Interfacing To VTK Data Objects

In this section we provide detailed information describing the interface to the many data objects in VTK. This ranges from datasets, which are processed by the filter objects in visualization pipelines, to data arrays, which are used represent a portion of a dataset (e.g., the scalar data).

To help you understand the relationship of data objects in VTK, refer to the object diagrams in **Figure 14–1** through **Figure 14–4**.

11.1 Data Arrays

Data arrays, implemented in the superclass vtkDataArray and its many subclasses, are the foundation upon which many of the VTK data objects are built. Data arrays represent contiguous arrays of data in native type (e.g., `char`, `int`, `float`, etc.) and have the ability to manage internal memory by dynamic allocation. Data arrays are commonly used to represent geometry (vtkPoints manages an instance of vtkDataArray under the hood), attribute data (scalars, vectors, normals, tensors, texture coordinates) and field data. Data arrays also provide an interface to their internal data, which is based on a tuple abstraction (refer to **Figure 11–1**).

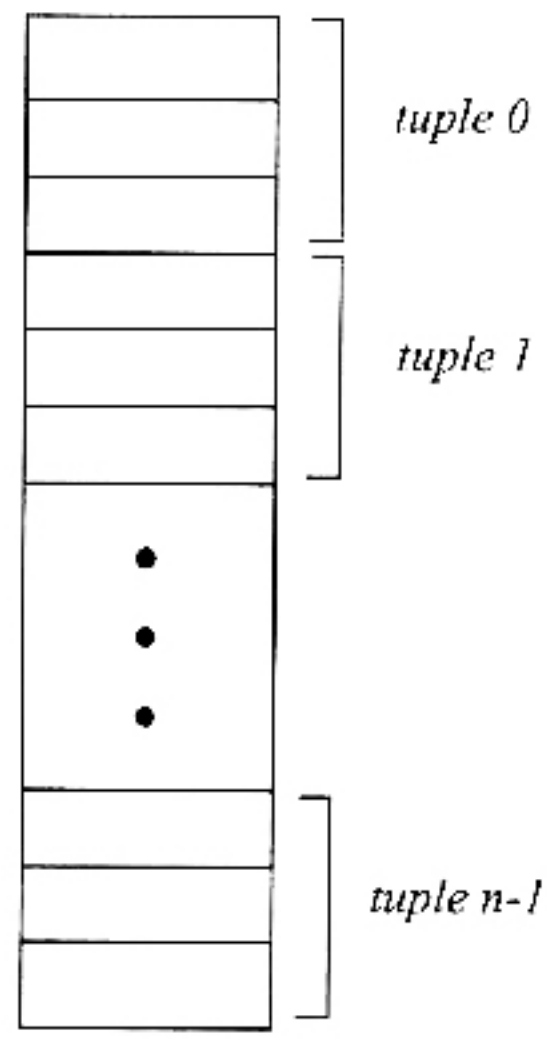


Figure 11–1 Data array structure. In this example, each tuple has three components.

In the tuple abstraction, vtkDataArray represents data as an array of tuples, each tuple consisting of the same number of components and each of the same native data type. In implementation, the tuples are actually subarrays within a contiguous array of data, as shown in the figure.

The power of data arrays and the tuple abstraction is that data can be represented in native type, and visualization data can be represented as a tuple. For example, we can represent vector data of native type `float` by creating a vtkFloatArray (a subclass of vtkDataArray) of tuple size (i.e., number of components) equal to 3. In this case, the number of vectors is simply the number of tuples; or alternatively, the number of float values divided by the number of components (for a vector, the number of components is always three).

Methods

The following is a summary of the methods of vtkDataArray. Note that these are the methods required by all subclasses of vtkDataArray; there are several other methods specialized to each subclass. The special methods (which deal mainly with pointers and data specific information) for one subclass, vtkFloatArray, are shown immediately following the vtkDataArray method summary.

`dataArray = MakeObject()`

Create an instance (`dataArray`) of the same type as the current data array. Also referred to as a "virtual" constructor.

`type = GetDataType()`

Return the native type of data as an integer token (tokens are defined in `vtkSetGet.h`). The possible types are `VTK_VOID`, `VTK_BIT`, `VTK_CHAR`, `VTK_UNSIGNED_CHAR`, `VTK_SHORT`, `VTK_UNSIGNED_SHORT`, `VTK_INT`, `VTK_UNSIGNED_INT`, `VTK_LONG`, `VTK_UNSIGNED_LONG`, `VTK_FLOAT`, and `VTK_DOUBLE`.

`SetNumberOfComponents(numComp)`

Specify the number of components per tuple.

`GetNumberOfComponents()`

Get the number of components per tuple.

`SetNumberOfTuples(number)`

Set the number of tuples in the data array. This method allocates storage, and depends on prior invocation of the method SetNumberOfComponents() for proper allocation.

`numTuples = GetNumberOfTuples()`

Return the number of tuples in the data array.

`tuple = GetTuple(i)`
: Return a pointer to an array that represents a particular tuple in the data array. This method is not thread-safe, and data may be cast to a common type (i.e., `float`).

`GetTuple(i, tuple)`
: Fill in a user-provided tuple (previously allocated) with data.

`SetTuple(i, tuple)`
: Specify the tuple at array location `i`. This method does not do range checking, and is faster than methods that do (e.g., vtkInsertTuple()). You must invoke SetNumberOf-Tuples() prior to inserting data with this method.

`InsertTuple(i, tuple)`
: Insert data at the tuple location `i`. This method performs range checking, and will allocate memory if necessary.

`i = InsertNextTuple(tuple)`
: Insert data at the end of the data array, and return its position in the array. This method performs range checking, and will allocate memory if necessary.

`c = GetComponent(i, j)`
: Get the component value as a `float` at tuple location `i` and component `j` (of the i^{th} tuple).

`SetComponent(i, j, c)`
: Set the j^{th}component value at tuple location `i`. This method does not perform range checking, you must have previously allocated memory with the method SetNumber-OfTuples().

`InsertComponent(i, j, c)`
: Insert the j^{th}component value at tuple location `i`. This method performs range checking and will allocate memory as necessary.

`GetData(tupleMin, tupleMax, compMin, compMax, data)`
: Extract a rectangular array of data (into `data`). The array `data` must have been pre-allocated. The rectangle is defined from the minimum and maximum ranges of the components and tuples.

`DeepCopy(dataArray)`
: Perform a deep copy of another object. Deep copy means that the data is actually copied, not reference counted.

`ptr = GetVoidPointer(id)`
Return a pointer to the data as a `void *` pointer. This can be used, in conjunction with the method GetDataType(), to cast data to and from the appropriate type.

`Squeeze()`
Reclaim any unused memory the data array may have allocated. This method is typically used when you use Insert() methods and cannot exactly specify the amount of data at initial allocation.

`Reset()`
Modify the data array so it looks empty but retains allocated storage. Useful to avoid excessive allocation and deallocation.

`array = MakeObject()`
Create a data array of the same type as this one.

`range = GetRange(i)`
Return the range (min,max) of ith component. This method is not thread safe.

`GetRange(range, i)`
Fill in the minimum/maxmimum values of ith component in a user-provided array.

`CreateDefaultLookupTable()`
If no lookup table is specified, and a lookup table is needed, then a default table is created.

`SetLookupTable(lut)`
Specifiy a lookup table to use for mapping array values to colors.

`lut = GetLookupTable()`
Return the lookup table to use for mapping array values to colors.

`GetTuples(ids, array)`
Given a list of ids, fill user-provided array with tuples corresponding to those ids. For example, the ids might be the ids defining a cell and the return list is the cell scalars.

The following methods are from vtkFloatArray, which is a subclass of vtkDataArray. Note that there is overlap between the functionality available from the superclass vtkDataArray and its concrete subclasses. The is because the superclass provides some generic functionality useful for quick/compact coding, while the subclasses let you get at the data directly, which can be manipulated via pointer manipulation and/or templated functions.

`v = GetValue(i)`
Return the value at the i^{th} data location in the array.

`SetNumberOfValues(number)`
Set the number of values in the array. This method performs memory allocation.

`SetValue(i, value)`
Set the value at the i^{th} data location in the array. This method requires prior invocation of SetNumberOfValues() or WritePointer(). The method is faster than the insertion methods because no range checking is performed.

`InsertValue(i, f)`
Insert the value at the i^{th} data location in the array. This method perform range checking and allocates memory as necessary.

`id = InsertNextValue(f)`
Insert the value `f` at the end of the data array, and return its position in the array. This method performs range checking, and will allocate memory if necessary.

`ptr = GetPointer(i)`
Return the pointer to the data array. The pointer is returned from the i^{th} data location (usually `i=0` and the method returns the pointer at the beginning of the array).

`ptr = WritePointer(i, number)`
Allocate memory and prepare the array for `number` direct writes starting at data location `i`. A pointer to the data starting at location `i` is returned.

`ptr = GetVoidPointer(id)`
Cast the data pointer to `void *`.

`SetArray(array, size, save)`
Directly set the data array from an outside source. This method is useful when you are interfacing to data and want to pass the data into VTK's pipeline. The `save` flag indicates whether the `array` passed in should be deleted when the data array is destructed. A value of one indicates that VTK should not delete the data array when it is done using it.

11.2 Datasets

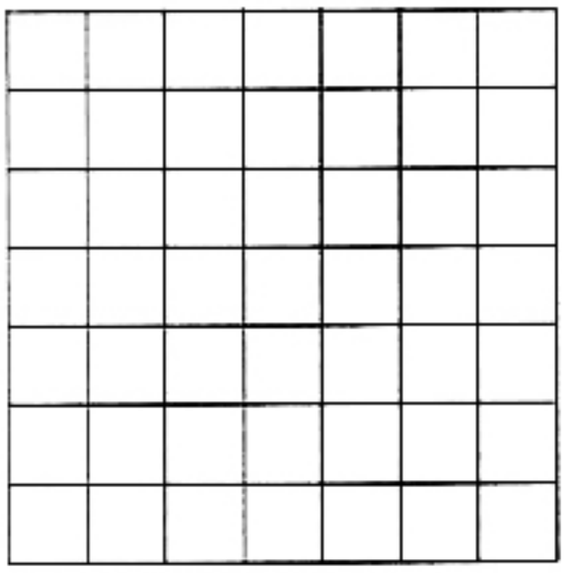
Often times the hardest part about writing a filter is interfacing to the VTK data objects. Chances are that as a filter developer, you are intimately familiar with the algorithm. It's learning how to manipulate VTK datasets—reading the input and creating the output—that is the key to writing an efficient, robust, and useful filter. Probably the single most important step in learning how to manipulate datasets is understanding the data model.

Figure 11–2 summarizes the dataset types in VTK. If you're not familiar with these objects, you'll want to read *The Visualization Toolkit* text, and/or spend some quality time with the code to understand the data model. The following paragraphs highlight some of the important features of the data model.

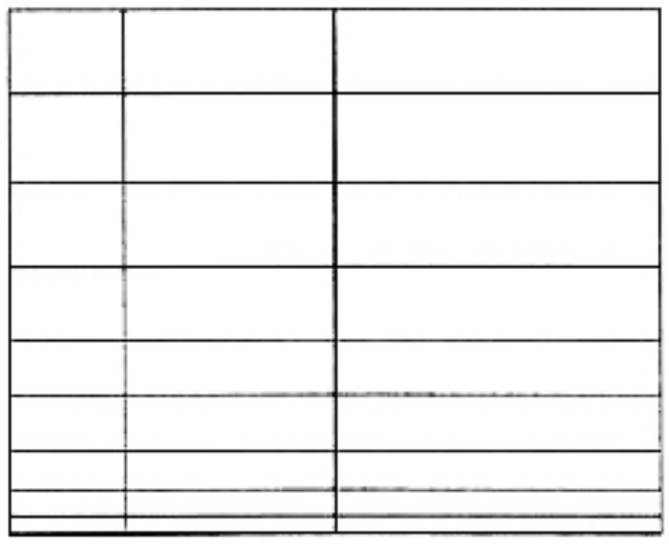
One important aspect of VTK's data model is the relationship of the *structure* and *data attributes* of a dataset. Datasets are a subclass of vtkDataObject. The difference between the two classes is that datasets have topological and geometric structure while data objects represent a general field of data (see the "The Visualization Model" on page 23). The dataset structure consists of the geometric and topological relationship of points and cells to one another. The structure is the framework to which the dataset attributes are attached. The dataset attributes are information (such as scalars, vectors, tensors, normals, texture coordinates, and field data) that is associated with either the geometry (e.g., points) or topology (e.g., cells) of the dataset. When we refer to dataset type, we are actually referring to the structure of the dataset. The data attributes are assumed to be associated with the dataset, and are created and manipulated the same way no matter the type of dataset with which they are associated.

Another important feature of VTK's data model is the relationship of datasets to cells, and whether cells are represented implicitly or explicitly. A cell can be thought of as the atoms that form a dataset. Cells are a topological organization of the dataset points (*x*-*y*-*z* positions) into an ordered list, or connectivity array. For example, in a polygonal dataset (vtkPolyData), the polygons are the cells of the dataset, and each polygonal cell is represented as an ordered list of points (i.e., the vertices of the polygon). Some datasets represent points and cells explicitly (e.g., a list of points and cells in vtkPolyData), while others represent the points and cells implicitly (e.g., an image vtkImageData represented by its dimensions, spacing, and origin). Implicit representation means that we do not explicitly store point coordinates or cell connectivity. Instead, this information is derived as necessary. You may wish to refer to **Figure 14–15** to view the cell types found in VTK.

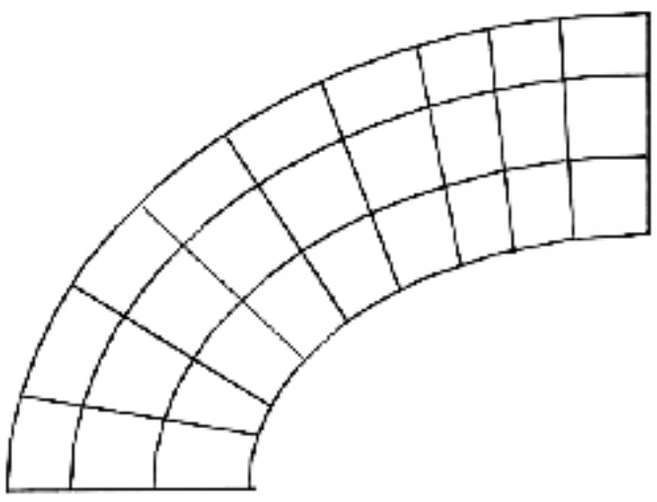
Assuming that you have a thorough understanding of VTK's data model, you'll need to know how to get data from the datasets, and how to create and put data into the datasets. These activities are dependent on the type of dataset—different datasets have different preferred ways to interface to them. Also, datasets can be accessed at different levels of abstraction corresponding to the inheritance hierarchy. For example, to get the coordinates of a point in a vtkPolyData, we can invoke the superclass method vtkDataSet::GetPoint(), or we can retrieve the points array in vtkPolyData with pts=vtkPolyData::GetPoints(), followed by access of the point coordinates pts->GetPoint(). Both approaches are valid, and



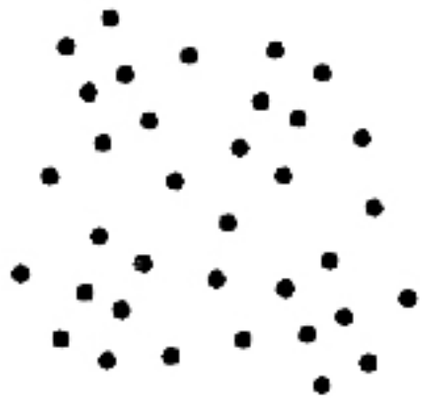
(a) Image Data
(vtkImageData)



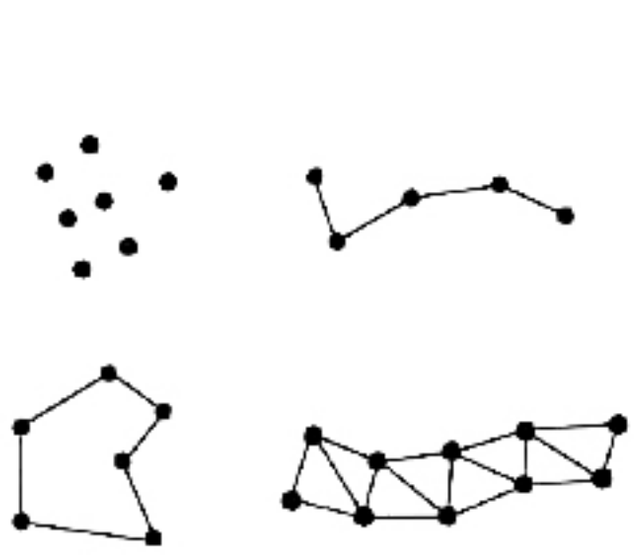
(b) Rectilinear Grid
(vtkRectilinearGrid)



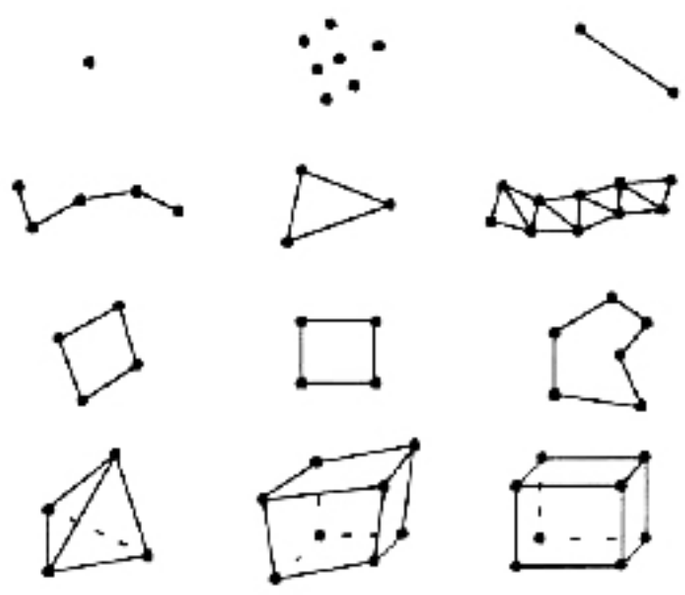
(c) Structured Grid
(vtkStructuredGrid)



(d) Unstructured Points
(use vtkPolyData)



(e) Polygonal Data
(vtkPolyData)



(f) Unstructured Grid
(vtkUnstructuredGrid)

Figure 11–2 Dataset types found in **VTK**. Note that unstructured points can be represented by either polygonal data or unstructured grids, so are not explicitly represented in the system.

both are used, depending on the circumstances. As a filter writer, you'll have to determine the correct level of abstraction by which to interface with the datasets.

Note: for the fasted access, you could get the pointer to the points array and then use templated methods to process the data. For example:

```
void *ptr = pts->GetData()->GetVoidPointer(0);
switch (pts->GetData()->GetDataType())
  case VTK_FLOAT:
    float *fptr = static_cast<float*>ptr;
    ...etc...
```

The next sections summarize how to manipulate the various types of datasets. You'll need this information if you're going to write a filter. Once you have understood the interface to the datasets, the actual construction of a graphics filter is relatively straightforward. As you read this material, you may wish to refer to the inheritance hierarchy (**Figure 14–3**). The summary information for each dataset is broken into three parts:

1. a general description of the dataset,
2. methods to create, manipulate, and extract information from the dataset,
3. and one or more examples demonstrating important concepts.

11.3 Interfacing To vtkDataSet

vtkDataSet is an abstract dataset type. Abstract objects provide an API to manipulate objects. Abstract objects cannot be (directly) instantiated; rather, they are used to manipulate concrete subclasses of the abstract object. In practice, abstract objects like vtkDataSet are used to create general filters that can manipulate any type of dataset.

Methods

vtkDataSet provides access methods that all other datasets inherit. This means that the methods described here can also be used by all other datasets since they are subclasses of vtkDataSet. Note that vtkDataSet is an abstract class typically used to refer to an underlying concrete class. Therefore, you must use the MakeObject() method to create an instance of the concrete class. This is almost always followed by the CopyStructure() method, which makes a copy of the geometric and topological structure of the dataset. (You may

also wish to copy the dataset attributes, see "Interface To Field and Attribute Data" on page 270 for more information.)

`dataSet = MakeObject()`
: Create an instance of the same type as the current dataset. Also referred to as a "virtual" constructor.

`CopyStructure(dataSet)`
: Update the current structure definition (i.e., geometry and topology) with the supplied dataset. Note that copying is done using reference counting.

`type = GetDataObjectType()`
: Return the type of data object (e.g., vtkDataObject, vtkPolyData, vtkImageData, vtkStructuredPoints, vtkStructuredGrid, vtkRectilinearGrid, or vtkUnstructuredGrid).

`numPoints = GetNumberOfPoints()`
: Return the number of points in the dataset.

`numCells = GetNumberOfCells()`
: Return the number of cells in the dataset.

`x = GetPoint(ptId)`
: Given a point id, return a pointer to the *(x,y,z)* coordinates of the point.

`GetPoint(ptId,x)`
: Given a point id, copy the *(x,y,z)* coordinates of the point into the array `x` provided. This is a thread-safe variant of the previous method.

`cell = GetCell(cellId)`
: Given a cell id, return a pointer to a cell object.

`GetCell(cellId, genericCell)`
: Given a cell id, return a cell in the user-provided instance of type vtkGenericCell. This is a thread-safe variant of the previous GetCell() method.

`type = GetCellType(cellId)`
: Return the type of the cell given by cell id. The type is an integer flag defined in the include file `vtkCellType.h`.

`GetCellTypes(types)`
: Generate a list of types of cells (supplied in `types`) that compose the dataset.

`cells = GetPointCells(ptId)`
: Given a point id, return a list of cells that use this point.

`GetCellPoints(cellId, ptIds)`
Given a cell id, return the point ids (e.g., connectivity list) defining the cell.

`GetCellNeighbors(cellId, ptIds, neighbors)`
Given a cell id and a list of points composing a boundary face or edge of the cell, return the neighbor(s) of that cell sharing the points.

`pointId = FindPoint(x)`
Locate the closest point to the global coordinate `x`. Return the closest point or `-1` if the point `x` is outside of the dataset.

`foundCellId = FindCell(x, cell, cellId, tol2, subId, pcoords, weights)`
Given a coordinate value `x`, an initial search cell defined by `cell` and `cellId`, and a tolerance measure (squared), return the cell id and sub-id of the cell containing the point and its interpolation function weights. The initial search cell (if `cellId>=0`) is used to speed up the search process when the position `x` is known to be near the cell. If no cell is found, `foundCellId` < 0 is returned.

`cell = FindAndGetCell(x, cell, cellId, tol2, subId, pcoords, weights)`
This is a variation of the previous method (FindCell()) that returns a pointer to the cell instead of the cell id.

`pointData = GetPointData()`
Return a pointer to the object maintaining point attribute data. This includes scalars, vectors, normals, tensors, texture coordinates, and field data.

`cellData = GetCellData()`
Return a pointer to the object maintaining cell attribute data. This includes scalars, vectors, normals, tensors, texture coordinates, and field data.

`bounds = GetBounds()`
Get the bounding box of the dataset. The return value is an array of `(xmin, xmax, ymin, ymax, zmin, zmax)`.

`GetBounds(bounds)`
Get the bounding box of the dataset. The return value is an array of `(xmin, xmax, ymin, ymax, zmin, zmax)`. This is a thread-safe variant of the previous method.

`length = GetLength()`
Return the length of the diagonal of the bounding box of the dataset.

`center = GetCenter()`
Get the center of the bounding box of the dataset.

`GetCenter(center)`
Get the center of the bounding box of the dataset. (A thread safe variant of the previous method).

`range = GetScalarRange()`
A convenience method to return the (minimum, maximum) range of the scalar attribute data associated with the dataset.

`GetScalarRange(range)`
A thread-safe variant of the previous method.

`Squeeze()`
Reclaim any extra memory used to store data. Typically used after creating and inserting data into the dataset.

Example

Here's a typical example of using the vtkDataSet API. The code fragment is from vtkProbeFilter. vtkProbeFilter samples data attribute values from one dataset onto the points of another (the filter has two inputs). Please ignore the references to point data attributes for now; these methods will be explained in "Interface To Field and Attribute Data" on page 270.

```
numPts = input->GetNumberOfPoints();
pd = source->GetPointData();

// Allocate storage for output PointData
outPD = output->GetPointData();
outPD->InterpolateAllocate(pd);

// Use tolerance as a function of size of source data
tol2 = source->GetLength();
tol2 = tol2*tol2 / 1000.0;

// Loop over all input points, interpolating source data
for (ptId=0; ptId < numPts; ptId++)
  {
  // Get the xyz coordinate of the point in the input dataset
  x = input->GetPoint(ptId);
```

```
// Find the cell that contains xyz and get it
cell = source->FindAndGetCell(x,NULL,
                        -1,tol2,subId,pcoords,weights);
if (cell)
  {
  // Interpolate the point data
  outPD->InterpolatePoint(pd,ptId,
                          &(cell->PointIds),weights);
  }
else
  {
  outPD->NullPoint(ptId);
  }
}
delete [] weights;
```

The following example shows how to create a reference-counted copy of a dataset using the vtkDataSet API. Both the variables `newDataSet` and `dataSet` are pointers to vtkDataSet.

```
newDataSet = dataSet->MakeObject();
newDataSet->CopyStructure(dataSet);
newDataSet->GetPointData()->PassData(
                                    dataSet->GetPointData())
newDataSet->GetCellData()->PassData(
                                    dataSet->GetCellData())
```

Now that we've covered the abstract API to vtkDataSet, we move to the concrete dataset types. Remember, every concrete subclass inherits the methods of its superclasses, including vtkDataSet.

11.4 Interface To vtkImageData

vtkImageData is a concrete dataset type representing a regular, *x-y-z* axis-aligned array of points. vtkImageData can represent 1D arrays, 2D images, and 3D volumes. Both the geometry and topology of the dataset structure are regular, and both are represented implicitly. A vtkImageData dataset is defined by data dimensions, interpoint spacing, and an origin of the lower-left corner of the dataset. If the dimension of the dataset is 2D, then we call vtkImageData dataset an image, and it is composed of vtkPixel cell types. If the dimension of the dataset is 3D, then we call vtkImageData dataset a volume, and it is composed of vtkVoxel cells.

Methods

`SetDimensions(i, j, k)`
Set the dimensions of the structured points dataset.

`SetDimensions(dim)`
An alternative form of the previous method where `dim` is an array of size 3.

`dims = GetDimensions()`
Return a pointer to an array of size 3 containing *i-j-k* dimensions of dataset.

`GetDimensions(dims)`
Thread-safe form of previous method.

`SetSpacing(sx, sy, sz)`
Set the spacing of the structured points dataset.

`SetSpacing(spacing)`
An alternative form of the previous method where `spacing` is an array of size 3.

`spacing = GetSpacing()`
Return a pointer to an array of size 3 containing *i-j-k* dimensions of dataset.

`GetSpacing(spacing)`
Thread-safe form of previous method.

`SetOrigin(x, y, z)`
Set the origin of the structured points dataset.

`SetOrigin(origin)`
An alternative form of the previous method where `origin` is an array of size 3.

`origin = GetOrigin()`
Return a pointer to an array of size 3 containing *i-j-k* dimensions of dataset.

`GetOrigin(origin)`
Thread-safe form of previous method.

`ComputeStructuredCoordinates(x, ijk, pcoords)`
Given a point *x* in the 3D modeling coordinate system, determine the structured coordinates *i-j-k* specifying which cell the point is in, as well as the parametric coordinates inside the cell.

`GetVoxelGradient(i, j, k, scalars, gradient)`
Given a cell specified by *i-j-k* structured coordinates, and the scalar data for the

structured points dataset, compute the gradient at each of the eight points defining the voxel.

`GetPointGradient(i, j, k, scalars, gradient)`
Given a point specified by *i-j-k* structured coordinates, and the scalar data for the structured points dataset, compute the gradient at the point (an array of size 3).

`d = GetDataDimension()`
Return the dimensionality of the dataset ranging from (0,3).

`pointId = ComputePointId(int ijk[3])`
Given a point specified by *i-j-k* structured coordinates, return the point id.

`cellId = ComputeCellId(int ijk[3])`
Given a cell specified by *i-j-k* structured coordinates, return the cell id of the point.

Examples

In this example, which is taken from the filter vtkExtractVOI, we subsample the input data to generate output data. In the initial portion of the filter (not shown), the dimensions, spacing, and origin of the output are determined and then set (shown). We then configure the output and copy the associated point attribute data.

```
output->SetDimensions(outDims);
output->SetSpacing(outAR);
output->SetOrigin(outOrigin);

// If output same as input, just pass data through
//
if ( outDims[0] == dims[0] && outDims[1] == dims[1] &&
outDims[2] == dims[2] &&
rate[0] == 1 && rate[1] == 1 && rate[2] == 1 )
  {
  output->GetPointData()->PassData(input->GetPointData());
  vtkDebugMacro(<<"Passed data through bacause input
                  and output are the same");
  return;
  }

// Allocate necessary objects
//
outPD->CopyAllocate(pd,outSize,outSize);
sliceSize = dims[0]*dims[1];
```

```
// Traverse input data and copy point attributes to output
//
newIdx = 0;
for ( k=voi[4]; k <= voi[5]; k += rate[2] )
  {
  kOffset = k * sliceSize;
  for ( j=voi[2]; j <= voi[3]; j += rate[1] )
    {
    jOffset = j * dims[0];
    for ( i=voi[0]; i <= voi[1]; i += rate[0] )
      {
      idx = i + jOffset + kOffset;
      outPD->CopyData(pd, idx, newIdx++);
      }
    }
  }
```

11.5 Interface To vtkPointSet

vtkPointSet is an abstract superclass for those classes that explicitly represent points (i.e., vtkPolyData, vtkStructuredGrid, vtkUnstructuredGrid). The basic function of vtkPointSet is to implement those vtkDataSet methods that access or manipulate points (e.g., GetPoint() or FindPoint()).

Methods

There are several access methods defined, but most overload vtkDataSet's API. Note that because this object is abstract, there are no direct creation methods. Refer to the vtkDataSet::MakeObject() creation method (see vtkDataSet "Methods" on page 242 for more information).

`numPoints = GetNumberOfPoints()`

Return the number of points in the dataset.

`points = GetPoints()`

Return a pointer to an object of type vtkPoints. This class explicitly represents the points in the dataset.

`SetPoints(points)`

Specify the explicit point representation for this object. The parameter `points` is an instance of vtkPoints.

Examples

Here's an example of a filter that exercises the vtkPointSet API. The following code is excerpted from vtkWarpVector.

```
inPts = input->GetPoints();
pd = input->GetPointData();
if ( !pd->GetVectors() || !inPts )
  {
  vtkErrorMacro(<<"No input data");
  return;
  }
inVectors = pd->GetVectors();
numPts = inPts->GetNumberOfPoints();
newPts = vtkPoints::New();
newPts->SetNumberOfPoints(numPts);

// Loop over all points, adjusting locations
//
for (ptId=0; ptId < numPts; ptId++)
  {
  x = inPts->GetPoint(ptId);
  v = inVectors->GetTuple(ptId);
  for (i=0; i<3; i++)
    {
    newX[i] = x[i] + this->ScaleFactor * v[i];
    }
  newPts->SetPoint(ptId, newX);
  }
```

The method to focus on is vtkPoint::GetPoints(), which returns a pointer to a vtkPoints instance (`inPts`). Also, notice that we use the invocation vtkPoints::GetPoint() to return the point coordinates of a particular point. We could replace this call with vtkPoint-Set::GetPoint() and achieve the same result.

11.6 Interface To vtkStructuredGrid

vtkStructuredGrid is a concrete dataset that represents information arranged on a topologically regular, but geometrically irregular, array of points. The cells are of type vtkHexahedron (in 3D) and of type vtkQuad (in 2D), and are represented implicitly with the Dimensions instance variable. The points are represented explicitly by the superclass vtkPointSet.

Methods

Most of vtkStructuredGrid's methods are inherited from it superclasses vtkPointSet and vtkDataSet.

`SetDimensions(i, j, k)`
: Specify the *i-j-k* dimensions of the structured grid. The number of points for the grid, specified by the product *i*j*k*, must match the number of points returned from vtkPointSet::GetNumberOfPoints().

`SetDimensions(dim)`
: An alternative form of the previous method. Dimensions are specified with an array of three integer values.

`dims = GetDimensions()`
: Return a pointer to an array of size 3 containing *i-j-k* dimensions of dataset.

`GetDimensions(dims)`
: Thread-safe form of the previous method.

`dim = GetDataDimensions()`
: Return the dimension of the dataset, i.e., whether the dataset is 0, 1, 2, or 3-dimensional.

11.7 Interface To vtkRectilinearGrid

Description

vtkRectilinearGrid is a concrete dataset that represents information arranged on a topologically regular, and geometrically semi-regular, array of points. The points are defined by

three vectors that contain coordinate values for the *x*, *y*, and *z* axes—thus the points are axis-aligned and only partially represented. The cells that make up vtkRectilinearGrid are implicitly represented and are of type vtkVoxel (3D) or vtkPixel (2D).

Creating a vtkRectilinearGrid requires specifying the dataset dimensions and three arrays defining the coordinates in the *x*, *y*, *z* directions (these arrays are represented by the XCoordinates, YCoordinates, and ZCoordinates instance variables). Make sure that the number of coordinate values is consistent with the dimensions specified.

Methods

The class vtkRectilinearGrid defines several methods beyond the inherited methods from vtkDataSet.

`SetDimensions(i, j, k)`
: Set the dimensions of the rectilinear grid dataset. Note that the dimension values in the *x*-*y*-*z* directions must match the number of values found in the XCoordinates, YCoordinates, and ZCoordinates arrays.

`SetDimensions(dim)`
: An alternative form of the previous method where `dim` is an array of size 3.

`dims = GetDimensions()`
: Return a pointer to an array of size 3 containing *i*-*j*-*k* dimensions of dataset.

`GetDimensions(dims)`
: Thread-safe form of previous method.

`ComputeStructuredCoordinates(x, ijk, pcoords)`
: Given a point in the 3D modeling coordinate system, determine the structured coordinates *i*-*j*-*k* specifying which cell the point is in, as well as the parametric coordinates inside the cell.

`GetVoxelGradient(i, j, k, scalars, gradient);`
: Given a cell specified by *i*-*j*-*k* structured coordinates, and the scalar data for the rectilinear grid dataset, compute the gradient at each of the eight points defining the voxel.

`GetPointGradient(i, j, k, scalars, gradient)`
: Given a point specified by *i*-*j*-*k* structured coordinates, and the scalar data for the rectilinear grid dataset, compute the gradient at the point (an array of size 3).

`d = GetDataDimension()`
Return the dimensionality of the dataset ranging from (0,3).

`pointId = ComputePointId(int ijk[3])`
Given a point specified by *i-j-k* structured coordinates, return the point id of the point.

`cellId = ComputeCellId(int ijk[3])`
Given a cell specified by *i-j-k* structured coordinates, return the cell id of the point.

`SetXCoordinates(xcoords)`
Specify the array of values which define the *x* coordinate values. The array `xcoords` is of type vtkDataArray.

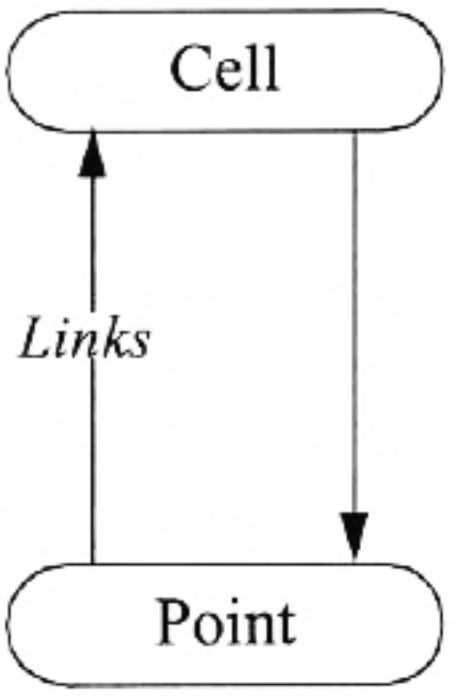
`SetYCoordinates(ycoords)`
Specify the array of values which define the *y* coordinate values. The array `ycoords` is of type vtkDataArray.

`SetZCoordinates(zcoords)`
Specify the array of values which define the *z* coordinate values.The array `zcoords` is of type vtkDataArray.

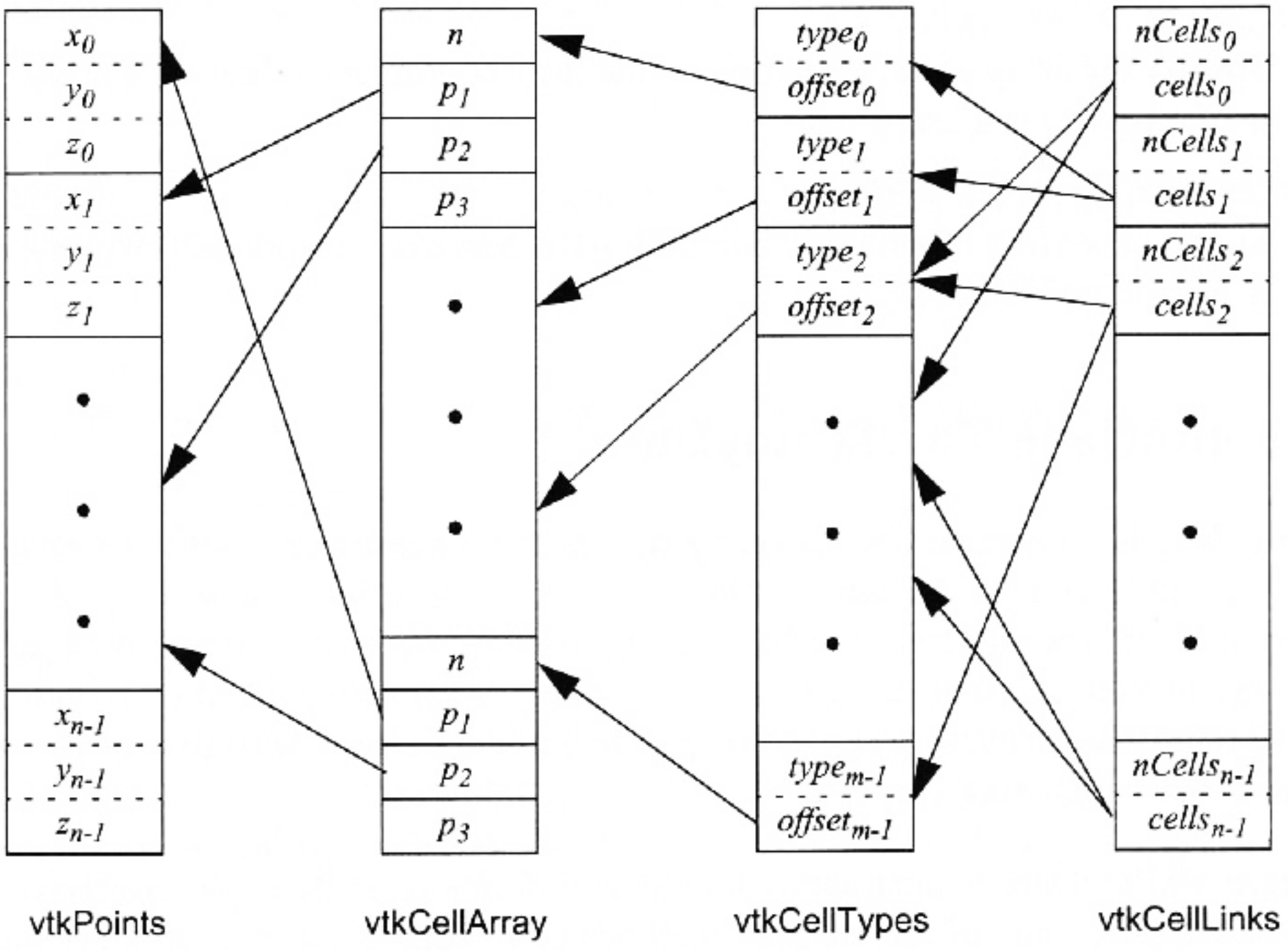
11.8 Interface To vtkPolyData

vtkPolyData is a concrete dataset type that represents rendering primitives such as vertices, lines, polygons, and triangle strips. The data is completely unstructured: points are represented in the superclass vtkPointSet, and the cells are represented using four instances of vtkCellArray, which is a connectivity list (**Figure 11–3**). The four vtkCellArrays represent vertices and polyvertices; lines and polylines; triangles, quads, and polygons; and triangle strips, respectively.

Because vtkPolyData is unstructured, cells and points must be explicitly represented. In order to support some of the required methods of its superclass vtkDataSet (mainly topological methods such as GetPointCells() and GetCellNeighbors()), vtkPolyData has a complex internal data structure as shown in **Figure 11–3**. Besides vtkPoints and the vtkCellArrays, the data structure consists of a list of cell types (vtkCellTypes) and cell links (vtkCellLinks). The cell types allows random access into the cells (note that the vtkCellArray cannot support random access) and the cell links supports topological operations by maintaining references to all cells that use a particular vertex (see *The Visualization Tool-*



(a) Full unstructured data representation for vtkPolyData and vtkUnstructuredGrid. Cells refer to the points that define them; points refer to the cells that use them. This structure can be used to determine vertex, edge, and face neighbors and other topological information.



(b) Unstructured data is represented by points (geometry and position in 3D space), a cell array (cell connectivity), cell types (provides random access to cells), and cell links (provides topological information). The cell types and cell links arrays are only constructed if required.

Figure 11–3 Representing unstructured data. This structure is used to represent polygonal and unstructured grid datasets.

kit text for more information). Instances of these two classes vtkCellTypes and vtkCellLinks are only instantiated if needed. That is, if random access to cells is required vtkCellTypes is instantiated; or if topological information is required, vtkCellLinks is instantiated.

While this structure is fairly complex, the good news is that for the most part the management of the internal structure is taken care of for you. Normally you'll never need to directly manipulate the structure as long as you use vtkDataSet's API to interface the information. In some cases, writing a filter or using some of the vtkPolyData or vtkUnstructuredGrid methods, you may have to explicitly create the cell types and/or cell links arrays using the BuildCells() and BuildLinks() methods. In rare cases you may want to directly manipulate the structure—deleting points or cells, and/or modifying the link array to reflect changing topology. Refer to section "Miscellaneous Interfaces" on page 264 for more information. Examples of code demonstrating the use these complex operators include the classes `Patented/vtkDecimate`, `Graphics/vtkDecimatePro`, and `Graphics/vtkDelaunay2D`.

Methods

The class vtkPolyData is fairly complex, overloading many of the methods inherited from its superclasses vtkDataSet and vtkPointSet. Most of the complexity is due to the definition of vertices, lines, polygons, and triangle strips in separate vtkCellArray's, and special (geometric) methods for operating on the mesh.

`SetVerts (verts)`
Specify the list of vertices `verts`. The parameter `verts` is an instance of vtkCellArray. Note: the difference between vertices and points are that vertices are rendered, while points are not. Points define cell geometry, vertices represent a cell.

`verts = GetVerts()`
Get the list of vertices. The list is an instance of vtkCellArray.

`SetLines (lines)`
Specify the list of lines `lines`. The parameter `lines` is an instance of vtkCellArray.

`lines = GetLines()`
Get the list of lines. The list is an instance of vtkCellArray.

`SetPolys(polys)`
Specify the list of polygons `polys`. The parameter `polys` is an instance of vtkCellArray.

`polys = GetPolys()`
Get the list of polygons. The list is an instance of vtkCellArray.

`SetStrips(strips)`
Specify the list of triangle strips `strips`. The parameter `strips` is an instance of vtkCellArray.

`strips = GetStrips()`
Get the list of triangle strips. The list is an instance of vtkCellArray.

`numVerts = GetNumberOfVerts()`
Return the number of vertices.

`numLines = GetNumberOfLines()`
Return the number of lines.

`numPolys = GetNumberOfPolys()`
Return the number of polygons.

`numStrips = GetNumberOfStrips()`
Return the number of triangle strips.

`Allocate(numCells, extend)`
Perform initial memory allocation prior to invoking the InsertNextCell() methods (described in next two items). The parameter `numCells` is an estimate of the number of cells to be inserted; `extend` is the size to extend the internal structure (if needed).

`cellId = InsertNextCell(type, npts, pts)`
Given a cell type `type`, number of points in the cell `npts`, and an integer list of point ids `pts`, insert a cell and return its cell id. See **Figure 14–15** for a definition of type values. Make sure to invoke Allocate() prior to invoking this method.

`cellId = InsertNextCell(type, pts)`
Given a cell type `type` and instance of `vtkIdList`, `pts`, insert a cell and return its cell id. Make sure to invoke Allocate() prior to invoking this method.

`Reset()`
Restores this instance of vtkPolyData to its initial condition without releasing allocated memory.

`BuildCells()`
Build the internal vtkCellTypes array. This allows random access to cells (e.g., GetCell() and other special vtkPolyData methods). Normally you don't need to invoke this method unless you are doing specialized operations on vtkPolyData.

`BuildLinks()`
Build the internal vtkCellLinks array. This enables access to topological information such as neighborhoods (e.g., vertex, edge, face neighbors). Normally you don't need to invoke this method unless you are doing special operations on vtkPolyData.

`GetPointCells(ptId, ncells, cells)`
Given a point id `ptId`, return the number of cells using the point `ncells`, and an integer array of cell ids that use the point `cells`.

`GetCellEdgeNeighbors(cellId, p1, p2, cellIds)`
Given a cell `cellId` and two points `p1` and `p2` forming an edge of the cell, fill in a user-supplied list of all cells using the edge (`p1`,`p2`).

`GetCellPoints(cellId, npts, pts)`
Given a cell `cellId`, return the number of points `npts` and an integer list of point ids which define the cell connectivity. This is a specialized version of the method inherited from vtkUnstructuredGrid superclass' GetCellPoints(npts, ptIds).

`flag = IsTriangle(p1, p2, p3)`
A special method meant for triangle meshes. Returns a 0/1 flag indicating whether the three points listed (`p1`,`p2`,`p3`) form a triangle in the mesh.

`flag = IsEdge(p1, p2)`
Returns a 0/1 flag indicating whether the two points listed (`p1`,`p2`) forms a edge in the mesh.

`flag = IsPointUsedByCell(ptId,cellId)`
Return a 0/1 flag indicating whether a particular point given by `ptId` is used by a particular cell `cellId`.

`ReplaceCell(cellId, npts, pts)`
Redefine a cell `cellId` with a new connectivity list `pts`. Note that the number of points `npts` must be equal to the original number of points in the cell.

`ReplaceCellPoint(cellId, oldPtId, newPtId)`
Redefine a cell's connectivity list by replacing one point id (`oldPtId`) with a new point id (`newPtId`).

`ReverseCell(cellId)`

Reverse the order of the connectivity list definition for the cell `cellId`. For example, if a triangle is defined (`p1,p2,p3`), after invocation of this method it will be defined by the points (`p3,p2,p1`).

`DeletePoint(ptId)`

Delete the point by removing all links from it to using cells. This method does not actually remove the point from the vtkPoints object.

`DeleteCell(cellId)`

Delete a cell by marking its type as `VTK_NULL_ELEMENT`. This operator only modified the vtkCellTypes array, it does not actually remove the cell's connectivity from the vtkCellArray object.

`ptId = InsertNextLinkedPoint(x, numLinks)`

If the instance of vtkCellLinks has been built, and you wish to insert a new point into the mesh, use this method. The parameter `numLinks` is the initial size of the list of cells using the point.

`cellId = InsertNextLinkedCell(type, npts, pts)`

Insert a new cell into the vtkPolyData after the cell links and cell types structures have been built (i.e., BuildCells() and BuildLinks() have been invoked).

`ReplaceLinkedCell(cellId, npts, pts)`

Replace one linked cell with another cell. Note that `npts` must be the same size for the replaced and replacing cell.

`RemoveCellReference(cellId)`

Remove all references to the cell `cellId` from the cell links structure. This effectively topologically disconnects the cell from the data structure.

`AddCellReference(cellId)`

Add references to the cell `cellId` into the cell links structure. That is, the links from all points are modified to reflect the use by the cell `cellId`.

`RemoveReferenceToCell(ptId,cellId)`

Remove the reference from the links of `ptId` to the cell `cellId`.

`AddReferenceToCell(ptId,cellId)`

Add a reference to the cell `cellId` into the point `ptId`'s link list.

`ResizeCellList(ptId, size)`

Resize the link list for the point `ptId` and make it of size `size`.

11.9 Interface To vtkUnstructuredGrid

vtkUnstructuredGrid is a concrete dataset type that represents all possible combinations of VTK cells (i.e., all those shown on **Figure 14–15**). The data is completely unstructured. Points are represented by the superclass vtkPointSet, and cells are represented by a combinations of objects including vtkCellArray, vtkCellTypes, and vtkCellLinks. Typically, these objects (excluding vtkPoints) are used internally and you do not directly manipulate them. Refer to the previous section "Interface To vtkPolyData" on page 253 for additional information about the unstructured data structure in VTK. Also, see "Miscellaneous Interfaces" on page 264 for detailed interface information about vtkCellArray, vtkCellTypes, and vtkCellLinks.

Although vtkUnstructuredGrid and vtkPolyData are similar, there are substantial differences. vtkPolyData can only represent cells of topological dimension 2 or less (i.e., triangle strips, polygons, lines, vertices) while vtkUnstructuredGrid can represent cells of dimension 3 and less. Also, vtkUnstructuredGrid maintains an internal instance of vtkCellTypes to allow random access to its cells. vtkPolyData will instantiate vtkCellTypes only when random access is required. Finally, vtkUnstructuredGrid maintains a single internal instance of vtkCellArray to represent cell connectivity; vtkPolyData maintains four separate arrays corresponding to triangle strips, polygons, lines, and vertices.

Access Methods

`Allocate(numCells, extend)`

Perform initial memory allocation prior to invoking the InsertNextCell() methods (described in the following two items). The parameter `numCells` is an estimate of the number of cells to be inserted; `extend` is the size to extend the internal structure (if needed).

`cellId = InsertNextCell(type, npts, pts)`

Given a cell type `type`, number of points in the cell `npts`, and an integer list of point ids `pts`, insert a cell and return its cell id. See **Figure 14–15** for a definition of type values. Make sure to invoke Allocate() prior to invoking this method.

`cellId = InsertNextCell(type, ptIds)`

Given a cell type `type` and instance of vtkIdList, `ptIds`, insert a cell and return its cell id. Make sure to invoke Allocate() prior to invoking this method.

`Reset()`

Restores this instance of vtkUnstructuredGrid to its initial condition without releasing allocated memory.

`SetCells(types, cells)`

This is a high performance method that allows you to define a set of cells all at once. You specify an integer list of cell types, `types`, followed by an instance of vtkCellArray.

`cells = GetCells()`

Return a pointer to the cell connectivity list. The return value `cells` is of type vtkCellArray.

`BuildLinks()`

Build the internal vtkCellLinks array. This enables access to topological information such as neighborhoods (e.g., vertex, edge, face neighbors). Normally you don't need to invoke this method unless you are doing specialized operations on vtkUnstructuredGrid.

`GetCellPoints(cellId, npts, pts)`

Given a cell `cellId`, return the number of points `npts` and an integer list of point ids that define the cell connectivity. This is a specialized version of the method inherited from vtkUnstructuredGrid superclass' GetCellPoints(npts, ptIds).

`ReplaceCell(cellId, npts, pts)`

Redefine the cell `cellId` with a new connectivity list `pts`. Note that the number of points `npts` must be equal to the original number of points in the cell.

`cellId = InsertNextLinkedCell(type, npts, pts)`

Insert a new cell into the vtkUnstructuredGrid after the cell links structure has been built (i.e., the method BuildLinks() has been invoked).

`RemoveReferenceToCell(ptId, cellId)`

Remove the reference from the links of `ptId` to the cell `cellId`.

`AddReferenceToCell(ptId, cellId)`

Add a reference to the cell `cellId` into the point `ptId`'s link list.

`ResizeCellList(ptId, size)`

Resize the link list for the point `ptId` and make it of size `size`.

11.10 Interface To Cells (Subclasses of vtkCell)

Cells are the atoms of datasets. They are used to perform local operations within the dataset such as interpolation, coordinate transformation and searching, and various geometric operations. Cells are often used as a "handle" into a dataset, returning local information necessary to the execution of an algorithm. To obtain a cell from a dataset, the method GetCell(int cellId) is used, and the returned cell can then be processed.

The class vtkCell represents data using an instance of vtkPoints and vtkIdList (representing point coordinates and point ids, respectively). These two instance variables are publicly accessible, one of the few exceptions in VTK where ivars are public. The following methods are available to all subclasses of vtkCell.

`type = GetCellType()`
Return the type of the cell as defined in `vtkCellType.h`.

`dim = GetCellDimension()`
Return the topological dimensional of the cell (0,1,2, or 3).

`order = GetInterpolationOrder()`
Return the interpolation order of the cell. Usually linear (=1).

`points = GetPoints()`
Get the point coordinates for the cell.

`numPts = GetNumberOfPoints()`
Return the number of points in the cell.

`numEdges = GetNumberOfEdges()`
Return the number of edges in the cell.

`numFaces = GetNumberOfFaces()`
Return the number of faces in the cell.

`ptIds = GetPointIds()`
Return the list of point ids defining the cell

`id = GetPointId(ptId)`
For cell point i, return the actual point id.

`edge = GetEdge(edgeId)`
Return the edge cell from the `edgeId` of the cell

`cell = GetFace(faceId)`

Return the face cell from the `faceId` of the cell.

`status = CellBoundary(subId, pcoords[3], pts)`

Given parametric coordinates of a point, return the closest cell boundary, and whether the point is inside or outside of the cell. The cell boundary is defined by a list of points (pts) that specify a face (3D cell), edge (2D cell), or vertex (1D cell). If the return value of the method is != 0, then the point is inside the cell.

`status = EvaluatePosition(x[3], closestPoint[3], subId, pcoords[3], dist2, weights[])`

Given a point `x[3]` return inside(=1) or outside(=0) cell; evaluate parametric coordinates, sub-cell id (!=0 only if cell is composite), distance squared of point `x[3]` to cell (in particular, the sub-cell indicated), closest point on cell to `x[3]`, and interpolation weights in cell. (The number of weights is equal to the number of points defining the cell). Note: on rare occasions a -1 is returned from the method. This means that numerical error has occurred and all data returned from this method should be ignored.

`EvaluateLocation(subId, pcoords[3], x[3], weights[])`

Determine global coordinate (`x[3]`) from subId and parametric coordinates. Also returns interpolation weights. (The number of weights is equal to the number of points in the cell.)

`Contour(value, cellScalars, locator, verts, lines, polys, inPd, outPd, inCd, cellId, outCd)`

Generate contouring primitives. The scalar list `cellScalars` are scalar values at each cell point. The `locator` is essentially a points list that merges points as they are inserted (i.e., prevents duplicates). Contouring primitives can be vertices, lines, or polygons. It is possible to interpolate point data along the edge by providing input and output point data - if `outPd` is NULL, then no interpolation is performed. Also, if the output cell data (`outCd`) is non-NULL, the cell data from the contoured cell is passed to the generated contouring primitives.

`Clip(value, cellScalars, locator, connectivity, inPd, outPd, inCd, cellId, outCd, insideOut)`

Cut (or clip) the cell based on the input `cellScalars` and the specified value. The output of the clip operation will be one or more cells of the same topological dimension as the original cell. The flag `insideOut` controls what part of the cell is considered inside - normally cell points whose scalar value is greater than "value" are considered inside. If `insideOut` is on, this is reversed. Also, if the output cell data

is non-NULL, the cell data from the clipped cell is passed to the generated contouring primitives.

`status = IntersectWithLine(p1[3], p2[3], tol, t, x[3], pcoords[3], subId)`

Intersect with the ray defined by `p1` and `p2`. Return parametric coordinates (both line and cell). A non-zero return value indicates that an intersection has occurred. You can specify a tolerance `tol` on the intersection operation.

`status = Triangulate(index, ptIds, pts)`

Generate simplices of the appropriate dimension that approximate the geometry of this cell. 3D cells will generate tetrahedra,; 2D cells triangles; and so on. If triangulation failure occurs, a zero is returned.

`Derivatives(subId, pcoords[3], values, dim, derivs)`

Compute the derivatives of the values given for this cell.

`GetBounds(bounds[6])`

Set the $(x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max})$ bounding box values of the cell.

`bounds = GetBounds()`

Return a pointer to the bounds of the cell.

`length2 = GetLength2()`

Return the length squared of the cell (the length is the diagonal of the cell's bounding box).

`status = GetParametricCenter(pcoords[3])`

Return the parametric coordinates of the center of the cell. If the cell is a composite cell, the particular subId that the center is in is returned.

`status = HitBBox(bounds[6], origin[3], dir[3], coord[3], t)`

Return non-zero value if cell bounding box is hit by ray starting at `origin` and in direction of `dir`. Intersection point and parametric coordinate of intersection are returned (valid intersection when 0 <= `t` <= 1).

`void ShallowCopy(cell)`

Perform shallow (reference counted) copy of the cell.

`void DeepCopy(cell)`

Perform deep (copy of all data) copy of the cell.

11.11 Miscellaneous Interfaces

We saw earlier that some dataset types required instantiation and manipulation of component objects. For example, vtkStructuredGrid requires the creation of an instance of vtkPoints to define its point locations, and vtkPolyData requires the instantiation of vtkCellArray to define cell connectivity. In this section we describe the interface to these supporting objects. You may want to refer to **Figure 11–3** for additional information describing the relationship of these objects.

vtkPoints

vtkPoints represents *x*-*y*-*z* point coordinate information. Instances of vtkPoints are used to explicitly represent points. vtkPoints depends on an internal instance of vtkDataArray, thereby supporting data of different native types (i.e., `int`, `float`, etc.) The methods for creating and manipulating `vtkPoints` are shown below.

`num = GetNumberOfPoints()`
: Return the number of points in the array.

`x = GetPoint(int id)`
: Return a pointer to an array of three floats: the *x*-*y*-*z* coordinate position. This method is not thread safe.

`GetPoint(id, x)`
: Given a point is `id`, fill in a user-provided `float` array of length 3 with the *x*-*y*-*z* coordinate position.

`SetNumberOfPoints(number)`
: Specify the number of points in the array, allocating memory if necessary. Use this method in conjunction with SetPoint() to put data into the vtkPoints array.

`SetPoint(id, x)`
: Directly set the point coordinate `x` at the location `id` specified. Range checking is not performed and as a result, is faster than the insertion methods. Make sure SetNumberOfPoints() is invoked prior to suing this method.

`InsertPoint(id, x)`
: Insert the point coordinate `x` at the location `id` specified. Range checking is performed and memory allocated as necessary.

`InsertPoint(id, x, y, z)`

Insert the point coordinate (x,y,z) at the location `id` specified. Range checking is performed and memory allocated as necessary.

`pointId = InsertNextPoint(x)`

Insert the point coordinate `x` at the end of the array returning its point id. Range checking is performed and memory allocated as necessary.

`pointId = InsertNextPoint(x, y, z)`

Insert the point coordinate (x,y,z) at the end of the array returning its point id. Range checking is performed and memory allocated as necessary.

`GetPoints(ptIds, pts)`

Given a list of points `ptIds`, fill-in a user-provided vtkPoints instance with corresponding coordinate values.

`bounds = GetBounds()`

Return a pointer to an array of size 6 containing the $(x_{min},x_{max}, y_{min},y_{max}, z_{min},z_{max})$ bounds of the points. This method id not thread-safe.

`GetBounds(bounds)`

Fill in a user-provided array of size 6 with the $(x_{min},x_{max}, y_{min},y_{max}, z_{min},z_{max})$ bounds of the points.

vtkCellArray

vtkCellArray represents the topology (i.e., connectivity) information of cells. The connectivity of a cell is defined by an ordered integer list of point ids. In addition, cells are defined by a `type` flag, indicating what type of cell they are (see **Figure 14–15** for more information about type flags). The methods for creating and manipulating vtkCellArrays are shown below.

`Allocate(size, extend)`

Perform initial memory allocation prior to invoking the InsertNextCell() methods. The parameter `numCells` is an estimate of the number of cells to be inserted; `extend` is the size to extend the internal structure (if needed).

`Initialize()`

Change the object to its original state, releasing memory that may have been allocated.

`size = EstimateSize(numCells, PtsPerCell)`

Estimate the size of the data to insert based on the number of cells, and the expected number of points per cell. This method returns an estimate; use it in conjunction with the Allocate() method to obtain initial memory allocation.

`void InitTraversal()`

Initialize the traversal of the connectivity array. Used in conjunction with GetNext-Cell().

`nonEmpty = GetNextCell(npts, pts)`

Get the next cell in the list, returning the number of points `npts` and an integer array of point ids `pts`. If the list is empty, return 0; non-zero otherwise.

`size = GetSize()`

Return the allocated size of the list.

`numEntries = GetNumberOfConnectivityEntries()`

Return the total number of entries in the list. This `size` parameter represents the total number of integer values required to represent the connectivity list.

`GetCell(loc, npts, pts)`

A special method to return the cell at location offset `loc`. This method is typically used by unstructured data for random access into the cell array. Note that `loc` is not the same as a cell id, it is a offset into the cell array.

`cellId = InsertNextCell(cell)`

Insert a cell into the array. The parameter `cell` is of type vtkCell. Return the cell id.

`cellId = InsertNextCell(npts, pts)`

Insert a cell into the array by specifying the number of points `npts` and an integer list of cell ids. Return the cell id.

`cellId = InsertNextCell(pts)`

Insert a cell into the array by supplying an id list (vtkIdList). Return the cell id.

`cellId = InsertNextCell(npts)`

Insert a cell into the cell array by specifying the number of cell points. This method is generally followed by multiple invocations of InsertCellPoint() to define the cell points, and possibly UpdateCellCount() to specify the final number of points.

`InsertCellPoint(id)`

Insert a cell point into the cell array. This method requires prior invocation of Insert-NextCell(npts).

`UpdateCellCount(npts)`

Specify the final number of points defining a cell after invoking InsertNextCell(npts) and InsertCellPoint(). This method allows you to adjust the number of points after estimating an initial point count with InsertNextCell(npts).

`location = GetInsertLocation(npts)`

Return the current insertion location in the cell array. The insertion location is used by methods such as InsertNextCell(). The location is an offset into the cell array.

`location = GetTraversalLocation(npts)`

Return the current traversal location in the cell array. The insertion location is used by methods such as GetCell(). The location is an offset into the cell array.

`ReverseCell(loc)`

Reverse the order of the connectivity list definition for the cell `cellId`. For example, if a triangle is defined (`p1`,`p2`,`p3`), after invocation of this method it will be defined by the points (`p3`,`p2`,`p1`).

`ReplaceCell(loc, npts, pts)`

Redefine the cell at offset location `loc` with a new connectivity list `pts`. Note that the number of points `npts` must be equal to the original number of points in the cell array.

`GetMaxCellSize()`

Return the maximum length of any cell in the cell array connectivity list.

`ptr = GetPointer()`

Return an integer pointer to the cell array. The structure of the data in the returned data is then number of points in a cell, followed by its connectivity list, which repeats for each cell: (`npts`, p_0, p_1, p_2, . . . , p_{npts-1}; `npts`, p_0, p_1, p_2, . . . , p_{npts-1}, . . .).

`ptr = WritePointer(ncells, size)`

Allocate memory for a cell array with `ncells` cells and of size specified. The `size` includes the connectivity entries as well as the count for each cell.

`Reset()`

Restore the object to its initial state with the exception that previously allocated memory is not released.

`Squeeze()`

Recover any unused space in the array.

vtkCellTypes

The class vtkCellTypes provides random access to cells. Instances of vtkCellTypes are always associated with one or more instances of vtkCellArray, which actually defines the connectivity list for the cells. The information contained in the vtkCellTypes is a type specified (an integer flag as defined in **Figure 14–15**) and a location offset, which an integer value representing offset into the associated vtkCellArray.

`Allocate(size, extend)`
Perform initial memory allocation prior to invoking the InsertNextCell() methods. The parameter `size` is an estimate of the number of cells to be inserted; `extend` is the size to extend the internal structure (if needed).

`cellId = InsertCell(id, type, loc)`
Given a cell type `type` and its location offset in an associated vtkCellArray, insert the cell type at the location (`id`) specified and return its cell id.

`cellId = InsertNextCell(type, loc)`
Given a cell type `type` and its location offset in an associated vtkCellArray, insert the cell type at the end of the array and return its cell id.

`DeleteCell(cellId)`
Delete a cell by marking its type as `VTK_NULL_ELEMENT`.

`numTypes = GetNumberOfTypes()`
Return the number of different types of cells in the vtkCellTypes array. Used to determine whether the array contains a homogeneous set of cells.

`IsType(type)`
Return 1 if type specified is contained in the vtkCellTypes array, or zero otherwise.

`cellId = InsertNextType(type)`
Insert the next type into the list.

`type = GetCellType(id)`
Return the type of the cell give by `id`.

`loc = GetCellLocation(id)`
Get the offset location into an associated vtkCellArray instance for the cell given by `id`.

`cell_s = GetCell(id)`
: Return a pointer to a structure containing the cell type and offset location for the given cell `id`.

`Squeeze()`
: Recover any unused space in the array.

`Reset()`
: Restore the object to its initialize state with the exception that previously allocated memory is not released.

vtkCellLinks

The class vtkCellLinks provides topological information describing the use of points by cells. Think of the vtkCellLinks object as a list of lists of cells using a particular point. (See **Figure 11–3**.) This information is used to derive secondary topological information such as face, edge, and vertex neighbors. Instances of vtkCellLinks are always associated with a vtkPoints instance, and access to the cells is through vtkCellTypes and vtkCellArray objects.

`link_s = GetLink(ptId)`
: Return a pointer to a structure containing the number of cells using the point `pointId`, as well as a pointer to a list of cells using the point.

`ncells = GetNcells(pointId)`
: Return the number of cells using the point `pointId`.

`BuildLinks(dataset)`
: Given a pointer to a VTK dataset, build the link topological structure.

`cellList = GetCells(ptId)`
: Return a pointer to the list of cells using the point `ptId`.

`ptId = InsertNextPoint(numLinks)`
: Allocate (if necessary) and insert space for a link at the end of the cell links array.

`InsertNextCellReference(ptId, cellId)`
: Insert a reference to the cell `cellId` for the point `ptId`. This implies that `cellId` uses the point `ptId` in its definition.

`DeletePoint(ptId)`

Delete the point by removing all links from it to using cells. This method does not actually remove the point from the vtkPoints object.

`RemoveCellReference(cellId, ptId)`

Remove all references to the cell `cellId` from the point `ptId`'s list of using cells.

`AddCellReference(cellId, ptId)`

Add a reference to the cell `cellId` to the point `ptId`'s list of using cells.

`ResizeCellList(ptId, size)`

Allocate (if necessary) and resize the list of cells using the point `ptId` to the size given.

`Squeeze()`

Recover any unused space in the array.

`Reset()`

Restore the object to its initial state with the exception that previously allocated memory is not released.

11.12 Interface To Field and Attribute Data

The previous section described how to create, access, and generate the structure of datasets. In this section we describe a class and its methods used to manage the processing of dataset attributes (scalars, vectors, tensors, normals, and texture coordinates) and field data as the filter executes. vtkDataSetAttributes and vtkFieldData are used for this purpose. They provide a number of convenience methods for copying, interpolating, and passing data from a filter's input to its output.

vtkFieldData Methods

As of VTK 4.0, vtkFieldData is the superclass of vtkDataSetAttributes (and therefore of vtkPointData and vtkCellData which inherit from vtkDataSetAttributes). Therefore, all fields and attributes (scalars, vectors, normals, tensors, texture coordinates) are stored in the field data and can be easily interchanged (see "Working With Field Data" on page 194 for more information on manipulating fields). It is now possible to associate a field (vtkDataArray) to, for example, vtkPointData and label it as the active vector array afterwards.

`PassData(fromData)`
Copy the field data from the input (fromData) to the output. Reference counting is used, and the copy flags (e.g., CopyFieldOn/Off) are used to control which fields are copied.

`num = GetNumberOfArrays()`
Get the number of arrays currently in the field data.

`array = GetArray(index)`
Given an index, return the corresponding array.

`array = GetArray(name)`
Given a name, return the corresponding array.

`AddArray(array)`
Add a field (vtkDataArray).

`RemoveArray(name)`
Remove the array with the given name.

`DeepCopy(data)`
Copy the instance data. Performs a deep copy, which means duplicating allocated memory.

`ShallowCopy(data)`
Copy the instance data. Performs a shallow copy, which means reference counting underlying data objects.

`Squeeze()`
Recover an extra space used by the attribute data.

`mtime = GetMTime()`
Return the modified time of this object by examining its own modified time as well as the modified time of the associated fields (arrays).

`CopyFieldOn/Off(name)`
These methods are used to control the copying and interpolation of individual fields from the input to the output. If off, the field with the given name is not copied or interpolated.

vtkDataSetAttributes Methods

Recall that there are two types of attributes: those associated with the points of the dataset (vtkPointData) and those associated with the cells of the dataset (vtkCellData). Both of these classes are subclasses of vtkDataSetAttributes (which is a subclass of vtkFieldData) and have nearly identical interfaces The following methods are defined by vtkDataSetAttributes and are common to both vtkPointData and vtkCellData. Remember: all datasets have attribute data (both cell and point), so all datasets' attribute data respond to these methods.

`PassData(fromData)`

Copy the attribute data from the input (fromData) to the output attribute data. Reference counting is used, and the copy flags (e.g., CopyScalars) are used to control which attribute data is copied.

`CopyAllocate(fromData, size, extend)`

Allocate memory and initialize the copy process. The copy process involves copying data on an item by item basis (e.g., point by point or cell by cell). The initial allocated size of each vtkAttributeData object is given by size, if the objects must be dynamically resized during the copy process, then the objects are extended by extend.

`CopyData(fromData, fromId, toId)`

Copy the input attribute data (fromData) at location fromId to location toId in the output attribute data.

`InterpolateAllocate(fromData, size, extend)`

Allocate memory and initialize the interpolate process. The interpolation process involves interpolating data across a cell or cell topological feature (e.g., an edge). The initial allocated size of each vtkAttributeData object is given by size, if the objects must be dynamically resized during the interpolate process, then the objects are extended by extend.

`InterpolatePoint(fromData, toId, Ids, weights)`

Interpolate from the dataset attributes given (fromData) to the point specified (toId). The interpolation is performed by summing the product of the attribute values given at each point in the list Ids with the interpolation weights provided.

`InterpolateEdge(fromData, toId, id1, id2, t)`

Similar to the previous method, except that the interpolation is performed between id1 and id2 using the parametric coordinate t.

`DeepCopy(data)`
Copy the instance data. Performs a deep copy, which means duplicating allocated memory.

`ShallowCopy(data)`
Copy the instance data. Performs a shallow copy, which means reference counting underlying data objects.

`Squeeze()`
Recover an extra space used by the attribute data.

`mtime = GetMTime()`
Return the modified time of this object by examining its own modified time as well as the modified time of the associated attribute data.

`SetScalars(scalars)`
Specify the scalar (vtkDataArray) attribute data.

`scalars = GetScalars()`
Retrieve the scalar (vtkDataArray) attribute data.

`SetVectors(vectors)`
Specify the vector (vtkDataArray) attribute data.

`vectors = GetVectors()`
Retrieve the vector (vtkDataArray) attribute data.

`SetTensors(tensors)`
Specify the tensor (vtkDataArray) attribute data.

`tensors = GetTensors()`
Retrieve the tensor (vtkDataArray) attribute data

`SetNormals(normals)`
Specify the normal (vtkDataArray) attribute data.

`normals = GetNormals()`
Retrieve the normal (vtkDataArray) attribute data.

`SetTCoords(tcoords)`
Specify the texture coordinate (vtkDataArray) attribute data.

`tcoords = GetTCoords()`
Retrieve the texture coordinate (vtkDataArray) attribute data.

`CopyScalarsOn/Off()`

These methods are used to control the copying and interpolation of scalar data from the input to the output. If off, scalar data is not copied or interpolated.

`CopyVectorsOn/Off()`

These methods are used to control the copying and interpolation of vector data from the input to the output. If off, vector data is not copied or interpolated.

`CopyTensorsOn/Off()`

These methods are used to control the copying and interpolation of tensor data from the input to the output. If off, tensor data is not copied or interpolated.

`CopyNormalsOn/Off()`

These methods are used to control the copying and interpolation of normal data from the input to the output. If off, normal data is not copied or interpolated.

`CopyTCoordsOn/Off()`

These methods are used to control the copying and interpolation of texture coordinate data from the input to the output. If off, texture coordinate data is not copied or interpolated.

`CopyAllOn/Off()`

These convenience methods set all the copy flags on or off at the same time.