Simulace Sluneční soustavy

Dokumentace k zápočtovému programu pro předmět

Programování I(NPRG030)

Jan Waltl

23. února 2017

Obsah

Organizace dokumentu

Tento text je organizován do následujících částí:

Úvod - První kapitola, zde je uvedena specifikace zápočtového programu a také teoretický popis problému.

Programátorká část - Kapitoly č.2 až 8, které popisují jak návrh celého programu, tak jednotlivých částí. Zaměřují se na použité algoritmy a jejich implementaci včetně ukázek zdrojových kódů v C++. Na konci jsou poté zmíněny možnosti dalšího rozšíření.

Kapitola 9 nabízí srovnání použitých algoritmů pro simulaci.

Kapitoly 10 a 11 popisují jak s programem pracovat z neprogramátorského pohledu a také jak ho zkompilovat.

Programátorská část vyžaduje určitou znalost C++ a OpenGL. Důležité koncepty a algoritmy jsou podrobně vysvětleny a vhodně doplněny zdrojovými kódy, které jsou psány následujícím formátem:

Název vystihující příklad

```
#include <iostream>

int main()
{
    std::cout<<"Hello World!\n";
    return 0;
}
```

Z praktických důvodů nemusí být tyto ukázky zkompilovatelné, popřípadě mohou být psány z části v pseudo-kódu, nejedná se tedy přímo o zdrojové kódy samotného programu. Primární účel je popisný, kde snaha je ilustrovat koncepty a algoritmy, které se v programu v nějaké formě skutečně vyskytují.

Zdroje a Reference

Můj hlavní zdroj informací byl předmět NOFY056 Programování pro fyziky 2015/2016 vedený RNDr. Ladislavem Hanykem, Ph.D. a cvičení s Mgr. Tomášem Ledvinkou, Ph.D., kde se vyučují mimo jiné právě numerické metody řešení ODR.

Zvlášť užitečné byly poznámky z `http://geo.mff.cuni.cz/~hanyk/NOFY056/index.htm` k výše zmíněné přednášce. Hlavně pro pochopení RK4 a také jsou zde přehledně napsány vícekrokové metody prediktor-korektor. Každopádně ale i anglická verze Wikipedie má celkem dobře sepsané všechny zde použité numerické metody.

Kapitola 1

Úvod

1.1 Specifikace

Program simuluje Sluneční soustavu za využití Newtonova gravitačního zákona a numerických metod. Fyzikálně se jedná řešení problému n-těles - tzn. každé těleso gravitačně působí na všechna ostatní. Tento problém je velmi těžko řešitelný analytickým metodami pro větší n. Výpočetní síla počítačů a numerické metody tak nabízí alternativní řešení tohoto problému.

Vstupem programu jsou strukturovaná data uložená v textovém souboru, která definují fyzikální vlastnosti simulované soustavy, což případně dovoluje snadné změny v zadání.

Výstup je 2D grafická reprezentace simulované soustavy v reálném čase. Uživatelské rozhraní dovoluje plynule měnit rychlost a přesnost simulace. Dále zobrazuje užitečné informace, jako jsou aktuální pozice a rychlosti pro každý simulovaný objekt vzhledem k jiným objektům. Celou simulaci je také možné nahrát a poté kdykoliv přehrát pomocí zabudovaného přehrávače.

Při vývoji programu byl kladen co největší důraz na pozdější rozšiřitelnost. Výsledný program tedy poskytuje několik simulačních metod a lze lehce rozšířit o další metody, popřípadě vstupy/výstupy.

1.2 Teorie

1.2.1 Analytický popis

Newtonův Gravitační zákon (NGZ) popisuje vzájemné silové působení F_g dvou hmotných bodů - Myšlené těleso, kde jeho veškerá hmotnost je soustředěna do jednoho místa. Kde výsledná síla je přitažlivá a m_1, m_2 jsou hmotnosti obou bodů a $\boldsymbol{x_1}, \boldsymbol{x_2}$ jejich polohy.

$$F_g = \kappa \frac{m_1 m_2}{(\boldsymbol{x_1} - \boldsymbol{x_2})^2} \tag{1.1}$$

Simulované objekty(hvězdy, planety, měsíce...) budeme dále pokládat za hmotné body, což je vzhledem k jejich rozměrům a vzdálenostem mezi nimi rozumná aproximace.

Nyní nám NGZ spolu s principem superpozice [1] a Newtonovým Zákonem síly (1.2) dává

[1] Princip superpozice říká, že výsledné silové účinky na těleso jsou dány součtem všech sil, které na něj působí.

pro n těles následující soustavu (1.3) n obyčejných diferenciálních rovnic. Kde neznámé $\boldsymbol{x}_i, \ddot{\boldsymbol{x}}_i$ jsou vektory polohy, resp. zrychlení simulovaných těles. Mínus je zde kvůli přitažlivosti výsledné síly.

$$F = m\ddot{\boldsymbol{x}} \tag{1.2}$$

$$\ddot{\boldsymbol{x}}_i(t) = -\kappa \sum_{j=1, j\neq i}^{n} \frac{m_i \left(\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right)}{\left(\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right)^3}. \quad \text{pro } i = 1 \ldots n \tag{1.3}$$

Analytické řešení této soustavy rovnic by nám dalo možnost zjistit polohu, rychlost a zrychlení libovolného simulovaného objektu v libovolném čase na základě počátečních podmínek. Ovšem vyřešit tuto soustavu je pro $n => 3$ velmi těžké.

1.2.2 Numerické řešení

Pokud se soustava nedá vyřešit analyticky, můžeme se alespoň pokusit získat aproximativní řešení. Numerické metody využívají toho, že nemusí být tak těžké spočítat derivace v libovolném čase. Navíc to dnešní počítače dokáží udělat velmi rychle. Pokud tedy dokážeme zjistit derivaci v každém bodě, tak bychom mohli původní funkci zrekonstruovat pomocí těchto derivací. Např. můžeme výslednou funkci aproximovat úsečkami, kde jejich směrnice je derivace hledané funkce. Toto přesně dělá nejjednodušší numerická metoda - Eulerova explicitní metoda(1.4). Mějme jednoduchou soustavu rovnic (1.5) . Je vidět, že řešením je funkce $y = e^t$, což se případně snadno ověří zpětnou derivací.

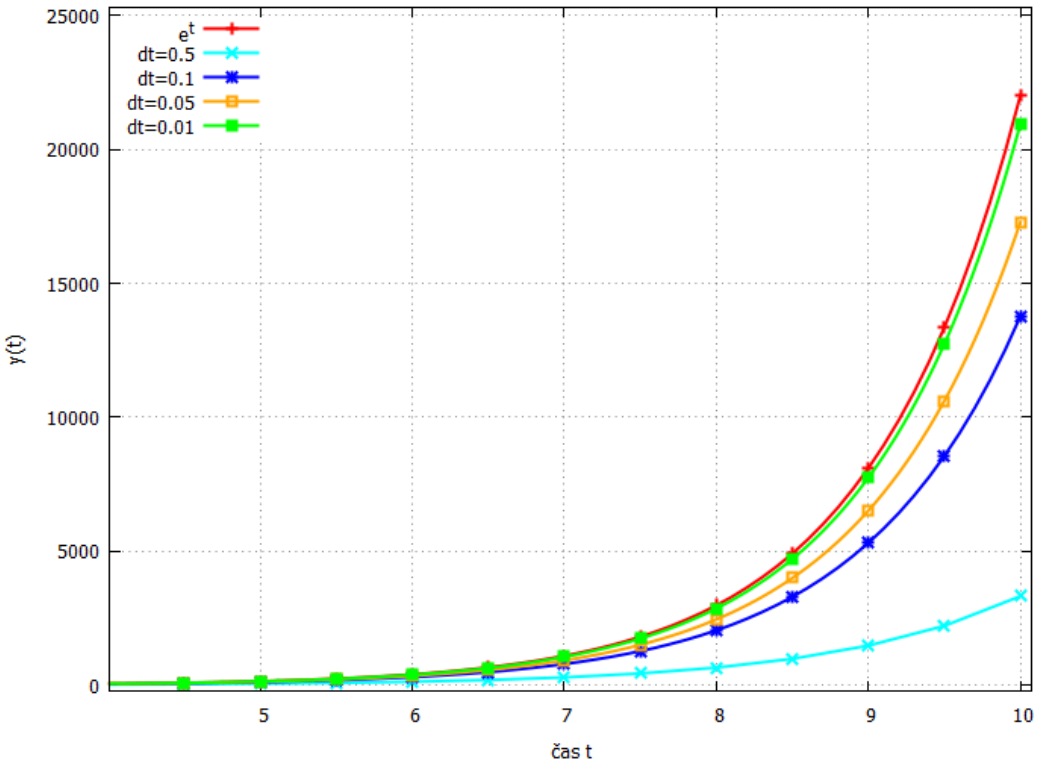
$$y(t + \Delta t) = y(t) + \Delta t.y'(t) \tag{1.4}$$

$$\dot{y} = y, y(0) = 1 \tag{1.5}$$

Zkusme tuto rovnici vyřešit numericky Eulerovou metodou (1.4). Výsledné hodnoty pro různé Δt jsou vykresleny na obrázku 1.1 spolu s analytickým řešením. Pro dostatečně malá Δt dostáváme relativně přesné řešení.

Eulerova metoda se dá použít i k řešení naší soustavy (1.3), což je podrobněji popsáno u její implementace v sekci 6.1. Také další metody - semi-implicitní Euler a RK4 jsou popsány u svých implementací 6.2 resp. 6.3. Kapitola 9 pak nabízí srovnání těchto metod pro řešení naší soustavy.

Obrázek 1.1: Srovnání různých Δt pro přesnost Eulerovy metody



Kapitola 2

Návrh programu

Tato kapitola se zabývá návrhem programu - jak je celý program navržen a popisuje jeho hlavní třídu - `Simulation`. V rámci které si představíme **moduly**, které se objevují v dalších kapitolách. Na úvod se podívejme jak by mohl vypadat jednoduchý **main.cpp**

Zdrojový kód 2.1: Hlavní program

```
#include <iostream>
#include <chrono>
#include "Simulation.h"

int main()
{
    //Celý program je zabalen do tohoto namespace
    using namespace solar;
    // Zpřístupňuje jednotky u čísel - např. 10s
    using namespace std::chrono_literals;
    try
    {
        auto parser = std::make_unique<FormattedFileParser>("vstup.txt");
        auto viewer = std::make_unique<IMGuiViewer>(1200, 700, "Title");
        auto method = std::make_unique<RK4>();

        Simulation sim(std::move(parser), std::move(method), std::move(viewer));
        sim.Start(10ms, 100, 180'000, 300s);
    }
    catch (const Exception& e)//Simulace používá vyjímky
    {
        std::cout << "Simulation failed, reason:\n" << e.what();
    }
    return 0;
}
```

Veškeré zdrojové kódy programu jsou zabaleny do jmenného prostoru `solar`. `Simulation` je hlavní třída, která se stará o celkový průběh simulace. Dále jsou zde vidět tři **moduly**, které jsou předány simulaci. Jejich detailní popis přijde později.

`Simulation` poté zajišťuje jejich vzájemnou spolupráci. Metoda `Simulation::Start()` následně spustí samotnou simulaci podle předepsaných parametrů. Pokud někde nastane chyba, dojde k vyvolání výjimky v podobě třídy `solar::Exception`, popřípadě jiných výjimek z rodiny `std::exception`.

2.1 Třída Simulation

Je třída, která spojuje jednotlivé moduly do funkčního celku a zajišťuje průběh celé simulace, proto stojí za to se na ni podrobněji podívat. Nejprve se podíváme na veřejné rozhraní, které dovoluje simulaci ovládat. Poté nahlédneme i dovnitř abychom zjistili jak to celé funguje.

2.1.1 Veřejné rozhraní

Přibližně takto vypadají veřejné metody třídy `Simulation` :

Zdrojový kód 2.2: Veřejně rozhraní `Simulation`

```
class Simulation
{
public:

    Simulation(parser_p parser, simMethod_p simMethod, viewer_p viewer);
    void Start(stepTime_t dt, size_t rawMult = 1, size_t DTMult = 1,
               seconds maxSimTime = 0);
    void StartNotTimed(stepTime_t dt, size_t rawMult = 1, seconds maxRunTime = 0);

    //Nasledující jsou řídící funkce
    void StopSimulation();
    void PauseSimulation();
    void ResumeSimulation();
    void StepSimulation();
    bool IsPaused();
    bool IsRunnig();
    double GetDtime() const;
    void SetDTime(double newDT);
    double GetRunTime() const;
    const simulatedTime& GetSimTime() const;
    void SetSimTime(simulatedTime newSimTime);
    double GetFrameTime() const;
    size_t GetRawMultiplier() const;
    size_t GetDTMultiplier() const;
    void SetRawMultiplier(size_t newRawMult);
    void SetDTMultiplier(size_t newDTMult);
};
```

Řídící funkce jsou funkce, které řídí spuštěnou simulaci. Jsou volány z modulů, ne z hlavního programu. Většinou se jedná o jednoduché funkce na 1-2 řádky, proto zde nejsou jednotlivě popsány, ale z jejich názvů je zřejmé co dělají. Případný náhled do zdrojových kódů by to měl objasnit.

Konstruktor očekává 3 moduly, které se budou v simulaci používat, podrobněji se na ně podíváme za okamžik. Zde jim také předá odkaz na simulovaná data.

Metoda `Start()` provádí samotnou simulaci. V této metodě stráví program většinu času, popíšeme ji podrobně v sekci2.1.3 - vysvětluje její teoretický návrh a implementaci, což také objasní její parametry.

2.1.2 Moduly

Nyní si vysvětlíme co to **moduly** jsou a k čemu slouží. Zkusme se zaměřit co by měla každá simulace vlastně udělat.

1. **Načíst data**, bez nich není co simulovat. Což se dá popsat dvěma slovy a implementovat stovky způsoby. Takže by stálo za to, aby simulace uměla všechny.
2. **Simulovat data**. Z teorie už víme, že také neexistuje pouze jedna integrační metoda, takže bychom chtěli mít na výběr. Určitě totiž není moc šťastné řešení zvolit jednu metodu a doufat, že bude stačit na všechny simulace.
3. **Uložit data**. Také se chceme našimi výsledky pochlubit, ale kdyby nám je simulace pracně získala a hned zahodila, tak to půjde těžko. Ovšem stálo by za to umět je ukládat v různých formátech nebo třeba rovnou odeslat někam po internetu.
4. **Prohlížet data**. Představme si situaci: 3:38 ráno, naše značně vylepšená verze simulace právě doběhla. Přidali jsme do ní nově objevené asteroidy u kterých chceme spočítat trajektorie. Z hrůzou ale zjistíme, že Země není tam kde má být, dokonce tam není vůbec! Co se mohlo asi stát? To je dobrá otázka, proto by mohlo být dobré mít přístup k datům i během simulace a třeba si je někam průběžně ukládat.

Teď už víme, co by měla každá simulace zvládnout, ale je vidět, že toho má umět celkem hodně a ještě různými způsoby. Nejlepší tedy bude, aby se simulace(ve formě třídy `Simulation`) opravdu starala jen o to, aby tyto kroky proběhly, ale to jak přesně proběhnou přenecháme někomu jinému - **modulům**, které se vyskytují ve 3 druzích:

parser obstarává výrobu vstupních dat. Také na konci simulace může výsledná data uložit.

simMethod provádí simulaci dat např. pomocí numerické integrace.

viewer má přístup k datům za běhu simulace a může je např. zobrazovat na obrazovku.

Práci jsme rozdělili, simulace by to celé měla tedy organizovat, což se děje právě pomocí metody `Start()`, tak si ji pojďme představit.

2.1.3 Návrh metody `Start()`

Takto nějak by mohla vypadat na první pohled rozumná implementace:

Zdrojový kód 2.3: Návrh metody `Start()`

```
1  void Simulation::Start(/*Parametry simulace*/)
2  {
3      //Uložíme si parametry simulace
4
5      //Parser načte data
6      auto data = parser->Load();
7      //Možná si potřebují simMethod a viewer něco připravit ještě před simulací,
8      //ale potřebují k tomu už znát data - např. jejich velikost
9      simMethod->Prepare();
10     viewer->Prepare();
11
12     while(!konec)//Necháme simulaci běžet
13     {
14         simMethod->Step(); // Uděláme jeden krok simulace
```

```
15          viewer->ViewData();// Podíváme se na simulovaná data
16      }
17      //Po doběhnutí simulace případně uložíme výsledná data
18      parser->Save(data);
19  }
```

Tato implementace by byla plně funkční, ale možná ne úplně vhodná pro náš cíl. Rádi bychom totiž prováděli a hlavně zobrazovali simulaci v reálném čase.

Zde ale není žádné časování, `while` smyčka bude probíhat jak nejrychleji může, bez jakékoliv kontroly. Což není špatné, pokud chceme nechat program běžet a podívat se jen na výsledky, popřípadě mezivýsledky uložené někde v souborech. K tomu přesně slouží metoda `Simulation::StartNotTimed()`, která má přibližně výše uvedenou implementaci.

Časování - K dosažení našeho cíle bude potřeba nějakým způsobem svázat reálný čas s tím simulovaným. Upravíme tedy předchozí příklad 2.3 následovně:

Zdrojový kód 2.4: Metoda `Start()` s časováním

```
1   void Simulation::Start(time deltaT,/*další parametry*/)
2   {
3       //Beze změny
4
5       unsigned int acc = 0;
6       while (!konec)//Necháme simulaci běžet
7       {
8           //Akumulátor času
9           acc += LastFrameTime();
10          while (acc > deltaT)
11          {
12              simMethod->Step(deltaT); // Uděláme jeden krok simulace
13              acc -= deltaT;
14          }
15          viewer->ViewData();// Podíváme se na simulovaná data
16      }
17      //Beze změny
18  }
```

Pokud bychom chtěli opravdu simulaci v reálném čase, tak se v každém průběhu smyčkou se podíváme jak dlouho trvala předchozí smyčka. Tolik času musíme odsimulovat. Naivní implementace by byla předat přímo tento čas `acc` metodě, která se stará o simulaci. Tím bychom dostali nedeterministický algoritmus [1]. Neboť trvání poslední smyčky je ovlivněno např. aktuálním vytížením počítače, což určitě není předvídatelné. A bohužel při počítaní s desetinnými čísly dochází k zaokrouhlovacím chybám, takže výsledek nezávisí pouze na vstupních datech, ale i na postupu výpočtu.

Řešení [2] je naštěstí jednoduché, budeme odečítat pevnou hodnotu `deltaT`. A to tolikrát, abychom odsimulovali potřebný čas uložený v `acc`. Je pravda, že nakonci nemusí platit $acc = 0$, ale bude platit $acc \leq deltaT$. A vzhledem k tomu, že `acc` si zachovává hodnotu mezi průběhy smyčkou, tak se zbytek času neztrácí, ale použije se v dalším průchodu. Takto dostaneme deterministický algoritmus.

Navíc máme objasněn první parametr - **deltaT(dt)** - základní časový krok simulace, který odpovídá Δt z teorie. Čím menší, tím je simulace přesnější, ale dochází k více voláním

[1] Algoritmus, který nemusí nutně vracet stejný výsledek při opakovaném volání se stejnými vstupními hodnotami.

[2] Zdroj: `http://gafferongames.com/game-physics/fix-your-timestep/` (23. února 2017)

simulační metody, což může být potencionálně náročné na CPU. Pro nízké hodnoty by se klidně mohlo snadno stát, že simulace přestane stíhat, tzn. že simulovat čas `deltaT` bude reálně trvat déle než `deltaT`. Např. simulace `10ms` bude konstantně trvat `15ms`, v dalším průběhu smyčkou se tedy bude simulace snažit simulovat uběhlých `15ms`, což ale může trvat `22,5ms`. V dalším průběhu se tedy musí odsimulovat `22,5ms`...takový případ velmi rychle celý program odrovná. Proto je vhodné proměnnou `acc` omezit nějakou konstantou, např. `500ms`. Poté sice začne docházet ke zpožďování simulace, ale kontrolovaným způsobem.

Změna rychlosti - Nyní bude naše simulace fungovat zcela správně a v reálném čase. Takže máme hotovo? Skoro, naše simulace je sice plně funkční, ale jediné co umí je předpovídat přítomnost a ještě jen s omezenou přesností. To není moc užitečné. Oběh Země kolem Slunce bude skutečně trvat 1 rok a Neptun to zvládne za 165 let. Tolik času nejspíše nemáme, proto by stálo za to najít nějaký způsob jak simulaci urychlit. Existují dva způsoby jak to udělat:

1. Volat simulační metodu častěji. Například pro každý krok `deltaT` ji můžeme zavolat `rawMult`krát. Toto zrychlení jde na úkor výpočetního výkonu nutného k udržení této rychlosti, viz. předchozí odstavec.

2. Volat simulační metodu s jiným `deltaT`, konkrétně s jeho `DTMult` násobkem. Tohoto zrychlení dosáhneme na úkor přesnosti. Protože předáváme větší Δt do simulační metody, což vede k menší přesnosti.

Parametry `rawMult` a `DTMult` přesně odpovídají argumentům implementované verze metody `Simulation::Start()`.

Poslední parametr, který nebyl ještě vysvětlen je `maxSimTime`. Vzhledem k tomu, že volání funkce `Start()` může trvat velmi dlouho, tak je dobré nastavit horní limit, který garantuje přerušení smyčky po překročení zadaného času. `maxSimTime=0` znamená `maxSimTime=`∞ ,tzn. simulace se přeruší pouze zavoláním funkce `Simulation::StopSimulation()`, kterou ale mohou volat pouze moduly, neboť jiné objekty nejsou při simulaci volány. Popřípadě se přeruší vyvoláním nějaké vyjímky.

2.1.4 Finální implementace metody `Start()`

Poučeni z předchozí části upravíme naši rozpracovanou implementaci 2.4 na 2.5. Což je už velmi podobné skutečné metodě použité v programu. Která navíc umožňuje simulaci pozastavit, krokovat a také pomocí C++ knihovny `chrono` implementuje opravdové časování , které zde bylo uvedeno spíše koncepčně.

Zdrojový kód 2.5: Finální verze `Simulation::Start()`

```
1  void Simulation::Start(time deltaT, size_t rawMult, size_t DTMult, time
       maxSimTime)
2  {
3      // Připadné uložení parametrů
4
5      //Parser načte data
6      auto data = parser->Load();
7      //Možná si potřebují simMethod a viewer něco připravit ještě před simulací,
8      //ale potřebují k tomu už znát data - např. jejich velikost
9      simMethod->_Prepare();
10     viewer->_Prepare();
11
12     auto acc = 0;
```

```
13      //Simulace může být ukončena uplynutím zadaného času
14      while (!konec && ElapsedTime() < maxSimTime)
15      {
16          //Akumulátor času
17          acc += LastFrameTime();
18          while (acc > deltaT)
19          {
20              for (size_t i = 0; i < rawMult; i++)
21              {
22                  // Uděláme jeden krok simulace
23                  simMethod->Step(deltaT*DTMult);
24                  acc -= deltaT*DTMult;
25              }
26          }
27          viewer->ViewData();// Podíváme se na simulovaná data
28      }
29      //Po doběhnutí simulace případně uložíme výsledná data
30      parser->Save(data);
31  }
```

2.2 Výjimky

Existují tři druhy výjimek, které program může vyvolat.

1. Nejčastěji to bude výjimka v podobě třídy `Exception`, která dědí z `std::runtime_error` a tedy i `std::exception`. K této výjimce dojde zejména při chybě v otevírání/čtení souborů a inicializace knihoven, ale jedná se o hlavní třídu výjimek v tomto programu, takže je použita i v modulech pro hlášení chyb.

2. Na výjimku v podobě třídy `GLError` můžeme narazit při chybě týkající se OpenGL - nepodařená inicializace, nedostatek paměti a podobně.

3. Výjimky dědící z `std::exception` - program využívá standardní C++ knihovny, které můžou potencionálně také vyvolat výjímky.

Všechny tři druhy nabízí metodu `what()`, která vrátí krátký popis vyvolané výjimky.

Kapitola 3

Popis modulů

Následující kapitoly se zabývají moduly popsanými v sekci 2.1.2. Nejprve si v této kapitole podrobně představíme každý ze 3 druhů modulů. V dalších kapitolách se pak podíváme na konkrétní moduly, které byly implementovány, včetně toho jak se program dá pomocí nich rozšířit.

Ale ještě před představením modulů musíme udělat malou odbočku a definovat si pár tříd a pojmů, které moduly hojně využívají.

Třída `SystemUnit` je jednoduchá třída, ze které dědí veškeré moduly a která modulům zajišťuje spojení s rozhraním třídy `Simulation`.

Matematický aparát ve formě tříd `Vec2` a `Vec4` nabízí základní matematické operace s vektory jako je sčítání, odčítání a násobení skalárem. Také jsou k dispozici funkce `length()` a `lengthsq()`, které počítají délku vektoru respektive její druhou mocninu.

třída `Unit` je základní jednotka simulace. Jedná se o jeden simulovaný objekt, který má své vlastnosti - polohu, rychlost, hmotnost, jméno a barvu. Kde poslední dvě jsou dobrovolné a simulace se bez nich obejde, ale jsou zde kvůli zobrazování na obrazovku. Její definice je uvedena v následujícím zdrojovém kódu 3.1

Zdrojový kód 3.1: Definice struktury `Unit`

```
1  class Unit
2  {
3  public:
4      Unit(const Vec2& pos, const Vec2&vel, double mass,
5           const Vec47 color, const std::string& name)
6          :pos(pos), vel(vel), mass(mass), name(name), color(color)
7      {}
8
9      Vec2 pos, vel;
10     double mass;
11     Vec4 color;
12     std::string name;
13 };
```

typ `simData_t` je `std::vector<Unit>` Používá jako typ pro simulovaná data.

3.1 Parser

Parser se stará o výrobu vstupních dat, také má přístup k výsledným datům, která tak může uložit. Všechny třídy **parser** modulů musí dědit z abstraktní třídy `Parser`, jejíž přesná implementace je zde:

Zdrojový kód 3.2: Abstraktní třída `Parser`

```
1 class Parser : public SystemUnit
2 {
3 public:
4     //Načte a vrátí data
5     virtual simData_t Load() = 0;
6     //Případně uloží výsledná data
7     virtual void Save(const simData_t&) {};
8     virtual ~Parser() = default;
9 };
```

Hlavní funkce, kterou každý **parser** musí implementovat je `Load()` . Její úkol je libovolným způsobem načíst data a vrátit je ve formě `simData_t`. Dále může přepsat metodu `Save()`, která je zavolána na konci simulace s výslednými daty.

3.2 SimMethod

Simulační metoda by měla zajistit samotnou simulaci. To jakým způsobem bude měnit data záleží pouze na ní. Může tedy použít libovolnou integrační metodu, ale pokud si bude házet imaginární kostkou, tak to bude fungovat také, ikdyž informační hodnota takové simulace je přinejlepším sporná. Podívejme se na přesnou definici abstraktní třídy `SimMethod`, která dává základ všem simulačním metodám

Zdrojový kód 3.3: Abstraktní třída `SimMethod`

```
1
2 class SimMethod : public SystemUnit
3 {
4 public:
5     //Provede jeden krok simulace
6     virtual void operator()(double deltaT) = 0;
7     //Voláno před spuštěním simulace
8     virtual void Prepare() {};
9     virtual ~SimMethod() = default;
10 protected:
11     //Ukazatel na simulovaná data NENÍ validní v konstruktoru dědících tříd,
12     //ale až ve volání Prepare() a operator() (=po spojeni s třídou Simulation)
13     simData_t* data;
14 };
```

Všechny zděděné moduly musí implementovat alespoň metodu `operator()()`, která provede jeden krok simulace. Simulovaná data si drží pomocí svého ukazatele, který inicializuje `Simulation` ze svého konstruktoru. Pokud potřebuje inicializovat své proměnné v závislosti na vstupních datech, tak může přepsat metodu `Prepare()`. Tato funkce je volána před startem simulace, ale po načtení dat, takže k ním má metoda už přístup. Což se může hodit pro metody, které potřebují znát velikost dat a podle toho si vytvořit dočasné proměnné.

3.3 Viewer

Poslední typ modulu - **viewer** - má přistup k datům za běhu simulace a je reprezentován abstraktní třídou `Viewer` jejíž definice je zde:

Zdrojový kód 3.4: Abstraktní třída `Viewer`

```
1
2  class Viewer :public SystemUnit
3  {
4  public:
5      //Volána pří každém průchodu smyčkou
6      virtual void operator()() = 0;
7      //Příprava svých dat, pokud je potřeba
8      virtual void Prepare() {}
9      virtual ~Viewer() = default;
10 protected:
11     //Ukazatel na simulovaná data NENÍ validní v konstruktoru dědících tříd,
12     //ale až ve volání Prepare() a operator()
13     simData_t* data;
14 };
```

Každý **viewer** musí alespoň implementovat metodu `operator()()`, která je volána při každém průchodu smyčkou a má tedy vždy přístup k aktuálně simulovaným datům. Také, pokud potřebuje, může přepsat metodu `Prepare()`

Kapitola 4

FormattedFileParser

Tento **parser** načítá strukturovaná data z textového souboru.

Parser očekává základní SI jednotky - metry, sekundy, kilogramy. Ale výstupní `simData_t` obsahuje objekty se vzdáleností v AU($1,5.10^9$ m), časem v rocích(pozemské - 365 dní) a hmotností v násobcích hmotnosti Slunce($1,988435.10^{30}$ kg přesně). Důvod je ten, že hodnoty v těchto jednotkách jsou více normalizované a dochází k menším zaokrouhlovacím chybám.

4.1 Struktura dat

Začneme hned příkladem, takto by mohl vypadat správný vstupní soubor s daty:

```
1  { name<Sun>
2  color<1.0 0.88 0.0 1.0>
3  position<0.0 0.0>
4  velocity<0.0 0.0>
5  mass<1.98843e30> }
6
7  { name<Earth>
8  color<0.17 0.63 0.83 1.0>
9  position< 149.6e9 0.0>
10 velocity<0 29800.0>
11 mass<5.9736e24> }
12
13 { name<Moon>
14 color<0.2 0.2 0.2 1.0>
15 position< 150.0e9 0.0>
16 velocity<0 30822.0>
17 mass<7.3476e22> }
```

`FormattedFileParser` by z tohoto souboru vytvořil data obsahující 3 objekty pojmenované - Sun, Earth, Moon s určitými vlastnostmi.

Každý objekt je popsán uvnitř páru složených závorek {}. Všechny parametry objektu jsou dobrovolné, čili `{}` je validní bezejmenný objekt, který se nachází v klidu na pozici `(0,0)` a má nulovou hmotnost. Pokud se ale parametr objeví, musí mít správný formát, který je obecně `název<hodnoty>`. Následuje přesný výčet a formát parametrů:

name<jméno> - jméno objektu. Jsou dovoleny pouze znaky ASCII, čili bez diakritiky. Pokud není uvedeno, pak je prázdné.

color<R G B A> - barva objektu. Očekávají se 4 desetinná čísla v rozmezi od 0.0 do 1.0 oddělená mezerou představující barvu ve formátu RGBA. Pokud není uvedeno, pak je bíla.

velocity<X Y> - počáteční rychlost objektu. Očekává dvě desetinná čísla oddělený mezerou reprezentující rychlost ve vodorovném a svislém směru. Nesmí zde být napsány fyzikální jednotky - tzn. `velocity<10e2 m/s 8e3 m/s>` **není** validní vstup. Ale implicitně by hodnoty měly být v m/s . Pokud není uvedeno, pak je (0,0).

position<X Y> - počáteční pozice objektu. Identické s rychlostí, očekávají se hodnoty v metrech.

mass <hmotnost> - hmotnost objektu vyjádřená jedním desetinným číslem v kilogramech. Doporučení je zde také neuvádět jednotky, i když to bude fungovat, tak budou ignorovány. Pokud není uvedeno, pak je nulová hmotnost.

Dodatek k číslům - jsou povoleny jak celá, tak desetinná čísla. Je dovolena pouze desetinná tečka. Číslo 104.25 můžeme zapsat například takto: 104.25; $1.0425e2$. **Parser** ovšem **neumí** aritmetiku, takže následující **není** validní vstup: $10425/100$; $104+0.25$; $1.0425*10^2$.

`FormattedFileParser` je relativně shovívavý k formátování, takže následující příklad je ekvivalentní definice objektu `Sun` z předchozího příkladu.

```
1 { Main star of our Solar system has a name <Sun>.
2   It's color is usually <1.0 0.88 0.0 1.0 which means yellow in RGBA format.
3   The Sun accounts for 99% mass of our entire system,
4   which makes it nearly 330 thousand times heavier than Earth
5   or to put it in another way it weights < 1.98843e30 kilograms>. }
```

4.2 Detaily implementace

Takto vypadá deklarace konstruktoru:

```
FormattedFileParser(const std::string& inputFileName,const std::string& outputFileName="");
```

Parser tedy očekává vstupní a případně výstupní jméno souboru, **včetně** cesty a přípony k souboru. Samotné načítání a ukládání probíhá v metodách `Load()` respektive `Save()`.

Na vlastní implementaci není koncepčně nic extra zajímavého. `Load()` načte celý soubor do `std::string` ve kterém si pak vždy najde dvojici {}, ve které se pokusí najít názvy parametrů. Pokud nějaký najde, pak k němu ještě najde nejbližší dvojici <> ve které poté očekává správné hodnoty. Tento jednoduchý způsob zajišťuje výše ukázanou flexibilitu formátování. Všechno ostatní, co by se v dokumentu mohlo nacházet(například komentáře) prostě ignoruje.

`Save()` vytvoří soubor, pokud byl nějaký zadán, do kterého předaná data uloží. Což se děje podobným způsobem - každý parametr "zabalí"do správného formátu a převede do jednotek SI. Při nekorektním vstupu, nemožnosti otevřít vstupní soubor nebo vytvořit soubor výstupní vyvolají funkce výjimku `Exception`.

Kapitola 5

SolarParser

5.1 Popis

Tento jednoduchý **parser** nedělá nic jiného, než načte data o Sluneční soustavě, která má zabudována pevně ve své implementaci. Momentálně se jedná o následující objekty:

1. Hvězdy - Slunce
2. Planety - Merkur, Venuše, Země, Mars, Jupiter, Saturn, Uran, Neptun
3. Trpasličí planety - Pluto
4. Měsíce - Měsíc, Phobos, Deimos, Io, Europa, Ganymed, Callisto

Výstupní `simData_t` obsahuje objekty se vzdáleností v AU$(1, 5.10^9$ m), časem v rocích(pozemské - 365 dní) a hmotností v násobcích hmotnosti Slunce$(1, 988435.10^{30}$ kg přesně). Důvod je ten, že hodnoty v těchto jednotkách jsou více normalizované a dochází k menším zaokrouhlovacím chybám.

Také dokáže uložit výsledky simulace do formátovaného textového souboru stejného jako `FormattedFileParser`.

Kapitola 6

Implementované simulační metody

6.1 Explicitní Eulerova metoda

Teorie za touto metodou byla popsána v úvodní kapitole. V naší soustavě (1.3) ale máme i druhé derivace, proto si pomůžeme tím, že každá obyčejná diferenciální rovnice n-tého stupně se dá převést na n rovnic prvního stupně, takže nám vzniknou dvě rovnice (6.1) na které aplikujeme Eulerovu metod (1.4) a dostaneme numerické řešení v podobě (6.2)

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{v}(t) \tag{6.1a}$$

$$\dot{\boldsymbol{v}}(t) = -\Delta t.\kappa \sum_{j=1, j\neq i}^{n} \frac{m_i}{\left[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right]^3}.\left[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right] \tag{6.1b}$$

$$\boldsymbol{x}_i(t + \Delta t) = \boldsymbol{x}_i(t) + \boldsymbol{v}_i(t) \tag{6.2a}$$

$$\boldsymbol{v}_i(t + \Delta t) = \boldsymbol{v}_i(t) - \Delta t.\kappa \sum_{j=1, j\neq i}^{n} \frac{m_i}{\left[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right]^3}.\left[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right] \tag{6.2b}$$

Implementace této metody by mohla vypadat například jako 6.1. Ve dvojité smyčce tedy projdeme všechny dvojice a spočítáme nové rychlosti a poté ještě upravíme polohu. Potřebujeme si v dočasné proměnné uchovat původní rychlost a polohu v čase t. Netvrdím, že se jedná o nejefektivnější algoritmus, ale tvrdím, že na tom nezáleží, protože tato metoda není moc vhodná pro náš problém, protože vyžaduje velmi malé Δt a je vždy nestabilní - viz. kapitola 9.

Zdrojový kód 6.1: Explicitní Eulerova metoda

```
1
2  void ExplicitEuler::operator()(double step)
3  {
4
5      //Dočasná proměnná pro uložení rychlostí a poloh objektů
6      std::vector<VelPos> temps < -data->vel, data->pos;
```

```
7
8       //Projde všechny dvojice
9       for (size_t i = 0; i < data->size(); ++i)
10      {
11          auto& left = (*data)[i];
12
13          for (size_t j = i + 1; j < data->size(); ++j)
14          {
15              auto& right = (*data)[j];
16              auto distLR = dist(temps[i].pos, temps[j].pos);
17              distLR = distLR*distLR*distLR;//R^3
18
19              Vec2 dir = temps[i].pos - temps[j].pos;//Vzdálenost
20              Vec2 acc = -physicsUnits::G / distLR * dir;//Zrychlení bez hmotnosti
21              // v(t+dt) = v(t) + dt*a(t); - explicitní Euler
22              left.vel += step*acc*right.mass;// doplněné správné hmotnosti
23              right.vel -= step*acc*left.mass;// doplněné správné hmotnosti, opačný
        směr
24          }
25          // x(t+dt) = x(t) + dt*v(t); - explicitní Euler
26
27          left.pos += step*temps[i].vel;
28      }
29  }
```

6.2 SemiImplicitEuler

6.2.1 Teorie

V minulé části jsme se věnovali explictiní Eulerově metodě, ještě existuje implicitní verze (6.3), která je teoreticky shodná s explicitní až na to, že derivaci vyčíslíme v čase $t + \Delta t$.

$$y(t + \Delta t) = y(t) + \Delta t.y'(t + \Delta t) \tag{6.3}$$

Nyní se podívejme jak bychom řešili jednoduchou diferenciální rovnici (6.4), kterou si stejně jako v minulé části upravíme na dvě rovnice (6.5a) a (6.5b)

$$\ddot{x}(t) = k.x(t) \tag{6.4}$$

$$\text{s počátečními podmínkami:} \quad x(0) = x_0, \quad \dot{x}(0) = v_0$$

$$\dot{x}(t) = v(t) \tag{6.5a}$$

$$\dot{v}(t) = k.x(t) \tag{6.5b}$$

Tak zkusme vyřešit nejdříve (6.5b) a použijeme implicitní verzi. Tím dostaneme rovnici (6.6). Teď použijeme stejný postup i na (6.5a) a výsledkem je (6.7). Nyní stačí dosadit první rovnici do druhé a po úpravě dostaneme hledaný výsledek, pak ho ještě dosadíme zpět

do první a tím máme obě proměnné explicitně vyjádřené rovnicemi (6.8) a (6.9).

$$v(t+\Delta t) = v(t) + \Delta t.\dot{v}(t+\Delta t) = v(t) + \Delta t.k.x(t+\Delta t) \tag{6.6}$$

$$x(t+\Delta t) = x(t) + \Delta t.\dot{x}(t+\Delta t) = x(t) + \Delta t.v(t+\Delta t) \tag{6.7}$$

$$x(t+\Delta t) = x(t) + \Delta t.\left[v(t) + \Delta t.k.x(t+\Delta)\right]$$

$$x(t+\Delta t) - \Delta t^2.k.x(t+\Delta t) = x(t) + \Delta t.v(t)$$

$$x(t+\Delta t) = \frac{x(t) + \Delta t.v(t)}{1 - \Delta t^2 k} \tag{6.8}$$

$$v(t+\Delta t) = v(t) + \Delta t.k.\frac{x(t) + \Delta t.v(t)}{1 - \Delta t^2 k} = \frac{\Delta t.k.x(t) + v(t)}{1 - \Delta t^2 k} \tag{6.9}$$

To sice dalo určitou práci, ale dostali jsme správné řešení. Na pravé straně máme proměnné pouze v čase t takže je známe z předchozího kroku. V implementaci je nejspíše efektivnější spočítat jen jednu a druhou pomocí (6.6) nebo (6.7).

Naše soustava rovnic (1.3) je nápadně podobná předchozí rovnici (6.4), což samozřejmě není náhoda. Pokud se ale nyní budeme stejný postup snažit aplikovat na naší soustavu rovnic, tak zjistíme, že to nebude fungovat. Narazíme totiž na to, že po dosazení obou rovnic nebudeme schopni explicitně vyjádřit $x(t+\Delta t)$. Bohužel toto je daň za vyšší stabilitu implicitních metod - nutnost vyřešení další rovnice. V našem případě bychom museli použít numerické řešení i pro samotnou rovnici, což dělat nebudeme.

Místo toho použijeme semi-implicitní Eulerovu metodu. Místo dvojitého použití implicitní metody použijeme implicitní pro (6.11) a explicitní verzi pro (6.10) . Explicitní verze nám dovolí snadno spočítat $v(t+\Delta t)$ neboť hodnoty v čase t už známe. Tím jsme ale získali i potřebnou hodnotu $v(t+\Delta t)$ pro implicitní verzi druhé rovnice. Vlastně jsme z obou metod vzali to nejlepší - jednoduchost explicitní a větší stabilitu implicitní metody. Výsledný mix je metoda, která je jednoduchá na implementaci a relativně přesná pro naše účely. Srovnání explicitní, implicitní a semi-implicitních integračních metod se věnuje kapitola 9

$$v(t+\Delta t) = v(t) + \Delta t.\dot{v}(t) \tag{6.10}$$

$$x(t+\Delta t) = x(t) + \Delta t.\dot{x}(t+\Delta t) = x(t) + \Delta t.v(t+\Delta t) \tag{6.11}$$

Naše finální soustava (1.3) po použití této metody bude (6.12) pro $i = 1 \dots N$

$$\boldsymbol{v}_i(t+\Delta t) = \boldsymbol{v}_i(t) - \Delta t.\kappa \sum_{j=1, j\neq i}^{n} \frac{m_i}{\left[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right]^3}.\left[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right] \tag{6.12a}$$

$$\boldsymbol{x}_i(t+\Delta t) = \boldsymbol{x}_i(t) + \boldsymbol{v}_i(t+\Delta t) \tag{6.12b}$$

6.2.2 Implementace

Když jsme si metodu pracně teoreticky popsali, tak se nyní podívejme na to, jak bychom ji implementovali do našeho programu. Implementace by mohla vypadat například jako v 6.2

Zdrojový kód 6.2: Semi-implicitní Eulerova metoda

```
1 void SemiImplicitEuler::operator()(double step)
2 {
3     //Projdeme všechny dvojice
4     for (auto left = data->begin(); left != data->end(); ++left)
5     {
```

```
6         for (auto right = left + 1; right != data->end(); ++right)
7         {
8             auto distLR = dist(left->pos, right->pos);
9             distLR = distLR*distLR*distLR;
10
11            Vec2 dir = left->pos - right->pos;
12            Vec2 acc = - kappa / distLR * dir;//Bez hmotnosti
13            // v(t+dt) = v(t) + dt*acc(t) - explicitní
14            left->vel += step*acc*right->mass;// Přidáme správnou hmotnost
15            right->vel -= step*acc*left->mass;// a opačný směr
16
17        }
18        //x(t+dt) = x(t) + dt * v(t + dt); - implicitní
19        //XXX-> je nyní v čase t+dt
20        left->pos += step*left->vel;
21    }
22 }
```

Soustava (6.12) nám říká, že nejdříve musíme spočítat novou rychlost objektu, která ale záleží na polohách všech ostatních. Takže musíme projít všechny dvojice, což nám zajistí dvojitá `for` smyčka. Dále potřebuje sečíst sumu, což se děje právě ve vnitřní smyčce. Protože je silové působení symetrické a pouze opačného směru, tak to můžeme udělat vždy pro celou dvojici najednou. Vnitřní smyčka nám spočítala správnou rychlost levého(`left`) objektu. Můžeme tedy spočítat jeho novou polohu dle (6.12).

Důležité je ověřit, že opravdu počítáme správně veličiny a hlavně ve správný čas. A skutečně je to takto správně, protože pozice levého objektu už v dalších smyčkách není použita a zároveň vnitřní smyčka opravdu správně spočítala novou rychlost levého objektu, kde pozice pravých objektů ještě upraveny nebyly a jsou tedy v čase t.

6.3 RK4

6.3.1 Teorie

V předchozí sekci jsme implementovali Eulerovu metodu. Tato metoda lokálně aproximovala hledanou funkci pomocí úseček. Což znamenalo, že jsme na intervalu $[t, t + \Delta t]$ považovali derivaci za konstantu, a to samozřejmě nemusí být pravda a v našem případě ani není. Proto se podívejme na další metodu - **Runge-Kutta čtvrtého řádu(RK4)** . Která počítá derivaci vícekrát v různých bodech časového intervalu $[t, t + \Delta t]$ a poté provede vážený průměr ze kterého poté dopočítá novou hodnotu hledané funkce. Mějme rovnici (6.13), pak RK4 dává numerické řešení ve formě rovnice (6.14). Jedná se o explicitní verzi, ke které existuje ještě varianta implicitní, kterou ale implementovat nebudeme.

$$\dot{\boldsymbol{y}}(t) = \boldsymbol{f}(\boldsymbol{y}, t) \quad \boldsymbol{y}(0) = \boldsymbol{y_0} \tag{6.13}$$

$$\boldsymbol{y}(t + \Delta t) = \boldsymbol{y}(t) + \frac{\Delta t}{6}\left[\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4\right] \tag{6.14}$$

$$\boldsymbol{k}_1 = \boldsymbol{f}(\boldsymbol{y}(t), t)$$

$$\boldsymbol{k}_2 = \boldsymbol{f}(\boldsymbol{y}(t) + \Delta t\frac{\boldsymbol{k}_1}{2}, t + \frac{\Delta t}{2})$$

$$\boldsymbol{k}_3 = \boldsymbol{f}(\boldsymbol{y}(t) + \Delta t\frac{\boldsymbol{k}_2}{2}, t + \frac{\Delta t}{2})$$

$$\boldsymbol{k}_4 = \boldsymbol{f}(\boldsymbol{y}(t) + \Delta t.\boldsymbol{k}_3, t + \Delta t)$$

Zkusme tedy tuto metodu aplikovat na naši soustavu (1.3). Použijeme stejný trik jako u Eulerovy metody (6.1) a z jedné rovnice druhého řádu uděláme dvě rovnice první řádu (6.15).

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{v}(t) \tag{6.15a}$$

$$\dot{\boldsymbol{v}}(t) = -\kappa \sum_{j=1, j\neq i}^{n} \frac{m_i}{[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)]^3} . [\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)] \tag{6.15b}$$

Ve výše zmíněném popisu RK4 (6.13) se žádná soustava na první pohled nevyskytuje, v zadání je pouze jedna funkce, ale vektorová. Proto definujme následující vektorovou funkci (6.16). Pak vlastně řešíme rovnici (6.17) s neznámou funkcí $\boldsymbol{u}(t)$, na kterou přesně aplikujeme RK4.

$$\boldsymbol{u}(t) := (\boldsymbol{x}(t), \boldsymbol{v}(t)) \equiv (x_x, x_y, v_x, v_y)(t) \tag{6.16}$$

$$\dot{\boldsymbol{u}}(t) = (\dot{\boldsymbol{x}}(t), \dot{\boldsymbol{v}}(t)) = (\boldsymbol{v}(t), -\kappa \sum_{j=1, j\neq i}^{n} \frac{m_i \, [\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)]}{[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)]^3}) \tag{6.17}$$

6.3.2 Implementace

Nyní se budeme věnovat tomu, jak by se RK4 dalo zakomponovat do našeho programu. Implementace už je trochu delší, ale podrobně si ji vysvětlíme.

Zdrojový kód 6.3: Runge-Kutta integrační metoda

```
1
2  struct AccVel//Derivace u(t)
3  { Vec2 acc, vel; };
4  struct VelPos//u(t)
5  { Vec2 vel, pos; };
6
7  //Pomocná proměnná pro každý objekt,
8  //která ukládá dočasný stav pro výpočet koeficientů
9  std::vector<VelPos> temps;
10 //Koeficienty k1,k2,k3,k4 pro každý objekt
11 std::array<std::vector<AccVel>, 4> kXs;
12
13 //Inicializujeme temps a vynulujeme koeficienty pro první voláni funkce.
14
15 void RK4::operator()(double step)
16 {
17     //Vypočte koeficient
18     auto computeKx = [&](size_t x, double mult)
19     {
20         x = x - 1;//Číslování od jedničky -  k1,k2,k3,k4
21         //Projdeme všechny dvojice
22         for (size_t i = 0; i < data->size(); ++i)
23         {
24             auto& left = temps[i];
25             for (size_t j = i + 1; j < data->size(); ++j)
26             {
27                 auto& right = temps[j];
28                 auto distLR = dist(left.pos, right.pos);
29                 distLR = distLR*distLR*distLR;
30                 Vec2 dir = left.pos - right.pos;
```

```
31                  Vec2 acc = -physicsUnits::G / distLR * dir;
32
33                  //Spočítáme zrychlení
34                  //Které uložíme do koefficientů
35                  kXs[x][i].acc += acc*(*data)[j].mass;
36                  kXs[x][j].acc -= acc*(*data)[i].mass;
37              }
38              //Spočítáme rychlost
39              kXs[x][i].vel = left.vel;
40              //Posuneme dočasný stav pro výpočet dalšího koeficientu
41              left.vel = (*data)[i].vel + mult*step*kXs[x][i].acc;
42              left.pos = (*data)[i].pos + mult*step*kXs[x][i].vel;
43          }
44      };
45
46      computeKx(1, 0.5);
47      computeKx(2, 0.5);
48      computeKx(3, 1.0);
49      computeKx(4, 0.0);//Už nikam neposouváme
50
51      for (size_t i = 0; i < data->size(); ++i)
52      {
53          //provedeme krok podle RK4
54
55          (*data)[i].vel += step / 6.0 *(kXs[0][i].acc + 2 * kXs[1][i].acc + 2 *
    kXs[2][i].acc + kXs[3][i].acc);
56          (*data)[i].pos += step / 6.0 *(kXs[0][i].vel + 2 * kXs[1][i].vel + 2 *
    kXs[2][i].vel + kXs[3][i].vel);
57
58          //Načteneme nový stav do dočasných proměnných
59          temps[i] = {(*data)[i].vel,(*data)[i].pos};
60          //Vynulujeme koeficienty
61          kXs[0][i] = {Vec2(),Vec2()};
62          kXs[1][i] = {Vec2(),Vec2()};
63          kXs[2][i] = {Vec2(),Vec2()};
64          kXs[3][i] = {Vec2(),Vec2()};
65      }
66
67  }
```

Nejprve si všimneme, že naše soustava nezávisí přímo na čase, ale pouze na aktuální poloze a rychlosti. Budeme také už potřebovat nějaké dočasné proměnné, konkrétně si budeme ukládat všechny 4 koeficienty pro každý objekt. Dále si ještě vytvoříme proměnnou, kam budeme ukládat mezistavy kroku - `temps`. Neboť pro spočítání k_{i+1} potřebujeme znát k_i.

Nyní se zaměřme na funkci `computeKx`, která počítá koeficient. Z toho jak jsou koeficienty definované je musíme počítat postupně, protože do koeficientu k_i dosazujeme stav $y(t) + mult.\Delta t.k_{i-1}$. Výpočet koeficientu je vlastně jen jeden krok explicitní Eulerovy metody. Akorát neintegrujeme simulovaná data, ale právě jen dočasný stav. Také nepočítáme posun o Δt ale o jeho násobek - `mult` argument.

Explicitní metodu už máme vymyšlenou z minula, tomu také odpovídá implementace. Na konci nám ale ještě přibude finální integrace dat pomocí (6.14). A také musíme nastavit dočasné proměnné pro další volání funkce.

Kapitola 7

IMGuiViewer

Je grafický **viewer**, který zobrazuje simulované objekty ve 2D a dovoluje ovládání simulace pomocí uživatelského rozhraní. Jeho implementace je rozdělena do relativně velkého množství tříd a využívá externí knihovny, které si představíme jako první.

7.1 Třída `OpenGLBackend` a GLFW&GLEW

Tento program používá k vykreslování grafiky OpenGL, tato třída má na starosti jeho správnou inicializaci. Používá k tomu knihovnu **GLFW** a **GLEW**. GLFW[1] poskytuje funkce k vytvoření okna do kterého může poté OpenGL kreslit. GLEW[2] se stará o získání OpenGL funkcí, které se musí načíst za běhu aplikace z konkrétních grafických ovladačů na cílovém počítači. Přesná implementace je dostupná ve zdrojových kódech programu. Ale jedná se ve větší míře o přepsání doporučené implementace na stránkách obou knihoven.

7.1.1 Třída `Shader`

Slouží pro jednoduší vytváření OpenGL shaderů, tedy malých programů určených pro grafickou kartu, které říkají jak má interpretovat a kreslit předaná data.

7.1.2 Řešení OpenGL chyb

OpenGL samo nemá výjimky a chyby sděluje pomocí funkce `glGetError()`, která vrací typ chyby pokud nějaká nastala. Proto byla vytvořena funkce `ThrowOnError`, která zavolá `glGetError()` a při jakékoliv chybě vyvolá výjimku právě v podobě třídy `GLError` s informacemi o chybě.

7.1.3 Třída `ImGuiBackend` a ImGui

Se stará o integraci knihovny **ImGui**[3], která poskytuje nástroje k vytvoření uživatelského rozhraní. Její implementace byla také vytvořena na základě příkladů uvedených na stránkách knihovny a dokumentace v souborech `imgui.h` a `imgui.cpp`. Pojďme si v rychlosti představit

[1] http://www.glfw.org/
[2] http://glew.sourceforge.net/
[3] https://github.com/ocornut/imgui

nějaké funkce knihovny, abychom pak lépe pochopili její použití při tvorbě uživatelského rozhraní.

ImGui je psáno v C++ ale z vnějšího pohledu vypadá spíše jako C, nejsou zde žádné složité třídy a většina prvků je z vnějšku bezstavová až na pár výjimek a např. funkce na výpis textu používá syntaxi podobnou `printf()`.

Zaprvé je ještě nutné zmínit jak OpenGL funguje. Ve zjednodušeném pohledu vykresluje do svého framebufferu(2D plátno/textura) všechny objekty, které pak zobrazí na obrazovku. Toto plátno si můžeme a musíme mazat sami kdykoliv potřebujeme, pokud bychom ho nemazali, tak na něm zůstanou objekty z „minulého“ kreslení. [4] Takže si nejdříve vždy vyčistíme plátno a pak do něj vše nakreslíme a tento postup opakujeme ve smyčce.

ImGui má podobný systém, což je ukázáno na příkladu 7.1, na kterém si nyní vysvětlíme jak funguje.

Zdrojový kód 7.1: Příklad použití knihovny ImGui

```
while (true)
{
    //Všechna volání funkcí musí být mezi NewFrame a Render
    ImGui::NewFrame();

    //Umístí levý horní roh okna 100 pixelů od horního levého okraje obrazovky
    ImGui::SetNextWindowPos(ImVec2(100.0f, 100.0f);
    if (ImGui::Begin("Window"))
    {
        if (ImGui::Button("Button"))
            ImGui::Text("Foo");
        else
            ImGui::TextColored(ImVec4(1.0f, 0.0f, 0.0f, 1.0f), "Bar");
        ImGui::End();//Aby bylo vidět, kde končí okno a může začít další
    }

    //ImGui si poznamenává všechny vykreslené objekty
    //a v této funkci je předá OpenGL pro vykreslení
    ImGui::Render();
}
```

Vnější smyčku nám v programu už obstarává samotná simulace, takže o ní se starat nemusíme. Dále je zde dvojice funkcí `NewFrame()` a `Render()`, kde po zavolání `NewFrame()` si **ImGui** zaznamenává všechny další volání všech svých funkcí. `Render()` pak předá zaznamenaná data OpenGL, které je vykreslí na obrazovku.

Tento konkrétní příklad vykresluje okno s jedním tlačítkem, které po zmáčknutí zobrazí text. Všimněme si, že zde není žádný objekt - tlačíko, okno, ani text. Vše je pouze volaní funkcí, což je klíčové pro pochopení fungování knihovny. Také je to důvod pro `if`, protože „Foo“ text se zobrazí jen a **pouze při konkrétním průchodu smyčkou**, ve kterém došlo k zmáčknutí tlačítka (reprezentováno návratovou hodnotou). Ve výsledku text na obrazovce jen problikne, samozřejmě v tomto případě asi chceme spíše přepínací tlačítko, k čemuž slouží jiná funkce. Také se vše zobrazí jen a pouze pokud je okno otevřené, což je jedna z výjimek, kde si **ImGui** uchovává stav. Nic nám ovšem nebrání tuto funkci občas nezavolat, což bude znamenat, že se nám okno občas nevykreslí vůbec. Tento přístup dovoluje rychlý

[4] Například by zde zůstali minulé pozice všech objektů, což je velmi rychlý způsob jak implementovat 7.2.3, ale časem by to bylo velmi nepřehledné. Na druhou stranu to OpenGL nemůže dělat za nás, protože neví které volání kreslících funkcí ještě patří do „tohoto“ času a co už bylo „minule“.

vývoj a velkou dynamičnost uživatelského rozhraní. Ze zdrojového kódu je tedy hned patrné co, kdy a za jakých podmínek se bude kreslit.

Většina funkcí využívající **ImGui** by teď už měla být alespoň koncepčně jasná, pokud ne, tak knihovna je dobře okomentovaná a když tak není problém si najít volné místo v našem kódu, a některé funkce si jen tak vyzkoušet, což je díky bezstavové implementaci velmi snadné. Popřípadě na stránkách knihovny je předpřipravený projekt, ve kterém se dá kód rychle otestovat a také je tam interaktivní demo se zdrojovými kódy představující jednotlivé funkce.

7.2 Kreslení simulace

Nyní představíme pár tříd, které se starají o vykreslení samotných simulovaných dat

7.2.1 třída `OMSAR`

`OMSAR`(z Offset-Move-Scale-Aspect-Ratio) slouží k transformaci mezi souřadnou soustavou simulovaných dat a normalizovanými souřadnicemi, které využívá OpenGL. Jedná se o třídu ze které mají ostatní třídy zdědit sadu funkcí, která právě popsanou transformaci zajišťuje. Pomocí této třídy je také zajištěno posouvání obrazovky a přibližování/oddalování, což se děje pomocí dvou proměnných `Vec2 offset`(posouvání) a `scale`(přibližování). Tyto dvě proměnné využívají ostatní třídy, aby si správně posunuly vykreslované prvky na obrazovku.

7.2.2 třídy `CircleBuffer` a `SimDataDrawer`

První jmenovaná třída umí vykreslit kruh o zadaném poloměru do středu obrazovky. K tomu využívá prvky OpenGL - konkrétně **buffery**. Pro posouvání se musí použít správný **shader**.

Druhá jmenovaná třída vykresluje data, kde jednomu objektu odpovídá právě jeden `CircleBuffer` - kruh. Data vykresluje pomocí funkce `Draw()`, která zajišťuje správný **shader** a využívá právě `OMSAR` k určení polohy jednotlivých kruhů.

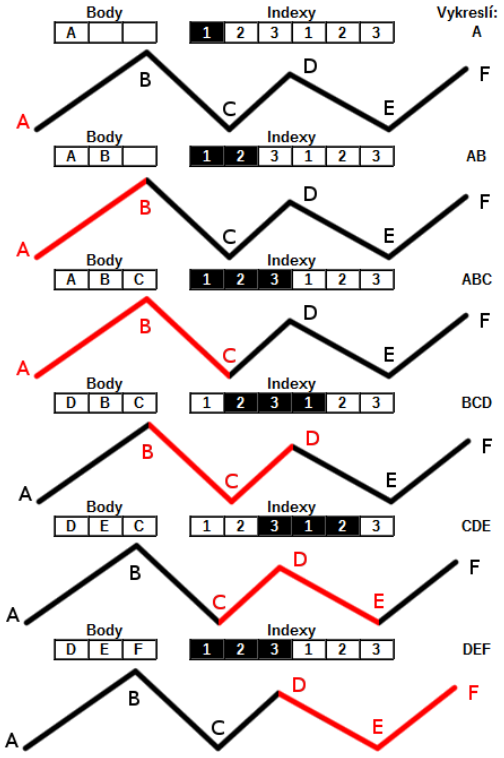
7.2.3 třída `UnitTrail`

Při simulaci Sluneční soustavy by se nám hodilo nejen vidět aktuální polohu všech objektů, ale i jejich minulé polohy. Tato třída vykresluje lomenou čáru, která pak slouží jako „stopa“ za simulovaným objektem. Ideálně bychom chtěli mít pouze funkci, která přidá další bod lomené čáry a to aktuální polohu.O to jak bude stopa uložena a kreslena se z vnějšku starat nechceme. Samozřejmě bude omezená maximální délku, ale to si má také ohlídat sama. Ještě pro úplnost existuje funkce `Clear`, která uloženou čáru smaže.

Algoritmus Využijeme toho, že OpenGL přímo podporuje kreslení lomených čar, stačí zadat množinu bodů a ono samo mezi nimi vykreslí lomenou čáru. Navíc použijeme indexované vykreslování, tedy, že řekneme OpenGL v jakém pořadí má body vykreslit, což ve výsledku zajistí, že celá množina bodů nemusí ležet ve správném pořadí a při přidávání bodů ji nebude nutné celou posouvat.

Mějme dvě pole - bodů velikosti **n** a indexů o délce **2n**. Dále máme objekt, který postupně projde body **A-B-C-D-E-F**. Tak pokud za ním chceme vykreslit stopu(lomenou čáru) o maximálně **n** bodech. Pak vytvoříme funkci `Push(Bod b)`, která umístí předaný bod do pole

Obrázek 7.1: Kreslení stopy za objektem procházejícím body A-B-C-D-E-F



Obrázek zobrazuje postupné(shora dolů) přidání 6 bodů do stopy o maximální délce 3. Červeně je vykreslena aktuální stopa. Pokud jsou černě vyplněné např. indexy(indexováno od jedničky) 2,3,1. Tak to znamená, že se vykreslí přímka v pořadí: `body[2],body[3],body[1]` *(což je napsáno vpravo).*

bodů. Pokud už je plné tak začne přepisovat body od začátku. Pole indexů vyplníme čísly od **1** do **n** dvakrát. Obrázek 7.1 popisuje jak bude algoritmus pro náš objekt fungovat při projití všech bodů, kde v každém bodě zavoláme `Push(bod)` a délka stopy je **n=3**.

Z obrázku si můžeme všimnout, že si musíme držet pouze ukazatel na prvek v poli bodů, kam budeme vkládat, a ukazatel na index v poli indexů od kterého máme kreslit. Oba ukazatele pak budeme vždy pouze cyklicky posouvat a odpadne tím nutnost přesouvat celé pole při přidání jednoho bodu.

Poznámka: *Teoreticky žádné indexy nepotřebujeme a mohli bychom kreslit vždy pouze úsečky. Ale zavolání jednoho vykreslení je v OpenGL relativně drahé, proto je lepší kreslit co nejvíce dat jedním příkazem. Proto také máme jeden kontinuální buffer pro všechny body. Tento algoritmus tedy považuji za kompromis mezi rychlostí a čitelností kódu. Teoreticky také stačí index buffer o velikosti* **n** *a pak čáru kreslit na dvakrát, protože je v bufferu buď uložená kontinuálně a nebo rozložená do dvou částí. Což by byla rozumná paměťová optimalizace, ale indexy zas tolik místa nezabírají a navíc existuje pouze jeden globální indexový buffer pro*

všechny instance třídy `UnitTrail` *neboť je jen pro čtení.*

7.2.4 Třída `LineTrailsDrawer`

Tato třída vytvoří pro každý simulovaný objekt jednu `UnitTrail`, do kterých pak periodicky ukládá jejich pozice a také všechny stopy vykresluje, k čemuž používá správný shader a informace z `OMSAR`. Nabízí také pár kontrolních funkcí jako selektivní mazání/vypínání/zapínání stop pro jednotlivé objekty.

7.3 Kreslení uživatelského rozhraní

Celé uživatelské rozhraní využívá plně knihovnu **ImGui** a je rozděleno do několika následujících souborů.

MouseControls Skupina funkcí, která implementuje ovládání pomocí myši. Tedy posouvání obrazovky a přibližování/oddalování.

SimProperties Skupina funkcí, které vykreslují okno s informacemi o simulaci a také dovolují simulaci ovládat.

Visuals Skupina funkcí, která vykresluje okno s ovládáním grafických prvků, tedy momentálně pouze kreslení stop za objekty. Dovoluje stopy selektivně vypínat/zapínat/mazat pro všechy objekty.

UnitsProperties třída, která vykresluje okno, které zobrazuje fyzikální informace o vybraném objektu.

Koncepčně na těchto třídách/funkcích není nic moc zajímavého. Ze zdrojových kódů a názvů funkcí v nich je celkem jasné co dělají.

Kapitola 8

Rozšířitelnost a vylepšení

Pokud jste už zkusili výsledný program spustit a samou radostí nad skvělou simulací vám něco uniklo a vy jste v panice začali hledat tlačítko na vrácení o krok zpět, tak jste zjistili, že tam žádné není. Simulace ve výchozím stavu neprovádí žádné cachování/ukládání mezivýsledků, takže se nelze vrátit zpět(A prostě zvolit záporné Δt u všech metod také nefunguje). Což je určitě rozumné rozšíření, ale my zkusíme něco trochu jiného - **přehrávač simulací**. Přehrávač by měl umět zaznamenat probíhající simulaci a poté ji přehrát jako video, tedy včetně přeskakování na libovolné místo simulace a zrychlení/zpomalení přehrávání.

8.1 Návrh přehrávače

Jak by se takový přehrávač dal implementovat? Co kdybychom vytvořili následující třídy:

1. Třída `ViewAndRecord`, která se chová jako libovolný jiný **viewer**, ale navíc si "tajně" zaznamenává simulaci do souboru.
2. Trojice tříd `ReplayerParser`, `ReplayerMethod` a `ReplayerViewer` které by se starali o samotné přehrávání simulace.
3. Dvoji tříd `RecordedSimulation` a `ReplayedSimulation` které vše zabalují do funkčního celku.

Celý přehrávač je plně funkční a součástí výsledného programu, proto si zde všechny koncepty alespoň přibližně představíme. Navíc se v původním programu vlastně koncepčně vůbec nic nemuselo měnit, vše plyne z flexibility modulů.

Poznámka autora: *Při pohledu do git* [1] *historie projektu zjistíte, že je to samozřejmě lež - měnilo se skoro všechno. Za prvé se úplně rozebral* `ImGuiViewer` *- vytáhly se z něj jednotlivé části, které se umístily do svých vlastních tříd.* `ImGuiViewer` *se z nich poté znovu sestavil zpět. Výsledná podoba je ta uvedená v dokumentaci, ve které jsou jednotlivé části co nejvíce nezávislé a hlavně znovupoužitelné. Za druhé se zpřehlednilo rozhraní třídy* `Simulation`*, hlavně práce s časem, kdy se přešlo od desetinných čísel k celočíselné reprezentaci, která netrpí na zaokrouhlování. Úpravy nebyly nutné z hlediska původní funkčnosti, ale už při implementaci se mi návrh moc nelíbil(zvlášť GUI bylo nepřehledné), každopádně cíl byl nejdřív*

[1] Viz. **Kapitola 10 Uživatelská příručka**

dostat funkční kód a pak ho zkrášlovat dle potřeby. Ve výsledku se velká část uživatelského rozhraní pro `ImGuiViewer` *použila i pro* `ReplayerViewer`*, což by předtím vyžadovalo ošklivé* `CTRL+C` a `CTRL+V`.

8.2 Zaznamenání simulace

Ideálně bychom chtěli zaznamenat jakoukoliv simulaci, což můžeme udělat například následujícím způsobem:

Zdrojový kód 8.1: Návrh jak by se simulace dala zaznamenávat

```
class ViewAndRecord : public Viewer
{
public:
    //Zaznamenává simulaci do souboru 'outFileName'
    ViewAndRecord(const std::string& outFileName, viewer_p viewer);
    ~ViewAndRecord()override final;
    void Prepare() override final;
    void operator()() override final;
private:
    //Implementační detaily...
};

class RecordedSimulation
{
public:
    //Outfile - soubor kam se uloží záznam
    RecordedSimulation(parser_p parser, simMethod_p simMethod, viewer_p viewer,
                       const std::string& outFile) :
        sim(std::move(parser), std::move(simMethod),
            std::make_unique<ViewAndRecord>(outFile, std::move(viewer)))
    {
    }
    //Stejné funkce pro spuštění,které zavolají sim.XXX() verzi.
    void Start(stepTime_t dt, size_t rawMult, size_t DTMult, seconds maxRunTime);
    void StartNotTimed(stepTime_t dt, size_t rawMult, seconds maxRunTime);
private:
    Simulation sim;
};
```

Díky modulům se zaznamenávání simulace docílí velmi jednoduše, protože jediné co musíme změnit je, že „zabalíme“ zvolený **viewer** do třídy `ViewerAndRecord` a poté ho předáme jako obyčejný **viewer** simulaci. Při běhu simulace bude pak volán `ViewerAndRecord`, který ale v rámci své `operator()()` také zavolá předaný **viewer** a navíc si bude na pozadí ukládat probíhající simulaci.

Čili takto jednoduše jsme docílili zaznamenání skoro libovolné simulace. Co se týče formátu záznamu, tak se jedná o binární soubor, kde jeho přesná specifikace je popsána u zdrojových kódů v souboru `FileFormats/ReplayerFile.txt`. Zaznamenávání je nastaveno tak, že jeden simulovaný krok odpovídá jednomu záznamu. Odsimulovaný čas si `ViewAndRecord` snadno zjistí pomocí funkce `simTime Simulation::GetSimTime()`. Slovo „skoro“ na začátku odstavce odkazuje na to, že tento postup selže, pokud někdo bude měnit simulovaný čas pomocí `void Simulation::SetSimTime(simTime)`, což ale klasická simulace dělat nepotřebuje.

8.3 Přehrávání

Přehrávání docílíme tím, že simulaci budeme podstrkovat data, která si přečteme ze souboru se zaznamenanou simulací místo toho abychom je simulovali sami. Tento podvod bude zajišťovat právě třída `ReplayerMethod`. Technicky potřebujeme ještě **parser** - `ReplayerParser`, ale ten jediné co udělá je, že přečte první data ze stejného souboru a tím vytvoří `simData_t`, která se od tohoto modulu očekávají. Pro zobrazení použijeme třídu `ReplayerViewer`, která využívá pracně vytvořené části z modulu `IMGuiViewer`. Navíc ještě přidává okno, které obsahuje ovládání přehrávání. Mezi funkce patří slíbené zpomalení/zrychlení přehrávání a také přeskočení do libovolného bodu záznamu.

Jak to funguje? `ReplayerMethod` si přečte aktuální odsimulovaný čas, podle toho si v souboru najde záznam, který tomu má odpovídat a simulovaná data prostě změní podle tohoto záznamu. Ve skutečnosti ještě provede lineární interpolaci mezi dvěma záznamy, aby byl výsledek plynulejší při menších rychlostech. `ReplayerMethod` je poté zobrazí bez ohledu na jejich původ. Pokud potřebujeme přehrávání zpomalit, tak využijeme parametru `DTMultiplier`. A pokud chceme přeskočit někam jinam, tak změníme odsimulovaný čas pomocí `void Simulation::SetSimTime(simTime)`.

Tím máme vlastně všechny moduly určené, ale nutit uživatele k tomu aby je vytvářel sám je zbytečné. Proto se o toto stará třída `ReplayedSim`, která obsahuje samotnou simulaci, které vytvoří a předá potřebné moduly.

8.4 Vylepšení a opravy

V průběhu prací na programu jsem si vedl v souboru plán toho, na čem budu pracovat, časem se tam objevily různé věci, které se do finálního programu z různých důvodů nedostaly, proto je zmíním alespoň zde jako možná vylepšení a také opravy toho co nebylo ještě opraveno.

Fyzikální jednotky Asi hlavní věc, která programu momentálně chybí je nějaký ucelený systém fyzikálních jednotek. `simData_t` obsahují pouze data, takže záleží na dohodě mezi moduly, v jakém formátu tyto data jsou. Což je samozřejmě nepraktické. Řešení by bylo změnit `simData_t` z vektoru na alespoň strukturu, která by měla pomocné proměnné označující v jakých jednotkách data jsou. Například pomocí **enum**, popřípadě rovnou dvojice (čitatel, jmenovatel) pro převod do jednotek SI, stejně jako používá knihovna `std::chrono`.

Šablony pro vektory Není potřeba nic extra složitého, ale stálo by za to mít obecný vektor pro libovolný počet prvků a hlavně pro libovolná data. V samotném vektoru není potřeba moc změn, jen přepsání **double** na **T** a přidání pár **template**, ale tím se rozbije valná většina kódu všude možně v programu, takže oprava je zdlouhavá a otravná. Každopádně obecné vektory by se na pár místech v kódu hodily - GUI, OpenGL.

Rozšíření do 3D Pokud bychom měli obecný vektor, tak rozšíření do 3D už je skoro hotové, protože integrační metody fungují pro libovolnou dimenzi a kód v nich by se skoro přepisovat nemusel. Do vykreslení pak stačí přidat projekční matici a „trackball“ pro ovládání myší a bylo by hotovo.

Zaznamenávání podle `simMethod` Momentálně je zaznamenávání simulace vázáno na **viewer**, což je nepraktické, protože data opravdu mění **simMethod** a může se stát díky

časování, že simulace proběhne vícekrát a my se na mezidata nepodíváme. Toto hodnotím jako špatný návrh, který by se měl opravit. Momentálně je to spíše opravdu jako zaznamenávání obrazovky při přehrávání simulace. Když se přehrávaná simulace bude sekat, tak bude zasekaný i záznam, přitom by stačilo si ukládat samotnou simulaci a ne zbytečně obrazovku se simulací.

Zpomalení přehrávané simulace Pokud byla simulace zaznamenána v reálném čase, tedy $DTmult = RawMult = 1$, tak nejde při přehrávání zpomalit, protože to k tomu využívá právě `DTmult`, který ale nemůže být menší než 1. Tomu by mohlo pomoct měnit také `deltaT` protože to je pořád uloženo jako desetinné číslo, ale situace by neměla nastat moc často, protože simulaci přítomnosti moc často nechceme. Navíc zpomalení neposkytuje žádné nové informace, jen dochází k lineární interpolaci mezi záznamy, aby i pomalejší přehrávání bylo plynulé.

Lepší než lineární interpolace Momentálně přehrávání používá lineární interpolaci polohy, což není moc přesné. Dala by se interpolovat i rychlost a zohlednit to v poloze. Aby se např. obíhající měsíc pohyboval po oblouku a ne po přímce mezi záznamy.

Jde to vidět například u měsíců Jupiteru, které jsou stabilní pro semi-implicitního Eulera i při vysokém simulačním kroku a při zpomaleném přehrávání se pohybují spíše po mnohoúhelníku než kružnici. Ale toto je jen vizuální detail, nic podstatného.

Lepší GUI Uživatelské rozhraní je podle mého názoru dostatečné, ale knihovna **ImGui** je celkem rozsáhlá a nabízí podporu např. pro kreslení grafů. Takže by se dal program rozšířit o velmi detailní statistiky - např. rozložení hmotnosti a rychlostí v soustavě. A také celkové energie systému, se kterou by se dala sledovat chyba numerické metody, protože energie uzavřeného systému je konstantní. Také by si GUI zasloužilo lepší podporu pro různé velikosti okna, což momentálně funguje, ale není to moc hezky napsané.

Nějaké zajímavé numerické metody Toto by neměl být vůbec problém, kvůli tomu přesně existují moduly. Původně jsme chtěl přidat alespoň jednu vícekrokovou metodu[2], ale upřímně se mi už moc nechce, program i dokumentace jsou už dostatečně rozsáhlé. Ale zajímalo by mě, jak si vedou v poměru přesnost/výkon hlavně vzhledem k RK4. Také využití paralelizace by bylo zajímavé, což by mohlo dovolit simulovat více objektů. Popřípadě rovnou pomocí GPU.

Komprese zaznamenaných simulací Aktuálně nedochází k žádné kompresi záznamu, takže po 5 minutách simulace při zaznamenání každých 10ms má soubor velikost kolem 15MB (30 tisíc záznamů). Avšak těžko říct kolik se bude dát reálně ušetřit, 7Zip tento konkrétní soubor zkomprimuje na 12MB pomocí algoritmu LZMA2, což není nic extra. Ale nutně takovou vzorkovací frekvenci ani potřebovat nebudeme, rozšíření o pořádnou interpolaci by tak mohlo být v důsledků účinější než komprese pokud jsou trajektorie dostatečně hladké.

Ale malý krok simulace nevolíme většinou kvůli nutnosti znát polohu v každém čase, ale kvůli přesnosti simulace a samotných výsledků, takže řešením by byla možnost změnit ve třídě `ViewAndRecord` jak často se má zaznamenávat, což je úprava na pár řádku (změna `timeStep` a `multiplier` v .cpp by měla stačit).

[2] Např. anglická verze Wikipedie má dobře sepsané prediktor-korektor metody `https://en.wikipedia.org/wiki/Linear_multistep_method`

Použití jako knihovna Původní záměr byl vytvořit program, který se dá použít i jako knihovna, což je možná v návrhu občas i vidět. Řekl bych, že je to skoro hotové, jen jsem to vůbec netestoval. Vím, že nejspíš bude problém, pokud bude víc instancí třídy `ImGuiViewer` neboť ta se spoléhá na `ImGuiBackEnd`, která se bude snažit inicializovat už inicializované **ImGui**. Také `OpenGLBackend` by možná nemuselo být dobré inicializovat vícekrát. Což by vyřešila statická inicializační metoda se statickou `bool` proměnnou nebo nějaká forma Singletonu.

Kapitola 9

Porovnání integračních metod

V této kapitole alespoň zběžně porovnáme implementované integrační metody. Protože potencionálního uživatele by mohlo zajímat, kterou metodu by měl použít pro co nejpřesnější ale i nejrychlejší výsledek.

Srovnání bude probíhat nejdříve na případu kruhového pohybu. Důvod je ten, že objekty v naší soustavě obíhají většinou po kružnicích/elipsách a také už pro něj máme vyřešenou implicitní verzi Eulerovy metody. Budeme hodnotit přesnost, stabilitu v závislost na Δt a rychlost výpočtu.

V dalších sekcích už nebudeme metody znovu vysvětlovat, pouze se odkážeme na odpovídající kapitoly.

9.1 Kruhový pohyb

Zdrojové kódy použitých metod, které byly použity k získání dat, jsou dostupné v souboru `Source/stabilita.cpp`. Dále je zde soubor `plot.gp` pomocí něhož a gnuplotu [1] vznikly všechny obrázky.

9.1.1 Teorie

Pohyb po kružnici v rovině můžeme zapsat rovnicí (9.1), kde ω je úhlová rychlost. Což je také řešení diferenciální rovnice (9.2)

$$\boldsymbol{u}(t) = (x, y)(t) = (cos(\omega t), sin(\omega t)) \tag{9.1}$$

$$\ddot{\boldsymbol{u}}(t) = -\omega^2 u(t) \quad \boldsymbol{u}(0) = (1, 0) \quad \dot{\boldsymbol{u}}(0) = (0, 1) \tag{9.2}$$

V dalších částech zvolíme $\omega = 1$.

9.1.2 Explicitní Euler

Explicitní Eulerovu metodu jsme si představili v úvodní teorii 1.2.2. Výsledky jsou zobrazeny v grafu na obrázku 9.1, z něhož je vidět, že explicitní metoda diverguje od správného řešení pro libovolné Δt Což není moc překvapivé, protože považuje derivaci na integračním oboru za konstantní, což je na počátku tečna procházející bodem (1,0) a tím dojde vždy k posunu

[1] `http://www.gnuplot.info/`

směrem ven z kružnice. Tato metoda tedy nevypadá jako vhodná pro naši simulaci, což lze pozorovat např. u měsíců Marsu, které se i pro malá DT(500+) pomalu oddalují po spirále. Tady by se právě hodilo mít rozšíření na vytváření grafů vzdáleností mezi jednotlivými objekty.

9.1.3 Implicitní Euler

Implicitní Eulerova metoda byla ukázána v části 6.2 i s řešením jedné rovnice (6.3), které můžeme použít, neboť platí, že $k = -\omega^2$. Bohužel ani tato metoda není ideální jak ukazuje obrázek 9.2. Dochází zde k opačnému jevu, což je způsobeno tím, že integrace probíhá po tečně v čase $t+\Delta t$, což vede k posunu směrem do kružnice. Implicitní metoda nebyla použita, protože se z naší soustavy (1.3) nedá jednoduše vyjádřit.

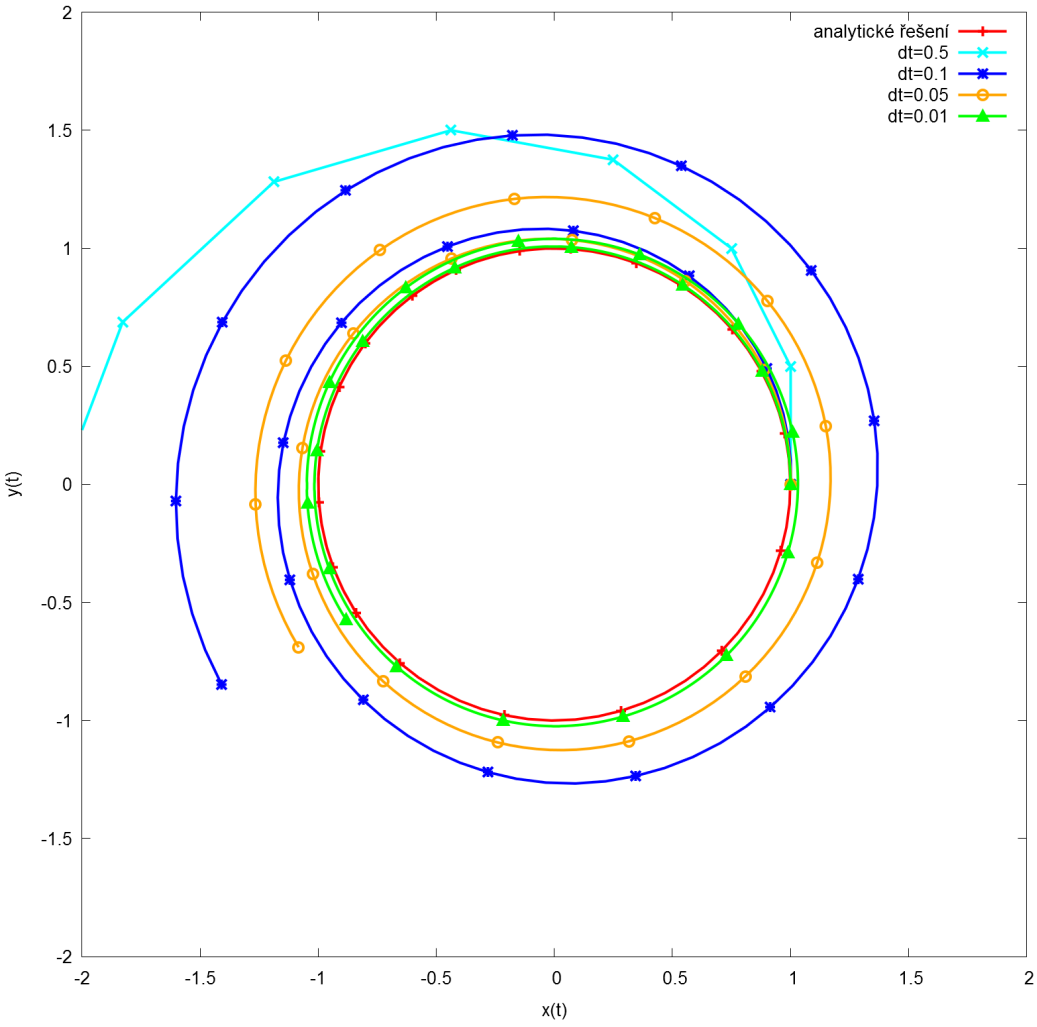
9.1.4 Semi-implicitní Euler

Tato metoda je navrhnuta také v části 6.2 jako jednodušší řešení implicitní verze. Z obrázků 9.3 a 9.4 je vidět, že je tato metoda navíc stabilní i pro větší Δt, ale pro ně není už moc přesná.

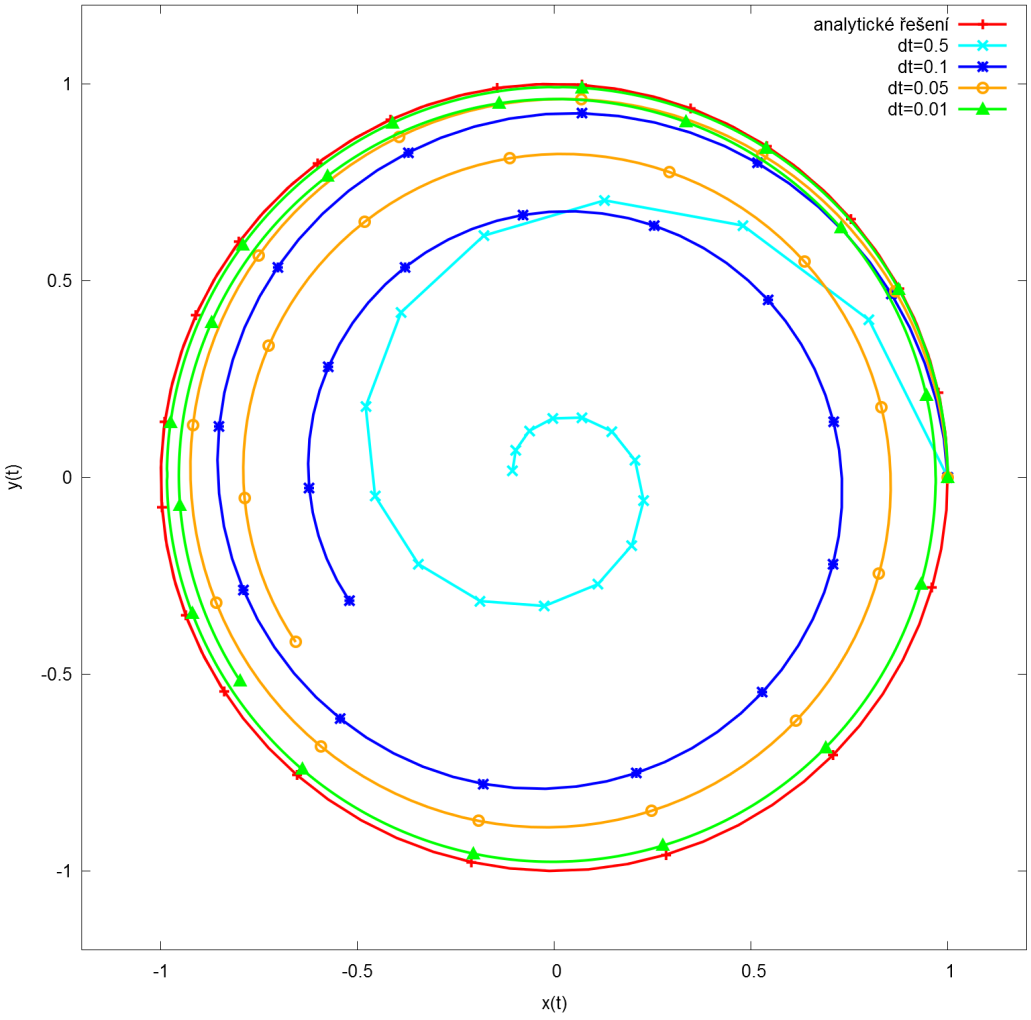
9.1.5 RK4

Poslední implementovaná metoda je RK4, která je popsána v sekci 6.3. Její přesnost zachycují obrázky 9.5 a 9.6. Tedy už při $\Delta t = 0.1$ se jedná o velmi stabilní metodu a je přesnější než semi-implicitní metoda . Ale naopak pro vyšší Δt stabilní není.

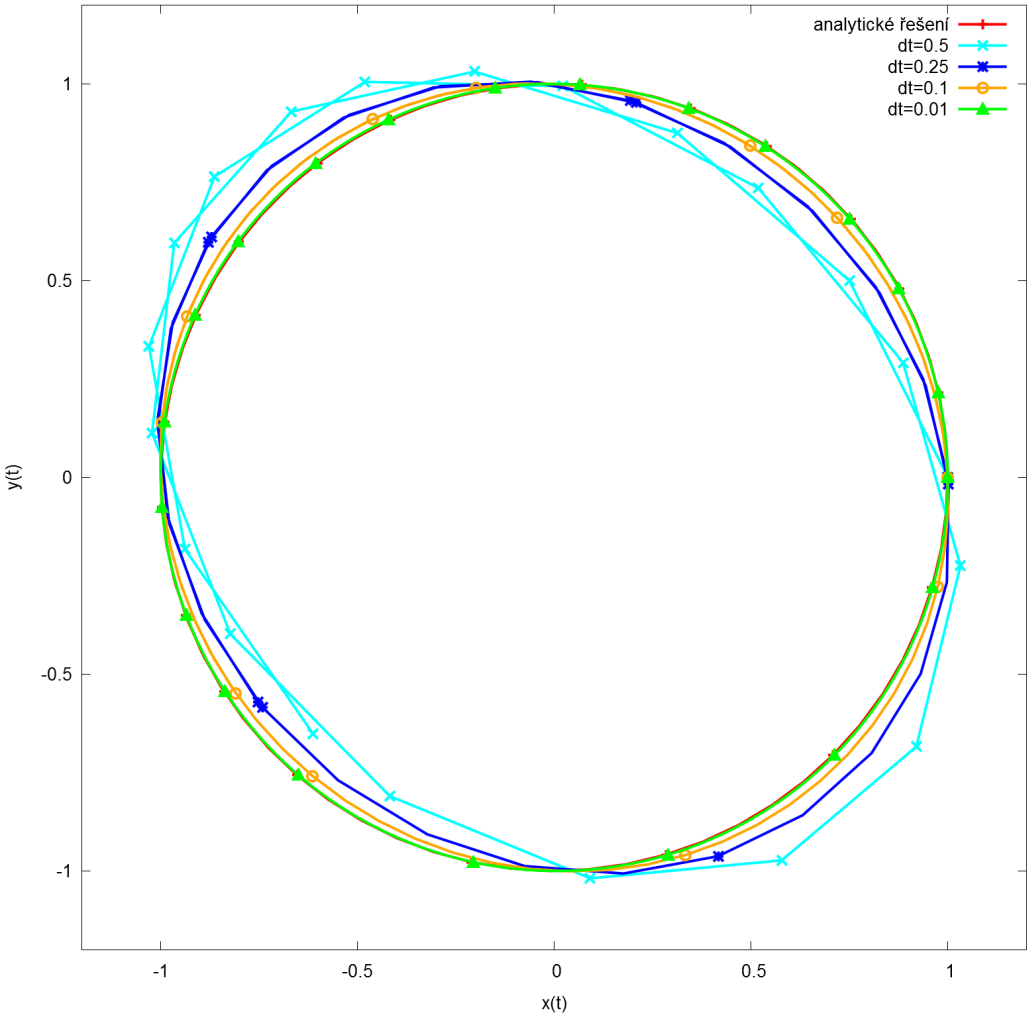
Obrázek 9.1: Explicitní Eulerova metoda $t \in [0, 10]$



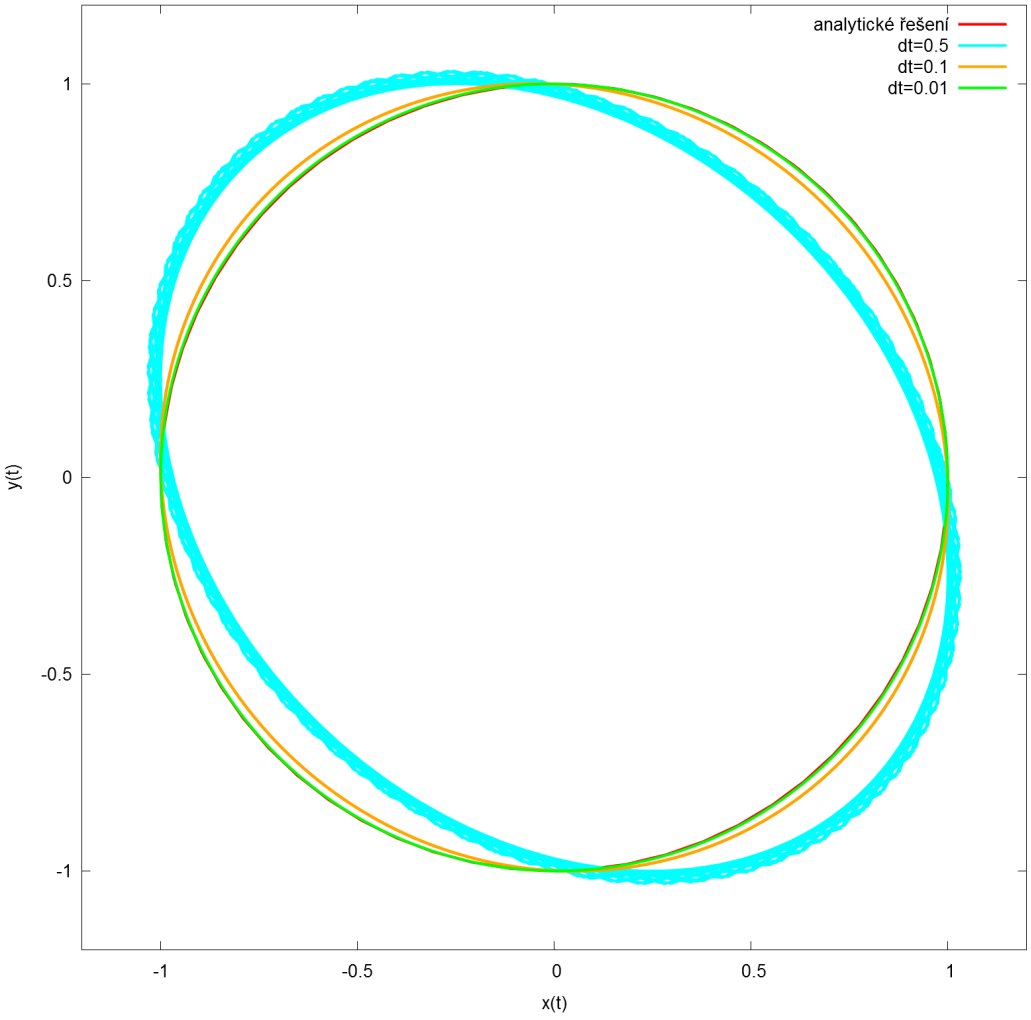
Obrázek 9.2: Implicitní Eulerova metoda $t \in [0, 10]$



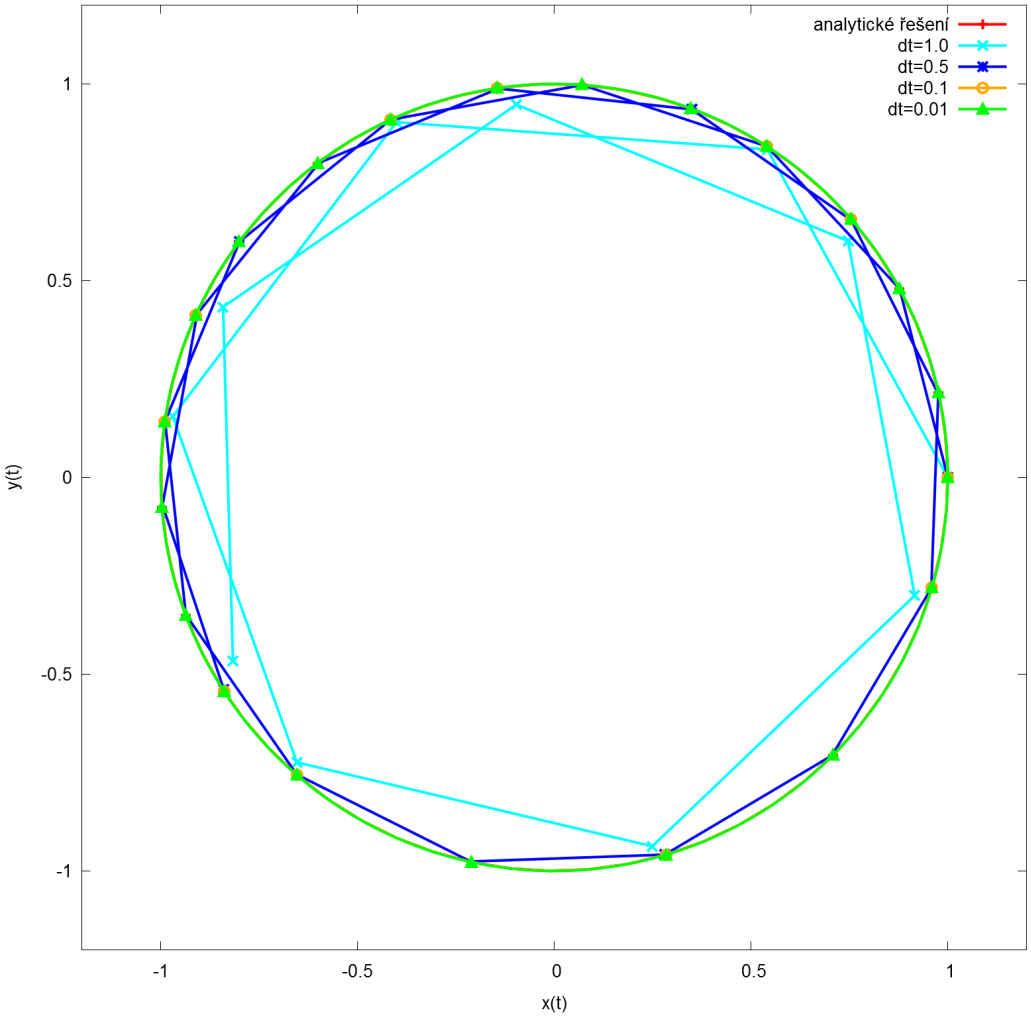
Obrázek 9.3: Semi-implicitní Eulerova metoda $t \in [0, 10]$



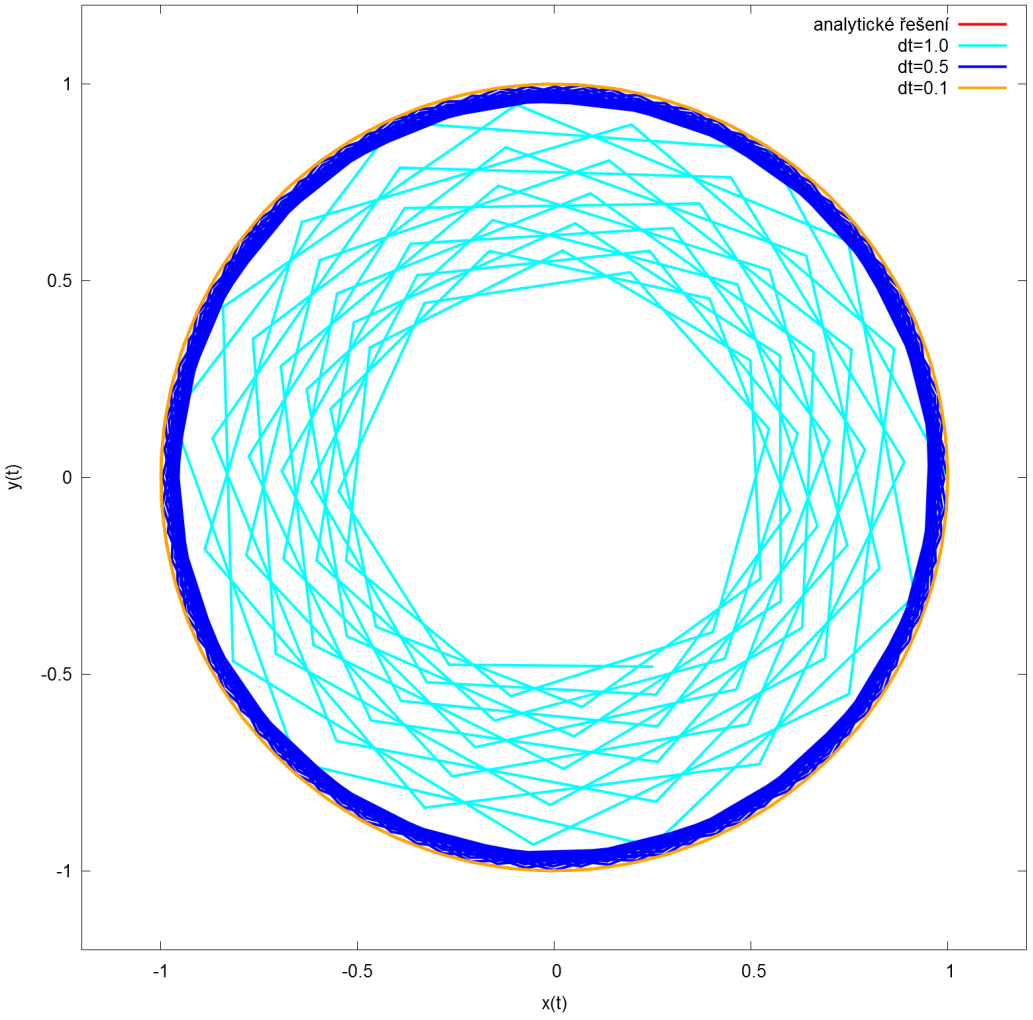
Obrázek 9.4: Semi-implicitní Eulerova metoda $t \in [0, 100]$



Obrázek 9.5: RK4 metoda $t \in [0, 10]$



Obrázek 9.6: RK4 metoda $t \in [0, 10]$



9.2 Sluneční soustava

Pojďme se ještě podívat na srovnání stability semi-implicitní a RK4 metody na příkladu naší Sluneční soustavy. Jako indikátor stability použijeme stabilitu oběžných drah měsíců, protože ty mají nejkratší oběžné doby a nestabilita se zde projeví jako první. Samozřejmě toto je pouze nutná a ne postačující podmínka správné simulace. Naměřené údaje jsou uvedeny v tabulce 9.1. Vyšší hodnoty nejsou podstatné, protože se zde ve velké míře projeví nepřesnost, která vyvolá nestabilitu systému. Každopádně to ukazuje, že semi-implicitní Eulerova metoda je stabilnější, což i trochu naznačovaly obrázky u kruhového pohybu, ale také obrázek 9.4 ukazuje stabilní, avšak nesprávný „orbit". Na druhou stranu obrázek 9.6 uvádí RK4 jako přesnější metodu pro podobné Δt. Přesnost se bohužel pro Sluneční soustavu nedá nijak jednoduše ověřit.

Tabulka 9.1: Stabilita měsíců pro semi-implicitní Eulerovu a RK4 metodu

	Měsíc		Phobos		Deimos		Io		Europa		Ganymed		Callisto		**Celkem**	
DTmult	S	R	S	R	S	R	S	R	S	R	S	R	S	R	S	R
70 000	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	**7**	**7**
80 000	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	**7**	**6**
300 000	✓	✓	✓*	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	**7**	**6**
320 000	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	**6**	**5**
400 000	✓	✓	✗	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	**6**	**4**
1 000 000	✓	✓	✗	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	**6**	**3**
1 200 000	✓	✓	✗	✗	✓**	✗	✓	✗	✗**	✗	✗**	✓	✓	✓	**4**	**3**
1 300 000	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	**3**	**3**

S=semi-implicitní; R=RK4 metoda. Krok simulace byl DTmult $*$ *10ms.*
Stabilita znamená, že měsíc stále obíhal kolem své planety po 5/20/60 letech simulovaného času pro DTmult $\geq 0/ \geq 3.10^5/ \geq 10^6$ *. Tabulka neobsahuje všechny hodnoty, pouze ty, kolem kterých došlo ke změnám.*
**Phobos opisuje 13-úhelník, ale je stabilní.*
***Ganyméd a Europa se nejspíše srazily, Deimos stabilně opisuje 12-úhelník.*

9.3 Rychlosti a Závěr

Srovnání rychlostí obou metod nabízí tabulka 9.2. Z které plyne, že RK4 je přibližně 3-4x pomalejší než Eulerova metoda. Takže ve výsledku je dle našich měření RK4 pomalejší a nestabilnější, přesto je použita jako výchozí. Důvod je ten, že je podle mne mnohem přesnější, stabilita Eulera vypadá sice dobře, ale silně pochybuji, že v tu chvíli simulace odpovídá realitě. Ale samozřejmě uživatel si může stále vybrat, což byl také účel a tato kapitola měla za cíl toto rozhodnutí alespoň trochu ulehčit. Takže pokud chce uživatel spíše stabilní(I když v tom případě bych uvažoval o implementaci Keplerových zákonů), avšak možná nepřesnou simulaci, tak je lepší semi-implicitní Eulerova metoda, pokud dává přednost přesnosti, tak bych volil RK4 nebo jiné. Bohužel „jiné" zde nejsou, což je škoda. Také by se pro lepší porovnání hodilo mít k dispozici přesné grafy rychlostí, vzdáleností, energií už v průběhu simulace. Proto jsou také uveden v možnostech rozšíření viz. 8.4.

Tabulka 9.2: Srovnání rychlostí semi-implicitní Eulerovy a RK4 metody

Metoda		**Počet objektů**		
		10	20	100
Euler	Debug	1ms	2ms	3ms
	Release	1ms	2ms	3ms
RK4	Debug	1ms	2ms	3ms
	Release	1ms	2ms	3ms

Přesné nastavení Debug a Release je uvedeno v .sln u zdrojových souborů. Release má zapnuté veškeré optimalizace - inlining, SIMD...;Debug je výchozí režim bez většiny optimalizací. Hodnoty byly vypočteny z 100 000 volání každé funkce na soustavě s procesorem AMD FX-4100 - 4x3.6Ghz se systémem Win7 64-bit(Ale aplikace je 32-bitová.)

Kapitola 10

Uživatelská příručka

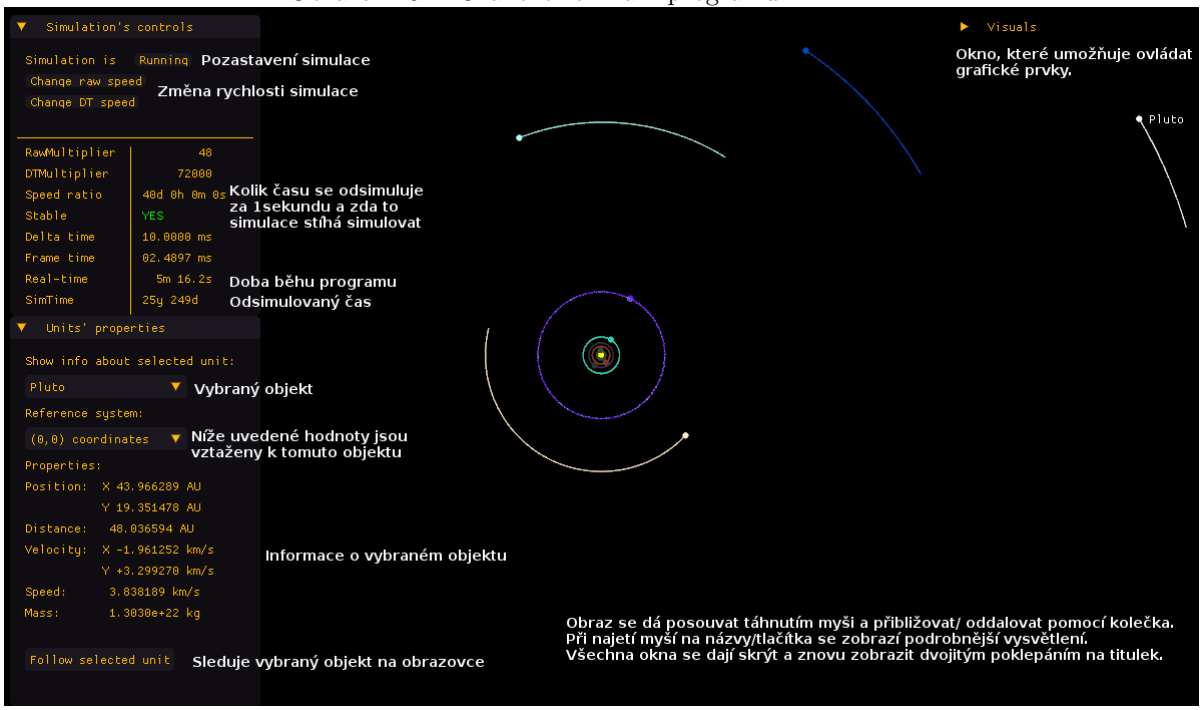
10.1 Požadavky

Vyžaduje verzi OpenGL 3 nebo vyšší, navíc je potřeba balíček Microsoft Visual C++ 2015 Redistributable, který je ve formě .dll přibalen. Pokud by s ním přesto byly problémy, tak doporučuji stáhnout a nainstalovat jeho aktuální verzi z `https://www.microsoft.com/cs-CZ/download/details.aspx?id=53840`

10.2 Základní ovladání

Při spuštění se program otevře v výchozím grafickém režimu, kde dojde k nahrání zabudované Sluneční soustavy. Simulace se ovládá pomocí grafického rozhraní - Obrázek 10.1.

Obrázek 10.1: Grafické rozhraní programu



10.3 Pokročilé možnosti

Program také nabízí pokročilejší ovládání pomocí příkazové řádky, s níž je možné načítat simulovaná data ze souboru, vybrat si integrační metody a také možnost zaznamenat a následně přehrát uložené simulace. Pro vypsání nápovědy a všech dostupných příkazů v českém jazyce použijte: **`SolarSystem.exe -help -cz`**

Uveďme zde alespoň pár dalších příkladů různých příkazů:

1. **`SolarSystem.exe vstup.txt`**
   Načte vstupní data z formátovaného souboru `vstup.txt` a spustí simulaci, která bude běžet v reálném čase a zobrazovat se v okně 1200x700 s uživatelským rozhraním.

2. **`SolarSystem.exe -sim -p formatted -i vstup.txt -m RK4 -v win`**
   Ekvivalentní zápis předchozího příkladu. S explicitním použitím modulů.

3. **`SolarSystem.exe -sim -rm 24 -dm 3600 -x 300`**
   Spustí podobnou simulaci jako v 1. příkladu. Akorát bude mít 3600x větší integrační krok a bude probíhat 10x rychleji. Ve výsledku se tedy odsimuluje jeden den za jednu reálnou sekundu. Simulace se vypne po 5 minutách běhu.

4. **`SolarSystem.exe -record -r zaznam.replay -p formatted -i vstup.txt -o vystup.txt -m semiEuler -rm 24 -dm 3600 -x 60`**
   Zaznamená 60 sekundovou simulaci do souboru `zaznam.replay`, vstupní data načte z formátovaného souboru `vstup.txt` a výsledná data uloží do `vystup.txt` ve stejném

formátu. Simulace bude simulovat jeden den za jednu sekundu pomocí semi-implicitní Eulerovy integrační metody. Simulace se spustí v grafickém prostředí stejně jako v 1. příkladě.

5. **`SolarSystem.exe zaznam.replay`**
   Přehraje zaznamenanou simulaci z předchozího příkladu v grafickém prostředí.

K .exe souboru je přiložen vzorový formátovaný text `vstup.txt` a také jeden záznam simulace `zaznam.replay`. Přesný popis formátovaného vstupního souboru je uveden v sekci 4.1 . Pro detailní vysvětlení parametrů simulace je k dispozici popis v sekci 2.1.3.

Kapitola 11

Kompilace programu

11.1 Git

Tato dokumentace spolu se zdrojovými kódy programu je veřejně dostupná na:

`https://bitbucket.org/Quimby/solar/src`

Popřípadě přímo stažitelná pomocí git HTTPS(11.1) nebo SSH(11.2):

`https://Quimby@bitbucket.org/Quimby/solar.git` (11.1)

`git@bitbucket.org:Quimby/solar.git` (11.2)

11.2 Windows

Program byl tvořen v programu Visual Studio 2015 Community Edition, takže je zde už předpřipravený .sln projekt, který by měl obsahovat vše potřebné pro správné zkompilování.

11.3 Linux

Program by zde měl být teoreticky také plně funkční, avšak není zde zatím žádný pomocný projekt ani makefile. Kdyžtak je potřeba nejdříve stáhnout a zkompilovat externí knihovny - GLFW, GLEW. Návod by měl být na jejich oficiálních stránkách . Momentálně obě knihovny nabízí vytvoření potřebných souborů pomocí programu CMake, takže kompilace by neměla být těžká. Dále kompilace samotného programu vyžaduje minimálně flag `GLEW_STATIC` a také include path do složky obsahující složku `Source/`(defaultně se nachází ve složce SolarSystem) a samozřejmě slinkovat obě knihovny, ale pak by už mělo vše fungovat.

Seznam zdrojových kódů

Seznam obrázků

Seznam tabulek