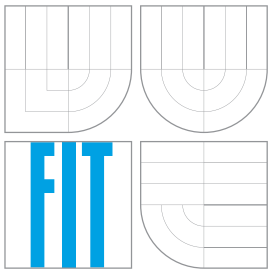
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

DEMONSTRAČNÍ APLIKACE PRO DISPLACEMENT MAPPING A ANALÝZU VÝKONNOSTI OPENGL PIPELINE

DEMONSTRATION APPLICATION FOR DISPLACEMENT MAPPING AND OPENGL PERFORMANCE ANALYSIS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN KUČERŇÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN PEČIVA, Ph.D.

BRNO 2013

Abstrakt

Tato diplomová práce se zabývá technikou displacement mappingu a analýzou výkonosti OpenGL pipeline s využitím volně dostupné proprietární knihovny GPUperfAPI, vytvořené společností AMD. V první části stručně popisuje OpenGL pipeline, její části jak hardwarové, tak softwarové. Následuje popis samotného displacement mappingu a jeho výhody, jemu příbuzných metod, a zmiňuje některé aplikace využívající nebo demonstrující displacement mapping. V další části popisuje rozhraní GPUperfAPI, implementační otázky jednotlivých metod a generování displacement map. V závěrečné části popisuje implementovanou aplikaci, její povahu, a účel.

Abstract

This thesis describes displacement mapping and performance analysis of OpenGL pipeline, with use of proprietary freeware library GPUperfAPI, made by AMD. First part briefly describes OpenGL pipeline, its parts, and then its counterparts in HW. Then its describes displacement mapping and similiar mapping methods, advantages of displacement mapping, and mentions few applications using displacement mapping.Next part describes interface of GPUperfAPI, implementation details of described methods and tools for generating of displacement maps. Last part describes nature of demonstration appliaction.

Klíčová slova

Teselace, Teselační shader, Displacement mapping, GPUperfAPI, OpenGL, OpenGL pipeline

Keywords

Tessellation, Tessellation shader, Displacement mapping, GPUperfAPI, OpenGL, OpenGL pipeline

Citace

Martin Kučerňák: Demonstrační aplikace pro displacement mapping a analýzu výkonnosti OpenGL pipeline, diplomová práce, Brno, FIT VUT v Brně, 2013

Demonstrační aplikace pro displacement mapping a analýzu výkonnosti OpenGL pipeline

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Ing. Jana Pečivy, Ph.D.

.....
Martin Kučerňák
30. května 2013

Poděkování

Rád bych tímto poděkoval svému vedoucímu, Ing. Janu Pečivovi, Ph.D., jeho kolegům, a všem kteří mě v této práci podporovali a drželi mi palce.

Obsah

Kapitola 1

Úvod

Ačkoliv existuje už velké množství způsobů a technik jak zlepšit vzhled vykreslovaného objektu, ať jde o bump mapping, parallax mapping, parallax occlusion mapping, používání specular textury, odlesky, tak žádná z těchto metod nemění geometrii objektu. To u některých objektů není problém, u jiných už ano. Displacement mapping, ačkoliv jde o poměrně obecný pojem, ve verzi zde prezentované umožňuje modifikovat geometrii objektu. Tyto zmiňované efekty je ale třeba vykreslovat pokud možno v reálném čase, proto se druhá část práce zabývá analýzou výkonnosti OpenGL. K této analýze slouží knihovna GPUperfAPI od společnosti AMD, která umožňuje sbírat data přímo z grafické karty. Také se zabývá knihovnou OpenSceneGraph která je využita pro vykreslování.

Kapitola 2

Popis OpenGL pipeline

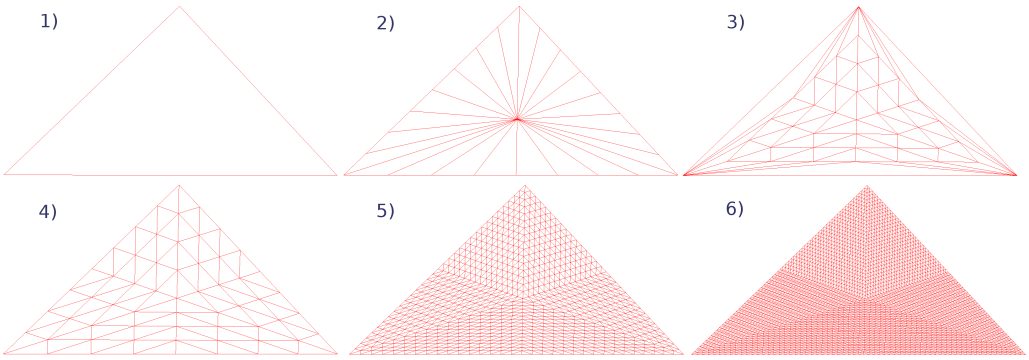
OpenGL vykreslovací pipeline je sekvence kroků, které provádí OpenGL při vykreslování objektů [5]. Její blokové znázornění je na obrázku 2.2, a dělí se na tyto kroky:

- Vertex Shader - je krok zpracování vrcholů vykreslovaného objektu. Vstupem je jeden vrchol (a přidružené hodnoty, jako jsou jeho normály, texturovací souřadnice, a další), výstupem také. Nemůže tedy vytvářet nové vrcholy, může ale vytvářet nové parametry, které se pak společně s aktuálně zpracovávaným vrcholem předají dále.
- Tessellation - je volitelný krok, ve kterém se, za pomoci hardwarové teselační jednotky grafické karty, aktuální polygon rozdělí na více menších dílů.
  - Tessellation Control Shader[1] - nastavuje teselační parametry pro aktuální polygon. Nastavované parametry jsou stupně vnitřní a vnější teselace. Podobně jako geometry shader vidí celý vstupní trojúhelník a může provádět podobné operace, jako je výpočet matice pro transformaci vektorů a vertexů z tangent space do object space a naopak. Při teselování mohou být podobné možnosti jako má geometry shader výhodou i v otázce výkonu, protože by vzrostl počet invokací řádové geometry shaderu, v případě maximálního stupně teselace, až 6000-krát. Při testování jednoho trojúhelníku s maximálním stupněm teselace byl počet trojúhelníků vstupujících do geometry shader 6144.

    TCS je také vhodné místo pro provádění back-face cullingu - ten by se normálně prováděl až po teselační fázi, a tedy až po vypočtení všech bodů nově vzniklých teselací (tzn po průchodu Tessellation Evaluation Shaderem). Tak by se zbytečně provádělo velké množství výpočtu, jako displacement mapping s množstvím přístupů do paměti, i samotný výpočet nových bodů - jeden nový vertex potřebuje na jednu hodnotu 14 instrukcí, a tento roste s každou další předávanou hodnotou, jako jsou normály, texturovací souřadnice, a jiné.
  - Tessellator - nejde o programovatelnou jednotku, je to fixní jednotka, která na základě parametrů nastavených v TCS vypočítá vstupními parametry určený počet barycentrických souřadnic pro nové body uvnitř aktuálního trojúhelníku (nebo čtyřúhelníku, záleží jaký typ polygonů se právě zpracovává), včetně krajních souřadnic představujících původní body. Vrcholy původního vstupního trojúhelníku jsou beze změny předány do Tessellation Evaluation Shaderu.

[1] V dalším textu bude označován zkratkou TCS

– Tessellation Evaluation Shader[2] - přepočítává barycentrické souřadnice (vygenerované teselátorem) nového vertexu na skutečné souřadnice v prostoru. Má podobné možnosti jako vertex shader. Vidí všechny body původního trojúhelníku, jak byly zpracovány v TCS, ale ne už barycentrické souřadnice jiných bodů v aktuálním trojúhelníku.



Obrázek 2.1: Příklad teselace jednoho trojúhelníku: 1) bez teselace 2) vnitřní/vnější teselace 1/8 3) vnitřní/vnější teselace 8/1 4) celková teselace 8 5) celková teselace 32 6) celková teselace 64 (maximum)

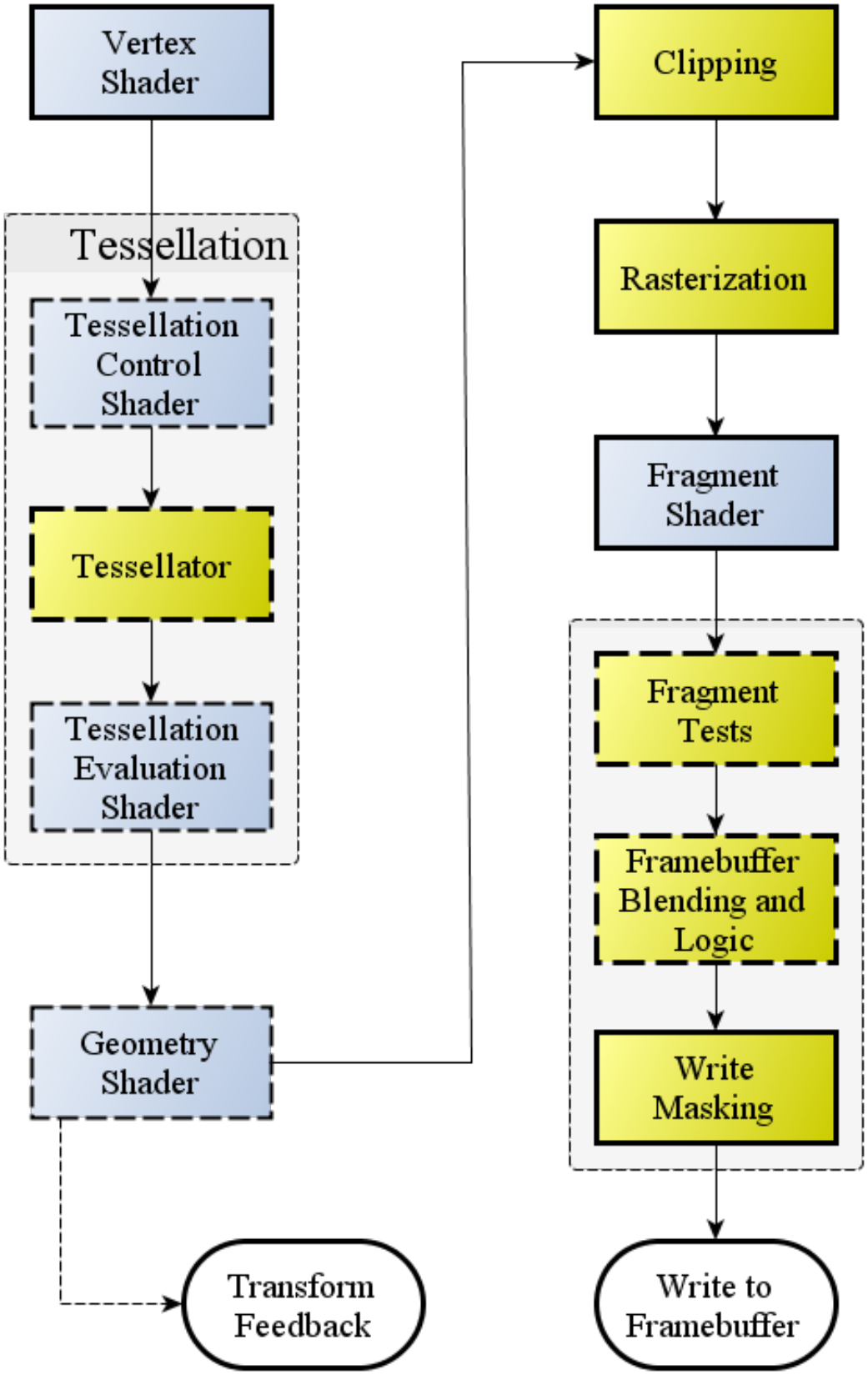
- Geometry Shader - vstupem je primitivum (může to být bod, linka, nebo polygon), které se zpracovává jako celek. Výstupem je také primitivum, které je možné měnit, jak přidáním atribitů (podobně jako ve vertex shaderu), tak přidáním celých vrcholů.
- Transform Feedback - je možnost zapsat výstup geometry shader do buffer objectu, se kterým lze dále pracovat, a využít ho opakovaně při dalších vykreslovacích operacích.
- Primitive Assembly (Clipping, Culling) - v této části probíhá předpříprava primitiv na rasterizaci, tato příprava se skládá z dělení primitiv, které nejsou vidět celá, a odstranění těch která nejsou vidět (například jsou mimo obrazovku, za kamerou, mimo view frustum, nebo je povoleno face culling a polygon je odvrácený od kamery).
- Rasterization - primitiva která prošla fází primitive assembly jsou zde zpracovány na fragmenty, a předány fragment shaderu. Fragment je soustava parametrů potřebných pro budoucí pixel (hodnota normály, texturovací souřadnice, pozice na obrazovce, pozice ve scéně).
- Fragment Shader - je krok zpracování jednotlivých fragmentů, ty jsou převáděna na hodnoty daného pixelu pro jednotlivé buffery, jako je framebuffer (barva pixelu), z-buffer (hloubka pixelu ve scéně).
- Fragment Tests, Framebuffer Blending and Logic, Write Masking - fáze kontroly a úprav výstupního pixelu, určuje se zda se má zapsat do framebufferu (zda není zakryt jiným pixelem, nebo není v části primitiva které leželo částečně mimo obrazovku), pokud je průhledný, tak se v případě, že se má zobrazit, zkombinuje jeho barva s již zapsaným pixelem. Pokud není pixel zahozen, tak se zapíše do framebufferu, a jeho hloubka do z-bufferu.

[2] V dalším textu bude označován zkratkou TES

2.1 Hardware

2.1.1 Unified Shader Architecture

V současných grafických kartách (počínaje čipy Radeon R600[7] a GeForce 8 Series[6], obě rok 2006) již nejsou samostatné bloky pro vertex, geometry a fragment operace, místo toho se používá architektura Unified Shader Architecture, kdy má karta velké množství univerzálních jednotek, které může dle potřeby alokovat mezi jednotlivé typy operací i v rámci vykreslování jednoho snímku obrazovky[15].



Obrázek 2.2: Znázornění vykreslovací OpenGL pipeline

Kapitola 3

Displacement mapping

Displacement mapping je metoda úpravy vykreslovaného objektu, kdy se za pomoci textury s výškovou informací upravuje geometrie objektu, a tak zvýšuje míra jeho detailnosti. Důvodů k použití této metody, oproti přímému použití detailnějšího modelu, je několik, lze mezi ně zařadit zejména:

- Komprese objektu
- Urychlení vykreslení
- Level of Detail

Pro potřeby této kapitoly se bude displacement mappingem rozumět postup, kdy se 3D objekt vykresluje s využitím teselační jednotky grafické karty. Ta zvýší počet vrcholů objektu, které se přesunou po normále nově vzniklého vrcholu o vzdálenost určenou namapovanou hodnotou v displacement mapě. Takto se sníží počet vrcholů které procházejí vertex shaderem.

3.1 Komprese objektu

V závislosti na využití objektu může být vhodné snížit velikost modelu, a tím snížit potřebný prostor pro jeho uložení v paměti, ať už fyzické, operační, nebo zejména paměť grafického čipu. Jedním ze způsobů jak toho docílit je právě Displacement mapping, kdy se část informace, která by byla normálně uložená v souřadnicích části vrcholů, uloží do textury.

Praktické hodnoty nabízí například článek [17], pro model froblina z demonstrační aplikace z kapitoly 4.1:

	Polygonů	Celková paměť
Nízké rozlišení modelu s teselací	5,160	Vertex a index buffer: 100KB 2K x 2K 16b displacement map: 10MB
Vysoké rozlišení modelu	15M+	Vertex buffer: 270MB Index buffer: 180MB

Tabulka 3.1: Velikost redukovaného modelu froblina a původního

V příkladu představuje displacement mapping značnou úsporu místa, ale je zřejmé že původní neredukovaný model má vyšší kvalitu než jaká by byla třeba pro jeho věrné vy-

kreslení[1]. Proto zde budou uvedeny další dva příklady, jeden je model přiložený k aplikaci GPU MeshMapper[2], druhý je pak model asteroidu vytvořený v programu Blender.

	Model (v kB)	Textury (v kB)	Celkem (v kB)
Nízké rozlišení modelu s displacement mappingem	638	Displacement map: 1025 Normal map: 4097	5760
Původní model ve vysokém rozlišení	21 620	-	21 620

Tabulka 3.2: Velikost modelu hlavy - redukovaného a původního - velikost na disku. Rozlišení map - 1024x1024, formát: dds

	Model (v kB)	Textury (v kB)	Celkem (v kB)
Nízké rozlišení modelu s displacement mappingem	34	Displacement map: 1025 Normal map: 4097	5156
Původní model ve vysokém rozlišení	32372	-	32372

Tabulka 3.3: Velikost modelu asteroidu - redukovaného a původního - velikost na disku. Rozlišení map - 1024x1024, formát: dds

Uložené soubory s modely využívající displacement mapping je dále možné zkomprimovat na ještě menší absolutní velikost bežnou kompresí, což může být výhodné v budoucnu, pro aplikace streamující data z internetu.

3.2 Urychlení vykreslení

Otázkou urychlení se bude zabývat jedna z následujících kapitol této práce. Lze očekávat, že se sníží počet vertex cache miss, protože se sníží počet vrcholů, které vstupují do pipeline, oproti tomu přibude zátěž pro fixní teselační jednotku (která by ale jinak zůstala nevyužita) a zejména pak nutnost na základě souřadnic vygenerovaných fixní teselační jednotkou vypočítat nové vrcholy.

3.3 Level of Detail

Fakt, že je vykreslovaný objekt v počáteční podobě uložen s nízkými detaily, a až při vykreslení prokreslen detailněji, například nastavením vyššího stupně teselace, umožňuje stupeň teselace nastavovat v závislosti na vzdálenosti objektu od kamery, a tím měnit počet vrcholů zpracovávaných TES a geometry shaderem.

Kromě dynamické změny stupně teselace je možné vzdálenější objekty vykreslovat s pomocí jiného shader programu, který nevyužívá teselaci. To může být výhodnější, protože už při stupni teselace 32 mohou být, v závislosti na zobrazovaném objektu, nově vzniklé polygony menší, než je jeden pixel. Takto to je řešeno například v demonstrační aplikaci March of the Froblins. V závislosti na tvaru modelu nemusí být takováto změna ani nápadná

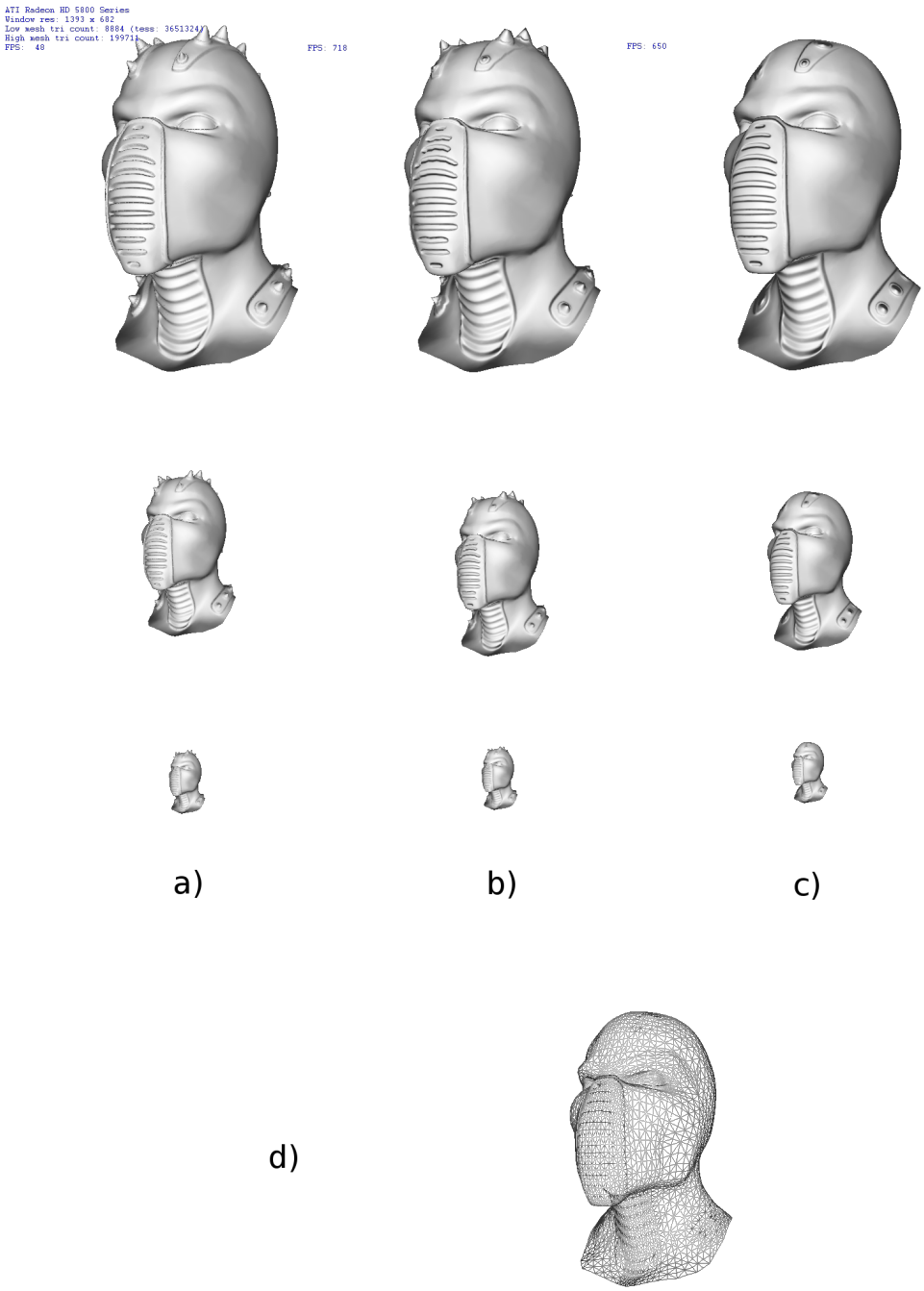
[1] Přece jen, jde o aplikaci vytvořenou za účelem demonstrace výhodnosti displacement mappingu, společností vyrábějící grafické karty které disponují technickými prostředky potřebnými pro displacement mapping

[2] například na obrázku 3.3

(pokud nejsou pomocí displacement mappingu modelovány prvky vystupující výrazně do prostoru)

Pro ilustraci možností LoD poslouží obrázek 3.3, vytvořený s pomocí aplikace GPU MeshMapper (kap.5.2). Zde lze vidět jak se od sebe liší vzhled displacement mappingu na teselovaném modelu, displacement mappingu na modelu bez teselace, a model bez teselace a displacement mappingu[3]. V tomto případě, vzhledem k některým prvkům modelu, jako jsou hroty na hlavě, je díky dobré struktuře modelu možné provést displacement původních vrcholů. Tím se neteselovaný model přiblíží vzhledu teselovaného modelu, s poměrně nízkou ztrátou výkonu, než jakou by představovalo vykreslení teselovaného modelu Jeho trojúhelníková síť je tak hustá, že i při přiblížení v horním řádku obrázku a při vykreslení jeho drátového modelu je pouze málo míst, kde prosvítá pozadí. Vykreslením modelu s displacementem, ale bez teselace lze tak v některých případech (jako je tento model) zamezit doskokům při změně úrovně LoD, které by nastaly, pokud by se nahrazoval teselovaný model modelem bez displacementu. Toto se liší případ od případu, například model asteroidu, nebo model dlažby, zobrazený v pozdější části by z displacementu bez teselace nijak neprofitoval.

[3]Z důvodů popsaných v kapitole 8.1.5 je na všechny případy aplikován normal mapping



Obrázek 3.1: Příklad možností LoD s pomocí displacement mappingu: a) teselace s displacement a normal mappingem b) pouze displacement a normal mapping na teselovaný model c) pouze normal mapping d) drátový model neteselovaného objektu

3.4 Teselátor

Předchůdce dnešní hardwarové teselace se objevil v roce 2001 na kartách řady Radeon 8500, od tehdejší společnosti ATI, jako technologie TruForm[3]. Ta spočívala v teselaci vykreslovaných trojúhelníků a posunu vnitřních bodů tak, aby ležely na pomyslné zakřivené rovině

procházející vrcholy původního trojúhelníku a zakřivené podle směru jejich normálových vektorů. Výsledkem byl méně hranatý povrch. Tato technologie byla přijata do OpenGL

Obrázek 3.2: Ukázka technologie TruForm (zdroj: http://wololo.net/2013/05/09/the-photo-realism-challenge-polygons/)

jako rozšíření ATI_pn_triangles[4], ale nikdy nebyla standardizována v DirectX, a NVidia nenabízela odpovídající alternativu (kromě Quintic-RT[1] patches, ale ty nebyly využity snad v žádném programu). Důsledek, společně s některými nedostatky které TruForm měl, jako zakulacení i objektů kde byly hrany úmyslem, byla podpora v nízkém počtu aplikací, a ATI proto v pozdějších kartách (od karet řady x700 a x800 není uváděn jako podporovaná technologie) od podpory této technologie upustilo.

Před příchodem DirectX 11 se hardwarová teselace objevila v grafickém čipu Xenos (z herní konzole Xbox 360)[1], a od něj odvozené rodiny čipů R600 (například Radeon HD2900). Tato teselace se dočkala podpory v OpenGL i DirectX formou rozšíření. Později se ukázalo že tato implementace není kompatibilní s tou, která byla použita v DirectX 11 a OpenGL, nelze tedy použít teselátor skrze tato API a je třeba využít původní rozšíření. Naštěstí ale současné grafické karty ATI (s podporou DirectX 11) podporují přístup k teselátoru přes toto rozšíření, takže je možné aplikace jej využívající stále používat.

Od čipů GF100 u Nvidie (GeForce GT280) a R700 u AMD (Radeon HD58xx) je přítomna podpora DirectX11/OpenGL 4.0, a s ní jejich implementace teselačních shaderů.

3.5 Další podobné metody

Obohacením vzhledu objektu o dojem plastičnosti nebo prostorovosti se zabývá více metod, než pouze displacement mapping. Zde jsou krátce uvedeny některé z nich.

[4] http://www.opengl.org/registry/specs/ATI/pn_triangles.txt

3.5.1 Bump mapping

Bump mapping je metoda navržená J. F. Blinnem[8] v roce 1978, ve své původní podobě spočívala v úpravě normály na základě výškové hodnoty získané z textury. Narozdíl od Normal mappingu byla tato hodnota jednorozměrná, podobně jako u Displacement mappingu. Samotná hodnota ovšem neumožňuje určit zamýšlené natočení povrchu, a bylo nutné získat vzorky z okolních bodů a na základě tohoto okolí určit nový směr okolí. Důvodem pro tento postup bylo způsob generování map, kdy se využívaly hodnoty z-bufferu pro vykreslení detailního modelu. V roce 1984 bylo představen Normal mapping v článku R. L. Cooka[10]. Oproti původnímu bump mappingu zde není třeba počítat směr normály z okolních bodů, ale normály jsou přímo uloženy v textuře, čímž se urychlí vykreslování. Později byly představeny metody jak jednoduše normálové mapy generovat.

Způsobů jak do textury uložit normály je více. Nejpoužívanější je ukládání v tangent space, které je sice na výpočet náročnější než object space, je ale o něco flexibilnější, nevadí mu například deformace způsobené animací modelu. Možné prostory jsou tyto:

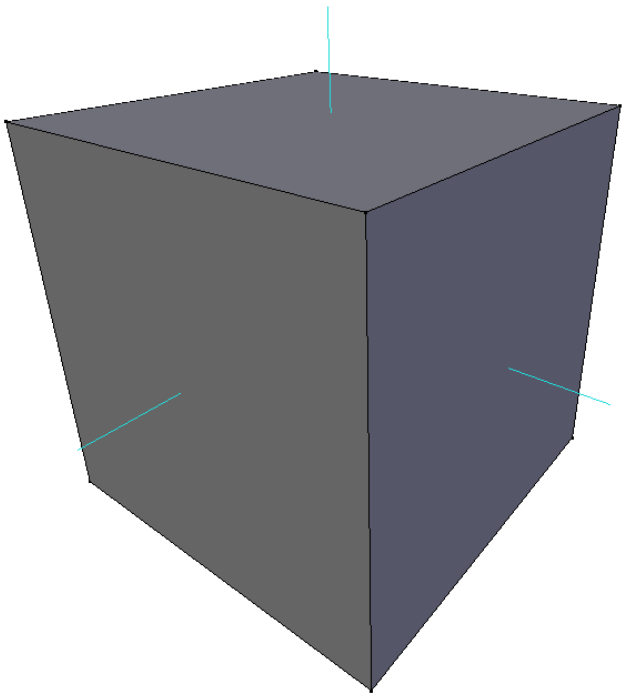
- Tangent space - je prostor určený polygonem kterého se týká[2]. Tento polygon představuje vlastní bázi, takže v tangent space by normálových vektorů zobrazené na obrázku 3.5.1 směřovaly přímo nahoru, jejich xyz složky by tedy byly 0.0, 0.0, 1.0.
- Object space - je prostor ve kterém je umístěný objekt. Normály v tomto prostoru mají stejnou bázi jako vrcholy, takže ty normály zobrazné na obrázku 3.5.1 v object space budou směřovat přesně tak jak jsou zobrazeny
- World space - uveden pro úplnost, protože bude zmíněn v jiné souvislosti dále. Všechny vektory a body v něm mají souřadnice a směrnice tak jak jsou umístěny ve scéně.

3.5.2 Parallax mapping

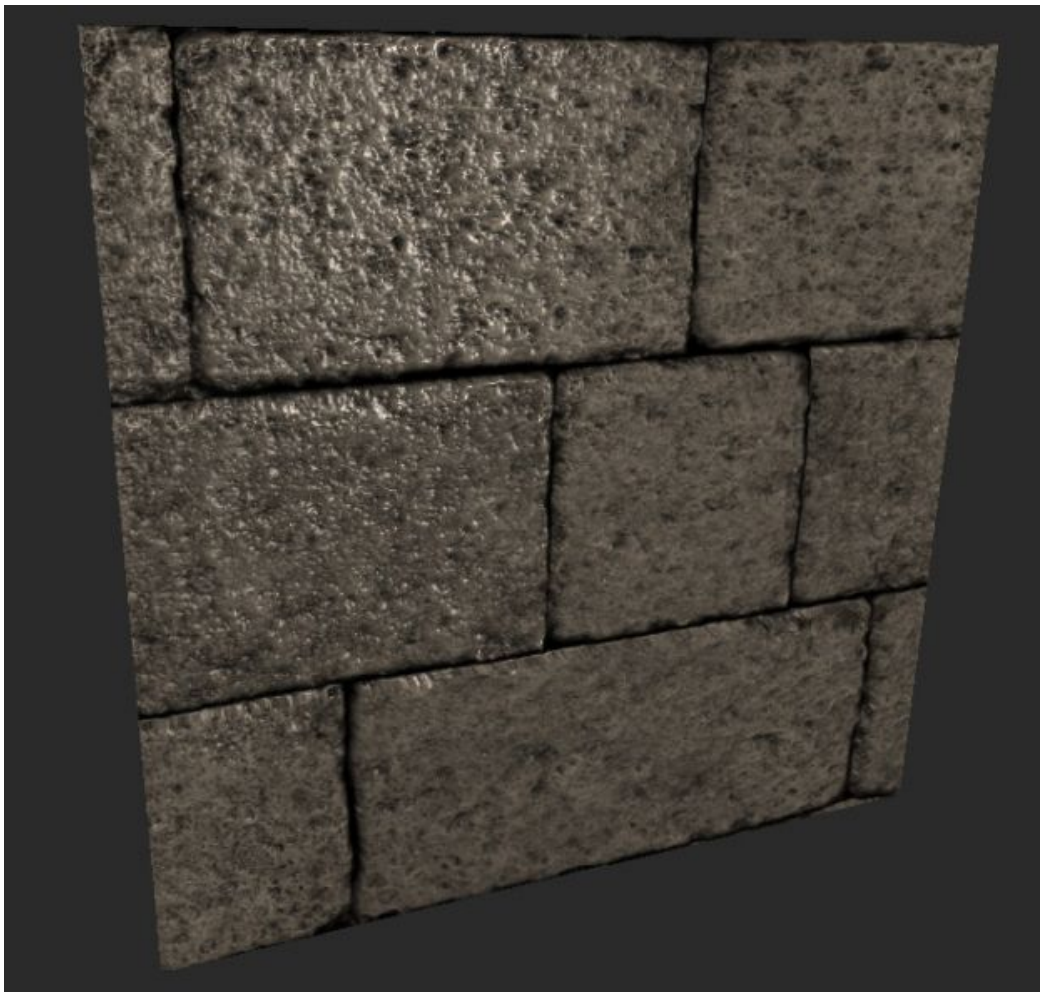
Paralax mapping je metoda kterou v roce 2001 navrhl Tomomichi Kaneko[13]. Jejím cílem je navodit dojem plastičnosti objektu. Za tímto účelem si při vykreslení trojúhelníku spočítá úhel jeho normály a pohledového vektoru kamery, ten následně vynásobí koeficientem získaným z výškové mapy, a o tuto výslednou hodnotu posune texturovací souřadnice. Neumožňuje ale aby se prvky textury překrývaly. Příklad využití displacement mappingu je na obrázku 9.

3.5.3 Per-pixel displacement mapping

Na per-pixel displacement mapping[4] lze pohlížet jako na inverzní paralax mapping. Stejně jako běžný displacement mapping používá výškovou mapu, ale s ní si dále spočítá vlastní displacement texturu, podle jejíchž hodnot upravuju texturovací souřadnice pro aktuální pixel. Oproti parallax umožňuje aby se nějaký objekt zakreslený do textury překryl jiný, toho je docíleno tak, že se displacement mapa nepočítá tak, aby se posunul aktuální pixel, ale přiřazuje bodům textury body objektu. Nevýhodou je nutnost tuto mapu předpočítat pro každou změnu pozice a rotace vůči kameře. Jeho implementace a detaily popisovány nebudou, protože v praxi se tato metoda, z důvodu potřeby přepočítat procesorem nevyužívá. Technicky si je blízká s parallax occlusion mappingem, ten ale probíhá jen v grafické kartě.

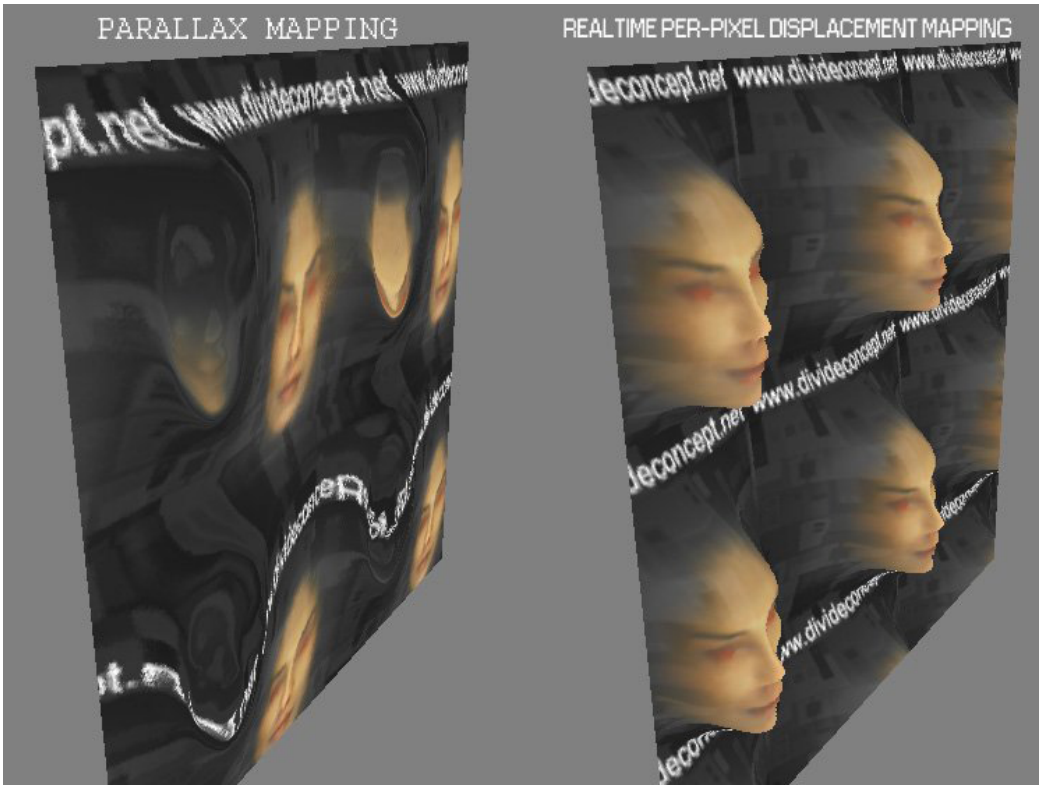


Obrázek 3.3: Směřování normál pro jednoduchý objekt



Obrázek 3.4: Příklad aplikace bump mappingu na zeď (zdroj: http://www.moddb.com/mods/new-vision/images/bump-mapping)

Obrázek 3.5: Ukázka parallax mappingu (zdroj: tomshardware.it)



Obrázek 3.6: Příklad aplikace per-pixel displacement mappingu a jeho srovnání s parallax mappingem (zdroj: [14])

Kapitola 4

Již existující demonstrační aplikace

Ačkoliv se využívání displacement mappingu v počítačových hrách zatím příliš nerozšířilo, existuje již slušné množství demonstračních aplikací. Některé z nich budou zmíněny v této kapitole, společně s několika hrami.

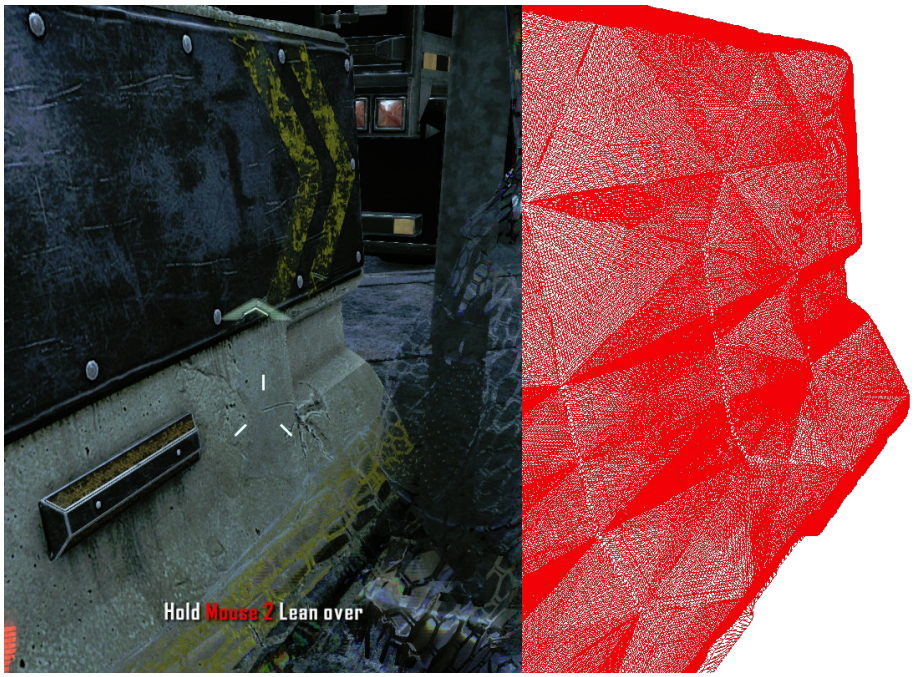
4.1 March of the Froblins

March of the Froblin [16] je demo vytvořené společností AMD pro grafické karty řady Radeon 4800 a rozhraní DirectX 10.1. Bylo součástí prezentace na konferenci SIGGRAPH 2008. Využívalo tehdejší teselaci, zpracovanou jako nestandardní rozšíření 3.4. Kromě teselace využívalo grafickou kartu pro další výpočty, jako pathfinding v prostoru. Jeho kladem je možnost předvedení jednotlivých prvků s předpřipraveným komentářem, a přítomnost rozsáhlého článku popisujícího jednotlivé detaily. I přes využití před- DirectX 11 / OpenGL 4.0 teselace lze spustit i na pozdějších grafických kartách Radeon.

Obrázek 4.1: March of the Froblins - jedna z komentovaných demonstrací

4.2 Crysis 2

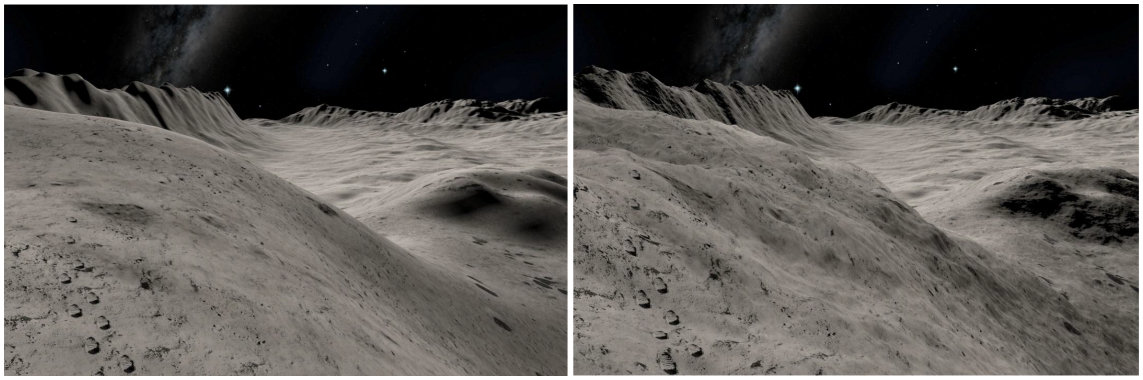
Crysis 2 je počítačová hra vydaná a vyvinutá neměckou společností Crytek v roce 2011. Verze vydaná na pc běží pod DirectX 9.0c, ale byl vydán patch pro podporu DirectX 11, společně s dodatečnými texturami. Jak ale ve svém článku rozebral S. Wasson[18], implementace teselace a displacement mappingu má určité nedostatky. Objekty se teselují s uniformním stupněm teselace, takže jsou rovné plochy často zbytečně teselovány, aniž by z toho byl faktický užitek, a některé prvky, jako třeba voda, se zbytečně teselují i když jsou pak v primitive assembly zahozeny.



Obrázek 4.2: Crysis 2 - ukázka zbytečně teselovaných oblastí modelu. zdroj: [18]

4.3 Tom Clancy's H.A.W.X 2

Tom Clancy's H.A.W.X 2 je počítačová hra vydaná společností Ubisoft a vyvinutá společností Ubisoft Romania. Displacement mapping využívá pro vykreslování detailnějšího terénu. Způsob, jakým tak dělá, popsal ve svém článku I. Cantlay[9]. Využívá dvou úrovní displacement map, kdy jedna určuje základní tvar terénu (kopce, údolí), a druhá jim dodá detail.



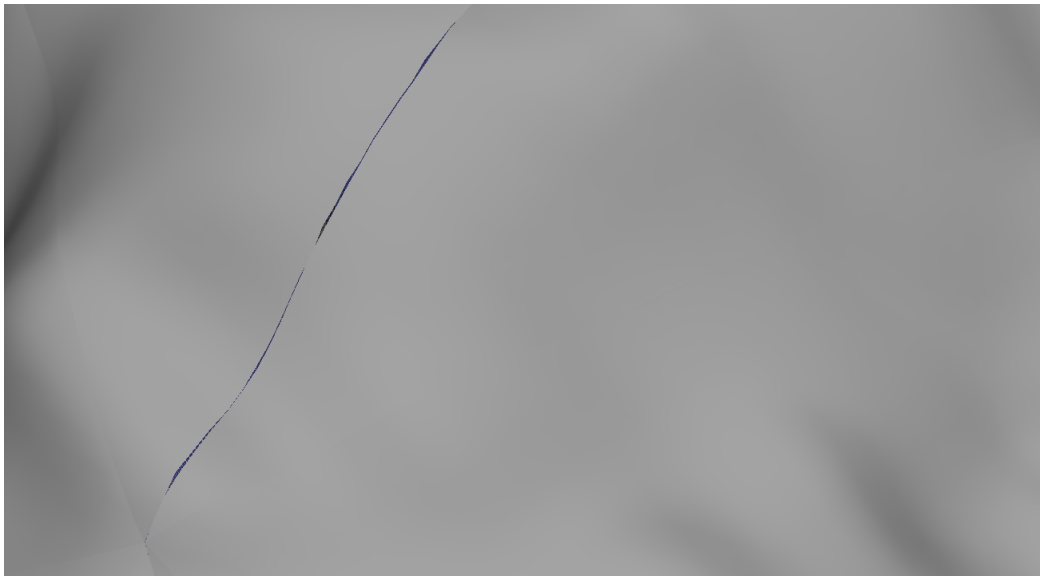
Obrázek 4.3: Ukázka teselace terénu metodou využitou v Tom Clancy's H.A.W.X 2. Levá strana - hrubá displacement mapa, určující podobu terénu. Pravá strana - stejný terén s použitím mapy druhé úrovně. zdroj: [9]

Kapitola 5

Generování displacement map

Pro zakomponování displacement mappingu do funkční aplikace je nejprve třeba mít připravený model s displacement mapou. Tu je možné buď vytvořit ručně, pokud nemáme k dispozici původní model, nebo pokud žádný nebyl, náš model byl původně vytvořený v této podobě, a displacement mappingem mu chceme dodat detaily (vytvořit některé prvky v textuře může být jednodušší než je vytvořit na 3D modelu). Nebo v případě že je model, který použijeme, vytvořený zjednodušením složitějšího modelu, můžeme vygenerovat displacement (a normal) mapu jako jejich rozdíl. Ze snadno dostupných nástrojů tuto funkcionalitu nabízejí například Blender, nebo GPU MeshMapper[1].

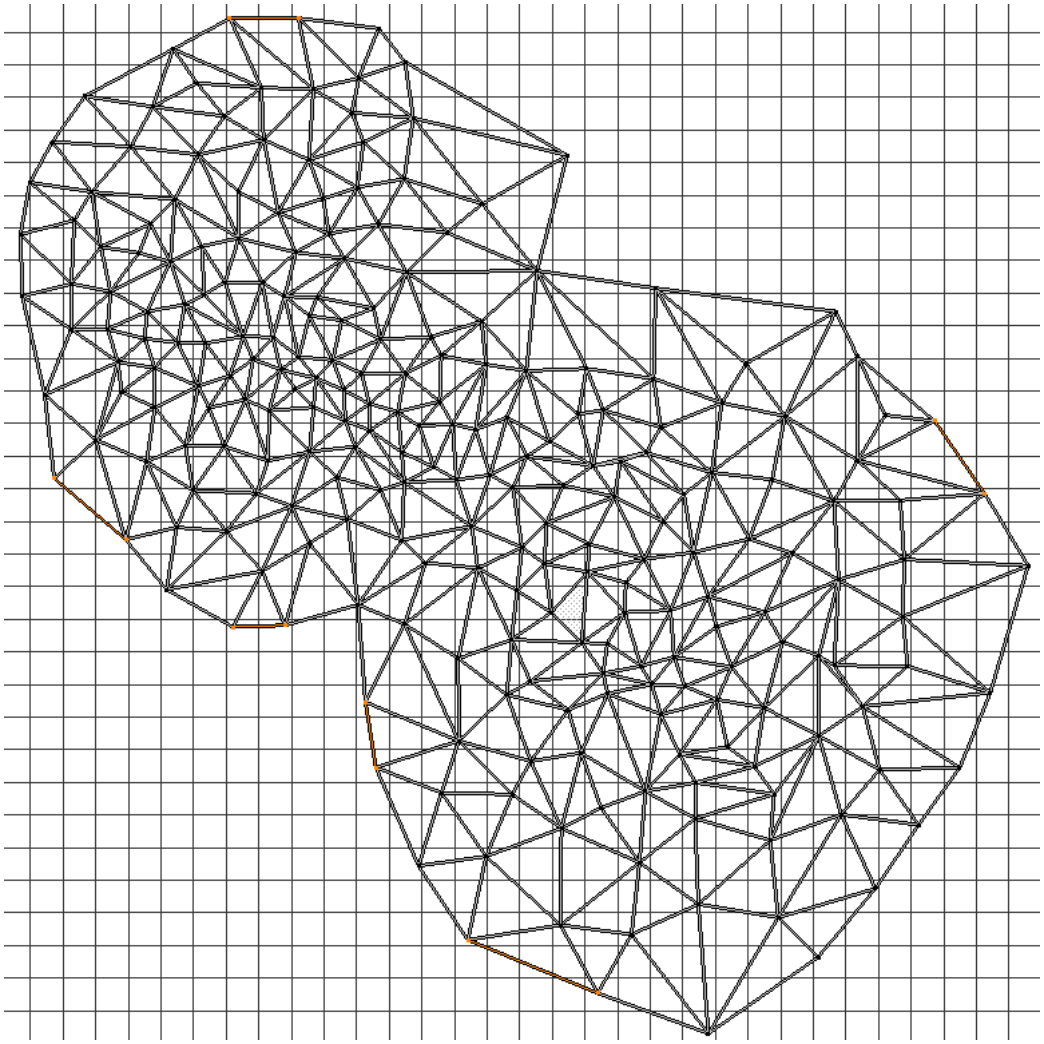
Generování displacement mapy je poměrně citlivé na chyby. Ty se týkají zejména nesymetrických hran texturovacích souřadnic podél řezu.



Obrázek 5.1: Příklad díry v modelu na hraně texturovacích souřadnic

Pokud mají tyto odpovídající hrany jinou délku, nebo i jinou směrnici, není zaručeno, že si navzájem odpovídají všechny hodnoty ležící na těchto hranách. Výsledkem může být, že při aplikaci displacementu bude vertex beroucí informaci z první hrany posunut o jinou vzdálenost než jeho dvojče z druhé strany, a ve vykresleném modelu tak vznikne mezera.

[1] http://developer.amd.com/resources/archive/archived-tools/gpu-tools-archive/amd-gpu-meshmapper/



Obrázek 5.2: Příklad navzájem si odpovídajících hran, s rozdílnou délkou

Možných řešení je více. Texturovací souřadnice mohou být vytvořeny tak, aby si hrany navzájem odpovídaly délkou, i směrem. To je ale poměrně časově náročné, protože přinejmenším žádný z testovaných nástrojů si s tím neporadí, a bylo by třeba texturovací souřadnice ručně upravovat.

Jiných možných řešení je více.

- Nabízí se myšlenka v daném místě rozpojit, a částečně obě hrany překrýt. To by ale mohlo narušit normály (které by měly být spojité po celé ploše modelu, tedy takzvaně smooth) a způsobit ještě větší díry.
- Vytvořit takzvané skirts, podobě jako se například v enginu Ogre3D používají na spojování částí terénu. Muselo by se ale ošetřit aby se neposunovaly v nevhodném směru, a aby se posunovaly společně s hranou k níž mají náležet (musely by přijmout jejich normály aniž by je ovlivňovaly. Byla by zřejmě potřeba ruční korekce každé normály v modelovacím programu)
- Průměrovat displacement na hraně sítě texturovacích souřadnic s odpovídající hodnotou na protější hraně (myšlenka na úpravu texturovacích souřadnic pochází z [11]).

Tím se odstraní rozpor mezi posunem na obou hranách. Nese to sebou při vykreslování určitou zátěž navíc, i když ne nijak velkou. Nejprve je třeba si předpřipravit model, a to tak že se vytvoří druhá sada texturovacích souřadnic. Ty vrcholy, které neleží na kraji sítě texturovacích souřadnic budou mít druhou souřadnici stejnou jako první. Pro ty, které leží na kraji, se vyhledá jejich protějšek (vrchol bude mít stejné souřadnice), a nazvájem si přiřadí jako druhé texturovací souřadnice první souřadnice toho druhého vrcholu.

Následně se v TCS určuje zda aktuální trojúhelník leží některým z bodů na kraji sítě porovnáváním texturovacích souřadnic - ty které leží uprostřed mají obě stejné, ty které neleží je mají různé. Podle toho se nastaví příznak pro TES. Pokud příznak určuje, že trojúhelník leží některou hranou nebo vrcholem na kraji texturovací sítě, podle barycentrických souřadnic se v TES určí zda se zpracovává bod ležící na kraji texturovací sítě. Pokud leží, vypočítají se texturovací souřadnice jeho protějšku, odečte se jeho hodnota, a zprůměruje se s hodnotou aktuálního bodu.

Kód z TCS

```
action[0] = 0;
if(vTescoord[0] != vSecondTC[0] && vTescoord[1] != vSecondTC[1])
{
action[0] = 1;    //je příznak předávající se dále
}
else if(vTescoord[0] != vSecondTC[0] && vTescoord[2] != vSecondTC[2])
{
action[0] = 2;
}
else if(vTescoord[1] != vSecondTC[1] && vTescoord[2] != vSecondTC[2])
{
action[0] = 3;
}
else if(vTescoord[0] != vSecondTC[0])    //porovnávají se vrcholy
{
action[0] = 4;
}
else if(vTescoord[1] != vSecondTC[1])
{
action[0] = 5;
}
else if(vTescoord[2] != vSecondTC[2])
{
action[0] = 6;
}
```

Kód z TES

```
if(action[0]!=0 && (gl_TessCoord.z<=0.0
|| gl_TessCoord.x<=0.0 || gl_TessCoord.y<=0.0 ))
{
vec2 nTexCoord;
```

```
int test = 0;
if(action[0]==1 && gl_TessCoord.z<=0.0)           //kontrola hrany
{
nTexCoord=gl_TessCoord.x * tcSecondTC[0] + gl_TessCoord.y * tcSecondTC[1]; test=1;
}
else if(action[0]==2 && gl_TessCoord.y<=0.0)
{
nTexCoord=gl_TessCoord.x * tcSecondTC[0] + gl_TessCoord.z* tcSecondTC[2];  test=1;
}
else if(action[0]==3 && gl_TessCoord.x<=0.0)
{
nTexCoord=gl_TessCoord.z * tcSecondTC[2] + gl_TessCoord.y * tcSecondTC[1];  test=1
}
else if(action[0]==4 && gl_TessCoord.x>=1.0)          //kontrola vrcholu
{
nTexCoord=gl_TessCoord.x * tcSecondTC[0];  test=1;
}
else if(action[0]==5 && gl_TessCoord.y>=1.0)
{
nTexCoord=gl_TessCoord.y * tcSecondTC[1];  test=1;
}
else if(action[0]==6 && gl_TessCoord.z>=1.0)
{
nTexCoord=gl_TessCoord.z * tcSecondTC[2];  test=1;
}
if(test==1)
{
  disp = (disp + (texture(dispmap, nTexCoord).b - 0.124426)*DisplacementFactor)/2;
}
}
```

Výhodou tohoto přístupu je jednoduchost, nevýhodou několik instrukcí v shaderu navíc.

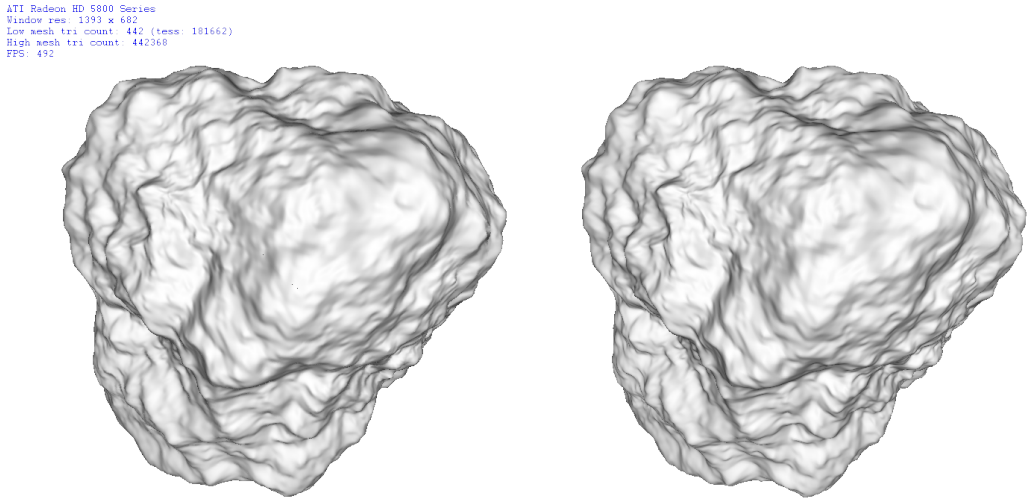
5.1 Blender

Blender umožňuje generovat mimojiné displacement a normal mapy, generování je nazvano "zapékáním"(baking). Narozdíl od GPU MeshMapper ale nevypočítá velikost posunu mezi nejvyšší a nejnižší hodnotou ve vygenerované displacement mapě. Stejně tak i modely z něj měly větší mezery podélk hran v texturovací síti.

5.2 GPU MeshMapper

GPU MeshMapper je volně dostupný nástroj určený pro generování normálových, displacement, a ambient occlusion map. Má oproti blenderu výhodu při generování displacement map, a to že určí rozsah posunu a středovou hodnotu, pomocí kterých je možné věrně napodobit vzhled původního modelu. Snaží se filtrovat texturu podél hrany, tak aby znikaly

minimální, pokud možno žádné, díry v teselovaném modelu. Výstup má ale stále menší nedostatky.



Obrázek 5.3: Napravo je původní objekt, nalevo je zjednodušený objekt vykreslený s využitím displacement a normal mappingu

Kapitola 6

GPUperfAPI

GPUperfAPI[1] je knihovna vytvořená společností AMD, pro použití s grafickýma kartamy Radeon. Slouží k inicializaci, nastavení a přistupování k různým čítačům grafické karty, a umožňuje tak zkoumat zatížení jednotlivých částí karty při vykreslení dané scény. Umožňuje pracovat s DirectX (od verze 10), OpenGL, OpenCL.

Čítače jsou rozdělené do skupin podle jejich povahy, jednu skupinu představují čítače časového vytížení karty a jednotlivých bloků a shaderů, dále jsou k dispozici podrobnější čítače jednotlivých bloků (například počet přístupů do paměti, čas strávený přístupy do paměti, počet zpracovaných prvků, vytížení ALU jednotky, atd.)

U shaderů se zaznamenává počet přístupů k textuře a čas strávený těmito instrukcemi, průměrný počet aritmetických instrukcí, poměr aritmetických a texturovacích instrukcí, počet vstupních a výstupních prvků, vytížení shaderu, a efektivitu.

Skupiny přístupných čítačů jsou tyto

- Timing - Celkový čas výpočtu, procento využití shaderů shaderů a rozložení zátěže mězi ně. Lze tak zjistit která z části pipeline je přetížená, a který z shaderů je třeba zoptimalizovat. Také slouží jako vodítko k určení části pipeline, na jejiž podrobné čítače pomohou analyzovat zátěž.
- VertexShader - Kromě údajů, které jsou i u ostatních shaderů nabízí i počet vertexů, které nebyly v cachi, a čas strávený jejich načítáním z paměti. Do počtu zpracovaných vertexů se započítává i počet vertexů zpracovaných domain shaderem.
- HullShader - Hull Shader je DirectX označení pro TCS, patří sem jeho čítače, které jsou základní shaderové čítače, popsané výše.
- DomainShader - Domain shader je DirectX označení pro TES.
- GeometryShader - Čítače geometry shaderu, kromě standardních čítačů nabízí i čas strávený exportováním primitiv.
- PrimitiveAssembly - Čítače operací týkajících se primitiv, tedy počet ořezaných primitiv, doba čekání na pixel shader s odesláním nového fragmentu, a počet fragmentů na trojúhelník.
- PixelShader - Kromě standardních čítačů přítomných u shaderů nabízí i čas strávený čekáním, až bude možné poslat fragment dále na Z-Test a do ColorBufferu

[1] http://developer.amd.com/tools/graphics-development/gpuperfapi/

- TextureUnit - Celkový počet přístupu k texturám, čekání na texture cache, čas strávený filtrováním.
- TextureFormat - Čas strávený prací s texturami jednotlivých formátů.
- ComputeShader - Čítače pro OpenCL.
- DepthAndStencil - Z-test a Stencil test. Počet fragmentů které prošly testováním,
- ColorBuffer - Počet paměťových přístupů

Protože není možné zaznamenávat hodnoty ze všech čítačů současně, jsou nezávisle na výše uvedeném rozdělení rozděleny do dalších skupin, podle toho které je možné zaznamenávat současně. Pro zaznamenání hodnot více čítačů je tedy třeba určit kolikrát musíme scénu vykreslit pro získání všech dat. Toto rozdělení je závislé na ovladači, API a konkrétní architektuře grafické karty, pro jeho získání je třeba použít jednu z funkcí této knihovny, a scénu vykreslit vícekrát, v závislosti na této hodnotě. Postup při použití této knihovny má přibližně následující podobu

- preinicialize - před vytvořením vykreslovacího kontextu se voláním funkce GPA_Initialize připraví ovladač na vytvoření kontextu s čítači
- incializace - po získání handle na vytvořený vykreslovací kontext se funkcí GPA_Initialize nastaví tento kontext jako aktuální, poté je třeba nastavit čítače (po spuštění jsou všechny deaktivované)
- analýza - pro celý průběh se nejprve vytvoří session, následně se získá počet potřebných průchodů (závislý na grafické kartě a kombinaci povolených čítačů; Průchod se zahají a ukončí dvojicí GPA_BeginPass, GPA_EndPass). V každém průchodu se musí vykreslit celá scéna, toto vykreslení se dále zaobalí dvojicí GPA_BeginSample(x), GPA_EndSample(). Pomocí proměnné X je možno označit zaznamenaný vzorek čítačů, a mít tak více dat. Nevýhodou je potřeba dalšího vykreslení scény.
- získání údajů - než je možné přistoupit k hodnotám čítačů, je třeba ukončit session, a následně čekat než bude uzavřená session připravená k čtení (GPA_IsSessionReady). Protože GPUperfAPI umožňuje pamatovat si jen omezený počet session, je následně třeba hodnoty ze session zaznamenat. Pro jejich čtení je třeba kontrolovat jakého typu je daný čítač (int32/64, float32/64), protože pro každý typ je možné číst hodnotu jen funkcí pro daný typ určenou.
- ukončení - před ukončením aplikace je třeba zavolat funkci GPA_Destroy, která deaktivuje čítače. Pokud by nebyly deaktivovány, může být negativně ovlivněn výkon dalších aplikací pracujících s grafickou kartou.

6.1 GPU PerfStudio 2

Je freeware aplikace založená na GPUperfAPI, vyvíjená společností AMD. Umožňuje analyzovat již existující 3D aplikaci bez úpravy kódu nebo nutnosti ji znovu zkompilovat, může sloužit jako 3D debugger, nebo poskytovat za běhu informace z jednotlivých čítačů.

6.2 NVPerfKIT

Knihovna NVPerfKIT je, podobně jako GPUperfAPI, rozhraní sloužící pro přístup k výkonovým čítačům grafické karty. Naneštěstí se s tímto rozhraním objevily problémy, jejichž původ se nepodařilo blíže objasnit. NVPerfKIT i k němu přiložená demonstrační aplikace bylo zkoušeno na dvou různých počítačích, a na obou měly čítače při odečítání hodnot téměř vždy nulovou hodnotu.

Kapitola 7

Popis prostředků použitých pro demonstrační aplikaci

Demonstrační aplikace je vytvořená za pomocí knihoven Qt a OpenSceneGraph. Navzdory platformové nezávislosti těchto knihoven aplikace samotná nezávislá není, z důvodu využití Windows API funkce pro získání vykreslovacího kontextu (bude ale upravena tak aby pro přizpůsobení jiné platformě ji stačilo na této platformě pouze znovu přeložit).

7.1 Qt

Qt je multiplatformní knihovna poskytující prostředky pro zobrazení a spravování uživatelského rozhraní aplikace. Poskytuje třídy pro základní uživatelské prvky, používá vlastní systém signálů a slotů pro komunikaci mezi těmito prvky a odvozenými třídami.

7.2 OpenSceneGraph

OpenSceneGraph je mutliplatformní opensource knihovna určená pro 3D aplikace. Slouží jako abstrakční vrstva nad OpenGL, takže není třeba vykreslovat jednotlivé modely, ale postačí scéna ke které jsou přiřazeny, a poskytuje metody pro načítání obsahu (modely, textury). Její hlavní přínos spočívá v uložení objektů v podobě grafu scény[1], kde jsou objekty seskupeny podle polohy, a podle vzájemného vztahu (například aby bylo auto ve stejném podstromu jako jeho kola). Kromě této organizace objektů poskytuje OpenSceneGraph i optimalizaci scény, kdy nevykresluje objekty které leží mimo scénu, a seskupí je podle podobnosti jejich StateSetů (StateSet je třída seskupující OpenGL parametry a nastavení pro daný objekt, například shader program).

[1] Graf ve smyslu grafové struktury, v tomto případě stromu

Kapitola 8

Implementace efektů

Tato kapitola bude popisovat implementaci jednotlivých efektů v shader jednotkách grafické karty a jazyce GLSL.

8.1 Displacement mapping

Jádro displacement mappingu se odehrává v TCS a hlavně v TES. Princip bude popsán v této kapitole, společně s dalšími potřebnými úpravami v kódu shaderů.

8.1.1 Model-view-projection transformace

Naneštěstí, do TES je také nutné přesunout transformace vrcholů z vertex shaderu, čímž je bude třeba provádět pro všechny vrcholy vzniklé teselací původního trojúhelníku. Jejich počet lze vidět na obrázku 2. ModelView transformace mění složku W vertexů vykreslovaného objektu, a posun vypočítaný z displacement mapy by neměl odpovídající velikost. Bylo by pochopitelně možné tento posun také přepočítat do view space, ale to by vyžadovalo prakticky stejné operace s maticemi jako přesun veškerých transformací až do TES.

8.1.2 Tessellation Control Shader

V Tessellation Control Shaderu je třeba pro základní displacement mapping pouze nastavit jednotlivé stupně teselace, například

```
if (gl_InvocationID == 0) {
gl_TessLevelInner[0] = TessLevelInner;
gl_TessLevelOuter[0] = TessLevelOuter;
gl_TessLevelOuter[1] = TessLevelOuter;
gl_TessLevelOuter[2] = TessLevelOuter;
}
```

kde gl_TessLevelOuter a gl_TessLevelInner jsou parametry určující stupeň teselace (Outer se nastavuje pro hrany polygonu), TessLevelInner a TessLevelOuter jsou uniformy odpovědné za předání hodnoty, a gl_InvocationID je proměnná informující o tom, který z bodů vstupního trojúhelníku (nebo přesněji, patche) je aktuálně TCS zpracováván. Tento blok tedy zajistí že stupně teselace nebudou zbytečně nastavovat vícekrát.

8.1.3 Tessellation Evaluation Shader

TES zajišťuje výpočet nových vertexů na základě barycentrických souřadnic vygenerovaných teselátorem. Výpočet jednotlivých prvků může mít následující podobu:

```
vec3 p0 = gl_TessCoord.x * tcPosition[0];
  vec3 p1 = gl_TessCoord.y * tcPosition[1];
  vec3 p2 = gl_TessCoord.z * tcPosition[2];
vec3 p_sum = (p0 + p1 + p2);
```

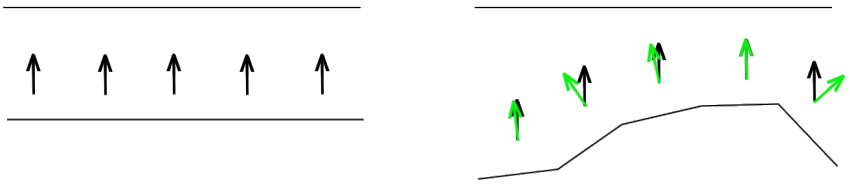
kde tcPosition jsou vrcholy původního trojúhelníku, a gl_TessCoord je vektor obsahující barycentrické souřadnice vygenerované teselátorem. Nový vertex vznikne váhovaným součtem původních vrcholů, tento součet, protože součet barycentrických souřadnic se rovná 1, představuje nový vertex na určené pozici uvnitř trojúhelníku. V TES se také provádí transformace nových vertexů pomocí ModelView a Projection matic.

8.1.4 Displacement vertexů

Aby se dosáhlo viditelné změny na vykreslovaném objektu a získal tak plastický vzhled, je třeba měnit polohu vygenerovaných vertexů. Pro to jsou třeba dvě věci, vzdálenost a směr. Vzdálenost je možné získat z displacement mapy, jako směr mohou posloužit normály vrcholů původního trojúhelníku.

8.1.5 Efekt na normály

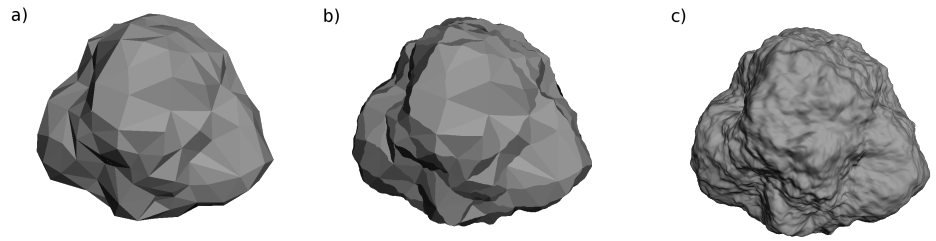
Jedna z věcí které je u displacement mappingu třeba řešit, je změna orientace normálovy vektoru nově vytvořených trojúhelníků oproti normálně původního trojúhelníku. Protože se jednotlivé vertexy neposunují o stejnou délku, orientace těchto nových trojúhelníků neodpovídá původní normále, viz obrázek 8.2.2. Výsledkem je že se osvětlení pro tyto nové trojúhelníky počítá stejně jako pro ten původní, takže na nich není vidět posun nebo rotace. Nejlepším řešením je Normal mapping 8.2.1. Je také možné přepočítat normály v geometry



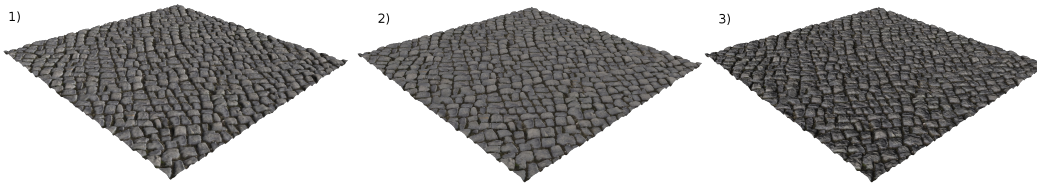
Obrázek 8.1: Změna směru skutečné normály oproti předem definované při displacement mappingu. Původní normála - černě, skutečná normála nového trojúhelníku - zeleně

shaderu, ale lze pracovat pouze s aktuálním trojúhelníkem[1]. Pokud by se ponechaly původní normály, u některých objektů, například těch s jednolitou barvou, výrazně to sníží přínos displacement mappingu.

[1] Není ani možné využít adjacency, po průchodu teselačními shadery se stanou nevalidními



Obrázek 8.2: Ilustrace efektu displacement mappingu na normály modelu asteroidu: 1) původní model, s normálami přepočítanými na flat (pro lepší přehlednost modelu) 2) původní normály, po teselaci a aplikaci displacement mappingu 3) stejný model, po teselaci a aplikaci displacement mappingu, s normálami přepočítanými v geometry shaderu



Obrázek 8.3: Ilustrace řešení normál na modelu dlážděné cesty po teselaci a aplikaci displacement mappingu: 1) normály pro zobrazení se načítají z uložené normálové mapy 2) normály jsou původní, které se využily při samotném displacementu 3) normály jsou dopočítané v geometry shaderu - ploché stínovaní, pomalé

8.2 Normal mapping

Normal mapping je metoda využívající, podobně jako bump mapping, texturu k uložení podrobnější informace o normálových vektorech (a tedy o chování osvětlení)

8.2.1 S využitím object space

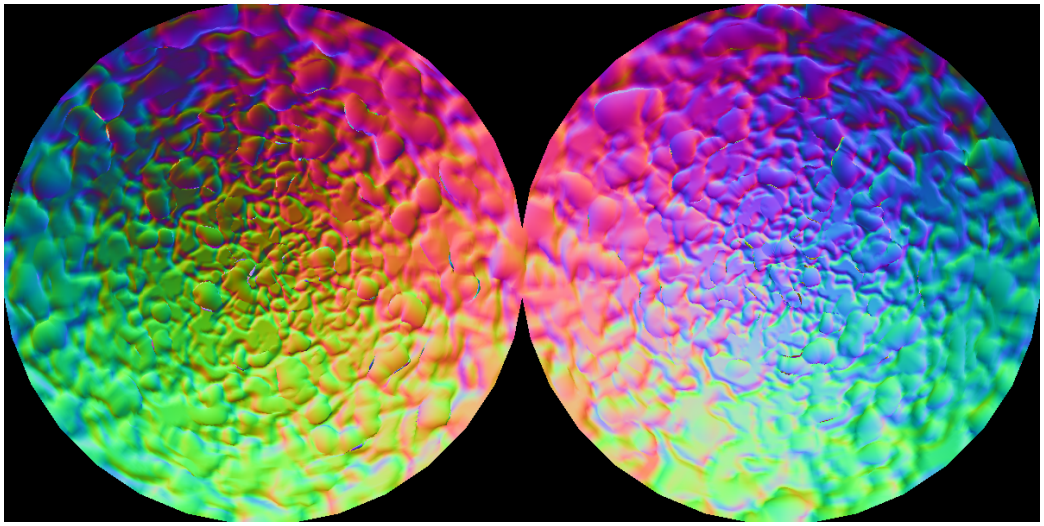
Pokud bychom uvažovali normal mapping v jeho minimální podobě, s normálovými vektory uloženými v object space normálové mapě, postačil by pro aplikaci normal mappingu jen dva řádky ve fragment shaderu, kde by se z textura načetla hodnota vektoru a transformovala dle natočení objektu

```
vec3 tmpVec3 = texture(normal_map, inTexCoord).rgb * 2.0 - 1.0;
nNormal = normalize(osg_NormalMatrix*tmpVec3);
```

kde první řádek přečte hodnotu normály pro aktuální fragment (pomocí jeho texturovacích souřadnic), a přepočítá jej z hodnoty barvy na normálu, a druhý jej transformuje pomocí normálové matice tak, aby odpovídal natočení modelu ve scéně. Přepočítání z barvy do vectoru je třeba, protože vektor (0, 0, -1) je v textuře zakódován jako barva s RGB hodnotami (128, 128, 255).

Uvažujme vektor představující normálu, načtený z textury. Původní normálový vektor, který je reprezentovaný právě načtenou barvou z textury, měl hodnotu (0, 0, 1), směřuje tedy přímo nahoru, ale díky způsobu reprezentace má barva načteného bodu hodnotu (0.5, 0.5, 1). Po vynásobení hodnotou 2 se zdvojnásobí jeho délka, vektor tak bude mít podobu

(1, 1, 2). Odečtením hodnoty 1 se všechny prvky vektoru sníží o tuto hodnotu, a vektor získá svou původní hodnotu (0, 0, 1).



Obrázek 8.4: Příklad object space normálové mapy

8.2.2 S využitím tangent space

Normal mapping s normálovou mapou s vektory uloženými v tangent space je složitější a výpočetně o něco náročnější. Protože jsou v této variantě normály uloženy s orientací k rovinně procházející příslušným trojúhelníkem, je třeba přepočítat je vzhledem k orientaci trojúhelníku v prostoru, nebo přepočítat polohy zdrojů světla do roviny zpracovávaného trojúhelníku. Tento přepočet se provádí pomocí tangentních a bitangentních vektorů. Ty je možné vypočítat pro jednotlivé vertexy dopředu, aby se ušetřil výpočetní čas v shaderech, nebo je vypočítat v průběhu, pro ušetření přenosové kapacity paměťové sběrnice a místa v paměti na grafické kartě, ať již tangentní i bitangentní vektor, nebo pouze tangentní (bitangentní není tak náročný na výpočet, jako tangentní).

Jak ale lze vidět na postupu výpočtu, je třeba mít model, pro který tangentní a bitangentní vektor počítáme, uložený bez sdílení vrcholů mezi trojúhelníky - výpočet těchto vektorů pracuje s vrcholy aktuálního trojúhelníku, a kvůli sdílení vrcholů by bylo možné uložit tyto vektory jen pro jeden výskyt zpracovávaného vrcholu.

V případě že provádíme výpočet předem, může mít tuto podobu

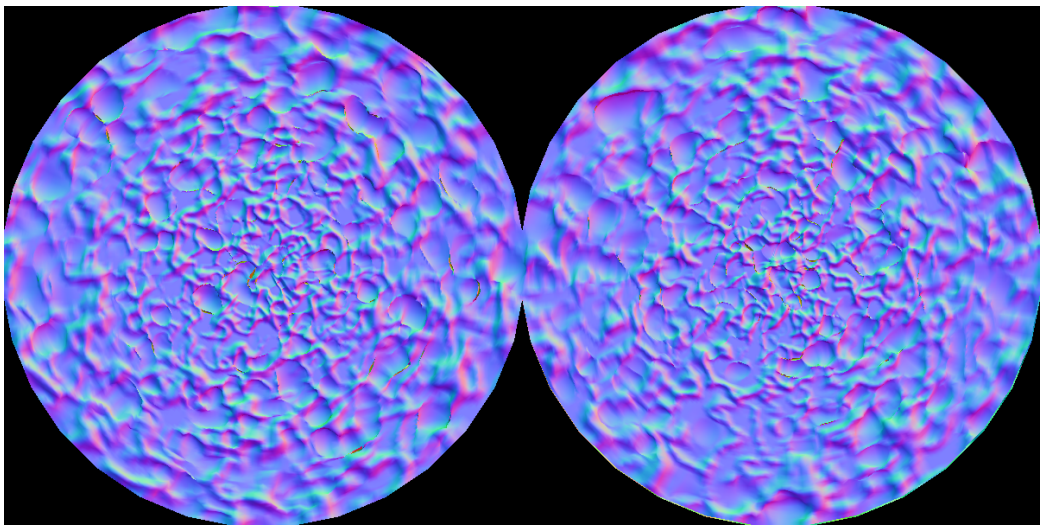
```
Vec3 p1, p2, p3; //body trojúhelníku
Vec2 tc1, tc2, tc3; //texturovací souřadnice trojúhelníku
//jejich inicializace
Vec3 pA = p2 - p1, pB = p3 - p1;
Vec2 tcA = tc1 - tc2, tcB = tc3 - tc1;

float div = tcA.v * tcB.u - tcA.u * tcB.v;

Vec3 tangent   = (pA * (-tcB.v) + pB * tcA.v) / div;
Vec3 bitangent = (pA * (-tcB.u) + pB * tcA.u) / div;
```

Ve vertex shaderu se pak vytvoří matice 3x3, tvořená tangentním, bitangentním a normálovým vektorem (TBN). Tato matice TBN transformuje vektory v tangent space, tedy vztažené k rovinně představované aktuálním trojúhelníkem, do object space, tedy správně natočené vzhledem k pozici a rotaci trojúhelníku ve scéně. Než ale ve fragment shaderu převádět každý vektor získaný z mapy do object space, je lepší vypočítat inverzní matici k TBN, která bude převádět vektory (nebo body) z object space do tangent space, s jejíž pomocí stačí pouze transformovat vektory určující směr kamery, a odkud dopadá světlo, do tangent space.

Problém s transformací světla pro celý trojúhelník do tangent space, místo opačné transformace tangent space normály do world space, může nastat u techniky deferred shading [2]. Zde není možné ušetřit výpočetní čas potřebný na transformaci normály do object space transformací pozice osvětlení, protože je potřeba uložit normálu ve world space (do kterého se transformuje z object space).



Obrázek 8.5: Příklad tangent space normálové mapy

8.2.3 Normal mapping ve spojitosti s displacement mappingem

Díky přítomnosti TCS, který může manipulovat s celým trojúhelníkem, je možné, pokud není i základní model příliš složitý, přesunout výpočet tangentního a bitangentního vektoru do shaderu. Toto by sice bylo možné i předtím, pomocí geometry shaderu, ale použití teselace umožňuje mít základní model poměrně jednoduchý, a tím snížit počet výpočtu prováděných v TCS. V TCS je také možné vypočítat základní normálový vektor.

8.3 Parallax mapping

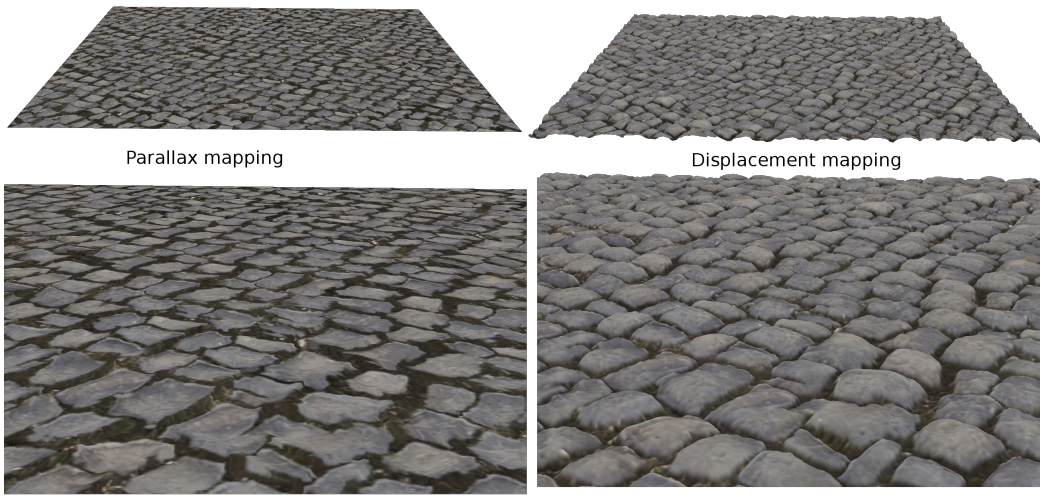
Parallax mapping se je záležitostí fragment shaderu, s předpočítáním některých hodnot pro TBN matici ve vertex shaderu. Je třeba určit kterým směrem se mají texturovací souřadnice posunout. To je možné podobně jako u normálového mapování, pomocí inverzní transformace vektoru, směřujícího z kamery do právě zpracovávaného fragmentu, maticí složenou z

[2] odložené stínování - osvětlení se počítá dodatečně, a je třeba mít uložené normály pro každý fragment ve world space

tangentového, bitangentového a normálového vektoru. Touto transformací získáme vektor v tangentním prostoru aktuálního trojúhelníku, a jeho složky X a Y pak tvoří dvojrozměrný vektor, představující směr z aktuálního framentu k pomyslné ose jdoucí z kamery po ose Z, převedené také do tangent space. Z tohoto podvektoru se následně vypočítá osa, po které se posunou texturovací souřadnice. Kód výpočtu[12] může mít tuto podobu

```
vec3 nView = normalize(viewDirT);
float height = texture2D(parmap, fTexCoord).b;
height = height * scale - bias;
newfTexCoord= teTexcoord + (height * nView.xy);
```

kde viewDirT jsou interpolované souřadnice aktuálního fragmentu transformované do tangent space aktuálního trojúhelníku, scale je míra posunu, pokud jsou textury čtvercové, pokud nejsou, je třeba mít posun jako dvojrozměrný vektor, s posunem pro každou osu, a bias určuje střed načtené hodnoty, je tedy závislý na velikosti scale - pokud má velikost poloviny scale, žádný posun představuje v textuře hodnota 0.5 (128), pokud má nulovou hodnotu, jakákoliv hodnota v textuře jiná než 0 znamená posun.

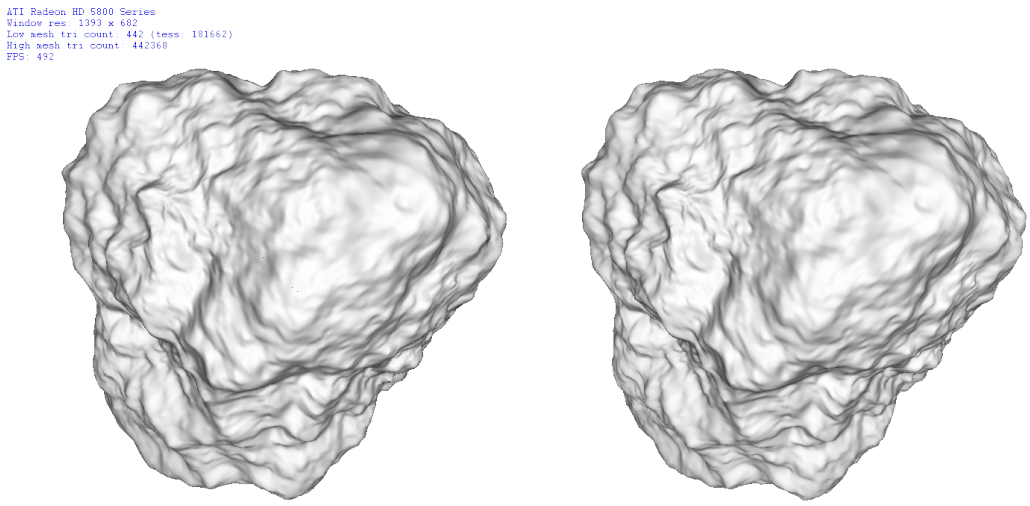


Obrázek 8.6: Parallax mapping ve srovnání s displacement mappingem

Kapitola 9

Povaha demonstrační aplikace

Demonstrační aplikace byla tvořena s cílem vytvořit jednoduchý editor shaderů, s možností načíst model vygenerovaný například v programu Blender, zkonvertovat ho do formátu knihovny OpenSceneGraph, a nastavit jej pro další použití. Využívá knihovnu Qt pro správu GUI.



Obrázek 9.1: Napravo je původní objekt, nalevo je zjednodušený objekt vykreslený s využitím displacement a normal mappingu

9.1 Analýza výkonnosti

Aplikace podporuje grafické karty společností AMD i NVidia, skrze rozhraní GPUperfAPI a NVperfKIT, ačkoliv NVperfKIT se ukázal být nespolehlivý. S využitím funkcionality knihovny Qt je možné čítače analyzovat jednou, průběžně, a jejich výstup uložit do souboru. V případě použití analýzy s intervalem je dobré mít na paměti, že pro získání všech hodnot vykreslují obě API jeden snímek opakovaně, v případě velmi náročných objektů tak může program strávit většinu času analýzou.

9.2 Konverze načtených objektů

Prvním krokem po načtení objektů je automatická konverze indexů na primitiva typu Triangles, protože například formát Wavefront .obj OSG načítá jako Triangle_strip. To by normálně nevadilo, ale pokud by chtěl uživatel převést některý z načtených modelů na primitiva Patches, bylo by opět nutné převést model na Triangles.

Součástí konverze je dále možnost převést data ze standardních polí (NormalArray, VertexArray), na upravitelná VertexAttribArray. V rámci konverze je také možné vypočítat druhou sadu texturovacích souřadnic, jak bylo popsáno v kapitole 5.

9.3 Strom scény a seznam statesetů

Jeden z důležitých prvků je možnost měni scénu. Pro ladění je připraveno odkladiště aktuálně nevyužívaných částí stromu. K dispozici je možnost úprav základních typů uzlů (Group, MatrixTransform, PositionAttitudeTransform), pomocí nichž je možné združovat a umísťovat objekty do scény, a úprava listů stromu 9.2.

9.4 Nastavování uniformů a state atributů

Pro potřeby ladění a testování shaderů je implementována možnost nastavovat state atributy, a uniformy. Ty nejsou implementovány všechny, protože byly doimplementovávány podle potřeb aktuálních potřeb při ladění shaderů. Jsou implementovány tyto:

- Uniformy
  - Celočíselné hodnoty - float, int, unsigned int, bool
  - Vektory - float, int, unsigned int, bool

  Jsou tedy implementované základní možnosti pro ovládání shaderů. Ve zdrojovém kódu je jejich nastavování řešeno jednoduchým parsováním textových řetězců z tabulky, kde je může upravovat uživatel. Tato jednoduchost má i stinnou stránku, a to že není možné jednoduše upravovat složitější uniformy, jako jsou například matice.

- Atributy
  - PolygonMode - nastavuje způsob rasterizace, možnosti jsou: plný model, drátový model, pouze body
  - Texture - reprezentuje texturu. Tu je možné měnit, načítat ze souboru, nastavovat její parametry, kterými jsou mag a min filtr, a wrap (způsob opakování textury v momentě kdy jsou texturovací souřadnice mimo rozsah)
  - Program - je možné nastavit existující program z panelu pro editaci shaderů. Mimo tento panel není možné tento state atribut upravovat

  Na rozdíl od uniformů má každý state atribut vlastní panel s nastavením, výběr panelu probíhá v momentě aktivace (označení) atributu. Do tohoto formátu je v plánu převést i uniformy.

9.5 Panel se shadery

Umožňuje modifikovat shadery, překládat je, nastavovat (zejména binding VertexAttribArray), a ukládat. Ukládá se celý atribute, skrz standardní ukládání souborů OSG - je vytvořen malý strom, který slouží jako schránka pro aktuální shader program. OSG totiž nenabízí možnost uložit program samostatně.

9.6 Implementované příklady

V programu je implementováno několik demonstračních příkladů

- Icosahedron - je převzatý z příkladu OpenSceneGraphu, který byl zase převzatý z [**?**]. Je nastavený jako úvodní příklad - slouží jako ověření funkčnosti teselace.
- Variable Teselation - Jednoduchý příklad ukázky možností teselace pro teselaci pouze jedné oblasti. Při aktivaci požádá o načtení textury, podle ní pak teseluje své oblasti různým stupněm.
- Simple Displacement - Dlážděná plocha - ukázka displacement mappingu s využitím normal mappingu na korekci normal
- Parallax mapping - Dlážděná plocha - ukázka srovnání displacement mappingu s parallax mappingem.

Kapitola 10

Závěr

Displacement mapping je poměrně zajímavá metoda, i když má jistá úskalí. Jak lze vidět v kapitole 4.2, je třeba ji používat s rozumem. Není možné vše teselovat na maximální stupeň, protože množství nových bodů může být enormní. Na každý jeden trojúhelník může vzniknout přes 6000 nových, například hlava na obrázku3.3 má po teselaci z původních 9k polygonů 3.5M, drátový model je tak hustý, že při nastavení PolygonMode na Wireframe skrz něj nelze vidět.

Displacement mapping může být také náročnější na předpřípravu modelů, jak je ukázáno v 5. Aby se zabránilo vzniku mezer v modelu, je třeba buď mít připravené extra texturovací souřadnice, nebo upravit geometrii. Několik způsobů úpravy geometrie bylo v této kapitole, ten nejméně invazivní byl i naimplementován.

Ačkoliv by naprogramová aplikace potřebovala (dle potřeby) rozšířit, je lze ji reálně využít pro přípravu jednoduššího modelu, například do nějaké hry. K exportu pro tyto účely by se hodilo doplnění o vytvoření popisu v XML, s možností nastavit uzlům parametry a typ. Konkrétní nastavení by ale záleželo na účelu. Budeme-li uvažovat model kosmické lodi s možností připojit další prvky, tyto by mohly být v modelu tvořeny prázdným pozicovacím nodem (například PositionAttitudeTransform), a v XML (aby se nemuselo zasahovat do kódu OpenSceneGraphu) by byly dodatečné parametry, navázané na uzly stromu objektu.

Kapitola 11

Přílohy

11.1 Obsah DVD

- Textová část
- Program - binární
- Program - zdrojový kód
- Readme.pdf

Literatura

[1] R600 (ASIC). http://en.wikipedia.org/wiki/Radeon_HD_3000_Series#Hardware_tessellation, 13 May 2013? [cit. 2013-05-24].

[2] Tangent space. http://en.wikipedia.org/wiki/Tangent_space, 19 April 2012? [cit. 2013-05-28].

[3] TruForm. https://en.wikipedia.org/wiki/TruForm, 19 April 2012? [cit. 2013-05-28].

[4] GPU Gems 2. 2005-01-01 [cit. 2013-01-08].

[5] Rendering Pipeline Overview. http://www.opengl.org/wiki/Rendering_Pipeline_Overview, 2012-11-02 [cit. 2013-01-08].

[6] GeForce 8 Series. `http://en.wikipedia.org/wiki/GeForce_8_Series`, [cit. 2013-01-08].

[7] R600 (ASIC). `http://en.wikipedia.org/wiki/Radeon_R600#Unified_shaders`, [cit. 2013-01-08].

[8] Blinn, J. F.: Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, New York, NY, USA: ACM Press, 1978, ISSN 0097-8930, s. 286–292, doi:10.1145/800248.507101. URL http://dx.doi.org/10.1145/800248.507101

[9] Cantlay, I.: DirectX 11 Terrain Tessellation. 2011.

[10] Cook, R. L.: Shade Trees. 1984. URL http://www.gdv.informatik.uni-frankfurt.de/lehre/ss2007/GDV/Uebung/Paper/99-1-cook-shadetrees.pdf

[11] Dudash, B.: My Tessellation has Cracks! (and solutions to other common tessellation problems). 5 March 2012[cit. 2013-05-26]. URL http://techreport.com/review/21404/crysis-2-tessellation-too-much-of-a-good-thing

[12] Hupka, D.: Computer game project focused on light effects. 2012. URL https://merlin.fit.vutbr.cz/wiki/index.php/Graphics_Projects_2012

[13] Kaneko, T.; Takahei, T.; Inami, M.; aj.: Detailed shape representation with parallax mapping. In *In Proceedings of the ICAT 2001*, 2001, s. 205–208.

[14] Lobel, R.: Realtime Per-Pixel Displacement Mapping on Arbitrary Geometry. 2004.
URL http://www.divideconcept.net/papers/RPPDM-RL04.pdf

[15] Luebke, D.; Humphreys, G.: How GPUs Work. 2007.
URL http://www.cs.virginia.edu/~gfx/papers/pdfs/59_HowThingsWork.pdf

[16] Shopf, J.; Barczak, J.; Oat, C.; aj.: March of the Froblins: simulation and rendering massive crowds of intelligent and detailed creatures on GPU. In *Proceeding SIGGRAPH '08 ACM SIGGRAPH 2008 Games*, 2008, s. 52–101.

[17] Tatarchuk, N.; Barczak, J.; Bilodeau, B.: Programming for Real-Time Tessellation on GPU.
http://www.amddevcentral.com/gpu_assets/Real-Time_Tessellation_on_GPU.pdf.

[18] Wasson, S.: Crysis 2 tessellation: too much of a good thing? 16 August 2011? [cit. 2013-05-26].
URL http://techreport.com/review/21404/
crysis-2-tessellation-too-much-of-a-good-thing