Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

MVE - 2

Knihovna Visualization

Miroslav Vavruška, Milan Frank
23. 5. 2005 (Verze: alfa-3)

Obsah

1. Úvod

Paralelně s vývojem systému MVE-2 byl započat vývoj knihovny *Visualization*, která by se měla stát určitým standardem a společnou styčnou plochou různých vývojářů a výzkumníků pracujících v oblasti počítačové grafiky, vizualizace dat a zpracování obrazu vůbec. Tento dokument si klade za cíl přiblížit filosofii této knihovny co nejširšímu okruhu zájemců.

Je těžké navrhnout a implementovat zcela od nuly rozumný systém pro reprezentaci dat tak různorodých jakými se vyznačuje právě počítačová grafika a vizualizace dat. Proto jsme se nechali inspirovat datovým modelem použitým ve VTK (Visualization Toolkit) od firmy Kitware.

2. Datový model pro vizualizaci

Vedle algoritmů představujících vlastní funkčnost hrají neméně důležitou roli i vlastní data. Vizualizační prostředí vznikla jako odpověď na požadavek zpracování obrovského množství informací, které nebylo možné dosud známými metodami účelně zpracovávat, analyzovat a zobrazovat. Orbitální družice chrlící terabyty dat každý den, laserové skenovací systémy generující 500.000 bodů každých 15 sekund, superpočítače předpovídající počasí pro celou planetu, New Yorská burza s 333 milióny transakcí každý den – to vše jsou oblasti, které by bez vizualizace nemohly existovat. Je tedy zřejmé, že je nesmírně důležité navrhnout efektivní datové struktury pro ukládání dat a přístup k nim.

Datové struktury by měli být:

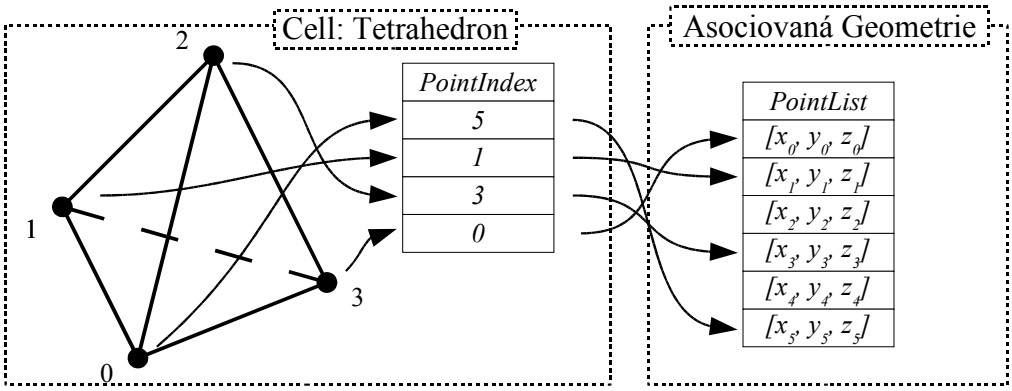
- *kompaktní* – vizualizovaná data bývají velká, je nutné věnovat dostatek pozornosti minimalizaci paměťových požadavků.
- *výkonné* – data musí být snadno a rychle přístupná, naší snahou je přístup k datům v konstantním čase (nezávisle na jejich velikosti).
- *jednoduché* – platí zlaté rčení „v jednoduchosti je genialita“, jednoduché struktury jsou navíc snadno pochopitelné a použitelné.

2.1. Dataset

Dataset neboli datová množina pro vizualizaci je abstraktní pojem představující datový objekt. Dataset je tvořen *strukturou* a *atributy*. Struktura má určitou *topologii* a *geometrii*. Topologie je množina vlastností, které se nemění geometrickými transformacemi (rotace, translace apod.). Geometrie je specifikace pozice v prostoru. Struktura datasetu se skládá z buněk (*cells*) a bodů (*points*). Buňky určují topologii, zatímco body určují geometrii. Atributy jsou libovolné dodatečné informace asociované se strukturou, čili s body nebo buňkami. Může se jednat například o teplotu v bodě, hmotnost buňky, tepelné proudění do/ze stěny buňky apod.

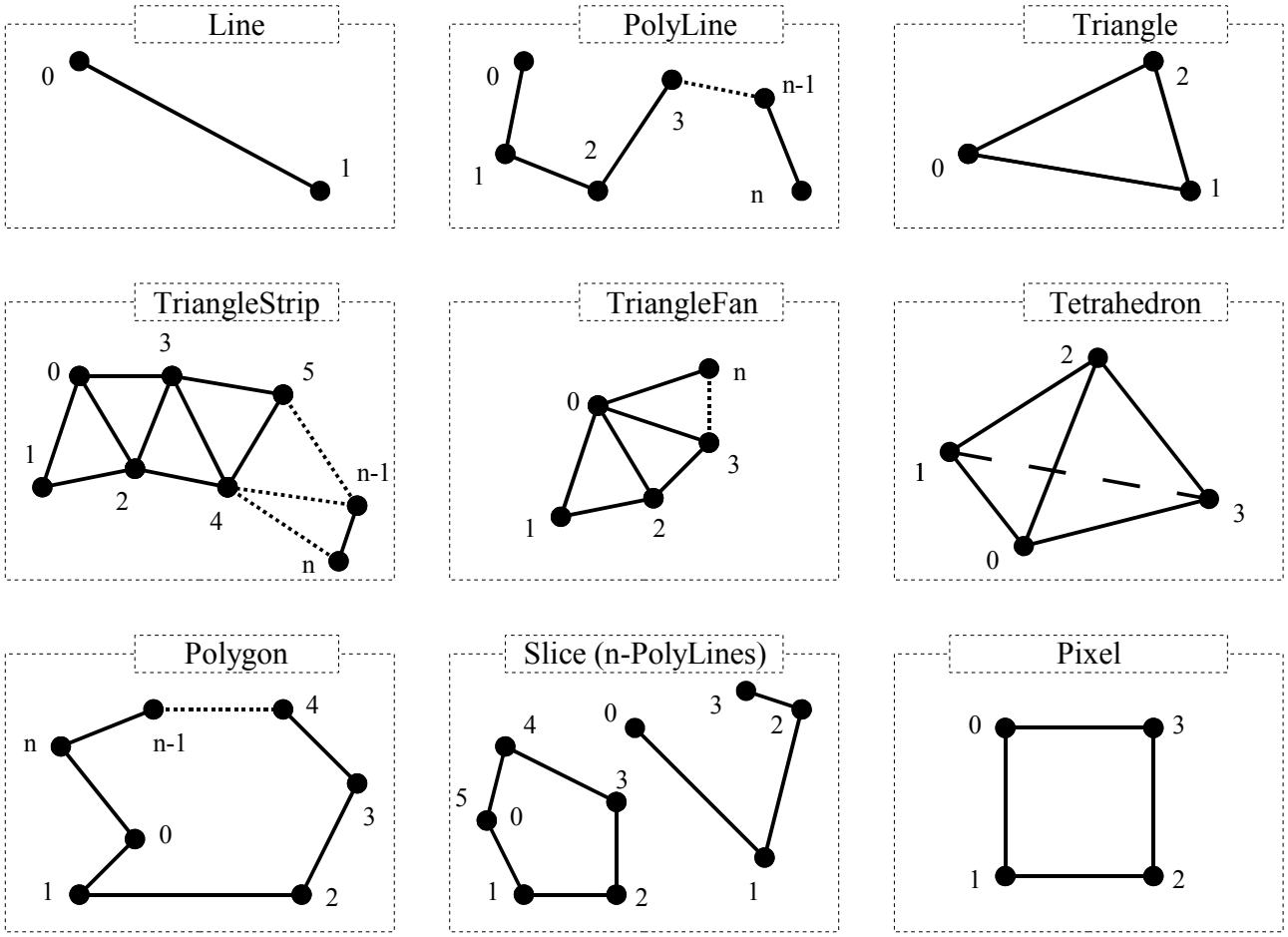
2.2. Buňky

Buňka je základní stavební prvek vizualizačního systému. Každý dataset se skládá z jedné nebo více buněk. Buňka je definována uspořádaným seznamem bodů (*connectivity list, point list*), které ji tvoří. Seznam bodů v kombinaci se specifikací buňky definuje její topologii. Buňka může být primární nebo složená. Složená buňka se skládá z jedné nebo více primárních buněk. Primární buňka je dále nedělitelná.



Obrázek 1: Topologický element čtyřstěn s naznačenou vazbou na geometrii.

Tetrahedron je dán uspořádaným seznamem 4 bodů. Topologie této buňky je implicitně známa: víme, že (1,2) je jedna z 6 hran tetrahedronu a (1,2,3) je jedna ze 4 jeho stěn.

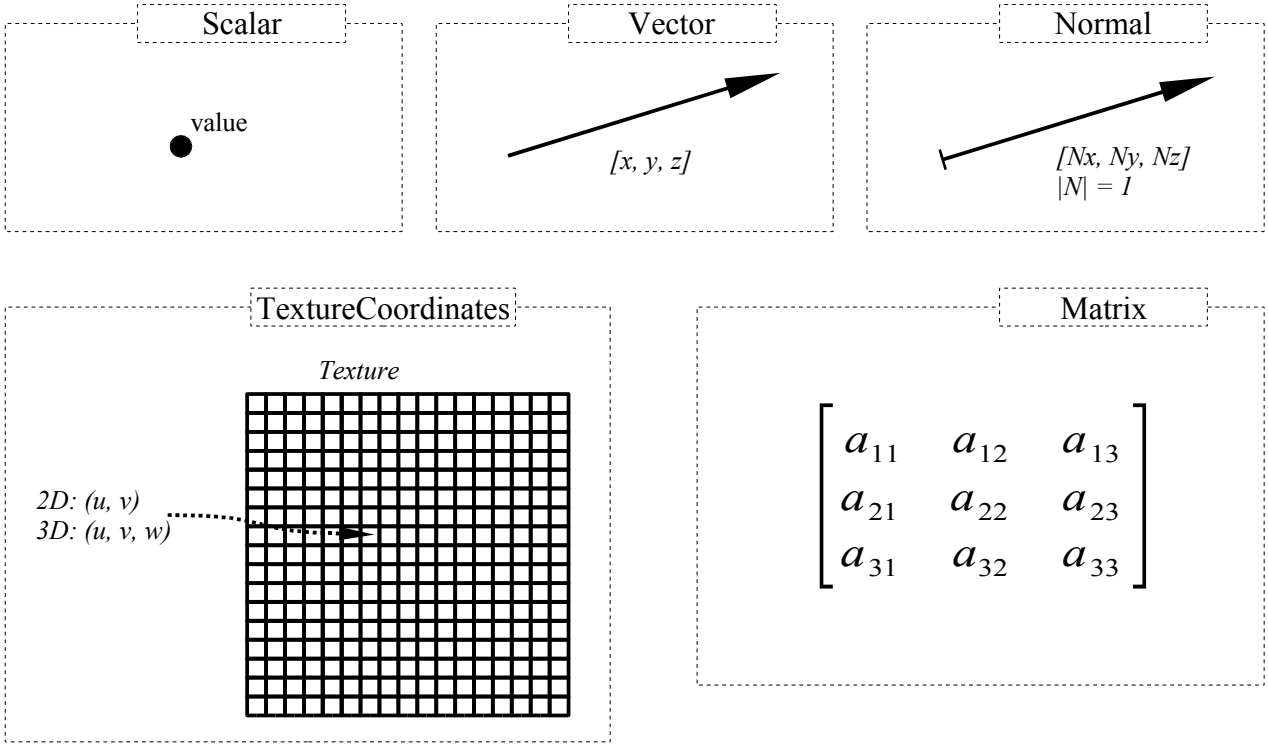


Obrázek 2: Základní typy buněk.

Samozřejmě lze vymyslet velké množství dalších druhů buněk. Zde je uveden základní soubor nejčastěji používaných buněk. Byly vybrány pro své vlastnosti a praktické použití v mnoha oblastech. Ve velkém množství případů tento soubor buněk postačuje.

2.3. Atributy

Atributy jsou informace asociované s body nebo buňkami. Typickými atributy jsou například teplota nebo rychlost v bodě, hmotnost buňky, tepelné proudění do/ze stěny buňky apod.



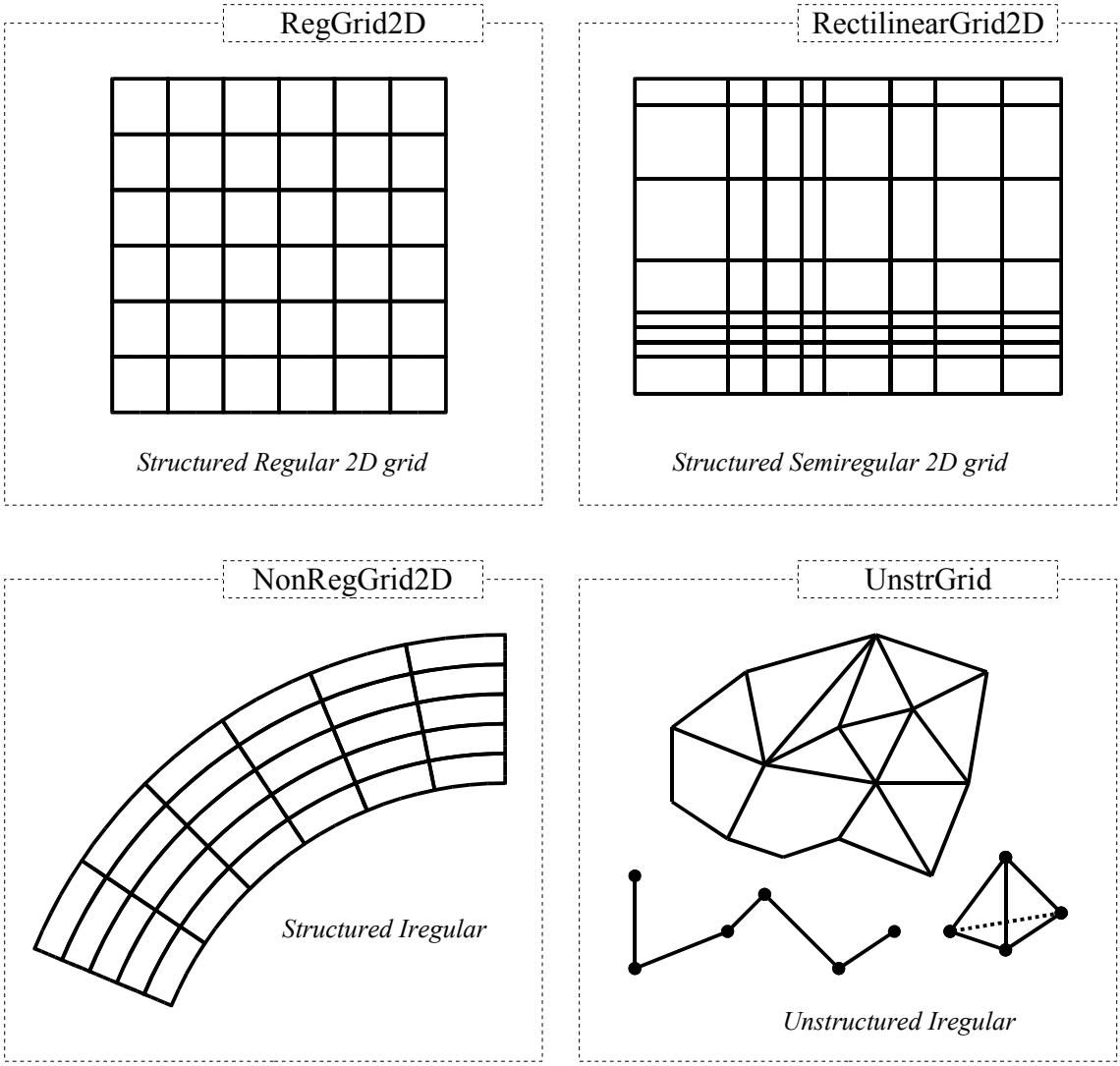
Obrázek 3: Základní typy atributů.

Máme několik základních atributů. Tyto atributy byly vybrány na základě běžně zpracovávaných dat a praktických zkušeností.

2.4. Typy datových množin

Datové množiny (*Datasets*) se skládají z organizační struktury a přidružených atributů. Struktura má topologické vlastnosti (buňky, cell) a geometrické vlastnosti (body, points) a skládá se z jednoho nebo více bodů a buněk. Datové množiny lze rozdělit na *strukturované* a *nestrukturované*. Strukturovaná datová množina je charakteristická tím, že její topologie je dána implicitně. Což má za následek úsporu paměti. Naproti tomu topologii nestrukturované datové množiny nelze popsat žádným matematickým vztahem. Jejich data musejí být reprezentována explicitně. Nestrukturované datové množiny jsou obecnější než strukturované, ale mají daleko větší paměťové požadavky. Používají se pouze tehdy, je-li to nezbytně nutné. Následující obrázek ukazuje několik základních datových množin. Opět platí, že jich lze vymyslet velké množství. Zde uvedeme pouze nejpoužívanější z nich.

Další důležitou vlastností datových množin je jejich pravidelnost. Pokud můžeme geometrii (pozici bodu) vyjádřit matematickým vztahem pak hovoříme o datech pravidelných či částečně pravidelných. Pokud jsme nuceni vyjadřovat explicitně pozici každého bodu, pak hovoříme o datech nepravidelných.



Obrázek 4: Typy datových množin.

3. Knihovna Visualization

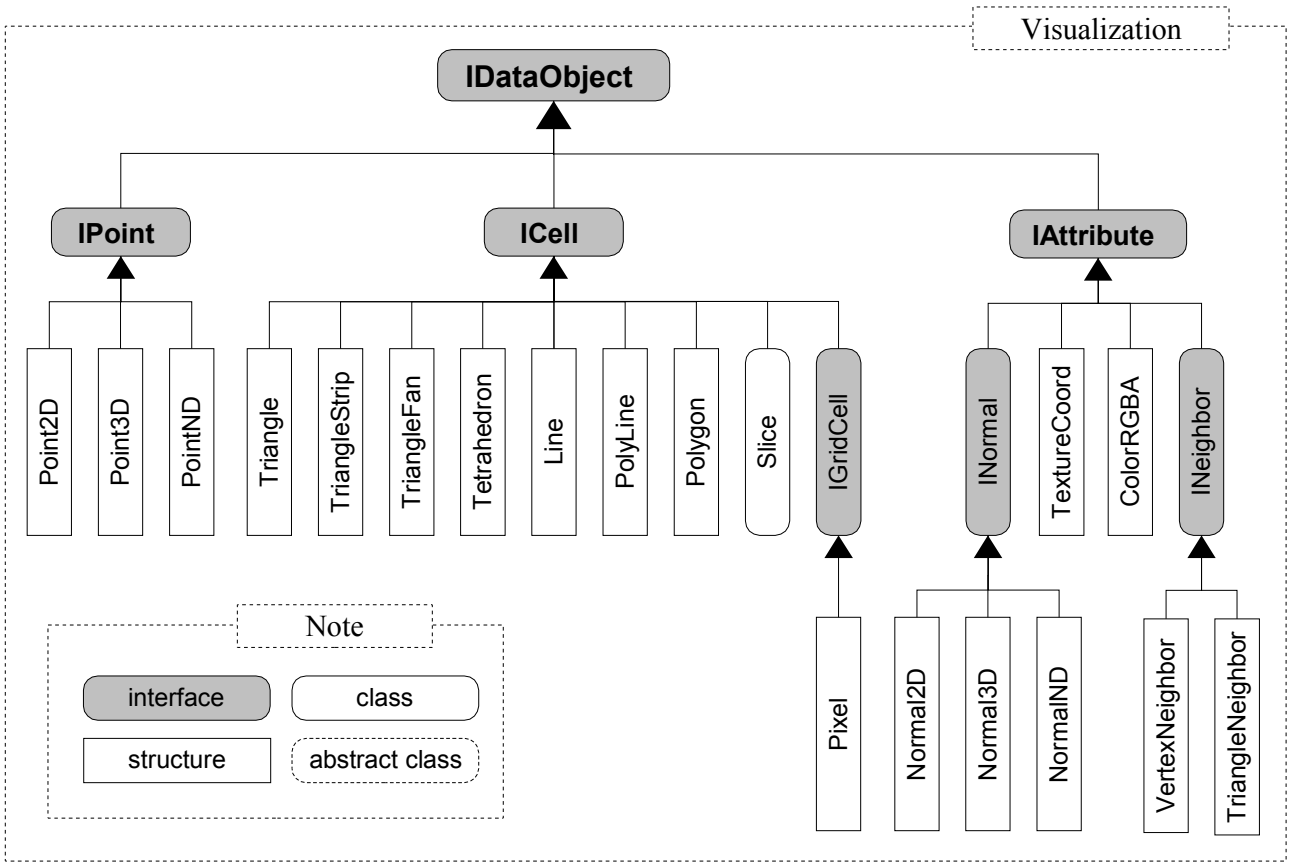
Součástí MVE2 je knihovna *Visualization*, ve které jsou implementovány základní datové struktury jako body, buňky, atributy a datové množiny. Tato kapitola nepopisuje princip práce s již vytvořenými strukturami (k tomu slouží automaticky generovaná dokumentace), nýbrž filozofii a přehled datových struktur této knihovny, konverze mezi datovými typy, ukládání datových struktur do XML souborů a princip tvorby vlastních, uživatelských datových struktur.

3.1. Implementované datové struktury

Tato kapitola popisuje množinu základních datových typů, která je primárně nezávislá na vlastním jádře systému. Předpokládá se však její doporučené použití v rámci vizualizačních a grafických aplikací. Je to nutný základ pro vzájemnou kompatibilitu modulů různých tvůrců. Samozřejmě, že mohou vznikat libovolné další datové typy, ale je bezpochyby nutné udržovat základní množinu datových typů, které budou používány širokou veřejností tvůrců modulů.

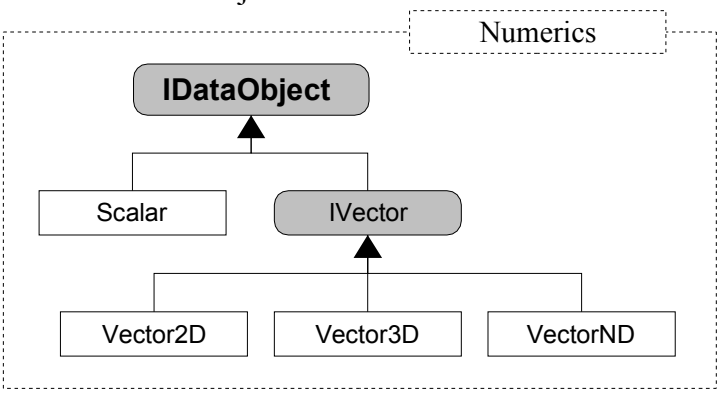
3.1.1. Základní datové typy

Následuje grafické znázornění dědičnosti základních datových typů. Každý z nich implementuje rozhraní *IDataObject*, tedy je možné aby byly tyto elementární datové struktury zasílány mezi moduly. Hlavní změnou mezi verzí *alfa-3* a *alfa-2* bylo předělání většiny elementárních typů ze tříd na struktury (tedy hodnotové datové typy) z důvodu efektivity. Zpracování objektů znamená poměrně velkou režii. Hierarchie datových typů je docíleno pomocí dědičnosti jednotlivých rozhraní. Struktury poté implementují odpovídající rozhraní podle toho, do jaké kategorie datových typů patří (například všechny body implementují rozhraní IPoint, buňky implementují ICell apod.).



Obrázek 5: Objektový diagram základních datových typů knihovny Visualization.

Poznámka: v knihovně *Numerics* jsou implementovány další důležité datové typy, které jsou obecně používané a užitečné. Jedná se o následující:



Obrázek 6: Objektový diagram datových typů knihovny Numerics.

3.1.2. Konverze mezi datovými typy

V některých případech požadujeme přetypování jednoho datového typu na jiný. Proto zde jsou konverze. Tam, kde nedochází ke ztrátě dat (rozšiřující konverze – např. *Point2D → Point3D*), je použita implicitní konverze (automatická) – při přetypování není nutno uvádět datový typ na který přetypováváme. Naopak tam, kde dochází ke ztrátě dat (zužující konverze – např. *Point3D → Point2D*), je použito explicitní konverze – při přetypování je nutno uvést datový typ. Explicitní konverze je rovněž implementována pro případy, kdy konvertujeme na typ který je něčím specifický a tato specifika je nutno si uvědomit – např. *Vector → Normal*.

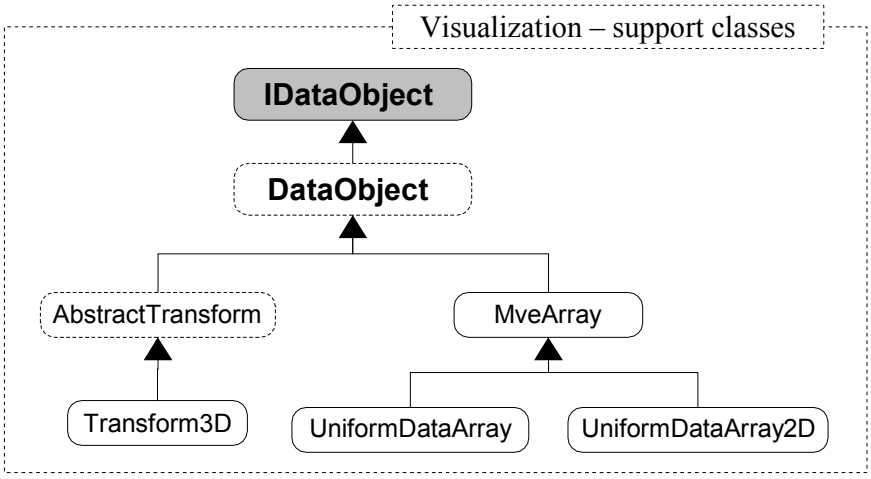
Implementované konverze:

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

3.1.3. Pomocné třídy

Následující obrázek zachycuje pomocné třídy knihovny Visualization:

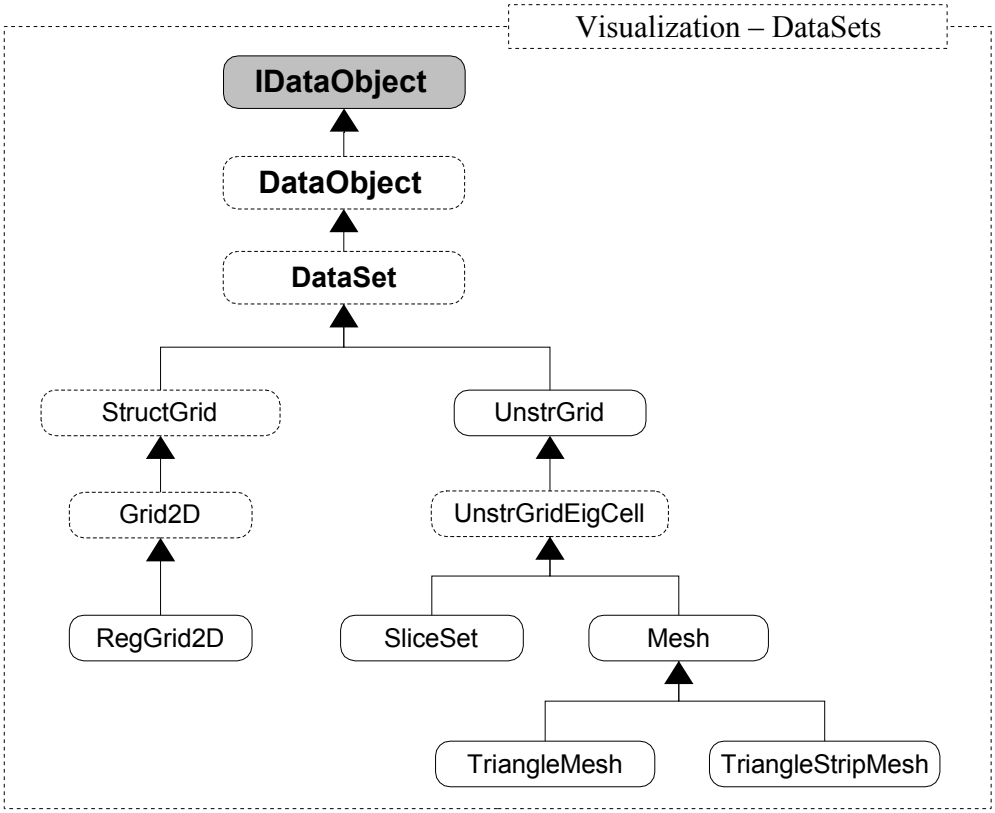
- Transform3D - obecná transformace 3D objektu
- UniformDataArray – jednorozměrné homogenní statické pole datových objektů
- UniformDataArray2D – dvourozměrné homogenní statické pole datových objektů



Obrázek 7: Pomocné třídy knihovny Visualization.

3.1.4. Datové množiny (Datasets)

Následující datové struktury reprezentují různé logické uspořádání výše popsaných elementárních datových struktur. Každý element může mít libovolné množství atributů. Atribut je s bodem či buňkou asociován pomocí indexu.



Obrázek 8: DataSety knihovny Visualization.

3.1.5. Ukládání datových struktur do XML

MVE2 přímo podporuje ukládání datových struktur do XML souborů. Formát XML byl vybrán především pro svůj vysoký informační obsah, snadnou čitelnost a rozšířenost. Jelikož XML je textový formát a používá tagy k oddělení dat, je soubor s daty téměř vždy větší než srovnatelná data v binárním formátu, což je jistá nevýhoda. Ovšem jak již bylo řečeno, MVE2 předpokládá využití možnosti uložení dat do XML především pro svoji názornost, přehlednost a čitelnost, nikoli pro archivaci velkých objemů dat.

Pro ukládání dat do XML souboru slouží modul jádra *XmlSaver*, pro načítání se používá modul *XmlLoader*. Obecně platí, že každá datová struktura implementuje rozhraní *IDataObject* (resp. je potomkem abstraktní třídy *DataObject*) a musí mimo jiné implementovat metodu *WriteData()* pro uložení dat do XML souboru a metodu *ReadData()* pro načtení dat z XML souboru, kde by měla být konkrétní data uložena především čitelně a v druhé řadě efektivně. Tyto ukládací/načítací metody píše tvůrce datové struktury, který nejlépe ví jak strukturu uložit v textovém formátu.

Následuje příklad uložení datového typu Point3D. Uzel *libraries* obsahuje seznam knihoven, které jsou potřebné k vytvoření datového typu. V případě jednoduchých datových typů je zde název jediné knihovny. Komplikovanější situace nastává, je-li datový typ složen z jiných typů, které se nacházejí v různých knihovnách (viz. datasety UnstrGrid, RegGrid2D apod.). V tom případě zde musí být uveden seznam všech potřebných knihoven. Atribut *type* uzlu *data* odpovídá názvu datového typu a uzel *data* obsahuje vlastní data. Moduly *XmlSaver*, *XmlLoader* mají na starosti hlavičku souboru, uzly *dataobject, libraries* a uzel *data* s atributem *type*. Obsah uzlu *data* je plně v moci autora ukládací/načítací metody.

```
<?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>
```

Pozn.: z důvodu snížení paměťových požadavků se při práci s XML soubory neudržuje v paměti celý jejich obsah, nýbrž se s nimi pracuje sekvenčně.

3.2. Použití datových struktur

Tato kapitola má za úkol ukázat princip práce s implementovanými datovými strukturami.

3.2.1. Základní datové typy

Práce se základními datovými typy je velice jednoduchá. Díky tomu, že všechny datové struktury mají překrytu metodu *ToString()*, lze o nich kdykoliv vypsat užitečné informace. Datové struktury, které mají charakter pole, mají implementován přístup pomocí indexeru. Situaci ilustruje následující příklad:

```
Triangle tri = new Triangle(0, 2, 4);  // vytvoreni struktury
Console.WriteLine(tri);                // vypise: Triangle: [0, 2, 4]
Console.WriteLine(tri[1]);             // vypise: 2
tri[1] = 3;                            // pristup pomoci indexeru
Console.WriteLine(tri);                // vypise: Triangle: [0, 3, 4]
```

Tam, kde je to vhodné a možné, je implementována rovněž konverze. Implicitní konverzi popisuje následující příklad, kde přetypujeme *Point2D* na *Point3D*:

```
Point2D p2 = new Point2D(1.5, 2.3);     // vytvoreni struktury
Console.WriteLine(p2);                  // vypise: Point2D: [1.5; 2.3]
Point3D p3 = p2;                        // implicitni konverze
Console.WriteLine(p3);                  // vypise: Point3D: [1.5; 2.3; 0]
```

A zde je ukázka opačné, explicitní konverze. *Point3D* zde přetypováváme na *Point2D*:

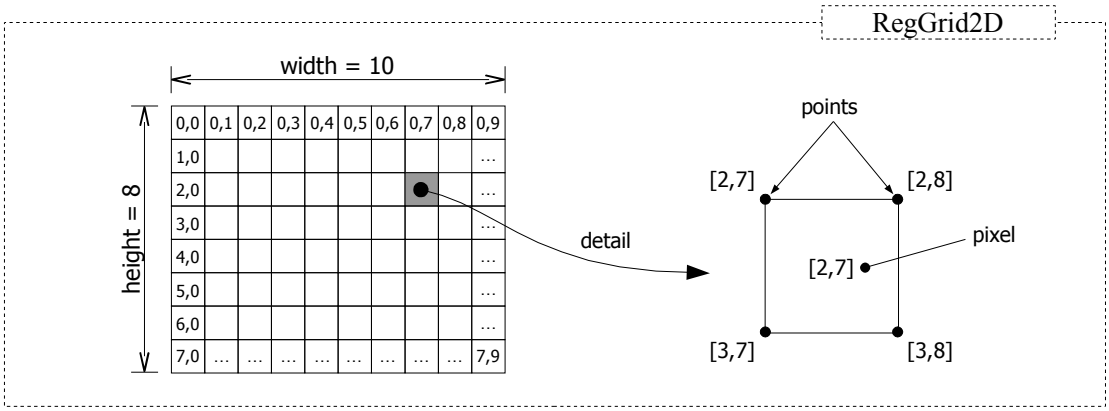
```
Point3D p3 = new Point3D(1.5, 2.3, 3.8); // vytvoreni struktury
Console.WriteLine(p3);                   // vypise: Point3D: [1.5; 2.3; 3.8
Point2D p2 = (Point2D) p3;               // explicitni konverze
Console.WriteLine(p2);                   // vypise: Point2D: [1.5; 2.3]
```

3.2.2. DataSety

RegGrid2D

Popis: jedná se o dataset, který představuje strukturovanou, pravidelnou, pravoúhlou mřížku. Ta se skládá z pixelů, přičemž každý pixel je tvořen čtyřmi body. K pixelům a bodům se přistupuje pomocí dvou indexů. S každým pixelem i bodem může být asociován libovolný počet atributů. Mřížka je dána rozlišením (width a height), což je počet pixelů ve směru osy x a y. Počet pixelů

mřížky je tedy width x height, zatímco počet bodů je (width + 1) x (height + 1). Tuto datovou strukturu lze použít například pro uložení obrázku (bitmap).



Obrázek 9: Filosofie datové struktury RegGrid2D.

Obsahuje:

- body a pixely – pouze logicky. Jelikož je mřížka strukturovaná, lze je kdykoliv implicitně určit a nemusejí tudíž být v paměti fyzicky uloženy.
- atributy bodů a pixelů – ve formě uniformního (homogenního) dvourozměrného pole.

Přehled základních metod:

- AddCellAttrs(), GetCellAttrs() – práce s atributy buněk
- AddPointAttrs(), GetPointAttrs() – práce s atributy bodů
- GetPoint() – vrací bod daný indexy v mřížce

Příklad: princip práce s RegGrid2D ukazuje následující příklad:

```
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);
```

Nejprve vytvoříme mřížku daných rozměrů. Následně vytvoříme uniformní dvourozměrné pole (uniformní znamená, že všechny prvky pole jsou jednoho datového typu), které bude reprezentovat atributy pixelů, což zajistíme pomocí metody AddCellAttrs(). Rovněž je ukázáno vkládání atributu do uniformního pole pomocí indexu a získání uniformního 2D pole atributů z objektu RegGrid2D pomocí metody GetCellAttrs(). Pro práci s atributy bodů slouží analogické metody. Nakonec je ukázán příklad získání konkrétního bodu zadaného pomocí indexů v mřížce.

UnstrGrid

Popis: tento dataset je daleko obecnější než předchozí. Zatímco RegGrid2D pracoval pouze s pixely, UnstrGrid je schopen pojmout libovolné typy buněk. Nevýhodou je však velká paměťová náročnost. Jelikož UnstrGrid nemá žádnou vnitřní strukturu, data v něm uložená musejí být reprezentována explicitně a musejí být v paměti fyzicky uložena. Indexace buněk uvnitř tohoto datasetu je lineární, pomocí jediného indexu.

Obsahuje:

- body – uniformní jednorozměrné pole (přístupné pomocí property Points)
- buňky – tabulka ve které každá položka představuje uniformní pole buněk jednoho typu

- atributy bodů – tabulka obsahující jako položky pole atributů bodů
- atributy buněk – tabulka, jejíž položkou je tabulka obsahující pole atributů buněk

Přehled základních metod a vlastností:

- Point – property pro přístup k poli bodů
- AddCells(), GetCells(), RemoveCells() - práce s buňkami
- AddPointAttrs(), GetPointAttrs(), RemovePointAttrs() - práce s atributy bodů
- AddCellAttrs(), GetCellAttrs(), RemoveCellAttrs() - práce s atributy buněk

Příklad: nevyžaduje podrobnější komentář. Pro práci s atributy bodů se používají analogické metody jako při práci s atributy buněk.

```
UnstrGrid ug = new UnstrGrid();
// vytvoreni a pridani pole bodu do DataSetu
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;
// vytvoreni pole bunek
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);
// pridani atributu bunek
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);
// odstraneni atributu bunek
Console.WriteLine(ug);
ug.RemoveCellAttrs("Triangle", "ColorRGBA");
// odstraneni bunek
ug.RemoveCells("Triangle");
```

SliceSet

Jedná se o dataset, který je potomkem UnstrGrid. Pracuje se s ním stejně a má i stejné vlastnosti s tím rozdílem, že jediné buňky které může obsahovat jsou řezy (Slices).

Mesh

Podobně jako předchozí dataset – může obsahovat pouze buňky typu: Triangle, TriangleFan, TriangleStrip, Line a PolyLine.
Tento dataset má navíc 2 potomky:

- *TriangleMesh* – jediné povolené buňky jsou Triangle.
- *TriangleStripMesh* – jediné povolené buňky jsou TriangleStrip.

3.3. Tvorba vlastních datových struktur

Knihovna *Visualization* obsahuje bohatou nabídku datových struktur, které ve většině případů postačují. Tuto množinu lze však libovolně rozšiřovat. Tato kapitola je věnována tvorbě vlastních datových struktur.

Všechny datové struktury implementují rozhraní *IDataObject* (resp. jsou potomky abstraktní třídy *DataObject)*, které je součástí jádra MVE2. Tím je zajištěno, že lze všechny tyto datové struktury zasílat mezi moduly. Navíc platí, že všechny body implementují rozhraní *IPoint*, buňky implementují *ICell*, atributy *IAttribute* a všechny datové množiny dědí od abstraktní třídy *DataSet*. Tím je v datových strukturách vytvořena hierarchie.

3.3.1. Povinně implementovat

Jelikož všechny datové struktury implementují rozhraní *IDataObject*, ve kterém je deklarováno několik metod, je nutné tyto metody ve strukturách implementovat.

Jedná se o následující metody:

- `public abstract void WriteData(XmlTextWriter xmlTextWriter);`
  Tuto metodu využívá modul *XmlSaver*, který slouží k ukládání dat do XML souboru. Modul pracuje tak, že ze vstupních dat zjistí typ dat, potřebné knihovny pro sestavení datového typu a do souboru zapíše sekvenčně pomocí třídy *XmlTextWriter* například následující data (pro *Point3D*) - označeno kurzívou:

```
<?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>
```

Začátek zápisu dat pomocí metody *WriteData()* příslušného datového typu – z modulu *XmlSaver* je zavolána metoda *WriteData()* a té je předán parametr *xmlTextWriter*

Konec zápisu dat pomocí metody *WriteData()*, zbytek zapíše opět *XmlSaver*.

  Úkolem metody *WriteData()* je zapsat do souboru vlastní data která datový typ uchovává. Data se do souboru zapisují sekvenčně pomocí parametru *xmlTextWriter*. Modul *XmlSaver* zapíše hlavičku souboru až do místa pro vlastní data. Následně zavolá metodu *WriteData()*, která tato data zapíše. Poté jsou pomocí *XmlSaveru* zapsány uzavírací tagy a soubor je uzavřen. Na příkladu vidíme tučně vyznačená data zapsaná do souboru pomocí metody *WriteData()* třídy *Point3D*.

- `public abstract void ReadData(XmlTextReader xmlTextReader);`
  Tuto metodu využívá modul jádra *XmlLoader*. Slouží k načtení dat z XML souboru a pracuje analogicky k metodě *WriteData()*. Jelikož data jsou v XML souboru uložena čitelně a přehledně, lze předpokládat manuální zásahy a úpravu dat uživatelem. S tím je vhodné počítat a při načítání mimo jiné použít metodu *Trim()* pro oříznutí bílých znaků. Například pro načtení souřadnic objektu Point3D:

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

- `public abstract DataObject DeepCopy();`
  Metoda pro vytvoření hluboké kopie instance datového typu. Je vrácen objekt, který obsahuje stejná data, ovšem alokuje v paměti jiný paměťový prostor.

- `public abstract bool CheckConsistence();`
  Metoda *CheckConsistence* je důležitá především pro komplikované datové struktury, kde může docházet k nekonzistencím. (např.: trojúhelník, jehož jeden z indexů překračuje počet bodů, normála jejíž velikost je různá od jedné +/- epsilon, atd.) V takových případech má metoda vrátit hodnotu *false* a tak uživatele datové struktury upozornit na možné problémy při dalším zpracování. V případě že metoda vrací *false* je velmi vhodné popsat příčinu nekonzistence na standardní výstup pomocí *Console.WriteLine()*. Dále je možné vypisovat určitá varování, i když je vrácena hodnotu *true*. Tedy i v případě, že jsou data uznána za konsistentní, ale přesto něco není zcela běžné.

3.3.2. Vhodné implementovat

Dále je vhodné překrýt u všech datových typů metodu *ToString()* tak, aby vracela řetězec obsahující základní informace o datovém typu a tam, kde je to vhodné a možné, i o vlastních datech. Jako příklad je uvedena metoda třídy *Point3D*:

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

Při tvorbě datových typů je nutno zvážit zda je vhodné a možné vytvářený typ konvertovat na jiný a zda použít implicitní nebo explicitní konverzi. Implicitní (rozšiřující) konverzi použijeme typicky tam, kde nedochází ke ztrátě dat, například *Point2D → Point3D*:

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

Naopak explicitní (zužující) konverzi použijeme tam, kde dochází ke ztrátě dat, například *Point3D → Point2D*:

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

Často se setkáme s tím, že některá datová struktura obsahuje jako členskou proměnnou pole, kterým je charakterizovaná. Existují rovněž struktury, na které lze pohlížet jako na pole ikdyž interně jsou řešeny jinak než pomocí pole. Například Triangle obsahuje 3 členské proměnné V1, V2, V3, které udávají jeho vrcholy. Lze si představit, že tyto 3 proměnné tvoří pole. V takovém případě je vhodné datové struktuře přidat zvláštní vlastnost (property) zvanou *indexer* – přístup k členským proměnným třídy bude možný pomocí indexů. Neboli třída se bude navenek jevit jako pole. Celou problematiku si ukážeme na již zmiňované třídě Triangle. Po přidání indexeru bude možné s instancí třídy Triangle pracovat například takto:

```
Triangle tri = new Triangle(0, 2, 4);  // vytvoreni struktury
Console.WriteLine(tri);                // vypise: Triangle: [0, 2, 4]
Console.WriteLine(tri[1]);             // vypise: 2
tri[1] = 3;                            // pristup pomoci indexeru
Console.WriteLine(tri);                // vypise: Triangle: [0, 3, 4]
```

Jak již bylo řečeno, *indexer* je vlastnost. Může mít tedy sekci *get* i *set*. Konkrétní *indexer* pro třídu *Triangle* vypadá následovně:

```
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. Tvorba složených datových struktur (datasetů apod.)

V případě, že vytváříme datové struktury, které se skládají z jiných struktur, musíme vzít v úvahu to, že tyto dílčí struktury nemusejí pocházet ze stejné knihovny. Při ukládání dat do XML souboru je nutno zajistit, aby sekce *libraries* obsahovala seznam všech potřebných knihoven:

```
<?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>
```

Tato sekce se zapisuje automaticky (stejně jako celá hlavička souboru) modulem *XmlSaver*. Každý datový typ obsahuje seznam všech potřebných knihoven *libNames* (dědí od DataObject). Do tohoto seznamu se automaticky přidá jméno knihovny, ze které pochází daný datový typ (u jednoduchých datových typů se proto o nic starat nemusíme). V případě, že datová struktura obsahuje struktury z jiných knihoven, musíme přidat jména těchto knihoven do seznamu *libNames* sami. K tomu slouží přetížená metoda datové struktury *AddLibName()* implementovaná v DataObject:

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

Jak je vidět, lze zadat přímo název knihovny, ale je možné zadat pouze vlastní objekt, název knihovny se pak zjistí automaticky. Metody zajišťují, že seznam *libNames* bude obsahovat každou

knihovnu pouze jednou.
Rovněž načítání dat z XML souboru je o něco složitější než u jednoduchých datových struktur. Modul *XmlLoader* přečte seznam knihoven, vytvoří datový objekt a následně vyvolá metodu *ReadData()* vytvořeného objektu. Jestliže potřebujeme v této metodě vytvořit instanci nějakého datového objektu uloženého v souboru, musíme daný typ nejprve nalézt v odpovídající assembly. K tomu slouží následující statická metoda třídy *Globals*:

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

Jejími parametry je seznam názvů knihoven, ve kterých se bude hledat (načte se automaticky v modulu XmlLoader) a název požadovaného typu. Jestliže metoda nenalezne typ v daných knihovnách, prohledá všechny dostupné knihovny MVE2, což je mnohem pomalejší. To je důvod, proč se spolu s datovou strukturou ukládá i seznam knihoven, které jsou pro ni potřebné.