MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Použitie OpenGL v jazyku Java

BAKALÁRSKA PRÁCA

Miloš Selečéni

Brno, jeseň 2013

Prehlásenie

Prehlasujem, že táto bakalárska práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Vedúci práce: RNDr. Adam Rambousek

Poďakovanie

V prvom rade sa chcem poďakovať svojmu vedúcemu bakalárskej práce RNDr. Adamovi Rambousekovi za rady a odborné vedenie pri spracovaní textovej časti práce. Poďakovanie patrí aj môjmu technickému vedúcemu Ing. Pavlovi Tišnovskému za odborné konzultácie pri tvorbe praktickej časti. V neposlednom rade ďakujem aj mojej rodine a priateľom za pomoc a podporu.

Zhrnutie

V bakalárskej práci sa autor zaoberal vývojom grafických akcelerátorov a moderným programovaním v OpenGL. V rámci práce bola implementovaná demo aplikácia v jazyku Java s využitím OpenGL knižnice LWJGL. Aplikácia demonštruje základné postupy a techniky používané pri vykresľovaní 3D objektov.

Kľúčové slová

OpenGL, TWL, shader, GLSL, DirectX, GPU, CPU

Obsah

1 **História grafických akcelerátorov** 3
1.1 *Grafické akcelerátory IBM* 3
1.2 *Grafické akcelerátory ATI* 4
1.3 *Grafické akcelerátory Nvidia* 5
1.4 *Grafické akcelerátory 3dfx* 6
2 **OpenGL** 7
2.1 *Vznik OpenGL* 7
2.2 *Popis OpenGL* 8
2.3 *OpenGL vs. DirectX* 8
2.4 *Fixná a programovateľná pipeline* 9
2.5 *Moderné OpenGL* 10
2.6 *OpenGL v Jave* 10
3 **Demo aplikácia** 12
3.1 *Popis dema* 12
3.2 *Tvorba GUI* 13
3.2.1 Knižnica TWL 14
3.2.2 Popis GUI 16
3.3 *3D modely* 18
3.4 *Osvetľovací model* 19
3.4.1 Textúry 21
Ambient Occlusion 21
3.5 *2D efekty* 22
3.5.1 Detekcia hrán 23
3.5.2 Rozmazanie obrazu 24
3.5.3 Odraz v podlahe 24
3.6 *Shader* 25
3.6.1 Vertex shader 26
3.6.2 Tessellation shader 26
3.6.3 Geometry shader 27
3.6.4 Fragment shader 27

4 **Záver** 28
A **Obsah priloženého CD** 29

Zoznam obrázkov

3.1 Úvodné nastavenia v aplikácii. *13*
3.2 Rôzne úrovne antialiasingu. *14*
3.3 Hlavné okno aplikácie. *15*
3.4 Módy vykreslenia voliteľné v aplikácii. *17*
3.5 Uhol svetla (svetelný kužeľ). *17*
3.6 Jednotlivé zložky farby *20*
3.7 Phongov svetelný model. *21*
3.8 Zap./Vyp. Ambient Occlusion *22*
3.9 Laplace filter *24*
3.10 Horizontal Sobel filter *24*
3.11 Shader pipeline. *26*

Úvod

Vzhľadom na zvyšujúce sa požiadavky používateľov, mnoho tvorcov moderných aplikácií kladie čoraz väčší dôraz na grafické rozhranie, čoho dôsledkom sú stále vyššie nároky na hardware umožňujúci efektívnu prácu s grafikou, špeciálne na grafický čip. Vylepšovanie technológií používaných v grafických čipoch je preto jednou z najdôležitejších oblastí súčasného vývoja hardwarových komponentov. Spolu s vývojom grafických čipov je potrebné vylepšovať aj programové knižnice, ktoré umožňujú tvorcom aplikácií naplno využívať možnosti hardwaru.

Programová knižnica OpenGL patrí medzi jednu z najpoužívanejších programových knižníc určených na tvorbu 2D a 3D aplikácií. Nie je žiadnym prekvapením, že súčasná moderná verzia OpenGL sa podstatne líši od prvých verzií, podobne ako sa aj prvé grafické čipy líšia od tých používaných dnes. V našej práci sa zameriavame práve na moderný spôsob programovania v OpenGL, špeciálne na možnosti použitia OpenGL v jazyku Java.

Práca je rozdelená do troch kapitol. V prvej kapitole stručne popisujeme históriu grafických akcelerátorov, od prvých pokusov firmy IBM, až po moderné verzie vyvinuté spoločnosťami Nvidia, ATI, či 3dfx.

Druhá kapitola sa zaoberá vznikom OpenGL, posunom od fixnej pipeline k modernej alternatíve vo forme programovateľnej pipeline. Priestor dostáva aj DirectX, knižnica vyvinutá spoločnosťou Microsoft, ktorá je dlhodobým konkurentom otvorenej knižnice OpenGL. Popisujeme ich vzájomné súperenie až po súčasnosť. V tejto kapitole tiež uvádzame projekty, ktoré sa zaoberajú prepojením OpenGL s jazykom Java. Detailnejšie sa zameriavame na knižnicu LWJGL, ktorú používame v praktickej časti práce.

Tretia kapitola je venovaná praktickej časti, v rámci ktorej tvoríme demo aplikáciu demonštrujúcu prácu s OpenGL v jazyku Java. De-

tailne popisujeme nami vytvorenú aplikáciu, vrátane techník a postupov použitých pri implementácii. Dokumentujeme tvorbu grafického rozhrania, fungovanie osvetľovacieho modelu, textúrovanie a použité 2D efekty.

Kapitola 1

História grafických akcelerátorov

Grafický akcelerátor je neodmysliteľnou súčasťou každého moderného počítača, inteligentného telefónu či tabletu. Jeho výkon sa v poslednom desaťročí mnohonásobne zvýšil a je viac než isté, že to tak bude aj naďalej. No nebolo to vždy tak. Prvé počítače nemali žiadny grafický akcelerátor. O vykresľovanie dát ako aj o všetky grafické výpočty sa staral CPU[1]. Postupom času si spoločnosti ako IBM, ATI, NVIDIA a 3dfx začali uvedomovať, že CPU je zbytočne zaťažovaný týmito výpočtami, a preto začali vyrábať vlastné grafické akcelerátory. V 90. rokoch minulého storočia sa začali vyrábať prvé grafické akcelerátory, ktoré sa trochu podobali tým dnešným.

1.1 Grafické akcelerátory IBM

IBM PC bol prvý masovo predávaný počítač s MDA[2] a CGA[3] grafickým akcelerátorom, lacnejšia verzia PC 5150 bola predávaná s MDA, tá drahšia s CGA čipom. Čip v oboch kartách bol od firmy Motorola a mal frekvenciu 1 až 2 MHz. Pamäť v MDA bola 4 kB a zobrazovala len text, CGA mala 16 kB pamäť a dokázala spracovať a zobrazovať už aj grafiku. Herný priemysel vždy určoval a stále určuje potreby a požiadavky kladené na grafické karty, a keďže hranie hier na týchto kartách bolo značne obmedzené, či už grafickým módom alebo farebnou paletou, obľúbené 8-bitové Atari počítače ich hravo porážali. V polovici osemdesiatych rokov minulého storočia sa objavila konkurenčná grafická karta Hercules od firmy Hercules Computer Technology, ktorá bola spojením MDA a CGA. Karta dokázala

1. CPU - hlavný procesor počítača, anglicky *Central Processing Unit*
2. MDA - *Monochrome Display Adapter*
3. CGA - *Color Graphics Adapter*

emulovať grafiku podobne ako CGA a práca v textovom režime bola podobná s MDA. Preto bola po svojom uvedení na trh veľmi obľúbená.

Nástupcom CGA (MDA) kariet bola karta EGA[4], ktorá zaručovala spätnú kompatibilitu. EGA ale stále nemala vlastnosti, ktoré programátori v tej dobe očakávali. Karta nemala rozsiahlejšiu farebnú paletu a nepodporovala *spirity*, ktoré sa používali napríklad na zobrazenie kurzoru myši. Hoci mala karta veľa nedostatkov, zmenila pohľad na osobný počítač, z počítača určeného na prácu s textom na počítač efektívne pracujúci s grafikou. Grafické CAD aplikácie alebo operačný systém MS Windows sa vďaka tejto karte stali použiteľnými v praxi. Ďalšou veľmi významnou kartou bola karta VGA[5]. Bola to prvá karta použitá v počítačoch IBM PC/2. Grafická karta mala 256 kB pamäť a podporovala rozlíšenie 720×400 obrazových bodov. Asi najznámejšou funkciou podporovanou v grafickom režime VGA bol horizontálny a vertikálny posun obrazu.

Začiatkom roku 1984 sa situácia zmenila, keď IBM predstavila PGC. Grafický čip PGC bol osadený čipom Intel 8088 a na svoju dobu bol veľmi pokročilý. Podporoval rozlíšenie 640×480 a mal 320 kB pamäť s 256 farbami. Poskytoval 2D a 3D akceleráciu, no pre svoju vysokú cenu si ho mohli dovoliť iba veľké firmy. Jeho nástupca IBM 8514/A sa stal veľmi obľúbeným a predávaným, hlavne kvôli akcelerácii CAD aplikácií a podpore pre Windows 3.

1.2 Grafické akcelerátory ATI

V roku 1991 spoločnosť ATI využila úspech IBM 8514/A a uviedla na trh jeho klon pod menom Mach 8. Mach 8 sa líšil v použitých zberniciach a pamäťových moduloch, ktoré mohli byť veľkosti 512 kB alebo 1 MB. O rok neskôr vyšiel Mach 32, ktorý priniesol farebnú hĺbku až 24-bitov, rozlíšenie 1280×1024 a pamäť s veľkosťou 2 MB. Priekopníkom v oblasti akcelerácie videa sa stal Mach 64 GT, ktorý prišiel na trh o dva roky neskôr. Mal nové 3D jadro a bola to prvá séria čipov 3D Rage. Veľkosť pamäte 2 MB bola dostačujúca pre 2D grafiku, no pre 3D to bolo málo. Taktiež im chýbala filtrácia textúr.

4. EGA - Enhanced Graphic Adapter
5. VGA - Video Graphics Array

ATI 3D Rage Pro bola predstavená v roku 1997. Bola to prvá karta, ktorá podporovala AGP[6] zbernicu. Karta priniesla akceleráciu vykresľovania hmly a *specular lighting*. Bohužiaľ to stále nestačilo na grafické karty od 3dfx a Nvidia. V nasledujúcich rokoch vyšli karty ako Rage 128 a Rage 128 Pro, ktoré priniesli podporu pre OpenGL a DirectX 6.0. ATI sa darilo na trhu s mobilnými grafickými kartami, kde pôsobila od roku 1996 s Rage LT. Dominovala predovšetkým akcelerovaným prehrávaním DVD a nízkou cenou.

Moderné grafické karty od ATI, akou je napríklad R9 290X, majú neporovnateľne vyšší výkon ako tie spomenuté vyššie. Pre porovnanie R9 290X má 2816 procesorových jednotiek, každé s frekvenciou 1 GHz, a pamäť s veľkosťou 4 GB.

1.3 Grafické akcelerátory Nvidia

Prvý grafický akcelerátor vydaný spoločnosťou Nvidia v roku 1995 niesol názov NV1. Karta využívala Quadro Texture Map systém. Nanešťastie to bolo v rozpore s Direct 3D API, ktoré predstavil Microsoft a karta sa vôbec nepredávala. O dva roky neskôr v roku 1998 Nvidia prišla s RIVA 128. Karta mala vysokú úroveň 2D akcelerácie a miestami prekonávala favorita 3dfx Voodoo Graphics. Chýbala jej však podpora pre OpenGL.

Koncom roku 1998 vyšla karta RIVA TNT. Skratka TNT znamenala TwiN Texel a označovala druhú *pixel pipeline* a teda až dvojnásobný výkon, ktorý bol porovnateľný s Voodoo 2 od 3dfx. Karta mala kvalitnejší obraz ako Voodoo 2 s hĺbkou až 32-bitov. Koncom roku 1999 Nvidia predstavila GeForce 256. S ňou prišla podpora pre OpenGL a DirectX 7.0. Spolu s GeForce vznikol nový rad kariet Quadro. Oba rady mali rovnaký čip s výnimkou toho, že Quadro bolo špecializované na CAD aplikácie. V roku 2000 Nvidia odkúpila časť krachujúceho 3dfx. Bol to jeden z najvýznamnejších krokov Nvidie, ktorý ju posunul až do dnešného postavenia na trhu.

V súčasnosti pribudol k radom GeForce a Quadro aj Tesla, ktorý je zameraný na čo najvyšší výkon. Vlajkovou loďou radu Tesla je Tesla K40 s viac než 2880 jadrami s frekvenciou 880 MHz a pamäťou až 12 GB.

6. AGP - Accelerated Graphics Port

1.4 Grafické akcelerátory 3dfx

Prvá karta od 3dfx vyšla pod názvom Voodoo Graphics v roku 1996. Karta síce nepodporovala OpenGL ani DirectX, ale mala vlastné API Glide. Po vyvinutí hry Tomb Raider firmou Core Design pre Voodoo Graphics sa 3dfx rozhodlo pridať podporu pre OpenGL. Tým sa sprístupnila vtedy veľmi obľúbená hra Quake a všetky hry založené na jeho jadre. Voodoo sa stalo lídrom na trhu a nemalo konkurenciu. Voodoo Graphics bol len 3D akcelerátor, čo znamenalo, že k nemu bola potrebná ďalšia 2D karta, napríklad VGA. Po úspešnom predstavení Voodoo sa rozbehol ďalší projekt s názvom *Rampage*. Bohužiaľ dokončenie projektu sa stále odkladalo a nakoniec sa 3dfx rozhodlo projekt úplne zrušiť a venovať sa radšej novej generácii nazvanej Voodoo 2. Po predstavení Voodoo 2 v roku 1998 opäť 3dfx ovládlo trh a dostalo sa na vrchol. Základný takt procesoru bol 100 MHz a 8 až 12 MB pamäte nemali konkurenciu. V rovnakej dobe bola predstavená nová technológia SLI[7]. Jednalo sa o hardwarové prepojenie dvoch grafických kariet pomocou krátkeho káblu, ktoré umožňovalo zvýšenie výkonu až na dvojnásobok. Ďalšou úspešnou kartou mala byť Voodoo Banshee, ktorá bola lacnejšou verziou Voodoo 2. Bohužiaľ, sa tak nestalo, pretože jej vývoj sa predĺžil rovnako ako pri Rampage a na trh sa dostala až v dobe, keď Voodoo 2 rapídne klesla cena. Akcelerátor Voodoo 3 vyšiel v troch variantách a to 2000, 3000 a 3500SE. Hlavným rozdielom bol takt procesoru (140 - 200 MHz) a veľkosť pamätí (16 MB). Rad kariet Voodoo 3 mal problémy s kvalitou obrazu oproti konkurenčnej Nvidii, ktorá ponúkala 32-bitovú hĺbku, zatiaľ čo 3dfx len 24-bitovú. Neskôr vyšli nové rady Voodoo 4 a 5. Najvyšší a zároveň najvýkonnejší rad Voodoo 5-6000 mal štyri čipy a až 128 MB pamäte. Podporoval DirectX 6.0 *full-scene antialiasing*, efekty ako *motion blur* a *soft shadows*. Karta sa mala stať lídrom a upevniť postavenie 3dfx na trhu, no nestalo sa tak. Séria neúspešných projektov ako Rampage a Banshee dostala 3dfx do finančnej krízy. Ako už bolo spomínané, v roku 2000 odkúpila Nvidia zvyšok krachujúceho 3dfx.

7. SLI - Scan-Line Interleave

Kapitola 2

OpenGL

2.1 Vznik OpenGL

Spoločnosť Silicon Graphics (SGI) vznikla v roku 1981, špecializovala sa na 3D počítačovú grafiku a s ňou spojený vývoj softwaru a hardwaru. Jeden z projektov, ktorému sa SGI venovala, bola programová knižnica Iris GL[1]. Táto knižnica mala niekoľko závažných nedostatkov, medzi ktoré patrila slabá dokumentácia a nedostatok formálnej špecifikácie, preto sa Iris GL stala nepoužiteľnou a nevhodnou pre širšie použitie. SGI odstránila všetky nesúvisiace časti s počítačovou grafikou a v roku 1992 vydala pod názvom OpenGL[2] platformovo nezávislé API[3] pre prácu s počítačovou grafikou v reálnom čase. Hoci OpenGL vychádzalo z Iris GL, prvá verzia OpenGL 1.0 neposkytovala takú funkcionalitu ako Iris GL, chýbala jej napríklad podpora textúr (pridaná v OpenGL 1.1). OpenGL patrí medzi najznámejšie API pre tvorbu 2D a 3D prenositeľných grafických aplikácií. Okrem toho v roku 1992 vzniklo konzorcium nazvané ARB[4]. To pozostávalo z veľkých spoločností ako napríklad Apple, Nvidia a IBM, ktoré rozhodovali o budúcnosti OpenGL. Pod ich vedením boli schvaľované nové štandardy a novo pridaná funkcionalita. Vzhľadom na to, že OpenGL neposkytuje konkrétnu implementáciu, iba rozhranie knižnice, ostala samotná implementácia na hardwarových výrobcoch. To im poskytlo voľnosť pri implementácii a možnosť optimalizácie na ich špecifický hardware.

Veľkou výhodou pre OpenGL bolo, že podporovalo rozšírenia.

1. Iris GL - Silicon Graphics Iris Graphics Library
2. OpenGL - Open Graphics Library
3. API - Application Programming Interface
4. ARB - OpenGL Architecture Review Board

Ak OpenGL špecifikácia nezahŕňala určitú funkcionalitu, ktorú grafická karta umožňovala, mohla byť táto funkcionalita doplnená ako OpenGL rozšírenie. Jednotlivé rozšírenia mohli byť distribuované pomocou prefixov, ako napr. NV_ pre NVIDIA, AGL_ pre Apple a pod.

2.2 Popis OpenGL

Hardwarová a softwarová nezávislosť umožňuje použiť OpenGL na širokej palete zariadení. Pre mobilné zariadenia a tablety, ktoré nemajú takú pamäťovú a výkonnostnú kapacitu ako bežné počítače, existuje odľahčená verzia OpenGL ES. Keďže je OpenGL platformovo nezávislé, neobsahuje žiadnu funkcionalitu, ktorá by mohla zapríčiniť platformovú závislosť. Neobsahuje funkcie na prácu s oknami, vytváranie kontextu ani na spracovanie užívateľského vstupu. Naopak obsahuje funkcie na transformáciu, vykresľovanie, tieňovanie a rasterizáciu základných geometrických primitív, ako sú body, úsečky a trojuholníky.

2.3 OpenGL vs. DirectX

OpenGL bolo dostupné na všetkých Unixovo založených staniciach. S príchodom Windows NT na trh v roku 1993 bola predstavená nová technológia Win32 API pre tvorbu Windows aplikácií. Prvá verzia Windows NT vôbec nepodporovala OpenGL, toho sa dočkala až vo verzii NT 3.5 v 1994. Táto implementácia bola však extrémne pomalá a neefektívna. Microsoft vyhlásil, že používaním knižnice OpenGL môže dôjsť k extrémnemu spomaleniu operačného systému.

Microsoft videl príležitosť presadiť sa na trhu s počítačovými hrami, preto začal vyvíjať svoje vlastné 3D API, ktoré predstavil pod menom Direct3D. Direct3D je šírené v SDK[5] nazvanom DirectX, spolu s ďalšími API ako DirectDraw a DirectMusic. Tak začal súboj medzi OpenGL a DirectX.

Prvá verzia Direct3D bola príliš pomalá a neefektívna na to, aby sa ujala na trhu a konkurovala OpenGL. Preto bol Microsoft nútený

5. SDK - Software Development Kit

aj naďalej podporovať OpenGL, zatiaľ čo zdokonaľoval Direct3D. Až verzia DirectX 5.0 sa mohla vyrovnať vtedajšiemu OpenGL a obe API boli užívateľsky prístupné a použiteľné. Veľký zlom nastal, keď v roku 2000 Direct3D predstavil novú technológiu a spôsob programovania grafických aplikácií, kedy prešiel od fixného spracovania dát k programovateľnému s použitím shaderov.

Programovanie *shaderov* nebolo vôbec jednoduché. Jazyk, v ktorom sa písali bol podobný asembleru. To si Microsoft uvedomoval a preto neskôr v roku 2003 predstavil nový jazyk HLSL[6], ktorého syntax bola príbuzná jazyku C. DirectX podporoval shadere vo verzii 9.0, no OpenGL zaostalo a shadere nepodporovalo do verzie 2.0 vydanej v roku 2004. To značne zvýšilo popularitu DirectX na úkor OpenGL. Neoficiálne boli shadere prístupné pomocou OpenGL rozšírení, no oficiálna špecifikácia chýbala. Popularita DirectX naďalej stúpala, čomu svedčí aj fakt, že len pár hier bolo vydaných s podporou OpenGL. Mnohé firmy opúšťali OpenGL a prechádzali na DirectX. V rokoch 2004 až 2006 sa OpenGL snažilo dohnať náskok, ktorý si DirectX vybudoval.

2.4 Fixná a programovateľná pipeline

Už od začiatku obsahovalo OpenGL nástroje na prácu s bodmi, ich transformáciu, zmenu farby a operácie nad fragmentami. Poradie a spracovanie dát bolo pevne dané a jediný spôsob ako ho ovplyvniť bolo pomocou funkcií, ktoré menili stav OpenGL. Tento prístup sa v OpenGL označuje ako fixná *pipeline*. Časom pribúdali pokročilejšie funkcie, no všetky narážali na ten istý problém a tým bol výkon a optimalizácia. Vo verzii OpenGL 2.0 prišla programovateľná *pipeline*, ktorá umožňovala riadiť spracovanie dát pomocou shaderov. Nástup shaderov znamenal veľký posun v počítačovej grafike. Viac podrobností o tom ako sa shadere delia a aké operácie sú schopné vykonávať je popísaných v sekcii 3.6.

6. HLSL - High-Level Shader Language

2.5 Moderné OpenGL

Na svetovo známej konferencii SIGGRAPH bolo v roku 2006 oznámené, že OpenGL bude riadené novým konzorciom KHRONOS. Začalo sa hovoriť o úplne novom OpenGL, ktoré by malo priniesť novú funkcionalitu a malo by konkurovať DirectX. V roku 2007 boli predstavené dva nové varianty OpenGL, a to konkrétne Long Peaks a Mt. Evans. Tieto varianty sa líšili v tom, že Long Peaks mal zachovať kompatibilitu so starým OpenGL a Mt.Evans nie. Koncom roku 2007 sa nič výraznejšie nestalo a ani v OpenGL 3.0, vydanom roku 2008, sa toho veľa nezmenilo. Zavedený bol však *deprecation model*, ktorý označoval funkcionalitu, ktorá mohla byť v budúcnosti odstránená z OpenGL. V marci 2009 bola vydaná verzia OpenGL 3.1, ktorá nadobro odstránila starý spôsob programovania v OpenGL. O pár mesiacov neskôr bola vydaná verzia 3.2, ktorá až na pár detailov konkurovala DirectX 10. Tým pádom sa OpenGL opäť stalo rovnocenným súperom DirectX. Nevýhodou bolo, že počas stagnovania OpenGL v rokoch 2004 až 2006, získalo DirectX vo svete veľkú obľubu.

K dnešnému dňu je najnovšie vydaná verzia OpenGL 4.4 spolu s OpenGL shading jazykom 4.40. Toto API obsahuje stovky funkcií a procedúr umožňujúcich špecifikovať shadere, buffer objekty a operácie nad nimi.

2.6 OpenGL v Jave

V roku 2003 začala SGI[7] spolupracovať so Sun na prepojení Javy s OpenGL. V dnešnej dobe existuje niekoľko projektov, ktoré sa venujú práve prepojeniu Javy s OpenGL.

- LWJGL *(The Lightweight Java Game Library)*
- JOGL *(Java OpenGL)*
- Java3D
- YAJOGLB *(Yet Another Java OpenGL Binding)*

7. SGI - Silicon Graphics International Corp.

Medzi najznámejšie patrí JOGL[1] a LWJGL[2].

LWJGL je minimalistické a multiplatformové API, ktoré neposkytuje prístup iba k OpenGL, ale aj k OpenCL (*Open computing language*) a OpenAL (*Open audio library*). Taktiež zjednodušuje prácu s hardwarovými zariadeniami, ako napríklad s joystickom alebo gamepadom. Ponúka efektívnejšie zdieľanie pamäte medzi OpenGL a OpenCL. Neobsahuje optimalizované funkcie ako `glColor3fv`, ktoré sú určené skôr pre jazyk C ako pre jazyk Java a nepodporuje ani *single buffering*. Vstupné parametre pre OpenGL funkcie sú predávané ako buffery, aby sa predišlo zbytočným chybám. Namiesto toho knižnica ponúka zjednodušenú prácu s vytváraním kontextu OpenGL, maticami alebo synchronizácie FPS.

Kapitola 3

Demo aplikácia

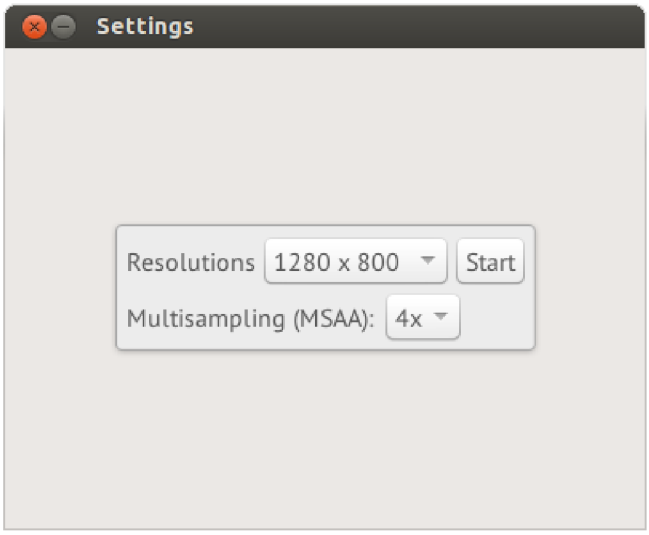
Praktickou časťou bakalárskej práce je demo aplikácia, ktorá demonštruje základné princípy a funkcionalitu moderného OpenGL. Aplikácia je naprogramovaná v jazyku Java a OpenGL s využitím knižnice LWJGL. Užívateľ si môže vybrať jeden z 3D modelov a rôzne vykresľovacie módy. Modely je možné otáčať a pohybovať s nimi pomocou klávesnice a myši. Grafické rozhranie aplikácie umožňuje výber 3D modelu, spôsob vykreslenia, zmenu polohy svetla a ďalšie. Aplikácia je multiplatformová, čo znamená, že je kompatibilná s operačnými systémami Windows, Linux a OS X. Správne spustenie aplikácie vyžaduje, aby grafická karta podporovala minimálne OpenGL 4.3.

3.1 Popis dema

Po spustení aplikácie sa zobrazí okno s nastaveniami (viď obr. 3.1), ktoré ponúka možnosť výberu rozlíšenia a *antialiasingu* (MSAA) použitými v hlavnom okne aplikácie. Výber rozlíšenia a MSAA sa po spustení hlavného okna nedá zmeniť. Pri výbere stupňa *antialiasingu* treba byť opatrný a treba brať do úvahy výkon grafickej karty v počítači, na ktorom sa aplikácia spúšťa. *Antialiasing* je proces, pri ktorom sa vyhladzujú hrany objektov a tým sa odstráni „zubatosť“. Je to náročný proces, ktorý môže výrazne spomaliť chod aplikácie. Obrázok 3.2 znázorňuje jednotlivé stupne MSAA.

Súčasne so štartom aplikácie sa spustí aj načítavanie všetkých 3D modelov z disku. Funkcionalita jednotlivých ovládacích prvkov aplikácie je detailnejšie popísaná na strane 16. Na pravej strane okna sa nachádza panel s nastaveniami (viď obr. 3.2.2).

Ako už bolo spomenuté, aplikácia vykresľuje 3D modely. Na vý-



Obr. 3.1: Úvodné nastavenia v aplikácii.

ber je jeden zo šiestich modelov. Každému modelu je možné nastaviť farbu (resp. materiál), mód vykreslenia (*wire*, *shade*, *wire shade*) a textúru. Aplikácia umožňuje meniť tieto vlastnosti svetla: polohu, farbu, intenzitu a uhol. Ako posledné je možné vybrať jeden z filtrov, ktorý sa aplikuje na vykreslený 2D obraz. Na výber je jeden zo siedmich filtrov, každý z nich je bližšie popísaný v sekcii 3.5.

3.2 Tvorba GUI

Java ponúka dve základné API na tvorbu grafického užívateľského rozhrania (GUI[1]) a to Swing [16] a AWT [17]. Obe API sú súčasťou Java Foundation Classes (JFC). Jedná sa o knižnice užívateľských prvkov, pomocou ktorých sa vytvára GUI, ako napríklad tlačítka, okná, dialógy, rozbaľovacie zoznamy atď. Práca s nimi je veľmi jednoduchá v porovnaní s knižnicou TWL[13] ktorú sme použili.

Knižnicu TWL sme sa rozhodli použiť z dôvodu, že vo Swing nie je možné vytvoriť okno aplikácie, do ktorého by sa dalo cez LWJGL vykresľovať. AWT je síce možné použiť s LWJGL cez komponent AWT Frame a Canvas ale pri naštudovaní ako komponent funguje

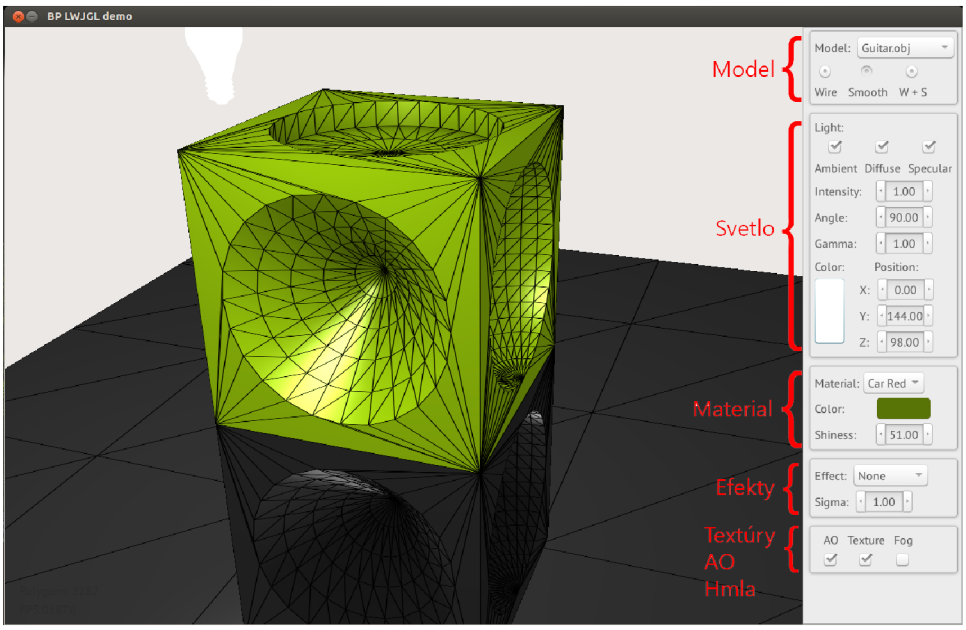
1. GUI - Graphical User Interface

Obr. 3.2: Rôzne úrovne antialiasingu.

a aká je jeho spoľahlivosť sme sa ju rozhodli nepoužiť. Navyše, TWL je spoľahlivá, robustná a rýchla knižnica, ktorá používa na vykresľovanie ovládacích prvkov priamo OpenGL 1.1.

3.2.1 Knižnica TWL

TWL je malá a veľmi nízkoúrovňová knižnica distribuovaná pod BSD [15] licenciou. Programátor si musí vytvoriť úplne všetko sám, od vzhľadu tlačítok, kurzoru, okien, scroll baru až po logiku a správanie komponentov. Chýba kvalitná dokumentácia, k dispozícii je len obmedzená sada zdrojových súborov a jednoduché príklady na oficiálnej stránke knižnice. Práca s touto knižnicou nebola vôbec jednoduchá. Napriek tomu TWL ponúka neobmedzené možnosti a nekladie žiadne prekážky kreativite. Každý vývojár grafických aplikácií môže naprogramovať správanie a fungovanie komponentov tak ako mu to vyhovuje a bez výraznejších obmedzení. Nevýhodou knižnice je, že aj jednoduché veci majú komplikovanú implementáciu a vyžadujú nemalé úsilie. Pri tvorbe komponentov a ich editácii sme použili jednoduchú aplikáciu, ktorá je taktiež k dispozícii k TWL a to TWL Theme Editor. Tento editor umožňuje jednoduchšie vytváranie nových komponentov a priame generovanie XML kódu, v ktorom sú



Obr. 3.3: Hlavné okno aplikácie.

uložené všetky potrebné informácie o ich vzhľade a logike. Princíp je veľmi podobný CSS štýlom, kde sú grafická a logická časť kódu oddelené. Preto k celkovej zmene vzhľadu GUI stačí, ak sa pozmení XML kód generovaný TWL Theme Editorom a nie sú potrebné žiadne zmeny v zdrojovom kóde aplikácie. Komponenty sa vytvárajú podobne ako pri použití Swing alebo AWT. Ako príklad použitia TWL uvádzame jednoduchý kód na vytvorenie tlačítka.

```
Button button = new Button("Stlač ma");
button.setTheme("button");
button.addCallback(buttonStlacmaCallback);
button.setPosition(50,50);

DialogLayout.Group vGroup =
container.createSequentialGroup(button);

DialogLayout.Group hGroup =
container.createParallelGroup(button);
```

```
container.setHorizontalGroup(vGroup);
container.setVerticalGroup(hGroup);
add(container);
```

Jednou z najväčších komplikácii pri tvorbe GUI pomocou TWL je vkladanie komponentov do skupín. Každý jeden komponent musí byť aspoň v jednej skupine. Existujú dva typy skupín, a to paralelná a sekvenčná. Ovládacie prvky sa vkladajú do skupín podľa toho, či majú byť zarovnané vertikálne alebo horizontálne. Tvorba komplikovanejších formulárov a okien plných ovládacích prvkov je dosť neprehľadná a zložitá, pretože jeden ovládací prvok sa môže nachádzať aj v desiatich skupinách a to len kvôli tomu, aby sa správne posúvali a zarovnávali všetky prvky, napríklad pri zmene veľkosti okna.

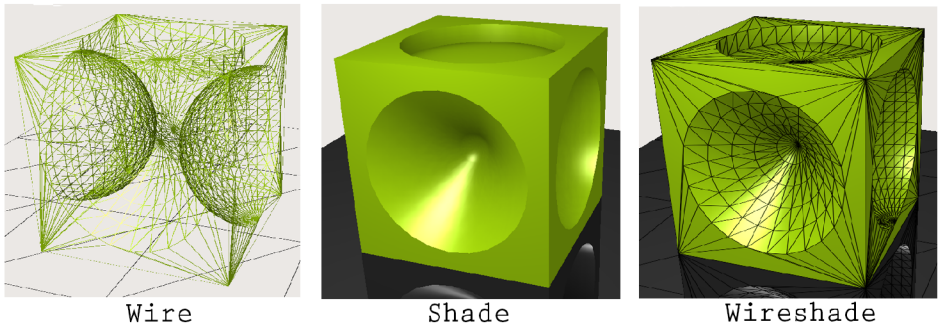
XML súbor vygenerovaný TWL Theme Editorom obsahuje vizuálnu stránku všetkých komponentov a ich logiku. Pod logikou je myslené, ako sa zmení vzhľad komponentu, keď sa napríklad klikne na tlačítko *button*. Tento prístup ponúka veľmi silný nástroj pri tvorbe GUI.

3.2.2 Popis GUI

Užívateľské rozhranie je opticky rozdelené na päť častí (obr. 3.3).

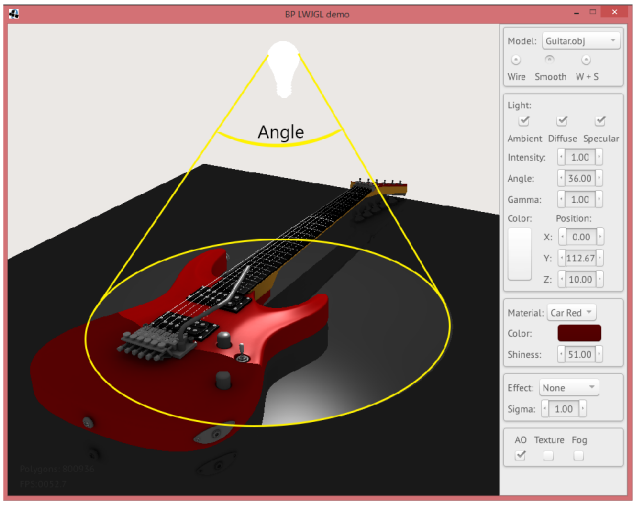
- Model
- Light
- Material
- Effect
- Other settings

Prvá časť ponúka na výber model, ktorý sa má vykresliť a vykresľovací mód. Módy sú tri a sú zobrazené na obr. 3.4. *Wire* mód alebo drôtený mód vykresľuje hrany trojuholníkov tvoriacich model. *Shade* mód vykresľuje trojuholníky na základe Phongovho osvetľovacieho modelu popísaného na strane 19. *Wireshade* je spojením predošlých dvoch módov.



Obr. 3.4: Módy vykreslenia voliteľné v aplikácii.

Druhá časť obsahuje ovládacie prvky na manipuláciu so svetlom. Svetlo sa skladá z troch zložiek, okolitej, difúznej a odrazovej, všetky tri zložky sa dajú zapínať a vypínať. Svetlo je kužeľovo smerové a je určené smerovým vektorom a uhlom. Smer vektoru je určený bodom [x,y,z] a bodom [0,0,0]. Takže svieti vždy do stredu súradnicového systému a teda presne na model. Uhol špecifikuje aký veľký rozptyl svetlo má. Na obrázku 3.5 je znázornené čo je pod uhlom svetla myslené a ako vyzerá svetelný kužeľ svetla. Ak je uhol rovný 90 stupňov, svetlo sa správa ako obyčajné smerové svetlo.



Obr. 3.5: Uhol svetla (svetelný kužeľ).

V tretej časti je možné meniť nastavenia materiálu. V položke *Materal* sú predpripravené materiály, ktoré zodpovedajú skutočným materiálom.

V štvrtej časti je možné aplikovať jeden z filtrov na vykreslený 2D obraz. Položka *Shiness* je aktívna iba pri Gaussovom filtri, kde určuje mieru rozmazania obrazu.

Piata a zároveň posledná sekcia ovládacieho panelu obsahuje tri zaškrtávacie políčka.

- *AO* (Ambient Occlusion)
- *Texture*
- *Fog*

Ambient Occlusion je efekt, ktorý pridáva hĺbku obrazu a model tak vyzerá viac realisticky. Efekt hmly je jednoduchý efekt, ktorý je prístupný už od prvých verzií OpenGL. V ľavom dolnom rohu aplikácie sa nachádza FPS počítadlo. FPS[2] určuje počet prekreslení okna za sekundu. Hodnoty FPS sa dajú považovať za jednoduchý test výkonu. Nad ním sa nachádza počet trojuholníkov tvoriacich aktuálne vykresľovaný model.

3.3 3D modely

Všetky modely sú uložené v OBJ formáte z dôvodu, že tento formát ukladá dáta pomocou ASCII znakov a je preto čitateľný aj pre človeka. Je jednoduché ho spracovať a pre potreby tejto bakalárskej práce je postačujúci. Každý model sa načítava vo vlastnom vlákne a keď sa model celý načíta, pridá sa do zoznamu v aplikácii. Medzitým aplikácia zobrazuje len tie modely, ktoré sú už kompletne načítané. Všetky 3D modely sme vymodelovali a namapovali v 3DS Max 2012. Pri programovaní sme sa snažili použiť najnovšie technológie, ktoré OpenGL poskytuje a preto sme sa rozhodli použiť VBO[3]. VBO je buffer, v ktorom sú uložené súradnice bodov modelu, normálové vektory k týmto bodom a poloha v textúre (UV). VBO v OpenGL

2. FPS - Frame per second
3. VBO - Vertex buffer object

používa iba jeden buffer indexov a OBJ formát až tri (zvlášť pre body, normály a súradnice v textúre). Museli sme vymyslieť spôsob ako upraviť pôvodný model tak, aby sa všetky tri polia indexov v OBJ formáte zhodovali a bolo možné použiť ľubovoľný z nich pre VBO. Algoritmus[19] postupne prechádza všetky body tvoriace 3D model a porovnáva ich s tými, ktoré už spracoval. Pritom duplikuje vrcholy a upravuje indexovanie. Výsledkom je opäť model vo formáte OBJ, ktorý spĺňa špecifikáciu, ale používa len jedno pole indexov pre vrcholy, normály a textúry. Použitý algoritmus má časovú zložitosť $O(n^2)$, preto boli všetky modely prekonvertované vopred.

3.4 Osvetľovací model

OpenGL do verzie 2.0 malo naimplementovaný svoj nie veľmi efektívny a vizuálne pekný svetelný model, funkcie na prácu so svetlom a materiálom. V modernom OpenGL sú všetky tieto funkcie nahradené programovateľnou *pipeline* a *shadermi*, čím je umožnené vytvoriť krajšie, efektívnejšie a zaujímavejšie svetelné modely, ktoré v dnešnej dobe existujú. Cenou je však náročnejšie programovanie. Existuje niekoľko svetelných modelov popisujúcich správanie svetla v reálnom svete. Medzi najznámejšie svetelné modely patrí

- Phongov svetelný model[5]
- Blinn – Phongov svetelný model[6]
- Lambertov svetelný model[7]

V aplikácii sme naimplementovali Phongov svetelný model a Phongovu interpolačnú techniku. Phongov svetelný model počíta výslednú farbu I pre každý pixel zvlášť podľa vzorca $I = I_a + I_d + I_s$, kde jednotlivé zložky vypočítame podľa nasledujúcich vzorcov.

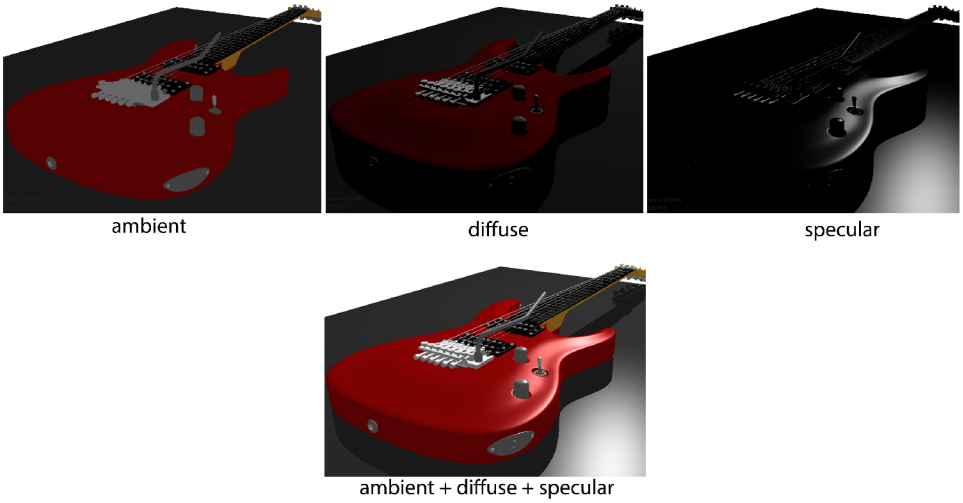
$$I_a = L_a.K_a$$

$$I_d = L_d.K_d.(s.n)$$

$$I_s = L_s.K_s(r.v)^f$$

Farba pixelu sa teda skladá z troch zložiek (viď obr. 3.6)

- I_a okolitá zložka[4]
- I_d difúzna zložka[5]
- I_s odrazová zložka[6]



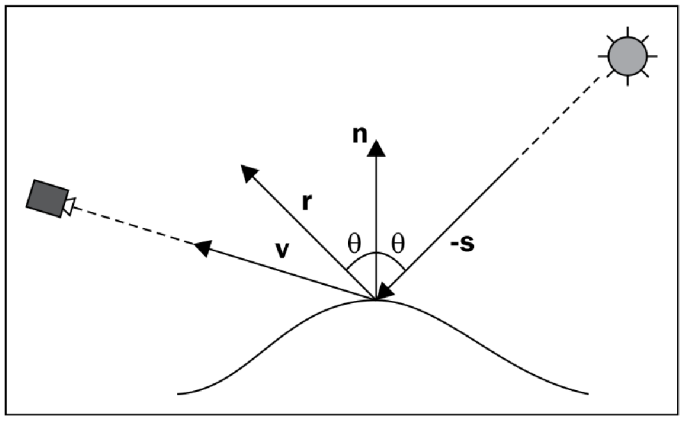
Obr. 3.6: Jednotlivé zložky farby

L_a, L_d, L_s je okolitá, difúzna a odrazová zložka svetla. K_a, K_d, K_s sú zložky materiálu. Vektory s, n, r, v sú znázornené na obr. 3.7 a majú nasledujúci význam:

- s - smerový vektor svetla
- n - normálový vektor aktuálne spracovávaného bodu
- r - odzrkadlený podľa vektoru n
- v - smerový vektor kamery (pozorovateľa)

Phongova interpolačná tieňovacia technika (tiež nazývaná *Per-Pixel Shading*) počíta farbu zvlášť pre každý pixel obrazovky, narozdiel od Guaraudovej tieňovacej techniky, ktorá počíta farbu pre každý vrchol modelu a farbu lineárne interpoluje.

4. ambient light
5. diffuse light
6. specular light



Obr. 3.7: Phongov svetelný model.

3.4.1 Textúry

Textúry sú veľmi silný nástroj, ktorým sa dajú dosiahnuť extrémne vizuálne efekty, ako napríklad pridanie detailov modelu, zvýšenie kvality, reálnosti (MSAA, AO) a dokonca zmeny polohy vrcholov modelu (Displacement mapping). Existuje množstvo efektov, ktoré možno dosiahnuť vďaka textúram. Medzi ne patria aj tieto:

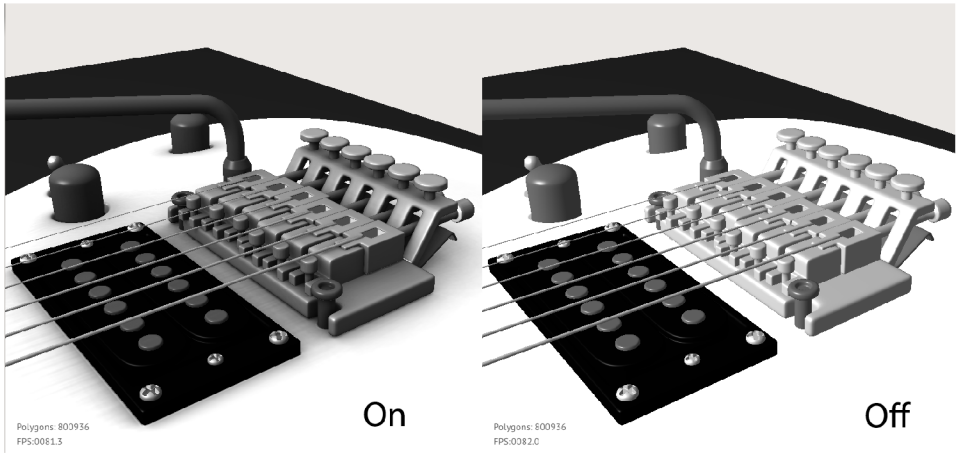
- Normal Mapping
- Ambient Occlusion
- Displacement Mapping
- Klasická aplikácia textúry
- Specular Mapping
- Cube Mapping

Ambient Occlusion

Ambient occlusion, ďalej len AO, je jeden zo základných efektov používaných pri realistickom vykresľovaní scén. Ide o zvýraznenie jemných tieňov, ktoré vznikajú v úzkych priestoroch, štrbinách, dierach alebo miestach, kam dopadá menej svetla a tým sa dosiahne väčšia hĺbka obrazu. Existuje niekoľko prístupov ako spočítať AO.

- Reálne v čase[7] - *SSAO, HBAO, HBAO+*
- Predpočítané[8]

Predpočítané AO je najjednoduchší spôsob na implementáciu. AO textúra sa predpočíta vopred a počas vykresľovania sa len aplikuje na model. Na generovanie AO máp sme použili program xNormal. Je to jednoduchý a veľmi užitočný program, ktorý dokáže generovať veľké množstvo rôznych typov textúr. Zastávame názor, že výstup z xNormal je v niektorých prípadoch kvalitnejší ako výstup z profesionálnych modelovacích programov (napríklad 3D Max). Na obrázku 3.8 je zobrazený rozdiel medzi zapnutím a vypnutím AO.



Obr. 3.8: Zap./Vyp. Ambient Occlusion

3.5 2D efekty

Úprava obrazu je veľmi častou procedúrou v počítačovej grafike. Konvolúcia patrí medzi jednu z mnohých úprav, ktoré môžeme s obrazom robiť. Zamerali sme sa na diskrétnu 2D konvolúciu. Diskrétnu preto, lebo pracujeme s obrazom, ktorý je tvorený obrazovými bodmi (pixelmi) a ich počet je konečný a 2D preto, lebo pracujeme len s 2D obrazom ktorý dostaneme na výstupe z grafickej karty.

7. Real Time AO
8. Pre-baked AO

Konvolúcia je matematická operácia, ktorá pri svojom výpočte používa jadro (*kernel*) a práve vhodnou voľbou jadra je možné dosiahnuť zaujímavé úpravy, efekty na obraze. Všetky nižšie popísané efekty sú dosiahnuté práve pomocou konvolúcie a jediná vec, v ktorej sa líšia je práve jadro, s ktorým konvolúcia pracuje. Ďalšou veľkou výhodou konvolúcie je, že je pomerne ľahko paralelizovateľná a preto je ideálna pre GPU. Keďže GPU obsahuje množstvo výpočtových jednotiek (jadier), môže každé z týchto jadier spracovávať iný bod obrazu, vďaka čomu je celý výpočet extrémne rýchly. Celý proces aplikácie 2D efektu je rozdelený na dve fázy. Pri aplikácii 2D efektu na obraz je potrebné mať už obraz uložený a následne naň aplikovať efekt. Pri klasickom postupe vykresľovania scény sa výsledný obraz z grafickej karty rovno zobrazuje na monitore. Preto sme si vytvorili vlastnú obrazovú pamäť (frame buffer), do ktorej sme vykreslili celú scénu bez aplikácie 2D efektu. Tento obraz z frame bufferu sa nikde nevykresľuje. Potom sme namiesto vykresľovania celej scény opäť na obrazovku vykreslil iba *jeden* obdĺžnik cez celé okno obrazovky a naň sme ako textúru použili obraz, ktorý bol vykreslený do frame bufferu na začiatku. Je to dvojfázový proces, ktorý je potrebný na to, aby sme mohli aplikovať 2D efekt na obraz. Takže v konečnom dôsledku to, čo sa zobrazuje, je obyčajný obdĺžnik, na ktorom je aplikovaná textúra, ktorá sa ale dynamicky mení.

3.5.1 Detekcia hrán

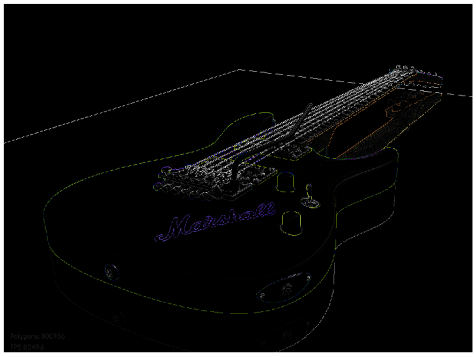
Najčastejšie používané filtre na detekciu hrán sú Sobel a Laplace. Laplaceho filter detekuje všetky hrany vo všetkých smeroch, ale pri Sobelovom filtri si môžeme vybrať či chceme detekovať horizontálne alebo vertikálne hrany. Na obrázkoch 3.9 a 3.10 sú znázornené filtre.

-1	-1	-1
-1	8	-1
-1	-1	-1

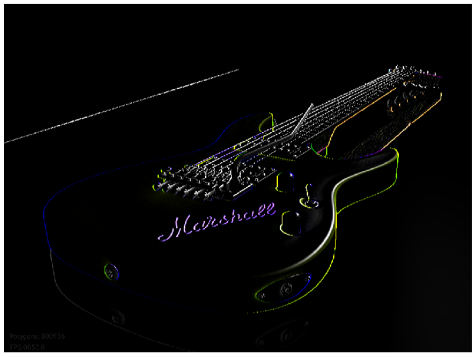
Tabuľka 3.1: Laplace jadro

1	0	- 1
2	0	-2
1	0	-1

Tabuľka 3.2: Sobel horiz. jadro



Obr. 3.9: Laplace filter



Obr. 3.10: Horizontal Sobel filter

3.5.2 Rozmazanie obrazu

Rozmazanie obrazu je možné docieliť rôznymi spôsobmi, Jedným z nich je použitie Gaussovho filtru[8]. Gaussov filter má kvalitnejší výstup ako obyčajné spriemerovanie pixelov a to vďaka funkcii, ktorou je definovaný resp. tvarom gaussovho jadra. Gaussova funkcia má tvar

$$G(x,y) = \frac{1}{2\pi\sigma^2}\varepsilon^{-\frac{x^2+y^2}{2\sigma^2}}$$

Sila rozmazania sa dá ovplyvňovať buď veľkosťou jadra alebo veľkosťou sigmy. Pre jednoduchosť bolo v demo aplikácii použité jadro 5x5 s predpočítanými hodnotami podľa gaussovej rovnice. K samotnej aplikácii Gaussovho filtru na obraz sa taktiež používa konvolúcia, ale s jadrom s veľkosťou 5x5, aby bol efekt rozmazania silnejší.

3.5.3 Odraz v podlahe

Phongov svetelný model neberie do úvahy odrazené svetelné paprsky. Výpočet farby pixelu je počítaný len z priameho neodrazeného paprsku svetla. Preto tvorba efektu, pri ktorom vidno odraz modelu v podlahe vyžaduje náročnejšiu implementáciu. Proces sa dá rozdeliť do troch krokov

1. vykreslenie podlahy, zápis do Stencil bufferu
2. vykreslenie prevráteného (odzrkadleného) modelu orezaného podľa Stencil bufferu

3. vykreslenie zvyšku scény

Stencil buffer je jeden z mnohých bufferov ktoré OpenGL ponúka. Stencil buffer má rovnakú veľkosť ako Color buffer, do ktorého sa vykresľuje scéna s tým rozdielom, že sa neukladá farba pixelu, ale 0 alebo 1. Neskôr sa môže tento stencil buffer použiť na odstránenie istých častí obrazu podľa toho, či sa na danom mieste nachádza 0 alebo 1. Jednotka znamená, že sa daný pixel ponechá a nula naopak, že sa odstráni. Celý postup môžeme zhrnúť do troch bodov.

1. Na začiatku sa vykreslí iba podlaha. Následne sa do Stencil bufferu zapíše 1 na miesto, kde je podlaha vykreslená a všade tam, kde podlaha nie je sa zapíše 0.

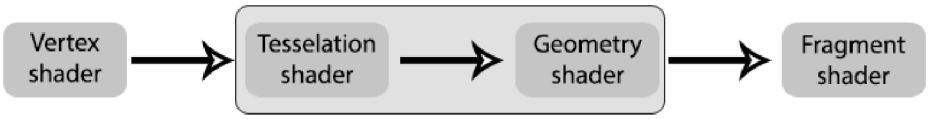
2. V druhej fáze sa vykreslí daný model, ale so zápornou Y-novou súradnicou (dolu hlavou). To bude náš odraz v podlahe. Teraz sa využije Stencil buffer z predošlého kroku a obraz, ktorý sa teraz vykreslil sa ešte oreže podľa Stencil bufferu, teda z nášho odzrkadleného obrazu ponecháme len tú časť, ktorá sa prekrýva s podlahou. Je to z toho dôvodu, že odraz v podlahe nie je možné vidieť tam, kde podlaha nie je.

3. V poslednej fáze sa vykreslí celá scéna normálne, ale už bez podlahy.

3.6 Shader

Shader bol prvý krát predstavený v OpenGL 1.4, ale len ako OpenGL rozšírenie. Oficiálne bol pridaný až vo verzii 2.0. V tejto verzii však existovali len *vertex* a *fragment shadere*. V ďalších verziách pribudol *geometry shader* (OpenGL 3.2), *tessellation shader* (OpenGL 4.0) a *compute shader* vo verzii 4.3. Vertex a fragment shader sú povinnou súčasťou programovateľnej pipeline na rozdiel od geometry, tessellation a compute shaderov.

Shader je krátky program napísaný v špecializovanom jazyku. Pre OpenGL bol vyvinutý jazyk GLSL. Je veľmi podobný jazyku C, no neobsahuje pointre a funkcie na prácu so vstupom a výstupom. Poradie, v ktorom sú jednotlivé shadere vyhodnocované, je na obrázku 3.11.



Obr. 3.11: Shader pipeline.

3.6.1 Vertex shader

Vertex shader má za úlohu transformovať súradnice bodov, najčastejšie podľa *Projection matrix*, *Model Matrix* a *View Matrix*. Vertex shader pracuje s jednotlivými bodmi samostatne. To znamená, že pre každý bod modelu sa spustí jeden vertex shader, ktorý transformuje jeho polohu. V spomenutých maticiach sú uložené informácie o perspektívnej (ortogonálnej) projekcii, poprípade posune, rotácii a zmene veľkosti modelu a kamery. Inými slovami, v tomto mieste sú body transformované z lokálnych (*Object space*) do globálnych súradníc (*World space*) a posledná transformácia je perspektívna projekcia. Okrem toho sa tu ešte počíta normálová matica, ktorá je potrebná pri spätnej transformácii normálových vektorov. Je to z toho dôvodu, že pri transformácii bodov sa transformujú aj ich normálové vektory, ktoré by však mali ostať nezmenené, aby bol výpočet osvetlenia správny. Vstupom pre vertex shader je jeden bod, ktorý sa transformuje a výstupom je opäť jeden bod. Nedochádza tu teda k vytváraniu nových bodov.

3.6.2 Tessellation shader

Pomocou tessellation shadera je možné meniť geometriu objektu, podobne ako aj pomocou geometry shadera. Jeho primárnou úlohou je pridávať alebo odoberať veľké množstvo bodov v nami určenej závislosti, napr. na vzdialenosti od kamery. Objekty veľmi vzdialené od kamery nepotrebujú množstvo detailov a nemusia sa skladať z tisícok trojuholníkov, pretože sú ďaleko a nie je ich dobre vidieť. V takom prípade je rozumné použiť tessellation shader, ktorý zredukuje počet trojuholníkov veľmi vzdialeným objektom a tým optimalizuje celé vykresľovanie scény.

3.6.3 Geometry shader

Geometry shader je jeden z nepovinných shaderov v programovateľnej pipeline. Je to posledné miesto, kde je možné meniť polygónovú sieť modelu. Rozdiel oproti vertex shaderu je v tom, že geometry shader pracuje so základnými primitívami, ako sú body, čiary, trojuholníky a dokáže vytvárať *nové* body, úsečky alebo trojuholníky. Dôvod, prečo sme zaviedli tento shader do pipeline bol kvôli módom zobrazenia (viď obr. 3.4). Počíta sa tu polygónová sieť, presnejšie čiary spájajúce jednotlivé vrcholy.

3.6.4 Fragment shader

Fragment shader je posledným miestom, kde môže programátor zasiahnuť do výpočtu obrazu. V tomto shaderi je implementovaný celý výpočet Phongovho svetelného modelu spolu s aplikáciou textúr, AO, hmly, efektov a vykresľovacieho módu. Pre každý pixel obrazu je spustený samostatný fragment shader, ktorý vypočíta jeho výslednú farbu. Tá sa zapíše do frame bufferu a následne sa obraz vykreslí na obrazovku.

Kapitola 4

Záver

V predchádzajúcich kapitolách sme zhrnuli doterajší vývoj grafických akcelerátorov a programovacej knižnice OpenGL. V praktickej časti práce sme demonštrovali možnosti a základnú funkcionalitu nových verzií OpenGL v spolupráci s jazykom Java. Po naštudovaní rôznych implementácií knižnice OpenGL pre Javu sme sa rozhodli použiť knižnicu LWJGL.

Vytvorená aplikácia je plne funkčná v rôznych operačných systémoch a demonštruje základné renderovacie techniky a postupy. Kód aplikácie je navrhnutý tak, aby sa dal jednoducho rozšíriť o ďalšiu funkcionalitu. V práci na aplikácii plánujeme pokračovať a v blízkej budúcnosti doimplementujeme pokročilejšie techniky, ako aj podporu animácií. Je možné, že výber inej knižnice, napríklad JOGL, by zjednodušil niektoré implementačné kroky, napríklad tvorbu GUI, keďže JOGL integruje OpenGL kontext priamo do okenného systému *Swing* alebo *AWT* z Javy. Knižnica TWL je vhodnejšia v prípadoch, keď potrebujeme vytvoriť netradičný vzhľad ovládacích prvkov, napríklad v hrách.

Vývoj OpenGL v poslednej dobe rýchlo napreduje, čo dokazuje aj fakt, že posledné štyri verzie vychádzajú v pravidelných ročných intervaloch. Postavenie OpenGL sa upevňuje aj v komerčnej sfére, špeciálne v hernom priemysle, kde v súčasnosti dominuje proprietárna knižnica DirectX. Popularita OpenGL stúpa aj vďaka projektu Steam od spoločnosti Valve bežiacim už aj pod operačným systémom GNU/Linux, ako aj vďaka známemu hernému enginu Unity 3D. Jednou z výhod OpenGL oproti DirectX je možnosť tvorby multiplatformových aplikácií, čo môže výrazne pomôcť obľúbenosti OpenGL hlavne v prípade rastúceho záujmu širokej verejnosti o operačné systémy založené na Unixe.

Dodatok A

Obsah priloženého CD

- **BP.pdf** - text práce vo formáte .pdf
- **BP.tex** - text práce co formáte .tex pre LaTeX
- **BP_demo.jar** - spustiteľná demo aplikácia
- **res** - priečinok obsahujúci 3D modely, textúry a shadere

Spustenie aplikácie:

Spustenie aplikácie si vyžaduje mať nainštalované najnovšie ovládače grafickej karty, ktoré podporujú OpenGL 4.3. V prípade, že táto podmienka nebude splnená, vypíše sa chybová hláška na konzolu a program sa predčasne ukončí. Ak program vypíše chybu

```
OutOfMemoryError: Direct buffer memory,
```

je potrebné nastaviť väčšiu kapacitu direct bufferu pre JVM pomocou prepínača `-XX:MaxDirectMemorySize=128M`, takže aplikácia sa spustí z konzoly príkazom

```
java -jar -XX:MaxDirectMemorySize=128M BP_demo.jar.
```

Aplikácia sa ovláda pomocou kláves *w, a, s, d, shift, space* a pomocou myši. Klávesy slúžia k posunu celej scény a myš na jej otáčanie.

- *a, d, space, shift* - posun scény vľavo, vpravo, dolu, hore
- *w, s* - priblíženie, oddialenie scény

Literatúra

[1] *Java OpenGL* [online]. 2013 [cit. 2013-05-21]. JOGL. Dostupné z: <http://jogamp.org/jogl/www/>.

[2] *Lightweight Java Game Library* [online]. 2013 [cit. 2013-05-21]. LWJGL. Dostupné z: <http://http://lwjgl.org/>.

[3] D. Wolff: *OpenGL 4.0 Shading Language Cookbook*. Packt Publishing Ltd., Birmingham UK, Júl 2011. ISBN 978-1-849514-76-7.

[4] R. J. Rost: *OpenGL Shading Language, Second Edition*. Addison Wesley Professional, Január 2006. ISBN 0-321-33489-2.

[5] Bui Tuong Phong: *Illumination of Computer-Generated Images*. Department of Computer Science, University of Utah, Júl 1973. UTEC-CSs-73-129.

[6] James F. Blinn: *Models of light reflection for computer synthesized pictures*. ACM SIGGRAPH Computer Graphics '77, New York, 1977. 10.1145/563858.563893.

[7] Angel, Edward: *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*. Addison-Wesley, ISBN 978-0-321-31252-5.

[8] Shapiro, L. G., Stockman: *Computer Vision*. Prentice Hall, 2011.

[9] D. Shreiner: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition)*. Addison Wesley Professional, Boston, Júl 2009. ISBN 0-321-55262-8.

[10] E. Luten. *OpenGLBook* [online]. 2011 [cit. 2013-05-22]. Dostupné z: <http://openglbook.com/>.

[11] *Video card* [online]. 2013 [cit. 2013-05-22]. Wikipedia, the free encyclopedia. Dostupné z: <http://en.wikipedia.org/wiki/Video_card/>.

[12] P. Cozzi, Ch. Riccio: *OpenGL Insights*. CRC Press, New York USA, 2012. ISBN 13:978-1-4398-9377-7.

[13] *Themable Widget Library* [online]. 2013 [cit. 2013-05-21]. TWL. Dostupné z: <http://l33tlabs.org//>.

[14] *20 let vývoje grafických karet* [online]. 2012 [cit. 2013-05-21]. ExtraHardware. Dostupné z: <http://http://extrahardware.cnews.cz/clanky/20-let-vyvoje-grafickych-karet-od-cga-az-po-konec-3dfx-dil-i>.

[15] *BSD licenses* [online]. 2013 [cit. 2013-05-21]. Wikipedia, the free encyclopedia. Dostupné z: <http://http://en.wikipedia.org/wiki/BSD_licenses/>.

[16] *SWING API* [online]. 2013 [cit. 2013-05-21]. ORACLE Java SE Documentation. Dostupné z: <http://docs.oracle.com/javase/7/docs/technotes/guides/swing/>.

[17] *AWT API* [online]. 2013 [cit. 2013-05-21]. ORACLE Java SE Documentation. Dostupné z: <http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html/>.

[18] *Open.gl* [online]. 2013 [cit. 2013-05-21]. OpenGL tutorial. Dostupné z: <http://open.gl/>.

[19] *VBO Indexing* [online]. 2013 [cit. 2013-05-21]. opengl-tutorial. Dostupné z: <http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-9-vbo-indexing/>.