VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

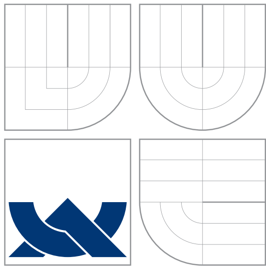
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

RENDEROVÁNÍ ROZSÁHLÉHO TERÉNU

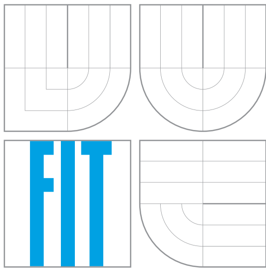
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MARTIN MARUŠIČ
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

RENDEROVÁNÍ ROZSÁHLÉHO TERÉNU

RENDERING OF LARGE SCALE TERRAIN

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE / AUTHOR: Bc. MARTIN MARUŠIČ

VEDOUCÍ PRÁCE / SUPERVISOR: Ing. JAROSLAV PŘIBYL

BRNO 2010

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2009/2010

Zadání diplomové práce

Řešitel: **Marušič Martin, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Renderování rozsáhlého terénu**
Rendering of Large Scale Terrain
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte moderní algoritmy pro renderování terénů.
2. Zvolte si jeden z algoritmů a vysvětlete detailně jeho princip.
3. Popište publikované optimalizace alogoritmu.
4. Implementujte zvolený algoritmus.
5. Zvažte možnost realizace části systému na GPU.
6. Zhodnoťte dosažené výsledky, navrhněte možnosti pokračování projektu a vytvořte reprezentační plakátek.

Literatura:

- dle pokynu vedoucího

Při obhajobě semestrální části diplomového projektu je požadováno:

- Body 1 - 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Přibyl Jaroslav, Ing.**, UPGM FIT VUT
Datum zadání: 21. září 2009
Datum odevzdání: 26. května 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt
Tato práce pojednává o renderování rozsáhlého terénu. V první části je popsána teorie renderování terénu a konkrétních LOD technik. Dále jsou v této části stručně popsány tři zajímavé moderní algoritmy. Hlavní důraz je kladen na popis algoritmu Geometry Clipmaps i s jeho optimalizací GPU-Based Geometry Clipmaps. Implementace tohoto optimalizovaného algoritmu je podrobně popsána v následujících kapitolách. Hlavní výhodou tohoto algoritmu je inkrementální aktualizace vertex textur a přesun zátěže z CPU na GPU. Poslední část je zaměřena na testování výkonu implementovaného algoritmu a porovnává výsledky s různým nastavením kvality výsledného terénu.

Abstract
This thesis deals with rendering of large scale terrain. The first part describes theory of terrain rendering and particular level of detail techniques. Three modern intriguing algorithms are briefly depicted after this theoretical part. Main work insists on description of Geometry Clipmaps algorithm along with its optimized version GPU-Based Geometry Clipmaps. Implementation of this optimized algorithm is depicted in detail. Main advantage of this approach is incremental update of vertex data, which allows to offload overhead from CPU to GPU. In the last chapter performance of my implementation is analysed using simple benchmark.

Klíčová slova
Terén, renderování, Geometry Clipmaps, GPU.

Keywords
Terrain, rendering, Geometry Clipmaps, GPU.

Citace
Martin Marušič: Renderování rozsáhlého terénu, diplomová práce, Brno, FIT VUT v Brně, 2010

Renderování rozsáhlého terénu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jaroslava Přibyla. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Martin Marušič
24. května 2010

Poděkování

Děkuji Ing. Jaroslavovi Přibylovi za cenné připomínky a vstřícnost při realizaci této práce.

© Martin Marušič, 2010.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Teorie renderování terénu **4**

2.1 Stručný přehled moderních algoritmů . . . 5
2.1.1 Strip Masks . . . 5
2.1.2 Seamless Patches . . . 6
2.1.3 GPU Ray-Casting . . . 6
2.2 Geometry Clipmaps . . . 7
2.2.1 Clipmapa . . . 8
2.2.2 Popis algoritmu . . . 9
2.2.3 Oblasti . . . 10
2.2.4 Textury . . . 11
2.2.5 Výpočet aktivních oblastí . . . 11
2.2.6 Aktualizace . . . 11
2.2.7 Vykreslení . . . 12
2.2.8 Přechodové oblasti . . . 13
2.2.9 LOD doprovodných textur . . . 14
2.2.10 Oříznutí pohledovým jehlanem . . . 14
2.2.11 Komprese a syntéza terénu . . . 14
2.3 GPU-Based Geometry clipmaps . . . 16
2.3.1 Vykreslení . . . 17
2.3.2 Aktualizace . . . 19

3 Návrh aplikace **20**

3.1 Použité jazyky a prostředí . . . 20
3.2 Popis součástí . . . 21
3.3 Vstupní data . . . 22

4 Implementace **24**

4.1 Datové struktury . . . 24
4.2 Vertex a index buffery úrovní . . . 26
4.3 Konečný terén . . . 30
4.4 Aktualizace úrovní . . . 30
4.5 Instancing . . . 31
4.6 Aktualizace výškové mapy na GPU . . . 33
4.7 Renderování . . . 34
4.8 Shadery . . . 35

5 Testy a výsledky **38**
5.1 Návrh a implementace testů 38
5.2 Výsledky 38

6 Závěr **40**

A Obsah CD **42**

B Ovládání aplikace **43**

C Programová dokumentace **44**
C.1 Třída ClipmapManager 44
C.2 Třída TextureManager 47

D Rendery **49**

Kapitola 1

Úvod

Renderování rozsáhlého terénu je obsáhlou oblastí počítačové grafiky a v dnešní době tvoří již nedílnou součást mnoha aplikací. Data terénu jsou obvykle uložena ve výškové mapě nebo generována se snahou přiblížit se realitě. Využití a zobrazení těchto dat pro interaktivní práci tvoří základ např. těchto aplikací – geografické informační systémy, letecké a jiné vojenské simulátory, hry a mnohé další. Renderování obrovského množství dat, která definují geometrii a další vlastnosti terénu, v reálném čase není možné. Proto byly vymyšleny techniky snižující vysokou komplexnost terénu tím, že efektivně snižují počet trojúhelníků potřebných pro vykreslení terénu se zachováním vysoké vizuální kvality. Tyto techniky řízení úrovně detailu jsou obecně nazývány LOD[1]. LOD techniky ale přinášejí celou řadu problémů, např. trhliny a T-spoje, které je nutné řešit.

Protože také neustále stoupají nároky na rozsah a kvalitu terénu, je nutné vyvíjet nové algoritmy, případně optimalizovat ty starší, aby využily vlastností nových grafických akcelerátorů. Trendem dnešní doby je přesunout co nejvíce zátěže z CPU na GPU, což umožní rychle zpracovat velké množství dat s využitím paralelizace. A proto bude také cílem této diplomové práce implementovat algoritmus pro renderování rozsáhlého terénu s využitím GPU.

Diplomová práce je strukturována do několika logických celků. Nejprve je věnována pozornost teorii renderování terénu, o níž pojednává kapitola 2. Tato kapitola dále obsahuje rozdělení LOD technik do několika skupin, stručný popis tří vybraných moderních algoritmů a detailní popis algoritmu Geometry Clipmaps a jeho optimalizace GPU-Based Geometry Clipmaps, který bude implementační náplní této práce. V kapitole 3 je popsán návrh aplikace pro renderování rozsáhlého terénu a výběr implementačních jazyků a prostředí. Dále je v této kapitole definován formát vstupních dat potřebných pro běh aplikace. Kapitola 4 podrobně popisuje implementační detaily jednotlivých částí algoritmu GPU-Based Geometry Clipmaps a uvádí rozdíly mé implementace oproti té popsané v původní publikaci. Návrh vhodných prostředků pro otestování výsledné aplikace i s prezentací výsledků testů je popsán v kapitole 5. V závěrečné kapitole jsou zhodnoceny dosažené výsledky, přínos této práce a možnosti budoucího pokračování.

Teoretická část této diplomové práce tématicky navazuje na semestrální projekt a rozšiřuje ho.

[1] Level of detail

Kapitola 2

Teorie renderování terénu

Terén je vertikálním rozměrem zemského povrchu. Jeho vlastnosti ovlivňují počasí, vodní toky a celou biosféru. A nejen proto je důležité umět modelovat a zobrazit realisticky různé typy terénu i s jejich různorodými vlastnostmi.

Schopnost reprezentovat rozsáhlé terény v digitální podobě se rychle vyvíjí. Ještě před několika lety se za rozsáhlý terén mohlo považovat pár kilometrů čtverečných. Ale dnes lze již zobrazit terén o rozloze stovek kilometrů čtverečných, a dokonce i celé planety. Hlavními omezeními jsou grafický hardware, procesor, dostupná paměť pro uložení terénu, způsob jeho pořízení a schopnosti použitého algoritmu. V případě generovaného terénu odpadají potíže s jeho pořízením a uchováním, ale vyskytují se problémy s jeho realističností.

Rozsáhlý terén nelze interaktivně renderovat přístupem hrubou silou, proto je nutné zapojit nějakou LOD techniku, která sníží počet trojúhelníků při snaze zachovat co nejvyšší kvalitu výsledného meshe. Algoritmy pro renderování terénu se dají rozdělit podle přístupu těchto LOD technik následovně.

- Shora dolů a zdola nahoru.
- Pravidelné mřížky a nepravidelné trojúhelníkové sítě.
- Quadtree[1] a bintree[2].

Toto rozdělení je i s popisem převzato z knihy *Level of Detail for 3D Graphics* [6].

Shora dolů a zdola nahoru

Jedním z hlavních rozdílů mezi LOD technikami je přístup ke zjednodušování geometrie terénu. Přístup shora dolů začíná s velmi jednoduchým terénem, např. se dvěma nebo čtyřmi trojúhelníky reprezentujícími celý terén, a progresivně přidává další trojúhelníky, dokud není dosaženo požadovaného detailu. Tento přístup je také často nazýván jako dělení nebo zjemňování.

Naproti tomu přístup zdola nahoru začíná s nejdetailnějším terénem a postupně odebírá vrcholy, dokud není dosaženo požadovaného zjednodušení. Tyto techniky se také často nazývají jako decimace nebo zjednodušování.

[1] Kvadrantový strom
[2] Binární strom

Pravidelné mřížky a nepravidelné trojúhelníkové sítě

Dalším důležitým rozdílem je struktura použitá pro reprezentaci terénu. Pravidelné mřížky lze chápat jako dvourozměrné pole pravidelně rozmístěných (x, y) souřadnic. Zatímco nepravidelné trojúhelníkové sítě dovolují nepravidelné rozmístění těchto souřadnic, a tudíž mohou aproximovat povrch terénu přesněji s méně trojúhelníky. Nevýhodou tohoto přístupu je náročnější uložení takovéto struktury a práce s ní. Dále bych poznamenal, že se od tohoto přístupu pro renderování terénu upouští, protože ho nelze snadno přenést na GPU oproti algoritmům s pravidelnou strukturou.

Quadtree a bintree

Dalším členěním je hierarchický způsob rozdělení terénu s pravidelnou mřížkou pro implementaci pohledově závislého LOD. Rozdělení terénu na menší části dovoluje, aby každá část měla jiné rozlišení v závislosti na pohledových parametrech či vlastnostech terénu (např. výška nebo vzdálenost od pozorovatele). Hierarchických struktur je více, ale nejčastěji používané jsou právě quadtree a bintree. Struktura quadtree dělí pravoúhlou oblast na čtyři menší oblasti, které lze dále dělit stejným způsobem. Binární strom funguje obdobným způsobem, ale vždy dělí trojúhelník na dvě poloviny.

S těmito strukturami souvisí také problémy jako trhliny a T-spoje, které jsou způsobeny sousedními trojúhelníky s různým rozlišením. Existuje několik možných řešení, jako např. rekurzivní rozdělení sousedních trojúhelníků tak, aby jejich geometrie na sebe navazovala. Podrobnější popis postupu řešení těchto problému lze nalézt v [6].

2.1 Stručný přehled moderních algoritmů

V dnešní době již existuje celá řada různých algoritmů pro renderování terénu, což potvrzuje i usilovný výzkum v této oblasti. S moderním a dokonalejším hardware, přinášejícím nové vlastnosti, se otevírají další možnosti vývoje, a tím pádem i vzniku nových algoritmů. Velmi obsáhlým přehledem všeho, co se týká terénu, je Virtual Terrain Project[3]. Cílem této webové stránky je usnadnění vývoje nástrojů pro vytváření všemožných částí reálného světa v 3D digitální formě. Pro tuto práci je na této stránce tím nejzajímavějším přehled publikovaných LOD technik pro renderování terénu a jistě bude cenným zdrojem informací i pro další studium.

Z této obsáhlé množiny jsem vybral tři moderní a zajímavé algoritmy. Tyto tři algoritmy budou stručně popsány v následujících kapitolách. Podrobněji bude popsán algoritmus Geometry Clipmaps a jeho optimlizace GPU-Based Geometry clipmaps, která bude implementační náplní této práce.

2.1.1 Strip Masks

Tento algoritmus byl publikován v roce 2005 [7]. Je složen ze dvou částí. První část se zabývá streamováním dat terénu ke klientské aplikaci. Také se stará o zachování největší čtvercové oblasti centrované okolo pozorovatele, která se vejde do paměti na straně klienta. Druhá část se zabývá adaptivním vykreslením této oblasti. Hlavním záměrem je vykreslení maximálního počtu trojúhelníků a texturových map nejvyšší kvality při zachování daného počtu snímků za sekundu.

[3] http://www.vterrain.org/

Obrázek 2.1: Strip masky [7].

K přenosu je použit klasický dlaždicový systém dělení terénu. Databáze dlaždic je vygenerována pouze jednou z plného DEM[4] a textur terénu. Geometrie každé dlaždice je zakódována do VRML souboru a fotometrie do JPEG souboru. Adaptivní renderování čtvercové oblasti je provedeno použitím dlaždic s různým rozlišením. Před vyrenderováním snímku globální algoritmus určí pro každou dlaždici vizuální důležitost založenou na její výšce a vzdálenosti od pozorovatele. Dlaždice může být vykreslena použitím strip masky (viz obr. 2.1) z množiny úrovní od hrubých po jemné. Maska je předpočítaná, a tudíž je znám potřebný počet trojúhelníků. Tato vlastnost dovoluje dosáhnout zvoleného počtu snímků za sekundu. Dlaždicové schéma je geometricky méně optimální než lokální LOD algoritmy, ale na druhou stranu přesouvá většinu zátěže z CPU na GPU.

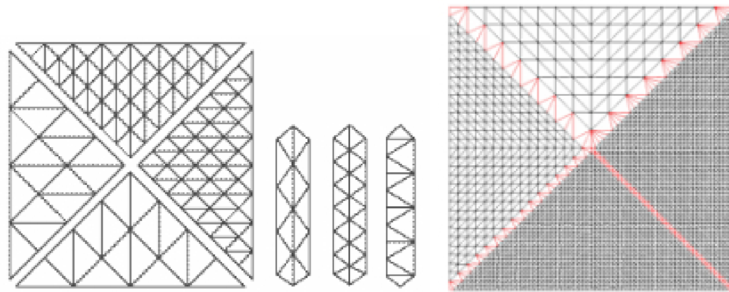
2.1.2 Seamless Patches

Tato moderní metoda se datuje k roku 2007 [4] a přišla s novým přístupem k interaktivnímu renderování rozsáhlého terénu. Je založena na dělení terénu na malé čtvercové plochy různých rozlišení. Každá plocha je reprezentována čtyřmi trojúhelníkovými díly (viz obr. 2.2), kde každý může mít jiné rozlišení, a čtyři pásy, které spojují tyto čtyři díly hladce k sobě. Výsledkem tohoto schématu jsou změny rozlišení uvnitř ploch a ne napříč jimi. Za běhu jsou plochy použity k vytvoření úrovně detailu závisle na pohledových parametrech. Jelikož rozlišení sousedících ploch na společných hranách souhlasí, výsledný mesh neobsahuje žádné nechtěné praskliny nebo degenerované trojúhelníky. GPU generuje meshe ploch použitím patřičně zvětšených instancí uložených dílů a každému vrcholu přiřazuje výšku z uložených textur. Tento algoritmus dosahuje vysoké vizuální kvality při vysokém počtu snímků za sekundu.

2.1.3 GPU Ray-Casting

Dalším zajímavým moderním algoritmem je GPU Ray-Casting [2] z roku 2009. Jeho cílem je překonat omezení maximální propustnosti geometrie dnešních GPU při renderování terénu s velkým rozlišením (ukázka výstupu viz obr. 2.3). Tento algoritmus pracuje metodou vrhání paprsků a k urychlení hledání průsečíků paprsků na GPU používá maximum mipmaps datové struktury výškové mapy. Maximum mipmaps [9] je předpočítaná hierarchická struktura, je ekvivaletní s plně rozděleným quadtree a pracuje téměř s logaritmickou složitostí. Dále přichází s novým způsobem texturování s plně anizotropním filtrováním, který dovolí interaktivně rednerovat terabytová data, včetně foto textur. Metodou vrhání

[4] Digital Elevation Model



Obrázek 2.2: Trojúhelníkové části, spojovací pásy, plocha [4].

Obrázek 2.3: Terén vyrenderovaný metodou GPU Ray-Casting zobrazující plochu 460 km × 600 km [2].

paprsku lze efektivně snížit geometrickou zátěž na GPU, a tím tedy zvýšit počet snímků za sekundu oproti klasickým rasterizačním přístupům.

Tato metoda renderování terénu pracuje s reprezentací modelu terénu založenou na dlaždicích s více rozlišeními. Každá dlaždice sestává z výškové mapy rozměru $N \times N$ vzorků a ortografické foto textury. V každém snímku je stanovena množina dlaždic dle aktuálního pohledu a tyto dlaždice jsou vykresleny odpředu dozadu použitím metody vrhání paprsku.

2.2 Geometry Clipmaps

Algoritmus Geometry Clipmaps je další moderní LOD technikou pro zobrazování rozsáhlého terénu. V roce 2004 tento algoritmus publikovali Frank Losasso a Hugues Hoppe [5]. Soustředili se hlavně na snížení počtu operací prováděných na CPU a co největšího využití možností GPU. Protože propustnost GPU přesáhla hranice 100 miliónů trojúhelníků za sekundu, bylo možné vyplnit celý obrazový buffer trojúhelníky o velikosti pixelu při zachování video rychlosti[5].

[5] Minimálně 25 FPS

Terén je v této metodě uložen jako množina vnořených pravidelných mřížek centrovaných okolo pozorovatele. Každá mřížka reprezentuje filtrovanou verzi terénu při rozlišeních rovných mocninám dvou a je uložena jako vertex buffer ve video paměti. S pohybem pozorovatele jsou jednotlivé úrovně clipmapy posouvány a inkrementálně aktualizovány daty.

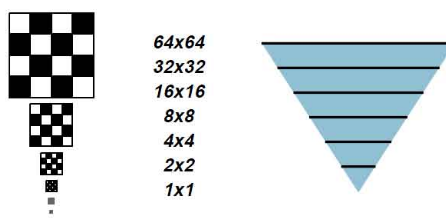
2.2.1 Clipmapa

Princip algoritmu Geometry Clipmaps vychází z clipmap popsaných v The Clipmap: A Virtual Mipmap [8]. Clipmapa je založena na mipmapě, kterou definoval Williams v roce 1983 [10].

Mipmapa

Mipmapa je kolekce souvztažných obrazů se snižujícím se rozlišením, kterou lze reprezentovat pyramidou. Pyramida začíná nultou úrovní, největší a nejdetailnější. Každá další úroveň reprezentuje obraz použitím poloviny texelů předešlé úrovně v každé dimenzi. Příklad 2D mipmapy je ukázán na obrázku 2.4.

Každý vykreslovaný pixel je odvozen od jednoho či více texelů z jedné či více úrovní mipmapy. Konkrétně jsou texely vybrány z okolí úrovně mipmapy, kde je mapování nejblíže poměru 1 : 1. Texely jsou poté přefiltrovány do jedné hodnoty, která je určena pro další zpracování. Hlavním přínosem mipmap je zrychlení renderování a redukce aliasingu.



Obrázek 2.4: 2D mipmapa [8].

Struktura

Clipmapa je aktualizovatelná reprezentace části mipmapy, ve které je každá úroveň oříznuta na specifikovanou maximální velikost. Z této parametrizace plyne obeliskový tvar clipmapy (viz obr. 2.5 a)) oproti pyramidovému tvaru mipmapy. Také definuje velikost texturovací paměti potřebnou pro plnou reprezentaci této hierarchie textury.

Na základě této reprezentace je dále definován pojem *Clipmap Stack*, což je množina úrovní, které vznikly oříznutím z mipmapy hodnotou velikosti clipmapy *ClipSize*. Tyto úrovně clipmapy neobsahují plnou reprezentaci dané úrovně mipmapy, ale pouze ukládají oblast velikosti $Clipsize^2$. Tyto úrovně jsou znázorněny na obrázku 2.5 b). Pod úrovněmi z *Clipmap Stack* se nachází *Clipmap Pyramid*, definována jako množina úrovní, jejichž velikost není větší než *ClipSize*. Úrovně z pyramidové části jsou celé uloženy v texturovací paměti a jsou identické s odpovídajícími úrovněmi mipmapy.

Obrázek 2.5: a) Oblast clipmapy v mipmapě, b) Clipmap Stack a Clipmap Pyramid [8].

Obrázek 2.6: Toroidní adresace [8].

Aktualizace

Paměť clipmapy je záměrně natolik velká, aby efektivně pojala texely potřebné pro vykreslení jednoho pohledu. Proto je tedy nezbytné aktualizovat paměť clipmapy zároveň s pohybem pozorovatele, obvykle před vykreslením každého snímku. Při aktualizaci je využito toho, že obsah paměti je mezi následujícími snímky koherentní. A tudíž lze využít již uložené texely pro vykreslení dalších snímků. Každá úroveň je definována jako nezávisle se pohybující paměť pro 2D obraz, která se pohybuje v celé oblasti odpovídající úrovně mipmapy za použití toroidní adresace.

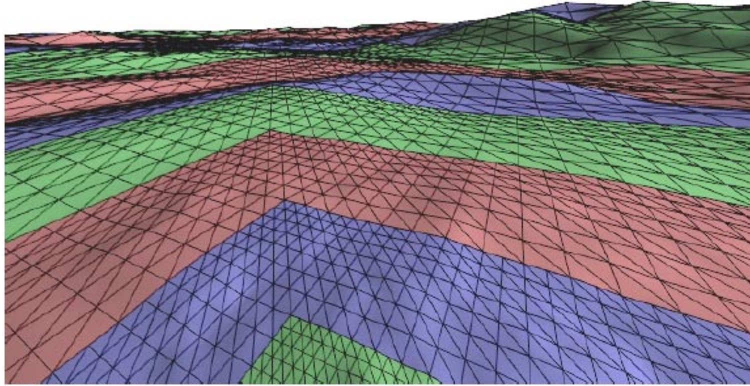
Toroidní adresace je běžně používaná technika v aplikacích zpracování obrazu. Pomocí tohoto adresování je každá úroveň inkrementálně aktualizovatelná. Souřadnice textury jsou určeny pomocí operace modulo, a tím se předejde přepisování velkých objemů dat. Toroidní adresace, jak je patrné z obrázku 2.6, zapíše data z horní části T dolů, z pravé časti R doleva, z pravé horní části C doleva dolů a největší část *Same* zůstává nezměněna.

2.2.2 Popis algoritmu

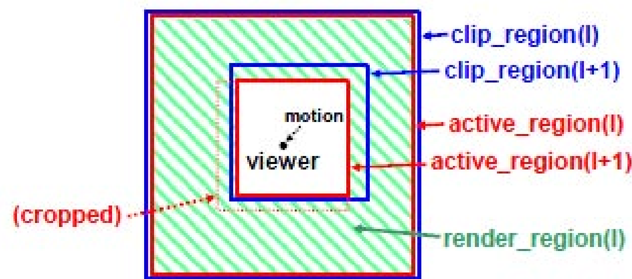
Algoritmus Geometry Clipmaps ukládá pyramidu terénu použitím množiny m úrovní, kde každá reprezentuje vnořenou plochu v po sobě jdoucích rozlišeních mocniny dvou (viz obr. 2.7). Každá úroveň obsahuje $n \times n$ vrcholů uložených ve vertex bufferu ve video paměti. Efektivní inkrementální aktualizace obsahu bufferů jsou prováděny toroidním adresováním, tzn. adresováním pomocí operace modulo na indexech x a y. Každý vrchol obsahuje čtveřici souřadnic (x, y, z, z_c). Hodnota z_c představuje výšku na souřadnicích (x, y) v následující méně detailní úrovni a je použita pro morfování přechodových oblastí.

Samotný algoritmus lze rozdělit do následujících částí, které jsou provedeny při výpočtu každého snímku:

- Určení požadovaných aktivních úrovní.



Obrázek 2.7: Terén vyrenderovaný metodou Geometry Clipmaps (ClipSize=31) [5].



Obrázek 2.8: Oblasti definované v rámci úrovní geometry clipmap [5].

- Aktualizace.
- Oříznutí aktivních oblastí na ořezové oblasti a vykreslení.

2.2.3 Oblasti

Pro každou úroveň l clipmapy je definována množina obdélníkových oblastí (viz obr. 2.8).

Ořezová oblast

Tato oblast vymezuje mřížkovou plochu světa o velikosti $n \times n$ uloženou na dané úrovni.

Aktivní oblast

Aktivní oblast určuje plochu, kterou si přejeme vykreslit, konkrétně čtverec velikosti $n \times n$ centrovaný okolo pozorovatele. Při pohybu pozorovatele jsou clipmapy aktualizovány posouváním ořezových oblastí každé úrovně tak, aby odpovídaly poloze požadovaných aktivních oblastí. Nicméně pokud je aktualizace příliš náročná, typicky při rychlém pohybu pozorovatele, ořezová oblast zaostává za pozorovatelem. Poté je aktivní oblast oříznuta podle dostupných dat. Tento proces je znázorněn na obrázku 2.8.

U nejdetailnější úrovně m je vždy aktivní oblast $m + 1$ definována jako prázdná.

Vykreslovací oblast

Samotná vykreslovací oblast má tvar rámu kolem detailnější úrovně. Její vnější obvod je tvořen aktivní oblastí úrovně l a vnitřní obvod aktivní úrovní $l + 1$.

2.2.4 Textury

Geometry Clipmaps mají velkou výhodu v tom, že lze použít stejnou LOD techniku i pro další textury než jen pro výškové mapy. Každá úroveň clipmapy také obsahuje doprovodné textury. Obvykle je zde uložena normálová mapa s 8 bity na kanál pro stínování povrchu. Normálová mapa se vypočítá z geometrie, kdykoliv je clipmapa aktualizována.

Samozřejmě je možné uchovávat v této hierarchii i další obrazy, jako jsou např. barevné textury nebo atributy terénu. Tyto mohou mít také různá rozlišení a je k nim také přistupováno toroidním adresováním pro efektivní aktualizace.

2.2.5 Výpočet aktivních oblastí

Pohledově závislé zjemňování je určeno výběrem aktivních oblastí pro každou úroveň clipmapy. Použitý postup je jednoduchý. Pro každou úroveň l s rozestupem mřížky $g_l = 2^{-l}$ ve světových souřadnicích je požadovaná aktivní oblast čtvercem o velikosti $ng_l \times ng_l$, kde n je velikost clipmapy, centrovaná kolem pozorovatele v bodě (x, y). Jinými slovy, požadovaná aktivní oblast je vycentrovaná kolem pozorovatele a přejeme si vykreslit celý obsah každé úrovně.

Pokud se pozorovatel vzdaluje od terénu, nejdetailnější úroveň je až příliš detailní a začíná vznikat aliasing. Řešením je deaktivovat vykreslování těchto detailních úrovní. Konkrétně se spočítá výška pozorovatele nad terénem z nejdetailnější aktivní úrovně clipmapy. Pro každou úroveň l je aktivní oblast nastavena jako prázdná, pokud je výška pozorovatele nad terénem větší než $0,4ng_l$.

Nevýhodou jednoduchého systému centrovaných oblastí okolo pozorovatele je, že velikost clipmapy n se musí zvětšovat se zúžujícím se úhlem zorného pole φ. Řešním by bylo přizpůsobit pozici a velikost oblastí clipmapy k pohledovému jehlanu. Ale místo toho jsou zvoleny centrované oblasti, protože dovolují okamžité rotování dle současného pohledu. Toto je požadavek mnoha aplikací jako jsou letecké simulátory, ve kterých se uživatel může dívat libovolným směrem. Algoritmus spoléhá na oříznutí geometrie podle pohledového jehlanu, aby zabránil vykreslování geometrie ležící mimo tento jehlan.

2.2.6 Aktualizace

Jak se aktivní oblasti posouvají s pohybem pozorovatele, tak by se měly také patřičně posouvat i ořezové oblasti. Poznamenejme, že s toroidním přístupem není třeba kopírovat stávající data s posunem úrovní. Místo toho stačí vyplnit nově odkrytou oblast tvaru „L“. Data mohou pocházet ze dvou zdrojů: z dekomprese určitého terénu nebo ze syntézy procedurálního terénu. Obvykle data hrubších úrovní pocházejí z dekomprimovaného terénu a jemnější úrovně jsou syntetizovány.

S rychlým pohybem pozorovatele se čas potřebný pro aktualizaci všech úrovní může stát neúměrným. Stějne jako v clipmapách textur [8], tak i v této metodě aktualizujeme úrovně v pořadí od hrubších k jemnějším. Ovšem nestihne-li se provést aktualizace všech úrovní v přiděleném čase, proces se přeruší. Autoři definovali tuto hranici na maximáně n^2

aktualizovaných hodnot. Ořezové oblasti, které se nestihly aktualizovat, zaostávají a postupně ořezávají přidružené aktivní oblasti, dokud nejsou prázdné. Efektem je ztráta vysokofrekvenčních detailů terénu při rychlém pohybu pozorovatele v malé výšce. Zajímavým následkem je snížení vykreslovací zátěže s vyšší rychlostí pozorovatele.

Omezení oblastí

1. Ořezová oblast $(l+1) \subseteq$ ořezová oblast $(l) \ominus 1$, kde $\ominus$ značí erozi skalární vzdálenosti. Ořezové oblasti musí být vnořeny kvůli predikci. Predikce vyžaduje zachování jedné mřížkové vzdálenosti na všech stranách.
2. Aktivní oblast $(l) \subseteq$ ořezová oblast (l). Vykreslovaná data musí být podmnožinou dat přítomných v clipmapě.
3. Obvod aktivní oblasti l musí ležet na sudých vrcholech, aby hranice s hrubší úrovní $l-1$ byla vodotěsná.
4. Aktivní oblast $(l+1) \subseteq$ aktivní oblast $(l) \ominus 2$. Vykreslovací oblast musí být minimálně široká jako dvě mřížkové vzdálenosti. Toto omezení zaručuje souvislý přechod mezi úrovněmi.

2.2.7 Vykreslení

Dle vypočítaných požadovaných aktivních oblastí renderujeme terén použitím následujícího algoritmu:

Algoritmus 2.1 Vykreslení terénu

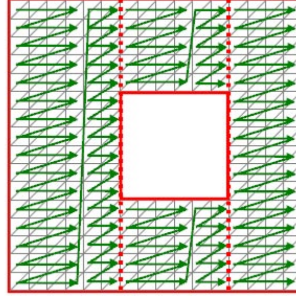
// oříznutí aktivních oblastí
for all úroveň $l \in [1, m]$ od hrubší k jemnější **do**
  ořízni aktivní úroveň (l) na ořezovou úroveň (l)
  ořízni aktivní úroveň (l) na aktivní úroveň $(l-1) \ominus 2$
end for
// vykreslení všech úrovní
for all úroveň $l \in [1, m]$ od jemných k hrubým **do**
  vykreslovací oblast (l) := aktivní oblast $(l)-$ aktivní oblast $(l+1)$
  vykresli vykreslovací oblast (l)
end for

Aktivní oblasti jsou oříznuty na ořezové oblasti a hrubší aktivní oblasti tak, aby splnily omezení 2-4 definované výše. Je poměrně časté, že detailnější úrovně mají aktivní oblasti prázdné. Buďto proto, že jejich ořezové oblasti nebyly včas aktualizovány (pozorovatel se pohybuje rychle), nebo proto, že detailnější teselace je nežádoucí (pozorovatel je vysoko nad terénem).

Protože detailnější úrovně jsou blíže k pozorovateli, vykreslujeme je v pořadí od jemnějších k hrubším, čímž je možné využít hardwarový occlusion culling[6]. Vykreslovací oblast je rozdělena na čtyři obdélníkové části, které jsou vykresleny použitím tristripů[7], jak je naznačeno na obrázku 2.9. Maximální délka tristripu, přibližně 20 dle autorů, je vybrána pro optimální ukládání vrcholů a ty jsou dále seskupeny do větších dávek primitiv.

[6] Odstranění skrytých ploch
[7] Triangle strip

Mřížkově-sekvenční přístup do paměti funguje dobře na všech úrovních hierarchie video paměti. V době publikování tohoto algoritmu bylo potřeba pro 2D toroidní přístup přepočítat indexy vrcholů na CPU v každém snímku. Nebyla to příliš velká režie a autoři předpokládali, že brzy bude tento nedostatek překonán. Za necelý rok přišli s optimalizovaným algoritmem GPU-Based Geometry Clipmaps, který tento nedostatek odstranil využitím vertex textur.



Obrázek 2.9: Rozložení tristripů ve vykreslovací oblasti [5].

2.2.8 Přechodové oblasti

Algoritmus, tak jak byl doposud popsán, trpí problémem trhlin mezi vykreslovacími oblastmi různých úrovní. To je způsobeno rozdílným rozlišením na okrajích úrovní. Řešením je morfování geometrie poblíž vnějších okrajů každé vykreslovací oblasti l (obr. 2.10) tak, aby přecházela v geometrii hrubší úrovně $l-1$. Tento přístup eliminuje trhliny a navrch přispívá k časové spojitosti. Morfování je funkcí prostorových souřadnic (x, y) mřížky vrcholů terénu relativních k pozici pozorovatele (v_x, v_y).

Experimentem přišli autoři k nejvhodnější hodnotě šířky přechodové oblasti $w = n/10$. Pokud by w bylo podstatně menší, okraje úrovní by byly patrné. Na druhou stranu, pokud by tato šířka byla mnohem větší, následkem by byla zbytečná ztráta detailu. Morfovaná souřadnice z' vrcholu se spočítá následovně

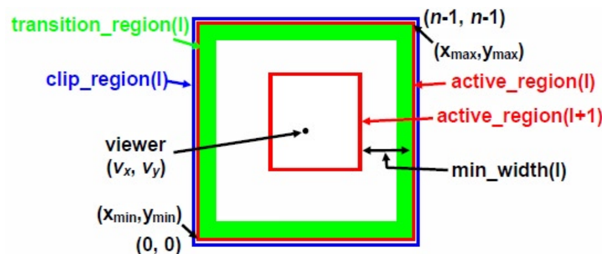
$$z' = (1-\alpha)z + \alpha z_c, \tag{2.1}$$

kde z je souřadnice aktuálního vrcholu, z_c je souřadnice vrcholu v následující hrubší úrovni a blend parametr $\alpha = max(\alpha_x, \alpha_y)$.

$$\alpha_x = min\left(max\left(\left(|x - v_x^l| - \left(\frac{x_{max} - x_{min}}{2} - w - 1\right)\right)\Big/ w, 0\right), 1\right) \tag{2.2}$$

Parametr α_y se spočítá podobně jako α_x z rovnice 2.2. V této rovnici (v_x^l, v_y^l) značí spojité souřadnice pozorovatele v mřížce ořezové oblasti l a x_{min}, x_{max} jsou celočíselné hodnoty mezí aktivní oblasti téže úrovně. Žádoucí vlastností je, že α se rovná nule, kromě přechodové oblasti. V této oblasti lineárně roste až do hodnoty jedna na vnějším okraji. Výpočty jsou prováděny ve vertex shaderu na GPU. Dle autorů stačí na tento výpočet přibližně 10 instrukcí, takže příliš nezvyšuje dobu renderování.

Přestože tyto přechodové oblasti odstraní trhliny, T-spoje podél okrajů zůstanou. Pro jejich odstranení a dosažení vodotěsného meshe je použito jednoduché řešení. Podél okrajů vykreslovací oblasti jsou vyrenderovány trojúhelníky s nulovou plochou, které tzv. sešíjí sousední úrovně.



Obrázek 2.10: Přechodová oblast úrovně l [5].

2.2.9 LOD doprovodných textur

Každá úroveň clipmapy obsahuje další textury potřebné pro rasterizaci (např. normálová mapa). Vyvstává problém, jak řídit LOD těchto textur. Jedním z možných řešení je použít hardwarové mipmapy. Textura v každé úrovni clipmapy by měla svoji pyramidu mipmapy, a tím by spotřebovala o 33 % více paměti. Nicméně toto řešení má praktičtější problém. Jestliže rozlišení textury není dostatečně velké, na okrajích vykreslovacích oblastí budou patrné ostré přechody. Ostré přechody se projevují při pohybu pozorovatele jako postupující přední části nad povrchem terénu.

Místo toho autoři navrhli jiné řešení. Tím je vypnutí mipmapování a použití přechodových oblastí jako u geometrie (obr. 2.11). Tím pádem je LOD textur založen na vzdálenosti od pozorovatele než na odvození od obrazového prostoru jako u hardwarového mipmapování. LOD textur založený na prostoru je snadno implementovatelný v pixel shaderu na GPU. Pro vykreslení úrovně l je použito textur z úrovně l a $l-1$. Tyto dvě texury se smíchají pomocí již spočítaného parametru α z vertex shaderu pro přechodové oblasti.

2.2.10 Oříznutí pohledovým jehlanem

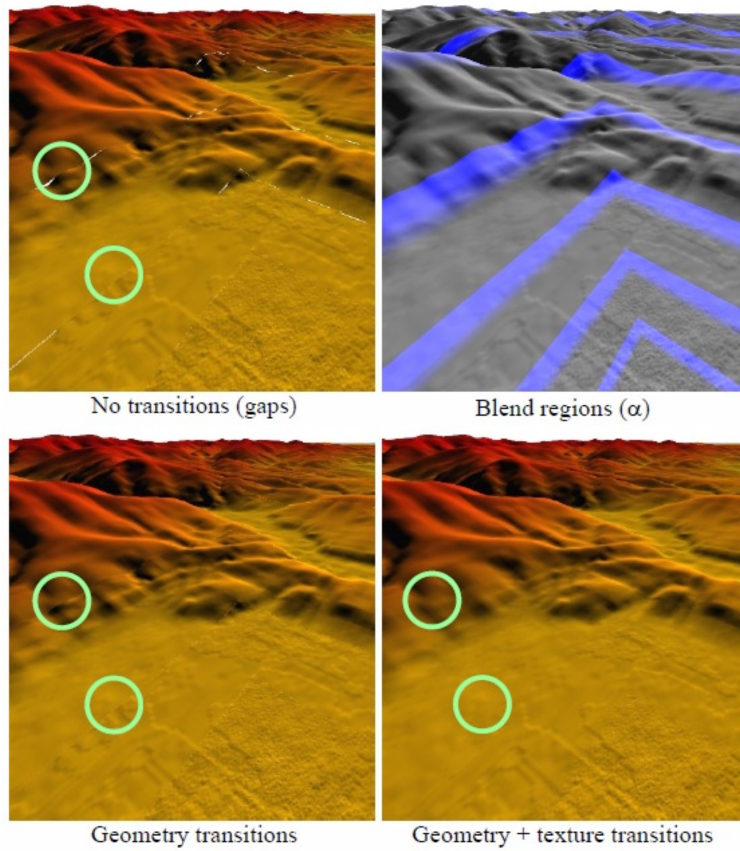
Pro urychlení renderování terénu je použito oříznutí pohledovým jehlanem (obr. 2.12). Protože se každá úroveň skládá ze čtyř obdélníkových částí, lze provést test na průnik těchto částí s pohledovým jehlanem. Výsledek průniku se zobrazí do roviny XY a jeho AABB[8](v tomto případě obdélník nebo kvádr s nulovou výškou) je použit pro oříznutí dané obdélníkové oblasti. Tato technika snižuje vykreslovací zátěž přibližně faktorem 3.

2.2.11 Komprese a syntéza terénu

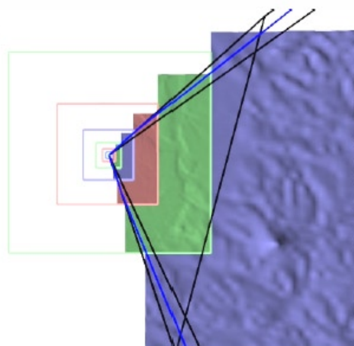
Výškové mapy jsou obvykle velmi koherentní, např. podstatněji více než typické barevné obrázky, a tudíž se nabízí použití komprese. Pro efektivní využití by měl algoritmus dekomprese podporovat tzv. dotazy na oblasti zájmu na všech rozlišeních mocniny dvou. Tento mechanizmus slouží ke zvýšení detailu v těchto oblastech zájmu.

Kompresní metoda je založena na pyramidě terénu $T_1..T_m$. Pyramida obsahuje úrovně vytvořené postupným podvzorkováním detailního terénu T_m použitím lineárního filtru $T_{l-1} = D(T_l)$. Káždá úroveň pyramidy T_l je predikována z hrubší úrovně T_{l-1} pomocí interpolačního dělení $U(T_{l-1})$ a reziduum $R_l = T_l - U(T_{l-1})$ je komprimováno obrázkovým kodérem. Z důvodu ztrátovosti komprese je R_l aproximováno $\tilde{R}_l$. Proto jsou úrovně rekonstruovány v pořadí od nejhrubší jako $\tilde{T}_l = U(\tilde{T}_{l-1}) + \tilde{R}_l$ a rezidua se zkomprimují jako

[8] Axis aligned bounding box



Obrázek 2.11: Morfování textur na okrajích úrovní [5].



Obrázek 2.12: Výsledek oříznutí pohledovým jehlanem [5].

$R_l = T_l - U(\tilde{T}_{l-1})$, aby nedocházelo k nárůstu chyby.

Využití syntézy detailu je u algoritmu Geometry clipmaps možné pomocí stochastického dělení nebo syntézy textury s více rozlišeními. Podmínkou je vytvoření vždy stejného terénu, tudíž je nutné, aby proces syntézy byl deterministický. Procedurální generování přináší terén s nekočnou plochou a detailem. Podle autorů je jednoduchý fraktální šum méně vizuálně zajímavý než opravdový terén, ale propracovanější techniky syntézy by mohly vést k realističtějším krajinám.

Syntéza ani komprese terénu nejsou součástí implementačních cílů této práce, protože nejsou natolik potřebné pro předvedení funkčnosti algoritmu.

2.3 GPU-Based Geometry clipmaps

Originální implementace algoritmu Geometry Clipmaps z roku 2004 využívala tradiční vertex buffery pro uložení úrovní clipmapy. V té době nebylo možné modifikovat vertex buffery přímo, ale bylo potřeba spolupráce s CPU. S příchodem GPU s novými schopnostmi bylo možné upravit původní algoritmus a zásadně ho zoptimalizovat. Tento přepracovaný algoritmus [1] byl publikován přibližne po roce od původního. Tou novou vlastností, která dovolila přesun skoro celého algoritmu na GPU, byly vertex textury [3]. Vertex textury byly zavedeny v shader modelu 3.0.

Použití vertex textur přináší i další výhodu v tom, že je mnohem přirozenější data mřížky každé úrovně uchovávat jako 2D texturu, než je uměle linearizovat do 1D vertex bufferu.

I zde je počet úrovní clipmapy L, kde každá obsahuje mřížku $n \times n$ vzorků geometrie. Dále pak nastává rozdíl v rozdělení (x, y, z) geometrie na dvě části:

- Souřadnice (x, y) jsou uloženy jako konstantní vertex data.
- Souřadnice z je uložena jako jednokanálová 2D textura, tzv. výšková mapa. Pro každou úroveň je definována samostatná výšková mapa o rozměru $n \times n$. Tyto textury jsou aktualizovány s pohybem pozorovatele.

Úrovně clipmapy jsou uniformní 2D mřížky, jejich (x, y) souřadnice jsou pravidelné, a proto konstatní k posunutí a změně velikosti. Proto je možné definovat pouze několik vertex a index bufferů určených pouze pro čtení. Tyto buffery popisují tzv. 2D otisky, které jsou opakovaně použity ve všech úrovních clipmapy.

Uložení výškových dat jako soubor obrázků dovoluje přímé zpracování na GPU. V případě syntetického terénu jsou všechny runtime[9] výpočty prováděny kompletně na GPU. U komprimovaného terénu CPU pouze postupně dekomprimuje a nahrává data do grafické karty.

Datové struktury

Hlavními datovými strukturami jsou předdefinované konstantní vertex a index buffery pro popis (x, y) souřadnic mřížek clipmapy. Dále je pro každou úroveň alokovaná výšková mapa (jednokanálová float 2D textura) a normálová mapa (čtyřkanálová osmibitová 2D textura). Všechny tyto datové struktury jsou uloženy ve video paměti.

[9] Za běhu programu

Velikost clipmapy

Důležitým parametrem je velikost clipmapy n. Tato hodnota udává počet vrcholů uložených v mřížce jedné úrovně. Její hodnota musí být lichá, protože vnější okraj každé úrovně musí ležet na mřížce následující hrubší úrovně. Hardware může být optimalizovaný na textury velikosti mocnin dvou, proto autoři zvolili $n = 2^k - 1$ (což je o jedničku méně než mocnina dvou a nechává jeden řádek a sloupec textury nevyužitý). Další výhodou této hodnoty je, že detailnější úroveň není nikdy přesně uprostřed vzhledem k následující hrubší úrovni. Jinými slovy, je vždy posunutá o jednu mřížkovou vzdálenost vlevo nebo vpravo a stejně tak nahoru nebo dolů v závislosti na pozici pozorovatele. Tato vlastnost je potřebná, aby se detailnější úroveň mohla pohybovat, zatímco hrubší úroveň zůstává na místě.

2.3.1 Vykreslení

Přestože je pro clipmapy alokováno L úrovní, často se vykresluje a aktualizuje pouze podmnožina aktivních úrovní. Aktivní úrovně mají indexy v rozsahu $\langle 0, L' - 1 \rangle$, kde nultá úroveň označuje nejméně detailní (nejhrubší) aktivní úroveň a $L' \leq L$ označuje nejdetailnější aktivní úroveň. Ta je stanovena na základě výšky pozorovatele nad terénem. Motivace je zde stejná jako u staršího algoritmu a to, že pokud je pozorovatel dostatečně vysoko nad terénem, začíná nejdetailnější aktivní úroveň způsobovat aliasing. Konkrétně jsou deaktivovány úrovně, jejichž rozměr mřížky je menší než $2,5 \cdot h$, kde h je výška pozorovatele nad terénem. Samozřejmě nejdetailnější úroveň $L' - 1$ je vykreslena jako celá a ne jen jako rám.

Implementace z roku 2004 dovolovala oříznutí úrovně, pokud nebyla celá zaktualizována. Pro zjednodušení implementace na GPU je tato vlastnost vynechána. Místo toho se úroveň považuje za plně aktualizovanou, nebo je prohlášena za deaktivovanou.

Vertex a index buffery

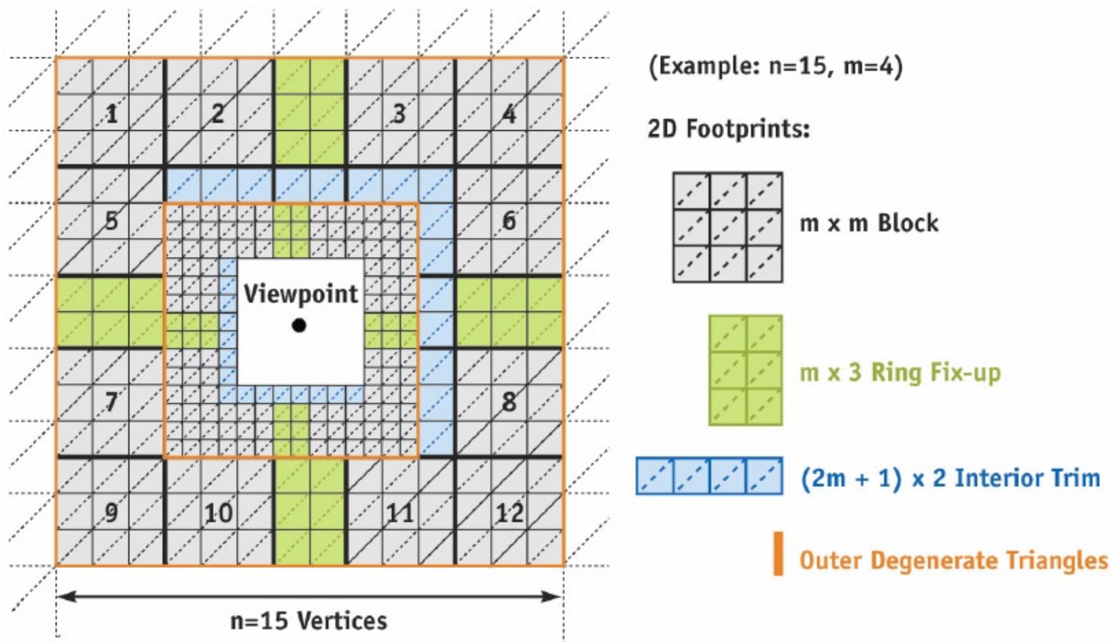
Přístupem hrubou silou by bylo definování jednoho vertex bufferu, který by obsahoval data pro celou úroveň. Ale z důvodu snížení nároků na paměť a možnosti oříznutí pohledovým jehlanem, je rám kolem detailnější úrovně rozdělen na menší 2D otisky (viz. obr. 2.13).

Většinu z tohoto rámu tvoří 12 bloků velikosti $m \times m$, kde $m = (n+1)/4$. Jak již bylo řečeno v úvodu této kapitoly, 2D mřížky jsou pravidelné a lze tedy využít pro vykreslení všech těchto bloků na všech úrovních jednoho vertex a index bufferu jen pro čtení. Vertex shader tyto bloky geometrie zvětší a přesune s pomocí několika parametrů.

Nicméně těchto 12 bloků nepokrývá celý rám, proto je potřeba definovat ještě několik dalších bloků a jejich vertex a index buffery. Tyto bloky mají oproti $m \times m$ blokům malou plochu. Mezera šířky dvou čtverců je zaplněna blokem typu $m \times 3$ za použití jednoho vertex a index bufferu. Dále mezera šířky jednoho čtverce na dvou stranách vnitřního obvodu rámu způsobující nevycentrování detailnější úrovně. Tento blok ve tvaru „L“ může ležet na všech čtyřech možných místech závisející na relativním umístění detailnější úrovně. Pro tento blok jsou definovány čtyři vertex a jeden index buffer. Také se vykreslují podél vnějšího okraje každé úrovně trojúhelníky s nulovou plochou a zabraňují T-spojům. Nakonec jsou u nejdetailnější úrovně vykresleny čtyři $m \times m$ bloky ve vnitřní části a dva bloky tvaru „L“.

Oříznutí pohledovým jehlanem

Implementace je skoro stejná jako u staršího algoritmu. Oříznutí je prováděno na úrovni bloků na CPU. Blok je vykreslen pouze tehdy, je-li průnik s pohledovým jehlanem ne-



Obrázek 2.13: Rozdělení rámu clipmapy do bloků a 2D otisky [1].

prázdný. V závislosti na směru pohledu je vykreslovací zátěž, dle autorů, snížena faktorem 2 až 3. Tato vlastnost nebude implementována, protože je pouze optimalizační částí celého algoritmu.

Instancing

Pro každou úroveň bez instancingu a oříznutí pohledovým jehlanem je potřeba 14 volání funkce pro vykreslení primitiv. S oříznutím pohledovým jehlanem lze snížit počet $m \times m$ bloků v průměru z 12 na 4 v každém snímku. Nejdetailnější úroveň potřebuje dalších 5 volání této funkce pro vyplnění vnitřní části rámu. Celkem je tedy potřeba průměrně $6 \cdot L + 5$ volání vykreslovací funkce primitiv. Toto číslo je dále možné snížit instancováním všech bloků v jedné úrovni na $3 \cdot L + 2$.

Vertex shader

Stejný vertex shader je použit pro vykreslení všech 2D otisků popsaných výše. Shader převádí (x, y) souřadnice otisků do světových souřadnic pomocí jednoduchého zvětšování a posouvání. Dále přečte souřadnici z z výškové mapy uložené ve vertex textuře. Filtrování není potřeba, protože vrcholy přímo odpovídají vzorkům textury.

Pro hladké přechody je použito stejného postupu jako v předchozím algoritmu. Mírně odlišná je rovnice pro výpočet parametru α_x:

$$\alpha_x = clamp\left(\left(|x - v_x| - \left(\frac{n-1}{2} - w - 1\right)\right) \Big/ w, 0, 1\right) \tag{2.3}$$

a podobně pro α_y. Všechny proměnné z této rovnice odpovídají těm z rovnice 2.2. Vzorec

pro lieární interpolaci geometrie je následující

$$z' = (1 - \alpha)z_f + \alpha z_c, \tag{2.4}$$

kde z_f je výška v detailnější úrovni a z_c je výška v hrubší úrovni. Obecně pozice vzorku z_f leží na hraně hrubší mřížky, tudíž je z_c spočteno průměrováním dvou vzorků na koncích této hrany hrubší úrovně. Tento průměr by se dal počítat za běhu, ale vyžaduje tři čtení z vertex textur, což je pomalá operace. Místo toho je z_c spočítáno v rámci aktualizace clipmapy a sbaleno spolu s z_f do stejné jednokanálové float textury.

Pixel shader

Pixel shader je použit pro stínování povrchu pomocí normálové mapy. Autoři doporučují, aby normálová mapa měla dvojnásobné rozlišení oproti výškové mapě, protože jedna normála na vrchol způsobuje rozmazaní. Normálová mapa je inkrementálně aktualizovaná z geometrie společně s clipmapou. Hladké přechody ve stínování jsou také zajištěny pomocí míchání vzorků ze dvou následujících úrovní za pomoci již vypočtené hodnoty α. Normálně by byly zapotřebí dvě čtení z normálových map, ale i zde jsou tyto hodnoty sbaleny dohromady, ve tvaru $(N_x, N_y, N_{cx}, N_{cy})$, do stejné čtyřkanálové osmibitové textury. Hodnoty $N_z = 1$ a $N_{cz} = 1$ jsou dány, a proto je nutné normály po rozbalení normalizovat. Barva pro stínování je získána z 1D textury podle hodnoty z.

2.3.2 Aktualizace

Aktualizace probíhá víceméně stejně jako v původním algoritmu. Rozdíl je v tom, že aktualizace textur se provádí pomocí pixel shaderu.

Kapitola 3

Návrh aplikace

Aplikace pro renderování rozsáhlého terénu je komplexní a výpočetně náročná, tudíž je důležité zvolit správnou metodologii návrhu. Zvolenou metodologií je OOP[1] mající mnoho přínosných vlastností, mezi něž patří abstrakce, zapouzdření a již z názvu vyplývající objekty. Tyto vlastnosti návrhu vedou k větší přehlednosti a lepší údržbě samotného kódu.

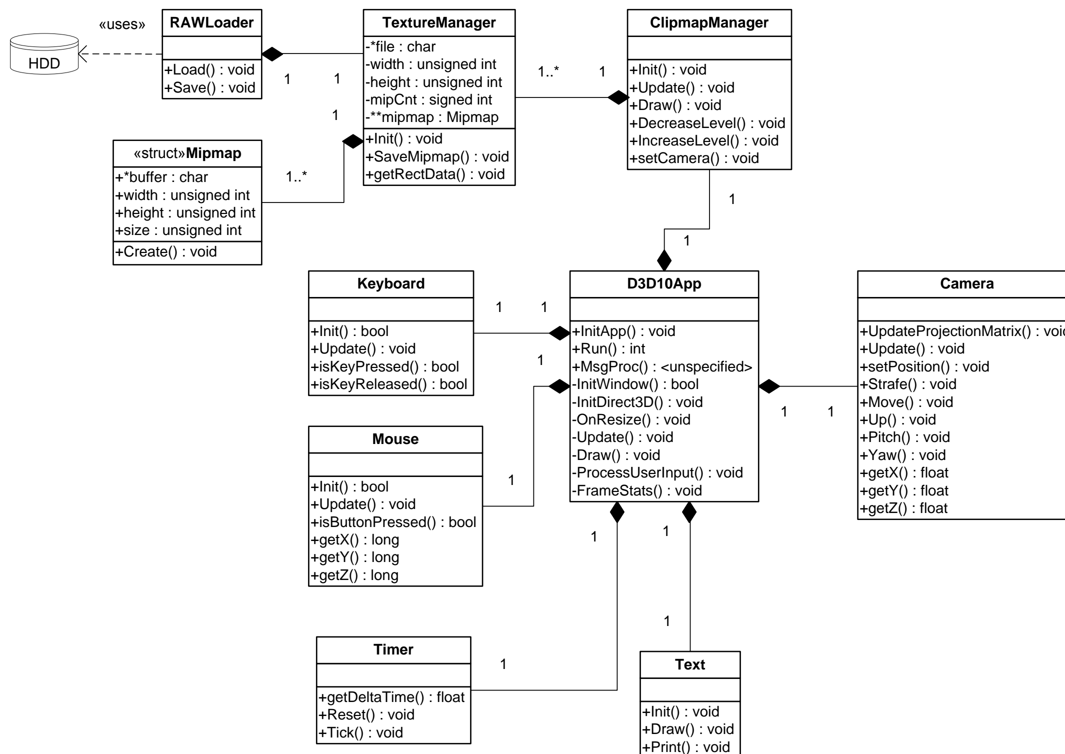
Samotná aplikace by měla být snadno konfigurovatelná pomocí jednoduchého uživatelského rozhraní, které dovolí uživateli nastavit základní parametry pro spuštění aplikace. Dále musí umět zpracovat vstup z klávesnice a myši pro interaktivní práci s terénem. Dalším důležitým vstupem bude výšková mapa uložená v nějakém grafickém formátu na pevném disku. Protože se předpokládá interaktivní práce s aplikací, je nutné, aby renderování probíhalo co možná nejrychleji. Toho je možné dosáhnout vhodnou volbou algoritmu a pečlivou optimalizací jednotlivých částí aplikace. Použitým algoritmem bude Geometry Clipmaps založený na GPU. Tím, že část algoritmu poběží zcela v režii GPU, lze dosáhnout slušného výkonu a odlehčení práce CPU.

3.1 Použité jazyky a prostředí

Důležitou volbou je programovací jazyk, ve kterém bude aplikace implementována. Vhodným jazykem pro rychlé renderování 3D grafických aplikací s objektově orientovaným návrhem je C++. Tento jazyk vyniká vysokým výkonem výsledných aplikací, poměrně slušnou přenositelností a velkou škálou nadstavbových knihoven. K jazyku C++ je velmi vhodná kombinace použití DirectX. DirectX je soubor knihoven poskytujících API[2] pro ovládání různého moderního hardwaru. Pro potřeby této práce bude hlavně využito knihovny Direct3D pro práci s grafickou kartou. Aplikace bude používat tuto knihovnu ve verzi 10. S tím také souvisí použití jazyka HLSL[3] ve verzi 4. Tento jazyk slouží k vytváření programů, takzvaných shaderů, které jsou určeny pro zpracování přímo na grafické kartě.

Podstaný je také výběr vývojového prostředí. Špičkou v této oblasti je Visual Studio od firmy Microsoft, a tudíž je i mou volbou. Použitá verze bude 2010 na operačním systému Windows 7. Toto prostředí nabízí mnoho funkcí, kvalitní debugger, dobrý editor s zvýrazňováním syntaxe a plno dalších.

[1] Objektově orientováné programování
[2] Aplikační rozhraní
[3] High Level Shading Language



Obrázek 3.1: Diagram tříd.

3.2 Popis součástí

Na obrázku 3.1 je zobrazen diagram tříd aplikace. Z důvodu přehlednosti jsou některé méně podstané atributy vypuštěny. Diagram ukazuje závislosti jednotlivých tříd a jejich veřejné rozhraní.

Třída D3D10App

Tato třída řídí chod celé aplikace. Jejím úkolem je inicializovat okno aplikace, subsystémy a Direct3D. Dále musí správně reagovat na změnu velikosti okna, pravidelně aktualizovat všechny subsystémy, sbírat statistiky o renderování, zpracovávat vstup z myši a klávesnice a vykreslit všechny objekty k tomu určené.

Třídy Keyboard a Mouse

Tyto dvě třídy využívají funkcí knihovny DirectInput. Jejich úkolem je zpracování vstupů z klávesnice a myši. Mají minimalistické rozhraní, které obsahuje nejzákladnější funkce. Mezi tyto funkce patří zjištění stavu tlačítek a pozice u myši a stisk kláves na klávesnici.

Třída Timer

Měření času s jemnou granularitou je u aplikací běžících v reálném čase důležité. Třída *Timer* využívá pro přesné měření času funkcí Windows API *QueryPerformanceCounter* a *QueryPerformanceFrequency*. Funkce *Tick* slouží k aktualizaci časů a je volána v každém snímku. Funkce *getDeltaTime* vrací rozdíl současného a minulého času, tzv. delta čas. Delta čas je potřebný pro aktualizace všech subsytémů založených na čase.

Třída Text

Aplikace by měla umět zobrazit také různé informace v textové formě, jako např. počet snímků za vteřinu, pozici kamery či třeba počet renderovaných trojúhelníků. To vše je implementováno v této třídě za použití funkcí DirectX pro práci s fonty.

Třída Camera

Každá 3D aplikace musí implementovat kameru, aby bylo možné pracovat s 3D daty. V této aplikaci je použita kamera s 5 stupni volnosti, která nepovoluje rotaci kolem pohledového vektoru. Tato třída spravuje pohledovou a projekční matici, pozici kamery a parametry jako je rychlost pohybu.

Třída ClipmapManager

Třída *ClipmapManager* je tou nejdůležitější, protože implementuje algoritmus Geometry Clipmaps. Vytváří a inicializuje datové struktury v operační paměti a na grafické kartě potřebné pro renderování terénu. Také je přímo propojená s effect souborem, který obsahuje shadery. Podrobnější popis této třídy je v kapitole 4, kde bude podrobně popsána implementace zvoleného algoritmu.

Třída TextureManager

Tato třída implementuje načítání textur a vytvoření požadovaného počtu jejich mipmap. Její hlavní náplní v této aplikaci je uchovávat informace a data výškové mapy terénu, která se běžně ukládá jako textura. Funkce *getRectData* naplní buffer požadovanou částí mipmapy zadaných rozměrů s toroidní adresací. Tato funkce je důležitá pro aktualizaci dat clipmapy na GPU.

Třída RAWLoader

Třída *RAWLoader* slouží k načítání textur typu RAW z pevného disku. Má velmi jednoduché rozhraní implementující pouze dvě funkce. Funkce *Load* pro načtení dat a funkce *Save* pro jejich uložení na disk. Díky objektovému návrhu je snadné implementovat další třídy pro jiné grafické formáty, pokud zůstane zachováno stejné rozhraní.

3.3 Vstupní data

Výšková mapa, reprezentující konkrétní terén, je nejpodstatnější částí vstupních dat. Aplikace obsahuje podporu souborů typu RAW, což je nezpracovaný obrazový soubor. Takovýto soubor obsahuje hodnoty pixelů uložené za sebou v pořadí červená, zelená a modrá (RGB). Protože pro výškovou mapu stačí jen jeden osmibitový kanál, tak každý byte v mapě značí

jednu hodnotu výšky. Důležitými parametry, které se u soubru bez hlavičky musí zadat, jsou výška a šířka obrázku. Aplikace omezuje tuto hodnotu na mocniny dvou a obě hodnoty musí být stejné.

Kapitola 4

Implementace

V této kapitole jsou popsány implementační detaily algoritmu GPU-Based Geometry Clipmaps. Nejprve se obeznámíme s popisem rozhraní hlavních tříd a jejich běhu. Poté se dostaneme k řešení konkrétních částí algoritmu. Vynechané parametry funkcí a jejich návratové hodnoty jsou i se stručným popisem uvedeny v programové dokumentaci v příloze C.

Veřejných implementací tohoto algoritmu není mnoho a jejich dokumentace je téměř neexistující. Ani původní publikace v některých částech není příliš konkrétní a tyto faktory ztěžují implementaci. Proto je mým cílem podrobně popsat implementaci a její problémy, a tím usnadnit případné další studium algoritmu GPU-Based Geometry Clipmaps. Jedním z hlavních nedostatků, již zmíněné publikace, je opomenutí konečného terénu a řešení problémů s tím spojených. Tento a další budou popsány v následujících kapitolách i s případnými rozdíly v mé implementaci.

Maximální velikost clipmapy je stanovena na $n_{max} = 1023$ a z toho plyne nejvyšší možná hodnota $m_{max} = 256$. Tyto hodnoty jsou dány použitými datovými typy vertex bufferů. Další důležitou hodnotou je velikost rozestupu mřížky. Tato hodnota se vypočítá jako

$$g_l = 2^{L-l-1}, \tag{4.1}$$

kde l značí index úrovně a L celkový počet úrovní. Výsledkem je tedy vzdálenost dvou sousedních vrcholů jedné úrovně. Velikosti rozestupů jsou, oproti těm definovaných v [5], rozdílné. V mé implementaci velikost rozestupu stoupá u méně detailních úrovní, zatímco v původní publikaci se začíná s velikostí jedna a s detailními úrovněmi se zmenšuje.

4.1 Datové struktury

Struktura Clipmap

Struktura *Clipmap* obsahuje souřadnici levého horního rohu jedné úrovně clipmapy. Je zde uložena jak současná pozice, tak i pozice minulá. Minulá pozice se aktualizuje pouze při pohybu úrovně a jen tehdy, je-li aktivní. Toto chování je nutné pro správnou inkrementální aktualizaci odpovídající textury na GPU.

```
struct Clipmap
{
  Vector2<int> prevPos;
  Vector2<int> pos;
};
```

Struktura BlockInstance

Struktura *BlockInstance* představuje parametry jednoho bloku z určité úrovně. Tyto parametry se předávají do vertex shaderu společně s odpovídající geometrií. Slouží ke správnému zvětšení a umístění bloku při použití metody zvané geometrické instancování. Instancování geometrie dovoluje najednou vykreslit více kopií stejného meshe. Souřadnice x, y vektoru $blockW$ představují souřadnice bloku v prostoru světa a hodnoty z, w uchovávají velikost rozestupu mřížky v tomtéž prostoru a typicky mají stejnou hodnotu, protože neuniformní zvětšení bloku není žádoucí.

```
struct BlockInstance
{
  Vector4<int> blockW;
};
```

Struktura Mipmap

Struktura *Mipmap* reprezentuje jednu úroveň hierarchie mipmapy, je definována šířkou a výškou. Tato struktura již obsahuje několik funkcí. Konstruktor *Mipmap* nastaví počáteční hodnoty na nulu a poté lze pomocí funkce *Create* alokovat buffer pro data. Tuto strukturu používá třída *TextureManager* pro uložení mipmapy výškové mapy terénu do operační paměti. Po celou dobu běhu aplikace je v paměti stále přítomna mipmapa celého terénu, takže dostupná operační paměť tvoří jediné omezení velikosti terénu. Toto omezení lze odstranit stejným přístupem jako je tomu u aktualizace vertex textur na GPU, a to inkrementální aktualizací bufferu mipmapy v operační paměti z výškové mapy terénu uložené na pevném disku.

```
struct Mipmap
{
  Mipmap() : buffer(0), size(0), width(0), height(0) {};
  ~Mipmap() { delete buffer; };

  void Create(unsigned int w, unsigned int h);

  char* buffer;
  unsigned int width;
  unsigned int height;
  unsigned int size;
};
```

Třída TextureManager

Třída *TextureManager* (obr. 4.1) se stará o načtení textur do paměti a vytvoření jejich mipmapy. V této aplikaci je použita jen pro správu výškové mapy v operační paměti. Parametry konstruktoru jsou jméno souboru s výškovou mapou, šířka, výška a požadovaný počet úrovní mipmapy. Mipmapy jsou vytvořeny voláním funkce *Init* podvzorkováním originální textury. Není použito žádné filtrování vzorků.

Funkce *getRectData* obstarává načítání dat z mipmapy do bufferu s toroidním adresováním, který je dále použit pro aktualizace vertex textur.

Třída ClipmapManager

Třída *ClipmapManager* (obr. 4.1) tvoří jádro implementovaného algoritmu. Stará se o alokaci datových struktur v operační paměti i na GPU a řídí veškerou logiku algoritmu od aktualizací až po vykreslení pomocí shaderů.

Vstupními parametry této třídy jsou požadovaná velikost clipmapy, jméno souboru s výškovou mapou a v případě RAW formátu rozměry této textury. Konstruktor inicializuje všechna data a proměnné do výchozího stavu a poté je vše alokováno voláním funkce *Init*. Tato funkce prvně načte effect soubor se shadery a propojí proměnné shaderu s aplikací. Dále jsou vytvořeny všechny potřebné vertex a index buffery s navázáním na grafickou pipeline. Následujícím krokem je vytvoření úrovní clipmapy a inicializace jejich pozic se správným zarovnáním mřížek. Poté jsou vytvořeny dva buffery. Jeden v operační paměti využitím struktury *BlockInstance*, která reprezentuje instance jednotlivých bloků, a druhý buffer na straně GPU, kam jsou tyto instance pravidelně nahrávány. Potom je inicializována třída *TextureManager*, jež načte výškovou mapu. V posledním kroku dochází k alokaci vertex textur pole o velikosti počtu úrovní pro reprezentaci $N \times N$ bufferů clipmapy.

Po inicializaci všech částí je v programové smyčce třída *ClipmapManager* v každém snímku aktualizována voláním funkce *Update* s parametrem delta, který představuje časový rozdíl mezi následujícími snímky. Prvně je získána pohledová a projekční matice z kamery pro transformaci souřadnic v shaderu a pozice pozorovatele. Pozice pozorovatele je použita pro následnou aktualizaci úrovní clipmapy. Poté jsou aktualizovány souřadnice instancí bloků dle odpovídajících pozic úrovní. Poslední fází tohoto procesu je inkrementální aktualizace vertex textur na GPU.

Před ukončením aplikace je nutné uvolnit všechny alokované zdroje. O to se automaticky stará destruktor třídy. Uvolnění zdrojů probíhá typicky v opačném pořadí než při inicializaci.

4.2 Vertex a index buffery úrovní

Vertex a index buffery úrovní slouží pro vykreslení geometrie a dělí se na tři typy – $M \times M$, $M \times 3$ a I bloky. Buffery jsou vytvořeny a uloženy do GPU v inicializační části aplikace. Každý blok je následně instancován tak, aby bylo možné vykreslit celou hierarchii clipmapy a umístěn pomocí vertex shaderu. Nákres geometrie těchto bloků v rámci pravidelné mřížky úrovně clipmapy a ilustrace postupu číslování souřadnic a indexů je znázorněn na obrázku 4.2.

$M \times M$ blok

Tento blok, jak již z názvu vyplývá, obsahuje m^2 vrcholů. Souřadnice vrcholů (obr. 4.2) jdou od $(0, 0)$ do $(m - 1, m - 1)$. Počet indexů, potřebných pro vytvoření trojúhelníkové sítě, je roven $6 \cdot (m-1)^2$. Každý trojúhelník je tvořen třemi indexy. Vykreslování indexovaných primitiv je efektivnější, protože indexování eliminuje duplicitní vrcholy. Indexy tvoří jednorozměrné pole a definují trojúhelníky ve směru hodinových ručiček. Algoritmus 4.1 naplní pole indexů pro vykreslení trojúheníkové sítě.

$M \times 3$ bloky

Pro tyto čtyři bloky (viz. obr. 4.2) je použit jeden vertex a index buffer. Každý z těchto $M \times 3$ bloků by mohl být zakódován zvlášť, ale poněvadž je jejich plocha oproti $M \times M$

ClipmapManager

+ClipmapManager()
+~ClipmapManager()
+Init() : void
+Update() : void
+Draw() : void
+DecreaseLevel() : void
+IncreaseLevel() : void
+setCamera() : void
-CreateEffect() : void
-ReleaseEffect() : void
-CreateGeometry() : void
-ReleaseGeometry() : void
-CreateMxM() : void
-CreateMx3() : void
-CreateI() : void
-CreateClipmap() : void
-InitClipmap() : void
-UpdateClipmap() : void
-ReleaseClipmap() : void
-CreateStackTexture() : void
-InitStackTexture() : void
-UpdateStackTexture() : void
-ReleaseStackTexture() : void
-CreateInstance() : void
-UpdateInstance() : void
-ReleaseInstance() : void

TextureManager

-*file : char
-width : unsigned int
-height : unsigned int
-mipCnt : unsigned int
-**mipmap : Mipmap

+TextureManager()
+~TextureManager()
+Init() : void
+SaveMipmap() : void
+getRectData() : void

Obrázek 4.1: Třída *ClipmapManager* (uvedeny jsou pouze veřejné a soukromé funkce) a třída *TextureManager*.

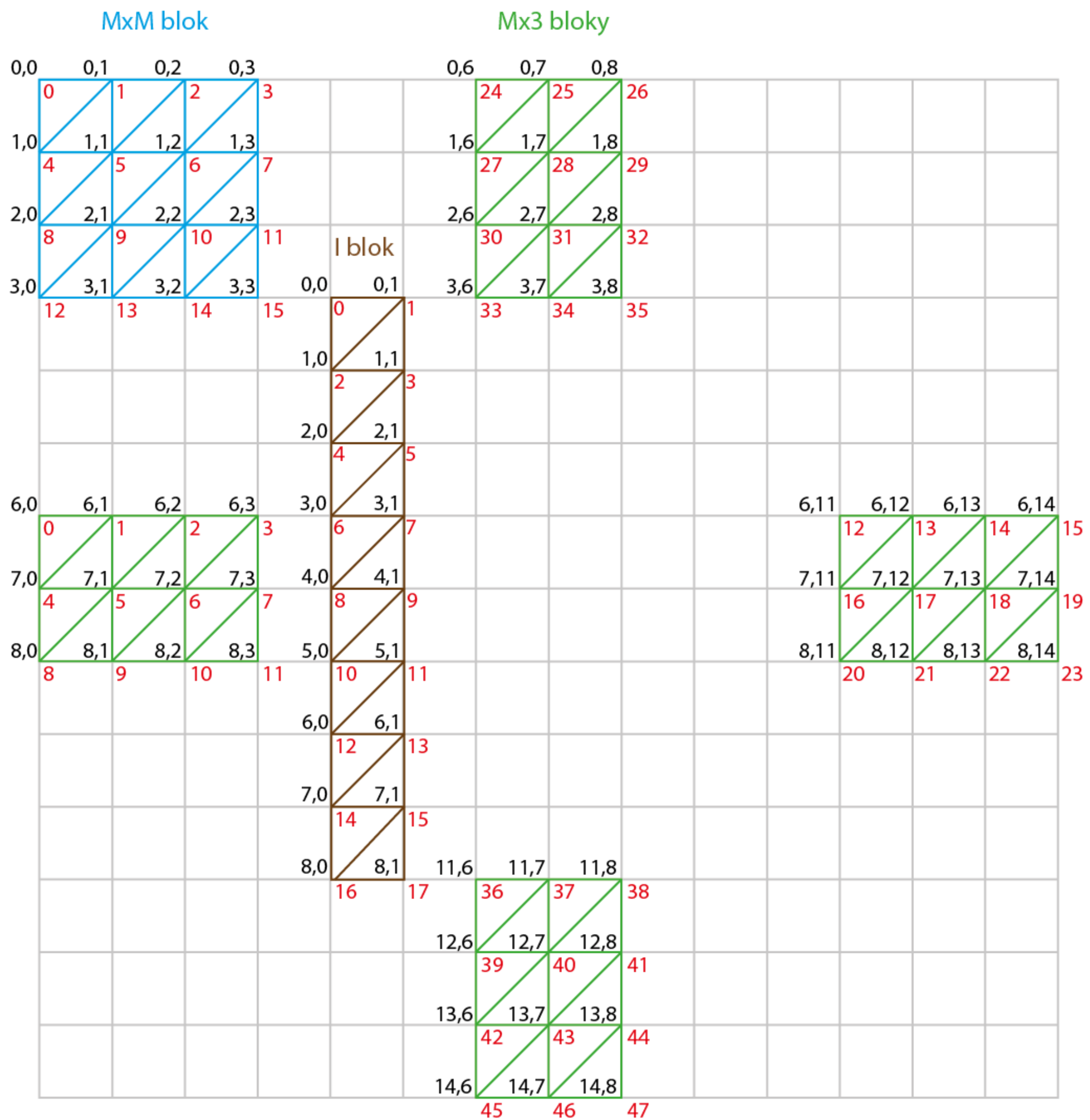
Algoritmus 4.1 Výpočet indexů $M \times M$ bloku

for $i := 0$ to $m - 2$ **do**
  for $j := 0$ to $m - 2$ **do**
    makeTriangle($m \cdot i + j$, $m \cdot i + (j + 1)$, $m \cdot (i + 1) + j$)
    makeTriangle($m \cdot i + (j + 1)$, $m \cdot (i + 1) + (j + 1)$, $m \cdot (i + 1) + j$)
  end for
end for

blokům malá, a tudíž je není ani potřeba ořezávat pohledovým jehlanem, lze je zakódovat do jednoho vertex bufferu. Čtyři $M \times 3$ bloky vyplní mezery šířky dvou čtverců mezi $M \times M$ bloky a jsou tvořeny $12 \cdot m$ vrcholy. Bloky se dají rozdělit na levý, pravý, horní a dolní. Vytvoření jejich souřadnic již není tak zřejmé jako u $M \times M$ bloku a proto je níže definován algoritmus 4.2. Počet indexů, potřebných pro vytvoření trojúhelníkové sítě, je $48 \cdot (m - 1)$. Algoritmus 4.3 popisuje vytvoření pole indexů.

I blok

Blok tvaru I je poslední částí potřebnou pro vykreslení célé úrovně clipmapy. Tvoří ho $2 \cdot (2 \cdot m + 1)$ vrcholů, jejichž souřadnice jdou od $(0, 0)$ do $(2, 2 \cdot m + 1)$. V [1] autoři definují čtyři vertex a jeden index buffer pro tento blok, zatímco moje implementace používá pouze jeden vertex a index buffer. Z výše definovaných souřadnic a obrázku 4.2 vyplývá, že tento blok je ve vertikální poloze. Ale dle definice může tento blok ležet na všech čtyřech možných pozicích, čehož je dosaženo pomocí prohození souřadnic (x, y) v shaderu a změny backface



Obrázek 4.2: Jednotlivé typy bloků, při velikosti clipmapy $n = 15$. Souřadnice jejich vrcholů jsou označeny černými číslicemi a červenými jsou označeny indexy. (Pozn. I blok je kvůli přehlednosti posunut a může se nacházet na kterékoliv ze čtyř pozic definovaných v 2.3.)

Algoritmus 4.2 Výpočet souřadnic vrcholů $M \times 3$ bloků

```
// levý blok
for i := 2·(m−1) to 2+2·(m−1) do
  for j := 0 to m−1 do
    addVertex(j, i)
  end for
end for
// pravý blok
for i := 2·(m−1) to 2+2·(m−1) do
  for j := 3·(m−1)+2 to 1+m+3·(m−1) do
    addVertex(j, i)
  end for
end for
// horní blok
for i := 0 to m−1 do
  for j := 2·(m−1) to 2+2·(m−1) do
    addVertex(j, i)
  end for
end for
// dolní blok
for i := 3·(m−1)+2 to 1+m+3·(m−1) do
  for j := 2·(m−1)+2 to 2+2·(m−1) do
    addVertex(j, i)
  end for
end for
```

Algoritmus 4.3 Výpočet pole indexů $M \times 3$ bloků

```
offset ← 0
// pravý a levý blok
for b := 0 to 1 do
  for i := 0 to 1 do
    for j := 0 to m−2 do
      makeTriangle(m·i+j+offset, m·i+(j+1)+offset, m·(i+1)+j+offset)
      makeTriangle(m·i+(j+1)+offset, m·(i+1)+(j+1)+offset, m·(i+1)+j+offset)
    end for
  end for
  offset ← offset+(m·3)
end for
// horní a dolní blok
for b := 0 to 1 do
  for i := 0 to m−2 do
    for j := 0 to 1 do
      makeTriangle(3·i+j+offset, 3·i+(j+1)+offset, 3·(i+1)+j+offset)
      makeTriangle(3·i+(j+1)+offset, 3·(i+1)+(j+1)+offset, 3·(i+1)+j+offset)
    end for
  end for
  offset ← offset+(m·3)
end for
```

cullingu, aby bylo možné použít i stejný index buffer, který bude po změně definovat trojúhelníky ve směru proti hodinovým ručičkám. Počet indexů je $12 \cdot m$ a algoritmus jejich vytvoření je 4.4.

Algoritmus 4.4 Výpočet pole indexů I bloku

```
for i := 0 to 2·m − 1 do
  makeTriangle(2·i, 2·i + 1, 2·i + 2)
  makeTriangle(2·i + 1, 2·i + 3, 2·i + 2)
end for
```

4.3 Konečný terén

Jak již bylo uvedeno, implementace konečného terénu nebyla v původních publikacích [5] a [1] vysvětlena. Tudíž jsem navrhl dvě možná řešení, která by tento problém uspokojivě řešila. Prvním z nich je nedovolit nejméně detailní úrovni clipmapy opustit rozsah textury s výškovou mapou. Tento způsob je celkem jednoduchý na implementaci, ale jeho zásadní nevýhodou je příliš velká ztráta detailu u okraje terénu. Druhou možností, jež jsem implementoval, je nedovolit pouze nejdetailnější úrovni opustit rozsah textury, čímž je zachován detail i u okrajů terénu. Nyní vyvstává otázka, kterak vyřešit chování částí úrovní mimo platnou oblast textury. Funkce *getRectData* z třídy *TextureManager* řeší zmíněný problém kontrolou souřadnic, spadají-li do platné oblasti. Pokud je souřadnice vrcholu mimo platnou oblast je vrácena hodnota výšky, která se v terénu nemůže vyskytovat (např. záporná hodnota). Vrcholy se zápornou výškou mohou být v pixel shaderu odstraněny z dalšího zpracování. Pravděpodobně lepším řešením, než vracet zápornou hodnotu, může být vrátit první platnou hodnotu z řádku nebo sloupce, ve kterém se daná souřadnice nachází. To by mělo zabránit zkreslení terénu s mícháním výšek pomocí parametru α.

4.4 Aktualizace úrovní

Aktualizace úrovní probíhá ve funkci *UpdateClipmap* třídy *ClipmapManager* s ohledem na nevycentrovaný pohyb, což zajišťuje optimalizaci v tom, že s malým pohybem detailní úrovně není nutné aktualizovat úrovně hrubší. Princip této aktualizace je popsán algoritmem 4.5. Nejdetailnější úroveň je aktualizována s pohybem pozorovatele tak, aby zůstala vycentrována kolem něj. Hlavní podmínkou je také dodržení definovaných omezení, aby na sebe vrcholy navazovaly na okrajích mřížek úrovní.

Dále je nutné, aby se souřadnice nejméně detailní úrovně nikdy nedostala do záporných hodnot při pohybu pozorovatele. Toto omezení je definované pro snažší výpočet toroidní adresace v shaderu a je zajištěno hodnotou

$$HMOffset = \sum_{l=0}^{L-2} (m-1) \cdot 2^{l+1}, \tag{4.2}$$

kde L je počet úrovní clipmapy a m je počet vrcholů bloku $M \times M$. Tato hodnota udává vzdálenost mezi počátečními souřadnicemi nejdetailnější a nejhrubší úrovně, čímž tedy vymezí pohyb této nejdetailnější úrovně v prostoru světa. Souřadnice nejdetailnější úrovně jsou definovány v rozsahu $\langle HMOffset, HMWidth + HMOffset)$ pro osu x s šířkou $HMWidth$ a obdobně pro osu y.

Algoritmus 4.5 Aktualizace souřadnice x všech úrovní clipmapy (obdobně souřadnice y)

$x \leftarrow eyeX - N/2$
$x \leftarrow x - (x\%2)$
$clipmap[L-1].pos.x \leftarrow x$
for $l := L-2$ downto $l = 0$ **do**
  $diffx \leftarrow clipmap[l+1].pos.x - clipmap[l].pos.x$
  $W \leftarrow 1 << (L-1-l)$
  if $diffx < (M-1) \cdot W$ **then**
    $clipmap[l].pos.x \leftarrow clipmap[l+1].pos.x - M \cdot W$
  else if $diffx > M \cdot W$ **then**
    $clipmap[l].pos.x \leftarrow clipmap[l+1].pos.x - (M-1) \cdot W$
  else
    break
  end if
end for

4.5 Instancing

Použitím instancingu lze využít stejnou geometrii, tedy vertex a index buffery, pro vykreslení instancí této geometrie s různými parametry. Předáním doprovodného bufferu s parametry instancí lze podstatně zredukovat počet volání funkce pro vykreslení primitiv (v Direct3D je to funkce *DrawIndexedInstanced*). Buffer s parametry je definován pomocí již popsáné struktury *BlockInstance*. Při inicializaci je vytvořeno pole těchto struktur o velikosti počtu bloků celé hierarchie clipmapy. Počet bloků lze vyjádřit jako

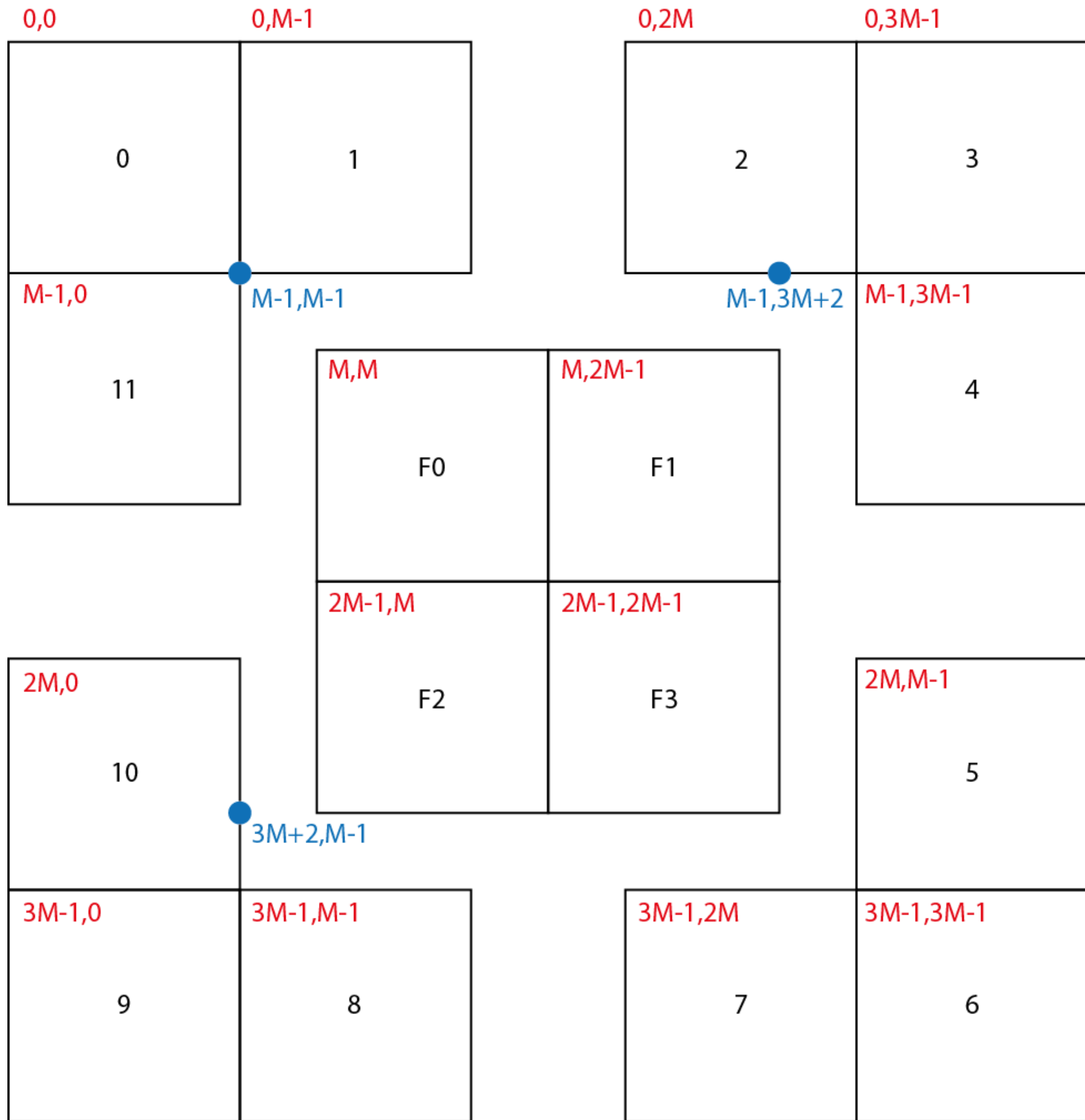
$$nBlocks = (12 \cdot L + 4) + L + (4 + 2 \cdot (L-1)). \tag{4.3}$$

Hodnota počtu bloků sestává z $12 \cdot L$ $M \times M$ bloků a dalších čtyř pro vyplnění nejdetailnější úrovně, L $M \times 3$ bloků, čtyř I bloků nejdetailnější úrovně a $2 \cdot (L-1)$ I bloků zbylých úrovní.

Klíčovým prvkem je výpočet souřadnic jednotlivých bloků a jejich velikosti mřížky. Souřadnice bloků jsou odvozeny od souřadnic jejich úrovně. K jejich přehlednějšímu výpočtu jsou definována dvě pole offsetů, jak lze vidět v následujícím kódu a také na obrázku 4.3.

```
MxMBlockPos[0]  = Vector2<unsigned int>( 0,       0);
MxMBlockPos[1]  = Vector2<unsigned int>( M-1,     0);
MxMBlockPos[2]  = Vector2<unsigned int>( M*2,     0);
MxMBlockPos[3]  = Vector2<unsigned int>( M*3 - 1, 0);
MxMBlockPos[4]  = Vector2<unsigned int>( M*3 - 1, M-1);
MxMBlockPos[5]  = Vector2<unsigned int>( M*3 - 1, M*2);
MxMBlockPos[6]  = Vector2<unsigned int>( M*3 - 1, M*3-1);
MxMBlockPos[7]  = Vector2<unsigned int>( M*2,      M*3 - 1);
MxMBlockPos[8]  = Vector2<unsigned int>( M-1,      M*3 - 1);
MxMBlockPos[9]  = Vector2<unsigned int>( 0,        M*3 - 1);
MxMBlockPos[10] = Vector2<unsigned int>( 0,        M*2);
MxMBlockPos[11] = Vector2<unsigned int>( 0,        M-1);

MxMFinestBlockPos[0] = Vector2<unsigned int>( M,       M);
MxMFinestBlockPos[1] = Vector2<unsigned int>( M*2 - 1, M);
```

```
MxMFinestBlockPos[2] = Vector2<unsigned int>( M,       M*2 - 1);
MxMFinestBlockPos[3] = Vector2<unsigned int>( M*2 - 1, M*2 - 1);
```



Obrázek 4.3: Offsety jednotlivých bloků úrovně. Souřadnice modrou barvou označují offsety I bloků.

Aktualizace

Buffer s instancemi je aktualizován od aktivní do nejméně detailní úrovně. Aktualizace probíhá v každém snímku. Možnou optimalizací by mohla být aktualizace jen při změně pozice odpovídající úrovně, a tím ušetření výpočetního výkonu CPU a GPU.

Aktualizace začíná výpočtem hodnoty g_l, definované rovnicí 4.1. Umístění $M \times M$ bloků do světových souřadnic je dáno součtem pozice úrovně a offsetu, odpovídajícímu bloku, vynásobený hodnotou velikosti rozestupu mřížky g_l. Souřadnice bloků typu $M \times 3$ jsou shodné

se souřadnicemi patřičných úrovní, protože bloky nejsou rozděleny na menší části, odpadá jejich adresování pomocí offsetů. Bloky tvaru I jsou aktualizovány v několika krocích:

- Vertikální.
  - Dva I bloky nejdetailnější úrovně. Jejich souřadnice jsou:
    $(clipmap[activeL].pos.x + g_l \cdot (M-1), clipmap[activeL].pos.y + g_l \cdot (M-1))$,
    $(clipmap[activeL].pos.x + g_l \cdot (3 \cdot (M-1)+1), clipmap[activeL].pos.y + g_l \cdot (M-1))$.
  - I bloky zbývajících úrovní.
    * Blok je vlevo:
      $(clipmap[l].pos.x + g_l \cdot (M-1), clipmap[l].pos.y + g_l \cdot (M-1))$.
    * Blok je vpravo:
      $(clipmap[l].pos.x + g_l \cdot (3 \cdot (M-1)+1), clipmap[l].pos.y + g_l \cdot (M-1))$.
- Horizontální.
  - Dva zbývající I bloky nejdetailnější úrovně se souřadnicemi:
    $(clipmap[activeL].pos.x + g_l \cdot (M-1), clipmap[activeL].pos.y + g_l \cdot (M-1))$,
    $(clipmap[activeL].pos.x + g_l \cdot (M-1), clipmap[activeL].pos.y + g_l \cdot (3 \cdot (M-1)+1))$.
  - I bloky zbývajících úrovní.
    * Blok je nahoře:
      $(clipmap[l].pos.x + g_l \cdot (M-1), clipmap[l].pos.y + g_l \cdot (M-1))$.
    * Blok je dole:
      $(clipmap[l].pos.x + g_l \cdot (M-1), clipmap[l].pos.y + g_l \cdot (3 \cdot (M-1)+1))$.

4.6 Aktualizace výškové mapy na GPU

Inkrementální aktualizace části výškové mapy uložené na GPU jako pole vertex textur je jednou z nejdůležitějších částí algoritmu a výtečnou optimalizací. Pole má velikost rovnající se počtu úrovní a velikost každé textury je n^2. Implementace inkrementální aktualizace je náročnější oproti jednoduchému zkopírování celé textury, ale právě díky malým částečným aktualizacím přináší znatelný nárůst výkonu.

Při inicializaci je vytvořen float buffer o velikosti n^2 v operační paměti. Tento buffer je použit jak pro inicializaci celého pole vertex textur, tak i pro již zmíněné inkrementální aktualizace. Dále jsou v inicializační části nahranány odpovídající oblasti mipmapy do této struktury na GPU. Oblasti potřebné pro aktuální vykreslení terénu jsou zapsány do float bufferu funkcí *getRectData* a dále funkcí Direct3D *UpdateSubresource* zkopírovány do paměti grafické karty.

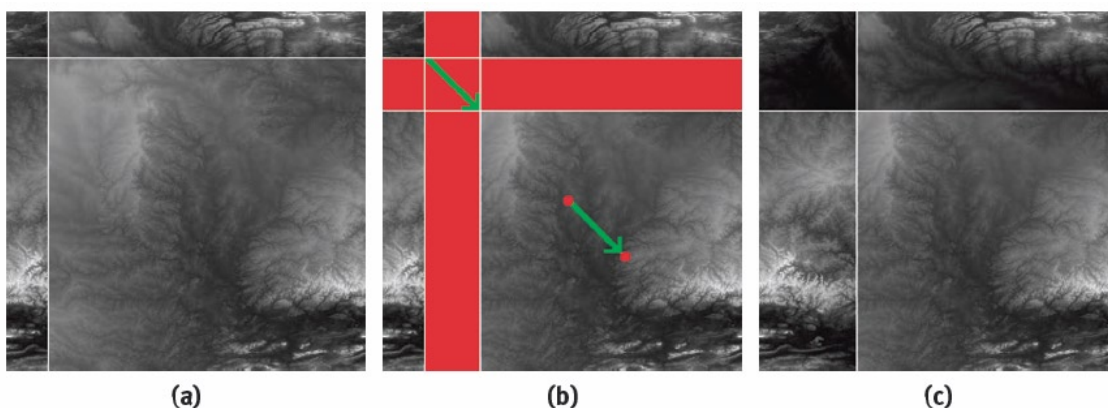
Aktualizace je provedena jen při změně pozice úrovní, a jen pro ty, které jsou označené za aktivní, čímž se uspoří další výpočetní prostředky. Aktualizace probíhá v cyklu od nejméně detailní do nejdetailnější aktivní úrovně. Prvně dochází k ověření, jestli úroveň změnila pozici od minulé aktualizace, pokud ne, pokračuje se s další úrovní. Pokud by probíhala aktualizace v opačném pořadí, bylo by možné celý cyklus v tomto místě ukončit, ale použitá implementace má výhodu v tom, že může ukončit aktualizaci po dosažení maximalní hranice aktualizovaných texelů a vždy budeme mít k dispozici minimálně aktuální verzi terénu s nejnižším detailem.

Dále se otestuje rozdíl souřadnic současných s předchozími, a pokud je alespoň jeden z rozdílů souřadnic větší než velikost clipmapy n, je nutné aktualizovat celou úroveň ve video

paměti. Tento jev ale obvykle často nenastává, protože pozorovatel nedosahuje takových rychlostí. Nejčastěji se tedy provádí inkrementální aktualizace, která je rozdělena na dvě části – horizontální a vertikální.

Horizontální aktualizace nejprve určí směr pohybu ze souřadnic. Dle směru je stanovena obdélníková oblast, která je zapotřebí k nahrání dat z mipmapy do odpovídající části vertex textury. Aktualizaci v horizontálním směru je možné provést jedním voláním funkce Direct3D *UpdateSubresource* nebo dvěma, v závisloti na pozici pozorovatele. Dvě volání jsou potřebné pro případ, kdy se výsledná oblast dat nahrává toroidně přes okraj textury. Vertikální aktualizace probíhá analogicky ke zde popsané horizontální aktualizaci.

Horizontální a vertikální aktualizace se protínají a tento průnik (obr. 4.4 b)) je aktualizován dvakrát. Proto se zde naskýtá možnost optimalizace, která by tomuto jevu předešla. Hlavním přínosem by byla u rychlého pohybu pozorovatele, kdy bude mít tato oblast větší plochu.



Obrázek 4.4: Obrázek ilustruje proces aktualizace úrovně clipmapy s výškovou mapou [1]. a) Výšková mapa před aktualizací. b) Pohyb pozorovatele a vyznačené oblasti, které se budou aktualizovat. c) Výsledek aktualizace.

4.7 Renderování

Jakmile jsou všechna data po aktualizacích připravena, je možné začít s renderováním terénu. Úrovně se renderují od nejdetailnějších aktivních ve čtyřech krocích. Nejprve jsou vykresleny všechny $M \times M$ bloky, poté $M \times 3$ bloky a naposled vertikální a horizontální bloky I. Celý terén je tedy vyrenderován, díky instancování bloků, voláním funkce pro vykreslení primitiv čtyřikrát nezávisle na počtu renderovaných úrovní. To je poměrně velký rozdíl oproti hodnotě $3 \cdot L + 2$ volání popsaných v [1]. Což je žádoucí, protože snížením počtu volání této funkce lze zvýšit rychlost renderování. Tento výsledek je dán tím, že v mé implementaci jsou bloky instancovány nejen v rámci jedné úrovně, ale napříč všemi úrovněmi clipmapy.

Na obrázku 4.5 je ukázán terén s náhodně vygenerovanou výškovou mapou pomocí grafického editoru. Je z něj patrné, že mezi některými úrovněmi došlo k nechtěnému posunu na jejich hranici, což vede k nepříjemným vizuálním artefaktům. I přes velkou snahu se nepodařilo tuto chybu odstranit. Pravděpodobnou příčinou této chyby je logika výpočtu

souřadnic úrovní clipmapy s offsetem výškové mapy konečného terénu. Kvůli této chybě nebyly implementovány přechodové oblasti, protože by vedly ke zkreslení výsledného terénu. U jedné z méně detailních úrovní dochází k výchýlení zarovnání mřížek. Další malé trhliny terénu lze odstranit implementací přechodových oblastí. Obrázek 4.6 ilustruje mřížky úrovní.

4.8 Shadery

Shadery potřebné pro renderování algoritmu GPU-Based Geometry clipmaps jsou definovány v effect souboru. Tento soubor obsahuje proměnné, rasterizační stavy, vertex a pixel shadery a techniku pro jejich nastavení.

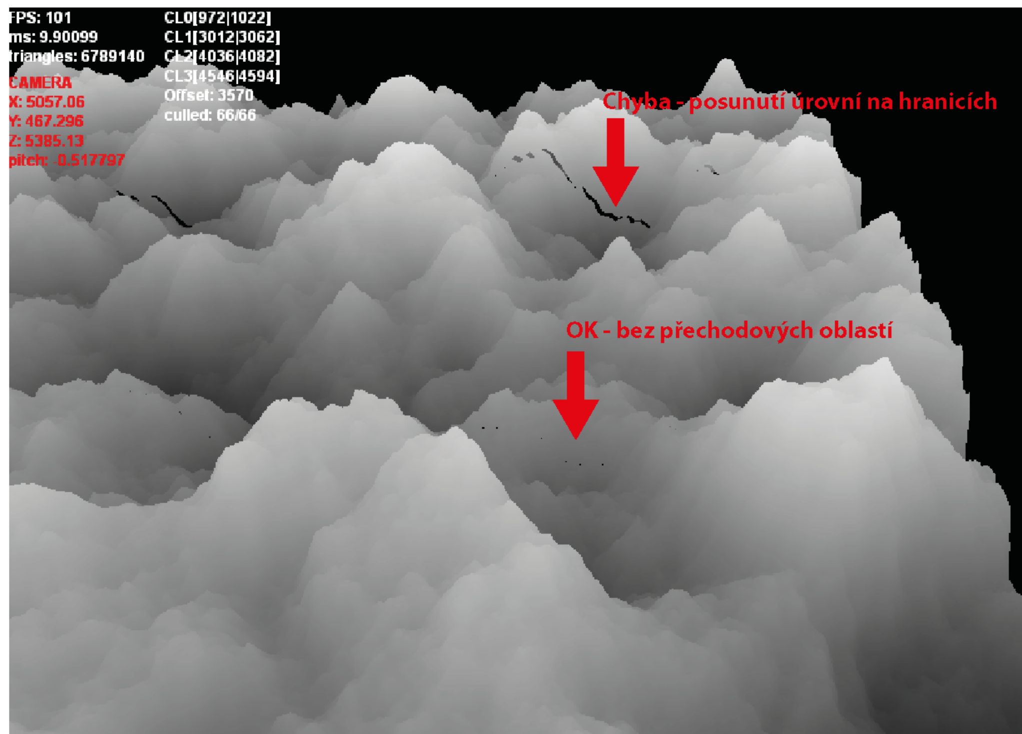
Data a struktury

Pole vertex textur s float datovým typem je nazváno *stackTexture* a obsahuje části mipmap výškové mapy. Dalšími proměnnými jsou matice WVP potřebná pro transformaci vrcholů a parametry N a L reprezentující velikost clipmapy a počet úrovní.

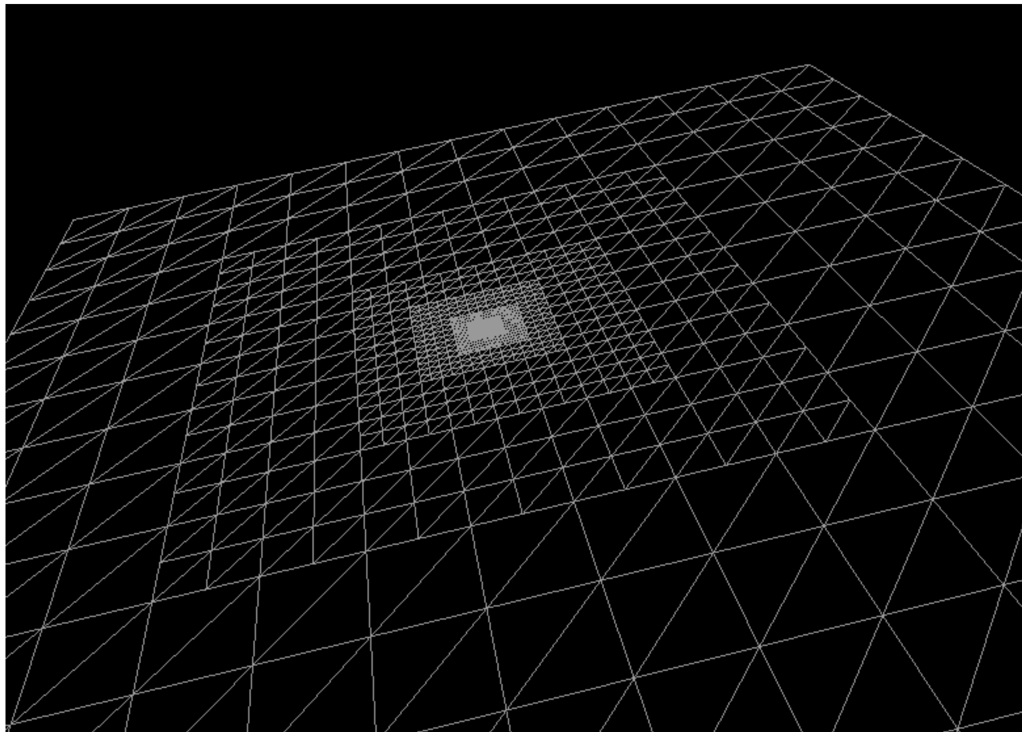
```
Texture2DArray<float> stackTexture;

cbuffer cbFrame
{
  float4x4 WVP;
};

cbuffer cbOnce
{
  int N;
  int L;
};
```

Datová struktura *VS_INPUT* obsahuje vstupní parametry pro vertex shader. Prvním parametrem je pozice vrcholu ve vertex bufferu a druhým je vektor čtyř hodnot sloužící k transformaci vrcholu do světových souřadnic. Struktura *VS_OUTPUT* předává hodnoty z vertex sahderu do pixel shaderu. Tato struktura tedy obsahuje výslednou pozici vrcholu, jeho barvu a výšku.

```
struct VS_INPUT
{
  int2 gridPos : POSITION;
  int4 scaleFactor : COLOR0;
};

struct VS_OUTPUT
{
  float4 pos : SV_POSITION;
  float4 color : COLOR0;
  float z : COLOR1;
};
```



Obrázek 4.5: Vyrenderovaný terén s velikostí clipmapy 1023 a $L = 4$.



Obrázek 4.6: Vyrenderované mřížky úrovní.

Vertex shader

Effect soubor obsahuje dvě verze vertex shaderů. Shader *DrawVSHorizontal* otáčí souřadnice vrcholů u horizontálních *I* bloků a poté volá *DrawVS* shader, který vykresluje ostatní geometrii. Úkolem tohoto shaderu je spočítat pozici vrcholu ve světových souřadnicích, z odpovídající vertex textury zjistit výšku vrcholu, vrchol transformovat pomocí matice *WVP* a předat pro další zpracování do pixel shaderu.

```
VS_OUTPUT DrawVS(VS_INPUT input)
{
  VS_OUTPUT output = (VS_OUTPUT)0;
  float2 worldPos = input.gridPos * input.scaleFactor.zw
                    + input.scaleFactor.xy;
  float l = log2(input.scaleFactor.z);
  float2 uv = (worldPos/(input.scaleFactor.z))%N/N;
  float z = stackTexture.SampleLevel(clipmapSampler,float3(uv,L-1-l),0).r;
  output.pos = mul(float4(worldPos.x, z, worldPos.y, 1), WVP);
  output.color = float4(0.6,0.6,0.6,1.0);
  output.z = z;
return output;
}

VS_OUTPUT DrawVSHorizontal(VS_INPUT input)
{
  VS_OUTPUT output = (VS_OUTPUT)0;
  input.gridPos.xy = input.gridPos.yx;
return DrawVS(input);
}
```

Pixel shader

V současné verzi aplikace je pixel shader poměrně jednoduchý. Jeho úkolem je určit výslednou barvu pixelu podle výšky vrcholu.

```
float4 DrawPS(VS_OUTPUT psInput) : SV_Target
{
  clip(psInput.z);
  psInput.color.r = psInput.z/255;
  psInput.color.g = psInput.z/255;
  psInput.color.b = psInput.z/255;
  return psInput.color;
}
```

Kapitola 5

Testy a výsledky

5.1 Návrh a implementace testů

Pro testování výkonu obsahuje aplikace jednoduchý benchmark, který umí vytvořit nový benchmark a přehrát benchmark uložený na disku. Nahrání benchmarku v diskrétních časových bodech ukládá pozici a orientaci kamery. Po ukončení nahrávání se data benchmarku uloží. Spuštěný benchmark rekoonstrujuje pohyb pozorovatele uložený v benchmark souboru pomocí lineární interpolace. Po dokončení běhu benchmarku jsou výsledky zaznamenány do souboru.

5.2 Výsledky

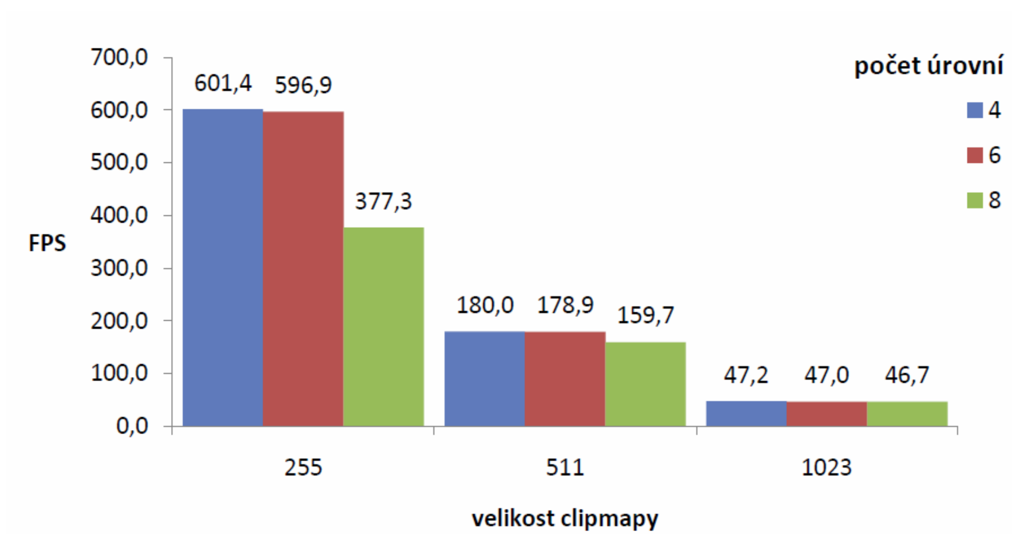
Testovací sestava

Všechny následující testy byly provedeny na sestavě s tímto hardware a software:

- Procesor: Intel Core i5-750 2,66 GHz.
- Základní deska: Asus P7P55D.
- Operační paměť: Kingston HyperX, 4 GB DDR3, 1600Mhz.
- Grafická karta: Sapphire ATI Radeon HD 4870 Vapor-X, 1 GB DDR5.
- Operační systém: Windows 7 Professional (x64).

Výsledek měření

Testování proběhlo na přiložené výškové mapě se souborem benchmarku, jenž je taktéž přiložen. Každé měření bylo provedeno pětkrát a zprůměrováno, aby se zahladily případné odchylky měření. Graf na obrázku 5.1 prezentuje výkon pro různé velikosti clipmapy a počty úrovní. Z grafu lze vypozorovat klesající počet snímků za sekundu se stoupajícím počtem renderovaných trojúhelníků, což není překvapivé. Výkon aplikace při renderování terénu s velikostí clipmapy 1023 s osmi úrovněmi přináší poměrně slušné výsledky. Aplikace při tomto nastavení renderuje přesně 13 056 036 trojúhelníků v každém snímku.



Obrázek 5.1: Graf závislosti výkonu na počtu úrovní a velikosti clipmapy.

Kapitola 6

Závěr

Cílem práce bylo implementovat algoritmus pro renderování rozsáhlého terénu. Problematika renderování terénu je poměrně obsáhlá, a proto bylo zapotřebí nejprve prostudovat teorii renderování terénu a principy moderních algoritmů. Z těchto moderních algoritmů jsem vybral Geometry Clipmaps, který přišel s novou metodou kontroly detailu terénu pomocí vnořených pravidelných mřížek. Tento algoritmus jsem podrobně prostudoval a popsal v této práci. Dále byla popsána publikovaná optimalizace GPU-Based Geometry Clipmaps, která využívá nových možností hardware a přesouvá značnou část zátěže z CPU na GPU. Jako implementační náplň mojí práce byl zvolen tento optimalizovaný algoritmus, jelikož zadáním této práce je stanoveno implementování části systému pro renderování rozsáhlého terénu na GPU.

Nejdříve byl sestaven návrh aplikace pracující se zvoleným algoritmem, podle něhož byla zrealizována aplikace pro renderování rozsáhlého terénu. Implementace některých částí algoritmu byla ztížena nejasnostmi v původní publikaci. Hlavním nedostatkem bylo nedefinované chování clipmap s konečným terénem v již zmíněné publikaci. Proto jsem přišel s vlastním řešením této problematiky. Celá implementace je v této práci podrobně popsána a může sloužit jako výchozí bod dalšího studia zabývajícího se tímto algoritmem.

Byla vytvořena aplikace pro renderování rozsáhlého terénu s využitím GPU, čímž bylo zadání diplomové práce splněno. Další možnosti pokračování tohoto projektu jsou poměrně široké. První z nich je doladění současné aplikace s odstraněním případných chyb. Také je zde dost prostoru pro další optimalizace rychlosti algoritmu a kvality výsledné vizualizace, tzn. především implementace osvětlovacího modelu a texturování povrchu terénu. Implementace osvětlovacího modelu v pixel shaderu je poměrně přímočará, avšak k jeho realizaci jsou zapotřebí normály vrcholů. Výpočet normál nebyl vzhledem k časové náročnosti implementace ostatních částí algoritmu a snaze odstranit již popsanou chybu implementován.

Literatura

[1] Asirvatham, A.; Hoppe, H.: *Terrain Rendering Using GPU-Based Geometry Clipmaps.* http://research.microsoft.com/en-us/um/people/hoppe/gpugcm.pdf, [cit. 9.5.2010].

[2] Dick, C.; Krüger, J.; Wastermann, R.: *GPU Ray-Casting for Scalable Terrain Rendering.* http://wwwcg.in.tum.de/Research/data/Publications/EG09AreasTerrain.pdf, [cit. 9.5.2010].

[3] Gerasimov, P.; Fernando, R.; Green, S.: *Shader Model 3.0 Using Vertex Textures.* ftp://download.nvidia.com/developer/Papers/2004/Vertex_Textures/Vertex_Textures.pdf, [cit. 9.5.2010].

[4] Livny, Y.; Kogan, Z.; El-Sana, J.: *Seamless Patches for GPU-Based Terrain Rendering.* http://wscg.zcu.cz/WSCG2007/Papers_2007/full/C43-full.pdf, [cit. 9.5.2010].

[5] Losasso, F.; Hoppe, H.: *Geometry clipmaps: Terrain rendering using nested regular grids.* ACM SIGGRAPH, 2004.

[6] Luebke, D.; Reddy, M.; Cohen, J. D.; aj.: *Level of Detail for 3D Graphics.* Morgan Kaufmann Publishers, 2003, ISBN 1-55860-838-9.

[7] Pouderoux, J.; Marvie, J.: *Adaptive Streaming and Rendering of Large Terrains using Strip Masks.* http://iparla.labri.fr/publications/2005/PM05/graphite05-pm.pdf, [cit. 9.5.2010].

[8] Tanner, C.; Migdal, C.; Jones, M.: *The Clipmap: A Virtual Mipmap.* ACM SIGGRAPH, 1998.

[9] Tevs, A.; Ihrke, I.; Seidel, H. P.: *Maximum Mipmaps for Fast, Accurate, and Scalable Dynamic Height Field Rendering.* http://www.tevs.eu/files/i3d08_lowres.pdf, [cit. 23.5.2010].

[10] Williams, L.: *Pyramidal Parametrics.* SIGGRAPH, 1983, ISBN 0-89791-109-1.

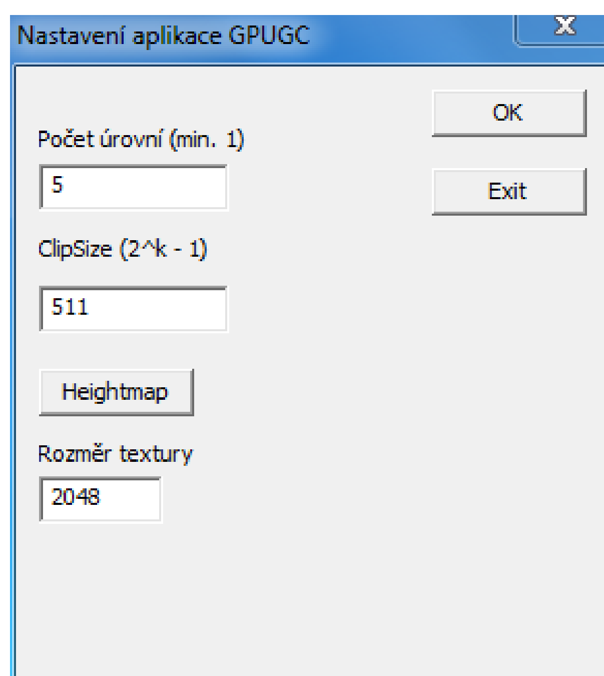
Dodatek A

Obsah CD

- bin – složka se spustitelnou aplikací a jejími daty.
- doc – technická zpráva v pdf.
  - src – složka se zdrojovými kódy technické zprávy a obrázky.
- poster – plakát v pdf.
- src – zdrojové kódy.

Dodatek B

Ovládání aplikace



Obrázek B.1: Dialog pro nastavení aplikace.

Na obrázku B.1 je ukázáno okno pro nastavení aplikace. Lze změnit počet úrovní, velikost clipmapy (maximálně 1023). Dále je možné vybrat výškovou mapu, pro kterou je nutné zadat její rozměr. Výchozím nastavením je použita výšková mapa přiložéná k aplikaci.

- kurzorové šipky – pohyb dopředu, dozadu, úkrok doleva, úkrok doprava.
- A, Z – pohyb dolu, nahoru.
- R, P – vytvoření benchmarku, spuštění benchmarku.
- ESC – ukončení aplikace.
- +, - numerické klávesnice – aktivace, deaktivace úrovní.

Dodatek C

Programová dokumentace

C.1 Třída ClipmapManager

Veřejné funkce

- `ClipmapManager(unsigned int _N, ID3D10Device* _device, unsingned int _L, int _HMSize, char* filename)`

Konstruktor třídy, inicializuje proměnné.	
parametr	**popis**
_N	velikost clipmapy
_device	ukazatel na Direct3D10 device
_L	počet úrovní
_HMSize	rozměr výškové mapy
_filename	jméno souboru s výškovou mapou

- `~ClipmapManager()`

Destruktor třídy, uvolňuje alokované proměnné.

- `void Init()`

Inicializuje všechny datové struktury.

- `void Update(float delta)`

Aktualizuje všechny datové struktury.	
parametr	**popis**
delta	delta čas mezi snímky v ms

- `void Draw()`

Vykresluje terén.

- `void DecreaseLevel()`

Deaktivuje úroveň.

- `void IncreaseLevel()`

Aktivuje úroveň.

- `void setCamera(Camera* c)`

Nastaví použitou kameru.	
parametr	**popis**
c	ukazatel na kameru

- `unsigned int getOffset()`

  Vrací hodnotu HMOffset.

- `unsigned int getOffset2()`

  Vrací hodnotu HMOffset2.

Veřejné proměnné

- `long triangleCnt` – počet renderovaných trojúhelníků.
- `string clipcoords` – řetězec pro výpis souřadnic úrovní.

Soukromé funkce

- `void CreateGeometry()`

  Vytváří vertex a index buffery úrovní.

- `void ReleaseGeometry()`

  Uvolňuje vertex a index buffery.

- `void CreateMxM()`

  Vytváří geometrii pro $M \times M$ bloky.

- `void CreateMx3()`

  Vytváří geometrii pro $M \times 3$ bloky.

- `void CreateI()`

  Vytváří geometrii pro I bloky.

- `void CreateClipmap()`

  Alokuje pamet pro L úrovní.

- `void InitClipmap()`

  Inicializuje výchozí souřadnice úrovní.

- `void UpdateClipmap()`

  Aktualizuje souřadnice úrovní podle pohynu pozorovatele.

- `void ReleaseClipmap()`

  Uvolní paměť úrovní.

- `void CreateStackTexture()`

  Vytvoří pole vertex textur úrovní.

- `void InitStackTexture()`

  Inicializace pole vertex textur výškovými daty.

- `void UpdateStackTexture()`

  Inkrementální aktualizace pole vertex textur.

- `void ReleaseStackTexture()`

  Uvolní paměť pole vertex textur a updateBuffer.

Soukromé proměnné

- `float eyeX` – pozice pozorovatele na ose X.
- `float eyeY` – pozice pozorovatele na ose Y.
- `unsigned int N` – velikost clipmapy.
- `unsigned int M` – počet vrcholů bloku M.
- `unsigned int L` – počet úrovní.
- `unsigned int activeL` – index nejdetailnější aktivní úrovně.
- `unsigned int activeLcnt` – počet aktivních úrovní.
- `int HMSize` – velikost textury výškové mapy.
- `int HMWidth` – šířka textury výškové mapy.
- `int HMHeight` – výška textury výškové mapy.
- `char* HMPath` – cesta k souboru s výškovou mapou.
- `int HMOffset` – posunutí terénu.
- `int HMOffset2` – posunutí terénu se zarovnáním nejméně detailní úrovně na mřížku.
- `Camera* camera` – ukazatel na kameru.
- `Clipmap* clipmap` – pole úrovní.
- `TextureManager* HM` – instance třídy $TextureManager$ spravující data výškové mapy.
- `ID3D10Device* device` – device slouží pro práci s grafickou kartou.
- `ID3D10Effect* effect` – rozhraní pro práci s effect souborem.
- `ID3D10EffectTechnique* technique` – technika z effect shaderu pro renderování.
- `ID3D10EffectMatrixVariable WVP` – reprezentuje matici pro transformaci v shaderu.
- `ID3D10EffectShaderResourceVariable* stackTextureSRVar` – rozhraní pro práci se shader zdroji.
- `ID3D10EffectScalarVariable* clipSize` – předává velikost clipmapy do shaderu.

- ID3D10EffectScalarVariable* levels – předává počet úrovní do shaderu.
- ID3D10Texture2D* stackTexture – pole vertex textur.
- ID3D10ShaderResourceView* stackTextureSRV – rozhraní pro přístup ke zdrojům v shaderu.
- ID3D10InputLayout* MxMIL – nastaví formát vstupních dat do shaderu.
- ID3D10Buffer* MxMVB – vertex buffer.
- ID3D10Buffer* MxMIB – index buffer.
- ID3D10InputLayout* Mx3IL – nastaví formát vstupních dat do shaderu.
- ID3D10Buffer* Mx3VB – vertex buffer.
- ID3D10Buffer* Mx3IB – index buffer.
- ID3D10Buffer* IVB – vertex buffer.
- ID3D10Buffer* IIB – index buffer.
- ID3D10buffer* instanceGPU – buffer s instancemi na GPU.
- BlockInstance* instanceCPU – buffer s instancemi v operační paměti.
- Vector2<unsigned int> MxMBlockPos[12] – .
- Vector2<unsigned int> MxMFinestBlockPos[4] – .
- D3DXMATRIX mWorld – transformační matice terénu.
- D3DXMATRIX fWVP – transformační matice.

C.2 Třída TextureManager

Veřejné funkce

- TextureManager(char* _file, unsigned int _width, unsigned int _height, unsigned int _mipCnt)

Konstruktor inicializuje proměnné do výchozího stavu.	
parametr	**popis**
_file	soubor s texturou
_width	šířka textury
_height	výška textury
_mipCnt	počet mipmap

- ~TextureManager()

Destruktor smaže vytvořené mipmapy.

- void Init()

Vytvoří požadovaný počet mipmap.

- `void SaveMipmap(unsigned int mipLevel)`

Uloží požadovanou mipmapu na disk.	
mipLevel	index mipmapy

- `void getRectData(float* buffer, RECT r, unsigned int mipLevel, int HMOffset, int L, int N )`

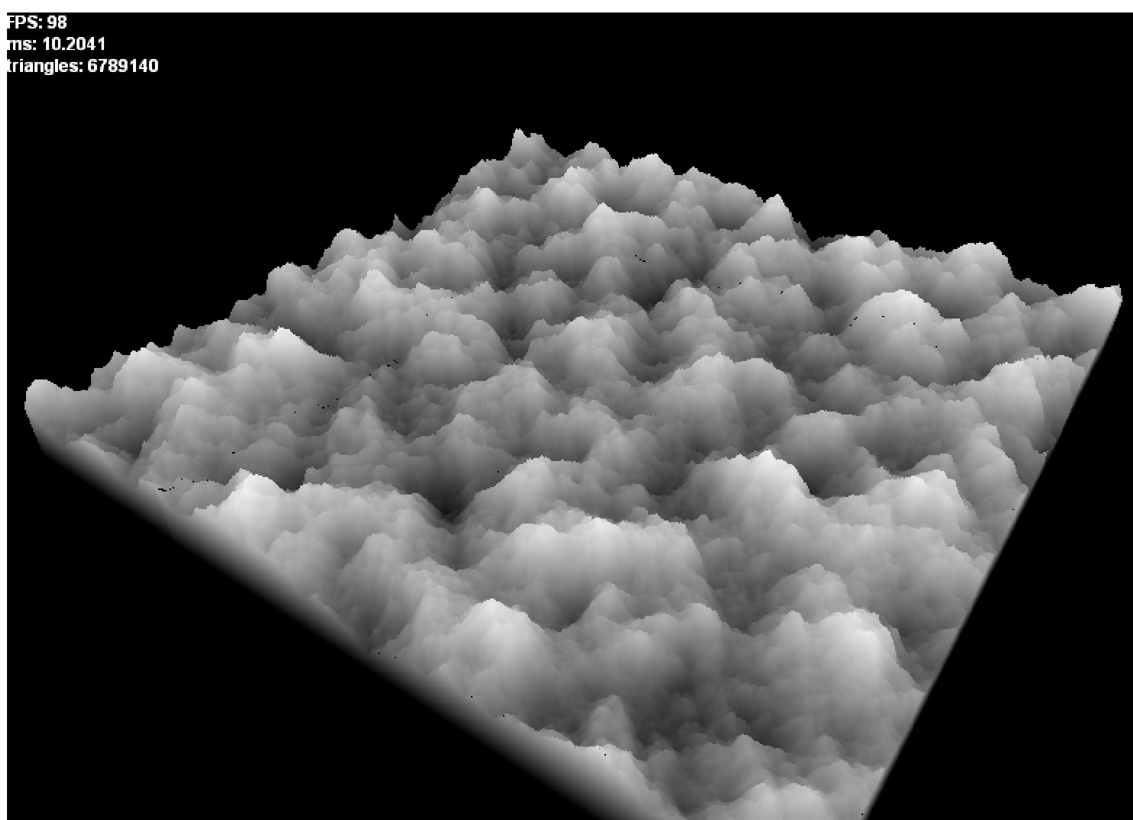
Naplní předaný buffer daty z mipmapy pomocí toroidní adresace.	
buffer	cílový buffer
r	souřadnice čtverce vymezující požadovaná data
mipLevel	index mipmapy
HMOffset	offset terénu
L	index počet úrovní
N	velikost clipmapy

Soukromé proměnné

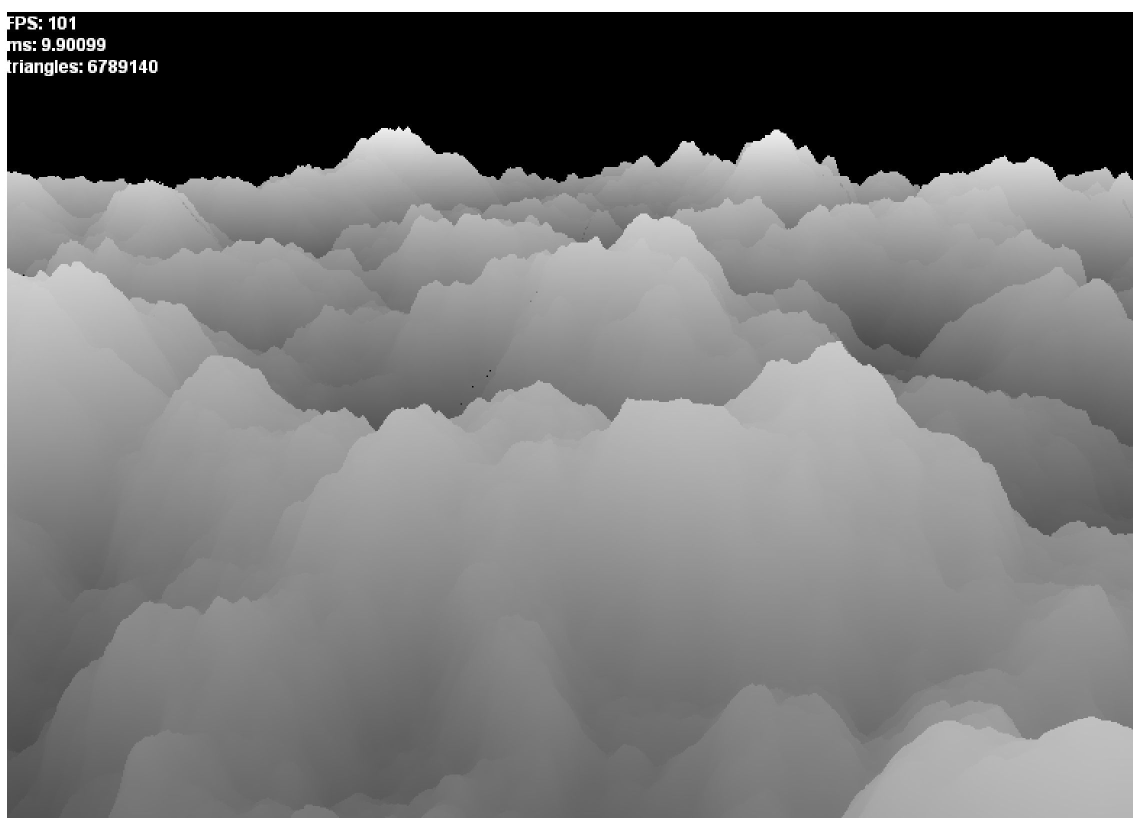
- `char* file` – soubor s texturou.
- `unsigned int width` – šířka textury.
- `unsigned int height` – výška textury.
- `unsigned int mipCnt` – počet mipmap.
- `Mipmap** mipmap` – ukazatel na pole s daty mipmap.

Dodatek D

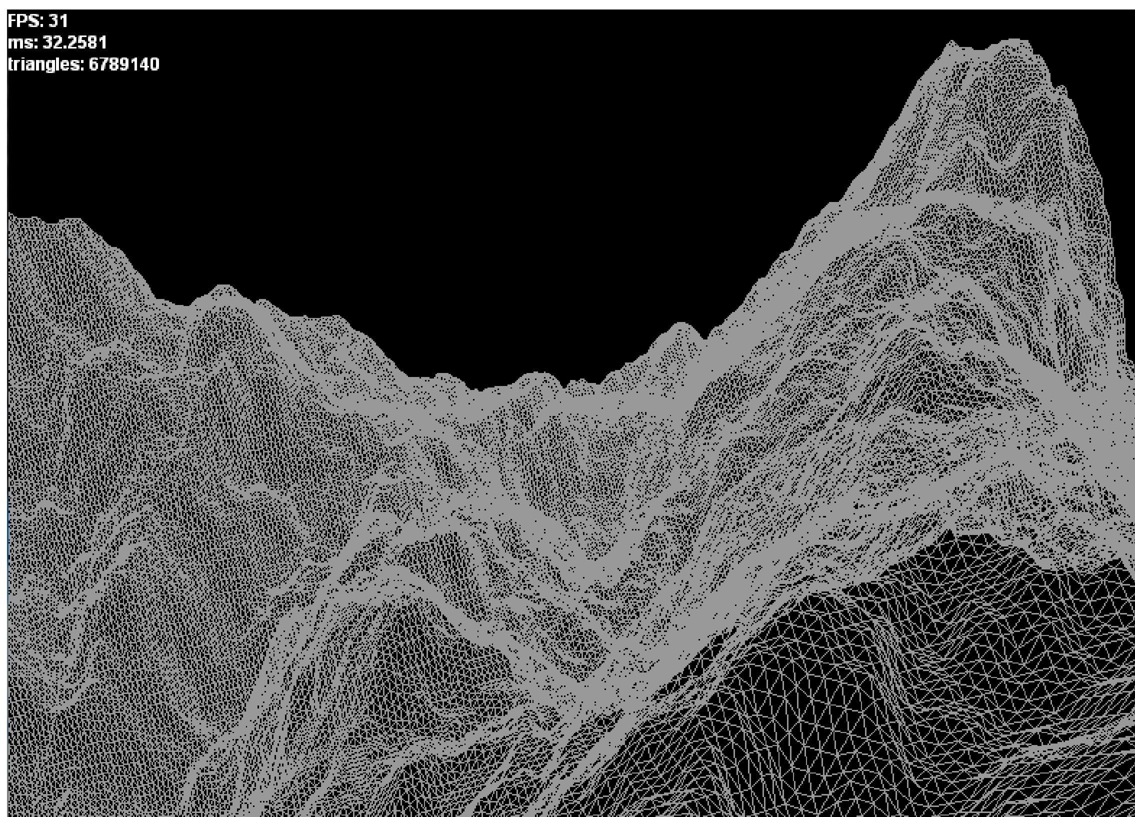
Rendery



Obrázek D.1: Pohled na terén z dálky, $N = 1023$, $L = 4$.



Obrázek D.2: Bližší pohled na terén.



Obrázek D.3: Drátový model terénu.