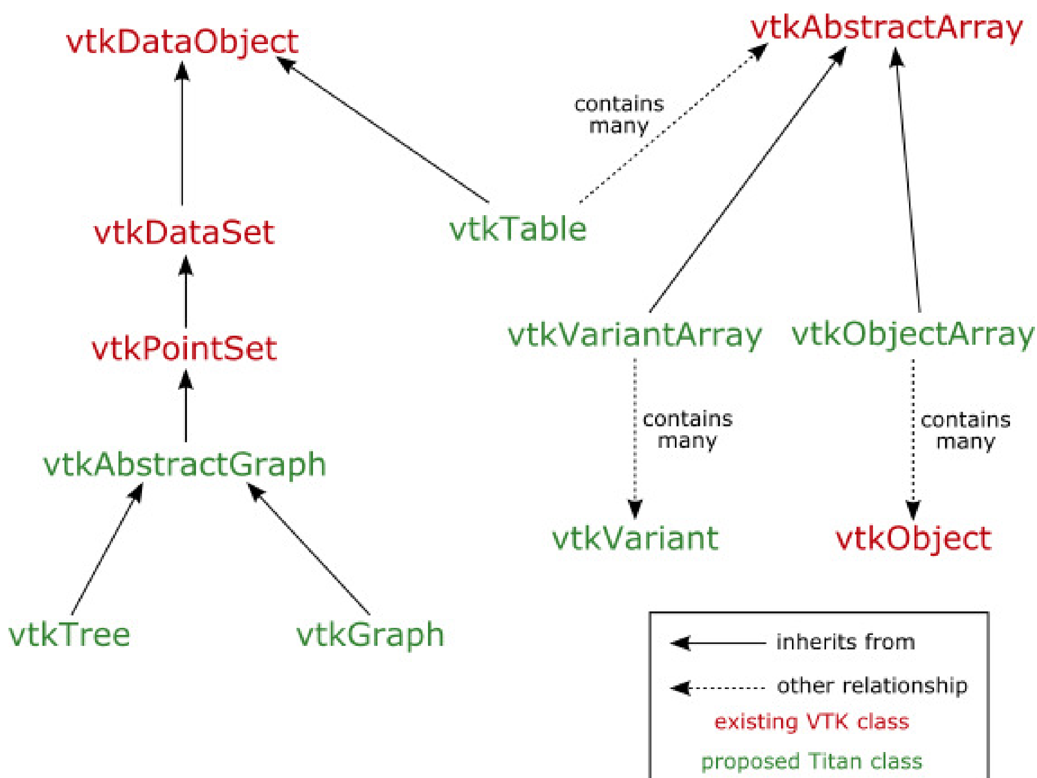
Titan Data Structures

From ParaQ Wiki

The data structures for Titan Infovis Toolkit and Applications iNfrastructure (Titan). These structures must be able to hold all types of data required for information visualization.

Overview

The following diagram illustrates the current version of the data structures.



vtkVariant

A variant represents a value for a property of an entity. An invalid value essentially means that the value is NULL (i.e. it is just a placeholder).

```
class vtkVariant
{
 vtkVariant();
 ~vtkVariant();
 vtkVariant(const vtkVariant & other);

 vtkVariant(vtkStdString value);
 vtkVariant(float value);
 vtkVariant(double value);
 vtkVariant(int value);
 vtkVariant(__int64 value);
 vtkVariant(vtkObject* value);

 const vtkVariant & operator= (const vtkVariant & other);
```

```
 bool IsValid() const;
 bool IsString() const;
 bool IsNumeric() const;
 bool IsFloat() const;
 bool IsDouble() const;
 bool IsInt() const;
 bool Is__Int64() const;
 bool IsObject() const;
 bool IsArray() const;

 unsigned int GetType() const;
 const char* GetTypeAsString() const;

 vtkStdString ToString() const;
 float ToFloat() const;
 double ToDouble() const;
 int ToInt() const;
 __int64 To__Int64() const;
 vtkObject* ToObject() const;
 vtkAbstractArray* ToArray() const;
};
```

It is imperative that vtkVariant have a minimal memory footprint (since we will be storing large arrays of them). The current implementation of vtkVariant contains a 64-bit union for the data and 16-bits worth of flags holding state (i.e. the valid state and the data type). To ensure this footprint, vtkVariant has no superclass and, more importantly, *no* virtual functions.

The behavior of vtkVariant changes slightly based on the type of data it holds. The To* method will silently do an appropriate conversion (or at least do something logical if a conversion is not possible). If the vtkVariant holds a vtkObject or an array type, those objects are referenced and dereferenced as appropriate (much like a vtkSmartPointer).

vtkVariantArray

This class is an array of #vtkVariant objects. The structure and functionality of this class is very similar in nature to vtkStringArray.

```
class vtkVariantArray : public vtkAbstractArray
{
  //
  // Implement abstract functions from vtkAbstractArray
  //

  // Additional functions:

  #vtkVariant & GetValue(vtkIdType id) const;
  void SetValue(vtkIdType id, #vtkVariant value);
  void InsertValue(vtkIdType id, #vtkVariant value);
  vtkIdType InsertNextValue(#vtkVariant value);
  #vtkVariant* GetPointer(vtkIdType id);
  void SetArray(#vtkVariant* arr, vtkIdType size, int save);
  void SetNumberOfValues(vtkIdType number);
};
```

vtkObjectArray

This array type contians an array of vtkObject objects. The structure of this class mimics #vtkVariantArray. This would allow arrays of any VTK object type.

vtkTable

A table is a two-dimensional array of information where each column represents a different property. A row in the

table may represent a single entity. The functions internally modify the vtkFieldData member field inherited from vtkDataObject. Using these methods only additionally ensures that all arrays are the same length. The GetValue() and GetRow() functions return the appropriate vtkVariants in a vtkVariantArray, or wrap values from other array types into #vtkVariants. Similarly, SetValue(), InsertRow(), and InsertNextRow() convert the #vtkVariant to the appropriate type for the column.

If a column array has more than one component per tuple, the value is set by passing a variant holding an array with one tuple with the same number of components.

```
class vtkTable : public vtkDataObject
{
  int GetNumberOfRows() const;
  #vtkVariantArray* GetRow(int row) const;
  int InsertRow(vtkIdType row);
  int InsertRow(vtkIdType row, #vtkVariantArray* values);
  int InsertNextRow();
  int InsertNextRow(#vtkVariantArray* values);
  void RemoveRow(vtkIdType row);

  int GetNumberOfColumns() const;
  const char* GetColumnName(int col) const;
  vtkAbstractArray* GetColumn(const char* name) const;
  vtkAbstractArray* GetColumn(int col) const;
  void AddColumn(vtkAbstractArray* arr);
  void RemoveColumn(const char* name);
  void RemoveColumn(int col);

  #vtkVariant GetValue(vtkIdType row, int col) const;
  #vtkVariant GetValue(vtkIdType row, const char* col) const;
  void SetValue(vtkIdType row, int col, #vtkVariant value);
  void SetValue(vtkIdType row, const char* col, #vtkVariant value);
};
```

vtkAbstractGraph

A graph consists of edges and nodes, which may have any number of associated properties (stored in vtkFieldData objects). An abstract graph contains functionality common to both #vtkGraph and #vtkTree. These are all read-only methods, so that methods such as `AddEdge` are not available to #vtkTree. This allows #vtkTree to enforce structural rules.

vtkAbstractGraph inherits from vtkPointSet since, when a graph is embedded, it will have coordinates for its nodes. However, before being embedded, a graph should not have to store a vtkPoints array. Thus, graphs store a NULL vtkPoints structure and GetPoint() will return (0,0,0). If GetPoints() is called, the graph will generate a new vtkPoints array with the appropriate size with all coordinates initialized to (0,0,0) and return this new structure.

```
class vtkAbstractGraph : public vtkPointSet
{
 //
 // Abstract functions required by vtkDataSet
 //
 vtkIdType GetNumberOfCells();                          // Calls GetNumberOfEdges
 void GetPointCells(vtkIdType ptId, vtkIdList* cellIds);  // Calls GetNodeEdges
 int GetMaxCellSize();                                  // Returns 2
 int GetCellType(vtkIdType cellId);                     // Returns VTK_LINE
 vtkCell* GetCell(vtkIdType cellId);                    // Returns a vtkLine for the edge
 void GetCell(vtkIdType cellId, vtkGenericCell * cell);   // Makes a generic line for the edge
 void GetCellPoints(vtkIdType cellId, vtkIdList* ptIds);  // Uses GetSourceNode, GetTargetNode

 //
 // Additional functions
 //
 vtkPointData* GetNodeData();                           // Returns PointData
 vtkCellData* GetEdgeData();                            // Returns CellData
 vtkIdType GetNumberOfNodes() const;
 vtkIdType GetNumberOfEdges() const;
 void GetNodeEdges(vtkIdType node, vtkIdList* edges);
```

```
 void GetNodeInEdges(vtkIdType node, vtkIdList* edges);
 void GetNodeOutEdges(vtkIdType node, vtkIdList* edges);
 vtkIdType GetSourceNode(vtkIdType edge) const;
 vtkIdType GetTargetNode(vtkIdType edge) const;
 vtkIdType GetOppositeNode(vtkIdType edge, vtkIdType node) const;

 //
 // Abstract functions
 //
 virtual bool IsDirected() const = 0;
};
```

vtkGraph

The data structure should be designed so that the following operations may be performed quickly:

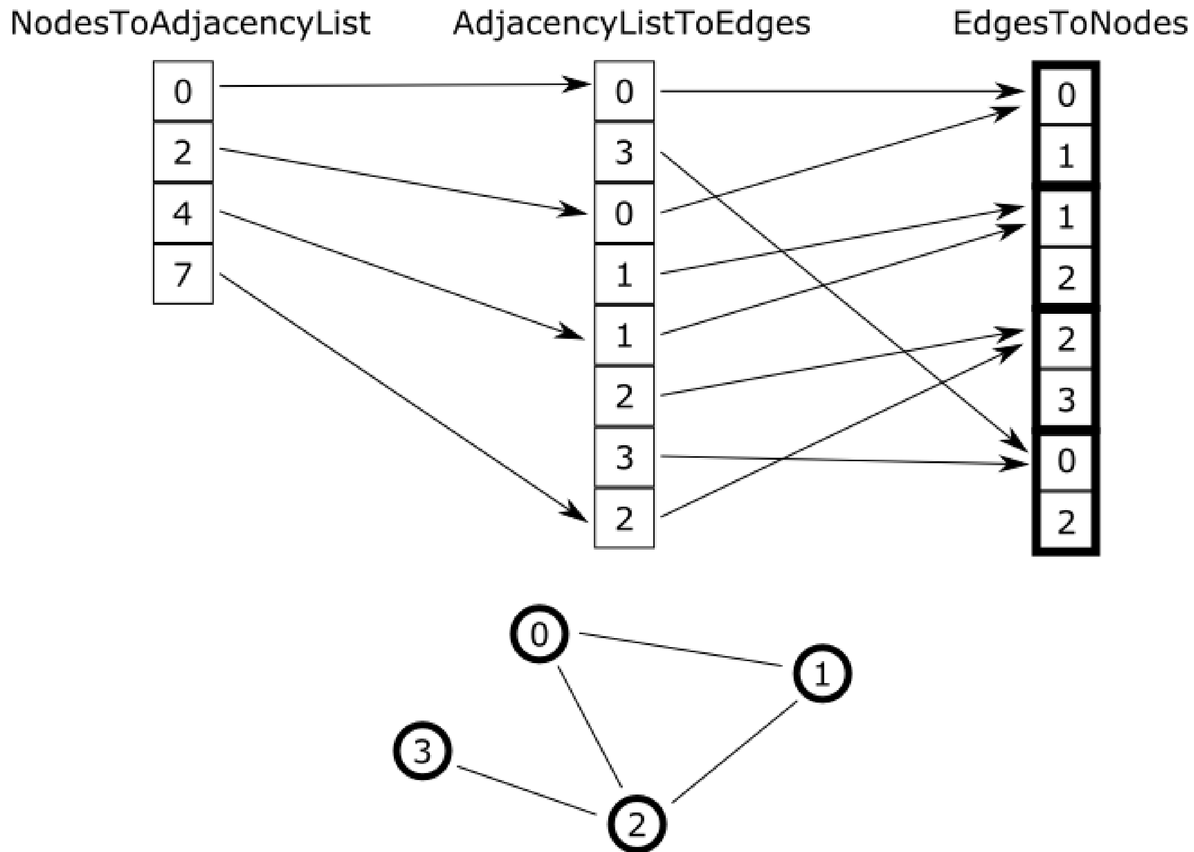
- Given an edge, retrieve its endpoint nodes.
- Given a node, retrieve the edges adjacent to it.

```
class vtkGraph : public #vtkAbstractGraph
{
 //
 // Functions required by vtkAbstractGraph
 //
 bool IsDirected() const;

 //
 // Additional functions
 //
 void SetDirected(bool directed);
 vtkIdType AddNode();
 vtkIdType AddEdge(vtkIdType source, vtkIdType target);
 void RemoveNode(vtkIdType node);
 void RemoveEdge(vtkIdType edge);
};
```

The graph is by default stored as a simple list of edges, implemented by a vtkIdTypeArray EdgesToNodes which has 2 components per tuple indicating the source and target node IDs. Modifying the graph will simply modify the edge list, and set a flag indicating that the adjacency list is invalid. Whenever the functions GetNodeEdges, GetNodeInEdges, or GetNodeOutEdges is called, the graph checks if the adjacency arrays are invalid. If so, it rebuilds the adjacency arrays, which allows fast lookup of edges adjacent to a node, and clears the invalid flag.

The first adjacency array, NodesToAdjacencyList, has NumberOfNodes entries. The values in this array are indicies in the second array where the adjacency list for that vertex begins. The indices in the array are monitonically nondecreasing. The second array, AdjacencyListToEdges, is a compact adjacency list representation of the graph. The entries AdjacencyListToEdges[NodesToAdjacencyList[nodeID]] to AdjacencyListToEdges[NodesToAdjacencyList[nodeID + 1] - 1] contain the IDs of the edges adjacent to the node with ID nodeID. The image below shows an example of a graph and its representation.



vtkTree

A tree is a graph which has no cycles and may contain a root. SetParent detaches a subtree from its current parent and attaches it under a new parent. SetRoot changes the root node by reversing the edges in the path from the new root to the old root.

```
class vtkTree : public #vtkAbstractGraph
{
 //
 // Functions required by #vtkAbstractGraph
 //
 bool IsDirected() const { return true; }

 //
 // Additional functions
 //
 vtkIdType GetRoot() const;
 void GetChildren(vtkIdType parent, vtkIdList* children);
 vtkIdType GetParent(vtkIdType child);
 vtkIdType AddRoot();
 vtkIdType AddChild(vtkIdType parent);
 void RemoveNodeAndDescendants(vtkIdType node);
 void SetParent(vtkIdType child, vtkIdType parent);
 void SetRoot(vtkIdType root);
};
```

vtkDocument

This is a proposed data type to add sometime in the future. A document is a significant portion of text that may be split into words, sentences, paragraphs, sections, etc. The document may be in plain text or rich text format, and be in any language. Text processing algorithms should be able to easily manipulate the document in meaningful ways.