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

DIPLOMOVÁ PRÁCE

Plzeň, 2006

Jan Kaiser

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

Diplomová práce

Obecný modulární grafický subsystém pro prostředí MVE-2

Plzeň, 2006 Jan Kaiser

Poděkování

Rád bych poděkoval Prof. Ing. Václavu Skalovi, Csc. za vedení mého studia v oboru počítačové grafiky, vedoucímu diplomové práce Ing. Milanu Frankovi za odbornou pomoc a rady při zpracování mé diplomové práce, Ing. Liboru Vášovi a Ing. Ivo Hanákovi za zodpovězení mnohých otázek týkajících se projektů MVE-2 a D3DUT. Nemalý dík též patří rodině a přátelům za podporu a porozumění.

Abstract

MVE-2 is a modular environment developed at the University of West Bohemia. Within this environment user can solve complex problems just by connecting modules. It is clear that value of such an environment grows with amount of useful modules and module libraries.

General interactive rendering system for MVE-2 is a contribution to the MVE-2 project. Goal of this diploma thesis is to create UnstrGridRendering library that contains modules for composing and rendering scene. Interactivity in scope of this project means real time response of rendering subsystem to changes made in the scene. Future migration of this library together with MVE-2 from Windows to Linux/Unix platform should be possible thanks to utilization of the D3DUT accelerated graphics interface.

Obsah

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 20.5.2006, Jan Kaiser, ...

1 Úvod

V roce 1996 vznikla na Západočeské univerzitě v Plzni idea vytvoření vlastního modulárního prostředí primárně určeného pro vizualizaci dat. V takovém prostředí je uživateli umožněno řešit komplexní problémy jednoduše, spojováním modulů. Úkolu se tehdy ujal Martin Roušal, coby své diplomové práce, a výsledkem byla první verze pojmenovaná MVE [MVE]. Prostředí se vyvíjelo pro v té době aktuální platformu Win32API, což přineslo některá omezení. Byla pevně dána množina použitelných datových typů, jednoduchý spouštěcí mechanismus mapy modulů, explicitní řízení uvolňování paměti a problémy s komponentami vyvíjenými v různých vývojářských nástrojích. Přes tato omezení bylo MVE úspěšně využíváno při výuce i výzkumné činnosti.

Uvedení technologie .NET firmou Microsoft spolu s několikaletými zkušenostmi s používáním MVE bylo impulsem k sestavení požadavků a započetí vývoje nové verze. S malým týmem studentů implementoval Ing. Milan Frank pod vedením prof. Václava Skaly druhou verzi MVE-2 [MVE2]. Stěžejními vlastnostmi jsou obecné jádro připravené spouštět moduly z různých aplikačních oblastí, možnost vytvářet mapy obsahující cykly a mapy nižší úrovně, intuitivní rozhraní modulů a datových struktur spolu s možností vytváření a definování vlastních, automatické generování grafické reprezentace a uživatelského prostředí modulu, zabudovaná podpora XML pro ukládání a načítaní datových typů a nastavení modulů, nástroj pro automatické generování dokumentace knihoven.

Je zřejmé, že hodnota a využitelnost modulárního prostředí silně závisí na množství a kvalitě pro něj vyvinutých knihoven modulů. Souběžně s MVE-2 byly proto vyvíjeny knihovny s moduly pro základní matematické operace a reprezentaci grafických dat. MVE-2 je také intenzivně využíváno při výuce a sami studenti přispívají novými knihovnami modulů. Evidentní je však absence knihovny, která by obsahovala moduly pro sestavení zpracovaných dat do scény a jejich následnou hardwarově akcelerovanou vizualizaci.

Předkládaná diplomová práce se snaží onu mezeru zaplnit a přinést řešení v podobě knihovny modulů s očekávanou funkčností. Cílem je tedy poskytnout sadu modulů pro kompozici scény ze zpracovaných dat a její následnou vizualizaci. Požadavek je též kladen na interaktivitu, tj. při sestavování mapy

možnost použít moduly, které v reálném čase mění svůj výstup. Takové moduly mohou a většinou také ovlivňují následně zobrazovaná data a modul, jenž se stará o vizualizaci těchto dat, musí tuto změnu reflektovat. Pro hardwarově akcelerovaný výstup se předpokládá použití rozhraní D3DUT [D3DUT].

D3DUT je rozhraní vyvíjené na Západočeské univerzitě v Plzni. Vzniklo jako potřeba jednotného grafického hardwarově akcelerovaného rozhraní pro CLI prostředí [CLI]. Takovým rozhraním je například Direct3D for Managed code [MD3D]. To je ovšem silně vázáno na Win32 CLI implementaci .NET. Snahou D3DUT je mít stejné rozhraní i pro ostatní CLI implementace, jako je např. Mono nebo SSCLI na platformě Linux. Rozhraní D3DUT je v současné době[1] ve vývoji a nabízí dostatečnou funkcionalitu ve verzi pro platformu Win32.NET.

Od čtenáře se předpokládá znalost objektově orientovaného programování, prostředí .NET a jazyka C# [KACM], přehled v oblasti vizualizace dat a počítačové grafiky [ZARA] a rozhraní Microsoft Direct3D for Managed code [MDX9].

[1] duben 2006

2 Teoretická část

Předkládaná práce si klade za cíl implementovat knihovnu pro modulární prostředí MVE-2 [MVE2]. Podle datové struktury, která slouží jako vstup, je pojmenována UnstrGridRendering. Moduly by měly umožnit kompozici zpracovaných dat a jejich následnou vizualizaci. Vytvoření takové knihovny se neobejde bez detailní znalosti prostředí a jeho spouštěcího mechanismu. Stejně tak důležitá je i znalost základních knihoven Numerics a Visualization.

Akcelerovaný grafický výstup je realizován přes rozhraní D3DUT [D3DUT]. Na místě je tedy popsat, jakým způsobem je třeba s D3DUT zacházet. Použití je ve většině případů shodné s rozhraním Microsoft Direct3D for Managed code [MD3D], navíc jsou přidány třídy ulehčující vytvoření okna pro vykreslování a obsluhy vstupních zařízení.

Inspirací při návrhu knihovny byla již exitujících řešení. Zmíněny jsou systémy VTK [VTK], IRIS Explorer [IRIS], OpenDX [ODX], LabView [LVIEW] a VisiQuest [VQST].

2.1 MVE-2[1]

MVE-2 je modulární prostředí založené na toku dat. Poskytuje univerzální, snadno pochopitelná rozhraní pro tvorbu modulů i datových struktur, které jsou mezi moduly předávány či sdíleny. Součástí systému je běhové prostředí s vlastnostmi, které poskytují dobré vyjadřovací schopnosti při propojování modulů. Jsou jimi možnost cyklického propojení modulů a modulem řízené spouštění části sítě.

K dispozici jsou dva způsoby jak definovat propojení modulů. Uživatelsky příjemnější je použití programu MapEditor. Ten umožňuje graficky, intuitivním způsobem, vytvářet a modifikovat různá propojení modulů. Vytvořená mapa

[1] Informace v této kapitole jsou se svolením autora převzaty a upraveny převážně z manuálů MVE-2 Handbook (beta-2) a MVE-2 Visualization library (alfa-4), které jsou dostupné z [MVE].

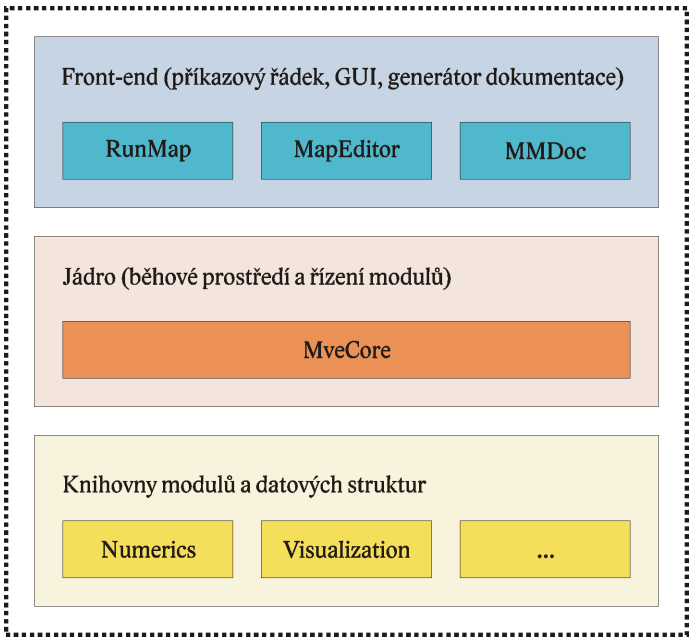
se ukládá do XML souboru, který je čitelný a upravitelný i v běžném textovém editoru. Vlastní spouštění mapy modulů je možné buď v samotném MapEditoru nebo z příkazové řádky programem RunMap.

Oproti předchozí verzi je návrh jádra velmi obecný. Původní systém definoval datové struktury pro vizualizaci ve svém jádře a přidání nové datové množiny vyžadovalo rekompilaci celého systému. Nové prostředí naproti tomu neposkytuje ve svém jádře žádné datové struktury. Místo nich je definováno rozhraní, jež je nutno dodržet, aby datovou množinu bylo možné sdílet mezi moduly. Každá knihovna modulů může definovat vlastní datové struktury. Jádro se tak stalo, z prostředí použitelného pouze pro vizualizaci, obecným modulárním prostředím využitelným v libovolné aplikační oblasti. Záleží jen na implementovaných datových množinách a modulech, které nad těmito množinami pracují.

Mají-li vznikat navzájem kompatibilní moduly, musí být nejprve definováno, s jakými datovými množinami budou tyto moduly pracovat. Kdyby byl každý autor modulu nucen vytvořit si vlastní datovou množinu, moduly by se staly navzájem nekompatibilními a modularita prostředí by ztratila smysl. Proto je spolu s jádrem systému dodávána knihovna modulů Visualization a Numerics. Knihovna Visualization obsahuje množinu datových struktur, které by měly vyhovovat širokému spektru aplikací z počítačové grafiky a vizualizace dat. Knihovna Numerics je kompilací základních číselných datových typů a modulů pro jednoduché matematické operace nad nimi.

2.1.1 Struktura MVE-2

Struktura projektu MVE-2 je znázorněna na (Obrázek 2.1-1).



Obrázek 2.1-1: Struktura MVE-2. Každý blok představuje jednu .NET assembly

Jádro poskytuje rodičovské třídy pro moduly, datové struktury a řídící systém spouštění modulů. Knihovny modulů obsahují samotné moduly a datové struktury. Ty jsou závislé na jádru a mohou být závislé na sobě navzájem.

Front-end představuje část, díky které mohou používat MVE-2 i uživatelé neprogramátoři. RunMap umožňuje spouštět mapu modulů uloženou v XML souboru z příkazového řádku. MapEditor je grafické uživatelské prostředí pro editaci a spouštění map modulů. MMDoc je nástrojem pro automatické generování dokumentace modulů a datových struktur.

2.1.2 Spouštěcí mechanismus

Mapa modulů je grafem. Skládá se z množiny modulů (uzly) a orientovaných spojení mezi moduly (hrany). Modul, ze kterého hrany vychází a žádné do něj nevedou, je modulem zdrojovým. Modul, do kterého vedou hrany a žádné z něho nevychází, je modulem terminálním. Modul, jehož všechny vycházející hrany vedou z portu označeném jako *non-evoke-update*, je modul terminální. Modul bez spojení je považován za modul terminální. Všechny ostatní moduly jsou chápány jako filtry.

Při spuštění mapy je třeba korektně spustit terminální moduly. Korektnost spočívá v připravení dat, která mají být přivedena na vstup terminálního modulu před jeho vlastním spuštěním. Musí být tedy spuštěny všechny moduly, ze kterých vedou spojení do terminálního modulu. Tím vzniká rekurzivní podmínka spuštění mapy modulů až ke zdrojovému modulu.

Místem propojení mezi moduly jsou vstupní a výstupní porty. Port nese informaci o datovém typu, který akceptuje (vstupní port) nebo který poskytuje (výstupní port). Lze tak provést kontrolu kompatibility spojení.

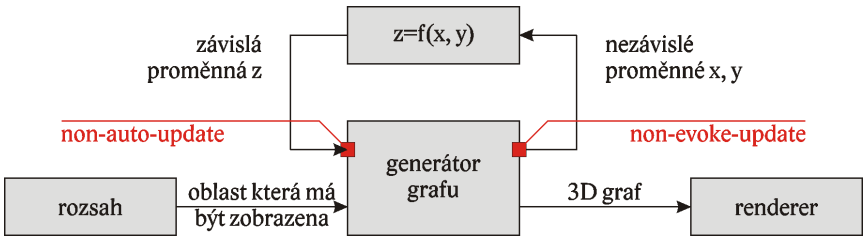
Mapa spojení modulů může obsahovat cykly, je však nutné dodržet jeden z povolených způsobů jejich vytvoření. Prvním korektním způsobem vytvoření cyklu je použití zdržovacího modulu. Ten má dvojici vstupních portů. Na první je přivedena inicializační hodnota, v dalších krocích je v cyklu vypočtená hodnota přebírána z portu druhého. Výstupem jsou data z N-1 kroku. Druhou možností, jak se dá vytvořit korektní cyklus, je místo zdržovacího modulu použít modul s výstupním portem se speciální vlastností *non-evoke-update*. Ta bude vysvětlena níže.

Po odstranění spojení, která končí ve vstupních portech zdržovacích modulů či začínají v *non-evoke-update* výstupních portech, se graf musí stát stromem. Kontrola na korektní použití cyklů a zdržovacích modulů je prováděna před spuštěním mapy a uživatel je na případnou nekonzistenci upozorněn.

Vstupní porty mohou mít u některých modulů speciální příznak *non-auto-update*. Port s tímto příznakem se explicitně vzdává přípravy vstupních dat před vlastním spuštěním modulu. Data jsou připravena až v okamžiku, kdy o ně modul za běhu explicitně požádá. Požadavek na přípravu dat může učinit libovolněkrát v průběhu jednoho běhu. Modul tak získává kontrolu nad částí mapy k němu připojené.

Dalším možným příznakem je *non-evoke-update*. Do značné míry se jedná o analogii k variantě *non-auto-update* portu vstupního. Pokud má port nastaven příznak *non-evoke-update* na *true*, pak požadavek na data z tohoto portu nezpůsobí spuštění modulu. Modul přebírá zodpovědnost za to, že data na takovém portu budou vždy dostupná.

Umožnit modulu řídit běh celé větve bylo hlavním důvodem zavedení non-*evoke-update* portu. Modul může na takový výstupní port vystavit nějakou hodnotu a poté způsobit aktualizaci části mapy, která vychází z jeho *non-auto-update* portu a končí v jeho *non-evoke-update portu*.



Obrázek 2.1-2: Generátor grafu, příklad využití *non-auto-update* a *non-evoke-update* portů

Použití je patrné z obrázku (Obrázek 2.1-2). Cílem mapy je nakreslit graf funkce počítané modulem $z = f(x, y)$ v oblasti dané modulem *rozsah*. Celý graf funkce je vykreslen během jednoho spuštění celé sítě. Přitom modul $z = f(x, y)$ může běžet libovolněkrát. Modul generátor grafu vystaví na svém výstupním *non-evoke-update* portu nezávislé proměnné *x, y* a pak požádá přes vstupní *non-auto-update* port o aktualizaci větve. Tím dojde ke spuštění připojeného modulu a výpočtu funkční hodnoty, která je následně k dispozici modulu generátor grafu. Takto si může modul napočítat funkční hodnoty v libovolných bodech a z nich vytvořit 3D graf. Ten po skončení výpočtu vystaví na normální výstupní port, ze kterého si jej renderer přečte a zobrazí.

Díky různým typům portů dostávají tvůrci modulů poměrně silný vyjadřovací prostředek. Pro autora mapy modulů je řízení běhu sítě transparentní. Můžeme říci, že část sítě připojená k modulu přes *not-auto-update* port je podsítí tohoto modulu a stává se součástí běhu téhož modulu, který tak přebírá zodpovědnost za to, kdy a kolikrát bude spuštěna.

Je zřejmé, že podsíť může opět obsahovat modul s *non-auto-update* portem, a tak může vzniknout podsíť podsítě. Proto je zaveden pojem úroveň sítě. Úrovně nula jsou všechny terminální moduly. Od nich se určuje úroveň zbytku mapy.

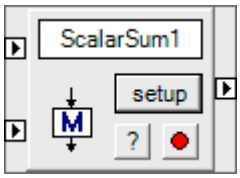
Existují pravidla, která zaručují korektní chování mapy:

- Zdrojový modul může být nižší úrovně než modul cílový pouze v případě, že vstupní port je typu non-auto-update.
- Zdrojový modul může být vyšší úrovně než modul cílový pouze v případě, že cílový modul je DelayModule nebo zdrojový port je non-evoke-update.
- Ostatní propojení musí vést z a do stejné úrovně.

Tato pravidla jsou kontrolována před spuštěním mapy a v případě nekonzistence je uživatel náležitě upozorněn. Úroveň sítě je možné v aktuální verzi MVE zobrazit.

2.1.3 Modul

Modul (Obrázek 2.1-3) představuje objekt s výkonným kódem. Po vložení do mapy generuje, filtruje, ukládá, kontroluje či zobrazuje data. V MVE-2 jsou moduly organizovány do .NET knihoven[1] a standardně uloženy ve složce *.\libs*. MapEditor zobrazuje načtené moduly ve stromové struktuře podle jmenného prostoru, v němž jsou uloženy.



Obrázek 2.1-3: Modul pro součet dvou čísel

Vytvoření modulu

Z pohledu programátora je modul třída zděděná od rodičovské třídy *Zcu.Mve.Core.Module*. Nutně musí být přetížena pouze metoda *Execute()*. Ta je volána při spuštění modulu a je předurčena k implementaci vlastní činnosti modulu.

Následuje kód (Kód 2.1-1) jednoduchého modulu HelloWorld[2], jehož cílem je na konzoli vypsat známou hlášku.

```
1   namespace Test
2   {
3     public class HelloWorld : Zcu.Mve.Core.Module
4     {
5       public override void Execute()
6       {
7         System.Console.WriteLine("Hello world!!!");
8       }
9     }
10  }
```

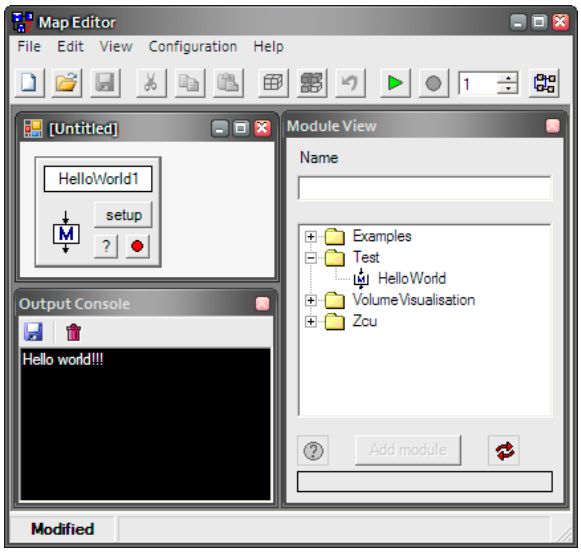
Kód 2.1-1: Modul “Hello world”.

Kód zkompilovaný do DLL knihovny je třeba zkopírovat do adresáře *.\libs* prostředí MVE-2. Po spuštění MapEditoru (Obrázek 2.1-4) je možné modul vybrat v okně ModuleView a vložit do mapy. Spuštění mapy způsobí vypsání očekávané hlášky do okna Output Console.

Modul HelloWorld nemá žádné porty. Ty se obvykle vytváří v konstruktoru třídy voláním metody *AddInputPort()*. Konstruktor je volán při načtení mapy do operační paměti. Instance tedy existuje přes nedefinované množství spuštění mapy.

[1] tzv. assembly
[2] První známý výskyt společného použití slov "hello" a "world" v počítačové literatuře byl roku 1973 v Úvodu do jazyka B od B. Kernighana, viz.[TPLB].

Autor modulu má k dispozici celou řadu dalších metod pro přepsání i volání, viz. [MVE2]. Třída *Zcu.Mve.Core.Module* také definuje několik událostí. Využitím těchto prostředků je možné tvořit moduly s pokročilými vlastnostmi.



Obrázek 2.1-4: MapEditor s mapou, vloženým a jednou spuštěným modulem HelloWorld

Dialog nastavení modulu

Pro každý modul umístěný do mapy je možné tlačítkem *setup* vyvolat dialog nastavení modulu. Jsou v zásadě dvě možnosti, jak se dá toto dialogové okno vytvořit.

První je oddědit potomka třídy *UserControl*, třídu *ModuleSetup*, a ta umožňuje vkládat běžný WinForms obsah. Ve třídě *ModuleSetup* je potřeba přetížit metody *OnOK()* a *OnStorno()*, které jsou volány jako reakce na akce uživatele.

Druhou možností je využít implicitního chování metody *InvokeSetup()*. Ta vrací automaticky generovaný PropertyGrid, který umožňuje pro jednoduché datové typy měnit hodnoty vlastností modulu. Zamezit zobrazení vlastnosti v PropertyGridu lze atributem *[Browsable()]*. Pomocí atributů *[Description()]* a *[CategoryAttibute()]* je možné nastavit popis a kategorii vlastnosti.

PropertyGrid umožňuje editovat i vlastní datové typy. Podmínkou je, že takový datový typ má pomocí atributů *[Editor()]* a *[TypeConverter()]* přiřazeny třídy, které se starají o způsob editace a konverzi[1]. O tvorbě takových tříd a použití PropertyGridu se lze dočíst v článku zveřejněném v MSDN [MPGC].

Ukládání a načítání nastavení modulu

Mapa modulů se ukládá do souboru XML. Součástí tohoto souboru může být i nastavení jednotlivých modulů v uzlu *<config />*. Chceme-li mít ukládání a načítaní konfigurace modulu pod kontrolou, je třeba přetížit metody *WriteConfig()* a *ReadConfig()*.

[1] Použitím nástroje Reflector [REFL] se lze podívat, jak jsou tyto třídy řešeny pro systémové datové typy. Například datový typ System.Drawing.Color má jako editor třídu System.Drawing.Design.ColorEditor a jako konvertor třídu System.Drawing.ColorConverter.

Mějme modul s vlastností *Greeting* ve které je uchován pozdrav[1]. Kód (Kód 2.1-2) ukazuje uložení vlastnosti *Greeting*. Metoda *WriteConfig()* získá jako parametr referenci na uzel *<config />* aktuálního modulu. V něm je vytvořen uzel *<Properties />*[2] a v něm uzel *<Greeting />*. Uzlu *<Greeting />* je přiřazen atribut typu vlastnosti *Greeting* a jako text je uložen pozdrav.

```
1   public override void WriteConfig(System.Xml.XmlElement config)
2   {
3     XmlElement xmlProperties = config.OwnerDocument.CreateElement("Properties");
4
5     XmlElement xmlGreeting = config.OwnerDocument.CreateElement("Greeting");
6     xmlGreeting.SetAttribute("type", Greeting.GetType().FullName);
7     xmlGreeting.InnerText = Greeting;
8     xmlProperties.AppendChild(xmlGreeting);
9
10    config.AppendChild(xmlProperties);
11  }
```

Kód 2.1-2: Přetížená metoda *WriteConfig()* pro uložení nastavení modulu

Kód (Kód 2.1-3) obsahuje část souboru XML s uloženým pozdravem, vlastností *Greeting* modulu.

```
1   <config>
2     <Properties>
3       <Greeting type="System.String">Hello world!!!</Greeting>
4     </Properties>
5   </config>
```

Kód 2.1-3: Část XML souboru s uloženým nastavením modulu

Při načítání mapy modulů jsou postupně pro jednotlivé moduly volány jejich metody *ReadConfig()* (Kód 2.1-4). Těm je předána reference na odpovídající uzel s nastavením. V našem případě by tento uzel měl obsahovat uzel *<Properties />*. Poduzlem uzlu *<Properties />* je uzel *<Greeting />*. Z atributu uzlu *<Greeting />* se lze dozvědět, že uložená hodnota je typu *string*. Vnitřek uzlu pak skrývá text pozdravu.

Větev *default* ošetřuje případ, že by uzel *<Properties />* obsahoval ještě nějaký další poduzel. Modul zkompilovaný v ladícím módu by při načtení svých nastavení vypsal do Debug okna MapEditoru chybovou hlášku. Tento přístup je vhodný při vývoji modulu, kde přibývají nové vlastnosti, a ukládání necháváme na vnitřním mechanismu. Ten nesystémové datové typy ukládá metodou *ToString()*. Nestane se proto, že by bylo opomenuto načtení nějaké přidané vlastnosti.

[1] Například "Hello world!!!"
[2] Uzel Properties je vytvořen z důvodu zachování konzistence způsobem, kterým je ukládán modul automaticky, tj. bez přetížení metody WriteConfig(). Spoléhat se na automatické ukládání nastavení modulu lze ovšem jen v případě, že modul obsahuje pouze vlastnosti systémových datových typů.

```
12  public override void ReadConfig(XmlElement config)
13  {
14    foreach (XmlElement xeConfig in config.ChildNodes)
15    {
16      if (xeConfig.Name == "Properties")
17      {
18        foreach (XmlElement xeProperties in xeConfig.ChildNodes)
19        {
20          switch (xeProperties.Name)
21          {
22            case "Greeting":
23              Greeting = xeProperties.InnerText;
24              break;
25            default:
26              Debug.WriteLine("Unreaded value:"
27              + xeProperties.Name + " - " +      xeProperties.InnerText);
28              break;
29          } // switch
30        } // foreach
31      } // if
32    } // foreach
33  }
```

Kód 2.1-4: Metoda *ReadConfig()* pro načtení nastavení modulu

Komentování modulu

Komentáře a informace o modulu je vhodné zapisovat přímo do zdrojového kódu modulu. Mohou zde být uloženy ve formě XML komentářů nebo atributů. Atributy se použijí k vygenerování uživatelské dokumentace nástrojem MMDoc a hlavně za běhu MapEditoru pro zobrazení rychlé nápovědy, detailních informacích o modulu, výpisu seznamu modulů atp.

XML komentáře se používají standardním způsobem. Jsou čtyři typy atributů. *[ModuleInfoAttribute()]* a *[PortInfoAttribute()]* slouží k okomentování modulu, *[DescriptionAttribute()]* a *[BrowsableAttibute()]* jsou standardními .NET atributy vlastností. Tyto atributy popisují jednotlivé nastavitelné položky modulu v dialogovém okně s použitím PropertyGridu.

ModuleInfoAttribute

Atribut (Kód 2.1-5) obsahuje informace o modulu a lze vložit pouze jeden před definici třídy.

```
1   [ModuleInfo("Jan Kaiser", "Hello world module.", IconName =
    "HelloWorld.ico", Assemblied = "2006-04-04", Category = "Test")]
```

Kód 2.1-5: Příklad použití atributu *[ModuleInfo()]*

Povinnými parametry jsou autor modulu a popis funkce modulu, volitelnými parametry je jméno ikony ve zkompilované knihovně, datum sestavení modulu, a kategorie[1].

[1] Parametr Category by měl do budoucna sloužit k zařazení do skupiny modulů v seznamu modulů v MapEditoru. Současná verze beta-3 k tomu používá plně kvalifikované jméno modulu.

PortInfoAttribute

[PortInfoAttribute()] (Kód 2.1-6) obsahuje informace o jednom portu modulu.

```
2   [PortInfo("Output", "String with greeting.")]
```

Kód 2.1-6: Příklad použití atributu *[PortInfo()]*

Povinnými parametry jsou jméno portu, které musí být v rámci modulu unikátní a shodné se jménem použitým při přidání portu v kódu modulu, a popis funkce portu.

2.1.4 Datový typ

Z pohledu uživatele je datový typ třída nebo struktura, jejíž instance nese data modifikovaná či předávaná mezi moduly. Spolu s moduly je uložena v .NET knihovně.

Vytvoření datového typu

Vytvoření nového datového typu by měl předcházet důkladný průzkum, zda již neexistuje nějaký podobný, který by bylo možné použít a pro který již existuje množina použitelných modulů. Vznik duplicitních datových typů by vedl k existenci navzájem nekompatibilních modulů.

Každý datový objekt, který má být sdílen mezi moduly, musí být potomkem třídy *Zcu.Mve.Core.DataObject*. Oproti tvorbě modulu je třeba přetížit více metod.

Datový objekt obvykle nese nějakou hodnotu, kterou je zvykem inicializovat v konstruktoru.

Metoda *ToString()* umožňuje převést obsah datového typu na čitelný textový řetězec. Příkladem budiž modul ConsolePrinter, který volá metodu *ToString()* datového objektu přivedeného na jeho vstup a získaný textový řetězec vypisuje na konzoli.

Úkolem metody *DeepCopy()* je alokovat nový paměťový prostor a provést zkopírování veškerých dat. Tato metoda je používána například pro kopírování obsahu vstupu DelayModulu, aby v dalším kroku mohla poskytnout data, jež měl na vstupu v kroku minulém.

U složitějších datových typů může docházet k nekonzistencím. Například geometrický objekt, jehož jeden z indexů do pole bodů je větší než počet bodů, či při počítaní s reálnými čísly překročení očekávané odchylky epsilon. V takových případech má metoda *CheckConsistence()* vrátit hodnotu *false*. Vhodné je také uživatele MVE-2 upozornit na možné problémy při dalším zpracování a příčinu nekonzistence. Nejjednodušším způsobem je vypsání chybové hlášky do konzole metodou *Console.WriteLine()*, případně do ladící konzole metodou *Debut.WriteLine()*.

Ukládání a načítání datového typu

Každý správně napsaný datový typ musí mít přetíženy metody *ReadData()* a *WriteData()*, které zprostředkovávají zápis a čtení dat obsažených v datovém objektu do a z XML souboru.

Na tvůrci datového typu je, jakou reprezentaci pro uložení dat zvolí. Může být ukládáno pole takových objektů a je tedy vhodné zvážit paměťovou náročnost a zároveň čitelnost uložených dat, bude-li někdo jiný mít potřebu je ručně měnit.

Na rozdíl od ukládání a čtení dat modulu je ukládání datového objektu do XML textového souboru řešeno sekvenčně. To s sebou přináší urychlení a menší paměťovou náročnost na straně jedné, na straně druhé pak požadavek na větší míru obezřetnosti, především při čtení.

Při ukládání je metodě *WriteData()* (Kód 2.1-7) předána reference na *XmlTextWriter* aktuálního souboru. Sekvenčně se zapisují vlastnosti objektu. Na řádce (Kód 2.1-7: 592-595) je uložen typ světla převodem do řetězcové reprezentace. Na řádce (Kód 2.1-7: 597-600) je vidět uložení složitějšího datového typu s vlastní metodou *WriteData()*.

```
588. public override void WriteData(XmlTextWriter xmlTextWriter)
589. {
590.   xmlTextWriter.WriteStartElement("light");
591.
592.   xmlTextWriter.WriteStartElement("type");
593.   xmlTextWriter.WriteAttributeString("type", this.type.GetType().FullName);
594.   xmlTextWriter.WriteString(this.type.ToString());
595.   xmlTextWriter.WriteEndElement();
596.
597.   xmlTextWriter.WriteStartElement("ambient");
598.   xmlTextWriter.WriteAttributeString("type", this.ambient.GetType().FullName);
599.   this.ambient.WriteData(xmlTextWriter);
600.   xmlTextWriter.WriteEndElement();
601.
657.   xmlTextWriter.WriteStartElement("enabled");
658.   xmlTextWriter.WriteAttributeString("type", this.enabled.GetType().FullName);
659.   xmlTextWriter.WriteString(this.enabled.ToString());
660.   xmlTextWriter.WriteEndElement();
661.
662.   xmlTextWriter.WriteEndElement();
663. }
```

Kód 2.1-7: Část metody *WriteData()* datového typu *Zcu.Mve.UnstrGridRendering.Light*

Navíc byl vytvořen obalující element (Kód 2.1-7: 590, 662) se jménem dle datového typu. Důvodem bylo chování při ukládání pole datových objektů. Standardně jsou datové objekty při ukládání obaleny elementem *<dataobject />*. Z důvodu snížení velikosti výsledného souboru je však při ukládání pole tento element jednoduše pojmenován *<i />*.

Kód (Kód 2.1-8) metody *ReadData()* obsluhuje načtení vlastností datového objektu. Jedná se o sekvenční čtení, není tedy dovoleno, aby metoda *ReadData()* četla data, která jí nenáleží. Díky obalujícímu elementu je možné identifikovat konec vlastních dat (Kód 2.1-8: 571-575) a čtení předat dál.

Řádek (Kód 2.1-8: 519-521) ukazuje načtení typu světla. Pro ulehčení převodu byla vytvořena statická třída *FromString* která obsahuje metody pro převod z řetězcové reprezentace.

Na řádcích (Kód 2.1-8: 522-525) je načtení složitějšího datového typu. Podle typu je nejprve vytvořena jeho instance. Metoda *ReadData()* vzniklého datového objektu je zavolána s referencí na *XmlTextReader*, aby si objekt načetl své vlastnosti.

Na řádku (Kód 2.1-8: 578) si lze všimnout metody *UpdateModificationTime()*. Jejím úkolem je aktualizovat čas vytvoření objektu. Zavedení této metody bude zdůvodněno později v kapitole o úpravách jádra MVE-2.

```
510. public override void ReadData(XmlTextReader xmlTextReader)
511. {
512.   Debug.WriteLine("Light reading...");
513.   while (xmlTextReader.Read())
514.   {
515.     if (xmlTextReader.NodeType == XmlNodeType.Element)
516.     {
517.       switch (xmlTextReader.Name)
518.       {
519.         case "type":
520.           type = FromString.GetLightType(xmlTextReader.ReadString(),
                        LightType.Directional);

521.           break;
522.         case "ambient":
523.           this.ambient = (ColorRGBA) Activator.CreateInstance(Globals.FindType(
                          xmlTextReader.GetAttribute("type").Trim()));

524.           this.ambient.ReadData(xmlTextReader);
525.           break;
…
563.         case "enabled":
564.           this.enabled = FromString.GetBool(xmlTextReader.ReadString(), true);
565.           break;
566.         default:
567.           break;
568.       } // switch
569.     } //if
570.
571.     if (xmlTextReader.NodeType == XmlNodeType.EndElement)
572.     {
573.       if (xmlTextReader.Name == "light")
574.         break;
575.     }
576.   } // while
577.
578.   UpdateModificationTime();
579.   Debug.WriteLine(" OK");
580.
581.   Console.WriteLine(this.ToString());
582. }
```

Kód 2.1-8: Část metody *ReadData()* datového typu *Zcu.Mve.UnstrGridRendering.Light*

Komentování datového typu

Podobně jako u modulu mohou být informace o datovém typu uloženy ve formě XML komentářů nebo atributů. XML komentáře se používají běžným způsobem. MVE-2 definuje jeden speciální atribut pro datový typ.

DataObjectAttribute

[DataObjectAttribute()] (Kód 2.1-9) obsahuje informace o datové struktuře. Smí být použit pouze jeden před definicí třídy.

```
10. [DataObject("Defines a set of lighting properties.")]
11. public class Light : DataObject
```

Kód 2.1-9: Použití atributu *[DataObjectAttribute()]* u datového typu *Zcu.Mve.UnstrGridRendering.Light*

Má-li být vytvořený datový typ zobrazitelný v PropertyGridu, musí mu být pomocí atributů *[Editor()]* a *[TypeConverter()]* přiřazeny třídy, starající se o editaci a konverzi (Kód 2.1-10).

```
11. [Editor(typeof(ColorRGBAEditor), typeof(System.Drawing.Design.UITypeEditor))]
12. [TypeConverter(typeof(ColorRGBAConverter))]
13. public struct ColorRGBA : IAttribute
```

Kód 2.1-10: Použití atributů *[Editor()]* a *[TypeConverter()]* u datového typu *Zcu.Mve.Visualization.ColorRGBA*

2.1.5 Knihovna Visualization

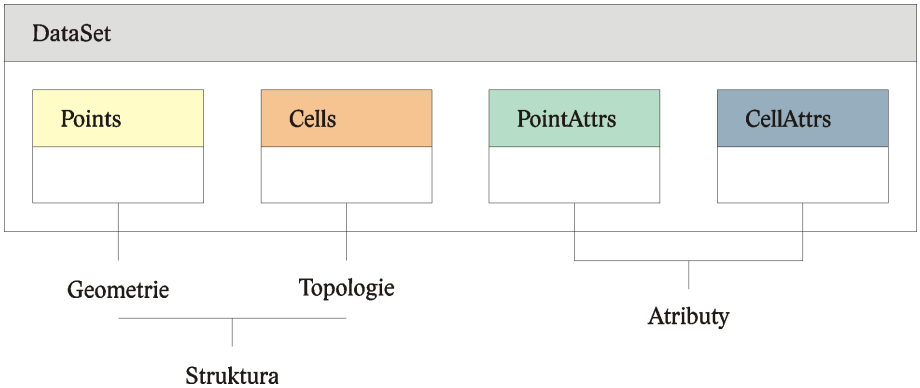
Knihovna Visualization nabízí moduly a datové struktury pro zpracování obrazu a vizualizaci dat.

Datový model

Úkolem počítačové grafiky a vizualizace dat je srozumitelně reprezentovat velké množství různorodých dat. Aby to bylo možné, je třeba mít pro taková data navrhnuté efektivní datové struktury. Musí být kompaktní, výkonné a jednoduché. Při návrhu knihovny, která není určena jen pro jedno konkrétní nasazení, pak do hry vstupuje ještě požadavek obecnosti. Inspirací knihovně Visualization byl datový model použitý ve VTK firmy Kitware [VTK].

DataSet

Dataset (Obrázek 2.1-5) je pojmenování abstraktního datového typu knihovny Visualization určeného pro vizualizaci dat. Je tvořen strukturou a atributy.

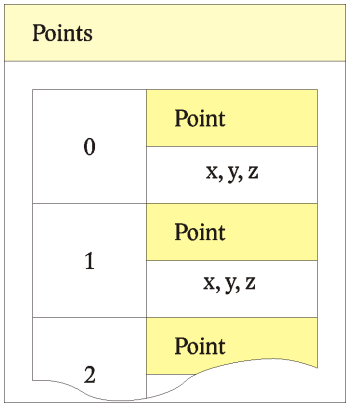


Obrázek 2.1-5: Datový objekt *DataSet*

Struktura má určitou geometrii a topologii. Geometrie je určena pozicí v prostoru, tedy body. Topologie je množina vlastností, které se nemění geometrickými transformacemi, představují ji buňky. Atributy jsou dodatečné vlastnosti svázané s body nebo buňkami. Může se jednat o teplotu, tlak, barvu, atd.

Geometrická informace

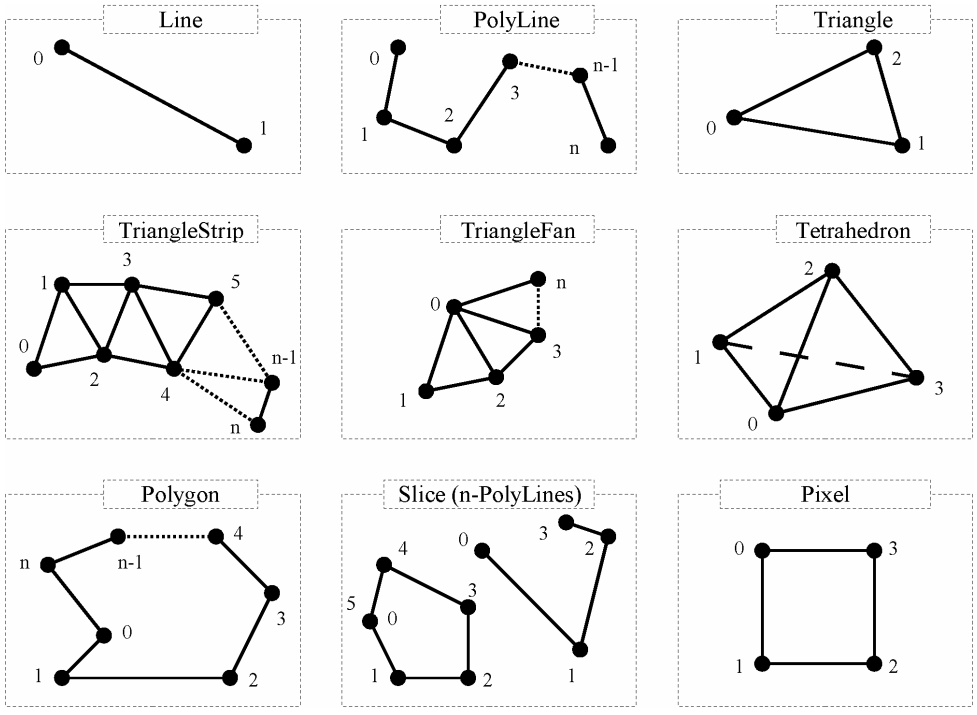
Geometrickou informací je specifikace pozice v prostoru. Jedná se tedy o nějaký bod v prostoru libovolné dimenze. Dataset takové body ukládá do pole *Points*.



Obrázek 2.1-6: Geometrie *DataSetu*, body *Points*

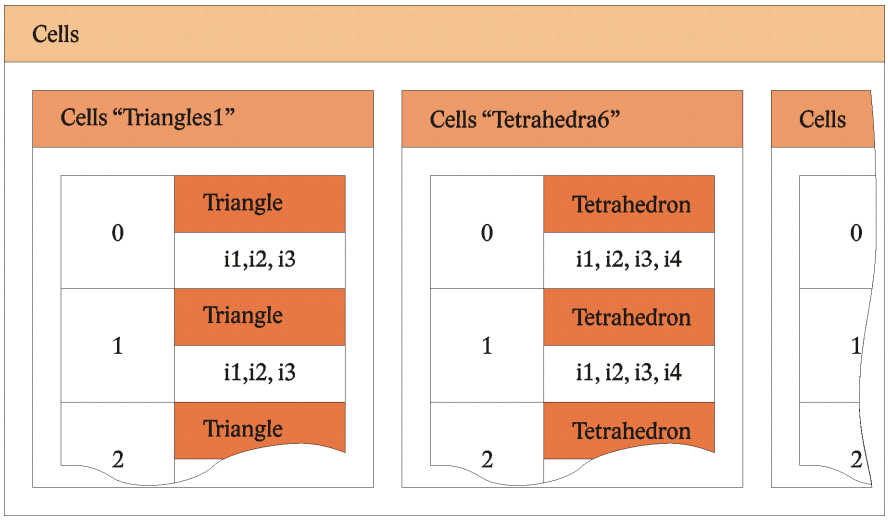
Topologická informace

Nositelem topologické informace je buňka (Cell). Základní soubor nejpoužívanějších buněk je uveden na (Obrázek 2.1-7). Obsahem buňky jsou indexy do pole bodů, topologie buňky je implicitně známa.



Obrázek 2.1-7: Základní typy buněk

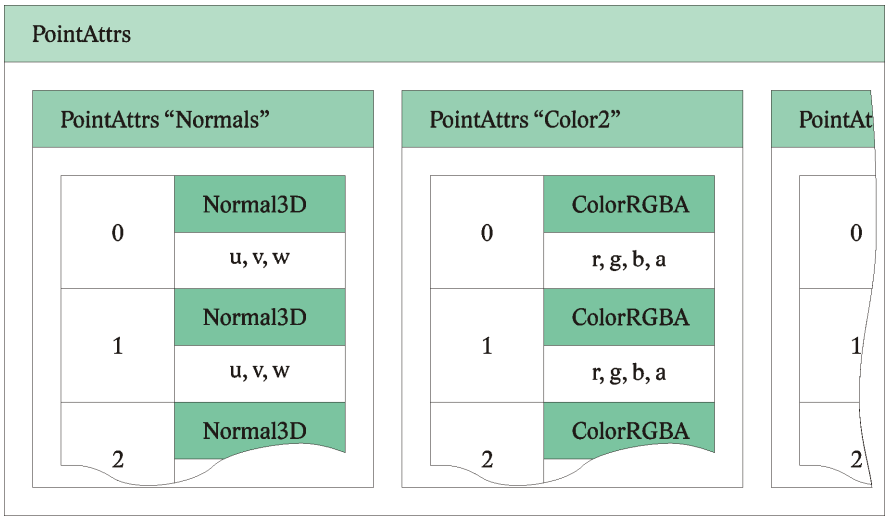
V datasetu lze buňky stejného typu sdružovat do pojmenovaných skupin. Na (Obrázek 2.1-8) je například skupina buněk *Cells* typu *Triangle* pojmenovaná “Triangles1” a skupina buněk typu *Tetrahedron* pojmenovaná “Tetrahedra6”.



Obrázek 2.1-8: Topologie *DataSetu*, buňky *Cells*

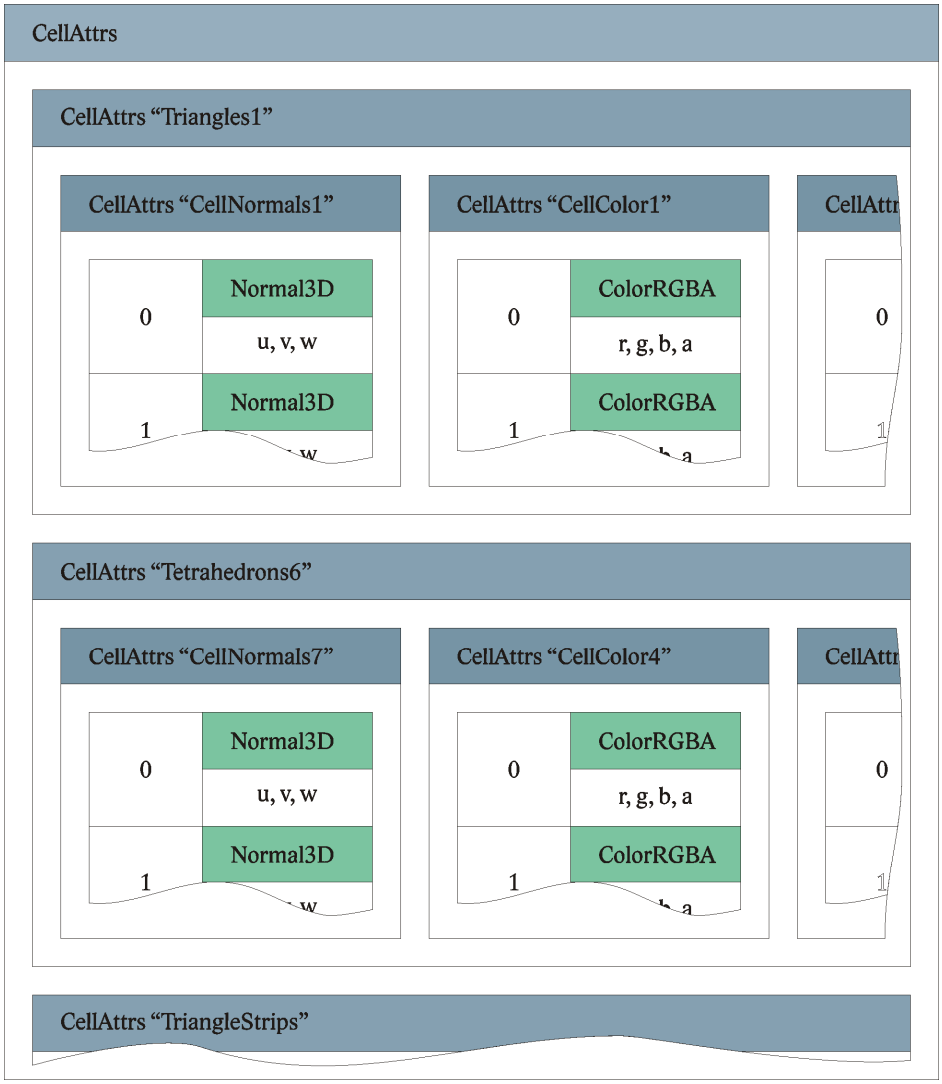
Asociovaná data

Atributy jsou informace asociované s body nebo buňkami. Ve struktuře datasetu mohou mít body více atributů. Na (Obrázek 2.1-9) je každému bodu přiřazen atribut normálového vektoru a barvy. Index atributu odpovídá indexu bodu.



Obrázek 2.1-9: Atributy bodu ve struktuře *DataSet*

Podobně jako body i buňky mohou mít více atributů. Na rozdíl od bodů, skupin buněk může být více. Atribut proto musí být adresován třemi parametry: skupina buněk, jméno atributu a index atributu odpovídající indexu buňky, viz. (Obrázek 2.1-10).



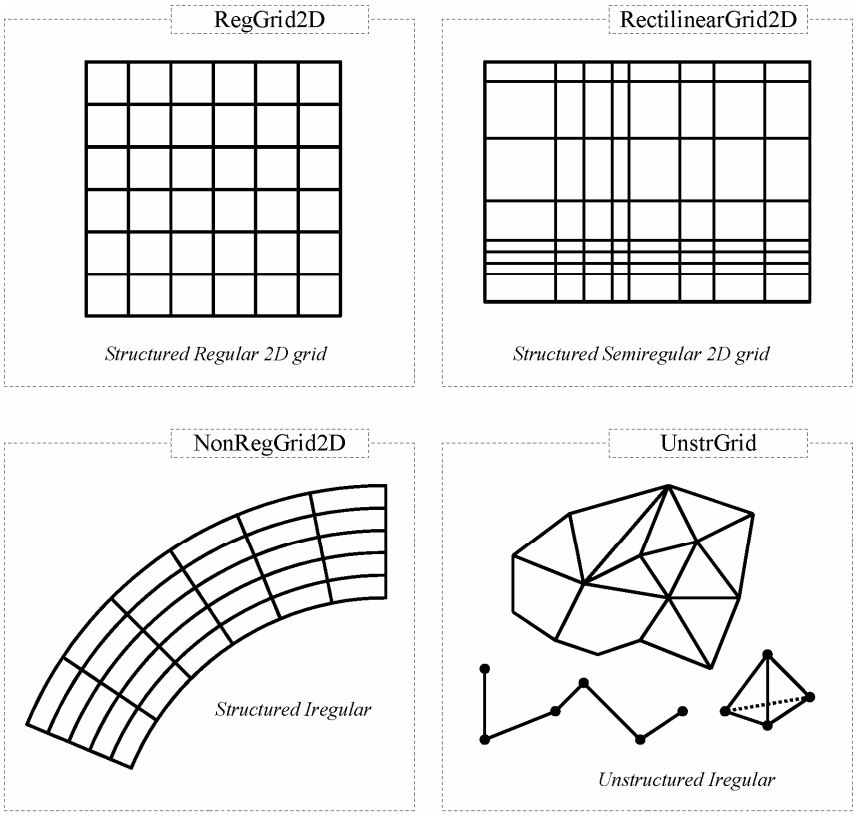
Obrázek 2.1-10: Atributy buňky *Cell* ve struktuře *DataSet*

Typy datových množin

Jak bylo uvedeno, datová množina *DataSet* se skládá z organizační struktury a přidružených atributů. Struktura má topologické a geometrické vlastnosti. Navíc lze datové množiny rozdělit na strukturované a nestrukturované.

Topologie strukturované datové množiny je dána implicitně, což má za následek úsporu paměti. Naproti tomu implicitní popis u nestrukturované topologie není znám a konektivita tedy musí být vyjádřena explicitně. Nestrukturované datové množiny jsou obecnější než strukturované, ale mají daleko větší paměťové požadavky. Na (Obrázek 2.1-11) je ukázáno několik základních datových množin.

Důležitou vlastností datových množin je také jejich pravidelnost. Pokud lze geometrii vyjádřit implicitně, pak hovoříme o datech pravidelných či částečně pravidelných. Pokud jsme nuceni vyjadřovat explicitně pozici každého bodu, pak hovoříme o datech nepravidelných.



Obrázek 2.1-11: Typy datových množin

Implementované datové struktury

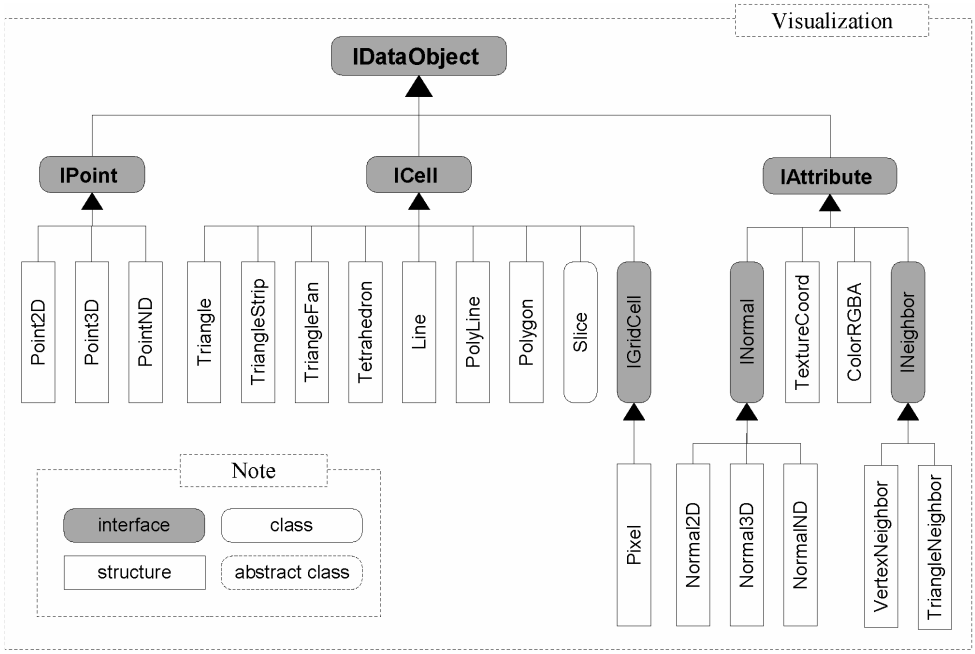
Následující text popisuje implementaci datových struktur v knihovně Visualization.

Základní datové typy

Aby bylo možné instance datových typů posílat mezi moduly, musí implementovat rozhraní *IDataObject*. Hierarchie datových typů je docílena mechanismem dědičnosti rozhraní (Obrázek 2.1-12). Elementární datové typy jsou z důvodu efektivity strukturami, tedy hodnotovými datovými typy, jejichž zpracování znamená menší režii než zpracování objektů. Struktury implementují rozhraní podle toho, do jaké kategorie patří. Body implementují rozhraní *IPoint*, buňky rozhraní *ICell* a atributy rozhraní *IAttribute*.

Knihovně Visualization je velmi blízká knihovna Numerics. Při vizualizaci dat z ní najdou uplatnění především datové typy *Scalar* a *Vector*, *Vector2D*, *Vector3D* a *VectorND* implementují rozhraní *IVector*.

Konverze umožňují přetypování jednoho datového typu na druhý. Tam, kde nedochází ke ztrátě dat, tzv. rozšiřující konverze, je přetypování provedeno implicitně. V opačném případě, tzv. zužující konverze, je třeba explicitně uvést, na který datový typ přetypováváme.

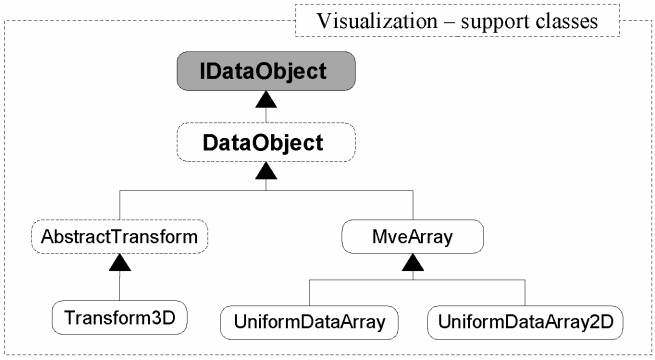


Obrázek 2.1-12: Základní datové typy knihovny Visualization

Pomocné třídy

Kromě datových typů předurčených pro efektivní uložení dat a jejich následnou manipulaci či vizualizaci obsahuje knihovna Visualization i několik pomocných tříd (Obrázek 2.1-13).

Transform3D je obecnou transformací objektu reprezentovaném v trojrozměrném prostoru. Třídy *UniformDataArray* a *UniformDataArray2D* představují jedno a dvourozměrné homogenní statické pole pro seskupení datových objektů stejného typu.



Obrázek 2.1-13: Pomocné třídy v knihovně Visualization

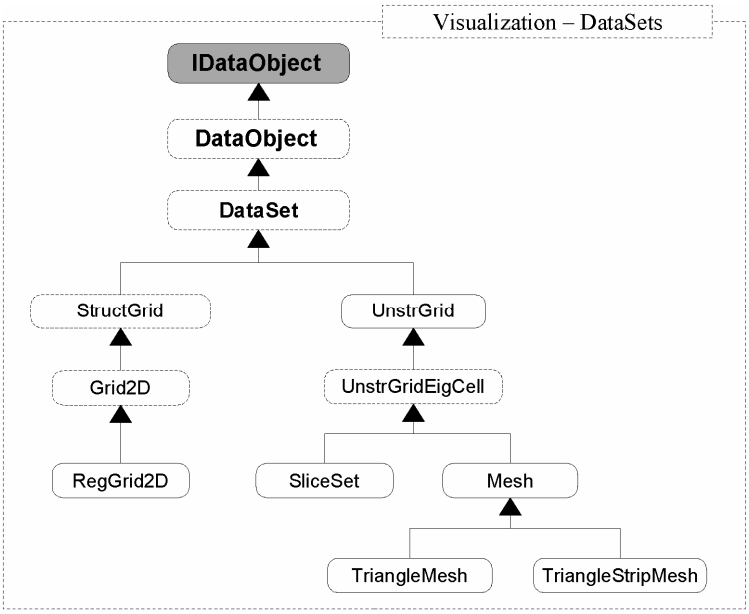
DataSety

Implementované datové množiny odpovídají jejich logickému dělení. *StructGrid* je dataset s pravidelnou mřížkou, *UnstrGrid* je dataset s nepravidelnou mřížkou.

Představitelem datasetu se strukturovanou pravidelnou pravoúhlou mřížkou je *RegGrid2D*. Skládá se 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 bitmapového obrázku.



Obrázek 2.1-14: DataSety knihovny Visualization

UnstrGrid

Moduly vytvořené v rámci této diplomové práce mají být schopny zobrazit data uložená v nejobecnější struktuře datasetu *UnstrGrid*. Ta dokáže pojmout libovolné typy buněk, ovšem za cenu velké paměťové náročnosti. *UnstrGrid* nemá žádnou vnitřní strukturu, data v něm uložená musí být reprezentována explicitně a musí být v paměti fyzicky uložena.

UnstrGrid obsahuje pole bodů *Points*, uniformní pole buněk *Cells*, atributy bodů *PointAttrs* a atributy buněk *CellAttrs*. Pro přístup k nim se používají následující metody a vlastnosti:

- Points – vlastnost pro přístup k poli bodů.
- AddCells(), GetCells(), RemoveCells() – metody pro práci s buňkami.
- AddPointAttrs(), GetPointAttrs(), RemovePointAttrs() – metody pro práci s atributy bodů.
- *AddCellAttrs()*, *GetCellAttrs()*, *RemoveCellAttrs()* – metody pro práci s atributy buněk.

Kód (Kód 2.1-11) ukazuje naplnění struktury *UnstrGrid*. Nejprve je vytvořeno uniformní pole bodů, naplněno a reference na něj přiřazena vlastnosti *Points* (Kód 2.1-11: 3-8).

Obdobně je vytvořeno pole atributů typu *ColorRGBA*. Přidání atributů bodů se provádí metodou *AddPointAttrs()*, atributy jsou souhrnně pojmenovány "Color" (Kód 2.1-11: 10-15). Atributů bodů může být více, podmínkou je jejich různé pojmenování.

Buňky typu *Triangle* jsou přidány na (Kód 2.1-9: 17-20). Atributy těchto buněk jsou normálové vektory typu *Normal3D*, pojmenované "Normals" (Kód 2.1-1: 22-25).

```
1.  UnstrGrid unstrGrid = new UnstrGrid();
2.
3.  UniformDataArray points = new UniformDataArray(typeof(Point3D), 4);
4.  points[0] = new Point3D(-1, 0, 0);
5.  points[1] = new Point3D( 1, 0, 0);
6.  points[2] = new Point3D( 1, 1, 0);
7.  points[3] = new Point3D(-1, 1, 0);
8.  unstrGrid.Points = points;
9.
10. UniformDataArray pointAttrs = new UniformDataArray(typeof(ColorRGBA), 4);
11. pointAttrs[0] = new ColorRGBA(255,   0,   0);
12. pointAttrs[1] = new ColorRGBA(  0, 255,   0);
13. pointAttrs[2] = new ColorRGBA(  0,   0, 255);
14. pointAttrs[3] = new ColorRGBA(255, 255,   0);
15. unstrGrid.AddPointAttrs("Color", pointAttrs);
16.
17. UniformDataArray cells = new UniformDataArray(typeof(Triangle), 2);
18. cells[0] = new Triangle(0, 1, 2);
19. cells[1] = new Triangle(2, 3, 0);
20. unstrGrid.AddCells("Triangles", cells);
21.
22. UniformDataArray cellAttrs = new UniformDataArray(typeof(Normal3D), 2);
23. cellAttrs[0] = new Normal3D(0, 0, 1);
24. cellAttrs[1] = new Normal3D(0, 0,-1);
25. unstrGrid.AddCellAttrs("Triangles", "Normals", cellAttrs);
```

Kód 2.1-11: Ukázka naplnění *DataSetu UnstrGrid*

Z datasetu *UnstrGrid* jsou odděděny datasety *SliceSet*, který smí obsahovat pouze buňky typu *Slice*, a *Mesh*, který může obsahovat pouze buňky typu *Triangle*, *TriangleFan*, *TriangleStrip*, *Line* a *PolyLine*.

Potomkem datasetu *Mesh* jsou *TriangleMesh*, s jedinými povolenými buňkami typu *Triangle*, a *TriangleStripMesh*, kde jediné povolené buňky jsou typu *TriangleStrip*.

2.2 D3DUT[1]

D3DUT je jednoduše přenositelným rozhraním Direct3D for Managed Languages. Umožňuje vytvořit Manged Direct3D aplikaci a spustit ji v prostředí, kde Direct3D není přítomno, případně CLI implementace není úplná. Typickým příkladem budiž neúplná implementace knihovny WinForms.

Zjednodušení přenositelnosti budoucích verzí MVE-2, včetně knihoven na jiné, než .NET CLI implementace bylo hlavním důvodem použití D3DUT. Zároveň je rozhraní D3DUT stále ve vývoji a nasazení v reálném projektu je tak vhodným způsobem ověření funkčnosti a použitelnosti současné verze.

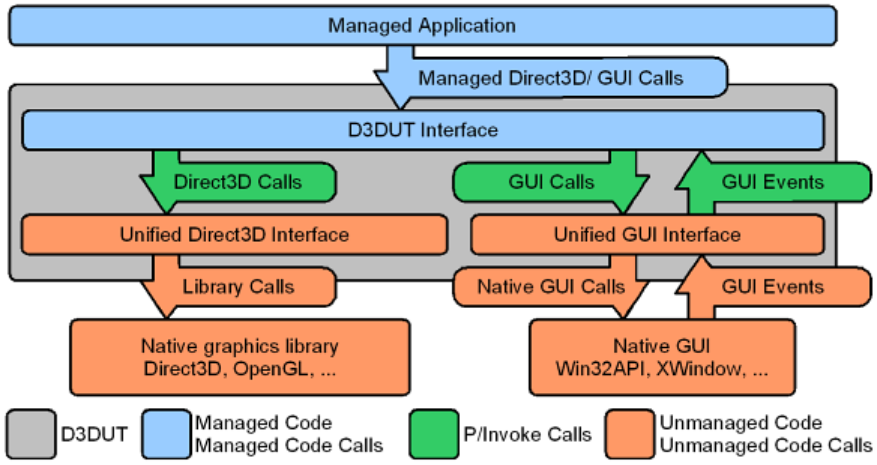
2.2.1 Struktura

Projekt D3DUT se skládá ze dvou významných částí, řízené a neřízené části (Obrázek 2.2-1).

Řízená část běžící v CLI poskytuje uživateli rozhraní pro psaní aplikací, funkcionalitu dodává neřízená část, která má sjednocené rozhraní napříč platformami.

Neřízená část je nativní binární knihovna[2]. Implementuje funkcionalitu jednotného rozhraní možnostmi dané platformy. Na platformě Linux tak využívá rozhraní OpenGL k emulaci funkcionality Direct3D.

Rozhraní pro tvorbu GUI je jednoduché, vyplývá z potřeb designu řízené části. Zahrnuje vytvoření okna, zpracování uživatelských vstupů a vyvolání kontextového menu. Rozhraní pro akcelerovaný 3D výstup je založené na rozhraní neřízeného Direct3D.



Obrázek 2.2-1: : Architektura D3DUT

[1] Tato kapitola čerpá z [D3DUT].
[2] Na platformě Win32 to jsou knihovny *.dll*, na platformě Linux/Unix knihovny *.so*

2.2.2 Použití

Z výpisu zdrojového kódu (Kód 2.1-1) je zřejmá jednoduchost použití D3DUT.

```
1.  using System;
2.  using Zcu.Graphics.D3DUT;
3.
4.  namespace D3DUTSimpleApp
5.  {
6.   public class D3DUTSimpleApp
7.   {
8.    private VertexBuffer vertexBuffer;
9.
10.   public D3DUTSimpleApp()
11.   {
12.    Runtime runtime = new Runtime(640, 400, "Simple Application");
13.
14.    runtime.DeviceCreated += new DeviceCreatedEventHandler(runtime_DeviceCreated);
15.    runtime.Paint += new PaintEventHandler(runtime_Paint);
16.    runtime.Idle += new IdleEventHandler(runtime_Idle);
17.
18.    runtime.CreateDevice();
19.
20.    runtime.Run();
21.   }
22.
23.   void runtime_DeviceCreated(Runtime sender, Device newDevice)
24.   {
25.    vertexBuffer = new VertexBuffer(typeof(CustomVertex.PositionColored),
26.                                    3, newDevice, Usage.None,
27.                                    CustomVertex.PositionColored.Format,
28.                                    Pool.Managed);
29.
30.    CustomVertex.PositionColored[] vertices = new CustomVertex.PositionColored[]
31.    {
32.     new CustomVertex.PositionColored( -0.75f, -0.75f, 0f, Color.Red.ToArgb()),
33.     new CustomVertex.PositionColored(     0f,  0.75f, 0f, Color.Blue.ToArgb()),
34.     new CustomVertex.PositionColored(  0.75f, -0.75f, 0f, Color.Green.ToArgb())
35.    };
36.
37.    vertexBuffer.SetData(vertices, 0, LockFlags.None);
38.   }
39.
40.   void runtime_Paint(Runtime sender)
41.   {
42.    sender.Device.Clear(ClearFlags.Target, Color.WhiteSmoke, 1f, 0);
43.    sender.Device.BeginScene();
44.
45.    sender.Device.SetStreamSource(0, vertexBuffer, 0);
46.    sender.Device.VertexFormat = CustomVertex.PositionColored.Format;
47.    sender.Device.RenderState.Lighting = false;
48.    sender.Device.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
49.
50.    sender.Device.EndScene();
51.    sender.Device.Present();
52.   }
53.
54.   void runtime_Idle(Runtime sender)
55.   {
56.    sender.Repaint();
57.   }
58.
59.   static void Main()
60.   {
61.    new D3DUTSimpleApp();
62.   }
63.  }
64. }
```

Kód 2.2-1: Jednoduchá D3DUT aplikace

Aby bylo možné používat rozhraní D3DUT, je třeba přidat referenci na jmenný prostor *Zcu.Graphics.D3DUT* (Kód 2.2-1: 2).

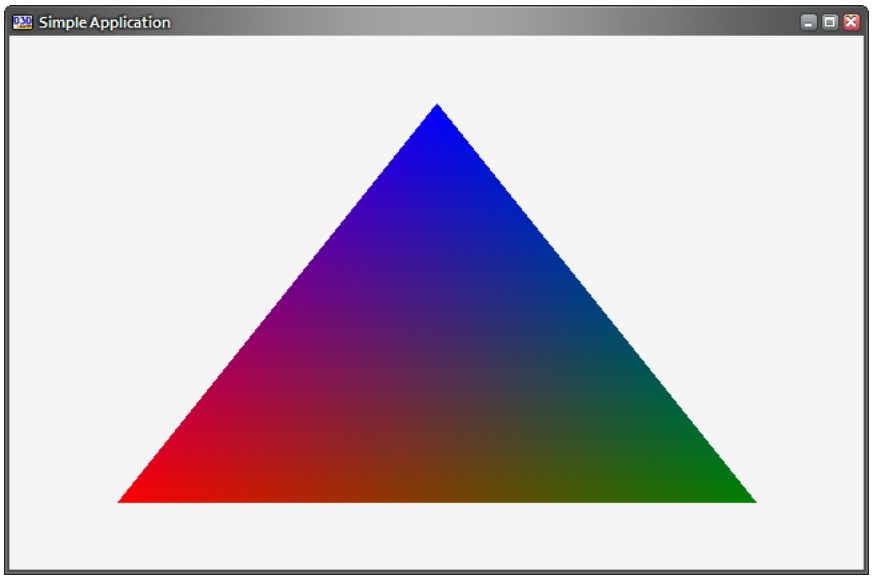
V konstruktoru vzorové aplikace (Kód 2.2-1: 10-21) je vytvořena instance třídy *Runtime*. Ta představuje běhové prostředí D3DUT. Následuje zaregistrování událostí *DeviceCreated*, *Paint* a *Idle*. Je také možno zaregistrovat události, které reagují na uživatelské vstupy, klávesnici či myš. Metodou *CreateDevice()* dojde k inicializaci zařízení a metodou *Run()* ke spuštění smyčky běhového prostředí.

Metoda *runtime_DeviceCreated()* (Kód 2.2-1: 23-35) je zaregistrována u události *DeviceCreated*. Ta je vyvolána po každém restartu zařízení, například při změně rozlišení nebo přepnutí do celoobrazovkového módu. V naší ukázkové aplikaci zde dojde k vytvoření vertex bufferu a jeho naplnění třemi body, které definují trojúhelník.

Metoda *runtime_Paint()* (Kód 2.2-1: 40-52) je volána pokaždé, když je třeba překreslit obsah okna. Nejprve je oblast určená pro vykreslení vyplněna světle šedou barvou, poté je vykreslen trojúhelník.

Metoda *runtime_Idle()* je volána, nemá-li běhové prostředí co na práci. Před vyvoláním události překreslení okna, *Repaint()*, je tak možné například implementovat logiku pro počítání počtu snímků zobrazených za jednu vteřinu.

Metoda *Main()* je standardní. Vytvoří pouze instanci třídy vzorové aplikace. Výsledek je vidět na (Obrázek 2.2-2).



Obrázek 2.2-2: Výstup jednoduché D3DUT aplikace

2.3 Přehled existujících řešení

MVE-2 není ojedinělým projektem svého typu. Existuje několik řešení ať už komerčních či povahy open source. Některá jsou změřena přímo na vizualizaci dat, jiná na zpracování signálu.

Z předpokládaného nasazení těchto řešení vyplývá i způsob, jakým budou zpracovaná data prezentována. Orientační přehled dostupných modulárních prostředí je shrnut v této kapitole.

2.3.1 VTK

Visualization Toolkit je open source software firmy Kitware Inc. [VTK]. Zaměřuje se na počítačovou grafiku, vizualizaci a zpracování obrazu. Využívá principu vizualizační pipeline a toku dat.

VTK samo o sobě neobsahuje grafické uživatelské prostředí. Uživatel píše aplikace v programovacím jazyce C++, případně využívá wrapperů pro jazyky Tcl, Java, Python či některého z jazyků pro platformu .NET [VTKNET]. V současnosti je dostupných přes 800 tříd pro nejrůznější použití.

Lze je rozdělit do těchto skupin:

- objekty grafické pipeline
- objekty vizualizační pipeline
- pomocné objekty

Objekty grafické pipeline jsou elementy scény a objekty potřebné pro rendering. Z objektů vizualizační pipeline se sestavuje graf toku dat, jejich hlavní úlohou je transformovat vstupní data do reprezentace, kterou je možné vykreslit.

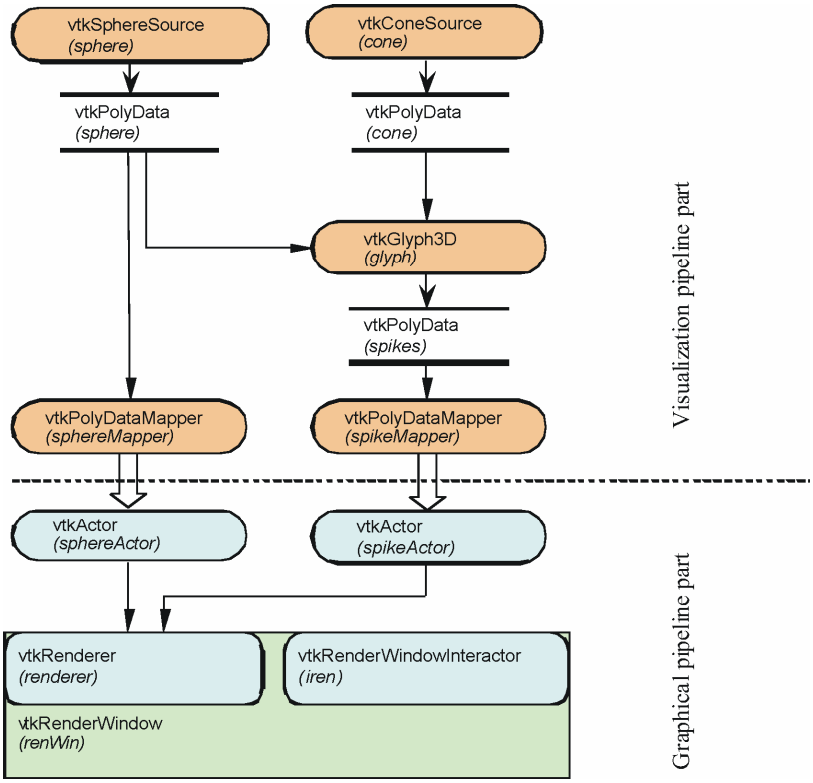
Zatímco objekty vizualizační pipeline jsou striktně rozděleny na objekty datové a výkonné, objekty grafické pipeline mají většinou charakteristiku obou. Jejich podobjekty jsou závislé na hardware, resp. na grafickém rozhraní.

Typickou „Hello World !“ aplikací demonstrující použití VTK je The Mace Example. Výsledkem je polygonální reprezentace koule s jehlanem v každém vrcholu, který má směr podle orientace normály v tomto vrcholu (Obrázek 2.3-2). Z grafu toku dat (Obrázek 2.3-1) je patrné rozdělení na část vizualizační a grafickou.

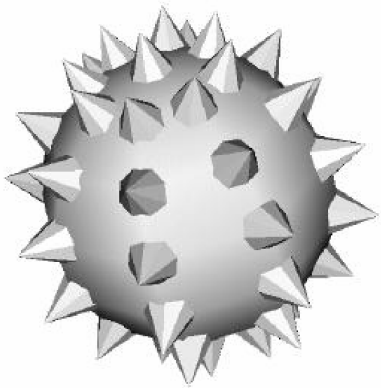
Vizualizační pipeline začíná zdrojem. Zde to jsou objekty *vtkSphereSource* a *vtkConeSource* generující polygonální reprezentaci jehlanu a koule. Koule je napojena na *vtkPolyDataMapper* a *vtkGlyph3D*. Filtr *vtkGlyph3D* na pozici každého vrcholu vstupního datasetu zkopíruje *Glyph*. *Glyph* je definován vstupními polygonálními daty filtru, jeho směr a velikost může být závislá na normále, resp. vektoru. V našem případě tedy vygeneruje ostny palcátu. Objekt *vtkPolyDataMapper* mapuje polygonální data na grafická primitiva. Hraje roli

supertřídy pro poly data mappery, které jsou již závislé na konkrétním zařízení či rozhraní[1].

Grafická pipeline začíná definováním základních elementů scény, objekty *vtkActor*. Ty jsou vloženy do *vtkRendereru*, který má za úkol jejich vykreslení, rendering. Obdobně by se do *vtkRendereru* vkládalo i světlo, *vtkLight*. Okno, do kterého lze renderovat, je poskytnuto objektem *vtkRenderWindow*. Základní interaktivitu, jakou je rotace a posuv objektu, zajišťuje objekt *vtkRenderWindowInteractor*.



Obrázek 2.3-1: Palcát, příklad sestavení grafu ve VTK



Obrázek 2.3-2: Palcát, výstup příkladu (Obrázek 2.3-1)

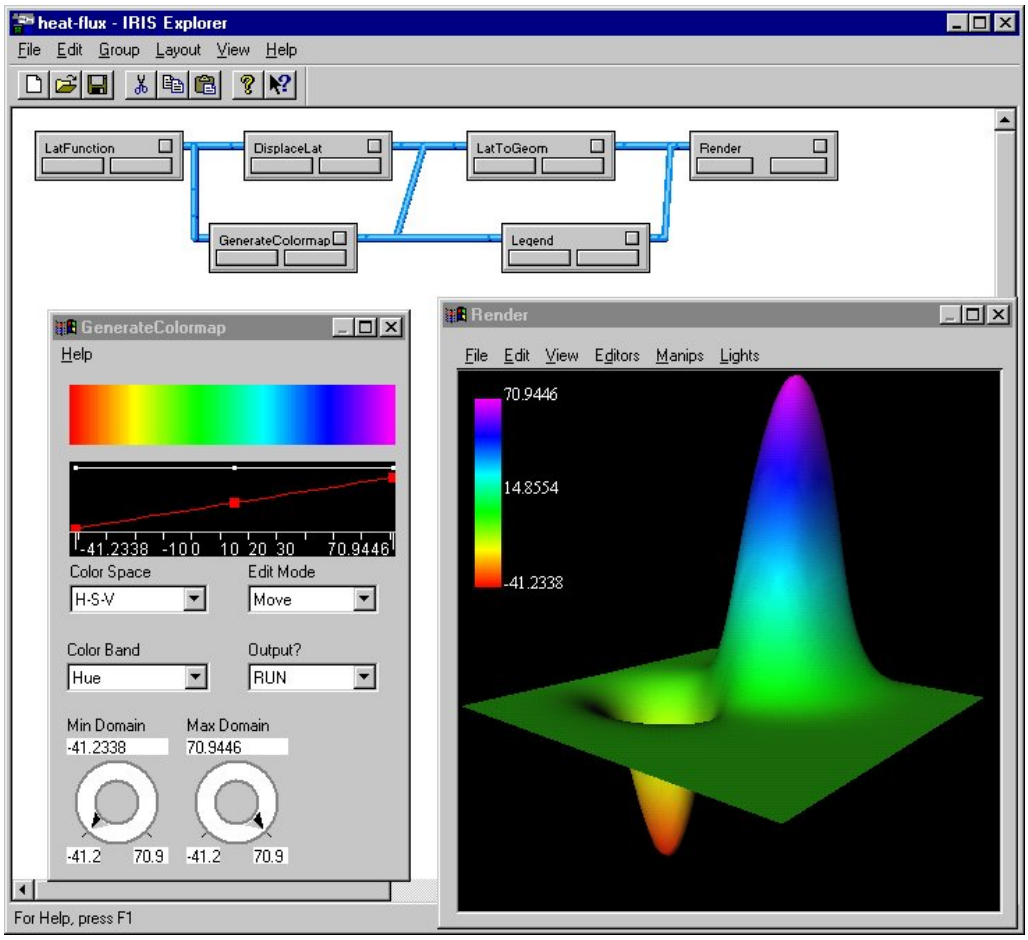
[1] Například *vtkOpenGLPolyDataMapper* pro rozhraní OpenGL, *vtkMesaPolyDataMapper* pro rozhraní Mesa.

2.3.2 Iris Explorer

Iris Explorer skupiny Numerical Algorithms Group [IRIS] je výkonný, sofistikovaný, vizualizační balík s grafickým uživatelským prostředím (Obrázek 2.3-3). Umožňuje uživatelům a programátorům vyvíjet aplikace pro vizualizaci rozsáhlých dat. Akcelerovaný grafický výstup využívá rozhraní OpenGL [OGL] a objektově orientovaný toolkit Open Inventor [OINV].

Standardní datové typy pro data předávaná mezi moduly jsou:

- *Lattice* – Skupina jedno či vícerozměrných polí, například volumetrická data získaná jako CT snímek kosti by byla uložena jako data tohoto datového typu.
- *Pyramidal* – Hierarchie dat typu Lattice spolu s informací o tom, jak jsou spolu propojeny.
- *Geometry* – Kolekce datových primitiv jako jsou polygony, úsečky a body, sloužící k vizuální reprezentaci datové množiny.
- *Parameter* – Datový typ určený k uložení textového řetězce, celočíselné či reálné hodnoty.
- *Pick* – Speciální datový typ, do kterého se ukládají informace o zobrazovaných datech v aktuálně vybrané oblasti výstupního okna.



Obrázek 2.3-3: Iris Explorer, příklad vizualizace tepelného toku

O vykreslení zpracovaných dat se stará jediný modul Renderer. Má šest vstupních portů, z toho čtyři akceptují data typu *Geometry* a dva data typu *Lattice*. Jsou-li data jiného typu, musí se převést na jeden z výše uvedených. Je také přítomen port Annotation – Geometry, který umožňuje vkládat popisky částí geometrie a zobrazit je v renderovacím okně. Výstupní porty jsou čtyři, vystavují *Pick* data, snímek scény, export scény a poskytují synchronizaci pro cykly.

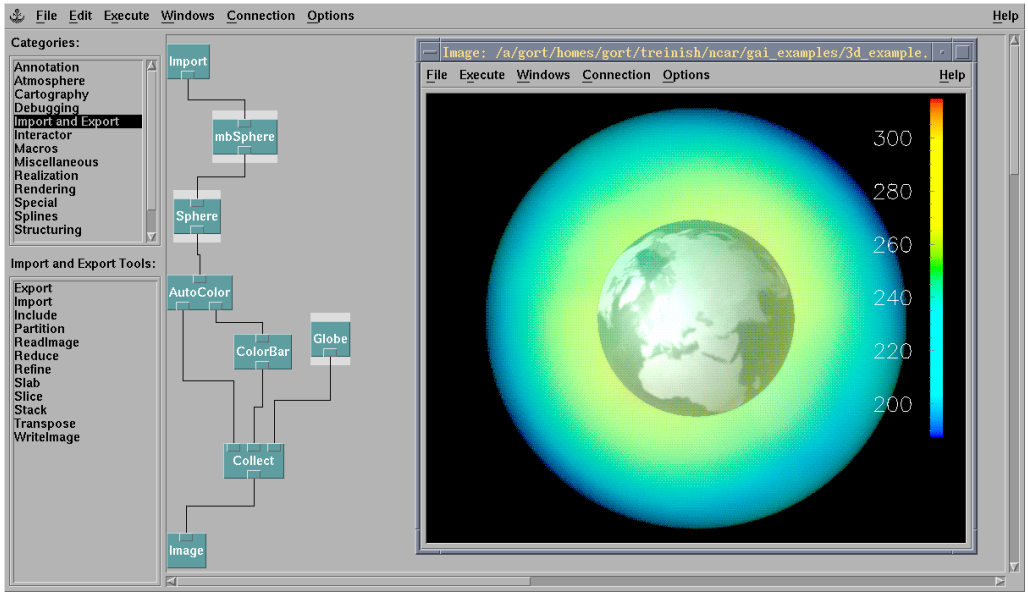
Renderer je poměrně komplexní modul. Nastavení, např. materiál objektů ve scéně, osvětlení scény, atd., se provádí přímo v editorech modulu.

2.3.3 OpenDX

Open Visualization Data Explorer je univerzální softwarový balík pro analýzu dat a vizualizaci. IBM Research uvedla na trh původní verzi v roce 1991. O necelých deset let později byl projekt převeden na open source a přejmenován z IBM Visualization Data Explorer na IBM Open Visualization Data Explorer.

OpenDX využívá modelu toku dat a je postaveno na robustní klient-server architektuře. Klientským procesem je grafické uživatelské prostředí běžící na pracovní stanici (Obrázek 2.3-4). Serverový proces vykonává veškeré výpočty, může běžet na stejné nebo jiné pracovní stanici, multiprocesorové stanici, serveru či farmě serverů. Je tak možné zpracovávat obsáhlá data s potřebným stupněm interaktivity. Požadavky na výpočet jsou serverové části předávány skriptem vygenerovaným klientskou částí. Akcelerovaný grafický výstup na pracovní stanici je realizován prostřednictvím rozhraní OpenGL [OGL].

Data předávaná mezi moduly vychází z typu objekt. Základním datovým typem je tzv. *Fields*, který se skládá z komponent. Těch může být několik a různého typu, většinou se jedná o kolekce geometrie, topologie, atributů, propojení na data z jiného objektu *Fileds*, atp.



Obrázek 2.3-4: OpenDX, vizualizace volumetrických dat ve sférických souřadnicích

Objekty akceptované moduly mohou být téměř libovolného datového typu. Modul buď s daným objektem umí pracovat a provede v něm očekávané změny, nebo s objektem pracovat neumí a beze změny jej pošle na výstup.

Pro vykreslení datových objektů je k dispozici trojice modulů, Render, Image a Display. Vstupem modulu Render je objekt a nastavení kamery. Výstupem je vykreslený obraz *Image*, který může být poslán do modulu WriteImage pro uložení do souboru, do modulu Display, pro zobrazení na obrazovce a nebo do jiných modulů, například pro kompozici více obrazů.

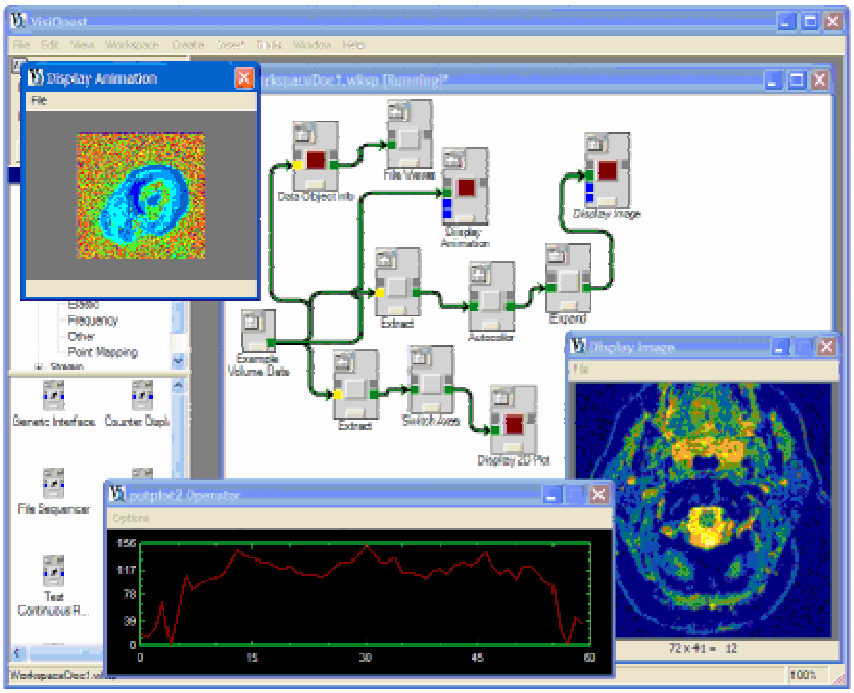
Vstupem modulu Image je pouze objekt. Nastavení kamery je možné díky rozšířením přidaným do okna ImageWindow, výstupem modulu je vykreslený obraz a nastavení kamery.

Modul Display umí zobrazit vykreslené obrazy i jejich kompozice do okna ImageWindow. Druhou možností je přivést na vstup objekt a nastavení kamery, potom dojde k vykreslení scény. Toto použití je vhodné pro vytváření animací, případně zajištění vlastní formy interakce.

2.3.4 VisiQuest

Vizuální programovací nástroj VisiQuest je komerčním produktem firmy AccuSoft [VQST].

Jeho datový model, označovaný jako polymorfní, je poměrně složitý. Umožňuje reprezentovat téměř jakákoliv data a jejich časovou proměnu, díky polymorfismu pak moduly dokáží správným způsobem tato data zpracovat.



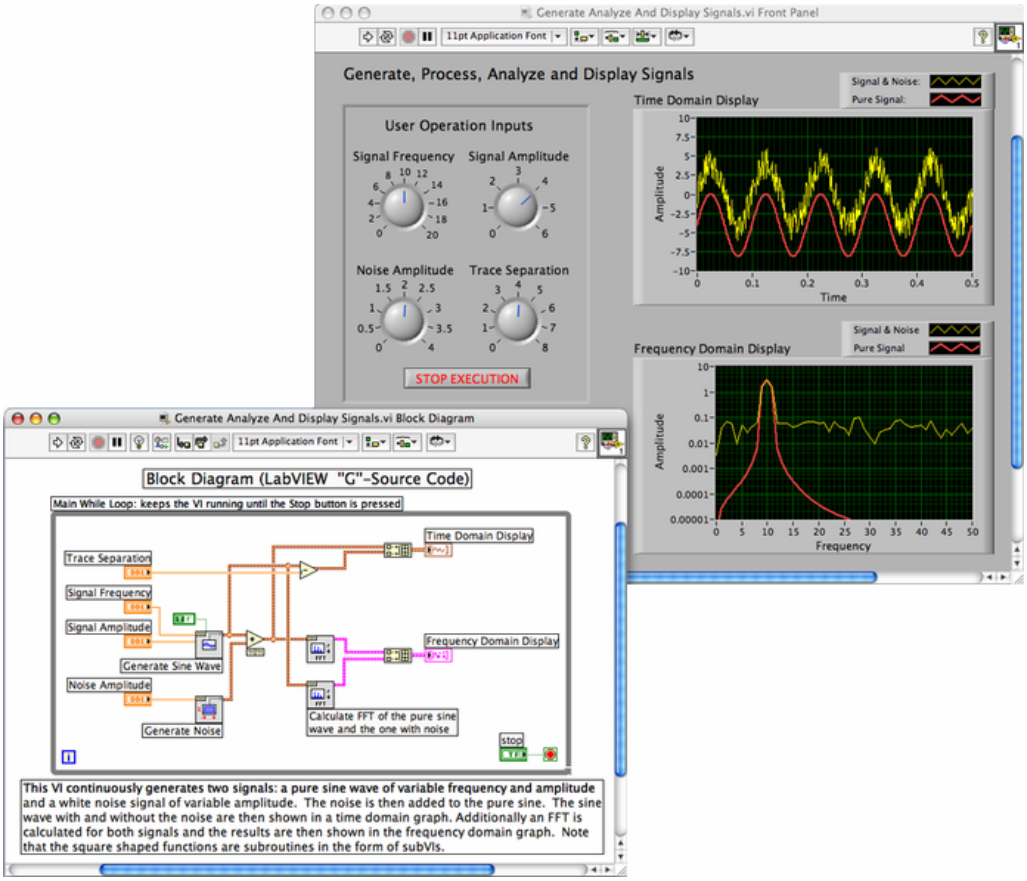
Obrázek 2.3-5: VisiQuest

Pro vizualizaci je dodáváno několik modulů pro různé použití, DisplayImage pro zobrazení obrazových dat, DisplayPlot pro zobrazení grafů, XPrism pro zobrazení 2D a 3D grafů.

2.3.5 LabView

LabView [LVIEW] je produktem firmy National Instruments. Jedná se o komplexní programovací nástroj pro vizuální programovací jazyk G. Jeho historie sahá až do roku 1986, kdy byl uveden pro platformu Apple Macintosh.

Prostředí je určené především do průmyslového nasazení a ke zpracování signálů. Tomu odpovídá dostupné množství ovladačů pro nejrůznější zařízení a sondy. Pro zobrazení naměřených dat jsou připraveny ovládací a vizualizační prvky, které se umisťují do řídících panelů.



Obrázek 2.3-6: LabView

3 Realizační část

Tato část je věnována vlastnímu řešení zadání diplomové práce. Byla vytvořena knihovna UnstrGridRendering, jež obsahuje moduly a třídy potřebné k sestavení scény a jejímu následnému vykreslení.

Realizace se neobešla beze změn v projektu MVE-2. Požadavek na interaktivitu a zároveň potřeba průhledného mechanismu synchronizace modulů vedla k zásahům přímo do jeho jádra, přispění do projektu D3DUT bylo minimální.

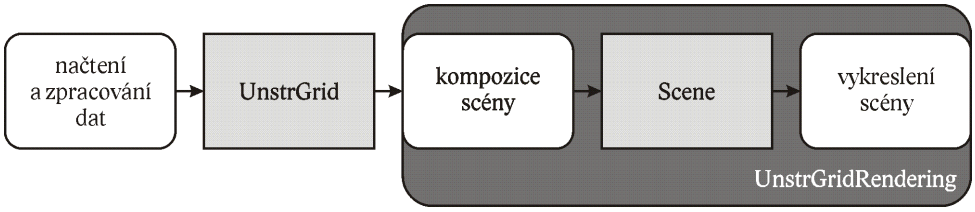
Výsledky řešení jsou demonstrovány na několika příkladech.

3.1 Knihovna UnstrGridRendering

Následující text popisuje vytvoření a strukturu knihovny UnstrGridRendering.

3.1.1 Vykreslení scény jako poslední krok vizualizace dat

Filosofie prostředí MVE-2 je založena na modelu toku dat. Tento model je vhodný i k jiným využitím než pouze k vizualizaci, tato práce je však zaměřena právě na ni.



Obrázek 3.1-1: Znázornění toku dat

Vizualizace dat dle modelu toku dat s použitím knihovny UnstrGridRendering je znázorněna na (Obrázek 3.1-1). Nejprve je třeba data načíst, ať už ze souboru,

který je umístěn na pevném disku, externího zařízení připojeného k některému z portů pracovní stanice, nebo je nechat vygenerovat modulem.

Načtená data jsou dále zpracována, může se jednat o rekonstrukci modelu, filtrování, apod. Přestože je tato část znázorněna jedním blokem, může být velmi obsáhlá a časově náročná. Důležitý je výstup, který musí být typu *UnstrGrid*, má-li být vizualizován moduly knihovny UnstGridRendering.

Následuje kompozice scény. Podobně jako scéna, kterou se snaží zachytit kameraman na film, i ta komponovaná v MVE-2 obsahuje objekty, kterých může být libovolné množství. V reálném světe je objektem židle, stůl či váza. V prostředí MVE-2 jsou objekty, ze kterých se komponuje scéna, zpracovaná data. Ta mohou být různé povahy. Naměřená data bude pravděpodobně stačit pro lepší vypovídající schopnost obarvit, naproti tomu rekonstruovaný model mající odpovídat co nejvíce svému vzoru bude modelován z nějakého materiálu, případně pokryt texturou. Má-li na filmovém plátně vyznít hra světel a stínů, neobejde se scéna bez osvětlení. Nejinak je tomu u scény komponované v MVE-2.

Sestavená a nasvícená scéna je uložena. Fotograf ji vyfotí, kameraman nafilmuje, modul z knihovny Elementals uloží do formátu Xml a modul UnstrGridRenderer z knihovny UnstrGridRendering vykreslí do okna na monitor pracovní stanice.

Datový typ UnstrGrid

Datový typ *UnstrGrid* již byl popsán v (Kapitola 2.1.5). Jedná se o nejobecnější datovou struktury knihovny Visualizationm, která obsahuje pole bodů *Points*, uniformní pole buněk *Cells*, atributy bodů *PointAttrs* a atributy buněk *CellAttrs*.

V poli buněk *Cells* mohou být libovolné buňky implementující rozhraní *ICell*. Není však možné, aby knihovna vytvořená v rámci této diplomové práce počítala i s typy buněk, které budou vytvořeny v budoucnu. Skupina buněk zobrazitelných knihovnou UnstrGridRendering byla tedy omezena na následující typy:

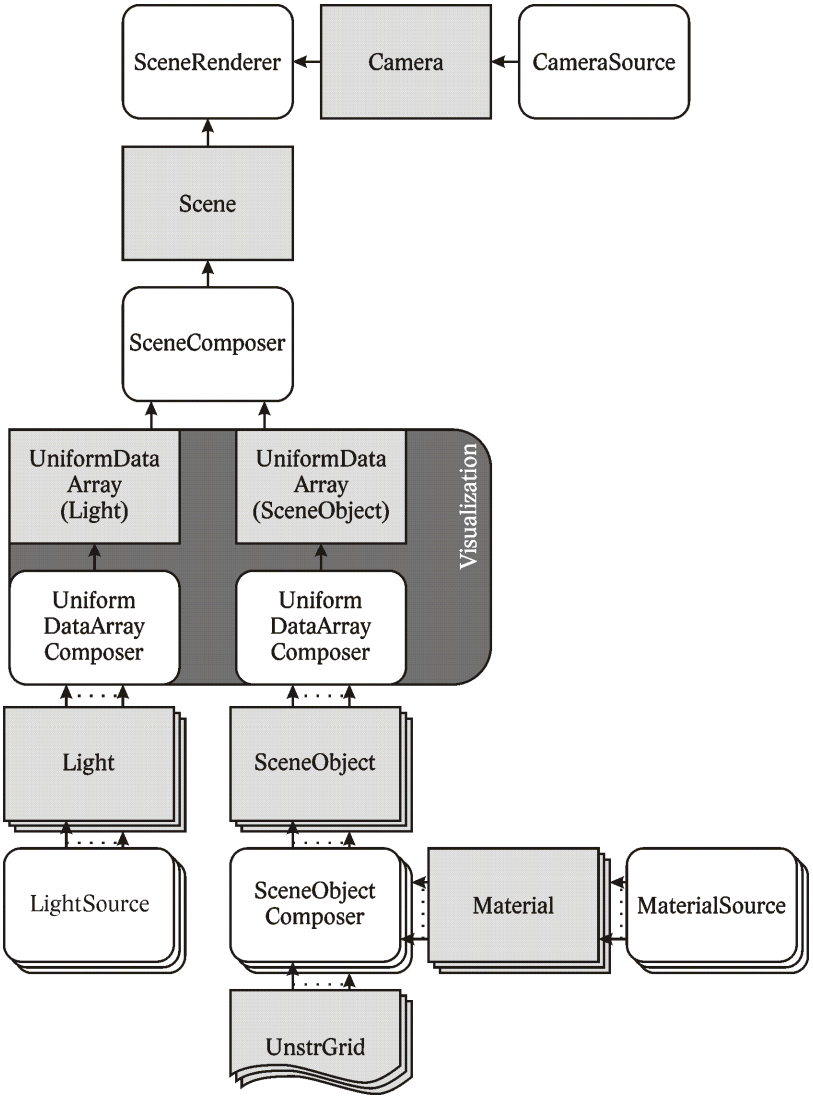
- *Line*
- *PolyLine*
- *Triangle*
- *TriangleStrip*
- *TriangleFan*
- *Tetrahedron*
- *Slice*

3.1.2 Kompozice scény

Modul, který se stará o vykreslování, je označován jako renderer. Obecně jsou jeho vstupem objekty scény a světla. Je několik možností, jak lze vyřešit

zobrazení určitého množství objektů scény a nasvícení více světelnými zdroji. Je též možné modulu renderer dodat informace o kameře, tj. umístění a způsobu projekce. V přehledu existujících řešení byly uvedeny různé přístupy, některé z nich nastavení kamery a osvětlení řešily až v ovládacích prvcích renderovacího modulu, přičemž vstupem takového modulu pak byla pouze data určená k vykreslení.

Při realizaci této diplomové práce byl zvolen velmi obecný model kompozice dat, která mají být vykreslena modulem renderer (Obrázek 3.1-2). Model umožňuje sestavit scénu o libovolném počtu objektů a světel[1]. Zároveň je možné mezi jednotlivé moduly komponující scénu vkládat moduly další, které upravují či transformují vygenerované objekty. Není tedy problém vytvořit například modul, který by měnil barvu či pozici světla.



Obrázek 3.1-2: Schéma kompozice scény

Následuje podrobnější popis datových objektů a modulů, které je generují.

[1] Počet světel ve scéně je omezen na úrovni D3DUT možnostmi hardwaru grafického akcelerátoru.

Light

Datový typ *Light* je určen k popisu světelného zdroje.

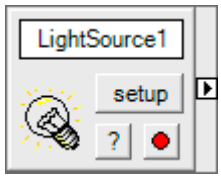
Vlastnosti:

- *Enabled* – Určuje, zda je světlo použito při vykreslování scény.
- *Type* – Typ světelného zdroje. Je možné vybrat mezi *Directional*, *Point* a *Spot*. Typ *Directional* představuje rovnoběžné paprsky světla procházející celou scénou, v reálu takové osvětlení odpovídá slunečním paprskům a je výpočetně nejméně náročné. Typ *Point* je bodovým zdrojem světla, jehož intenzita se vzdáleností od středu klesá, reálným příkladem je žárovka bez stínítka. Výpočetně nejnáročnější je typ *Spotlight*, který vytváří světelný kužel podobně jako v reálném světě svítilna.
- *Position* – Pozice světla ve scéně. Tato hodnota nemá vliv na světla typu *Directional*.
- *Direction* – Směr světla, hodnota má smysl u světelných zdrojů typu *Directional* a *Spotlight*.
- *ShowLight* – Umožňuje zobrazit ve scéně pozici a směr světelného zdroje.
- *Attenuation[0 | 1 | 2]* – Konstanty útlumu určují, jak se světelná intenzita mění se vzdáleností od zdroje světla. Výsledná intenzita je počítána dle vzorce (Vzorec 3.1-1), hodnota nemá smysl u světelného zdroje typu *Directional*.

$$LightIntensity = \frac{SourceIntensity}{At_0 + At_1 \cdot Distance + At_2 \cdot Distance^2}$$

Vzorec 3.1-1: Výpočet intenzity osvětlení

- *Range* – Udává vzdálenost, za kterou již objekty nejsou ovlivněny zdrojem světla. Vlastnost se neuplatňuje u světel typu *Directional*.
- *InnerConeAngle* – Úhel vnitřního světelného kužele. Hodnota má vliv na světla typu *Spotlight*.
- *OuterConeAngle* – Úhel vnějšího světelného kužele. Hodnota má vliv na světla typu *Spotlight*.
- *Falloff* – Útlum mezi vnitřním a vnějším světelným kuželem světla typu *Spotlight*.
- *Ambient* – Ambientní barevná složka světelného zdroje.
- *Diffuse* – Difúzní barevná složka světelného zdroje.
- *Specular* – Odrazivá barevná složka světelného zdroje.



Obrázek 3.1-3: Modul LightSource

O generování a nastavení datového objektu *Light* se stará modul LightSource (Obrázek 3.1-3).

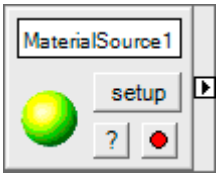
Material

Datový typ *Material* popisuje barevné vlastnosti materiálu objektu scény.

Vlastnosti:

- *Ambient* – Ambientní barevná složka materiálu.
- *Diffuse* – Difúzní barevná složka materiálu.
- *Specular* – Odrazivá barevná složka materiálu.
- *Emmisive* – Emisivní barevná složka materiálu.
- *SpecularEnable* – Určuje, zda se má při výpočtu osvětlení počítat s odrazivou barevnou složkou.
- *SpecularSharpness* – Odrazivost materiálu.

O generování a nastavení datového objektu *Material* se stará modul Material-Source (Obrázek 3.1-4).



Obrázek 3.1-4: Modul MaterialSource

SceneObject

Datový typ *SceneObject* popisuje objekt scény. Je jakýmsi obalem dat, která chceme zobrazit. Povinně obsahuje dataset *UnstrGrid*, který definuje geometrii a topologii objektu, nepovinně materiál. Mimo to jsou jeho součástí nastavení, jež určují jakým způsobem má být vykreslen.

Vlastnosti:

- *SceneObjectName* – Jméno objektu scény.
- *ShowSceneObject* – Určuje, zda má být objekt vykreslen ve scéně.
- *ShowBoundingBox* – Určuje, zda má být během vykreslování zobrazen bounding box objektu.
- *TextureAttributeName* – Jméno atributů, které budou při vykreslování použity jako atributy s texturovacími souřadnicemi.
- *NormalAttributeName* – Jméno atributů, které budou při vykreslování použity jako atributy s normálovými vektory.
- *ColorAttributeName* – Jméno atributů, které budou při vykreslování použity jako atributy s informací o barvě.
- *PrefferedVertexAttributes* – Určuje, které atributy, případně jejich kombinace mají být použity při stínování.
- *ColorVertex* – Určuje, zda má být při vykreslování objektu zapnuto obarvování vrcholů objektu.

- *Lighting* – Určuje, zda má být při vykreslování zapnuto stínování.
- *MaterialEnable* – Určuje, zda při vykreslování bude použit materiál.
- *Cull* – Určuje, jestli dojde k vykreslení odvrácených trojúhelníků. Je možno zvolit, zda mají být vykresleny trojúhelníky po směru či proti směru hodinových ručiček.
- *Render* – Umožňuje vybrat, zda se mají vykreslovat buňky datasetu, nebo jen jeho body.
- *FillMode* – Nastavuje způsob výplně, k výběru jsou varianty: *Point*, vykreslí pouze vrcholy, *Wireframe*, vykreslí drátěný model, *Solid*, vykreslí primitiva s výplní.
- *ShadeMode* – Určuje, která ze stínovacích metod bude použita, možné volby jsou *Flat* a *Gouraud*.
- *PointSize* – Definuje velikost bodu, jsou-li vykreslovány pouze vrcholy objektu.
- *NormalizeNormals* – Umožňuje zapnout automatickou normalizaci normálových vektorů.

Kompozici a nastavení datového objektu *SceneObject* zajišťuje modul SceneObjectComposer (Obrázek 3.1-5).

Obrázek 3.1-5: Modul SceneObjectComposer

Scene

Datový typ *Scene* je abstrakcí scény, představuje úplný popis uspořádání objektů a světel, který je připraven k vykreslení renderovacím modulem, uložení do souboru, atd. Scéna povinně obsahuje objekty scény, pole objektů typu *SceneObject*. Pole objektů jednotného typu se vytváří modulem UniformDataArrayComposer z knihovny Visualization. Do scény je také vhodné vložit světla, pole objektů typu *Light*, chceme-li, aby scéna byla nejen vykreslena, ale i vystínována.

Vlastnosti:

- *ShowSceneOrigin* – Určuje, zda má být vykreslen počátek scény.
- *SceneOriginSize* – Určuje velikost elementu zobrazujícího počátek scény.

Úkolem modulu SceneComposer (Obrázek 3.1-6) je vytvořit kompozici scény, jako objekt nesoucí reference na pole objektů scény a pole světel.

Obrázek 3.1-6: Modul SceneComposer

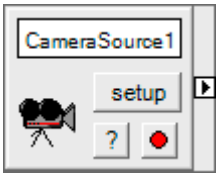
Camera

Datový typ *Camera* slouží k uložení informací o kameře, která snímá scénu. Jedná se o volitelnou, dodatečnou informaci pro modul SceneRenderer.

Vlastnosti:

- *Position* – Umístění kamery ve scéně.
- *Target* – Vektor určující směr kamery, tj. kam se kamera dívá.
- *UpVector* – Vektor určující orientaci kamery, tj. kde je nahoře.
- *Fov* – Úhel záběru kamery.
- *ZFarPlane* – Vzdálenost od kamery, za kterou již umístěné objekty nemají být vykreslovány.
- *ZNearPlane* – Vzdálenost od kamery, před kterou již umístěné objekty nemají být vykreslovány.
- *ProjectionType* – Určuje typ projekce, je možnost vybrat projekci perspektivní, ortogonální nebo dodat vlastní matici projekce.
- *Scale* – Měřítko rozsahu zobrazení scény při ortogonální projekci. Standardně je zobrazen rozsah -1 až 1 v horizontálním i vertikálním směru, je-li například hodnota Scale rovna například 2, zobrazí se rozsah -2 až 2.

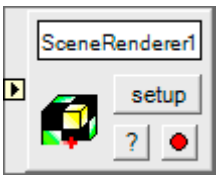
Vygenerování objektu typu *Camera* má na starosti modul CameraSource (Obrázek 3.1-7). Lze v něm nastavit vystavení vstupních portů a tím měnit pozici kamery a projekční matici mimo modul.



Obrázek 3.1-7: Modul CameraSource

3.1.3 Vykreslení scény

O vykreslení scény se stará modul SceneRenderer (Obrázek 3.1-8), jehož vstupem je povinně datový objekt *Scene* a nepovinně datový objekt *Camera*. Přímo v nastavení modulu lze zadat velikost výstupního okna a barvu pozadí.



Obrázek 3.1-8: Modul SceneRenderer

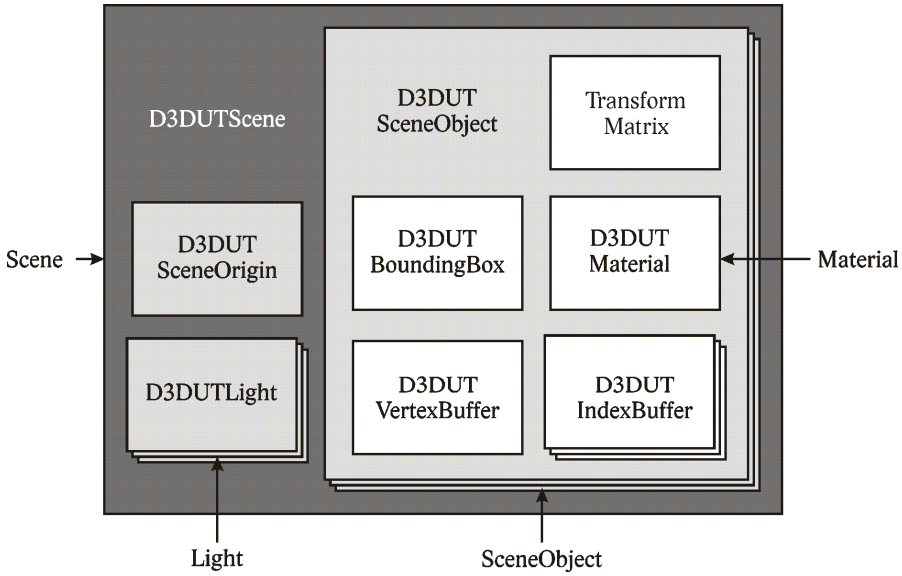
Modul zajišťuje také základní obsluhu uživatelských vstupů. Zobrazenou scénou lze otáčet, přibližovat ji, či oddalovat. Zároveň je uživateli ponechán prostor napsat si vlastní modul pro obsluhu vstupních zařízení, připojit jej na vstup modulu CameraSource a zajistit si tak vlastní způsob pohybu scénou.

K akcelerovanému vykreslení scény se používá rozhraní D3DUT. To řeší problémy s budoucím přechodem na platformu Linux či vykreslování na zařízeních pro stereoskopické zobrazení.

Spuštěním modulu dojde k vygenerování reprezentace scény z objektů, které již využívají rozhraní D3DUT (Obrázek 3.1-9), odtud jejich pojmenování.

Objekty, jejichž název začíná na D3DUT, uchovávají data ve strukturách či objektech D3DUT a mají metody k jejich updatu a vykreslení, zároveň si uchovávají referenci na odpovídající rodičovský objekt.

Situace je jednoduchá u objektů *D3DUTLight* a *D3DUTMaterial*. Ty pouze uchovávají vlastnosti svých rodičovských objektů *Light* a *Material*. Objekt *D3DUTSceneObject* je poněkud komplikovanější, protože musí efektivně zajistit převod geometrie a topologie datasetu *UnstrGrid*, do paměti grafického akcelerátoru.



Obrázek 3.1-9: Kompozice scény z objektů využívajících rozhraní D3DUT

O naplnění verte bufferu body datasetu *UnstrGrid* se stará třída *D3DUTVertexBuffer*. Převod bodů spolu s atributy uloženými v datasetu *UnstrGrid* na body typu *CustomVertex* je proveden dle tabulky (Tabulka 3.1-1). Atributy podporované knihovnou UnstrGridRendering odpovídají atributům bodů rozhraní D3DUT, jedná se tedy v podstatě o převod jedna ku jedné.

Standardní jména atributů	Preferované verte atributy	VertexFormat
----------	None	PositionOnly
TextureUV	TextureUV	PositionTextured
ColorRGBA	ColorRGBA	PositionColored
ColorRGBA, TextureUV	ColorRGBATextureUV	PositionColoredTextured
Normal3D	Normal3D	PositionNormal
Normal3D, ColorRGBA	Normal3DColorRGBA	PositionNormalColored
Normal3D, TextureUV	Normal3DTextureUV	PositionNormalTextured

Tabulka 3.1-1: Převod bodů a atributů datasetu *UnstrGrid* na D3DUT body typu *CustomVertex*

Podobně se o naplnění index bufferů stará třída *D3DUTIndexBuffer*. Převod buňek datasetu *UnstrGrid* na D3DUT primitiva se děje podle (Tabulka 3.1-2).

Typ buňky datasetu UnstrGrid	PrimitiveType
Triangle	TriangleList
TriangleStrip	TriangleStrip
TriangleFan	TriangleFan
Tetrahedron	TriangleStrip
Line	LineList
Polyline	LineStrip
Slice	LineStrip

Tabulka 3.1-2: Převod buněk datasetu *UnstrGrid* na D3DUT primitiva

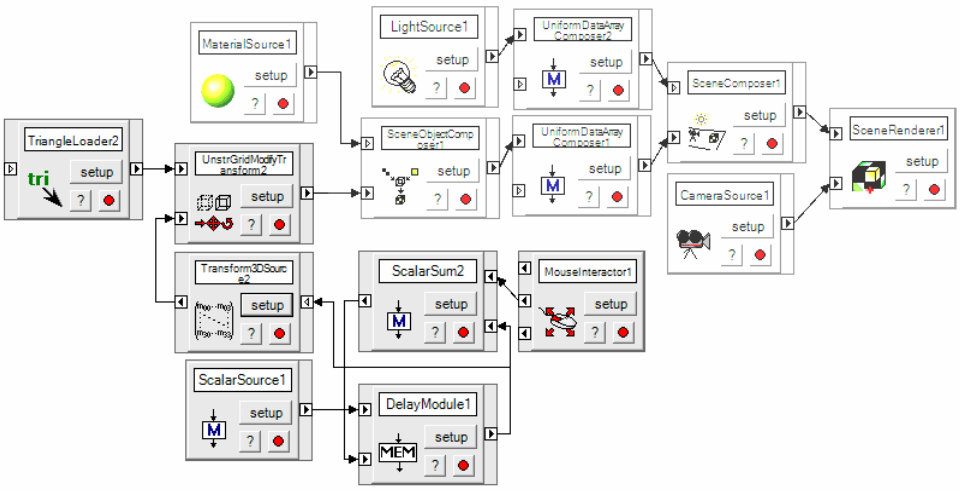
Zde se již na straně datasetu *UnstrGrid* objevují typy buněk, které nemají přímý protějšek v primitivech rozhraní D3DUT. Čtyřstěn, *Tetrahedron*, je tedy nahrazen reprezentací zřetězených trojúhelníků, *TringleStrip*. Řezy, *Slice*, jsou reprezentovány několika zřetězenými úsečkami *LineStrip*.

Vlastní vykreslování respektuje hierarchii kompozice objektů. Renderer zavolá metodu *Draw()* objektu typu *D3DUTScene*. Objekt typu *D3DUTScene* postupně volá metody *SetLight()* objektů typu *D3DUTLight*, metodu *Draw()* objektů typu *D3DUTSceneObject* a *D3DUTSceneOrigin*.

Objekt typu *D3DUTSceneObject* provede nejprve nastavení transformační matice, následuje přiřazení materiálu, vertex bufferu, index bufferů a zobrazení bounding boxu.

3.1.4 Reakce na změny v zobrazované scéně

Scéna určená k vykreslení nemusí být statická. Před vykreslením každého obrázku modul SceneRenderer aktualizuje scénu vyvoláním události *Update* na svém vstupním portu *inScene*, ve scéně tedy může dojít ke změnám a modul SceneRenderer tyto změny musí reflektovat.



Obrázek 3.1-10: Mapa s modulem MouseInteractor

Jako příklad uveďme mapu na (Obrázek 3.1-10). Modul MouseInteractor zachytává vstupní zařízení, myš a na výstupní port vystavuje hodnotu odpovídající změně polohy myši. Modul je připojen k modulu ScalarSum, který tuto změnu přičítá k aktuální hodnotě udržované modulem DelayModule. Vypočtená hodnota je vložena do transformační matice modulem Transform3DSource, modul

UnstrGridModifyTransform touto maticí nahradí stávající transformační matici v datasetu *TriangleMesh*, načteném modulem TriangleLoader. Výsledkem této části mapy je, že po spuštění bude nahraný *TriangleMesh* transformován podle pohybu myši, může jít o posuv, rotaci, zpolštění, atd.

Znovu vytvářet paralelní kompozici "D3DUT objektů": před vykreslením každého obrázku by bylo časově náročné, proto je tedy potřeba nějakým způsobem zabránit znovuvytvoření nezměněných objektů.

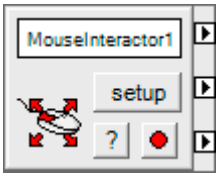
Standardním mechanismem v MVE-2 je dotazování se portu metodou *isDataSame()*, zda-li data na vstupu byla změněna. Tento mechanismus vyžaduje správně napsané moduly, neboť změna provedená v kterémkoliv objektu kompozice automaticky indikuje změnu celé kompozice. Je-li tedy například změněna transformační matice objektu scény, modul SceneRenderer se dozví, že došlo ke změně ve scéně. Nepříjemností je, že se již nedozví, který objekt scény, natož pak jeho část, změnu vyvolal. Tento způsob tedy zabrání znovuvytvoření objektů jen za předpokladu, že se ve scéně nic nezmění.

S takovým způsobem se samozřejmě při vykreslování v reálném čase nelze spokojit. K vyřešení problému byl zaveden mechanismus nový, který velmi dobře funguje například ve VTK [VTK]. Abstraktní třídě *DataObject* byla přidána vlastnost *ModificationTime*. Správně napsaná třída nyní při vytvoření instance či při změně svých dat nastaví tuto vlastnost na aktuální čas. Tuto časovou značku může s výhodou využívat prostředí MVE-2 k automatickému nastavování příznaků portů *isDataSame*. Stejně tak dobře ji ale může použít i tvůrce modulů, který si o důležitých objektech může uchovat čas jejich poslední změny a objekty aktualizovat pouze, je-li třeba.

Vraťme se k mapě (Obrázek 3.1-10) a ukažme si nový mechanismus na reálném příkladu. Každý pohyb myši má za následek změnu transformační matice objektu typu *TriangleMesh*. Díky tomu je změna propagována do objektu *SceneObject* a následně do objektu *Scene*. SceneRenderer zjistí, že scéna byla změněna. Provede tedy aktualizaci scény. Objekt typu *D3DUTScene* v metodě *Update()* zjistí podle časové značky, že scéna byla doopravdy změněna. Zavolá tedy postupně metody *Update()* pro všechny své objekty. Ty si opět porovnají své časové značky a jediný objekt, kterému se nebude shodovat, bude typu *SceneObject*. Ten opět provede porovnání, kdy byly naposled změněny body či buňky datasetu. Zde se časová značka shodne, takže nakonec dojde pouze k aktualizaci transformační matice. index buffer ani vertex buffer nebylo třeba znovu vytvářet, a proto se tak tedy nestane.

Modul MouseInteractor

Modul MouseInteractor (Obrázek 3.1-11) je vzorovým příkladem modulu, který zpracovává vstupy zařízení a propaguje je do prostředí MVE-2.



Obrázek 3.1-11: Modul MouseInteractor

Konkrétní implementace využívá rozhraní Microsoft DirectInput. Při spuštění simulace dojde k zachycení zařízení, myši, při ukončení simulace k jejímu uvolnění. Volání metody *Execute()* má za následek vystavení změny pozice myši od posledního spuštění modulu.

Není těžké si představit namísto myši jiné vstupní zařízení. Může jím být klávesnice nebo nějaké jiné, velmi specializované, zařízení připojené k rozhraní pracovní stanice. Vždy jde jen o získání jím produkovaných hodnot, ať už s použitím nějakého rozhraní jako v tomto případě DirectInput, nebo přímým přístupem na porty počítače a jejich následné vystavení na výstupní porty modulu.

3.2 Přispění do projektů MVE-2 a D3DUT

Tato diplomová práce se opírá o projekty MVE-2 a D3DUT. V současné době jsou oba ve vývoji, a tudíž použití obou projektů bylo i jakousi formou jejich testování. V jádru MVE-2 bylo provedeno několik úprav, hlavně kvůli již zmíněnému požadavku na interaktivitu.

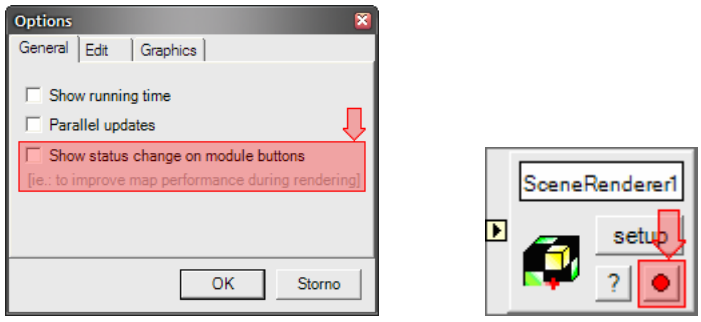
V projektu D3DUT byla odhalena chyba pro počítání vektorového součinu a přidána metoda pro výpočet pohledové matice.

3.2.1 MVE-2

Změny v jádru MVE-2 a zavedení vlastnosti objektů *ModificationTime* bylo popsáno v předchozí kapitole (3.1.4 Reakce na změny v zobrazované scéně). Byly však provedeny i úpravy v samotném MapEditoru a přidány užitečné moduly do knihovny Visualization.

MapEditor

Vlastností MapEditoru je indikace stavu modulů. Ikona na modulu ukazuje, zda je činnost modulu spuštěna, zastavena či zda modul aktualizuje vstupy.



Obrázek 3.2-1: Možnost vypnutí zobrazovaní stavu modulu

U modulů, jejichž doba výpočtu trvá dlouho[1], je systémový čas potřebný ke změně ikony indikující stav zanedbatelný. Vykreslujeme-li ale scénu šedesátkrát za vteřinu, zpomalení je již zřetelné. I když modul ve své metodě *Execute()* nevykoná žádnou časově náročnou činnost a pouze předá referenci na již dříve vypočtená data na výstupní port, musí dojít ke změně ikony, což stojí systémové prostředky. Byla proto přidána do nastavení MapEditoru (Obrázek 3.2-1) možnost zobrazování změny stavu modulu vypnout a zvýšit tím rychlost aktualizace mapy a tím i vykreslování.

[1] řekněme více než vteřinu

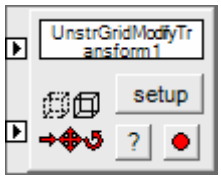
Knihovna Visualization

Datové objekty knihovny Visualization byly upraveny, aby správně nastavovaly vlastnost *ModificationTime*. Moduly PictureLoader, TriangleLoader a UnstrGridTransform byly upraveny tak, aby správně aktualizovaly svůj výstupní port. Podobné úpravy byly provedeny i u modulů z dalších knihoven, které byly použity při testování modulů z knihovny UnstrGridRendering a u kterých je předpokládána častá spolupráce. Jednalo se především o moduly Elevation, CalculateDisplace a DisplaceMap z knihovny GSVD.DisplaceMap a modul NormalComputer z knihovny Examples. Dále byly do knihovny Visualization přidány moduly UnstrGridModifyTransform, Transform3DSource a RotationTransformSource.

Modul UnstrGridModifyTransform

Knihovna Visualization obsahuje modul UnstrGridTransform. Tento modul funguje tím způsobem, že transformuje všechny body datasetu *UnstrGrid* dodanou transformační maticí. V některých případech je toto chování žádoucí, někdy je naopak dostačující a časově méně náročné pouze změnit transformační matici datasetu, vlastní transformace bodů pak může být provedena na grafickém akcelerátoru.

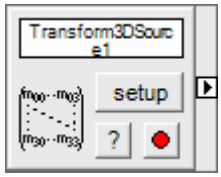
Byl proto vytvořen modul UnstrGridModifyTransform (Obrázek 3.2-2), který ovlivňuje pouze transformační matici datasetu. V nastavení modulu je možné zvolit, zda má být původní matice nahrazena, či vynásobena maticí dodanou. Je tak možné moduly zřetězit a složit transformační matici, která například provede zvětšení, posunutí a rotaci najednou.



Obrázek 3.2-2: Modul UnstrGridModifyTransform

Modul Transform3DSource

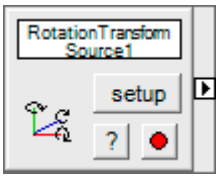
Modul Transform3DSource (Obrázek 3.2-3) generuje datový objekt typu *Transform3D*. Hodnoty generované transformační matice je možno zadávat přímo v nastavení modulu. Je také možné vystavit libovolný element matice jako vstupní port modulu a ovlivňovat tak jeho hodnotu jiným modulem. Tím může být modul ScalarSource z knihovny Numerics, stejně tak dobře ale i modul MouseInteractor.



Obrázek 3.2-3: Modul Transform3DSource

Modul RotationTransformSource

Modul RotationTransformSource (Obrázek 3.2-4) je zdrojem transformační matice rotace. Můžeme zvolit, podle které osy se má rotovat a zda zadaná hodnota je udána ve stupních či radiánech. Úhel rotace lze vystavit jako vstupní port.



Obrázek 3.2-4: Modul RotationTransformSource

3.2.2 D3DUT

Funkcionalita poskytovaná rozhraním D3DUT pokryla požadavky této diplomové práce. Jediným přispěním bylo dopsání metody *LookAtLH()* v třídě *Matrix*. Z dodaných vektorů určujících pozici, směr a orientaci kamery metoda složí levotočivou pohledovou matici.

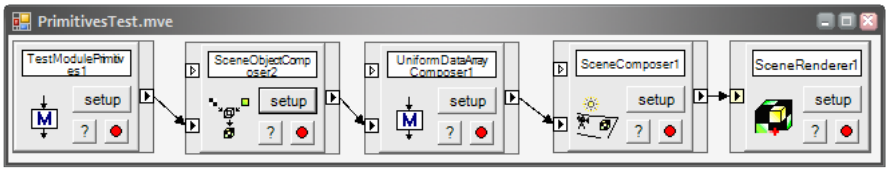
3.3 Výsledky a testování

Následující testy demonstrují funkčnost vytvořených modulů. U všech modulů bylo ověřeno, zda správně ukládají své nastavení, chovají se očekávaným způsobem při operacích kopírovat a vložit a zda spolupracují s moduly ostatních knihoven.

Testy probíhaly na laptopu Acer Aspire 1363WLMi, osazeném procesorem Mobile AMD Sempron 3000+, 512MB paměti RAM a grafickým akcelerátorem nVidia GeForce FX Go5200.

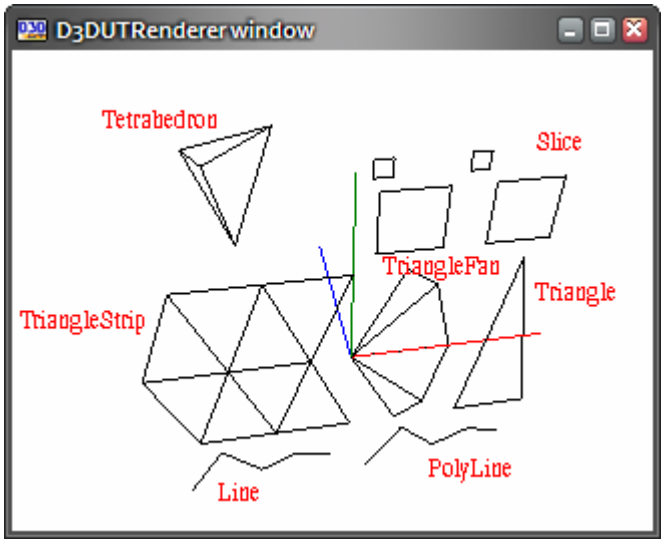
3.3.1 Vykreslení podporovaných primitiv

Pro otestování vykreslení všech podporovaných primitiv byl vytvořen modul TestModulePrimitives. Modul generuje dataset *UnstrGrid* a naplní jej. Je možné nastavit, zda se má dataset znovu generovat při každém spuštění metody modulu *Execute()*, případně v jakém časovém intervalu mají být data obnovována. Testovací mapa je vidět na obrázku (Obrázek 3.3-1).



Obrázek 3.3-1: Testovací mapa s modulem TestModulePrimitives

Výsledkem spuštění mapy je okno, ve kterém jsou vykreslena podporovaná primitiva (Obrázek 3.3-2). Správně jsou tedy zobrazeny *Line*, *PolyLine*, dvakrát *TriangleStrip*, *TriangleFan*, *Triangle*, *Tetrahedron* a *Slice*.



Obrázek 3.3-2: Výstup spuštění mapy testující zobrazení podporovaných primitiv

I na takto jednoduchém testu je možné provést měření výkonnosti. Testovaly se dva případy. V prvním byl dataset vygenerován při prvním spuštění metody

Execute() modulu TestModulePrimitives a při dalším spuštění byla už vystavena vygenerovaná data. V druhém případě se při každém spuštění modulu generoval dataset znovu.

Nastavení modulu TestModulePrimitives		Počet snímků za sekundu
Update	UpdateDelay	
False	----------	1664
True	0	1216

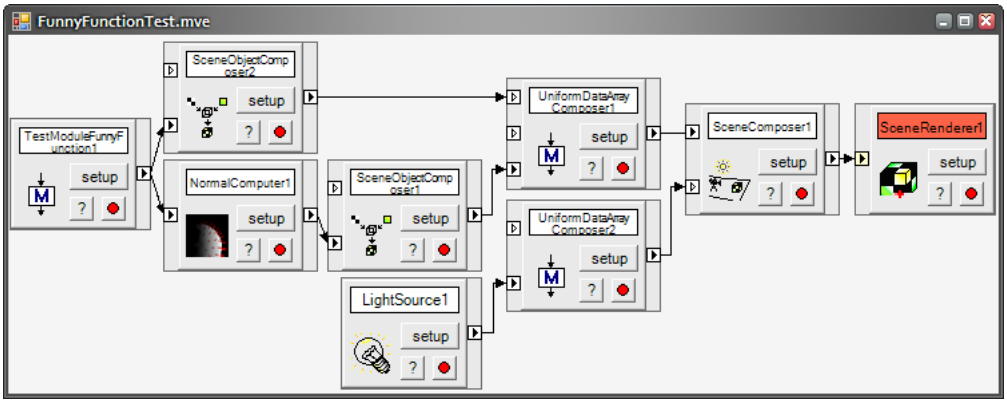
Tabulka 3.3-1: Porovnání počtu vykreslených snímků za sekundu při velikosti výstupního okna 320x240

Zpomalení při znovuvytváření datasetu je přibližně 35% (Tabulka 3.3-1), je zapříčiněno časem nutným k vygenerování datasetu a následného vytvoření verte bufferu a index bufferů. Měření není zcela objektivní, neboť se jedná o velmi jednoduchou scénu, kde počet vygenerovaných snímků za vteřinu je značný, tj. grafický akcelerátor je brzděn zbytkem systému. Na druhou stranu tento test dobře demonstruje účinnost mechanizmu, který zamezuje znovuvytvoření objektů, jejichž data nebyla změněna.

Zapneme-li zobrazování změny stavu modulu, počet vykreslených snímků za vteřinu poklesne z 1664 na 54. Takové drastické zpomalení ospravedlňuje dočasné řešení v podobě možnosti indikaci vypnout (3.2.1 MVE-2: MapEditor). Zároveň by bylo vhodné zjistit, zda je zobrazování mapy modulů v MapEditoru řešeno efektivně.

3.3.2 Scéna proměnná v čase

Druhý test má ukázat schopnost modulů knihovny UnstrGridRendering reagovat na data měnící se v reálném čase. Pro tento účel byl napsán modul TestModuleFunnyFunction. Modul generuje postupující funkci. Uchovává tedy dvourozměrné pole hodnot, přičemž hodnoty prvního řádku pole jsou vygenerovány upravenou funkcí kosinus, další řádky vždy kopírují hodnoty řádku předchozího. Je možné nastavit velikost pole a časový interval generování nového řádku, tj. jak rychle má graf funkce postupovat.

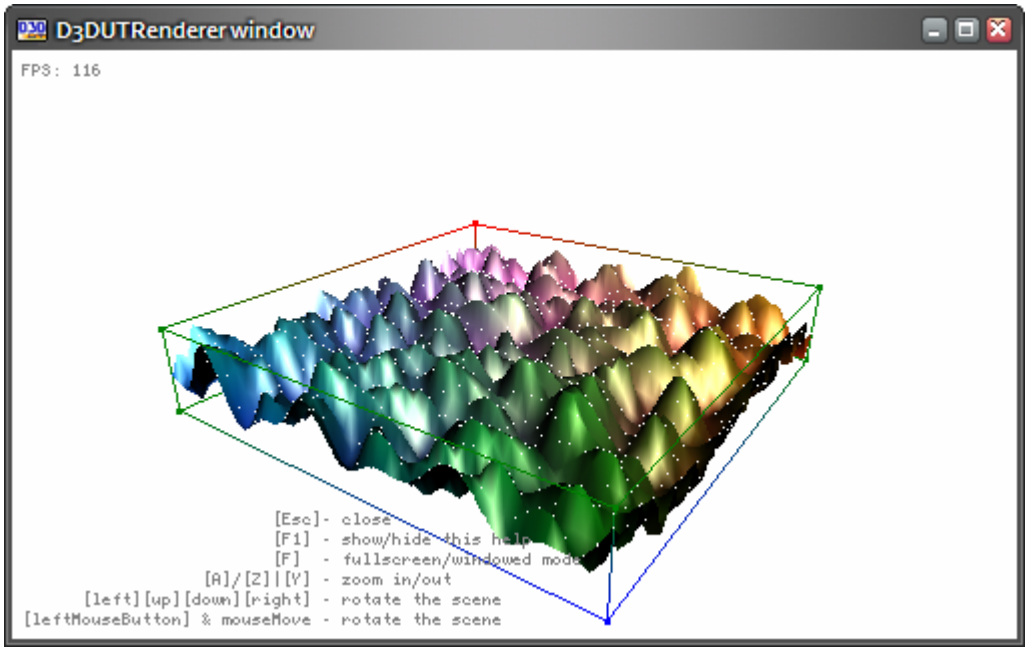


Obrázek 3.3-3: Testovací mapa s modulem TestModuleFunnyFunction

Výstup modulu TestModuleFunnyFunction1, dataset *UnstrGrid*, je přiveden na vstup modulu NormalComputer1. Ten k vygenerovanému povrchu vypočte normálové vektory a jako atributy je přidá do datasetu. Modul SceneObjectComposer1 z něj vytvoří *SceneObject* s parametry, které určují, že objekt bude vykreslen vystínovaně. Paralelně je modulem TestModuleFunnyFunction1

vygenerovaný dataset *UnstrGrid* přiveden do modulu SceneObjectComposer2, který mu nastaví vlastnosti tak, aby byly vykresleny pouze jeho vrcholy. Oba *SceneObjecty* jsou vloženy do pole objektů scény, to je zároveň s polem světel vloženo do scény, o jejíž vykreslení se stará modul SceneRenderer1.

Výsledek spuštění mapy je na obrázku (Obrázek 3.3-4). Zobrazená postupující funkce je složena ze dvou objektů. Jeden je obarvený a vystínovaný, druhý vykresluje pouze své vrcholy. Ve vykreslené scéně je také vidět bounding box, který je definován mezními vrcholy objektu.



Obrázek 3.3-4: Vykreslená postupující funkce

Tento testovací příklad ukázal na nepříjemnou vlastnost modelu toku dat. Časy potřebné ke zpracování úloh sériově zapojených modulů se sčítají. Je tedy vhodné psát metody *Execute()* modulů co nejefektivněji.

```
1.  public override void Execute()
2.  {
3.    if ((System.Environment.TickCount - lastTime) > updateDelay)
4.    {
5.      lastTime = System.Environment.TickCount;
6.
7.      while (backgroundWorker.IsBusy)
8.        System.Threading.Thread.Sleep(0);
9.
10.     triangleMesh = tmpTriangleMesh;
11.
12.     SetOutput("outTriangleMesh", triangleMesh, false);
13.     backgroundWorker.RunWorkerAsync();
14.   }
15.   else
16.     SetOutput("outTriangleMesh", triangleMesh);
17. }
```

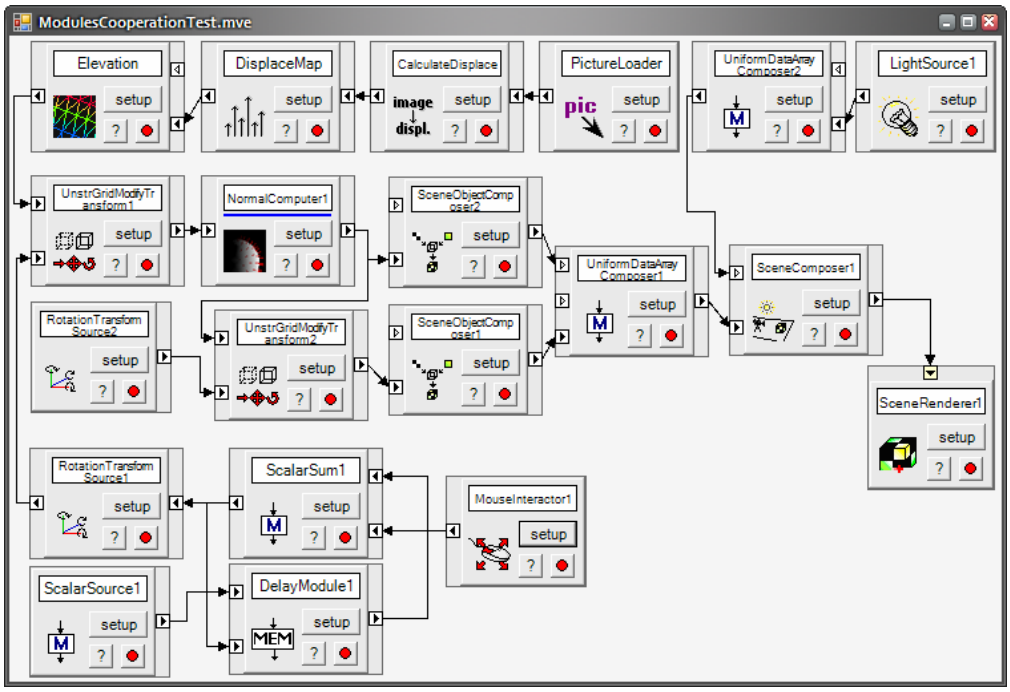
Kód 3.3-1: Metoda *Execute()* modulu TestModuleFunnyFunction

Předpočítání dat ve většině případů, kdy moduly fungují jako filtry, není možné, neboť nejdříve musí dostat data, která mají zpracovat. Z toho vyplývá, že předpočítání dat má smysl hlavně u modulů, které data generují. Ty pak v metodě *Execute()* již nemusí nic počítat a pouze vystaví připravená data.

Tak byla implementována i metoda *Execute()* modulu TestModule-FunnyFunction (Kód 3.3-1). V modulu jsou udržovány dva datasety. Jeden obsahuje data, která jsou aktuální a vystavována na výstupní port, druhý dataset je pouze dočasný, do něj jsou na pozadí napočítávány nové hodnoty. Ten se stane aktuálním po uplynutí daného časového intervalu, resp. dopočítání všech hodnot. Neuplynul-li časový interval posuvu grafu, metoda *Execute()* vystavuje aktuální dataset s příznakem *DataSame*. Po uplynutí časového intervalu posuvu se čeká na dokončení vlákna, které na pozadí počítá dataset nový. Vypočtený dataset, prozatím označovaný jako dočasný, se stane aktuálním, je vystaven na výstupní port s příznakem *DataChanged* a zároveň je spuštěno na pozadí nové vlákno, které počítá dataset následující.

3.3.3 Spolupráce s existujícími moduly

Tento test má na poměrně složité mapě (Obrázek 3.3-5) ukázat schopnost spolupráce modulů knihovny UnstrGridRendering s moduly ostatních existujících knihoven.



Obrázek 3.3-5: Testovací mapa demonstrující spolupráci s existujícími moduly z různých knihoven

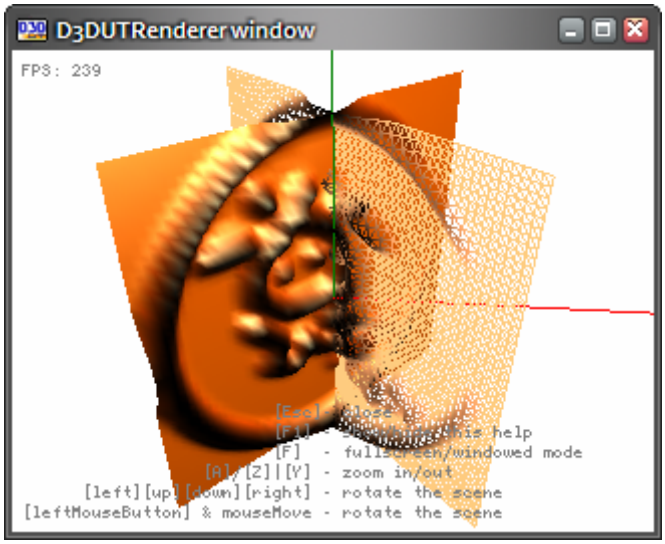
Moduly[1] CalculateDisplace, DisplaceMap a Elevation podle obrázku, který byl načten modulem PictureLoader, vygenerují a obarví dataset *TriangleMesh*.

[1] Tyto moduly byly vytvořeny studentem Jiřím Skálou v rámci předmětu GSVD, 2005.

Moduly ScalarSource1, ScalarSum1, DelayModule1 a MouseInteractor snímají pohyb myši a uchovávají hodnoty představující posunutí myši v horizontálním směru (3.1.4 Reakce na změny v zobrazované scéně). Tato hodnota je přivedena do modulu RotationTransformSource1, který ji interpretuje jako úhel a generuje matici rotace tohoto úhlu podle osy y. Transformační matice datasetu *TriangeMesh* je touto maticí nahrazena, následuje výpočet normálových vektorů a jejich vložení do scény.

Druhý objekt scény je vytvořen z téhož datasetu *TriangleMesh*, ovšem s maticí transformace vynásobenou maticí rotace podle osy y. Objekt je tedy vůči prvnímu otočen o 90°. Pro odlišení ve scéně mu je nastaven způsob výplně "drátěný model".

Do scény je vloženo světlo a scéna je předána k vykreslení modulu SceneRenderer. Výsledek je vidět na obrázku (Obrázek 3.3-6). Podle pohybu myší v horizontálním směru jsou objekty scény rotovány a přitom vůči sobě stále posunuty o 90°.



Obrázek 3.3-6: Výstup spuštěné mapy

Mapa je sice rozsáhlá, ale přesto zde nedochází ke zbytečnému znovuvytváření dat. Mohlo by se zdát, že modul UnstrGridModifyTransform1 modifikuje dataset *TriangleMesh*, a proto před vykreslením každého snímku musí modul NormalComputer1 znovu vypočítat jeho normálové vektory. Ve skutečnosti tak tomu není. Modul NormalComputer1 detekuje, že objekt typu *TriangleMesh* byl změněn podle příznaku *isDataSame*, ovšem podle posledního času modifikace pole bodů a buněk zjistí, že nedošlo ke změně v geometrii ani topologii objektu, a tudíž není třeba normálové vektory počítat znovu. Připomeňme, že modul UnstrGridModifyTransform pouze modifikuje matici transformace.

3.4 Možnosti rozšíření

Možných rozšíření knihovny UnstrGridRendering je mnoho. Uveďme tedy alespoň ty, se kterými se počítalo při návrhu spolu s nastíněním možného řešení.

3.4.1 Obarvování buněk

Představme si, že máme za úkol vymyslet algoritmus pro převedení trojúhelníkové sítě na několik pásů trojúhelníků, tzv. *TriangleStrip*. Taková úloha má své opodstatnění, neboť *TriangleStripy* jsou současnými grafickými akcelerátory vykreslovány rychleji, než osamocené trojúhelníky. Při vývoji algoritmu by zcela jistě přišla vhod možnost obarvovat jednotlivé pásy trojúhelníků.

Na úrovni datového modelu tento úkol není problém. dataset *UnstrGrid* obsahuje pole buněk a jim odpovídající atributy, kdy buňkami mohou být trojúhelníkové pásy a atributem barva.

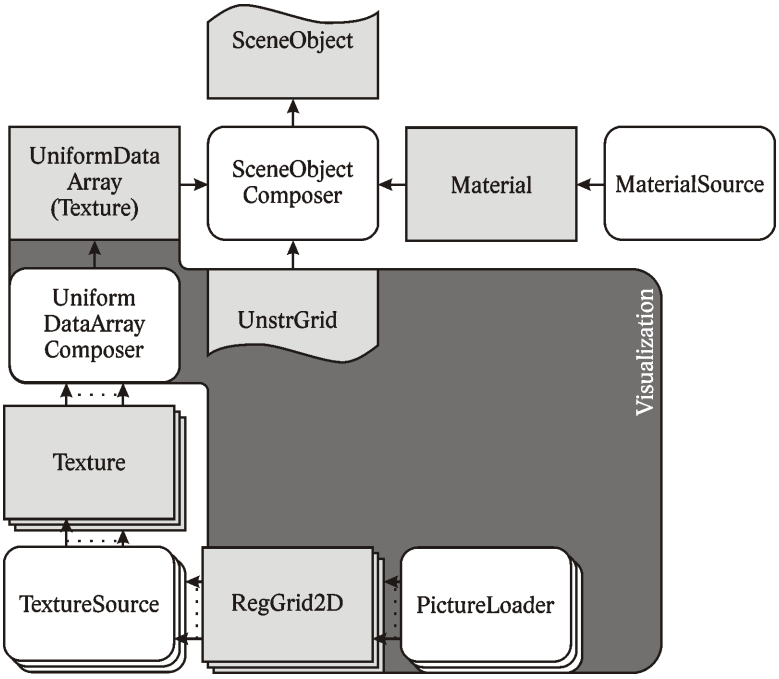
Jde tedy o samotné vykreslení obarvených buněk. Problém se komplikuje tím, že rozhraní Direct3D nemá metody pro obarvování primitiv. Chceme-li obarvit trojúhelník, existují dvě možnosti. Tou první je obarvit vrcholy trojúhelníku. V trojúhelníkové síti jsou ale vrcholy sousedících trojúhelníků sdíleny, což implikuje vytvoření duplicitních vrcholů. Druhou možností je použít materiál.

Možnost použití materiálu zavrhneme, neboť primitivum chceme pouze obarvit, nikoliv stínovat. Zbývá nám tedy možnost vytvoření duplicitních vrcholů. Při realizaci máme dvě možné cesty. První je přidat vytváření duplicitních vrcholů přímo do modulu SceneRenderer, druhou možností je vytvořit modul, který vygeneruje potřebné duplicitní vrcholy a barevné atributy buněk převede na atributy vrcholů. Takový dataset lze vykreslit stávajícím SceneRendererem.

3.4.2 Texturování

Textura je popisem vlastnosti povrchu tělesa a výrazně přispívá k realistickému vnímání vykresleného objektu scény. Bylo by tedy vhodné v budoucnu přidat do knihovny UnstrGridRendering moduly, které by umožňovaly na objekt scény aplikovat texturu.

Otázkou zůstává, do jaké míry chceme využít možností grafických akcelerátorů. Rozhraní Direct3D umožňuje na jedno primitivum aplikovat až osm textur, tj. každý vrchol může mít až osm texturovacích souřadnic. Mezi jednotlivými nanášenými texturami je možné provádět operace, jako jsou průhlednost, matematické operace nad barvami, aj.



Obrázek 3.4-1: Rozšíření kompozice scény o textury

Možnosti se nabízejí také v podporovaných rozměrech textur. Běžně se nanáší textury dvojrozměrné, je však možné i použití trojrozměrných textur. Co nejobecnější systém by se měl také vypořádat s různými možnostmi mapování, od klasického mapování rovinné textury přes souřadnice u, v, po mapování přes objekt, kterým může být krychle, koule, válec, apod.

Příjemné by bylo mít modul, který by umožňoval editovat jazyk pro pixel shader a verte shader grafického procesoru, např. HLSL.

Předkládaný návrh možného rozšíření (Obrázek 3.4-1) počítá s dvourozměrnými texturami a mapováním přes souřadnice u, v. Modul PictureLoader z knihovny Visualization by se postaral o načtení obrázku do datasetu *RegGrid2D*. Ten by samozřejmě mohl být generován i jiným modulem nebo skupinou modulů. Úkolem modulu TextureSource by bylo vygenerování datového objektu *Texture*. Ten by kromě datasetu *RegGrid2D* obsahoval informace o jménu atributu vrcholu *UnstrGridu* objektu scény, který nese texturovací souřadnice a možnostech prolnutí s dalšími texturami. Textury by byly přidány do pole modulem UniformDataArrayComposer a vloženy do datového objektu *SceneObject*.

3.4.3 Výstup pro stereoskopická zařízení

Člověk má dvě oči. Při pozorování okolního světa jimi mozek získává dva obrazy, ze kterých je schopen složit prostorový obraz a docílit tím prostorového vnímání.

Stereoskopická zařízení zprostředkovávají člověku prostorový vjem. Nejpřirozenějším způsobem je poskytnout pro každé oko jiný, posunutý obraz. Toho lze docílit několika způsoby, kdy nejčastější jsou brýle, které nějakým způsobem filtrují obraz promítaný na monitor či projekční stěnu. Takový obraz je přirozeně potřeba nejprve vygenerovat. V případě projekční stěny se jedná o dva proje-

ktory, pro které grafická karta pracovní stanice musí generovat obraz. Programátor aplikace pro vykreslování na dva videovýstupy grafické karty[1] využívá rozhraní, nejčastěji Direct3D či OpenGL.

Přidání podpory pro stereoskopickou stěnu v rozhraní D3DUT je plánováno. Modul SceneRenderer vykresluje právě přes rozhraní D3DUT a přidaní možnosti stereoskopického výstupu by tedy mělo být triviální.

3.4.4 Průhlednost

Dostupné grafické akcelerátory řeší pouze pohledově závislou průhlednost. Běžným trikem, který se používá, pokud jsou objekty uzavřené, konvexní a díváme se na ně z vnějšku, je vykreslení nejprve odvrácených a poté přivrácených trojúhelníků. V obecném případě je však vykreslovaná primitiva potřeba před vykreslením setřídit. To je časově náročné a pro interaktivní systém nevhodné. Vykreslování průhledných objektů, tedy nebylo implementováno a je zde prostor pro možná rozšíření.

[1] tzv. dualhead

4 Závěr

V rámci této diplomové práce byla napsána knihovna UnstrGridRendering pro modulární prostředí MVE-2. Moduly obsažené v této knihovně umožňují vykreslit scénu, která je kompozicí světel a objektů scény, objekt scény zaobaluje obecný dataset vyprodukovaný moduly jiných knihoven.

Nutným předpokladem k vytvoření takové knihovny byla znalost projektu MVE-2 a již vytvořených knihoven modulů. Bez detailního porozumění spouštěcího mechanismu map prostředí MVE-2 by nebylo možné knihovnu vytvořit. Stejně tak bylo nutné se seznámit s rozhraním D3DUT, které zajišťuje akcelerovaný grafický výstup. Oba zmíněné projekty jsou stále ve vývoji a bylo do nich přispíváno.

Splněn byl požadavek na interaktivitu subsystému. Modul zajišťující vykreslení scény reflektuje změny v ní provedené za běhu. Pro docílení lepší odezvy byl v jádru MVE-2 zaveden nový systém zjišťování aktuálnosti dat, díky kterému je možné detekovat, které části kompozice byly změněny, a zamezit tak neefektivnímu znovuvytvoření objektů. Důraz byl také kladen na spolupráci s existujícími moduly.

Diplomová práce je přirozeně omezena svým rozsahem. Bylo tedy rozebráno alespoň několik dalších možných rozšíření a nástin jejich realizace.

Literatura[1]

[CLI] *Standard ECMA-335, Common Language Infrastructure (CLI),* [WWW] http://www.ecma-international.org/publications/standards/Ecma-335.htm

[D3DUT] *Direct3D Utility Toolkit*, [WWW] http://herakles.zcu.cz/research/projects/12/index.php

[IRIS] *IRIS Explorer*, [WWW] http://www.nag.co.uk/Welcome_IEC.asp

[KACM] Kačmář D.: *Programujeme .NET aplikace ve Visual Studiu .NET*, Computer Press, Praha, 2001

[LVIEW] *LabView*, [WWW] http://www.ni.com/labview

[MDX9] Miller T.: *Managed DirectX® 9 Kick Start: Graphics and Game Programming*, Sams Publishing, Indianapolis, 2003

[MD3D] *Direct3D for Managed code documentation,* [WWW] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_m/directx/directx9m.asp

[MPGC] *Getting the Most Out of the .NET Framework PropertyGrid control*, [WWW] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/usingpropgrid.asp

[MVE] *Modular Virtual Environment*, [WWW] http://herakles.zcu.cz/research/projects/13/index.php

[1] řazeno abecedně

[MVE2] *Modular Virtual Environment 2*,
[WWW] http://herakles.zcu.cz/research/projects/11/index.php

[ODX] *OpenDX*,
[WWW] http://www.opendx.org

[OGL] *OpenGL*,
[WWW] http://www.sgi.com/products/software/opengl/

[OINV] *Open Inventor*,
[WWW] http://oss.sgi.com/projects/inventor/

[REFL] *Reflector for .NET*,
[WWW] http://www.aisto.com/roeder/dotnet/

[SCHR] Schreder W., Avila L., Martin K., Hoffman W., Law C.:
The VTK User's Guide, Prentice Hall, New Jersey, 2001

[TPLB] *The Programming Language B*,
[WWW] http://cm.bell-labs.com/cm/cs/who/dmr/bintro.html

[VQST] *VisiQuest*,
[WWW] http://www.accusoft.com/imaging/visiquest

[VTK] *The Visualization Toolkit*,
[WWW] http://www.vtk.org

[VTKNET] *VTK for .NET platform,*
[WWW] http://herakles.zcu.cz/research/vtk.net/index.php

[ZARA] Žára J., Beneš B., Sochor J., Felkel P.:
Moderní počítačová grafika, Computer Press, Brno, 2004

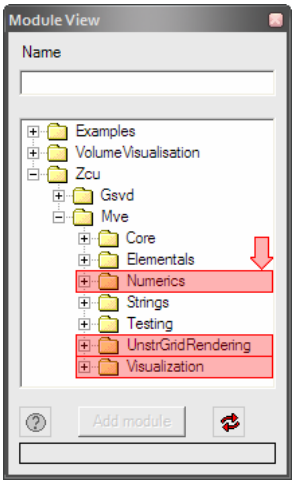
Přílohy

Příloha A: Uživatelský manuál

Tato příloha provede uživatele instalací knihovny UnstrGridRendering do prostředí MVE-2, nastaveními, které jsou vhodná provést v MapEditoru, a vytvořením vizualizace dat s využitím většiny modulů knihovny.

A.1 Instalace

Doporučený způsob instalace je zkopírování knihovny *UnstrGridRendering.dll* spolu s knihovnami D3DUT, *d3dut-win32.dll* a *Zcu.Graphics.D3DUT.dll*, do adresáře *.\libs* prostředí MVE-2. Knihovna *UnstrGridRendering.dll* je závislá na knihovnách *Visualization.dll* a *Numerics.dll*. I ty by se měly vyskytovat v adresáři *.\libs*. Není-li tomu tak, je třeba je tam nakopírovat.

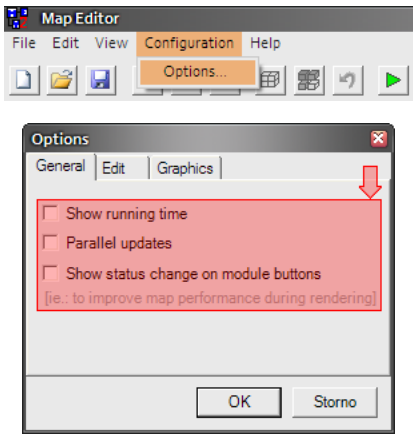


Obrázek A.1-1: Potřebné knihovny modulů nahrané v MapEditoru a zobrazené v okně ModuleView

Spustíme-li MapEditor, je možné se v okně ModuleView přesvědčit, zda-li knihovny byly nahrány (Obrázek A.1-2).

A.2 Nastavení prostředí MapEditoru

Pro dosažení co nejlepší odezvy při vykreslování scény je vhodné vypnout v nastaveních MapEditoru zobrazování doby běhu modulů, paralelní spouštění modulů a zobrazování změny stavu modulu (Obrázek A.2-3).



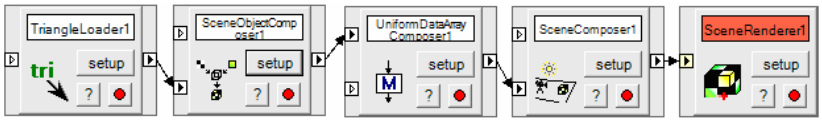
Obrázek A.2-4: Nastavení MapEditoru

A.3 Tutoriál – Vizualizace lebky

Návod nás v několika krocích provede vizualizací načteného .tri souboru.

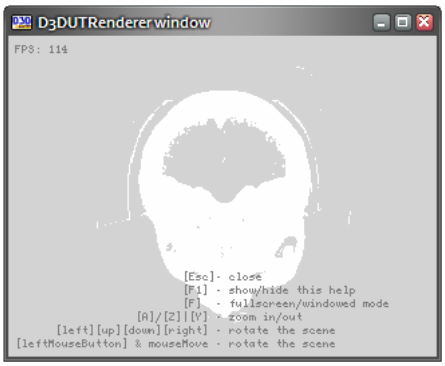
Krok 1: Načtení .tri souboru

Nejprve sestavíme mapu pro načtení a zobrazení .tri souboru. V MapEditoru zvolíme vytvořit novou mapu. Do prázdné mapy vložíme moduly SceneObject-Composer, SceneComposer a SceneRenderer z knihovny UnstrGridRendering, dále pak TrangleLoader a UniformDataArrayComposer z knihovny Visualization. Moduly spojíme, jak je vidět na obrázku (Obrázek A.3-5). Standardní nastavení změníme pouze u modulu TriangleLoader, kde nastavíme cestu k souboru *ctmayo.tri*.



Obrázek A.3-6: Krok 1, mapa

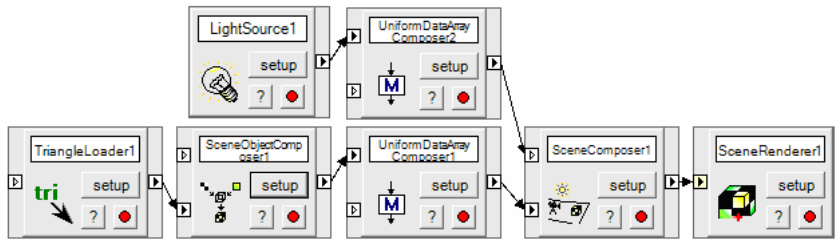
Mapu spustíme a výstupem je nevystínovaná lebka (Obrázek A.3-2).



Obrázek A.3-2: Krok 1, výstup

Krok 2: Stínování

Nyní lebku vystínujeme. Do mapy vložíme modul LightSource z knihovny UnstrGridRendering a modul UniformDataArrayComposer z knihovny Visualization (Obrázek A.3-3).



Obrázek A.3-3: Krok 2, mapa

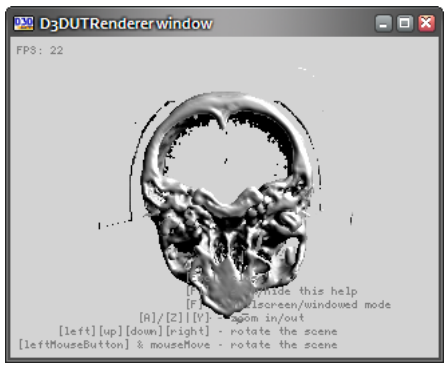
Vlastnosti světla v modulu LightSource1 změníme na:

- *Position*: 2; 2; -2
- *Type*: *Point*

Vlastnosti objektu scény v modulu SceneObjectComposer1 změníme na:

- *ColorVertex*: *False*
- *Lighting*: *True*
- *MaterialEnable*: *True*
- *PrefferedVertexAttributes*: *Normal3D*

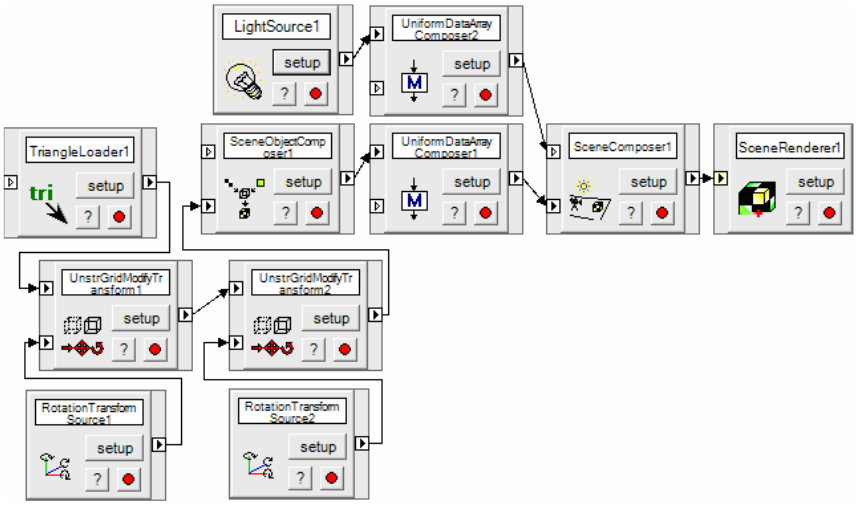
Výsledkem spuštění mapy je vystínovaná lebka (Obrázek A.3-4).



Obrázek A.3-4: Krok 2, výstup

Krok 3: Transformace

Nelze si nevšimnout, že na lebku se díváme zespodu. Bylo by tedy lepší ji otočit tak, aby se hned po spuštění mapy se na nás dívala čelem. Vložme tedy do mapy moduly RotationTransformSource a UnstrGridModifyTransform (Obrázek A.3-5). Modulem TriangleLoader1 načtený dataset bude nejprve rotován o -90° podle osy x, poté o 180° podle osy y.



Obrázek A.3-5: Krok 3, mapa

Nastavení modulu RotationTransfomSource1 změníme na:

- *Angle*: -90

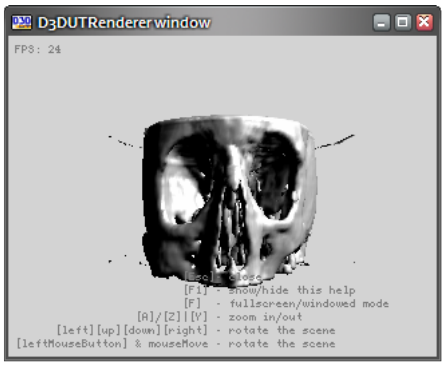
Nastavení modulu RotationTransformSource2 změníme na:

- *Angle*: 180
- *RotationAxis*: Y

Nastavení modulu UnstrGridModifyTransform2 změníme na:

- *TransformOperation*: *Multiply*

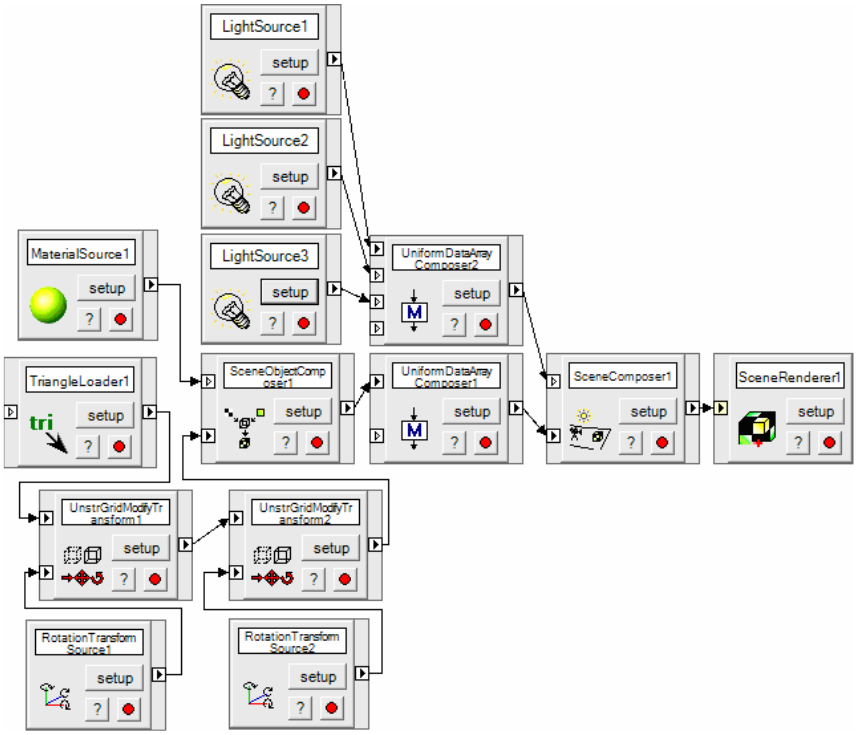
Po spuštění mapy je již lebka otočena tak, jak bychom čekali (Obrázek A.3-6).



Obrázek A.3-6: Krok 3, výstup

Krok 4: Materiál

Vykreslený obrázek ještě můžeme vylepšit materiálem a přidáním dalších světel. Vložme tedy do scény moduly MaterialSource a další moduly LightSource (Obrázek A.3-7).



Obrázek A.3-7: Krok 4, mapa

Vlastnosti materiálu změníme v modulu MaterialSource1 na:

- *Diffuse*: 255; 255; 192
- *Specular*: 224; 224; 224
- *SpecularSharpness*: 1000

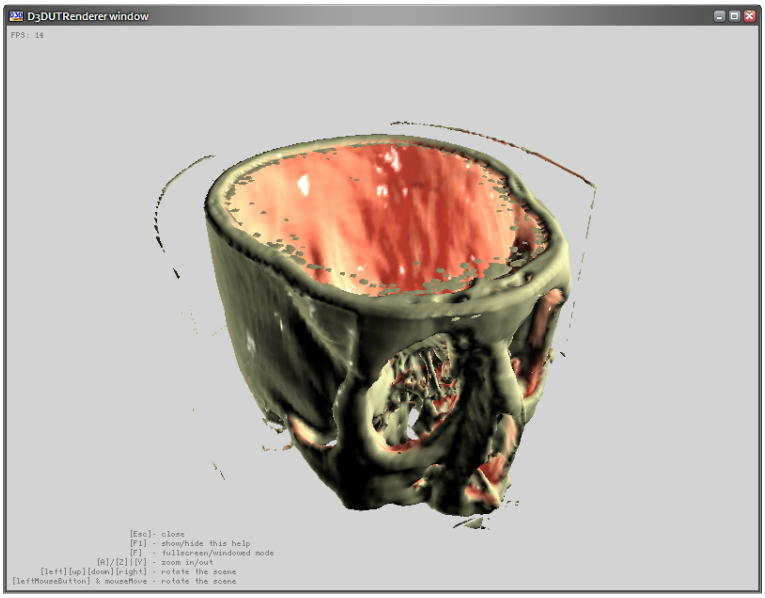
Vlastnosti druhého světla změníme v modulu LightSource2 na:

- *Position*: -2; -2; 2
- *Type*: *Point*

Třetím, červeným světlem nasvítíme vnitřek lebky. Změníme tedy nastavení v modulu LightSource3 na:

- *Type*: *Point*
- *Diffuse*: 128; 0; 0

Výsledek, se kterým již můžeme být spokojeni je vidět na obrázku (Obrázek A.3-8).



Obrázek A.3-8: Krok 4, výstup

Příloha B: Přiložené CD

Obsah přiloženého CD:

- *DiplomaThesis_Text* – Text diplomové práce ve formátech .pdf a .doc.
- *UnstrGridRendering_Source* – Zdrojový kód implementované knihovny UnstrGridendering.
- *MVE2_Test* – Spustitelná verze MVE-2 s nainstalovanou knihovnou UnstrGridRendering, testovacími mapami a daty.

Verze CD pro interní použití Katedry Informatiky Západočeské Univerzity v Plzni navíc obsahuje:

- *MVE2_Source* – Atualní zdrojový kód MVE-2.
- *D3DUT_Source* – Aktuální zdrojový kód D3DUT.

Evidenční list

Souhlasím s tím, aby moje diplomová práce byla půjčována k prezenčnímu studiu v Univerzitní knihovně ZČU v Plzni.

Datum: Podpis:

Uživatel stvrzuje svým čitelným podpisem, že tuto diplomovou práci použil ke studijním účelům a prohlašuje, že ji uvede mezi použitými prameny.

Jméno	**Fakulta/katedra**	**Datum**	**Podpis**