FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Dynamické generování a vizualizace planety
Student:	Jaroslav Kravec
Vedoucí:	Ing. Jiří Chludil
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Cílem práce je dynamické generování rozsáhlých světů (planet) s využitím pseudonáhodnosti.
1) Analyzujte způsoby dynamického generování světů, primárně se zaměřte na generování tvaru a typu terénu. (např. voda, louka, les, poušť)
2) Analyzujte problematiku vizualizace na zařízeních s vysokým rozlišením.
3) Navrhněte parametrizovatelný prototyp generátoru světů.
4) Navrhněte prototyp pro vizualizaci světů s podporou zařízení s vysokým rozlišením.
5) Implementujte oba prototypy s využitím OpenGL.
6) Podrobte oba prototypy vhodným testům (výkonnostní a zátěžové)

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 2. února 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalárska práca

Dynamické generování a vizualizace planety

Jaroslav Kravec

Katedra softwarového inženýrství
Vedúci práce: Ing. Jiří Chludil

15. mája 2018

Poďakovanie

Rád by som poďakoval svojmu vedúcemu Ing. Jiřímu Chludilovi za jeho čas, ochotu a pomoc pri vypracovaní tejto práce.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 15. mája 2018 …………………

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2018 Jaroslav Kravec. Všetky práva vyhradené.
Táto práca vznikla ako školské dielo na FIT ČVUT v Prahe. Práca je chránená medzinárodnými predpismi a zmluvami o autorskom práve a právach súvisiacich s autorským právom. Na jej využitie, s výnimkou bezplatných zákonných licencií, je nutný súhlas autora.

Odkaz na túto prácu

Kravec, Jaroslav. *Dynamické generování a vizualizace planety.* Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Práca sa zaoberá návrhom a tvorbou modulárneho systému na procedurálne generovanie detailných planét v reálnom čase a ich renderovaním na zariadeniach s vysokým rozlíšením. Vytvorený prototyp používa OpenGL na renderovanie aj generovania a v spojení s technológiou na distribuované renderovanie Eyescale Equalizer umožňuje súčasné použitie viacerých GPU a PC na renderovanie jednej scény. Najprv sú analyzované algoritmy procedurálneho generovania a spôsoby distribúcii renderovania. Následne je vytvorený návrh systému, z ktorého sa implementoval prototyp, ktorý sa podrobil výkonnostnými testami na video stene s vysokým rozlíšením.

Klíčová slova procedurálne generovanie, 3D vizualizácia, C++, OpenGL, paralelne renderovanie

Abstract

The bachelor's thesis deals with design and implementation of modular system for realtime procedular generation of detailed planets and rendering on devices with high resolution. Created prototype use OpenGL for rendering and generation and Eyescale Equalizer for distribion rendering. Equalizer enables applications to use multiple GPU and PC to render same scene. Created prototype is tested on video-wall with hight resolution.

Keywords procedural generation, 3D visualisation, C++, OpenGL, parallel rendering

Obsah

Odkaz na túto prácu vi

Úvod **1**
Ciele 1

1 Analýza **3**
1.1 Generovanie 3
1.1.1 Midpoint-displacement 3
1.1.2 Funkcie šumu 4
1.1.3 Porovnanie technik 5
1.2 Renderovanie a LOD 6
1.2.1 Distribuované renderovanie 9
1.3 Existujúce aplikácie 10
1.3.1 Proland[15] 10
1.3.2 Outerra 11
1.3.3 Terragen 11
1.3.4 World Machine 11
1.3.5 OpenSceneGraph 12

2 Návrh **13**
2.1 Požiadavky 13
2.2 Návrh riešenia založené na Prolande 13
2.2.1 Generovanie 14
2.2.2 Návrh na rozšírenia 15
2.2.3 Testovanie Prolandu 15
2.2.4 Dôvod nepoužitia Prolandu 16
2.3 Vlastné riešenie 16
2.3.1 Organizácia dláždic planéty a LOD 16
2.3.2 System generovania 20
2.3.3 Typy dat 26

2.3.4 Typy uzlov . . . 26
2.3.5 Sprava prostriedkov . . . 28
2.3.6 Funkcie na procedurálne generovanie (v GLSL) . . . 28
2.4 Paralelné renderovanie a generovanie . . . 28
2.4.1 Asynchrónna komunikácia . . . 29
2.4.2 Synchronizácia výstupu . . . 29
2.4.3 Vynechávanie snímkov . . . 30
2.4.4 Eyescale Equalizer . . . 30
2.4.5 Zdieľanie generovaných dát . . . 31
2.4.6 Sprava prostriedkov . . . 31

3 Implementácia **33**
3.1 Použite technológie . . . 33
3.2 Generator . . . 34
3.3 GLSL programy . . . 35
3.4 Paralelne renderovanie . . . 37

4 Testovanie **41**
4.1 Unit Testy . . . 41
4.2 Výkonové testy . . . 42
4.2.1 Spôsob testovania . . . 42
4.2.2 Výsledky . . . 42

5 Užívateľská príručka **45**
5.1 Kompilacia . . . 45
5.2 Parametre . . . 45
5.3 Lokalne spustenie . . . 46
5.4 Viacej zariadení . . . 46
5.5 Ovládanie . . . 47

Záver **49**

Bibliografia **51**

A Zoznam použitých skratiek **55**

B Obsah priloženého CD **57**

Zoznam obrázkov

1.1 Fraktálny šum 5

2.1 Základ návrhu 17
2.2 Systém uzlov generátora 18
2.3 Ukážková konfigurácia Eyescale Equalizera (obrázok z dokumentácie prístupnej cez [19]) 31

3.1 Ukážka celej planéty 38
3.2 Ukážka priblíženej časti planéty 39

4.1 Časy spracovania snímkov klientoch 43
4.2 Počty pridaných dlaždíc a zmeny pozície v snímkoch klienta 1 . . . 44
4.3 Prehľad časov spracovania snímkov klienta + ... 44

Zoznam tabuliek

4.1 Štatistika výkonového testovania 43

Úvod

Počítačovo simulované a vizualizované priestory sú už dlhú dobu bežnou záležitosťou. Používajú sa v počítačových hrách, pri tvorbe filmov alebo simuláciách. S rozvojom technológii sa zároveň zvyšujú požiadavky na kvalitu a rozsiahlosť scén. Platí to aj naopak, dopyt pre kvalitnejšiu grafiku v scénach poháňajú rozvoj technológii napred. Často je veľmi časovo náročné vytvárať cele scény ručne, preto sa používajú techniky procedurálneho generovania, ktoré pomáhajú tvorcom vytvárať scény alebo umožňujúcu generovať priamo cele svety. Jeden z ďalších dôvodov použitia procedurálnych generovaných scén je obísť limity obmedzenej pamäti a zároveň zabezpečiť rozmanitosť a rozsiahlosť sveta. S rozvojom technik pre virtuálnu a rozšírenú realitu sa rapidne zvyšujú požiadavky na kvalitu scén a rozsiahlosť scén. Dane technológie umožňujú ešte vierohodnejšie herne zážitky alebo simulácie, ako sú letecké, automobilové, či vojenské simulátory.

Ciele

Cieľom práce je návrh a implementácia prototypu systému na procedurálne generovanie planét a ich vizualizáciu v reálnom čase. Systém ma fungovať nie len na bežných počítačoch, ale zariadeniach ponúkajúce vyššiu kvalitu zobrazenia, ako sú VR okuliare, video stena (napr. SAGE2), či iné typy virtuálnej alebo zdielanej reality (CAVE). Tieto zariadenia už nemusia byť typické stolové počítačové zostavy, ale môžu obsahovať viacero grafických kariet, či byť zložené z viacerých počítačoch komunikujúcich prostredníctvom siete. Systém generovania má byť dostatočne flexibilný a modulárny, aby ho komunita programátorov mohla rozširovať o nove princípy generovania, či iné typy modulov, ale zároveň jednoduchý, aby sa dal rozšíriť o intuitívne grafické rozhranie, ktoré by ho umožnilo používať aj inym užívateľom. Z tohto dôvodu som sa rozhodol to vyvíjať ako open-source projekt, čo zároveň znamená stavať na open-source technológiách.

Kapitola 1

Analýza

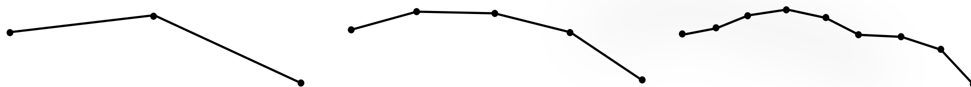
1.1 Generovanie

Tato sekcia obsahuje prehlad algoritmov a princípov, ktoré sa používajú na generovanie rozsiahlych pripadne nekonečných svetov. To sú algoritmy, ktoré umožňujú vygenerovať požadovanú časť sveta, bez toho aby bolo potrebne vygenerovať okolie alebo celý svet. Zameriavajú sa hlavne na generovanie terénu (výškovej mapy), no podobne princípy je možné použiť aj na ostatne vlastnosti, ako je typ či farba povrchu.

1.1.1 Midpoint-displacement

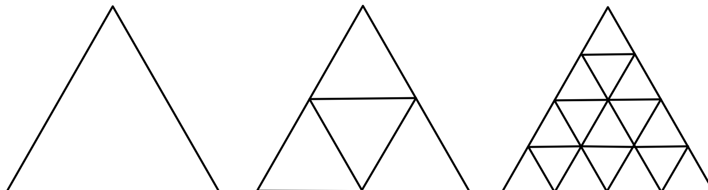
Midpoint-displacement je rekurzívny algoritmus použiteľný na generovanie terénu v podobe výškovej mapy. Algoritmus ma viacej implementácii podľa použitého útvaru na akom pracuje: čiara, trojuholník alebo mriežka. Najprv sa vytvoria krajne body útvaru. V každej iterácii sa pridajú nové body, ktoré sa vypočítajú ako priemer hodnôt susedných bodov (midpoint), ku ktorým sa pripočíta odchýlka (displacement) v závislosti na hlbke rekurzii. Dana odchýlka môže byť úplne náhodná alebo získaná pomocou funkcii šumov, či iného zdroja (napr. textúry).

Najjednoduchší útvar je úsečka – výsledkom je pole hodnôt. Najprv sa vytvoria 2 krajne body a každá iterácia vytvori pre každé 2 susedné body novy bod z ich priemeru s pripočítaním vertikálnej odchýlky. Tato metóda je použiteľná na generovanie 2D terénu. Znázornenie postupu:



Ďalším útvarom je trojuholník. Najprv sa vytvori rovnostranný trojuholník na rovine v 3D priestore. Nové body sa vytvárajú ako priemer z každých

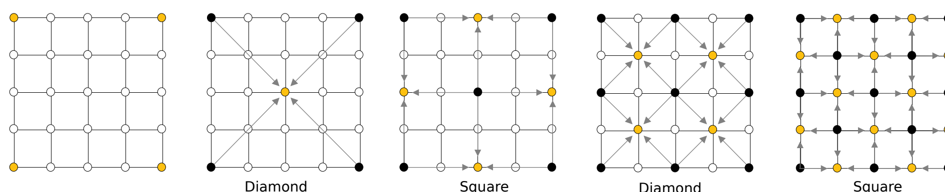
2 susedných bodov a pripočítava sa odchýlka do výšky (resp. hlbky). Týmto spôsobom je možné generovať 3D terén. Znázornenie postupu:



Diamond-square[1] je implementácia pracujúca na 2D mriežke, čo je výhodou oproti trojuholníku, keďže to umožňuje jednoduchšie ukladanie a spracovávanie dát (hlavne na GPU). Iterácia pozostáva z z 2 krokov:

- krok „diamond“ – vypočíta prostredné hodnoty zo 4 susedných hodnôt.
- krok „square“ – doplnenie chýbajúcich hodnôtzo 4 susedných hodnôt.

Znázornenie priebehu 2 iterácii (Zdroj obrazku [2]):

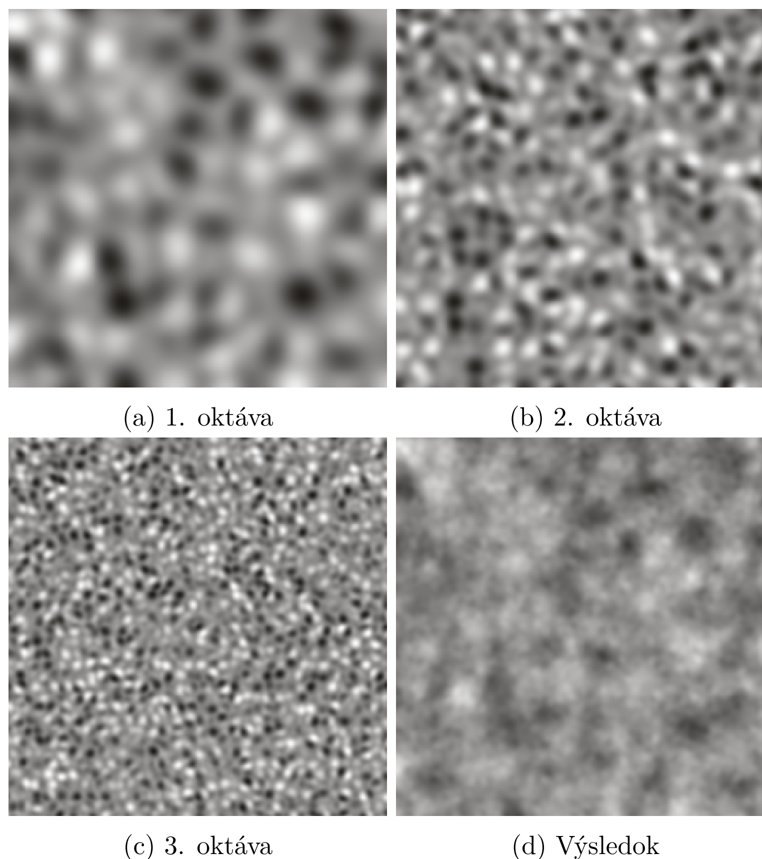


1.1.2 Funkcie šumu

V počítačovej grafike funkcie šumu (coherent noise) znamenajú pseudo-náhodne spojité funkcie, ktoré pre n-árny vstup desatinných čísiel vrátia hodnotu v určitom rozsahu (často 0 až 1 alebo -1 až 1).

Jednoduchá varianta pozostáva z vytvorenia n-dimenzionálnej mriežky so pseudonáhodnými hodnotami. Pre n-argument sa najprv vyhľadá bunku v mriežke, do ktorej daná hodnota spadá a vráti interpolovanú hodnotu z rohových bodov bunky. Pokročilejšie funkcie používajú mriežky s pseudonáhodnými gradientami určené vektormi. Prvou takou implementáciou bol Perlin Noise, ktorú predstavil Ken Perlin [3] Následne vytvoril Simplex Noise[4], ktorý poskytuje menej viditeľných chýb a je výkonovo efektívnejši, ale je pokrytý patentom. Ako alternatíva ku Simplex Noise vznikol OpenSimplex Noise[5].

Pri generovaní sa používa kombinácia viacero funkcii. Pomocou metódy Fractional Brownian Motion je možné vytvoriť fraktálny šum, ktorý poskytuje väčšiu rôznorodosť a vyššie detaily. Ide o postupné sčítanie oktáv zvolenej funkcii šumu, kde so zvyšovaním frekvencii sa znižuje ich amplitúda[6]. Na obrázku 1.1 je ukážka s použitím 3 oktáv Perlin Noise. Funkcie sa tiež môžu navzájom ovplyvňovať napr. funkcia generujúca klimatické podmienky (tep-



(a) 1. oktáva (b) 2. oktáva

(c) 3. oktáva (d) Výsledok

Obrázok 1.1: Fraktalny sum s pouzitim Perlin Noise[1]

lota, vlhkosť) ovplyvni parametre funkciám na generovanie výšky terénu a typu povrchu (pohoria, puste roviny).

1.1.3 Porovnanie technik

Tato sekcia sa zameriava na porovnanie techník generovania terénu hlavne z implementačného hľadiska. Techniky popísane vyššie za dajú rozdeliť na rekurzívne a priame. Rekurzívne techniky sú tie, ktoré fungujú na baze skvalitňovania oblasti (z oblasti s nižšou úrovňou LOD sa vytvori oblasť vo vyššej úrovni LOD), čiže pre určitú LOD musíme najprv vygenerovať dané oblasti z nižšej úrovni LOD. Midpoint-displacemenet je príklad tejto techniky. Pod pojmom priame techniky sa označujú tie, z ktorých je možné vygenerovať oblasť v hocakej úrovni LOD, bez nutnosti mať vygenerované rodičovské oblasti, dá sa povedať, že funguje to na báze vzorkovania – z pozícii/súradníc sa získajú údaje o bode (výška, farba, normál...). Príkladom je použitie funkcii šumov. V skutočnosti obe techniky môžu byť zameniteľné:

[1] generovane cez: http://cpetry.github.io/TextureGenerator-Online

- midpoind-displacement je možné implementovať ako priamu techniku, keďže jedná iterácia rekurzii sa sprava ako jednoduchá funkcia šumu, čo sa dá skombinovať s Fractional Brownian Motion
- funkcie šumu je možné použiť aj v rekurzii, napr. ako odchýlka v midpoint-displacement

1.1.3.1 V prospech priameho prístupu

Výhodou priameho prístupu je jednoduchšia implementácia (hlavne s použitím GPU), keďže nie je nutné riešiť závislosti medzi jednotlivými úrovňami a dá sa jednoduchšie konfigurovať - nerieši sa čo na akej úrovni sa ma diať, ale iba z globálneho hľadiska. Výhodou priameho spôsobu, je väčšia nezávislosť na implementácii/vyberú LOD techniky. Pri rekurzívnom je LOD zavisla na typu použitého útvaru a spôsobu delenia. Nevýhodou rekurzie je, že schopnosti generátora sú obmedzene na typ útvaru a spôsob delenia (napr. trojuholník alebo diamond-square), čo môže viesť k tvorbe artefaktov (napr. terén je viditeľne pravidelne štvorcový alebo trojuholníkoví v závislosti). To je možné čiastočne zmierniť vhodným použitím funkcii šumov.

1.1.3.2 V prospech rekurzívneho prístupu

Nevýhodou priamej techniky je, že na vygenerovanie rovnakých dát môže by potreba viac výpočtového výkonu. To čo pri rekurzívnom prístupe je možne získať (a následne rozšíriť) z nadradenej úrovni rekurzie, je pri priamej potrebné vypočítať úplne nanovo. Výhodou rekurzie je možnosť kompletne zmeniť pozíciu generovaného bodu - generovania zložitejších štruktúr. Keďže priama technika funguje na baze vzorkovania, je možne upraviť len jednú súradnicu a teda je veľmi náročne generovania útvarov, ktoré sa nedajú zaznamenať v podobe výškovej mapy (napr. skalné previsy). Výhodou rekurzie sú možnosti optimalizácie. Keďže potomkovia závisia na rodičovi, je možne z rodiča zistiť, ktoré cesty výpočtu nie sú potrebne, a tie preskočiť. Tým je možné dosiahnuť generovanie veľmi rôznorodých oblasti (z globálneho hľadiska) efektívnejšie. Napr. v strede púšte alebo na Sibíri nie je potrebné generovať vysoké pohoria pripadne rieky alebo naopak, na pohoriach sa nepredpokladá pusť. V prípade žeby to bolo na hrane (z vysokých pohorí sa prechádza priamo do púšte), optimalizácia sa neaplikuje.

1.2 Renderovanie a LOD

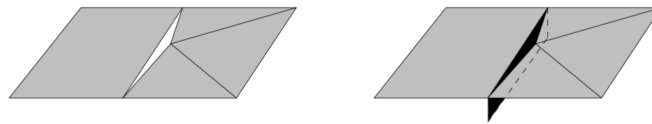
Rozsiahly terén je veľmi neefektívne renderovať ako celok, ak je viditeľná iba mala časť povrchu. Z tohto dôvodu sa často delí na menšie časti, ktoré sú načítavané (či generované) a renderované nezávisle na ostatných. V prípade presunutia pohľadu sa postupe dočítavajú ďalšie oblasti a uvoľňujú nepotrebne. Tento systém dobre spolupracuje s technikami LOD, keďže je možné

určovať kvalitu jednotlivých oblasti na základe vzdialenosti od pozorovateľa alebo inej metriky.

Väčšina existujúcich LOD technik pre terén pracujú s výškovou mapou a používajú stromovú štruktúru na organizovanie dát (quadtree - Seamless Patches[7], CDLOD[8], Chunked LOD[9], alebo bintree – ROAM, BDAM). Funguje to tak, že jednotlive bunky stromu sa delia dovtedy, kým nie je postačujúca kvalita oblasti vzhľadom na pozorovateľa – čím väčšia hlbka v strome, tým je oblasť rozmerovo menšia ale kvalitnejšia. Pre každú bunku sa načíta geometria a potrebné textúry. Geometria sa vytvára z výškovej mapy (textúry), ktorá sa priamo pri renderovaní namapuje na mriežku (vo vertex shader) – hodnota v textúre na pozícii (súradnica X, Y) určuje výsku bodu (súradnica Z).

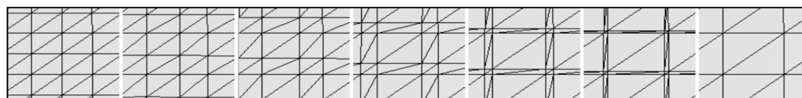
Keďže vedľa seba sa môžu nachádzať oblasti s odlišnou úrovňou LOD, môžu vznikať vizuálne chyby ("praskliny" - cracks) na hranách medzi tými oblasťami. Na to existuje viacero riešení:

- "skirts" (Chunked LOD[9]) – pridaním zvislých plôch na hranách oblasti. Postačujúca metóda, ak nie sú vertikálne rozdieli medzi oblasťami veľmi veľké. Tento princíp dobre LOD funguje iba na geometriách založených na výškovej mape:



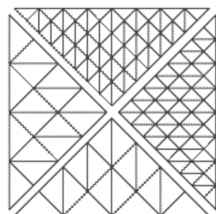
(Zdroj obrázku [8])

- horizontálny morphing (CDLOD[8]) – každý 2. bod sa postupne približuje k susedovi až ho spoji, čim sa zabezpečí, že oblasť z vyššej úrovni sa prispôsobí oblasti z nižšej. Nevýhoda je, že bod geometrie sa nemapuje priamo na bod vo výškovej mape (v textúre), čo môže viesť k vizuálnym chybám:



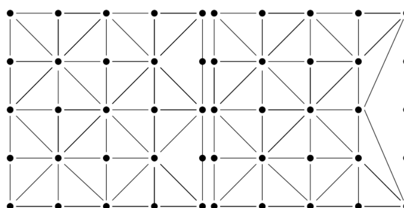
(Zdroj obrázku [8])

- Oblasť pozostáva z 4 trojuholníkov spojenými v strede v tvare X (Seamless Patches[7]). Podobne ako pri predošlej metóda sa geometria nemapuje priamo na textúru a vyžaduje zložitejší manažment geometrie oblasti:



(Zdroj obrázku [7])

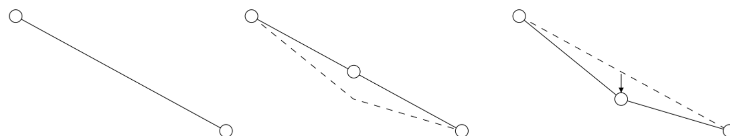
- Prispôsobenie hrán na oblasti s väčšou LOD na oblasť s nižšou LOD (Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail[10]). Vyžaduje úpravu geometrie:



(Zdroj obrázku [10])

Zmeny geometrie alebo výškových textúr oblasti medzi úrovňami LOD môže byť viditeľne, čim vznika rušivý jav (popping). Na to je niekoľko technik ako tomu predísť:

- blending – vykresľujú sa obe LOD po krátku domu, s tým ze sa pomocou alpha blending postupne prechádza medzi nimi. Tento spôsob sa používa najme pri diskrétnom LOD.
- geomorphing (Chunked LOD[9])– plynulý prechod medzi LOD na úrovni geometrie. Pri zmene do vyššej LOD sú nové body (tie ktoré v nižšej LOD neboli) umiestnene najprv na pozícii, aby vizuálne zodpovedalo pôvodnej LOD. Tieto body sú postupne presúvané k ich originálnej (novej) pozícií vrámci krátkej dobe alebo na základe vzdialenosti k pozorovateľovi:

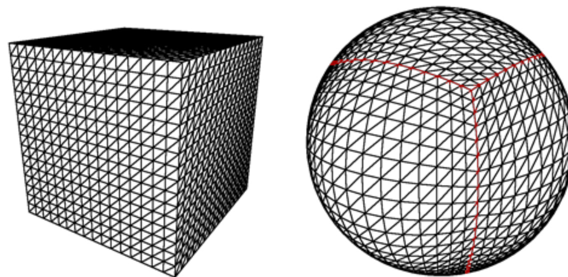


Za zmienku stoji ROAM technika, ktorá používa bintree a sa postupne buduje pridávaním a odoberaním trojuholníkov medzi jednotlivými snímkami (jeden uzol v bintree – jeden trojuholník). Nevýhodou je, že cely algoritmus pracuje na CPU, čo v súčasnosti nestačí na veľké množstvo trojuholníkov a nevyužije sa optimálne GPU. Tento nedostatok rieši BDAM[11], ktorý pracuje

nad skupinou trojuholníkov (patch) a pridáva podporu textúrovania (organizovanej v podobe quadtree). Ďalšou technikou je Geometry Clipmaps[12] (Terrain Rendering Using GPU-Based Geometry Clipmaps), ide o spôsob, kde je okolo centra pozorovateľa štvorec s najvyššou LOD a následne obdĺžnikové prstence pre nižšie LOD. Medzi jednotlivými LOD sa prechádza pomocou blendingu.

Z týchto technik vznikli niektoré prispôsobené na planéty:

- P-BDAM[13] – odvodene od BDAM, rieši aj problémy s presnosťou desatinných čísel pri veľkých vzdialenostiach
- Terrain Rendering using Spherical Clipmaps[14] – nadväzuje na Geometry Clipmaps[12], používa iný typ mapovania textur (Sperical mapping) vhodnejsi pre planéty, ale s nerovnomernou kvalitou textúr v rôznych oblastiach planéty
- Mapovanie kocky na guľu (Proland[15]) – ide o princíp, kde sa vytvori 6 plochých terénov, v tvare kocky, ktoré sú následne deformovane do gule. Podobne ako pri Sperical Clipmaps[14] aj tu nie je texturovanie uniformne rozdelene, tie sú menej pozorovatelne:



1.2.1 Distribuované renderovanie

Zariadenia s vysokým rozlisenim ako sú video stena (SAGE2[**SAGE2**]) uz nemusia byť typické jednopocitacove zostavy, ale môžu obsahovať viacero grafických kariet, či byť zlozene z viacerých pocitacov komunikujúcich prostredníctvom siete. Distribuované renderovanie je spôsob ako využiť viacero GPU na dosiahnutie väčšieho výkonu alebo kvality pri renderovani jednej scény.

Existuje viacero metód ako riešiť distribúciu [16]:

- Frame distribution – každé GPU renderuje cely snímok v inom čase. Poskytuje dobre škálovanie výkonu, ale žiadne škálovanie dát –- musia byť načítane súčasne na všetkých GPU. Nevýhoda je tiež zvýšenie latencie, medzi začiatkom renderovania a zobrazenia snímku.
- Pixel distribution (first sort) – distribúcia skupiny pixlov jednej snímky. Distribúcia prekladaním čiar (napr. parne a nepárne stĺpce) poskytuje dobre škálovanie výkonu, ale znemožňuje škálovanie dat. Distribucia

celých 2D plôch (napr. rozdelením na ľavú a pravú stranu na 2 GPU) poskytuje škálovanie dát, no môže vzniknúť väčší nepomer medzi komplexivitou scény na jednotlivých plochách, čo zhoršuje balancovanie výkonu.

- Objekt distribution (last sort) – rozdelením objektov (časti scény), Každe GPU renderuje na celu plochu snímku svoju pridelenú časť scény, výsledný snímok sa dosiahne spojením (kompozíciou) výstupou na základe hlbky. Výhodou je teda dobre škálovanie dat.
- Hybridne – kombinácia viacerých metód

Z programátorského hľadiska riešenia môžu byť transparentne, príkladom je ClusterGL, ktorý nahrádza OpenGL API [17]. Týmto spôsobom je možné spustiť existujúce aplikácie neprispôsobenie na distribuované renderovanie. Nevýhodou je náročné škálovanie výkonu. Ďalším spôsobom je použiť grafickú knižnicu (engine/framework), ktorá podporuje distribuované renderovanie (OpenSG[18], Eyescale Equalizer[19]).

OpenSG[18] je graficky engine s grafom scény, ktorý je prispôsobený na multivláknove a distribuované spracovanie a renderovanie scény. Aplikácia musí používať ich graf scény, čiže nie je teda vhodný na portovanie existujúcej aplikácie. V dobe písania textu vyzerá byť neaktívny, oficiálna stránka už nie je k dispozícii, tým pádom je ťažko sa zorientovať v knižnici, keďže sa nedá nájsť dokumentácia.

Pri použití Eyescale Equalizer[19] knižnice sa aplikácia skladá z jadra aplikácie a klientov, ktorý sú určený na renderovanie. Obe tieto časti je potrebne naprogramovať. Jadro aplikácie sa spusti raz a pomocou Equalizer servera komunikuje s klientami. Equalizer je postavený na knižnici Eyescale Collage[20], ktorá je určená na tvorbu heterogénnych distribuovaných aplikácii. Tato knižnica zabezpečuje komunikáciu, synchronizáciu a zdieľanie dát s podporou verziovania. Equalizer umožňuje použiť všetky metódy riešenia distribúcie (vypísané vyššie) a podporuje automatické balancovanie na základe zátaže. [19]

1.3 Existujúce aplikácie

V tejto sekcii sú popísane riešenia, ktoré sa zameriavajú na generovanie a renderovanie terénov, pripadne celých planét. Ďalej sú popísane open source C++ frameworky, ktoré môžu byť použité ako základ pri tvorbe tohto projektu.

1.3.1 Proland[15]

Crossplatform (Windows, Linux) open source projekt (BSD3) vytvorený na akademickej pôde, zameraný na generovanie a renderovanie celej planéty s podporou LOD primárne na základe podkladov zeme. Projekt je postavený

na Ork[21] frameworku – graficky engine, vytvorený tými istými tvorcami. Ork a tým pádom aj Proland sú napisané v C++ s použitím OpenGL minimálne 3.3 a (4.X na poskytnutie všetkých funkcii). Ork podporuje manažment a plánovanie úloh pre CPU aj GPU. Generovanie v Prolande prebieha plne na GPU. Podporuje generovanie výškovej mapy, textúr, inštancie objektov (stromov) a vektorové dáta v podobe ciest, či riek. Podporuje renderovanie oceánu, atmosféry a oblakov, či samotného terénu. Je to výskumný prototyp, ktorý bol vyvíjaný v rokoch od 2008-2015 a testovaný iba na niekoľkých NVIDIA GPU. Nevýhodou je že ide iba o prototyp, ktorý už nie je aktívne vyvíjaný, čiže môžu sa naskytnúť problémy s kompatibilitou na odlišnom HW. Výhodou je open source licencia, čo umožnuje to to ďalej rozširovať a opravovať.

1.3.2 Outerra

Herný a simulačný engine pre Windows postavený na OpenGL 3.3+. Podporuje rengerovanie a generovanie celej planéty s podporou LOD a dobrou úrovňou kvality. Je to komerčný (platený) closed-source projekt, ale poskytuje demo s názvom Anteworld. Generovanie prebieha plne na GPU. [22]

1.3.3 Terragen

Platený nastroj na vytváranie a renderovanie realistických prírodných prostredí od veľkosti planéty do úrovni niekoľkých centimetrov. Poskytuje obmedzenú voľnú verziu pre nekomerčné použitie. Umožňuje vytvárať terén, atmosféru (oblaky), rozmiestňovanie objektov (population, napr. stromy). Generovanie je primárne založené na procedurálnych (fraktálnych) algoritmov, ale podporuje aj načítanie výškových máp a modelov. Nie je viazaný iba na generovanie výšky terénu, čiže umožňuje vytvárať aj zložitejšie útvary. Konfigurácia generovania je založená na grafu uzlov, kde uzly predstavuje určitú transformáciu alebo tvorbu nových dat. Každý typ uzlu ma definované vstupy a výstupy, čim je možné uzly spájať a vytvárať pokročilú funkcionalitu. Používa hybridný mikropolygonový a ray-tracing renderovaci systém, ktorý nie je realtime, ale poskytuje vierohodne výsledky.[23]

1.3.4 World Machine

Projekt zameraný na generovanie terénu v podobe výškovej mapy a farebnej textúry určený pre export. Systém generovania používa graf uzlov, podporuje vektorové dáta a umožňuje množstvo operácii robiť interaktívne priamo na teréne. Obsahuje rýchly náhľad s možnosťou zobrazenia nekonečného terénu. Je to komerčný projekt, dostupný zadarmo pre nekomerčné použitie s obmedzením maximálneho rozlíšenia pre export.[24]

1.3.5 OpenSceneGraph

OpenSceneGraph obsahuje modul osgEarth, ktorý je určený na renderovanie planét. Ma dobru podporu na získavanie dát z externých zdrojov (napr. z internetu pomocou GDAL). Obsabuje modulárny systém na získavanie a renderovanie dát, čo umožňuje ho rozšíriť o procedurálne generovanie. Ďalším modulom je osgScaleViewer, čo je príklad na prepojenie OpenSceneGraph s Equalizerom, čiže umožnenie distribuovaného renderovania. [25]

KAPITOLA 2

Návrh

Návrh počíta s generátorom založeným na grafe uzlov, čo je inšpirované Terragen a World Machine. Oproti týmto riešeniam je prispôsobený na realtime renderovanie a generovanie s vyžitím viacerých GPU.

2.1 Požiadavky

Funkčne požiadavky:

- generovanie celej planéty s minimálnou presnosťou na metre
- realtime generovanie - efektívne natoľko aby sa nepozorovateľne generovali nove oblasti oblasti pri pohybe po scéne
- realtime seamless renderovanie s podporou LOD - plynulý prechod medzi rôznymi LOD
- podpora zariadení s viacerými GPU resp. pozostávajúcich z viacerých PC (paralelné renderovanie)
- vysoká modularita generátora

Nefunkčné požiadavky:

- C++11 a viac
- OpenGL 3.0 a viac
- open-source
- cross-platform (min. Windows, Linux)

2.2 Návrh riešenia založené na Prolande

V tejto kapitole najprv popíšem jednoduchý technicky prehlad ako funguje Ork a Proland, následne návrhy na rozšírenia a ku konci dôvod prečo v tom nepokračujem.

Proland resp. Ork poskytuje veľmi flexibilný a univerzálny systém na pracú nad grafom. Prostredníctvom XML súborov sa konfiguruje resp. "programuje" postupnosť krokov, čo sa ma pri každom snímku spraviť. Každý uzol je univerzálny a môže obsahovať:

- príznaky – slúži ako filter pri prechádzaní scény grafu
- metódy – môže byť sekvencie krokov alebo natívna metóda
- mesh – geometria na renderovanie
- moduly – GLSL shader alebo iba jeho časť
- iné vlastnosti

Každý z týchto prvkov ma svoje meno, na ktoré je možné sa v metódach (sekvenciach) odkazovať a tým ich predávať do iných metód (napr. drawMesh vyžaduje odkaz na mesh, ktorý je možné získať z uzlu). Tento systém sa používa jak na renderovanie nodov tak ich aktualizáciu (napr. aktualizácia terénu alebo kamery)

2.2.1 Generovanie

Proland reprezentuje terén v podobe quad-tree, ktorý je dynamicky delený na základe pozície kamery. Na dosiahnutie generovania planéty Proland poskyje globálne deformácie terénu, konkrétne planétu je možne vytvoriť z 6 terénov pre každú hranu kocky, ktoré sa následne deformujú do podoby gule.

Základom samotného generovania je TileProducer, čo je abstraktná trieda na generovanie dát pre jednotlive dlaždice (tile) v quad tree. Proland obsahuje implementovaných niekoľko producerov, ktoré umožňujú načítavať predpočítané dáta z disku, generovať dáta pomocou funkcii šumov, alebo kombinácie oboch. Výsledkom produceru je Tile, ktorú je možné použiť v inom procudery alebo namapovať na GLSL program prostredníctvom TileSampler.

Samotne dáta sú ukladane v TileStorage (GPUTileStorage, CPUTileStorage, ObjectTileStorage). Je to štruktúra umožňujúca uložiť dáta rovnakého typu s konštantnou kapacitou slotov, ktorá sa nedá zmeniť za behu. Spojenie medzi producermi a TileStorage zabezpečuje TileCache - mapuje medzi logickými koordinátami dláždic vo svete a slotmi v TileStorage. Umožňuje "zamknúť" dláždice, ktoré sa používajú pri renderovani alebo v iných produceroch. Každý TileProducer ma pridelený TileCache (správneho typu, podľa toho aký vyžaduje). Keď sa producer vyžiada o získanie určitej dlaždice pomocou getTile() najprv sa vyhľadá vo vyrovnávacej pamäti až keď nie je nájdený vráti sa Task, pomocou ktorého je možné vygenerovať danú dláždicu s pouzitim Ork task manažéra.

2.2.2 Návrh na rozšírenia

Návrhom bolo použiť Ork resp. Proland ako základ, na ktorom by sa stavali ďalšie funkcionality:

- rozšíriť o nové TileProducery zamerane čisto na procedurálne generovanie
- pridať podporu jednoduchécho paralelného renderovania. Jednalo by sa o architektúru master (hlavná aplikácia s uživateľskym vstupom) + klienti (renderovacie a generovacie jednotky):
  - použila by sa pixel distribúcia jeden klient renderuje jednu časť celkovej obrazovky a každý klient by si generoval dáta nezávisle na ostatných a to iba tie ktoré sú nutne k vyrenderovaniu jeho časti obrazovky
  - graf scény by bol rovnaký na mastrovy aj klientoch, s tým že master by mal iba základnú štruktúru bez načítaných dát (ako sú moduly, modely, textúry..) o aktualizáciu grafu by sa staral master a dáta by sa následne museli poslať klientom (zosynchronizovať)
  - pridala by sa metóda na externe volanie iných metód na klientoch – master vyžiada zavolanie metód na klienoch. Tie by mohli fungovať synchrónne (počká sa na dokončenie) alebo asynchronne
- rozšíriť ResourceManager o získavanie dát prostredníctvom siete. Jednou možnosťou je priamo rozšíriť Ork alebo alternatívnou variantou je použiť existujúce riešenia na zdieľanie priečinkov cez sieť (napr. SMB na Windows, alebo curlftpfs, sshfs na namapovanie (S)FTP na lokálny priečinok)

2.2.3 Testovanie Prolandu

Ork aj Proland bol naposedy oficiálne aktualizovaný v roku 2012 (v roku 2015 sa zmenila licencia na BSD3), bol písaný v CodeBlocku a testovaný iba na NVIDIA GPU (napr. NVIDIA GeForce 470 GTX, NVidia GeForce 260M GTX). Otvorenie tohto projektu mi v súčasnom CodeBlock robilo problémy, takže som začal vyvíjať od jeho klonu na githube (LarsFlaeten/Proland_dev), ktorý ho portol do CMake a zároveň postupne opravil už nejaké chyby s kompatibilitou na súčasných GPU.

Najprv som skúšal samotný Ork framework (taktiež od LarsFlaeten) bez Proland nadstavby. Ork obsahuje množstvo unit testov, ktoré z počiatku neprechádzali. Postupne som zistil, že boli samotne testy zle napísané aj keď funkcionalita bola väčšinou správna (až na drobné chyby, konkrétne pri AMD RX 550 v Program::initUniforms)

Výpis zvyšných unit testov, ktoré neprešli na GTX 1080 a RX 550:

```
Zostali tieto testy ktoré nepresli:
cpuMeshModificationDirect...                            [FAILED]
cpuMeshModificationIndices...                           [FAILED]
testProgramPipelineAutomaticSamplerBinding...           [FAILED]
automaticSamplerBinding...                              [FAILED]
```

CpuMeshModification a CPUMesh implementácia nie je v súlade s sucastnym OpenGL štandardom, takže to nemôže ani prechádzať a v rámci Ork a Proland sa tento pristúp nepoužíva (alebo bol opravený od LarsFlaeten). Zostava automaticSampler, ktorý robí pravdepodobne hlavne problémy, ktoré sa vyskytujú aj v Prolande.

2.2.4 Dôvod nepoužitia Prolandu

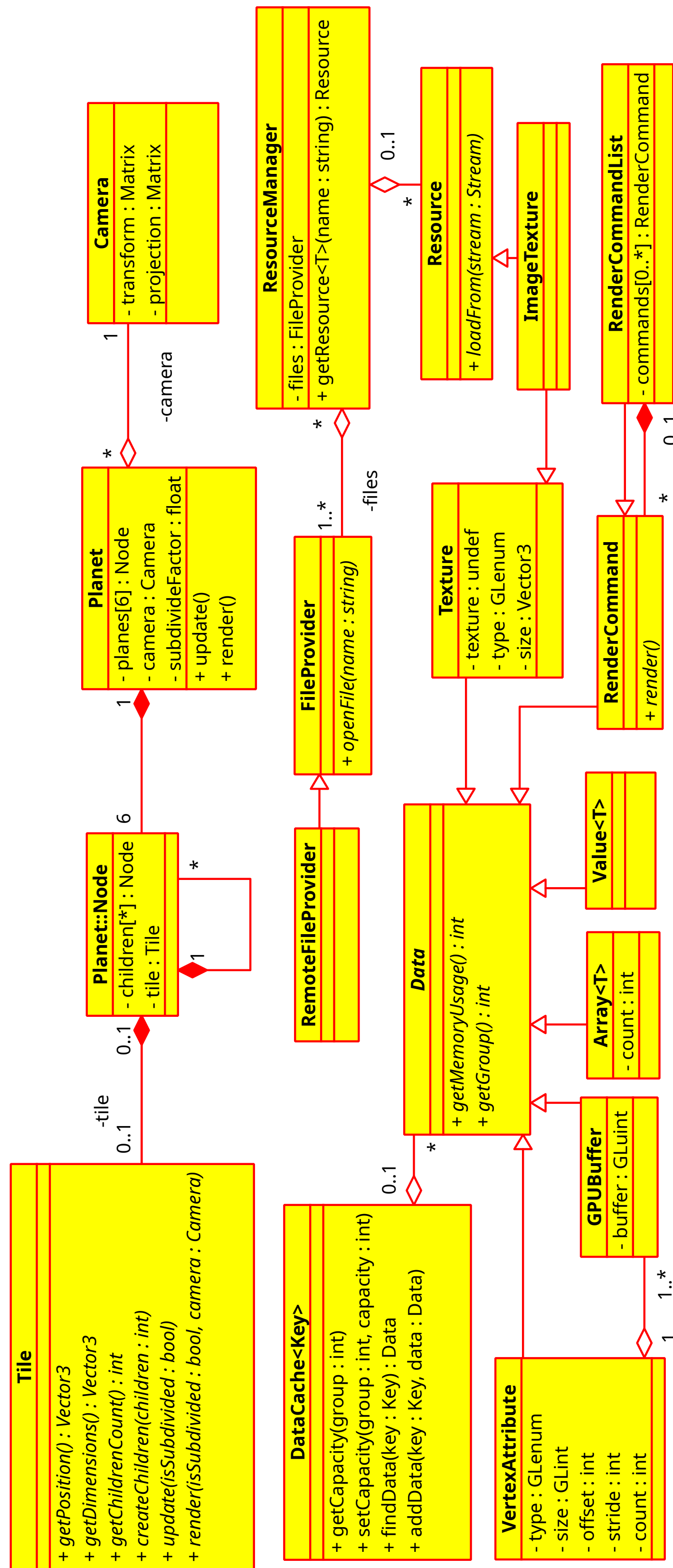
Pri testovaní Prolandu som narazil na očakávané problémy s kompatibilitou. Niektoré ukážky scén dodávané s Prolandom nešlo spustiť, pri iných sa nesprávne generovala alebo renderovala planéta. Začal som so študovaním kódu, identifikáciami chýb a ich odstraňovaním. Niektoré drobné problémy sa mi podarilo opraviť alebo aspoň zlepšiť. Dane problémy sa ukázali väčšie ako som predpokladal a v danom čase vzhľadom na rozsiahlosť projektu som nemal dostatok informácii na to aby som určil koľko mi zaberie riešenie zvyšných problémov a či sa mi to vôbec podari dať do stabilnej podoby, ktorá by bolo vhodná na rozširovanie. Rozhodol som sa teda, že nebudem v tom pokračovať, čiže nepoužijem Proland ako základ prototypu. Dôvodom je teda, že som pri analýze nesprávne odhadol závažnosť problémov s kompatibilitou a predpokladal som, že dane problémy budem schopný vyriešiť v kratšom čase.

2.3 Vlastné riešenie

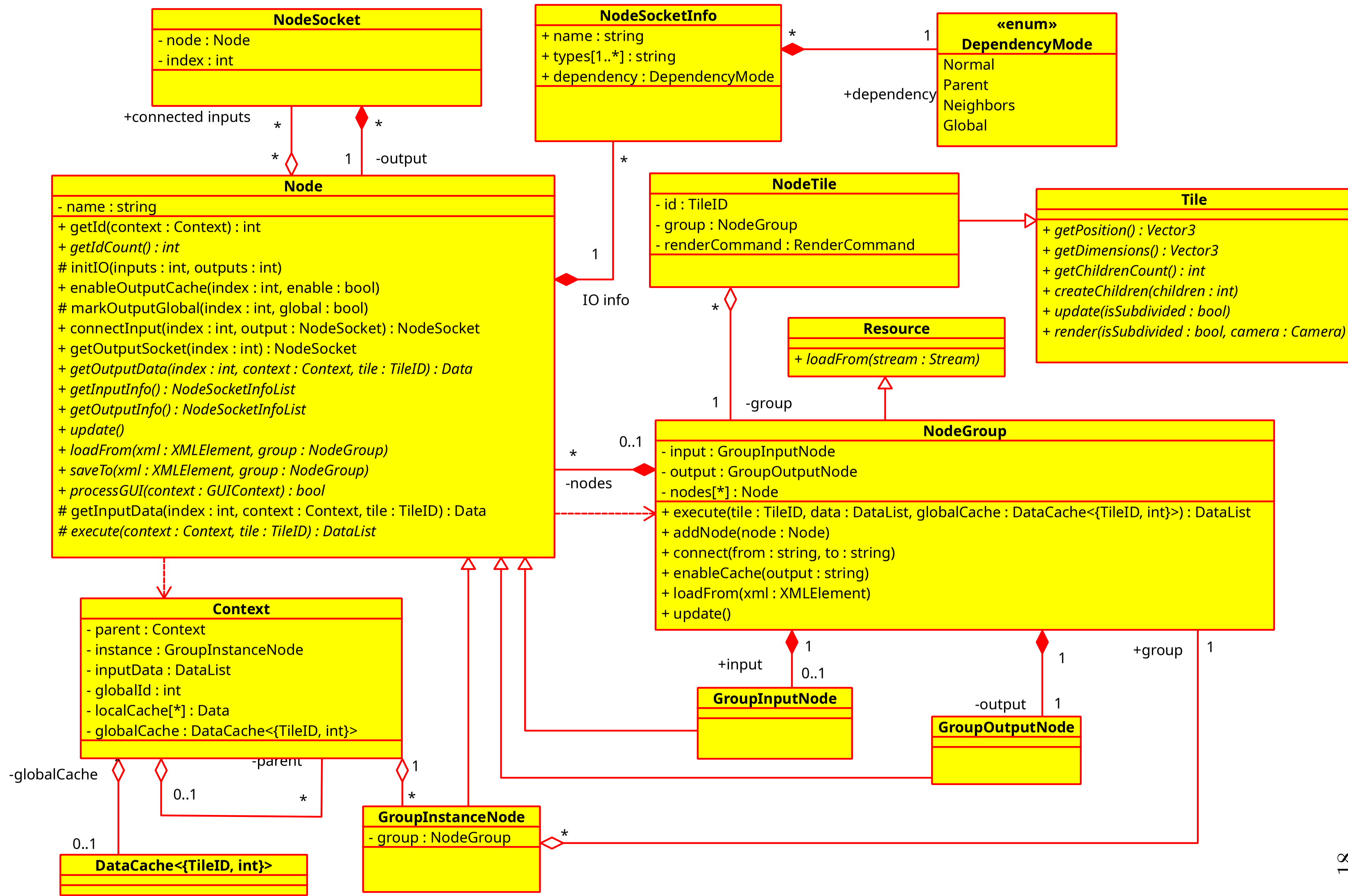
Proland obsahuje množstvo zaujímavých princípov, ktorými som sa inšpiroval pri navrhovaní vlastného riešenia. Návrh stavia na OpenGL technológii v spojení s technológiou na paralelne renderovanie Eyescale Equalizer[19]. Štruktúru návrhu je znázornená v 2.1 a 2.2.

2.3.1 Organizácia dláždic planéty a LOD

Povrch je reprezentovaný a organizovaný v stromovej štruktúre dláždic, ktorá je dynamický delená v závislosti od pozícii kamery, konkrétne podľa pomeru vzdialenosti a veľkosti dlaždíc. Do úvahy sa berie aj viditeľnosť na obrazovke (frustum culling), teda dlaždice ktoré sú úplne neviditeľné sa nevytvárajú. Pri pohybe či otočení kamery sa štruktúra aktualizuje – vytvoria sa nove dlaždice a uvoľnia nepotrebné dlaždice. Tento princíp funguje dobre pri plochých terénom, no pri planéte je to potrebné prispôsobím. Planéta je reprezentovaná ako kocka obalená na guľu, teda obsahuje 6 quad-tree terénov pre každú



Obrázok 2.1: Základ návrhu



Obrázok 2.2: Systém uzlov generátora

plochu kocky, ktoré sú globálne deformované do tvaru gule. Tento princíp je prevzatý z Prolandu.

`class Tile`

Abstraktná trieda reprezentujúca jednu dláždicu. Jej úlohou je načítanie dát danej dlaždice a ich renderovanie. Okrem toho poskytuje informácie o pozícii, veľkosti a koľko časti (potomkov) sa dá rozdeliť. Tato trieda sa nestará o delenie a neobsahuje priamo v sebe odkazy na potomkov.

`int getChildrenCount()`

Vráti počet potomkov, ktorá tato dláždica vytvori v prípade delenia alebo 0 - neumožňuje ďalšie delenie.

`Tile* createChildren(int i)`

Vytvori i-tého potomka. `i` musí byť menšie ako `getChildrenCount()`

`void update(bool isSubdivided)`

Načítanie resp. aktualizovanie dat. `isSubdivided` určuje či je dláždica v stromovej štruktúre rozdelená. Funkcia sa volá iba pri zmene, čiže pri vytvorení novej dláždice, rozdelení a spojení.

`void render(bool isSubdivided)`

Renderovanie dat, získaných pomocou `update`. Hodnota `isSubdivided` určuje či je dláždica rozdelená a je zaručené, že je totožná s hodnotou `isSubdivided` akú malo posledne volanie funkcii `Tile::update`. Funkcia je volaná na všetkých úrovniach stromovej štruktúry (nie len na listoch) a to v poradí od rodičov ku potomkom.

`class Planet`

Obsahuje stromovú štruktúru dláždic, ktoré sú uložené v `Planet::Node`. Táto trieda je zodpovedná za spôsob delenia (LOD) na základe informácii získane z `Tile`. Oddelenie ukladania stromovej štruktúry dláždic `Planet::Node` od dát dláždic `Tile` je z dôvodu poskytnutia priestoru na tvorbu alternatívnych systémov LOD a organizácii dat.

Dláždica sa rozdelí v prípade:

```
lenght < size * subdivideFactor
```

`length` je vzdialenosť kamery od pozície dláždice, získanej s `Tile::getPosition()`, `size` je najväčšia hodnota z rozmerov veľkosti dláždice, získanej pomocou `Tile::getDimension()` a `subdivideFactor` je konfiguračná konštanta. Pre správne fungovanie by mala byt minimálne 2.

`void update()`

Aktualizuje stromovú štruktúru na základe pozície kamery a dát získaných

z `Tile`. Tato funkcia aktualizuje dláždice, teda volá funciu `Tile::update` pri zmenách a `Tile::createChildren` v prípade rozdelenia. Nepotrebne dlaždice sa uvoľňujú príkazom `delete`.

`void render()`

Rekurzívne vyrenderuje všetky dlaždice v stromovej štruktúre pomocou `Tile::render(bool isSubdivided)`.

2.3.2 System generovania

Systém generovania je založený na baze grafu uzlov (`class NodeGroup`), kde jednotlivé uzly (podtriedy `class Node`) reprezentujú generovanie nových dát, transformácie dát alebo výstup - renderovanie. Jedna sa o podobný pristúp aký je použitý v Terragen[23], či World Machine[24]. Generovanie prebieha na úrovni dlaždíc, čiže naraz sa vygeneruje obsah celej dlaždice - vstupom je identifikácia dlaždice v rámci planéty (`struct TileID`, obsahujúce lokáciu a LOD úroveň) a výstupom sú dáta potrebné na renderovanie obsahu dlaždice (`class RenderCommand`). Vzhľadom na stromovú štruktúru organizácie dlaždíc, systém je navrhnutý s podporou závislosti na dátach nadriadených (rodičovských) dláždic alebo aj susedných dlaždíc na tej istej úrovni. Uzly sú implementované tak aby využili GPU v prípade, kde je to možné. Dáta (podtriedy `class Data`) predávané medzi uzlami poskytujú informácie o type a veľkosti použitej pamäti, ktoré sa používajú pri ukladaní vo vyrovnávacej pamäti.

Uzol

Uzol je reprezentovaný abstraktnou triedou `class Node`. Implementácia pri inicializácii urči počet vstupov a výstupov cez funkciu `initIO(inputs, outputs)` a informácie o nich poskytuje prostredníctvom funkcii `getInputInfo` resp. `getOutputInfo`. Počas inicializácii po zavolaní `initIO` môže označiť určité výstupy ako globálne použitím `markOutputGlobal(index, global)`. Toto označenie znamená, že dáta výstupu sú nezávisle na dláždici, čiže sú rovnaké pri generovanie hocijakej dláždice.

Samotne prepájanie uzlov funguje na základe indexov vstupu resp. výstupu s pouzitim `NodeSocket`. Príklad vytvorenia prepojenia 3. výstupu z uzlu `A` na 2. vstup uzlu `B`:

```
B->connectInput(2, A->getOutputSocket(3));
```

`DataList Node::execute(Context& context, TileID tile)`

Funkcia implementuje logiku daného uzlu. Je volana, ak je potrebny nejaký výstup z tohto uzlu pre určitú dláždicu identifikovanou s `TileID`. Je zabezpečené, že pre konkrétne `TileID` sa vrámci jedného vykonávania generovania tato funkcia vola iba raz. `Context` obsahuje potrebne informácie o vykonávaní generovania, ako je odkaz na inštanciu skupiny, z ktorej je tento uzol spus-

teny či odkaz na vyrovnávaciu pamäť, ktorá sa používa aby dáta negenerovali zbytočne viac krát.

Vrámci tejto funkcie sa dáta vstupov získavajú pomocou `DataList getInputData(int index, Context& context, TileID tile)`, kde `index` je index vstupu. `TileID` je identifikuje dláždicu, pre ktorú chceme dáta získať. Môže byť rovnaký ako vstupný `TileID` alebo od neho odvodený (napr. rodič). Tu istú funkciu je možné volať viac krát s rôznym `TileID` (napr. ak chceme získať dáta susedov). Implementácia musí pretypovať `Data` na očakávane podtriedu alebo použiť dynamickú konverziu na zistenie typu triedy (`dynamic_cast`). Je na implementácii zabezpečiť, aby nedošlo k nekonečnej rekurzii (napr. zavolať s TileID odkazujúcich na potomkoch).

Výstupom je pole dát (`Data`), ktoré musia zodpovedať poctu výstupom, ktoré sa nastavili prostredníctvom `initIO` a štruktúre, ktorá sa poskytuje cez `getOutputInfo`.

Skupina uzlov

Skupina uzlov reprezentuje graf uzlov a je implementovaná v `class NodeGroup`. Vstupy a výstupy sa určujú pomocou uzlov `GroupInputNode` a `GroupOutputNode`. Graf môže obsahovať maximálne jeden uzol z oboch typov. Skupina uzlov existuje hlavne z dôvodu lepšej organizácii dát, teda rozdelenia grafu do menších podgrafov, ktoré je možné opakovane používať - vytvárať inštancie. To sa robu použitím uzlu `GroupInstanceNode`, ktorému sa pridelí odkaz na požadovanú skupinu. Vstupy a výstupy inštancie sú automaticky zistene z pridelenej skupiny. Skupiny uzlov je možné používať viac krát vrámci jedného grafu (resp. inej skupiny uzlov) a je povolene vnáranie, napr. skupina A je použitá v skupine B a skupina B je použitá v skupine C).

Skupinu je možné vytvoriť programátorsky pridávaním uzlov pomocou `addNode` alebo načítanim zo pomocou `loadFrom`. Pri programátorskom vytváraní je potrebne po poslednej zmene zavolať funkciu `update`, ktorá pripraví skupinu aby bola spustiteľná. Do skupiny stačí pridať iba výstupný uzol (`GroupOutputNode`) a ostatne uzly pripojené uzly sa automaticky pridajú pri zavolaní `update` .

Identifikácia výstupov uzlov

Každý uzol v rámci celého grafu je unikátne identifikovateľný. Ako cely graf sa mysli hlavná skupinu uzlov (ta ktorá sa priamo spusta) vrátane všetkých inštancii podskupin uzlov. Identifikácia (ID) sa získava cez `Node::getId(Context)`. Každý uzol ma pridelene lokálne ID vrámci skupiny uzlov, v ktorej sa nachádza a v priebehu vykonávania uzlov kontext obsahuje globálne ID inštancii aktuálne vykonávanej skupiny uzlov (alebo 0 v prípade, ak sa jedna o hlavnú skupinu, teda inštancia neexistuje). Globálne ID uzlu sa vypočíta ako súčet týchto dvoch hodnôt. Každý uzol poskytuje počet ID, koľko potrebuje cez `getIdCount`, čo je predvolené počet výstupov. Identifikácia výstupu je teda ID uzlu plus index výstupu. Implementácia si to môže zväčšiť, napr.

`GroupInstanceNode` vracia počet, koľko potrebujú všetky uzly, ktoré sa v pridelenej `NodeGroup` nachádzajú.

Vyrovnavacia pamät

Aby sa zabránilo zbytočnému opakovanému generovaniu dát, používa sa vyrovnávacia pamäť. Každému výstupu uzlu je možné povoliť ukladanie do vyrovnávacej pamäti cez `Node::enableOutputCache`. Identifikácia výstupu uzla získavaná s `Node::getId` v spojení s identifikáciou dlaždice `TileID` tvorí kľúč pre vyhľadávanie vo vyrovnávacej pamäti. V prípade globálneho výstupu oznaceneho s `Node::markOutputGlobal` sa `TileID` ignoruje.

Výrovnávacia pamäť (`class DataCache<Key>`) používa LRU algoritmus, čiže naposledy použité dáta budu najneskôr odstranené. Vyrovnavacej pamäti sa nastaví kapacita pre každú skupinu, ktorá určuje maximálne množstvo Bytov, koľko môžu dáta dokopy v danej skupinu zaberať. Každa implementácia `class Data` poskytuje informácie o mnozstv pouzitej pamäti cez `Data::getMemoryUsage` a skupine cez `Data::getGroup`.

Skupina znamená typ pamäti v ktorej sa dáta ukladaju. Standardne bude minimálne jedna skupina pre RAM a jedna pre GPU RAM.

Spustenie skupiny uzlov

Skupina sa spusta s `NodeGroup::execute`, kde sa predá identifikácia dláždice, vstupné parametre a odkaz na globálnu vyrovnávaciu pamäť.

Priebeh `NodeGroup::execute(TileID, DataList inputs, DataCache cache)`:

- vytvori novy `Context`
- priradí globálne `cache` do `Context::globalCache`
- vytvori lokálnu cache a priradi do `Context::localCahe`
- priradí vstupne dáta `inputs` do `Context::inputData`
- nastaví `Context::globalId` na 0
- ziska dáta cez `Node::getOutputData` zo všetkých výstupov z `GroupOutputNode`
- vráti dáta v `DataList`

V rámci vykonávania skupiny sa postupne rekurzívne volaju `Node::getOutputData` na jednotlivých uzlov podľa ich prepojeniach a implementáciách `Node::execute`.

Priebeh vykonávania `Node::getOutputData`:

- vypočíta globálne id výstupu: `id = Node::getId(context) + index`
- skúsi vyhľadať dáta výstupu v `Context::localCache`
- ak neexistujú, skúsi vyhľadať v `Context::globalCache`
- ak neexistujú:
  - vykoná `Node::execute`
  - uloží všetke dáta do `Context::localCache`, aby v prípade opakovanej potreby dát nejakého výstupu z tohto uzlu sa nevykonávalo znova `Node::execute`
  - uloží všetke dáta, ktoré majú nastavené `Node::enableOutputCache`

do Context::globalCache
- vráti dáta

Pri volaní inštancii skupiny je priebeh zložitejší, keďže sa vstupuje do inej skupine uzlu, čo vyžaduje vytvorenia nového kontextu.

Priebeh vykonávania GroupInstanceNode::getOutputData:

- vypočíta globálne id seba: id = GroupInstanceNode::getId(context)
- vytvori nový Context odvodeny od vstupneho:
  - nový odkazuje na vstupny cez Context::parent
  - zdieľajú Context::localCache a Context::globalCache
  - priradí svoje globálne id do Context::globalId
  - priradí odkaz na seba do Context::instance
- z GroupInstanceNode::group zisti výstupny uzol GroupOutputNode, z ktorého získa dáta s novým kontextom cez GroupOutputNode::getInputData, teda dáta, ktoré sú pripojene na vstupe do GroupOutputNode
- vráti dáta

Pri vykonávaní skupiny môže byť potrebne získať dáta zo vstupov do skupiny. To vyžaduje sa vrátiť do nadradenej skupiny prostredníctvom inštancie uloženej v kontexte a znova používať rodičovsky kontext.

Priebeh vykonávania GroupInputNode::getOutputData:

- ak nie je inštancia v Context::instance, vráti dáta z Context::inputData
- inač, vráti dáta z inštancii zavolanim GroupInstanceNode::getInputData s rodičovskym kontextom, teda Context::parent

Načítavanie skupin uzlov z XML

Skupinu uzlov je možné načítať zo súboru alebo ineho zdroja vo formáte XML prostredníctvom NodeGroup::loadFrom. class XMLElement reprezentuje jeden XML element (nie cely súbor alebo dokument), ktorý je implementovaný použitím knižnice na spracovanie XML formátu. Každý uzol musí mať priradene unikátne meno v rámci danej skupiny, ktoré sa používa na identifikáciu uzlu pri vytváraní spojení. Nazvy vstupov a výstupov sa zisťujú z NodeSocketInfo::name prostredníctvom Node::getInputInfo resp. Node::getOutputInfo.

XML formát

Rodičovským elementom XML je <group>, ktorý obsahuje elementy typov:

- pripojenie jedného výstupu uzlu na jeden alebo viac vstupov, príklad: <connect from="uzol1.výstup" to="uzol2.vstup,uzol3.vstup2"/> pripojí výstup s názvom **výstup** z uzlu **uzol1** na vstupy s názvom **vstup** uzlu **uzol2** a **vstup2** uzlu **uzol3**

- <cache outputs="uzol1.výstup1,uzol2.výstup2"/> - zapnutie ukladania do globálnej vyrovnávacej pamäti pre výstupy určene v **outputs**,

ktorý ma rovnaký formát ako **to** v `<connect>`

- `<type name="nazov">` - uzol, kde **type** je typ uzlu (podtrieda z `class Node`) a obsahuje:
  - povinný atribút **name**, ktorý musí byť unikátny pre každú uzol vrámci celej skupiny, povoľujú sa hocijaké znaky okrem prázdnych znakov (medzera, tabulator...) a bodky `'.'`
  - nepovinne elementy **connect** a **cache** - jedna sa o lokálne alternatíva ako v `<group>` rozšírený o priame odkazovanie na svoje vstupy a výstupy neuvedením názvu uzlu (a bodky za nim), príklad: `<connect from="moj_vystup" to="iny_uzol.vstup,iny_moj_vstup"/>`
  - ostatne atributy a elementy definovane typom uzlu

Nezáleží na poradí elementov v `<group>` (napr. pripojenia môžu byť určene pred definovaním uzlov) a nezáleží na tom, či je **connect** a **cache** definovane lokálne (vo vnútru uzlu) alebo globálne (priamo v `<group>`). Počas načítavania sa vždy najprv načítajú všetky uzly a až potom prepájajú a nastavujú ukladanie do vyrovnávacej pamäte.

Priebeh `NodeGroup::loadFrom`:

- pre kazde `<connect>` registruje pripojenie cez `NodeGroup::connect`, tato funkcia zatiaľ nepripojí prostredníctvom `Node::connect`, to sa udeje až v `NodeGroup::update`
- pre kazde `<cache>` registruje zapnutia ukladania do vyrovnávacej pamäti cez `NodeGroup::enableCache`, taktiež samotne zavolanie `Node::enableOutputCache` sa udeje až v `NodeGroup::update`
- pre každý ostatný element skúsi vytvoriť uzol z názvu elementu pomocou `Factory::create` a zavolá na nom `Node::loadFrom` s odkazom na dany element (nie na `<group>`), štandardná implementácia `Node:loadFrom` vykoná:
  - načíta názov uzlu
  - pre každý lokálny `<connect>` resp. `<cache>` registruje pripojenie resp. ukladanie do vyrovnávacej pamäti
- zavolá `NodeGroup::update`

Priebeh `NodeGroup::update`:

- vytvori pripojenia s `Node::connectInput` pre všetky čakajúce pripojenia registrované cez `NodeGroup::connect`
- povoli vyrovnávaciu pamät s `Node::enableOutputCache` pre všetky čakajúce povolanie pridané cez `NodeGroup::enableCache`
- z `GroupOutputNode` rekurzívne prechádza všetky pripojene uzly a pridáva ich cez `NodeGroup::addNode`
- pre každý uzol sa aktualizuje jeho ID. Najprv sa uzly zoradia podľa `Node::name` a potom ID každého uzlu sa vypočíta ako súčet `Node::getIdCount` uzlov nachádzajúce sa pred nim.

- pre každý uzol sa zavolá `Node::update`

Príklad XML súboru

```
<group>
  <GroupInput name="input" sockets="some_input_attribute,other_attribute"/>
  <GroupOutput name="output" sockets="render_command"/>

  <!--
    nezalezi na~poradí
    order of nodes and connection is not important,
    connection are always created after nodes are loaded
  -->

  <GenerationNode name="terrain_generator" shader="terrain.glsl">
    <!-- other GenerationNode configuration -->

    <!-- connect input from other node -->
    <connect from="input.some_input_attribute" to="terrain_attribute"/>

    <!-- connect output to other node -->
    <connect from="height" to="terrain_renderer.heightmap"/>

    <!-- connection within same node -->
    <connect from="height" to="parent_height"/>

    <!-- enable output caching -->
    <cache outputs="height,colors">
  </GenerationNode>

  <RenderNode name="terrain_renderer" shader="terrain_render.glsl">
    <!-- RenderNode configuration -->

    <connect from="terrain_generator.colors" to="color"/>
    <connect from="output" to="output.render_command"/>
  </RenderNode>

  <!-- global connection and caching specification -->
  <connect from="normal_generation.normals" to="terrain_renderer.normals"/>
  <cache outputs="normal_generator.colors">

  <NormalsGenerationNode name="normal_generator"/>

  <!-- ... -->
```

```
</group>
```

2.3.3 Typy dat

Každý typ používaný v generátoru je podtriedou `class Data`. Typy sa delia podľa druhu primárne použitej pamäti resp. typu procesora na ktorom sa dáta používajú: RAM (CPU typ) a grafická RAM (GPU typ).

Základné CPU typy sú šablóna `class Value<T>` a `class Array<T>`, ktoré obsahujú 1 resp. pola dát primitívneho typy T (napr. `int`, `float`)

Medzi GPU typy patri `class Texture` a `class GPUBuffer` - reprezentujúce textúru a pole dát v GPU. Ďalším typom je `class VertexAttribute`, ktorý umožňuje použiť `class GPUBuffer` ako vertex atribúty pri renderovaní (napr. `GPUNode`). Tento typ nedrží priamo dáta, ale odkazuje sa na `class GPUBuffer`. Na ten istý `class GPUBuffer`, môže odkazovať viacero `class VertexAttribute`.

Špeciálnym typom je renderovaci príkaz – `class RenderCommand`, ktorý sluši na renderovanie obsahu dláždic s a je to výstupny typ z celého generátora. Každý príkaz je možné spúšťať opakovane. Príkazy si uchovávajú odkazy na dáta, potrebne na renderovanie. Odkazy na príkazy sa držia po dobu životností dláždic.

Uzly produkujúce určitý typ dát môžu vrátiť podtriedu daného typu, príkladom je `class RenderCommandList`, ktorý združuje viac renderovacich príkazov do jedného.

2.3.4 Typy uzlov

V tejto sekcii je popísaný návrh základných uzlov generátora. Konfigurácia uzlu znamenaj nemenné parametre nastavene pri inicializácii uzla alebo načítane z XML. Vstupy a výstupy sú vypísané v rovnakom poradí ako sa reálne nachádzajú v uzlu. Počet a typy vstupov resp. výstupov je často krát závisle od poskytnutej konfigurácií.

GroupInputNode a GroupOutputNode

Konfigurácia - počet a názvy vstupov resp. výstupov, typy sa automaticky zistia z pripojeni na ostatne uzly

GPUNode

Abstrakný uzol (trieda) umožňujúce vykonávanie operácii na GPU prostredníctvom GLSL.

Konfigurácia - GLSL shader program

Vstup - väčšina vstupov je automaticky zistenych z GLSL a tie sú zoradené podľa názvu:

- `Value<T>` uniformy - automaticky zistene z GLSL
- `Teture` - automaticky zistene textúry z GLSL + specialne sample (ParentSampler, NeighbodSampler)
- `VertexAttribute` - automaticky zistene vertex atributy z GLSL

Výstup - urceny podtriedou

TextureGenerationNode

Generovanie dát do textúry. Odvodene od `GPUNode`.

Konfigurácia

- GLSL shader - iba fragment (vertex je jednotný a to vyrenderovanie do celej dláždice)
- rozlíšenie výstupu (s možnosťou nechať automaticky určiť z rozlíšení vstupných textúr)

Vstup - podľa GPUNode
Výstup - `Texture` - niekoľko textúr, nutnosť nastaviť typ pre každý výstup

RenderNode

Renderovanie výstupu, automaticky zapína ukladanie vstupov do vyrovnávacej pamäti. Uzol nevykonáva priamo renderovanie, ale vráti renderovaci príkaz (`RenderCommand`), ktorý sa následne vola, keď je potrebne vyrenderovať obsah dláždice. Renderovaci príkaz si zachová odkaz na vstupne dáta.

Konfigurácia - GLSL shader
Vstup - podľa `GPUNode`
Výstup - `RenderCommand command`

TextureNode

Umožňuje načítať textúru zo súboru v bežne podporovaných obrazových formátoch (png, jpg, tga...)

Konfigurácia - názov súboru s textúrou
Výstup - `Texture texture` (globálne dáta)

GridNode

Generuje dáta pre renderovanie mriežky alebo terénu. Podporuje ”skirts” pre implementáciu LOD terénov.

Konfigurácia

- rozlíšenie vertex mriežky
- hlbka “skirt” (alebo ziadne skirt)

Výstup - `VertexAttribute position` positions (globálne data)

TileInfoNode

Poskytuje základne informácie o vykonávanej dlaždice získane z `struct TileID`.

Výstup - `Value<T>` hodnoty

RenderCommandListNode

Umožňuje spájať viacero renderovacich príkazov do jedného. Slúži hlavne na organizáciu postupnosti renderovania a zjednodušenie výstupov zo skupín uzlov. Príkazy sa vykonávajú v tom istom poradí ako sú pripojene na vstupe do uzlu.

Vstup - `RenderCommand` - ľubovolný počet renderovacich príkazov
Výstup - `RenderCommandList command`

2.3.5 Sprava prostriedkov

Spravu a načítavanie zdielaných prostriedkov zabezpečuje `class ResourceManager`. Prostriedky sa získavajú s `getResource<T>(string name)`, kde **T** je typ (podtrieda `class Resource`) a `name` je názov súboru, z ktorého sa ma načítať pomocou `Resource::loadFrom`. Prostiedky sú zdielané a automaticky odstránené ak sa nepoužívajú (nie je na nich referencia mimo manažéra). Prostriedok sa načítava tak, že manažér vytvori **T** a zavolá na nom `Resource::loadFrom` s odkazom dáta súboru.

2.3.6 Funkcie na procedurálne generovanie (v GLSL)

Generovanie plne prebieha na GPU prostredníctvom podtried `class GPUNode`. Základným programovacím jazykom je GLSL, ktorý sa v OpenGL používa na tvorbu programov pre GPU (shader). Tieto programy sa načítavajú priamo zo zdrojového kódu za behu aplikácie. V rámci návrhu sa počíta rozšírenie GLSL o jednoduchú funkcionalitu vkladať iných časti kódu (include) aby sa mohla predvytvoriť sada funkcii (napr. funkcie šumu) uľahčujúcé programovanie daných uzlov.

2.4 Paralelné renderovanie a generovanie

Návrh používa sort-first distribúciu, konkrétne kde jeden PC (alebo GPU) renderuje jednu súvislosti časť z celkového výstupu na zobrazovacom zariadení. Architektúra aplikácie je navrhnutá ako master (server) + klienti. Server spracováva užívateľské vstupy (myš + klávesnica) a vykonáva hlavnú logiku (aktualizácia scény, pozície kamery, zmenu parametrov a konfigurácie generátora). Klienti sú určené na renderovanie a generovanie dat. Klienti sa pripoja na server prostredníctvom sieťového protokolu a prijímajú príkazy. Každý klient generuje len dáta potrebne na renderovanie svojej časti výstupu.

Vyber tejto konfigurácii je hlavne prispôsobený zostave, na ktorej sa prototyp bude testovať. Ta pozostáva z 5x4 FullHD monitoroch a 5 PC. Každý PC ma 1 grafiku (NVIDIA GTX 1080 Ti), na ktorom sú pripojene 4 monitory, teda jeden stĺpec monitorov. K tomu je dispozícii 6. PC (bez GPU výstupu), ktorú je možné použiť ako server. Všetky tieto PC sú prepojený 10Gb ethernet linkou cez switch.

Konfigurácia

- server ma informácie o rozlíšení celkového výstupu, na základe toho vypočíta globálnu projekčnú maticu kamery (frustum matrix)
- každý klient ma informáciu o jeho časti z celkového výstupu (viewport)

Priebeh jedného snímku:

- server na základe vstupov aktualizuje kameru a ostatne dáta
- server pošle zmeny oproti poslednému snímku klientom
- server pošle príkaz na renderovanie snímky
- klienti si aktualizujú zmeny dat
- klienti z globálnej projekčnej matice a viewportu vypočítajú lokálnu projekčnú maticu – projekcia pokrývajúcu iba ich časti výstupu
- klienti aktualizujú stromovú štruktúru, vygenerujú pripadne nove dáta s `Planet::update` a vyrenderujú ich s a `Planet::render`
- klienti informujú server o dokončení spracovania daného snímku

2.4.1 Asynchrónna komunikácia

Komunikácia medzi serverom a klientmi môže byť asynchrónna, teda server môže posielať dáta na renderovanie ďalších snímkov predtým nez dostane potvrdenie o vyrenderovaní aktuálneho snímku. Server by mal nastavený počet snímkov, koľko môže predbiehať pred potvrdením od všetkých klientov. Na to je potrebne aby klienti používali frontu na ukladanie príkazov od servera čakajúcich na spracovanie. Tento pristúp umožňuje využiť paralelne komunikačnú linku na posielanie dát medzi serverom a klientmi a popri renderovaní na klientoch. Tým sa môže zvýšiť FPS, ale aj zväčšiť odozva (hlavne v prípade posielania niekoľkých snímkov dopredu).

2.4.2 Synchronizácia výstupu

Pre správne užívateľské vnímanie môže byť doležité, aby výstup bol zobrazený naraz v tom istom momente, čiže nezáleží na tom kedy klienti začali renderovať určitý snímok, ale na tom kedy sa dokonči a zobrazí (vykoná sa swap buffers). Synchronizovať výstup je možné tak ze klienti počkajú na príkaz zo servera, ktorý ho vyšle potom, keď všetci klienti budu mať vyrenderovaný daný snímok (informujú o tom server). V prípade asynchrónnej komunikácii nastáva, že príkaz na zobrazenie určitého snímku príde po sérii iných príkazov, preto je potrebne tento typ príkazu spracovať prioritne pred ostatnými.

2.4.3 Vynechávanie snímkov

V prípade asynchrónnej komunikácii sa môže stat, že niektorí klienti môžu zaostávať o vacsie množstvo snímkov oproti ostatným (napr. ze potrebovali vygenerovať naraz väčšie množstvo dat). Aby takéto prípady neblokovalo ostatných klientov, zaostávajúci klienti môžu preskočiť renderovanie niekoľkých snímkov. Tento princíp nie jekompatibilny so synchronizáciou výstupu. Synchronizácia výstupu znamená prispôsobiť sa najpomalšiemu klientovi a vynechávanie snímku najrychlejsiemu klientovi. Server môže predbiehať aj pred najrychlejsim klientom, ale maximálne o 1 snímok (ak by bolo viac, tak by sa zbytočne preskočili).

To je možné riešiť tak, že sa vykoná iba posledný príkaz na renderovanie snímku čakajúci vo fronte. Ostatne príkazy je nutne pred samotným renderovanim spracovať normálne, keďže dáta zo servera sú posielané vo forme zmien. Počas spracovávania ostatných príkazov je možné, že medzitým príde novší renderovaci príkaz, preto je nutne si najprv vyhľadať a označiť aktuálne posledný renderovaci príkaz vo fronte, následne vykonávať postupne ostatne príkazy vo fronte pred nim a potom vykonať renderovanie. Je to z dôvodu, aby sa nedostalo k teoretickému nekonečnému zacykleniu, keď príkazy na aktualizovanie dát prichádzajú rýchlejšie ako trvá ich spracovanie a tým pádom sa stále renderovaci príkaz nahrádza novším.

2.4.4 Eyescale Equalizer

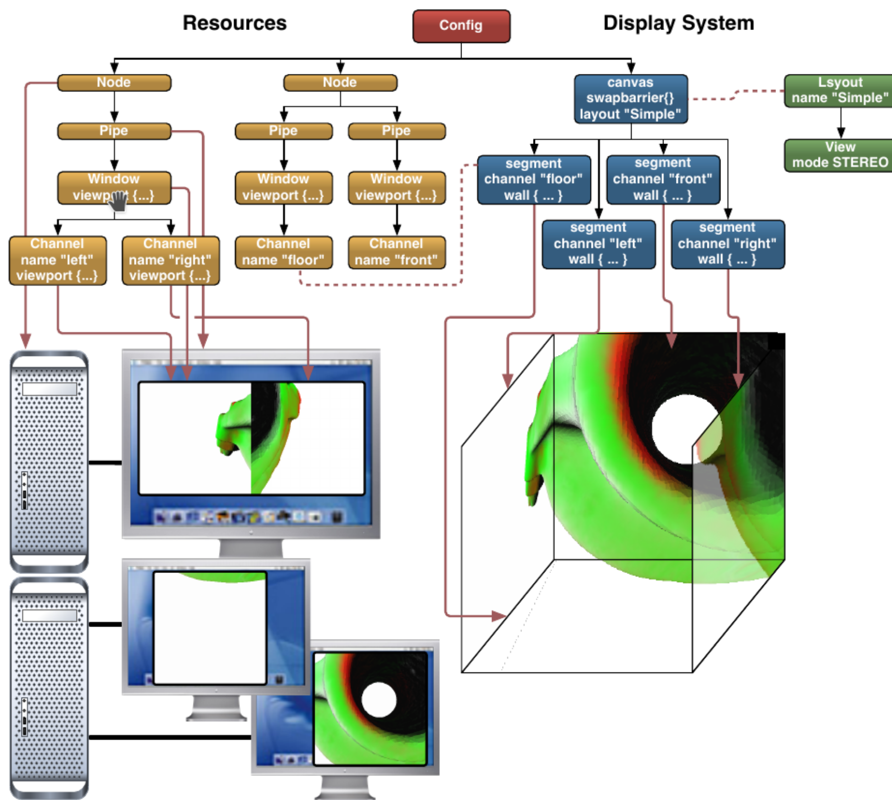
Jednou z možnou implementáciou je použiť framework určený pravé na tvorbu paralelných OpenGL aplikácii. Podporuje vymenované vlastnosti okrem vynechávania snímkov. Klienti sa nepripájajú priamo na server resp. hlavnú aplikáciu ale na predprogramovaný Equalizer server, ktorý hrá rolu manažéra – stara sa o načítanie konfigurácii a riadenie renderovacich uloh. Equalizer server, hlavná aplikácia aj klient môžu byť spustené vrámci jedného procesu.

Framework je určený na rozsiahle systémy, ukážka jeho konfigurácie je znázornená na 2.3. Stručný popis, čo je potrebne nakonfigurovať:

- štruktúra zariadení: node (PC), pipe(GPU), window, channel (cast okna)
- rozmiestnenie výstupov: canvas – popisuje súvislí plochu, často poskladaný z viacerých segmentov (napr. obrazoviek) ktoré odkazujú na channel
- logicke rozdelenie výstupu: layout, pozostavajuci z logickych časti (view), layout sa priradí do canvasu

Equalizer používa knižnicu Collage[20], určenú na tvorbu sieťových distribuovaných aplikácii. Tato knižnica poskytuje systém zdieľania dát medzi PC prostrenictvom `co:Object` alebo `co:Serializable`.

Konfigurácia priebehu renderovania jedného snímku sa robí v compounds tree – strom úloh popisujúcich, akým spôsobom a na akých GPU sa ma scéna



Obrázok 2.3: Ukážková konfigurácia Eyescale Equalizera (obrázok z dokumentácie prístupnej cez [19])

renderovať a následne spájať do výstupov. V rámci tohto stromu sa špecifikujú autoballancing systémy (interne nazývaných ako „equalizer“)

2.4.5 Zdieľanie generovaných dát

Aby čiastočne zabránilo opakovanému generovaniu dát je možné implementovať vyrovnávacou pamäť, ktorá zdieľa dáta medzi klientmi. Ak nejaký klient ma už dane dáta vygenerované informuje o tom ostatných klientov, ktorý si ich môžu nechať preposlať. Tento prístup je použiteľný iba, ak čas potrebný na poslanie dát je menši ako čas potrebný na ich lokálne vygenerovanie. Jedna sa teda o rozšírenie `class DataCache`.

2.4.6 Sprava prostriedkov

rozšírenia Na spravu prostriedkov sa používa `class ResourceManager`, ktorý pristupuje k súborov prostredníctvom `class FileProvider`. V prípade kli-

entoch sa použije podtrieda `class RemoteFileProvider`, ktorá získava dáta súborov cez serveru v prípade, že sa nenasli lokálne.

Kapitola 3

Implementácia

Projekt bol vyvíjaný a testovaný na Linuxe (Arch Linux) s použitím GPU AMD RX 550. Používa iba crossplatform knižnice, čiže nemalo by robiť problém ho rozbehať na Windowse. Pre účely prototypu stačilo implementovať iba určitú časť návrhu. Kvôli jednoduchosti je vynechaná sprava prostriedkov (dáta sa získavajú priamo zo súborov). Niektoré uzly generátora sú nahradené alebo vynechané a XML formát skupín uzlov je zjednodušený.

3.1 Použite technológie

Použite technológie sú v súlade s požiadavkami, avšak minimálne verzie niektorých technológii boli zvýšene: C++14 namiesto C++11 a OpenGL 4.0 namiesto 3.0. Z počiatku sa začal vývoj na lokálnom bežnom počítači s použitím knižníc GLFW, GLEW. Po integrácii s Eyescale Equalizer už dane knižnice nie sú potrebne.

Zoznam technológie a knižníc:

- C++14
- CMake - build systém
- OpenGL 4.0 a vyššie
- GLSL - programovanie GPU shadrov pre OpenGL
- GLFW - vytvorenie a sprava OpenGL okien
- GLEW - spojenie (bindovanie) na OpenGL API
- Eyescale Equalzer - paralelne renderovanie a sprava okien
- catch2 - unit testy
- imGUI - knižnica pre grafické rozhranie
- TinyXML2 - práca s XML súbormi
- vlastna matematická knižnica na pracú s vektormi a maticami

3.2 Generator

Implementácia systému generovania sa drží štruktúre vytvorenej v návrhu až na mierne odlišnosti. Informácie o vstupoch a výstupoch uzlov boli zjednodušené iba na poskytovanie jeho názov a sú priamo integrovane do `class Node`. Ostatne informácie nie sú na ich funkčnosť potrebne. Sú v návrhu hlavne z dôvodu poskytnutia informácii užívateľom v prípadnom rozšírení o grafické rozhranie. Zjednodušená bola aj konfigurácia prepojenia uzlov v XML. Implementovaná je globálna varianta konfigurácie prepojení.

`class GPUNode` ma implementované automatické zistenie OpenGL vstupov z GLSL: uniformov a textúr, vrátanie odlíšenia či sa jedna o obyčajnú textúru alebo odkaz na dáta rodiča (napr. v GLSL typ **sampler2DParent**). Informácie sa zisťujú pomocou OpenGL API pomocou **glGetActiveUniform()**. Vynechané je univerzálne pripájanie atribútov – `class VertexAtrribute` a `class GPUBuffer` a teda aj uzol na renderovanie `class RenderNode`. Namiesto toho je implementovaný uzol `class TerrainRenderNode`, ktorý nahrádza funkcionalitu pôvodného `class RenderNode` v kombinácii s `class GridNode`.

`class TextureGenerationNode` je plne implementovaná a podporuje automatickú detekciu výstupov. Na to OpenGL API nemá funkcie, preto sa tieto informácie získavajú jednoduchým parsovaním priamo zo zdrojového kódu GLSL. Tieto informácie sa používajú na určenie typov textúr. Textúry vytvárajú počas behu uzlu spolu s Framebuffer Object, kde sa tieto textúry pripoja a ten sa použije ako cieľ renderovania.

Pridaná bola trieda `class ValueBase` (z ktorej dedí `class Value<T>`), ktorá obsahuje virtuálnu metódu, umožňujúca nastavenia hodnoty priamo nastavenie OpenGL uniformu pomocou `glUniform*` funkcii. Dedičná `class Value<T>` ma pomocou šablóny a makier implementovanú danú metódu na väčšinou používaných typov: vektory, matica a skalary v typoch `float`, `double`, `int`, `unsigned int`.

Špeciálnou novou podtriedou `class ValueBase` je `class DynamicValue<T>`. Jej úlohou je poskytovať nestatické informácie pri renderovani – dáta, ktoré sa môžu meniť medzi jednotnými snímkami (napr. pozícia kamery). Trieda je určená primárne na nastavovanie uniformov GLSL programom. Dáta nie sú uložené priamo v triede, ale trieda obsahuje odkaz na funkciu, ktorá sa zavolá s indexom uniformu, na ktorý je potrebne ich nastaviť.

Ukážka zdrojového kódu implementácie uzlu:

```
//.h
class ConditionNode : public Node {
public:
  ConditionNode();
  virtual DataList execute(Context&, const TileID&) override;
};
```

```
//.cpp
ConditionNode::ConditionNode() {
  //konfiguracia vstupov a vystupov
  addInput("condition");
  addInput("true");
  addInput("false");
  addOutput("value");
}

DataList ConditionNode::execute(Context & ctx, const TileID & tile) {
  // vstupne data sa ziskavaju indexami podla
  // poradia pridavania vstupov v~konfiguracii
  auto condition = getInputData<Value<bool>>(0, ctx, tile);
  ASSERT(condition != nullptr);
  return { getInputData(condition ? 1 : 2, ctx, tile) };
}
```

3.3 GLSL programy

Normálne GLSL programy (shaders) sú rozdelené do viacerých zdrojových kódov (pre vertex, fragment...). Aby sa zjednodušila ich tvorba a umožnilo zdieľanie spoločných časti kódu je implementovaný jednoduchý preprocesor, ktorý z jedného kódu vytvorí zdrojové kódy pre každý nájde shader. Pomocou `#pragma include "subor.glsl"` sa vkladá kód z iného súboru (podobne ako to ma C/C++), ktorý sa tiež spracuje preprocesorom. S `#pragma shaders TYP1 TYP2..` sa špecifikujú typy shadrov (VERTEX, TESS_CONTROL, TESS_EVALUATION, GEOMETRY alebo FRAGMENT), pre ktoré je určená nasledujúca časť kódu. Pomocou `#pragma shaders ALL` sa nastaví aplikácia kódu pre všetky shadre. Každý typ môže byt použitý viac krát a výsledný kód sa vytvori spojením všetkých časti.

Následne je ukážka kódu generovania – určene pre `class TextureGenerationNode` a renderovania – určené pre `class TerrainRenderNode` s pripojenými vstupmi z generátora.

Ukážka kódu generovania:

```
#version 440 //verzia GLSL
// kniznice
#pragma include "generation.lib.glsl"
#pragma include "tile.lib.glsl"
#include "noise.lib.glsl"

// nastavuje pocet bitov za farbu (zlosku) pre~nasledujuci vystup (texturu)
#pragma texture 8
```

```
out vec3 colorTex;
#pragma texture 32
out float heightTex;

//...

// jednuducha implementacia mierneho pasma
vec4 temp(float H){
    vec3 P = vec3(getGlobalPosition());

    //fbm(vstup, pocet oktav) - Fractional Brownian motion s~Perlin Noise
    float h = H + fbm(P/2000, 2) * 1000;

    //fbmo - ako fbm ale vracia hodnotu v~rozsahu 0 az 1 namiesto -1 az 1
    // green, yellow su preddefinovane fraby
    vec3 col = green * fbmo(P + 2, 5) + yellow * fbmo(P/250 + 10, 5) * 0.2;

    return vec4(col, h);
}

// podobne ostatne: desert, tropical, cold

void main(){
    vec3 p = vec3(getGlobalPosition());
    float size = float(getPlanetSize());

    //generovanie rozlosenie kontinentov
    float continents = clamp(fbm(p/size, 10), -1, 1) * 3000;

    //vypocet biom
    vec4 de = desert(continents);
    vec4 te = temp(continents);
    vec4 tr = tropical(continents);
    vec4 co = cold(continents);

    // zmiesanie biom na~zaklade vlhkosti
    float hum = fbmo(p/(size/2) + 1,10);
    vec4 v2 = mix(de, tr, clpo(hum, 0.6));

    // zmiesanie biom na~zaklade teploty
    float tem = fbm(p/(size/2) + 2,10) * 4;
    vec4 v;
    if(tem > 0) v~= mix(te, v2, clpo( tem, 0.4));
    else        v~= mix(te, co, clpo(-tem, 0.4));

    //vystupy
    colorTex = v.rgb;;
    heightTex = v.w;
}
```

Ukážka kodu renderovania:

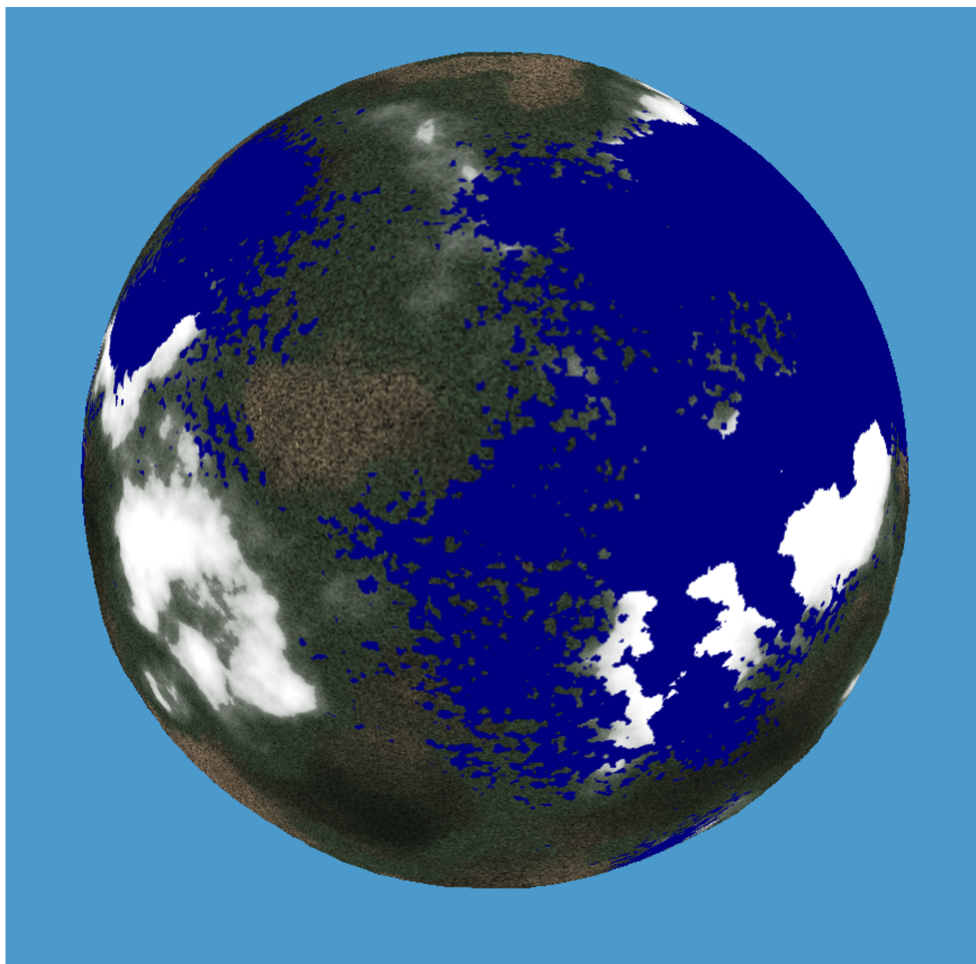
```
#version 440
#pragma include "render.lib.glsl"
#pragma include "tile.lib.glsl"

//vstupy z~generatora
uniform sampler2D heightTex;
uniform sampler2D colorTex;

#pragma shader VERTEX
void main() {
    process(); //preda poziciu do fragment shader
    dvec3 gpos = getGlobalPosition();

    //aplikovanie vysky terenu
    float h = texture(heightTex, getPosition().xy).x;
    gpos = addHeight(gpos, h);

    vec4 rpos = vec4(camera * dvec4(gpos, 1));
    rpos.z = logZ(rpos.w, float(maxDistance)); //pouzie logaritckej hlbky
    gl_Position = rpos;
}

#pragma shader FRAGMENT
void main() {
    color = texture(colorTex, getPosition().xy);
}
```

V 3.1 a 3.2 sú ukážky planéty vygenerovanej s použitím tejto konfigurácii.

3.4 Paralelne renderovanie

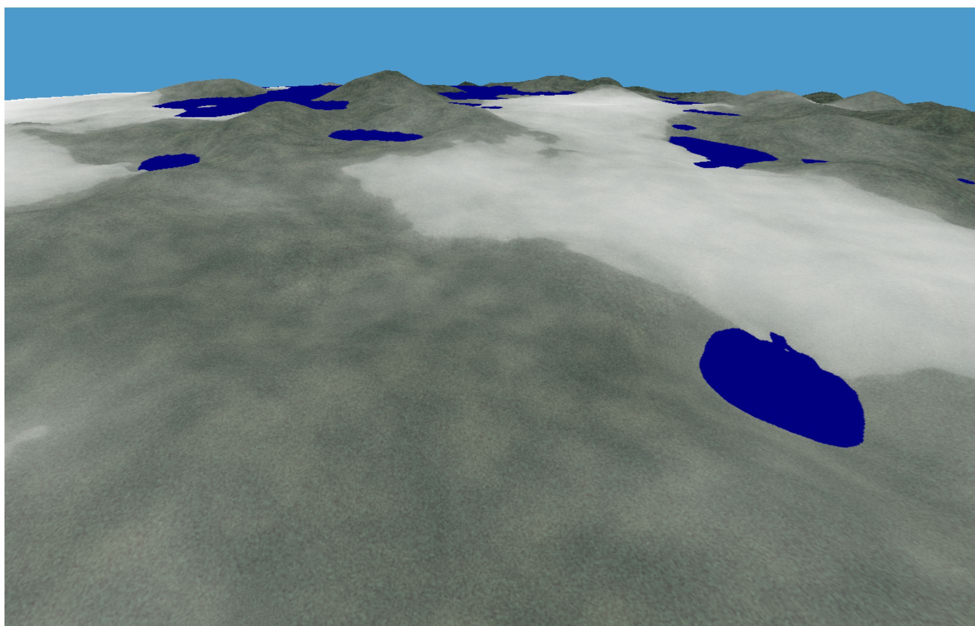
Podla návrhu bol použití Equalizer v spojení s first-sort distribúciou. Ako základ projektu sa použil Sequel. Jedna sa o zjednodušené API postavené nad Equalizer API, kde je uz implementovaná väčšina hlavnej logiky paralelného renderovanie (node, pipe, channel...).

Počas implementácii sa naskytli problémy s kompatibilitou, keďže Equalizer používa časti OpenGL API, ktoré od verzii OpenGL 3.2 s použitím Core profilu nie sú dostupné. Môže byť dostupná s použitím Compatibilty profilu, ale ten podľa špecifikácie nemusí byť ovládačmi implementovaný a na vývojovom a testovacom zariadení neboli. To si vyžadovalo úpravu interných kódov knižnice. Pre potreby prototypu bolo potrebne upraviť inicializáciu OpenGL (aby vytvorilo kontext pre verziu 3.2 a vyššie) a nahradenie funkcii na pracú



Obrázok 3.1: Ukážka celej planéty

s framebuffer objektami (napr. `glBindBuffer`, pôvodný kód používal rozšírenie glBindBufferARB). Pre plnú podporu celého Equalizer frameworku je potrebné upraviť omnoho viac časti, preto pri pokusu o použitia inej konfigurácii distribúcii, prototyp nemusí fungovať.



Obrázok 3.2: Ukážka priblíženej časti planéty

Kapitola 4

Testovanie

4.1 Unit Testy

Na unit testy je použitá C++ knižnica **catch2** a zdrojové kódy testov sa nachádzajú v priečinku **tests**. Príklad jednoduchého testu na kapacitu vyrovnávacej pamäti:

```
TEST_CASE( "cache", "[cache]" ) {

    DataCache<int> cache;

    SECTION( "capacity" ) {

        REQUIRE(cache.getDefaultCapacity() == 0);

        REQUIRE(cache.getCapacity(1) == 0);

        cache.setDefaultCapacity(20);
        cache.setCapacity(2, 30);

        REQUIRE(cache.getDefaultCapacity() == 20);
        REQUIRE(cache.getCapacity(1) == 20);
        REQUIRE(cache.getCapacity(2) == 30);
    }
}
```

4.2 Výkonové testy

4.2.1 Spôsob testovania

Testovalo sa na video-stene v SAGELab, ktorá pozostáva z 5 PC po jednej GPU (NVIDIA GTX 1080 Ti), každá s výstupom na 1 stĺpec monitorov (dokopy 5x4 FullHD monitorov). Všetky tieto PC sú prepojený 10Gb ethernet linkou. Prebiehalo tak, ze sa aplikácia normálne ovládala užívateľom (pohyb a smer pohľadu) a popritom sa pre každý snímok zaznamenávali potrebne informácie do súborov vo formáte CSV, ktoré sa následne spracovali pomocou analytického SW (python: jupyter + pandas). Okrem interaktívneho prechádzania sa použil automaticky režim, kedy sa kamera sama pohybuje vpred v určenej nadmorskej výške. Parametrom testovania je scéna (konfigurácia generátora), násobok rýchlosti (reálna rýchlosť je relatívna vzhľadom na nadmorskú výsku), faktor delenia dlaždíc a maximálna kapacita vyrovnávacích pamäti.

Zaznamenávané údaje:

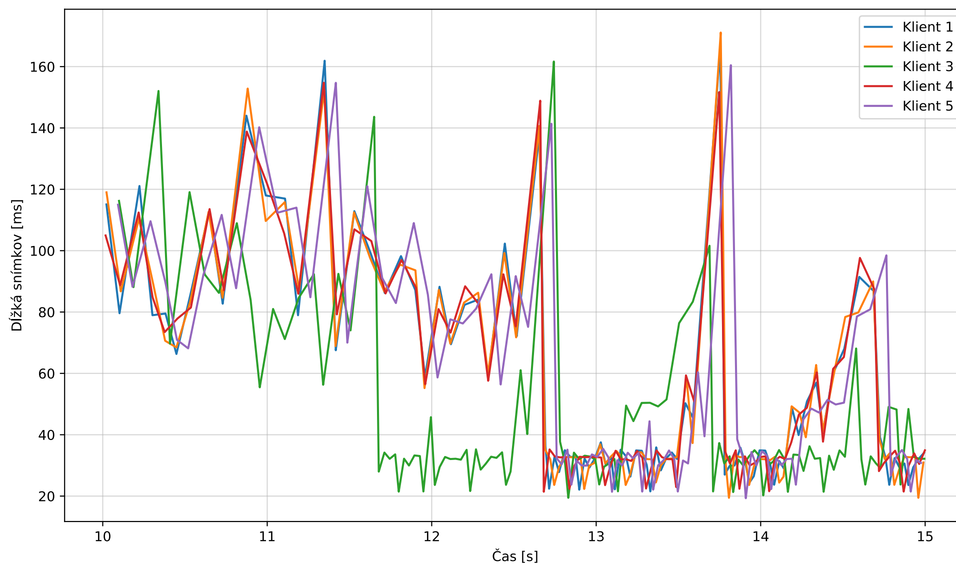
- poloha a rotácia kamery
- doby trvania renderovania a generovania
- počet vyrenderovanych dlaždíc
- pocty pridaných (generovanych) a odobraných dlaždíc
- informácie o vyrovnávacej pamäti:
  - množstvo vyuzitej kapacity
  - pocty a veľkosti pridaných a uvolnených dát
  - pocty úspešných a neúspešných pokusov vyhľadania dát

4.2.2 Výsledky

V nasledujúcich grafov je vyber 5-sekundový úsek z testovania jednej scény. V grafu 4.1 sú zobrazene doby trvania spracovania snímkov na všetkých 5 klientov. Je možné vidieť ze vo väčšinu prípadoch majú rovnakú priebeh, ale najviac sa odlišuje tretí klient, keďže bol zároveň aj aplikáciou (čiže nebola nutná komunikácia cez sieť). V ostatných ukázaných grafov je zobrazovaná štatistika 1. klienta z toho istého testovania v tom istom časovom úseku. V grafe 4.2 je zobrazený počet nových dlaždíc, ktoré bolo potrebne získať v jednotlivých snímkoch a zmena pozície pozície kamery. Pri tom pocte dlaždíc sa nerozlišuje medzi vygenerovanými a dátami získanými z vyrovnávacej pamäti. V grafe 4.3 sú zobrazene časy spracúvania jedného snímku. Je vidieť, ze dĺžka renderovanie tvori najmenší čas a je približne stabilná. Najväčšiu rolu hra generovanie. Najvyššie hodnoty zobrazujú koľko trvalo prechod na nasledujúci snímok. Rozdiel medzi renderovaniim + generovanim a celým snímkom

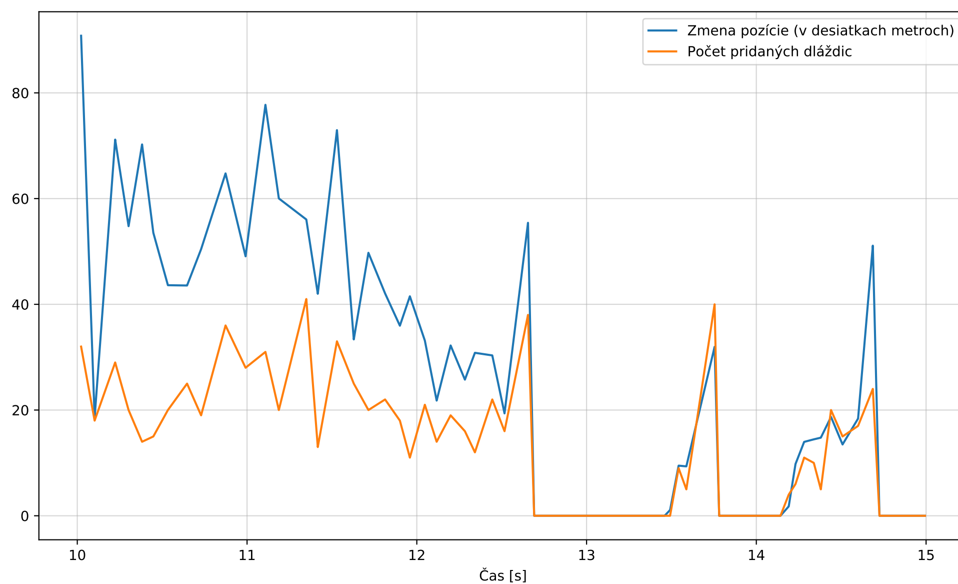
Statistika	**20% kvantil**	**Priemer**	**80% kvantil**
FPS	10.29	26.46	40.4
Pridané dláždice	0	8.88	23.2
Renderované dlaždice	182	606.10	930
Uspešné vyhľadanie v cache	0	4.41	16
Neúspešné vyhľadanie v cache	0	24.44	64

Tabuľka 4.1: Štatistika výkonového testovania

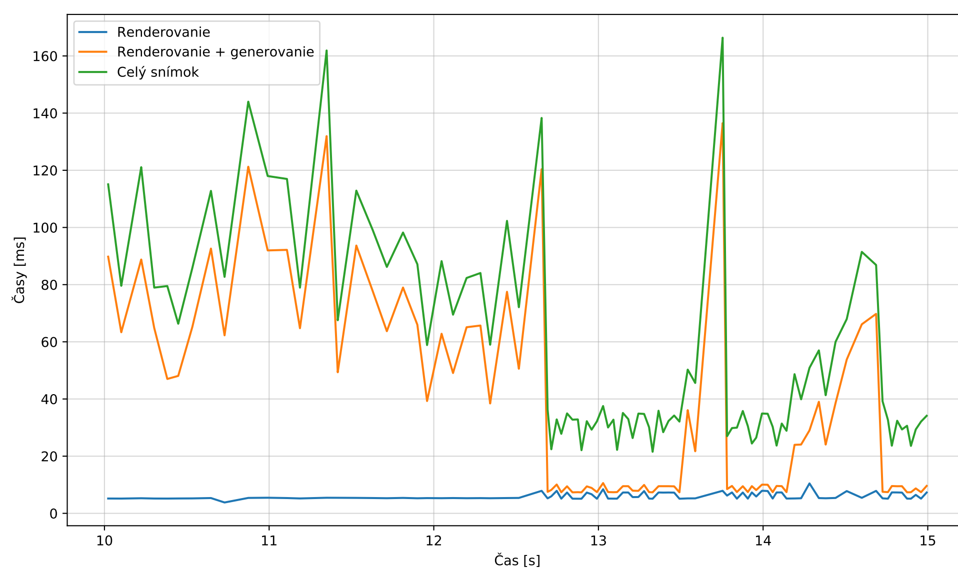


Obrázok 4.1: Časy spracovania snímkov klientoch

je z dôvodu synchronizácií výstupov klientov na 60 FPS. V tabuľke 4.1 je zhrnutá štatistika testovania danej scény, ostatne grafy, detailnejšie štatistiky a použité skripty sú v prílohe.



Obrázok 4.2: Počty pridaných dlaždíc a zmeny pozície v snímkoch klienta 1



Obrázok 4.3: Prehľad časov spracovania snímkov klienta +

KAPITOLA 5

Užívateľská príručka

Pred použitím je potrebne projekt skompilovať. Väčšina závislosti je priamo kódy Equalizeru sú v projekte z dôvodu, že ich bolo potrebné prispôsobiť. Ostatne použité technológie sú tiež v projekte vo forme zdrojových kódov.

5.1 Kompilacia

Pred kompilaciou je potrebne mať nainštalované:

- CMAKE a build system (testovane na `cmake` 3.11 s `make` 4.2)
- C++14 kompilator (testovane na `gcc` 7.3)
- OpenGL vyvojarske kniznice

Skompilovat na Linuxe je mozne spustenim v priecinku s projektom `./build.sh` alebo rucne:

```
mkdir build
cd ./build
cmake ..
make
```

Po uspesnej kompilacii sa v priecinku `build/bin` sa vytvoria programy `app` a `eqServer`.

5.2 Parametre

Aplikácia sa spúšťa s argumentami, čiže najjednoduchšie s príkazového riadku. Obsahuje 2 skupiny argumentov – pre Eyescale Equalizer a pre aplikáciu. Argumenty začínajúce na `--eq-` resp. `--co-` sú určené pre konfiguráciu Equalizera a argumenty začínajúce na `--app` sú určene pre nastavenia aplikácií.

Konfiguracne parametre Equalizera je mozne najst v jeho dokumentacii (možnosť stiahnuť z [19] v sekcii `A. Command Line Options`.

Parametre pre aplikáciu:

`--app-base DIRECTORY` – cesta k priečinku s dátami, ak sa neurci nastavi sa na aktualny pracovny priecinok

`--app-config FILE` – cesta k suboru s konfiguraciou, ak sa neurci skusi sa nacitat z `config.xml` z datoveho priecinka, ak sa nenajde pouziju sa predvolene hodnoty.Popis parametrov konfigurácii je v prílohe v ukážkovom konfiguračnom súbore `impl/data/config.xml`

`--app-scene FILE` – cesta k suboru so scenou (generatorom) – povinny parameter

`--app-seed NUMBER` – nastaví `seed` pre generátor, ak sa nešpecifikuje použije sa 0

5.3 Lokalne spustenie

V pripade spustenia ako normalnej aplikacie staci specifikoat parametre aplikacie. Equalizer si vytvori automaticku konfiguraciu s jednym oknom.
Priklad:

```
./build/bin --app-data ./data/ --app-scene ./data/scenes/simple.xml
```

5.4 Viacej zariadení

Pri použití viacerých zariadení je potrebne najprv nakonfigurovať ich štruktúru pre Equalizer. Ta sa rovi v `.eqc` súboroch. Následne je potrebne spustiť server, klientov a samotnú aplikáciu. Je možné nakonfigurovať Equalizer aby klientov spúšťal automaticky podla potreby. Podrobne informácie sú v dokumentácii [19] pre Equalizer v sekciách I. User Guide – 3. Configuring Equalizer Clusters a 4. Setting up a Visualization Cluster. Konfigurácia použitá v SAGELab je v prílohe.

Jeden s postupov spustenia:

- nakopírovať aplikáciu a dáta na všetky PC
- nakonfigurovať `.eqc` na jednom PC, ktorý bude slúžiť ako server (môže byt aj 6. PC)
- spustiť server cez `eqServer config.eqc`
- spustiť klientov ručne na každom PC (okrem toho, kde bude spustená aplikácia, kedže ona može byt zároveň aj klientom) s:
  `app -eq-listen CONNECTION ...`, kde `CONNECTION` je hostname alebo IP daneho klienta,

- spustiť aplikáciu s `app ...` ak je server na rovnakom PC ako aplikacia, alebo s `app --eq-server SERVER`, kde `SERVER` je hostname alebo IP servera.

Nezáleží na poradí spúšťania klientov a servera, no obe sa musia spustiť pred samotnou aplikáciou. Je možné nakonfigurovať automatické spúšťanie klientov. Klienti automaticky sa vypnú, keď sa ukonči aplikácia, ale pomocou `-r` je možné nastaviť aby zostali naďalej zapnuté. Server môže byť spustený priamo v procese s aplikáciou pomocou `app --eq-config config.eqc`.

Užívateľský vstup sa získava zo všetkých klientov (posiela sa späť do aplikácie). V prípade konfigurácie ako ma SAGELab je možné ovládať tak, ze sa spusti ďalší klient na lokálnom počítači (vyžaduje upraviť konfiguráciu Equalizera) alebo s použitím nástroju na zdieľania ovládania cez sieť.

5.5 Ovládanie

Pohyb kamery sa ovlada pomocou klavesnice a rotacia kamery prostrednictvom mysi. V Equalizeri (pravdepodobne pouzitim starsej kniznice GLX) je chyba, ktora znemoznuje pouzitie tlacidie `SHIFT`, v pripade pouzitia počas pohybovania moze nastat ze sa pohyb nezastavi. Rychlost pohybu je relativna ku nadmorskej vyske.

Klavesnica:

- `WASD` – pohyb dopredu, dozadu, doľava a doprava
- `medzera (space)` - pohyb smerom nahor (od stredu zeme)
- `control (ctrl)` - pohyb smerom nadol (ku stredu zeme)
- `R` - restart – načíta nanovo scénu (generátor) zo súboru a vymaže vyrovnávaciu pamät
- `F` - forward – prepína automaticky pohyb vpred (vodorovný s povrchom zeme – nadmorská výška sa nemení), ostatne ovládanie popritom funguje normálne
- `O` - outline – prepínanie medzi normálnym zobrazením a zobrazovaním čiar (určené pre debug)
- `L` - log – zapnutie/vypnutie zaznamenania informácii o každom snímku do súboru (určené pre testovanie výkonu)
- `page up, page down` – zvýši/zníži `seed` generátora a nanovo načíta scénu

Myš:

- držaním ľavého tlačidla sa otáča kamera

Záver

Cieľom prace bolo zanalyzoval a navrhnúť systém na generovanie a renderovanie planéty s podporou netradičných zariadení s vysokým rozlíšením a na základe toho vytvoriť prototyp. Výsledkom analýzy ukázali, že projektov a štúdii jednotlivých časti tejto problematiky existuje veľké množstvo a v dobrej kvalite, čiže existujú množstvo aplikácii na tvorbu kvalitných terénov s vhodným užívateľským rozhraním a podobne sú na tom systémy na renderovanie na zariadeniach pozostávajúcich z viacerých počítačov s využitím viacerých GPU. No zároveň sa ukázalo, že sa nenašiel projekt, ktorý by všetky tieto časti spájal dokopy a zároveň splňoval ostatne požiadavky alebo sú projekty uz neaktuálne a tým pádom majú problémy s kompatibilitou s novšími technológiami a grafickými kartami

Návrh aplikácie pozostáva primárne z kombinácii myšlienok získaných z existujúcich projektov, rozšírený o niektoré nove princípy. Z tohto sa vytvoril najprv návrh počítajúc s Prolandom ako základom pre prototyp, čo sa neskor ukázalo ako nevhodná cesta, kvôli problémami s kompatibilitou Prolandu s používaným HW. Na základe týchto skúsenosti sa vytvoril alternatívny návrh pozostávajúci na OpenGL v spojení s Eyescale Equalizer na paralelne renderovanie. Systému generovania sa navrhol na baze grafu uzlov a navrhli sa základne uzli určené na generovanie a renderovanie.

Z návrhu sa implementoval prototyp so základnými uzlami na generovanie povrchu planéty a podporou distribuovaného renderovanie. Implementácia je vo veľkej miere v súlade s návrhom, ale v rámci prototypu boli niektoré boli zjednodušené alebo vynechané. Prototyp sa otestoval na zariadení v SAGELab. Testovanie ukázalo, že najviac zaťažuje samotné generovanie a výkon zavisí najmä od konfigurácii generátora a rýchlosti pohyby v pomere ku vzdialenosti od planéty. Ak sa ten pomer zanechá v do rozumnej hodnote (cca max 2-4 podľa zložitosti použitého generátora) dosahuje prototyp na testovacích scénach v priemere 26 FPS. Okrem samotného obsahu generátora ma výrazný vplyv aj faktor delenia scény a veľkosť rozlíšenia generovaných textúr dlaždíc.

Bibliografia

1. FOURNIER, Alain; FUSSELL, Don; CARPENTER, Loren. Computer Rendering of Stochastic Models. *Commun. ACM*. 6. 1982, roč. 25, č. 6, s. 371–384. ISSN 0001-0782. Dostupné z DOI: 10.1145/358523.358553.
2. WIKIPEDIA. *Diamond-square algorithm — Wikipedia, The Free Encyclopedia* [online]. 2018 [cit. 23. 4. 2018]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Diamond-square%5C%20algorithm&oldid=821715646.
3. PERLIN, Ken. An Image Synthesizer. *SIGGRAPH Comput. Graph.* 7. 1985, roč. 19, č. 3, s. 287–296. ISSN 0097-8930. Dostupné z DOI: 10.1145/325165.325247.
4. OLANO, Marc. Modified Noise for Evaluation on Graphics Hardware. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. Los Angeles, California: ACM, 2005, s. 105–110. HWWS '05. ISBN 1-59593-086-8. Dostupné z DOI: 10.1145/1071866.1071883.
5. KDOTJPG. Noise! *Uniblock Dev Blog* [online]. 19. 9. 2014 [cit. 23. 4. 2018]. Dostupné z: http://uniblock.tumblr.com/post/97868843242/noise.
6. fractalterraingeneration - Fractional Brownian Motion. *Google Code Archive* [online]. 16. 12. 2010 [cit. 23. 4. 2018]. Dostupné z: https://code.google.com/archive/p/fractalterraingeneration/wikis/Fractional_Brownian_Motion.wiki.
7. LIVNY, Yotam; KOGAN, Zvi; EL-SANA, Jihad. Seamless Patches for GPU-based Terrain Rendering. *Vis. Comput.* 2. 2009, roč. 25, č. 3, s. 197–208. ISSN 0178-2789. Dostupné z DOI: 10.1007/s00371-008-0214-3.

8. STRUGAR, Filip. Continuous Distance-Dependent Level of Detail for Rendering Heightmaps. *Journal of Graphics, GPU, and Game Tools*. 2009, roč. 14, č. 4, s. 57–74. Dostupné z DOI: 10.1080/2151237X.2009.10129287.
9. ULRICH, T. Rendering massive terrains using chunked level of detail control. *SIGGRAPH Course Notes*. 2002, roč. 3. Dostupné tiež z: http://tulrich.com/geekstuff/sig-notes.pdf.
10. XIAO, Xiangyun; ZHANG, Shuai; YANG, Xubo. Real-time High-quality Surface Rendering for Large Scale Particle-based Fluids. In: *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. San Francisco, California: ACM, 2017, 12:1–12:8. I3D '17. ISBN 978-1-4503-4886-7. Dostupné z DOI: 10.1145/3023368.3023377.
11. CIGNONI, Paolo; GANOVELLI, Fabio; GOBBETTI, Enrico; MARTON, Fabio; PONCHIO, Federico; SCOPIGNO, Roberto. BDAM - Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization. *Comput. Graph. Forum*. 2003, roč. 22, s. 505–514.
12. FRANK LOSASSO, Hugues Hoppe. Terrain rendering using GPU-based geometry clipmaps. *GPU Gems 2*. 3. 2005. Dostupné tiež z: http://hhoppe.com/proj/gpugcm/.
13. CIGNONI, Paolo; GANOVELLI, Fabio; GOBBETTI, Enrico; MARTON, Fabio; PONCHIO, Federico; SCOPIGNO, Roberto. Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM). In: *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*. Washington, DC, USA: IEEE Computer Society, 2003, s. 20–. VIS '03. ISBN 0-7695-2030-8. Dostupné z DOI: 10.1109/VISUAL.2003.1250366.
14. CLASEN, Malte; HEGE, Hans-Christian. Terrain Rendering using Spherical Clipmaps. In: SANTOS, Beatriz Sousa; ERTL, Thomas; JOY, Ken (ed.). *EUROVIS - Eurographics /IEEE VGTC Symposium on Visualization*. The Eurographics Association, 2006. ISBN 3-905673-31-2. ISSN 1727-5296. Dostupné z DOI: 10.2312/VisSym/EuroVis06/091-098.
15. BRUNETON, Eric; BEGAULT, Antoine; PIOLAT, Guillaume. *Proland* [software]. INRIA - LJK (CNRS - Grenoble University), ©2008-2015 [cit. 23. 4. 2018]. Dostupné z: https://proland.inrialpes.fr/.
16. MOLNAR, Steven; COX, Michael; ELLSWORTH, David; FUCHS, Henry. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.* 7. 1994, roč. 14, č. 4, s. 23–32. ISSN 0272-1716. Dostupné z DOI: 10.1109/38.291528.

17. NEAL, Braden; HUNKIN, Paul; MCGREGOR, Antony. Distributed OpenGL Rendering in Network Bandwidth Constrained Environments. In: KUHLEN, Torsten; PAJAROLA, Renato; ZHOU, Kun (ed.). *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2011. ISBN 978-3-905674-32-3. ISSN 1727-348X. Dostupné z DOI: `10.2312/EGPGV/EGPGV11/021-029`.

18. PRISPEVATEIA. *OpenSG* [software]. sourceforge [cit. 22. 4. 2018]. Dostupné z: `https://sourceforge.net/projects/opensg/`.

19. NACHBAUR, Daniel; EILEMANN, Stefan et. al. Eyescale Equalizer. *GitHub repository* [online]. ©2005-2012 [cit. 23. 4. 2018]. Dostupné z: `https://github.com/Eyescale/Equalizer`.

20. NACHBAUR, Daniel; EILEMANN, Stefan et. al. Eyescale Collage. *GitHub repository* [online]. ©2005-2012 [cit. 23. 4. 2018]. Dostupné z: `https://github.com/Eyescale/Equalizer`.

21. BRUNETON, Eric; BEGAULT, Antoine; PIOLAT, Guillaume. *Ork* [online]. INRIA - LJK (CNRS - Grenoble University), ©2008-2015 [cit. 23. 4. 2018]. Dostupné z: `http://ork.gforge.inria.fr/v3.1/index.html`.

22. KEMEN, Brano; HRABCAK, Laco. *Outtera* [software] [cit. 23. 4. 2018]. Dostupné z: `http://www.outerra.com/`.

23. FAIRCLOUGH, Matt. *Terragen* [software]. PLANETSIDE SOFTWARE LLC, ©2016 [cit. 23. 4. 2018]. Dostupné z: `https://planetside.co.uk/`.

24. SCHMITT, Stephen. *World Machine* [software]. World Machine Software, LLC, ©2018 [cit. 23. 4. 2018]. Dostupné z: `https://www.world-machine.com/`.

25. PRISPEVATELIA, Robert Osfield a. *http://www.openscenegraph.org* [software]. ©2002 [cit. 20. 4. 2018]. Dostupné z: `http://www.openscenegraph.org/`.

Dodatok A

Zoznam použitých skratiek

GUI Graphical user interface

IP Internet Protocol

XML Extensible markup language

CPU Central processing unit

CAVE Cave automatic virtual environment

GPU Graphics processing unit

HW Hardware

LOD Level of detail

LRU Least-recently-used

PC Personal computer

GLSL OpenGL Shading Language

SAGE2 Scalable Amplified Group Environment

TCP Transmission Control Protocol

FPS Frames per second

Dodatok B

Obsah priloženého CD

```
readme.txt ............................. popis obsahu CD
impl ................................... projekt s implementáciou
├── src ................................ zdrojové kódy aplikácie
├── lib ................................ použité knižnice
├── test ............................... testovanie výkonu
└── data ............................... súbory používané v programe
    ├── scenes ......................... konfigurácie generátora
    └── equalizer ...................... konfigurácie Equalizera
text ................................... text práce
├── thesis.pdf ......................... text práce vo formáte PDF
└── src ................................ zdrojová forma práce vo formáte LaTeX
```