Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Modelování povrchových detailů pomocí mapování textur

2017

Petr Kaňák, DiS.

Vedoucí práce: Mgr. Tomáš Kühr, Ph.D.

Studijní obor: Aplikovaná informatika, prezenční forma

Bibliografické údaje

Autor:	Petr Kaňák, DiS.
Název práce:	Modelování povrchových detailů pomocí mapování textur
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2017
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Tomáš Kühr, Ph.D.
Počet stran:	61
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Petr Kaňák, DiS.
Title:	Modeling surface details using texture mapping techniques
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2017
Study field:	Applied Computer Science, full-time form
Supervisor:	Mgr. Tomáš Kühr, Ph.D.
Page count:	61
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Cílem bakalářské práce bylo analyzovat v praxi nejpoužívanější techniky pro modelování povrchových detailů pomocí mapování textur převážně v aplikacích pracujících v reálném čase. Ve výsledné aplikaci je možno zobrazit jednotlivé techniky, poukázat na jejich klady i zápory, dále pak spustit test, který je porovná z hlediska rychlosti zpracování.

Synopsis

The aim of the thesis was to analyze the most widely used techniques for surface details using texture mapping mainly in applications working in real-time. In the resulting application can display various techniques, to identify their pros and cons, then run a test that compares them in terms of processing speed.

Klíčová slova: OpenGL; Sponza; Textura; Tessellation; GLFW; GLEW; SOIL; Texture mapping; Normal mapping; Parallax mapping; Steep parallax mapping; Relief mapping; Parallax occlusion mapping; Displacement mapping

Keywords: OpenGL; Sponza; Texture; Tessellation; GLFW; GLEW; SOIL; Texture mapping; Normal mapping; Parallax mapping; Steep parallax mapping; Relief mapping; Parallax occlusion mapping; Displacement mapping

Rád bych poděkoval panu doktoru Kührovi za vedení této bakalářské práce. Dále pak členům mé rodiny za poskytnuté zázemí.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam zdrojových kódů

1 Úvod

Tato práce se zabývá analýzou v praxi nejpoužívanějších technik pro modelování povrchových detailů v aplikacích, které jsou vykreslovány v reálném čase. Tyto poznatky jsou pak využity v aplikaci *TMTechs* (Texture mapping techniques). V této aplikaci je možné popisované modelovací techniky: vizuálně porovnávat, ukázat si jejich silné a slabé stránky a tyto techniky porovnat z hlediska rychlosti jejich vykreslování.

Text samotné práce začíná teoretickou kapitolou. V této kapitole se nejprve zabývám nezbytnými základními pojmy počítačové grafiky. Dále zmiňuji souřadnicové systémy, které hrají důležitou roli pro modelovací techniky. Potom následuje subkapitolka ohledně rozhraní OpenGL a jazyka GLSL. GLSL je určen pro psaní programů na grafické kartě. Nakonec se v této teoretické kapitole věnuji Phongovu osvětlovacímu modelu, pomocí něhož je v aplikaci implementováno osvětlení.

Další kapitola se zabývá popisem samotných modelovacích technik. Jsou zde také zmiňovány silné a slabé stránky technik. Následuje zkrácená kapitola zabývající se porovnáním modelovacích technik. Text práce je ukončen programátorskou a uživatelskou příručkou.

2 Teoretická část

V teoretické části budou popsány veškeré nutné informace k pochopení implementačních detailů technik pro modelování povrchových detailů.

2.1 Základní pojmy

V této kapitole bude čtenář seznámen se základními pojmy spojenými s počítačovou grafikou, které budou v následujícím textu používány.

Vrchol je soubor dat. Tyto data nejčastěji popisují např: souřadnice pozice bodu ve specifickém souřadnicovém systému, souřadnice normálového vektoru v souřadnicovém systému nebo souřadnice v textuře. V podstatě na 3D objekt, takzvaný model, se lze dívat jako na kolekci vrcholů nebo polygonů. Samozřejmě model lze definovat i jinak, ale v našem případě se na model budeme dívat jako na kolekci vrcholů nebo polygonů.

Polygon je jeden z nejjednoduších stavebních prvků modelu. Polygon je pak tvořen z několika vrcholů. Takže příkladem polygonu může být trojúhelník, čtyřúhelník či jiný jednoduchý geometrický objekt. Náš polygon bude mít jednu důležitou vlastnost, a to že bude vždy konvexní. Tato definice polygonu je sice zjednodušená, ale pro naše účely nám bude plně dostačovat. Podrobnější definici můžeme nalézt v [2].

Mesh je kolekce polygonů definující nějakou část modelu. Jedná se například o kolo, které je součástí modelu automobilu.

Model je kolekce meshů. Pod pojmem model si můžeme představit třeba automobil, lampu nebo budovu ve městě.

Scéna je tvořena kolekcí modelů. Příkladem scény může být třeba město.

Texel (texture element) je základní jednotkou textury. Je to obdoba pixelu jako základní jednotky obrazu.

Fragment je kolekce hodnot vytvořených ve fázi rasterization. Fragmenty pokrývají oblast primitiv (polygony) a jsou vytvořeny na základě dat vrcholů.
Na fragment se lze dívat jako na potenciální pixel ve výsledném obraze. Fragment nese například informace o: souřadnicích pozice v určitém souřadnicovém systému, souřadnicích normálového vektoru v určitém souřadnicovém systému, ...). Podrobněji se zabývá vznikem fragmentů podkapitola Popis jednotlivých fází PZŘ - Rasterization.

Framebuffer je dvourozměrné vícevrstvé (nemusí být) pole, do kterého se ukládají data z výstupu programovatelného grafického řetězce. Framebuffer je složen

z několika polí (bufferů), která nesou data o barvě (color buffer), hloubce (depth buffer) a o pomocných informacích (stencil buffer). Pro výsledný obraz je nejdůležitější color buffer. Zbylé dva buffery pak slouží převážně k provádění testů. Framebuffer můžeme nastavit tak, že některá z vrstev bude ignorována.
Díky tomu lze dosáhnout jednovrstvého framebufferu. Důležitým pojmem je defaultní framebuffer, který je spojen s oknem aplikace. V tomto framebufferu jsou pak uložena data (obraz), která vidíme na zobrazovacím zařízení (monitor, display, ...). Pokud implementujeme složitější techniky, můžeme využít framebuffer i jinak. Například při implementaci techniky shadow mapping, která zprostředkovává stíny ve scéně. Dokonce lze nastavit i několik více vrstvých framebufferů viz. deferred shading. Více se lze dozvědět v [11].

2.2 Souřadnicové systémy

Pro správné pochopení dalšího textu je nezbytné popsat specifické souřadnicové systémy. Názvy některých souřadnicových systémů se mohou v literatuře lišit, jejich význam ale bývá totožný.

Object-space je souřadnicový systém definovaný vzhledem k lokálnímu počátku. Většina modelů je definována vzhledem k tomuto souřadnicovém systému z praktických důvodů. Tyto důvody lze popsat následujícím způsobem:

- Pokud se model vyskytuje ve scéně vícekrát, nebylo by ho moudré definovat vzhledem k *world-space*, protože by docházelo ke zbytečné duplicitě dat.
- Při technice nazvané instancování je model definován pouze jednou vzhledem k *object-space*. Dále je pak definována kolekce transformačních matic z *object-space* do *world-space*. Tato data jsou předána grafické kartě, která je pak schopna tyto modely zobrazit s pomocí transformačních matic. Díky této technice můžeme ušetřit značné množství místa v paměti grafické karty.
- Data modelu bývají rozumně nadefinována vůči osám *object-space*. To vede například k intuitivnímu chování modelu při transformaci v podobě rotace modelu.

World-space je souřadnicový systém definovaný vzhledem ke globálnímu počátku. Tento souřadnicový systém bývá pevně spojen se světem, který modelujeme. Transformační matice mezi *object-space* a *world-space* většinou vzniká složením afinních operací jako posunutí, otočení a změna měřítka. Dále v textu bude transformační matice označena jako *os-to-ws* nebo *gWorld*.

View-space je souřadnicový systém definovaný vzhledem k pozici kamery nebo, jinak řečeno, vůči pozici pozorovatele. Tento systém reprezentuje pohled, jakým svět vidí pozorovatel. Transformační matice mezi *world-space* a *view-space* vzniká složením afinních operací posunutí a otočení. Dále v textu bude transfor-

mační matice označena jako *ws-to-vs* nebo *gView*.

Clip-space je speciální souřadnicový systém. OpenGL očekává, že souřadnice pozice vrcholů bude na konci jedné z fází programovatelného zobrazovacího řetězce: *vertex-shader*, *geometry-shader* nebo *tessellation-evaluation-shader* právě v tomto souřadnicovém systému. Více se o tomto souřadnicovém systému můžeme více dozvědět v příslušné literatuře [4]. Zjednodušeně se jedná o souřadnicový systém, který vyžaduje, aby souřadnice pozice vrcholů byly v určitém rozsahu. Všechny ostatní vrcholy, které toto omezení nesplňují, budou ořezány (clip). Pro naše účely postačí tato zjednodušená definice. Transformační matici mezi *view-space* a *clip-space* pak bude dále v textu mé práce označena *vs-to-cs* nebo *gProj*.

Tangent-space je souřadnicový systém definovaný vzhledem k určitému vrcholu modelu. K získání transformační matice mezi určitým souřadnicovým systémem a *tangent-space* musíme mít pro každý vrchol definované tři vektory. Tyto vektory budeme označovat jako *normal*, *tangent* a *bitangent* (*bitangent* bývá v některých textech označována jako *binormal*). Vektory dohromady tvoří bázi *tangent-space*. Jméno transformační matice pak bude určeno podle toho, z jakého souřadnicového systému transformujeme souřadnice. Data jsou definována vůči *world-space*, takže transformační matice bude dále v textu označována jako *ws-to-ts* nebo *gWsToTs*. Bližší informace o definici bázových vektorů a o definici transformační matice budou podrobněji popsány v další kapitole.

Další informace V textu se dále vyskytují různé varianty transformačních matic jako *gVP* nebo *gWVP*. Tyto matice a jim podobné vznikají společným vynásobením několika transformačních matic. Př: $gVP = gView * gProj$ nebo $gWVP = gWorld * gVP = gWorld * gView * gProj$. Specielním případem je pak matice $gNormal = transpose(inverse(gWorld))$, která je určena k transformaci normálových vektorů. Její specifický formát vychází z potřeby, aby nevznikaly problémy při změně měřítka modelu. Funkce inverse vyrobí inverzní matici ze vstupní matice a funkce transpose vyrobí transponovanou matici ze vstupní matice. Více informací nalezneme [13].

2.3 OpenGL

OpenGL (Open Graphics Library) je průmyslový standart specifikující multiplatformní rozhraní (API), které slouží k vykreslování 2D a 3D vektorové grafiky. Toto rozhraní je navrženo ke spolupráci s grafickou kartou, takže může být dosaženo hardwarově akcelerovaného vykreslování. Využití této knihovny je všestrané: od uplatnění ve vědě přes stojírenctví až k zábavnímu průmyslu. Standard OpenGL byl nejprve spravován konsorciem ARB (Architecture Review Board), jehož členy jsou firmy jako např. NVIDIA nebo AMD. Nicméně v roce 2006 byla kontrola nad OpenGL standartem předána nezávaznému kon-

sorciu Khronos Group (KG). Z tohoto důvodu jsou nové verze OpenGL pravidelně vydávány skupinou KG. Nové verze rozhraní přináší novou funkcionalitu. Každá nová funkcionalita (dále jen rozšíření) je opatřena identifikátorem např: GL_ARB_compute_shader nebo GL_EXT_texture_sRGB. Tento způsob označování je výhodný, protože chceme-li zjišťovat, jestli určité zařízení podporuje určitou verzi OpenGL, stačí jen zjistit, podporuje-li určitou množinu rozšíření. Výhodou tohoto systému je také skutečnost, že členové KG mohou poskytovat další rozšíření, která jsou třeba spjata pouze s určitým hardwarem nebo mají inovativní funkcionalitu. Tato rozšíření pak mají v názvu nezaměnitelné identifikátory (NVIDIA - NV, ATI Technologies - ATI). Nejnovější verze této knihovny je 4.5 ku dni 29. 6. 2017. Více informací lze nalézt v [1].

2.3.1 Zobrazovací řetězec knihovny OpenGL

Zobrazovací řetězec (anglicky rendering pipeline nebo function pipeline) popisuje systém, který převádí trojrozměrnou scénu, popsanou různými objekty (modely, světelné zdroje, kamera, ...), do dvojrozměrného prostoru, který je nejčastěji reprezentován obrazem na monitoru (data se zapíší do defaultního framebufferu). Tento systém je rozdělen do série na sebe navazujících fází (anglicky stages), kde výstup aktuální fáze odpovídá vstupu následující fáze. Z pohledu historie tento systém prodělal mnoho změn. Jedna z nejzásadnějších změn byl přechod z fixního zobrazovacího řetězce na programovatelný řetězec. To se projevilo tak, že některé fáze jsou přímo programovatelné pomocí speciálních programů (anglicky shader, v mé práci dále jen shader). Ve své práci se budu zabývat pouze programovatelným zobrazovacím řetězcem (PZŘ).

2.3.2 Grafické karty s programovatelným zobrazovacím řetězcem

Grafické karty s PZŘ nám umožňují pomocí shaderů programovat některé z fází PZŘ. Tyto shadery mohou být psány buďto v nižších programovacích jazycích, nebo ve vyšších. Ve své práci jsem použil vyšší programovací jazyk glsl, o kterém bude řeč dále. Následující popis PZŘ je zjednodušen, podrobnější popis lze nalézt v [3, 4]. Na obrázku 1 můžeme vidět jednotlivé fáze PZŘ.

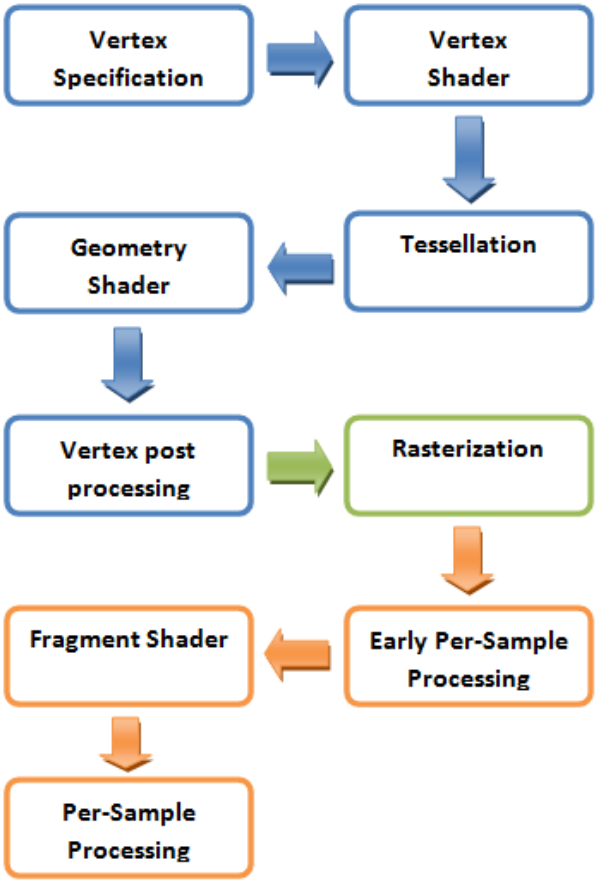
PZŘ lze dělit z několika hledisek.

Programovatelné

- Vertex Shader
- Tessellation
- Geometry Shader
- Fragment Shader

Neprogramovatelné

- Ostatní



Obrázek 1: Fáze zobrazovacího řetězce.

Povinné a nepovinné není jednoduché přesně rozdělit. Podrobnější informace mohou být nalezeny v [3, 4].

Povinné

- Vertex Specification
- Vertex Shader
- Rasterization
- Fragment Shader

Nepovinné

- Tessellation
- Geometry Shader

2.3.3 Popis jednotlivých fází PZŘ

Vertex Specification je počáteční fáze. Vstupem PZŘ jsou vrcholy reprezentované volitelným počtem dat (souřadnice pozice vrcholu ve specifickém prostoru nebo souřadnice normálových vektorů ve specifickém prostoru), které tvoří geometrická primitiva (body, linky, trojúhelníky, záplaty, ...). Tato data vrcholů jsou uložena ve vhodné struktuře či strukturách, převážně typu pole. Na vstupu PZŘ mohou být použity i další pomocné struktury jako pole indexů (používá se

k zamezení duplicit dat vrcholů) nebo pole přechodových matic (používá se při instancování).

Vertex Shader dále jen VS, je povinná programovatelná fáze. Shader této fáze je spuštěn nad každým vrcholem. V této fázi dochází k operacím typu: transformace dat vrcholů mezi souřadnicovými sytémy, výpočet přechodových matic nebo výpočet nových dat ze stávajících. Data z této fáze jsou předána do jedné z nepovinných fází: tesselation nebo geometry shader a nebo na vstup fáze vertex post processing.

Tessellation je nepovinná fáze, kde geometrická primitiva typu záplaty (anglicky patch) definovaná řídícími body jsou vhodně rozdělena (anglicky subdivided) do menších celků. Pokud chceme tuto nepovinou fázi využít, je nutno tuto funkcionalitu aktivovat pomocí série příslušných OpenGL API. Dále je pak nutné předat fázi Vertex Specification sdělení (nastavením příslušných OpenGL API), že vstupní data jsou geometrická primitiva typu záplaty. Proces tessellace je rozdělen do tří podfází, ze kterých dvě jsou programovatelné. Těmito podfázemi jsou: Tessellation Control Shader (TCS), tessellation primitive generator (TPG) a Tessellation Evaluation Shader (TES).

TCS je programovatelná fáze. Tato fáze hlavně rozhoduje o tom, do jaké míry bude záplata rozdělena pomocí stanovení vnitřního a vnějšího faktoru rozdělení (anglicky tessellation level).

TPG je fixní fáze zodpovědná za vytvoření dat, která budou použita
k výpočtu nových dat vrcholů, provedeného na základě dat získaných ze záplat a z faktorů rozdělení vypočítaných v předchozí fázi.

TES je programovatelná fáze zodpovědná za výpočet dat nových vrcholů pomocí lineární interpolace na základě dat z předchozí fáze. Data z této fáze jsou předána do geometry shader a nebo do fáze vertex post processing.

Geometry Shader je opět nepovinná programovatelná fáze. Tato fáze přímá data buď z fáze VS nebo z fáze Tessellation. Program této fáze je spuštěn nad každým primitivem. (Př.: 12 dat vrcholů tvoří 4 trojúhelníky. Program je spuštěn nad každým trojúhelníkem.) Uvedená fáze je schopna vygenerovat nová primitiva či odstranit stávající. Je dále shopna změnit typ primitiva. (Př.: Pokud jsou vstupními primitivy trojúhelníky, je shader schopen tyto trojúhelníky změnit třeba na úsečky. Tímto způsobem jsme schopni napsat třeba shader zobrazující normálové vektory trojúhelníků). Více se lze dozvědět v [3, 4].

Vertex post processing je rozdělena na několk podfází, které lze řídit pomocí funkcí OpenGL. První podfáze je Primitive Assembly,
kde jsou data vrcholů shlukována do určitých primitiv (trojúhelníky, čtverce,

úsečky, ...). Dále dojde k ořezu primitiv (clipping) podle pohledového tělesa (viewing frustum). Tímto procesem docílíme, že primitiva, která nejsou vidět, budou smazána, a primitiva která nejsou vydět jen z časti, budou upravena tak, aby byly odstraněny jen jejich neviditelné časti. Primitiva, která jsou orientována směrem od pozorovatele (facing away), lze též odstranit pomocí Face culling. Díky tomu můžeme zabránit provádění výpočtů nad primitivy, která nelze ve vysledném obrazu vidět. (Př.: Nechť máme krychli, kde jsou viditelné pouze tři přední strany. Zbylé tři strany nelze vidět. Díky cullingu jsme schopni tyto strany včas odstranit, a tím zefektivnit další výpočty.)

Rasterization je povinnou neprogramovatelnou fází. V této fázi dochází k mapování primitiv do pomyslné mřížky (raster). Tento jev si můžeme představit následovně. Máme definovanou mřížku a zjišťujeme kolik buněk, této mřížky, pokrývá dané primitivum. Těmto buňkám budeme říkat fragmenty. Pomocí lineární interpolace jsou následně poskytnuty data vrcholů fragmentům. Rozhodnutí, která data budou poskytnuta fragmentům může částečne ovlivnit programátor prostřednictvím definice vstupních a výstupních dat v programovatelných fázích PZŘ. Fragmenty jsou pak uloženy ve speciální kolekci. Následující fází je pak early Per-Sample Processing nebo fragment Shader.

Early Per-Sample Processing je fáze, kde lze provést určité testy z fáze Per-Sample Processing a tím zamezit zbytečným výpočtům ve fázi fragment shader. Typy testů: scissor test, stencil test, depth test a další. Více o chování těchto testů se lze dozvědět v [3, 4].

Fragment Shader je programovatelnou fází, kde na základě dat fragmentu vypočteme výslednou barvu fragmentu. Tato barva fragmentu pak může být konečnou barvou určitého pixelu ve výsledném obraze. Nicméně rozhodnutí, zda-li barva fragmentu bude výslednou barvou v obraze rozhodne další fáze.

Per-Sample Processing je fáze, kde fragment podstupuje řadu testů. Chování těchto testů lze ovlivnit pomoci OpenGL API. Pokud fragment projde těmito testy jsou určité hodnoty fragmentu (barva, hloubka) zapsáný do framebufferu nebo ovlivní určité hodnoty ve framebufferu (blending).

2.4 Jazyk GLSL

Jazyk GLSL (OpenGL Shading Language, dále jen GLSL) je vyšší programovací jazyk se syntaxí velice podobnou jazyku C, který se používá pro psaní shaderů. Také zde jsou zakomponovány některé mechanismy z C++, jako např. deklarace proměnných na libovolné pozici v kódu. GLSL je multiplatformní jazyk včetně GNU/Linux, Mac OS a Windows. Další podrobnosti lze nalézt v [5, 6].

2.4.1 Základní informace

Zdrojový kód shaderu musí vždy začínat specifikací verze jazyka GLSL. Toho dosáhneme použitím speciální preprocesorové direktivy version. Obecný formát direktivy: #version <verze_GLSL>. Příklad direktivy: #version 450. Dále pak text programu musí obsahovat základní funkci main s návratovou hodnotou typu void. Tato funkce nemá vstupní argumenty a slouží jako hlavní funkce shaderu. Jazyk GLSL má podobnou politiku z hlediska rozšíření (extensions) funkcionality, jako knihovna OpenGL. Pokud chceme ve zdrojovém kódu použít speciální funkcionalitu, musíme ji tam vložit pomocí další speciální preprocesorové direktivy extension. Obecný formát direktivy: #extension <jmeno_rozsireni> : <chovani>. Příklad direktivy: #extension GL_OES_EGL_image_external : require.

2.4.2 Operatory

Jazyk GLSL poskytuje stejnou paletu operátorů jako jazyk C, kromě operátorů souvisejících s ukazateli, které nejsou v GLSL podporovány. Oproti jazyku C je zde navíc operátor swizzle, který umožňuje přeskládání nebo opakování složek u datového typu vektor. Příklad: vec4 v1 = vec4(1.0, 2.0, 3.0, 4.0); vec4 v2 = v1.xxwy;

2.4.3 Datové typy

Jazyk GLSL obsahuje kromě skalárních typů (bool, int, uint, float a double) i datové typy specifické pro grafické výpočty jako vektorové (vec2, vec3 a vec4) a maticové (mat2, mat3 a mat4). Dále jsou zde i speciální datové typy - takzvané opaque typy. Tyto typy nám umožňují přístup ke speciálním externím objektům jako data textur či atomické čítače. Opaque typy je možné deklarovat pouze jako globální proměnné nebo jako argumenty uživatelsky definovaných funkcí. GLSL nám umožňuje definovat i struktury, díky čemuž lze lépe organizovat data. Více můžeme nalézt v [7].

2.4.4 Funkce

Kromě hlavní funkce main, která musí být v shaderu vždy přítomná a musí mít pevně daný formát, umožňuje GLSL tvorbu uživatelsky definovaných funkcí. Dále pak oplývá širokou paletou vestavěných funkcí převážně opět zaměřených na grafické výpočty. Například funkce: reflect, refract, dot (skalární součin), cross (vektorový součin) nebo normalize (normalizace vektoru).

2.4.5 Direktivy preprocesoru

Pro řízení předzpracování zdrojového kódu se v GLSL, stejně, jako v C, používají direktivy preprocesoru. Balíček Direktiv v GLSL je skoro totožný s obdobným

balíčkem v jazyce C, avšak například direktiva #include chybí. Kromě již zmíněných direktiv #extension a #version jsou hojně používané direktivy #if, #elif, #else a #endif pro podmíněný překlad a k implementaci takzvaných uber shaders.

2.4.6 Uber shader

Uber shader je jeden z přístupů jak organizovat shadery. Jako uber shader považujeme:

- Komplexní shadery z velkým počtem větvení (branching).
- Komplexní jediný soubor, ve kterém se vyskytují direktivy preprocesoru pro podmíněný překlad. Tento soubor je pak kompilován vicenásobně s různými parametry (pomocí direktivy define) a generuje permutace shaderu.

Primární snaha je pak snížit počet větvení v shaderu. Toho se dá docílit použitím právě již zmíněných direktiv preprocesoru, určených pro podmíněný překlad. V neposlední řadě lze za pomocí již zmíněných direktiv sloučit několik velice podobných shaders do jednoho, což může vést k lepší (ne vždy) přehlednosti a manipulovatelsnosti s kódem. Při návrhu shaderů, bylo hojně využito direktivit pro podmíňený překlad, což vedlo k velké redukci počtu souborů, na kterých jsou uloženy kódy shaderů. Díky tomuto přístupu je i jednoduší implementovat změny napříč několika shadery.

2.5 Generování bázových vektorů tangent-space

Souřadnicový systém *tangent-space hraje důležitou* roli při implementaci technik *normal mapping, parallax mapping, steep parallax mapping, ...* K vytvoření přechodové matice potřebujeme bázové vektory *normal (N), tangent (T)* a *bitangent (B)* pro každý vrchol. Na obrázku 2 vlevo můžeme tyto bázové vektory vidět. V další části této podkapitoly si ukážeme způsob, jak vygenerovat vektory T *a* B. Vektory N máme většinou k dispozici. Pokud nejsou vektory N k dispozici, lze je jednoduše vygenerovat (viz. [21]). Nyní si ukážeme způsob, jak vygenerovat vektory T *a* B pro vrcholy trojúhelníku. Dále je dobré zmínit, že ve výsledné aplikaci použijeme pouze vektor T. Vektor B budeme generovat až na grafické kartě pomocí vektorového součinu vektroů N a T. Díky tomu ušetříme paměť grafické karty. Představený algoritmus však generuje vektory T a B, což nemusí být pro určité případy na škodu. Předpokládejme situaci vyobrazenou na obrázku 2 vpravo. V obrázku představují proměnné p_0 - p_2 souřadnice pozic trojúhelníku, (u_0, v_0) - (u_2, v_2) souřadnice v textuře a e_0 a e_1 hrany. Nejprve nadefinujeme pomocné proměnné:

$$a = (u_1, v_1) - (u_0, v_0)$$
$$b = (u_2, v_2) - (u_0, v_0)$$

Pomocí obrázku 2 vpravo (lineární kombinace vektorů), můžeme e_1 a e_2 vyjádřit jako:

$$e_0 = a.x * T + a.y * B$$
$$e_1 = b.x * T + b.y * B$$

Tento systém dvou rovnic o dvou neznámých T a B lze také vyjádřit jako:

$$(e_0.x, e_0.y, e_0.z) = a.x * (T.x, T.y, T.z) + a.y * (B.x, B.y, B.z)$$
$$(e_1.x, e_1.y, e_1.z) = b.x * (T.x, T.y, T.z) + b.y * (B.x, B.y, B.z)$$

Systém rovnic převedeme do maticové reprezentace:

$$\begin{bmatrix} e_0.x & e_0.y & e_0.z \\ e_1.x & e_1.y & e_1.z \end{bmatrix} = \begin{bmatrix} a.x & a.y \\ b.x & b.y \end{bmatrix} \begin{bmatrix} T.x & T.y & T.z \\ B.x & B.y & B.z \end{bmatrix}$$

Rovnici upravíme (vynásobíme obě strany inverzní maticí):

$$\begin{bmatrix} a.x & a.y \\ b.x & b.y \end{bmatrix}^{-1} \begin{bmatrix} e_0.x & e_0.y & e_0.z \\ e_1.x & e_1.y & e_1.z \end{bmatrix} = \begin{bmatrix} T.x & T.y & T.z \\ B.x & B.y & B.z \end{bmatrix}$$

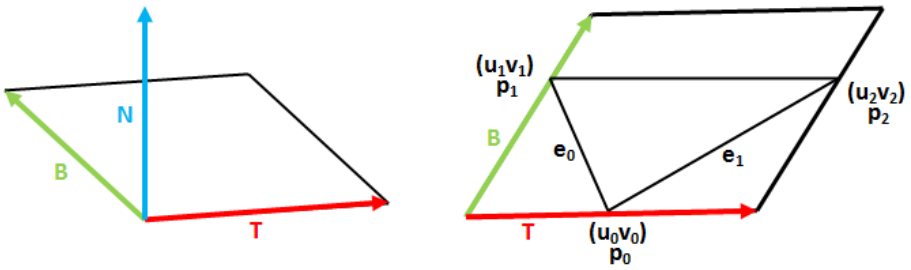
Rovnici upravíme do finální podoby:

$$\begin{bmatrix} T.x & T.y & T.z \\ B.x & B.y & B.z \end{bmatrix} = \frac{1}{a.x\,b.y - a.y\,b.x} \begin{bmatrix} a.x & -a.y \\ -b.x & b.y \end{bmatrix} \begin{bmatrix} e_0.x & e_0.y & e_0.z \\ e_1.x & e_1.y & e_1.z \end{bmatrix}$$

Pomocí této rovnice jsme pak schopni vypočítat T a B vektory trojúhelníku.

Implementace algoritmu

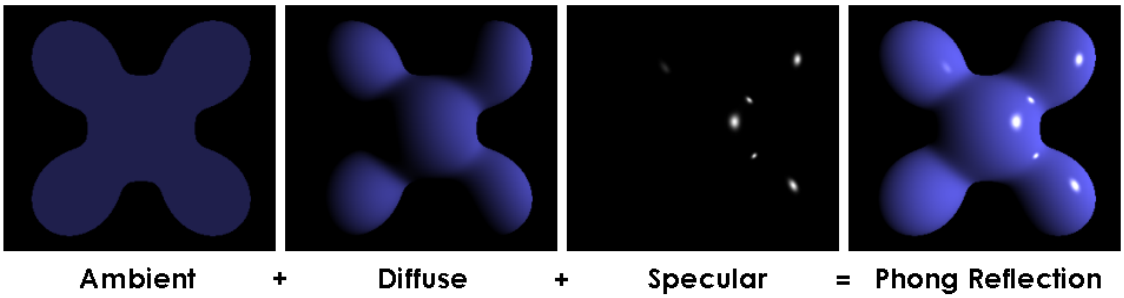
- Algoritmus počítá T a B vektory pro větší celky než jeden trojúhelník, protože výpočet vektorů T a B můžou (nemusí) ovlivnit i sousední vrcholy. Výsledná hodnota vektoru je pak dána sumou i okolních T a B vektorů.
- Data vrcholu jsou rozdělena do více kolekcí.
- Všechny výsledné T a B vektory jsou ortogonalizovány pomocí Gramova-Schmidtovy ortogonalizace (viz. [17]).



Obrázek 2: Zobrazení vektorů TBN.

2.6 Phongův osvětlovací model

Je to empirický, lokální osvětlovací model pro výpočet odraženého světla z povrchu nějakého objektu. I když tento model není fyzikálně založen, je velmi oblíbený v real-time grafických aplikacích. Jeho oblíbenost tkví v tom, že podává uspokojivé vyzuální výsledky vzhledem k příznivé výpočetní ceně. Tento osvětlovací model je postaven na tom, že výsledný odraz světla se skládá ze tří složek a to *okolní složka (ambient reflection)*, *difúzní (diffuse reflection)* a *lesklá (specular reflection)*. Jednotlivé složky můžeme vidět na obrázku 3.



Obrázek 3: Složky Phongova osvětlovacího modelu převzané z [9].

2.6.1 Popis složek osvětlovacího modelu

Okolní složka popisuje odraz okolního světla přícházejícího ze všech směrů. Toto světlo vzniklo mnohonásobnými odrazy od ostatních těles a rozptylem způsobeným molekulami vzduchu. Prakticky v real-time aplikacích nám tato složka zajišťuje, aby objekt nebo části objektů odvrácené od světelných zdrojů nebyly zcela černé.

Difúzní složka popisuje intenzitu části světla, která se od povrchu tělesa rovnoměrně odráží do všech směrů a její použití vytváří trojrozměrný vzhled ve scéně.

Lesklá složka udává intenzitu té části světla, která se od tělesa odráží převážně v jednom směru podle zákona odrazu.

Výsledný odraz světla z povrchu nějakého objektu je dán vztahem:

$$C = C_a + C_d + C_s$$

Jednotlivé části lze dále rozepsat:

$$C_a = I_a k_a$$
$$C_d = I_d k_d (L \cdot N)$$
$$C_s = I_s k_s (V \cdot R)^n$$

Popis složek:

$\mathbf{I_a}$ vyjadřuje intenzitu (barvu) okolního světla včetně barvy materiálu (bývá konstantní v jednoduchých empirických osvětlovacích modelech). I_a bývá stejný jako I_d, který bude popsán níže.

$\mathbf{k_a}$ je to odrazivý koeficient materiálu, pro hodnoty platí: $k_a \in (0, 1)$. Tento koeficient určuje schopnost povrchu odrážet okolní světlo.

$\mathbf{I_d}$ vyjadřuje intenzitu (barvu) difúzního světla včetně barvy materiálu.

$\mathbf{k_d}$ je to odrazivý koeficient materiálu. Pro hodnoty platí: $k_d \in (0, 1)$.

$(\mathbf{L} \cdot \mathbf{N})$ vyjadřuje skalární součin mezi L, což je směr příchodu světla na povrch, a N, což je normálový vektor povrchu v místě dopadu paprsku. Vztah má smysl pouze pro $(L \cdot N) > 0$, protože v opačném případě je povrch od světla odvrácen a difúzní složka je nulová.

$\mathbf{I_s}$ vyjadřuje intenzitu (barvu) lesklého světla včetně barvy materiálu.

$\mathbf{k_s}$ je to odrazivý koeficient materiálu. Opět pro hodnoty platí$k_s \in (0, 1)$

$(\mathbf{V} \cdot \mathbf{R})$ vyjadřuje skalární součin mezi V (vektor směřující od povrchu k pozici pozorovatele) a R (směr odrazu paprsku)

$\mathbf{R}$ $R = 2(L \cdot N)N - L$

$\mathbf{n}$ vyjadřuje ostrost zrcadlového odrazu. Pro hodnoty n platí: $n \in (0, \infty)$

Výsledný odraz pro více světelných zdrojů. Okolní složka je definována globálně.

$$C = I_a k_a + \sum_{i \in Lights} [k_d (L_i \cdot N) I_{i,d} + k_s (V \cdot R_i)^n I_{i,s}]$$

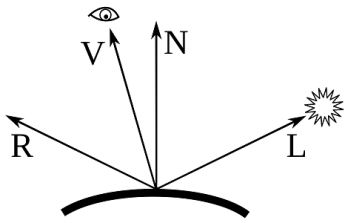
Výsledný odraz pro vícero světelných zdrojů. Okolní složka je definována pro každý světelný zdroj.

$$C = \sum_{i \in Lights} [k_a I_{i,a} + k_d (L_i \cdot N) I_{i,d} + k_s (V \cdot R_i)^n I_{i,s}]$$

Rozložení vektorů L, N, VaR je zobrazeno na obrázku 4.

2.6.2 Typy světelných zdrojů

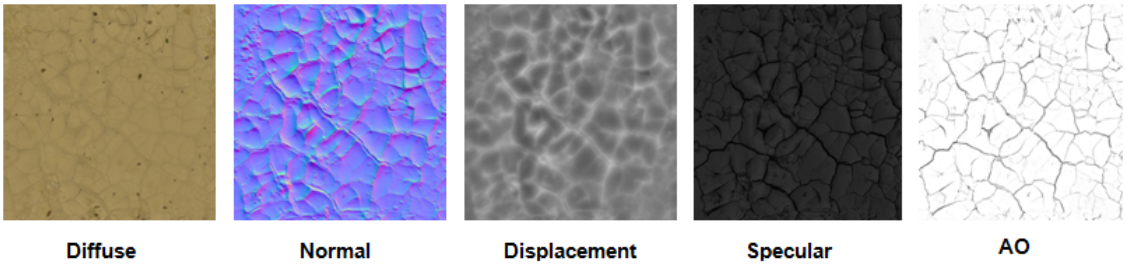
V této části zmíníme světelné zdroje, které jsou použity ve výsledné aplikaci. Těmito zdroji jsou: *rovnoběžný světelný zdroj*, *bodový světelný zdroj* a *reflektor*. Podrobnější popis lze nalézt v [11] a [12].



Obrázek 4: Důležité vektory Phongova osvětlovacího modelu převzaté z [9].

2.7 Typy textur

V této části jsou popsány typy textur vyskytující se v textu a ve výsledné aplikaci. Nejdůležitější textury z hlediska představených technik jsou normálová (normal) a výšková (height). Ostatní typy textur plní jen podružnou úlohu. Jednotlivé typy textur jsou zobrazeny na obrázku 5.



Obrázek 5: Ukázka normálové textury.

Barevná textura (diffuse/albedo texture) uchovává informaci o barvě povrchu. Tato textura bývá někdy označována jako albedo. Nicméně diffuse a albedo textury uchovávají trochu rozdílné hodnoty. Pro zjednodušení v této práci a aplikaci je budeme považovat za totožné. Podrobnější informace lze nalézt v [14].

Normálová textura (normal texture) uchovává informaci o souřadnicích normálových vektorů vůči tangent-space. Tato textura je vhodná pro zobrazení drobných detailů na povrchu. V nejjednoduším tvaru každý texel popisuje souřadnice jednoho normálového vektoru. X-ová souřadnice je uložena v červeném kanálu, y-ová souřadnice je v zeleném kanálu a z-ová souřadnice je v modrém kanálu. Protože normálový vektor míří většinou směrem z povrchu pryč, je dominantní právě z-ová hodnota, což dává normálové textuře charakteristickou namodralou barvu. Některé normálové textury mají však přehozený y-ový a z-ový kanál. V tomto případě je dominantní barva zelená. Další důležitý fakt spočívá v tom, že vektory jsou v textuře uloženy v intervalu $(0, 1)$ pro každý kanál. Nicméně pro každou souřadnici normálového vektoru platí, že je v intervalu $(-1, 1)$. Z tohoto důvodu je nutné transformovat hodnotu přečtenou z textury podle vztahu $\vec{n} = texSample * 2.0 - 1.0$.

Výšková textura (height texture) uchovává informaci o výšce povrchu. Je vhodná pro modelování větších detailů. Černá barva odpovídá nulové výšce, bílá barva pak maximální výšce. Podobným typem textury je hloubková textura. Hloubková textura je v podstatě invertovaná výšková textura. To znamená že nulové výšce odpovídá bílá barva a maximální výšce černá barva.

Lesklá textura (specular texture) uchovává informaci o lesklosti povrchu. Černá barva odpovídá nulové lesklosti, bílá barva pak maximální lesklosti. Vzorky této textury jsou pak použity k modifikaci lesklé složky Phongova osvětlovacího modelu.

Ambient occlusion texture uchovává informaci o zastínění okolím. Černá barva odpovídá maximálnímu zastínění, bílá barva pak žádnému zastínění. Vzorky této textury jsou pak použity k modifikaci okolní složky Phongova osvětlovacího modelu. Více informací lze nalézt v [15].

3 Techniky modelování povrchových detailů

Tato kapitola se zabývá popisem a praktickou implementací jednotlivých technik, které modelují povrchové detaily. Dále pak popisuje výhody a nevýhody těchto technik. Hlavní myšlenka všech popisovaných technik je vizuálně vylepšit povrch modelu pomocí dat textur (viz. Typy textur). U každé z technik jsou vždy popsány nejdůležitější kroky výpočtu v programovatelných fázích. Zbylé, méně důležité kroky lze vyčíst z kódu shaderů. Ještě před samotným popisem technik si zkráceně a zjednodušeně zhrneme poznatky z teoretické části a přidáme pár technických detailů. Toto zhrnutí se bude týkat převážné PZŘ.

Modely jsou tvořeny kolekcí vrcholů, která nese data jako: souřadnice pozice, souřadnice normálových vektorů, souřadnice v textuře. Tyto vrcholy jsou shlukovány pomocí kolekce indexů do polygonů (trojúhelníků). Uvedené kolekce je nutné uložit ve fázi *vertex specification* (viz. PZŘ) do paměti grafické karty. Ve stejné fázi uložíme do paměti grafické karty i přechodové matice *gWVP, gVP, ..* a textury.

Dále bude následovat fáze *vertex shader*. Shader pro tuto fázi PZŘ bude vyhodnocen nad každým vrcholem. V této fázi budeme například tvořit přechodovou matici *ws-to-ts* (viz. *normal mapping*) nebo *paraVec (viz. steep parallax mapping)*. Mezi programovatelnými fázemi budeme předávat data pomocí struktur. Tyto struktury definují buď vstupní nebo výstupní data. Výstup z fáze *vertex shader* bude pokračovat *tessellací*, ve které dojde k vygenerování nových vrcholů, nebo fází *rasterization*.

Ve fázi *rasterization* dochází ke tvorbě fragmentů (pixelů), které pokrývají polygony. Data fragmentů budou vytvořena za pomoci lineární interpolace z dat vrcholů (výstupní data z fází *vertex shader* nebo *tessellation*). Výstup této fáze bude použit na vstup fáze *fragment shader*.

Fáze *fragment shader* bude vyhodnocena nad každým fragmentem. V této fázi dochází například k výpočtu posunu v textuře (*offset*), který je důležitý pro techniky: *steep parallax mapping, relief mapping*. Dále zde dojde k výpočtu *Phongova osvětlovacího modelu*, který je důležitý pro všechny představené techniky.

3.1 Texture mapping

Texture mapping je základní technika, která se snaží vizuálně vylepšit povrch modelu. Jejím principem je mapování vzorků barevné textury na povrch polygonů modelu. Vrcholy modelu musí mít k dispozici souřadnice v textuře. Tyto souřadnice jsou poskytnuty, za pomocí lineární interpolace, jednotlivým fragmentům. Pomocí těchto souřadnic dochází ke čtení vzorku z barevné textury. Následně tento vzorek určí barvu povrchu. Ukázku této techniky vidíme na obrázku 6.

Vertex shader

- Pro každý vrchol transformujeme souřadnice normálového vektoru z *object-space* do *world-space* pomocí matice *gNormal*.
- Texturové souřadnice upravíme pomocí modifikátoru *gTexModif* a pošleme je na výstup této fáze.
- Nakonec transformujeme pozici vrcholu z *object-space* do *clip-space* za použití matice *gWVP*.

Fragment shader

- Pro každý fragment nejprve aplikujeme normalizaci na vektory: směr světla (*lightDirWS*, L), normálový vektor (*normalWS*, N) a směr k pozici pozorovatele (*toEyeWS*, V). Tyto vektory použijeme při výpočtu osvětlení ve *world-space*.
- Načteme vzorek barvy z barevné textury za pomocí texturových souřadnic.
- Provedeme výpočet osvětlení ve *world-space* a výslednou hodnotu uložíme do *gFragColor*. Tato hodnota pak představuje výslednou barvu fragmentu.

Pro

- Jednoduchá implementace.
- Velmy rychlá technika.

Proti

- Malá interakce se světelnými zdroji.

Obrázek 6: Demonstrace techniky texture mapping.

3.2 Normal mapping

Technika *normal mapping* se opět snaží vizuálně vylepšit povrch modelu. Strategie této techniky je postavena na zprostředkování přesnějších dat při výpočtu osvětlení. Přesněji se jedná o normálové vektory. Neformálně řečeno, budeme fingovat detaily prostřednictvím normálové textury, která vznikla z mnohem detailnějšího modelu. Vzorky z normálové textury pak namapujeme na povrch modelu stejným procesem, jakým mapujeme vzorky z barevné textury. Protože data uložená v normálové textuře jsou definována vůči *tangent-space*, je nutné vyrobit

matici přechodu z *tangent-space* do *world-space*. Následně převedeme vzorky z normálové textury do *world-space* pomocí této matice a výpočet osvětlení provedeme ve *world-space*. Nicméně tento způsob vede k násobení matice a vektoru ve fázi *fragment shader* (převod vzorku normálové textury za pomocí matice do *world-space*). To je však celkem drahá záležitost. Zbavit se takové operace ve fázi *fragment shader* je silnou motivací. Místo toho provedeme sérii operací ve fázi *vertex shader*, protože to statisticky stojí méně. Uveďme příklad: model je tvořen 200 vrcholy a zabírá 75000 fragmentů (pixelů) v obraze. *Fragment shader* je vyhodnocen pro každý fragment zvlašť a *vertex shader* je vyhodnocen pro každý vrchol zvlašť. Takže řešení problému je následující. Výpočet osvětlení provedeme v *tangent-space*. Díky tomu se zbavíme převodu vzorku z normálové textury do *world-space*. Dále popsané výpočty proběhnou ve fázi *vertex shader*. K realizaci výpočtů budeme potřebovat převedení vektorů směr světla (*lightDirWS, L*) a směr k pozici pozorovatele (*toEyeWS, V*) do *tangent-space*. K převedení bude nutno vyrobit matici přechodu z *world-space* do *tangent-space*. K výrobě matice potřebujeme, aby vrcholy modelu měly k dispozici souřadnice normálového vektoru a souřadnice tangent vektoru. Souřadnice bitangent vektoru si buď vyrobíme za použití vektorového součinu z normálového a tangent vektoru, nebo je musíme zprostředkovat. Vektory *normal (N), tangent (T) a bitangent (B)* tvoří bázi *tangent-space* a jsou nutné k vytvoření přechodové matice (*ws-to-ts*) mezi *world-space* a *tangent-space*. Pomocí přechodové matice pak transformujeme vektory *L a V* do *tangent-space*. Tyto vektory jsou nutné pro výpočet osvětlení. Přechodová matice vznikne z následujícího vztahu: *ws-to-ts = inverse(transpose(mat3x3(N, T, B)))*. Funkce *inverse* provede inverzi matice a funkce *transpose* provede transpozici matice. Protože *N, T, B* tvoří ortogonální matici (pro ortogonální matici platí, že transponovaná matice je současně maticí inverzní), tak platí *ws-to-ts = transpose(mat3x3(N, T, B))*. Dále můžeme vypustit i transponování matice, protože matici *ws-to-ts* používáme jen při násobení matice a vektoru. Násobení matic v GLSL (platí i pro HLSL - DirectX, Cg - Nvidia) lze vyjádřit pomocí série operací skalární součin. Není nutné přímo tvořit transponovanou matici, ale stačí pouze vhodně provést serii operací skalarní součin bázových vektorů s vektory L a V. Díky tomu ušetříme operace v shaderu. Ukázku techniky můžeme vidět na obrázku 7.

Vertex shader

- Nejprve transformujeme bázové vektory (matice *ws-to-ts*) T a N do *world-space* pomocí *gNormal*. Následně dopočítáme vektor B. Z hlediska optimalizace nebudeme vytvářet matici přímo.
- Pomocí serie skalarního součinu a T,B a N transformujeme vektory L *a* V do *tangent-space*.
- Nakonec transformujeme souřadnice pozice vrcholu z *object-space* do *clip-space* (gl_Position) za pomocí transformační matice $gWVP$.

Obrázek 7: Demonstrace techniky normal mapping.

Fragment shader

- Normalizujeme vektory L a V (interpolace je mohla ovlivnit).
- Načteme souřadnice normálového vektoru pro určitý fragment z textury a transformujeme tyto souřadnice z intervalu $(0, 1)$ do $(-1, 1)$. Protože souřadnice jsou již v *tangent-space* nemusíme je dále upravovat.
- Provedeme výpočet osvětlení v *tangent-space* a výslednou hodnotu uložíme do *gFragColor*. Tato hodnota pak představuje výslednou barvu fragmentu.

Pro

- Dobrý poměr vzhled vs. výpočetní cena.
- Lepší interakce se světelnými zdroji.
- Vhodná technika pro vizualizaci malinkých detailů.

Proti

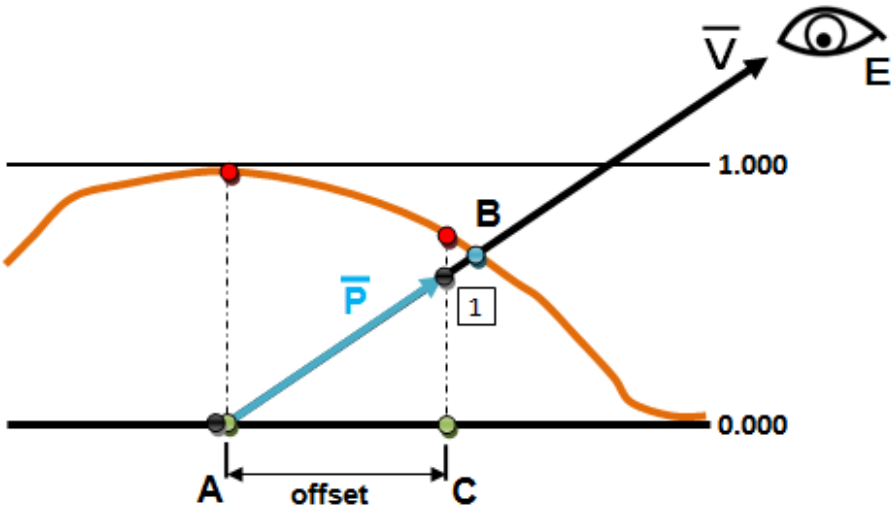
- Nepříliš dobré výsledky při vizualizaci větších detailů (Vše se jeví placatě). Tento problém se snaží řešit další techniky (*parallax map.*, *relief map.*, ...).

3.3 Parallax mapping

Technika parallax mapping se snaží opět vizuálně zkrášlit povrch modelu. Nicméně oproti *normal mapping*, která přidává detaily prostřednictvím výpočtu osvětlení, *parallax mapping* mění geometrii povrchu za pomocí výškové textury. Tato technika využívá data z výškové textury ke stanovení faktoru posunu dále jen *offset* v textuře díky kterému je docíleno aproxymace hloubky (depth) nebo výběžků (bump) na povrchu. Tuto techniku automaticky zkombinujeme s technikou *normal mapping*. Vycházíme z faktu, že potřebujeme přechodovou matici *ws-to-ts* (Data uložená ve výškové/hloubkové textuře jsou definována vůči povrchu (*tangent-space*)). Výsledný povrch díky kombinaci obou technik vypadá lépe. Ukázku techniky *parallax mapping* v kombinaci s *normal mapping* můžeme vidět na obrázku 10.

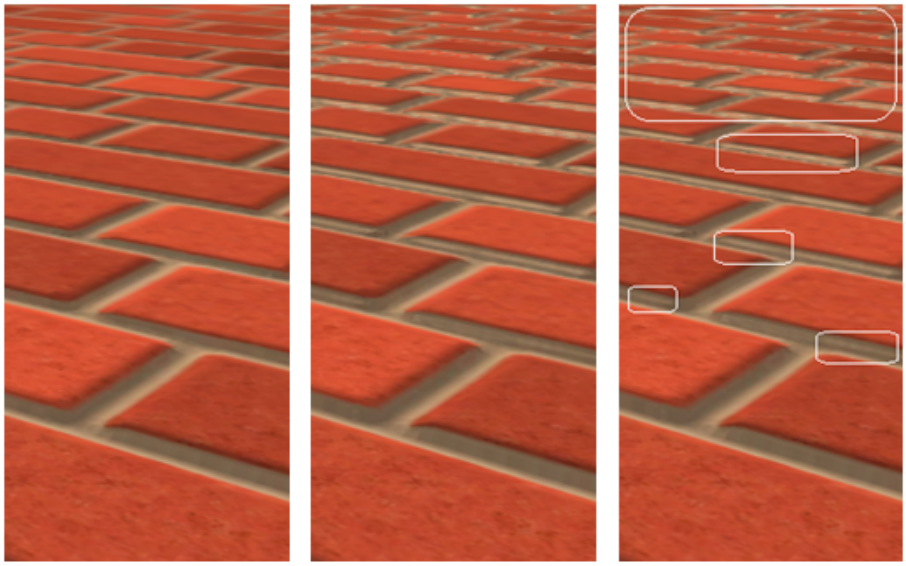
Na obrázku 8 můžeme vidět ukázkovou situaci pro modelování výběžků. Tlustá spodní čára představuje plochu na kterou se díváme jako pozorovatel.

Pozice pozorovatele je určena bodem E (obrázek oka). Oranžová křivka představuje data uložená ve výškové textuře. Pokud nebereme v potaz data z výškové textury, bod který vidíme na ploše je bod A. Tato situace nastává v případě technik *texture mapping* a *normal mapping*. Pokud bereme v potaz data z výškové textury situace se změní. Nyní bychom měli v ideálním případě vidět bod B. Abychom se k této situaci alespoň přiblížili, potřebujeme vypočítat *offset* v souřadnicích textury. *Offset* je vypočítán z parallax vektoru (*paraVec,* P) dále jen *paraVec*. Vektor *paraVec* je v podstatě zkrácený vektor (*toEyeDirTS,* V) dále jen V. Vektor V vyrobíme odečtením pozice pozorovatele *gEyePosWS* od pozice fragmentu *posWS*. Tyto data máme k dispozici. Vektor V je následně nutné převézt z *world-space* do *tangent-space* prostřednictvím přechodové matice *ws-to-ts*. Výsledný *offset* je počítán ve fázi *fragment shader* a je následně přičten k souřadnicím v textuře, které jsou spjaty s bodem A. Tak vypočítáme, v ideálním případě, bod B nebo jeho aproximaci bod C.



Obrázek 8: Výpočet *offsetu* souřadnic v textuře.

V základní podobě je technika *parallax mapping* prakticky skoro nepoužitelná. Je to způsobeno faktem, že popisuje povrch velice hrubě (hrubá aproximace povrchu). Na obrázku 9 uprostřed a vpravo lze vidět, že s přibívající vzdáleností od pozorovatele se objevují poměrně brzy grafické artefakty (chyby v obraze). Tento jev je způsoben faktem, že při výpočtu použijeme pouze jeden vzorek z výškové textury. Pokud chceme dosáhnout přesnějšího výpočtu je nutné použít víc vzorků (viz. *steep parallax mapping, relief mapping* nebo *parallax occlusion mapping*). Dalším řešením (kompromisem), je po určité vzálenosti nezvětšovat *offset* (viz obr. 9 vlevo). K tomu budeme potřebovat zjistit zkrácenou délku (*pLen*) a v závislosti na ní potlačit *offset*.



Obrázek 9: Ukázka grafických artefaktů (chyb v obraze).

Obrázek 10: Demonstrace techniky parallax mapping.

Vertex shader

- Všechny kroky jsou stejné jako pro techniku *normal mapping*.

Fragment shader

- Všechny kroky jsou stejné jako pro techniku *normal mapping*. Jediný rozdíl je v nutnosti přepočtu texturových souřadnic (*texCoordTS*) před krokem 2 (manipulace s normálovou texturou). Viz. funkce *parallaxV0* ve zdrojovém kódu shaderu.

Pro

- Technika vylepšuje *normal mapping*.
- Vhodná technika pro vizualizaci větších detailů.

Proti

- Nevhodné pro výškové / hloubkové textury s drobnými detaily. Kde informace o drobných detailech je uložena ve výškové mapě.
- Problem se simulaci "Vysokých detailů". Poměrně brzo dochází k chybám v obraze. Je nutné modifikovat vzorek z textury.

3.4 Steep parallax mapping

Tato technika rozšiřuje techniku *parallax mapping*. Opět je automaticky zkombinována s technikou *normal mapping*. Hlavní myšlenka spočívá v poskytnutí přesnější aproximace bodu B. Toho docílíme pomocí sofistikovanějšího přístupu, který zpracovává větší počet vzorků z výškové textury. Při popisu budeme vycházet z obrázku 11 a pseudokódu algoritmu 1. Je nutné brát v potaz, že zde představený algoritmus je zjednodušený oproti *GLSL* verzi použité ve výsledném programu. Oproti technice *parallax mapping* jsou patrné určité rozdíly. Oproti *parallax mapping* budeme modelovat hloubku povrchu. Z toho důvodu je vektor *paraVec* negován. Dále budeme veškeré vzorky z výškové textury invertovat (hloubková textura). Na celý problém se dá zjednodušeně dívat jako na lineární vyhledávání ve směru vektoru *paraVec*. Neformální popis algortimu je následující. Celkovou hloubku rozdělíme na určitý počet hladin pomocí *numLayers*. Výšku jedné hladiny bude určovat proměnná *dHeight*. Dále rozdělíme vektor *paraVec* na určité části (viz. *dTex*). Aktuální výška hladiny bude akumulována v *currLHeight* a aktuální souřadnice v textuře budou akumulovány v *currTex*. Pak se budeme pohybovat po směru vektoru *paraVec* po krocích pomocí *dTex*. Hodnotu *dTex* budeme, jak již bylo naznačeno, postupně akumulovat v *currTex*. Pomocí *currTex* budeme postupně číst vzorky z výškové textury. Ty následně invertujeme a budeme je porovnávat s hodnotou *currLHeight*. Algoritmus končí v momentu, kdy vzorek z výškové textury (*heightSample*) je větší jak *currLHeight*. Výsledné souřadnice v textuře včetně *offsetu* jsou v proměnné *currTex*.

Demonstraci techniky *steep parallax mapping* můžeme vidět na obrázku 12. Na obrázku 13 můžeme vidět přiblížení k povrchu. Pokud je pozorovatel příliš blízko povrchu a počet hladin je malý, tak jsou na povrchu patrné nedokonalosti (viz. obr. 13). Tyto nedokonalosti nelze odstranit, ale pouze zmenšit zvětšením počtu hladin. Nicméně větší počet vzorků přináši větší výpočetní cenu. Jak ale uvidíme, tyto nedokonalosti lze snížit použitím technik *relief mapping* nebo *parallax occlusion mapping*.

Další nepřijemnou grafickou nedokonalostí technik *steep parallax mapping, relief mapping a parallax occlusion mapping* je neschopnost správně vypočítat hrany primitiv. To je způsobeno změnou geometrie (výpočet offsetu), ke které dochází ve fázi fragment shader, kde máme pouze omezené možnosti, jak tento

problém řešit. Uvedená skutečnost má za následek grafickou nedokonalost na hranách (viz. obr. 14 vlevo). V aplikaci je použito jednoduché řešení tohoto problému, které ale funguje pouze na modelu plochy (viz. obr. 14 uprostřed). Na obecné úrovni je poměrně složité tento problém řešit. Z tohoto důvodu se tímto problémem v práci dále nezabývám. Jen pro úplnost: techniky *displacement mapping* a *displacement tessellation mapping* tímto neduhem netrpí (viz. obr. 14 vpravo).

Algoritmus 1 Výpočet souřadnic v textuře pomocí techniky steep parallax mapping pro určitý fragment.

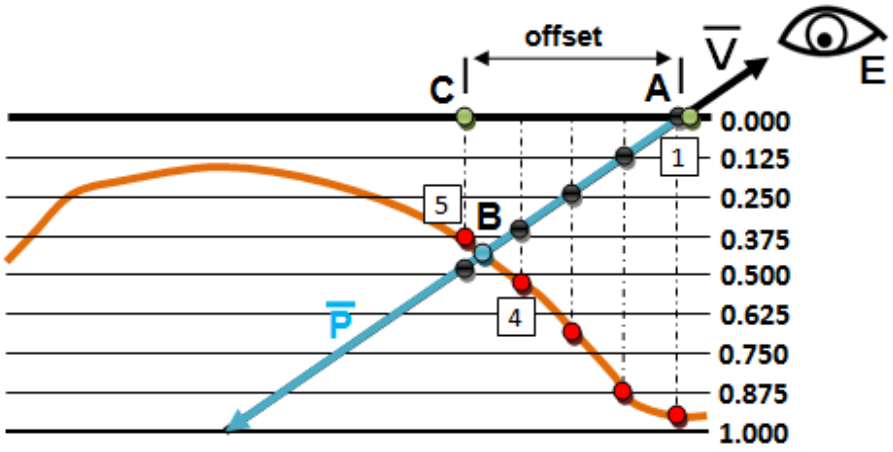
```
define steepPara(paraVec, texCoords, tech, heightTexture)
  numLayers = 8
  dHeight = 1.0 / numLayers
  currLHeight = 0.0
  dTex = paraVec * dHeight
  currTex = texCoords
  heightSample = 1.0 - loadSample(currTex, heightTexture)
  while (heightSample > currLHeight) do
    currLHeight += dHeight
    currTex -= dTex
    heightSample = 1.0 - loadSample(currTex, heightTexture)
  end
  if tech == RELIEF_MAP then
    currTex = relief(currLHeight, dHeight,
      currTex, dTex, heightTexture)
  elseif tech == PARAOCC_MAP then
    currTex = paraOcc(currLHeight, dHeight,
      heightSample, currTex, dTex, heightTexture)
  end
  return currTex
end
```

Vertex shader

- Všechny kroky jsou stejné jako pro techniku *normal mapping*.
- Navíc vypočítáme vektor *paraVec*.

Fragment shader

- Všechny kroky jsou stejné jako pro techniku *normal mapping*. Jediný rozdíl je v nutnosti přepočtu texturových souřadnic (*texCoordTS*) před krokem 2 (manipulace s normálovou texturou). Viz. funkce *parallaxV1* ve zdrojovém kódu shaderu nebo zjednodušená verze popsaná v algoritmu 1.



Obrázek 11: Zpřesnění souřadnic v textuře pomocí techniky steep parallax mapping.

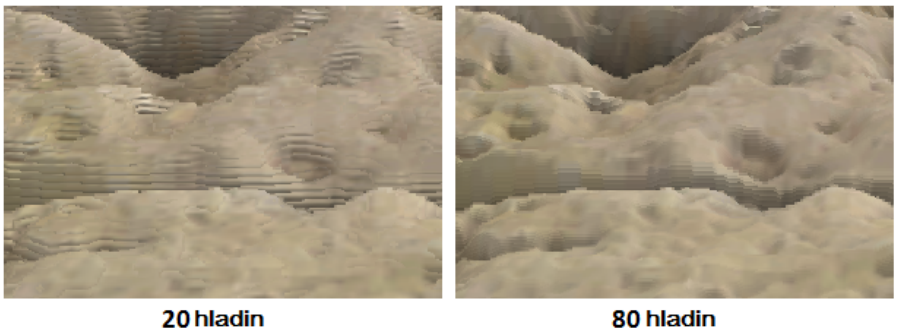
Obrázek 12: Demonstrace techniky steep parallax mapping.

Pro

- Technika vylepšuje *parallax mapping*.
- Vhodná technika pro vizualizaci větších detailů.

Proti

- Oproti technice *normal mapping* a *parallax mapping* je výpočetně náročnější.
- Při malém počtu hladin dochází ke grafickým nedokonalostem, které lze pouze zmenšit.
- Problém s grafickými nedokonalostmi, které vznikají na hranách primitiv.

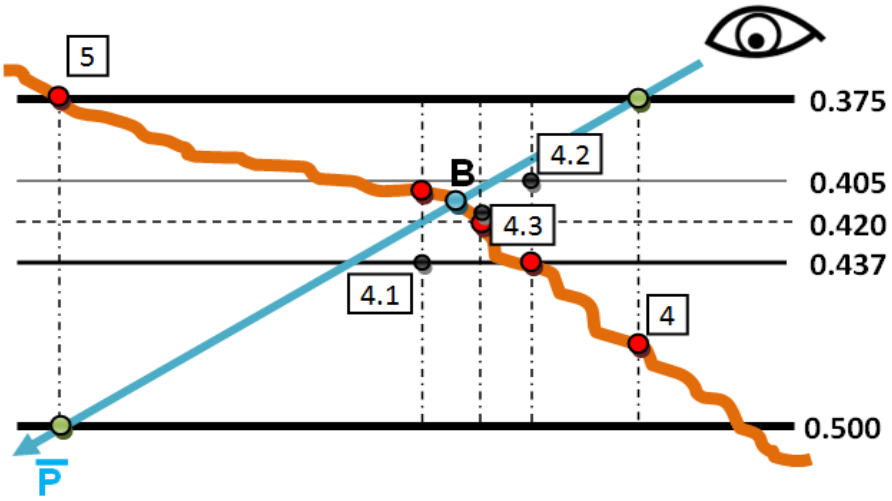


Obrázek 13: Porovnání počtu vzorků techniky steep parallax mapping.

Obrázek 14: Zobrazení nedokonalostí na hraně modelu plochy.

3.5 Relief mapping

Relief mapping se snaží vylepšit techniku *steep parallax mapping*. Opět se snažíme zpřesnit aproximaci bodu B. Opět tuto techniku zkombinujeme s technikou *normal mapping*. Vyjdeme z obrázku 11. Tento obrázek naznačuje, že algoritmus *steep parallax mapping* skončil v pátém kroku. Následující neformální popis *relief mapping* algoritmu vychází z intervalu definovaného mezi čtvrtým a pátým krokem. Při neformálním popisu algoritmu budeme vycházet z obrázku 15 a pseudokódu algoritmu 2. Velice zjednodušeně se dá říci, že *relief mapping* využívá binárního vyhledávání ve směru *paraVec* ke stanovení aproximace bodu B. Algortitmus začíná mezi čtvrtým a pátým krokem uprostřed (viz. obr. 15, krok 4.1). Následně pak dochází k přibližování k bodu B. Proces, kterým se přibližujeme k bodu B, je popsán v pseudokódu. Na obrázku 16 můžeme vidět srovnání *steep parallax mapping* (vlevo), *relief mapping* (uprostřed) a nakonec (vpravo) *parallax occlusion mapping*. Na obrázku 17 můžeme vidět demonstraci techniky *relief mapping*.



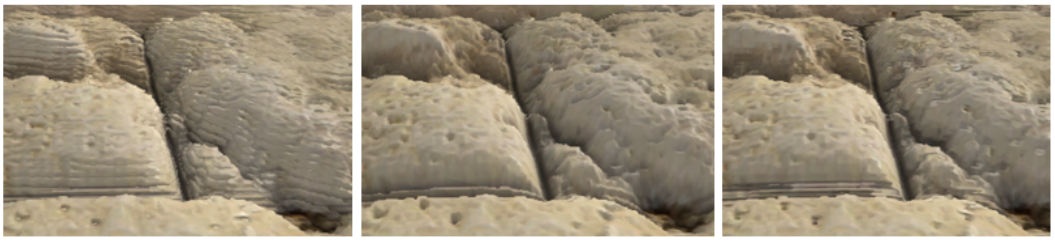
Obrázek 15: Zpřesnění souřadnic v textuře pomocí techniky relief mapping.

Algoritmus 2 Výpočet souřadnic v textuře pomocí techniky relief mapping pro určitý fragment.

```
define relief(_currLHeight, dHeight,
  _currTex, dTex, heightTexture)
  binSteps = 3
  deltaHeight = dHeight * 0.5
  currLHeight = _currLHeight - deltaHeight
  deltaTex = dTex * 0.5
  currTex = _currTex + deltaTex
  for i = 0 to binSteps do
    deltaHeight *= 0.5
    deltaTex *= 0.5
    heightSample = 1.0 - loadSample(currTex, heightTexture)
    if heightSample > currLHeight then
      currLHeight += deltaHeight
      currTex -= deltaTex
    else
      currLHeight -= deltaHeight
      currTex += deltaTex
    end
  end
  return currTex
end
```

Vertex shader

- Všechny kroky jsou stejné jako pro techniku *normal mapping*.
- Navíc vypočítáme vektor *paraVec*.



Obrázek 16: Porovnání technik steep parallax mapping, relief mapping a parallax occlusion mapping.

Obrázek 17: Demonstrace techniky relief mapping.

Fragment shader

- Všechny kroky jsou stejné jako pro techniku *normal mapping*. Jediný rozdíl je v nutnosti přepočtu texturových souřadnic (*texCoordTS*) před krokem 2 (manipulace s normálovou texturou). Viz. funkce *parallaxV1* ve zdrojovém kódu shaderu nebo zjednodušená verze popsaná v algoritmu 2.

Pro

- Technika vylepšuje *steep parallax mapping*.
- Vhodná technika pro vizualizaci větších detailů.

Proti

- Oproti technice *normal mapping, parallax mapping a steep parallax mapping* je výpočetně náročnější.
- Při malém počtu hladin dochází ke grafickým nedokonalostem, které lze pouze zmenšit.
- Problém s grafickými nedokonalostmi, které vznikají na hranách primitiv.
- Technika je dražší, než *parallax occlusion mapping*.

3.6 Parallax occlusion mapping

Parallax occlusion mapping se snaží vylepšit techniku *steep parallax mapping*. Opět se snažíme zpřesnit aproximaci bodu B a techniku zkombinujeme s technikou *normal mapping*. Vyjdeme z obrázku 11. Tento obrázek naznačuje, že algoritmus *steep parallax mapping* skončil v pátém kroku. Technika *parallax occlusion mapping* se opět zaměřuje na interval mezi čtvrtým a pátým krokem (viz. obr. 11). Nicméně oproti *relief mapping* je aproximace bodu B řešena jako problém hledání průsečíku paprsku a přímky. Vyjdeme z obrázku 18. Následující vyjádření parametru t vychází z [18]. Přesnější popis kroků lze opět vyčíst z algoritmu 3. Na obrázku 19 můžeme vidět demonstraci této techniky.

Nejprve nadefinujeme paprsek.

$$r_1(t) = (0,\ pl) + t(1,\ cl - pl)$$

Dále nadefinujeme přímku.

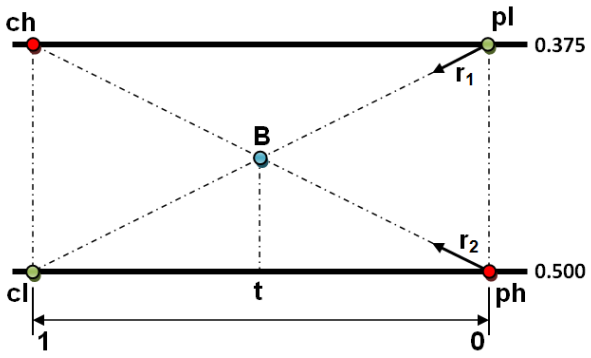
$$r_2(t) = (0,\ ph) + t(1,\ ch - ph)$$

Následně porovnáme paprsek a přímku.

$$r_1(t) = r_2(t)$$
$$(0,\ pl) + t(1,\ cl - pl) = (0,\ ph) + t(1,\ ch - ph)$$

Nakonec vyjádříme hledaný parametr t. Souřadnici .x budeme ignorovat viz. [18].

$$t = \frac{ph - pl}{cl - pl - ch + ph}$$



Obrázek 18: Hledání průsečíku paprsku a přímky.

Obrázek 19: Demonstrace techniky *parallax occlusion mapping*.

Algoritmus 3 Výpočet souřadnic v textuře pomocí techniky *parallax occlusion mapping* pro určitý fragment.

```
define paraOcc(_currLHeight, dHeight, heightSample,
  _currTex, dTex, heightTexture)
  currTex = _currTex
  ph = 1.0 - loadSample(currTex + dTex, heightTexture)
  ch = heightSample
  pl = _currLHeight - dHeight
  cl = _currLHeight
  t = (ph - pl) / (cl - pl - ch + ph)
  currTex += (1.0 - t) * dTex
  return currTex
end
```

Vertex shader

- Všechny kroky jsou stejné jako pro techniku *normal mapping*.
- Navíc vypočítáme vektor *paraVec*.

Fragment shader

- Všechny kroky jsou stejné jako pro techniku *normal mapping*. Jediný rozdíl je v nutnosti přepočtu texturových souřadnic (*texCoordTS*) před krokem 2 (manipulace s normálovou texturou). Viz. funkce *parallaxV1* ve zdrojovém kódu shaderu nebo zjednodušená verze popsaná v algoritmu 3.

Pro

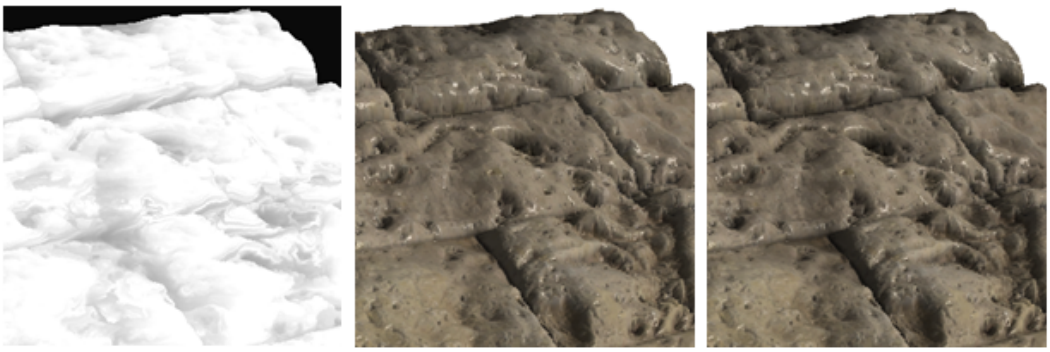
- Technika vylepšuje *steep parallax mapping*.
- Vhodná technika pro vizualizaci větších detailů.
- Technika je méně výpočetně náročnější než technika *relief mapping*.

Proti

- Oproti technice *normal mapping, parallax mapping a steep parallax mapping* je *parallax occlusion mapping* výpočetně náročnější.
- Při malém počtu hladin dochází ke grafickým nedokonalostem, které lze pouze zmenšit.
- Problém s grafickými nedokonalostmi, které vznikají na hranách primitiv.
- Technika poskytuje horší výsledky, než *relief mapping*.

3.7 Self shadows

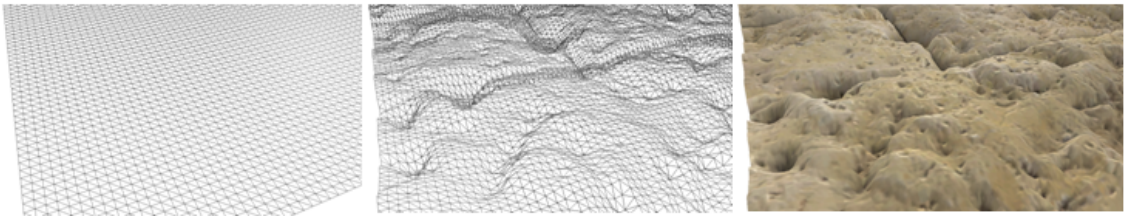
Self shadows je kosmetické rozšíření technik *steep parallax mapping, relief mapping a parallax occlusion mapping*. Jedná se o simulaci stínění na povrchu způsobenou jedním nebo více světelnými zdroji. Ve výsledné aplikaci je toto rozšíření implementováno pouze pro světelný zdroj typu *rovnoběžný světelný zdroj* a pouze pro světelný zdroj s indexem nula. Toto rozšíření bylo nad rámec této práce. Zde bude prezentováno pouze grafické znázornění tohoto rozšíření. (viz. obr. 20). Na obrázku 20 vlevo můžeme vidět samotné stínění. Na obrázku 20 uprostřed je povrch bez stínění a na obrázku 20 vpravo je povrch s aplikovaným stíněním. Stínění je lépe patrné v samotné aplikaci, než na statickém obrázku. Pokud ve výsledné aplikaci měníme směr svitu světelného zdroje s indexem nula, stínění je vizuálně dobře patrné. Podrobnější informace lze získat v [19].



Obrázek 20: Ukázka použití rozšíření *self shadows*.

3.8 Displacement mapping

Tato technika je opět postavena na změně aktuální geometrie povrchu modelu. K dosažení tohoto efektu nejprve načteme vzorek z výškové textury. Tímto vzorkem pak modifikujeme normalizovaný normálový vektor určitého vrcholu, který následně přičteme k souřadnicím pozice tohoto vrcholu. Tímto způsobem docílíme změny pozice ve směru normálového vektoru. Je nutné mít k dispozici dostatečně kvalitní model (viz. obr. 21 vlevo), jenž je tvořen velkým počtem vrcholů, které jsou vhodně rozmístěny po povrchu modelu. Tyto vrcholy pak tvoří polygony, jejichž velikost by měla být v ideálním případě totožná nebo menší, než je velikost fragmentu (pixelu). Na obrázku 21 uprostřed pak vidíme vrcholy po změně souřadnic pozice a na pravé straně pak vidíme výsledný efekt na povrchu. Na obrázku 22 je potom možné vidět demonstraci této techniky ve větším formátu.



Obrázek 21: Ukázka vhodného povrchu modelu.

Obrázek 22: Demonstrace techniky displacement mapping.

Vertex shader

- Nejprve transformujeme bázové vektory (matice *ws-to-ts*) T a N do *world-space* pomocí *gNormal*. Následně dopočítáme vektor B. Z hlediska optimalizace nebudeme vytvářet matici přímo. Vektory T,B a N budou na konci tohoto kroku normalizovány.
- Dále načteme vzorek z výškové textury. Pak použijeme normalizované souřadnice normálového vektoru, které modifikujeme vzorkem z výškové textury a dále uživatelsky nastavitelným modifikátorem (*gHeightScale*).

- Pomocí T,B a N transformujeme vektory: směr světla ($lightDirWS$, L) a směr k pozici pozorovatele ($toEyeWS$, V) do *tangent-space*.
- Nakonec transformujeme souřadnice pozice vrcholu z object-space do clip-space ($gl\_Position$) za pomocí transformační matice $gWVP$.

Fragment shader

- Všechny kroky jsou stejné, jako pro techniku normal mapping.

Pro

- Schopnost zobrazit detailní povrch.
- Možnost zkombinovat s technikou normal mapping.

Proti

- Je nutné mít k dispozici velice kvalitní model (velký počet vrcholů).
- Pokud zobrazujeme velké množství modelů, tato technika je velice náročná na výpočet.
- Modely zabírají velké množství paměti na grafické kartě.

3.9 Displacement tessellation mapping

Tato technika vylepšuje *displacement mapping*. Vylepšení spočívá v řešení obou hlavních problémů techniky displacement mapping, kterými jsou: *paměťová náročnost* (velký počet vrcholů) a *výpočetní náročnost* (velký počet vrcholů = velký počet výpočtů nad vrcholy). Hlavní myšlenka je mít k dispozici méně kvalitní model (model je tvořen malým počtem vrcholů) a v případě potřeby dopočítat další vrcholy modelu na grafické kartě za běhu programu. Nové vrcholy jsou počítány s pomocí barycentrických souřadnic viz. [25].

Ve výsledné aplikaci bylo přidáno rozšíření, které umožňuje automaticky přidávat nové vrcholy v závislosti na pozici pozorovatele a modelu. Pokud se pozorovatel přibližuje k modelu, roste počet nově vygenerovaných vrcholů. Pokud se pozorovatel vzdaluje od modelu, počet nově vygenerovaných vrcholů klesá. Uživatel může tento rys ovlivňovat pomocí proměnných *Tess. dist. min.* a *Tess. dist. max.* (viz. uživatelská příručka). Na obrázku 23 můžeme vidět, jak se počet vrcholů zvětšuje, když se pozorovatel přibližuje k povrchu modelu. Na obrázku 24 můžeme vidět demonstraci techniky *displacement tessellation mapping*.

Vertex shader

- Nejprve transformujeme bázové vektory (matice *ws-to-ts*) T a N do *world-space* pomocí *gNormal*. Následně dopočítáme vektor B. Z hlediska optimalizace nebudeme vytvářet matici přímo. Vektory T,B a N budou na konci tohoto kroku normalizovány.
- Dále vypočítáme *distFactor*. Tento faktor určuje vzdálenost pozorovatele od souřadnice pozice vrcholu. Faktor $distFactor \in (0.0, 1.0)$. Faktor je vypočítán

Obrázek 23: Ukázka několika úrovní tessellace.

Obrázek 24: Demonstraci techniky displacement tessellation mapping.

ve funkci *adaptiveDistFactor*, která je odvozena z [16].

- Pomocí T,B a N transformujeme vektory: směr světla (*lightDirWS*, L) a směr k pozici pozorovatele (*toEyeWS*, V) do *tangent-space*.
- Transformace souřadnic pozice vrcholu z object-space do clip-space je posunuta do fáze tessellation evaluation shader.

Tessellation control shader

- Nejprve vypočítáme faktory *gl_TessLevelInner[0]*, *gl_TessLevelOuter[0]*, *gl_TessLevelOuter[1]* a *gl_TessLevelOuter[2]* (vnitřní a vnější faktory rozdělení záplat). Při výpočtu bude figurovat již zmíněný faktor *distFactor*. Tyto faktory budou ovlivňovat počet nově vygenerovaných vrcholů. I když je tento shader volán pro každý kontrolní bod zvlášť, tyto faktory budou vypočítány pouze jednou za pomocí podmínky if(gl_InvocationID == 0).
- Nakonec předáme všechna ostatní potřebná data kontrolních bodů (vrcholů) do další fáze. Zde je vzorek dat: *posWS*, *texCoordsTS*, *normalWS*, *toEyeDirTS*, ...).

Tessellation primitive generator

- V této fázi se použijí *gl_TessLevelInner[0]*, *gl_TessLevelOuter[0]*,

gl_TessLevelOuter[1] a *gl_TessLevelOuter[2]* (vnitřní a vnější faktory rozdělení záplat) k vygenerování baricentrických souřadnic, které následně použijeme k výpočtu nových dat vrcholů.

Tessellation evaluation shader

- Nejprve vypočítáme pomocí barycentrických souřadnic nová data vrcholů. Zde je vzorek dat: *posWS*, *texCoordsTS*, *normalWS*, *toEyeDirTS*.
- Dále načteme invertovaný vzorek z výškové textury. Pak použijeme normalizované souřadnice normálového vektoru, které modifikujeme vzorkem z výškové textury a dále uživatelsky nastavitelným modifikátorem (*gHeightScale*).
- Nakonec transformujeme souřadnice pozice vrcholu z world-space do clip-space (*gl_Position*) s pomocí transformační matice *gVP*.

Fragment shader

- Všechny kroky jsou stejné, jako pro techniku normal mapping.

Pro

- Možnost zkombinovat s technikou *normal mapping*.
- Vylepšení paměťové a výpočetní náročnosti oproti *displacement mapping*.
- Schopnost přizpůsobit počet vrcholů dle určitých kritérií. Například implementované rozšíření, kde se počet vrcholů mění v závislosti na pozici pozorovatele.

Proti

- Není jednoduché přesně definovat nějaké zásadní proti.

4 Porovnání technik

V této části se z časových důvodů jen okrajově zaměříme na porovnání popsaných technik. Nejprve se zaměříme na jejich silné a slabé stránky. Dále si tyto techniky popíšeme z hlediska časové a paměťové složitosti a nakonec porovnáme z hlediska průměrného trvání vykreslení jednoho obrázku.

4.1 Silné a slabé stránky jednotlivých technik

U každé z představených technik je vždy uveden popis silných a slabých stránek. V některých případech dochází k přímému porovnání s jinou představenou technikou (např: displacement tessellation mapping vs displacement mapping).

4.2 Časová a paměťová složitost

Rozbor paměťové a časové složitosti nám může prozradit zajímavé skutečnosti o popsaných technikách. Při rozboru časové složitosti budeme jako základní operaci uvažovat načtení vzorku z textury (texelFetch). Dále při rozboru paměťové složitosti budeme brát v potaz velikost textury v paměti a velikost dalších potřebných dat pro danou techniku. Při rozboru technik budeme ignorovat lesklou texturu a ambient occlusion texturu nebo další nepodstatné fragmenty kódu. Veškeré úvahy budou jen okrajové a neformální.

Texture mapping
Technika texture mapping potřebuje načíst pro každý fragment pouze jeden vzorek z barevné textury. Z pohledu paměťové složitosti potřebujeme uložit jednu texturu v paměti grafické karty.

Normal mapping
Oproti texture mapping technika normal mapping potřebuje načíst pro každý fragment nejen vzorek z barevné textury, ale také vzorek z normálové textury. Z pohledu paměťové složitosti potřebujeme pro každý vrchol navíc informaci o tangent vektoru. Pokud negenerujeme bitangentní vektory v kódu shaderu, budeme potřebovat ještě navíc zprostředkování bitangentního vektoru. Dále potřebujeme uložit normálovou texturu v paměti grafické karty.

Parallax mapping
Tato technika oproti normal mapping opět přidává navíc načtení vzorku z textury. V tomto případě se jedná navíc o vzorek z výškové textury. Paměťová složitost je stejná jako pro normal mapping. Navíc je potřeba uložit výškovou texturu v paměti grafické karty.

Steep parallax mapping
Tato technika využívá lineárního vyhledávání, takže počet načtených vzorků z textury je lineární. Tento fakt se nicméně týká pouze načítání vzorků z výškové textury. Jinak pro načítání vzorků z barevné a normálové textury je situace stejná, jako u techniky normal mapping. Paměťová složitost je stejná jako pro parallax mapping.

Relief mapping
Tato technika je rozšířením techniky steep parallax mapping, která zužuje vyhledávání na menší interval. Relief mapping pak s pomocí binárního vyhledávání opět zúží již zúžený interval. Druhé zúžení pak probíhá opět v lineárním počtu kroků. Pro načítání vzorků z barevné a normálové textury je situace stejná jako u techniky normal mapping. Paměťová složitost je stejná jako pro parallax mapping.

Parallax occlusion mapping
Tato technika je opět rozšířením techniky steep parallax mapping, která zúžuje vyhledávání na menší interval. V tomto případě ale při výpočtu opětovného zúžení intervalu dojde k načtení pouze jednoho vzorku z výškové textury. To je jeden z důvodů, proč se tato verze v praxi nejvíce využívá. Paměťová složitost je stejná jako pro parallax mapping.

Self shadows
Tato technika je jen kosmetické rozšíření pro techniky steep parallax mapping, relief mapping a parallax occlusion mapping. V podstatě dojde k dalšímu lineárnímu počtu čtení vzorků z výškové mapy. Paměťová složitost je stejná jako pro parallax mapping.

Displacement mapping
V této technice dojde k načtení vzorku z výškové textury pro každý vrchol zvlášť. Zbytek je stejný jako pro normal mapping. Paměťová složitost je stejná jako pro normal mapping. Nicméně tato technika potřebuje dostatečně kvalitní model (velký počet vrcholů), který je uložen v paměti grafické karty.

Displacement tessellation mapping
V této technice dojde k načtení vzorku z výškové textury opět pro každý vrchol zvlášť. Ale dojde k tomu až po vygenerování nových vrcholů. Zbytek je stejný jako pro normal mapping. Paměťovou složitost navíc ovlivňuje počet nově vygenerovaných vrcholů.

4.3 Rychlostní test

Ve výsledné aplikaci je možné implementované techniky porovnat z hlediska počtu vykreslených obrázků v jedné sekundě. Metoda pro měření doby vykreslení obrázku je sice velice jednoduchá a nepočítá s asynchronní komunikací s grafickou kartou, ale i tak má určitou vypovídající hodnotu.

5 Programátorská příručka

Program byl psán v jazyce *C++* ve vývojovém prostředí *Microsoft Visual Studio 2015*. Program funguje pouze v operačním systému *Windows*. Program byl testován pouze ve verzích *Windows 7 a Windows 8.1*. Není doporučeno aplikaci zkoušet pod staršími verzemi OS *Windows*. Pro psaní shaderů byl zvolen jazyk *GLSL* kvůli provázanosti s technologií *OpenGL*. Dále byl využit program *Blender* pro zhotovení a editaci modelů. Pro úpravy a generování textur byl použit program *Adobe Photoshop CS6* a plugin *nDo2*. V neposlední řadě byly také použity programy *Microsoft Paint* a *Microsoft Office Word 2007*.

5.1 Struktura programu

Výsledná aplikace se skládá ze dvou částí. Těmito částmi jsou *testovací subaplikace* a *načtení scény*.

Testovací subaplikace umožňuje zobrazit jednotlivé techniky a porovnat je. Testovací subaplikace umožňuje:

- Zobrazení jednotlivých technik a možnost nastavení celé škály parametrů.
- Výběr *presetů*. Jedná se soubor přednastavených příkladů technik.
- Výběr ze sedmi typů modelů. Těmito modely jsou: *plocha, krychle, koule, dvacetistěn, válec, kruh a torus*. Tyto modely nefungují ideálně se všemi technikami. Nicméně s technikami *texture mapping* a *normal mapping* je vše ok. V aplikaci je umožněna omezená manipulace s modely. Modely lze *rotovat v osách x,y a z*. Dále lze *měnit pozici modelu* a v neposlední řadě i *velikost*.
- Jednoduchá manipulace se světelnými zdroji. Druhy světelných zdrojů: *rovnoběžný světelný zdroj (directional)*, *bodový světelný zdroj (point)* a *reflektor (spot)*.
- Načítání textur uživatelem.

Načtení scény umožňuje načítání scén v *.xml* souborech. Tyto soubory obsahují informaci o kameře, světelných zdrojích a modelech. Modely jsou načítány ze souborů typu *.obj, .mtl* a *.mdl*. Soubory typu *.obj* popisují geometrii modelu a soubory typu *.mtl* pak materiál (povrch) modelů. Popis struktury *.obj, .mtl* lze nalézt v [23]. Soubor typu *.mdl* je inspirovaný formátem modelu představeným v knize [24].

5.2 GLFW

GLFW je multiplatformní knihovna, která poskytuje abstrakci nad tvorbou okna, *OpenGL* contextem (objekt jehož prostřednictvím komunikujeme s GPU v programu) nebo I/O zařízeními (klávesnice, myš nebo joystick) nezávisle na platformě. Ve výsledné aplikaci byla použita verze 3.1.2.

5.3 GLEW

GLEW (The OpenGL Extension Wrangler Library) je multiplatformní knihovna určena k jednoduché práci s OpenGL rozšířeními (GL_ARB_compute_shader, GL_EXT_texture_sRGB, ...). Knihovna nám poskytuje funkcionalitu, díky níž je práce s rozšířeními na specifické platformě jednoduší. Nemusíme tak třeba všechny *OpenGL* funkce definovat ručně a nemusíme též zjišťovat adresy funkcí za pomocí driveru, samozřejmě na každé platformé zvlášť. Díky uvedené knihovně jsme tedy od těchto problémů odstíněni. Více informací o použití rozšíření přímo v programu lze nalézt v citovaném článku [8]

5.4 Další knihovny

- **SOIL** (Simple OpenGL Image Library) je knihovna určená k práci se soubory (obrázky) ve formátu jako např: *.BMP, .PNG, .JPG nebo .DDS*.
- **FreeType** je knihovna, která nám ulehčuje práci s fonty. Ve výsledné aplikaci je použita k načtení dat z formátu *.ttf*.
- **AntTweakBar** je knihovna, která nám umožňuje integrovat jednoduché grafické uživatelské rozhraní do *OpenGL* aplikace.
- **TinyXML** je knihovna, která nám umožňuje pracovat se soubory ve formátu *.xml*. Její funkcionalita je sice omezená, nicméně pro účely této práce je plně dostačující.

5.5 Popis tříd

V této části budou popsány pouze nejdůležitější třídy aplikace.

Application je hlavní třída celé aplikace. Dochází v ní k inicializaci všech potřebných instancí objektů. V této třídé je definován hlavní cyklus aplikace.

SubApplication je rozhraní definující soubor čistě virtuálních metod pro potomky. Potomci ve výsledné aplikaci jsou *TestSubApplication* a *SALoadScene*.

SALoadScene je potomek rozhraní *SubApplication*, ve kterém je implementována logika ohledně zobrazování scén uložených v *.xml* souborech.

TestSubApplication je potomek rozhraní *SubApplication*, ve kterém je implementována logika testovací sub. aplikace.

AppHelper obsahuje celou řadu užitečných metod a maker. Například: konverze základních typu na typ string nebo metody pro práci s *.xml* soubory.

AppSettings je kontejner pro konfigurační data ze souboru *settings.xml*. Uvedená třída pak zprostředkovává ostatním objektům tato data.

Camera je implementace kamery v aplikaci.

ContentManager realizuje správu prostředků, které jsou uloženy v kolekci *contentPacks* typu *ContentPack*. Typy prostředků jsou textury a shaders.

ContentPack obsahuje kolekce textur a shaderů. Dále jsou zde definovány metody pro manipulaci s těmito kolekcemi.

Renderer je rozhraní pro objekt vykreslovač (renderer). Definuje souhrn čistě virtuálních metod, které musí implementovat potomek této třídy.

DefaultRenderer je potomek třídy *Renderer*. Tato třída realizuje veškerou funkcionalitu ohledně vykreslování modelů.

RenderRecord je třída, se kterou pracuje vykreslovač (renderer). Tato třída obsahuje pouze ukazatele na instance typu mesh, material, shader a vykreslovač.

ShaderProgram implementuje metody pro práci s různými shaders. Například: vertex a fragment shaders.

FrameBuffer nám zprostředkovává funkcionalitu s framebufferem.

FrameCounter je třída počítající počet vykreslených obrázků za určitý časový interval. V aplikaci byl vybrán interval jedna sekunda (1s).

LightManager je třída poskytující funkcionalitu ohledně práce se světelnými zdroji.

Material obsahuje data popisující povrch modelů. Například: Identifikátory textur, barva povrchu, výška povrchu nebo lesklost povrchu.

GeometryFactory je třída generující geometrické objekty jako: plocha, krychle nebo rastrovaná plocha (vhodná pro displacement mapping).

Mesh popisuje data geometrie. Jedná se o základní stavební prvek modelu. Jsou v něm uloženy identifikátory na kolekci vrcholů nebo na kolekci polygonů.

Model popisuje modely a metody pro jejich manipulaci.

Scene popisuje určitou scénu a definuje metody pro její manipulaci jako: Najdi určitý mesh nebo sestav scénu dle souboru typu *.xml*.

ModelLoader implementuje metody pro načítání modelu. Podporuje formát typu *.obj*, ale jen v ořezané míře. Dále implementuje metody pro načtení *.mdl* formátu.

PostProcessManager implementuje funkcionalitu pro práci z full screen quad (Obdelník pokrývající plochu zobrazovacího zařízení). Pomocí tohoto obdelníku a dalších komponent můžeme realizovat vykreslování do již naplněného framebufferu a tak realizovat post process efekty jako: Antialiasing ($FXAA$) nebo převod obrázku na šedotónový.

SpriteManager implementuje metody pro práci s obrázky (sprites), které se vykreslují do uživatelského rozhraní aplikace. Tyto obrázky jsou vykresleny ne-

závazně na *AntTweakBar* knihovně.

TextManager implementuje metody pro práci s textem, které se vykreslují do uživatelského rozhraní aplikace. Tento text je vykreslen nezávazně na *AntTweakBar* knihovně.

TextureManager implementuje práci s texturami. Tento manager je schopen fungovat nezávazně na *contentManager*. Dále zprostředkovává pro *contentManager* načítání textur.

Timer je implementace časovače.

TimersManager je kolekce časovaču typu *Timer*. Dále implementuje metody pro práci s časovači.

AABoundingBox implementuje axis-aligned bounding box (AABB). Tato třída je ořezaná a implementuje pouze metodu, která zjišťuje, jestli bod leží v AABB.

MathHelper implementuje pomocné matematické metody jako: Výpočet normal vektorů pro daný mesh, výpočet TBN vektorů pro daný mesh nebo metody pro přepočet stupňů na radiány.

Vector2f, Vector3f, Vector4f, Vector2i, Vector3i, Matrix3f, Matrix4f implementují matematické třídy pro práci s vektory a maticemi.

ATBarManager implementuje metody pro práci s *AntTweakBar* knihovnou.

LogManager implementuje metody pro logování aplikace.

PresetManager je třída pro spravování Presetů. Pod pojmem *preset* si můžeme představit určitý přednastavený stav testovací subaplikace.

6 Uživatelská příručka

V této části budou popsány veškeré informace týkající se ovládání aplikace *TM-Techs* (Texture mapping techniques). Obsáhlejší uživatelská příručka je implementována přímo v aplikaci. Tuto

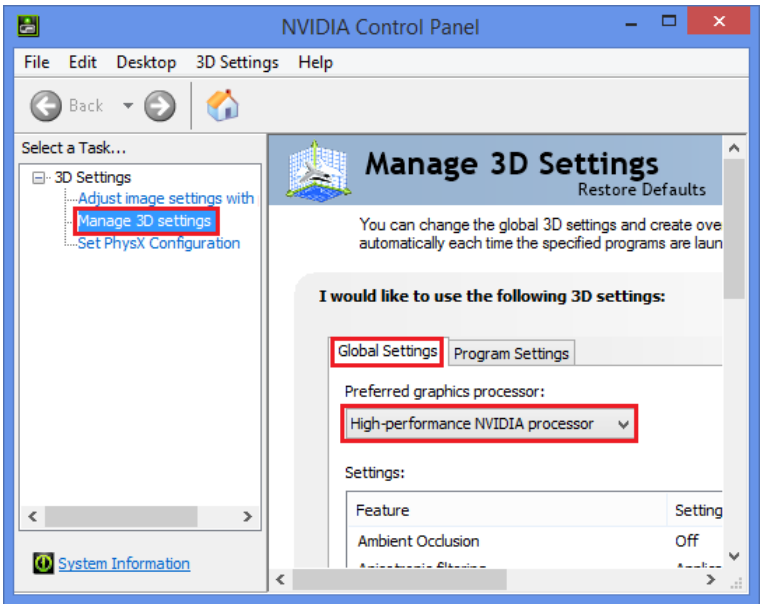
Programové prerekvizity

Pro správnou funkci aplikace je nutné mít nainstalované *distribuovatelné balíčky jazyka Visual C++ pro Visual Studio 2015 (Microsoft Visual C++ Redistributable 2015).* Tyto balíčky je možné stahnout například z [20].

Hardware prerekvizity

K úspěšnému spuštění aplikace je nutné vlastnit počítáč s dostatečně výkonnou grafickou kartou, která podporuje *OpenGL 4.3*. V dnešní době (17. 6. 2017) by neměl být velký problém vlastnit takový počítač. Dokonce i integrované grafické karty nemají většinou problém s podporou aplikace. Nicméně u starších strojů (+- 6 let), které neoplývají dedikovanou grafickou kartou, se mohou vyskytnout problémy.

Dalším zajímavým problémem je občasná neschopnost aplikace nalézt a použít dedikovanou grafickou kartu (knihovna *GLFW*). Tento problém se vyskytl na starších typech strojů, které obsahovaly jak integrovanou, tak i dedikovanou grafickou kartu. Aplikace byla schopna nalézt pouze integrovanou grafickou kartu, která nebyla vhodná pro spuštění aplikace. Tomuto chování se dá zabránit použitím vhodného nástroje. Následující postup řešení problému byl vyzkoušen pouze na grafické kartě od společnosti *NVIDIA* (NVIDIA GeForce GT 630M) za pomocí programu *NVIDIA Control Panel* (verze 8.1.940.0). Předpokládá se, že podobný postup lze aplikovat i na nástroje a grafické karty jiných výrobců. Při popisu budeme vycházet z obrázku 25. V programu nejprve v levé části okna vybereme položku *Manage 3D settings*. Dále v právé části okna vybereme položku *Global Settings*. Nakonec při výběru preferované grafické karty (Preferred graphics processor) vybereme položku *High-performance NVIDIA processor*.



Obrázek 25: Vynucený výběr grafické karty v programu NVIDIA Control Panel.

Ovládání

Pohyb v aplikaci je zajištěn pomocí implementace kamery z pohledu první osoby (*first person camera, fps camera*). Pomocí kláves **W, S, A** a **D** nebo **up, down, left** a **right** se pohybujeme **dopředu, dozadu, doleva** a **doprava**. **Ovládání pohledu** je zajištěno **stisknutím a držením levého tlačítka myši**. **Restart kamery do výchozího nastavení** lze provést v hlavním menu (*MainBar*) v sekci *Camera* tlačítko *Restart*.

Test sub. scene

Test sub. scene je základní mód aplikace, který je aktivovaný při spuštění aplikace. Uživatel se může do tohoto módu kdykoli přepnout pomocí položky *MainPanel.Choose sub. application* kde vybere hodnotu *Test*. V tomto módu může uživatel:

- **Nastavit materiál** pomocí položek *Material*.
- **Nastavit techniku** pomocí *Material.Technique*.
- **Nastavit balíček textur** pomocí *Material.Texture set*.
- **Načíst vlastní textury** v sekci *Material*.
- **Nastavit model** pomocí položek *Model*.
- **Nastavit typ modelu** pomocí *Model.Shape*.

Zobrazení presetů

Presety (přednastavené stavy aplikace) je možné zobrazit v módu *Test* pomocí položky *MainPanel.Choose preset*. Presety byly vytvořeny, protože není úplně jednoduché nastavit veškeré parametry aplikace. Uživatel má tak možnost zobrazit různé příklady rychle a pohodlně. Další výhodou presetů je možnost, že při nesprávném nastavení parametrů aplikace (př.: kamera směřuje směrem pryč od modelu nebo byly nastaveny parametry, při kterých vzezření modelu nevypadá dobře) je uživatel rychle schopen uvést aplikaci do rozumného stavu. Preset můžeme také využít jako šablonu pro unikátní nastavení aplikace.

Načtení vlastních textur

Uživatel může načíst vlastní textury v módu *Test* (*MainPanel.Choose sub. application*). Textury je pak možné načíst v sekci *Material (Material.Load diffuse, Material.Load normal, Material.Load specular, Material.Load height, Material.Load AO)*. Uživatel může načítat textury pouze typu *.DDS, .jpg a .bmp*. Je doporučeno načítat pouze dodané textury, které se nalézají v složce *content.Textures* nebo *content.Models*. Samozřejmě by neměl být problém načíst i textury z jiných zdrojů za předpokladu, že budou dodržena určitá pravidla. Je nutné aby textura byla tvořena třemi (př.: *RGB*) nebo čtyrmi komponentami (př.: *RGBA*)

a aby každá komponenta měla barevnou hloubku pouze 8 bit. I tak se v některých případech stává, že některé textury, které tato kritéria splňují, nefungují. V takovém případě je doporučeno tuto texturu exportovat v nějakém nástroji (př.: *Microsoft Paint*) do jiného či stejného (aplikací podporovaného) formátu. Textury nesplňující tyto požadavky (př.: textury tvořeny pouze jedním kanálem nebo textury s barevnou hloubkou 16 bit. na kanál) mohou vést k pádu aplikace.

Load scene

Tento mód umožňuje zobrazit složitější scény, které nalezneme v adresáři *content.Scenes*. Přepnout se do tohoto módu můžeme prostřednictvím *MainPanel.Choose sub. application*, kde vybereme hodnotu *LoadScene*. Po přepnutí se do módu *LoadScene* se v grafickém rozhraní zobrazí sekce *SceneMan*. Zde pak pomocí tlačítka *SceneMan.LoadScene* můžeme načíst scény z již zmiňovaného adresáře *content.Scenes*. Nejpovedenější scéna je *SponzaScene.xml*

Popis položek menu

- **MainPanel.Choose sub. application** slouží k výběru sub. aplikace. Na výběr máme z možností *Test a LoadScene*.
- **MainPanel.Choose preset** slouží k výběru přednastaveného stavu testovací aplikace (*preset*). Tato položka je viditelná pouze tehdy, pokud je vybrána testovací sub. aplikace (*Test*).
- **MainPanel.Help** zobrazí soubor *userGuideV0.chm*. Tento soubor slouží jako uživatelská příručka k aplikaci.
- **LightManager.Enable ambient term** aktivuje okolní složku osvětlovacího modelu pro všechny aktivní světelné zdroje.
- **LightManager.Enable diffuse term** aktivuje difúzní složku osvětlovacího modelu pro všechny aktivní světelné zdroje.
- **LightManager.Enable specular term** aktivuje lesklou složku osvětlovacího modelu pro všechny aktivní světelné zdroje.
- **LightManager.Light index** slouží pro výběr určitého světelného zdroje.
- **LightManager.Enable** aktivuje/deaktivuje určitý světelný zdroj.
- **LightManager.Type** slouží k výběru typu (bodový světelný zdroj, reflektor, ...) určitého světelného zdroje podle hodnoty *LightManager.Light index*.
- **LightManager.Color** určuje barvu světelného zdroje. Je možné si vybrat z barevného modelu *RGB* nebo *HSV*. Světelný zdroj je určen podle hodnoty *LightManager.Light index*.
- **LightManager.Ambient** je modifikátor okolní složky světelného zdroje. Světelný zdroj je určen podle hodnoty *LightManager.Light index*.
- **LightManager.Diffuse** je modifikátor difúzní složky světelného zdroje. Světelný zdroj je určen podle hodnoty *LightManager.Light index*.

- **LightManager.Specular** je modifikátor lesklé složky světelného zdroje. Světelný zdroj je určen podle hodnoty *LightManager.Light index*.
- **LightManager.Direction** určuje směr svitu světla. Položka se zobrazí pouze pro světelný zdroj typu *rovnoběžný světelný zdroj (Directional) a reflektor (Spot)*. Světelný zdroj je určen podle hodnoty *LightManager.Light index*.
- **LightManager.Constant** určuje konstantní faktor při výpočtu útlumu (attenuation) světelného zdroje. Položka se zobrazí pouze pro světelný zdroj typu *bodový světelný zdroj (Point)* viz. [12]. Světelný zdroj je určen podle hodnoty *LightManager.Light index*.
- **LightManager.Linear** určuje lineární faktor při výpočtu útlumu (attenuation) světelného zdroje. Položka se zobrazí pouze pro světelný zdroj typu *bodový světelný zdroj (Point)* viz. [12]. Světelný zdroj je určen podle hodnoty *LightManager.Light index*.
- **LightManager.Quadratic** určuje kvadratický faktor při výpočtu útlumu (attenuation) světelného zdroje. Položka se zobrazí pouze pro světelný zdroj typu *bodový světelný zdroj (Point)* viz. [12]. Světelný zdroj je určen podle hodnoty *LightManager.Light index*.
- **LightManager.Inner cut off** slouží k nastavení poloměru kuželu, který definuje vyzařovací charakteristiku světelného zdroje. Hodnota musí být menší než *LightManager.Outer cut off*. Položka se zobrazí pouze pro světelný zdroj typu *reflektor (Spot)* viz. [12]. Světelný zdroj je určen podle hodnoty *LightManager.Light index*.
- **LightManager.Outer cut off** slouží k nastavení poloměru kuželu, který definuje vyzařovací charakteristiku světelného zdroje. Hodnota musí být větší než *LightManager.Inner cut off*. Položka se zobrazí pouze pro světelný zdroj typu *reflektor (Spot)* viz. [12]. Světelný zdroj je určen podle hodnoty *LightManager.Light index*.
- **Settings.Show wireframe** zapne/vypne vyplnění povrchu polygonů. V literatuře se pro tento mód vyskytuje také termín drátový model.
- **Settings.Background color** nastaví barvu pozadí.
- **Camera** panel slouží k zobrazení informací o kameře (pozorovateli). Panel slouží převážne pro čtení. Pomocí *Camera.Position* můžeme třeba nastavit pozice světelných zdrojů.
- **Camera.Restart** tlačítko slouží k nastavení výchozích parametrů kamery.
- **Model** panel slouží k modifikaci parametrů modelu. Mužeme zde vybrat typ modelu (*Model.Shape*). Dále můžeme měnit pozici, rotaci a měřítko modelu (Model.Position, Model.Rotation a Model.Scale).
- **Material.Texture set** umožňuje výběr z devíti druhů balíčků textur. Hodnota *UserDefined* je pak vyhrazena pro balíček, který může načíst uživatel prostřednictvím postupu popsaného dále.
- **Material.Technique** slouží k výběru z osmi druhů výše popsaných technik.

- **Material.Shininess** slouží k nastavení modifikátoru lesklosti povrchu modelu.
- **Material.Tex coords. modif.** umožňuje modifikovat texturové souřadnice modelu. Vizuálně tím dosáhneme opakování textury na povrchu nebo zobrazení jen určité části textury.
- **Material.Enable diff. tex.** zobrazí/skryje barevnou texturu na povrchu modelu.
- **Material.Enable spec. tex.** zobrazí/skryje lesklou texturu na povrchu modelu.
- **Material.Enable ao. tex.** zobrazí/skryje texturu, která zobrazuje zastínění okolím (*ambient occlusion*) na povrchu modelu.
- **Material.AO. modif.** modifikátor ovlivňující hodnotu zastínění okolím (*ambient occluison*).
- **Material.Height scale** je modifikátor ovlivňující výšku/hloubku z výškové/hloubkové textury.
- **Material.Para. efect clamp** je modifikátor, který určuje vzdálenost působení efektu techniky *parallax mapping*.
- **Material.Height sample bias** je modifikátor, který zvyšuje/snižuje celkovou výšku/hloubku z výškové/hloubkové textury. Tento modifikátor slouží pro zlepšení výsledků techniky *parallax mapping*. Celková výška/hloubka je ovlivněna až po užití modifikátoru *Material.Height scale*.
- **Material.Min. linear steps** určuje minimální počet hladin. Celkový počet hladin je pak vypočítán z této hodnoty a z hodnoty proměnné *Material.Max. linear steps*. Celkový počet hladin se zvyšuje s rostoucí vzdáleností od pozorovatele. Tato proměnná ovlivňuje techniky *steep parallax mapping, relief mapping a parallax occlusion mapping*.
- **Material.Max. linear steps** určuje maximální počet hladin. Celkový počet hladin je pak vypočítán z této hodnoty a z hodnoty proměnné *Material.Min. linear steps*. Celkový počet hladin se zvyšuje s rostoucí vzdáleností od pozorovatele. Tato proměnná ovlivňuje techniky *steep parallax mapping, relief mapping a parallax occlusion mapping*.
- **Material.Enable tex. edge discard** ořízne hrany modelu. Správné chování je zajištěno pouze pro model plochy (*plane*). Tato proměnná ovlivňuje techniky *steep parallax mapping, relief mapping a parallax occlusion mapping*.
- **Material.Enable parallax self shadow** zapne/vypne simulaci stíňení na povrchu (self shadows) modelu. Tato proměnná ovlivňuje techniky *steep parallax mapping, relief mapping a parallax occlusion mapping*.
- **Material.Enable show only shadow factor** zapne/vypne pouze zobrazení stínění na povrchu (self shadows) modelu. Tato proměnná ovlivňuje techniky *steep parallax mapping, relief mapping a parallax occlusion mapping*.

- **Material.Parallax self shadow mod.** modifikuje stínění na povrchu (self shadows) modelu.Tato proměnná ovlivňuje techniky *steep parallax mapping, relief mapping a parallax occlusion mapping.*
- **Material.Parallax self shadow mod.** modifikuje stínění na povrchu (self shadows) modelu.Tato proměnná ovlivňuje techniky *steep parallax mapping, relief mapping a parallax occlusion mapping.*
- **Material.Binary steps** určuje počet dodatečných hladin pro techniku *relief mapping.*
- **Material.Tess. dist. min.** je faktor ovlivňující úroveň tessellace na povrchu modelu v závislosti na vzdálenosti pozorovatele od povrchu modelu. Tato proměnná ovlivňuje techniku *displacement tessellation mapping.*
- **Material.Tess. dist. max.** je faktor ovlivňující úroveň tessellace na povrchu modelu v závislosti na vzdálenosti pozorovatele od povrchu modelu. Tato proměnná ovlivňuje techniku *displacement tessellation mapping.*
- **Material.Tess factor** je faktor ovlivňující úroveň tessellace na povrchu modelu. Spolu s *Material.Tess. dist. min. a Material.Tess. dist. max* stanoví výslednou úroveň tessellace na povrchu modelu. Tato proměnná ovlivňuje techniku *displacement tessellation mapping.*
- **Material.Pos. modif.** vychyluje souřadnice pozice ve směru souřadnic normálového vektoru. Tato proměnná ovlivňuje techniku *displacement tessellation mapping.*
- **Material.Load diffuse** slouží k načtení uživatelem vybrané barevné textury.
- **Material.Load normal** slouží k načtení uživatelem vybrané normálové textury.
- **Material.Load specular** slouží k načtení uživatelem vybrané lesklé textury.
- **Material.Load height** slouží k načtení uživatelem vybrané výškové textury.
- **Material.Load AO** slouží k načtení uživatelem vybrané ambient occlusion (stínění okolím) textury.
- **SceneMan.Load Scene** slouží k načtení uživatelem vybrané scény.

Závěr

Cílem práce bylo zpracovat a popsat problematiku technik, které pomocí textur modelují detaily na povrchu modelu. Výstupem práce je text bakalářské práce a program *TMTechs*. V textu bakalářské práce se nejprve zabývám teorií (př: souřadnicové systémy, OpenGL, GLSL, Phongův osvětlovací model, ...). Dále popisuji jednotlivé mapovací techniky (př: normal mapping, parallax mapping, displacement mapping, ...). Potom navazuje kapitola, která se pouze povrchně zabývá porovnáváním jednotlivých technik. Tato kapitola je méně obsáhlá, než bylo původně zamýšleno. Je to způsobeno faktem, že vývoj aplikace byl pro mě velice časově náročný. Dále následuje programátorská příručka, kde popisuji strukturu programu nebo použité knihovny třetí strany (př: GLFW nebo GLEW). Nakonec text obsahuje uživatelskou příručku. Dalším výstupem je program *TMTechs*. Program umožňuje primárně porovnávat jednotlivé techniky modelování detailů. Uživateli je umožněno v omezené míře: provádění editace osvětlení scény, výběr modelů (př: plocha, krychle, ...), výběr balíčků textur ovlivňujících povrch modelů, načítání libovolných textur a načítání scén, kde se vyskytují složitější modely.

Conclusions

The aim of the thesis was to elaborate and describe the problems of techniques, which using textures model the details on the surface of the model. The output of the thesis is the text of the bachelor thesis and the program *TMTechs*. In the bachelor thesis I first deal with theory (for example, coordinate systems, OpenGL, GLSL, Phong lighting model, ...). I also describe individual mapping techniques (eg: normal mapping, parallax mapping, displacement mapping, ...). Then follows a chapter, which deals only with the comparison of individual techniques. This chapter is less comprehensive than originally intended. This is due to the fact that the development of the application was very time consuming for me. The following is followed by a programmer's guide describing the program structure or the third-party library (for example: GLFW or GLEW). Finally, the text contains a user guide. Another output is the *TMTechs* program. The program allows you to compare each modeling technique. The user is allowed to: performing scene lighting editing, selecting models (for example, area, cube, ...), selecting texture packets affecting the surface of models, loading any texture, and loading scenes where more complex models occur.

A Obsah přiloženého DVD

Popis obsahu adresářové struktury přiloženého dvd.

bin/
: Obsahuje adresář *TMTechs*, ve kterém je spustitelný program *TMTechs.exe*.

doc/
: Obsahuje text bakalářské práce a všechny potřebné soubory k jejímu vytvoření.

src/
: Obsahuje kompletní zdrojové texty programu *TMTechs*.

readme.txt
: Instrukce pro spuštění programu.

B Soubor obrázků

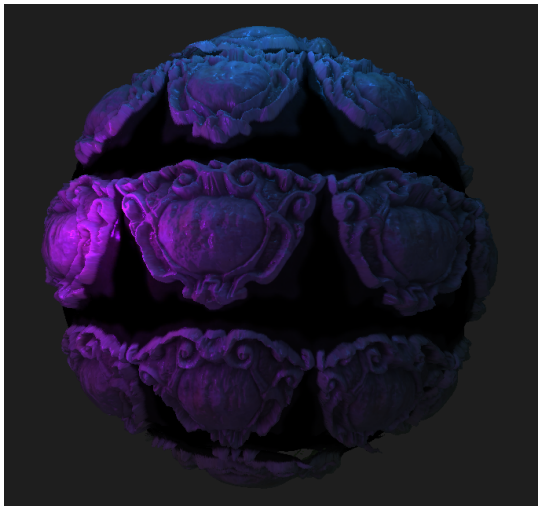
V této příloze je možné shlédnout obrazové výstupy aplikace *TMTechs*. Část obrázků jsou *presety* a scény: *SponzaScene.xml, Cerberus.xml a GeometryPack.xml*. Nakonec pak uživatelem načtený set *SoilClay (TMTechs\content\Textures\...)*.

Obrázek 26: Preset: Texture map. ...

Obrázek 27: Preset: Normal map. ...

Obrázek 28: Preset: Relief map. ...

Obrázek 29: Preset: Disp. tess. ...

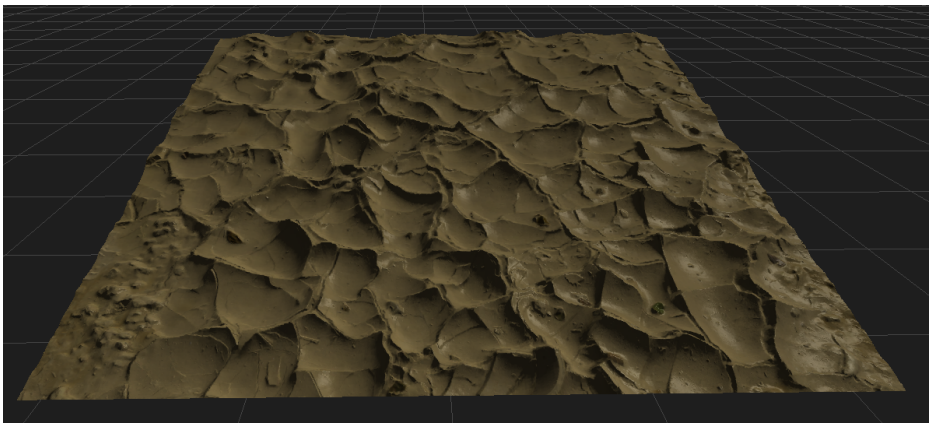


Obrázek 30: Preset: Tess. Sphere

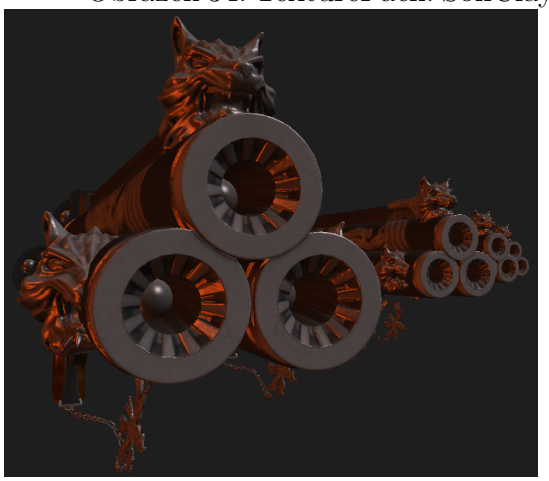
Obrázek 31: Scene: SponzaScene.xml

Obrázek 32: Scene: SponzaScene.xml

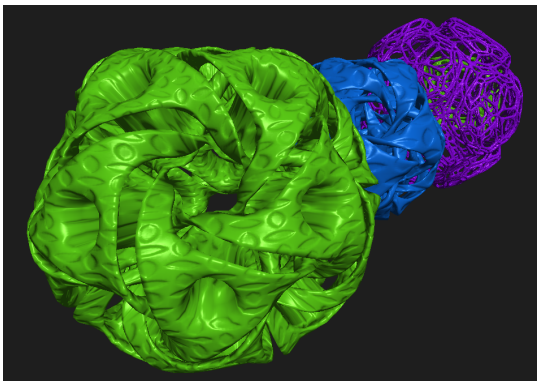
Obrázek 33: Scene: SponzaScene.xml



Obrázek 34: TexturePack: SoilClay, Quixel MEGASCANS, Free pack



Obrázek 35: Scene: Cerberus.xml



Obrázek 36: Scene: GeometryPack.xml

Literatura

[1] Wikipedia. Popis knihovny OpenGL. [online]. [cit. 8-6-2017]. Dostupné z: https://en.wikipedia.org/wiki/OpenGL

[2] Wikipedia. Definice polygonu. [online]. [cit. 8-6-2017]. Dostupné z: https://en.wikipedia.org/wiki/Polygon

[3] Wikipedia. Popis zobrazovacího řetězce knihovny openGL. [online]. [cit. 8-6-2017]. Dostupné z: https://www.opengl.org/wiki/Rendering_Pipeline_Overview

[4] Graham Sellers, Richard S. Wright, Nicholas Haemel. OpenGL SuperBible: Comprehensive Tutorial and Reference, Sixth Edition. Addison-Wesley, 2014.

[5] Wikipedia. Popis jazyka GLSL. [online]. [cit. 8-6-2017]. Dostupné z: https://cs.wikipedia.org/wiki/OpenGL_Shading_Language

[6] Wikipedia. GLSL vektorové a maticové operace. [online]. [cit. 8-6-2017]. Dostupné z: https://en.wikibooks.org/wiki/GLSL_Programming/Vector_and_Matrix_Operations

[7] Wikipedia. GLSL datové typy. [online]. [cit. 8-6-2017]. Dostupné z: https://www.opengl.org/wiki/Data_Type_(GLSL)#Basic_types

[8] OpenGL. Používání OpenGL rozšířeních (extensions) v programu. [online]. [cit. 8-6-2017]. Dostupné z: https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/extensions.php

[9] Wikipedia. Popis Phongova osvětlovacího modelu. [online]. [cit. 8-6-2017]. Dostupné z: https://en.wikipedia.org/wiki/Phong_reflection_model

[10] UberShader. Popis koncepce uber shaderu. [online]. [cit. 8-6-2017]. Dostupné z: https://www.gamedev.net/topic/659145-what-is-a-uber-shader/

[11] Kurz OpenGL. Online kurz OpenGL. [online]. [cit. 8-6-2017]. Dostupné z: https://learnopengl.com/

[12] Kurz OpenGL. Popis typů světelných zdrojů [online]. [cit. 8-6-2017]. Dostupné z: https://learnopengl.com/#!Lighting/Light-casters

[13] Kurz OpenGL. Definice matice gNormal. [online]. [cit. 8-6-2017]. Dostupné z: https://learnopengl.com/#!Lighting/Basic-Lighting

[14] Typy textur. Rozdíl mezi diffuse a albedo texturou. [online]. [cit. 8-6-2017]. Dostupné z: https://computergraphics.stackexchange.com/questions/350/albedo-vs-diffuse

[15] Typ textury. Detailní popis ambient occlusion textury. [online]. [cit. 8-6-2017]. Dostupné z: http://wiki.polycount.com/wiki/Ambient_occlusion_map

[16] Implementace funkce výpočet faktoru tessellace v závislosti na vzdálenosti. [online]. [cit. 2017-20-05]. Dostupné z: https://wiki.unrealengine.com/Distance_Based_DX11_Tesselation_-_Video

[17] Wikipedia. Gramova-Schmidtova ortogonalizace. [online]. [cit. 2017-24-05]. Dostupné z: https://cs.wikipedia.org/wiki/Gramova-Schmidtova_ortogonalizace

[18] Frank Luna. Implementace parallax occlusion mapping. [online]. [cit. 2017-06-04]. Dostupné z: http://www.d3dcoder.net/Data/Resources/ParallaxOcclusion.pdf

[19] Kurz OpenGL. Popis rozšíření self shadow. [online]. [cit. 2017-08-06]. Dostupné z: https://learnopengl.com/#!Advanced-Lighting/Parallax-Mapping

[20] Microsoft Visual C++ Redistributable 2015. [online]. [cit. 2017-17-06]. Dostupné z: https://www.microsoft.com/cs-cz/download/details.aspx?id=48145

[21] Generování normálových vektorů. [online]. [cit. 2017-27-06]. Dostupné z: http://www.lighthouse3d.com/opengl/terrain/index.php?normals

[22] Mgr. Lukáš Stehlík. Zobrazování povrchových detailů pomocí mapování textur. [online]. [cit. 2017-29-06]. Dostupné z: https://is.cuni.cz/webapps/zzp/detail/46520/

[23] Wikipedia. Popis formátu .obj a .mtl. [online]. [cit. 2017-05-07]. Dostupné z: https://en.wikipedia.org/wiki/Wavefront_.obj_file

[24] Frank D. Luna. Introduction to 3D Game Programming with DirectX11. First. Dules (Virginia): Mercury Learning and Information, 2012, 754s. ISBN: 978-1-9364202-2-3.

[25] Wikipedia. Barycentrické souřadnice. [online]. [cit. 2017-05-07]. Dostupné z: https://en.wikipedia.org/wiki/Barycentric_coordinate_system