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

POUŽITÍ OPENGL Z JAZYKA JAVA

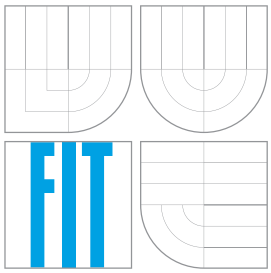
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL ŽIDEK

BRNO 2013

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

POUŽITÍ OPENGL Z JAZYKA JAVA

USE OF OPENGL FROM JAVA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MICHAL ŽIDEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN NAVRÁTIL
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce se zabývá problematikou použitelnosti jazyka Java v aplikacích využívajících grafickou knihovnu OpenGL. Poskytuje základní informace o samotné knihovně OpenGL a o nejběžněji používaných způsobech jejího propojení s jazykem Java. Cílem je také porovnat výkon Javy při vykreslovaní 3D scén s jazykem C++. K tomuto účelu byli vytvořeny dva testovací programy, jeden v Javě a jeden v C++. Oba programy byli podrobeny sadě testú, které změřili výkon implementací při vykreslování stejné scény.

Abstract

This thesis deals with the usability of Java in applications that use OpenGL graphics library. It provides basic information about the OpenGL library itself, and most commonly used methods to bind it with the Java language. It also aims to compare the performance of Java with C++ in applications that render 3D scenes. Two test programs were developed for this purpose, one in Java and one in C++. Both programs were tested in a set of tests, to measure the performance of both implementations while rendering the same scene.

Klíčová slova

OpenGL, Java, Java Native Interface, JOGL, 3D zobrazování

Keywords

OpenGL, Java, Java Native Interface, JOGL, 3D rendering

Citace

Michal Židek: Použití OpenGL z jazyka Java, bakalářská práce, Brno, FIT VUT v Brně, 2013

Použití OpenGL z jazyka Java

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Navrátila

.....
Michal Židek
15. května 2013

Poděkování

Na tomto mieste by som rád poďakoval vedúcemu bakalárskej práce Ing. Janovi Navrátilovi a zadávateľovi práce Ing. Pavlovi Tišnovskému, Ph.D. za cenné pripomienky a rady, ktorými prispeli k dokončeniu tejto práce.

Obsah

Kapitola 1

Úvod

Cieľom tejto práce je oboznámiť čitateľa s programovaním OpenGL aplikácií v jazyku Java a zistiť, aký vplyv má použitie jazyka Java na výkon 3D aplikácií využívajúcich OpenGL API. Najrozšírenejšie knižnice sprístupňujúce OpenGL v jazyku Java sú dnes JOGL a LWJGL. Obe k tomuto účelu používajú technológiu Java Native Interface (JNI), ktorá predstavuje dodatočnú réžiu oproti jazykom prekladaným do strojového kódu, ako je napríklad C alebo C++.

Stručný popis základných vlastností grafickej knižnice OpenGL, ako aj predstavenie niektorých fundamentálnych princípov spojených s touto knižnicou je v kapitole 2. Najpoužívanejšie technológie, ktorými je možné sprístupniť grafickú knižnicu OpenGL v programoch napísaných v Jave sú zhrnuté v kapitole 3. V tejto kapitole sú tiež poskytnuté základné informácie o technológii Java Native Interface (JNI), ktorá umožňuje programom v Jave používať funkcie napísané v iných programovacích jazykoch. Použitie JNI je demonštrované na jednoduchom príklade.

V kapitole 4 je stručne popísaná implementácia jednoduchého testovacieho programu na vykreslenie 3D scény naprogramovaného v Jave s použitím OpenGL a takmer identického programu v C++. Tieto programy demonštrujú základné vlastnosti OpenGL pri vykresľovaní 3D scén, ako sú hraničný popis geometrie priestorových telies, textúrovanie, osvetľovací model, double buffering a podobne. V Jave je na sprístupnenie OpenGL použitá knižnica JOGL.

V kapitole 5 sú tieto programy použité ako základ pre sadu testov, ktorých cieľom je zistiť vplyv použitia Javy v OpenGL aplikáciách na rýchlosť vykresľovania 3D scény. Implementácia v Jave tu bude porovnávaná s implementáciou v C++.

V závere sa diskutuje stav práce, dosiahnuté výsledky a možnosti ďalšieho vývoja a použitia vytvorených programov.

Kapitola 2

Grafická knižnica OpenGL

OpenGL je v súčastnosti najpoužívanejším API pre tvorbu 2D a 3D grafiky. Jedná sa o dobre zdokumentovanú otvorenú špecifikáciu, ktorá sa stala priemyselným štandardom vhodným pre široké spektrum aplikácií, akými sú napríklad vizualizácia medicínskych a vedeckých dát, virtuálna realita, letecké, vojenské či závodné simulácie alebo počítačové hry. Implementáciu OpenGL je možné nájsť v rôznych formách na takmer každej modernej platforme, ktorá umožňuje zobrazovať 3D grafiku.

OpenGL špecifikácia nediktuje ako má byť realizované zobrazovanie, ktorá časť výpočtu má prebiehať na grafickom akcelerátore, ktorá na procesore. To sú všetko detaily konkrétnej implementácie OpenGL, ktoré sú v rukách jej autorov, čo sú napríklad výrobcovia grafických kariet alebo vstavaných zariadení, ktoré sú schopné akcelerovať OpenGL. Existujú aj čisto softwarové implementácie OpenGL, ktoré umožňujú zariadeniam bez grafického akcelerátoru spúšťať OpenGL aplikácie[1]. Typicky však OpenGL predstavuje spôsob ako využiť výkon grafického akcelerátoru, čomu je celkový dizajn API prispôsobený. To platí najmä pre novšie verzie OpenGL využívajúce programovateľnú pipeline pomocou shaderov.

2.1 Grafická pipeline a proces rasterizácie

Aby prechod od fixnej k programovateľnej pipeline, ktorému je venovaná nasledovná podkapitola, bol zrozumiteľnejší, je dobré aspoň konceptuálne vedieť, z akých krokov grafická pipeline pozostáva, a ako prebieha proces rasterizácie 3D scény na 2D obrazovku, čomu sa venuje táto podkapitola. Jedná sa o zložitý proces a tento dokument nemá ambície ho vysvetliť dopodrobna, ale poskytuje zjednodušený pohľad na problematiku, ktorý je vhodný pre programátora OpenGL aplikácií.

Pre potreby tohoto textu budú rozlíšené nasledovné fázy grafickej pipeline:

- per-vertex operácie,
- odstránenie/odrezanie neviditeľných častí),
- rasterizácia,
- per-fragment operácie,
- zápis do framebufferu.

[1] Mesa 3D[15] je open source implementácia OpenGL, ktorá umožňuje čisto softwarové zobrazovanie. Zdrojový kód tohoto projektu je bohatým zdrojom algoritmov a techník používaných v 3D a 2D grafike.

Jednotlivé fázy a ich funkcie budú demonštrované na príklade spracovania jednoduchej scény s dvoma trojuholníkmi. Trojuholník sa môže zdať ako veľmi zjednodušený príklad, no jedná sa o najčastejšie spracovávaný útvar v 3D grafike. Najmä vo fáze rasterizácie to má svoje výhody, nakoľko existuje množstvo optimalizovaných techník aplikovateľných iba na rasterizáciu trojuholníkov. Iné polygóny sa typicky najskôr prevedú na trojuholníky, ktoré sa potom rasterizujú každý zvlášť, čo býva vo výsledku rýchlejšie než rasterizovať zložitejší polygón priamo, menej výkonnými všeobecnejšími technikami.

Prvou fázou sú per-vertex operácie. Ako názov napovedá, jedná sa o operácie vykonávané pre každý vrchol (anglicky vertex), teda sa vykonajú toľkokrát, koľko je v scéne vrcholov. V najjednoduchšom prípade sa iba prečítajú pozície vrcholov na vstupe a pošlú sa na výstup. Typicky je ale nutné vynásobiť súradnice vrcholov MVP maticou. MVP matica je kombináciou troch transformačných matíc:

- **Model** - transformuje súradnice zo súradnicového systému modelu do súradnicového systému sveta. Objekt v súradnicovom systéme modelu má centrum v počiatku súradnicovej sústavy. Pri načítaní 3D objektov sa objekty zvyčajne nachádzajú v tomto systéme. V súradnicovom systéme sveta môže byť objekt posunutý alebo rotovaný relatívne k počiatku scény a môže sa preto vyskytovať na rôznych pozíciach v scéne. Ak chceme s objektom pohnúť (zmeniť jeho súradnice v súradnicovom systéme sveta) modifikujeme túto maticu.

- **View** - transformuje súradnice zo súradnicového systému sveta do súradnicového systému kamery/pozorovateľa. V súradnicovom systéme kamery sa kamera nachádza na súradniciach $x = 0$, $y = 0$, $z = 0$ a sníma scénu v smere vektora *(0, 0, -1)*, teda predmety sa budú od kamery vzďaľovať s klesajúcou hodnotou súradnice z. Os x rastie zľava doprava, os y rastie v smere zdola nahor.

- **Projection** - projekčnou maticou sa transformujú súradnice zo súradnicového systému kamery do normalizovaného súradnicového systému s rozsahom všetkých osí od *-1* do *1*.

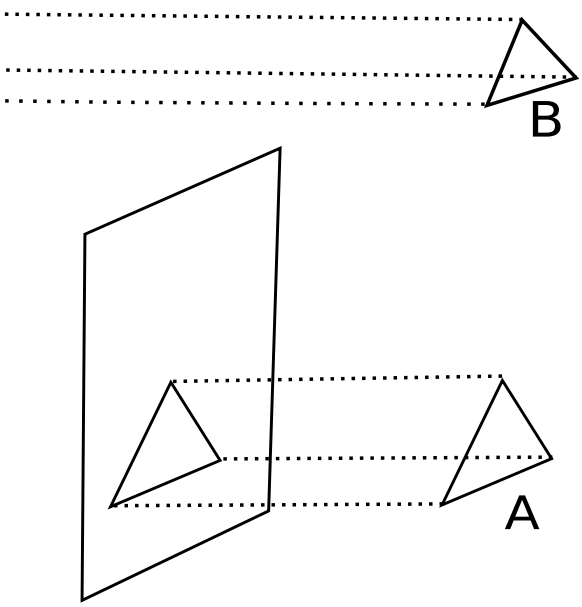
Zatiaľ čo prvé dve matice predstavujú jednoduché rotácie a posuvy, projekčná matica je trochu zaujímavejšia. Ak z normalizovaného súradnicového systému odstránime os z, získame normalizované 2D súradnice vrcholov vo finálnom obraze scény. Jedná sa o analógiu projekcie na priemetňu ležiacej tesne pred kamerou. Rozlišujeme dva typy projekcie:

- pravouhlá projekcia,

- perspektívna projekcia.

Pravouhlá projekcia (tiež ortogonálna projekcia) je znázornená na obrázku 2.1 . Na priemetňu sa dostanú iba tie vrcholy, ktoré pretína aspoň jedna z kolmíc na jej plochu. Tieto vrcholy sa zobrazia na mieste, kde sa kolmica pretne s priemetňou. Vrcholy trojuholníka A z obrázka 2.1 sa na priemetni zobrazia, zatiaľ čo vrcholy trojuholníka B nie. Z výsledného obrazu na priemetni sa nedá zistiť, ako ďaleko od priemetne sa trojuholník nachádzal, nakoľko všetky vrcholy odlišujúce sa iba súradnicou z sa na priemetni zobrazia do toho istého bodu. Pri pravouhlej projekcií sa tak z obrazu stráca informácia o hĺbke.

Obrázok 2.2 znázorňuje **perspektívnu projekciu**. Na rozdiel od pravouhlej projekcie sa zohľadňuje zorné pole kamery (anglicky field-of-view) a to ako v smere osi x, tak y. Vďaka tomu sa na rozdiel od pravouhlej projekcie na priemetni zobrazí aj trojuholník B.



Obrázek 2.1: Pravouhlá projekcia

Podobne ako by sme očakávali pri pozorovaní scény v reálnom svete, tak aj pri perspektívnej projekcii sa vzdialenejšie objekty javia menšie než objekty bližšie ku kamere. Prečo je tomu tak naznačuje obrázok 2.3. Obe úsečky na obrázku sú rovnako veľké, avšak pomer veľkostí podobných trojuholníkov CA'B' ku CAB a CX'Y' ku CXY je rozdielny, preto majú úsečky AB a XY rôzne veľké obrazy na priemetni.

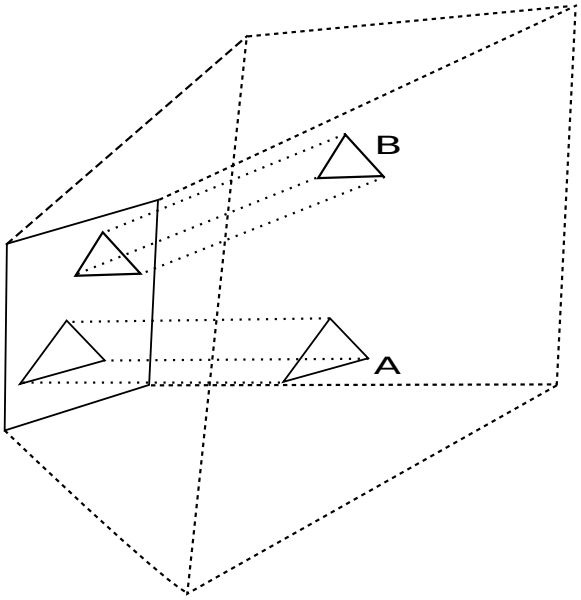
V starších verziách OpenGL do verzie 2.0 bolo možné použiť funkciu glPerspective a glOrtho na vytvorenie perspektívnej a pravouhlej projekčnej matice. Tieto funkcie sú z novších verzií OpenGL odstránené a je nutné si ich vytvoriť alebo použiť ďalšiu knižnicu. Implementácia funkcie na vytvorenie projekčnej matice a niektorých ďalších transformačných matíc v jazyku Java je v zdrojových súboroch sady testov na priloženom DVD, prípadne je možné nahliadnuť do C++ zdrojových súborov projektu GLM [19].

Po zistení pozícií vrcholov na priemetni sa prechádza do ďalšej fázy, ktorou je **odstránenie neviditeľných častí scény**. Jedná sa o vrcholy, ktoré sa po aplikácií projekčnej matice nevtesnali do rozmedzia *-1, 1* na niektorej z osí a teda nie sú v zornom poli kamery. Špeciálna pozornosť je venovaná trojuholníkom, ktoré sa nachádzajú v zornom poly iba čiastočne. V miestach, kde sa orezávajú tieto trojuholníky, je treba vytvoriť nové vrcholy. Bez toho, by totiž došlo k strate informácie o ploche trojuholníka a rasterizácia by bola neúplná. V tejto fáze sa tiež typicky odstraňujú odvrátené plochy.

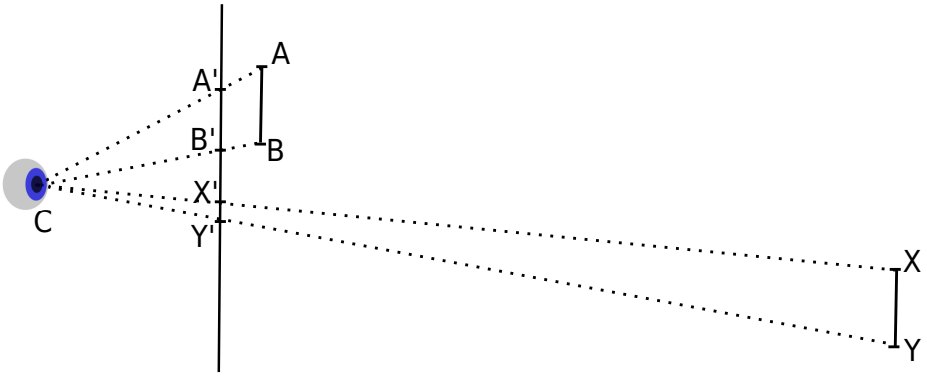
Ďalšou fázou je **rasterizácia**. Tá spočíva v identifikovaní fragmentov, ktoré treba vyfarbiť. Nie vždy si totiž želáme vyfarbiť celé vnútro trojuholníka. Obrázok 2.4 ukazuje niektoré možné spôsoby voľby fragmentov ako aj názvy OpenGL konštánt, ktoré každú metódu identifikujú. Existuje viacero algoritmov, ktoré sa používajú pri identifikovaní týchto fragmentov. Niektoré z nich je možné nájsť v [24], vrátane algoritmov používaných pri rasterizácii iných primitív[2].

Samotnú farbu fragmentu určíme v ďalšej fáze, ktorou sú **per-fragment operácie**.

[2]Publikácia [24] poskytuje systematický pohľad na problematiku počítačovej grafiky od základných pojmov a princípov, po pokročilejšie témy, pričom sa neviaže na žiadnu konkrétnu technológiu. Odporúčam každému záujemcovi o počítačovú grafiku.



Obrázek 2.2: Perspektívna projekcia



Obrázek 2.3: Ďalší pohľad na perspektívnu projekciu

Vstupom je fragment identifikovaný v predošlej fáze, resp. jeho poloha. Výstupom je farba tohoto fragmentu. V najjednoduchšom prípade sa farba fragmentu získa interpoláciou farieb vrcholov, medzi ktorými fragment leží, prípadne interpoláciou ich súradníc do textúry. Jedná sa v podstate o ďalšiu fázu rasterizácie, ale má zmysel ju oddeliť, lebo má špecifický význam, najmä v súvislosti so shaderami, nakoľko tieto operácie sú súčasťou tzv. fragment shaderu, ale tom viac až v podkapitole 2.2.2.

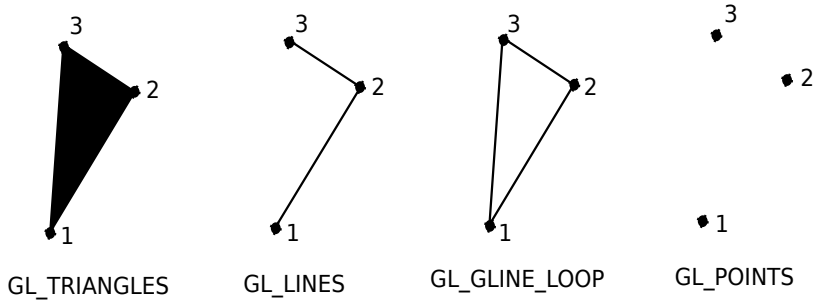
Poslednou fázou je **zápis do framebufferu**[3]. V prípade rasterizácie viacerých trojuholníkov je možné, že sa fragmenty budú prekrývať. Aby vo framebuffery zostala iba farba fragmentu, ktorý je bližšie ku kamere, prebehne **test na hĺbku (anglicky depth test)**. Tento test využíva pomocný buffer, tzv. depth buffer alebo z-buffer, s rovnakým rozmerom ako framebuffer, do ktorého ukladá hodnoty z-ovej súradnice príslušných fragmentov, ktorých farba je vo frambuffery. Vzdialenosť práve spracovávaného fragmentu sa porovná s hodnotou uloženou v depth buffery. Fragment sa zahodí ak je vo framebuffery uložená

[3]Do framebuffera sa ukladajú výsledné farby vrcholov. Jedná sa vlastne o bitovú mapu, do ktorej je postupne uložený obraz scény. Farba vo framebuffery môže byť uložená v rôznych formátoch.

farba fragmentu bližšie ku kamere.

Hodnoty súradníc z v depth buffery sú uložené až po aplikovaní projekčnej matice, teda ich hodnota je v rozmedzí od -1 do 1. Perspektívna projekčná matica namapuje vzdialenejšie objekty na súradnice bližšie pri sebe. Preto pre porovnanie súradníc z vzdialených bodov potrebujeme viac bitov v bunke z-bufferu, aby bolo možné rozlíšiť aj malé vzdialenosti medzi bodmi. Aby sa odstránil nepomer medzi rozlíšením vzdialeností blízkych a vzdialených objektov, je možné použiť tzv. w-buffer, ktorý porovnáva súradnice z predtým ako boli transformované projekčnou maticou. Tieto pôvodné hodnoty z môžu byť priamo uložené vo w-buffery, alebo je vo w-buffery uložená hodnota w, pomocou ktorej je možné z normalizovanej súradnice získať spätne pôvodnú hodnotu.

Na podobnom princípe ako depth test pracuje aj **stencil test**. Využíva stencil buffer rovnakých rozmerov ako framebuffer. Na rozdiel od depth bufferu, ktorého hodnoty sú ukladané automaticky, je možné do stencil bufferu ukladať hodnoty ručne a tiež meniť jeho hodnotu v priebehu programu podľa potreby. Hodnota v stencil buffery sa porovná s referenčnou hodnotou, na základe čoho môže stencil test uspieť alebo neuspieť. Neúspešný stencil test vylúči fragment zo spracovania. Referenčnú hodnotu ako aj funkciu na určenie úspešnosti stencil testu je možné nastaviť pomocou OpenGL funkcie *glStencilFunc*. Depth test aj stencil test sú vypnuté pokým ich programátor explicítne nezapne pomocou funkcie *glEnable*.



Obrázek 2.4: Voľba fragmentov podľa zvolenej metódy kreslenia

Farby fragmentov, ktoré uspejú v depth a stencil teste nakoniec vo framebuffere vytvoria výsledný obraz scény, ktorý sa môže zobraziť na obrazovku, prípadne uložiť pre ďalšie spracovanie alebo použiť ako textúra. Jednotlivé bunky vo framebuffere s farbami týchto fragmentov, určujú farbu pixelov výsledného obrazu.

Framebuffer môže byť rozdelený na ľavý alebo pravý. Pri použití double bufferingu tiež predný a zadný framebuffer. Programátor môže špecifikovať, do ktorého framebufferu chce kresliť. Pri použití double bufferingu sa kreslí implicítne do zadného framebufferu, zatiaľ čo na obrazovke sa zobrazuje predný framebuffer. Po vykreslení celej scény do zadného framebuffer sa zavolá funkcia na výmenu predného framebuffera za zadný. Zadný framebuffer sa tak stane predným a predný sa stane zadným. Týmto spôsobom je možné vyhnúť sa trhanému obrazu, ktorý spôsobuje postupné vykresľovanie scény na obrazovku.

2.2 Prechod od fixnej k programovateľnej pipeline

OpenGL sa od vydania verzie 1.0 (1992) značne pretvorilo. Výraznou zmenou bol príchod programovateľnej pipeline a predstavenie shaderov (OpenGL verzia 2.0, rok 2004). V neskorších verziách OpenGL sa použitie shaderov stáva nie len voliteľným rozšírením, ale

povinnou súčasťou, zatiaľ čo funkcie pracujúce s fixnou pipeline sú označené ako zastarané a nemali by sa používať v novo vznikajúcich aplikáciách[4]. Toto staré OpenGL sa dnes označuje ako Legacy OpenGL. Aby bolo možné lepšie pochopiť výhody moderného OpenGL, nasleduje krátka podkapitola venovaná Legacy OpenGL nasledovaná podkapitolou o modernom OpenGL.

2.2.1 Legacy OpenGL

Výhodou Legacy OpenGL bola nepopierateľne jednoduchosť. Vykresliť trojuholník na obrazovku (ekvivalent programu "Hello World!"pre grafické aplikácie) predstavoval iba zopár riadkov kódu[5], ako ukazuje príklad 2.1.

```
gl.glColor3f(0.0f, 0.0f, 1.0f);

gl.glBegin(gl.GL_TRIANGLES);
    gl.Color3f(1.0f, 0.0f, 0.0f);
    gl.glVertex3f( 0.0f, 1.0f, 0.0f);
    gl.Color3f(0.0f, 1.0f, 0.0f);
    gl.glVertex3f(-1.0f,-1.0f, 0.0f);
    gl.Color3f(0.0f, 0.0f, 1.0f);
    gl.glVertex3f( 1.0f,-1.0f, 0.0f);
gl.glEnd();
```

Príklad 2.1: Legacy OpenGL v Jave pomocou JOGL

Medzi príkazmi *glBegin* a *glEnd* sa nachádzajú príkazy *glVertex*, ktorými sa grafická pipeline plní vertaxami s danou pozíciou a aktuálne nastavenou farbou. Farba sa mení príkazom *glColor*. OpenGL funguje ako stavový stroj a príkazom *glColor* sa nastaví aktuálna farba a tá zostane rovnaká až do ďalšieho zavolania *glColor*. Podstatné je, že tento kus kódu je súčasťou programu vykonávaného vždy na procesore, a to bez ohľadu na to, či je k dispozícii grafický akcelerátor. Samotné vykreslenie trojuholníka nakoniec môže byť akcelerované, ale i tak je zaťaženie procesora významné. 3D scény môžu pozostávať zo stoviek tisíc vrcholov a nastaviť každý pomocou *glVertex* vedie na veľké množstvo volaní do OpenGL API na strane procesora.

Za povšimnutie stojí fakt, že farba sa programátorom nastavuje pre každý vrchol a pre jednotlivé fragmenty sa získa interpoláciou z farieb vrcholov, medzi ktorými sa nachádza. V prípade použitia textúry sa podobne budú interpolovať súradnice pozície farby v textúre, ktorá sa má pre fragment použiť. Tento proces prebieha automaticky a Legacy OpenGL neposkytuje programátorovi prostriedky ako do tohoto procesu vstúpiť.

Dobrým materiálom na oboznámenie sa s Legacy OpenGL sú NeHe tutoriály[16], ktoré boli portované do množstva rôznych jazykov, vrátane Javy, prípadne staršia verzia oficiálneho sprievodcu[7] k OpenGL.

2.2.2 Moderný prístup s využitím shaderov

V prípade použitia shaderov by bol program na vykreslenie trojuholníka o niečo zložitejší a predstavuje podstatne viac riadkov kódu. Je v ňom nutné urobiť tieto kroky:

[4] Aj keď z dôvodu spätnej kompatibility so starším hardvérom je vhodné použiť staršie OpenGL API ako zálohu pre prípad, že sa nedetekuje podpora shaderov.

[5] Ukážka je iba výsek z kódu, kompletný kód programu je možné nájsť na priloženom DVD.

- vytvorenie vertex buffer objektu so súradnicami vrcholov a ich farbami,
- vytvorenie vertex shaderu,
- vytvorenie fragment shaderu,
- zlinkovanie vertex a fragment shaderu do jedného shader programu,
- previazať vertex shader s dátami v hlavnom programe,
- poslať dáta z vertex buffer objektu do grafickej pipeline.

Najdôležitejšími zmenami, na ktoré sa zameriame sú použitie buffer objektu a možnosť vytvoriť **vertex a fragment shader.**

Shadery sú programy, ktoré prebiehajú na grafickom akcelerátore, ktorý pri ich vykonávaní využíva masívny paralelizmus. V OpenGL sa pri ich programovaní používa jazyk GLSL. Jedná sa o jazyk vyššej úrovne podobný C. Má preddefinované niektoré dátové typy bežne používané v shaderoch, ako sú napríklad matice a vektory a poskytuje množstvo operácií nad nimi. Pomocou vertex shaderu je možné predefinovať fázu grafickej pipeline, kde sa vykonávajú per-vertex operácie a pomocou fragment shaderu fázu per-fragment operácií. Toto otvára nové možnosti, ktoré v Legacy OpenGL neboli k dispozícií. Množstvo grafických efektov je tak možné vytvoriť oveľa jednoduchšie.

Do vertex shaderu sa z hlavného programu predávajú atribúty pomocou tzv. vertex buffer objektov. K vertex buffer objektu je asociované pole s dátami, ktoré v hlavnom programe vytvoríme. Napríklad je možné pripraviť pole so súradnicami vertexov, vytvoriť vertex buffer objekt a definovať, že pripravené pole predstavuje dáta vytvoreného buffer objektu. Niekoľkými OpenGL volaniami je tak možné predať do vertex shaderu informácie o všetkých súradniciach vrcholov v scéne. To je podstatný rozdiel oproti fixnej pipeline, kde sa takéto údaje zadávali pre každý vertex zvlášť, čo malo negatívny dopad na výkon. Do jedného vertex buffer objektu je navyše možné uložiť viac druhov informácií, napríklad o farbe polygónu, normále, súradniciach do UV mapy a podobne.

Exituje množstvo nekvalitných zdrojov informácií o modernom OpenGL, ktoré človeka skôr zmätú než poučia, preto uvediem aspoň zopár zdrojov, ktoré sa oplatí pri oboznamovaní sa s OpenGL prečítať, aby prípadní záujemcovia nenarazili na tie horšie kúsky ako prvé. Veľmi dobrý tutoriál je k dispozícií tu [21]. Tutoriálovým spôsobom je tiež písaná vynikajúca internetová kniha [5], ktorá poteší najmä majiteľov grafických kariet podporujúcich OpenGL 4, ale veľká časť tejto knihy je písaná kompatibilne s OpenGL 3.3. Ďalšími dobrými zdrojmi sú internetová kniha [6] a tutoriály na Wikibooks [20]. Určite existujú aj ďalšie dobré zdroje.

Kapitola 3

OpenGL a jazyk Java

Java je moderný programovací jazyk využívaný programátormi aplikácií rôzneho druhu. Je teda logické, že vznikla potreba previazať Javu s OpenGL a umožniť tak vývoj grafických aplikácií využívajúcich toto API aj v Jave. Existuje viacero možností ako OpenGL v programoch napísaných v Jave použiť. V tejto kapitole budú predstavné niektoré z nich. Konkrétne to bude Lightweight Java Game Library (LWJGL), Java Bindings for OpenGL (JOGL), Java3D, Xith3D a OpenGL for Java (GL4Java). Keďže sa v tejto kapitole viackrát spomína technológia JNI (Java Native Interface) tak si ju v prvej podkapitole stručne predstavíme.

3.1 Java Native Interfaface (JNI)

Pri vytváraní aplikácií v Jave má programátor k dispozícií širokú škálu knižníc a nástrojov napísaných priamo v Jave a vo väčšine prípadov si s nimi vystačí. Občas je ale nevyhnutné mať možnosť zavolať z programu v Jave kód napísaný v inom jazyku.

Technológia JNI umožnuje virtuálnym strojom, presnejšie kódu vykonávanému týmito strojmi, komunikovať s aplikáciami a knižnicami napísanými v iných programovacích jazykoch preložených do strojového kódu, ako sú C, C++ alebo Assembler.

Aby bolo ľahšie pochopiteľné ako JNI funguje, nasleduje jednoduchá ukážka. Bude v nej vytvorená funkcia *myFunction* napísaná v jazyku C, ktorá vypíše na obrazovku text *"Hello Java Native Interface!"*. Aby sme túto funkciu mohli zavolať z programu napísaného v Jave, je treba vykonať tieto kroky:

- vytvoriť triedu v Jave s metódou, ktorá bude reprezentovať C funkciu na strane Javy,
- spustiť nástroj **javah** na *.class* súbore tejto triedy, ktorý vygeneruje C hlavičkový súbor,
- implementovať samotnú funkciu v jazyku C,
- vytvoriť zdieľanú knižnicu s touto funkciou,
- spustiť program v Jave.

Príklad 3.1 ukazuje kód triedy v Jave s *native* metódou, v tomto prípade statickou. Táto metóda na strane Javy nemá implementáciu a bude vyhľadaná v zdieľanej knižnici. Tú je nutné načítať pomocou funkcie *System.loadLibrary()*, ktorá predaný parameter rozgeneruje

na platformovo špecifický názov zdieľanej knižnice, napríklad *lib*myfunc.*so* na Linuxe alebo myfunc.*dll* na Windows.

```
public class ExampleJNI {
    static {
        System.loadLibrary("myfunc");
    }

    static native void myFunction();

    public static void main(String[] args) {
        myFunction();
    }
}
```

Príklad 3.1: ExampleJNI.java: Ukážka natívnej funkcie v Jave

Po preložení tejto triedy do .class súboru treba spustiť nástroj **javah** a ako parameter mu predať názov triedy, čo vygeneruje hlavičkový súbor z príkladu 3.2.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class ExampleJNI */

#ifndef _Included_ExampleJNI
#define _Included_ExampleJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     ExampleJNI
 * Method:    myFunction
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_ExampleJNI_myFunction
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

Príklad 3.2: ExampleJNI.h: Výstup programu javah

Názov vygenerovanej funkcie pre jazyk C začína slovom *Java*, nasleduje názov triedy a názov metódy. Oddeľovačom je znak '_'. V prípade vnorenej triedy budú pred jej názvom najskôr vymenované všetky nadradené triedy oddelené opäť znakom '_'. V tomto prípade sa vygeneroval názov *Java_ExampleJNI_myFunction*. Použitie hlavičkového súboru vygenerovaného pomocou javah je vidieť v príklade 3.3.

Z kódu v tomto príklade treba vygenerovať zdieľanú knižnicu. Tú je potom nutné umiestniť tak, aby bola v ceste definovanej v premennej **java.library.path**. Po spustení

```
#include <stdio.h>
#include <jni.h>

#include "ExampleJNI.h"

JNIEXPORT void JNICALL
Java_ExampleJNI_myFunction(JNIEnv *env, jobject job)
{
    printf("Hello Java Native Interface!\n");
    return;
}
```

Príklad 3.3: myFunc.c: Použitie rozhrania vygenerovaného pomocou nástroja javah

programu ExampleJNI, by sa na obrazovku mal vypísať text *"Hello Java Native Interface!"*. Na priloženom DVD je k dispozící vyššie popísaný príklad aj s prekladovým a spúšťacím skriptom ukazujúcim jednotlivé kroky.

JNI samozrejme umožňuje predávať parametre do natívnych funkcií. Pri predávaní primitívnych dátových typov na strane Javy, je potrebné v natívnom kóde používať dátové typy s prefixom **j-**, napríklad jint, jfloat a podobne. Pri práci so zložitejšími dátovými typmi sa používajú funkcie dostupné prostredníctvom smerníka **env**, ktorý ukazuje na štruktúru **JNIEnv**. Ako je vidieť v príklade 3.3, je tento smerník k dispozícii v každej natívnej funkcii. Napríklad ak predáme do natívnej funkcie v Jave parameter *int[] intarr*, získame na strane natívneho kódu parameter *jintArray intarr*. Pomocou funkcie GetIntArrayElements, ktorej predáme intarr ako jeden z parametrov, získame smerník na dáta v poli, ktoré bolo predané z kódu v Jave.

Toto bol asi najjednoduchší možný príklad použitia JNI. Pre oboznámenie sa s pokročilejšími technikami je vhodné naštudovať špecifikáciu JNI[1], ktorá je dobre a zrozumiteľne napísaná.

JNI nie je vlastnosť jazyka Java, ale súčasť virtuálneho stroja JVM, a je teda k disozícií aj iným jazykom, ktoré su prekladané do javovského byte kódu.

3.2 Lightweight Java Game Library (LWJGL)

Lightweight Java Game Library (LWJGL), ako názov napovedá, je riešenie určené predovšetkým pre tvorcov hier. Podľa autorov tejto knižnice je LWJGL určené ako pre začiatočníkov, tak aj profesionálnych vývojárov. Knižnica je komplexným riešením pre vývojárov a preto okrem OpenGL umožňuje použiť tiež OpenCL (Open Computing Language) a OpenAL (Open Audio Library). Všetky tieto natívne knižnice sprístupňuje cez JNI. LWJGL ďalej sprístupňuje gamepady, herné volanty a joysticky. Autori LWJGL tvrdia, že ich cieľom nie je urobiť tvorbu hier jednoduchou, ale poskytnúť vývojárom nástroje, ktoré v Jave chýbajú, alebo nie sú dobre implementované, no sú kľúčové pri tvorbe hier. LWJGL tak v mnohom pripomína známu C/C++ knižnicu SDL (Simple Directmedia Layer), ktorá má podobné ambície.

LWJGL sa snaží pokryť čo najväčšie spektrum možných prípadov použitia, ale zároveň má ambície poskytnúť riešenie v čo najmenšom balení. Jedná sa o veľmi malú knižnicu, ktorá nezaberie veľa miesta v pamäti mobilných zariadení s obmedzenejšími zdrojmi. Zároveň ale nepodporuje niektoré vlastnosti, ktoré sú pri tvorbe 3D hier menej dôležité, ako napríklad

režim v okne. LWJGL primárne podporuje vykresľovanie v režime celej obrazovky, zatiaľ čo režim v okne je k dispozícií iba ako ladiaci mód a nemusí fungovať na všetkých platformách. To sa môže zdať ako veľký nedostatok, ale v skutočnosti tomu tak nie je. Na desktopoch väčšinou režim v okne funguje a na mobilných platformách nie je typické spúšťať hry iba na časti obrazovky.

LWJGL je v súčastnosti asi najpoužívanejšia knižnica pre tvorbu 3D hier v Jave. Je k dispozícii pod BSD licenciou čo umožnilo vznik množstva komerčných i nekomerčných hier a herných enginov. Na domovskej stránke projektu[13], je k dispozícii kvalitná dokumentácia s príkladmi, a tiež odkazy na množstvo projektov vytvorených pomocou LWJGL.

3.3 Java Binding for OpenGL API (JOGL)

Riešenie Java Binding for OpenGL API (JOGL) sa na rozdiel od LWJGL nezameriava iba na tvorbu hier, no aj tak v ňom vzniklo množstvo hier a herných enginov. Pomocou JNI sprístupňuje kompletné OpenGL API a väčšinu bežne používaných rozšírení. Je ľahko integrovateľné do existujúcich technológií ako sú AWT, Swing alebo SWT, čo umožňuje bezproblémovú manipuláciu v okne. OpenGL plátno sa môže vložiť na akúkoľvek polohu v okne, prípadne do rôznych ovládacích prvkov napríklad do tlačidiel.

JOGL je z veľkej časti vytvorený pomocou nástroja **GlueGen**, ktorý z hlavičkového súboru v jazyku C vygeneruje kód v Jave a JNI, ktorý je potrebný na spustenie natívnych C funkcí. Vďaka tomu, že je proces automatizovaný, môže byť s príchodom novej verzie OpenGL API a príslušného C hlavičkového súboru vytvorená aj nová verzia JOGL, ktroá bude toto API podporovať. JOGL teda nebýva pozadu s podporovanou verziou OpenGL. Na oficiálnych stránkach projektu[12] možno nájsť aj linky k súvisiacim projektom, ako je JOCL, sprístupňujúci OpenCL, a JOAL sprístupňujúci OpenAL. Tieto projekty tiež využívajú GlueGen. Ďalej je na stránkach možno nájsť odkazy na projekty, ktoré JOGL používajú.

JOGL je referenčnou implementáciu špecifikácie **JSR-231** (Java Specification Request)[1]. Vývoj JOGL začal v Sun Microsystems Game Technology Group, no od roku 2010 sa jedná o nezávislý open source projekt. K dispozícií je pod licenciou BSD.

3.4 Java 3D

Cieľom autorov Java 3D je poskytnúť jednoduché, ale výkonné API pre tvorbu 3D aplikácií. Na rozdiel od JOGL alebo LWJGL neposkytuje Java 3D API priamy prístup k OpenGL API, dokonca nemusí OpenGL ani využívať. Môže mať väzbu napríklad na DirectX, no na väčšine platformách je postavené práve nad OpenGL. Jedná sa o API na vyššej úrovni abstrakcie než OpenGL a obsahuje množstvo funkcií, ktoré by si programátor OpenGL musel implementovať sám alebo bol nútený použiť ďalšiu knižnicu. Výhodou Java 3D oproti LWJGL alebo JOGL, je čistejší objektový prístup. Scéna je v Java 3D reprezentovaná ako kolekcia objektov. Používa vysokoúrovňovú metódu popisu scény známu ako **scene-graph** (graf scény), ktorá zvyšuje robustnosť kódu, jeho čitateľnosť a uľahčuje tak jeho správu a to predovšetkým v prípade veľkých scén s množstvom objektov. Nevýhodou je práve nemožnosť pristupovať k nízkoúrovňovému OpenGL API, ktoré je vhodné napríklad

[1]Dokumenty JSR sú spravované v rámci Java Comunity Process (JCP) programu. Viac informácií o JCP je možno nájsť na oficiálnych stránkach [11]

pri optimalizácií alebo niektorých pokročilejších technikách. Java 3D implementuje API definované v dokumente JSR-926. Dokumentácia je k dispozícií na stránkach projektu [10].

3.5 Xith3D

Xith3D je ďalším riešením, ktoré reprezentuje scénu pomocou stromovej štruktúry, scenegraph, čím pripomína Java 3D. Xith3D však nie je iba API, ale 3D engine implementovaný v Jave pomocou LWJGL alebo JOGL. Okrem nástrojov na manipuláciu 3D scény poskytuje nástroje na načítanie modelov a textúr v rôznych formátoch, prácu so zvukmi, časticové efekty a mnoho ďalších funkcií. Xith3D je výkonný a dobre zoptimalizovaný nástroj, a je teda vhodný aj pre tvorbu 3D hier, čo bol jeho pôvodný účel[2], no je využívaný aplikáciami rôzneho druhu. Podobnosť s Java 3D umožňuje jednoduchú migráciu programov využívajúcich Java 3D na Xith3D. Jedná sa o open source komunitný projekt. Na oficiálnych stránkach projektu Xith3D[23] možno nájsť odkazy na projekty vytvorené týmto nástrojom.

3.6 OpenGL for Java (GL4Java)

Riešenie OpenGL for Java, známe skôr ako GL4Java, je jedným z prvých, ktoré OpenGL v Jave sprístupnilo. Podobne ako JOGL, aj GL4Java umožňuje prístup k OpenGL pomocou JNI, no v čase písania tejto práce iba k Legacy OpenGL. Na projekte sa už dlhšiu dobu aktívne nepracuje, je však naďalej dostupný, a to pod licenciou GNU LGPL. Pre novo vznikajúce projekty **je vhodnejšie použiť JOGL**, čo odporúčajú aj oficiálne stránky projektu GL4Java[18], nakoľko je aktívne vyvíjaný a hlavne umožňuje prístup k novým verziám OpenGL.

[2] Xith3D vznikol pôvodne ako engine pre hru Magicosm.

Kapitola 4

Implementácia 3D scény v Jave a v C++

V tejto kapitole bude popísaná implementácia programu na vykreslenie 3D scény v Jave s využitím knižnice JOGL a v C++ s využitím knižnice SDL. Okrem toho bude poukázané na hlavné rozdiely medzi oboma implementáciami. Výsledkom bude testovací program, ktorý bude použitý v sade testov na zmeranie rozdielov v rýchlosti vykresľovania tejto scény v oboch jazykoch. Program bude možné spustiť vo viacerých módoch líšiacich sa použitými shaderami, prípadne zapnúť použitie postprocesingových efektov využívajúcich vykreslovanie mimo obrazovku (anglicky offscreen rendering). Viac o spôsobe testovania je v kapitole 5.

4.1 Operácie s maticami v Jave

Pri programovaní 3D aplikácií s využitím OpenGL je často potrebné manipulovať s maticami, napríkald pri pohybe objektov alebo kamery. GLSL poskytuje väčšinu potrebných funkcií, no tie sú k dispozícií iba v shaderoch. Zmenu pozície kamery alebo objektu typicky chceme vypočítať už na strane hlavného programu, v našom prípade v Jave alebo v C++. V C++ je k dispozícií knižnica GLM[19], ktorá túto funkcionalitu pokrýva. V Jave bolo potrebné chýbajúcu funkcionalitu doprogramovať. Implementované boli iba operácie potrebné v tomto programe, konkrétne ide o tieto funkcie:

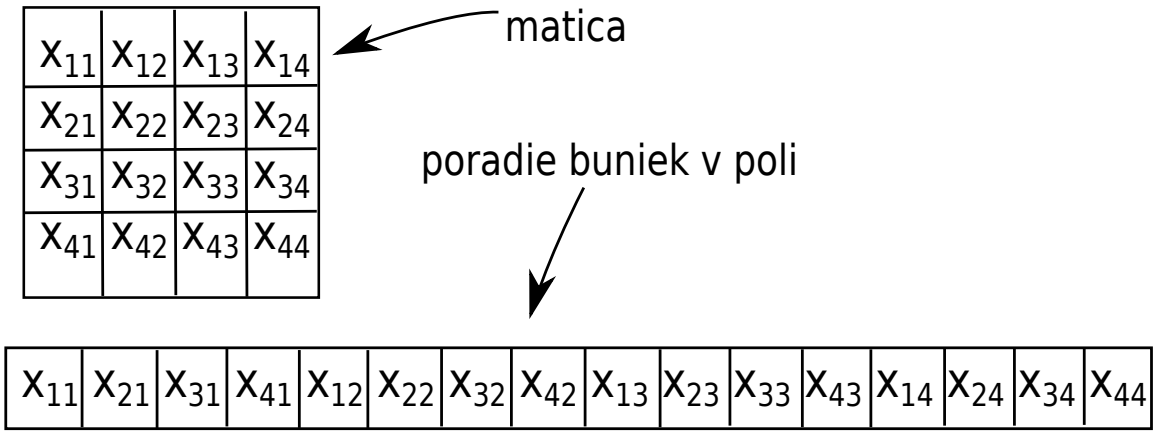
- násobenie matíc,
- vytvorenie posuvnej matice,
- vytvorenie matice na zväčšenie / zmenšenie objektu,
- vytvorenie rotačných matíc okolo osí x, y a z,
- ekvivalent k funkcii lookAt,
- vytvorenie perspektívnej projekčnej matice.

Všetky tieto funkcie budú implementované ako metódy triedy **mat4**[1]. Zdrojový kód je možné nájsť na priloženom DVD. Implementácia väčšiny metód je inšpirovaná zdrojovými

[1] Identifikátory tried v Jave sú zvyčajne písané v štýle CamelCase. Trieda mat4 je výnimkou aby jej názov pripomínal dátový typ mat4 z GLSL.

kódmi príslušných funkcií z GLM. Metóda *lookAt* je inšpirovaná implementáciou funkcie *gluLookAt* z projektu Mesa3D. Dobrým zdrojom informácií o transformačných maticiach je článok [4]. Princíp funkcie projekčnej matice je vysvetlený v článku [3].

Pri implementovaní vlastných funkcí pre prácu s maticami je dôležité dodržiavať konvencie OpenGL a ukladať matice do poľa po stĺpcoch, ako naznačuje obrázok 4.1. Funkcie v GLSL očakávajú matice v tomto formáte a pri jeho nedodržaní by museli byť v shaderoch prevedené do správneho tvaru, čo je samozrejme nežiadúce.



Obrázek 4.1: Usporiadanie buniek matice v poli

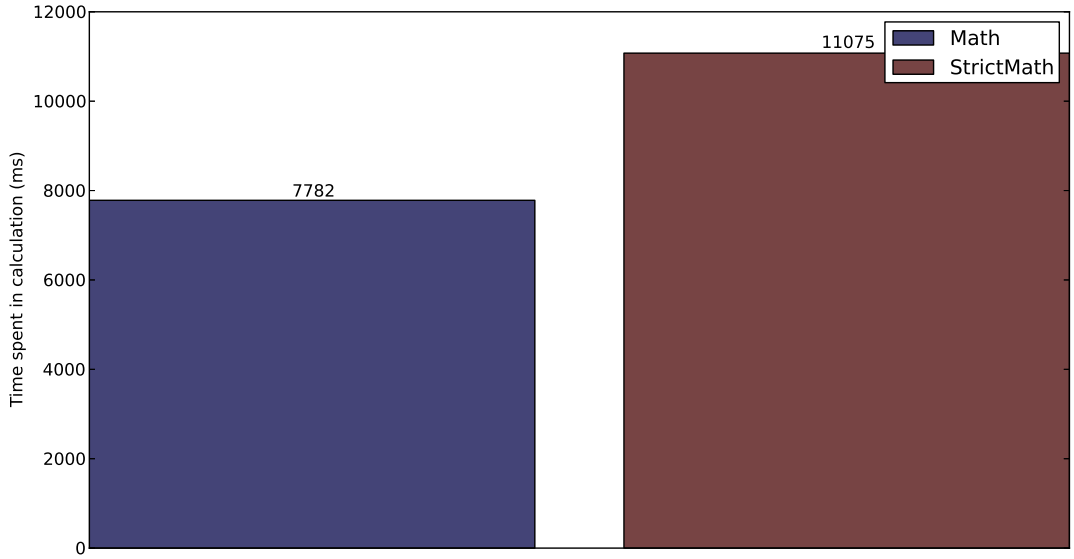
Usporiadanie po stĺpcoch sa môže zdať neprirodzené, preto boli do triedy mat4 pridané konštanty vo tvare *Mxy*, kde x je číslo riadku a y je číslo stĺpca, číslované od jednotky.

Maticové operácie sú tak frekventované, že je vhodné z nich odstrániť zbytočné skoky a vetvenia, aby bol výpočet čo najefektívnejší. Tento fakt je najviac viditeľný v metóde *mult*, ktorá násobí dve matice. Je v nej rozvinutý vnorený cyklus a je preto dlhšia.

Operácie s maticami využívajú niektoré matematické funkcie pre prácu s goniometrickými funkciami. Tieto funkcie sú v Jave k dispozícií v dvoch rôznych knižniciach: **Math** a **StrictMath**. Obe poskytujú rovnaké funkcie. Rozdiel je v tom, že StrictMath garantuje použitie algoritmov publikovaných v internetovej knižnici **netlib** [17], konkrétne v knižnici **fdlibm** (Free Distributable Math Library)[9]. Tam kde fdlibm poskytuje viacero algoritmov, je použitý ten, čo spĺňa štandard IEEE 754 [2]. Výhodou funkcií StrictMath je to, že pre rovnaký vstup vrátia na ľubovoľnej platforme vždy rovnaký výstup. Pri 3D scéne určenej na testovanie výkonnosti nie je nutné, aby sa pri výpočtoch hodnoty rovnali na bit presne na všetkých platformách. V tomto prípade je rozhodujúca rýchlosť výpočtu. Obrázok 4.2 ukazuje pomer dĺžky trvania výpočtu niekoľko miliónov maticových operácií implementovaný pomocou knižnice Math a StrictMath. Z výsledkov je jasné, že Math je rýchlejšia alternatíva, a preto je použitá vo výslednom programe.

4.2 Popis významných tried a ich vzťahy

V tejto podkapitole budú predstavené významné triedy, z ktorých pozostávajú oba testovacie programy, a bude vysvetlený ich význam. Triedy sú v oboch jazykoch pomenované rovnako, líšia sa však v niektorých detailoch, ktoré budú uvedené pri každej triede zvlášť.



Obrázek 4.2: Porovnanie rýchlosti maticových operácií pri použití Math a StrictMath

4.2.1 Object3D

Ako prvú si predstavíme triedu **Object3D**. Táto trieda reprezentuje objekty a ich geometriu. Metóda *loadObj* načíta geometriu zo súboru vo formáte OBJ[22], ktorý spracováva riadok po riadku. Metóda *loadObj* podporuje iba podmnožinu formátu OBJ, konkrétne je schopná načítať iba trojuholníkové plochy, UV súradnice a normály vrcholov. Jedná sa o minimum, ktoré je potrebné pre prácu s textúrami a osvetlením.

Ďalšou metódou triedy Object3D je *loadTexture*, ktorá náčita textúru pre konkrétny objekt. V C++ táto metóda využíva knižnicu DevIL [8]. V Jave je použitá trieda AWTTextureIO z JOGL a trieda ImageIO z balíčka javax.imageio. ImageIO nepodporuje toľko formátov ako DevIL, ale v tomto prípade to nebude prekážať.

Objekt triedy Object3D si uchováva informáciu o polohe v 3D scéne v zozname polôh. Jedná sa vlastne o zoznam transformačných matíc, ktoré prevedú súradnice vrcholov modelu zo súradnicového systému modelu do súradnicového systému sveta (modelová matica, anglicky model matrix). Ak chceme mať v scéne viac objektov s rovnakou geometriou, pridáme do tohoto zoznamu viac matíc metódou *addInstance*, ktorá preberá ako parameter začiatočnú polohu objektu v scéne. Jedna inštancia triedy Object3D teda môže reprezentovať viac objektov v scéne, no všetky musia mať rovnakú geometriu a textúru.

Metóda *initOnEvent* sa nachádza iba v Java implementácií. Táto metóda v podstate dopĺňa metódy *loadObj* a *loadTexture*. Obsahuje tie príkazy, ktoré vyžadujú platný OpenGL kontext. V C++ implementácií tieto príkazy môžu byť volané priamo z *loadObj* a *loadTexture*, nakoľko k načítaniu objektov dochádza až po inicializovaní OpenGL kontextu. Pri použití JOGL v AWT okne, nemusí byť OpenGL kontext platný po celú dobu behu programu. Preto je možné používať OpenGL/JOGL príkazy iba na niektorých miestach, kde je garantované, že OpenGL kontext je platný. Konkrétne je to možné v metódach *reshape*, *init*, *dispose* a *display* triedy GLEventListener. Programátor musí tieto metódy predefino-

vať[2]. Všetky OpenGL príkazy je možné použiť v týchto metódach alebo v metódach z nich volaných, ktorým je objekt uchovávajúci OpenGL kontext predaný ako parameter. Ak by došlo k deštrukcií tohoto objektu, bude v rámci jeho následnej konštrukcie zavolaná metóda *init*, do ktorej je pridané aj volanie metódy *initOnEvent* objektov triedy Object3D, čím sa dokončí ich inicializácia. Dôvod, prečo inicializáciu objektov triedy Object3D nie je vhodné vykonať kompletne v rámci volania *init*, kde je OpenGL kontext platný, je jednoduchý. Načítanie geometrie z OBJ súboru je časovo náročná operácia, ktorú nechceme spúšťať vždy, keď sa zavolá *init*.

4.2.2 ShaderProgram

Trieda ShaderProgram je bázovou triedou pre triedy, ktoré uchovávajú vertex a fragment shadere skompilované do jedného programu. Každá dvojica vertex a fragment shaderov implementujúca iný efekt, má definovanú vlastnú triedu odvodenú od tejto triedy. Ak v ďalšom texte bude reč o shader programoch na strane hlavného programu v C++ alebo v Jave, bude sa to vzťahovať na objekty tried, ktoré dedia z tejto triedy.

Metóda *attachObject* vezme existujúci objekt triedy Object3D a vloží ho do zoznamu objektov priradených shaderu programu, ktorý ju zavolal. Pri zavolaní metódy *execute*, budú vykreslené všetky objekty z tohoto zoznamu. Jeden objekt môže byť pomocou *attachObject* pridaný do zoznamov viacerých shader programov. Metóda *execute* musí byť predefinovaná v dcérskej triede, nakoľko každý shader program vyžaduje predanie iných atribútov do vertex shaderu.

V C++ je inicalizácia riešená kompletne v dcérskych triedach. V Jave obsahuje bázová trieda pomocné rutiny na načítanie súborov s GLSL kódom a kompiláciu vertex a fragment shaderov do jedného programu. Metóda *initOnEvent*, ktorú treba tiež predefinovať v dcérskych triedach, má rovnaký účel ako metóda s rovnakým menom v triede Object3D a prestavuje fázu inicializácie, kde sú potrebné príkazy OpenGL.

Z triedy ShaderProgram dedia tieto triedy: Wireframe, SimpleFill, SimpleTextures, PerVertexAdsFill a PerVertexAdsTex.

Trieda **Wireframe** slúži na vykreslenie tzv. **drôtového modelu**. V ňom sú jednotlivé vrcholy pospájané bielou čiarou a vnútro trojuholníkov nie je vyplnené. Podstatné je, že pri volaní *glDrawArrays* nie je v tomto prípade použitá konštanta *GL_TRIANGLES*, ale *GL_LINE_LOOP*, ako ukazuje príklad 4.1. Tento fakt je dôležitý, pretože vedie k veľkému množstvu OpenGL volaní pre vykreslenie celej geometrie (jedno volanie glDrawArrays pre každý trojuholník). V Jave teda pri volaní metódy *execute* triedy Wireframe, dôjde k veľkému počtu volaní cez JNI.

```
/* Vedie na arrsize/3 volaní glDrawArrays */
for (int b = 0; b < arrsize; b += 3) {
    gl.glDrawArrays(gl.GL_LINE_LOOP, b, 3);
}
```

Príklad 4.1: Vykreslenie drôtového modelu pomocou GL_LINE_LOOP

Trieda **SimpleFill** vyplní trojuholníky šedou farbou bez použitia osvetľovacieho modelu. Bez osvetľovacieho modelu vykresľuje aj **SimpleTextures**, no namiesto sivej farby použije textúru objektu. Triedy **PerVertexAdsFill** a **PerVertexAdsTex** vykresľujú objekty podobne ako predošlé dve, ale pridávajú navyše osvetľovací model. Skratka Ads v

[2]Minimálne musí poskytnúť prázdne metódy.

názve znamená ambient, difuse a specular. Názov má tiež naznačiť, že k výpočtu osvetlenia dochádza vo vertex shadere.

4.2.3 Scene

Trieda **Scene** spája všetky triedy do jedného celku. V metóde *init* sa volajú inicializačné metódy objektov scény, nastavia sa ich počiatočné pozície. Niektoré objekty tried dediacich od ShaderProgram sa v tu vložia do zoznamu aktívnych shader programov. Ktoré to konkrétne budú záleží na tom, s akými parametrami[3] bol program na vykreslenie scény spustený. Pri zavolaní metódy na vykreslenie scény sa zavolajú metódy *execute* všetkých shader programov v tomto zozname. V C++ sa v metóde *init* tiež inicializujú knižnice SDL a DevIL, v Jave sa inicializuje AWT okno a vytvorí sa OpenGL kontext. V prípade zapnutia postprocesingových efektov sa tu vytvorí pomocný framebuffer pre vykresľovanie mimo obrazovku.

V metóde *run* je implementovaná hlavná slučka programu. Testuje sa tu podmienka ukončenia programu a tiež tu prebehe meranie dĺžky trvania vykreslenia 1000 snímkov scény. V Jave je na tento účel použitá funkcia *System.currentTimeMillis*[4], ktorá vráti aktuálny čas v milisekundách. V C++ na bola použitá funkcia z SDL knižnice *SDL_GetTicks*. V každom cykle sa tiež volá metóda *update*, ktorá aktualizuje pozíciu kamery a objektov v scéne.

[3] Parametre sa počas testovania nastavujú automaticky skriptom napísaným v jazyku Python. Viac o testovaní je v kapitole reftestovanie.

[4] Java poskytuje tiež funkciu *System.nanoTime*, ktorá je vo väčšine implentácií JVM presnejšia, no v tomto prípade postačuje *System.currentTimeMillis*.

Kapitola 5

Testovanie

V tejto kapitole bude popísaný spôsob testovania a význam jednotlivých testov. Budú tiež uvedené výsledky testov a stručný komentár k nim. Súhrnné zhodnotenie všetkých testov je v podkapitole 5.2.

5.1 Rozbor testov

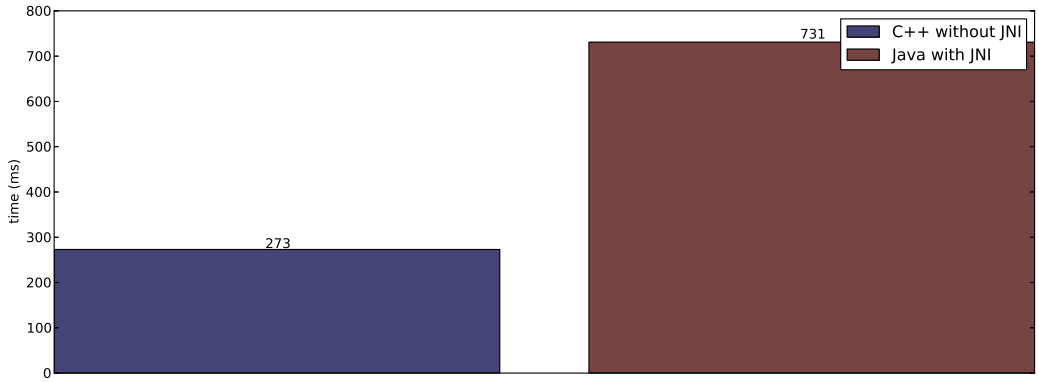
Programy na vykreslenie 3D scény v Jave a v C++, ktorých implementácia je popísaná v kapitole 4, nie sú spúšťané priamo, ale skriptom napísaným v jazyku Python. Tento skript automaticky spúšťa testovacie programy, nastavuje ich parametre a priebežne zbiera dáta. Na konci všetkých testov vykreslí graf, kde porovná priemerný počet snímkov za sekundu v jednotlivých testoch implementácie v Jave a v C++. Skript si tiež ukladá informácie o celkovej dĺžke trvania behu jednotlivých testov. Grafy sú vykresľované pomocou knižnice **matplotlib**[14].

5.1.1 Demonštrácia dodatočnej réžie na strane Javy

Ako bolo spomenuté v kapitole 3, JOGL používa JNI na sprístupnenie OpenGL API. Oproti OpenGL volaniam z programov napísaných v C, C++ alebo v jazyku symbolických inštrukcií vzniká pri JOGL dodatočná réžia súvisiaca so zapuzdrením OpenGL funkcií do objektu v Jave a použitím JNI. Prvý test nevykresľuje žiadnu scénu a slúži na demonštrovanie tejto dodatočnej réžie. Ide o zmeranie relatívneho pomeru dĺžky trvania dvadsiatich miliónov volaní OpenGL funkcie *glEnable*, ktorá v tomto teste vypína a zapína test na hĺbku. Táto funkcia bola zvolená preto, že sa jedná o veľmi rýchlu OpenGL operáciu a podiel času dodatočnej réžie tak bude významnejší. Výsledky testu sú vidieť na obrázku 5.1. Z obrázka je vidieť, že Java bola na testovacom stroji viac ako dvakrát pomalšia.

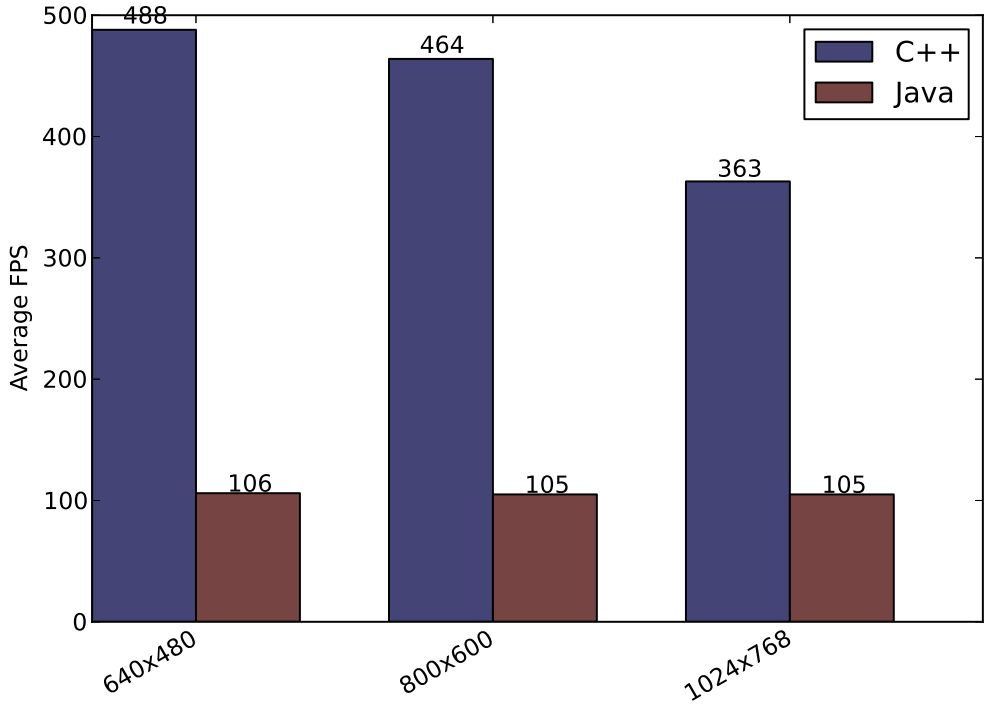
5.1.2 Drôtový model

Vykreslenie drôtového modelu je veľmi významný test, ktorý demonštruje spôsob použitia JOGL, ktorému je lepšie sa vyhnúť. Ako bolo spomenuté v kapitole 4 je drôtový model v testovacích programoch implementovaný tak, že pre každý trojuholník v scéne je zavolaná funkcia *glDrawArrays* s parametrom GL_LINE_LOOP. To vedie na obrovský počet volaní *glDrawArrays* a teda aj na spomalenie súvisiace s réžiu na strane Javy. Výsledok testu je na obrázku 5.2.



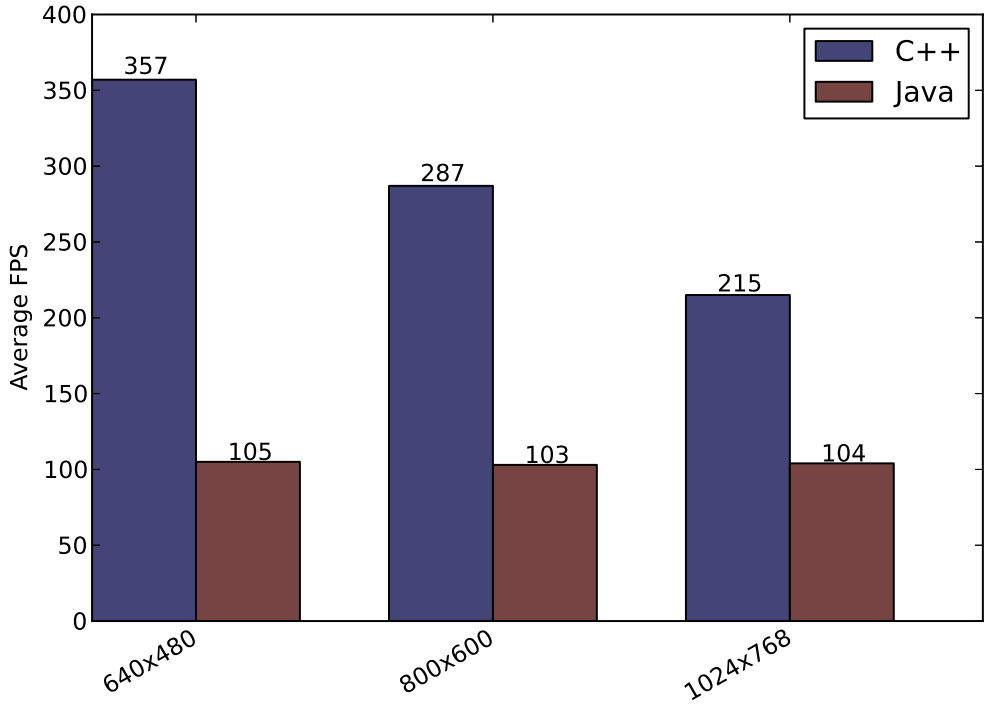
Obrázek 5.1: 20 miliónov volaní glEnable z C++ a v Jave pomocou JOGL

.



Obrázek 5.2: Drôtový model pomocou GL_LINE_LOOP.

.



Obrázek 5.3: Drôtový model a vyplnenie konštantnou farbou.

.

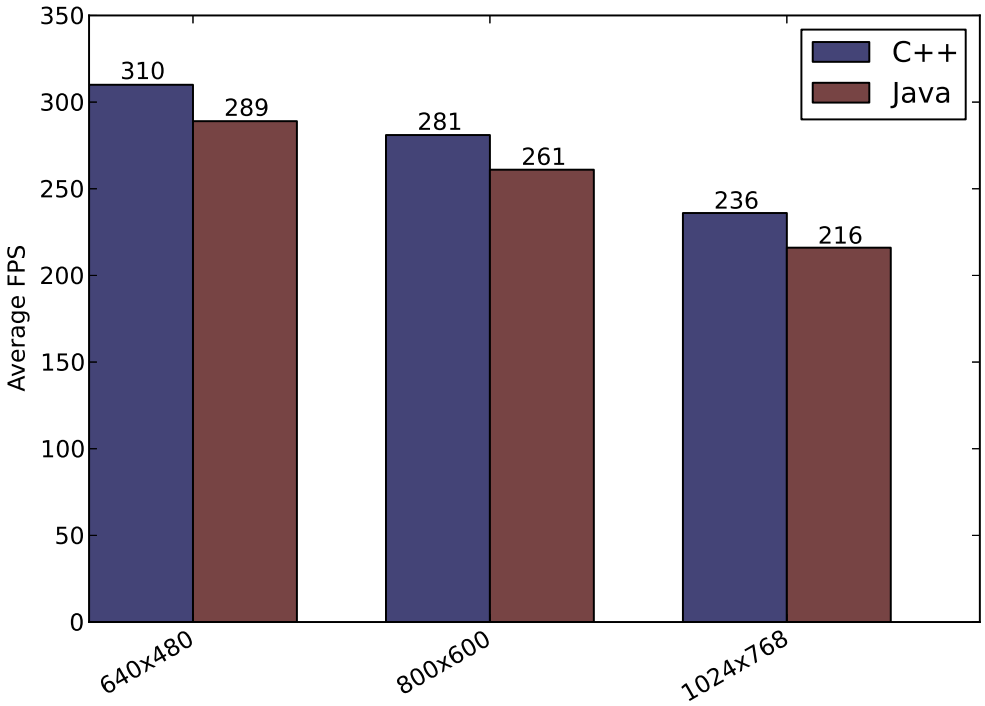
Podobne drastický rozdiel dostaneme aj pri použití Legacy OpenGL, ktoré tiež vedie na veľké množstvo OpenGL volaní.

5.1.3 Drôtový model a vyplnenie konštantnou farbou

V tomto teste je popri drôtovom modele vyplnené vnútro trojuholníkov konštantnou farbou. Fragment shader teda musí spracovať podstatne viac plochy, čo spôsobí väčšiu záťaž grafického akcelerátora. Výsledok testu ukazuje obrázok 5.3. Nakoľko Java má stále najväčší problém s drôtovým modelom, je spomalenie na strane procesora stále významnejšie než to na strane grafického akcelerátora. V C++ je to naopak. Z toho dôvodu je pokles počtu snímkov za sekundu oproti testu 5.1.2 viditeľný iba v C++ implementácií, no stále dosahuje C++ podstatne lepší výsledok.

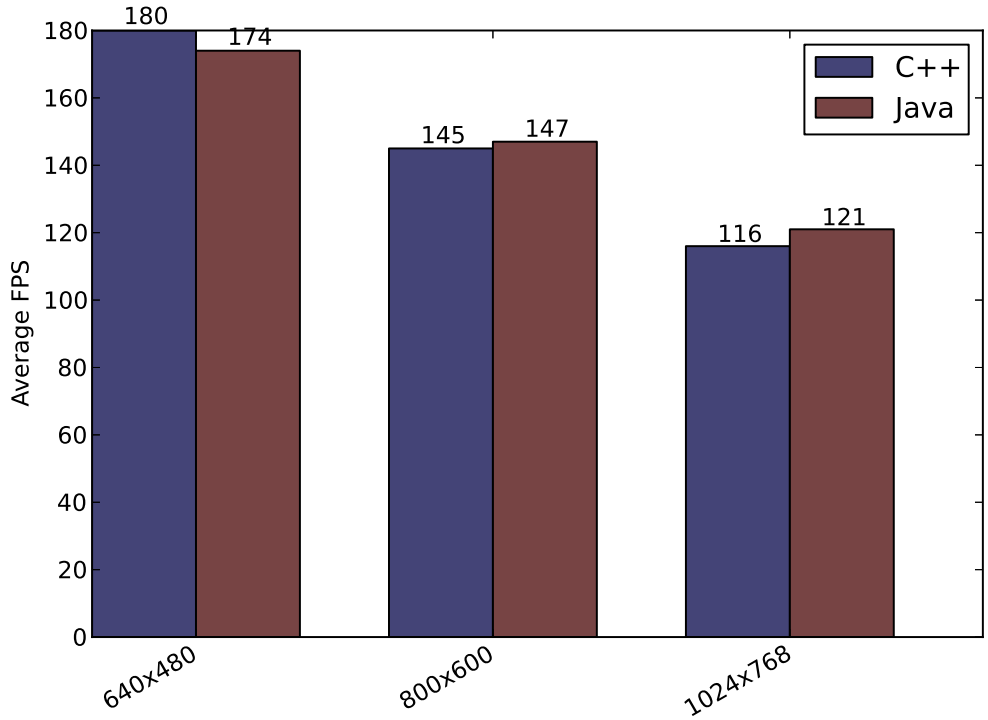
5.1.4 Osvetľovací model bez textúr

V tomto teste na strane grafického akcelerátora prebieha výpočet osvetlenia, čo spôsobuje oproti predošlým testom ďalšie zaťaženie a počet snímkov za sekundu v C++ implementácií opäť klesne. V Jave naopak počet snímkov za sekundu stúpne, pretože už nie je použitý pomalý drôtový model. Rozdiel medzi výkonom C++ a Java implementácie v tomto teste už nie je taký dramatický ako ukazuje obrázok 5.4.



Obrázek 5.4: Osvetľovací model bez textúr.

.



Obrázek 5.5: Použitie textúr bez osvetľovacieho modelu.

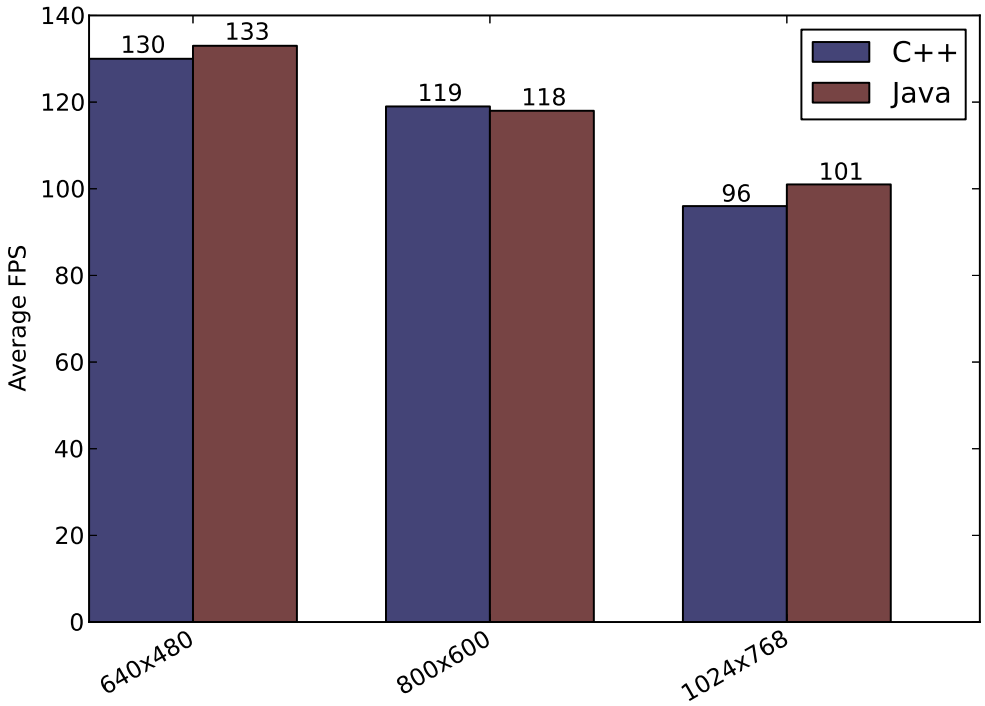
.

5.1.5 Použitie textúr bez osvetľovacieho modelu

Použitie textúr predstavuje zaťaženie predovšetkým vo fáze fragment shaderu, no keďže je v tomto teste vypnuté osvetlenie, celková záťaž grafického akcelerátoru je nízka. Od Java implementácie by sa dalo čakať, že v testoch, kde oba programy dosahujú vyššieho počtu snímkov za sekundu, bude výraznejšie zaostávať za C++ implementáciou. To z dôvodu, že pri nižšej záťaži grafického akcelerátoru má väčší podiel na celkovej rýchlosti dianie na procesore. Obe implementácie si však s tým to testom poradili približne rovnako dobre, ako ukazujú výsledky testu na obrázku 5.5. To je ďalšou známkou toho, že v oboch implementáciách je významnejšia doba strávená výpočtom na strane grafického akcelerátora a nie na strane programu v C++ alebo v Jave a to aj pri nižšej záťaži grafického akcelerátora, čo naznačuje, že oba jazyky pracujú veľmi efektívne.

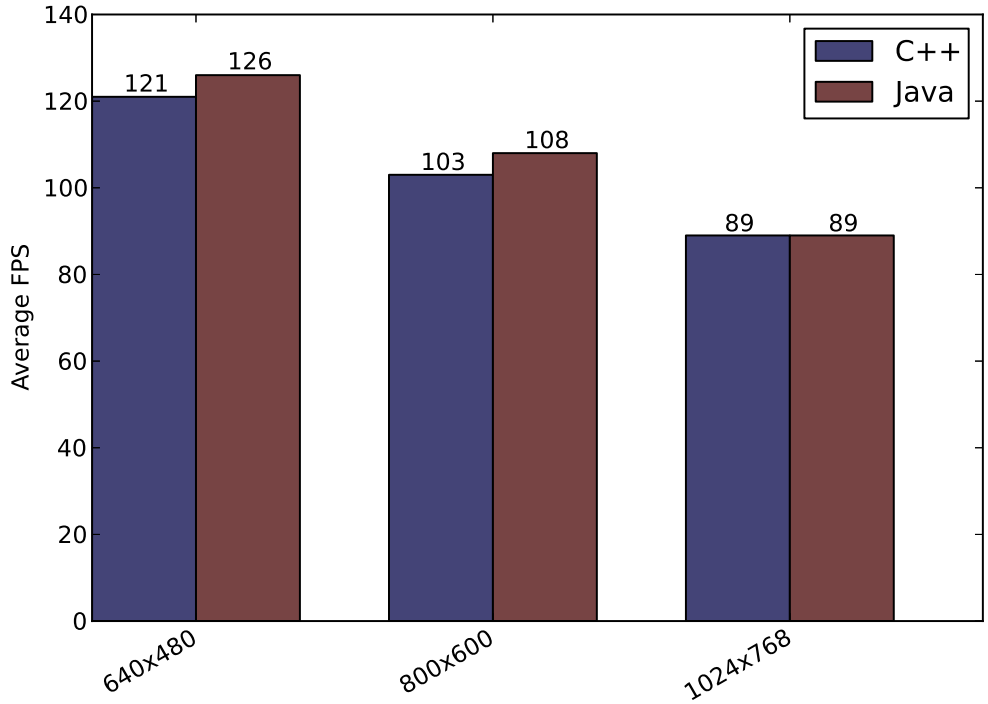
5.1.6 Použitie textúr s osvetľovacím modelom

Tento test pridáva k textúram aj výpočet osvetľovacieho modelu, čo spôsobí ďalšiu záťaž na grafickom akcelerátore. Graf na obrázku 5.6 ukazuje, že oproti predošlému testu klesla rýchlosť zobrazovania scény v oboch implementáciách približne rovnako. Vplyv dodatočnej réžie JOGL teda nezohral v tomto teste významnú rolu.



Obrázek 5.6: Použitie textúr s osvetľovacím modelom.

.



Obrázek 5.7: Zapnutie postprocesingových efektov.

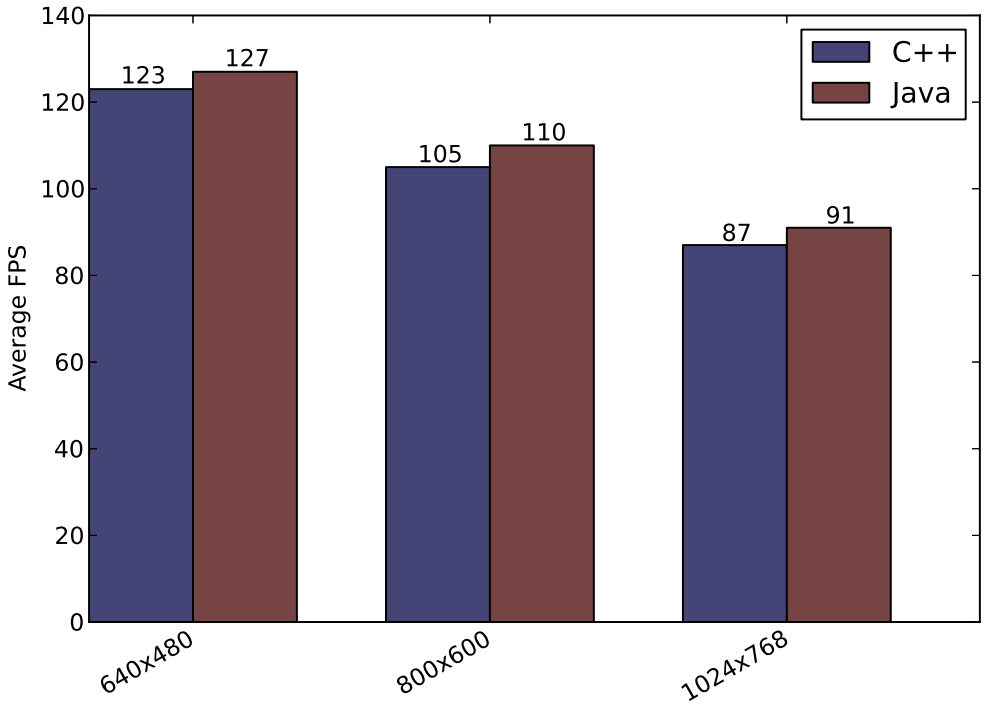
.

5.1.7 Zapnutie postprocesingových efektov

V tomto prípade sa jedná o dva podobné testy. V oboch sa zapne druhý pomocný framebuffer pre kreslenie mimo obrazovku. Do tohoto framebuffera sa vykreslí rovnaký obraz ako v predošlom teste. Tento obraz je použitý v ďalšej fáze kreslenia ako textúra štvorca prekrývajúceho obrazovku. Polovica textúry je vykreslená normálne, polovica je vykreslená akoby v nižšom rozlíšení (anglicky pixelation). V jednom teste je tento obdĺžnik statický, v druhom rotuje okolo osi y. Pri týchto testoch je opäť malý rozdiel medzi Java a C++ implementáciou v rýchlosti kreslenia scény na obrazovku, ako ukazujú výsledky na obrázkoch 5.7 a 5.8.

5.2 Zhodnotenie výsledkov testu

Pri použití Javy na programovanie aplikácií využívajúcich JOGL je nutné minimalizovať OpenGL operácie na strane hlavného programu. Pri veľkom počte OpenGL príkazov totiž dochádza k výraznému spomaleniu. Oveľa výraznejšiemu než pri použití C++, čo ukazujú testy 5.1.2 a 5.1.3. Pri náročnejších scénach, v ktorých je použité malé množstvo JOGL/OpenGL volaní, sa rozdiel vo výkone Javy a C++ podstatne zmenšuje, čo naznačujú výsledky zvyšných testov. Pri dobrom návrhu aplikácie by teda malo byť možné v Jave dosiahnuť porovnateľných rýchlostí zobrazovania ako v C++, ktoré je bez dodatočnej

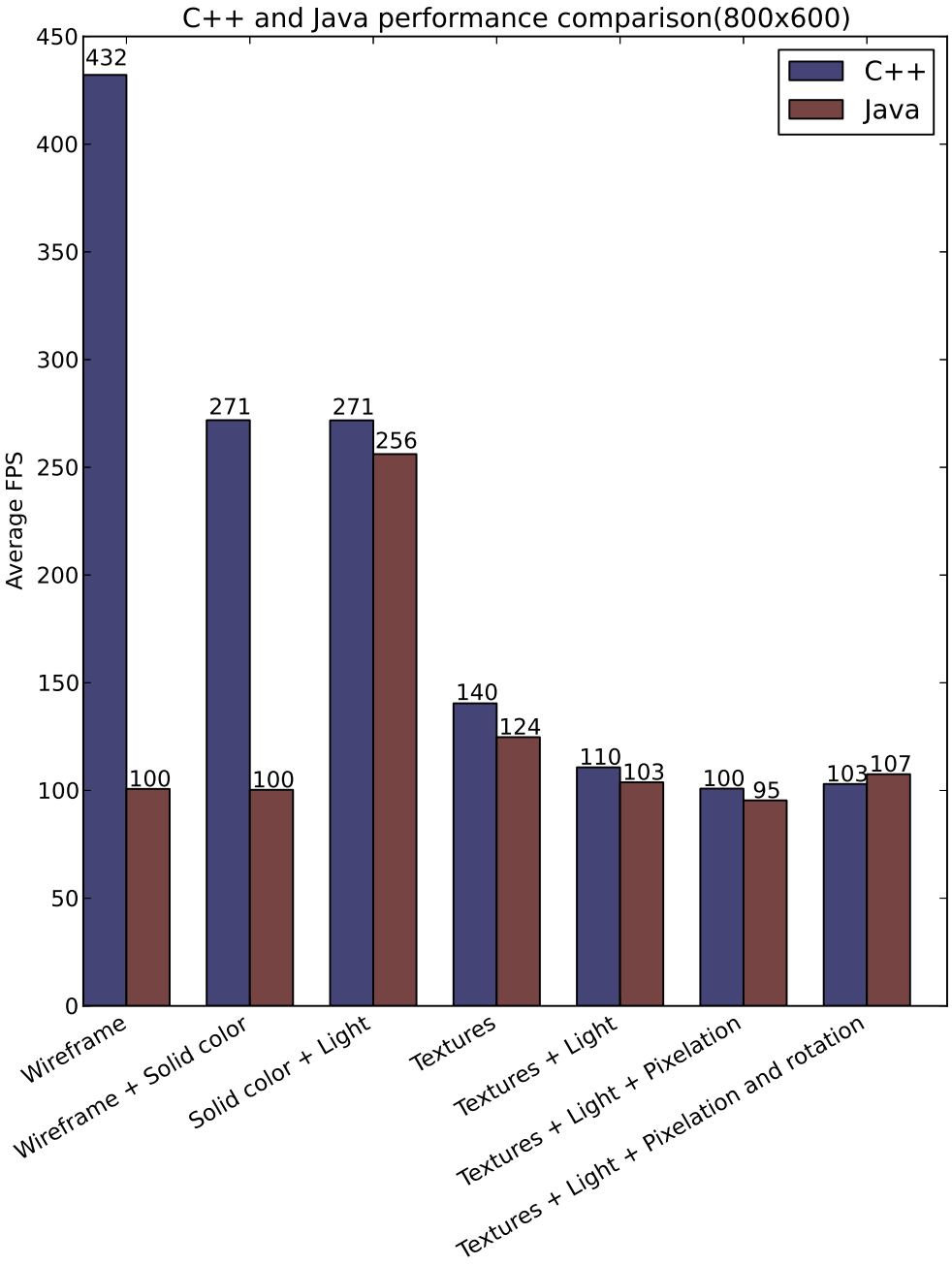


Obrázek 5.8: Zapnutie postprocesingových efektov a rotácie.

.

réžie JNI. Teda minimálne pokiaľ ide o zobrazovanie 3D scény. Obrázok 5.9 ukazuje súhrn všetkých testov pri rozlíšení 800x600 v jednom obrázku, okrem testu na demonštrovanie dodatočnej réžie JOGL.

Testovanie tiež ukázalo, že implementácia v Jave má výrazne dlhší nahrávací čas. V C++ implementácií trvalo načítanie scény na testovacej platforme menej ako 1 sekundu, v Jave to bolo miestami viac ako 6 sekúnd.



Obrázek 5.9: Súhrn testov v rozlíšení 800x600

.

Kapitola 6

Záver

Hlavným cieľom tejto práce bolo oboznámiť čitateľa s problematikou písania aplikácií využívajúcich OpenGL API v jazyku Java. V kapitole 2 bola stručne charakterizovaná OpenGL knižnica. Bola tiež naznačená činnosť grafickej pipeline v OpenGL a rozdiely medzi Legacy OpenGL a modernejšími technikami používanými v novších OpenGL. Touto kapitolou bol splnený prvý bod zadania práce. Druhý bod zadania bol splnený v kapitole 3. Boli v nej predstavené najznámejšie nástroje na prepojenie Javy a OpenGL ako aj technológia Java Native Interface, ktorá je niektorými týmito nástrojmi využívaná.

Vlastný prínos tejto práce spočíval vo vytvorení testovacích programov na vykreslenie 3D scény v Jave a v C++, ktoré slúžia na porovnanie efektivity Javy a C++ v 3D grafických aplikáciách. Implementácia týchto programov je popísaná v kapitole 4, ktorá je riešením bodu 3 zadania práce.

Z výsledkov testov v kapitole 5 vyplýva, že pri správnom použití knižnice JOGL je možné v Jave vytvoriť program, ktorý využíva knižnicu OpenGL rovnako, alebo približne rovnako, efektívne ako pri použití jazyka C++. To platí pre moderné spôsoby využitia OpenGL s použití shaderov. Pri použití Legacy OpenGL alebo takej techniky moderného OpenGL, ktorá vedie na veľký počet volaní do OpenGL API, je pokles výkonu veľmi výrazný. Využívanie shaderov zefektívňuje vykresľovanie pri luboľnom použitom jazyku, no v prípade Javy, kedy každé volanie OpenGL vyvolá volanie natívnej metódy v JNI, je nárast výkonu pri použití moderných techník oveľa výraznejší než napríklad v jazyku C.

Testy by bolo možné ďalej rozšíriť tak aby merali dĺžku trvania konkrétnych JOGL volaní spolu s počítaním, koľkokrát bola ktorá funkcia zavolaná. Tým by mohol byť vytvorený jednoduchý program na identifikovanie tých volaní OpenGL, ktoré spôsobujú najväčšie spomalenie programu.

Literatura

[1] Java Native Interface Specification. http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html, [Online], [cit. 2013-02-20].

[2] IEEE Standard for Binary Floating-Point Arithmetic. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=30711, 1985, digital Object Identifier: 10.1109/IEEESTD.1985.82928.

[3] Ahn, S. H.: OpenGL Projection Matrix. http://www.songho.ca/opengl/gl_projectionmatrix.html, [Online], [cit. 2013-02-20].

[4] Ahn, S. H.: OpenGL Transformation. http://www.songho.ca/opengl/gl_transform.html, [Online], [cit. 2013-02-20].

[5] Luten, E.: OpenGLBook.com. http://openglbook.com/, [Online], [cit. 2013-02-20].

[6] McKesson, J. L.: Learning Modern 3D Graphics Programming. http://www.arcsynthesis.org/gltut/, 2012, [Online], [cit. 2013-02-20].

[7] Shreiner, D.; Woo, M.; Neider, J.; aj.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2*. Addison-Wesley, páté vydání, 2005, ISBN 978-0321335739.

[8] WWW stránky: Developer's Image Library. http://openil.sourceforge.net/.

[9] WWW stránky: Free Distributable Math Library. http://www.netlib.org/fdlibm/.

[10] WWW stránky: Java 3D. https://java3d.java.net/.

[11] WWW stránky: Java Community Process. http://jcp.org/.

[12] WWW stránky: Java™ Binding for the OpenGL® API. https://jogamp.org/jogl/www/.

[13] WWW stránky: Lightweight Java Game Library. http://www.lwjgl.org/.

[14] WWW stránky: matplotlib: A ploting library for Python. http://matplotlib.org/.

[15] WWW stránky: The Mesa 3D Graphics Library. http://www.mesa3d.org/.

[16] WWW stránky: NeHe Productions. http://nehe.gamedev.net/.

[17] WWW stránky: The Netlib repository. http://www.netlib.org/.

[18] WWW stránky: OpenGL for Java. jausoft.com/gl4java/.

[19] WWW stránky: OpenGL Mathematics (GLM). http://glm.g-truc.net/.

[20] WWW stránky: OpenGL Programming.
http://en.wikibooks.org/wiki/OpenGL_Programming.

[21] WWW stránky: Tutorials for modern OpenGL (3.3+).
http://www.opengl-tutorial.org/.

[22] WWW stránky: Wavefront OBJ File Format Summary.
http://www.fileformat.info/format/wavefrontobj/egff.htm.

[23] WWW stránky: Xith3D. http://xith.org/.

[24] Žára, J.; Beneš, B.; Sochor, J.; aj.: *Moderní počítačová grafika*. Computer Press Brno, 2004, ISBN 80-251-0454-0.