VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

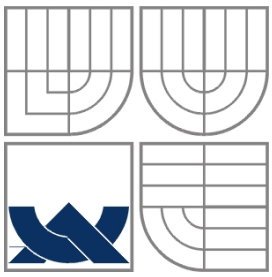
MĚŘENÍ VÝKONNOSTI OPENGL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

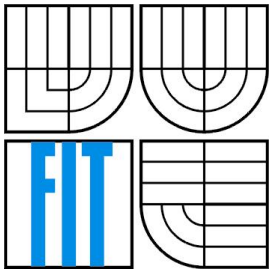
AUTOR PRÁCE
AUTHOR

PAVOL REHORČÍK

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

MĚŘENÍ VÝKONNOSTI OPENGL

OPENGL PERFORMANCE MEASUREMENT TOOL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PAVOL REHORČÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN PEČIVA, Ph.D.

BRNO 2012

Abstrakt

Tato práce se zabývá teoretickou a praktickou stránkou tvorby testovací aplikace výkonu OpenGL se zaměřením na shadere. V teoretické části práce jsou jednotlivě popsány užity techniky a přincíp použití shaderů. Jádrem celé práce je praktická část, tedy tvorba a implementace tito testovací aplikace. V závěrečné části práce zhodnocuji naměřené výsledky a možná rozšíření.

Abstract

This thesis talks about theoretical and practical side of development of testing application for OpenGL performance with focus to shaders. In theoretical part are individually described used techniques and principle of usage of shaders. The core of thesis is practical part, so creation and implementation of this application. In final part I discuss measured results and possible extensions.

Klíčová slova

OpenGL, shader, vertex shader, fragment shader, shader procesor, OpenGL shading language, GLSL, měření výkonu, point light, textúrování, simple parallax mapping, steep parallax mapping, parallax occlusion mapping, self-shadowing.

Keywords

OpenGL, shader, vertex shader, fragment shader, shader processor, OpenGL shading language, GLSL, measurement of performance, point light, texturing, simple parallax mapping, steep parallax mapping, parallax occlusion mapping, self-shadowing.

Citace

Pavol Rehorčík: Měření výkonnosti OpenGL, bakalářská práce, Brno, FIT VUT v Brně, 2012

Měření výkonnosti OpenGL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jana Pečivu, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Pavol Rehorčík
16. května 2012

Poděkování

Ďakujem p. Ing. Janovi Pečivovi, Ph.D. za ochotu a ústretovosť na konzultáciách a tiež za odbornú pomoc pri vytváraní mojej aplikácie. Ďakujem aj za poskytnutú príležitosť si vyskúšať programovanie počítačovej grafiky, hlavne shaderov.

Obsah

Úvod

V tejto práci sa venujem grafickej testovacej aplikácii a v nej použitým technikám.

V kapitole 1 stručne vysvetľujem pojem GPU, uvádzam aké aplikácie sa dajú použiť na zmeranie výkonu, čo je benchmark a aká jednotka sa najčastejšie používa ako ukazovateľ výkonu. Tiež spomínam od čoho závisí výkon GPU.

V kapitole 2 popisujem pojmy shader, shader procesor, vertex shader, fragment shader a v závere tejto kapitoly popisujem, aké sú špeciálne typy premenných, ktoré shadere používajú.

Kapitola 3 obsahuje popis jazyka pre popis shaderov a tiež ich preklad.

V kapitole 4 popisujem všetky použité technológie od jednoduchého textúrovania až po pokročilé textúrovacie metódy. Táto kapitola spolu s kapitolou treťou a druhou popisujú všetko, čo bolo potrebné sa naučiť pre vytvorenie tejto aplikácie.

V kapitole 5 je popísaná štruktúra aplikácie, na akom hardvéry bola aplikácia vyvíjaná, aká verzia OpenGL bola použitá, spôsob uloženia shaderov, ako sa získava a vypisuje FPS a hlavne implementácia každej techniky popísanej v kapitole 4 pomocou shaderov. V prílohe A uvedený je screenshot z každej scénky.

V kapitole 6 sa venujem jednotlivým testovaným grafickým procesorom, kde uvádzam ich základné parametre a aj výsledky podľa mojej aplikácie.

V závere popisujem postup vytvárania aplikácie a najväčšie problémy s ktorými som sa pri vytváraní stretol. Diskutujem aj vhodné a možné rozšírenia.

Pre vytvorenie najzákladnejšej štruktúry aplikácie a aj pri pochopení najzákladnejších pojmov súvisiacich s počítačovou grafikou mi bola nápomocná kniha v použitej literatúre pod bodom [1]. Pre pochopenie pojmu shader a s ním spojenej práce mi boli nápomocné knihy [1] [2] a publikácia [7]. Princíp použitých techník mi objasnili publikácie [3] [4] [5] [6] [8] [9] [10]. Pri programovaní jednotlivých techník pomocou shaderov som čerpal z knihy [2] a publikácii [6] [11]. Pre popis jednotlivých GPU a pojmu GPU som čerpal z bodov [12] [13] [14].

Kapitola 1

1 GPU a meranie jeho výkonu

GPU, čiže „graphics processing unit“, alebo jednoducho grafický procesor, je jednotka určená pre akceleráciu výpočtu obrazu. Použitie tohto termínu sa spája s rokom 1999, kedy spoločnosť nVidia vydala prvý grafický procesor pre verejnú sféru, Geforce 256, ktorý podporoval hardwarové transform and lighting (T&L). Hardware T&L je predchodcom shaderov, ktorým sa venujem v nasledujúcich kapitolách.

Toto je tiež doba, kedy sa začalo rozširovať viac modelov GPU aj od konkurenčných spoločností a teda sa testovanie ich výkonu stalo postupne populárnym.

Ako ukazovateľ výkonu sa najčastejšie používa počet snímkov za sekundu (FPS – frames per second), ktoré dokáže GPU vykresliť. Pre zmeranie je možné použiť akúkoľvek dnes dostupnú hru (merať FPS pomocou externej aplikácie, napr. Fraps, v nejakom úseku, tiež množstvo moderných hier ma vstavané benchmarky), alebo použiť syntetický benchmark. Syntetický benchmark je všeobecne aplikácia na meranie výkonu pomocou pred pripravenej množiny testov. V počítačovej grafike sa používajú pred pripravené grafické scénky. Týmto sa zaručí, že pri každom spustení benchmarku bude scénka presne identická a teda aj porovnanie výkonu bude hodnoverné. Dnes sú veľmi populárne syntetické benchmarky od spoločnosti Futuremark.

Tento spôsob, syntetický benchmark, som si zvolil aj ja pre svoju aplikáciu.

Výkon GPU dnes závisí hlavne od taktu samotného GPU, taktu shader procesorov a ich počtu (pojmu shader sa venujem v kapitole 2). Prvé GPU schopné spracovať jednoduchý vertex a fragment shader boli Geforce 3 a Radeon 8500 s počtom programovateľných jednotiek (vertex a fragment procesorov) rádovo v jednotkách. Dnes je ich počet pre najvýkonnejšie GPU niekoľko sto násobne vyšší, Geforce GTX 680 má 1536 shader procesorov a Radeon HD 7970 má 2048 shader procesorov.

Samozrejme, tieto nové shader procesory sú oveľa viac zložitejšie ako tie v začiatkoch.

Kapitola 2

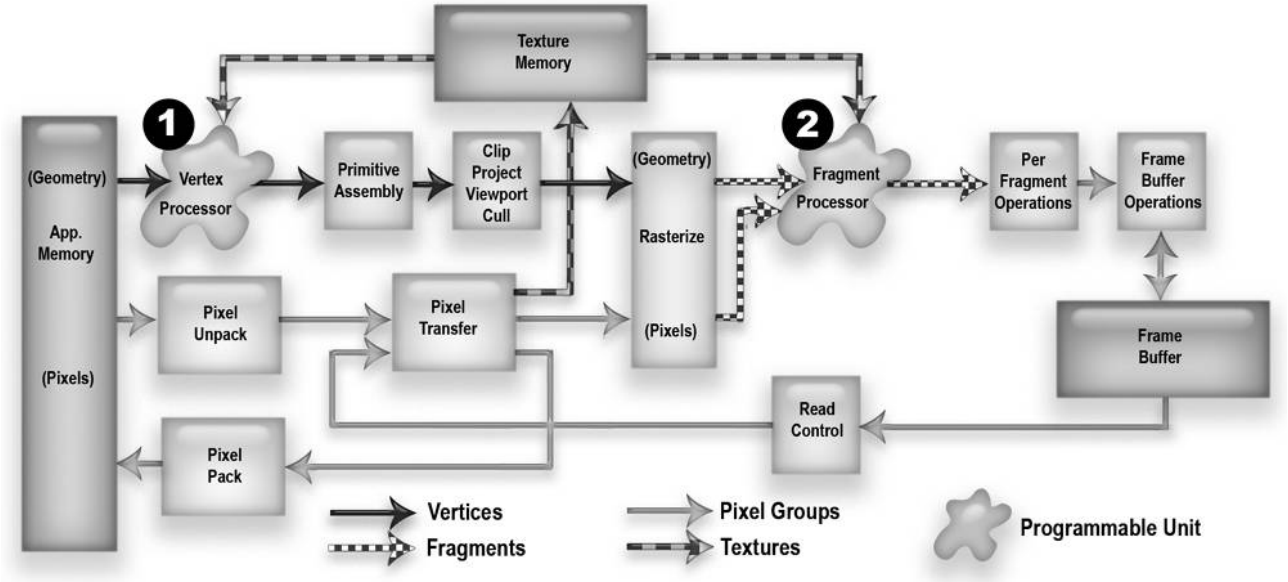
2 Shadere

2.1 Vykresľovacie potrubie – pipeline

Pevné vykresľovacie potrubie (fixed pipeline) sa nahradilo programovateľným potrubím (programmable pipeline) pre časti, ktoré sa stali mimoriadne zložité. Tieto časti sú spracovanie vertexu a fragmentu. Spracovanie vertexu zahŕňa operácie, ktoré sa dejú s každý vertexom, najviac transformácie a osvetlenie. Fragmenty sú „per-pixel“ dátové štruktúry, ktoré sú vytvorené rasterizáciou grafických primitív. Fragment obsahuje všetky údaje potrebné na aktualizovanie jedného miesta vo „frame buffery“. Spracovanie fragmentu obsahuje operácie, ktoré sa dejú na úrovni fragmentu, najviac čítanie z pamäti textúr a aplikovanie hodnoty textúr na každý fragment.

2.2 Shader

Shader je krátky program, ktorý počíta vykresľovanie na grafickom hardvéry s vysokou flexibilitou. Shader programuje programovateľné potrubie grafického hardvéru a teda je možné vytvoriť špecializované efekty. Vertex shader program sa volá pre každý vertex a je vykonávaný na vertex procesore, fragment shader program sa volá pre každý pixel a je vykonávaný na fragment procesore. Kombinácia pevného potrubia a programovateľného potrubia pre vykreslenie jedného primitíva nieje možná. Nie je možné napríklad existujúcim pevným potrubím vypočítať transformáciu vertexu a mať vertex shader iba pre osvetlenie. Tiež nie je možné počítať pevným potrubím hmlu a mať fragment shader iba pre textúrovanie. Ak sa už raz rozhodneme pre použitie programovateľného potrubia, tak ním musia byť vykonané všetky úkony. Pevné potrubie medzi vertex procesorom a fragment procesorom zostáva zachované. Obrázok 2.2 ilustruje programovateľné potrubie OpenGL 2.0.



Obrázok 2.2: programovateľné potrubie OpenGL 2.0

Programovateľné jednotky OpenGL verzie 2.0 sú:

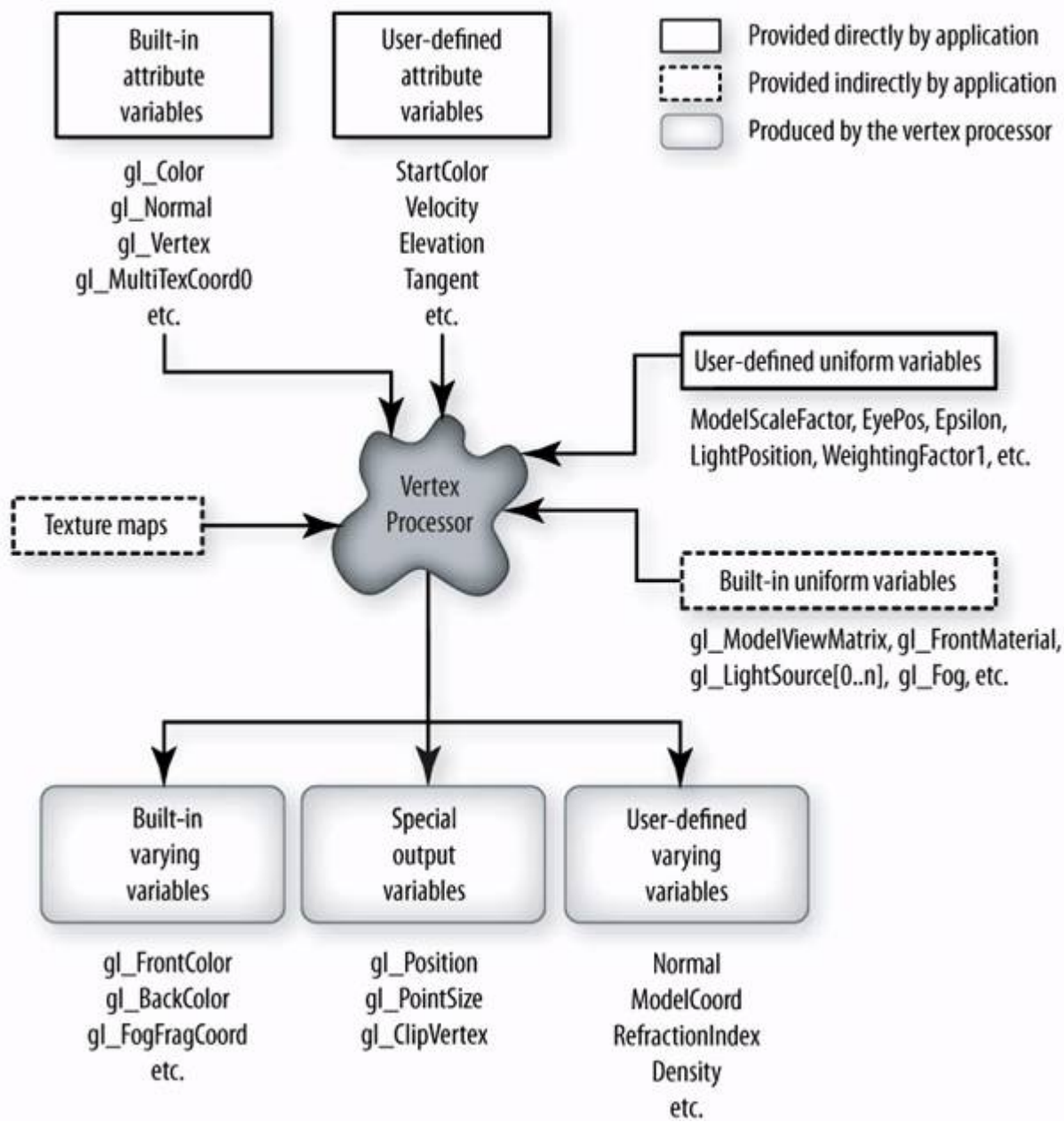
1) Vertex procesor
2) Fragment procesor

2.3 Vertex procesor

Vertex procesor je programovateľná jednotka programovateľného potrubia, ktorá operuje s prichádzajúcim vertexom a sním spájanými dátami. Vertex procesor obyčajne vykonáva grafické operácie ako sú:

- transformácia vertexu
- normálová transformácia a normalizácia
- generovanie textúrovacích súradníc
- osvetlenie
- aplikácia farby

Vertex shader program produkuje výstupné dáta na základe poskytovaných vstupných dát (obrázok 2.3).



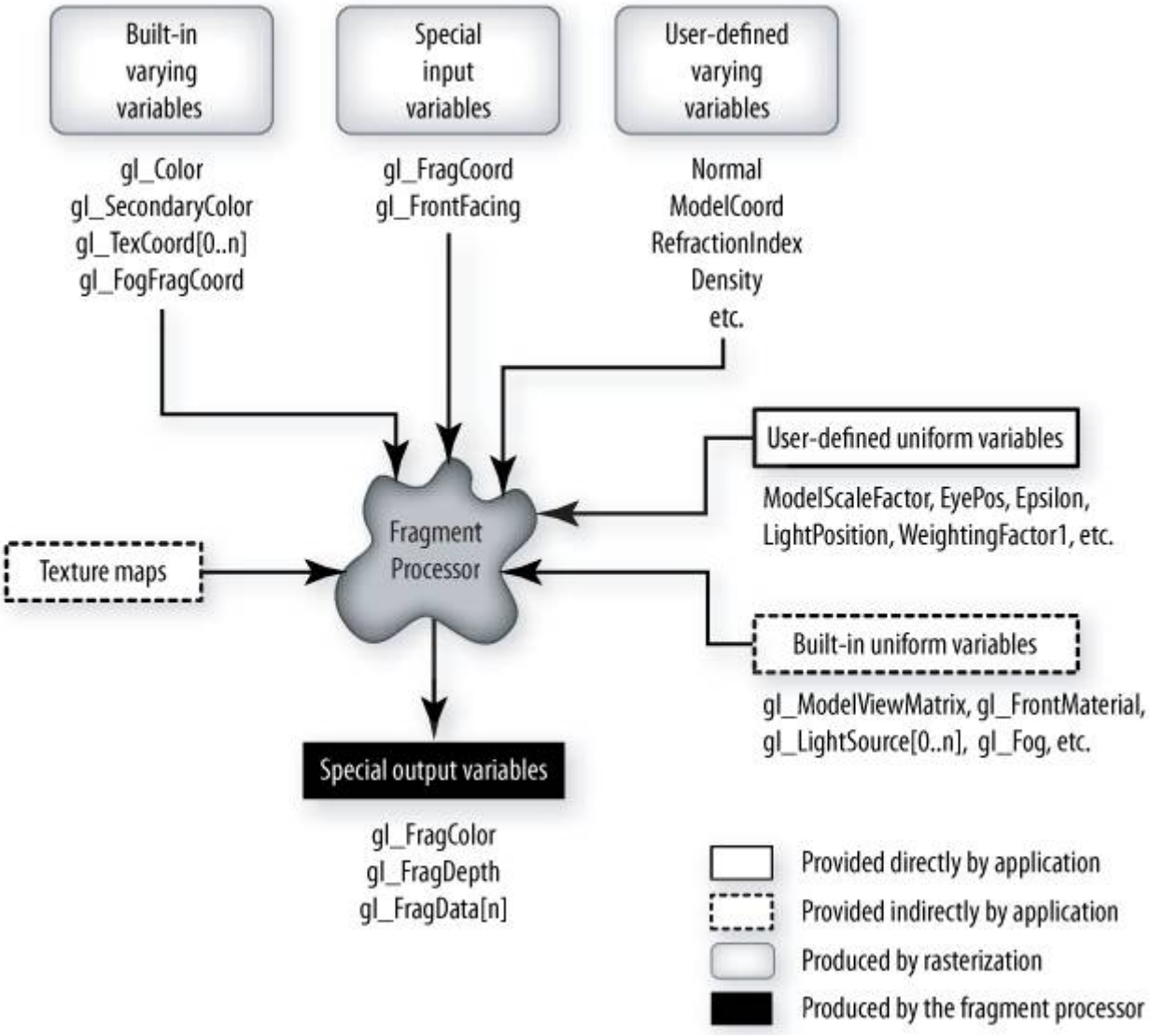
Obrázok 2.3: vstupy a výstupy vertex procesoru

2.4 Fragment procesor

Fragment procesor je programovateľná jednotka programovateľného potrubia, ktorá operuje nad fragmentom a sním spájanými dátami. Fragment procesor nemôže meniť pozíciu fragmentu. Obyčajne vykonáva tieto grafické operácie:

- operácie nad interpolovanými hodnotami
- prístup k textúre a jej aplikovanie
- hmla
- „per-pixel“ osvetlenie
- súčet farieb

Fragment procesor pracuje s interpolovanými hodnotami z vertex procesoru. Spracovanými hodnotami napĺňa špecializované premenné. Napríklad gl_FragColor napĺňa výslednou farbou (obrázok 2.4).



Obrázok 2.4: vstupy a výstupy fragment procesoru

2.5 Shader procesor

V dnešnej dobe sa používa pojem „shader procesor“. Shader procesor je univerzálna programovateľná jednotka pre spracovanie vertexu alebo fragmentu. Spojením vertex procesoru a fragment procesoru do jednej univerzálnej jednotky umožňuje širšie použitie. Teda aplikácie už nie sú závislé na fixnom počte vertex a fragment procesorov, ale podľa potreby sa rozhodne na koľkých shader procesoroch budú spracovávané vertexy a na koľkých fragmenty. Avšak funkcionalita vertex shaderu a fragment shaderu zostala zachovaná, len sa spracovávajú na tej istej jednotke.

2.6 Špeciálne typy premenných využívané shaderami

Premenné definované vo vertex shadery sú označené ako „attribute“, atribútové premenné. Atribútové premenné reprezentujú hodnoty, ktoré sú často posunuté z aplikácie do vertex procesoru. Pretože tento typ premenných sa používa iba pre dáta z aplikácie pre definovanie vertexov, je povolený iba pre vertex shader. Existujú dva typy atribútových premenných, vstavané a používateľom definované. Aplikácie môžu ponúkať atribútové hodnoty medzi volaniami **glBegin** a **glEnd**, alebo s volaním vertexového poľa a teda hodnota atribútu môže byť pre každý vertex iná. Vstavané atribútové premenné sú napríklad **gl_Color** reprezentujúca farbu, **gl_Normal** reprezentujúca normálu, **gl_Vertex** reprezentujúca polohu vertexu a ďalšie. Programátor si môže definovať vlastné atribútové premenné pomocou príkazu **glVertexAttrib** a pomocou príkazu **glBindAttribLocation** naviazať index atribútovej premennej na jej meno vo vertex shadery.

„Uniform“ pemenné posúvajú dáta z aplikácie do vertex procesoru aj do fragment procesoru. Ale „Uniform“ premenné nemôžu byť špecifikované medzi volaniami **glBegin** a **glEnd**, takže ich hodnota sa môže zmeniť maximálne raz za primitívum. Existujú tak isto vstavané aj používateľom definované „uniform“ premenné. Používateľom definované „uniform“ premenné sa definujú pomocou príkazu **glUniform** a ich index sa naviaže na meno v shader procesory pomocou príkazu **glGetUniformLocation**.

Premenné, ktoré definujú dáta, ktoré sú posúvané z vertex procesoru cez interpolátor do fragment procesoru, sa nazývajú „varying“ premenné. Taktiež existujú vstavané a používateľom definované. Vstavané „varying“ premenné zahŕňajú farbu a súradnice textúr. Vertex shader používa používateľom definované „varying“ premenné pre čokoľvek, čo potrebuje byť interpolované. Takéto „varying“ premenné musia byť definované aj vo fragment shadery a ich typy musia byť zhodné s tými, definovanými vo vertex shadery.

Kapitola 3

3 Jazyk a preklad shaderov

3.1 OpenGL Shading Language

OpenGL Shading Language (GLSL) je založený na syntaxi programovacieho jazyka ANSI C. Toto je zámerné, aby bol jazyk ľahko používateľný tými, kto ho bude najviac využívať, teda programátori grafických aplikácii v C a C++.

Základná štruktúra programu napísaného v GLSL je taká istá ako programov napísaných v C. Vstupný bod je funkcia void main(). Konštanty, identifikátory, operátory, výrazy sú rovnaké ako v C. Riadenie toku pre cykly, if-else a funkcie je virtuálne identické s C.

Avšak GLSL má tiež špecializované črty vďaka jeho špecializovanému charakteru, zamýšľaného pre použitie v grafických algoritmoch. Tu sú niektoré črty ktoré boli pridané do OpenGL Shading Language, ktoré sú odlišné od ANSI C.

Vektorové typy sú podporované pre hodnoty floating-point, boolean a integerov. Pre floating-point hodnoty sa hovorí o týchto vektorových typoch ako vec2 (dva floaty), vec3 (tri floaty) a vec4 (štyri floaty). Operátory s týmito typmi pracujú rovnako jednoducho ako s so skalárnymi typmi. Napríklad pre sčítanie vektora v1 a vektora v2 stačí uviesť v1 + v2. Ďalej, samostatné zložky vektorov môžu byť pristupované ako zložky poľa, alebo ako zložky štruktúry. Na hodnoty farby sa pristupuje pridaním .r pre prvú farebnú zložku (zložku vektora), .g pre druhú zložku, .b pre tretiu zložku a .a pre štvrtú zložku, za identifikátor vektora. Na hodnoty pozície sa pristupuje pridaním .x, .y, .z, .w za identifikátor vektora a na textúrovacie hodnoty sa pristupuje pridaním .s, .t, .p a .q. Tiež sa dá pristúpiť na viaceré zložky jedného vektora pridaním za jeho meno napríklad .xy.

Floating-point maticové typy sú tiež podporované. Typ mat2 zodpovedá matici 2x2 floatig-point hodnôt. Typ mat3 zodpovedá matici 3x3 a typ mat4 zodpovedá matici 4x4. Stĺpce matice môžu byť zvolené so syntaxou poľa a ďalej sa s takto zvoleným stĺpcom môže narábať ako s vektorom, spôsobom, ktorý je zmienený vyššie.

Ďalej bol pridaný mechanizmus pre prístup do textúrovacej pamäte. Toto umožňuje špeciálny typ nazývaný samplers. Samplers je špeciálny typ premennej, ktorá získava konkrétnu textúru. Premenná typu sampler1D sa použije pre získanie 1D textúry, premenná typu sampler2D pre získanie 2D textúry a pod.

Tiež boli pridané kvalifikátory attribute, uniform a varying pre určenie vstupov a výstupov shaderov. Attribute premenné prenášajú často sa meniace hodnoty z aplikácie do vertex shaderu, uniform premenné prenášajú nie často sa meniace hodnoty z aplikácie do akéhokoľvek shaderu a varying premenné prenášajú interpolované hodnoty z vertex shaderu do fragment shaderu. Shadere napísané v GLSL môžu používať vstavané premenné začínajúce sa prefixom „**gl_**“.

Z jazyka C++ si jazyk GLSL priniesol preťažovanie funkcii, ktoré je v GLSL veľmi využívané vstavanými funkciami, potom konštruktory, premenné môžu byť deklarované vtedy keď ich treba, typ bool je rovnako podporovaný ako v C++ a funkcie musia byť deklarované pred použitím buď definíciou, alebo iba prototypom.

GLSL ale na rozdiel od ANSI C nepodporuje automatickú konverziu dátových typov, ďalej ukazovatele, reťazce alebo znaky, byte-y, short-y alebo long integer-e, lebo tieto nie sú potrebné.

3.2 Preklad a spustenie shaderu

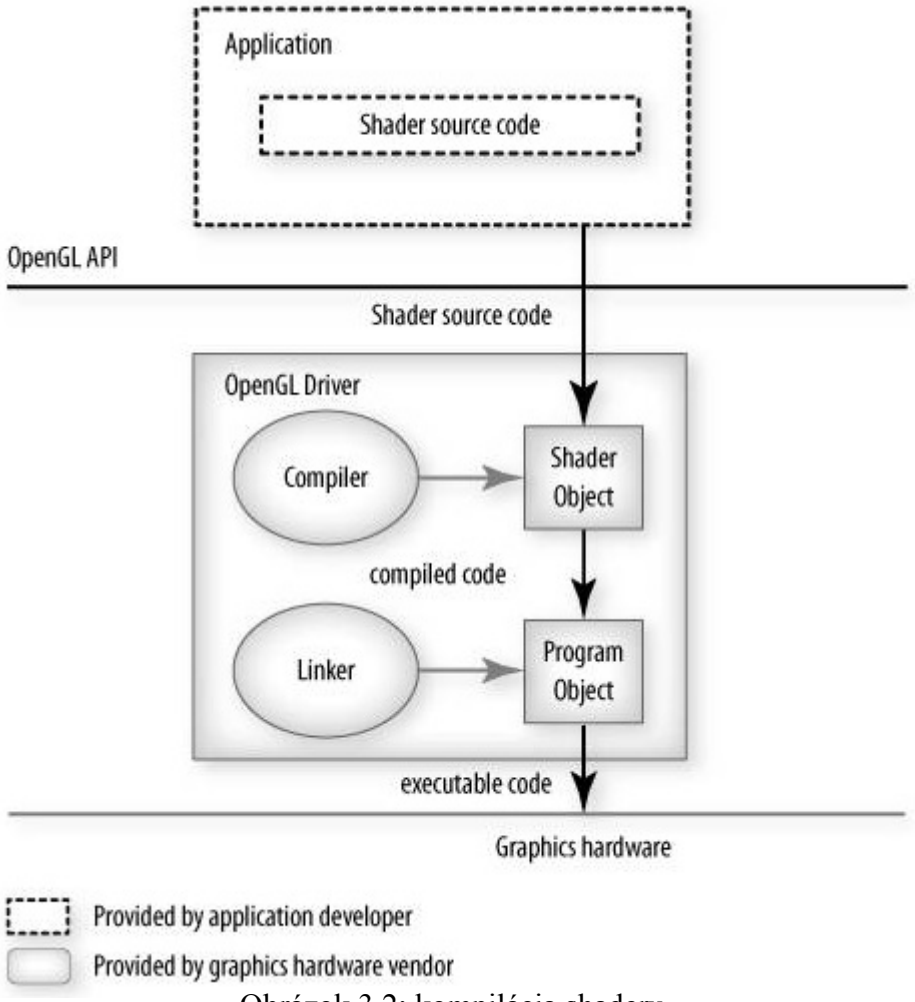
Nezáleží na prostredí v ktorom je OpenGL implementované, vždy spadá do kategórie ovládačov, pretože OpenGL riadi zdielaný prístup na grafický hardware. Zdrojový kód shaderov je poskytnutý OpenGL ovládaču a v tomto prostredí je skompilovaný do strojového kódu.

Aplikácia komunikuje s OpenGL cez volanie funkcii, ktoré sú súčasťou OpenGL API. Nová OpenGL funkcia **glCreateShader**, povoluje aplikácii alokovať vo vnútri OpenGL ovládača dátové štruktúry, ktoré sú nevyhnutné pre uloženie shadera. Tieto dátové štruktúry sa volajú „shader objects“, alebo objekty shaderu. Potom ako bol „shader object“ vytvorený, aplikácia poskytuje zdrojový kód shaderu volaním **glShaderSource**. Potom ako bol zdrojový kód shaderu pripojený k objektu shaderu, je skompilovaný volaním **glCompileShader**.

OpenGL manažér dátových štruktúr, ktoré sa chovajú ako kontajner pre objekty shaderov, sa nazýva „program object“, alebo objekt programu. Objekt programu sa vytvorí pomocou **glCreateProgram**. Skompilované objekty shaderov sa pripoja k objektu programu cez volanie **glAttachShader**. Potom ako sú pripojené k objektu programu sa zlinkujú volaním **glLinkProgram**.

Výsledok je jeden alebo viac spustiteľných shaderov, ktoré môžu byť zavolané pomocou pomocou **glUseProgram** ako časť aktuálneho stavu OpenGL. Takto zavolaný shader na grafický objekt je zodpovedný za celé spracovanie jeho vertexov a framentov.

Túto činnosť ilustruje obrázok 3.2.



Obrázok 3.2: kompilácia shaderu

Kapitola 4

4 Požité Technológie

4.1 Textúrovanie pomocou shaderov

Po aplikácií je požadované, aby nastavila počiatočný textúrovací stav správne, pred tým ako bude shader pristupovať do textúrovacej pamäte. Aplikácia musí teda vykonať nasledujúce kroky, aby nastavila textúru pre použitie v shadery:

1. Vybrať textúrovaciu jednotku a aktivovať ju pomocou **glActiveTexture**.
2. Vytvoriť objekt textúry a naviazať ho na aktivovanú textúrovaciu jednotku pomocou **glBindTexture**.
3. Nastaviť rôzne parametre textúrovacieho objektu pomocou **glTexParameter**.
4. Definovať textúru volaním **glTexImage**.

Po takto nastavenom textúrovacom stave, môže shader k textúre pristúpiť. K tomu využíva uniform premennú typu „sampler“. V shadery musí byť deklarovaná takáto uniform premenná pre každú textúru na ktorú chce shader pristúpiť. Typ sampleru určuje typ texúry ku ktorej chceme pristúpiť. Existuje viacero typov samplerov, ale v mojom projekte sa jedná o typ „sampler2D“, ktorý sprístupňuje 2D textúru. Potom vstavané funkcie vykonávajú prístup na textúru vo vnútri shaderu, funkcia **texture2D**. Prvým argumentom týchto funkcii je typ sampleru a ten musí korešpondovať s menom funkcie. Teda pre sampler2D sa používa vstavaná funkcia texture2D. Ako druhý argument sú textúrovacie súradnice. Hardware používa tieto súradnice aby rozhodol, ktoré časti textúry majú byť sprístupnené. Pre 2D textúru sú súradnice typu vec2, čo značí vektor s dvoma zložkami, x a y.

„Multitexturing“ sa v shadery uskutoční tiež veľmi jednoducho. Stačí mať takto aplikáciou pripravených viacero textúr, v shadery k nim pristúpiť a hodnoty z nich získané medzi sebou ľubovoľným algoritmom kombinovať.

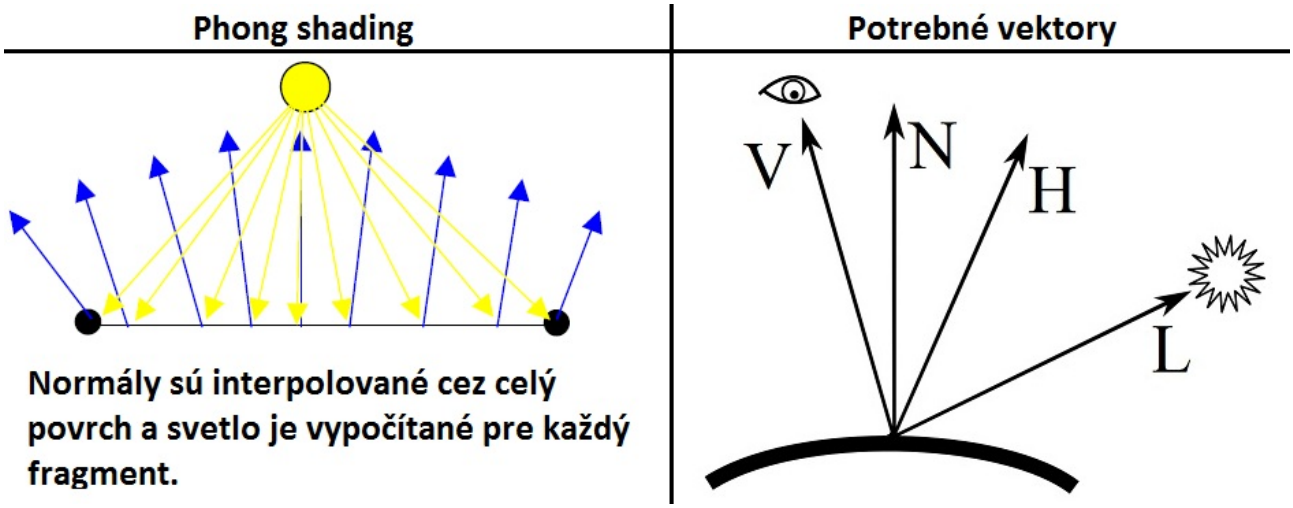
4.2 Point Light

Point light, alebo bodové svetlo, napodobňuje svetlo, ktoré je blízko scény, alebo priamo v nej, napríklad lampy. Pre bodové svetlo platí, že svetlo, ktoré dopadá na povrch nejakého objektu, sa zoslabuje, ako sa zdroj bodového svetla dostáva ďalej od osvetleného objektu. Toto sa naźyva „attenuation“, čiže útlm. Každé bodové svetlo má konštantný, lineárny a kvadratický faktor útlmu. Útlm sa počíta: **attenuation = 1,0 / (c + l.d + q.d^2)**, kde **c** je konštantný faktor, **l** lineárny faktor, **q** kvadratický faktor útlmu a **d** je vzdialenosť svetelného zdroja od osvetlenej plochy.

Ďalej pre bodové svetlo platí, že pre výpočet spekulárnej zložky sa už nedajú použiť predpočítané hodnoty z **gl_LightSource[i].halfVector**. Tento vektor, „half vektor“, čo je vektor v polovici uhla medzi vektorom pozorovateľa a vektorom svetelného zdroja, sa musí prepočítať pre každý fragment, **H = L + V**, kde **H** je hľadaný half vektor, **L** je vektor k zdroju svetla a **V** je vektor k pozorovateľovi, alebo kamere. Vektory **L** a **V** máme k dispozícii. Potom sa intenzita spekulárnej zložky vypočíta ako skalárny súčin half vektoru a normálového vektoru umocnený žiarivosťou materiálu a následne vynásobený útlmom.

Intenzita osvetlenia sa vypočíta ako skalárny súčin normálového vektoru a vektoru svetelného zdroja a vynásobené útlmom.

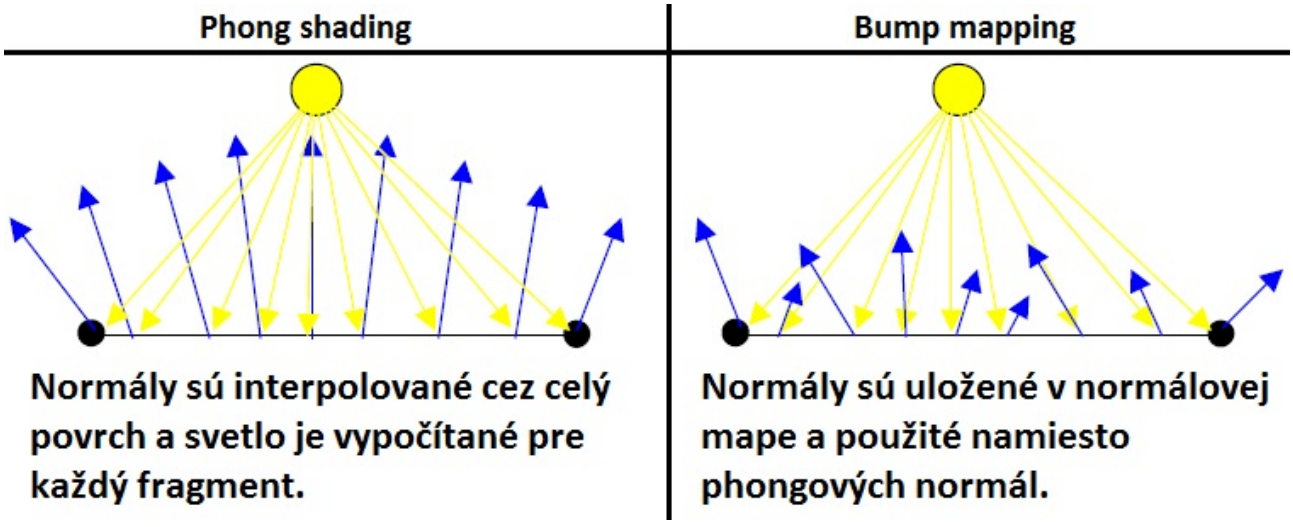
Použitý je Phongov osvetľovací model, alebo „Phong shading“, kde sú normály interpolované cez celý povrch a intenzita svetla je vypočítaná pre každý fragment. Jedná sa teda o „per pixel“ osvetlenie (obrázok 4.2).



Obrázok 4.2: Phongov osvetľovací model a potrebné vektory pre výpočet intenzity osvetlenia bodovým svetlom aj so spekulárnou zložkou, kde L je svetelný vektor, H je half vektor, N je normálový vektor a V je vektor ku kamere.

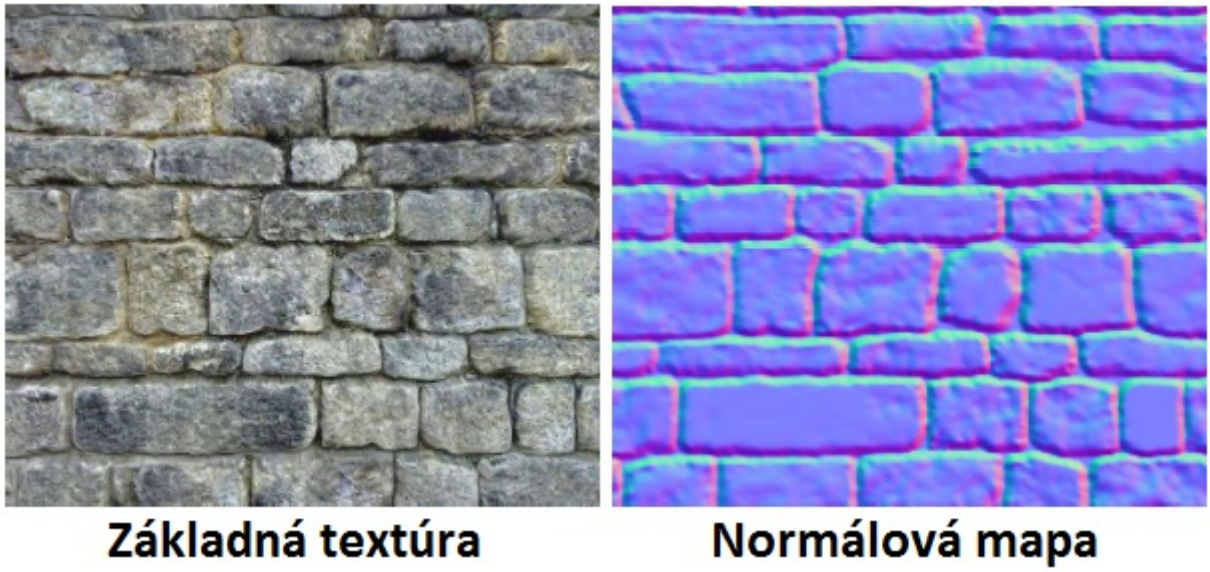
4.3 Bump mapping

Bump mapping upravuje normály povrchu pred tým, ako je aplikované svetlo. Táto technika nemení geometriu, alebo povrch objektu, iba pridáva zdanlivú geometrickú zložitosť. Preto je veľmi vhodná na imitovanie rôznych malých nerovností povrchu. Pri použití bump mappingu na väčšie nerovnosti vhodné nie je, keďže nemení geometriu objektu, objekt prestáva pôsobiť realistickým dojmom a začína pozorovateľa vyrušovať. Tiež je bump mapping veľmi vhodný ako doplnkový efekt k ďalším, ako napríklad k parallax mappingu. Kľúč spočíva v mapovaní normál (obrázok 4.3.1).



Obrázok 4.3.1: rozdiel medzi phongovým osvetľovacím modelom a bump mappingom

Normály povrchu sú shaderu poskytované obyčajne pomocou normálovej mapy. Normálová mapa je textúra, v ktorej sú normály zakódované pomocou farby a táto textúra definuje normálu pre každý fragment namiesto interpolovaných normál, ako ich poznáme z Phongovho osvetľovacieho modelu (obrázok 4.3.2). Vektory avšak môžu mať záporné hodnoty, ale dátový typ, ktorý ukladá farbu textúry, záporné hodnoty mať nemôže. Preto treba hodnoty vektorov prispôsobiť na uloženie do mapy, čiže textúry. Potom, keď čítame hodnoty z tejto mapy, sú v intervale [0, 1]. Preto pred aplikovaním osvetlenia je potrebné v shadery tieto normály transformovať naspäť, aby boli opäť v rozsahu [-1, 1].



Obrázok 4.3.2: Základná mapa a jej normálová mapa. Z normálovej mapy je cítiť nádych priestoru.

Avšak tieto normály nie sú v rovnakom priestore so svetelným vektorom. Preto treba vypočítať svetelný vektor vo vertex shadery, transformovať ho do rovnakého priestoru v ktorom sú normály a podať ho interpolovaný fragment shaderu.

Tento spoločný priestor sa vola tangentový priestor, alebo „tangent space“ a je definovaný pre každý vertex. Potreba tohto priestoru spočíva v tom, aby normály zostali zachované. Napríklad, keby normály boli uložené v priestore objektu a keby sme model rotovali, museli by sme rotovať aj normály, aby bola zachovaná koherencia. Ale s normálami relatívnymi ku každému vertexu, toto nieje potrebné. Normály v tangentovom priestore sú nezávislé na polohe a orientácii modelu.

Aby sme vytvorili tangentový priestor, potrebujeme definovať orthonormálnu bázu pre každý vertex, ktorá nám definuje tento priestor. K tomu potrebujeme 3 vektory. Normálový vektor vertexu, ktorý je bežne definovaný spolu s vertexom, tangentový vektor (po tomto vektore sa priestor nazýva tangentový priestor), ktorý je kolmý na normálový, tento vektor je potrené tiež definovať spolu s vertexom ako jeho atribút a nakoniec binormálový vektor, ktorý je kolmý na normálový vektor a zároveň aj na tangentný vektor. Tento vektor sa dopočíta vo vertex shadery ako vektorový súčin normálového vektoru a tangentového vektoru. Nesprávne definovanie tangentného vektoru spôsobí chyby v osvetlení.

Z týchto 3 vektorov zhotovýme maticu TBN (Tangent, Binormal, Normal) a touto maticou vynásobíme vektor ktorý chceme transformovať, vektor v objektovom priestore, „object space vector“, v tomto prípade svetelný vektor a dostaneme vektor v lokálnom priestore, „surface local vector“.

$$TBN = \begin{pmatrix} Tx\ Ty\ Tz \\ Bx\ By\ Bz \\ Nx\ Ny\ Nz \end{pmatrix},$$

$$(Sx\ Sy\ Sz) = (Ox\ Oy\ Oz) . \begin{pmatrix} Tx\ Ty\ Tz \\ Bx\ By\ Bz \\ Nx\ Ny\ Nz \end{pmatrix},$$

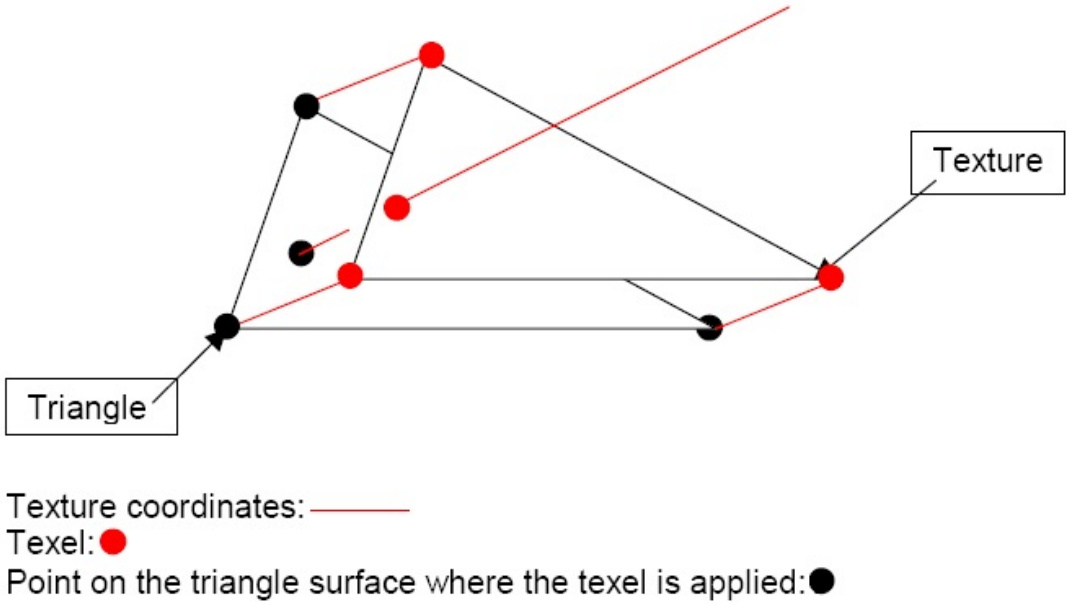
kde S je vektor v lokálnom priestore, O je vektor v objektovom priestore.

Normálová mapa sa už len teraz namapuje pomocou rovnakých textúrovacích súradníc ako základná mapa, hodnota farby sa prepočíta na hodnotu normálového vektoru prislúchajúceho danému fragmentu a takto získaný normálový vektor sa použije pre výpočet osvetlenia fragmentu (obrázok 4.3.3).

Obrázok 4.3.3: dojem priestoru pomocou bump mappingu

4.4 Jednoduchý parallax mapping

Parallax mapping je texhnika, ktorá modifikuje textúrovacie súradnice. Keď sa textúruje povrch, tak sa textúra aplikuje na povrch presne podľa textúrovacích súradníc jedna k jednej, čo značí, že pre každý texel (pixel s textúrou) sú lúče paralelné, ako znázorňuje obrázok 4.4.1.



Obrázok 4.4.1: mapovanie textúry na povrch

Pomocou informácie o výške a vektoru kamery (eye vektor) je možné vykonať posuv virtuálne vyšších texelov. Takto sa dosiahne perspektívny efekt. Jednoduchý parallax mapping je neiteratívna metóda, ktorá raz prečíta výšku z výškovej mapy spracovávaného bodu.

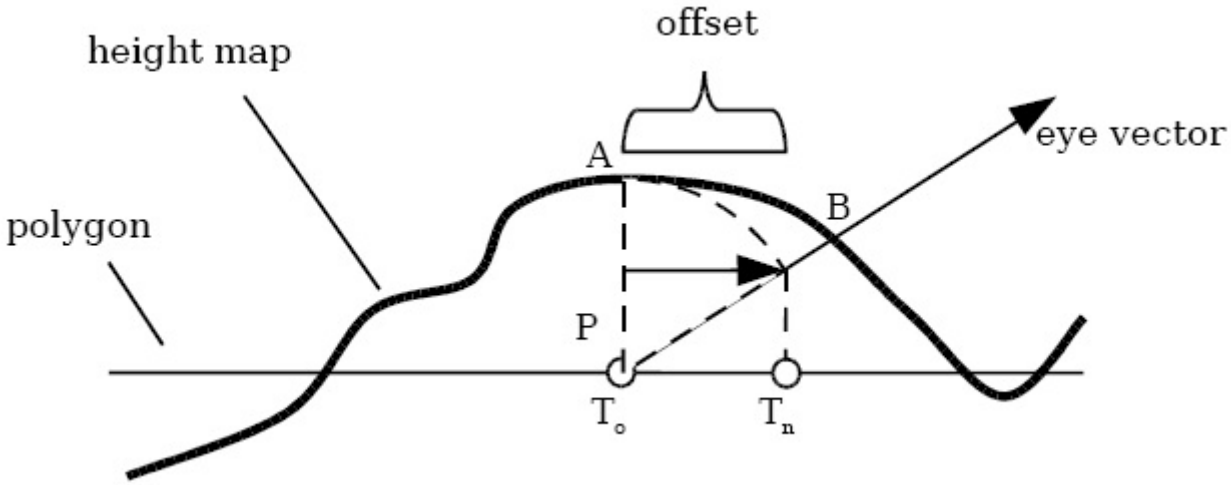
Výšková mapa je textúra, ktorá sa skladá iba z odtieňov šedej farby (obrázok 4.4.2).

Obrázok 4.4.2: základná textúra a jej výšková mapa

Výšku treba pretransformovať z rozsahu [0, 1], v ktorom je uložená vo výškovej mape do rozsahu lepšie zodpovedajúcemu fyzikálnym vlastnostiam povrchu, ktorý je simulovaný. Preto sa

pôvodná výška **h** z výškovej mapy upraví pomocou „scale faktoru“ **s** a „bias-u“ **b** a to nasledovne: **hsb = h . s + b**, kde **hsb** je upravená výška, pripravená pre ďalšie použitie.

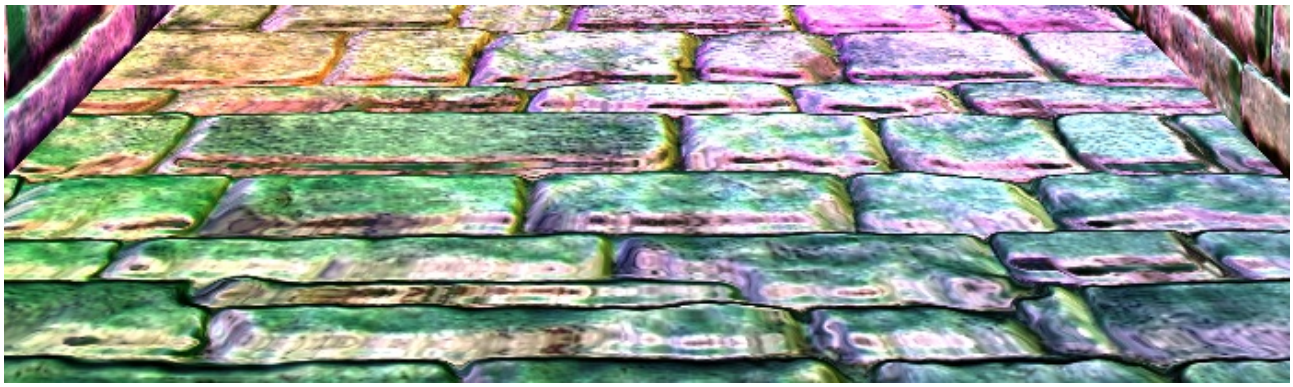
Teraz treba vypočítať posun textúrovacích súradníc. Obrázok 4.4.3 znázorňuje tento výpočet.



Obrázok 4.4.3: Výpočet novej textúrovacej súradnice, kde To je pôvodná súradnica a Tn je nová súradnica.

Bod **A** reprezentuje bod, ktorý by sa namapoval pri bežnom textúrovaní, ale ak by sa pozorovateľ pozeral na skutočný povrch nie iba na otextúrovaný polygón, videl by bod **B**. Na výpočet offsetu textúrovacích súradníc v bode **P,** vektor pozorovateľa, alebo „eye vektor“, musí byť normalizovaný. Výška sa prečíta z výškovej mapy v bode pôvodnej textúrovacej súradnice **To** a upraví sa vyššie opísaným spôsobom. Offset je vypočítaný vystopovaním vektora paralelného k polygónu z bodu P smerom k eye vektoru a jeho dĺžka je obmedzená na veľkost **hsb**. Tento nový vektor je hľadaný offset a pripočíta sa k pôvodnej textúrovacej súradnici **To** a vznikne nová textúrovacia súradnica **Tn**, ktorá sa použije na nanesenie základnej textúry: **Tn = To + (hsb . E{x, y})**, kde E je eye vektor. Ďalej eye vektor musí byť v tangentnom priestore z tých istých dôvodov ktoré platia pre light vektor pri bump mappingu.

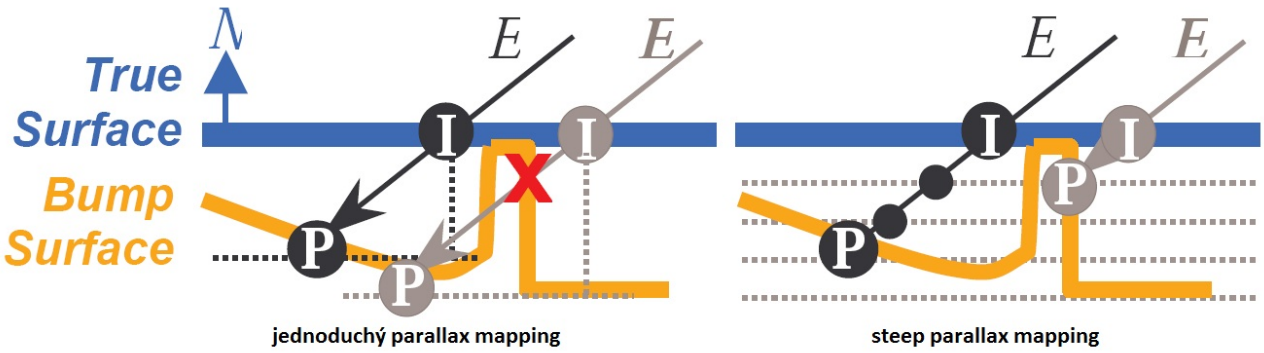
Keďže jednoduchý parallax mapping je iba aproximácia, akákoľvek hodnota môže byť použitá na limitovanie dĺžky offsetu, ale **hsb** pracuje dostatočne dobre a navyše je použitý menší počet inštrukcii. Tento efekt sa dobre kombinuje bump mappingom, ale nesmie sa zabudnúť na použitie nových textúrovacích súradníc aj pre normálovú mapu (obrázok 4.4.4). Ako je z obrázka 4.4.4 vidieť, táto metóda spôsobuje „chyby na hranách“. Tieto chyby v zobrazení eliminujú iteratívne metódy parallax mappingu, ktorým sa venujem v nasledujúcich kapitolách.



Obrázok 4.4.4: Jednoduchý parallax mapping v kombinácii s bump mappingom. Je vidieť „roztiahnutie na hranách“. Obrázok pochádza priamo z mojej aplikácie.

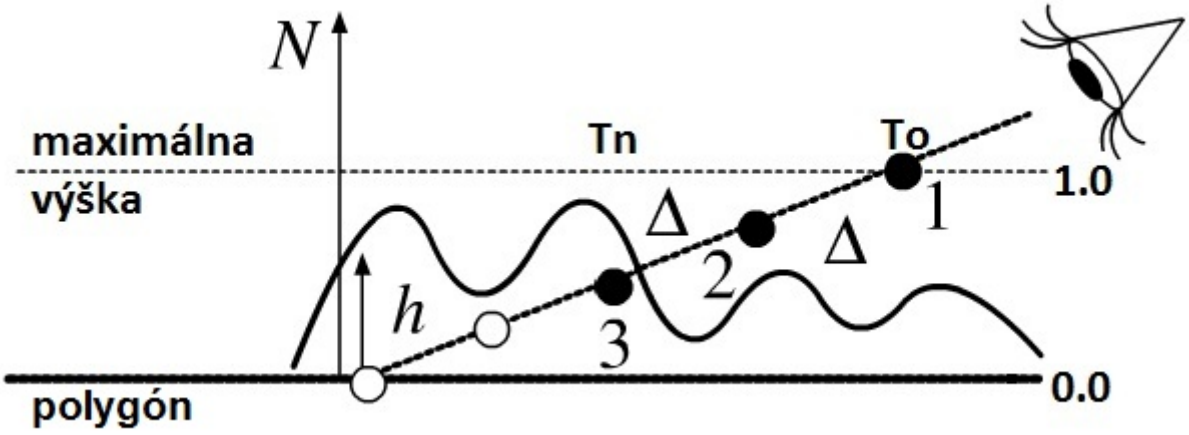
4.5 Steep parallax mapping

Steep parallax mapping, ktorý sa tiež nazýva „linear search“, alebo „ray marching“, ako názov napovedá, tiež upravuje textúrovacie súradnice, ale oproti predošlej metóde je táto metóda iteratívna. Iteratívne metódy skúmajú výškovú mapu globálne aby našli priesečník lúča a najvyššieho poľa (obrázok 4.5.1). Preto oveľa častejšie pristupujú k výškovej mape v textúrovacej pamäti.



Obrázok 4.5.1: Rozdiel medzi iteratívnym parallax mappingom a jednoduchým parallax mappingom. Iteratívny parallax mapping vo viacerých krokoch prechádza eye vektorom E a skúma aktuálnu výšku v každom kroku. Bod P je bod ktorý pozorovateľ vidí a bod I priesečník exe vektora E a skutočného povrchu.

Steep parallax mapping pracuje nasledujúc (obrázok 4.5.2). Vo výškovej mape je reprezentovaná výška hodnotami [0, 1]. Tento interval sa navrstvuje na požadovaný počet vrstiev, čiže počet krokov. Prechádza sa od najvyššej vrstvy až kým výška reprezentovaná výškovou mapou v tomto offsete nie je vyššia. S každou vrstvou sa zväčšuje aj offset. Krok o ktorý sa zväčšuje, závisí od eye vektora, teda od polohy pozorovateľa. Keďže súradnice textúry sú typu vec2, aj krok o ktorý sa offset zväčšuje bude typu vec2:
delta = vec2(-eyeVector.x, eyeVector.y) * hightScale / (eyeVector.z * numSteps), kde **delta** je rozdiel o ktorý sa bude offset zväčšovať, **eyeVector** je eye vektor (vektor pozorovateľa), **hightScale** je virtuálna maximálna výška (čím väčšie číslo, tým bude výškový rozdiel medzi najnižším bodom a najvyšším bodom väčší) a **numSteps** počet vrstiev (počet krokov).



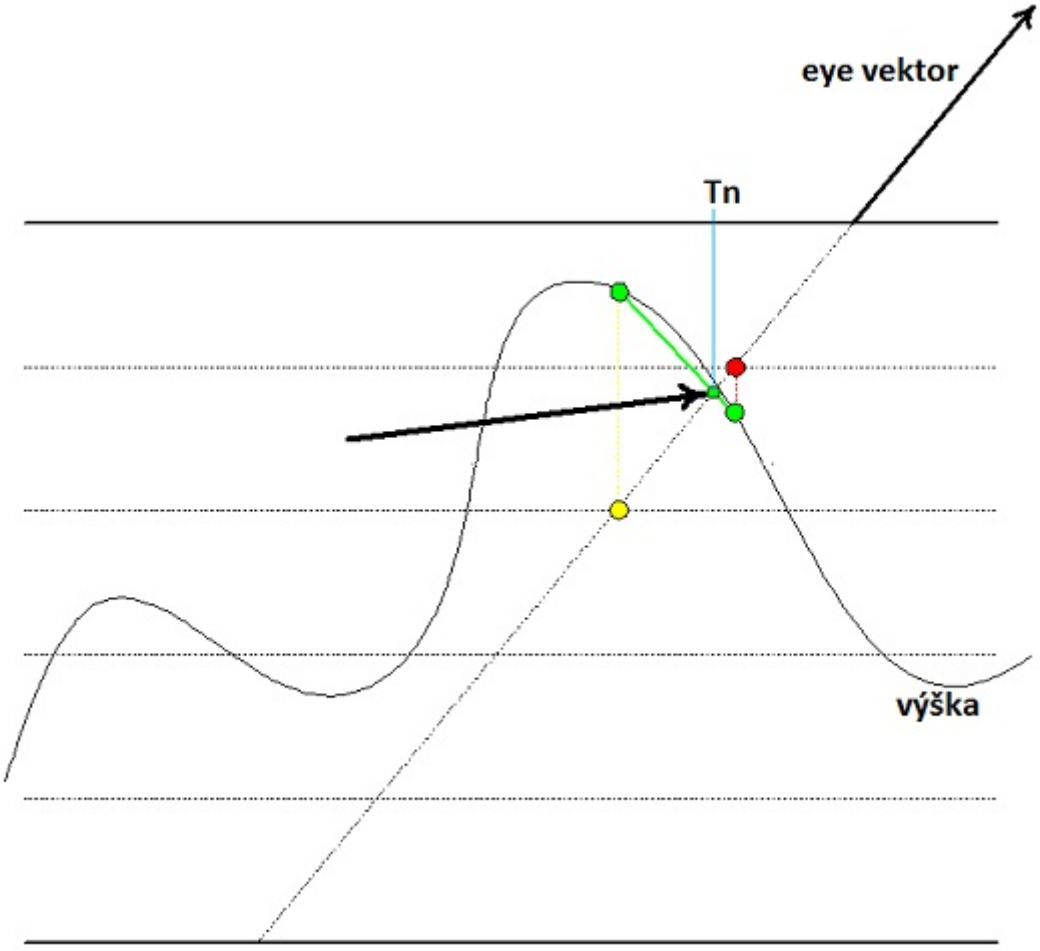
Obrázok 4.5.2: Princíp steep parallax mappingu, kde To je pôvodná textúrovacia súradnica, Tn je nová textúrovacia súradnica, h je výška.

Ako je z obrázka 4.5.2 zrejmé, pri použití malého počtu krokov sa stratí množstvo výškových detailov. Použitím veľkého množstva krokov sa detail zvýši, ale dopad na výkon bude tiež veľký.

Obrázok 4.5.3: Steep parallax mapping v praxi pri použití tridsiatich krokov, na obrázku je badať jemné „vrstvy“. Obrázok pochádza priamo z mojej aplikácie.

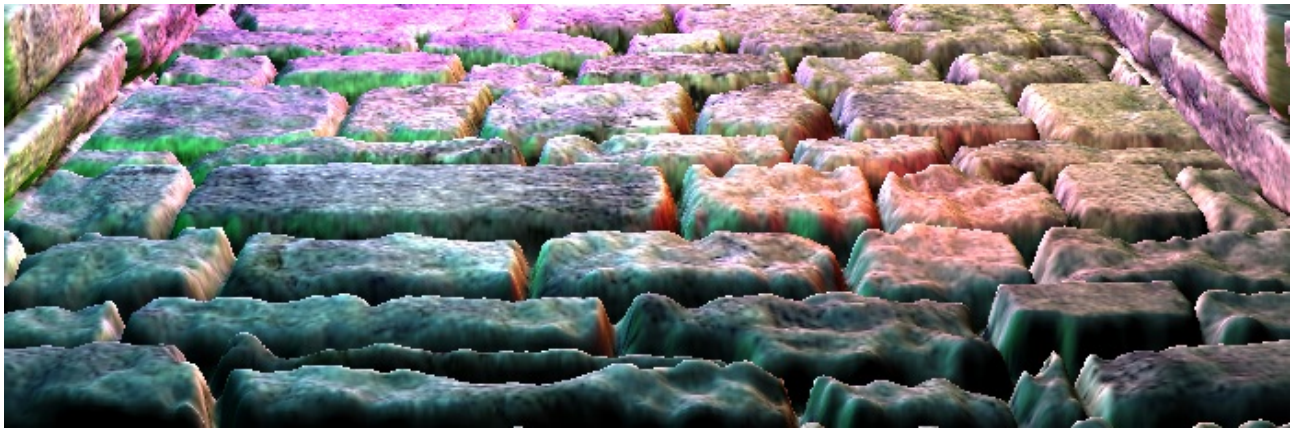
4.6 Parallax occlusion mapping

Parallax occlusion mapping funguje na veľmi podobnom princípe ako predošlá metóda, len k výpočtu konečného offsetu používa výšku nie len v aktuálnom kroku, ale aj v predchádzajúcom. Tieto výšky slúžia k lineárnej interpolácii. Konečný offset sa nájde cez lineárnu interpoláciu offsetu v aktuálnom kroku a predošlom (obrázok 4.6.1).



Obrázok 4.6.1: Princíp parallax occlusion mappingu, kde Tn je nová textúrovacia súradnica.

Táto metóda podáva oproti predošlej kvalitnejší obraz aj už pri použití oveľa menšieho počtu krokov (obrázok 4.6.2). Avšak je tiež možné, že pri použití príliš malého počtu krokov sa niektoré výškové detaily „prekročia“.



Obrázok 4.6.2: Parallax occlusion mapping v praxi. Je vidieť, že „vrstvy“ sú „vyhladené“. Použitý je taký istý počet krokov (30) ako pri steep parallax mappingu, avšak aj pri použití polovičného počtu krokov je obraz stále kvalitnejší. Obrázok pochádza z mojej aplikácie.

4.7 Self shadowing

„Self shadow“ môže byť ľahko pridaný k iteratívnim metódam parallax mappingu. Pri vytváraní tejto práce som sa stretol s dvoma prístupmi.

1. prístup: stopovanie svetelného lúča (obrázok 4.7.1):

Princíp je rovnaký ako pri počítaní novej textúrovacej súradnice, len namiesto eye vektora sa bude stopovať „light vektor“, čo je vektor od osvetleného fragmentu k svetelnému zdroju. Rozdiel sa ešte vyskytne v počítaní výšky a v orientácii light vektora. Ako základ sa vezme minimálna výška, teda 0, a krok sa bude pripočítavať a delta offsetu sa vypočíta nasledujúc:
delta = vec2(lightVector.x, -lightVector.y) * highScale / (lightVector.z * numSteps), kde **delta** je rozdiel o ktorý sa bude offset zväčšovať, **lightVector** je light vektor, **hightScale** je virtuálna maximálna výška a **numSteps** počet vrstiev (počet krokov).

Obrázok 4.7.1: Self shadowing vystopovaním lúča v praxi. Obrázok pochádza priamo z mojej aplikácie.

2. prístup: pomocou výškovej mapy:

Na zafarbenie fragmentu ležiaceho v tieni sa využíva farba z výškovej mapy s príslušným offsetom. Táto metóda nie je vhodná pre určité druhy povrchov a oproti predošlej metóde pristupuje viackrát do textúrovacej pamäte a teda je aj pomalšia. Má ale aj výhodu, keďže pracuje s textúrov nie je problém s posunutím tieňa vzhľadom na offset, teda tieň sa nachádza na správnom mieste, čo nemusí byť pravidlo pri predošlej metóde.

Obrázok 4.7.2: Self shadowing pomocou výškovej mapy. Obrázok pochádza priamo z mojej aplikácie.

Kapitola 5

5 Implementácia

Použitá technológia je OpenGL vo verzii 2.0. Aplikácia bola vyvíjaná na Mobility Radeone HD 3650. Skladá sa z niekoľkých častí, z deklarácie globálnych premenných, následne z implementácie shaderov, z pomocných funkcii, inicializačných funkcii, zobrazovacích funkcii a ovládacích funkcii. Po spustení sa otvorí okno veľkosti 700 x 700 bodov, aby zostali testy porovnateľné je vhodné veľkosť okna nemeniť, lebo pri väčšom rozlíšení je aplikácia samozrejme náročnejšia, pretože shadere sa musia spustiť pre viac fragmentov. Aplikáciu som poňal primárne ako testovaciu aplikáciu, ale aj ako porovnanie implementovaných technológii.

Aplikácia využíva 5 scénok. V každej scénke sa meria počet FPS. Každá scénka využíva niektoré zo spomínaných techník. Pre orientačné zmeranie výkonu postačí vstavaný ukazovateľ FPS. Pre presné zmeranie je potrebné použiť iný špecializovaný nástroj na zaznamenávanie FPS do súboru, pretože aplikácia FPS nezaznamenáva a teda ani následne nevyhodnocuje priemerný výsledok.

V aplikácii je implementovaných viacero shaderov, podľa ich určenia.

5.1 Popis scénok

Scénka 1 používa:

- 8 point lights spolu so spekulárnou zložkou,
- bump mapping,
- parallax occlusion mapping,
- self shadowing vystopovaním lúča,
- v objektoch spolu 22086 polygónov.

Scénka 2 používa:

- 8 point lights,
- bump mapping,
- simple parallax mapping.

Scénka 3 používa:

- 8 point lights
- bump mapping
- steep parallax mapping

Scénka 4 používa:

- 8 point lights
- bump mapping
- parallax occlusion mapping

Scénka 5 Používa:

- 8 point lights
- bump mapping
- parallax occlusion mapping
- self shadowing pomocou výškovej mapy

Scénky sú prepínané pomocou funkcie **switching**.

5.2 Implementácia shaderov

Existujú dve možnosti implementácie shaderov:

1) implementácia v samostatných textových súboroch s príponou reprezentujúcou konkrétny druh shaderu, napríklad .vert pre vertex shader .frag pre fragment shader,
2) implementácia shaderov priamo v zdrojom kóde aplikácie, kde budú shadere uložené v textových poliach.

Rozhodol som sa pre použitie spôsobu 2 z dvoch dôvodov:

1) moja aplikácia nieje natoľko rozsiahla aby bolo nutné držať implementáciu shaderov v samostatných súboroch,
2) takáto implementácia je jednoduchšia, pretože odpadajú problémy spojené s načítavaním zdrojového textu z takýchto súborov.

Príklad uloženia shaderu v poli:

```
const GLchar* shaderSrcVertex[] = {
  "void main()"
  "{"
  "  gl_Position = ftransform();"
  "}"
};
```

Takto implementované shadere sa kompilujú vo funkcii **shaderInit**. Postup kompilácie je nasledovný:

```
vertexSh = glCreateShader(GL_VERTEX_SHADER); // vytvorenie objektu shaderu
fragmentSh = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(vertexSh, 1, shaderSrcVertex, NULL); // priradenie zdrojového kódu
glShaderSource(fragmentSh, 1, shaderSrcFragment, NULL);
glCompileShader(vertexSh); // kompilácia zdrojového kódu
glCompileShader(fragmentSh);
program = glCreateProgram(); // vytvorenie objektu programu
glAttachShader(program, vertexSh); // priradenie skompilovaného kódu do objektu programu
glAttachShader(program, fragmentSh);
glLinkProgram(program); // linkovanie shader programu
```

Skompilovaný shader sa v mieste použitia „zavolá“ pomocou **glUseProgram**, ako parameter sa použije meno shaderu. Po použití na konkrétny objekt je potrebné shader „vypnúť“, tiež pomocou **glUseProgram** s parametrom 0. Naviazanie atribútov a naviazanie premenných typu uniform je potrebné vykonať medzi týmito dvoma volaniami **glUseProgram**:

```
glUseProgram(programOfPodlahy); // volanie shaderu
int texture_location3 = glGetUniformLocation(programOfPodlahy, "myTexture3");
glUniform1i(texture_location3, 2); // naviazanie uniform
  podlaha(); // vykreslim
glUseProgram(0); // vypnutie shaderu
```

Shadere v mojej aplikácii využívajú dve funkcie. Funkciu **main**, kde sa počíta výsledná farba z čiastkových farieb. Čiastková farba pre každé svetlo sa vypočíta vo funkcii **funkcia**. Parametre pre túto funkciu sa pre každý shader podľa potreby mierne odlišujú, ale spravidla sú to

vlastnosti svetiel. Pri niektorých shaderoch aj vektory, ktoré bolo treba transformovať do tangentného priestoru.

Výpočet výslednej farby fragmentu v každom zo shaderov vo funkcii **main** pomocou volania funkcie **funkcia**:

```
for(int i = 0; i < 8; i++)
{
  ciastkova_farba[i] = funkcia(lightVector[i], …, gl_LightSource[i].position.xyz);
  vysledna_farba = vysledna_farba + ciastkova_farba[i];
}
gl_FragColor = vysledna_farba;
```

5.3 Výpis FPS

Pre výpis FPS je potrebné najprv zistiť čas medzi vykreslením dvoch snímkov. Využíva sa funkcia **getTime**, ktorá meria čas od spustenia aplikácie. Čas medzi dvoma snímkami sa vypočíta nasledovne:

```
double pgettime = getTime();
if (cas == -1) {
  cas = pgettime;
  rozdiel = 0.;
} else {
  rozdiel = pgettime - cas;
  cas = pgettime;
}
```

Z takto zisteného rozdielu času, ktorý sa ukladá do premennej `rozdiel` sa počet snímkov za sekundu vypočíta jednoducho, `1/rozdiel`. Vypisuje sa každú štvrť sekundu.

Vypočítaný počet snímkov za sekundu sa zobrazuje pomocou funkcie **renderBitmapString**:

```
if( lastUpdate == -1.0 || pgettime - lastUpdate >= 0.25)
{
  sprintf(FPS,"FPS: %f", 1/rozdiel);
  lastUpdate = pgettime;
}
renderBitmapString(10.0, 15.0, 0.0, FPS);
```

5.4 Implementácia techník pomocou shaderov

Táto podkapitola hovorí o implementácii použitých techník pomocou shaderov.

5.4.1 Textúrovanie pomocou shaderov

Vo vertex shadery sú vygenerované textúrovacie súradnice príkazom:

```
gl_TexCoord[0] = gl_MultiTexCoord0;
```

Potom vo fragment shadery je získa farba z textúry nasledovne:

```
uniform sampler2D tex;
vec4 tex_color = texture2D(tex, gl_TexCoord[0].st);
```

Uniform premenná `tex` musí byť aplikáciou naviazaná:

```
int texture_location = glGetUniformLocation(programOfTeapotTexture, "tex");
glUniform1i(texture_location, 0);
```

Použitá textúra je nastavená aplikáciou nasledovne:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textura_id[TEXTURA_ZEME]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, img.w, img.h, 0, GL_RGB, GL_UNSIGNED_BYTE, img.data);
```

5.4.2 Implementácia point light-u

Ako už bolo povedané, point light je bodové svetlo, ktoré žiary rovnomerne do každej strany a intenzita osvetlenia klesá so vzdialenosťou. Z týchto dôvodov budeme pre výpočet potrebovať „light vektor“ a jeho dĺžku pre výpočet útlmu. Potom sa musí vypočítať skalárny súčin normálového vektoru a light vektoru, aby mohla byť určená intenzita osvetlenia vzhľadom na uhol dopadu lúča. Celkový výpočet je teda nasledovný:

```
vec3 lightVector = lightPosition - vertex_position;
float distance = length(lightVector);
float attenuation = 1.0/(constantAtt + linearAtt * distance + quadraticAtt * distance^2);
float skalarnySucin = max(dot(vertex_normal_normalized, normalize(lightVector)), 0.0);
diffuse_color = gl_FrontMaterial.diffuse * lightDiffuse * skalarnySucin * attenuation;
```

Spekulárna zložka sa počíta iba tam, kde dopadá svetlo. To sa docieli podmienkou pre skalárny súčin normály a light vektoru. Potom sa musí nájsť potrebný „half vektor“, jeho skalárny súčin s normálovým vektorom a nakoniec vypočítať sila odlesku:

```
float specularPower = 0.0;
if(skalarnySucin > 0.0)
{
  vec3 halfVector = normalize(normalize(lightVector) + cameraVector);
  float skalarnyHalf = max(dot(vertex_normal_normalized, halfVector), 0.0);
  specularPower = 0.5 * pow(skalarnyHalf, gl_FrontMaterial.shininess) * attenuation;
}
specular_color = gl_FrontMaterial.specular * lightSpecular * specularPower;
```

Rotácia svetla sa docieli pomocou kosínusu premennej ktorej hodnota sa zvyšuje o pevnú čiastku. Tento kosínus sa zapíše do poľa, ktoré určuje polohu svetla a poloha sa nanovo určí volaním **glLightfv**. Ak sa správne inicializujú dve premenné, pre dve osi, je možné docieliť pohyb svetla po kružnici. Rýchlosť pohybu sa určuje pomocou veľkosti čiastky, ktorá sa pripočítava k premennej z ktorej sa počíta kosínus. Určovanie novej polohy svetla sa vykonáva v hlavnej aplikácii:

```
float x1 = 1.5707963267949, z1 = -3.1415926535898;
cosx1 = cos(x1) * 4; // výpočet kosínusu x-ovej osi
x1 += 0.15;
cosz1 = cos(z1) * 4; // výpočet kosínusu z-ovej osi
z1 += 0.15;
Light1Position[0] = cosx1;
Light1Position[2] = cosz1;
glLoadIdentity();
glLightfv(GL_LIGHT1, GL_POSITION, Light1Position);
```

5.4.3 Implementácia bump mappingu

Pri bump mappingu je najprv potrebné transformovať použité vektory do tangentného priestoru. Vo vertex shadery treba načítať tangentový vektor ako atribút, potom vypočítať vektorový súčin normálového vektoru a tangentového vektoru a zostrojiť TBN maticu:

```
attribute vec3 tangent;
vec3 vertex_normal = normalize(gl_NormalMatrix * gl_Normal);
vec3 tangent_normalized = normalize(gl_NormalMatrix * tangent);
vec3 binormal = cross(vertex_normal, tangent_normalized);
mat3 TBNMatrix = mat3(tangent_normalized, binormal, vertex_normal);
```

Následná transformácia „light vektorov“ pre každé svetlo:

```
for(int i = 0; i < 8; i++)
{
  lightVector[i] = (gl_LightSource[i].position.xyz - vertex_position) * TBNMatrix;
}
```

Vo fragment shadery je teraz potrebné načítať a dostať ich do rozsahu [-1, 1] a použiť na výpočet intenzity svetla daného fragmentu:

```
vec3 normaly_z_mapy = texture2D(normalMap, gl_TexCoord[5].st).rgb * 2.0 - 1.0;
float skalarnySucin = max(dot(normaly_z_mapy, lightVector), 0.0);
```

Nesmie sa zabudnúť na naviazanie atribútovej premennej pre tangentový vektor z aplikácie pre určitý shader program a identifikátor atribútovej premennej:

```
int attrib_location1 = glGetAttribLocation(programOfStien, "tangent");
```

Takto alokovaný atribút sa definuje pre každý vertex medzi volaniami **glBegin** a **glEnd**:

```
glVertexAttrib3f(attrib_location1, 0, 0, -1); glVertex3f(-5.0, 0.0, 5.0);
glVertexAttrib3f(attrib_location1, 0, 0, -1); glVertex3f(-5.0, 10.0, 5.0);
glVertexAttrib3f(attrib_location1, 0, 0, -1); glVertex3f(-5.0, 10.0, -5.0);
glVertexAttrib3f(attrib_location1, 0, 0, -1); glVertex3f(-5.0, 0.0, -5.0);
```

5.4.4 Implementácia jednoduchého parallax mappingu

Pre každý parallax mapping je potrebné vo vertex shadery transformovať do tangentového priestoru pohľadový vektor, čiže „eye vektor“:

```
eyeVector = (camera_vector - vertex_position) * TBNMatrix;
```

Vo fragment shadery sa načíta výška z výškovej mapy (ktorá je uložená vo forme farby). Následne sa výška upraví pomocou „scale faktoru“ a „bias-u“ a potom sa vypočítajú nové textúrovacie súradnice:

```
float height = texture2D(heightMap, gl_TexCoord[5].st).r;
vec2 scaleBias = vec2(0.04, 0.02);
float v = height * scaleBias.r - scaleBias.g;
newCoords = gl_TexCoord[5].st + (v * (normalize(eyeVector).xy));
```

Tieto nové súradnice sa použijú pre normálovú mapu a základnú textúru:

```
normaly_z_mapy = texture2D(normalMap, newCoords).rgb * 2.0 - 1.0;
tex_color = texture2D(texture, newCoords);
```

5.4.5 Implementácia steep parallax mappingu

Keďže steep parallax mapping je iteratívna metóda, bude potrebné určiť počet krokov, dĺžku kroku, počiatočnú výšku a počiatočný offset, ktorý bude základná textúrovacia súradnica. Potom už nasleduje výpočet nových súradníc a ich použitie:

```
const int numSteps = 30;
float height = 1.0;
float step = 1.0 / numSteps;
vec2 offset = gl_TexCoord[5].st;
float h = texture2D(heightMap, offset).r;
vec2 delta = vec2(-eyeVector.x, eyeVector.y) * bump_scale / (eyeVector.z * numSteps);
while(h < height)
{
  height = height - step;
  offset = offset + delta;
  h = texture2D(heightMap, offset).r;
}

normaly_z_mapy = texture2D(normalMap, offset).rgb * 2.0 - 1.0;
tex_color = texture2D(stropColor, offset);
```

5.4.6 Implementácia parallax occlusion mappingu

Implementácia parallax occlusion mappingu je veľmi podobná ako implementácia steep parallax mappingu. Za iteratívny výpočet offsetu, treba vložiť kód pre návrat o krok späť a polohu nového offsetu vypočítať pomocou lineárnej interpolácie medzi týmito dvoma krokmi:

```
// iteratívny výpočet offsetu (steep parallax mapping)

vec2 prev = offset - delta;
float hPrev = texture2D(heightMap, prev).r - (height + step);
float hCur = h - height;
float weight = hCur / (hCur - hPrev);
offset = weight * prev + (1.0 - weight) * offset;

// použitie
```

5.4.7 Self shadowing

Nasledujúce dve podkapitoly budú popisovať implementáciu použitých metód „self shadowingu“.

5.4.7.1 Self shadowing vystopovaním lúča

Princíp je skoro rovnaký ako iteratívne určovanie ofsetu, ale bude sa pracovať s „light vektorom“ a výška sa bude každým krokom zvyšovať, teda sa pôjde od naj spodnejšej výšky. Tiež sa preskočí prvá iterácia aby sa vyhlo niektorým chybám v tieni. Miesto tieňa sa bude hľadať iba tam, kam dopadá svetlo (podmienka so skalárnym súčinom, viď implementáciu point lightu):

```
float selfShadow = 0.0;
if(skalarnySucin > 0)
{
  const int numShadowSteps = 30;
  float step = 1.0 / numShadowSteps;
  Delta = vec2(lightVector.x, -lightVector.y) * bumpScale / (lightVector.z * numShadowSteps);
  vec2 offset = gl_TexCoord[1].st;
  vec4 NB = texture2D(heightMap, offset);
  float height = NB.r + step;
  while(((NB.r < height) && (height < 1)))
  {
    height = height + step;
    offset = offset + LDelta;
    NB = texture2D(heightMap, offset);
  }
  if (NB.r < height)
    {selfShadow = 1.0;}
}
```

Získané miesto tieňa je označené premennou `selfShadow` a ňou sa vynásobí celkový výpočet farby osvetleného fragmentu (z jedného svetla):

```
diffuse_color = ... * lightDiffuse * skalarnySucin * attenuation * selfShadow;
```

5.4.7.2 Self shadowing pomocou výškovej mapy

Pre zafarbenie miesta, kde sa nachádza tieň, sa využíva farba z výškovej mapy:

```
vec2 vLightRayTS = vec2(lightVector.x, -lightVector.y) * bumpScale;
float ShS = 1.0; // shadow softening
float sh0 = texture2D(heightMap, offset.st).r;
float shA = (texture2D(heightMap, offset.st + vLightRayTS * 0.88 * 4.0).r - sh0 - 0.88) * 4.22 * ShS;
float sh9 = (texture2D(heightMap, offset.st + vLightRayTS * 0.77 * 4.0).r - sh0 - 0.77) * 4.33 * ShS;
float sh8 = (texture2D(heightMap, offset.st + vLightRayTS * 0.66 * 4.0).r - sh0 - 0.66) * 4.44 * ShS;
float sh7 = (texture2D(heightMap, offset.st + vLightRayTS * 0.55 * 4.0).r - sh0 - 0.55) * 4.55 * ShS;
float sh6 = (texture2D(heightMap, offset.st + vLightRayTS * 0.44 * 4.0).r - sh0 - 0.44) * 4.66 * ShS;
float sh5 = (texture2D(heightMap, offset.st + vLightRayTS * 0.33 * 4.0).r - sh0 - 0.33) * 4.77 * ShS;
float sh4 = (texture2D(heightMap, offset.st + vLightRayTS * 0.22 * 4.0).r - sh0 - 0.22) * 4.88 * ShS;
fOcclusionShadow = 1.0 - max(max(max(max(max(shA, sh9), sh8), sh7), sh6), sh5), sh4);
fOcclusionShadow = fOcclusionShadow * 0.45;
```

Spôsob výpočtu osvetleného fragmentu je rovnaký ako v predošlej metóde.

Kapitola 6

6 Výsledky

Výsledky boli namerané na viacerých grafických procesoroch pomocou vstavaného ukazovateľa FPS. Aplikácia FPS neukladá a teda ani nevyhodnocuje priemer. Čiže hlavne vysoké hodnoty, kde na výkonných procesoroch hodnoty zvyknú kolísať rádovo v stovkách FPS pre jednoduché testy, sú orientačné.

Nvidia od generácie Geforce 8000 po generáciu Geforce 500 používa rozdielny takt pre jadro GPU a pre shader procesory. Preto je pri kartách týchto generácii okrem GPU clock uvedený aj shader clock.

6.1 Namerané hodnoty na jednotlivých grafických procesoroch

Mobility Radeon HD 3650:
GPU clock: 600 MHz,
Pipelines: 120,
DirectX 10.1, OpenGL 3.2,
Dátum vydania: január 2008

scénka 1: 10.5 FPS
scénka 2: 280 FPS
scénka 3: 75 FPS
scénka 4: 70 FPS
scénka 5: 6.6 FPS

Mobility Radeon HD 5470:
GPU clock: 750 MHz,
Pipelines: 80,
DirectX 11, OpenGL 4.0,
Dátum vydania: január 2010

scénka 1: 13 FPS
scénka 2: 240 FPS
scénka 3: 104 FPS
scénka 4: 100 FPS
scénka 5: 14.5 FPS

Geforce 8600M GT malfunctioning:
GPU clock: 275 MHz (default: 475 MHz),
Shader clock: 550 MHz (default: 950 MHz),
Pipelines: 32,
DirectX 10.0, OpenGL 3.3,
Dátum vydania: máj 2007

scénka 1: 7 FPS
scénka 2: 80 FPS
scénka 3: 47 FPS
scénka 4: 45 FPS
scénka 5: 8 FPS

Geforce 8400M GS:
GPU clock: 400MHz,
Shader clock: 800MHz,
Pipelines: 16,
DirectX 10.0, OpenGL 3.3,
Dátum vydania: máj 2007

scénka 1: 10.2 FPS
scénka 2: 80 FPS
scénka 3: 48 FPS
scénka 4: 46 FPS
scénka 5: 7.1 FPS

Radeon HD 6750:
GPU clock: 700 MHz,
Pipelines: 720,
DirectX 11, OpenGL 4.1,
Dátum vydania: január 2011

scénka 1: 16 FPS
scénka 2: 1200 FPS
scénka 3: 700 FPS
scénka 4: 650 FPS
scénka 5: 80 FPS

Radeon HD 5750:
GPU clock: 700 MHz,
Pipelines: 720,
DirectX 11, OpenGL 4.1,
Dátum vydania: október 2009

scénka 1: 19.7 FPS
scénka 2: 1650 FPS
scénka 3: 720 FPS
scénka 4: 690 FPS
scénka 5: 80 FPS

Radeon HD 2600 XT:
GPU clock: 800 MHz,
Pipelines: 120,
DirectX 10.0, OpenGL 3.1,
Dátum vydania: jún 2007

scénka 1: 5.7 FPS
scénka 2: 206 FPS
scénka 3: 91 FPS
scénka 4: 80 FPS
scénka 5: 8.7 FPS

Geforce GTX 260:
GPU clock: 576 MHz,
Shader clock: 1242 MHz,
Pipelines: 192,
DirectX 10.0, OpenGL 3.3,
Dátum vydania: jún 2008

scénka 1: 18 FPS
scénka 2: 1700 FPS
scénka 3: 1200 FPS
scénka 4: 1150 FPS
scénka 5: 140 FPS

Quadro FX 380:
GPU clock: 450 MHz,
Pipelines: 16,
DirectX 10.0, OpenGL 3.0,

scénka 1: 11 FPS
scénka 2: 150 FPS
scénka 3: 88 FPS
scénka 4: 84 FPS
scénka 5: 9.6 FPS

Geforce 8600 GTS:
GPU clock: 675 MHz,
Shader clock: 1450 MHz,
Pipelines: 32,
DirectX 10.0, OpenGL 3.3,
Dátum vydania: november 2006

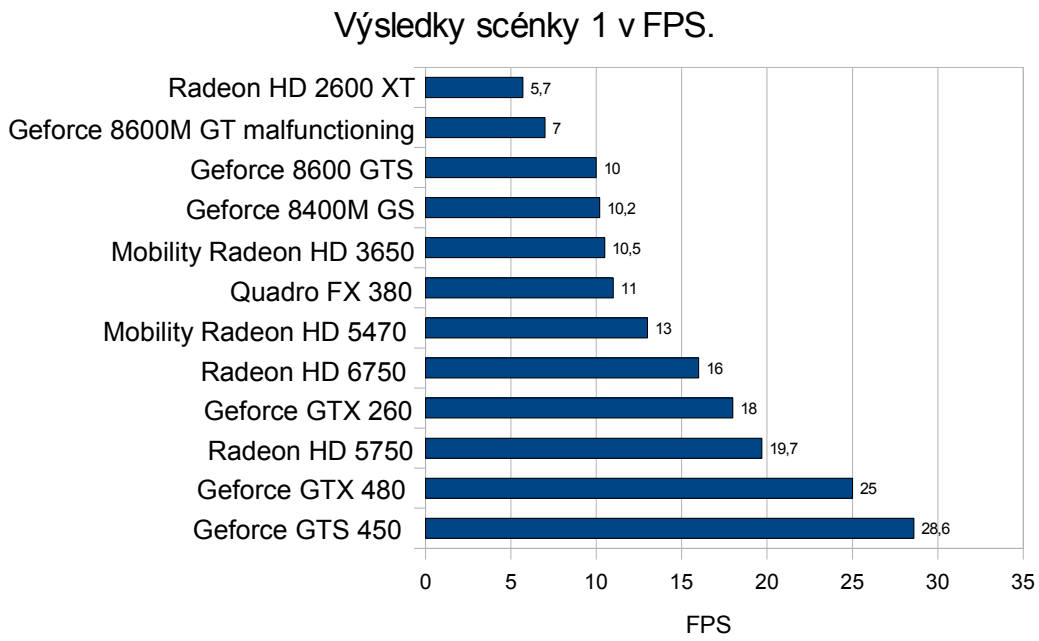
scénka 1: 10 FPS
scénka 2: 300 FPS
scénka 3: 200 FPS
scénka 4: 180 FPS
scénka 5: 27 FPS

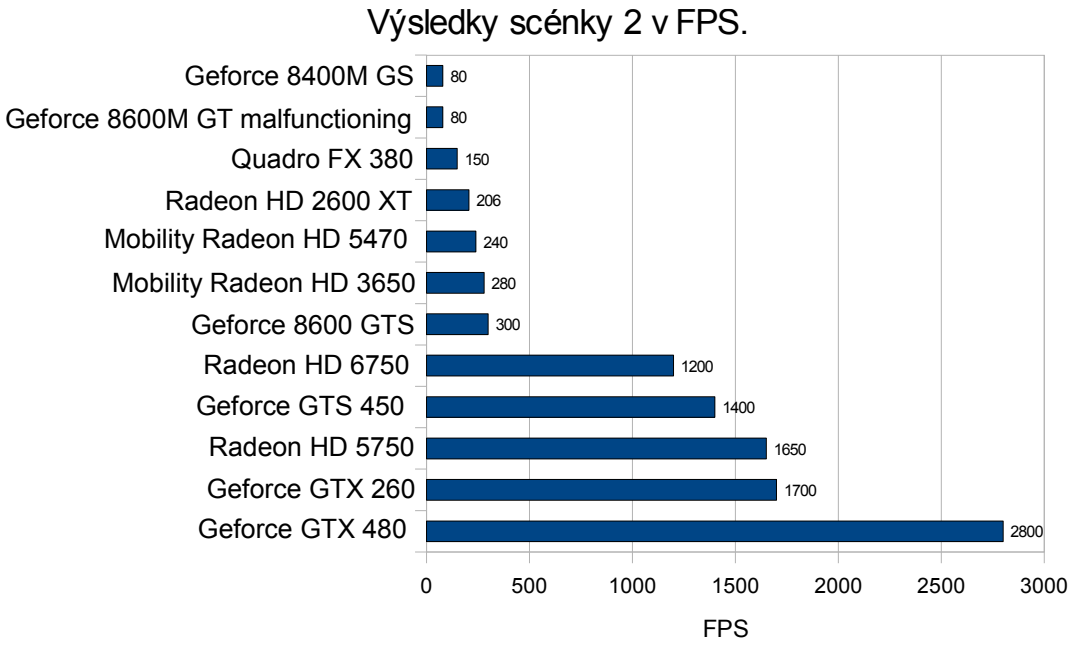
Geforce GTS 450:
GPU clock: 783 MHz,
Shader clock: 1566 MHz.
Pipelines: 192,
DirectX 11, OpenGL 4.2,
Dátum vydania: september 2010

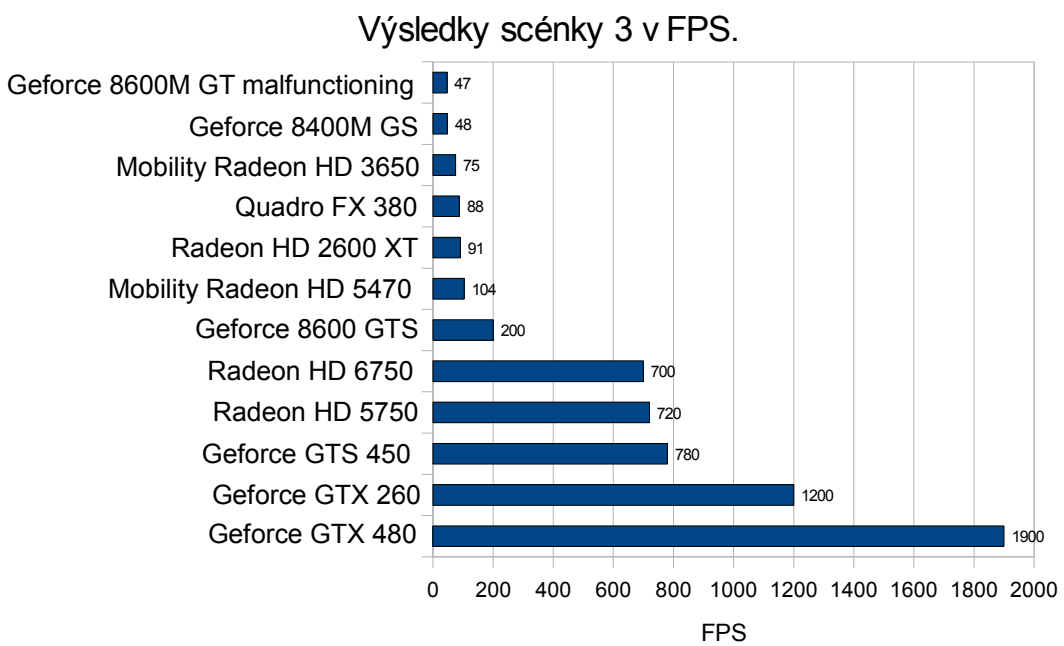
scénka 1: 28.6 FPS
scénka 2: 1400 FPS
scénka 3: 780 FPS
scénka 4: 750 FPS
scénka 5: 100 FPS

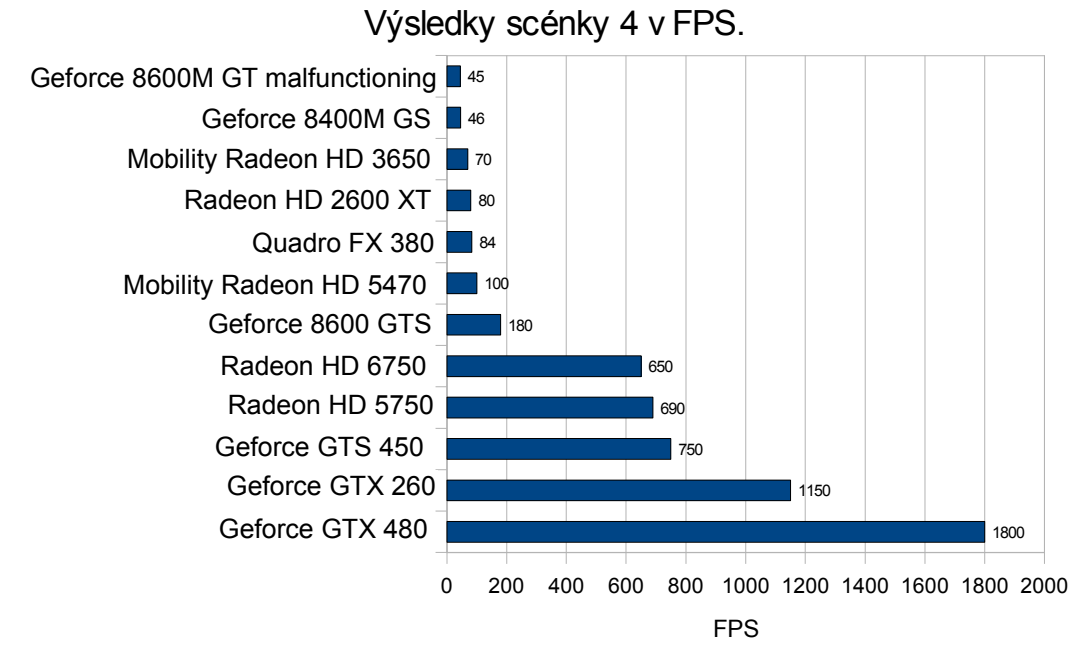
Geforce GTX 480:
GPU clock: 700 MHz,
Shader clock: 1401 MHz,
Pipelines: 480,
DirectX 11, OpenGL 4.2,
Dátum vydania: marec 2010

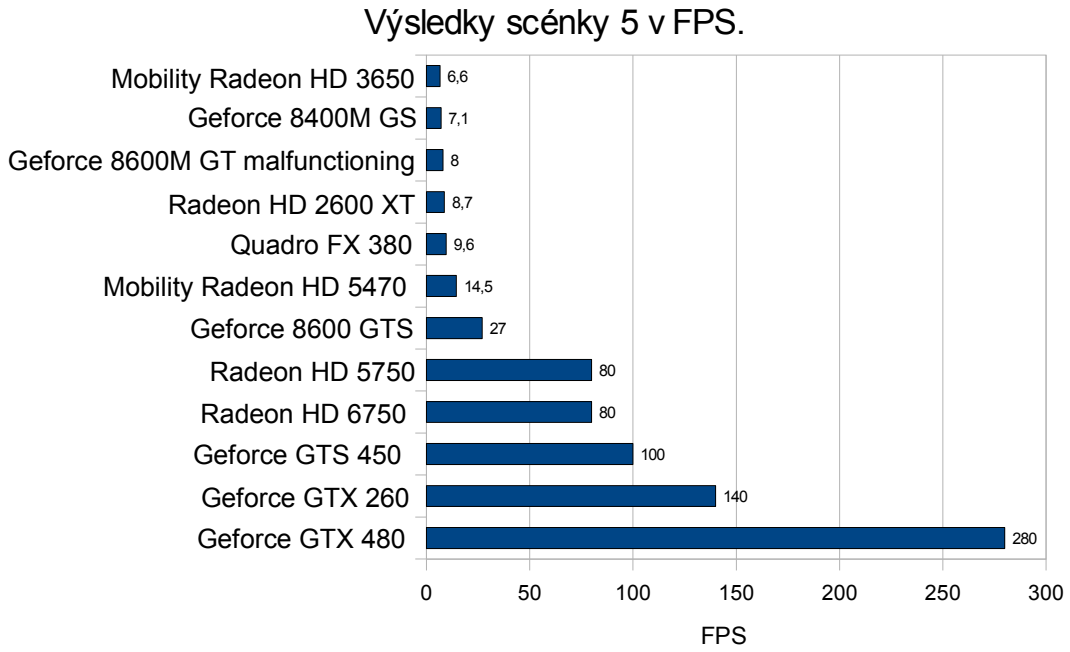
scénka 1: 25 FPS
scénka 2: 2800 FPS
scénka 3: 1900 FPS
scénka 4: 1800 FPS
scénka 5: 280 FPS


Výsledky scénky 1 v FPS.
Radeon HD 2600 XT
Geforce 8600M GT malfunctioning
Geforce 8600 GTS
Geforce 8400M GS
Mobility Radeon HD 3650
Quadro FX 380
Mobility Radeon HD 5470
Radeon HD 6750
Geforce GTX 260
Radeon HD 5750
Geforce GTX 480
Geforce GTS 450
5,7
7
10
10,2
10,5
11
13
16
18
19,7
25
28,6
0
5
10
15
20
25
30
35
FPS


Výsledky scénky 2 v FPS.
Geforce 8400M GS
Geforce 8600M GT malfunctioning
Quadro FX 380
Radeon HD 2600 XT
Mobility Radeon HD 5470
Mobility Radeon HD 3650
Geforce 8600 GTS
Radeon HD 6750
Geforce GTS 450
Radeon HD 5750
Geforce GTX 260
Geforce GTX 480
80
80
150
206
240
280
300
1200
1400
1650
1700
2800
0
500
1000
1500
2000
2500
3000
FPS


Výsledky scénky 3 v FPS.
Geforce 8600M GT malfunctioning
Geforce 8400M GS
Mobility Radeon HD 3650
Quadro FX 380
Radeon HD 2600 XT
Mobility Radeon HD 5470
Geforce 8600 GTS
Radeon HD 6750
Radeon HD 5750
Geforce GTS 450
Geforce GTX 260
Geforce GTX 480
47
48
75
88
91
104
200
700
720
780
1200
1900
0
200
400
600
800
1000
1200
1400
1600
1800
2000
FPS


Výsledky scénky 4 v FPS.
Geforce 8600M GT malfunctioning
Geforce 8400M GS
Mobility Radeon HD 3650
Radeon HD 2600 XT
Quadro FX 380
Mobility Radeon HD 5470
Geforce 8600 GTS
Radeon HD 6750
Radeon HD 5750
Geforce GTS 450
Geforce GTX 260
Geforce GTX 480
45
46
70
80
84
100
180
650
690
750
1150
1800
0 200 400 600 800 1000 1200 1400 1600 1800 2000
FPS


Výsledky scénky 5 v FPS.
Mobility Radeon HD 3650
Geforce 8400M GS
Geforce 8600M GT malfunctioning
Radeon HD 2600 XT
Quadro FX 380
Mobility Radeon HD 5470
Geforce 8600 GTS
Radeon HD 5750
Radeon HD 6750
Geforce GTS 450
Geforce GTX 260
Geforce GTX 480
6,6
7,1
8
8,7
9,6
14,5
27
80
80
100
140
280
0 50 100 150 200 250 300
FPS

Pokračovanie a možné rozšírenia

Aplikáciu je možné rozšíriť hlavne o ukladanie hodnôt FPS napríklad každú sekundu a následne vyhodnotiť výsledok. Aplikácia by sa spustila prepínačom `-test`, následne by sa spustila prvá scénka, FPS by sa ukladalo po dobu 30 sekúnd. Po uplynutí tejto doby by sa spustila druhá scénka a a takto až scénka posledná. Po skončení testovania by sa spočítala priemerná hodnota FPS pre každú scénku a zobrazila by sa v novom okne.

Pri dopisovaní tejto práce a pri testovaní grafických procesorov sa ukázal problém, že najmä pre grafické procesory starších generácii (Geforce 6100 a Radeon x1650 Pro) sa nezkompilujú shadere, ktoré majú na starosti nejaký druh parallax mappingu. Tento problém môže byť spôsobený napríklad staršími ovládačmi. S týmto ma napadlo, že by bolo vhodné overiť či kompilácia a linkovanie shaderov prebehlo v poriadku a keď nie, tak používateľa varovať. Ale keďže moja aplikácie je v hotovom stave, kde sa konzolové okno nezobrazuje, rozhodol som sa toto nemeniť.

K porovnaniu pridať ďalšiu techniku ktorá využíva výškovú mapu a to „displacement mapping“.

Ďalšie možné rozšírenie je pridanie tieňov pomocou „shadow mappingu“, tiež aj pomocou „shadow volumes“.

Záver

Cieľom tejto aplikácie bolo zoznámiť sa s programovaním počítačovej grafiky pomocou shaderov. Keďže v čase, keď som začínal pracovať na svojej aplikácii som bol v tejto oblasti úplný začiatočník a prakticky som o programovaní shaderov nevedel nič, tak bolo miestami náročné pochopiť niektoré princípy. Každým krokom dopredu, som sa stretával s niečím novým. Od pochopenia kompilácie shaderu a vykreslenie prvého jednofarebného trojuholníka pomocou shadera, cez pochopenie interakcie aplikácie a shadera, interakcie medzi vertex shaderom a fragment shaderom až po konečnú podobu aplikácie.

V prvom bode práce som sa pustil do štúdia prekladu shaderu a jazyka GLSL. Musel som si osvojiť vektorové myslenie. Tiež bolo treba vyriešiť problém s knižnicou GLEW.

V druhom bode som sa pustil do štúdia a postupnej implementácie rôznych techník. Napredoval som postupne, najprv som rozširoval jeden shader, ktorý som používal pre vykresľovanie vstavaného čajníku **glutSolidTeapot**. Pridal som postupne textúrovanie a direkcionálne osvetlenie a neskôr rotáciu tohto svetla. Ako sa scénka stávala zložitejšou, bolo treba tento shader rozdeliť na viacero, pretože na rôzne objekty bolo treba použiť rôzne techniky. Pri implementácii bump mappingu bolo treba vyriešiť problém multitexturingu (nefungovalo zo začiatku správne), predávania vertexových atribútov a implementáciu bodového svetla. Ďalej bolo treba vyriešiť zobrazovanie FPS priamo v scénke.

Na rad prišla orientácia mojej testovacej aplikácie. Nápadov bolo niekoľko, ale pri implementácii parallax occlusion mappingu, som sa rozhodol použiť tieto techniky pre testovanie a tiež ich zároveň aj porovnať.

V závere som dal do povedomia najzaujímavejšie rozšírenia, ktoré sa mi nepodarilo implementovať, som nemal dostatočné prostriedky k ich realizácii, alebo ma napadli príliš neskoro.

Literatúra

[1] DAVIS, Tom, Dave SHREINER, Mason WOO a Jackie NEIDER. *OpenGL*: *Průvodce programátora*. Praha: COMPUTER PRESS, 2006. ISBN 8025112756.

[2] ROST, Randi. *OpenGL*: *Shading Language*. Second Edition. : Addison Wesley Professional, 2006. ISBN 978-0-321-33489-3.

[3] RODRIGUEZ VILLAR, Jacobo. OpenGL Shading Language Course: Introduction to GLSL. [online]. [cit. 2012-05-06]. Dostupné z: http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/Chapter_1.pdf

[4] RODRIGUEZ VILLAR, Jacobo. OpenGL Shading Language Course: GLSL Basics. [online]. [cit. 2012-05-06]. Dostupné z: http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/Chapter_2.pdf

[5] RODRIGUEZ VILLAR, Jacobo. OpenGL Shading Language Course: Basic Shaders. [online]. [cit. 2012-05-06]. Dostupné z: http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/Chapter_3.pdf

[6] RODRIGUEZ VILLAR, Jacobo. OpenGL Shading Language Course: Advanced Shaders. [online]. [cit. 2012-05-06]. Dostupné z: http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/Chapter_4.pdf

[7] RODRIGUEZ VILLAR, Jacobo. OpenGL Shading Language Course: Appendix. [online]. [cit. 2012-05-06]. Dostupné z: http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/Chapter_5.pdf

[8] SZIRMAY-KALOS, László a Tamás UMENHOFFER. Displacement Mapping on the GPU: State of the Art. [online]. 2006 [cit. 2012-05-06]. Dostupné z: http://sirkan.iit.bme.hu/~szirmay/egdisfinal3.pdf

[9] TATARCHUK, Natalya. Practical Parallax Occlusion Mapping with Approximate Soft Shadows for Detailed Surface Rendering. *Advanced Real-Time Rendering in 3D Graphics and Games* [online]. 2006 [cit. 2012-05-06]. Dostupné z: http://developer.amd.com/media/gpu_assets/Course_26_SIGGRAPH_2006.pdf

[10] WELSH, Terry. Parallax Mapping with Offset Limiting: A PerPixel Approximation of Uneven Surfaces. [online]. 2004 [cit. 2012-05-06]. Dostupné z: http://www8.cs.umu.se/kurser/5DV051/VT09/lab/parallax_mapping.pdf

[11] MCGUIRE, Morgan a Max MCGUIRE. Steep Parallax Mapping. [online]. 2005 [cit. 2012-05-06]. Dostupné z: http://graphics.cs.brown.edu/games/SteepParallax/mcguire-steepparallax.pdf

[12] Graphics processing unit. In: *Wikipedia*: *the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2012-05-06]. Dostupné z: http://en.wikipedia.org/wiki/Graphics_processing_unit

[13] Comparison of Nvidia graphics processing units. In: *Wikipedia*: *the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2012-05-06]. Dostupné z: http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units

[14] Comparison of AMD graphics processing units. In: *Wikipedia*: *the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2012-05-06]. Dostupné z: http://en.wikipedia.org/wiki/Comparison_of_AMD_graphics_processing_units

Zoznam príloh

Príloha A: Ukážky z aplikácie

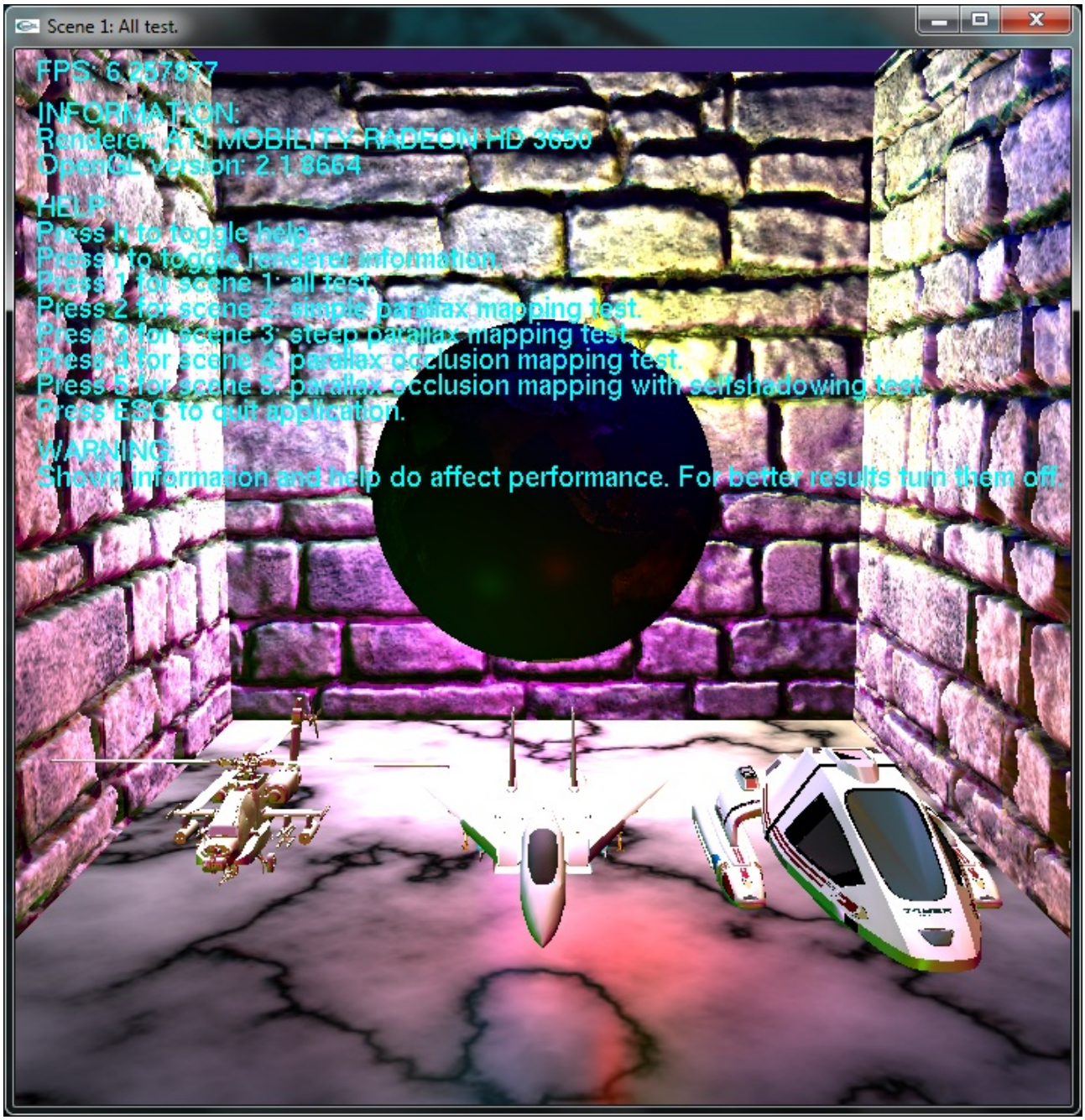
Obsah CD:
- adresár „Binaries“ obsahuje:
  pod adresár „Windows 7“: spustiteľný súbor pre Windows 7
  pod adresár „Windows XP“: spustiteľný súbor pre Windows XP

- adresár „Source“ obsahuje:
  zdrojové kódy aplikácie a súbor Makefile

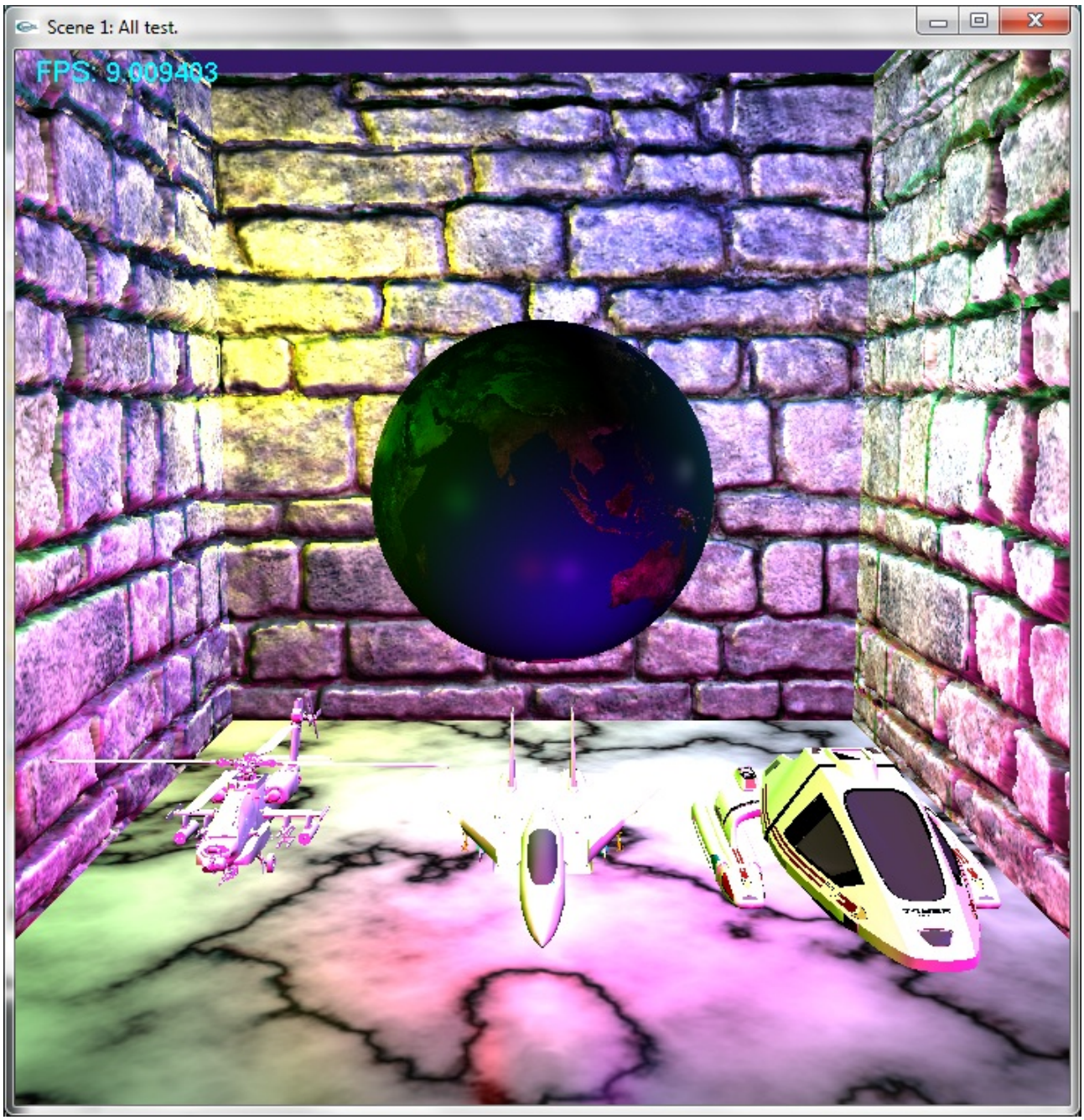
- adresár „Thesis“ obsahuje:
  pod adresár „.pdf“: bakalárska práca vo formáte pdf
  pod adresár „Open Office“: bakalárska práca vo formáte odt.
  pod adresár „Used Images“: použité obrázky v bakalárskej práci

Príloha A

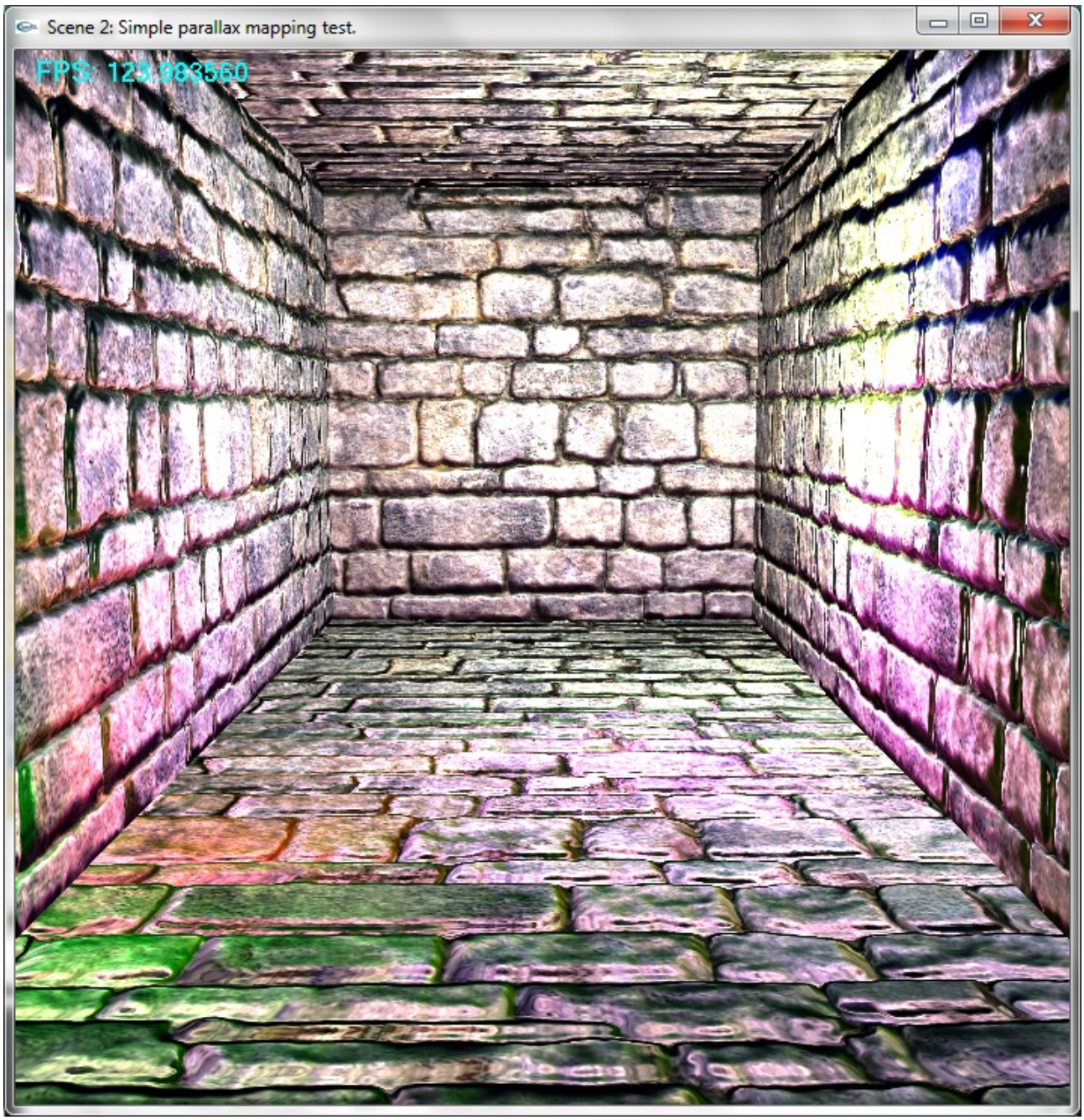
Ukážky z aplikácie.



Scénka 1 so zobrazenou nápovedou a informáciami.

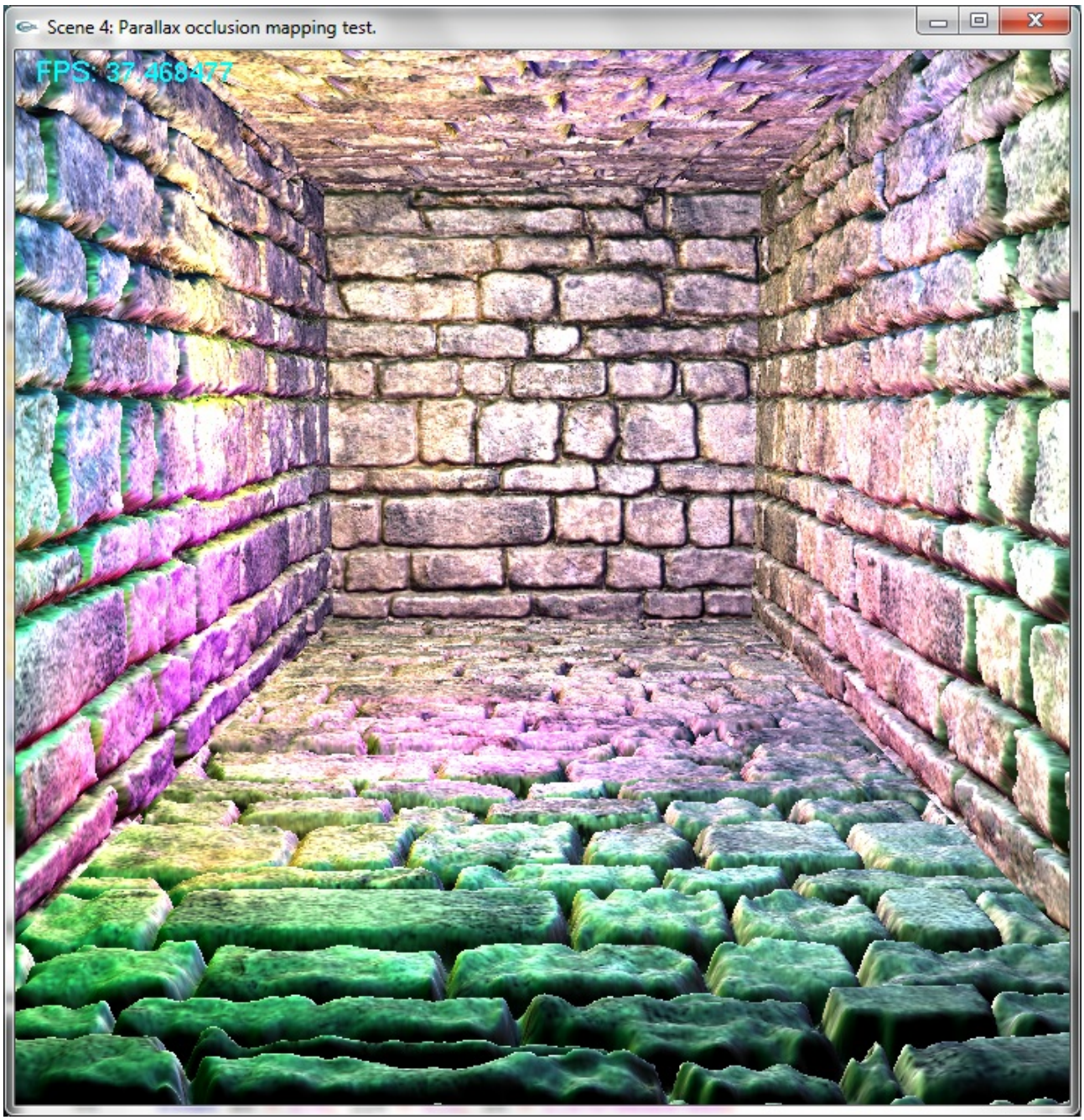


Scénka 1 bez nápovedy a informácii.



Scénka 2.

Scénka 3.



Scénka 4.


Scene 5: Parallax occlusion mapping with selfshadowing test.
FPS: 3.367889

Scénka 5.