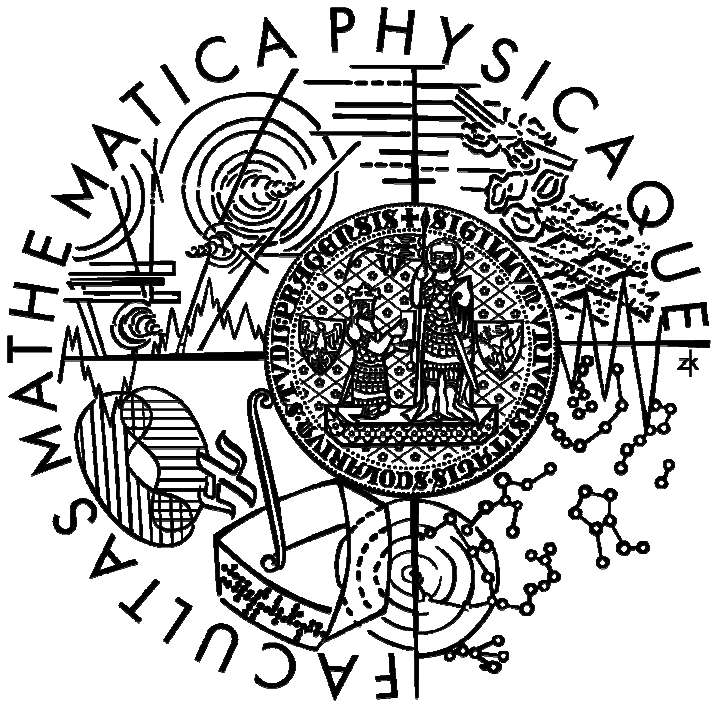
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



LUKÁŠ STEHLÍK

Zobrazování povrchových detailů pomocí mapování textur

Kabinet software a výuky informatiky
Vedoucí diplomové práce: Ing. David Ambrož
Studijní program: Informatika, Softwarové systémy, Počítačová grafika

Rád bych poděkoval Ing. Davidu Ambrožovi za jeho cenné rady a připomínky a za ochotu po celou dobu vedení mé diplomové práce.

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

V Praze dne 13. prosince 2007

Lukáš Stehlík

Obsah

Název práce: Zobrazování povrchových detailů pomocí mapování textur
Autor: Lukáš Stehlík
Katedra: Katedra software a výuky informatiky
Vedoucí diplomové práce: Ing. David Ambrož
e-mail vedoucího: ambrozd@fel.cvut.cz
Abstrakt: Práce se zabývá algoritmy počítačové grafiky využívajícími pokročilé techniky mapování textur pro zvyšování úrovně detailů nerovných povrchů. Stručně seznamuje s historií vývoje a architekturou moderních grafických karet a základními vlastnosti programovacího jazyka Cg pro grafické akcelerátory. Podrobně jsou popsány v praxi používané algoritmy pro simulaci zakřivení povrchu jako jsou normal (bump) mapping a parallax mapping, včetně vysvětlení základních používaných pojmů a principů. Zvláštní pozornost je věnována metodě displacement mapping a její realizaci na moderních grafických kartách. Popsána jsou možná vylepšení uvedených metod se zaměřením na problémy implementace metody displacement mapping. Součástí práce je program umožňující vizualizovat popsané metody včetně vylepšení. Diskutovány jsou výsledky otestování programu na různých grafických kartách. Jednotlivé metody a vylepšení jsou porovnány, a to jak z hlediska kvality zobrazení, tak i rychlosti.
Klíčová slova: normal mapping, parallax mapping, displacement mapping, Cg

Title: Rendering surface detail with advanced mapping techniques
Author: Lukáš Stehlík
Department: Department of Software and Computer Science Education
Supervisor: Ing. David Ambrož
Supervisor's e-mail address: ambrozd@fel.cvut.cz
Abstract: The thesis deals with algorithms of computer graphics which exploit advanced techniques of textures mapping with the aim of improving the projection of details of wrinkled surfaces. The evolution and architecture of modern graphic cards are described. There is the description of basic characteristics of the Cg language for graphical accelerators too. Algorithms for simulation of wrinkled surfaces such as normal (bump) mapping and parallax mapping are described in detail including the explanation of basic terms and principles. Extra focus is laid on the method named displacement mapping and its application on modern graphical cards. The thesis describes possible improvements of the above mentioned methods with a view to problem of implementation of displacement mapping method. Part of the work is a program that visualizes methods including improvements. There is a discussion on results obtained from testing the program on different graphical cards. All the methods and their improvements are compared with respect to both the projection quality and the speed of processing.
Keywords: normal mapping, parallax mapping, displacement mapping, Cg

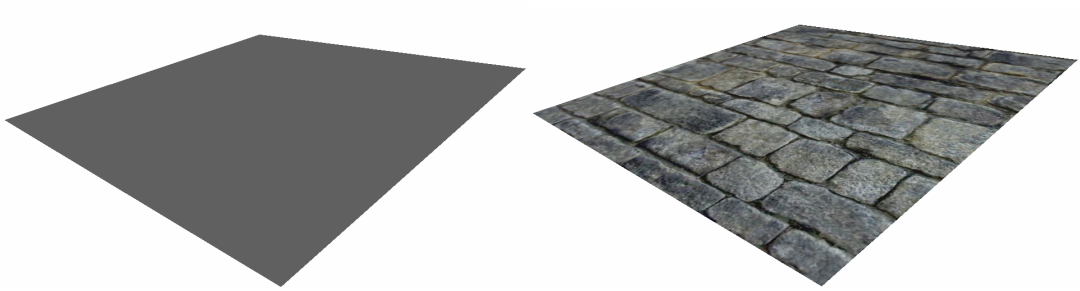
Kapitola 1

Úvod

Díky rychlému a masivnímu rozvoji počítačové techniky v posledních desetiletích a jejímu následnému rozšíření do kanceláří a domácností došlo ke zvýšení nároků nejen na výpočetní výkon, ale také na grafiku používaných aplikací, a to zejména v důsledku obliby počítačových her. Požadavky uživatelů, kteří se nechtěli spokojit s primitivní grafikou programů, donutily výrobce počítačových komponent věnovat značnou pozornost také oblasti grafického hardware. Díky jeho dynamickému vývoji se tak mohou tvůrci moderních grafických algoritmů zaměřit na stále větší detaily.

V 80. letech 20. století se pro vykreslení trojrozměrného obrazu používal drátový model a jednoduchá vektorová grafika umožňující celistvé zobrazení výsledného povrchu objektů. Plošky objektů byly vybarveny jednou barvou, popř. byla tato barva interpolována pro vytvoření jemnějších barevných přechodů. To však v žádném případě nemohlo splnit požadavky uživatelů. Výkon tehdejšího hardware však stěží zvládal i takovéto primitivní zobrazovací techniky.

Počátkem 90. let nastal velký průlom v kvalitě grafického výstupu počítače. Pro vykreslení povrchu se začala používat technika mapování textur („nalepení“ obrázků na povrch objektu). Textury dodávají i velmi primitivním objektům skutečný vzhled a jejich použití tedy značným způsobem ovlivňuje nejenom realističnost scény, ale také její složitost.



Obrázek č. 1: Vizuální porovnání vzhledu objektu bez a s použitím textury

Počítačová grafika se neomezuje na použití jediné textury pro jeden povrch, ale bohatě využívá aplikace více textur současně (*multitexturing*). Vhodným použitím textur lze usnadnit práci grafickému hardware například přípravou osvětlení nebo stínů do samostatné textury. Na textury navíc není třeba nahlížet jen jako na obrázky, ale obecně jako na zobrazení z jednoho oboru hodnot do druhého. Kromě barvy totiž textura může nést libovolná jiná data - v následujícím textu tomu tak je například v případě normálové mapy nebo výškové mapy, kde jednotlivé pixely textury (tzv. *texely*) nenesou barevnou informaci potřebnou pro modulaci barvy na povrchu objektu, ale obsahují souřadnice normálového vektoru nebo informace o posunutí daného pixelu na povrchu objektu. Aplikací textur je tedy možné vytvořit i pokročilejší grafické efekty jako odraz okolí nebo simulovat zakřivení povrchu.

1.1 Zobrazování povrchových detailů

Povrch, na který jsou 2D textury běžně nanášeny, je plochý. To nevadí, je-li objekt s aplikovanou texturou zobrazován z velké vzdálenosti, ovšem při pohledu zblízka může některým texturám scházet hloubka (například nejsou vidět různé výstupky, rýhy apod.). Tento problém by samozřejmě vyřešilo použití podrobné geometrie s vymodelovanými veškerými výstupky, pracovat s takovým povrchem by však bylo neúnosně výpočetně i paměťově náročné, nemluvě o zobrazování celých rozsáhlých scenérií se stovkami objektů – naprosto běžné v případě 3D počítačových her.

A tady proto přichází ke slovu zobrazování povrchových detailů pomocí pokročilejších metod využívajících mapování textur. Základním cílem je zpracovat 2D texturu takovým způsobem, aby na naneseném povrchu vypadala alespoň zčásti trojrozměrně. Výsledkem je detailnější a mnohem realističtější zobrazení daného povrchu. Efektu trojrozměrnosti může být

docíleno například zvýrazněním některých stínů na textuře (*bump mapping*, *normal mapping*) nebo změnou způsobu, jakým je textura na povrch mapována (*parallax mapping*). Vrcholem v oblasti zobrazování geometrických detailů povrchu na základě mapování textur je metoda zvaná *displacement mapping* (ukázka viz Obrázek č. 2), která pomocí textury zvyšuje úroveň detailu objektu skutečnou změnou geometrie a její implementace v real-time počítačové grafice je možná až díky nástupu moderních grafických karet.

Obrázek č. 2: Ukázka zobrazení povrchu pomocí metody displacement mapping

V následující kapitole je stručně popsána historie vývoje a architektura moderních grafických karet, kapitola 3 pak uvádí základní vlastnosti programovacího jazyka Cg pro tyto moderní grafické akcelerátory.

Kapitola 4 se věnuje v praxi používaným algoritmům pro simulaci zakřivení povrchu. Vysvětleny jsou základní pojmy a principy používané v této oblasti, například normálové mapy či problematika zobrazovacích prostorů. Dále jsou podrobněji popsány metody normal mapping (bump mapping) a parallax mapping. U jednotlivých metod jsou uvedeny programy pro vertex a fragment procesor, kterými je daná metoda implementována.

Metodou displacement mapping, které je věnována zvláštní pozornost, a její realizací na moderních grafických kartách se zabývá Kapitola 5. Na rozdíl od metod popsaných v předchozí kapitole, které nerovnosti na povrchu pouze simulují, tato metoda skutečně mění geometrii objektu, a to přímo grafickým procesorem na základě informací ze speciálních textur. Vedle podrobného seznámení s algoritmem a implementací obsahuje tato kapitola také možná vylepšení v práci uvedených metod se zaměřením na problémy implementace metody displacement mapping.

Kapitola 6 obsahuje seznámení s programem, který je součástí práce a umožňuje vizualizovat popsané metody včetně vylepšení a měnit jejich základní parametry i použité textury. Je tak možné porovnat popsané metody i posoudit přínos implementovaných vylepšení. Jednotlivé metody a vylepšení jsou porovnány v závěru kapitoly, a to jak z hlediska kvality zobrazení, tak i z hlediska rychlosti. Diskutovány jsou také výsledky otestování programu na různých grafických kartách.

Kapitola 2

Moderní grafický hardware

Motorem vývoje moderního grafického hardware jsou propracované 3D počítačové hry, které využívají stále přibývající možnosti grafických procesorů (GPU – *graphics processing unit*). V konzumním sektoru se grafické 3D akcelerátory objevily teprve ve druhé polovině 90. let (příkladem jsou grafické karty 3dfx Voodoo, NVIDIA GeForce) a ačkoliv ve své době dosáhly velkého úspěchu, jejich možnosti byly značně omezené, neboť využívaly tzv. **fixní zobrazovací řetězec** (*fixed function pipeline* – podrobněji popsaný v následující podkapitole).

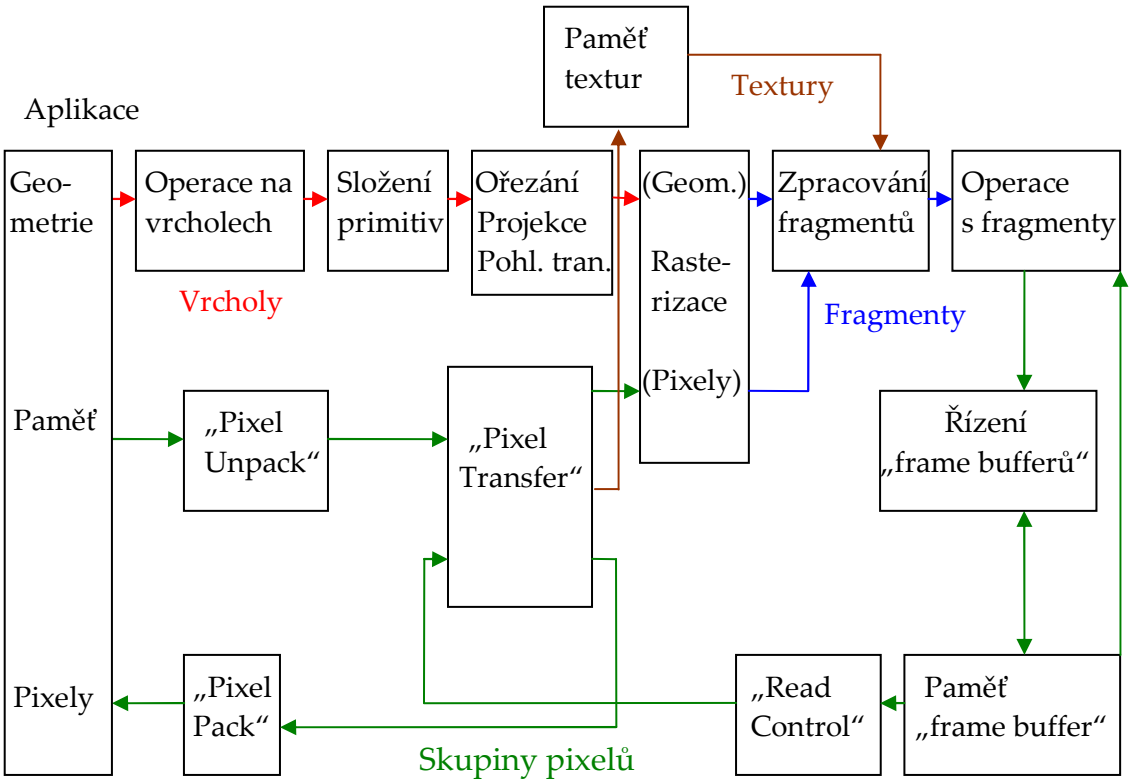
2.1 Fixní a programovatelný zobrazovací řetězec

Grafická karta převádí trojrozměrnou scénu (virtuální svět tvořený objekty, světelným zdrojem, …) z paměti počítače do dvourozměrného prostoru – typicky obraz na monitoru. Postup je rozdělen do několika částí zobrazovacího řetězce:

1) zaslání geometrie scény z aplikace do grafické karty,
2) zpracování geometrie na základě vrcholů[1],
3) složení geometrických primitiv (trojúhelníků, čtyřúhelníků, …) z jednotlivých vrcholů,
4) ořezání geometrie (*clipping*) na hranici tzv. pohledového tělesa (*viewing frustum*)
5) odstranění zbytečné geometrie:
    a. mimo záběr kamery (*viewing frustum culling*),
    b. odvrácená od kamery (*back-face culling*),
6) převedení pohledu kamery ve 3D scéně na 2D obraz (rasterizace),
7) zpracování fragmentů[2],

[1] Každý objekt ve scéně je složen z několika plošek (polygonů), které jsou definovány hraničními vrcholy (vertexy)

8) uložení fragmentů do paměti a jejich vykreslení na obrazovce, případně zpětné načtení do aplikace.



Obrázek č. 3: Schéma fixního zobrazovacího řetězce OpenGL

Podrobnější popis všech fází lze nalézt v [1].

Možnosti práce s fixním zobrazovacím řetězcem jsou velmi omezené, funkce jednotlivých bloků je totiž neměnná, a to včetně pořadí, ve kterém se provádí. Průchod zobrazovaných dat řetězcem lze ovlivnit pouze pomocí přepínačů (například provést / neprovést) a parametrů grafického rozhraní (například barva a umístění světla) – to však značně limituje programátory.

Například ve fázi zpracování geometrie (operace na vrcholech) je na starších grafických kartách prováděna automatická transformace vrcholů do soustavy souřadnic s kamerou v počátku (podrobněji viz 4.1.1 Zobrazovací prostory) a u osvětlení je možné zvolit jen předdefinované modely osvětlení z grafického API – celkově je tato fáze označována jako *Transform & Lightning*, resp. T & L. Zpočátku byla i tato fáze prováděna na CPU, až od let 1999-2000 po příchodu nové generace GPU (NVIDIA GeForce 256 a 2, ATI Radeon 7500 a S3 Savage3D) se o transformace vrcholů a osvětlení stará grafický hardware.

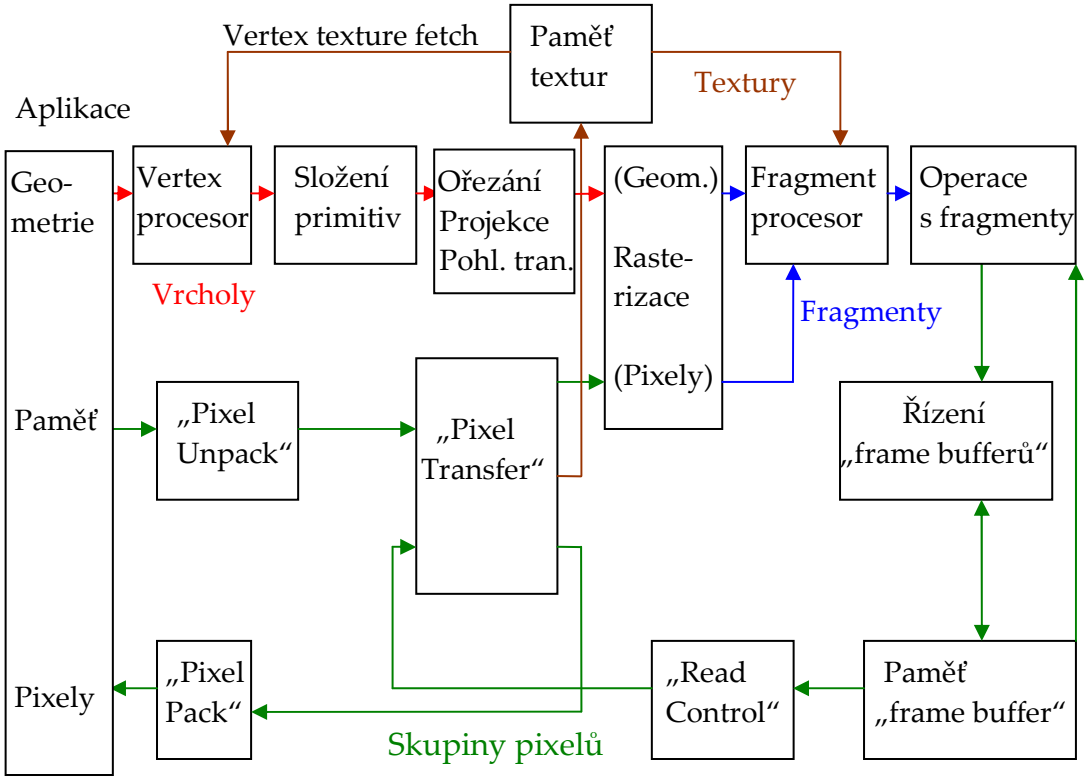
---

[2] Fragmentem se v počítačové grafice rozumí soubor dat (např. barva, pozice a hloubka), který je potřebný pro vygenerování pixelu do paměti zvané frame buffer

Zároveň se objevily první pokusy o programovatelnost grafických procesorů. První generace takových grafických karet obsahovaly tzv. *register combiners*, díky nimž bylo možné programovat jednoduché operace s fragmenty.

Pro zvýšení flexibility grafických karet se v roce 2001 na trhu objevily první grafické akcelerátory (NVIDIA GeForce 3 a 4 Ti, ATI Radeon 8500) s tzv. **programovatelným zobrazovacím řetězcem** (*programmable pipeline*). Bohužel možnosti programování nových GPU byly zpočátku velmi omezené – hlavní nedostatek představoval velmi malý počet instrukcí programů pro GPU (tzv. *shadery*), bylo totiž povoleno jen několik assembler instrukcí. A proto byl hned v roce 2002 jejich rozsah rozšířen, ovšem nijak významně (maximálně 256 instrukcí), navíc stále chyběly důležité vlastnosti jako např. podpora cyklů a dynamického větvení programů – příkladem jsou grafické karty ATI Radeon 9500 a NVIDIA GeForce FX z let 2003-2004.

Programovatelný zobrazovací řetězec obsahuje dva nové zcela programovatelné bloky – vertex procesor a fragment procesor, které nahrazují části fixního řetězce pracující s vrcholy a fragmenty.



Obrázek č. 4: Schéma programovatelného grafického řetězce OpenGL

Vertex/fragment procesory se programují pomocí speciálních programovacích jazyků pro grafické karty (více v následující kapitole). Programátor tak může vytvořit zcela nový způsob výpočtu osvětlení a manipulace s vrcholy, nebo libovolně pracovat s parametry jednotlivých fragmentů. Je třeba zdůraznit, že pokud je použit vertex program (tzv. *vertex shader*) nebo fragment program (tzv. *fragment shader*) jsou tím ve skutečnosti nahrazeny operace na vrcholech a zpracování fragmentů ve fixním řetězci. Do vertex programu je tedy nutné zahrnout transformace souřadnic vrcholu a fragment program musí provádět výpočty barvy jednotlivých fragmentů. Osvětlení se počítá buď ve vertex shaderu nebo ve fragment shaderu v závislosti na použitém osvětlovacím modelu.

2.2 NVIDIA GeForce 6x00, 7x00 a ATI Radeon X1x00

Novější architektury GPU, které podporují mnohem více instrukcí a vlastností, na sebe nenechaly dlouho čekat (2004-2006: NVIDIA GeForce 6x00 a 7x00, ATI X1x00).

Jedním z největších přínosů nových NVIDIA GPU je podpora přístupu vertex shaderu do paměti textur (*vertex texture fetch*, viz šipka vedoucí z paměti textur do vertex procesoru na předchozím schématu) - přestože se jedná o časově náročnou operaci (ekvivalent 20 instrukcí u GeForce 6800), má velký potenciál a umožňuje implementovat nové efekty jako například simulaci kapalin, vodní hladiny, exploze a uplatní se také v metodě displacement mapping. Grafické karty ATI X1x00 funkci vertex texture fetch nepodporují a obcházejí tuto funkcionalitu tak, že data vypočítaná ve fragment shaderu posílají do vertex shaderu pomocí funkce zvané *render to vertex buffer* (R2VB). Další řady grafických karet ATI již podporují přímo čtení textur z vertex programu.

Podpora dynamického větvení programu (*dynamic branching*) umožňuje kontrolu nad probíhajícím výpočtem a ukončení výpočtu dříve než je dosažen konec programu. Je tak možné přeskočit nepotřebné výpočty a program optimalizovat.

2.3 NVIDIA GeForce 8x00 a ATI Radeon HD 2x00, HD 3x00

V listopadu 2006 uvedla firma NVIDIA na trh řadu GeForce 8x00, v roce 2007 následovala firma ATI (resp. firma AMD, která ATI Technologies koupila) s Radeon HD 2x00 a ke konci roku 2007 s Radeon HD 3x00 se sníženou spotřebou. Nejnovější řady karet nabízejí unifikovanou

architekturu (viz následující odstavec) a novou geometrickou jednotku, která nově umožňuje vytvářet a modifikovat geometrii přímo uvnitř GPU – díky tomu lze implementovat např. adaptivní teselaci[3] přímo na grafickém procesoru. Zásadním způsobem se tak opět zvyšují možnosti grafických karet.

Unifikovaná architektura - předchozí generace grafických karet obsahovaly určitý počet specializovaných výpočetních jednotek pro zpracování vrcholů pomocí vertex programů a zpravidla větší počet jednotek pro zpracovaní fragmentů analogicky pomocí fragment programů. Pokud byla zobrazována scéna náročná na zpracování geometrie, část hardware zpracovávající vrcholy byl plně vytížen, zatímco na straně výpočetních jednotek pro zpracování fragmentů tomu bylo naopak. Obdobně v opačném případě při scéně náročné na zpracování fragmentů (např. vodní hladina) nebyl plně vytížen hardware pro práci s vrcholy. Unifikovaná architektura znamená, že jednotlivé procesory mohou být využívány jak pro zpracování vrcholů, tak i fragmentů. Takový návrh je výhodnější a grafický hardware může plně využít své výpočetní síly i v obou extrémních případech.

Dalším přínosem je schopnost grafického procesoru kopírovat geometrii (*geometry instancing*) v OpenGL 2.0. Nachází-li se ve vykreslované scéně několik stejných objektů, namísto zasílání geometrie a vykreslování každého objektu zvlášť lze vykreslit pouze jeden objekt a na ostatní místa vykreslený objekt okopírovat jen za použití informací pro specifikaci jednotlivých instancí objektů. Nedocílí se tím zvýšení kvality obrazu, vykreslení scény však bude na nových GPU rychlejší.

2.4 Specifikace vybraných grafických karet

V tabulce č. 1 jsou uvedeny podrobné specifikace několika vybraných grafických akcelerátorů. Ke stěžejním výhodám moderních grafický karet obecně patří velmi vysoká šířka paměťové sběrnice (až 512 bit), vysoká frekvence pamětí (přes 1 GHz) a tedy i obrovská maximální propustnost pamětí.

[3] Rozdělení geometrie na jemnější síť trojúhelníků.

Tabulka č. 1: Specifikace vybraných grafických karet[4]

Model	Max. frek-vence jádra (MHz)	Max. fillrate (mld texelů/s)	Shader		Paměť				Počet tranzistorů (miliony)	Výkon shaderů (GFLOPS)
			Po-čet[5]	Frek-vence (MHz)	Max.pro-pustnost (GB/s)	Šířka sběrnice (bit)	MB	Frek-vence (MHz)		
Radeon X850 XT PE	540	8,6	6+16	N/A	37,8	256	256	1180	160	N/A
7900 GTX	650	15,6	8+24	N/A	51,2	256	512	1600	278	N/A
GF 8800 GTS	500	24	96	1200	64	320	320, 640	1600	681	345,6
GF 8800 GTX	575	36,8	128	1350	86,4	384	768	1800	681	518,4
GF 8800 Ultra	612	39,16	128	1500	103,68	384	768	2160	681	576,0
Radeon HD 2900XT	740	36,8	320	1650	105,6	512	512	1650	700	475
Radeon HD 3870	775	N/A	320	N/A	N/A	256	512	2250	666	N/A

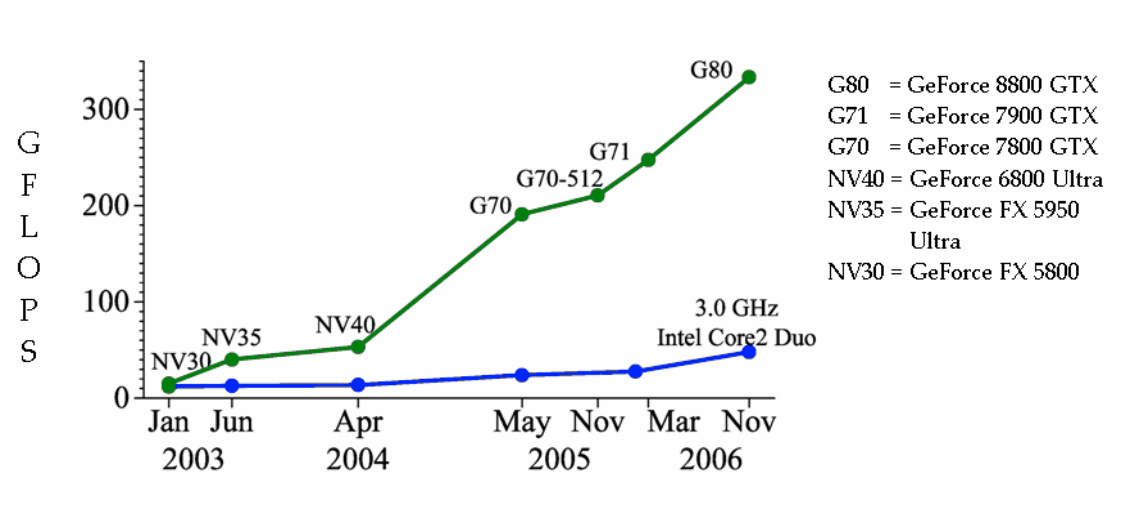
Architektura GPU je optimalizována pro vektorové výpočty, které se v počítačové grafice často vyskytují. Tradiční CPU jsou naopak optimalizovány pro sekvenční kód, velká část procesoru není určena přímo pro výpočty, ale například pro předvídání větvení nebo vyrovnávací paměť. Dnešní GPU jsou velmi dobře programovatelné a skutečnost, že se stávají výkonnějšími než CPU, podporuje jejich využití i mimo oblast počítačové grafiky[6].

[4] Specifikace jednotlivých grafických karet se mohou mírně lišit podle výrobce.

[5] Počet unifikovaných výpočetních jednotek pro zpracování shaderů v případě jednoho čísla; v případě součtu počet vertex + počet fragment procesorů.

[6] Známé jako GPGPU – General Purpose computation on a GPU, http://www.gpgpu.org.

Ohromný výkon grafického hardware je docílen také vysokou úrovní paralelismu – nejnovější GPU obsahují až 320 výpočetních jednotek pro zpracování shaderů o celkovém výpočetním výkonu v řádu 100 miliard operací v pohyblivé řádové čárce (GFLOPS). Výpočetní výkon moderních grafických karet znázorňuje graf převzatý z [2].



Obrázek č. 5: Výpočetní výkon moderních grafických karet

Kapitola 3

Úvod do programování GPU

3.1 Programovací jazyky pro GPU

Programování grafických procesorů je stejně jako u CPU možné v nižším programovacím jazyku, který je blízký jazyku symbolických adres (*assembler*). Takto vytvořené programy jsou však silně vázané na použitý grafický hardware a tedy z velké části nepřenositelné mezi jednotlivými typy grafických akcelerátorů. Také horší čitelnost a udržovatelnost takového kódu byly hlavní důvody vzniku několika vyšších programovacích jazyků pro GPU. Běžně používané jazyky vychází z modelu tří samostatných procesů:

1. aplikace pro OS (Windows, Linux, …)
   (vykonává CPU)
2. vertex shader, který zpracovává každý vrchol
   (vykonává GPU vertex procesor)
3. fragment shader[7], který zpracovává každý fragment
   (vykonává GPU fragment procesor)

Firma Microsoft pro své grafické rozhraní Direct3D vytvořila High Level Shading Language (HLSL). Jeho nevýhodou však je, že Direct3D je k dispozici pouze pro operační systém Windows. Konsorcium OpenGL ARB (Architecture Review Board), které je od roku 2006 součástí konsorcia Khronos Group, jehož členy jsou mimo jiné firmy AMD (ATI), Apple, Dell, Intel, NVIDIA a Sun Microsystems, vyvinulo programovací jazyk OpenGL Shading Language (GLSL). GLSL je od OpenGL verze 2.0 součástí jeho standardu a lze ho využít na všech operačních systémech podporujících rozhraní OpenGL (pokud je verze 2.0 podporována) - vedle Windows zejména na Linuxu a Mac OS. Protože je však Direct3D stále nejrozšířenějším používaným rozhraním, vyvinula firma NVIDIA navíc samostatně

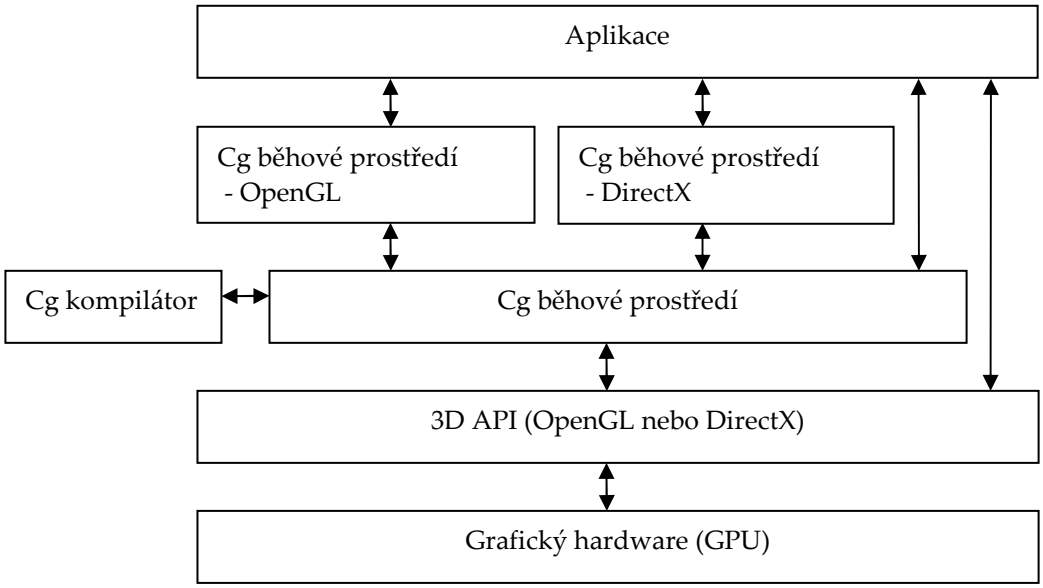
[7] Fragment shader v terminologii OpenGL, někdy se nazývá také pixel shader.

programovací jazyk Cg, který podporuje obě rozhraní: Direct3D i OpenGL a jeho použití je tedy nejuniverzálnější.

3.2 Jazyk Cg

Jazyk Cg (*C for graphics*) vychází primárně ze syntaxe jazyka C, ale přebírá některé vlastnosti novějších programovacích jazyků jako jsou C++ nebo Java. Navíc je jazyk Cg upravený pro lepší využití grafického hardware, například nabízí nové datové typy, které se v algoritmech počítačové grafiky často používají. Některé vlastnosti známých programovacích jazyků Cg naopak postrádá, nepodporuje ukazatele, instrukce skoku a jinak pracuje s poli.

Cg není jen programovací jazyk pro GPU, ale také běhové prostředí, díky němuž je možné až za běhu programu identifikovat grafický hardware a přeložit Cg program s optimalizacemi pro konkrétní GPU a vybrané 3D grafické rozhraní. Propojení technologií znázorňuje následující schéma:



Obrázek č. 6: Schéma propojení technologií

Dynamický vývoj grafických karet, který v posledních letech zásadně ovlivňuje možnosti grafického hardware, si v Cg vyžádal použití tzv. profilů. Každý z nich specifikuje konkrétní množinu vlastností jazyka, které jsou pro daný profil podporovány a také určuje přesnost datových typů. V Cg existuje několik profilů, které přibližně odpovídají různým generacím grafických karet. Tyto profily se dělí na profily pro vertex programy a profily pro fragment programy.

Pro každý program v Cg je třeba externě z aplikace nebo přímo v Cg kódu určit konkrétní profil, v rámci kterého bude program kompilován. Programátor tak může pro různé profily předem připravit odlišné implementace (nejenom) grafických efektů. To umožňuje optimalizaci programu pro jednotlivé třídy grafických karet. Dále například pokud Cg program využívá cyklus a v daném profilu není for-cyklus podporován, Cg tento cyklus rozloží do sekvenční formy a pokud není přesažena maximální povolená délka programu, je program úspěšně spuštěn i na hardware bez podpory cyklů.

Vedle obecného kódu aplikace vytvořené v OpenGL nebo Direct3D zpracovává Cg dva speciální typy programů - vertex shader a fragment shader. Na každý vrchol ve scéně je nejdříve aplikován výpočet pomocí vertex programu, který pracuje s parametry vrcholů (souřadnice a barva vrcholu, normála a další vlastnosti) a transformuje jednotlivé vrcholy do soustavy souřadnic pro zobrazení na obrazovce (popis zobrazovacích prostorů viz Kapitola 4.1.1. Zobrazovací prostory). Atributy předávané mezi vertex shaderem a fragment shaderem lze ve vertex shaderu definovat pomocí prefixu `out` a poskytnout je tak fragment shaderu. Výstupy vertex programu jsou následně interpolovány a rasterizovány na fragmenty v rámci geometrických primitiv (bod, úsečka, trojúhelník,…) vymezených vrcholy a na každý fragment obrazu je aplikován fragment program, který vypočte výslednou barvu, popřípadě upraví hloubku každého fragmentu na základě interpolovaných informací z vertex programu, textury a případně dalších parametrů poskytnutých z aplikace.

Vertex a fragment programy mohou navíc pracovat s parametry z aplikace, Cg programy mají standardně definovanou vstupní funkci `main` (obdobně jako v C/C++, Java), tuto vstupní funkci lze přednastavit z aplikace a vytvořit tak v jednom .cg souboru více vstupních funkcí například pro různé profily. Samozřejmostí je možnost vytvořit nové funkce včetně jejich přetěžování.

3.2.1 Datové typy a operandy

Jazyk Cg pracuje s některými datovými typy, které jsou známé již z jazyka C (například float – číslo v pohyblivé řádové čárce, int – celé číslo). Na těchto datových typech jsou definovány standardní aritmetické operace a práce s nimi je tedy v podstatě stejná (součet +, násobení *, …).

Pro počítačovou grafiku je však charakteristická práce s vektory a maticemi, pro které jazyk Cg obsahuje speciální datové typy – například:

`float2, float3, float4` – 2, 3 a 4 složkový vektor čísel v pohyblivé řádové čárce,

`float4x4` – matice čísel v pohyblivé řádové čárce o rozměrech 4 x 4.

Cg podporuje nad těmito datovými typy maticové násobení, skalární (operace `dot`) i vektorový součin vektorů (`cross`), násobení matice skalárem nebo vektorem (`mul`), normalizace vektorů (`normalize`), lineární interpolaci (`lerp`) a další. Navíc kompilátor může v případě těchto speciálních typů některé operace optimalizovat a provádět v jedné instrukci.

Ke složkám vektoru se přistupuje pomocí operátoru `.` (tečka), jednotlivé složky se nazývají buď `x`, `y`, `z`, `w`, nebo `r`, `g`, `b`, `a` (1. až 4. složka vektoru). Příklad vytvoření vektoru (1, 0, 0) pomocí operátoru tečka:

```
float3 barva;
barva.r = 1;
barva.g = 0;
barva.b = 0;
```

je samozřejmě ekvivalentní zápisu:

```
float3 barva = {1, 0, 0}
```

Navíc je operátor . využíván v případě efektivního prohazovaní složek vektoru (tzv. *swizzling*):

```
barva.rgb = barva.bgr; // barva = {0, 0, 1}
```

nebo k vytváření vektorů replikací skaláru (tzv. *smearing*). Příklad:

```
float blue = barva.b; // blue = 1;
barva = blue.xxx // barva = {1,1,1}
```

Klíčovým datovým typem v metodách pro zobrazování povrchových detailů pomocí mapování textur je datový typ `sampler2D`. Jedná se o ukazatel na texturu, přes který je možné číst informace v ní uložené pomocí operace `tex2D`:

```
sampler2D textura : TEXUNIT0;
float2 souradnice : TEXCOORD0;
…
float3 barva = tex2D(textura,souradnice);
```

V prvních dvou řádcích příkladu za názvem a dvojtečkou je tzv. sémantika, která informuje grafický hardware o tom, jak má data propojit se zbytkem zobrazovacího řetězce. Sémantika TEXUNIT0 u proměnné textura znamená, že tato proměnná ukazuje na texturu uloženou v první, resp. nulté texturovací jednotce. Sémantika TEXCOORD0 značí příslušné texturovací souřadnice pro aktuálně zpracovávaný vrchol nebo fragment. Mezi další důležité sémantiky patří COLOR (barva vrcholu, fragmentu) a POSITION (pozice).

Důležitým modifikátorem datových typů je klíčové slovo `uniform`. Uniformní proměnné v Cg programu jsou globální proměnné, které jsou inicializovány externě aplikací nebo běhovým prostředím Cg. Například:

```
uniform float4x4 matice
```

se nejčastěji používá pro předání transformačních matic z aplikace do vertex programu, jehož nejdůležitějším úkolem je určit novou pozici vrcholu po transformaci (podrobnosti v následující kapitole), a to právě jednoduchým přenásobením transformační matici 4x4.

3.2.2 Další vývoj jazyka Cg

Teprve ke konci roku 2007, rok po vydání prvních grafických karet (NVIDIA GeForce 8800) s novou geometrickou jednotkou, byla vydána beta verze Cg 2.0, která oproti poslední verzi Cg 1.5 ze září 2007 podporuje nový typ programů pro geometrickou jednotku (geometry shader) – vyžaduje grafické rozhraní DirectX 10.0 (součástí Windows Vista) nebo OpenGL 2.0 s rozšířeními.

Pro více informací o jazyce Cg viz [3], bohužel kniha zatím nepokrývá Cg 2.0.

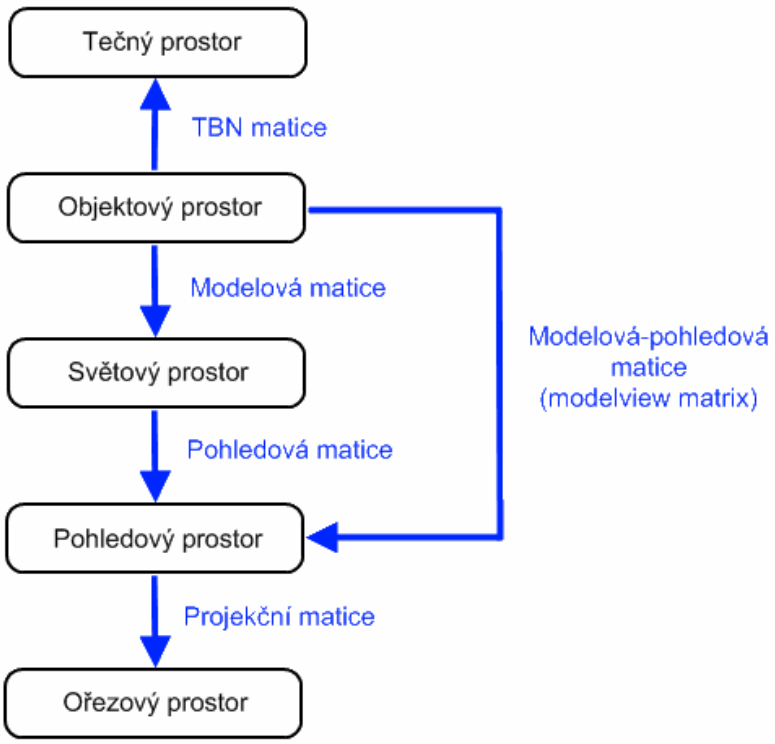
Kapitola 4

Metody pro simulaci zakřivení povrchu

4.1 Základní pojmy

4.1.1 Zobrazovací prostory

Jedním z předpokladů pro práci v OpenGL a Cg je správné pochopení funkce zobrazovacích prostorů a příslušných transformací. Vykreslovaná scéna je při průchodu zobrazovacím grafickým řetězcem převáděna mezi různými zobrazovacími prostory.



Obrázek č. 7: Vztahy mezi používanými zobrazovacími prostory

Při psaní kódu OpenGL aplikace pracuje programátor se souřadnicemi objektového prostoru (*object space*). Pozice vrcholů v objektovém prostoru jsou z aplikace předány do GPU, resp. vertex shaderu. Jak již bylo zmíněno, vertex shader má transformovat vrcholy do ořezového prostoru.

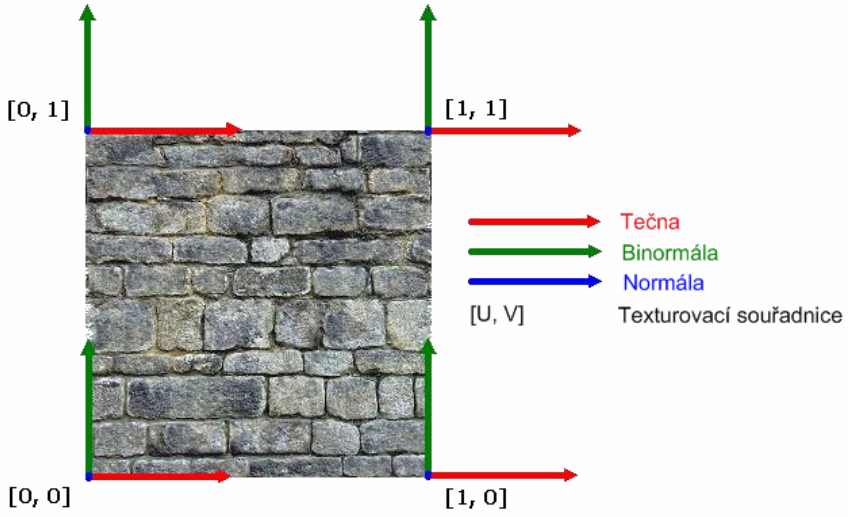
Obecně může mít každý objekt ve scéně svůj objektový prostor. Vzájemné umístění objektů se projeví při jejich reprezentaci v souřadnicích světového prostoru (*world space*). V OpenGL se však světový prostor standardně nepoužívá.

Dalším prostorem, kterým zobrazovaný objekt prochází v rámci zobrazovacího řetězce je pohledový prostor (*view space, eye space*). Tento prostor představuje pohled na scénu z pozice kamery (pozorovatele), která je umístěna v počátku soustavy souřadnic. Souřadnice z objektového do pohledového prostoru převádí tzv. modelová-pohledová matice (*modelview matrix*). Tato matice reprezentuje například rotaci a posunutí objektu a posunutí pozice kamery do počátku včetně správného natočení pohledu.

Dalším krokem je určení bodů, které je možné vidět z pohledu kamery. Pohledový prostor je tedy pro zobrazení scény na obrazovce monitoru ořezán podle pohledového tělesa (standardně určeného šesti rovinami). Takto vzniklý prostor se nazývá ořezový prostor (*clip space*). Transformace z pohledového do ořezového prostoru je prováděna pomocí tzv. projekční matice (*projection matrix*).

Tečný prostor

Do této chvíle ještě nebyl zmíněn tečný prostor. Jedná se o prostor vztažený k povrchu. Pro každý vrchol je možné určit jeho normálu, tečnu a binormálu, jak ukazuje následující obrázek částečně převzatý z [4]:



Obrázek č. 8: Normály, binormály a tečny ve vrcholech

Tečna jako osa x, binormála jako osa y a normála jako osa z tvoří dohromady bázi tečného prostoru. Při aplikaci metod pro zvyšování úrovně detailů nerovných povrchů je často třeba převést určitý vektor z objektového prostoru do tečného prostoru. Transformace se provádí vynásobením daného vektoru inverzí tzv. TBN matice, jejíž sloupce jsou tvořeny tečnou **T**, binormálou **B** a normálou **N** (odtud název TBN matice):

$$\begin{pmatrix} T_{(x)} & B_{(x)} & N_{(x)} \\ T_{(y)} & B_{(y)} & N_{(y)} \\ T_{(z)} & B_{(z)} & N_{(z)} \end{pmatrix}$$

Při implementaci lze tečnu a binormálu pro každý vrchol určit na základě výpočtu. Odvození vzorců je možné najít v [5], zde jsou uvedeny pouze výsledné vzorce:

$$T = \frac{1}{M} * \left(\Delta t3t1_{(v)} * \Delta v2v1 - \Delta t2t1_{(v)} * \Delta v3v1\right)$$

a **(1)**

$$B = \frac{1}{M} * \left(-\Delta t3t1_{(u)} * \Delta v2v1 + \Delta t2t1_{(u)} * \Delta v3v1\right),$$

kde

$\Delta v2v1$ značí vektor z vrcholu 1 do vrcholu 2,

$\Delta v3v1$ značí vektor z vrcholu 1 do vrcholu 3,

$\Delta t2t1_{(u)}$ značí rozdíl textur. souřadnice U vrcholu 2 – textur. s. U vrcholu 1,

$\Delta t3t1_{(u)}$ značí rozdíl textur. s. U vrcholu 3 – textur. s. U vrcholu 1,

$\Delta t2t1_{(v)}$ značí rozdíl textur. s. V vrcholu 2 – textur. s. V vrcholu 1,

$\Delta t3t1_{(v)}$ značí rozdíl textur. s. V vrcholu 3 – textur. s. V vrcholu 1

a

$M = \Delta t2t1_{(u)} * \Delta t3t1_{(v)} - \Delta t3t1_{(u)} * \Delta t2t1_{(v)}$.

Normála je pak většinou vypočítána jako vektorový součin tečny a binormály. Výhodou je, že TBN matice je potom ortogonální a k získání její inverze stačí matici transponovat.

Každý vrchol má svůj vlastní tečný prostor a je zřejmé, že osy soustavy souřadnic tečného prostoru se mění při rotaci objektu, ale nemění se při jeho posouvání. TBN matici je tedy nutné přepočítat po každé rotaci objektu, při pouhém posouvání nikoliv. Vymezují-li navíc vrcholy trojúhelník (případně čtyřúhelník), leží v jedné rovině a mají stejné tečny, binormály i normály. TBN matici pak není nutné počítat pro každý vrchol, ale pouze pro každý trojúhelník.

4.1.2 Osvětlovací model

Pro pochopení metod pro simulaci zakřivení povrchu je dále třeba vysvětlit, jakým způsobem je barva na povrchu objektu vypočtena. Standardní OpenGL používá pro výpočet osvětlení vrcholů tzv. Phongův model, který publikoval Bui-Tuong Phong v roce 1973 [6]. Jedná se o lokální osvětlovací model, to znamená, že neuvažuje světlo, které se na cestě od světelného zdroje k pozorovateli (kameře) odrazí od více než jednoho povrchu. Phongův osvětlovací model nevychází z fyzikální podstaty šíření a odrážení světla, ale je založen na empiricky stanoveném vzorci. Jeho výhodou je snadná pochopitelnost a použitelnost v real-time aplikacích. Vypočtením osvětlovacího modelu v každém bodě zobrazovaného povrchu se získá tzv. Phongovo stínování.

Ve Phongově modelu se světlo odrážející směrem k pozorovateli skládá ze tří složek:

Zbytkové světlo **A** (*ambient light*) = konstantní množství světla, které je aplikováno na každý bod. Nahrazuje světlo, které by osvětlilo části povrchu po vícenásobném odrazu. Zabrání se tak výskytu nerealisticky černých stínů v místech, na které nedopadá přímé světlo.

Difusní odraz **D** (*diffuse light*) = světlo, které se po dopadu na povrch rovnoměrně rozptýlí do všech směrů. Jeho intenzita závisí na úhlu, pod kterým světlo na povrch dopadá – čím větší je úhel dopadu světla, tím jasnější bude povrch.

Lesklý odraz **S** (*specular light*) = světlo odrážející se od povrchu v daném směru. Díky němu může povrch vypadat leskle.

Základní vzorec pro výpočet výsledné intenzity světla **I** v daném bodě má tedy tvar

$$I = A + D + S \ . \qquad \textbf{(2)}$$

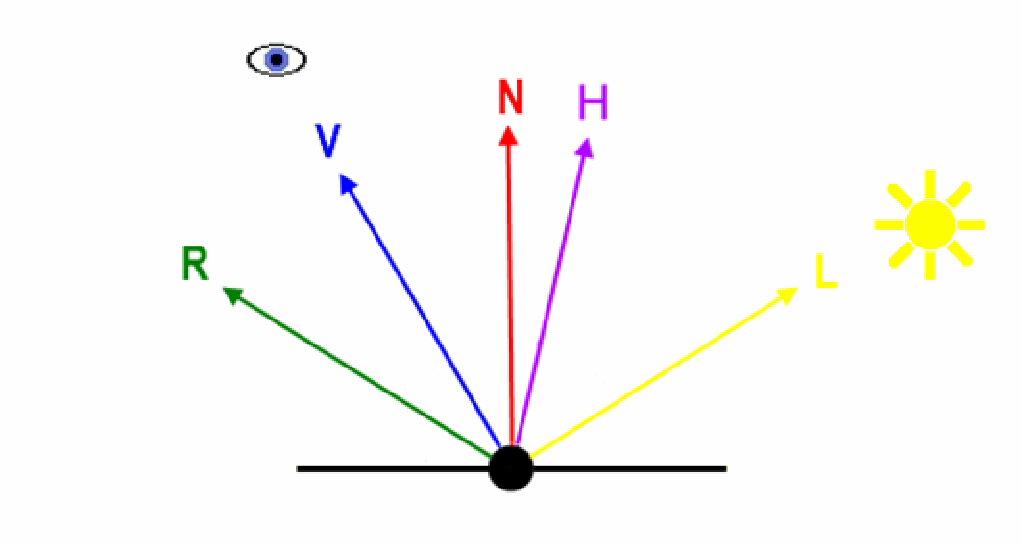
Zatím však nebyla brána v úvahu skutečnost, že intenzita světla se přirozeně zmenšuje se vzdáleností osvětlovaného bodu od světelného zdroje. Zdroj světla si lze představit jako střed koule o poloměru r obsahující všechny body, na které z něj ještě dopadá světlo. Koeficient zeslabení světla (*attenuation*) se často počítá například podle vzorce

$$att = \max\left(1 - \left(\frac{vzd}{r}\right)^2, 0\right), \qquad \textbf{(3)}$$

kde *vzd* je vzdálenost bodu na povrchu od zdroje světla. Další možné přístupy k výpočtu koeficientu zeslabení jsou uvedeny například v článku [7]. Zbytkové světlo není ovlivněno vzdáleností od světelného zdroje a upravený vzorec pro výpočet intenzity světla má pak tvar

$$I = A + att * (D + S) \ . \qquad (4)$$

Následující obrázek zobrazuje vektory, se kterými Phongův model pracuje:



Obrázek č. 9: Vektory ve Phongově osvětlovacím modelu

L … normovaný vektor směřující z bodu na povrchu ke zdroji světla
N … normovaný normálový vektor k povrchu
V … normovaný vektor směřující z bodu na povrchu k pozorovateli
R … normovaný vektor dokonalého (zrcadlového) odrazu
H … normovaný půlící vektor (mezi vektory L a V)

Zbytkové světlo lze získat jednoduše jako součin konstantní ambientní složky světla A_l a konstantní ambientní složky barvy materiálu A_m

$$A = A_l * A_m \ . \qquad (5)$$

Výpočet difusního odrazu je založen na hodnotě funkce kosinus úhlu mezi vektory L a N podle vzorce

$$D = D_l * D_m * \max(L \bullet N, 0) \ , \qquad (6)$$

kde • značí skalární součin vektorů (kosinus úhlu, který svírají dva jednotkové vektory, je možné spočítat jako jejich skalární součin), D_l je difusní barva světla a D_m je difusní barva materiálu. Díky funkci kosinus se intenzita difusní složky světla zvyšuje se zmenšováním úhlu mezi normálou k povrchu v daném bodě a vektorem světla. Funkce maxima ve vzorci zajišťuje správnost výsledku v situaci, kdy je světlo na opačné straně objektu než pozorovatel.

K výpočtu difusní složky světla v daném bodě je tedy třeba znát a vektory a L a N. Vektor světla L se získá jednoduše odečtením souřadnic daného bodu na povrchu od souřadnic pozice světelného zdroje. Zbývá najít

normálový vektor N – ten lze snadno vyčíst z tzv. normálové mapy, které se věnuje následující podkapitola.

Původní výpočet lesklého odrazu (spekulární složky) světla navržený Bui-Tuong Phongem je založený na vektoru odraženého světla. Tento vektor je však třeba nejprve spočíst a vzhledem k relativní náročnosti takového výpočtu se dnes používá spíše modifikace navržená J. F. Blinnem. Blinn navrhl použít tzv. půlící vektor H, který se určuje jako

$$H = \frac{L+V}{|L+V|}.$$

K výpočtu spekulární složky je dále třeba znát vektor V, který se získává odečtením souřadnice pozice sledovaného bodu povrchu od souřadnice pozice kamery ve scéně, a normálový vektor N vyčtený, obdobně jako při výpočtu difusního odrazu, z normálové mapy. Hodnota spekulární složky světla se počítá na základě vzorce

$$S = S_l * S_m * \max(N \bullet H, 0)^n \ , \qquad (7)$$

kde S_l značí spekulární složku světla a S_m spekulární složku barvy materiálu, obě tyto hodnoty bývají uživatelsky nastaveny. Exponent n se nazývá spekulární exponent a určuje lesklost daného povrchu. Při vysoké hodnotě n jsou zrcadlové odlesky menší a ostřejší, zatímco při malé hodnotě jsou odlesky větší a měkčí. Také exponent n bývá nastavován uživatelsky.

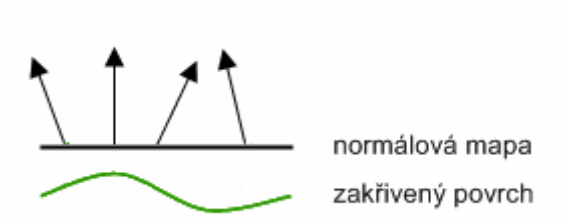
4.1.3 Normálová mapa

Normálová mapa je speciální textura, která uchovává v každém svém bodě informaci o směru normálového vektoru k povrchu v odpovídajícím bodě textury. Při aplikaci běžné 2D textury na rovný povrch vypadají normálové vektory následovně:

Obrázek č. 10: Normálové vektory rovného povrchu

Cílem při realizaci metod pro simulaci zakřivení povrchu je ale simulovat nerovný povrch textury a normálová mapa tak bude obsahovat informace o normálách, které by měl zakřivený povrch zobrazovaný na textuře v reálném světě.

Například:



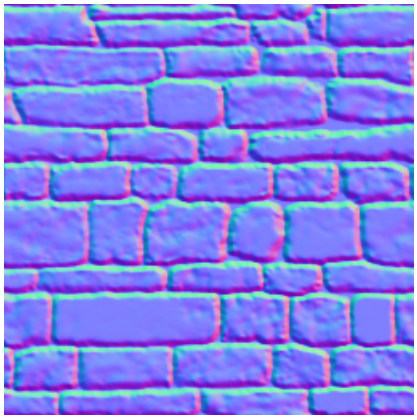
Obrázek č. 11: Normálové vektory zakřiveného povrchu

Směr každého normálového vektoru je v normálové mapě uložen ve formě barvy příslušného pixelu. Normálový vektor v soustavě souřadnic s osou x směřující doprava, osou y směřující nahoru a osou z směřující ven z obrazovky je pak definován třemi souřadnicemi (x, y, z), přičemž všechny náleží do intervalu <-1, 1> (jedná se o normovaný vektor). Barvy jednotlivých pixelů normálové mapy jsou uloženy ve formátu RGB. Přiřadí-li se k ose x hodnoty červené složky, k ose z hodnoty zelené složky a k ose z hodnoty modré složky barvy, je možné uložit souřadnice vektoru jako barvu. Hodnoty jednotlivých složek barvy se ovšem pohybují pouze v intervalu <0, 1>, proto je k získání souřadnic normálového vektoru třeba ještě vypočíst

```
float3 rgbNormal(float3 normal){
      return 2 * (normal – 0.5);
}
```

při zapsání jako funkce Cg.

Jak ukazuje následující příklad, převažující barvou typické normálové mapy je modrá.



Obrázek č. 12: Normálová mapa

Tato vlastnost normálových map je způsobena skutečností, že většina normálových vektorů na běžné textuře směřuje ven z obrazovky, tedy

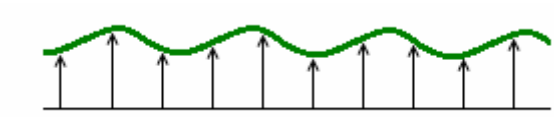
ve směru osy z. Při uložení souřadnic vektoru do barvy pixelu pak dominuje modrá složka barvy.

4.2 Bump mapping a normal mapping

První metodu pro simulaci zakřiveného povrchu (bump mapping) publikoval James F. Blinn již v roce 1978 [8]. Tato metoda vychází z chování stínů v reálném světě. Není-li osvětlovaný povrch plochý, projeví se nerovnosti mimo jiné tak, že hrany obrácené směrem ke zdroji světla budou jasnější a naopak hrany obrácené směrem od zdroje budou tmavé. Cílem bump mappingu je spočíst v každém bodě intenzitu světla tak, aby uvažovala i tento jev.

Základní myšlenka této metody tkví v modulaci normálového vektoru v každém pixelu, resp. texelu na povrchu určitého objektu. Modifikovaná normála se použije pro výpočet osvětlovacího modelu v daném místě a tím se napodobí drobné nerovnosti a hrbolatý povrch bez nutnosti přidání podrobnější geometrie. Normála je modifikována na základě informací ze speciální textury.

Pro každý texel textury zobrazující nějaký nerovný povrch je možné určit vzdálenost, o kterou by byl daný bod ve skutečnosti posunutý nad hladký povrch:



Obrázek č. 13: Posuny bodů nad hladký povrch

Mapa, která má v každém svém pixelu obsaženu informaci o velikosti takového posunu pro odpovídající bod textury, bývá označována jako výšková mapa, případně jako *bump mapa*, mluví-li se o metodách pro simulaci zakřivení povrchu. Podobně jako u výše popsané normálové mapy je informace o délce posunutí uložena ve formě složek RGB barvy pixelu. V tomto případě je však třeba uchovat pouze jedno číslo (nikoliv vektor), a proto se stejná hodnota ukládá do všech barevných složek.

V bump mapě se tak zřejmě vyskytují jen odstíny černé a bílé, jak je vidět například na obrázku:

Obrázek č. 14: Výšková mapa

Metoda bump mappingu popsaná James F. Blinnem používá přímo bump mapu a modifikovaný normálový vektor vypočítává podle gradientu této mapy v příslušném bodě. Je-li povrch definován pomocí tří funkcí X(u,v), Y(u,v) a Z(u,v), pak v libovolném jeho bodě, lze určit směr normálového vektoru k povrchu pomocí vektorového součinu parciálních derivací $\left(\frac{\partial X}{\partial u},\frac{\partial Y}{\partial u},\frac{\partial Z}{\partial u}\right)$ a $\left(\frac{\partial X}{\partial v},\frac{\partial Y}{\partial v},\frac{\partial Z}{\partial v}\right)$. Při realizaci bump mappingu je každý bod povrchu nejprve posunut podél normálového vektoru na základě informace z bump mapy a následně je spočítán modifikovaný normálový vektor jako vektorový součet vektorů parciálních derivací v posunutém bodě. Podrobnosti lze nalézt v článku [8].

Tento původní postup se ovšem dnes již v praxi nepoužívá. Základní myšlenka v současnosti používané metody je stejná, modifikovaný normálový vektor však není počítán na základě bump mapy, ale přímo se získává z normálové mapy. Takto upravená metoda se nazývá normal mapping, často je ale vzhledem ke shodnému principu označována také jako bump mapping.

K výpočtu intenzity světla v určitém bodě je ovšem třeba spočíst skalární součin normálového vektoru a vektoru světla. Je proto ještě nutné vyřešit problém týkající se různých zobrazovacích prostorů. Zatímco normály uložené v normálové mapě jsou definovány v tečném prostoru, světelný vektor je vypočítán v objektovém prostoru. Ze dvou možností – převést všechny normály do objektového prostoru nebo převést jeden vektor světla do tečného prostoru – je zřejmě výhodnější druhá varianta, kterou provedeme pomocí výše popsané TBN matice.

Obdobná situace nastává v případě výpočtu spekulární složky světla pro vektor V směřující k pozorovateli. Vektor V se tedy stejně jako světelný vektor převede pomocí TBN matice do tečného vektoru a poté je znormován.

4.2.1 Implementace metody normal mapping

Nyní již není těžké algoritmus normal mappingu implementovat:

- Pro každý trojúhelník, respektive čtyřúhelník, který je vymezen vrcholy (vertexy) se připraví inverzní TBN matice. Pomocí vypočtené matice je pro každý z vrcholů vektor světla a kamery převeden z objektového prostoru do tečného prostoru. Viz kód vertex shaderu.

- Světelný vektor a vektor kamery jsou znormovány a z normálové mapy je pro každý pixel na povrchu přečten normálový vektor a transformován do intervalu <-1,1>. Pomocí získaného normálového vektoru se na základě Phongova osvětlovacího modelu spočte intenzita světla v daném bodě. Viz kód fragment shaderu.

Cg kód vertex shaderu pro normal mapping:

```
void main(
// pozice vrcholu v objektovém prostoru
    in float4 pozice : POSITION,
// binormála a tangenta
    in float3 binormal,
    in float3 tangent,
// texturovací souřadnice textury
    in float2 texsrd : TEXCOORD0,
// pozice vrcholu v ořezovém prostoru
    out float4 pozice_f : POSITION,
// texturovací souřadnice textury a normálové mapy pro FS
    out float2 texsrd_f : TEXCOORD0,
    out float2 texsrd_normal_f : TEXCOORD1,
// vektor světla a kamery (pozorovatele)
    out float3 svetloVektor : TEXCOORD2,
    out float3 kameraVektor : TEXCOORD3,
// transformační matice z objektového do ořezávacího prostoru
    const uniform float4x4 modelViewProj,
// pozice světla a kamery
    const uniform float3 svetloPozice,
    const uniform float3 kameraPozice)
{
// transformace vrcholu z objektového do ořezávacího prostoru
    pozice_f = mul(modelViewProj, pozice);

// výpočet vektorů světla a a kamery (pozorovatele)
    svetloVektor = svetloPozice - pozice.xyz;
    kameraVektor = kameraPozice - pozice.xyz;

// TBN matice
    float3 normal = cross(tangent,binormal);
    float3x3 tbnMatice = float3x3(  normalize(tangent),
                                    normalize(binormal),
                                    normalize(normal));

// vektor světla a kamery z objektové do tečného prostoru
    svetloVektor.xyz = mul(tbnMatice, svetloVektor);
    kameraVektor.xyz = mul(tbnMatice, kameraVektor);

// přeposlání texturovacích souřadnic pro texturu a normálovou
mapu do FS
    texsrd_f = texsrd;
    texsrd_normal_f = texsrd;
}
```

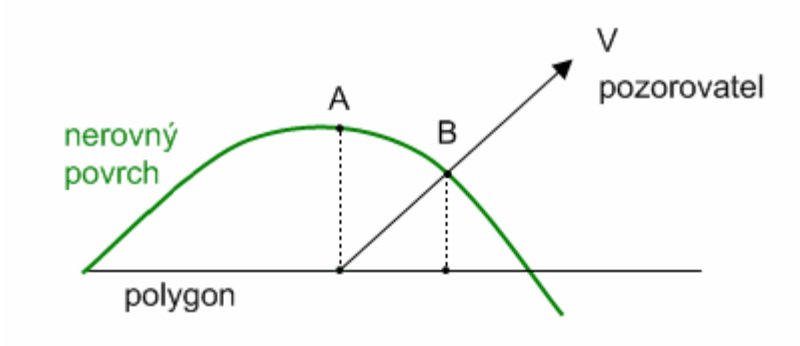
Cg kód fragment shaderu pro normal mapping:

```
void main(
// texturovací souřadnice textury
    in float2 texsrd : TEXCOORD0,
    in float2 texsrd_normal : TEXCOORD1,
// vektor světla a kamery (pozorovatele)
    in float3 svetloVektor : TEXCOORD2,
    in float3 kameraVektor : TEXCOORD3,
// barva fragmentu
    out float3 barva : COLOR,
// textura
    uniform sampler2D tex : TEXUNIT0,
// normálová mapa
    uniform sampler2D tex_normal : TEXUNIT1,
// barva světla
    uniform float3 barvaSvetla,
// spekulární exponent
    uniform float n,
// spekulární koeficient materiálu
    uniform float Sm,
// ambient
    uniform float3 A
)
{
// normalizace vektorů
    svetloVektor = normalize(svetloVektor);
    kameraVektor = normalize(kameraVektor);

// načtení normálového vektoru,
// jeho přepočtení do intervalu (-1, 1) a normalizace
    float3 normal = normalize(2.0f *
      (tex2D(tex_normal, texsrd_normal).rgb - 0.5f));

// phong
    float3 pulVektor = normalize(kameraVektor + svetloVektor);
    float3 D = max(dot(svetloVektor,normal),0)*barvaSvetla;
    float3 S = pow(max(dot(pulVektor,normal),0),n) *
               Sm * barvaSvetla;
    barva.rgb = (tex2D(tex,texsrd).rgb*(D+A))+S;

}
```

4.3 *Parallax mapping*

Jednou z nevýhod bump mappingu, která limituje možnosti této metody při simulaci nerovných povrchů, je neschopnost zobrazit tzv. parallax efekt. Jedná se o jev, který lze pozorovat na jakémkoliv neplochém povrchu – při pohybu pozorovatele vypadají jednotlivé části povrchu, jako by se vůči sobě navzájem pohybovaly. Tento problém řeší metoda parallax mapping, známá také jako offset texture mapping nebo virtual displacement mapping, kterou představil Tomomichi Kaneko v roce 2001 [9].

Základní myšlenku parallax mappingu lze nejlépe znázornit pomocí obrázku:

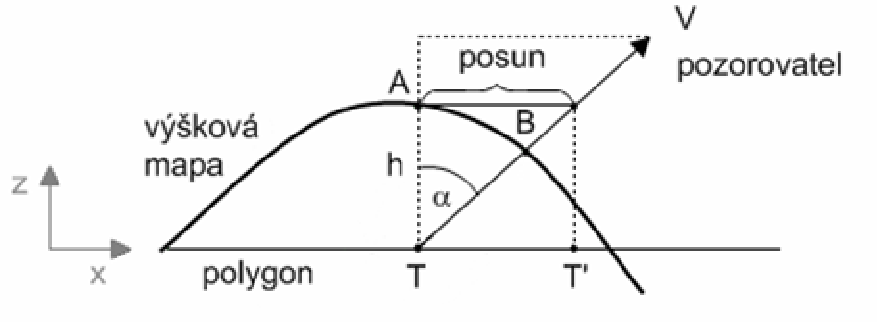


Obrázek č. 15: Základní myšlenka parallax mappingu

Zobrazený směrový vektor V k pozorovateli názorně ukazuje, že při pohledu na plochý polygon, na který je ve scéně aplikována textura nerovného povrchu, uvidí pozorovatel bod textury odpovídají bodu A, zatímco v reálném světe by při pohledu na skutečně nerovný povrch sledoval bod B. K dosažení věrohodnější simulace povrchu by tedy zřejmě bylo výhodné upravit souřadnice textury tak, aby v bodě A povrchu byl zobrazen texel odpovídající souřadnicemi bodu B povrchu. Při aplikaci analogického postupu na každý bod textury se budou vyšší oblasti simulovaného povrchu posouvat směrem od pozorovatele a nižší oblasti naopak směrem k pozorovateli. Tím je docíleno simulace parallax efektu.

Zbývá určit, o kolik a jakým směrem se mají souřadnice v konkrétním případě posunout. Metoda parallax mapping využívá informace obsažené ve výškové mapě (popsána v kapitole o bump mappingu).

Opět je nejvhodnější vyjít z obrázku:



Obrázek č. 16: Výpočet posunu texelu

Je třeba si uvědomit, že vzhledem k dvourozměrnosti zobrazuje obrázek pouze souřadnice na ose x a na ose z. Pro souřadnice na ose y jsou však obrázek i následující postup zcela analogické. Z obrázku je zřejmé, že

$$posun = \tan(\alpha) * h \ , \qquad \textbf{(8)}$$

kde h je hodnota získaná z výškové mapy pro souřadnice bodu T. Výšková mapa uchovává hodnoty pouze v intervalu <0, 1>, proto se přečtená výška h často upravuje tak, aby hodnoty lépe odpovídaly skutečnému vzhledu povrchu. Je pak $h' = h * s + p$, kde s a p jsou vhodné koeficienty škálování a posunutí.

Dále lze vyjádřit

$$posun = \frac{V_{(x)}}{V_{(z)}} * h', \qquad \textbf{(9)}$$

kde $V_{(x)}$ je x-ová souřadnice vektoru V a $V_{(z)}$ je z-ová souřadnice vektoru V.

Při finálním zobrazení povrchu pak bude algoritmus pro vykreslení pixelu povrchu se souřadnicemi bodu T pracovat s hodnotami (texturové souřadnice, souřadnice v normálové mapě) pro bod T':

$$T' = T + posun \, . \qquad \textbf{(10)}$$

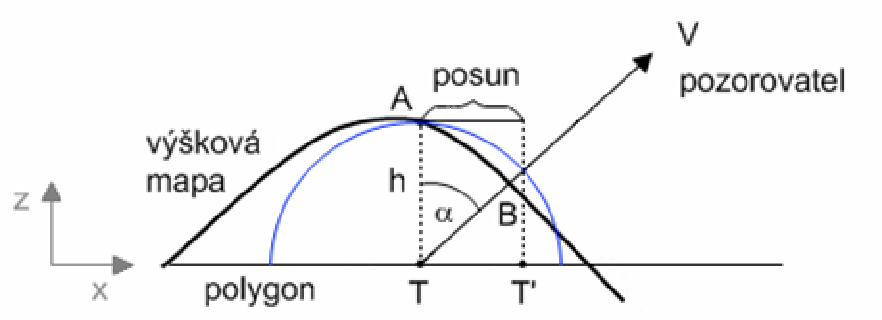
4.3.1 Vylepšení metodou offset limiting

Na obrázku č. 16 si lze povšimnout, že uvedený postup získání posunu souřadnic texelu má jednu vadu. K přesnému určení požadovaného posunutí by totiž bylo třeba, aby výška h odpovídající bodu T' byla shodná s výškou odpovídající bodu T. To samozřejmě není u většiny povrchů splněno. Řešení tohoto problému vychází z myšlenky, že bude-li délka posunutí malá, bude u většiny povrchů i rozdíl mezi příslušnými výškami malý, a taková

nepřesnost výpočtu neovlivní věrohodnost konečného zobrazení povrchu. Stačí tedy omezit délku vypočteného posunutí nějakou hodnotou, jak uvádí [10] vhodná je například hodnota h'.

Obrázek č. 17 ukazuje, že při použití omezující hodnoty h' lze pro výpočet posunu souřadnic použít zjednodušený vzorec

$$posun = V_{(x)} * h' \,. \qquad \textbf{(11)}$$



Obrázek č. 17: Výpočet posunu v případě použití offset limiting

4.3.2 Implementace metody parallax mapping

- Pro každý trojúhelník, respektive čtyřúhelník, který je vymezen vrcholy (vertexy) se připraví inverzní TBN matice. Pomocí vypočtené matice je pro každý z vrcholů vektor světla a kamery převeden z objektového prostoru do tečného prostoru. Cg kód vertex shaderu je totožný s metodou normal mapping.

- Světelný vektor a vektor kamery jsou znormovány. Z výškové mapy je pro každý pixel na povrchu přečtena výška h, která je dále upravena pomocí koeficientu škálování param1 a posunutí param2. Na základě zjednodušeného vzorce po aplikaci vylepšení offset limiting se spočítá posun. S použitím nově získaných souřadnic a normály z normálové mapy je na základě Phongova osvětlovacího modelu spočtena intenzita světla v daném bodě. Viz kód fragment shaderu.

Cg kód fragment shaderu pro parallax mapping:

```
void main(
// texturovací souřadnice textury a normálové mapy
    in float2 texsrd : TEXCOORD0,
    in float2 texsrd_normal : TEXCOORD1,
// vektor světla a kamery (pozorovatele)
    in float3 svetloVektor : TEXCOORD2,
    in float3 kameraVektor : TEXCOORD3,
// barva fragmentu
    out float3 barva : COLOR,
// textura
    uniform sampler2D tex : TEXUNIT0,
// normálová mapa
    uniform sampler2D tex_normal : TEXUNIT1,
// výšková mapa
    uniform sampler2D tex_height : TEXUNIT2,
// barva světla
    uniform float3 barvaSvetla,
// parametr pro nastavení škálovacího faktoru
    uniform float param1,
// parametr pro nastavení posunutí
    uniform float param2,
// spekulární exponent, spekulární koeficient materiálu
    uniform float n,
    uniform float Sm,
// ambient
    uniform float3 A)
{
// normalizace vektorů
    svetloVektor = normalize(svetloVektor);
    kameraVektor = normalize(kameraVektor);
// načtení normálového vektoru,
// jeho přepočtení do intervalu (-1, 1) a normalizace
    float3 normal = normalize(2.0f *
      (tex2D(tex_normal, texsrd_normal).rgb- 0.5f));

// načtení výšky
    float4 vyska = tex2D(tex_height,texsrd);

// úprava hloubky škálovacím faktorem a parametrem posunutí
    float hloubka = vyska.r * param1 - param2;

// posun texturovacích souřadnic
    texsrd.x += hloubka * kameraVektor.x;
    texsrd.y -= hloubka * kameraVektor.y;

// phong
    float3 pulVektor = normalize(kameraVektor + svetloVektor);
    float3 D = max(dot(svetloVektor,normal),0)*barvaSvetla;
    float3 S = pow(max(dot(pulVektor,normal),0),n) *
               Sm * barvaSvetla;
    barva.rgb = (tex2D(tex,texsrd).rgb*(D+A))+S;
}
```

Kapitola 5

Displacement mapping

Displacement mapping je pokročilá technika pro zvýšení úrovně detailů na povrchu objektu. Pomocí výškové textury zvané displacement mapa, která obsahuje informace potřebné k posunu vrcholů, upraví objekt s nízkými detaily a zvýší úroveň detailů objektu. V extrémním případě je možné pomocí displacement mapy vymodelovat celý objekt přímo na GPU.

Výhoda displacement mappingu oproti normal mappingu a parallax mappingu spočívá v tom, že ve skutečnosti mění geometrii objektu, takže je na výsledném povrchu objektu realističtěji vypočteno stínování, správně zobrazen parallax efekt a při pohledu „ze strany“ lze pozorovat změněnou siluetu odpovídající skutečnému vzhledu povrchu.

Autor paralax mappingu Kaneko ve svém článku z roku 2001 zdůraznil, že displacement mapping potřebuje generovat velké množství polygonů a není vhodný pro real-time počítačovou grafiku. Dnes je však možné i displacement mapping, který ve skutečnosti mění povrch objektu, implementovat v reálném čase, ovšem až na nejmodernějších grafických kartách. Již zmíněná nová vlastnost moderních grafických karet, a to podpora přístupu vertex shaderu do paměti textur, totiž umožňuje nový přístup k implementaci metody displacement mapping a její efektivní a plynulé použití v aplikacích v reálném čase.

5.1 Základní princip vertex displacement mappingu

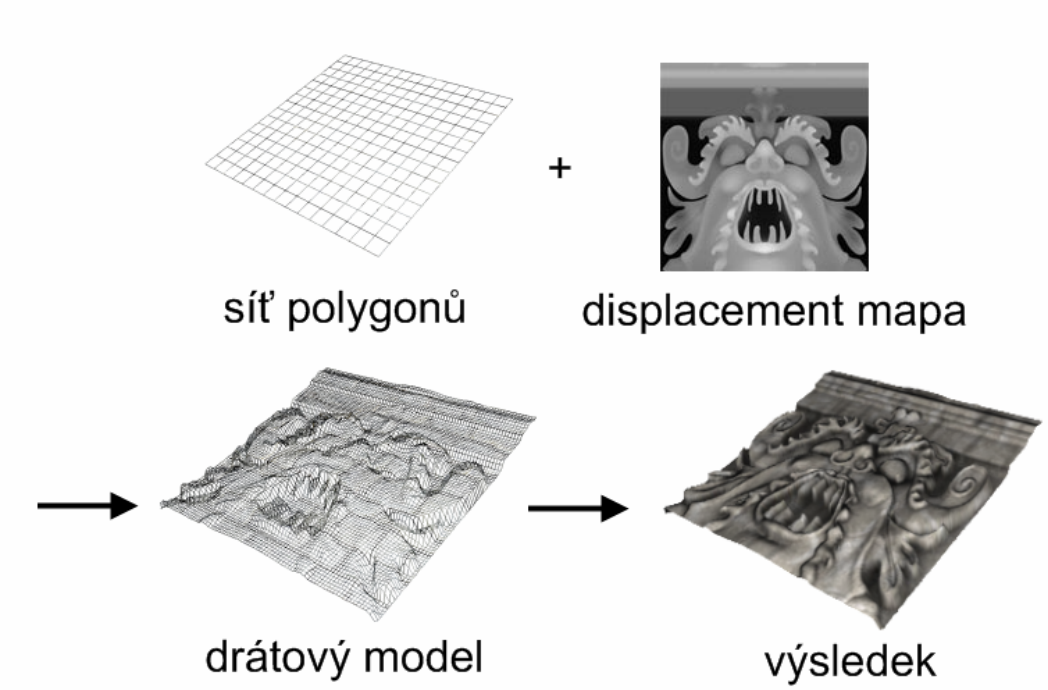
Model objektu ve scéně je tvořen polygony, které jsou vymezeny vrcholy. Cílem metody displacement mapping je zvýšit úroveň geometrických detailů modelu, a proto je zřejmě třeba zvýšit počet vrcholů, na základě nichž je objekt vykreslován. Polygony v modelu jsou tedy dále rozděleny na menší polygony a vznikají tak nové vrcholy. Původní i nově vzniklé vrcholy jsou pak ve vertex shaderu posunuty o určitou vzdálenost ve směru normálového vektoru k povrchu v daném bodě.

K určení velikosti posunu používá displacement mapping výškovou mapu, v níž jsou pro každý vrchol uloženy informace o délce posunutí jednotlivých vrcholů podél normálového vektoru. Tato textura se nazývá displacement mapa.

Je-li z displacement mapy přečtena velikost posunutí d pro daný vrchol, je nová pozice V' vrcholu spočtena podle vzorce

$$V' = V + (N * D * s) \qquad \textbf{(12)}$$

kde V je původní pozice vrcholu a s je uživatelem nastavený koeficient škálování. Výsledkem je drátový model objektu, na který je dále aplikována textura povrchu.



Obrázek č. 18: Postup displacement mappingu

Zřejmým problémem metody displacement mapping je určení normálových vektorů pro výpočet osvětlení. Původní normály z neupraveného modelu již po změně geometrie neplatí. Protože ale k posunutí vrcholu dochází ve vertex shaderu a ten nemá přístup k ostatním vrcholům, není možné normály přepočítat, aby odpovídaly změněnému modelu.

Jak je uvedeno v [11], je tento problém možné řešit dvěma způsoby. Jednou možností je normálové vektory vůbec nepoužívat a k určení osvětlení využívat tzv. light mapu, to znamená vypočítat osvětlení jednotlivých pixelů předem a uložit je do speciální textury. Tento postup lze ale použít pouze u statických scén. Druhou možností vhodnou i pro pohyblivé scény je

k výpočtu osvětlení používat stejně jako u normal mappingu normálovou mapu.

5.1.1 Implementace metody displacement mapping

- Pro každý trojúhelník, respektive čtyřúhelník, který je vymezen vrcholy (vertexy) se připraví inverzní TBN matice. Pomocí vypočtené matice je pro každý z vrcholů vektor světla a kamery převeden z objektového prostoru do tečného prostoru. Z displacement mapy je načtena velikost posunu a vrchol je o zjištěnou vzdálenost posunut podél normálového vektoru. Viz kód vertex shaderu.
- Světelný vektor a vektor kamery jsou znormovány. a z normálové mapy je pro každý pixel na povrchu přečten normálový vektor a transformován do intervalu <-1,1>. Pomocí získaného normálového vektoru se na základě Phongova osvětlovacího modelu spočte intenzita světla v daném bodě. Cg kód fragment shaderu je totožný s metodou normal mapping.

Cg kód vertex shaderu pro displacement mapping:

```
void main(
// pozice vrcholu v objektovém prostoru
    in float4 pozice : POSITION,
// texturovací souřadnice textury
    in float2 texsrd : TEXCOORD0,
// normálový vektor
    in float3 normal : NORMAL,
// pozice vrcholu v ořezovém prostoru
    out float4 pozice_f : POSITION,
// texturovací souřadnice textury a normálové mapy pro FS
    out float2 texsrd_f : TEXCOORD0,
    out float2 texsrd_normal_f : TEXCOORD1,
// vektor světla a kamery (pozorovatele)
    out float3 svetloVektor : TEXCOORD2,
    out float3 kameraVektor : TEXCOORD3,
// displacement mapa
    uniform sampler2D tex_height : TEXUNIT2,
// transformační matice z objektového do ořezávacího prostoru
    const uniform float4x4 modelViewProj,
// pozice světla a kamery
    const uniform float3 svetloPozice,
    const uniform float3 kameraPozice,
// parametr pro nastavení posunu
    uniform float param1
)
{
// spočtení posunu vrcholu
    float3 posun = tex2D(tex_height, texsrd).xyz * param1;

// posun pozice vrcholu podél normálového vektoru
    pozice.xyz = normal*posun + pozice.xyz;

// transformace posunutého vrcholu
// z objektového do ořezávacího prostoru
    pozice_f = mul(modelViewProj, pozice);

// výpočet vektorů světla a a kamery (pozorovatele)
    svetloVektor = svetloPozice - pozice.xyz;
    kameraVektor = kameraPozice - pozice.xyz;

// svetloVektor, kameraVektor -> TBN matice -> ořezový prostor
// … viz vertex shader k metodě normal mapping

// přeposlání texturovacích souřadnic
// pro texturu a normálovou mapu do FS
    texsrd_f = texsrd;
    texsrd_normal_f = texsrd;
}
```

5.2 Použití displacement mappingu

Použitím metody displacement mapping lze při zobrazování nerovných povrchů dosáhnout kvalitních a zajímavých efektů. Velmi dobré výsledky je možné získat například při simulaci vodního povrchu. Rozsáhlé možnosti se do budoucna skrývají také v interaktivně kontrolovaném posunutí vrcholů, které je metodou prováděno.

Výhodou displacement mappingu při srovnání s využitím detailního geometrického modelu je relativně malý počet použitých polygonů, díky čemuž je ulehčena práce grafické karty se zpracováním vrcholů. Všechny informace o pozicích vrcholů jsou navíc uloženy v 8-bitové nebo 16-bitové 2D textuře a tím jsou značně sníženy nároky na paměť a paměťovou sběrnici grafické karty při vytváření modelu.

Displacement mapping má ovšem i některá omezení. Nelze ho například dobře použít na objekty s velkým množstvím úhlových zlomů nebo na vysoce členité povrchy, jako jsou například stromy nebo tráva. Použití displacement výškové mapy k určení velikosti posunu vrcholu omezuje možnosti displacement mappingu při zobrazování povrchů se záhyby (například látky, oblečení), neboť v takovém případě by jeden bod mapy odpovídal dvěma texelům v původním modelu.

Nevýhodou je také skutečnost, že nově vytvořené vrcholy jsou uloženy v paměti grafické karty a je tak omezena použitelnost některých algoritmů pracujících s vrcholy z paměti systému (například využití stínových těles). Dále je tu již výše popsaný problém s neznámými normálovými vektory změněného modelu a omezující požadavek grafické karty s podporou přístupu vertex shaderu do paměti textur.

5.3 Vylepšení pomocí testu viditelnosti

Jak již bylo zmíněno, operace čtení dat z textury ve vertex programu, na kterém je metoda displacement mappingu založená (skutečný posun vrcholů), je velmi „drahá“ operace. Na novějších grafický kartách podporujících dynamické větvení programů na GPU, lze proto výpočet při zobrazování objektu urychlit jednoduchým testem provedeným ve vertex shaderu, který zjistí, zda je daný vrchol vůbec možné vidět na obrazovce.

Vertex program je běžně prováděn na všech vrcholech, nevyjímaje ty, které jsou ve fázi ořezávání zahozeny (tj. jsou mimo záběr kamery). Pokud tedy test zjistí, že daný vrchol bude v konečné fázi ořezán, urychlí výpočet

přeskočením dalších zbytečných operací ve vertex shaderu včetně čtení dat z textury.

Cg kód rozšíření vertex shaderu pro displacement mapping implementující test viditelnosti:

```
// transformace vrcholu z objektového do ořezávacího prostoru
pozice_f = mul(modelViewProj,pozice);

// test viditelnosti
float3 bClip = abs(pozice_f.xyz) < (pozice_f.www + p);

// když je vrchol v pohledovém tělese
if (all(bClip)){
      // displacement mapping
}
```

Parametr p určuje šířku pásu kolem pohledového tělesa, v kterém se vrchol ještě počítá, přestože už není vidět na obrazovce. Tato tolerance je nutná, neboť pozice vrcholu včetně posunutí ovlivňuje sousední vrcholy, které mohou být ještě vidět na okraji obrazu.

5.4 Optimalizace pomocí úprav textur

Při aplikaci textury na objekt může dojít k situaci, kdy nebude pro každý zobrazovaný pixel existovat odpovídající bod textury a bez provedení vhodných úprav by se tak při zobrazení na povrchu objevily mezery. V trojrozměrném prostoru mohou navíc na zobrazovaném povrchu vznikat různé vizuální chyby i v důsledku pohybu daného objektu. Obraz objektu na obrazovce se totiž zmenšuje nebo zvětšuje v závislosti na tom, jak je zobrazovaný objekt daleko od kamery. Stejně tak se musí zmenšovat, zvětšovat nebo jinak deformovat i aplikovaná textura, což vede k výskytu artefaktů a skokových přechodů. Při použití více textur na daný povrch, je tento problém třeba řešit u všech textur, to znamená u metod pro zvyšování úrovně detailů nerovných povrchů je třeba věnovat pozornost nejen povrchové textuře, ale i normálové a případně výškové mapě.

5.4.1 Filtrování textur

Nejjednodušším způsobem, jak zabránit výskytu různých chyb a artefaktů, je aplikovat na textury vhodný typ filtrace. Základní typy filtrací jsou metoda nejbližšího souseda (vybere se nejbližší možný texel k zobrazovanému pixelu) a bilineární filtrace (spočte se vážený průměr z pole 2x2 texelů, které leží nejblíže středu zobrazovaného pixelu). V OpenGL lze

filtraci textur metodou nejbližšího souseda nebo bilineární filtraci jednoduše zajistit jedním příkazem.

Při implementaci metody displacement mapping je však situace o něco obtížnější. Při přístupu k textuře z vertex shaderu závisí možnost využití konkrétního typu filtrace na používané grafické kartě, například u GeForce řady 6x00 je podle [12] podporováno pouze filtrování metodou nejbližšího souseda. Použití jiného typu filtrování značně zpomalí výpočet, použije se totiž hardwarově neakcelerované vykreslování (softwarové vykreslování). Naštěstí je možné bilineární filtraci displacement mapy speciálně naprogramovat a provést ji přímo ve vertex shaderu.

Cg kód rozšíření vertex shaderu pro displacement mapping implementující bilineární filtraci:

```
// je nutné definovat velikost textury a velikost texelu,
// například jako uniform proměnné nastavené aplikací

// výpočet posunu vrcholu je nahrazen následujícím kódem

// bilineární filtrace
float2 f = frac(texsrd.xy * velikostTextury);
float4 t00 = tex2D(tex_height,texsrd);
float4 t10 = tex2D(tex_height,texsrd +
      float2(velikostTexelu,0.0f));
float4 tA = lerp(t00,t10,f.x);

float4 t01 = tex2D(tex_height,texsrd +
      float2(0.0f,velikostTexelu));
float4 t11 = tex2D(tex_height,texsrd +
float2(velikostTexelu,velikostTexelu));
        float4 tB = lerp(t01,t11,f.x);

// spočtení posunu vrcholu
float3 posun = lerp(tA,tB,f.y).xyz * param1;
```

Nevýhodou je, že potřebný čtyřnásobný přístup k textuře z vertex shaderu zpomalí celkový výpočet. Vzhledem k tomu, že se jedná o výškovou mapu, lze několikanásobné čtení z textury obejít trikem uvedeným v [13], a to uložením 4 pixelů (2 x 2) z původní výškové mapy do jednotlivých RGBA složek 1 pixelu bez ztráty informace.

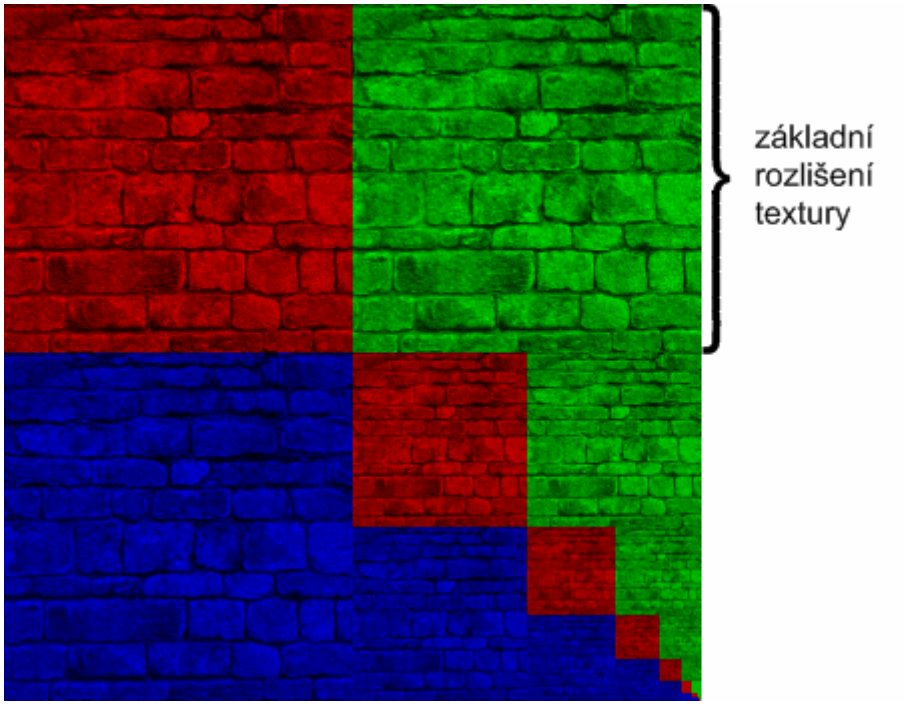
5.4.2 Mipmapping

Na některé vizuální chyby vzniklé při zobrazování pohybujícího se objektu však již filtrace textur nestačí. Například na vzdálených objektech je textura velmi zmenšená a dochází k viditelnému poblikávání pixelů. Navíc neustálá

změna velikosti textury snižuje výpočetní výkon dané aplikace. V takovém případě se používá metoda zvaná mipmapping[8].

Základní myšlenka mipmappingu spočívá v uložení používané textury ve více rozlišeních, to znamená s různou úrovní detailů. Při zobrazování objektu v různých vzdálenostech od kamery se pak podle aktuální velikosti sledovaného polygonu, resp. podle vzdálenosti bodu od kamery vybere nejvhodnější předpočítané rozlišení textury a aplikuje se na daný polygon objektu.

Jednotlivá rozlišení textury bývají uložena v textuře nazývané mipmapa, která je vytvořena v paměti grafické karty. Po připomenutí, že barva bodů textury je definována třemi složkami formátu RGB, následující obrázek jasně znázorňuje, jakým způsobem je textura v mipmapě reprezentována:



Obrázek č. 19: Mipmapa

Základní nezměněné rozlišení textury, které se použije, je-li objekt blízko ke kameře, bývá označováno jako úroveň 0. Každé další rozlišení uložené v mipmapě obsahuje vždy čtvrtinu texelů předchozího rozlišení až k rozlišení textury o velikosti 1 pixel. Druhá nejdetailnější reprezentace textury se označuje jako úroveň 1, třetí jako úroveň 2, atd. Z textury v základním rozlišení se textury v nižším rozlišení získávají filtrací.

Výše popsaný základní postup při mipmappingu, kdy se na daný polygon aplikuje nejbližší vhodné rozlišení textury, má nepříjemnou nevýhodu. Při pohybu objektu od kamery nebo ke kameře dochází

[8] mip pochází z latinského multum in parvo = mnohé v malém

ke skokovým změnám v zobrazení ve chvíli, kdy polygon přejde od jednoho rozlišení textury ke druhému. Často se proto využívá vylepšení metody a barvy pixelů se počítají pomocí lineární interpolace odpovídajících pixelů z nejbližšího většího a nejbližšího menšího rozlišení.

Stejně jako filtraci lze aplikování metody mipmapping na používané textury v OpenGL zajistit jednoduchým příkazem. U metody displacement mapping ale nastává problém v důsledku toho, že výšková mapa se využívá pro posunutí vrcholů již ve vertex shaderu, ve kterém není ještě známa vzdálenost vrcholu od kamery a automaticky je vybrána úroveň 0 displacement mapy. Řešením je spočíst vzdálenost vrcholu od kamery pomocí vlastního kódu a správnou barvu pixelu odpovídajícího danému vrcholu (v případě displacement mapy správnou výšku, resp. posun) vypočítat přímo ve vertex shaderu.

Cg kód rozšíření vertex shaderu pro displacement mapping implementující mipmapping s lineární interpolací:

```
// spočtení úrovně mipmapy na základě z-ové souřadnice vrcholu
float mipmapLevel = (pozice_f.z/pozice_f.w) * maxMipmapLevel;

// určení nejbližší menší a nejbližší větší mipmap úrovně
float mipmapLevelDolni = floor(mipmapLevel);
float mipmapLevelHorni = mipmapLevelDolni+1;
float mipmapLevelFrac = frac(mipmapLevel);

//načtení texelů z mipmap úrovní
float4 posunDolni = tex2Dbias(tex,
      float4(texsrd,mipmapLevelDolni,mipmapLevelDolni));
float4 posunHorni = tex2Dbias(tex,
      float4(texsrd, mipmapLevelDolni, mipmapLevelDolni));

// spočtení posunu vrcholu interpolací mipmap úrovní
float4 posun = lerp(posunDolni, posunHorni, mipmapLevelFrac);
```

5.4.3 Optimalizace výpočtu zrcadlových odlesků

Při filtraci nebo aplikaci metody mipmapping na normálovou mapu dochází k průměrování a interpolacím jednotkových normálových vektorů. Pokud si nejsou všechny normálové vektory rovny, nemají výsledné vektory jednotkovou délku. Tento jev má mimo jiné vliv na výpočet difusní a spekulární složky intenzity světla. Většinou se proto výsledné normálové vektory po provedení všech výpočtů znovu normují.

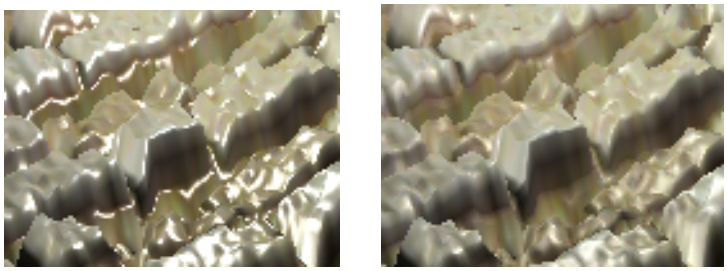
Michael Toksvig v článku [14] navrhl vylepšený postup, který zlepšuje zejména výsledný vzhled zrcadlových odlesků. Při obvyklém postupu s opětovným znormováním výsledného normálového vektoru N by se spekulární složka světla (viz kapitola 4.1.2) počítala na základě vzorce

$$S = S_l * S_m * \max\left(\frac{N \bullet H}{|N|}, 0\right)^n . \qquad \textbf{(13)}$$

Toksvig představil tzv. Toksvigův faktor *f*, kterým se upraví spekulární exponent a spekulární složka světla se vypočte pomocí vztahu

$$S = S_l * S_m * \frac{1 + f * n}{1 + n} * \max\left(\frac{N \bullet H}{|N|}, 0\right)^{f*n} , \qquad \textbf{(14)}$$

kde $f = \frac{|N|}{|N| + n * (1 - |N|)}$. Je dobré si povšimnout, že při výpočtu pro normálový vektor s jednotkovou délkou bude platit $f = 1$ a spekulární složka zůstane ve srovnání s obvyklým postupem nezměněna.



Obrázek č. 20: Porovnání detailu zobrazení metodou displacement mapping s bilineární filtrací normálové mapy bez použití Toksvigova faktoru (vlevo) a s Toksvigovým faktorem (vpravo)

5.5 Komprese normálových map

Při použití metod pro zvyšování úrovně detailů nerovných povrchů ve složitějších scénách, například ve hrách, je třeba vedle běžných textur s obrázkem povrchu uchovávat také normálové a výškové mapy. Ke snížení paměťových nároků aplikací se nabízí tyto textury zkomprimovat. V OpenGL lze ke kompresi textur použít metodu S3 Texture Compression (S3TC) nazývanou také DXT komprese. Existuje pět typů této komprese, a to DXT1 až DXT5. Tyto metody komprese poskytují dobré výsledky v případě běžných barevných textur, nemusí však stejně dobře fungovat pro normálové a výškové mapy.

Například algoritmus DXT1 používaný pro kompresi textur obsahujících zcela průhledné části je pro kompresi normálových map

ve většině případů nepoužitelný. Použije-li se takto komprimovaná normálová mapa pro některou metodu pro zvyšování úrovně detailů nerovných povrchů, výsledný povrch bude obsahovat obdélníkové artefakty.

Algoritmus DXT5 je pro kompresi normálových map možné použít po menší úpravě. Výhodou varianty DXT5 oproti DXT1 je podpora uchování alfa kanálu, který je komprimován samostatně odděleně od RGB složek barvy. Vzhledem k tomu, že alfa kanál se u normálových map nevyužívá, je výhodné do něj při kompresi uložit hodnotu červené složky. K dalšímu zlepšení může přispět dopočítání modré složky. Normálový vektor (x, y, z) uložený v normálové mapě má totiž vždy nezápornou složku z a jednotkovou délku. Díky tomu je ze složek x a y možné dopočítat $z = \sqrt{1 - x^2 - y^2}$.

Tímto způsobem lze dosáhnout poměrně ucházejících výsledků. Jak uvádí [15], algoritmus DXT5 je výhodné použít u těch aplikací, kde nejvíce záleží na zmenšení paměťových nároků i za cenu výskytu drobných chyb při zobrazování. Je ovšem nutné poznamenat, že v případě dopočítávání modrých složek není možné použít Toksvigovo vylepšení výpočtu spekulární složky, neboť normálový vektor má pak vždy jednotkovou délku.

Firma ATI navrhla pro kompresi normálových map nový formát nazvaný 3Dc. Algoritmus 3Dc využívá také možnosti dopočítání třetí složky normálových vektorů a poskytuje kvalitní výsledky, tuto kompresi je však možné využít jen na grafických kartách firmy ATI, a to u ATI X800 a lepších.

I u aplikací, které si nemohou dovolit zmenšení kvality výsledného obrazu, je možné ušetřit paměť použitím dvousložkového formátu k uložení normálových map. Příkladem je formát A8L8. Červená a zelená složka jsou uloženy v textuře a modrá složka je dopočítána stejně jako u DXT5 algoritmu. Je tak možné uchovat nezkomprimovanou normálovou mapu ve dvou bytech na texel namísto tří bytů.

Kapitola 6

Program Advanced Mapping Techniques a srovnání metod

6.1 Program Advanced Mapping Techniques

Program Advanced Mapping Techniques vytvořený v rámci této práce, demonstruje diskutované techniky zobrazovaní povrchových detailů a jejich vylepšení v praxi. Skládá se z několika částí. Grafické uživatelské prostředí (GUI) je napsáno v jazyce Java za použití vývojového prostředí NetBeans 6. GUI, které je díky jazyku Java nezávislé na platformě, používá k vizualizaci metod knihovnu JOGL (Java bindings for OpenGL, http://jogl.dev.java.net), která zpřístupňuje grafické API OpenGL z programovacího jazyka Java včetně hardwarové 3D akcelerace, rozšíření OpenGL od různých výrobců grafický karet a umožňuje integrovat OpenGL do Java GUI (AWT, Swing). K aplikaci je dále připojeno několik vertex a fragment programů naprogramovaných v jazyce Cg, které implementují části algoritmů pracujících na vertex a fragment procesorech v GPU.

Program pro svůj běh vyžaduje minimálně grafickou kartu NVIDIA GeForce 6600 nebo ATI HD 2x00 a lepší. Je to dáno potřebou funkce přístupu vertex programů do paměti textur, což umožňují až moderní grafické karty. Pro plynulý běh metody displacement mapping je doporučena grafická karta NVIDIA GeForce 8600.

6.1.1 Spuštění programu

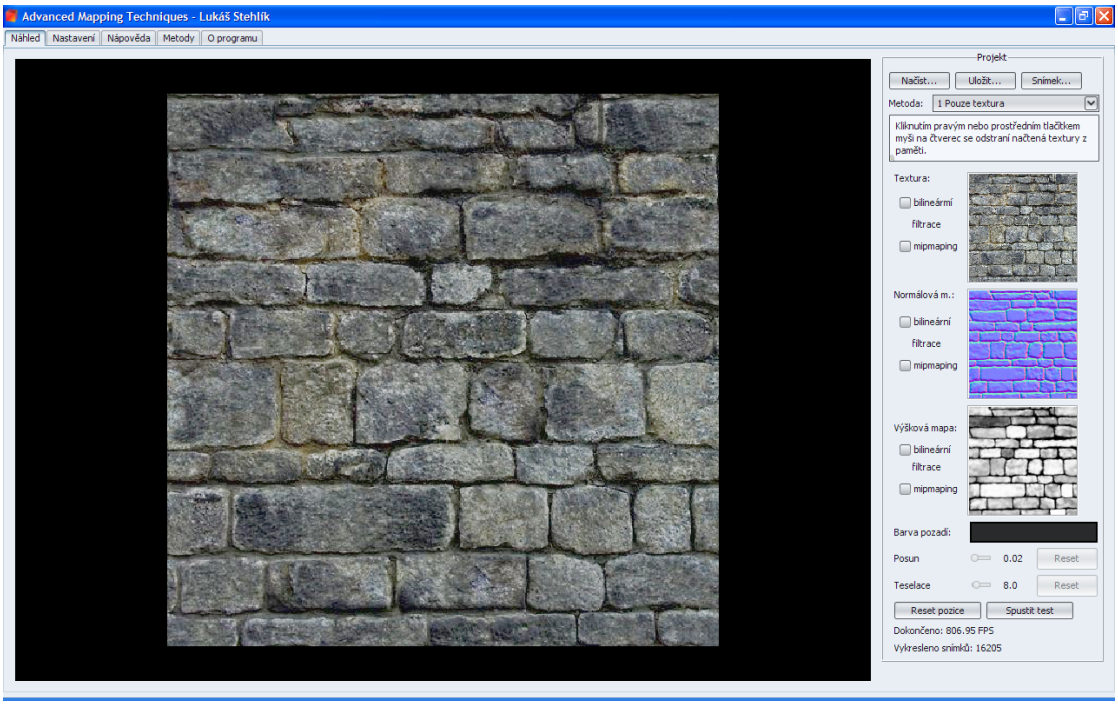
Program Advanced Mapping Techniques se pod operačním systémem Windows spustí pomocí souboru AdvMapTech.exe. Jedná se o tzv. *wrapper*, který lze spustit i bez nainstalovaného jazyka Java na rozdíl od Java souborů JAR. Wrapper inicializuje běhové prostředí Java, které je přibaleno k aplikaci v adresáři jre (Java Runtime Environment 1.6.0_03) a spustí integrovaný

AdvMapTech.jar. Požadované dynamické knihovny jazyka Cg jsou připraveny u programu.

Na ostatních operačních systémech (Linux, Solaris, …) je nutné mít nainstalované běhové prostředí JRE 6 (resp. 1.6) a Cg, knihovna JOGL je připravena u aplikace. Program se spustí pomocí souboru AdvMapTech.jar.

6.1.2 Grafické prostředí

Po spuštění programu Advanced Mapping Techniques se otevře hlavní okno aplikace (viz následující obrázek) a automaticky se načte ukázkový projekt „Kamenný povrch“.



Obrázek č. 21: Hlavní okno programu Advanced Mapping Techniques

Projekt v Advanced Mapping Techniques reprezentuje nastavení cest k texturám pro metodu pro zobrazování povrchových detailů a barvu pozadí. Parametry projektu lze měnit v pravém panelu aplikace na první záložce Náhled – panel Projekt. Panel pro nastavení projektu dále obsahuje ovládací prvky (tlačítka) pro načítání a ukládání projektu do souboru pro snadné pozdější použití, pro resetování pozice objektu a spuštění testu rychlosti. Velkou část panelu vyplňují náhledy používaných textur (textura, normálová mapa, výšková mapa). Největší část hlavního okna aplikace tvoří interaktivní 3D náhled zobrazovaného povrchu.

Ovládání 3D náhledu

Interaktivní 3D náhled je ovládán zejména pomocí myši:

Otáčení objektu
... levé tlačítko myši + pohyb kurzorem

Posun objektu
... pravé tlačítko myši + pohyb kurzorem

Přiblížení/oddálení objektu
... levé a pravé tlačítko myši + pohyb kurzorem dolů/nahoru

a dále pomocí GUI nebo klávesových zkratek:

1 - 9 výběr zobrazovací metody,
B dialog pro výběr barvy pozadí,
L zapnutí / vypnutí zobrazení pozice světla,
R reset pozice objektu, objekt je umístěn do základní polohy,
S snímek obrazovky uložený na disk do obrázku ve formátu PNG,
T spuštění testu, viz níže popis k tlačítku „Spustit test“,
W zapnutí / vypnutí drátového modelu, viz panel Nastavení.

Panel Projekt

Načítání a ukládání textur:

Načtení textury do paměti

... kliknutí levým tlačítkem myši na jednom ze 3 náhledů textur, zobrazí se dialog pro výběr textury ze souboru. Aplikace podporuje textury ve formátech BMP, GIF, JPEG.

Odstranění textury z paměti

... kliknutí prostředním nebo levým tlačítkem myši na příslušném náhledu textury

Vedle každé textury jsou dvě možnosti pro její filtrování:

Bilineární filtrace

… zapne/vypne bilineární filtraci pomocí OpenGL na příslušné textuře

Mipmapping

… zapne/vypne mipmapping na příslušné textuře

Kliknutím na tlačítko „Reset pozice“ je zobrazovaný objekt vrácen do základní polohy.

Tlačítko „Spustit test“ zapne kruhový pohyb světla pro zviditelnění stínování na povrchu objektu a po dobu 20 sekund bude měřen výkon grafické karty počítáním snímků – výsledek udaný v počtu snímků za sekundu (tzv. FPS z anglického *frames per second*) bude po dokončení testu zobrazen pod tlačítky.

Za nápisem „Vykresleno snímků:“ je udáván počet celkem vykreslených snímků. Aplikace zbytečně nezatěžuje procesor a grafickou kartu, takže v případě klidové polohy (žádný pohyb s objektem a světlem) není náhled opakovaně překreslován.

Kliknutím na barevný obdélník vedle nápisu „Barva pozadí:“ se otevře dialog pro výběr barvy pozadí 3D náhledu, který má několik režimů pro výběr barvy a pomocí ikony lupy dokáže okopírovat barvu libovolného pixelu na obrazovce. Jedná se o barevný dialog z knihovny Substance (http://substance.dev.java.net), avšak mírně opravený vzhledem ke špatnému českému překladu v originálních zdrojových kódech. Knihovna Substance byla dále použita pro změnu standardního vzhledu Java aplikací a dialogů pro výběr souborů.

Změna metody a parametrů

Zobrazovací metodu je možné vybrat z rolovacího seznamu *„Metoda:“*. Na výběr je 9 možností:

1. **Pouze textura**
   Prosté mapování textury pomocí OpenGL - vyžaduje načtenou texturu.
2. **Textura + Phong**
   Implementace Phongova stínování - vyžaduje načtenou texturu.
3. **Normal Mapping**
   Implementace metody normal mapping - vyžaduje načtenou texturu a normálovou mapu.
4. **Parallax Bump Mapping**
   Implementace metody parallax mapping s osvětlením spočteným pomocí normal mappingu - vyžaduje načtenou texturu, normálovou a výškovou mapu.
5. **Displacement Bump Mapping (DBM)**
   Implementace metody displacement mapping s osvětlením spočteným pomocí normal mappingu - vyžaduje načtenou texturu, normálovou a výškovou mapu.

6. **Displacement Bump M. bilin.f.**
   DBM s implementací bilineární filtrace výškové mapy ve vertex shaderu, s testem viditelnosti vrcholů ve vertex shaderu.
7. **Displacement Bump M. b. f. notest**
   DBM s implementací bilineární filtrace výškové mapy ve vertex shaderu bez testu viditelnosti vrcholů ve vertex shaderu.
8. **Displacement Bump M. mipmap**
   DBM se zapnutým mipmappingem pro výškovou mapu a výpočtem mipmapping úrovně ve vertex shaderu, s testem viditelnosti vrcholů ve vertex shaderu..
9. **Displacement Bump M. mip. int.**
   DBM se zapnutým mipmappingem pro výškovou mapu, výpočtem mipmapping úrovně ve vertex shaderu a interpolací dvou mipmapping úrovní, s testem viditelnosti vrcholů ve vertex shaderu.

Bez načtené textury je zobrazen jednobarevný povrch.

Základní parametry jednotlivých metod je možné měnit pomocí posuvníků v dolní části panelu Projekt. Nastavitelné parametry (škálovací koeficient, posunutí, teselace,…) se mění podle vybrané metody.

Načítání a ukládání projektu

Nastavení barvy pozadí a cest k texturám lze načíst ze souboru, z tzv. projektu, pomocí tlačítka *„Načíst…“* v panelu Projekt. V adresáři *projekty* (do kterého program automaticky vstoupí) je několik takových projektů připraveno, jedná se o soubory s příponou AMT.

Obdobně lze aktuální nastavení projektu uložit do souboru kliknutím na tlačítko *„Uložit…“*.

Záložka Nastavení

Drátový model

… zapnuto = objekt v náhledu je zobrazován jako drátový model pro znázornění složitosti geometrie

… vypnuto = objekt v náhledu je zobrazován s povrchem (barva nebo textura)

Vertikální synchronizace

... zapnuto = náhled je aktualizován maximálně frekvencí monitoru
... vypnuto = náhled je aktualizován bez omezení (hodí se pro test snímků za sekundu)

Zobrazení pozice světla

... zapnuto = v náhledu je znázorněna pozice světla malým bílým čtverečkem
... vypnuto = pozice světla není v náhledu znázorněna malým bílým čtverečkem

6.2 Srovnání metod

Bez použití pokročilých metod pro zobrazení detailů nerovných povrchů lze na objekt pouze nanést texturu. Takovýto způsob zobrazení je vysoce efektivní z hlediska výkonu, neboť dnes již mají všechny grafické akcelerátory zobrazování textur vysoce optimalizováno.

Obrázek č. 22: Zobrazení pomocí textury

Tento postup lze vylepšit případným výpočtem osvětlení jednotlivých bodů včetně světelných odlesků (Phongovo stínování). Stínování zkvalitní vzhled objektu, není však nijak ovlivněno nerovnostmi skutečného povrchu, který je pomocí textury znázorňován. Takto zobrazený povrch ale nevypadá příliš realisticky, a to zejména při pohybu.

Obrázek č. 23: Textura s vypočteným Phongovým stínováním

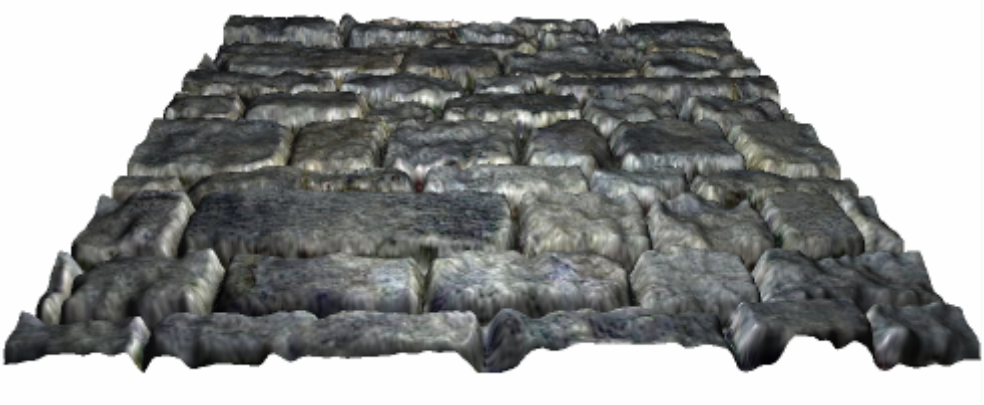
Za cenu relativně malých ztrát v rychlosti vykreslování lze získat oproti samotné textuře s Phongovým stínování podstatně lepší výsledek metodou normal mapping. Při pohybu se odpovídajícím způsobem mění stíny a odlesky na povrchu objektu objektu a vzniká tak částečný dojem plastičnosti. Tato metoda je velmi populární v současných počítačových hrách.

Obrázek č. 24: Normal mapping

Další možností je zobrazit povrch pomocí metody parallax mapping. Ve srovnání s metodou normal mapping dává přesvědčivější výsledky. Zatímco předchozí metoda manipuluje pouze s osvětlením a zobrazuje každý povrchový bod na stále stejné pozici relativně k celému povrchu, u parallax mappingu dochází k posunům pixelů a vzniká tak efekt většího vyvýšení a natáčení nerovných částí při pohybu pozorovatele vzhledem k objektu. Navíc ani parallax mapping nezpůsobuje výrazný pokles výkonu aplikace.

Obrázek č. 25: Parallax mapping

Jak ukazují obrázky, všechny předchozí metody nemění siluetu objektu, tzn. že při pohledu ze strany se ztrácí efekt prostorovosti. Tento nedostatek je odstraněn při zobrazení pomocí displacement mappingu. Tato metoda je sice poměrně výpočetně náročná, ale poskytuje bezpochyby nejlepší výsledky. Výhodou metody displacement mapping je, že nerovnosti povrchu se při různých úhlech pohledu vzájemně zakrývají jako na skutečném povrchu. Navíc lze pomocí parametrů snadno nastavit i velké vyvýšení nerovností



Obrázek č. 26: Displacement mapping

6.2.1 Porovnání rychlosti metod na různých grafických kartách

Program byl otestován na několika konfiguracích stolních PC i notebooků s různými grafickými kartami, v rozlišení monitoru 1024 x 768 bodů s vypnutou vertikální synchronizací obrazu.

Testované konfigurace:

6600	PC Intel Pentium IV 3.6 GHz, NVIDIA GeForce 6600
7600GS	PC AMD Sempron 2500+, NVIDIA GeForce 7600 GS
8600GT	PC Intel Core 2 Duo 2,33 GHz, NVIDIA GeForce 8600 GT
8800GT	PC AMD Athlon64 X2 5200+, NVIDIA GeForce 8800 GT
6100M	notebook AMD Sempron 3500+, NVIDIA GeForce 6100M
7600M	notebook AMD Turion64 X2 TL-52, NVIDIA GeForce 7600M
8600M GT	notebook Intel Core 2 Duo T7500, NVIDIA GeForce 8600M GT

Tabulka č. 2: Srovnání rychlosti metod na různých grafických kartách (ve FPS):

Metoda	**6600**	**7600GS**	**8600GT**	**8800GT**	**6100M**	**7600M**	**8600M GT**
Textura	669	803	1996	2784	112	863	933
Textura +Phong	302	534	1350	2261	72	453	639
Normal mapping	290	511	1185	2206	72	446	637
Parallax mapping	271	467	1100	2107	68	423	592
Displacement mapping	176	372	687	2468	52	297	445

Displacement mapping byl testován pro síť tvořenou 31 tis. vrcholy.

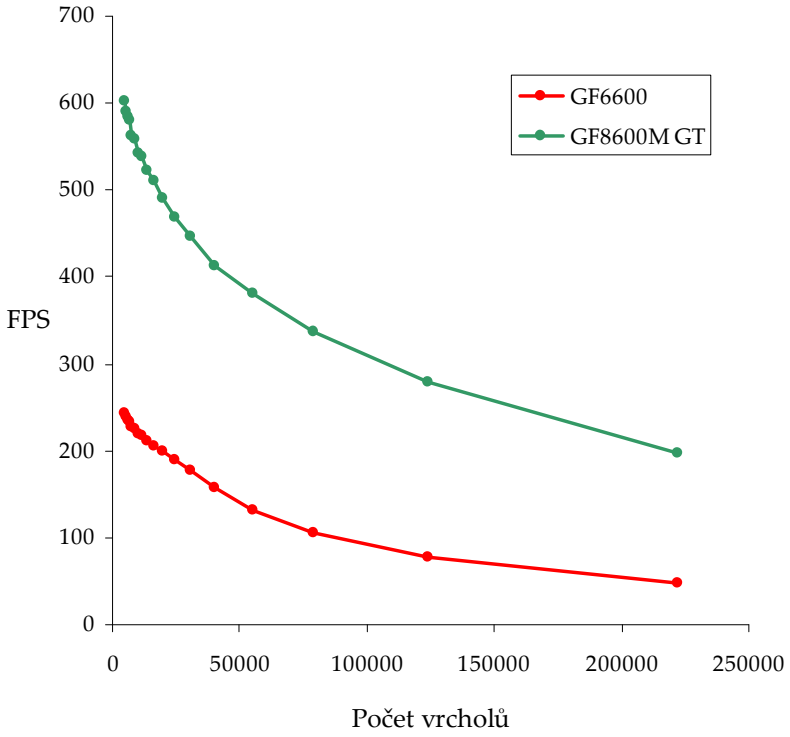
Tabulka jasně ukazuje pokrok ve výkonnosti v posledních třech generacích grafických karet NVIDIA GeForce jak v oblasti stolních počítačů, tak i notebooků. Je zřejmé, že i náročnou metodu displacement mapping, lze na novějších GPU s úspěchem realizovat.

6.2.2 Vliv jemnosti sítě u metody displacement mapping

Jemnost sítě, tzn. počet vrcholů, na které je zobrazovaný povrch rozdělen, má samozřejmě vliv na přesnost vymodelování geometrie z displacement mapy a následně i na kvalitu zobrazení.

Obrázek č. 27: Porovnání detailu zobrazení metodou displacement mapping při rozdělení povrchu na síť 5 000 (vlevo), 20 000 (uprostřed) a 220 000 vrcholů (vpravo)

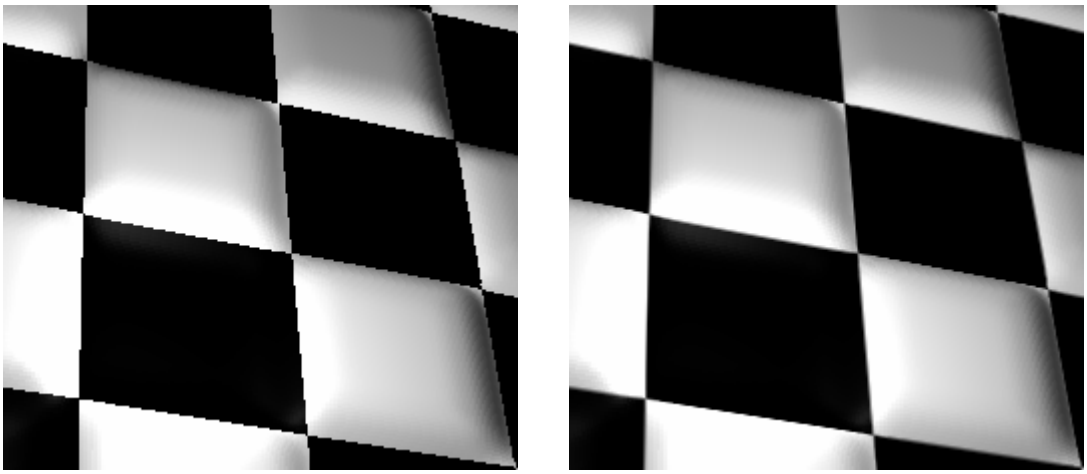
Za zlepšený vzhled zobrazovaných objektů se ovšem platí znatelným zvýšením výpočetní náročnosti a tedy snížením rychlosti vykreslování. Jemnou povrchovou síť lze ale v OpenGL efektivně zobrazovat například použitím vykreslování pomocí metody *vertex buffer object*, která nahrává data vrcholů do vysokorychlostní paměti grafické karty a podstatně tak zkracuje čas potřebný pro zobrazení. Je tak možné dosáhnou poměrně dobrých výsledků i na starších a mobilních grafických kartách.



Obrázek č. 28: Vliv jemnosti sítě na rychlost vykreslování

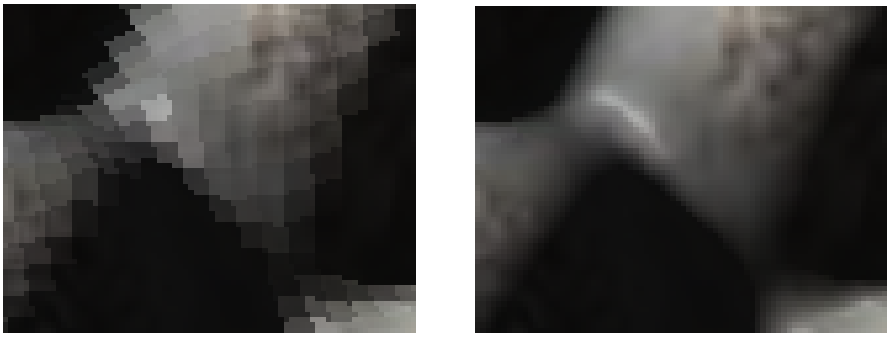
6.2.3 Vliv bilineární filtrace

V programu lze u jednotlivých textur přepínat mezi filtrací metodou nejbližšího souseda a bilineární filtrací. Stejně jako ve známějším případě použití filtrace na pouhou texturu, i v případě metod pro zobrazování detailů nerovných povrchů vyhladí nastavení bilineární filtrace pro texturu barevné přechody, jak nejlépe znázorňuje tradiční příklad šachovnice:



Obrázek č. 29: Porovnání zobrazení metodou displacement mapping s filtrací textury metodou nejbližšího souseda (vlevo) a s bilineární filtrací textury (vpravo)

V případě metod normal mapping, parallax mapping a displacement mapping má vliv také bilineární filtrace normálové mapy, a to na kvalitu osvětlení, zásadním způsobem totiž vyhladí odlesky a přechody světlo-stín:



Obrázek č. 30: Porovnání detailu zobrazení metodou displacement mapping s filtrací normálové mapy metodou nejbližšího souseda (vlevo) a s bilineární filtrací (vpravo)

Bilineární filtrace výškové mapy odstraní u metody parallax mapping drobné artefakty v posunu:

Obrázek č. 31: Porovnání detailu zobrazení metodou parallax mapping s filtrací výškové mapy metodou nejbližšího souseda (vlevo) a s bilineární filtrací (vpravo)

Při použití metody displacement mapping je nutné se na starších grafických kartách (NVIDIA GeForce řady 6x00 a 7x00) vyvarovat jiné filtrace výškové mapy než metodou nejbližšího souseda. Při přístupu k textuře z vertex programu funkcí vertex texture fetch je totiž na těchto grafických kartách použitelná pouze filtrace metodou nejbližšího souseda, neboť při použití bilineární filtrace dojde k prudkému poklesu výkonu. Vykreslování objektu se přepne do softwarového režimu zpracovávaného CPU, což zapříčiní pokles FPS o několik řádů dolů a maximální vytížení operačního systému. Na nejnovějších grafických kartách (NVIDIA GeForce řady 8x00) je již možné na výškovou mapu použít i bilineární filtraci.

Jak již bylo uvedeno, problém nepodporované filtrace u starších grafických karet lze vyřešit naprogramováním vlastní funkce pro vertex shader. Díky několikanásobnému přístupu k textuře z vertex programu se pak zobrazování zpomalí přibližně o 30 %, ve srovnání s použitou bilineární filtrací pomocí OpenGL je to však stále velmi dobrý výsledek.

Tabulka č. 3: Srovnání rychlosti vykreslování metodou displacement mapping s použitím a bez použití bilineární filtrace výškové mapy (ve FPS)

Metoda	**6600**	**8600M GT**
Displacement mapping bez použití bilineární filtrace	176	445
Displacement mapping s použitím bilineární filtrace pomocí OpenGL	7	441
Displacement mapping s použitím vlastní bilineární filtrace	127	432

6.2.4 Vliv mipmappingu

Mipmapping má vliv pouze při vzdálenějším pohledu na objekt, eliminuje zrnitost textury (mipmapping aplikovaný na texturu), odstraní vzdálené ostré střídavé přechody světlo-stín (mipmapping aplikovaný na normálovou mapu) znatelné zejména při pohledu na objekt ze strany a při pohybu.

Vedle zlepšení vzhledu zobrazovaného objektu má mipmapping ovšem pozitivní vliv i na rychlost vykreslování vzdálených objektů. Použitím mipmappingu se totiž zlepšuje efektivita využití vyrovnávací paměti, neboť se v mipmapě přistupuje k připravené textuře v menším rozlišení a do vyrovnávací paměti tak může být najednou načtena větší část povrchu.

U starších grafických karet (NVIDIA GeForce řady 6x00 a 7x00) je při realizaci metody displacement mapping největším přínosem pro rychlost mipmapping aplikovaný na výškovou mapu, protože je k ní přistupováno z vertex shaderu. U nejnovějších grafických akcelerátor je vliv mipmappingu na rychlost vykreslování o poznání menší.

Tabulka č. 4: Srovnání rychlosti vykreslování vzdáleného objektu metodou displacement mapping s použitím a bez použití mipmappingu výškové mapy (ve FPS)

Metoda	**6600**	**8600M GT**
Displacement mapping bez mipmappingu výškové mapy	313	1081
Displacement mapping s mipmappingem výškové mapy	368	1106
Displacement mapping s mipmappingem výškové mapy (včetně interpolace úrovní)	356	1093

6.2.5 Vliv testu viditelnosti

Zvláště u prvních grafických karet s podporou přístupu k textuře z vertex shaderu, u kterých ještě není tato funkce optimalizována, je výhodné implementovat do vertex programu displacement mappingu test viditelnosti. Je-li na výškovou mapu použita vlastní naprogramovaná bilineární filtrace nebo mipmapping, přináší test viditelnosti nezanedbatelné zvýšení rychlosti vykreslování objektů, jejichž část leží mimo záběr kamery.

Tabulka č. 5: Srovnání rychlosti vykreslování objektu, jehož část leží mimo záběr kamery, metodou displacement mapping s použitím a bez použití testu viditelnosti (ve FPS)

Metoda	6600		8600M GT	
	bez testu	s testem	bez testu	s testem
Displacement mapping s použitím vlastní bilineární filtrace	134	352	1010	1048
Displacement mapping s mipmappingem výškové mapy	312	507	1060	1077
Displacement mapping s mipmappingem výškové mapy (včetně interpolace úrovní)	297	500	1075	1090

6.2.6 Vliv interního formátu textury

Velmi důležitá je při implementaci metod zobrazování povrchových detailů volba správných interních formátů pro práci s texturami. Obzvlášť citlivé na formát jsou starší řady grafických karet GeForce 6x00 a 7x00. Pro displacement mapping a výškovou mapu je nutné zvolit OpenGL formát zvaný GL_RGBA_FLOAT32_ATI, který je funkcí vertex texture fetch podporován. V opačném případě je zaznamenán prudký pokles výkonu o několik řádů obdobný zapnuté bilineární filtraci ve stejném případě. Na grafických kartách řady GeForce 8x00 již takové problémy s interním formátem nenastávají.

Bohužel formát GL_RGBA_FLOAT32_ATI vyžadovaný pro vertex texture fetch v metodě displacement mapping není naopak vhodný pro textury a normálové mapy, popř. výškovou mapu v parallax mappingu. Při použití tohoto formátu se na grafických kartách GeForce 6x00 a 7x00 výpočet zpomalí zhruba na polovinu. Při zapnutí bilineární filtrace, která je u textur a normálových map více než vhodná pro vyhlazený obraz, je výpočet opět o několik řádů pomalejší oproti základně používanému internímu formátu GL_BGR.

Rejstřík

Literatura

[1] Fromek D. (2006): Výpočet geometrie na GPU, diplomová práce, Matematicko-fyzikální fakulta Univerzity Karlovy v Praze.

[2] Phillips J. (2007): Refactoring NAMD for Petascale Machines and Graphics Processors, 5th Annual Workshop on Charm++ and its Applications.

[3] Fernando R., Kilgard M. J. (2003): Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, Addison-Wesley.

[4] Dreijer S. (2007): Bump Mapping Using Cg, Blacksmith Studios, www.blacksmith- studios.dk/projects/downloads/ bumpmapping_using_cg.php.

[5] Gath J. (2006): Derivation of the Tangent Space Matrix, Blacksmith Studios, www.blacksmith-studios.dk/projects/downloads/ tangent_matrix_derivation.php.

[6] Phong B. T. (1973): Illumination for Computer-Generated Images, Ph.D. Dissertation, Department of Computer Science, University of Utah, Salt Lake City.

[7] Nuydens T.: Phong for Dummies, www.delphi3d.net/articles/viewarticle.php?article=phong.htm.

[8] Blinn J. F. (1978): Simulation of wrinkled surfaces, SIGGRAPH 78, s. 286-292.

[9] Kaneko T. a kol. (2001): Detailed Shape Representation with Parallax Mapping. In Proceedings of ICAT 2001, s. 205-208.

[10] Welsh T. (2004): Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces, Infiscape Corporation, www.cs.cmu.edu/afs/cs/academic/class/15462/web.06s/asst/project3/ parallax_mapping.pdf.

[11] Guinot J. (2006): Vertex Displacement Mapping using GLSL, www.ozone3d.net/tutorials/vertex_displacement_mapping_p04.php.

[12] Gerasimov P. a kol. (2004): Shader Model 3.0, Using Vertex Textures, NVIDIA Whitepaper

[13] Kryachko Y. (2005): Using Vertex Texture Displacement for Realistic Water Rendering, 1C:Maddox Games, GPU Gems 2, download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch18.pdf.

[14] Toksvig M. (2004): Mipmapping Normal Maps, NVIDIA Technical Brief.

[15] Green S. (2004): Bump map compression, NVIDIA Technical report.