# Úvod

Jako zadání semestrální práce jsem dostal naprogramovat vizualizaci simulace vesmíru s planetami. Práce je rozdělena na tři části, které budu postupně plnit spolu s touto dokumentací. Toto není finální verze mé práce.

# Práce v první části

Snažil jsem se od začátku dělit program tak, aby v každé třídě byly ty části kódu, které spolu souvisí. Nemělo by se tedy stát, že by se v třídě, která má na starosti vykreslovat plátno, počítaly fyzikální zákony pro pohyb planet. Na první pohled program obsahuje více souborů. V této dokumentaci chci popsat, jak jednotlivé soubory pracují a tvoří dohromady simulaci vesmíru.

## DoubleVector2D.java

Jelikož budu pracovat s hmotnými body, udělal jsem si třídu, která dokáže uložit pozici x a y bodu. Třída je jednoduchá přepravka s getry (bez setrů) a metodou toString(). Body v přepravce jsem neměl potřebu měnit a když jsem chtěl změnit pozici planety, jednoduše jsem si vytvořil novou instanci DoubleVector2D. Specialita co tato třída má, je metoda computeDistance(). Ta vypočítá vzdálenost od jiného bodu třídy DoubleVector2D.

## Planeta.java

Název je poněkud zavádějící, nejedná se nutně o planetu ale o vesmírné těleso. Později z této třídy udělám abstraktní třídu a vytvořím jí potomky Planeta, Kometa a Raketa. Zatím v tom nevidím smysl a budu všechno nazývat planetou.

Každá planeta je charakteristická svým názvem, typem (Planeta, Raketa, Kometa), pozicí středu, rychlostí a hmotností. Ve třídě jsou getry, setry a toString(). Dále bude potřeba atribut zrychlení a poloměru. Poloměr se vypočítá z hmotnosti a hustoty, kterou mají všechny planety 1 kg/m3. Zrychlení bude nastavovat simulace. Jelikož pozice x, y představuje pozici středu planety, udělal jsem si speciální getry, které vrací střed změněný o poloměr planety. To mi usnadňuje práci při kreslení, jelikož elipsy se kreslí ne od středu, ale od levého horního rohu.

## VstupDat.java

Na vytvoření této třídy jsem použil znalosti z předmětu OOP. Udělal jsem ze třídy jedináčka, který umí číst soubor. Předpokládám, že data v souboru mají strukturu stejnou jako popisuje zadání, tj. na první řádce dvě čísla a poté n řádků s daty planet. Všechny údaje na řádce jsou odděleny čárkou. Jinou strukturu by třída nepřečetla. Metoda nactiData() nejprve hledá dvě čísla, gravitační konstantu G a časový skok, až poté načítá libovolný počet planet. Konstanty a planety si k sobě uloží do atributů, aby k datům měli přístup ostatní třídy.

## Galaxy\_SP2022.java

Hlavní třída programu. Načte data, spustí simulaci, vizualizaci a časovač. Zároveň se stará o vstup z klávesnice (pauza mezerníkem) a obsluhuje kliknutí myši na planetu.

## SimulationTimer.java

Časovač celého vesmíru. Předává simulaci čas a překresluje plátno. Důležitým parametrem pro simulaci je casovySkok ze vstupního souboru. Ten říká, kolik sekund simulace uběhne za jednu sekundu reálného času. Třída také umí simulaci pozastavit a poté na ni správně navázat. Časovač zařizuje, že se každých 20 ms počítají nové pozice planet a také se vykreslí.

## Simulace.java

Třída pracuje s konstantou G a časovým skokem ze souboru. Simuluje pohyb planet pomocí N-objektů. Metoda updateAllAcceleration() nastaví všem planetám najednou zrychlení podle vzorce. Samotný pohyb planet zařizuje metoda updateSystem(). Její výpočet je náročný a provádí se často. Pokud simulujeme mnoho planet, počítač nemusí stihnout dokončit počítání do nového volání metody a tím zpomalí celou simulaci. Z tohoto důvodu jsem se rozhodl, že přesnost simulace se bude odvíjet od počtu planet. Čím méně bude planet, tím více se provede výpočtů a tím přesnější simulace bude. Naopak čím více bude planet, tím se provede méně výpočtů a simulace bude nepřesná.

## Vizualizace.java

Správce plátna, vykresluje vše co vidíme. Aby byla co nejvíce využita daná plocha, při každém volání repaint() se musí vypočítat okraje vesmíru. Všechny planety tak mohou být vidět najednou a maximálně využijí plochu okna. Okraj vesmíru je vždy ta nejmenší nebo největší pozice hrany elipsy. Když jsem měl výpočet vesmíru hotový, musel jsem ho ještě aplikovat podle tohoto postupu:

1. Vycentrovat vizualizaci na střed okna
2. Aplikovat škálování na osu x a y
3. Vycentrovat vizualizaci na střed vesmíru
4. Odebrat minima vesmíru v osách x, y

Po této aplikaci se všechny nakreslené tvary budou řídit těmito pravidly. Nezbývá nic jiného než rozhýbat planety.

Jelikož chci ještě na obrazovku vypisovat simulační čas a informace o vybrané planetě, uložil jsem si transformaci původního okna. Aby byl text na popředí, volám jeho vykreslení jako úplně poslední.

Při vykreslování planet, které jsou velmi vzdálené, je jejich poloměr tak malý, že nejsou vidět. Proto jsem vytvořil metodu checkIsSomeSmall(), která zajistí, aby byla nějaká hranice jak malé planety mohou být. Metoda projde všechny planety v seznamu a pokud je jedna z nich menší než nastavené minimum, budou se vykreslovat všechny planety podle minimální velikosti.

Dalším úkolem vizualizace je vyhodnocení HitTestu myši. Vyhodnocení provádí metoda getHitPlanet(). K tomu si uložím výše vypsaná pravidla a udělám jejich inverzi. Když kliknu myší do okna, souřadnice kliku se převede na skutečnou velikost planet. V této soustavě vytvářím nové planety a zjišťuji, jestli se v nějaké náhodou nenachází souřadnice kliknutí. S touto metodou jsou ale problémy. Zjistil jsem, že pokud je operační systém uživatele nastavený tak, aby zvětšoval velikost okna, transformace se posouvají a HitTest je prakticky nepoužitelný. Zatím mě nenapadá jiný způsob, jak bych mohl HitTest vyřešit. Pokud jsme klikli na planetu, zobrazí se nám její informace a planeta se překreslí šedým kruhem. Zatím nastavuji atribut selectedPlanet na instanci kliknuté planety a na černý kruh kreslím další šedý. Tuto metodu chci ještě předělat, aby se planeta vykreslila jinou barvou.

# Vyhodnocení první části

Na práci jsem využil znalosti nejen ze cvičení, ale také z ostatních předmětů, za což jsem rád. Rozhodně vidím místa, kde bych mohl program vylepšit. Ať už optimalizovat výkon, tak vylepšit čitelnost kódu. Přesto jsem s prací v první části spokojen a nezbývá mi než doufat, že ji nebudu muset předělávat.

# Práce v druhé části

V druhé části jsem se zaměřil na zlepšení jak čitelnosti, tak funkčnosti dosavadního programu. Při implementaci nových funkcí jsem se držel zachování dělení programu do tříd tak, aby spolu souviseli.

## Úpravy první části

Zásadní změna, kterou jsem provedl vzhledem k první části, byla reprezentace dat ve vizualizaci. Namísto využití seznamu ArrayList<Planeta> a tvorby nových elips kde bylo zrovna potřeba, jsem si uložil všechny planety do mapy HashMap<Planeta, Ellipse2D>. Tato změna zpřehlednila kód a zjednodušila možnosti vykreslení.

Dále jsem upravil přepnutí minimální velikosti planet v metodě checkIsSomeSmall(). Nyní metoda funguje tak, že nastaví minimální velikost planety pouze těm, které by se vykreslili menší než tato hranice. Na plátně tedy může být jedna obrovská planeta a druhá malá.

Další změny nejsou tak významné, jedná se především o bugfixy.

## Kolize

Na vyhodnocení kolize jsem přidal do třídy Planeta.java metodu intersectWith(). V simulaci procházím všechny planety a volám tuto metodu. Při kolizi vytvářím novou planetu se zachovanou hmotností a hybností kolidovaných planet. Střed umístím blíže ke středu větší podle poloměrů obou planet. Nová planeta přepíše v seznamu planet větší, menší ze seznamu odstraním. Jméno jí nastavím na C(vetsiPlaneta\_mensiPlaneta), aby bylo zřejmé, že je vytvořená kolizí.

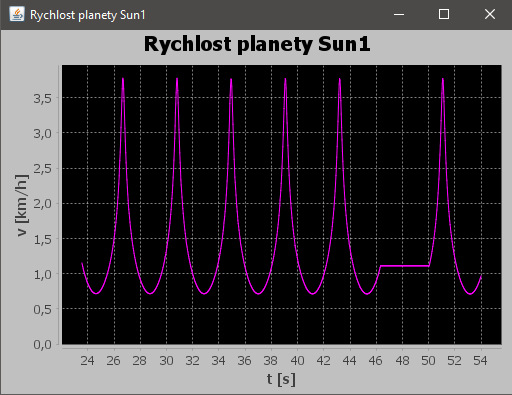
Ve vizualizaci jsem upravil zobrazení informací o planetě tak, aby se po kolizi přepnul na nově vzniklou.

## Graf

Rozhodl jsem se pro možnost zobrazení rychlosti za posledních 30 reálných sekund. Graf se zobrazí po kliknutí na planetu pravým tlačítkem myši. Jelikož kliknutí se provádí metodou getHitPlanet(), kterou se mi nepodařilo nahradit, pořád platí, že okna operačního systému nesmí být zvětšeny. Grafy jsou tvořeny pomocí knihovny jfreechart.

K uložení dat jsem přidal do třídy Planeta.java novou mapu TreeMap<Long, OldPlanetTraits>. Do mapy se ukládá čas od spuštění simulace v ms a přepravka OldPlanetTraits obsahující rychlost v km/h a pozici. Aby nedošlo k přetečení paměti, data starší jak 30 sekund se mažou. Při zobrazení nového grafu se načtou hodnoty rychlosti z této mapy spolu s časem. Poté se spustí časovač, který se stará o to, aby do grafu přidával nová data a ubíral ty stará.

Umožnil jsem spuštění grafů pro každou planetu zvlášť. Množina HashSet<Planeta> má v sobě uložené všechny právě spuštěné grafy, můžeme tedy zobrazit několik grafů rychlosti najednou a zároveň nevznikají duplikáty.

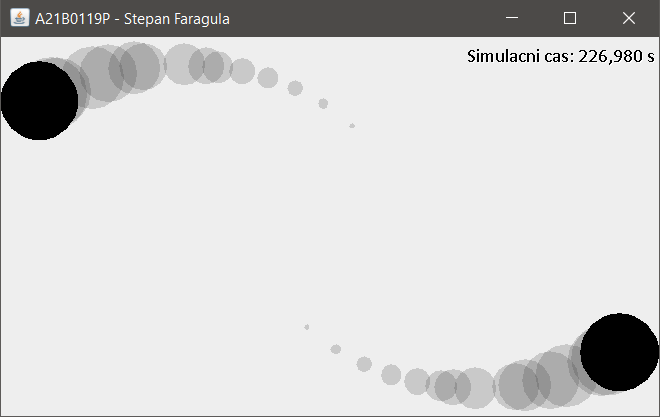
Při zastavení simulace graf zachovává rychlost.

Graf planety s pozastavenou simulací mezi 46. a 50. sekundou

## Trajektorie

Na vykreslení trajektorií jsem opět použil mapu s přepravkou OldMapTraits. Z této mapy jsem vyfiltroval záznamy pozic za poslední sekundu do nového seznamu. Jelikož jsem získal veliké množství záznamů (~ 50 pozic za vteřinu), zredukoval jsem tento počet ještě na polovinu.

Do vizualizace jsem přidal metodu drawTrajectory(), která přečte vyfiltrovaný seznam a vytvoří elipsy trajektorií. Elipsy vykresluji od největší po nejmenší podle času vzniku.



Výsledek vizualizace z druhé části

# Vyhodnocení druhé části

Vzhledem k úkolům z první části jsem měl tuto část poměrně rychle hotovou. Víceméně jsem vždy implementoval první věc, která mě napadla, a ta opravdu fungovala. Přesto byla práce náročná. Při programování jsem se zasekl hlavně kvůli řešení různých typů vyjímek.