TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Implementace pokročilých vizualizačních technik do systému Aurora

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Adam Franců**
Vedoucí práce: Ing. Jiří Jeníček, Ph.D.

Liberec 2015

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Implementation of advanced visualization techniques in Aurora system

Diploma thesis

Study programme: N2612 – Electrotechnology and informatics
Study branch: 1802T007 – Information technology

Author: **Adam Franců**
Supervisor: Ing. Jiří Jeníček, Ph.D.

Liberec 2015

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2014/2015**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Adam Franců**
Osobní číslo:	**M13000180**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Informační technologie**
Název tématu:	**Implementace pokročilých vizualizačních technik do systému Aurora**
Zadávající katedra:	**Ústav informačních technologií a elektroniky**

Zásady pro vypracování:

1. Navrhněte a realizujte změny v celkovém vykreslování rozsáhlých lokací planetárních systémů (využití metody floating point origin). Rovněž se zaměřte na pozice jednotlivých planetárních systémů (ať už reálných, uživatelsky definovaných či náhodných) v rámci jejich širšího okolí a zdůrazněte velikost prostoru, ve kterém se nachází.

2. Stávající verze aplikace umožňuje simulovat pohyb jednotlivých těles v planetárních systémech pomocí analytického řešení 2 těles. Navrhněte a realizujte algoritmus, který obohatí simulační schopnosti aplikace o numerické řešení problému n těles, jehož výpočet bude realizován skrz GPGPU.

3. Ve stávající verzi aplikace se nachází větší databáze textur, která se stará o různorodost povrchů na tělesech (planety, měsíce,..). Navrhněte a realizujte algoritmus, který bude generovat rozdílné povrchy (2D, pouze na texturové úrovni) na vhodných tělesech s cílem zmenšit šanci na nalezení stejně vypadajících objektů na minimum.

Rozsah grafických prací:	**Dle potřeby dokumentace**
Rozsah pracovní zprávy:	**cca 40 až 50 stran**
Forma zpracování diplomové práce:	**tištěná/elektronická**

Seznam odborné literatury:

[1] **Thorne, C.: Using a floating origin to improve fidelity and performance of large, distributed virtual worlds, IEEE Computer Society Press, ISBN 0-7695-2378-1**

[2] **Wilt, N.: The CUDA Handbook: A Comprehensive Guide to GPU Programming, ISBN 0-321-8094-67**

[3] **Nguyen, H.: GPU Gems 3, Addison-Wesley Professional, ISBN 0-3215-1526-9**

Vedoucí diplomové práce:	**Ing. Jiří Jeníček, Ph.D.** Ústav informačních technologií a elektroniky
Datum zadání diplomové práce:	**12. září 2014**
Termín odevzdání diplomové práce:	**15. května 2015**

prof. Ing. Václav Kopecký, CSc.
děkan

Technická univerzita v Liberci - 14 -

prof. Ing. Zdeněk Plíva, Ph.D.
vedoucí ústavu

V Liberci dne 12. září 2014

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum: 14.5.2015

Podpis:

Abstrakt

Autor se v této práci věnuje problematice vylepšení klíčových částí aplikace Aurora. Tato aplikace se zaměřuje na simulaci a vizualizaci pohybu těles v planetárních systémech. Tyto systémy jsou založeny buď na reálných datech či procedurálně vygenerované. Práce je rozdělena na 3 části.

První část se týká vizualizace rozsáhlých prostor v rozdílných měřítkách. Rovněž popisuje postup, který umožňuje vizualizaci statisíců objektů v jedné scéně a náležité operace s nimi.

Druhá část práce se týká simulační stránky, konkrétně implementace problému n těles. Tento problém řeší pomocí numerické simulace. Výpočet realizuje skrz masivní paralelizmus na GPU pomocí OpenCL API. Pro tento výpočet autor realizuje klient-server architekturu, která umožňuje kompletní oddělení simulační a vizualizační stránky aplikace. Rovněž s touto implementací odpadá potřeba konkrétního hardwaru a výpočetního výkonu na klientské straně.

Třetí část práce se týká problematiky tvorby rozdílně vypadajících těles. V této části autor popisuje, jakým způsobem lze docílit náhodně vypadajících povrchů těles pomocí Perlinova šumu a rovněž popisuje nutné modifikace stávající verze aplikace Aurora, které předcházely implementaci této problematiky.

Abstract

Author of this thesis is focused at improving critical parts of Aurora application, which is focused on authentic simulation and visualization of bodies within different planetary systems. These systems can be based on real data or procedurally generated. This thesis is focused on three main topics.

First part is focused on visualization of large areas within different scale modes. It also describes procedures, which allows visulization of hundreds of thousands objects within one scene and following operations with them.

Second part is focused on simulation part, where author describes the implementation of n body problem using numerical simulation. Calculations are made with massive paralelism using GPU and OpenCL API. For this problem, author implements client-server architecture, which allows complete separation of simulation and visualization. It also removes the need of specific hardware and computation power to be present at client side.

Third part of this thesis is focused on creating bodies with different looking surface. In this part, author describes the use of Perlin noise to achieve such task and also describes followup changes to the application in order to support such algorithms.

Poděkování

Mé poděkování patří panu Ing. Jiřímu Jeníčkovi, Ph.D. za odborné vedení, trpělivost a ochotu, kterou mi v průběhu zpracování diplomové práce věnoval.

Obsah

Seznam obrázků

Seznam zkratek

AU	Astronomická jednotka, 1 AU je rovna 149,597,870,700 metrům
pc	parsek, 1 pc je roven 206.264,806 astronomickým jednotkám
kpc	kiloparsek, 1 kpc je roven 10^3 parsekům
HB	hmotný bod
API	Application Programming Interface, rozhraní pro programování
OpenCL	Open Computing Language, průmyslový standard pro paralelní programování heterogenních počítačových systémů
CPU	Central Processing Unit - centrální procesorová jednotka
GPU	Graphics Processing Unit - grafický procesor
GPGPU	General-purpose computing on graphics processing units, využití grafické karty pro obecné výpočty

1 Úvod

Aurora je software, který dokáže simulovat a vizualizovat tělesa a jejich pohyb v planetárních systémech. Planetární systém může například být naše sluneční soustava. Může to ale být i jakýkoliv z tisíců, které aplikace umí načítat z reálných databází exoplanet (planety mimo sluneční soustavu). Aplikace rovněž umí procedurálně generovat planetární systémy. Klientská část aplikace běží pod herním enginem Unity[1] a jádro spolu s dalšími podpůrnými knihovnami je napsané v jazyce C#.

Protože tato aplikace byla již součástí mé bakalářské práce[2] a téma programování grafických aplikací, vizualizace dat a astronomie mě obecně velmi zajímá, nebylo pro mě těžké nacházet motivaci k rozvíjení funkcionality této aplikace.

V průběhu bakalářské práce jsem si uvědomil, že v oblasti, kterou se aplikace zabývá, existuje spousta velmi zajímavých témat. Bylo nicméně jasné, že tato témata definitivně nelze vyřešit v rámci jedné práce. Na konci mé bakalářské práce jsem si tedy vypracoval seznam věcí, které bych rád v budoucnu implementoval. Zde se dostávám do současnosti a k mé diplomové práci. Tato práce se některých problémů, zmíněných v mé bakalářské práci, týká a řeší je.

Plánované změny razantně rozšíří schopnosti aplikace a postaví ji tak opět dále od doposud existujících aplikací. Z rešerše, kterou jsem provedl, vyplývá, že spousta již existujících aplikací se zaměřuje na data jako taková a případně ještě nabízí zjednodušený náhled na danou situaci. Simulační stránka se obvykle opírá o primitivní abstrakci bez přesnějších orbitálních parametrů. Vizualizace pak primárně spočívá v prostém náhledu na daný systém v malém měřítku.

Například oficiální aplikace Eyes[3] od NASA. Tato aplikace je schopna vizualizovat mimo spousty dalších dat i data z databází exoplanet, nicméně vše probíhá na relativně malém měřítku a v simulaci problému 2 těles. Uživatel nemá v náhledu na systém kompletní volnost pohybu a obloha není interaktivní (uživatel nemůže zjistit informace o okolních hvězdách/systémech).

Další z aplikací, které jsem měl možnost vyzkoušet, je Universe Sandbox[4]. Je to aplikace, která má poměrně propracovanou a konfigurovatelnou simulační část, opírající se o problém n těles. V této aplikaci jsem si tak mohl už napřed vyzkoušet, jak funguje implementace problému n těles a jakým problémům budu muset čelit při implementaci v mé aplikaci. Vizualizace a simulace jednotlivých systémů spočívá v načítání scénářů, které oddělují data od sebe a tedy nabízí uživateli pouze různá pískoviště, které mohou využít k experimentům. Aplikace v základu poskytuje jen zlomek dat, které jsou v dnešní době volně k dispozici. Vizuální stránka rovněž postrádá mnoho elementů a díky zaměření pouze na numerické řešení problému n těles aplikace nenabízí větší měřítko vizualizace.

2 Analýza

2.1 Vizualizace rozsáhlých prostor

Původní verze aplikace si s problémem vizualizace prostor, na kterých se rozkládají planetární systémy, musela poradit víceméně pomocí nejrůznějších triků, které sice ve výsledku umožnily ukázat vše důležité, nicméně zároveň omezovaly volný pohyb uživatele. Řešení se rozhodně nedalo považovat dlouhodobě za přijatelné.

Dalším důležitým bodem byla absence jakéhokoliv vztahu jednotlivých planetárních systémů. Aplikace již v té době uměla načítat tisíce nejrůznějších systémů. Spojitost mezi nimi byla ale maximálně na takové úrovni, že pocházely ze stejné databáze či od stejného generátoru. A jelikož data se v jednotlivých databázích striktně netýkají pouze těles jako takových, ale obsahují i data z pohledu pozorovatele (relativní pozice ze Země), je užitečné tato data vhodně využít.

Je velmi důležité, aby aplikace tohoto typu dokázala zachytit co nejlépe velikosti a vzdálenosti, na kterých se rozkládají jednotlivé planetární systémy. Čím větší měřítko bude možné implementovat, o to větší pocit bude uživatel z výsledku mít. Rovněž je velmi důležité uživateli říci, kde se vlastně daný systém nachází a co se nachází v daném momentě okolo něj (ať už se jedná o vzdálenější tělesa či i systémy). Chtěl bych, aby aplikace dala uživateli jasný signál, že to, čeho jsme všichni součástí, je něco tak obrovského a složitého, co se jen těžko dá představit podle toho, co vidíme v každodenním životě.

Analýza prvního bodu zadání se dá prakticky rozložit do dvou bodů:

- **Vizualizace okolí** jednotlivých planetárních systémů aneb umožnit uživateli získávat informace o tom, co je v okolí daného systému.
- **Vizualizace rozsáhlých prostor** aneb jakým způsobem je možné docílit pohyb uživatele po všech koutech systému aniž by mu byly kladeny nějaké překážky.

Vizualizace okolí je podmíněna faktem, že každý systém, respektive jeho hvězda, má v databázi uloženou vzdálenost, která byla naměřena skrz dlouhodobé pozorování v návaznosti na pohyb pozorovatele (v našem případě je to samozřejmě Země, kde pozorovací období obvykle trvá jeden pozemský rok). Rovněž má každá hvězda uložené sférické koordináty - deklinaci a rektascenzi, kterou se určuje poloha dané hvězdy (respektive samotného systému) na obloze z pohledu pozorovatele (Země).

Toto jsou 3 klíčové hodnoty, které nám postačují na to, abychom mohli vyřešit relativní polohu jednotlivých systémů v databázi (ať už se jedná o reálné či fiktivní) od pozorovatele.

Databáze, které aplikace v současné době využívá:

- Databáze PHL (Planetary Habitability Laboratory) z katalogu exoplanet.[5] Tato data shrnují většinu objevů extrasolárních těles (planet) za posledních 20 let. A to jak z pozemních pozorovacích stanic, tak i z teleskopů ve vesmíru (Kepler).

- Databáze HYG ve verzi 3.0.[6] Tato data jsou výsledkem sloučení několika hvězdných katalogů (*Hipparcos*, *Yale Bright Star* a *Gliese*). Všechny hvězdy v databázi jsou buď 50 parseců od Země či mají zdánlivou hvězdnou velikost (magnitudu) pod hranicí někde mezi 7.5 až 9.5.

Obě tyto databáze přinášejí velké množství dat, se kterými je možné pracovat. Zároveň ale toto množství také přináší problémy při jejich vizualizaci, které popisuji v této kapitole.

Vizualizace rozsáhlých prostor je problém, na který jsem při vývoji bakalářské práce narazil jako jeden z prvních. Jakékoliv přímé vykreslování v měřítku, ve kterém by jedna vizualizační souřadnice odpovídala jednomu metru, nemá význam díky množství chyb ve vizualizaci a chování enginu. Nejenom, že zde existují tělesa, která již sama o sobě vyplní podstatnou část přijatelného prostoru ve vizualizačním koordinátovém systému, tyto tělesa navíc ještě mají mezi sebou vzdálenost, která překročí limity ve vizualizaci o několik řádů.

Jak jsem již zmínil výše, původní verze aplikace si s tímto problémem poradila zavedením umělého omezení. Prakticky zamezila pohyb uživateli v rámci systému a omezila se pouze na připravená místa (planeta a její měsíce). A i zde docházelo k problémům, kdy nejvzdálenější měsíce již neměly tolik místa v koordinátech a jejich pohyb probíhal skrz chvění a cukání. Je tedy nutné pro menší měřítka přijít s kompletně odlišnou metodou vizualizace, která nebude převádět přímé simulační koordináty na vizualizační.

2.2 Problém n těles

Původní verze aplikace se zaměřovala (co se týče simulační stránky) na simulaci problému dvou těles. Tento problém byl vyřešen analyticky a spočívá v tom, že vždy řeší vztah pouze dvou těles, přičemž jedno (hmotnější) se nachází v ohnisku eliptické dráhy, kterou opisuje druhé (méně hmotné) těleso. Případy jsou tedy jasné - hvězda - planeta, asteroid, planeta - měsíc, asteroid. Problém dvou těles je nicméně zcela abstraktní a nelze ho tedy v reálné situaci nalézt. V některých situacích bude tento simulační přístup krajně nedostačující (nestabilní systémy), nicméně v některých (jako třeba naše sluneční soustava) se jedná o celkem dostačující aproximaci toho, jak se tělesa skutečně pohybují (bereme-li v potaz, že se bavíme pouze o hmotnějších

tělesech jako jsou planety a o časových úsecích v rámci pár desítek let). Aplikace je ale každopádně schopna simulovat nejenom naši sluneční soustavu, ale i spoustu dalších systémů, ve kterých se problém dvou těles nemusí vůbec přibližovat reálné situaci.

Řešení výše zmíněného problému spočívá v implementaci tzv. problému n těles. Problém n těles je úloha o pohybu vzájemně integrujících hmotných bodů. Pohyb jednotlivých bodů je dán druhým Newtonovým zákonem, který říká, že zrychlení hmotného bodu je přímo úměrné silám působícím na tento bod. V mém případě jde o gravitační sílu danou Newtonových gravitačním zákonem, který zní následovně:

Každá dvě tělesa o hmotnostech m_1 a m_2 na sebe působí gravitační silou přímo úměrnou hmotnostem těles a nepřímo úměrnou čtverci jejich vzdálenosti.[13]

$$\vec{F}_{12} = -G\frac{m_1 m_2}{r^2{}_{12}}\frac{\vec{r}_{12}}{r_{12}} \tag{2.1}$$

Newtonův zákon lze zapsat vzathem (2.1), kde $\vec{r}_{12}$ je vektor směřující od prvního tělesa k druhému, r_{12} je jeho norma a G je nyní blíže nespecifikovaná konstanta. Označíme polohu í-tého HB $\vec{r}_i = (x_i, y_i, z_i)$ a m_i jeho hmotnost. Spojením 2. Newtonova zákona a gravitačního (2.1) získáme diferenciální rovnici (2.2) s poč. podmínkami (2.3) pro funkci trajektorie í-tého HB fce tvaru $\vec{r}_i = \vec{r}_i(t)$.

$$\frac{d^2(m_i\vec{r}_i)}{dt^2} = -G\sum_{j=1,j\neq i}^{n}\frac{m_i m_j}{r^3{}_{ij}}\vec{r}_{ij} \tag{2.2}$$

$$\vec{r}_i(0) = \vec{r}_i^{(0)} \qquad \dot{\vec{r}}_i(0) = \vec{v}_i^{(0)} \tag{2.3}$$

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \tag{2.4}$$

Pro řešení těchto rovnic je nutné použít numerické metody, které jsou založeny na aproximaci časových derivací. Jejich přesnost je závislá na zvoleném časovém kroku, který musím volit dostatečně malý ($\leq 10^{-2}$ s). Vzhledem k časovému měřítku popisovaných dějů (měsíce, roky až desítky let) by bylo velmi nevhodné používat reálné hodnoty parametrů a proto je nutné provést jejich škálování. Navíc ani díky konečné přesnosti výpočtů v pohyblivé řádové čárce podle normy IEEE754[7] není dobře možné počítat s hodnotami, které se na jedné straně pohybují v řádech $\geq 10^{10}$ v případě hmotností, vzdáleností a na straně druhé v již zmíněných časových intervalech nutných pro stabilitu.

Vzdálenosti jsou přeškálovány pomocí astronomických jednotek (AU) a hmotnosti pomocí $M_\oplus$ (hmotnosti Země). Tím jsem získal hodnoty v rozmezí 10^{-10} až 10^{10}, které už lze dobře interpretovat pomocí datového typu double. Dosazením škálovacích konstant do (2.1) jsem určil parametr G, tak aby se vyrušil s vypočtenou škálovací konstantou. Celkově jsem změnil fyzikální měřítko celé úlohy, protože se řešení úlohy přesunulo na vzdálenosti desítky metrů a gravitační konstantou rovnu jedné.

Pro řešení diferenciálních rovnic je nutné dodat počáteční rychlost a polohu. Mou snahou bylo, aby i při zapnutí simulace n-těles systém zůstal stabilní tj. nezhroutil se sám do sebe. Z tohoto pohledu kritickým bodem je právě volba počáteční rychlosti, kterou jsem volil s pomocí vzorce pro výpočet orbitální rychlosti

$$v = \sqrt{\mu(\frac{2}{r} - \frac{1}{a})} \tag{2.5}$$

kde **a** je délka hlavní poloosy, **r** je vzdálenost mezi tělesy a μ je standardní gravitační parametr hmotnějšího tělesa, který se vypočítá

$$\mu = Gm \tag{2.6}$$

kde G je gravitační konstanta a m je hmotnost tělesa.
Díky tomu, že problém řeším v trojrozměrném prostoru, je nutné mít údaje o aktuální rychlosti ve směru x,y,z. K tomu potřebuji vypočítat tečnu na danou elipsu k aktuální pozici tělesa. Simulace řeší problém dvou těles ve dvourozměrném prostoru a všechny výstupní x,y koordináty transformuje pomocí metody (např. *transformPlanetOrbitPosition*) na trojrozměrné koordináty, které berou v potaz orbitální parametry (inklinace,..). To samé tedy aplikuji na vektor tečny na elipsu v aktuální pozici tělesa dle (2.7), kde x a y je aktuální pozice na eliptické dráze, e je excentricita elipsy, a hlavní poloosa a b vedlejší poloosa.

$$x_t = \frac{(x + ea)}{(a^2)} \qquad y_t = \frac{(-y - eb)}{(b^2)} \tag{2.7}$$

Výsledný trojrozměrný vektor tečny normalizuji a vynásobím rychlostí na eliptické dráze. Vznikají parametry vx,vy,vz, které se každý snímek aktualizují a nastavují ve třídě *Aurora.KeplerianOrbit*. Pro počáteční podmínky potřebuji také pozici, která se rovněž nachází přímo ve třídě *Aurora.KeplerianOrbit*, v parametrech x,y,z. V neposlední řadě je nutné vědět i aktuální hmotnost, která se rovněž nachází ve třídě *Aurora.KeplerianOrbit*. Slovo aktuální jsem zmínil proto, že je v simulaci problému n těles nutné, aby jednotlivá tělesa mezi sebou mohla kolidovat. Tato hodnota hmotnosti se tedy, narozdíl od pevně dané hmotnosti v instanci třídy *Aurora.celestialType*, může v průběhu simulace problému n těles měnit. Všechny hodnoty počátečních podmínek jsou neustále přepočítávány v režimu simulace 2 těles. Počáteční rychlosti měsíců jsou výsledkem vektorového součtu daného měsíce a rychlosti planety, kterou orbituje.

Tvar diferenciálních rovnic (2.2) vybízí k využití paralelního zpracování, protože potřebujeme provést v každé iteraci průchod přes všechna tělesa, přičemž není nutné čekat na zpracování předchozích. Každý přímý algoritmus vykazuje složitost $\mathcal{O}(n^2)$, která by teoreticky při použití paralelizace klesnou na $\mathcal{O}(n)$. Vzhledem k použitému schématu řešení diferenciálních rovnic a vzhledem k nutnosti provádět simulaci v reálném čase, je pro mě tato byť jenom teoretická složitost značnou motivací.

V dnešní době velice dostupným hardwarem pro paralelní výpočty jsou grafické karty, které disponují stovkami až tisíci jednoduchých procesorů. Ke komunikaci s nimi využiji multiplatformní OpenCL API[8], implementace od firmy NVIDIA, konkrétně ve verzi 1.1.

Simulace problému n těles by měla sloužit k zobrazení dynamiky pohybu těles a jednotlivé interakce mezi tělesy. Díky tomu, že odráží reálnou situaci, můžeme tuto simulaci použít jako test, zda planetární systémy v dané konfiguraci mohou vůbec existovat - bude pohyb těles stabilní, či dojde ke kolapsu?

2.3 Tvorba rozdílně vypadajících těles

Jak jsem již zmínil, aplikace Aurora dokáže simulovat a vizualizovat značný počet nejrůznějších těles, které můžeme nalézt v planetárních systémech. Aplikace si musí poradit s počty, které jsou řádově v tisícovkách. Ať už se jedná o reálné, doposud objevené či fiktivní (vytvořené procedurálně). Původní verze byla vybavena značně objemnou databází textur, takže byla schopná si na základě náhody v běžných situacích poradit s tím, aby v dané lokaci (planetárním systému), tělesa vypadala dostatečně rozdílně. Nicméně generování náhodných čísel pro výběr textury pro danou lokaci zdaleka nepovažuji za uspokojivé řešení.

Je rovněž vhodné doplnit, že aplikace řešila a řeší pouze dvourozměrný povrch těles. Jedná se tedy o sférické textury, které jsou namapovány na model koule. Aplikace neřešila a nadále neřeší samotnou trojrozměrnou síť pro terén, všechno úsilí se nadále stále vkládá v dostatečně velkém rozlišení jednotlivých textur a přidávání plastičnosti (pohoří, krátery,..) skrz normálovou texturu.

Problém tvorby rozdílně vypadajících těles je samozřejmě něco, co se prakticky nedá nikdy dokonale vyřešit. Vždy zde bude šance, že nalezneme tělesa, která vypadají prakticky stejně či se liší o malé detaily. Nicméně cílem tohoto bodu zadání bylo se pokusit o zmenšení šance na nalezení podobně/stejně vypadajících těles a (pokud možno) vylepšit či alespoň ponechat jejich vizuální stránku z původní verze.

Když jsem začal pátrat po různých přístupech k řešení problémů, které se týkají řešení procedurálního generování terénů (a nejenom terénů), brzy jsem zjistil, že jednou z nejvhodnějších metod je **použití šumu**. Šum je typicky aspekt, který nevyhledáváme a ve spoustě situacích nám akorát přináší technické komplikace. Nicméně pokud chceme například vytvořit něco, co vypadá přirozeně, obvykle se právě obracíme k šumu.

Existuje mnoho kategorií šumů a funkcí, kterými se dají generovat. Pro tento případ jsem zvolil Perlinův[15, 20] šum, který dokáže relativně rychle vygenerovat velmi kvalitní a nastavitelný šum. Ačkoliv je Perlinův šum sám o sobě velmi zajímavý, po studii článku[16] jsem se rozhodl využít rovnou komplexnější funkce, konkrétně fraktálního brownova pohybu - fBm. Tato funkce je vlastně suma šumů s různými parametry, přičemž počet prvků v součtu reprezentuje obvykle tzv. počet

oktáv. FBm lze definovat jako (2.8) (funkce noise může být například Perlinův šum).

$$noise(p) + \frac{1}{2}noise(2p) + \frac{1}{4}noise(4p) + ... \tag{2.8}$$

Implementace Perlinova šumu sama o sobě není složitá, nicméně není třeba implementovat vlastní řešení, když v dnešní době existují volně dostupné implementace. Při hledání vhodných knihoven jsem narazil na fakt, že Perlinův šum je k dispozici skoro všude (včetně integrované funkce přímo v Unity - *Mathf.PerlinNoise*) a existuje nespočetně mnoho implemenetací. Hledání vhodných implementací jsem zakončil na[17], která, mimo standardního perlinova šumu v 1/2/3 rozměrech, obsahuje právě i konfigurovatelnou (změna počtu oktáv) implementaci fBm. Celá implementace byla navíc pomocí jednoduché C# třídy *PerlinNoise*.

Nyní je důležité si přesně stanovit, jaká tělesa pomocí této metody budu řešit, co vše je třeba modifikovat než-li bude možné zobrazit první těleso, které má proceduralně vytvořený povrch. Kapitola 3.3 se zaměří na tyto problémy a popíše, jakým způsobem vlastně výstupní dvourozměrný šum v aplikaci používám.

3 Realizace

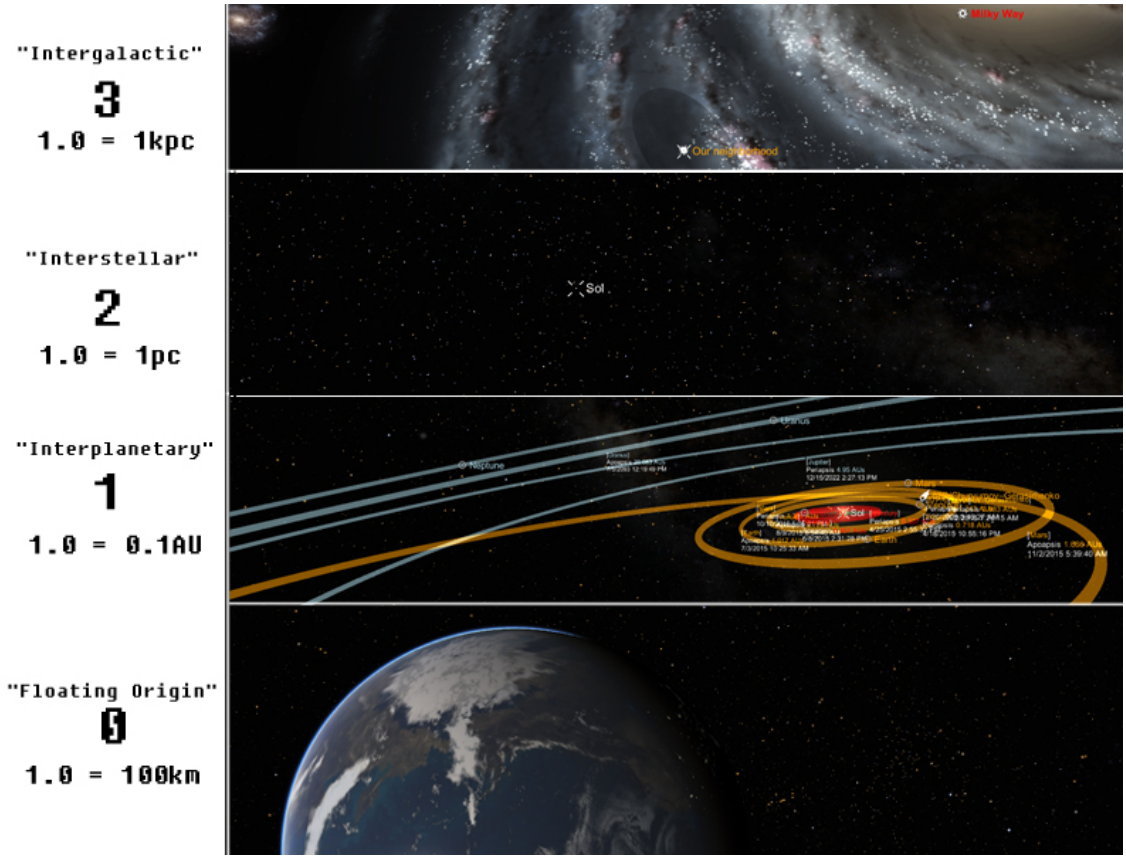
3.1 Vizualizace rozsáhlých prostor

Původní verze aplikace přistupovala ke všem možnostem skrz hlavní menu. To jsem se již s výše zmíněnými změnami rozhodl kompletně odstranit. Důvodem byl fakt, že by aplikace měla být více intuitivní a měla by co nejlépe reprezentovat prostor, který se snaží vizualizovat. Měla by uživateli poskytnout volný pohyb po prostoru s minimem hledání a klikání v nejrůznějších nabídkách. Nicméně vznikla samozřejmě otázka - jakým způsobem to budu řešit, co ještě řešit budu a co už ne.

Aplikace se snaží o vizuální reprezentaci vesmíru, který se sám o sobě nedá pořádně změřit a říci, kde vlastně končí. Máme zde objekty jako galaxie, hvězdokupy, mlhoviny, černé díry, atd. Dalo by se samozřejmě pokračovat. Definitivně je tedy důležité si stanovit takový limit, který pokud možno realizaci omezí vzdálenostně tak, že - zatím - nebude řešit extrémní mezigalaktické vzdálenosti, nebude řešit nejrůznější vesmírné objekty se omezí na to, co naše databáze poskytují - planetární systémy. Okolí jsem tedy v aktuální verzi realizoval jako tzv. hvězdokupu (star cluster). Hvězdokupa (může být kulová, otevřená), se skládá většinou ze sférického uspořádání hvězd, jejichž hustota klesá se vzdáleností od středu hvězdokupy. Toto dobře sedí do situace, ve které se aplikace nachází - mám totiž rovněž k dispozici data, které se rozkládají do sférické oblasti a hustota klesá čím dále se uživatel nachází od středu (pochopitelné - naše pozorovací prostředky jsou schopné zachytit více dat blíže k nám).

Nabízí se ale další otázka - pravděpodobně budu chtít, aby aplikace mohla vizualizovat více těchto hvězdokup. Konkrétně - mám zde reálná data o cca 125 tisících systémech, pozorovaných z našeho systému (sluneční soustava), nicméně mám tady rovněž možnost procedurálně generovat systémy či definovat své vlastní. Jestliže bylo odstraněno menu, je třeba nalézt řešení, jakým způsobem poskytnout možnost si vybrat cílovou hvězdokupu na detailní průzkum. Rozhodl jsem se, že bude třeba reprezentovat (v přiměřené míře) i úroveň nad hvězdokupami, čili galaxie. Pomalu a jistě se při vývoji začaly rýsovat první náznaky toho, že aplikace bude řešit více měřítek najednou. Tato volba se ukázala později jako klíčová a nezbytná, protože bez ní by byla implementace mnohokrát složitější. Obrázek 3.1 poskytuje přehled o všech měřítkách, které byly pro aplikaci realizovány (a to včetně názvu, reálné vzdálenosti na jednu souřadnici ve vizualizačním prostoru).

Přepínání těchto módů řeší třída jádra klientské aplikace *AuroraVECore* skrz



Obrázek 3.1: Měřítka, která jsou dostupná v aplikaci Aurora

hlavní metodu zvanou *DynScaleChange*, která dostává hlášení obvykle od třídy kamery *ControllerCam* či gui třídy *ControllerGUI*. Jedná se o volání jako například: uživatel je blízko objektu x či uživatel je již daleko od objektu y a je třeba změnit měřítko a přepočítat pozici uživatele pro nové měřítko. Měřítka 0 a 1 může uživatel rovněž přepínat skrz horní nabídku v uživatelském rozhraní (tlačítko 100km/1AU). Protože současná verze aplikace již neobsahuje více scén, je změna měřítka podmíněna řádným vyčištěním scény, ve které se nacházejí objekty z posledního měřítka. O to se stará privátní metoda *clearScene* ve třídě *AuroraVECore*. Každý objekt, který existuje ve scéně v jakémkoliv měřítku, má tzv. tag, čili je označen a zařazen do určité skupiny. Tento systém dělá čištění scény a mnoho dalších operací mnohem přehlednější a jednodušší, nemluvě o tom, že pro vestavěné Unity metody je vyhledávání pomocí tagů mnohokrát rychlejší než-li skrz globální vyhledávání objektů ve scéně.

V následujícím textu popíši jednotlivá měřítka a poskytnu podrobnější informace o problémech, na které jsem při implementaci těchto měřítek narazil.

Mezigalaktické měřítko je určené pro vizualizaci objektů na úrovni hvězdokup a galaxií. Jak jsem již zmínil výše, okolí jsem omezil pouze na hvězdokupy, nicméně neomezuji se pouze na jednu a díky odstranění menu a změně stylu, kterým uži-

vatel aplikaci ovládá, bylo nutné toto měřítko implementovat. Naznačuje uživateli přibližné pozice jednotlivých hvězdokup, které následně může vybrat a po přiblížení i navštívit. Dalo by se říci, že mezigalaktické měřítko momentálně slouží jako nové, interaktivní menu, do kterého se aplikace přepne ihned po startu.

Toto měřítko je nejmenší možné, které aplikace zvládne. Jedna souřadnice ve vizualizačním prostoru odpovídá 1 kpc a je možné se při pohybu k vybraným hvězdokupám přepnout do mezihvězdného měřítka.

Mezihvězdné měřítko má za úkol vizualizovat hvězdokupy. Toto měřítko je druhé nejmenší možné, které aplikace zvládne. Jedna souřadnice ve vizualizačním prostoru odpovídá 1 pc a je možné se při pohybu k vybraným hvězdám přepnout do meziplanetárního měřítka. Uživatel se ale může přesunout i výše - do mezigalatického měřítka a vrátit se k výběru hvězdokup. Oba přesuny probíhají intuitivně pouze pomocí přesunu kamery v prostoru.

Již z názvu hvězdokupa vyplývá, že toto měřítko musí být schopné vizuálně reprezentovat potenciálně statisíce objektů aniž by docházelo k vážným problémům s pomalejším vykreslováním a tedy ke zhoršení či úplné zamezení ovládání aplikace. Každá hvězda (reprezentující planetární systém) pro toto měřítko je založena na herním objektu *StarDistant* se stejnojmennou třídou. Tato třída se stará o to, aby vizuálně reprezentovala danou hvězdu (systém) a byla schopna detekovat, zda má uživatel zájem se o této hvězdě (systému) dozvědět více. To může znamenat dvě věci - buď se uživatel rozhodne na danou hvězdu kliknout, aby zjistil základní informace o hvězdě a co se okolo ní nachází, nebo bude chtít rovnout zjistit, jak daný systém vypadá, čili daný systém přímo navštívit (tato akce na určité vzdálenosti spustí změnu měřítka na meziplanetární).

Nicméně jak jsem již zmínil výše, instancování statisíců objektů třídy *StarDistant* v reálném čase nepřipadá díky vysokým nárokům v úvahu. Bylo tedy nutné přijít s dostatečně robustním řešením, které umožní plynulý běh a pokud možno co nejlépe bude zobrazovat, jak daná hvězdokupa vypadá a kolik hvězd se v ní nachází. Ukázalo se, že nejlepším řešením bude rozdělení daného prostoru hvězdokupy na prostor krychlí, který bude měnit hustotu dle vzdálenosti od středu hvězdokupy (to je vlastně definice hvězdokupy). Vznikl tedy nový objekt, nazvaný *StarCube*, který se po vytvoření stará o rozdělení hvězdokupy na krychle, které se stanou jeho potomky. Jednotlivé krychle disponují seznamem, kde jsou uložené všechny hvězdy (planetární systémy), které se v dané oblasti nacházejí. Každá krychle má rovněž objekt typu Collider, který umožňuje detekovat pohyb a tudíž je schopen detekovat vstup uživatele do dané oblasti (resp. vstup kamery).

Tento systém nicméně ještě nic moc sám o sobě neřeší - podařilo se mi sice rozdělit oblast s velkou hustotou dat na menší bloky, stále ale jednoduše nemohu přistupovat k lokálním seznamům v daných oblastech a říci, že chci instancovat objekty typu *StarDistant* pro hvězdy, na které lokální seznamy odkazují.

Důvod je jasný - koncentrace může být i tak stále velká, že by mohlo dojít k přeplnění i těch nejjemnějších oblastí (nějaké minimum existovat musí, nelze dělit prostor na minimální jednotky - mohlo by se stát, že budeme mít pak více krychlových oblastí než-li samotných hvězd). Navrhnul jsem tedy adaptivní algoritmus,

který instancuje třídy objektů *StarDistant* v rámci jedné oblasti, na základě aktuální pozice uživatele a vzdálenosti daných pozic v dané oblasti. Algoritmus jsem nazval adaptivní, protože pokud může, pokusí se nainstancovat nejvyšší možný (únosný, obvykle se jedná o cca 100) počet objektů, který v dané chvíli může nainstancovat. Je rovněž velmi vhodné zmínit, že tato operace se provádí pouze v momentě, kdy řídící algoritmus kamery detekuje dostatečně velký pohyb uživatele. To zabrání neustálému mazání a tvorbě stejných objektů, která by probíhala třeba i 60x za vteřinu. Zmínil jsem mazání - při přesunu tento algoritmus nejenom tvoří, ale i maže hvězdy (systémy), které nevyhovují aktuálním podmínkám. Každá oblast má své pole, kde jsou uloženy *Vector3* pozice jednotlivých systémů, názvy a i pole datového typu boolean, které značí zda danou hvězdu (systém) v dané chvíli instancovat či ne.

V kódu se o tuto funkčnost stará více částí najednou - jak jsem již zmínil, třída *StarCube* reprezentuje a uchovává informace o jednotlivých krychlových oblastech. Dále je to metoda *spawn_InterstellarVicinity*, která se nachází ve třídě *AuroraVECore*. Ta se stará o samotné mazání a tvorbu objektů na základě aktuální situace. Nelze přehlédnout ani místo, ze kterého se tato metoda volá (resp. se nastavuje spínač, který spustí metodu v jádře a ta po dokončení tento spínač přepne na false) - třída *ControllerCam*, metoda *NewCam_VicinityCheck*, která se obecně stará o správu akcí, které probíhají okolo uživatele (kamery) - upravuje citlivost pohybu v prostoru na základě aktuálně vybraného objektu či vzdálenosti objektů okolo kamery, stará se o kontrolu podmínek pro změnu měřítka, atd.

Ačkoliv jsem vyvinul dostatečně robustní systém na vizualizaci objektů blízkých uživateli, zobrazovaly se pouze ty blízké a ty, které by za normálních podmínek měl uživatel vidět, byly schované. Bylo tedy nutné přijít se systémem, který si poradí se statisíci objekty tak, že je jednoduše vykreslí všechny. Na počátku jsem otestoval hned několik metod, které všechny měly společné jednu věc - možnost vykreslovat rychle statisíce jednoduchých objektů. Byl otestován například částicový systém, který je dostupný v Unity, nicméně pro toto použití se jevil nakonec jako velmi nepraktický. Metoda, která se nakonec ukázala jako nejlepší (co se týče použití i vizuální stránky), spočívá ve tvorbě a vykreslování dynamického modelu, který má jednoduše zadefinované pouze vertexy a neřeším jeho vnitřní strukturu (trojúhelníkové plochy). Vznikl nový herní objekt s třídou *StarCloud*. Tato třída se stará o přiřazená data a v průběhu běhu aplikace umožňuje i tento dynamicky vytvořený model modifikovat skrz pracovní vlákna.

Instancování těchto objektů se provádí opět ve třídě jádra *AuroraVECore*, konkrétně pak v privátní metodě *prepareInterstellarMode*. Existují zde ale limitace - lze vytvořit model, který má maximálně 65535 vertexů (body, vrcholy). Jestliže tedy existuje více než 65 tisíc hvězd, herní objekt třídy StarCloud se nainstancuje ve větším počtu. Důležitý je fakt, že každý objekt třídy *StarCloud* se vytvoří na souřadnicích 0,0,0 a že koordináty jednotlivých vertexů v rámci dynamicky tvořených modelů odpovídají pozicím v rámci hvězdokupy. Z toho vyplývá, že všechny dynamicky tvořené objekty třídy *StarDistant* (popsané výše) budou přesně na pozicích daných vertexů v modelech, o které se starají třídy *StarCloud*. Nastala ale otázka, jakým způsobem by tento dynamicky vytvořený model, resp. jeho vertexy,

měly reprezentovat danou hvězdokupu. Mám k dispozici dostatečně silný prostředek - změnu rgba barvy jednotlivých vertexů. Barva je jedním z klíčových prvků, kterou můžeme jednotlivé hvězdy rozpoznávat. Prvně jsem tedy implementoval do třídy *StarCloud* jednoduché pole typu byte, které definovalo, jakou barvu hvězdy by měl určitý vertex reprezentovat (0-7, každé číslo odpovídá různé spektrální třídě hvězdy). Využil jsem rovněž složku průhlednosti, která jednotlivé barvy oslabila tak, aby nebyly dohromady příliš silné (tohoto faktu využívá i postup, popsaný níže). Modely (meshe) se po této implementaci chovaly různobarevně a již se nejednalo pouze o bílý chumel vertexů. Nicméně každá hvězda má určitý zářivý výkon a její světlo se vzdáleností slábne. Bylo mi jasné, že pro zdárný výsledek je třeba tento proces reprezentovat. V astronomii se používá pojem hvězdná velikost (magnituda), která udává jasnost hvězdy (či i jiných těles) v závislosti na pozici pozorovatele. Čím je hodnota menší, tím jasnější dané těleso z pozice pozorovatele je. Výpočet hvězdné velikosti (konkrétně zdánlivé) má podobu (3.1), kde l je zářivý výkon dané hvězdy a d vzdálenost pozorovatele v pc.

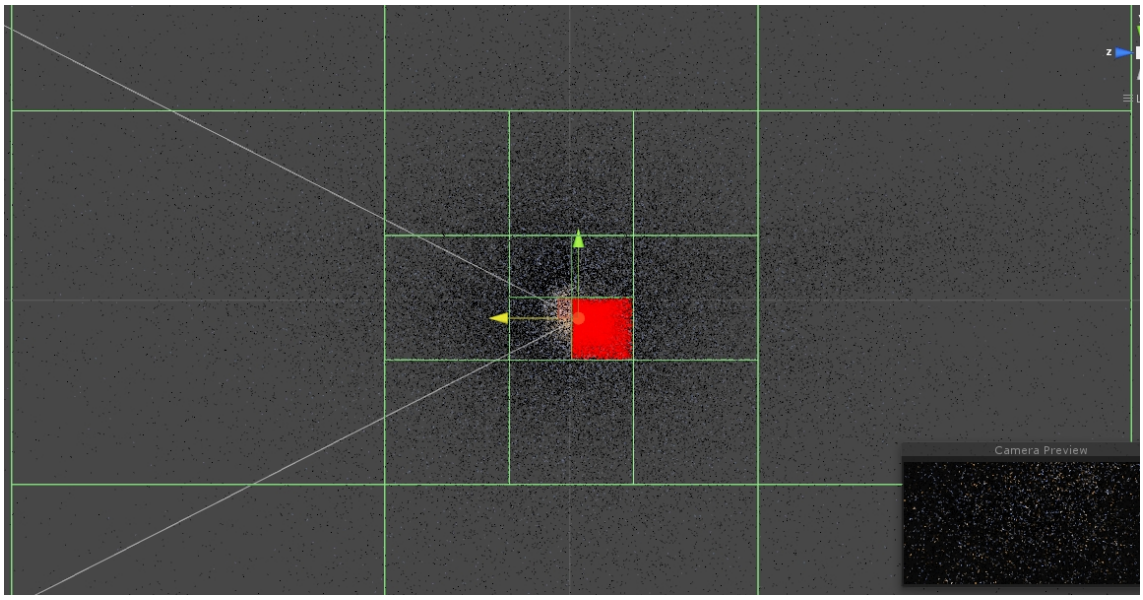
$$-2.72 - 2.5 \log_{10}(l/d^2) \tag{3.1}$$

Metoda *getApparentMagnitude*, která tento výpočet provádí, se nachází ve třídě *Aurora.Star*. Rovněž jsem implementoval tzv. magnitudový limit, který simuluje délku expozice kamery (čím delší expozice, tím bohatější obloha je). Tento limit může uživatel ovlivňovat. Obecně s tímto přístupem platí, že hvězdy, které mají vysoký zářivý výkon, jsou vidět na větší vzdálenost než hvězdy, které mají naopak malý zářivý výkon. Podle hvězdné velikosti můžeme řídit, jakým způsobem zobrazujeme jednotlivé hvězdy - konkrétně tedy které jsou vidět a které už ne. A jestliže mají být vidět, tak jak moc velká alfa složka barvy by měla být (jak jsem již zmínil výše, alfa složka v barvě jednotlivých vertexů ovlivňuje viditelnost). Když tento algoritmus byl připraven, byl nastaven tak, aby fungoval ze středu hvězdokupy (pozorovatel umístěn ve středu).

Nicméně aplikace umožňuje uživateli volný pohyb a je tedy nutné, aby se hvězdné velikosti a tedy i viditelnost jednotlivých hvězd (vertexů v dynamicky vytvořených modelech) modifikovala na základě jeho pohybu a nastaveného magnitudového limitu. Opět nastal ale problém - přepočítávat hvězdné velikosti a upravovat jednotlivé vertexy dynamických modelů každý snímek je krajně neefektivní a navíc přináší sebou vysoké nároky na čas. Obnovovací algoritmus byl ten nejdříve zpracován tak, že obnova se zapne, pokud uživatel změní pozici (popsáno výše). Nicméně i to mělo razantní dopad na plynulost aplikace. Obvykle to spočívalo ve 2 vteřinovém zaseknutí, což je nepřijatelné. Byl jsem tedy nucen tuto práci přesunout do jiných vláken. Vznikla další třída, nazvaná *StarCloudRefresh*, která se zaměřuje na výpočet hvězdných velikostí a aktualizaci pole typu *Color*, které posléze aplikujeme na dynamicky vytvořený model pomocí atributu *mesh.colors*. Tato třída je potomkem třídy *ThreadedJob*, která víceméně reprezentuje funkční vlákno, které je možné spustit pomocí metody *Start*. O volání se stará třída *StarCloud*. Jestliže existuje více herních objektů třídy *StarCloud*, existuje i více vláken, které se starají o přepočítávání hvězdných velikostí. Při tomuto přesunu zátěže do jiných vláken je důležité psát kód tak, aby nevyužíval nic z aktivní Unity API, protože většina Unity metod pracuje pouze v

hlavním vykreslovacím vlákně. Pokud je tedy možné vyhovět těmto podmínkám, určitě se vyplatí přesouvat náročnější výpočty, u kterých nepotřebujeme mít výsledky každý snímek, do jiných vláken a tedy ulevit hlavním vláknu, které se stará o vykreslování.

Obrázek 3.2 ukazuje výše zmíněné postupy, které jsem implementoval pro mezihvězdné měřítko. Žlutá a zelená šipka značí osy kamery (čili současná pozice uživatele). Zelené čáry naznačují rozdělení prostoru hvězdokupy na objekty typu *StarCube*. Červené (tzv. *Debug.DrawLine*) čáry značí aktuální čtenou databázi z dané *StarCube* třídy. Rovněž je na tomto obrázku možné vidět (viditelné hlavně okolo prostřední části) dynamicky vytvořené modely, které jsou součástí objektu třídy *StarCloud*. Obrázek rovněž zachycuje i úpravu viditelnosti okolních hvězd v závislosti na pozici uživatele a nastaveného limitu hvězdné velikosti.



Obrázek 3.2: Ukázka funkčnosti navržených algoritmů v mezihvězdném měřítku (ortogonální projekce)

Meziplanetární měřítko má za úkol vizualizovat zvolený planetární systém. Toto meřítko již prakticky bylo přítomno už od prvotní verze aplikace, nicméně s aktuální verzí prošlo mnoha změnami, které usnadňují pohled na daný systém. Jedna souřadnice ve vizualizačním světě odpovídá 0,1 AU v reálném s jistými vyjímkami (popsáno níže). Do tohoto meřítka je možné se přepnout z mezihvězdného. Po přepnutí se vypočítá tzv. vektor úniku, který je určen pro případ, že se uživatel rozhodne opustit daný systém (a přepnout se zpět do mezihvězdného, připraven navštívit další systém či opustit danou hvězdokupu). Přepnutí se provádí opět intuitivně - stačí víceméně odletět co nejdále od hvězdy v daném systému a jakmile přesáhnete hodnotu únikového vektoru, sepne se změna meřítka na mezihvězdné. Druhou možností je se přepnout do největšího možného meřítka v aplikaci - měřítko, které je nazvané

floating origin (více o přepnutí a tomuto meřítku níže).

První změna, která byla do tohoto měřítka implementována, se týká kamery a tzv. *vicinity* systému. Kamera prakticky ve všech režimech pracuje tak, že upravuje citlivost pohybu na základě okolí. Jestliže se okolo uživatele (kamery) se nic nenachází, citlivost ovládání je daleko vyšší a je tedy možné se přesunout rychle tam, kam uživatel potřebuje. Citlivost ovládání se začne snižovat čím blíže se kamera nachází od určitých objektů. Citlivost má samozřejmě své limity, aby nedošlo k situacím, kdy se s kamera nedá vůbec pohnout či se bude pohybovat tak rychle, že ovládání bude postrádat kompletně smysl. Celý tento postup je řízen z třídy *ControllerCam*, konkrétně z metody *NewCam_VicinityCheck*. Tato metoda skenuje okolní objekty (které mají tag *CBodyVicinity*) a dle nejbližšího objektu modifikuje citlivost ovládání kamery. Tento algoritmus platí hlavně pro meziplanetární a floating origin meřítka, pro mezihvězdné měřítko se skenování okolních objektů díky náročnosti neprovádí. Vicinity systém se nicméně netýká jen toho, jakým způsobem ovlivňuje rychlost kamery. V meziplanetárním a i ve floating origin meřítku jsou standardně malé tělesa (měsíce) schované v celkovém pohledu (jinak by došlo ke značnému chaosu v uživatelském rozhraní). Vicinity systém umožňuje detekovat, kdy nastala správná chvíle pro začátek vykreslování lokálního systému měsíců okolo dané planety. Tuto změnu zaregistruje i grafické uživatelské rozhraní a spustí vykreslování nejrůznějších detailů jako například orbitální dráhy měsíců.

V souvislosti s vicinity systémem je důležité neopomenout fakt, že meziplanetární měřítko již **nemá** všechny vzdálenosti a velikosti přesně fixované na cílové měřítko. Důvody pro to existují hned dva:

- S implementací měřítkového módu floating origin již meziplanetární měřítko není tím největším měřítkem, které aplikace poskytuje. Meziplanetární měřítko tedy může zastupovat funkci mapy. Na mapě je důležité, aby uživatel viděl vše. Výše zmíněný vicinity systém se stará o škálování orbitálních vzdáleností jednotlivých měsíců tak, aby je uživatel mohl bez problémů rozeznat a pozorovat jejich pohyb. Pro toto měřítko jsem rovněž upravil globální proměnnou, která se stará o úpravu velikostí všech těles. A to opět pro snadnější rozpoznání jednotlivých těles. Obě hodnoty (upravující vzdálenosti měsíců od planet a velikosti objektů) jsem zvolil tak, aby pokud možno co nejméně narušovaly výsledný vzhled a působily stále věrohodně.

- Jak jsem již zmínil výše v analýze, problém n těles jsem se rozhodl implementovat primárně s cílem zobrazení dynamiky pohybu těles v systému. Dává tedy smysl tuto simulaci provádět právě v tomto měřítku. Rovněž je pro výsledek simulace n těles žádoucí, aby simulovaná tělesa měla dostatečně velké orbitální vzdálenosti.

Další, podstatnou změnou oproti původní aplikaci, je vykreslování oblohy. Tato změna se týká tématu prvního bodu zadání. Zmínil jsem, že by bylo vhodné, aby jednotlivé systémy měly určitý vzájemný vztah. Toho jsem docílil systémem, který jsem nazval *InterstellarNeighborhood*. Tento systém bere data z hvězdokupy, do které daný systém patří. Tato data následovně využívá k tvorbě odpovídající oblohy.

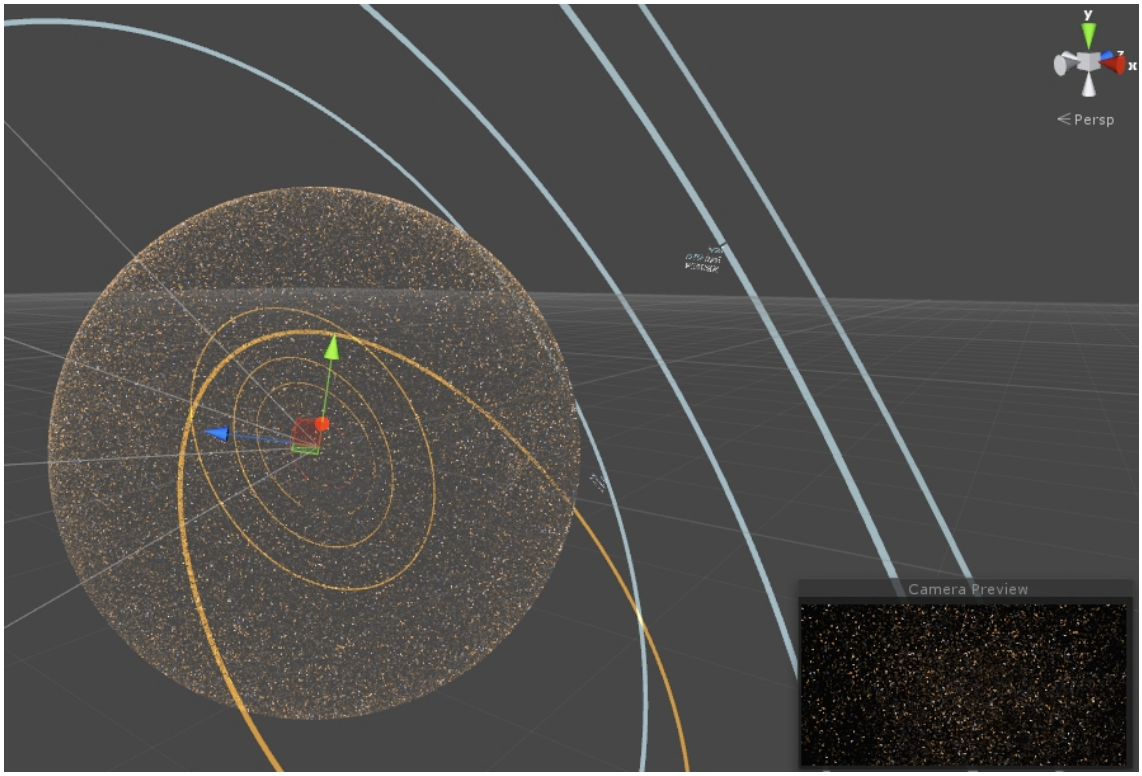
Obloha tedy bude pokaždé jiná a bude odpovídat pozici systému v dané hvězdokupě, měla by rovněž být interaktivní, aby uživatel měl možnost získat informace o okolí daného systému.

Základní algoritmus spočívá v metodě *spawn_InterstellarNeighborhood*, která se volá na konci tvorby meziplanetárního měřítka v metodě *prepareInterplanetaryMode*. Metoda *spawn_InterstellarNeighborhood* se stará o dvě hlavní věci - vytvoření objektů třídy *StarCloud* a objektů třídy *StarDistant*. Jak jsem již zmínil výše (mezihvězdné měřítko), *StarCloud* třída se stará o tvorbu dynamického modelu, který obvykle reprezentuje až na 65535 hvězd, které jsou zpracovány jako vertexy modelu. Tuto funkčnost využívá i meziplanetární měřítko a meřítko floating origin, nicméně zde se dynamický model tvoří tak, aby dával smysl z pohledu z daného systému v dané hvězdokupě. Opět jsou součástí vlákna, která se starají o výpočet hvězdné velikosti a grafické uživatelské rozhraní v obou meřítkách (jak meziplanetární, tak i floating origin) poskytuje volbu limitovat magnitudu (hvězdnou velikost) na zvolenou hodnotu - jinak řečeno, ovlivňovat množství zobrazených hvězd na obloze. *StarDistant* je naopak interaktivní objekt, na který uživatel může kliknout a zjistit, o jaký systém se jedná, resp. jaký tento objekt reprezentuje. Jak tomu bylo již v předchozím měřítku, i zde je nutné omezit počet vytvořených interaktivních objektů typu *StarDistant* na hodnotu, kterou Unity bez problémů zvládne. Jinak řečeno, uživatel bude moci zjistit informace jen o pár stech nejbližších hvězdách a ne o hvězdách, které ve skutečnosti jsou stovky pc daleko.

Otázkou zůstává, jak toto realizovat v prostoru, kde se uživatel (kamera) pohybuje volně. *StarCloud* i *StarDistant* objekty musí neustále být v pozadí. Dalo by se říci, že tyto objekty by měly fungovat obdobně jako skybox (krychle s texturou, která tvoří pozadí celé scény).

K řešení tohoto problému jsem využil další z funkcí Unity enginu, tentokrát se jedná o vrstvy (layers). Vytvořeným *StarCloud* a *StarDistant* objektům byla přiřazena jiná vrstva než mají všechny ostatní objekty. Vertexy v modelech, vytvořených pomocí objektů *StarCloud*, byly vytvořeny pro jednoduchost na konstatní vzdálenosti od 0,0,0 a interaktivní *StarDistant* objekty byly s menší náhodností okolo stejné vzdálenosti od 0,0,0, jako vertexy ve *StarCloud* objektech (modelech). A ve finále je vytvořena nová kamera, řízená třídou *ControllerCamInterstellar*, která kopíruje rotaci hlavní kamery uživatele (míří tedy tam, kam míří hlavní kamera) a vykresluje pouze vrstvu, do které byly vloženy *StarCloud* a *StarDistant* objekty (čili nenastává situace, že by 2 kamery vykreslovaly stejné věci). Pokud na tuto situaci nahlédnu z hlavní kamery (kamery uživatele), tato kamera nevykresluje - čili nevidí, objekty z jiné než standardní vrstvy. Takže vlastně ve výsledku nevadí, že okolo středu souřadného systému mám na konstatní vzdálenosti vytvořenou kouli z objektů, které vizualizují okolí. Obrázek 3.3 ukazuje výsledek vykreslování oblohy v reálné situaci včetně náhledu druhé kamery.

Je nutné podotknout, že toto vyřešilo primárně vizuální část. Nicméně jak jsem již zmínil výše, okolí (obloha) musí plnit i interaktivní část, ve které si uživatel může zjistit informace o nejbližších okolních hvězdách (systémech). Zde nastal menší problém. Detekce kliku uživatele na daný *StarDistant* (resp. jeho Collider) objekt funguje i z více vrstev. Jinak řečeno - není limitována pouze z pohledu druhé ka-



Obrázek 3.3: Ukázka výsledku algoritmu vykreslování oblohy (okolí daného systému)

mery, která se stará o vykreslování těchto interaktivních objektů. Uživatel by tedy mohl nechtěně kliknout a označit objekt na místě, na kterém vlastně není vykreslován (označil by fyzickou polohu objektu na konstatní vzdálenosti od 0,0,0 - čili tak, jak to vidí hlavní kamera). Z tohoto důvodu jsem implementoval nový mód (nejedná se o měřítkový mód), který jsem nazval *SkyWatcher*. Do tohoto módu se lze přepnout pomocí grafického uživatelského rozhraní. Tento mód přesune hlavní kameru na pozici 0,0,0 a umožní uživateli klikat na dané objekty typu *StarDistant*, které mají jinak mimo tento mód třídu Collider vyplou (čili mimo tento mód nejsou interaktivní a jsou pouze jako vizuální doplněk). Nevzniknou tedy problémy s označováním objektů, které na první pohled nejsou tam, kde by měly být. SkyWatcher mód tedy umožní uživateli vskutku sledovat oblohu a zkoumat, co se okolo daného systému nachází.

Floating Origin je měřítkový mód, který má za úkol vizualizovat zvolený planetární systém v co největším možném měřítku. Všechna předchozí měřítka byla natolik malá a limitovala se na relatině malé oblasti, že nebylo třeba uvažovat nad problémem, že mi souřadný systém vizualizace stačit nebude. Nicméně jak jsem již výše zmínil, aktuální verze by měla poskytnout nový pohled a umožnit zobrazení planetárních systémů v tom největším možném měřítku. Aby bylo možné vizualizovat takto gigantické prostory, na kterých se planetárních systémy rozkládají, musel jsem přejít ke vskutku razatním změnám vizualizace. Floating origin je měřítkový

mód, kde jedna souřadnice ve vizualizačním světě odpovídá 100km v reálném. Přepnutí do tohoto módu je možné pouze skrz grafické uživatelské rozhraní (tlačítko s červeným textem 100km) a to v měřítku meziplanetárním.

Vizualizace standardně běží v datovém typu float. To znamená, že všechny pozice, rotace a případně i škálování (obecně transformační matice) běží v datovém typu float. Jestliže bych chtěl, aby například jedna souřadnice ve vizualizačním světě odpovídala 1 kilometru v reálném, znamenalo by to, že 1 AU by se rovnala 149597870.7 souřadnicím ve vizualizačním světě. Na takové hodnotě datový typ float ztrácí rozlišení a transformuje se do 1.495979e+08 (nižší řády vyplní nulami). Planetární systémy se navíc nerozkládají pouze na 1 AU, ale třeba i na 30 až 60 AU. Vizualizace na takových hodnotách ztrácí význam. Rozlišení pohybu na krátkých vzdálenost je v takovém případě nemožné, objekty se skokově pohybují v momentě, kdy urazí dostatečně velkou vzdálenost, kterou lze již rozlišit v aktuální velikost koordinátu.

Řešení tohoto problému naštěstí existuje a není nijak komplikované. Nazývá se **floating origin**[11, 12] a spočívá v posuvu souřadného středu vizualizace v rámci vizualizačního světa. Jinak řečeno - vše se pohybuje, zatímco uživatel stojí stále na 0,0,0. Tento způsob zajistí, že okolo kamery (uživatele) bude vždy dostatek prostoru na vizualizaci plynulého pohybu v řádech stovek metrů. Je jasné, že objekty, u kterých se uživatel nenachází, rozhodně nemají malé koordináty a prostrádají přesnost, nicméně to není na závadu, protože jsou tak daleko, že je není možné vidět. Poměr mezi x,y,z koordináty se neustále udržuje a tak je například pro prvky uživatelského rozhraní možné signalizovat jejich polohu a tedy nabídnout uživateli představu o směru, ve kterém se nachází.

Řešení uvnitř aplikace Aurora je již značně ulehčené faktem, že již od počátku je strikně oddělená simulace od vizualizace. To znamená, že interně probíhá simulace pohybu těles (konkrétně v datovém typu double) a vizualizace s těmito daty poté nakládá dle aktuální potřeby (jak často vypočítaná data sbírá, jak rychle se mají počítat další data, atd.). Floating origin jsem tedy realizoval v rámci aplikace relativně rychle. Potřeboval jsem si ale ujasnit nároky, které tato metoda potřebuje. Jestliže chci pohybovat s celou scénou, potřebuji mít prvně přehled o tom, jaké objekty ve scéně existují a které budou přítomné ve floating origin měřítku a které ne. Opět využívám Unity systém tagů, který mi jednak usnadňuje kategorizaci jednotlivých objektů a rovněž usnadňuje výsledné hledání, které budu muset provést, abych mohl s těmito objekty manipulovat. Jakmile je tento krok hotov, je možné přestoupit k implementaci samotné metody floating origin.

Implementace proběhla v řídící třídě hlavní kamery - *ControllerCam* v metodě *Update*. Jak jsem již zmínil výše, uživatel (kamera) zůstává na koordinátech 0,0,0 - to je víceméně pravda, kdybych ale takto každý snímek držel uživatele na jednom místě, nebylo by možné zjistit, kam se vlastně chce pohybovat. Proto jsem zavedl prahovou vzdálenost pohybu, po kterou dovolím uživateli (kameře) se volně pohybovat a zjistím tak vlastně směr, kterým se chce uživatel vydat. Algoritmus prakticky čeká, až uživatel (kamera) přesáhne tento vzdálenostní práh od 0,0,0. Jakmile se tak stane, tak se spustí část floating origin metody, která najde všechny objekty ve scéně, určené k přesunu, a těmto objektům změní pozici tak, že jejich pozici odečte

od aktuální pozice uživatele (kamery). Jakmile se tato úprava souřadnic provede pro všechny typy objektů, samotná kamera se vrátí na souřadnice 0,0,0. Takto funguje absolutní základ floating origin metody v aplikaci.

Měřítko jsem zvolil takové, že jedna souřadnice ve vizualizačním prostoru odpovídá 100km v reálném světě. Cílem do budoucna je toto meřítko posunout ještě výše, konkrétně na 1 km = 1 souřadnice. Nicméně budu muset přijít se systémem, který si lépe poradí s vykreslováním textur v blízkosti jednotlivých těles (každé těleso sice má sice textury s velmi vysokým rozlišením, čím větší ale bude, tím blíže se uživatel může dostat a tedy tím hůře bude výsledek vypadat) či rovnou dojde k implementaci trojrozměrného povrchu daného tělesa. Měřítko jsem tedy prozatím nechal na 100 km = 1 souřadnice. I to již dostatečně zobrazuje velikosti a vzdálenosti těles.

Přepnutí do tohoto módu se provádí ručně skrz tlačítko (jak jsem již zmínil výše) a o samotné přepnutí se opět stará jádro *AuroraVECore*, které dostane volání od třídy *ControllerGUI*. Provede se přepočítání pozice kamery z meziplanetárního na floating origin a nastaví se spínač, který povolí floating origin algoritmus, který je umístěn ve třídě *ControllerCam* a popsaný výše.

3.2 Problém n těles

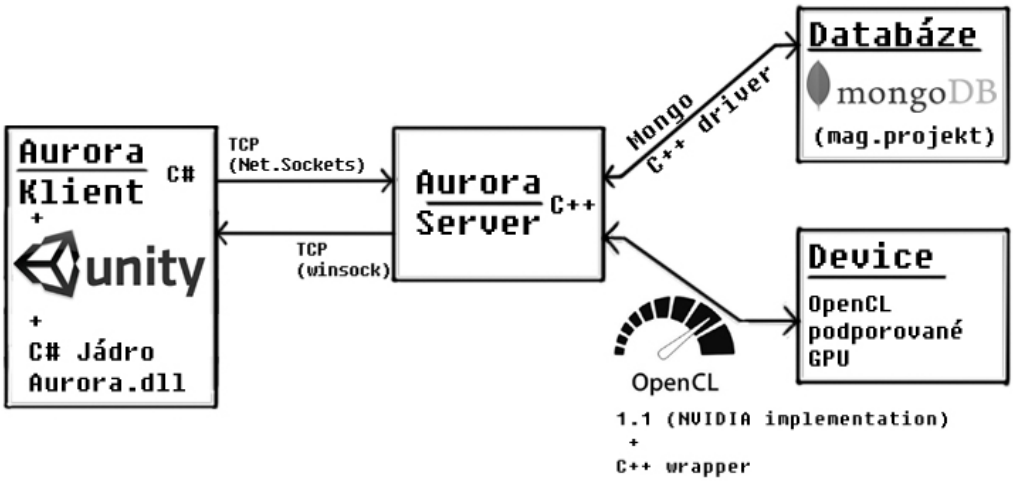
Přímá komunikace s OpenCL zařízením pod Unity enginem není prozatím možná. Abych tedy mohl výpočet problému n těles realizovat pomocí GPU, musel jsem navrhnout dedikovanou aplikaci. Tato aplikace se bude zaměřovat na master-slave komunikaci s výpočetním zařízením a klientem (aplikace Aurora pod Unity). Tuto aplikaci jsem nazval AuroraServer. Realizace serverové aplikace má v tomto případě jasné výhody - odpadá totiž řešení kompatibility OpenCL na všech klientských zařízeních. Stačí, že daný HW má server. Rovněž jsem byl schopen přes serverovou aplikaci odklonit komunikaci s NoSQL Mongo databází[9], ve které se ukládají všechna data, která jsou v klientské aplikaci k dispozici. Databáze je tímto způsobem schovaná a klienti komunikují pouze se serverem. Nicméně téma návrh a realizace databáze bylo součástí mého magisterského projektu[10] a v této práci ho tedy dál zmíním pouze skrz obrázek 3.4, který naznačuje základní pracovní bloky.

Nyní bych se chtěl zaměřit na popis algoritmů, které jsem navrhl pro serverovou aplikaci. Server aplikace je kompletně napsaná v jazyce C++ (na rozdíl od klientské části). Protože tvorba rosbustní serverové aplikace by svým rozsahem vydala na samostatnou diplomovou práci, aktuální implementace je schopna zpracovávat výpočetní požadavky pouze od **jednoho** klienta.

Z obrázku 3.4 vyplývá, že komunikace mezi klientem a serverem je skrz protokol TCP. Komunikace probíhá v ASCII. Na straně serveru využívám standardní Windows Socket API. Díky této knihovně jsem navrhl metody, které mi umožní komunikaci s klientem. Tedy metody, které budou posílat vlajky (flags), které se obvykle využívají ke změně stavu automatů na klientské či serverové části. Dále se jedná o posílání základních datových typů jako jsou int, float, double, std::string.

Komunikační pakety, jejichž formát sdílí klient i server, mají následující formát:

typ_paketu:data_k_odeslání\n



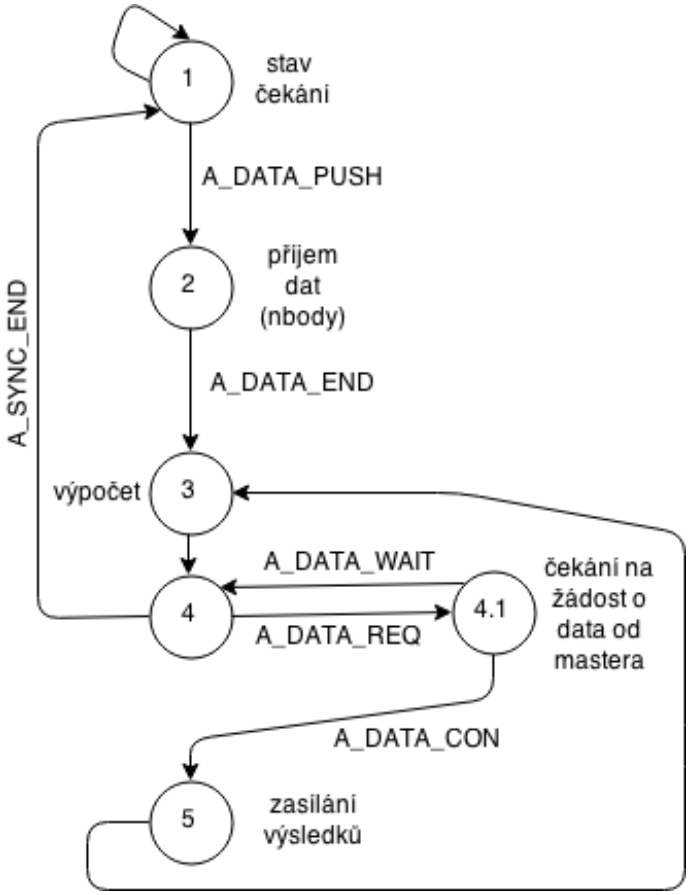
Obrázek 3.4: Jednoduchý náhled na jednotlivé pracovní bloky

typ_paketu mohou být typy, které jsou zadefinované ve výčtu PacketTypes. Jedná se o následující:

- **A_SYNC_END** (klient) restartování serverového automatu do stavu čekání
- **A_DATA_PUSH** (klient) posílám počáteční podmínky
- **A_DATA_END** (klient) konec počátečních podmínek
- **A_DATA_REQ** (klient) jsou nová data již připravena?
- **A_DATA_WAIT** (server) čekej na data, počítám
- **A_DATA_CON** (server) data připravené na odeslání
- **A_DEVICE_INIT** (klient) inicializuj OpenCL zařízení
- **A_DEVICE_SUCCESS** (server) zařízení nalezeno a úspěšně inicializováno
- **A_DEVICE_FAIL** (server) inicializace zařízení selhala

Ve výčtu PacketTypes se nachází popis i pro datové, databázové i debugovací pakety. Data k odeslání jsou pak uspořádaná v poli znaků.

O příjem paketů od klienta se stará metoda *receiveFromClients*, která se každý serverový snímek volá a případně zpracovává jakákoliv příchozí data. Jestliže se objeví čekající data na zpracování, *receiveFromClients* je zpracuje a nastaví potřebné



Obrázek 3.5: Schéma stavového automatu pro třídu AuroraServer

interní proměnné, které pak zpracuje stavový automat, realizovaný v metodě *Update*. Obrázek 3.5 popisuje funkci tohoto automatu.

Přepínání stavů, až na vyjímky, ovlivňuje klient (master), který dle aktuální potřeby vyžaduje po serveru (slave) výpočet a odeslání nových výsledků. Považuji za nutné zmínit fakt, že tento automat běží v jiném vlákně, než skutečná main funkce aplikace AuroraServer. V této funkci se totiž pracuje s třídou *ServerDevice*, o které se zmíním více níže. Důvodem umístění do jiného vlákna je fakt, že považuji za nepřípustné, aby byla schopnost serveru odpovídat klientovi na požadavky zpomalována výpočty problému n těles. Tyto výpočty totiž mohou trvat v řádech jednotek až desítek milisekund.

Nyní bych rád popsal funkci automatu v jednotlivých krocích a doplnil tak obrázek 3.5:

- **Stav 1** je stav čekání, ve kterém se server nachází ihned po startu či po vyresetování automatu pomocí volání A_SYNC_END
- Přechod do **stavu 2** je podmíněn přijetím paketu A_DATA_PUSH. Toto volání říká, že klient je připraven zaslat počáteční podmínky simulace. V tomto

stavu server nejdříve přijme celkový počet těles. Pomocí této informace inicializuje pole *systemBodies* ve třídě *ServerDevice*. Do tohoto pole pak vkládá jednotlivá data, která přijme.

- Server zůstává ve stavu 2 dokud nepřijme paket A_DATA_END, který značí, že klient již zaslal vše. Server se tedy může přesunout do **stavu 3**, který se nazývá výpočetním. Zde zavolá zařízení, konkrétně metodu *computeCycle*. Po tomto volání se automaticky přepne do stavu 4.

- Ve **stavu 4** server čeká na žádosti od klienta. Jestliže přijde A_DATA_REQ, server zkontroluje, zda zařízení má již spočítaná data. Pokud ano, odešle klientovi A_DATA_CON. Pokud ne, odešle klientovi A_DATA_WAIT.

- **Stav 5** spočívá v zasílání výsledků zpět klientovi. Po odeslání všech potřebných dat z *systemBodies* se server automaticky přepne do stavu 3 a zažádá o nový výpočet zařízení.

Na řadě je nyní třída *ServerDevice*. Tato třída je zodpovědná za samotné řešení problému n těles. Tento problém by měl být řešen pomocí OpenCL zařízení, nicméně jestliže inicializace OpenCL zařízení selže (žádné zařízení nelze najít, chyba v kompilaci kernel kódu,..), bude se pro výpočet používat standardně CPU. Toto chování může být přepsáno definicí v konfiguračním souboru serveru (více v příloze C).

OpenCL zařízení inicializuji v konstruktoru třídy *ServerDevice*. Protože je tato operace časově náročná, snažím se zde o implementaci všech věcí, které je možné připravit předem. Nejdříve je třeba získat platformy, které jsou dostupné na daném pc. Toho mohu dosáhnout pomocí metody cl::Platform::get. V dané platformě pak vyberu konkrétní zařízení a vytvořím pole cl_context_properties. Toto pole pak použijeme k tvorbě kontextu (cl::Context). Kontext OpenCL používá pro tvorbu všech důležitých částí programu. Pro zvolené zařízení a kontext vytvořím frontu pro příkazy (cl::CommandQueue). Následuje deklarace samotného kernel kódu.

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n
__kernel void nbody_step(
    const int n,
    __global double4 *buffer_posistion,
    __global double4 *buffer_velocity)
{
 int i = get_global_id(0);
 double4 p = buffer_posistion[i];
 double4 v = buffer_velocity[i];
 if (v.w == 0.0) {
  double4 a = (double4)(0.0, 0.0, 0.0, 0.0);
  double dt = 0.001, eps = 0.0001, comp = -5.6058567e8;
  for (int k = 0; k < 2500; k++) {
   for (int j = 0; j < n; j++) {
    if (buffer_velocity[j].w == 0.0) {
     double4 d = p - buffer_posistion[j];
     double invr = rsqrt(d.x*d.x + d.y*d.y + d.z*d.z + eps);
```

```
      double f = (buffer_posistion[j].w) * (invr * invr * invr) * (comp);
      a += f * d;
     }
    }
    v += (dt * a);
    p += (dt * v);
    barrier(CLK_GLOBAL_MEM_FENCE);
    buffer_posistion[i] = p;
    buffer_velocity[i] = v;
   }
  }
}
```

Nyní popíši funkci tohoto kernel kódu. Masivní paralelismus spočívá v rozdělení práce. V mém případě bude každý kernel (jádro) zastupovat jedno těleso v daném systému.

Kód začíná povolením cl_khr_fp64, což mi umožňuje využívat datový typ double, double4 (vektor x,y,z,w) a celkové výpočty ve dvojité přesnosti. Datový typ double je pro tento konkrétní případ nutností, protože jsem si již při vývoji ověřil (pomocí sekvenční implementace na CPU), že při použití datového typu float dochází k příliš velkému zanášení chyb do simulace. Grafická karta nicméně toto rozšíření musí podporovat - použitá grafická karta musí umět počítat s datovým typem double.

Následuje deklarace samotné procedury pro kernel a vstupní parametry:

- **n** je počet těles - algoritmus v každém kernelu potřebuje vědět, kolik těles vstupně výstupní buffery obsahují
- **buffer_position** je READ/WRITE buffer, datového typu double4, ve kterém je uložena počáteční pozice tělesa a do kterého se bude ukládat i přepočítaná pozice tělesa
- **buffer_velocity** je READ/WRITE buffer, datového typu double4, ve kterém je uložena počáteční rychlost tělesa a rovněž se zde uloží i přepočítaná rychlost tělesa

Abych co nejvíce omezil počet čtení a zápis bufferů, zvolil jsem datový typ *double4*. Tento vektorový datový typ mi totiž umožní do čtvrtého prvku (.w) uložit další parametry, nutné pro simulaci. V *buffer_position* se v prvku w nachází aktuální hmotnost a v *buffer_velocity* pak informace o tom, zda již dané těleso kolidovalo či ještě ne.

Funkce *get_global_id(0)* vrací identifikátor kernelu. Proměnná tedy řekne, které těleso daný kernel reprezentuje. Následující 2 řádky pak značí výběr pozice a rychlosti dle daného identifikátoru.

Následuje kontrola čtvrté hodnoty ve vektoru rychlosti, která značí, zda-li dané těleso již kolidovalo či ne (0 = nekolidovalo). Jestliže těleso již kolidovalo, kernel nebude vůbec měnit obsah bufferů s pozicemi a rychlostmi daného tělesa. Ve vizualizaci tělesa po kolizi již stojí tam, kde kolidovala a díky této podmínce již nezasahují do simulace.

Následuje inicializace vektoru zrychlení a definice 3 konstant:

- **dt** je časový krok.
- **eps** je konstanta, která zajišťuje numerickou stabilitu (pokud vzdálenost vyjde blízká nule, vyjde v situaci, kdy druhé těleso je to samé či jsou velmi blízko).
- **comp** je gravitační konstanta pro přeškálované hodnoty, je záporná, protože gravitační síla je přitažlivá.

První for cyklus odpovídá časové integraci. Vzhledem k tomu, že se snažím o co největší přesnost simulace, musím volit časový krok dt dostatečně malý (v mém případě 0,001). Avšak nepotřebuji (ani by to nebylo efektivní) posílat výsledky z každé časové hladiny, proto nechávám vždy nasimulovat 2,5s a následně na klientské straně provádím interpolaci.

Pro řešení diferenciálních rovnic tvaru (2.2) používám Eulerovu metodu. Označíme si $\vec{r}_i^{(k)}$ resp. $\vec{v}_i^{(k)}$ jako polohu resp. rychlost í-tého tělesa v k-tém časovém kroku. Potom lze přepsat daná diferenciální rovnice 2. řádu přepsat na soustavu dvou diferenciálních rovnic prvního řádu (3.2) s původními počátečními podmínkami (2.3).

$$\begin{aligned} \frac{d(\vec{v}_i)}{dt} &= -G \sum_{j=1, j\neq i}^{n} \frac{m_j}{{r^3}_{ij}} \vec{r}_{ij} \\ \frac{d\vec{r}}{dt} &= \vec{v}_i \end{aligned} \tag{3.2}$$

Numerické schéma odvozené pomocí Eulerovy metody pro danou diferenciální rovnici vypadá následovně:

- $\vec{v}_i^{(k+1)} = \vec{v}_i^{(k)} + dt \left(-G \sum_{j=1, j\neq i}^{n} \frac{m_j}{{r^3}_{ij}} \vec{r}_{ij} \right)$
- $\vec{r}_i^{(k+1)} = \vec{r}_i^{(k)} + dt\, \vec{v}_i^{(k+1)}$

Vnitřní for cyklus probíhá všechny ostatní tělesa a napočítává jejich příspěvky ke zrychlení i-tého tělesa. Následně jsou vypočteny hodnoty na vyšší časové hladině. Kernel pak čeká na ostatní až dokončí tento cyklus. To je zajištěno voláním *barrier(CLK_GLOBAL_MEM_FENCE)*. Jakmile všechny kernely synchronizují, vypočítané hodnoty se uloží do globálních bufferů, ze kterých se data využívají na další iteraci. Pokud bych toto čekání neprováděl, mohlo by se stát, že by se při simulaci používaly hodnoty z různých časových hladin, což může mít nežádoucí účinky na její výsledky.
Kód je deklarován přímo ve třídě *ServerDevice*, konkrétně v proměnné datového typu std::string. Tento kód následovně převedu na pole znaků pomocí vestavěné metody std::string - c_str() a vložím ho do konstruktoru cl::Program::Sources. Jestliže mám vytvořenou instanci cl::Program::Sources, mohu vytvořit instanci samotného programu s daným zdrojovým kódem a na daném kontextu. Tento program následně mohu zkompilovat pomocí metody build ve třídě cl::Program. Jestliže metoda build

vrátí nulu, kompilace kernel kódu byla úspěšná a je možné vytvořit samotnou instanci cl::Kernel pomocí zkompilovaného programu a názvu kernel metody (v našem případě *nbody_step*).

Jestliže inicializace selže (ať už v jakémkoliv kroku), algoritmus nastaví proměnnou *initialized* na hodnotu false. Výpočet problému n těles bude pak zpracován pomocí CPU, konkrétně pomocí metody *computeCycleOnCPU*. Tato metoda je víceméně podobná kernel kódu (který jsem do detailů popsal výše). Důležitou součástí je nicméně přítomnost třetího for cyklu, který nahrazuje paralelní zpracování. Metoda *computeCycleOnCPU* je tedy zpracována zcela sekvenčně. Má rovněž menší rozlišení oproti výpočtům na GPU, konkrétně 250 iterací s časový krokem 0,01. Důvodem je rychlost. Sekvenční zpracování by totiž při časovém kroku 0,001, použitém na GPU, trvalo pomocí CPU v řádech stovek milisekund (nepřijatelné pro simulaci v reálném čase).

Jak jsem již zmínil výše, *ServerDevice* obsahuje alokované pole *systemBodies*, kde délka odpovídá aktuálnímu počtu těles. Nezmínil jsem ale zatím datový typ. Jedná se o strukturu *body*. Atributy této struktury jsou k vidění na obrázku 3.6.

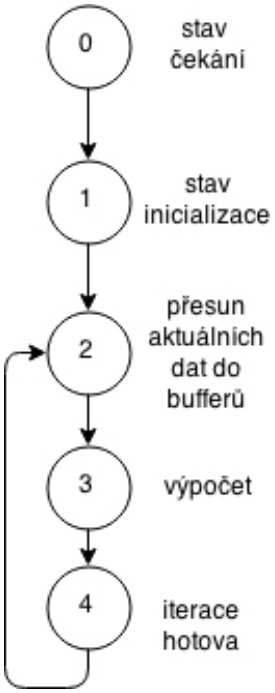
```
struct body {
        double x,y,z;           // aktualni pozice
        double nx,ny,nz;        // nova pozice
        double vx,vy,vz;        // rychlost
        float mass;             // hmotnost
        float collided;         // kolize (0,1?)
};
```

Obrázek 3.6: Atributy datového typu bodystruct

Algoritmus, který se stará o řešení problému n těles (umístěn ve třídě *ServerDevice*), využívá všechny atributy v této struktuře. Příjem a odesílání zpět ke klientovi se týká nicméně pouze některých atributů. Při přijímání dat se aktualizuje pozice (x,y,z), rychlost (vx,vy,vz), hmotnost (mass) a informace, zda-li těleso již kolidovalo či ne (collision). Příjem se provádí pouze jednou. *ServerDevice* přijímá počáteční podmínky simulace a pak již pracuje se svými daty a ty sdílí skrz třídu *AuroraServer* s klientem. Server klientovi regulérně odesílá zpět pouze jednotlivé pozice (x,y,z).

Třída *ServerDevice* má, stejně jako třída *AuroraServer*, svůj vlastní řídící stavový automat. Tento automat se nachází v členské metodě *DeviceLoop*. Opět jsem vytvořil jednoduché schéma, které naznačuje jeho funkčnost. Toto schéma je k vidění na obrázku 3.7 Stavový automat ve třídě *ServerDevice* řídí automat ve třídě *ServerDevice*. Jednotlivé funkce stavů se liší dle toho, zda OpenCL zařízení bylo incializováno úspěšně či ne. Nyní jednotlivé kroky v obrázku 3.7 popíši. Nejdříve pro situaci s CPU, kde je podstatně méně práce:

- **Stav 0** je základní stav, ve které zařízení čeká.



Obrázek 3.7: Schéma stavového automatu pro třídu ServerDevice

- **Stav 1** je stav inicializační. Značí, že simulační data jsou již připravena v datovém poli *systemBodies*. Tento stav v tomto případě přepne automat do stavu 3.
- **Stav 2** není aktivní v tomto případě (data jsou pro CPU přístupná přímo bez nutnosti přesunů).
- **Stav 3** je výpočetní stav. V tomto případě se volá metoda *computeCycleOnCPU*.
- **Stav 4** signalizuje hotový výpočet a fakt, že data byla přečtena třídou *AuroraServer* (zaslána zpět klientovi). Automat se v tomto případě vrací do stavu 2.

S OpenCL zařízením:

- **Stav 0** je základní stav, ve které zařízení čeká.
- **Stav 1** je stav inicializační. Značí, že simulační data jsou již připravena v datovém poli *systemBodies*. Pro daný cl::Context se inicializují zásobníky typu cl::Buffer, konkrétně pak zásobníky pro pozici (buffer_position) a pro rychlost (buffer_velocity), všechny o velikostech počet_těles * sizeof(cl_double4). Tyto zásobníky následně přiřazujeme cl::Kernel jako argumenty samotné kernel funkce. Rovněž přiřadíme kernelu celkový počet těles (numberOfBodies).

- **Stav 2** se zaměřuje na samotný přesun aktuálních dat do předem vytvořených bufferů. Přesun je podmíněn vytvořením lokálních polí cl_double4 *input_position, *input_velocity, ve kterých se nacházejí všechna data, důležitá k simulaci nového kroku. Tato pole přiřadí do bufferů za pomoci příkazové fronty a příkazu enqueueWriteBuffer.

- **Stav 3** je výpočetní stav, ve kterém se volá metoda *computeCycleOnGPU*. Tato metoda již nastaví počet vláken pomocí cl::NDRange global(počet_těles) a rozměr, ve kterém budeme výpočet provádět pomocí cl::NDRange local(počet_těles) (je nutné mít všechny vlákna v jedné pracovní skupině - aby fungovala synchronizační bariéria). Do vytvořené příkazové fronty pošleme příkaz enqueueNDRangeKernel(kernel, cl::NullRange, global, local) a tím tedy spustíme vlastní výpočet. Metoda *computeCycleOnGPU* má rovnou na starosti i samotný výběr dat. Výběr se opět provádí skrz příkazovou frontu, konkrétně s příkazem enqueueReadBuffer. Tomuto kroku předchází ale alokace polí *output_position a *output_velocity, do kterých se přesune obsah bufferů buffer_position a buffer_velocity. Obsah dočasně alokovaných polí *output_position a *output_velocity se pak přesune do *systemBodies* a tímto se zakončí výpočet.

- **Stav 4** signalizuje hotový výpočet a fakt, že data byla přečtena třídou *AuroraServer* (zaslána zpět klientovi). Automat se vrací v tomto případě do stavu 2.

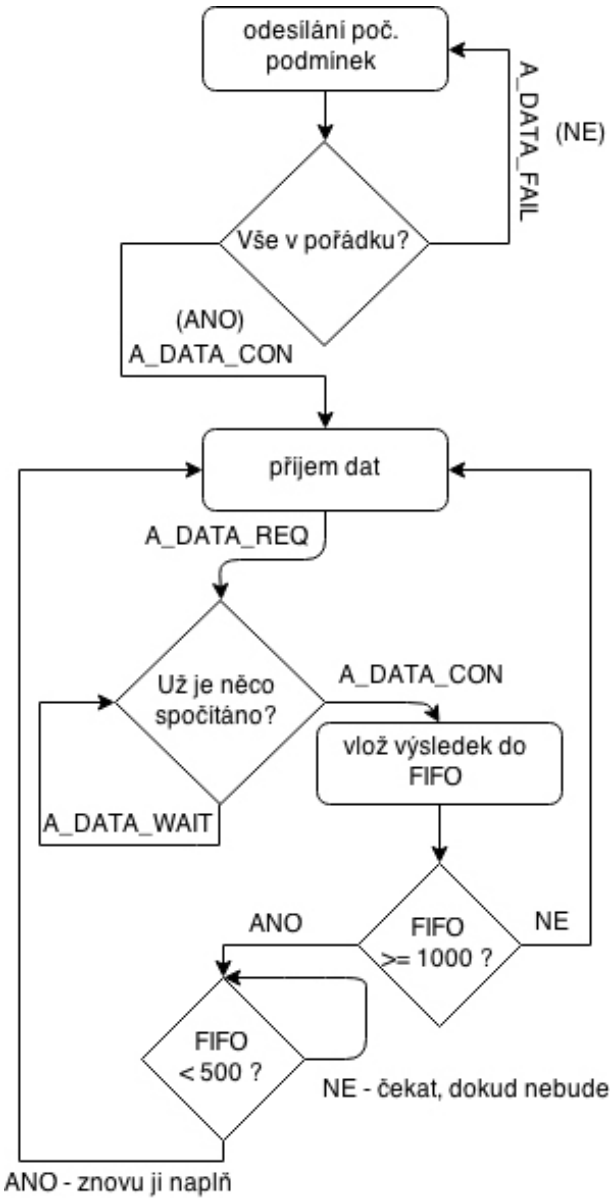
Klientská strana tohoto problému je opět založena na algoritmu, který je umístěn v jiném vlákně. Metoda *Update*, která běží v jiném vlákně, se nachází ve třídě *Aurora.Physics* (třída, která je nainstancována s každou třídou *Aurora.StarSystem*). Obrázek 3.8 popisuje základní kroky, které metoda *Update* provádí v návaznosti na problematiku řešení problému n těles.

Jestliže se počáteční podmínky pro každé těleso v planetárním systému automaticky aktualizují každý snímek, algoritmus řízení simulace problém n těles má data, potřebná k zaslání počátečních podmínek na server. Jakmile tedy uživatel o změnu simulace na problém n těles zažádá, výše zmíněné vlákno (metoda *Update* ve třídě *Aurora.Physics*) přepne režim z problému 2 těles na problém n těles.

Nejprve klient pošle serveru A_DATA_PUSH, poté počet těles v současném systému. Pak nastane čas posílat jednotlivé souřadnice a rychlosti těles. Pokud se v této fázi něco nepodaří, restartuje se odesílání pomocí paketu A_DATA_FAIL. Pokud bude vše v pořádku, klient pošle A_DATA_END a čeká na příjem paketu A_DATA_CON ze serveru.

Algoritmus klientského vlákna se nyní přepne do části, ve které přijímá data, která zpracuje server z již odeslaných počátečních podmínek. Aby výpočet na serveru probíhal nezávisle na tom, jakou frekvencí vizualizace odebírá jednotlivé výsledky výpočtu problému n těles, navrhl jsem ve třídě *Aurora.Physics* tzv. nbody fifo (frontu, ve které se dají ukládat pole struktur typu *body*). Ve třídě *Aurora.Physics* jsou dále definované dolní a horní limity pro tuto frontu. Pro aktuální verzi jsem zvolil 500 a 1000. To znamená, že klientské vlákno bude vyžadovat po serveru data tak

dlouho, dokud frontu nenaplní tak, že bude obsahovat 1000 výsledků výpočtů ze serveru. Zatímco algoritmus, který se stará o tvorbu polí se strukturami typu *body*, plní frontu. Vizualizace tuto frontu naopak vyprazdňuje. Konkrétně pomocí metody *UpdateNBPositions* ve třídě *Aurora.Physics*. Tato metoda vyjme aktuální pole z fronty a aktualizuje pozice jednotlivých těles v jejich třídách *Aurora.KeplerianOrbit*. Vizualizace pak hodnoty z *Aurora.KeplerianOrbit* přebírá a provádí lineární interpolaci mezi jednotlivými iteracemi. Jakmile počet výsledků výpočtů bude ve frontě menší než 500, klientské vlákno opět začne žádat o data, dokud nebude opět 1000 výsledků výpočtů.

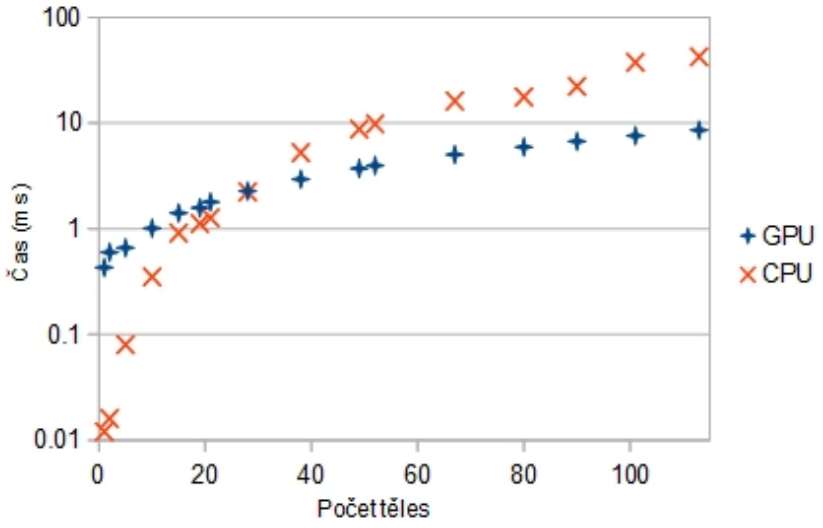


Obrázek 3.8: Diagram základní logiky klientské části simulace n těles

Další důležitou součástí jsou kolize. Pokud bych kolize ignoroval, znamenalo by to, že bych automaticky připouštěl zavádění chyb do simulace. V aktuální verzi řeším zatím jednoduché kolize, které se zakládají na změně hmotnosti. Modifikace vektoru rychlosti tělesa po kolizi zatím implementována nebyla.

Kolize je v rámci simulace asynchronní událost. Pro detekci kolizí využívám přímo Unity. Každé těleso má komponentu typu *Collider* a ve třídě, která toto těleso spravuje, pak metodu *OnTriggerEnter*. Tato metoda vyhodnotí, zda je v dané kolizi těleso méně hmotné (v tom případě nastaví boolean proměnnou collided na true), či je to hmotnější (v tom případě zvýší svou aktuální hmotnost o hmotnost tělesa, se kterým kolidovalo). Tato událost následně vyvolá resynchronizaci dat se serverem. Ta se provádí co nejdříve (jakmile se dokončí aktuání výpočet) pomocí paketu A_SYNC_END.

Rychlost výpočtu na CPU a GPU lze velmi dobře porovnat, protože obě metody - *computeCycleOnGPU* a *computeCycleOnCPU*, měřím pomocí metody *omp_get_wtime*, což je časovač, který měří a zároveň bere v potaz změnu kontextu. Mohl jsem tedy otestovat, jak/jestli je vlastně výhodné počítat problém n těles pro aplikaci Aurora právě pomocí GPU. Výsledky jsou k vidění na obrázku 3.9.



Obrázek 3.9: Srovnání výkonu CPU a GPU na výpočtech pro aplikaci Aurora

Tyto výsledky jsem naměřil na procesoru Intel Core i7-4710HQ a na grafické kartě NVIDIA GeForce GTX 860M s jádrem GM107 série Maxwell. Data, která byla použita při tomto měření, pocházejí z generátoru systémů, kterým aplikace Aurora disponuje. Byly vybrány systémy od těch nejednodušších po ty nejsložitější, na které je možné v aplikaci narazit. Hodnota času vznikla jako průměr rychlosti 1000 výpočtů (tj. prvotní naplnění klientské datové fronty).

Z výsledků je vidět, že masivní paralelismus problému n těles vskutku urychluje celkové řešení. Prakticky přes všechny testovací systémy je čas, potřebný k řešení,

menší než 10 milisekund. Na menších systémech (20 a méně) dominuje ale CPU. To je způsobené faktem, že pro výpočet na GPU je nutné provést zápis a čtení z bufferů (příkazy *enqueueWriteBuffer* a *enqueueReadBuffer* pro příkazovou frontu). Tyto operace trvají v řádech desetin milisekund (zjištěno pomocí diagnostického doplňku Nsight[14] pro Visual Studio) a tvoří podstatnou část řešícího cyklu pro malé systémy.

3.3 Tvorba rozdílně vypadajících těles

Při realizaci jsem se rozhodl, že je nutné vytvořit dostatečně robustní algoritmus, který lze bez problému aplikovat na jakékoliv těleso. Nicméně hlavně kvůli větším časovým nárokům, které bych primárně musel vynaložit na výběr vhodných textur, jsem se zaměřil na tělesa, která změnu potřebují úplně nejvíce. Jedná se o tělesa s atmosférou v obyvatelné zóně. Tyto druhy těles jsou klenotem, který rozhodně v jakémkoliv systému stojí za prozkoumání. A bylo by velmi nevhodné, kdyby těleso používalo stejné textury, které byly již na předešlé planetě, kterou uživatel navštívil. Z pohledu databáze se jedná konkrétně o tělesa s atmosférou, s teplotní třídou Warm, obyvatelnostní třídou hypopsychroplanet, psychroplanet, mesoplanet či thermoplanet.

Generování a úpravy masky

Jak jsem již zmínil v analýze, cílem není vytvořit komplexní trojrozměrný model terénu na planetě, povrchy nadále zůstanou dvourozměrné. Vygenerovaný šum je tedy nakonfigurován takovým způsobem, aby vracel šum, který se podobá co nejvíce masce. Na masce by převážně měly převládat rgb hodnoty 1 a 0. Přechody mezi by měly být plynulé, nicméně mělo by se tak stát pouze na krátkém úseku v rámci pár pixelů. Obrázek 3.10 naznačuje, jak vypadá výsledný šum, který bude aplikace používat.

Tato maska bude pak použita v modifikovaném shaderu pro povrch těles tak, že dle ní bude míchat například texturu vody a vhodnou texturu země. Přičemž černá bude odpovídat vodě a bílá zemi. O potřebných modifikacích shaderu pro tělesa se zmíním později.

Zmínil jsem parametr seed. Tento parametr zajišťuje, že výstupní šum bude **vždy stejný**, čili že generátor náhodných čísel (v tomto případě je to třída *UnityEngine.Random*) bude číst z určité série pseudonáhodných čísel. Tento systém se mi velmi hodí, protože chci, aby planeta po každém startu aplikace vypadala tak, jak vypadala, když se poprvé vygenerovala. Měla by rovněž vypadat úplně stejně na jakémkoliv zařízení. Vytvoření instance třídy *PerlinNoise* je podmíněno právě předání argumentu seed do jejího konstruktoru. Jakmile mám vytvořenou instanci, je možné využívat metody, které tato třída nabízí. Jedná se konkrétně o metody *FractalNoise2D* a *FractalNoise1D*. Tyto metody přijímají parametry o aktuální pozici v šumu (x a případně i y, záleží, zda využíváme dvourozměrný či jednorozměrný), počet oktáv (jak jsem již zmínil v popisu fBm v analýze), frekvenci (jak často bude docházet ke změnám) a amplitudu (jak velké rozdíly budou mezi jednotlivými

výstupními hodnotami).

Ve třídě *staticMethods* (v klientské části) byla vytvořena nová metoda, kterou jsem nazval *generateSurfaceMask*. Tato metoda přijímá instanci třídy *PerlinNoise* (čili reference na již připravenou třídu s určitým seedem) a pak hodnotu šířky a výšky výsledné textury šumu. Nicméně kvůli způsobu implementace celkové tvorby masky, kterou popíšu dále, tato metoda vrací pouze pole typu *Color*, které má délku šířka * výška. V metodě *generateSurfaceMask* je zadefinovaná počáteční pozice šumu, která je ve všech případech stejná - různorodost tedy ovlivňuje pouze seed. Nicméně využití stejného seedu a změna pozice v šumu je velmi silným nástrojem, který mám v plánu do budoucna využívat. Tento algoritmus by mi umožnil zajistit

Obrázek 3.10: Ukázka výstupního šumu, zde konkrétně pro seed=12460

to, že generované systémy budou vypadat vždy stejně a jejich podobu lze ovlivnit jednoduchou změnou jednoho čísla - seedu.

Metoda *generateSurfaceMask* obsahuje 2 hlavní for cyckly, které prochází všechny pozice v připravované textuře a na každé x,y pozici zjišťují hodnotu *FractalNoise2D*, kde pozice je rovna iteračním proměnným for cyklů.

Když tento základní algoritmus procházel testováním, bylo jasné, že mapování takto vypadající masky na kouli bude mít jisté vizuální vady. Problematické části jsou 3 - severní, jižní pól a kraj, kde se setkává levý a pravý kraj textury masky. Jak jsem již zmínil, maska bude určena k míchání dvou typů textur - černá barva bude odpovídat moři, bílá bude odpovídat zemi. Rozhodl jsem se pro implementaci algoritmu, který bude ovlivňovat výstupní masku již při získávání jednotlivých hodnot ze šumu.

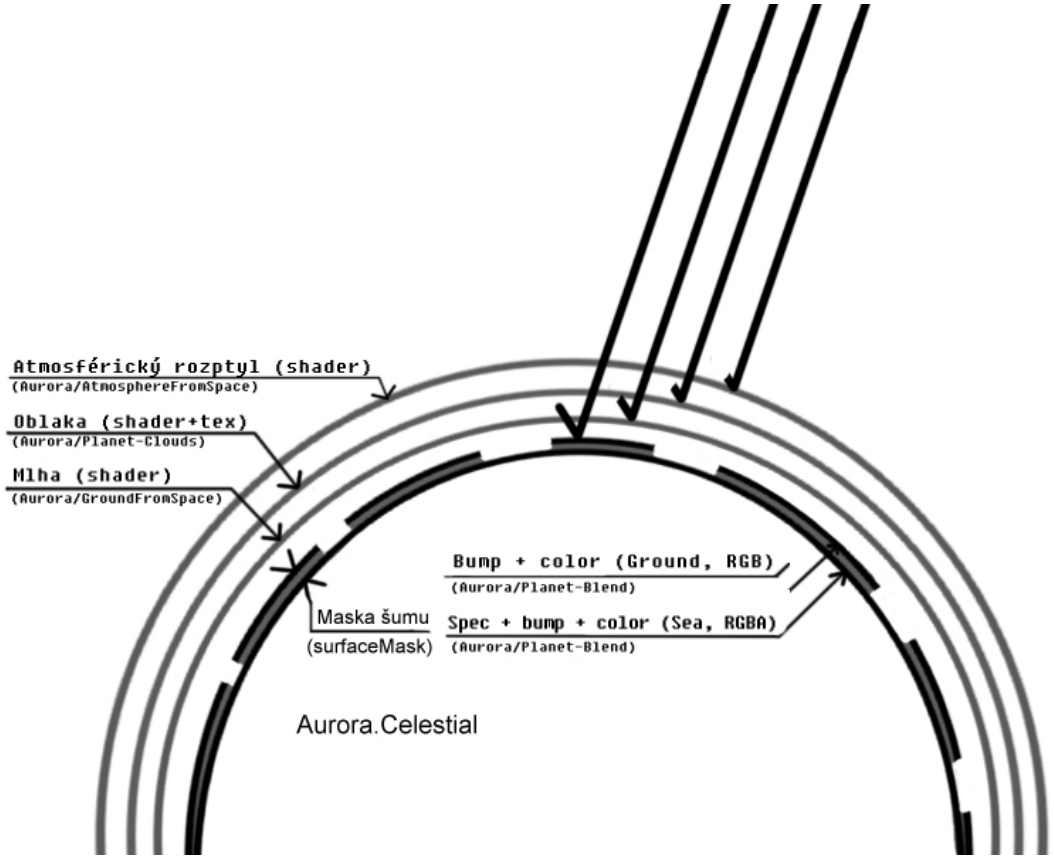
Nejdříve jsem vyřešil póly. Protože polární čepičky jsou obsažené přímo v textuře vody, není třeba řešit masku v oblastech pólů. Proto zde existuje limit, který je roven $\frac{1}{9}$ výšky výsledné masky a udává, že za tento se již hodnoty ze šumu získávat nebudou a místo nich použijeme přímo černou barvu. Nicméně aby okraj nebyl příliš rovný a nepřirozený, k jeho narušení jsem využil jednorozměrného šumu (*FractalNoise1D*), který předem nastavený limit (výše zmíněná $\frac{1}{9}$) upravuje. Výsledkem tedy je vcelku zajímavé pobřeží. Na obrázku 3.10 je již možné vidět tuto změnu - horní a dolní část masky obsahuje černé pásy, které jsou právě vyhrazeny polárním čepičkám. Pobřeží, které je ovlivněné jednorozměrným šumem, se může zdát v této projekci nezajímavé, nicméně je nutné brát v potaz, že při mapování takovéto textury na kouli dojde ke smrštění jednotlivých hodnot čím blíže k pólu jsou.

Složitějším problémem je levý a pravý kraj masky. Masku mapuji na kouli a tyto kraje musí tedy být opakovací. Tuto úpravu jsem musel provést až po vygenerování masky v další dvojici for cyklů. Cílem této úpravy bylo buď rozpojit či spojit bílou barvu (nebo barvu blízkou bílé) na těchto krajích. Byly zadefinovány limity - jak pro barvu (jaká šedá je ještě bílá?), tak pro velikost (je spojení široké pouze pár pixelů či se jedná o nějaký širší kus země - bílé barvy v masce?). Jestliže algoritmus považoval za vhodné v daném místě kraje rozpojit, opět k tomuto kroku využil jednorozměrného šumu (*FractalNoise1D*). Ve většině případů si s problémem tento algoritmus poradí celkem dobře a zbaví se toho, čeho chci - ostré přechody na krajích masky. Nicméně občas je možné, že vytvoří nepřirozené přechody.

Použití masky

Maska sice již připravená je, nicméně abych ji mohl využít k míchání dvou textur, je nutné ji zakomponovat do shaderu, který se stará o povrch tělesa. Obrázek 3.11 znázorňuje, jakým způsobem fungují v aktuální verzi vrstvy každého tělesa.

Jak je vidět na obrázku 3.11, *Aurora.Celestial* je objekt, který se ve scéně skládá hned z několika objektů, které dohromady tvoří strom objektů (mimo již zmíněných na obrázku zde můžeme například najít i objekt, který reprezentuje prstenec planety). Unity API umožňuje k těmto potomkům přistupovat skrz *Transform.Find*. Přiřazená třída *Celestial* má pak skrz metodu *initializeChildNames* najít a referencovat všechny potomky, se kterými bude třída *Celestial*, resp. její potomci jako např.



Obrázek 3.11: Ukázka jednotlivých vizuálních vrstev pro tělesa třídy Celestial

CelestialPlanet, pracovat a modifikovat je dle toho, jaké těleso zrovna bude reprezentovat. Mohu takto například vypínat atmosférické efekty[18, 19], měnit texturu mraků, atd. Je tak zajištěna potřebná konfigurovatelnost objektu.

Tyto interní reference samozřejmě využívám při řešení tohoto problému. Zajímá mě pak hlavně ta nejnižší část, která, stejně jako předchozí vrstvy, je koule. Nicméně má již přiřazen neprůhledný materiál, který vzniká ze shaderu *Aurora/Planet-Blend* (založen na unity surface shaderu). Tento shader vznikl ze zásadních úprav shaderu, který se v původní verzi staral o vizualizaci terénu. Úpravy spočívaly hlavně v implementaci míchání dvou textur (voda+země) dle masky a ve změně původních algoritmů tak, aby shader zachoval původní vizuální kvalitu. Při úpravách jsem zjistil, že pro ponechání všech původních efektů - tedy odrazivá (spekulární) složka na vodě, normálová (plastičnost) složka terénu a noční světla, bylo nutné ke stávající masce vygenerovat i invertovanou masku. Barevná složka pak vznikla jako

$$(texMsk * texGnd.rgb) + (texMskInv * texSea.rgb)$$

kde texMsk je maska, texMskInv je invertovaná maska, texGnd je textura země a texSea je textura vody. Míra odrazivosti, definovaná v alfa kanálu textury vody, byla rovněž pronásobena s invertovanou maskou. Normálová složka je výsledkem součtu

$$(tex2D(\_BumpTex, IN.uv\_GroundTex) * texMsk) +$$
$$(tex2D(\_NoBumpTex, IN.uv\_GroundTex) * texMskInv)$$

který, dle aktuální pozice, buď vrátí hodnotu z textury země (_BumpTex), či vrátí hodnotu z textury, kde jsou všechny barvy konstantní (čili na daném místě nevytvoří efekt plastičnosti). Výpočet normálové složky je tedy opět řízen původní a invertovanou maskou.

Ze strany samotného kódu, který se stará o řízení vzhledu daného tělesa, jsem implementoval novou logiku. Metodu *initializeVisuals* ve třídě *Celestial* jsem modifikoval tak, aby jednak podporovala novou strukturu objektů a rovněž byla schopna případně pracovat právě s maskou (privátní atribut *surfaceMask* datového typu *Texture2D*). Všechna tělesa standardně řídí svůj vzhled dle toho, jaká data mají uložené ve svém atributu *TextureID*. Jestliže je tento atribut roven *predef*, znamená to, že konfigurace vizuálních charakteristik jsou manuálně zapracované přímo do kódu (konkrétně v metodě *initializeVisuals*). Jestliže je ale TextureID atribut roven například 2-3, znamená to, že má algoritmus v *initializeVisuals* načíst druhou texturu povrchu a třetí texturu oblak z databáze textur pro daný druh objektu (velikostní a teplotní třídu). Tento algoritmus jsem v aktuální verzi rozšířil o zpracování masky. Jakmile dojde ke zpracování atributu TextureID, algoritmus v *initializeVisuals*, zkontroluje, zda atribut TextureID má i třetí část (čili např. 2-3-123456). Třetí část pak definuje právě již výše zmíněný seed, pomocí kterého je vytvořena třída šumu, která se později využije ke generování masky. Pokud se stane, že atribut nemá zadefinovaný seed a přeci vyhovuje podmínkám, bude danému tělesu seed náhodně vygenerován. Tento algoritmus se dá jednoduše rozšiřovat pomocí dalších bloků case, které budou spravovat více další druhy těles. Jak jsem již zmínil výše, rozvoj tohoto systému je hlavně podmíněn grafickou prací (tvorba textury pro vodu) a důsledným výběrem hodných textur pro povrch (země).

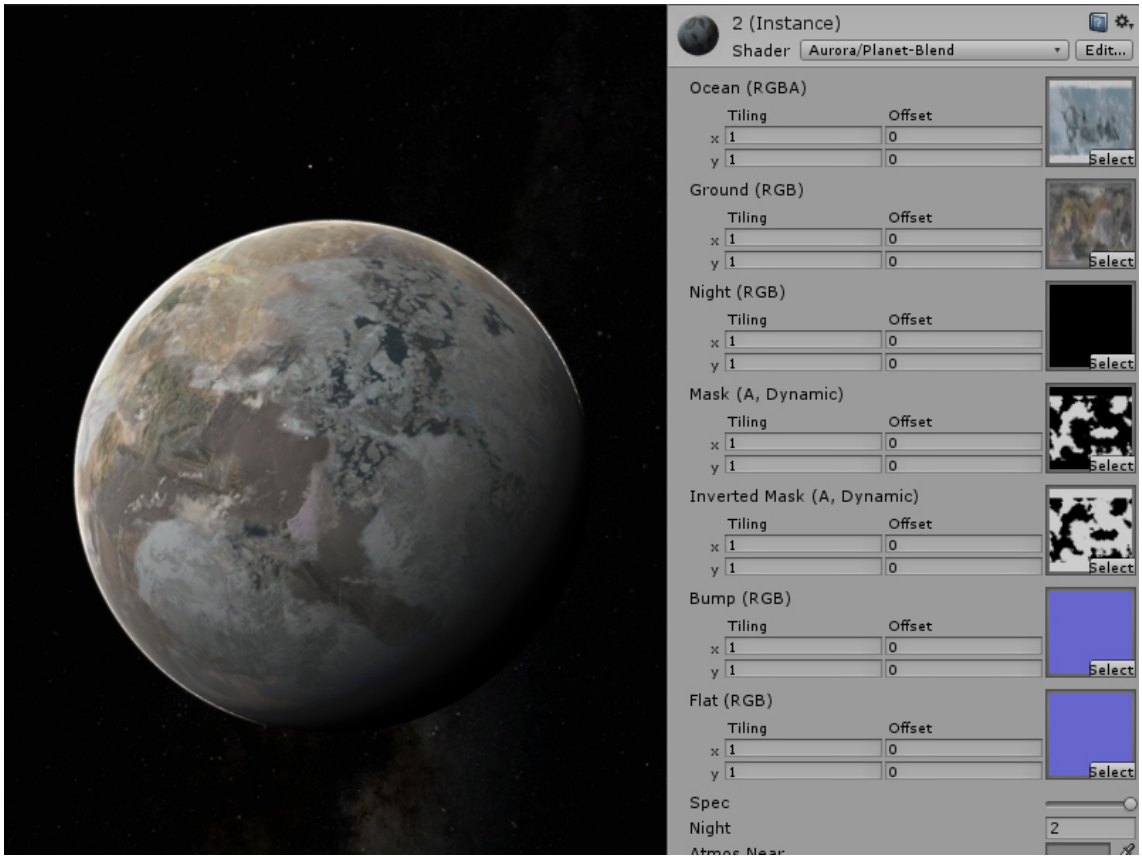
Důležité je rovněž zmínit, že díky změnám v shaderu pro planety je nyní nutné přiřazovat masku i na objektech, na kterých se síla tohoto algoritmu vůbec nepoužívá. Toto přiřazení je podmíněné faktem, že bez přiřazení masky by se povrch špatně vykresloval. Tento problém lze vyřešit přiřazením kompletně černé masky o velikosti 2x2 pixelů.

V neposlední řadě musím zmínit, že tvorba masky v reálném čase může značně zpomalit běh (jakmile je v systému třeba i více těles, které mají masky). Maska musí mít ideálně alespoň poloviční rozlišení původní textury povrchu (kvůli vizuání stránce). Nicméně tvorba masky a její následné úpravy (póly a kraje) vyžadují více času, hlavně pak, když se jedná o masky s velikostí 2048x2048 pixelů.

Čili jsem opět zvolil cestu, kdy jsem činnost (čili algoritmus v metodě *generateSurfaceMask*) přesunul do jiného vlákna. Základní algoritmus přiřazení masky nyní funguje tak, že metoda *initializeVisuals* rozhodne o tvorbě masky. Masku následně vytvoří, nicméně ale ve čtvrtinovém rozlišení originálu. Dále vytvoří a spustí vlákno (*noiseThread*), kterému předá finální velikost masky a danou třídu *PerlinNoise* (s již přečteným seedem). Třída *Celestial*, obvykle již po inicializaci a v metodě *Update* (Unity metoda, která se volá každý snímek), kontroluje za běhu, zda již vlákno dokončilo tvorbu masky. Pokud se tomu tak stane, z vygenerovaných dat se vytvoří

nová proměnná datového typu *Texture2D* a ta se následně přiřadí materiálu. Bylo by rovněž vhodné zmínit, že po každé tvorbě a přiřazení masky je nutné z této masky vytvořit i inverzní masku. O tuto činnost se stará statická metoda *inverseMask* (třída *staticMethods*). Tvorba inverzní masky je již podstatně rychlejší (oproti samotné tvorbě masky pomocí metody *generateSurfaceMask*), nicméně i tento algoritmus je $\mathcal{O}(mn)$ a do budoucna by bylo vhodné ho rovněž zpracovat pro vlákno, aby nedocházelo ke zpomalení běhu programu.

Obrázek 3.12 reprezentuje výsledek procesu, popsaného v této kapitole. Jedná se konkrétně o planetu, která má teplotní třídu warm a obyvatelnostní třídu hypopsychroplanet (čili je umístěna v krajních částech obyvatelné zóny). Pomocí algoritmu, popsaného v této kapitole, planeta získala nejenom více detailů, ale i další typické rysy (zmrzlá krajina, moře).



Obrázek 3.12: Ukázka výsledku aplikování masky na těleso (vlevo) a náhled na konfiguraci materiálu (vpravo) daného tělesa

4 Závěr

4.1 Shrnutí

Ve všech třech bodech jsem při implementaci došel ke splnění zadání. Aktuální verze aplikace umožňuje interaktivní přístup k astronomickým datům o statisících planetárních systémech. Rovněž byla obohacena o zcela novou metodu simulace pohybu těles, která umožňuje reálnější pohled na určité situace, na které může uživatel při průzkumu prostoru, který aplikace vizualizuje, narazit. A do aplikace byly implementovány algoritmy, které umožňují tvorbu rozdílně vypadajích planet. Pocit z průzkumu prostoru bude o to silnější.

Za klíčový postup (z pohledu implementace výše zmíněných funkcí) považuji mé rozhodnutí o přesunutí výpočetně náročných operací do jiných vláken. Ačkoliv synchronizace se může z počátku jevit jako velký problém. Oddělení simulace a vizualizace (a jejich propojení skrz datovou frontu) je definitvně správný krok dopředu, který umožnil více času na vizualizaci a rovněž i více času na simulaci, která v aktuální verzi aplikace vlastně probíhá v kompletně jiné aplikaci (server), která může běžet i na rozdílném PC (odpadá tedy nutnost vlastnit specifický hardware pro výpočet nebo dostatečný výpočetní výkon). Na základě výsledků z kapitoly o řešení problému n těles jsem si rovněž potvrdil, že masivní paralelismus pomocí GPU opravdu funguje a umožňuje rapidní zrychlení řešení problém n těles. Čím rychleji budou daná data připravená, tím více jich můžeme zpracovat (bude více času na více klientů).

Samotný výsledek implementace problému n těles považuji za dobrý začátek. Stabilita simulace pro hvězdy a planety je dostatečná, nicméně po dobu implementace jsem měl problémy se simulací pohybu měsíců. Zatím jsem nenašel vhodné parametry pro dané lokální měřítko, které by zajistily optimální počáteční podmínky rychlosti měsíců. Zatímco některé měsíce se korektně pohybují okolo jejich planety. Existují zde i situace, kdy měsíce mají příliš vysokou rychlost a odpojí se od planety. Nebo má naopak malou rychlost a nastane kolize s planetou. Vysledoval jsem, že tento problém se týká hlavně těles s velmi malou hmotností. Nicméně hlavní části jsou již implementovány (samotný algoritmus komunikace a simulace) a tento problém je tedy otázkou dalšího ladění parametrů simulace. Rovněž počítám s analýzou a implementací systému, který by mi umožnil implementaci dalších vhodných numerických metod a jednoduchou změnu aktuálně používané numerické metody pro řešení tohoto problému. V simulaci problému n těles existuje spousta způsobů, kterými lze provádět konfiguraci a ladění výsledku.

4.2 Cíle do budoucna

Jednou z možností je optimalizace celkového běhu aplikace. Vzhledem k rozsáhlým změnám v kódu bude vhodné provést celkovou revizi a prověřit, co všechno se nemusí dělat každý snímek a případně i co by se dalo přesunout do jiných vláken. Mým cílem by mělo být co nejvíce ulevit hlavnímu vykreslovacímu vláknu a udělat změny mezi meřítkovými módy plynulé bez jakýkoliv chvilkových zaseknutí.

Přesun do Unity 5. Přesun jsem, vzhledem k možným problémům, které by mohly vzniknout přesunem hlavního projektu klientské aplikace, raději odložil na dobu neurčitou. Klientská aplikace tedy stále běží na Unity 4.6.1. Nicméně do budoucna plánuji přesun provést a doufám, že klientská aplikace bude schopna využít nových funkcí a vylepšení, které Unity 5 nabízí.

S plánovaným přesunem do Unity 5 bych rád vyzkoušel kompilaci klientské aplikace do WebGL. WebGL je v dnešní době velmi populární a výkonná platforma ve webových prohlížečích (není závislá na zásuvných modulech). Jestliže by byl tento přesun úspěšný, implementoval bych potřebné algoritmy, které by mi umožnily zmenšit velikost výsledné aplikace tak, že by se stahovala většina dat za běhu dle potřeby přímo ze serveru.

Plánované změny by pro server znamenaly, že by byl schopen zvládnout i více simulací naráz a tedy poskytovat simulační data více klientům najednou.

Další z věcí je implementace konstalací hvězd. V aktuální verzi se mi podařilo implementovat pouze menší počet vybraných konstalací. Mým cílem bylo hlavně otestovat, zda jsou pozice jednotlivých hvězd na obloze korektní. Nicméně úplná implementace se hlavně díky shromažďování jednotlivých hvězd z konstalací ukázala být časově náročná. Je třeba rovněž vymyslet vhodný algoritmus, který se bude starat o správu jednotlivých konstalací a jejich vykreslování. Vykreslování může být pak provedeno pomocí zobrazování jednotlivých hran mezi vertexy v objektech typu *StarCloud*.

Literatura

[1] Unity Technologies. Unity., *Development platform for creating multiplatform 3D and 2D games and interactive experiences.*, [online]. 2015-05-12 [cit. 2015-05-12]. Dostupné z: http://unity3d.com/unity

[2] Franců, A. Aurora., *Solar system simulation and visualization*, Technická univerzita v Liberci, 2012

[3] California Institute of Technology. Eyes on the Solar System., *3D environment full of real NASA mission data*, [online]. 2015-05-12 [cit. 2015-05-12]. Dostupné z: http://eyes.nasa.gov/

[4] Giant Army. Universe Sandbox., *Physics based space simulator.*, [online]. 2015-05-12 [cit. 2015-05-12]. Dostupné z: http://universesandbox.com/

[5] University of Puerto Rico at Arecibo. PHL., *The Planetary Habitability Laboratory*, [online]. 2015-04-11 [cit. 2015-04-11]. Dostupné z: http://phl.upr.edu/

[6] Nash, D. The HYG Database., *The Astronomy Nexus*, [online]. 2015-04-11 [cit. 2015-04-11]. Dostupné z: http://www.astronexus.com/hyg

[7] IEEE *Standard for Floating-Point Arithmetic*, IEEE Std 754-2008, vol., no., pp.1,70, 2008-08-29 [cit. 2015-05-12].

[8] Khronos Group. OpenCL., *First open, royalty-free standard for cross-platform, parallel programming of modern processors*, [online]. 2015-05-12 [cit. 2015-05-12]. Dostupné z: https://www.khronos.org/opencl/

[9] MongoDB, Inc. MongoDB., *NoSQL database*, [online]. 2015-05-12 [cit. 2015-05-12]. Dostupné z: http://www.mongodb.org/

[10] Franců, A. Realizace serveru a databáze pro aplikaci Aurora., Technická univerzita v Liberci, 2015

[11] Thorne, C., *Using a floating origin to improve fidelity and performance of large, distributed virtual worlds*, IEEE Computer Society Press, ISBN 0-7695-2378-1

[12] Thorne, C. Floating Origin., *Hi Fidelity Visualization and Simulation*, [online]. 2007-12-12 [cit. 2015-04-11]. Dostupné z: http://www.floatingorigin.com/

[13] Bednařík, Milan a Miroslava Široká. Fyzika pro gymnázia. 3. vyd. Praha: Prometheus, 2007, 288 s. ISBN 978-807-1961-765.

[14] NVIDIA. Nsight, *Powerful debugging and profiling tools*, [online]. 2015-05-12 [cit. 2015-05-12]. Dostupné z: http://www.nvidia.com/object/nsight.html

[15] Perlin, K.. Perlin Noise., *MAKING NOISE*, [online]. 1999-09-12 [cit. 2015-04-14]. Dostupné z: http://www.noisemachine.com/talk1/

[16] O'Neil, S. Gamasutra, *A Real-Time Procedural Universe, Part One: Generating Planetary Bodies*, [online]. 2001-03-02 [cit. 2015-04-14]. Dostupné z: http://www.gamasutra.com/view/feature/131507/a_realtime_procedural_universe_.php

[17] Scrawk. Procedural Noise., *Perlin noise plugin for Unity*, [online]. 2013-03-05 [cit. 2015-04-15]. Dostupné z: http://scrawkblog.com/2013/03/05/perlin-noise-pulgin-for-unity/

[18] Scrawk. Atmosphere., *GPU gems to Unity: Atmospheric scatter*, [online]. 2013-04-13 [cit. 2015-04-15]. Dostupné z: http://scrawkblog.com/2013/04/13/gpu-gems-to-unity-atmospheric-scattering/

[19] O'Neil, Sean. GPU Gems 2., *Accurate Atmospheric Scattering*, [online]. 2005-04-17 [cit. 2015-04-15]. Dostupné z: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter16.html

[20] Green, Simon. GPU Gems 2., *Implementing Improved Perlin Noise.* [online]. 2005-04-17 [cit. 2015-04-15]. Dostupné z: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter26.html

Přílohy

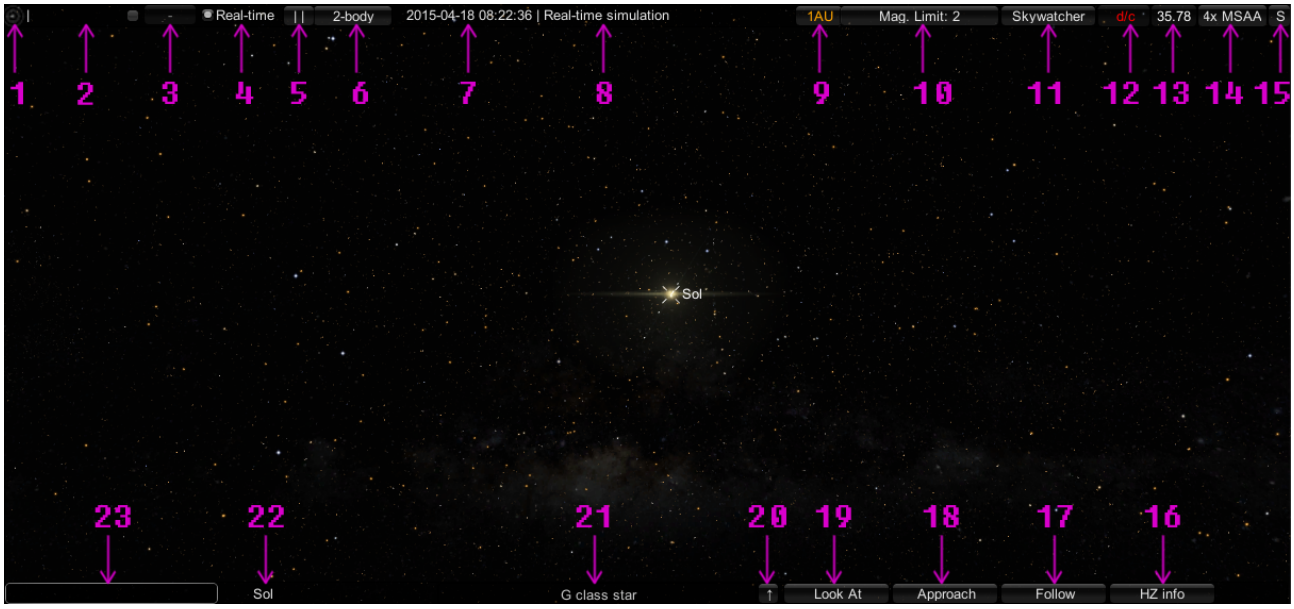
A Ovládání aplikace

Klávesnice a myš

- w/šipka nahoru - pohyb dopředu
- s/šipka dolu - pohyb dozadu
- a/šipka doleva - pohyb doleva
- d/šipka doprava - pohyb doprava
- levý/pravý shift - rychlejší pohyb
- page up - pohyb nahoru
- page down - pohyb dolů
- num 4 - rotace vlevo
- num 6 - rotace vpravo
- num 8 - rotace nahoru
- num 2 - rotace dolů
- kolečko myši - pohyb dopředu / dozadu
- levé tlačítko myši + posuv - pohyb
- pravé tlačítku myši + posuv - rotace
- q - náklon doleva
- e - náklon doprava
- x - natočení kamery na označený cíl (podržet)
- f - schování grafického uživatelského rozhraní

B Funkce uživatelského rozhraní

Na obrázku B.1 je ukázka grafického uživatelského rozhraní s číselnými referencemi na jednotlivé funkce, které popisuji v následujícím textu.



Obrázek B.1: Popis jednotlivých funkcí grafického uživatelského rozhraní

1. Ikona, která signalizuje aktuální měřítkový mód
2. Posuvník, kterým je možné měnit rychlost simulace
3. Změna rozlišení simulace (hodiny/dny)
4. Vypnout/zapnout simulaci v reálném čase
5. Start/stop simulace
6. Přepnutí simulační metody 2 těles / n těles
7. Aktuální čas
8. Dodatečné informace k simulaci
9. Změna měřítka (pouze v meziplanetárním módu)

10. Změna limitu hvězdné velikosti
11. Zapnutí módu skywatcher - sledování oblohy
12. Signalizace připojení k serveru/pingu
13. Aktuální FPS (snímky za sekundu)
14. Změna MSAA (Multisample anti-aliasing)
15. Přepnutí do módu celé obrazovky / okna
16. Zobrazení informací o obyvatelné zóně
17. Funkce kamery - následovat označený objekt
18. Funkce kamery - přiblížení k označenému objektu
19. Funkce kamery - rotace kamery k označenému objektu
20. Rozbalení/schování informačního okna
21. Rychlé info o označeném objektu
22. Název označeného objektu
23. Vyhledávání objektů v daném měřítkovém módu

C Spuštění, běh a příklady systémů

Aplikace serveru vyžaduje pro spuštění Visual C++ Redistributable 2012. Server je nutné spustit manuálně před zapnutím samotného klienta. Konfigurace IP adresy se nachází pro server v souboru config, který se nachází vedle exe souboru serveru - AuroraServer.exe. Tento soubor rovněž umožňuje konfiguraci zařízení pro výpočet problému n těles (volby jsou cpu či gpu). Konfigurace IP adresy serveru pro klienta se rovněž nachází v souboru config, tento soubor je umístěn ve složce Aurora_Data\saved\. Komunikace probíhá skrz port 8869 a defaultní IP je 127.0.0.1. Po každém vypnutí klienta je nutné restartovat server. Server prozatím zvládne pouze jednoho klienta pro simulaci problému n těles. Pro simulaci problému těles pomocí GPU je nutné, aby daná GPU uměla počítat s datovým typem double a podporovala OpenCL ve verzi alespoň 1.1.

V případě problémů s objevováním hvězd či extrémně malými snímky za sekundu v mezihvězdném měřítku lze použít tlačítka v levém horním rohu (zvané Manual refresh!), které by po pár kliknutích mělo algoritmus na tvorbu interaktivního okolí vrátit do normálního běhu (změna pozice kamery rovněž pomáhá).

Před přepnutím na simulaci problému n těles je nutné pamatovat na to, že aktuální počáteční podmínky (pozice, rychlost) určují následující výsledky simulace. V různých momementech přepnutí je tedy možné dosáhnout různých výsledků.

Existují systémy (přístupné skrz *Unknown cluster*), ve kterých je možno nalézt tělesa, která mají již zadefinovaný seed a tedy funkční povrch (kapitola Tvorba rozdílně vypadajících těles). Tyto systémy je nazývají **Bekywun** a **Inwu**. Rovněž zde existují systémy, které jsou zadefinované tak, aby zde schválně probíhaly kolize či zajímavé interakce mezi tělesy. Tyto systémy se nazývají **Ekom** (nestabilní systém) a **Auga** (vícehvězdný systém). Místo pro definici dalších systémů je ve složce Aurora_Data\saved\systems\custom\.