Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Roman Margold

Zpracování terénu na moderních GPU

Kabinet software a výuky informatiky
Vedoucí diplomové práce: RNDr. Josef Pelikán
Studijní program: Informatika

Rád bych poděkoval RNDr. Josefu Pelikánovi za počáteční nasměrování, Nicolasi Thibierovi za konzultace ohledně interního chování hardwaru, Mgr. Václavu Krajíčkovi a Mgr. Pavlu Celbovi za možnost diskutování řešených problémů.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 14. 12. 2007 Roman Margold

Obsah

1 Úvod **9**

1.1 Cíle 10

1.2 Struktura textu 12

1.2.1 Editační poznámka 13

2 Zobrazování terénu **14**

2.1 Datová reprezentace terénu 14

2.2 Zjednodušování geometrie 17

2.2.1 Statický a dynamický LOD 18

2.2.2 Defekty při LOD technikách 19

2.2.3 Pohledově závislý LOD 20

2.2.4 Metrika chyby 21

2.2.5 Shora-dolů nebo zdola-nahoru 22

2.3 Známé postupy 22

2.3.1 Obecné TIN 24

2.3.2 Progressive meshes 24

2.3.3 Kvadrantový strom 27

2.3.4 Binary triangle tree 30

2.3.5 Statický a hybridní LOD 35

2.3.6 Clipmap 38

2.3.7 Alternativní přístupy 42

3 Zjemňování geometrie **44**

3.1 Parametrické plochy 44

3.2 Subdivision surfaces 48

3.2.1 Catmull-Clark 50

3.2.2 Loop 52

3.2.3 Butterfly 53

3.2.4 Jiná schémata 55

4 Naše řešení **59**

4.1 Iniciální návrh řešení 59

4.1.1 Volba hlavního přístupu 59

4.1.2 Volba datové reprezentace 60

4.2 Adaptivní dělení terénu 62

4.2.1 Úvahy pro volbu dělícího schématu 63

4.2.2 Experimentování se subdivision surfaces 65

4.2.3 Implementační poznámky 69

4.3 Použití fixní mřížky 73

4.3.1 Struktura trapezmap 74

4.3.2 Struktura ringmap 76

4.3.3 Ořezávání pohledovým jehlanem 80
4.4 Generování dat 82
4.4.1 Generování reziduí 82
4.4.2 Generování normálové mapy 83
4.5 Zpracování neomezeného terénu 87
4.5.1 Renderer 89
4.5.2 Server 90
4.5.3 Loader 91
4.5.4 Worker 93
4.6 Měření a výsledky 93
4.6.1 Rekonstrukce výšek 93
4.6.2 Výkon 97
4.7 Budoucí práce 99
4.7.1 Modifikace terénu 99
4.7.2 Potlačení aliasu 100
4.7.3 Mapování textury na terén 101
4.7.4 Jiná vylepšení 102

5 Závěr 103

Literatura 106

Dodatek 114
Obsah CD 114
Uživatelský manuál 114
Prerekvizity 114
Terra Artifex 115
PerlinNoiseGenerator 117
HeightMapProcessor 118

Název práce: *Zpracování terénu na moderních GPU*
Autor: *Roman Margold*
Katedra (ústav): *Kabinet software a výuky informatiky*
Vedoucí diplomové práce: *RNDr. Josef Pelikán*
e-mail vedoucího: *josef.pelikan@mff.cuni.cz*
Abstrakt: *Prudký vývoj grafického hardwaru umožňuje nacházet stále nové techniky pro zobrazování terénu. Dříve používané postupy jsou postaveny na silné redukci geometrie prováděné na CPU. Novější postupy se snaží přenést zátěž na GPU a ponechat CPU volné pro jiné výpočty, což je obzvlášť důležité v počítačových hrách. Většina takových postupů je však limitována užitím statických dat nebo omezenou velikostí dat. Nový postup, jež v této práci navrhujeme, umožňuje modifikovat data za běhu a je snadno použitelný na zobrazování rozsáhlých krajin. Je celý implementován na GPU, kromě ořezávání pohledovým jehlanem. Ořezávání je stále prováděno na CPU, ale díky navrženému schématu vzorkování terénu je zcela triviální a vysoce efektivní. Použitá dvouúrovňová reprezentace dat nabízí poměrně snadné zacházení, a přitom nevylučuje takové operace jako je náhlá změna zorného úhlu nebo změna směru pohledu pozorovatele, se kterými mají jiné postupy problémy. Navíc má díky využití blokové komprese nízké paměťové nároky.*
Dále byl navržen a implementován univerzální načítací mechanismus pro asynchronní získávání dat z externího média. Systém byl optimalizován pro čtení z médií se sériovým přístupem a zároveň paralelní zpracování násobných požadavků na data. Byl využit pro průběžné načítání dat terénu, která se nevejdou do operační paměti.
Klíčová slova: *Zobrazování terénu. LOD. Neomezená data. Geometry shader. Subdivision surfaces.*

Title: *Terrain processing on modern GPU*
Author: *Roman Margold*
Department: *Department of Software and Computer Science Education*
Supervisor: *RNDr. Josef Pelikán*
Supervisor's e-mail address: *josef.pelikan@mff.cuni.cz*
Abstract: *Recent development in graphics hardware opened the possibility for new terrain rendering techniques. Former techniques are based on rapid geometry reduction performed on CPU. Recent approaches are moving the load from CPU to GPU, thus keeping CPU available for other tasks, which is especially important for game development. A majority of these approaches is restricted to static datasets or limited data size. We introduce a novel approach which is capable of modifying data during runtime and is easily applicable to potentially infinite landscapes. It is implemented entirely on GPU, except view frustum culling, which is still performed by CPU. However, thanks to the proposed terrain sampling scheme it is trivial and extremely efficient. Employed two-level data representation offers simple implementation without loss of functionality such as sudden change of visual angle or view direction. Those are common problems of other approaches. We also adopted block compression to keep memory consumptions low.*
Further, a general loading mechanism has been designed and implemented in order to allow asynchronous data read from external medium. This system has been optimized for reading with serial access, although data demands are processed in parallel. The system has been used for continuous terrain data retrieval and management to handle out-of-core datasets.

Keywords: *Terrain visualization. LOD. Out-of-core. Geometry shader. Subdivision surfaces.*

1 Úvod

Interaktivní zobrazování jakýchkoli dat je široce škálovatelná a komplexní úloha. Není možné na každý typ dat použít stejný přístup – jinak budeme postupovat při zobrazování městských ulic v interaktivním průvodci městem, jinak k předvádění automobilu v reklamní aplikaci, jiný přístup zvolíme pro vizualizaci proudění vzduchu kolem křídel letadla, pohybu oblačností nad kontinentem nebo námahy materiálu při jeho zatížení. Z těchto příkladů nejen vidíme, že zvolený postup silně závisí na typu zobrazovaných dat, ale také že je vázán celkovým cílem či záměrem aplikace, v níž je použit. Proto vznikla menší či větší odvětví počítačové grafiky, která se specializují výhradně na konkrétní typ dat. Výjimkou v tomto směru není ani zobrazování terénu, které je pro běžné velikosti vstupních dat odedávna náročným a vyzývavým úkolem.

Existuje celá řada běžně rozšířených oblastí, ve kterých se (interaktivního) zobrazování terénu využívá. Mezi ty nejčastější patří patrně počítačové hry a letecké či jiné simulátory v trenažérech využívaných armádou, výcvikovými středisky apod. Najdeme však i jiná uplatnění, kupříkladu při simulaci povodňových aktivit v zátopových oblastech je interaktivnost důležitým faktorem použitelnosti programu. V mnohých oblastech uplatnění je navíc nutné, aby zobrazování terénu vzalo co nejmenší množství některých prostředků (paměti, procesorového času, datového toku,…). Nemluvíme tu o obecné snaze snížit nároky na systémové prostředky, ale o konkrétních potřebách dané aplikace. Uveďme si příklad využití procesoru ve výše zmíněných aplikacích – zatímco v leteckém simulátoru máme zpravidla většinu procesorového času k dispozici k vykreslení terénu o rozloze stovek čtverečních kilometrů, v počítačové hře máme kvůli výpočtům herních mechanismů (chování umělých bytostí, fyzikální interakce objektů,…) a celkové složitosti zobrazované scény jen zlomek procesorového času. Při simulaci povodňových aktivit je dokonce zobrazování pouze okrajovou záležitostí.

Z právě uvedeného nám vyplývají další faktory pro zobrazování terénu. Můžeme je charakterizovat otázkami, které si musíme položit před zahájením jakéhokoli návrhu algoritmu a designu datových struktur. Například: Jaká bude běžná velikost zobrazovaného terénu? Jaká bude maximální velikost zobrazovaného terénu? Jaká přesnost je požadována na jeho grafickou reprezentaci? Jak daleké okolí bude pozorovatelné v jeden okamžik? Jak velkou rychlostí bude možné měnit pozici pozorovatele? Bude povolena změna polohy pozorovatele pouze spojitá nebo i skoková? Z jakého média budou terénní data dostupná (pevný disk, optické médium, lokální síť, Internet,…)? Jaké jsou parametry cílové platformy? … Mohli bychom ještě chvíli pokračovat, avšak přesnějšímu rozboru případu se budeme věnovat později. Ne vždy je nutné znát přesnou odpověď na všechny takovéto otázky, ale je zřejmé, že specifičtější zadání vede k užšímu poli možností při návrhu řešení, zatímco méně přesné zadání vede ke generalizaci řešení, jeho širšímu použití, ale také slabším výsledkům v konkrétních případech.

Zobrazování terénu bylo již věnováno velké množství článků a publikací, avšak stále je živou a otevřenou kapitolou, jejíž charakter se s časem mění. K hlavním změnám dochází souběžně se

změnami v architektuře a výkonu hardware. Například v (1) se autoři zmiňují, že tehdejší (1995) grafická stanice dokáže zobrazit přibližně 300 000 trojúhelníků za vteřinu. Pokud se usneseme, že za „interaktivní" zobrazování budeme považovat takové, kde dosáhneme alespoň 30 snímků za vteřinu, jsme na takovém stroji schopni interaktivně zobrazovat terén složený maximálně z 10 tisíc trojúhelníků. Tento fakt vedl celé roky k agresivním redukcím počtu zobrazovaných trojúhelníků, ať už v off-line kroku při předzpracování dat, či v reálném čase za běhu programu. S trvalým nárůstem výkonu počítačů, a pak především s masivním nástupem grafických akcelerátorů (pro srovnání – dnes běžně dostupné grafické akcelerátory jsou schopné transformovat stovky milionů až miliardu trojúhelníků za vteřinu) dedikovaných výhradně zobrazování dat, se nejen adekvátně zvyšuje kvalita výstupů, ale mění se i priority při návrhu algoritmů. Proto vedle objemu zobrazovaných dat je dnes důležitým charakteristickým prvkem také úroveň paralelního zpracování.

1.1 Cíle

V nedávné době se architektura běžně dostupných grafických akcelerátorů (graphics processing unit, dále jen GPU) dočkala velké reformy související s návrhem nového Shader Modelu, konkrétně verze 4.0[1]. Tato téměř revoluční změna významně rozšířila možnosti programovatelnosti GPU, a vytvořila tak prostor pro vznik nových algoritmů a postupů, případně efektivnější implementaci dříve navrhnutých. Stejně jako s příchodem předchozích Shader Modelů tedy vznikla příležitost přenést řešení dalších problémů z CPU na GPU. Primárním cílem této práce je tedy podívat se na dnes i dříve používané postupy zobrazování terénu a zhodnotit možnost využití nové architektury GPU pro jejich implementaci, případně navrhnout postup nový.

V předešlých odstavcích byl rozebrán význam co nejpřesnější znalosti parametrů cílového systému a způsobu aplikace navrhovaného řešení. S ohledem na to si právě zde uvedeme hlavní požadavky. Předně chceme, aby navržený postup zobrazování terénu byl dobře aplikovatelný v počítačových hrách. To samo o sobě už vymezuje určitý omezený prostor pro řešení, ale stále ještě poměrně široký. Vezměme si dva odlišné typy her, na kterých provedeme stručné srovnání: RTS[2] a RPG[3]. V RTS hře se hráči obvykle nabídne pouze pohled z malé výšky přímo seshora, případně pod dostatečně velkým úhlem, aby neviděl moc velkou část terénu. Ten navíc nebývá příliš členitý a tvoří spíše minoritní prvek scény. Naproti tomu RPG žánr vám běžně nabídne pohled na rozlehlou krajinu bez omezení zorného úhlu, a i když dohled v takové krajině je zatím silně limitován (500-1 500 metrů), velikost celého dostupného světa v takových hrách se pohybuje v řádu desítek až stovek kilometrů (obzvlášť velká prostředí nabízí stále více oblíbený MMORPG žánr, kde v jednom světě hrají i tisíce hráčů).

---

[1] V době psaní této práce je již platný návrh Shader Modelu 4.1, ten však oproti verzi 4.0 nepřináší velké změny.

[2] Z angl. Real Time Strategy volně přeložíme jako strategickou hru probíhající v reálném čase.

[3] Angl. Role Playing Game se volně překládá jako „hra na hrdiny". Hráč obvykle hraje za jednu postavu nebo velmi malou skupinku postav, s nimiž je svázán pohled na scénu.

Nechceme se přímo vázat na konkrétní typ hry, případně na konkrétní zadání parametrů, naopak chceme navrhnout systém, který by byl ve větší míře škálovatelný alespoň v herním prostoru. Navíc je dobré si uvědomit, že řada číselných parametrů je vzájemně svázaných. Kupříkladu pokud slevíme na požadované přesnosti reprezentace, můžeme zvětšit velikost terénu uloženého ve stejném množství paměti. Z tohoto pohledu chceme tedy nabídnout řešení, které bude parametrizovatelné a aplikovatelné v širším spektru her. Přesto si definujeme určité cíle, které se nám zdají být zajímavé, nebo přímo vyplývají z předpokládaného nasazení, tj. použití v počítačových hrách:

1. Zpracování potenciálně nekonečného terénu.
2. Umožnění modifikací terénu za běhu aplikace.
3. Snadná škálovatelnost hardwarové náročnosti při startu aplikace.

Kromě těchto bodů, které považujeme za klíčové, máme ještě další požadavky, od kterých však neočekáváme, že je splníme všechny najednou. Přesto dosažení každého jednoho považujeme za významný bonus libovolného řešení:

4. Minimální nároky na čas CPU.
5. Možnost paralelizace a tedy využití více hardwarových vláken.
6. Snadná škálovatelnost hardwarové náročnosti za běhu aplikace.
7. Snadná implementace.

Většina bodů je patrně srozumitelná, ale bude vhodné zmínit také pohnutky, které nás vedly k jejich stanovení. Proto je rozeberme více do hloubky.

Pod pojmem *nekonečného terénu* si nepředstavujeme procedurálně za běhu generovaná data, ačkoli to samo o sobě je pěkným úkolem, ale data, která se obecně nevejdou do operační paměti systému, a je tedy nutné získávat je z nějakého média za běhu (mluvíme o tzv. *out-of-core* algoritmu). To je aktuální požadavek téměř všech dnešních her RPG nebo MMORPG žánru, ale i mnohých jiných. Také je to oblast zmapovaná podstatně méně než zobrazování terénních dat, která jsou deterministicky načítána v konkrétních situacích (např. přechod z jedné úrovně hry do jiné, automatická cesta mezi oddělenými lokacemi), a navíc, nebo právě proto, se nám tato úloha zdá být komplexnější a zajímavější.

Při zobrazování statických dat se nabízí široké pole optimalizačních algoritmů (již implementovaných v komerčních či volně šiřitelných knihovnách), které se dají použít k předzpracování dat v době vývoje, a není složité taková data potom zobrazovat s vysokou frekvencí za běhu programu. Situace se zkomplikuje požadavkem na *modifikovatelnost dat* – tím

výrazně omezíme možnosti jejich předzpracování a větší tíha tedy padne na běh programu. V současné době nám není známa žádná hra, která by splňovala první dvě podmínky[4].

Podmínka snadné *škálovatelnosti výkonu* pro různé konfigurace cílového hardwaru je pro hry kritická. V prostoru herních konzolí jsou sice daleko přesnější specifika platformy a má význam optimalizovat implementaci algoritmu pro jeden zvolený systém, ale v pojetí celkového návrhu algoritmu by se to nemělo příliš odrazit, neboť ten by měl být použitelný v různých herních projektech pro různé cílové platformy (s ohledem na výše definované cíle). Požadavek jsme v jiné formě přidali i do kategorie méně důležitých cílů – dynamická škálovatelnost výkonu za běhu je velice užitečná pro udržování plynulého pohybu či přehrávání animací. Zátěž systému se totiž dynamicky mění v závislosti na činnosti hráče a jeho interakce s herním prostředím. Při vývoji se tak musí dělat odhady nejen průměrné zátěže systému, ale také maximální (minimální je vždy irelevantní). Pokud jsme schopni různé části kódu dynamicky konfigurovat tak, aby prováděly svoji činnost rychleji (za cenu snížení kvalitativních kritérií), můžeme kritické situace, kdy je zátěž systému nejvyšší, eliminovat vhodnou konfigurací prováděných úkonů. Takto definovaný požadavek však považujeme pouze za užitečný, nikoli nutný.

Další dva požadavky spojené s hardwarovými nároky souvisí s menšími změnami v architektuře cílových počítačů. Jedním ze současných trendů je šetření cyklů CPU na úkor GPU. Je to dáno tím, že výkon GPU narůstá v současné době větší měrou a že je tato jednotka dedikována jedinému úkolu (totiž zobrazování), zatímco CPU se musí starat nejen o všechny ostatní úkoly v aplikaci, ale také o zásobování GPU správnými daty. Často se tak stává, že CPU zaměstnané herní logikou není schopno dodávat data k zobrazení dostatečně rychle a výkon GPU závislého na těchto datech je nevyužit. Jiný trend, jímž je paralelizace, se teprve zvolna rozvíjí a souvisí s nástupem počítačů s více procesory nebo více jádry na pozice běžných herních strojů. Výše uvedené dva požadavky, tedy minimální nároky na CPU a možnost paralelizace, se v jistém smyslu vylučují navzájem. Vysloveno explicitně - pokud se nám podaří navrhnout zobrazování terénu tak, aby zatěžovalo CPU minimálně, je zbytečné řešit ho paralelně ve více procesorových vláknech. Při návrhu přesto chceme brát v potaz obě kritéria a zohlednit je dle možností.

Konečně významnou výhodou každého algoritmu, zvláště pak aplikovaného v komerčním sektoru, je jednoduchost implementace. To v praxi znamená snížení ceny vývoje nebo získání času pro vyzkoušení více algoritmů, z nichž se vybere jeden podle nejlepších výsledků pro daný projekt. Pro tuto práci je to však kritérium minoritní, nechceme se bránit ani komplexním řešením, která budou v intencích dříve zmíněných cílů.

1.2 Struktura textu

Hlavní text jsme rozdělili do tří větších celků. V prvním (kapitola 2) se věnujeme základním kamenům zobrazování terénu. Vysvětlujeme důležité pojmy s tím související a provádíme širší průzkum algoritmů, které se pro zobrazování terénu používaly nebo používají.

---

[4] Ve hrách absence těchto prvků nemusí být nutně spojena se zobrazováním. S dynamickým terénem se kupříkladu komplikují i výpočty interakce objektů s terénem z fyzikálního hlediska.

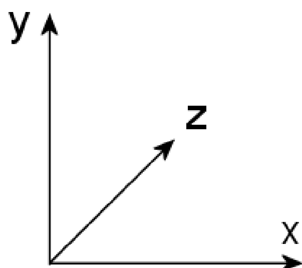
V kapitole 3 se zjednodušeně věnujeme postupům zjemňování geometrie, ze kterých největší pozornost věnujeme subdivision surfaces. Ukazujeme, jakou souvislost mohou mít se zobrazováním terénu a jak jich chceme využít. Rozebíráme zde také nejčastěji používaná dělící schémata a stručně shrnujeme práce, které se této problematice více věnují.

Kapitola 4 představuje hlavní přínos naší práce. Nejprve se v ní věnujeme rozboru rozhodnutí dělaných za účelem návrhu zobrazovacího postupu. Poté sledujeme kroky, které jsme dělali při hledání vhodné cesty pro nalezení takového postupu, až se dostaneme k samotnému popisu řešení, které v této práci nabízíme. Na konci kapitoly provádíme rozsáhlé testování a srovnávání výsledků získaných různými konfiguracemi, a také navrhujeme různé další cesty výzkumu a vývoje.

V kapitole 5 provádíme stručné shrnutí našich výsledků postavené na předešlém rozsáhlejším rozboru.

1.2.1 Editační poznámka

Existuje několik různých souřadných systému popisujících trojrozměrný prostor, které různé skupiny lidí (podle jejich vědního oboru) považují za standard. Chceme se vyhnout pochybnostem a nejasnostem, proto hned zde v úvodu zvolíme jeden souřadný systém a ten budeme používat v celém textu. To se týká i případných převzatých algoritmů. Zvolili jsme levoruký kartézský systém souřadnic s osou x orientovanou vodorovně vpravo a osou y svisle vzhůru jak ukazuje Obrázek 1.1.



Obrázek 1.1: Použitý systém souřadnic.

2 Zobrazování terénu

Zobrazování terénu již byla věnována nejedna publikace a ve hrách byl implementován nejeden algoritmus, který problém lépe či hůře řešil s ohledem na dané prostředky. Starší algoritmy často nejsou dnes plně použitelné, protože v době jejich vzniku byla významnější jiná kritéria. My jsme se však na tento jev podívali z jiné strany – mnohé algoritmy a myšlenky, které v průběhu času přestaly být atraktivní, se dnes mohou ukázat opět užitečné. Stejně tak mnohé návrhy, které se objevily již dříve, ale jejichž implementace byla limitována architekturou či systémovými prostředky, se mohou uplatnit nyní. Proto chceme nabídnout rekapitulaci významnějších algoritmů, které za posledních 20 let vznikly a měly přímé uplatnění při zobrazování terénu. Není cílem této práce vytvořit studii všech navržených přístupů, ale rádi bychom na příkladech těch podle nás významnějších poukázali na některé důležité prvky návrhu a vlastnosti finálního řešení.

V této kapitole rozebereme způsoby datové reprezentace terénních dat (paragraf 2.1) a pojmy související s jejich zjednodušováním a zobrazováním (paragraf 2.2). Nakonec provedeme průřez používanými technikami a algoritmy (paragraf 2.3).

2.1 Datová reprezentace terénu

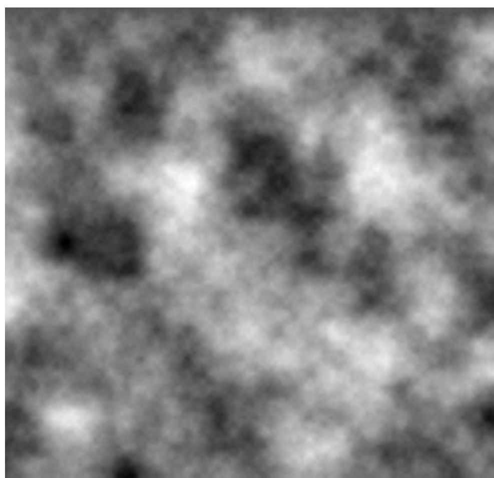
S návrhem algoritmu na zobrazování jakýchkoli dat velmi úzce souvisí volba jejich datové reprezentace, a to v několika úrovních zpracování. Jednu reprezentaci si svým způsobem vynutí GPU, které je stavěno na zobrazování trojúhelníků, i když máme poměrně velkou volnost ve způsobu dodání dat pro tyto trojúhelníky[5]. Jinou reprezentaci můžeme zvolit pro udržování dat za běhu programu, avšak v praxi jsou tyto dvě reprezentace zpravidla totožné, protože rozdílné uložení dat v operační paměti a při zpracování na GPU si vyžaduje časté (obvykle každý snímek) konverze jednoho formátu do druhého. Není to nepoužitelný přístup, ale pro zobrazování terénu je nevhodný, neboť v případě terénu se jedná o velké množství dat, při jehož zpracování hrají významnou roli i jinde zanedbatelné parametry. Konečně jinou reprezentaci si může vynutit uložení dat na zdrojovém médiu, zvláště z důvodu limitujících přenosových rychlostí (zdrojem může být server na Internetu, nebo optické médium). My se zde zaměříme na uložení dat v operační paměti, tedy dat připravených na zobrazení pomocí GPU. Popíšeme si dva základní přístupy:

1. Triangulated irregular network
2. Heightmap

---

[5] Oproti prvním GPU, na kterých byla topologie těsně vázána na geometrii (každý vrchol nějakého trojúhelníku měl přímo definovány souřadnice), nyní můžeme oddělit geometrii a uložit v podobě textury, či dokonce utvářet topologii přímo na grafické kartě.

Triangulated irregular network (dále jen TIN) je obecná síť vrcholů nepravidelně rozmístěných v prostoru a pospojovaných hranami tak, že stěny mezi vrcholy tvoří trojúhelníky, které se navzájem neprotínají. Z matematického hlediska si můžeme TIN představit jako speciální případ implicitní funkce v trojrozměrném prostoru. Tato reprezentace je přirozená pro různé typy spojitých povrchů, tedy i pro terén. Zřídkakdy představuje TIN původní formát dat. Obvykle ji získáme aplikací nějakého předzpracujícího algoritmu na jiný vstupní formát (často výšková mapa), za použití Delaunayovy či jiné triangulace (viz paragraf 2.2).



Obrázek 2.1: Výšková mapa reprezentovaná formou obrázku, ve kterém jsou všechny tři kanály (RGB) nastaveny na stejnou hodnotu.

Druhá velmi rozšířená reprezentace vychází z představy terénu jako povrchu vzniklého vrásněním roviny, tedy povrchu, který jde namapovat na rovinu pouhou rovnoběžnou projekcí. Tato reprezentace bývá označována jako *výšková mapa*[6] a korektně nadefinovat ji můžeme jako diskrétní funkci $y = h(x,z)$, kde funkční hodnoty y mohou nabývat reálných hodnot předem zvoleného rozsahu. Funkce přiřazuje jednotlivým bodům definičního oboru určitou relativní výšku. Takto zapsána nám jistě připomene dvourozměrnou obrazovou funkci definovanou na diskrétní mřížce (rastru), a tak ji zde také budeme chápat. Ostatně, stejně jako obrazová funkce může nabývat funkčních hodnot z vícerozměrného prostoru (základní složky zvoleného barevného modelu), také funkci h bychom mohli přiřadit širší význam tím, že by kromě výšky terénu v daném bodě definovala také fyzikální vlastnosti povrchu, tečný či normálový vektor povrchu a jiné atributy. Mapující funkci bychom už patrně nenazvali výškovou mapou, ale k tomu se vrátíme v dalších částech práce. Abychom podpořili představu výškové mapy jako dvourozměrného obrazu, uvedeme, že častým způsobem jejího uložení je právě podoba jednokanálového obrázku – ten získáme přemapováním intervalu funkčních hodnot funkce h na interval $\langle 0,1 \rangle$ reprezentující rozsah odstínů zvolené barvy. Jednu takovou výškovou mapu ukazuje Obrázek 2.1. Z charakteru dat, která terén definují, je výšková mapa velmi přirozenou reprezentací.

[6] Angl. heightmap pro svůj „mapovací" charakter, případně heightfield.

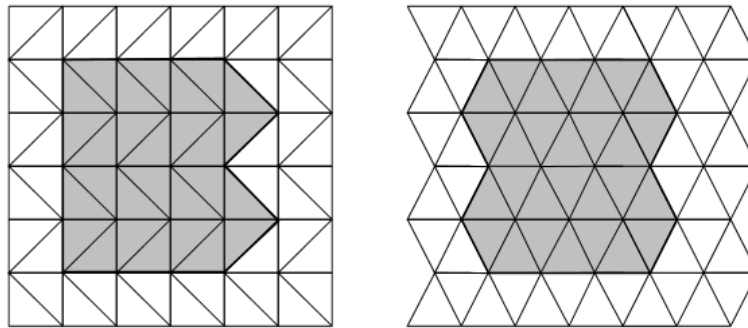
Rozdíl mezi TIN a výškovou mapou je zřejmý – zatímco TIN jsme uvedli jako implicitní funkci, výšková mapa je skutečná funkce definovaná na dvourozměrné doméně, a tedy každému prvku domény odpovídá pouze jedna funkční hodnota. Z toho je patrné velké omezení výškových map – nemůžeme jejich pomocí vyjádřit některé prvky ve skutečné krajině běžné – převis, jeskyně apod. S tímto nedostatkem se vývojáři a designéři zpravidla vypořádávají pomocí objektů „připojených“ k terénu, reprezentovaných obecnější datovou strukturou. Stejně jako hlavní nedostatek výškových map, je snadné uvidět i výhody, které s sebou tato struktura nese. Předně je pravidelná, proto se s ní lépe pracuje – lépe se prochází, ukládá i komprimuje. Dále si uvědomme, že na zadání jednoho bodu v TIN potřebujeme 3 hodnoty (souřadnice v trojrozměrném prostoru), ale na zadání bodu ve výškové mapě pouze jednu, a to její funkční hodnotu (souřadnice y v námi zvolené souřadné soustavě). Hodnoty x a z jsou dány implicitně dotazem.

Dalším významným rozdílem TIN a výškové mapy je přesnost aproximace reprezentovaného terénu, která závisí na geometrii vzorových dat. Rozlišení výškové mapy je konstantní po celé její ploše, zatímco pro TIN je možno dosáhnout adaptivního chování podle geometrie dat. Budeme-li kupříkladu reprezentovat krajinu obsahující vysoká pohoří i velké rovné planiny, budeme se v případě výškové mapy muset rozhodnout pro nějaký kompromis mezi ztrátou informace v oblasti hor kvůli podvzorkování, a ukládání přebytečných dat v ostatních oblastech kvůli nadvzorkování rovných ploch. Paralelu obrazové funkce jsme již použili, takže na ni můžeme navázat, představit si terén jako spojitou funkci a využít termínů a tvrzení z teorie zpracování obrazu nebo signálu obecně. Víme-li, že původní data jsou frekvenčně omezená, a to zpravidla jsou, pokud zrovna nepracujeme s daty generovanými nějakým fraktálovým generátorem, pak ze Shannonova vzorkovacího teorému[7] umíme určit velikost výškové mapy potřebné k bezchybnému zachycení původních dat. Tato velikost však může být zbytečně nadsazená právě proto, že nezohledňuje lokální chování, ale pouze celek. Můžeme si tedy představit různé terény, které půjdou reprezentovat lépe pomocí TIN, jiné pomocí výškových map, obecný závěr však dělat nebudeme. Intenzivnímu srovnání se věnuje například (2), kde je závěr k našemu podivu kladný pro výškové mapy, neboť pro zadanou velikost dostupné paměti aproximují obecný terén přesněji než TIN. Nutno však dodat, že přesnost reprezentace závisí velmi na algoritmu, kterým vznikla při zpracování původních dat.

Doplníme ještě, že existuje i tzv. izometrická verze výškové mapy. Tato alternativa představuje mřížku definovanou na dvou osách svírajících úhel 60 stupňů (viz Obrázek 2.2). Triangulace povrchu pomocí pravoúhlé mřížky silně trpí změnami geometrie prováděnými přes diagonály. Použitím izometrické mřížky dospějeme obecně k vizuálně hladším povrchům. Nevýhodou takovéto mřížky je nutný přepočet souřadnic, neboť textury jsou ukládány jako pravoúhlé mřížky. McGuire a Sibley (3) navrhují posunout liché řádky o polovinu délky hrany. Ve svých měřeních zmiňují o 25 % přesnější stínování než při použití běžné mřížky s diagonální triangulací.

---

[7] Shannonův teorém říká, že bezchybná rekonstrukce spojitého, frekvenčně omezeného signálu je možná tehdy, pokud jej vzorkujeme s alespoň dvojnásobnou frekvencí, než je maximální frekvence vzorkovaného signálu.



Obrázek 2.2: Běžná (vlevo) a izometrická (vpravo) mřížka. Zvýrazněná část reprezentuje v obou případech stejnou oblast.

2.2 Zjednodušování geometrie

Mezi různými typy objektů, kterými se běžně zabýváme při zobrazování (nejen herní) scény, má terén své zvláštní postavení, které je dáno jednak jeho mohutností, jednak jeho tvarem. Mohutností rozumíme počet trojúhelníků, ze kterých se skládá. Pro lepší představu si můžeme uvést srovnání – menší a jednodušší objekt v běžné herní scéně může mít desítky až stovky trojúhelníků. Komplikovanější objekty, například postavy, se mohou pohybovat v řádu tisíců trojúhelníků a výjimečně velké a komplikované objekty se mohou skládat až z desetitisíců. Naproti tomu základní triangulace výškové mapy o velikosti 10 000 × 10 000 vzorků představuje 200 milionů trojúhelníků. Není to číslo, se kterým by si dnešní GPU nemohlo poradit, ale pokud by si s ním mělo poradit 30 krát za vteřinu a ještě při tom ponechat čas na jiné věci, museli bychom počkat pár let na adekvátní hardware (zatím ponecháváme stranou fakt, že takto velká data bychom velmi těžko uchovávali v paměti).

Zmínili jsme také tvar terénu – ten je specifický svým podélným (či rozlehlým) charakterem. Zatímco u jiných objektů se můžeme rozhodovat na bázi celé instance o tom jak, a jestli vůbec, jej zobrazíme, v případě terénu tomu tak není. Zobrazit je potřeba téměř vždy, ale viditelná je z něj pouze malá část. Navíc platí, že čím větší terén je, tím menší část je z něj vidět, a o to více se nám tedy vyplatí strávit nějaký čas volbou toho, co přesně se z něj zobrazí. Navíc i ta malá zobrazovaná část terénu (třeba pouhé procento celkové rozlohy) v sobě může zahrnovat různá místa, jejichž vzdálenost k pozorovateli se vzájemně liší o několik řádů. Díky perspektivní projekci se stejně radikálně liší i velikost projekce takovýchto ploch na obrazovku, a je tedy krajně nevhodné zobrazovat celý terén v jednom rozlišení. Proto, stranou jakékoli datové reprezentace, je důležité zvolit správně rozlišení, v jakém zobrazíme tu kterou část terénu podle jejího významu a vzdálenosti od pozorovatele.

Již před 30 lety vyzdvihl James Clark výhody uchování geometrie modelů použitých v zobrazované scéně v různých rozlišeních(4). Od té doby se stalo toto téma velmi populárním a vznikla již dlouhá řada postupů automatizujících proces získávání takovýchto modelů. Nejčastěji mluvíme o *zjednodušování geometrie* (geometry simplification), ale v určitých případech můžeme mít i model nižšího rozlišení a různými postupy se z něj pokoušet získat kvalitnější variantu. Pro problém vytvoření a užití těchto modelů v různých úrovních detailu se vžilo označení *Level of*

Detail (dále jen LOD) a vznikly celé rodiny algoritmů, které ho řeší určitým způsobem, podle určitých kritérií nebo nad určitým typem dat. Pro celkovou složitost této problematiky a množství již navržených parciálních či komplexních řešení vznikly pokusy zavést do ní taxonomii. V tomto směru velmi zdařilá je práce Garlanda a Heckberta (5), v níž se autoři věnují teoretickému i praktickému porovnání desítek algoritmů a zařazují je do hlavních kategorií podle typů dat, nad jakými daný algoritmus umí pracovat. Přiřazují jim však i celou řadu dalších atributů, například strukturu výstupních dat, použitou metriku chyby (viz odstavec 2.2.4) nebo směr, jakým se výsledná geometrie generuje (viz odstavec 2.2.5).

Velmi obsáhlá a podrobná je kniha (6), kde najdeme o LODu téměř vše, tedy nejenom soupis a porovnání algoritmů, ale také různé dílčí metody používané při zjednodušování geometrie jako je výběr prvku pro odebrání, triangulace díry, měření vzniklé chyby apod. Pro nás je zajímavé, že kniha obsahuje zvláštní kapitoly o použití LOD algoritmů ve hrách a aplikace LOD na terén. Od hlavního z autorů je možné přečíst si též méně obsáhlou verzi (7), která se podobně jako (5) věnuje taxonomii a zachycení důležitých algoritmů. Konečně užitečné rady při volbě správného algoritmu pro zadaný typ projektu je možné nalézt v (8).

Nechceme opakovat práci výše zmíněných děl, zvláště pak nechceme vyvolávat přístupy, které nejsou vhodné pro zobrazování terénu. Z důvodů uvedených v začátku této sekce je však zřejmý význam a spjatost řešení LODu se zobrazováním terénu. Kdykoli se v literatuře mluví o řešení zobrazování terénu, implicitně je tím mimo jiné myšleno řešení LODu a výpočet viditelnosti závislé na poloze a zorném úhlu pozorovatele. Proto považujeme za důležité tu vysvětlit některé pojmy s tím související, abychom snáze nahlédli do vybraných algoritmů, které pak stručně uvedeme v sekci 2.3.

2.2.1 Statický a dynamický LOD

Pod pojmem *statický LOD* rozumíme řešení, při kterém se ještě v produkční fázi vytvoří pro model několik jeho variant v různých úrovních detailu, a tyto varianty se při zobrazení nikterak nemodifikují (odtud název statický). Není pevně dáno, jak tyto varianty vzniknou – mohou být vymodelovány každá zvlášť, nebo mohou být generovány automaticky pomocí některého z algoritmů pro redukci geometrie. Důležité je, že na jejich tvorbu je k dispozici velké množství času během produkce. Za běhu programu, tedy při zobrazování daného modelu, se provádí pouze volba varianty pro zobrazení podle vzdálenosti od pozorovatele a jiných parametrů. Nevýhoda statického LODu je ta, že při jeho užití jsme nuceni „občas“ zobrazit model vyšší kvality, než je nutné, nebo naopak nižší, než je potřeba. Jak často tato situace nastává, závisí především na počtu variant modelu, které jsme připravili. Větší počet variant vede k lepším výsledkům, ale nese s sebou velké nároky na paměť. Doplňme ještě, že statickému LODu se též někdy říká *diskrétní*, neboť při zobrazování dochází k náhlým, nespojitým změnám, kdy je při pohybu pozorovatele zaměněna jedna varianta modelu za jinou.

Zpravidla náročnějším řešením je *dynamický LOD*. Při něm tíha redukce složitosti modelu padá na dobu zobrazování a čas, který redukce vezme, je tedy kritický. Toto řešení má samozřejmě své výhody. Předně máme větší kontrolu nad tím, co právě zobrazujeme. Nejsou to už jen diskrétní

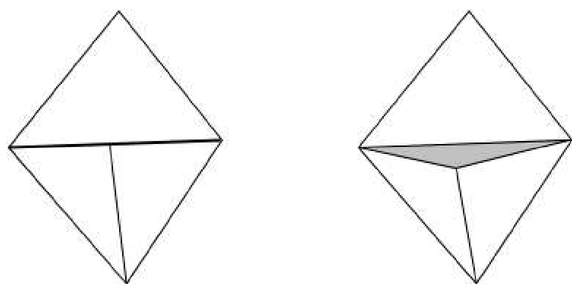
pozice, ve kterých dojde k záměně jednoho modelu za jiný, ale jsou to do určité míry plynulé přechody od jedné varianty ke druhé. Proto se také často tento postup označuje jako *spojitý*[8]. Další výhodou bývají nižší paměťové nároky než v případě statického LODu, ale to už záleží na konkrétním řešení.

Někdy je těžké označit algoritmus za čistě statický nebo dynamický. To platí obzvlášť při řešení LODu pro terén. Dalo by se říci, že čistě statický LOD pro terén ani není možné použít. Avšak při dělení terénu na různé podoblasti, ať už konstantní velikosti (například pravidelná mřížka), nebo designérem vytvořené logické celky, můžeme na každou oblast aplikovat statický LOD samostatně. Při zobrazování pak nebudeme terén řešit jako celek, ale budeme k němu přistupovat po částech jako k různým objektům, z nichž každý může být zpracován jinak. I takovéto algoritmy bývají zpravidla řazeny do kategorie statického LODu, zvláště při řešení terénu, kde základní pojem statického LODu ztrácí význam. My se přikláníme spíše ke zvláštní kategorii *hybridních algoritmů*, protože někdy je jejich komplikované řešení jen těžko zařaditelné do jedné z výše uvedených kategorií (existence vhodné definice nám není známa a zde se o ni pokoušet nebudeme).

2.2.2 Defekty při LOD technikách

Při použití většiny LOD technik může docházet k různým defektům v geometrii, případně jiným vizuálně se projevujícím artefaktům. Četnost jejich vzniku i možnosti jejich odstranění jsou dalšími důležitými atributy při volbě algoritmu. Ukážeme si zde tři základní: T-vrchol, prasklinu a popping[9].

Jako *T-vrchol* se označuje vrchol na hraně, která sousedí s více než dvěma trojúhelníky. V korektní síti patří každá hrana právě dvěma trojúhelníkům, případně pouze jednomu, pokud je okrajová. Tato chyba může nastat nekorektním odstraněním hrany, případně spojením dvou částí sítě, které se decimovaly nezávisle na sobě. Příklad T-vrcholu ukazuje pro ilustraci Obrázek 2.3. Existence T-vrcholu nemusí být vždy na první pohled patrná, geometrie je v daném místě neporušena. Problém způsobuje především omezená přesnost při výpočtech osvětlení, kde se může projevit chybějící normála z jedné strany spoje.



Obrázek 2.3: Vlevo T-vrchol. Zvýrazněna je hrana nesprávně rozdělena vrcholem. Hrana je sdílena 3 trojúhelníky. Vpravo je prasklina vzniklá posunutím T-vrcholu mimo hranu, na níž ležel.

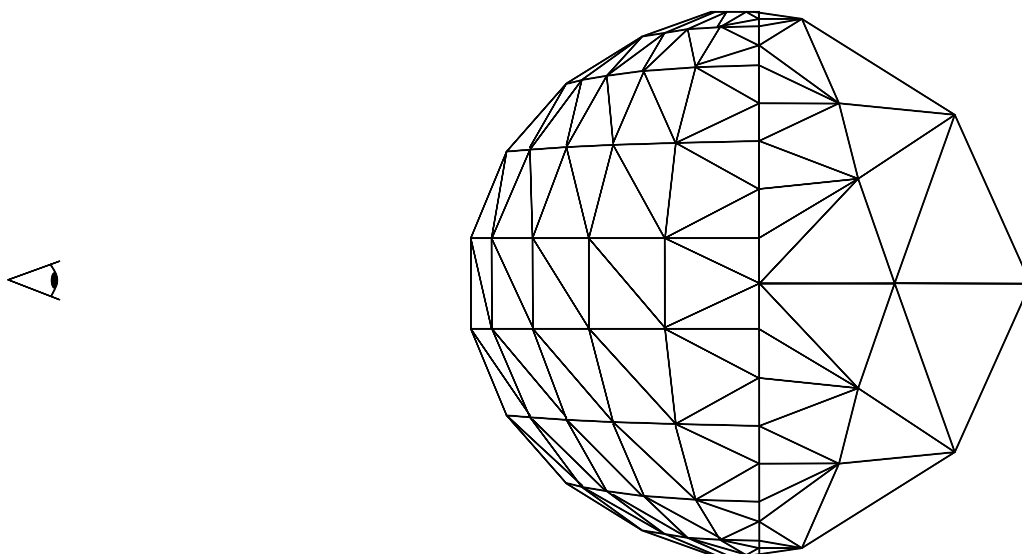
[8] V anglické literatuře „continuous LOD“.

[9] Angl. označení T-junction a crack. Pro efekt zvaný popping neznáme žádný ustálený překlad.

Daleko nepříjemnějším artefaktem je *prasklina*, která vznikne změnou polohy T-vrcholu. Jakmile se vrchol posune mimo hranu, na které ležel, utvoří se nespojitost v geometrii. Přestože vzniklá prasklina bývá velmi malá, může být dobře patrná, pokud se za ní kreslí objekt výrazně odlišné barvy nebo jasu. Příklad je opět na Obrázek 2.3 napravo.

Posledním zmiňovaným artefaktem je *popping*. To je vizuálně velmi nepříjemný jev, který nastává při (náhlé) změně geometrie. Při pohybu pozorovatele nastane okamžik, kdy se přidá nebo odebere vrchol z modelu, protože LOD algoritmus reflektuje změny v poloze kamery. Celý jev je tím rušivější, čím výraznější změna geometrie nastane. Obecně tedy tímto problémem daleko více trpí statický LOD, u kterého se mění vždy větší množství vrcholů najednou.

Účinnou obranou proti poppingu je tzv. *morfování* vrcholů[10]. Jde o způsob potlačení náhlé změny v geometrii tím, že se změna začne provádět o něco dříve a provádí se v určitém smyslu spojitě během časového úseku. Konkrétně se vezme vrchol, který se má v brzké době odstranit, a zvolna se přesouvá na místo, kde splyne s ostatní geometrií a kde jeho odstranění nezpůsobí náhlou změnu. Stejný způsob lze samozřejmě aplikovat také při přidávání vrcholu. Morfování také můžeme snadno popsat interpolací souřadnic vrcholu (interpolujeme mezi dvěma různými polohami). Parametr, který interpolaci řídí, pak může být definován různými způsoby. Tři nejčastěji používané jsou dány číslem snímku, pohybem kamery a reálným časem.



Obrázek 2.4: Aplikace pohledově závislého LODu na model koule. Silueta modelu je dělena drobněji než jiné části, naopak odvrácená strana koule je zobrazena jen velmi hrubě.

2.2.3 Pohledově závislý LOD

O algoritmu řekneme, že je *pohledově závislý*[11], pokud při redukci modelu zohlední polohu či úhel pohledu pozorovatele vůči právě redukované části. Z toho plyne, že statický LOD nemůže být

[10] V angl. literatuře se používá termín morphing nebo geomorphing.
[11] Nejčastěji se používá angl. termín view-dependent LOD, ale někdy se též mluví o anizotropním LODu.

pohledově závislý, neboť redukce modelu se u něj provádí ve fázi předzpracování, kdy nejsou potřebné informace k dispozici. Kromě vzdálenosti od pozorovatele může algoritmus také zohlednit, zda se jedná o přivrácenou či odvrácenou část objektu, nebo zda se jedná o část tvořící siluetu. Výsledek takového postupu ukazuje Obrázek 2.4.

Při zobrazování terénu je aplikace pohledově závislého LODu nevyhnutelností. Různé algoritmy však mohou zohlednit různé faktory při redukci, často se používá pouze vzdálenost k pozorovateli pro její snadné vyčíslení za běhu.

2.2.4 Metrika chyby

Bez ohledu na volbu řešení LODu, dříve nebo později jsme nuceni porovnávat výsledky a nějakým způsobem zhodnotit, který je pro nás lepší. Při použití statického LODu bývá často tato práce ponechána na modeláři, který jednotlivé varianty modelu ručně vytvoří. Ten se může subjektivně rozhodnout podle vizuálního hlediska. Při větším množství modelů je však tento přístup nepraktický a při dynamickém LODu dokonce nemožný. Potřebujeme tedy automatické rozhodovací kritérium, které dokáže zhodnotit kvalitu zvolené konfigurace (tj. aktuální geometrie). Z tohoto důvodu se zavádí *metrika chyby*.

Volba metriky chyby je velice zásadní pro kvalitu výstupu. Používáme ji během redukce geometrie pro řízení celého procesu a špatně zvolená metrika může způsobit nechtěnou redukci geometrie na nesprávných místech. Ideální by bylo najít vždy optimální síť pro zobrazení zadané geometrie pomocí určitého počtu trojúhelníků s minimální chybou. Hledání takové sítě je však NP-úplný problém, a proto se často využívá nějaký hladový algoritmus, který vybírá nejlepší krok podle zvolené metriky. Je zřejmé, že takový algoritmus nemusí dát optimální výsledek, vylepšit se dá kupříkladu hodnocením několika kroků dopředu, ale i to je časově náročné.

Metrika chyby může, ale také nemusí, být pohledově závislá. Pokud není pohledově závislá, můžeme ji využít pro generování variant pro statický LOD, avšak při zobrazování ji musíme doplnit dalším kritériem, které rozhodne, jaká chyba je přípustná z daného pohledu. Pohledově závislou metriku můžeme použít pro výpočet pohledově závislého LODu během zobrazování.

Metrika může řídit nejenom další krok algoritmu pro zjednodušení geometrie, ale také rozhodnout o jeho ukončení. Běžně se tak používá podmínka maximální chyby, která specifikuje, jaká maximální chyba je pro daný objekt povolena. Algoritmus zjednodušuje geometrii tak dlouho, dokud neklesne chyba reprezentace zpracovávaného modelu pod určitou mez. Tento typ zjednodušování nazýváme *error-based* či *error-bound*, protože je vázán hodnotou chyby. Využívá se tam, kde je pro nás důležitá věrnost reprezentace. Použijeme jej kupříkladu při generování statického LODu, jeho úrovně mohou být zrovna tak definovány pomocí chyby.

Ve hrách je naopak důležitější plynulé chování i za cenu dočasného snížení kvality obrazu. Tomuto přístupu je více nakloněno zjednodušování omezené časem – předem určíme, jaký čas máme k dispozici a po uplynutí daného času proces zjednodušování ukončíme bez ohledu na to, jaká je chyba současné reprezentace. Ne vždy je možné takovéto řešení uplatnit, algoritmus k tomu musí být navržen tak, aby se redukce prováděla „rovnoměrně“, tedy aby po překročení

stanoveného limitu nebyla část modelu zredukována výrazně více než jiná, ke které se algoritmus ještě nedostal. Hladové algoritmy nám toto zpravidla zajistí.

Jiný model řízení, který je ve hrách použitelný, se nazývá *budget-based*. Jeho název je odvozen od toho, že na zadaný model a situaci máme určitý „rozpočet“, tedy počet trojúhelníků, které máme k dispozici pro reprezentaci modelu. Proces zjednodušování se ukončí ve chvíli, kdy překročíme zadanou hranici počtu trojúhelníků. Výhodou tohoto přístupu je přímá kontrola nad paměťovými nároky získané reprezentace. Pro zobrazování má také význam, neboť čas strávený zobrazováním modelu je závislý na objemu vstupních dat. Dříve byl tento přístup oblíbený pro použití na dynamický LOD, protože podobně jako error-based je snadné jím kontrolovat zobrazovací frekvenci úpravou rozpočtu trojúhelníků pro daný snímek. S výkonem dnešních GPU však neexistuje tak přesná lineární korelace mezi množstvím zobrazených trojúhelníků a časem potřebným k jejich zobrazení – zvlášť obtížné je najít korelaci mezi časem CPU a GPU, neboť oba procesory pracují do jisté míry asynchronně.

Je nad rámec této práce sestavit ucelenější přehled běžně používaných metrik. Velmi podrobně se tomuto tématu věnuje například (6).

2.2.5 Shora-dolů nebo zdola-nahoru

V celé kapitole používáme pojem zjednodušování geometrie, v některých případech to ale není přesné vyjádření. Zatímco model, ke kterému zvoleným postupem spějeme, je zjednodušenou verzí původního, postup samotný nemusí být prováděn inkrementálním zjednodušováním, ale naopak zesložiťováním. Postup je obvykle založen na opakovaném přidávání nebo odebírání různých entit – vrcholů, hran, skupin vrcholů či ploch. Odebírání se provádí z kopie původního modelu a lze jej v pravém slova smyslu nazvat zjednodušováním, nebo též decimací (angl. decimation). Když si různé konfigurace modelu představíme jako strom, jehož kořenem je jeden vrchol zastupující celý model, vidíme, že v tomto stromě postupujeme *zdola-nahoru*.

S opačným postupem, tedy přidáváním vrcholů, začneme na nějaké množině vrcholů představující minimální aproximaci modelu, a do této množiny budeme postupně přidávat vrcholy z původního modelu, až dosáhneme požadované mety. Postupná obnova (angl. refinement) modelu zde probíhá směrem *shora-dolů*. Velkou výhodou tohoto postupu (oproti směru zdola-nahoru) je časová nezávislost na velikosti vstupu – zpracováváme pouze ty entity, které nakonec vyprodukujeme, a těch bývá řádově méně, než vstupních dat. Nevýhodou může být o něco horší aproximace; při sestupu máme k dispozici méně informací, což může ovlivnit kvalitu výsledku.

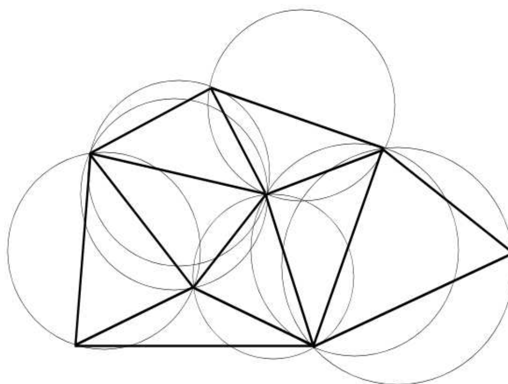
2.3 Známé postupy

Pokusíme se zde uvést a stručně vysvětlit některé významnější postupy, případně jejich modifikace a odnože. Pro lepší přehlednost je zde rozřazujeme do skupin, aniž bychom si dělali nějaké nároky na zavedení opravdové taxonomie. Uvádíme zde i starší algoritmy, které se dnes na zobrazování terénu nepoužívají, jednak pro jistou úroveň úplnosti, jednak proto, že nevylučujeme, že s patřičnou modifikací bude možné takovýto algoritmus použít.

Předtím si však ještě vysvětleme hlavní důvod obtížnější aplikace starších algoritmů. Je to především přílišná zátěž CPU. Při interaktivním zobrazování připadá na každý snímek maximálně 30μs. Z pohledu GPU je to čas, za který je potřeba nakreslit všechny objekty viditelné části scény (resp. všechny objekty, které jsme označili za viditelné, nebo o jejichž viditelnosti jsme nedokázali rozhodnout). Z pohledu CPU je to čas, během kterého se musí provést všechna logika aplikace, která se synchronně nebo asynchronně provádí přibližně se stejnou frekvencí jako zobrazování. V případě počítačové hry to může být například řešení vzájemných kolizí objektů, přehrávání prostorového zvuku, řízení chování umělých bytostí nebo zpracování komunikace hry s hráčem či herním serverem. V neposlední řadě musí zbýt nějaký čas na rozhodnutí, co je a co není vidět, a vše co by mohlo být vidět, musí být předáno GPU ke zpracování. Čím více času CPU strávíme výpočty viditelnosti, tím více času GPU můžeme ušetřit při zobrazování vynecháním neviděných dat. Dokud bylo zobrazování prováděno pomocí CPU, nebo dokud CPU bylo řádově výkonnější než GPU, bylo zpravidla výhodné strávit velké množství času rozhodováním a optimalizací vykreslovaných dat. Při dnešních rychlostech GPU ale řada optimalizací ztrácí význam, neboť šetří čas (GPU), kterého je dostatek, za cenu času (CPU), kterého se nedostává.

Dalším negativem starších algoritmů je tzv. „*small batch problem*" popsaný v (9) nebo (10). Souvisí s tím, že každá utilizace grafické karty stojí nějaký čas, který si můžeme představit jako konstantní cenu za to, že využijeme její službu[12]. Jestliže stokrát využijeme její službu pro zobrazení 1 trojúhelníku, zaplatíme stokrát více, než kdybychom zobrazili všech 100 trojúhelníků najednou. S přibývajícím počtem trojúhelníků zpracovaných najednou se snižuje význam ceny utilizace GPU v porovnání s časem potřebným na jejich zobrazení.

Je pochopitelné, že algoritmy, které patřičným způsobem nezohledňují výkon GPU, budou dnes hůře použitelné. Mnohdy je však možné takovýto algoritmus modifikovat tak, aby i dnes poskytoval zajímavé výsledky.



Obrázek 2.5: Delaunayova triangulace nad 8 vrcholy. Žádný z vrcholů nezasahuje do kruhů opsaných trojúhelníkům, jichž není součástí.

[12] Velmi zjednodušená představa. Tato cena ve skutečnosti není konstantní, ale také není dobře měřitelná a toto zjednodušení je zpravidla postačující.

2.3.1 Obecné TIN

Garland a Heckbert navrhli optimalizaci hladového algoritmu pro zjednodušení vstupní výškové mapy na dvourozměrnou TIN (1). Algoritmus postupuje směrem shora-dolů. Do triviální triangulace základny terénu postupně přidává vrcholy s největší chybou vzhledem k aktuální aproximaci. Síť vrcholů je vždy aktualizována pomocí *Delaunayovy triangulace*, což je triangulace, v níž žádný vrchol neleží uvnitř kružnice opsané trojúhelníku, jež není tvořen daným vrcholem. Nalézt ji můžeme například stejným způsobem, jako tvoříme Voronoiův diagram nad danými vrcholy – ten tvoří duální graf k Delaunayově triangulaci (11). Má velmi dobrou vlastnost, že do jisté míry zabraňuje tvorbě příliš úzkých trojúhelníků (viz Obrázek 2.5). Proto bývá hojně využívána k triangulaci sítě definované nad dvourozměrnou základnou. Nevýhoda Delaunayovy triangulace spočívá v jejím dvourozměrném charakteru – zanedbává y -ovou souřadnici, takže pracuje pouze s projekcemi trojúhelníků do roviny xz . Vytvořené „silné" projekce trojúhelníků tak mohou ve skutečnosti tvořit úzké trojúhelníky v trojrozměrném prostoru.

Garland a Heckbert ve své implementaci využívají lokality změn prováděných v jednotlivých krocích, aby se vyhnuli přepočtu chyby všech vrcholů v každém kroku. Dosahují dobré výsledné časové složitosti $O((m+n)\log m)$ pro počet vzorků vstupní výškové mapy n a počet výstupních vrcholů m . Nabízí také náhradu Delaunayovy triangulace *triangulací řízenou daty* (tzv. data-driven triangulace), s níž, jak uvádí, dosahují lepších aproximací. Algoritmus není navržený pro řešení dynamického LODu, ale je velmi efektivní pro generování TIN reprezentací vstupní výškové mapy pro použití v rámci statického nebo hybridního LOD systému.

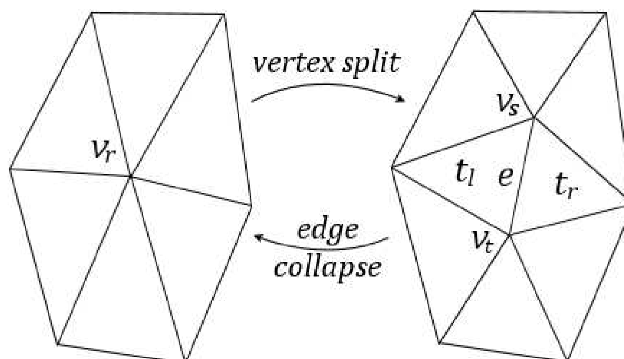
Suglobov rozšířil Garlandův a Heckbertův hladový algoritmus o podporu atributu spojeného se vstupními daty (12). Metrika chyby (přesná definice není zmíněna) zohledňuje diferenci v geometrii i v přidaném atributu, který je využit pro popis barvy terénu. Nepoužívá se jediná textura na pokrytí celého povrchu, ale atribut slouží jako index do pole materiálů, které jsou spojeny s menšími texturami. Výsledná TIN struktura má horší aproximaci vstupních dat, ale pro terén s texturou podává vizuálně lepší výsledky.

2.3.2 Progressive meshes

Progressive mesh je datová struktura navržená Hoppem (13), která společně s hrubou aproximací nějakého modelu obsahuje také informace, podle kterých je možné přesně rekonstruovat původní model. Informaci na rekonstrukci modelu představuje operace *vertex split*, která nahradí jeden vrchol sítě dvěma vrcholy za současného přidání jedné hrany tyto dva vrcholy spojující a dvou stěn náležících nové hraně (viz Obrázek 2.6). K operaci vertex split můžeme nadefinovat také inverzní operaci *edge collapse*, která sloučí dva vrcholy do jednoho a odstraní hranu, která je spojovala, a trojúhelníky připadající odebrané hraně. Progressive mesh pak můžeme přímo nadefinovat jako uspořádanou dvojici (M^0, S), kde $S = (s_0, \ldots, s_{n-1})$ je uspořádaná n -tice operací vertex split a M^0 je trojúhelníková síť představující aproximaci nějakého zjednodušeného modelu. Obecně můžeme takovouto síť nadefinovat uspořádanou trojicí

$$M = (V = (v_0, \dots, v_m), E = \{(v_i, v_j); v_i, v_j \in V\}, T = \{(v_i, v_j, v_k); v_i, v_j, v_k \in V\}),$$

která je velmi podobná definici grafu omezeného na trojúhelníkové stěny, ale rozšířená o „díry“. Struktury V, E a T v tomto pořadí tedy označují vrcholy, hrany a stěny (trojúhelníky) sítě.



Obrázek 2.6: Operace vertex split nahradí jeden vrchol v_r dvěma vrcholy v_s a v_t za současného přidání hrany e a trojúhelníků t_l a t_r. Inverzní operace edge collapse odstraní hranu e a spojí vrcholy v_s a v_t do jednoho vrcholu v_r, za současného odebrání trojúhelníků t_l a t_r.

Označme si M^n původní model. Jeho rekonstrukce se provede postupnou aplikací operací vertex split (značeno $vsplit$) na síť M^0 podle pořadí v S :

$$M^n = vsplit(s_{n-1}, vsplit(s_{n-2}, vsplit(\dots vsplit(s_0, M^0)))).$$

Zůstává otázkou, jak progressive mesh pro zvolený model získat. Hoppe s kolektivem autorů navrhli obecný postup, ve kterém chápou redukci geometrie jako problém minimalizace energetické funkce, jež ohodnocuje redukovanou trojúhelníkovou síť M^r ve vztahu k původní síti M^n (14). Funkce je definována součtem tří výrazů:

$$e(M^r, M^n) = \sum_{v \in V^n} d^2(v, M^r) + c_{vertex}|V^r| + \sum_{u,v \in V^r} c_{edge} \|u - v\|^2 .$$

V^r, resp. V^n označuje množinu vrcholů redukovaného modelu M^r, resp. M^n. Funkce d vyhodnocuje vzdálenost bodu od zadaného modelu a konečně c_{vertex} a c_{edge} jsou konstanty, které slouží k řízení výstupu algoritmu. Po krátkém náhledu do výrazu je patrné, že energetická funkce znevýhodňuje tu síť, která hůře aproximuje originální model (velké vzdálenosti původních vrcholů od aktuální sítě), má velký počet vrcholů nebo má příliš dlouhé hrany. První dvě hlediska jsou logická – hledáme reprezentaci, která bude co nejmenší a co nejvěrnější. (14) však uvádí, že bez ohodnocení hran neexistuje lokální minimum funkce a algoritmus nedává dobré výsledky. Doplněním definice funkce o ohodnocení hrany pomocí její délky je možné nalézt lokální minimum (globální minimum algoritmus nehledá, protože je to příliš časově

náročné). Především konstantou c_{vertex} je možné ovlivnit, zda algoritmus více preferuje přesnost reprezentace, nebo redukci dat.

Energetická funkce v tomto případě není nic jiného než metrika chyby. Algoritmus samotný je pak založený na provádění částečně náhodných operací vertex split, edge collapse a edge swap (záměna diagonály ve čtyřúhelníku), ze kterých ponechává ty, které nejlépe vyhovují zvoleným měřítkům, tj. minimalizují energetickou funkci.

Hoppe později přináší pozměněný algoritmus (13), který provádí pouze operaci edge collapse. Zdůvodňuje to daleko rychlejší konvergencí bez ztráty kvality výsledné sítě. Navíc rozšiřuje metriku chyby o zavedení atributů spojených se stěnami (materiál) nebo vrcholy (texturové souřadnice, normály) sítě. Původní energetická funkce pracuje pouze s geometrií, což je ve většině případů nedostatečné. Navíc je dříve navržený algoritmus pohledově nezávislý. Hoppe to řeší pomocí zpětné vazby do aplikace, která má sama rozhodnout, zda se má daná operace vertex split provést, nebo ne. Operace se navíc provede pouze tehdy, pokud je pro aktuální síť platná, což závisí na tom, zda byly aplikovány vertex split operace, na nichž je tato závislá. I tento přístup má však slabinu v podobě velké časové náročnosti při adaptivním označování operací, které se mají provést (nulová koherence mezi snímky).

Skutečné pohledově závislé řešení pro progressive mesh navrhli Xia a Varshney (15) a nezávisle na nich Hoppe (16). Xia a Varshney navrhli použití *merge tree* struktury, což je binární strom, do kterého uložili hierarchii vrcholů propojených pomocí edge collapse operací (resp. vertex split operací, pokud procházíme stromem od kořene k listům). Průchod stromem obnáší provádění operací nad vrcholy a zjednodušování nebo rekonstruování geometrie podle směru průchodu. Při průchodu je potřeba dodržovat restrikci na korektnost geometrie – operaci je možné provést pouze tehdy, pokud aktuální okolí daného vrcholu souhlasí s okolím definovaným v merge tree. To zabrání vzniku prasklin mezi nekonzistentně rekonstruovanými oblastmi. Hoppeho řešení je velmi obdobné, pouze rozšířil definice vertex split a edge collapse operací o závislost na okolních trojúhelnících, a zbavil se tak restrikcí uvnitř merge tree. Oba postupy definují novou metriku chyby založenou na parametrech pohledu (náležení do zobrazovaného prostoru, orientace povrchu, chyba projekce do prostoru obrazovky). Podobné řešení nabízí také (17), avšak to je generalizováno na obecnější objekty, nejenom manifoldy[13].

Ve své další práci (18) Hoppe navrhl úpravu algoritmu pro zobrazování terénu. Obohatil zobrazování o morfování geometrie, aby potlačil popping artefakt. Operaci vertex split neprovádí okamžitě, ale nově vzniklým vrcholům dává stejné souřadnice, které jsou během několika zobrazených snímků interpolovány do správných poloh. Použití parametru čísla snímku pro řízení morfování je zpravidla nevhodné, protože není snadné zajistit konstantní zobrazovací frekvenci, a na rychlém stroji se tedy může opět projevit popping. Aby byl algoritmus použitelný na velká data, je navíc vygenerována hierarchická struktura jednotlivých sítí pro rozdělení terénu na menší oblasti, jejichž plynulé napojení vyřešil zafixováním vrcholů na okrajích.

---

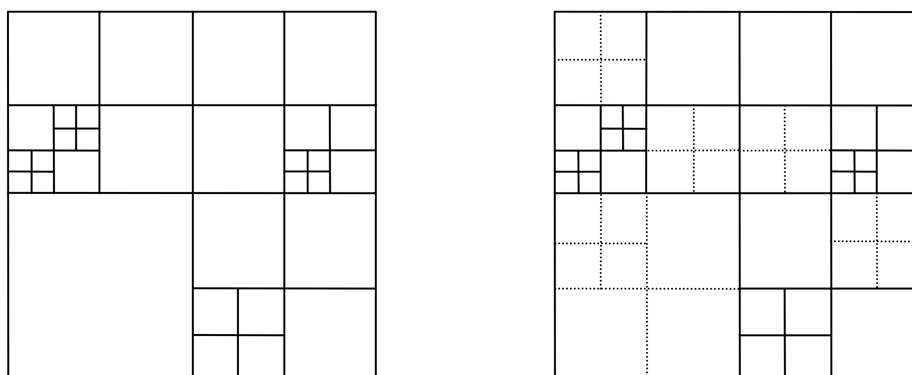
[13] (n-)manifold je objekt (prostor), který je lokálně podobný n-rozměrnému Euklidovskému prostoru. V matematice se nazývá též varieta.

Progressive mesh je dnes pro zobrazování terénu jen velmi špatně použitelný, protože složité hierarchické struktury pro operace nad vrcholy není možné efektivně uložit a zpracovávat na GPU. Pro CPU tak progressive mesh představuje nevítanou zátěž související i s nutnou rekonstrukcí dat, která se předávají grafické kartě. Pro jiné účely má však progressive mesh ještě svůj význam. Například postupná rekonstrukce modelu je dobře využitelná pro zobrazování dat získávaných přes silně limitovaný datový kanál. Návrh implementace je možné nalézt v (19), ačkoli zdrojový kód není dostupný, neboť se stal součástí rozšíření Direct3D.

2.3.3 Kvadrantový strom

Kvadrantový či jiný strom se zpravidla používá pro hierarchické rozdělení zobrazované scény, ve které potřebujeme často eliminovat velké množství objektů, jež prostorově splňují nějakou podmínku (například nejsou vidět). Je přirozené využít již existující struktury a aplikovat ji v modifikované podobě i na terén, zvláště pokud je reprezentovaný výškovou mapou – přirozeně pravidelnou strukturou. Vznikla proto celá řada algoritmů pro zobrazování terénu, které v nějaké podobě využívají kvadrantový strom (ale také například binární strom, jak si ukážeme v odstavci 2.3.4).

Povaha kvadrantového stromu jej předurčuje k horším výsledkům, než jakých jsme schopni dosáhnout méně svázanými strukturami, jako je například progressive mesh, u kterého se při zjednodušování bere v úvahu celý objekt. Pravidelné struktury se však snáze implementují a výrazně lépe se v nich přistupuje ke konkrétním datům. Generování stromu je velice snadné – dělíme plochu terénu rekurzivně na čtyři stejné části tak dlouho, dokud nesplňuje zpracovávaná oblast podmínku danou zvolenou metrikou. Příklad dělení takovým stromem ukazuje Obrázek 2.7.

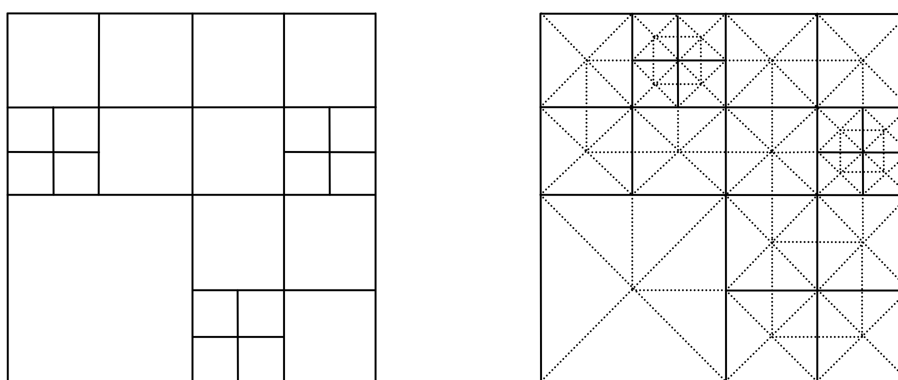


Obrázek 2.7: Obecný kvadrantový strom (vlevo) a omezený kvadrantový strom vytvořený nad stejnými daty (vpravo)

Pro kvadrantový strom je zásadní především způsob triangulace výsledného útvaru. Algoritmem pro dělení totiž získáme pouze informace o tom, jak věrně je třeba kterou část terénu reprezentovat. Zde se otevírá obecný problémem s napojováním různých částí modelu, které byly triangulovány s jiným nastavením. Je to ideální příležitost pro vznik prasklin a T-spojů, kterým se chceme vyhnout. Jednou možností je tzv. *omezený kvadrantový strom* (restricted quadtree), což je varianta klasického kvadrantového stromu vázaná podmínkou na dělení jednotlivých částí. Tato

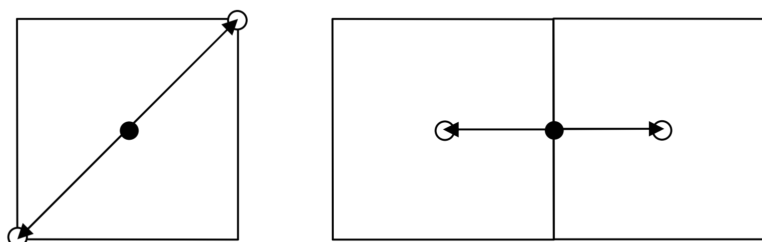
podmínka stanovuje maximální rozdíl v úrovních dělení dvou sousedních oblastí na 1. Tedy jestliže jednu oblast rekurzivně rozdělíme třikrát, žádného jejího souseda nesmíme rozdělit více než 4x nebo méně než 2x (viz Obrázek 2.7).

I pro omezený kvadrantový strom je nutné řešit triangulaci oblasti s ohledem na okolní oblasti, avšak situace je daleko snazší. Rozdělíme čtverec pomocí diagonál na čtyři pravoúhlé trojúhelníky a každý z nich rozdělíme ještě jednou za podmínky, že sousední oblast je dělená stejně nebo více. Příklad triangulace ukazuje Obrázek 2.8. Zde se nám už ukazuje zásadní problém čistě dynamického LODu nad kvadrantovým stromem, a to je sestavení dat pro GPU. Řešení, které se intuitivně nabízí při rekurzivním procházení stromu, tedy zobrazit právě zpracovávaný čtverec v podobě například vějíře trojúhelníků, je dnes neakceptovatelné. Muselo by nevyhnutelně vést k „small batch problem" zmiňovanému dříve.



Obrázek 2.8: Triangulace omezeného kvadrantového stromu.

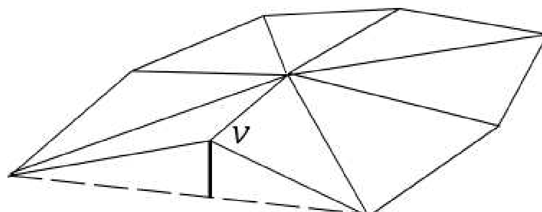
Jeden z významnějších algoritmů využívajících omezený kvadrantový strom navrhli Lindstrom a další (20); často ho v literatuře najdeme pod označením *CLOD*. Procházení stromu v něm probíhá směrem zdola-nahoru a během něj dochází ke spojování sousedních trojúhelníků z jednoho čtverce. Přínos spočíval v zavedení závislostí mezi vrcholy stromu, jejichž použití zabraňuje vzniku prasklin v geometrii. Závislosti mezi vrcholy utváří binární strom, ve kterém je snadné rekurzivním způsobem zjistit, zda je daný vrchol možné přidat do sítě, resp. jaké jiné vrcholy jsou k tomu potřeba (viz Obrázek 2.9).



Obrázek 2.9: Středový vrchol čtverce je závislý na koncových vrcholech diagonály. Vrcholy na hraně čtverce jsou závislé na středových vrcholech okolních čtverců.

Jako metriku chyby zvolili autoři (20) velikost projekce vzdálenosti diskutovaného vrcholu od jeho umístění v úrovni o jeden stupeň méně detailní. Tato jde získat interpolací sousedních

vrcholů, jak ukazuje Obrázek 2.10. Metrika není zcela ideální, protože zohledňuje pouze vztah vrcholu k sousední hrubší úrovni, namísto originálních dat. Nelze ji tedy použít pro garanci maximální chyby. Jiné algoritmy naopak využívají faktu, že při použití ryze monotónní metriky není třeba definovat závislosti vrcholů a při běžném průchodu stromu shora-dolů se zachová korektní struktura omezeného kvadrantového stromu (21), (22), (23).



Obrázek 2.10: Tučně je zobrazena vzdálenost, jejíž projekce na obrazovku vyjadřuje chybu vrcholu v .

Pro nás je dnes zajímavé, že v (20) byl navržen způsob, jak vytvořit z triangulace aktuální sítě jediný generalizovaný řetězec trojúhelníků, tj. řetězec využívající i degenerovaných trojúhelníků[14]. Avšak provedení nepočítá s koherencí mezi snímky, takže řetězec se musí vytvořit znovu při každé změně aktuální reprezentace (ačkoli strom samotný je aktualizován z předchozího snímku). Věříme, že by bylo možné navrhnout postup takový, který by alespoň na bázi desítek snímků ponechal jeden buffer s předgenerovaným řetězcem, ale z celkového hlediska je algoritmus stále špatně použitelný kvůli velké zátěži CPU (postup zdola-nahoru je závislý na velikosti vstupu) a obtížnému převodu na GPU.

Omezený kvadrantový strom je využíván také postupem popsaným v (23), který podobně jako (20) vytváří strom závislostí vrcholů. Na vytvoření řetězce trojúhelníků se zde nahlíží jako na problém nalezení hamiltonovské cesty v duálním grafu ke grafu aktuální triangulace. Problém je zde lépe popsán a je navržena konkrétní implementace. Kromě jiného zde najdeme řešení zobrazování dat, která se nevejdou do paměti, a na rozdíl od (20) je zde také využita koherence mezi snímky, takže struktura se přestavuje pouze tehdy, když změny přerostou zvolenou mez.

Na některé z nedostatků (20) poukazuje také (24). Autoři navrhli algoritmus, který prochází omezený strom shora-dolů, takže není časově přímo závislý na velikosti vstupní sítě, ale na velikosti sítě výstupní, tj. redukované. Tím dosahují daleko vyšších zobrazovacích frekvencí. Nevýhoda tohoto postupu je zřejmě pouze ta, že složitější ohodnocující kriteria, která ke svému vyčíslení potřebují znát originální geometrii, nejsou aplikovatelná (například zachování siluet). V (24) je také prováděno morfování založené na vzdálenosti od pozorovatele, takže veškeré přechody jsou plynulé. Zásadní nedostatek má algoritmus pouze v předávání dat na GPU, protože každý list stromu zobrazuje jako jeden vějíř trojúhelníků, což je velmi pomalá a dnes již dokonce nepodporovaná metoda (25).

---

[14] Degenerovaným trojúhelníkem nazýváme trojúhelník, jehož některé vrcholy jsou shodné. Degenerovaných trojúhelníků se často využívá například pro vytvoření náhlé změny při indexaci řetězce, neboť je dnešní GPU automaticky vyřazují ze zpracování a jejich užití je tedy časově velmi levné.

Odlišným způsobem než předchozí algoritmy využívá kvadrantový strom algoritmus Q-morph (26). Ten pracuje ve třech krocích. V prvním kroku se terén rozdělí do většího počtu stromových struktur a každá se rekurzivně rozdělí podle vzdálenosti k pozorovateli. V krajním případě se dělí až na plochu zahrnující pouze několik vzorků výškové mapy. Listy těchto struktur se označí, zda jsou právě viditelné, nebo mimo záběr. V druhém kroku se pro viditelné listy jednotlivých stromů spočítá hodnota LOD vyjádřená pomocí desetinného čísla. V posledním kroku se vyplní viditelná část pomocné výškové mapy hodnotami výšek terénu v odpovídajících lokacích. Tento krok se provádí tak, že se zpracovávají listy stromu a každý se zapíše pouze do určitých polí pomocné výškové mapy s rozestupem, který odpovídá celočíselné části LODu přiřazeného danému listu. Desetinná část hodnoty LODu se zde používá pro morfování mezi dvěma sousedními LODy, které je tak plynulé, závislé na vzdálenosti, nikoli na čase. Algoritmus má zajímavé řešení LODu, ale je málo efektivní kvůli nutnosti naplnit pomocnou výškovou mapu aktuálními daty každý snímek znovu.

Ulrich použil CLOD ve své implementaci zobrazování terénu, ale na rozdíl od (20) a jiných prací navrhl uložit do kvadrantového stromu i zobrazovaná data (není tedy použita výšková mapa) (27). Tento přístup má dvě velké výhody. Jednak je možné za běhu modifikovat nejen hodnoty dat, ale i jejich rozložení a rozlišení. Druhá výhoda pak spočívá v možném použití oblastí zájmu, které lze aplikovat například na definování detailnějšího terénu v důležitých oblastech. K implementaci je také možné získat zdrojový kód.

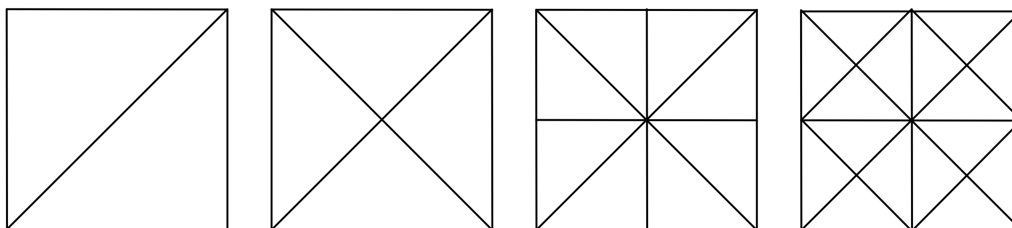
Inovativní přístup ke zpracování kvadrantového stromu je popsán v (28). Jeho zvláštností je práce se stromem na GPU. Model je nejdříve obalen velmi hrubě dělenou strukturou složenou ze stejných krychlí. Na ty je namapována geometrie (a jiné atributy) původního modelu. Jednotlivé stěny krychlí jsou uloženy do jediné velké textury (nazvaná LOD atlas) jako listy stromu. Společně s nimi jsou do LOD atlasu uloženy všechny ostatní uzly stromu a celá tato textura je najednou zpracována na GPU, kde se pro každý uzel rozhoduje, zda má či nemá být dělen. Výsledek je použit v druhém průchodu, během kterého proběhne ořezání i nanesení geometrie na připravené vrcholy pomocí čtení dat z LOD atlasu. Algoritmus je implementovaný ve starším Shader Modelu (konkrétně 2.0) a dá se očekávat, že novější implementace by dosáhla daleko lepších výsledků. Přesto není aplikovaný na terén a zbývá tedy otázka, zda by k tomuto účelu byl použitelný. Z jistého hlediska by byla aplikace na terén snazší – mapování na kvadrantový strom za použití výškové mapy je přímočaré, ovšem algoritmus má poměrně velké paměťové nároky, které by nejspíše příliš limitovaly možnou velikost terénu. Přesto by si podle nás zasloužila tato metoda hlubší prozkoumání.

Jak již bylo zmíněno, kvadrantový strom představuje velmi přirozenou volbu při řešení zobrazování terénu. Průřez různými (ale staršími) pohledy a podrobnější rozbor některých z nich je možné nalézt kupříkladu v (29).

2.3.4 Binary triangle tree

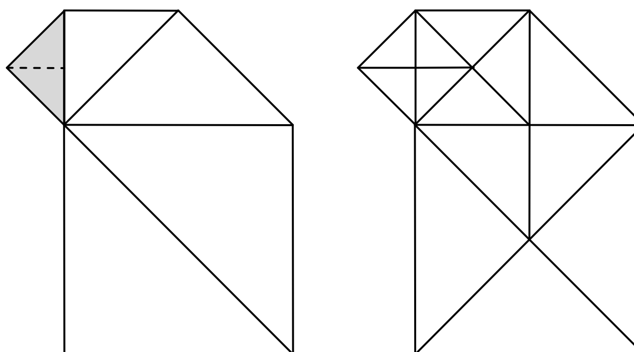
Binární trojúhelníkový strom (binary triangle tree, dále jen BTT) je hierarchická struktura, v níž je klasický binární strom použit k reprezentaci povrchu pomocí rekurzivně dělených trojúhelníků. Ty bývají obvykle pravoúhlé, ale existují výjimky. Algoritmy zde uvedené jsou velmi podobné algoritmům v odstavci **Chyba! Nenalezen zdroj odkazů.**, ale využívají jiným způsobem dělené

hierarchické struktury. Výhodou BTT oproti kvadrantovému stromu by mohla být přirozená triangulace daná přímo listy stromu, avšak algoritmy povětšinou trpí stejným nedostatkem, a to velkými nároky na CPU způsobenými mimo jiné i nutnou restavbou vertex či index bufferu[15] každý snímek.



Obrázek 2.11: Dělení čtvercové plochy pomocí dvou BTT tvořených pravoúhlými trojúhelníky. Každá další úroveň vznikne dělením předchozí pomocí hrany spojující vrchol při pravém úhlu se středem protější strany. Zobrazeny první 4 úrovně dělení.

BTT postavené nad TIN využívá například algoritmus popsaný v (30). Ve fázi předzpracování buduje strukturu nazvanou RTIN, která popisuje LOD hierarchii pro celý terén včetně definice chyby pro všechny uzly. Metrika chyby je postavena na maximu ze vzdáleností původních vrcholů od aproximující plochy daného LODu. Za běhu se rekurzivně prochází strom shora-dolů a utváří se aktuální konfigurace podle parametrů pohledu. Algoritmus postupuje velmi podobně jako (20), ale protože využívá binárního stromu namísto kvadrantového, definuje jinak závislosti vrcholů a restrikce na dělení a spojování oblastí, tj. trojúhelníků. Celou strukturu tvoří dva BTT, jejichž kořenové uzly reprezentují čtvercový terén rozdělený diagonálou. Každá další úroveň stromu vznikne rozdělením trojúhelníku hranou spojující vrchol při pravém úhlu se středem protilehlé strany. Postup lze nejlépe osvětlit obrázkem (viz Obrázek 2.11).



Obrázek 2.12: Operace nuceného dělení BTT. Vlevo je zvýrazněn trojúhelník, který je potřeba rozdělit. Vpravo vidíme výslednou strukturu po provedení nuceného dělení.

(30) zavádí restrikci mezi sousedními trojúhelníky podobnu té, kterou definuje omezený kvadrantový strom, tedy rozdíl úrovní sousedních trojúhelníků se smí lišit maximálně o 1. To zajistí spojité napojení sousedních oblastí bez prasklin. Namísto stromu závislostí popsaného v

[15] Buffery použité k definování geometrie pro GPU.

(20), se zde užívá operací „nuceného dělení" a „nuceného spojení"[16], ale princip zůstává stejný, tj. při potřebě rozdělit nějakou oblast se dělení provede přímo, pokud je to v souladu s definovanou restrikcí, nebo se vynutí dělení sousední oblasti, které může mít za následek další rekurzivní dělení jiných oblastí, dokud není pro celý strom opět splněna restrikční podmínka (viz Obrázek 2.12). Obdobně se řeší spojování. Není těžké nahlédnout, že tento proces je rekurzivně aplikován nejvýše -krát, kde l označuje hloubku BTT stromu.

Uvedený algoritmus pro RTIN strukturu také využívá implicitního definování vyváženého binárního stromu. Namísto budování skutečné struktury je možné využít pole, v němž se pro výpočet souseda nějakého uzlu využije index daného uzlu. K tomu je třeba použít konzistentní označení jednotlivých uzlů; (30) navrhuje sestavení bitového pole, kde n -tý bit udává směr průchodu stromem na n -té hladině – 0 pro levý trojúhelník, 1 pro pravý trojúhelník. Hledání aktuálního souseda, který může být obecně na jiné hladině, je pak založeno na přímém přístupu k sousedovi stejné hladiny a odtud pak k adekvátnímu uzlu podle aktuální triangulace. Přístup k sousedům v konstantním čase je významnou výhodou použití BTT, navíc díky předpočítaným hodnotám chyby pro jednotlivé uzly je možné užít na algoritmus jak budget-based, tak i error-based řízení. Autoři uvádí budget-based pro lepší kontrolu nad plynulostí zobrazování bez konkrétních detailů implementace.

Nezávisle na (30) byl navržen velmi obdobný algoritmus pojmenovaný *ROAM* (22). Jeho autoři používají jiná označení pro navržené struktury a postupy, ale princip zůstává stejný. V práci je však rozebráno více implementačních detailů a je přesněji definována metrika chyby.

Jedna z hlavních předností ROAM algoritmu je vysoká koherence mezi snímky závislá na rychlosti změn pohledu pozorovatele. Důležitý je fakt, že z triangulace použité při posledním snímku se dá k té následující dostat posloupností spojení a rozdělení trojúhelníků. Algoritmus využívá dvou samostatných front pro řazení trojúhelníků pro spojení a rozdělení podle jejich priority, resp. podle chyby, kterou dané trojúhelníky právě nesou. Hladový algoritmus, který opakovaně vybírá prvky z fronty, zaručuje, že kdykoli je ukončen, vygenerovaná síť je pro danou strukturu a metriku chyby nejlepší možná při dosaženém počtu trojúhelníků. Proto je také snadné využít budget-based nebo time-based řízení, a tím ovládat aktuální rychlost zobrazování. Nejnáročnější operací zůstává aktualizace front. Z toho důvodu se přepočet všech hodnot celého stromu neprovádí každý snímek, ale je rozdělen vždy do více snímků.

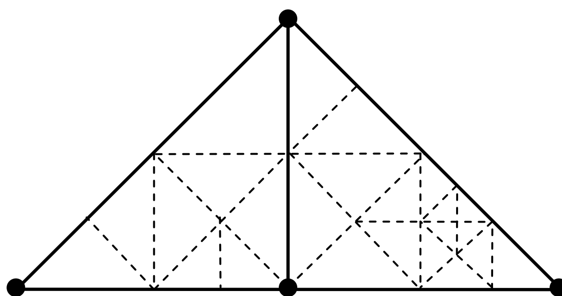
Za nedostatek ROAM algoritmu se dá považovat špatná koherence mezi snímky při otáčení pozorovatele. Zatímco při pohybu je má algoritmus velmi dobré výsledky, při otáčení se musí přetvořit celý BTT. V takové chvíli je dokonce výhodnější použít klasický přístup shora-dolů od počátku, místo využití dat předchozího snímku. Navíc je ROAM ve své plné šíři implementačně náročný. Přesto se algoritmus pro své velmi dobré výsledky stal na nějaký čas vyhledávaným tématem dalšího výzkumu. Zdrojový kód a dobré implementační poznámky najdeme například v (31). Z dnešního hlediska je stejně jako většina dynamicky generovaných triangulací velmi nepraktický pro nadbytečnou zátěž CPU.

---

[16] Z angl. forced split a forced merge.

V (32) je ROAM kritizován pro špatnou volbu metriky a je navržen zcela odlišný přístup. Oproti jednorozměrné informaci používané v ROAM, která udává vzdálenost aproximace vůči původním datům, zde je hledána izoplocha, která přesněji označí oblast, v níž je daná aproximace platná (splňuje určitou podmínku na přesnost). Touto izoplochou je pro jednoduchost implementace zvolen plášť koule. Vždy, když se pozorovatel dostane do prostoru vymezeného nějakou koulí, nahradí je asociovaná oblast přidána do seznamu pro rozdělení. Naopak, když pozorovatel kouli opustí, je oblast přidána do seznamu pro sloučení. Zbytek algoritmu probíhá stejně – hladový algoritmus za pomoci dvou front vybírá oblasti pro sloučení a rozdělení a při přechodu na další snímek využívá struktury zhotovené v předchozím snímku. Pro urychlení testů se využívá monotónnosti této metriky, a testují se tedy pouze aktivní obalové koule. Dalšího urychlení se dosahuje dočasnou agregací různých skupin koulí podle polohy pozorovatele a jejich vzájemných vzdáleností. Detailnější implementaci práce nezmiňuje, věnuje se spíše texturování terénních dat. Také nenabízí žádná číselná srovnání, pouze zmiňuje výrazně lepší výsledky než ROAM, zvláště při vyšších rychlostech pohybu a velikosti terénu.

Princip algoritmu ROAM je využit také v (33), ve kterém jsou části BTT nahrazeny předem generovanými bloky terénu namísto jednotlivých trojúhelníků. Algoritmus nazvaný RUSTiC ve fázi předzpracování pro data terénu vytvoří trojúhelníkové bloky reprezentující možná zakončení BTT. Každý takový blok je sestaven až z 16 trojúhelníků a je dělen určitým unikátním způsobem, aby se dal spojitě napojit na sousední bloky. ROAM algoritmus je použit na rychlé řešení oblastí, které je potřeba rozdělit či spojit a skupiny trojúhelníků jsou pak ve fázi zobrazování nahrazeny jedním blokem, jak ukazuje Obrázek 2.13.

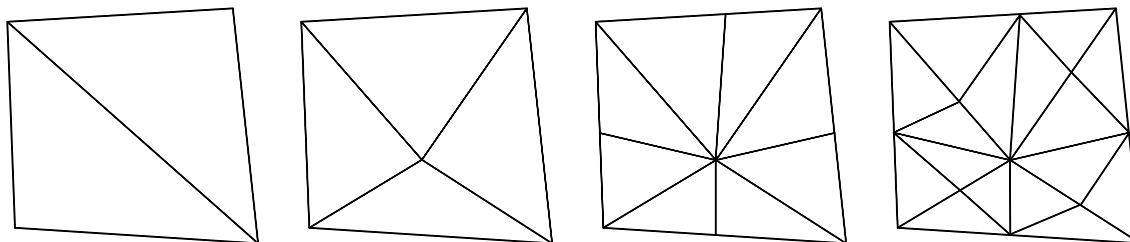


Obrázek 2.13: Napojení dvou sousedních bloků, které musí být stejně děleny na společné hraně.

Efektivnější agregování trojúhelníků do větších celků, než jaké používá RUSTiC algoritmus, navrhuje Levenberg v (34). Zatímco RUSTiC využívá předzpracování terénu pro vygenerování použitých bloků, které vystupují jako jakési šablony, Levenberg navrhuje vytvářet tyto bloky až za běhu. Důležité je, že vygenerovaný blok se uchovává v paměti tak dlouho, dokud nemá dojít k jeho rozdělení či sloučení. Tento přístup má výrazně menší nároky na paměť – uchovávají se v každý čas jen aktuálně používané agregované trojúhelníky – a ne potřebuje fázi předzpracování. O velikostech agregovaných trojúhelníků se může rozhodnout kdykoli za běhu podle poměru výkonu CPU a GPU (větší agregované trojúhelníky znamenají více nadbytečné geometrie, ale menší dobu zpracování ROAM algoritmu).

Zvláštní adaptace ROAM algoritmu byla navržena v (35). Autoři použili algoritmus při zobrazování modifikovatelného terénu. Využili přitom výhody BTT struktury, která je nezávislá na datech co do hodnoty i objemu (v případě větší komplexnosti dat se pouze přidají úrovně stromu). Navržený algoritmus pracující nad výškovou mapou dělí terén do buněk podle toho, jak je potřeba přidat detail, případně změnit data v určitých oblastech. Při potřebě zvýšení rozlišení se ohraničí čtvercová oblast, kde je potřeba rozlišení zvýšit a vytvoří se pro ni nová menší výšková mapa nahrazující odpovídající část méně detailní původní mapy. Tato se vyplní modifikovanými či nově vytvořenými daty. Poté se pouze aktualizuje BTT struktura – vytvoří se uzly reprezentující danou oblast a přepočítají se zdola-nahoru hodnoty chyby aproximace pro jednotlivé úrovně. I přes nedostatky ROAM algoritmu se nám toto řešení zdá být zajímavé zejména pro možnost dodatečného dělení každé oblasti terénu v jiném detailu.

BTT užívá nad výškovými mapami také algoritmus SOAR (36). Jeho základ tvoří pouhé sestavení triangulace průchodem BTT shora-dolů, v čemž je implementačně velmi jednoduchý. Nepoužívá žádné prioritní fronty jako ROAM a provádí triangulaci každý snímek od počátku (nulová koherence mezi snímky). To ho pro nás staví do úrovně těžko použitelných algoritmů, neboť extrémní nároky na CPU při větším terénu by se těžce odstraňovaly. Přínosem je však nově nadefinovaná metrika, která je monotónní, má monotónní projekci a je snadno vyčíslitelná. Algoritmus byl rozšířen v (37) o alternativní izotropní i anizotropní[17] metriku, podrobný popis implementace, a především o návrh zpracování terénů, které se nevejdou do paměti. V implementačních poznámkách najdeme také popis řešení viditelnosti a ořezání scény využitím BTT stromu a elegantně řešené morfování, které parametr pro interpolaci odvozuje z metriky chyby, což zajišťuje plynulý přechod závislý na pohybu pozorovatele. Ačkoli algoritmus řešení LODu není kompetitivní vzhledem k ROAM nebo RUSTiC, práce je velmi bohatá, řeší všechny detaily zobrazování terénu a nalezneme zde i pěknou rešerši předchozích prací.



Obrázek 2.14: Nerovnoměrné dělení povrchu strukturou QuadTIN. Při dělení se využívá existujících vrcholů.

Trochu odlišné využití BTT přinesl algoritmus postavený na QuadTIN struktuře popsaný v (21). Na rozdíl od ostatních zde zmíněných, tento algoritmus umí pracovat s TIN daty namísto výškové mapy. Ve fázi předzpracování vytváří QuadTIN strukturu ze vstupních dat. Tato struktura obsahuje informace o postupném binárním dělení terénu. Zvláštností je, že namísto pravidelného dělení pomocí pravoúhlých trojúhelníků, jako to dělá např. (22), se zde k dělení používají vrcholy

[17] Anizotropní metrika má odlišné hodnoty v různých směrech měření (pro různé pozice pozorovatele ve stejné vzdálenosti). Například pokud je pozorovatel kolmo nad terénem, chyby jsou na něm daleko méně patrné a izotropní metrika nedává optimální výsledky.

původní sítě. Několik prvních dělení ukazuje Obrázek 2.14. Pokud žádný vhodný vrchol neexistuje, ale plochu je z důvodu restrikce stromové struktury třeba dále dělit, doplňují se náhradní vrcholy. Tato fáze je časově náročná (při každém dělení se hledá bod nejblíže středu dělené hrany), ale provádí se off-line. Během zobrazování se prochází binární strom podle aktuálního pohledu a volí se části k zobrazení. (21) v implementaci nepoužívá monotónní metriku chyby, takže je nutné řešit závislosti, ale tato vlastnost by se dala změnit vhodnou metrikou. Pro zobrazování je použit stejný postup jako v (23), tedy vytvoření hamiltonovské cesty v duálním grafu. Za nedostatek algoritmu by se dala považovat problematická implementace morfování. Autoři uvádí, že při odstraňování vrcholu je těžké zabránit popping efektu bez ponechání vrcholu v síti a vedení záznamu o vrcholech, které se mají v čase odstranit.

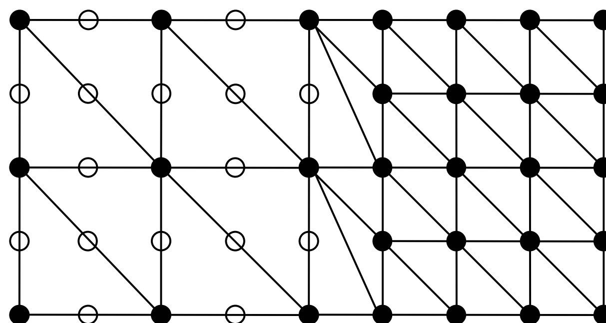
Lépe než QuadTIN využívá vlastností TIN algoritmus popsaný v (38) a nazvaný BDAM. Používá opět BTT na reprezentaci dělení terénu, ale listy BTT představují obecné TIN, které jsou vytvořeny v intenzivní fázi předzpracování. Každý jeden velký blok zahrnuje až 8 192 trojúhelníků, které byly vygenerovány nějakým pohledově nezávislým LOD algoritmem a které byly uloženy do jediné dvojice vertex a index bufferu optimalizované pro využití vertex cache na grafické kartě. Je to jeden z prvních algoritmů na zobrazování terénu, který intenzivně využívá výkonu GPU. Na rozdíl od jiných prací, (38) se také zabývá texturováním terénu za použití kvadrantového stromu a out-of-core systému na načítání. V (39) je algoritmus rozšířen o zpracování extrémně velkých dat, pro která není 32bitový IEEE formát čísla v plovoucí řádové čárce dostačující z hlediska přesnosti. Řešení je založeno na rozdělení terénu do obdélníkových oblastí a používání dvou souřadnic pro určení segmentu a offsetu.

2.3.5 Statický a hybridní LOD

Již v předchozích odstavcích jsme vzpomenuli některé algoritmy, které dynamicky řeší LOD pro terén s využitím předem generovaných bloků dat, například RUSTiC (33) či BDAM (38). Svou povahou by tedy bylo možné označit je za hybridní přístupy a zmínit je v tomto odstavci. Pro těsnou vázanost na jiné algoritmy jsme je však uvedli jinde. Přesto jsou v jednom ze základních principů podobné algoritmům uvedeným dále, totiž větším využitím GPU na základě méně intenzivního hledání optimálních aproximací. V tomto odstavci se pokusíme představit různé nápadité přístupy k řešení LODu pro terén, které zmíněný princip dále rozvíjejí. Jak jsme již zmínili dříve – čistě statický LOD není aplikovatelný na zobrazování terénu, a tyto hybridní metody využívají tedy statických dat v souznění s nějakým dynamickým řízením jejich napojení. Často vynikají snadnou implementací, ale zároveň si řešení některých jejich postranních problémů (například spojité napojování statických bloků) vynutí mnohé experimentování a ladění pro konkrétní aplikaci. Obsáhlé srovnání dvou hlavních zástupců čistě dynamického (ROAM) a statického (GeoMipMapping) přístupu najdeme v (40).

Jeden z prvních algoritmů navržených pro intenzivnější využití GPU při zobrazování terénu navrhl de Boer jako paralelu k mipmapám používaným u textur (41). Ačkoli dnes se nám de Boerovo řešení jeví jako přirozené, v průběhu několika let po jeho publikaci bylo velice často používáno a vylepšováno. Algoritmus nazvaný *GeoMipMapping* je založen na reprezentaci terénu pomocí kvadrantového stromu, v němž listy představují čtvercové pravidelné mřížky o velikosti

. Tyto mřížky – bloky – jsou uloženy v různých úrovních detailu; každá nižší úroveň je tvořena implicitně stejnými daty, z nichž jsou vybrány jen liché vrcholy. Za běhu se vybírá patřičné dělení každého bloku pouze na základě horizontální vzdálenosti bloku od pozorovatele. Napojování sousedních bloků zobrazených v různém detailu je provedeno pouze volbou jiného index bufferu, který na okraji detailnějšího bloku vynechá vrcholy, jež nejsou součástí sousedního bloku (viz Obrázek 2.15). V práci není zmíněna žádná explicitní restrikce na sousední bloky, ale dá se předpokládat, že použitá metrika chyby neumožňuje dvěma sousedním blokům přiřadit úrovně o větším rozdílu než 1. V opačném případě by totiž bylo nutné vytvořit velké množství index bufferů pro pokrytí všech kombinací.

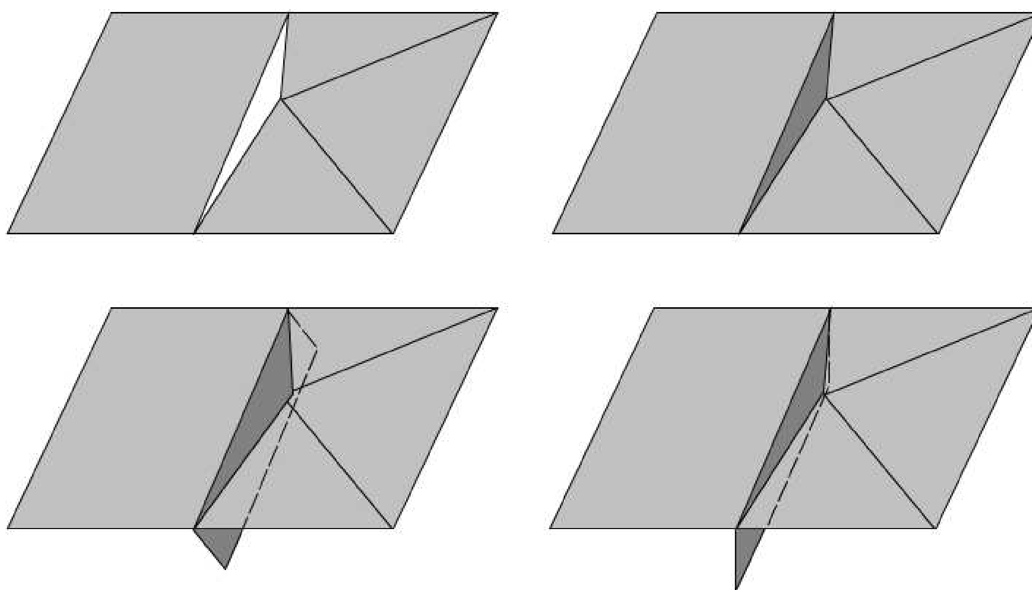


Obrázek 2.15: GeoMipMapping: Napojení dvou sousedních odlišně dělených bloků. Nevyplněné puntíky představují vrcholy, které nejsou v méně detailní úrovni použity.

Nedostatkem algoritmu je zmíněná metrika chyby, která zohledňuje pouze horizontální vzdálenost od pozorovatele a ne úhel pohledu, resp. velikost chyby v prostoru obrazovky po projekci. To ho činí hůře použitelný pro situace, kdy se pozorovateli umožní velká vertikální volnost (například letecký simulátor), ale metrika by šla jistě nahradit jinou. Hlavní výhoda algoritmu je pak v zobrazování celých bloků najednou. Pravidelná mřížka navíc umožňuje velmi dobrou optimalizaci pro využití vertex cache grafické karty. Konečně jednoduchá implementace zajistila, že se tento algoritmus stal na delší čas velmi oblíbeným.

Podobné řešení jako de Boer navrhl i Ulrich (42), avšak namísto pravidelných mřížek reprezentujících napojované bloky se rozhodl použít TIN vytvořené ve fázi předzpracování. Výhoda tohoto přístupu spočívá v obecně lepší aproximaci za užití stejného počtu trojúhelníků. Nevýhodou je jednak daleko náročnější proces předzpracování, jednak problematičtější řešení spojitého napojení sousedních bloků. S problémem spojitosti se však Ulrich potýká nápaditým způsobem a nabízí hned několik řešení. Prvním z nich je dynamické generování přesné výplně vzniklých mezer, jak ukazuje Obrázek 2.16. Mezery je sice snadné odhalit, ale generování geometrie při každé změně je časově náročné. Jako další řešení navrhuje již ve fázi předzpracování napevno vytvořit ke každému okraji zkosenou obrubu, která díru překryje. Obrubu je možné zobrazovat stále, ale nemusí být snadné najít takovou velikost a úhel, aby viditelně nenarušovala sousední blok (viz Obrázek 2.16).

Jako neposlední řešení navrhuje Ulrich vytvoření svislého závoje o konstantní výšce kolem všech bloků, jak ukazuje Obrázek 2.16 vpravo dole. Toto řešení nakonec implementuje do svého programu. Nápaditost dvou naposled zmíněných řešení spočívá v tom, že se nehledá perfektní spojité napojení geometrie, ale vizuálně dobrý výsledek. V jiných oblastech se Ulrichův postup od GeoMipMappingu příliš neliší, (42) však dále rozebírá potahování terénu texturou, kopresi dat a out-of-core zpracování.



Obrázek 2.16: Různé způsoby zakrytí praskliny mezi bloky. Vlevo nahoře vidíme prasklinu vzniklou spojením dvou odlišně dělených bloků. Vpravo nahoře je znázorněno vyplnění praskliny přesně generovanou geometrií. Vlevo dole vidíme užití šikmé obruby a vpravo dole svislý závoj.

V naší dřívější práci (43) jsme použili GeoMipMapping na pravidelné mřížky jako de Boer, ale protože naše metrika nezajišťovala žádné restrikce na úroveň detailu sousedních bloků, byli jsme nuceni vyzkoušet Ulrichovo řešení (nebo vygenerovat velké množství index bufferů pro různá napojení). Spojité napojování geometrie se v našem případě však ukázalo příliš časově náročné, a vytváření obruby i závoje mělo velké vizuální nedostatky, často horší než prasklina samotná. Pro přidanou geometrii (obrubu či závoj) nebylo v konkrétním případě triviální nalézt vhodné normály a souřadnice textury. Textura byla proto roztažená v jednom směru a špatná normála často produkovala nesprávné osvětlení, opticky velmi rušivé. Námi navržené řešení spočívalo v zobrazování terénu dvakrát pro jeden snímek. První zobrazení proběhne jako obvykle, avšak při druhém se celý terén posune kousek směrem dolů. To zaplní díry v prasklinách daty velmi podobnými těm, která by tam správně měla být (za předpokladu ne příliš náhlé změny na tak malém úseku – jedná se řádově o jednotky až desítky centimetrů). Teoreticky by se tak náročnost zobrazování mohla až zdvojnásobit, avšak řešení využívá toho, že většina pixelů se vyjme ze zpracování díky hloubkovému testu. Naše experimenty ukázaly, že výsledná rychlost klesne pouze v rozsahu 5 %.

Další užití GeoMipMappingu můžeme nalézt v (44), kde je napojování sousedních oblastí pro změnu řešeno přechodovými pásy. Tyto pásy jsou generovány pro všechny možné kombinace

sousedních oblastí. Při zobrazování se kreslí každý blok pomocí 5 nezávislých částí (4 okraje a 1 střed), což oproti předchozím přístupům obnáší pětkrát více volání patřičných API funkcí na zobrazování. To je z našeho současného hlediska daleko méně efektivní. Algoritmus je však velice snadno implementovatelný, což může být důležitým kritériem.

Jinou adaptaci uvedl Wagner v (45), kde představuje zřejmě jednu z prvních implementací morfování na GPU (další nalezneme kupříkladu v (46)). Dřívější implementace byly zpravidla prováděny při předání vrcholu grafické kartě, nebo přepočtem celého jednoho bloku v cyklu na CPU. Wagner dokonce uvádí srovnání, ze kterého je patrné, že morfování prováděné na grafické kartě je téměř „zdarma" (méně než 1 % výkonu), zatímco prováděné na CPU sníží výkon systému až na polovinu. Další optimalizací, kterou v (45) najdeme, je užití *PVS* algoritmu (potentially visible sets). PVS představuje úplný graf, ve kterém jednotlivé vrcholy zastupují vybrané oblasti terénu. V náročné fázi předzpracování se provede test vzájemné viditelnosti každého páru oblastí a uchová se jednobitová informace o tom, zda je z jedné oblasti do druhé vidět. Během zobrazování se pak zbytečně nevykreslují ty oblasti, které nejsou vidět z oblasti, v níž se zrovna nachází pozorovatel. PVS můžeme přirozeně použít téměř v každém řešení zobrazování terénu, přesto je (45) jedna z mála prací, kde je věnována pozornost ořezávání zakrytých částí scény.

Inovativní řešení napojování sousedních oblastí navrhuje (47). Poměrně nový algoritmus dělí čtvercový terén pomocí kvadrantového stromu na různě velké oblasti, které jsou mezi sebou vždy spojitě napojeny. Toho je docíleno použitou metrikou, která zohledňuje délku hrany oblasti a její projekce. Dva sousední bloky vždy sdílí hranu, nebo část hrany a tato je tedy dělena stejně pro oba bloky. Samotné napojování pak probíhá uvnitř oblastí, ne mezi oblastmi, takže není třeba zavádět restrikce a mít znalosti o okolí. Každá oblast se nakonec rozdělí do 4 trojúhelníkových sektorů, kterou mohou být obecně různé úrovně detailu, a tyto se spojí pomocí předem vygenerovaných přechodových pásků.

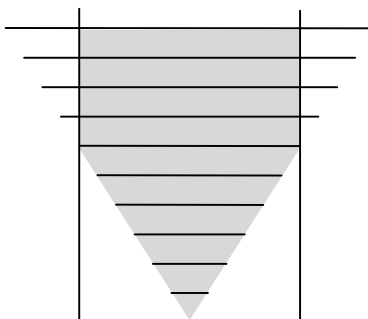
V řadě dalších prací najdeme opakování již popsaných postupů, jejich různé permutace, adaptace na konkrétní podmínky či drobná vylepšení v tom či onom směru. (48) například provádí progresivní aktualizaci dat geometrie na grafické kartě – detailnější modely jsou tvořeny méně detailními (ty už jsou zpravidla v paměti grafické karty) společně s nově přidanými daty. V (49) pro změnu najdeme stručný popis stránkovacího mechanismu pro trojúrovňovou vyrovnávací paměť. V práci bohužel není uveden žádný komentář k vysvětlení velmi špatných výsledků pro větší vstupní data; domníváme se, že to bylo způsobeno neideálním využitím paměti grafické karty. Využití Shader Modelu 3.0 najdeme v (50), kde jsou data geometrie oddělena od topologie pomocí výškové mapy uložené v textuře. Pro všechny bloky jedné úrovně se tedy používá jediná topologie a geometrická data jsou načítána ve vertex shaderu z textury.

2.3.6 Clipmap

Již před 10 lety byla navržena zvláštní struktura pro mapování souvislé textury na rozlehlé terény (51) a později byla dopodrobna rozebrána (52). Struktura nazvaná *clipmap* byla použita do systému pro zobrazování potenciálně nekonečného terénu bez omezení dohledové vzdálenosti. Její základní myšlenka se opírá o vlastnosti mipmap – textura je uložena v různých rozlišeních a při vzorkování se vždy využije ta úroveň detailu, která přísluší aktuálnímu gradientu texturové souřadnice na obrazovce. Na větší trojrozměrné objekty tak může být

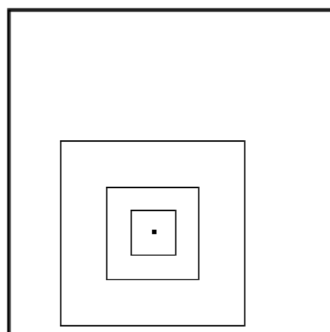
v různých místech použito několik úrovní. Terén představuje extrémně velký objekt, který je třeba v různých jeho částech zobrazit s různou přesností, a tato se může lišit i o několik řádů. Mipmapa je hierarchie, v níž každá úroveň reprezentuje tatáž data v jiném rozlišení.

My si však povšimneme, že při zobrazování není třeba mít pro každou úroveň k dispozici všechna data. Z každé úrovně je vidět maximálně to, co se vejde na obrazovku. Velikost jednotlivých úrovní tedy můžeme limitovat nějakou konstantou (podle požadavku kvality), tu označíme jako velikost clipmapy. Za clipmapu pak označíme hierarchickou strukturu, v níž se nachází výřezy z různě detailní reprezentace jedněch dat. Každá úroveň má stejnou velikost, ale odlišné rozlišení, tudíž reprezentuje jinak velkou oblast. Konečně souvislost všech výřezů je dána jejich společným středem. Nejlépe to ukazuje Obrázek 2.17. Můžeme si na něm všimnout také toho, že nižší úrovně clipmapy mohou mít menší velikost, než je stanovena velikost clipmapy. Všechny stejně velké nebo větší jsou však ořezány na jednou velikost.



Obrázek 2.17: Úrovně clipmapy. Z každé úrovně je vybrána jen nutná část, zbytek je ořezán. Nejméně detailní úrovně mohou mít menší velikost než je stanovena velikost clipmapy.

Princip použití clipmapy je velmi snadný – do jednotlivých úrovní clipmapy se nahrají data odpovídající čtvercové oblasti vycentrované kolem pozorovatele. Každá úroveň reprezentuje jinak velkou oblast, ale mají společný průnik, jímž je nejdetailnější úroveň clipmapy (viz Obrázek 2.18). Při zobrazování zvolené oblasti použijeme vždy tu nejdetailnější úroveň clipmapy, která zahrnuje danou pozici. Když se pozorovatel pohybuje, také úrovně clipmapy by se měly pohybovat s ním, proto je potřeba aktualizovat data v nich uložená. Výhoda tohoto přístupu je ta, že nižší úrovně je potřeba aktualizovat jen velmi zřídka, neboť jejich vzorek reprezentuje velkou vzdálenost v prostoru scény.



Obrázek 2.18: Tři nejdetailnější úrovně clipmapy. Úrovně jsou vycentrovány kolem pozorovatele, ale každá reprezentuje jinak velkou oblast.

Ačkoli původní využití clipmapy bylo pro nanesení textury na terén, již v (52) bylo navrženo další využití pro reprezentaci geometrie. Přesto ji pro geometrii poprvé využili až Losasso a Hoppe v (53). Princip použití clipmapy zůstal naprosto stejný, pouze se namísto barevné textury aplikovala na výškovou mapu. Terén se zobrazuje pomocí dynamicky generovaných index bufferů, které indexují vrcholy v pravidelných čtvercových mřížkách reprezentujících jednotlivé úrovně. Nejdetailnější mřížka je vždy zobrazována celá, z ostatních je vybrán pouze okraj kolem výřezu, jež pokrývá vyšší úroveň. Speciálně je třeba ošetřit přechody mezi mřížkami, aby mezi nimi nevznikly praskliny. Losasso a Hoppe to řeší pomocí definování přechodových oblastí, což jsou oblasti na okrajích každé mřížky, kde se výška terénu interpoluje mezi dvěma sousedními úrovněmi. Na začátku přechodové oblasti je výška vrcholů plně dána výškou v dané úrovni a na konci přechodové oblasti, tj. na hranici mřížky, je naopak rovna výšce v nižší úrovni clipmapy. Tento postup řeší nejen praskliny, ale také vytváří přirozené morfování pro potlačení popping efektu.

Hlavní výhodou clipmap jsou paměťové nároky. Každá úroveň vezme stejné množství paměti, takže s rozměry terénu se paměťové nároky zvyšují pouze logaritmicky. Uveďme si příklad s použitím clipmapy o velikosti 1 024: Na čtvercový terén o šířce 8 192 vzorků potřebujeme 5 úrovně clipmapy, abychom zobrazili celý terén i v případě, že se nacházíme na jeho okraji. To je dohromady 20 MB dat, pokud počítáme se 4 byty na jeden vzorek. Na zobrazování terénu o šířce 32 768 vzorků potřebujeme 7 úrovní clipmapy, tedy 28 MB za stejných předpokladů – to je o polovinu více dat na 16x větší terén.

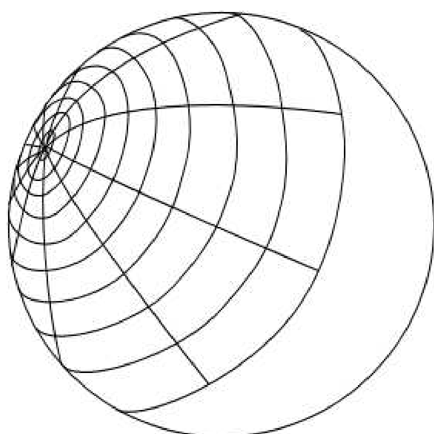
Stejně jako nízké paměťové nároky clipmap, také vysoká efektivita zobrazování je založena na tom, že v danou chvíli pracujeme s daty jen v takovém rozlišení, v jakém jsou právě třeba. Vzdálené oblasti tedy uchováváme i zobrazujeme v nízkém rozlišení, a tím dosahují clipmapy excelentních výsledků při zobrazování terénu. Jedná se však o dvousečnou zbraň. Představme si situaci, že potřebujeme rychle změnit úroveň detailu zobrazované oblasti (příkladem může být použití dalekohledu, který najdeme v každé akční hře). S takovou situací má samozřejmě problém většina algoritmů založených na koherenci mezi snímky, nejlépe si povedou statické LODy se všemi daty stále připravenými.

Další nevýhodou je samozřejmě použití pravidelné mřížky, čímž ztrácíme různé lokální vysokofrekvenční informace (pokud nepředimenzujeme hustotu mřížky). Představme si menší kopec, který v dálce leží celý mezi dvěma hranami málo detailní mřížky – v lepším případě uvidíme jakousi vyvýšeninu, v horším se kopec ztratí docela. Vyšší hustotou mřížky, tedy velikostí clipmapy, můžeme tento efekt posunout k vyšším frekvenčním spektrům, ale nemůžeme jej odstranit. Nicméně to je globální problém pravidelného vzorkování.

Losasso a Hoppe také popisují postup progresivního nahrávání detailu. Jednotlivé úrovně clipmap totiž na zdrojovém médiu nemusíme mít uchovány přímo, protože více detailní úroveň můžeme definovat na základě nižší úrovně s přidáním detailů. Při aktualizaci každé úrovně načteme pouze data definující vysokofrekvenční změny a rekonstrukci provedeme s využitím nižších úrovní, které jsou již načtené. Rekonstrukce dat (ať už výškových nebo normál) je však nejpomalejší částí celého algoritmu, a tedy nevítaná zátěž CPU. I v případě, že máme všechna data uložena v té podobě, v jaké je zobrazujeme, můžeme s výhodou využít progresivního načítání detailu k udržení konstantní frekvence zobrazování. Nadefinujeme časový limit pro načítání dat během jednoho snímku a po překročení tohoto limitu začneme se zobrazováním terénu, i pokud všechna data nejsou ještě k dispozici. Musíme pouze zajistit, abychom načítali nejdříve data nižších úrovní a nezobrazovali data, která ještě nejsou načtena. Oblasti, kde nemáme potřebný detail k dispozici, použijeme nejvyšší nižší úroveň, v níž jsou data pro zobrazovanou oblast už načtena. Za cenu jisté ztráty kvality tak zajistíme plynulý chod.

Hlavní význam clipmap souvisí s příchodem Shader Modelu 3.0, který umožňuje číst data z textury ve vertex bufferu, byť s omezeními. Protože výšku můžeme vrcholu přiřadit až ve vertex shaderu, nemusíme s každou aktualizací úrovně clipmapy přegenerovat její index buffer; stačí ve vertex bufferu použít správnou souřadnici pomocí toroidního mapování. Losasso a Hoppe, poukázali na možnost budoucí implementace clipmap na GPU, ale hardware podporující Shader Model 3.0 ještě nebyl k dispozici. V krátké době po jeho uvedení se však objevila celá řada různě upravených implementací. Holkner (54) použil clipmapu nejen na data výšek, ale také pro uložení předpočítaného osvětlení povrchu (tzv. lightmap). Obecně je možné použít clipmapu na jakákoli data, která dokážeme uložit do textury. Rose navrhl čtyři různé modifikace clipmap (55), například změnu poměru velikostí mezi úrovněmi clipmap z dvojnásobku na trojnásobek. Brettell ve své práci uvádí experimentální srovnání clipmap s ROAM a GeoMipMapingem (56). Velmi podrobný popis implementace clipmap na GPU najdeme v (57), kde je mimo jiné řešeno i ořezávání částí terénu podle viditelnosti. Méně rozsáhlou implementace s českým rozborem nalezneme v (58).

Se Shader Modelem 4.0 přišla možnost definovat jednorozměrné pole textur, které je možné přímo indexovat ve vertex či pixel shaderu. Toho se dá také využít při implementace clipmap, neboť všechny úrovně je možné uložit do jediného pole a výběr úrovně řešit přímo na GPU. Při vhodně navržené implementaci by tak bylo možné zobrazovat všechny úrovně clipmapy najednou. Pole texturu užívá například (59), ovšem ne pro uložení výškové mapy, ale barevné textury podle původního návrhu (51). Existence řešení geometrických clipmap pomocí pole textur nám není známa.



Obrázek 2.19: Síť představující sférickou clipmapu je vždy centrována kolem pozorovatele.

Poněkud odlišný přístup zvolili Clasen a Hege, kteří pro zobrazování povrchu planet navrhli strukturu nazvanou *spherical clipmap* (60). Klasická clipmapa se pro takový účel nedá snadno použít, proto přišli se sítí reprezentující povrch polokoule nebo její části. Síť je, podobně jako clipmapa, složena z vycentrovaných útvarů (v tomto případě kulových prstenců) narůstající velikosti, ale klesajícího detailu, jak ukazuje Obrázek 2.19. Problémem tohoto přístupu je komplikovanější parametrizace obdélníkové textury na základě sférických souřadnic. Textura – výšková mapa – je anizotropně roztažena do dvourozměrné obdélníkové plochy a odtud za běhu vzorkována. Postup byl později rozšířen o asynchronní načítání a syntézu dat (61).

2.3.7 Alternativní přístupy

Některé algoritmy se nám pro jejich nekonvenční přístup nepodařilo přímo zařadit, přesto však stojí za zmínku. S velmi zajímavým návrhem přišli Dachsbacher a Stamminger (62). Jejich algoritmus měl několik kroků. V prvním se z výškové mapy vybrala oblast, která je právě vidět. Ve druhém kroku se vzorkům této oblasti přiřadily hodnoty jejich důležitosti – vzorky blíže pozorovateli mají větší důležitost, stejně tak vzorky přivrácené směrem k pozorovateli… Celá oblast se pak roztáhne do textury se zohledněním důležitosti vzorků tak, aby všechny vzorky po projekci zabíraly přibližně stejně velkou oblast obrazovky. Z takto předzpracovaných dat se vytvoří definice geometrie, stále ještě uložená v textuře – jednotlivé souřadnice vrcholů se zapíší do barevných složek. Nakonec se tato síť zobrazí pomocí připraveného bufferu indexujícího jednotlivé vzorky. Autoři navíc aplikují v různých fázích různé úrovně procedurálně generovaného detailu. Také barva je generována částečně procedurálně – každý vzorek má přiřazen index do seznamu materiálů, ale turbulentní funkce je užita pro rozostření přechodů mezi materiály. Za hlavní negativum tohoto algoritmu považujeme roztahování sítě podle důležitosti, neboť to je prováděno na CPU. Není nám známa novější implementace, která by více využila grafické karty.

Podobný přístup, který se nazývá *projekční mřížk*a, byl použit například v (63), ale i v jiných pracích. Základní princip je snadný – na pravidelnou mřížku vytvořenou v prostoru obrazovky se aplikuje inverzní operace k projekci a mřížka se tak roztáhne přes viditelnou část terénu. Odtud

se načtou vzorky geometrie, tedy zpravidla data výškové mapy, která se již transformují běžným způsobem.

3 Zjemňování geometrie

V předchozí kapitole jsme se zeširoka věnovali možnostem zjednodušování geometrie ať už z důvodu rychlejšímu zobrazování, menších paměťových nároků či progresivního načítání. Ledaskdy však máme i opačnou potřebu geometrii „zkrášlit“ – přidat detail, vyhladit či jen zvýšit hustotu reprezentující sítě. Je celá řada možností jak toho dosáhnout a volba té které závisí předně na tom, co nás k tomu vede. Uveďme si několik příkladů, které nás budou dále motivovat:

1. Výpočet osvětlení počítaný pouze ve vrcholech sítě (nikoli pro každý pixel) závisí svým výsledkem na hustotě sítě. V případě kuželového zdroje světla může v extrémním případě řídké sítě dojít až k úplné absenci světla (kužel protne povrch sítě, aniž by zasáhl kterýkoli vrchol). Lepších výsledků dosáhneme větší hustotou sítě bez změny tvaru objektu, pro kterou s úspěchem využijeme lineární interpolaci původních vrcholů.

2. Na siluetě málo detailního modelu je vždy snadno vidět hrubost reprezentace, která se projeví ostrými hranami tam, kde pozorovatel očekává hladkou křivku. Jednoduché zvýšení hustoty sítě zde nepomůže, protože silueta zůstane stejná. Je zapotřebí přidané vrcholy umístit tak, abychom původní hrany zaoblili. K tomu se často využívá složitější interpolace či aproximace povrchu pomocí parametrických ploch nebo subdivision surfaces.

3. Dodefinováním vysokofrekvenční informace lze často výrazně zlepšit vizuální dojem z prezentovaných dat. Detaily lze úspěšně vytvořit procedurálně, aniž by si pozorovatel všiml jejich syntetické povahy, například využitím fraktálů, šumu nebo turbulentní funkce na reálná data. Aplikovat se dá jak na barevnou texturu, tak i geometrii.

Při zobrazování terénu nám jde zpravidla o přírodní útvar, a o těch obecně plyne, že pro jejich náhodný a často soběpodobný charakter, je lze velmi dobře synteticky utvářet. Proto bylo procedurálnímu generování dat v oblasti zobrazování terénu věnováno už mnoho pozornosti, a my se mu v této práci vyhneme za účelem důkladnějšího prozkoumání jiných oblastí. Naproti tomu zvyšování hustoty sítě zmíněné v bodě 1 je triviální operace, která si zde nezaslouží většího rozboru. My se zde chceme věnovat technikám zjemnění a vyhlazení povrchu pomocí různých složitějších ploch nahrazujících geometrická primitiva původní sítě. Konkrétně se zaměříme na pojem subdivision surfaces a okrajově se dostaneme i parametrickým plochám. Náš celkový záměr je pochopitelně najít prostor pro tyto zmíněné objekty při zobrazování terénu.

3.1 Parametrické plochy

Parametrickou plochu můžeme definovat pomocí spojité injektivní funkce $S: R^2 \rightarrow R^3$. $\mathrm{Im}(S)$ představuje parametrickou plochu v trojrozměrném prostoru a funkci S nazýváme její parametrizací. Parametrickou plochu tedy můžeme vyjádřit soustavou parametrických rovnic o dvou parametrech a třech neznámých (neznámé v tomto případě představují souřadnice

v prostoru). S takto definovanou plochou se velmi dobře pracuje, předně je snadné vyčíslit ji v jakémkoli bodě definičního oboru (vyhodnocení tří explicitně daných funkcí pro získání souřadnic pro dané hodnoty parametrů), případně k ní nalézt tečnou rovinu. Více než obecná parametrická plocha nás bude zajímat *Bézierův plát*, což je speciální případ parametrické plochy aproximující zvolenou množinu bodů. K jeho korektní definici potřebujeme však jiný pojem:

i -tým *Bernsteinovým polynomem* stupně n nazveme polynomiální funkci

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}.$$

Bézierovým plátem řádu (n, m) definovaným $(n+1)(m+1)$ řídícími vrcholy $P_{i,j}$ nazveme parametrickou plochu, jejíž body P jsou dány vztahem

$$P(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) P_{i,j}.$$

Bézierovy pláty mají řadu užitečných vlastností. Již bylo zmíněno, že se jedná o aproximační plochy, tedy obecně nemusí procházet všemi řídícími vrcholy, avšak leží v jejich konvexní obálce a rohy plochy jsou totožné s 4 rohovými řídícími vrcholy. Hranice plochy tvoří polynomiální křivky stupně n , resp. m , a jednotlivé pláty lze spojitě napojovat, jestliže sousedním plátům přiřadíme totožné krajní body. Konečně jednou z nejdůležitějších vlastností je invariance vůči afinním transformacím – nezáleží tedy na tom, zda transformaci provedeme na řídící vrcholy nebo na výslednou plochu. V počítačové grafice se nejčastěji používá bikubický Bézierův plát, tj. takový, kde $m = n = 3$.

Adaptací obecného Bézierova plátu je *Bézierův trojúhelníkový plát* n -tého řádu zadaný $\frac{(n+1)(n+2)}{2}$ řídícími vrcholy $P_{i,j,k};\ 0 \leq i, j, k;\ i + j + k = n$. Bod plátu P je vázaný vztahem

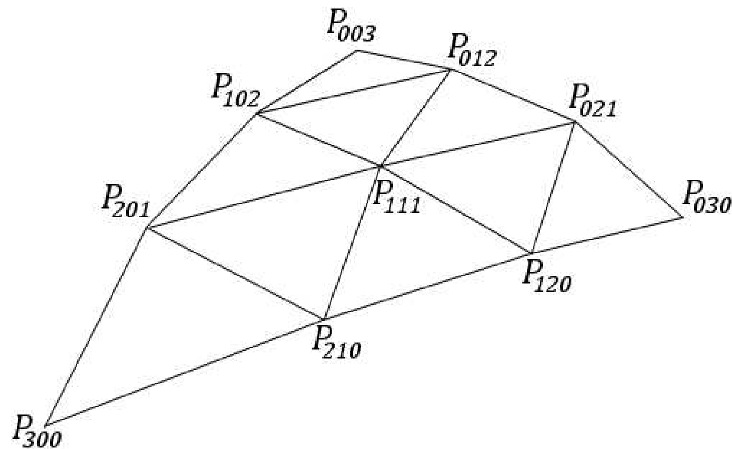
$$P(u, v, w) = \sum_{i,j,k \geq 0; i+j+k=n} \binom{n}{i\ j\ k} u^i v^j w^k P_{i,j,k} = \sum_{i,j,k \geq 0; i+j+k=n} \frac{n!}{i!\, j!\, k!} u^i v^j w^k P_{i,j,k}$$

pro všechny trojice parametrů u, v, w takové, že $u + v + w = 1$. Opět nejpoužívanější je kubický Bézierův trojúhelníkový plát zadaný 10 řídícími vrcholy, jehož povrch můžeme vyjádřit rovnicí

$$P(u, v, w) = u^3 P_{300} + v^3 P_{030} + w^3 P_{003} + 3u^2 v P_{210} + 3uv^2 P_{120} + 3v^2 w P_{021} + 3vw^2 P_{012} + 3uw^2 P_{102} + 3u^2 w P_{20}$$

Tento typ plochy má obdobné vlastnosti jako bikubický Bézierův plát, ale jeho doménu tvoří trojúhelníková plocha, ne čtvercová. To se nám hodí především tam, kde máme plochu vyjádřenou trojúhelníkovou sítí, což je nejčastější forma, chceme-li zobrazovat data na grafické kartě. Obvykle jsou parametrické plochy méně vhodné pro definici geometrie, protože běžné grafické karty neumí taková primitiva přímo zobrazovat, tudíž se musí nejdříve triangulovat do sítě, kterou grafická karta přijme. Proto při interaktivním zobrazování nebývají parametrické plochy využívány přímo. Můžeme však provést implicitní nahrazení trojúhelníků původní hrubé

sítě Bézierovými trojúhelníkovými pláty pro předpis nové (oblejší) geometrie, a tuto triangulovat tak, že triangulujeme každý plát zvlášť podle jednotného předpisu. Při vhodné implementaci skutečné nahrazení trojúhelníků Bézierovými pláty vůbec nemusíme provést a výslednou jemnější síť můžeme tedy generovat přímo z původní sítě. Co více, pokud použijeme pro definici plátu pouze informace uchované pro trojúhelník, který nahrazuje, a žádné informace z okolí, můžeme toto zjemnění geometrie provést na všech trojúhelnících nezávisle. Příklad rozdělení jednoho trojúhelníku devíti jinými pomocí Bézierova plátu ukazuje Obrázek 1.1.



Obrázek 3.1: Původní trojúhelník $P_{300}P_{030}P_{003}$ byl použit jako doména pro parametrizaci Bézierova trojúhelníkového plátu. Ten byl následně rozdělen na 9 menších trojúhelníků.

Zmínili jsme se, že pro definici plátu užíváme pouze informací uložených v původních vrcholech nosného trojúhelníku, tedy souřadnice v prostoru a normála plochy (ta je potřebná pro rozeznání orientace trojúhelníka). Nyní je třeba uvést, jak získat kontrolní body Bézierova plátu z těchto informací. K tomu si rozdělíme vrcholy do tří skupin: původní vrcholy ($P_{300}, P_{030}, P_{003}$), tečné vrcholy ($P_{210}, P_{201}, P_{120}, P_{102}, P_{021}, P_{012}$) a samostatný centrální vrchol (P_{111}). Poté pravidelně rozmístíme tečné vrcholy a centrální vrchol s využitím lineární interpolace:

$$P_{i,j,k} := \frac{(iP_{300} + jP_{030} + kP_{003})}{3}.$$

Následně využijeme normál v původních vrcholech k definici tečných rovin, a do těchto rovin provedeme projekce tečných vrcholů. Jejich výsledná poloha tedy bude dána vztahem

$$P_{i,j,k} := P_{i,j,k} - \left((P_{i,j,k} - P_{r,s,t}) \cdot N_{r,s,t}\right) N_{r,s,t},$$

kde $N_{r,s,t}$ označuje normálu vrcholu $P_{r,s,t}$, což je jeden z původních vrcholů zvolený takto:

$$(r,s,t) = \begin{cases} (3,0,0), & i == 2 \\ (0,3,0), & j == 2 \\ (0,0,3), & k == 2 \end{cases}.$$

Nakonec spočítáme těžiště tečných vrcholů a centrální vrchol přemístíme jeho směrem o jeden a půl násobek vzdálenosti, tedy

$$T := \frac{P_{210} + P_{201} + P_{120} + P_{102} + P_{021} + P_{012}}{6}$$

$$P_{111} := T + \frac{(T - P_{111})}{2}.$$

Obdobným postupem můžeme interpolovat i jiné atributy, nejenom souřadnice. Přinejmenším je potřeba interpolovat normálu, ale často také texturové souřadnice atp. Není však nutné, abychom všechny informace interpolovali pomocí kubického plátu, (64) navrhuje kvadratický plát pro normály a lineární interpolaci pro ostatní atributy.

Povšimněme si, že k definici řídících vrcholů jsme využili normál vrcholů. To může způsobit problémy v případě, že pro sousední trojúhelník je definována jiná normála v daném vrcholu, což je poměrně běžné, neboť jedině tak se dá správně zobrazit ostrá hrana na modelu. Důsledkem bude nekonzistentní výpočet v sousedních trojúhelnících a následná díra v geometrii. Naneštěstí není těžké nahlédnout, že tomuto efektu není možné se vyhnout bez znalosti sousední geometrie. Řešením, které se nabízí, je označit ostré hrany a zpracovat je jinak. Ovšem to zcela odstraní transparentnost užití Bézierových plátů, neboť všechna geometrie se již v době tvorby bude muset dělat s ohledem na taková pravidla. To je velmi zásadní, ale zřejmě jediný nedostatek tohoto přístupu.

Užití kubických Bézierových trojúhelníkových plátů na transparentní zjemnění geometrie, jak jej uvádíme zde, bylo detailně popsáno v (64) a také implementováno hardwarově na grafických kartách (65). Jiné hardwarové řešení využívalo obecnou definici Bézierových plátů a bylo závislé na explicitním zadání řídících vrcholů (66). Od prvního zmíněného řešení bylo během času upuštěno, druhé zůstává nepoužito, pravděpodobně pro vzájemnou nekompatibilitu umocněnou nutností vytvářet speciální modely pro různý hardware (v tomto směru lepší řešení bylo (65) pro svoji transparentnost, ale případné praskliny v geometrii nebyly tolerovatelné). S největší pravděpodobností se však v blízké budoucnosti Bézierovy pláty nebo jiné parametrické plochy opět začnou hardwarově implementovat.

Bézierovy trojúhelníky použili pro zjemnění geometrie také Boubekeur a Schlick (67). S nespojitostí geometrie si poradili dodáváním patřičných dat spolu s geometrií. Jejich přístup je zajímavý v tom, že geometrii (ale i jiná data) trojúhelníku předávají na grafickou kartu v podobě konstant s každým zobrazovaným trojúhelníkem hrubé sítě zvlášť. Model pak zobrazují po trojúhelnících hrubé sítě. Pro každý se použije předem vytvořená velmi jemná síť i o stovkách trojúhelníků, v jejíchž vrcholech se vyhodnotí Bézierův plát podle předaných konstant. Autoři zobrazují celý model v jediné úrovni detailu, takže neřeší nespojitosti mezi jinak dělenými oblastmi. Hlavním nedostatkem této metody je zpracování každého trojúhelníku samostatně, což je limitující i pro zobrazování jiných modelů, než terénu (na který by postup patrně nebyl aplikovatelný).

Aplikaci bikubických Bézierových plátů přímo na terén nalezneme v (68). Algoritmus využívá různé úrovně dělení plátu pro řešení LODu. Terén je dělen do čtvercových ploch, které jsou

triangulovány zvlášť v potřebném detailu. Návaznosti mezi odlišně dělenými oblastmi se řeší dynamickým vytvořením triangulace přesně podle potřeb. I všechna data jednotlivých oblastí jsou počítána za běhu, dokonce každý snímek znovu, takže se tento přístup pro vysokou zátěž CPU nedá považovat za příliš úspěšný. Implementace s lepším využitím GPU by však mohla poskytnout zajímavé výsledky.

3.2 Subdivision surfaces

Mezi parametrickými plochami se nejvíce využívají různé formy po částech polynomiálních ploch, se kterými se v mnoha směrech dobře pracuje. Jejich velkým nedostatkem je však problematické vyjádření nepravidelných sítí, přesněji sítí, jejichž vrcholy nejsou všechny jednoho stupně (6 pro trojúhelníkové sítě, 4 pro čtyřúhelníkové). Přitom z Eulerovy formule pro rovinné grafy víme, že takovouto pravidelnou sítí můžeme vyjádřit pouze plochy topologicky ekvivalentní nekonečné rovině, válci nebo torusu. Z toho plyne, že většinu skutečných modelů (počínaje koulí), které chceme vytvořit, nemůžeme takto reprezentovat, a tedy že do sítě musíme vložit vrcholy odlišného stupně, které nazýváme *výjimečnými vrcholy*. Vyjádření sítí s výjimečnými vrcholy pomocí polynomiálních parametrických ploch se vždy jevilo obtížné, především při potřebě spojitého napojení vyššího řádu. I z tohoto důvodu byly hledány jiné způsoby vyjádření hladké plochy. Jedním ze způsobů je užití *subdivision surface*s, která nezávisle na sobě před 30 lety objevily dvojice Catmull s Clarkem (69) a Doo se Sabinem (70). Subdivision surface můžeme zjednodušeně popsat jako plochu limitně vyjádřenou opakováním zvoleného postupu zjemňování sítě. Zjemňující postup si můžeme také představit jako funkci aplikovanou na model reprezentovaný polygonální sítí (nemusí to být pouze trojúhelníky), jejímž výsledkem je složitější a hladší aproximace původní sítě. Nekonečným počtem opakování aplikace této funkce na výsledek předešlého výstupu bychom měli dojít k limitní ploše, jejíž vlastnosti (například stupeň spojitosti) jsou dány zvoleným dělícím schématem.

Dělení lze vždy vyjádřit sadou pravidel aplikovaných na všechna primitiva (vrcholy, hrany, stěny) sítě. Sada pravidel využívá konečné okolí zvoleného primitiva a na jeho základě mění lokálně síť. Pokud je pravidlo určeno pro vrcholy, pozměňuje jejich pozici. Pokud je určeno pro hrany nebo stěny sítě, vloží za dané primitivum do sítě nový vrchol. Takto vytvořený vrchol nazýváme *hranový*, resp. *stěnový*. Pravidla kromě definice souřadnic také říkají, jak nově přidané vrcholy spojit. Na pravidlo dělícího schématu můžeme také nahlížet jako na funkci, která ohodnocuje blízké vrcholy nějakého primitiva a výsledný vrchol je pak lineární kombinací vrcholů ohodnocených nenulovou vahou. Pro takovéto chápání pravidla můžeme použít pojem nosiče funkce k definování *nosiče schématu* S (značme $\mathrm{supp}(S)$):

$$\mathrm{supp}(S) \cong \bigcup_{f \in A} \mathrm{supp}(f),$$

kde A je množina pravidel schématu S a $\mathrm{supp}(f)$ je nosič funkce definovaný jako uzávěr množiny

$$\mathrm{supp}(f) \cong \overline{\{x \in D(f) : f(x) \neq 0\}}.$$

Samozřejmě pro snadnou implementaci a efektivní výpočet je výhodou, pokud má schéma co nejmenší nosič. Zpravidla mluvíme o n *-okolí* vrcholu nebo stěny (značme $\mathrm{ring}_n(x)$), která popisujeme pomocí indukce. Pro vrchol V tedy definujeme

$$\mathrm{ring}_0(V) \stackrel{\mathrm{def}}{=} \{V\}$$

a

$$\mathrm{ring}_{i+1}(V) \stackrel{\mathrm{def}}{=} \mathrm{ring}_i \cup \{W \text{ vrchol: } \exists \text{ hrana } (V, W)\}.$$

Za pomoci okolí vrcholu definujeme i n -okolí stěny F jako množinu

$$\mathrm{ring}_n(F) \stackrel{\mathrm{def}}{=} \bigcup_{V \in F} \mathrm{ring}_n(V).$$

Pro většinu běžně užívaných schémat tvoří jejich nosič 1-okolí nebo 2-okolí stěny. Větší okolí zpravidla přináší příliš velké komplikace a menší užitek. Velikost nosiče nepřímo ovlivňuje také matematické vlastnosti výsledné plochy, především úroveň spojitosti. Připomeňme, že při použití Bézierových trojúhelníkových plátů jaké jsme dříve popsali (64) využíváme pouze 0-okolí trojúhelníka původní sítě. Zmínili jsme také, že 0-okolí je nedostatečné pro spojité (C^0) navázání, pokud povolíme násobné normály ve vrcholech. Nosič schématu je paralelou k řídícím vrcholům parametrické plochy, proto dříve zavedené označení také budeme používat.

Jak jsme již uvedli, při aplikaci dělícího schématu využíváme lineární kombinace řídících vrcholů pro určení souřadnic nových či změněných vrcholů. Ty se záhy stávají řídícími vrcholy pro další iteraci. Z lineární algebry víme, že tento postup můžeme vyjádřit také maticí (matice schématu), kterou vynásobíme vektor řídících vrcholů. Takovýto náhled je velice užitečný, protože zkoumáním vlastností matice schématu můžeme odvodit také lokální vlastnosti limitní plochy (především úroveň spojitosti). S úspěchem se přitom užívá analýzy vlastních čísel a vlastních vektorů této matice. Touto oblastí se zde nechceme zabývat, podrobnější rozbor je možné nalézt v (71) nebo (72).

Pokud chceme použít subdivision surfaces pro interaktivní zobrazování, je z hlediska efektivity velmi důležité, jakým způsobem je vyhodnotíme. Jednou z možností je využití přirozeného iterativního postupu a dělit a zjemňovat síť podle pravidel dělícího schématu. Výhoda tohoto postupu je zřejmá – můžeme zastavit dělení na základě nějaké podmínky, například vypršení časového limitu nebo překročení povoleného počtu polygonů. S každou iterací je síť hladší, takže můžeme také definovat podmínku na základě kritéria konvergence. Po každém kroku provedeném na celou síť je síť v konzistentním tvaru, tj. spojitá. Pokud používané schéma není trojúhelníkové, je třeba ještě provést triangulaci, avšak obecně je síť lehce předatelná grafické kartě.

Jiný způsob na vyhodnocení sítě je zkonstruovat pevnou topologii výstupní sítě podle původních dat a zvolené složitosti a souřadnice vrcholů této nově vzniklé sítě vyhodnotit přímo, jakoby ležely na limitní ploše. Obtížnost tohoto postupu velmi závisí na zvoleném schématu a pro různá schémata najdeme mnohou literaturu na toto téma.

Důležitou vlastností při rozhodování, které schéma zvolit pro zjemnění geometrie, je způsob, jakým nahrazuje původní síť. *Interpolační* schéma v každém kroku nechává řídící vrcholy na svých místech a pouze vytváří nové. Naproti tomu *aproximační* schéma hýbe se všemi vrcholy, takže výsledná plocha obecně neprochází původními vrcholy. Interpolační schéma je těsněji vázáno se vstupní množinou vrcholů, takže dokážeme lépe odhadnout jeho chování a podobu limitní plochy. Toto svázání má obvykle za následek horší vizuální výsledky, zvláště pro data s vysokou frekvencí změny.

Množina pravidel daného schématu nemusí být vždy stejná pro všechny iterace, ale může se dynamicky měnit. Takovému schématu říkáme *nestacionární* (přirozeně schéma, které není nestacionární, je stacionární). Vedle toho definujeme také pojem *uniformního* schématu, což je takové, které zpracovává všechna primitiva jednoho typu stejně – nerozdělí tedy jednu hranu jinak než jinou.

Naším záměrem není věnovat se matematickému rozboru subdivision surfaces, ani vytvořit naučný text pro jejich studium. Možnosti těchto ploch nás zajímají pouze s ohledem na jejich případné použití při zobrazování terénu, ať už za účelem vyhlazování povrchu nebo řešení úrovně detailu, k čemuž se nám jeví být obzvlášť praktické. V dalším textu si proto pouze stručně popíšeme několik základních dělících schémat, která byla již dříve navržena a jsou hojně využívána v modelování a počítačové animaci (srovnání výsledků aplikace různých schémat nabízí Obrázek 3.8). K jednotlivým zmíníme základní literaturu, ve které se dají nalézt podrobnosti o jejich analýze, implementaci atp. V další kapitole se pak budeme věnovat výběru schématu, které by nám pomohlo řešit LOD pro zobrazování terénu.

3.2.1 Catmull-Clark

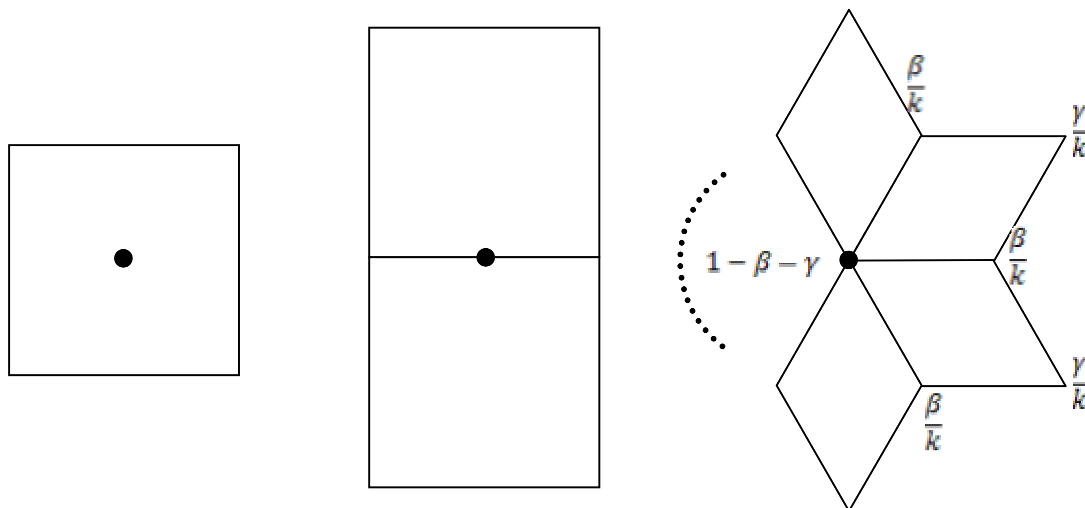
Catmull a Clark společně navrhli čtyřúhelníkové aproximační schéma (69), jehož iterace probíhá ve třech krocích, ve kterých se vytvoří postupně stěnové vrcholy, poté hranové vrcholy a nakonec se změní poloha řídících vrcholů. Stěnovému vrcholu se přiřadí souřadnice těžiště vrcholů tvořících danou stěnu (obecně to nemusí být čtyřúhelník). Hranový vrchol vznikne zprůměrováním sousedních stěnových vrcholů a řídících vrcholů tvořících danou hranu. Stěnové vrcholy se poté spojí s hranovými vrcholy, které odpovídají hranám dané stěny. Konečně novou souřadnici původního vrcholu P stanovíme jako

$$P := \frac{F + 2R + (k-3)P}{k},$$

kde F označuje průměr všech sousedních stěnových vrcholů, R označuje průměr všech hran vycházejících z P a k je stupeň vrcholu, tedy počet sousedních stěn a zároveň počet vycházejících hran. Pravidla můžeme také vyjádřit pomocí schematické masky, jak ukazuje Obrázek 3.2.

Významnou vlastností Catmull-Clarkova schématu je to, že ač je čtyřúhelníkové, vstupní síť nemusí být ani pravidelná, ani složená ze čtyřúhelníků, neboť pravidla jsou sestavena pro obecný počet stěn. To je zároveň důvodem toho, že Catmull-Clarkovo schéma může vytvořit výjimečný vrchol tam, kde dříve nebyl (konkrétně se vytvoří jako stěnový vrchol ve stěně, kterou netvoří

čtyřúhelník), což jiná schémata zpravidla nedělají. Naštěstí se toto může stát pouze při první iteraci, neboť ta z obecné sítě udělá čtyřúhelníkovou a dalšími iteracemi už tedy nové výjimečné vrcholy nevznikají (ale také nezanikají). Této vlastnosti Catmull-Clarkova schématu se hojně využívá tam, kde potřebujeme omezit vstup na čtyřúhelníkovou síť – na obecnou síť použijeme jednu iteraci schématu. Také je dobré zmínit fakt, že násobnou iterací schématu se od sebe výjimečné vrcholy topologicky vzdalují, proto obecná analýza schématu může zkoumat pouze osamocené výjimečné vrcholy, tj. takové, v jejichž okolí se nenachází jiné výjimečné vrcholy.



Obrázek 3.2: Základní masky Catmull-Clark schématu: vlevo maska pro stěnový vrchol, uprostřed maska pro hranový vrchol a vpravo pro výjimečný vrchol. Catmull a Clark doporučují koeficienty $\beta = \frac{3}{2k}$ a $\gamma = \frac{1}{4k}$.

Schéma je C^2 spojité všude kromě konečného počtu výjimečných vrcholů, kde si stále zachovává alespoň C^1 spojitost. Není schopné zachytit ostré hrany a jiné nehladké rysy, proto bylo schéma později rozšířeno o možnost definování ostré hrany užitím jiného pravidla v konkrétním počtu iterací. To však z původně uniformního stacionárního schématu dělá neuniformní a nestacionární, a především složitější na vyhodnocení. Catmull-Clark schéma je proto hojně využíváno při animacích, kde není doba výpočtu kritická. Uniformní sada pravidel byla navržena v (73) jako součást univerzálního postupu na dodání ostrých hran a jiných rysů i pro obecná schémata.

V (74) nalezneme postup, jak použít Catmull-Clarkovo schéma jako schéma interpolační. Postup zahrnuje jednak fázi zjemnění a úpravu vstupní sítě, aby splňovala podmínky na ni kladené dalším postupem. Poté přichází fáze řešení systému rovnic, který definuje podmínky interpolace a který vytvoří modifikovanou síť, na níž bude možné přímo použít schéma tak, aby původní vrcholy byly součástí výsledné sítě. Pro vysokou implementační i výpočetní složitost postupu je často vhodné spíše zvolit přirozeně interpolační schéma, je-li to třeba.

Stam navrhl algoritmus na přímé vyhodnocení limitní plochy schématu ve zvoleném bodě (75). Využívá vyjádření řídících vrcholů pomocí báze vlastních vektorů matice schématu.

Velmi efektivní a dobře paralelizovatelný postup pro výpočet limitní plochy Catmull-Clarkova schématu navrhli Bolz a Schröder (76). Jejich postup je založená na definování bazických funkcí řídících vrcholů, které říkají, jak je který nově vytvářený vrchol ovlivněn tímto řídícím. Takováto funkce lze snadno tabelovat a výsledný postup aplikace se tedy dá provést pomocí pronásobení předem spočítaných tabulek s hodnotami řídících vrcholů. Autoři tento postup následně implementovali i na GPU (77) a v nedávné době byl postup také použit společně s displacement mapou (78) pro dodefinování detailu o vysokých frekvencích (79).

Catmull-Clarkovo schéma bylo také použito jako systém adaptivního dělení za účelem řešení LOD (80). Adaptivní kritéria jsou použita na každý trojúhelník původní sítě zvlášť a zohledňují například zakřivení plochy, viditelnost, velikost projekce na obrazovku či fakt, zda se hrana podílí na siluetě objektu. Postup bohužel zobrazuje každý trojúhelník zvlášť, což, jak bylo již vícekrát uvedeno, není v souladu s efektivním využitím GPU. Jiné využití GPU bylo navrženo v (81), kde se vygeneruje dělící síť pro jeden trojúhelník původního modelu a tato síť se uloží ve formě textury. Textura se zpracovává v pixel shaderu a výstup, tedy nové souřadnice vrcholů, se překopíruje do jiné (větší) textury pro další dělení. Postup se iterativně opakuje, dokud se nedosáhne patřičného dělení. Z výsledné textury se vytvoří vertex buffer, který se použije pro zobrazení vstupního děleného trojúhelníku. Původní práce zmiňuje i výsledky, které byly silně poznamenány faktem, že bylo použito Shader Modelu 2.0, který neposkytoval dostatečné množství instrukcí a neumožňoval číst texturu ve vertex shaderu. Předpokládáme, že s využitím nové technologie by byly výsledky rozdílné o několik řádů, ale existence takové implementace nám není známa.

Konečný provedl intenzivní srovnání 5 různých přístupů k vyhodnocení Catmull-Clark schématu implementovaných na CPU i GPU (82). V práci se bohužel zaměřil pouze na aplikaci schématu na sítě bez výjimečných vrcholů. Toto srovnání obhajuje tím, že i takto zjednodušené srovnání nabízí dostatečný ukazatel pro vytváření závěrů. My se naopak domníváme, že pouze plnou implementaci včetně složitějších pravidel je možno srovnávat, neboť volba pravidel může některé přístupy ovlivnit radikálně, zatímco jiné téměř vůbec (především z důvodu odlišné architektury CPU a GPU).

3.2.2 Loop

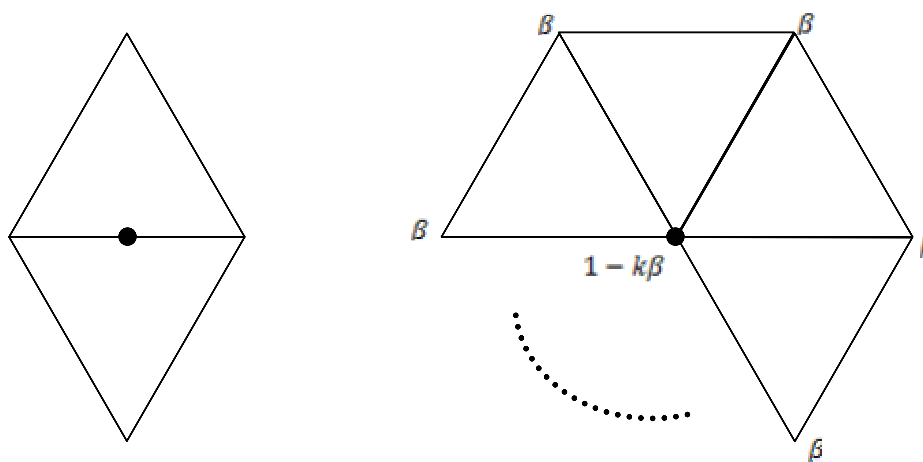
Jiným velmi oblíbeným aproximačním schématem je Loopovo schéma (83). Je to trojúhelníkové schéma utvářející C^2 spojitost skoro všude až na konečný počet výjimečných vrcholů, kde je limitní plocha C^1 spojitá. Protože jeho pravidla neumožňují zachovat v síti ostré hrany, byla sada pravidel později doplněna o několik dalších, která povolují definovat konvexní rohy (vrchol s alespoň 3 ostrými hranami) a zářezy (vrchol s právě jednou ostrou hranou) (84), a o pár let později i konkávní rohy (73). Protože schéma je trojúhelníkové, je aplikovatelné na každou triangulizovatelnou síť po patřičném předzpracování.

Základní podoba schématu má pouze dvě pravidla. První pro vytvoření hranových vrcholů na základě 4 vrcholů tvořících trojúhelníky, které danou hranu obsahují. Druhé pravidlo modifikuje vrcholy původní sítě. Pro novou pozici starých vrcholů dáváme všem sousedním vrcholům

stejnou váhu a původnímu vrcholu přiřazujeme doplněk vah do 1. Loop navrhl přiřadit sousedním vrcholům váhu

$$\beta = \frac{\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{k}\right)^2}{k},$$

kde k je počet sousedních vrcholů, resp. stupeň vrcholu. Masky pro tato pravidla schematicky znázorňuje Obrázek 3.3.



Obrázek 3.3: Masky Loopova schématu: vlevo hranový vrchol, vpravo modifikace původního vrcholu - všechny sousední vrcholy mají stejnou váhu.

Loopovo schéma bylo implementováno do herního grafického systému pro řešení LOD (85). Tato implementace nazvaná VALENTE Subdivision byla postavená na předpočítání mocnin matice schématu a její přímé aplikaci na řídící body. Je to tedy obdoba součtu bazických funkcí, se kterou jsme se setkali u (76), lišící se v zásadě pouze v terminologii. Pro řešení prasklin mezi odlišně dělenými částmi modelu je generována dodatečná geometrie.

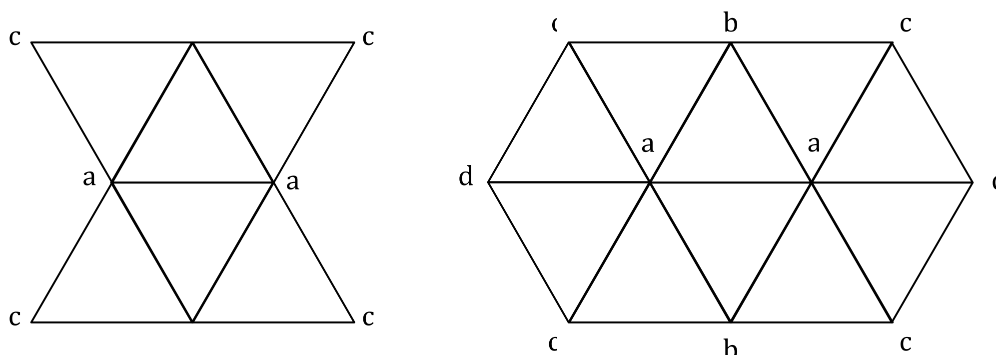
Podobně jako Stam navrhl přímé vyhodnocení Catmull-Clarkova schématu pomocí báze vlastních vektorů (75), Zorin a Kristjansson navrhli postup přímého vyhodnocení Loopova schématu (86). Jejich postup však, jak sami uvádí, je numericky stabilnější díky použití jiné báze vektorů.

Také dříve zmíněná GPU implementace Catmull-Clarka pomocí opakovaného zpracování vrcholů v pixel shaderu (81) byla aplikována i na Loopovo schéma (87) s tím rozdílem, že se sítě pro několik trojúhelníků původního modelu ukládají do jedné textury.

3.2.3 Butterfly

Trojúhelníkové, tentokrát však interpolační, schéma navrhli Dyn, Levin a Gregory (88) a pojmenovali ho butterfly podle toho, jak schematicky vypadá jeho maska (viz Obrázek 3.4).

Schéma je pouze C^1 spojité a v některých výjimečných vrcholech dokonce pouze C^0. Proto byla později navržena jeho modifikace, která se dnes používá a která zaručuje C^1 spojitost všude (viz Obrázek 3.5). Pro základní schéma existuje jediné pravidlo, které definuje pozici nového vrcholu vytvořeného na hraně. Protože schéma je interpolační, s původními vrcholy se nehýbe. Nově vytvořené hranové vrcholy společné jednomu trojúhelníku se navzájem spojí a rozdělí tak původní trojúhelník na 4 nové.



Obrázek 3.4: Maska původního Butterfly schématu definuje pouze hranový vrchol pro regulární síť. Vlevo vidíme variantu pro 8 řídících bodů, vpravo složitější variantu s 10 vrcholy.

Autoři původního butterfly schématu navrhli jednotlivým vrcholům váhy $a = \frac{1}{2}$, $b = \frac{1}{8} + 2w$ a $c = -\frac{1}{16} - w$, kde w označují jako parametr napětí, kterým je možné ovlivnit, jak těsně je limitní plocha svázána s řídícími vrcholy (označení reflektuje Obrázek 3.4). Později lehce modifikovali schéma přidáním dvou dalších vrcholů do jeho masky. Pro nově zavedené schéma se změnila pouze váha $a = \frac{1}{2} - w$ nejbližších vrcholů a nově přidaným vrcholům se nastavila váha $d = w$. Vidíme, že pokud parametr napětí zvolíme nulový, dostáváme to samé schéma jako v případě osmibodové masky.

Protože původní maska nezohledňovala výjimečné vrcholy, bylo pro ně o pár let později přidáno zvláštní pravidlo (89). Jeho masku zachycuje Obrázek 3.5, na kterém vidíme váhy s_i přiřazené jednotlivým vrcholům sousedícím s výjimečným vrcholem. Tyto váhy byly definovány zvlášť pro různé stupně k výjimečného vrcholu v :

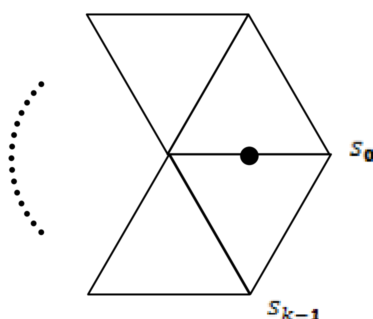
$$k = 3:\ v = \frac{3}{4}, s_0 = \frac{5}{12}, s_1 = s_2 = -\frac{1}{12},$$

$$k = 4:\ v = \frac{3}{4}, s_0 = \frac{3}{8}, s_1 = 0, s_2 = -\frac{1}{8}, s_3 = 0,$$

$$k \geq 5:\ v = \frac{3}{4}, s_i = \frac{1}{k}\left(\frac{1}{4} + \cos\frac{2i\pi}{k} + \frac{1}{2}\cos\frac{4i\pi}{k}\right).$$

Pokud jsou oba dva vrcholy dané hrany výjimečné, spočítají se polohy pro každý zvlášť a tyto polohy se poté zprůměrují.

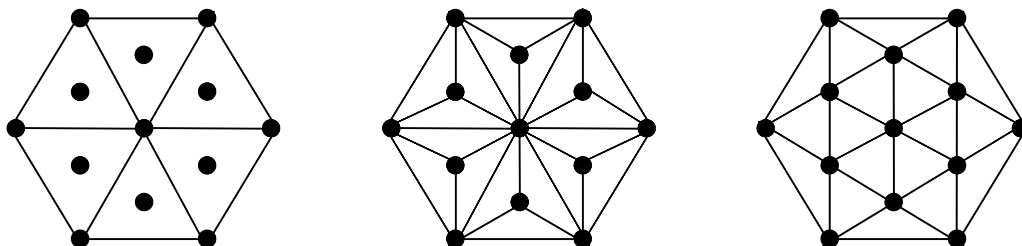
Implementaci butterfly schématu a mnoho poznámek můžeme najít například v (90). Jiný popis ryze paralelního přístupu nalezneme v (91), kde jsou navrhnuty postupy pro statický i dynamický paralelismus. Ačkoli paralelní zpracování je v dnešní době stále více aktuální, pokud chceme implementovat subdivision surfaces na GPU, nemá pro nás explicitní řešení paralelismu valný význam, neboť GPU je svou architekturou veskrze paralelní stroj a složité části této úlohy řeší za nás. V práci je také rozebrán postup dělení vstupní sítě na podoblasti, které se zpracovávají zvlášť. Bohužel navrhované řešení při spojování těchto oblastí nezachovává C^1 spojitost, neboť při výpočtu hranic nepoužívá informace ze sousedních oblastí.



Obrázek 3.5: Maska modifikovaného Butterfly schématu pro hranu s výjimečným vrcholem.

3.2.4 Jiná schémata

Schémat a jejich modifikací již byla navržena celá řada. Ty nejčastěji používané jsme již zmínili. Srovnání jejich výstupu ukazuje Obrázek 3.8. Zde chceme ještě vzpomenout dvě méně obvyklá, ale neméně zajímavá, a někam je zařadit společně s ostatními.

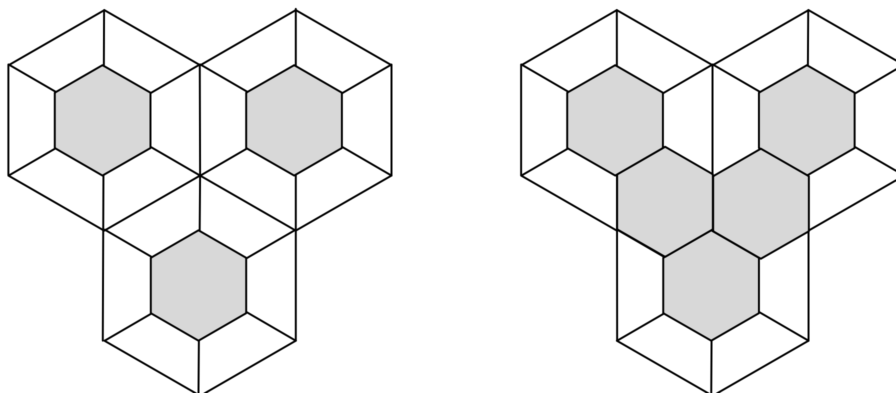


Obrázek 3.6: Kobbeltovo schéma $\sqrt{3}$. Vlevo původní síť s nově přidávanými vrcholy. Uprostřed jsou přidané vrcholy spojeny hranami s původními vrcholy. Vpravo vidíme výsledek druhého kroku - vznikla opět pravidelná trojúhelníková síť, avšak smrštěná s koeficientem $\sqrt{3}$ a otočená o 30°.

Kobbelt navrhl trojúhelníkové aproximační schéma, které je C^2 spojité na regulárních sítích (92). Zatímco ostatní trojúhelníková schémata zpravidla rozdělují trojúhelník na čtyři menší za pomoci dělení hrany, a tím zčtyřnásobí počet trojúhelníků původní sítě, Kobbeltovo schéma tento počet pouze ztrojnásobí. Funguje dvoukrokově. V prvním kroku se do středů stěn vloží nové vrcholy, které se spojí s vrcholy trojúhelníku, kterému patří. Tím vznikne požadované množství primitiv. V druhém kroku se odstraní původní hrany a vzniklé čtyřúhelníky se rozdělí opačnou diagonálou (tzv. záměna hrany). Lépe to popisuje Obrázek 3.6.

Toto zvláštní schéma má velkou výhodu v pomalejší konvergenci. To se hodí například v případě, že potřebujeme odlišit více úrovní dělení, než překročíme nějakou mez povoleného počtu vrcholů. Protože hrana nově vzniklého trojúhelníku má $\sqrt{3}$ délky hrany původního, bylo toto schéma Kobbeltem pojmenováno $\sqrt{3}$.

Druhým zajímavým schématem je hexagonální schéma, ale najdeme jej také pod názvem honeycomb (včelí plástev). Bylo navrženo Dynem, Levinem a Liu (první dva se podíleli na butterfly schématu) jako nestacionární dělení. Podobně jako předchozí uvedené je tvořeno dvěma kroky. První krok umístí do každého šestiúhelníka 6 nových vrcholů, které vytvoří nový poloviční šestiúhelník a 6 čtyřúhelníků na hranicích. V druhém kroku se sloučí sousední čtyřúhelníky dvou různých původních stěn tím, že se odebere jejich společná hrana. Tímto sloučením vznikne nový šestiúhelník. Popsaný postup názorně ukazuje Obrázek 3.7.



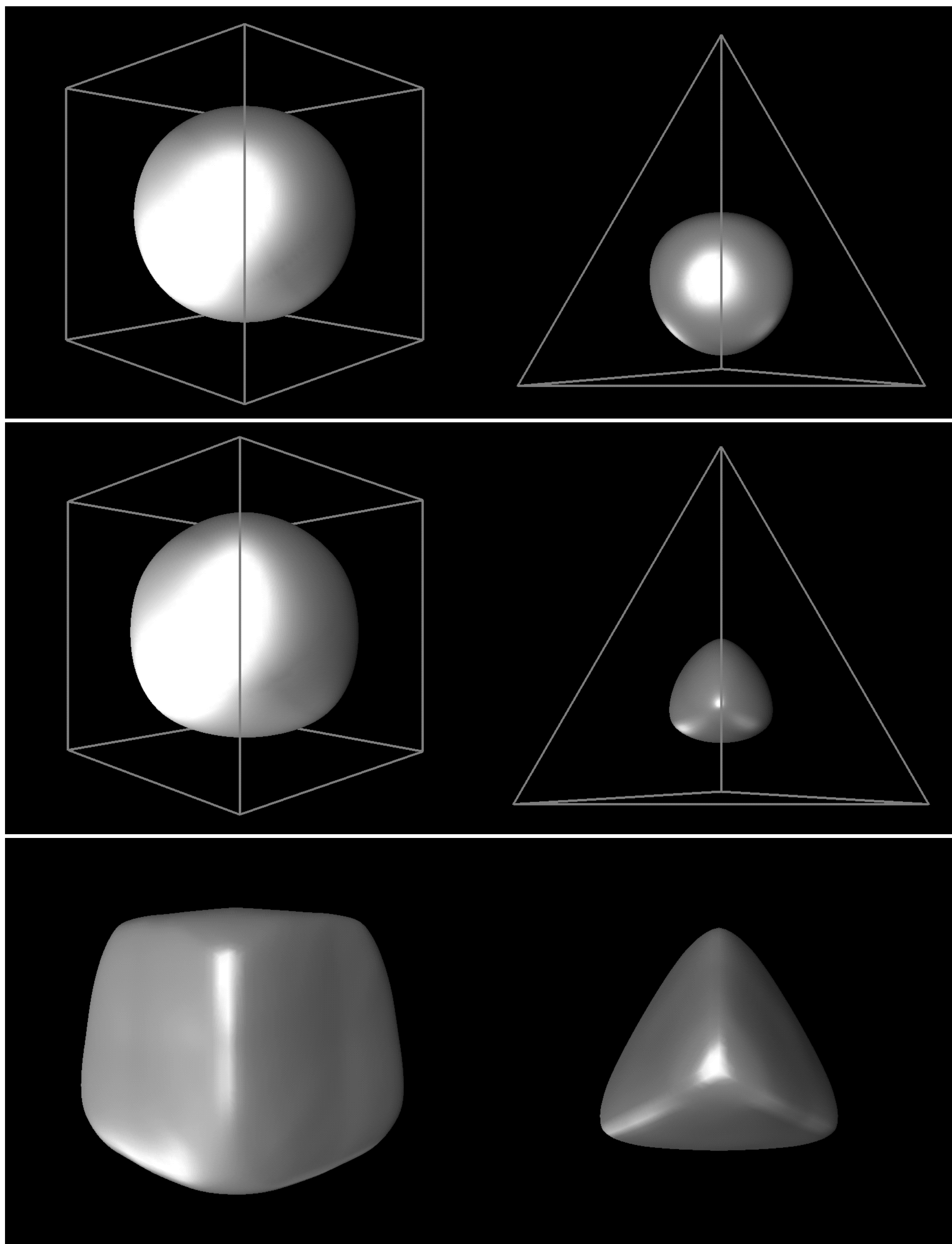
Obrázek 3.7: Dělení pomocí hexagonálního schématu. V každém původním šestiúhelníku se vytvoří jeden nový poloviční (vlevo) a okrajové oblasti se následně spojí a vytvoří stejné šestiúhelníky (vpravo).

Novým vrcholům je přiřazena souřadnice zprůměrováním středu stěny s odpovídajícím původním vrcholem. Přesněji

$$\forall i \in 0, \dots, 5: P_i^{j+1} := \frac{1}{2}\left(P_i^j + \frac{1}{6}\sum_{k=0}^{5} P_k^j\right),$$

kde P_i^j značíme i -tý vrchol šestiúhelníka při j -tém dělení.

Pro nás je opět zajímavé, že toto schéma bylo použito na adaptivní řešení LODu při zobrazování terénu v (93). Bohužel zmíněná implementace byla opět velmi neefektivní vzhledem k využití GPU (každý hexagon byl zobrazován zvlášť pomocí jednoho vějíře trojúhelníků) a její výsledky tedy nejsou přesvědčivé.



Obrázek 3.8: Srovnání výsledků aplikace základních schémat na krychli (vlevo) a čtyřstěn (vpravo). Použitá schémata jsou v pořadí odshora Catmull-Clark, Loop a modifikovaný butterfly. Obrázky převzaty z (72).

4 Naše řešení

V úvodu jsme rozebrali cíle, kterých chceme dosáhnout při návrhu řešení zobrazování terénu. V dalších kapitolách jsme se věnovali podrobnějšímu rozboru problému a popisu používaných technik. Nyní se pokusíme navrhnout postup, který bude splňovat hlavní vytyčené cíle a zohledňovat některé vedlejší. V průběhu vývoje jsme se potýkali s řadou problémů u různých postupů a řešili jsme nedostatky parciálních i celkového řešení. Některé z těchto problémů zde rozebereme společně s postupy, které jsme vyzkoušeli pro jejich odstranění, aby bylo možné na tuto práci dále navázat bez nutnosti opětovného průchodu každé slepé cesty. Řešení, které v pozdějších odstavcích navrhujeme, není finální a řeší pouze některé aspekty zobrazování terénu s ohledem na naše definované cíle. Mnohá další související témata ponecháváme na budoucí práci. Návrhy jak některé oblasti řešit, jak dále postupovat či co vylepšit uvádíme v závěrečném odstavci.

4.1 Iniciální návrh řešení

Připomeňme si nyní cíle, které jsme si stanovili:

1. Zpracování potenciálně nekonečného terénu.
2. Umožnění modifikací terénu za běhu aplikace.
3. Snadná škálovatelnost hardwarové náročnosti při startu aplikace.

Při návrhu řešení se zaměříme na první dva zmíněné, neboť jsou rozhodující pro volbu datové struktury a hlavní linie postupu. Po provedení volby se změříme na to, abychom dosáhli snadné škálovatelnosti.

4.1.1 Volba hlavního přístupu

V paragrafu 2.3 jsme prezentovali mnoho různých algoritmů na zobrazování terénu. Nejjednodušší by bylo zvolit některý z nich a implementovat jej podle dostupných materiálů. Protože chceme navrhnout postup, který bude co nejméně náročný na procesor (v souladu s užitím výsledné techniky ve hrách, kde má CPU mnoho dalších úkolů kromě zobrazování), jsme nuceni rovnou zavrhnout jakékoli varianty progressive meshes a většinu algoritmů založených na omezeném kvadrantovém stromu nebo BTT, které každý snímek dynamicky aktualizují seznam trojúhelníků, nebo dokonce generují triangulaci od základu. V úvahu tedy přichází některé hybridní metody, které využívají statických předem generovaných bloků, dále různé metody založené na adaptivním dělení pomocí parametrických ploch či jiných schémat a konečně clipmapy.

Nyní je třeba přidat do úvah požadavek na modifikovatelnost terénu. Žádná z implementací zmíněných algoritmů, se kterou jsme se setkali, nebyla prováděna nad neomezenými daty, která je možné modifikovat za běhu. Metody závislé na procesu předzpracování dat jsou k tomuto

účelu velmi nevhodné (což neznamená, že by vůbec nešly použít), a to tím více, čím náročnější je předzpracující proces. Teoreticky snadná modifikace se nabízí pro clipmapy, které mají data uchována přímo ve výškové mapě a nepoužívají žádnou explicitní metriku závislou na datech. Protože však uchovávají data ve vysokém detailu jenom v malém okolí kolem pozorovatele, bylo by řešení změn komplikované (nebo by vyžadovalo zápis na disk). Poměrně málo prozkoumaná oblast využití různých typů ploch, které je možné adaptivně dělit, se nabízí jako jeden z možných směrů – aplikace LOD je intuitivní, prostor řešení veliký a ne zcela prozkoumaný. Jinou možností by byla vhodná úprava efektivního algoritmu, například clipmap.

Naše další rozhodování bylo ovlivněno dvěma faktory. Tím prvním byla snaha nalézt nové řešení, které by samo o sobě nabízelo širší pole možností. Druhým významným faktorem byl příchod Shader Modelu 4.0. Ten přinesl mnoho důležitých změn, z nichž pro nás nejzajímavější je přidání geometry shaderu, nového stupně zobrazovacího procesu, a s tím spojené nové možnosti výstupu dat ze zobrazovací jednotky. Řada i zde zmíněných algoritmů využívá pixel shaderu k definování geometrie. Ta musí být uložena do textury ve formě pixelů a teprve na CPU překopírována do vertex bufferu, který se použije při zobrazování. Tento postup je nejen komplikovaný, ale také silně narušuje asynchronní chod CPU a GPU. Shader Model 4.0 definuje možnost výstupu vrcholů již v prvních fázích zobrazovacího procesu, tedy po provedení vertex shaderu nebo nově přidaného geometry shaderu. Geometry shader pak nejen přináší možnost pracovat nad celým trojúhelníkem najednou, ale především jako jediný stupeň zobrazovacího procesu umožňuje měnit počet výstupních primitiv. Proto zatímco z vertex shaderu vystoupí právě jeden vrchol pro každý vrchol, který do něj vstoupil, z geometry shaderu může vystoupit různý počet trojúhelníků podle toho, jak daný program rozhodne.

Silně nás zaujala možnost generování dat přímo na grafické kartě. Zdá se být přímo přirozené využít tuto novou schopnost pro adaptivní zjemňování a utváření geometrie na grafické kartě, bez opakovaných smyček přes CPU a kopírování dat do hlavní paměti. Proto jsme se uchýlili ke zkoumání možností zapojení subdivision surfaces do zobrazování terénu za intenzivního využití moderních grafických procesorů.

4.1.2 Volba datové reprezentace

Znovu se vrátíme k prvním dvěma definovaným cílům – neomezený terén a modifikovatelnost. Neomezeným terénem sice nechápeme nekonečný objem dat, ale obecně takový, který nemůžeme načíst do operační paměti. To znamená, že bude nutné navrhnout a implementovat systém, který bude data za běhu získávat z nějakého média. Datová průchodnost média přitom může být limitující. Volba datové reprezentace by toto měla reflektovat. Přitom paměťové nároky i implementační složitost jsou další významná kritéria.

Ideální řešení by zahrnovalo možnost progresivního načítání dat. Pro hry je velmi důležitá plynulost běhu. Není přijatelné, aby se při pomalém načítání dat zastavil celý chod. Umožněním progresivního načítání se dá zajistit plynulý běh tak, že využíváme data nižších úrovní detailu, pokud nejsou ta detailní ještě k dispozici. To samozřejmě předpokládá, že alespoň nějaká základní data umíme zajistit vždy (například clipmapy mají za všech okolností uchován celý terén v podobě nejméně detailní úrovně). Zároveň by však rekonstrukce dat neměla klást velké

nároky na CPU (ideálně žádné) – v případě dekomprese spojené s rekonstrukcí či syntézou dat na procesoru to mohou být citelné ztráty výkonu.

Zamýšleli jsme se nad implementační složitostí, která pro takový systém může být vysoká. Souvisí to s tím, jak jsou data členěna, aby bylo možné definovat požadavek na jejich zajištění. V tomto směru by bylo nejsnazší rozdělit terén pomocí pravidelné mřížky a pracovat s oblastmi dělení jako s atomickou jednotkou. Z tohoto pohledu je nevhodné použít strukturu jako je TIN, pokud ji patřičným způsobem neomezíme.

Konečně přišla do úvahy i snadná modifikovatelnost dat, a to jak z pohledu implementačního, tak především z pohledu systémových nároků. Přitom je třeba brát v úvahu i povahu média. Jestliže se kupříkladu jedná o médium bez možnosti zápisu, musíme být schopni všechny změny provedené na datech terénu reprodukovat, resp. zobrazit, alespoň po nějaký omezený čas, dokud se pozorovatel pohybuje v jejich blízkosti (prostorové i časové – to velmi záleží na rychlosti pohybu atd.).

Všechny úvahy byly společně srovnány a jako nejlepší možnost byla nakonec vybrána dvouúrovňová výšková mapa. Princip této reprezentace spočívá v uložení dat v podobě velmi hrubé výškové mapy o nízkém rozlišení, ale vysoké přesnosti uchovaných hodnot, a druhé, tzv. *mapy reziduí*, která je v řádově větším rozlišení, ale její hodnoty jsou daleko nižšího rozsahu. Tyto mapy se použijí společně pro získání skutečné výšky terénu na základě hrubé aproximace výškovou mapou a dodefinováním detailu mapou reziduí. Takto navržená datová reprezentace má několik dobrých vlastností s ohledem na předešlá kritéria:

1. Data jsou snadno modifikovatelná.
2. Rekonstrukce dat není složitá a je snadné ji provést na grafické kartě (žádná zátěž CPU).
3. Obě mapy lze triviálně rozdělit pravoúhlou mřížkou na menší oblasti pro snadné načítání.
4. Datová průchodnost zdrojového média není příliš limitujícím faktorem.

Výšková mapa je velmi hrubá a je možné ji uchovat v paměti celou, případně načítat jen velmi zřídka. Z toho vyvozujeme předpoklad, že tato data máme vždy k dispozici a nejsme tedy závislí na okamžitém doručení dat reziduí. Mapa reziduí by měla obsahovat hodnoty jen velmi malého rozsahu, a tedy objem dat pro aktualizaci by opět neměl být velký. To již však závisí na rychlosti pohybu pozorovatele.

Více limitujícím faktorem je zde paměť. Ačkoli užitím této dvouúrovňového struktury máme daleko nižší nároky na paměť, než kdybychom definovali výškovou mapu v detailu mapy reziduí, klademe výrazně vyšší nároky na paměť než metoda clipmap. Je to dáno tím, že uchováváme v paměti vždy celou oblast, která je právě viditelná. Je zřejmé, že ne celá viditelná oblast vyžaduje tak detailní záznam, jaký poskytuje kombinace výškové mapy a mapy reziduí, ale právě tento postup nám umožní snadnou modifikovatelnost terénu, ve které tradiční clipmapy selhávají. Kromě toho to odbourává již dříve zmíněný velký nedostatek clipmap, které špatně reagují na náhlou změnu velikosti zorného úhlu pozorovatele, při které je třeba okamžité změny přesnosti

dat ve viditelné oblasti (naše dvouúrovňová struktura má všechna potřebná data stále k dispozici).

Zmínili jsme se o snadné modifikaci. V tomto směru je třeba doplnit, že ačkoli to není nutné, rozhodli jsme se omezit modifikovatelnost terénu na modifikaci mapy reziduí, která neumožňuje úplnou svobodu, neboť data v ní uložená jsou velmi malého rozsahu. Rozsah velmi záleží na naší volbě a dalších potřebách systému, ve kterém se struktura použije. K tomu se dostaneme ještě v dalších částech, zde můžeme pouze předeslat, že pracujeme zpravidla s rozmezím v řádu desítek metrů, což se nám zdá být dostatečný rozsah pro většinu potřeb.

Rekonstrukce dat je velice triviální, neboť na rozdíl od hierarchických struktur bez omezení počtu úrovní, naše reprezentace poskytuje konstantní dobu výpočtu výšky ve zvoleném bodě. Tato je navíc snadno implementovatelná na GPU.

4.2 Adaptivní dělení terénu

Zvolili jsme základní podobu datové reprezentace (přesná specifika můžeme řešit později) a hlavní směr, kterým se chceme ubírat, tedy adaptivní dělení terénu prováděné pomocí GPU. Nyní nás čeká úkol provést několik dalších dílčích rozhodnutí, kterými dotvoříme celkovou myšlenku:

1. Na základě jakého schématu dělení provádět?
2. Na jaké základní části dělení aplikovat?
3. Jakým způsobem na sebe napojit odlišně dělené části?
4. Kde a jak rozhodovat o hloubce dělení?

Přestože řada algoritmů, z nichž mnohé jsme zmínili dříve, provádí adaptivní dělení terénu s ohledem na GPU, všechny provádí rozhodovací proces volby LODu pro jednotlivé oblasti terénu na CPU. Utilizace grafické karty pak probíhá pro každou oblast zvlášť. Pokud je terén na oblasti dělen uniformně, znamená to zpravidla velké množství takových oblastí (řádově stovky až tisíce) a s tím související „small batch problem" popsaný dříve. Tímto problémem netrpí kupříkladu clipmapy, neboť se zobrazují po úrovních, resp. částech úrovní, a celkový počet volání některé ze zobrazovacích funkcí se tak pohybuje v řádu desítek (57). S takovýmto výsledkem bychom mohli být spokojeni, ale znamená to, že bychom museli dělit terén hierarchicky, nebo na velké bloky. Hierarchie by zřejmě přinášela nevítané komplikace při napojování bloků, proto pro nás lepší volbu představují uniformní bloky.

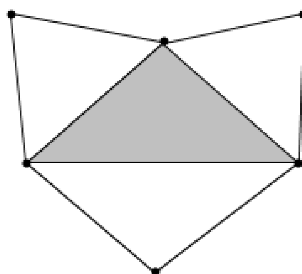
Řekli jsme si také, že chceme využít možnosti generování geometrie a ideálně i rozhodování o volbě LODu na GPU. Hlubším promyšlením takového přístupu dojdeme k tomu, že pokud řešíme úroveň dělení na GPU, musíme řešit i napojování odlišně dělených oblastí na GPU. To je v souladu s možností generování nové geometrie, takže když to shrneme, naskýtá se nám možnost přesunout úplně celou logiku zobrazování terénu na GPU. To navíc znamená, že můžeme teoreticky dospět k řešení, které bude zobrazovat celý terén najednou, tj. pomocí jediného volání některé ze zobrazovacích funkcí. Celý terén se v takovém případě vytvoří na grafické kartě a

v případě statických dat by jedinou prací CPU bylo zavolání jedné API funkce každý snímek. Domníváme se, že pokud se nám takové řešení podaří najít, a bude výkonnostně srovnatelné s jinými používanými, mohlo by výrazně ovlivnit budoucí zobrazovací algoritmy a kvalitu zobrazovaných terénů.

4.2.1 Úvahy pro volbu dělícího schématu

Udělali jsme si přesnější představu o tom, čeho dosáhnout i jak toho dosáhnout. Čeká nás problém nalezení vhodného dělícího schématu, které bude snadno vyhodnotitelné, přirozeně adaptivní, a především bude nabízet dobré vizuální výsledky. Jsme přesvědčeni, že všechny tyto parametry může nabídnout některé z používaných subdivision surfaces, nebo jeho patřičná modifikace. Předně si zdůrazněme, že při výběru takového schématu stavíme vizuální výsledky před pěkné matematické vlastnosti, tj. nemusíme kupříkladu nutně hledat schéma, které bude C^2 spojité, často můžeme spoléhat i na schopnosti designéra, který případné nedostatky výstupu zakryje.

První rozhodnutí (později změněna) byla založena čistě na racionální úvaze o tom, jaké existující schéma má správné vlastnosti: chceme hledat pouze trojúhelníková schémata, protože jsou to jediné mnohoúhelníky, se kterými můžeme pracovat na grafické kartě. Dále se zřejmě budeme chtít omezit na interpolační schémata, neboť potřebujeme, aby zobrazovaný terén více držel tvar se vstupními daty. Především by bylo nevhodné, pokud by se v jednotlivých iteracích měnila celková podoba terénu – vizuálně by se to dalo řešit morfováním, ale výpočet fyzikálních interakcí objektů s terénem by pak nemusel být korektní a konzistentní se zobrazováním.



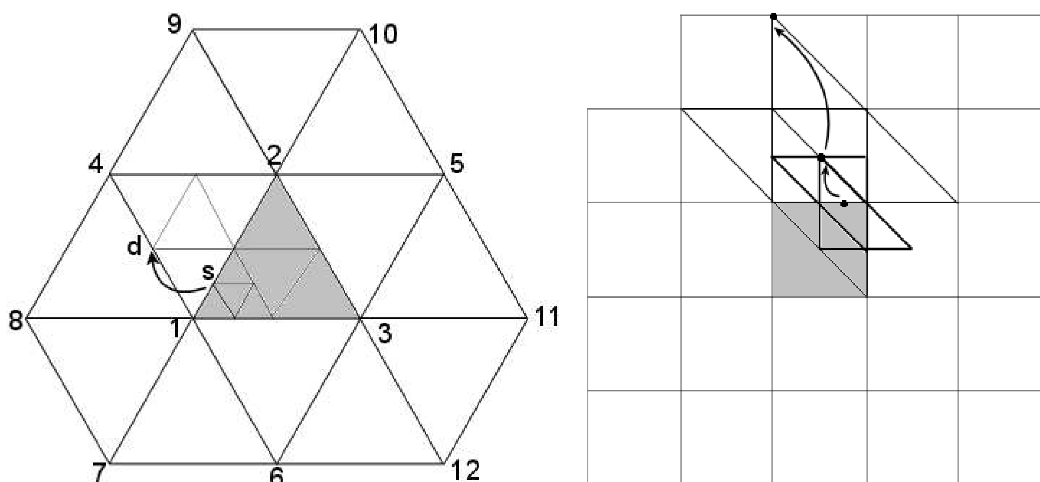
Obrázek 4.1: V geometry shaderu máme k dispozici informace o zpracovávaném trojúhelníku (zvýrazněný) a 3 dalších vrcholech sousedních trojúhelníků.

Dalším, zřejmě nejkritičtějším požadavkem na schéma je co nejmenší velikost jeho nosiče. V geometry shaderu nemáme k dispozici informace ani pro celé 1-okolí zpracovávaného trojúhelníku. Konkrétně je dostupná informace o 3 vrcholech trojúhelníku a dalších 3 vrcholech sousedních trojúhelníků, jak ukazuje Obrázek 4.1. To je nedostatečné pro jakékoli běžně používané schéma. Obejít to můžeme hned několika způsoby. První možnost je přímo do vrcholů přidat nějakou informaci o jejich okolí. Tím by mohla značně narůst velikost dat, která, jak si záhy ukážeme, přímo ovlivňuje rychlost algoritmu. Druhou možností je uložení přídavných informací do textury a adresovat texturu za pomoci znalosti indexu primitiva (to je opět funkcionalita přidaná nově do Shader Modelu 4.0). Toto řešení je možné, ale jeho nedostatkem je velmi intenzivní využití texturovacích jednotech. S tím se pojí problém toho, že už samotná rekonstrukce dat je silně závislá na čtení dat z textur. I přesto, že výpočet subdivision surface

může obnášet velké množství výpočtů, těžko by se poměr aritmetických a texturovacích operací přiblížil aktuálně doporučovanému poměru 4:1 (94) a celé zobrazování by bylo závislé na dostatečně rychlém přísunu dat. V tomto směru by řešení založené na uložení pomocných hodnot společně s vrcholy mohlo vést k lepším výsledkům, neboť texturovací jednotky a data vertex bufferů jsou oddělené datové zdroje a je obecným doporučení rozložit tíhu na oba.

My jsme dospěli k názoru, že ani jedno z navržených řešení by neposkytovalo dostatečné výsledky[18], a proto, v souladu s dříve navrženým postupem, jsme se uchýlili k jednoprůchodovému řešení, které všechna data pro jeden dělený blok spočítá najednou bez nutnosti opakovaného čtení a ukládání pomocných informací. Toto řešení je navíc podporováno faktem, že dodatečné informace jsou společné vždy pro mnoho počítaných vrcholů, proto jejich (násobná) duplikace nebo opakovaná čtení logicky musí vést ke snížení efektivity. Vyvodili jsme z toho, že data je třeba dodat všechna najednou, ať už ve formě textury nebo bufferu, a jejich užití provést jednorázově pro daný blok terénu. I v tomto případě však chceme, aby potřebná data, tj. nosič schématu, byla co nejmenší.

Běžným požadavkem, který na schémata kladou jiné partie počítačové grafiky, je definování chování schématu v okolí výjimečného vrcholu. To je velmi omezující krok, který často silně zesložiťuje použití schématu. My se tohoto požadavku zříkáme, neboť použitím adaptace výškové mapy můžeme přirozeně počítat s regulární sítí, ať už čtyřúhelníkovou (přímý přepis mřížky), tak i trojúhelníkovou (triviální triangulace po diagonálách polí). Stejně můžeme ignorovat i zvláštní ošetřování okrajů mapy, protože v praxi pozorovateli vždy znemožníme přístup do blízkosti okrajů, kde by viděl omezenost prostředí. Jediné rozšíření, kterému dáváme váhu, jsou pravidla pro přidání ostrých hran.



Obrázek 4.2: Nosiče pro Loopovo (vlevo) a butterfly (vpravo) schéma. Vybarvená plocha znázroňuje zpracovávaný polygon. Vrchol s vzniklý dvojnásobným dělením Loopova schématu je

[18] Žádné konkrétní experimenty nebyly provedeny, opíráme se pouze o vlastní zkušenost. Výsledky z takto postavených přístupů by jistě mohly být zajímavé a můžeme je pouze doporučit pro další pole zkoumání.

závislý na vrcholu d , který není součástí původní sítě. U butterfly schématu máme opět závislost vrcholů při dvojnásobném dělení znázorněnu šipkami. Tučnější čára reprezentuje masku schématu pro první a druhé dělení.

4.2.2 Experimentování se subdivision surfaces

Z trojúhelníkových schémat se přirozeně nabízí Loopovo schéma a butterfly. První zmíněné je aproximační, tedy v rozporu s tím, co jsme hledali. Druhé – butterfly – je interpolační, ale má příliš velký nosič (srovnej schémata, jež znázorňuje Obrázek 4.2). Protože žádné známé schéma se nám nezdálo bez úprav vhodnější, začali jsme s těmito schématy a provedli s nimi celou řadu experimentů. Později jsme se pokusili přejít i na čtyřúhelníková schémata, nalézt vlastní schéma, a konečně ozkoušet i parametrické plochy. Dlouhý proces experimentů zde shrneme alespoň ve zkratce, abychom pomohli případným pokračovatelům v rozhodování.

První důležitý poznatek spočívá v omezené potřebě dat. Podívejme se na Obrázek 4.2 vlevo. Tvrdíme, že k vyjádření vrcholu s , jež vznikl dvojnásobným použitím Loopova schématu, potřebuje přímo pouze informace o vrcholech 1-12 původní mřížky. Naše tvrzení je založeno na jednoduché matematice. Nejdříve se pomocí Loopova schématu pokusme vyjádřit vrchol s . Z pravidla pro dělení hrany plyne

$$s = \frac{1}{8}(d + v_{1-3}) + \frac{3}{8}(v_1 + v_{1-2}),$$

kde v_i označuje vrchol i původní mřížky a v_{i-j} označuje vrchol vzniklý dělením hrany $[(v]_i, v_j)$. Jak vidno, vrchol s je závislý na vrcholu d , v_{1-3} a v_{1-2}, které nejsou součástí původních dat. Pokud však i tyto vyjádříme pomocí pravidla Loopova schématu jako

$$v_{1-2} = \frac{1}{8}(v_4 + v_3) + \frac{3}{8}(v_1 + v_2),$$

$$v_{1-3} = \frac{1}{8}(v_2 + v_6) + \frac{3}{8}(v_1 + v_3),$$

$$d = \frac{1}{8}(v_2 + v_8) + \frac{3}{8}(v_1 + v_4),$$

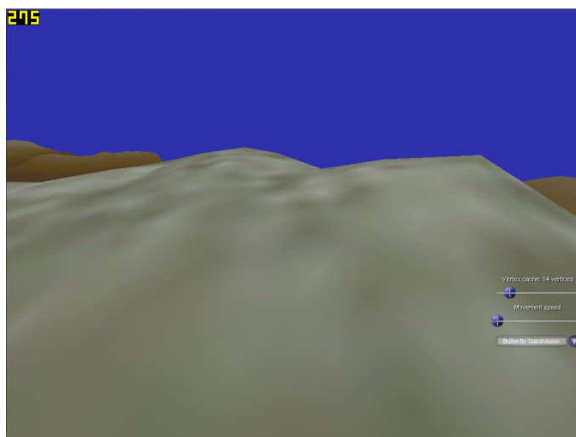
můžeme rozvinout také vztah pro vrchol s a vyjádřit jej pouze na základě řídících vrcholů, tedy

$$s = \frac{1}{8}\left(\frac{1}{8}(v_2 + v_8) + \frac{3}{8}(v_1 + v_4) + \frac{1}{8}(v_2 + v_6) + \frac{3}{8}(v_1 + v_3)\right) + \frac{3}{8}\left(v_1 + \frac{1}{8}(v_4 + v_3) + \frac{3}{8}(v_1 + v_2)\right) = \frac{39}{64}v_1 + \frac{11}{64}v_2 + \dots$$

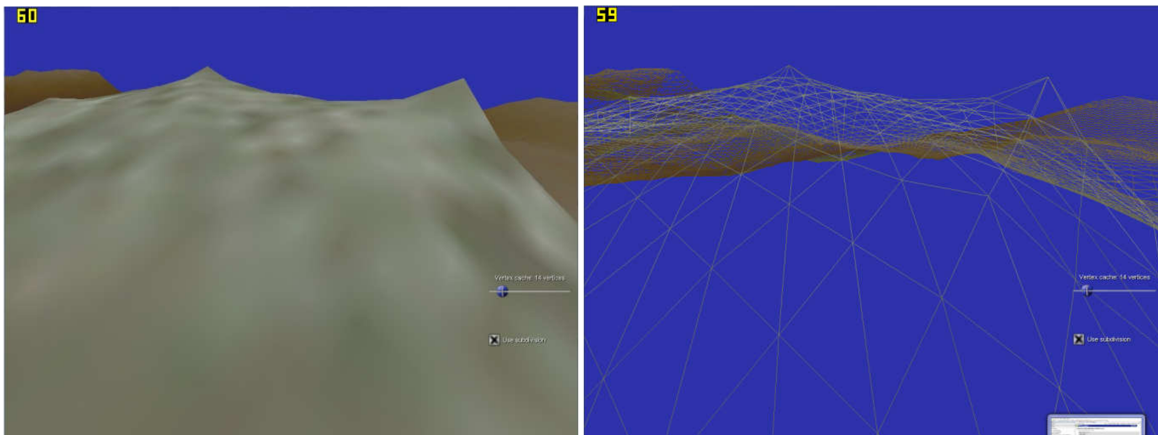
Nebudeme zde provádět rigorózní důkaz, ale platí, že stejným způsobem můžeme vyjádřit všechny vrcholy různých stupňů dělení zvoleného trojúhelníka. Obdobný postup můžeme

aplikovat samozřejmě i na jiná schémata, ale pro různá schémata získáme jinak velký nosič, tj. jiný počet řídících vrcholů. Pro butterfly schéma toto ukazuje Obrázek 4.2, na kterém je vidět, že nosičem je téměř celá mřížka 6×6 okolních vrcholů, kromě dvou rohových. Pokud bychom se omezili pouze na jeden ze zvýrazněných trojúhelníků (konečně butterfly schéma je trojúhelníkové), byla by to stále mřížka 6×6, pouze by v ní chybělo více vrcholů. Pro přílišnou velikost nosiče butterfly schématu jsme se rozhodli jej nepoužít, přestože po prvních dvou úrovních dělení podával vizuálně velmi slibné výsledky (srov. Obrázek 4.3 a Obrázek 4.4).

Po butterfly schématu jsme se uchýlili k Loopovu schématu, které má menší nosič. Obrázek 4.2 nám jej ukazuje v trojúhelníkové síti, kde představuje mřížku 4×4 bez 3 vrcholů (mřížka na obrázku je posunutá, aby tvořila rovnostranné trojúhelníky). Pokud bychom zamýšleli zpracování dvou trojúhelníků společně jako v případě butterfly, tvořila by maska mřížku 4×4 bez dvou rohových vrcholů. Loopovo schéma je původně aproximační, a to jsme již dříve zavrhli, proto jsme se pokusili o triviální modifikaci, která ze schématu dělá interpolační. Jednoduše jsme vynechali pravidlo pro řídící vrcholy a ponechali pouze vytvoření hranového vrcholu. Tím zůstávají řídící vrcholy na svém místě, jak jsme požadovali, avšak vyvstává problém související s tím, že Loopovo schéma geometrii „smršťuje“. Takže zatímco síť jako celek se s každým dělením přibližuje jakési centrální aproximující hladině, pravidelně rozmístěné původní vrcholy vytváří nepěkné rohy na jinak hladkém terénu. Popsaný stav dobře zachycuje Obrázek 4.4. Řešit tento stav je komplikované, neboť charakter smršťování je vlastní všem aproximačním schématům, která mají vlastnost uchování výsledku v konvexní obálce řídících vrcholů. Pro přirozeně interpolační schémata se objem objektu spíše zvětšuje, protože plocha musí procházet řídícími vrcholy, při čem je těžké dosáhnout hladkého povrchu a platnosti konvexní obálky.



Obrázek 4.3: Výsledek prvních dvou dělení butterfly schématem. Dvě vyvýšeniny v popředí představují řídící vrcholy.



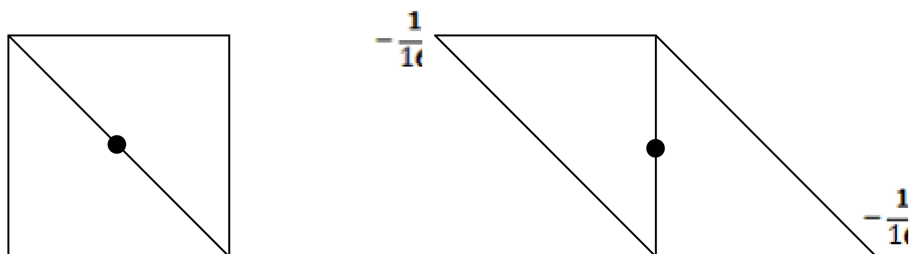
Obrázek 4.4: Výsledek prvních tří dělení modifikovaným Loopovým schématem. Dva vyčnívající hroty představující řídící vrcholy, které zůstaly na svých místech v důsledku převedení schématu na interpolující. Zbytek terénu se „smršťuje" k limitní ploše.

Udělali jsme proto ještě krátké testy na jiných dělících schématech a s podobně neuspokojivými výsledky jsme se jali hledat nové schéma, které by nám vyhovovalo. Předně jsme chtěli udržet velikost nosiče takovou, aby se řídící vrcholy vešly do mřížky 4×4, jakou používá Loopovo schéma. Od něj jsme tedy vyšli s definicí jediného pravidla pro hranový vrchol a pokoušeli se pravidlo pozměnit tak, aby se terén méně smršťoval. Připomeňme, že pravidlo přiřazuje hranovému vrcholu hodnotu

$$V = \frac{3}{8}(E_0 + E_1) + \frac{1}{8}(A_0 + A_1),$$

kde E_i reprezentuje koncový vrchol dané hrany a A_i je protilehlý vrchol v trojúhelníku obsahujícím danou hranu (viz Obrázek 3.3). Je snadné nahlédnout, že problém je ukryt ve třech osminách z obou koncových vrcholů, které dohromady představují hodnotu nižší, než jakou získáme běžnou lineární interpolací. Jedna osmina protilehlých vrcholů pak vrchol zpravidla ještě více „stáhne". První logická změna tedy byla použít pět osmin pro konce hran a odečíst jednu osminu protilehlých vrcholů. Výsledek byl lepší, ale stále velmi neuspokojivý. Ani větším experimentováním se nám nepodařilo nalézt na základě takto jednoduchého schématu žádnou výrazně lepší funkci.

Další postup byl založen na triangulaci mřížky, se kterou pracujeme. Snažili jsme se totiž vesměs aplikovat trojúhelníkové schéma na mřížku, která je od základu pravoúhlá (rodělená po diagonálách). To nás přivedlo k definování schématu, které by více respektovalo naše pravidelné dělení plochy a bylo citelnější k rozdílným úhlům trojúhelníku. Na základě pokusů se nám podařilo nalézt schéma, které pro nižší úrovně dělení dávalo poměrně pěkné výsledky blízké těm, které jsme získali užitím butterfly schématu. Dvě pravidla tohoto schémata znázorňuje Obrázek 4.5. Pravidlo pro horizontální hrany používá stejné koeficienty jako pravidlo pro vertikální.



Obrázek 4.5: Masky námi zvoleného dělícího schématu. Vlevo maska pro diagonálu, vpravo maska pro horizontální a vertikální hranu.

Když naše pravidlo pozvolna konvergovalo k vizuálně zajímavým výsledkům, rozhodli jsme se rozšířit geometry shader, ve kterém jsme dělící schéma implementovali, tak, aby prováděl adaptivní dělení na základě metriky chyby definované pohledem pozorovatele. Zde jsme narazili na jedno velmi zásadní omezení současného návrhu geometry shaderů, které se nám ani následně s jeho znalostí nepodařilo najít v žádné dokumentaci (nicméně nám bylo potvrzeno zástupci firem Microsoft a AMD). Toto omezení definuje limit pro objem dat, která je možno v geometry shaderu vyprodukovat na jedno vstupní primitivum. Limit je pevně stanoven na 1 024 32bitových položek ve výstupu shaderu, tj. 4 096 bytů. Protože na každý vrchol potřebujeme minimálně 16 bytů pro definování jeho homogenních souřadnic v prostoru, znamená tento limit teoretické maximum 256 vrcholů. Protože vrcholy generované v geometry shaderu není možné indexovat, musíme se snažit vytvářet co nejdelší řetězce trojúhelníků, abychom nepotřebovali na jeden trojúhelník 3 vrcholy. V ideálním případě tak získáme jeden řetězec dlouhý 254 trojúhelníků (3 vrcholy na první trojúhelník a každý další už je dán pouze jedním vrcholem). Opět jde spíše o teoretické maximum, v praxi je těžké představit si situaci, kdy bychom dokázali vytvořit tak ideální řetězec. Navíc naše vrcholy dost pravděpodobně budou potřebovat i jiné informace, než jen homogenní souřadnice (přinejmenším dalších 8 bytů na souřadnice textury). I po optimalizacích kódu a datové reprezentace (komprimace některých dat a jejich dekomprese v pixel shaderu (94)) jsme však dosáhli stropu přibližně 100 trojúhelníků, což je bohužel příliš málo, než aby se dalo využít na masivní znásobení geometrie přímo na grafické kartě. Toto je jediný zásadní důvod, který nás přinutil opustit myšlenku generování terénu jednoprůchodově na GPU.

Stále zbývá možnost generování dat ve více průchodech, tj. omezit hloubku dělení při jednom průchodu a výstup z jedné iterace použít jako vstup pro následující iteraci. Pokud bychom provedli několik dělení v jednom průchodu, stačily by nám pravděpodobně 3, maximálně 4 takové iterace. Problém, se kterým se takové řešení potýká, jsme však už zmínili dříve – je to objem dat, která jsou potřeba pro provedení dělení. Námi navržený jednoprůchodový způsob těžil z toho, že data, která jsou společná pro větší množství trojúhelníků, jsou čtena jenom jednou. Takto bychom museli data číst opakovaně, a co hůře, pro všechna výstupní data generovat informace o sousedech. Data by postupně procházela zobrazovací jednotkou, byla zapisována do výstupního bufferu a opět by byla zařazována na další zpracování. Jen tento fakt samotný nám zajistí několikanásobně nižší výkon (dále rozebráno v odstavci 4.2.3).

Vývoj grafického hardware je stále velmi dynamický. Před pár lety byl prvně definován Shader Model 1.0, který umožňoval použít 12 instrukcí v jednom pixel shaderu. Pro Shader Model 2.0 to bylo 96 instrukcí a dnes je toto množství neomezené. Je to pouze jediný příklad. Dá se předpokládat, že i zmíněný limit na velikost výstupních dat z geometry shaderu bude v budoucnu zvýšen, možná odstraněn docela. Otázkou však je nejen, kdy se tak stane v rámci specifikace, ale předně kdy bude hardware takovou specifikaci splňující více rozšířen do domácích počítačů, aby bylo možné této schopnosti využít.

4.2.3 Implementační poznámky

Předchozí text jsme se snažili oprostit od implementačních detailů, abychom udrželi přehlednost a mohli snáze popsat celý rozhodovací proces. Nyní chceme nabídnout několik poznámek k řešení některých podúloh a nabídnout nějaké další závěry, ke kterým jsme dospěli.

4.2.3.1 Předání vrcholů grafické kartě

Celý terén byl reprezentován pravidelnou pravoúhlou mřížkou, v níž jsou jednotlivé čtverce rozděleny diagonálou na dva pravoúhlé trojúhelníky. Nejjednodušší postup, jak zobrazovat takovouto mřížku, je naplnit vertex buffer vrcholy pro každý trojúhelník zvlášť, což je samozřejmě velmi neefektivní z hlediska paměťových nároků i zbytečného transformování některých vrcholů vícekrát. Zavedení index bufferu a použití řetězce trojúhelníků namísto prostého seznamu je pouze jedna, samozřejmá část optimalizace. Druhá spočívá v lepším využití vertex cache, tj. vyrovnávací paměti pro transformované vrcholy. Do té jsou ukládány všechny vrcholy, které prošly vertex shaderem, a podle FIFO pravidla jsou z ní vyjímány ve chvíli, když přijde nový transformovaný vrchol. Vrcholy jsou zpracovávány přesně v tom pořadí, v jakém jsou indexovány, takže můžeme předem přesně říci, kolikrát se projeví „cache-miss“ na základě jejich pořadí. Předpokládejme, že známe velikost vertex cache. V takovém případě jsme schopni přeskládat indexy vrcholů tak, že se trojúhelníky zpracovávají po řádkách a ve chvíli, kdy zaplníme vyrovnávací paměť, můžeme přejít na další řádek, kde použijeme polovinu již dříve transformovaných vrcholů, neboť jsou společné vždy několika trojúhelníkům a jsou dosud drženy ve vyrovnávací paměti.

Když se na tento problém podíváme z teoretického hlediska, tak oproti naivní implementaci transformace 3 vrcholů pro každý trojúhelník, můžeme s využitím vyrovnávací paměti transformovat pouze tolik vrcholů, kolik jich je v naší pravidelné struktuře. Vezmeme-li v úvahy velmi velkou mřížku, kde se příliš neprojeví okrajové vrcholy, jež náleží méně než 6 trojúhelníkům, tak můžeme teoreticky transformovat až šestkrát méně vrcholů. Pokud bychom tedy pominuli práci pixel shaderu a dokázali transformovat každý vrchol pouze jednou, můžeme provést zobrazování až šestkrát rychleji. Realita je samozřejmě odlišná – pixel shader provádí nějakou netriviální operaci (v případě unifikovaných shaderů navíc bere výkon z celkového rozpočtu) a vertex cache není nekonečná, takže se do ní nevejdou všechny vrcholy a i inteligentním indexováním je začneme ztrácet. Přesto postupem, který jsme popsali výše, můžeme reálně dosáhnout až dvojnásobné rychlosti zobrazování. Důležité však je znát velikost cache a napsat algoritmus indexování vrcholů správně i s ohledem na jejich pořadí v trojúhelníku. Dozvědět se velikost cache je někdy snadné, neboť výrobci grafických karet tato čísla občas zveřejňují, v jiných případech jsou tato čísla neznámá a je třeba je zjistit

experimentálně. Stačí výše zmíněný postup opakovat několikrát za sebou a předpokládat v něm vždy jinou velikost vyrovnávací paměti. Nejlepší výsledek nám pak ukáže pravděpodobnou hodnotu. Postup jsme úspěšně aplikovali a dosáhli experimentálního nalezení hodnoty, která souhlasí s výrobcem proklamovanou hodnotou. Konečně s využitím znalosti této hodnoty se nám podařilo v jednodušších případech získat dvojnásobek až trojnásobek původní rychlosti zobrazování. Nutno však dodat, že v celém systému se to projeví daleko méně, a že provedení správné indexace složitějších struktur, než je pravidelná mřížka, může být velmi obtížné. Více se touto problematikou zabývá například (95), (96) nebo (97). Hloubkový rozbor algoritmů pracujících nad obecnou sítí najdeme v (98).

Časem se ukázalo zbytečné číst z vertex bufferu kompletní data o vrcholech trojúhelníků, které se stejně nenakreslí, neboť jsou nahrazeny menšími trojúhelníky při adaptivním dělení. Navíc i pro trojúhelníkové dělící schéma se jevilo nepraktické zpracovávat samostatně jeden trojúhelník, když dva trojúhelníky tvořící jeden čtverec sítě potřebují téměř stejná data pro své dělení. Nahradili jsme tedy triangulovanou mřížku mřížkou obsahující pouze zástupné vrcholy – pro každé pole mřížky jeden. Geometry shader, který implementoval dané schéma, dostal pouze jediný bod a vygeneroval na jeho základě geometrii pro dva trojúhelníky základní mřížky. Konečně poslední úprava opět využila nové vlastnosti Shader Modelu 4.0, který umožňuje využít implicitní proměnné pro zpracovávaná primitiva. Pro vrchol je to například pořadí tohoto vrcholu. Když známe velikost zobrazované mřížky, můžeme z pořadí vrcholu přesně určit, o jaký vrchol se jedná, a nepotřebujeme tak jeho souřadnice. Zmíněnou proměnnou (VertexID) jsme nazvali implicitní, protože není potřeba explicitně ji definovat – obdržíme ji v shaderu automaticky. Takto nemusíme dodávat vůbec žádná data – stačí specifikovat, kolik vrcholů chceme zpracovat, a ve vertex shaderu se nám uklizuje pro každý vrchol s jediným vstupem, a tím bude jeho ID.

4.2.3.2 Využití geometry shaderu

Geometry shader, který jsme napsali pro implementaci subdivision surfaces, je trochu neobvyklý svou komplexitou. Jeho úloha spočívá v určení, do jaké míry má být vstupní blok rozdělen, vyzvednutí dat potřebných k takovému dělení a konečně provedení dělení samotného, tzn. vygenerování velkého množství vrcholů, které budou každý jeden transformovány jako by byly, kdyby prošly vertex shaderem. Toto obnáší o několik řádů více instrukcí než „běžný“ vertex shader (to vychází už z toho, že transformace vrcholu tu může být provedena i tisíckrát). Zdálo se nám tedy velmi důležité optimalizovat jej tak, aby sám nespotřeboval celou vyrovnávací paměť instrukcí a aby prováděl opakované výpočty co nejméně krát – to je poměrně složitý úkol, neboť na jeho druhé straně stojí omezené množství registrů a omezená možnost uchovat všechna spočítaná data. O to více nás však překvapilo, že i přes toto velké množství aritmetických operací byl výkon geometry shaderu přímo úměrný počtu vygenerovaných vrcholů – ne počtu vypočítaných vrcholů!

Pokusili jsme se tedy provést více experimentů a nalézt zdroj problému. Ukázalo se, že problémem je způsob, jakým je výstup z geometry shaderu hardwarově implementovaný. První experiment byl založený na nahrazení vertex shaderu geometry shaderem. Přesněji řečeno, ve vertex shaderu jsme pouze předali data dál pro zpracování geometry shaderem, a ten

transformoval všechny vrcholy stejným způsobem, jakým by to udělal vertex shader. Samozřejmě každý vrchol byl použit pouze v jednom trojúhelníku, takže bylo znemožněno využití vyrovnávací paměti a znehodnocení výsledků. Výsledky ukázaly, že zpracování vrcholů geometry shaderem bylo přibližně dvakrát pomalejší pro zvolený kód.

Následoval další experiment, ve kterém jsme měnili velikost výstupu geometry shaderu a komplexitu prováděného kódu. Zde se ukázalo, že dvakrát větší objem dat generovaných shaderem znamená téměř dvojnásobné snížení rychlosti. Dodejme, že to není samozřejmé chování – shader prováděl netriviální výpočty a rozšíření velikosti dat bylo spojeno pouze s kopírováním již jednou spočítaných dat. Vedle toho jsme se snažili zajistit, aby ani exekuce pixel shaderu nemohl zásadně ovlivnit výsledky – pixel shader byl triviální a data jsme generovali tak, aby nebyla vidět, nebo aby se hojně využilo „Early Z“ vyloučení[19]. Naopak změny ve složitosti prováděného kódu se neprojevily konzistentním poklesem výkonu.

Konečně poslední experiment byl odvozen z výsledků předchozích. Transformace vrcholů jsme prováděli pomocí vertex shaderu, ale tentokrát jsme zapojili i geometry shader, který však pouze předával data dále pro další zpracování (v prvním testu nebyl geometry shader zapojený vůbec). Výsledek tohoto postupu ukázal, že pouhé zapojení geometry shaderu do zobrazovacího procesu může dramaticky snížit výkon (v našem případě klesl výkon na polovinu). Z těchto našich experimentů jsme vyvodili vlastní závěr, že geometry shader zapisuje data do paměti grafické karty, dokud jsou pak znovu načítána pro další zpracování. Bez geometry shaderu se zpracovávaná data udržují ve (vyrovnávací) paměti výkonné jednotky dokud nejsou vyloučena ze zpracování nebo zapsána do výstupního bufferu. Jiným vysvětlením by mohly být špatně napsané ovladače grafické karty, ale při testování byly vyzkoušeny dvě verze, proto spíše věříme výše uvedenému závěru. Pokud je tomu tak, zdá se být geometry shader ještě nezralý pro některé způsoby použití.

4.2.3.3 Minimalizace objemu vstupních dat pro geometry shader

Při experimentování s různými dělícími schématy se záhy ukázalo, že není možné psát zvláštní kód pro každé. Zobecnili jsme tedy tento postup a vyjádřili každý vrchol v několikrát dělené síti jako lineární kombinaci řídících vrcholů. Díky tomuto přístupu je možné spočítat výšku jednoho vrcholu pomocí 16 skalárních násobení a 16 sčítání, nebo pouze 5 skalárních součinů čtyřprvkových vektorů. Takovéto vyjádření však znamená velké množství dat, protože musíme vyjádřit každý vrchol výsledné sítě. Pro pevně zvolenou maximální úroveň dělení n má nejdetailnější síť $[(2]^n + 1)^2$ vrcholů, což pro mřížku řídících vrcholů o rozměrech 4×4 a potřebě 4 bytů na vyjádření jednoho koeficientu představuje $64(2^n + 1)^2$, tedy přibližně 70 kB pro $n = 5$. V případě mřížky 6×6 pro butterfly by to bylo dokonce 150 kB, proto se zde velmi projeví malý nosič schématu.

Nejdříve jsme uložili data v podobě pole matic (každá matice má 64 bytů) a hledali jsme způsob, jak efektivně využít toho, že ne všechny koeficienty každé matice jsou nenulové. (Nulový prvek

[19] Early Z se označuje technika vyloučení pixelu ze zpracování na základě jeho negativního z-testu ještě před tím, než se spustí kód pixel shaderu. Podrobněji je technika popsána například v (103).

odpovídá tomu, že daný řídící vrchol nemá na zpracovávaný vrchol žádný vliv.) Pro schéma, se kterým jsme nejvíce pracovali, jsme tedy vygenerovali tabulky pokrytí, které nám ukázaly, kolik nulových koeficientů která matice má a na kterých místech matice se často vyskytují (ne)nulové koeficienty. Tabulka 4.1 ukazuje, kolikrát se na kterém místě matice objevil nenulový koeficient mezi maticemi pro všechny výstupní vrcholy. Z tabulky je vidět, že celkový počet nenulových koeficientů není ani 10 000, ale my jich ukládáme více než 17 000 (stále pro $n = 5$). Toho by bylo možné využít.

Počet nenulových koeficientů			
439	747	225	0
747	1 086	1 076	225
225	1 076	1 086	747
0	225	747	439

Tabulka 4.1: Celkové počty nenulových koeficientů nacházejících se v odpovídajících místech matice (součet pro všechny vrcholy).

Udělali jsme druhý test, ve kterém jsme považovali za nulové i koeficienty blízké nule (konkrétně menší než jedna tisícina). Využití koeficientů výrazně kleslo. To nás přivedlo na myšlenku pokusit se najít nějakou mez, pod níž budeme koeficienty považovat za nulové (nedostatečně přispívající k celkové poloze vrcholu) a která výrazně sníží využití matic koeficientů. Podařilo se nám dospět k hodnotě 0,0195, která se nám stále zdála dostatečně nízká na to, aby viditelně ovlivnila polohy vrcholu, a zároveň jsme s ní částečně získali požadované výsledky, které ukazuje Tabulka 4.2. Celkový počet „nenulových“ koeficientů klesl na 5 220 a jak je vidět z tabulky, polovina koeficientů je vždy nulová. Takto modifikované koeficienty můžeme velmi snadno uschovat do polovičního prostoru – de facto jsme zmenšili nosič schématu na 8 vrcholů. My je však ukládáme do jediného čtyřprvkového vektoru tak, že celá část jedné komponenty reprezentuje jeden koeficient vynásobený 1 000 a desetinná část reprezentuje druhý koeficient. Celkově jsme tak snížili velikost jedné matice na 16 bytů, tj. přibližně 17 kB pro $n = 5$. Tato data předáváme do shaderu v podobě bufferu konstant.

Počet nenulových koeficientů			
0	332	0	0
332	1 086	860	0
0	860	1 086	332
0	0	332	0

Tabulka 4.2: Celkové počty koeficientů větších než 0,0195 nacházejících se v odpovídajících místech matice (součet pro všechny vrcholy).

Dodejme, že jako další možnost uchování koeficientů řídké matice se nabízí zakódovat do jedné komponenty umístění nenulových koeficientů uvnitř matice a do zbylých komponent uložit jednotlivé použité koeficienty (způsobem, který jsme již popsali). Význam to má však pouze tehdy, pokud máme nižší maximální počet nenulových koeficientů uvnitř matice, ale nemáme přesně dáno, které koeficienty to mohou být. Dále pokud mají matice navzájem symetrický charakter (a to mají), je možné uložit jenom část z nich a za cenu dalších instrukcí při

dekompozici matice provádět otáčení podle toho, jaký vrchol zrovna zpracováváme. K tomuto postupu jsme se již nedostali, neboť celou myšlenku generování terénu na GPU jsme z dříve uvedeného důvodu opustili.

4.3 Použití fixní mřížky

Po neúspěchu s adaptivním dělením jsme chtěli navrhnout nějakou strukturu, která by pokryla první fáze dělení, podobně jako to dělá omezený kvadrantový strom. Další fáze bychom už mohli dokončit na GPU. Po různých experimentech s geometry shaderem, které jsme popsali v odstavci 4.2.3.2, jsme však usoudili, že rychlost tak nezískáme a kvalitativně téměř stejného výsledku můžeme dosáhnout pomocí clipmap s jiným vertex shaderem. Obrátili jsme tedy pozornost na definování nějakého vzorkovacího schématu, které by vyřešilo pohledově závislý LOD podobně jako to dělají clipmapy. Připomeňme, že clipmapa nebere v úvahu charakter terénu ani úhel pohledu, pouze vzdálenost od pozorovatele. Tento fakt by bylo možné změnit, ale k tomu se vrátíme až v odstavci 4.7.2. Shrnuli jsme tedy nedostatky clipmap, na které se chceme zaměřit:

1. Komplikovaná změna zorného úhlu, přesněji zmenšení zorného úhlu (použití dalekohledu).

2. Modifikace terénu ve vyšších frekvencích.

3. Výpočty fyzikálních interakcí objektů se vzdálenějším terénem.

4. Poměrně složitá implementace, která zahrnuje nevyhnutelné morfování, skládání každé úrovně clipmapy z různých dílů, protože úrovně nejsou přesně centrovány, atp.

5. Implicitní metrika chyby nezohledňuje nerovnost povrchu.

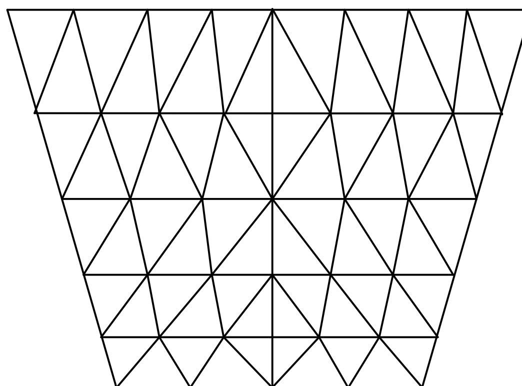
Pro výpočet fyzikálních interakcí se zpravidla používá jiná datová struktura než pro zobrazování, takže tento bod můžeme vynechat. První dva body mají stejný původ, a to různorodou kvalitu uchování viditelné části terénu. Tento problém jsme již rozebírali v odstavci 4.1.2 a z větší části jej řeší námi navržená dvouúrovňová struktura. Zúžení zorného úhlu však vyžaduje kromě detailnějších dat také jiné vzorkování a v závislosti na implementaci i jiné ořezávání pohledovým jehlanem. Tyto a další skryté detaily dělají z clipmap poměrně náročný algoritmus pro úplnou implementaci; jednoduché zobrazování mřížky v okolí pozorovatele je triviální, ale samo o sobě ne použitelné. Konečně pravidelné vzorkování má za následek ztrátu informací ve vyšších frekvencích, nebo případné převzorkování méně zvrásněných oblastí.

Pokusíme se navrhnout alternativní přístup, který bude podávat výsledky výkonnostně srovnatelné s tradičními clipmapami a odstraní některé z uvedených problémů. Doplňme ještě, že i čtvercová mřížka, kterou používají clipmapy, by šla efektivně použít na námi navrženou dvouúrovňovou reprezentaci terénu, ale přesto se pokusíme navrhnout jiné schéma vzorkování a zhodnotit jeho vlastnosti. V následujících odstavcích tedy představíme dvě taková schémata, se kterými jsme nejvíce pracovali, a ukážeme, jakým způsobem jich lze využít pro efektivní ořezávání neviděných oblastí terénu.

4.3.1 Struktura trapezmap

Struktura, kterou jsme nazvali *trapezmap*, se zakládá na tom, že v každou chvíli je z celého terénu vidět pouze lichoběžníková výseč, kterou dokážeme poměrně přesně určit. Zdá se tedy být zbytečné řešit složitými postupy ořezání geometrie pomocí pohledového jehlanu, nebo dokonce zpracovávat data, která jsou daleko za pozorovatelem. Stejná myšlenka vedla také návrh algoritmů popsaných v (62) a (63), kde se na terén nahlíží v prostoru obrazovky a tam se také provádí jeho dělení na trojúhelníky. Teprve po zpětné projekci do prostoru scény se vyzvednou požadované vzorky z výškové mapy. Zdánlivě ideální řešení se nám však v praxi neosvědčilo – bylo příliš citlivé na jakýkoli pohyb kamery ve všech směrech. Výšková mapa je totiž vzorkována v různých pozicích, které se mění s každým nepatrným pohybem. Při chůzi se tak neustále mění výšky terénu a dochází k velmi nepříjemnému aliasu. Problém je o to horší, že alias se projevuje i při pouhé změně směru pohledu.

Námi navržená struktura trapezmap odstraňuje závislost na sklonu kamery vzhledem k vodorovné rovině. Při takové změně se nemění pozice vzorků, takže se neprojeví alias. Struktura je navržena tak, že se podle velikosti zorného úhlu a přibližné výšce pozorovatele nad terénem (v rámci jistého rozpětí) spočítá lichoběžníkový útvar, který vznikne projekcí pohledového jehlanu do roviny kolmé na osu Y. Vzniklý lichoběžníkový útvar však nepoužíváme přímo. Vždy jej protáhneme až k pozorovateli, a dokonce malý kousek za něj, a na vzdálené straně jej taktéž protáhneme až do maximální dohledové vzdálenosti. Toto nám způsobí, že pro pevnou velikost zorného úhlu má vzniklý lichoběžník stále přibližně stejný tvar a my se jednoho takového tvaru můžeme držet tak dlouho, dokud se radikálně nezmění pohledové parametry (především velikost zorného úhlu). To, co nás na něm zajímá, je především šířka jeho základny, která je blíže k pozorovateli, a úhel, který svírají rozbíhavé hrany. Pro takovýto lichoběžník máme připravenu triangulaci, kterou použijeme pro zobrazování viditelné části terénu a která tvoří definici struktury trapezmap (viz Obrázek 4.6).



Obrázek 4.6: Trapezmap struktura - schéma pro vzorkování výškové mapy. V blízkosti pozorovatele je nejhrubší triangulace, s rostoucí vzdáleností se zvětšuje i relativní úroveň dělení.

Triangulace lichoběžníku je založena na rozdělení lichoběžníku na pásy kolmé na směr pohledu pozorovatele. Šířka každého pásu se zvětšuje se vzdáleností od pozorovatele – tím se dosahuje nižší frekvence vzorkování méně významných částí terénu. (62) navrhuje provést rovnoměrné vzorkování po celé obrazovce. My s tímto přístupem nesouhlasíme. Ve vzdálených oblastech totiž

potřebujeme pro dobré vizuální výsledky vzorkovací frekvenci srovnatelnou s desetinou či dvacetinou pixelu (odpovídá vzdálenosti 10, resp. 20 pixelů mezi vzorky). Ještě vyšší frekvenci potřebujeme na korektní zachycení horizontu. Naproti tomu v těsné blízkosti pozorovatele nám stačí pořizovat vzorky s rozpětím 50 i více pixelů. Je to dáno jednak tím, že jsou to především vzdálené oblasti, které tvoří siluety, na něž je lidské oko tolik citlivé, jednak také tím, že vysoká vzorkovací frekvence v blízkosti pozorovatele dokáže výrazně překročit frekvenci, v níž máme data uložena (tj. detail výškové mapy). Proto navrhujeme rozdělit oblast obrazovky nerovnoměrně s důrazem na partie, do nichž se promítne vzdálená část terénu. Od toho se odvíjí také triangulace jednotlivých pásů trapezmap struktury. Počet trojúhelníků v jednotlivých pásech narůstá se vzdáleností od pozorovatele. Problém navazování odlišně dělených oblastí, jaký řeší většina LOD algoritmů, clipmapy nevyjímaje, jsme však vyřešili velice snadno pomocí restrikce přidání maximálně dvou trojúhelníků na pás trapezmapy (To opět názorně předvádí Obrázek 4.6.). Pokud má tedy nejužší pás n trojúhelníků, ten nejširší jich nemůže mít více než $n + 2(k - 1)$, kde k reprezentuje počet pásů trapezmapy.

Počet trojúhelníků v jednotlivých pásech, resp. nárůst jejich počtu, je parametrem struktury. Není založen na matematickém výpočtu požadované vzorkovací frekvence, protože ta závisí na subjektivním vnímání, a je tedy těžké ji popsat vztahem. Parametry, ke kterým jsme po nějakém čase dospěli, jsou založeny pouze na experimentování a subjektivním vjemu. Při použití trapezmapy jiným způsobem by bylo zřejmě nutné nalézt jiné vhodné parametry. Neplatí to pro šířku jednotlivých pásů, kterou počítáme na základě definovaného rozestupu prvních dvou, celkového počtu pásů k a maximální vzdálenosti dohledu. Nahlížíme přitom na vzestupnou velikost pásů jako na geometrickou posloupnost. Ze vztahu pro součet prvních k členů geometrické posloupnosti

$$S = \frac{a(1 - q^{k+1})}{1 - q},$$

ve kterém S vyjadřuje součet, a velikost prvního členu, k počet sčítaných členů a q hledaný kvocient, vyjádříme rovnici

$$q * (S - a * q^k) + a - S = 0,$$

v níž q představuje proměnnou. Tuto rovnici řešíme Newtonovou metodou tečen a získáme z ní hledaný kvocient nárůstu velikosti jednotlivých pásů trapezmapy. Tímto postupem spojeným se správnou volbou parametrů jsme dosáhli požadovaného rozložení vzorků po ploše terénu.

Přestože jsme odstranili závislost čtení vzorků na sklonu kamery, alias se stále silně projevuje. Velmi rušivé je to při otáčení pozorovatele kolem osy Y, protože to je změna, kterou lze provést daleko rychleji ve srovnání se změnou polohy.

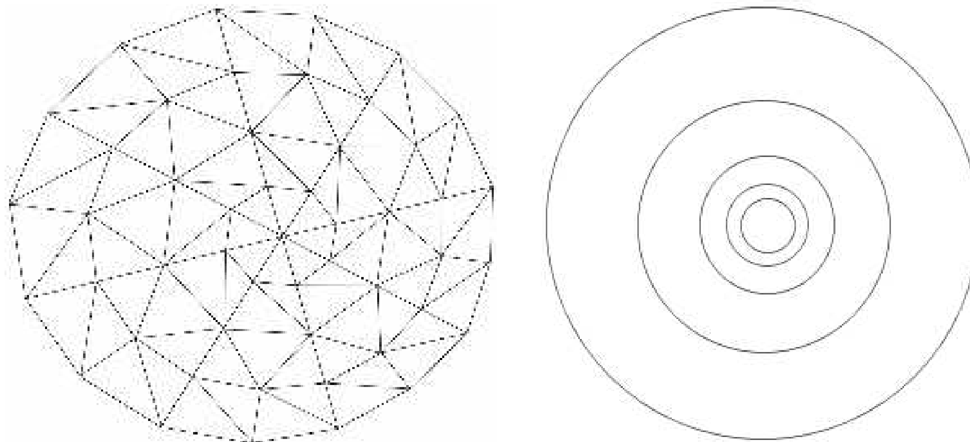
Další negativní vlastností trapezmapy je její nestálý charakter. Při změně výšky pozorovatele nad okolním terénem nebo při změně zorného úhlu je třeba vypočítat nový lichoběžník a strukturu adaptovat na jeho vlastnosti. Není to náročný úkol, ale přináší to komplikace při hledání

vhodných parametrů pro vytvoření struktury (připomeňme, že parametry jsme hledali experimentálně). Souvisí to i s tím, že pozorovatel se může dívat přímo do země a je třeba zobrazovat oblast velmi odlišného charakteru s jiným schématem vzorkování. Metoda se dá tedy velmi těžko popsat pro obecné účely a je třeba ji implementovat s ohledem na zadané podmínky. To z ní dělá implementačně poměrně komplikovanou metodu, nikoli vhodnou náhradu za clipmapu. Její výhodou je implicitní řešení ořezávání pohledovým jehlanem. Zbývá nám pouze ošetřit zvláštní případy, o kterých jsme již mluvili (pohled do země či lehce vzhůru do dálky).

4.3.2 Struktura ringmap

Pro problémy s trapezmapou jsme hledali různá jiná schémata pro vzorkování. Ukázalo se, že nejdůležitější je zachovat vzorky na svých místech při otáčení kolem osy Y. Při pohybu se alias také projevuje, ale daleko méně. Odtud jsme vyšli a navrhli strukturu, které jsme dali název *ringmap*, pro její prstencový charakter. Struktura je složena ze dvou částí – centrálního kruhu a soustavy prstenců. Centrální kruh je centrovaný na pozici pozorovatele a zajišťuje snadné řešení ořezávání pohledovým jehlanem. Na něj je navázána soustava prstenců, které jsou triangulovány vzhledem k souřadnému systému scény, jsou tedy invariantní vzhledem k otáčení pozorovatele kolem osy Y, a tím výrazně přispívají k odstranění aliasu.

Hlavní část ringmapy představuje soustava prstenců, které se směrem od středu zvětšují stejným způsobem, jako pásy trapezmapy, jde tedy opět o geometrickou posloupnost (viz Obrázek 4.7 vpravo). Každý prstenec je triangulován pomocí stejného počtu trojúhelníků. Protože jednotlivé prstence se zvětšují, zvětšuje se i celková plocha těchto trojúhelníků, a tedy frekvence, s níž vzorkují terén ve vzdálenějších oblastech. Tento přirozený nárůst má velikou výhodu v tom, že počet trojúhelníků je stále stejný a jejich zvětšování probíhá plynule, nikoli skokově jako tomu bývá při použití pravoúhlé mřížky. Proto nevzniká potřeba ošetřovat T-vrcholy a praskliny – síť je vždy spojitá. Problém nastává v blízkosti pozorovatele, kde není možné použít stejný počet trojúhelníků na prstenci, jako pro vzdálené oblasti. Jedná se o příliš malou plochu, a pokud bychom se ji snažili dělit stovkami trojúhelníků, měly by jednotlivé trojúhelníky velikost často menší než jeden pixel. Nejen, že bychom tak vytvářeli zbytečnou zátěž grafické karty, ale především silné nadvzorkování dat, které je v čase (při pohybu) nestálé, způsobuje velmi nepříjemné vizuální efekty. Z tohoto důvodu je ringmapa složena ze dvou částí a terén blízký pozorovateli se zobrazuje pomocí jiným způsobem stavěné centrální části.

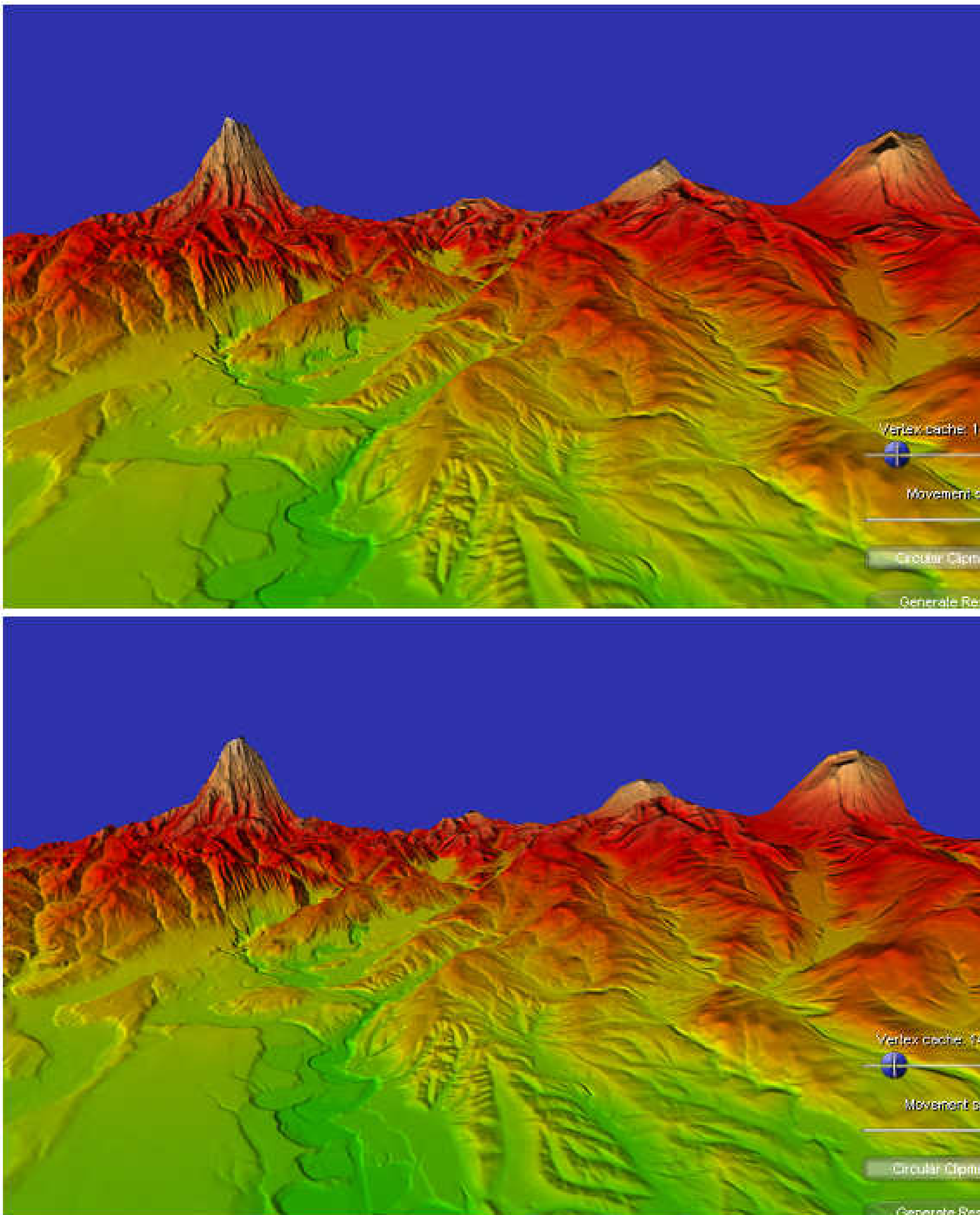


Obrázek 4.7: Vlevo 4 prstence centrálního kruhu ringmapy. Třetí prstenec je přechodový – utváří plynulý přechod mezi 8 a 16 segmenty. Vpravo schematicky zobrazena vnější soustava prstenců s narůstajícím poloměrem.

Centrální kruh nepředstavuje nic jiného než triangulaci díry uprostřed ringmapy. Je sestaven také z prstenců, resp. mezikruží, ale jednotlivé prstence mají různé počty trojúhelníků. Ze středu vychází pouze několik trojúhelníků a na některých, tzv. přechodových prstencích se provede rozdvojení, jak ukazuje Obrázek 4.7 vlevo. Takto sestavená triangulace je opět bez jakýchkoli prasklin nebo T-vrcholů. Nutné je pouze zajistit, aby na sebe spojitě navázal centrální kruh s vnější oblastí. Toho není těžké dosáhnout, pokud použijeme stejné pravidlo na rozestup prstenců a použijeme stejný shader kód na transformaci obou spojených vrcholů.

Velikost centrálního kruhu, rychlost nabývání počtu trojúhelníků od středu, celkový počet prstenců, počet trojúhelníků na jednom prstenci, rychlost růstu velikosti prstenců,… to všechno jsou parametry, kterými lze ovlivnit charakter ringmapy tak, aby splňovala dané potřeby. Přestože parametrů na nastavení je celá řada, jejich volba je poměrně přirozená, a proto na rozdíl od trapezmapy není těžké najít vhodnou konfiguraci, případně vytvořit obecnou, použitelnou ve většině situací.

Velkým problémem stále zůstává alias (viz Obrázek 4.8). Podařilo se nám jej omezit pouze na pohyb pozorovatele, ale může být stále neakceptovatelný; to závisí na mnoha parametrech - charakteru zobrazovaného terénu, rychlosti pohybu, konfiguraci ringmapy atd. Odstraňováním tohoto problému jsme strávili velké množství času a podařilo se nám jej do jisté míry potlačit. Přesto zůstává hlavním negativem našeho postupu.



Obrázek 4.8: Praktické projevení aliasu při vzorkování geometrie terénu. Snímky byly pořízeny ze stejného úhlu, ale odlišných pozic (posun přibližně 100 metrů). I malý posun znamenal jiné rozložení vzorků, a tedy jinou výslednou geometrii. Nejvíce je to patrné na siluetách kopců. Při animovaném pohybu jsou tyto rozdíly daleko patrnější.

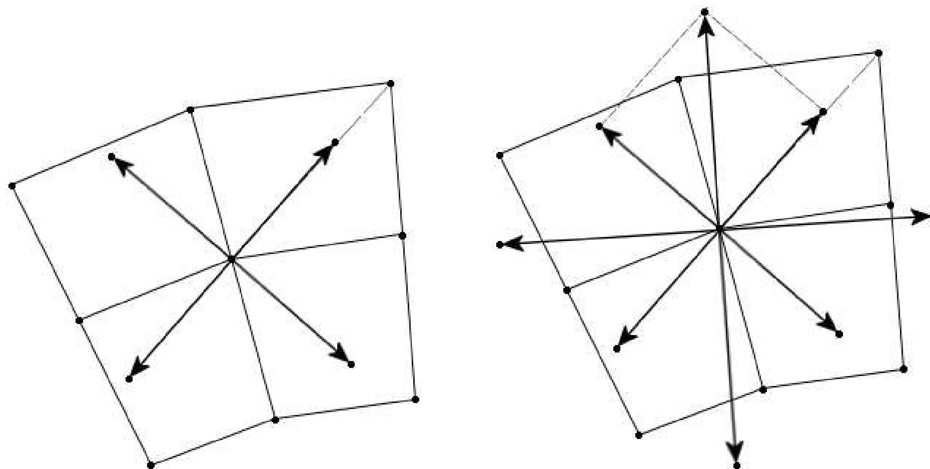
Problém tkví přirozeně v nekonzistentním vzorkování. Zatímco clipmapy jsou definované v světových souřadnicích a posun úrovně clipmapy se děje po diskrétních vzdálenostech, naše struktura se pohybuje spojitě současně s pozorovatelem. Při pohybu jsou proto vzorky sbírány ve stále jiných pozicích. Řešením by bylo posunovat ringmapu po diskrétních pozicích, jako se to dělá v případě clipmap. To je však komplikovaný úkol, neboť struktura je kruhová a vrcholy v ní nejsou rozmístněny pravidelně – i při skokovém posunu po diskrétních vzdálenostech není možné zajistit, aby při udržení stejné geometrie ringmapy všechny vrcholy padly do „vzorkovacích" pozic, tj. pozic, odkud se četly vzorky pro minulou polohu. Mohla by vzniknout otázka, zda není možné udělat geometrii ringmapy dynamickou a posunovat každý její prstenec nezávisle na ostatních. To by teoreticky bylo možné, ale pouze při pohybu kupředu. Při obecném pohybu není triviální najít vzorkovací pozici, do které by měl vrchol ringmapy padnout.

Současné řešení problému s aliasem je založeno na nadvzorkování mapy reziduí. S tím souvisí i celá rekonstrukce výšky terénu, proto ji zde velmi stručně popíšeme: Do vertex shaderu vstoupí index vrcholu ringmapy. Ten jednoznačně identifikuje vrchol – pořadí prstence i index vrcholu v prstenci. Podle konstantních parametrů ringmapy se spočítají relativní souřadnice vrcholu vůči středu struktury. Tyto souřadnice jsou dvourozměrné. Na základě aktuálních parametrů kamery se spočítají souřadnice v prostoru scény (stále dvourozměrné – projekce v rovině $y = 0$), a ty použijí jako základ pro souřadnice do textury. Nezávisle na sobě jsou z těchto texturových souřadnic odvozeny souřadnice do výškové mapy a do mapy reziduí na základě parametrů zmíněných map. Z výškové mapy jsou bez jakékoli filtrace vyzvednuty 4 nejbližší vzorky. Z nich je bilineární interpolací spočítána předpokládaná hodnota výšky terénu v daném místě a z mapy reziduí je načtena hodnota, která tuto výšku opraví. Nakonec se transformují souřadnice se zjištěnou výškou pomocí projekční matice.

Šetrně jsme obešli informaci o načtení hodnoty z mapy reziduí. To je právě místo, kde se provádí boj s aliasem pomocí nadvzorkování. Podle aktuální konfigurace se totiž vyzvedne z malého okolí 1, 5 nebo 9 vzorků, z nichž je počítán aritmetický průměr. Každý vzorek je buď načten bez filtrace, nebo s bilineární filtrací. Takto odstupňované metody nabízí již poměrně široké pole pro řízení poměru kvality a výkonu (více viz odstavec 4.6).

Umístění jednotlivých vzorků je postaveno na základě lokální frekvence vzorkování, aby byl zachycen charakter terénu v té úrovni detailu (tj. frekvenční oblasti), jaké je adekvátní pro danou oblast. Obrázek 4.9 ukazuje, odkud jsou vzorky brány v souvislosti s okolními vrcholy ringmapy.

Ringmapa se ukázala být v závěru velmi efektivním způsobem, jak vzorkovat výškovou mapu. Její implementace je poměrně snadná. Nějaké rozmyšlení si zaslouží triangulace centrálního kruhu. Tak, jak jsme ji provedli my, se zdá být poměrně triviální a vizuální výsledky neukazovaly žádné nepěkné charakteristiky. Nejkomplikovanějším místem tedy zůstává shader pro vzorkování mapy reziduí. Je třeba v něm správně převést souřadnice, a především provést nějaký supersampling na potlačení aliasu. Schéma pro zvolení vzorků, které jsme ukázali, není jediné možné. Naopak je potřeba s ním experimentovat pro získání dobrých výsledků na co nejnižší možný počet přístupů do textury. Toto je tedy asi jediná větší slabina zvoleného přístupu.



Obrázek 4.9: Umístění vzorků mapy reziduí vzhledem k umístění okolních vrcholů (soused na stejném prstenci a soused na stejné výseči). Mřížka reprezentuje část ringmapy, šipky ukazují na odebírané vzorky. Vlevo případ 4 vzorků, vpravo 9 vzorků.

4.3.3 Ořezávání pohledovým jehlanem

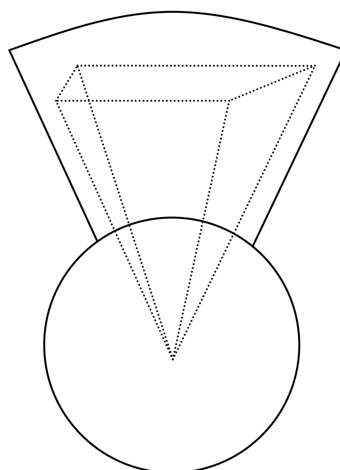
Většina algoritmů využívá pro ořezávání scény různé varianty binárních, kvadrant, oktalových a jiných stromů. Pro terén to platí zvláště v případě použití některého z algoritmů popsaných v odstavcích **Chyba! Nenalezen zdroj odkazů.** a 2.3.4, které jsou také postaveny nad hierarchickou strukturou. Často je ořezávání zakomponováno přímo do metriky LOD algoritmu. My jsme přišli se zcela odlišným postupem již v návrhu algoritmu pro zobrazování terénu, ve kterém je účinné ořezávání velmi jednoduchou záležitostí.

V případě trapezmapy dokonce není třeba řešit ořezávání vůbec – tak byla navržena: pokrývá přibližně ty části terénu, které jsou vidět (s jistým „bezpečnostním přesahem“, neboť je lépe zobrazit pár trojúhelníků navíc, než aby v obraze chyběly). Jediné, co můžeme vylepšit, je zkrácení dohledové vzdálenosti tehdy, když se pozorovatel dívá směrem k zemi. V takovém případě stačí zobrazovat podstatně méně než celou trapezmapu. Tuto optimalizaci jsme neprováděli, protože jsme brali v úvahu praktické použití. Když se pozorovatel v komplexnější scéně podívá do země, zobrazuje se velmi malá část scény, a proto není třeba tolik šetřit časem. Naproti tomu většinu doby se pozorovatel dívá směrem k horizontu, a tak celková rychlost musí být stejně stavěna na plnou zátěž.

Také s ringmapou se velmi dobře pracuje. Stačí na ni nahlížet jako na soustavu rozbíhavých výsečí, namísto soustavy prstenců. Kdybychom brali v úvahu pouze dvourozměrný svět, tvořila by kruhová výseč oblast, kterou pozorovatel vidí. Naše scéna je sice trojrozměrná, ale zvláště na terén můžeme pohlížet jako na objekt s více dvourozměrným charakterem. Toho využijeme a zjednodušíme tedy ořezávání pohledovým jehlanem na ořezávání kruhovou výsečí. Pro aktuální pozici pozorovatele a směr jeho pohledu takovou výseč musíme najít a ve chvíli, kdy ji máme spočtenou, je zobrazování terénu užitím ringmapy triviální. Důležité je mít vhodně definovaný index buffer pro strukturu vnější části ringmapy. Pokud indexujeme vrcholy po jednotlivých výsečích (tzv. „slice-major-order“), nachází se indexy viditelných vrcholů v index bufferu za

sebou. Spočítáme, které z výsečí jsou viditelné, a právě toto rozmezí necháme zobrazit grafickou kartou.

Protože jsme situaci představou dvourozměrné scény příliš zjednodušili, musíme ji vrátit zpět do trojrozměrného světa. Zorný prostor pozorovatele tvoří pohledový jehlan. Spočítáme tedy jeho projekci do roviny $y = 0$ a následně vzniklou omezenou plochu uzavřeme kruhovou výsečí se středovým vrcholem umístěným do pozice pozorovatele (resp. jeho projekce do téže roviny). Trojrozměrný případ připouští také komplikovanější situaci, při které se pozorovatel dívá směrem k zemi. Tehdy si nevystačíme s kruhovou výsečí, protože někdy je pokryt plný úhel kolem pozorovatele. Naproti tomu do dálky je vidět pouze několik prvních prstenců. Tuto situaci do jisté míry řeší fakt, že ringmapa se skládá ze dvou částí. Centrální kruh zobrazujeme za všechno okolností celý, a proto, není-li pozorovatel příliš vysoko nad povrchem, není třeba ani vnější část ringmapy zobrazovat. Obrázek 4.10 schematicky znázorňuje, co je v obecné situaci zobrazováno.



Obrázek 4.10: Projekce pohledového jehlanu do roviny $y = 0$, ve které se uzavře vzniklá plocha kruhovou výsečí. Výseč se aplikuje na vnější okruhy ringmapy, zatímco centrální kruh se zobrazuje vždy celý.

Rozdělení ringmapy do dvou částí tedy řeší dva problémy – vysokou frekvenci vzorkování v blízkosti pozorovatele a komplikovaný výpočet viditelnosti při některých pohledech. Velikost centrální ringmapy je proto dobré nastavit tak, aby porývala všechen terén, který je vidět z maximální výšky pozorovatele při pohledu do země. To není těžké zajistit v případě chodce nebo vozidla, ale v případě leteckého simulátoru bychom museli vytvořit robustnější řešení i na centrální kruh, aby se nezobrazoval stále ve stejné úrovni detailu.

Je také vhodné mít pro vnější oblasti ringmapy dva index buffery. Jeden, který indexuje vrcholy po prstencích, a druhý, který je indexuje po výsečích. V případě, že se pozorovatel dívá spíše do země, je dobré mít možnost zobrazit jen několik málo vnějších prstenců. Pokud se naopak dívá do dálky, je efektivnější zobrazovat terén po výsečích. Pohled na horizont je sice daleko častější

případ, přesto se vyplatí jednoduchým testem zjistit, kterým typem indexace v danou chvíli dosáhneme menšího počtu trojúhelníků pro vykreslení.

Popsaný způsob ořezávání pohledovým jehlanem je téměř ideální. Produkuje jen velmi malé množství trojúhelníků, je výpočetní nenáročný a implementačně velice jednoduchý díky efektivnímu využití struktury ringmapy. Za negativum by se dala považovat nesouhra se zbytkem scény. Objekty ve scéně budou totiž pravděpodobně uchovány v nějakém stromě a návaznost našeho postupu na takový strom by byla netriviální. S ohledem na zmíněné výhody se nám však zdá tato stránka zcela zanedbatelná.

4.4 Generování dat

V odstavci 4.1.2 jsme popsali dvouúrovňovou datovou strukturu, kterou užíváme k reprezentaci terénu. Řekli jsme si, že struktura se skládá z výškové mapy a z mapy reziduí. Výšková mapa obsahuje hrubý popis terénu a mapa reziduí je použita pro rekonstrukci detailu. V odstavci 4.3.2 jsme nastínili jednoduchý proces rekonstrukce skutečné výšky terénu ve zvoleném bodě na základě dat z obou map. V následujícím odstavci se budeme věnovat tomu, jak obě mapy vytvořit, neboť tato reprezentace není běžná a nezískáme ji nikde přímo. V ukázkové aplikaci, do které jsme námi navrhovaný algoritmus s použitím zmíněné struktury implementovali, je kromě těchto dvou map použita ještě normálová mapa k získání lepších vizuálních výsledků při stínování. Postup vytvoření normálové mapy popisuje odstavec 4.4.2.

4.4.1 Generování reziduí

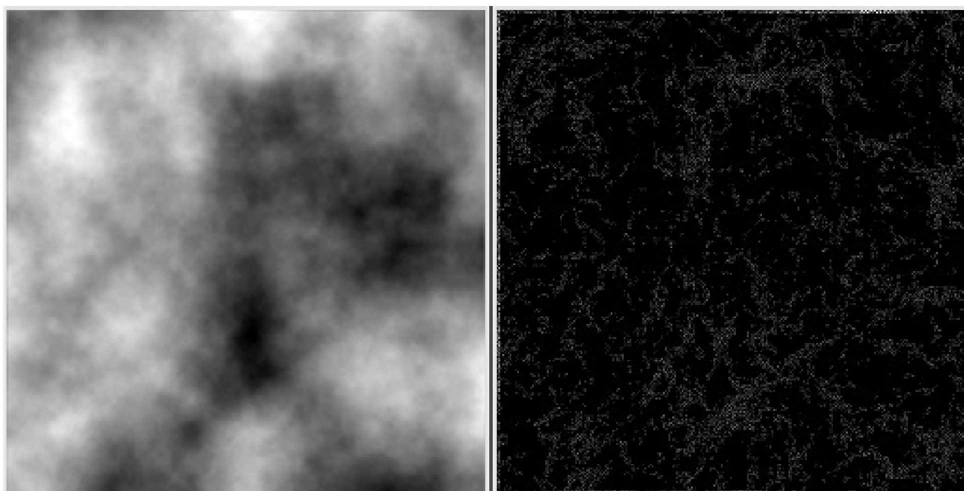
Pro účely testování bylo třeba vytvořit nějaká data, na kterých by bylo možné studovat a porovnávat dosažené výsledky. Za tím účelem jsme vytvořili jednoduchý nástroj na přeměnu vstupní výškové mapy na výškovou mapu nižšího rozlišení a mapu reziduí k rekonstrukci detailu. Bylo možné zvolit i jinou vstupní reprezentaci, ale protože výšková mapa je často volenou reprezentací a není těžké nějaká reálná data získat na Internetu (například USGS), rozhodli jsme se ji jako vstupní formát použít i my. I přes dostupnost reálných dat jsme však nakonec použili data generovaná automaticky pomocí k tomu vytvořeného nástroje.

Generování výškové mapy nízkého rozlišení ze vstupní výškové mapy je realizováno pouhým průměrováním všech vstupních hodnot, které pokrývá jeden vzorek výstupní výškové mapy. K tomu je definován faktor zmenšení, který zároveň udává počet vzorků mapy reziduí mezi dvěma vzorky výstupní výškové mapy.

Mapa reziduí má stejné rozlišení jako vstupní výšková mapa, proto se při výpočtu rezidua nepoužívá žádné filtrování vstupních dat, ale posuzuje se pouze jeden odpovídající vzorek. Reziduum je definováno jako rozdíl mezi hodnotou definovanou ve vstupní výškové mapě a hodnotou aproximovanou pomocí výškové mapy nižšího rozlišení. Volba aproximace může být libovolná. My jsme pro jednoduchost použili bilineární interpolaci čtyř nejbližších vzorků, ale kosinová interpolace by jistě fungovala stejně dobře, možná lépe. To záleží na charakteru dat terénu. Jistě by bylo možné pro konkrétní data zvolit vhodné aproximační schéma z nějakého pevně definovaného seznamu, které bude produkovat nejnižší chybu. Tomu však musí odpovídat postup zpětné rekonstrukce.

Abychom vůbec ušetřili nějakou paměť, musíme využít toho, že jsou hodnoty reziduí v menším rozsahu, než hodnoty výšek. K jejich vyjádření je tedy zapotřebí menší počet bitů. Navíc je možné využít koherence mezi sousedními hodnotami. Na snížení paměťových nároků by bylo možné využít různé známé komprimační algoritmy používané na kompresi obrazových dat. Pro nás by to však znamenalo nutnou dekompresi při rekonstrukci hodnot, což by výrazně snižovalo efektivitu navrženého algoritmu. Rozhodli jsme se proto využít blokovou kompresi, která je dnes implementovaná hardwarově na každém GPU a pro shader využívající komprimovanou strukturu se dekomprese provádí zcela transparentně bez výkonnostních ztrát.

Bloková komprese je založena na koherenci hodnot sousedních vzorků. Čtvercový blok 16 vzorků je vždy zpracováván najednou a je z něj vytvořeno bitové pole reprezentující hodnoty všech vzorků v bloku. Při kompresi se v bloku najdou mezní hodnoty (minimum a maximum) a mezi těmito mezemi je lineární interpolací nalezeno několik dalších mezihodnot. Každému z 16 vzorků bloku je pak přiřazen index jedné z hodnot, které je nejblíže. Blok je následně uložen jako bitové pole, do kterého se uvedou hodnoty maxima a minima a indexy všech 16 vzorků. V praxi se používá 8 hodnot pro jeden blok, které je možné indexovat pomocí 3 bitů. Meze jsou vyjádřeny osmibitovým číslem, takže dohromady zabere jeden blok 8 bytů (16·3 + 2·8 bitů), což jsou průměrně 4 bity na vzorek. Hodnoty ve vstupní výškové mapě jsou vyjádřeny pomocí 32 bitů, takže jsme dosáhli kompresního poměru téměř 8:1. Samozřejmě jde o ztrátovou kompresi, proto nemůžeme stejně radikální postup použít vždy, resp. ne vždy bude takto zvolený postup podávat uspokojivé výsledky. Ztrátovost blokové komprese na osmibitové výškové mapě ukazuje Obrázek 4.11. Další výsledky s kompresí jsou shrnuty v odstavci 4.6.1.



Obrázek 4.11: Bloková komprese použitá na osmibitovou výškovou mapu (vlevo) zanesla do obrázku chyby, které jsou znázorněny jasem po 64násobném přesvětlení (vpravo). Velikost maximální odchylky činí 0.8%.

4.4.2 Generování normálové mapy

Aby bylo možné terén korektně vystínovat, je třeba definovat normálové vektory v místech, kde je počítán lokální světelný model. Protože původním vstupem našeho programu (před fází předzpracování) je výšková mapa, nejsou informace o normálách povrchu zpravidla k dispozici.

Z uniformně vzorkovaného terénu je možné rekonstruovat přibližnou hodnotu normálového vektoru na základě diferencí sousedních hodnot. To je možné provádět během zobrazování, nebo je možné vygenerovat normálovou mapu, ve které budou již normálové vektory spočítány (opět fáze předzpracování). My jsme nejdříve vyzkoušeli první variantu – rekonstrukci normály za běhu. To jsme provedli na základě postupu popsaného v (99), který zjednodušuje výpočet normály na výpočet diferencí.

Normála $\overline{n}$ ve vrcholu V_0 je korektně definována váženým průměrem normál trojúhelníků, jichž je vrchol V_0 součástí. Tedy přesněji

$$\overline{n} = \frac{\sum_{i=1}^{n} \varphi_i \overline{n_i}}{\sum_{i=1}^{n} \varphi_i},$$

kde n reprezentuje počet trojúhelníků, které obsahují vrchol V_0, $\overline{n_i}$ představuje normálu trojúhelníku s indexem i, a φ_i je úhel tohoto trojúhelníku při vrcholu V_0. Vyjdeme z představy pravidelné pravoúhlé mřížky, v níž je každý vrchol obklopen čtyřmi ploškami (jsou to sice čtverce, ale v naší představě si je vždy doplníme vhodnou diagonálou). Vrchol v této mřížce tedy obklopují tři trojúhelníky s obecně různými úhly. Pokud projekcí do horizontálně položené roviny zanedbáme prostorovou odlišnost jejich úhlů, můžeme v uvedeném vztahu vynechat váhy jednotlivých normál a zůstane nám vztah pro aritmetický průměr:

$$\overline{n} = \frac{1}{n} \sum_{i=1}^{n} \overline{n_i}.$$

Nyní nám tedy zbývá pouze spočítat normály jednotlivých trojúhelníků. Víme, že vektorovým součinem dvou vektorů získáme třetí vektor kolmý na prvé dva. Toho využijeme při získání normály trojúhelníka tak, že vyjádříme dvě jeho strany jako vektory. K tomu si označíme okolní vrcholy ve stejném pořadí jako trojúhelníky V_1-V_4. Jednotlivým vrcholům můžeme dát souřadnice v prostoru dané měřítkem výškové mapy, tj.

$$V_0 = [x, h(x, z), z],$$

$$V_1 = [x + d, h(x, z) + h_1, z],$$

$$V_2 = [x, h(x, z) + h_2, z - d],$$

$$V_3 = [x - d, \boldsymbol{h}(x, z) + \boldsymbol{h}_3, z],$$

$$V_4 = [x, \boldsymbol{h}(x, z) + \boldsymbol{h}_4, z + d],$$

kde $\boldsymbol{h}(x, z)$ reprezentuje výšku terénu v bodě $[x, z]$, d je vzdálenost dvou vzorků (V_0 a V_i) ve světových souřadnicích a $\boldsymbol{h}_i$ představuje diferenci mezi výškou vrcholu V_i a výškou vrcholu V_0. Důležité hrany trojúhelníků vyjádříme za pomoci uvedených vrcholů jako vektory

$$\overline{v_i} = V_i - V_0$$

pro $i \in \{1 \dots 4\}$, tedy po dosazení

$$\overline{v_1} = \left(d, \boldsymbol{h}_{1,0}\right),$$

$$\overline{v_2} = (0, \boldsymbol{h}_2, -d),$$

$$\overline{v_3} = \left(-d, \boldsymbol{h}_{3,0}\right),$$

$$\overline{v_4} = (0, \boldsymbol{h}_4, d),$$

odkud za pomoci vektorového součinu vyjádříme normály trojúhelníků jako

$$\overline{n_1} = \overline{v_1} \times \overline{v_2} = (-d\boldsymbol{h}_1, d^2, d\boldsymbol{h}_2),$$

$$\overline{n_2} = \overline{v_2} \times \overline{v_3} = (d\boldsymbol{h}_3, d^2, d\boldsymbol{h}_2),$$

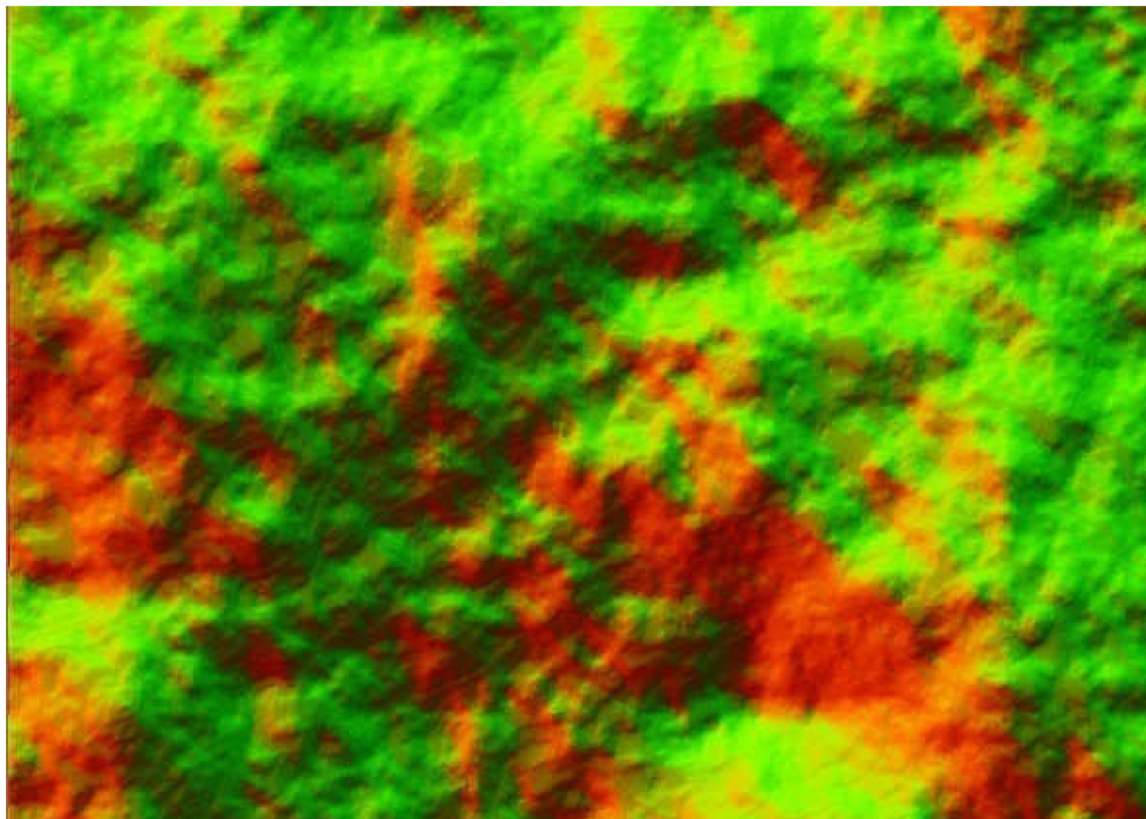
$$\overline{n_3} = \overline{v_3} \times \overline{v_4} = (d\boldsymbol{h_3}, d^2, -d\boldsymbol{h_4}),$$

$$\overline{n_4} = \overline{v_4} \times \overline{v_1} = (-d\boldsymbol{h_1}, d^2, -d\boldsymbol{h_4}).$$

Nyní se můžeme vrátit zpět a dosadit normály trojúhelníků do vztahu pro normálu vrcholu, tedy

$$\overline{n} = \frac{1}{4}\sum_{i=1}^{4}\overline{n_i} = \left(\frac{d}{2}(\boldsymbol{h_3} - \boldsymbol{h_1}), d^2, \frac{d}{2}(\boldsymbol{h_4} - \boldsymbol{h_2})\right).$$

Tento postup jsme vyzkoušeli pro rekonstrukci normály z výškové mapy během zobrazování, ale narazili jsme na dvě negativní vlastnosti. Tou první je velmi nepřesný výsledek, který pramení jednak z našeho zjednodušeného odvození, jednak z toho, že bereme v úvahu jenom lokální chování terénu. Druhou negativní vlastností je velká potřeba čtení dat z textury. Vzhledem k tomu, že výška samotná vyžaduje velké množství opakovaných čtení, aproximace normály pomocí 5 hodnot výšek zajistí přetížení texturovacích jednotek, a tedy i výrazné snížení výkonu. Výhoda postupu je ta, že nepotřebujeme normálovou mapu, která si může vyžádat velké množství paměti.



Obrázek 4.12: Normálová mapa. Červená složka zobrazuje natočení ve směru osy x, zelená ve směru osy z. Ypsilonová komponenta je vyjádřena implicitně.

Jedním možným řešením nedostatečného výkonu je vygenerování dočasné normálové mapy při načítání dat. Otázkou je, k čemu by takové chování bylo. Ušetřili bychom sice nějaké místo na disku, ale to nás dnes zpravidla tíží nejméně. My jsme se rozhodli proto vygenerovat normálovou mapu ve fázi předzpracování současně s mapou reziduí. Vzhledem k tomu, že na tento proces máme téměř neomezené množství času, zvolili jsme výkonově poměrně náročný postup, který pro velké terény může trvat i několik hodin. Vychází ze stejného základu jako postup předchozí, ale namísto aritmetického průměru normál okolních trojúhelníků se berou v úvahu jejich prostorové úhly, které jsou použity jako váhy pro jednotlivé normály. Postup je navíc pro každý vzorek několikrát opakován, přičemž při každé iteraci je zohledněno jinak velké okolí (použit jiný parametr d a jsou použity jiné vrcholy). Z takovýchto dílčích normál je opět spočítán vážený průměr, pro který byly váhy hledány experimentálně, aby bylo dosaženo vizuálně dobrých výsledků. Tímto postupem zohlední normála širší charakter terénu a ne pouze lokální chování, zároveň dojde k jistému vyhlazení. Výsledek generování normál ukazuje Obrázek 4.12.

Výsledná normála je vždy vektor v trojrozměrném prostoru. Uchovávat všechny tři jeho komponenty je však zbytečné. Když totiž vektor normalizujeme, můžeme původní normálu rekonstruovat na základě pouze dvou složek x a z a vztahu

$$x^2 + y^2 + z^2 = 1.$$

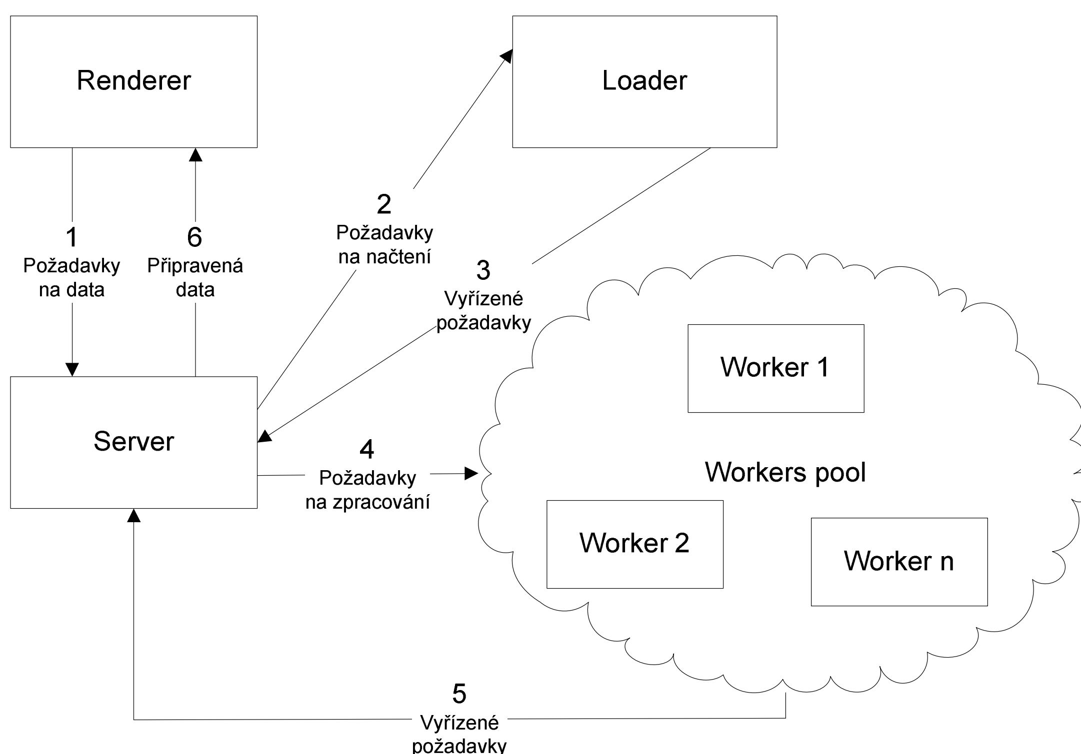
Rovnice má sice dva kořeny, ale my víme, že y je vždy kladné (to plyne z orientace terénu). Díky tomu nám stačí uchovávat dvě komponenty. Na ty používáme stejnou blokovou kompresi jako na mapu reziduí, pouze na každou složku zvlášť. Jedna normála je tak definována pomocí 8 bitů (oproti původním 96). Ztrátovost komprese nás v případě normál většinou nezajímá. Ani prvotní redukce 32bitové reprezentace komponenty do 8bitové není okem rozeznatelná.

4.5 Zpracování neomezeného terénu

Jedním z hlavních cílů, které jsme si v úvodu položili, bylo zobrazování potenciálně nekonečného terénu, tedy takového, který nemůžeme celý najednou uložit do operační paměti. U některých implementací dříve zmíněných algoritmů najdeme různá řešení. V (100) najdeme řešení pro LOD nad obecnými daty, tedy nejen pro terén. Hoppe (18) rozděluje terén na čtvercové bloky a vytváří progressive mesh nad každým blokem zvlášť. Původní implementace clipmap na texturu terénu (51) využívala rozdělení na malé čtvercové bloky, které byly stránkovány do paměti celé najednou, jakmile byla data vyžadována pro zobrazování. Díky progresivnímu načítání od nižších úrovní detailu bylo vždy zajištěno, že jsou vždy k dispozici alespoň nějaká data pro zobrazování. Rozdělení na čtverce využil i Pajarola (23) při výstavbě databáze složené z částí terénu, která byla optimalizována pro rychlé zpracování požadavků na dvourozměrné souvislé úseky. Lindstrom a Pascucci (37) navrhli řešení více zohledňující koherenci při čtení dat z disku i z paměti, ve kterém využívají charakteru stromové struktury, která je procházena odshora. V (101) najdeme jednoduché paralelizované řešení se zdrojovým kódem, které však nezajišťuje včasnou dostupnost dat.

Protože od počátku uvažujeme kroky takové, aby bylo možné použít algoritmus do počítačové hry, chtěli jsme se v našem řešení oprostit o těsnou vázanost na terén. Pokud má být takový systém někde použit, dá se očekávat, že nebude jednoúčelový, ale že bude využit kromě načítání terénu také k obecnému plynulému zajišťování dat. Proto se nám nezalíbila myšlenka explicitního rozdělení dat do čtverců, které nemusí mít v případě jiných typů dat smysl (například hudební záznam).

Druhým důležitým požadavkem bylo využití potenciálu většího počtu výkonných jednotek (procesorů a jader), resp. hardwarových vláken (například herní konzole Xbox 360 má tři výkonné jednotky a každá z nich má podporu dvou hardwarových vláken). Obecný příklon dnešního hardwaru k paralelismu je zřejmý. Vychází především z konstrukčních problémů při dosahování vyšších frekvencí procesorů. Zvyšování počtu výkonných jednotek je jednou z možností dalšího růstu výkonu a dá se předpokládat, že tato tendence zvyšování počtu jader či procesorů v jednom stroji se ještě řadu let udrží.



Obrázek 4.13: Rozdělení činnosti programu do různých vláken a jejich komunikace.

Navrhli jsme a implementovali systém pro načítání a správu dat, který si dále stručně popíšeme a který má tyto vlastnosti:

1. Paralelní zpracování načítaných dat.

2. Nezávislost na charakteru načítaných dat.

3. Optimalizace přístupu k zdrojovému médiu vzhledem k sériovému čtení.

4. Vlastní správa paměti.

5. Blokující i neblokující přijímání požadavků.

6. Možnost přiřazovat požadavkům prioritu a měnit ji dokud nejsou data načtena.

K rozboru jednotlivých vlastností se dostaneme postupně souběžně s popisem různých částí systému. Základní rozdělení práce do samostatných vláken ukazuje Obrázek 4.13, ze kterého budeme vycházet při popisu činnosti. Číselně jsou v něm označeny kroky vedoucí od vytvoření požadavku na data až po jejich vyzvednutí. Obrázek také reflektuje názvy vláken, jak je používáme v dalším textu.

4.5.1 Renderer

Renderer představuje to vlákno programu, které se stará o zobrazování. Kvůli návaznosti na handle okna to často bývá zároveň hlavní vlákno, které se stará o všechno řízení a logiku. Činnost tohoto vlákna může být velmi různorodá a komplikovaná, nás však zajímá pouze to, že je to toto vlákno, kde vzniknou požadavky na nějaká data. Ty jsou způsobeny různou činností uživatele uvnitř virtuálního světa. V našem konkrétním případě je to pouze pohyb, ale obecně to mohou být i reakce na jiné události. Například rozdělání ohně ve virtuálním světě si vyžádá načtení animace hoření, která dosud nebyla použita, nebo už byla odstraněna z paměti, protože již dlouho nebyla použita. Důležité je dokázat rozpoznat situaci, která si vyžádá načtení nějakých dat, ještě dříve, než nastane, aby bylo možné skrýt zpoždění při načítání. Například v naší demonstrační aplikaci děláme podle pohybu pozorovatele predikci oblastí terénu, které budou zanedlouho vidět, a které je tedy třeba připravit. Oblasti jsou čtvercové o velikosti 128*128 vzorků a v souboru je každá uložena jako souvislý blok o velikosti 8 kB pro mapu reziduí a 16 kB pro normálovou mapu.

Druhou prací Renderer vlákna je zpracování příchozích dat, která vznikla na základě nějakého předchozího požadavku. V zjednodušené podobě je činnost Renderer vlákna složena ze čtyř kroků, které se provádí neustále dokola:

1. Aktualizace scény.

2. Vytvoření požadavků na data.

3. Zpracování hotových požadavků.

4. Zobrazení scény.

Do aktualizace scény spadá veškerá reakce na vstup od uživatele, přepočet uběhlého času, pohyb kamery, … V případě by se zde prováděla činnost související s veškerou herní logikou atp. Vytváření požadavků předchází predikce potřebných dat. Jednak se na základě zaktualizované scény zjistí, co bude v blízké budoucnosti potřeba a ještě o to nebylo požádáno, ale také se zkontroluje, zda není některá data třeba zajistit urgentně, či zda naopak dříve vytvořený požadavek neztratil význam (uživatel se zachoval jinak, než jak dříve předurčila predikce). Všechny tyto informace se převedou na formu požadavků, případně aktualizací požadavků, a jsou předány do fronty na zpracování Server vláknem. Důležité je, že proces vložení požadavku do

fronty je neblokující, takže Renderer se okamžitě může věnovat další práci a o tyto požadavky se dále nemusí starat. V další fázi se zkontroluje fronta příchozích dat. Ta obsahuje kladně vyřízené požadavky, které byly dříve vzneseny. Renderer vybírá požadavky, resp. data s nimi spojená, a kopíruje je do svých interních struktur, odkud probíhá zobrazování. Aby se zajistila plynulost, provede pouze omezené množství těchto operací za jeden cyklus, tj. za jeden snímek. Nakonec provede zobrazení celé scény na základě zaktualizovaných dat.

4.5.2 Server

Hlavním vláknem našeho načítacího systému je *Server*. Server řídí veškeré zpracování požadavků od jejich přijetí až po jejich zpětné vyzvednutí žadatelem (tím je v našem případě Renderer). Kromě toho provádí správu paměti a aktualizaci všech struktur podle toho, jak jsou požadavky zpracovávány.

Aby bylo možné přidávat požadavky neblokujícím způsobem, je hlavní vstupní fronta řešena pomocí dvou kopií, z nichž každá obsahuje jiné záznamy. Tímto způsobem je možné z fronty číst i do ní zapisovat zároveň ze dvou různých vláken. Takový přístup je nutný, aby zobrazovací vlákno nemuselo nikdy čekat kvůli synchronizaci. Hlavní činnost serveru spočívá v třízení požadavků a jejich předávání správným vláknům pro další zpracování. Pro takové řízení obsahuje server několik různých front a pomocných vyhledávacích struktur, aby bylo možné co nejrychleji přistupovat k různým požadavkům.

Každý požadavek zahájí z pohledu Serveru svůj život tím, že je přidán do vstupní fronty. Odtud jsou požadavky pravidelně vybírány a je kontrolována jejich existence. Každý požadavek má svůj implicitní jednoznačný identifikátor, podle kterého je možno jej rozpoznat. V případě, že požadavek již existuje, je pouze aktualizován na základě nově nastavených parametrů. V opačném případě je třeba vytvořit zvláštní strukturu, jež reprezentuje požadavek interně. Ta obsahuje mimo jiné i buffer, do nějž budou později uložena načtená data. Zde přichází na řadu správce paměti, který je zažádán o přidělení bloku potřebné velikosti. Velikost závisí na typu požadovaných dat a je specifikována data providerem, o kterém se ještě zmíníme v odstavci 4.5.3. Správce paměti rozhodne, zda je možné specifikovanému požadavku vyhovět. Pokud ne (nedostatek paměti), Server se pokusí uvolnit nějakou paměť. To dělá opakovaně tak dlouho, dokud mu správce paměti nepřidělí požadovaný blok, nebo dokud je co uvolňovat. Uvolňována jsou data spojená s požadavky, které již byly vyřízeny a vyzvednuty Rendererem. Z takových se vezme vždy ten požadavek, který byl nejdéle nepoužit (LRU metoda).

Ať již byl požadavek vytvořen nový, nebo pouze aktualizován, je zařazen na konec fronty, jež odpovídá jeho prioritě (tu stanoví Renderer). Zvlášť jsou skládány požadavky, které byly označeny za urgentní – takové musí být zpracovány co nejdříve. Po zařazení požadavku do jakékoli fronty je probuzeno (pokud spí) Loader vlákno, které vybere jeden z připravených požadavků, a ten vyřídí tím, že načte požadovaná data ze správného souboru na disku. Více tuto část rozvedeme v odstavci 4.5.3.

Požadavky, které byly úspěšně načteny Loader vláknem, jsou opět řazeny do další fronty, která už je řízena tradičním FIFO přístupem. V této frontě čekají na zpracování některým z Worker

vláken. S přidáním požadavku do fronty je některé z Worker vláken probuzeno, aby se o požadavek postaralo. Práce Worker vlákna je více popsána v odstavci 4.5.4.

Zpracované požadavky se zařadí do výstupní fronty, kde jsou připraveny pro vyzvednutí Renderer vláknem. Renderer má možnost si vzít požadavky jeden po druhém, nebo vybrat jeden konkrétní, který je pro něj důležitý. To je možné provést i blokujícím způsobem, kdy žadatel čeká, dokud není požadavek zpracován. Vyzvednutí požadavku z této fronty obnáší jeho dočasné uzamknutí. S požadavkem není možné manipulovat – nemůže být tedy odstraněn ani modifikován, dokud jej Renderer explicitně neodemkne, čímž říká, že všechna data načetl a už je nepotřebuje. Po odemknutí zpracovaného požadavku, je tento přesunut do úložného prostoru, ve kterém jsou odloženy všechny zpracované požadavky. Pokud by byl požadavek někdy znovu vznesen a nacházel by se v tomto úložišti, nemusel by procházet běžným procesem zpracování, ale rovnou by byl vložen do fronty pro vyzvednutí Rendererem. Z tohoto úložiště jsou také vybírány požadavky pro uvolnění, pokud je potřeba získat nějakou paměť. Můžeme zmínit také to, že požadavek je možné označit za persistentní, nebo jednorázový. Pokud je požadavek persistentní, není nikdy uvolněn z paměti (alespoň do chvíle, než je mu tato vlastnost Rendererem odebrána). Naopak jednorázový požadavek je dán do zvláštního prostoru, z něhož jsou požadavky uvolněny při nedostatku paměti nejdříve. Kromě posledních popsaných úložišť znázorňuje všechny fáze zpracování požadavku Obrázek 4.13. Pořadí fází je dáno očíslováním šipek.

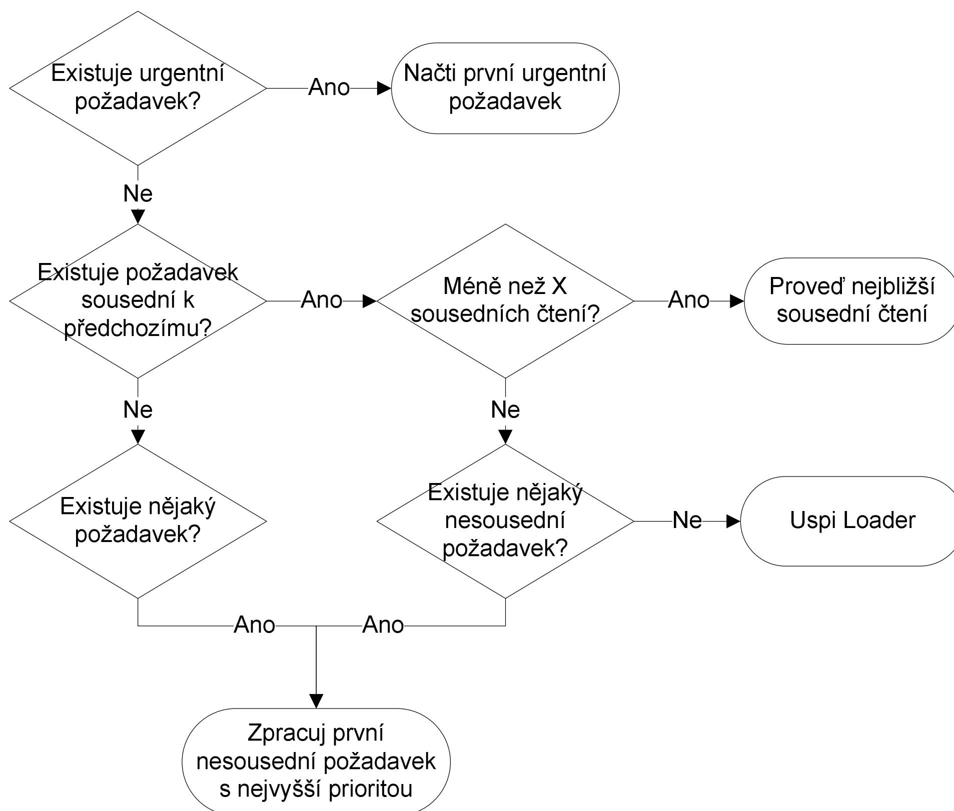
Zmínili jsme se, že jsme použili vlastního správce paměti. K tomu jsme se rozhodli z více důvodů. Předně naše správa paměti zahrnuje jedinou alokaci a jedinou dealokaci. Za běhu programu se tak nemusí stále provádět časově náročné operace alokace paměti. Také v kterémkoli okamžiku přesně víme, kolik paměti máme k dispozici. Toto je obzvláště výhodné při programování na herní konzole, kde je poměrně málo operační paměti a je důležité rozvrhnout správně její využití. Navíc je na konzolích informace o dostupné paměti obecně známa, proto je vůbec možné optimalizovat pro konkrétní „rozpočet“. Konečně využitím jediného souvislého bloku paměti do jisté míry zamezíme její fragmentaci (nebo ji alespoň dostaneme pod naši kontrolu).

4.5.3 Loader

Loader provádí načítání požadovaných dat ze správného souboru. S tím souvisí tzv. data provider, o kterém si nyní povíme. *Data provider* představuje implementaci abstraktní třídy, která reprezentuje nějaký druh nebo formát dat. Pro každý typ dat, která chceme načítat pomocí našeho načítacího systému, musíme definovat specializovaný provider, který bude umět data načítat a zpracovávat. Příkladem v naší demonstrační aplikaci je mapa reziduí, z níž v čase chceme načítat různé úseky. Specializovaný data provider umí pro zadaný požadavek rozhodnout, kolik paměti si daný požadavek vyžádá, umí načíst potřebná data ze souboru a umí data zpracovat. V případě mapy reziduí není žádné zpracování potřeba, ale kdybychom kupříkladu chtěli generovat normálovou mapu za běhu, musel by specializovaný data provider pro normálovou mapu načítat data z mapy reziduí a z nich počítat normály. Tento návrh zcela oprošťuje univerzální systém pro správu dat od znalosti dat, se kterými pracuje. Může být tedy snadno použitelný do mnoha různých projektů. Instance data provideru při své inicializaci mimo

jiné otevře soubor, se kterým je asociovaná, čímž zabrání opakovanému otvírání a zavírání při jednotlivých čteních.

Loader má za úkol pouze data načíst, k čemuž využívá funkce data provideru spojeného s požadavkem. Významná je volba požadavku, který se má v danou chvíli Loaderem zpracovat. Tato část je optimalizována s ohledem na sériové čtení zdrojového média (počítá se s pevným diskem nebo optickým médiem). Nejdříve je zkontrolována fronta urgentních požadavků – ty musí být zpracovány co nejdříve, bez ohledu na sériovost čtení. Pokud tedy existuje nějaký urgentní požadavek, je zpracován podle FIFO kritéria. Pokud žádný urgentní neexistuje, je zohledněno, z jakého souboru byl načítán poslední požadavek. Protože požadavky jsou třízeny také podle souboru a pořadí v souboru, je snadné nalézt požadavek, který čeká ve frontě na načtení a je mu asociován stejný soubor, z jakého bylo provedeno poslední čtení. Přitom jsou však posuzovány jen takové požadavky, jejichž data jsou v souboru umístěna až za předchozím čtením. Jinak by se stejně musela čtecí hlava disku vrátit zpět a narušila by sériový přístup. Tato čtení prováděná ze stejného souboru jsou však limitována definovaným parametrem, aby se zajistilo, že se dostanou i jiné soubory na řadu. My jsme použili hodnotu 6, tj. maximálně 6 čtení z jednoho souboru, další je provedeno z jiného, pokud takový požadavek existuje. Celý proces rozhodování lépe popisuje Obrázek 4.14.



Obrázek 4.14: Proces rozhodování při volbě požadavku ke zpracování Loaderem.

Je také důležité zmínit, že existuje pouze jedno Loader vlákno. To proto, že paralelní načítání dat z jednoho média by mohlo výrazně zbrzdit celý proces. Ačkoli operační systém, případně hardwarový řadič, by mohl požadavky utřídit sám, my se na takovou funkcionalitu nechceme spoléhat.

4.5.4 Worker

Jak naznačuje Obrázek 4.13, načítací systém zahrnuje několik Worker vláken. Jsou to vlákna, která zpracovávají již načtená data. Jako běžnou práci Workeru si můžeme představit dekompresi dat nebo jen jednoduchou deserializaci z bloku, který byl načten Loaderem. Zpracování jednotlivých požadavků musí být zcela nezávislé na jiných požadavcích, aby bylo možné využít paralelismu. Toto je místo, kde se dá plně uživit výpočetní výkon několika hardwarových vláken. My jsme do přiložené demonstrační aplikace žádnou konkrétní práci Workeru nepřidělili, protože jak mapa reziduí, tak normálová mapa i výšková mapa jsou uchovány v souboru přesně v tom formátu, v jakém jsou následně čteny grafickou kartou. Ačkoli je snadné nadefinovat data provider pro normálovou mapu tak, aby zrekonstruoval normály terénu za běhu, naznali jsme, že je to kontraproduktivní – výsledky nejsou tak kvalitní, jako když se připraví ve fázi předzpracování, a zatížíme zbytečně systém, který ještě více jádry nedisponuje.

Počet Worker vláken, která se vytvoří, závisí na nastavení načítacího systému (stejně jako mnohé další parametry). Je možné toto rozhodnutí ponechat také na systému samotném, který v takovém případě vytvoří o dvě Worker vlákna více, než kolik jich je hardwarově podporovaných na daném stroji. Pořadí, v jakém jsou požadavky zpracovány, je dáno pořadím, v jakém jsou načteny Loaderem, který je vkládá do FIFO struktury.

4.6 Měření a výsledky

V tomto odstavci bychom rádi shrnuli praktické výsledky, kterých jsme dosáhli. Nejprve se podíváme, jak věrně se nám daří reprezentovat vstupní data, a následně provedeme testy rychlosti pro různé konfigurace.

4.6.1 Rekonstrukce výšek

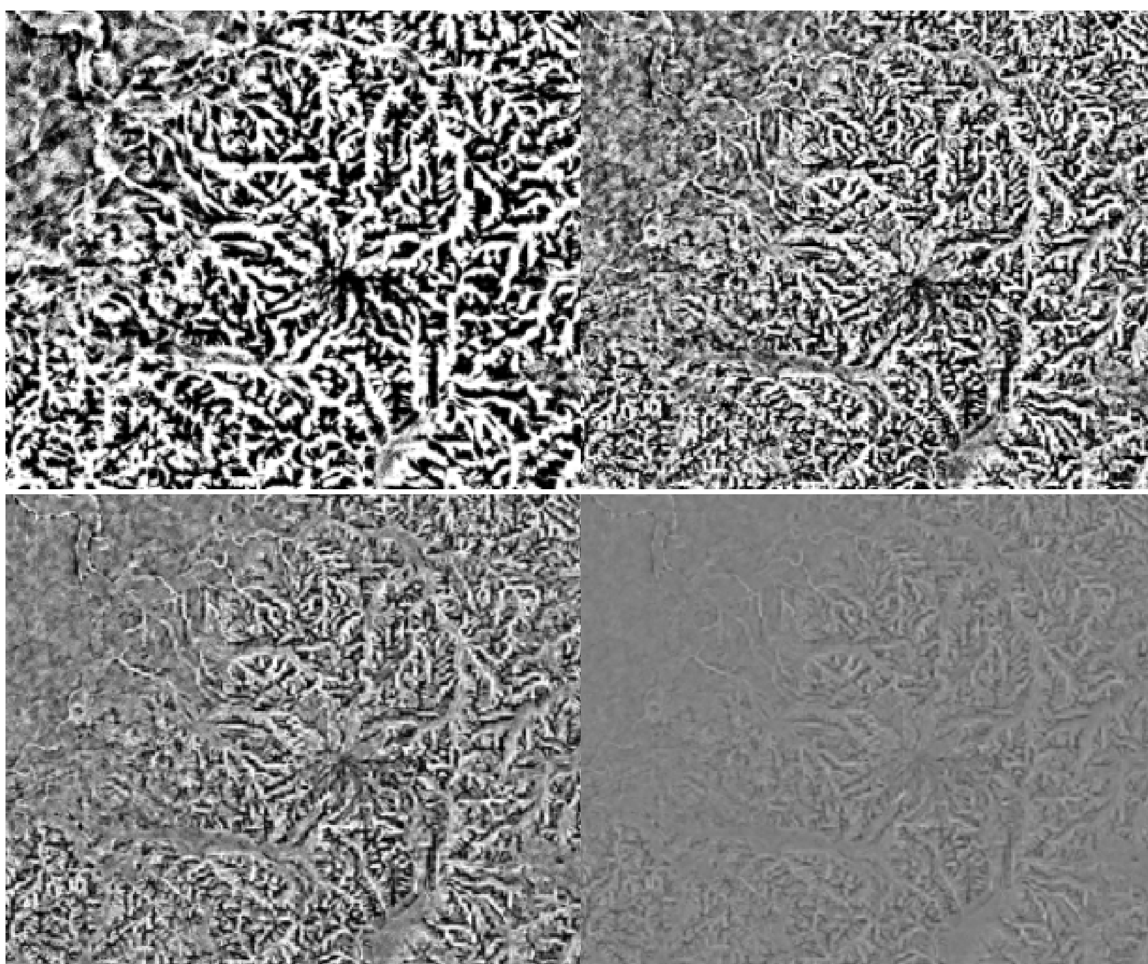
Použití dvouúrovňové reprezentace společně s blokovou kompresí mapy reziduí šetří velké množství paměti, ale zároveň vede k informačním ztrátám. Většinou se jedná především o ztrátu některé vysokofrekvenční informace, ke které dojde při ořezání hodnot reziduí do zvoleného rozsahu a pak znovu při blokové kompresi. Velikost ztrát, resp. chybovost reprezentace, závisí na charakteru dat a zvolených parametrech, tzn. například povolený rozsah reziduí nebo rozlišení výškové mapy nižší úrovně. Není snadné najít parametry, které bychom mohli použít na obecný terén, proto je třeba při generování mapy reziduí všechny výsledky prověřit. Může se ukázat, že pro vysokou degradaci detailu dat není vhodné v konkrétní situaci naši reprezentaci použít. Z našich zkušeností však usuzujeme, že pro krajiny, které se ve hrách běžně vyskytují a které jsou designérem pečlivě upravovány podle potřeb hry, je struktura dobře použitelná. Negativem může být dlouhý čas potřebný na generování dat, která potřebuje designér vidět ihned po aplikaci úprav. K těmto potřebám by bylo nutno navrhnout optimalizovaný proces, který by

prováděl aktualizaci reprezentace rychleji, než to dělá náš pomocný nástroj, jenž byl určen výhradně pro off-line předzpracování dat.

Dále se budeme zajímat o tyto parametry, které nepřímo určují věrnost reprezentace:

1. Rozsah výšek terénu
2. Rozsah hodnot reziduí
3. Rozlišení výškové mapy nižší úrovně
4. Počet bitů na uchování rezidua
5. Počet bitů na uchování hodnot výšky

Mezi tyto parametry by se měla také zařadit členitost terénu, ale ta je jen velmi špatně vyčíslitelná, proto budeme spíše zkoumat, jaké výsledky dostáváme s konkrétními parametry pro zvolená data. K posouzení jsme vybrali reálná data, která reprezentují lokaci Puget Sound ve státě Washington. Vstupní výšková mapa měla 8 192 × 8 192 vzorků, každý reprezentující plochu 2 m × 2 m. Rozsah hodnot výšek činil 3 000 m. Z této výškové mapy byla generována mapa o nižším rozlišení, postupně 64 až 512 pixelů, a s každou byly zkoumány výsledky zvlášť. Pro hodnoty výškové mapy bylo použito 16bitové a 32bitové reprezentace, avšak mezi těmito nebyl rozpoznán žádný rozdíl. Rezidua byla vždy reprezentována pomocí 8 bitů, ale díky blokové kompresi stačily pouze 4 bity na vzorek. 8 bitů reprezentuje 256 různých hodnot, jejichž věrnost reprezentace závisí na rozsahu, který výškám přidělíme. Zkoušeli jsme různé rozsahy a porovnávali se změnami v rozlišení výškové mapy. Některé z výsledků zachycuje Obrázek 4.15.



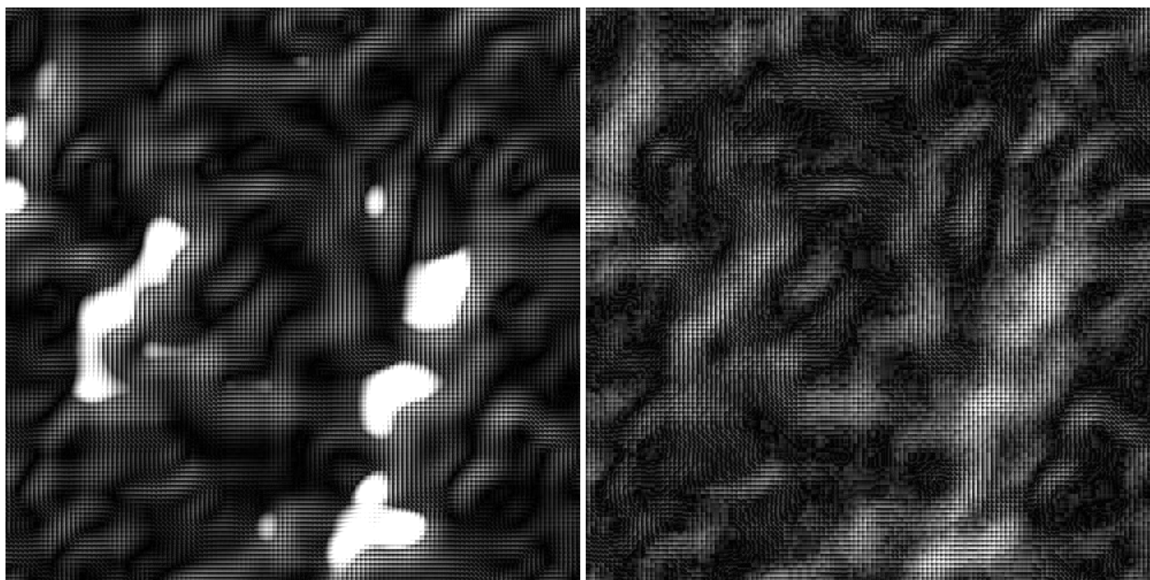
Obrázek 4.15: Mapy reziduí pro oblast Puget Sound vygenerované za použití různých parametrů. Vlevo nahoře maximální reziduum 10 m a rozlišení výškové mapy 64 pixelů. Vpravo nahoře maximální reziduum zvýšeno na 30 m. Vlevo dole rozlišení výškové mapy zvětšeno na 256 pixelů. Vpravo dole je výšková mapa opět 256 pixelů, ale rozsah reziduí zvýšen na 100 m. Zcela bílé a zcela černé oblasti představují místa, kde jsou rezidua příliš velká na zachycení zvoleným rozsahem.

Na obrázku je patrné, jak velký vliv má na výsledek povolený rozsah hodnot reziduí a rozlišení výškové mapy. Pro nižší rozlišení potřebujeme velký rozsah reziduí, abychom byli schopni reprezentovat rozsah hodnot v oblasti jednoho vzorku výstupní výškové mapy. Na obrázku jsou patrna místa (úplně černá a bílá), kde rozsah reziduí nestačil. Na druhou stranu zvýšení rozsahu má za následek jeho méně věrnou reprezentaci, neboť na uchování rezidua máme pouze 8 bitů (větší počet si nemůžeme dovolit, pokud chceme šetřit pamětí). Na nepříliš zvrásněných mapách jsme dosáhli přijatelných výsledků s použitím výškové mapy o 32 krát menším rozlišení než originál. Pro některé mapy by bylo lepší využít poměr rozlišení 16, ale tím se zvyšují i paměťové nároky. Konkrétní srovnání paměťových nároků s jinými přístupy ukazuje Tabulka 4.3. Při srovnání je však třeba brát v úvahu odlišné vlastnosti i kvalitou podávaných výsledků.

Rozměr mapy	Originál	Naše reprezentace,	Naše reprezentace,	Clipmapy

		16:1	32:1	
1 024	4 MB	528 kB	516 kB	8 MB
4 096	64 MB	8 448 kB	8 256 kB	16 MB
16 384	1 GB	~ 132 MB	~ 129 MB	24 MB
32 768	4 GB	~ 528 MB	~ 516 MB	28 MB

Tabulka 4.3: Srovnání paměťových nároků naší struktury v různých poměrech rozlišení (16:1 a 32:1) s naivním přístupem a clipmapami o rozměru 1 024 vzorků.



Obrázek 4.16: Diference rekonstruovaných a originálních hodnot. Vlevo použit rozsah reziduí 10 m, vpravo 240 m. Je vidět, že zatímco malý rozsah reziudí nebyl dostačující pro jejich zachycení v některých oblastech, velký rozsah způsobil větší nepřesnosti po celé ploše. MSE v prvním případě činí 3,7 m, ve druhém případě 0,6 m. Z obrázků je také patrný důsledek blokové komprese („kostičkovost“).

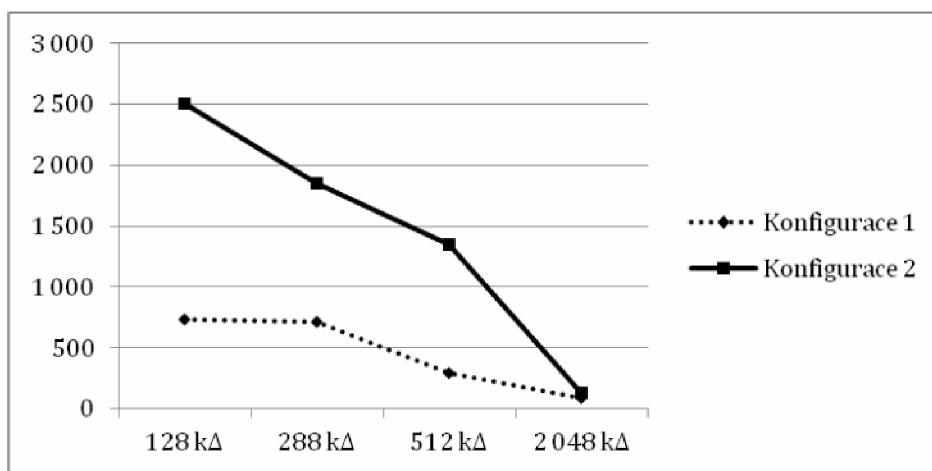
Pro získání přesnějších údajů jsme také vytvořili komparátor, ve kterém jsme rekonstruovali data a porovnávali s originálními hodnotami. Tím jsme dostali jasnou představu nejen o potřebném rozsahu reziduí, ale o celkové nepřesnosti reprezentace. Vybrali jsme k tomuto účelu silně zvrásněný terén vygenerovaný jako Perlinův šum o rozměrech 1 024 × 1 024 vzorků. Graficky znázorněné diference mezi rekonstruovanými a originálními hodnotami pro odlišná nastavení Obrázek 4.16.

Nejlepších výsledků jsme dosáhli v tomto případě při použití rozsahu reziduí 60 m, pro který činila průměrná kvadratická odchylka 0,41 m. V naší demonstrační aplikaci jsme použili čtvercový terén o šířce 16 384 vzorků, kde každý vzorek reprezentuje 1 m^2, a maximálním rozsahu výšek 1 000 m. Pro takový terén jsme použili výškovou mapu o šířce 512 vzorků (opět poměr 32:1) a rozsah reziduí 30 m. Jak ukazuje Tabulka 4.3, snížili jsme paměťové nároky z 1 GB na 128 MB a průměrná kvadratická odchylka činí 0,2 m. Protože maximální odchylka představuje až 2 m, může být v intencích designéra provést zpětnou úpravu terénu takovou, aby se inkriminovaná místa reprezentovala přesněji. Předpokládáme, že takové řešení je přípustné a není příliš náročné. Dále, jelikož zvolený postup potlačuje ostré hrany, může být též řešením

domodelování nezachycených prvků pomocí přidaných objektů, které se umístí na požadované místo v krajině. Totéž platí pro převisy, jeskyně a podobné krajinné útvary.

4.6.2 Výkon

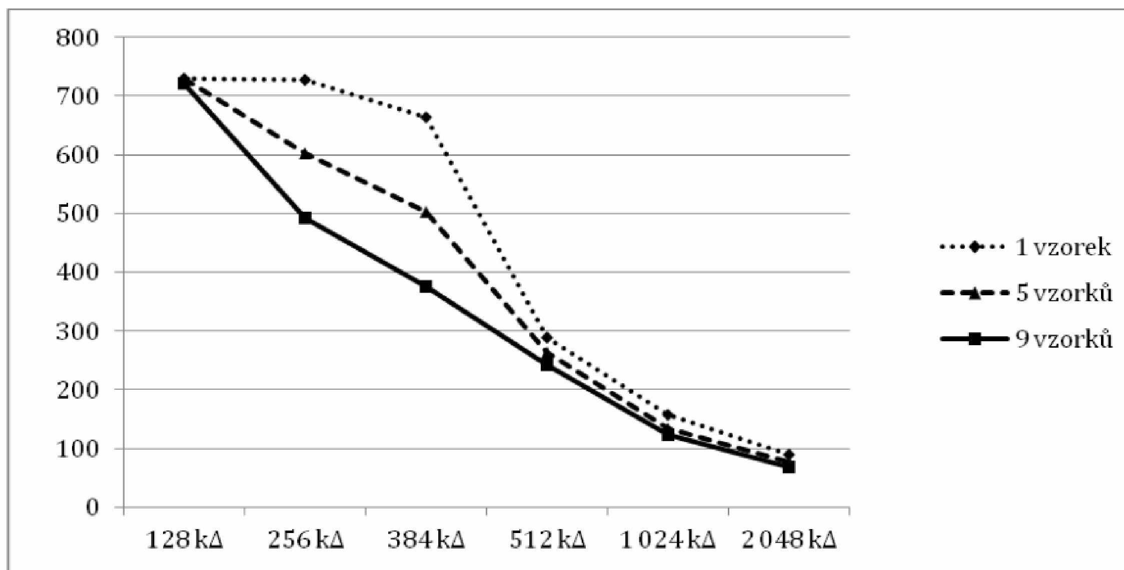
Všechny výsledky, které nabízíme ke srovnání, byly pořízeny na dvou strojích. Prvním byl jednojádrový Athlon 64 3000+ se 2 GB RAM a grafickou kartou Radeon HD 2900 XT. Dále jej budeme označovat jako Konfiguraci 1. Druhým byl čtyřjádrový Core 2 Quad s taktem 2,66 GHz, 8 GB RAM a grafickou kartou GeForce 8800 Ultra, který označíme jako Konfiguraci 2. Metoda zobrazování terénu, kterou jsme zde prezentovali, není téměř vůbec závislá na rychlosti procesoru, proto výše zobrazovací frekvence odráží výkon grafické karty. Naproti tomu načítání a zpracování dat je operace čistě závislá na rychlosti CPU a čtení dat z disku.



Obrázek 4.17: Graf závislosti počtu snímků za vteřinu na počtu trojúhelníků v síti.

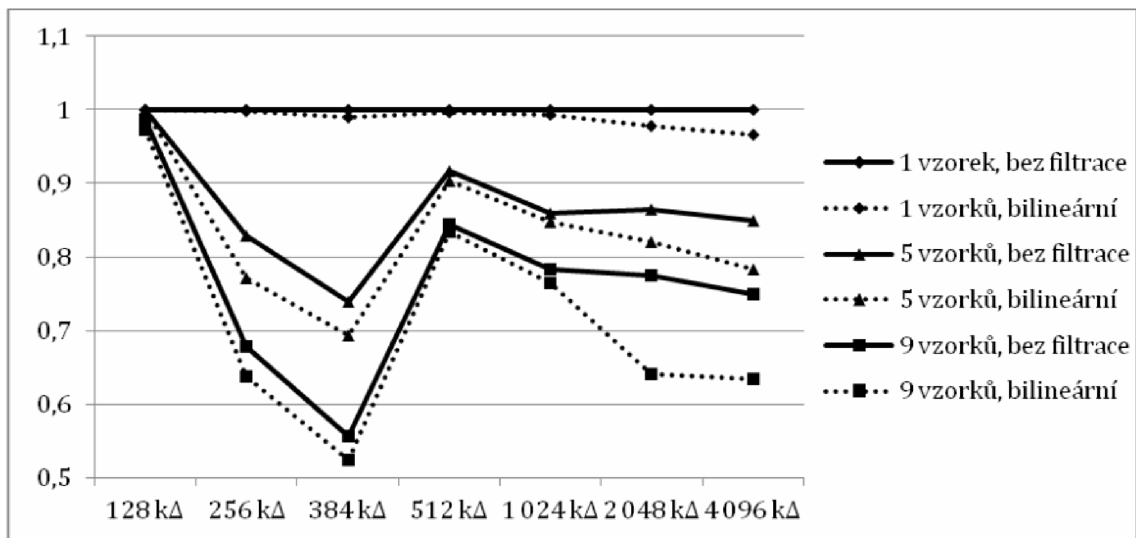
Hlavní faktor, na který jsme se zaměřili při testování, je potlačení aliasu, čehož se dá docílit vyšší úrovní vzorkování mapy reziduí, nebo vyšší hustotou sítě ringmapy. Proto nabízíme několik grafů, z nichž je patrné, jak tyto parametry ovlivňují výkon. Obrázek 4.17 představuje graf závislosti zobrazovací frekvence na hustotě zobrazované sítě. Dodejme, že hodnoty uvedené v grafu nezohledňují ořezání pohledovým jehlanem. Počty skutečně kreslených trojúhelníků jsou přibližně třetinové až čtvrtinové. Nicméně ve všech testech je tento podíl stejný. Také je dobré zmínit, že ač jsme uvedli, že zobrazování je závislé pouze na grafické kartě, pro vysoké frekvence zobrazování to neplatí, neboť ovladače grafické karty provádí nějakou činnost na CPU, která se v těchto frekvencích projeví natolik, že silně zbrzdí výkon grafické karty. To je patrné u Konfigurace 1 při zobrazování menšího počtu trojúhelníků.

Obrázek 4.18 ukazuje velmi obdobný graf, ale tentokrát namísto různých konfigurací počítače porovnává různě náročná vzorkování. Výsledky můžeme interpretovat tak, že do jistého množství trojúhelníků byl výkon silně brzděn procesorem, případně nebyl vliv texturových instrukcí tak velký. S přibývajícím počtem trojúhelníků se však zvyšuje masivní zátěž grafické karty, která se projevuje prudkým poklesem výkonu. Je také vidět, že k tomuto poklesu dojde u náročnějších technik dříve.



Obrázek 4.18: Graf závislosti počtu snímků za vteřinu na počtu trojúhelníků v síti pro různé způsoby vzorkování. Pro hodnoty v grafu použita Konfigurace 1.

Obdobné závěry můžeme vyvodit také z pohledu na graf, který ukazuje Obrázek 4.19. Je postavený na stejných datech, ale rozšířen o další typy vzorkování. Ukazuje nám, jaký dopad má na danou situaci (počet trojúhelníků) volba jiné vzorkovací techniky, než je ta základní s jedním nefiltrovaným vzorkem. Graf toho však ukazuje více. Můžeme si všimnout velmi zajímavé tendence projevující se u většího množství trojúhelníků, kdy rapidně roste rozdíl mezi bilineární filtrací a přímým vzorkováním bez filtrace. Výkon bilineárního vzorkování pěti vzorků dokonce klesá až téměř na úroveň techniky s devíti vzorky bez filtrace. Domníváme se, že tento jev je způsoben intenzivním využitím vyrovnávací paměti pro textury. Zobrazovaná síť je pro takovéto množství trojúhelníků již tak hustá, že vzorky mapy reziduí se překrývají pro sousední vrcholy, i v rámci jednoho vrcholu. To patrně vede k méně častému čtení přímo z paměti, protože hledaná hodnota je nalezena ve vyrovnávací paměti. Užití bilineárního filtrování však způsobuje vynucené čtení okolních vzorků, které ve vyrovnávací paměti být nemusí. Předpokládáme, že to je také důvodem náhlého srovnání výkonu různých technik pro síť s 512 tisíci trojúhelníky, kdy patrně dojde k překročení jisté meze, za kterou jsou již vrcholy příliš blízko sebe. Z toho můžeme vyvodit závěr, že se nám z hlediska vizuálních výsledků v takovou chvíli nemůže vyplatit užít bilineární filtrování, protože vrcholy sítě jsou položeny hustěji než vzorky mapy reziduí, a nadbytečná filtrace by dokonce provedla nechtěné vyhlazení. Je také vidět, že pro výškovou mapu daného rozlišení nemá hustší síť s více než 512 trojúhelníky žádný význam. Přitom si uvědomme, že v danou chvíli zobrazujeme pouze asi 150 tisíc trojúhelníků. Tomu také odpovídá pěkná zobrazovací frekvence 300 snímků za vteřinu pro Konfiguraci 1, resp. 1 300 pro Konfiguraci 2.



Obrázek 4.19: Graf ukazuje poměr výkonu různých vzorkovacích technik vůči základní technice používající 1 nefiltrovaný vzorek. Pro hodnoty v grafu použita Konfigurace 1.

Všechny naměřené výsledky byly získány za použití mapy reziduí o rozlišení 16 384 × 16 384 vzorků a hrubé výškové mapy o rozlišení 512 × 512 vzorků. Každý vzorek mapy reziduí představoval 1 m^2, tudíž jsme se volně procházeli po ploše o celkové rozloze 256 km^2. Z ní však byla v každý okamžik vidět pouze menší část daná maximální vzdáleností dohledu, která byla pevně nastavena na 3 800 m. Na mapu reziduí byla použita čtvercová textura o velikosti 8 192 texelů, pro kterou jsme s využitím blokové komprese potřebovali 32 MB paměti grafické karty. Další 1 MB si vyžádala hrubá výšková mapa. Maximální převýšení mapy činilo 1 000 m a rozsah reziduí byl zvolen 30 m. Průměrná kvadratická odchylka v tomto případě činila 0,47 m. Bez použití násobného vzorkování jsme byli schopni pohybovat se krajinou rychlostí 100 kmh^{-1} bez znatelnějších vizuálních defektů, přičemž zobrazovací frekvence se blížila 1 300 snímků za vteřinu (Konfigurace 2). Za použití devítinásobného vzorkování mapy reziduí byla geometrie zcela stabilní i při rychlostech převyšujících 1 000 kmh^{-1}. Za takových podmínek jsme byli schopni udržet rychlost zobrazování nad 250 snímky za vteřinu (Konfigurace 1).

4.7 Budoucí práce

Dělali jsme velké množství pokusů, ale ne se všemi se nám podařilo dostat tak daleko, jak bychom rádi. Zobrazování terénu je obsáhlý a komplikovaný úkol, zvláště pokud si dáme cíle, které byly stanoveny v úvodu. Mnohé oblasti zůstávají nedořešeny a nabízí široké pole možností navázání na tuto práci. V následujících odstavcích chceme shrnout některé body případného dalšího postupu a předem navrhnout možná řešení.

4.7.1 Modifikace terénu

Modifikovatelnost terénu byla jedním ze tří hlavních požadavků na volbu řešení zobrazování. Při návrhu jsme tento požadavek mnohokrát zohlednili a věříme, že jsme dosáhli dostatečné volnosti. Avšak v přiložené demonstrační aplikaci faktu možné modifikovatelnosti využito není,

data jsou tam po částech nahrána do paměti a udržována ve statické podobě. Na základě zvolené reprezentace si přesto troufáme tvrdit, že modifikace terénu by bylo možné dosáhnout pomocí přímé úpravy mapy reziduí.

Oproti běžné výškové mapě má naše reprezentace jisté limitace. Rezidua jsou pouze v určitém rozsahu a přesnosti dané použitím 8 bitů a současnou kompresí. To znamená, že pokud maximální odchylka rezidua činí 15 m, nemůžeme terén modifikovat libovolně, ale pouze v rámci této odchylky. Přestože 15 m je hodně, není jisté, zda pro dané místo terénu bude k dispozici. Samotné původní hodnoty reziduí jsou již udány v rámci této odchylky, a pokud se nějaký vzorek nachází na její hranici, není možné jej za tuto hranici volně posunout. Další limitací je přesnost reprezentace. Pro maximální odchylku 15 m je celkový rozsah hodnot rezidua roven 30 m, které jsou vyjádřeny pomocí 256 různých hodnot. Z toho vyplývá, že dvě sousední hodnoty pomyslné stupnice jsou od sebe vzdáleny přibližně 0,12 m, a tedy každá změna se dá provést se zajištěnou přesností 6 cm. Pro větší rozsahy reziduí získáme samozřejmě nižší přesnost. Konečně jistá omezení přináší i bloková komprese, pro kterou musíme každou šestnácticí vzorků aktualizovat společně a přepočítat ji tak, aby se hodnoty co nejvíce přiblížily požadovaným, ať již modifikovaným, či nemodifikovaným. Při změně mezní (maximální či minimální) hodnoty uvnitř bloku totiž dojde k celkové změně reprezentace hodnot v bloku.

Z uvedeného je patrné, že ač je modifikace principielně možná, není zdaleka tak snadná jako v případě modifikace obyčejné výškové mapy. Má své limitace a záleží na požadavcích kladených aplikací, jak ji implementovat. Zcela jistě bude dostatečná například pro vojenské hry, které počítají s menšími úpravami terénu na základě explozí. V případě drastičtějších změn terénu by bylo nutné modifikovat i výškovou mapu nižší úrovně, což by si však vyžádalo přepočet reziduí celého bloku a několika okolních, které by závisely na měněné hodnotě. Taková činnost by zřejmě musela být rozložena do více snímků.

4.7.2 Potlačení aliasu

Přestože jsme věnovali značné úsilí potlačení nepříjemného efektu vlnění geometrie způsobeného nekonzistentním vzorkováním mapy reziduí při pohybu, stále zůstává převzorkování mapy reziduí nejlepším řešením. V odstavci 4.6.2 jsme ukázali praktické důsledky vyšších úrovní vzorkování na rychlosti zobrazování. Zvláště pro terén, který budeme modifikovat a nemůžeme jej tedy optimalizovat ve fázi předzpracování, je důležité využít co nejvíce možností pro potlačení zmíněného efektu. Nabízíme zde několik řešení, které by bylo možné dále více zkoumat.

Prvním z návrhů je pokládání ringmapy do předem definovaných pozic. Tyto pozice můžeme definovat pomocí diskrétní mřížky, nebo pomocí vztahu k poslední pozici ringmapy. Důležité je, aby se ringmapa neposouvala spojitě s pozorovatelem, ale aby se pohybovala v diskrétních krocích. Kdybychom toho dosáhli, můžeme pomocí triviálního morfování zajistit plynulý přechod z jedné geometrie na druhou. To by silně ovlivnilo poměr texturových a aritmetických instrukcí. Problém tohoto přístupu vyplývá z nepravidelné struktury ringmapy. Jednotlivým vrcholům ringmapy v jednom umístění je totiž potřeba nalézt odpovídající vrcholy v druhém umístění, mezi kterými se morfování provede. Tento úkol se ukázal být značně netriviální, neboť síť ringmapy má v různých částech různou hustotu a my potřebujeme vytvořit párovací schéma (ne

nutně bijekci), které by každému vrcholu přiřadilo nějakého reprezentanta nacházejícího se blízko ve smyslu světového souřadného systému. Takové schéma by navíc muselo být generalizované, aby šlo použít na různé situace, případně by se takových dala vytvořit sada a párování pro každé z nich by se uložilo v podobě textury. Vyřešení tohoto úkolu by mohlo mít velmi pozitivní vliv na kvalitu zobrazování, resp. rychlost při dané kvalitě.

Není těžké si povšimnout, že nepříjemný efekt se projevuje především na siluetách. Proto naším dalším návrhem je provést dodatečné dělení struktury ringmapy v geometry shaderu, kde podle normály vrcholů trojúhelníku můžeme provést násobné dělení hrany podezřelé z podílení se na siluetě. Pokud budeme podávat trojúhelníky grafické kartě s informacemi o sousedech, můžeme siluety dokonce označit přesně. Kromě toho můžeme adaptivně dělit i jiné trojúhelníky podle rozdílu výšek jejich vrcholů. Otázkou je, jakého výkonu bychom dosáhli v souladu s našimi experimenty popsanými v odstavci 4.2.3.2. Výkon by mohl být ještě nižší než při velmi intenzivním převzorkování mapy reziduí.

Další možností řešení aliasu je použití imposterů, tj. dvourozměrných zástupných objektů, kterými nahradíme vzdálené části terénu. Impostery se vytváří zobrazením nějakého objektu nebo skupiny objektů do textury. V našem případě by se do textury zobrazila vzdálená část terénu v nějakém úhlu kolem pozorovatele. Na pokrytí plného úhlu by nám mohlo stačit 8 až 16 imposterů, neboť by byly použity až pro větší vzdálenosti. Při zobrazování se objekt nebo skupina objektů, která se dříve zobrazila do imposteru, nahradí jediným velkým čtyřúhelníkem, na který se promítne získaný imposter. Protože imposter se nemusí vytvářet každý snímek znovu, je možné věnovat jeho tvorbě větší množství času, takže skýtá možnost lepších vizuálních výsledků než přímé zobrazování. Pro terén by nám to mohlo přinést hned několik vylepšení. Předně bychom naprosto potlačili vlnění geometrie v dálce, zobrazený terén by mohl být detailnější a konečně bychom tak mohli zvýšit dohledovou vzdálenost bez újmy na rychlosti zobrazování. Pokud předpokládáme rychlost pohybu chodce nebo běžce, můžeme udržet jeden imposter platný po dobu stovek až tisíců snímků. Problémy s impostery bývají často spojeny se správným napojením na objekty, které jsou zobrazovány přímo. Plně funkční implementace nemusí být snadná, ale zcela určitě představují impostery možnost dalšího postupu a rozšíření naší metody.

4.7.3 Mapování textury na terén

Prozatím používáme na terén barvu, kterou načteme z jednorozměrné textury na základě výšky terénu v daném místě. Je to tedy obdoba barvení používaná v kartografii. Tento postup však nepodává nikterak důvěryhodné výsledky. Proto by bylo vhodné zaměřit se v další práci na aspekt dodání realističnosti pomocí barev.

Jednou z možností je pokrýt celý terén jedinou velkou texturou, kterou připraví designér. Pro mapování je možno využít již vytvořeného načítacího frameworku s rozdělením terénu na čtverce. Vedle mapy reziduí a normálové mapy by se tak načítala i barevná textura, což by znamenalo výrazně vyšší paměťové nároky i větší zatížení systému při načítání, neboť barevná informace musí být většího rozlišení než geometrie či normály.

Jiným řešením je použití tradičních clipmap podle (52). Při omezení se na Shader Model 4.0 bychom je mohli lehce implementovat pomocí pole textur, jako to dělá (102). Oproti předchozímu řešení bychom s clipmapami ušetřili velké množství paměti, což je zásadní především pro konzolové hry. Obě dosud zmíněná řešení však vyžadují objemné množství dat, pravděpodobně přípustné spíše tam, kde je skutečná neopakující se textura přímo vyžadována charakterem aplikace (zobrazování reálné krajiny).

Jinou možností je syntetické generování textury na základě nějakých předloh. Například turbulentní funkce se často používá pro narušení pravidelnosti při vzorkování jinak pravidelného vzoru. Turbulentní funkci bychom mohli použít i my při vzorkování barevného přechodu. Častěji se však používá při indexaci menších předloh, které se opakují v terénu, ale na které je aplikována lokální modifikace. Možností je celá řada, velmi záleží na koncovém použití, protože ne každý způsob poskytuje dostatečnou kontrolu pro designéra. S tímto postupem souvisí také míchání více vrstev textur. Je například možné předdefinovat barevná schémata odpovídající druhu povrchu (písek, sníh, tráva, bláto,…) a pomocí syntézy s bezbarvou texturou a případnou úpravou dalšího lokálního detailu dojít k velmi realistickým výsledkům.

Do této práce se již texturování terénu nevešlo, neboť stojí stranou zvoleného zobrazovacího postupu a různé metody mapování textury lze zpravidla použít na kterýkoli algoritmus. Volba tedy stojí spíše na potřebě vynucené danou aplikací. Pro vizuální vjem je však textura velmi důležitá, a bylo by vhodné některou z navržených cest v budoucnu použít pro vytvoření ucelenějšího řešení.

4.7.4 Jiná vylepšení

Kromě již zmíněných větších celků stojí za úvahu řada menších úprav či experimentů. Jedním z nich je například změna interpolačního schématu použitého pro generování a rekonstrukci výšek terénu. V současné době používáme bilineární interpolaci 4 sousedů, což má výhodu velmi snadného vyjádření. Zvláště díky současnému velkému využití texturovacích instrukcí by bylo možné nahradit lineární interpolaci kupříkladu kosinovou. Výpočet kosinu je sice výpočetně náročný, ale mohl by se snadno schovat za latence pamětí. Jinou možností je zohlednit i okolní vzorky pro kubickou či jinou interpolaci, nebo dokonce použít nějaké z dělících schémat popsaných v paragrafu 3.2. To by ovšem znamenalo ještě větší množství vzorků, takže zřejmě další omezení výkonu. Závěry se však dají udělat až po patřičném otestování.

Načítací framework má své limitace a v současné chvíli se silně projeví, když nejsou data dostatečně rychle načtena a zobrazování probíhá bez aktualizovaných informací. Například absence normálové mapy způsobí zcela černá nebo špatně osvětlená místa terénu. Tento problém by šel řešit pomocí velmi malé textury, která by nesla informace o tom, které části terénu jsou platné a které ne. Ve vertex shaderu by se načetl z této textury status pro zpracovávaný vrchol a podle informace v textuře by se mohl změnit způsob jeho zpracování. Například by se použila pouze hrubá výšková mapa bez mapy reziduí, pokud by tato ještě nebyla načtena. Takovýto postup bychom také rádi do budoucna implementovali.

5 Závěr

V předložené práci jsme provedli rozsáhlý průzkum dříve i nyní používaných postupů pro zobrazování terénu. Ukázalo se, že většinu z nich je dnes obtížné implementovat efektivně, tj. za použití současných grafických akcelerátorů. Starší metody jsou většinou založeny na hierarchických strukturách a jejich iterativním procházení, které se špatně implementuje na masivně paralelním GPU.

V kapitole 3 jsme se seznámili s teorií parametrických ploch a subdivision surfaces. Ukázali jsme, jaké může být jejich využití při zobrazování terénu, a jaké možnosti jejich implementace skýtá nový design grafických karet, především díky rozšíření o geometry shader jednotku. Narazili jsme zde na vážné problémy z hlediska výkonu, které naznačují, že současný hardware ani přes jeho nové možnosti a stále se rapidně zvyšující výpočetní sílu není zralý na to, aby metody adaptivního dělení geometrie byly plně implementovány na grafické kartě v takovémto měřítku. K tomuto závěru jsme dospěli na základě většího množství experimentů, které jsme v práci také rozebrali. Věříme, že na tuto část naší práce bude možné v blízké době navázat.

Hlavním tématem práce bylo nalezení vhodného řešení interaktivního zobrazování terénu, které by splňovalo tři základní cíle, jež jsme stanovili v úvodu. Má pracovat nad obecně neomezeným objemem dat, umožnit modifikaci dat během zobrazování a má být snadno škálovatelné, aby bylo použitelné v širším spektru konfigurací. Kapitolu 4 jsme věnovali podrobnému popisu cest, které jsme vyzkoušeli při hledání adekvátního řešení, a některých úskalí, která s tím byla spojena. Nakonec jsme navrhli inovativní postup a věnovali se jeho rozboru. S navrženým postupem jsme na grafické kartě Radeon HD 2900 XT dosáhli zobrazovací frekvence téměř 300 snímků za vteřinu při dohledové vzdálenosti 3 800 m na terénu o rozloze 256 km^2 a rozlišení jednoho vzorku na metr.

Kromě splnění zvolených hlavních cílů má naše řešení řadu dalších kladných vlastností. Předně velmi málo zatěžuje procesor. Samotné zobrazování je provedeno pouze dvěma voláními funkcí grafické knihovny. Zbytek práce odvádí grafická karta. Reprezentace dat, kterou jsme použili, má nízké paměťové nároky s ohledem na získanou funkcionalitu. Pro zobrazování dat, která se nevejdou do paměti, jsme navrhli a implementovali obecný asynchronně pracující načítací systém. Systém je optimalizovaný pro získávání dat z média se sériovým čtením, ale je zcela oddělen od zdrojů dat i způsobu jejich získávání. Úroveň jeho abstrakce tedy umožňuje nasazení do široké škály jiných projektů. Tento systém zohledňuje aktuální požadavky a provádí zpracování dat paralelně ve více vláknech. Obě hlavní komponenty, tedy načítací systém i zobrazování terénu, jsou snadno konfigurovatelné a umožňují aplikaci pro širokou škálu cílového hardware.

Je velmi obtížné porovnávat výsledky, které jsme získali, s výsledky jiných algoritmů. Všeříkajícím kritériem nemůže být ani počet trojúhelníků, na jaký algoritmus terén redukuje, ani počet trojúhelníků, které zobrazí za vteřinu. Například v porovnání s clipmapami produkuje náš

algoritmus nižší počet trojúhelníků na dosažení obdobně detailní reprezentace. Oproti algoritmu jako je ROAM naopak výrazně vyšší. Dále každá metoda potřebuje jiný čas na zpracování jednoho vrcholu. V případě té naší je výkon velmi závislý na výkonu texturovacích jednotek kvůli potřebě násobného čtení dat z textury. Tato závislost může představovat i v budoucnu limitující faktor, protože výpočetní výkon GPU roste rychleji než rychlost přístupu do paměti. Při porovnávání grafických výsledků zjistíme, že různé algoritmy produkují různě kvalitní reprezentaci pro stejný počet trojúhelníků. V argumentech bychom mohli pokračovat.

Je zřejmé, že srovnávat jednotlivé postupy je možné pouze na základě pevně stanoveného kriteria, resp. skupiny kriterií, kterým přidělíme určitou váhu. Volba takových kriterií pak závisí na účelu, za jakým výběr zobrazovacího řešení provádíme. Postup, který jsme presentovali zde, byl od počátku směrován pro nasazení do počítačových her, ve kterých se hráč může procházet ve velké virtuální krajině. Ačkoli naše řešení umožňuje řádově menší dohledové vzdálenosti než clipmapy, nabízí funkcionalitu, která je dle všech současných herních titulů důležitější, a to náhlou změnu zorného úhlu. Modifikace terénu se ve hrách zatím neusadily, ale jsme přesvědčeni, že je to pouze otázka technických možností současného hardware a volby správné metody. V tomto směru přicházíme s naším řešením jako návrhem na změnu, a zároveň jako alternativou pro metody dnes používané.

Literatura

1. **Garland, Michael a Heckbert, Paul S.** *Fast Polygonal Approximation of Terrains and Height Fields.* Pittsburgh, PA : Carnegie Mellon University. Computer Science Department, 1995. Technical Report CMU-CS-95-181.

2. **Kumler, Mark P.** *An intensive comparison of triangulated irregular networks (TINs) and digital elevation models (DEMs).* In Cartographica, 31 (2). 1994. stránky 1-48.

3. **McGuire, Morgan a Sibley, Peter G.** *A Heightfield on an Isometric Grid.* In ACM SIGGRAPH 2004 Sketches. New York, NY, USA : ACM, 2004. str. 141. ISBN 1-59593-896-2.

4. **Clark, James.** *Hierarchical Geometric Models for Visible Surface Algorithms.* In Communications of the ACM 19, 10. New York, NY, USA : ACM, 1976. stránky 547-554. ISSN 0001-0782.

5. **Heckbert, Paul S. a Garland, Michael.** *Survey of Polygonal Surface Simplification Algorithms.* In Siggraph 97 Course Notes. New York : ACM Press, 1997.

6. **Luebke, D., a další.** *Level of Detail for 3D Graphics.* 1st edition. [s.l.] : Morgan Kaufmann, 2002. ISBN 978-1558608382.

7. **Luebke, David.** *A Survey of Polygonal Simplification Algorithms.* Chapel Hill, NC, USA : University of North Carolina at Chapel Hill, 1997. Technical Report TR97-045.

8. **Luebke, David P.** *A Developer's Survey of Polygonal Simplification Algorithms.* In IEEE Computer Graphics and Applications, 21(3). Los Alamitos, CA, USA : IEEE Computer Society Press, 2001. stránky 24-35. ISSN 0272-1716.

9. **Huddy, Richard.** *Optimizing DirectX9 Graphics.* In Game Developers Conference. [s.l.] : AMD, Inc., 2006. Dostupné na adrese http://ati.amd.com/developer/gdc/2006/GDC06-Advanced_D3D_Tutorial_Day-Huddy-Optimization.pdf.

10. **Huddy, Richard**. *Graphics Performance.* In Develop Brighton - The ATI Developer Day. [s.l.] : Advanced Micro Devices, Inc., 2006.

11. **Fortune, Steven.** *Voronoi Diagrams and Delaunay Triangulations.* In Computing in Euclidean Geometry. Singapore : World Scientific, 1992. stránky 193-233.

12. **Suglobov, Vladislav I.** *Appearance-Preserving Terrain Simplification.* Moscow : Lomonosov Moscow State University, 2000. Technical Report.

13. **Hoppe, Hugues.** *Progressive Meshes.* In Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. New York, NY, USA : ACM SIGGRAPH, 1996. stránky 99-108. ISBN 0-89791-746-4.

14. **Hoppe, Hugues, a další.** *Mesh Optimization.* In Proceedings of the 20th annual conference on Computer graphics and interactive techniques. New York, NY, USA : ACM SIGGRAPH, 1993. stránky 19-26. ISBN 0-89791-601-8.

15. **Xia, Julie C. a Varshney, Amitabh.** *Dynamic View-Dependent Simplification for Polygonal Models.* In Proceedings of the 7th conference on Visualization '96. Los Alamitos, CA, USA : IEEE Computer Society Press, 1996. stránky 327-ff. ISBN 0-89791-864-9.

16. **Hoppe, Hugues.** *View-Dependent Refinement of Progressive Meshes.* In Proceedings of the 24th annual conference on Computer graphics and interactive techniques. New York, NY, USA : ACM Press, 1997. stránky 189-198. ISBN 0-89791-896-7.

17. **El-Sana, Jihad a Varshney, Amitabh.** *Generalized View-Dependent Simplification.* In Computer Graphics Forum 18 (3). [s.l.] : Blackwell Publishing, 1999. stránky 83-94.

18. **Hoppe, Hugues.** *Smooth View-Dependent level-of-detail Control and its Application to Terrain Rendering.* In Proceedings of the conference on Visualization '98. Los Alamitos, CA, USA : IEEE Computer Society Press, 1998. stránky 35-42. ISBN 1-58113-106-2.

19. **Hoppe, Hugues.** *Efficient Implementation of Progressive Meshes.* In Computers and Graphics 22 (1). 1998. stránky 27-36.

20. **Lindstrom, Peter, a další.** *Real-Time, Continuous LoD Rendering of Height Fields.* In Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. New York, NY, USA : ACM, 1996. stránky 109-118. ISBN 0-89791-746-4.

21. **Pajarola, Renato, Antonijuan, Marc a Lario, Roberto.** *QuadTIN: Quadtree based Triangulated Irregular Networks.* In Proceedings of the conference on Visualization '02. Washington, DC, USA : IEEE Computer Society, 2002. stránky 395-402. ISBN 0-7803-7498-3.

22. **Duchaineau, Mark, a další.** *ROAMing Terrain: Real-time Optimally Adapting Meshes.* In Proceedings of IEEE Conference on Visualization 1997. Los Alamitos, CA, USA : IEEE Computer Society Press, 1997. stránky 81-88. ISBN:1-58113-011-2.

23. **Pajarola, Renato.** *Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation.* In Proceedings of the conference on Visualization '98. Los Alamitos, CA, USA : IEEE Computer Society Press, 1998. stránky 19-26. ISBN 1-58113-106-2.

24. **Röttger, Stefan, a další.** *Real-Time Generation of Continuous Levels of Detail for Height Fields.* In Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization. 1998. stránky 315-322.

25. **Microsoft Corporation.** *DirectX Documentation.* [online] listopad 2007. Dostupné z: http://msdn2.microsoft.com/en-us/library/bb205066.aspx.

26. **Cline, David a Egbert, Parris K.** *Terrain Decimation through Quadtree Morphing.* In Transactions on Visualization and Computer Graphics 7, 1. Salt Lake City, UT : Sterling Wentworth, 2001. stránky 62-69. ISSN 1077-2626.

27. **Ulrich, Thatcher.** Continuous LOD Terrain Meshing Using Adaptive Quadtrees. *Gamasutra.* [Online] 28. únor 2000. [Citace: 7. červen 2007.] http://www.gamasutra.com/features/2000 0228/ulrich_01.htm.

28. **Ji, Junfeng, a další.** *Dynamic LOD on GPU.* In Proceedings of Computer Graphics International 2005. Beijing : Chinese Acad. of Sci. Lab. of Comput. Sci., 2005. stránky 108-114. ISBN 0-7803-9330-9.

29. **Pajarola, Renato.** *Overview of Quadtree-based Terrain Triangulation and Visualization.* Irvine : University of California. Information & Computer Science, 2002. Technical Report UCI-ICS-02-01.

30. **Evans, William, Kirkpatrick, David a Townsend, Gregg.** *Right Triangular Irregular Networks.* Tucson, AZ, USA : University of Arizona, 1997.

31. **Turner, Bryan.** Real-Time Dynamic Level of Detail Terrain Rendering with ROAM. *Gamasutra.* [Online] 3. duben 2000. [Citace: 13. červen 2007.] http://www.gamasutra.com/f eatures/20000403/turner_01.htm.

32. **Blow, Jonathan.** *Terrain Rendering at High Levels of Detail.* San Jose, CA : Game Developers Conference, 2000.

33. **Pomeranz, Alex A.** *ROAM Using Surface Triangle Clusters (RUSTiC).* Davis : University of California. Center for Image Processing and Integrated Computing, 2000. Master's thesis.

34. **Levenberg, Joshua.** *Fast View-Dependent Level-of-Detail Rendering Using Cached Geometry.* In Proceedings of the conference on Visualization '02. Washington, DC, USA : IEEE Computer Society, 2002. stránky 259-266. ISBN 0-7803-7498-3.

35. **He, Yefei a Cremer, James.** *Real-time Extendible-resolution Display of On-line Dynamic Terrain.* In Graphics Interface 2002. 2002. stránky 151-160.

36. **Lindstrom, Peter a Pascucci, Valerio.** *Visualization of Large Terrains Made Easy.* In Proceedings of the conference on Visualization '01. Washington, DC, USA : IEEE Computer Society, 2001. stránky 363-371. ISSN 1070-2385.

37. **Lindstrom, Peter a Pascucci, Valerio.** *Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization.* In IEEE Transactions on Visualization and Computer Graphics, 8(3). Piscataway, NJ, USA : IEEE Educational Activities Department, 2002. stránky 239-254. ISSN 1077-2626.

38. **Cignoni, Paolo, a další.** *BDAM – Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization.* In Computer Graphics Forum 22 (3). 2003. stránky 505-514.

39. **Cignoni, Paolo, a další.** *Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM).* In Proc. IEEE Visualization. 2003. stránky 147-155.

40. **Yuting, Ye a Guoping, Wang.** *View-dependent Real-time Terrain Rendering Using Static LOD.* Beijing : Peking University. Department of Computer Science and Technology, 2004. Technical Report.

41. **Boer, Willem H. de.** *Fast Terrain Rendering Using Geometrical MipMapping.* 2000. Nepublikovaný článek dostupný na adrese http://www.daimi.au.dk/~rip/cg04/GeoMipMapping.pdf.

42. **Ulrich, Thatcher.** *Rendering Massive Terrains Using Chunked Level of Detail Control.* In SIGGRAPH 2002 Course Notes. San Antonio, Texas : ACM SIGGRAPH, 2002.

43. **Celba, Pavel, a další.** *Unlimited Racer.* Univerzita Karlova. Matematicko-fyzikální fakulta. 2006. Softwarový projekt.

44. **Snook, Greg.** Simplified Terrain Using Interlocking Tiles. [autor knihy] Mark DeLoura. *Game Programming Gems 2.* 1st edition. [s.l.] : Charles River Media, 2001. Kapitola 4.2, stránky 377-383.

45. **Wagner, Daniel.** Terrain Geomorphing in the Vertex Shader. [autor knihy] Wolfgang Engel. *ShaderX^2: Shader Programming Tips and Tricks with DirectX 9.0.* [s.l.] : Wordware Publishing, 2003. Kapitola 1.3.

46. **Larsen, Bent Dalgaard a Christensen, Niels Jørgen.** *Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail.* In Journal of WSCG, 11(2). 2003. stránky 282-289.

47. **Livny, Yotam, Kogan, Zvi a El-Sana, Jihad.** *Seamless Patches for GPU-Based Terrain Rendering.* In Proceeding of WSCG 2007. 2007.

48. **Schneider, Jens a Westermann, Rüdiger.** *GPU-Friendly High-Quality Terrain Rendering.* In Journal of WSCG, 14(1-3). 2006. stránky 49-56. ISSN 1213-6972.

49. **Rupnik, Bojan.** *Rendering Large Terrains in Real-Time.* In CESCG 2007 Conference Proceedings. 2007.

50. **Vistnes, Harald.** GPU Terrain Rendering. [autor knihy] Michael Dickheiser. *Game Programming Gems 6.* [s.l.] : Charles River Media, 2006. Kapitola 5.5, stránky 461-472.

51. **Montrym, John S., a další.** *InfiniteReality - A Real-Time Graphics System.* In Proceedings of the 24th annual conference on Computer graphics and interactive techniques. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1997. stránky 293-302. ISBN 0-89791-896-7.

52. **Tanner, Christopher C., Migdal, Christopher J. a Jones, Michael T.** *The Clipmap - A Virtual Mipmap.* In Proceedings of the 25th annual conference on Computer graphics and interactive techniques. New York, NY, USA : ACM, 1998. stránky 151-158. ISBN 0-89791-999-8.

53. **Losasso, Frank a Hoppe, Hugues.** *Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids.* In ACM Transactions on Graphics, 23(3). New York, NY, USA : ACM, 2004. stránky 769-776.

54. **Holkner, Alex.** *Hardware Based Terrain Clipmapping.* 2004. Dostupný na adrese http://yallar a.cs.rmit.edu.au/~aholkner/rr/ah-terrain.pdf.

55. **Rose, Robert.** *Terrain Rendering using Geometry Clipmaps.* 2005. Dostupné z: http://www.ro bertwrose.com/cg/terrain/rose-geoclipmaps.pdf.

56. **Brettell, Nick.** *Terrain Rendering Using Geometry Clipmaps.* University of Canterbury. Department of Computer Science. Computer Graphics & Image Processing Research Group. 2005. Hons. Report. Vedoucí práce Dr R. Mukundan.

57. **Asirvatham, Arul a Hoppe, Hugues.** Terrain Rendering using GPU-Based Geometry Clipmaps. [editor] Matt Pharr a Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Boston : Addison-Wesley, 2005, stránky 27-46.

58. **Šrámek, Ondřej.** *Zobrazování terénních dat v reálném čase.* Praha : Univerzita Karlova. Matematicko-fyzikální fakulta, 2007. Diplomová práce. Vedoucí práce Mgr. Lukáš Maršálek.

59. **Makarov, Evgeny.** *Clipmaps.* [s.l.] : NVIDIA Corporation, 2007. White Paper.

60. **Clasen, Malte a Hege, Hans-Christian.** *Terrain Rendering Using Spherical Clipmaps.* In EuroVis 2006 – Proc. Eurographics / IEEE VGTC Symposium on Visualization. 2006. stránky 91-98.

61. **Clasen, Malte a Hege, Hans-Christian.** *Clipmap-based Terrain Data Synthesis.* In Simulation und Visualisierung 2007. San Diego : SDS Publishing House, 2007. stránky 385-398.

62. **Dachsbacher, Carsten a Stamminger, Marc.** *Rendering Procedural Terrain by Geometry Image Warping.* In Rendering Techniques 2004 (Proceedings of Eurographics Symposium on Rendering). 2004. stránky 103-110.

63. **Schneider, Jens, Boldte, Tobias a Westermann, Rüdiger.** *Real-Time Editing, Synthesis, and Rendering of Infinite Landscapes on GPUs.* In Conference on Vision, Modeling, and Visualization. 2006.

64. **Vlachos, Alex, a další.** *Curved PN Triangles.* In Symposium on Interactive 3D Graphics. New York, NY, USA : ACM, 2001. stránky 159-166. ISBN 1-58113-292-1.

65. **Nalasco, David.** *TRUFORM.* místo neznámé : ATI Technologies, 2001. White Paper.

66. **Moreton, Henry.** *Watertight Tessellation using Forward Differencing.* In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware. New York, NY, USA : ACM, 2001. stránky 25-32. ISBN 1-58113-407-X.

67. **Boubekeur, Tamy a Schlick, Christophe.** *Generic Mesh Refinement on GPU.* In Eurographics Workshop on Graphics Hardware. Los Angeles : ACM SIGGRAPH, 2005.

68. **Rayner, Mike.** Dynamic Level of Detail Terrain Rendering with Bézier Patches. *Gamasutra.* [Online] 21. březen 2002. [Citace: 29. červen 2007.] http://www.gamasutra.com/gdc2002/features/rayner/rayner_01.htm.

69. **Catmull, Edwin a Clark, Jim.** *Recursively generated B-spline surfaces on arbitrary topological meshes.* In Computer-Aided Design 10. 1978. stránky 350-355.

70. **Doo, Daniel a Sabin, Malcolm.** *Behavior of recursive division surfaces near extraordinary points.* In Computer-Aided Design 10 (6). 1978. stránky 356-360.

71. **Farin, G., Hoschek, J. a Kim, M.-S.** *Handbook of Computer Aided Geometric Design.* [s.l.] : Elsevier, 2002. Chapter 13, Interrogation of Subdivision Surfaces. ISBN 978-0-444-51104-1.

72. **Zorin, Denis a Schröder, Peter.** *Subdivision for Modeling and Animation.* Siggraph 2000. [s.l.] : ACM SIGGRAPH, 2000. Course Notes.

73. **Biermann, Henning, Levin, Adi a Zorin, Denis.** *Piecewise Smooth Subdivision Surfaces with Normal Control.* [s.l.] : ACM SIGGRAPH, 2000. ISBN 1-58113-208-5.

74. **Halstead, Mark, Kass, Michael a DeRose, Tony.** *Efficient, Fair Interpolation using Catmull-Clark Surfaces.* In Proceedings of the 20th annual conference on Computer graphics and interactive techniques. New York, NY, USA : ACM SIGGRAPH, 1993. stránky 35-44. ISBN 0-89791-601-8.

75. **Stam, Jos.** *Exact Evaluation Of Catmull-Clark Subdivision Surfaces At Arbitrary Parameter Values.* In Proceedings of the 25th annual conference on Computer graphics and interactive techniques. New York, NY, USA : ACM, 1998. stránky 395-404. ISBN 0-89791-999-8.

76. **Bolz, Jeffrey a Schröder, Peter.** *Rapid Evaluation of Catmull-Clark Subdivision Surfaces.* In Web3D '02: Proceeding of the seventh international conference on 3D Web technology. New York, NY, USA : ACM, 2002. stránky 11-17. ISBN 1-58113-468-1.

77. **Bolz, Jeffrey a Schröder, Peter.** Evaluation of Subdivision Surfaces on Programmable Graphics Hardware. *Caltech Multi-Res Modeling Group.* [Online] 2003. [Citace: 20. červenec 2007.] http://www.multires.caltech .edu/pubs/GPUSubD.pdf.

78. **Lee, Aaron, Moreton, Henry a Hoppe, Hugues.** *Displaced Subdivision Surfaces.* In Proceedings of the 27th annual conference on Computer graphics and interactive techniques. New York, NY, USA : ACM Press/Addison-Wesley, 2000. stránky 85-94. ISBN 1-58113-208-5.

79. **Huang, Xin, Li, Sheng a Wang, Guoping.** *A GPU Based Interactive Modeling Approach to Designing Fine Level Features.* In Proceedings of Graphics Interface 2007. New York, NY, USA : ACM, 2007. stránky 305-311. ISSN 0713-5424.

80. **Settgast, V., a další.** *Adaptive Tesselation of Subdivision Surfaces.* Braunschweig : University of Technology. Institute of Computer Graphics, 2004. Technical Report TUBS-CG-2004-05.

81. **Shiue, Le-Jeng, Jones, Ian a Peters, Jörg.** *A Realtime GPU Subdivision Kernel.* In ACM Transactions on Graphics, 24(3). New York, NY, USA : ACM, 2005. stránky 1010-1015. ISSN 0730-0301.

82. **Konečný, Juraj.** *Catmull–Clark Subdivision Surfaces on GPU.* In CESCG 2007 Conference Proceedings. 2007.

83. **Loop, Charles Teorell.** *Smooth Subdivision Surfaces Based on Triangles.* [s.l.] : The University of Utah. Department of Mathematics, 1987. Masters Thesis.

84. **Hoppe, Hugues, a další.** *Piecewise Smooth Surface Reconstruction.* In Computer Graphics 28. 1994. stránky 295-302.

85. **Brickhill, David.** Practical Implementation Techniques for Multi-Resolution Subdivision Surfaces. *Gamasutra.* [Online] 10. duben 2002. [Citace: 28. červen 2007.] http://www.gamas utra.com/features/20020410/brickhill_03.htm.

86. **Zorin, Denis a Kristjansson, Daniel.** *Evaluation of Piecewise Smooth Subdivision Surfaces.* In The Visual Computer, 18(5-6). Berlin : Springer, 2002. stránky 299-315. ISSN 0178-2789.

87. **Kim, Minho a Peters, Jörg.** *Realtime Loop subdivision on the GPU.* In ACM SIGGRAPH 2005 Posters. New York, NY, USA : ACM, 2005.

88. **Dyn, Nira, Levin, David a Gregory, John A.** *A Butterfly Subdivision Scheme for Surface Interpolation with Tension Control.* In ACM Transactions on Graphics 9 (2). New York, NY, USA : ACM, 1990. stránky 160-169. ISSN 0730-0301.

89. **Zorin, Denis, Schröder, Peter a Sweldens, Wim.** *Interpolating Subdivision for Meshes with Arbitrary Topology.* In Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. New York, NY, USA : ACM, 1996. stránky 189-192. ISBN 0-89791-746-4.

90. **Sharp, Brian.** Implementing Subdivision Surface Theory. *Gamasutra.* [Online] 25. duben 2000. [Citace: 11. červenec 2007.] http://www.gamasutra.com/features/20000425/sharp.h tm.

91. **Padrón, E. J., a další.** *Efficient Parallel Implementations for Surface Subdivision.* In Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization. Aire-la-Ville, Switzerland : Eurographics Association, 2002. stránky 113-121. ISBN 1-58113-579-3.

92. **Kobbelt, Leif.** *Sqrt(3) Subdivision.* In SIGGRAPH 2000 Conference Proceedings. 2000.

93. **Sußner, Gerd, Dachsbacher, Carsten a Greiner, Günther.** *Hexagonal LOD for Interactive Terrain Rendering.* In Vision, Modeling and Visualization 2005. 2005. stránky 437-444.

94. **Persson, Emil.** *ATI Radeon™ HD 2000 programming guide.* [s.l.] : Advanced Micro Devices, 2007. White Paper. Dostupné z: http://ati.amd.com/developer/SDK/AMD_SDK_Samples_May2007/Documentations/ATI_Radeon_HD_2000_programming_guide.pdf.

95. **Nehab, Diego, Barczak, Joshua a Sander, Pedro V.** *Triangle Order Optimization for Graphics Hardware Computation Culling.* In Proceedings of the 2006 symposium on Interactive 3D graphics and games. New York, NY, USA : ACM, 2006. stránky 207-211. ISBN 1-59593-295-X.

96. **Sander, Pedro V., Nehab, Diego a Barczak, Joshua.** *Fast Triangle Reordering for Vertex Locality and Reduced Overdraw.* In ACM SIGGRAPH 2007 papers. New York, NY, USA : ACM, 2007. ISSN 0730-0301.

97. **Evans, Francine, Skiena, Steven a Varshney, Amitabh.** *Optimizing Triangle Strips for Fast Rendering.* In Proceedings of the 7th conference on Visualization '96. Los Alamitos, CA, USA : IEEE Computer Society Press, 1996. stránky 319-326. ISBN 0-89791-864-9.

98. **El-Sana, Jihad, a další.** *Efficiently Computing and Updating Triangle Strips for Real-Time Rendering.* In Computer-Aided Design 32 (13). [s.l.] : Elsevier, 2000. stránky 753-772.

99. **Shankel, Jason.** Fast Heightfield Normal Calculation. [editor] Dante Treglia. *Game Programming Gems 3.* [s.l.] : Charles River Media, 2002. Kapitola 4.2, stránky 344-348.

100. **Cignoni, Paolo, a další.** *Batched Multi Triangulation.* In Proceedings IEEE Visualization. Minneapolis, MI, USA : IEEE Computer Society Press, 2005. stránky 207-214.

101. **Microsoft Corporation.** ContentStreaming Sample. *DirectX Developer Center.* [Online] 2007. [Citace: 8. listopad 2007.] http://msdn2.microsoft.com/en-us/library/bb204903.aspx.

102. **Dudash, Bryan.** *Texture Arrays (terrain).* [s.l.] : NVIDIA Corporation, 2007. White Paper.

103. **Persson, Emil.** *Depth In-depth.* [s.l.] : Advanced Micro Devices, 2007. White Paper. Dostupné z: http://ati.amd.com/developer/SDK/AMD_SDK_Samples_May2007/Documentations/Depth_in-depth.pdf.

Dodatek

Obsah CD

Binary\	Přeložený hlavní projekt Terra Artifex a pomocné nástroje.
Shaders\	HLSL shadery potřebné pro aplikace Terra Artifex a DDSViewer.
Heightmaps\	Data terénu pro zobrazování programem Terra Artifex.
Textures\	Textury použité programem Terra Artifex.
DXSetup\	Instalační balík rozšiřujících komponent DirectX.
Results\	Nasnímané obrazovky z běhu programu Terra Artifex a krátká demonstrační videa.
Sources\	Root složka zdrojových souborů. Projektové soubory MS Visual C++ 2005. Zdrojové kódy hlavní demonstrační aplikace Terra Artifex.
_compilation\	
Shaders\	HLSL shadery používané v projektech Terra Artifex a DDSViewer.
ContentStreamer\	Systém pro asynchronní načítání dat.
DDSViewer\	Program na vizualizaci datových souborů a prohlížení DDS souborů.
DXUT\	Rozšíření DirectX použité pro GUI. (Knihovna třetí strany.)
Globals\	Globální funkce používané ve více programech.
HeightMapProcessor\	Program na předzpracování dat výškové mapy.
PerlinNoiseGenerator\	Program na generování výškové mapy.
Resources\	Resources použité v projektu Terra Artifex.
TableGenerator\	Program na generování tabulek subdivision surfaces schémat.
Thesis\	Text diplomové práce v .pdf a .docx formátech.

Uživatelský manuál

Prerekvizity

Pro běh programů Terra Artifex, DDSViewer a HeightMapProcessor je třeba mít:

- Některou z verzí operačního systému Microsoft Windows Vista.
- Knihovny DirectX 10 (součást systému).
- Nainstalován DirectX End-User Runtimes November 2007 (instalační balík se nachází na přiloženém CD).
- Programy Terra Artifex a DDSViewer navíc vyžadují grafickou kartu kompatibilní s DirectX 10.

Programy PerlinNoiseGenerator a TableGenerator vyžadují pouze operační systém Microsoft Windows XP.

Terra Artifex

Terra Artifex je hlavní aplikací demonstrující výsledky našeho postupu pro zobrazování terénu. Spouští se souborem „Binary\Terra Artifex.exe“. V příkazové řádce je možné použít tyto argumenty:

-adapter:#	Umožňuje použít konkrétní grafickou kartu v případě jejich násobného počtu.
-windowed	Spustí aplikaci v okně (defaultní chování). Neslučuje se s argumentem *fullscreen*.
-fullscreen	Spustí aplikaci v celoobrazovkovém režimu. Neslučuje se s argumentem *windowed*.
-forcehal	Pro zobrazování bude použit Hardware Abstraction Layer. Neslučuje se s argumentem *forceref*.
-forceref	Pro zobrazování bude použit Reference Rasterizer. Neslučuje se s argumentem *forcehal*.
-forcevsync:#	Pokud je parametr 0, potlačí vertikální synchronizaci s kreslícím paprskem. Pro vyšší hodnoty nastaví periodu synchronizace.
-width:#	Nastaví šířku okna programu. Pokud je program spouštěn v celoobrazovkovém režimu, považuje se tato hodnota pouze za doporučení a bude nalezena nejbližší vhodná.
-height:#	Nastaví výšku okna programu. Pokud je program spouštěn v celoobrazovkovém režimu, považuje se tato hodnota pouze za doporučení a bude nalezena nejbližší vhodná.
-startx:#	Nastaví x-ovou souřadnici okna programu na ploše, pokud je zobrazován v okně.
-starty:#	Nastaví y-ovou souřadnici okna programu na ploše, pokud je zobrazován v okně.
-constantframetime:#	Aplikace poběží s konstantní zobrazovací frekvencí. Zadané číslo udává čas na jeden snímek v sekundách.

-quitafterframe:#	Aplikace bude automaticky ukončena po zobrazení zadaného počtu snímků.

Další nastavení programu se provádí editací konfiguračního souboru config.ini. V něm je možné nastavit tyto parametry:

RunTimeLimit	Umožňuje nastavit čas trvání běhu aplikace. Po uplynutí zadaného počtu vteřin se program sám ukončí. Umožňuje přesné měření pro účely testování. Pokud je zadaná hodnota záporná, je nastavení ignorováno.
TimeCounterStart	Pokud je *RunTimeLimit* kladný, nastavuje tento parametr čas běhu programu, od kterého se započne měření výkonu a časového limitu.
CameraMovStep	Ovlivňuje rychlost pohybu kamery. Hodnota udává, o kolik metrů se kamera posune při stisku klávesy.
CameraRotStep	Rychlost otáčení kamery. Hodnota udává, o kolik stupňů se kamera otočí při stisku klávesy.
CameraMouseStep	Udává citlivost myši při otáčení kamery.
FreeMovement	Pokud je nastaveno na 0, pozice kamery je zafixována do určité výšky nad terénem a lze ovlivňovat pouze pohyb ve dvou osách. Vhodné na „procházky“ krajinou.
FixedAltitude	Pokud je parametr *FreeMovement* nastaven na 0, tato hodnota udává výšku nad terénem, do jaké je kamera posazována.
DebugLevel	Nastaví podrobnost textového výstupu do souboru. 0 označuje žádný výstup a 4 maximálně podrobný výstup.
ScreenWidth	Umožňuje nastavit grafické rozlišení, v jakém se program spustí. Tento parametr může být předefinován z příkazové řádky použitím argumentu *width*.
ScreenHeight	Umožňuje nastavit grafické rozlišení, v jakém se program spustí. Tento parametr může být předefinován z příkazové řádky použitím argumentu *height*.
ShowHelp	Povolí nebo zakáže zobrazování nápovědy. Možné hodnoty 0 a 1.
ShowStats	Povolí nebo zakáže zobrazování informací o zobrazování. Možné hodnoty 0 a 1.
ShowGUI	Povolí nebo zakáže zobrazování ovládacích prvků. Možné hodnoty 0 a 1.
WireFrameMode	Povolí nebo zakáže zobrazování pomocí drátového modelu. Možné hodnoty 0 a 1.

MaxVisibleDistance	Nastaví maximální dohled v metrech. Tohle číslo vydělené počtem vzorků mapy reziduí na metr nesmí přesáhnout 4 000 a nesmí být menší než velikost mapy reziduí.
ViewAngle	Zorný úhel v radiánech.
VertexCacheSize	Umožňuje optimalizovat přístup do některých bufferů pomocí nastavení velikosti vertex cache.
HeightMapFileName	Jméno výškové mapy pro zobrazování. Používá se jméno souboru bez přípony.
CircMapRingsCount	Udává počet prstenců ringmapy.
CircMapSlicesCount	Počet výsečí ringmapy.
CircMapBaseSize	Velikost strany nejmenšího trojúhelníku ringmapy v metrech.
TrapezMapLinesCount	Počet řad vrcholů trapezmapy.
CircMapExpansionStep	Nastaví rychlost zvětšování hustoty sítě centrální části ringmapy. Možné jsou celočíselné hodnoty v rozsahu 0–100, kde 0 je nejrychlejší růst a 100 nejpomalejší.
SDPThreadsCount	Počet Worker vláken, která načítací systém vytvoří. Nastvením 0 je ponecháno rozhodnutí na aplikaci.
SDPMaxMemory	Povolené množství paměti v megabytech na uchování dat terénu. Větší hodnoty snižují četnost načítání dat.

V aplikaci je možné nastavit rychlost pohybu kamery a fixní výšku nad terénem pomocí grafických ovládacích prvků. Kamerou je možné otáčet pohybem myši při stištěném levém tlačítku. Zbylé ovládání probíhá pomocí kláves:

Šipky, Insert, Delete, Home, End, Page Up, Page Down	Pohyb a rotace kamery.
F1	Vypnutí či zapnutí zobrazování nápovědy.
F2	Vypnutí či zapnutí zobrazování statistiky a dalších informací.
F3	Vypnutí či zapnutí zobrazování ovládacích prvků.
W	Vypnutí či zapnutí zobrazování pomocí drátěného modelu.
B	Vypnutí či zapnutí zobrazovacího módi „dalekohled“.

PerlinNoiseGenerator

Program slouží ke generování výškové mapy pomocí Perlinova šumu. Výstupem programu je soubor RAW formátu zobrazitelný běžnými programy na úpravu obrázků, nebo formátu výškové

mapy, který používá pro vstup program HeightMapProcessor. Program se spouští souborem „Binary\PerlinNoiseGenerator.exe“ s následujícími argumenty:

-h	Namísto generování výškové mapy pouze zobrazí nápovědu pro použití programu.
-F name	Definuje jméno výstupního souboru. Defaultně pnoise2D.raw.
-s #	Definuje rozměr generované mapy. Hodnotou musí být celá mocnina dvou. Defaultně 1 024.
-o #	Počet oktáv, ze kterých se výsledný šum skládá. Defaultní hodnota je 8.
-p #	Tzv. persistence. Je to převrácená hodnota multiplikátoru amplitudy mezi jednotlivými oktávami. Defaultní hodnota je 2,0.
-f #	Základní frekvence použita pro první oktávu. Každá další oktáva používá dvakrát větší frekvenci. Defaultně 0,01.
-t #	Definuje formát výstupu. Možné jsou tři hodnoty: • 0 – výstupní formát je shodný s formátem používaným pro vstup programu HeightMapProcessor. • 1 – RAW data, kde každý vzorek je reprezentovaný jední bytem. • 2 – RAW data, kde každý vzorek je reprezentovaný dvěma byty.

HeightMapProcessor

Program je určený k vytvoření dat potřebných při zobrazování terénu aplikací Terra Artifex na základě vstupní výškové mapy (zpravidla vytvořené programem PerlinNoiseGenerator). Program vytváří 4 soubory stejného jména lišící se v příponě. Soubor s příponou *.lowRes* obsahuje definici hrubé výškové mapy. Příponu *.residue* dostane soubor obsahující mapu reziduí a normálová mapa je uložena do souboru s příponou *.normal*. Posledním vytvářeným souborem je soubor diferencí s příponou *.diff*, který se používá pouze pro zkoumání výsledků komprese a rekonstrukce dat. Program se spouští souborem „Binary\HeightMapProcessor.exe“ s následujícími argumenty:

-h	Namísto generování dat se pouze zobrazí nápověda pro použití programu.
-i name	Definuje jméno vstupního souboru. Defaultně test.heightmap.
-o name	Definuje jméno výstupního souboru bez přípony. Přípona je přidána každému z generovaných souborů automaticky podle typu dat. Defaultní jméno je test.
-r	Povolí generování mapy reziduí. Defaultně je vypnuto.
-l	Povolí generování hrubé výškové mapy. Defaultně je vypnuto.
-n	Povolí generování normálové mapy. Defaultně je vypnuto.
-a	Povolí generování mapy reziduí, výškové mapy a normálové mapy.

	Tento argument má stejný význam jako současné použití argumentů '–r –l –n'.
-d	Povolí generování souboru diferencí. Defaultně je vypnuto.
-tw #	Definuje šířku (v texelech) jednoho čtvercového bloku, do kterých je terén rozdělen. Defaultní velikost je 128.
-th #	Definuje výšku (v texelech) jednoho čtvercového bloku, do kterých je terén rozdělen. Defaultní velikost je 128.
-lr #	Poměr rozlišení originální výškové mapy a generované hrubé výškové mapy. Defaultní hodnota je 32.
-ma #	Maximální převýšení terénu v metrech. Defaultní hodnota je 1 000,0.
-mr #	Rozsah reziduí v metrech. Defaultní hodnota je 30,0.
-sw #	Rozměr jednoho vzorku mapy reziduí v metrech. Defaultně 1,0.