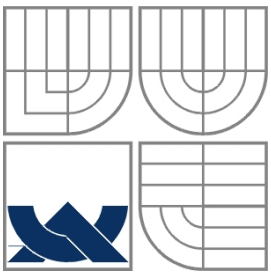
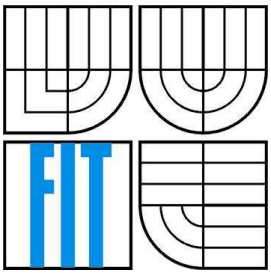


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

REALISTICKÉ ZOBRAZENÍ 3D KRAJINY

REALISTIC VISUALIZATION OF 3D LANDSCAPE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN WILCZÁK

VEDOUCÍ PRÁCE
SUPERVISOR

ING. TOMÁŠ MIKOLOV

BRNO 2009

- ZADÁNÍ BAKALÁŘSKÉ PRÁCE -

- VLOŽIT POUZE DO PEVNÉ VAZBY -

Abstrakt

Tato práce zkoumá a popisuje metody používané pro realistické zobrazení 3D krajiny v reálném čase. Krajinou je zde myšlen terén, obloha s různými atmosférickými jevy, vodní plochy a vegetace.
U každé dílčí části jsou rozebrány různé metody, jejichž výhody a nevýhody jsou poté brány na zřetel. Dále je popsán navržený program, který umožňuje zobrazování modelu terénu a oblohy, s použitím vybraných metod pro jednotlivé součásti. Pro zobrazování jsou použity knihovny SDL a OpenGL.

Abstract

This thesis analyses and describes different methods used for realistic visualization of 3D landscape in real-time. In this case, landscape means terrain, sky, atmosferic effects, water areas and vegetation. For each part there are several methods, whose pros and cons are considered in following work. Afterwards there is description of designed program, which allows visualization of terrain and sky using suitable methods. Libraries SDL and OpenGL are used for rendering.

Klíčová slova

Krajina, terén, obloha, vodní plochy, vegetace, 3D grafika, SDL, OpenGL.

Keywords

Landscape, terrain, sky, water areas, vegetation, 3D graphics, SDL, OpenGL.

Citace

Martin Wilczák: Realistické zobrazení 3D krajiny, bakalářská práce, Brno, FIT VUT v Brně, 2009

Realistické zobrazení 3D krajiny

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením ing. Tomáše Mikolova. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Martin Wilczák
18.5.2009

Obsah

1 Úvod

Každým rokem je na trh uvedeno mnoho nových počítačových her, které se velmi liší. Některé jsou z pohledu vlastních očí, u jiných se díváme hrdinovi zpoza zad. V jedné hře se ocitáme ve válečné vřavě a v druhé řešíme logické hádanky. V některých hrách bloudíme v místnostech, v dalších se volně procházíme po krajině.

Krajina je velmi komplexní, je zde mnoho různých částí a navíc tyto části jsou mnohdy jiné v různých koutech světa. Vysušená savana vypadá úplně jinak než kopce porostlé jehličnatými lesy. Jinak vypadá tropický ostrov a jinak severská tundra. Realistické zobrazení je tak složitým problémem. Proto je nutné některé oblasti zjednodušit a celý problém rozčlenit.

Výraznou oblastí je zobrazování terénu, kdy potřebujeme naznačit kopce a údolí, převisy či jeskyně. Dále je potřeba zabývat se zobrazením oblohy. Obloha není jen modrá, ale mění barvy podle polohy slunce. Také je třeba věnovat se počasí, od zobrazení samotných mraků po zobrazení deště. Neméně důležitá je v krajině i voda. V údolích se táhnou potoky a řeky, někde se vytvoří jezera, moře či oceány. Přičemž pokaždé vypadají tyto vodní plochy odlišně. Rovněž je vhodné prozkoumat zobrazení vegetace. Velké plochy jsou pokryté travinami a všude rostou různé keře a stromy.

Tato práce se zabývá některými z nastíněných problémů a jsou navrženy způsoby k řešení jejich zobrazování. Konkrétněji jsou popsány metody k zobrazování terénu a oblohy.

Spolu s teorií byl vyvinut vlastní program v jazyce C++, který využívá některých dostupných knihoven (SDL, OpenGL) a který umožňuje interaktivní zobrazení zkoumaných částí krajiny. Díky tomuto programu lze rozhodnout, které metody se pro zobrazování v reálném čase hodí více a které méně. Nejde jen o rychlost zobrazování, ale i o paměťovou náročnost, časovou náročnost v procesu předzpracování a především kvalitu výstupních snímků.

V závěru jsou shrnuty nejdůležitější poznatky a navržena další možná pokračování práce.

2 Používané metody

V této kapitole se zabýváme některými z existujících a více či méně používaných metod k zobrazování různých částí krajiny. Mnoho metod pro řešení různých aspektů terénu je uvedeno v [6].

2.1 Metody pro zobrazování terénu

Zobrazování terénu je velmi složitý problém. Terén totiž není jen rovina, jsou zde kopce a údolí, různé zlomy a převisy a toto všechno je potřeba postihnout. Musíme ale také celý terén vhodně nasvítit pro dosažení plastičnosti. S osvětlením se váže i problém stínů. K dosažení realistických výsledků je u terénu vhodné nejen stínit části samotného terénu, ale taktéž i jiných objektů, které se v krajině nacházejí. V různých nadmořských výškách je jiný povrch terénu a tedy i jeho zdánlivá barva.

2.1.1 Model terénu

Před zobrazováním terénu potřebujeme tento terén vhodným způsobem namodelovat. Vzhledem k tomu, že se zabýváme interaktivním zobrazováním na současném a běžně dostupném hardware, výsledné modely budou vždy brány jako množství bodů v prostoru a trojúhelníků z nich složených.

Ručně připravený model terénu

Jednou z možných variant zobrazování terénu je ručně připravený model. V takovém případě grafik ve vhodném 3D studiu terén vytváří podle předlohy ve formě náčrtků.

Hlavní výhodou je získání velmi kvalitního modelu, který odpovídá představám o výsledku. Díky přesnému vyjádření modelu je možné na terénu vytvářet jakékoli nerovnosti včetně jeskyní. Trojúhelníková síť je při pečlivé práci grafika optimálně adaptována. Grafik totiž ve 3D studiu okamžitě vidí, jak model vypadá a tak se snáze dosáhne dobrého grafického výsledku. Důležitá místa jsou detailnější než oblasti, které nejsou ve středu zájmu. S tím také souvisí dobrá paměťová náročnost. Při velmi dobrém vizuálním výstupu je dosaženo malých velikostí souboru pro popis modelu. Na výsledný model se následně namapují textury a model se uloží do vhodného formátu. Ten už se pouze načte a zobrazí.

Problém ale nastává při zobrazování větších terénů. Model je totiž statický a tak je ve velké dálce příliš velký detail. Na jeden zobrazovaný bod pak může odpovídat mnoho trojúhelníků a plýtvá se tak výkonem hardwaru. Tento problém se dá řešit pomocí Level Of Detail (dále jen LOD). V takovém případě dojde k ohodnocení významnosti vrcholů či trojúhelníků a ty nejméně důležité se pak nahrazují menším počtem vrcholů a trojúhelníků. Většinou se ohodnocuje chybou, která vznikne náhradou původního vrcholu, a vzdáleností od kamery. Jiným způsobem je rozdělení modelu na menší úseky a vytvořit několik úrovní detailu.

Dalším negativem této metody vytváření modelu je velká časová náročnost. Grafik musí respektovat návrhy designérů, ale zároveň je na něj kladen požadavek optimálního modelu. Většinou je tak nejdříve vytvořen velmi detailní model, který je přesným obrazem k původním návrhům, ale

zrovna tak je nešetrný k hardwaru a je také paměťově náročný. Pak přichází na řadu zjednodušování. To se může provádět buď automaticky nebo lépe opět ručně. Z ruční úpravy vyplývá, že model bude stále velmi kvalitní. Vzhledem k tomu, že v dnešních dobách je hardware velmi výkonný a zvládá zobrazit velké množství trojúhelníků, není v lidských silách ručně upravit celý model. Proto se využívá automatických úprav, které fungují na podobném principu jako LOD. Tyto automatické úpravy ale znehodnocují kvalitu modelu. Proto se kombinuje metoda ručních i automatických úprav. Dá se tak upravit i velmi rozsáhlý model a zároveň se ruční prací dosahuje kvalitních výsledků.

Kromě ručního vytváření modelu se dají použít plně automatické generátory. Tak jako grafik potřebuje nákresy ke své práci, i tyto generátory potřebují zdroj dat. Tím často bývá výšková mapa (Height-map). Jedná se o dvourozměrný obrázek, kde každý pixel znázorňuje výšku bodu terénu na daném místě. Generátor pak na základě této výškové mapy sestaví síť trojúhelníků.

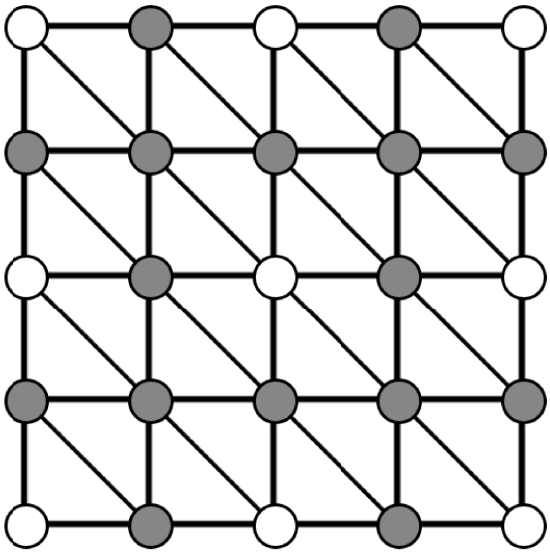
Existuje velké množství takových generátorů a každý používá jiné metody k vytvoření trojúhelníkové sítě. Liší se chybovými metrikami k ohodnocování částí terénu, způsobem členění modelu a samozřejmě také rychlostí a paměťovými nároky.

Model generovaný metodou Geo-MipMapping

Jednou z technik pro vytvoření modelu je Geo-MipMapping [1]. Princip je inspirován způsobem, jakým funguje mipmapping při nanášení textur na vykreslované modely.

Pro reprezentaci modelu je použito množství bloků, které jsou na sebe stranami napojené. Tyto bloky jsou ohodnoceny a podle tohoto ohodnocení jsou pak nahrazeny sítí trojúhelníků, jejichž počet odpovídá spočítanému ohodnocení.

Jeden blok je čtvercového tvaru o velikostí strany odpovídající 2^n. Tento blok je složen z trojúhelníků a vrcholů. Na každé straně je 2^n+1 vrcholů. Způsob sestavení trojúhelníků se může různit podle programu (např. Obrázek 2.1).



Obrázek 2.1: Blok užitý metodou Geo-MipMapping. Šedé vrcholy jsou u další úrovně vypuštěny.

Tento blok je nejvyšším možným stupněm detailu, který lze použít. Další stupně jsou vytvořeny vypouštěním některých vrcholů a tedy jiným sestavením trojúhelníků. Tento nižší stupeň

strukturou odpovídá bloku o poloviční rozměrech. Těchto stupňů nemůže být vytvořeno nekonečné množství. Počet závisí na velikosti základního stupně a rovná se tak číslu n.

Ohodnocení bloku se počítá na základě chyby, která vznikne když použijeme menší stupeň detailu. Tato chyba může být počítána různě, ale často se používá rozdíl výšky vypuštěného bodu a průměrné výšky dvou bodů, mezi kterými se vypuštěný bod nacházel. Tato chyba zůstává po celou dobu stejná.

Takto ale máme model upravený jen na základě různorodosti terénu v blocích, nezohledňujeme vzdálenost bloků od kamery. Proto při počítání vezmeme v úvahu vzdálenost mezi pozorovatelem a středy bloků. Tím sice ve vzdálených oblastech zanedbáme detail, ale tato ztráta není příliš znatelná a navíc drasticky snížíme výkonové a paměťové nároky.

Pokud chceme zabránit doskakování, které vzniká při změně vzdálenosti od kamery k bloku a tedy použitím vyššího či nižšího detailu bloku, pak je potřeba využít další analogie k mipmappingu. Tou je fakt, že další úroveň mipmappingu se použije až tehdy, když jednomu pixelu na obrazovce odpovídá jeden texel textury. V Geo-MipMappingu se tak chyba přepočítá na počet pixelů, na kterých by se chyba promítla, namísto statického rozhodování o změně detailu. Pro rozhodnutí, zdali použít jinou úroveň detailu pak stanovíme práh, při kterém ke změně dojde. Potom porovnáváme tento práh a počet pixelů, na kterých se promítne chyba.

Při každé změně pozice kamery musíme přepočítat tyto chyby a tím snadno dosáhneme výkonových limitů procesoru. Hodily by se nám tak určité optimalizace. Jednou z nich je, že budeme uvažovat kameru, jejíž pohledový vektor má jen horizontální charakter, a ta se tak zdánlivě otáčí pouze kolem svislé osy. Tím ovlivníme přepočet chyby na počet pixelů obrazovky, na kterých se chyba promítne. Zjednodušením tohoto výpočtu lze pak jeho větší část předzpracovat a procesor tak nemusí v každém snímku provést tolik instrukcí

Úskalím ale je, že blok, který je přímo pod kamerou, má zbytečně velký detail. Zjednodušením výpočtu se totiž chyba přímo pod pozorovatelem projeví stejně jako podobná chyba na horizontu. Fakt, že grafická karta je zatížená více než je nezbytné, není příliš důležitý, protože ztráta na jejím výkonu není tak významná jako zisk u procesoru.

Dalšího zvýšení výkonu lze dosáhnout použitím minimální vzdálenosti, při které na bloku přepneme na určitý detail. Tuto vzdálenost si uložíme pro každý detail, kterého lze na bloku dosáhnout. Při rozhodování pak již jenom porovnáváme skutečnou vzdálenost s těmito spočítanými vzdálenostmi. Tyto vzdálenosti lze navíc spočítat ve fázi předzpracování. Pro každou vzdálenost určité úrovně D_n počítáme podle následujících vztahů:

$$D_n = |\delta| \cdot C \tag{1}$$

$$C = \frac{A}{T} \tag{2}$$

$$A = \frac{n}{|t|} \tag{3}$$

$$T = \frac{2 \cdot \tau}{v_{res}} \tag{4}$$

δ je chyba, které se dopouštíme vynecháním detailů. n označuje near-clipping rovinu. t je vrchní souřadnice near-clipping roviny. τ je chyba přepočítaná na pixely. v_{res} je šířka obrazu.

Při počítání vzdálenosti od kamery potřebujeme spočítat druhou odmocninu. Pokud ale z D_n spočítáme druhou mocninu a u vzdálenosti odmocninu z výpočtu vypustíme, pak nám stačí porovnávat tyto mocniny. Touto úpravou se zbavíme časově náročné části výpočtu.

Při prostém použití trojúhelníkových sítí, které odpovídají stupni mipmappingu, vznikají díry na rozhraních mezi bloky s odlišným stupněm detailu. Tyto díry je nutné eliminovat. Lze použít několik způsobů.

Prvním je, že u bloků s nižším detailem přidáme vrcholy a trojúhelníky tak, aby navazovaly na nejvyšší možný stupeň detailu. Tím ale plýtváme pamětí, protože ukládáme spoustu vrcholů navíc.

Lepším způsobem je úprava bloků s vyšším detailem tak, aby navazovaly na sousední blok s nižším detailem. Mírně tak snížíme detail na okrajích tohoto bloku, ale neplýtváme pamětí. Podle toho, jaký je rozdíl mezi bloky, musíme vytvořit odpovídající trojúhelníky. Musíme proto udržovat reference mezi sousedy, abychom byli schopni zjistit jaký detail soused používá a podle toho na něj navázat.

Při použití dosud zmíněných technik získáme velmi kvalitní výsledky. Pokud se ale s kamerou posuneme, může dojít k významnému skoku. Důvodem je, že se najednou objeví či zmizí část vrcholů.

Tento problém můžeme vyřešit podobným způsobem, jakým funguje filtrování textur. Pokud použijeme mipmapping a žádné filtrování, můžeme pozorovat „švy“, které vznikají v místech, kde se použijí textury s jinou úrovní mipmappingu. Filtrování textur tyto švy odstraňuje tím, že postupně přechází z jedné textury na druhou jejich mícháním podle vzdálenosti od hranice, kde by mezi nimi vznikal skok. Podobného principu lze použít i u Geo MipMappingu.

U vrcholů, které by byly v nižším stupni detailu zanedbány, upravujeme jejich výšku. Tato výška se bude pohybovat mezi skutečnou výškou a výškou, která odpovídá průměru výšek sousedních vrcholů. Výška je ovlivněna vzdáleností od hranice mezi bloky s odlišnou úrovní detailu. Poměr, s jakým bude použita jedna nebo druhá výška, lze získat následujícím výpočtem:

$$t = \frac{d - D_n}{D_{n+1} - D_n} \tag{5}$$

d je vzdálenost od kamery k bloku. D_n a D_{n+1} jsou vzdálenosti kdy dojde k použití úrovně n (resp. $n+1$).

Pro výpočet nové pozice vrcholu pak použijeme následující vzorec:

$$v' = v - t \cdot \delta_v \tag{6}$$

Vygenerovat a zobrazit model je v tomto případě jednoduché. Pro bloky používáme struktury, které můžeme přímo použít k zobrazení. Lze použít i jiné struktury, ale pak musíme data z bloků vyextrahovat a vložit je do struktury připravené k zobrazování. V takovém případě lze eliminovat případné duplikáty vrcholů, které se objevují na hranicích bloků.

Hlavní výhodou metody Geo-MipMapping je přijatelné vytížení procesoru. Dobré je i zatížení grafické karty v poměru k procesoru. Procesor totiž zvládá plnit grafickou kartu dostatečným množstvím geometrie a získáváme velmi detailní model.

Nevýhodou je nižší přesnost modelu, protože používáme celé bloky a ne jednotlivé trojúhelníky. Tím se dopouštíme chyb v reprezentaci modelu a nedostáváme optimální model terénu.

Modelování metodou ROAM

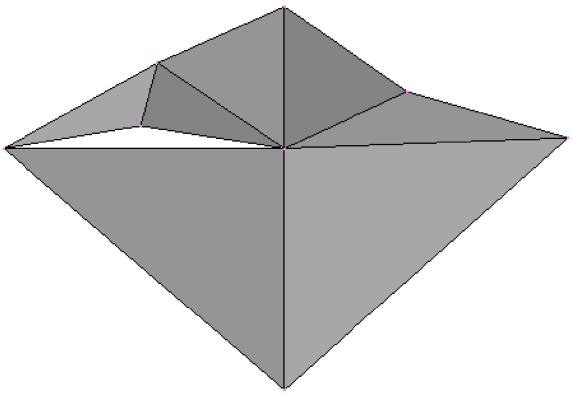
ROAM [2] je zkratka pro Realtime Optimally Adapting Meshes, což znamená modely optimálně se adaptující v reálném čase. Konkrétněji jde o to, že složitý model se podle určité metriky adaptuje na optimální model a to dostatečně rychle pro zobrazování v reálném čase.

Základním kamenem této metody je binární strom. Každý uzel tohoto stromu představuje pravoúhlý trojúhelník, který má obě odvěsny stejné. Pokud se dle vyhodnocení ukáže, že chyba v zobrazení je příliš velká, tento uzel se rozdělí na dva synovské uzly. Toto vyhodnocení a dělení se provádí do té doby, než je dosaženo cílových podmínek.

Podle chybové metriky se rozhoduje, zdali se má uzel stromu dále dělit či nikoliv. Bodem určujícím chybu je střed přepony, který je v případě rozdělení vrcholem s pravým úhlem u obou synovských trojúhelníků. Pro poskytnutí dobrých vlastností metriky je vhodné, aby byla monotónní. Tedy aby chyba uzlu byla větší nebo rovna chybám synovských uzlů.

Samotnou chybu je možné počítat různými způsoby. Často se ale chyba udává jako rozdíl výšky spočítané z výšek krajních bodů přepony a skutečné výšky, která je uložená ve výškové mapě. Tím model optimálně adaptujeme podle členitosti terénu. Často je ale potřeba model terénu detailnější blíže u kamery a ve velkých vzdálenostech model stačí jen velmi hrubý pro zachování siluety obzoru. Proto se do chybové metriky zahrnuje vzdálenost středu přepony od kamery. Tím dochází k posílení chyby uzlů v popředí a potlačení chyby uzlů v pozadí.

Metoda ROAM má zásadní nevýhodu – je počítána procesorem a pro velký terén je příliš časově náročná. K optimalizaci lze přistoupit několika způsoby. Co se chybové metriky týče, významným prvkem je vyloučení uzlů, které nebudou zobrazeny – nacházejí se mimo zorné pole kamery. Pokud je uzel mimo zobrazitelný prostor, není nutné jej dále dělit a proto mu chybu můžeme nastavit na minimum.

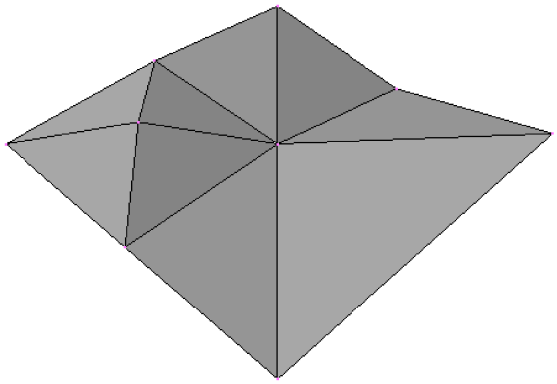


Obrázek 2.2: Díra vznikající u ROAM algoritmu bez Force-splittingu

Když jsou spočítané chyby, přichází na řadu samotné dělení uzlů. Dělit lze až do úplného konce, kdy už nemáme podrobnější data, ale pak dostaneme maximálně detailní model. Nebude to model optimální a určitě nebude vytvořen v přijatelném čase. Z toho vyplývá, že je nutné zvolit nějaký práh či hranici, při které už k dělení nedochází.

Tento práh je odlišný pro různé metriky a často i pro různé terény. Je tedy třeba jej experimentováním dobře určit, či jej odhadnout a v průběhu počítání pak dynamicky zvyšovat či snižovat podle vlastností výsledného modelu. Většinou se jako faktor ovlivňující práh dělení bere počet trojúhelníků výsledného modelu. Pokud je model příliš detailní, hranice se zvýší a v příštím „průchodu“ tak dojde k menšímu počtu dělení. Prakticky se tak hranice nastaví poměrně vysoká a pak se postupně snižuje. V opačném případě by na počátku mohl být terén příliš složitý a jeho tvorba by trvala velice dlouho.

Problémem tohoto jednoduchého dělení je vytváření „děr“ na rozhraní rozdělených a nerozdělených uzlů (viz Obrázek 2.2). Z tohoto důvodu musíme dělit i uzel sousedící přes přeponu (Obrázek 2.3). Tento soused může s přeponou děleného uzlu sdílet jakoukoliv svoji stranu, proto je potřeba mít u každého uzlu zaznamenány všechny sousedy a při dělení je upravovat podle změn ve výsledné trojúhelníkové síti. Tato metoda dělení souseda se nazývá Force-split (násilné rozdělení), protože takto můžeme rozdělit i uzel s chybou, která je natolik malá, že by k rozdělení uzlu bez použití Force-split nedošlo.



Obrázek 2.3: Zacelená díra při použití Force-splittingu

Nakonec musíme výsledný strom nějak vhodně vyjádřit do podoby vrcholů a trojúhelníků a zobrazit jej. Již od počátku je strom uvažován jako síť postupně dělených trojúhelníků. Toho se pro reprezentaci vhodně využívá.

Strom získaný dělením postupně procházíme a zobrazujeme listy tohoto stromu – to jsou dále nedělené trojúhelníky, které model reprezentují. Vše co je třeba provádět, je předávat trojúhelníkům souřadnice jejich bodů a postupně tyto trojúhelníky zobrazovat, či ukládat do jiných struktur, vhodných k zobrazení [4].

Prostým vytvářením trojúhelníků a bodů z předaných souřadnic ale dostáváme příliš mnoho redundantních informací. Jeden vrchol totiž může být sdílen několika trojúhelníky a zbytečně tak zatěžujeme sběrnici pro grafickou kartu či paměť.

Pro optimalizaci lze opět použít několik metod. V případě že model rovnou zobrazujeme, jsou trojúhelníky pro grafickou kartu reprezentovány jako posloupnost několika (i opakujících se) bodů. Moderní grafické karty disponují technologií zvanou vertex-cache, která umožňuje rychlé použití nedávno zadaných bodů. Při přímém zobrazování je tedy velice výhodné zadávat trojúhelníky ve vhodném pořadí.

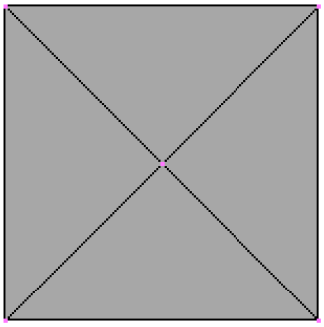
V případě, že chceme model vložit do paměti a vykreslit až později, musíme využít jiných způsobů, abychom šetřili paměť a zároveň nebylo složité model zobrazit. Toho lze dosáhnout

indexováním vrcholů. Vrcholy jsou (ideálně) zadány jen jednou a pro reprezentaci trojúhelníků jsou použity pouze jejich indexy.

Můžeme použít hrubou sílu a pro každý vrchol, který hodláme vytvořit, nejdříve zkontrolujeme, zdali už neexistuje. Podle toho se vrchol buď skutečně vytvoří nebo se použije index již dříve vytvořeného vrcholu.

Dále lze použít metodu, kdy synovským uzlům předáváme prázdné ukazatele na vrcholy a ty postupně naplňujeme. Naplněné ukazatele lze pak použít pro sousední uzel se stejným otcovským uzlem a nemusíme tak u těchto předaných bodů kontrolovat jejich existenci. Díky naplněným ukazatelům už víme, že bod existuje a přesně víme, který to je. Model je ale příliš složitý a nelze takto obsáhnout všechny možnosti sousedství, takže se kontrole existence vrcholu nevyhneme. „Nový“ bod se ale vytváří méně často, kontrola tedy nezdržuje tolik jako použití hrubé síly.

Metodu ROAM lze několika způsoby optimalizovat. První možností je zavést Split-queue. Jde o prioritní frontu, do které se zařazují veškeré trojúhelníky, které potřebujeme rozdělit. Prioritou je zde vypočtená chyba. Při inicializaci se do této fronty přidá kořen stromu. Poté se odebere první prvek fronty, rozdělí se a synovské uzly se podle priority vloží na odpovídající místo do fronty. Dělí se až do požadované hranice dělení, počtu trojúhelníků či určitého časového limitu. V dalším průchodu se vše zahodí a začíná se od začátku.



Obrázek 2.4: Diamant

Dalším způsobem je přidání druhé prioritní fronty, takzvané Merge-queue. Do ní se zařazují skupiny trojúhelníků, které tvoří diamant. Diamant je čtveřice trojúhelníků, kdy dvě dvojice trojúhelníků dají dohromady své otcovské trojúhelníky (viz Obrázek 2.4). Diamantů lze využít k rychlému spojení trojúhelníků a zjednodušení modelu. Při startu algoritmu se naplní Split-queue, Merge-queue zůstává prázdná. Když pak během dělení vznikne diamant, přidá se do Merge-queue. Pokud je nějaký diamant narušen dalším dělením některého z jeho trojúhelníků, pak se diamant zahodí. Když dojde k překročení počtu trojúhelníků nebo v případě že priorita spojování je větší než priorita dělení, vyjme se první diamant a dojde ke spojení. Takto se střídá dělení a spojování až do uplynutí času či při dosažení minimální trojúhelníkové sítě.

Při použití Split-queue je hlavní výhoda v tom, že máme plnou kontrolu nad počtem trojúhelníků a v optimálním pořadí dělíme otcovské trojúhelníky. Bez jejího použití totiž dělíme vše dokud nedosáhneme prahu dělení a pokud bychom omezili počet trojúhelníků, mohlo by se stát, že model bude nevhodně rozdělený.

Když použijeme Split-queue i Merge-queue, máme k dispozici velmi rychlý algoritmus. To z toho důvodu, že můžeme využít výsledku z minulého průchodu pro výpočet dalšího snímku.

Přepočítáme priority a trojúhelníky a diamanty podle priorit seřadíme. Pak spustíme algoritmus dělení a spojování. Výsledkem je, že se při pohybu kamery upravuje jen malá část trojúhelníků.

ROAM je velmi dobrý algoritmus pro optimální úpravu modelu. Při vhodném počtu trojúhelníků či prahu dělení dostaneme velmi kvalitní model, který při dobře zvolené chybové metrice neplýtvá pamětí pro vzdálené oblasti a v částech blízko kamery zachovává vysoký detail.

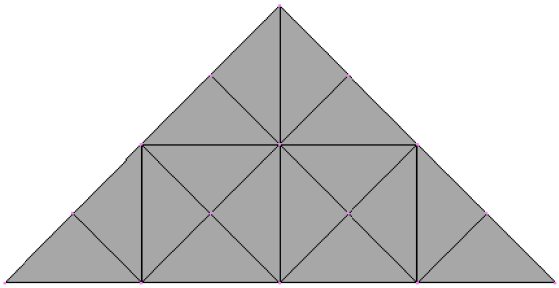
Velkou nevýhodou je ale časová náročnost algoritmu. Dnes jsou totiž grafické karty schopny zobrazit obrovské množství trojúhelníků a procesor tak není schopen dostatečně rychle připravit optimálně detailní model. Proto se dnes ROAM příliš často nepoužívá.

Metoda RUSTiC

Zkratka RUSTiC [5] znamená ROAM Using Surface Triangle Clusters. Jak již z názvu vyplývá, používá algoritmu ROAM. Ten je ale modifikován pro dosažení lepších vlastností. RUSTiC se snaží potlačit nevýhody originálního algoritmu a maximálně využít jeho výhod.

Hlavní nevýhodou ROAMu je jeho velká časová náročnost při tvorbě velkých a detailních modelů. RUSTiC tedy místo jednotlivých trojúhelníků zavádí celé shluky trojúhelníků. Samotný algoritmus se příliš nezměnil a počítání chyb a dělení je stejné jako u základního ROAM algoritmu. Počítání chyby a dělení se ale neprovádí do takové hloubky, protože nepotřebujeme výsledky pro trojúhelníky. Stačí nám chyby a dělení pro oblasti velikostí odpovídající velikosti shluku.

Listy stromu potom nereprezentujeme jednotlivými trojúhelníky, ale celými shluky trojúhelníků. Pro tyto shluky musí platit, že jakkoliv jsou poskládané vedle sebe, musí si odpovídat počty a pozice vrcholů stěn (viz Obrázek 2.5). V opačném případě by docházelo ke vzniku děr.



Obrázek 2.5: Blok trojúhelníků pro použití algoritmem RUSTiC. Počet vrcholů na stranách je stejný.

Na vzhledu shluku jinak nezáleží, takže můžeme použít jednoduchého rozdělení na menší trojúhelníky či použít nějaký algoritmus pro nalezení optimálního shluku. Tyto algoritmy ale nejsou triviální a proto se shluky tvoří ve fázi předzpracování nebo ještě před spouštěním programu.

RUSTiC tedy adaptuje i velmi složitý model při přijatelné časové náročnosti. Procesor tak dodává dostatečné množství dat pro grafickou kartu. Jedinou nevýhodou je ne zcela optimální model terénu.

2.1.2 Osvětlení terénu

Po vytvoření modelu terénu stojíme před dalším krokem. Tím je osvětlení terénu. To je velmi důležité, protože osvětlení a s tím spojené stínování dodávají modelu na plastičnosti.

Než ale můžeme řešit problém osvětlování, potřebujeme získat normálové vektory (dále jen normály) ve vrcholech modelu terénu. Ty jsou pro výpočet osvětlení nezbytné, protože podle úhlu, který svírá normála se směrovým vektorem osvětlení, se počítá stínování těchto vrcholů. V rámci tohoto problému máme dvě metody jak normály získat.

Normály počítané z vytvořeného modelu

V této metodě projdeme celý model a z jeho vrcholů a trojúhelníků spočítáme normálové vektory. Aby terén vypadal jako pozvolna se měnící plocha, je nutné pro každý vrchol vypočítat právě jeden normálový vektor.

V prvé řadě musíme spočítat normálové vektory pro trojúhelníky. Ze souřadnic jejich bodů dostaneme dva vektory (7) a jejich vektorovým součinem (8) získáme normálu trojúhelníku.

$$\vec{v}_i = B_i - A_i \tag{7}$$

$$\vec{u} \times \vec{v} = (u_2 \cdot v_3 - u_3 \cdot v_2 ; u_3 \cdot v_1 - u_1 \cdot v_3 ; u_1 \cdot v_2 - u_2 \cdot v_1) \tag{8}$$

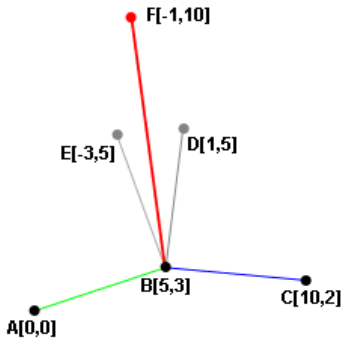
Poté potřebujeme pro každý vrchol sečíst normály všech trojúhelníků, do kterých vrchol patří. Tím získáme interpolovaný normálový vektor pro vrchol. Pro další práci potřebujeme normálové vektory znormalizovat. Normalizovaný vektor je takový vektor, jehož délka se rovná jedné. Z obecného vektoru dostaneme normalizovaný vektor podle následujícího vztahu:

$$\vec{n} = \frac{\vec{v}}{|\vec{v}|} \tag{9}$$

Normálová mapa

Normálová mapa je dvojrozměrné pole normálových vektorů pro každý bod modelu. Bodem může být i malá plocha trojúhelníku, ale v tomto případě máme na mysli vrchol modelu.

Normálovou mapu lze vytvořit softwarem třetí strany a pak už ji jen načíst. Je ale vhodnější si ji spočítat až těsně před zobrazováním, protože můžeme změnit výškovou mapu a aplikace sama se postará o aktuální normálové vektory. Pro výpočet potřebujeme výškovou mapu a můžeme využít dvou různých metod.



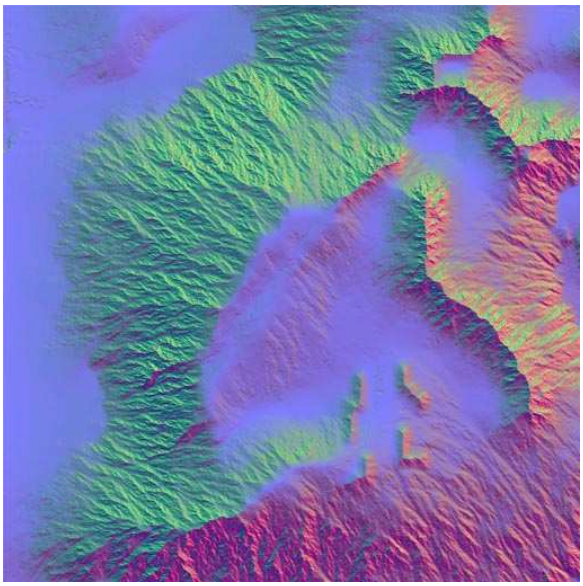
Obrázek 2.6: Výpočet normály ze souřadnic aktuálního a sousedních bodů

První metodou je dočasně si vytvořit trojúhelníky, které z vrcholu vycházejí. Výpočet je pak stejný jako při počítání normál z vytvořeného modelu. Tato metoda je ale pomalá, protože si musíme vytvořit několik trojúhelníků, spočítat jejich normálové vektory, ty pak sloučit a znormalizovat.

Víme, že Height-mapa je dvourozměrné pole výšek všech možných vrcholů. Toto pole tvoří pravidelnou mřížku. Pokud si z mřížky vykrojíme jeden řádek a z tohoto řádku pak jen sousední body vrcholu, pro který zjišťujeme normálu, můžeme snadno spočítat vektor, který by odpovídal normále zobrazené zboku. Pokud uvažujeme, že ve výškové mapě osa X směřuje doprava, osa Y vystupuje směrem k nám a osa Z směřuje dolů, pak při vykrojení požadovaného vrcholu a jeho sousedů a zobrazení zboku zanedbáváme vychýlení normály po ose Z (Obrázek 2.6).

Takto lze tedy získat vychýlení po ose X a Y. Prostou analogií na vykrojení ze sloupečku dostaneme vychýlení po ose Z a Y. Takto pak složíme celý normálový vektor. Poté lze ještě upravit vychýlení po ose Y, čímž uměle upravujeme plasticitu modelu. Výsledný vektor už pouze znormalizujeme. Tato metoda je mnohem výhodnější, protože pouze sčítáme a odčítáme celá čísla.

Spočítané normály nakonec uložíme do vhodné struktury. Pokud bychom ukládali každou normálu jako jeden pixel složený ze tří bytů (v každém bytu jedna souřadnice), pak bychom dostali obrázek podobný Obrázku 2.7.



Obrázek 2.7: Normálová mapa se souřadnicemi vektorů zobrazenými do RGB obrázku

Hardwarové osvětlení

Po spočítání normál je již vše připraveno pro osvětlování. Při použití hardwarového osvětlení využíváme rychlosti grafických karet. Ty jsou dnes dostatečně výkonné pro výpočet osvětlení několika světelnými zdroji najednou. V této metodě si pro osvětlení terénu vyhradíme jeden zdroj pro simulaci slunečního osvětlení. Nastavíme parametry světla, zapneme osvětlování a necháme vykreslit model. Grafická karta se o vše postará.

Tato jednoduchost je ale vykoupena náročností metody. Dnešní hardware je sice poměrně výkonný, není ale důvod výkonem plýtvat. V případě složité a rozsáhlé scény potřebujeme každé procento výkonu využít rozumně. A zde se osvětlení počítá v každém zobrazeném snímku znovu a znovu, přestože se parametry světelného zdroje nemění.

Použití Light-mapy

Proto je výhodnější osvětlení předpočítat a získat takzvanou Light-mapu – mapa obsahující informace o osvětlení vrcholů terénu. Osvětlení počítáme podobným způsobem jako grafická karta. Pro každý možný vrchol známe jeho normálový vektor a známe parametry světla. Pro výpočet použijeme vzorec pro Phongův osvětlovací model.

$$I_V = I_A + I_D + I_S \tag{10}$$

$$I_A = I_a \cdot r_A \tag{11}$$

$$I_D = I_L \cdot r_D \cdot (\vec{N} \cdot \vec{L}) \tag{12}$$

$$I_S = I_L \cdot r_S \cdot (\vec{V} \cdot \vec{R})^{n_S} \tag{13}$$

$$R = 2 \cdot (\vec{N} \cdot \vec{L}) \cdot \vec{N} - \vec{L} \tag{14}$$

I_A, I_D a I_S představují ambientní, difúzní a spekulární složku barvy. I_V je výsledná barva. I_L je intenzita světla dopadajícího na povrch. I_a je intenzita ambientního světla. r_A, r_D a r_S jsou vlastnosti povrchu ovlivňující příslušné složky. $\vec{N}$, $\vec{L}$, $\vec{V}$ a $\vec{R}$ jsou vektory – normálový vektor povrchu, směrový vektor osvětlení, vektor odrazu a vektor směřující na pozici kamery. Nakonec n_S je vlastnost povrchu ovlivňující ostrost odrazivosti.

Spočítanou Light-mapu poté použijeme jako barevnou texturu. Při správném použití je rozdíl mezi hardwarovým osvětlením a použitím Light-mapy jen těžko pozorovatelný. Při použití Light-mapy tak dostáváme velmi dobré výsledky a zároveň si šetříme výkon pro jiné části krajiny či grafické efekty.

2.1.3 Stíny vrhané terénem

Prvkem, který velmi přidává na realističnosti, jsou vrhané stíny. Stín nám totiž dodává velmi jasnou představu o tvaru pozorovaných objektů a jejich vzájemné poloze. U terénu je tak dobré uvažovat stíny tvořené za osvětlenými kopci.

Pro získání stínů existuje mnoho metod, které lze akcelerovat grafickou kartou. Metody se liší rychlostí, přesností stínů, složitostí i možnostmi použití. Zde prozkoumáme použití Shadow-mapy.

Shadow-mapa

Tato metoda pracuje za využití Depth-bufferu. Podrobně je popsána například v [8]. Princip metody je takový, že se kamera nastaví podle pozice a natočení světelného zdroje, jehož vrhané stíny chceme postihnout. Poté se vykreslí všechny objekty scény, které mají vrhat stíny. Vše, co potřebujeme z vykreslené scény, je Depth-buffer. Proto není nutné počítat jakékoliv stínování či texturování.

Po vykreslení a získání Depth-bufferu se normálním způsobem vykreslí celá scéna. Osvětlení scény se v tomto průchodu nastaví tak, aby scéna vypadala jako zastíněná.

Poté musíme osvětlit části, které nejsou zastíněné. V této fázi využijeme získaný Depth-buffer. Při vykreslování scény se testují všechny pixely na vzdálenost od světla – tedy hloubku

v Depth-bufferu. Pokud je pixel blíže světlu než jaká je hodnota uložená v Depth-bufferu, pak je pixel osvětlený a vykreslí se. V opačném případě je pixel zastíněn a nekreslí se.

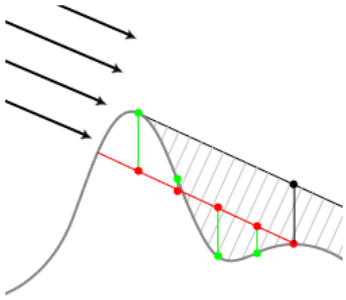
Tato i jiné metody stínění jsou relativně přesné. Díky akceleraci grafickou kartou jsou i rychle proveditelné. Nevýhodou je ale nutnost několikerého vykreslování. Ač jsou metody akcelerované grafickým adaptérem, pro složité scény jsou nutná nějaká zjednodušení. Často se tak vypouští některé objekty, které pak nevrhají stíny.

Jak již bylo naznačeno v kapitole osvětlení, u zobrazení krajiny je naším zdrojem světla slunce. To ale nemění svou pozici nebo ji mění jen zřídka. Tak jako u osvětlení by počítání stínění v každém snímku bylo plýtváním výkonu.

ShadowVolume-mapa

Proto můžeme přistoupit k nějaké metodě, která nepatří mezi nejrychlejší, ale pro zobrazení stínů krajiny je dostatečná. Možností je opět více a některé jsou popsané v [6].

Ze směrového vektoru slunce a výškové mapy lze spočítat, které vrcholy terénu budou zastíněné. Pro každý pixel z výškové mapy získáme několik vzorků. Vzorek získáme tak, že se posunujeme ve směru ke světelnému zdroji. Po každém posunutí načteme údaje z výškové mapy a tuto výšku upravíme podle směrového vektoru světla. Vzorek porovnáme s výškou zkoumaného pixelu. Pokud je výška vzorku vyšší než výška zkoumaného pixelu, je tento pixel ve stínu kopce, který je tvořen vzorkem. Tento princip zobrazuje Obrázek 2.8.



Obrázek 2.8: Stínění terénu

Stínících vzorků můžeme získat různé množství, s větším počtem vzorků se zvyšuje přesnost metody. Také se můžeme po výškové mapě posouvat o větší vzdálenosti a tím pak získáme lepší „dosah“ metody. Zároveň ale snížíme přesnost, protože vynecháme některé pixely z výškové mapy.

Výslednou bitovou mapu následně použijeme při počítání osvětlení terénu. Pokud je vrchol zastíněn, nepočítáme pro tento vrchol osvětlení sluncem.

Problémem metody ale je, že získáváme pouze představu o zastínění samotného terénu. Výsledkem je totiž pouze bitová mapa, která obsahuje informace o tom, zdali je potenciální vrchol zastíněn či nikoliv. Vzhledem k tomu, že můžeme potřebovat stínit i jiné objekty scény (například stromy), musíme metodu trochu vylepšit.

Vhodným vylepšením je uložit si výšku stínového „tělesa“ terénu. Tuto výšku lehce získáme ze vzorků počítaných u předchozí metody. Pro každý pixel máme vzorků několik a jako výslednou hodnotu bereme maximální výšku stínu.

Rozhodnutí o zastínění pak řešíme při počítání osvětlení. Pokud je výška pixelu větší než výška stínu, spočítáme osvětlení sluncem. V opačném případě osvětlení nepočítáme.

Výhodou této metody je, že se můžeme snadno rozhodovat, zdali jsou zastíněné i ostatní objekty. Pokud je objekt pod výškou stínu, je zřejmé, že je stíněný tak jako terén.

Tato metoda je při vhodném nastavení dostatečně přesná. Lze ji tedy pro slunce jako zdroje světla použít a výkon ušetřený na grafické kartě využít lépe.

2.1.4 Barvení terénu

Dosud jsme se zabývali vytvářením modelu a jeho vhodným osvětlením a stíněním. Terén by ale byl jen jednobarevná plocha, pokud bychom se nezabývali obarvením modelu. Toho lze opět dosáhnout několika způsoby.

Nezáleží, jak získáme barvy k pokrytí terénu. Můžeme je načíst z obrázku či je třeba spočítat podle výšky vrcholu. Jednu či kombinaci těchto variant lze použít pro jakoukoliv metodu obarvování.

Jedním způsobem obarvení je nastavit každému vrcholu modelu terénu jeho barvu a nechat na grafické kartě, aby se při vykreslování postarala o správné barevné přechody. Nevýhodou je velmi nízký detail terénu.

Dále je možné použít k obarvení texturu. Pokud použijeme texturu o nízkém rozlišení a natáhneme ji přes celý terén, nedostaneme o nic lepší výsledek než je barvení vrcholů. Pokud ale použijeme texturu s vysokým rozlišením, získáme terén s velmi pěknými barvami i detailem. Zároveň ale vznikají enormní nároky na paměť. Současně klesá rychlost vykreslování.

Využijeme tak schopností moderních grafických karet, které dovedou na jeden model nanést několik textur zároveň. Pro základní obarvení použijeme texturu s nízkým rozlišením. Získáme tak hrubé obarvení terénu, které respektuje oblast, jež se snažíme zobrazovat. Odstíny zelené pokrývají zatravněné plochy, šedivé barvy mohou reprezentovat skály a podobně.

Jako druhou texturu použijeme obrázek zachycující určitý detail. Například zrnka písku či hroudy hlíny. Tuto texturu pak nanášíme na model přes první texturu a mícháním vzniká kombinace nanesených barev. Máme tak terén obarvený podle větších oblastí a zároveň jsme získali detail, který dodá na realističnosti.

Tuto metodu lze obohatit použitím více textur pro zobrazení detailu (Texture Splatting) [7]. Pro jejich správné rozložení po terénu použijeme masku, která určuje, ve kterých oblastech se použije konkrétní textura detailu. Můžeme tak zobrazit množství variací terénu v jednom modelu.

2.2 Metody pro zobrazování oblohy a slunce

Dalším aspektem realistického zobrazení je obloha a s ní spojené zobrazení slunce či měsíce. Zároveň sem patří další atmosferické efekty, mezi něž patří hlavně mlha, která stírá rozdíly mezi oblohou a vzdálenými oblastmi terénu.

Bez oblohy bychom viděli pouze terén a za ním výplňovou barvu. Tu sice můžeme nastavit na odstín modré, ale tato barva rovnoměrně pokrývá veškeré oblasti nezakryté modelem terénu, což nevypadá příliš realisticky. Zároveň lze vyloučit i složité počítání osvětlení atmosféry, protože chceme zobrazovat oblohu pouze pro malý terén, ne celou planetu.

Potřebujeme tedy něco, co by poskytlo vhodné obarvení. Můžeme použít například Skybox, což je krychle, která obaluje celý terén. Tuto krychli můžeme obarvit na úrovni vrcholů, či lépe, potáhneme ji texturou oblohy.

Obě varianty ale mají svá úskalí. Pokud obarvíme vrcholy Skyboxu, dostáváme velmi málo rozdílů. I když využijeme barevných přechodů mezi různě obarvenými vrcholy, máme velmi malou kontrolu nad celkovým vzhledem oblohy.

V případě nanášení textury oblohy dostaneme velmi dobrý grafický výstup. Tuto metodu lze dokonce použít k zobrazení mraků či slunce. Pokud ale vyžadujeme změny v závislosti na části dne, stojíme před velmi složitým problémem. Nemůžeme totiž nijak snadno a hlavně rychle přepočítávat obarvení této textury. O to hůře, pokud by tato textura sloužila i k zobrazení mraků.

Pro obě metody ale platí, že různé body ve stěnách krychle mají různé vzdálenosti od pozorovatele. Dochází pak k nepříjemným rozdílům v množství mlhy na různých místech terénu.

Kvůli uvedeným problémům není Skybox zrovna nejvhodnější k použití pro zobrazování oblohy. Pro dosažení lepších výsledků lze použít kouli namísto krychle. Přestože jde opět o reprezentaci pomocí trojúhelníků a nejedná se tedy o ideální kouli, rozdíly mezi vzdálenostmi různých bodů ve stěnách jsou mnohem menší než u krychle. Díky tomu se vyhneme problému, který vzniká při zamlžování vzdálených částí terénu.

Možnosti obarvení modelu oblohy jsou stejné jako u Skyboxu. Obarvování vrcholů má ale významnou výhodu. Zvýšením detailu modelu koule získáváme větší kontrolu nad obarvením a zároveň se méně projevuje problém se zamlžováním. Potažením koule texturou ale vznikají naprosto stejné problémy jako u Skyboxu. Stejné jsou ale i klady a proto velmi záleží na našich požadavcích.

Tak jako u oblohy i u zobrazování slunce se nevyplatí snažit se zobrazovat jej jako plynnou kouli a terénem rotovat kolem něj. Místo toho terén stojí na místě a hýbeme sluncem. Slunce pak lze zobrazovat jako kouli, kruh či jednoduchý polygon. Dokonce jej lze i zobrazit na textuře použité pro obarvení oblohy. Při zobrazování koulí či kruhem pouze nastavíme barvu těmto modelům. Pokud zobrazujeme slunce jako polygon, potřebujeme nějak získat iluzi kruhu. Toho lze dosáhnout použitím textury, kterou na polygon naneseme a na vhodných místech nastavíme průhlednost.

Mezi těmito metodami nejsou velké rozdíly, je ale nutné brát v úvahu, že nemáme neomezený výkon a proto nelze použít složité modely, u kterých již nelze pozorovat, že se jedná o mnohoúhelník. Taktéž není nutné používat model koule, protože stačí pouze iluze disku. Toho lze snáze dosáhnout s pouhým kruhem či otexturovaným polygonem.

Pokud zanedbáme fakt, že slunce během dne mění svou pozici, značně si ulehčíme práci. Není totiž nutné přepočítávat obarvení oblohy. Stejné také zůstává zabarvení terénu vzniklé osvětlením a vrhané stíny. Pro tuto variantu se pak hodí zobrazovat slunce na textuře natažené na model oblohy.

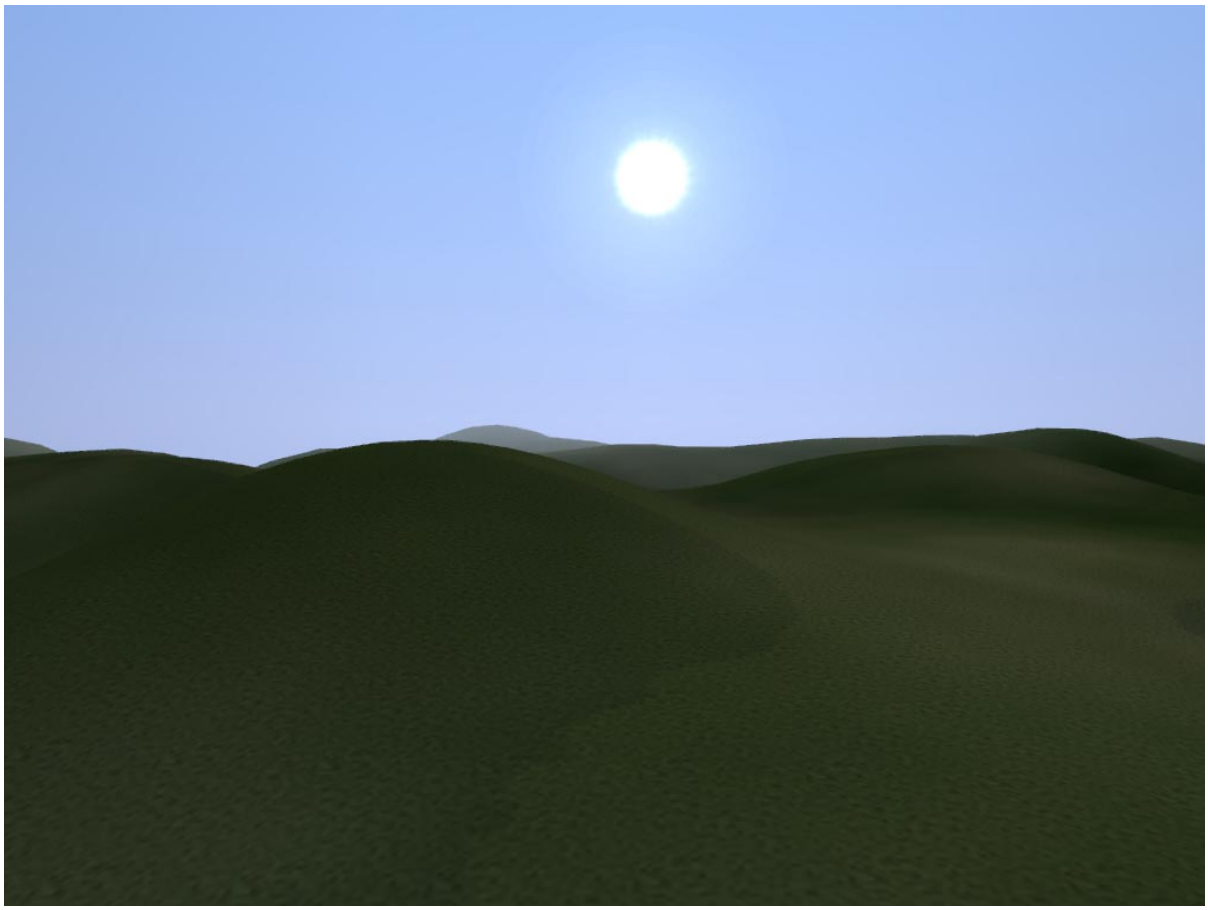
Pokud se rozhodneme pro střídání dne a noci, vyvstává celá řada problémů, které musíme vyřešit. Velmi záleží na tom, jak často chceme tuto změnu aplikovat. Protože jsme již vyloučili složité metody pro výpočet zbarvení atmosféry kolem celé planety, potřebujeme při každé této změně určit novou barvu slunce a jeho polohu.

Co se týče terénu, je nutné při každé změně přepočítat zastínění a osvětlení, protože se změnou barvy a polohy slunce se mění i zabarvení terénu. Dalším aspektem je změna obarvení oblohy. V noci je černá, popřípadě má velmi tmavý odstín modré. Naproti tomu v poledne je světle modrá a během večera dostává fialovou až růžovou barvu. Tyto barvy jsou navíc jiné v různých částech oblohy.

Denní cyklus velmi přidává na realističnosti. Je ale třeba mít na paměti, že je nutné při každé změně polohy slunce přepočítat spoustu dat a to nás stojí výkon. Tyto výpočty tedy musí být dostatečně rychlé, aby při přechodu nedocházelo k příliš významným poklesům počtu zobrazených snímků za sekundu.

3 Realizace vlastního řešení

V této kapitole je popsána vlastní realizace – cíle našeho řešení a metody implementované v navrženém programu. Screenshot výsledné aplikace je na Obrázku 3.1.



Obrázek 3.1: Screenshot výsledné aplikace

3.1 Cíle a návrh našeho řešení

Naším hlavním cílem bylo zobrazení jednoduchého modelu krajiny. Tento cíl se ale časem rozrostl o několik dalších částí. Nakonec bylo cílem navrhnout vhodný způsob pro zobrazování modelu terénu, počítání osvětlení a stínění a také zobrazení oblohy a slunce.

Rozhodli jsme se zobrazovat krajinu o omezené velikosti, nebylo tedy třeba zkoumat problémy spojené s efektivním používáním rozsáhlých textur. Požadavkem ale bylo implementovat zjednodušování modelu terénu alespoň na základě vzdálenosti.

Dále jsme se rozhodli pro slunce, které se nepohybuje po obloze. Algoritmy pro výpočet osvětlení a zastínění tak nemusí být extrémně rychlé.

3.2 Implementace vlastního řešení

Pro pouhé splnění našeho cíle stačilo navrhnout relativně jednoduchý program, který by zvládl vygenerovat modely terénu a oblohy, spočítat pro ně důležité součásti a načíst textury.

Nakonec jsem ale navrhl robustnější řešení, které lze snadno upravovat a rozšiřovat o další vlastnosti. Vzhledem k rozsáhlosti celkového řešení není detailně popsán celý program, ale pouze části, které úzce souvisí se splněním stanovených cílů.

Navržený program je jakýmsi enginem, který by se dal použít i pro vytvoření počítačové hry. Jde o rozpracovaný projekt, takže některé části jsou implementovány jen velmi zběžně či vůbec. Celý program je napsán v jazyce C++ a je použito objektové paradigma.

Hlavní třídou programu je třída `wEngine`, která reprezentuje bežící engine. Tato třída využívá dílčích částí, které se starají o důležité aspekty v počítačových hrách. Jsou to grafika (`wGraphics`), fyzika (`wPhysics`) a logika hry (`wLogic`). Je zde ale ještě množství dalších částí.

Třída `wGraphics` se stará o různé funkce programu, které jsou nezbytné pro vykreslování. Jedná se tak o seřazení objektů podle vzdálenosti od pozorovatele, sestavení oktalového stromu, ořezání pohledovým tělesem a samotné vykreslení objektů. Třída `wPhysics` slouží k počítání různých fyzikálních vlastností objektů. Poslední třídou je `wLogic`, která má za úlohu počítat chování umělé inteligence a další části, které zatím nejsou uvažovány.

Do výčtu dalších potřebných tříd patří `wSettings`, která ze souboru načítá informace o zvoleném detailu oblohy, omezení počtu snímků za sekundu apod. Tato nastavení jsou vyžadována pro nejrůznější části programu a jsou proto distribuována dále.

Další důležitou částí je třída `wEngineScene`, která si uchovává informace o načtené scéně. Program si udržuje seznam těchto scén a lze tak mezi nimi rychle přepínat. Každá tato scéna si uchovává seznam všech objektů, které se v ní nachází. Dále se uchovává informace o virtuálním čase pomocí třídy `wGameTime` a také o událostech, ke kterým v této scéně dochází. Tyto události má na starosti objekt třídy `wEventManager`.

Každý objekt scény je reprezentován instancí třídy `wEngineObj`, který se skládá z částí, určených k manipulaci ze tříd pro grafiku, logiku a fyziku. Tyto části objektu jsou tedy instancemi tříd `wPhysicsObj`, `wGraphicsObj` a `wLogicObj`. Tyto třídy nejsou využívány přímo, ale jsou použity k dědění a přetěžují se některé z jejich metod.

Některé části programu potřebují přístup k různým datům, zde jsou to textury a modely. Zároveň je ale vhodné, aby se neplýtvalo pamětí – je tedy nutné zajistit, aby model, který je v jedné či více scénách, nebyl v paměti načten vícekrát. Proto existuje třída `wResManager`, která spravuje tyto datové zdroje a zpřístupňuje data dalším částem programu.

Tento správce zdrojů na požádání z jiných částí programu načítá modely a textury. Tímto načítáním ale není jen načtení dat ze souboru, ale i generování těch dat, které v souborech uloženy nejsou. Příkladem budiž například Light-mapa, která se generuje z normálové mapy a výškové mapy.

Aby se předešlo opakovanému načítání či generování stejného modelu, musí být u každého objektu zdroje nějaký identifikátor. Pro účely programu byly zavedeny identifikátory dva, protože můžeme požadovat ve dvou různých scénách stejný terén, ale pokaždé z jiné oblasti výškové mapy. Je tak potřeba základní identifikátor a potom dodatečná varianta, která postihuje různé odchylky.

Správce zdrojů si udržuje přehled o načtených objektech a pokud si jiná část zažádá o již načtený objekt, správce mu tento ihned zpřístupní. Je ale nutné zahazovat nepoužívané objekty,

abychom uvolnili jimi alokovanou paměť. U každého objektu zdroje dat je tak uchováván počet referencí a správce pak objekty s nulovým počtem referencí zahazuje.

Všechny tyto objekty jsou vytvářeny správcem zdrojů, ale samotná data do nich naplní řada loaderů a generátorů. Tyto jsou v područí správce zdrojů, který podle informací o objektu zvolí příslušný zdroj dat a na odpovídajícímu loaderu nechá načtení či vytvoření dat. Právě tyto loadery budou hlavním předmětem zájmu, protože v nich jsou implementovány způsoby pro přípravu modelů krajiny.

Informace o všech objektech jsou zatím pevně nastavené v programu a časem by měly být načítány ze souborů. Tyto informace slouží k nastavení různých fyzikálních či logických limitů, typu grafického objektu, či také ke zvolení souborů, z nichž se budou načítat data. K úschově a distribuci těchto informací za běhu programu slouží řada interních „databází“.

3.2.1 Běh programu

Na začátku programu se vytvoří objekt třídy `wEngine`. Ten si vytvoří instanci `wSettings`, která ze souboru načte nastavení, dále pak instance tříd `wGraphics`, `wPhysics` a `wLogic`. Dále ještě databázi objektů enginu a instanci třídy `wResManager`. Při vytváření objektů tříd `wGraphics`, `wPhysics`, `wLogic` a `wResManager`, si každý tento objekt vytvoří svou interní databázi. Poté jsou načteny scény.

Načítání scény spočívá v načítání objektů. Scéna tak zažádá o nějaký objekt a engine pak distribuuje tento požadavek na dílčí části. Tedy část pro grafiku, fyziku a logiku. V tomto momentě dochází k načítání či generování modelů a textur. Z vytvořených částí je pak sestaven celý objekt a ten je předán scéně. Scén může být načteno několik a engine si uchovává, která scéna je aktivní. K přepnutí scény pak dochází na základě událostí.

Nakonec se program „rozběhne“. Jde o sérii volání funkcí a metod, která se opakuje v každém snímku a zajišťuje aktualizaci dat a zobrazování. Načítají a rozpoznávají se uživatelské vstupy, dále se počítá logika hry, pohyb a změny rychlostí objektů a na závěr se vše zobrazí. Poté se pro další snímek zpracují události, které vznikly během předchozích činností.

Před ukončením programu se zruší všechny do té doby vytvořené či načtené objekty.

3.2.2 Zobrazování terénu

Pro zobrazování terénu je použito několik loaderů, protože jeho generování je v programu rozděleno do několika částí.

Model – metoda ROAM

V prvním návrhu bylo počítáno pouze s jednou – vlastní – metodou. Po konzultacích jsme se ale rozhodli najít a upravit nějakou stávající metodu. Nakonec byl implementován loader s ROAM algoritmem. Pro tyto účely jsem nalezl implementaci Bryana Turnera [3]. Tu jsem prostudoval a přepracoval pro použití v našem programu. Některé části ale zůstaly beze změny.

Prvním loaderem je tedy `wLoaderTerrainC`. Tento implementuje ROAM algoritmus bez jakýchkoliv optimalizací a proto není tak rychlý jak by mohl teoreticky být. Oproti nalezené verzi nezjednodušuje počítání priorit a je tak přesnější.

Třída `wLoaderTerrainC` obsahuje několik metod, důležitá je ale především metoda `Load(...)`. Ta podle zadaných parametrů zajistí vytvoření modelu terénu a jeho nahrání do předané struktury. Vzhledem k návrhu programu si loader musí uchovávat informace ke každému terénu, protože například binární strom trojúhelníků nelze ukládat do datové struktury pro reprezentaci modelu – vykreslovací část enginu totiž binární strom k ničemu nepotřebuje. Proto je definována třída `wContTerrain` a loader obsahuje seznam instancí této třídy. Podle potřeby pak upravuje konkrétní terén.

Generování je přeneseno na nižší úroveň. Samotný loader pouze volá metody `CalculatePriority(...)` pro přepočítání priorit, dále pak `AdaptModel(...)` pro vytvoření a rozvětvení binárního stromu a nakonec je volána metoda `LoadModel(...)`, která z dosud vytvořených informací generuje vrcholy a trojúhelníky.

Ani třída `wContTerrain` není přímo odpovědná za generování modelu. Udržuje informace o výškové mapě, normálové mapě, dále pak o chybách v jednotlivých trojúhelnících a jejich prioritách. Třída `wContTerrain` dále obsahuje seznam segmentů terénu. U algoritmu ROAM totiž nelze generovat model pro potenciálně nekonečně velký terén. To z toho důvodu, že je třeba počítat priority pro celý terén a na jejich základě se pak rozvětvuje binární strom. Proto je vhodné terén rozdělit do několika částí. Tyto segmenty jsou čtvercové a skládají se ze dvou základních trojúhelníků, které spolu sdílejí přeponu. Aby se předešlo vzniku děr na rozhraní segmentů, musí se všechny segmenty navázat se svými sousedy. S použitím Force-splittingu se tak zajistí korektní dělení trojúhelníků a díry se v modelu neobjevují. Díky použití segmentů by bylo možné generovat i potenciálně nekonečný terén.

Pro segmenty je definována třída `wContTerrainSegment`. Objekt třídy `wContTerrain` tak nad všemi segmenty volá jejich metody a tím vygeneruje model.

V první fázi se spočítá statická chyba. Tou je rozdíl mezi skutečnou výškou bodu na středu přepony a průměrem výšek bodů přepony. Tato fáze se provádí pouze jednou a to při vytváření segmentu. Samotný výpočet obstarává metoda `RecurseCalculateError(...)`, která je nastartována právě při vytváření segmentu. V parametrech metody jsou body trojúhelníku, pro který chceme spočítat chybu. Spočítají se souřadnice bodu uprostřed přepony a spočítá se chyba. Následně se využijí předané body a souřadnice čtvrtého „prostředního“ bodu a rekurzivně se zavolá metoda `RecurseCalculateError(...)`. Tím získáme chyby synovských trojúhelníků. Výslednou chybou je pak maximální hodnota z chyby aktuálního trojúhelníku a chyb synovských trojúhelníků. Rekurze se zastaví v případě, že již nejsou podrobnější data. V takovém případě je chybou pouze rozdíl mezi průměrnou výškou a skutečnou výškou. Nakonec se chyba uloží a vrátí jako návratová hodnota funkce.

Po spočítání statické chyby musíme spočítat prioritu. Ta se počítá velmi podobným způsobem jako statická chyba. Start výpočtu se provede zavoláním metody `CalculatePriority(...)`, která poté volá rekurzivní metodu `RecurseCalculatePriority(...)`. Parametry funkce jsou souřadnice trojúhelníku, pro který se počítá priorita. Dále pak mapa s chybami a mapa s prioritami. Pro bod na středu přepony se načte jeho chyba a spočítá se jeho vzdálenost od pozorovatele. Podílem těchto dvou hodnot se spočítá priorita trojúhelníku. Ta se nakonec uloží do Priority-mapy a metoda se rekurzivně volá pro synovské trojúhelníky.

Nyní je nutné nachystat binární strom. Pro tyto účely slouží metoda `AdaptModel(...)`. Ta nastartuje rekurzivní metodu `RecurseAdaptModel(...)`. Do této metody se předává příslušný uzel

binárního stromu, vrcholy trojúhelníku, mapa s prioritami a mez, při které již nedochází k dělení. V metodě se pak porovná hodnota načtená z Priority-mapy a mez. Pokud lze dělit, provede se rozdělení trojúhelníku metodou `Split(...)` a rekurzivně se zavolá metoda `RecurseAdaptModel(...)` pro synovské trojúhelníky.

V metodě `Split(...)` se nejdříve ověří, zdali již uzel není rozdělený. V případě, že soused přes přeponu (takzvaný BaseNeighbour) je k trojúhelníku napojený jinak než přes přeponu, provede se Force-split. Na to se vytvoří synovské uzly a provede se provázání s existujícími sousedy a případně Force-split BaseNeighbour trojúhelníku.

Tímto máme nachystaný strom, který reprezentuje výsledný model. Pro vykreslování ale potřebujeme sadu vrcholů a trojúhelníků. Na řadu tak přichází metody `CreateGrid(...)` a `RecurseCreateGrid(...)`. Zde se jako parametry předává příslušný uzel stromu, souřadnice trojúhelníku, seznam vrcholů, seznam trojúhelníků a reference na indexy vrcholů, které přísluší danému trojúhelníku. Pokud se nenacházíme v listu stromu, pak přenecháme vytváření vrcholů a trojúhelníků na synovských uzlech. Zavolá se rekurzivně `RecurseCreateGrid(...)`. Pokud ale jsme v listu stromu, zkontrolujeme reference na indexy vrcholů. Pokud jsou naplněné, znamená to, že dané vrcholy již byly vygenerované a použijeme tyto indexy pro vytvoření trojúhelníku. Vrcholy znovu negenerujeme. Pokud reference naplněné nejsou, vytvoříme vrcholy i trojúhelník a reference naplníme. Tímto se distribuují indexy vrcholů a zmenšuje se tak nutnost prohledávat celý seznam vrcholů na duplikáty. K tomu ale občas dochází a to při vytváření nových vrcholů.

Nyní máme připravenou síť trojúhelníků. Zbývá načíst výšky vytvořených vrcholů, normály pro tyto vrcholy a souřadnice pro textury, k čemuž se využijí příslušné mapy. Všechna tato data se nahrají do předaných struktur. K tomu jsou připraveny metody `CreateModelData(...)` a `CopyModelData(...)`, které jsou volány metodou `LoadModel(...)`.

Implementovaný algoritmus má nevýhody standardního ROAM algoritmu bez optimalizací. Až na statickou chybu se v každém snímku vše přepočítává. Naše verze nepočítá ořezávání pohledovým tělesem, takže se přepočítává celý terén. Na druhé straně se pak ale nemusí přepočítávat model při rotaci kamery, ale jen při jejím posuvu. V originále byl také zmenšen rozměr obou map a nepočítalo se proto do takové hloubky. Náš algoritmus toto počítání nezjednodušuje. Tím získáme vyšší přesnost metody, ale za cenu větší časové náročnosti. Posledním rozdílem je, že originální program si připravil celý blok paměti pro uzly stromu předem a programově se pak řídí přidělování pro jednotlivé uzly. Měření vlastní implementace ROAM algoritmu jsou v kapitole Experimenty.

Model – vlastní metoda

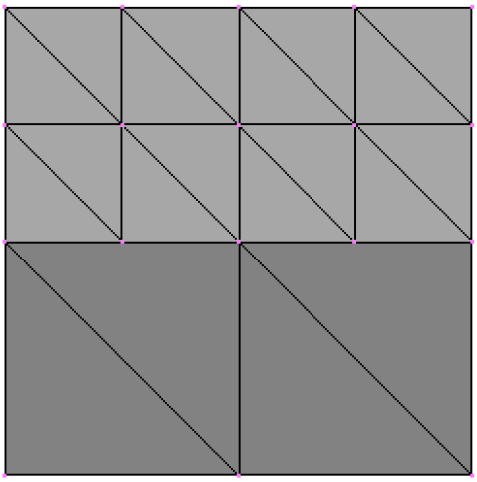
Pro model jsme se snažili najít metodu, která by byla velmi jednoduchá na implementaci a přitom umožňovala LOD. Žádnou jsme ale nenašli, proto jsme se rozhodli vytvořit vlastní, která bude zohledňovat vzdálenost od pozorovatele a bude řešit korektní navazování částí s odlišným detailem.

Vzhledem k tomu, že v metodě zohledňujeme pouze vzdálenost od pozorovatele, můžeme celý model vygenerovat hned při startu a vyhneme se tak výraznému upravování modelu za běhu. V terénu se mění výška vrcholů, ale trojúhelníková síť zůstává stále stejná, proto při úpravě modelu načteme pouze nové výšky.

Pro vytvoření modelu si nejprve vytvoříme bloky, které představují stupně detailu. Těmito bloky nepokryjeme celý terén, ale pouze čtvrtinovou výseč. Trojúhelníková síť je totiž osově

souměrná a lze jí tak vygenerovat pouze část a tu pak transformovat na zbylé plochy. Výšky vrcholů se načtou až dodatečně.

V první fázi se tak vytvoří definice jednotlivých stupňů detailu. Poté dojde ke generování vrcholů a trojúhelníků. V cyklu se prochází interval daný limity souřadnic a postupuje se s daným krokem. V tomto momentě vytvoříme trojúhelník a jeho vrcholy. Při vytváření vrcholů kontrolujeme,



Obrázek 3.2: Rozhraní dvou bloků bez korekcí

zdali už tyto vrcholy neexistují. V takovém případě pro definici trojúhelníku použijeme indexy dříve vytvořených vrcholů.

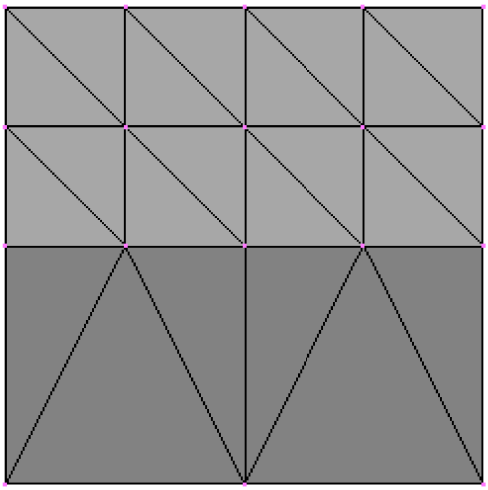
Tím generujeme bloky, které na sebe nenavazují ideálně (Obrázek 3.2) a dochází ke vzniku děr. Proto před vytvořením trojúhelníku musíme vědět, že je tento trojúhelník na hranici. Díky startovní a konečné souřadnici toto lehce poznáme. Pak generujeme trojúhelníky odlišným způsobem (viz Obrázek 3.3). a vzniku děr tím zabráníme.

Takto vygenerovanou čtvrtinu terénu využijeme pro zbytek plochy. Postupně projdeme všechny vrcholy a trojúhelníky a vytvoříme z nich zbylé tři čtvrtiny modelu. Toho dosáhneme prostou úpravou souřadnic vrcholů a použití indexů nových vrcholů.

V samotné implementaci se o vytvoření modelu stará `wLoaderTerrainD`. Pro definici bloku je použita třída `wLOD`. Ta si uchovává velikost kroku, počet kroků a seznam indexů vrcholů. Těchto bloků se vytvoří několik a jejich vlastnosti jsou vhodně definovány. Pak se uvnitř těla metody `CreateGrid(...)` cyklem projdou všechny tyto bloky a metodou `GenerateGrid(...)` se generuje síť trojúhelníků. Této metodě se předávají hranice bloků. Generování pak postupuje podle těchto a vnitřních parametrů.

V metodě `GenerateGrid(...)` je generování rozděleno na tři části. V cyklu se čítají souřadnice a jeho počátek, konec a krok jsou určeny parametry. V této fázi se podle souřadnic rozhodneme jestli generujeme síť uvnitř nebo na rozhraní bloku. Podle toho pak generujeme trojúhelníky. K tomu slouží metoda `MakeFace(...)`. Té předáme body trojúhelníku, který chceme vytvořit. Zkontrolujeme již vytvořené body a pokud jsou mezi nimi body, které nyní potřebujeme, použijeme jejich indexy. V opačném případě vygenerujeme body nové. Poté pomocí indexů

vytvoříme trojúhelník. Pro vyhledávání existujících bodů jsem použil několik metod k urychlení, jejichž rozdíly jsou popsané a otestované v kapitole Experimenty.



Obrázek 3.3: Rozhraní dvou bloků s korekcemi navázání

Další fází metody `CreateGrid(...)` je dotvoření sítě pro zbylé části terénu. Toho dosáhneme „převrácením“ hotové části. Tím získáme polovinu modelu. Stejný princip použijeme ještě jednou. Převrátíme tedy hotovou polovinu a získáme tak zbytek modelu.

V závěru musíme ještě načíst výšky, normálové vektory a vytvořit texturovací souřadnice. Výšky a normály načteme z výškové a normálové mapy. Nakonec vytvoříme texturovací koordináty pro každý vrchol. Ty spočítáme z X a Z souřadnic vrcholů.

Vrcholy a trojúhelníky modelu jsou uloženy uvnitř loaderu. Když pak potřebujeme model s daným posuvem po mapě, zavoláme metodu `Load(...)`. Posuv do ní předáme pomocí varianty modelu. Voláním dalších metod se zajistí načtení vrcholů a trojúhelníků, nahrání odpovídajících normál a výpočet texturovacích souřadnic. To vše je nakonec přesunuto do připravených struktur.

Metoda má své nedostatky. Hlavním je, že zohledňuje pouze vzdálenost od kamery. Navíc pracuje po blocích, takže optimálnost modelu opět klesá. Výhodou ale je, že model je z velké části předpočítaný. Úpravy, které se provádějí za běhu, jsou díky tomu časově nenáročné. Dále máme kontrolu nad rozsáhlostí a komplexností modelu a můžeme tak dobře zatížit grafickou kartu.

Osvětlení

Pro osvětlení jsem použil metodu počítání Light-mapy. O to se tak stará třída `wLoaderLight`. Vytvoření světelné mapy je obsaženo přímo v metodě `Load(...)`.

V první fázi se zpracují parametry metody. Od správce zdrojů se získají textury, které jsou nezbytné pro vygenerování světelné mapy. Těmi jsou výšková mapa, normálová mapa a mapa výšky stínu. Dále se připraví struktura, do které budou nahrány vypočtené hodnoty osvětlení. Nastaví se rozměry textury a počet bytů na pixel.

V další fázi je vypočten směrový vektor osvětlení. Ten se získá z varianty textury a pro použití v dalších výpočtech jej převedeme do vhodného tvaru.

Poté již počítáme osvětlení pro každý pixel Light-mapy. Načte se normálový vektor, výška pixelu a výška stínu. Pomocí vzorce pro kartézský součin ze směrového vektoru osvětlení a normálového vektoru spočítáme intenzitu dopadajícího světla. Na to ještě zjistíme ambientní složku osvětlení. Rozdílem výšky pixelu a výšky stínu zjistíme, zdali a jak intenzivně je pixel zastíněn, a podle toho pak spočítáme zabarvení pixelu jednotlivými složkami a výslednou hodnotu osvětlení. Tu nakonec uložíme do připravené struktury.

Původní implementace měla dva nedostatky. Při zobrazování noční krajiny nebylo vůbec výrazné stínování kopců, protože difúzní složka byla jen velmi malá a použitá ambientní složka byla pro celý terén konstantní. Proto jsem se rozhodl pro úpravu, kdy ambientní složku počítáme jako difúzní složku osvětlení, které vždy míří shora dolů. Jedná se tak o ekvivalent osvětlení dvěma zdroji světla s použitím pouze jejich difúzních složek. Ve dne je sice rozdíl méně znatelný, ale při zobrazení noční krajiny nepřicházíme o stínování kopců. Druhým nedostatkem byly ostré stíny, protože se pouze porovnávala výška pixelu a stínu. To jsem vyřešil použitím intenzity zastínění (resp. osvětlení) a tím tak dosáhl měkkých stínů.

Vrhané stíny

Stíny jsou v programu řešené výpočtem výšky stínu pro každý pixel. O to se stará `wLoaderShadow` a jeho metoda `Load(...)`. Ke správnému výpočtu potřebuje pouze výškovou mapu a od správce zdrojů si ji získá.

Tak jako u výpočtu osvětlení, i zde je ve variantě textury zakódovaný směrový vektor osvětlení, a ten si tedy stejným způsobem spočítáme a upravíme do vhodné formy. Poté se v cyklu počítá výška stínu pro každý pixel. Pro vyšší přesnost se počítá několik vzorků. Výsledná hodnota výšky stínu se pak uloží do připravené textury.

Experimentováním jsem v algoritmu zvolil počet vzorků a jejich rozestupy, takže je dosaženo přiměřené přesnosti výpočtu a zároveň je výpočet proveden poměrně rychle.

Obarvení

Pro obarvení terénu není použit žádný speciální loader. To proto, že jsou k obarvení použity textury. K jejich načítání slouží několik loaderů rozlišených podle načítaného formátu. Kromě formátu RAW, jehož načítání jsem řešil vlastními silami, je použito knihovny SDL_image.

Pro použití textury je nutné modelu terénu nastavovat texturovací souřadnice. Tento výpočet se provádí při generování modelu a jde o jednoduchou transformaci souřadnic vrcholu na texturovací koordináty.

Použitá textura je v nižším rozlišení. Je tím dosaženo základního obarvení a detail je tak nutné zajistit jinou texturou. V našem řešení zobrazujeme pouze malý terén, implementováno tak je pouze použití jedné detailní textury. K namíchání obarvení je pak využito multitexturingu, který dnes podporuje snad každá grafická karta. Multitexturing se využívá i pro aplikování vypočtené světelné mapy.

3.2.3 Zobrazování oblohy a slunce

Pro náš účel zobrazení malé části krajiny stačí použít Skybox či Skydome. Na základě jejich vlastností jsme se rozhodli implementovat zobrazování oblohy pomocí Skydome, který je obarvován po vrcholech. Slunce je proto zobrazováno jako polygon s texturou.

Vytváření modelu oblohy i následné barvení zajišťuje `wLoaderSkydome`. Konkrétně se jedná o metodu `Load(...)` a dále metody `CreateDome(...)`, `CalculateColors(...)` a `CopyModelData(...)`.

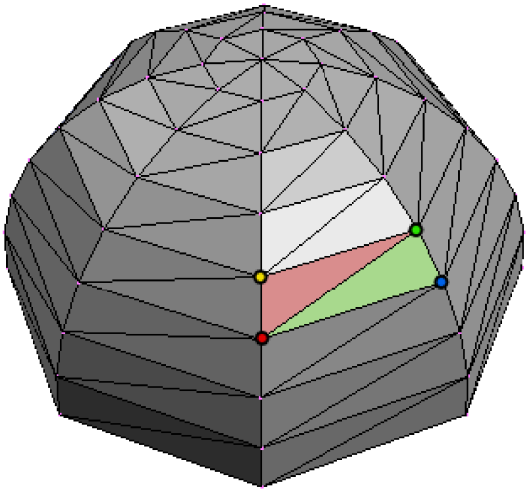
Při načtení modelu se podle varianty rozpozná pro jakou denní dobu se obloha generuje. Model se ale vytvoří pouze při prvním zavolání metody `Load(...)` a loader si jej uchovává pro další použití. Při dalším volání metody `Load(...)` se model jenom zkopíruje z uchované struktury. Přepočítává se pouze obarvení modelu.

Model oblohy a slunce

Generování modelu oblohy zajišťuje metoda `CreateDome(...)` třídy `wLoaderSkydome`. Generování je zajištěno pomocí postupného vytvoření vrcholů. Ty se vytvářejí po úrovních, kdy jedna úroveň odpovídá určité výšce. Začíná se spodní hranou a postupně se přidávají další řady vrcholů.

Souřadnice vrcholů se počítají z úhlů, které jsou postupně čítány. Generování vrcholů je rozděleno do dvou částí, kdy se v prvním cyklu generuje řada a ve vnořeném cyklu se tvoří vrcholy této řady.

Generování řad probíhá tak, že se začne úhlem 0°. Vygeneruje se řada vrcholů a přičte se úhel odpovídající detailu. Pak se generuje další řada. Takto se generuje až do 90°, kdy už žádnou další řadu netvoříme. Tvoření řady začíná úhlem 0° a postupně se přičítá úhel odpovídající detailu modelu. Čítání se zastaví na 360°.



Obrázek 3.4: Vrcholy ČZŽ a ČMZ tvoří trojúhelníky při generování modelu

Po vygenerování všech těchto vrcholů se přidá ještě jeden vrchol – poslední. Dal by se označit za „špičku“ Skydomu.

Poté se musí vygenerované vrcholy spojit do trojúhelníků. Vrcholy jsou tvořeny v přesně daném pořadí, takže z nich lze celkem snadno skládat trojúhelníky. Pro každý vrchol se tak provede navázání s dalším vrcholem v další řadě a s vrcholem na stejné pozici, ale v další řadě. Hned na to se ještě vytvoří druhý trojúhelník. Naváže se tak aktuální vrchol s dalším vrcholem ve stejné řadě a s dalším vrcholem na další řadě. Tyto trojúhelníky a vrcholy jsou znázorněné na obrázku (Obrázek 3.4).

Takto se tvoří trojúhelníky až po předposlední řadu. Poslední řada totiž musí být navázána na zmíněnou špičku Skydomu, kdy každý vrchol poslední řady navážeme s dalším vrcholem v řadě a se špičkou.

Pak se pro model počítá obarvení. Tyto barvy jsou spolu s vygenerovaným modelem uložené ve vnitřních strukturách loaderu, odkud je model zkopírován do předaných struktur a poté je již možné model zobrazit.

Zbývá způsob vytvoření modelu slunce. Jedná se pouze o dvojici trojúhelníků a jednu texturu. Proto jsem pro slunce netvořil žádný speciální loader a model se vytvoří až těsně před jeho odesláním do grafické karty. Vytvoří se dva trojúhelníky, které dávají iluzi čtverce. Spočítá se poloha a rotace, aby slunce vždy směřovalo ke kameře. Od správce zdrojů se získá textura slunce a nakonec dojde k vykreslení.

Barvení oblohy

Dále popsaná metoda je výsledkem postupného upravování a experimentování, nejde o aproximace metod popisujících přesné výpočty obarvení oblohy.

Při vymýšlení výpočtu jsme uvažovali, že obloha má vždy spíše modrou barvu. Dále jsme brali v úvahu, že se obloha blízká horizontu jeví světlejší a zrovna tak je obloha světlejší v místech blízko slunce. Veškeré tyto projevy jsou vidět kdykoliv během dne. Různí se ale jejich intenzita podle polohy slunce. To jsme se taktéž snažili zohlednit.

Barvení oblohy je prováděno po vrcholech, kdy se pro každý vrchol na základě jeho souřadnic a normálového vektoru spočítá jeho barva. Při rasterizaci se pak tyto barvy interpolují na plochy trojúhelníků. Obarvení vrcholů je součástí loaderu pro generování Skydome a jedná se o metodu `CalculateColors(...)`. Pro počítání barev musíme znát barvu světelného zdroje. Tuto barvu loader vždy dostane těsně před žádostí o vytvoření modelu.

V metodě jsou ještě definovány další barvy – AttColor, FarColor a SkyColor. AttColor popisuje modrý „odstín“ oblohy. FarColor určuje barvu, která odpovídá barvě blízko horizontu a SkyColor pak slouží k barvě použité jako barva oblohy v zenitu. Barva použitá přímo pro vrchol modelu se počítá z této sady barev s využitím směrového vektoru světla.

Barva v zenitu se spočítá po složkách. Každá složka je součinem ambientní složky osvětlení a AttColor. Jako barva blízko horizontu je použita barva ambientní složky světelného zdroje, popřípadě barva v zenitu. Použije se ta světlejší. To zajistí, že ve večerních hodinách, kdy je ambientní složka velmi tmavá, nedojde k začernění vzdálených oblastí, ale obloha je spíše rovnoměrně zbarvená.

Nyní probíhá cyklus, který projde všechny vrcholy modelu a spočítá se pro ně obarvení. Díky tomu, že vrcholy modelu jsou body koule se středem v počátku souřadnicového systému [0.0, 0.0, 0.0], dají se jejich souřadnice použít i jako normálové vektory v nich. Stačí provést normalizaci.

Na to je spočítán skalární součin tohoto normálového vektoru a směrového vektoru osvětlení. Tohoto výsledku je pak využito pro zesvětlení oblastí v blízkosti pozice slunce. Poté je na základě výšky vrcholu spočítán poměr FarColor a SkyColor pro tento bod.

Výsledná barva vrcholu je pak spočítána podle následujícího vztahu:

$$OutColor = (1 - Height) \cdot FarColor + Height \cdot SkyColor + (\vec{N} \cdot \vec{D}) \cdot LightAmbiColor \qquad (15)$$

Spočítaná barva je nakonec uložena do nachystané struktury a společně s modelem použita pro zobrazování.

Denní cyklus

Ač to nebylo v požadavcích, již od počátku návrhu programu jsem chtěl, aby byl engine schopen znázornit různé doby dne a lépe je i v průběhu zobrazování měnit. Design programu je tak pro toto použití navržený.

Čas scény je udržován v třídě `wGameTime`, která obsahuje datum a čas až do přesnosti milisekund. Tento čas je při běhu programu v každém snímku aktualizován a přičítá se čas, který byl potřeba k vytvoření a vykreslení minulého snímku. Pro testování byl přidán i násobič, takže je možné čas ve scéně posouvat rychleji než běží ten reálný.

Samotný čas ale neslouží k získání barvy a polohy slunce. Pro tu činnost je navržena třída `wSunLight`. Metodou `Update(...)` jí předáme tento čas a ona si z něj spočítá směrový vektor a barvu, které si pak jen vyžádáme. Pokud části programu potřebují polohu slunce, spočítá se právě ze směrového vektoru.

Při počítání směrového vektoru jsme celou záležitost značně zjednodušili. Zanedbali jsme totiž změny polohy slunce, které vznikají během roku. Slunce tak obíhá kolem modelovaného světa a to po stále stejné kružnici.

Polohu slunce nemusíme měnit příliš často. Proto se poloha počítá pouze s přesností minut. Tím získáme 1440 možných poloh slunce, které mohou během dne nastat. Jednoduchým způsobem lze spočítat úhlovou polohu slunce na zvolené kružnici. Z té se pak pomocí goniometrických funkcí sinus a cosinus spočítá směrový vektor osvětlení.

Dalším problémem je barva slunce. Ta se v realitě nemění, slunce svítí stále stejně. Zabarvení je pak ovlivněno úhlem pod jakým sluneční paprsky dopadají na naši planetu a průchodem atmosférou. Jak již ale bylo několikrát zmíněno, takové počítání je příliš složité a pro náš účel nevhodné. Proto musíme tyto změny osvětlení simulovat změnou barvy slunce.

Odhadem jsem tak vybral dvanáct barev pro určité časové body a vložil je do programu. Aby barva slunce po každých dvou hodinách nepřirozeně neskočila, počítá se aktuální barva lineárním přechodem mezi dvěma krajními barvami. Spočítá se tedy jenom to, jak daleko se v současném čase nacházíme oproti zvoleným bodům a podle toho se vezme příslušná část barvy v těchto bodech.

4 Experimenty

V této kapitole se věnuji testům implementovaných částí. Jsou provedena měření různých částí s různými parametry nastavení a to na několika testovacích strojích. Některé části nelze testovat jen na časovou či paměťovou náročnost, ale je třeba porovnávat kvalitu jejich obrazového výstupu. V těchto částech jsou výsledky testů doloženy obrázky.

Pro pokud možno co nejkonzistentnější testy byla implementována automatická kamera, která se v kruzích pohybuje po terénu a rotuje. Je tak zajištěno otestování funkcionality úprav resp. změn modelu terénu na základě změn polohy kamery.

4.1 Testovací stroje a metodika

	Stroj č. 1	Stroj č. 2	Stroj č. 3
Procesor	Pentium 4 – M, 2.20 GHz	Pentium D 915, 2.8 GHz	Pentium DualCore E5200, 2.5GHz
Grafická karta	Mobility Radeon 7500	Radeon X300	Radeon HD3650

Tabulka 1: Testovací stroje

Všechny testy jsou prováděny v rozlišení 1024 x 768 pixelů s 32 bitovou barevnou hloubkou. Na všech strojích je vypnuta vertikální synchronizace a program je nastaven tak, aby sám neomezoval FPS. Zjistíme tak maximum, kterého jsou dané sestavy schopné, a tím i rezervy, které sestavy mají.

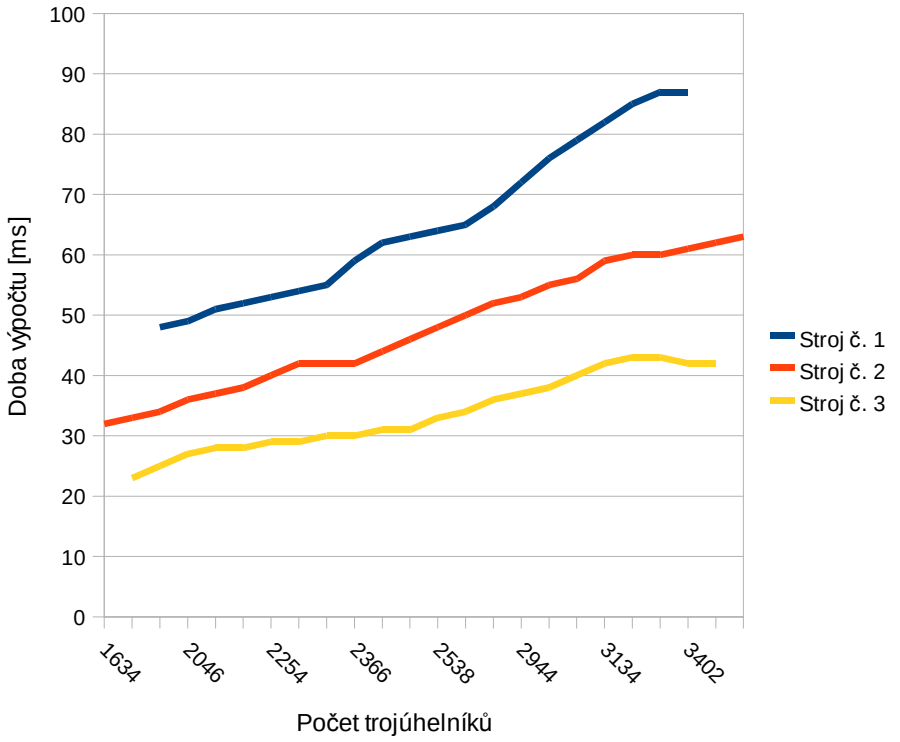
Pomocí měření uvnitř programu, která se vypisují do souboru, můžeme zjistit trvání dílčích výpočtů. Dále se vypisují vlastnosti generovaných modelů. Těmito vlastnostmi jsou počty vrcholů, trojúhelníků, normál a texturovacích souřadnic. Z nich se dají určit paměťové nároky, které odpovídají daným modelům.

4.2 ROAM metoda

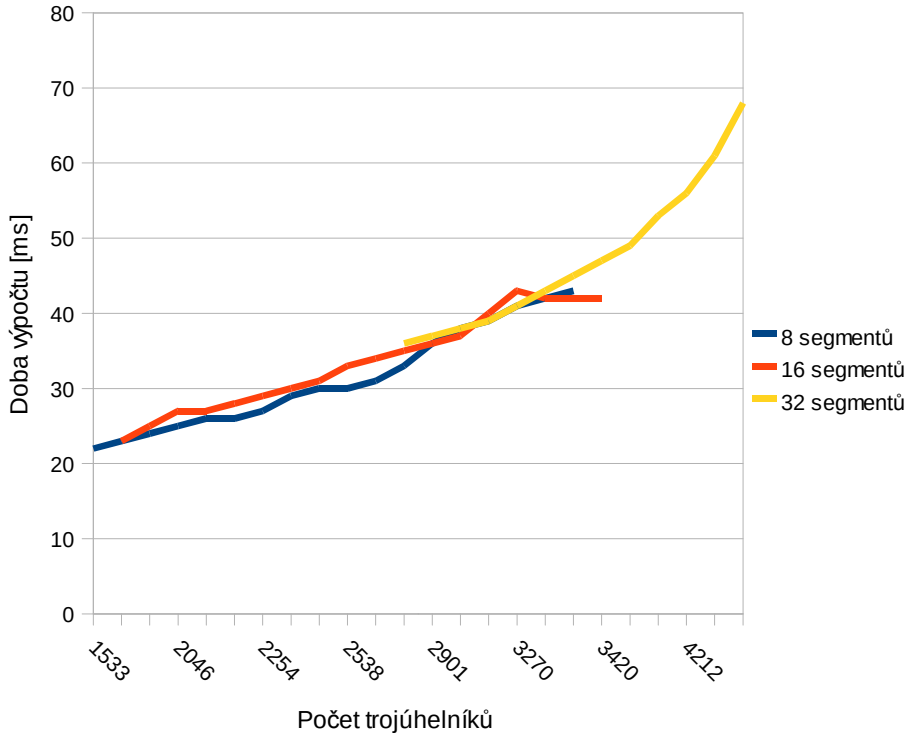
Všechny testy jsou provedeny při velikosti terénu 1025 x 1025 vrcholů. V testech jsem se zaměřil na vliv počtu segmentů, do kterých je terén rozdělen, a také na náročnost implementované metody při zvoleném prahu. Měřena je doba nutná k výpočtu a počet vygenerovaných trojúhelníků, který přímo ovlivňuje kvalitu výstupu a použitelnost pro interaktivní režim.

Pro dostatečných 25 snímků za sekundu potřebujeme, aby vše bylo spočítáno rychleji než za 40 milisekund. Z grafu (Graf 1) lze vyčíst, že metoda není příliš rychlá ani na relativně výkonném stroji č. 3. Za požadovaný čas získáme model s přibližně 3000 trojúhelníky, což je velmi málo.

Na základě dalšího měření (Graf 2) jsem zjistil, že počet segmentů rychlost výpočtu ovlivňuje jen velmi nevýrazně. Použitím vyššího počtu segmentů ale dosáhneme většího detailu ve vzdálených oblastech terénu – počtem segmentů totiž určíme nejhorší možný detail.



Graf 1: Závislost rychlosti výpočtu na počtu generovaných trojúhelníků u testovacích strojů.



Graf 2: Závislost rychlosti výpočtu na počtu generovaných trojúhelníků při použití různého počtu segmentů.

4.3 Vlastní metoda

Vzhledem k tomu, že má implementace ROAM algoritmu není dostatečně rychlá, aby stačila této vlastní metodě, při které se na rozdíl od ROAM varianty model mění jen zřídkakdy, zaměřím se zde především na porovnávání vlastností modelu, generovaného s různými úpravami pro vyhledávání duplicitních vrcholů v modelu. Pro porovnání s ROAM algoritmem ale uvedu i FPS, kterého dosahuje moje vlastní metoda.

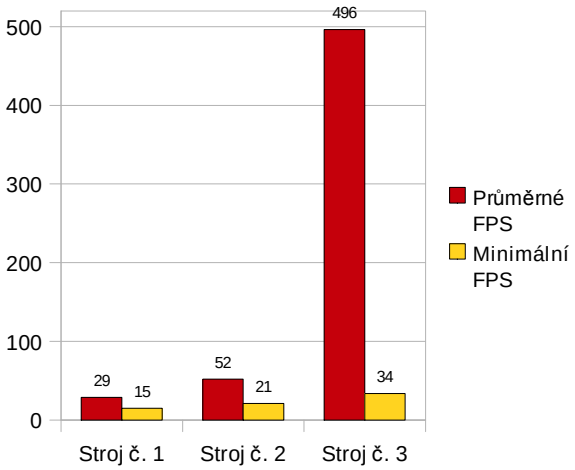
V následující tabulce (Tabulka 2) jsou uvedeny časové údaje (v milisekundách) změřené během generování modelu a vlastnosti tohoto modelu s použitím různých úprav algoritmu pro vyhledání a eliminaci duplicitních vrcholů.

Vyhledávání	Počet vrcholů	Obsazená paměť [kB]	Čas potřebný k vytvoření modelu [ms]			Čas potřebný k úpravě modelu [ms]		
			Stroj č. 1	Stroj č. 2	Stroj č. 3	Stroj č. 1	Stroj č. 2	Stroj č. 3
žádné	118656	4171	234	179	157	211	153	118
1	115416	4070	11134	9158	8233	205	148	115
2	21656	1140	1439	1160	1051	46	33	26
3	21996	1150	281	229	203	47	34	27
1+2	20356	1099	2373	1923	1736	48	32	25
1+3	20676	1109	1313	1094	980	54	33	25

Tabulka 2: Změřená doba trvání generování modelu s různými úpravami

- Žádné: varianta bez jakéhokoliv vyhledávání. V tomto případě se pro každý trojúhelník vygeneruje trojice zcela nových bodů. Tato metoda je rychlá na vytvoření, ale všechny další operace s modelem jsou velmi náročné.
- Úprava č. 1: V tomto případě se prohledávají vrcholy trojúhelníků ze sousedního bloku. Eliminují se tak duplikáty na švech mezi bloky. Těch je velmi malé množství, proto výsledky nejsou o mnoho lepší než při generování bez vyhledávání.
- Úprava č. 2: V této variantě se prohledávají vrcholy, které již byly v daném bloku vygenerovány. Tato úprava je již velmi efektivní a výsledný model není příliš komplexní pro úpravy za běhu programu.
- Úprava č. 3: Tato úprava vylepšuje předešlou úpravu. Prohledávají se jen nedávno vytvořené vrcholy. Některé duplicitní vrcholy se sice zanedbají, ale algoritmus je časově mnohem méně náročný.
- Kombinace úprav 1 a 2: Tato varianta kombinuje předešlé metody. Časová náročnost je vyšší než při použití pouze varianty č. 2, ale nalezneme více duplicitních vrcholů.
- Kombinace úprav 1 a 3: Tato varianta opět kombinuje předešlé metody. Časová náročnost je taktéž vyšší než při použití pouze varianty č. 3, ale zároveň nižší než předchozí kombinace.

Na základě naměřených hodnot je v programu použita varianta č. 3. Získáváme tak model s nízkým počtem vrcholů. Při generování modelu se vytváří i stejný počet normál a texturovacích souřadnic. Díky nízkému počtu těchto dat má výsledný model i nízké paměťové nároky. Navíc lze model rychle upravit při běhu programu a netrpí tak plynulost zobrazování.

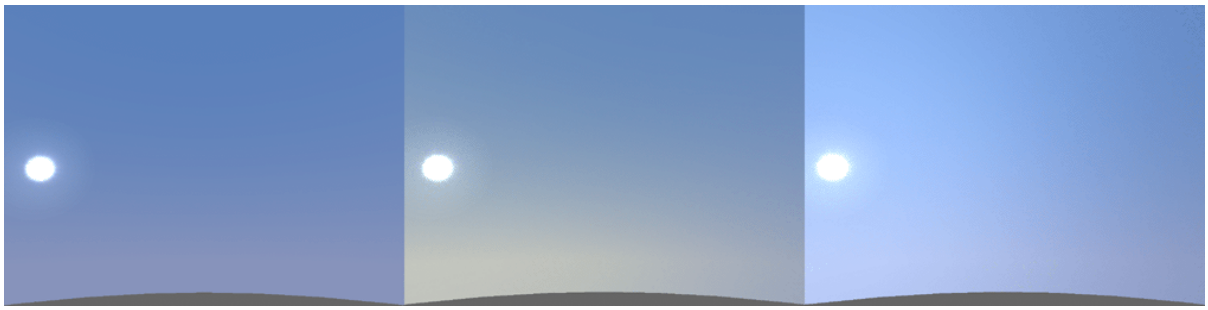


Graf 3: FPS dosahované na různých konfiguracích

Na základě hodnot v grafu (Graf 3) lze rozhodnout, že tato metoda je rychlejší než implementovaný ROAM algoritmus. Minimální FPS je sice nízké, ale jedná se pouze o jednorázové poklesy, které nejsou za běhu příliš znatelné. Průměrné FPS značí, že metoda je použitelná i na strojích s výkonově slabší hardwarovou konfigurací. V testu byl použit model s téměř 40000 trojúhelníky. Jeho generování se dá upravit nastavením generátoru či úpravou externího souboru s parametry pro program.

4.4 SkyDome

V této kapitole se zaměřím na rozdíly mezi variantami obarvení, které byly postupně implementovány, a na základě kterých jsem se rozhodl pro výběr finální varianty obarvení.



Obrázek 4.1: Varianty obarvení oblohy

Na obrázku (Obrázek 4.1) jsou zobrazeny různé způsoby obarvování oblohy. První varianta (vlevo) implementuje změnu obarvení pouze na základě výšky vrcholu. Je tak simulováno zamlžení vzdálených oblastí blízko horizontu. Druhá varianta (uprostřed) přidává změnu barvy zamlžení podle úhlu, který svírá normálový vektor vrcholu a směrového vektoru osvětlení. Je patrný značný posun v kvalitě. Pro dosažení ještě vyššího realismu obarvení jsem metodu č. 2 dále upravil. Třetí varianta (vpravo) tak mění obarvení na základě výšky i podle svíraného úhlu mezi normálou vrcholu a směrovým vektorem osvětlení. Barva upravená podle úhlu se ale nemění pouze pro zamlžené oblasti, ale pro celý model.

Všechny tyto metody byly vyvinuty postupně a jsou jistou evolucí původního obarvovacího algoritmu. Ten užíval pouze jednu barvu pro SkyDome a zamlžení simuloval použitím mlhy v OpenGL.

Podle srovnávacích snímků je zřejmé, že poslední implementovaná metoda zohledňuje nejvíce jevů, které lze na obloze pozorovat, a je tedy z těchto metod nejrealističtější.

4.5 Další testy

Zde nejsou testovány různé varianty algoritmů, ale pouze jsou změřena trvání výpočtů Normal-mapy, Light-mapy a ShadowVolume-mapy a dopad na použitelnost v real-time aplikacích.

Podle naměřených výsledků uvedených v tabulce (Tabulka 3) lze prohlásit, že výpočet stínů a osvětlení není příliš vhodný pro použití k zobrazování dynamického denního cyklu. Při použití jsou patrné prudké poklesy snímkové frekvence. K tomuto účelu by tak byly potřeba vhodnější metody výpočtů či současné výpočty upravit a využít programovatelných jednotek na grafických kartách.

Naproti tomu výpočet mapy s chybami pro použití ROAM algoritmem a výpočet normálové mapy mohou být zachovány beze změn, protože se provedou pouze jednou.

Doba výpočtů [ms]	**Stroj č. 1**	**Stroj č. 2**	**Stroj č. 3**
Výpočet Error-mapy	20349	14549	10282
Výpočet Light-mapy	248	172	127
Výpočet Normal-mapy	1423	1187	351
Výpočet ShadowVolume-mapy	549	329	184

Tabulka 3: Doba trvání výpočtů vybraných dat na několika hardwarových konfiguracích.

5 Budoucí práce a rozšíření

V této práci jsem postihl několik velmi důležitých aspektů zobrazování krajiny. V žádném případě ale nebyly všechny a rád bych se tomuto tématu věnoval dále.

Bylo by možné vylepšit generování modelu terénu. Především metoda ROAM, která byla implementována jen v základní verzi bez optimalizací. S jen mírnými úpravami by ji bylo možné transformovat na RUSTiC metodu, čímž by se lépe využil výkon grafické karty, popřípadě snížilo zatížení procesoru. Významné by bylo i ořezávání pohledovým tělesem.

Ve vlastní metodě generování modelu již pravděpodobně nelze postoupit dále. Byla navržena velmi specificky a svůj účel plní dobře. Na čem by se ale ještě nejspíš dalo zapracovat, je vyhledávání duplicitních vrcholů či případně generovat vrcholy ve vhodnějším pořadí, kde by bylo možné duplikáty kompletně eliminovat jako v případě generování Skydome.

Dále bych chtěl prozkoumat možnosti pro zobrazování potenciálně nekonečného terénu, kde vzniká problém použití velkého množství rozměrných textur k získávání dat pro model a zobrazování. Mimo to zůstaly neprozkoumány další oblastí, jako třeba zobrazování mraků a zobrazení mlhy v prostoru. Dále nesmím opomenout vodní plochy a vegetaci.

Co se týče výkonu, zde je pořád co vylepšovat. Při zobrazování lze místo vertex arrays použít vertex buffer objects. Nárůst výkonu by mohlo zajistit doimplementování oktalového stromu a ořezávání pohledovým tělesem a tím tedy vykreslováním jen těch objektů, u kterých to má smysl. Pro přípravu světelné mapy a mapy stínů by se pravděpodobně daly využít technologie moderních grafických karet a výrazně tak urychlit tyto výpočty. Tím by bylo možné zobrazovat plynulý a rychlý denní cyklus a snížily by se nároky na výkon procesoru. V některých částech by se dalo uvažovat i o paralelizaci a pracovat s více vlákny a opět tak lépe využít dnešních technologií a jejich výkonu.

6 Závěr

Cílem této práce bylo prozkoumat metody používané pro zobrazování krajiny. Dále pak implementace vhodných metod či navržení a implementace metod vlastních. To vše pro zobrazení v reálném čase.

Při vývoji jsem myslel na možnost budoucí práce a tak jsem vyloučil navržení velmi prosté aplikace a namísto toho byl vytvořen program dostatečně robustní a modifikovatelný. Na základě experimentů se mi podařilo splnit cíl o interaktivitě a stále jsou určité rezervy pro zobrazování dalších částí krajiny či detailnějšího modelu terénu. V některých ohledech se mi podařilo překonat má původní očekávání o běhu programu a s mírnými omezeními je možné zobrazovat v reálném čase i denní cyklus, přestože je vše počítáno procesorem. Další vlastnosti programu, které zatím nebyly implementovány, plánuji prozkoumat a prakticky otestovat v budoucí práci.

Literatura

[1] de Boer, W. H.: Fast terrain rendering using geometrical mipmapping [online]. Říjen 2000 [cit. 28.4.2009]. Dostupný z WWW: <http://www.flipcode.com/archives/article_geomipmaps.pdf>

[2] Duchaineau, M. a kolektiv: ROAMing terrain: Real-time optimally adapting meshes [online]. [cit. 28.4.2009]. Dostupný z WWW: <http://www.llnl.gov/graphics/ROAM/roam.pdf>

[3] Turner, Bryan: Real-Time dynamic level of detail terrain rendering using ROAM [online]. 3.4.2000 [cit. 28.4.2009]. Dostupný z WWW: <http://www.gamasutra.com/features/20000403/turner_01.htm>

[4] Gyurchev, Y.: ROAM Implementation optimizations [online]. 24.7.2001 [cit. 28.4.2009]. Dostupný z WWW: <http://www.flipcode.com/archives/ROAM_Implementation_Optimizations.shtml>

[5] Pomeranz, A. A.: ROAM using surface triangle clusters [online]. 1998 [cit. 28.4.2009]. Dostupný z WWW: <http://www.cognigraph.com/ROAM_homepage/PomeranzThesis.pdf>

[6] Kolektiv autorů: Virtual Terrain Project [online]. 23.9.2009 [cit. 28.4.2009]. Dostupný z WWW: <http://www.vterrain.org/>

[7] Bloom, Ch.: Terrain Texture Compositing by Blending in the Frame-Buffer (aka "Splatting" Textures) [online]. 2.11.2000 [cit. 28.4.2009]. Dostupný z WWW: <http://http://cbloom.com/3d/techdocs/splatting.txt>

[8] Wright, R. S., Lipchak, B., Haemel, N.: *OpenGL Superbible, Fourth Edition: comprehensive tutorial and reference*. Addison-Wesley, 2007, ISBN 0-321-49882-8

Seznam příloh

Příloha 1. CD se zdrojovými kódy a spustitelným programem