Masarykova univerzita
Fakulta informatiky

Zobrazování rozsáhlých terénů pomocí prostorových třídících struktur

Bakalářská práce

Podzim 2003 Václav Samec

Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

_____

Rád bych poděkoval svému vedoucímu, Mgr. Radku Ošlejškovi, za pomoc a vedení při tvorbě tohoto bakalářského projektu.

Shrnutí

Cílem práce bylo implementovat knihovnu umožňující efektivní zobrazení rozsáhlých terénů pomocí knihovny OpenGL. Současně byla vytvořena testovací aplikace pro porovnávání rychlosti zobrazení scény využívající implementovanou třídící strukturu a scény používající pouze základní techniky OpenGL. Tato práce charakterizuje používané techniky pro efektivní zobrazování modelů terénu.

Klíčová slova

výšková mapa, LOD (Level Of Detail), trojúhelník, binární trojúhelníkový strom, kvadrantový strom, triangulace, OpenGL, díra

Obsah:

1 Úvod

Počítačová vizualizace se zejména v posledních letech stala významnou oblastí v IT průmyslu. Ještě před několika lety bylo téměř nemyslitelné vytvořit na počítači přesný model skutečného světa. Díky rychlému vývoji v oblasti hardware, zejména grafických karet, se problém modelování skutečnosti snížil na minimum. Přestože nám stále novější grafické akcelerátory usnadňují práci, jsme limitováni jejich možnostmi a proto musíme najít efektivní způsob pro vizualizaci.
Trojrozměrná grafika se stala nepostradatelnou součástí počítačové vizualizace. Setkáváme se s ní při projektování stavebních objektů, v simulacích v počítačové chemii, při vytváření filmových efektů, v počítačových hrách apod.
Stále větší popularita počítačových her způsobuje, že se na poli výzkumu trojrozměrné grafiky zasluhuje více lidí z oblasti vývoje her než z výzkumných institucích. Tento fenomén nemůžeme přehlížet, neboť se zařadil mezi nejvýnosnější oblasti počítačového průmyslu a jsou to právě počítačové hry, které určují budoucí trendy grafického hardware.

Tato práce pojednává o trojrozměrné vizualizaci terénů v počítačové grafice. Představí používané techniky k dosáhnutí efektivního a přesného zobrazování modelů terénu.

Práce se skládá z těchto kapitol:

- *Vizualizace terénů* stručně popisuje problematiku zobrazovacích algoritmů a shrnuje potřeby efektivního algoritmu.
- *Výškové mapy* popisuje způsob reprezentace dat pro zobrazování terénu.
- *Principy zobrazovacích algoritmů* vysvětluje jemným způsobem, jak probíhá vytváření trojúhelníkových sítí a vykreslování pomocí efektivních algoritmů.
- *Používané algoritmy* se zabývá v podrobném měřítku konkrétními datovými strukturami a technikami.
- *Optimalizace* je kapitola věnovaná speciálním metodám pro urychlení zobrazení.
- *Implementace algoritmu* popisuje naprogramovaný algoritmus a výsledky naměřené z jeho testování.

U některých popisů algoritmů uvádím možnou realizaci v pseudoprogramovacím jazyce, syntaxí podobnou C++. Při testech byl použit počítač AMD Duron 1.2 GHz, 320 MB operační paměti a grafická karta ATI Radeon 9100 64 MB.

2 Vizualizace terénů

Interaktivní zobrazování složitých geometrických objektů, jako rozsáhlých terénů, je obecně těžký problém. Algoritmy pro zobrazování terénů hrají důležitou roli ve virtuální realitě, počítačových hrách, GIS, leteckých simulacích a ve vojenském plánování. Modely terénu představují rozsáhlou databázi souřadnic. Musíme najít efektivní způsob, jak data uspořádat, vykreslit a při tom zachovat vysokou rychlost zobrazování. V typických aplikacích pro zobrazování terénů vidíme na obrazovce jen malou část celého modelu. Vzdálené objekty není třeba vykreslovat v celé jejich složitosti, neboť detail jen stěží rozeznáme. Potřebujeme robustní algoritmus, který nám umožní zobrazit scénu ve vysokém detailu s minimálními nároky na výpočetní sílu.

Potřeby efektivního algoritmu lze shrnout do následujících bodů:

- **Vysoká úroveň detailu:** S vyšším detailem stoupá přesnost zobrazených dat, ale klesá rychlost. Hledáme způsob, jak vyvážit oba požadavky.

- **Rychlost:** Úzce souvisí s předchozím bodem. Přijatelná zobrazovací rychlost pro člověka se odhaduje na 30 snímků za sekundu. Pokud používáme algoritmus v komerčních aplikacích, jako jsou počítačové hry, neměla by rychlost klesnout pod řečenou hranici. Další důležitou podmínkou je fakt, že by rychlost neměla kolísat okolo stanoveného průměru.

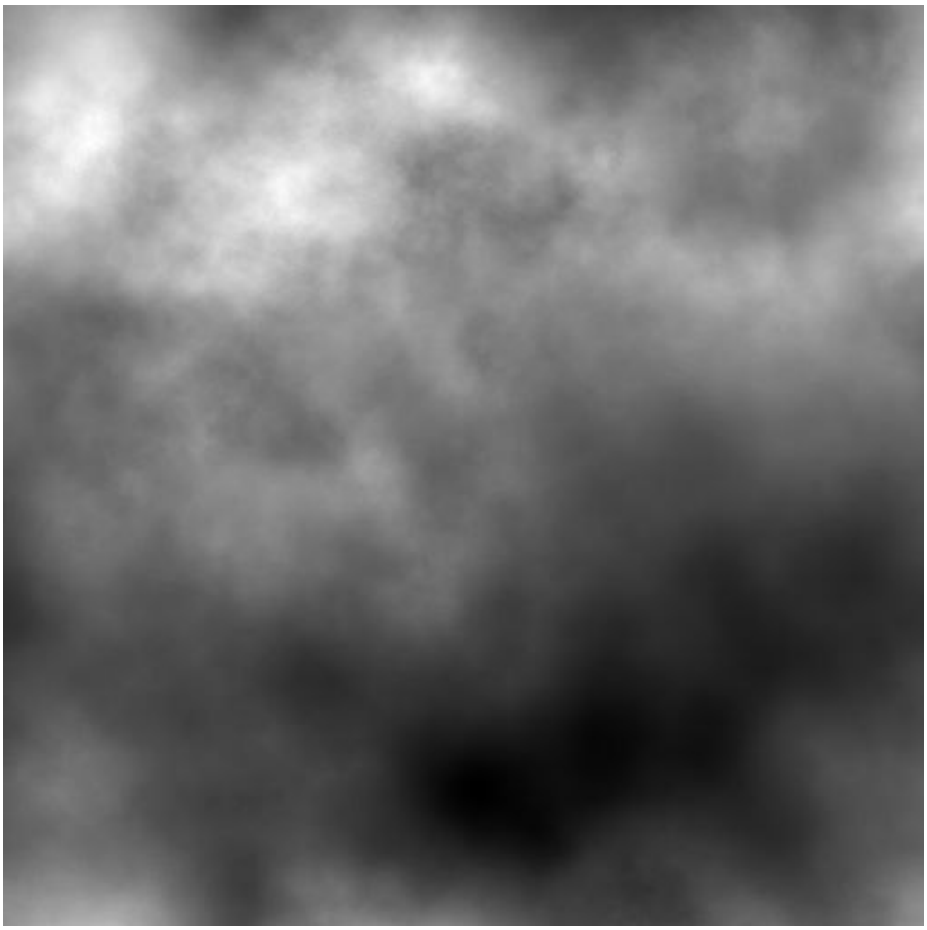
- **Přesná reprezentace dat:** Jestliže je software zaměřen na letecké simulace či vojenské plánování, požadujeme přesný model terénu.

- **Jednoduchost algoritmu:** Nechceme příliš složitý algoritmus. Hůře se provádějí případné modifikace či optimalizace.

- **Paměťové nároky:** Některé modely rozsáhlých terénů zabírají desítky gigabyte dat. Hledáme efektivní způsob, jak objem dat uspořádat a v paměti uchovat pouze potřebnou část.

3 Výškové mapy

Nejpoužívanější formát dat pro uchovávání a manipulaci s geografickými daty jsou výškové mapy. Jsou to 2D obrazová data obsahující informace pouze o výšce. Častá reprezentace výškové mapy je prostřednictvím známých grafických formátů (BMP, TGA, PNG, RAW), kde jsou data uložena po osmi bitech na jeden pixel. Výška je tedy v rozmezí 0 až 255, což je dostačující. Následující obrázek znázorňuje výškovou mapu v rozlišení 512x512, světlá místa představují vysoká místa (blíží se horní hranici 255), tmavá naopak.



Obrázek 1: Výšková mapa 512x512.

Výškové mapy se používají mimo jiné proto, že představují velice efektivní způsob pro reprezentaci 3D objektů. Prostřednictvím 1 MB osmibitové výškové mapy můžeme zobrazit až jeden milión trojúhelníků.

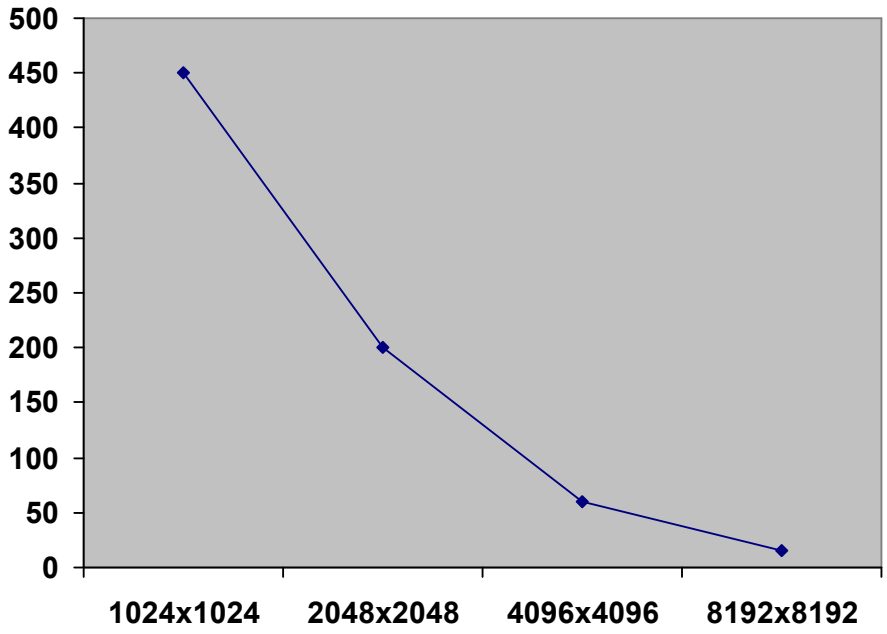
Při zobrazování se prochází výškovou mapou přes osu X a Y a počítají se 3D souřadnice jednotlivých trojúhelníků. Výšková souřadnice se spočítá jednoduchou rovnicí:

$$index = x + (y * délka\_mapy)$$

Kde x a y jsou 2D souřadnice a výsledný index je pozice ukazatele na pole výškové mapy v paměti.

4 Principy zobrazovacích algoritmů

Máme tedy výškovou mapu a chceme ji zobrazit pomocí některé knihovny pro 3D grafiku (OpenGL nebo Direct3D). Moderní grafické akcelerátory umí vykreslit až několik stovek tisíc trojúhelníků za sekundu v přijatelné rychlosti zobrazování. Může se proto zdát postačující vykreslit najednou celou výškovou mapu v jedné úrovni detailu (každý trojúhelník má stejně velký obsah). Reálné výškové mapy kopírující například pohoří Alp nelze a ani není v možnostech dostupného hardware zobrazit tímto způsobem. Následující graf ukazuje rychlost zobrazených snímků za sekundu při vykreslování výškové mapy o velikosti NxN.



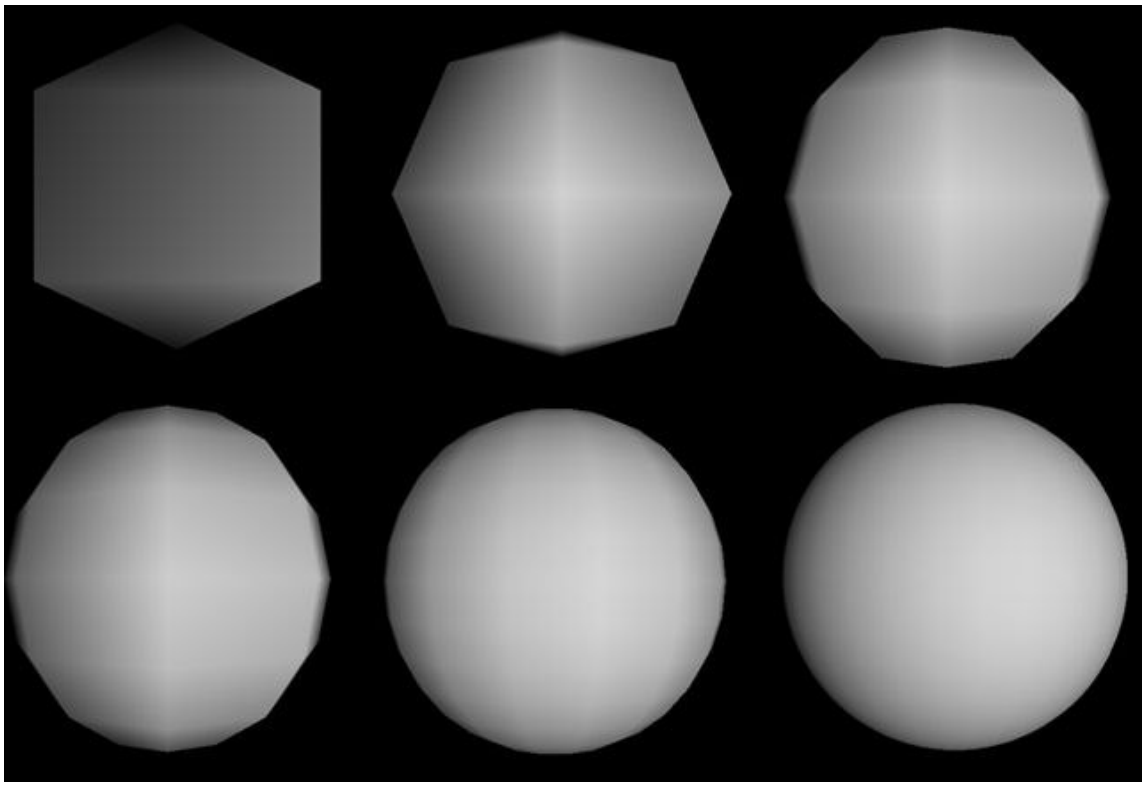
Obrázek 2: Počet snímků za sekundu při velikosti výškové mapy NxN.

Vidíme tedy, že tento způsob zobrazování není vhodný pro velké výškové mapy (respektive rozsáhlé terény). Z toho důvodu se používá pro efektivní zobrazení terénu dynamická změna detailu, známý jako Level of Detail (LOD).

4.1 Level of Detail

Cílem systému LOD je dosáhnout vysoké efektivity při zobrazování složitých geometrických objektů. Snaží se dosáhnout vysokého detailu modelu a zachovat přitom přijatelnou rychlost zobrazování. Balancuje mezi rychlostí a kvalitou. Jako příklad systému LOD si představme model lidské postavy vykreslený daleko od kamery. V tom případě vidíme pouze obrysy postavy a nezachytíme detailní strukturu (např. detail tváře, prsty apod.). Nemá tedy smysl posílat grafickému akcelerátoru statisíce trojúhelníků. Systém LOD přepočítá strukturu postavy a výsledek bude tvořit třeba jen několik desítek trojúhelníků. Samozřejmě s postupným přibližováním postavy ke kameře bude i stoupat úroveň detailu, neboť LOD

bude neustále zjemňovat strukturu postavy. Následující obrázek ukazuje vykreslenou kouli v šesti úrovních detailu.



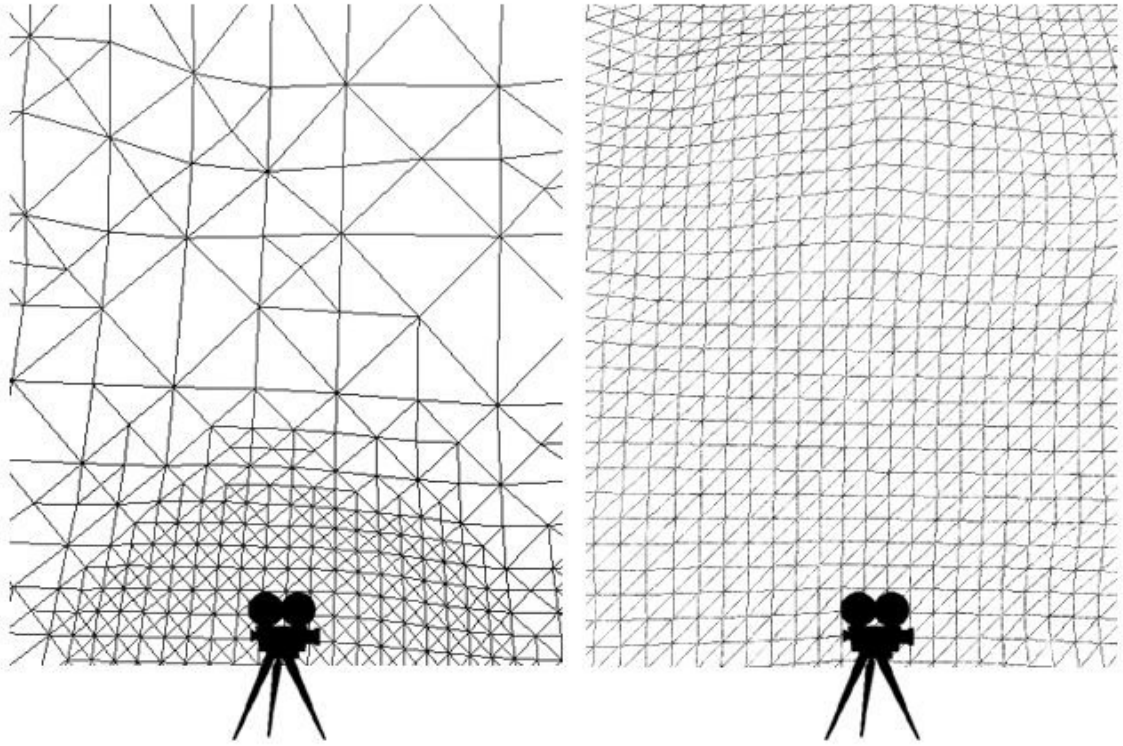
Obrázek 3: Ukázka LOD

Na obrázku můžeme pozorovat úrovně detailu jedné koule ve stejné vzdálenosti od kamery. V reálných aplikacích bychom měli pozorovat stále tentýž objekt. První model na obrázku jen vzdáleně připomíná kouli, pokud se však s kamerou dostatečně vzdálíme, obrys koule nám to připomínat bude, navíc vzdálené objekty jsou pro člověka méně podstatné a nevěnujeme jim takovou pozornost jako na objekty blízké.

4.2 Zobrazování terénů pomocí LOD

Existuje celá řada LOD algoritmů. Některé pracují tím způsobem, že si připraví dopředu několik modelů v různých úrovních detailu. Při zobrazování se pak jednotlivé úrovně střídají (v závislosti na pozici kamery a jiných faktorech). Výhodou je rychlost, protože jednotlivé úrovně detailu jsou již připravené (tj. uložené v paměti) a rychle je tedy můžeme poslat grafické kartě. Nevýhodou je neefektivní využívání paměti, N úrovní detailu znamená N modelů v paměti. Navíc dochází k vizuálním deformacím modelu při přechodech mezi úrovněmi detailu, tzv. *vertex popping*. Existují ale i algoritmy pro výpočet dynamické úrovně detailu. Princip je zřejmý, jednotlivé úrovně detailu se počítají při zobrazování, což má za následek snížení rychlosti. Dochází ale k plynulým přechodům mezi úrovněmi a v paměti je stále jeden model. Ani zde se nevyhneme viditelným skokům v přechodech, můžeme ale tento problém snadno odstranit tím, že jednotlivé body v úrovni L plynule přesuneme (tj. v nějakém čase) do další požadované úrovně L±1.

Modely terénu představují zvláštní kategorii. Výše uvedené postupy se hodí na malé a uzavřené geometrické objekty. Systém LOD vždy vypočítá jednu úroveň detailu pro celý model. Pro efektivní zobrazení terénu potřebujeme algoritmus, který nám spočítá několik úrovní detailu najednou a ty pak reprezentují terén jako celek. Jako typický příklad si představme kameru uprostřed rozsáhlého terénu. V okolí kamery je terén členitý (je nastavena nejvyšší úroveň detailu), s přibývající vzdáleností od kamery úroveň klesá. Rozdíl oproti zobrazení v plném rozlišení je zanedbatelný, neboť detail na vzdálených objektech nepostřehneme. Následující obrázek ukazuje rozdíl mezi terénem zobrazeným pomocí LOD a v plném rozlišení, oba modely jsou projektovány z ptačí perspektivy.



Obrázek 4: LOD vs. jedna úroveň detailu.

Hlavní rozdíl je ihned vidět, v okolí kamery je trojúhelníková síť hustá, postupem od kamery se jednotlivé trojúhelníky zvětšují. Je pak velký rozdíl poslat grafické kartě 10.000 nebo 100.000 trojúhelníků.

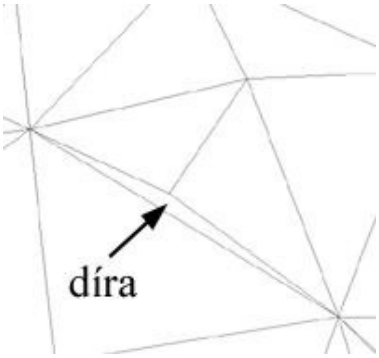
Obecný princip těchto algoritmů spočívá v rekurzivním rozdělování trojúhelníkové sítě do požadované hloubky. Tento proces nazýváme *triangulace*.

Jsou tři klíčové problémy, se kterými je třeba se vypořádat při vytváření trojúhelníkové sítě:

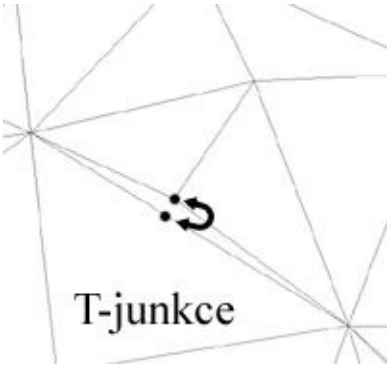
- Správný výběr úrovně detailu. Rozdíl mezi výsledným modelem a terénem v plném rozlišení (v jedné úrovni detailu) musí být minimální, tento rozdíl se také někdy nazývá *viditelná chyba*.

- Eliminace děr. Při vytváření víceúrovňového detailu je důležité zachovat souvislé přechody mezi sousedními úrovněmi. Ve výsledné síti se mohou objevovat tzv. díry – viz. obrázek 5. Jsou výsledkem špatného napojení úrovně L s úrovní L±1.

- Vyvarování se T-junkcí. T-junkce je operace používaná k odstraňování děr. Spočívá ve spojení dvou problematických částí sítě, které tvoří díru – viz. obrázek 6. Důvody, proč se jim vyhnout, jsou dva. Může se objevit velmi tenká („vlasová“) díra a způsobit tak viditelné prosvítání. Za druhé T-junkce způsobují problémy při použití Gouraudova stínování (Gouraud shading).



Obrázek 5: Díra – nenavazující úrovně detailu.



Obrázek 6: T-junkce – spojení problematických částí.

4.3 Chybová metrika

Zatím jsme jenom zmiňovali, že se trojúhelníková síť rozděluje do hloubky v okolí kamery a na vzdálený terén se použije hrubší detail. Problém ale nastává, pokud jsou v terénu časté výškové změny, např. kopcovitý terén. V tom případě má výsledný model často vysokou viditelnou chybu, neboť nelze vytvořit přesný model kopcového terénu pomocí nízkého detailu. Tento problém řeší chybová metrika. Je to mechanizmus, který projde do hloubky celý terén a ohodnotí jeho části podle výškových rozdílů. Často je to časově náročný proces a proto se provádí na začátku při inicializaci.

Výsledky chybové metriky jsou uloženy v databázi, při triangulaci systém LOD porovnává hodnoty v databázi a podle velikosti případně rozdělí terén do větší hloubky. V některých aplikacích (např. v počítačových hrách) je někdy potřeba měnit strukturu terénu za běhu, musí se tedy znovu spočítat i chybová metrika. V těchto případech se používá „odlehčená verze“, která není příliš přesná, ale je rychlá a většinou dostačující.

Nyní již máme dostatek prostředků k definování obecného algoritmu pro zobrazování terénů pomocí systému LOD.

```
void inicializace() {

        vyskovaMapa = nactiMapu();          //načte výškovou mapu a předá ukazatel
        spocitejChybovouMetriku();          //spočítá metriku

}

void spocitejChybovouMetriku() {

        if (aktualniHloubkaZanoreni < maximalniHloubka) {

                rozdelTeren();                      //rozdělí terén na části
                spocitejChybovouMetriku();          //spočítá metriku v rozdělených částech

        } else {

                ziskejSouradnice(vyskovaMapa);
                spocitejHodnotuMetriky();
                pridejHodnotuDoDatabaze();
        }

}

//vypočítá LOD a pošle trojúhelníky na výstup, provádí se v každém snímku
void spocitejTriangulaci()    {

        zjistiHodnotuMetriky();             //zjistí metriku v aktuálním zanoření
        zjistiPoziciKamery();
        spocitejUkazatelZanoreni();         //vezme v úvahu metriku a pozici kamery

        if (ukazatelZanoreni > limit) {

                rozdelTeren();              //rozdělí terén na části
                spocitejTriangulaci();      //spočítá triangulaci v rozdělených částech

        } else {
```

```
                ziskejSouradnice(vyskovaMapa);
                posliTrojuhelnikNaVystup();
        }

}
```

5 Používané algoritmy

Algoritmy používající LOD pro efektivní zobrazování terénů můžeme rozdělit do dvou základních skupin:

1. Algoritmy založené na pravidelném (uniformním) rozdělování

Dělíme do skupin podle použitých datových struktur:

(a) kvadrantové stromy

- při triangulaci terén rozdělujeme vždy na čtyři stejně velké části
- pokud dojdeme do požadované hloubky, zastavíme dělení a vytvoříme trojúhelníkový vějíř na každém listu

(b) binární trojúhelníkové stromy

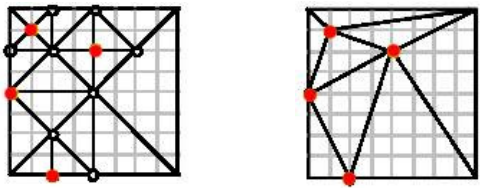
- terén nejdříve rozdělíme na dva rovnoramenné trojúhelníky a ty pak rozdělujeme podle přepony
- po skončeném dělení každý list reprezentuje finální trojúhelník

(c) ostatní

- v převážné většině aplikací se používají zmíněné datové struktury, můžeme se však setkat s kombinací obou

2. Algoritmy založené na nepravidelném rozdělování

Někdy také nazývané „Triangulated Irregular Network“ (TIN). Vytváří nepravidelnou trojúhelníkovou síť na základě chybové metriky. TIN má schopnost reprezentovat terén menším počtem trojúhelníků se stejnou viditelnou chybou než algoritmy založené na pravidelném rozdělování podle datových struktur. Další výhodou je, že při triangulaci nevznikají díry, odpadá tedy problém jejich odstraňování. Nevýhody ale převažují, algoritmy jsou komplikované a je tudíž složité provádět optimalizace. Díky nepravidelné stromové struktuře trojúhelníků se obtížně provádí prohledávání sousedních listů, což celkově zvyšuje náročnost algoritmu. Z toho plynoucí problémy mohou nastat např. při hledání texturových souřadnic či ořezávání neviditelných částí terénu.



Obrázek 7: Pravidelné rozdělování podle binárního stromu vs. TIN

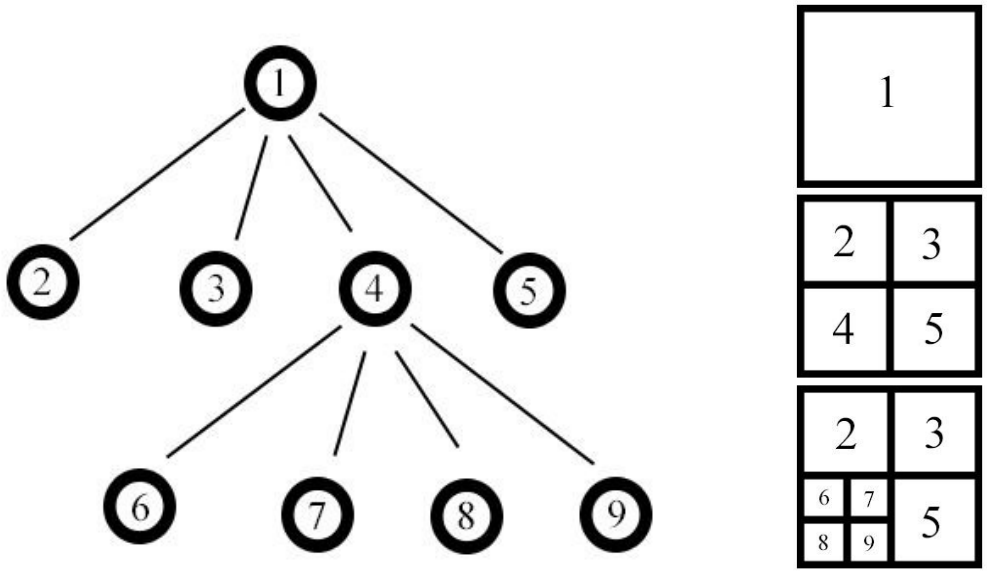
Na předchozím obrázku je vidět rozdíl ve vytvořené trojúhelníkové síti. Označené body určují hlavní výšky terénu. Výsledný model je v obou případech stejný, TIN si vystačil s osmi trojúhelníky, druhý model je tvořen osmnácti, z toho čtyři vznikly kvůli eliminaci děr. I přes tento velký rozdíl se v převážné většině aplikací používají algoritmy pro pravidelné rozdělování pomocí stromových struktur. Následující odstavce se budou věnovat detailnějšímu pohledu na příslušné datové struktury. Budou v nich popsány některé konkrétní techniky pro triangulaci.

5.1 Kvadrantové stromy

Kvadrantový strom je strom, kde každý uzel má čtyři potomky nebo je listem.

```
struct QuadTree {

        QuadTree *pChild1;
        QuadTree *pChild2;
        QuadTree *pChild3;
        QuadTree *pChild4;

        QuadTree *pParent;

        Leaf *pLeaf;
}
```

Při reprezentaci terénu kořen stromu chápeme jako čtverec ohraničující základní terén. Při triangulaci ho rozdělíme na čtyři stejné bloky a přiřadíme je následníkům kořene. Stejným způsobem zvětšujeme strom do požadované hloubky.

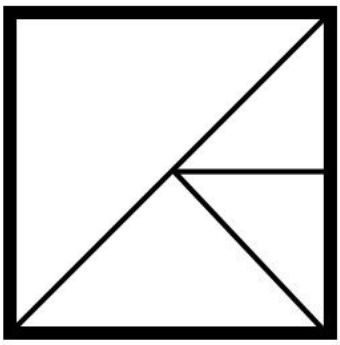


Obrázek 8: Kvadrantový strom a rozdělování terénu.

5.1.1 Vytváření trojúhelníků

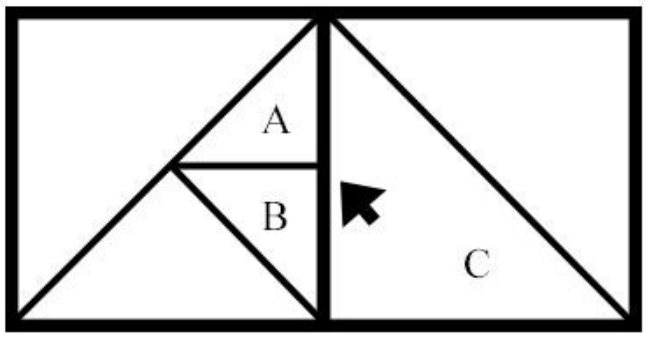
Podle hodnot chybové metriky zastavíme dělení. Listy stromu tvoří čtverce. Každý list rozdělíme na dva rovnoramenné trojúhelníky. Podle chybové metriky se ještě případně dělí „listový“ trojúhelník. Ale pouze do jedné úrovně, vytváří se trojúhelníkový vějíř kolem centrálního bodu na přeponě „listového“ trojúhelníku. Maximální počet dělených trojúhelníků je čtyři. Celkem tedy maximum trojúhelníků na jeden list je osm. Je to z důvodu limitace OpenGL při vykreslování trojúhelníkových vějířů (samozřejmě se můžeme setkat s různými implementaci kvadrantového stromu, kde probíhá dělení na jiném principu, tento způsob byl vybrán pro svoji jednoduchost).

```
struct Leaf {

        TriangleFan *pLeftChild;
        TriangleFan *pRightChild;

        QuadTree *pParent;
}
```



Obrázek 9: List a jeho rozdělování na trojúhelníky.

Při tomto dělení je třeba dávat pozor na úroveň detailu trojúhelníku v sousedním listu, může dojít k vytvoření díry. V případě, že se úrovně liší, jednoduše ignorujeme prostřední bod v hraniční části trojúhelníku.



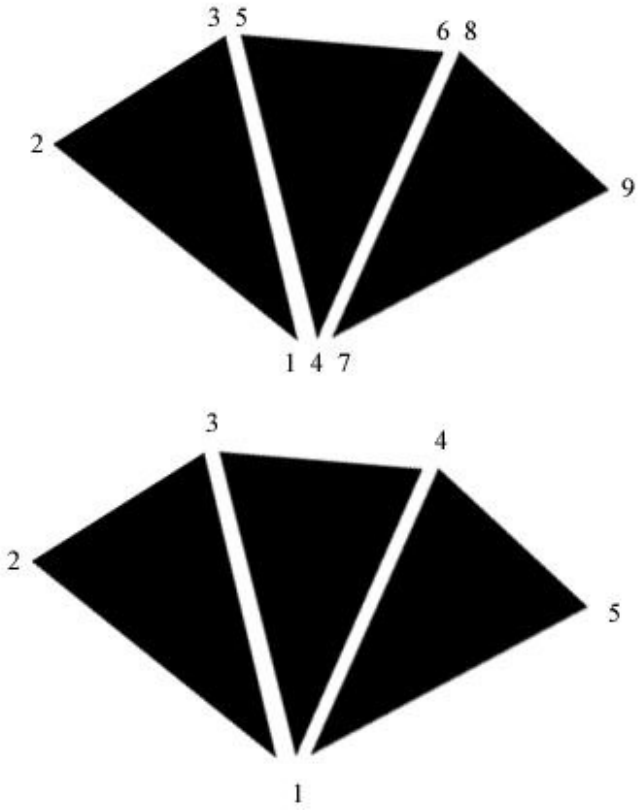
Obrázek 10: Rozdíl v navazujících úrovních detailu, v terénu bude díra.

Podle předchozího obrázku by došlo ve výsledném modelu s nejvyšší pravděpodobností k díře. Společný krajní bod trojúhelníků A a B bude mít nejspíš jinou výšku než ostatní dva krajní body. Ignorací „problémového“ bodu odstraníme díru a spojíme tím trojúhelníky A a B. Tato metoda je velice jednoduchá a rychlá, zjednodušuje nám však výslednou síť. Podobně by šel tento problém vyřešit opačným způsobem, tedy rozdělením trojúhelníku C. Toto rozdělení by však mohlo způsobit další rozdělení atd. Nehledě na to, že bychom museli nějakým mechanismem propagovat rozdělení do trojúhelníku C.

5.1.2 Vykreslování

Při vykreslování procházíme strom od kořene přes všechny uzly. Pokud narazíme na list, vykreslíme ho pomocí trojúhelníkových vějířů. Používání vějířů je efektivnější způsob při práci s trojúhelníky v OpenGL. Zatímco posloupnost nezávislých trojúhelníků vyžaduje pro každý trojúhelník tři souřadnice, vějíře odstraňují redundanci bodů. Používají se v situaci, kdy máme v řadě několik trojúhelníků navzájem sousedících a všechny sdílejí jeden bod.

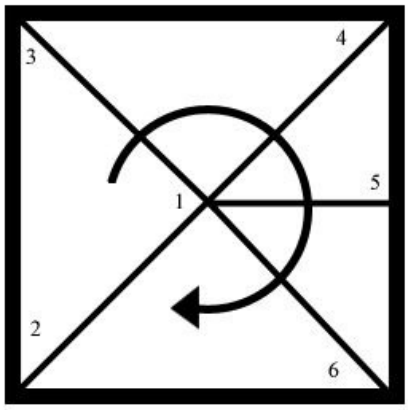
```
struct TriangleFan {

        Vertex3D *pVertex;
        int count;
}
```



Obrázek 11: Nezávislé trojúhelníky a vějíř.

Podle testů je zobrazování modelů pomocí vějířů až o 15% rychlejší.

Pokud narazíme na list kvadrantového stromu, navštívíme nejdříve levou část, zadáme souřadnice prvního trojúhelníku a postupujeme podle směru hodinových ručiček. Na každý další navštívený trojúhelník přidáme souřadnice jednoho bodu. Maximální počet trojúhelníků na jeden vějíř je osm, to však splníme, neboť připouštíme nejvýše čtyři trojúhelníky na každou část listu.



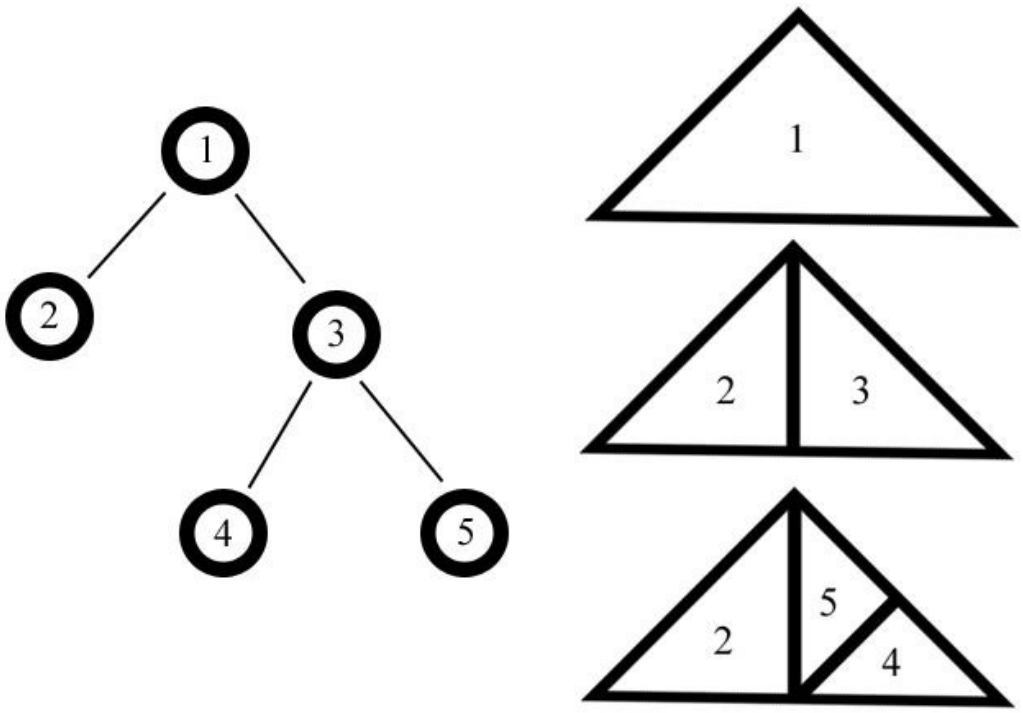
Obrázek 12: Vytváření trojúhelníkového vějíře.

Při konstrukci vějíře plníme strukturu TriangleFan jednotlivými body. V závěrečné fázi příkazy OpenGL vykreslíme vějíř.

```
void RenderLeaf(Leaf *pLeaf) {

        glBegin(GL_TRIANGLE_FAN);

        RenderFan(pLeaf->pLeftChild);
        RenderFan(pLeaf->pRightChild);

        glEnd();
}
```

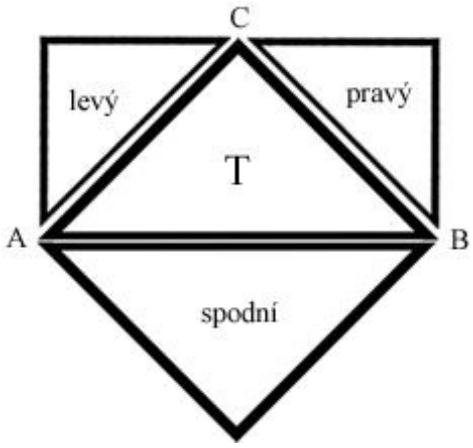
5.2 Binární trojúhelníkové stromy

Binární trojúhelníkové stromy byly poprvé představeny v systému ROAM (Real-time Optimally Adapting Meshes) [1]. Jsou speciálním případem binárního stromu. Uzel má právě dva potomky nebo je sám listem. Každý uzel je pravoúhlý rovnoramenný trojúhelník. Pokud existuje uzel, jeho potomky definujeme jako pravoúhlé rovnoramenné trojúhelníky vzniklé rozdělením uzlu podle prostředního bodu na přeponě. Následující obrázek ilustruje binární trojúhelníkový strom rozdělený do třetí úrovně.



Obrázek 13: Binární trojúhelníkový strom.

Každý trojúhelník kromě kořenového má tři sousedy. Pro trojúhelník T s body A, B, C, kde hrana (A, B) je přeponou, definujeme levého souseda takového, který sdílí hranu (A, C). Podobně pravý soused sdílí hranu (B, C). Konečně definujeme spodního souseda, který sdílí hranu (A, B).



Obrázek 14: Definice sousedů.

Jestliže trojúhelník T má úroveň detailu (hloubku uzlu) L, pak pro jeho sousedy platí následující. Pravý a levý soused mají úroveň detailu právě L nebo L+1. Spodní soused může mít právě úroveň L nebo L-1. Na obrázku 14 mají pravý a levý soused úroveň L+1, spodní soused úroveň L.

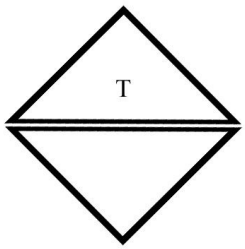
Programová definice binárního trojúhelníkového stromu:

```
struct BTTree {

        BTTree *pLeftChild;
        BTTree *pRightChild;

        BTTree *pLeftNeighbor;
        BTTree *pRightNeighbor;
        BTTree *pBaseNeighbor;
}
```

5.2.1 Triangulace

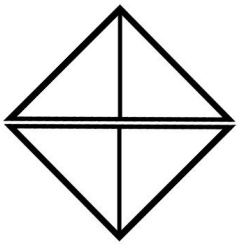
Reprezentace terénu pomocí binárních trojúhelníkových stromů vyžaduje terén o velikosti (2^N + 1) * (2^N + 1). Nic nám však nebrání rozdělit vstupní data na menší části v požadovaných velikostech. Triangulace probíhá následujícím způsobem. Vstupní terén rozdělíme na dva pravoúhlé rovnoramenné trojúhelníky a na oba pak voláme stejnou triangulační funkci. Základní princip je stejný jako u kvadrantových stromů, trojúhelníky rozdělujeme do hloubky podle chybové metriky a pozice kamery ve scéně.

Při rozdělování musíme nějakým mechanismem zamezit vytváření děr. K tomu se používá jedno pravidlo, a to, že trojúhelník rozdělíme pouze tehdy, pokud jeho spodní soused je ve stejné úrovni detailu. Tuto dvojici nazýváme diamant.



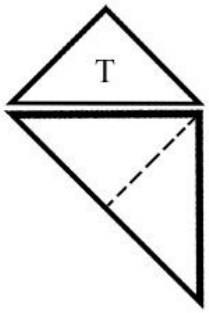
Obrázek 15: Diamant.

Z definice sousedů víme, že spodní soused může být právě ve stejné nebo nižší úrovni detailu. Pokud je ve stejné úrovni, rozdělíme náš trojúhelník T a s ním i jeho spodního souseda.



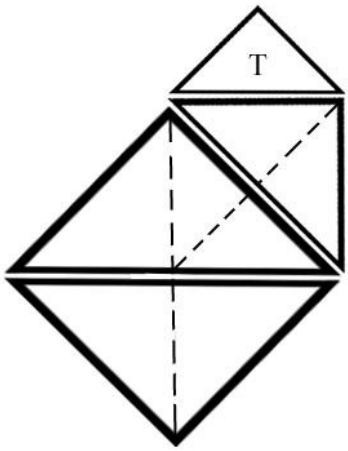
Obrázek 16: Rozpůlení diamantu.

V případě, že je spodní soused v nižší úrovni, musíme ho rozdělením dostat na stejnou úroveň a poté postupujeme jako v předchozím případě.



Obrázek 17: Vynucené rozdělení spodního souseda.

To ale samozřejmě může způsobit řetězovou reakci vynucených rozdělení. Tímto způsobem se nám může rychle zvýšit celkový počet trojúhelníků ve výsledné síti.



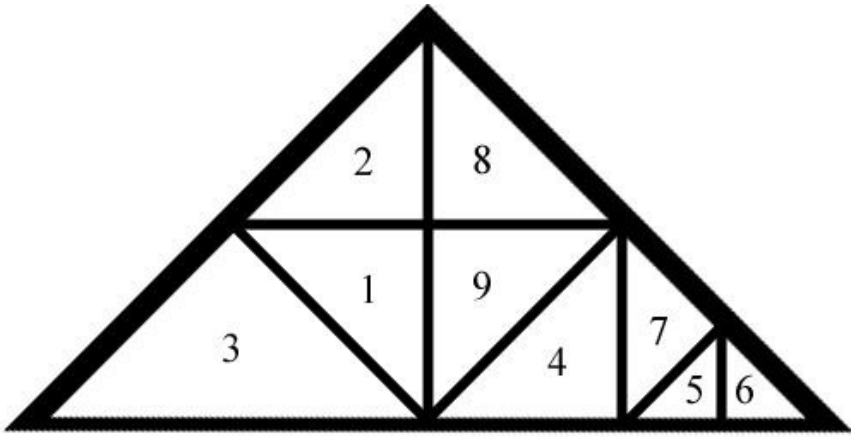
Obrázek 18: Řetězová reakce.

V praxi to ale považujeme za nejmenší zlo ve srovnání s „děravým“ modelem.

Tímto způsobem dojdeme do požadované hloubky a zastavíme dělení. Každý list představuje finální trojúhelník.

5.2.2 Vykreslování

Při vykreslování procházíme strom zleva doprava. Počínaje kořenem navštívíme levého potomka. Rekurzivně navštěvujeme levé potomky, až dojdeme k listu. Ten vykreslíme. Pak se rekurzí vracíme a navštívíme nejbližšího pravého potomka. Opět rekurzivně procházíme přes levé potomky až narazíme na list. Pokud je při návratu pravý potomek listem, tak ho vykreslíme. Tímto způsobem pokračujeme, dokud nevykreslíme všechny listy binárního stromu.

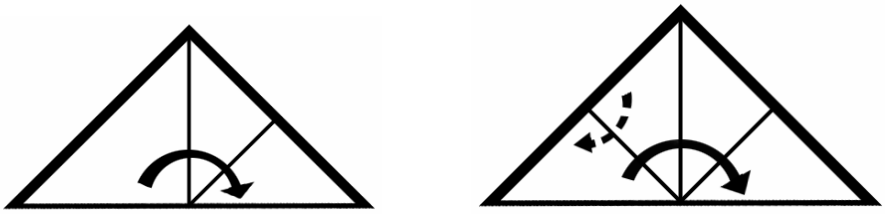


Obrázek 19: Procházení binárním trojúhelníkovým stromem.

```
void Render() {

        if (! isLeaf() ) {

                Render(pLeftChild);
                Render(pRightChild);

        } else
                Draw(Leaf);
}
```

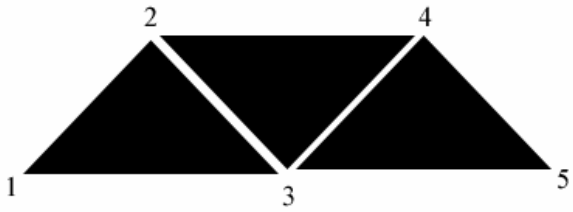
5.2.3 Optimalizace vykreslování

Binární trojúhelníkové stromy nám nenabízí tak snadnou cestu pro efektivní vykreslování pomocí vějířů jako kvadrantové stromy. Pokud se rozhodneme pro vějířovou variantu, musíme při procházení listů dávat pozor na směr a velikost detailu. V principu se to provádí tak, že pokud narazíme na list, přidáme jeho souřadnice do pomocného bufferu s tím, že první souřadnice je centrální bod, okolo kterého vykreslíme vějíř. V dalším listu porovnáme souřadnice s prvním prvkem v bufferu a přidáme jeden bod na konec. Pokud souřadnice neodpovídají, buffer vykreslíme. V některých případech musíme přesypat obsah bufferu a změnit první (centrální) prvek. Trojúhelníky musí jít podle směru hodinových ručiček, jinak je nám OpenGL ořízne. Průměrný počet trojúhelníků na vějíř je tři až čtyři. Přestože je tato technika poměrně komplikovaná, jisté zrychlení nastane.



Obrázek 20: Vykreslení vějíře.

Stejného zrychlení dosáhneme pomocí trojúhelníkových pásů (triangle strip). Ty jsou podobné jako vějíře, nepřipouštějí redundanci bodů. Používají se v případech, kdy máme souvislou řadu navzájem sousedících trojúhelníků a vytváří tak pás.

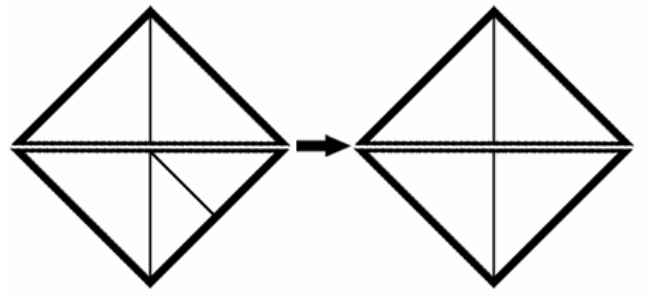


Obrázek 21: Trojúhelníkový pás.

Při použití pásu pomůže indexovat jednotlivé vrcholy listu celými čísly. Pokud přijdeme na vrchol, který má již index, nic neděláme. Např. na prvním listu přiřadíme vrcholům trojúhelníka čísla 1, 2, 3. Další list sdílí s předchozím indexy 1, 2, jeho indexace bude tedy 1, 2, 4. Podobně další listy mohou sdílet některé vrcholy. Indexaci provádíme pro stanovený počet listů. Při vykreslování porovnáváme indexy vrcholů a podle toho vykreslujeme trojúhelníkový pás. Najít optimální pás je poměrně těžký problém, průměrný počet trojúhelníků vychází pět až šest na jeden pás.

5.2.4 Prioritní fronty

Existuje jistá optimalizace, která nám garantuje při zobrazování zachování snímkové koherence. Klasické rozdělovací metody jsou výpočetně náročné a v některých aplikacích (zejména v počítačových hrách) potřebujeme usměrnit výkon bez snížení kvality zobrazení. Systém ROAM navrhl takovou optimalizaci založenou na dvou prioritních frontách. Do front ukládáme operaci rozdělení a spojení. Zavádíme tedy novou operaci spojení, která spojí dva listy do jednoho. Stejně jako u rozdělení klademe podmínku, že při spojení musí být spodní soused předka na stejné úrovni. V opačném případě si jeho spojením vynutíme stejnou úroveň.


Obrázek 22: Vynucené spojení.

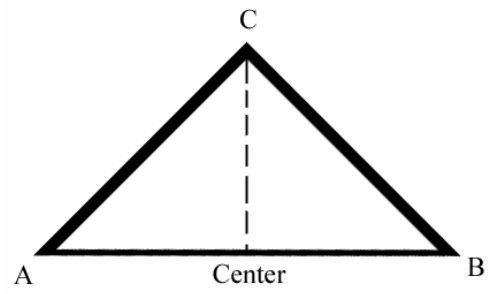
V první části algoritmu rozdělujeme známým způsobem terén do požadované hloubky podle chybové metriky. Poté už jenom spojujeme nebo rozdělujeme listy stromu podle pozice kamery. Fronty jsou řazeny podle chybové a kamerové metriky.

5.3 Chybové metriky

V této části si uvedeme některé metody k výpočtu chybových metrik.

První metrika, kterou si ukážeme, je velmi jednoduchá na implementaci a umožňuje poměrně rychlý výpočet. Pro každý trojúhelník v triangulaci počítáme chybovou metriku takto:

Mějme rovnoramenný pravoúhlý trojúhelník T s vrcholy A, B, C, kde hrana (A, B) je přeponou. Mějme bod Center, který určuje prostřední bod na přeponě.



Obrázek 23: Výpočet chybové metriky.

Pak chybová metrika pro trojúhelník T je dána metrikou jejích potomků T1, T2 jako absolutní hodnota rozdílu výškové souřadnice bodu Center a průměru výškových souřadnic bodů A, B.

metrika(T) = max(metrika(T1), metrika(T2)) + |vyska(Center) – prumer(vyska(A), vyska(B))|

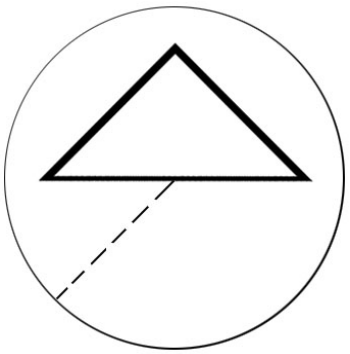
Při výpočtu procházíme rekurzivně terén do hloubky a počítáme chybovou metriku trojúhelníku v aktuálním zanoření, trojúhelník rozpůlíme a opět spočítáme jeho metriku. Zastavíme se požadované hloubce, použijeme pro to podmínku nejmenšího trojúhelníku.

Další metrika je daleko přesnější a náročnější na výpočet, vychází z hodnot předchozí metriky. Je kamerově závislá. Nechť (p, q ,r) je pozice bodu $v \in T$ v kamerových souřadnicích, kde T je trojúhelník v triangulaci. Nechť (a, b, c) je vektor v kamerových souřadnicích odpovídající vektoru w = (0, 0, mT) ve světových souřadnicích, kde mT je chybová metrika trojúhelníku T vypočítaná podle předchozího odstavce. Pak počítáme metriku bodu $v \in T$ takto:

$$metrika(v) = \frac{2}{r^2 - c^2}\sqrt{(ar - cp)^2 + (br - cq)^2}$$

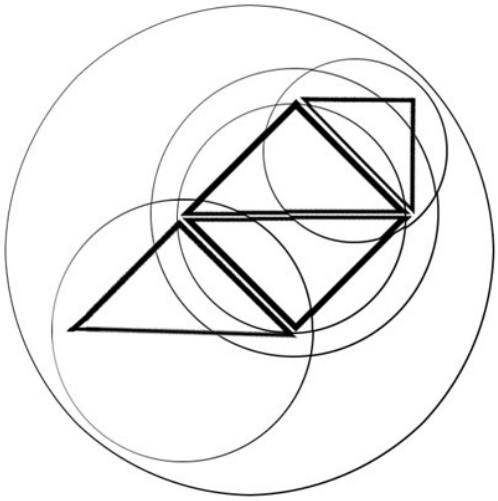
Pro trojúhelník T počítáme maximální metriku ze všech bodů $v \in T$.

Pro efektivní testování trojúhelníků, zda je máme či nemáme rozdělit podle pozice kamery, se používá tzv. *sférická* metrika. Vychází z principu, že každý trojúhelník je obalen kružnicí se středem prostředního bodu na přeponě.



Obrázek 24: Sférická metrika.

Pokud kamera vstoupí do oblasti kružnice, trojúhelník rozdělíme. Trojúhelníků je však ve scéně velké množství, z toho důvodu „obalujeme“ do kružnic celé seskupení. Ty zase obalíme. Vytvoříme tím stromovou strukturu kružnic. Testujeme pak jenom malou část trojúhelníků, což značně urychlí výpočet.



Obrázek 25: Struktura sférické metriky.

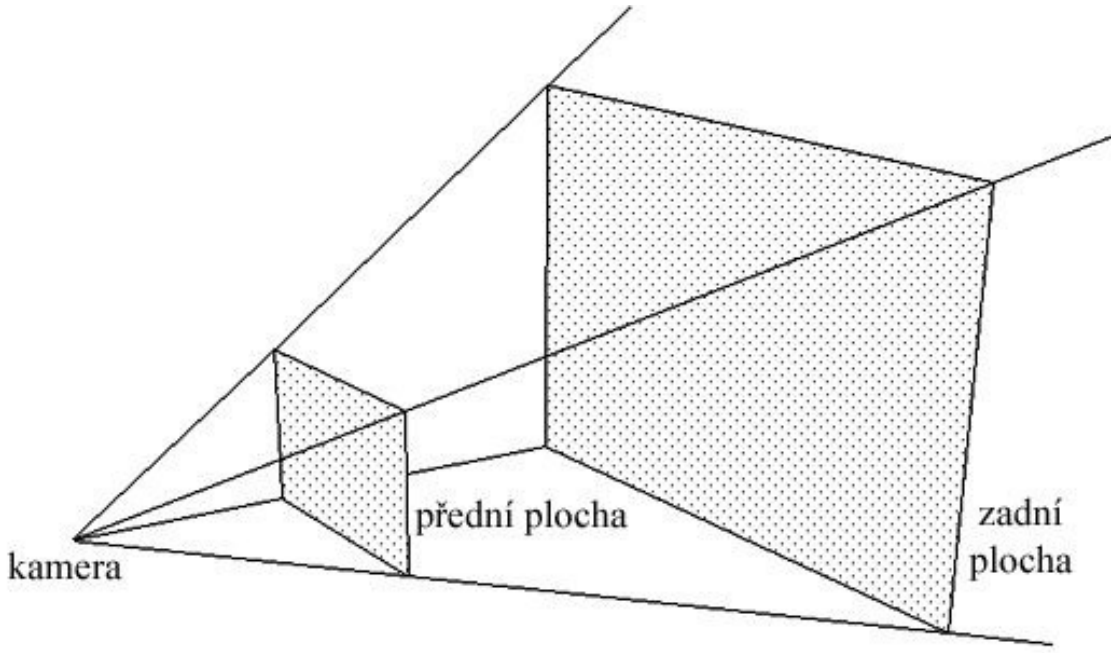
6 Optimalizace

Ukážeme si některé používané techniky na zvýšení rychlosti a kvality zobrazování.

Již v předchozích odstavcích jsme zmínily použití trojúhelníkových vějířů nebo pásů. S jejich užitím zvýšíme rychlost zobrazení až o 15%. Dalším snadným a efektivním způsobem, jak zrychlit zobrazení, je používání vertexových polí místo standardních příkazů OpenGL. Příkazem *glVertex* řekneme OpenGL pozici jednoho bodu kresleného polygonu. Pro jeden trojúhelník zavoláme třikrát příkaz *glVertex*. Pro 10.000 trojúhelníků je to 30.000 volání jedné funkce. V těchto případech se vyplatí použít vertexová pole. Body jednotlivých trojúhelníků ukládáme do bufferu a ten na konci voláním jedné funkce vykreslíme.

6.1 Frustum Culling

Frustum Culling je hojně používaná technika k odstraňování neviditelných částí scény, která nejsou v úhlu kamery. Testujeme objekty ve scéně proti rovnicím roviny definující náš zorný úhel. V OpenGL tvoří kamerový úhel šest ploch – přední, zadní, horní, spodní, levá a pravá.



Obrázek 26: Kamerový systém v OpenGL.

Při testování každého bodu všech objektů ve scéně by jistě došlo k extrémnímu zpomalení. Proto každý model „obalíme“ v neviditelném geometrickém objektu a ten pak testujeme proti rovnicím. Typicky se používá koule nebo kvádr pro jejich jednoduchou matematickou reprezentaci. Při testování jednoduše dosadíme každý bod objektu do rovnice roviny. Pokud je výsledek větší než nula, jsme uvnitř zorného úhlu.

Při vykreslování terénu rozdělíme model na stejně velké bloky (kvádry) a vykreslíme viditelné.

Zmíněnou metodu lze značně urychlit testováním pouze tří rovnic roviny, a to přední, levou a pravou. Při zobrazení se nám však ořežou i některé viditelné části. To nastane hlavně v případech, kdy kamera směřuje dolu nebo nahoru. Tuto metodu proto používáme v aplikacích čistě zaměřených na efektivní zobrazení terénu.

6.2 Back-Face Culling

Každý polygon má dvě strany – přední a zadní. Touto jednoduchou metodou ořízneme zadní (neviditelnou) část polygonu. Na moderních grafických kartách je však rozdíl v rychlosti zobrazení velmi malý.

6.3 Geomorphing

Geomorphing je metoda používaná k odstranění viditelných skoků v přechodech mezi úrovněmi detailu, známých jako *vertex popping*. Používá se k tomu jednoduchá rovnice na plynulé přesunutí výškového bodu do požadované pozice.

Pro nějakou časovou proměnnou t z intervalu <0, 1> a výškový bod A, který chceme přesunout do pozice B, definujeme bod At takto:

$$At = (1 - t)*A + B*t$$

At je pak výškový bod, který simuluje plynulý přechod z bodu A do bodu B.

6.4 Pokročilé techniky

Existuje několik algoritmů, které za účelem zvýšení rychlosti používají pro vykreslování trojúhelníkové klastry. Algoritmus RUSTiC [7] přišel první s nápadem použití klastrů. Využívá stejného principu jako binární trojúhelníkové stromy, negeneruje však při triangulaci samostatné trojúhelníky. Dopředu si vypočítá podmnožiny binárních trojúhelníkových stromů a vykresluje najednou celé seskupení. Tato metoda je rychlejší, zvyšuje počet trojúhelníků ve scéně za nulovou ztrátu rychlosti. Je však vyšší složitost algoritmu, vysoké nároky na paměť a je potřeba další extra výkon k vypočítání klastrů v přípravné fázi. Velmi podobným směrem se ubírá algoritmus CABTT [4], který nemá takové nároky na paměť, neboť trojúhelníkové klastry ukládá přímo na grafické kartě.

7 Implementace algoritmu

Byl implementován algoritmus založený na binárních trojúhelníkových stromech. Současně byla vytvořena testovací aplikace pro porovnání rychlosti zobrazení s modelem terénu v plném rozlišení.

Aplikace byla naprogramována pomocí knihovny OpenGL a je určena pro prostředí Windows. Samotný algoritmus používá pouze funkce z knihoven C++ a OpenGL, je tedy platformově nezávislý. Ostatní části testovací aplikace volají systémové funkce Win32 API pro vytvoření okna a interaktivních prvků, jako jsou menu a dialogová okna. Program byl implementován ve vývojovém prostředí Visual C++ 6.0.

Kvůli efektivnímu ořezávání neviditelných částí, byl terén rozdělen na stejně velké části. S jednotlivými bloky se pak zacházelo jako se základním terénem, tedy počítání chybové metriky, triangulace a vykreslování. Hraniční trojúhelníky sousedních bloků jsou navzájem propojeny, takže nedochází k vytváření děr. Byla použita chybová metrika založená na rozdílu výšky centrálního bodu na přeponě a její průměrné výšky.

Algoritmus na zobrazení modelu v plném rozlišení byl pojat podobným způsobem, terén byl rozdělen na samostatné bloky. V přípravné fázi se vytvořil pro každý blok DisplayList, který se používá k rychlému vykreslování statických částí scény.

Testovací aplikace umožňuje přepínání mezi jednotlivými implementacemi. K dispozici je možnost zapnout Frustum či Back-Face Culling. Bylo implementováno také automatické generování texturových souřadnic pro lepší vizualizaci.

Celá aplikace je rozdělena pro lepší orientaci do několika nezávislých tříd, kde každá třída je uložena v samostatném souboru. V následujícím přehledu bude uveden stručný popis základních tříd:

CCamera: třída pro práci s kamerou, zachytává zprávy Windows pro klávesnici a myš a počítá aktuální pozici kamery, její metody jsou volány v každém snímku

CFrustum: třída pro výpočet ořezávacích rovin, z aktuální pozice kamery a zorného úhlu vypočítá v každém snímku tři rovnice roviny definující přední, levou a pravou plochu

CNoLOD: tato třída spravuje model terénu v plném rozlišení, jeho rozdělení na části, uložení do DisplayListu a vykreslování

CRoam: obsahuje základní metody pro vytváření trojúhelníkové sítě pomocí binárních trojúhelníkových stromů, podobně jako třída CNoLOD skrytě volá metody jiných tříd pro rozdělování a počítání metrik

CTerrain: slouží k načítání výškových map ze souboru do paměti, také obsahuje metodu pro výpočet automatických texturových souřadnic

CTexture: načítá 24-bitové bitmapy a ukládá je do textur OpenGL

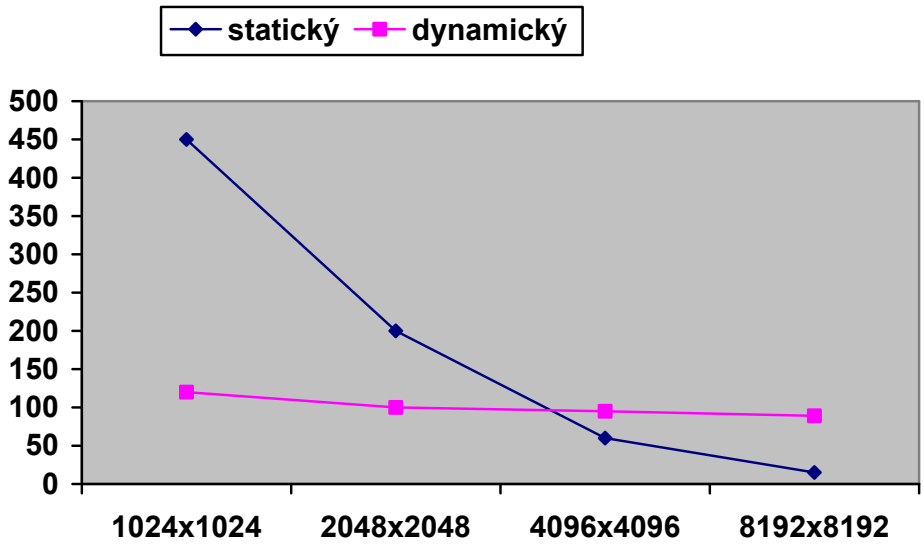
CVector: třída pro práci s trojrozměrnými vektory, obsahuje přetížené matematické operátory a typické metody pro práci s trojrozměrnou matematikou, jako je délka vektoru a skalární či vektorový součin

Ovládání testovací aplikace je vzhledem k typickému uspořádání okenních aplikací ve Windows velice intuitivní. Pomocí menu otevřeme výškovou mapu a počkáme než proběhne

inicializace. Stiskem levého tlačítka myši v oblasti okna aktivujeme interaktivní režim. Pohybem myši rotujeme kamerou ve scéně a klávesami A, S, D, W nebo šipkami hýbeme kamerou do světových stran. Opětovným stiskem levého tlačítka se vypne interaktivní režim a my můžeme opět ovládat aplikaci myší.

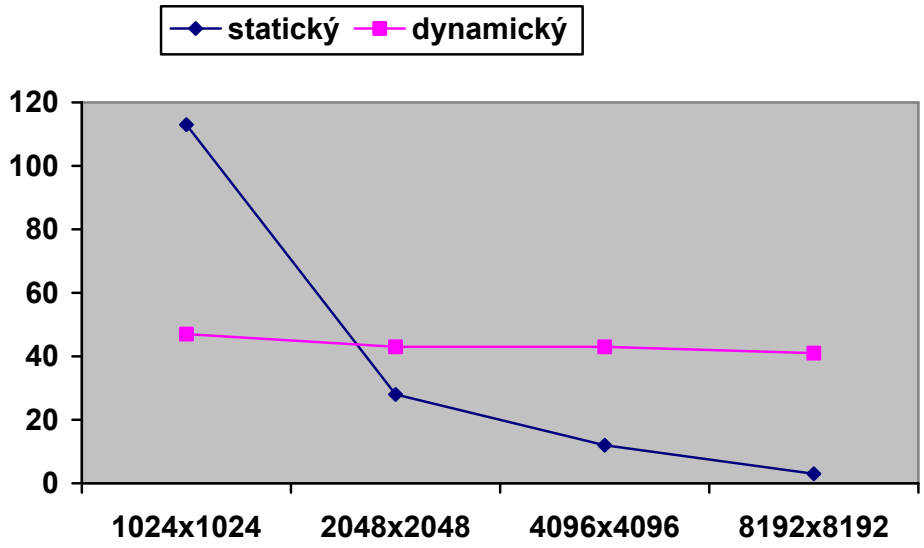
7.1 Výsledky testů

Při testování na různých výškových mapách bylo vidět, že model v plném rozlišení s přibývajícím počtem trojúhelníků ztrácí na rychlosti. V malých výškových mapách s velkým náskokem vede, neboť byla při testech použita poměrně moderní grafická karta s rychlým grafickým procesorem.



Obrázek 27: Počet snímků za sekundu při zobrazení výškové mapy NxN.

Můžeme vidět, že dynamický algoritmus téměř neztrácí rychlost na větších výškových mapách. Je to dáno tím, že se jen velmi málo zvyšuje celkový počet trojúhelníků ve scéně. V nejvyšší testované mapě se statický algoritmus dostal pod únosnou hranici 20 snímků za sekundu. Dynamický se stále pohybuje vysoko nad 80 snímky. Při zobrazování drátového modelu terénu je situace ještě zřetelnější. Model v plném rozlišení ztrácí na rychlosti již při zobrazení druhé výškové mapy o 45%.



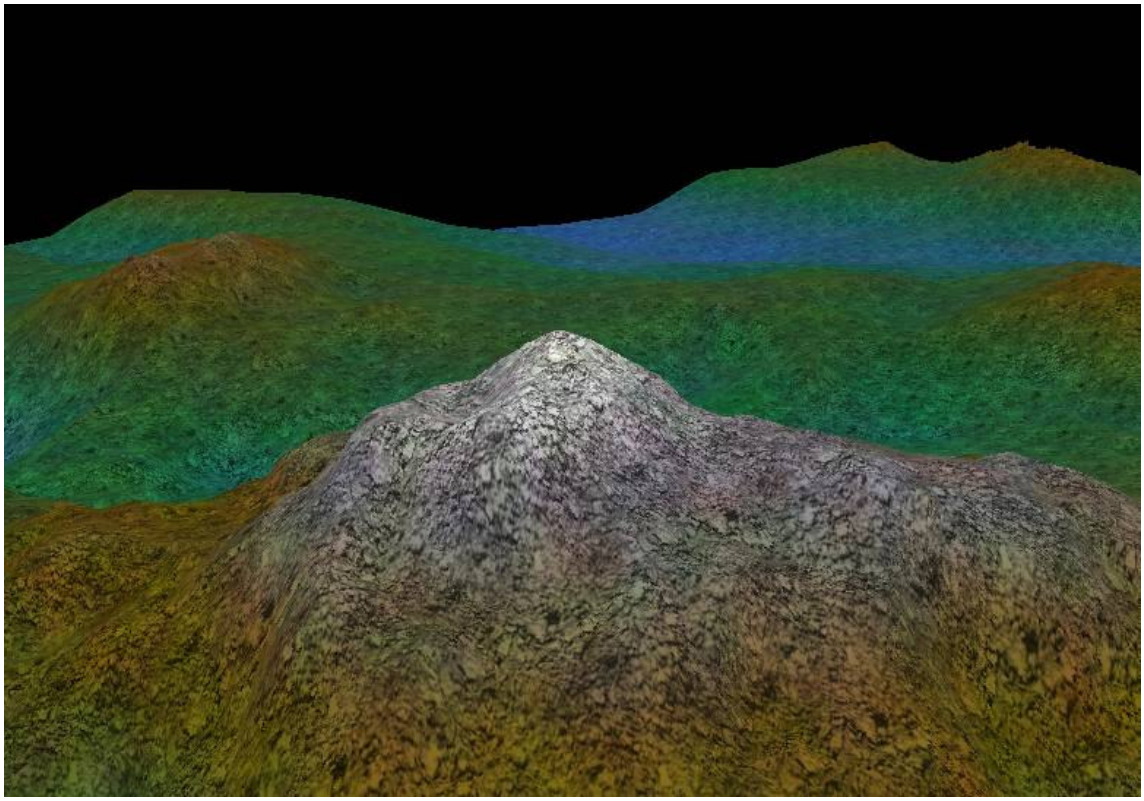
Obrázek 28: Počet snímků za sekundu při zobrazení drátového modelu výškové mapy NxN.

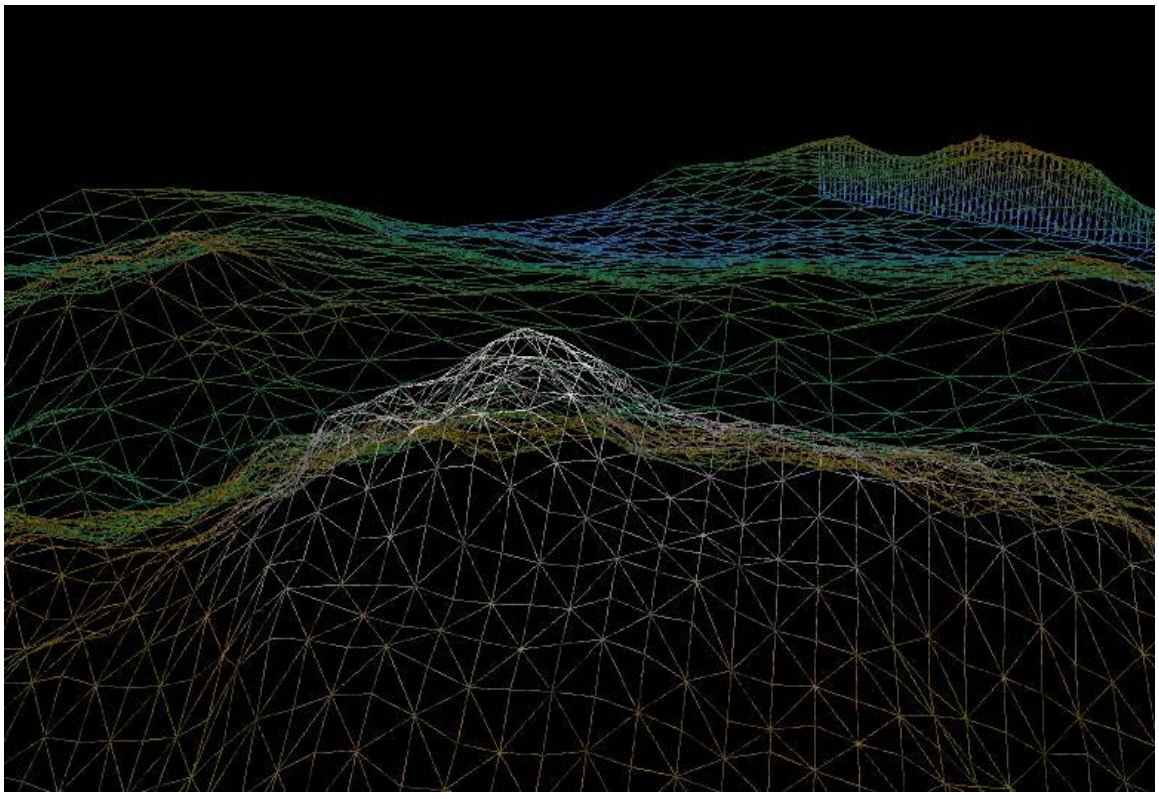
V příloze této práce je CD s implementovaným algoritmem a testovací aplikací. Společně jsou také přiloženy výškové mapy použité při testování.

8 Závěr

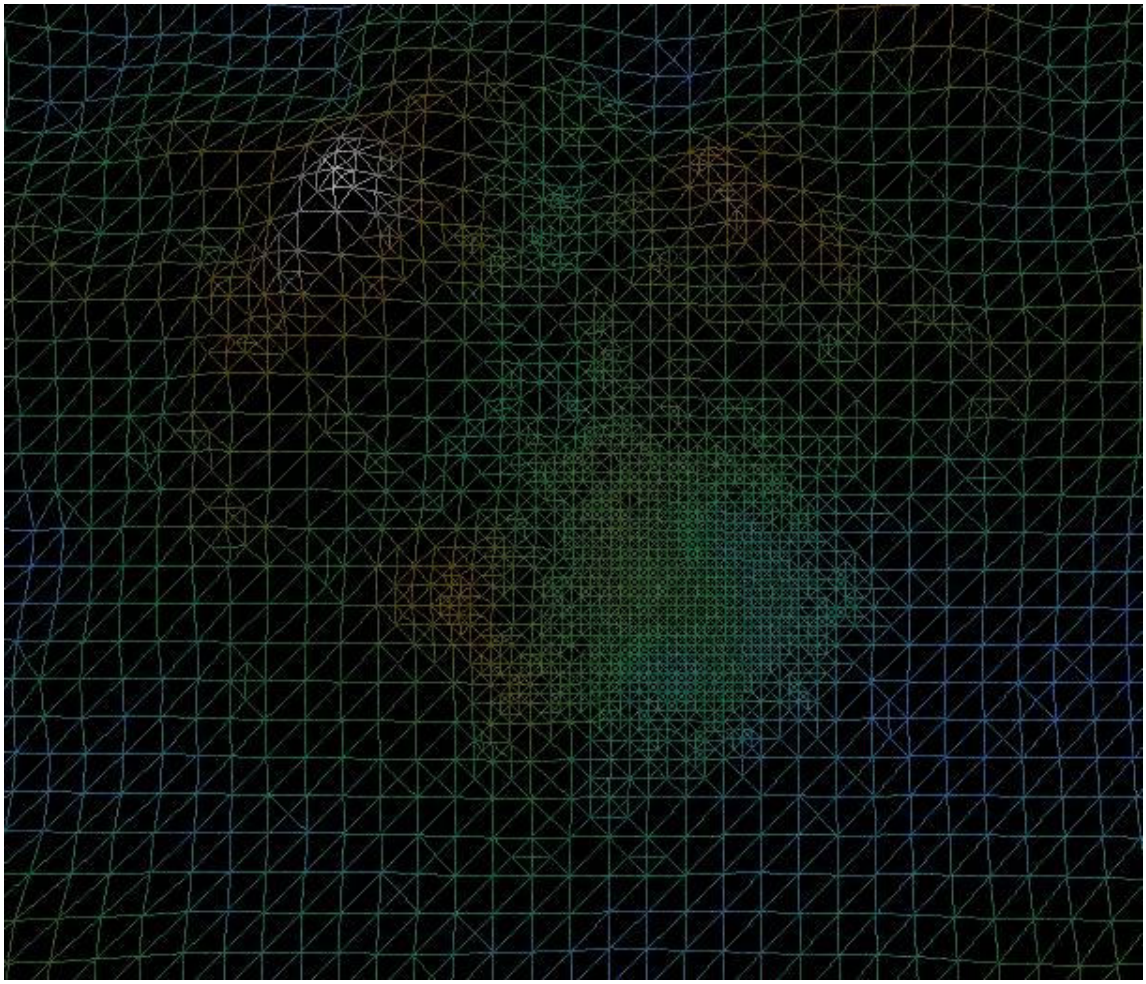
Efektivní algoritmy na zobrazování terénů je poměrně mladé téma. Stávající algoritmy nedokáží využít potenciál moderních grafických karet, proto lze očekávat v této oblasti dalšího vývoje. Díky rychlému růstu výkonných grafických akcelerátorů, není potřeba zabývat se příliš velkou optimalizací při zobrazování méně složitých scén. Avšak lze očekávat růst současného trendu, a tak se někdy v budoucnu může stát, že díky výkonnému hardwaru nebudeme potřebovat provádět žádnou extra optimalizaci při vykreslování terénů. Do té doby budou lidé stále přicházet s novými nápady a usnadňovat vývojářům život.

9 Galerie





Obrázek 29 a 30: Model terénu zobrazený implementovaným algoritmem.



Obrázek 31: Terén z ptačí perspektivy.

10 Použitá literatura

[1] M.Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, M. B. Mineev-Weinstein, ROAMing Terrain: Real-time Optimally adapting meshes, *Proc. Visulalization '97*, 1997

[2] Renato Pajarola, Large Scale Terrain Visualization Using The Restricted Quadtree Triangulation, *ETH Zürich*, 1998

[3] Stefan Rötger, Wolfgang Heidrich, Philipp Slusallek, Hans-Peter Seidel, Real-Time Generation of Continuous Levels of Detail for Height Fields, *Proc. WSCG '98*, 1998

[4] Joshua Levenberg, Fast View-Dependent Level-of-Detail Rendering Using Cached Geometry, *University of California at Berkeley*, 2002

[5] Zhao Youbing, Zhou Ji, Shi Jiaoying, Pan Zhigeng, A Fast Algorithm For Large Scale Terrain Walkthrough, *Zhejiang University*, 2001

[6] Peter Lindstrom, Valerio Pascucci, Visualization of Large Terrains Made Easy, *Center for Applied Scientific Computing Lawrence Livermore National Laboratory*, 2001

[7] Alex A. Pomeranz, ROAM Using Surface Triangle Clusters (RUSTiC), *Diplomová práce - University of Kalifornia at Daviss*, 1998

[8] H. Hoppe, Progressive meshes, *Proc. SIGGRAPH '96*, 1996

[9] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, Gregory A. Turner, Real-Time Continuous Level of Detail Rendering of Height Fields, *ACM SIGGRAPH '96*, 1996

[10] OpenGL Programming Guide (OpenGL Red Book), *Addison-Wesley Publishing Company*, 2002

[11] http://www.gamasutra.com/features/20000403/turner_01.htm (podzim 2003)

[12] http://www.vterrain.org (podzim 2003)

[13] http://www.gametutorials.com (podzim 2003)