VTK Data Object API

李健

VTK Data Object API

VTK使用管线架构来处理数据。

编写适配器代码需要了解管线架构。

第一类是处理数据的对象和VTK中的filters，不允许修改任何输入的数据对象。

第二类是因为VTK是可视化系统，一旦创建了数据对象，他们一般不能被修改（例如从网格删掉一个单元）。

这里的方法大多是”setter”方法，没有介绍对象的”getter”方法。可阅读VTK和ParaView手册文档。

vtkObject

几乎所有的VTK类都是由vtkObject派生的。

下面的代码片段显示了如何创建、引用和删除VTK对象：

```
vtkDoubleArray* arr = vtkDoubleArray::New(); // arr's ref count = 1
vtkPointData* pd = vtkPointData::New();        // pd's ref count = 1
pd->AddArray(arr); // arr's ref count = 2
arr->Delete();        // arr's ref count = 1
arr->SetName("an array");  // valid as arr hasn't been deleted
pd->Delete();   // deletes both pd and arr
```

To simplify the management of objects that derive from vtkObject, vtkWeakPointer, vtkSmartPointer and vtkNew can be used. These are covered in Appendix 6.1.1.

vtkDataArray

实施 vtkDoubleArray, vtkIntArray, vtkFloatArray, vtkIdTypeArray, 等

以连续的内存块存储数据，数据不是稀疏的。

使用const char* GetName() and void SetName(const char* name)方法获取和设置数据的名称。

tuple

VTK支持AOS和SOA数组结构，但更倾向AoS。

为vtkFloatArray再使用AoS的模拟内存：

void SetArray(float* array, vtkIdType size, int save)

void SetArray(float* array, vtkIdType size, int save, int deleteMethod)

vtkDataArray

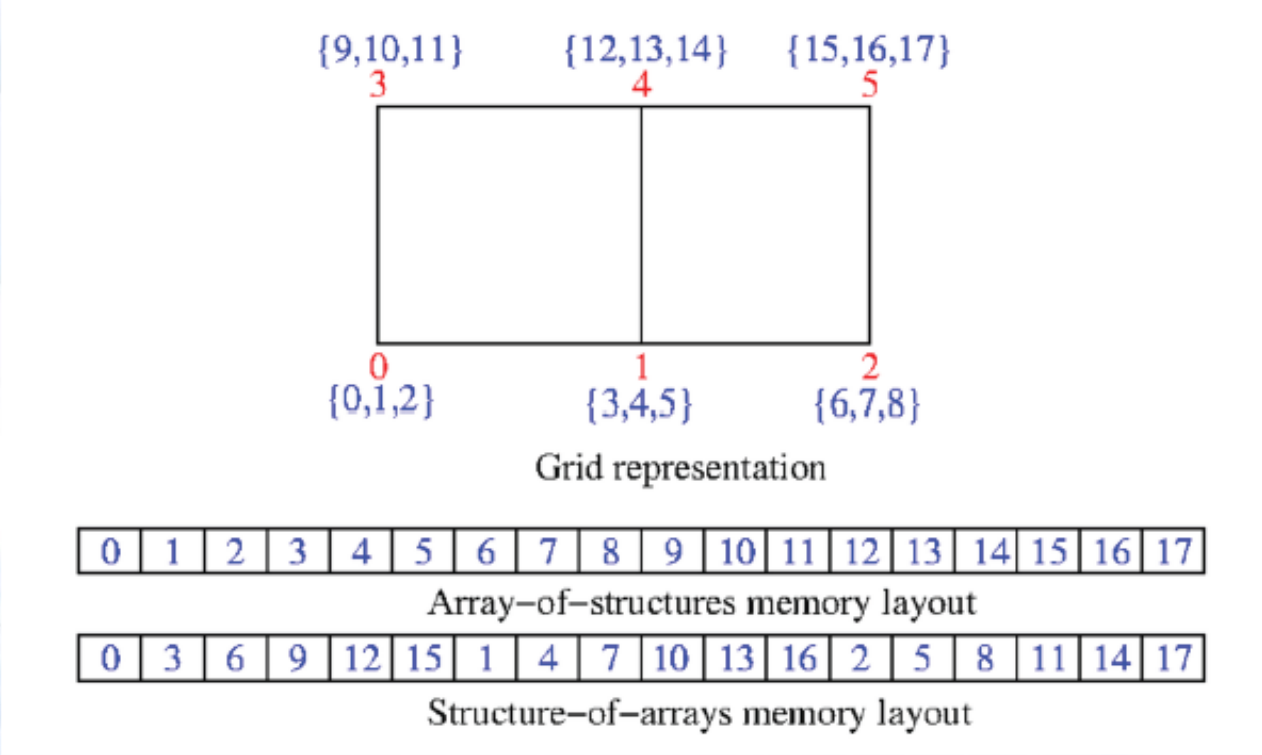


Figure 3.4: Grid representation of a vtkDataArray specified at nodes of the grid. The node index is in red and the array layout of each tuple component is shown for an AOS and SOA format in blue.

vtkDataArray

创建一个vtkFloatArray的例子：

```
vtkFloatArray* arr = vtkFloatArray::New();
arr->SetName("an array");
float* values = new float[300];
arr->SetArray(values, 300, 0, vtkDoubleArray::VTK_DATA_ARRAY_DELETE);
arr->SetNumberOfComponents(3);
```

有100个tuples和3和分量。

vtkDataArray

如果内存布置不符合VTK需要的，adaptor能分配额外的内存，来将数据传递给Catalyst。有很多设置数组大小的方式。在构建数组之前，adaptor已知数组长度。用户可调用SetNumberOfComponents(int)，然后调用SetNumberOfTuples(vtkIdType)来设置合适的长度。使用如下的方法设置数组的值，假设使用vtkFloatArray对象。

- void SetValue(vtkIdType id, float value) – set a single value at location id in the array.
- void SetTypedTuple(vtkIdType i, const float* tuple) – set all components of the i'th tuple in the array.

vtkDataArray

没有检查数组的范围大小，提高执行效率，但有风险。

```
vtkIntArray* arr = vtkIntArray::New();
arr->SetNumberOfComponents(3);
arr->SetNumberOfTuples(100);
arr->SetName("an array");
int tuple[3];
for(vtkIdType i=0;i<100;i++)
{
  tuple[0] = <value>;
  tuple[1] = <value>;
  tuple[2] = <value>;
  arr->SetTypedTuple(i, tuple);
}
```

- vtkIdType InsertNextValue(float value) – set a single value in the next location in the array and return the newly created array index.
- vtkIdType InsertNextTypedTuple(const float* tuple) – set the next tuple of values in the array and return the newly created tuple index.
- void InsertValue(vtkIdType id, float value) – set the value at location id in the array to value.
- void InsertTypedTuple(vtkIdType i, const float* tuple) – set the tuple values for the array at tuple location i.

GridTypes

vtkPolyData, vtkUnstructuredGrid, vtkStructuredGrid, vtkRectilinearGrid and vtkImageData/vtkUniformGrid

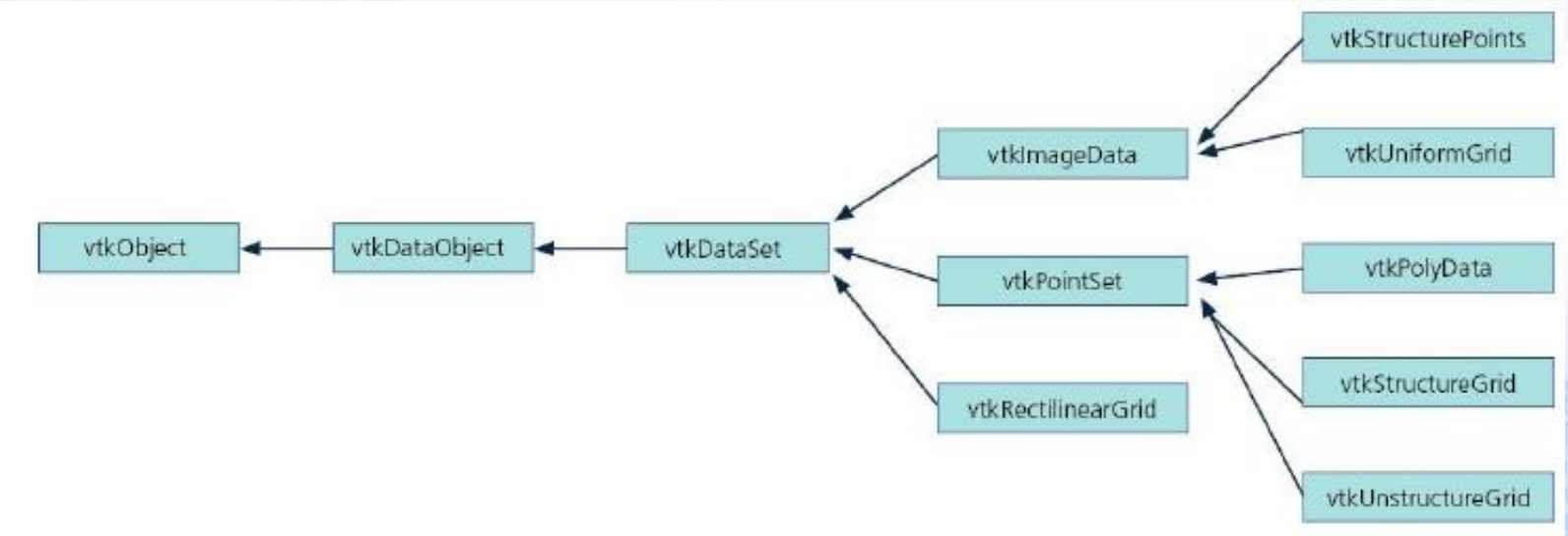
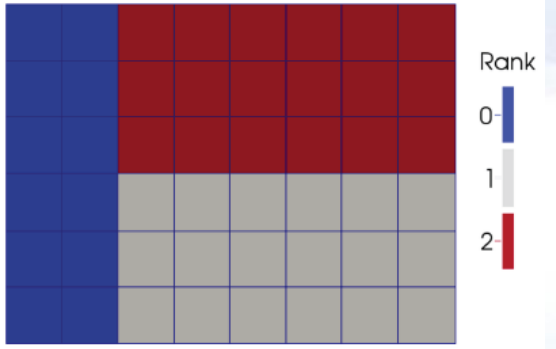


图3.5 VTK的类层级

GridTypes 结构网格

vtkImageData, vtkUniformData, vtkRectilinearGrid, vtkStructuredGrid

对每个VTK结构网格类型，用户必须设置各进程的网格范围，可使用以下2种方法之一来实现：



- void SetExtent (int x1, int x2, int y1, int y2, int z1, int z2)
- void SetExtent (int extent[6])

✓ 网格范围可使用负值

✓ 用户不能使用SetDimensions()方法，这会引起并行化分区结构网格的问题。

✓ 用户的adaptor必须调用以下vtkCPInputDataDescription类的2种方法之一来设置网格的整体范围：

- void SetWholeExtent (int x1, int x2, int y1, int y2, int z1, int z2)
- void SetWholeExtent (int extent[6])

GridTypes

结构网格

iterating over points and cells is fastest in the logical i direction, then the logical j direction, and slowest in the logical k direction. Indexing of points and cells is done independent of its whole extent but the logical coordinates are with respect to the whole extent. For example, the flat indexing and logical indexing of the points and cells are shown in Figure 3.7 below for process 2's partition in Figure 3.6.

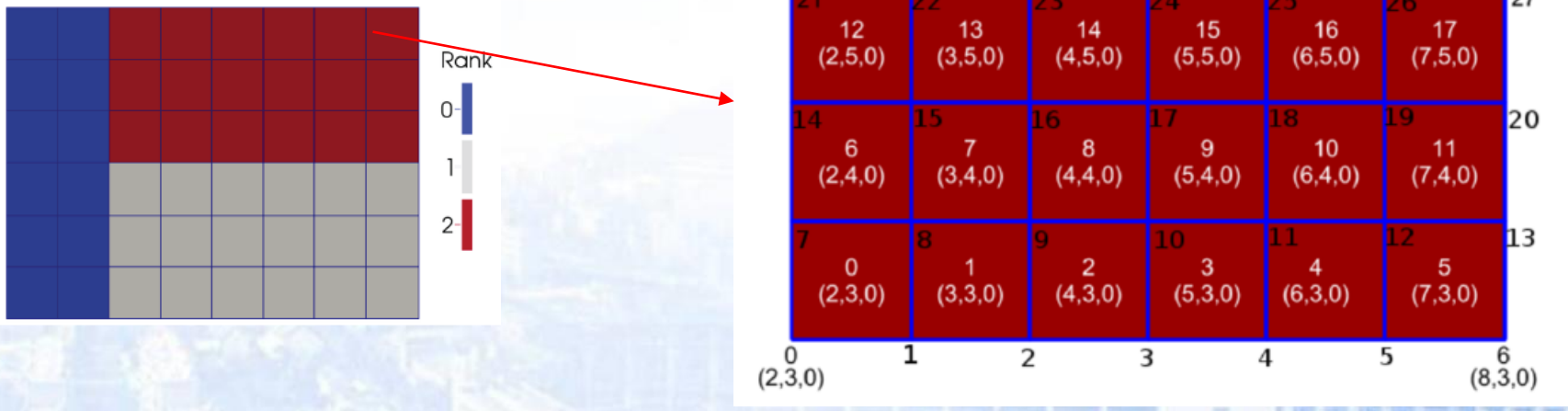


Figure 3.7: Showing the cell numbering in white and the point numbering in black for process 2's extents in the above figure. The first number is its local index and the set of numbers in parentheses are its logical global coordinates.

GridTypes

vtkImageData and vtkUniformGrid

vtkImageData and vtkUniformGrid, which derives from vtkImageData, are axis-aligned grids with constant spacing between the points. vtkUniformGrid adds in blanking to vtkImageData which can be useful for overset grid types (covered later). Beyond setting the extents of the grid, the spacing between points in each direction must be specified as well as the geometric location of logical point (0,0,0), i.e. the origin of the grid. The spacing can be set as:

- void SetSpacing(double x, double y, double z)
- void SetSpacing(double x[3])

The origin of the grid can be set as:

void SetOrigin(double x, double y, double z)
void SetOrigin(double x[3])

GridTypes

vtkImageData and vtkUniformGrid

This is for logical point coordinate (0,0,0), even if the whole extent does not contain (0,0,0). This is all that is required to set the geometry and topology of a vtkImageData object. The example below shows how to create a vtkImageData:

```
vtkImageData* grid = vtkImageData::New();
grid->SetExtent(-10, 10, -20, 20, -10, 10);
grid->SetSpacing(2.0, 1.0, 2.0);
grid->SetOrigin(100., 100., 100.);
```

In this example, the grid will have 18,081 points and 16,000 cells. The bounds of grid will be 80x,y,z120. If blanking is needed then the following vtkUniformGrid methods can be used for blanking points and/or cells:

```
void BlankPoint(vtkIdType ptId)
void BlankPoint(const int i, const int j, const int k)
void BlankCell(vtkIdType cellId)
void BlankCell(const int i, const int j, const int k)
```

GridTypes vtkRectilinearGrid

The vtkRectilinearGrid class is the next more general grid representation in VTK. It is still topologically structured but geometrically it is a semi-regular array of points. The cells are still axis-aligned but the spacing between the points in each direction is specified with a vtkDataArray. This is done with the following methods:

- void SetXCoordinates(vtkDataArray *xCoordinates)
- void SetYCoordinates(vtkDataArray *yCoordinates)
- void SetZCoordinates(vtkDataArray *zCoordinates)

GridTypes

vtkRectilinearGrid

```
vtkRectilinearGrid* grid = vtkRectilinearGrid::New();
grid->SetExtent(0, 10, 0, 20, 0, 0);
vtkFloatArray* xCoords = vtkFloatArray::New();
xCoords->SetNumberOfTuples(11);
for(vtkIdType i=0;i<11;i++)
{
  xCoords->SetValue(i, i*i);
}
vtkFloatArray* yCoords = vtkFloatArray::New();
yCoords->SetNumberOfTuples(21);
for(vtkIdType i=0;i<21;i++)
{
  yCoords->SetValue(i, i*i);
}
grid->SetXCoordinates(xCoords);
xCoords->Delete();
grid->SetYCoordinates(yCoords);
yCoords->Delete();
```

An example of how to construct a rectilinear grid is:

the grid has 231 points and 200 2D cells. The points are irregularly spaced in both the X and Y directions. Since the grid only has one layer of points in the z direction the z coordinates array does not need to be set explicitly. This results in having the grid lie in the z=0 plane.

GridTypes vtkPointSet

The remaining grids, vtkStructuredGrid, vtkPolyData and vtkUnstructuredGrid, are all geometrically irregular grids. Subsequently, they all derive from vtkPointSet which explicitly stores the point locations in a vtkPoints object which has a vtkDataArray as a data member.

The coordinates can be set with the following methods:

- void SetPoint(vtkIdType id, double x, double y, double z)
- void SetPoint(vtkIdType id, float x[3])
- void SetPoint(vtkIdType id, double x[3])

GridTypes vtkPointSet

As with vtkDataArray, there are Insert methods to add in coordinate values to vtkPoints and allocate memory as needed. They are:

- void InsertPoint (vtkIdType id, double x, double y, double z)
- void InsertPoint (vtkIdType id, const float x[3])
- void InsertPoint (vtkIdType id, const double x[3])
- vtkIdType InsertNextPoint (double x, double y, double z)
- vtkIdType InsertNextPoint (const float x[3])
- vtkIdType InsertNextPoint (const double x[3])

We reiterate the warning that using InsertPoint() improperly may lead to having uninitialized data in the array. Use void Squeeze() to reclaim unused memory. The final step is to define the vtkPoints in the vtkPointSet via the void SetPoints(vtkPoints* points) method.

GridTypes

vtkStructuredGrid

vtkStructuredGrid is still a topologically regular grid but is geometrically irregular.

An example of creating a structured grid is:

```
vtkStructuredGrid* grid = vtkStructuredGrid::New();
grid->SetExtent(0, 10, 0, 20, 0, 0);
vtkPoints* points = vtkPoints::New();
points->SetNumberOfPoints(11*21);
for(int j=0;j<21;j++)
{
   for(int i=0;i<11;i++)
   {
      points->SetPoint(i+j*11, i, j, 0);
   }
}
grid->SetPoints(points);
points->Delete();
```

vtkStructuredGrid also supports blanking. This functionality uses the same method names that are used in vtkUniformGrid which are discussed in Section 3.3.3.

GridTypes

Cell Types

GridTypes

vtkIdList

The following is some code demonstrating the use of vtkIdList:

```
vtkIdList* idList = vtkIdList::New();
idList->SetNumberOfIds(1);
idList->SetId(0, 5); // first Id is 5
idList->InsertId(1, 3); // second Id is 3
```

GridTypes

vtkPolyData

An example of creating a vtkPolyData with all quadrilateral cells is:

```
vtkPolyData* grid = vtkPolyData::New();
vtkPoints* points = vtkPoints::New();
points->SetNumberOfPoints(11*21);
for(int j=0;j<21;j++)
{
  for(int i=0;i<11;i++)
  {
      points->SetPoint(i+j*11, i, j, 0);
   }
}
grid->SetPoints(points);
points->Delete();
for(int i=0;i<10;i++)
{
   for(int j=0;j<20;j++)
   {
      vtkIdType ids[4] = {i+1+j*11, i+j*11, i+(j+1)*11, i+1+(j+1)*11};
      grid->InsertNextCell(9, 4, ids);
    }
}
```

GridTypes

vtkUnstructuredGrid

vtkUnstructuredGrid supports all VTK cell types. It also derives from vtkPointSet for storing point information. It uses a single vtkCellArray to store all of the cells. Similar to vtkPolyData, we recommend using void Allocate(vtkIdType size) to pre-allocate memory for storing cells. In this case though we recommend a value of numCells(numPointsPerCell+1) for the size. Similarly, for inserting cells, either the following methods should be used:

- vtkIdType InsertNextCell(int type, vtkIdType numPoints, vtkIdType* pts)
- vtkIdType InsertNextCell(int type, vtkIdList* pts)

These are the same as for vtkPolyData. Similarly, the example for creating points and cells for a vtkUnstructuredGrid is the same as for vtkPolyData.

GridTypes

vtkCellArray

GridTypes

vtkCellArray

An example is shown below for creating the cell topology data structures for an unstructured grid. Note that it assumes the points have already been added to the grid.

```
// create the cell data structures
vtkCellArray* cellArray = vtkCellArray::New();
vtkIdTypeArray* offsets = vtkIdTypeArray::New();
vtkUnsignedCharArray* types = vtkUnsignedCharArray::New();
vtkIdType ids[8];
// create a triangle
ids[0] = 0; ids[1] = 1; ids[2] = 2;
cellArray->InsertNextCell(3, ids);
offsets->InsertNextValue(0);
types->InsertNextValue(VTK_TRIANGLE);
// create a quad
ids[0] = 0; ids[1] = 1; ids[2] = 2; ids[3] = 3;
cellArray->InsertNextCell(4, ids);
offsets->InsertNextValue(4);
types->InsertNextValue(VTK_QUAD);
```

GridTypes

vtkCellArray

```
// create a tet
ids[0] = 0; ids[1] = 1; ids[2] = 2; ids[3] = 4;
cellArray->InsertNextCell(4, ids);
offsets->InsertNextValue(9);
types->InsertNextValue(VTK_TETRA);
// create a hex
ids[0] = 0; ids[1] = 1; ids[2] = 2; ids[3] = 3;
ids[4] = 4; ids[5] = 5; ids[6] = 6; ids[7] = 7;
cellArray->InsertNextCell(8, ids);
offsets->InsertNextValue(14);
types->InsertNextValue(VTK_HEXAHEDRON);
// add the cell data to the unstructured grid
grid->SetCells(types, offsets, cellArray);
types->Delete();
offsets->Delete();
cellArray->Delete();
```

GridTypes

Field Data

The following snippet of code demonstrates how arrays are added to point data and cell data.

```
vtkDoubleArray* pressure = vtkDoubleArray::New();
pressure->SetNumberOfTuples(grid->GetNumberOfPoints());
pressure->SetName("pressure");
grid->GetPointData()->AddArray(pressure);
pressure->Delete();
vtkFloatArray* temperature = vtkFloatArray::New();
temperature->SetName("temperature");
temperature->SetNumberOfTuples(grid->GetNumberOfCells());
grid->GetCellData()->AddArray(temperature);
temperature->Delete();
```

Multi-Block Datasets

有一些情况，需要使用多个datasets来表征模拟的数据结构。例如：overlapping grids (AMR)，或当一个单独的数据集类型不适合存储单元拓扑（例如使用a vtkUniformGrid 和 a vtkUnstructuredGrid）。

主要的类是vtkCompositeDataSet，这是一个抽象类，目的是简化存储在不同衍生类中的vtkDataSets的循环方式。

有2类符合数据集类型：

（1） AMR类型网格，仅使用vtkUniformGrid数据集，离散计算域。vtkOverlappingAMR and vtkNonOverlappingAMR

（2）复合数据集类型，支持所有衍生自vtkDataSet的网格，但需要blanking needed for overlapping grids，显式管理。

Multi-Block Datasets

2 个 类 ： vtkMultiBlockDataSet 和 vtkMultiPieceDataSet ， 都 衍 生 自 vtkDataObjectTree

多块数据集层级见图3.12。

Because vtkCompositeDataSet derives from vtkDataObject it has a vtkFieldData object that can be accessed by the vtkFieldData* GetFieldData() method. This can be useful for storing meta-data.

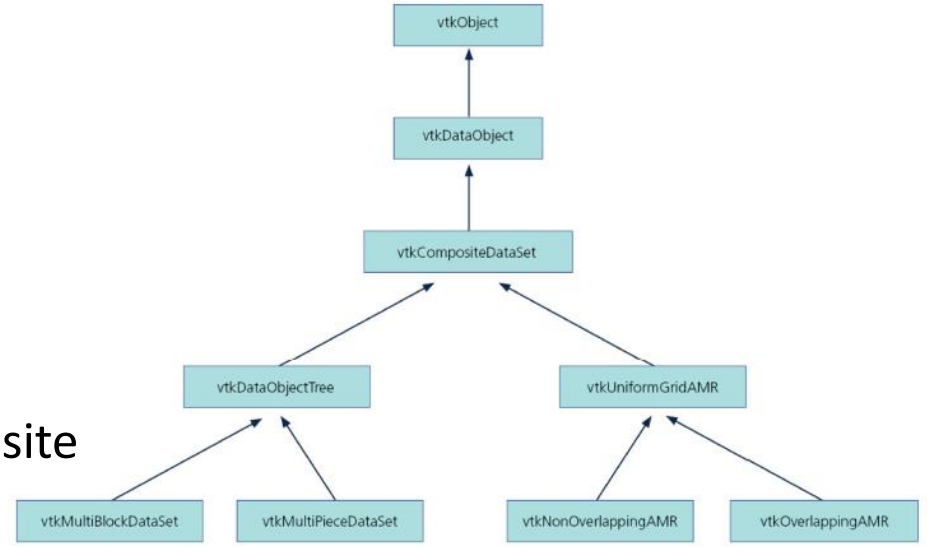


Figure 3.12: Class hierarchy for VTK composite datasets

Multi-Block Datasets

vtkMultiBlockDataSet

Multi-Block Datasets

vtkMultiPieceDataSet

Multi-Block Datasets

vtkUniformGridAMR

Multi-Block Datasets

vtkOverlappingAMR

Multi-Block Datasets

vtkNonOverlappingAMR