Simulace Sluneční soustavy

Dokumentace k zápočtovému programu pro předmět

Programování I(NPRG030)

Jan Waltl

29. ledna 2017

Obsah

Seznam zdrojových kódů

Organizace dokumentu

Tento text je organizován do následujících částí:

Úvod - První kapitola, zde je uvedena specifikace zápočtového programu a vysvětlena teoretická část.

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

Uživatelská příručka - Poslední dvě kapitoly popisují jak program správně zkompilovat a také jak s ním pracovat z neprogramátorského pohledu.

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:

Text vystihující příklad (Název souboru)

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout<<"Hello World!\n";
6      return 0;
7  }
```

Všechny další uvedené zdrojové kódy se nachází ve složce `Source/`, která by měla být připojena k této dokumentaci. Z praktických důvodů **nemusí** být tyto soubory zkompilovatelné, popřípadě mohou být psány z části v pseudo-kódu. **Nejedná** se o zdrojové kódy samotného programu. Primární účel je pouze **popisný**, kde snaha je ilustrovat koncepty a algoritmy, které se v programu v nějaké formě skutečně vyskytují.

Zdrojové kódy samotného programu jsou také součástí dokumentace a obsahují komentáře, které mohou stát za přečtení.

Kapitola 1

Úvod

1.1 Specifikace

Program simuluje Sluneční soustavu za využití numerické integrace a Newtonova gravitačního zákona. 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 spolu s metodami numerické integrace tak nabízí alternativní řešení tohoto problému.

Vstupem programu jsou strukturovaná data uložená v textovém souboru, která definují fyzikální veličiny simulované soustavy.Tedy polohy, rychlosti a hmotnosti simulovaných objektů.

Výstup je 2D grafická reprezentace simulované soustavy v reálném čase. Uživatelské rozhraní dovoluje měnit rychlost a přesnost simulace. Dále zobrazuje užitečné informace, jako jsou aktuální pozice, rychlosti pro každý simulovaný objekt vzhledem k jiným objektům.

Při vývoji programu byl kladen co největší důraz na pozdější rozšířitelnost. Výsledný program tedy poskytuje několik simulačních metod a možností vstupů a výstupů.

1.2 Teorie

1.2.1 Analytický popis

Newtonův Gravitační zákon (dále NGZ) popisuje vzájemné silové působení F_g dvou hmotných bodů [1] , kde výsledná síla je přitažlivá.

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

m_1, m_2 jsou hmotnosti obou bodů a $\boldsymbol{x_1}, \boldsymbol{x_2}$ jejich polohy.

Dále budeme pokládat simulované objekty za hmotné body, což je vzhledem k rozměrům hvězd, planet, měsíců a jejich vzdálenostem rozumná aproximace.

[1] Myšlené těleso, kde jeho veškerá hmotnost je soustředěna do jednoho místa - **hmotného bodu**.

Nyní nám NGZ spolu s principem superpozice [2] a Newtonovým Zákonem síly (1.2) dává pro n těles následující (1.3) soustavu 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.

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

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

Analytické řešení této soustavy rovnic by nám dalo možnost zjistit polohu, rychlost a zrychlení libovolného simulovaného objektu v libovolném čase na základě počátečních podmínek.

Bohužel 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. Numerická integrace využívá toho, že nemusí být 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 integrační metoda - Eulerova explicitní metoda(1.5). Mějme jednoduchou soustavu rovnic (1.4) . Je vidět, že řešením je funkce$y = e^t$, což se snadno ověří zpětnou derivací.

$$\dot{y} = e^t, y(0) = 1 \tag{1.4}$$

Zkusme tuto rovnici vyřešit numericky Eulerovou metodou (1.5). Výsledné hodnoty jsou uvedeny v tabulce 1.2.2, kde bylo použito postupně $\Delta t = 1;\ 0.1;\ 0.01$ a také analytické řešení.

$$y(t + \Delta t) = y(t) + \Delta t.y'(t) \tag{1.5}$$

$\boldsymbol{t}$	$\boldsymbol{e^x}$	**Euler**
1	1	1
1.1	1.254	1.255

Tabulka 1.1: Caption for the table.

Z tabulky vidíme, že pro dostatečně malé Δt dostáváme dostatečně přesné hodnoty. 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 integrační metody - semi-implicitní Euler a RK4 jsou popsány u svých implementací 6.2.

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

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 (Source/main.cpp)

```
#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` (Source/simPublic.cpp)

```
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, proto se na ni podrobně podíváme. Nejprve si vysvětlíme její teoretický návrh a poté i jeji implementaci, což nám 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 víme, že také neexistuje pouze jedna integrační metoda, takže bychom chtěli mít na výběr. Určitě 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.
4. **Prohlížet data**. Představme si situaci: 3:38 ráno, naše nová 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 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()` (Source/startPseudo.cpp)

```
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 (Source/startPseudoTimed.cpp)

```
1   void Simulation::Start(time deltaT,/*další parametry*/)
2   {
3       //Beze změny
4
5       auto 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.

[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/`

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 simulační metody, což může být potencionálně náročné na CPU. Pro nízké hodnoty by se 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 Implementace metody `Start()`

Poučeni nutnou dávkou teorie 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 a také pomocí C++ knihovny `chrono` implementuje opravdové časování , které zde bylo uvedeno spíše koncepčně.

Zdrojový kód 2.5: Finální verze `Simulation::Start()` (Source/startFinal.cpp)

```
1 void Simulation::Start(time deltaT, size_t rawMult, size_t DTMult, time
      maxSimTime)
2 {
3     // Případné 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              simMethod->Step(deltaT*DTMult); // Uděláme jeden krok simulace
23              acc -= deltaT*DTMult;
24              }
25          }
26          viewer->ViewData();// Podíváme se na simulovaná data
27      }
28      //Po doběhnutí simulace případně uložíme výsledná data
29      parser->Save(data);
30  }
```

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` (Source/unit.cpp)

```
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` (Source/parser.cpp)

```
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` (Source/simMethod.cpp)

```
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` (Source/viewer.cpp)

```
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.

4.0.1 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 ANSCII, 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 představující barvu ve formátu RGBA. Pokud není uvedeno, pak je bíla.

velocity<X Y> - počáteční rychlost objektu. Očekává dvě desetinná čísla 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. Také zde nesmí být jednotky uvedeny. 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 následujícími způsoby: 104.25 $\quad$ 1.0425e2. **Parser** ovšem **neumí** aritmetiku, takže následující **není** validní vstup: 10425/100 $\quad$ 104 + 0.25 $\quad$ $1.0425 * 10^2$.

`FormattedFileParser` je relativně shovívavý k formátování, takže následující je opravdu ekvivalentní definice objektu `Sun` z předchozího příkladu, ovšem čitelnost takového vstupu ponecháme bez komentáře.

```
1                { name      Bylo nebylo <Sun> za
2 sedmerocolor    horami <1.0        0.88             0.0     1.0 a
3         sedmero
4     velocityrekamiposition      jedno                              <
5 0.0
6  0.0  male      mass
7  <                 1.98843e30  kralovstvi
8 > }
```

4.0.2 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.

Při nekorektním vstupu, nemožnosti otevřít vstupní soubor nebo vytvořit soubor výstupní vyvolají funkce výjimku `Exception`.

Poznámka autora: *Je pravda, že načítání by nutně nemuselo načítat celý soubor najednou, ale pouze bloky závorek. Ale při velikosti textových souborů a operační paměti průměrného počítače nebyla paměť nejvyšší prioritou při programování zpracování vstupu.*

Kapitola 5

SolarParser

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 SemiImplicitEuler

6.1.1 Teorie

Při představení numerické integrace jsme si ukázali explicitní Eulerovu metodu pro numerické řešení obyčejné diferenciální rovnice. Ještě existuje implicitní verze (6.1) této metody, která je teoreticky shodná s explicitní až na to, že derivaci vyčíslíme v čase $t + \Delta t$.

$$y(t + \Delta t) = y(t) + \Delta t.y'(t + \Delta t) \tag{6.1}$$

Implicitní verze dává při stejném Δt přesnější, protože dochází k interpolaci, která na rozdíl od extrapolace neakumuluje chybu.

Nyní se podívejme jak bychom řešili jednoduchou diferenciální rovnici (6.2). Na ní nelze Eulerovu metodu přímo použít neboť ta řeší jen rovnice s první derivaci. Ale pomůžeme si tím, že každá obyčejná diferenciální rovnice n-tého stupně se dá převést na n rovnic prvního stupně. Takže rovnici z (6.2) si upravíme na dvě rovnice (6.3a) a (6.3b)

$$\ddot{x}(t) = k.x(t) \tag{6.2}$$
$$\text{s počátečními podmínkami:} \quad x(0) = x_0, \quad \dot{x}(0) = v_0$$

$$\dot{x}(t) = v(t) \tag{6.3a}$$
$$\dot{v}(t) = k.x(t) \tag{6.3b}$$

Toto jsou sice dvě rovnice, ale obě obsahují pouze první derivaci, už tedy můžeme použít Eulera. Tak to zkusme nejdříve s (6.3b) a implicitní verzí. Tím dostaneme rovnici (6.4). Teď použijeme stejný postup i na (6.3a) a výsledkem je (6.5). Nyní stačí dosadit první rovnici

do druhé a po úpravě dostaneme hledaný výsledek v podobě rovnice (6.6).

$$v(t+\Delta t) = v(t) + \Delta t.\dot{v}(t+\Delta t) = v(t) + \Delta t.k.x(t+\Delta) \tag{6.4}$$

$$x(t+\Delta t) = x(t) + \Delta t.\dot{x}(t+\Delta t) = x(t) + \Delta t.v(t+\Delta t) \tag{6.5}$$

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

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

$$x(t+\Delta t) = \frac{x(t) + \Delta t.v(t)}{1 - \Delta t^2 k} \tag{6.6}$$

To sice dalo určitou práci, ale dostali jsme správné řešení. Na pravé straně (6.6) máme proměnné pouze v čase t. Ty už umíme spočítat, protože $x(t)$ dostaneme z předchozího integračního kroku a $v(t)$ spočítáme z levé strany (6.4), kde napravo budeme mít $v(t-\Delta t)$ a $x(t)$. Obě tyto hodnoty jsou také známy z předchozího integračního kroku.

Naše soustava rovnic (1.3) je nápadně podobná předchozí rovnicí, 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 (6.8) a explicitní verzi pro (6.7) . 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 přesnost implicitní metody. Výsledný mix je metoda, která je jednoduchá na implementaci a relativně přesná pro naše účely.

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

$$x(t+\Delta t) = x(t) + \Delta t.\dot{x}(t+\Delta t) = x(t) + \Delta t.v(t+\Delta t) \tag{6.8}$$

Naše finální soustava (1.3) po použití této metody bude (6.9) pro $i = 1 \dots N$

$$\boldsymbol{v}_i(t+\Delta t) = \boldsymbol{v}_i(t) - \Delta t.\kappa \sum_{j=1, j\neq i}^{n} \frac{m_i}{\left[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right]^3}.\left[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right] \tag{6.9a}$$

$$\boldsymbol{x}_i(t+\Delta t) = \boldsymbol{x}_i(t) + \boldsymbol{v}_i(t+\Delta t) \tag{6.9b}$$

6.1.2 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 6.1

Zdrojový kód 6.1: Semi-implicitní Eulerova integrační metoda (Source/euler.cpp)

```
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 (6.9) 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 (6.9).

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.

6.2 RK4

6.2.1 Teorie

V předchozí sekci jsme implementovali semi-implicitní Eulerovu integrační 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. 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 (6.10), pak RK4 dává numerické řešení ve formě rovnice (6.11). Jedná se o explicitní verzi, ke které existuje ještě varianta implicitní, kterou ale implementovat nebudeme.

$$\dot{\boldsymbol{y}}(t) = \boldsymbol{f}(\boldsymbol{y}, t) \quad \boldsymbol{y}(0) = \boldsymbol{y_0} \tag{6.10}$$

$$\boldsymbol{y}(t + \Delta t) = \boldsymbol{y}(t) + \frac{\Delta t}{6}\left[\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4\right] \tag{6.11}$$

$$\begin{aligned}
\boldsymbol{k}_1 &= \boldsymbol{f}(\boldsymbol{y}(t), t) \\
\boldsymbol{k}_2 &= \boldsymbol{f}(\boldsymbol{y}(t) + \Delta t\frac{\boldsymbol{k}_1}{2}, t + \frac{\Delta t}{2}) \\
\boldsymbol{k}_3 &= \boldsymbol{f}(\boldsymbol{y}(t) + \Delta t\frac{\boldsymbol{k}_2}{2}, t + \frac{\Delta t}{2}) \\
\boldsymbol{k}_4 &= \boldsymbol{f}(\boldsymbol{y}(t) + \Delta t.\boldsymbol{k}_3, t + \Delta t)
\end{aligned}$$

Zkusme tedy tuto metodu aplikovat na naši soustavu (1.3). Použijeme stejný trik jako u Eulerovy metody (6.3) a z jedné rovnice druhého řádu uděláme dvě rovnice první řádu

(6.12).

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{v}(t) \tag{6.12a}$$

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

Ve výše zmíněném popisu RK4 (6.10) se žádná soustava na první pohled nevyskytuje, v zadání je pouze jedna funkce, ale vektorová. Proto definujme následující vektorovou funkci (6.13). Pak vlastně řešíme rovnici (6.14) s neznámou funkcí $\boldsymbol{u}(t)$, na kterou přesně aplikujeme RK4.

$$\boldsymbol{u}(t) := (\boldsymbol{x}(t), \boldsymbol{v}(t)) \equiv (x_x, x_y, v_x, v_y)(t) \tag{6.13}$$

$$\dot{\boldsymbol{u}}(t) = (\dot{\boldsymbol{x}}(t), \dot{\boldsymbol{v}}(t)) = (\boldsymbol{v}(t), -\kappa \sum_{j=1, j\neq i}^{n} \frac{m_i \left[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right]}{\left[\boldsymbol{x_i}(t) - \boldsymbol{x_j}(t)\right]^3}) \tag{6.14}$$

6.2.2 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 6.2: Runge-Kutta integrační metoda (Source/RK4.cpp)

```
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 void RK4::operator()(double step)
14 {
15     //Vypočte koeficient
16     auto computeKx = [&](size_t x, double mult)
17     {
18         x = x - 1;//Číslování od jedničky -  k1,k2,k3,k4
19         //Projdeme všechny dvojice
20         for (size_t i = 0; i < data->size(); ++i)
21         {
22             auto& left = temps[i];
23             for (size_t j = i + 1; j < data->size(); ++j)
24             {
25                 auto& right = temps[j];
26                 auto distLR = dist(left.pos, right.pos);
27                 distLR = distLR*distLR*distLR;
28                 Vec2 dir = left.pos - right.pos;
29                 Vec2 acc = -physicsUnits::G / distLR * dir;
30
31                 //Spočítáme zrychlení
32                 //Které uložíme do koefficientů
33                 kXs[x][i].acc += acc*(*data)[j].mass;
```

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

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 simulace - `temps`.

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 (6.11). A také musíme nastavit dočasné proměnné pro další volání funkce.

Kapitola 7

Implementované viewery

No, tak holt budeme tu angličtinu skloňovat.

7.1 IMGuiViewer

je **viewer**, který vykresluje simulaci do okna. Jeho implementace je rozdělena do relativně velkého množství tříd. Následuje jejich úplný výčet:

třída `IMGuiViewer` hlavní třída, která váže ostatní do funkčního celku.

třída `OpenGLBackend` se stará o inicializaci **OpenGL**.

třída `ImGuiBackend` zajišťuje integraci knihovny **ImGui**, která poskytuje nástroje k tvorbě uživatelského rozhraní.

třídy používající OpenGL :

- **`CircleBuffer`** kreslí kruh o určitém poloměru na určenou pozici.
- **`Shader`** vytváří rozhraní pro tvorbu a použití shaderů.
- **`UnitTrail`** kreslí lomenou čáru.
- **`GLError`** překládá chybové kódy generované OpenGL na vyjímky.

třídy dědící z `Drawer` vykreslují jednotlivé části simulace:

- **`GUIDrawer`** vykresluje pomocí **ImGui** uživatelské rozhraní.
- **`SimDataDrawer`** zobrazuje samotná simulovaná data na obrazovku, používá k tomu `Shader` a `CircleBuffer`
- **`LineTrailsDrawer`** kreslí dráhy simulovaný objektů pomocí `UnitTrail`.

Pojďme si teď některé z nich představit podrobněji.

7.1.1 `OpenGLBackend`

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

7.1.2 `ImGuiBackend`

Jak už bylo psáno, tak tato třída inicializuje knihovnu ImGui. Její implemetance byla také vytvořena na základě implementace uvedené na stránkách knihovny a dokumentací v souboru `imgui.cpp`. Zmínit newframe, render - k čemu jsou a tak...Pak ještě callbacky do GLFLW.

7.1.3 Třídy používající OpenGL

OpenGL je psané v duchu jazyka C, takže neobsahuje objektově orientované prvky. Což je škoda, páč je to nepřehledný. Tyhle třídy se to snaží napravit.

Shader zabaluje shader. Což jsou malé programy určené pro grafickou kartu, které říkají jak má grafická data interpretovat a vykreslit.

CircleBuffer zabaluje VAO,VBO,IBO. Je předpřipravená třída, která vyvolá sadu OpenGL funkcí, které kreslí kruh...

UnitTrail zabaluje VAO,VBO,IBO. Dokáže kreslit lomenou čáru, kde body jsou něco jako fronta- first in, first drawn(and overwritten).

GLError je vyjímka pro OpenGL, protože OpenGL ji samo nemá, tak pokud volání nějaké funkce selže, tak se jen nastaví enum na nějaký error, na což se musíme zeptat. Což dělá funkce za nás a rovnou vyjímku vyhodí. Prostě wrapper na glGetError...

7.1.4 Drawer

abstraktní třída, která něco kreslí, existuje kvůli čistějšímu přístupu k simulaci a třídě ImGuiViewer.

7.1.5 GUIDrawer

Třída s relativně dlouho draw metodou, která pomocí ImGUI kreslí celé uživatelské rozhraní. Což by stálo za to trochu ukázat a popsat.

7.1.6 SimDataDrawer

Kreslí prostě planety jako tečky no.

7.1.7 LineTrailsDrawer

Dokresluje ocásek za planetama aby bylo vidět kde byly. Chtělo by popsat algoritmus jak se stará o VBO a IBO.

7.1.8 IMGuiViewer

By se mohlo postnout celý a vysvětlit na tom zoom, normalized coordinate system, offset, move...

Kapitola 8

Rozšířitelnost a vylepšení

Poznámka autora: *Následující sekce popisuje rozšíření programu o novou funkci a to přehrávání uložených simulací. I když je tato myšlenka plně implementována a je funkční, tak celkem zneužívá design třídy* `Simulation`*, což nebylo nutné. Lepší verze je vložena jako sekce 8.2 Praktický příklad rozšíření programu V2.0, ale rozhodl jsem se zde ponechat i původní verzi.*

8.1 Praktický příklad rozšíření programu V1.0

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. Bohužel program v aktuální verzi neprovádí žádné cachování výsledků, takže se nelze vrátit zpět. Což je určitě užitečné 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, zastavení a zpětné přehrávání. Jak by se takový přehrávač dal implementovat? Co kdybychom vytvořili následující třídy:

1. Třída `ViewAndRecord`, která se chová jako `viewer`, ale navíc simulaci zaznamenává do souboru.
2. Trojice tříd `ReplayerParser` `ReplayerMethod` a `ReplayerViewer`, které by se starali o přehrávání simulace.

8.1 ukazuje výše zmíněnou myšlenku převedenou do C++ a 8.2 pak jak by se pak dala použít v hlavním programu.

Zdrojový kód 8.1: Návrh přehrávače simulací (Source/extIdea.cpp)

```
1
2  template<typename someViewer>
3  class ViewAndRecord :public Viewer{
4  public:
5      template<typename... someViewerArgs>
6      ViewAndRecord(std::string outFile, someViewerArgs&&.. args):
7          outFile(outFile),viewer(std::forward<soViewerArgs...>(args...))
8      void operator()(){
```

```
9           //Necháme simulaci přehrávat libovolným způsobem
10          viewer();
11          //Ale poznamenáme si aktuální data, včetně aktuálního času do souboru
12          AddToFile(simTime, data);
13      }
14      //Prepare metoda
15  private:
16      void AddToFile();
17      someViewer viewer;
18      std::string outFile;//Soubor kam budeme provádět záznam
19  };
20
21  class ReplayerParser:public Parser
22  {
23  public:
24      ReplayerParser(std::string inFile)
25      simData_t Load();
26  };
27
28  class ReplayerMethod :public SimMethod
29  public:
30      ReplayerMethod(std::string inFile);
31      void operator()(time deltaT){
32          simTime += deltaT;
33          //Načteme data ve správný čas
34          data = GetDataFromFile(simTime);
35      }
36      void
37  private:
38      void GetDataFromFile();
39      time simTime;
40  };
41
42  class ReplayerViewer :public solar::Viewer{
43  public:
44      template<typename... someViewerArgs>
45      ReplayerViewer(ReplayerMethod*);
46      void operator()(){
47          //Necháme vykreslit uživatelské rozhraní
48          viewer();
49          //Dokreslíme si potřebné ovládání pro přehrávání
50          DrawReplayControls();
51      }
52  private:
53      void DrawReplayControls();
54      solar::IMGuiViewer viewer;
55      ReplayerMethod* method;
56  };
```

Zdrojový kód 8.2: Příklad použití přehrávače (Source/extIdeaUsage.cpp)

```
1   int main()
2   {
3       //Nejdříve simulaci zaznamenáme
4
5       auto parser = std::make_unique<FormattedFileParser>("vstup.txt");
6       auto method = std::make_unique<RK4>();
7       //auto viewer = std::make_unique<IMGuiViewer>(1200, 700, "Title");
8       auto viewer = std::make_unique<ViewAndRecord<IMGUIViewer>>("zaznam.txt",
        1200, 700, "Title");
```

```
9
10      Simulation record(std::move(parser), std::move(method), std::move(viewer));
11      //Simulaci klasicky pustíme a nikdo ani nepozná, že je zaznamenávána
12      record.Start(/*Parametry simulace...*/);
13
14      //Pak ji můžeme přehrát
15      parser = std::make_unique<ReplayerParser>("zaznam.txt");
16      method = std::make_unique<ReplayerMethod>("zaznam.txt");
17      viewer = std::make_unique<ReplayerViewer>(/*...*/);
18
19      Simulation replay(std::move(parser), std::move(method), std::move(viewer));
20
21      replay.Start(10ms);
22      return 0;
23  }
```

Díky návrhu celého programu 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. Zabalení znamená, že ho předáme jako parametr pro šablonu `ViewerAndRecord`. Při běhu simulace bude pak volán `ViewerAndRecord`, který ale také zavolá předaný viewer a navíc bude na pozadí ukládat probíhající simulaci.

Přehrávání pak docílíme tím, že simulaci budeme podstrkovat data, která si přečteme ze souboru místo toho abychom je simulovaly. 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. Pro zobrazení použijeme třídu `ReplayerViewer`, který využívá pracně vytvořený `IMGuiViewer`. Pomocí knihovny ImGui můžeme pak dokreslit potřebné ovládání simulace.

Bohužel narazíme na problém. `Simulation` neumí měnit libovolně čas. `deltaT` je pevně dané, takže nemůžeme simulaci přehrávat pozpátku. Což se dá vyřešit tím, že čas simulace budeme do jisté míry ignorovat a `ReplayerViewer` si přímo řekne `ReplayerMethod`, že by měl nyní vracet data pozpátku.

8.2 Praktický příklad rozšíření programu V2.0

templatize SystemUnit

8.3 Vylepšení

- Jak by se dal program vylepšit
-

Kapitola 9

Porovnání integračních metod

V této kapitole alespoň zběžně porovnáme implementované algoritmy. Možná to nepatří přímo k programátorské dokumentaci, ale potencionálního uživatele by mohlo zajímat, kterou integrační metodu by měl použít, zvláště by nás mohl zajímat poměr přesnost/rychlost. Tedy srovnání rychlostí a přesností jednotlivých metod.

Kapitola 10

Uživatelská příručka

Jak program spustit, organizace programu - kde je uloženej a tak... Dodatek k linuxu asi...

Tato dokumentace spolu se zdrojovými kódy programu je dostupná na:
`https://bitbucket.org/Quimby/solar/src`
Popřípadě přímo stažitelná pomocí git:
`https://Quimby@bitbucket.org/Quimby/solar.git`
V případě problému mne můžete kontaktovat na adrese: waltl.jan@gmail.com