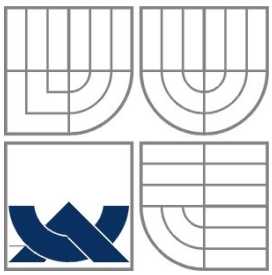
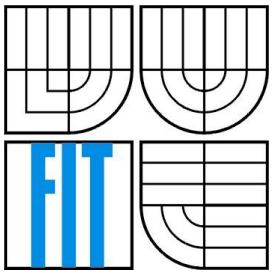


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

RENDEROVÁNÍ ROZSÁHLÉHO TERÉNU

LARGE TERRAIN RENDERING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Boris Bondarenko
AUTHOR

VEDOUCÍ PRÁCE — Ing. Rudolf Kajan
SUPERVISOR

BRNO 2011

Abstrakt

Realizace renderování rozsáhlých terénů je v moderních aplikacích, které zahrnují tuto tematiku známým problémem. S renderováním rozsáhlých terénů se můžeme střetnout ve velké škále aplikací, od počítačových her, až po profesionální nástroje používané pro tvorbu prostředí a vizuálních efektů. Právě pro implementaci takových aplikací existuje mnoho různých algoritmů, které používají různé techniky pro uchovávání, zpracování a zobrazování dat. V oblasti renderování grafických dat je důležité limitování výkonem systému, ale stejně i limitování rozlišovací schopností lidského oka. Další důležitou součástí je výběr API, pomocí kterého bude program muset spolupracovat s grafickou kartou a pamětí. V téhle práci je použité rozhraní Direct3D z balíku DirectX9. Pro porovnání přístupů v realizaci renderování rozsáhlých terénů budou porovnávané přístupy renderování pomocí tzv. Brutreforce, quadtree a ROAM algoritmů.

Abstract

Realization of large terrain rendering is well-known problem in modern applications which covers this topic. Large terrain rendering can be found in large scale of programs from video games to professional tools used for environment rendering and visual effects processing. For implementation of such programs exists much algorithms, using different techniques for storing, processing and rendering of data. Important subject in domain of graphics data rendering are system limits and also distinctive capacity of human eye. Next important part is the pick of API through which program will communicate with graphics card. In this thesis Direct3D interface from DirectX9 package have been chosen. For comparison in approach to large terrain rendering realization, following algorithms are going to be used : brute force, quadtree and ROAM algorithm.

Klíčová slova

Renderování rozsáhlých terénů, počítačové hry, ROAM, quadtree, brute force, Direct3D, DirectX9

Keywords

Large terrain rendering, video games, ROAM, quadtree, brute force, Direct3D, DirectX9

Citace

Boris Bondarenko: Renderování rozsáhlého terénu, bakalářská práce, Brno, FIT VUT v Brně, 2011

Renderování rozsáhlého terénu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Rudolfa Kajana. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Boris Bondarenko
17.5.2011

Poděkování

Chcel by som týmto poďakovať vedúcemu mojej bakalárskej práce Ing. Rudolfovi Kajanovi, za jeho rady a podporu pri vypracovávaní bakalárskej práce a taktiež svojej rodine a priateľom, ktorí mi vždy boli oporou a stáli za mnou.

Obsah

1 Úvod

Oblasť zobrazovania dát v počítačovej grafike je natoľko členitá, že si vyslúžila obrovskú pozornosť ľudí, o čom svedčí množstvo algoritmov a prístupov v implementácií, ktoré boli za ten čas vytvorené. V modernej dobre prevláda názor, že treba čo najviac odbremeniť procesor od všetkých možných zaťažujúcich operácií a previesť túto záťaž na grafickú jednotku, ktorá sa postará o spracovanie grafických dát. Avšak tento prístup sa zaužíval až s príchodom nových grafických kariet, ktoré majú dostatočnú výpočetnú silu.

Renderovanie rozsiahlych terénov je vlastné viacerým aplikáciám používaných v širokom spektre, ako sú napríklad vojenské a letecké simulátory, software pre vytváranie scén, animácií a špeciálnych efektov vo filmovom priemysle, alebo video hry.

Jedným z hlavných problémov pri zobrazovaní terénu, je zobraziť iba tie dáta, ktoré sú relevantné pre danú scénu a to tak, aby sa odbremenila grafická jednotka od zbytočných dát.

Problematika renderovania rozsiahleho terénu je obsiahla oblasť, ktorej sa ľudia venujú už desaťročia a obsiahnuť celý problém v tejto práci by bolo nemožné. Preto sa zaoberá iba vybranými algoritmami pre renderovanie terénu, ktoré budú taktiež implementované. Všetky algoritmy budú v nasledujúcich kapitolách rozobrané a vysvetlené.

Ciele tejto práce sú nasledovné :

- popísať jednotlivé vybrané algoritmy
- implementovať tieto algoritmy
- vyhodnotiť rozdiely v prístupoch pri zobrazovaní rozsiahleho terénu.

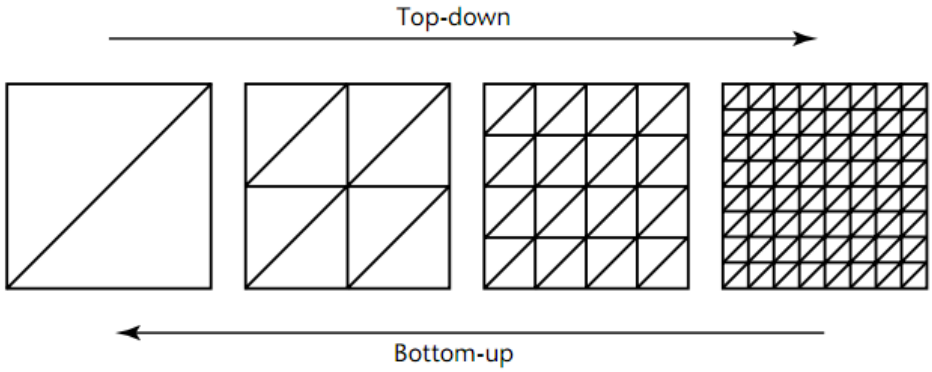
Po prečítaní tejto bakalárskej práce by mal čitateľ pochopiť princípy renderovania rozsiahlych terénov a obsiahnutých algoritmov. V nasledujúcej kapitole sa čitateľ dozvie o prístupoch používaných pri práci s terénom v grafike a mala by zároveň uviesť čitateľa do základov ďalej rozobranej problematiky. Obsah tretej kapitoly nesie dôležité informácie o algoritmoch a metódach použitých v implementácií bakalárskej práce, ktorých implementácia je vysvetlená vo štvrtej kapitole. V piatej a šiestej kapitole je porovnanie týchto algoritmov na základe rôznych kritérií a ich testovanie. V poslednej kapitole sú zhrnuté dosiahnuté výsledky a návrh pre pokračovanie v práci. Na záver je priložený zoznam použitej literatúry a príloh.

2 Prístupy v renderovaní rozsiahlych terénov

Existuje mnoho algoritmov a prístupov k renderovaniu terénov, ktoré sa počas rokov vyvíjali, a v dnešnej dobre máme k dispozícií širokú škálu možných riešení tohto špecifického problému. V tejto práci sú použité hierarchické stromy pre reprezentáciu a segmentáciu scény, a teda nás budú zaujímať prístupy *top-down* a *bottom-up*.

Top-down prístup začína pracovať od začiatku koreňa hierarchického stromu a pokračuje smerom k listovým uzlom. Začína teda od najmenej detailnej úrovne a postupne pridáva detaily priechodom stromom smerom k listovým uzlom. Tieto algoritmy si vyžadujú hneď na začiatku celý model, a preto majú vyššie pamäťové nároky [1].

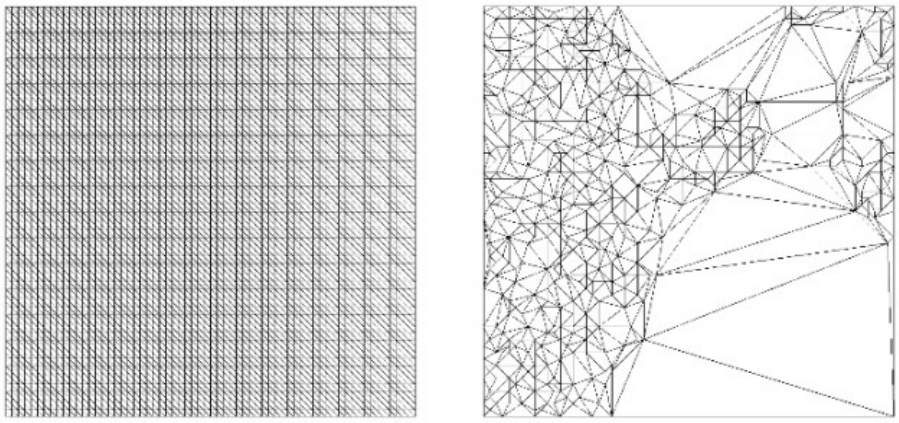
Bottom-up prístup začína spracovávať scénu od listov stromu (posledné uzly bez nasledovníkov), čiže od najväčších detailov, a postupne iteratívne odstraňuje vrcholy až do požadovaného stupňa zjednodušenia. Tento prístup sa taktiež nazýva *decimácia* [1].



Obrázok 2.1: top-down a bottom-up prístup [1]

Ďalším rozdielom v prístupe zobrazovania scény je použitie trojuholníkovej siete. V našom prípade sme použili *pravidelne rozložené výškové pole (regular gridded heightfield)*. Charakteristikou tejto trojuholníkovej siete je pravidelné rozloženie vrcholov trojuholníkov medzi sebou. Pravidelne rozložené výškové pole používa pre svoju reprezentáciu údaje o výške v dvojrozmernom poli.

Iný prístup má názov *TIN* (*triangulated irregular network*), ktorý predstavuje opak v rovnomernosti ukladania bodov, avšak uloží menej vrcholov pre požadované zobrazenie [1]. Pravidelne rozložené výškové polia sú vo výsledku menej optimalizované, keďže dovoľujú rovnaké rozlíšenie pre ploché oblasti ako aj pre tie viac členité.



Obrázok 2.2: rovnomerné výškové pole a TIN [2]

2.1 Level of detail

Level of detail (úroveň detailu) zahŕňa zmenu zložitosti objektu vzhľadom na meniacu sa vzdialenosť k pozorovateľovi, podľa istých stanovených podmienok, ako je napríklad dôležitosť objektu v scéne. Pomocou úrovne detailu vieme optimalizovať záťaž a zefektívniť využívanie zdrojov [1]. Podľa základných prístupov môžeme zadeliť jednotlivé algoritmy do skupín úrovní detailu podľa rôznych kritérií.

2.1.1 Discrete level of detail

Táto podkapitola bola prevzatá z [1].

Tento prístup sa označuje za tradičný. Prístup spočíva v tom, že sa z každého objektu vytvorí sada úrovní jeho detailu a v programe sa použije ten, ktorý je práve vhodný. Pretože sa výpočty úrovní detailu prevádzajú počas predspracovávania a nie za behu programu, zjednodušovací proces nedokáže predpovedať z akého smeru bude objekt viditeľný. Zjednodušovanie preto zredukuje detail rovnako na celom objekte, a preto sa taktiež odkazuje na diskrétnu úroveň detailu ako pohľadovo nezávislú. Diskrétna úroveň detailu má viacero výhod. Zjednodušenie odstraňovaním detailu a vykresľovanie robí diskrétnu úroveň detailu najjednoduchšou na vytvorenie. Zjednodušovací algoritmus môže spracovávať dáta tak dlho, ako potrebuje na vytvorenie úrovní detailu a počas behu programu sa iba zvolí daná úroveň pre každý objekt.

2.1.2 Continuous level of detail (CLOD)

Na rozdiel od *discrete Level of detail* nepredpočítava každú úroveň objektu pred behom programu, ale vypočítava ju na základe daného algoritmu počas jeho behu [1]. Výhodou týchto algoritmov je lepšie rozmiestnenie detailu do oblastí, kde je to potrebné. Taktiež je schopný upravovať počet

vygenerovaných trojuholníkov na základe kritérií. Vďaka tomu je možné vygenerovať požadované množstvo trojuholníkov a zbytočne tak nezaťažovať pamäť [1]. V tejto práci sú použité dva algoritmy pracujúce na základe CLOD.

2.1.3 View-dependent level of detail

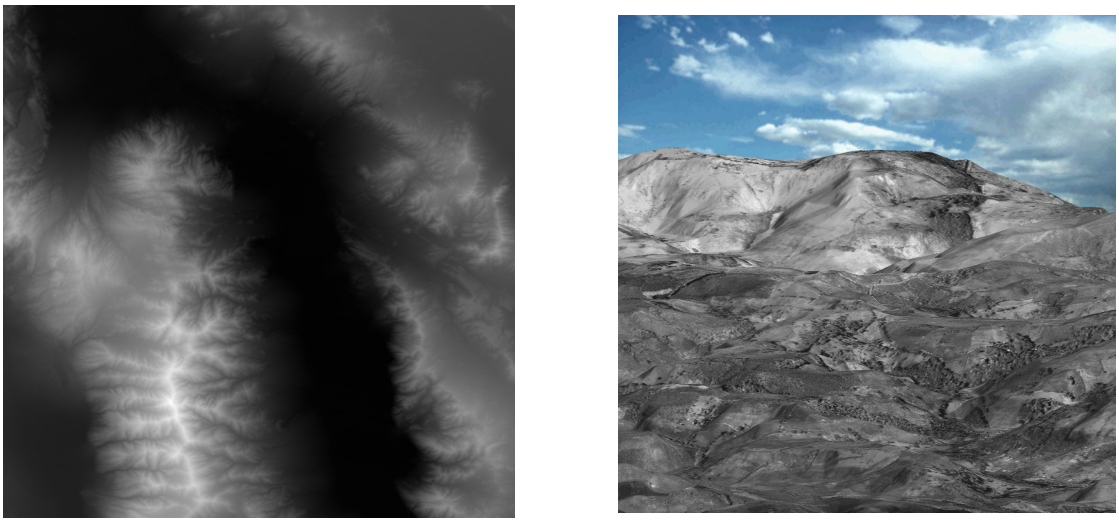
Rozširuje praktiky CLOD o to, že zahŕňa zjednodušovacie kritériá založené na viditeľnosti, aby mohol dynamicky určiť vhodnú úroveň detailu. Vskutku komplexné modely reprezentujúce fyzicky veľké objekty, ako napríklad terén, často nemôžu byť adekvátne zjednodušené bez techník závisejúcich od viditeľnosti [1].

2.2 Reprezentácia dát terénu

Najmenšou jednotkou, ktorá reprezentuje terén je v tomto prípade trojuholník. V konečnom dôsledku je to správny výber vrcholov trojuholníkov, ktorý môže zaručiť dobrú výslednú scénu. Pre reprezentáciu dát existujú rôzne formy uloženia. *Výškové mapy* sú veľmi častým spôsobom používaným pre reprezentáciu rozdielov vo výške terénu. Výšková mapa je dvojrozmerná funkcia, ktorá je v našom prípade reprezentovaná rastrovým obrázkom v odtieni šedej.

Reprezentácia farby v odtieňoch šedej určuje rozdiel vo výške. Miesta v mape, ktoré sú svetlejšie (v RGB sú bližšie k hodnote 255), sú vyššie ako tmavšie miesta (v RGB bližšie k hodnote 0). Tento spôsob reprezentovania výškovej mapy sa nazýva pravidelne rozmiestnená výšková mapa. Jedným z najväčších problémov výškových máp je fakt, že nedokážu reprezentovať prírodné útvary ako skalné previsy či jaskyne, a preto sa im taktiež hovorí že sú 2,5 rozmerné.

Iným spôsobom pre reprezentáciu dát sú *DEM* (*Digital Elevation Model*) súbory. *Digital evaluation model* je digitálny súbor, ktorý obsahuje výšky terénu pre pozície na zemi v pravidelne rozložených intervaloch [8]. Tieto súbory sú taktiež rastrovým formátom, rovnako ako výškové mapy. Tento formát je veľmi populárny v oblastiach archeológie či geológie.



Obrázok 2.3: Výšková mapa a renderovaný DEM model[9]

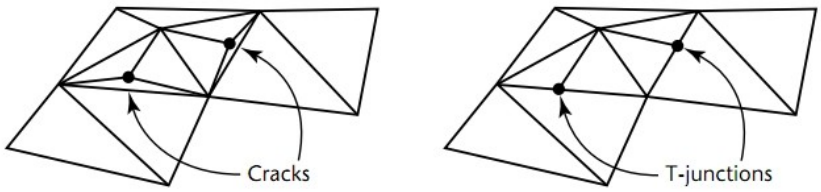
2.3 Problémy pri zobrazovaní rozsiahlych terénov a ich riešenia

Pri implementácií algoritmov, ktoré majú za úlohu zobrazovanie rozsiahlych terénov, sa často stretávame s viacerými problémami špecifickými pre zjednodušovacie algoritmy.

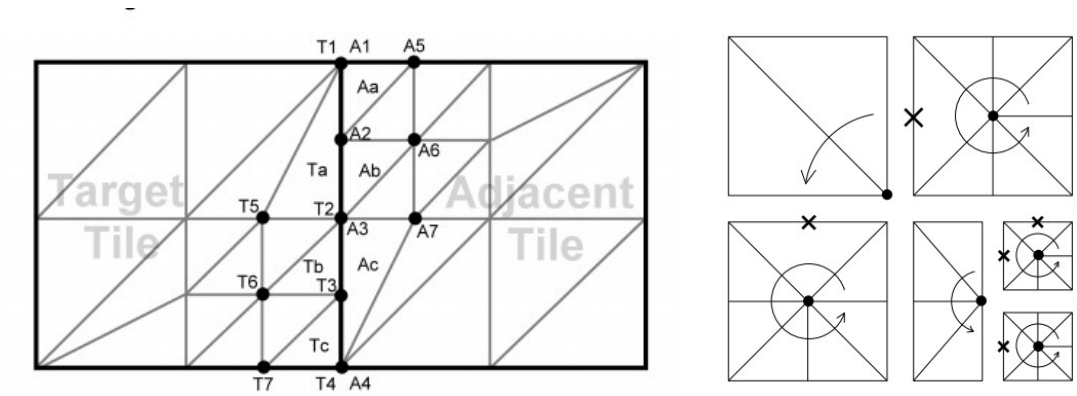
2.3.1 Trhliny a t-spoje

Tieto problémy nastávajú hlavne pokiaľ dva vzájomne susedné uzly majú inú úroveň detailu. V tomto prípade môžu nastať viditeľné *trhliny* v teréne, pretože jeden z uzlov má na zdieľanej strane o vrchol navyše [1]. Ak má zmienený vrchol inú výšku ako zdieľaná hrana, je tak viditeľné pozadie za terénom a výsledná scéna je nespojitá. Ďalší nežiaduci jav je *t-spoj*, ktorý vzniká pokiaľ uzol s vyššou úrovňou detailu nezdieľa vrchol s uzlom z nižšej úrovne detailu, čo môže taktiež viesť k *trhlinám* [1]. *Trhliny* a *t-spoje* sú častým problémom hlavne pri blokových štruktúrach ako kvadrantové stromy [1].

Riešenie býva často špecifické pre daný algoritmus, a preto tu budú uvedené príkladné riešenia niektorých algoritmov. Algoritmy používajúce blokové štruktúry riešia problém buď odoberaním vrcholov, kde nastáva t-spoj ako v [3], alebo iteratívnym prechodom a prepájaním súvisiacich vrcholov na zdieľanej hrane ako v [11]. Pri použití binárnych trojuholníkových stromov je riešenie jednoduchšie z hľadiska manuálneho vykonávania zmien v štruktúre vrcholov. V tomto prípade dochádza k rekurzívnemu rozdeľovaniu súvisiacich uzlov a v konečnom dôsledku pridávaniu detailu ako v [4] a [5] .



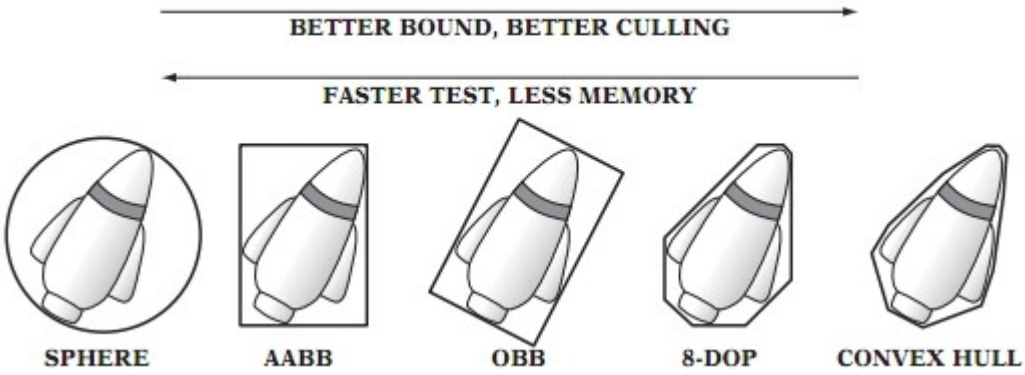
Obrázok 2.4: Trhliny a t-spoje [1]



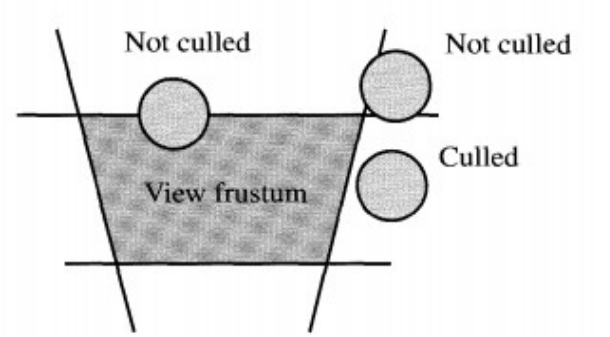
Obrázok 2.5: odstranovanie t-spojov [11] a [3]

2.3.2 Orezávanie neviditeľných častí scény

Ďalším z problémov je vykresľovanie neviditeľných častí, alebo častí mimo scénu. Vykresľovanie týchto častí môže viesť k zníženiu výkonu aplikácie. Metóda odstraňovania častí, ktoré sú neviditeľné z dôvodu zatienenia iným objektom, sa nazýva *occlusion culling*. Metóda pre odstraňovanie častí mimo zorného poľa kamery sa nazýva *view-frustum culling* (oreaznie pohľadovým ihlanom). Pri reprezentácií scény hierarchickými stromovými štruktúrami je jednoduché vykonať orezanie scény podľa pohľadového ihlanu kamery, najmä pri použití *top-down* algoritmov. Princíp orezávania spočíva vo vytvorení obaľovacej štruktúry uzla stromu a následné vyhodnotenie vzhľadom k stenám pohľadového ihlanu. Každá stena ihlanu je matematicky popísaná ako plocha, a tak po výpočte jej normály je jednoduché zistiť pozíciu bodu vzhľadom k ploche. Medzi najčastejšie používané obálky patria *axis aligned bounding box (AABB)*, *bounding sphere* (*obaľovacia guľa*) a *oriented bounding box (OBB)*. *Bounding sphere* sa často používa kvôli rýchlosti vyhodnotenia príslušnosti objektu v rámci scény, ktorý obaľuje. Pri zložitejších objektoch však nemusí objekt dostatočne tesne obaliť, a preto sa používa ako prvý test príslušnosti a ak testom prejde, tak sa pokračuje s testovaním na zložitejšej obálke.



Obrázok 2.6: obaľovacie štruktúry [7]



Obrázok 2.7: orezanie pohľadovým ihlanomp [12]

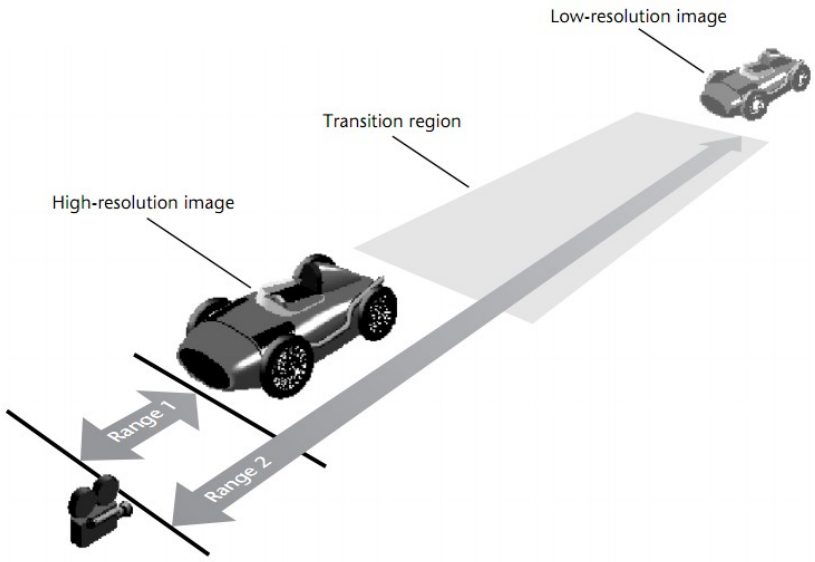
2.3.3 Popping

Popping je jav, ktorý nastáva pri prechode medzi dvoma úrovňami detailu a je viditeľný ako náhla zmena geometrie v scéne. *Geomorphing*, alebo *blending* sú metódy pomocou ktorých je možné odstrániť toto nežiadúce chovanie.

2.3.4 Blending

Blending je metóda používaná pri zjemnení náhlych prechodov v úrovniach detailu miešaním jednej úrovne detailu do druhej. Pre každú úroveň detailu systém nastaví vzdialenosť od ktorej sa bude meniť úroveň *alfa*. *Alfa* je nastaviteľná hodnota od 0.0 do 1.0, kde 1.0 znamená plnú viditeľnosť a 0.0 znamená neviditeľnosť objektu. Pri približovaní sa a odďaľovaní sa od objektu sa mení úroveň *alfa* každej úrovne detailu. Teda niekedy je viditeľnejšia jedna úroveň ako druhá a naopak.

Hoci výhodou tejto metódy sú jemnejšie prechody medzi úrovňami detailu, nevýhoda je v udržovaní väčšieho počtu trojuholníkov. Tým môže systém vyvolať zmenu úrovne detailu, a preto je vhodné udržovať oblasť pri ktorej dochádza ku zmene viditeľnosti čo najmenšiu, aby došlo ku zamedzeniu nežiadúcemu javu [1].



Obrázok 2.8: znázornenie fukcie alpha blendingu na modely auta [1]

2.4 Pridávanie detailu do scény

Po vygenerovaní scény je vhodné scénu priblížiť realite pomocou pridávania efektov. V poslednej dobe sa stretávame s čoraz zväčšujúcimi sa požiadavkami na výkon systému, ktoré sú v nemalej miere podmienené náročnými operáciami, ktoré vytvárajú dojem reality, svetla, odleskov a tieňov. Medzi základy pridaného detailu patrí textúrovanie. Textúry môžeme medzi sebou miešať, a tým pridávať ešte viac detailu do scény. Tak isto odraz svetla a tieňovanie má zásadný vplyv na farbu textúry. Existujú rôzne spôsoby druhov a použitia textúr, no v tomto projekte bolo použité procedurálne textúrovanie, ktorým sa budeme zaoberať.

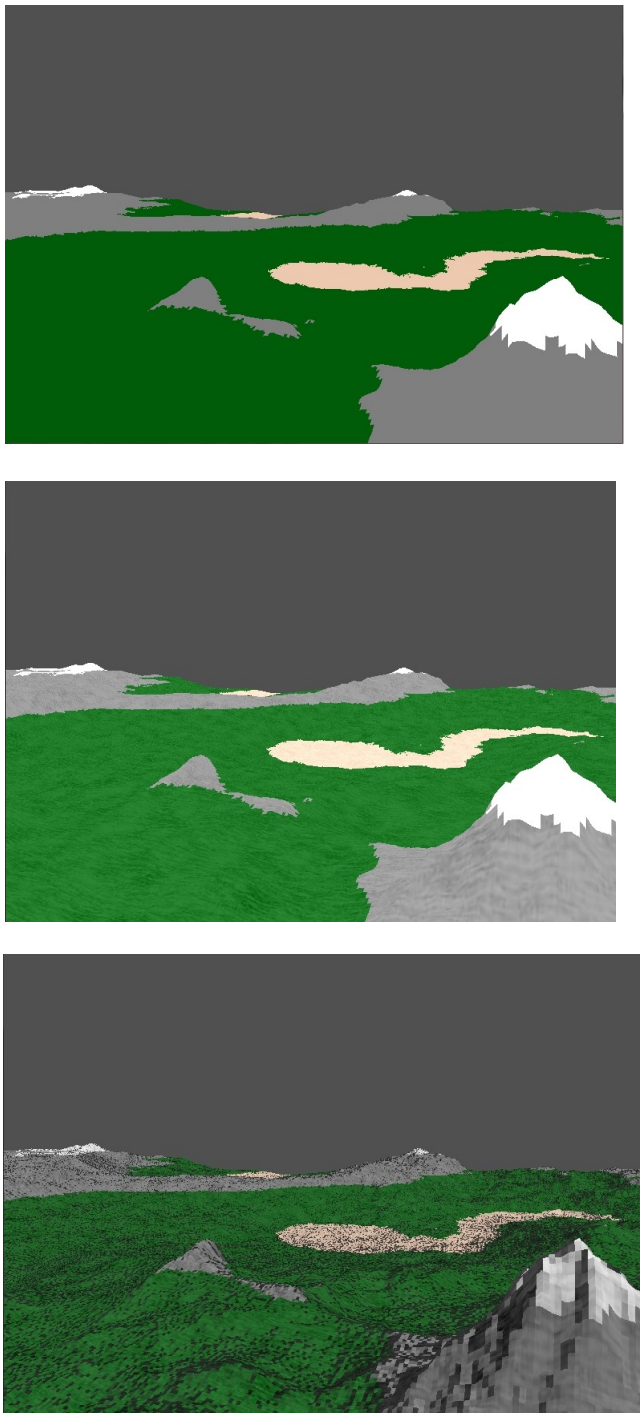
Procedurálne techniky sú časti kódu, alebo algoritmov, ktoré upresňujú niektoré črty počítačovo generovaného modelu alebo efektu. Napríklad procedurálna textúra pre mramorovú plochu nepoužíva oskenovaný obraz aby mohla definovať hodnoty farieb, no namiesto toho používa algoritmy a matematické funkcie pre zistenie farby [10].

Takýmto spôsobom môžeme pridať terénu farbu na požadované miesta a vytvoriť tak aproximovaný povrch. Môžeme pridať ešte viac detailu do scény aplikovaním *detail-mapy* na textúru. Tým docielime pridanie detailu do textúry a tak skvalitnenie výsledného obrazu.

Detail môže znamenať stmavenie, alebo zosvetlenie častí textúry. Zmiešaním farieb textúry a *detail-mapy* získame výsledný povrch, ktorý vyzerá lepšie ako samotná textúra.

Jeden z najväčších efektov, ktorý dokáže dodať scéne tretí rozmer, je svetlo. Pomocou nasvietenia scény sme schopní simulovať dopad a odraz lúčov, a tak zosvetliť alebo stmaviť farbu povrchu. K osvetleniu povrchu je možné pristupovať taktiež rôznymi spôsobmi.

Buď môžeme použiť rozhranie na výpočet osvetľovacieho modelu (v tomto prípade DirectX), alebo môžeme použiť spôsob nazvaný *light-mapping*. Tento spôsob spočíva v aplikovaní textúry, ktorá už bola osvetlená.



Obrázok 2.9: postupné pridávanie detail-mapy a dark mapy na textúru

3 Algoritmy pre renderovanie rozsiahleho terénu

V tejto práci sú použité nasledujúce algoritmy :

- Brute force (hrubá sila)
- Quadtree algoritmus [3]
- Real Time Opimally Adapting Meshes [4]

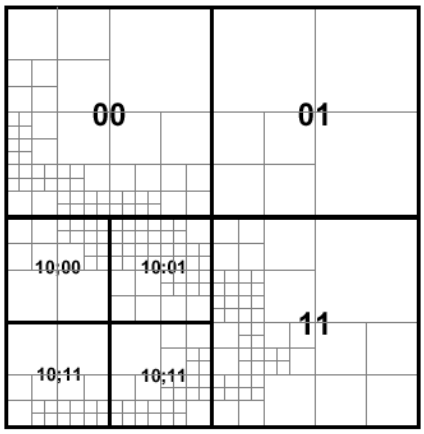
Každý z týchto algoritmov má nepochybne svoje výhody a nevýhody. Líšia sa hlavne v prístupe k organizovaniu a optimalizácií výsledných dát, čo bude hlavným zdrojom pri porovnávaní týchto algoritmov.

3.1 Algoritmus Brute force (hrubá sila)

V modernej dobe síce zažívame nárast výkonu grafických a centrálnych procesorových jednotiek, ale stále sa považuje použitie algoritmov, ktoré redukujú objem spracovaných dát za dôležité v oblasti počítačovej grafiky. Princíp tohoto algoritmu spočíva v tom, že bez ohľadu na akékoľvek okolnosti vykreslí všetky informácie o scéne. Neobsahuje žiadnu optimalizáciu výstupných dát a preto ho v náročnejších aplikáciách nemožno používať. Pri veľkom počte objektov v scéne a výpočtov spojených so stále sa rozširujúcimi vizuálnymi efektmi, toto mrhanie voľnými prostriedkami môže viesť k nežiadúcemu zníženiu počtu snímkov. Posielať dáta o celom teréne v prípade, keď je potrebné vykresliť iba jeho viditeľnú časť, je pre systém veľmi náročné a hlavne zbytočné. Taktiež sa väčšina aplikácií nezaoberá iba vykreslením terénu, ale aj objektov a riadeniu ich vzájomných interakcií, čo by pri odosielaní veľkého množstva dát malo negatívny vplyv na výkon aplikácie. Tento algoritmus je použitý hlavne ako kontrast voči ostatným implementovaným algoritmom, a preto nebude porovnávaný ani z hľadiska implementácie, ani z hľadiska výkonu.

3.2 Quadtree algoritmus

Tento algoritmus bol predstavený Stefanom Röttgerom et al. a je súčasťou rodiny CLOD algoritmov. Pre reprezentáciu dát scény používa *kvadrantový strom* (*quadtree*). *Kvadrantový strom* je dvojrozmerná hierarchická dátová stromová štruktúra, ktorá slúži k uchovávaniu informácií o scéne. Rekurzívne rozkladá mapu na menšie časti, pričom najvrchnejší uzol reprezentuje celú mapu. Každý uzol má štyri dcérske uzly, z ktorých každá reprezentuje jednu štvrtinu scény, ktorá im bola predaná jeho rodičovským uzlom. Tieto uzly sa ďalej rekurzívne rozvetvujú až k listovým uzlom, ktoré nemajú žiadnych potomkov. Táto štruktúra je vhodná na reprezentáciu scény, ako aj polohy objektov v scéne, kde každý objekt je zaradený do tých uzlov, ktoré pretína. Použitie kvadrantového stromu umožňuje efektívne orezávanie neviditeľných častí scény.



Obrázok 3.1: štruktúra quadtree pri delení priestoru

Algoritmus prezentovaný Stefanom Röttgerom et al. používa kvadrantový strom, ktorý je reprezentovaný booleovou maticou, ktorá je rovnakých rozmerov ako výškové pole. V tomto prípade predpokladáme, že výškové pole je o veľkosti $2^n + 1 * 2^n + 1$. Matica obsahuje údaj o rozvetvení každého uzla v strome. Uzly, ktoré sú ďalej rozvetvené sú reprezentované v matici hodnotou 1 a tie, ktoré nie sú ďalej rozvetvené, ako listové uzly a uzly mimo vykresľovanú scénu, sú označené hodnotou 0.

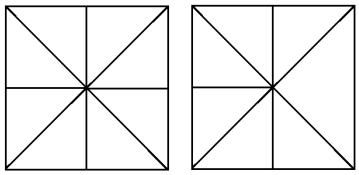
$$\begin{pmatrix} ? & ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & 0 & ? & 0 & ? \\ ? & ? & 0 & ? & ? & ? & 1 & ? & ? \\ ? & ? & ? & ? & ? & 0 & ? & 0 & ? \\ ? & ? & ? & ? & 1 & ? & ? & ? & ? \\ ? & 0 & ? & 0 & ? & 0 & ? & 1 & ? \\ ? & ? & 1 & ? & ? & ? & 1 & ? & ? \\ ? & 0 & ? & 0 & ? & 0 & ? & 1 & ? \\ ? & ? & ? & ? & ? & ? & ? & ? & ? \end{pmatrix}$$

Obrázok 3.2: matica kvadrantového stromu [3]

Hodnoty matice označené otáznikom nemusia byť nastavené počas výpočtu triangulácie, pretože tieto hodnoty nebudú dosiahnuté top-down algoritmom pre danú trianguláciu [3]. Každá nižšia úroveň stromu znamená detailnejšiu reprezentáciu tej časti scény, ktorú daný uzol reprezentuje.

3.2.1 Vykresľovanie výškového poľa

Vykreslenie trojuholníkovej siete výškového poľa je vykonané rekurzívnym prechádzaním stromu po tých uzloch, ktoré sú nastavené na hodnotu 1. Hocikedy, keď sa dosiahne listového uzlu, tak je vykreslený čiastočne, alebo plno [3] ako je znázornené na obrázku 3.3.



Obrázok 3.3: v ľavo plno a na pravo čiastočne vykreslený uzol

K vykresleniu čiastočného uzla dôjde ak s ním susediaci uzol je o úroveň v strome vyššie. To je zaistené vynechaním prostredného uzla na tej strane, kde sa nachádza susedný uzol s nižšou úrovňou detailu. Či je susedný uzol rovnakej úrovne, alebo nie, zistíme kontrolovaním jeho hodnoty v booleovej matici. Avšak na zaistenie správneho zobrazenia bez t-spojov je dôležité, aby sa susedné uzly nelíšili od seba viacej ako o jednu úroveň. Toto kritérium sa ukázalo byť najproblematickejším v riešení algoritmu.

3.2.2 Výpočet triangulácie

Keďže vzdialenosť je jedným z najdôležitejších faktorov ovplyvňujúcich výslednú scénu, je nutné zaistiť jej správny výpočet. V tomto prípade bola použitá rovnica L^1-Norm [2]. Táto rovnica je jednoduchšia na výpočet ako rovnica pre výpočet euklidovskej vzdialenosti medzi dvoma bodmi, avšak markantne nezhoršuje presnosť pri rozhodovaní vo výbere úrovne detailu. Rovnica pre výpočet vzdialenosti stredu uzla od kamery v trojrozmernom priestore:

$$l = |x_2 - x_1| + |y_2 - y_1| + |z_2 - z_1|$$

Rovnica 1: výpočet vzdialenosti

Keďže je zaistená správna vzdialenosť, je nutné ju aplikovať v rámci kritéria, ktoré zistí či je potrebné postúpiť ďalej k vyššej úrovni detailu, alebo nie. Kritérium pre rozdelenie daného uzla:

$$(l / d) < C$$

Rovnica 2: prvé kritérium pre rozdelenie uzla

Kde *l* je vzdialenosť od stredu aktuálneho uzla do kamery a *C* je konštanta kontrolujúca úroveň detailu. Práve konštanta *C* nám umožňuje kontrolovať globálnu úroveň detailu. Podľa autorov tohto algoritmu by mala byť táto hodnota nastavená na deväť [2]. Toto kritérium zaistí iba to, že so zväčšujúcou sa vzdialenosťou sa bude zväčšovať detail scény, avšak nezaistí to, že oblasti s väčšou členitosťou terénu budú detailnejšie vykreslené ako relatívne ploché časti terénu. To zaistíme ďalším kritériom pre rozdelenie uzla, ktoré berie do úvahy aj členitosť terénu.

$$f = (l / (d * C * max(c * d2, 1)))$$

Rovnica 3: konečné kritérium pre rozdelenie uzla

V tejto rovnici figurujú ďalšie členy, ktoré ovplyvnia výslednú scénu. Konštanta *c* reprezentuje požadovanú úroveň detailu a hodnota *d2* nesie informáciu o členitosti terénu. Upravovaním konštanty *c* sa ovplyvní počet vykresľovaných trojuholníkov v scéne. Uzol sa podľa tohto kritéria bude ďalej rozdeľovať ak hodnota *f* vyjde menšia ako jedna.

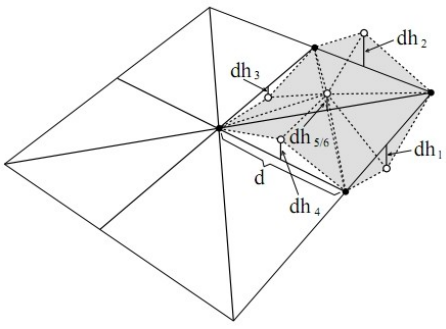
3.2.3 Členitosť terénu

Pri zostúpení jednej úrovne v hierarchickom strome vzniká nová chyba presne na piatich miestach: v strede uzla, a na stredoch jeho štyroch hrán [3]. Horná hranica pre odhadovanú chybu sa vypočíta ako maximum absolútnych hodnôt rozdielov vo výškach jednotlivých vrcholov. Tie vypočítame ako rozdiel výšok na hranách uzla a na jeho diagonálach. Teda pre výpočet hodnoty jedného bodu použijeme vzorec:

$$dh_i = |(((a+b)/2) - c)|$$

Rovnica 4: výpočet lokálnej chyby

Kde *a,b* sú hodnoty výšok bodov ležiacich na rovnakej hrane ako dh_i a *c* hodnota bodu ležiaceho na strede hrany.

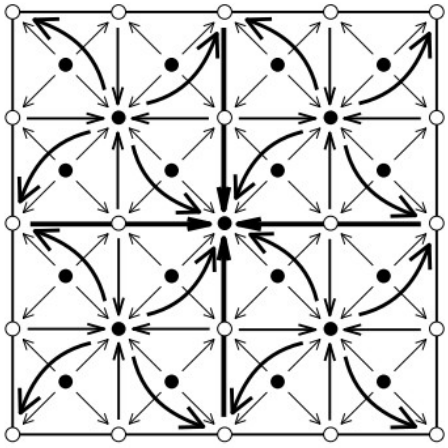


Obrázok 5.3: rozdiely v elevácii na jednotlivých vrcholoch uzla [3]

Aby sme zaistili, že nenastanú t-spoje a budeme schopní splniť podmienku, aby sa nelíšili dva vzájomne susediace uzly od seba o viacej ako jednu úroveň, je nutné spropagovať vypočítanú členitosť uzla ku všetkým jeho susedným uzlom, ktoré sú o jednu úroveň v hierarchickom strome vyššie a takisto k rodičovskému uzlu. Pričom v tomto prípade postupujeme od najnižšej úrovne k najvyššej v hierarchickom strome, a teda sa jedná o metódu *bottom-up.* Výsledná členitosť uzla ku ktorému sa propagujú hodnoty z uzlov s vyššou úrovňou detailu sa rovná maximu z jeho vlastnej členitosť a členitosť týchto uzlov vynásobených konštantou *K*.

$$K = C/2*(C-1)$$

Rovnica 5: konštanta, ktorá zabezpečí spojitosť



Obrázok 3.4: propagácia členitosti povrchu od najmenších uzlov k najväčším [3]

Správna propagácia členitosti zaručí scénu, ktorá neobsahuje žiadne t-spoje s úrovňou detailu pridanou v regiónoch, ktoré to potrebujú. Takéto regióny v reálnom svete môžeme považovať za kaňony, meandre, korytá riek alebo hrebene vrcholov, ktoré chceme, aby boli rozlíšiteľné aj z väčšej vzdialenosti. Keďže počítame v aritmetike s plávajúcou čiarkou, čo je výpočetne náročné, a výškové pole sa nemení, tak je vhodné predpočítať a uložiť hodnoty členitosti.

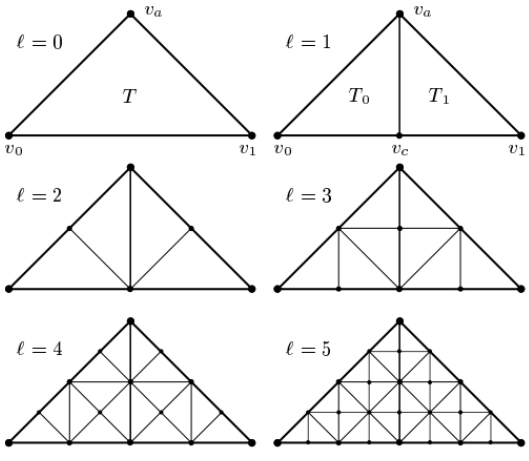
Geomorphing je možné do tohto algoritmu zakomponovať a zamedziť tak náhlym viditeľným prechodom medzi úrovňami detailu v scéne, avšak ten nebol v tejto práci implementovaný a preto sa ním ďalej zaoberať nebudeme.

3.3 Real-time optimally adapting meshes

Po dlhú dobu bol jedným z najpopulárnejších algoritmov pre renderovanie terénu. Jeho popularizáciu priniesol *Seamus McNally* s jeho implementáciou v hre TreadMarks. Na rozdiel od algoritmu prezentovaného Stefanom Röttgerom et al., ROAM pre reprezentáciu scény používa binárny trojuholníkový strom [2]. Pôvodný algoritmus vytváral trojuholníkovú sieť na základe aplikovania *split* a *merge* operácií na trojuholníky ktoré zdieľajú preponu. Takáto formácia dvoch trojuholníkov zdieľajúcich preponu sa nazýva *diamant*. Algoritmus používa dve prioritné fronty na uskutočnenie *split* a *merge* operácií [1]. *Split* fronta udržiava poradie trojuholníkov, ktoré sa majú rozdeliť, a teda vytváranie terénu jednoducho znamená rozdeľovanie trojuholníkov s najväčšou prioritou. *Merge* fronta obsahuje prioritne zoradený zoznam trojuholníkov, ktoré je treba spojiť a tak zjednodušiť terén. Toto umožňuje algoritmu využívať informácie z predchádzajúcej scény a pridať alebo odobrať trojuholníky podľa potreby [1] . Dôležité je, že ľubovoľná triangulácia môže byť dosiahnutá práve za pomoci týchto operácií. Priority vo frontách sa rátajú pomocou viacerých chybových metrík. Jednou z najväčších výhod algoritmu je možnosť priamo ovplyvňovať výsledný počet trojuholníkov, a tak dosiahnuť optimálny počet snímkov za jednotku času.

3.3.1 Binárny trojuholníkový strom

Postup pri tvorení tohto stromu je založený na delení rodičovského uzla na polovicu tak, aby vznikli dva pravouhlé rovnoramenné trojuholníky. Toto delenie prebieha od vrchného bodu, ktorý je naproti prepone, až do stredu prepony. Postupnou iteráciou môžeme takto rozvetvovať strom až na požadovanú úroveň. Každým rozvetvením sa znásobí počet trojuholníkov dvakrát [2]. Podobne ako pri implementácií kvadrantového stromu od Röttgera et al., aj v tomto prípade sú úrovne susedných uzlov od seba vzdialené maximálne o jednu úroveň.



Obrázok 3.5 názorná ukážka rekurzívneho delenia od koreňového uzla

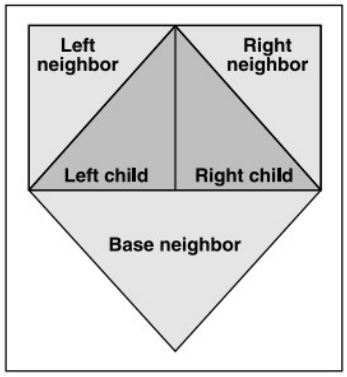
Keďže v tejto práci je implementovaný algoritmus ROAM so zmenami prezentovanými *Seumasom McNallym*, bude sa ďalej táto práca zaoberať iba implementáciou tejto variácie algoritmu.

3.3.2 Seumas McNally a jeho zmeny

Seumas McNally spravil niekoľko veľmi významných zmien v ROAM algoritme, a tie sa hlavne týkajú odbremenenia grafickej karty od prebytočných operácií a taktiež prebytočnej pamäťovej záťaže. Zmienené zmeny:

- Algoritmus neukladá v uzloch žiadne dáta pre vykresľovanie trojuholníkov
- jednoduchšia chybová metrika
- žiadna spojitosť medzi predchádzajúcou a aktuálnou scénou

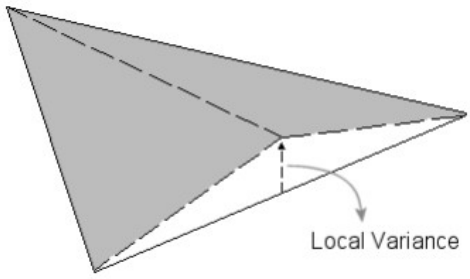
Namiesto toho, aby si každý uzol binárneho stromu ukladal informácie o scéne, obsahuje dva ukazatele na uzly potomkov, dva pre susedné uzly a jeden pre rodiča. Je výhodné alokovať viacero uzlov počas inicializácie, čo odstráni alokáciu uzlov za behu programu. Binárny uzol musí spĺňať podmienku, že jeho pravý a ľavý susedia sú rovnakej úrovne, alebo o jednu úroveň detailnejší. Jeho spodný sused, s ktorým zdieľa svoju preponu, musí byť rovnakej úrovne alebo o úroveň menej detailný. Každý trojuholník môže vytvoriť diamantovú formáciu iba so svojím spodným susedom iba v tom prípade, keď sú rovnakej úrovne. Pôvodná chybová metrika použitá v algoritme ROAM bola príliš pomalá, v tejto variácií je použitá rýchlejšia metrika. Táto chybová metrika sa nazýva *variance***.** Metrika sa zakladá na výpočte rozdielu medzi interpolovanou výškou a reálnou výškou na strede prepony trojuholníka. Jedná sa o podobnú metriku aká je použitá pri algoritme od Röttgera et al. V tomto prípade je však výsledná chyba uzla vypočítaná ako maximum z chýb jeho potomkov.



Obrázok 3.6: informácie, ktoré musí uzol obsahovať [2]

$$chyba = |cY - ((aY + bY)/2)|$$

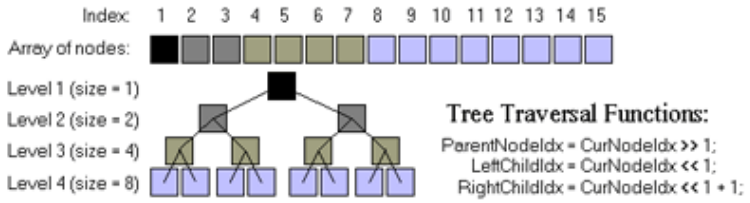
Rovnica 6: výpočet variance



Obrázok 3.7: získavanie lokálnej variance [6]

3.3.3 Implicitný binárny strom a chybová metrika

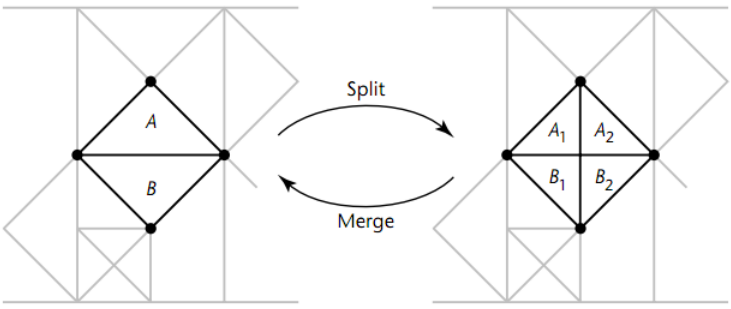
Keďže sa nám nemení podkladová výšková mapa, tak môžeme predpočítať jednotlivé chyby a tak urýchliť proces rozhodovania o rozdelení uzla. Uložiť chybovú metriku je možné do implicitného binárneho stromu. Implicitný binárny strom je uložený vo forme poľa, a tak je možné pri použití správnych techník dostať uzol z poľa v relatívne krátkom čase.



Obrázok 3.8: uloženie a navigovanie v rámci implicitného binárneho stromu [5]

3.3.4 Split-Only vs. Split-Merge

Hoci zdieľajú tieto dva algoritmy spoločný pojem a využívajú sa v ROAM algoritme, ich filozofia sa značne líši. V prvom rade *split-only* nepotrebuje udržovať prioritné fronty pre rozhodovanie o priorite rozdelenie daného trojuholníka. Algoritmus *split-only* eliminuje závislosť vykreslenia novej snímky na základe predchádzajúceho snímku.



Obrázok 3.9: obrazový popis algoritmu split-merge [1]

3.3.5 Vykresľovanie scény

Pokiaľ sa pracuje s výškovým poľom, je výhodné rozdeliť scénu na dva binárne trojuholníky kde oba korene si budú vzájomne spodnými susedmi. Tým, že dôjde k prepojeniu týchto trojuholníkov sa nemusíme obávať t-spojov, alebo trhlín, ktoré by mohli vzniknúť na hranici týchto trojuholníkov. Na začiatku nového snímku vždy začína od koreňa stromu a rekurzívne vykresľuje scénu k požadovanej úrovni detailu na základe rozhodovacieho kritéria použitého v [5].

$$f = (variance * s * 2) / d$$

Rovnica 7: rozdeľovacie kritérium

Kde *variance* je predpočítaná chyba, *s* je rozmer výškovej mapy a *d* je vzdialenosť od uzla do kamery. Pre výpočet vzdialenosti je použitá rovnica *L^2-Norm*, ktorá je presnejšia ako *L^1-Norm*. Jedná sa o výpočet vzdialenosti dvoch bodov v euklidovskom priestore.

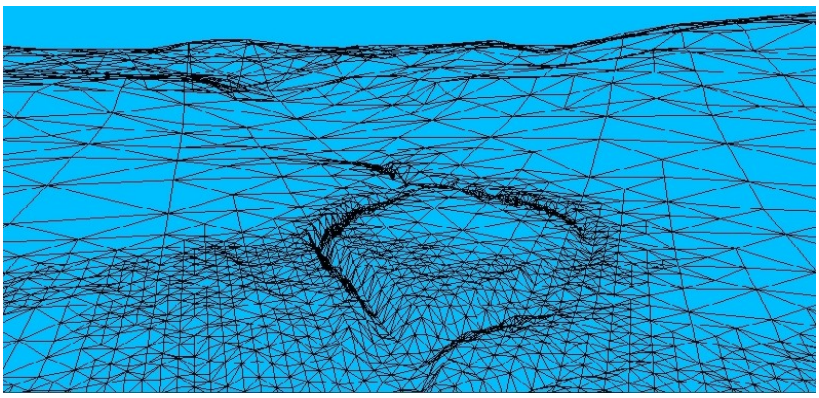
$$d = \sqrt{((camera_x - mid_x)^2 + (camera_y - mid_y)^2 + (camera_z - mid_z)^2)}$$

Rovnica 8: euklidovská vzdialenosť medzi dvoma bodmi

Kde *camera* je pozícia súradníc kamery a *mid* je pozícia bodu na strede prepony trojuholníka v trojrozmernom priestore.

Konečná podmienka pre rozdelenie uzla : $f > threshold$

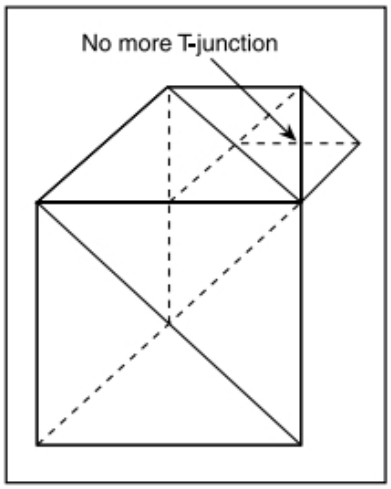
Pokiaľ je splnená táto podmienka, trojuholník môže byť rozdelený na potomkov. *Threshold* je buď konštantne zadaná hranica, alebo je po každom snímku nastavovaná algoritmom tak, aby sa po vypočítaní podmienky priblížil počet vykreslených trojuholníkov k požadovanému počtu. Ak je *threshold* nastavovaná programom, tak sa jedná o jednoduché aproximovanie hodnoty takým spôsobom, aby pri ďalšom vykreslenom snímku bol počet vykreslených trojuholníkov čo najbližšie k požadovanému počtu.



Obrázok 3.10: výsledok triangulácie

3.3.6 Rozdeľovanie trojuholníkov

Split-only algoritmus využíva v tejto situácií vlastností uzla o rozdieloch úrovne jeho a jeho susedov. Pokiaľ chceme rozdeliť uzol na potomkov, musíme najskôr skontrolovať na akej úrovni sa nachádza jeho spodný sused. Pokiaľ sa nachádza na rovnakej úrovni, a teda tvorí s ním diamant, tak ho jednoducho rozdelíme na potomkov. Pokiaľ je však o úroveň nižšie, a teda nezdieľajú rovnakú preponu, je nutné ho rozdeliť a rekurzívne rozdeliť aj jeho susedov. Tento proces sa nazýva *force-split*. Keďže tá istá situácia môže nastať pri rozdeľovaní spodného suseda, tento proces sa opakuje rekurzívne dovtedy, pokiaľ sa neodstránia všetky t-spoje a trhliny, ktoré by pri delení prvého uzla mohli vzniknúť. Následne sa musia už iba upraviť ukazatele susedných uzlov a potomkov. Tento spôsob riešenia t-spojov je oveľa jednoduchší ako spôsob, ktorý používajú blokové stromové štruktúry, ale vyžaduje rekurziu.



Obrázok 3.11: operácia force-split rekurzívne vykonávaná na viacerých uzloch [2]

4 Implementačné detaily

Implementácia programu sa dá koncepčne rozdeliť na jadro aplikácie a súčasti slúžiace pre inicializáciu grafického rozhrania a vstupných periférií. Ako programovací jazyk bol použitý jazyk C++ a rozhranie pre prácu s grafickou kartou DirectX9 a WinAPI pre prácu s oknami systému Windows. Jadro aplikácie zahrňuje trieda *terrain*, ktorá je zároveň základná trieda pre od nej odvodené triedy *roam*, *quadtree* a *bruteforce*. Každá z týchto derivovaných tried obstaráva funkcionalitu daného algoritmu. Pre zachytávanie a spracovávanie správ od vstupných periférnych zariadení, ako sú myš a klávesnica, je implementovaná trieda *inputControl*. V tej istej triede prebieha zároveň inicializácia rozhrania DirectInput. Trieda *camera* obsataráva vykreslovanie a polohovú zmenu kamery v scéne. Inicializácia rozhrania Direct3D je vykonaná v triede *initD3D*. Všetky spomínané triedy majú príponu súboru .cpp a im zodpovedajúce hlavičkové súbory príponu .h.

4.1 Inicializácia rozhraní

Inicializácia rozhrania a prostredia okien systému Windows je prvý krok v inicializácií celého programu. V inicializácií prebehne vytvorenie a naplnenie štruktúr okien za použitia rozhrania WinAPI. Následne je vytvorená trieda *initD3D*, ktorá zaobstaráva inicializáciu rozhrania Direct3D. Táto trieda taktiež nastaví osvetlenie, výplň scény a kontrolné výpisy na obrazovku. Po úspešnom inicializovaní rozhrania nastane vytvorenie a inicializácia jednej z tried algoritmov použitých pre spracovanie terénu.

4.2 Inicializácia tried spracujúcich terén

Druhým krokom je po inicializácií výber algoritmu pre spracovanie terénu. Všetky algoritmy, ktoré spracujú dáta terénu pred tým, než sú odoslané na grafickú kartu, zdieľajú niektoré základné funkcie, ktoré sú im spoločné. Tieto funkcie sú zdedené z triedy *terrain*. Sú to funkcie slúžiace pre načítanie výškovej mapy *loadHeightMap* a získanie dát o výške *getVertexHeight*. Výšková mapa je načítaná zo súboru formátu .raw. V tejto triede je implementovaná funkcia pre vytvorenie procedurálnej textúry *generateProceduralTexture,* a taktiež jej aplikovanie do výslednej scény *renderTexture*. Pri vytvorení objektu niektorej z tried nastane načítanie výškovej mapy a generovanie procedurálnej textúry. Každý z algoritmov CLOD má pre jeho vlastné potreby inicializované premenné a štruktúry, ktoré potrebuje k svojej funkcionalite.

Trieda *bruteForce* nepotrebuje inicializovať žiadne pomocné štruktúry pre delenie priestoru. Jej implementácia zahŕňa iba načítanie dát do *index bufferu*, nastavenie zdrojov dát pre vykreslenie a samotné vykreslenie grafických primitív.

Trieda *roam*, narozdiel od triedy *bruteForce* potrebuje inicializovať štruktúry pre rozdelenie dát v priestore. Pre túto úlohu používa binárny strom, ktorého uzly sú predalokované v poli uzlov binárneho stromu, okrem dvoch koreňových uzlov, ktoré sú alokované oddelene. Tieto dva koreňové uzly sú pred použitím inicializované. K uloženiu výpočtu viditeľnej chyby používa implicitný binárny strom implementovaný vo forme poľa. Tieto úkony sú prevádzané počas inicializácie, aby sa zamedzilo prebytočného využívania cyklov procesora počas behu programu.
Posledná zmienená trieda *quadtree* počas svojej inicializácie alokuje pole, ktoré slúži ako reprezentácia kvadrantového stromu a druhé pole slúžiace na uloženie vypočítanej chyby. Po úspešnej alokácií dôjde k výpočtu a propagácií chyby.

4.3 Inicializácia kamery a hlavný cyklus

Trieda *camera* počas svojej inicializácie nastaví čas, od ktorého sa začalo jej spustenie. Tento čas sa ďalej používa pre výpočet počtu snímkov za sekundu. V rámci inicializácie dochádza k nastaveniu vektorov kamery *DV_cameraPos*, *DV_lookAtPos* a *DV_lookUpPos*. *DV_cameraPos* nastaví kameru v rámci priestoru, *DV_lookAtPos* orientuje pohľad kamery do bodu v priestore. Vektor *DV_lookUpPos* ukazuje smerom hore a je použitý pre výpočty súvisiace s polohou kamery.
Hlavný cyklus programu je spustený po inicializácií všetkých potrebných tried. Cyklus je vykonávaný až pokiaľ nie je ukončený program. V každom priechode cyklu je volaná funkcia triedy *camera render_frame*, ktorá vypočítava počet snímkov za sekundu a zabezpečuje posielanie dát na grafickú kartu. Podľa výberu algoritmu pre spracovanie terénu zvolí zodpovedajúcu funkciu. Pred každým zobrazením snímku dôjde k nastaveniu transformácií prepočítaniu stien a normál pohľadového ihlanu.

4.4 Implementačné detaily algoritmu ROAM

Algoritmus ROAM používa predalokované pole uzlov *nodePool*, aby zamedzil alokácií počas behu programu, a teda má konštantnú veľkosť. Ďalšia konštanta, ktorá je v tomto programe dôležitá, je množstvo zobrazených trojuholníkov *TrianglesCount*. Pre každý binárny strom je vytvorené pole o konštantnej veľkosti(*varianceLeft, varianceRight*), slúžiace na uloženie chyby zvanej *variance*. Pokiaľ by počas prepočítavania scény bola požadovaná chybová hodnota pre uzol, ktorý sa nezmestil do tohto poľa, je jeho hodnota vypočítaná za behu pomocou nerekurzívnej variácie funkcie pre výpočet chyby *calculateVariance*. Pri vykresľovaní snímkov je volaná metóda *render*, ktorá volá rekurzívne metódy pre prepočítanie a vytvorenie scény. Tieto metódy vždy pracujú so scénou od začiatku, bez použitia dát z predchádzajúcej scény. Pri vykresľovaní trojuholníkov do scény sa

vykresľujú iba uzly, ktoré nemajú žiadnych nasledovníkov. Na základe orezania scény je redukovaný počet trojuholníkov vo výslednej scéne.

4.5 Implementačné detaily algoritmu quadtree

Algoritmus quadtree používa pre svoju činnosť dve matice. Prvá matica o veľkosti podkladového výškového poľa, ktorá stelesňuje kvadrantový strom je booleova matica. Táto matica naberá dve hodnoty 0 a 1 a pri inicializácií je celá nastavená na hodnotu 1. Pre správne propagovanie hodnôt chýb sú použité dve metódy a to *precalculateD2*, ktorá je volaná pre každý uzol a *propagateD2*, ktorá je volaná z predchádzajúcej metódy a propaguje chybu k susedným uzlom.

Pri volaní funkcie *render* sa volajú iné rekurzívne funkcie, ktoré spracovávajú scénu. Keďže sa jedná o *top-down* algoritmus, tieto funkcie sú volané iba pre istú čas dát scény a nie pre celú scénu. Scéna je redukovaná na viditeľnú časť pomocou metódy orezania scény. Avšak orezanie pomocou *AABB* je náročné a preto je nutné ho optimalizovať a čo najdôraznejšie znížiť počet iterácií v cykle metódy *performFrustumClipping*. Maximálny počet iterácií je rovný 48 pre jeden uzol, čo pri veľkom počte uzlov môže znamenať pokles výkonu aplikácie. Metóda *refineNode* začína pracovať od koreňa stromu a je volaná pre všetky jeho listy spĺňajúce podmienky rozvetvenia. Po ukončení vyhodnocovania scény, je volaná metóda *drawNode*, ktorá uloží zodpovedajúce indexy listových uzlov a následne je možné scénu vykresliť.

5 Porovnanie algoritmov

Keďže algoritmy, ktoré sme doteraz rozoberali boli navrhnuté za účelom segmentovať scénu a v závislosti na kritériách rozhodnúť o tom, akým spôsobom budú či nebudú zobrazené časti scény, môžeme ich porovnávať z viacerých perspektív. V tomto prípade sa zameriame na implementáciu a na to, aký má vo výsledku dopad na pamäťové nároky a výkon.

5.1 Štruktúra použitá pre segmentáciu scény

Oba algoritmy používajú stromové štruktúry, avšak každý rozdielnou metódou a spôsobom jej reprezentácie.

Pri algoritme *ROAM* je použitý binárny trojuholníkový strom, ktorého uzly sú implicitne uložené v predalokovanom poli. Každý jeho uzol je dátová štruktúra pozostávajúca z ukazateľov na príslušné uzly. Tieto uzly sú počas procesu vytvárania scény na seba naviazané, a tak je zaručená vzájomná spojitosť. Avšak ani jeden z uzlov neobsahuje údaje o priestorovom rozmiestnení trojuholníka, ktorého reprezentuje, a preto je nutné v každej metóde zahrnúť jeho pozíciu.

Algoritmus quadtree je založený na blokovej štruktúre kvadrantového stromu. Avšak narozdiel od algoritmu ROAM je jeho štruktúra reprezentovaná pomocou matice. V tomto prípade jednotlivé uzly na seba neobsahujú ukazatele a ich vzájomné vzťahy sú vyjadrené pomocou tejto matice. Zistenie pozície je jednoduché, a to preto, že každý kvadrant je reprezentovaný v matici.

5.1.1 Riešene t-spojov a trhlín

Tento problém rieši najjednoduchšie algoritmus ROAM vďaka jeho trojuholníkovej štruktúre. V tomto prípade pri vzniku t-spoja rozdelením uzla sa začnú rekurzívne deliť okolité uzly, na ktorých by mohol vzniknúť t-spoj. Toto delenie prebieha do vtedy, pokiaľ sa vzniknutý t-spoj neodstráni.

Pri blokových štruktúrach, ako je kvadrantový strom je však nutné použiť iné riešenie problému, pretože nie je možné rekurzívne pridávať úroveň detailu na okolitých uzloch. Riešenie je presne opačné ako v predchádzajúcom prípade. Dochádza k odoberaniu stredného vrcholu na hranách uzlov s vyššou úrovňou detailu. Avšak musí byť splnená podmienka že sa susediace uzly nesmú líšiť vo svojej úrovni o viacej ako jeden stupeň.

5.1.2 Chybová metrika

Rozdiel vo výpočte chybovej metriky sa zakladá na podobnom princípe pri oboch algoritmoch. Oba rátajú rozdiel v interpolovanej a skutočnej veľkosti pre získanie chyby.

Algoritmus ROAM používa pre výpočet chyby maximálnu chybu zo seba a svojich potomkov, pričom výpočet sa odohráva iba na prepone trojuholníkov. Výpočet tejto chybovej metriky je jednoduchý a nenáročný z hľadiska výpočtu.
Algoritmus quadtree používa pre výpočet chyby maximum z vypočítaných chýb na piatich miestach jedného uzla a to na jeho štyroch hranách a na diagonále. Ďalej sa musí táto chyba spropagovať ku susediacim uzlom, ktoré sú o jednu úroveň detailu nižšie. Táto metóda zaistí, že pri delení uzlov nenastane situácia, v ktorej by boli dva susediace uzly od seba rozdielne viac, ako o jednu úroveň detailu.

5.1.3 Orezávanie neviditeľných častí scény

Orezanie neviditeľných častí scény vedie k zvýšeniu výkonu aplikácie, a preto je výhodne ho pri zobrazovaní rozsiahlych scén aplikovať. Keďže sú použité rozdielne stromové štruktúry, tak to umožňuje použiť rozdielne obaľovacie štruktúry pre uzly stromu. Aj vhodné zvolenie obaľovacej štruktúry môže mať vplyv na výkon aplikácie.
V prípade použitia trojuholníkového binárneho stromu je možné použiť *bounding sphere*. Na základe tohto geometrického útvaru je možné rýchlo vyhodnotiť, či uzol v ňom obsiahnutý prislúcha viditeľnej časti scény alebo nie. V najhoršom prípade je nutné vyhodnotiť, či je súčet polomeru gule a skalárneho súčinu stredu gule s každou zo šiestich rovín pohľadového ihlanu vyhovujúci.
V prípade kvadrantového stromu bol použitý *axis aligned bounding box*. Tento útvar je náročnejší na vyhodnotenie, lebo v najhoršom prípade musí vyhodnotiť osem vrcholov kvádra, každý voči šiestim stenám pohľadového ihlanu. Pre kvadrantový strom nie je možné použiť *bounding sphere* ako primárny obaľovací mechanizmus, pretože nemusí dostatočne obaliť kvadrant.

5.2 Pamäťová náročnosť algoritmov

V tejto sekcií bude rozobratá pamäťová náročnosť implementácie v tejto práci. Keďže nároky na pamäť aplikácií bývajú často vysoké, oba tieto algoritmy sa snažia o zachovanie rýchlosti a pamäťovej nenáročnosti.
V prípade algoritmu ROAM je pamäťová náročnosť závislá na veľkosti podkladového výškového poľa, predalokovaného priestoru pre uzly a dve polia pre uloženie chybové metriky. Ostatné výpočty sú ukladané na zásobník systému. Ukladanie na zásobník sa použije v prípade, ak sa hodnota chyby pre aktuálny uzol nenachádza v predalokovanom poli a je nutné ju rekurzívne získať, čo závisí od úrovne vnorenia uzla v poli a v dodatočných výpočtoch sprevádzajúcich proces rozhodovania a delenia uzlov.
Algoritmus quadtree je taktiež závislý na podkladovej výškovej mape, ktorá je uložená v poli. Ďalšie pole, ktoré je nutné pre výpočet, je dvojrozmerné pole ktoré do ktorého sa ukladajú hodnoty chýb.

Ostatné sprievodné výpočty sú ukladané na zásobník systému. Pretože počet uzlov, ktoré musia byť navštívené počas každého snímku závisí od kvality zobrazenia, ale nie od veľkosti výškového poľa, požadovaná veľkosť pamäte závisí od požadovanej kvality obrazu.
V prehľadnej tabuľke je možné vidieť aké entity a dátové typy sú použité.

ROAM		**QUADTREE**	
Dátový typ	*Entita*	*Dátový typ*	*Entita*
unsigned char	Výškové pole	unsigned char	Výškové pole
binNode	Pole uzlov	float	Chybová metrika
unsigned char	Chybová metrika	unsigned char	Matica kvadrantového stromu
unsigned char	Chybová metrika		
binNode	Ľavý strom		
binNode	Pravý strom		

Tabuľka 1: pamäťová náročnosť implementácie algoritmov

Pre názornú ukážku môžeme brať do úvahy výškové pole o rozmeroch 4096 * 4096 pixelov a zistiť ako môže vyzerať predpokladaná pamäťová náročnosť algoritmov.

ROAM			**QUADTREE**		
Dátový typ	*veľkosť[Bajt]*	*entita*	*Dátový typ*	*veľkosť[Bajt]*	*entita*
unsigned char	4097*4097*1	Výškové pole	unsigned char	4097*4097*1	Výškové pole
unsigned char	2048*2048*1	Chybová metrika	unsigned char	4097*4097*1	Matica kvadrantového stromu
unsigned char	2048*2048*1	Chybová metrika	float	4097*4097*4	Chybová metrika
binNode	16384*4	Ľavý strom			
binNode	1*4	Pravý strom			
binNode	1*4	Pole uzlov			
Suma [MB]	24,07MB			96,05MB	

Tabuľka 2: názorná ukážka pamäťových požiadavkov algoritmov

Ako je možné z tabuľky vidieť, algoritmus ROAM na prvý pohľad zaberá oveľa menej miesta ako algoritmus quadtree. Najväčší rozdiel robí pole algoritmu quadtree, v ktorom je uchovaná chybová metrika, pretože každá hodnota v ňom uchovaná je typu float.
Nasledujúca tabuľka ukazuje rozdiely medzi algoritmami z hľadiska implementácie.

	ROAM	**QUADTREE**
Priestorové delenie	Binárny strom	Kvadrantový strom
Reprezentácia stromových štr.	Predalokované pole uzlov	Booleova matica
Chybová metrika	Maximálna chyba z o 4 hrán a diagonály uzla a jeho potomkov	Maximálna chyba z prepony trojuholníka a jeho potomkov
Spôsob uloženia chyby	Predalokované pole o obmedzenej veľkosti	Predalokované pole
Obaľovacie štruktúry	Bounding sphere	AABB
Odstraňovanie t-spojov	Zvyšovanie detailu na zodpovedajúcich uzloch	Odstránenie stredného vrcholu na hrane stretu dvoch úrovní detailu

Tabuľka 3: implementačný prehľad

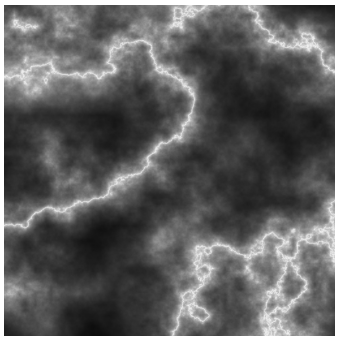
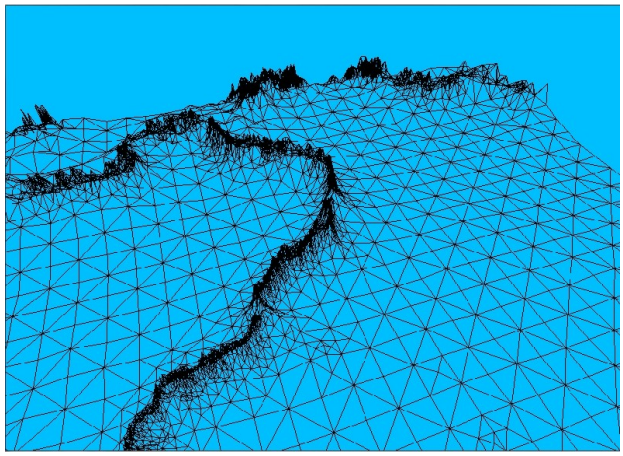
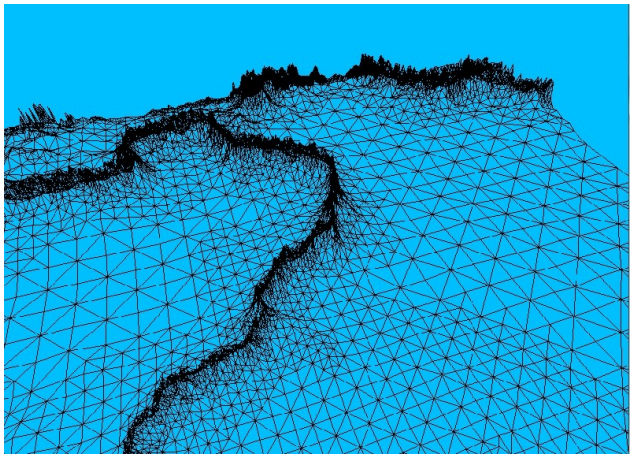
5.3 Vizualizácia a výkon algoritmov

Jednu z najdôležitejších vlastností algoritmov je správne rozmiestnenie trojuholníkov do oblastí, ktoré to vyžadujú, či už na základe vzdialenosti, alebo iných kritérií. V tejto časti bude na rôznych scénach ukázaná práca jednotlivých algoritmov. Zameriame sa hlavne na zobrazovanie členitých častí scény, aby bolo viditeľné aký má vplyv chybová metrika na výslednú scénu. Pri oboch algoritmoch sa budeme snažiť vyhodnotiť scénu pri približne rovnakom počte trojuholníkov. Po vyhodnotení vizualizácie bude zmeraný výkon implementácie v snímkoch za sekundu.

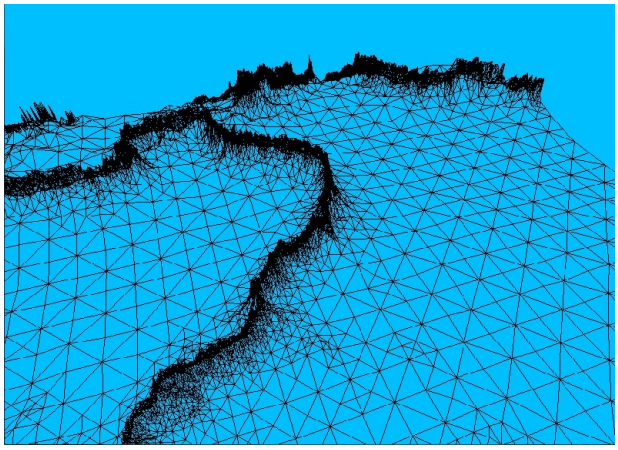
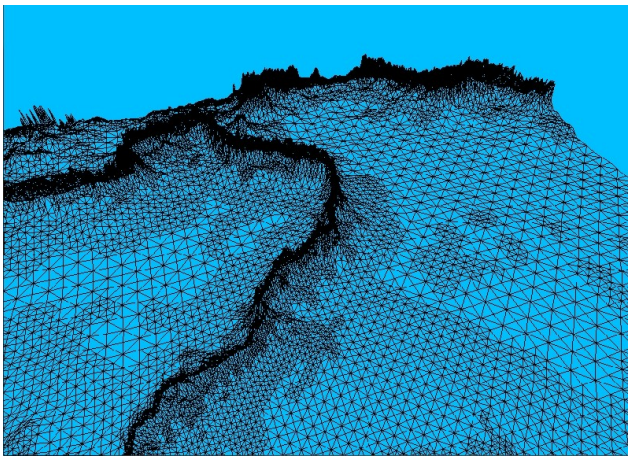
Systém na ktorom prebiehalo vyhodnotenie:

CPU	Intel Mobile Core 2 Duo, T6600 2.20GHz
Grafická karta	NVIDIA GeForce GT 240M
RAM	DDR3 4096MB
OS	Windows 7, 64bit

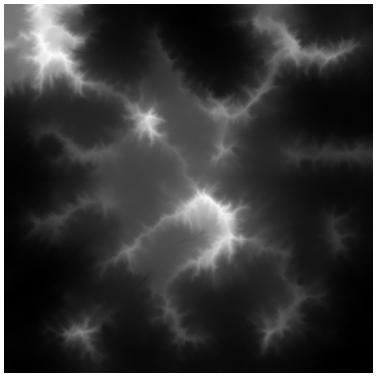
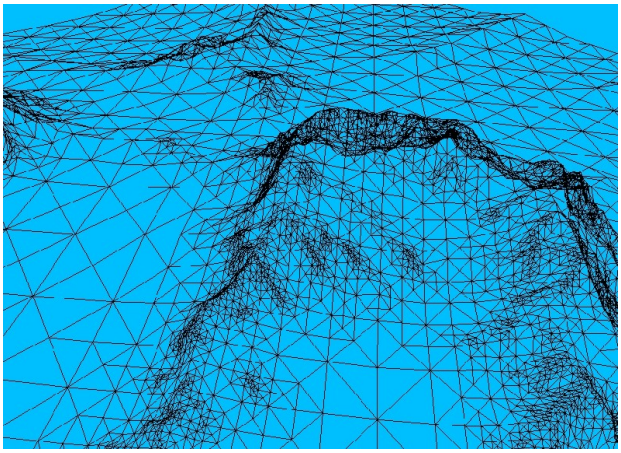
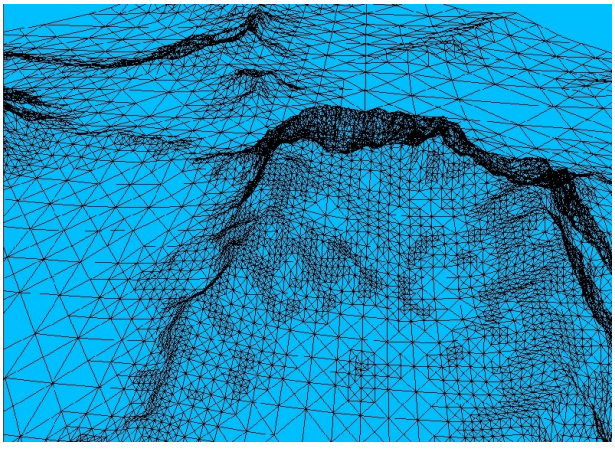
Tabuľka 4: použitý systém pri vyhodnocovaní

Názov	relief.raw [14]
rozmery [pixel]	1025 * 1025
Obrázok	
Cieľový počet trojuholníkov	*7000*
algoritmus	
ROAM Počet trojuholníkov: 7143 variance: 89.7936	
Quadtree Počet trojuholníkov: 6606 požadovaná kvalita: 4 globálna kvalita: 4	

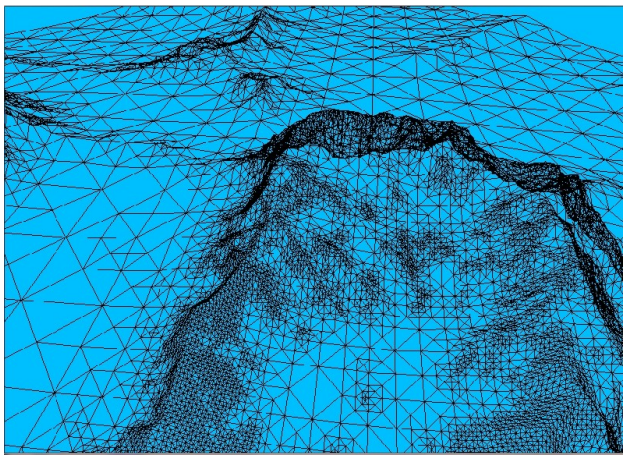
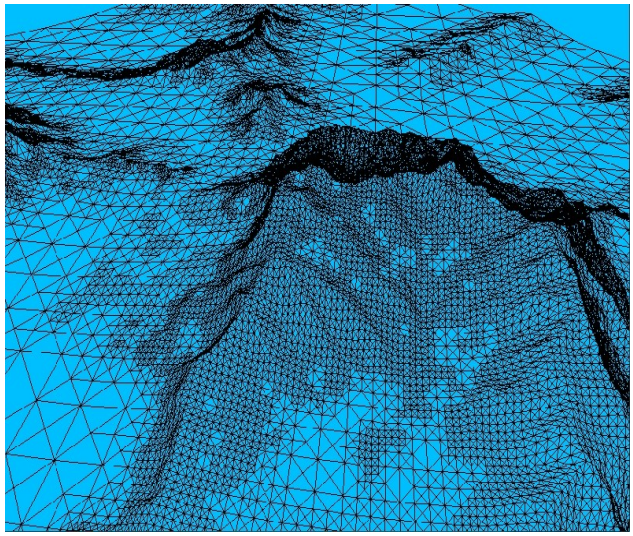
Tabuľka 5: zobrazovaná scéna : horský hrebeň

Cieľový počet trojuholníkov	*16000*
algoritmus	
ROAM Počet trojuholníkov: 16056 variance: 55.0038	
Quadtree Počet trojuholníkov: 16096 požadovaná kvalita: 12 globálna kvalita: 4	

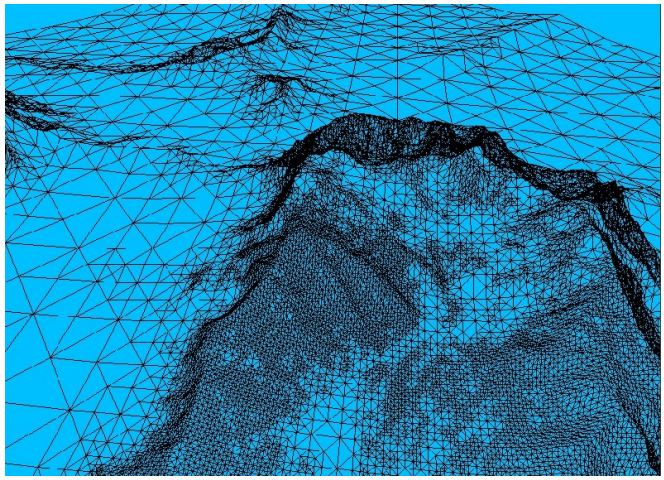
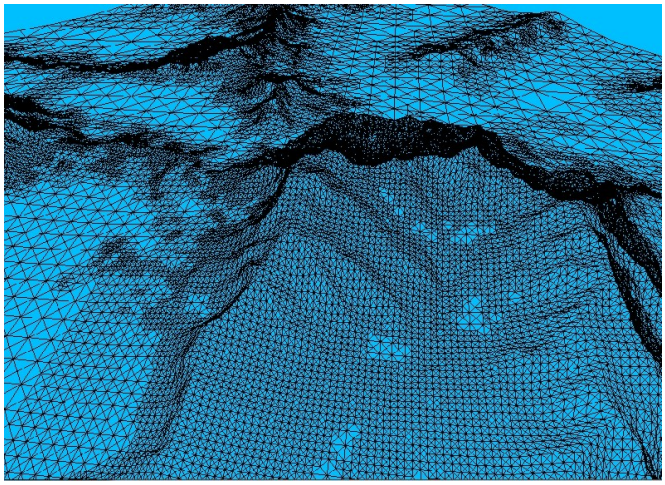
Tabuľka 6: zobrazovaná scéna: horský hrebeň

Názov	mountains.raw [17]
Rozmery [pixel]	1025 * 1025
Obrázok	
Cieľový počet trojuholníkov	*7000*
algoritmus	
ROAM Počet trojuholníkov: 6866 variance: 61.4593	
Quadtree Počet trojuholníkov: 7794 požadovaná kvalita: 8 globálna kvalita: 8	

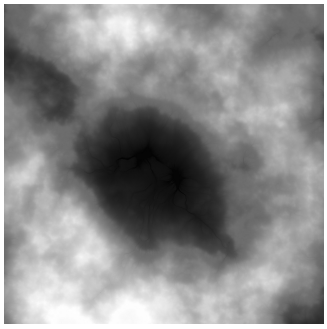
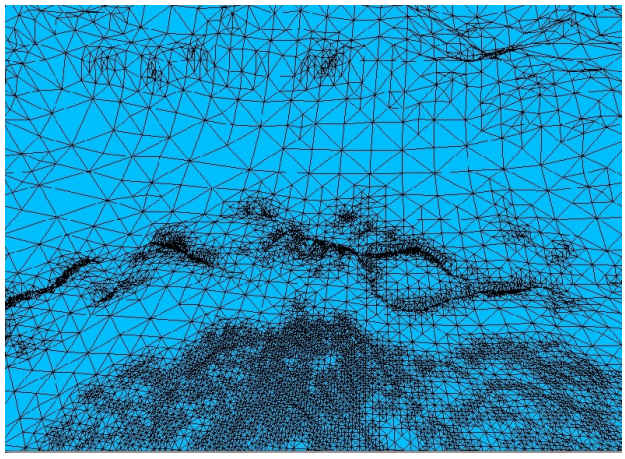
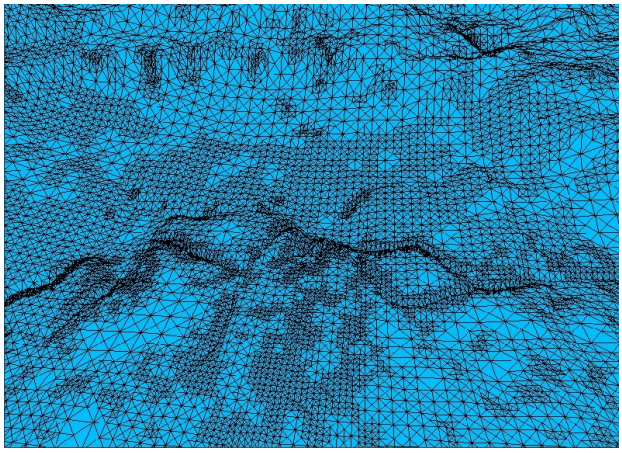
Tabuľka 7: zobrazovaná scéna:kotlina

Cieľový počet trojuholníkov	*16000*
algoritmus	
ROAM Počet trojuholníkov: 15617 variance: 45.1941	
Quadtree Počet trojuholníkov: 15909 požadovaná kvalita: 14 globálna kvalita: 8	

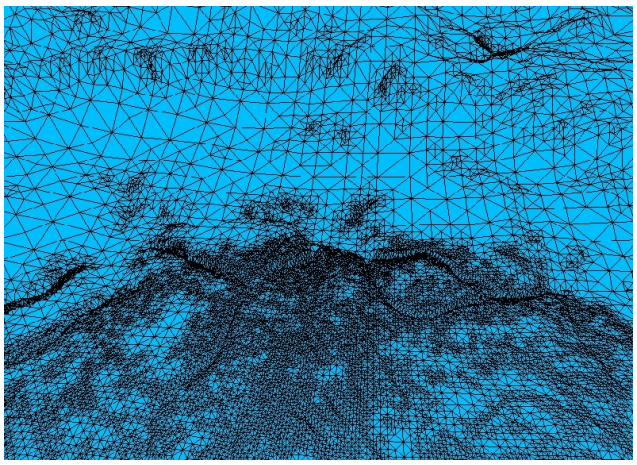
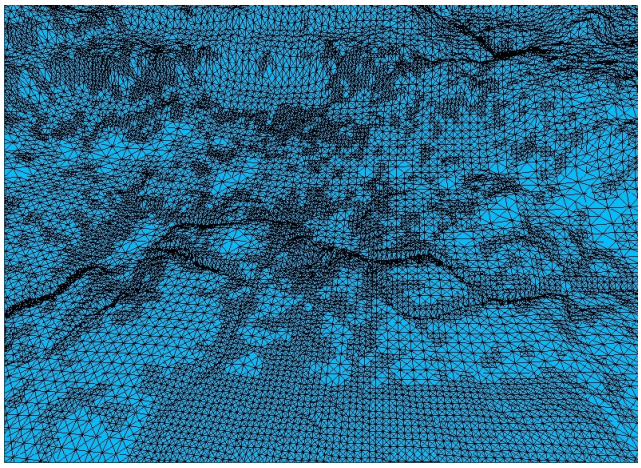
Tabuľka 8: zobrazovaná scéna:kotlina

Cieľový počet trojuholníkov	*25000*
algoritmus	
ROAM Počet trojuholníkov: 25429 variance: 38.602	
Quadtree Počet trojuholníkov: 25325 požadovaná kvalita: 23 globálna kvalita: 8	

Tabuľka 9: zobrazovaná scéna:kotlina

Názov	craterlake.raw [15]
Rozmery [pixel]	1025 * 1025
Obrázok	
Cieľový počet trojuholníkov	*16000*
algoritmus	
ROAM Počet trojuholníkov : 16 variance:38.602	
Quadtree Počet trojuholníkov: 15682 požadovaná kvalita: 19 globálna kvalita: 8	

Tabuľka 10: zobrazovaná scéna:dno jazera

Cieľový počet trojuholníkov	*30000*
algoritmus	
ROAM Počet trojuholníkov: 29711 variance: 24.4639	
Quadtree Počet trojuholníkov: 31074 požadovaná kvalita: 27 globálna kvalita: 8	

Tabuľka 11: zobrazovaná scéna:dno jazera

5.4 Zhodnotenie vizualizácie a výkonu

V predchádzajúcej časti kapitoly bolo možné vidieť rozdiely, aké nastávajú pri zobrazení scény dvoma rôznymi algoritmami. Algoritmus quadtree oveľa lepšie aproximuje výslednú scénu a to vďaka dvom faktorom:

- tesnejšia obaľovacia schránka pomocou *AABB*
- nerekurzívne delenie uzlov pri výskyte t-spojov

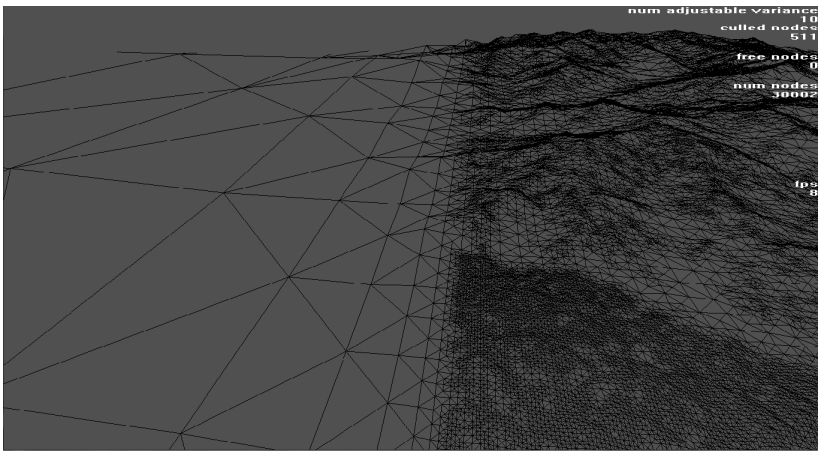
Použitie *AABB* má výhodu v tom, že umožní rozlišovať medzi troma stavmi uzla voči scéne a to:

- v scéne
- mimo
- pretína scénu

A z toho dôvodu vie lepšie určiť pozíciu ako pri algoritme ROAM, ktorý vďaka *bounding sphere* neobsahuje stav, pretína scénu, a tak môže dôjsť k vykresleniu trojuholníkov mimo scény. Tento spôsob je však rýchlejší ako pri *AABB*.

Rekurzívne delenie pri operácií *forced-split* má za následok rozdeľovanie uzlov aj mimo scénu, aby bola zachovaná spojitosť. Tým prichádzame o zobrazené uzly v scéne. V konečnom dôsledku je algoritmus quadtree presnejší v aproximácií scény.

Na tomto obrázku je viditeľné ako operácia *forced-split* rozdelí aj tie časti, ktoré by nemali byť viditeľné ako na obr. 5.1.



Obrázok 5.1: rozdeľovanie uzlov orezanej časti scény (vľavo)

Výkon implementácie:

Počet trojuholníkov	**ROAM [fps]**	**Quadtree[fps]**
5000	40	52
10000	23	37
15000	17	25
30000	8	15

Tabuľka 12: výkon algoritmov v snímkoch za sekundu

Údaje, ktoré sú uvedené v tabuľke o počtoch *fps*, sa môžu na iných systémoch líšiť. Oba algoritmy sú viazané na procesor a nie na grafickú kartu, pretože všetky výpočty prebiehajú na procesore a nie na grafickej karte, kam sú iba odosielané výsledné dáta.

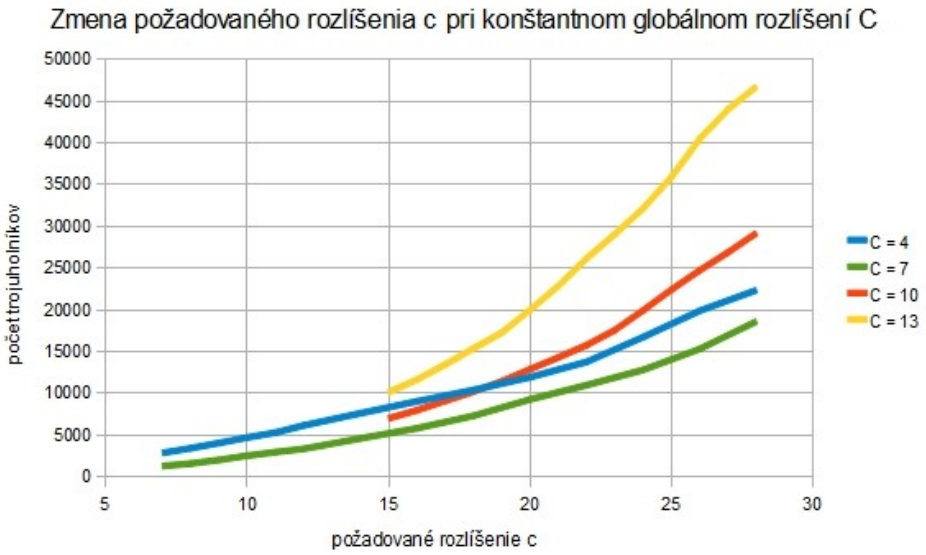
Z výsledkov porovnania týchto dvoch algoritmov, algoritmus *quadtree* vychádza ako rýchlejší a taktiež lepšie aproximuje výslednú scénu, no môže mať vyššie pamäťové nároky. V tejto kapitole boli oba algoritmy porovnané na základe ich implementačných vlastností, pamäťových nárokov, výslednej aproximácie scény a rýchlosti zobrazovania snímok za sekundu.

6 Testovanie

Testovanie je neodlučiteľná časť vytvárania programov, a preto je v tejto práci zahrnutá. V testovaní sa zameriame na kritické časti implementácie, kde môže algoritmus zmeniť chovanie a budú popísané podmienky za ktorých tak môže nastať.

6.1 Testovanie algoritmu quadtree

Z testovania bolo zistené že je možné vytvoriť t-spoje aj pri správnej propagácií. Tento nežiadúci jav je však možné vyvolať pri nesprávnom použití parametrov a to ak je parameter globálneho rozlíšenia väčší, ako parameter požadovaného rozlíšenia. Pokiaľ zostane zachované pravidlo c > C, tak v tom prípade je zaručené, že bude správne vyrátaná hodnota rozdelenia a nenastanú t-spoje.
Ďalej bolo počas testovania zistené, že veľkosť globálneho a požadovaného rozlíšenia priamo neovplyvní výsledný počet zobrazených trojuholníkov v scéne. Dôležitým aspektom je aj rozdiel medzi týmito dvoma hodnotami.

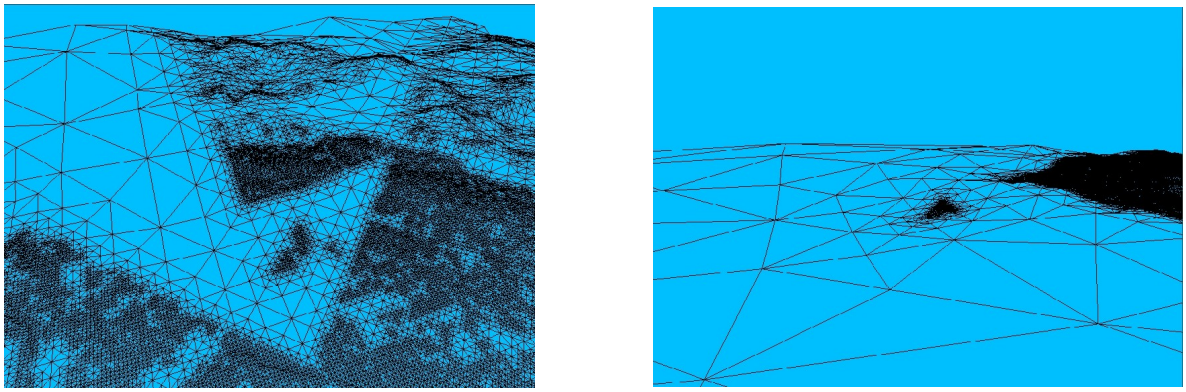


*Graf 1: zmena rozlíšení zo súboru craterlake.raw pri veľkosti mapy 1025*1025 pixelov pri statickom pohľade na scénu*

Na základe testovania sme zistili, že pokiaľ nepriradíme vypočítanej hodnote chyby dostatočnú presnosť, problém s t-spojmi pretrváva naďalej. Na tomto základe bol použitý dátový typ *float.* Hoci jeho veľkosť má zásadný vplyv na pamäťové potreby algoritmu, zabezpečuje jeho správne chovanie.

6.2 Testovanie algoritmu ROAM

Pri testovaní tohto algoritmu bolo zistené, že presnosť aproximácie scény je ovplyvnená fungovaním *split* operácií a veľkosťou alokovaného poľa uzlov, ktoré je zároveň najväčšie obmedzenie z hľadiska počtu vygenerovaných trojuholníkov. Pokiaľ dôjde k presiahnutiu tohto počtu, tak nastáva veľmi zvláštne chovanie. Pri tomto chovaní môže dôjsť k t-spojom, pretože už neexistuje žiaden voľný uzol, a tak nie je možné vykonať operáciu *forced-split* na ďalšie uzly.



Obrázok 6.1: naplnenie maximálneho počtu alokovaných uzlov

Počas testovania aplikácie bol objavený ďalší nežiadúci jav, ktorý nastal pri približovaní sa požadovanému počtu vykreslených trojuholníkov. Tento jav je možné charakterizovať ako striedanie sa dvoch trojuholníkových sietí pri pohľade na terén bez pohybu kamery. Je ho možné zamedziť tým, že nastavíme do výpočtu aproximácie chybovej hodnoty povolenú odchýlku. Tým sa zamedzí, aby sa dostala hodnota na kritický bod, ktorý je možné zvrátiť iba zmenením polohy kamery.

Pri náhlych zmenách v úrovni detailu môže nastať situácia, keď pri aproximovaní k požadovanému počtu trojuholníkov, bude pri statickom pohľade na scénu prebiehať zmena triangulácie v scéne aj po zastavení kamery na jednom mieste. To je spôsobené dlhšou odozvou medzi náhlou zmenou v umiestnení kamery a aproximáciou chyby na požadovanú hodnotu. Avšak nie je to chyba, pretože výsledná scéna sa do pár sekúnd napraví.

6.3 Záver testovania

Počas testovania aplikácie boli nájdené a odstránené všetky nežiadúce javy, ktoré sú v tejto kapitole spomínané. Testovanie prebiehalo na viacerých súboroch výškových máp a vo všetkých prípadoch bolo vykázané nezmenené chovanie algoritmov, a teda je možné predpokladať správnu implementáciu.

7 Záver

V tejto práci boli implementované a porovnané dva algoritmy používajúce techniky *CLOD*. Na základe ich implementačných vlastností, pamäťových nárokov a výkonu boli porovnané, vyhodnotené a zosumarizované. Vo výsledkoch týchto porovnávaní nakoniec obstál algoritmus *quadtree* lepšie ako algoritmus ROAM z hľadiska výkonu a lepšej aproximácie scény. Tieto algoritmy mali v posledných rokoch veľký význam a našli si uplatnenie v širokom spektre odvetví, no s nastupujúcim trendom výkonných grafických kariet je ich budúcnosť otázna. Ako môj najväčší prínos beriem zdokumentovanie, podrobné porovnanie a vystavenie silných i slabých stránok týchto algoritmov. Ciele práce sú tým pádom splnené a celkovo v siedmich kapitolách zdokumentované.

Pre pokračovanie v tejto práci existuje veľa možností. Hoci boli implementované všetky najdôležitejšie aspekty, ktoré boli nutné pre porovnávanie, má tento projekt stále miesta, na ktorých sa dá pracovať. Jedným z miest pre pokračovanie tvorby je doplnenie algoritmov o *geomorphing* a zamedziť tak náhlym zmenám v prechodoch medzi úrovňami detailu.

Ďalšou zmenou by mohlo byť zníženie pamäťovej náročnosti algoritmu *quadtree*. Keďže je veľmi dôležitý výkon aplikácie, bolo by vhodné ešte viac zefektívniť orezávanie, alebo implementovať orezanie častí zakrytých inými objektmi v scéne. V tejto práci boli použité základné procedurálne metódy požívané pre prácu s textúrami. Z tohto hľadiska je široká škála možností, ako skvalitniť výslednú scénu aplikovaním pokročilejších procedurálnych metód v textúrovaní. V neposlednom rade je dôležité, z hľadiska prehľadnosti, vytvoriť interaktívne GUI aplikácie, aby ju bolo možné jednoduchšie ovládať. Keďže sú tieto algoritmy a ich variácie využívané v počítačových hrách a simuláciách, je možné orientovať budúce pokračovania práce aj týmto smerom a pridávať tak do scény objekty, riadiť interakcie medzi nimi a pridávať vizuálne efekty.

A Použitá literatúra

[1] LUEBKE, David, et al. *Level of Detail for 3D Graphics*. [s.l.] : Morgan Kaufmann Publishers, 2003. 390 s. ISBN 1-55860-838-9.

[2] POLACK, Trent. *Focus on 3D terrain programming*. [s.l.] : Premier Press, 2003. 218 s. ISBN 1-59200-028-2.

[3] RÖTTGER, Stefan, et al. *Real-Time Generation of Continuous Levels of Detail for Height Fields* [online]. [s.l.] : [s.n.], 1998 [cit. 2011-05-16]. Dostupné z WWW: <http://www.stereofx.org/papers/TERRAIN.PDF>.

[4] DUCHAINEAU, Mark, et al. *ROAMing Terrain: Real-time Optimally Adapting Meshes* [online]. [s.l.] : [s.n.], 1997 [cit. 2011-05-16]. Dostupné z WWW: <https://graphics.llnl.gov/ROAM/roam.pdf>.

[5] *Gamasutra - the art & business of making games* [online]. 3.04.2000 [cit. 2011-05-16]. Real-Time Dynamic Level of Detail Terrain Rendering with ROAM. Dostupné z WWW: <http://www.gamasutra.com/view/feature/3188/realtime_dynamic_level_of_detail_.php>.

[6] Bradley D. (2003) Evaluation of Real-Time Continuous Terrain Level of Detail Algorithms, URL: http://zurich.disneyresearch.com/derekbradley/Papers/carleton03_TerrainLOD.pdf

[7] Ericson, Ch., Real-Time Collision Detection (2005) ISBN: 1-55860-732-3

[8] *U.S. Geological Survey* [online]. April 04, 2011 . 2009 [cit. 2011-05-10]. Glossary. Dostupné z WWW: <http://landslides.usgs.gov/learning/glossary.php#d>.

[9] *Satellite Imaging Corporation* [online]. c2001 [cit. 2011-05-10]. Digital Elevation Models. Dostupné z WWW: <http://www.satimagingcorp.com/gallery-dem.html>.

[10] S. EBERT, David, et al. *Texturing & Modeling: A Procedural Approach*. 3rd edition. [s.l.] : Morgan Kaufmann Publishers, 2003. 687 s. ISBN 1–55860–848–6.

[11] FUAN , Tsai; WAN-RONG , Lin; LIANG-CHIEN , Chen. *MULTI-RESOLUTION REPRESENTATION OF DIGITAL TERRAIN AND BUILDING MODELS* [online]. [s.l.] : [s.n.], 2008 [cit. 2011-05-13]. Dostupné z WWW: <http://isprsserv.ifp.uni-stuttgart.de/proceedings/XXXVIII/part2/Papers/33_Paper.pdf>.

[12] H. EBERLY, David . *Game Physics*. [s.l.] : MORGAN KAUFMANN PUBLISHERS, 2004. 776 s. ISBN 1-55860-740-4.

[14] *A free Terrain Synthesizer and Renderer* [online]. 2008, 2008-09-23 [cit. 2011-05-14]. Quadtree enhanced. Dostupné z WWW: <http://picogen.org/news.php?category=news>.

[15] http://www.c4d-jack.de/php/textures/index.php?spgmGal=Heightmaps

[16] *ETH Zürich - Eidgenössische Technische Hochschule Zürich* [online]. c2010 [cit. 2011-05-16]. Geodata Visualization & Interactive Training Environment. Dostupné z WWW: <http://geodata.ethz.ch/geovite/tutorials/L2GeodataStructuresAndDataModels/en/html/unit_u2Raster.html>.

[17] *Www.jmonkeyengine.org* [online]. c2007 [cit. 2011-05-16]. 3D Game Development Terminology. Dostupné z WWW: <http://jmonkeyengine.org/wiki/doku.php/jme3:terminology>.

B Zoznam príloh

Príloha 1: CD obsahuje implementáciu algoritmov, spustiteľné súbory, plagát, webstránku a manuál pre prácu s programom.