TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: B 2612 - Elektronika a informatika
Studijní obor: 2612R011 - Elektronické informační a řídící systémy

3D aplikace pracující v prostředí internetu v jazyce Java

3D application developed for the internet in Java

Bakalářská práce

Autor: **Michal Kašpárek**
Vedoucí BP práce: Ing. Roman Špánek

V Liberci 18. 5. 2007

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé BP a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum

Podpis

Předmluva

Při zpracování zadání jsem během seznamování se s jazykem Java a s problematikou trojrozměrné animace prošel několika slepými vývojovými fázemi. Poznatky, které jsem během návrhu aplikace nashromáždil, se snažím prezentovat formou srozumitelnou pro čtenáře se zájmem o tuto softwarovou oblast a s alespoň základními znalostmi programování.

První část textu obsahuje nezbytný teoretický základ o Javě, prostředí NetBeans IDE, standardu Gmsh a zejména o problematice trojrozměrné grafiky v Javě 3D. Část druhá je koncipována tak, aby kdokoliv s určitou elementární znalostí Javy mohl zopakovat kteroukoliv etapu vývoje programu, ať už by se jednalo o návrh uživatelského prostředí, hru v Javě 3D či zpracování souboru Gmsh.

Poděkování

Na tomto místě bych rád poděkoval všem, kteří nějak přispěli ke zdařilému dokončení této práce, a to zejména vedoucímu mé bakalářské práce Ing. Romanu Špánkovi za cenné rady, impulzy a ochotu, se kterou věnoval svůj čas konzultacím. Dále bych chtěl poděkovat svému otci Ing. Vratislavu Kašpárkovi a dědečkovi Doc. Ing. Jaromíru Kašpárkovi, CSc. FTI (hon) za podporu a připomínky k textu této zprávy. V neposlední řadě patří velký dík Kačence, bez jejíž podpory a zázemí by tato práce jistě nevznikla tak hladce.

Abstrakt

Tato bakalářská práce se zabývá problematikou trojrozměrné grafiky v programovacím jazyce Java. Konkrétně jsou použity knihovny Java 3D. Navržené metody animace a transformace scény jsou poté aplikovány na zobrazení trojrozměrných konečně-prvkových souřadnicových sítí. V textu jsou vysvětleny teoretické základy, které jsou následně prakticky aplikovány na program v jazyce Java.

Abstract

This bachelor thesis deals with three dimensional graphic programming in Java language. Particularly Java 3D library modules are used. Designed methods for animation and scene transformation are applied to display three dimensional finite element grid of coordinates. The following text explains theoretical basics which are then adopted to Java program.

Obsah

1. Programovací jazyk JavaTM – Úvod

Java [výslovnost: džava] je programovací jazyk vyvinutý firmou Sun Microsystems (homepage: http://java.sun.com); poprvé byl představen 23. května 1995.

Jedná se o programovací jazyk, který je díky svému open-source statutu oblíben a používán nejen internetovou komunitou po celém světě. Díky své přenositelnosti je využíván na různých systémech a platformách.

Java je objektově orientovaný jazyk vycházející z C++, který však během svého vývoje prošel několika podstatnými vylepšeními. Neobsahuje některé problematické konstrukce (např. ukazatele) a přináší několik zajímavých a užitečných vlastností:

- *Garbage Collector* se automaticky stará o přidělování a uvolňování(!) paměti – programátor se již nemusí zabývat procedurami typu `free`.
- Je implementován mechanizmus vláken (*threads*).
- Je implementován mechanizmus *výjimek*, runtime chyby lze odchytit a zpracovat.
- Součástí jsou rozsáhlé standardně dodávané *knihovny* např. pro tvorbu grafického rozhraní.
- Je kladen velký důraz na *bezpečnost* – integrované bezpečnostní mechanizmy (přidělování práv pro různé akce aj.) chrání počítač, na kterém je program zpracováván před napadením nepřátelským kódem.
- *Jednoduchost a přehlednost*.

Další kapitolou hovořící ve prospěch Javy je její *hardwarová nezávislost*. Program je nejprve „předzpracován“ na tzv. bajtkód (*bytecode*), který je na koncovém uživatelském počítači překládán do nativního (strojového) kódu pomocí *Just-In-Time* překladače. Stejný program v jazyce Java lze tedy spustit na libovolné platformě (např.: Windows, MacIntosh, Linux atd.), kde je k dispozici *Java runtime.*

Mezi *nevýhody* Javy patří pomalejší start programů způsobený *Just-In-Time* kompilací a větší paměťová náročnost způsobená nutností mít v paměti celé *run-time* prostředí.

Slovo „java“ pocházející z americké slangové angličtiny znamená v překladu „kafe“. Vznik tohoto jazyka je spojen s rozmachem sítě Internet, kde se můžeme s malými Java Applety setkat nejčastěji. Java však není určena výhradně pro prostředí Internetu, ale je plnohodnotným jazykem obecně použitelným pro tvorbu běžných aplikací.

Jedním z poměrně rozšířených mýtů je zaměňování Javy a JavaScriptu. Přestože syntakticky jsou si velmi podobné, existují mezi nimi velké rozdíly. JavaScript pochází od firmy Netscape a je primárně určen pro WWW stránky.

1.1. JRE, JDK – co znamenají?

Instalační balíčky Javy, které jsou volně ke stažení na stránkách Sun Microsystems, se dělí na dva typy:

- *Java Runtime Enviroment (JRE)*, tedy prostředí pro běh javovských programů, které se skládá ze standardních knihoven *(Java Core API)* a kompilátoru, interpretujícího univerzální *bytecode* pro potřeby uživatelské platformy *(Java Virtual Machine).*
- *Java Development Kit (JDK)* je vývojový balíček obsahující JRE, překladač do bajtkódu a další vývojové nástroje (*debugger, javadoc* aj.).

1.2. Číslování verzí:

Číslování verzí JRE a JDK je shodné a má tento význam: *JDK 1.4.0*

- 1 → Major verze, od nedávné doby vynechávané
- 4 → Minor verze, mění se vždy, když dojde ke změnám či rozšířením v jazyce nebo v knihovnách
- 0 → Bug Fix Numer, čím vyšší má hodnotu, tím méně obsahuje chyb (tím více jich je opravených)

Od roku 2006 se Sun rozhodl změnit tvar číslování verzí do následující podoby:
Od verze 1.5.0 se následující vydání číslují ve tvaru 5.0.

Poslední verzí JDK je **Java(TM) SE Development Kit 6**.

Dále rozlišujeme tři verze JDK:

- Java Platform, **Enterprise Edition** *(Java EE)*: je průmyslový standard pro vývoj bezpečných a stabilních síťových aplikací.
- Java Platform, **Standard Edition** *(Java SE)*: také známá jako Java 2 Platform, umožňuje vytvářet aplikace pro desktopová a serverová řešení.
- Java Platform, **Micro Edition** *(Java ME)*: edice určená speciálně pro psaní aplikací do mobilních zařízení jako jsou mobilní telefony, PDA apod.

1.3. Konvence psaní v Javě

Fontem `Courier New` budeme označovat části kódu.

V Javě se doporučuje používat několik přesně definovaných typografických konvencí (pokud se je programátor rozhodne nedodržovat, přeložitelnost kódu tím samozřejmě nebude ohrožena, ovšem přenositelnost a srozumitelnost tím utrpí nezanedbatelnou újmu):

- *Třída, rozhraní, konstruktor*: první slovo v sousloví začíná vždy velkým písmenem, například: **`String, StringBuffer`**.
- *Metody a proměnné:* identifikátor se skládá z malých písmen. U složených jmen začíná každé slovo kromě prvního velkým písmenem: **`start`**(), **`getSize`**().
- *Balíky (packages):* identifikátor se skládá jen z malých písmen, ve složených jménech je oddělovačem tečka: **`java.io`**
- *Konstanty:* identifikátor se obvykle skládá pouze z velkých písmen. Ve víceslovných identifikátorech je oddělovačem podtržítko: **`MAX_VALUE.`**

Java je tzv. *case-sensitive* jazyk, což znamená, že rozlišuje malá a velká písmena. Další důvod, proč dodržovat typografické konvence.

1.4. Aplikace versus Applet

1.4.1. Aplikace:

Aplikace je samostatný program, který ovšem vyžaduje pro svůj běh Java Platformu, nikoliv prohlížeč jako applet. Na aplikaci obecně nejsou kladena bezpečnostní omezení (může zapisovat do souboru – u appletu se jedná o preventivní opatření proti nežádoucímu kódu jako jsou viry apod.).

Příklad jednoduché aplikace:

```
/**
 * Toto je komentar.
 * Aplikace pouze vypisuje text do standardního systémového
 * vystupu. Prevzato z tutorialu java.sun.com
 */
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
               // Toto je komentar do konce radku
    }
}
```

Struktura aplikace:

Hlavní část je uzavřena ve veřejné třídě, jejíž identifikátor je uveden klíčovými slovy **`public class`** a která se musí nacházet v souboru *identifikator.java*, tedy v tomto případě *HelloWorldApp.java.*

Tato třída *musí* nezbytně obsahovat metodu **main()**, která se skládá z jednotlivých příkazů. Jednotlivé bloky jsou uzavřeny do složených závorek jako například v Pascalu BEGIN a END.

Zdrojový soubor **.java* přeložíme z příkazového řádku **javac HelloWorldApp.java**, vznikne soubor **HelloWorldApp.class** obsahující přenositelný bytový kód (byte-code). Tento můžeme spustit též z příkazového řádku pomocí interpreta:

java HelloWorldApp.

Výsledkem bude výpis textu Hello World.

1.4.2. Applet:

Applet je program určený pro umístění na WWW server, kde je pomocí speciální značky včleněn do HTML dokumentu tvořícího WWW stránku. Při návštěvě této stránky Java-kompatibilním prohlížečem se applet automaticky nahraje do klientského počítače, kde se spustí. Applety bývají většinou kratší, aby se příliš neprodlužovala doba nahrávání WWW stránky přes síť.

Z důvodu bezpečnosti platí pro applety některá omezení. Applet například nemůže zapisovat do souborů na straně klienta (prohlížeče) či spouštět programy na straně klienta.

Ukázkový applet načte a zobrazí obrázek:

```
//prevzato z http://dione.zcu.cz/java/sbornik/4.html
import java.applet.Applet;
import java.awt.*;
public class Obrazek extends Applet {
   Image obrazek = null;      //deklarace promenne obrazek

public void init() {
obrazek = getImage(getCodeBase(), getParameter("obrazek"));
}
public void paint(Graphics g) {
      if (obrazek != null)
         g.drawImage(obrazek, 0, 0, this);
   }
}
```

Struktura appletu:

Hlavní část appletu je opět uzavřena ve veřejné třídě, která je v tomto případě potomkem Appletu (předka): **`extends Applet.`** Má proto odlišnou strukturu než aplikace. Hlavní částí třídy již není metoda **`main()`**, nýbrž metody **`init()`**, kde se provádí inicializace appletu, a **`paint()`**, která se volá při požadavku na překreslení.

Na začátku kódu vidíme příkazy **`import`**, které pouze informují překladač na nutnost importu jmenovaných tříd, což umožní použití zkrácených jmen: **`Applet`** místo **`java.applet.Applet`**.

Soubor **Obrazek.java** je nutné přeložit stejně jako výše zmíněnou aplikaci, ke spuštění appletu však potřebujeme ještě HTML dokument obsahující odkaz na náš přeložený applet.

Možný kód souboru Obrazek.html:

```
<APPLET
     CODE = Obrazek.class
     HEIGHT = 100
     WIDTH = 100
>
<PARAM
     NAME = obrazek
     VALUE = jménoSouboru
>
   Váš prohlížeč nespouští Java applety.
</APPLET>
```

První část kódu obsahuje odkaz na přeložený applet a parametry oblasti, kam se bude applet zobrazovat. Ve druhé části je třeba parametr *jmenoSouboru* nahradit jménem zobrazovaného obrázku ve formátu GIF nebo JPEG, který se musí nacházet ve stejném adresáři.

Narozdíl od třídy v Javě se HTML soubor nemusí jmenovat stejně jako vytvořený applet.

1.5. Obecná struktura programu

Zdrojový text programu se skládá z jedné nebo více tříd a rozhraní. Java (na rozdíl od například Delphi) nezná globální proměnné.

Každá *veřejná* třída musí být uložená v samostatném, stejně se jmenujícím souboru s příponou **.java**, toto se netýká pouze tzv. vnořených tříd.

Neveřejné třídy je možné umístit do jednoho souboru společně s jednou třídou veřejnou (viz. následující ukázka obecného kódu) nebo do samostatného, libovolně pojmenovaného souboru ***.java**.

Třídy můžeme dál seskupovat do balíčků za pomoci klíčového slova **`package`**. Tato vlastnost je velmi užitečná u velkých programů obsahujících mnoho tříd, jejichž struktura je takto mnohem přehlednější.

Formát zdrojového souboru vypadá obecně takto:

```
package jménoBalíku ;


import argument1;
import argument2;
// ...
import argumentN;


interface Rozhraní1 { /* tělo rozhraní 1 */ }
interface Rozhraní2 { /* tělo rozhraní 2 */ }
// ...
interface RozhraníN { /* tělo rozhraní N */ }


class Třída1 { /* tělo třídy 1 */ }
class Třída2 { /* tělo třídy 2 */ }
// ...
class TřídaN { /* tělo třídy N */ }


public class VeřejnáTřída { /* tělo veřejné třídy */}

      // nebo:

public interface VeřejnéRozhraní {
          /*tělo veřejného rozhraní*/
          }
```

1.6. Běh programu

Kód programu, obsažený v jednotlivých ***.java** souborech, je třeba nejdřív přeložit Java Compilatorem (**javac *.java**). Po spuštění (**java** *JmenoSouboru*) se nejprve nahraje do paměti třída v daném ***.class** souboru a proběhne verifikace jejího *bytového kódu*.

V případě aplikace je poté spuštěna metoda **`main()`** s parametry, přijatými z příkazové řádky. Běh appletu dále řídí kompatibilní prohlížeč.

Další třídy, které program používá, jsou nahrávány za běhu dle potřeby programu a i u nich se provádí verifikace.

Program končí, pokud je vykonána metoda **`exit()`** ze třídy **`Runtime`**, případně pokud všechna zbývající vlákna jsou *démoni* (vlákno, které pouze poskytuje služby ostatním threadům) a je ukončena metoda **`main(),`** případně je zavolána metoda **`stop()`** u appletu.

Díky tomu, že třídy jsou uchovávány v samostatných souborech, mohou se chovat podobně jako dynamické knihovny *dll*, neboť každá veřejná třída definuje rozhraní metod, jež mohou být používány i z jiných programů. Tak lze snadno rozšiřovat existující aplikace, vytvářet rozhraní pro plug-iny apod.

2. NetBeans IDE

2.1. Úvod

NetBeans IDE (Integrated Development Environment) je integrované vývojové prostředí pro Javu, které původně vzniklo na MFF UK v Praze (www.mff.cuni.cz), poté bylo v červnu 2000 koupeno samotným Sun Microsystems a po nějaké době uvolněno jako OpenSource, což přispělo k jeho rychlému rozvoji.

Prostředí NetBeans je právem označováno jako multiplatformní IDE, neboť jej lze používat na všech platformách, kde funguje JDK (tedy kromě MS Windows či Linux také například Apple Macintosh nebo HP-UX aj.).

Internetová domovská stránka projektu je www.netbeans.org.

Dnes existují dva produkty: vývojové prostředí NetBeans (NetBeans IDE) a vývojová platforma NetBeans (The NetBeans Platform), což je modulární a rozšiřitelný základ pro použití při vytváření rozsáhlých aplikací. Oba projekty jsou vyvíjeny jako Open Source a lze je zdarma využívat pro komerční i nekomerční účely.

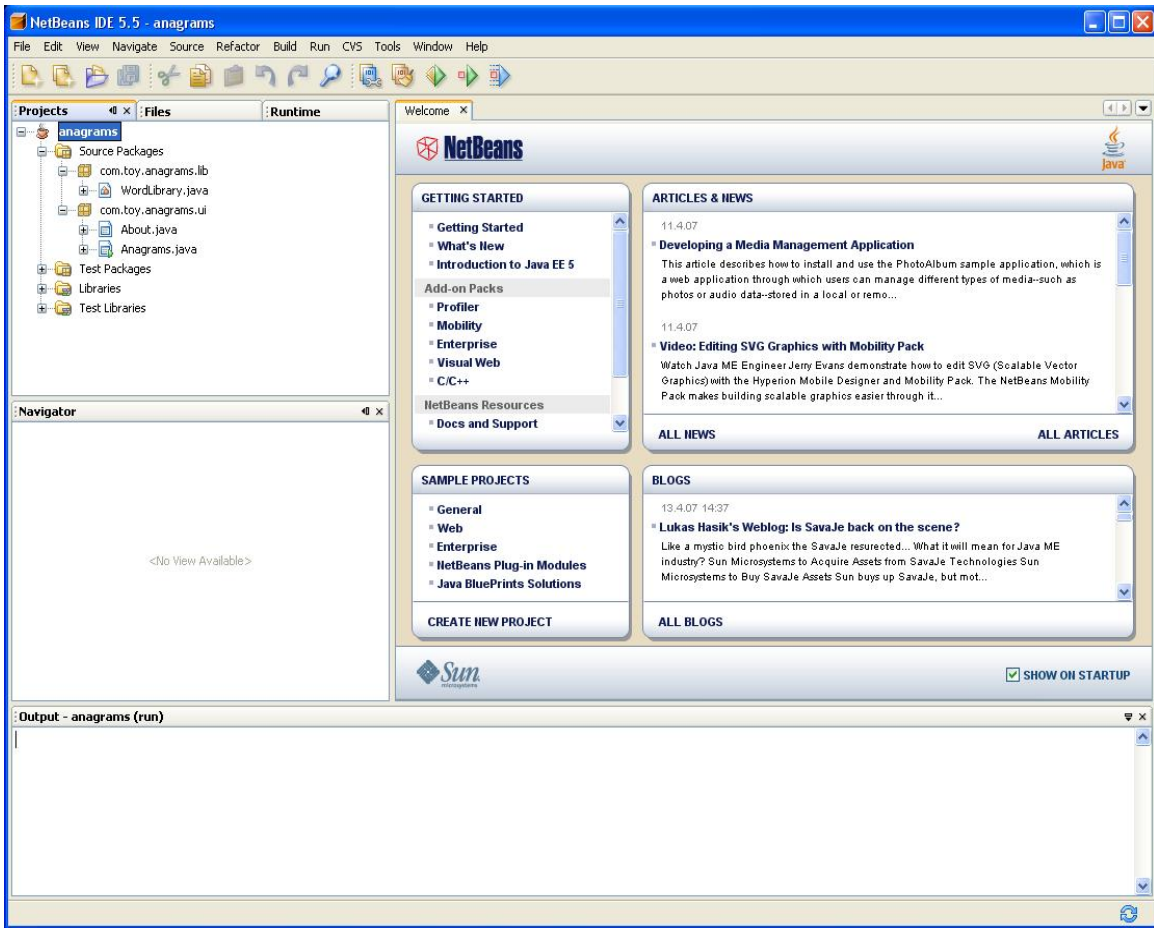
Za pomoci nástroje NetBeans IDE mohou programátoři psát, překládat a ladit aplikace. Vývojové prostředí je vytvářeno v jazyce Java, ale může podporovat jakýkoliv programovací jazyk. Kromě toho také existuje velké množství modulů, které toto prostředí rozšiřují.

Nejnovější verzí (duben 2007) je NetBeans 5.5, které, ač původně vyvinuto výhradně pro práci v jazyce Java, navíc obsahuje klienta CVS (Concurrent Version System, sloužící ke správě verzí projektu), XML editor (eXtensible Markup Language, česky: Rozšiřitelný značkovací jazyk, určen především pro výměnu dat mezi aplikacemi) či nově i podporu pro programování v jazyce C/C++.

Vývojové prostředí NetBeans vzniká z podstatné části v Pražském vývojovém centru společnosti Sun Microsystems, překlady jsou pak výsledkem práce skupin uživatelů

Javy po celém světě. Na lokalizaci do češtiny se v současné době pracuje. Do projektu se může každý zájemce zapojit na stránkách http://translatedfiles.netbeans.org/index_cs.html.

Pokud se ptáte na oblíbenost a rozšířenost tohoto IDE, pak řečí čísel zaznamenal Sun do současné doby přes deset miliónů stažení ze svých webových stánek. Případnou alternativou k NetBeans může být *Eclipse* (www.eclipse.org), *IntelliJ IDEA* (www.jetbrains.com/idea), případně *Jbuilder* (www.codegear.com/Products/JBuilder).



obr. 1: NetBeans

Záznam přednášky Romana Štrobla *NetBeans IDE - Vývoj Java aplikací v NetBeans IDE 5.0* je k dispozici ke stažení na stránkách ČVUT: http://avc.sh.cvut.cz/archiv/index.php?id=1041&rid=334

2.2. Layout manager

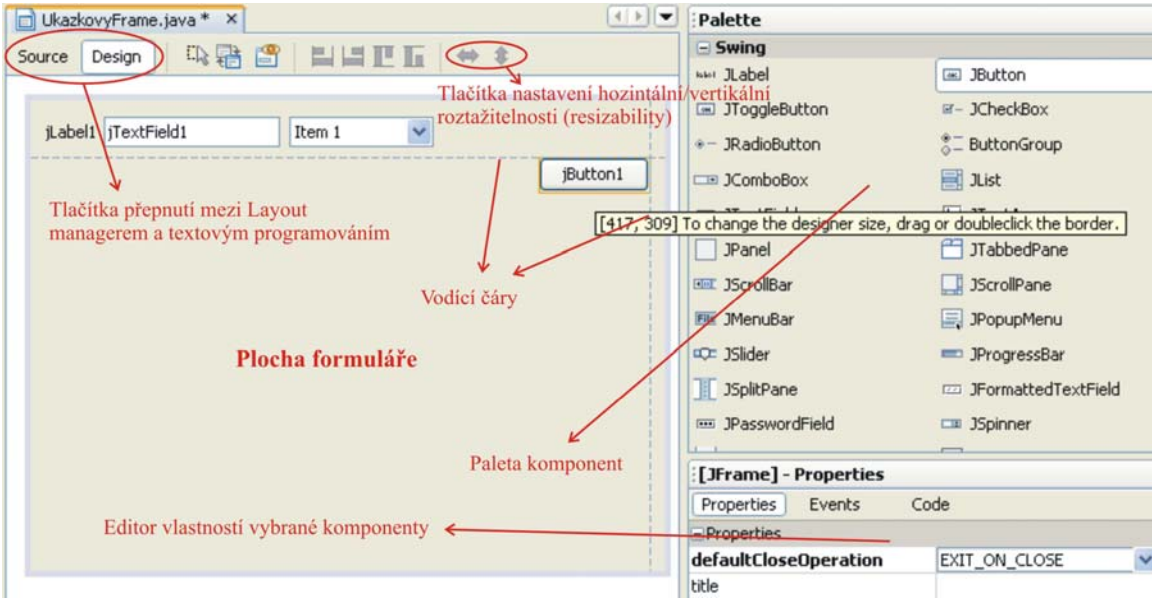
NetBeans IDE obsahuje od verze 5.0 nový layout manager zvaný *GroupLayout*, který se díky velkému tlaku vývojářské komunity objevil v JDK 6. Tento nový (a oproti dřívějším verzím značně vylepšený) manager umožňuje výrazně snazší a intuitivní návrh grafického rozhraní programu, tzv. GUI (graphical user interface). V NetBeans editoru je tento layout označován jako *FreeLayout*, ačkoliv sami vývojáři o něm hovoří jako o *GL*, což může být zdrojem nedorozumění. Věžme tedy, že *GroupLayout* a *FreeLayout* jsou jedno a to samé.

GroupLayout umožňuje (za pomoci tzv. „grupování“ komponent) inteligentní změnu velikosti formulářů, používá vodící čáry k pohodlnému a přesnému umístění komponent na formulář, jehož vzhled lze (včetně všeho co obsahuje) komplexně upravovat pomocí pravidel Look and Feel. V nejbližší době se chystá vydání nové stabilní verze NetBeans 6.0, ve které budou vlastnosti Layout editoru (dle slov vývojářů na www.netbeans.org) ještě intuitivnější a lepší.

Chování a výsledný vzhled navrženého formuláře lze snadno vyzkoušet tlačítkem *Preview Design* v panelu manageru. V níže navrženém ukázkovém programu se bude při akci resize (změna velikosti okna) pole *jTextField* automaticky přizpůsobovat šířce okna (Nastavena *horizontal resizability*), rozvržení ostatních komponent zůstane zachováno.

2.2.1. Tipy pro snazší používání Layout editoru:

- Vícenásobné vkládání komponent stejného typu realizujeme se stisknutým tlačítkem Shift.
- Komponenty lze zarovnávat, ukotvovat či zadat stejnou velikost – po označení skupiny komponent stačí vybrat funkci z pop-up menu (pravé tlačítko myši).
- Změnu textu komponenty, která má focus, provedeme stisknutím mezerníku.
- U všech komponent lze nastavit horizontální/vertikální změnu velikosti – *resizeability* – pomocí tlačítek viz. obrázek.



obr. 2: NetBeans Layout manager

2.3. Další užitečné vlastnosti

2.3.1. Encapsulate Fields

NetBeans 5.5 obsahují užitečnou funkci inteligentního zapouzdření, dostupnou v záložce *Refactor/ Encapsulate Fields*. Co znamená inteligentní zapouzdření? Pokud vytváříme třídu s veřejnými proměnnými libovolného typu, obvykle potřebujeme též vytvořit funkce pro získání a nastavení hodnoty těchto proměnných, například:

Třída *KatalogCD* bude obsahovat proměnné *CisloCD, JmenoCD* a *Interpret*. Abychom mohli v objektově orientované Javě k těmto proměnným přistupovat, definujeme procedury *setCisloCD, getCisloCD, setJmenoCD* atd. V tuto chvíli přichází na scénu tzv. *Generating Getter and Setter Methods* za pomoci uvedené funkce *Encapsulate Fields*. Během procesu generování metod máme navíc možnost v dialogovém boxu vybrat žádoucí, případně negenerovat nechtěné metody.

2.3.2. Fix Imports

V záložce *Source/Fix Imports* se nachází další vylepšení NetBeans, automatická kontrola „**`imports`**“ tedy importovaných knihoven JDK. Funkce zkontroluje kód a v případě více možností importovaných knihoven dá uživateli na výběr.

2.3.3. Code Templates

Uživatelsky velmi příjemná záležitost je používání tzv. templates, tedy šablon. V případě často se opakujících metod či částí kódu (i v různých programech) lze v menu *Tools/Options/Editor/Code Templates* nadefinovat vlastní šablonu, případně použít některou z již přítomných. Najdeme zde šablonu pro metodu **`public static void main()`**, ale například i pouzdro **`try-catch`** nebo předdefinovaný **`for`** cyklus.

2.3.4. Hints

Pokud má psaný kód nějaké chyby, případně obsahuje neimportované metody, objeví se po levé straně malá žárovka, která nabídne několik dostupných způsobů, jak nastalou situaci vyřešit (aby se objevila, musí se kurzor nacházet na daném řádku). Tento „pomocník“ může automaticky opravit import, vytvořit skeleton nové metody či deklarovat proměnnou.

2.3.5. Developer Colaboration

Developer colaboration je funkce NetBeans umožňující sdílení, vzdálenou editaci a on-line komunikaci mezi programátory na vzdálených stanicích. Pro snadné a přehledné sledování verzí, obzvlášť důležité při spolupráci více vývojářů, obsahuje NetBeans integrovaný CVS server.

2.3.6. Mobilní zařízení

Netbeans IDE obsahuje v současnosti pravděpodobně nejlepší balík funkcí (tzv. *Mobility pack*), určených pro mobilní zařízení (PDA, telefony aj.). Tento pack je dostupný jako extra balíček na stránkách projektu (nutno jej doinstalovat do standardní instalace).

3. Java 3D

3.1. Úvod

Java 3D API je dodatečnou knihovnou, jejíž vývoj byl přesunut na stránky Java komunity https://java3d.dev.java.net/. Nejnovější instalační balíček lze stáhnout na: java.sun.com/products/java-media/3D/downloads/index.html.

Tato „nadstavba“ standardní instalace JDK (případně JRE) slouží, jak je ostatně jednoznačně patrno z názvu, k zobrazování trojrozměrné grafiky. Java 3D integruje využití stávajících technologií DirectX a OpenGL a využívá tak hardwarových prostředků grafické karty. Rendering grafických dat proto neprobíhá pouze softwarově, což významně odlehčuje procesoru počítače a běh programů je rychlejší a plynulejší.

Java 3D také umožňuje import objektů vytvořených modelovacími nástroji jako TrueSpace a VRML modelů.

3.2. Základní kameny 3D vesmíru

K vytvoření virtuálního prostoru v Javě3D potřebujeme několik základních kroků:

- Vytvořit virtuální vesmír.
- Vytvořit datovou strukturu obsahující zobrazované objekty.
- Přidat objekt do skupiny dané struktury.
- Umístit pozorovatele na scénu.
- Přidat skupinu s objektem do vesmíru.

V první části budeme kvůli zjednodušení pracovat s elementárními geometrickými objekty, tzv. *primitivy*, jež později zaměníme za uživatelské objekty sestavené z konkrétních vstupních dat.

3.2.1. Primitiva

Primitivy myslíme jednoduchá geometrická tělesa jako např. koule, kvádr atd., sdružená v balíčku **`com.sun.j3d.utils.geometry`**. Manipulace s primitivy je jednoduchá

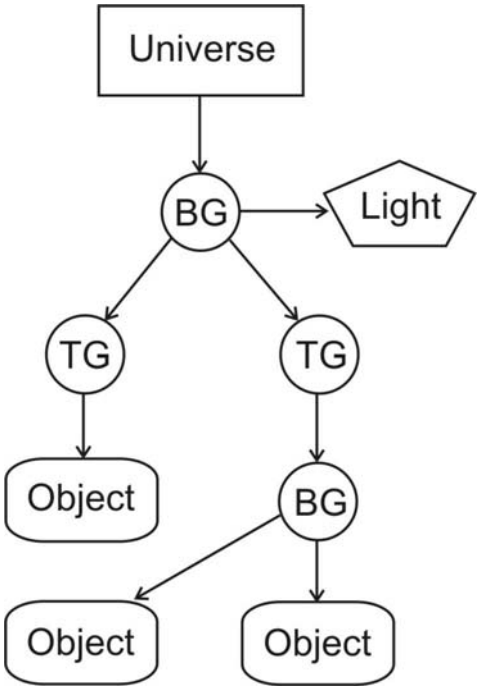
a přímočará. Pokud například u koule nechceme definovat „rozlišení“ a použít složitější konstruktor, stačí nám k jejímu vytvoření pouze zadat poloměr. Základní konstruktory primitiv vypadají následovně:

```
Box(float xdim,   float ydim,   float zdim,   Appearance ap)
     //kvádr
Cone(float radius, float height)
     //kužel
Cylinder(float radius, float height)
     //válec
Sphere(float radius)
     //koule
```

Všechna primitiva mají také konstruktory typu **`Box()`**, které vytvoří objekt s výchozími parametry.

3.2.2. Graf scény

Na následujícím obrázku ukážeme příklad strukturování scény:



Universe – virtuální vesmír

BG (*Branch Group*) – skupina objektů

TG (*Transform Group*) – podskupina objektů, na kterou aplikujeme určitou transformaci

Object – např. některé z grafických primitiv

Light – světlo, aplikované na danou větev/skupinu objektů

obr. 3: Graf scény

Základem každé scény v Javě3D je objekt *VirtualUniverse*, který však pro naši potřebu nahradíme mnohem jednodušším a přesto naprosto dostačujícím *SimpleUniverse*, což je podtřída *VirtualUniverse* zbavená objektů *PhysicalBody*, *ViewPlatform* aj.

Vesmír *SimpleUniverse* dále odkazuje (přes počátek souřadné soustavy scény) na objekty *BranchGroup*, které představují kořen jednotlivých částí scény. Pro potřebu jednodušších programů bohatě postačí jediná *BranchGroup*.

Objekt *TransformGroup*, poslední uzel grafu před samotným tělesem (*Object*), kombinuje různé možnosti transformace: *rotaci* podle jednotlivých os, *změnu měřítka* a *posun*.

Skupině transformací (*TransformGroup*) můžeme přiřadit libovolné množství objektů a pracovat s nimi jako s jedním složitým tělesem: například vytvoříme-li stůl z kvádru a čtyř válců, můžeme jej takto snadno transformovat za pomoci metod **`rotX(), rotY(), rotZ(), setScale(), setTranslation();`** ty aplikujeme na třídu *Transform3D*, kterou posléze přiřadíme skupině *TransformGroup*.

3.2.3. Appearance

Vzhled objektů na scéně závisí mimo jiné na vlastnostech jejich materiálu, tedy vlastnostech povrchu vzhledem k osvětlení. „Materiál“ obsahuje pět definovatelných vlastností: barvy ambientní, emisivní a difuzní, barvu odlesku a lesk samotný.

Konstruktor
```
Material(Color3f ambientColor, Color3f
emissiveColor, Color3f diffuseColor, Color3f
specularColor, float shininess);
```

- Ambientní barva se vytváří pomocí *AmbientLight* (o světlech později). První parametr určuje, jakou část ambientního světla materiál odrazí; toto se počítá jako součin intenzity jednotlivých složek světla a materiálu.
- Emisivní barva udává světlo vyzařované samotným objektem (použití například jako Slunce).

- Difuzní složka představuje odraz světla pocházejícího z určitého zdroje. Od povrchu tělesa se odráží do všech směrů stejně a určuje barvu osvětleného materiálu. Lze ji použít k vytvoření matných objektů.
- Specular color, čili barva odlesku, je dále určena posledním parametrem typu **`float`**, který určuje velikost odlesku tím, o kolik se odlesková složka odchýlí od zákona odrazu. Číslo v intervalu 1 až 128 definuje úhel rozptylu světla. Výchozí hodnota je 64.

Vlastnosti materiálu, vytvořeného za pomoci výše uvedeného konstruktoru, nejprve přiřadíme objektu *Appearance* metodou **`setMaterial(Material material)`**. Vzhled použijeme buď již v konstruktoru objektu, nebo jej přiřadíme metodou **`setAppearance(Appearance a)`**.

Třída *Appearance* se ovšem neomezuje pouze na materiál. Z velkého množství jejích definovatelných vlastností se omezíme pouze na ty, které se nám budou hodit v dalších částech tvorby 3D aplikace: barvu objektu při nulovém osvětlení určují *ColoringAttributes*. Vzhled polygonu a způsob jeho zobrazení definují *PolygonAttributes*. *TransparencyAttributes* obsahují parametry pro výpočet průhlednosti objektu. Poslední dvě zmíněné skupiny vlastností *LineAttributes* a *PointAttributes* vzhledem k jejich názvu netřeba dále představovat.

3.2.4. Světla

V Javě3D můžeme použít celkem čtyři typy světel: ***ambientní***, přicházející ze všech směrů, ***směrové***, jehož směr a intenzita jsou stejné ve všech bodech scény, ***bodové*** světlo, jehož intenzita se snižuje se vzdáleností od zdroje, a světlo ***reflektorové*** s podobnými vlastnostmi jako bodové, které se šíří pouze v zadaném kuželu (podobně jako u divadelního reflektoru).

Konstruktory jednotlivých typů světel vypadají následovně:

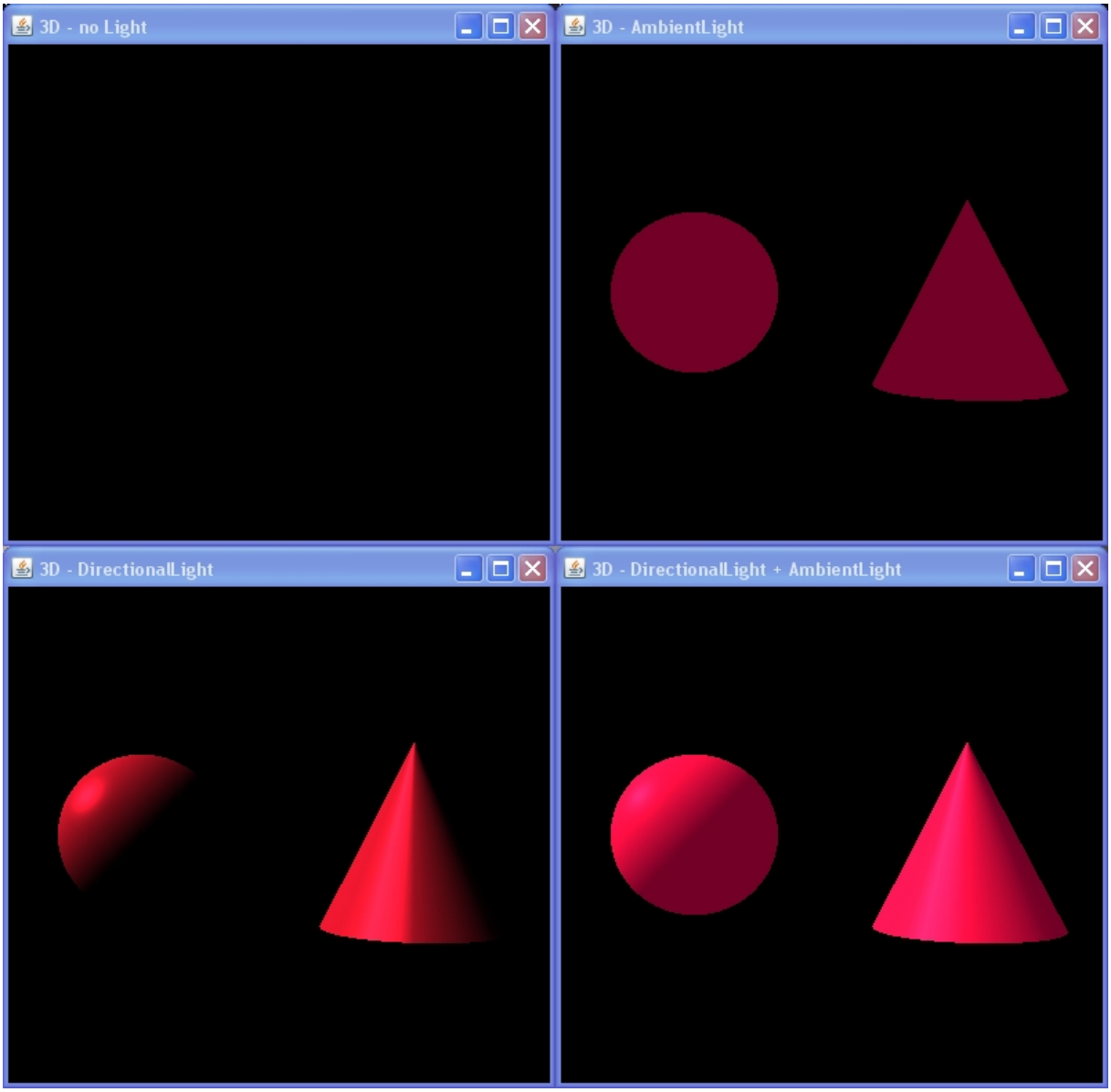
- **`AmbientLight(Color3f color)`**
- **`DirectionalLight(Color3f color, Vector3f direction)`**
- **`PointLight(Color3f color, Point3f position, Point3f attentuation)`**
- **`SpotLight(Color3f color, Point3f position, Point3f attentuation, Vector3f direction, float spreadAngle, float concentration)`**

Parametr **`direction`** určuje směr působení světla, **`position`** umístění zdroje světla, **`attenuation`** obsahuje konstanty pro výpočet intenzity světla, **`spreadAngle`** udává úhel v radiánech mezi směrem a polopřímkou vedenou pláštěm kuželu, konstanta **`concentration`** určuje, jak se mění intenzita světla se vzdáleností od osy kužele.

Světlu dále musíme zadat osvětlovanou oblast, tedy určit část scény, jejíž objekty budou osvětleny, pomocí metody **`setInfluencingBounds(Bounds region)`**, které jako parametr předáme například instanci třídy **`BoundingSphere`**.

Java 3D využívá k výpočtu odrazu světla *normálových vektorů* tělesa. Pokud osvětlujeme některé z geometrických primitiv, je třeba v konstruktoru zadat parametr GENERATE_NORMALS. V případě vytváření vlastních objektů musíme tyto vektory vytvořit například s využitím objektu *NormalGenerator*.

Příklad aplikace různých typů světel na scénu je znázorněná na následujícím obrázku. Zobrazovány jsou objekty s určeným materiálem a bez definované barvy, v neosvětlené scéně tedy nebudou vidět. Při aplikaci ambientního světla budou primitiva zobrazena, ovšem bez jakéhokoliv náznaku plastičnosti. Dojem třetího rozměru vytvoří až směrové *DirectionalLight*, nejlepší a nejrealističtější výsledek však dostaneme vhodnou kombinací obou typů světla.



obr. 4: Světla v Javě3D

3.2.5. Žijící a zkompilované objekty

Jednotlivé části scény jsou vkládány do vesmíru a poté vykreslovány na trojrozměrné plátno (*canvas3D*) prostřednictvím objektu **`BranchGroup`**. Přidáním tohoto objektu do vesmíru stává se **`BranchGroup`** a všechny obsažené komponenty tzv. žijícím objektem. To způsobí změnu vnitřní struktury grafu scény na výkonově výhodnější podobu, což přináší nevýhodu v tom, že již nebude možné graf dále měnit. Pokud tedy chceme měnit transformační matice **`Transform3D`**, případně přidávat objekty do scény, musíme toto povolit specifickým příznakem typu Integer v metodě **`setCapability`**.

Příznaky, jimiž můžeme nastavovat možnosti scény, jsou mimo jiných zejména **`ALLOW_CHILDREN_EXTEND`**, dovolující přidávat další objekty do dané skupiny, a **`ALLOW_TRANSFORM_WRITE`**, povolující měnit transformační matici za běhu programu a tím animovat scénu.

Objekty **`BranchGroup`** lze pro potřeby zvýšení rychlosti a efektivity vykonávání programu navíc zkompilovat metodou **`compile()`**. Tato metoda způsobí další změnu vnitřní struktury skupiny a všech jejích prvků. Po kompilaci již není možné do skupiny přidávat žádné další prvky. Pokud bude skupina „živá“, tedy již přiřazená objektu Universe, dojde při pokusu o zkompilování kódu k výjimce **`javax.media.j3d.RestrictedAccessException`**. Metodu **`compile()`** tedy voláme až v závěru, nejlépe těsně před tím, než přiřadíme skupinu do vesmíru.

4. Gmsh mesh file – použitý datový standard

Jedním z podmínek zadání je práce s datovými soubory Gmsh. Gmsh je trojrozměrný generátor konečně-prvkových souřadnicových sítí, který vyvinuli Christophe Geuzaine a Jean-François Remacle. Domovské stránky projektu jsou http://geuz.org/gmsh/ odkud je tento program volně distribuovatelný pod podmínkami „*GNU General Public License*“, což jest Obecná veřejná licence GNU.

V současné době existují tři verze formátu Gmsh souborů: verze 1.0, kterou budu ve svém programu používat, a verze 2.0 ASCII a 2.0 Binary, které se liší pouze reprezentací dat - jejich struktura je téměř totožná.

Gmsh je textový soubor o přesně definované struktuře s příponou *.msh. Soubor je rozdělen do dvou částí, z nichž první, ohraničená klíčovými slovy ($NOD-$ENDNOD), obsahuje libovolné množství bodů (resp. uzlů). Část druhá mezi ($ELM-$ENDELM) uzavírá různé druhy elementů, odvozené od souřadnic první části.

4.1. Struktura souboru

Jednou z výhod a současně i komplikací tohoto formátu je fakt, že body i elementy mohou následovat v absolutně libovolném pořadí (v rámci odpovídající sekce) a dokonce nemusí být v seznamu obsažena ani všechna čísla v daném rozsahu. Například uvažujme soubor obsahující pouhé tři body s indexy 2, 7 a 54.

Struktura souboru (schéma převzato z oficiální dokumentace geuz.org):

```
$NOD
number-of-nodes
node-number x-coord y-coord z-coord
...
$ENDNOD
$ELM
number-of-elements
elm-number elm-type reg-phys reg-elem number-of-nodes
     node-number-list
...
$ENDELM
```

Zde *number-of-nodes* je počet vrcholů v síti (tento se musí shodovat se skutečným počtem následujících bodů).
node-number je index, či chcete-li číslo n-tého uzlu sítě (v libovolném pořadí), následovaný souřadnicemi X, Y a Z ve formátu čísla s plovoucí řádovou čárkou.

number-of-elements je opět počet elementů následovaný výčtem prvků – jeden prvek na jednom řádku. Každý element je uveden indexem `elm-number`, následuje typ elementu `elm-type` (o tom později), `reg-phys` neboli číslo fyzické entity, které element náleží, a `reg-elem`, označující indexem elementární entitu. Počet bodů určujících n-tý element je určen parametrem `number-of-nodes`, následovaným výčtem jednotlivých bodů.

Jedná se tady o jakýsi typ indexované geometrie, kde souřadnice každého vrcholu jsou uloženy pouze jednou pod unikátním indexem, kterým je na ně později libovolně odkazováno.

V Gmsh standardu verze 1.0 rozlišujeme celkem 19 typů elementů označených čísly 1 až 19. V tomto programu se budeme zabývat pouze prvními třemi typy elementů, a to úsečkami (lines, typ = 1), trojúhelníky (triangles, typ = 2) a čtyřúhelníky (quadrangles, typ = 3). Standard dále rozeznává mnoho prostorových útvarů jako čtyřstěn, kvádr, jehlan, pyramida a jejich modifikace.

Příklad struktury Gmsh souboru:

```
$NOD
4
2 1 0 0
8 0.9749279121818258 0.2225209339563045 0
4 -0.5563510131363865 0.5891913365526474 0.5859412249701538
5 0.4105268835831876 0.552121102724671 0.7256927304941428
$ENDNOD
$ELM
3
1 1 1 1 2 2 8
9 2 1 1 3 2 4 5
3 3 7 7 4 2 4 5 8
$ENDELM
```

Tento soubor definuje čtyři body s indexy 2, 8, 4 a 5, na které je v druhé části odkazováno jako na koncové body přímky (body 2 a 8) a jako na vrcholy trojúhelníku a čtyřúhelníku. V reálných situacích se obvykle nesetkáme se stavem, kdy by indexy prvků byly voleny takto náhodně, příklad nicméně ilustruje skutečnou vlastnost Gmsh: variabilitu a přizpůsobivost.

Během práce s Gmsh bude zřejmě třeba vytvořit několik pomocných datových struktur, do kterých budeme načítat jednotlivé elementy, například pole bodů **`Point3f[]`** obsahující souřadnice bodů či pole celých čísel **`int[]`**, kam se budou ukládat indexy bodů pro jednotlivé elementy.

Dále bude nutné ošetřit převod textového formátu na souřadnice typu **`float`** či indexy typu **`integer`**, na případné chybové stavy a poškozená vstupní data nezapomínaje.

5. Vývoj programu v prostředí NetBeans

V NetBeans IDE byl vytvořen projekt neobsahující třídu Main, do něhož byl dále vložen tzv. JForm, tedy třída, pracující s vizuálním editorem NetBeans. To znamená, že vývojář navrhne rozhraní grafickým způsobem a IDE automaticky vygeneruje potřebný kód všech komponent (například vytvoří instance tlačítek, které umístí na formulář na požadované pozice).

5.1. Návrh grafického uživatelského rozhraní

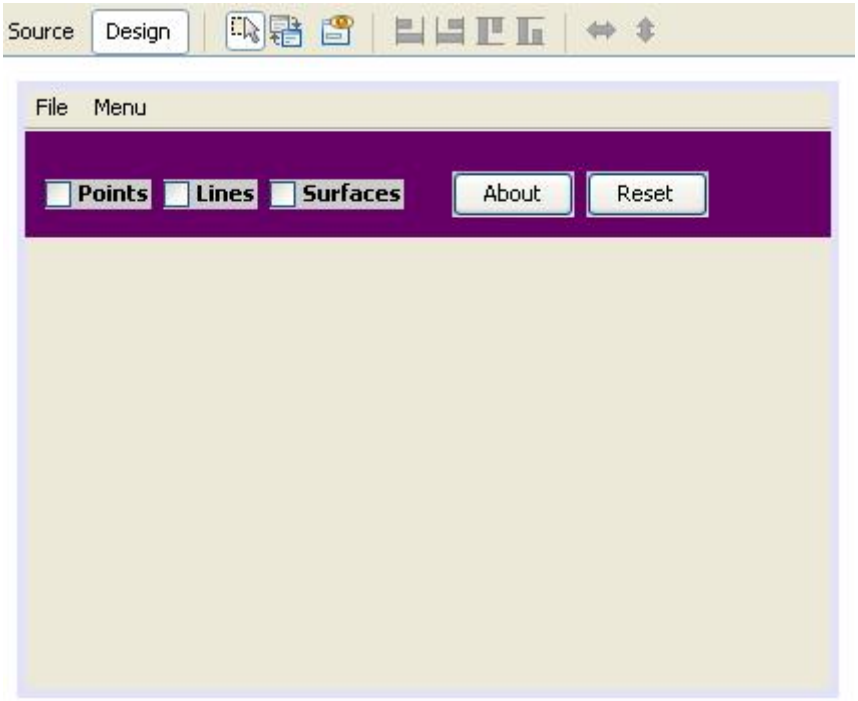
Při navrhování designu formuláře využijeme služeb Layout editoru, popsaného výše. Po mnohých pokusech a různých navržených verzích formuláře byla jako nejlepší zvolena následující kombinace dvou různých *layout*ů:

Pro plochu formuláře byl nastaven *BorderLayout*, který se svými vlastnostmi přesně hodí pro zvolené rozvržení komponent a v kódu k němu lze velmi snadno, efektivně a přímočaře přistupovat.

Do horní části byl umístěn standardní *MenuBar*, jehož položky budou upravovány dále dle potřeby, a pod něj hlavní ovládací panel s tlačítky a jinými prvky. Panelu byl dále nastaven nový *GroupLayout*, díky kterému bylo dosaženo přehledného a vizuálně příjemného rozvržení komponent. Barva panelu byla zvolena vzhledem k pozadí *Canvasu3D* (tj. černá), tedy tmavší a současně od černé snadno odlišitelná.

Zmiňovaný *Canvas3D* bude v kódu této třídy umístěn na zbylou plochu formuláře, kterou beze zbytku vyplní.

Tato třída byla jakožto hlavní součást aplikace pojmenována *Main* a umístěna do balíčku *demo*. Její zdrojový kód se tedy bude nacházet v adresáři src/Demo v souboru Main.java.



obr. 5: Návrh grafického rozhraní v NetBeans

5.2. Zobrazení 3D objektů na canvas

Abychom mohli ve formuláři kreslit trojrozměrné objekty, musíme vytvořit a umístit na plochu *Canvas3D* a jemu přiřadit graf scény, což uděláme následovně.

5.2.1. Canvas3D:

Nejprve získáme odkaz na tzv. kontejner, neboli třídu, do které budeme umísťovat komponenty (tuto funkci je nutné použít u všech **`javax.swing`** kontejnerů jako *JDialog, JFrame, JWindow, JApplet, JInternalFrame*):

```
Container pane = getContentPane();
```

Poté vytvoříme nový *Canvas3D* za použití výchozího nastavení pro *SimpleUniverse*, který přidáme do kontejneru (tím je vytvořený *JFrame*):

```
GraphicsConfiguration conf =
    SimpleUniverse.getPreferredConfiguration();
Canvas3D canvas = new Canvas3D(conf);
pane.add(canvas);
```

5.2.2. Graf scény:

Ve chvíli, kdy máme hotový *canvas* (po spuštění by měla mít *canvasem* pokrytá plocha formuláře černou barvu), mu potřebujeme přiřadit různé objekty grafu scény, jak je popsáno v kapitole *3.2.2. Graf scény*. Vytvoříme tedy *SimpleUniverse*, kterému budeme postupně přiřazovat *BranchGroup* a dvě *TransformGroup* (jednu pro posun - translaci a druhou pro rotaci objektu). Abychom však mohli měnit parametry žijících skupin, musíme jim povolit příslušné schopnosti:

```
SimpleUniverse universe = new SimpleUniverse(canvas);
BranchGroup group = new BranchGroup();
TransformGroup tgT = new TransformGroup();
tgT.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
TransformGroup tgR = new TransformGroup();
tgR.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
tgR.setCapability(TransformGroup.ALLOW_CHILDREN_WRITE);
tgR.setCapability(TransformGroup.ALLOW_CHILDREN_EXTEND);

group.addChild(tgT);
tgT.addChild(tgR);
```

Abychom mohli přidat potomky (v našem případě zobrazovaný objekt) do skupiny i za běhu programu, je nutné nastavit `ALLOW_CHILDREN_EXTEND` poslední TG – tedy tgR. Této skupině pro začátek přiřadíme některé z geometrických primitiv, abychom si mohli vyzkoušet možnosti animace (tyto nejsou v tuto chvíli implementovány).

Když tuto scénu (s například kostkou či koulí) spustíme - nejprve je ovšem nutné přiřadit *group* vesmíru - dočkáme se pouze černé obrazovky (pokud nebyla objektu přiřazena *Appearance* s nastavenou barvou). Scénu bude nyní třeba osvětlit: do skupiny *group* přidáme světlo.

5.2.3. Světlo

Použité osvětlení se bude skládat ze dvou složek, a to z všesměrového *AmbientLight* a vektorem orientovaného *DirectionalLight*. Těmito dvěma prvky budeme nastavovat také barvu objektů. Zvolíme například modrou barvu, která se bude pro jednotlivé složky nepatrně odlišovat:

```
BoundingSphere bounds =
     new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
Color3f light1Color = new Color3f(0.15f, 0.1f, 0.8f);
Vector3f light1Direction = new Vector3f(20f,-15f,-10f);
DirectionalLight light1 =
     new DirectionalLight(light1Color, light1Direction);
light1.setInfluencingBounds(bounds);
group.addChild(light1);
AmbientLight light2 =
     new AmbientLight (new Color3f(0.1f,0.1f,0.65f));
light2.setInfluencingBounds(bounds);
group.addChild(light2);
```

Všimněte si, že světlo přidáme do scény dřív, než *group* zkompilujeme, nemusíme jí tedy povolovat žádné schopnosti (*capability*), neboť za běhu se s ní (s *group*) nebude manipulovat.

Aby se světlo „odráželo“ od objektu, musí java znát normálové vektory zobrazovaného povrchu (viz. *3.2.4. Světla*):

```
Appearance appearance = new Appearance();
Material m = new Material();
m.setAmbientColor(new Color3f(0.6f, 0.6f, 0.6f));
m.setShininess(30);
appearance.setMaterial(m);
Box kvadr = new
     Box(0.2f,0.2f,0.2f,Box.GENERATE_NORMALS, appearance);
```

```
tgR.addChild(kvadr);

group.compile();
universe.getViewingPlatform().setNominalViewingTransform();
universe.addBranchGraph(group);
```

V tuto chvíli se nám po spuštění zobrazí přední stěna krychle. Můžeme do kódu přidat nejrůznější nastavení transformací za pomoci třídy Transform3D:

```
Transform3D trR = new Transform3D();
trR.rotX(Math.toRadians(30));
tgR.setTransform(trR);
```

Nicméně s každou změnou pohledu musíme dělat změny v kódu. Proto musíme implementovat tzv. „posluchače událostí“ pro každou akci, na kterou budeme chtít reagovat.

5.3. Implementace uživatelského rozhraní

Reakce na uživatelovu činnost se v Javě realizuje pomocí tzv. *subinterfaces EventListener*u. Těch existuje několik desítek, v tomto programu však stačí využít služeb několika základních, a to:

- KeyListener – ošetřuje stisk klávesy
- MouseMotionListener – procedury prováděné při pohybu myší
- MouseListener – volá se při stisknutí/uvolnění/kliknutí tlačítka myši a při vstupu/opuštění prostoru nad komponentou
- MouseWheelListener – zabývá se výhradně otáčením roll kolečka myši

Tato rozhraní se implementují následujícím způsobem:

```
public class Main extends javax.swing.JFrame implements
     KeyListener, MouseMotionListener,MouseListener,
     MouseWheelListener
```

Ve chvíli, kdy toto napíšeme do deklarační části třídy, objeví se po levé straně žárovka „hint“, která nám nabídne automatický import použitých tříd a vytvoří skeletony všech abstraktních metod, vyžadovaných pro správný chod „*listenerů*“.

Události, kterými bude uživatel animovat zobrazený předmět, naprogramujeme tímto způsobem:
Ve třídě *Main* si nadefinujeme „soukromé“ proměnné typu **`float`** udávající pozici a otočení v prostoru, dvě proměnné typu *Transform3D* a dvě typu *TransformGroup*:

```
private TransformGroup tgT, tgR;     //translace, rotace
private Transform3D trT,trR;
private float posX = 0;
private float rotX = 0;       //atd. pro Y+Z
```

K těmto proměnným (číselné hodnoty je třeba inicializovat, abychom se vyhnuli **`nullPointer Exception`**, tj. výjimka, kdy odkazujeme a snažíme se pracovat s hodnotou typu **`null`** - podstatě se jedná o odkaz „nikam“) budeme přistupovat v metodách ošetřujících žádoucí události – stisk klávesy, pohyb myší atd.

5.3.1. Událost klávesnice

Stisk tlačítka lze ošetřit pomocí metod **`keyPressed(KeyEvent e)`**, **`keyTyped(KeyEvent e)`** a **`keyReleased(KeyEvent e)`**. Z těchto tří použijeme *KeyPressed*, neboť se chová jako například pohyb kurzoru v MS Word: po stisku se posune o jedno pole, kde chvíli čeká, a zhruba po půl sekundě se plynule přesunuje ve směru pohybu.

Informaci, jaká klávesa byla stisknutá, obsahuje parametr KeyEvent jako příznak typu **`Integer`**. Metodu tedy rozdělíme podle stisknuté klávesy:

- posun po ose Y – šipky nahoru (KeyEvent.VK_UP) a dolu (VK_DOWN)
- posun po ose X – šipky doleva (VK_LEFT) a doprava (VK_RIGHT)
- posun po ose Z – klávesy plus (+) a mínus (-). Tyto budou pro ilustraci různého přístupu ošetřeny trochu jiným způsobem.

```
if (e.getKeyCode()== KeyEvent.VK_UP) {
```

```
        posY+=.05;
        trT.setTranslation(new Vector3f(posX,posY,posZ));
        tgT.setTransform(trT);
}
if (e.getKeyChar()=='-') {
        posZ-=.05;
        trT.setTranslation(new Vector3f(posX,posY,posZ));
        tgT.setTransform(trT);
}
```

Takto budou vypadat metody pro jednotlivá tlačítka – celkem jich musí být ošetřeno šest. Operace nastavení translace by mohla být umístěna za všemi bloky **`if`**, aby se stejný kód nemusel opakovat pro každou klávesu. Znamenalo by to však, že se tyto operace budou provádět při stisku jakéhokoliv tlačítka (ačkoliv pozice se nezmění), což by znamenalo zbytečnou zátěž systému navíc. Bude proto vhodnější přepočítávat pozici pouze při změně parametrů polohy.

Pokud takto naprogramujeme obsluhu klávesnice, bude kód přeložen a spuštěn, ale kostka se ani nepohne. To napravíme tak, že v třídě *public Main()* přidáme *canvasu* „posluchače“ událostí:

```
canvas.addKeyListener(this);
```

Obdobně ošetříme i všechny následující *EventListener*y.

5.3.2. Stisk tlačítka myši

V takovémto typu programu je vhodné animovat objekt pohybem myši. To je realizováno jednak pomocí metody interface *MouseListener*: **`mousePressed()`**, jednak *MouseMotionListener*: **`mouseDragged()`**. Nejprve si nadefinujeme soukromé proměnné, které budou uchovávat informace o souřadnicích a stisknutém tlačítku:

```
        private int mouseX, mouseX, mouseButton.
```

Z dostupných metod **`mouseClicked(MouseEvent e), mouseReleased (MouseEvent e)`** a **`mousePressed(MouseEvent e)`** použijeme poslední jmenovanou. Parametr stisknutého tlačítka (tj. hodnoty 1, 2 nebo 3) získáme metodou

```
mouseButton = e.getButton();
```

a aktuální hodnoty pozice kurzoru:

```
mouseY = e.getY();
mouseX = e.getX();
```

Samotnou animaci objektů budeme realizovat podobným způsobem jako při události klávesnice. Použitá metoda **`mouseDragged(MouseEvent e)`** pochází z interface *MouseMotionListener*.

5.3.3. Pohyb myši

Předmětem chceme hýbat pouze se stisknutým tlačítkem myši, což přímo implementuje **`mouseDragged(MouseEvent e)`**. Na nás tedy zůstává rozlišit, o jaké tlačítko se jedná, a podle toho naprogramovat transformaci.

```
public void mouseDragged(MouseEvent e) {
  int xx = mouseX - e.getX();
  int yy = mouseY - e.getY();
  if (mouseButton == 3){ //translace
     int sirkaPanelu = jPanel1.getWidth();
     posX -= ((float)2*xx/sirkaPanelu);
     posY += ((float)2*yy/sirkaPanelu);
     trT.setTranslation(new Vector3f(posX,posY,posZ));
     tgT.setTransform(trT);
  }
  if (mouseButton == 2){ //rotace kolem Z}
  if (mouseButton == 1){ //rotace x+y}
mouseX = e.getX();
mouseY = e.getY();
}
```

Při stisknutí tlačítka se uloží kód tlačítka a aktuální pozice. Pokud je s myší nadále hýbáno (drag), dojde k vyhodnocení stisknutého tlačítka a změně parametrů jedné z *TransformGroups*. Pravým tlačítkem budeme těleso posouvat (o kolik závisí na aktuální šířce okna – pokud by byl parametr konstantní, těleso by se ve full screen módu pohybovalo o menší vzdálenost, takto kompenzujeme přizpůsobení canvasu velikosti okna).

Kód dalších dvou tlačítek je obdobný, pouze měníme velikost rotačních proměnných, které vzápětí přiřazujeme *tgR*. Jediný rozdíl je v nutnosti použít součin pro rotaci kolem os:

```
trR.rotX(rotX);
trLocal.rotY(rotY);
trR.mul(trLocal);
trLocal.rotZ(rotZ);
trR.mul(trLocal);
```

Pokud bychom nenásobili všechny tři proměnné, docházelo by k chybám zobrazování; obraz by přeskakoval a transformace by se mohla chovat nepředvídatelně – jakákoliv dosavadní rotace kolem vynechaných os by byla ignorována, dokud by nedošlo k její změně, kterou by následovala prudká skokové změna polohy.

5.3.4. Roll kolečko

Realizace tohoto ovládacího prvku je shodná s klávesami +/-.
K implementaci pouze zmíníme, že parametr otočení lze získat jako celé číslo metodou **`getWheelRotation()`**.

5.4. Menu bar

Nesmíme opomenout tento základní ovládací prvek vyskytující se v téměř každém programu. V tomto případě zvolíme dvě funkční skupiny: **File** (obsahující tlačítka **Open** a **Exit**) a **Menu** (s tlačítky **About** a **Reset**).

5.4.1. Exit

Tyto „podtlačítka“, tedy jednotlivé položky menu, vložíme pravým kliknutím na **File** v Layout editoru a výběrem možnosti *Add/JMenuItem*. „Dvojklikem“ na komponentu v Inspektoru (prohlížeč stromové struktury formuláře v levé části NetBeans) se automaticky vygeneruje metoda actionPerformed, jejíž tělo doplníme jediným řádkem kódu:

```
System.exit(WindowConstants.EXIT_ON_CLOSE);
```

5.4.2. Open

Tuto volbu budeme používat k načítání datových Gmsh souborů, k čemuž použijeme komponentu JFileChooser. Tato třída představuje standardní Open dialog s možností nastavení souborové masky.

```
JFileChooser chooser = new JFileChooser();
FileNameExtensionFilter filter = new
     FileNameExtensionFilter("Gmsh mesh file", "msh");
chooser.setFileFilter(filter);
int returnVal = chooser.showOpenDialog(null);
if(returnVal == JFileChooser.APPROVE_OPTION) {
     System.out.println("Vybran datovy soubor: " +
     chooser.getSelectedFile().getPath());
}
```

V takto nastaveném dialogu se vyfiltrují všechny soubory s příponou *.msh a tyto budou označeny jako *Gmsh mesh file*. Pokud dojde k potvrzení výběru, prozatím pouze vypíšeme název souboru do standardního systémového výstupu.

5.4.3. Reset

Funkce **Reset** vrátí všechny transformace do výchozího stavu (tzn. všechny transformační proměnné nastaví do nuly). Mým záměrem je mít tuto funkci obsaženou jak v tomto menu, tak dostupnou ve formě tlačítka na ovládacím panelu. Přesto není nutné ji programovat dvakrát – stačí již hotovou metodu přiřadit druhé komponentě jako tzv. *actionPerformed* událost. Obslužný kód je velmi jednoduchý a není třeba ho uvádět.

5.4.4. About

V první části procesu navrhování GUI (Graphic User Interface – Grafické Uživatelské Rozhraní) byla tato funkce implementována jako tlačítko na ovládacím panelu. Toto řešení však zabíralo zbytečně mnoho místa a jeho užitečnost byla poměrně rozporuplná, ve finální verzi se proto vyskytuje pouze v rámci panelu *Menu bar*.

Vzhledem k požadovaným funkcím About boxu (objeví se v novém okně, se kterým půjde manipulovat) jej naprogramujeme jako nový *JFrame* (tedy panel z **`javax.swing.JFrame`**). V balíčku **`demo`** vytvoříme novou veřejnou třídu s názvem *AboutBox*. V této třídě naprogramujeme nový panel, jehož tři základní vlastnosti (název, velikost a text) budou parametry konstruktoru:

```
public AboutFrame(String jmeno,String obsah,
     Dimension velikost) {}
```

Nejprve vytvoříme panel: **`final JFrame aboutFrame = new JFrame();`** jehož vlastnosti nastavíme pomocí metod **`setLocation()`**, **`setSize()`**, **`setResizable()`**, **`setDefaultCloseOperation()`**, **`setTitle()`** a **`setLayout()`**, samozřejmě vždy s patřičnými parametry. Poté vytvoříme nový JTextPane, ve kterém se bude zobrazovat parametr *obsah*. Dále stačí už jen přidat textový panel do *aboutFrame* a přiřadit mu text a About box by mohl být hotov. Pro snazší skrývání okna však ještě vytvoříme na spodní části panel (obdobný postup jako při vytváření textového pole) obsahující tlačítko *Close*:

```
Button but = new Button("Close");
but.addActionListener(new java.awt.event.ActionListener(){
 public void actionPerformed(java.awt.event.ActionEvent e){
   aboutFrame.dispose(); }
 }
);
```

Panelu nastavíme libovolnou barvu (například tmavě šedou) a velikost (šířka bude shodná se šířkou formuláře), přidáme na něj tlačítko a zobrazíme okno: **`aboutFrame.show();`**

Do třídy *AboutFrame.java* můžeme ještě přidat konstruktor s výchozími parametry:

```
public AboutFrame(){
     new AboutFrame("","",new Dimension(300,200));
}
```

To pro případné budoucí použití této třídy.

6. Manipulace s Gmsh

Načítání a zobrazování obsahu **.msh* bude nejobtížnější programátorskou částí této aplikace. Struktura tohoto typu souborů byla popsána v kapitole 4.1.

Otevření souboru realizujeme pomocí dvou tříd: *PointReader.java* a *Teleso.java*. První z nich bude realizovat načtení dat z textové podoby do proměnných jazyka Javy, které třída *Teleso* použije k vytvoření prostorových struktur a jejich umístění do virtuálního vesmíru.

6.1. PointReader

6.1.1. Proměnné

Tato třída bude obsahovat několik zásadních veřejných proměnných:

```
public String fileName = "unnamed.msh";
public Point3f poleBodu[];
public Point3f poleSouradnic[];
public String poleElementu[];
public int pocetBodu = -1;
public int nejmensiIndexBodu = 2147483647;
public int pocetElementu = -1;
public Float meritko = 1f;
```

Proměnná *fileName* obsahuje jméno otvíraného souboru s kompletní cestou – výchozí hodnota je „nepojmenovaný“ – tuto proměnnou naplníme při potvrzení volby v *openDialog*u **`chooser.getSelectedFile().getPath()`**. Další dvě pole typu *Points3f*, tedy pole bodů o třech souřadnicích typu **`float`**, budou obsahovat stejná data, pouze jinak seřazená. Do *poleBodu* načteme souřadnice vrcholů v pořadí, v jakém jdou po sobě v **.msh* souboru; z tohoto pole budeme později zobrazovat body jako součást tělesa.

poleSouradnic bude obsahovat tatáž data, pouze seřazeny podle indexů. Příčinou tohoto uspořádání je flexibilita Gmsh, kdy jednotlivé body nemusí být seřazeny vzestupně dle

indexů či některé indexy mohou být zcela vynechány (viz. kapitola 4.1.). Jedním z efektů může být nutnost vytvořit *poleSouradnic* delší, než je počet bodů.

Proces načítání tedy proběhne jednou pro *poleBodu*; během tohoto cyklu se naplní lokální proměnná **`int nejvetsiIndex`**, podle které později inicializuji *poleSouradnic*.

Druhým efektem standardu Gmsh je fakt, že některé položky *poleSouradnic* nemusí být naplněny a stanou se odkazem „nikam“. Abychom se vyhnuli **`nullPointer Exception`** při pozdějším použití tohoto pole, musíme nejprve inicializovat všechny prvky nějakou výchozí hodnotou, např. **`Point3f(0,0,0)`**. Tato hodnota může být zcela libovolná, neboť pokud nedojde k jejímu přepsání během načítání, znamená to, že se bod s tímto indexem v souboru vůbec nevyskytuje a tedy by na něho (pokud není chyba v *.msh) nemělo být odkazováno.

6.1.2. Načtení bodů

Tento proces realizujeme pomocí dvou tříd určených pro práci s řetězci:

```
FileReader pointFile = new FileReader(fileName);
BufferedReader in = new BufferedReader(pointFile);
```

BufferReader disponuje dvěma zásadními metodami:

`in.readLine()`, která vrátí hodnotu typu **`String`** odpovídající jednomu řádku načteného souboru a posune se na další řádek.

`in.mark(int readAheadLimit)`, která označí aktuální pozici (řádek) v souboru a po zavolání metody **`in.reset()`** se na ní vrátí. Parametr *readAheadLimit* udává maximální počet znaků, po který se bude značka uchovávat.

Postupovat musíme v souladu se strukturou Gmsh: první řádek musí obsahovat text "$NOD", druhý udává počet bodů. Číselné hodnoty extrahujeme z textové podoby pomocí funkcí **`Integer.parseInt(String s), Float.parseFloat (String s)`**. Poslední potřebná funkce náleží třídě **`String`** a jejím účelem je rozdělit řetězec do pole „substringů“ podle zadaného znaku: **`String.split(" ")`**.

Během načítání naplníme několik pomocných proměnných: *meritko* bude obsahovat hodnotu největší souřadnice, ze které pomocí převrácené hodnoty získáme výchozí měřítko pro zobrazení celého tělesa (obzvlášť užitečné u velkých a malých objektů). Dále získáme celkový počet bodů v proměnné *pocetBodů* a nejnižší index bodu v odpovídající proměnné – indexy Gmsh nemusí začínat od jedničky.

Poslední součást načítání bodů musí být ošetření chybových stavů, například: první řádek souboru nebude odpovídat standardu.

Celý kód bude proto uzavřen do dvou bloků **`try`**, které při výjimce vypíší chybové hlášení „Soubor nenalezen“, pokud dojde k chybě při otvírání *FileReader*em, nebo „Struktura souboru neodpovídá standardu Gmsh“, pokud dojde k jiné chybě během načítáni bodů či elementů.

Samotná část čtení bodů je uzavřena do bloku **`if`** a bude se provádět pouze, pokud první řádek obsahuje text "$NOD". V opačném případě následuje zpráva „Špatná struktura souboru - nelze načíst body“.

6.1.3. Elementy

Druhá část třídy *PointReader* se zabývá druhou částí Gmsh: elementy. Ty jsou načítány jako celé řádky do pole řetězců a není s nimi nijak manipulováno. Kód, stejně jako v předchozí části, uzavřeme do bloku **`if`**, ve kterém naplníme proměnné *poleElementu* a *pocetElementu*. Chybná struktura souboru bude opět příčinou chybového hlášení.

6.2. Třída Teleso

Kód této třídy je velice obsáhlý (při zachování formátování tohoto textu se jedná o přibližně dvanáct stran A4), jeho obsah proto rozebereme pouze ve zkratce. Jedná se o část programu, kde budou reálně vznikat prostorové modely na základě vstupních dat z *PointReader*u. Nejvýznamnější veřejné proměnné jsou instance abstraktní třídy **`Node()`**, což je jakýsi obecný uzel grafu scény:

```
public Node Points = null,Lines = null,
            Triangles = null, Quadrangles = null;
```

Tyto objekty představují čtyři prvky Gmsh souboru, které budeme zobrazovat. Z toho trojúhelníky a čtyřúhelníky budou zahrnuty pod jedním souhrnným označením **`Surfaces`** (povrchy).

Třída *Teleso* bude obsahovat čtyři základní metody, pomocí nichž budou vytvářeny jednotlivé prvky.

6.2.1. Points – Body

V tomto pojetí zadání se budou zobrazovat body, čáry a plochy nezávisle na sobě s tím, že výchozí nastavení po načtení souboru bude vykreslovat pouze body. Objekt *Points*, který není po inicializaci ničím jiným než prázdným ukazatelem, vytvoří metoda **`createSit(float scale)`**. Jak je patrno z názvu metody, v této části programu se vytvoří síť bodů a naplní se proměnné, které později použijeme ke generování ostatních těles.

Zkrácená podoba metody vypadá následovně:

```
public Shape3D createSit(float scale) {
     ptRead.pointRead(scale);
     GeometryInfo gi = new GeometryInfo(GeometryInfo.QUAD_
          ARRAY);
     gi.setCoordinates(ptRead.poleBodu);
     Appearance pointsApp = new Appearance();
     PolygonAttributes pa = new PolygonAttributes();
     pa.setPolygonMode(PolygonAttributes.POLYGON_POINT);
     pa.setCullFace(PolygonAttributes.CULL_NONE);
     pointsApp.setPolygonAttributes(pa);
     Shape3D points = new Shape3D(gi.getGeometryArray(),
          pointsApp ) ;
return points;
}
```

Objekt *Shape3D* je potomkem abstraktní třídy *Node*. V jeho konstruktoru zadáváme geometrii, vytvořenou pomocí *GeometryInfo*, a *Appearance,* určující vzhled a vlastnosti. Takto vytvořené body budou mít velikost jednoho pixelu a bílou barvu.

Jejich vlastnosti můžeme měnit v atributu *Appearance* – například lze vytvořit *PointAttributes,* obsahující jinou velikost bodu, *ColoringAttributes,* definující barvu, *Material*, který ošetří vzhled osvětleného objektu, či *TransparencyAttributes*, jež definují průhlednost.

Pokud nadefinujeme materiál, stane se to, že vizuální vlastnosti bodů se budou měnit s pozorovacím úhlem, což v tomto případě není žádoucí. Z tohoto důvodu není v programu materiál jako jediný z výše jmenovaných atributů použit. Důležitá vlastnost je *Transparency*, kterou budeme využívat k zobrazení, i naopak skrytí jednotlivých částí scény – žádoucí objekty budou jednoduše nastaveny jako viditelné.

Nejdůležitější třídou této metody je *GeometryInfo*, díky které snadno a rychle vytvoříme základní geometrii objektu. Tato třída je zvláštní tím, že nemusíme předem definovat počet prvků v ní obsažených. Jedná se vlastně o jakýsi „nafukovací“ kontejner uchovávající data pro zpracování dalšími nástroji. Těmi jsou *NormalGenerator*, který do geometrie bez normálových vektorů tyto automaticky přidá, a *Striptifier*, který objekt v *GeometryInfo* modifikuje do formy pásů (tuto funkci nebudeme potřebovat, *NormalGenerator* se však ukáže být velmi užitečný u ploch).

6.2.2. Lines – čáry

Tento prvek již pochází ze skupiny elementů (viz. Gmsh 4.1. Struktura souboru), proto tato část kódu bude odlišná od **`Points()`**. Objekt *Lines* bude opět *Shape3D*, u kterého definujeme *Appearance* a *Geometry*. Nejprve je však nutné zpracovat celé *poleElementů* (tj. pole řetězců vytvořené *PointReader*em) a extrahovat z něho souřadnice čar. Protože Gmsh není nutně posloupně uspořádaný, je třeba testovat všechny prvky *poleElementu*.

Protože předem nevíme, kolik jednotlivých typů elementů se v souboru vyskytuje, zvolíme jako mezistupeň mezi polem řetězců a samotnými souřadnicemi *ArrayList*, který se automaticky přizpůsobuje vkládaným datům. Realokace místa potřebná při každém zvětšování ArrayListu však zabírá nezanedbatelně mnoho systémových prostředků (a tedy zpomaluje průběh načítání). Proto je vhodné pomocí metody ensureCapacity(int i) jednou za čas rozšířit ArrayList o větší počet míst (zvolil jsem krok 100 prvků). Tato část kódu zde nebude pro ušetření místa uvedena.

```
ArrayList indexySouradnic = new ArrayList(100);
String s[];
for (int i=0;i < poleElementu.length; i++) {
  s = poleElementu[i].split(" ");
  if (Integer.parseInt(s[1])==1){  //1 = Lines
     indexySouradnic.add(Integer.parseInt(s[5]));
     indexySouradnic.add(Integer.parseInt(s[6]));
     linesCount += 1;  }
}
```

Parametrem *linesCount* lze snadno ošetřit případ, kdy datový soubor neobsahuje žádné přímky – pokud bude nulový, není třeba provádět další operace a vrátí se **`null`**.

Pokud soubor obsahuje nějaké linky, vytvoříme objekt **`IndexedLineArray(int vertexCount, int vertexFormat, int indexCount)`**, který je přímo určený k použití v indexované geometrii. Parametry zde znamenají postupně: celkový počet souřadnic v poli, formát souřadnic (souřadnice, normály atd.) a celkový počet indexů.

Pomocí metod **`setCoordinates(0,ptRead.poleSouradnic)`** a **`setCoordinateIndices(0,indexy)`** nastavíme požadované vstupní parametry (parametr *indexy* je zde pole celých čísel obsahující hodnoty z *ArrayList*u). Prvek *IndexedGeometryArray* poté přiřadíme *Lines* jako *Geometry*, vytvoříme *Appearance* s takřka identickými vlastnostmi jako v případě *Points* a druhý prostorový objekt je hotov.

6.2.3. Surfaces

Plochy složené z trojúhelníků a čtyřúhelníků budeme vytvářet ve dvou velmi podobných metodách **`createTriangles()`** a **`createQuadrangles()`**. Opět použijeme *ArrayList* jako dočasné odkladiště indexů a opět budeme procházet pole řetězců obsahující souřadnice elementů a extrahovat části konkrétních řádků. Pro trojúhelníky budeme potřebovat *IndexedTriangleArray* a tři souřadnice pro každou plochu; geometrii čtyřúhelníků vytvoříme pomocí *IndexedQuadArray* a čtyř souřadnic. Zkrácený kód případu trojúhelníků následuje:

```
if (trianglesCount > 0) {
int[] indexy = new int[trianglesCount*3];
for (int i = 0; i < (trianglesCount*3);i++){
     indexy[i]=(Integer)indexySouradnic.get(i);
}    //indexySouradnic = ArrayList s indexy
IndexedTriangleArray ita = new IndexedTriangleArray
     (ptRead.poleSouradnic.length,
     IndexedLineArray.COORDINATES, (trianglesCount*3) );
try {
     ita.setCoordinates(0,ptRead.poleSouradnic);
     ita.setCoordinateIndices(0,indexy);
}
catch (ArrayIndexOutOfBoundsException e){
     System.out.println("Neplatny index - chyba v datovem
          souboru.");
}
GeometryInfo gi = new GeometryInfo(ita);
NormalGenerator ng = new NormalGenerator();
ng.setCreaseAngle(Math.toRadians(30));
ng.generateNormals(gi);
Triangles.setGeometry(gi.getGeometryArray());
}
```

Dále je potřeba definovat materiál a *Appearance* – plochy by jinak nebyly na světle vidět (pokud by jim nebyla přiřazena barva v *ColoringAttributes*, potom by ovšem scéna ztratila dojem trojrozměrnosti).

Zásadní rozdíl mezi plochami a prvními dvěma objekty (body a čárami) je ten, že u ploch je žádoucí rozdílné chování při různě osvětlené scéně vytvářející onu plastičnost trojrozměrného objektu. Abychom toho dosáhli, je třeba splnit tři podmínky, zmíněné v kapitolách 3.2.3. a 3.2.4.: musíme osvítit scénu (viz. 5.2.3.), definovat materiál objektu a normálové vektory. O poslední podmínku se za nás elegantně postará *NormalGenarator*, který z již zadané geometrie umí vygenerovat normály bez asistence programátora. Příklad nadefinovaného materiálu je uveden v *5.2.3. Světlo*.

6.2.4. Další metody třídy Teleso

Obsahem hlavní veřejné metody je kód obsluhující vznik tělesa:

```
public Teleso(String path){
     ptRead.fileName = path;
     ptRead.pointRead(1);

     Points = createSit(ptRead.meritko);
     Lines = createLines();
     Triangles = createTriangles();
     Quadrangles = createQuadrangles();
}
```

Zde se nejprve provede jedno prozkoumání datového souboru a vzniklý parametr *měřítko* se použije k vytvoření vhodně velkého modelu.

Další metody budou nastavovat průhlednost (*Transparency*) jednotlivých částí scény. Definujeme **`setPointsVisible()`**, **`setPointsInvisible()`** atd. pro jednotlivé položky. Aby bylo možné toto nastavit již zkompilovanému objektu, je nutné nastavit v metodách **`create`** (*createLines* atd.) konkrétní schopnosti, a to:

```
TransparencyAttributes.setTransparencyMode
     (TransparencyAttributes.ALLOW_VALUE_WRITE);
TransparencyAttributes.setTransparencyMode
     (TransparencyAttributes.ALLOW_MODE_WRITE);
```

První schopnost umožní nastavit hodnotu průhlednosti (udává se v rozsahu 0-1, kdy 1 je zcela průhledná) a druhá měnit metodu výpočtu transparence.

Pomocí průhlednosti je možné snadno skrývat či zobrazit různé prvky objektu, v tomto případě body, čáry či plochy. Tato funkce bude ovládána zatrhávacími *checkPoint*y na ovládacím panelu, jak je vidět z obrázku v kap. 5.1.

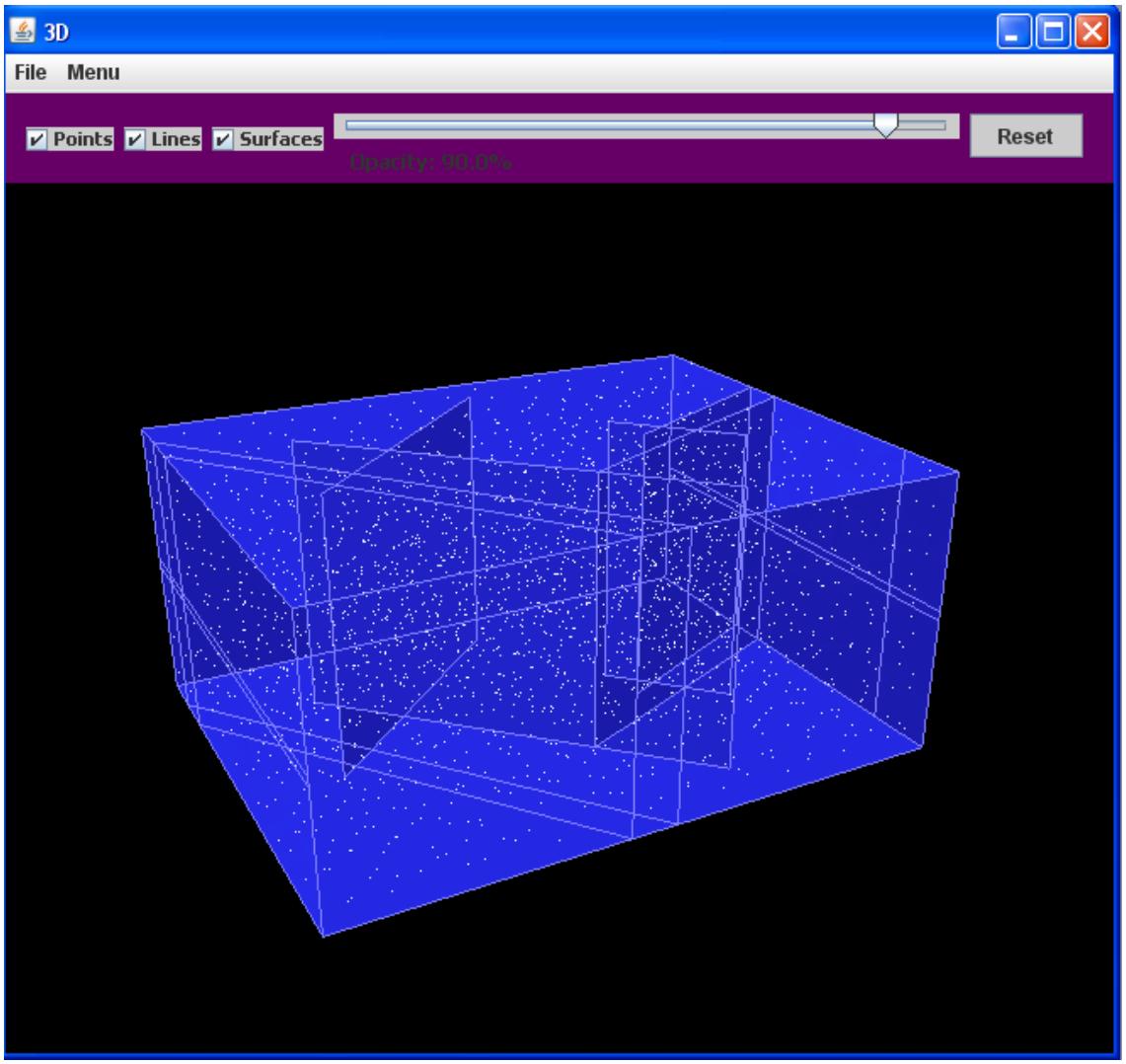
Pokud v této fázi spustíme aplikaci, bude zobrazovat jednotlivé části scény dle výběru ovládacími prvky, s objektem půjde libovolně manipulovat, ovšem při zobrazení ploch bude případná vnitřní struktura zcela skryta pod neprůhledným povrchem.

6.3. Dynamická změna průhlednosti

K zobrazení celé vnitřní struktury objektu lze elegantně využít již hotových vlastností, a to nastavení viditelnosti elementu pomocí *TransparencyAttributes*. Mezi ovládací prvky zařadíme například JSlider, či jiný grafický posuvník a mezi proměnné třídy Main přidáme `float pruhlednostPloch = 0.0f`. Hodnotu této proměnné budeme měnit v závislosti na poloze posuvníku. Jejím účelem bude působit jako parametr v metodě `setSurfacesVisible(float pruhlednost)`, kterou jsme již dříve definovali jako metodu bez parametru, která nastaví průhlednost na nulu (což je zcela neprůhledná a tedy viditelná).

Pokud nezapomeneme při změně hodnoty *slider*u aplikovat nastavení na plochy objektu, budou se tyto „zprůhledňovat“ dle použité metody (pro nejlepší vizuální výsledek použijeme *Blended*, která je však také odpovídajícím způsobem výpočetně náročná).

Finální podoba 3D aplikace je následující:



Obr. 6.: Finální grafická podoba aplikace

7. Hardwarové nároky

Hotovou aplikaci přeložíme do finálního Java archivu *.jar příkazem NetBeans *Build*. Archiv bude uložen v adresáři projektu ve složce Dist. Jeho snadné spouštění lze zajistit vytvořením spouštěcího souboru **.bat* s následujícím textem:

```
java -jar "Nazev_souboru.jar"
```

Alternativně jej lze spouštět identickým příkazem přímo z příkazové řádky (což v podstatě dělá soubor **.bat*).

K bezproblémovému běhu vyžaduje tato aplikace minimálně JRE 5 s nainstalovanými knihovnami Java 3D (v současné době poslední verze je 1.5.0). Ideální je použít nejnovější vydání Javy.

Při vývoji aplikace jsem se setkal s několika zajímavými a důležitými vlastnostmi jazyka Java. Velmi výrazně se projevila deklarovaná zvýšená paměťová náročnost: pouze vývojové prostředí NetBeans si pro běh samotného IDE nárokovalo přes 200 MB operační paměti (nehovořím o stránkovacím souboru, ale pouze fyzické paměti) a každá spuštěná instance mé aplikace zabrala cca 40-50 MB. Programátor v Javě by proto měl mít k dispozici hardwarovou konfiguraci o minimálně 1 GB operační paměti.

Rovněž spouštění aplikace není z nejrychlejších, na mém osobním pc (konfigurace: AMD Athlon 3200+, 1 GB RAM, GeForce 6600) trvá přibližně dvě sekundy, samotný běh programu je však již dostatečně rychlý a žádné zpomalení vlivem platformy JRE jsem nezaznamenal.

7.1. Test grafického výkonu

Osobně jsem byl velmi zvědav na údajnou nativní hardwarovou akceleraci Javy 3D, která dle specifikací již ze své podstaty podporuje standardy OpenGL a DirectX. Abych toto prakticky otestoval, načetl jsem poměrně složité těleso obsahující přibližně 15 000 trojúhelníků. Poté jsem v programu nastavil průhlednost na 15% a v celoobrazovkovém režimu měřil pomocí aplikace *Fraps* frame rate (počet snímků za sekundu) během

animace za různých nastavení grafické karty (různé takty jádra a pamětí). Výsledky mého měření jsou uvedeny v následující tabulce:

Takt jádra/pamětí [MHz]	FPS (snímky za vteřinu)	FPS [%]	takt jádra [%]
420/550	19	136	140
300/500	14	100	100
150/450	7	50	50

Vzhledem k tomu, že ostatní nastavení (např. takt CPU) zůstala nezměněna, mohu s klidným svědomím prohlásit, že výkon Javy 3D je přímo úměrný výkonu grafického jádra.

Zde se jedná o skutečně nejnáročnější nastavení, neboť vzhledem k průhlednosti celého tělesa musí systém propočítávat všech 15 000 trojúhelníků. Při běžné manipulaci s objekty ve full-screen módu se počet snímků za sekundu pohyboval v průměru kolem 50 FPS při standardních taktech grafické karty (300/500 MHz).

8. Závěr

Knihovny Javy 3D se při tvorbě trojrozměrného prostředí ukázaly být velmi komplexním a mocným nástrojem. Obsahují velké množství tříd a funkcí, které lze snadno a efektivně použít k zobrazení libovolných objektů. Jedinou nevýhodu, která vlastně nevýhodou ani není, spatřuji v množství nabízených funkcí, ve kterém se člověk nezasvěcený nebo začínající jen velmi špatně zorientuje.

Kapitolou samo o sobě jsou možnosti animace, které jsou s tímto nástrojem téměř intuitivní a velmi snadno se s nimi pracuje – zjednodušeně by se dalo říci, že programátor pouze zadá parametry transformace a samotnou animaci provede Java sama prostřednictvím různých transformačních skupin.

Během vývoje aplikace jsem se postupně seznamoval s různými vlastnostmi tohoto programovacího jazyka. Java, přes handicap delšího startu vyplývajícího ze samotné podstaty interpretovaného jazyka, není výrazně pomalejší než jiné programovací jazyky. Jakýkoliv minimální rozdíl v rychlosti bohatě vynahradí její početné pozitivní vlastnosti jako stabilita, bezpečnost (z tohoto důvodu je stále rozšířenější mezi serverovými aplikacemi), univerzálnost či uživatelská a zejména programátorská přívětivost.

Jedinou podstatnou nevýhodu Javy lze spatřit ve velké paměťové náročnosti, což ovšem v nových verzích JDK zlepšují různé způsoby sdílení tříd mezi aplikacemi a optimalizace JVM častěji spouštěných programů.

Vyvinutý 3D program v některých ohledech (například dynamické skrývání částí objektu) dokonce překračuje původní zadání, což ovšem neznamená, že již nelze jeho funkčnost něčím doplnit či vylepšit.

Výsledný JAR archiv obsahující zkompilované třídy má velikost cca 20 kB, což je velké plus při stahování aplikace z internetu prostřednictvím pomalejšího připojení. Proto, pro svou dynamičnost a mnohé pozitivní vlastnosti je Java nejen jazykem dnešního internetu ale i budoucích aplikací.

Prameny:

VIRIUS, Miroslav. *Java pro zelenáče*. Praha: Neocortex, 2001. 240 stran.
ISBN 80-902230-9-5

TOMAN, Petr. DIONÉ - Studentský Informační Server: *Programovací jazyk Java*. [online]. 2006. URL: <http://dione.zcu.cz/java/>

KOTALA, Z. – TOMAN, P. *Java – popis jazyka (sborník)*. [online]. Západočeská univerzita, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky, 2001. Vedoucí projektu: Ing. Pavel Herout. URL: <http://dione.zcu.cz/java/sbornik.html>

HOPKINS, Greg. *Java 3D Tutorial.* [online]. 2001.
URL: <http://www.java3d.org/tutorial.html>

KUŽELKA, Ondřej. *Java a 3D grafika*. [online]. Interval.cz, 2004. Série článků.
URL: <http://interval.cz/clanky/java-a-3d-grafika-uvod/>

Sun Microsystems, Inc. *Sun Developer Network – Java Technology*. [online].
URL: <http://java.sun.com/>

Sun Microsystems, Inc. *Java™ Platform, Standard Edition 6 API Specification.* [online]. 2006. URL: < http://java.sun.com/javase/6/docs/api/index.html>

Sun Microsystems, Inc. *Java 3D 1.5.0 Documentation*. 2006. Volně ke stažení na URL: < http://java.sun.com/products/java-media/3D/download.html>

Sun Microsystems, Inc. *NetBeans homepage*. [online].
URL: <http://www.netbeans.org/