Software pro simulaci mnohočásticových soustav

Software for particle simulations

Bc. Tomáš Dřímal

Diplomová práce
2010

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2009/2010

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Tomáš DŘÍMAL**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Software pro simulaci mnohočásticových soustav**

Zásady pro vypracování:

Mnoho fyzikálních systémů je tvořeno velkým množstvím vzájemně interagujících částic.
Cílem práce je vytvoření programu, umožňujícího po doplnění fyzikálních interakcí popsat časový vývoj takových jevů jako je např. difuze, sedimentace, chování plynu, pohyb těles ve Sluneční soustavě.

1. **Vypracujte literární rešerši na téma simulace mnohočásticových soustav.**
2. **Navrhněte vhodné softwarové prostředí pro návrh aplikace.**
3. **Napište a otestujte aplikaci pro simulaci mnohočásticových soustav.**

Rozsah práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **Herout P.: Učebnice jazyka Java, Nakladatelství Kopp, 2006**
2. **Herout P.: Java – bohatství knihoven, Nakladatelství Kopp, 2008**
3. **Greenspan D.: Particle Modeling (Modeling and Simulation in Science, Engineering and Technology) Birkhauser, 1997**
4. **Hockney R.W, Eastwood J.W: Computer Simulation Using Particles, Taylor & Francis, 1999**
5. **Birdsall C.K., Langdon A.B : Plasma Physics via Computer Simulation, Taylor & Francis, 2004**
6. **Tremaine S.: Galactic Dynamics, Princeton Series in Astrophysics, 2004**
7. **Lau H.T.: A Numerical Library in Java for Scientists and Engineers, Chapman & Hall 2003**
8. **Thomas B: Integrated Graphic and Computer Modelling, Springer 2008**
9. **Kol. autorů: Java 6 – Výukový kurz, Computer Press, 2007**
10. **Elimelech M et al: Particle Deposition & Aggregation: Measurement, Modelling and Simulation, Butterworth-Heinemann, 1998**

Vedoucí diplomové práce: **doc. RNDr. Petr Ponížil, Ph.D.**
Ústav fyziky a mater. inženýrství

Datum zadání diplomové práce: **19. února 2010**

Termín odevzdání diplomové práce: **8. června 2010**

Ve Zlíně dne 19. února 2010

L.S.

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky

prof. Ing. Vladimír Vašek, CSc.
děkan

prof. Ing. Vladimír Vašek, CSc.
ředitel ústavu

Abstrakt

Cílem této diplomové práce bylo vytvořit program pro simulaci pohybu částic v mnohočásticových soustavách. Požadavky na simulaci byly, aby bylo možné sledovat pohyb jednotlivých částic na obrazovce a také, aby bylo možné exportovat vlastnosti částic v numerickém tvaru do souborů. Tyto požadavky byly splněny, postup tvorby samotného programu se nachází v praktické části této práce.

Klíčová slova: simulace částic, Java, programování

Abstract

The goal of this master thesis was to create program for particle simulation in many particle systems. The requirements was to draw the particle moving into the screen and to write the numerical attributes of particles into the text files. This requiremnts was satisfied. How the program was created is described in practical part.

Keywords: particle simulation, Java, programming

Na tomto místě bych chtěl zejména poděkovat svému vedoucímu, doc. RNDr. Petrovi Ponížilovi Ph.D., za téma této práce a její odborné vedení. Také děkuji tvůrcům svobodného software, protože tato práce i samotný program byly vytvořeny v programech jimi vytvořenými.

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

. . . . . .

podpis diplomanta

Obsah

Úvod

Tato práce se zabývá tvorbou programu pro simulaci pohybu částic v mnohočásticových soustavách. Jedná se například o molekuly plynu či planety sluneční soustavy. Jako programovací jazyk byl zvolen jazyk Java, který sice není pro tyto účely nejvhodnější, protože se jedná o interpretovaný programovací jazyk a jeho rychlost je ve srovnání s kompilovanými jazyky nižší, ale je dostačující, protože se tyto rychlosti liší pouze v řádech jednotek.
Pokud má být uvažována změna počtu částic v mnohočásticové soustavě, tak je potřeba měnit počet částic minimálně o jeden řád. K výraznému zpomalení by tedy došlo i v případě kompilovaného jazyka, takže v této aplikaci není tato úvaha o rychlosti relevantní.
Programovací jazyk Java byl zvolen z toho důvodu, že program pro simulaci částic je v podstatě jen kostra programu, kterou bude možné dále rozšiřovat. Z toho důvodu je vhodnější zvolit jazyk Java, který je pro neprogramátora (například fyzik, který bude doprogramovávat další metody vzájemných interakcí částic) syntakticky jednodušší.
Pro komunikaci a sdílení zdrojových kódů s vedoucím práce byl zvolen verzovací systém SVN, který nejenže umožňuje sdílet zdrojové kódy, ale uchovává i informace o provedených změnách, takže je možné vracet se k předchozím verzím. Zdrojové kódy programu jsou zveřejněny na webové adrese `http://code.google.com/p/ptcssim/` pod licencí *GNU General Public Licence v3*. Je možné je modifikovat a dále distribuovat.

I. Teoretická část

1 Fyzikální základ

Při simulaci pohybu částic jsme vycházeli z představ klasické mechaniky. Pro urychlení a zjednodušení výpočtů, považujeme každou částici za hmotný bod. Hmotný bod je myšlený objekt, který z hlediska vzájemného působení s jinými hmotnými body má vlastnosti reálného tělesa (zejména jeho hmotnost) a jsou zanedbány jeho rozměry, tvar nebo orientace v prostoru, které při vyšetřování pohybu tělesa nejsou významné. Dalším zjednodušením je zredukování původně prostorového (3D) problému na rovinný (2D). Toto zjednodušení bylo zvoleno jednak kvůli zrychlení výpočtu, ale zejména tím bylo umožněno jednoduché vykreslování drah částic.
V mechanice je rychlost v definována jako derivace polohového vektoru r podle času t:

$$v = \frac{dr}{dt} \tag{1}$$

Zrychlení a je definováno jako derivace rychlosti v podle času t:

$$a = \frac{dv}{dt} \tag{2}$$

Podle druhého Newtonova zákona:

$$F = ma \tag{3}$$

je zrychlení hmotného bodu přímo úměrné výslednici sil F, které na hmotný bod působí a nepřímo úměrné hmotnosti hmotného bodu m.
Při simulaci pohybu částic postupujeme následujícím způsobem:

1. Pro dané rozložení částic v prostoru v čase t_0 vypočítáme výslednici všech sil, které na každou částici působí.

2. Podle vztahu (3) vypočítáme vektor zrychlení částice.

3. Podle vztahu (2) lze spočítat změnu rychlosti částice během časového intervalu $\Delta t : \Delta v = a\Delta t$. Rychlost po čase Δt pak bude $v_1 = v_0 + a\Delta t$, kde v_0 je rychlost na začátku časového intervalu Δt.

4. Polohu částice na konci časového intervalu Δt lze spočítat podle vztahu (1):

   $\Delta r = v\Delta t$.

   Polohu po čase Δt pak lze spočítat podle:

$$r_1 = r_0 + v_0\Delta t, \tag{4}$$

kde v_0 je rychlost a r_0 poloha částice na začátku intervalu Δt. V průběhu časového intervalu Δt se rychlost změnila z v_0 na v_1. Stejně dobře bychom jako rychlost v průběhu intervalu mohli použít i v_1. Lepší, ale výpočetně o něco náročnější možností je použít průměrnou rychlost $(v_0 + V_1)/2$.

Po provedení popsaných čtyř kroků máme polohu a rychlost částice po čase Δt a můžeme znovu pokračovat bodem 1 s tím, že vypočítáme novou sílu působící na částici.

Popsaný postup je programátorsky i výpočetně jednoduchý. Jeho omezením je předpoklad, že síla a tedy i zrychlení je po celý časový interval konstantní. Toho je možné dosáhnout pouze tím, že časový interval zvolíme hodně krátký a tím také zpomalíme celý výpočet.

Pro program bylo plánováno využití při simulaci sedimentace částic. To znamená, že na částice se díváme jako na částice ideálního plynu, které na sebe mimo okamžik dotyku nepůsobí žádnými silami a při srážce se chovají jako dokonale pružné koule. Bylo zjištěno, že spočítat interakce pružných koulí je velmi náročné. V každém kroku je nejdříve třeba vypočítat jestli vůbec došlo ke srážce částic a v případě, že ano, řešit srážku jako obecný dokonale pružný ráz koulí. To je algoritmicky i výpočetně tak náročné, že jsme od této aproximace upustili a zvolili elektrickou interakci mezi částicemi. V této aproximaci se na částice díváme jako na elektricky nabité se stejným nábojem a odpudivá síla mezi nimi je pak popsána Coulombovým zákonem:

$$F = \frac{1}{4\pi\varepsilon}\frac{q_1 q_2}{r^2} \tag{5}$$

Při použití Newtonova gravitačního zákona

$$F = G\frac{m_1 m_2}{r^2} \tag{6}$$

jako popisu přitažlivé interakce mezi částicemi lze pak program použít např. pro modelování pohybu těles ve sluneční soustavě nebo hvězd v galaxii.

2 Programovací jazyky a vývojová prostředí

Součástí této diplomové práce bylo napsat program, který bude simulovat pohyb částic a jejich vzájemné interakce. Programovací jazyk jsem vybral Javu a pracoval jsem ve vývojovém prostředí NetBeans.

2.1 Java

Programovací jazyk Java byl vyvíjen společností Sun Microsystems od roku 1991, v roce 2009 tuto společnost i s projektem NetBeans koupila společnost Oracle. Původně byl vyvíjen pro vestavěná zařízení[1], v roce 1993 si však tvůrci uvědomili rostoucí vliv WWW a rozšířili tento projekt na tvorbu aplikací pro web. V roce 1995 byl představen. Od té doby jeho popularita roste a v současné době jej lze nalézt jak ve webových aplikacích, tak i v mobilních telefonech a dalších zařízeních.

Programy napsané v Javě nejsou závislé na platformě, na které budou spuštěny. Zdrojový kód, uložený v souboru s příponou `.java`, je překladačem (SDK – Software Development Kit) přeložen do takzvaného byte-kódu. Ten je uložen v souboru s příponou `.class`. Tento byte-kód může být spuštěn na libovolném počítači na libovolné platformě. Jedinou podmínkou je, že na daném stroji je nainstalována JVM (Java Virtual Machine). Ta zajistí „dopřeložení“ zdrojového kódu pro daný operační systém a jeho spuštění. [1]

Java je striktně objektově orientovaný jazyk. Jinak, než objektově v něm programovat nelze. Dokonce i datové typy jsou třídami, kromě typů základních (`integer, float, char, ...`). I tyto primitivní typy mají však svoje objektové ekvivalenty (`Integer, Float, Char, ...`), jejichž názvy se liší jen ve velikosti prvního písmene. Například pole už je ale objektem. Tedy v případě, že se pole jmenuje `pole1`, tak velikost (délku) nezjistíme, že na toto pole zavoláme funkci `Length` v tomto tvaru: `Delka = Length(pole1)`, ale tak, že zavoláme metodu dané instance `pole1`: `Delka = pole1.Length`. Tato vlastnost má spoustu výhod, například jednoduchou znovupoužitelnost kódu, ale odrazuje od tohoto jazyka začátečníky a dává vznikat fámám, že je nejsložitějším jazykem. Ve srovnání s C/C++ je však jednodušší, protože se například automaticky stará o uvolňování paměti. [4]

[1] jedná se o běžná elektronická zařízení, jak například pračky, mikrovlnné trouby a podobné

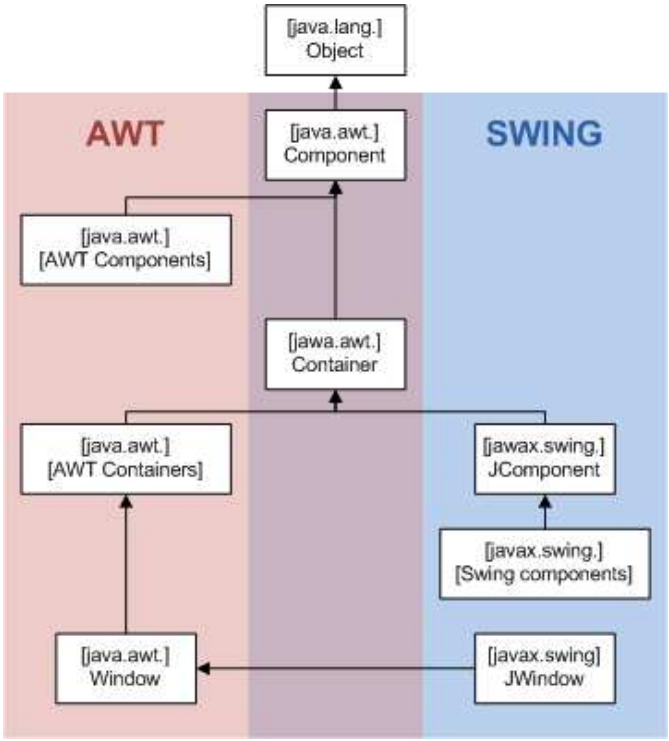
2.1.1 Kapitoly Javy použité v praktické části

2.1.1.1 Java Swing

Swing je knihovna uživatelských prvků na platformě Java pro ovládání počítače pomocí grafického rozhraní (takzvaného GUI). Knihovna Swing poskytuje aplikační rozhraní pro tvorbu a obsluhu klasického grafického uživatelského rozhraní. Pomocí Swingu je možno vytvářet okna, dialogy, tlačítka, rámečky, rozbalovací seznamy, atd... [10]

Historie vývoje Původním nástrojem pro tvorbu uživatelského prostředí (GUI) je knihovna AWT vyvíjená firmou Sun Microsystems jako součást Javy. Jeho první verze vyšla v roce 1995 spolu s první verzí jazyka Java. Během vývoje se však v AWT objevily chyby a taky bylo závislé na platformě, takže porušovalo jeden z proncipů jazyka Java. V roce 1997 Sun Microsystems vývoj AWT ukončil a začal vyvíjet Swing. [11]

Hierarchie Knihovna Swing je založena na knihovně AWT. Všechny objekty (okna, tlačítka, apod.) mají za rodiče třídu `Object`. Z této třídy je zděděna třída `Component`. Její instance představují objekty, které mají svou grafickou reprezentaci a mohou interagovat s uživatelem.



Obrázek 2.1: Hierarchická struktura tříd knihovny Swing [10]

Od třídy `Component` pak dědí třída `Container`, což je třída, která dokáže držet další komponenty. Obsahuje správce rozvržení, který se stará o to, jak jsou vložené komponenty v tomto kontejneru graficky rozloženy. V AWT jsou od třídy `Container` odvozeny například komponenty `Window` nebo `Panel`.[11]
Základním kamenem knihovny Swing je třída `JComponent`. Tato je rodičem každé komponenty z kihovny Swing. Vzhledem k tomu, že tato třída je potomkem třídy `Container`, tak v knivovně Swing může každá komponenta sdružovat další. Potomci třídy `JComponent` jsou pak již samotnými ovládacími prvky, jako je například *JButton* (tlačítko) nebo `JPanel` (obecný panel). Všechny komponenty z knihovny Swing začínají písmenem `J`, aby byly odlišitelné od svých protějšků z AWT.
V této hierarchii však je výjimka, a tou jsou okna a všechny ostatní nejvyšší kontejnery. Okna jsou obecně potomci třídy `Window` a nemohou být potomky `JComponent`, protože v Javě nelze dědit o více tříd. Grafické znázornění celé hierarchie je na obrázku 2.1. [11]
Ukázka zdrojového kódu, který vytvoří okno a v něm vypíše známý řetězec „Hello World“:

```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloWorld
{
public static void main(String[] args){
//vytvoreni okna
JFrame frame = new JFrame("Hello World example");
//pridani textu do okna
frame.add(new JLabel("Hello World!"));

// zmenseni velikosti okna podle potreby
frame.pack();
// nastaveni, ze bude po zavreni okna ukoncen program
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//vycentrovani okna na obrazovce
frame.setLocationRelativeTo(null);

// zobrazeni okna
frame.setVisible(true);
}
}
```

[10]

2.1.1.2 Thread

Java umožňuje `multithreading`, neboli paralelní běh dvou či více částí programu najednou. Této vlastnosti využije například Internetový prohlížeč – může nahrávat obrázky po síti, zároveň formátovat a zobrazovat WWW stránku a ještě k tomu spouštět applet. Nebo například textový editor, který může provádět kontrolu pravopisu, zatímco uživatel píše dokument.
Každá paralelně běžící část programu se v Javě nazývá vlákno (`thread`). Důležité je, že jednotlivá vlákna lze naprogramovat téměř nezávisle a pouze v případě, že sdílí společná data nebo používají stejné prostředky (zařízení), je třeba zajistit jejich řádnou synchronizaci.
V současné době se již mnohdy jedná skutečně o souběžný (paralelní) běh více vláken, protože mnoho současných procesorů je již vícejádrových.
Každé vlákno v Javě je instancí třídy Thread (z balíku java.lang) nebo jejího potomka. Tato třída definuje základní metody jako je spuštění, zastavení a ukončení vlákna. Jednoduché vlákno se naprogramuje tak, že se vytvoří potomek třídy Thread, který definuje metodu `public void run()` obsahující kód, který bude prováděn paralelně. Spuštění vlákna se provede zavoláním metody `start()` (metoda `run()` se přímo nevolá).
Nejdůležitější metody definované ve třídě `Thread` jsou:
Veřejné konstruktory: `Thread(), Thread(Runnable r), Thread(String s), Thread(String s, Runnable r), Thread(ThreadGroup g, Runnable r, String s), Thread(ThreadGroup g, Runnable r), Thread(ThreadGroup g, String s)`
`public String getName()` – vrací jméno vlákna
`public void join()` – čeká na ukončení vlákna
`public void resume()` – probudí vlákno "odstavené"metodou suspend()
`public void run()` – jádro vlákna (definované jako prázdná metoda)
`public void start()` – zahájí běh vlákna, tj. provádění metody run()
`public static void sleep(long ms)` - uspí (dočasně zastaví) vlákno na `ms` milisekund. Pak jej probudí a vlákno může pokračovat v běhu
`public void stop()` – ukončí vlákno
`public void suspend()` – "odstaví"vlákno. Odstavené vlákno může probudit pouze metoda `resume()`
`public static void yield()` – umožní běh jiného vlákna [1]

2.1.1.3 SAX – Simple API for XML

Toto API umožňuje sériový přístup k XML dokumentu. Dokument je zpracováván proudově, je tedy rozdělen podle své struktury a tyto části jsou pak zpracovávány. Postupně se volají události, které ohlašují nalezení konkrétní části. Programátor má k dispozici tyto události a podle potřeb programu lze tyto události ošetřovat. [2]
XML dokument je textový soubor (Unicode), který je vytvořen v syntaxi XML, tedy obecného značkovacího jazyka. V českém prostředí se nejčastěji používá kódování UTF-8. Na rozdíl od např. HTML, efektivita XML je silně závislá na struktuře, obsahu a integritě. Aby byl dokument považován za správně strukturovaný, musí mít nejméně následující vlastnosti:

- Musí mít právě jeden kořenový (root) element.
- Neprázdné elementy musí být ohraničeny startovací a ukončovací značkou. Prázdné elementy mohou být označeny tagem „prázdný element“.
- Všechny hodnoty atributů musí být uzavřeny v uvozovkách – jednoduchých (') nebo dvojitých ("), ale jednoduchá uvozovka musí být uzavřena jednoduchou a dvojitá dvojitou. Opačný pár uvozovek může být použit uvnitř hodnot.
- Elementy mohou být vnořeny, ale nemohou se překrývat; to znamená, že každý (ne kořenový) element musí být celý obsažen v jiném elementu.

Jména elementů v XML rozlišují malá a velká písmena: např. `<Particle>` a `</Particle>` je pár, který vyhovuje správně strukturovanému dokumentu, pár `<Particle>` a `</particle>` je však chybný. [3]
V programovacím jazyce Java se SAX nachází v knihovně `javax.xml.parsers` a `org.xml.sax`. Pro vytvoření SAX parseru je třeba následující zdrojový kód:

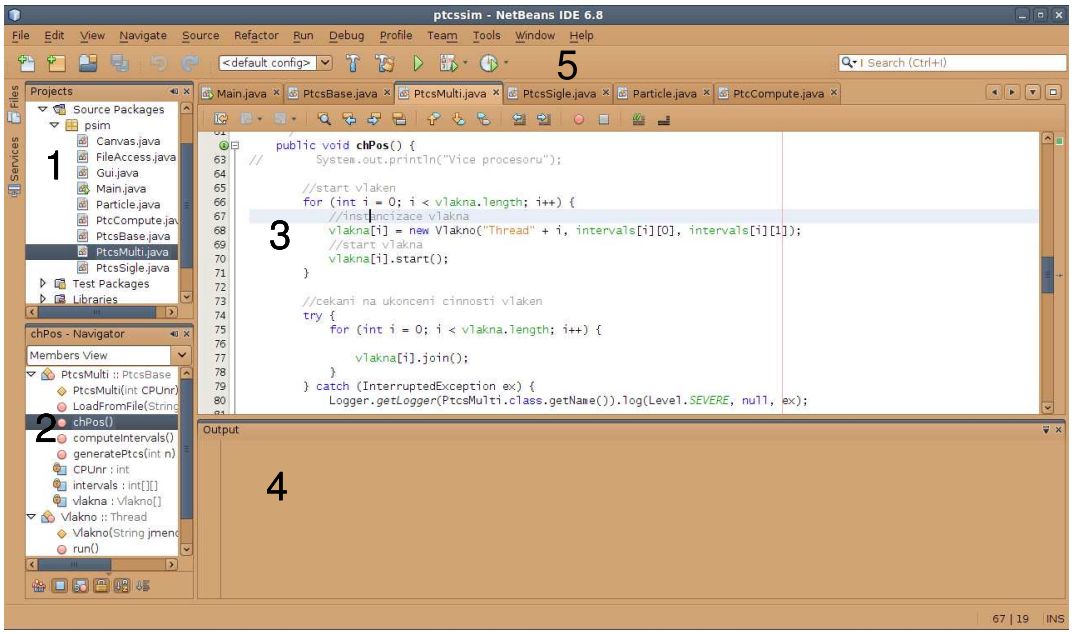
```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating(false);
SAXParser parser = factory.newSAXParser();
XMLReader reader = parser.getXMLReader();
FileAccess test = new FileAccess();
reader.setContentHandler(test);
reader.setErrorHandler(test);
reader.parse(name);
```

Tato sekvence příkazů instanciuje XML SAX parser a spustí proces parsování. Vyvolává události, které je potřeba obsloužit. Konkrétně se jedná o tyto metody:

`public void startDocument()` – metoda volána při začátku dokumentu
`public void startElement(String uri, String localName, String qName, Attributes atts)` – metoda volána při začátku elementu (tagu)
`public void characters(char[] ch, int start, int length)` – metoda, která je volána při čtení znaků mezi elementy
`public void endDocument()` – metoda volána na konci dokumentu
Seznam událostí, které jsou volány není úplný, více jich však při tvorbě programu pro simulaci mnohočásticových soustav nebylo potřeba.

2.1.2 Vývojové prostředí pro práci s Javou

Programátor v Javě si vystačí jen s SDK. Zdrojový kód napíše v libovolném programu pro tvorbu textových souborů (Poznámkový blok, PSPad, ...) a přeloží a spustí pomocí příkazové řádky. Toto vybavení však neposkytuje kontrolu syntaktických chyb a práce s ním je nepraktická. Proto existují IDE pro práci s Javou, jako například NetBeans[2]. Toto vývojové prostředí jsem použil pro práci s Javou. Jak je vidět na obrázku 2.2, tak se okno



Obrázek 2.2: Snímek vývojového prostředí NetBeans

prostředí NetBeans skládá z několika částí. Hlavní okno (5) obsahuje hlavní nabídku a tlačítka typu otevření projektu, jeho uložení a další. Okno Projects (1) obsahuje aktuální projekty, které jsou momentálně otevřené. Z nich je jeden hlavní, se kterým se pracuje a lze jej spouštět a debugovat. V části (2) je okno Navigator, ve kterém je zobrazena interaktivní

[2]Open source projekt sponzorován společností Oracle (dříve Sun Microsystems)

struktura zrovna editovaného souboru. Jsou zde zobrazeny třídy z tohoto souboru, jejich členské proměnné a metody. Kliknutím na tyto položky je přesunut kurzor editoru (3) na místo deklarace dané položky. V části (3) je editor, ve kterém jsou v jednotlivých záložkách otevřeny zdrojové kódy jednotlivých tříd. V oblasti (4) je konzole, ve které běží Java aplikace vytvořené bez GUI (Graphical User Interface), tedy takové, které s uživatelem komunikují jen prostřednictvím příkazové řádky. Zde se také nachází okno s varováními a s chybami.

3 Subversion (SVN)

Subversion (zkráceně SVN) je systém pro správu a verzování zdrojových kódů. Je to náhrada za starší systém CVS. Je to multiplatformní systém, který pracuje na principu klient-server. Jedná se o to, že zdrojové kódy projektu jsou uloženy na serveru. Programátoři pracují na svých lokálních kopiích. V okamžiku, kdy programátor udělá nějakou změnu, nahraje upravené zdrojové kódy na server. Ten přjme změny a vytvoří novou revizi, což je poslední verze.
Výhodou tohoto systému je, že na projektu může pracovat více programátorů najednou a že jsou uloženy informace o všech změnách. Lze se tedy vrátit ke všem změnám.

3.1 Základní pojmy a postupy při práci s SVN

Repozitář je centrální úložiště. Zde jsou uloženy všechny soubory se zdrojovými kódy. Je součástí serverové části a fyzicky jsou soubory umístěny na souborovém systému serveru. K repozitáři se přistupuje přes Repository Access Layer a správa se provádí klientskými nástroji. Pro každý projekt je potřeba jednoho repozitáře.
Branch je větev, která analogicky odpovídá adresáři na klientské straně aplikace.
Revize je pořadové číslo každé změny. Revize slouží ke sledování změn v čase. Pokud klient provede změnu byť jen v jedné větvi, tak dojde k nové revizi celého repozitáře. Každá revize obsahuje informace o tom kdy byla provedena, kým a obsahuje podrobnosti o provedených změnách. Každá revize také obsahuje poznámku tvůrce revize.
Pracovní kopie je kopie repozitáře, nebo jen větve, na straně klienta. Programátor pracuje s touto pracovní kopí a provádí v ní změny.
Commit je nahrání změn provedených v pracovní kopii na server. Jedná se o nejčastěji prováděnou operaci s repozitáři. Pokud dojde k chybě při přenosu, tak není vytvořena nová revize.
Konflikt je stav, který signalizuje, že objekt, který má být právě commitován byl změnen někým jiným a nachází se v repozitáři v aktuální revizi v jiné podobě, než v jaké je v pracovní kopii. Je třeba aktualizovat pracovní kopii a vyřešit konflikt, aby bylo možné objekt commitovat.
Changeset je sada změn, které se posílají z pracovní kopie do repozitáře. Subversion ukládá jen změny mezi jednotlivými revizemi, neukládá celé soubory. Snižuje se tím objem dat přenesených od klienta na server a taky místo na disku na straně serveru. [5]

3.2 Subversion v NetBeans

V IDE NetBeans (část 2.1.2) je integrován SVN klient prostřednictvím zásuvného modulu. Volby, jak jsou uvedeny v části 3.1, se nacházejí v oblasti *Projects* (na obrázku 2.2 v části 1). Po kliknutí na hlavní projekt pravým tlačítkem myši se zobrazí kontextová nabídka, která v podnabídce *Subversion* nabízí možnosti pro práci s verzováním daného projektu. Jsou zde položky jako commit (aktivní pouze v případě, že byly se soubory v pracovní kopii provedeny změny od posledního commitu), update, show diff nebo resolve conflicts. Prostřednictvím tohotu zásuvného modulu lze obsluhovat repozitář bez potřeby dalšího klientského programu.

3.3 SVN – serverová část

Pro využívání výhod SVN (kapitola 3) je potřeba mít, kromě klienta (např. 3.2), i serverovou část celého systému. Jednou možností je zprovoznit si vlastní Subversion server (například na základě open sourcových projektů), nebo v případě, že se jedná o projekt s otevřenou licencí (např. GNU a podobné), tak je možné využít služeb veřejných SVN serverů. Ty jsou například kenai.com, unfuddle.com nebo Google Code. Při programování programu pro simulaci mnohočásticových soustav byla využita služba Google Code.

3.3.1 Google Code

Google Code je služba společnosti Google Inc. Jedná se mimo jiné i o službu, která umožňuje bezplatně umístit repozitář se zdrojovými kódy a tento pak na tomto serveru spravovat.

Na stránce `http://code.google.com/hosting/` (obr. 3.1) je v dolní části odkaz *Create a new project*. Po kliknutí na tento odkaz se zobrazí webový formulář, který po zakladateli projektu vyžaduje určité údaje týkající se zakládaného projektu. Konkrétně se jedná o název projektu, jeho popis, požadovaný verzovací systém (kromě SVN je možné vybrat systém Mercurial), licenci projektu a klíčová slova projektu, podle kterých jej bude možné vyhledat. Licence projektu jsou k dispozici *Apache License 2.0*, *Artistic License/GPL*, *Eclipse Public License 1.0*, *GNU General Public License v2*, *GNU General Public License v3*, *GNU Lesser General Public License*, *MIT License*, *Mozilla Public License 1.1* a *New BSD License*.

Po zadání všech těchto údajů a jejich odeslání na server je vytvořen nový projekt a je možné na něm začít pracovat. Google Code jednak umožňuje správu prostřednictvím SVN klienta, a taky poskytuje webové rozhraní, kde lze projekt procházet, sledovat změny

Obrázek 3.1: Úvodní stránka při tvorbě nového projektu na Google Code.

v jednotlivých revizích a porovnávat jednotlivé revize. Dále lze přidávat účastníky na projektu a přidělovat jim práva, jako vývojář, administrátor projektu a přispěvatel.

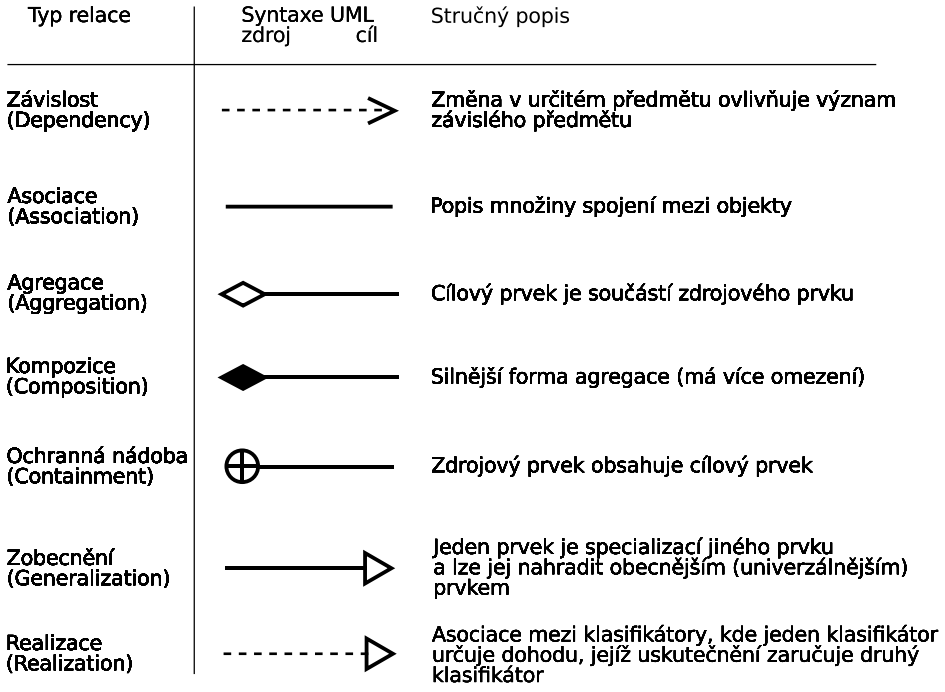
4 Jazyk UML

Jazyk UML (Unified Modeling Language, unifikovaný modelovací jazyk) je univerzální jazyk pro modelování systémů. Přestože je nejčastěji spojován s modelováním objektově orientovaných softwarových systémů, má mnohem širší využití, což vyplývá z jeho zabudovaných rozšiřovacích mechanismů. Jazyk UML byl navržen proto, aby spojil nejlepší existující postupy modelovacích technik a softwarového inženýrství. [6]
Jazyk UML byl použit při návrhu software pro simulaci mnohočásticových soustav, proto jsou v praktické části uvedeny UML diagramy popisující funkci a strukturu programu.

4.1 Stavební bloky jazyka UML

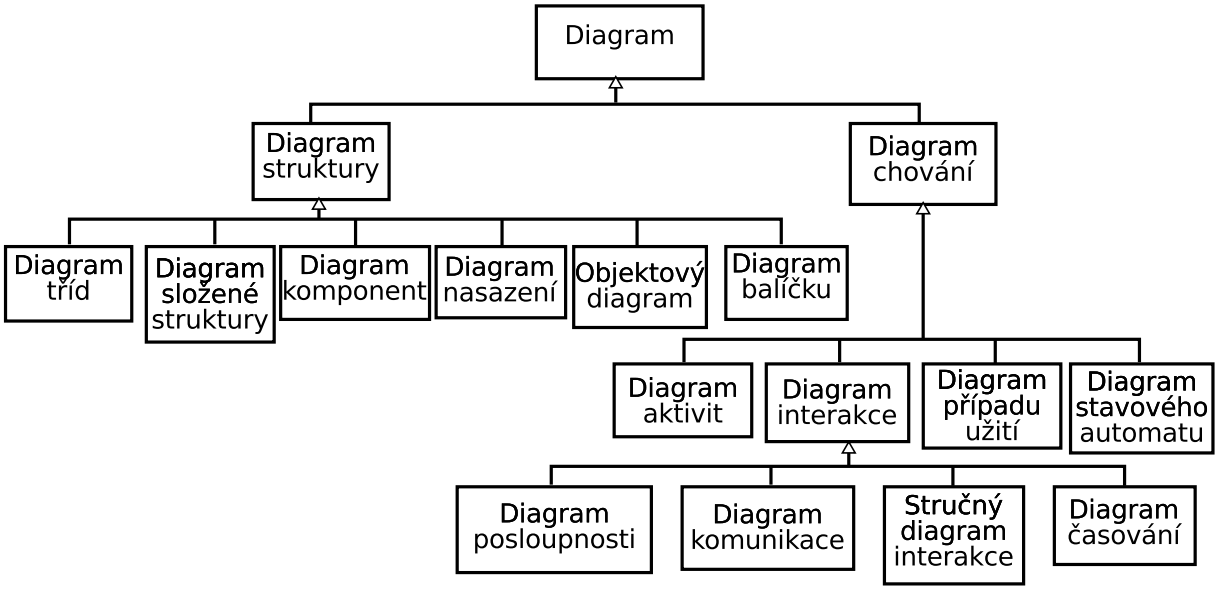
Jazyk UML se zkládá z pouhých tří základních stavebních bloků:

- Předměty – jsou označovány jako „podstatná jména“ modelů UML.
- Relace – umožňuje ukázat na modelu, jaký je vztah mezi dvěma předměty. Stručný přehled jednotlivých druhů relací v jazyce UML je na obrázku 4.1:



Obrázek 4.1: Přehled druhů relací používaných v jazyce UML

- Diagramy – jsou pohledy na model. Celkem existuje třináct různých typů diagramů (UML2), všechny jsou zobrazeny na obrázku 4.2.

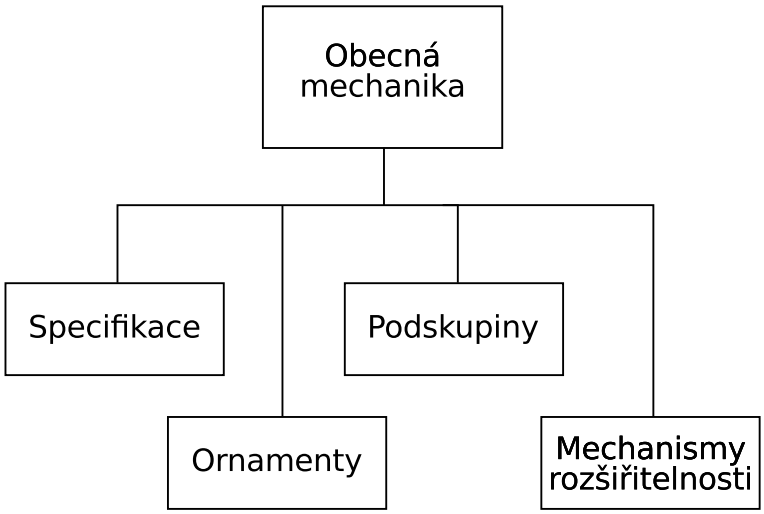


Obrázek 4.2: Přehled druhů diagramů používaných v jazyce UML

[6]

4.2 Mechanika jazyka UML

Jazyk UML obsahuje čtyři společné nechanismy využívané v celém jazyku konzistentně. Popisují čtyři strategie cesty k modelování objektů, jež jsou opakovaně používány v různých kontextech v celém UML.



Obrázek 4.3: Čtyři strategie cesty k modelování objektů jazyka UML

Specifikace Modely UML mají alespoň dva rozměry – grafický, který umožňuje vizualizovat model a textový, jenž se skládá ze specifikací různých prvků modelů. Specifikace jsou textovým popisem sémantiky jednotlivých prvků. Například třída `BankovniUcet` lze vyjádřit, jako schránku s několika přihrádkami, dělící symbol na několik oddílů. Tato podoba ovšem nic nesděluje obchodní (podnikatelské) podstatě této třídy. To je právě úkolem sémantiky.

Ornamenty Každý prvek v jazyce UML lze vyjádřit velmi jednoduchým symbolem, který lze obohatit velkým množstvím ornamentů – rozšiřujících symbolů, které sdělují další informace.

Podskupiny V UML jsou dvě podskupiny: skupina klasifikátorů a instancí (klasifikátor je obecný popis určitého třídy předmětů, kdežto instance je jeden konkrétní jedinec z této třídy) a skupina rozhraní a implementace (rozhraní je to, co předmět vykonává a implementace je to, jak to vykonává).

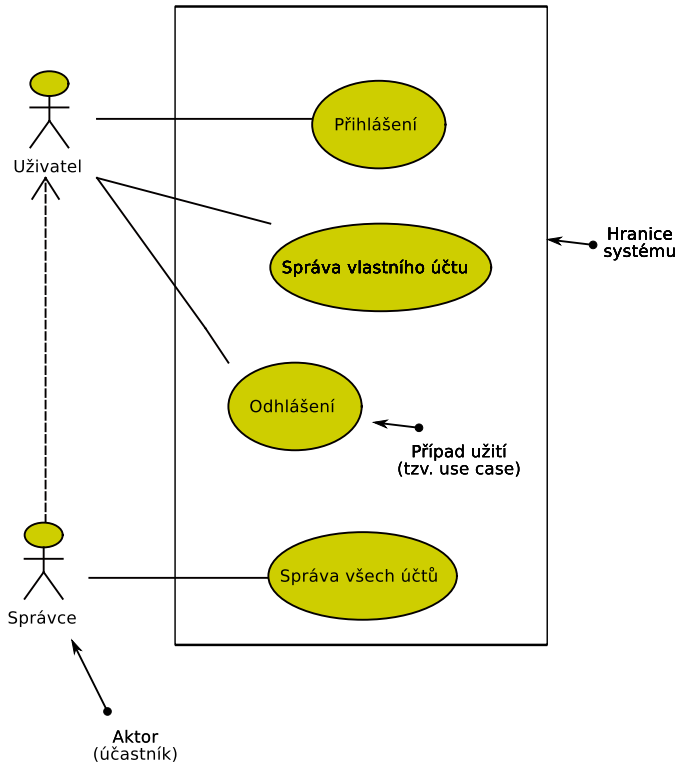
Mechanismy rozšiřitelnosti Autoři UML zahrnuli do jazyka mechanismy, které do UML umožňují přidávat nová pravidla do sémantiky, definovat nové prvky a rozšiřovat specifikaci prvků. [8]

4.3 UML diagramy

V této části jsou stručně představeny diagramy, které budou použity v praktické části při návrhu simulátoru pohybu částic.

4.3.1 Diagram případů užití

Je to diagram, který bývá vytvořen proto, aby byla jasná funkce programu či systému, pro který je tento diagram vytvářen. V případě zákaznicky orientované tvorby programu je tento diagram součástí zadání. Je tvořen tak, aby mu rozuměl i ten, kdo tvorbě programů a systémů nerozumí.
Případy užití napomáhají nalezení hraniční čáry mezi systémem a jeho okolím a informují, jaké jsou vazby systému k okolí. Pod pojmem okolí systému rozumějme uživatele, procesy a vztahy, jež sice systém ovlivňují, ale nejsou jeho přímou součástí.
Jak je vidět na obrázku 4.4, tak se diagram případů užití skládá z účastníků, případů užití, vazeb mezi těmito dvěma druhy prvků a hranic navrhovaného systém. Ukázkový diagram by se dal slovně popsat například takto: Do navrhovaného systému se bude uživatel moci



Obrázek 4.4: Příklad diagramu případů užití

přihlásit a odhlásit. Bude mít možnost spravovat svůj vlastní účet. Správce bude mít ty samé možnosti, jako uživatel, dále ale bude moci spravovat i účty ostatních, tedy účty všech. Hranice systému znázorňuje, co všechno je součástí systému a co už ne.

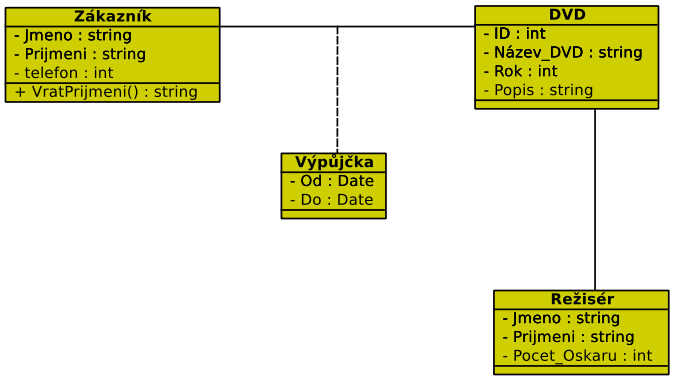
4.3.2 Diagram tříd

Diagram tříd zobrazuje statický pohled na systém, tj. zejména třídy (class) jako typy objektů, obsah tříd a statické vztahy, které mezi nimi existují. Může obsahovat též balíky (pakckage). Může obsahovat elementy chování (behavioral), např. operace, ale jejich dynamika je vyjádřena jinými diagramy.

Na obrázku 4.5 je příklad diagramu tříd, v tomto případě se jedná o diagram tříd systému DVD půjčovny. Jsou zde třídy *Zákazník*, *DVD* a *Režisér*, dále pak asociativní třída *Výpujčka*.

Každá třída má v horní části název, ve druhé části je seznam atributů a v dolní části je seznam operací této třídy. Povinný je pouze název, atributy a operace v tomto diagramu uvedeny být nemusí. Asociace mezi třídami je znázorněna plnou čarou, která může mít různé ozdoby. Na obrázku 4.5 žádné uvedené nejsou, ale kdybychom například chtěli znázornit, že zákazník může mít půjčený libovolná počet DVD, tak bychom u asociace na

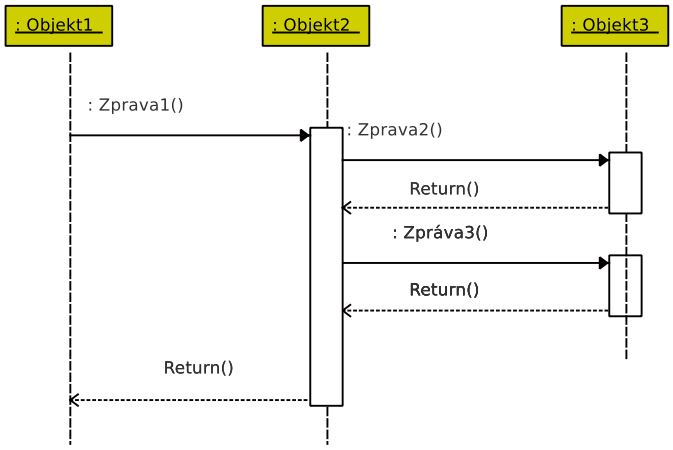
straně u DVD připsali znak '*'. V případě, že bychom chtěli znázornit, že DVD může být buď půjčeno, nebo na místě, pak bychom u asociace mezi zákazníkem a DVD na straně u zákazníka připsali '0..1'.
Asociativní třída *Výpůjčka* má za úkol udržovat informace o výpůjčce – jedná se o způsob rozšíření vazby mezi třídami.



Obrázek 4.5: Příklad diagramu tříd – DVD půjčovna

4.3.3 Sekvenční diagram

Sekvenční diagram patří do skupiny interakčních diagramů. Využití tohoto diagramu je v případě, když je potřeba znázornit časové souvislosti interakcí. Cílem tohoto diagramu je ukázat, kdy v čase který objekt s čím operuje. Příklad sekvenčního diagramu je na obrázku 4.6.



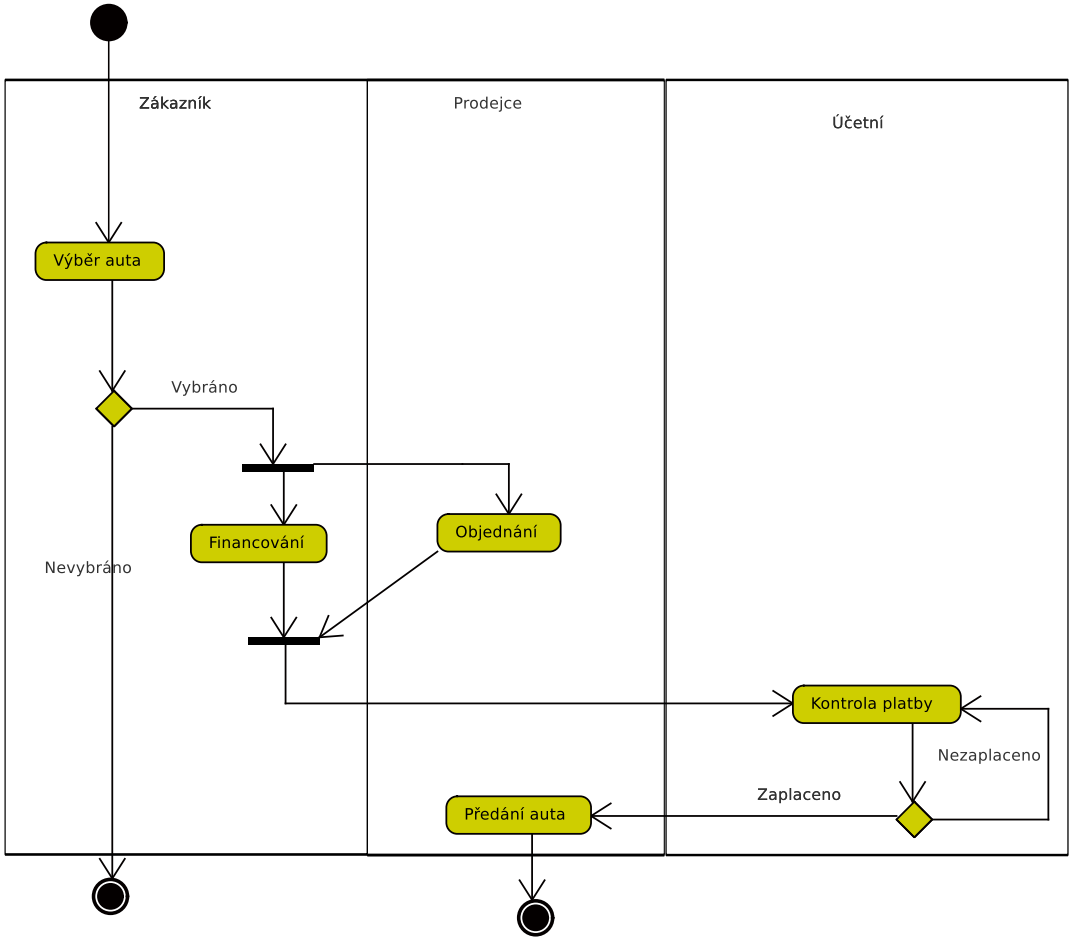
Obrázek 4.6: Příklad sekvenčního diagramu

Na obrázku 4.6 je příklad sekvenčního diagramu. Tento diagram popisuje, jak si objekty Objekt1, Objekt2 a Objekt3 posílají v čase zprávy, a kdy je který z těchto objektů aktivní.

Každý objekt je zobrazen podobně, jako v diagramu tříd (4.3.2), neuvádí se zde ovšem atributy a operace. Pod každým z těchto objektů je svislá čárkovaná čára. Tato se nazývá čára života objektu a znázorňuje, kdy objekt existuje. V tomto typu diagramu čas plyne vertikálně, od vrchu dolů. Na příkladu 4.6 objekty existovaly již před posláním první zprávy.
Aktivita objektu ukazuje, kdy je který objekt aktivní. Je znázorněna bílým obdélníkem zesilujícím v daném čase čáru života daného objektu. Zprávy se zobrazují plnými šipkami a mohou mít návratové hodnoty. Návratová hodnota může být v návratové zprávě. Zobrazuje se pak čárkovanou šipkou (v případě 4.6 má vždy název Return()).
Na obrázku 4.6 je tato sekvence:
Na začátku je aktivován *Objekt1* a ten posílá zprávu *Zprava1* objektu *Objekt2*. *Objekt2* přebírá aktivitu a posílá postupně *Zprava2* a *Zprava3* objektu *Objekt3*, který v obou případech vrací návratovou hodnotu objektu *Objekt2*. Až *Objekt2* dokončí práci, tak předá zaměření zpět objektu *Objekt1* a skončí platnost všech objektů. [8]
Časová osa sekvenčních diagramů nemá měřítko – od délky obdélníků nelze usuzovat o délce aktivit v čase.

4.3.4 Diagram aktivit

Diagram aktivit je diagram, který popisuje chování. Používá se pro modelování procedurální logiky, procesů a workflow. Umožňuje prostřednictvím tzv. drah (swimlines) kdo kterou činnost provádí a kdo má za co zodpovědnost. Příklad aktivitního diagramu je na obrázku 4.7.
Tento diagram znázorňuje proces nákupu auta. Vystupují zde aktéři Zákazník, Prodejce a Účetní. Z diagramu je patrné, že zákazník má na starosti vybrat auto a pak jej případně zaplatit. Prodejce má za úkol vybrané auto objednat a pokud zákazník zaplatil, tak jej předat. Účetní se stará o to, jestli je auto zákazníkem zaplaceno.
V diagramu aktivit se vyskytují elementy aktivit (například Předání automobilu), elementy rozdělení toku aktivit (v případě, kdy je třeba provádět více aktivit zároveň – např. financování a objednání auta). Dále jsou zde elementy větvení toku aktivit s podmínkou, například kontrola platby. Zde účetní kontroluje, jestli bylo auto zaplaceno. To provádí tak, že zkontroluje, jestli bylo zaplaceno. Pokud ano, tak tok aktivit pokračuje dál, pokud ne, tak znovu platbu kontroluje. Pak jsou zde symboly začátku a konce aktivit – jedná se o černé vyplněné kružnice.
Cílem diagramu aktivit je tedy znázornit časovou posloupnost aktivit a také zodpovědnosti aktérů za tyto aktivity. [9]



Obrázek 4.7: Příklad diagramu aktivit

II. Praktická část

Cílem této diplomové práce bylo vytvořit program, který bude simulovat pohyb částic v mnohočásticových soustavách. Tento program byl vytvořen. Jeho název je *Ptcsim*, jako zkratka anglických slov Particle Simulator.

5 Program Ptcsim

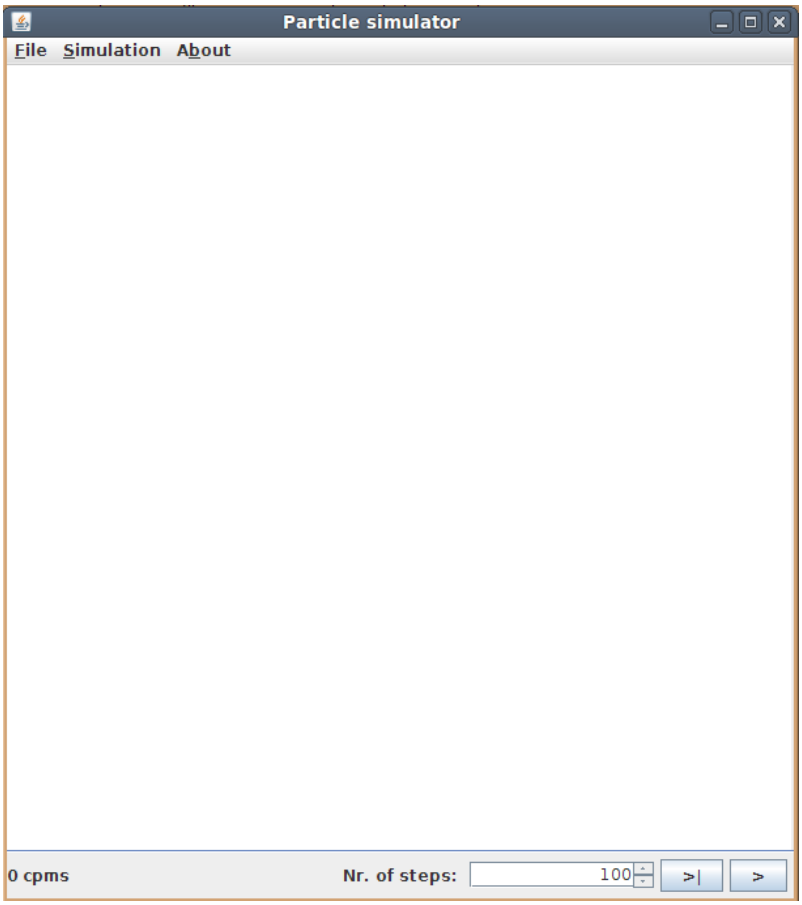
Program Ptcsim je aplikace vytvořená v jazyce Java. Vytvářena byla v prostředí GNOME v linuxové distribuci Ubuntu. Díky použití technologie jazyka Java, který zdrojový kód při kompilaci překládá na takzvaný mezikód (bytekód), je však možné tuto aplikaci spouštět na jiných operačních systémech, lépe řečeno na těch, pro které existuje Java Virtual Machine (JVM).
Zde v praktické části bude program Ptcsim popsán nejprve z hlediska uživatelského a potom z hlediska programátorského, tedy jak byl vytvořen. V přílohách jsou pak uvedeny vybrané zdrojové kódy v jazyce Java. Kompletní zdrojové kódy se nachází na přiloženém CD.

5.1 Program Ptcsim z uživatelského hlediska

Jak již bylo uvedeno, tak program Ptcsim je aplikace, která slouží k simulaci pohybu částic v mnohočásticových soustavách. Ovládání programu spočívá v tom, že částice ve výchozích polohách jsou uloženy v souboru, jehož struktuře se věnuje část 5.1.1.1. Tento soubor je načten programem Ptcsim. V tuto chvíli může uživatel spustit simulaci, pozastavovat, měnit některé konstanty simulace a částice v současných polohách a se současným nastavením ukládat zpět do souborů.
Program Ptcsim je okenní aplikace využívající knihovnu Java Swing (2.1.1.1). Po spuštění vypadá aplikace tak, jak je uvedeno na obrázku 5.1.

5.1.1 Otevírání a ukládání souborů částic

Program Particle Simulator pracuje se soubory, které musím být ve tvaru, který je uveden v části 5.1.1.1. Tyto soubory umí program Ptcsim otevírat, modifikovat a upravovat.
Otevření souboru lze provést kliknutím na nabídku *Open...* v menu *File*. Pro tuto operaci existuje i klávesová zkratka *Ctrl+O*. Následuje zobrazení dialogového okna pro výběr souboru. Je možné vybrat pouze soubory s příponou **.pts*. Pokud uživatel nějaký takovýto soubor vybere a zvolí jej k otevření, pak se obsah tohoto souboru načte do programu `Ptcsim` a zároveň se zobrazí částice, jak je vidět na obrázku 5.2.
V případě, že byl otevřen nějaký soubor s částicemi, stane se aktivní nabídka *Reopen* (klávesová zkratka *Ctrl+R*), která znovuotevře již otevřený soubor. Tedy jej v podstatě restartuje.
Pro uložení souboru slouží nabídka *Save as...* (klávesová zkratka *Ctrl+S*), která se také nachází v menu *File*. Po zvolení této nabídky se stejně jako v případě otevírání souboru



Obrázek 5.1: Okno programu Ptcsim po spuštění v linuxové distribuci Ubuntu v prostředí Gnome

zobrazí dialogové okno pro výběr souboru. Zde je možné buď vybrat existující soubor (pouze typu **.pts*), nebo zadat jméno nového. V případě zadávání nového souboru je k tomuto názvu přidána přípona *pts*. To však pouze v případě, že nebyla zadána přípona jiná.

Předchozí tři uvedené volby z menu *File* pracují se soubory typu **.pts*, který je popsán v části 5.1.1.1. Následující volba (*Export as CSV...*, klávesová zkratka *Ctrl+E*) vyexportuje částice do souboru typu **.CSV*, tedy souboru, který lze otevřít v tabulkovém procesoru. Jedná se o obyčejný textový soubor, kde jsou jednotlivé hodnoty odděleny jednotným znakem oddělovače (v případě Ptcsim je to znak čárky (","")) a tyto hodnoty jsou rozděleny na řádky. Příkaz *Export as CSV...* tedy otevře (stejně jako v případě ukládání) dialogové okno pro uložení souboru, jen s tím rozdílem, že je možné vybírat soubory typu **.CSV*. V případě, že uživatel vybere soubor, do kterého chce exportovat částice, pak budou hodnoty vlastností jednotlivých částic uloženy do tohoto souboru ve tvaru:

```
x,y,vx,vy,m,q,d,barva
```

kde každý řádek představuje hodnoty jedné částice. Tyto hodnoty jsou uloženy ve stejném pořadí, jak je uvedeno v části 5.1.1.1 mezi tagy `<ptc></ptc>`, jenom s tím rozdílem, že nejsou odděleny mezerami, ale čárkami.
Do souboru *CSV* nejsou ukládány hodnoty simulačních konstant, jak je tomu v případě souboru *pts*.

5.1.1.1 Struktura soubru *.pts

Program Ptcsim používá pro ukládání hodnot poloh, rychlostí a dalších vlastností jednotlivých souborů částic soubor, který je ve tvaru *xml*. 2.1.1.3 Na následujícím příkladu bude prezentováno, jaká musím být struktura souboru, aby mohl být programem Ptcsim správně načten.
Příklad souboru `pts`:

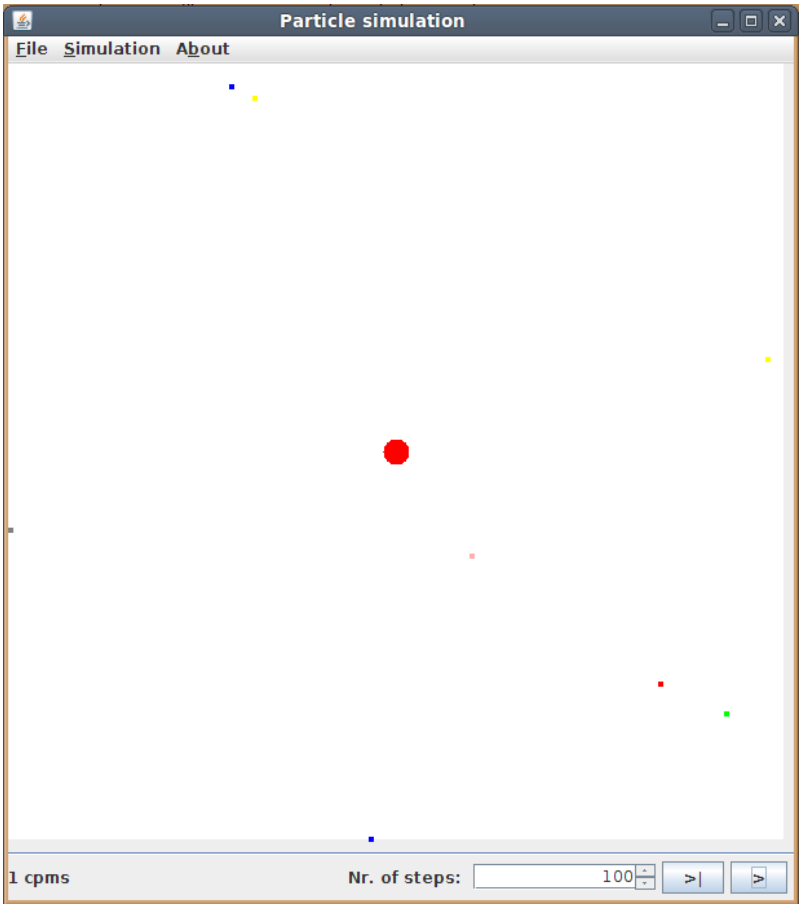
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ptcs>
3      <dt>100.0</dt>
4      <scale_x>1.0E12</scale_x>
5      <scale_y>1.0E12</scale_y>
6      <bound>true</bound>
7      <ptc>5.006694E11 5.0010895E11 0.328103 0.79910869 2.0E30 0.0 20.0 0</ptc>
8      <ptc>7.9711E11 8.9379E11 4622.667 14088.22847145041 1.0E25 0.0 5.0 1</ptc>
9      <ptc>2.8686E11 2.84922E10 11410.1605 12444.55461007 1.0E25 0.0 5.0 2</ptc>
10     <ptc>9.78379E11 3.793E11 13429.2639 28031.8633714 1.0E25 0.0 5.0 3</ptc>
11     <ptc>2.9619E9 5.99996E11 16343.3611075 34379.0261 1.0E25 0.0 5.0 4</ptc>
12     <ptc>9.25208E11 8.375311E11 -13161.7727 6062.25416 1.0E25 0.0 5.0 5</ptc>
13     <ptc>5.975377E11 6.34793E11 -6307.63465 -5883.88819 1.0E25 0.0 5.0 6</ptc>
14     <ptc>8.401083E11 7.986E11 4653.607776496 -15481.178 1.0E25 0.0 5.0 0</ptc>
15     <ptc>7.725732E11 8.8906E11 3488.11965 -15015.8178 1.0E25 0.0 5.0 1</ptc>
16     <ptc>4.667008E11 9.994935E11 -5012.9388 12358.3377 1.0E25 0.0 5.0 2</ptc>
17     <ptc>3.1752E11 4.40498E10 16771.8263 6059.853893341 1.0E25 0.0 5.0 3</ptc>
18 </ptcs>
```

Ze zde uvedené ukázky souboru `pts` je patrné, že se jedná o validní *xml* soubor (2.1.1.3). Jednak obsahuje *xml* hlavičku (první řádek) a pak taky kořenový *xml* tag – v tomto případě se jedná o dvojici `<ptcs></ptcs>`.
Začátek tohoto souboru obsahuje hodnoty konstant důležitých pro simulaci. První je hodnota *dt*, což je délka časového kroku simulace v sekundách. Další položky jsou *scale_x* a

scale_y. Jedná se o defionování velkosti prostoru simulace. V tomto případě je zde soubor, který si klade za cíl přibližně definovat planety sluneční soustavy, velikost prostoru tedy přibližně odpovídá prostoru, v jakém se pohybují planety sluneční soustavy. Jako poslední z konstant je zde pravdivostní hodnota, která určuje, zda se částice mají odrážet od hranic definovaného prostoru či ne. Pokud je hodnota mezi tagy `<bound>` a `</bound>` `true`, pak se částice budou odrážet od hranic prostoru, pokud `false`, tak se nebudou odrážet.
Dále zde mezi tagy `<ptc>` a `</ptc>` následuje definice jednotlivých částic. Každá částice (může jich být libovolný počet) je definovaná sedmi hodnotami s desetinou čárkou a jednou celočíselnou. První hodnotou je poloha v ose x, druhá je poloha v ose y. Následují hodnoty rychlosti, nejprve hodnota x-ová a pak hodnota rychlosti ve směru osy y. Hodnoty poloh jsou ve stejných jednotkách, jako jsou hodnoty velikosti prostoru definovaného mezi tagy *scale_x* a *scale_y*. Rychlosti jsou v těchto jednotkách za jednotku času $[s]$.
Následuje hodnota, která definuje hmotnost částice v kg. Další hodnota je náboj částice v jednotkách Coulomb a další je hodnota průměru částice v *px*, hodnota v podstatě jen z důvodu zobrazení. Poslední hodnota je celočíselná a může nabývat jen hodnoty od 0 do 9. Určuje barvu, kterou bude částice zobrazena.
Všechny tyto hodnoty mezi tagy `<ptc>` a `</ptc>` jsou vypsány za sebou a musí být odděleny mezerou, ale pouze jednou. Není přípustný žádný jiný znak. Kdyby tomu tak nebylo, tak program Ptcsim tyto částice nenačte.

5.1.2 Ovládání simulace

V okamžiku, kdy je otevřen soubor s částicemi, je možné začít se simulací těchto částic. Jsou zde dvě možnosti spuštění. Jednou je spuštění bez kontroly počtu simulovaných kroků a druhou je simulovat určitý počet kroků.
Obě tyto volby se nacházejí v dolní části okna programu(5.1). Spuštění bez kontroly počtu simulovaných kroků je možné tlačítkem „>“. V tu chvíli se začne v levém dolním rohu okna zobrazovat počet výpočtů za jednu milisekundu (zkratka cpms - computes per milisecond).Výpočtem se rozumí přepočet poloh všech částic s požadovaným krokem *dt*. V případě, že je potřeba provést jen určitý počet simulačních kroků, tak je třeba tento počet zadat do textového pole s popiskem „Nr. of steps“ (výchozí hodnota je 100 kroků) a pak spustit simulaci stiskem tlačítka „> |“.
Pokud byla simulace spuštěna tlačítkem „>“, pak se toto tlačítko změní na „||“ a lze tímto simulaci pozastavit. V případě spuštění simulace tlačítkem „> |“ se obě ovládací tlačítka stanou neaktivními a nelze na ně klikat. Aktivními se stanou až poté, co program *Ptcsim* odsimuluje požadovaný počet kroků.
Průběh simulace lze v průběhu sledovat v hlavní části okna programu. Zde se jednotlivé



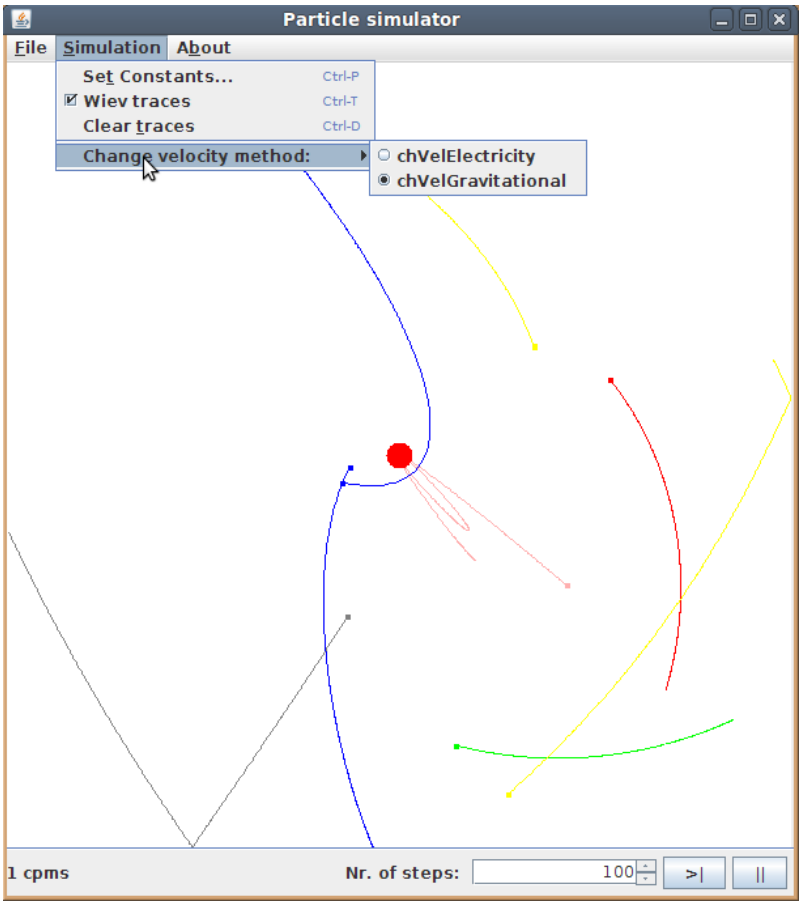
Obrázek 5.2: Načtený pts soubor v programu Ptcsim

částice zobrazují jako vyplněné kružnice, a to vždy v té barvě, jakou byly definovány v souboru, ze kterého jsou načteny. Dále se zde příslušnou barvou zobrazují dráhy jednotlivých částic, jak je vidět na obrázku 5.3.

Další volby simulace a zobrazování částic lze nastavit v menu *Simulation*, které je zobrazeno na obrázku 5.3. První položka (*Set Constants*, Ctrl+P) umožňuje nastavit simulační konstanty. Po zvolení této položky se zobrazí okno (obr. 5.4).

V tomto okně lze nastavit velikost prostoru, ve kterém simulace probíhá. Jednak velikost prostoru skutečného (Real space size), která je v metrech. Dále pak velikost simulačního prostoru (Simulation space size), který je v pixelech. Ve druhém případě se jedná o velikost zobrazovaného prostoru, ve kterém se pohybují částice na obrazovce.

Pod oddělovačem se nachází volba pro změnu časové konstany *dt*. Tato konstanta určuje, jak dlouhý čas se bude uvažovat mezi jednotlivými simulačními kroky. Jednotkou této konstanty je sekunda. Dále je zde zaškrtávací políčko „Rebounding of walls“, jehož změnou lze nastavit, jestli se částice budou odrážet od okrajů prostoru (simulačního i skutečného), nebo jestli se budou pohybovat nezávisle na velikosti tohoto prostoru. Všechny tyto hod-

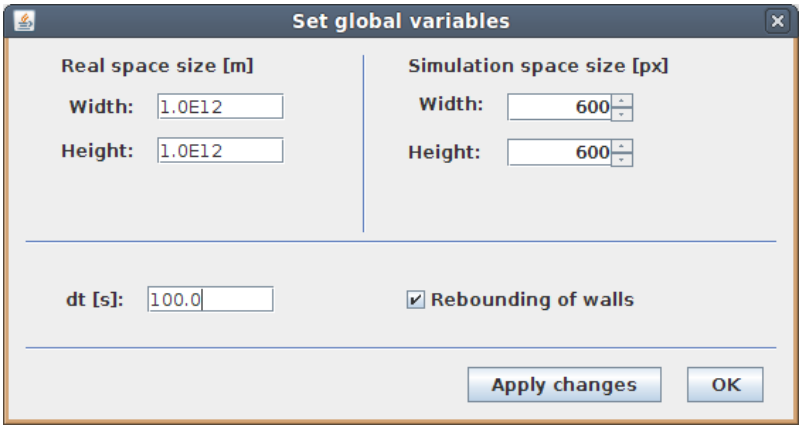


Obrázek 5.3: Program Ptcsim s probíhající simulací a zobrazeným menu *Simulation*

noty, kromě velikosti simulačního prostoru, jsou načítány ze souboru *pts*.

V menu *Simulation* jsou dále položky, které ovlivňují zobrazování simulovaných částic. Jedná se o položku *View traces*, která zajišťuje zobrazování, respektive skrývání stop částic. Tato položka dráhy pouze zobrazuje či skrývá. Na rozdíl od toho položka *Clear traces* tyto stopy smaže a začínají se kreslit znova od aktuálních poloh částic. Tyto dvě volby mají vliv na přehlednost simulace.

Poslední položkou v menu *Simulation* je *Change velocity method*. Tato obsahuje podmenu, které ve výchozím stavu aplikace obsahuje dvě položky: *chVelElectricity* a *chVelGravitational*. Z těchto metod může být vybraná pouze jedna, na obrázku 5.3 je to metoda *chVelGravitational*. Jedná se o výběr metody, která bude použita pro přepočet rychlostí jednotlivých částic. V prvním případě se uvažují elektrické náboje částic, vhodná je například na simulování pohybu molekul plynu. Druhá metoda uvažuje gravitační síly mezi částicemi, takže je možné jí použít pro simulaci pohybu planet ve Sluneční soustavě. Toto podmenu je dynamicky (při překladu programu na byte kód) generováno podle toho, jaké metody



Obrázek 5.4: Okno pro nastavení konstant simulace

jsou k dispozici. Pokročilý uživatel má možnost doprogramovat libovolné další metody a program *Ptcsim* lze použít i na simulaci jiných interakcí. Další informace o tom, jak lze dodat do programu další metody, jsou v části 5.2.5.

5.2 Program Ptcsim z programátorského hlediska

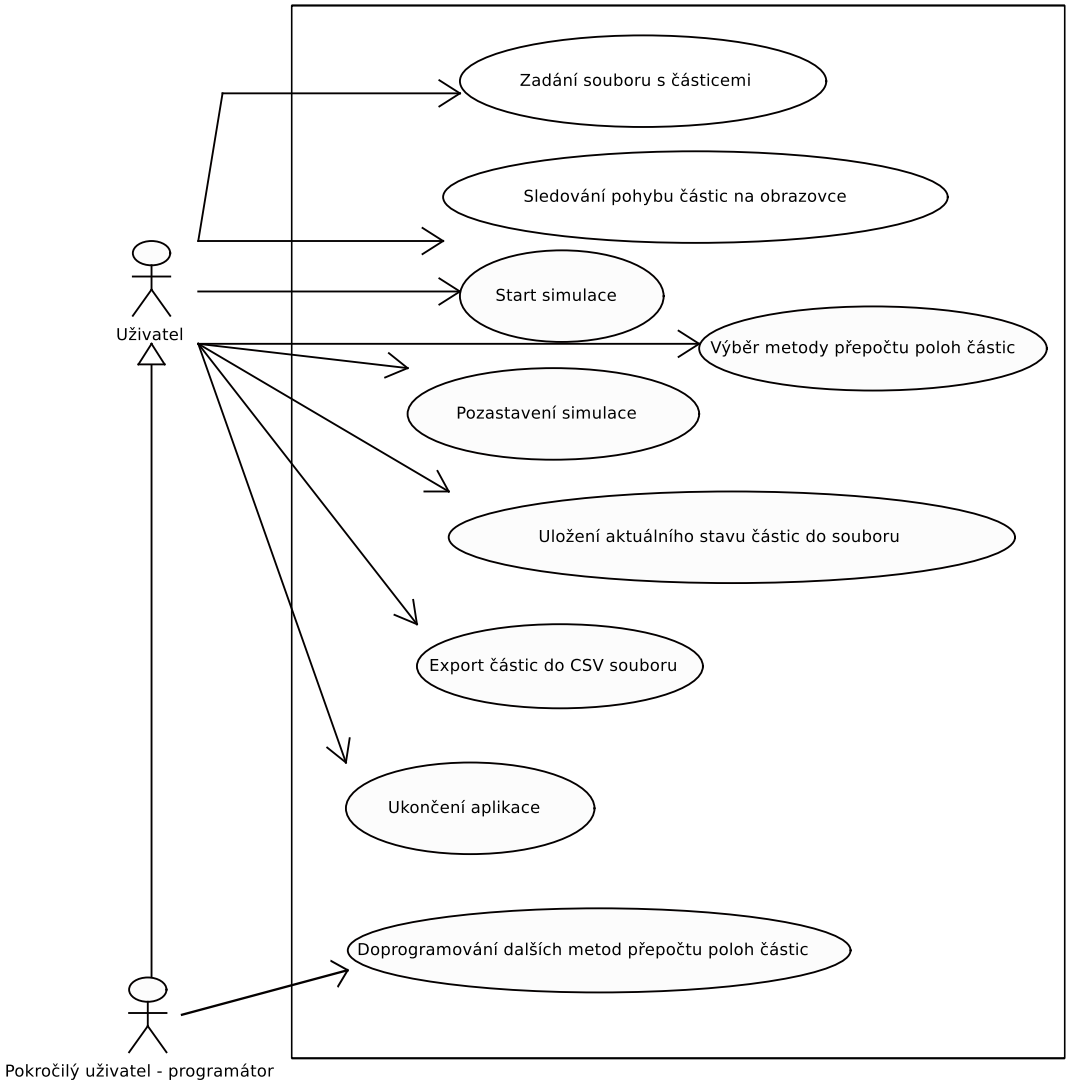
Program Ptcsim je napsán v jazyce Java. Zdrojové kódy jsou umístěny ve třinácti zdrojových souborech. Tyto soubory jsou rozděleny do dvou balíčků (package) podle typu úloh, které jednotlivé třídy (soubory) řeší. Jedná se o balíček `psim`, ve kterém jsou soubory, které souvisí se samotnou simulací. Druhým balíčkem je `GUI`, ve kterém jsou třídy starající se o vykreslování a interakci s uživatelem. Diagram případů užití programu Ptcsim je uveden na obrázku 5.5.

5.2.1 Hlavní třída – Main.java

Hlavní třídou programu Ptcsim je třída Main.java, která se nachází v balíčku `psim`. Tato je vytvořena po spuštění programu a je zavolána její metoda `Main`. Zdrojový kód této třídy je v příloze P I.

V deklarační části této třídy jsou deklarované „globální proměnné“. Takové, ke kterým je možné přistupovat i z ostatních tříd, tedy statické. Tyto proměnné je možné používat, aniž by bylo potřeba instanciovat třídu, ve které jsou definovány.

Metoda `Main`, která je volána po spuštění program *Ptcsim*, nejprve prostřednictvím instance třídy `Runtime` zjistí, kolik je k dispozici procesorů. Pokud je k dispozici jeden více jádrový, pak je výsledkem počet těchto jader. Tento počet je důležitý pro následnou paralelizaci výpočtů (5.2.4.1). Podle tohoto počtu se následně instanciuje třída `PtcsSingle`

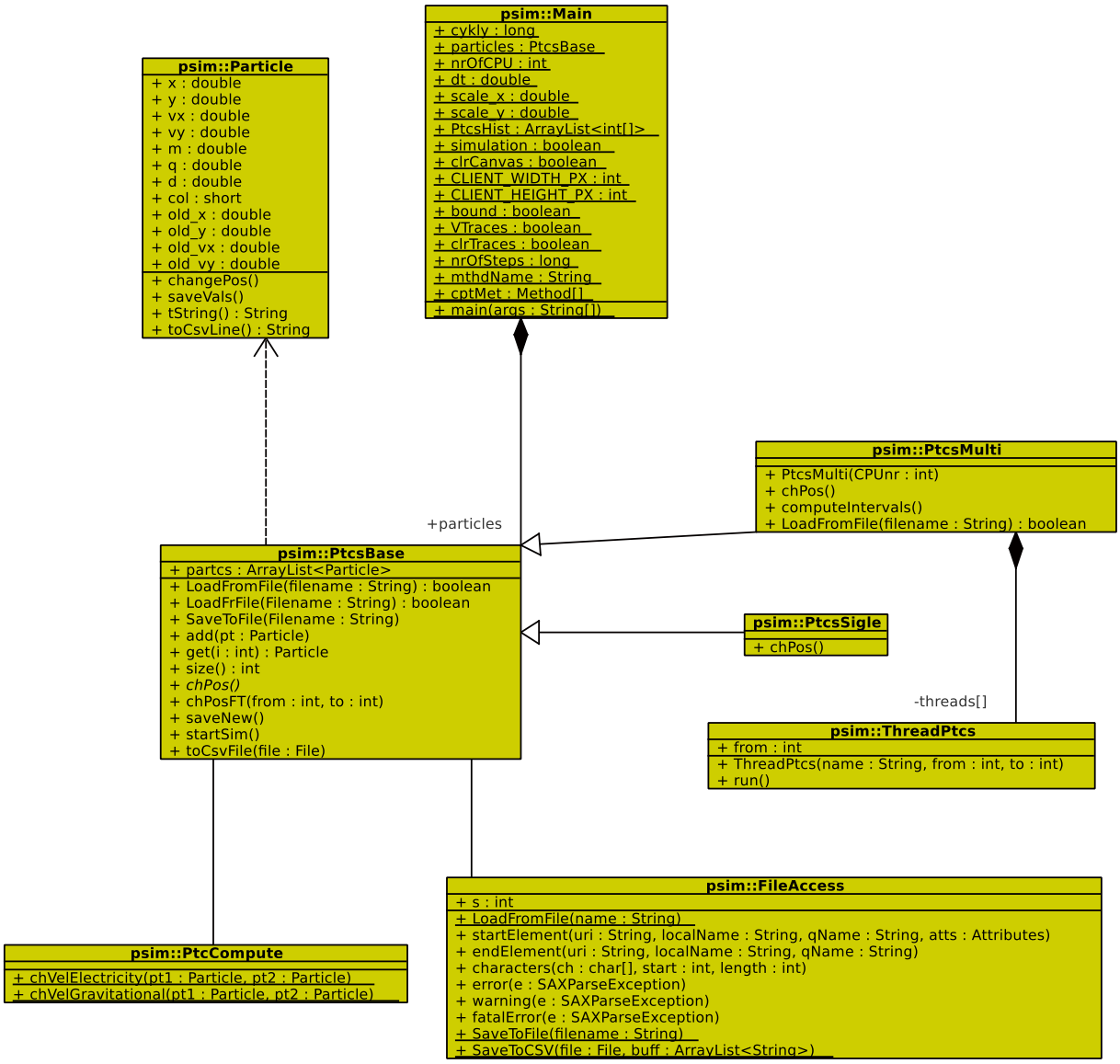


Obrázek 5.5: UML diagram případů užití programu Ptcsim

nebo `PtcsMulti`. Následuje vytvoření okna programu. To je prováděno voláním metody `Gui.CreateAndShowGUI()`. Jako poslední provádí metoda `Main` spuštění simulace. To ve skutečnosti probíhá již od spuštění aplikace, ne až po spuštění uživatelem. V okamžiku, kdy uživatel spustí simulaci (5.1.2), se začnou uvažovat částice. Po spuštění tedy program Ptcsim běží v nekonečné smyčce, kterou lze přerušit jen zavřením hlavního okna programu. Diagram tříd (4.3.2) balíčku `psim` je na obrázku 5.6.

5.2.2 Particle.java – třída reprezentující jednu částici

Třída `Particle.java` obsahuje členské proměnné, které udržují informace o vlastnostech dané částice. Jedná se o x-ovou a y-ovou polohu, rychlost v obou osách, hmotnost, náboj, průměr a barvu. Dalšími členskými proměnnými jsou `old_x`, `old_y`, `old_vx` a `old_vy`. Tyto



Obrázek 5.6: Diagram tříd balíčku `psim`

vlastnosti udržují hodnoty poloh a rychlostí z předchozího kroku. To z důvodu výpočtů nových rychlostí, protože při přepočtech dochází k modifikacím aktuálních vlastností a je potřeba vycházet z původních hodnot.

Konstruktor této třídy přebírá jako parametry všechny tyto vlastnosti (kromě těch s prefixem `old`) a tyto parametry ukládá do vlastností uvedených v předchozím odstavci. Pak volá metodu této třídy `SaveVals()`, která uloží hodnoty z proměnných s aktuálními hodnotami do proměnných s hodnotami předchozími. Při tomto volání v konstruktoru se jedná pouze o inicializaci proměnných se starými hodnotami.

Metoda `changePos()` je volána při každém přepočtu poloh částic. Jejím úkolem je přičíst k aktuální poloze hodnotu rychlosti v každé ose zvlášť. Vzhledem k možnosti nastavení

velikosti časového kroku *dt* je však nutné vynásobit hodnotu přičítané rychlosti velikostí tohoto kroku. Dále pak tato metoda zajišťuje odraz od stěn simulačního prostoru. Pokud je tedy vybrána možnost odrážení od stěn (5.1.2 – jestli se mají částice odrážet od stěn je uloženo ve statické proměnné `Main.bound`), tak tato metoda kontroluje, jestli není částice mimo prostor (od 0 do `scale_x`, respektive `scale_y`). Pokud ano, tak inveruje (vynásobí hodnotou -1) hodnotu patřičné rychlosti.
Medoda `SaveVals()`, která byla zmíněna již v odstavci o konstruktoru této třídy je volána po každém přepočtu poloh a rychlostí částice. Ukládá jen hodnoty poloh a rychlostí do proměnných s prefixem `old`.
Poslední dvě metody v této třídě mají tu funkci, že převádějí hodnoty vlastností dané instance na speciální řetězce pro uložení do souborů. V prvním případě (metoda `tString()`) se jedná o tvar pro uložení do souboru `pts` a druhá (`toCsvLine()`) exportuje tyto hodnoty jako řádek souboru `CSV`. Obě tyto metody instanciují třídu `StringBuffer`, do které přidávají hodnoty vlastností částice převedené na řetězec oddělení v případě souboru `pts` znakem mezery a `CSV` znakem čárky. Návratovou hodnotou je pak instance `StringBuffer` převedená na řetězec.
Zdrojový kód třídy `Particle.java` je uveden v příloze P II.

5.2.3 Třída PtcsBase.java

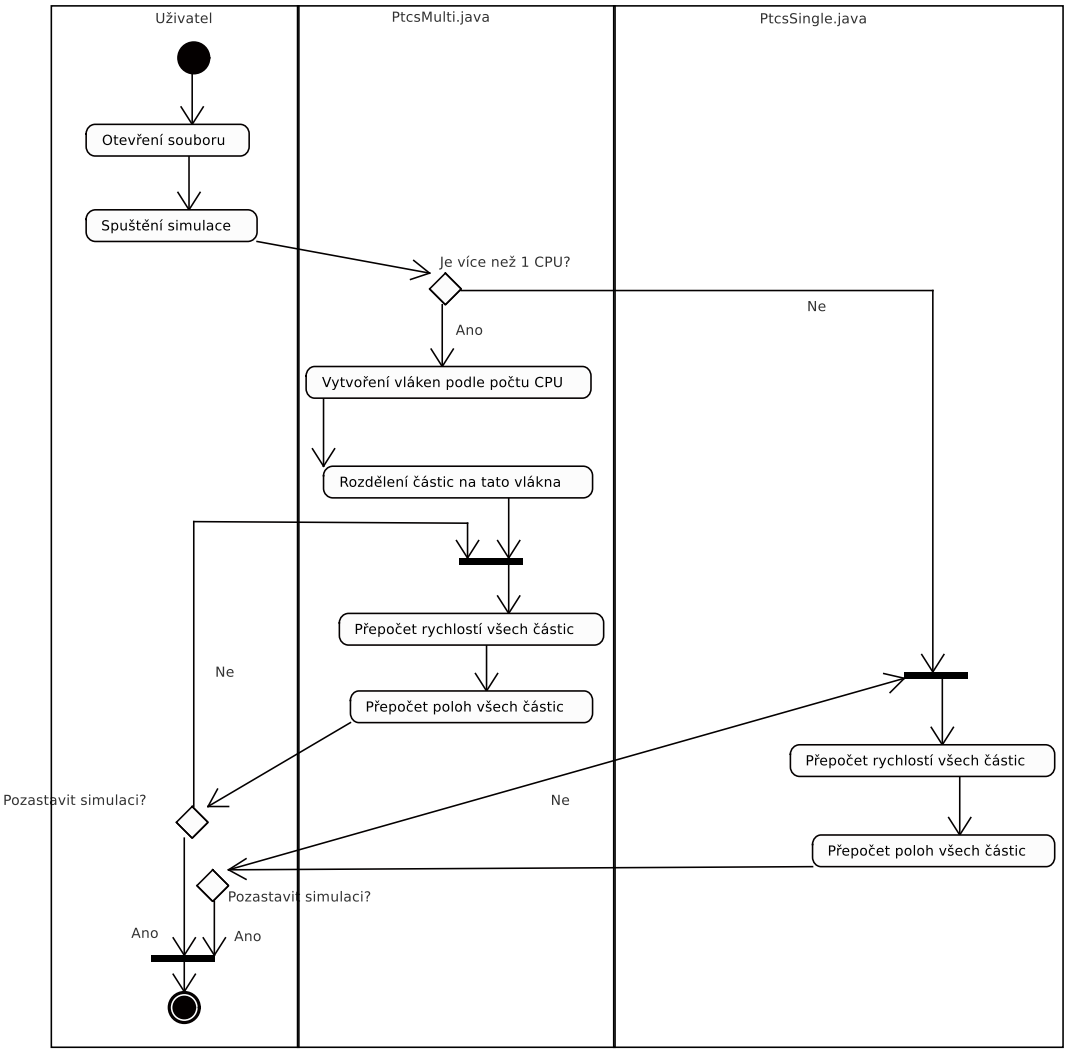
Třída `PtcsBase` je třída, která reprezentuje pole částic a operace s nimi. Jejím „jádrem“ je pole částic (`ArrayList` instancí třídy `Particle.java` (5.2.2)). To je vlastností s názvem `partcs`.
Třída PtcsBase je třídou základní, ze které dědí třídy `PtcsSingle` a `PtcsMulti`. K tomuto dělení dochází z toho důvodu, že výpočty, u kterých je potřeba, aby probíhaly co nejrychleji, se liší podle toho, jestli je k dispozici jeden či více procesorů. Kdyby to měla zajišťovat pouze jedna třída, tak by bylo nutné v každém cyklu výpočet dělit na dvě větve pomocí příkazu `if`, což by znamenalo časovou ztrátu a snížení počtu výpočtů za sekundu. Proto dochází v metodě `main` ve třídě `Main.java` k rozhodování, jestli se bude instanciovat třída `PtcsMulti` nebo `PtcsSingle` (5.2.1). Pro toto následné dědění je zde abstraktní metoda `chPos()`, jejíž implementace se liší podle počtu procesorů.
Třída `PtcsBase` obsahuje metody, které umožňují načíst pole částic ze souboru `pts` nebo jej do něj uložit (`LoadToFile()` a `SaveToFile()`). Dále metodu pro uložení do `CSV` soboru (`toCsvFile()`).
Tři metody z této třídy slouží jen pro zapouzdření metod `ArrayListu` `partcs`. Jedná se o metodu `add()`, která připojí na konec pole instanci třídy `Particle`, metodu `size()`, která vrací celočíselnou hodnotu počtu částic v poli a metodu `get(int i)`, která vrací

instanci třídy `Particle` na pozici `i` v poli `partcs`.
Metoda `chPosFT(int from, int to)` má tu funkci, že volá metodu přepočtu poloh částic (5.2.5) pro interval částic od hodnoty `from` do `to`. Jedná se o dva vnořené cykly `for`, které zajišťují, že jsou z pole `partcs` vybírány částice tak, aby interagovala každá s každou, ale právě jednou. Vevnitř těchto vnořených cyklů jsou k dispozici indexy částic, které spolu mají interagovat. Ve statické proměnné `Main.mthdName` je uložen řetězec názvu metody, která je pak nad každou dvojicí částic volána. Takto jsou vypočítány interakce všech částic se všemi, a to podle metody, kterou vybral uživatel v nabídce `Simulation`.
Metoda `startSim()` běží v nekonečné smyčce. Pokud je statická proměnná `Main.simulation` v hodnotě `true`, tak tato metoda volá metodu `chPos`. Pokud uživatel spustil simulaci pro určitý počet kroků (5.1.2), pak se tato metoda stará o to, aby byla metoda `chPos` zavolána právě tolikrát, kolik kroků chtěl uživatel odsimulovat. Po tuto dobu také nastavuje vlastnost `Enabled` u tlačítek spouštění simulace (5.1.2) na hodnotu `false`.
Metoda `saveNew()` ukládá přepočítané hodnoty poloh a rychlostí částic do vlastností s prefixem `old`. Provádí to tak, že nad všemi částicemi v poli `partcs` volá metodu `saveVals` (5.2.2). Zdrojový kód třídy `PtcsBase.java` je v příloze P III.

5.2.4 Třídy PtcsSingle a PtcsMulti

Tyto třídy dědí třídu `PtcsBase.java` (5.2.3). Třída `PtcsSingle.java` je instanciována v případě, že je program `Ptcsim` spuštěn na počítači, který disponuje právě jedním procesorem. Jedinou metodou, která je v této třídě implementována je metoda `chPos()`. Tato metoda zavolá metodu `chPosFT(0, this.partcs.size())` definovanou ve třídě `PtcsBase`, čímž se přepočítají hodnoty poloh a rychlostí všech částic. Pak volá metodu `SaveNew()`, která uloží nově vypočtené hodnoty do starých. To je vpodstatě vše, co tato třída implementuje. Z toho důvodu není ani uveden zdrojový kód v příloze. Nachází se pouze na přiloženém CD.
Narozdíl od toho třída `PtcsMulti.java` se nachází v příloze P IV. V tomto souboru je kromě třídy `PtcsMulti` definována ještě třída `ThreadPtcs`, která dědí od třídy `Thread` (2.1.1.2). Jedná se o třídu, která reprezentuje jedno vlákno, tedy výpočet, který bude prováděn na jednom procesoru. Konstruktor této třídy přebírá jako parametry jednak jméno daného vlákna, a také interval částic, který bude následně zpracovávat – jedná se o dvě celočíselné hodnoty. Metoda `run()` této třídy pak volá metodu `chPosFT(from, to)`, která se nachází ve třídě `PtcsBase` (5.2.3).
V případě, že je k dipozici více než jeden procesor, tak se pro načtení částic ze souboru `pts` volá přetížená abstraktní metoda této třídy `LoadFromFile(String filename)`. Tato metoda jednak načte částice do pole `partcs` (5.2.3), ale také volá metodu `computeInter-`

`vals()` této třídy. Metoda `computeIntervals()` dělí částice na intervaly podle počtu procesorů (postup výpočtu je uveden v části 5.2.4.1).
Metoda `chPos` třídy `PtcsMulti` vychází z intervalů vypočítaných metodou `computeIntervals()` uložených v poli `intervals`. Podle počtu procesorů instanciuje tolik vláken (instancí třídy `ThreadPtcs`, kolik je procesorů. Pak všechna tato vlákna spustí (zavolá jejich metodu `Run` voláním metody `Start()` (2.1.1.2). Následně v cyklu čeká na dokončení výpočtu všech vláken, což je realizováno voláním metody `join()` (2.1.1.2). Následuje uložení nových hodnot do starých a zvýšení čítače počtu cyklů pro účely výpisu hodnoty počtu výpočtů za milisekundu.
Diagram aktivit (4.3.4) přepočtu poloh částic a rychlostí je na obrázku 5.7.



Obrázek 5.7: UML diagram aktivit přepočtu poloh a rychlostí částic v závislosti na počtu procesorů

5.2.4.1 Paralelizace výpočtu

Vzhledem k tom, že je možné, že bude změna pozic částic počítána na vícejádrovém procesoru, tak je třeba pole částic rozdělit na části tak, aby každá z částí počítala přibližně stejný počet kroků. Pokud by byl interval rozdělen lineárně, tedy na stejné díly, tak by první část počítala nejvíc, a další pak méně a méně. To z toho důvodu, že každá částice interaguje pouze s částicemi, které mají vyšší index. Mějme tedy pole částic, kde počet částic je n, i je index právě počítáné částice, c je počet částí na který je potřeba částice rozdělit (tedy počet jader procesoru) a j je část, která je aktuálně počítaná.
Pokud by byl interval rozdělen na části lineárně, tak by například v případě, že by bylo 10 částic a 2 procesory, první interval by byl od 0 do 4 a druhý od 5 do 9 (tak jako v programu *ptcsim* se zde uvažuje s indexováním od 0). Pak by se pro částici s indexem 0 počítaly interakce s částicemi 1 až 9, pro částici s indexem 1 by se počítaly interakce s částicemi 2 až 9. Pro první část celého intervalu by to tedy bylo $8 + 7 + 6 + 5 = 26$ operací. Pro druhu polovinu intervalu by to však bylo jen $4 + 3 + 2 + 1 = 10$ operací.
Z tohoto důvodu bylo třeba najít vzorec, podle kterého by bylo možné rozdělit pole částic tak, aby všechny procesory, které jsou k dispozici, byly zatíženy přibližně stejně.
Výpočet rozdělení celého intervalu na části vychází ze vzorce, který určuje, kolik operací se provede s částicí na i té pozici:

$$(n - \frac{i+j}{2})(j-i) \tag{1}$$

Po dosazení vzorce 1 do rovnice v tomto tvaru:

$$(n - \frac{x+1}{2})(x-1) = (n - \frac{x+n}{2})(n-x) \tag{2}$$

získáme po vyřešení dva kořeny x, z nichž jeden je mimo interval a druhý je ve tvaru:

$$-\frac{\sqrt{(-1+n)^2}}{\sqrt{2}} + n \tag{3}$$

Po dosazení hodnoty n do rovnice 3 by bylo výsledkem číslo, podle nějž lze rozdělit interval na dvě části. Univerzální vzorec pro rozdělení na c částí je pak ve tvaru:

$$d = n\left(1 - \sqrt{\frac{c-i}{c}}\right) \tag{4}$$

kde d je číslo, které dělí interval částic na části, n je počet částic, c je počet částí, na které je třeba interval rozdělit (je rovno počtu procesorů k dispozici) a i je pořadové číslo intervalu, pro který se počítá dělící hodnota.

5.2.5 Třídy PtcCompute.java a FileAccess.java

Třída `PtcCompute.java` (příloha P V) je třída, která obsahuje pouze statické metody. Jedná se o metody, do kterých jako parametry vstupují dvě částice a tyto metody přepočítávají jejich nové rychlosti. V okamžiku psaní této diplomové práce jsou k dispozici dvě metody pro přepočet rychlostí jednotlivých částic. První je `chVelElectricity (Particle pt1, Particle pt2)`, která přepočítává rychlosti obou zúčastněných částic s ohledem na jejich elektrické vlastnosti, tedy hlavně náboj částice. Druhá metoda – `chVelGravitational (Particle pt1, Particle pt2)` – také přepočítává velikosti a směry rychlostí, jenom s tím rozdílem, že bere v úvahu hmotnosti částic a uvažuje gravitační síly mezi částicemi.

Jak bylo uvedeno v části 5.1.2, tak je možné tuto třídu rozšířit o další metody. Jedinou podmínkou je, že musí tato nová metoda mít jedinečný název (jiný, než dvě předchozí) a musí mít jako parametry dvě částice s názvy `pt1` a `pt2`. Po přeložení programu do byte kódu a jejím spuštění se pak tato metoda zobrazí v nabídce *Simulation* v podmenu *Change velocity method:*. Pak je možné ji vybrat a pohyb částic bude řízen touto metodou.

Třída `FileAccess.java` sdružuje statické metody pro přístup k souborům. Její zdrojový kód je uveden v příloze P VI.

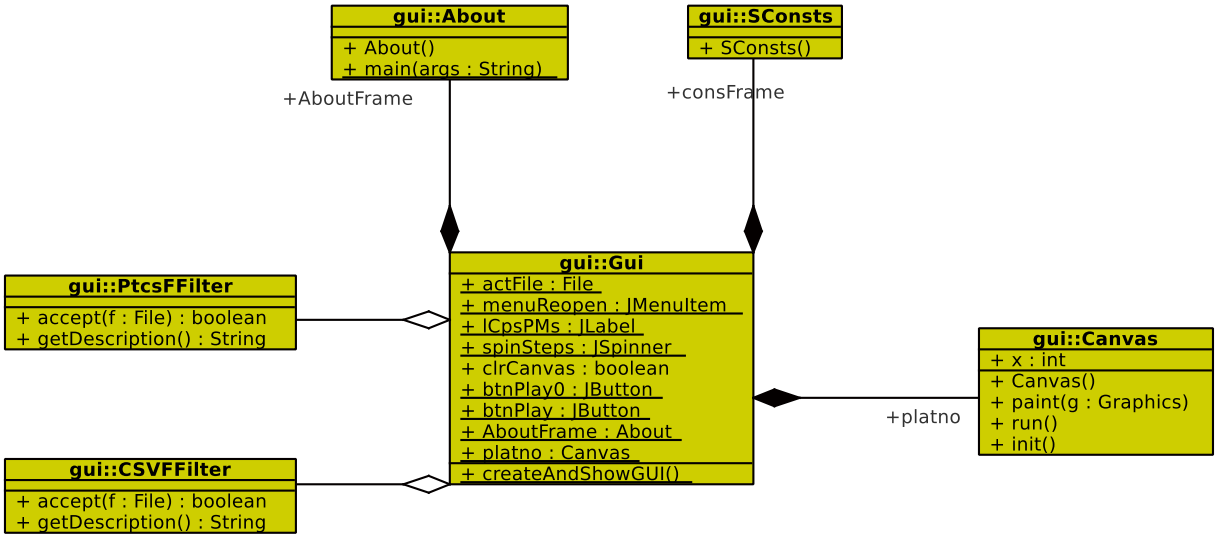
Pro export částic do souboru typu CSV slouží metoda `SaveToCSV(File file, ArrayList <String> buff)`, která přebírá jako parametr instanci třídy `File`, která reprezentuje soubor, do kterého se má ukládat, a pole řádků, které se do tohoto souboru mají uložit. Tato metoda nejprve ošetří všechny výjimky, ke kterým by mohlo dojít, kdyby soubor nebyl platný. Pak tento soubor otevře a v cyklu uloží všechny řádky. Pak tento soubor uzavře.

Metoda `LoadFromFile(String name)` přebírá jako parametr cestu k souboru typu `pts`, ze kterého má načíst částice a hodnoty globálních konstant. Tato metoda instanciuje SAX XML parser a tímto pak požadovaný soubor načte (2.1.1.3). O samotné parsování XML souboru se starají metody zejména `startElement` a `endElement`. První z nich podle počátečního elementu nastaví příznak (proměnná `s`) na příslušnou celočíselnou hodnotu. Metoda `characters`, která je volána v okamžiku, kdy parser načítá znaky mezi elementy, tyto znaky ukládá do zásobníku `buff`. Metoda `endElemnt` podle příznaku, jak jej nastavila metoda `startElement`, převede znaky ze zásobníku do příslušného tvaru (datového typu) a uloží je do příslušné proměnné.

Metoda `SaveToFile(String filename)` ukládá částice a globální konstanty do XML souboru s využitím SAX serializeru. Nejprve instanciuje výstupní proud, pak serializer a nad `ContentHandler`em tohoto serializeru pak volá metody pro začátek elementu, konec elementu a znaky mezi elementy. Postupně takto ukládá do výstupního proudu globální konstanty a pak v cyklu vlastnosti všech částic.

5.2.6 Balíček gui

V balíčku `gui` se nachází třídy, které zprostředkovávají uživatelské rozhraní programu `Ptcsim`. Třídy `Gui.java`, `About.java` a `SConst.java` reprezentují jednotlivá okna programu. Třídy `CSVFFilter.java` a `PtcsFFilter.java` dědí ze třídy `FileFilter` a tuto třídu rozšiřují v prvním případě o filtr souborů typu `CSV` a ve druhém případě o filtr souborů typu `Pts`. Tyto třídy nacházejí využití v případě, když uživatel otevře dialogové okno pro otevření či uložení souboru. Třída `Canvas.java` dědí ze třídy `JPanel` a rozšiřuje její možnosti o překreslování částic. Diagram tříd (4.3.2) tohoto balíčku se nachází na obrázku 5.8.



Obrázek 5.8: Diagram tříd balíčku `gui`

5.2.6.1 Třídy oken

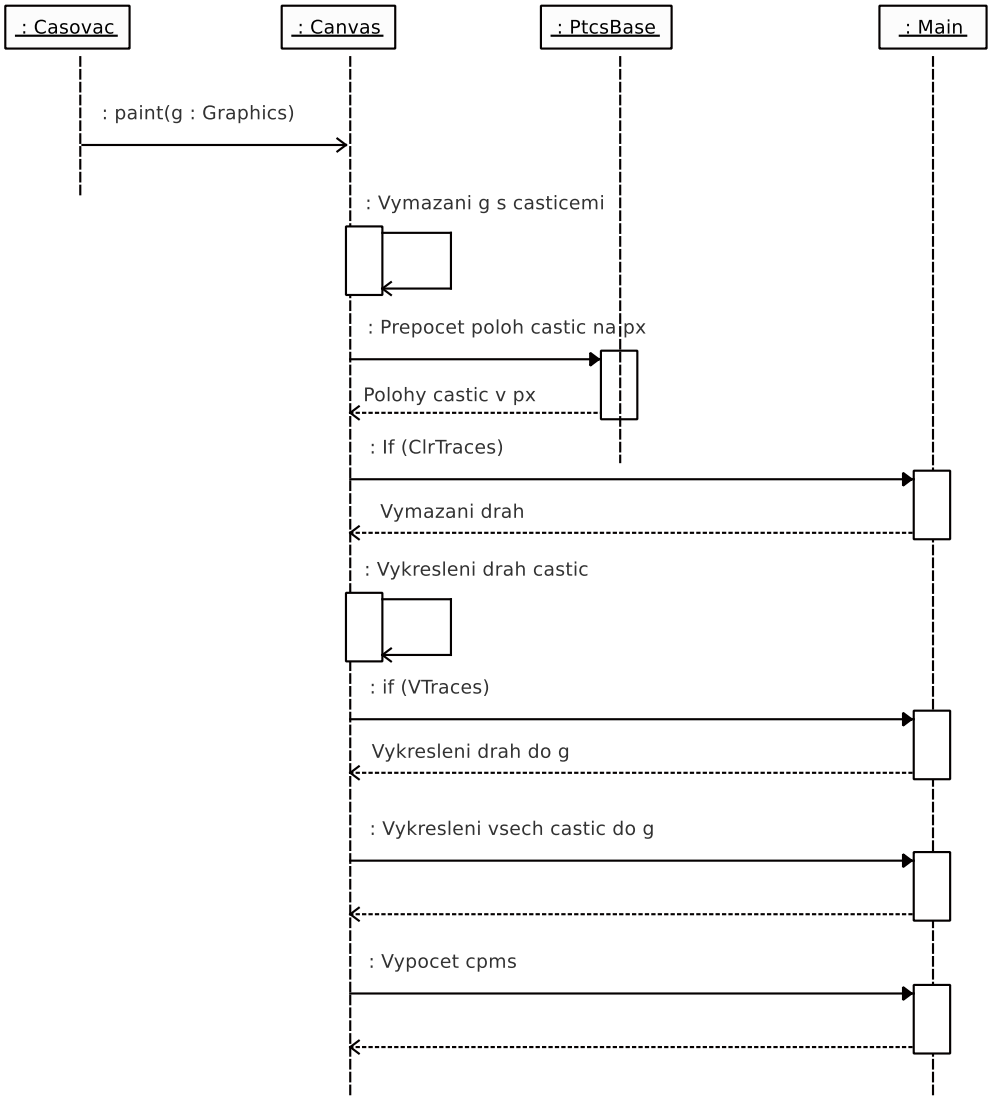
Hlavní okno programu `Ptcsim` je vytvářeno statickou metodou `createAndShowGUI()`, která se nachází ve třídě `Gui.java`. Při psaní této třídy nebyl, na rozdíl od ostatních dvou, použit okenní editor IDE NetBeans (2.1.2). To proto, že je vykreslování částic realizováno v komponentě `JPanel`, kterou bylo třeba upravit. V IDE Netbeans nelze zasahovat do zdrojového kódu vygenerovaného okenním editorem, což bylo potřeba v případě hlavního okna.

V metodě `createAndShowGUI()` je nejprve vytvořeno samotné okno (instance třídy `JFrame`), s použitím okenního manažeru `GridBagLayout`. Do tohoto okna je pak vložen panel pro vykreslování částic (`Canvas.java` (5.2.6.2)). Pod tímto „kreslícím plátnem“ je vložen panel, na kterém se nachází `JLabel` pro vypisování počtu výpočtů za jednu milisekundu a

pak ovládací prvky simulace (instance tříd `JSpinner` a `JButton`). Pro bližší informace ohledně vytvoření hlavního okna je zdrojový kód souboru `Gui.java` přiložen na CD.
Dalším třídou, která dědí ze třídy `JFrame` je třída `SConst.java`. Jedná se o okno, které je na obrázku 5.4. Toto okno bylo vytvořeno v okenním editoru IDE NetBeans. Obsahuje instance tříd `JLabel` (popisky jednotlivých polí), `JTextField` (pole pro zadávání čísel), `JSpinner` (pole pro zadávání simulačního prostoru), `JCheckBox` (volba odrážení od hranic simulace) a `JButton` (tlačítka). Při zobrazení tohoto okna (uživatelem, když v nabídce *Simulation* vybere volbu *Set constants...*, jsou příslušná pole vyplněna hodnotami, jak jsou nastaveny v programu. Zde uživatel může měnit hodnoty jednotlivých konstant. V případě, že nebyla zaškrtnuta volba odrážení od hranic simulace a uživatel zadá, že chce odrážet částice od stěn, tak se zobrazí informační dialog, který informuje o tom, že částice, které byly mimo simulační prostor do něj budou vráceny, tudíž dojde k chybě simulace a ptá se, je-li si uživatel jistý. Zaškrtnutí obsluhuje událost `ActionPerformed` tohoto zaškrtávacího políčka. Když uživatel klikne na tlačítko *Apply Changes*, tak je volána obsluha této události, která uloží hodnoty ze všech polí do patřičných proměnných a skryje okno.
Tlačítka *Apply Changes* a *OK* mají velmi podobnou úlohu, rozdíl je jen v tom, že tlačítko *OK* toto okno skryje. Při kliknutí na tlačítko *Apply Changes* je volána metoda `btnApplyActionPerformed()`, která obsahuje načtení hodnot ze všech polí okna a jejich uložení do příslušných proměnných. Ke kontrole, jestli jsou v polích zadaná čísla dochází v okamžiku, kdy uživatel hodnoty zadává. Pokud byla změněna hodnota velikosti simulačního prostoru (v *px*), tak tato metoda volá inicializační metodu třídy `Canvas` (5.2.6.2).
Okno About (`About.java`) je v programu dostupné z nabídky *About* z jediné položky tohoto menu *About...*. Toto oknou slouží pouze k výpisu informací o programu – hlavně o autorovi a tomu, proč a kde byl vytvořen. Tento text je vložen ve víceřádkovém textovém poli (`JTextArea`) a jediné tlačítko OK má pouze funkci skrytí tohoto okna.

5.2.6.2 Třída Canvas.java

Třída Canvas dědí ze třídy `JPanel` a rozšiřuje její možnosti. Zejména přetížením metody `Paint()`. Dále implementuje rozhraní `Runnable`, které této třídě dává možnosti stejné, jako kdyby byla vláknem.
Součástí této třídy je „tabulka barev“. Jedná se o pole barev. Z této tabulky se vybírá barva podle hodnoty, jaká je uložena v proměnné `col` v instanci částice (`Particle.java` 5.2.2). Konstruktor tuto tabulku inicializuje. Pak volá metodu `init()`, která vytváří obrázek pro budoucí kreslení tras částic. Jedná se o instanci třídy `BufferedImage`. Z té pak získává grafický kontext, instanci třídy `Graphics2D`. Nakonec konstruktor vyplní celý



Obrázek 5.9: Sekvenční diagram vykreslování částic ve třídě `Canvas.java`

obrázek pro kreslení drah částic bílou barvou.

Metoda `Paint()` překresluje částice na ploše okna a vykresluje dráhy částic. Tato metoda je volána každých 50 ms, takže je vytvářen dojem, že se jedná o plynulý pohyb. V této metodě figurují dva grafické kontexty. Jedná se o `g`, což je grafický kontext, do kterého jsou vykreslovány samotné částice reprezentované vyplněnými kružnicemi. Druhým je `g2D`, do kterého jsou vykreslovány dráhy částic. Oddělené je to z důvodu nezávislosti vykreslování, a také z toho důvodu, že částice je třeba při každém překreslení vymazat, kdežto trasy částic ne. Postup překreslování je takový, že se nejprve přepočítají polohy x a y pro každou částici tak, aby byla tato hodnota v px v daném rozsahu kreslící plochy. Pak se do `g2D` vykreslí dráhy částic od předchozího kroku k současnému. Pro každou částici

v barvě, která jí náleží. Pokud uživatel vybral v menu *Simulation* volbu *Clear traces*, tak se ještě před tím `g2D` vyplní bílou barvou, čímž se předchozí dráhy vymažou. Pokud se mají vykreslit dráhy částic (je zaškrtnutá položka *View traces* v menu *Simulation*), tak se `g2D` vykreslí do `g`. Pak je možné vykreslit samotné částice – vyplněné kružnice, každá v náležité barvě částice.

Metoda `run()`, která je volána po spuštění vlákna překreslování po vytvoření instance této třídy v metodě `CreateAndShowGUI()` ve třídě `Gui.java` (5.2.6.1) v těle obsahuje nekonečnou smyčku, ve které se neustále volá metoda `paint()` příkazem `this.repaint()`. Pak se vlákno překreslování uspává na 50 *ms*.

Sekvenční diagram (4.3.3) vykreslování částic je uveden na obrázku 5.9.

Závěr

Byl vytvořen program pro simulaci částic v mnohočásticových soustavách. Program je zveřejněn pod svobodnou licencí *GPL*, což umožňuje jeho další rozšiřování.

Program nesplnil některé požadavky, které na něj byly kladeny. Jedním z nich bylo zesvětlování drah částic. Idea byla taková, že by se plátno s drahami po každém vykreslení zesvětlilo o určité procento jasu a tím by se dosáhlo efektu zesvětlování drah. Dráhy by se tím pádem ztrácely a nebylo by třeba funkce v menu *Simulation Clear Traces*. Bohužel se ale nepodařilo napsat algoritmus zesvětlování, aby byl dostatečné rychlý. Pokud by byl na obrázek aplikován, tak by se snížila frekvence překreslování až dvacetkár, čímž by byl zbytečně zaměstnáván procesor, který je potřeba na přepočty poloh částí.

Dalším nedostatkem programu *Ptcsim* je jeho rychlost. V případě přepočtů poloh 10 částic stíhá přepočítat polohy méně než 1000x za sekundu. To je způsobeno zejména tím, že je metoda přepočtu částic ze třídy `PtcCompute.java` (5.2.5) volána podle jména. Pokud byla tato metoda volána explicitně, tak program dosahoval až desetkrát lepších výsledků. Pro praktické použití tohoto programu tedy bude třeba volat metodu přímo a ne podle jména.

Program splňuje všechny požadavky, tedy vykresluje částice v reálném čase na obrazovku a umožňuje i výpis atributů jednotlivých částic do souborů. Taktéž využívá výpočetní výkon všech procesorů, které jsou k dispozici.

Závěr v angličtině

Program for particle simulation in many particles systems was created. Program is published uder freedom license `GPL`. It enable its spreading.

The program did'n fulfil some requirements that were placed. One of this was the lightening of traces of the particles. Idea was, that after draw particle traces the canvas is lightening per some percent brightness and it will create the lightening effect. The traces would be lost and the item in menu *Simulation* `Clear Craces` would be pointless. Unfortunately, the algorithm of lightening could not be fast enought. If it was applied for image the frequency would be reduced 20x. It would be use the processor unnecessarily which is need for recalculatig particles positions.

The next imperfection of program *Ptcsim* is it's speed. In case recomputing particle positions of 10 particles the speed is less than 1000x per second. It is due to calling method for recalculing from the class `PtcCompute.java` (5.2.5) by the name. The program Ptcsim would achieve better results, if it would be called explicitly. The method must be call explicitly for practise use of Ptcsim program.

The program meet all requirements. It draw the particles in real time on screen. It enable writing particle attributes into the text files in `XML` format and `csv` format. The program uses computing output of all processors available.

Literatura

[1] Herout, P., Učebnice jazyka Java. Praha: Kopp, 2000, ISBN 80-7232-115-3

[2] Herout, Pavel,. Java a XML /. 1. vyd. České Budějovice : Kopp, 2007. 313 s. : ISBN 978-80-7232-307-4 (brož.).

[3] Young, Michael J.,. XML krok za krokem /. 2. vyd. Brno : Computer Press, 2006. 471 s. : ISBN 80-251-1070-2 (brož.) :.

[4] Dřímal, Tomáš. Obrazová analýza radiálně symetrických vzorků. 2008. 46 s., 18s

[5] PILATO, C. Michael; COLLINS-SUSSMAN, Ben; FITZPATRICK, Brian W. Version Control with Subversion. 2. USA: O'Reilly, 2008. 432 s. Dostupné z WWW: <http://svnbook.red-bean.com/> . ISBN 978-0596510336.

[6] Arlow, Jim,. UML 2 a unifikovaný proces vývoje aplikací : objektově orientovaná analýza a návrh prakticky /. 2., aktualiz. a dopl. vyd. Brno : Computer Press, 2007. 567 s. : ISBN 978-80-251-1503-9 (brož.).

[7] Unified Modeling Language In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, 14. 2. 2005, 28. 11. 2009 [cit. 2010-03-11]. Dostupné z WWW: <http:// cs.wikipedia.org/wiki/Unified_ Modeling_ Language>.

[8] Arlow, Jim,. UML a unifikovaný proces vývoje aplikací : průvodce analýzou a návrhem objektově orientovaného softwaru /. Vyd. 1. Brno : Computer Press, 2003. xviii, 387 s. : ISBN 807226947X (brož.).

[9] CSC., Ivo. Úvod do softwarového inženýrství. Ostrava, 2002. 74 s. Skripta. VŠB – Technická univerzita Ostrava, Fakulta elektrotechniky a informatiky, katedra informatiky.

[10] Swing (Java) In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, 11. 5. 2009, 22. 12. 2009 [cit. 2010-03-15]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Swing_%28Java%29>.

[11] ZAKHOUR, Sharon, et al. Java 6 : Výukový kurz. 1. Brno : Computer Press, Výukový kurz. 536 s. ISBN 978-80-251-1575-6.

[12] KOTALA, Z.; TOMAN, P. Vlákna (threads). Dioné [online]. 2001, [cit. 2010-03-16]. Dostupný z WWW: <http://v1.dione.zcu.cz/java/sbornik/16.html>.

Seznam použitých symbolů a zkratek

Částice je velmi malá část hmoty, která se projevuje svými charakteristickými vlastnostmi (energií, hmotností, elektrickým nábojem, spinem, chemickou reaktivností, dobou života, aj.)

IDE Integrated Development Environment – programové vývojové prostředí

SDK Software Development Kit – množina vývojových nástrojů pro určitý programovací jazyk nebo programové prostředí

API Application Programming Interface – rozhraní pro programování aplikací

XML Extensible Markup Language – je obecný značkovací jazyk

HTML HyperText Markup Language – značkovací jazyk pro hypertext

Unicode tabulka znaků všech existujících abeced, která v současnosti obsahuje více než 100 000 znaků

UTF-8 způsob kódování řetězců znaků Unicode/UCS do sekvencí bajtů

GUI Graphical User Interface - Grafické uživatelské rozhraní; uživatelské rozhraní, které umožňuje ovládat počítač pomocí interaktivních grafických ovládacích prvků

Seznam obrázků

Seznam příloh

Příloha P I: Zdrojový kód souboru Main.java

```
1  package psim;
2
3  import gui.Gui;
4  import java.lang.reflect.Method;
5  import java.util.ArrayList;
6  import javax.swing.SwingUtilities;
7
8  /**
9   * Hlavni trida programu Ptcsim
10  * @author Tomas Drimal
11  */
12 public class Main {
13     // promenna udrzujici hodnotu poctu cyklu
14     public static long cykly = 0;
15     // instance tridy castic
16     public static PtcsBase particles;
17 //    pocet procesoru
18     public static int nrOfCPU = 1;
19     public static double dt = 100; //krok casu v sekundach
20 //    rozmery simulovaneho prostoru
21     public static double scale_x ;
22     public static double scale_y ;
23     // pole historie vykreslenych castic
24     public static ArrayList<int[]> PtcsHist = new ArrayList<int[]>();
25     //globalni promenna, ktera udrzuje informaci, jestli simulovat, nebo ne
26     public static boolean simulation = false;
27     //boolean promenna, ktera rika, jestli se ma vymazat platno
28     public static boolean clrCanvas = false;
29 //    Rozmery simulacniho zobrazovaneho prostoru
30     public static int CLIENT_WIDTH_PX = 600;
31     public static int CLIENT_HEIGHT_PX = 600;
32     // jestli se maji castice odrazet od sten
33     public static boolean bound = true;
34     // jestli zobrazovat stopy castic
35     public static boolean VTraces = true;
```

```
36      // jestli smazat stopy castic
37      public static boolean clrTraces = false;
38  //    Pocet kroku, ktere se maji provest
39      public static long nrOfSteps = -1;
40  //    jmeno metody, ktera se pouziva pro prepocet rychlosti castic
41      public static String mthdName ="";
42  //    seznam vsech dostupnych metod pro prepocet rychlosti castic
43      public static Method[] cptMet = PtcCompute.class.getDeclaredMethods();
44
45      /**
46       * @param args the command line arguments
47       */
48      public static void main(String[] args) {
49
50          //zjisteni poctu jader procesoru
51          Runtime runtime = Runtime.getRuntime();
52          nrOfCPU = runtime.availableProcessors();
53
54          if (nrOfCPU == 1) {
55              particles = new PtcsSigle();
56          } else {
57              particles = new PtcsMulti(nrOfCPU);
58          }
59  //        vyrvoreni GUI
60          SwingUtilities.invokeLater(new Runnable() {
61
62              public void run() {
63                  Gui.createAndShowGUI();
64              }
65          });
66  //          spusteni simulace
67              particles.startSim();
68
69
70      }
71  }
```

Příloha P II: Zdrojový kód souboru Particle.java

```
1  package psim;
2
3  /**
4   * Tato třída zapouzdruje jednu castici - všechny její parametry
5   * @author Tomas Drimal
6   */
7  public class Particle {
8
9      public double x; //x-ova poloha
10     public double y; //y-nova poloha
11     public double vx; //rychlost ve smeru osy x
12     public double vy; //rychlost ve smeru osy y
13     public double m; //hmotnost castice
14     public double q; //naboj castice
15     public double d; //prumer castice
16     public short col; //barva castice
17     public double old_x; //predchozi x-ova poloha
18     public double old_y; //predchozi y-nova poloha
19     public double old_vx; //predchozi rychlost ve smeru osy x
20     public double old_vy; //predchozi rychlost ve smeru osy y
21
22     /**
23      * Konstruktor tridy particle -
24      * jako parametry jsou vsechny vlastnosti tridy
25      * @param c_x - xova poloho
26      * @param c_y - ynov poloha
27      * @param c_vx - xova rychlost
28      * @param c_vy - ynova rychlost
29      * @param c_m - hmotnost castice
30      * @param c_q - naboj castice
31      * @param c_d - prumer castice
32      * @param c_col - barva castice
33      */
34     Particle(double c_x, double c_y, double c_vx, double c_vy,
35             double c_m, double c_q, double c_d, short c_col) {
```

```
36          this.x = c_x;
37          this.y = c_y;
38          this.vx = c_vx;
39          this.vy = c_vy;
40          this.m = c_m;
41          this.q = c_q;
42          this.d = c_d;
43          this.col = c_col;
44
45          saveVals();
46      }
47
48      /**
49       * Metoda pro vypocet zmeny polohy
50       * pricita k aktualni poloze hodnotu rychlosti
51       * pro kazdou slozku zvlast
52       */
53      synchronized public void changePos() {
54          x = x + (vx * Main.dt);
55          y = y + (vy * Main.dt);
56          if (Main.bound) {
57
58              if (x > Main.scale_x) {
59                  vx = vx * (-1);
60              }
61              if (y > Main.scale_y) {
62                  vy = vy * (-1);
63              }
64              if (x < 0) {
65                  vx = vx * (-1);
66              }
67              if (y < 0) {
68                  vy = vy * (-1);
69              }
70          }
71
72      }
```

```
73
74      /**
75       * Metoda, ktera preklopi nove hodnoty do starych
76       */
77      public void saveVals() {
78          this.old_x = this.x;
79          this.old_y = this.y;
80          this.old_vx = this.vx;
81          this.old_vy = this.vy;
82      }
83      /**
84       * Metoda, ktera hodnoty castice prevede do retezce pro ulozeni
85       * do souboru typu pts (XML)
86       * @return retezec, ktery SAX serializer vlozi mezi tagy "<ptc>" a "</ptc>"
87       */
88      public String tString() {
89          StringBuffer buff = new StringBuffer();
90          buff.append(Double.toString(x) + " ");
91          buff.append(Double.toString(y) + " ");
92          buff.append(Double.toString(vx) + " ");
93          buff.append(Double.toString(vy) + " ");
94          buff.append(Double.toString(m) + " ");
95          buff.append(Double.toString(q) + " ");
96          buff.append(Double.toString(d) + " ");
97          buff.append(Short.toString(col));
98          return buff.toString();
99      }
100     /**
101      * Metoda, ktera prevede hodnoty vlastnosti castice na
102      * retezec, ktery je jednim z radku csv souboru
103      * @return retezec, ktery je jednim radkem csv souboru
104      */
105     public String toCsvLine() {
106         StringBuffer buff = new StringBuffer();
107         buff.append(Double.toString(x) + ",");
108         buff.append(Double.toString(y) + ",");
109         buff.append(Double.toString(vx) + ",");
```

```
110             buff.append(Double.toString(vy) + ",");
111             buff.append(Double.toString(m) + ",");
112             buff.append(Double.toString(q) + ",");
113             buff.append(Double.toString(d) + ",");
114             buff.append(Short.toString(col));
115             return buff.toString();
116         }
117     }
```

Příloha P III: Zdrojový kód souboru PtcsBase.java

```
1  package psim;
2
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.io.IOException;
6  import java.lang.reflect.InvocationTargetException;
7  import java.lang.reflect.Method;
8  import java.util.ArrayList;
9  import java.util.logging.Level;
10 import java.util.logging.Logger;
11 import org.xml.sax.SAXException;
12
13 /**
14  * Trida, ktera zapouzdruje pole castic a operace s nimi
15  * Je to rodicovska trida, jsou od ni odvozene tridy PtcsMulti a PtcsSingle
16  *
17  * @author Tomas Drimal
18  */
19 public abstract class PtcsBase {
20     //pole castic
21     public ArrayList<Particle> partcs = new ArrayList<Particle>();
22     // pole dvou castic, z duvodu volani metody podle jmena
23     Class pts[] = {Particle.class,Particle.class};
24
25     /**
26      * Metoda, která volá LoadFromFile(String Filename), vpodstate ji
27      * zapouzdruje. Z duvodu nasledneho dedeni teto PtcsBase tridy.
28      * @param filename - jméno csv souboru s částicemi
29      * @return true, pokud je vse v pořádku, jinak vraci false
30      * @see #LoadFromFile(String Filename) LoadFromFile(String Filename)
31      */
32     public boolean LoadFromFile(String filename) {
33         if (LoadFrFile(filename)) {
34             return true;
35         } else {
```

```
36              return false;
37          }
38      }
39
40      /**
41       * Metoda, ktera nacte do pole castice ze souboru
42       * @param Filename - jméno csv souboru s casticemi
43       * @return true, pokud je vse v pořádku, jinak vraci false
44       */
45      public boolean LoadFrFile(String Filename) {
46
47          //smazani pole castic
48          partcs.clear();
49          //smazani pole s historii
50          Main.PtcsHist.clear();
51          FileAccess.LoadFromFile(Filename);
52          //Inicializace simulace
53          for (int i = 0; i < Main.particles.size(); i++) {
54              //x-ova a y-ova souradnice aktualni polohy castice (dano stredem)
55              int x = (int) (partcs.get(i).x / Main.scale_x * psim.Main.CLIENT_WIDTH_
56              int y = (int) (partcs.get(i).y / Main.scale_y * psim.Main.CLIENT_HEIGHT
57
58              int pole[] = {x, y, (int) partcs.get(i).d / 2, x, y};
59              Main.PtcsHist.add(pole);
60          }
61          Main.cykly = 0;
62          return true;
63      }
64      /*
65       * Metoda, ktera ulozi castice i s nastavenim simulace
66       * do souboru typu pts
67       */
68      public void SaveToFile(String Filename) {
69          try {
70              FileAccess.SaveToFile(Filename);
71          } catch (FileNotFoundException ex) {
72              Logger.getLogger(PtcsBase.class.getName()).log(Level.SEVERE, null, ex);
```

```
73          } catch (IOException ex) {
74              Logger.getLogger(PtcsBase.class.getName()).log(Level.SEVERE, null, ex);
75          } catch (SAXException ex) {
76              Logger.getLogger(PtcsBase.class.getName()).log(Level.SEVERE, null, ex);
77          }
78      }
79
80      /**
81       * Metoda prida instanci Particle na konec pole
82       * @param pts Prticle, ktere ma byt pridano
83       */
84      public void add(Particle pt) {
85          this.partcs.add(pt);
86      }
87
88      /**
89       * Metoda vrati Particle na pozici i
90       * @param i index pozadovane instance Particle
91       * @return pozadovanou castici na indexu i
92       */
93      public Particle get(int i) {
94          return this.partcs.get(i);
95      }
96
97      /**
98       * Metoda pro zjisteni poctu castic v poli
99       * @return Pocet castic v poli
100      */
101     public int size() {
102         return this.partcs.size();
103     }
104
105     /**
106      * Abstraktní metoda, jeji telo je ve tridach PtcsSingle a PtcsMulti
107      */
108     public abstract void chPos();
109
```

```
110     /**
111      * Metoda, ktera zavola metodu Patricle.changePos v intervalu from-to
112      * @param from zacatek intervalu
113      * @param to konec intervalu
114      */
115     public void chPosFT(int from, int to) throws NoSuchMethodException,
116             IllegalAccessException, IllegalArgumentException,
117             InvocationTargetException {
118         for (int i = from; i < to; i++) {
119             for (int j = i + 1; j < this.partcs.size(); j++) {
120                 Method mthd=PtcCompute.class.getMethod(Main.mthdName,pts);
121                 mthd.invoke(null,partcs.get(i), partcs.get(j));
122             }
123             Main.particles.get(i).changePos();
124         }
125 
126     }
127 
128     /**
129      * metoda, ktera ulozi hodnoty poloh a rychlosti z predchoziho (soucasneho)
130      * kroku
131      */
132     public void saveNew() {
133         for (Particle pt : partcs) {
134             pt.saveVals();
135         }
136     }
137     /*
138      * Metoda spusteni simulace
139      */
140     public void startSim() {
141         while (true) {
142             if (Main.simulation) {
143                 if (Main.nrOfSteps == -1) {
144                     chPos();
145                 }
146                 if (Main.nrOfSteps == 0) {
```

```
147                     Main.simulation = false;
148                     Main.nrOfSteps = -1;
149                     gui.Gui.btnPlay.setEnabled(true);
150                     gui.Gui.btnPlay0.setEnabled(true);
151                 }
152                 if (Main.nrOfSteps > 0) {
153                     for (long i = Main.nrOfSteps; i > 0; i--) {
154                         chPos();
155                         Main.nrOfSteps--;
156 
157                     }
158                 }
159             }
160 
161         }
162     }
163     /*
164      * Metoda, ktera ulozi castice do souboru typu csv
165      */
166     public void toCsvFile(File file)
167             throws FileNotFoundException, IOException {
168         ArrayList<String> buff = new ArrayList<String>();
169         for (Particle pt : partcs) {
170             buff.add(pt.toCsvLine());
171         }
172         FileAccess.SaveToCSV(file, buff);
173     }
174 }
```

Příloha P IV: Zdrojový kód souboru PtcsMulti.java

```
1  package psim;
2
3  import java.lang.reflect.InvocationTargetException;
4  import java.util.logging.Level;
5  import java.util.logging.Logger;
6
7  /**
8   * Trida threads, ktera je instanciovana tolikrat,
9   * kolik je procesoru k dispozici
10  * @author Tomas Drimal
11  */
12 class ThreadPtcs extends Thread {
13     //dolni hranice intervalu vypoctu
14     int from;
15     //horni hranice intrevalu vypoctu
16     int to;
17
18     /**
19      * Kostruktor teto tridy
20      * @param name jmeno threads
21      * @param from dolni hranice intervalu, pro ktery bude tato instance pocitat
22      * @param to horni hranice intervalu, pro ktery bude tato instance pocitat
23      */
24     public ThreadPtcs(String name, int from, int to) {
25         super(name);
26         this.from = from;
27         this.to = to;
28     }
29
30     /**
31      * Metoda run, obsahuje vykonnou cast tohoto threads
32      */
33     @Override
34     public void run() {
35         try {
```

```
36              Main.particles.chPosFT(this.from, this.to);
37          } catch (NoSuchMethodException ex) {
38              Logger.getLogger(ThreadPtcs.class.getName()).log(Level.SEVERE,
39                      null, ex);
40          } catch (IllegalAccessException ex) {
41              Logger.getLogger(ThreadPtcs.class.getName()).log(Level.SEVERE,
42                      null, ex);
43          } catch (IllegalArgumentException ex) {
44              Logger.getLogger(ThreadPtcs.class.getName()).log(Level.SEVERE,
45                      null, ex);
46          } catch (InvocationTargetException ex) {
47              Logger.getLogger(ThreadPtcs.class.getName()).log(Level.SEVERE,
48                      null, ex);
49          }
50
51      }
52  }
53
54  /**
55   * Trida, ktera dedi vlastnosti tridy PtcsBase a rozsiruje je pro praci s vice
56   * vlakny
57   * @author Tomas Drimal
58   */
59  public class PtcsMulti extends PtcsBase {
60      //pole vlaken
61
62      private ThreadPtcs threads[];
63      //pole intervalu, aby se meze nemusely pocitat stale dokola
64      private int intervals[][];
65      //pocet CPU k dispozici
66      private int CPUnr;
67
68      //konstruktor
69      public PtcsMulti(int CPUnr) {
70          //rodicovky konstruktor
71          super();
72          this.CPUnr = CPUnr;
```

```
73          //pocet vlaken podle poctu procesoru
74          threads = new ThreadPtcs[CPUnr];
75      }
76
77      /*
78       * Metoda, ktera rozdeluje vypocet na vice procesoru
79       */
80      public void chPos() {
81  //        System.out.println("Vice procesoru");
82              //start vlaken
83              for (int i = 0; i < threads.length; i++) {
84                  //instancizace threads
85                  threads[i] = new ThreadPtcs("Thread",
86                          intervals[i][0], intervals[i][1]);
87                  //start threads
88                  threads[i].start();
89              }
90              //cekani na ukonceni cinnosti vlaken
91              try {
92                  for (int i = 0; i < threads.length; i++) {
93                      threads[i].join();
94                  }
95              } catch (InterruptedException ex) {
96                  Logger.getLogger(PtcsMulti.class.getName()).log(Level.SEVERE,
97                          null, ex);
98              }
99              this.saveNew(); //ulozeni novych hodnot do starych
100             Main.cykly++;
101     }
102
103     /*
104      * Metoda, ktera pocita intervaly v zavislosti na poctu procesoru a
105      * poctu castic
106      */
107     public void computeIntervals() {
108         //vypocet hranic intervalu pro rozdeleni castic na procesory
109         intervals = new int[CPUnr][2]; //inicializace pole
```

```
110         double tmp = 0;
111         intervals[0][0] = 0;
112         intervals[0][1] = (int) (partcs.size() * (1 -
113                 (Math.sqrt((Double.valueOf(CPUnr) - 1) / CPUnr))));
114 
115         for (int i = 1; i < CPUnr; i++) {
116             intervals[i][0] = intervals[i - 1][1];
117             tmp = partcs.size() * (1 - (Math.sqrt((Double.valueOf(CPUnr)
118                     - (i + 1)) / CPUnr)));
119             intervals[i][1] = (int) (tmp);
120         }
121     }
122 
123     /**
124      * Pretizeni metody z rodicovske tridy. Jednak vola metodu nacteni
125      * castic ze tridy PtcsBase a pak pocita intervaly pro ucely
126      * multitaskingu
127      * @param filename name pts souboru s ulozenymi casticemi
128      * @return true, jestli nedoslo k chybe
129      */
130     @Override
131     public boolean LoadFromFile(String filename) {
132         if (LoadFrFile(filename)) {
133             computeIntervals();
134             return true;
135         } else {
136             return false;
137         }
138     }
139 }
```

Příloha P V: Zdrojový kód souboru PtcCompute.java

```
1  package psim;
2
3  /**
4   * Trida, ktera zapouzdruje staticke metody, ktere pocitaji interakci
5   * samotnych castic
6   * @author Tomas Drimal
7   */
8  public class PtcCompute {
9
10     /**
11      * Metoda, ktera pocita novou rychlost castice pt1 z castice pt2
12      * uvazuji se naboje jednotlivych castic
13      * @param pt1 castice, pro kterou se pocita nova rychlost
14      * @param pt2 castice, ze ktere se pocita nova rychlost
15      */
16    public static void chVelElectricity(Particle pt1, Particle pt2){
17         // x ova vydalenost mezi pt1 a pt2
18         double dx = pt2.old_x - pt1.old_x;
19         // z nova vzdalenost mezi pt1 a pt2
20         double dy = pt2.old_y - pt1.old_y;
21
22         double d2 = (dx*dx) + (dy*dy);
23         double k_d2 = 9e16/d2*1;
24         double a1=k_d2*pt2.q; //zrychleni 1. castice
25         double a2=k_d2*pt1.q; //zrychleni 2. castice
26         // distance mezi obema casticemi
27         double d=Math.sqrt(d2);
28         double r0x=dx/d; //x-slozka jednotkoveho vektoru od pt1 k pt2
29         double r0y=dy/d; //y-slozka jednotkoveho vektoru od pt1 k pt2
30
31         pt1.vx = pt1.vx-a1*Main.dt*r0x;
32         pt1.vy = pt1.vy-a1*Main.dt*r0y;
33         pt2.vx = pt2.vx+a2*Main.dt*r0x;
34         pt2.vy = pt2.vy+a2*Main.dt*r0y;
35     }
```

```
36
37      /**
38       * Metoda, ktera pocita novou rychlost castice pt1 z castice pt2
39       * uvazuje se gravitacni sila mezi casticemi
40       * @param pt1 castice, pro kterou se pocita nova rychlost
41       * @param pt2 castice, ze ktere se pocita nova rychlost
42       */
43      public static void chVelGravitational(Particle pt1, Particle pt2){
44          // x ova vydalenost mezi pt1 a pt2
45          double dx = pt2.old_x - pt1.old_x;
46          // z nova vzdalenost mezi pt1 a pt2
47          double dy = pt2.old_y - pt1.old_y;
48
49          double d2 = (dx*dx) + (dy*dy);
50          double kappa_d2 = 6.67e-11/d2*1;
51          double a1=kappa_d2*pt2.m; //zrychleni 1. castice
52          double a2=kappa_d2*pt1.m; //zrychleni 2. castice
53          // distance mezi obema casticemi
54          double d=Math.sqrt(d2);
55          double r0x=dx/d; //x-slozka jednotkoveho vektoru od pt1 k pt2
56          double r0y=dy/d; //y-slozka jednotkoveho vektoru od pt1 k pt2
57
58          pt1.vx = pt1.vx+a1*Main.dt*r0x;
59          pt1.vy = pt1.vy+a1*Main.dt*r0y;
60          pt2.vx = pt2.vx-a2*Main.dt*r0x;
61          pt2.vy = pt2.vy-a2*Main.dt*r0y;
62      }
63  }
```

Příloha P VI: Zdrojový kód souboru FileAccess.java

```
1  package psim;
2
3  import com.sun.org.apache.xml.internal.serialize.OutputFormat;
4  import com.sun.org.apache.xml.internal.serialize.XMLSerializer;
5  import java.io.BufferedWriter;
6  import java.io.File;
7  import java.io.FileNotFoundException;
8  import java.io.FileOutputStream;
9  import java.io.FileWriter;
10 import java.io.IOException;
11 import java.io.Writer;
12 import java.util.ArrayList;
13 import javax.xml.parsers.SAXParser;
14 import javax.xml.parsers.SAXParserFactory;
15 import org.xml.sax.Attributes;
16 import org.xml.sax.ContentHandler;
17 import org.xml.sax.XMLReader;
18 import org.xml.sax.SAXException;
19 import org.xml.sax.SAXParseException;
20 import org.xml.sax.helpers.AttributesImpl;
21 import org.xml.sax.helpers.DefaultHandler;
22
23 /**
24  * Trida, ktera prostrednictvim statickych metod implementuje
25  * XML SAX parser a serializer.
26  * @author Tomas Drimal
27  */
28 public class FileAccess extends DefaultHandler {
29
30     int priznak = 500;
31     private String[] StrNums;
32     private StringBuffer buff = new StringBuffer();
33
34     /**
35      * metoda ktera nacte xml soubor s casticemi
```

```
36          * @param name jmeno souboru
37          */
38         public static void LoadFromFile(String name) {
39             try {
40                 SAXParserFactory factory = SAXParserFactory.newInstance();
41                 factory.setValidating(false);
42 
43                 SAXParser parser = factory.newSAXParser();
44 
45                 XMLReader reader = parser.getXMLReader();
46 
47                 FileAccess test = new FileAccess();
48                 reader.setContentHandler(test);
49                 reader.setErrorHandler(test);
50                 reader.parse(name);
51             } catch (Exception e) {
52                 System.err.println(e.getMessage());
53             }
54         }
55 
56         /**
57          * Obsluzna metoda, je volana vzdy, kdyz
58          * parser narazi na zacatek XML elementu
59          * @param uri
60          * @param localName
61          * @param qName
62          * @param atts
63          * @throws SAXException
64          */
65         @Override
66         public void startElement(String uri, String localName,
67                 String qName, Attributes atts) throws SAXException {
68 
69             if (qName.equalsIgnoreCase("dt")) {
70                 priznak = 0;
71             }
72             if (qName.equalsIgnoreCase("scale_x")) {
```

```
73              priznak = 1;
74          }
75          if (qName.equalsIgnoreCase("scale_y")) {
76              priznak = 2;
77          }
78          if (qName.equalsIgnoreCase("bound")) {
79              priznak = 3;
80          }
81          if (qName.equalsIgnoreCase("ptc")) {
82              priznak = 4;
83          }
84      }
85
86      /**
87       * Obsluzna metoda parseru, je volana vzdy,
88       * kdyz parser narazi na konec XML elementu
89       * @param uri
90       * @param localName
91       * @param qName
92       * @throws SAXException
93       */
94      @Override
95      public void endElement(String uri, String localName,
96              String qName) throws SAXException {
97          switch (priznak) {
98              case 0:
99                  Main.dt = Double.valueOf(buff.toString());
100             case 1:
101                 Main.scale_x = Double.valueOf(buff.toString());
102                 break;
103             case 2:
104                 Main.scale_y = Double.valueOf(buff.toString());
105                 break;
106             case 3:
107                 Main.bound = Boolean.parseBoolean(buff.toString());
108                 break;
109             case 4:
```

```
110                 StrNums = buff.toString().split("[ ]");
111                 double x = Double.valueOf(StrNums[0]); //x-ova poloha
112                 double y = Double.valueOf(StrNums[1]); //y-nova poloha
113                 double vx = Double.valueOf(StrNums[2]); //rychlost ve smeru osy x
114                 double vy = Double.valueOf(StrNums[3]); //rychlost ve smeru osy y
115                 double m = Double.valueOf(StrNums[4]); //hmotnost castice
116                 double q = Double.valueOf(StrNums[5]); //naboj castice
117                 double d = Double.valueOf(StrNums[6]); //prumer castice
118                 short col = Short.valueOf(StrNums[7]); //barva castice
119 
120                 Main.particles.add(new Particle(x, y, vx, vy, m, q, d, col));
121                 break;
122             default:
123         }
124         priznak = 500;
125         buff.delete(0, buff.length());
126     }
127 
128     /**
129      * Metoda, ktera je volana nad znaky, ktere jsou mezi
130      * elementy XML dokumentu - do bufferu uklada znaky,
131      * ktere se nachazi mezi temito elementy
132      * @param ch
133      * @param start
134      * @param length
135      * @throws SAXException
136      */
137     @Override
138     public void characters(char[] ch, int start, int length)
139             throws SAXException {
140         buff.append(new String(ch, start, length));
141     }
142 
143     @Override
144     public void error(SAXParseException e) {
145         System.out.println(e.getMessage());
146     }
```

```
147
148     @Override
149     public void warning(SAXParseException e) {
150         System.out.println(e.getMessage());
151     }
152
153     @Override
154     public void fatalError(SAXParseException e) {
155         System.out.println(e.getMessage());
156     }
157
158     public static void SaveToFile(String filename) throws FileNotFoundException, IO
159         FileOutputStream fos = new FileOutputStream(filename);
160 // XERCES 1 or 2 additionnal classes.
161         OutputFormat of = new OutputFormat("XML", "UTF-8", false);
162         XMLSerializer serializer = new XMLSerializer(fos, of);
163 // SAX2.0 ContentHandler.
164         ContentHandler hd = serializer.asContentHandler();
165         hd.startDocument();
166 // ptcs attributes - vpodstate atributy vsech elementu
167         AttributesImpl atts = new AttributesImpl();
168 // ptcs tag.
169         hd.startElement("", "", "ptcs", atts);
170 // dt tag.
171         hd.startElement("", "", "dt", atts);
172         hd.characters(toChArr(Main.dt), 0, toChArr(Main.dt).length);
173         hd.endElement("", "", "dt");
174 // scale_x tag.
175         hd.startElement("", "", "scale_x", atts);
176         hd.characters(toChArr(Main.scale_x), 0, toChArr(Main.scale_x).length);
177         hd.endElement("", "", "scale_x");
178 // scale_y tag.
179         hd.startElement("", "", "scale_y", atts);
180         hd.characters(toChArr(Main.scale_y), 0, toChArr(Main.scale_y).length);
181         hd.endElement("", "", "scale_x");
182 // bound tag.
183         hd.startElement("", "", "bound", atts);
```

```
184         hd.characters(Boolean.toString(Main.bound).toCharArray(), 0,
185                 Boolean.toString(Main.bound).length());
186         hd.endElement("", "", "bound");
187 // ptc tags.
188         String ptc = new String();
189         for (int i = 0; i < Main.particles.size(); i++) {
190             hd.startElement("", "", "ptc", atts);
191             ptc = Main.particles.get(i).tString();
192             hd.characters(ptc.toCharArray(), 0, ptc.length());
193             hd.endElement("", "", "ptc");
194         }
195         hd.endElement("", "", "ptcs");
196         hd.endDocument();
197         fos.close();
198     }
199 
200     private static char[] toChArr(double nr) {
201         return Double.toString(nr).toCharArray();
202     }
203 
204     public static void SaveToCSV(File file, ArrayList<String> buff)
205             throws FileNotFoundException, IOException {
206         if (file == null) {
207             throw new IllegalArgumentException("File should not be null.");
208         }
209         if (!file.exists()) {
210             throw new FileNotFoundException("File does not exist: " + file);
211         }
212         if (!file.isFile()) {
213             throw new IllegalArgumentException(
214                     "Should not be a directory: " + file);
215         }
216         if (!file.canWrite()) {
217             throw new IllegalArgumentException(
218                     "File cannot be written: " + file);
219         }
220 
```

```
221         //use buffering
222         Writer output = new BufferedWriter(new FileWriter(file));
223         try {
224             //FileWriter always assumes default encoding is OK!
225             for (int i = 0; i < buff.size(); i++) {
226                 output.write(buff.get(i)+"\n");
227             }
228         } finally {
229             output.close();
230         }
231     }
232 }
```

Příloha P VII: Zdrojový kód souboru Canvas.java

```
1  package gui;
2
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.Graphics2D;
6  import java.awt.image.BufferedImage;
7  import java.util.ArrayList;
8  import java.util.logging.Level;
9  import java.util.logging.Logger;
10 import javax.swing.JPanel;
11 import psim.Main;
12
13 /**
14  * Trida Canvas dedi JPanel, pretezuje vykreslovani za ucelem
15  * vykreslovani castic a vubec doublebufferingu
16  *
17  * @author Tomas Drimal
18  */
19 public class Canvas extends JPanel implements Runnable {
20
21     int x = 0;
22     int y = 0;
23     int x_old = 0;
24     int y_old = 0;
25     int Width = 0;
26     int Height = 0;
27     long cyklyOld = 0;
28     ArrayList<Color> barvy = new ArrayList<Color>();
29     BufferedImage imgTraces;
30     Graphics2D g2D;
31
32     public Canvas() {
33         //pole barev castic
34         barvy.add(Color.red);
35         barvy.add(Color.white);
```

```
36          barvy.add(Color.blue);
37          barvy.add(Color.yellow);
38          barvy.add(Color.gray);
39          barvy.add(Color.green);
40          barvy.add(Color.pink);
41          barvy.add(Color.orange);
42          barvy.add(Color.magenta);
43          barvy.add(Color.cyan);
44          init();
45
46          // vybarveni obrazku bílou barvou
47          g2D.setColor(Color.white);
48          g2D.fillRect(0, 0, imgTraces.getWidth(), imgTraces.getHeight());
49      }
50
51      @Override
52      public void paint(Graphics g) {
53          //vymazani platna
54          g.setColor(Color.white);
55          g.clearRect(0, 0, getWidth(), getHeight());
56
57          //cyklus for, ktery vypocita x a y polohy stredu castic
58          for (int i = 0; i < Main.particles.size(); i++) {
59
60              //x-ova a y-ova souradnice castice (dano stredem)
61              x = (int) (Main.particles.get(i).x / Main.scale_x
62                      * psim.Main.CLIENT_WIDTH_PX);
63              y = (int) (Main.particles.get(i).y / Main.scale_y
64                      * psim.Main.CLIENT_HEIGHT_PX);
65
66              psim.Main.PtcsHist.get(i)[3] = x;
67              psim.Main.PtcsHist.get(i)[4] = y;
68          }
69          //cyklus, ktery vykresluje historii drah castic
70          for (int i = 0; i < Main.particles.size(); i++) {
71              g2D.setColor(barvy.get((int) Main.particles.get(i).col));
72              g2D.drawLine(Main.PtcsHist.get(i)[0], Main.PtcsHist.get(i)[1],
```

```
73                      Main.PtcsHist.get(i)[3], Main.PtcsHist.get(i)[4]);
74              }
75          // jestli je zadano vymazani drah castic
76          if (Main.clrTraces) {
77              g2D.setColor(Color.white);
78              g2D.fillRect(0, 0, imgTraces.getWidth(), imgTraces.getHeight());
79              Main.clrTraces = false;
80          }
81          // jestli se maji zobrazit drahy castic
82          if (Main.VTraces) {
83              g.drawImage(imgTraces, 0, 0, this);
84          }
85  //        Cyklus vykreslujici castice
86          for (int i = 0; i < Main.particles.size(); i++) {
87              //nastaveni barvy castice
88              g.setColor(barvy.get((int) Main.particles.get(i).col));
89
90              //samotne vykresleni
91              g.fillOval(psim.Main.PtcsHist.get(i)[3] - psim.Main.PtcsHist.get(i)[2],
92                      psim.Main.PtcsHist.get(i)[4] - psim.Main.PtcsHist.get(i)[2],
93                      (int) Main.particles.get(i).d,
94                      (int) Main.particles.get(i).d);
95              psim.Main.PtcsHist.get(i)[0] = psim.Main.PtcsHist.get(i)[3];
96              psim.Main.PtcsHist.get(i)[1] = psim.Main.PtcsHist.get(i)[4];
97          }
98          //získání poctu cyklu za ms
99          gui.Gui.lCpsPMs.setText(Long.toString((Main.cykly - cyklyOld) / 50) + " cpm
100         cyklyOld = Main.cykly;
101     }
102
103     @SuppressWarnings("static-access")
104     public void run() {
105         while (true) {
106             //prekresleni castic
107             if (Main.simulation) {
108                 this.repaint();
109             }
```

```
110             try {
111                 //uspani vlakna prekreslovani na 50 ms,
112                 //aby se prekreslovalo 24/s
113                 Thread.sleep(50);
114             } catch (InterruptedException ex) {
115                 Logger.getLogger(Canvas.class.getName()).
116                         log(Level.SEVERE, null, ex);
117                 System.out.println("Nepodarilo se uspat");
118             }
119 
120         }
121     }
122     /**
123      * Inicializacni metoda, vola se pri vytvareni a
124      * pri zmene velikosti platna
125      */
126     public void init() {
127         this.imgTraces = new BufferedImage(psim.Main.CLIENT_WIDTH_PX,
128                 psim.Main.CLIENT_HEIGHT_PX, BufferedImage.TYPE_INT_RGB);
129         this.g2D = imgTraces.createGraphics();
130         Main.clrTraces = true;
131     }
132 }
133 
```

Příloha P VIII: Obsah přiloženého CD