask pattern to the image components.

- vtkImageMathematics — apply basic mathematical operations to one or two images.
- vtkImageMedian3D — compute a median filter in a rectangular neighborhood
- vtkImageMirrorPad — pad the input image with a mirror image.
- vtkImageNonMaximumSuppression — perform non-maximum suppression.
- vtkImageNormalize — normalize the scalar components of an image.
- vtkImageOpenClose3D — perform two dilate / erode operations.
- vtkImagePermute — permute the axes of an image.
- vtkImageQuantizeRGBToIndex — quantize an RGB image to an index image and a lookup table.
- vtkImageRange3D — compute the range (max - min) in an ellipsoidal neighborhood.
- vtkImageResample — resample an image to be larger or smaller.
- vtkImageRFFT — perform a Reverse Fast Fourier Transform.
- vtkImageRGBToHSV — convert RGB components to HSV.
- vtkImageReslice — reslice the volume along a specified axis.
- vtkImageSeedConnectivity — evaluate connectivity with user supplied seeds.
- vtkImageShiftScale — perform a shift and scale operation on the input image.
- vtkImageShrink3D — shrink an image by subsampling on a uniform grid.
- vtkImageSkeleton2D — perform a skeleton operation in 2D.
- vtkImageSobel2D — compute the vector field of an image using Sobel functions.
- vtkImageSobel3D — compute the vector field of a volume using Sobel functions.
- vtkImageThreshold — perform binary or continuous thresholding.
- vtkImageVariance3D — compute the variance within an ellipsoidal neighborhood.
- vtkImageWrapPad — pad an image using a mod operation on the pixel index.
- vtkLinkEdgels — link edgels together to form digital curves.
- vtkMarchingCubes — high-performance isocontouring algorithm.
- vtkMarchingSquares — high-performance isocontouring algorithm in 2D.
- vtkRecursiveDividingCubes — generate an isocontour as a cloud of points.

- vtkStructuredPointsGeometryFilter — extract geometry (points, lines, planes) as vtkPolyData.
- vtkSweptSurface — generate a swept surface of a moving part.
- vtkSynchronizedTemplates2D — high-performance isocontouring algorithm in 2D.
- vtkSynchronizedTemplates3D — high-performance isocontouring algorithm in 3D.

Visualization Filters

The classes listed below are organized to the type of data they input. Each class contains a brief description of what it does and any special notations regarding multiple inputs or outputs.

Input Type vtkDataSet. These filters will process any type of dataset (that is, subclasses of vtkDataSet).

- vtkAppendFilter — appends one or more datasets into a single unstructured grid.
- vtkAsynchronousBuffer — enables asynchronous pipeline execution.
- vtkAttributeDataToFieldDataFilter — transform attribute data (either point or cell) into field data.
- vtkBrownianPoints — assign random vectors to points.
- vtkCastToConcrete — cast a abstract type of input (e.g., vtkDataSet) to a concrete form (e.g., vtkPolyData).
- vtkCellCenters — generate points (vtkPolyData) marking cell centers.
- vtkCellDataToPointData — convert cell data to point data.
- vtkCellDerivatives — compute derivatives of scalar and vectors.
- vtkClipDataSet — clips arbitrary vtkDataSets with an implicit function.
- vtkConnectivityFilter — extract connected cells into an unstructured grid.
- vtkContourFilter — generate isosurface(s).
- vtkCutter — generate a *n-1* dimensional cut surface from a *n*-dimensional dataset.
- vtkDashedStreamLine — generate a streamline with dashes representing elapsed time.
- vtkDataSetToDataObjectFilter — converts a dataset into a general data object.

- vtkDicer — generate data values based on spatial (or other) segregation.
- vtkEdgePoints — generate points along edge that intersect an isosurface value.
- vtkElevationFilter — generate scalars according to projection along vector.
- vtkExtractEdges — extract the edges of a dataset as lines.
- vtkExtractGeometry — extract cells that lie either entirely inside or outside of an implicit function.
- vtkExtractTensorComponents — extract the components of a tensor as scalars, vectors, normals, or texture coordinates.
- vtkFieldDataToAttributeDataFilter — convert general field data into point or cell attribute data.
- vtkGaussianSplatter — generate a scalar field in a volume by splatting points with a Gaussian distribution.
- vtkGeometryFilter — extract surface geometry from a dataset; convert a dataset into vtkPolyData.
- vtkGlyph2D — a 2D specialization of vtkGlyph3D. Translation, rotations, and scaling is constrained to the x-y plane.
- vtkGlyph3D — copy a vtkPolyData (second input defines the glyph) to every point in the input.
- vtkGraphLayout — distribute undirected graph network into pleasing arrangement.
- vtkHedgeHog — generate scaled and oriented lines at each point from the associated vector field.
- vtkHyperStreamline — use tensor data to generate a streamtube; the tube cross section is warped according to eigenvectors.
- vtkIdFilter — generate integer id values (useful for plotting).
- vtkImplicitModeller — generate a distance field from the input geometry.
- vtkImplicitTextureCoords — create texture coordinates with an implicit function.
- vtkInterpolateDataSetAttributes — interpolate attribute data between two datasets (useful for animation).
- vtkMaskPoints — select a subset of input points.
- vtkMergeDataObjectFilter — merge a data object and dataset to form a new dataset (useful for separating geometry from solution files).

- vtkMergeFilter — merge pieces from different datasets into a single dataset.
- vtkOBBDicer — divide a dataset into pieces using oriented bounding boxes.
- vtkOutlineFilter — create an outline around the dataset.
- vtkPointDataToCellData — convert point data to cell data.
- vtkProbeFilter — probe, or resample, one dataset with another.
- vtkProgrammableAttributeDataFilter — a run-time programmable filter that operates on data attributes.
- vtkProgrammableFilter — a run-time programmable filter.
- vtkProgrammableGlyphFilter — a run-time programmable filter that can generate glyphs that vary arbitrarily based on data value.
- vtkProjectedTexture — generate texture coordinates projected onto an arbitrary surface.
- vtkSelectVisiblePoints — select the subset of points that are visible; hidden points are culled.
- vtkShepardMethod — resample a set of points into a volume.
- vtkShrinkFilter — shrink the cells of a dataset.
- vtkSimpleElevationFilter — generate scalars based on z-coordinate value.
- vtkSpatialRepresentationFilter — create a polygonal representation of spatial search (i.e., locator) objects.
- vtkStreamer — abstract superclass performs vector field particle integration.
- vtkStreamLine — generate a streamline from a vector field.
- vtkStreamPoints — generate a set of points along a streamline from a vector field.
- vtkSurfaceReconstructionFilter — constructs a surface from unorganized points.
- vtkTensorGlyph — generate glyphs based on tensor values.
- vtkTextureMapToBox — generates 3-D texture coordinates.
- vtkTextureMapToCylinder — generate 2-D texture coordinates using cylindrical coordinates.
- vtkTextureMapToPlane — generate 2-D texture coordinates by projecting data onto a plane.

- vtkTextureMapToSphere — generate 2-D texture coordinates using spherical coordinates.
- vtkThreshold — extract cells whose scalar values lie below, above, or between a threshold range.
- vtkThresholdPoints — extract points whose scalar values lie below, above, or between a threshold range.
- vtkThresholdTextureCoords — compute texture coordinates based on satisfying threshold criterion.
- vtkTransformTextureCoords — transform (e.g., scale, etc.) texture coordinates.
- vtkVectorDot — compute scalars from dot product between vectors and normals.
- vtkVectorNorm — compute scalars from Euclidean norm of vectors.
- vtkVectorTopology — mark points where the vector field vanishes (i.e., singularities exist).
- vtkVoxelModeller — convert an arbitrary dataset into a voxel representation.

Input Type vtkPointSet. These filters will process datasets that are a subclass of vtkPointSet. (These classes explicitly represent their points with an instance of vtkPoints.)

- vtkDelaunay2D — create constrained and unconstrained Delaunay triangulations including alpha shapes.
- vtkDelaunay3D — create 3D Delaunay triangulation including alpha shapes.
- vtkTransformFilter — transform the points using a 4x4 transformation matrix.
- vtkWarpLens — transform points according to lens distortion.
- vtkWarpScalar — modify point coordinates by scaling according to scalar values.
- vtkWarpTo — modify point coordinates by warping towards a point.
- vtkWarpVector — modify point coordinates by scaling in the direction of the point vectors.

Input Type vtkPolyData. The input type must be vtkPolyData. Remember that filters that accept vtkDataSet and vtkPointSet will also process vtkPolyData.

- vtkAppendPolyData — append one or more vtkPolyData datasets into a single vtkPolyData.

- vtkApproximatingSubdivisionFilter — generate a subdivision surface using an approximating scheme.
- vtkArcPlotter — plot data along an arbitrary polyline.
- vtkButterflySubdivisionFilter — subdivide a triangular, polygonal surface using a butterfly subdivision scheme.
- vtkCleanPolyData — merge coincident points, remove degenerate primitives.
- vtkClipPolyData — clip a polygonal dataset with an implicit function (or scalar value).
- vtkDecimate — reduce the number of triangles in a triangle mesh (patented).
- vtkDecimatePro — reduce the number of triangles in a triangle mesh (non-patented).
- vtkDepthSortPolyData — Sort polygons based on depth, used for translucent rendering.
- vtkExtractPolyDataGeometry — extract polygonal cells that lie entirely inside or outside of an implicit function.
- vtkFeatureEdges — extract edges that meet certain conditions (feature, boundary, non-manifold edges).
- vtkHull — generate a convex hull from six or more independent planes.
- vtkLinearExtrusionFilter — generate polygonal data by extruding another vtkPolyData.
- vtkLinearSubdivisionFilter — subdivide a triangular, polygonal surface using a linear subdivision scheme.
- vtkLoopSubdivisionFilter — subdivide a triangular, polygonal surface using a loop subdivision scheme.
- vtkMaskPolyData — select pieces of polygonal data.
- vtkPolyDataConnectivityFilter — extract connected regions.
- vtkPolyDataNormals — generate surface normals.
- vtkQuadricClustering — a decimation algorithm for very large datasets.
- vtkQuadricDecimation — a decimation algorithm using the quadric error measure.
- vtkReverseSense — reverse the connectivity order or the direction of surface normals.
- vtkRibbonFilter — create oriented ribbons from lines.

- vtkRotationalExtrusionFilter — generate polygonal data by rotationally extruding another vtkPolyData.
- vtkRuledSurfaceFilter — construct a surface from two or more "parallel" lines. Typically used to create stream surfaces from a rake of streamlines.
- vtkSelectPolyData — select polygonal data by drawing a loop.
- vtkShrinkPolyData — shrink polygonal data by shrinking each cell towards its centroid.
- vtkSmoothPolyDataFilter — use Laplacian smoothing to improve the mesh.
- vtkStripper — generate triangle strips from input triangle mesh.
- vtkSubPixelPositionEdgels — adjust edgel (line) positions based on gradients.
- vtkThinPlateSplineMeshWarp — warp (or morph) polygonal meshes using landmarks.
- vtkTransformPolyDataFilter — transform the polygonal data according to a 4x4 transformation matrix.
- vtkTriangleFilter — generate triangles from polygons or triangle strips.
- vtkTriangularTCoords — generate 2D triangular texture map.
- vtkTubeFilter — wrap lines with tubes.
- vtkVoxelContoursToSurfaceFilter — convert line contours into a surface.
- vtkWindowedSincPolyDataFilter — smooths meshes using a windowed sinc function.

Input Type vtkStructuredGrid. The input type must be vtkStructuredGrid. Remember that filters that accept vtkDataSet and vtkPointSet will also process vtkStructuredGrid.

- vtkExtractGrid — extract a region of interest / subsample a vtkStructuredGrid.
- vtkGridSynchronizedTemplates3D — high-performance isocontouring algorithm.
- vtkStructuredGridGeometryFilter — extract portions of the grid as polygonal geometry (points, lines, surfaces)
- vtkStructuredGridOutlineFilter — generate a wire outline of the boundaries of the structured grid.

Input Type vtkUnstructuredGrid. These filters take vtkUnstructuredGrid as input. Remember that filters that accept vtkDataSet will also process vtkUnstructuredGrid's.

- vtkExtractUnstructuredGrid — extract a region of interest, points, or cells from an unstructured grid.
- vtkSubdivideTetra — subdivide a tetrahedral mesh into 12 tetrahedra for every original tetrahedron.

Input Type vtkRectilinearGrid. The input type must be vtkRectilinearGrid. Remember that filters that accept vtkDataSet will also process vtkRectilinearGrid.

- vtkRectilinearGridGeometryFilter — extract portions of the grid as polygonal geometry (points, lines, surfaces)

Mapper Objects

In this section we provide a brief description of mapper objects. mapper objects terminate the visualization pipeline. Note that writers (mapper objects that write files) are not listed here. Instead, find them in Chapter 8 "Data Interface & Miscellaneous" on page 185. Each entry includes a brief description including the type of input they require.

- vtkDataSetMapper — accept any type of dataset as input and map to the graphics system.
- vtkImageMapper — 2D image display.
- vtkLabeledDataMapper — generates 3D text labels for data.
- vtkPolyDataMapper — maps polygonal data to the graphics system.
- vtkPolyDataMapper2D — draw vtkPolyData onto the overlay plane.
- vtkTextMapper — generates 2D text annotation.
- vtkVolumeProMapper — maps volumes to an image via the VolumePro hardware.
- vtkVolumeRayCastMapper — maps volumes to an image via ray casting.
- vtkVolumeTextureMapper2D — maps volumes to an image via 2D textures.

Actor (Prop) Objects

The following is a brief description of the various types of vtkProp (e.g., vtkProp3D and vtkActor) available in the system.

- vtkActor2D — type of prop drawn in the overlay plane.
- vtkAssembly — an ordered grouping of vtkProp3D's with shared transformation matrix.
- vtkAxisActor2D — a single labeled axis drawn in the overlay plane.
- vtkCaptionActor2D — attach a text caption to an object.
- vtkCubeAxesActor2D — draw the x-y-z axes of a vtkProp.
- vtkFollower — a vtkProp3D that allows faces the camera.
- vtkImageActor — a special type of vtkProp that draws an image as a texture map on top of a single polygon.
- vtk Legend Box Actor — used by vtkXYPlotActor to draw curve legends; combines text, symbols, and lines into a curve legend.
- vtkLODActor — a simple level-of-detail scheme for rendering 3D geometry.
- vtkLODProp3D — level-of-detail method for vtkProp3D's.
- vtk Parallel Coordinates Actor — multivariate visualization technique.
- vtkPropAssembly — a group of vtkProps
- vtkProp3D — a transformable (i.e., has a matrix) type of vtkProp
- vtkScalarBar Actor — a labeled, colored bar that visually expresses the relationship between color and scalar value.
- vtkScaledTextActor — text that scales as the viewport changes size.
- vtkVolume — a vtkProp used for volume rendering.
- vtkXYPlotActor — draw an *x*-*y* plot of data.

14.3 VTK File Formats

The *Visualization Toolkit* provides a number of source and writer objects to read and write various data file formats. The *Visualization Toolkit* also provides some of its own file for-

mats. 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, however, that these formats are not supported by many other tools.

The visualization 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 14–14**. 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.

```
# vtk DataFile Version 2.0                    ] (1)
Really cool data                  ] (2)
ASCII | BINARY                    ] (3)
DATASET type          ]
...                   ] (4)
POINT_DATA n          ]
...                   ] (5)
CELL_DATA n           ]
...                   ]
```

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 14–14 Overview of five parts of VTK data file format.

- *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.
- 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. 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 14–15**).

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 14–14**), 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 194 and the fourth example in this chapter "Examples" on page 354.

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*
$t_{00}\ t_{01} \cdots t_{0(dim-1)}$
$t_{10}\ t_{11} \cdots t_{1(dim-1)}$
...
$t_{(n-1)0}\ t_{(n-1)1} \cdots t_{(n-1)(dim-1)}$

- 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
```

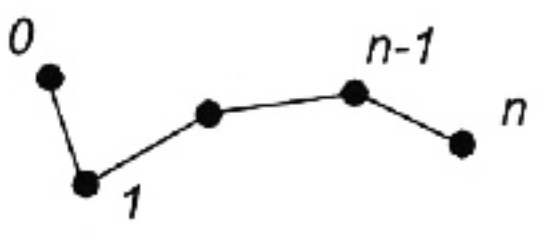
The third example is an unstructured grid containing 12 of the 14 VTK cell types. The file contains scalar and vector data.

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

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
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
```

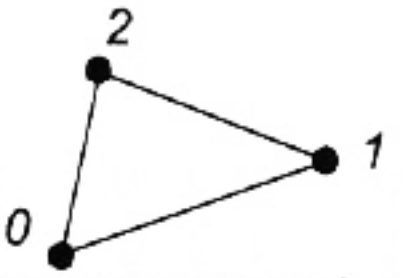
VTK_VERTEX (=1)

VTK_POLY_VERTEX (=2)

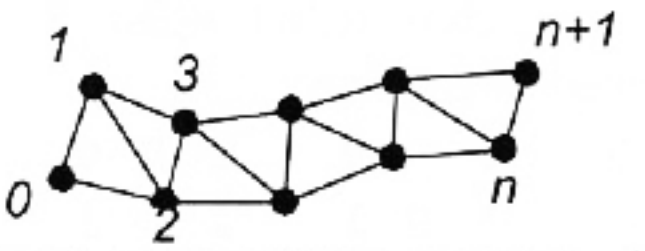
VTK_LINE (=3)



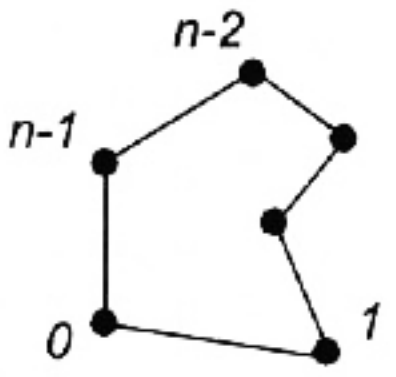
VTK_POLY_LINE (=4)



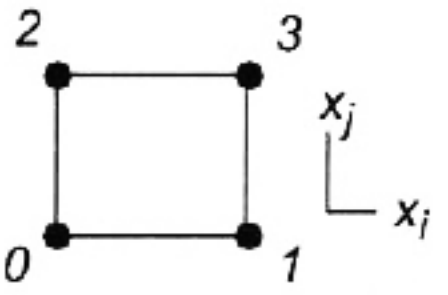
VTK_TRIANGLE(=5)



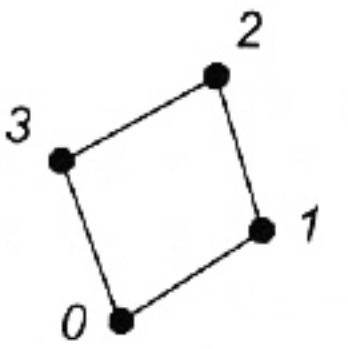
VTK_TRIANGLE_STRIP (=6)



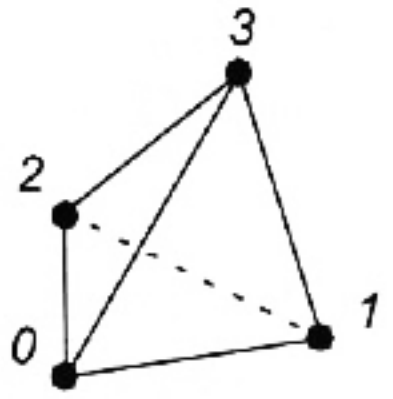
VTK_POLYGON (=7)



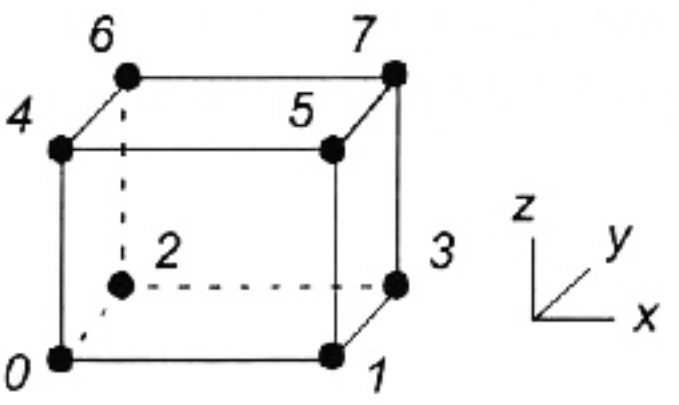
VTK_PIXEL (=8)



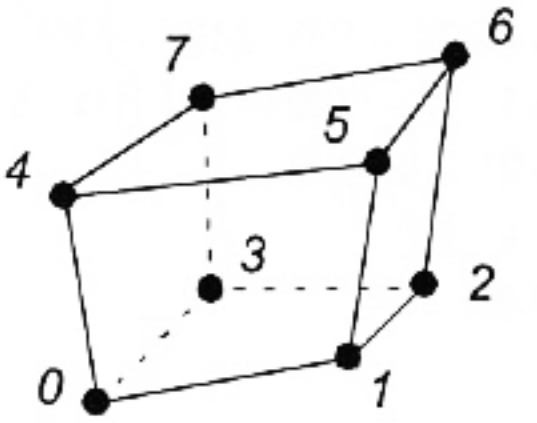
VTK_QUAD (=9)



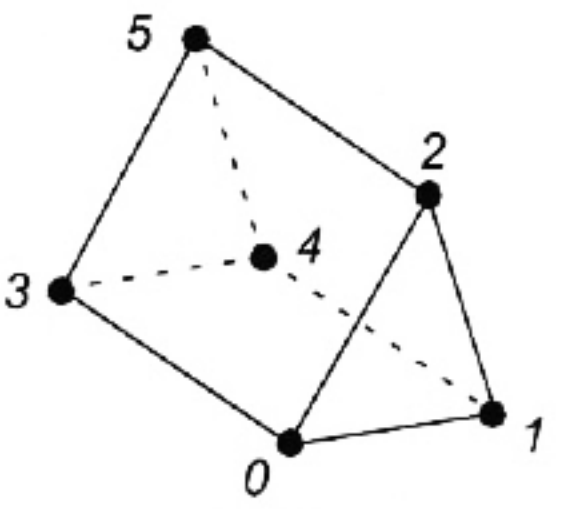
VTK_TETRA (=10)



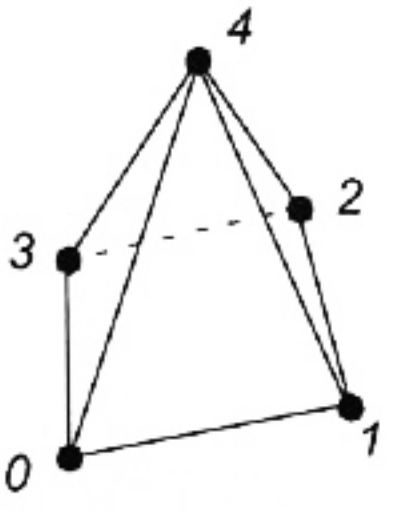
VTK_VOXEL (=11)



VTK_HEXAHEDRON (=12)



VTK_WEDGE (=13)



VTK_PYRAMID (=14)

Figure 14–15 Cell type specification. Use the include file CellType.h to manipulate cell types.

```
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 194 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.