VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

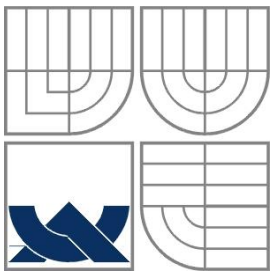
IMPLEMENTACE ALGORITMU SEAMLESS PATCHES FOR GPU-BASED TERRAIN RENDERING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

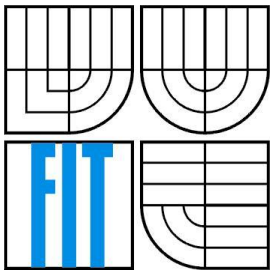
AUTOR PRÁCE
AUTHOR

BC. DAVID JOZEFOV

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

IMPLEMENTACE ALGORITMU SEAMLESS PATCHES FOR GPU-BASED TERRAIN RENDERING

SEAMLESS PATCHES FOR GPU-BASED TERRAIN RENDERING ALGORITHM IMPLEMENTATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. DAVID JOZEFOV

VEDOUCÍ PRÁCE
SUPERVISOR

ING. RADEK BARTOŇ

BRNO 2011

Abstrakt

Tato diplomová práce se zabývá vykreslováním terénu s využitím moderního algoritmu pro adaptivní úroveň detailů. Popisuje dvě v současnosti nejpoužívanější grafická aplikační rozhraní a jejich high-level nadstavby a shrnuje princip a vlastnosti několika používaných level-of-detail algoritmů pro zobrazování terénu. Podrobněji pak popisuje implementaci algoritmu Seamless patches for GPU-based terrain rendering.

Abstract

This master‘s thesis deals with terrain rendering using a modern algorithm for adaptive level of detail. It describes two currently most used graphical application interfaces and high-level libraries that use them and summarizes principles and features of several level-of-detail algorithms for terrain rendering. In more detail it then describes the implementation of Seamless patches for GPU-based terrain rendering algorithm.

Klíčová slova

3D počítačová grafika, zobrazování terénu, DirectX, XNA, LOD, Seamless patches.

Keywords

3D computer graphics, terrain rendering, DirectX, XNA, LOD, Seamless patches.

Citace

Jozefov David: Implementace algoritmu Seamless Patches for GPU-Based Terrain Rendering, diplomová práce, Brno, FIT VUT v Brně, 2011

Implementace algoritmu Seamless Patches for GPU-based Terrain Rendering

Prohlášení

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

.....
David Jozefov
25. května 2011

Poděkování

Děkuji svému vedoucímu Ing. Radku Bartoňovi za poskytnutý čas a odbornou pomoc při práci na mém projektu a svým rodičům a přítelkyni za podporu během mého studia.

Obsah

1 Úvod

Terén je velmi důležitou součástí grafických aplikací, jako jsou počítačové hry, letecké simulátory nebo virtuální prohlídky světa či cizích planet (např. aplikace Google Earth [1]). U všech těchto aplikací je nutné zobrazovat velmi rozsáhlá prostředí v reálném čase, ať již uměle vytvořená herními návrháři nebo ta skutečná zmapovaná například díky projektu SRTM [2]. Současný běžně dostupný hardware grafických karet nám však stále nedovoluje zobrazovat tak objemná data v jejich plné komplexnosti. Navíc si ve většině případů ani nemůžeme dovolit využít veškerý dostupný výpočetní výkon pouze pro vykreslování terénu, takže musíme využívat LOD algoritmů pro snížení jeho složitosti tak, abychom jej mohli zobrazovat v reálném čase a v ideálním případě bez viditelného snížení kvality výsledného obrazu.

V následující kapitole *Počítačová grafika* si řekneme, jak počítačová grafika vlastně vzniká, popíšeme si dvě hlavní aplikační rozhraní, která se dnes používají, a blíže se seznámíme s knihovnou XNA, kterou využijeme pro implementaci zadaného algoritmu.

Ve třetí kapitole *Algoritmy zobrazování terénu* se podíváme na různé starší i současné algoritmy, které lze pro zobrazování terénu použít, detailněji si vysvětlíme princip algoritmu *Seamless Patches for GPU-based Terrain rendering* a objasníme si, co je na něm zajímavého a proč je důležité vyvíjet stále nové a modernější LOD algoritmy.

V další kapitole *Návrh knihovny* bude nejdříve popsán základní návrh funkčnosti výsledné knihovny a jejího zakomponování do vybraného rozhraní, který bude dále podrobně rozepsán. Jednotlivé součásti budou řešeny v samostatných podkapitolách.

V kapitole *Implementace knihovny* si popíšeme podrobnosti ohledně implementace jednotlivých částí knihovny. Detailně se zde zaměříme hlavně na kroky vedoucí k funkční implementaci algoritmu *Seamless Patches for GPU-based Terrain rendering*. Popíšeme si v ní také funkčnost algoritmu umožňujícího použití velkých výškových map, které by normálně nebylo možné v grafické paměti vytvořit. Stručně se také zmíníme o dalších komponentách, které byly využity v testovací aplikaci.

Kapitola *Dosažené výsledky* obsahuje statistické údaje týkající se výkonu naší implementace algoritmu a jejich zhodnocení. Také ukazuje, jak je možné naši knihovnu použít ve vlastním XNA projektu. Nakonec je uveden příklad jejího reálného využití.

2 Počítačová grafika

Termín počítačová grafika označuje součást informatiky, která se zabývá analýzou nebo syntézou obrazové informace [3]. Nás ovšem zajímá pouze druhá možnost, pod kterou si představíme proces převodu modelu scény vytvořeného v paměti počítače na grafický výstup zobrazitelný na monitoru. Scéna může být v paměti reprezentována různými způsoby (CSG, objemové modely, hraniční modely – viz [4]), ale současné běžně dostupné grafické karty se soustředí hlavně na hraniční polygonální modely. Ty bývají nejčastěji složené z trojúhelníků – tzv. *faces* (nejmodernější grafické karty však již dovolují například také hardwarově generovat trojúhelníkovou síť zadaných parametrů z řídicích bodů Bézierových křivek a ploch – viz [5] a [6]).

2.1 Aplikační rozhraní

Využití širokých hardwarových možností grafických karet dnes umožňují hlavně dvě aplikační rozhraní – OpenGL a Direct3D. V této podkapitole si krátce popíšeme jejich historii a vlastnosti a shrneme si aktuální situaci na trhu.

2.1.1 OpenGL

OpenGL se vyvinulo z IRIS GL od Silicon Graphics (SGI) a jeho první verze byla představena v roce 1992. Hned od začátku bylo zaměřeno na zpracování 3D geometrie a koncipováno jako otevřený standard. Přestože licenci vlastnila firma SGI, o jeho vývoj se starala OpenGL Architecture Review Board (ARB) skládající se z hardwarových i softwarových firem, která se čtyřikrát ročně scházela, aby prodiskutovala změny ve specifikaci OpenGL a budoucí plány. V roce 2006 však firma převedla kontrolu nad licencí z ARB na novou skupinu The Khronos Group, do které patří také většina členů původní ARB. The Khronos Group se o vývoj OpenGL stará i v současnosti.

OpenGL je nezávislé jak na programovacím jazyku, tak i na hardwarovém vybavení. Aby bylo možné dosáhnout nezávislosti na programovacím jazyku, jeho rozhraní je procedurální, jelikož ne všechny jazyky podporují objektově orientované programování. Nezávislosti na hardwarovém vybavení je dosaženo tím, že každá OpenGL operace může být vykonána buď ovladačem grafického adaptéru, nebo pokud ji nepodporuje, tak softwarově pomocí CPU. Většina funkcí OpenGL je určena buď k odesílání primitiv ke zpracování vykreslovacím řetězcem, nebo ke změně jeho stavu, který určuje, jak se budou tato primitiva zpracovávat. OpenGL je tedy stavový stroj – po nastavení určitého stavu se používá pro všechna další primitiva, dokud není opět změněn.

OpenGL podporuje rozšiřování své funkcionality pomocí tzv. extenzí. Výrobce grafické karty může do svého ovladače implementovat funkčnost, kterou ostatní grafické karty nemají, a vývojář ji pak může využít právě pomocí extenze. Díky tomu může OpenGL velmi rychle reagovat na vývoj nového hardwaru a podporovat všechny jeho dostupné prostředky. Nevýhodou tohoto přístupu je vnášení jisté nepřehlednosti kvůli velkému množství dostupných extenzí a jejich pojmenovávání – předpona jejich názvu totiž nejdříve obsahuje zkratku výrobce, který ji jako první implementoval. Když se později více výrobců shodne na implementaci dané extenze, dojde ke změně této zkratky na EXT, a pokud se na implementaci nakonec dohodne i rada ARB (dnes již Khronos), dojde k další změně předpony na ARB.

Základní informace o OpenGL lze najít v [7] a jeho podrobný popis v [8].

2.1.2 Direct3D

Direct3D je součástí knihovny DirectX zabývající se zpracováním 3D grafiky. DirectX je uzavřený standard, na jehož vývoj dohlíží pouze jediná firma – Microsoft. První verze DirectX (tehdy ještě nazvaná Game SDK) vznikla v roce 1994, aby bylo možné s její pomocí získat přímý přístup k hardwaru počítače, což jinak tehdejší Windows 95 příliš neumožňoval. Kvůli tomu Windows do té doby nebyl příliš lákavou platformou pro vývoj 3D aplikací. Microsoft začal na DirectX usilovně pracovat a přibližně každým rokem vydává novou vylepšenou verzi. Kompletní historie rozhraní DirectX je popsána v [9], [10] a [11].

Jelikož Direct3D neumožňuje rozšiřování funkčnosti pomocí extenzí, podpora nových vlastností grafických karet je vždy přidávána vydáním nové verze knihovny DirectX. Toto byla zpočátku nevýhoda, protože byl kvůli tomu Direct3D oproti OpenGL opožděn, avšak dnes již Microsoft spolupracuje velmi úzce s výrobci grafických karet, takže nové verze DirectX jsou většinou vydávány zároveň nebo dokonce ještě dříve než grafické karty, které nové technologie implementují.

Direct3D je na rozdíl od OpenGL objektově orientovaný a je zaměřen pouze na platformu Microsoft Windows, i když existují snahy o jeho implementaci i na Unixových systémech (viz [12]). Vykreslování primitiv a komunikace s grafickou kartou probíhá podobně jako v OpenGL s tím rozdílem, že v Direct3D je grafický adaptér a jeho aktuální stav reprezentován jako objekt a jeho vlastnosti. Dalším rozdílem je, že u starších verzí Direct3D nemáme zaručenu podporu všech funkcí – pokud chceme pracovat v hardwarově akcelerovaném režimu, musíme ověřovat, zda jsou námi používané funkce v hardwaru implementovány, kdežto OpenGL dokáže chybějící funkčnost softwarově emulovat na úrovni jednotlivých funkcí. Toto však již pro DirectX od verze 10 neplatí, místo toho jsou definovány minimální vlastnosti, které musí každý grafický adaptér kompatibilní s novější verzí DirectX podporovat. Rozhraní Direct3D je Microsoftem plně zdokumentováno v [13].

2.1.3 Aktuální situace

Momentálně je v oblasti počítačových her vedoucím rozhraním Direct3D. Za svou pozici vděčí jak velmi dobré taktice spolupráce Microsoftu s výrobci grafického hardwaru, tak i několika zaváháním ze strany vývojářů OpenGL. Díky tomu se DirectX ve verzi 8 stal velmi populárním, čímž získal větší podporu výrobců běžně dostupných grafických karet, což vedlo k jeho ještě větší oblíbenosti u vývojářů počítačových her. OpenGL je však díky jeho přenositelnosti stále velmi populární hlavně při vývoji nezávislých počítačových her a v profesionálních 3D aplikacích. Podrobněji je situace popsána v [14].

2.2 High-level knihovny

Aplikační rozhraní zmíněná v minulé podkapitole umožňují přístup ke všem možnostem grafických karet na nejnižší úrovni, což není vždy úplně pohodlné. Proto vznikla celá řada nadstaveb nad těmito knihovnami, které se snaží vývojářům zpříjemnit a zjednodušit práci při jejich používání. Většinou jsou objektově orientované, implementují určitý algoritmus pro práci s grafem scény a často jsou vydány pod některou z otevřených licencí. Navzájem se liší podporou různých platforem a grafických aplikačních rozhraní, mírou abstrakce jejich funkčnosti a vlastní filozofií této abstrakce (neboli návrhem objektů, jejich vzájemných vazeb a rozhraním pro jejich používání).

Některé nejznámější otevřené knihovny jsou:

- OpenSceneGraph – je multiplatformní, podporuje OpenGL od verze 1.1 po 3.0 včetně nejaktuálnějších extenzí, implementuje například algoritmy pro zvýšení výkonu (ořezání pohledovým tělesem či *level of detail*) a vícevláknové zpracování – viz [15]
- Open Inventor – je vytvořen firmou SGI, postaven na OpenGL, nezávislý na platformně, implementuje například model událostí pro interakci 3D objektů – viz [16]
- OGRE – podporuje různé platformy a Direct3D i OpenGL, má vestavěnou podporu pro mnoho grafických efektů, používá architekturu *pluginů* pro rozšíření jeho funkčnosti bez nutnosti úpravy jádra – viz [17]

Naše práce však bude využívat uzavřenou knihovnu XNA od společnosti Microsoft, která bude podrobněji popsána v následující podkapitole.

2.2.1 XNA Game Studio

XNA Game Studio je uzavřená sada nástrojů a knihoven zaměřených na snadný vývoj počítačových her. Pochází od firmy Microsoft a je jednotnou platformou pro vývoj aplikací pro Windows, Xbox 360 a dokonce Windows Phone 7. Knihovny XNA jsou určeny pro *managed* kód, .NET Framework a primárně pro jazyk C# (ale je možné využít i VB .NET). Jedná se o *high-level* zapouzdření funkčnosti grafického rozhraní DirectX 9 s mnoha funkcemi navíc pomáhajícími při práci s grafikou – například je velmi jednoduché načítání modelů a jejich následné vykreslení. Umožňuje však i využití *low-level* funkcí rozhraní DirectX, takže programátor má i přímý přístup k možnostem grafického adaptéru.

Základem každé aplikace je objekt třídy `Game` obsahující následující metody, které XNA automaticky volá a do kterých programátor vkládá veškerou logiku programu:

- `Initialize` – slouží pro vytvoření dalších potřebných objektů
- `LoadContent` – používá se pro načtení grafických objektů (modelů, textur, efektů)
- `UnloadContent` – slouží pro uvolnění grafických objektů z paměti
- `Update` – aktualizuje herní logiku (fyzikální simulace, pohyb hráče v prostředí)
- `Draw` – vykresluje aktuální snímek

Jakkoliv XNA usnadňuje tvorbu her, nejedná se o kompletní řešení – například vůbec nemá vestavěnou podporu pro graf scény. Umožňuje však jednoduché a logické rozdělování kódu na jednotlivé znovupoužitelné moduly pomocí komponent a služeb. Programátor může vytvořit modul jako objekt třídy `GameComponent` nebo `DrawableGameComponent` (v případě, že se jedná o objekt, který přímo něco vykresluje do scény), které obsahují stejné automaticky volané funkce jako objekt třídy `Game`. Tyto komponenty je pak možné použít v libovolné XNA aplikaci. Jakákoliv třída může dále implementovat libovolné rozhraní, které může nabízet jako tzv. službu – stačí, když se jen v objektu `Game` zaregistruje jako její poskytovatel. Jakákoliv komponenta pak může snadno najít poskytovatele této služby a využívat jej, aniž by byla nějak vázána na jeho konkrétní třídu. Takto lze velmi snadno v různých aplikacích měnit implementaci jednotlivých služeb, aniž by bylo třeba upravovat jakýkoliv kód komponent, které je využívají. Tento koncept je popsán v [18].

XNA také umožňuje programátorovi rozšiřovat formáty grafických objektů, které je schopno načítat a používat. K tomu slouží *content importers* a *content processors*, které zpracovávají objekty během vývoje a převádí je do formátu distribuovaného s XNA, a dále pak *content loaders*, které za běhu z tohoto souboru načítají data do použitelných objektů.

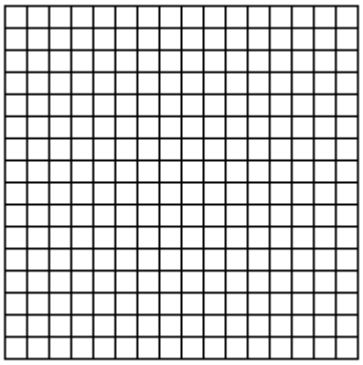
Více je možné se o XNA dočíst v [19], kde je postupováno od základů až k pokročilým technikám, nebo přímo v dokumentaci [20].

3 Algoritmy zobrazování terénu

Terén je nedílnou součástí mnoha počítačových her a 3D aplikací běžících v reálném čase, takže na jeho realistické a hlavně rychlé vykreslování je kladen hodně velký důraz. Hlavně ve hrách je velmi důležitý, jelikož jeho realističnost určuje schopnost hry vtáhnout hráče do děje a jeho tvar a členitost vymezují herní prostor a závisí na nich hráčova taktika.

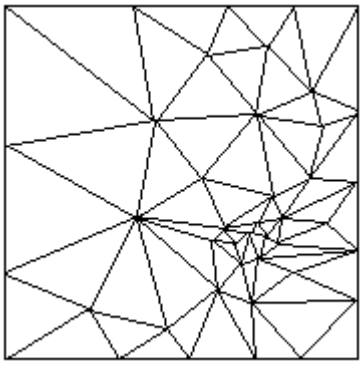
Při vykreslování povrchu terénu se pro jeho reprezentaci v paměti počítače využívají hlavně následující tři způsoby (viz [21]):

- Výšková mapa – množina bodů určujících výšku terénu v daném místě. Tyto jsou většinou 1:1 přiřazeny jednotlivým vrcholům pravidelné trojúhelníkové sítě tvořící povrch terénu. Tento způsob bývá nejnáročnější na paměťový prostor, ale zároveň je nejjednodušší, a proto se používá nejčastěji.



Obrázek 3.1: Síť pro výškovou mapu

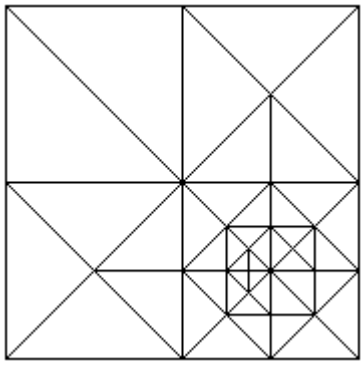
- Triangulated Irregular Network (TIN) – obecná nepravidelná síť trojúhelníků, která je vytvořená podle tvaru terénu tak, aby odpovídala určité chybové metrice. Díky tomu obsahuje pouze nejdůležitější body pro reprezentaci tvaru povrchu, takže většinou zabírá nejméně místa, ovšem práce s touto reprezentací je poměrně náročná.



Obrázek 3.2: Triangulated Irregular Network

- Right-triangulated Irregular Network (RTIN) – síť pravoúhlých trojúhelníků, jejichž hustota rozložení je nepravidelná tak, aby co nejvěrněji modelovaly daný povrch

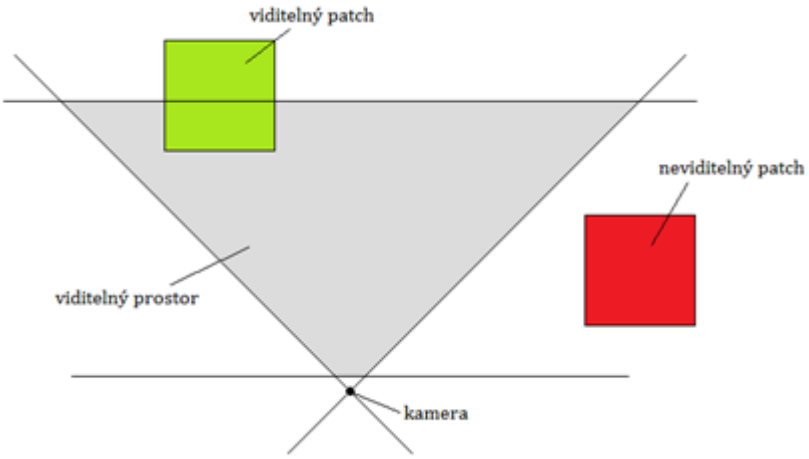
terénu. Tato metoda je kompromisem mezi dvěma předchozími. K její reprezentaci lze s výhodou využít stromových struktur, jako jsou binární a kvadrantové stromy.



Obrázek 3.3: Right-triangulated Irregular Network

3.1 Brute force

Nejjednodušším způsobem, jak vykreslit povrch terénu, je odeslat do grafické karty polygony přímo tak, jak je máme reprezentovány v paměti počítače. Díky současným grafickým kartám schopným vykreslit stovky milionů trojúhelníků za sekundu je tento způsob dnes doporučován pro počítačové hry, které nepotřebují zobrazovat obrovské terény. Tímto způsobem nevzniká žádná režie na CPU, který tak není nijak limitován ve výpočtech fyziky či umělé inteligence. Navíc rozdělením terénu na menší části (tzv. *patche*) lze velmi jednoduše vykreslovat jen ty, které jsou opravdu vidět, čímž ušetříme grafické kartě spoustu práce, viz obrázek 3.4 – tato metoda se nazývá ořezání pohledovým tělesem.



Obrázek 3.4: 2D ořezání pohledovým tělesem

3.2 Přehled LOD algoritmů

Pro aplikace vykreslující terén o velikosti miliónů kilometrů čtverečních, či dokonce celých planet, však ani dnešní grafické adaptéry nemají dostatečný výkon pro použití brute force. Proto se stále hojně využívají tzv. LOD (*level of detail*) algoritmy, které se snaží snížit počet vykreslovaných trojúhelníků s co nejmenším dopadem na výslednou kvalitu zobrazeného terénu.

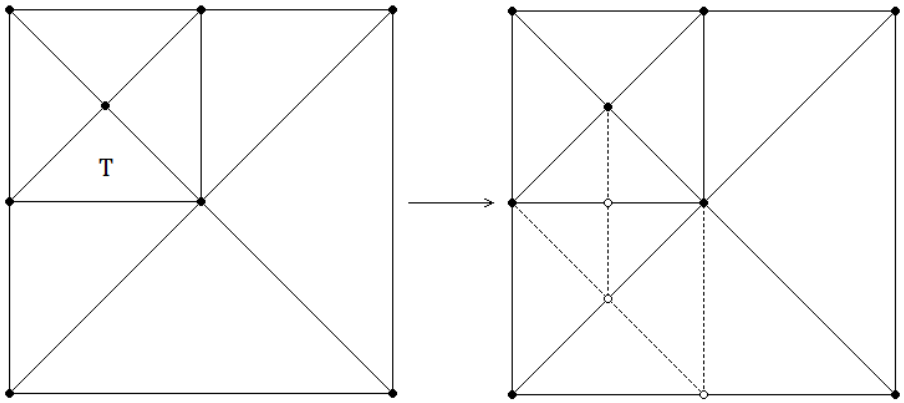
Pro dosažení optimálního výkonu je velmi důležité používat co nejaktuálnější algoritmy, které odpovídají stále se měnícím architekturám a výkonu současných grafických adaptérů. Zpočátku, kdy byly grafické karty málo výkonné, bylo cílem LOD algoritmů přesunout co nejvíce výpočtů na procesor a k vykreslení odeslat co nejmenší počet trojúhelníků i za cenu velkého zatížení CPU. V té době se LOD používal pro vykreslování terénů všech velikostí. Se vzrůstajícím výkonem grafického hardwaru je však výhodnější menší povrchy zobrazovat metodou brute force a LOD použít jen při vykreslování obrovských terénů. Tomu se postupně přizpůsobovaly i LOD algoritmy, které již většinou nepracují na úrovni jednotlivých trojúhelníků. Místo toho většinou používají stromové struktury pro rozdělení terénu na *patche*, které potom zpracovávají najednou. Zároveň se snaží co nejlépe využít schopností současného hardwaru, takže například odesílají geometrii v co největších blocích a uspořádanou tak, aby byla co nejlépe využita cache vrcholů. Některé se zaměřují na použití nových výpočetních jednotek, jako jsou *geometry shadery* nebo teselátory, takže se větší část algoritmu již vykonává přímo hardwarem grafické karty.

V této podkapitole si popíšeme nejpoužívanější LOD algoritmy a jejich vlastnosti.

3.2.1 ROAM 2.0

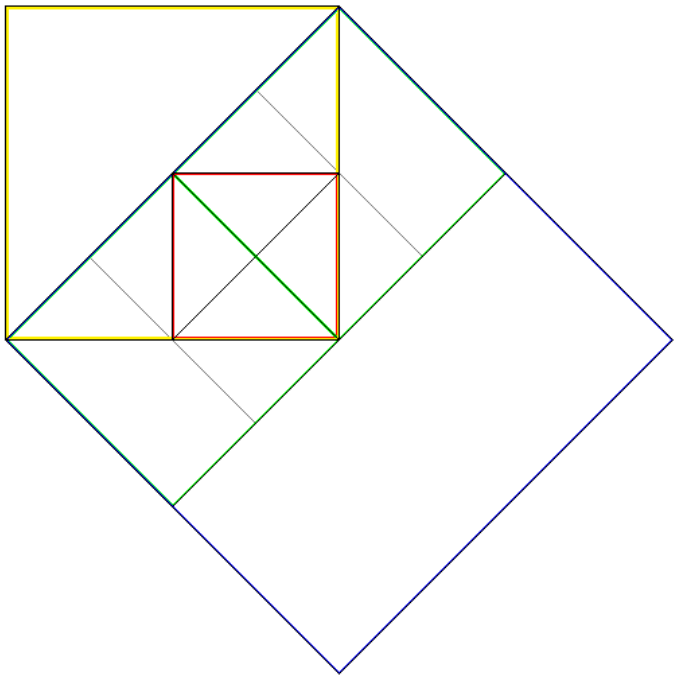
Tento algoritmus vychází z již poměrně zastaralého algoritmu ROAM (zkratka pro Real-time Optimally Adapting Meshes) z roku 1997, popsaného v [22]. Základní strukturou původního algoritmu je binární strom trojúhelníků. První úroveň detailu se většinou skládá ze dvou trojúhelníků tvořících čtverec a další úrovně jsou vytvářeny jejich rekurzívním rozdělováním (*splitting*) vždy na dva menší trojúhelníky. Výsledná geometrie povrchu vznikne selektivním rozdělením trojúhelníků podle určené chybové metriky, která vychází ze zakřivení terénu v daném místě a vzdálenosti trojúhelníku od pozorovatele. Aby nebylo nutné budovat celou hierarchii trojúhelníků v každém snímku znova, algoritmus využívá dvě prioritní fronty – *split queue* (pro rozdělování) a *merge queue* (pro slučování). Všechny trojúhelníky se vyskytují v jedné z front a podle vypočítaných priorit, které se mění hlavně s pohybem pozorovatele, jsou trojúhelníky buď dále rozdělovány nebo spojovány, čímž využijeme podobnosti povrchu mezi jednotlivými snímky. Určením limitu pro počet *split* a *merge* operací můžeme jednoduše kontrolovat výpočetní náročnost algoritmu.

Při rozdělování trojúhelníků je důležité, aby nedošlo k vytvoření tzv. T-spojů, což by způsobilo vznik trhliny v povrchu. Tomu lze snadno předcházet nuceným rozdělením sousedního trojúhelníku a v případě potřeby aplikováním tohoto pravidla rekurzívně na další trojúhelníky, což ilustruje obrázek 3.5.



Obrázek 3.5: Vlevo před a vpravo po rozdělení trojúhelníku T.

Druhá verze algoritmu popsaná v [23] využívá novou optimálnější hierarchickou strukturu – místo trojúhelníků zavádí tzv. *diamondy*. Provázanost struktur objasňuje obrázek 3.6. Uprostřed je červeně vyznačen hlavní *diamond* úrovně k, zeleně jsou označeni jeho předchůdci úrovně $k-1$, žlutě je zvýrazněn předchůdce úrovně $k-2$ a modře předchůdce úrovně $k-3$. Nejmenší šedé *diamondy* jsou jeho čtyři následníci úrovně $k+1$. Na obrázku není zobrazen celý strom *diamondů* ale jen ty, které jsou přímo navázané na červeně vyznačený – celý strom musí obsahovat ještě další *diamondy*, aby v povrchu nevznikaly praskliny. Výhodou je, že k jednoznačné identifikaci každého *diamondu* stačí znát pozici jeho středu, nemusíme tedy pro každý z nich ukládat pozice všech vrcholů – ty jdou velice snadno zjistit z pozic jeho předchůdců.

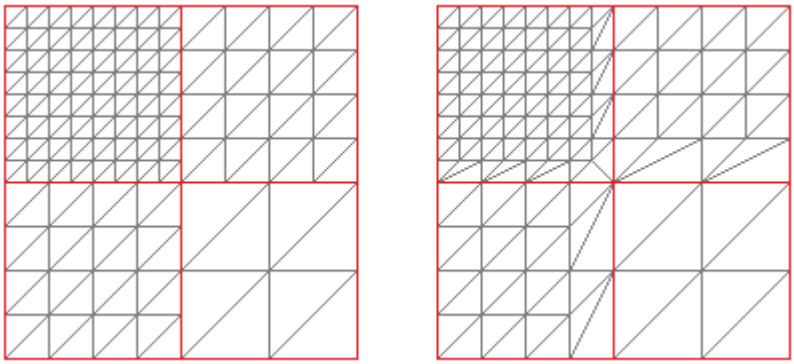


Obrázek 3.6: Provázanost struktur algoritmu ROAM 2.0

V této podobě má však algoritmus poměrně vysoké nároky na výpočet na procesoru a není příliš vhodný pro současné grafické karty, jelikož nevyužívá výhod dnešních architektur. Dalších vylepšení se ROAM dočkal v roce 2005 v článku [24], kde autor popisuje shlukování trojúhelníků pro vykreslování ve větších dávkách najednou, způsob načítání velkých dat z disku a indexování *diamondů* s pomocí Sierpinského křivky a zavádí další dvojici prioritních front pro textury, takže geometrie a textury jsou zobrazovány s různou úrovní detailů. Díky těmto inovacím je algoritmus stále použitelný pro zobrazování rozlehlých povrchů i na moderním hardwaru.

3.2.2 GeoMipMapping

Jak už jeho název napovídá, tento algoritmus je inspirován *mipmappingem* pro zobrazování textur. Během předzpracování je celý terén rozdělen na bloky o stejné velikosti, které musí mít na každé straně $2^n + 1$ vrcholů. Každému bloku je poté předpočítáno několik úrovní detailů jednoduchým způsobem – z přesnější úrovně je méně přesná vytvořena odebráním sudých vrcholů v obou směrech. Dále je vytvořen kvadrantový strom obalových kvádrů, jehož kořen obsahuje celý terén a pouze listy obsahují geometrii. Tento strom je použit pro ořezávání pohledovým tělesem. Při vykreslování viditelných listů je poté podle členitosti povrchu daného bloku a jeho vzdálenosti od pozorovatele vybrána úroveň detailů, která se vykreslí – zde je podobnost s klasickým *mipmappingem*.



Obrázek 3.7: Geometrie vzniklá algoritmem GeoMipMapping; vlevo před a vpravo po vyřešení děr v povrchu (červeně jsou vyznačeny jednotlivé bloky)

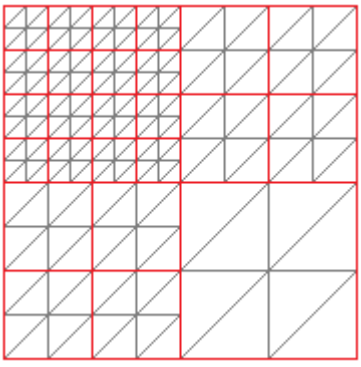
Tímto způsobem ovšem velmi snadno vznikají díry v povrchu na rozhraních jednotlivých bloků. GeoMipMapping tomu zabraňuje tak, že z bloku ve vyšším rozlišení vykreslí pouze vnitřní část a na jeho okraji, který se dotýká bloku s nižším rozlišením, vytvoří *triangle fany* (trojúhelníky sdílející jeden středový vrchol) přesně navazující na vrcholy sousedního bloku. Příklad geometrie vytvořené tímto algoritmem znázorňuje obrázek 3.7.

Aby nedocházelo k náhlému objevování a mizení vrcholů při změně úrovně detailů jednotlivých bloků, využívá se další analogie s texturami – trilineárního filtrování. Podle vzdálenosti od pozorovatele dochází k lineární interpolaci výšky vrcholů mezi dvěma úrovněmi detailů, takže ani při skokové změně geometrie není jejich odebrání či přidání patrné. Tato technika je nazývána *geomorphing* a využívá ji většina LOD algoritmů.

Algoritmus byl poprvé popsán v roce 2000 v [25]. V [26] je popsána jeho mírně upravená implementace v komerčním projektu spolu s datovými strukturami pro podporu velkých textur.

3.2.3 Chunked LOD

Podobně jako algoritmus GeoMipMapping vytváří při předzpracování kvadrantový strom s tím rozdílem, že každý uzel nyní obsahuje geometrii. V kořeni stromu je uložen povrch celého terénu ve velmi malém rozlišení, v jeho čtyřech potomcích se pak nachází geometrie vždy jedné čtvrtiny povrchu, avšak ve dvojnásobném rozlišení, atd. Každý uzel navíc ukládá obalový kvádr své geometrie, takže je strom využíván jak pro ořezávání pohledovým tělesem, tak i pro samotný výběr jednotlivých bloků pro vykreslování. Během procházení stromu je porovnávána předem spočítaná chyba geometrie daného uzlu s požadovanou přesností (podle vzdálenosti od pozorovatele), a pokud vyhovuje, je uzel vykreslen, jinak se rekurzívně procházejí jeho potomci.



Obrázek 3.8: Příklad geometrie vzniklé algoritmem Chunked LOD (červeně jsou vyznačeny jednotlivé bloky)

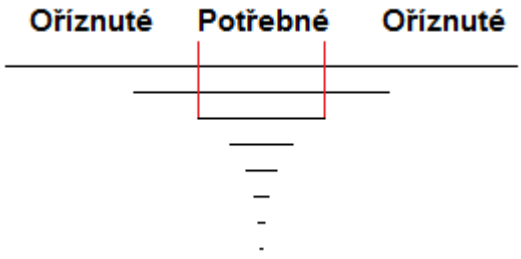
Během předzpracování stromu je každý blok komprimován tak, že odstraníme vrcholy, které jsou pro aktuální přesnost bloku zbytečné. Tímto jednoduchým opatřením lze u běžných povrchů docílit velkého snížení objemu dat a zvýšit rychlost zobrazování bloků. Bez použití komprese vypadá výsledná triangulace povrchu velmi podobně jako výsledek algoritmu GeoMipMapping, pouze má rozdílné rozložení jednotlivých bloků, viz obrázek 3.8.

Autor dále navrhuje několik možností, jak se vyhnout dírám v povrchu, z nichž doporučuje velmi jednoduchou metodu – každý blok má kolem svých okrajů svislou stěnu, která používá stejnou texturu jako samotná geometrie. Textura je tímto opatřením sice svisle protažená, což je ovšem v běžných aplikacích téměř nepozorovatelné, jelikož část vyplňující díru zabírá maximálně několik pixelů. Navíc díky tomu není nutné, aby jednotlivé bloky měly informace o použitém rozlišení sousedních bloků, a také se nemusí za běhu nijak upravovat jejich geometrie, což je velkou výhodou, neboť celý blok může být snadno vykreslen pouze jedním *Draw* voláním grafického API. Algoritmus ještě podporuje *geomorphing* s využitím *vertex shaderů* a načítání geometrických dat z disku pro jednotlivé bloky na vyžádání, čímž je možné jej využít pro zobrazování obrovských terénů, jejichž data nemohou být načtena v paměti najednou.

Algoritmus pochází z roku 2002 a je popsán v [27].

3.2.4 Geometry Clipmaps

Clipmapy byly popsány již v roce 1998 v [28]. Vychází z poznatku o struktuře *mipmap* pro textury několikrát větší než rozlišení monitoru. Pokud je textura více než dvojnásobně větší, než je rozlišení monitoru, pak při jejím zobrazování bude vždy použita buď jen část všech texelů (při pohledu zblízka), nebo se použijí texely z nižší úrovně *mipmapy* (při vzdálenějším pohledu), které se lineárně smíchají s texely z vyšší úrovně maximálně ve dvojnásobném rozsahu rozlišení obrazovky. Z toho vyplývá, že v žádný okamžik nepotřebujeme mít v grafické paměti žádnou úroveň *mipmap* větší, než je dvojnásobek aktuálního rozlišení obrazovky. Úrovně *mipmap* s vyššími rozměry tedy můžeme oříznout a ponechat si v paměti jen potřebnou část, viz obrázek 3.9.

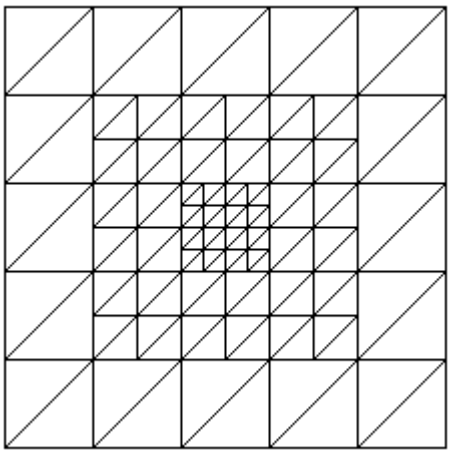


Obrázek 3.9: Princip *clipmap*

Při změně pohledu na texturu se potřebná část v oříznutých úrovních posouvá a do grafické paměti se díky využití toroidního adresování pomocí operace modulo načítají pouze nová chybějící data postupně od úrovní s nižším rozlišením nahoru. Tím můžeme docílit kontroly času stráveného v každém snímku načítáním nových dat. Při pomalém pohybu stačí aktualizovat pouze malou část dat, takže se v každém rámci stihnou načíst data pro všechny úrovně detailů, kdežto při rychlém pohybu můžeme omezit aktualizaci pouze na úrovně s nižším rozlišením, jelikož chybějící přesnost nebude kvůli rychlosti patrná. V roce 2007 byl v [29] představen způsob implementace *clipmap* v hardwaru grafické karty s využitím schopností rozhraní DirectX 10.

Clipmapy využil v roce 2004 autor dokumentu [30] pro vykreslování terénu. Celý povrch je v komprimované formě uložen v hlavní paměti a do grafické karty jsou posílány pouze části potřebné

pro aktuální pohled. Kolem pozice pozorovatele jsou vykreslovány regiony povrchu s postupně se snižující úrovní detailů. Při pohybu se tyto regiony posouvají s pozorovatelem a jejich výšková data jsou aktualizována podle potřeby stejně jako u textur. Vzniklou geometrii znázorňuje obrázek 3.10.

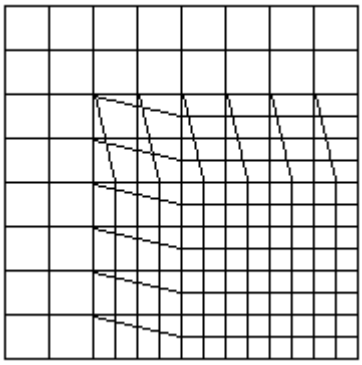


Obrázek 3.10: Geometrie povrchu algoritmu Geometry Clipmaps

Navržený algoritmus řeší díry v povrchu pomocí modifikace *geomorphingu* tak, že na okrajích regionů s vyšší úrovní detailů jsou vrcholy posunuty, aby lícovaly s trojúhelníky sousedního regionu. Dále využívá ořezávání pohledovým tělesem a procedurální generování výškových dat pro vysoké úrovně detailů. V [31] je popsána úprava algoritmu pro využití GPU pro většinu výpočtů, čímž se stal velmi vhodným pro současný hardware. [32] popisuje rozšíření algoritmu na kulová tělesa, takže je možné jej využít pro vykreslování celých planet.

3.2.5 CDLOD

Posledním a nejnovějším algoritmem v našem přehledu je Continuous Distance-dependent Level of Detail, který byl představen v roce 2009 v [33]. Vychází ze dvou předchozích algoritmů (tedy Geometry Clipmaps a Chunked LOD) a snaží se kombinovat jejich výhody a zároveň eliminovat nevýhody. Podobně jako Geometry Clipmaps používá rovnoměrné sítě a výškovou mapu aplikuje na vrcholy ve *vertex shaderu*, ale namísto vnořování sítí různých rozlišení vytváří kvadrantový strom, ve kterém každý uzel obsahuje síť se stejným rozlišením, ovšem pokrývající různou plochu podle úrovně uzlu ve stromu, podobně jako Chunked LOD. Autor díky tomu slibuje rovnoměrnější rozložení trojúhelníků na obrazovce a zároveň lze využít pokročilých metod moderních grafických karet, jako je hardwarový *instancing* geometrie [40]. Dírám v povrchu se tento algoritmus vyhýbá opět upravenou verzí *geomorphingu* – sudé vrcholy na okrajích uzlů s vyšší úrovní detailů jsou *vertex shaderem* posunuty nejen v ose Y, ale také v osách X a Z tak, že postupně splynou s trojúhelníky sousedního uzlu s nižší úrovní detailů (viz obrázek 3.11).



Obrázek 3.11: Přechod mezi dvěma úrovněmi detailů algoritmu CDLOD

3.3 Seamless Patches

Algoritmus, který jsme vybrali pro naši práci, pochází z roku 2007, kdy byl popsán v [34], takže je poměrně nový. Zavádí inovativní způsob dělení terénu na obdélníkové *patche* a trojúhelníkové díly a zaměřuje se na omezení množství přenášených dat po sběrnici do grafické karty a velké využití současných schopností GPU. Díky tomu jeho autoři odhadují zvýšení výkonu o 15 až 23% oproti Geometry Clipmaps a ROAM 2.0.

Tento algoritmus jsme si vybrali kvůli jeho předpokládanému velkému výkonu, elegantnímu řešení děr v povrchu terénu a využívání GPU, takže by jeho implementace měla držet krok i s vývojem budoucího grafického hardwaru. Podrobněji bude algoritmus popsán v následujících podkapitolách.

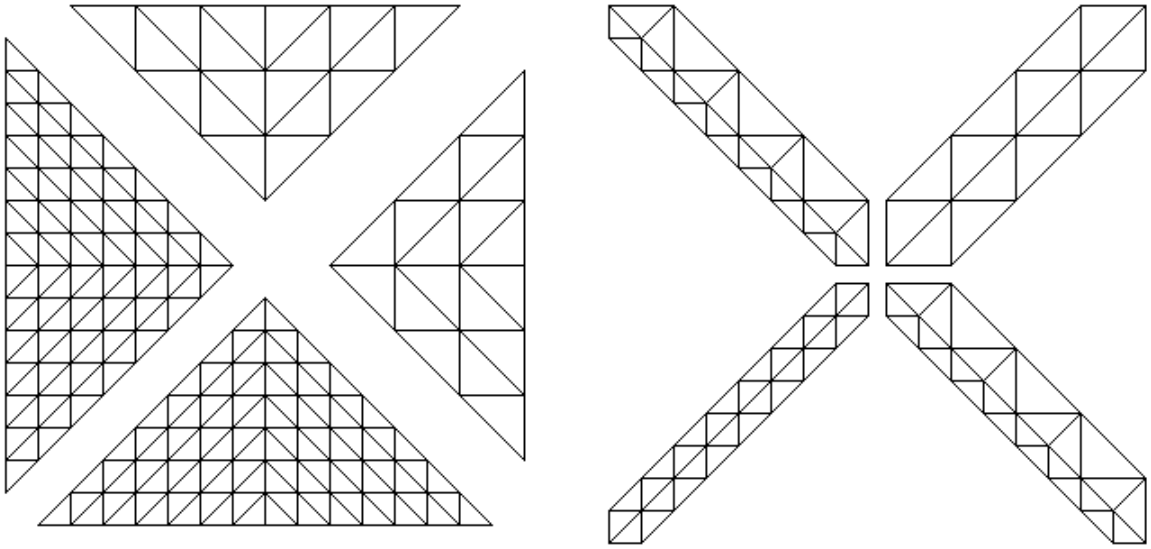
3.3.1 Struktura patchů

Patche jsou hierarchicky organizovány ve stromové struktuře, kde kořenový *patch* představuje celý terén. Každý *patch* se skládá ze čtyř trojúhelníkových dílů neboli dlaždic (*tiles*). Během vykreslování může mít každá dlaždice rozdílnou úroveň detailů, proto jsou jednotlivé díly spojeny pomocnými pruhy trojúhelníků (*strips*) tak, aby na jejich hranici nedocházelo k prasklinám v terénu. Obrázek 3.12 znázorňuje, jak může být tvořen jeden *patch* terénu ze čtyř dílů o dvou různých rozlišeních včetně odpovídajících spojovacích pruhů. Rozlišením dílu rozumíme počet trojúhelníků tvořících jeho nejdelší stranu (včetně trojúhelníků patřících spojovacím pruhům). Jednotlivá rozlišení dílů odpovídají po sobě jdoucím mocninám čísla 2.

Hierarchická struktura *patchů* je tvořena shora dolů rozdělováním jednotlivých *patchů* na R x R potomků, kde R je faktor větvení (*branching factor*). R je určeno podle počtu možných rozlišení trojúhelníkových dílů – je to poměr mezi největším a nejmenším povoleným rozlišením. Například pro 2 různá rozlišení platí $R = 2$ a pro 3 různá rozlišení je $R = 4$. Celá struktura je v paměti uložena implicitně (v souvislém poli bez použití ukazatelů), což zvyšuje efektivitu při jejím procházení.

Počet různých rozlišení dílů je často malý (2 až 4), takže je velmi jednoduché předem vygenerovat jak trojúhelníkové díly, tak i všechny možnosti pruhů pro jejich spojování. Tyto části povrchu mají jednotkovou velikost a jsou vytvořeny přímo v paměti grafické karty, z čehož vyplývá největší přednost algoritmu – během vykreslování terénu je použito hardwarového *instancingu* a čtení výškových dat z textur ve *vertex shaderu*, takže po sběrnici posíláme pouze informace o transformaci jednotlivých dlaždic k vykreslení a o jejich správné zobrazení se kompletně postará grafický adaptér.

Zároveň díky tomu, že jednotlivých částí tvořících *patche* bývá málo (pro dvě různá rozlišení to jsou dva díly a tři spojovací pruhy), můžeme v grafické paměti připravit všechny části přímo ve čtyřech různých orientacích, takže během vykreslování stačí přenášet jen jejich pozice a velikost, což velmi redukuje množství přenášených dat.



Obrázek 3.12: Ukázka dílů patche s dvěma úrovněmi detailů (vlevo) a jim odpovídající spojovací pruhy (vpravo)

3.3.2 Postup vykreslování terénu

Během vykreslování povrchu je v každém snímku procházena stromová struktura shora dolů od kořenového *patche*. Každý uzel obsahuje svůj obalový kvádr, který je využit pro ořezávání pohledovým tělesem. Pokud je *patch* viditelný, určíme pro každou jeho stranu požadované rozlišení z poměru její délky a vzdálenosti od pozorovatele. Pokud je tento poměr alespoň u jedné strany větší než 1, je tento *patch* rozdělen a dále se budou procházet jeho potomci. Jinak pro každou stranu *patche* vynásobíme spočítaný poměr maximálním rozlišením, výsledek zaokrouhlíme nahoru na nejbližší námi podporované rozlišení a do seznamu pro vykreslení přidáme díl v tomto rozlišení. Pro kompletní vykreslení *patche* ještě k takto vytvořeným dílům přidáme odpovídající spojovací pruhy podle jednotlivých výsledných rozlišení. Díky tomu, že použitá úroveň detailů jednotlivých dílů je určována z vlastností stran *patche* a faktor větvení odpovídá počtu použitých úrovní, je zajištěno, že sousední *patche* na sebe navazují bez vzniku prasklin v povrchu, což je pro zájemce v původním dokumentu [34] i formálně dokázáno. Navíc díky tomu žádný *patch* nemusí zjišťovat úroveň detailů sousedních *patchů*, což také dramaticky přispívá ke snížení využití procesoru při zpracování algoritmu.

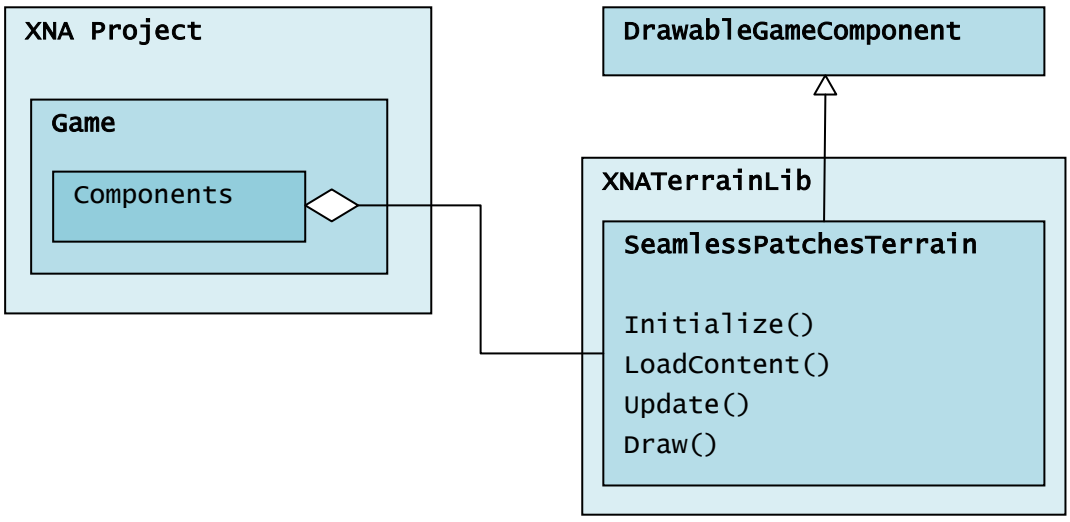
Tímto postupem je v každém snímku vytvořena rovinná síť trojúhelníků s úrovní detailů nepřímo úměrnou vzdálenosti od pozorovatele. *Vertex shader* poté přiřadí jednotlivým bodům správnou výšku načtením dat z výškové mapy podle jejich X a Z souřadnic a *pixel shader* může výsledný terén obarvit s využitím dalších textur či procedurálně.

4 Návrh knihovny

Cílem mé diplomové práce bylo stanoveno vytvoření DLL knihovny XNATerrainLib obsahující komponentu pro XNA Game Studio 4.0, která může být snadno připojena k libovolnému XNA projektu pro platformu Windows a použita pro vykreslování terénu metodou *Seamless Patches*. Tato knihovna bude testována v aplikaci, která by měla být schopna načítat standardní testovací data používaná v pracích zabývajících se vykreslováním terénu (výšková mapa oblasti Puget Sound – Pugetův záliv, o rozměrech 16385 x 16385 s rozlišením 10 metrů na pixel, viz [35]) a zobrazovat vygenerovaný terén v reálném čase.

4.1 Základní specifikace funkcionality

V první fázi bylo nutné navrhnout způsob, jakým by se měla knihovna integrovat do XNA projektů. Toto znázorňuje obrázek 4.1. Při inicializaci objektu `Game` je vytvořena instance třídy `SeamlessPatchesTerrain`, která je zděděná z třídy `DrawableGameComponent`, a poté je zaregistrována do seznamu komponent hry. Rozhraní XNA se poté samo postará o volání jejích jednotlivých funkcí. Nejdříve je volána funkce `Initialize()`, která v paměti vytvoří implicitní hierarchii *patchů* terénu podle zadaných parametrů – rozměrů výškové mapy a povolených rozlišení dlaždic. Poté jsou ve funkci `LoadContent()` připraveny *vertex* a *index buffery* s geometrickými daty pro dlaždice a spojovací pruhy. Následně jsou již pro každý snímek volány funkce `Update()` a `Draw()`. `Update()` aktualizuje potřebná data podle aktuální pozice pozorovatele a `Draw()` vybírá aktivní patche a rozlišení dílů pro vykreslení a poté všechny tyto části odesílá k zobrazení grafické kartě s využitím hardwarového *instancingu*.



Obrázek 4.1: Blokové schéma použití výsledné knihovny

4.2 Popis navrženého systému

Když byl připraven základní návrh, jak by měla knihovna fungovat, bylo jej třeba dále podrobněji rozvést.

Každý dobrý objektově orientovaný návrh bývá vytvářen s vědomím, že jej možná v budoucnu budeme chtít rozšířit o další funkcionalitu, takže ani naše knihovna by neměla být výjimkou. Proto nebude třída `SeamlessPatchesTerrain` zděděná přímo z třídy `DrawableGameComponent`, ale bude mezi nimi existovat ještě jedna úroveň abstrakce. Tou bude abstraktní třída `Terrain` obsahující základní vlastnosti jakéhokoliv terénu, takže časem bude možné do knihovny přidat další třídy pro vykreslování terénu jinými metodami než *Seamless Patches*. Základní vlastností této třídy bude `Size` určující velikost terénu v modelované scéně. Pro naše účely bude také vhodné, aby terén poskytoval určité statistické údaje pro měření výkonu, takže dalšími vlastnostmi budou `RenderedFaces` a `DrawCalls` – počet vykreslených trojúhelníků a počet volání vykreslovacích funkcí z XNA Frameworku během posledního snímku. Neméně vhodná pro nás bude možnost nechat si geometrii terénu vykreslit jako drátěný model – k tomu bude sloužit přepínač `UseWireframe`.

Pro vykreslení jakéhokoliv objektu 3D scény a tedy i našeho terénu, je dále nutné znát použité matice pro aktuální pohled pozorovatele a pro následnou projekci 3D prostoru na obrazovku monitoru. Algoritmy pro dynamickou změnu úrovně detailů objektů tyto informace navíc používají právě pro určení ideální úrovně detailů pro současný snímek. Tyto matice bývají většinou součástí třídy objektu představujícího virtuální kameru, proto jsem připravil rozhraní pojmenované `ICamera`, které bude snadno implementovatelné třídou pro kameru. Toto rozhraní obsahuje pouze několik důležitých vlastností – `Position` pro aktuální pozici kamery a tedy pozorovatele scény a `View` a `Projection` pro dvě již zmíněné matice. Pro usnadnění testování během implementace algoritmu bude třída `Terrain` obsahovat dvě vlastnosti pro objekty kamery – jeden bude použit pro výpočet úrovně detailů a druhý pro samotné vykreslování. Při použití dvou různých instancí kamery bude tedy možné pozorovat funkčnost algoritmu z jiného místa, než pro které se aktualizuje samotná geometrie terénu.

Samotný `SeamlessPatchesTerrain` bude také obsahovat statistické údaje specifické pro algoritmus, který bude používat, jmenovitě `TraversedPatches`, `CulledPatches` a `RenderedPatches` pro počet *patchů*, které algoritmus během posledního snímku v hierarchii prošel, kolik jich leží mimo aktuální pohled kamery a kolik jich bylo nakonec opravdu vykresleno. Dále bude třída obsahovat přepínač pro použití hardwarového *instancingu* během vykreslování a několik specifických vlastností terénu – minimální a maximální rozlišení jednotlivých částí a hodnotu faktoru větvení pro dané parametry.

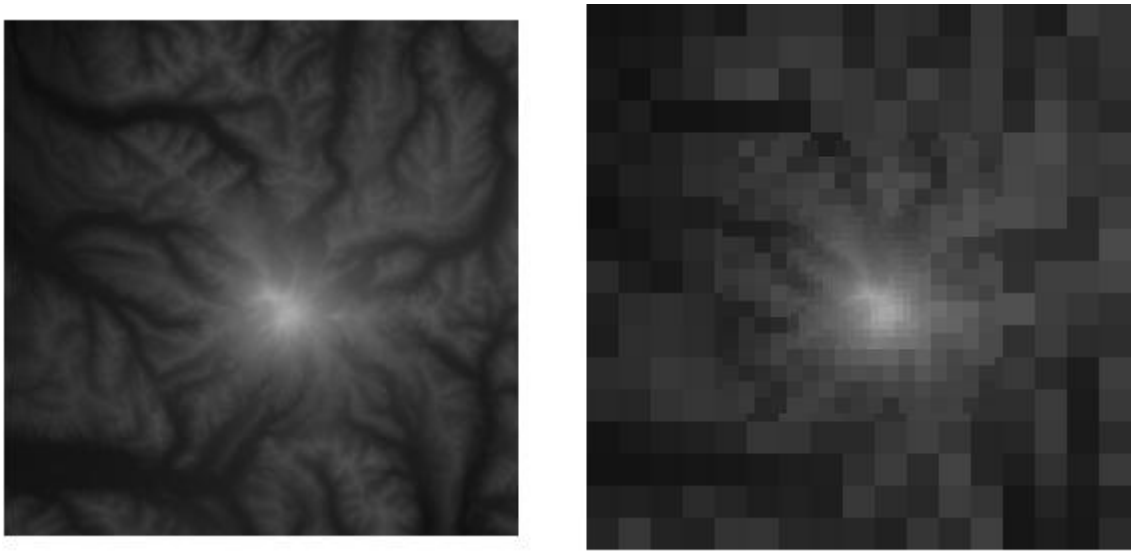
V následujících dvou podkapitolách bude popsán návrh propojení algoritmu s výškovými daty pro terén a způsob texturování výsledné geometrie.

4.2.1 Výšková data

Základní způsob, jak reprezentovat výšková data terénu, je použít výškovou mapu nejčastěji ve formátu obrázku v odstínech šedé, kde úroveň intenzity každého bodu udává výšku terénu v daném místě. Každému vrcholu geometrie terénu je poté přiřazen bod z dané výškové mapy a podle jeho intenzity je určena souřadnice vrcholu na výškové ose (nejčastěji na ose Y). Nevýhodou výškových map je, že přiřazováním výšky bodům na rovinné síti nelze vytvořit krajinné útvary, jako jsou skalní převisy či jeskyně. Pro jejich jednoduchost jsou však výškové mapy velmi rozšířené a tato nevýhoda se nejčastěji obchází přidáním těchto útvarů do scény ve formě samostatných modelů.

Námi vybraný algoritmus *Seamless Patches* je také navržen pro podporu výškových map, z nichž přiřazuje hodnoty bodů jednotlivým vrcholům geometrie přímo v hardwaru grafické karty. Aby byla tato funkčnost možná, je nutné v programu vytvořit z výškové mapy texturu, která je uložena v paměti grafického čipu. My však chceme, aby výsledný program zvládal zobrazovat terén s výškovou mapou o rozměrech 16385 x 16385 bodů a s přesností 16 bitů na jeden bod. Takováto výšková mapa by však v grafické paměti zabrala přes 512 MB prostoru, čehož by se sice dalo se současnými adaptéry dosáhnout, ale pro většinu grafických aplikací není přijatelné, aby více než polovinu paměti *high-endové* grafické karty zabrala pouze výšková mapa terénu – současné aplikace vyžadují nejvíce paměti pro kvalitní textury dalších objektů ve scéně. Navíc moderní způsoby vykreslování scén, jako je *deferred shading* [39], mají poměrně velké nároky na grafickou paměť samy o sobě, takže si programátoři nemohou dovolit využít tak velké množství paměti pouze pro terén – například ve hře Tabula Rasa je spotřebováno 100 MB paměti pouze pro textury, do kterých se vykreslují potřebné informace o scéně [36].

XNA Framework navíc nepodporuje textury větší než 4096 x 4096 bodů a také většina současného hardwaru podporuje textury o maximálním rozlišení 8192 x 8192, takže ani není možné tak velkou výškovou mapu v paměti vytvořit jako jednu texturu. Jedním z možných řešení by bylo výškovou mapu rozdělit na několik menších, které by se aplikovaly na různé části terénu, ale z hlediska využití paměti bude výhodnější implementovat do knihovny podporu pro *clipmapy*, jejichž základní princip je popsán v kapitole *3.2.4 Geometry Clipmaps*. Díky svým vlastnostem jsou ideálním kandidátem pro použití v kombinaci s algoritmem pro dynamickou úroveň detailů, protože nabízí přesně to, co potřebujeme – vysokou úroveň detailů v blízkosti pozorovatele, která se postupně se vzdáleností od něj snižuje. Tuto vlastnost ilustruje obrázek 4.2, který srovnává klasickou výškovou mapu s její reprezentací ve formě *clipmap* – pozice pozorovatele se nachází v jejím středu.



Obrázek 4.2: Srovnání klasické výškové mapy (vlevo) s *clipmapami* (vpravo)

Při použití *clipmap* obsadí celá jejich hierarchie v grafické paměti při ořezu o velikosti 2049 x 2049 bodů pouze 34,6 MB, což je 15krát méně než celá výšková mapa. Navíc je možné podle rozlišení obrazovky a požadované přesnosti zobrazení terénu velikost ořezu ještě snížit, čímž bychom dosáhli dalšího markantního uvolnění paměti – např. pro velikost ořezu 513 x 513 bodů zabere celá hierarchie *clipmap* pouze 3,2 MB v paměti grafické karty, což představuje pouze 0,6% původní velikosti.

Knihovna bude samozřejmě pro jejich jednoduchost a kvůli srovnání podporovat i klasické výškové mapy. Požadovanou reprezentaci výškových dat si uživatel vybere po vytvoření objektu typu `Terrain` zavoláním jedné ze dvou funkcí, buď `UseHeightmapHeight()` pro použití klasické výškové mapy nebo `UseClipmapsHeight()` pro hierarchii *clipmap*. Pokud nezavolá ani jednu z funkcí, bude terén vykreslen bez použití výškových dat – algoritmus bude sice upravovat úroveň detailů geometrie, ale celý povrch terénu bude plochý.

4.2.2 Manažer clipmap

Základem *clipmap* je hierarchie jednotlivých úrovní, z nichž ta nejvyšší má maximální možné rozlišení, a každá další nižší úroveň má rozlišení poloviční. V případě využití *clipmap* pro výškovou mapu terénu má každá úroveň rozlišení $2^n + 1$ bodů. V našem případě má tedy nejvyšší 14. úroveň rozlišení 16385 x 16385, které se postupně snižuje až do 0. úrovně, která má rozlišení 2 x 2. Celou tuto hierarchii budeme mít uloženou v operační paměti počítače, kde nám kvůli své struktuře zabere místo 512 MB rovnou 682 MB, díky čemuž ovšem velmi snížíme nároky na grafickou paměť. Paměťovou náročnost hierarchie *clipmap* je možné spočítat pomocí rovnice (4.1), kde M je výsledná paměťová náročnost, n je číslo nejvyšší úrovně a k je počet bytů na jeden bod.

$$M = \sum_{i=0}^{n} \left[k \cdot \left(2^i + 1\right)^2 \right] \tag{4.1}$$

Implementaci funkčnosti *clipmap* bude vhodné oddělit od samotného terénu, proto byla navržena třída `ClipMaps`, která se bude starat o vše, co s nimi souvisí. Tento manažer dostane při svém vytvoření cestu k souborům s daty pro jednotlivé *clipmapy*, nejvyšší požadovanou úroveň hierarchie a velikost ořezu. Při načítání dat ze souborů ve funkci `Load()` vytvoří pole objektů třídy `ClipMap` představující jednotlivé úrovně hierarchie. Dále bude třída `ClipMaps` obsahovat funkci `GetClipMap()` pro získání *clipmapy* požadované úrovně a `UpdateClipRegions()` pro aktualizaci dat v ořezových oblastech jednotlivých úrovní.

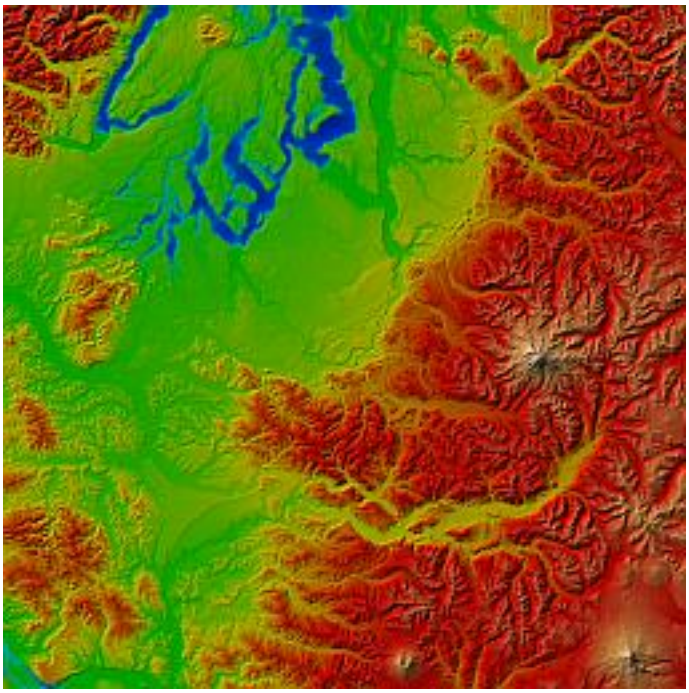
Každý objekt třídy `ClipMap` bude obsahovat objekt třídy `Texture2D` s aktuálními daty ořízntuté oblasti *clipmapy*, rozlišení dané úrovně a reálnou velikost, hranice a střed ořízntuté oblasti ve scéně. Dále bude také obsahovat funkce `Load()` a `UpdateClipRegions()`, které se budou volat pro všechny úrovně z nadřazeného objektu `ClipMaps`.

4.2.3 Možnosti texturování

Dalším krokem k vykreslení finálního terénu je namapování textury na vygenerovanou geometrii, aby se nejednalo pouze o jednobarevný model. Opět je samozřejmě možné si vybrat z několika různých technik.

Základním způsobem, jak otexturovat terén, je předem si vytvořit jednu velkou texturu, která odpovídá naší výškové mapě, a tu jednoduše namapovat na vygenerovanou plochu. Tento způsob bude v naší knihovně podporován zavoláním funkce `UseOneTexture()`. Je velmi jednoduchý na implementaci, ale má jednu zásadní nevýhodu. Tou je nutnost použití velmi velké textury, abychom dosáhli alespoň slušné úrovně detailů. Pokud však máme velký a detailní terén, tak při blízkém pohledu na něj pravděpodobně stejně nebude výsledek příliš kvalitní. Pro naši testovací aplikaci však bude tento způsob dostatečný, jelikož se nám jedná hlavně o implementaci LOD algoritmu. Textura,

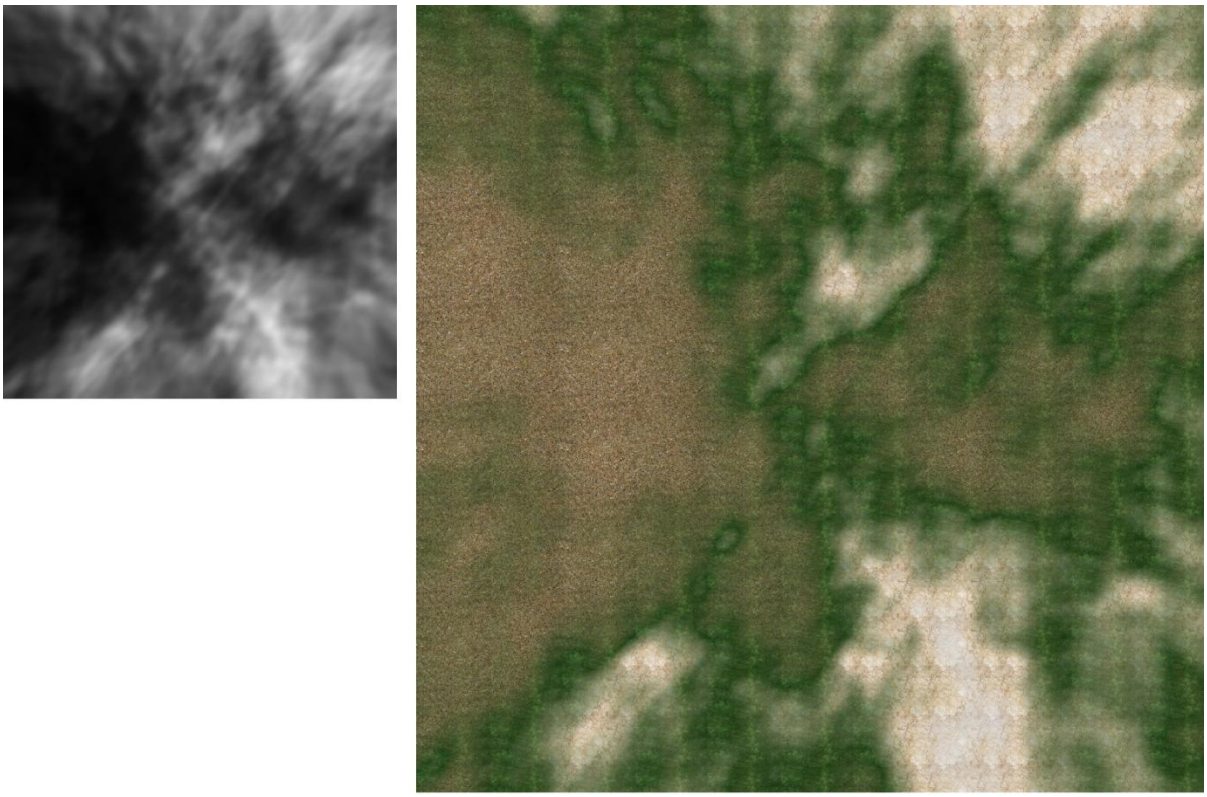
kterou použijeme, je přímo na míru vytvořená testovacímu modelu Puget Sound v rozlišení 4096 x 4096 obrazových bodů, což je pro naše testování přijatelné. Její zmenšeninu zobrazuje obrázek 4.3.



Obrázek 4.3: Textura pro testovací model Puget Sound

Takováto textura může být vytvořena buď ručně přímo v některém z kreslicích programů, což ovšem vyžaduje dobrou znalost ovládání daného programu, čas a v neposlední řadě také talent. Mnohem jednodušší je texturu vygenerovat procedurálně. Tato technika spočívá v tom, že si určíme, jakým materiálem bude tvořen povrch terénu v závislosti na jeho výšce a sklonu – například na dně jezera budou kamínky, ty budou na úrovni vodní hladiny přecházet v písek, potom v trávu, výš na kopcích ve štěrk a nakonec na vrcholcích hor bude sníh. Pro každý z materiálů použijeme relativně malou texturu, kterou je možné opakovat (její levý a pravý a horní a spodní okraj na sebe navazují), a podle určených kritérií je na základě údajů z výškové mapy smícháme, a vytvoříme tak výslednou texturu pro terén. Tento přístup využívá například volně dostupný software T2 – viz [37]. V současnosti lze tuto metodu implementovat také pomocí programovatelných *pixel shaderů* přímo pro hardware grafické karty, takže je možné v reálném čase dynamicky upravovat parametry jednotlivých materiálů a ihned vidět změny. Nevýhodou této techniky je menší kontrola nad vzhledem výsledné textury. Příklad jejího použití ilustrují obrázek 4.4 a obrázek 4.5.

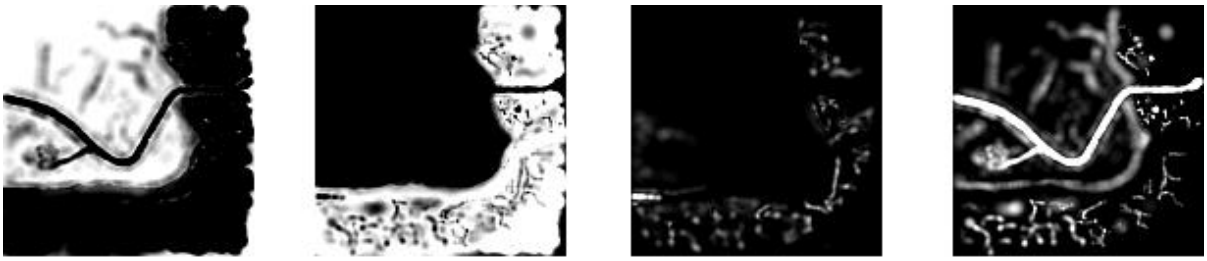
Obrázek 4.4: Textury pro jednotlivé materiály



Obrázek 4.5: Příklad výškové mapy a k ní procedurálně vygenerované textury

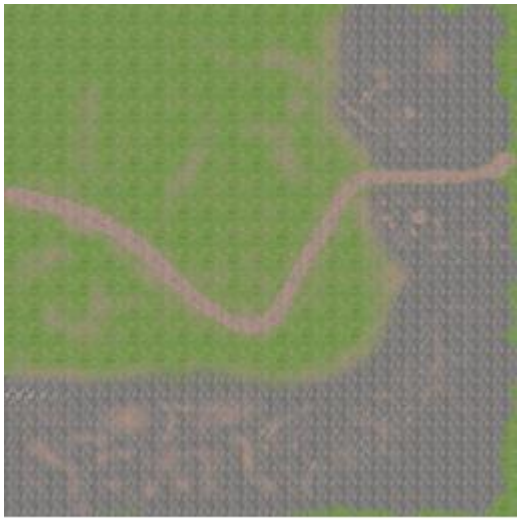
Namísto hodnot z výškové mapy je možné pro míchání textur jednotlivých materiálů použít vlastnoručně vytvořenou masku, kterou představuje obrázek se čtyřmi barevnými kanály RGBA, kde hodnota každého kanálu udává váhu pro jeden materiál v daném bodě. Tyto váhy jsou poté využity k lineární interpolaci barev mezi jednotlivými texturami. Tuto metodu používá například software EarthSculptor – viz [38]. Jelikož jsme tento program využili pro generování terénů do naší hry FireFighters pro soutěž ImagineCup 2011, která bude využívat naši knihovnu, bude tato metoda podporována přímo v *shaderech* knihovny zavoláním funkce `UseDetailTexture()`. Příklad textury vytvořené touto technikou znázorňují obrázek 4.6, obrázek 4.7 a obrázek 4.8.

Pokud uživatel nezvolí žádný způsob texturování terénu, budou jednotlivé části pouze jednoduše obarveny – v případě algoritmu *Seamless Patches* budou jednotlivé dlaždice bílé a spojovací pruhy červené. Toto obarvení nám také pomůže při testování správné funkčnosti algoritmu.



Obrázek 4.6: Jednotlivé RGBA složky masky (zleva)

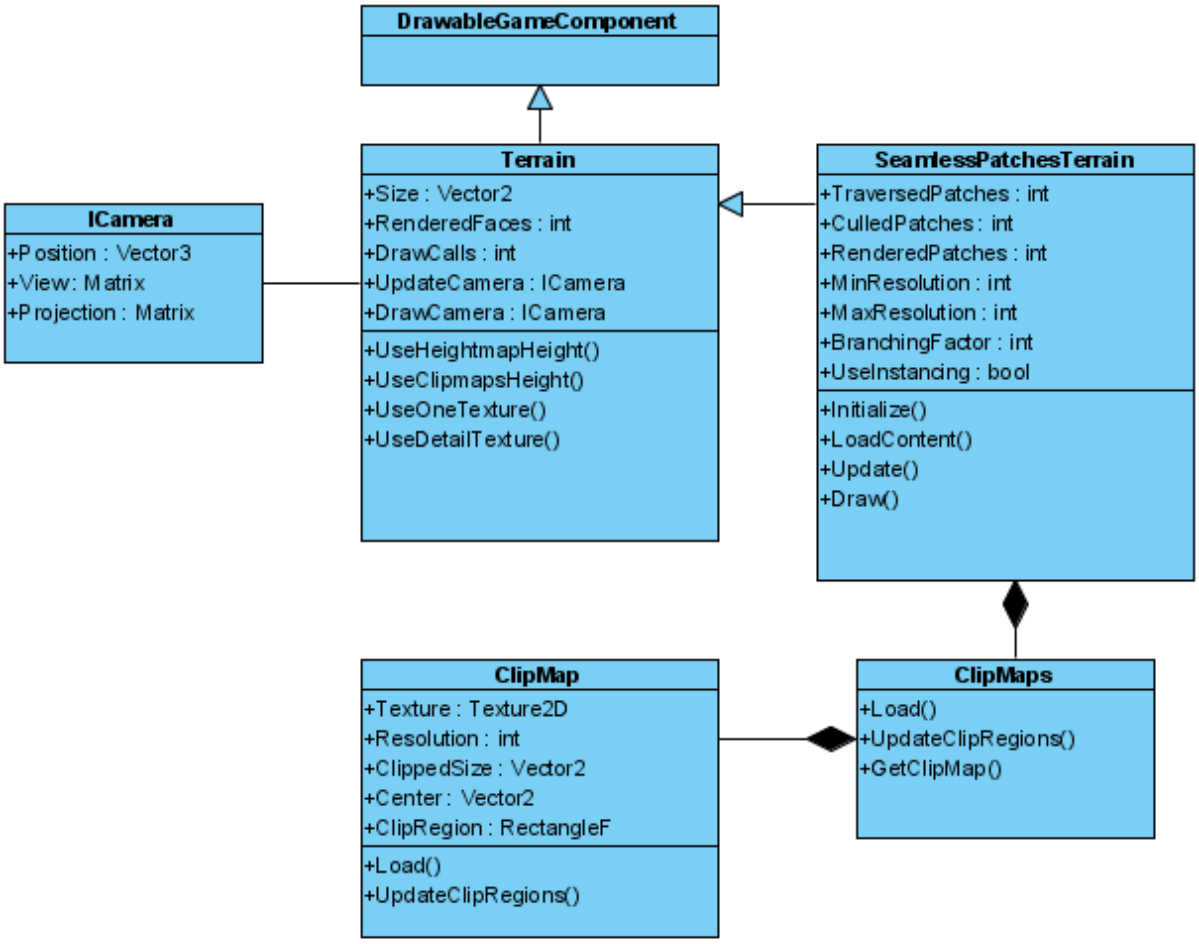
Obrázek 4.7: Textury materiálů odpovídající jednotlivým složkám masky



Obrázek 4.8: Výsledná textura vzniklá kombinací masky a textur materiálů

4.3 Model navrženého systému

Z předchozích částí byl sestaven výsledný návrh knihovny XNATerrainLib, jehož diagram tříd znázorňuje obrázek 4.9. Tento diagram zobrazuje pouze veřejné vlastnosti a funkce navrhovaných tříd, které budou přístupné uživateli z jeho vlastního projektu. Definuje tak pouze rozhraní vytvářené knihovny. Během implementace veškeré plánované funkcionality bude jistě nutné vytvořit ještě další množství privátních proměnných a funkcí hlavně v třídách `SeamlessPatchesTerrain` a `ClipMap`.



Obrázek 4.9: Diagram tříd výsledného návrhu knihovny

5 Implementace knihovny

Po dokončení návrhu modelu knihovny byla dalším krokem postupná implementace jednotlivých částí popsaných v předchozí kapitole. Jako programovací jazyk pro tuto knihovnu byl zvolen C#. Vývojovým prostředím bylo Visual Studio 2010, které mají studenti naší fakulty díky její účasti v programu MSDN Academic Alliance zdarma dostupné dokonce ve verzi Ultimate.

V následujících podkapitolách budou podrobně popsány kroky vedoucí k implementaci jednotlivých částí knihovny a dalších pomocných komponent, které vznikly během tvorby diplomové práce.

5.1 Seamless Patches

Implementace algoritmu *Seamless Patches* je hlavní náplní diplomové práce a také hlavní součástí vytvářené knihovny. Je implementován v třídě `SeamlessPatchesTerrain`, která je komponentou použitelnou v rozhraní XNA, podle jehož modelu implementuje čtyři základní kroky – inicializaci, přípravu grafických dat, aktualizaci a vykreslení.

Během inicializace jsou připraveny potřebné struktury jako seznamy částí pro vykreslení a hierarchie *patchů* v paměti. Poté jsou připravena všechna grafická data – jsou vytvořeny *vertex* a *index buffery* s geometrií částí terénu, načteny výškové mapy a textury pro povrch a vybrán efekt použitý při jejich vykreslování. Ve fázi aktualizace jsou pouze načítána chybějící data v jednotlivých *clipmapách*, pokud jsou použity. Při vykreslování se nejdříve prochází hierarchie *patchů* a podle aktuálních parametrů pohledu jsou naplňovány seznamy součástí pro vykreslení, které jsou následně odeslány ke zpracování grafické kartě buď klasicky, nebo s využitím hardwarového *instancingu*. Právě kvůli němu je nutné nejdříve vytvářet seznamy součástí, není možné jednotlivé části vykreslovat přímo během procházení hierarchické struktury.

Inicializace a vytváření grafických dat se provede pouze jednou při spuštění aplikace. Aktualizace dat a vykreslování poté probíhá v každém snímku a jejich časování je v režii XNA. Všechny fáze budou podrobně rozepsány v následujících podkapitolách.

5.1.1 Inicializace

Během inicializace jsou nejdříve z parametrů zadaných při vytváření objektu terénu vypočítány hodnoty některých důležitých vnitřních proměnných. Mezi ně patří hlavně faktor větvení, hloubka hierarchie *patchů* a jejich celkový počet, počet všech různých částí, pro které budou dále vytvářeny *vertex* a *index buffery*, nebo dvojkový logaritmus maximálního rozlišení. Poté je alokován paměťový prostor pro několik polí, v nichž budou uloženy informace důležité pro vykreslování jednotlivých částí – umístění prvního vrcholu a indexu v *bufferech* a počet trojúhelníků dané části. Dále je připraveno pole seznamů pro části, které mají být vykresleny – každý index v poli, a tedy každý seznam, odpovídá jedné části. Díky tomu bude možné velmi snadno připravit data nutná pro vykreslování jednotlivých částí metodou hardwarového *instancingu*.

Dalším krokem při inicializaci je příprava stromové hierarchie *patchů*. Každý *patch* je představován strukturou `Patch`, která obsahuje dvě položky – `Center` pro jeho střed a `Extent` představující polovinu jeho velikosti (vzdálenost od středu *patche* k jeho okraji). Kořenový *patch* má střed v počátku systému souřadnic a rozprostírá se přes celý terén, jeho velikost tedy přímo odpovídá

požadovaným rozměrům terénu. Každý *patch* v hierarchii má $N = R \times R$ potomků, kde R je faktor větvení. Hloubka D celého stromu lze spočítat podle rovnice (5.1), kde R_T je požadované maximální rozlišení terénu (rozlišení výškové mapy) a R_{max} je maximální rozlišení *patche*. Z hloubky můžeme dále pomocí rovnice (5.2) vypočítat celkový počet *patchů* ve stromu a alokovat pro ně pole vhodné velikosti. Celou stromovou strukturu budeme mít totiž uloženou implicitně v poli, takže každý *patch* s indexem i má rodiče s indexem $\left\lfloor \frac{i-1}{N} \right\rfloor$ a N potomků s indexy $(i \cdot N + 1..N)$. Během jejich inicializace rekurzívně projdeme celou hierarchii a každému *patchi* vypočítáme jeho střed a velikost.

$$D = \left\lceil \log_R \left(\frac{R_T}{R_{max}} \right) \right\rceil + 1 \tag{5.1}$$

$$C = \frac{N^D - 1}{N - 1} \tag{5.2}$$

5.1.2 Vytvoření grafických dat

Ve fázi vytváření grafických dat jsou načtena a vytvořena veškerá data, která jsou vyžadována grafickou kartou pro vykreslování objektů. Tato data jsou organizována pomocí *XNA Content Pipeline*. Pro každý typ dat (například 3D modely, efekty, textury, zvuky) obsahuje XNA třídy, které jsou schopny je během kompilace projektu převést do interního formátu souborů, ve kterém jsou spolu s výsledným projektem distribuovány. Za běhu aplikace je poté XNA schopno data z těchto souborů načíst a vytvořit z nich objekty použitelné přímo s hardwarem grafické karty. Pro uživatele je to velmi pohodlné, protože veškeré načítání obstará jeden řádek zdrojového kódu.

Nejdříve jsou podle zadaných parametrů načtena výšková data, tedy buď je vytvořena textura pro výškovou mapu, nebo objekt třídy `ClipMaps`, který se postará o načtení jednotlivých úrovní *clipmap*. Podobně jsou načteny veškeré textury podle vybrané možnosti texturování terénu.

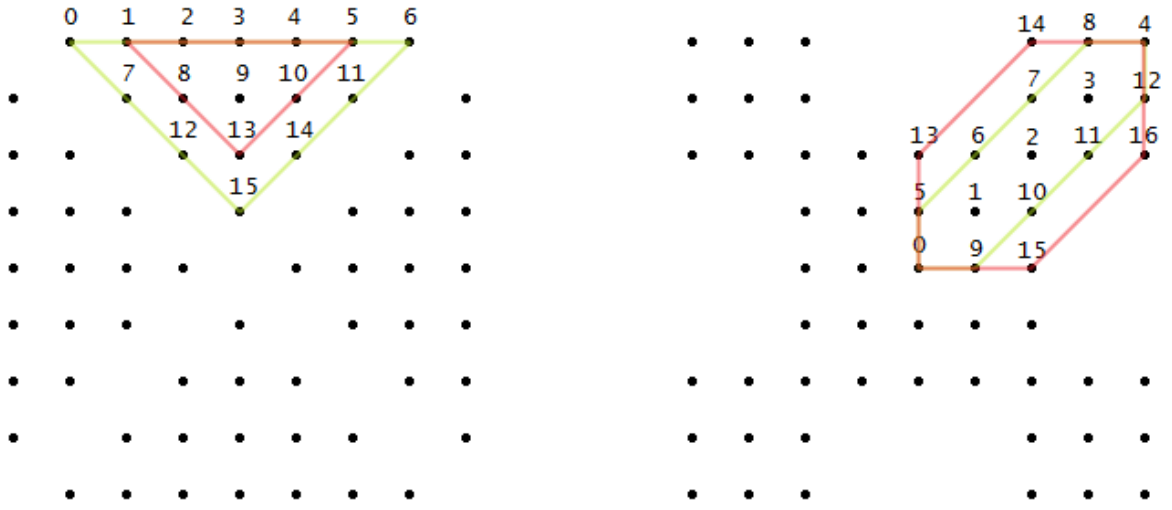
Velmi důležitou součástí je efekt, který bude použit pro vykreslování geometrie. Efekt je soubor psaný v jazyce HLSL a obsahující kód pro *vertex* a *pixel shadery*, které sdružuje do pojmenovaných technik. Každá technika určuje, který *vertex* a *pixel shader* bude použit pro vykreslení objektů. My ve *vertex shaderech* potřebujeme dvě různé funkce pro výpočet pozice vrcholů, pojmenované `Classic` a `Instanced` – pro klasické vykreslování a pro hardwarový *instancing*. Dále musíme mít tři různé funkce pro určení výšky vrcholů – `Flat` v případě, že se výšková data nepoužívají, `Heightmap` pro výškovou mapu a `Clipmaps` pro *clipmapy*. Nakonec ještě musíme vytvořit tři různé *pixel shadery* – `Color` pro jednoduché obarvení vrcholů, `Texture` pro použití jedné velké textury a `Detailmaps` pro použití techniky detail map s maskou. Ve výsledku tedy máme šest různých *vertex shaderů* a tři *pixel shadery*, které nakombinujeme do 18 různých technik. K tomu můžeme pro zjednodušení zdrojového kódu efektu s úspěchem použít makra preprocesoru, která HLSL podporuje podobně jako například C++. Název každé techniky bude kombinací názvů uvedených dříve, jako příklad můžeme uvést `FlatColorInstanced` či `HeightmapDetailmapsClassic`. Během samotného vykreslování poté podle zvolených vlastností terénu jenom velmi jednoduše vybereme požadovanou techniku podle jejího názvu.

Náš efekt je jediný grafický objekt, který budeme chtít distribuovat přímo s knihovnou, všechna ostatní data dodá její uživatel. Protože nutnost mít s DLL knihovnou ve stejném adresáři navíc další soubory není příliš atraktivní, připojíme náš efekt do projektu jako tzv. *embedded resource*, takže bude zkompilován přímo do DLL souboru [41]. Tato technika je ovšem vhodná pouze

pro malé soubory, jelikož při načtení aplikace nebo knihovny do paměti jsou zároveň s ní načteny i veškeré takto přiložené soubory.

Nejdůležitějšími grafickými daty jsou však pro nás informace potřebné k vykreslování samotné geometrie jednotlivých *patchů*. Ty jsou uloženy ve *vertex* a *index bufferech*. Tyto *buffery* jsou vytvořeny přímo v paměti grafické karty, jelikož nám stačí do nich vygenerovat veškerá důležitá data pouze jednou. *Vertex buffer* bude obsahovat pozice všech potřebných vrcholů a *index buffer* pořadí, v jakém se budou tyto vrcholy vykreslovat – vždy tři po sobě jdoucí vrcholy tvoří jeden trojúhelník.

Jak již bylo popsáno v kapitole *3.3*, každý *patch* se skládá ze čtyř trojúhelníkových dlaždic a čtyř spojovacích pruhů o různých rozlišeních. Nejdříve vygenerujeme vrcholy pro všechny tyto části do *vertex bufferu*. Stačí nám vytvořit body pro nejvyšší možné rozlišení jednotlivých částí a nižších rozlišení poté dosáhneme vhodným výběrem vrcholů pomocí hodnot v *index bufferu*. Jelikož jsou všechny trojúhelníkové části a všechny spojovací pruhy tvořeny stejně, mohli bychom vytvořit vrcholy pouze pro jeden trojúhelník a jeden pruh, ovšem náš algoritmus je zaměřen na co největší omezení přenosu dat do grafické karty během vykreslování, takže abychom ušetřili přenos informací o natočení jednotlivých částí, vytvoříme ve *vertex bufferu* vrcholy pro všechny čtyři orientace. Obrázek 5.1 znázorňuje příklad vygenerovaných vrcholů pro dvě různá rozlišení *patche* (4 a 8 trojúhelníků na stranu) s pořadím bodů pro jednotlivé části a tabulka 5.1 ilustruje výsledné rozložení dat ve *vertex bufferu*.



Obrázek 5.1: Vygenerované vrcholy pro trojúhelníkové díly (vlevo) a spojovací pruhy (vpravo). Zeleně jsou vyznačeny části v rozlišení 8 a červeně v rozlišení 4

Adresa	Typ části patche
0..15	Horní díl
16..31	Pravý díl
32..47	Spodní díl
48..63	Levý díl
64..80	Pravý horní pruh
81..97	Pravý spodní pruh
98..114	Levý spodní pruh
115..131	Levý horní pruh

Tabulka 5.1: Rozložení vrcholů jednotlivých částí ve *vertex bufferu*

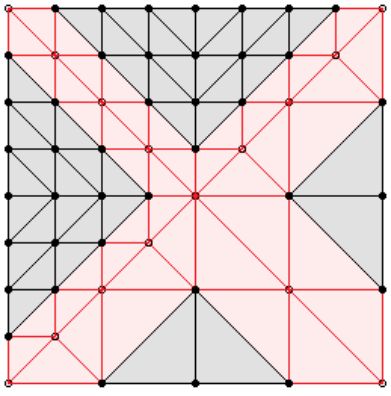
Také by bylo možné vygenerovat všechny vrcholy jednoduše jako čtvercovou síť zleva doprava a shora dolů a všechny části určit pouze hodnotami v *index bufferu*. Z důvodu použití 16 bitových indexů kvůli kompatibilitě s hardwarem jsme ale omezeni pouze na 65536 různých hodnot indexů, takže maximální dostupné rozlišení *patchů* by pro nás bylo pouze 128 trojúhelníků, jelikož pro rozlišení 256 bychom potřebovali 257 vrcholů na každé straně a na takový počet už by nám nestačily indexy. Kdežto pokud vytvoříme vrcholy pro každý díl zvlášť, můžeme indexy pro každou část číslovat znovu od 0 (všechny potřebné body budou ve *vertex bufferu* v řadě za sebou), čímž si zvýšíme maximální rozlišení patche na 512 trojúhelníků na každé straně – v takovém případě totiž bude mít každý díl akorát 65536 vrcholů.

Všechny indexy, určující jak budou z vytvořených vrcholů vykresleny jednotlivé části, jsou také uloženy pouze v jednom *index bufferu*. Díky tomu nemusíme během celého vykreslování měnit používané *vertex* a *index buffery*, čímž opět zvýšíme výkon knihovny. Na začátku in*dex bufferu* jsou vytvořeny indexy pro vykreslení trojúhelníkových dílů od největšího rozlišení po nejmenší a za nimi následují indexy pro jednotlivé rozlišení spojovacích pruhů. Ty jsou však určeny dvěma parametry – rozlišením dílu na každé straně spojovacího pruhu. Jejich celkový počet je tedy druhá mocnina počtu rozlišení. V našem příkladě pro dvě různá rozlišení máme tedy indexy pro dva typy trojúhelníkových dílů a čtyři typy spojovacích pruhů. Tabulka 5.2 ukazuje rozložení indexů v *index bufferu* pro náš příklad o dvou rozlišeních.

Adresa	**Typ částí patche**
0..53	Díl v rozlišení 8
54..59	Díl v rozlišení 4
60..101	Pruh pro rozlišení 8 a 8
102..137	Pruh pro rozlišení 8 a 4
138..173	Pruh pro rozlišení 4 a 8
174..191	Pruh pro rozlišení 4 a 4

Tabulka 5.2: Rozložení indexů jednotlivých částí v *index bufferu*

Během přípravy všech těchto dat si zároveň ukládáme adresy, na kterých v *bufferech* začínají vrcholy a indexy pro jednotlivé typy částí a jejich rozlišení. Při vykreslování určujeme, která část a v jakém rozlišení má být vykreslena, pouze pomocí změny těchto adres. Konkrétní příklad vykreslení *patche* v různých rozlišeních i s použitými adresami ukazuje obrázek 5.2.



Adresa vrcholů	Adresa indexů
0	0
64	102
16	54
81	174
32	54
98	138
48	0
115	60

Obrázek 5.2: Příklad vykreslení *patche* (vlevo) a použitých adres vrcholů a indexů (vpravo)

5.1.3 Aktualizace a vykreslení

Syntéza každého snímku, který je zobrazen na obrazovce, se skládá ze dvou částí. Nejdříve je aktualizována simulace virtuálního světa a poté je celá scéna vykreslena. Většina projektů využívá fázi aktualizace právě pro výpočet změn ve scéně za uplynulý čas (například pohyb objektů, rozhodování umělé inteligence, určení fyzikálních interakcí mezi objekty). Naše knihovna však žádné takovéto výpočty neprovádí a aktualizaci využívá pouze v případě použití *clipmap* pro případné načtení chybějících dat do grafické paměti. Tento krok bude popsán samostatně v kapitole *5.2 Clipmap*.

Nejpodstatnější část algoritmu probíhá při samotném vykreslování terénu. Nejdříve jsou vynulovány statistické údaje a smazány všechny seznamy vykreslovaných částí. Poté dojde k rekurzívnímu průchodu stromu *patchů* v závislosti na aktuálním pohledu kamery. Pro každý navštívený *patch* se nejdříve provede ořezání pohledovým tělesem, tedy určení, zda se vůbec nachází v pohledu kamery – pokud ne, jeho další zpracování je zastaveno. Jinak se pro každou stranu *patche* určí na základě jednoduché metriky požadované rozlišení. Pokud je toto rozlišení větší než maximální dostupné, dojde k rozdělení *patche* a rekurzívní průchod pokračuje jeho potomky. V opačném případě jsou do seznamů částí pro vykreslení přidány jednotlivé části tvořící daný *patch*. Index seznamu, do kterého bude daná část přiřazená, je určen z úrovně *clipmapy* použité pro její vykreslení (pokud *clipmapy* nejsou používány, je vždy nulová), natočení dané části, samotného typu části a požadovaného rozlišení. Do seznamu je přidán pouze index *patche*, ze kterého je možné určit pozici a velikost vykreslované části.

Metrika navržená v původním dokumentu popisujícím algoritmus *Seamless Patches* je velmi jednoduchá – plně ji popisuje rovnice (5.3). V té je l délka strany, pro kterou metriku právě počítáme, d je vzdálenost pozorovatele od této strany, ρ je faktor přesnosti a ε je výsledná metrika. Požadované rozlišení strany dostaneme vynásobením ε maximálním dostupným rozlišením *patche* a zaokrouhlením výsledku nahoru na nejbližší použitelné rozlišení. Na rozdíl od většiny metrik používaných v jiných LOD algoritmech tato neumožňuje přímo určit požadovanou chybu v pixelech. Jedinou kontrolu představuje parametr ρ - čím je větší, tím detailnější bude výsledný terén. Aby však bylo možné určit přesnou chybu v pixelech, bylo by nutné jeho hodnotu určit na základě nastaveného rozlišení, faktoru větvení, rozlišení obrazovky a velikosti úhlu pohledu kamery. Toto však v naší knihovně není implementováno.

$$\varepsilon = \rho \frac{l}{d} \tag{5.3}$$

Samotné vykreslení jednotlivých částí poté probíhá postupným procházením všech seznamů částí k vykreslení. Pro každý seznam jsou z jeho indexu zpětně určeny parametry části, která se má vykreslit. Podle nich jsou určeny adresy ve *vertex* a *index bufferu* pro potřebnou geometrii. Všechny části v seznamu jsou pak vykresleny pomocí stejného nastavení geometrie – jediné, co se mění, je pozice a velikost vykreslovaných částí. Při klasickém vykreslování je pro každou část volána jedna *Draw* funkce. Použijeme-li však hardwarový *instancing*, parametry všech částí ze seznamu jsou nejprve uloženy do pomocného dynamického *vertex bufferu*, a poté vykresleny všechny najednou pouze jedním *Draw* voláním. Geometrie je poté zpracovávána grafickou kartou několikrát, pokaždé s jinými parametry z dynamického *vertex bufferu*.

5.2 Clipmapy

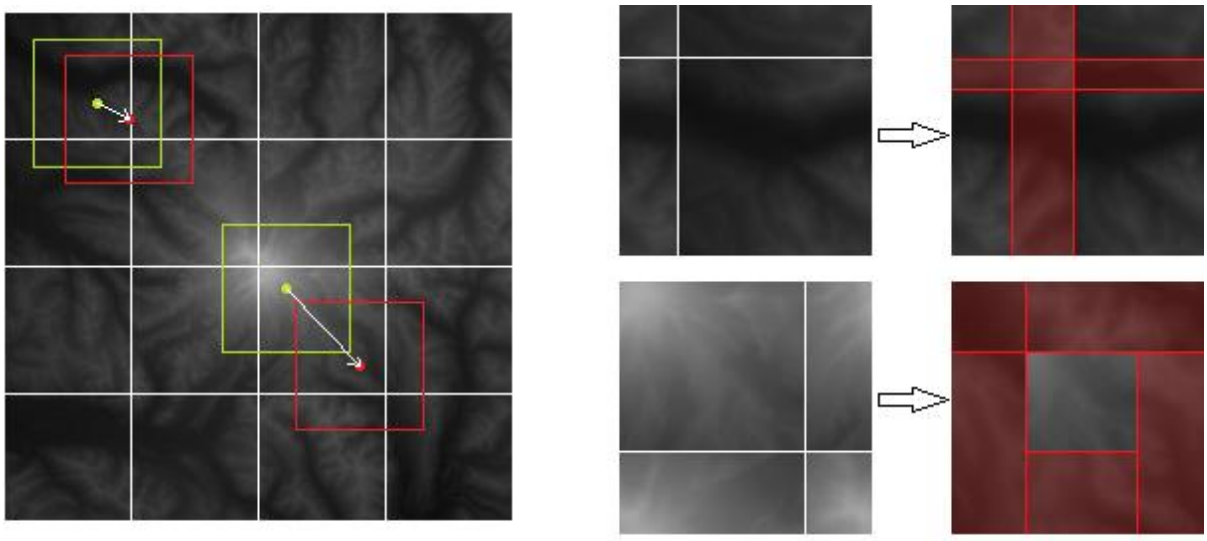
Podpora *clipmap* je druhou důležitou vlastností naší knihovny, protože nám umožňuje přenést paměťové nároky rozsáhlých terénů z grafické karty na operační paměť počítače, která je v současnosti levnější a lépe dostupná.

Celá hierarchie *clipmap* je uložena na pevném disku v jednom adresáři. Každá úroveň je představována jedním souborem v RAW formátu bez jakýchkoliv hlaviček. Soubor tedy obsahuje pouze přímo binární výšková data pro jednotlivé body. Na každý bod je vymezeno 16 bitů, což umožňuje zachytit výškový rozsah od nejhlubšího místa Země (Mariánský příkop, -10 924 m) po nejvyšší bod planety (Mount Everest, 8850 m) v rozlišení 30 cm. Parametrem třídy `ClipMaps` pro načítání souborů je proměnná `name` představující první část jejich názvu. Jméno každého souboru poté odpovídá vzoru `name_x.raw`, kde x je číslo úrovně ve struktuře *clipmap*. Tím je také určeno rozlišení daného souboru, které je $2^x + 1$. Jednotlivé soubory jsou načítány postupně od 0. úrovně po nejvyšší zadanou, v našem případě po 14. úroveň o rozlišení 16385 x 16385.

Během vytváření jednotlivých úrovní je pro každou z nich v grafické paměti vytvořena jedna textura v rozlišení dané úrovně, maximálně však o zadané velikosti ořezu. Například při ořezu o velikosti 2049 bodů budou textury přesně odpovídat rozlišení až po 11. úroveň. Pro všechny vyšší úrovně však bude rozlišení textur zadaných 2049 bodů bez ohledu na rozlišení úrovní. Aby bylo možné tyto textury použít ve *vertex shaderu*, musí být v některém z formátů s plovoucí řádovou čárkou. Z nabízených formátů v XNA nám nejlépe vyhovuje `HalfSingle`, ve kterém je každý bod textury reprezentován 16 bitovým číslem v plovoucí řádové čárce. Pro každou úroveň je alokováno pole, do kterého jsou načteny hodnoty všech bodů ze souboru. Tyto hodnoty jsou během načítání převedeny z rozsahu 0 – 65535 bez desetinných míst do rozsahu 0 – 1, se kterým se nám bude ve *vertex shaderu* lépe pracovat.

Po načtení všech dat jsou textury nižších úrovní, jejichž rozlišení je menší nebo stejné jako rozlišení ořezových textur, naplněny hodnotami jednotlivých bodů. Jelikož tyto textury obsahují data pro celou odpovídající úroveň, nebudou již muset být za běhu nijak upravovány. Textury vyšších úrovní však budou muset být v každém snímku aktualizovány tak, aby obsahovaly důležitá data v okolí aktuální pozice pozorovatele. Aby nemusela být pokaždé obnovena celá textura, využívají *clipmapy* toroidního adresování, kdy je pro každý z načtených bodů v dané úrovni určena adresa v textuře jednoduše pomocí operace modulo tak, aby odpovídala velikosti rozlišení textury. Obrázek 5.3 ilustruje rozložení dat v textuře a několik z případů, ke kterým může dojít během pohybu pozorovatele po povrchu terénu. Vlevo je zobrazena celá úroveň *clipmapy*, zelené čtverce znázorňují původní ořezané oblasti dat v textuře, červené pak oblasti po posunu pozorovatele. Vpravo jsou vidět konkrétní data v texturách před a po posunu. Červeně podbarvené regiony je nutné aktualizovat novými daty.

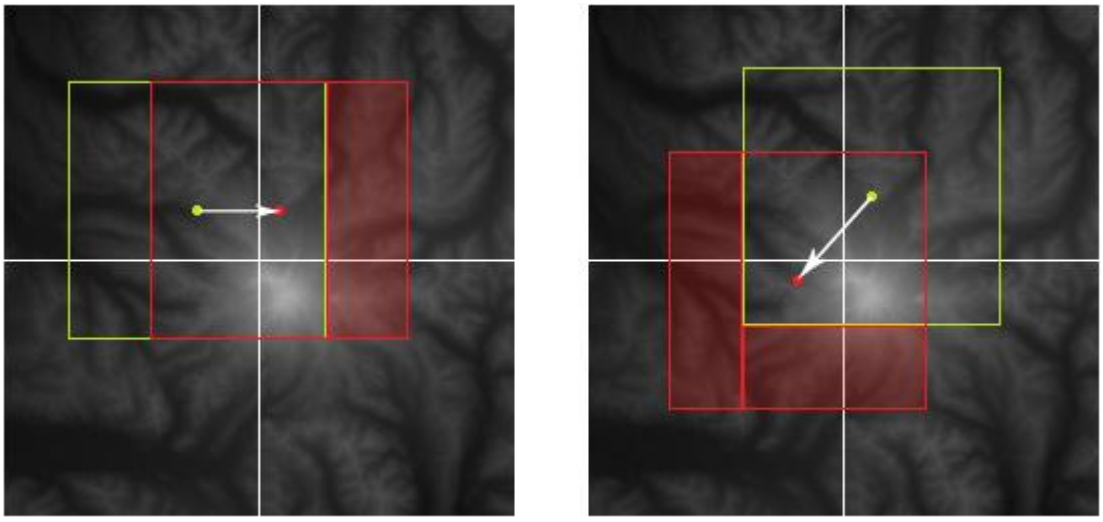
Pro aktualizaci dat v texturách je v XNA nutné použít funkci `SetData()`, které jako parametry předáme obdélníkovou oblast v textuře, kterou chceme nahradit novými daty, a pole samotných dat. Bohužel v XNA, na rozdíl od čistého DirectX, není možné přistupovat k datům v textuře přímo. Jedinou možností je využití právě funkce `SetData()` a nahradit tak celou obdélníkovou oblast aktualizovanými daty. Další nevýhodou XNA je, že si musíme data nejdříve připravit do samostatného pole, z kterého budou poté přenesena do textury v grafické paměti. Abychom pokaždé nealokovali nové pole, využíváme k tomuto účelu pouze jedno o velikosti celé textury (takže máme jistotu, že se nám do něj vždy vejdou veškerá potřebná data) sdílené mezi všemi úrovněmi *clipmap*.



Obrázek 5.3: Znázornění pohybu pozorovatele v rámci *clipmapy*, červeně jsou znázorněny oblasti, které je nutné aktualizovat

Samotná aktualizace probíhá tak, že se nejdříve ověří, zda se pozorovatel posunul tak daleko, že je nutné nahradit novými daty celou texturu, nebo stačí pouze částečná aktualizace. V prvním případě je situace jednodušší – celé sdílené pole se naplní novými daty, přičemž využíváme již zmíněné operace modulo, abychom všechny body připravili na správná místa, a poté je jedním voláním funkce `SetData()` celá textura aktualizována.

Druhý případ je o něco složitější. Nejprve podle směru pohybu pozorovatele určíme jednu nebo dvě obdélníkové oblasti, které je nutné nahrát do textury, což znázorňuje obrázek 5.4. Na tyto regiony je poté zavolána naše metoda `LoadRegionData()`. Ta zkontroluje, zda daná oblast protíná okraje textury, a v případě potřeby ji rozdělí na menší oblasti, které je protínat nebudou. Takové oblasti je již poté možné snadno aktualizovat přípravou jejich dat do pole a zavoláním funkce `SetData()`. Tímto postupem si zajistíme, že v jakémkoliv případě rozdělíme celou aktualizovanou oblast na co nejmenší počet obdélníkových regionů, které musíme odeslat do grafické paměti.



Obrázek 5.4: Vlevo je příklad aktualizace jednoho obdélníkového regionu, vpravo dvou

5.3 Pomocné komponenty

Po implementaci celé knihovny bylo ještě nutné vytvořit testovací aplikaci, na které by byla předvedena její funkčnost. K tomu ovšem bylo potřeba využít několika dalších komponent pro interakci s uživatelem a pro vygenerování hierarchie *clipmap* z jedné velké výškové mapy. Tyto součásti budou popsány v následujících podkapitolách.

5.3.1 Input, FPS, Camera

Tyto tři XNA komponenty slouží k interakci aplikace s uživatelem. Všechny byly vytvořeny během naší práce na hře FireFighters pro soutěž ImagineCup, v této práci tedy budou zmíněny jen jejich hlavní vlastnosti.

Komponenta `Input` umožňuje reagovat na vstupní zařízení počítače, jmenovitě klávesnici a myš. Vnitřně si udržuje informace o aktuálním stavu všech kláves, pozice myši a jejích tlačítek a o stavu z minulé aktualizace. Díky tomu dovoluje zjišťovat změny, ke kterým došlo. Těmi mohou být stisknutí nebo uvolnění některé klávesy nebo tlačítka myši či pohyb ukazatelem myši. Dále umožňuje zafixovat ukazatel myši na pevné pozici, což je v 3D aplikacích výhodné, protože uživatel není při pohybu omezen naražením ukazatele do okrajů pracovní plochy monitoru.

`FPS` je komponenta zobrazující statistické údaje o aktuálním počtu vykreslovaných snímků za sekundu. Funguje tak, že počítá uplynulý čas mezi jednotlivými voláními její vykreslovací funkce. Z něj poté vypočítá počet snímků vykreslených za sekundu, který zobrazí v levém horním rohu okna XNA aplikace.

Objekt `Camera` představuje virtuální kameru reagující na pohyb myši a stisk kláves, kterými je ovládán její pohyb ve scéně a směr pohledu. V každém snímku jsou podle těchto údajů aktualizovány její pohledová a projekční matice, které poté využívají další komponenty pro vykreslování svých modelů.

5.3.2 MipmapsGenerator

MipmapsGenerator je samostatný program bez grafického uživatelského rozhraní, který ze zadané výškové mapy vytvoří soubory pro jednotlivé úrovně kompletní hierarchie *clipmap*. Zdrojová výšková mapa musí být ve formátu RAW bez hlavičky s 16 bitovými hodnotami a její rozlišení v obou směrech musí odpovídat $2^n + 1$. Výsledné soubory jsou pojmenovány podle vzoru uvedeného v kapitole *5.2 Clipmapy*, takže je možné je rovnou použít v naší knihovně.

Program funguje tak, že postupně prochází hodnoty ve vstupním souboru a vytváří z nich výstupní soubor pro *clipmapu* o jednu úroveň nižší. Toho docílí velmi snadno přeskočením každé sudé hodnoty v řádku a každý sudý řádek vynechá úplně. Tím dojde ke snížení rozlišení na $2^{n-1} + 1$. Tento postup opakuje tolikrát, dokud nemá výsledná *clipmapa* rozlišení 2 x 2, což odpovídá nulté úrovni. Pro každé opakování použije vždy poslední vygenerovaný soubor. Díky proudovému zpracování dat není nutné celý soubor nejdříve načítat do operační paměti, takže nejsme omezení její velikostí, a můžeme tedy jako vstup použít libovolně velký soubor.

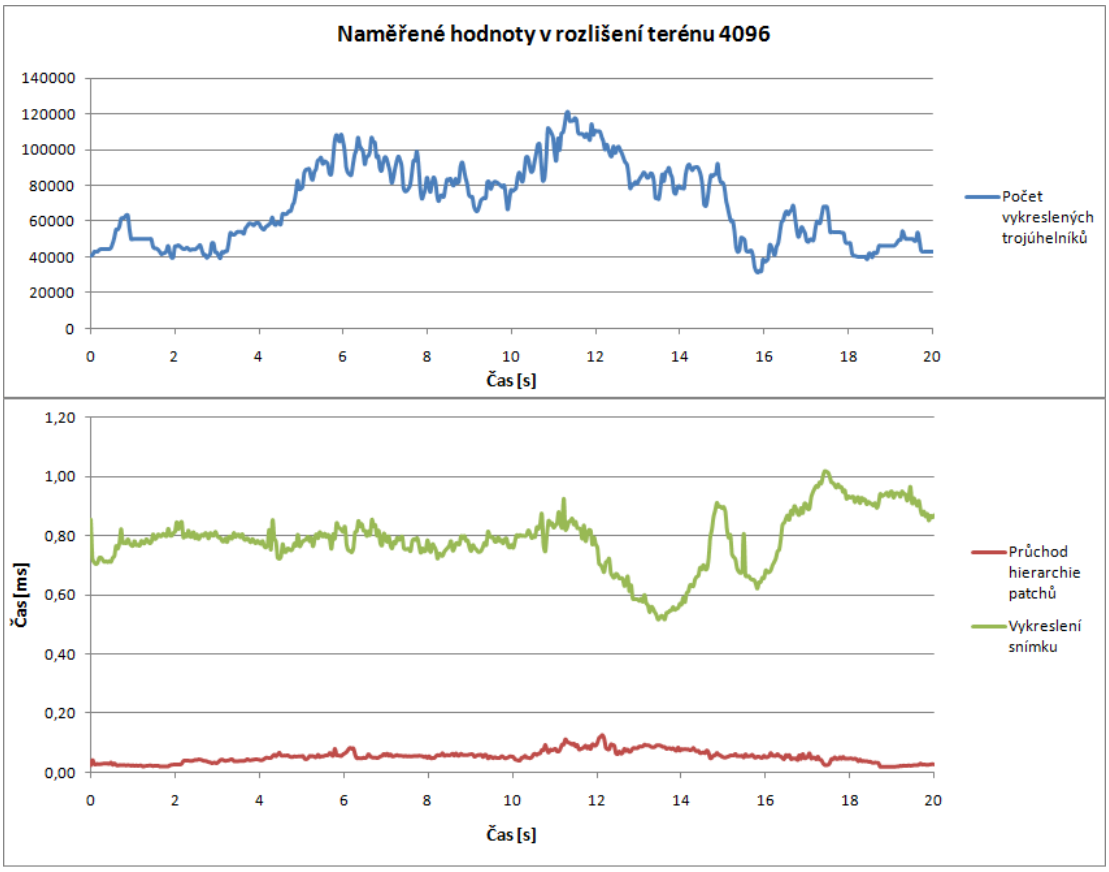
6 Dosažené výsledky

V této kapitole budou ukázány výsledky naší práce. V následující podkapitole budou ukázány a zhodnoceny naměřené údaje o rychlosti našeho algoritmu. Další podkapitola nás pak seznámí s konkrétním použitím naší knihovny v opravdovém XNA projektu.

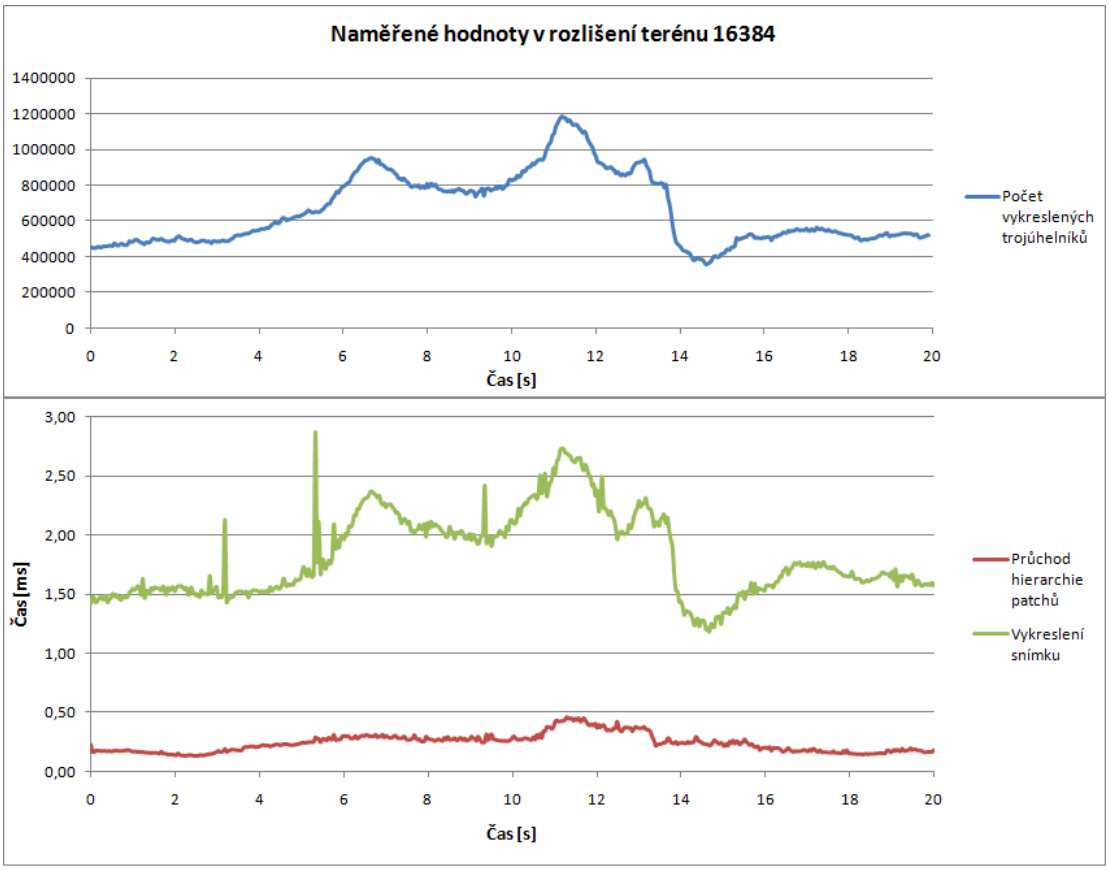
6.1 Rychlost algoritmu

Testovací aplikace používá naši knihovnu pro vykreslení terénu vytvořeného z výškových dat oblasti Puget Sound v maximálním rozlišení 16385 x 16385, jehož povrch je obarven jednou texturou o rozměrech 4096 x 4096 bodů. Pro měření byl použit 20 sekund trvající průlet nad tímto terénem. Všechny hodnoty byly naměřeny při vykreslování v okně o rozměrech 1024 x 768 na stolním počítači v konfiguraci Intel Core 2 Duo 3.0 GHz, 4 GB DDR2 RAM, GeForce 9600 GT 512MB.

V prvním případě jsme měřili čas, který algoritmus *Seamless Patches* potřebuje k projití celé hierarchie patchů a vytvoření seznamů částí pro vykreslení, a samotnou dobu vykreslování připravených částí v závislosti na počtu vykreslovaných trojúhelníků. Obrázek 6.1 a obrázek 6.2 ukazují naměřené hodnoty pro dvě různá rozlišení terénu. Z grafů je patrné, že zpracování našeho algoritmu je poměrně rychlé a nezabírá mnoho z celkového času vykreslení snímků, což je pro LOD velmi důležitá vlastnost. Z druhého grafu dokonce vyplývá, že při velkém rozlišení terénu jsme opravdu omezeni pouze výkonem grafické karty, nikoliv CPU, což je patrné z toho, jak křivka času vykreslení snímku přesně kopíruje křivku počtu vykreslených trojúhelníků.



Obrázek 6.1: Naměřené hodnoty v rozlišení terénu 4096



Obrázek 6.2: Naměřené hodnoty v rozlišení terénu 16384

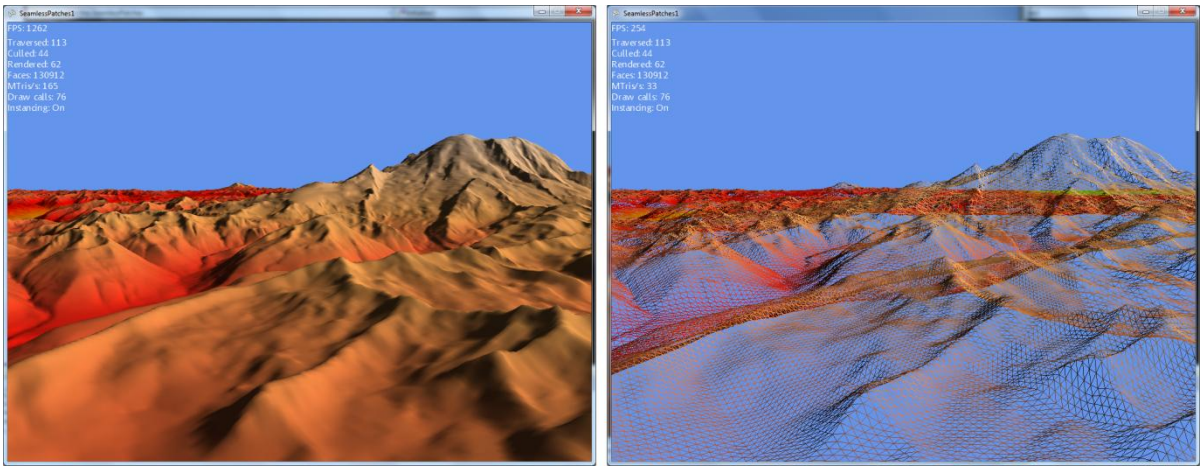
Druhým měřením jsme chtěli porovnat výkon při použití hardwarového *instancingu* oproti běžnému vykreslování. Tentokrát jsme měřili průměrný počet *Draw* volání a vykreslených trojúhelníků v každém snímku a průměrný čas potřebný k vykreslení jednoho snímku. Rozlišení terénu bylo v obou případech 16384 trojúhelníků na stranu, ale použili jsme dvě různé hodnoty faktoru přesnosti. Naměřené hodnoty ukazuje tabulka 6.1. Z měření vyplývá, že hardwarový *instancing* velmi redukuje počet *Draw* volání, což snižuje zatížení procesoru při odesílání dat ke zpracování grafické kartě. Pokud je tedy naše aplikace omezena výkonem CPU, může se vyplatit jej využít. Dále je z údajů patrné, že je vhodné jej použít při velkých počtech vykreslovaných trojúhelníků – v tom případě dosahuje zrychlení téměř 37%, na rozdíl od necelých 4% při vykreslování přibližně desetinového množství trojúhelníků, což se odvíjí právě od velkého rozdílu v počtu *Draw* volání.

Způsob vykreslování	**Faktor přesnosti**	**Průměrný počet Draw volání**	**Průměrný počet trojúhelníků**	**Průměrný čas na jeden snímek [ms]**
Instancing	4	46	656 702	2,079
Klasický	4	1839	650 459	2,819
Instancing	1	43	68 562	0,926
Klasický	1	250	67 971	0,961

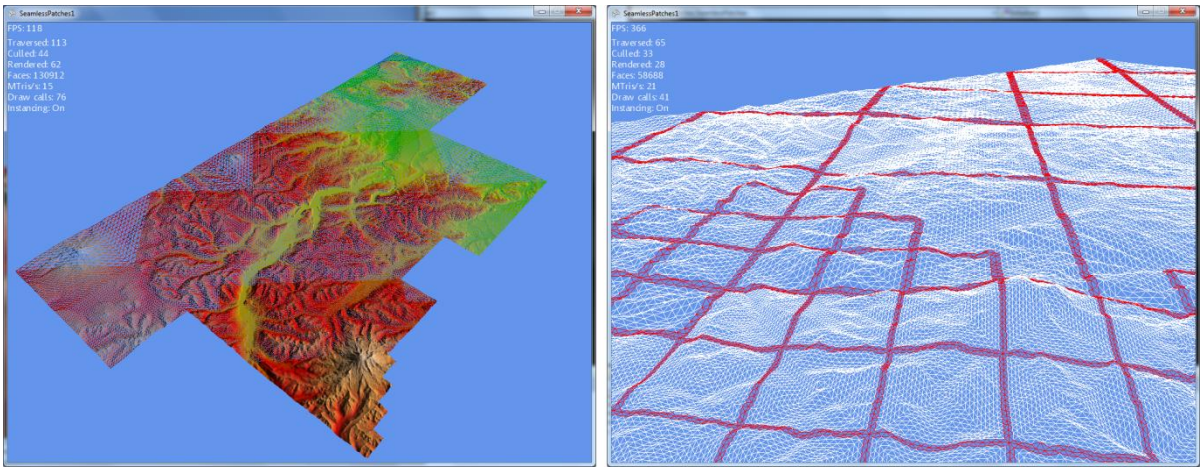
Tabulka 6.1: Porovnání vykreslování klasickou metodou a pomocí hardwarového *instancingu*

6.2 Ukázky z testovací aplikace

Obrázek 6.3 ukazuje vykreslený povrch terénu a jemu odpovídající drátěný model v testovací aplikaci. Funkčnost ořezávání pohledovým tělesem a vytvořenou stromovou strukturu *patchů* pro povrch terénu pak ilustruje obrázek 6.4.



Obrázek 6.3: Ukázka povrchu terénu (vlevo) a jeho drátěného modelu (vpravo)



Obrázek 6.4: Ukázka ořezávání pohledovým tělesem (vlevo) a hierarchie *patchů* tvořících terén (vpravo)

6.3 Použití knihovny

Naši knihovnu je díky jejímu návrhu velmi jednoduché použít v libovolném XNA projektu. Představme si případ, kdy chceme v aplikaci vykreslovat terén o šířce 256, délce 256 a výšce 10 jednotek v maximálním bodě. Terén by měl podporovat maximální rozlišení 4096 trojúhelníků na stranu. Nejmenší rozlišení pro jeden *patch* bychom chtěli mít 16 trojúhelníků a největší 64, jedná se tedy o 3 úrovně rozlišení. Výšková data chceme načítat ze souborů *clipmap*, které máme připravené v podsložce Data, a jmenují se ps_height_x.raw. Velikost ořezu *clipmap* bychom chtěli mít 513 bodů. Pro texturování máme připravenou jednu texturu s názvem ps_texture_4k. Vykreslovat jej budeme pomocí hardwarového *instancingu* s faktorem přesnosti rovným hodnotě 2. Toto všechno nám zajistí následujících několik řádků zdrojového kódu zavolaného v inicializaci objektu třídy `Game` (kód předpokládá existenci objektu `camera` implementujícího rozhraní `ICamera`):

```
SeamlessPatchesTerrain terrain =
  new SeamlessPatchesTerrain(this, camera, new Vector3(256, 10, 256),
                             4096, 16, 3);
terrain.UseClipmapsHeight("/Data/ps_height", 513);
terrain.UseOneTexture("ps_texture_4k");
terrain.UseInstancing = true;
terrain.Precision = 2.0f;
this.Components.Add(terrain);
```

Knihovna byla reálně použita ve hře FireFighters: Whatever It Takes vytvořené pro soutěž ImagineCup 2011 v kategorii Game Design. Vzhled terénu ve hře ilustruje obrázek 6.5.

Obrázek 6.5: Ukázka použití knihovny ve hře FireFighters: Whatever It Takes

7 Závěr

Tato diplomová práce si kladla za cíl seznámit čtenáře s několika grafickými knihovnami a s některými dnes používanými *level-of-detail* algoritmy pro zobrazování terénu v počítačové grafice. Možnost vykreslování obrovských terénů v reálném čase je dnes velmi důležitá jak pro letecké simulátory pro výcvik pilotů, tak i pro počítačové hry, kde je nutné hráče vtáhnout do děje co největším realismem obrazu.

Dále byla vysvětlena důležitost používání nových algoritmů, které dokážou plně využít potenciálu dnešních grafických adaptérů a přesunout většinu výpočtů z procesoru na GPU, jelikož výkon grafických karet roste v současnosti mnohem rychleji než výpočetní síla procesorů počítače.

Podrobně byl popsán návrh a implementace knihovny pro rozhraní XNA, která umožňuje v jakémkoliv projektu jednoduše vykreslovat terén algoritmem *Seamless Patches for GPU-based Terrain Rendering*. Poté byla zhodnocena efektivita naší implementace algoritmu a ukázáno její reálné nasazení ve skutečném projektu.

Knihovna by v budoucnu mohla být dále rozšířena například o určitou podporu komprimovaných výškových dat, aby v operační paměti počítače nezabíraly tolik místa, nebo o přesný výpočet hodnoty faktoru přesnosti z požadované velikosti vykreslovaných trojúhelníků v pixelech, čímž by se metrika použitá v tomto algoritmu více přiblížila jiným metrikám.

Literatura

[1] Domovská webová stránka aplikace Google Earth. Dostupné na URL: http://www.google.com/earth/index.html

[2] Domovská webová stránka projektu SRTM. Dostupné na URL: http://www2.jpl.nasa.gov/srtm/

[3] Kršek, P., Španěl, M.: *Základy počítačové grafiky: Úvod do předmětu* [online]. Aktualizováno 2007 [cit. 2010-12-19]. Přístupné studentům FIT VUT v Brně.

[4] Kršek, P., Španěl, M.: *Základy počítačové grafiky: Reprezentace 3D objektů* [online]. Aktualizováno 2008 [cit. 2010-12-19]. Přístupné studentům FIT VUT v Brně.

[5] Tariq, S.: D3D11 Tessellation. In *Game Developers Conference*, March 23-27, 2009, Moscone Center, San Francisco.

[6] Shebanow, M.C.: *The Fermi Architecture* [online]. [cit. 2010-12-19]. Dostupné na URL: http://stanford-cs193g-sp2010.googlecode.com/svn/trunk/lectures/lecture_11/the_fermi_architecture.pdf

[7] Kršek, P., Španěl, M.: *Základy počítačové grafiky: OpenGL* [online]. Aktualizováno 2008 [cit. 2010-12-20]. Přístupné studentům FIT VUT v Brně.

[8] Wright, R.S., aj.: *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Fifth Edition. Ann Arbor (Michigan): Edwards Brothers, 2010. 969 stran. ISBN 0-32-171261-7.

[9] Eisler, C.: *DirectX Then and Now* [online]. Aktualizováno 2006-02-20 [cit. 2010-12-20]. Dostupné na URL: http://craig.theeislers.com/2006/02/directx_then_and_now_part_1.php

[10] Eisler, C.: *Random Discovery - A Brief History of DirectX* [online]. Aktualizováno 2006-08-04 [cit. 2010-12-20]. Dostupné na URL: http://craig.theeislers.com/2006/09/random_discovery_an_brief_hist.php

[11] Limaye, J.: *Evolution of DirectX* [online]. Aktualizováno 2009-07-14 [cit. 2010-12-20]. Dostupné na URL: http://www.techtree.com/India/Guides/Evolution_of_DirectX/551-104330-584.html

[12] Kolektiv autorů: *Wine Status - DirectX DLLs* [online]. [cit. 2010-12-20]. Dostupné na URL: http://www.winehq.org/status/directx

[13] Kolektiv autorů: *Windows DirectX Graphics Documentation* [online]. Aktualizováno 2010 [cit. 2010-12-20]. Dostupné na URL: http://msdn.microsoft.com/cs-cz/library/ee663301(en-us,VS.85).aspx

[14] Abi-Chahla, F.: *OpenGL 3 & DirectX 11: The War Is Over* [online]. Aktualizováno 2008-08-16 [cit. 2010-12-20]. Dostupné na URL: http://www.tomshardware.com/reviews/opengl-directx,2019.html

[15] Domovská webová stránka projektu OpenSceneGraph. Dostupné na URL: http://www.openscenegraph.org/projects/osg [cit. 2010-12-21]

[16] Domovská webová stránka projektu Open Inventor. Dostupné na URL: http://oss.sgi.com/projects/inventor/ [cit. 2010-12-21]

[17] Domovská webová stránka projektu OGRE. Dostupné na URL: http://www.ogre3d.org/ [cit. 2010-12-21]

[18] Ewald, M.: *Game Components and Game Services* [online]. Aktualizováno 2006-11-01 [cit. 2011-01-04]. Dostupné na URL: http://www.nuclex.org/articles/4-architecture/6-game-components-and-game-services

[19] Reed, A.: Learning XNA 4.0. O'Reilly Media Inc., California, 2010. 518 stran. ISBN 978-1-449-39462-2.

[20] Kolektiv autorů: *XNA Game Studio 4.0* [online]. Dostupné na URL: http://msdn.microsoft.com/en-us/library/bb200104.aspx

[21] Bartoň, R.: Modern Algorithms for Real-Time Terrain Visualization on Commodity Hardware. In *Geoinformatics FCE CTU*, May 1-2, 2010, Prague (Czech Republic).

[22] Duchaineau, M., aj.: ROAMing Terrain: Real-time Optimally Adapting Meshes. In *VIS '97 Proceedings of the 8th conference on Visualization '97*, 1997. S. 81-88.

[23] Polack, T.: *Focus on 3D terrain programming*. Premier Press, Ohio, 2003. 218 stran. ISBN 1-59200-028-2.

[24] Hwa, L.M.: Real-time optimal adaptation for planetary geometry and texture: 4-8 tile hierarchies. *IEEE transactions on visualization and computer graphics*, 2005, vol. 11, no. 4, s. 355-368.

[25] De Boer, W. H.: *Fast Terrain Rendering Using Geometrical MipMapping* [online]. Aktualizováno 2000-10 [cit. 2010-12-21]. Dostupné na URL: http://www.flipcode.com/archives/article_geomipmaps.pdf

[26] Brodersen, A.: Real-Time Visualization of Large Textured Terrains. In *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, November 29 – December 02, 2005, Dunedin (New Zealand). S. 439-442.

[27] Ulrich, T.: Rendering Massive Terrains Using Chunked Level of Detail Control. *SIGGRAPH Course Notes*, 2002, vol. 3, no. 5.

[28] Tanner, C., Migdal, C., Jones, M.: The Clipmap: A Virtual Mipmap. In *SIGGRAPH '98 Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM SIGGRAPH. 1998. S. 151-158.

[29] *White Paper on Clipmaps* [online]. NVidia, 2007. Dostupné na URL: http://developer.download.nvidia.com/whitepapers/2007/SDK10/Clipmaps.pdf

[30] Losasso, F., Hoppe, H.: Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. *ACM Transactions on Graphics*, 2004, vol. 23, no. 3, s. 769-776. ISSN: 0730-0301.

[31] Asirvatham, A., Hoppe, H.: Terrain Rendering Using GPU-Based Geometry Clipmaps. *GPU Gems 2*, 2005, vol. 2, s. 27-45.

[32] Clasen, M., Hege, H. C.: Terrain Rendering using Spherical Clipmaps. In *Eurographics/IEEE-VGTC Symposium on Visualization*, May 8-10, 2006, Lisbon (Portugal).

[33] Strugar, F.: Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD). *Journal of Graphics, GPU and Game Tools*, 2009, vol. 14, no. 4, s. 57-74.

[34] Livny, Y., Kogan, Z., El-Sana, J.: Seamless Patches for GPU-based Terrain Rendering. In *WSCG 2007: The 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, January 29 – February 1, 2007, Plzeň (Czech Republic).

[35] Puget Sound Terrain Model [online]. Dostupné na URL: http://www.cc.gatech.edu/projects/large_models/ps.html

[36] Koonce, R.: Deferred Shading in Tabula Rasa. In *GPU Gems 3*. 2. vyd. Kendallville, Indiana: Addison-Wesley Professional, 2007. Kapitola 19, s. 429-458.

[37] Domovská webová stránka aplikace T2. Dostupné na URL: http://www.toymaker.info/html/texgen.html

[38] Domovská webová stránka aplikace EarthSculptor. Dostupné na URL: http://www.earthsculptor.com/

[39] Hargreaves, S.: Deferred Shading [online]. In *Game Developers Conference*, March 24-27, 2004, San Jose, CA. Dostupné na URL: http://www.talula.demon.co.uk/DeferredShading.pdf

[40] Cebenoyan, C.: Direct3D API Issues: Instancing and Floating-point Specials. In *Game Developers Conference*, March 7-11, 2005, San Francisco, CA. Dostupné na URL: http://http.download.nvidia.com/developer/presentations/2005/GDC/Direct3D_Day/D3DTutorial05_Instancing_FPSpecials.pdf

[41] Kolektiv autorů: *Adding and Editing Resources* [online]. Dostupné na URL: http://msdn.microsoft.com/en-us/library/7k989cfy(v=vs.80).aspx

Seznam příloh

Příloha 1. DVD se zdrojovými kódy, spustitelnými programy a elektronickou podobou textu

Příloha 1.

Přiložené DVD obsahuje následující soubory a adresáře:

`Binaries`	spustitelné soubory aplikací
`Source`	zdrojové kódy aplikací
`XNA 4.0`	instalace rozhraní XNA
`readme.txt`	soubor s popisem aplikací
`Thesis.docx`	zdrojový tvar textu
`Thesis.pdf`	elektronická podoba textu diplomové práce