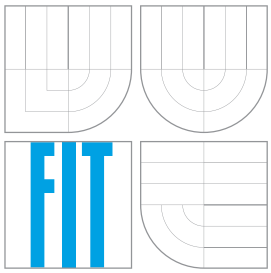
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

EFEKTIVNÍ VÝPOČET OSVĚTLENÍ

EFFICIENT COMPUTATION OF LIGHTING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — TOMÁŠ KUBOVČÍK
AUTHOR

VEDOUCÍ PRÁCE — Ing. TOMÁŠ MILET
SUPERVISOR

BRNO 2014

Abstrakt

Tato práce se zabývá efektivním výpočtem osvětlení v grafických scénách s velkým počtem světel. Představeny budou základní techniky výpočtu osvětlení i pokročilé techniky, které jsou z nich odvozeny. Těžištěm práce je zejména technika tiled shading a její optimalizace popsané v teoretické i praktické části. Závěr práce se zaměřuje na experimenty prováděné na těchto technikách zachycující jejich přínos v efektivitě a rychlosti výpočtů osvětlení jakož i implementačním podrobnostem některých důležitých částí.

Abstract

This bachelor's thesis deals with efficient computation of lighting in graphics scenes containing many light sources. Basic lighting calculation techniques as well as techniques derived from them will be introduced. The main goal of this thesis is to investigate tiled shading and its optimalizations described more in detail in both theoretical and practical sections. Concluding part consists of experiments performed on this techniques measuring their efficiency as well as implementation details of some interesting or important parts of them.

Klíčová slova

OpenGL, GLSL, Phongův osvětlovací model, atenuace, shader, stínování, osvětlení, odložené stínování, forward shading, tiled shading, clustered shading, snižování počtu vzorků, optimalizace hloubkového bufferu, vyřazování světel, přiřazování světel, mřížka světel, hraniční obdélník, komolý jehlan, intenzita světla, RGB barevný model, úhel dopadu, odrazivost světla, Fresnelovy rovnice

Keywords

OpenGL, GLSL, Phong reflection model, attenuation, shader, shading, lighting, deferred shading, forward shading, tiled shading, clustered shading, downsampling, depth optimization, light culling, light insertion, lightgrid, bounding quad, frustum, light intensity, RGB color model, angle of incidence, light reflectivity, Fresnel's equations.

Citace

Tomáš Kubovčík: Efektivní výpočet osvětlení, bakalářská práce, Brno, FIT VUT v Brně, 2014

Efektivní výpočet osvětlení

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Kubovčík
13. mája 2014

Poděkování

Na tomto místě bych rád poděkoval vedoucímu práce panu Ing. Tomáši Miletovi za jeho podporu, trpělivost a originální nápady při tvorbě práce.

© Tomáš Kubovčík, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Výpočet osvetlenia v grafických scénach je kľúčovou metódou vo vykresľovacích aplikáciách, ktoré sa snažia čo najpresnejšie interpretovať realitu. Presné výpočty si však vyžadujú vysoké nároky na výpočetný výkon rovnako ako aj podrobnú znalosť tejto problematiky. Preto sa tento proces značne zjednodušuje a na výpočet osvetlenia sa využívajú rôzne aproximácie (Phongov osvetľovací model 3.1). Pri základných technikách výpočtu osvetlenia často dochádza k zbytočným a redundantným kalkuláciám, kvôli ktorým sa so zvyšujúcim počtom zdrojov svetiel efektivita a výkonnosť týchto techník rapídne znižuje a ich použiteľnosť v súčasných grafických aplikáciách upadá.

Táto práca rozoberá základné techniky výpočtu osvetlenia uvedené v kapitole 3, ktoré sa využívajú v jednoduchých aplikáciách pri malých počtoch zdrojov svetiel, konkrétne **forward shading** a **deferred shading** ako aj techniky pre komplexnejšie grafické scény s tisíckami svetiel (**tiled shading** (kapitola 4), **clustered shading** (kapitola 5)). Prevažná časť práce sa zameriava najmä na techniku tiled shading a jej optimalizácie, ktorú je možné využiť v kombinácii s oboma základnými osvetľovacími technikami. Jej cieľom je zredukovať potrebné výpočty, prípadne ich vhodne paralelizovať aby sa celý proces urýchlil.

V závere práce sú uvedené experimenty, ktoré testujú osvetľovacie techniky s použitím rôznych parametrov priamo ovyplyvňujúcich náročnosť výpočtov.

Kapitola 2

Svetlo v počítačovej grafike

Cieľom tejto kapitoly je uviesť čitateľa do problematiky výpočtu osvetlenia v počítačovej grafike ako aj oboznámiť ho so základnými pojmami, ktoré s ňou úzko súvisia.

V dostupnej literatúre sa často používajú pojmy lighting/illumination a shading, ktoré sú si veľmi blízke, no líšia sa v drobnom detaile, preto treba uviesť ich význam. *Lighting*, slovensky osvetlenie, popisuje proces určenia farby a intenzity svetla, ktoré **dopadá** na povrch telesa. Narozdiel od toho pojem *shading* reprezentuje metódy použité k určeniu farby a intenzity svetla **odrazeného** smerom k pozorovateľovi pre každý pixel na povrchu. Pojem osvetlenie v tomto texte predstavuje oba tieto procesy.

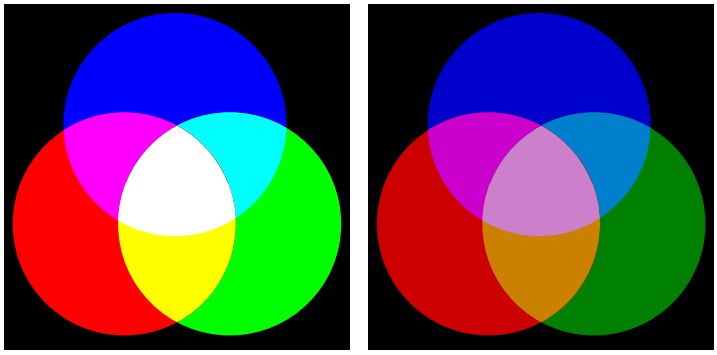
2.1 Intenzita svetla

Fyzikálne korektný výpočet odrazu svetla od povrchu telesa musí počítať so všetkými vlnovými dĺžkami svetla vo viditeľnom spektre. Ľudské oko je však najcitlivejšie na tri prekrývajúce sa oblasti viditeľného spektra, ktoré odpovedajú červenej, zelenej a modrej farbe. Práve preto sa v počítačovej grafike, prípadne v TV príjmačoch pre reprezentáciu farieb a ich intenzít často využíva *RGB* model, prípadne jeho modifikácie (v súčasnosti monitory využívajú prevažne *sRGB* model). Tento model tvoria tri zložky, ktoré predstavujú práve spomínané farby (červená, zelená, modrá) pričom každú zložku môžeme reprezentovať hodnotou z rozsahu $\langle 0, 1\rangle$, $\langle 0, 255\rangle$ prípadne v hexadecimálnom tvare $\langle 0, FF\rangle$. Tieto hodnoty udávajú spektrálnu kompozíciu svetla, teda farbu ktorú vníma ľudské oko rovnako ako aj *intenzitu* danej farebnej zložky, pričom maximálnu hodnotu reprezentuje najväčšie číslo z uvedených intervalov.

Už v stredoškolskej fyzike sa dozvedáme, že biele svetlo obsahuje všetky farby, všetkých vlnových dĺžok. V RGB modely je táto skutočnosť reprezentovaná miešaním maximálnych intenzít primárnych farebných zložiek, ktoré produkuje práve biele svetlo. Aditívnym miešaním jednotlivých zložiek (viď obrázok 2.1a) môžeme z primárnych farieb tvoriť nové farby, rovnako ako menením intenzít jednotlivých zložiek (viď obrázok 2.1b). Výsledná reprezentácia farby v RGB modely je teda trojica intenzít jednotlivých farebných zložiek.

2.2 Absorbcia a odraz svetla

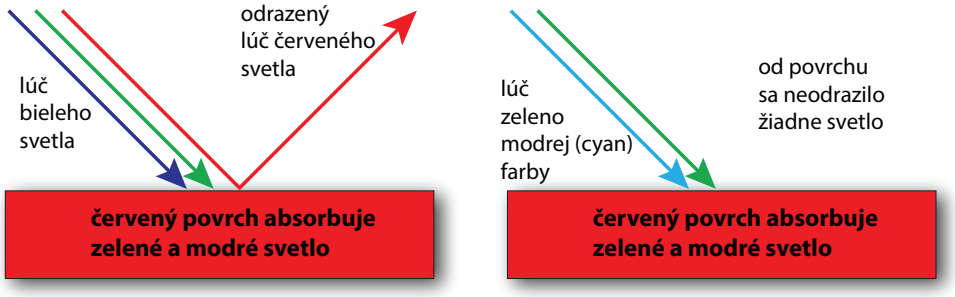
Povedzme, že sa pozeráme na auto, ktoré je červenej farby. Slnko vyšle lúč bieleho svetla, ktoré sa odrazí od povrchu auta priamo do oka pozorovateľa. Ako už bolo spomínané biele svetlo podľa korektného fyzikálneho modelu obsahuje všetky farby, no aj napriek tomu



(a) vznik bieleho svetla (b) zmena intenzity farieb

Obr. 2.1: Aditívne miešanie farieb

odrazený lúč obsahuje iba červené svetlo a preto vidíme červené auto a nie biele. Všetky ostatné farby pohltil povrch auta a iba červené svetlo sa odrazilo (viď obrázok 2.2a). Ak však nasvietime povrch tohoto auta svetlo modrým svetlom zdalo by sa byť čierne pretože by absorbovalo 100% svetla (viď obrázok 2.2b).



(a) odraz bieleho svetla (b) odraz svetlo modrého svetla

Obr. 2.2: Absorbcia a odraz svetla

Pri výpočte osvetlenia sa však často korektná absorbcia svetla zanedbáva. Narozdiel od absorbcie svetla má v tomto procese odraz svetla dôležitú úlohu, pretože zvyšuje výslednému osvetleniu mieru realistickosti.

2.3 Uhol dopadu

Uhol pod ktorým lúče svetla dopadajú na povrch priamo ovplyvňuje svetlosť povrchu a vo Phongovom osvetľovacom modele je základným prvkom difúznej zložky svetla. V prípade, že je povrch telesa kolmý na lúče svetla naň dopadajúce (uhol dopadu je 0°) je povrch v tomto bode maximálne svetlý. Ak je lúč svetla rovnobežný s povrchom telesa (uhol dopadu je 90°) svetlo ho neovplyvňuje. Môže nastať prípad, že uhol dopadu je väčší ako 90° kedy svetlo dopadá na zadnú stenu telesa a predná strana nie je vôbec osvetlená. Pre maximálnu svetlosť platí vzťah:

$$\cos(uhol\ dopadu) = svetlost \tag{2.1}$$

2.4 Zdroje svetla

Existuje niekoľko typov zdrojov svetla, ktoré priamo ovplyvňujú spôsob výpočtu osvetlenia. Môžu sa totiž líšiť v smere ktorým vyžarujú, dosahom prípadne spôsobom straty intenzity v závislosti od vzdialenosti (*atenuáciou*). Štandardnými zdrojmi svetla v 3D grafike sú:

1. ambientné svetlo (*ambient light*)
2. smerové svetlo (*directional light*)
3. bodové svetlo (*point light*)
4. svetlomet/reflektor (*spot light*)

2.4.1 Ambientné svetlo

Ambientné svetlo (obrázok 2.3a) predstavuje zdroj svetla, ktoré je konštantné pre všetky povrchy. Bližšie bude popísané ako súčasť Phongovho osvetľovacieho modelu.

2.4.2 Smerové svetlo

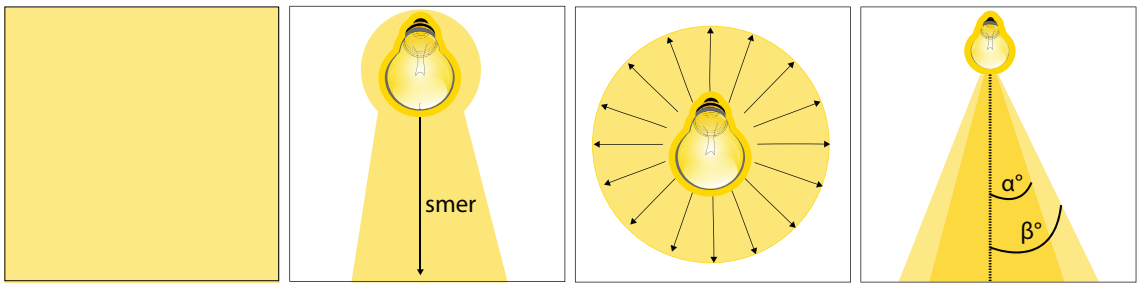
Smerové svetlo (2.3b), taktiež známe ako *nekonečný zdroj svetla* vyžaruje svetlo v jednom smere z nekonečnej vzdialenosti. *Directional lights* sa typicky používajú na modelovanie zdrojov svetiel ako napríklad slnko, ktorého lúče môžme považovať za paralelné. Keďže nemajú pozíciu v priestore majú nekonečne veľký dosah a nestrácajú na intenzite na základe vzdialenosti.

2.4.3 Bodové svetlo

Bodové svetlá (obrázok 2.3c) sú definované farbou a pozíciou. Nedefinuje ich však žiaden smer, ktorým by vyžarovali, pretože vyžarujú svetlo do všetkých smerov rovnako, pričom intenzita klesá so zväčšujúcou sa vzdialenosťou od pozície zdroja. Veľmi dobrým príkladom bodového svetla je napríklad žiarovka.

2.4.4 Reflektor

Posledným základným typom zdrojov svetiel je reflektor (obrázok 2.3d), resp. svetlomet, ktorý je veľmi podobný bodovému svetlu. Líši sa však v tom, že má definovaný smer vyžarovania. Taktiež stráca na intenzite na základe vzdialenosti od pozície z ktorej vyžaruje, ale aj vzdialenosti od miesta dopadu. Dosah svetla v konečnom dôsledku reprezentujú dva kužele, pričom vnútorný kužel reprezentuje výraznejšiu časť svetla a vonkajší tu rozptýlenú.



(a) ambientné svetlo (b) smerové svetlo (c) bodové svetlo (d) reflektorové svetlo

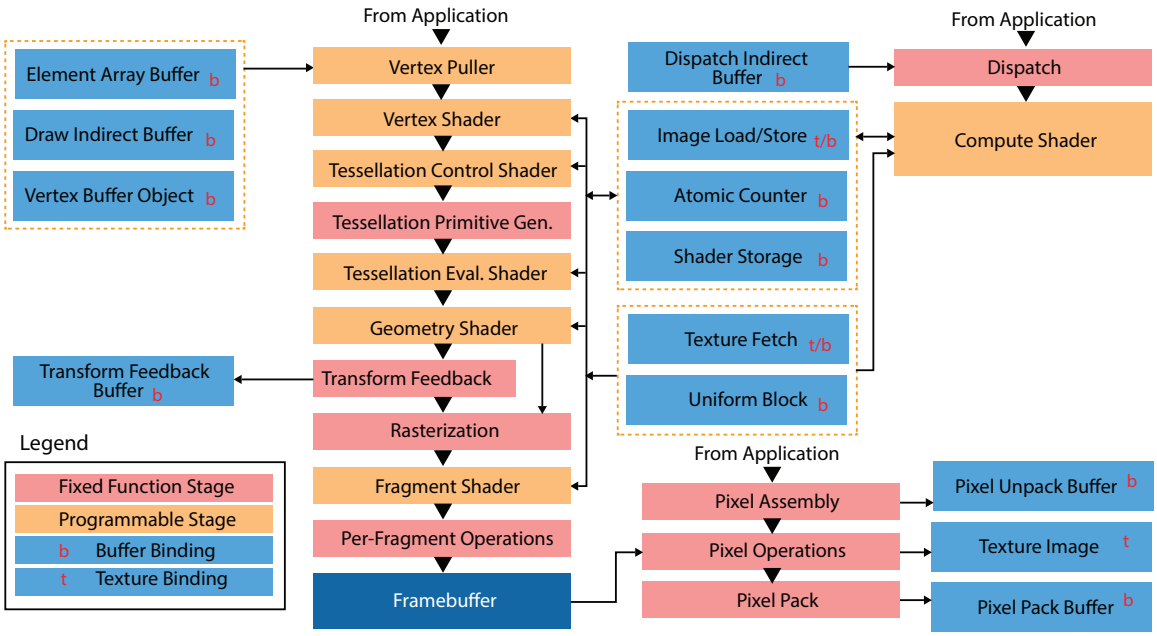
Obr. 2.3: Zdroje svetiel

Kapitola 3

Základné techniky výpočtu osvetlenia v grafických scénach

Pre lepšie pochopenie problematiky osvetľovania grafických scén bude na začiatku tejto kapitoly popísaný stručný úvod do *moderného (programovateľného) grafického pipelinu*. Ďalej sa bude zaoberať Phongovým osvetľovacím modelom a základnými technikami výpočtu osvetlenia, budú popísané ich výhody, nevýhody a optimalizácie.

Pred niekoľkými rokmi nám grafické adaptéry pre vykreslenie 3D objektov umožňovali iba nemenný postup krokov, ktorý sa označuje ako *pevný grafický pipeline*. Programátor teda nemohol ovplyvniť akým spôsobom sa bude ktorý pixel vykresľovať, prípadne upraviť vrchol ktorý už bol nahraný do pamäti grafickej karty. Vývojom grafického hardvéru sa však ukázalo, že tento prístup nie je efektívny a preto vznikol programovateľný grafický pipeline (viď obrázok 3.1), ktorý je ovládaný paralelne bežiacimi programami na grafickej karte, ktoré sa nazývajú *shadery*.



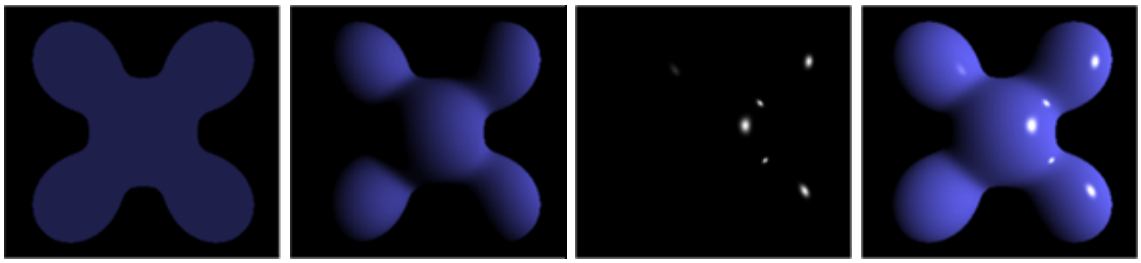
Obr. 3.1: Moderný grafický pipeline

3.1 Phongov osvetľovací model

Úlohou vykresľovacích aplikácií je jednoznačne simulácia svetla, či už sa jedná o jednoduché geometrické primitíva, zložité scény v grafických hrách prípadne špeciálne filmové efekty. Našim cieľom je presvedčiť užívateľa (diváka), že sa pozerá na reálny svet prípadne na jeho analógiu. Aby sme toho dosiahli potrebujeme využiť vhodný spôsob, ktorým svetlá ovplyvňujú povrchy telies v scéne. Využivajú sa na to tzv. *osvetľovacie modely*, ktoré sú aproximáciou fyzikálne presných modelov svetla a poskytujú tak uspokojivé vizuálne výsledky aj napriek svojej nepresnosti.

Jedným z týchto modelov je práve Phongov osvetľovací model [11]. Ide o empyrický osvetľovací model, ktorý využíva metódu výpočtu farby pixelu v závislosti na parametroch zdrojového svetla a povrchu telesa. Parametrami svetla rozumieme jeho pozíciu, prípadne smer a farbu/intenzitu zatiaľ čo parametre povrchu zahŕňajú jeho farbu, smer ktorým je natočený (normála) a lesklosť. Snahou modelu je vykresliť teleso realisticky za rozumnú dobu a jeho výsledky sú natoľko realistické, že sa už niekoľko rokov uplatňuje v real-time grafike.

Samotný model delí zdroj svetla na tri zložky: okolité svetlo (*ambient*), rozptýlené svetlo (*diffuse*) a lesklé svetlo (*specular*), pričom každá je reprezentovaná intenzitou popísanej v kapitole 2.1. Rovnaké zložky sú priradené pre materiály, pričom v tomto prípade predstavujú odrazivosť (*reflectivity*) a tiež sú reprezentované intenzitou. Výslednou hodnotou je suma interakcie zložiek svetiel a materiálov.



(a) ambientná zložka (b) difúzna zložka (c) lesklá zložka (d) Phongovo osvetlenie

Obr. 3.2: Zložky svetla vo Phongovom osvetľovacom modely

3.1.1 Okolité svetlo (ambient)

Okolité svetlo (obrázok 3.2a) vo Phongovom osvetľovacom modele je svetlo ktoré na telesá nedopadá pod konkrétnym uhlom a teda zabezpečí to, že povrchy všetkých telies budú osvetlené rovnako vo všetkých smeroch. Veľkosť jeho zložky je teda konštantná a preto reprezentuje minimálny jas v scéne. Popisuje ju rovnica:

$$A = i_a k_a \tag{3.1}$$

kde:

i_a je intenzita okolitého svetla

k_a je odrazivý koeficient materiálu pre okolité svetlo

3.1.2 Difúzne svetlo (diffuse)

Difúzna zložka (obrázok 3.2b) svetla udáva intenzitu časti svetla, ktorá sa od matného pvrchu telesa rovnomerne odráža do všetkých smerov a dodáva tak telesu trojrozmerný vzhľad. Charakterizuje ju rovnica:

$$D = i_d k_d (\vec{N} \cdot \vec{L}) \tag{3.2}$$

kde:

i_d	je intenzita difúzneho svetla
k_d	je odrazivý koeficient materiálu pre difúzne svetlo
$\vec{N}$	je normála povrchu v osvetlovanom bode
$\vec{L}$	udáva smer svetla

3.1.3 Lesklé svetlo (specular)

Ako už bolo spomínané lesklé svetlo (obrázok 3.2c), ktoré sa niekedy označuje ako zrkadlová zložka je zodpovedná za to aby objekt vypadal žiarivo. Slovo specular znamená „ako zrkadlo“ a prakticky vypadá ako umelý odlesk svetla od povrchu telesa simulujúc odraz svetla od zrkadla kedy sa uhol odrazu (uhol pozorovania) rovná uhlu dopadu. To však platí len v prípade, že je povrch zrkadla dokonale rovný. Ak je povrch zakrivený, svetlo sa odráža do rôznych smerov, čo sa využíva napríklad v technike normal mapping[1]. Rozdiel medzi difúznou a lesklou zložkou teda spočíva v tom, že difúzna zložka modeluje drsné povrchy a lesklá tie žiarivé. Pre lesklé svetlo môžme odvodiť nasledovnú rovnicu:

$$S = i_s k_s (\vec{V} \cdot \vec{R})^a \tag{3.3}$$

kde:

i_s	je intenzita lesklého svetla
k_s	je odrazivý koeficient materiálu pre lesklé svetlo
$\vec{V}$	je vektor pohľadu
$\vec{R}$	udáva vektor odrazu svetla
a	reprezentuje mieru lesklosti povrchu *shininess*

S cieľom získať realistickejšie výsledky je možné výpočet lesklej zložky obohatiť prostredníctvom Fresnelových rovníc [6][7, str. 192], ktoré udávajú intenzitu lomeného a odrazeného svetla.

3.1.4 Atenuácia

Atenuáciou [7, str. 159] rozumieme stratu intenzity svetla v závislosti od jeho vzdialenosti. V skutočnosti je atenuácia priamo úmerná invertovanej druhej mocnine vzdialenosti:

$$intenzita \propto \frac{1}{vzdialenost^2} \tag{3.4}$$

Pre bodové svetlo (*point light*) so stredom v bode P je výsledná intenzita svetla att v bode Q daná vzťahom:

$$att = \frac{1}{k_c + k_l d + k_q d^2} C_0 \tag{3.5}$$

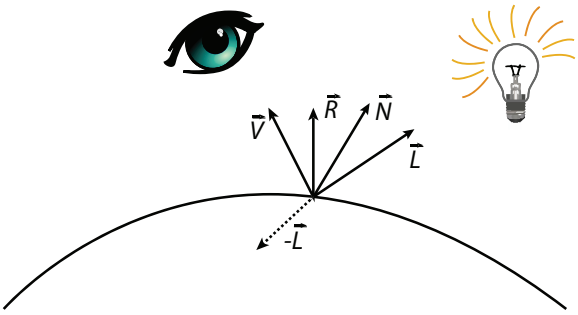
[1] normal mapping (bump mapping) http://www.antongerdelan.net/opengl/normal_mapping.html

kde:

C_0	je farba svetla
d	je vzdialenosť zdroja svetla od bodu Q – $\|\mathbf{P} - \mathbf{Q}\|$
k_c	je nemenná atenuačná konštanta
k_l	je lineárna atenuačná konštanta
k_d	je kvadratická atenuačná konštanta

Z uvedených vzťahov môžme odvodiť konečnú rovnicu pre výpočet Phongovho osvetlenia daného fragmentu bodovým svetlom:

$$L_p = A + (D + S) * att \tag{3.6}$$



Obr. 3.3: Vektory vo Phongovom osvetlovacom modely

3.2 Forward shading

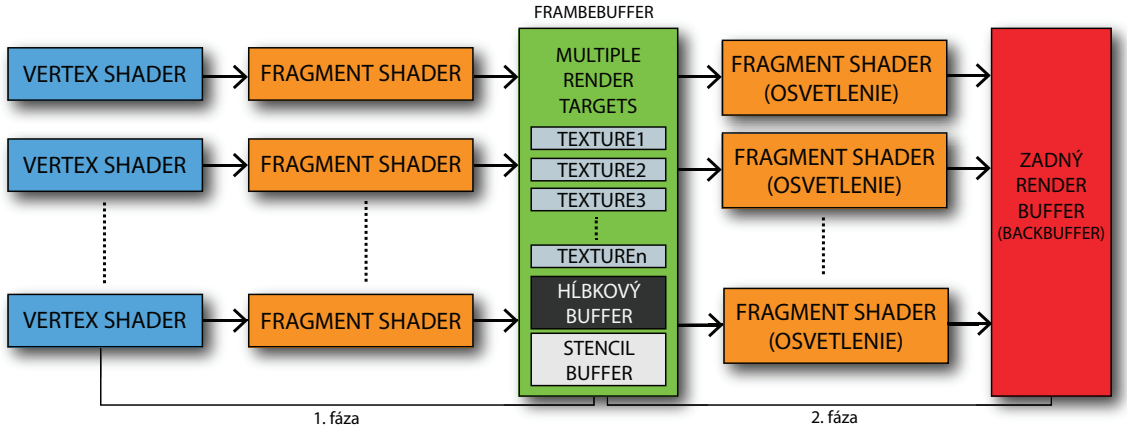
Forward shading je štandardná vykresľovacia metóda, ktorá bola dominantná v real-time 3D grafike práve v čase pevného grafického pipelinu no dnes sa stále využíva vo väčšine jednoduchých grafických enginov. Postup výpočtu tejto techniky je jednoduchý: grafickej karte poskytneme informácie o geometrii, GPU ich rozloží na vrcholy, transformuje a rozdelí fragmenty na ktoré následne aplikuje osvetlenie. Výsledné fragmenty sú následne vykreslené na obrazovku.

Pri zložitejších scénách s množstvom dynamických svetiel je však táto metóda značne problematická, protože so zvyšujúcim počtom svetiel sa zvyšuje potreba ich selektívnej aplikácie s cieľom redukcie počtu výpočtov vo fragment shaderoch. Zmena svetiel aplikovaných na geometriu je však možné uskutočniť iba medzi volaním vykresľovacích funkcií, čo značne obmedzuje možnosť vyraďovať nepotrebné svetlá a zvyšuje počet invokácií shaderov. Spoločným znakom týchto nevýhod je základný problém forward shadingu, ktorým je úzka spojitosť rasterizácie scény a výpočtu osvetlenia. Riešením tohoto problému môže byť viacpriechodové vykresľovanie z ktorého vychádza deferred shading, alebo využiť výhody techniky tiled shading popísanej v nasledujúcej kapitole.

3.3 Deferred shading

Ďalšou zo štandardných techník výpočtu Phongovho osvetlenia v moderných počítačových hrách je deferred shading, v preklade tzv. *odložené tieňovanie*. Narozdiel od klasického jednopriechodového forward shadingu je výpočet osvetlenia posunutý až do druhého priechodu po vykreslení geometrie.

Základným prvkom tejto techniky je štruktúra nazývaná *geometry buffer (G-buffer)* do ktorej sa v prvom priechode vykreslí celá scéna po výsledných geometrických výpočtoch a následne je v druhom priechode na každý pixel/fragment tejto štruktúry aplikované osvetlenie a tieňovanie. Tento postup demonštruje obrázok 3.4. Algoritmus je však možné doplniť o tretiu fázu, ktorou sú post processing techniky. Aj napriek tomu, že táto technika predstavuje prelomový prístup k výpočtu osvetlenia v súčasnosti sa v štandardnej forme príliš nevyužíva a nahradzujú ju pokročilé techniky odvodené práve z deferred shadingu. Sú nimi napríklad tiled deferred shading či clustered deferred shading.



Obr. 3.4: Fázy algoritmu deferred shading. V prvej fáze sú vertex shaderu zaslané informácie o vrcholoch (*pozície, normály, koordináty textúr atď.*), na ktoré sú následne aplikované transformačné matice. Transformované vrcholy sú spolu s informáciami o materiáloch zaslané do fragment shaderu, ktorý ich uloží do textúr framebufferu. V druhej fáze sa pre každé svetlo spočíta osvetlenie vo fragment shadery s využitím textúr z predchádzajúcej fázy.

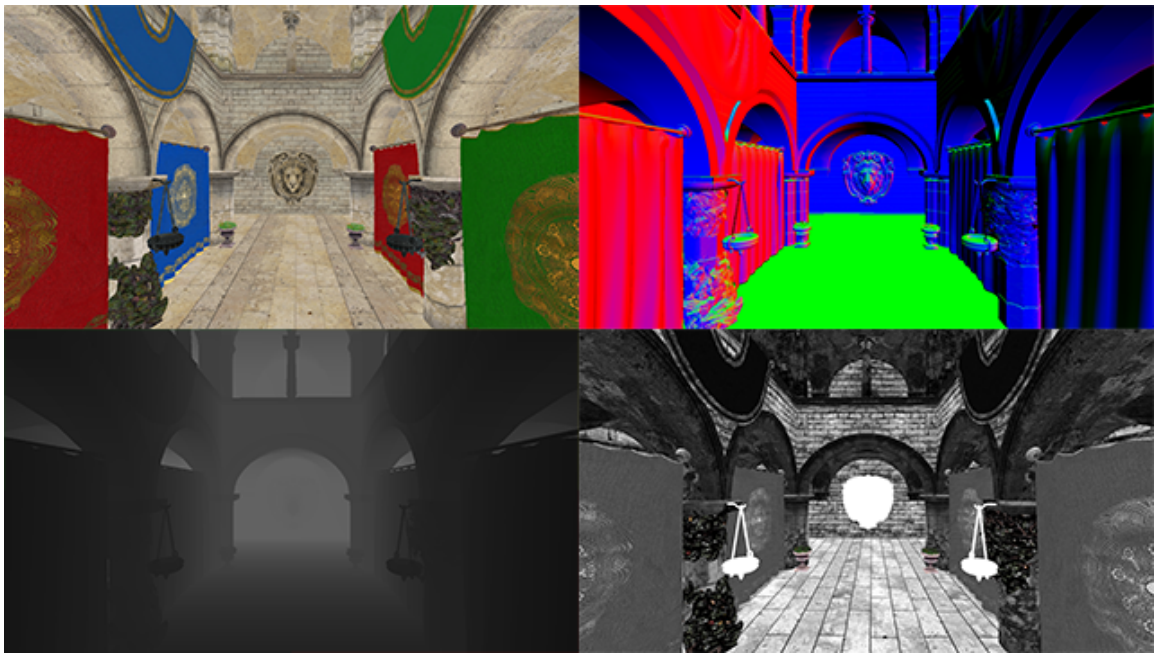
3.3.1 Vykreslenie geometrie do G-bufferu

Ako už bolo spomínané v predchádzajúcej kapitole, technika forward shading je jednopriechodová, čo prakticky znamená, že vrcholy spracované vertex shadermi sú následne priamo zaslané na vstup fragment shaderu (za predpokladu, že nevyužívame ďalšie typy shaderov moderného grafického pipelinu zobrazeného na obrázku 3.1) a po spracovaní uložené do defaultného *framebufferu*[2]. Nakoniec je jeho obsah zobrazený na grafický výstup.

OpenGL nám však umožňuje ukladať výstupné informácie z fragment shaderu do pamäti, konkrétne napr. do textúry, prípadne bufferu. V technike deferred shading sa využíva práve tejto funkcionality a to za účelom uchovania per-pixel informácií o geometrii ako sú

[2]Framebuffer (FBO) reprezentuje posledné štádium OpenGL grafického pipelinu. Jedná sa o kolekciu bufferov, ktoré sa využívajú ako cieľová vykresľovacia destinácia.

normály, hĺbka, difúzne textúry materiálov či odrazové (specular) textúry. S využitím niekoľkých textúr dokážeme uchovať rôzne výstupy fragment shaderu. Možnosť zapisovať tieto vlastnosti do osobitných textúr je v OpenGL známa ako *Multiple Render Targets*. Dáta ktoré sa ukladajú do textúr sú výsledkom interpolácie vykonanej rasterizérom na jednotlivých geometrických primitívach. Pri technike deferred shading je vhodné si tieto informácie logicky oddeliť od klasického toku spracovania do osobitného framebufferu. Prakticky sa teda pod pojmom G-buffer neskrýva nič iné ako programátorom definovaný framebuffer obsahujúci zoskupenie týchto textúr (obrázok 3.5).



Obr. 3.5: Ukážka G-bufferu z výslednej aplikácie tejto práce: *albedo diffuse* (vľavo hore), *view space normály* (vpravo hore), *linearizovaný hĺbkový buffer* (vľavo dole), *albedo specular* (vpravo dole).

OpenGL vyžaduje aby bola každá textúra priradená konkrétnemu framebufferu. Pokiaľ programátor nedefinuje inak, sú štandardne textúry pripojené do defaultného framebufferu, konkrétne mapovaním na jeho prípojky (*attachments*). Počet prípojok je však obmedzený. Každému framebufferu môžme priradiť maximálne 8, pričom existuje niekoľko typov prípojok, či už na uloženie informácií o farbe (`GL_COLOR_ATTACHMENTx` kde `x` značí index prípojky). Každá prípojka je taktiež obmedzená počtom a veľkosťou kanálov pre ukladanie dát. Počet kanálov môže byť maximálne 4 (RGBA) pričom rozsah každého môže byť maximálne 32 bitov. Dôležité je, že každý kanál prípojky zaberá určitú šírku pamäťového prenosového pásma a preto treba zvážiť s akou presnosťou potrebujeme dané data ukladať, pretože pri nevhodnom návrhu framebufferu môžme znevážiť výhody tohto prístupu. Na uloženie informácií o farbe si napríklad vystačíme s rozsahom 16 bitov, ale pri rozsiahlych scénach je potrebné zvážiť použitie 32 bitového kanálu pre uloženie world space pozície, aby sa zachovala požadovaná presnosť.

Samotný zápis do G-bufferu je jednoduchá operácia, pri ktorej si vystačíme s jednoduchým vertex a fragment shaderom. Úlohou vertex shaderu je konverzia vrcholu do clip space pre neskoršiu prácu rasterizéru, zatiaľ čo fragment shader sa stará o ukladanie separátnych informácií do jednotlivých prípojok framebufferu.

3.3.2 Výpočet osvetlenia

Druhou fázou techniky deferred shading je výpočet a následná aplikácia osvetlenia na jednotlivé pixely geometrie uloženej v G-buffery. Podstata tejto fázy je jednoduchá: každý zdroj svetla sa vykreslí ako guľa (uvažujeme iba bodové svetlá), pričom všetky vrcholy, ktoré sú vo vnútri hraničnej gule a na jej povrchu budú ovplyvnené daným svetlom. Tento krok sa opakuje pre všetky svetlá. V prípade, že sa niektoré svetlá prekrývajú, výslednú farbu pixelu získame zmiešaním farieb jednotlivých svetiel. Na výpočet osvetlenia v tomto kroku sa využívajú osvetľovacie modely ako napr. Phongov osvetľovací model popísaný v kapitole 3.1. Deferred shading týmto úspešne redukuje potrebné výpočty osvetlenia avšak za cenu vysokých pamäťových nárokov a réžie spojenej s prístupom do tejto pamäte.

```
for each light
{
   sample = load attributes from G-buffer

   color = calculateLight (sample , light )

   output color

   blend color with other lights
}
```

Listing 3.1: Pseudokód pre výpočet osvelenia v deferred shadingu, množstvo výpočtov v tomto prípade sa rovná $O(N_{svetiel} \cdot N_{vzorkov})$

Kapitola 4

Tiled Shading

V tejto kapitole bude predstavená modifikácia predchádzajúcich dvoch techník[1], ktorou je tiled shading [9]. Podstata tejto metódy spočíva v rozdelení obrazovky na štvorce rovnakej veľkosti nazývané tily (*tiles*), ktoré predstavujú samostatné jednotky pri prevádzaní výpočtov osvetlenia. Každý *tile* obsahuje zoznam svetiel, ktoré ho potencionálne ovplyvňujú. Tento prístup nám poskytuje veľké výhody, či už sa jedná o značnú redukciu potrebných výpočtov tieňovania (vedie k podstatnému urýchleniu výslednej aplikácie) alebo relatívne jednoduchý prechod medzi technikami *tiled forward* a *tiled deferred*.

Za východisko pre vznik tiled shadingu možno považovať techniku *Tiled Rendering* [4], s ktorou má určité podobnosti. Tiled Rendering bol predstavený v roku 1989, a jeho základom bolo, že tily sa neaplikovali na svetlá ale na geometrické primitíva. S narastajúcou komplexnosťou grafických scén však táto technika prestala byť efektívna pre použitie v real-time aplikáciách.

Samotný algoritmus Tiled shadingu pozostáva z nasledujúcich fáz:

1. rozdelenie obrazovky do mriežky tilov
2. vykreslenie hĺbky do bufferu
   (a) *depth pre-pass* [tiled forward](optimalizácia)
   (b) vykreslenie scény do G-bufferu [tiled deferred]
3. vkladanie svetiel do mriežky
   (a) vyraďovanie svetiel
   (b) výpočet minimálnej a maximálnej hĺbky pre každý tile (optimalizácia)
4. výpočet osvetlenia
5. vykreslenie osvetlenej scény

[1] Informácie uvedené tejto kapitole sú spoločné pre tiled forward aj tiled deferred shading. V závere kapitoly budú uvedené špecifiká jednotlivých techník.

4.1 Vykreslenie hĺbky

Za počiatočný krok algoritmu v technike tiled shading možno považovať vykreslenie hĺbky do bufferu. Pri deferred metódach je tento krok triviálny, pretože sa jedná o klasické vykreslenie geometrie do G-bufferu popísané v kapitole 3.3.1.

V tiled forward shadingu je tento krok označovaný ako *Depth pre-pass*, prípadne *Early-Z pass*, pretože po jeho vykonaní poznáme informácie o hĺbke ešte pred samotným vykreslením scény. V tomto prípade je však považovaný za optimalizáciu a teda nie je priamo vyžadovaný. Nepoužitím tohto kroku sa zamedzuje ďalším možným optimalizáciám hĺbkového bufferu, ktoré budú popísané v ďalších kapitolách. Samotné vykreslenie je veľmi podobné tomu v deferred shadingu s tým rozdieľom, že do framebufferu nie je potrebné vykresľovať všetky informácie ako normály atď. ale iba hĺbku. Nedochádza tak k vytváraniu pamäťovo náročných štruktúr.

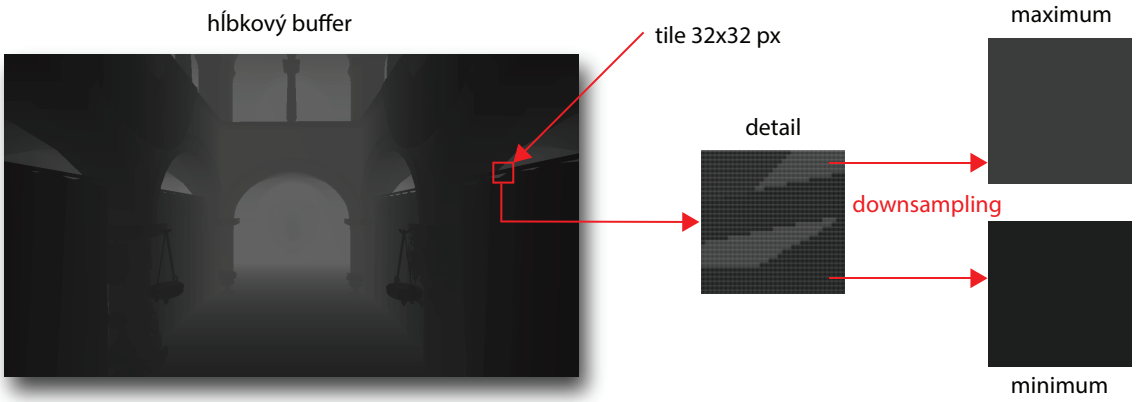
4.2 Optimalizácie hĺbkového bufferu

V tejto kapitole budú predstavené možné optimalizácie založené na práci s hĺbkovým bufferom. Cieľom týchto optimalizácií je najmä zvýšenie efektivity vyraďovania svetiel (viď kapitola 4.3), ktoré neovplyvňujú geometrické primitíva zasahujúce do zvoleného tilu.

4.2.1 Minimum a maximum hĺbkového bufferu

Veľmi efektívnu a na implementáciu pomerne jednoduchú optimalizáciu tiled shadingu predstavuje optimalizácia hĺbkové bufferu výpočtom konzervatívnych hodnôt minima a maxima pre každý tile (obrázok 4.2). Prerekvizitou pre tento výpočet je existencia hĺbkového bufferu (kapitola 4.1).

Zatiaľ čo klasický deferred shading s hĺbkovým bufferom štandardne ďalej nepracuje, v technike tiled shading hrá dôležitú úlohu. Po vykreslení scény/hĺbky do framebufferu sa totiž môže využiť na výpočet minimálnej a maximálnej hĺbku pre každý tile pričom sú tieto hodnoty neskôr využité pri vyraďovaní svetiel ktoré daný tile môžu ovplyvňovať. Z pôvodného hĺbkového bufferu tak získame nový buffer ktorý je zmenšený *t-krát*, pričom *t* je veľkosť tilu. Preto sa tento proces nazýva *downsampling*(viď obrázok 4.1), teda znižovanie počtu vzorkov.



Obr. 4.1: Downsampling hĺbkového bufferu.

Výpočet je vhodné vykonať na GPU pričom existuje niekoľko možností ako ho uskutočniť. Pri práci so staršími grafickými kartami, ktoré nepodporujú najnovšie verzie OpenGL (nie je možné použiť *compute shader*) je možné k výpočtu použiť napríklad jednoduchý fragment shader. Ďalšou možnosťou je na tento účel využiť framework OpenCL[2] a metódu paralelnej redukcie[3]. Ak cieľový grafický akcelerátor podporuje najnovšie verzie OpenGL, je výhodné použiť *compute shader* a to najmä z dôvodu možnosti využitia atomických operácií a zdieľaných premenných.

(a) hĺbkový buffer (b) minimá (c) maximá

Obr. 4.2: Konzervatívne hodnoty minima a maxima per tile. Svetlé plochy predstavujú vzdialenejšie vrcholy zatiaľ čo tmavé tie bližšie. Veľkosť tilu na obrázkoch je 32x32

Vypočítané hodnoty budú neskôr ďalej spracovávné a preto je potrebné uchovávať ich vo vhodnej forme napr. vo forme číselného bufferu alebo textúry. Záleží na prostriedkoch ktoré chceme využiť na konštrukciu mriežky svetiel (môže prebiehať na CPU alebo na GPU).

4.2.2 2.5D Culling

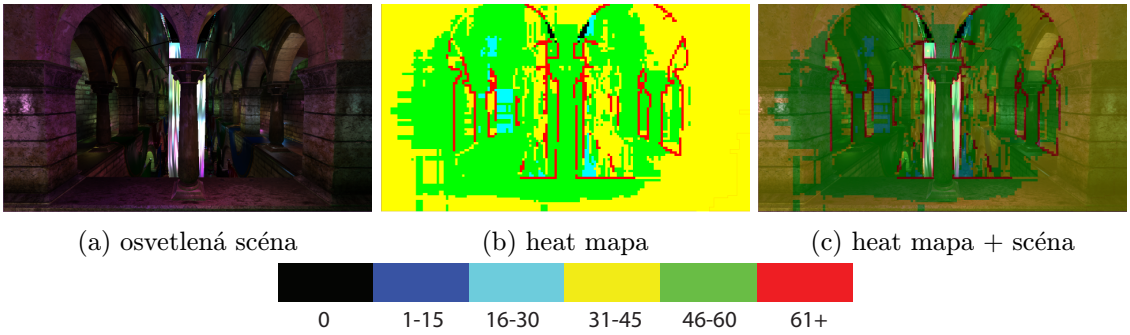
Predchádzajúca metóda predstavuje výhodnú optimalizáciu, ktorá značne redukuje potrebné výpočty osvetlenia. Aj napriek tomu má však svoje nedostatky, z ktorých za najzávažnejší možno považovať nesprávne vyraďovanie svetiel pri nespojitosti hĺbky (viď obrázok 4.3). Pri veľkých rozdieloch v hĺbke v rámci tilu sa do jeho zoznamu svetiel vkladajú aj svetlá, ktoré síce spadajú do rozmedzia tvoreného minimom a maximom, ale neovplyvňujú žiadne povrchy a teda dochádza zbytočným výpočtom osvetlenia. Nespojitosť hĺbky tak vzniká najmä na hranách objektov ako je možné vidieť na obrázku 4.3b.

Riešením tohto problému, nie však stopercentným, je metóda *2.5D Culling* [5], ktorá rozdeľuje hĺbkový buffer v každom tile v intervale $\langle min, max \rangle$ na niekoľko častí (často sa využíva maximálna veľkosť údajového typu `integer` na danej architektúre, 32/64 bitov). Každý tile je teda reprezentovaný bitovou maskou vo formáte `100..001`, pričom každá vzniknutá časť je reprezentovaná jedným bitom a počiatočná koncová 1 reprezentuje hranice hĺbky v tile (minimum a maximum). Pre každé svetlo v existujúcom light liste sa následne tiež vytvorí bitová maska a bitovým súčinom s maskou tilu sa testuje na prekrytie. Dochádza tak k eliminácii tzv. *false positives*, teda svetiel, ktoré neovplyvňujú geometriu.

Keďže si táto optimalizácia vyžaduje zložitejšiu prácu s hĺbkovým bufferom využíva sa najmä pri metóde vyraďovania svetiel založeného na výpočte view space frusta pre každý tile, ktorý je popísaný v kapitole 4.3.1.

[2] OpenCL (*Open Computing Language*) http://www.khronos.org/opencl/

[3] Paralelné redukcie v OpenCL http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/



(a) osvetlená scéna (b) heat mapa (c) heat mapa + scéna

(d) počet svetiel v tile

Obr. 4.3: Nespojitosť hĺbky, model Crytek Sponza osvetlený 4096 statickými svetlami, veľkosť tilu 8x8

4.3 Vyraďovanie svetiel

Proces inak nazývaný *light culling* možno považovať za najdôležitejšiu fázu techniky tiled shading. Práve v tomto kroku sa prakticky vytvára mriežka svetiel v ktorej sa uchovávajú zoznamy svetiel ovplyvňujúce jednotlivé tily. Cieľom je efektívne zistiť ktoré svetlá zasahujú do ktorých tilov, čo vyžaduje zvážiť spôsob akým vyraďovať svetlá, ktoré nebudú zaradené do mriežky svetiel. Najčastejšie sa využívajú dva prístupy:

1. výpočet *view space frusta*[4] pre každý tile
2. výpočet *screen space bounding quadu* (hraničného obdĺžnika) pre aktívne svetlá

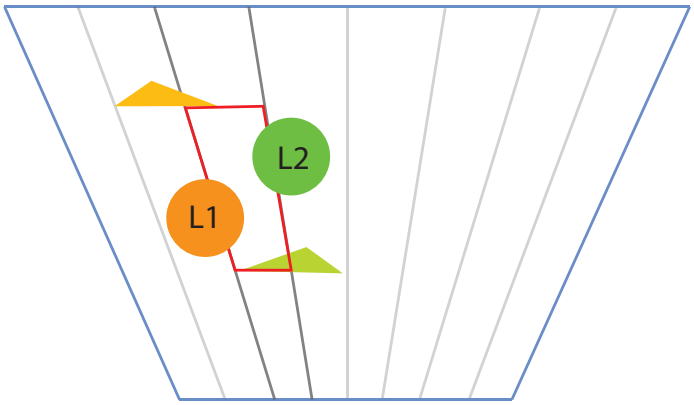
Výhody a nevýhody týchto metód sú popísané v nasledujúcich podkapitolách. Je potrebné uviesť, že v prípade použitia optimalizácií hĺbkového bufferu (4.2) musia byť tieto optimalizácie zavedené práve v tomto kroku pred konštrukciou mriežky svetiel, nakoľko proces vkladania svetiel do nej je na nich závislý.

4.3.1 Tile frustum

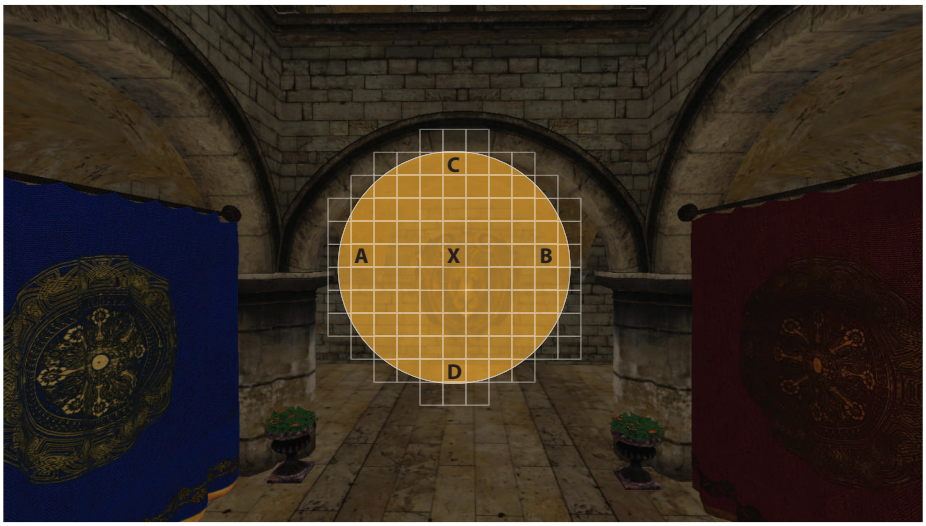
Konštrukcia view space frusta pre jednotlivé tily je pri technike tiled shading využívaná pomerne často. Postup [2] je pomerne jednoduchý, stačí vypočítať 4 steny (vrchnú, spodnú, ľavú a pravú) tvoriace frustum každého tilu pričom prednú stenu (*near plane*) a zadná stenu (*far plane*) poznáme. V prípade, že je použitá optimalizácia hĺbkového bufferu predstavujú tieto hranice hodnoty minima a maxima hĺbky (obrázok 4.4). Je zrejmé, že táto metóda vedie k veľkému počtu redundantných výpočtov, protože tily v riadkoch zdieľajú spoločné steny rovnako ako tily v stĺpcoch. Ďalej je potrebné zvážiť situáciu kedy svetlo zasahuje do susedných frúst pomyseľného tilu. Je jasné, že svetlo zasahuje aj do pomyseľného tilu, a teda je zbytočné túto skutočnosť počítať (viď obrázok 4.5).

Na výpočet je vhodné použiť compute shader, čo však vyžaduje atomické operácie a synchronizáciu vlákien a preto je táto metóda nevhodná pre staršie grafické karty (podpora OpenGL 4.2 a nižšie verzie). Je však pomerne jednoduchá na implementáciu a pri jej použití nie je potrebné široké prenosové pásmo, protože zoznamy svetiel sú uchovávané lokálne ako zdieľané premenné. Na druhej strane tieto záznamy sú pomerne malé a preto sa nejedná o veľkú úsporu.

[4] frustum - zrezaný ihlan



Obr. 4.4: Ukážka rozdelenia view space frusta na tile frustá. Červenou farbou je zobrazené tile frustum s použitím optimalizácie hĺbky výpočtom minima a maxima hĺbky. L1 a L2 predstavujú svetlá ovplyvňujúce zvolený tile.



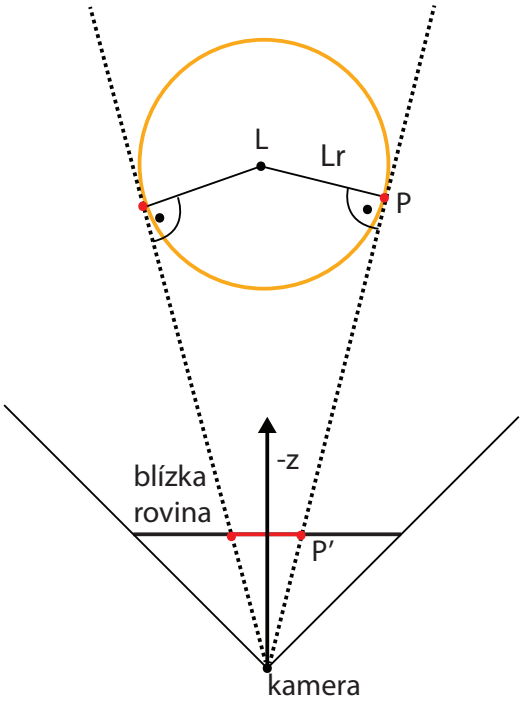
Obr. 4.5: Zobrazenie redundantních výpočtov. V prípade že uvažujeme tile X a vieme, že svetlo pretína frustum tilu A a tilu B, prípadne tile C a tile D je jasné, že nie je potrebné testovať prienik svetla s ohľadom na frustá tilov medzi týmito bodmi.

Pri konštrukcii tile frusta môže čitateľa napadnúť, že pomerne často sa môže stať, že hĺbka frusta je príliš veľká a môže dochádzať k nežiadúcej situácii ktorou je *diskontuita hĺbky*. Uvažujme situáciu kedy do tilu zasahuje objekt umiestnený blízko ku kamere a zároveň doň zasahuje objekt umiestnený vo veľkej vzdialenosti, napríklad vzdialená nerovnosť terénu. Oba tieto objekty predstavujú hranice frusta (minimum a maximum). Medzi týmito objektami môže byť umiestnených niekoľko svetiel, ktoré síce pretínajú frustum zvoleného tilu, ale reálne neosvetľujú žiaden objekt. Aj napriek tejto skutočnosti sú však svetlá uložené do zoznamu svetiel tohto tilu. Existuje niekoľko optimalizácii vyraďovania svetiel, ktoré dbajú na túto skutočnosť. Najznámejšia z nich je metóda *2.5D culling*(kapitola 4.2.2), ktorá našla svoje uplatnenie najmä v technike tiled forward shading.

4.3.2 Screen space bounding quad

Ďalšou možnosťou vyraďovania svetiel je výpočet screen space bounding quadu pre každé aktívne svetlo, ktorý je implementovaný vo vzorovej aplikácii tejto práce. Cieľom je nájsť súradnice ľavého spodného a pravého horného rohu obdĺžnika ohraničujúceho svetlo, ktoré je viditeľné na obrazovke. Keďže získané súradnice sú v screen space je veľmi jednoduché odvodiť, ktoré tily sú ovplyvnené zvoleným svetlom. Už v tomto momente je jasné, že efektivita týchto výpočtov je vyššia ako pri predchádzajúcej metóde pretože neobsahuje množstvo redundantných výpočtov. Výpočet je zložitejší ako sa na prvý pohľad môže zdať preto ho odvodím.

Predpokladajme, že je dané bodové svetlo, ktoré má stred v bode L vo view space a dosah (polomer) označme Lr. Hľadáme 4 roviny prechádzajúce bodom, v ktorom je umiestnená kamera, pričom dve sú rovnobežné s rovinou osi x a dve s rovinou osi y. Tieto roviny svetlo ohraničujú, sú kolmé na jeho polomer a dotýkajú sa povrchu gule, ktorá svetlo reprezentuje. Nájdením rovníc týchto rovín, je už jednoduché nájst ich prieniky s blízkou rovinou a získať tak bounding quad premietnutého svetla. Situáciu podrobne zobrazuje zobrazuje obrázok 4.6.



Obr. 4.6: Priemet svetla na obrazovku.

Ďalším predpokladom pre výpočet je skutočnosť že roviny rovnobežné s osou y majú jednotkový normálový vektor $\vec{N}$, ktorého y súradnica je 0. Pretože tieto roviny prechádzajú počiatkom sústavy (bod, kde je umiestnená kamera) je možné každú reprezentovať 4 rozmerným vektorom $\vec{T} = (N_x, 0, N_z, 0)$. Cieľom je vypočítať hodnoty N_x a N_z tak aby platili nasledujúce podmienky:

$$T \cdot L = Lr \tag{4.1}$$

$$N_x^2 + N_z^2 = 1 \tag{4.2}$$

Skalárnym súčinom vektorov získame vzťah:

$$N_x L_x + N_z L_z = Lr \tag{4.3}$$

Rovnicu prepíšeme do vhodného tvaru a umocníme:

$$N_z^2 L_z^2 = (Lr - N_x L_x)^2 \tag{4.4}$$

V tomto kroku môžeme použiť substitúciu N_z^2 z rovnice 4.2:

$$subst. N_z^2 = 1 - N_x^2 \tag{4.5}$$

Dosadíme a roznásobíme:

$$L_z^2 - N_x^2 L_x^2 = Lr^2 - 2N_x L_x Lr + N_x^2 L_x^2 \tag{4.6}$$

Rovnicu 4.6 je možné prepísať do tvaru kvadratickej rovnice premennej N_x:

$$N_x^2(L_x^2 + L_z^2) - N_x(2L_x r) + Lr^2 - L_z^2 = 0 \tag{4.7}$$

Spočítame diskriminant:

$$D = 4[L_x^2 Lr^2 - (L_x^2 + L_z^2)(Lr^2 - L_z^2)] \tag{4.8}$$

$D \leq 0$ práve vtedy keď $L_x^2 + L_z^2 < Lr^2$. V tomto prípade guľa reprezentujúca svetlo vypĺňa celú obrazovku a výpočet ukončíme. V opačnom prípade, keď $D > 0$ pokračujeme riešením rovnice 4.8:

$$N_x = \frac{2LrL_x \pm \sqrt{D}}{2(L_x^2 + L_z^2)} = \frac{(LrL_x \pm \sqrt{\frac{D}{4}})}{L_x^2 + L_z^2} \tag{4.9}$$

V poslednom kroku sme získali prvú hľadanú hodnotu normálového vektoru $\vec{N}$. Potrebujeme ešte spočítať hodnotu N_z, ktorú odvodíme z rovnice 4.4:

$$N_z = \frac{(Lr - N_x L_x)}{L_z} \tag{4.10}$$

V tomto momente máme všetko potrebné pre výpočet vektoru $\vec{N}$. Netreba zabudnúť na to, že nás zaujímajú len roviny ktorých dotykový bod s guľou reprezentujúcou svetlo leží pred kamerou. Takýto bod je napríklad bod P zobrazený na 4.6. Tento bod leží v rovine $\vec{T}$ vo vzdialenosti Lr od *view space* pozície svetla L. Z nasledujúcich skutočností vyplýva:

$$P \cdot N = 0 \tag{4.11}$$

$$(P - L)^2 = Lr^2 \tag{4.12}$$

Rovnicu 4.12 môžeme roznásobiť a s využitím Pytagorovej vety je možné nahradiť P^2 vzťahom $L^2 - Lr^2$:

$$P \cdot L = L^2 - Lr^2 \tag{4.13}$$

Pre hľadaný bod P ležiaci pred kamerou platí, že $P_z < 0$. Keďže rovina $\vec{T}$ je rovnobežná s osou y, hodnoty P_y a L_y sú rovnaké a hodnota L_y^2 z rovnice vypadne. Z rovnice 4.11 môžeme odvodiť:

$$P_x = -\frac{P_z N_z}{N_x} \tag{4.14}$$

Po dosadení tohto vzťahu do rovnice 4.13 a drobnej úprave získame:

$$P_z = \frac{L_x^2 + L_z^2 - Lr^2}{L_z - (\frac{N_z}{N_x})L_x} \tag{4.15}$$

Následne je potrebné vypočítať roviny rovnobežné s osou x. V tomto prípade sú roviny reprezentované 4 rozmerným vektorom $\vec{T} = (0, N_y, N_z, 0)$. Rovnakým postupom ako v predchádzajúcom prípade získame rovnice pre N_y a N_z:

$$N_y = \frac{LrL_x \pm \sqrt{L_y Lr^2 - (L_y^2 + L_z^2)(Lr^2 - L_z^2)}}{L_y^2 + L_z^2} \tag{4.16}$$

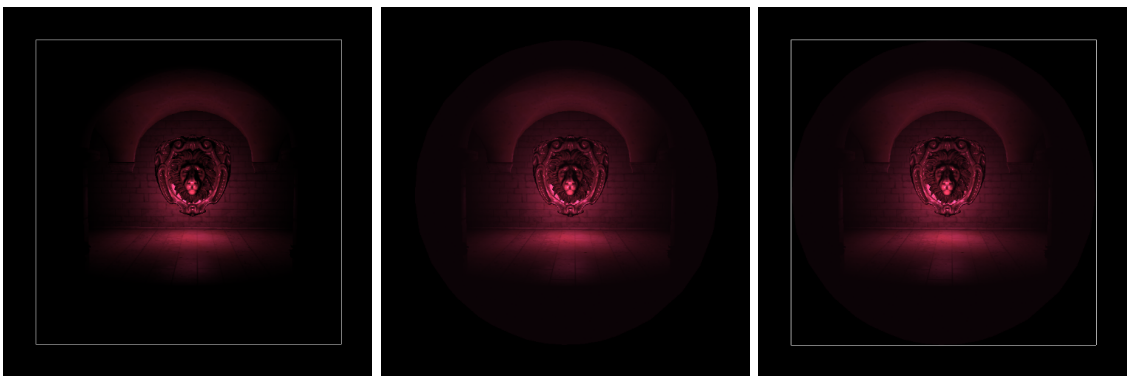
$$N_z = \frac{Lr - N_y L_y}{L_z} \tag{4.17}$$

A rovnice pre bod P:

$$P_z = \frac{L_y^2 + L_z^2 - Lr^2}{L_z - (\frac{N_z}{N_y})L_y} \tag{4.18}$$

$$P_y = -\frac{P_z N_z}{N_y} \tag{4.19}$$

Získaný bod P je v oboch prípadoch potrebné premietnuť s využitím projekčnej matice na blízku rovinu. Týmto získame hodnotu v clip space, teda v rozsahu $(-1.0, 1.0)$, ktorú je potrebné previesť do rozlíšenia okna aplikácie. Výsledkom celého uvedeného postupu je štvorica hodnôt reprezentujúca ľavý spodný roh a pravý horný roh hraničného obdĺžnika pre zvolené svetlo. Tieto hodnoty je potrebné otestovať aby sme overili, či bol výpočet správny. Korektnosť výpočtov je je možné overiť napríklad vykreslením výsledného quadu na obrazovku s využitím funkcií zo starších verzií OpenGL čo znázorňuje obrázok 4.7.



(a) bounding quad (b) bounding volume (c) quad + volume

Obr. 4.7: Overenie výsledných hodnôt

4.3.3 Konštrukcia mriežky svetiel

Nasledujúcim krokom je tvorba mriežky svetiel, v ktorej sa uchovávajú zoznamy svetiel ovplyvňujúce daný tile. Pred samotnou konštrukciou mriežky je dôležité vhodne zvoliť veľkosť tilov. Pri voľbe veľkosti je potrebné zvážiť komplexnosť grafickej scény, počet svetiel a operácie spojené s týmito prvkami. Na základe týchto informácií totiž závisí množstvo výpočtov osvetlenia či prípadné prenosy dát z pamäte (*memory bandwith*). Ak zvolíme väčšie tily ušetríme na pamäťovej náročnosti a množstve prenášaných dát, avšak sa zväčšuje množstvo výpočtov hraníc svetiel. Najčastejšie sa využívajú veľkosti *32x32* alebo *16x16*.

Ďalej je potrebné rozhodnúť akým spôsobom chceme mriežku svetiel konštruovať a aké prostriedky na to využiť. Výpočet totiž môžme prevádzať na CPU ako aj na GPU (s využitím compute shaderov).

4.4 Shading

Spôsob výpočtu osvetlenia je mierne odlišný ako v základných osvetľovacích technikách, kde sa každé svetlo spracováva sériovo, zatiaľ čo v tiled technikách sa spracovávajú naraz všetky svetlá ovplyvňujúce tile. Okrem toho sa však postup shadingu pre forward a deferred techniky nemení a prevádza sa rovnako ako pri základných variantách. V praxi to znamená, že sa zmení iba fragment shader.

```
for each G-buffer sample
{
   sample = load attributes from G-buffer
   count = get number of lights affecting tile
   offset = get offset to global light list

   for each light affecting tile
   {
      color += calculateLight (sample , light )
   }
   out color;
}
```

Listing 4.1: Pseudokód pre výpočet osvelenia v tiled deferred shadingu

```
render geometry as usual

for each generated fragment
{
   count = get number of lights affecting tile
   offset = get offset to global light list

   for each light affecting tile
   {
      color += calculateLight (sample , light )
   }
   out color;
}
```

Listing 4.2: Pseudokód pre výpočet osvelenia v tiled forward shadingu

4.5 Tiled Deferred Shading

Veľmi populárnoym sa nedávno stal práve Tiled *Deferred* Shading o čom svedčí napríklad úspech herného titulu Battlefield 3[1], ktorý využíva práve túto techniku. Za zmienku stojí aj fakt, že táto technika bola implementovaná na herných konzolách Microsoft XBOX 360 ako aj na Sony Playstation 3. Jej cieľom je vyriešiť hlavný problém, ktorý vzniká s využitím klasického deferred shadingu - množstvo prenášaných dát z pamäte do shaderov. Pre každý fragment ktorý je osvetlený je totiž potrebné získať informácie z G-bufferu, pričom sa postup opakuje pre každé svetlo, čo vzhľadom na časovú zložitosť funkcií, ktoré čítajú dáta z textúr spôsobuje značný pokles plynulosti výslednej aplikácie.

Práve prístup ktorý prináša technika tiled shading je značnou optimalizáciou tohto problému. Keďže sa výpočet osvetlenia prevádza na úrovni tilu, dochádza k značnému zredukovaniu počtu prístupov do pamäti. Pre každý osvetlený fragment je potrebné získať dáta z G-bufferu iba raz. Okrem toho prináša tiled deferred shading tieto výhody:

- fragmenty v rámci rovnakých tilov spracovávajú rovnaké svetlá
- ukladanie svetiel sa uskutočňuje v registroch, s presnosťou desatinných čísel

Stále však nie sú vyriešené ďalšie problémy, ktoré deferred shading všeobecne prináša ako napríklad zložité spracovania priehľadných objektov, či použiteľnosť antialiasingu. Do úvahy treba taktiež vziať pamäťové nároky, ktoré si vyžaduje jej použitie. Napríklad pri 16-násobnom vzorkovaní v HD prípadne full HD rozlíšení narastajú pamäťové nároky G-bufferu do pomerne vysokých hodnôt.

4.6 Tiled Forward Shading

Narozdiel od predchádzajúcej techniky tiled shading značne zvyšuje použiteľnosť klasického forward shadingu, kde sa náročnosť výpočtov zvyšuje lineárne s každým pridaným svetlom. V tiled forward shadingu[5][10, 9] sú svetlá uložené do pamäti práve raz a to pred samotným výpočtom osvetlenia. Práve táto skutočnosť umožňuje použiteľnosť omnoho väčšieho počtu svetiel. Stále však môže dochádzať k prekresľovaniu rovnakého fragmentu, čomu sa zamedzuje práve zavedením optimalizácie Early-Z fázy. Za ďalšie výhody, ktoré tiled forward shading prináša možno považovať:

- ukladanie svetiel sa uskutočňuje v registroch, s presnosťou desatinných čísel
- jednoduché riešenie transparentnosti
- podpora Full Screen Antialiasing

Cieľom tiled forward shadingu je využiť výhody (tiled) deferred techník (redukciu potrebných výpočtov osvetlenia) a klasického forward shadingu (nízke pamäťové nároky a réžia). Keďže táto technika nepotrebuje uchovávať informácie o geometrii v bufferoch má nízke pamäťové nároky.

[5]Spoločnosť AMD túto techniku označuje ako *Forward+*

Kapitola 5

Clustered Shading

V tejto kapitole si veľmi stručne predstavíme techniku pre výpočet osvetlenia, ktorá sa do povedomia dostala pomerne nedávno – Clustered shading. Jej podstata je podobná ako v technike tiled shading kde sa vzorky (pixely/fragmenty) spracovávajú na základe 2D pozícií zoskupovaním do tilov. Clustered shading zoskupuje vzorky zoskupuje do tzv. zhlukov(*clustrov*) na základe podobných 3D vlastností ako napríklad pozícií a normál. Výhodou tejto techniky je, že značne znižuje výpočty osvetlenia a zvyšuje ich efektivitu. Pri výpočte osvetlenia je rovnako možné použiť forward aj deferred metódy, pre teoretický pohľad však budeme predpokladať použitie clustered deferred shadingu.

Algoritmus je podobný ako pri tiled deferred shadingu:

1. vykreslenie scény do G-bufferu
2. určenie clustrov
3. redukcia clustrov na unikátne clustre
4. priradenie svetiel do clustrov
5. výpočet osvetlenia

5.1 Vykreslenie geometrie

Prvým krokom nie je nič iné ako klasická fáza vykreslenia geometrie v deferred shadingu, ktorá bola spomínaná už niekoľko krát(kapitoly 3.3.1 a 4.1).

5.2 Určenie clustrov

V ďalšom kroku je potrebné zoskupovať vzorky na základe určitých spoločných vlastností, pričom sa pre každý vzorok počíta tzv. *kľúč clustru*. Ten je reprezentovaný jedinečnou hodnotou (napríklad celočíselnou), ktorá sa uchováva do bufferu a neskôr sa využíva pre hľadanie unikátnych clustrov. Na určovanie clustrov je vhodné využiť 3D pozíciu vykresľovaného fragmentu nakoľko je zaručené že susedné fragmenty majú podobné pozície. Prakticky tak clustre predstavujú subfrustá tvoriace view space frustum. Keďže na vytvorenie týchto frúst je potrebné rozdeliť hĺbkový buffer na základe normalizovaných súradníc vytvorené clustre

nemajú rovnaké rozmery, pretože tie, ktoré sú bližšie blízkej rovine (*near plane*) budú úzke a naopak tie pri vzdialenej rovine(*far plane*) široké. V ideálnom prípade však majú clustre tvar kocky, čomu sa dá priblížiť exponenciálnym delením hĺbky.

5.3 Unikátne clustre

Je nutné predpokladať, že kľúče clustrov môžu byť pre niektoré clustre rovnaké. V tomto prípade je potrebné buffer kľúčov vhodne zredukovať, napríklad zotriedením (v prípade reprezentácie číselných hodnôt zoradením) a následnou komprimáciou rovnakých kľúčov. Jedná sa však o pomerne náročné operácie, preto je vhodné tento postup optimalizovať napríklad zotriedením na základe screen space tilov, na čo môžu byť využité prostriedky na paralelné výpočty (OpenCL, CUDA, Compute shadery).

5.4 Priradenie svetiel do clustrov

Cieľom tohto kroku je rovnako ako v tiled shadingu konštrukcia zoznamu svetiel tentokrát však pre jednotlivé clustry. Tiled shading v tomto kroku využíva pomerne náročný spôsob, kedy sa pri tvorbe mriežky svetiel testujú všetky svetlá na viditeľnosť vo view space, a následne všetky svetlá vo view space na prienik s tilami. Pri veľkých počtoch svetiel (desaťtisíce) je tento spôsob značne neefektívny a je potrebné využiť hierarchické štruktúry.

5.5 Výpočet osvetlenia

Osvetlenie a tieňovanie je opäť veľmi podobné tomu v tiled shadingu. Rozdiel spočíva v spôsobe prístupu k zoznamu svetiel pre špecifický fragment (zoznam svetiel tilu, do ktorého spadá daný fragment). Problémom však je to, že neexistuje priamy spôsob mapovania kľúča clustru na index do zoznamu unikátnych clustrov, preto je potrebné uchovávať si túto informáciu do osobitného bufferu už pri tvorbe zoznamu unikátnych kľúčov clustrov.
Prevzaté z [8].

Kapitola 6

Návrh, implementácia a experimenty

V tejto kapitole sú obsahnuté bližšie informácie o použitých knižniciach, programovacích jazykoch a konštrukciách vo výslednej aplikácii. Ďalej budú uvedené a zhodnotené výsledky experimentov prevádzaných na aplikácii. Taktiež v nej budú spomenuté implementačné podrobnosti niektorých kľučových, prípadne zaujímavých častí problematiky tejto práce.

6.1 OpenGL

OpenGL (*Open Graphics Library*)[3][12] je grafická knižnica navrhnutá ako aplikačné programové rozhranie (API) k akcelerovaným grafickým kartám resp. k celým grafickým subsystémom. Jej rozhranie je vytvorené tak aby bol použiteľná v takmer ľubovoľnom programovacom jazyku nezávisle od platformy, či správcu okien. Preto sa často spolu so samotnou knižnicou OpenGL používajú rôzne nadstavby ako `GLUT`, `GLFW` či iné, ktoré to umožňujú. Využíva sa najmä pri tvorbe rôznych 3D aplikácií, počítačových hier, vedeckých vizualizácií či iných. Z programátorského hladiska pracuje OpenGL ako stavový automat. To znamená, že při tvorbe programu je možné meniť niektoré vlastnosti grafických primitív prípadne celej zobrazovanej scény a tieto vlastnosti ostávajú v platnosti, pokiaľ ich programátor explicitne nezmení. V súčasnosti je dostupná verzia 4.4, ktorá bola vydaná v júli 2013.

6.2 GLSL

Náročnosť a hlavne množstvo výpočtov, ktoré je potrebné pri zobrazovaní grafickej scény je často príliš veľké a potreba uskutočňovať ich v reálnom čase si vyžadujú aby prebiehali priamo na grafickom akcelerátore. To umožnuje jazyk GLSL (*OpenGL Shading Language*) [13] založený na syntaxi programovacieho jazyka ANSI C, ktorý vývojárom ponúka vačšiu kontrolu nad grafickým pipelinom a zväčenie flexibility pri vykresľovaní. GLSL teda poskytuje možnosť vytvárať programy nazývané shadery, ktoré priamo ovplyvňujú výsledný obraz na obrazovke. Tieto programy sa vykonávajú paralelne pre jednotlivé pixely/fragmenty v závislosti od počtu shader jednotiek ktoré grafická karta poskytuje. Existuje niekoľko druhov shaderov a sice vertex shadery, fragment shadery, geometry shadery či compute shadery, ktoré boli predstavené s príchodom OpenGL 4.3.

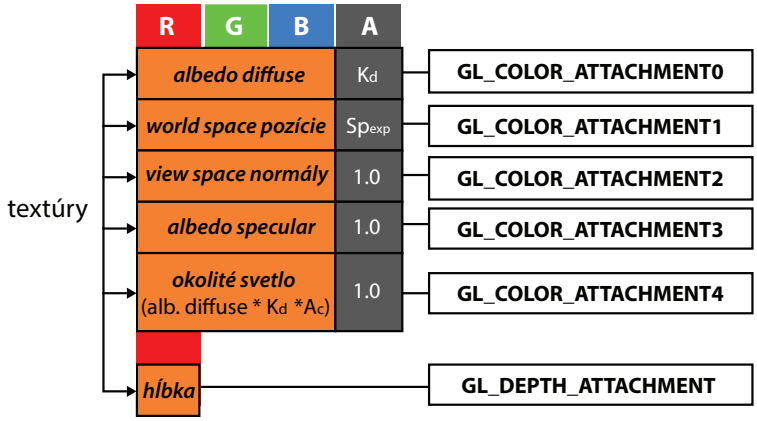
6.3 Pomocné knižnice

Kľúčovou knižnicou využitou v aplikácii je `GLFW`, ktorá zabezpečuje vytváranie aplikačného okna a spracovávanie vstupu z myši a klávesnice. Pre zvýšenie intuitívnosti a jednoduchosti ovládania aplikácie bola nasadená knižnica `AntTweakBar`, ktorá v kombinácii s `GLFW` umožňuje v aplikácii vytvárať jednoduché GUI ovládacie panely. Ďalej je využitá knižnica `GLEW` (*OpenGL Extension Wrangler Library*)zabezpečujúca efektívne run-time mechanizmy pre detekciu podporovaných rozšírení na zvolenej platforme. Výpočty matíc, vektorov a ostatných potrebných matematických prvkov zabezpečuje knižnica `GLM` (*OpenGL Mathematics*), ktorá poskytuje podobné údajové typy a funkcie ako sú použité v jazyku GLSL. Pre zjedodušené načítavanie 3D modelov ako sú napr. *wavefront .obj* a iné je využitá knižnica `Assimp` (*Open Asset Import Library*), pričom načítavanie textúr zabezpečuje knižnica `stb_image.c`.

6.4 Implementačné podrobnosti

6.4.1 Návrh G-bufferu

Štruktúru G-bufferu (viď obrázok 6.1) vo vytvorenej aplikácii tvorí celkovo 6 pripojených textúr, pričom 5 z nich uchováva informáciu o farbe a 1 slúži ako hĺbkový buffer. Formát uloženia farebných textúr je `GL_RGBA16F`, teda 4B pre každý kanál v RGBA modely. Pre hĺbkový buffer bol zvolený formát `GL_DEPTH_COMPONENT32F` kvôli väčšej presnosti.



Obr. 6.1: Implementovaný G-buffer. K_d reprezentuje odrazivý koeficient materiálu a Sp_{exp} (*shininess*) lesklosť povrchu, popísané v kapitole 3.1

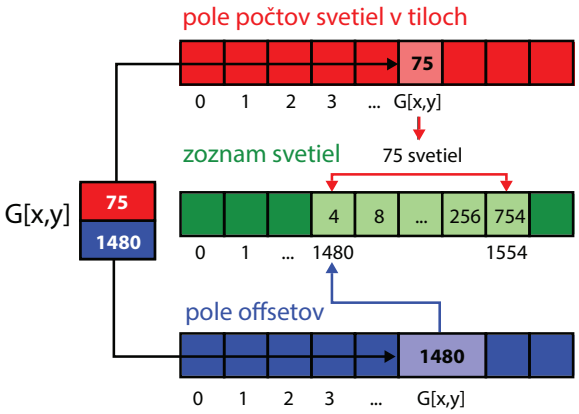
.

Možnou optimalizáciou tohto návrhu je neuchovávať informácie o pozícii vrcholu, pretože môže byť odvodená z hĺbky. Tento prístup však prináša značné navýšenie počtu výpočtov v shaderoch, a teda nemusí byť zrovna optimálne. Rovnako by bolo možné eliminovať textúru s informáciami o okolitom svetle keďže je možné ich získať až pri samotnom výpočte osvetlenia. To by však bolo možné iba v prípade tiled deferred shadingu, kde výpočet prebieha v *screen space*, zatiaľ čo v deferred shadingu by to viedlo k nesprávnej aplikácii osvetlenia, pretože ambientné osvetlenie by sa aplikovalo len v oblasti ktorú ovplyvňujú jednotlivé svetlá. V aplikácii sa však táto textúra využíva nielen na uchovanie okolitého svetla, ale aj na uchovanie celkového osvetlenia.

Keďže deferred shading a tiled deferred shading nevyžadujú odlišné metódy uchovania informácii o geometrii, obe techniky môžu v aplikácii využívať rovnaký G-buffer. Navyše pri použití depth pre-pass v technike tiled forward shading je možné využiť rovnakú textúru s informáciami o hĺbke akú využívajú deferred techniky.

6.4.2 Mriežka svetiel

Na implementáciu mriežky svetiel bol v aplikácii použitý procesor. Dáta sa teda uchovávajú v internej pamäti počítača a samotná mriežka tak prakticky pozostáva z niekoľkých dátových kolekcií, ktoré je v neskoršom štádiu potrebné uložiť do pamäti na grafickej karte. Na toto sú v aplikácii využité `Uniform Buffer Objects (UBOs)`, ktoré uchovávajú informácie o počtoch svetiel a o offsetoch, zatiaľ čo zoznamy svetiel sú uložené v `Texture Buffer Object` sériovo za sebou a vytvárajú tak jednotnú štruktúru – globálny zoznam svetiel. Tieto dáta sa však menia pri každom pohybe kamery, preto je potrebné dať túto informáciu grafickej karte. To zabezpečuje volanie posledný parameter funkcie `glBufferData(target, size, data, usage)`, ktorý je potrebné nastaviť na hodnotu `GL_DYNAMIC_COPY`.



Obr. 6.2: Štruktúry uchovávajúce informácie o svetlách zasahujúcich do jednotlivých tilov.

Pri konštrukcii mriežky sa v prvom kroku pre všetky existujúce svetlá spočíta screen space bounding quad(kapitola 4.3.2), pričom v tomto kroku dochádza k vyradeniu svetiel ktoré ležia mimo záberu kamery a nijak viditeľnú scénu neovyplyvňujú. Viditeľné svetlá sa uchovávajú do osobitného vektoru, pričom je najskôr pozícia každého transformovaná *view* maticou. Následne sa pre každé viditeľné svetlo spočítajú zasiahnuté tily.

V ďalšom kroku sa spočítajú celkové počty svetiel zasahujúce jednotlivé tily. V prípade, že je v aplikácii zapnutá optimalizácia hĺbkového bufferu sa zároveň v tejto chvíly vyraďujú svetlá na základe minimálnej a maximálnej hĺbky. Z vypočítaných hodnôt sa následne odvodia offsety do zoznamu svetiel a obe hodnoty sa ukladajú do osobitných polí.

V tomto momente poznáme všetko čo potrebujeme pre konštrukciu zoznamov svetiel v tiloch. Jeden zoznam svetiel nie je teda ničím iným ako postupnosťou celočíselných (`int`) hodnôt. Preto sa v aplikácii tieto zoznamy spájajú do jedného gobálneho. Na záver sa z polí počtov a offsetov vytvoria spomínané UBO a z globálneho zoznamu svetiel TBO.

6.4.3 Optimalizácia hĺbky

Pre zvýšenie efektivity výpočtov osvetlenia tiled shading techník bola v aplikácii implementovaná optimalizácia hĺbkové bufferu – výpočet minimálnej a maximálnej hranice hĺbky pre každý tile (viď kapitola 4.2). Na *downsampling* sa využíva samostatný framebuffer ku ktorému je pripojená textúra pre výsledné konzervatívne hĺbky (veľkosť textúry `rozlíšenie / veľkosť tilu`) a jednoduchý fragment shader. Každá invokácia fragment shaderu spočíta potrebný `offset` a `rozsah` fragmentov, z ktorého sa budú vzorkovať hodnoty hĺbky z hĺbkového bufferu.

V tiloch na hraniciach okna aplikácie však môže dochádzať k nadbytočným výpočtom v závilosti na zvolenom rozlíšení. Hraničné tily totiž nemusia mať zvolenú veľkosť a môžu spadať do intervalu ⟨`0,TILE_DIM`⟩, preto je potrebné rozsah tilu obmedziť na rozlíšenie okna (ukážka 6.1, riadok 16).

```
1  uniform sampler2D depthTex;
2  uniform mat4 inverseProjectionMatrix;
3
4  out vec4 resultMinMax;
5
6  float convertToSS(float depth){
7     vec4 pt = inverseProjectionMatrix * vec4(0.0, 0.0, 2.0 * depth - 1.0, 1.0);
8     return pt.z/pt.w;
9  }
10
11 void main(){
12    ivec2 resolution = ivec2(WIDTH,HEIGHT);
13
14    vec2 minMax = vec2(1.0f, -1.0f);
15    ivec2 tileOffset = ivec2(gl_FragCoord.xy) * ivec2(TILE_DIM, TILE_DIM);
16    ivec2 tileRange = min(resolution, offset + ivec2(TILE_DIM, TILE_DIM));
17
18    for (int j = tileOffset.y; j < tileRange.y; ++j){
19       for (int i = tileOffset.x; i < tileRange.x; ++i){
20          float d = texelFetch(depthTex, ivec2(i, j), 0).x;
21
22          if (d < 1.0){
23             minMax.x = min(minMax.x, d);
24             minMax.y = max(minMax.y, d);
25          }
26       }
27    }
28    //normalised depth
29    vec2 result = minMax;
30
31    //screen space depth
32    minMax = vec2(convertToSS(minMax.x), convertToSS(minMax.y));
33    resultMinMax = vec4(minMax,result);
34 }
```

Listing 6.1: Pseudokód pre výpočet osvelenia v tiled forward shadingu

6.5 Experimenty

Výsledná aplikácia bola vyvíjaná s cieľom dosiahnuť dobrých výsledkov, jednak z hľadiska množstva výpočtov ale aj realistickosti výsledného vzhľadu vykreslenej scény. Účelom experimentov v tejto kapitole je porovnať implementované metódy (deferred shading, tiled deferred shading, tiled forward shading) a poukázať na ich výkonnosť, či už vzhľadom na počet statických svetiel v scéne, počtu pixelov na ktorých sú prevádzané výpočty osvetlenia alebo použitie optimalizácií.

Experimenty boli prevádzané na voľne dostupnom modely Crytek Sponza[1] v ktorom boli náhodne rozmiestnené zdroje bodových svetiel popísaných v kapitole 2.4.3. Pri každom experimente sa v scéne pohybovala kamera po rovnakej dráhe, pričom boli súčasne zaznamenávané FPS(*Frames Per Second*) s periódou $0.1s$. Za účelom získania presných dát bola pri všetkých experimentoch vypnutá natívna podpora vertikálnej synchronizácie (*vertical sync*). Pokiaľ nie je v texte prípadne v popise grafu uvedené inak, experimenty boli prevádzané v aplikačnom okne s HD rozlíšením (1280×720 pixelov), veľkosti tilu 32×32 s použitím optimalizácie hĺbky. V prípade použitia tejto optimalizácie bude uvedený tag `MM`.

6.5.1 Architektúra

Vývoj aplikácie a väčšina experimentov bolo uskutočnenených na architektúre 1[2] uvedenej v tabuľke 6.1. Aplikácia bola vyvíjaná najmä s cieľom použiteľnosti na grafických kartách od spoločnosti AMD, bola však upravená pre použiteľnosť aj na grafických kartách NVIDIA.

		Architektúra 1	**Architektúra 2**	**Architektúra 3**
Procesor		Intel(R) Core(TM) i5-2410M CPU @ 2.30Ghz 2.29 GHz	Intel(R) Core(TM) i5-3210i 2,5ghz	Intel(R) Core(TM) i7-3630QM CPU @ 2.40Ghz 2.395 MHz
RAM		4GB	6GB	12GB
HDD		Samsung SSD 840 EVO 120GB	Seagate HDD 1000GB 5400ot	SanDisk SSD U100 120GB
Operačný systém		Windows 8.1	Windows 7	Windows 8.1
Grafická karta	**Typ**	AMD Mobility Radeon HD 6490 1GB	NVIDIA GeForce GT630M 2GB	NVIDIA GeForce GTX 660M 2GB
	Architektúra	Terascale 2	Fermi	Kepler
	Počet shaderov	160	96	384
	Rýchlosť jadra	700 / 750 / 800	672	835
	Rýchlosť shaderov	700 / 750 / 800	1344	835
	Pamäťová zbernica	64bit	128bit	128bit

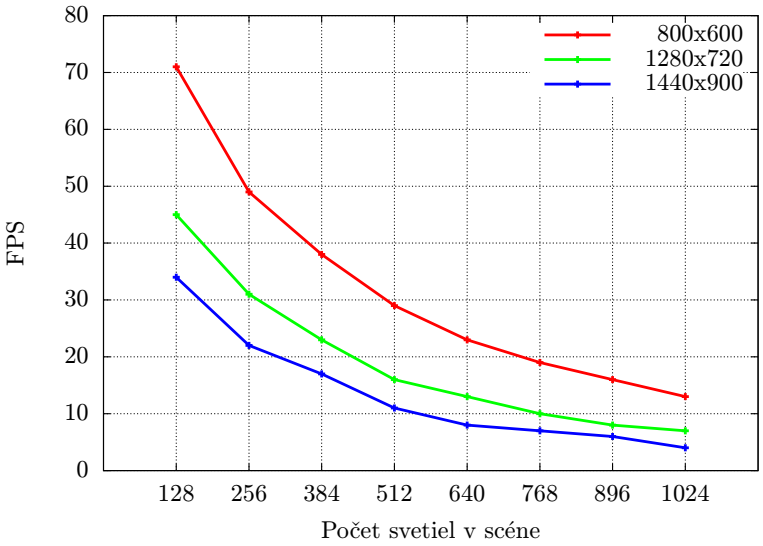
Tabuľka 6.1: Tabuľka architektúr na ktorých boli prevádzané experimenty.

[1] Dostupný na http://www.crytek.com/cryengine/cryengine3/downloads

[2] Pokiaľ nie je v popise grafu explicitne definovaná architektúra, na daný experiment bola využitá architektúra 1.

6.5.2 Výkonnosť deferred shadingu

Technika deferred shading bola spočiatku experimentálne testovaná na použitie pri rôznych počtoch svetiel. Pre väčšiu komplexnosť zobrazenia zaťaženia sa taktiež testovala na výkonnosť pri rôznom rozlíšení, kedy sa značne menili počty ovplyvnených fragmentov pre ktoré bolo potrebné počítať osvetlenie.



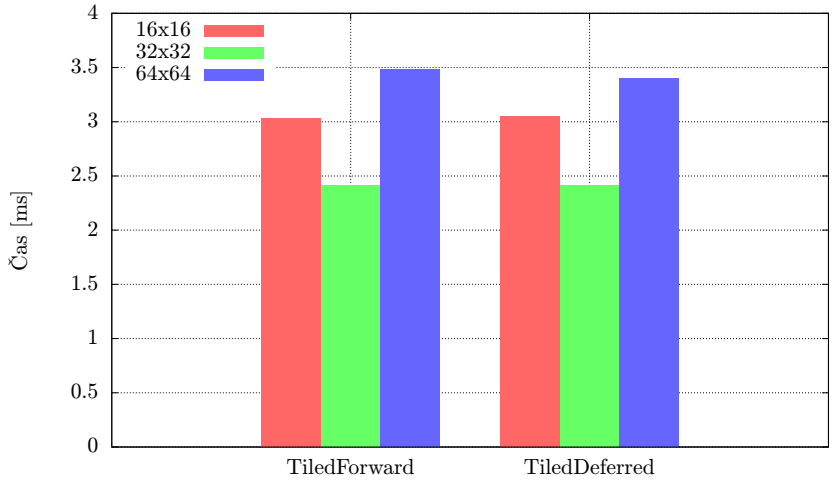
Obr. 6.3: Výkonnosť deferred shadingu. Namerané hodnoty FPS pri pohybe kamery v scéne boli pre každé meranie spriemerované.

Z nameraných hodnôt, ktoré sú vynesené v grafe 6.3 môžme odvodiť záver, že pri použití niekoľkých stovák svetiel výpočetná náročnost značne klesá ale pri počte svetiel blížiacemu sa 1000 sa výpočetná náročnosť exponenciálne znižuje. Avšak pri vysokom počte svetiel, klesá úroveň do veľmi nizkych hodnôt, čo má viditeľný dopad na plynulosť aplikácie. Zároveň je z grafu možné vyčítať, že závislosť náročnosti výpočtov na rozlíšení klesá zo zväčšujúcom sa počtom fragmentov takmer lineárne.

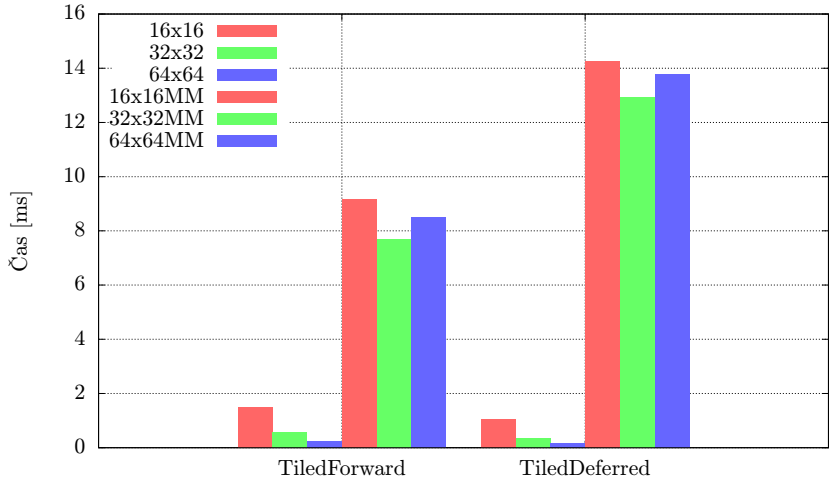
6.5.3 Výkonnosť tiled deferred shadingu

Pred samotným experimentovaním s tiled technikami bolo potrebné zistiť optimálny rozmer tilu pre konkrétnu implementáciu aplikácie. Ako bolo spomínané v kapitole 4.3.3 závisí na viacerých faktoroch no mnohé zdroje uvádzajú informáciu, že optimálna veľkosť tilu v technike tiled shading je 16×16 prípadne 32×32. Cieľom tohto experimentu bolo overiť pravdivosť tohto tvrdenia a v prípade jeho vyvrátenia zistiť veľkosť tilu, pre ktorý je výpočet osvetlenia v zvolenej scéne najoptimálnejší. Pre testovanie boli zvolené veľkosti uvedené hodnoty ako aj rozmer 64×64.

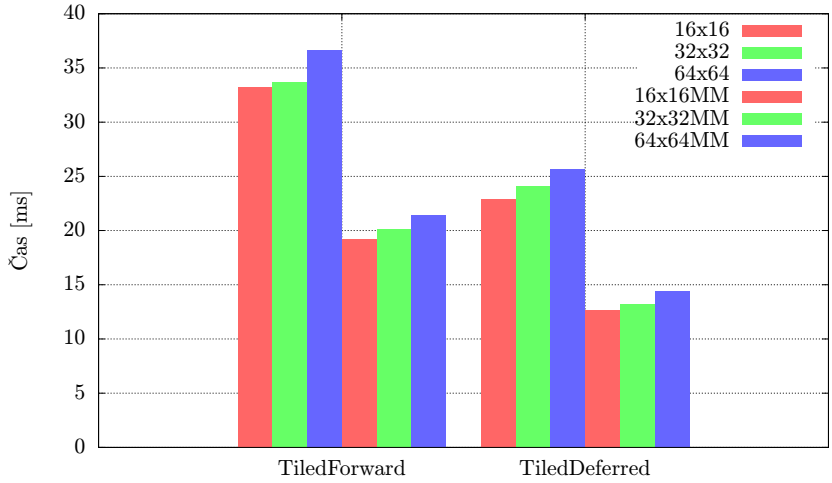
Experiment spočíval v meraní času kľúčových častí algoritmu (downsampling hĺbkového bufferu, zostrojenie mriežky svetiel, výpočet osvetlenia) s/bez optimalizácie hĺbky. Keďže downsampling a výpočet osvetlenia prebiehajú v shaderoch meranie času prebiehalo pomocou `OpenGL Queries`. Naopak zostrojenie mriežky svetiel prebieha na CPU, preto bolo potrebné implementovať čo možno najpresnejší merač času spracovania inštrukcií programu. Na to bola využitá sada funkcií `QueryPerformanceCounter` a `QueryPerformanceFrequency`. Namerané hodnoty boli spriemerované a vynesené do grafov na obrázku 6.4.



(a) Downsampling



(b) Konštrukcia mriežky svetiel



(c) Výpočet osvetlenia

Obr. 6.4: Doba výpočtu kľúčových častí tiled shadingu. Hodnoty v stĺpcoch zľava doprava odpovedajú legende v smere zhora nadol.

Počas experimentu sa taktiež zaznamenávali FPS zobrazené v tabuľke 6.2. Z grafu 6.4a je vidieť, že pri downsamplingu dominuje rozmer tilu 32×32 v oboch technikách. Vo všetkých prípadoch sú však hodnoty pomerne nízke a na celkový výkon tento krok preto nemá veľký vplyv.

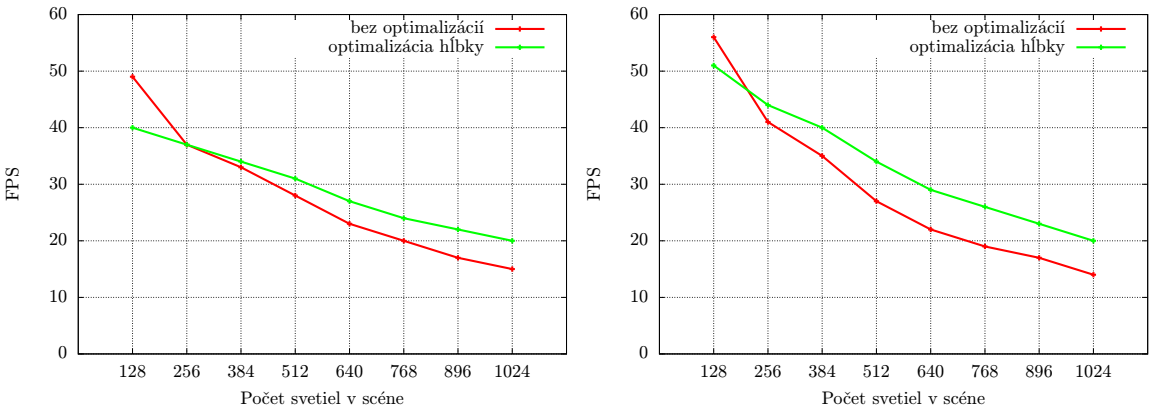
	Technika			
Tile	**TiledForward**	**TiledForward(MM)**	**TiledDeferred**	**TiledDeferred(MM)**
16x16	18,5	24,1	20,6	24,5
32x32	18,7	24,9	20,7	26,5
64x64	16,6	21,7	19,6	23,2

Tabuľka 6.2: FPS namerané počas experimentu

Väčšiu váhu majú výpočetné nároky potrebné na konštrukciu mriežky svetiel (graf 6.4b) spolu s osvetlovaním (graf 6.4c). V prípadoch kedy nebola použitá optimalizácia hĺbky je vidieť, že doba konštrukcia mriežky svetiel sa pohybuje okolo 1 milisekundy. V tomto prípade jednoznačne dominuje veľkosť tilu 64×64, pretože použitie veľkých tilov vedie k zníženiu zložitosti výpočtov cyklických operácií. S použitím optimalizácie hĺbky však doba konštrukcie mriežky svetiel niekoľkonásobne narastá, pretože dochádza k presnejšiemu vyraďovaniu svetiel(kapitola 4.3) vzhľadom na hranice tilu. V tomto prípade je rovnako ako pri downsamplingu dominantná veľkosť tilu 32×32.

Výpočet osvetlenia je bez zavedenia optimalizácie hĺbky pomerne náročný nakoľko doba výpočtu osvetlenia pre jeden snímok sa pohybuje okolo hodnoty 35 milisekúnd. S použitím optimalizácie sa počet viditeľných svetiel značne redukuje čo v konečnom dôsledku vedie v oboch prípadoch k takmer dvojnásobnému urýchleniu výpočtov a značnému zvýšeniu výkonu. V oboch prípadoch v tomto kroku však dominuje veľkosť tilu 16×16, pravdepodobne vďaka optimálnejšej paralelizácie shaderov.

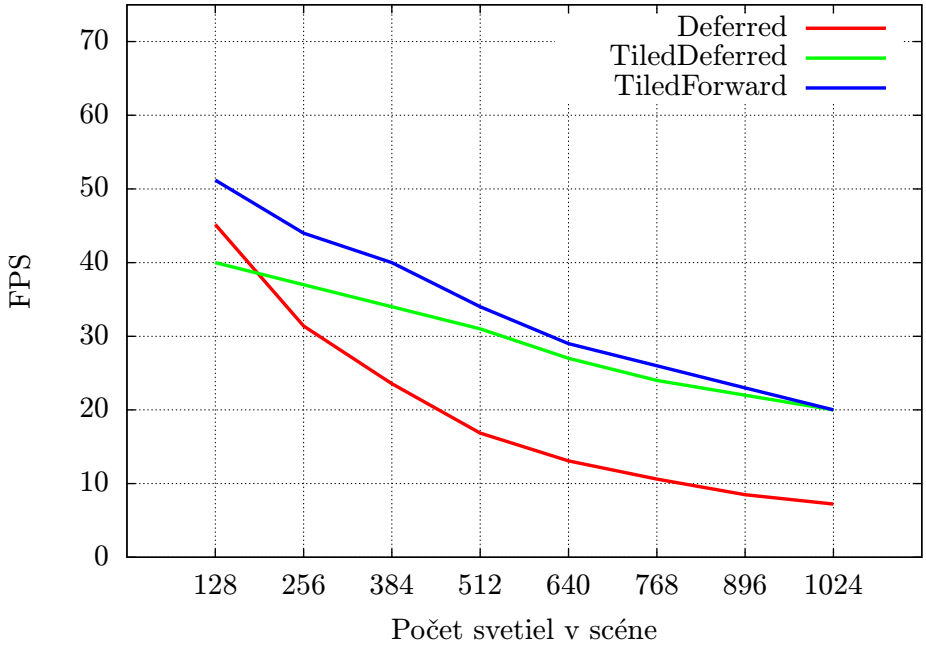
Optimálna veľkosť tilu pre techniku tiled shading je taká, pre ktorú je výpočet osvetlenia čo najnižší nakoľko na efektivitu vplýva najkritickejšie. Zároveň je potrebné dosiahnuť čo najnižších časov konštrukcie mriežky svetiel. Porovnaním odpovedajúcich si hodnôt v grafoch 6.4b a 6.4c môžme tvrdiť, že vhodnou strednou cestou je veľkosť tilu 32×32.



(a) Tiled Deferred Shading

(b) Tiled Forward Shading

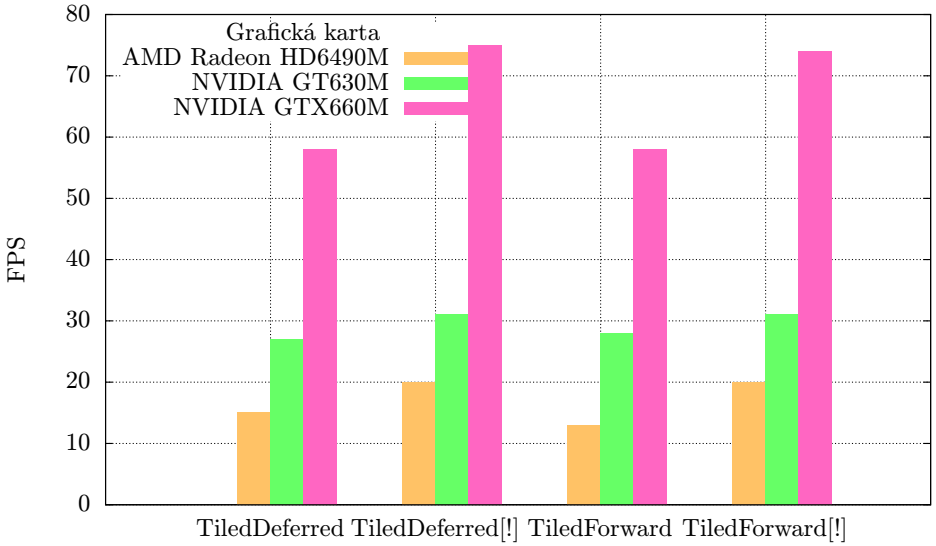
Obr. 6.5: Tiled shadingu pri rôznych počtoch svetiel, s a bez použitia optimalizácie hĺbky



Obr. 6.6: Porovnanie výkonnosti implementovaných techník.

Rovnako ako pri klasickom deferred shadingu aj výkonnosť tiled shadingu bola testovaná pri rôznych počtoch zdrojov svetiel. Výsledky tohto merania zobrazujú grafy 6.5a a 6.5b. Porovnanie všetkých troch implementovaných metód znázorňuje graf 6.6.

Okrem toho sa aplikácia testovala aj na výkonnejších grafických kartách od spoločnosti NVIDIA na architektúrach 2 a 3 uvedených v tabuľke 6.1. Toto meranie znázorňuje graf 6.7.



Obr. 6.7: Porovnanie výkonnosti tiled techník na rôznych grafických kartách. Tag [!] za názvom techniky značí použitie optimalizácie hĺbky(minimum a maximum hĺbky).

Kapitola 7

Závěr

Cieľom tejto bakalárskej práce bolo vytvoriť aplikáciu, ktorá sa zameriava na efektívny real-time výpočet svetla v grafických scénach s veľkým počtom svetiel. Veľká časť práce sa zameriava na techniky deferred shading a tiled shading [9], ktorý bol taktiež rozšírený o optimalizáciu hĺbkového bufferu výpočtom minimálnych a maximálnych hodnôt pre každý tile.

Implementované metódy dosahujú dobré výsledky na grafických kartách s priemerným výkonom, pri použití tisíc svietiel zatiaľ čo na výkonnejších GPU sa jedná o použiteľnosť niekoľkých tísíc zdrojov svetiel. Najvýkonnejšou technikou pri počte aktívnych svetiel maximálne 1000 svetiel bol tiled forward shading, no pri vyšších počtoch o niečo viac dominoval tiled deferred shading.

Vytvorená aplikácia taktiež umožňuje meniť techniku výpočtu osvetlenia za behu čo pri vhodnej kombinácii s aktuálnymi parametrami zvyšuje efektivitu výpočtov. Väčšina potrebných výpočtov osvetlenia prebieha na grafickej karte s výnimkou zostavenia mriežky svetiel a priradenia svetiel do nej. Na tieto operácie sa využíva CPU. Aplikácia bola navrhnutá pre použiteľnosť na grafických kartách s podporou minimálne OpenGL verzie 3.3. Pôvodným cieľom bolo postaviť aplikáciu na výpočtoch s využitím compute shaderov, ale kvôli chybnej podpore tejto súčasti na vývojovej architektúre a množstve komplikácií, ktoré obchádzanie čiastočnej podpory spôsobovalo bolo od tohto cieľu upustené.

Ako rozšírenie tejto práce sa teda ponúka možnosť implementácie konštrukcie mriežky svetiel a výpočtu osvetlenia s využitím paralelných výpočtov prostredníctvom compute shaderov, ktoré by viedlo k radikálnemu zvýšeniu výkonnosti aplikácie. Taktiež by bolo vhodné implementovať vyraďovanie svetiel metódou výpočtu frusta pre každý tile a optimalizovať túto metódu technikou 2.5D Culling [5] pre porovnanie výkonnosti s implementovanou technikou v tejto práci. S cieľom niekoľkonásobného zvýšenia výkonnosti je taktiež vhodné implementovať techniku clustered shading [8], ktorej použitie by mohlo viesť k použiteľnosti stoviek tisíc svetiel. Tieto rozšírenia si však vyžadujú výkonný grafický adaptér s podporou najnovších technológií.

Literatúra

[1] Andersson, J.: DirectX Rendering in Battlefield 3. Presentation, 2011.

[2] Assarsson, U.; Möller, T.: Optimized View Frustum Culling Algorithms for Bounding Boxes. *J. Graph. Tools*, ročník 5, č. 1, Leden 2000: s. 9–22, ISSN 1086-7651, doi:10.1080/10867651.2000.10487517.
URL http://dx.doi.org/10.1080/10867651.2000.10487517

[3] Board, O. A. R.; Shreiner, D.; Woo, M.; aj.: *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1*. Addison-Wesley Professional, 6 vydání, 2007, ISBN 0321481003, 9780321481009.

[4] Fuchs, H.; Poulton, J.; Eyles, J.; aj.: Pixel-planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-enhanced Memories. *SIGGRAPH Comput. Graph.*, ročník 23, č. 3, jul 1989: s. 79–88, ISSN 0097-8930, doi:10.1145/74334.74341.
URL http://doi.acm.org/10.1145/74334.74341

[5] Harada, T.: A 2.5D Culling for Forward+. 2012: s. 18:1–18:4, doi:10.1145/2407746.2407764.
URL http://doi.acm.org/10.1145/2407746.2407764

[6] Lagarde, S.: Adopting a physically based shading model [online]. `http://seblagarde.wordpress.com/2011/08/17/hello-world/`, 2011-8-17 [cit. 2014-26-3].

[7] Lengyel, E.: *Mathematics for 3D Game Programming and Computer Graphics, Second Edition*. Rockland, MA, USA: Charles River Media, Inc., 2003, ISBN 1584502770.

[8] Olsson, M. A. U., Ola; Billeter: Clustered Deferred and Forward Shading. 2012, doi:10.2312/EGGH/HPG12/087-096.
URL http://dx.doi.org/10.2312/EGGH/HPG12/087-096

[9] Olsson, O.; Assarsson, U.: Tiled Shading. *Journal of Graphics, GPU, and Game Tools*, ročník 15, č. 4, 2011: s. 235–251, doi:10.1080/2151237X.2011.621761.
URL http://www.tandfonline.com/doi/abs/10.1080/2151237X.2011.621761

[10] Olsson, O.; Billeter, M.; Assarsson, U.: Tiled and Clustered Forward Shading. In *SIGGRAPH '12: ACM SIGGRAPH 2012 Talks*, New York, NY, USA: ACM, 2012.

[11] Phong, B. T.: Illumination for Computer Generated Pictures. *Commun. ACM*, ročník 18, č. 6, jun 1975: s. 311–317, ISSN 0001-0782, doi:10.1145/360825.360839.
URL http://doi.acm.org.ezproxy.lib.vutbr.cz/10.1145/360825.360839

[12] Sellers, G.; Wright, R. S.; Haemel, N.: *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional, 6 vydání, 2013, ISBN 0321902947, 9780321902948.

[13] Wolff, D.: *OpenGL 4.0 Shading Language Cookbook*. Packt Publishing, 2011, ISBN 1849514763, 9781849514767.