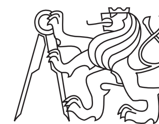
Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Softwarového Inženýrství



Diplomová práce

Efektivní transformace terénních dat s GPU akcelerací

Bc. Marek Manukjan

Vedoucí práce: Mgr. Vladimír Nejedlý

10. ledna 2016

Poděkování

Chtěl bych poděkovat své manželce za trpělivost a podporu při studiu a Vadímíru Nejedlému a společnosti Bohemia Interactive Simulations za možnost realizovat tuto práci v rámci pracovního poměru.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 10. ledna 2016 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2016 Marek Manukjan. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Manukjan, Marek. *Efektivní transformace terénních dat s GPU akcelerací.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

VBS Blue je simulátor renderující celou planetu v reálném čase. Tato práce se zabývá návrhem a implementací softwarové komponenty pro čtení a transformaci trénních dat v tomto simulátoru. Důraz je kladen na dobrou škálovatelnost a efektivitu, neboť velikost zdrojových dat se může pohybovat v řádu stovek gigabytů. Dále se zabýváme rozšiřitelností komponenty o nové datové formáty a akcelerací výpočtů pomocí GPU technologie NVIDIA CUDA.

Klíčová slova GIS, CUDA, mapy, C++, DLL

Abstract

VBS Blue is a simulator that renders the entire planet in real time. This thesis deals with design and implementation of a software component for reading and transformation of terrain data in this simulator. Emphasis is taken for scalability and effectivity, since the size of source data can be hundreds of gigabytes. We also deal with extensibility of the component with new data formats and GPU acceleration using NVIDIA CUDA.

Keywords GIS, CUDA, maps, C++, DLL

Obsah

Úvod **1**
VBS . . . 1
VBS Blue . . . 1

1 Cíl práce **3**
1.1 Výstup práce . . . 3
1.2 Požadavky na výstup . . . 3

2 Analýza a návrh **9**
2.1 Analýza existujících řešení . . . 9
2.2 Cílová projekce . . . 12
2.3 Převod mezi geodetickými a krychlovými souřadnicemi . . . 20
2.4 Architektura . . . 24
2.5 GPU Akcelerace . . . 29
2.6 Slučování dat . . . 30

3 Realizace **33**
3.1 GlobeTextureManager . . . 33
3.2 DataManager . . . 39
3.3 Pluginy . . . 40
3.4 GDAL Plugin . . . 45
3.5 Stav implementace . . . 54

4 Výsledky **55**
4.1 Rychlost načítání . . . 55
4.2 Rychlost binarizace . . . 56
4.3 Efektivita kernelů . . . 57

Závěr **61**

Literatura **63**

A Seznam použitých zkratek **65**

B Obsah přiloženého CD **67**

Seznam obrázků

0.1 Ukázka scény VBS Blue zblízka a z dálky 2

2.1 Krychlová projekce v enginu Outerra 10
2.2 Deformace textury na pólu koule v cylindrické projekci 12
2.3 Horní strana krychle v cube mappingu (vlevo) a Quadrilateralized spherical cube (vpravo) 13
2.4 Základní krychle pro vykreslování VBS Blue 14
2.5 Transformace bodu na elipsoid (2D) 15
2.6 Krychle transformovaná na elipsoid 17
2.7 Problém s geocentrickou transformací vertexů 18
2.8 Geodetická transformace vertexů 19
2.9 Číslování a orientace stran krychle 20
2.10 Transformace krychlových souřadnic do ekvidistantní válcové projekce 22
2.11 Transformace geografických souřadnic do krychlových 24
2.12 Tok dat v komponentě pro načítání map 25
2.13 Algoritmus na slučování více mapových zdrojů 31

3.1 Přechody stavu DataManagerJobu 40
3.2 Transformace krychlových souřadnic do sinusoidové projekce . . . 46

Seznam tabulek

4.1 Hardwarová konfigurace testovacího PC 55
4.2 Doba zpracování menšího datasetu 56
4.3 Doba zpracování většího datasetu 57
4.4 Vytížení GPU CUDA kernely 57

Úvod

VBS

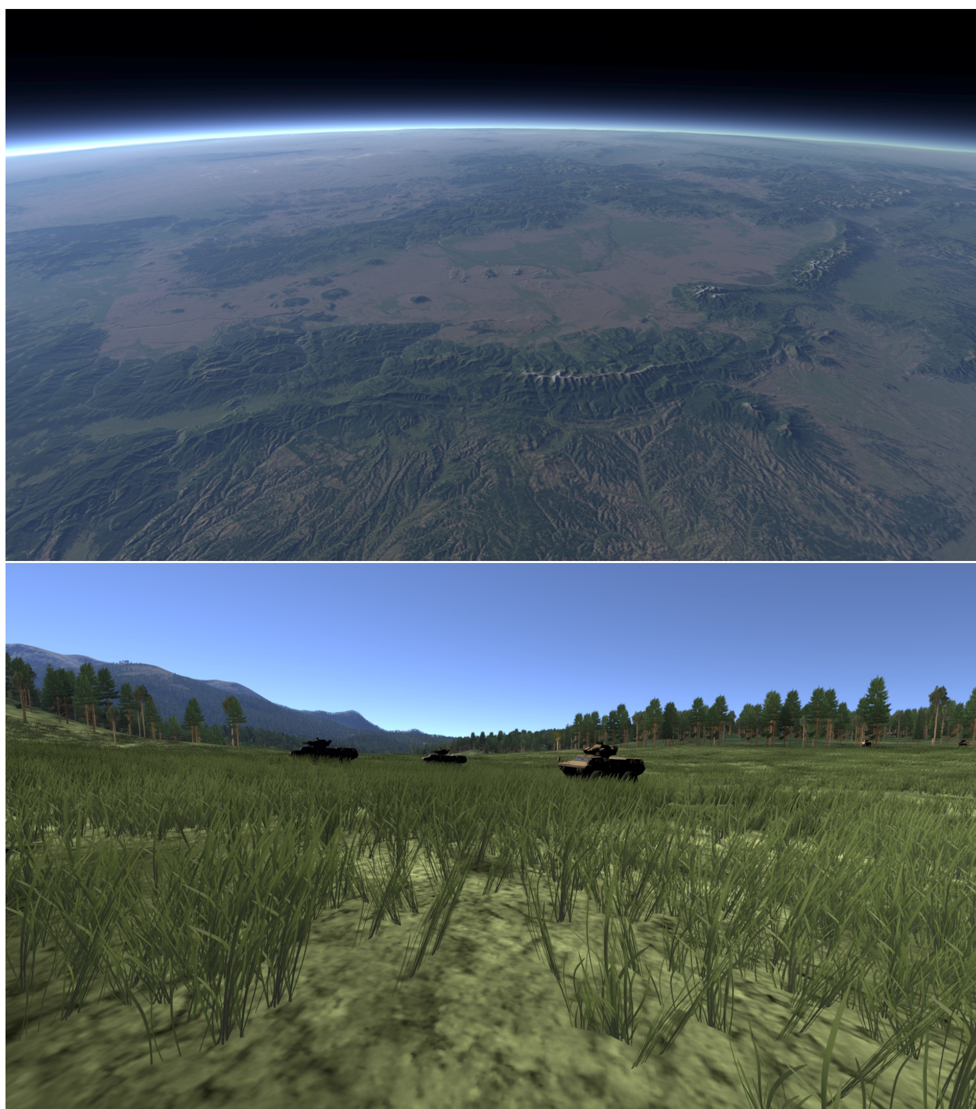
Virtual Battlespace (dále jen VBS) je vojenský simulátor od společnosti Bohemia Interactive Simulations. Původně vznikl jako modifikace české hry ArmA. Nyní se sice jedná o zcela samostatný produkt, stále ale vychází ze zdrojového kódu hry ArmA II.

VBS podporuje integraci s množstvím hardwaru, což z něj dělá univerzální simulátor. Primárně je zaměřen na simulaci vojáka pohledem z první osoby, kdy se ovládá jako standardní akční hra. Hráč má možnost se společně s ostatními volně pohybovat po geospecifické mapě, používat zbraně, vozidla, letadla, helikoptéry, lodě, speciální výstroj atd. VBS však lze integrovat i do speciálních kokpitů simulujících konkrétní vozidlo, čehož se využívá hlavně u simulátorů letadel.

Využití VBS jako leteckého simulátoru vyžaduje patřičně velkou herní mapu, což je jedno z hlavních vylepšení VBS oproti původnímu hernímu enginu. VBS3 dokáže z disku streamovat obrovské mapy, které mají desítky GB. Jednou z největších map pro VBS je mapa východního pobřeží USA, která má rozlohu 2200×2200 km. Při takovém rozměru mapy přestávají být problémem pouze dostupné systémové zdroje – důležitou otázkou je například zakřivení země, se kterým původní engine nepočítal. Zákazníci by samozřejmě uvítali ještě větší rozlohu pro simulaci, a tak je tedy logicky dalším krokem vývoje VBS pojmout celou planetu. Právě tím se zabývá projekt VBS Blue.

VBS Blue

VBS Blue[1] je nový projekt od Bohemia Interactive Simulations, který je v době psaní této práce aktivně ve vývoji. Cílem projektu je fotorealisticky vykreslovat Zemi nejen z dálky, ale v každém měřítku od několika centimetrů nad



Obrázek 0.1: Ukázka scény VBS Blue zblízka a z dálky

povrchem, až po pohled z Měsíce. To vše má probíhat v reálném čase bez přerušení pro načtení dat, kterých pochopitelně bude nemalé množství.

Tato práce se bude zabývat právě návrhem a počáteční implementací softwarové komponenty, která bude provádět načítání terénních dat pro VBS Blue.

[1]Název byl původně VBS Blue Marble, podle názvu prvního barevného snímku Země pořízeného z vesmíru. Kvůli problémům s autorskými právy byl však přejmenován pouze na VBS Blue.

Kapitola 1

Cíl práce

1.1 Výstup práce

Výstupem této práce by měla být logicky ohraničená softwarová komponenta, která bude součástí programu VBS Blue. Tato komponenta bude mít na starost vše od načítání zdrojových dat přes jejich převod do optimálního formátu, jejich ukládání, načítání z optimálního formátu až po jejich předání vykreslovacímu enginu.

1.2 Požadavky na výstup

Zpracovávání mapových dat bývá hlavní slabinou podobných simulačních produktů. Vytvořit geospecifický terén z běžně dostupných GIS dat je například v případě simulátoru VBS3 natolik náročný proces, že se jím zabývají specializované společnosti, od kterých si zákazníci musí nechat simulační mapu vyrobit. To samozřejmě stojí nemalé peníze, přestože v principu nejde o nic složitého.

Kvůli tomu mají společnosti, které projekt VBS Blue financují, jisté funkční i nefunkční požadavky, které musí výsledný produkt splňovat. Velká část z nich se týká právě komponenty pro přípravu a načítání dat, aby bylo vytváření geospecifického[2] virtuálního prostředí co nejjednodušší a nejefektivnější.

1.2.1 Modularita

VBS Blue by nemělo být omezeno na jeden konkrétní formát vstupních dat, ani na sadu GIS formátů které jsou standardně používány. Uživatelé by měli mít možnost rozšířit základní sadu podporovaných formátů mapových podkladů o vlastní. Nemyslí se tím pouze formát souboru, ale i projekce mapy[3]. Nelze

[2]Geospecifické prostředí je takové, které odpovídá skutečnému místu na Zemi. Jeho opakem je geotypické prostředí, které pouze nějaké místo připomíná, ale není jeho kopií.

dokonce ani předpokládat, že data se budou načítat ze souboru. K získání mapových podkladů by mělo být možné využít například i databázi.

1.2.2 Načítání map přes síť

Dalším požadavkem je možnost načítat zdrojová data nejen z místního disku, ale i ze vzdálených serverů s mapovými zdroji. Zdrojové mapy mohou mít typicky desítky či stovky GB a mít takové množství dat rozkopírované na každém počítači[4] není ekonomicky výhodné. Pokud jsou simulační klienti propojeni sítí, mohou být mapové zdroje uloženy na centrálním serveru s rychlým a velkým úložištěm. Nejen že zákazník ušetří výdaje za hardware, ale také čas, který by trvalo nahrát vše na každý počítač.

1.2.3 Procedurálně generované mapy

Kromě zobrazování geospecifických map by měl být systém VBS Blue také schopný za chodu generovat geotypické mapy v případě, že nejsou dostupná žádná reálná zdrojová data.

Geotypická mapa je taková mapa, která sice neodpovídá žádnému skutečnému místu na Zemi, ale na první pohled se mu podobá, protože využívá podobné rysy krajiny jako její předloha. Velikost předlohy je libovolná. Geotypická mapa může být například jak mapa určitého pohoří, kde by byly podobné výšky a vegetace jako v předloze, tak i celé planety, u které by generátor vytvořil celou planetu podobnou Zemi.

1.2.4 Korelace

Hlavním způsobem použití VBS Blue bude z počátku pouze zobrazování grafiky, takzvaný Image Generator (IG). O samotnou simulaci objektů se v takovém zapojení stará jiný program, který VBS Blue zasílá informace o stavu objektů. VBS Blue poté scénu pouze vykresluje. Aby takové zapojení fungovalo, musí mít jak simulační server, tak i renderer naprosto stejné informace o pozici objektů vůči terénu a to i přes to, že systémy pracují v jiných souřadnicových systémech. Pokud by simulace měla jiná data než IG, mohlo by se stát, že vozidla budou levitovat ve vzduchu nebo naopak budou zanořena pod zem.

VBS Blue používá pro renderování terénu svůj vlastní souřadnicový systém, ve kterém jsou mapová data transformována na krychli. Po převodu dat

[3] Mapová projekce, také nazývaná mapové zobrazení, popisuje způsob, kterým se trojrozměrný povrch planety promítá do jiného prostoru, typicky $(\mathbb{R}, \mathbb{R}, \mathbb{R}) \to (\mathbb{R}, \mathbb{R})$ pro snadné zobrazení na papíře. Jsou však možná i jiná zobrazení, v této práci se budeme často zabývat krychlovým zobrazením $(\mathbb{R}, \mathbb{R}, \mathbb{R}) \to (\mathbb{Z}, \mathbb{R}, \mathbb{R})$.

[4] VBS3 se typicky istaluje hromadně na počítače do učebny propojené místní sítí. Podobné použití lze předpokládat i pro VBS Blue.

do tohoto systému nesmí být data pozměněna tak, aby byl výsledek rozeznatelný od originálu.

1.2.5 Převod dat v reálném čase

U simulátorů bývá obvyklé, že zdrojové mapové podklady se převádí do nativního formátu simulátoru. V nativním formátu jsou data uložena tak, aby byla co nejoptimálnější pro načtení do simulátoru. Zákazníci nutnost tohoto převodu chápou, přesto by ale uvítali možnost načítat mapy přímo ze zdrojových dat s tím, že načítání může být zpomalené a ne zcela přesné.

Této vlastnosti by se využívalo při vývoji map pro rychlé zobrazení náhledu výsledku v VBS Blue. Je také nezbytnou podmínkou pro čtení zdrojových dat z mapového serveru, neboť si nemůžeme dovolit ze serveru stáhnout všechna data a převést je hromadně.

1.2.6 Rychlost načítání

VBS Blue nemá na rozdíl od starších simulátorů ohraničenou mapu. Kamera se může přesouvat kdykoli na jakékoli místo na Zemi nebo ve vesmíru. Program se v tu chvíli nesmí pro načítání dat v dané oblasti zastavit. Operační paměť (natož videopaměť) ale není schopná veškerá data pro celou planetu pojmout. Renderer v takovém případě zobrazí nejlepší data, která má v danou chvíli k dispozici. Do scény se načítaná data přidávají postupně podle toho, kdy se jaký zdroj zrovna načte. Stejný mechanismus zajišťuje postupné načítání dat i v případě, že se kamera pohybuje plynule jedním směrem.

Požadavek zákazníků je, aby VBS Blue dokázalo načítat mapu bez ztráty detailů při pohybu kamery rychlostí Mach 2, tedy 680 m/s. V požadavcích ovšem chybí údaj o požadované letové výšce, což je zásadní nedostatek. Ve výškách, kdy je kamera blíže k zemi, je potřeba načítat podstatně vyšší množství dat pro zachování plného detailu scény. Byla tedy zvolena přiměřená výška 1 km.

1.2.7 Další požadavky

Některé požadavky na komponentu pro načítání dat nejsou dány přímo zákazníkem, ale architekturou systému VBS Blue.

1.2.7.1 GPU Akcelerace

Při hromadném převodu zdrojových dat do nativního formátu simulátoru[5] se nic nerenderuje a GPU je tedy nevyužito. Hromadný převod mapových dat je přitom výpočetně poměrně náročný proces, kde se každý pixel dá zpracovávat

nezávisle. Při návrhu VBS Blue bylo tedy rozhodnuto, že se pro převod dat bude používat GPU akcelerace za pomoci technologie NVIDIA CUDA.

CUDA byla zvolena proto, že vývoj pod touto technologií je o něco snazší než MS AMP, OpenCL, nebo DirectCompute. Pro první prototyp, který poběží pouze na počítačích vývojářů, je nemultiplatformnost technologie CUDA přijatelnou nevýhodou. Pokud se ukáže, že je GPU akcelerace pro převod dat užitečná, bude pro produkční prostředí CUDA kód přeimplementován do vhodné multiplatformní náhrady. Bylo by tedy dobré, aby CUDA kód nebyl příliš provázán s okolím, protože později bude potřeba jej nahradit jinou technologií.

1.2.7.2 Vícevláknová bezpečnost

VBS Blue silně využívá multithreading. Hlavní vlákno slouží pro simulaci a hlavní aplikační smyčku, příprava vykreslování probíhá paralelně (je však inicializována synchronně s hlavním vláknem). Vykreslování běží v samostatném vlákně, načítání z disku také probíhá ve speciálním vlákně. Kterékoli vlákno může přistupovat v jakýkoli moment k mapovým datům a vlákna se nesmí navzájem blokovat. Implementace musí být tedy naprosto bezpečná pro přístup z více vláken najednou.

1.2.7.3 Asynchronní načítání dat vůči hlavnímu vláknu

VBS Blue má ideálně udržovat vykreslovací frekvenci alespoň 120 snímků za sekundu, což znamená přibližně 8,3 ms na zpracování jednoho snímku. Jak hlavní, tak renderovací vlákno běží neustále a není možné je pozastavit na dobu, kterou by trvalo načíst soubor s daty z disku. Při vysokém zatížení pevného disku se přitom pohybuje jen samotný seek time v desítkách milisekund, a to je ještě potřeba mít čas na další operace, které je nutné provést každý snímek. Pokud navíc nejsou data připravena v optimálním formátu, jak specifikuje požadavek na převod dat v reálném čase, nebo jsou dokonce umístěna na vzdáleném serveru, může načítání dat trvat i celé vteřiny.

Aby se předešlo zamrznutí hlavního vlákna po dobu načítání, musí se vůči němu načítat data asynchronně v samostatném vláknu. Pokud dorazí žádost o mapová data, která nejsou okamžitě dostupná, je vrácen status, že data nejsou připravena a na pozadí se spustí jejich nahrávání. Žádost přijde opět v některém z dalších snímků a pokud tou dobou už budou data připravena, komponenta je vrátí společně se statusem, že operace proběhla úspěšně.

[5] Interně se tomuto procesu poněkud nepřesně říká „binarizace“. Tato praktika je široce používaná, neexistuje pro ni však žádný obecně uznávaný termín. Například NVIDIA pro ni používá ve své knihovně PhysX výraz „cooking“.[1]

1.2.7.4 Cachování načtených dat

V současné době platí, že videopaměť bývá podstatně menší než systémová operační paměť. Ve VBS Blue je videopaměť využita na maximum. Viditelná scéna v zorném poli kamery zabírá stovky MB. Kvůli tomu textury často vypadávají z videopaměti a jsou nahrazovány dalšími z nového směru, do kterého se kamera dívá. Pokud se však vrátí na původní místo, musí se textury načíst znovu. Kvůli tomu by bylo vhodné implementovat mechanismus, který by naposledy použité textury uchovával v systémové paměti, odkud se můžou velice rychle opět nahrát na grafickou kartu.

Kapitola 2

Analýza a návrh

2.1 Analýza existujících řešení

Vykreslováním planet se zabývá několik existujících programů, většina kvalitních bohužel není open-source. I tak se z nich dá zjistit mnoho užitečných informací, ať už z jejich webových stránek, adresářové struktury programu, nebo ze způsobu běhu programu.

2.1.1 Outerra engine

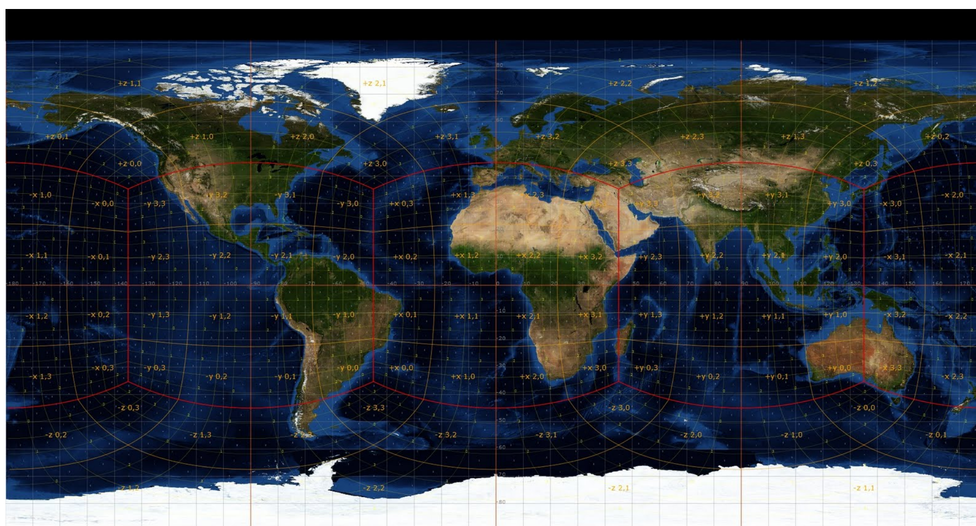
Slovenský Outerra engine je nejpokročilejším rendererem planet, který je momentálně k dispozici. Ve vývoji je už od roku 2008. Je zaměřen na co nejvěrnější grafické podání, k čemuž využívá OpenGL 3.3. Právě tento engine je použit v simulátoru TitanIM, který je VBS Blue momentálně největším konkurentem.

Přestože je Outerra vyvíjena v úzké spolupráci s komunitou svých uživatelů, není open-source a nejlepším zdrojem informací o ní je blog vývojářů[6]. Právě zde je k dispozici článek, ze kterého lze v omezeném rozsahu zjistit, jakou v tomto enginu používají strukturu dat a jak je zpracovávají.

V enginu Outerra používají je pro vykreslování planety použita také jistá forma krychlové projekce. O jakou variantu této projekce se konkrétně jedná sice není specifikováno, ale velice pravděpodobně se jedná o standardní krychlovou projekci, která vznikne protažením paprsku ze středu elipsoidu přes bod jeho povrchu na povrch opsané krychle okolo elipsoidu. Jak tato projekce vypadá, je znázorněno na obrázku 2.1.

Outerra engine také podporuje streamování textur planety přes internet. Planetární data jsou rozdělena do několika tisíců archivů s texturami. Každý archiv má přibližně deset až patnáct MB. Streamování textury probíhá tak, že se celý archiv automaticky stáhne na disk, odkud se poté použije jako standardní soubor dodávaný již při instalaci. Nevýhodou tohoto přístupu je

[6] `http://outerra.blogspot.cz`



Obrázek 2.1: Krychlová projekce v enginu Outerra

nízká flexibilita. Poněkud nepříjemné také je, že textury planety se dlouho neobjevují, dokud se nestáhne celý archiv s texturami. Po jeho stažení se načtou všechna data rychle najednou. Na druhou stranu to pravděpodobně redukuje overhead, který by vznikal individuálním stahováním každé textury.

2.1.1.1 Převod dat v enginu Outerra

Nástroj pro převod dat do interního formátu enginu je přímo jeho součástí. Pro samotnou transformaci dat využívá GPU akceleraci, místo standardních API pro výpočty na GPU jako je např. CUDA nebo OpenCL však využívá přímo GLSL shadery. Tato možnost byla údajně zvolena kvůli lepší integraci se zbytkem enginu, který OpenGL už využívá pro vykreslování.[2]

Vstupní data mohou být pouze v různých raw formátech podporovaných OpenGL a aktuálně používanou grafickou kartou. Dále musí být rozdělena na dlaždicově uspořádané soubory o přiměřené velikosti. S různými vlastnostmi souborů z různých zdrojů se nástroj vypořádává tak, že ke každému zdroji je požadován konfigurační soubor. Tento soubor napsaný v javascriptu popisuje nejen vlastnosti jako např. formát a rozlišení vstupních dlaždic, ale také funkci pro převod z indexu dlaždice na název souboru, ve kterém je uložena. Jedinou podporovanou projekcí zdrojových dat je ekvidistantní válcová projekce[7].

[7] Ekvidistantní válcová projekce je jednou z nejjednodušších a nejstarších mapových projekcí. Poměr mapy je 2:1, kde v ose X je lineárně rozložená zeměpisná šířka v rozsahu $(-180°, 180°)$ a v ose Y zeměpisná délka s rozsahem $(-90°, 90°)$.

Pro průchod quadtree se při převodu dat používá Hilbertova křivka. Její hlavní výhodou oproti tradičnímu průchodu stromem je, že její průchod po povrchu krychle je spojitý, což omezuje výpadky cachí, a má tedy pozitivní vliv na výkon.

2.1.2 Space Engine

Space Engine je projekt ruského programátora Vladimira Romanyuka, který si klade za cíl simulaci a vykreslování celého vesmíru v reálném čase. Rozměrem simulované oblasti sice přesahuje jakýkoliv jiný podobný produkt, v kvalitě vykreslování planet však poněkud zaostává. Space Engine využívá krychlové projekce pro vykreslování planet – stejně jako Outerra.

2.1.2.1 Nástroj na reprojekci dat

Autor poskytuje pro Space Engine nástroj na převod mapových podkladů do krychlové projekce. Je tvořen třemi samostatnými programy - CubeMap, Optimizer a Glue.

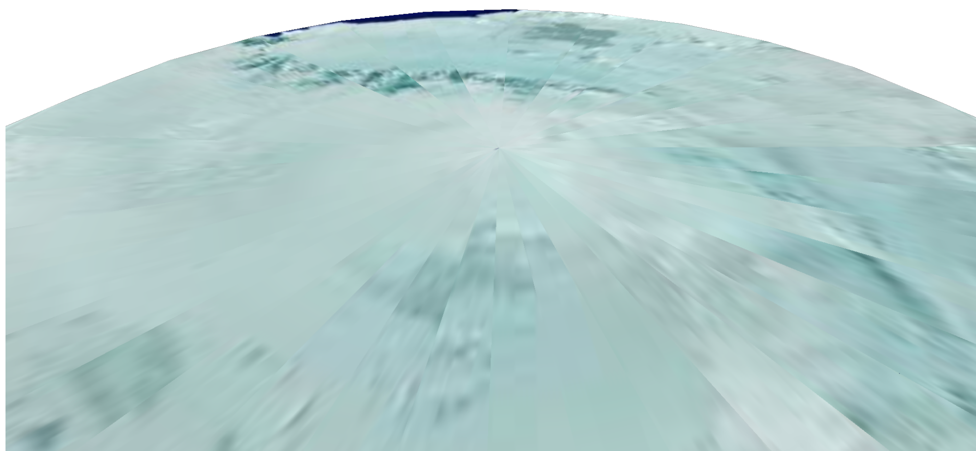
Program CubeMap provádí samotnou reprojekci dat ze zdrojového souboru. Neakceptuje však žádný standardní formát GIS dat. Vstupní data je potřeba mít v nekomprimovaném surovém formátu tak, jaký je binární otisk obrázku mapy v paměti RAM. Podporovány jsou jen osmi nebo šestnáctibitové celočíselné formáty po jednom, třech, nebo čtyřech kanálech v little nebo big endian. Podpora pro data s plovoucí čárkou chybí, což je velkým omezením zejména pro výškové mapy, které tak mohou mít vertikální přesnost pouze na celé metry. Zdrojová data program akceptuje pouze v ekvidistantní cylidrické projekci.

Nástroj Optimizer detekuje a maže prázdné soubory generované programem CubeMap. Proč CubeMap vůbec takové soubory vytváří však autor neuvádí. Posledním nástrojem je Glue: výsledné soubory zabalí do jednoho archivu, který je poté možné načíst ve Space Enginu. Rozdělit nástroj na tři samostatné programy je však poněkud nepraktické.

2.1.3 Google Earth

Veřejnosti nejznámější program pro renderování planet je Google Earth. Paradoxně je také jedním z mála programů, které používají pro zobrazení planety cylindrické UV mapování, konkrétně z Webové Mercatorovy[8] projekce. To je velmi pravděpodobně dané tím, že zdrojem dat jsou mu Google Maps, které tuto projekci používají. Tato projekce se na krychli v reálném čase transformuje problematicky.

[8] Webová Mercatorova projekce je odvozena od Mercatorova zobrazení, ve kterém se tvar Země zjedodušeně uvažuje jako koule. S Mercatorovým zobrazením má také společné to, že kvůli extrémnímu zkreslení na pólech nedokáže typicky zobrazit oblasti ležící za 85° severní a jižní šířky.



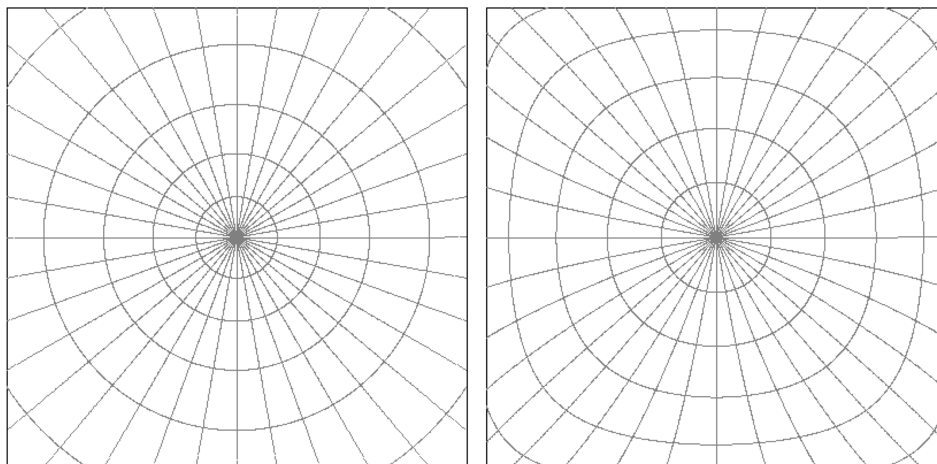
Obrázek 2.2: Deformace textury na pólu koule v cylindrické projekci

Nevýhoda UV mapování je v Google Earth na první pohled viditelná při zobrazení jednoho z pólů. Textura povrchu je zde velmi zkreslená a načítání trvá dlouho, protože ve výřezu kamery jsou viditelné stovky segmentů, které jsou smrštěné do jediného bodu. Pro každý segment se přitom načítá textura v plném rozlišení.

2.2 Cílová projekce

Populární metodou pro vykreslování planet a podobných těles je mapování textury na elipsoid pomocí krychlové projekce. Tato metoda byla použita i ve VBS Blue. Je výpočetně i implementačně složitější než cylindrická projekce, netrpí však jejími nedostatky. Problém běžně používaného UV mapování koulí/elipsoidů, na které je textura mapována pomocí cylindrické projekce, totiž tkví v tom, že poblíž pólů je textura velmi deformovaná. Tento efekt je možné pozorovat na obrázku 2.2.

Projekce, do které mají být zdrojová data převedena, je definována tím, jakým způsobem je mapována textura na elipsoid ve VBS Blue. K tomu byla zvolena technika zvaná „cube mapping“. Spočívá v tom, že se body v prostoru promítnou na povrch vepsané krychle ve směru do jejího středu. Výsledkem je šest čtvercových textur, každá odpovídá jedné straně krychle. Obraz v texturách je mírně zkreslen, zdaleka však ne tak jako v cylindrické projekci. Ještě menší zkreslení než cube mapping poskytuje projekce zvaná „Quadrilateralized spherical cube“[3], která byla vytvořena v NASA pro pro-



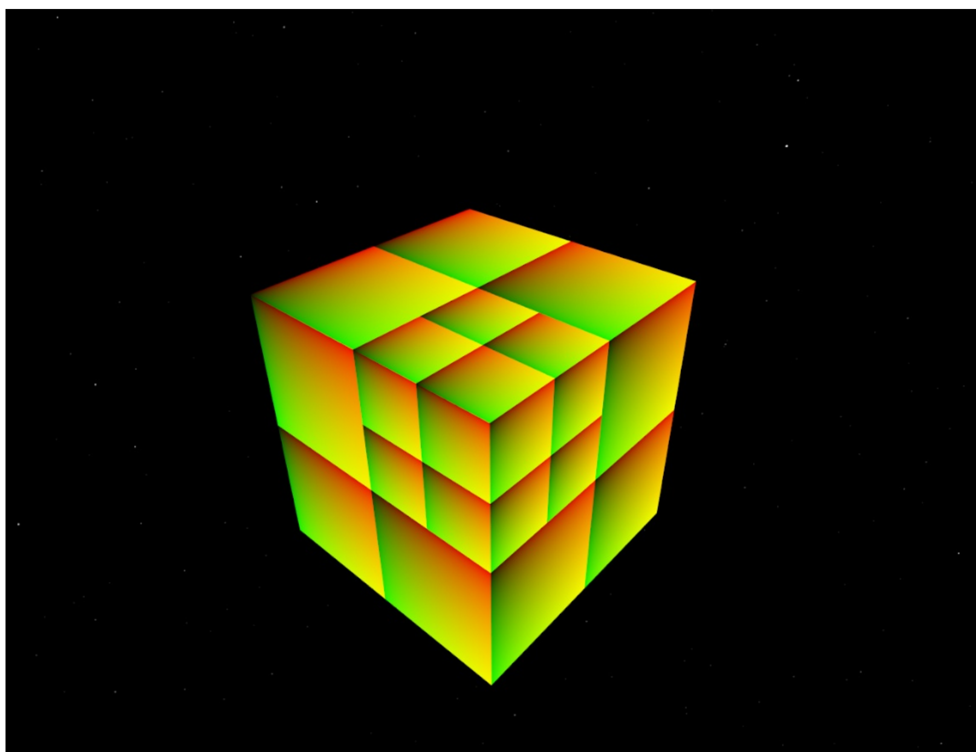
Obrázek 2.3: Horní strana krychle v cube mappingu (vlevo) a Quadrilateralized spherical cube (vpravo)

ject Cosmic Background Explorer. Quadrilateralized spherical cube v podstatě funguje podobně jako cube mapping, na který se aplikuje ještě jedna transformace, která zredukuje zkreslení v rámci textury strany krychle. Tato transformace je však výpočetně poměrně náročná a při inverzní transformaci využívá aproximaci Taylorovým polynomem dvanáctého stupně. Nepřesnost při použití na Zemi se pak i v double precision pohybuje řádově ve stovkách metrů[4]. To je hlavním důvodem, proč je tato projekce naprosto nevhodná pro použití ve VBS Blue.

Základem krychlové projekce je krychle vepsaná do elipsoidu. Každá její strana je složená ze segmentů, kde na každý segment připadá jedna textura každého typu (výškové pole, mapa typu povrchu, mapa biomů...), nebo její výřez. Segmenty jsou hierarchicky uspořádány do quad-tree struktury. Čím je segment ve stromu hlouběji, tím menší plochu pokrývá a tím detailnější povrch reprezentuje při stejném počtu vertexů a texelů na segment. Maximální hloubka stromu ve VBS Blue je 19 úrovní. To, z jaké hloubky se segmenty budou kreslit, je dáno jejich velikostí na obrazovce. Tím je zajištěna rovnoměrná hustota vertexů a texelů na celé obrazovce, přestože jsou některé segmenty mnohem vzdálenější než jiné. Toto schéma znázorňuje obrázek 2.4.

Každý segment se renderuje identickým vertex a index bufferem. Ten je tvořen pravidelnou mřížkou vertexů, v aktuálním nastavení konkrétně 33x33 vertexů. Největší vzdálenost mezi jednotlivými vertexy na maximální úrovni detailů lze tedy spočítat vzorcem

$$2\pi r/4/2^{maxLOD}/vertexCount, \tag{2.1}$$



Obrázek 2.4: Základní krychle pro vykreslování VBS Blue

kde r je poloměr Země. V hlavní ose a čtyřmi se dělí protože po obvodu má krychle čtyři strany. Pokud dosadíme poloměr Země ze standardu WGS84 (6378137 m), vyjde nám 57,9 cm, což je dostatečná jemnost pro věrohodnou reprezentaci zemského povrchu.

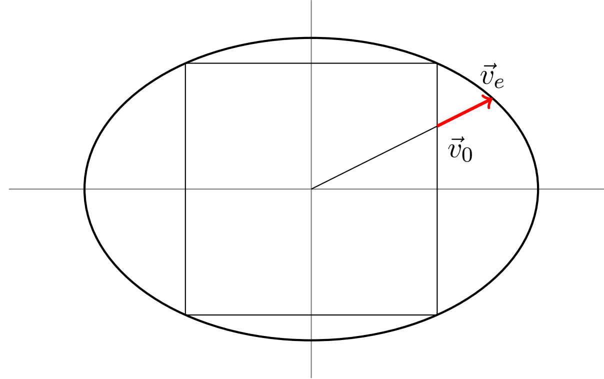
2.2.1 Transformace vertexu na elipsoid

Aby se docílilo oblého tvaru planety, jsou jednotlivé vertexy segmentů posunuty směrem od středu planety na její referenční elipsoid (viz 2.5). Pro Zemi se jedná o referenční elipsoid WGS84 s hlavním poloměrem $6378137,0$ m a sekundárním $6356752,314245$ m. Vzorec pro posun vyplývá z matematického vyjádření elipsoidu:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \tag{2.2}$$

Výchozí pozice vertexu je na povrchu krychle vepsané do tohoto elipsoidu. Jeho pozici zadefinujeme jako:

$$\vec{v}_0 = \{x_0, y_0, z_0\} \tag{2.3}$$



Obrázek 2.5: Transformace bodu na elipsoid (2D)

Vertex chceme posunout na povrch elipsoidu. Výslednou pozici $\vec{v}_e$ získáme posunem $\vec{v}_0$ ve směru $\hat{n}$ o vzdálenost t, zapsané tedy jako

$$\vec{v}_e = \vec{v}_0 + \hat{n} * t \tag{2.4}$$

Pro jednoduchost zde uvažujeme, že střed planety je v bodě $\{0, 0, 0\}$. V takovém případě je směr $\hat{n}$, kterým se bude vertex posouvat, normalizovanou pozicí vertexu samotného. Normalizovaný vektor $\hat{v}$ je takový, pro který platí, že $||\vec{v}|| = 1$ a zároveň $\hat{v} = \vec{v} * d$. Po dosazení do rovnice dostáváme následující vzorec:

$$\vec{v}_e = \vec{v}_0 + \hat{v}_0 * d * t \tag{2.5}$$

$$\vec{v}_e = \vec{v}_0 * (d * t + 1) \tag{2.6}$$

Vzhledem k tomu, že takovým posunem v podstatě pouze měníme velikost vektoru $\vec{v}_0$, lze provést substituci $u = d * t + 1$, čímž získáme jednodušší formu vzorce:

$$\vec{v}_e = \vec{v}_0 * u \tag{2.7}$$

Teď již můžeme dosadit do vzorce pro elipsoid na získání průsečíku:

$$\frac{(x_0 * u)^2}{a^2} + \frac{(y_0 * u)^2}{b^2} + \frac{(z_0 * u)^2}{c^2} = 1 \tag{2.8}$$

$$\frac{x_0^2 * u^2}{a^2} + \frac{y_0^2 * u^2}{b^2} + \frac{z_0^2 * u^2}{c^2} = 1 \tag{2.9}$$

$$u^2 * (\frac{x_0^2}{a^2} + \frac{y_0^2}{b^2} + \frac{z_0^2}{c^2}) = 1 \tag{2.10}$$

$$u = \frac{1}{\sqrt{\frac{x_0^2}{a^2} + \frac{y_0^2}{b^2} + \frac{z_0^2}{c^2}}} \tag{2.11}$$

Výpočet lze ještě optimalizovat pro menší počet instrukcí a vyšší přesnost kvůli velkým číslům takto:

$$u = \frac{1}{\sqrt{(\frac{x_0}{a})^2 + (\frac{y_0}{b})^2 + (\frac{z_0}{c})^2}} \tag{2.12}$$

Po přepsání do C++ dostáváme relativně jednoduchou funkci na transformaci bodu z krychle na elipsoid, která zaoblí tvar Země, jak je vidět na obrázku 2.6:

```
Vector3D Ellipsoid::CubeToEllipsoid(Vector3D cubePos)
{
  double u = 1 / sqrt(
    cubePos.X() * cubePos.X()) / (_radii.X() * _radii.X()) +
    cubePos.Y() * cubePos.Y()) / (_radii.Y() * _radii.Y()) +
    cubePos.Z() * cubePos.Z()) / (_radii.Z() * _radii.Z()));
  return cubePos * u;
}
```

2.2.2 Transformace vertexu z elipsoidu na povrch terénu

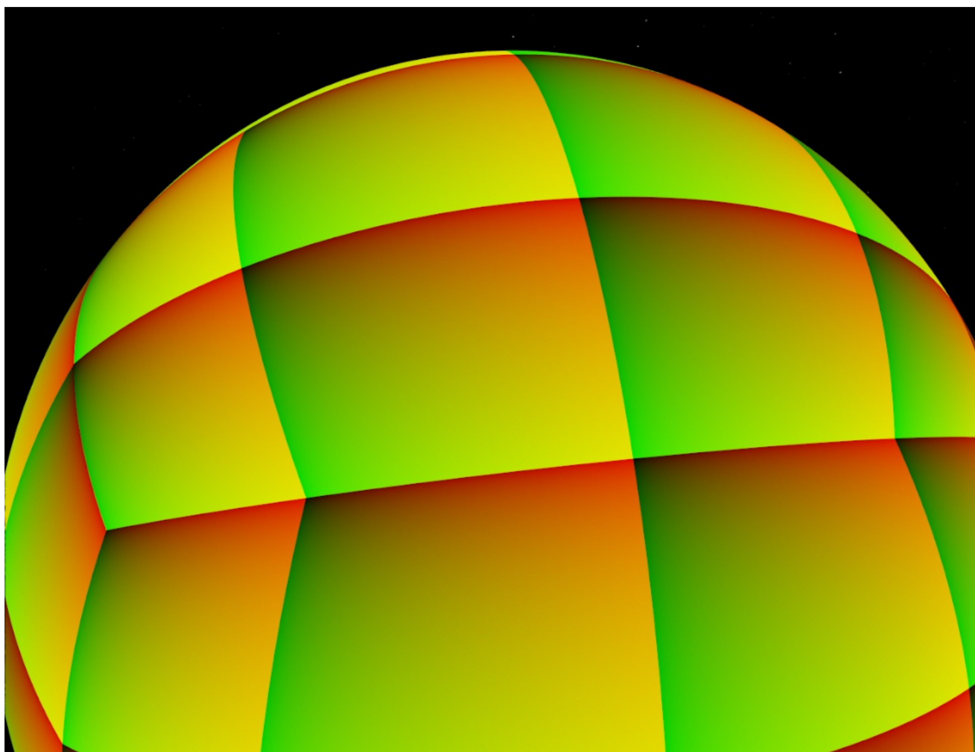
Když teď víme, jak z krychle získat hladký elipsoid, je potřeba jednotlivé vertexy posunout na zemský povrch podle jejich výšky nad referenčním elipsoidem. Způsob, jak toho dosáhnout, je závislý na vstupních datech. Pokud je vstupní výškové pole geocentrické, musí být vertex posunut podél geocentrické normály (směrem od středu planety). Pokud jsou data geodetická, musí se vertex posunout podél geodetické normály, což je normála povrchu elipsoidu. Ta se od geocentrické normály liší, pokud nemá elipsoid stejnou velikost ve všech osách.

2.2.2.1 Geocentrický přístup

Pro renderování byl ve VBS Blue zprvu zvolen přístup pomocí geocentrické normály, protože je výpočetně jednodušší.

$$\vec{v}_s = \vec{v}_e + \hat{v}_e * h \tag{2.13}$$

```
Vector3D Globe::EllipsoidToSurface(Vector3D ellipsoidPos, double
    height)
```



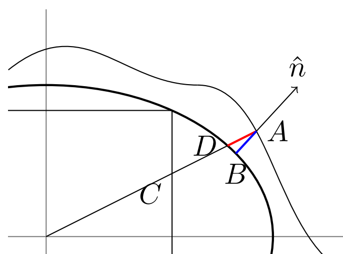
Obrázek 2.6: Krychle transformovaná na elipsoid

```
{
   return ellipsoidPos + ellipsoidPos.Normalized() * height;
}

Vector3D Vector3D::Normalized()
{
   double sizeInv = InvSqrt(_x * _x + _y * _y + _z * _z);
   return *this * sizeInv;
}
```

U tohoto přístupu se však brzy projevilo několik zásadních nedostatků. První, který je sice nepříjemný, ale stále řešitelný, je fakt, že většina vstupních výškových dat používá geodetickou projekci, zatímco renderer vertexy posouvá geocentricky.

Problém je znázorněn na obrázku 2.7. Chceme zjistit výšku, kterou máme uložit na krychli do bodu C. Ta je znázorněna červeně. Potřebujeme tedy určit vzdálenost mezi body D a A. Pozici bodu D lze získat postupem uvedeným výše z bodu C, který máme definován jako součást zadání. Pozici bodu A však zatím neznáme, závisí na výškovém poli. Výšky však v paměti nemáme uloženy



Obrázek 2.7: Problém s geocentrickou transformací vertexů

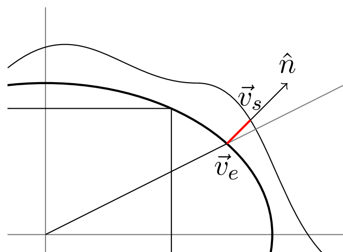
v podobě křivky, jak je znázorněno na obrázku, ale v podobě textury, tedy v podstatě 2D polem. Hodnotami jsou však výšky podél geodetické normály adresované taktéž geografickými souřadnicemi (znázorněno modře). Geografické souřadnice lze vypočítat z pozice na povrchu elipsoidu, tedy bodu B. Jeho pozice je ale závislá na hodnotách ve výškovém poli.

Dostáváme se tedy do situace, kdy potřebujeme získat hodnotu z výškového pole, ale pro to, abychom se k ní dostali, ji už musíme předem znát. Podobný problém se vyskytuje u implementace Parallax Occulsion Mappingu, kde se řeší například aproximací a iterováním k výsledku s přijatelnou chybou.[5] Problém lze řešit i exaktně, tj. zkonstruováním 3D modelu výškového pole a hledáním průniku paprsku s modelem. Oba přístupy jsou však podstatně náročnější než standardní transformace s filtrováním. Dá se předpokládat, že by doba potřebná pro převod dat do krychlové projekce narostla několikanásobně, a to i s použitím GPU akcelerace.

Ještě větší problém než u výškových map by byl u ostatních typů dat. Pro správnou transformaci jakéhokoli typu geodetických dat do krychlové projekce je potřeba znát výšky. Ty však například v biomové mapě nemáme a vytvářet závislost na konkrétních výškových datech také pokud možno nechceme.

Druhý problém, který rozhodl o tom, proč od geocentrické projekce upustit, se projevil mnohem později – až při zapojení VBS Blue jako IG do simulátoru letounu F-16. Simulátor letadla sám o sobě informace o profilu terénu nemá. Na výšky se IG dotazuje pomocí CIGI[9] interfacu více jak stokrát za sekundu na každou entitu v simulaci. Simulátor však pracuje v geografickém souřadnicovém systému a dotaz na výšku posílá právě v něm. To nás však dostává do stejné situace jako u prvního problému. Jediný způsob, jak přesnou výšku terénu získat, je opět relativně náročným výpočtem, který si rozhodně nemůžeme dovolit opakovat tisíckrát za sekundu.

[9]CIGI je protokol pro propojení simulace s nezávislým rendererem. Komunikace v něm probíhá po TCP/IP socketu. Simulační server zasílá rendereru informace o pozicích objektů a kamery. Renderer zpět posílá například informace o relativní pozici objektů vůči terénu.



Obrázek 2.8: Geodetická transformace vertexů

2.2.2.2 Geodetický přístup

Vzhledem k tomu, že nevýhody geocentrické transformace vertexů převážily nad výhodami, byla ve VBS Blue provedena zásadní změna implementace a vertexy jsou nyní transformovány geodeticky. I tento přístup má své nevýhody, pojďme nejprve se ale budeme věnovat tomu, jak funguje.

Začínáme s vektorem na povrchu elipsoidu $\vec{v}_e$. Ten však tentokrát neposouváme směrem od středu planety, ale ve směru normály povrchu elipsoidu. Pokud máme bod na povrchu elipsoidu, vypočítáme normálový vektor vzorcem[6]

$$\vec{n} = (\frac{v_{ex}}{a^2}, \frac{v_{ey}}{b^2}, \frac{v_{ez}}{c^2}) \tag{2.14}$$

$$\hat{n} = \vec{n}/||\vec{n}|| \tag{2.15}$$

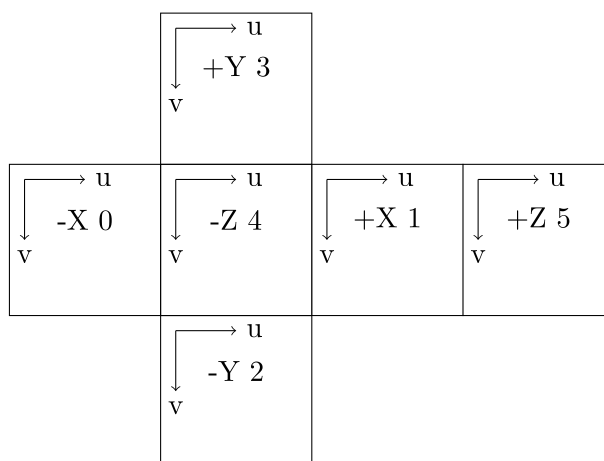
Bod poté dostaneme na povrch terénu přičtením normálového vektoru pronásobeného vzdáleností d, o kterou bod posouváme:

$$\vec{v}_s = \vec{v}_e + d\hat{n} \tag{2.16}$$

Transformace z elipsoidu na povrch terénu je tedy pouze o několik instrukcí náročnější než u geocentrického přístupu. Hlavní nevýhoda ovšem spočívá v inverzní transformaci, která je výrazně složitější. Pokud totiž máme bod v prostoru, potřebujeme k němu získat pozici na elipsoidu ve směru geodetické normály. Tento problém je přímo převoditelný na velice známý problém v geodézii – převod z karézských souřadnic do geodetických. Zabývá se jím množství vědeckých prací[7] a až do roku 1973 se mělo za to, že pro něj neexistuje exaktní řešení[8].

2.2.3 Formát cílových textur

Všechny textury mají identickou velikost 256 × 256 px. V této hodnotě je obsažen i 8 px overlap na každé straně, takže efektivní rozlišení je pouze 240 × 240 px. Po dosazení do vzorce 2.1 dostaneme maximální detail povrchu pouze 7,96 cm na texel. Tyto textury se však používají pouze na posun vertexů,



Obrázek 2.9: Číslování a orientace stran krychle

určení typu povrchu a biotopu. Při renderování na blízko se využívá opakující se detailní textura odpovídající typu povrchu, která je modulovaná hrubou barvou.

2.3 Převod mezi geodetickými a krychlovými souřadnicemi

Přestože specifikace zní, že bude možné převést jakoukoli mapovou projekci do krychlové, v typickém případě půjde o převod z geodetických souřadnic (nebo od nich odvozených). Podívejme se tedy, jak přesně mezi souřadnicemi převádět. Mezi geodetickými a krychlovými souřadnicemi není přímá spojitost, využijeme tedy mezikroku. Krychlové souřadnice dokážeme převést na bod v prostoru na povrchu elipsoidu. Ten lze převést do geodetických souřadnic. Inverzní transformace z geodetických souřadnic do krychlových bude fungovat stejně.

2.3.1 Převod z krychlových souřadnic do zeměpisných

Krychlové souřadnice máme zadány ve formátu $(strana, u, v)$, kde $strana$ je číslo strany krychle podle uspořádání na obrázku 2.9, u a v vyjadřují pozici na straně krychle v rozsahu $\langle 0, 1 \rangle$. Nejprve zkonstruujeme bod na krychli v prostoru. Na velikosti krychle nezáleží, důležité však je, aby měla střed v bodě $(0, 0, 0)$. Poté posuneme bod na elipsoid podle vzorce 2.12.

Zeměpisnou délku a šířku spočítáme pomocí trigonometrických funkcí z normály povrchu ve vypočítaném bodě. Pokud bychom počítali souřadnice přímo z pozice bodu na povrchu elipsoidu, dostali bychom totiž geocentrickou ze-

měpisnou šířku. Normálu tedy vypočítáme podle vzorce 2.14. Získáme tím následující funkci:

```
GeographicCoord EllipsoidBIWGS84::ToGeographicCoord(const CubeCoord&
    coord) const
{
  // calculate position on ellipsoid surface
  Vector3D surfacePos = ToEllipsoidSurface(coord);

  // calculate normal on ellipsoid surface
  Vector3D surfaceNormal = GetSurfaceNormal(surfacePos);

  // angles of the surface position vector relative to origin are
      latitude and longitude
  double lat = asin(surfaceNormal.Y() / surfaceNormal.Size());
  double lon = atan2(surfaceNormal.Z(), surfaceNormal.X());

  return GeographicCoord(lat, lon);
}
```

Výsledné mapování z geografických souřadnic na krychli je znázorněno na obrázku 2.11. Podle něj by se mohlo zdát, že na pólech v krychlové projekci vzniká singularita[10] stejně jako v projekci ekvidistantní. Je však třeba si uvědomit, že vyobrazený prostor znázorňuje ekvidistantní projekci, singularita tedy pochází z ní.

2.3.2 Převod ze zeměpisných souřadnic do krychlových

Převod ze zeměpisných souřadnic do krychlových je o něco složitější. Stejně jako předchozí algoritmus se skládá ze dvou kroků. Nejprve převedeme zeměpisné souřadnice na bod na povrchu elipsoidu a poté tento bod zprojektujeme na krychli. První krok je velice dobře popsán v knize 3D Engine Design for Virtual Globes.[6] Nejprve získáme normálu povrchu podle následujícího vzorce:

$$\hat{n}_s = cos\phi cos\lambda\hat{x} + cos\phi sin\lambda\hat{y} + sin\phi\hat{z}, \tag{2.17}$$

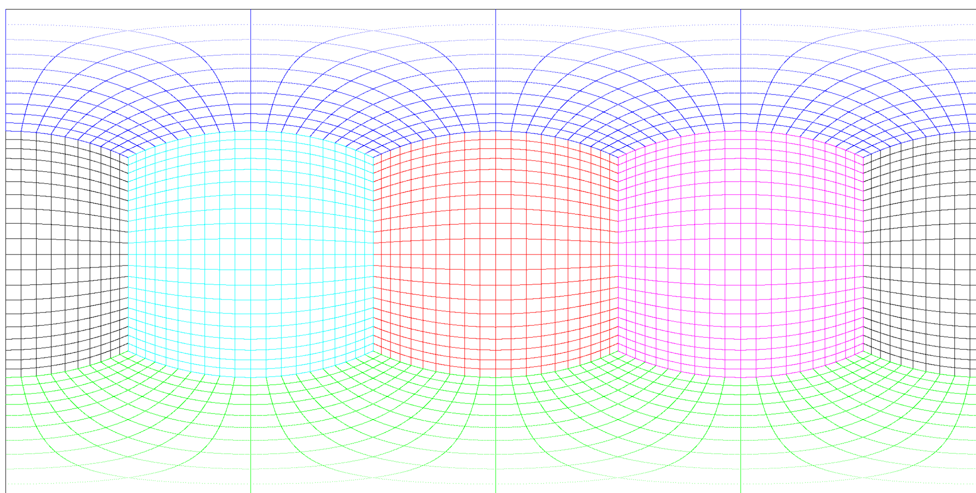
kde ϕ je zeměpisná šířka a λ zeměpisná délka. Ze vzorce 2.14 známe vztah mezi pozicí bodu na elipsoidu a nenormalizovanou normálou v tomto bodě. Víme, že mezi normalizovanou a nenormalizovanou normálou platí vztah

$$\hat{n}_s = \gamma\vec{n}_s. \tag{2.18}$$

Substituováním vzorce 2.14 za $\vec{n}_s$ dostaneme

$$\hat{n}_s = \gamma(\frac{v_{ex}}{a^2}, \frac{v_{ey}}{b^2}, \frac{v_{ez}}{c^2}). \tag{2.19}$$

[10] V geometrii se jako singularita označuje bod, kterému lze v dané projekci přiřadit více rozdílných souřadnic.



Obrázek 2.10: Transformace krychlových souřadnic do ekvidistantní válcové projekce

Chceme spočítat v_e, z rovnice tedy vyjádříme jeho komponenty:

$$
\begin{aligned}
\vec{v}_{ex} &= \frac{a^2 \hat{n}_{sx}}{\gamma} \\
\vec{v}_{ey} &= \frac{b^2 \hat{n}_{sy}}{\gamma} \\
\vec{v}_{ez} &= \frac{c^2 \hat{n}_{sz}}{\gamma}.
\end{aligned}
\tag{2.20}
$$

K vyřešení této soustavy nám již schází pouze zjistit hodnotu γ. Víme, že výsledný bod leží na povrchu elipsoidu, jehož rovnici známe ze vzorce 2.2:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1.$$

Po dosazení $\vec{v}_e$ dostaneme:

$$
\begin{aligned}
\frac{\left(\frac{a^2 \hat{n}_{sx}}{\gamma}\right)^2}{a^2} + \frac{\left(\frac{b^2 \hat{n}_{sy}}{\gamma}\right)^2}{b^2} + \frac{\left(\frac{c^2 \hat{n}_{sz}}{\gamma}\right)^2}{c^2} &= 1 \\
a^2 \hat{n}_{sx}^2 + b^2 \hat{n}_{sy}^2 + c^2 \hat{n}_{sz}^2 &= \gamma^2 \\
\gamma = \sqrt{a^2 \hat{n}_{sx}^2 + b^2 \hat{n}_{sy}^2 + c^2 \hat{n}_{sz}^2}.
\end{aligned}
\tag{2.21}
$$

Nyní známe všechny potřebné hodnoty pro výpočet $\vec{v}_e$ vzorcem 2.20. Zbývá tedy transformovat tento bod na krychli.

Přestože by šlo pojmout projekci bodu na krychli čistě matematicky, tedy kontrolou průsečíku úsečky vedoucí z bodu $\vec{v}_e$ do středu krychle, využijeme několika jednoduchých vlastností krychlového prostoru ke konstrukci mnohem jednoduššího algoritmu. Na velikosti krychle totiž nezáleží – úlohu si tedy zjednodušíme tím, že budeme uvažovat krychli procházející bodem $\vec{v}_e$. Hledaná strana a tedy i velikost této krychle je dána největší absolutní hodnotou bodu v jednotlivých osách.

Pokud máme krychli procházející bodem $\vec{v}_e$, koordináty u v rozsahu $(-1, 1)$ získáme prostým vydělením hodnoty tohoto bodu v ose, která jde ve směru u velikostí krychle. To samé samozřejmě platí i pro v. Ve VBS Blue mají u a v rozsah $(0, 1)$, tak hodnotu pouze upravíme do tohoto rozsahu. Výsledný kód vypadá takto:

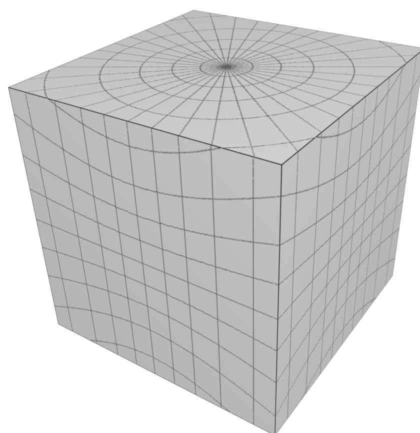
```
Vector3D EllipsoidBIWGS84::ToEllipsoidSurface(const GeographicCoord&
    geographic) const
{
  Vector3D n = GetSurfaceNormal(geographic);
  Vector3D k = _radii2.MultiplyComponents(n);
  double gamma = sqrt(k.X() * n.X() + k.Y() * n .Y() + k.Z() * n.Z());
  return k / gamma;
}

CubeCoord EllipsoidBIWGS84::EllipsoidSurfaceToCubeCoord(const
    Vector3D& pos) const
{
  // side is determined by greatest absolute value
  CubeSide side = ...;

  double u, v;
  switch (side)
  {
    case CubeSide::XP: u = +pos.Z() / pos.X(); v = -pos.Y() /
        pos.X(); break;
    ...
  }

  return CubeCoord(side, u * 0.5 + 0.5, v * 0.5 + 0.5);
}

CubeCoord EllipsoidBIWGS84::ToCubeCoord(const GeographicCoord&
    geographic) const
{
  return EllipsoidSurfaceToCubeCoord(ToEllipsoidSurface(geographic));
}
```



Obrázek 2.11: Transformace geografických souřadnic do krychlových

2.4 Architektura

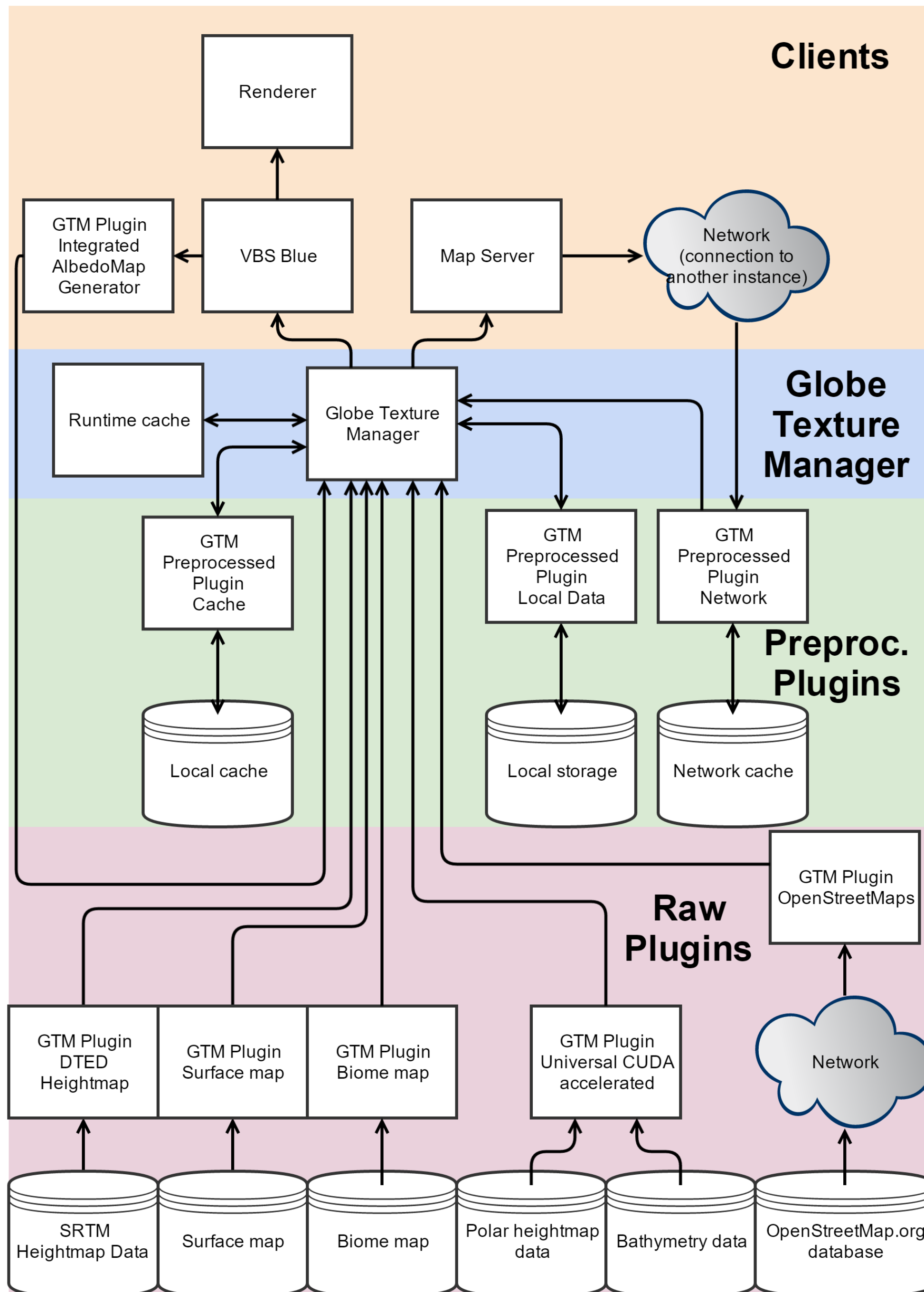
Architektura komponenty pro načítání dat je výrazně ovlivněna požadavky, které jsou na ni kladeny. Nejprve se podívejme na hotový návrh komponenty (obrázek 2.12), poté zdůvodníme proč jsme zvolili právě tento návrh.

2.4.1 VBS Blue a renderer

Realizujeme komponentu pro načítání dat do VBS Blue, je proto logické, že všechny cesty toku dat v diagramu 2.12 nakonec končí právě tam. Je nutné si ovšem vyjasnit, jaký je rozdíl mezi VBS Blue a rendererem. Přestože VBS Blue je název celého projektu a výsledného produktu, je třeba podotknout, že VBS Blue samo o sobě není funkčním celkem. Jedná se o softwarovou komponentu většího systému, podobně jako je komponenta pro načítání dat pouze součástí VBS Blue.

Knihovna VBS Blue definuje a spravuje datový model vesmíru, určuje co a kde se má vykreslovat a poskytuje pro vykreslování potřebná data. V (prozatím) omezeném rozsahu také provádí simulaci dynamických objektů ve scéně. Tato knihovna není vázaná na žádné konkrétní renderovací API, je možné ji dokonce provozovat zcela bez vykreslování.

Rendererem se myslí samotný vykreslovací engine. V našem případě se konkrétně jedná o modifikovaný engine Real Virtuality 3, který je prozatím sdílen s projektem VBS3. Až zde se textury kopírují ze systémové paměti do videopaměti.



Obrázek 2.12: Tok dat v komponentě pro načítání map

2.4.2 GlobeTextureManager a runtime cache

Třída `GlobeTextureManager` reprezentuje komponentu pro načítání rastrových dat, poskytuje tedy data pro textury VBS Blue v nativním formátu simulátoru. To, jakým způsobem data získá, je odstíněno interfacem, který `GlobeTextureManager` poskytuje. Jedním z hlavních požadavků na tuto komponentu je modularita, proto sama o sobě nic neukládá ani nenačítá. Načítání a ukládání dat je realizováno systémem pluginů, které nejsou pevnou součástí binárního modulu VBS Blue.

`GlobeTextureManager` obsahuje runtime cache, která drží veškerá načtená data v systémové paměti. Když se systémová paměť zaplní, začnou se uvolňovat textury, které byly nejdéle nepoužívané. Přestože je cache součástí `GlobeTextureManager`u, v diagramu je zobrazena samostatně, protože se v principu jedná o podobnou věc jako je local cache a network cache. Rozdíl je však v tom, že se nejedná o fyzické úložiště. Data jsou přítomna pouze v RAM.

Kromě toho, že runtime cache plní funkci rozšířené videopaměti dle uživatelských požadavků, má ještě jednu velice důležitou funkci – poskytuje data pro simulaci. Pokud VBS Blue potřebuje zjistit výšku terénu v určitém bodě na planetě, musí k tomu mít dostupná výšková data. Číst je z videopaměti je pomalé a nepraktické a duplikovat je v systémové paměti uvnitř modulu VBS Blue je také nevýhodné. Proto si simulace v případě potřeby zažádá o data z runtime cache, která jsou dostupná okamžitě.

Kromě propojení VBS Blue se systémem pluginů má `GlobeTextureManager` na starosti ještě zajištění asynchonnosti operace s hlavním/dotazujícím se vláknem, optimální využití všech CPU jader, správu runtime cache a slučování dat z více zdrojů do jednoho. To vše je zajištěno tak, že ve chvíli, kdy se VBS Blue dotáže `GlobeTextureManager`u na data, která jsou přítomna v runtime cachi, udělí k nim komponenta přístup. Pokud data načtena nejsou, spustí se na jiném vlákně procedura, která požadovaná data načte, zpracuje a uloží do runtime cache. Odtud si je VBS Blue při opětovném dotazování na data pro stejnou oblast přečte.

2.4.3 Pluginy na nezpracovaná data

Vstupní data do VBS Blue mají pouze jediný formát, který je optimální pro zpracovávání uvnitř simulátoru. Tento formát ovšem není žádným uznávaným standardem a jiný software jej nepodporuje. Zdrojové mapy v něm tedy nikde dostupné nejsou a musí se do něj převést. Právě k tomu slouží sada pluginů na nezpracovaná data.

Plugin je realizován buď formou dynamicky linkované knihovny, nebo je přímo součástí jiného modulu[11], např. VBS Blue. Plugin je definován velice jednoduchým interfacem, jehož konkrétní implementace už je na autorovi pluginu. Mapová data se na interfacu předávají přímo v podobě pole. Způsob,

[11] Viz 2.4.5.

jakým byla data obstarána, je interfacem skryto. Mohla být např. klasicky načtena ze zdrojového souboru a převedena do nativního formátu simulátoru, jak zobrazují uvedené příklady pluginů DTED Heightmap, Surface map a Biome map. Universal CUDA accelerated plugin pracuje podobně, převod dat však provádí s použitím GPU akcelerace, pokud je k dispozici.

Tento systém ale umožňuje mnohem zajímavější způsoby získávání dat. Mapové zdroje nemusí být uloženy na místním počítači a jejich počet, nebo oblast kterou pokrývají, může být také neznámá. Jak ukazuje příklad s OpenStreetMap pluginem, zdrojové mapy mohou být stahovány ze serveru `http://openstreetmap.org` ve vektorovém formátu, dále převedeny do krychlové projekce, rasterizovány, a poté předány VBS Blue k renderování.

Nemusí přitom jít ani o statické mapy. V plánu je i plugin, který bude stahovat aktuální informace o počasí z `http://openweathermap.org`. Tato mapová vrstva se ve VBS Blue posléze použije pro generování oblačnosti.

Dalším možným druhem pluginů jsou ty, které produkují geotypický nebo zcela náhodný terén. Plugin data jednoduše vytvoří skládáním šumových textur podle různých pravidel.

2.4.4 Pluginy na zpracovaná data

Pluginy na zpracovaná data (binarizační pluginy) jsou podobné pluginům na data nezpracovaná. Rozdíl je však v tom, že ke zpracovaným datům je přidané záhlaví obsahující souhrnné informace o textuře. Jeho obsah závisí na typu textury. U výškových dat je zde uložena například informace o minimální a maximální výšce v textuře a bounding box[12] segmentu, který texturu obsahuje. Ve VBS Blue se tyto informace používají k různým optimalizacím, jako je například ořezávání vykreslované oblasti podle zorného pole kamery. Minimální a maximální výška se pro změnu používá při vykreslování vody, aby se vodní hladina nevykreslovala tam, kde je kompletně zakryta zemí.

Další rozdíl oproti standardním pluginům je ten, že tyto pluginy podporují ukládání. Zpracovávání dat do optimálního formátu tedy probíhá tak, že se vstupní data načtou standardními pluginy, `GlobeTextureManager` k nim připraví záhlaví a uloží je pomocí binarizačníh pluginů. Typicky však budeme chtít mít aktivní pouze jeden plugin, který podporuje ukládání.

Ani u binarizačních pluginů interface nevynucuje ukládání/načítání dat v konkrétním formátu. Jedinou podmínkou, kterou by měly tyto pluginy splňovat, aby správně plnily svůj účel, je to, aby načítání dat z nich probíhalo rychle.

Je několik druhů binarizačních pluginů, s jejichž existencí `GlobeTextureManager` počítá. Pokud je přítomen jeden zástupce každého druhu těchto pluginů, bude

[12]Bounding box je informace o rozměru objektu v prostoru, který se využívá k urychlení vykreslování scény. Místo toho, aby se objekt vykresloval vždy, vykreslí se pouze tehdy, pokud je jeho bounding box v zorném poli kamery. Kontrolovat, zda je bounding box v zorném poli, je mnohonásobně rychlejší než kontrolovat každý vertex objektu zvlášť.

celá komponenta pracovat nejoptimálněji. Pokud jeden z nich bude chybět, `GlobeTextureManager` bude fungovat dál, ale přijde o část své funkcionality kterou plugin zastupuje. Tyto pluginy, přestože mají speciální funkcionalitu, mají jednotný interface společný se všemi binarizačními pluginy.

2.4.4.1 Standardní binarizační plugin

Standardní binarizační plugin je plugin, který podporuje ukládání a načítání zbinarizovaných dat. Typicky je implementován tak, že data při hromadné binarizaci ukládá do souboru. Při standardním spuštění VBS Blue má funkci ukládání vypnutou, protože ukládání pouze těch dat, která si VBS Blue zrovna vyžaduje pro běh, žádnému praktickému účelu neslouží.

Pro ukládání zpravidla využívá silnou kompresi, aby kompletní data planety zabírala na pevném disku co nejméně místa. Komprese je také optimalizována tak, aby načítání bylo co nejrychlejší.

2.4.4.2 Cachovací plugin

Je-li to zapotřebí, slouží tento plugin k cachování mapových dat z ostatních puginů. Načítání nezpracovaných dat může trvat velmi dlouho, a tak je výhodné si jejich výsledky po nějakou dobu uchovávat na disku, aby opětovné načítání stejné oblasti bylo rychlejší.

Cachovací plugin by měl být optimalizován pro maximální rychlost ukládání a načítání, jinak by nic nezrychloval. Cachují se totiž i relativně rychle načtená data z ostatních pluginů v případě, že se jich načte více pro jeden segment. V takovém případě se totiž musí tyto textury sloučit do jedné, což sice není tak časově náročná operace jako převod textury, který dělají pluginy na nezpracovaná data, ale stále je několikanásobně delší než načtení jedné optimalizované textury (už jenom proto že se jich musí načíst více než jedna).

2.4.4.3 Síťový plugin

Síťový plugin je navenek podobný standardnímu binarizačnímu pluginu, který nemá schopnost ukládat data. Uvnitř však funguje zcela jinak a zdrojová data nenačítá ze souboru, ale ze vzdáleného serveru, který mu je poskytuje ve zbinarizované podobě.

Načítání ze vzdáleného serveru je časově relativně náročná operace, kterou má význam cachovat samostatně. I když je totiž standardní cache aktuální, musí se `GlobeTextureManager` stejně dotázat všech pluginů, zda mají dostupná data pro zpracovávanou oblast. To by v případě tohoto pluginu trvalo opět dlouho, protože by se musela přes síť zasílat žádost o to, zda data existují, a poté by se muselo ještě čekat na odpověď. Proto by měly mít síťové pluginy svou vlastní cache, do které ukládají již stažená data, čímž minimalizují síťový provoz.

2.4.5 Integrované pluginy

Ne každý plugin musí být vyměnitelný a realizovaný dynamickou knihovnou. Některé pluginy sloužící speciálnímu účelu mohou být přímo součástí klientské aplikace, která `GlobeTextureManager` využívá. VBS Blue například používá integrovaný plugin k vygenerování barevné textury povrchu. Ten je navíc zajímavý tím, že k vygenerování jedné textury potřebuje tři zdrojové textury – výškovou mapu, mapu typu povrchů a biomovou mapu. Z těch vytvoří barevnou texturu, kterou předá standardním způsobem `GlobeTextureManageru`. Ten ji předá zpět do VBS Blue, jako by byla barevná mapa načtena z běžného souboru.

2.4.6 Map Server

Pro síťový plugin samozřejmě musí existovat server, který by data posílal. Na konkrétní implementaci serveru nezáleží, uvažujeme klasický vícevláknový TCP/IP server. Do něj se zapojí `GlobeTextureManager` podobně jako do VBS Blue a může tak poskytovat mapová data s implementací dlouhou pouze několik řádků. Všechny vlastnosti `GlobeTextureManageru` zůstanou zachovány a právě cachování map v RAM bude na serveru obzvláště užitečnou vlastností. Dá se předpokládat, že klienti serveru budou v rámci jednoho cvičení operovat ve stejné oblasti. Data na serveru tedy už budou v cachi okamžitě připravena k odeslání.

2.5 GPU Akcelerace

Z navržené architektury vyplývá, že samotná transformace dat bude probíhat v pluginech. Způsob, kterým se v pluginu data převádí není pevně daný. V některých pluginech nemusí probíhat transformace vůbec. Není proto možné implementovat univerzální transformaci přímo jako součást `GlobeTextureManageru`.

V době psaní tohoto textu již pro `GlobeTextureManager` existují čtyři pluginy, které řeší transformaci na CPU:

- DTED Heightmap plugin, který načítá výšková data projektu SRTM ve formátu DTED v ekvidistantní válcové projekci,
- MODIS Surfacemap plugin, který načítá informace o typu povrchu z dat satelitu MODIS ve formátu PNG v sinusoidové projekci,
- MODIS Biomemap plugin, který načítá informace o biomech z dat satelitu MODIS ve formátu PNG v sinusoidové projekci,
- Random heightmap plugin, který generuje náhodnou výškovou mapu s neomezeným rozlišením.

Random heightmap plugin se již nepoužívá, byl vytvořen dočasně pro testování rendereru do doby, než byla k dispozici reálná výšková data. Tři základní pluginy pro načítání reálných dat jsou velice jednoúčelové a celková doba binarizace dat pro celou planetu se pohybuje v řádu několika hodin. GPU akcelerace by u nich sice pomohla, ale není nezbytně nutná.

Ve vývoji je však pátý plugin, u kterého by byla akcelerace výpočtů na GPU vhodná. Jedná se o GDAL plugin, který využívá knihovnu GDAL pro čtení zdrojových dat (prozatím pouze výškových) ze širokého spektra podporovaných formátů a projekcí. Transformace probíhá ve dvou krocích. Nejprve se data načtou ze zdrojového souboru a převedou do pevně dané projekce knihovnou GDAL a poté je plugin převede do krychlové projekce (knihovna GDAL přímo krychlovou projekci nepodporuje).

Právě druhý krok je velice vhodný k paralelizaci na GPU. Jedná se o převod z pevně dané projekce do jiné a může probíhat na GPU ve chvíli, kdy je CPU zaneprázdněno první fází. K výpočtům na GPU bude využita technologie NVIDIA CUDA.

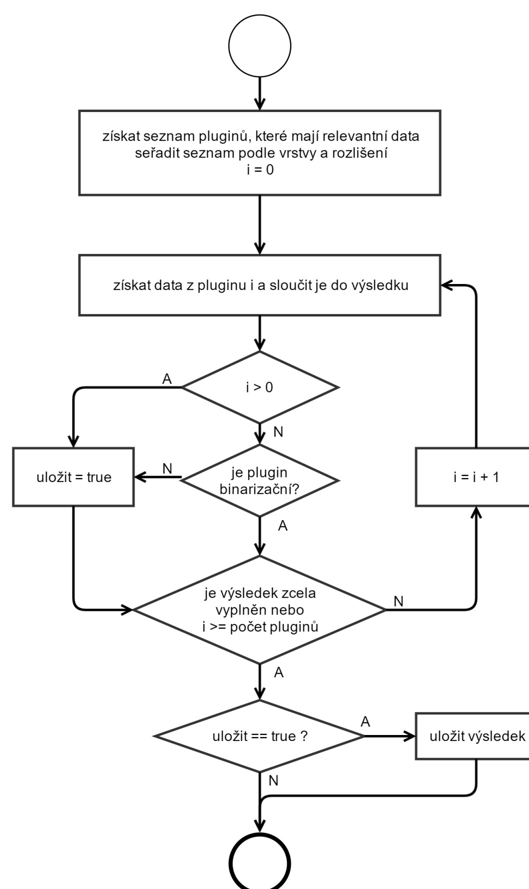
2.6 Slučování dat

Z předchozího návrhu je zřejmé, že může často nastat situace, kdy je dostupných více mapových zdrojů pro stejnou oblast. Na první pohled se může zdát, že by se měl jednoduše vybrat „nejlepší“ dostupný zdroj. Jednoznačně určit nejlepší zdroj ovšem není tak jednoduché. Je to ten, který pokrývá největší plochu textury? Ten, který má nejvyšší rozlišení? Ten který se nejvíce blíží realitě? Ten který má největší vážený průměr všech těchto vlastností?

Nejlepší zdroj bude ten, který vznikne sloučením všech zdrojů, kde se z každého použije právě to nejlepší. Také je potřeba, aby uživatel mohl ovlivnit, který mapový zdroj má přednost v případě, že standardní slučovací algoritmus nepřináší požadované výsledky. Tento vstup uživatele zakódujme do parametru „vrstva“. Ta bude použita jako hlavní parametr pro řazení mapových zdrojů – bude tedy fungovat analogicky k překrývání papírových map přes sebe na stole. Mapa ve vrstvě 1 překrývá mapu ve vrstvě 0 pod ní.

Dalším parametrem je věrohodnost mapy. Veřejně dostupné mapové zdroje mají například zaretušované oblasti s vojenskými objekty. Mapa takové základny, i kdyby měla menší rozlišení než zaretušovaný zdroj, je pro výsledek užitečnější. Nemusí přitom jít jen o zaretušované mapy. Některé mapové zdroje, jako například výšková SRTM data v hornatých oblastech, trpí artefakty způsobenými nepřesným snímáním. Tento parametr však nejde jednoduše automaticky detekovat, a proto jeho nastavení necháme opět na uživateli pomocí parametru vrstva.

Rozlišení zdrojových dat je naopak měřitelné velice snadno a neexistuje důvod proč neupřednostňovat data ve vyšším rozlišení než v nižším.



Obrázek 2.13: Algoritmus na slučování více mapových zdrojů

Posledním parametrem je zaplněnost cílové textury. Zdrojová data nemusí kompletně vyplňovat cílovou texturu ze dvou důvodů. První důvod je ten, že zdrojová textura nemusí být (a zpravidla také není) zarovnaná na cílovou texturu v krychlové projekci. Může to být jak tím, že zdroj pokrývá menší oblast než cíl, ale také tím, že obdélník s mapovými daty ve zdrojové projekci nemá stejný tvar v projekci cílové. Druhým důvodem je, že ani zdrojový soubor nemusí být kompletně vyplněný. V GISových programech se běžně používá hodnota `NoValue`, která označuje, že v daném místě nejsou informace dostupné.

Zaplněnost ale pro řazení vůbec využívat nebudeme. Výslednou texturu vytvoříme spojením všech dostupných zdrojů. Pro vybrání nejlepšího zdroje mapy seřadíme nejdříve podle vrstvy a v rámci stejné vrstvy seřadíme podle rozlišení. Tam, kde v nejlepším zdroji data chybí, dosadíme data ze druhého nejlepšího, třetího, atd. Algoritmus je znázorněn na obrázku 2.13.

Kapitola 3

Realizace

Implementace je realizována v C++11, použité knihovny se však liší v závislosti na implementovaném modulu. Cílovou platformou je 64bitový systém Windows 7 a novější. Všude, kde je to možné, se ovšem používá multiplatformní kód. Existuje totiž jistá šance, že by například uživatelé mohli chtít provozovat `MapServer` na Linuxovém serveru. Snažíme se tedy co nejvíce usnadnit portování na jinou platformu.

3.1 GlobeTextureManager

`GlobeTextureManager` je hlavní třída reprezentující celou komponentu pro načítání a transformaci dat. Je součástí modulu VBS Blue, protože jeho funkcionalita je s ním úzce spjata. K vytvoření záhlaví textur je totiž potřeba znát kontext, ve kterém se budou používat. Například záhlaví výškových textur obsahuje bounding boxy[13] segmentu, ke kterému je textura přiřazena. Výpočet bounding boxu se ale přímo odvíjí od implementace segmentu ve VBS Blue, a tak je potřeba buď výpočet v `GlobeTextureManager`u duplikovat, nebo vypočítat na straně VBS Blue a tím vytvořit závislost `GlobeTextureManager`u na VBS Blue. Přijatelnější alternativou je vytvoření závislosti.

Vzhledem k tomu, že je součástí VBS Blue, nevyužívá `GlobeTextureManager` standardní knihovnu STL, ale interní knihovny Element a Essence, s kterými pracuje zbytek enginu. Tyto knihovny jsou na rozdíl od STL zaměřeny výhradně na výkon.

3.1.1 Vnější interface ke GlobeTextureManageru

Interface `GlobeTextureManager`u se odvíjí od potřeb VBS Blue, konkrétně tedy:

[13] VBS Blue používá dva druhy bounding boxů – axis-aligned a orientovaný. S axis aligned jsou výpočty jednodušší a rychlejší, orientovaná prozměnu lépe odpovídá reálnému tvaru objektu.

- zjistit, zda data pro určitou oblast existují,
- načíst data synchronně,
- načíst data asynchronně,
- zjistit stav načítání,
- určit prioritu načítaných dat,
- načítat data po jednotlivých mipmapách,[14]
- načítat záhlaví dat.

Tyto požadavky lze shrnout do pouhých dvou funkcí:

```
//! Access a map source. Whether this function blocks until the
    source is loaded is determined by access argument.
MapSourceAccess AccessMapSource(MapSourceType type, const
    CubeRegionExt& region, AccessType access, float priority = 0.0f);
//! Access map data. Whether this function blocks until the source is
    loaded is determined by access argument.
MapDataAccess AccessMapData(MapSourceType type, const CubeRegionExt&
    region, int level, AccessType access, float priority = 0.0f);
```

`GlobeTextureManager` tedy definuje „MapSource“, který reprezentuje texturu jako celek četně záhlaví a „MapData“, která představují data konkrétní mipmapy textury. Obě funkce mají několik společných parametrů:

- `MapSourceType type` – určuje typ textury (viz 3.1.3),
- `CubeRegionExt region` – oblast na planetě pro kterou data získáváme,
- `AccessType access` – rozlišuje synchronní požadavek od asynchronního,
- `float priority` – čím se priorita blíží více nule, tím má požadavek větší přednost.

[14]Mipmaping je princip používaný v počítačové grafice, který má tři hlavní funkce: redukovat aliasing[15], zabránit načítání nepotřebných dat a načítat scénu postupně a rovnoměrně. To je zajištěno tak, že pokud textury ještě nejsou kompletně načtené, nebo pokud jsou daleko od kamery, vykreslují se v nižším rozlišení.

[15]Aliasing je nežádoucí efekt, který vzniká, pokud se barva sousedních pixelů na obrazovce výrazně liší. Objevuje se například když je v dálce vykreslována cihlová zeď, která má texturu s vyšším rozlišením, než kolik zabírá na obrazovce. Jeden obrazovkový pixel totiž pokrývá jak červenou cihlu, tak bílé spáry mezi nimi. Bez mipmapingu bude pixel buď červený, nebo bílý a okolní pixely také, což vytváří nepřirozený efekt podobný šumu. Pokud se navíc kamera hýbe, barvy pixelů zpravidla alternují a aliasing je o to více viditelný. V dálce se proto používá pro vykreslování mipmapa, což je identická textura, která má snížené rozlišení. V mipmapě je texel například ze 75 % červený a 25 % bílý a pokrývá větší plochu, což aliasingu zabrání.

CubeRegionExt je rozšířením třídy CubeRegion o ID segmentu [16] . Jedná se o optimalizaci na úkor flexibility GlobeTextureManageru. Pokud se totiž bude k získávání dat používat ID, bude GlobeTextureManager omezen pouze na načítání čtvercových textur odpovídajících segmentům namísto libovolných obdélníkových oblastí na krychli. Po pečlivém uvážení bylo rozhodnuto, že se tato optimalizace vyplatí, protože hledání v cachi podle ID je rychlejší a spolehlivější[17] než podle čtyř koodrinátů.

V CubeRegionExt je tedy duplicitní informace – SegmentId lze vypočítat z CubeRegion a naopak. Výpočet však není triviální a GlobeTextureManager interně potřebuje znát obě informace – SegmentId pro cachování a CubeRegion pro pluginy. Kvůli optimalizaci tedy CubeRegionExt uzavírá obě tyto informace do jedné struktury.

AccessType je výčtový seznam tří možných zpsobů přístupu k datům:

- AccessType::Load vynucuje synchronní načítání. Metoda zavolaná s tímto parametrem se nevrátí, dokud nebudou požadovaná data kompletně načtena, nebo dokud nebude zjištěna jejich nedostupnost. Volání metody s tímto parametrem se na hlavním vlákně během renderování silně nedoporučuje, protože může způsobit zamrznutí hlavního vlákna na neurčitě dlouhou dobu.

- AccessType::Preload vrátí data, pokud jsou již načtena. Pokud ne, spustí na pozadí asynchronní načítání vyžádaných dat. Tento typ přístupu je na hlavním vlákně upřednostňován.

- AccessType::CheckLoaded vrátí data pouze tehdy, pokud jsou už načtena. Nespouští jejich načítání na pozadí.

Výběr správného typu přístupu pro správnou příležitost je naprosto klíčový. Podobný mechanismus je zaveden ve VBS3, kde příliš časté používání synchronního přístupu k datům způsobovalo problémy s výkonem.

Parametr priority určuje, který požadavek bude mít přednost. Kompletní načtení scény ve VBS Blue totiž může chvíli trvat, a tak se věci, které mají největší dopad na vzhled scény, načítají nejdřív. Rozhodujícím faktorem je velikost na obrazovce. Chceme totiž, aby co největší plocha obrazovky byla načtena co nejdříve. Načtení malých objektů v dálce není tak důležité. Načtením nejmenších segmentů v zorném poli však načítání scény nekončí. Dále se totiž přednačítají[18] segmenty za kamerou pro poměrně pravděpodobný případ, že by se kamera otočila. Toto přednačítání probíhá paralelně s načítáním segmentů v zorném poli, protože to, aby se kamera náhle nepodívala do zcela

[16] Každý segment ve VBS Blue má své SegmentId, což je jeho unikátní 64bitové číslo, ve kterém je zakódována pozice segmentu na planetě.

[17] Souřadnice jsou určeny typem double, což má nevýhodu v tom, že při opakovaném výpočtu hranic stejné oblasti se mohou výsledky mírně lišit. Celočíselné ID je oproti tomu určeno jednoznačně.

nenačtené oblasti je důležitější než jemné detaily v dálce. Dále probíhá přednačtení jemnějších úrovní detailu, než jaké se aktuálně vykreslují, pro okamžité přepnutí detailu v případě, že se kamera přiblíží k segmentu. Z těchto všech faktorů se vypočítá výsledná priorita, která ovlivňuje pořadí, ve kterém budou data načtena. Stojí za povšimnutí, že `AccessMapData` nevrací přímo data, ani pointer na ně. Vracet kopii dat by zbytečně způsobovalo kopírování paměti v případě, že jsou data potřeba jen na velice krátkou dobu. Na druhou stranu vracet na ně ukazatel není bezpečné, protože ve vícevláknovém prostředí se může stát, že budou smazána jiným vláknem při pokusu o uvolnění paměti. Návratovou hodnotou metody `AccessMapData` je tedy instance třídy `MapDataAccess`, která poskytuje přístup k samotným datům a garantuje jejich existenci až do své destrukce. Odtud má metoda `AccessMapData` svůj název – místo `GetMapData`, což by naznačovalo, že data získáváme, k nim místo toho pouze udělujeme přístup (access).

Interně je třída `MapDataAccess` implementována smart pointerem, který drží referenci na záznam textury v interní cachi[19]. Proč tedy `AccessMapData` nevrací přímo pouze smart pointer na záznam v cachi? Předávání smart pointeru přímo na záznam by totiž odhalovalo implementaci `GlobeTextureManager`u, což by mohl být v budoucnu problém z hlediska udržovatelnosti kódu při změně implementace cache. Takto je celý zapouzdřen a jeho implementace je uživateli skryta. Vše, co platí pro `AccessMapData`, samozřejmě platí analogicky i pro `AccessMapSource`.

Jak tedy uživatel interfacu `GlobeTextureManager` získá informaci o stavu načítání? `DataManager`[20] definuje čtyři stavy, ve kterých se načítaná data mohou nacházet:

```
//! Status of loading of a resource
enum class LoadStatus : char
{
  //! Loading didn't start yet
  Unknown,
  //! Sources are not present for specified location
  NoSource,
  //! Sources are present but not loaded yet
  Loading,
  //! Data are prepared
  Ready
};
```

Stav načítání se získává z `MapSourceAccess` nebo `MapDataAccess` metodou `GetStatus`. Stav se přes tyto třídy získává ze stejného důvodu jako data. Jejich instance totiž garantuje, že hodnota stavu bude mezi jednotlivými volá-

[18] V angličtině se používá ustálený termín „preload“. Jedná se o načtení dat dříve, než jsou potřeba.

[19] Viz kapitola 3.1.2.

[20] Viz kapitola 3.2.

nními vždy stejná nebo vyšší. Bez tohoto mechanismu by mohlo mezi kontrolou, zda jsou data načtena, a samotným získáním dat, dojít k jejich uvolnění jiným vláknem.

3.1.2 Interní cache

Pro interní cache bylo potřeba zvolit ideální kontejner, který poskytuje co nejrychlejší vyhledávání podle zadaného klíče, kterým je v tomto případě `CubeRegionExt`. Rychlost přidávání nových prvků a mazání je vedlejší, protože přidávání typicky probíhá asynchronně k dotazujícímu se vláknu a mazání se děje ve chvíli, kdy se žádná data nenačítají. Těmto požadavkům nejlépe odpovídá hashovací tabulka, v knihovně Essence reprezentovaná třídou `MapStringToClass`[21]. Operace vyhledávání, vkládání a mazání mají v průměrném případě složitost $\mathcal{O}(1)$.

Položka v cachi je reprezentovaná třídou `CacheItem`. Ta obsahuje ukazatel na záhlaví, stav načítání a statické pole[22] s mipmapami. Každá mipmapa obsahuje svůj vlastní stav načítání a ukazatel na data.

Interní cache implementuje interface na správu paměti používaný v RV enginu zvaný `MemoryFreeOnDemandHelper`. V principu funguje tak, že když dochází systémová paměť, je vytvořen vážený seznam z cachí[23], které jsou do něj zaregistrované. Z tohoto seznamu se vybere kandidát, který drží největší množství nejméně potřebné paměti a uvolní ji. Proces se opakuje, dokud není uvolněno dostatečné množství paměti.

`MemoryFreeOnDemandHelper` tato cache implementuje tak, že každá položka je zároveň zaregistrovaná v LRU seznamu. Při každém použití záznamu v cachi se položka přesune na začátek seznamu. Tento přesun je při použití spojového seznamu knihovny Essence podstatně rychlejší než stejná operace v `std::list`. Díky tomu se neztrácí výkon při načítání scény, kdy se přistupuje na stovky prvků v cachi každý snímek. Při uvolňování paměti se v LRU seznamu najde první prvek od konce, který se momentálně nepoužívá[25], a smaže se.

[21] Název `MapStringToClass` je poněkud zavádějící, přetrval však z historických důvodů. První implementace tohoto kontejneru v devadesátých letech opravdu umožňovala asociovat objekty pouze k řetězcovému klíči, v současnosti už ovšem klíč může být libovolného typu. Tento kontejner svým interfacem a vlastnostmi nejblíže připomíná `std::unordered_set`.

[22] Počet mipmap je v `GlobeTextureManager`u konstantní stejně jako rozlišení textur, které je vždy $2^{mipCount}$.

[23] Cachí v tomto případě myslíme jakoukoli strukturu, která alokuje větší množství paměti, kterou lze kdykoli uvolňovat a v případě potřeby znovu načíst nebo vytvořit. Takovými cachemi jsou například načtená data `FileServer`u[24], seznamy objektů, načtené modely, načtené textury objektů, navigační rastry umělé inteligence a další.

[24] `FileServer` je třída umožňující bezpečný a efektivní přístup k souborům z více vláken.

[25] Počet aktivních instancí MapDataAccess a MapSourceAccess se pozná podle počtu referencí ve smart pointeru na prvek v cachi. Pokud je počet referencí roven jedné, není prvek cache aktuálně používán a je možné jej smazat.

3.1.3 Typy mapových zdrojů

`GlobeTextureManager` umožňuje načítání různých typů dat. Podle původního návrhu jimi měly být jen tři základní typy dat pro vykreslování segmentů, tedy výšková mapa, mapa typu provrchů a biomová mapa. Ukázalo se však, že `GlobeTextureManager` je natolik flexibilní, že je výhodné jej používat i na typy textur, na které původně ani nebyl navržen. K základním typům tedy přibyly metatextury používané k vykreslovnání silniční sítě, které se generují z vektorových dat, barevná textura, která se generuje ze tří základních typů textur, a výšková mapa vodní hladiny, která se vytváří složením geoidu[26] s maskou vodního pokrytí.

Každý typ dat má své specifické vlastnosti. Přesto bychom však chtěli, aby se všemi dokázal pracovat jediný kus sdíleného kódu, aby nebyla po zdrojovém kódu `GlobeTextureManager`u roztroušena různá místa, kde by se kód větvil pro každý typ dat zvlášť. Přidat nový typ by v takovém případě byl namáhavý proces náchylný na chyby.

Všechny specifické vlastnosti mapových zdrojů tedy uzavřeme do tříd, které musejí implementovat interface `MapTypeDescriptor`. Ten popisuje formát dat jako například velikost jednoho pixelu v bytech a hodnotu prázdného pixelu. Proměnlivá velikost pixelu bohužel interface poněkud komplikuje. Dále deskriptor definuje některé základní algoritmy pro práci s daným typem rastrových dat. Tyto algoritmy, jako například filtrování hodnot, pracují nad poli pixelů místo toho, aby se volaly na každý pixel zvlášť. To sice produkuje poněkud delší kód, který se místy opakuje, nicméně opakované volání virtuálních funkcí na každý pixel by způsobovalo nežádoucí zpomalení.

3.1.4 PluginGC

Cachování je vhodné provádět nejen na straně enginu, ale i v pluginech. Tím pádem se i v pluginech musí spouštět garbage collector, což komplikuje fakt, že pluginy nejsou součástí modulu enginu a nemohou tak být do garbage collectoru přímo zaregistrovány. `GlobeTextureManager` tedy obsahuje třídu `PluginGC`, která implementuje `MemoryFreeOnDemandHelper`. `PluginGC` poté komunikuje s pluginy pomocí jejich interfacu[27], ve kterém jsou přidány pouze metody nezbytně nutné pro funkčnost garbage collectoru. Nechceme totiž zákazníky zatěžovat vynucenou implementací složitého interfacu, který by někteří ani nehodlali využívat.

Všechny pluginy jsou tedy zaregistrovány společně jako jeden vážený prvek v garbage collectoru. Ze kterého pluginu však uvolnit paměť ve chvíli, kdy je pluginů více a uvolnit nějaké množství paměti může každý z nich? Jednotlivé

[26] Vodní hladina na povrchu Země netvoří ani po vyloučení gravitačního vlivu Měsíce dokonale hladký elipsoid. Je ovlivněna gravitačním polem Země, které není rovnoměrné. Těleso, které má tvar takto deformované mořské hladiny, jež je definovaná i v oblasti kde jsou kontinenty, se v geodézii nazývá geoid.

[27] Viz 3.3.1.

pluginy nemůžou mít své vlastní váhy jako v případě tříd uvnitř enginu implementujících `MemoryFreeOnDemandHelper`. Zákazníci totiž nemohou vědět, jaké váhy mají ostatní pluginy a podle toho zvolit váhu svého. Paměť bude tedy jednoduše uvolňovat vždy plugin, který jí má naalokováno nejvíce.

3.2 DataManager

Textury mapových podkladů nejsou jediným typem dat, který je potřeba načítat. Tyto jiné druhy dat sice nebude načítat `GlobeTextureManager`, jejich načítání by však mělo také probíhat vůči hlavnímu vlákno asynchronně. Místo toho, aby měla každá taková komponenta vlastní správu vláken, bude vše probíhat centrálně přes třídu `DataManager`. Ta bude zajišťovat rozdělování veškeré práce týkající se načítání a zpracování vstupních dat dostupným procesorům.

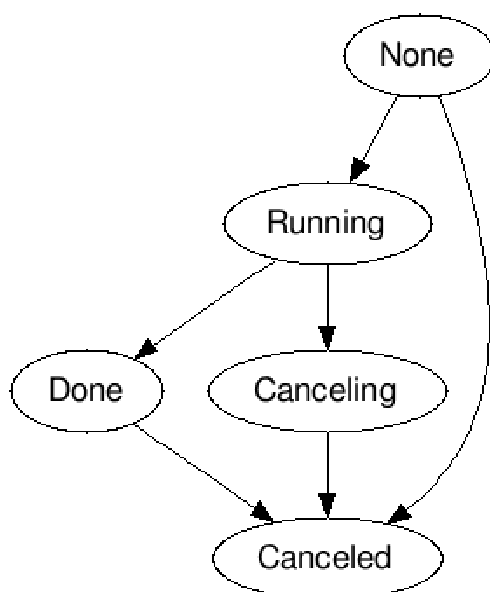
`DataManager` disponuje sadou vláken, jejichž počet se rovná počtu fyzických procesorů mínus jeden. Logická jádra vytvořená hyperthreadingem nevyužíváme, neboť měření ukázala, že nepřináší zvýšení výkonu. Jedno jádro `DataManager`em záměrně nevyužíváme, neboť na něm běží hlavní vlákno, které nechceme za žádnou cenu zpomalovat.

Používání `DataManager`u je poměrně jednoduché. Uživatel vytvoří třídu implementující interface `DataManagerJob`, ve které implementuje metodu `Process`. Objektu této třídy jsou v konstruktoru poskytnuty potřebné parametry a následně je předán `DataManager`u metodou `SubmitJob`. Objekt se zařadí na správné místo do interní fronty podle priority a později jej z odtud odebere volné vlákno, které na něm zavolá metodu `Process`.

Pokud si uživatel ponechá referenci na DataManagerJob po jeho zařazení do fronty, může sledovat jeho stav metodou `GetState`. Možné stavy a přechody mezi nimi jsou zobrazeny na obrázku 3.1. Na počátku se job nachází ve stavu `None`, ve kterém setrvá i po zařazení do fronty. Když běží metoda `Process`, je job ve stavu `Running` a po jejím dokončení se stav změní na `Done`. Na `DataManagerJob` lze kdykoli zavolat metodu `Cancel`, která jej uvede do stavu `Canceled` značícího, že výsledek operace není platný. Toho se využívá například při ukončování aplikace, aby se požadavky ve frontě přestaly zpracovávat, ale zároveň se legálně ukončily. Přechod do stavu `Canceled` je ze stavu `None` a `Done` okamžitý. Přechod z `Running` přímo do `Canceled` není možný, protože `Process` ještě pořád běží a na jeho dokončení čekat v metodě `Cancel` nechceme. Stav se tedy nastaví na `Canceling` a `DataManagerJob` se po ukončení metody `Process` uvede do stavu `Canceled` místo `Done`.

3.2.1 DataJobTracker

Interface `DataManager`u je sice jednoduchý, jeho typické použití je však poněkud náročné na režii. Většinou totiž uživatel vytváří více `DataManagerJob`ů najednou, jejichž průběh chce sledovat a po jejich dokončení z nich případně



Obrázek 3.1: Přechody stavu DataManagerJobu

vyzvednout výsledek. Kód, který by to zajišťoval není úplně triviální, a tak zavedeme třídu `DataJobTracker`, která usnadňuje manipulaci s `DataManagerJoby`.

`DataJobTracker` usnadňuje následující často prováděné operace s `DataManagerJoby` a přidává k nim garanci vícevláknové bezpečnosti:

- vyhledání odeslaného `DataManagerJobu`,
- údržbu kolekce aktivních `DataManagerJobů`,
- čekání na dokončení konkrétního `DataManagerJobu`,
- hromadné zrušení všech `DataManagerJobů` při ukončování aplikace.

3.3 Pluginy

Pluginy pro `GlobeTextureManager` mají obvykle podobu DLL modulu, který obsahuje veškerou logiku pro načítání a transformaci mapových dat do krychlové projekce. `GlobeTextureManager` při inicializaci vyhledá všechny DLL soubory v adresáři s pluginy a pokusí se je načíst.

3.3.1 Interface k pluginům

3.3.1.1 Návrh interfacu

Správně navržený interface k pluginům je pro funkčnost a udržovatelnost celé komponenty pro načítání dat naprosto klíčový. Tento interface navíc musí být

kompletně celý dostupný pro zákazníky, aby mohli implementovat své vlastní pluginy. Z toho také vyplývá, že interface musí být co nejjednodušší, nejrobustnější a nesmí být závislý na vnitřní implementaci ani `GlobeTextureManageru`, ani pluginů.

U interfacu očekáváme časem jisté změny a novou funkcionalitu, zároveň je ale třeba zajistit zpětnou kompatibilitu. Chceme také, aby VBS Blue mohlo ve svém modulu obsahovat více než jeden integrovaný plugin. Těmto požadavkům nejlépe vyhovuje systém čistě virtuálních, objektově orientovaných interfaců. Princip je takový, že modul s pluginem obsahuje pouze jedinou funkci exportovanou z DLL. Tato funkce vrací ukazatel na hlavní třídu modulu, která implementuje čistě virtuální interface.

Uživatel knihovny[28] tedy definuje rozhraní pomocí sady headerů tříd s čistě virtuálními metodami. Plugin tyto třídy implementuje, vrátí ukazatel na instanci této třídy a uživatel volá tyto virtuální metody, jejichž implementace se nalézá v DLL modulu.

Zpětnou kompatibilitu zajistíme děděním. Každá třída interfacu dědí od třídy `IVersionedInterface`, která má jedinou metodu `GetVersion`. Návratová hodnota této metody odpovídá výčtovému typu verzí, jenž zase odpovídá názvu určité verze této třídy. Každá nová verze má jako svého předka předchozí verzi. Nejlépe se to uvádí na příkladu:

```
class IMapProvider : public IVersionedInterface
{
public:
  //! Interface versions
  enum Version
  {
    Initial, //! Initial version
    WithMask, //! Adds masking capability to map provider
    NumberOfVersions, //! Total number of versions, should be defined
        after all versions
    ActVersion = NumberOfVersions - 1 //! Actual version, must be last
  };
};

class IMapProviderInitial : public IMapProvider
{
public:
  //! Initial Version
  virtual int GetVersion() const override {return Initial;}
  //! Obtains some data (initial feature)
  virtual void GetData(...) = 0;
};

class IMapProviderWithMask : public IMapProviderInitial
```

[28] V tomto případě je v roli uživatele knihovny `GlobeTextureManager`.

```
{
public:
  //! Second version
  virtual int GetVersion() const override {return WithMask;}
  //! Obtains data mask (new feature)
  virtual void GetMask(...) = 0;
};
```

S instancí `IMapProvider` se pak pracuje takto:

```
IMapProvider* provider = ...; // assume we obtained the instance from
    the DLL
if (provider->GetVersion() >= IMapProvider::WithMask)
{
  // we can now use the instace as provider with masking capability
  IMapProviderWithMask* providerWithMask =
      (IMapProviderWithMask*)provider;
  providerWithMask->GetMask(...);
}
else
{
  // use fallback method for plugins without masking feature
  ...
}
```

Při vytváření headeru s veřejným interfacem je potřeba dodržovat jistá pravidla, která se vztahují na sdílený kód. Za prvé rozhodně nechceme zákazníkům poskytovat headery VBS Blue, neboť by toho příliš prozrazovaly o implementaci, čehož by mohla využít konkurence. Headery interfacu tedy musí tvořit zcela samostatný celek, který nikde neincluduje headery `GlobeTextureManageru` ani enginu. Opačně to možné ovšem je, a tak nezbytně nutné minimum struktur a enumerací přesuneme do headerů interfacu a ty potom budeme includovat v `GlobeTextureManageru`.

Stejné pravidlo se vztahuje i na třídy standardní knihovny, které na interfacu používat také nemůžeme. Různé kompilátory mají různé implementace standardní knihovny, které nejsou navzájem kompatibilní. Rozhodně tedy nechceme zákazníky při kompilaci pluginů omezovat na použití konkrétního kompilátoru.

Dalším pravidlem je, že paměť alokovaná v jednom modulu musí být tím samým modulem dealokována. Nelze tedy předávat ani alokované bloky paměti ve smart pointerech, ani automaticky alokovaná pole jako např. std::vector. V ideálním případě by `GlobeTextureManager` neměl používat žádnou paměť, která byla naalokována pluginem.

3.3.1.2 MapPlugin

Hlavním interfacem pluginu je `IMapPlugin`. Jeho rozhraní poskytuje přístup k jednotlivým poskytovatelům mapových dat a umožňuje správu paměti pluginu.

```
//! Plugin interface for getting map providers contained in this
    plugin
class IMapPluginInitial : public IMapPlugin
{
public:
  //! @brief Get list of map providers
  //! @param providers Array of IMapProvider* to be filled with map
      providers
  //! @param providersSize Size of providers array
  //! @param nProviders Output value indicating number of map
      providers. If the output value is greater than providersSize,
      you must call this function again with bigger array.
  virtual void GetMapProviders(IMapProvider** providers, int
      providersSize, int& nProviders) = 0;
  //! Free some memory and return amount of memory freed. Won't be
      called if MemoryControlled reports 0.
  virtual size_t FreeOneItem() = 0;
  //! How much memory can be freed using all FreeOneItem calls.
  virtual size_t MemoryControlled() const = 0;
  //! Prepare for shutdown - cancel all processes that can take long
  virtual void CancelAll() = 0;
  //! Initialize plugin in either live or offline binarization mode
  virtual void Init(bool binarize) {}
  //! Version 1
  virtual int GetVersion() const override {return Initial;}
};
```

Za zmínku zde stojí metoda `GetMapProviders`, která má poněkud krkolomný způsob použití právě kvůli výše zmíněnému pravidlu o alokaci paměti. Metoda má vracet pole poskytovatelů mapových podkladů. Volající funkce tedy poskytne předalokované pole a informaci o jeho velikosti. Pokud bylo poskytnuté pole dostatečně velké, bude vyplněno vyžádanými informacemi. To, jaká je potřebná délka pole, se volající dozví až po skončení této funkce. Pokud velikost pole v prvním volání nebyla dostatečná, je potřeba volání metody zopakovat se správnou velikostí, která byla zjištěna po prvním volání. Toto druhé volání však obvykle není potřeba.

3.3.1.3 MapProvider

Interface `IMapProvider` slouží k samotnému získávání dat z pluginu. Jeden `MapProvider` může poskytovat pouze jeden typ dat v jednom rozlišení. Plugin však může obsahovat více `MapProviderů`. Interface vypadá takto:

```
//! Map provider supplies map sources to DataManager
class IMapProviderInitial : public IMapProvider
{
public:
  //! Check if data are only supplementary (to be used with some real
      data)
  virtual bool IsSupplementary() = 0;
  //! Check if any data exists for given cube side area
  virtual bool DataExists(int side, double uBeg, double vBeg, double
      uEnd, double vEnd, int resolution) = 0;
  //! Get data for given cube side area, correctly allocated memory
      is given by the caller
  virtual void GetData(int side, double uBeg, double vBeg, double
      uEnd, double vEnd, int resolution, void* data) = 0;
  //! Get layer this map data belongs to
  virtual int GetLayer() const = 0;
  //! Get average resolution in pixels / km
  virtual float GetDetail() const = 0;
  //! Get type of data this provider provides
  virtual MapSourceType GetType() const = 0;
  //! Version 1
  virtual int GetVersion() const override {return Initial;}
};
```

Stěžejní funkcí je zde pochopitelně metoda `GetData`. Jejími vstupními parametry je oblast na krychli o libovolné velikosti, rozlišení cílové textury a ukazatel na předalokované pole, do kterého bude plugin data zapisovat. `GlobeTextureManager` dokáže získávat data pouze po blocích, které velikostí odpovídají segmentu, navíc pouze s konstantním rozlišením. Proč tedy po pluginech požadujeme libovolnou oblast a libovolné rozlišení? Rozlišení se totiž může na straně enginu časem změnit, některé typy textur by například mohly požadovat větší. Libovolnou oblast pro změnu požadujeme kvůli tomu, aby pluginy nebyly závislé na vnitřní implementaci VBS Blue, tedy na reprezentaci strany krychle pomocí quad-tree segmentů, a abychom pluginy oprostili od logiky overlapů.

Overlapy jsou přidané okraje textur, které obsahově patří do vedlejší textury. Tyto okraje jsou potřeba pokud chceme zobrazit povrch složený z více dlaždicově uspořádaných textur bez viditelných hran, které jsou způsobeny filtrováním textur. Šířka okraje je závislá na použitém typu filtrování. Bez filtrování nebo při použití bilineárního[29] filtrování je potřeba okraj pouze 1px, při trilineárním nebo anizotropním filtrování je potřeba okraj tak široký, kolik máme mipmap. V aktuální konfiguraci používá `GlobeTextureManager` 8 mipmap a rozlišení textury $2^8 = 256$. Potřebujeme tedy okraj 8 px, takže samotný obsah textury je redukován na rozlišení 240×240 px. Místo toho, aby tedy zákazník při implementaci pluginu musel řešit ještě navíc přidávání overlapů do textury, předáváme do metody `GetData` v parametrech oblast, do které je

už overlap započítán.

Další důležitou metodou je `DataExists`. Do té se předávají podobné parametry jako do `GetData`, na rozdíl od ní však vrací pouze stav, zda jsou v dané lokaci data přítomna, nebo ne. Tato funkce je důležitá zejména při binarizaci. Zjištěním, zda data existují se rozhoduje, jestli si z pluginu vyžádat nová, nebo použít data z hrubší úrovně detailů. Dále je důležitá i při samostatném průchodu binarizace za účelem uložení předzpracovaných dat na disk, kdy se používá pro zjištění, zda se má quadtree na daném místě procházet do hloubky.

3.3.2 PluginSDK

Při vývoji pluginů bylo zjištěno, že velké množství kódu se v různých pluginech opakuje. Pro usnadnění tvorby nových pluginů jak pro nás, tak pro zákazníky tyto a jiné užitečné funkce vyčleníme do knihovny PluginSDK. Jedná se o staticky linkovanou knihovnu užitečných funkcí pro tvorbu pluginů, která obsahuje například:

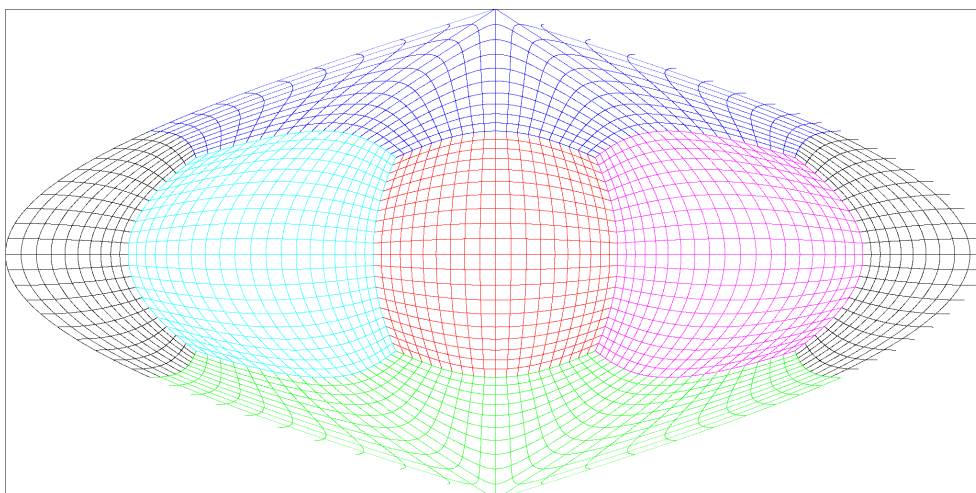
- funkce na převod mezi zeměpisnými a krychlovými souřadnicemi,
- funkce na získávání hodnot zemského geoidu,
- rychlý alokátor a správu paměti z VBS Blue,
- utility na logování, operace s filesystémem.

3.4 GDAL Plugin

GDAL plugin je sice až pátý plugin, který byl pro VBS Blue naimplementovaný, je však nejkomplexnější a jediný, který je univerzální a podporuje GPU akceleraci.

Hlavním účelem tohoto pluginu je mít možnost načítat data[30] z libovolného standardně používaného formátu mapového souboru s libovolnou projekcí. Implementovat svépomocí čtení desítek formátů souborů a projekcí nepřipadá v úvahu, naštěstí ale existuje knihovna GDAL, která se přesně touto problematikou zabývá. GDAL umí nejen číst všechny standardně používané formáty souborů, ale umí i převádět mezi různými standardními mapovými projekcemi. Krychlová projekce bohužel běžně používaná není, a tak ji GDAL nepodporuje. Můžeme však zdrojový soubor převést do meziprojekce, ze které

[29]Při bilineárním filtrování se použije UV koordinát vykreslovaného pixelu v textuře a zaokrouhlí se nahoru a dolů na celé texely. Výsledná barva vznikne lineární interpolací lineárně interpolovaných hodnot texeů v obou osách. Problém s navazováním při dlaždicovém uspořádaní je způsoben tím, že střed prvního texelu v každé ose má UV koordináty větší než nula, a pro správnou interpolaci hodnot mezi nulou a prvním texelem bychom potřebovali informaci z „mínus prvního“ texelu.



Obrázek 3.2: Transformace krychlových souřadnic do sinusoidové projekce

data poté převedeme do projekce krychlové. Tento druhý převod už má pevně stanovený algoritmus a můžeme jej provést na GPU.

3.4.1 Meziprojekce

Na první pohled je ideální meziprojekcí ekvidistantní válcová projekce, ve které je stejně poskytována většina zdrojových map. GDAL při převodu ze stejné projekce do té samé takový případ detekuje a ušetří čas, který zabírá převod dat. Ekvidistantní válcová projekce bohužel pro převod na GPU do krychlové není vhodná. Problém nastává obzvlášť v oblasti severního a jižního pólu. V ekvidistantní válcové projekci totiž platí, že čím blíže jsou data pólu, tím více jsou roztažená. Tento efekt je vidět na obrázku 2.10. Při převodu dat postupujeme po oblastech na krychli, které jsou na obrázku vyznačeny. Pro efektivní využití GPU je potřeba mít všechna data zpracovávané oblasti přítomna ve videopaměti. V případě oblastí u severního pólu jsou však oblasti natolik roztáhlé, že se do videopaměti nevejdou.

Tento problém je možné vyřešit použitím sinusoidové projekce místo ekvidistantní válcové. Sinusoidová projekce (viz obrázek 3.2) se blíže k pólům zužuje, čímž na rozdíl od ekvidistantní, která plochy u pólů zvětšuje a přidává tak množství zbytečných dat, realisticky zachovává plochy oblastí. Tím pádem je potřeba načíst stejné množství dat na rovníku i u pólů a data se do videopaměti bez problémů vejdou.

[30]Plugin prozatím bude načítat pouze výšková data. O možnost načítání jiného typu mapových zdrojů bude rozšířen až v době, kdy budeme mít k dispozici i jinou mapu povrchů a biomovou mapu než se satelitu MODIS.

Postup, který bude plugin při zjišťování přítomnosti dat vykonávat, je tedy následující:

- získat seznam potřebných dlaždic sinusoidové mapy:
  - vypočítat sinusoidové souřadnice každého pixelu výsledné textury,
  - vypočítat z každé souřadnice, do které dlaždice sinusoidové mapy patří,
  - získat pouze unikátní dlaždice,
- zkontrolovat, zda máme dostupná data alespoň pro jednu dlaždici.

Získávání souřadnic každého pixelu se může zdát jako zbytečné, žádný jiný algoritmus pro naprosto přesné získání dlaždic však nebyl nalezen ani pro sinusoidovou, ani ekvidistantní válcovou projekci. Výsledek této operace však můžeme použít při získávání samotných dat, které v případě použití probíhá následovně:

- získat seznam potřebných dlaždic sinusoidové mapy z předchozí operace kontroly přítomnosti dat,
- načíst data potřebných dlaždic do paměti:
  - získat seznam zdrojových souborů, které zasahují do oblasti dlaždice,
  - transformovat data zdrojových souborů do sinusoidové projekce a zapsat do dlaždice,
- transformovat načtená data do krychlové projekce.

3.4.2 Mipmaping dlaždic

Plugin musí být schopen zvládat i případy, kdy `GlobeTextureManager` chce získat data pro úroveň detailu 0, což je celá strana krychle, a tedy šestina planety. Co ale dělat v případě, kdy jsou zdrojová data ve vysokém rozlišení a mají pro dotazovanou oblast velikost desítek GB? Zde nám pomůže stejný koncept mipmap jako ten, který používáme pro renderování.

Pokud tedy budeme transformovat nějakou velkou oblast planety, spočítáme, v kolikáté úrovni mipmapy se požadovaná oblast nachází a vytvoříme ji rekurzivně vždy ze čtyř oblastí nižší úrovně mipmapy. K tomu, aby při tomto rekurzivním procházení nedošla paměť, využíváme třídu `TileCache`.

3.4.3 TileCache

Nejpomalejším článkem pluginu bude načítání a transformace dat knihovnou GDAL. Implementace knihovny není rozhodně špatná, ale vzhledem k její univerzálnosti jednoduše nemůže pracovat tak rychle jako zbytek tohoto pluginu. Výsledky transformací provedených GDALem se tedy cachují ve třídě `TileCache`. Cache je sdílená pro všechny úrovně detailů a mipmap. Klíč k datům v cachi má tedy následující strukturu:

```
//! Key for uniquely addressing sinusoidal tiles
struct TileKey
{
  //! Level of detail of the source data
  int _level;
  //! Mipmap level
  int _mip;
  //! X index of the tile within current level/mipmap
  int _x;
  //! Y index of the tile within current level/mipmap
  int _y;
}
```

Ke GDAL datům se tedy nepřistupuje přímo, ale vždy prostřednictvím `TileCache` metodou `GetTile`, která přijímá pouze jediný argument typu `TileKey`. Cache se poté postará o vrácení patřičných dat, ať už pocházejí odkudkoli, což může být:

- načtení ze zdrojových dat knihovnou GDAL,
- sloučení ze čtyř dlaždic nižší úrovně mipmapy,
- existující data cachovaná v paměti RAM,
- data cachovaná na pevném disku.

Nejvyšší přednost mají pochopitelně data, která jsou již nacachovaná v RAM, protože ta jsou dostupná téměř okamžitě. Pokud nejsou k dispozici, `TileCache` zkontroluje cache na disku, odkud se data načítají nekomprimovaně pouze jako surové pole. Přístupová doba se tedy pohybuje v desítkách milisekund. Pokud nejsou cachovaná data dostupná ani na disku, nezbývá než je vytvořit ze zdrojových dat. Pokud vytváříme mipmapu úrovně větší než nula, získáme data rekurzivně sloučením z nižších úrovní. Pouze na úrovni nula získáváme data načítáním a transformací z knihovny GDAL, čímž minimalizujeme zpomalení, které způsobuje.

3.4.4 GPU akcelerace obecně

Přes to, že požadavkem na GDAL plugin je GPU akcelerace pouze technologií CUDA, bylo by vhodné mít možnost později přidat akceleraci i jinými

technologiemi nebo použít CPU. Proto není CUDA integrovaná do pluginu přímo, ale jako samostatný, zaměnitelný modul s pevně daným interfacem. Vedlejší efekt je také to, že ke kompilaci GDAL pluginu nebude zapotřebí mít nainstalované SDK všech podporovaných technologií GPU akcelerace. To je obzvlášť výhodné ve firemním prostředí, kde na dalším vývoji GDAL pluginu nebude pracovat pouze jeden člověk.

```
//! Represents a generic algorithm used for transforming data from
    sinusoidal to cubic projection
class IGPUAlgorithm
{
public:
  //! Returns whether this algorithm is supported on local machine
  virtual bool IsSupported() = 0;
  //! Create a transformation operation
  virtual IGPUTranformOp* CreateTransOp(IGPUDataProvider* provider,
      const TransformOpParams& params) = 0;
  //! Release memory of a transformation operation
  virtual void FreeTransOp(IGPUTranformOp* op) = 0;
  //! Virtual destructor is needed as the instance may be deleted via
      the interface type
  virtual ~IGPUAlgorithm() {}
};

//! A single transformation from sinusoidal to cubic projection
class IGPUTranformOp
{
public:
  //! Check whether there are some data available in the region
      defined by this transform operation
  virtual bool HasData() = 0;
  //! Transform the data and fill it into the output parameter
  virtual void FillData(float* data) = 0;
  //! Virtual destructor is needed as the instance may be deleted via
      the interface type
  virtual ~IGPUTranformOp() {}
};

//! This interface provides source data for IGPUTransformOp
class IGPUDataProvider
{
  public:
  //! Check whether source data are available for the specified
      source tile
  virtual bool DataExists(int tx, int ty, int mip) = 0;
  //! The source tile's data
  virtual void GetTileData(float* data, int tx, int ty, int mip) = 0;
};
```

GDAL plugin při spuštění prohledá adresář s pluginy a vybere první, který vrátí `IsSupported == true`. Pořadí určuje uživatel názvem souboru pluginu.

Interface se využívá následovně: GDAL plugin při volání metody `DataExists` vytvoří novou operaci metodou `IGPUAlgorithm::CreateTransOp`, které předá v parametrech oblast planety, kterou zpracováváme, a „poskytovatele dat", ze kterého může operace čerpat zdrojová data. Na tuto operaci se následně zavolá metoda `HasData`. Pokud vrátí `true`, bude následovat i metoda `GetData`, ve které se zavolá na operaci metoda `IGPUTranformOp::FillData`. Díky tomu, že se metody volají na objekt transformační operace, je možné využít některé výsledky z `HasData` v metodě `FillData`.

3.4.5 CUDA akcelerace

Samotná CUDA akcelerace je tedy realizována v modulu CudaAlgorithm.dll. Použité bylo nejnovější CUDA SDK 7.5. Projekt ovšem cílí i na starší hardware, konkrétně na `compute_30,sm_30`.

Metodu `IsSupported` implementujeme tak, že zkontrolujeme počet a vlastnosti CUDA zařízení funkcemi `cudaGetDeviceCount` a `cudaGetDeviceProperties`. Pokud vrátí úspěšný status a alespoň jedno zařízení má v properties podporu SM 3.0, je CUDA podporována a můžeme tento plugin využít.

Metodu `CUDATransformOp::HasData`[31] CUDA plugin implementuje tak, že ve videopaměti vytvoří 2D pole o stejném rozlišení jako cílová textura a toto pole naplní sinusoidovými souřadnicemi mapování každého pixelu textury. O samotný výpočet transformace se starají metody z PluginSDK[32]. Ukazatel na toto pole se udržuje v objektu `CUDATransformOp` pro pozdější použití. Dále se naalokuje ještě jedno pole stejné velikosti, do kterého se z výsledku z prvního pole spočítají celočíselné souřadnice sinusoidových dlaždic, do nichž každý pixel spadá.

Nyní máme v paměti zařízení[33] poměrně velké pole s mnoha podobnými hodnotami, ze kterých nás zajímají pouze hodnoty unikátní. Naštěstí máme k dispozici knihovnu Thrust, kde jsou už běžně používané algoritmy optimálně pro CUDU implementovány. Pomocí sekvence funkcí `thrust::sort` a `thrust::unique` získáme v paměti zařízení malé pole unikátních souřadnic sinusoidových dlaždic, které zkopírujeme do RAM a přes `IGPUDataProvider` už jednoduše zkontrolujeme, zda alespoň pro jednu dlaždici máme dostupná nějaká data.

[31] CUDA implementace interfacu `IGPUTranformOp`

[32] Jejich použití v CUDA je popsáno v kapitole 3.4.5.4.

[33] Vidopaměť se v manuálech a CUDA API označuje jako „device memory", tedy „paměť zařízení".

3.4.5.1 Využití knihovny Thrust

Při implementaci byla využita knihovna Thrust, konkrétně operace `sort`, `unique` a struktury `host_vector` a `device_vector`. Při implementaci jsme ale narazili na nepříjemnou vlastnost této knihovny, která je sice zdokumentovaná, nicméně působí značné problémy a je těžko odhalitelná. Od verze nvcc 4.2 totiž kód Thrustu v debugovací verzi nepracuje spolehlivě.

V projektu je debugování potřeba, naštěstí je možné tento problém obejít: nejprve rozdělíme kód na headery a cpp/cu. K tomu je potřeba použít při kompilaci nvcc parametr -rdc[34] a při linkování nvlink -dlink[35]. Veškeré funkce a metody, které pak obsahují problémový Thrust kód přesuneme do samostatného *.cu souboru, u kterého generování debugovacích informací vypneme. Tak zůstane zbytek projektu bez problémů debugovatelný.

3.4.5.2 Cachování načtených dlaždic

Před spuštěním kernelu, který vytváří finální data v krychlové projekci, je potřeba mít v paměti zařízení načtené všechny potřebné sinusoidové dlaždice. Jedna dlaždice přitom typicky zasahuje do několika krychlových segmentů, takže nahrávat je do videopaměti opakovaně, je plýtvání výkonem.

Zavedeme tedy další cache, tentokrát je však její obsah uložen ve videopaměti. Sinusoidové dlaždice udržujeme načtené napříč jednotlivými spuštěními transformačního kernelu a využijeme veškerou dostupnou paměť zařízení. Když paměti už není dostatek a alokace selže, postupně uvolňujeme nejdéle nepoužívané dlaždice, dokud alokace nebude úspěšná. Touto cachí tedy minimalizujeme zbytečnou komunikaci mezi CPU a GPU.

3.4.5.3 Alokace paměti

Cache na sinusoidové dlaždice však není jediným zdrojem alokací na GPU. Je potřeba v poli předávat dlaždice na GPU, ukládat mezivýsledky apod. Můžeme sice nastavit určitou rezervu, aby cache nevyplnila celou VRAM, ale takový přístup by nebyl nikdy stoprocentně bezpečný. Místo toho zavedeme třídu `CUDAAllocator`, která se centrálně stará o alokaci a uvolňování paměti. Pokud není pro alokaci dostatek volné paměti, způsobí tato třída uvolnění několika dlaždic cache, dokud se jí neuvolní dostatečné množství.

Menší problém nastává, pokud paměť nealokujeme přímo z vlastního kódu, ale z knihovny Thrust. Naštěstí však má podporu vlastních alokátorů stejně jako knihovna STL. Při deklaraci `device_vector`u v template argumentech specifikujeme tedy nejen typ obsahu, ale i použitý alokátor, který musí přesně vyhovovat interfacu alokátoru Thrust. Interně je však jednoduše implementován voláním metod třídy `CUDAAllocator`.

[34] Generování Relocatable Device Code. V podstatě jde o kompilaci do object filů podobně jako v C++. Před jejich použitím je potřeba je slinkovat.

[35] Device Link

3.4.5.4 Sdílení kódu mezi CUDA a C++

PluginSDK již obsahuje funkce na transformaci mezi sinusoidovou a krychlovou projekcí. Bylo by tedy vhodné je využít, aby se zamezilo duplikaci kódu. CUDA SDK naštěstí od verze 5.0 umožňuje díky výše zmíněnému RDC a device linku vytvářet statické knihovny CUDA funkcí. Je tedy pouze potřeba zařídit, aby PluginSDK exportovalo symboly nejen pro host kód, ale i pro device kód. To by však znamenalo, že by PluginSDK samotné nebylo kompilovatelné bez CUDA SDK.

Vytvoříme tedy samostatný projekt CUDAPluginSDK, který bude sdílet zdrojový kód s PluginSDK, bude však produkovat jinou staticky linkovanou knihovnu obsahující CUDA symboly. Využijeme preprocesorových maker k tomu, aby se v PluginSDK kód zkompiloval pro CPU a v CUDAPluginSDK pro GPU. CUDA naštěstí podporuje téměř kompletní syntax C++, takže není nutné dělat příliš mnoho změn v kódu, aby tento mechanismus fungoval. Výsledný kód vypadá následovně:

Listing 3.1: geoMath.h

```
#if _CUDA
  #define SHAREDCALL __device__
#else
  #define SHAREDCALL
#endif

class GeoMath
{
public:
  //! Calculate position on the Ellipsoid's surface from CubeCoord
  SHAREDCALL static Vector3D ToEllipsoidSurface(const CubeCoord&
      coord);
  //! Transform cube surface position to ellipsoid surface position
  SHAREDCALL static Vector3D CubeToEllipsoid(const Vector3D& pos);
  ...
}
```

Listing 3.2: geoMathCuda.cuh

```
#pragma once
#define _CUDA 1
#include <geoMath.h>
```

Největší zaznamenaný problém byl ten, že CUDA nepodporuje statické třídní konstanty. Musely proto být nahrazeny konstantami globálními, které se manuálně inicializují před prvním spuštěním kernelu.

3.4.5.5 Spouštění kernelů

GDAL Plugin s CUDA akcelerací není cílen na konkrétní hardwarovou konfiguraci ani GPU. Nastává tedy problém s nastavením velikosti a počtem bloků pro spuštění kernelu tak, aby fungoval na každé kartě, ale aby zároveň využíval každý model GPU co nejefektivněji. K tomu využijeme funkci `cudaOccupancyMaxPotentialBlockSize`, která do druhého parametru vrátí maximální velikost bloku, kterou je aktuálně používané GPU pro daný kernel schopné zvládnout. Tím maximalizujeme využití GPU. Volání této funkce není úplně levné, přestože vrací pokaždé stejnou hodnotu. Zařídíme tedy, aby se volala pouze jednou. Výsledné použití vypadá následovně:

```
// determine block size for CalculateTransformKernel
static int calcTransformThreads = -1;
if (calcTransformThreads == -1)
  cudaOccupancyMaxPotentialBlockSize(&minGridSize,
      &calcTransformThreads, (void*) CalculateTransformKernel, 0, 0);
// determine grid size (pixelCount / calcTransformThreads rounded up)
int blocks = (pixelCount + calcTransformThreads - 1) /
    calcTransformThreads;
// calculate CubeCoord -> Sinusoidal mapping in CUDA kernel
CalculateTransformKernel<<<blocks,
    calcTransformThreads>>>(thrust::raw_pointer_cast(_geoTransform.data()),
    _params);
```

3.4.5.6 Optimalizace

K dosažení maximální efektivity CUDA akcelerace byl výsledný modul profilován nástrojem NVIDIA Visual Profiler. Na základě analýzy provedené tímto nástrojem bylo zjištěno několik možných optimalizací, kterými se podařilo zvýšit využití GPU během binarizace s předem vytvořenou přednačtenou GDAL cachí z přibližně 19 % na 70 %.

Nejvíce času se při binarizaci tráví na hlavním vlákně ve funkcích CUDA runtime API, zejména alokacích a dealokacích paměti zařízení. Tyto alokace jsou součástí příprav ke spuštění kernelů, například alokace vstupních a výstupních polí. Bylo by tedy vhodné, aby se pole naalokovala pouze jednou při spuštění aplikace, a používala se opakovaně. Problém však je, že proces binarizace je silně multithreadový, pole jsou tedy potřeba z více vláken naráz. Každé vlákno má tedy vlastní instance opakovaně používaných polí a proměnných.

Je více cest, jak toho docílit, byla však zvolena metoda za pomoci thread-local proměnných. Nejedná se o nejelegantnější řešení, je však nejefektivnější a vyžaduje nejméně změn v kódu. Pro jednoduchost jejich používání byla vytvořena makra `DECL_THREAD_PERSISTENT`, kterým se deklaruje použití proměnné v globálním namespacu a `USE_THREAD_PERSISTENT`, kerým se uvozuje její první použití ve funkci. Proměnná bude automaticky inicializována při

prvním použití daným vláknem a přetrvá napříč jednotlivými voláními metod, podobně jako by šlo o proměnnou globální nebo statickou.

Ani tím však opakované alokace zcela nevymizely. Metody `thrust::sort` a `thrust::unique` totiž nepracují in-place a alokují pomocná pole, kterým navíc nelze specifikovat vlastní alokátor. To již dříve způsobovalo problémy při cachování dlaždic ve videopaměti – musela se udržovat rezervní volná paměť, aby selhání alokace v knihovně Thrust nezpůsobilo pád aplikace. Kombinace `thrust::sort` a `thrust::unique` navíc tvořila většinu výpočetního času.

Vytvořili jsme proto nový UniqueKernel, který je schopen pracovat s nesetříděnou posloupností a získat z ní unikátní hodnoty. Jeho implementace je optimalizována pro malé množství unikátních hodnot v dlouhém poli, což je právě tento případ. K získání unikátních hodnot tento kernel navíc nepotřebuje žádnou dodatečnou paměť, vše probíhá in-place.

Další výhodu, kterou použití vlastního kernelu oproti knihovně Thrust přináší, je menší počet synchronizačních bodů. Thrust sice lze používat i v asynchronním režimu, použitím vlastního kernelu však můžeme spustit posloupnost `CalculateTransformKernel`, `GetTilesKernel` a `UniqueKernel` a až poté vynutit synchronizaci s GPU, což podstatně snižuje overhead.

Standardní chování vícevláknové CUDA aplikace je řadit příkazy ze všech vláken do jedné interní fronty (streamu). Pro dosažení nejvyššího možného výkonu lze streamy používat i manuálně, jedná se ovšem o poměrně náročný proces náchylný na chyby. Kompilačním parametrem `--default-stream per-thread` se zajistí, že každé vlákno bude mít standardně přiřazen svůj vlastní stream a nebudou se navzájem blokovat.

Poslední optimalizací bylo přidání zvláštního streamu pro načítání zdrojových dlaždic do videopaměti. Jedna dlaždice má velikost přesně 16 MB a přenést ji na GPU již trvá nezanedbatelnou dobu. Přenos byl tedy změněn na asynchronní, aby neblokoval ostatní vlákna.

Původní a optimalizovanou timeline progamu NVIDIA Visual Profiler lze najít na přiloženém CD v adresáři profilerResults.

3.5 Stav implementace

Vzhledem k rozsahu projektu a tomu, jak se postupně vyvíjely priority ve VBS Blue, nebyly všechny navržené součásti zcela implementovány nebo dokončeny. Lokální cachovací plugin, mapový server a síťový plugin prozatím nebyly realizovány vůbec, neboť momentálně nejsou potřeba. Interface pro binarizační pluginy není zcela dokončen, zatím totiž postačuje specifický interface pro jediný implementovaný binarizační plugin.

Na implementaci komponenty pro načítání dat se bude pracovat i v budoucnu, ovšem už mimo rámec této práce. Bude rozšířena například i o načítání vektorových dat, tedy silniční sítě, apod.

Kapitola **4**

Výsledky

Veškerá měření byla provedena na výkonném PC s herním GPU. Většina výpočtů, které v CUDA kernelech provádíme, však pracuje v double precision, která je na herních kartách výrazně pomalejší než na kartách řady Tesla. Jak ale ukazují měření, nejsme limitováni výpočetním výkonem této karty. Konkrétní hardwarová konfigurace je následující:

Tabulka 4.1: Hardwarová konfigurace testovacího PC

CPU	Intel Core i7-6700K, Quad Core, 4.0GHz (4.2GHz TB)
Deska	MSI Z170A PC Mate
RAM	Kingston 16GB KIT DDR4 3000MHz CL15
GPU	ASUS TURBO-GTX970-OC-4GD5
Disk 1	SSD ADATA Premier Pro SP900 256GB
Disk 2	HDD Seagate Barracuda 7200.14 2000GB
Disk 3	External HDD WD My Passport 2000GB, USB 3.0

Na interní disk 1 jsou zapisována výstupní data, neboť se jedná o velké množství malých souborů, se kterými si nejlépe poradí SSD. `TileCache` je tvořena malým množstvím 16MB souborů, je proto umístěna na disku 2. Zdrojová data map jsou umístěna na externím disku 3.

4.1 Rychlost načítání

Požadavek na rychlost načítání, který byl zmíněn v kapitole 1.2.6, totiž že se nesmí ztrácet detaily, když se kamera pohybuje ve výšce 1km rychlostí Mach 2, byl splněn. Nejjemnější detaily se začaly ztrácet až při rychlosti okolo 10000 m/s (Mach 29). Stanovení konkrétní přesné hranice není možné kvůli velkým výkyvům v rychlosti načítání a nemožnosti toto měření automatizovat.

Měření bylo provedeno průletem konstantní rychlostí nad Alpami, kde jsou přítomna data ve vysokém rozlišení 1111 px/km^2. Tuto část by sice bylo možné

automatizovat, ale žádným exaktním testem nelze ověřit, zda je rychlost načítání dat dostatečná.

4.2 Rychlost binarizace

4.2.1 Menší zdrojová data

Byl změřen čas, který trvala transformace výškové mapy pokrývající svět o rozlišení 0,25 px/km^2. Zdrojový soubor je v nekomprimovaném formátu GeoTiff, data o velikosti 4,63 GB. Výstupem je bez ohledu na použitý transformační plugin přibližně 30 000 souborů o celkové velikosti 150 MB. Toto podstatné snížení velikosti je způsobené binarizačním pluginem, který používá silnou ztrátovou kompresi[9]. Následující tabulka zobrazuje čas potřebný ke zpracování:

Tabulka 4.2: Doba zpracování menšího datasetu

Test	Doba běhu [min]
Space Engine[36]	39,40
CPU bez cache	22,55
CPU s cachí	13,08
CUDA bez cache	17,03
CUDA s cachí	6,01

Jak je vidět, použití GPU akcelerace má na výkon pozitivní vliv, ne však tak velký, jak bychom na první pohled od masivně paralelního výpočtu na GPU očekávali. Je třeba si však uvědomit, že nejde pouze o čas výpočtu transformací, ale i načtení zdrojových dat, vygenerování mipmap, komprese a zápisu na disk.

Porovnáním doby zpracování bez a s cachí lze vypozorovat, že overhead, který je tvořen čtením dat GDALem a mezitransformací do sinusoidové projekce, tvoří pro tuto mapu přibližně 10 minut. Při použití CUDA akcelerace se tedy jedná o 59 % veškerého času.

4.2.2 Větší zdrojová data

Dále byl změřen čas zpracování větších zdrojových dat. Jedná se o původní dataset SRTM od NASA o rozlišení 123 px/km^2 a velikosti 37,7 GB v 14 050 souborech. Výstupem transformace je 955 247 souborů o velikosti pouhých 3,06 GB. Vzhledem k době, kterou tato operace trvá, nebyla měřena doba, kterou trvá transformace na CPU bez použití cache. Porovnání času zpracování se Space Enginem také není možné, protože převod takto velkého datasetu nezvládá.

Tabulka 4.3: Doba zpracování většího datasetu

Test	Doba běhu [hod]
CPU s cachí	9,01
CUDA bez cache	5,48
CUDA s cachí	1,66

4.3 Efektivita kernelů

CUDA algorithm plugin obsahuje čtyři CUDA kernely: `CalculateTransformKernel`, `GetTilesKernel`, `GetDataKernel` a `UniqueKernel`. Podrobné zprávy o jejich efektivitě, vygenerované nástrojem NVIDIA Visual Profiler, je možné najít na přiloženém CD v adresáři `profilerResults`.

Kernely byly během vývoje profilovány a byly provedeny pouze optimalizace, které nevyžadují podstatný zásah do návrhu aplikace a které výrazně nezhorší čitelnost kódu. Celkové využití GPU v nejoptimističtějším případě, kdy jsou všechna zdrojová data přítomna v operační paměti, tvoří až 70 %.

Tabulka 4.4: Vytížení GPU CUDA kernely

Kernel	Průměrná doba [μs]	Dosažená vytíženost [%]
CalculateTransformKernel	365	86,8
GetTilesKernel	63	84,3
GetDataKernel	150	48,8
UniqueKernel	230	50,0

4.3.1 CalculateTransformKernel

`CalculateTransformKernel` pracuje téměř optimálně, provádí totiž takřka ideální úlohu pro GPU – potřebuje minimální množství vstupních dat, ze kterých provádí netriviální výpočet, který je téměř bez větvení. Limitujícím faktorem tohoto kernelu je přílišné vytížení instrukční jednotky pro výpočty typu `double`. Tomu se však nelze nijak vyhnout, účelem je totiž počítat co nejpřesněji a jiný typ tudíž využít nelze.

Analýza také ukazuje, že se celý algoritmus nevejde do registrů a swapuje do lokální paměti. Nastavení počtu použitých registrů by sice mohlo mít pozitivní vliv na výkon jednotlivých vláken, omezovalo by to však jejich maximální počet.

Také schéma přístupu do paměti by bylo možné optimalizovat. Kernel podává nejlepší výkon pokud při přístupu do paměti i-té vlákno ve warpu přistupuje na i-tý word[37] v cachové řádce.[10] Aktuální struktura paměti, do

[37] V CUDA má word 32 bitů.

které kernel zapisuje, je kvůli tomu, že využíváme stejné struktury a kód jako na CPU, rozložena následovně:

```
struct
{
  double _x;
  double _y;
} data[65536];
```

Aby byl přístup do paměti optimální, museli bychom využívat takovéto struktury:

```
struct
{
  word _xHigh[65536];
  word _yHigh[65536];
  word _xLow[65536];
  word _yLow[65536];
} data;
```

Přístup kernelu do paměti by byl v takovém případě optimální, přibyla by však režie skládání a rozkládání `double` hodnot na dvě `word` hodnoty a jejich adresování v poli. Výše uvedená struktura je navíc zjednodušená, v praxi je pole alokováno dynamicky. Celkově však tento kernel pracuje dostatečně efektivně, aby takto dramatická optimalizace nebyla potřeba.

4.3.2 GetTilesKernel

Analýza `GetTilesKernel`u opět poukazuje na problém se schématem přístupu do paměti, což je přímý důsledek toho, jak paměť zapisuje `CalculateTransformKernel`. Dále analýza odhaluje v podstatě identické problémy jako v případě předchozího kernelu.

Nejlepší optimalizací by bylo tento kernel zrušit a sloučit s `CalculateTransformKernel`, kde zdrojové hodnoty jsou již v registrech a stačí z nich pouze vypočítat a zapsat hodnoty výstupní.

4.3.3 GetDataKernel

Analýza `GetDataKernel`u odhaluje problém s počtem použitých registrů. Na jednu stranu je jich příliš málo, což způsobuje zvýšenou komunikaci s pamětí, na druhou stranu je jich příliš mnoho, což zabraňuje tomu, aby se využila všechna dostupná vlákna. Situaci navíc komplikuje to, že tento kernel spouštíme ve 2D kvůli lepšímu využití texturovací cache. Průměrná doba jeho běhu však nenaznačuje, že by bylo potřeba jej dramaticky optimalizovat.

4.3.4 UniqueKernel

Není žádným překvapením, že `UniqueKernel` nevyužívá GPU příliš efektivně. Za prvé je navržen tak, aby se spouštěl pouze v jednom bloku, a za druhé pracuje na principu paralelní redukce, u které v každém kroku pracuje pouze polovina vláken než v předchozím, dokud nepracuje pouze jedno. Přesto dosáhl poměrně dobrého vytížení – 50 %.

Závěr

Cílem této práce bylo navrhnout a implementovat komponentu pro načítání terénních dat. Úspěšně jsme implementovali všechny její podstatné části a integrovali ji do simulátoru VBS Blue. Implementace funguje bez problémů a v rámci předem stanovených parametrů. Naše metoda není oproti podobným implementacím omezená na jeden konkrétní formát vstupních ani výstupních dat, je uživatelsky rozšiřitelná a dostatečně flexibilní.

Rozšiřitelnost jsme dokázali implementací vlastního pluginu, který přidává podporu pro načítání desítek zdrojových formátů podporovaných knihovnou GDAL. Nepříliš ideální výkon této knihovny jsme se pokusili kompenzovat akcelerací výpočtů na GPU pomocí technologie CUDA. Jisté zrychlení se sice dostavilo, nezměnilo ovšem fakt, že program může být pouze tak výkonný, jako jeho nejpomalejší článek. Vývoj CUDA implementace však nebyl zcela zbytečný. Díky modularitě, se kterou byla tato část implementována, by mělo být možné ji s minimálním množstvím změn využít v jiném pluginu, kde bude pracovat efektivněji.

Kvůli rozsahu této práce nebyly všechny navržené části implementovány, nebo byly zpracovány pouze částečně. Veškeré důležité části pro chod komponenty jsou však implementovány a pracují dobře. Na komponentě se v rámci projektu VBS Blue neustále pracuje a v budoucnu bude dále rozšiřována o další funkcionalitu.

Literatura

[1] NVIDIA Corporation: *NVIDIA® PhysX® SDK Documentation*. Dostupné z: http://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Index.html

[2] Kemen, B.: A new terrain mapper tool. Technická zpráva, Outerra, 2011. Dostupné z: http://outerra.blogspot.cz/2011/04/new-terrain-mapper-tool.html

[3] Chan, F.; O'Neill, E.; DIV., C. S. C. S. S. M. S. S.; aj.: *Feasibility Study of a Quadrilateralized Spherical Cube Earth Data Base: Final Report*. EPRF technical report, Computer Sciences Corporation, System Sciences Division, 1975. Dostupné z: https://books.google.cz/books?id=94OWNwAACAAJ

[4] Rincón, C.: Quadrilateralised spherical cube projection. 2 2013. Dostupné z: https://github.com/cix/QuadSphere/blob/master/lib/quad_sphere/csc.rb

[5] Tatarchuk, N.: Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-295-X, s. 63–69, doi:10.1145/1111411.1111423. Dostupné z: http://doi.acm.org/10.1145/1111411.1111423

[6] Cozzi, P.; Ring, K.: *3D Engine Design for Virtual Globes*. CRC Press, první vydání, June 2011, ISBN 978-1568817118.

[7] Featherstone, W.; Claessens, S.: Closed-form transformation between geodetic and ellipsoidal coordinates. *Studia Geophysica et Geodaetica*, ročník 52, č. 1, 2008: str. 2, ISSN 0039-3169, doi:10.1007/s11200-008-0002-6. Dostupné z: http://dx.doi.org/10.1007/s11200-008-0002-6

[8] Borkowski, K. M.: Transformation of Geocentric to Geodetic Coordinates without Approximations. *Astrophysics and Space Science*, ročník 139, 7 1987: s. 1–4. Dostupné z: http://www.astro.uni.torun.pl/~kb/Papers/ASS/Geod-ASS.htm

[9] Lašan, M.: *Height map compression techniques*. Diplomová práce, Univerzita Karlova v Praze, 2016.

[10] *CUDA Toolkit Documentation v7.5*. 9 2015. Dostupné z: http://docs.nvidia.com/cuda/index.html

Příloha **A**

Seznam použitých zkratek

AMP Advanced Massive Parallelism

API Application programming interface

BISim Bohemia Interactive Simulations

CIGI Common Image Generator Interface

CUDA Compute Unified Device Architecture

DLL Dynamic Link Library

DTED Digital Terrain Elevation Data

GC Garbage Collector

GDAL Geospatial Data Abstraction Library

GIS Geografic Information System

GPGPU General Purpose Graphic Processing Unit

GPU Graphic Processing Unit

IG Image Generator

LRU Least Recently Used

MODIS Moderate Resolution Imaging Spectroradiometer

MS Microsoft

PNG Portable Network Graphics

RAM Random Access Memory

RV Real Virtuality

SDK Software Development Kit

SM Shader Model

SRTM Shuttle Radar Topography Mission

STL Standard Template Library

VBS Virtual Battlespace

VRAM Video RAM

Příloha B

Obsah přiloženého CD

```
readme.txt ........................... stručný popis obsahu CD
├── profilerReports .................. zprávy o efektivitě kernelů
├── profilerTimelines ................ výsledky aplikace NVIDIA Visual Profiler
│   ├── timelineOriginal.nvprof ...... timeline před optimalizacemi
│   ├── timelineOptimized.nvprof ..... timeline po optimalizacích
│   ├── kernelAnalysis.nvprof ........ timeline obsahující analýzu jednotlivých
│   │                                  kernelů
│   └── timelineNoCache.nvprof ....... timeline bez předgenerované GDAL cache
├── src
│   ├── impl ......................... zdrojové kódy implementace
│   │   ├── GDALPlugin
│   │   ├── PluginSDK
│   │   └── CUDAPluginSDK
│   └── thesis ....................... zdrojová forma práce ve formátu LaTeX
└── text ............................. text práce
    ├── thesis.pdf ................... text práce ve formátu PDF
    └── thesis.ps .................... text práce ve formátu PS
```