

# Simulace Sluneční soustavy

Dokumentace k zápočtovému programu pro předmět

Programování II(NPRG031)

Jan Walzl  
5. září 2017

# Obsah

<b>1</b>	<b>Programátorská část</b>	<b>1</b>
1.1	Matematická knihovna . . . . .	1
1.2	Kamera . . . . .	2
1.2.1	LookAt . . . . .	2
1.2.2	Ovládání kamery . . . . .	3
1.3	GLEW → GLAD . . . . .	3
1.4	Fyzikální jednotky . . . . .	3
1.5	Reorganizace ImGuiViewer . . . . .	3
1.6	Vykreslování simulace . . . . .	4
1.6.1	SimDataDrawer . . . . .	4
1.6.2	GridDrawer . . . . .	6
1.7	Nové uživatelské rozhraní . . . . .	7
1.8	Vylepšení a opravy . . . . .	8
1.9	TODO . . . . .	9
<b>2</b>	<b>Kompilace programu</b>	<b>10</b>
2.1	Git . . . . .	10
2.2	Windows . . . . .	10
2.3	Unix . . . . .	10
<b>3</b>	<b>Uživatelská příručka</b>	<b>11</b>
3.1	Požadavky . . . . .	11
3.2	Základní ovládání . . . . .	11
3.3	Pokročilé možnosti . . . . .	17

# Organizace dokumentu

Tento text je organizován do následujících částí:

**Programátorská část** - Dokumentuje veškerou práci, která proběhla od zimního semestru ve formě changelogu. Jsou zde popsány hlavní změny v programu a také nově přidané prvky.

**Kapitoly 2 a 3** popisují jak program zkompileovat a také jak s ním pracovat z neprogramátorského pohledu.

Před přečtením doporučuji přečíst alespoň úvodní dvě kapitoly zimní dokumentace pro seznámení se s celkovou architekturou programu.

Všechny odkazy uvedené v tomto textu jsou funkční k 4. září 2017.

# Kapitola 1

## Programátorská část

Cíl pro letní semestr byl přenést simulaci do 3D a také vylepšit uživatelské rozhraní se kterým sem nebyl moc spokojený. Drtivá většina práce je tedy ve třídách spojených s `ImGuiViewer` neboť to je momentálně hlavní grafický výstup programu. Níže jsou uvedeny jednotlivé věci, které prošly změnami a podrobněji se jim věnují další části této kapitoly. Pár věcí je ze seznamu rozšíření ze zimní dokumentace, ale i zde se nestihlo vše co by stálo za to implementovat. Takže na konci této kapitoly je seznam budoucích rozšíření/oprav.

1. Předělána matematická knihovna a nově jsou podporovány generické 2D,3D,4D vektory a také 4D matice.
2. Vytvořena kamera (třída `Camera`), která nahradila třídu `OMSAR`. Kamera slouží k orientaci a pohybu v prostoru, k čemuž využívá především matic.
3. Knihovna **GLEW** nahrazena knihovnou **GLAD**
4. Byla implementována podpora fyzikálních jednotek. Takže každá třída, která má přístup k datům si může zjistit v jakých jsou jednotkách a nemusí se spoléhat na konvence v komentářích.
5. Reorganizace `ImGuiViewer`, kdy jeho podtřídy byly rozděleny na vykreslování uživatelského rozhraní a simulace.
6. Kompletní přepsání kódu s uživatelským rozhráním.
7. Vykreslování simulace je nyní plně ve 3D díky podpoře kamery včetně pár nových grafických prvků.

### 1.1 Matematická knihovna

Návrh ze zimního semestru mi nepřišel moc šťastný, protože měl pevně zafixovaný datový typ na `double`. Což je sice dostačující pro samotnou simulaci, ale **OpenGL** má radši `float` a **ImGui** využívá pro kreslení UI `float` a `int`. Což si v důsledku vyžádalo zbytečné přetypování. Nová knihovna nabízí generické vektory, takže lze jednoduše použít `Vec3<double>` pro pozice simulovaných objektů, `Vec3<float>` pro potřeby OpenGL renderování. Nové generické matice využívá hlavně kamera při změně souřadnicových systémů při renderování

scény. Rychlý úvod do vektorů a matic pro potřeby 3D grafiky nabízí [3]. Knihovna také rovnou obsahuje funkce pro perspektivní, ortogonální projekci, posunutí, otočení a škálování. Jejich implementace čerpá z [1, 2, 3], kde jsou jejich vzorce intuitivně odvozeny, projekční matice jsou ještě upraveny pro potřebu OpenGL, což je popsáno v sekci 1.6.1.

## 1.2 Kamera

Ve 2D stačilo pro globální pozici kamery definovat posunutí a škálování neboť byla kamera zafixována pro pohled ze shora bez možnosti rotace. Ve 3D je to vyřešené pomocí matic.

Kamera je reprezentována abstraktní třídou *Camera*. Její orientace je dána pozicí kamery, sledovaného cíle a vektorem, který ukazuje vzhůru. Z toho je pomocí metody *LookAt()*, která má předchozí 3 argumenty, postavena *view* matice, která převádí objekty z globálních souřadnic (*world space*) do souřadnic kamery (*view space*). Dále je zde projekční matice, která převádí z *view space* do normalizovaných souřadnic (*clip space*), po vydělení čtvrtou souřadnicí *w* mají viditelné objekty souřadnice *x*, *y*, *z* v rozsahu  $-1.0, 1.0$ . S konvencí  $+x$  vpravo,  $+y$  nahoru a  $-z$  do obrazovky. Jak je uvedeno v sekci 1.6.1 tak je lepší z přemapovat do  $\pm 1.0, 0.0$ . Souřadné systémy nenáročně vysvětluje [4].

### 1.2.1 LookAt

Metoda má prototyp *LookAt(Vec3d poziceKamery, Vec3d poziceCile, Vec3d upVec)*, můžeme tedy ještě rovnou zavést *smerKamery* jako *poziceCile-poziceKamery*.

Předchozí sekce nám říká, že po transformaci *view* a projekční maticí by směr kamery měl být  $(0, 0, -1)$ , směr vzhůru  $(0, 1, 0)$  a její pozice  $(0, 0, 0)$ . První dva vektory totiž perspektivní ani pravoúhlá projekce nemění, projekční matice ale definují *near, farplanes*, které jsou dány relativně vůči pozici kamery a definují rozsah viditelné hloubky. Proto je pozice v počátku. Ještě dodefinujeme *rightVec* jako vektorový součin *smerKamery* a *upVec*, tedy vektor ukazující napravo od kamery. Poté ještě přepočítáme *upVec* jako vektorový součin *rightVec* a *smerKamery*. Tím dostaneme 3 ortogonální, po znormalizování ortonormální, vektory.

Hledanou *view* matici můžeme sestavit ve dvou krocích ( Stejně je to popsáno v [5], ale orientace kamery není podle mě moc zdůvodněna), nejdříve je potřeba správně přesunout pozici kamery do počátku. To lze udělat následující maticí translace *T* ( $T \times \text{poziceKamery}$  při rozšíření o *poziceKamery.w = 1* opravdu dává počátek)

$$T = \begin{pmatrix} 0 & 0 & 0 & -\text{poziceKamery.x} \\ 0 & 0 & 0 & -\text{poziceKamery.y} \\ 0 & 0 & 0 & -\text{poziceKamery.z} \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.1)$$

Nyní potřebujeme ještě kameru natočit správným směrem. K tomu lze použít lineární algebru, můžeme si uvědomit, že otočení v tomto případě znamená změnu báze  $B_1 = \{\text{rightVec}, \text{upVec}, \text{smerKamery}\}$  na bázi  $B_2 = \{(1, 0, 0), (0, 1, 0), (0, 0, -1)\}$ . Jednodušší je vyjádřit matici přechodu o  $B_2$  k  $B_1$ , protože to stačí dát bázi  $B_1$  do sloupců matice a otočit znaménko u směru kamery neboť  $B_2$  je skoro kanonická báze. Hledaná matice otočení *O* je pak její inverze, což je v tomto případě pouze transpozice, protože je ortonormální. Výsledná

matice  $O$  po rozšíření na  $4 \times 4$  je uvedena v 1.2.

$$O = \begin{pmatrix} rightVec.x & rightVec.y & rightVec.z & 0 \\ upVec.x & upVec.y & upVec.z & 0 \\ -smerKamery.x & -smerKamery.y & -smerKamery.z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.2)$$

Hledaná *view* matice je pak pouze  $view = O \times T$ . Výchozí implementace v `Camera.cpp` pracuje s otočením směrem kamery, proto tam není opačné znaménko a také jsou obě matice už vynásobené.

### 1.2.2 Ovládání kamery

Je implementováno pomocí ovládáním myši, kdy táhnutím myš otáčí myšlenou jednotkovou kouli umístěnou uprostřed obrazovky, tento pohyb se přenáší na otáčení kamery kolem svého cíle. Pro výpočet stačí myšlenku otočit a představit si, že myš se pohybuje po statické kouli, pak k výpočtu stačí pouze její počáteční a koncová pozice. Tyto pozice se převedou na vektory ze středu koule do obou bodů. Cosinus úhlu mezi oběma vektory je skalární součin (neboť mají délku 1) a vektorový součin dává osu rotace. Pak stačí využít matici rotace okolo dané osy a tu aplikovat na kameru. Pokud náhodou myš neleží na kouli (typicky rohy obrazovky), tak se její pozice definuje jako nejbližší na kouli.

Implementaci zajišťuje třída `CameraControls`. Navíc je k dispozici přibližování a oddalování kamery a také otáčení okolo své pozice nebo pohyb do stran.

## 1.3 GLEW → GLAD

Toto není moc velká změna, ale GLAD dovoluje přesně stanovit, které rozšíření bude mít a četl jsem, že v GLEW jsou neopravené chyby. Také je přímo vložena do projektu (jsou to jen dva soubory) a tím odpadla jedna ze dvou statických knihoven.

## 1.4 Fyzikální jednotky

Veškeré informace o simulovaných datech jsou uloženy ve třídě `SimData`. Ta ovšem neobsahovala žádnou informaci o tom v jakých jednotkách tyto data jsou a spoléhala se, že si to moduly nějak dohodnou mezi sebou. Což bylo jen formou komentářů v kódu.

Nyní byla vytvořena třída `PhysUnits`, která obsahuje jednotky délky, času a hmotnosti a dovoluje mezi nimi převádět. `SimData` pak používá instanci této třídy k implementaci různých funkcí pro převody hodnot v simulovaných datech. Navíc kdokoli kdo dostane přístup k datům si může zjistit v jakých jsou jednotkách a podle toho se zařídit. Jednotlivé moduly se už nemusí domlouvat a každý si může pracovat ve vlastních jednotkách.

## 1.5 Reorganizace `ImGuiViewer`

Tato třída je nyní jen jednoduchý wrapper okolo 4 tříd:

**OpenGLBackend** inicializuje OpenGL, GLFW, GLAD a vytváří okno pro kreslení.

**ImGuiBackend** inicializuje knihovnu ImGui

**GUIDrawer** Vykresluje uživatelské rozhraní.

**SceneDrawer** Vykresluje 3D simulaci.

## 1.6 Vykreslování simulace

Vykreslování celé simulace organizuje třída **SceneDrawer**. Využívá k tomu následující třídy:

**Camera, PerspectiveCamera, ScaledOrthoCamera** První třída je známá abstraktní třída z dřívější sekce. Zbylé dvě jsou jejich konkrétní implementace s příslušnými projekcemi. **SceneDrawer** se o obě třídy stará a dává je k dispozici ostatním částem programu.

**CameraControls** ovládá kameru pomocí myši, které bylo také popsáno v dřívější sekci. Ovládání je vysvětleno v uživatelské příručce.

**LineTrailsDrawer** Tato třída je už známá ze zimního semestru a zůstala skoro nezměněná až na 2D→3D pozici. Vykresluje historii pohybu simulovaných objektů.

**SimDataDrawer** Se stará o vykreslování samotných objektů. Zde už bylo více změn, které jsou psány níže.

**GridDrawer** Zcela nová třída, která vykresluje dynamickou 2D mřížku pro lepší orientaci v prostoru. Také nabízí projekci objektů na mříž ve formě třídy **PinHeads**. Její návrh je také vysvětlen níže.

### 1.6.1 SimDataDrawer

Nejdřív uděláme menší odbočku k OpenGL, protože vykreslování takových vzdáleností ve správném měřítku se neobešlo bez problémů.

**Depth buffer** Je buffer, ve kterém je uložena hloubka objektů ve scéně. Při kreslení se tento buffer využívá k určení, které objekty mají být viditelné a které jsou schované za jinými. Většinou se jedná o 24bitový nebo 32bitový formát s fixed-point desetinnými čísly. Jejich rozsah může být dostatečný pro klasické aplikace, ale pro sluneční soustavu je nedostačující a dochází k *z-fighting*, neboť program není schopen rozlišit hloubku mezi objekty a správně je vykreslit. Hloubka není v bufferu uložena lineárně, ale větší rozsah je pro blízké objekty a zbylá část pro vzdálené, což je dobrá myšlenka neboť u vzdálených objektů stejně nebudou malé detaily vidět. Dobrý náhled toho jak jsou hodnoty uloženy a jak depth buffer funguje v OpenGL a co je *z-fighting* je [6]

Limitující faktor je nastavení rozsahu *near, far plane* v projekční matici. Pro vyšší přesnost by tento rozsah měl být co nejmenší. Což ale znamená, že pro zachování přesnosti se do scény velmi vzdálené objekty prostě nevměstnají. Jednou z technik, kterou se dá docílit dostatečné přesnosti je, že se objekty rozdělí na blízké a vzdálené. Nejdříve se vykreslí vzdálené s *nearplane* posunutým co nejdál a *farplane* za všemi objekty. Poté se buffer vymaže a vykreslí se blízké objekty s *nearplane* blízko u kamery a *farplane* za blízkými objekty. Nevýhoda je, že se musí udržovat seznamy objektů dle vzdáleností, které se s pohybem kamery budou měnit.

Tato technika by v našem případě fungovala, protože moc objektů nebudeme kreslit a není problém je tedy za běhu řadit. Lepší řešení je ale použít *floating point* čísla, které dokáží uložit velké rozsahy hodnot. Zde, ale nastává problém pro OpenGL, protože Z je v

rozsahu  $-1.0, +1.0$ , tak nám *float* nepomůže, protože blízko  $\pm 1.0$  je exponent pevně daný a dostáváme přibližně stejné rozlišení jako *fixed-point* čísla. Toto platí pro libovolné číslo kromě 0, čehož využívá následující technika (*reversed depth buffer*) popsána v [8]. Místo rozsahu  $-1.0, +1.0$  použijeme rozsah  $1.0, 0.0$ . Blízko kamery bude přesnost zachována díky nelinearitě uložení a vzdálené objekty ji neztratí díky přesnosti *float* čísel blízko 0. Podobná technika je popsána i v [7], která zvyšuje přesnost použitím logaritmů a je o něco přesnější v případě 32bit formátu.

Rozhodl jsem se implementovat první techniku neboť nepotřebuje žádný zásah do shaderů. Ale na druhou stranu jsou potřeba rozšíření do OpenGL( `GL_ARB_clip_control` nebo `GL_NV_depth_buffer_float`), popřípadě OpenGL 4.3+. Ovšem i starší grafiky by s tím neměly mít problém, dle [9] jsou obě rozšíření celkem rozšířená. Navíc jak popisuje článek [8], tak je potřeba pozměnit projekční matice právě protože se Z mapuje jinak. Zde se s článkem trochu rozcháším, protože používám jinou projekční matici než je tam uvedena. Detaily změn jsou ve zdrojových kódech. Výsledek je ovšem stejný a k srovnání přesností se dá použít interaktivní graf v článku. Je nutné přepnout formát na *float* a pak je vidět obrovský rozdíl mezi *Standard* a *Reversed*. Popřípadě je dobrá webová aplikace [10](napsána v WebGL), která ukazuje rozdíl mezi klasickým a logaritmickým depth bufferem. Pro větší vzdálenosti klasická verze naprosto selhává.

Díky této technice není problém nastavit fixně *near plane* na 1 metr a *far plane* na 100 světelných let  $\approx 10^{18}$  metrů bez jakýkoliv problémů. Její implementace je dostatečně popsána v samotném článku [8], ale bylo čerpáno také z [11].

**Limity floating-point čísel** V minulé části jsme vyřešili přesnost použitím *floating-point čísel*, ovšem i tato čísla mají limity. Konkrétně 32bitový *float* má přesnost přibližně 7 platných číslic a 64bitový *double* 15. Pokud tedy bude mít objekt souřadnice  $(1m, 0, 0)$  tak v okolí nám *float/double* dává přesnost přibližně  $0, 1\mu m/1fm$ , což naše simulace určitě nikdy nevyužije. Co se stane když náš objekt bude v okolí Pluta? Tedy souřadnice  $(7 \times 10^{12}m, 0, 0)$ , pak nám *float/double* dává přesnost  $700km, 7mm$ . To už je mnohem horší, Pluto má poloměr přibližně 1000km, takže ho ani *float* nedokáže přesně reprezentovat. Bohužel platí, že čím dál jsme od počátku souřadnic, tím to bude horší. Více nejen o přesnosti *floating-point čísel* nabízí tento blog [12]. V našem případě bylo zvoleno řešení už v zimě a to použít 64bit *double*, jeho přesnost je o něco méně než metr ve vzdálenosti jednoho světelného roku od počátku.

Nyní ale nastává problém, že OpenGL a grafické karty mají radši 32bitová čísla, takže jednoduché řešení, že počátek umístíme do středu soustavy (cca Slunce) nám v případě Pluta nevykreslí hezkou kulatou planetku, ale něco co má do toho hodně daleko. Řešení, které jsem zvolil je, že počátek pro renderování nebude ve středu soustavy, ale na pozici kamery. Tím cokoliv co je blízko kamery bude i blízko počátku. Toto je možné jednoduše díky tomu, že přesné pozice, včetně kamery, zajišťuje použití *double*, před posláním dat na grafiku stačí od pozic objektů odečíst pozici kamery, výsledek lze pak celkem bezpečně přetypovat na *float*. Při použití této techniky je ale v shaderech naopak potřeba ignorovat posunutí objektů vůči kameře pomocí *view* matice, což se udělá jednoduše vynulováním 3. sloupce matice viz. odvození matice v sekci 1.2.1.

Tato technika je implementována pro všechny následující třídy, ale nepoužívá ji `LineTrailsDrawer`. Což je vidět při přiblížení např. k Plutu, že čára za pohybem různě přeskakuje. Tahle podobně by skákaly všechny vertexy Pluta a nevypadá to moc hezky.



**SimDataDrawer** Nyní už k samotnému kreslení, všechny simulované objekty mají také svůj poloměr a je možné je vykreslit jako barevné 3D koule, což se také přesně děje. Ke kreslení využívá třídu **Sphere**, která vytvoří kouli a potřebná data pro OpenGL.

Ovšem oddálení tak abychom viděli přibližně celou soustavu, zjistíme, že kromě Slunce nic nevidíme. Proto třída nabízí i vykreslování objektů s poloměrem v souřadnicích obrazovky, čili, že každý objekt bude tak velký aby byl vidět bez ohledu na jeho skutečnou velikost. Uživatelské rozhraní pak nabízí přepínání mezi oběma režimy. Korekce probíhá na základě hloubky objektu na obrazovce, neboť pokud je objekt dvakrát dále, tak je dvakrát menší, proto stačí poloměr právě jeho hloubkou vynásobit a tím bude nový poloměr invariantní vůči jeho hloubce. Malý detail je jednoduché osvětlení objektů Sluncem.

### 1.6.2 GridDrawer

Pokud bychom kreslili pouze samotné objekty, tak díky velkým vzdálenost a 360° pohybu kamery může člověk snadno ztratit orientaci, proto byla vytvořena tato třída která kreslí 2D mříž v prostoru a poskytuje tak orientační bod. Veřejné rozhraní také dovoluje zvolit přesnou pozici, velikost a orientaci mříže. Navíc nabízí zapnutí/vypnutí projekce jednotlivých objektů na mříž a spojení projekce s původním objektem, tím je hned vidět, kde se objekty v prostoru nachází vzhledem k ostatním.

**Grid** je třída, která se stará o samotná data mříže (jednotlivé čáry) pro OpenGL. Kreslení pak organizuje **GridDrawer** pomocí dvou instancí této třídy. Dvě instance - větší a menší - nám dovolují dynamickou změnu velikosti mříže. Takže bez ohledu na to jak daleko jsme od objektů, vždy vidíme mříž s dostatečným rozlišením ( které je zobrazeno v uživatelském rozhraní ).

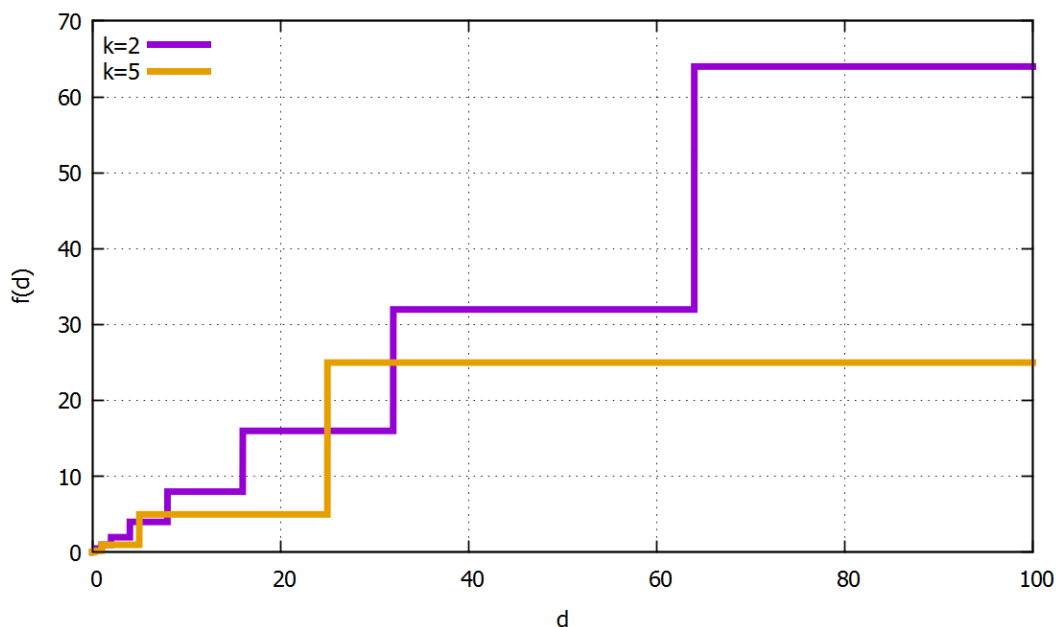
<sup>1</sup> Implementace by tedy měla zajistit, že pokud dostatečně oddálíme kameru, tak malé čtverečky se skokově změni na větší a objeví se ještě větší a naopak. To přesně umožňují tyto dvě instance - menší se zvětší na větší a větší se také zvětší. Pokud bude tato změna správně napsaná, tak si uživatel ani ničeho nevšimne. Pro obě kamery **PerspectiveCamera** a **ScaledOrthoCamera** platí, že když ji desetkrát oddálíme, tak toho vidíme desetkrát více, nebo-li objekty se desetkrát zmenší. Pokud je tedy větší mříž  $k$ -větší než menší, tak je potřeba najít funkci  $f(d)$ , kde  $d > 0$  je vzdálenost kamery. Pro kterou platí, že když se  $d$  zvětší  $k$ -krát, tak se také  $f(d)$  zvětší  $k$ -krát a zároveň je vždy na intervalu  $(x_i, kx_i)$  konstantní, kde  $x_i$  jsou dána níže. Druhá podmínka je zde proto, aby se mříž změnila skokově a uživatel měl pocit, že mříž je součástí světa. Taková funkce je například 1.3, její hodnoty jsou vykresleny v grafu 1.1 pro  $k = 2, 5$

$$f(d) = k^{\lfloor \log_k d \rfloor} \quad (1.3)$$

Kde " $\lfloor \cdot \rfloor$ " je dolní celá část. Bez ní by se jednalo o identitu  $f(d) = d$ . Toto zaokrouhlení zajistí konstantnost na intervalech pro  $(k^i, k^{i+1})$ ,  $i \in \mathbb{Z}$ . Pro úplnost tedy  $x_i = k^i$ . Tuto funkci tedy využívá **GridDrawer** k správnému nastavení velikostí obou mříží **Grid**. Také na základě toho kde se v konstantním intervalu kamera vrovna nachází **GridDrawer** zajistí správné obarvení obou mříží, čímž se docílí hladkého přechodu. Stejně jako u **SimDataDrawer** se ze stejných důvodů pro pozici mříže používají relativní souřadnice vůči kameře, navíc je pozice upravena tak, aby byla mříž vždy ve středu obrazovky.

<sup>1</sup>Před přečtením tohoto odstavce doporučuji pro lepší pochopení nejdříve program pustit a podívat se na výsledný pohyb mříže.

Obrázek 1.1: Graf funkce 1.3



**PinHeads**<sup>2</sup> je třída, která se stará o vykreslení projekcí objektů na mříž k čemuž jí stačí znát pozici všech těchto objektů a samotné mřížky, poté pro každý objekt vykreslí (pomocí *geometry shaderů* OpenGL) projekci i spojení s objektem. Pro pozice také používá relativní souřadnice vůči kameře.

## 1.7 Nové uživatelské rozhraní

Uživatelské rozhraní bylo kompletně přepsáno. Jeho hlavní třídou je **GUIDrawer**, který vykresluje horní, spodní lištu a popřípadě všechna okna implementovaná v následujících třídách:

1. **UnitsProperties** - Seznam všech simulovaných objektů a jejich fyzikálních vlastností.
2. **Graphs** - Dovoluje vytvářet grafy různých fyzikálních veličin - vzdálenost, rychlost, energie objektů.
3. **ObjectContextMenu** - Najetím/Kliknutím myši na objekty se objeví toto okno, které ukazuje jméno objektu a dovoluje objekt sledovat nebo se k němu přiblížit.
4. **VisualPreferences** - Okno s různými nastaveními grafiky a kamery.
5. **SimProperties** - Zobrazuje detaily o průběhu simulace.

Detaily implementace těchto tříd by měly být jasné při pohledu do zdrojových kódů, které hlavně využívají knihovnu **ImGui**.

<sup>2</sup>Protože to spolu s vykreslením objektů jako koulí vypadá jako špendlíková hlavička zapíchnutá v mřížce.

## 1.8 Vylepšení a opravy

Stejně jako v zimě jsem si průběhu práce na programu vedl v souboru plán toho, na čem budu pracovat, časem se tam objevily různé nápady, které se do programu hlavně z časových důvodů nevešly, proto je zmíním alespoň zde:

### Zaznamenávání a přehrávání simulace

S tímto tématem jsem od zimy vůbec nepohnul až na pár oprav aby zůstalo funkční. Ale myslím si, že je užitečné a stálo by za to ho dodělat. Pořád platí, že by se mělo ukládat v `simMethod`, že lineární interpolace je nic moc a že uživatel to může také celé rozbít změnou rychlosti simulace pomocí tlačítek, které jsou zapnutá. S tím souvisí další bod.

**Traits pro moduly** Momentálně může každý modul měnit vlastnosti simulace, případně přidávat/mazat objekty za chodu bez ohledu na to, zda to ostatní moduly zvládnou, jakože momentálně spíš nezvládnou. Můj návrh by byl pro každý modul definovat specializaci třídy `SimTraits`, která by v sobě držela podmínky, které modul vyžaduje. Pak by mělo stačit aby konstruktor `Simulation` používal šablony (*templates*) a ověřil při kompilaci zda jsou jednotlivé moduly navzájem kompatibilní. Navíc by to mohlo vést k tomu, že by třída `SimData` mohla obsahovat různé proměnné(nebo pointer na nějaké `ExtraInfo`) právě dle požadavků modulů místo fixní barvy a poloměru, které se simulací nespojují.

**LineTrails** by měla používat relativní souřadnice. Ale bylo nutné držet pozice v *double* a kopírovat relativní pozice do vertex bufferu každý frame, což je nic moc. Proto by bylo lepší zavést něco jako strom souřadnic - třeba s *octree*, kde každý blok by si počítal souřadnice vzhledem ke svému počátku. Pokud by spodní bloky s objekty byly dostatečně malé, tak by na ně stačil *float*. Ale to za přesnější čáry za objekty nestojí.

**Lepší ovládání kamery** Kamera se momentálně ovládá pouze myší, chtělo by to přidat ovládání i pomocí klávesnice. Také by stálo za to přidat možnost zafixování pozice kamery relativně k danému objektu.

**Vylepšit grafy** Nyní je to taková verze 0.8, kdy většina věcí funguje, ale je potřeba doladit vizuální mouchy a ovládání je takové nic moc = neintuitivní. Například by bylo lepší použít stejný algoritmus pro mřížku jako `GridDrawer` a nějaký ucelený systém jednotek.

**Keplerovy zákony** Nebylo by špatné vidět implementaci Keplerových zákonů a případně srovnat výstup s numerickými metodami.

**Launch window** S trochou snahy by nemělo být těžké aby aplikace po spuštění vytvořila malé okno s nastavením simulace a poté ji spustila, bylo by to hezčí než používat příkazovou řádku (ale ta by také zůstala). Navíc vše pro tvorbu okna už je implementované v `ImGuiViewer`.

**Hezčí vzhled** je spíš můj osobní a hodně vzdálený cíl, aby Sluneční soustava vypadala realisticky. Což by obnášelo použít textury pro planety, přidat simulaci atmosféry<sup>3</sup> a také přidat třeba skybox s hvězdami nebo prstence kolem planet a hezčí osvětlení.

<sup>3</sup> <http://www-evasion.imag.fr/people/Eric.Bruneton/> - *Atmosphere* - Moc tomu ještě nerozumím, ale vypadá to hezky.

## **1.9 TODO**

- Celou dokumentaci commitnout do rep. - Zkusit build na linuxu - clone do nové složky a pustit buildUbuntu/ a čistej build. Oboje otestovat. - otestovat funkčnost programu. - Otestovat různá rozlišení a co to dělá.
- Odevzdat.

## Kapitola 2

# Kompilace programu

### 2.1 Git

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

<https://bitbucket.org/Quimby/solar/src>

Popřípadě přímo stažitelná pomocí git HTTPS(2.1) nebo SSH(2.2):

<https://Quimby@bitbucket.org/Quimby/solar.git> (2.1)

<git@bitbucket.org:Quimby/solar.git> (2.2)

### 2.2 Windows

Ve složce buildVS/ je předpřipravený .sln projekt pro Visual Studio 2015 Community Edition, který by měl obsahovat vše potřebné pro správné zkompileování. Druhou možností je stejně jako na Unix vytvořit si vlastní projekt pomocí programu **CMake**.

### 2.3 Unix

Ke kompilaci je potřeba mít nainstalován balíček **xorg-dev**(vyžaduje ho GLFW, viz. [13]). Dále jsou potřeba programy **CMake**, **make** a nějaký kompilátor, který **CMake** pozná - např. **g++**. Poté by mělo stačit pustit následující sekvenci příkazů:

```
1 cd <složka se stazenými soubory(obsahuje CMakeLists.txt)>
2 mkdir build
3 cd build/
4 #případně Debug
5 cmake .. -DCMAKE_BUILD_TYPE=Release
6 make
```

Což by mělo vytvořit složku build/bin/ s potřebnými soubory. Také zde je připravena složka buildUbuntu/ která obsahuje předem vytvořený makefile, ale nevím kde všude bude fungční.

## Kapitola 3

# Uživatelská příručka

### 3.1 Požadavky

Vyžaduje verzi OpenGL 3.3 nebo vyšší a rozšíření `GL_ARB_clip_control` nebo `GL_NV_depth_buffer_float`. Jinými slovy by program měl fungovat na většině dnešních počítačů s aktuálními grafickými ovladači( výjimkou by mohli být starší integrované grafiky od Intelu). Ke spuštění `.exe` na Windows je navíc potřeba balíček Microsoft Visual C++ 2015 Redistributable - aktuální verze na <https://www.microsoft.com/cs-CZ/download/details.aspx?id=53840>. Také v případě grafického výstupu(který se zapne jako výchozí) je doporučená velikost okna minimálně 1000x500 pixelů.

### 3.2 Základní ovládání

Při spuštění se program otevře v výchozím grafickém režimu, kde dojde k nahrání zabudované Sluneční soustavy. Simulace se ovládá pomocí grafického rozhraní, které je popsáno na následující okomentované sadě obrázků. Snaha byla, aby tato sekce nemusela existovat a uživatelské rozhraní bylo dostatečně intuitivní samo o sobě. Proto je šance, že při najetí myši na většinu grafických prvků se zobrazí doplňující popis.

Ovládání myši:

**levé tlačítko** otáčení kamery okolo svého cíle

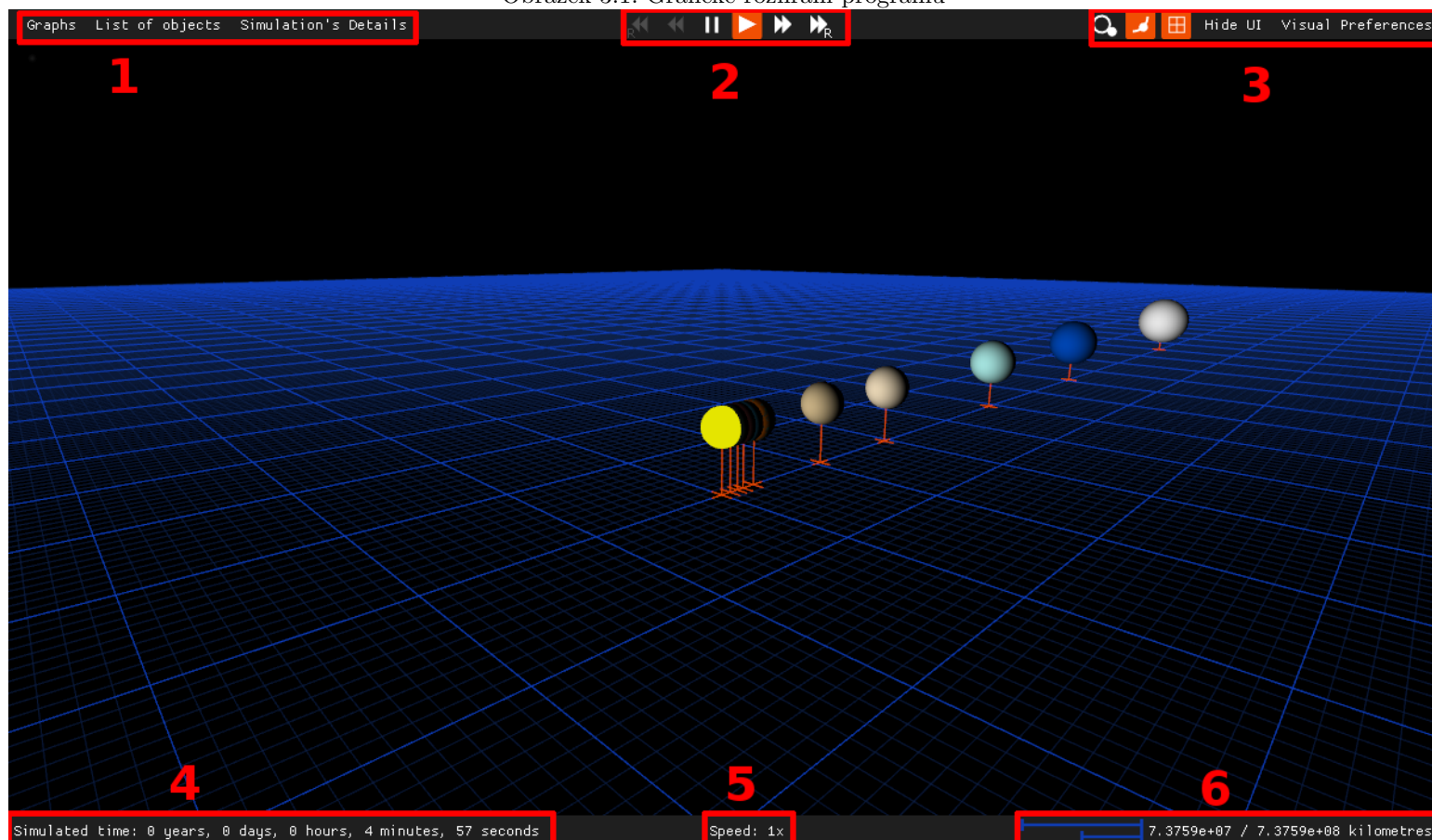
**pravé tlačítko** pohyb kamery do stran

**ALT + levé tlačítko** rozhlížení do stran

**Kolečko** přibližování a oddalování cíle

**ALT + kolečko** pohyb dopředu/dozadu.

Obrázek 3.1: Grafické rozhraní programu



1. Tlačítka s jednotlivými okny. Zleva - grafy, seznam simulovaných objektů, detaily simulace

2. Ovládání simulace - zpomalení/zrychlení, zvětšení/zmenší sim. kroku a pozastavení

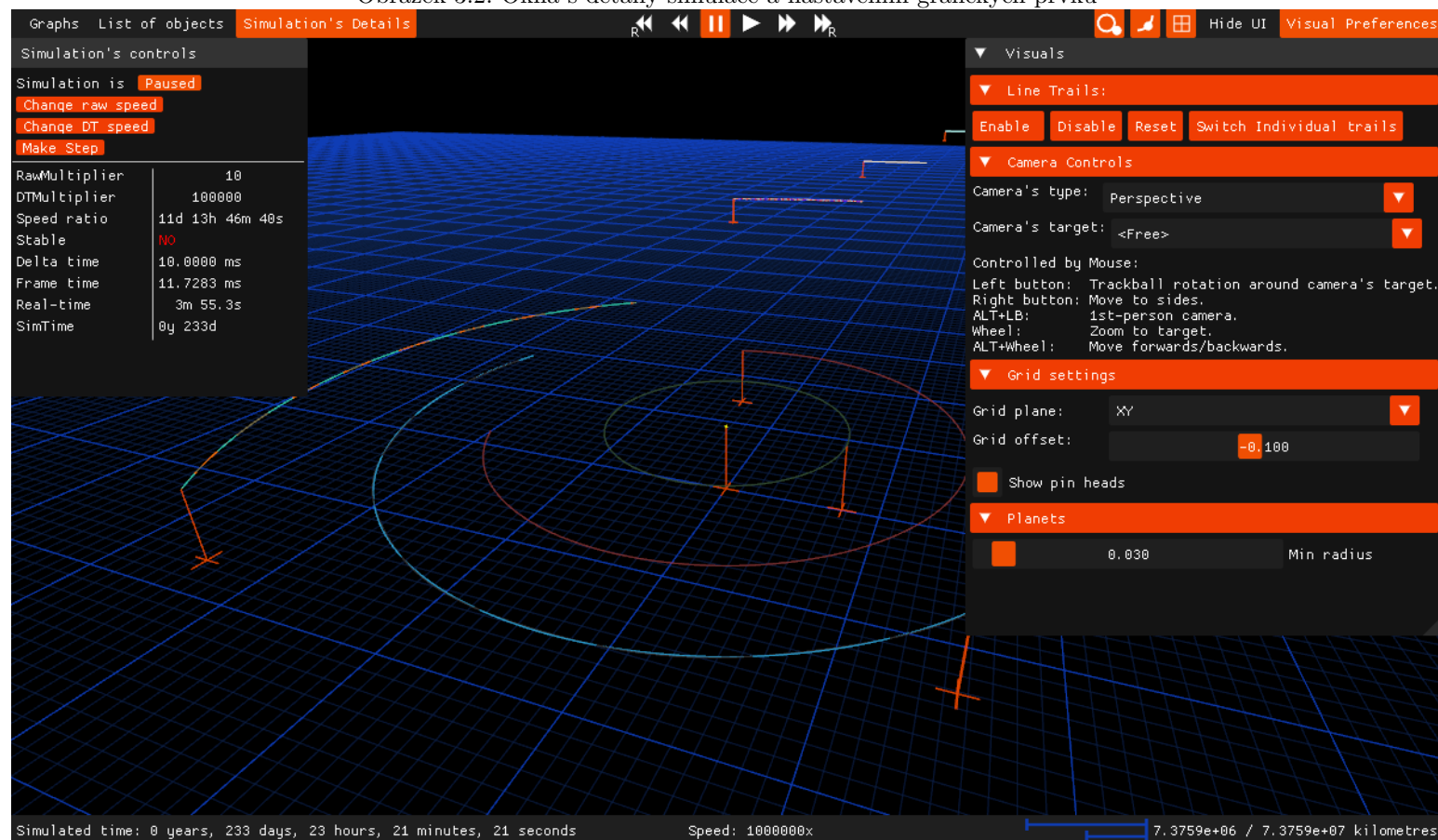
3. Tři přepínače zleva zapínají/vypínají - Skutečné velikosti objektů, LineTrails, vykreslování mřížky. Dále je zde tlačítko **na vypnutí** uživatelského rozhraní, které se **objeví zmáčknutím F1**. Poslední tlačítko otevře okno s nastavením grafiky.

4. Odsimulovaný čas

5. Rychlost simulace vůči reálnému času.

6. Velikost menší a větší mřížky.

Obrázek 3.2: Okna s detaily simulace a nastavením grafických prvků



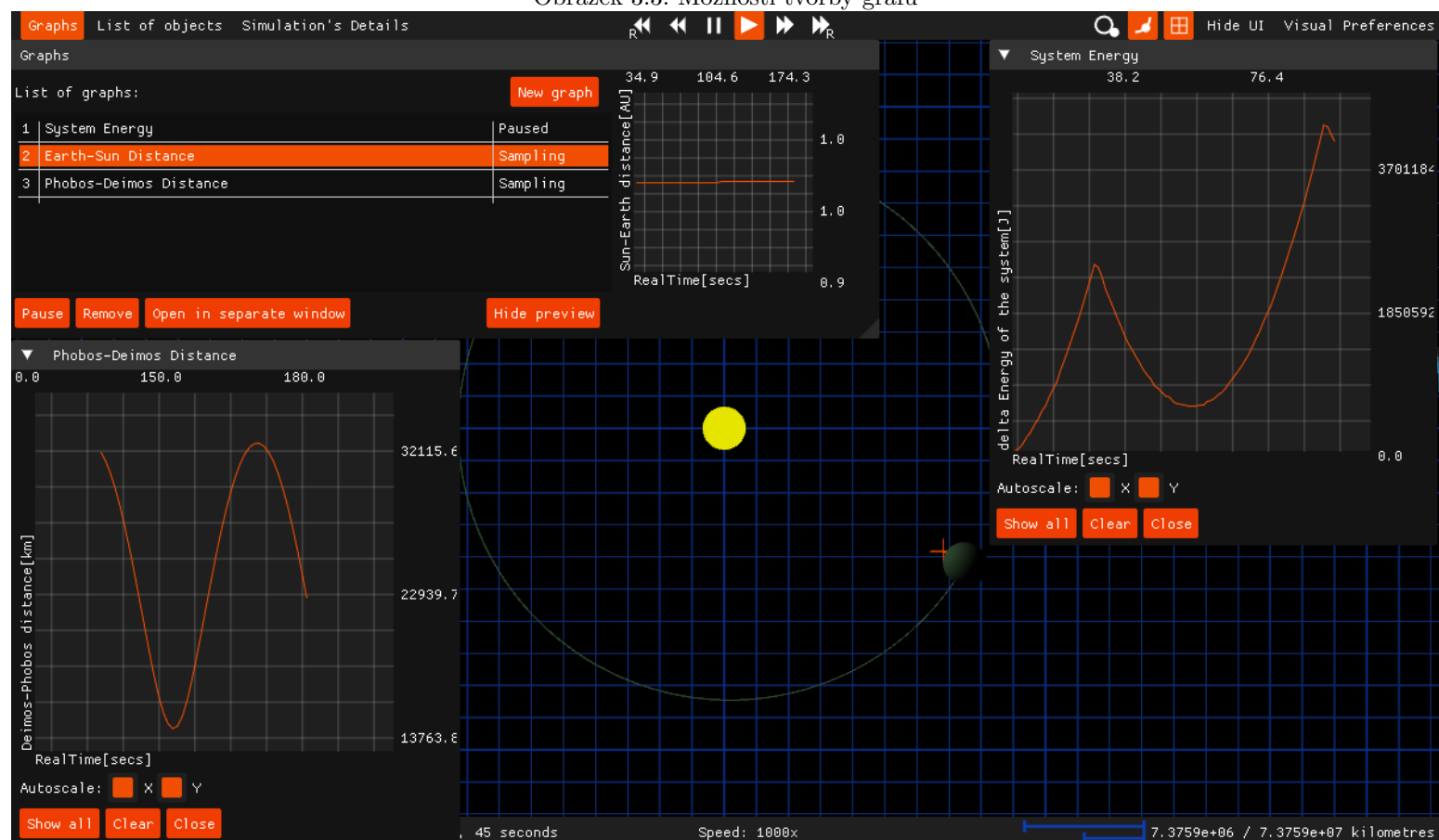
Levé okno nabízí pohromadě detailní informace o simulaci.

Vpravo jsou pak různá nastavení. Je zde uvedeno ovládání kamery pomocí myši nebo nastavení typu kamery, orientaci mřížky a minimální velikosti objektů při vypnutí reálných velikostí.

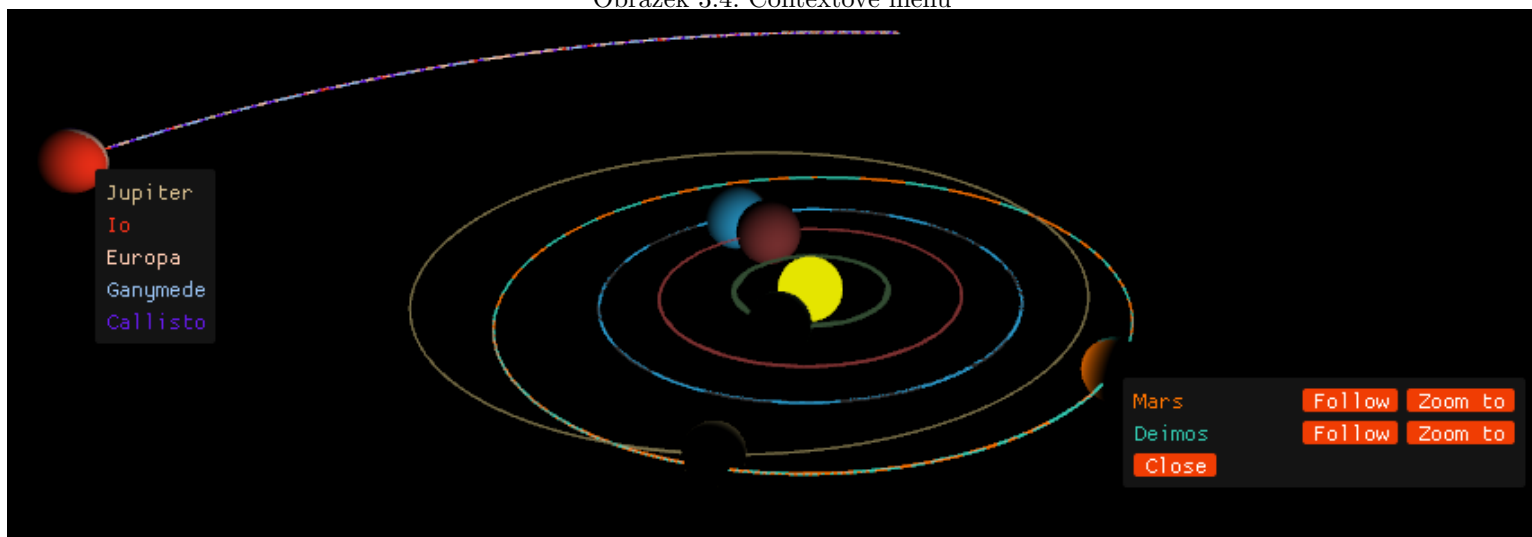
Na tomto obrázku jsou reálné velikosti zapnuty a proto nejsou jednotlivé planety vůbec vidět.



Obrázek 3.3: Možnosti tvorby grafů



Obrázek 3.4: Contextové menu










Na tomto obrázku je vidět, že při najetí myši na objekty se objeví jejich jména(zde Jupiter...). Při pravém kliknutí myši na objekt se otevře okno(Mars,Deimos) které navíc umožňuje zafixovat kameru na daný objekt, popřípadě se k němu přiblížit.

Obrázek 3.5: Seznam simulovaných objektů

Units' properties

Frame of reference: (?) Center ▼ Time: Seconds ▼ Length: Meters ▼ Mass: Kilograms ▼

(?)	Name (?)	Distance ▼	Speed ▼	Mass
	Sun	0.000965 m	2.33e-05 m/s	1.99e+30 kg
	Mercury	5.79e+10 m	4.74e+04 m/s	3.3e+23 kg
	Venus	1.08e+11 m	3.5e+04 m/s	4.87e+24 kg
	Earth	1.5e+11 m	2.98e+04 m/s	5.97e+24 kg
	Mars	2.26e+11 m	2.41e+04 m/s	6.42e+23 kg
	Jupiter	7.79e+11 m	1.31e+04 m/s	1.9e+27 kg
	Saturn	1.43e+12 m	9.64e+03 m/s	5.68e+26 kg

Tento obrázek zobrazuje detail otevřeného okna se seznamem simulovaných objektů.  
Dovoluje měřit vzájemné polohy a rychlosti všech objektů a zobrazovat je ve vybraných jednotkách.

### 3.3 Pokročilé možnosti

Program také nabízí pokročilejší ovládání pomocí příkazové řádky, s nímž je možné načítat simulovaná data ze souborů, vybrat si simulační 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 solar -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 24x 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.

**POZOR:** Momentálně je možné měnit pomocí uživatelského rozhraní parametry simulace za běhu. Ovšem **nedělejte** to, nahrávací algoritmus s tím nepočítá a je to jedno z témat oprav v programátorské sekci.

5. **SolarSystem.exe zaznam.replay**

Přehraje zaznamenanou simulaci z předchozího příkladu v grafickém prostředí.

Ke zdrojovým souborům je ve složce `data/` přiložen vzorový formátovaný text `vstup.txt` a také jeden záznam simulace `zaznam.replay`. Přesný popis formátu vstupního souboru je uvnitř ve formě komentářů a také v sekci 4.1 ze zimního semestru. Pro vysvětlení parametrů simulace je k dispozici popis v sekci ?? a detailněji ve druhé kapitole zimní dokumentace.

# Literatura

- [1] Song Ho Ahn - *OpenGL Projection Matrix*  
[http://www.songho.ca/opengl/gl\\_projectionmatrix.html](http://www.songho.ca/opengl/gl_projectionmatrix.html)
- [2] Glen Murray - *Rotation About an Arbitrary Axis in 3 Dimensions*, 6.června 2013  
<https://sites.google.com/site/glenmurray/Home/rotation-matrices-and-formulas>
- [3] Joey De Vries - *Úvod k maticím a vektorům* <https://learnopengl.com/#!Getting-started/Transformations>
- [4] Joey De Vries - *Coordinate Systems*  
<https://learnopengl.com/#!Getting-started/Coordinate-Systems>
- [5] Joey De Vries - *LookAt matice*  
<https://learnopengl.com/#!Getting-started/Camera>
- [6] Joey De Vries - *Depth buffer*  
<https://learnopengl.com/#!Advanced-OpenGL/Depth-testing>
- [7] Brano Kemen - *Maximizing Depth Buffer Range and Precision*  
<http://outerra.blogspot.cz/2012/11/maximizing-depth-buffer-range-and.html>
- [8] theo - *Depth Precision*  
<http://dev.theomader.com/depth-precision/>
- [9] Podpora OpenGL rozšíření u různých GPU  
[http://opengl.gpuiinfo.org/gl\\_extensions.php](http://opengl.gpuiinfo.org/gl_extensions.php)
- [10] Interaktivní srovnání logaritmického a klasického depth bufferu  
[baicoianu.com/~bai/three.js/examples/webgl\\_camera\\_logarithmicdepthbuffer.html](http://baicoianu.com/~bai/three.js/examples/webgl_camera_logarithmicdepthbuffer.html)
- [11] Implementace reversed depth bufferu pro OpenGL  
<https://nlguillemot.wordpress.com/2016/12/07/reversed-z-in-opengl/>
- [12] Bruce Dawson - *Blog s výbornými články o problematice floating-point čísel*  
<https://randomascii.wordpress.com/2012/02/13/dont-store-that-in-a-float/>
- [13] GLFW - dokumentace ke kompilaci knihovny [http://www.glfw.org/docs/latest/compile\\_guide.html](http://www.glfw.org/docs/latest/compile_guide.html)