University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering
Centre of Computer Graphics and Data Visualization

MVE - 2

Visualization library

Miroslav Vavruška, Milan Frank
23. 5. 2005 (Version: alfa-4)

Contents

1. Introduction

Together with *MVE-2* development started development of *Visualization* library that would be certain standard and common base for developers and researchers in the area of computer graphics, data visualization and image processing. This document serves to demonstrate principles and design of Visualization library.

It is hard to design and implement a system of data structures for computer graphics and data visualization from scratch. Therefore we were inspired by the data model used in *VTK* (Visualization ToolKit developed by Kitware Inc.). The MVE-2 structures are more general and more “object-like”. Many differences arises from C#/.NET and C++ environments.

2. Visualization data model

Beside algorithms that pose functionality, perform no less important role also several data. Visualization environments arose as reply to demand for processing of huge data volume, that wasn't possible efficient process, analyse and represent. Orbital space stations producing terabytes of data every day, laser scan systems generating 500.000 of points every 15 seconds, super computers forecasting weather for whole planet, New York's stock exchange working with 333 million transactions every day – this all are areas, that couldn't exist without visualization. Now is very important to design efficient data structures for data storage and access.

Data structures should be:

- *compact* – visualization data tends to be large, so we need compact storage schemes to minimize computer memory requirements.
- *efficient* – data must be easily and quickly accessible, we want to retrieve and store data in constant time (i.e. independent of data size).
- *simple* – simplicity makes structures easier to understand and use, and therefore, optimal.

2.1. Datasets

Data objects in the visualization system are called d*atasets*. The dataset is an abstract form, consists of two pieces: an organizing *structure* and supplemental data *attributes* associated with the structure.

Structure has certain *topology* and *geometry*. Topology is the set of properties invariant under certain geometric transformations (rotation, translation, and nonuniform scaling). Geometry is the specification of position in 3D space. The structure of dataset consists of *cells* and *points*. The cells specify the topology, while the points specify the geometry.

The *attributes* are supplemental informations associated with the structure, therefore with points or cells. Typical attributes include scalars, vectors, normals, texture coordinates, tensors, and user-defined data.

2.2. Cells

Cells are the fundamental building blocks of visualization systems. A dataset consists of one or more cells. Cells are defined by specifying a type in combination with an ordered list of points. The ordered list, often referred to as the *connectivity list*, combined with the type specification, implicitly defines the topology of the cell. The x-y-z point coordinates define the cell geometry. Cells can be *primary* or *composite*. Composite cells consist of one or more primary cells, while primary cells cannot be decomposed into combination of other primary cell types.

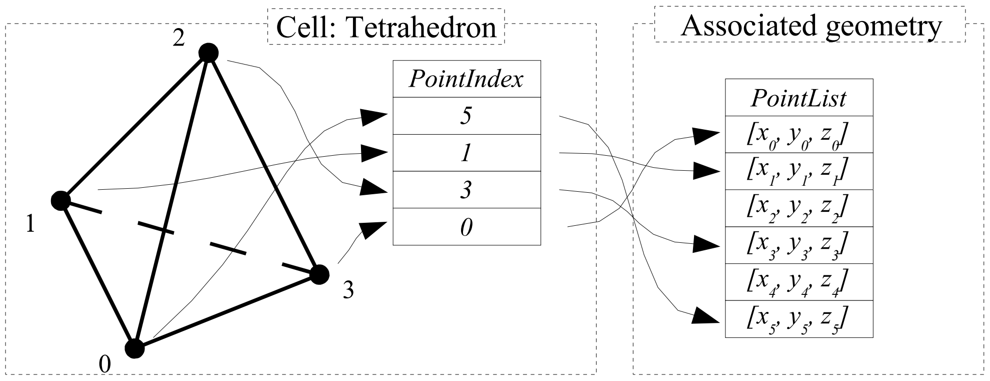


Figure 1: Topological element tetrahedron with associated geometry.

Tetrahedron is defined by the ordered list of 4 points. The topology of this cell is implicitly known: we know that (1,2) is one of the 6 edges of the tetrahedron and (1,2,3) is one of its 4 faces.

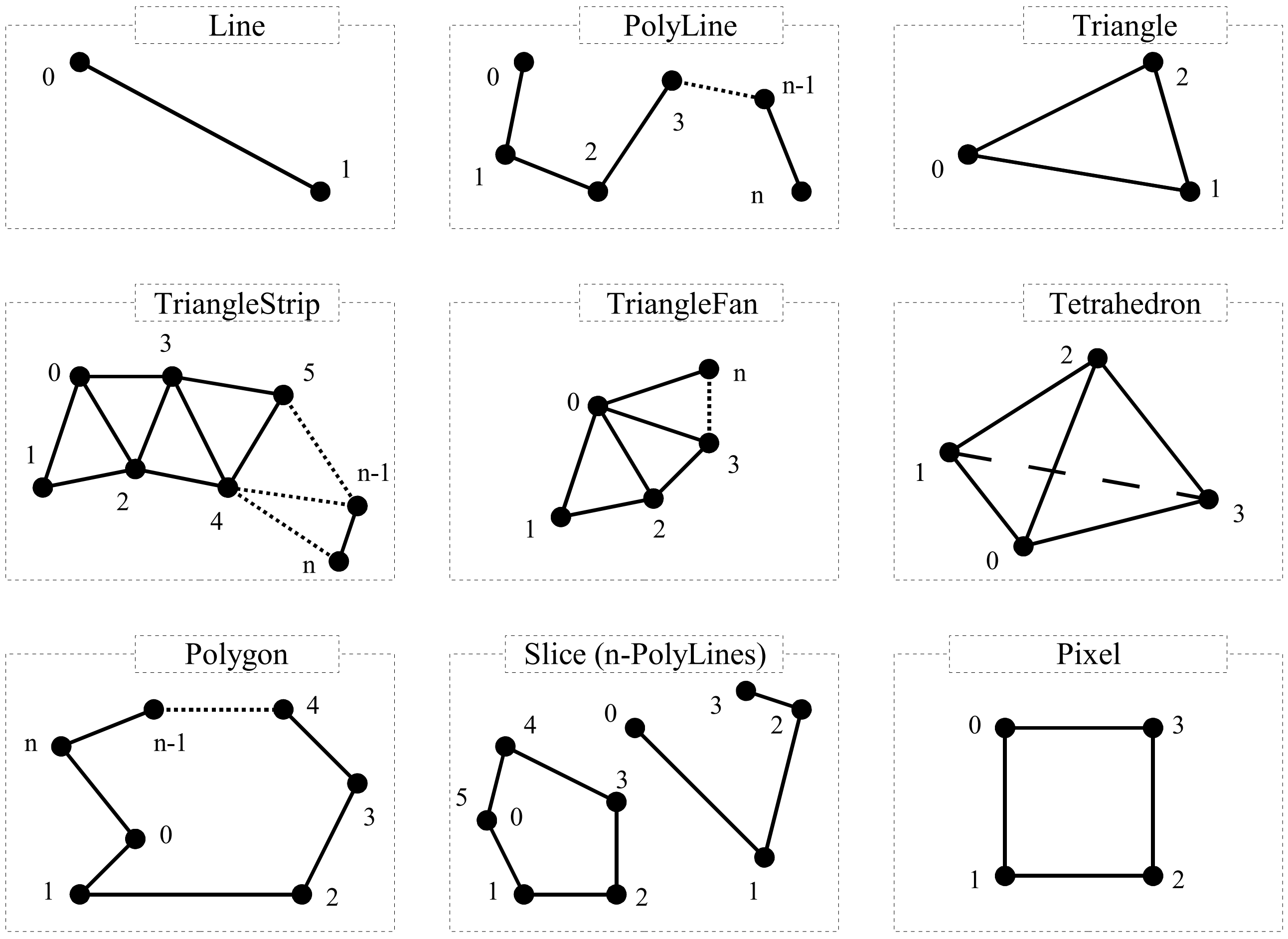
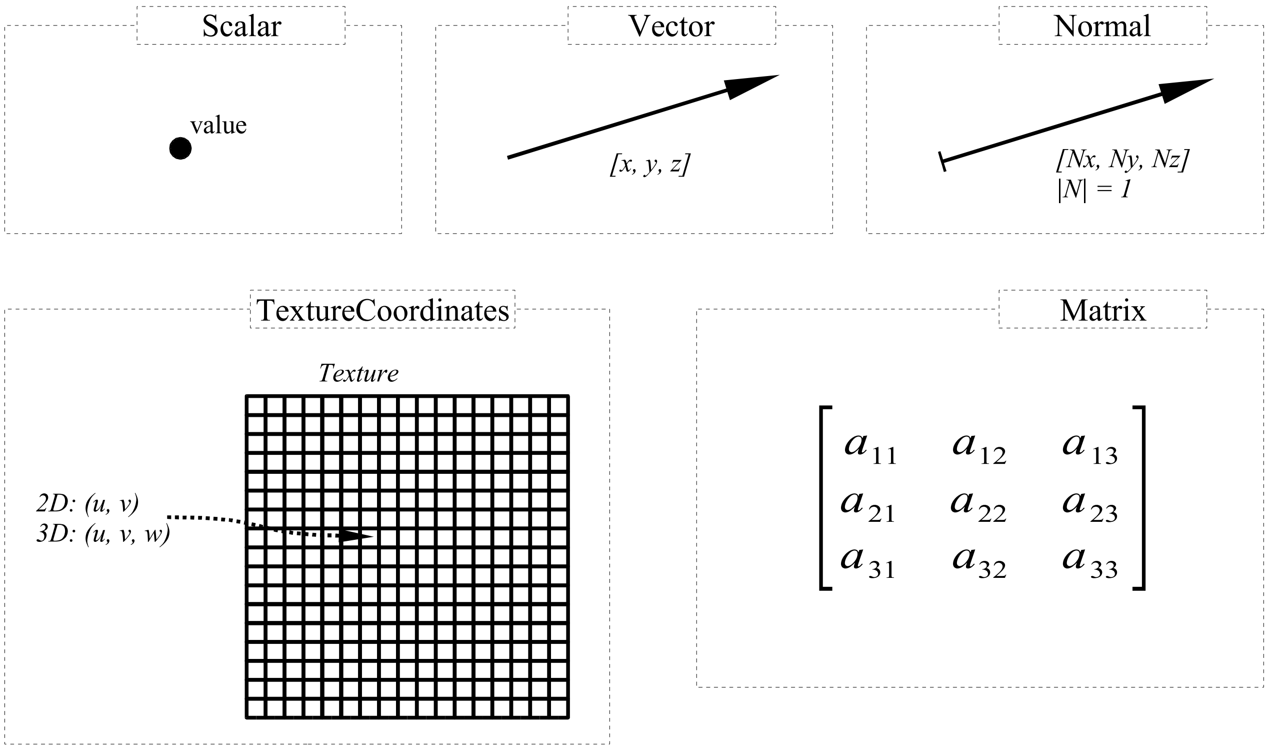


Figure 2: Cell types.

Besides these types, many other potential cell types exist. There is a basic set of most often used cells. They were chosen for their qualities and practical using in many areas.

2.3. Attributes

Attribute data is an information associated with the structure of the dataset. Typical attribute include temperature or velocity at a point, mass of a cell, or heat flux into and out of a cell face.



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Figure 3: Attribute types.

There is a basic set of most often used attributes. They were chosen for their qualities and practical using in many areas.

2.4. Types of Datasets

A dataset consists of an organizing *structure* plus associated *attribute* data. The structure has both *topological* and *geometric* properties and is composed of one or more points and cells. A dataset is characterized according to whether its structure is *regular* or *irregular*.

A dataset is *regular* if there is a single mathematical relationship within the composing points and cells. If the points are regular, then the geometry of the dataset is regular. If the topological relationship of cells is regular, then the topology of the dataset is regular. Regular (or structured) data can be implicitly represented, at great savings in memory and computation.

Irregular (or unstructured) data must be explicitly represented, since there is no inherent pattern that can be compactly described. Unstructured data tends to be more general, but requires greater memory and computational resources. We typically use unstructured datasets to storage data only when absolutely necessary.

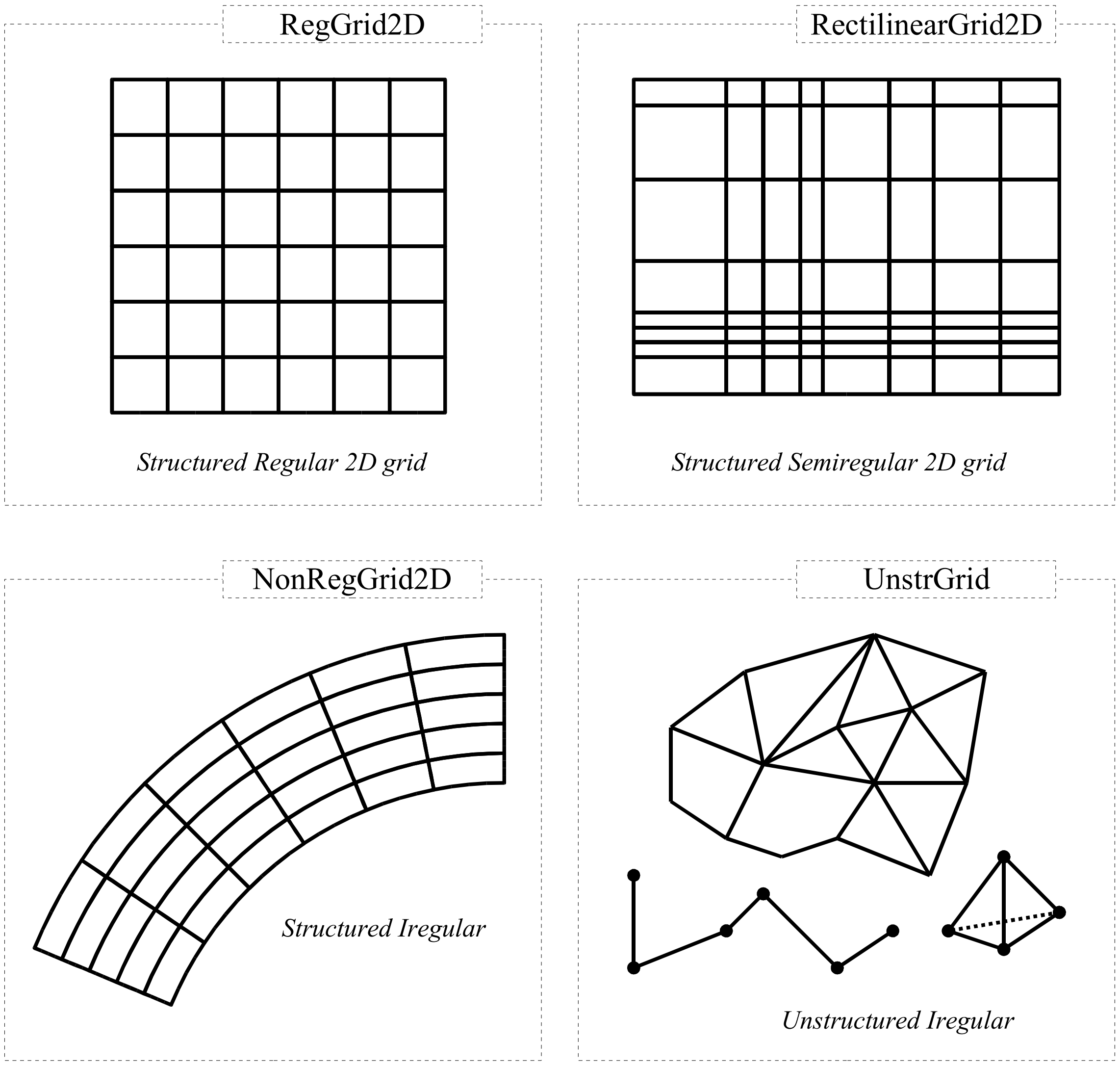


Figure 4: Dataset types.

3. Visualization library

The *Visualization* library is a component of MVE-2 system. In this library there are implemented basic data types such as points, cells, attributes and datasets. This chapter describes principles and design of Visualization library, data structures summary, data types conversions, XML representation and finally creating of user data structures.

3.1. Implemented data structures

Extensive recognition should proceed introduction of a new data type. Some useful data structure may already exist. Existence of duplicity in data structures is very contra productive and result into incompatibility of modules. In this chapter we describe basic set of data types, that have been already implemented.

3.1.1. Basic data types

The following picture shows the hierarchy of Visualization *basic data types*. Each of them implements *IdataObject* interface. Therefore it is possible to send these basic data types between modules. Because processing of objects includes relatively big overhead, all of these data types are implemented as structures. The data types hierarchy is attainment by inheritance of several interfaces. Structure implements corresponding interface according to into which data type category it belongs. For example all points implement *IPoint* interface, cells implement *ICell* etc.

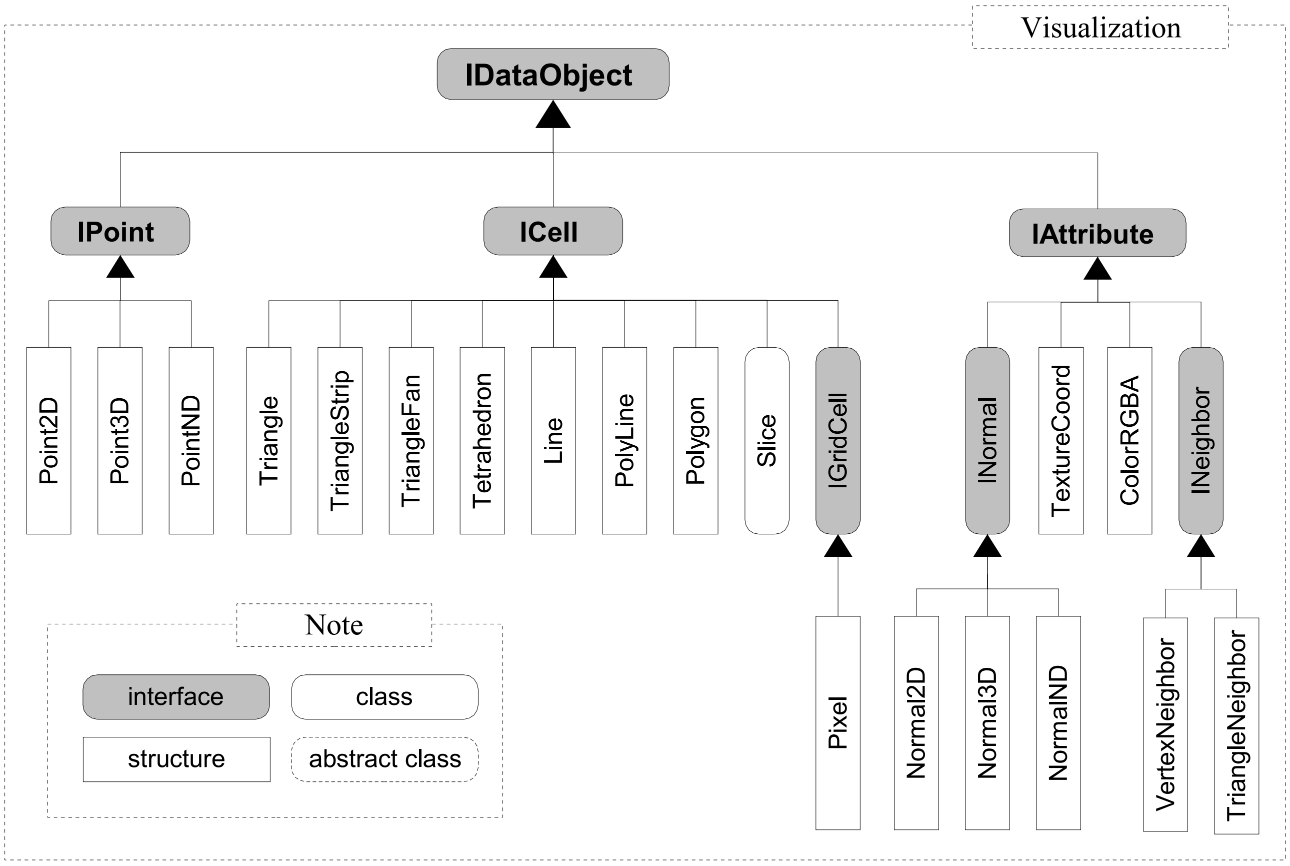


Figure 5: Diagram of Visualization library basic data types.

Note: there are implemented another important, generally used and helpful data types in the *Numerics* library.

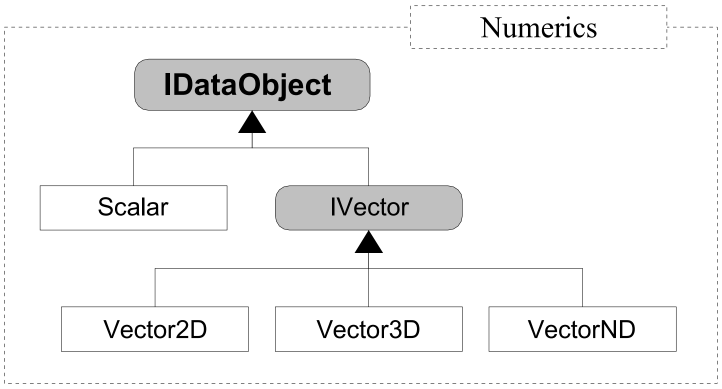


Figure 6: Diagram of Numerics library data types.

3.1.2. Data type conversions

Sometimes we need the *type casting* of one to another data type. Then we use data type conversions. If there is no data loss (for example *Point2D → Point3D)*, it is used implicit conversion and there's not necessary to indicate name of type whereon we overtype. Reversely, if there is some data loss (for example *Point3D → Point2D)*, it is used explicit conversion and we must launch the name of target type. Explicit conversion is also used for some specific cases (*Vector → Normal)* when user must be aware of conversion reason.

Implemented conversions:

- Point2D → Point3D → PointND – *implicit*
- Point2D ← Point3D ← PointND – *explicit*
- Vector2D → Vector3D → VectorND – *implicit*
- Vector2D ← Vector3D ← VectorND – *explicit*
- Normal2D → Normal3D → NormalND – *implicit*
- Normal2D ← Normal3D ← NormalND – *explicit*
- Point ↔ Vector – *implicit*
- Vector → Normal – *explicit*
- Normal → Vector – *implicit*
- Scalar ↔ double – *implicit*
- ColorRGBA ↔ System.Drawing.Color – *implicit*

3.1.3. Support classes

The following picture shows the hierarchy of Visualization *support classes*.

- Transform3D – general transformation of 3D object
- UniformDataArray – one-dimensional homogenous static array of data objects
- UniformDataArray2D – two-dimensional homogenous static array of data objects

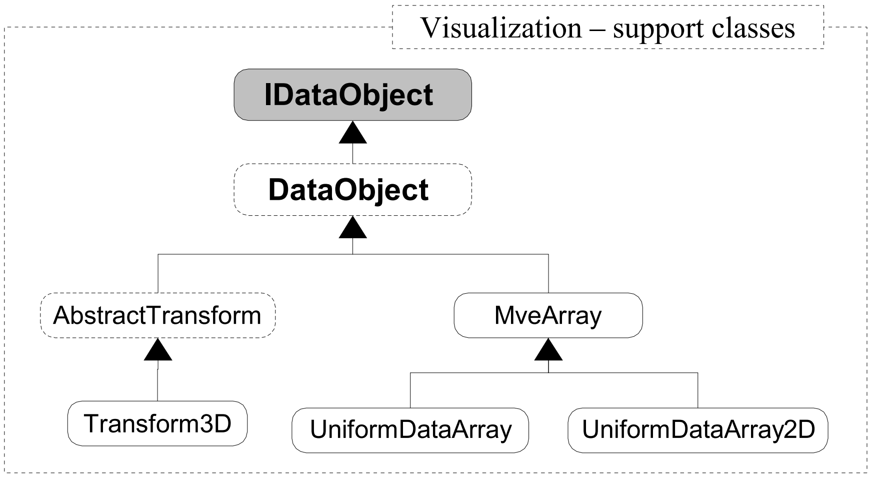


Figure 7: Diagram of Visualization library support classes.

3.1.4. Datasets

Following data structures represent variety of logical ordering of basic data types. Each element can have any attributes. The attribute is with the point or cell associated by the index.

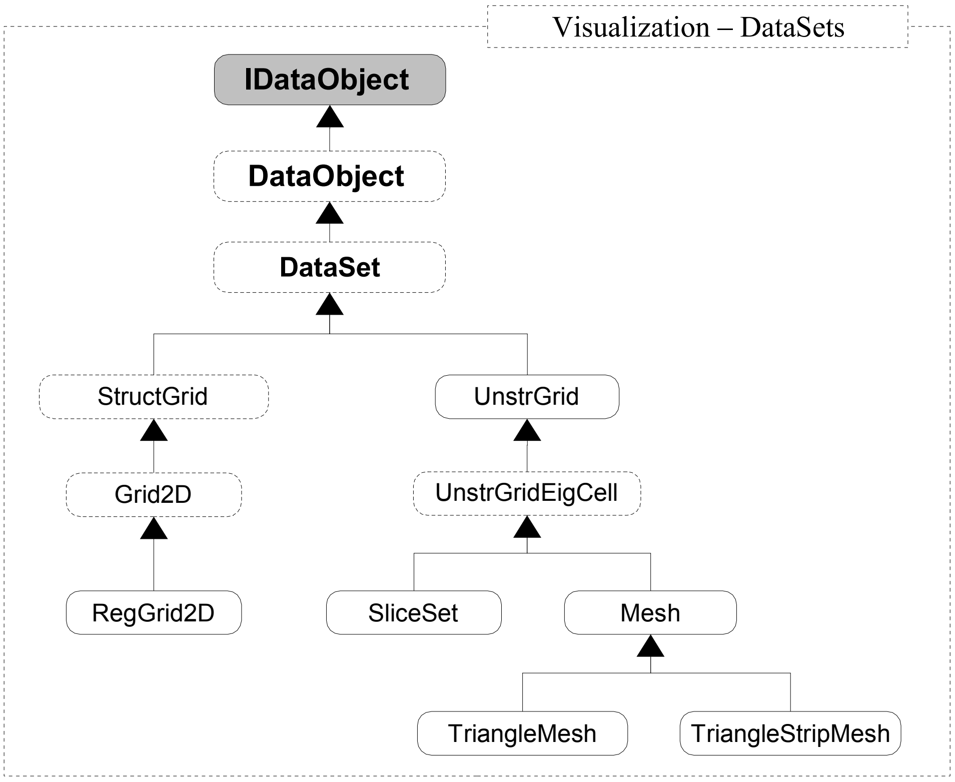


Figure 8: DataSets of Visualization library.

3.1.5. XML data structures serialization

MVE-2 directly supports XML data structures serialization. The format XML was chosen for its high informative content, easy readibility and enlargement. Because XML is the text format and use tags for data separation, it tends to be large. MVE-2 expects XML data serialization for illustration and readibility, not for storage of large data volume. By the reason of memory requirements decrease we work with XML files sequential.

Modules for XML serialization are implemented in the MVE-2 core. For saving data into XML is used *XmlSaver* module, for data reading is used *XmlLoader*. Each of the data structure must implement *IDataObject* interface. There's declared *WriteData()* abstract method for data saving and *ReadData()* method for data reading. These methods must be implemented by the author of specific data structure.

The following example shows storage of *Point3D* data type. The node *libraries* includes the list of libraries, that are necessary for data dype creation. In the event of basic data types this node containts the single name of library. The attribute *type* of node *data* is corresponding to data type name and the node *data* containts actual data. Modules *XmlSaver* and *XmlLoader* manage the file header, nodes *dataobject*, *libraries* and the node *data* with its attribute *type*. The content of *data* node is fully under control of author of specific data structure.

```
<?xml version="1.0" encoding="windows-1250"?>
<dataobject>
    <libraries count="1">
        <lib>Visualization.dll</lib>
    </libraries>
    <data type="Point3D">1.5 -2.8 1.2</data>
</dataobject>
```

3.2. Using of data structures

This chapter describes priciples of working with implemented data structures.

3.2.1. Basic data types

Using of basic data structures is very simple. The *ToString()* method is overriden for all data structures. Therefore we can simply get useful information about it. Data structures which contain array or look like array, have implemented access through indexer. The following example illustrate these situations.

```
Triangle tri = new Triangle(0, 2, 4);  // creation of structure
Console.WriteLine(tri);                // print: Triangle: [0, 2, 4]
Console.WriteLine(tri[1]);             // print: 2
tri[1] = 3;                            // use of indexer
Console.WriteLine(tri);                // print: Triangle: [0, 3, 4]
```

In cases where is it suitable and possible, there is the conversion implemented. The following example describes implicit conversion *Point2D → Point3D*:

```
Point2D p2 = new Point2D(1.5, 2.3);     // creation of structure
Console.WriteLine(p2);                  // print: Point2D: [1.5; 2.3]
Point3D p3 = p2;                        // implicit conversion
Console.WriteLine(p3);                  // print: Point3D: [1.5; 2.3; 0]
```

The next example shows the using of explicit conversion *Point3D → Point2D*:

```
Point3D p3 = new Point3D(1.5, 2.3, 3.8); // creation of structure
Console.WriteLine(p3);                   // print: Point3D: [1.5; 2.3; 3.8]
Point2D p2 = (Point2D) p3;               // explicit conversion
Console.WriteLine(p2);                   // print: Point2D: [1.5; 2.3]
```

3.2.2. DataSets

RegGrid2D

Description: this dataset represents a structured regular rectangular grid, that consists of pixels. Each pixel is created by four points. We access to points and pixels through indexes. With each point or pixel can be associated a number of attributes. The grid is given by resolution (width, height), which is the count of pixels in the x- and y-axis direction. The number of grid pixels is then width x height, while the number of points is (width + 1) x (height + 1). This dataset can be used for example for picture representation (bitmap).

Containts:

- points and pixels - logically only. Because the grid is structured, the points and pixels can be implicitly determinated and needn't to be physicaly stored in the memory.
- pixels and points attributes – as uniform homogenous two-dimensional array

Summary of basic methods:

- AddCellAttrs(), GetCellAttrs() – operations with cell attributes
- AddPointAttrs(), GetPointAttrs() – operations with point attributes
- GetPoint() – returns the point determinated by indices in the grid

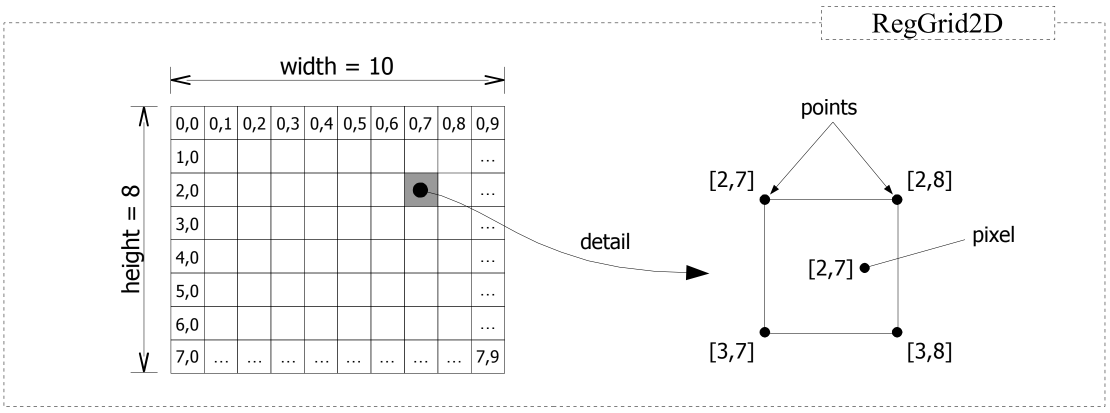


Figure 9: Structure of RegGrid2D DataSet.

Example: operations with the RegGrid2D

```
RegGrid2D rg = new RegGrid2D(10, 20);
UniformDataArray2D attrs = new UniformDataArray2D(typeof(ColorRGBA),10,20);
attrs[0,0] = new ColorRGBA(255, 0, 0);
rg.AddCellAttrs("ColorRGBA", attrs);
UniformDataArray2D myAttrs = rg.GetCellAttrs("ColorRGBA");
Point2D point = rg.GetPoint(5, 5);
```

First we create the grid with given proportions and we create uniform two-dimensional array, which will containts pixel attributes. For the association of attributes with pixels we use method *AddCellAttrs()*. If we want to get the attributes of certain pixel, we use the *GetCellAttrs()* method. For manipulation with point attributes there are analogous methods.

UnstrGrid

Description: this dataset is more general than RegGrid2D, which works with pixels only. UnstrGrid is able to contain arbitrary cell types. However, the memory requirements are greater. UnstrGrid has no internal structure. Data must be represented explicitly and must be physical stored in the memory. Indexing in this dataset is linear, we use one index only.

Containts:

- points – uniform one-dimensional array (accessible through *Points* property).
- cells – a table of uniform arrays
- point attributes – a table of point attributes
- cell attributes – a table of cell attributes

Summary of basic methods:

- Point – a property for access to points array
- AddCells(), GetCells(), RemoveCells() – manipulation with cells
- AddPointAttrs(), GetPointAttrs(), RemovePointAttrs() – manipulation with point attributes
- AddCellAttrs(), GetCellAttrs(), RemoveCellAttrs() – manipulation with cell attributes

Example: operations with the UnstrGrid

```
UnstrGrid ug = new UnstrGrid();
// create and add points array into the DataSet
UniformDataArray points = new UniformDataArray(typeof(Point3D), 4);
points[0] = new Point3D(0, 0, 0);
points[1] = new Point3D(0.5, 1, 3.2);
points[2] = new Point3D(-1.5, 4.8, 3.2);
points[3] = new Point3D(2.2, 3.6, -1.4);
ug.Points = points;
// create cells array
UniformDataArray cells = new UniformDataArray(typeof(Triangle), 2);
cells[0] = new Triangle(0, 1, 2);
cells[1] = new Triangle(3, 0, 1);
ug.AddCells("Triangle", cells);
// add cell attributes
UniformDataArray cellAttrs = new UniformDataArray(typeof(ColorRGBA), 2);
cellAttrs[0] = new ColorRGBA(255, 0, 0);
cellAttrs[1] = new ColorRGBA(0, 255, 0);
ug.AddCellAttrs("Triangle", "ColorRGBA", cellAttrs);
// remove cell attributes
Console.WriteLine(ug);
ug.RemoveCellAttrs("Triangle", "ColorRGBA");
// remove cells
ug.RemoveCells("Triangle");
```

SliceSet
This dataset is inherited from UnstrGrid. Principle of SliceSet is similar to UnstrGrid, but there is one diference. SliceSet can contain Slices as cells only.

Mesh
Mesh is similar to SliceSet. Mesh can contain Triangles, TriangleFans, TriangleStrips, Lines and PolyLines only.
Mesh has two childs:

- *TriangleMesh* – can contain Triangles only.
- *TriangleStripMesh* – can contain TriangleStrips only.

3.3. Creation of user data structure

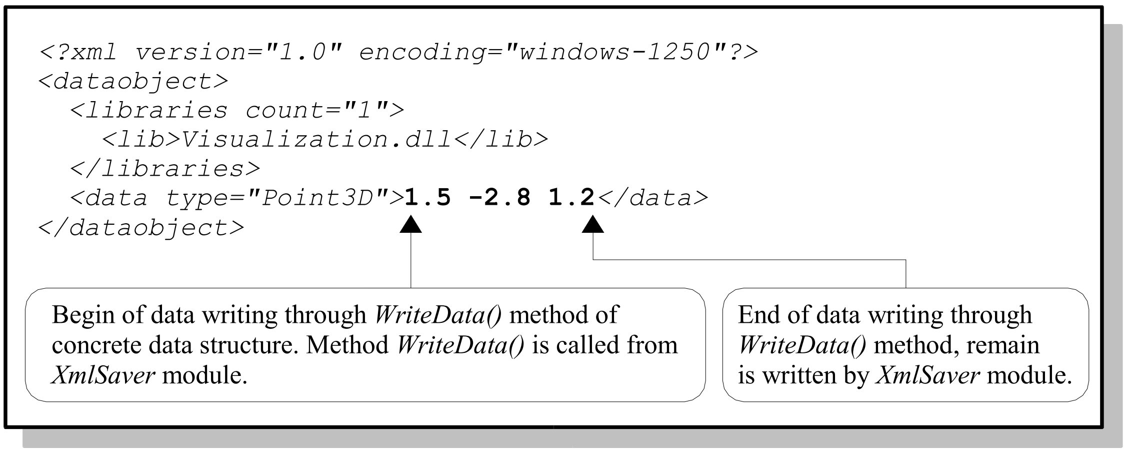
The *Visualization* library includes rich offer of data structures, which mostly suffices. This basic set of data structures can be arbitrary expanded. This chapter describes how to create user defined data structure. All of data structures implement *IDataObject* interface, which is a component of MVE-2 core. Therefore all of data structures can be sended between modules. Additional each of points implement *IPoint* interface, cells implement *ICell*, attributes *IAttribute* interface and all of datasets ihnerit from the abstract class *DataSet*. Thereby in data structures is created hierarchy.

3.3.1. Mandatory overriding

Each of data structure must implement *IDataObject* interface, which declares following methods:

- `public abstract void WriteData(XmlTextWriter xmlTextWriter);`
  This method is used by *XmlSaver* module for saving data into XML. XmlSaver module finds out

a library list to data type composition and write this list into the file. Own data of structure are written into the file through *WriteData()* method.



```
<?xml version="1.0" encoding="windows-1250"?>
<dataobject>
  <libraries count="1">
    <lib>Visualization.dll</lib>
  </libraries>
  <data type="Point3D">1.5 -2.8 1.2</data>
</dataobject>
```

Begin of data writing through *WriteData()* method of concrete data structure. Method *WriteData()* is called from *XmlSaver* module.

End of data writing through *WriteData()* method, remain is written by *XmlSaver* module.

- `public abstract void ReadData(XmlTextReader xmlTextReader);`
  This method is used by *XmlLoader* module. It reads data from XML and works analogous to *WriteData()* method. It is possible manually edit data in the file by user because data format is human readable and transparent. Therefore it is suitable have in mind and - for example - use *Trim()* method for removing all white characters from begin and end of the string.

```
string[] coordinates = xmlTextReader.ReadString().Trim().Split(' ');
```

- `public abstract DataObject DeepCopy();`
  Method for deep copy of instance. Returned object contains identical data but allocates another memory space.

- `public abstract bool CheckConsistence();`
  This method is important for complicated data structures in which there can be some unconsistency. For example a triangle with one index out of bounds, a normal with size greater than 1 +/- epsilon etc. In these cases *CheckConsistence()* method returns *false* and print info about unconsistency on standard output using *Console.WriteLine()*. Also it is possible in this method print some warnings although *true* value is returned - in case that data are consistent but something there isn't absolutely correct.

3.3.2. Possible overriding

It is suitable to override the *ToString()* method.

```
public override string ToString()
{
   return "Point3D: [" + this.X + "; " + this.Y + "; " + this.Z + "]";
}
```

Sometimes it is useful to implement implicit conversion. For example *Point2D* → *Point3D*:

```
public static implicit operator Point3D(Point2D p)
{
   return new Point3D(p.X, p.Y, 0);
}
```

And sometimes we need also explicit conversion. For example *Point3D* → *Point2D*:

```
public static explicit operator Point2D(Point3D p)
{
    return new Point2D(p.X, p.Y);
}
```

Data structure can contain array as field. There are also structures with array character (but their internal structure don't contain array). For example the *Triangle* has 3 fields named V1, V2, V3, which determine vertices of triangle. We can imagine that these 3 fields create one array. In such a case it is suitable to implement *indexer* (a special property of object) for this structure.

```
Triangle tri = new Triangle(0, 2, 4);  // creation of structure
Console.WriteLine(tri);                // print: Triangle: [0, 2, 4]
Console.WriteLine(tri[1]);             // print: 2
tri[1] = 3;                            // use of indexer
Console.WriteLine(tri);                // print: Triangle: [0, 3, 4]
```

Indexer is a property. So it can contain *get* and *set* section.

```
public int this[int index]
{
      get
      {
          switch (index)
          {
            case 0 : return V1; break;
            case 1 : return V2; break;
            case 2 : return V3; break;
            default: throw new MveException("Index out of range...");
          }
      }
      set
      {
          switch (index)
          {
            case 0 : V1 = value; break;
            case 1 : V2 = value; break;
            case 2 : V3 = value; break;
            default: throw new MveException("Index out of range...");
          }
      }
}
```

3.3.3. Creation of DataSets

One compound data structure can be distributed into a few libraries. When we save data into XML format, the node *libraries* must contain list of all used libraries.

```
<?xml version="1.0" encoding="windows-1250"?>
<dataobject>
  <libraries count="2">
    <lib>Visualization.dll</lib>
    <lib>Numerics.dll</lib>
  </libraries>
  <data type="UnstrGrid">
    ...
  </data>
</dataobject>
```

This node is written automatically (as well as whole file header) by *XmlSaver* module. Each of data structure contains its own libraries list named *libNames*. Into this list is automatically added the name of structure base library. If we want add a new library name into libNames list, we use *AddLibName()* method:

```
public void AddLibName(string libName)
public void AddLibName(IDataObject dataObject)
```

For XML data reading we use *XmlLoader* module. This module reads the libraries list from the file, creates object and calls *ReadData()* method. If we need create some object saved in XML, at first we must find the type in assembly. For this we use the static method *Globals.FindType():*

```
public static Type FindType(IEnumerable libNames, string typeName)
```

If the structure isn't found in the *libNames* list, MVE-2 searchs all available libraries, which is much slower. That is the reason why the list *libNames* is saved into XML file.