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

DEMONSTRACE PROGRAMOVATELNÝCH SHADERŮ POMOCÍ KNIHOVEN OPENSCENEGRAPH A QT

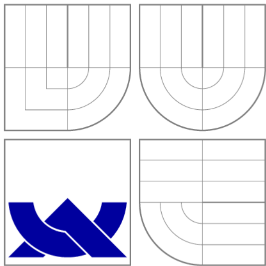
BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

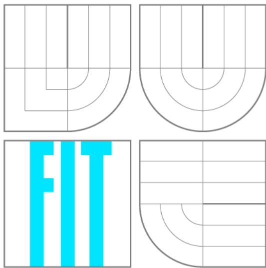
AUTOR PRÁCE PETER HARMAN

AUTHOR

BRNO 2010



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

DEMONSTRACE PROGRAMOVATELNÝCH SHADERŮ POMOCÍ KNIHOVEN OPENSCENEGRAPH A QT

SHADER DEMONSTRATION USING OPENSCENEGRAPH AND QT LIBRARIES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PETER HARMAN
AUTHOR

VEDOUCÍ PRÁCE — Ing. MIROSLAV ŠVUB
SUPERVISOR

BRNO 2010

Abstrakt

Úlohou této práce je přiblížit čtenáři práci s vertex a fragment procesorem. Programy pro tyto procesory se nazývají vertex a fragment shadery. Mohou být napsané v různých programovacích jazycích pro ně určených(*HLSL, Cg...*), avšak v práci bude diskutovaný jazyk `OpenGL Shading Language`(*GLSL*).
Budou demonstrované techniky pokročilého renderingu:

- Phongův model, Blinn-Phong model
- Lambertovo osvětlení, Gouraudovo tónování

Dále bude popsaná práce s knihovnou `OpenSceneGraph`, jako s knihovnou založenou na `OpenGL`, a její integrací s knihovnou na tvorbu uživatelského rozhraní `QT`. Výsledkem bude multiplatformová aplikace demonstrující propojení knihoven `QT` a `OpenSceneGraph` s integrovaným návodem popisujícím celý proces implementace. Nebudou chybět teoretické základy.

Abstract

Assignment of this work is to zoom in the work with vertex and fragment processor for users. Programs for these processors are called vertex and fragment shaders. They may be written in a various programming languages intended for them (*HLSL, Cg...*), however in the work is going to be discussed `OpenGL Shading Language` (*GLSL*). There are going to be demonstrated this techniques of advanced rendering:

- Phong shading, Blinn-Phong shading
- Lambert illumination, Gouraud shading

Later on is going to be described work with library `OpenSceneGraph`, as a library based on `OpenGL` and its integration with library for generating user's interface. The result is going to be a multiplatform application demonstrating connection between `QT` and `OpenSpaceGraph` libraries with integrated tutorial describing whole process of implementation. Theoretical background is going to be included as well.

Klíčová slova

- OpenSceneGraph, Qt, GLSL, Shadery, Vertex shader
- Fragment shader, Phongův model, Gouraudův model, Blinn-Phong

Keywords

- OpenSceneGraph, Qt, GLSL, Shaders, Vertex shader
- Fragment shader, Phong shading, Gouraud shading, Blinn-Phong

Citácia

Peter Harman: Demonštrácia programovateľných shaderov pomocou knižníc
OpenSceneGraph a QT, bakalárska práca, Brno, FIT VUT v Brně, 2010

Demonstrace programovatelných shaderů pomocí knihoven OpenSceneGraph a QT

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Miroslava Švuba.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Peter Harman
19. mája 2010

Poděkování

Chcel by som poďakovať vedúcemu práce, Ing. Miroslavovi Švubovi, ktorý mi poskytol všetky potrebné materiály a informácie, ako ja podporu, ktorú mi prejavoval po celú dobu písania práce.

© Peter Harman, 2010.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**
- 1.1 História 3
- 1.2 Cieľ projektu 3
- 1.3 Obsah jednotlivých kapitol 4

2 OpenGL Shading Language **5**
- 2.1 Úvod 5
- 2.2 Základy jazyka GLSL 6
  - 2.2.1 Dátové typy 6
  - 2.2.2 Typové kvalifikátory 7
  - 2.2.3 Vstavané funkcie 7
  - 2.2.4 Fragment a vertex shader 8
  - 2.2.5 Ďalšie rysy jazyka GLSL 8

3 OpenSceneGraph **10**
- 3.1 Úvod 10
- 3.2 História 10
- 3.3 Základy OpenSceneGraph 11
  - 3.3.1 Scene graph 11
  - 3.3.2 Zloženie OpenSceneGraph 11
  - 3.3.3 Knižnica `osg` 12
  - 3.3.4 Knižnica `osgViewer` 12
  - 3.3.5 Nodekits 13
  - 3.3.6 Zásuvné moduly 13
  - 3.3.7 Súradnicový systém 13

4 Qt framework **14**
- 4.1 Úvod 14
- 4.2 Základy Qt 15
  - 4.2.1 Qt moduly 15
  - 4.2.2 Projektové súbory 15
  - 4.2.3 Signály a sloty 16
  - 4.2.4 Vývojové nástroje 18

5 Teória vybraných techník **19**
- 5.1 Lambertov osvetľovací model 19
- 5.2 Gouraudovo tieňovanie 19
- 5.3 Phongov model 21

5.3.1 Phongov reflexný model 21
5.3.2 Phongova metóda tieňovania 22
5.3.3 Blinn-Phongova metóda tieňovania 23

6 Návrh a implementácia **24**
6.1 Požiadavky 24
6.2 Vývojové prostredie 24
6.3 Návrh aplikácie 25
6.4 Implementácia 25
6.4.1 Implementácia vrámci `Qt` 26
6.4.2 Implementácia vrámci `OpenSceneGraph` 28
6.4.3 Shader Programy 30

7 Záver **33**

A Obsah CD **35**

Kapitola 1

Úvod

1.1 História

Počítačová grafika behom posledných 40 rokov prešla priam až neuveriteľným vývojom. História grafických kariet sa začala písať v 70 rokoch, keď bola potreba previesť zobrazovanie informácii pomocou tlačiarni na ďalšiu, interaktívnejšiu úroveň. Prvé grafické karty pracovali len v textovom režime a boli len monochromatické. Neskôr sa objavili integrované obvody ANTIC a CTIA, ktoré využívali aj grafický mód na 8 bitových ATARI počítačoch. Výstup bol vedený do televízneho prijímača.

V 80 rokoch vydala firma IBM svoju kartu IBM Professional Graphics Controller, ktorá sa už začína podobať na novodobé grafické karty. Obsahovala vlastnú operačnú pamäť a procesor a umožňovala akceleráciu 2D a 3D grafiky pre CAD programy.
Pracovala s rozlíšením 640x480, 256 farbami a 60 obrázkami za sekundu.

Začiatkom 90 rokov vyústila potreba programovateľného grafického hardwaru k vzniku OpenGL ako profesionálneho grafického API(*application programming interfaces*). To umožnilo dizajnérom a programátorom lepšie využiť možnosti grafického hardwaru. Asi v polovici 90 rokov vzniklo s rozvojom hier a aplikácii na operačnom systéme MS Windows aj iné grafické API, a to DirectX.
Neskôr sa začlenila ku grafickým kartám aj podpora programovania pixel a vertex shaderov vo forme krátkych programov a vznikol jazyk GLSL (*OpenGL Shading Language*).

V súčasnosti smeruje vývoj grafických kariet smerom k fotorealistickej počítačovej grafike v realtime aplikáciách. Hnacím motorom tohto vývoja nových grafických kariet je hlavne herný sektor, ktorý tlačí na vývoj nového a výkonnejšieho hardware a k využitiu jeho veľkého potenciálu z hľadiska softwaru. Avšak netreba zabúdať na nového hráča na poli počítačovej grafiky a jej vývoja, ktorého predstavuje filmový priemysel a jeho inovátorský prístup ku grafickým efektom.[9]

1.2 Cieľ projektu

Cieľom mojej práce je vytvoriť multiplatformovú demonštračnú aplikáciu, ktorá pomôže čitateľovi pochopiť základné techniky pokročilého renderingu[1]. Tie budú realizované pomocou knižnice `OpenSceneGraph`, ktorá je objektovo orientovaná a je nadstavbou nad `OpenGL`.

Keďže je `OpenSceneGraph` objektovo orientovaná, čo mimo iné znamená, že zapuzdruje niektoré rysy `OpenGL`, a to umožňuje podstatne rýchlejšie vytvárať 3D scény, pridávať

[1]Rendering – proces generovania obrazu z modelu, pomocou počítačového programu.

osvetlenie alebo nastavovať jeho vlastnosti.
Pokročilé techniky si vyžadujú aj prácu s *fragment* a *vertex* shaderami, a preto bude diskutovaný aj jazyk `GLSL` a jeho využitie v spolupráci s knižnicou `OpenSceneGraph`.

Keďže jedna z požiadaviek je multiplatformovosť, tak *GUI*[2] bude realizované pomocou multiplatformového toolkitu[3] `Qt`. Nakoniec bude zostavený jednoduchý tutoriál[4], ktorý bude popisovať jednotlivé techniky jak z hľadiska teoretického, tak aj praktického. Tento tutoriál bude súčasťou aplikácie.

1.3 Obsah jednotlivých kapitol

V nasledujúcej kapitole si priblížime základy jazyka `GLSL`. Preberieme si jeho základné dátové typy, ako aj prácu fragment a vertex procesorov. Kapitola číslo 3 bude venovaná knižnici `OpenSceneGraph`. Priblížime si jej históriu a jej hlavné črty.

Knižnici `Qt` sa budeme venovať v kapitole 4. K nej si povieme niečo o moduloch, z ktorích sa skladá. Ďalej si popíšeme si jej význačné vlastnosti.

Teóriu jednotlivých implementovaných techník pokročilého renderingu si priblížime v kapitole 5. Praktické využitie získaných poznatkov popisuje kapitola 6.

[2]Graphical User Interface – grafické užívateľské rozhranie.
[3]Toolkit – v informatike je to sada nástrojov na vývoj aplikácií.
[4]Tutoriál – návod, poučenie, ako niečo robiť alebo používať, inštruktáž.

Kapitola 2

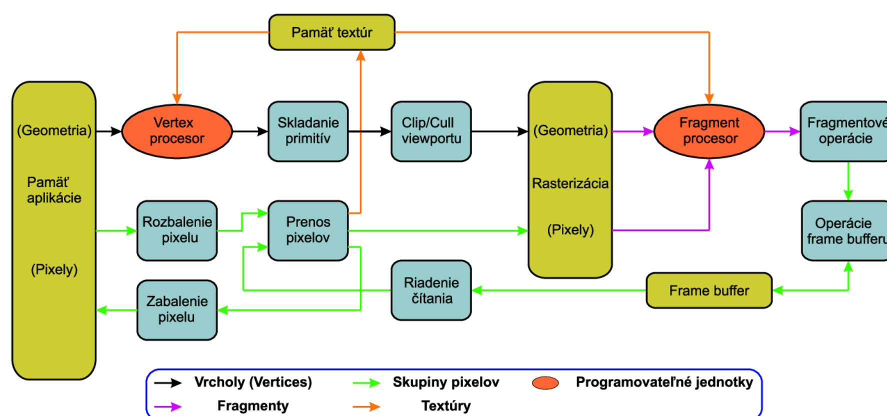
OpenGL Shading Language

2.1 Úvod

Jazyk `OpenGL Shading Language` (*GLSL*) je vysokoúrovňový jazyk založený na jazyku `ANSI C`, ktorého štandard je rozšírený o nové typy, ako sú *vektory* a *matice*. Tie umožňujú výstižnejšie opísať prvky vyskytujúce sa v 3D počítačovej grafike. Avšak niektoré konfliktné vlastnosti museli byť odstránené, aby sa zachovala výkonnosť a jednoduchosť implementácie.

Taktiež boli prevzaté niektoré vlastnosti jazyka `ANSI/ISO C++`, ako napríklad preťažovanie funkcii založené na typoch argumentov alebo možnosť deklarovať premenné tam, kde sú po prvýkrát potrebné, namiesto deklarovania na začiatku bloku. `GLSL` bol vyvinutý korporáciou OpenGL ARB (*OpenGL Architecture Review Board*). Členmi tejto korporácie sú napríklad 3Dlabs, Apple, ATI, Dell, IBM, Intel, NVIDIA, SGI a Sun Microsystems.

Hlavným zámerom vzniku jazyka `GLSL` bolo dať možnosť vývojárom pristupovať k jednotlivým častiam grafického komunikačného kanálu (*graphics pipeline*) bez nutnosti poznať hardwarovo špecifický asembler. Takto môžu vývojári pristupovať k trom rozdielnym procesorom, ako je *fragment procesor*, *vertex procesor* a *geometry procesor* za pomoci jedného programovacieho jazyka. Jedinou povinnosťou vývojárov je uvedomiť si drobné rozdiely pri práci s jednotlivými procesormi.



Obrázok 2.1: Komunikačný kanál OpenGL

Jazyk `GLSL` je súčasťou štandardu `OpenGL` od verzie 2.0.[11]

2.2 Základy jazyka GLSL

Ako bolo spomenuté v úvode, jazyk `GLSL` je založený na jazyku `ANSI C` s niektorými prvkami objektovo orientovaného `ANSI/ISO C++`. Z toho vychádza aj skladba základných dátových typov.

2.2.1 Dátové typy

Tu prebral jazyk `GLSL` z jazyka `ANSI C` typy ako *int*, *float* a *bool*. Je možné využiť aj *štruktúry* a *polia* s rovnakou syntaxou ako v `ANSI C`, s tým rozdielom, že pole **nemôže byť** inicializovane pri deklarácii. Prístup k prvkom štruktúry a pola je rovnaký ako v jazyku `ANSI C`.

`GLSL` pridal aj svoje vlastné dátové typy, ktoré pomáhajú abstrahovať základné prvky počítačovej grafiky. Medzi tieto prvky patria *vektory* (*vecX*/*ivecX*/*bvecX*), *matice* (*matX*) a *sampler*(*samplerXD*/*samplerCube*/*samplerXDShadow*), ktorý slúži na prácu s textúrami[1]. Nasledujúca tabuľka obsahuje prehľad dátových typov GLSL.

int	celé čísla
float	reálne čísla
bool	pravdivostné hodnoty
vec2, vec3, vec4	2, 3 a 4 rozmerný vektor s prvkami typu float
ivec2, ivec3, ivec4	2, 3 a 4 rozmerný vektor s prvkami typu int
bvec2, bvec3, bvec4	2, 3 a 4 rozmerný vektor s prvkami typu bool
sampler1D, sampler2D, sampler3D	prístup k textúram, reprezentuje ich ID
samplerCube	reprezentuje textúrovú kocku
sampler1DShadow, sampler2DShadow	reprezentuje textúru s tieňovou mapou

Tabuľka 2.1: Dátové typy v *GLSL*

[1]pozn. X značí číslo v intervale <1,4> podľa veľkosti daného dátového typu

Dátový typ *vektor* vznikol najmä z potreby ukladať vektorové informácie, ktoré sa v počítačovej grafike často vyskytujú. Medzi ne patrí napríklad hodnota farby alebo pozícia svetla. Pristupovať k prvkom vektoru je možné troma spôsobmi:

- Klasicky ako k prvkom pola v `ANSI C` a to pomocou indexov (*vec[i]*)
- Pomocou názvu komponentov, ktorý berie v úvahu použitie[2] vektora. Napríklad *Color.r* alebo *EyePos.z*. Prípustné sú nasledujúce kombinácie:

  x, y, z, w – vektor reprezentuje pozíciu alebo smer

  r, g, b, a – vektor reprezentuje farbu

  s, t, p, q – vektor reprezentuje textúrové koordináty

- Pomocou tzv. *Swizzling*, kedy sa pristupuje k viacerým prvkom vektoru naraz, ktoré musia byť rovnakého typu. Nie je možné sprístupniť napríklad *vektor.rgx*.

Matice ako dátový typ v `GLSL` sú vždy zostavené z hodnôt typu *float*.
Číslo pri type (*mat[2-4]*) predstavuje počet stĺpcov a riadkov v matici, hovoríme teda o *n–rozmernej matici*. K hodnotám v matici sa pristupuje tak isto ako k hodnotám v dvojrozmernom poly v jazyku `ANSI C`. Napríklad hodnotu v prvom riadku a druhom stĺpci získame zápisom *mat3[1][2]*.

Textúrový dátový typ *sampler* slúži k sprístupneniu textúr. Tento dátový typ je potrebný preto, lebo shader napísaný v `GLSL` nemôže načítať textúru priamo, ale môže len prijať ID textúry od aplikácie, ktorá ju nahrá do pamäte textúr.

2.2.2 Typové kvalifikátory

Deklarujú sa pred dátovým typom premennej a určujú pôvod a možnosti použitia premennej. Lokálne premenné nemajú žiadny kvalifikátor, alebo majú kvalifikátor *const*. Globálne premenné nesmú byť bez kvalifikátora. Prehľad typových kvalifikátorov je v nasledujúcej tabuľke.

const	Konštantná premenná.
uniform	Premenná, ktorá nemení svoju hodnotu počas doby spracovania primitívy. Spája aplikáciu, OpenGL a shader.
varying	Premenná, ktorá spája vertex a fragment shader.
attribute	Premenná, ktorá spája OpenGL a vertex shader.

Tabuľka 2.2: Typové kvalifikátory v *GLSL*

2.2.3 Vstavané funkcie

Vstavaných funkcií má jazyk `GLSL` veľa, preto uvediem len odkaz na pdf dokument dostupný online, ktorý obsahuje výpis všetkych podporovaných vstavaných funkcií. Daný dokument je k stiahnutiu na [2].

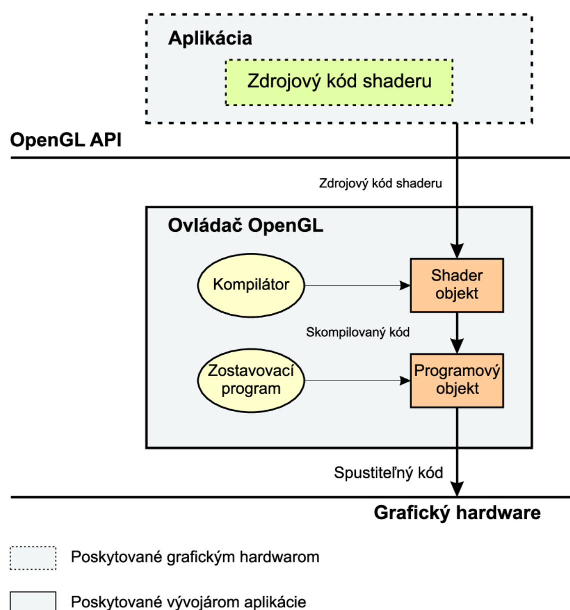
[2]Len z pohľadu užívateľa, pretože GLSL rozlišuje len dátový typ vektoru a nie jeho použitie.

2.2.4 Fragment a vertex shader

Ako bolo uvedené vyššie, *fragment* a *vertex* shader sú vlastne programy bežiace na *fragment* a *vertex procesore*. K spoločným znakom *fragment* a *vertex* shaderov patrí to, že každý shader **musí mať** hlavnú funkciu **main()**, ktorá má návratový dátový typ **void** a musí vykonávať svoju hlavnú úlohu. Touto úlohou je pri:

- **Vertex shadery**
  - *transformácia polohy bodu vzhľadom ku kamere*
- **Fragment shadery**
  - *nastavenie farby, alebo textúry fragmentu*

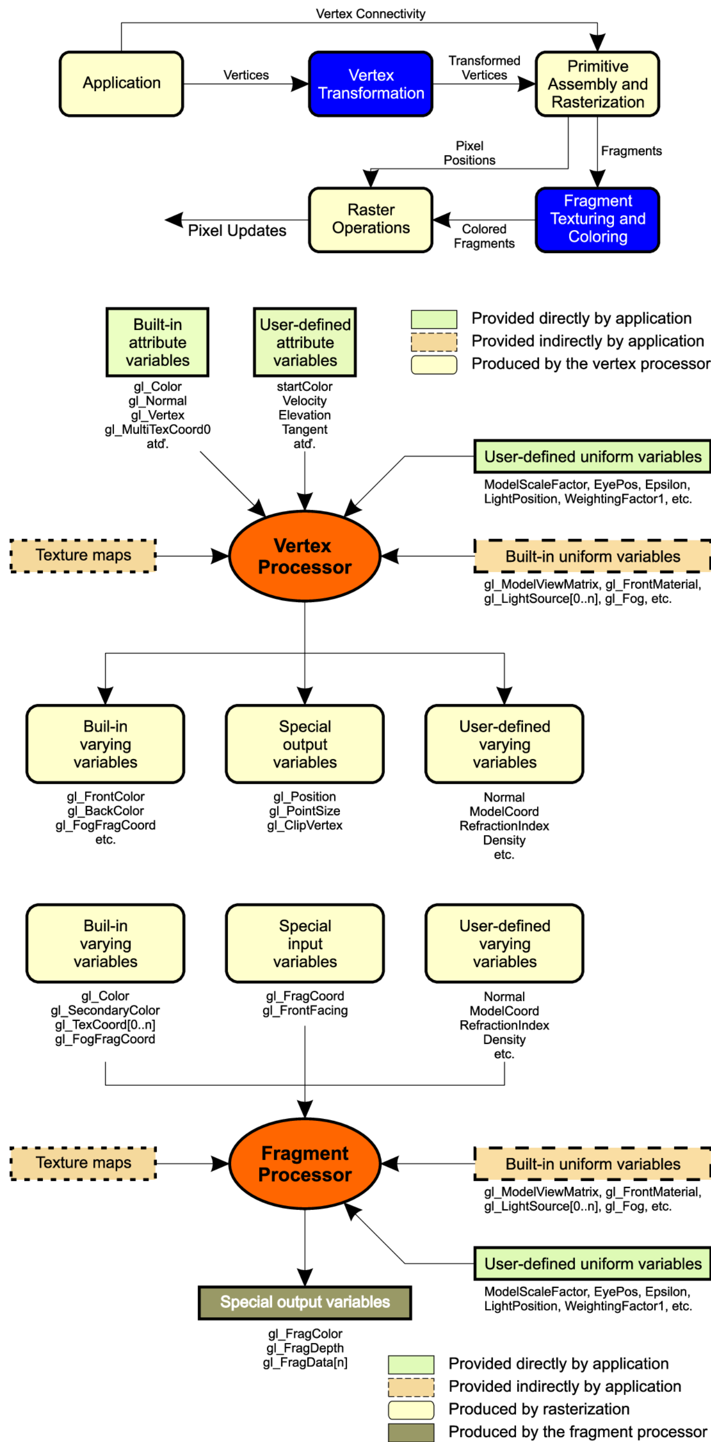
Tieto hodnoty sa zapisujú pri *vertex* shadery do systémovej premennej *gl_Position* a pri *fragment* shadery do systémovej premennej *gl_FragColor*. Je potrebné spomenúť, že existujú aj iné programovacie jazyky na písanie fragment a vertex shaderov. Medzi ne patrí napríklad jazyk `HLSL` od DirextX, `Cg` od NVIDIA alebo `ARB low-level assembly language`, ktorý vytvorila organizácia `OpenGL` a `GLSL` sa prekladá do neho a následne vykonáva. Niečo podobné ako preklad `ANSI C` do assembleru a následný výkon na cieľovej platforme.



Obrázok 2.2: Vykonávanie shader programov v OpenGL

2.2.5 Ďalšie rysy jazyka GLSL

Keďže tematika jazyka `GLSL` je veľmi rozsiahla uvediem už len obrázok týkajúci sa vertex a fragment procesorov. Na ňom je vidieť ukážku vstupov týchto procesorov a ich výstupov ako aj komunikáciu medzi nimi. Všetky potrebné informácie je možné nájsť v manuály jazyka `GLSL`, ktorý je voľne stiahnuteľný na [1].



Obrázok 2.3: Demonštrácia vstupov a výstupov vertex a fragment procesoru

Kapitola 3

OpenSceneGraph

3.1 Úvod

`OpenSceneGraph` (ďalej už len *OSG*) je voľne dostupná, multiplatformová sada nástrojov, ktorá je určená na rýchlu tvorbu grafických aplikácií, ako sú hry, animácie alebo simulátory. Sada nástrojov `OSG` je založená na koncepte `Scene Graph` a poskytujúca framework k `OpenGL`.

Framework je prostredie, v ktorom je organizovaná a napísaná ďalšia aplikácia. Je napísaný v tom istom programovacom jazyku ako aplikácia. Je to súbor knižníc a kódu usporiadaných tak, aby pokrývali čo najviac funkčných požiadaviek spoločných pre rôzne aplikácie. Framework má za úlohu ušetriť programátorovi čas tak, aby sa pri vývoji venoval len špecifickým požiadavkám pre aplikáciu, ktorú práve vyvíja, a ktorú nejde zovšeobecniť.

Tým pridáva `OSG` k `OpenGL` výhody objektovo orientovaného programovania, ako je vysoká úroveň abstrakcie alebo zapúzdrenosť objektu. To značne urýchľuje vývoj aplikácií a mieru chýb spôsobených napríklad volaním nízkoúrovňových funkcií v nesprávnom poradí. Koncept `Scene Graph` sa často využíva v aplikáciách pracujúcich s vektorovou grafikou.

3.2 História

Za začiatok písania histórie `OSG` môžeme považovať rok 1998, kedy sa **Don Burn** rozhodol vytvoriť simulátor závesného klzáku. Tu využil *Scene Graph Performer* od spoločnosti `Silicon Graphics, Inc.` pre ktorú pracoval, a ktorý bežal na operačnom systéme IRIX. V roku 1999 sa do projektu zapojil jeho kamarát a spolupracovník **Robert Osfield**, ktorý mu pomáhal s vývojom simulátora a taktiež s uspôsobením `Scene Graphu` na platformu Windows.

Robert Osfield sa snažil presvedčiť Dona Burna, aby sprístupnil zdrojové kódy k `Scene Graphu` a vytvoril tak opensource[1] projekt. To sa mu podarilo v septembri 1999, kedy sa sprístupnili zdrojové kódy a vznikla internetová stránka *www.openscenegraph.org*. Projekt `Scene Graph` prebral Robert Osfield a Don Burn sa zameral na vývoj simulátoru závesného klzáka. Cez jeseň a zimu toho roku Robert Osfield prepracoval `Scene Graph` za pomoci jazyka `ANSI/ISO C++` a jeho návrhových vzorov, aby ho prispôsobil novým požiadavkám a využil nové metodológie návrhu. Od roku 1999 sa `OSG` len rozrastá, o čom svedčí aj vznik

[1] Open source – je vo všeobecnosti akákoľvek informácia dostupná verejnosti, za podmienky, že možnosť jej slobodného šírenia zostane zachovaná.

novej spoločnosti Roberta Osfielda, `Andes Computer Engineering`. Tá zabezpečuje vývoj a podporu projektu `OpenSceneGraph`.[5]

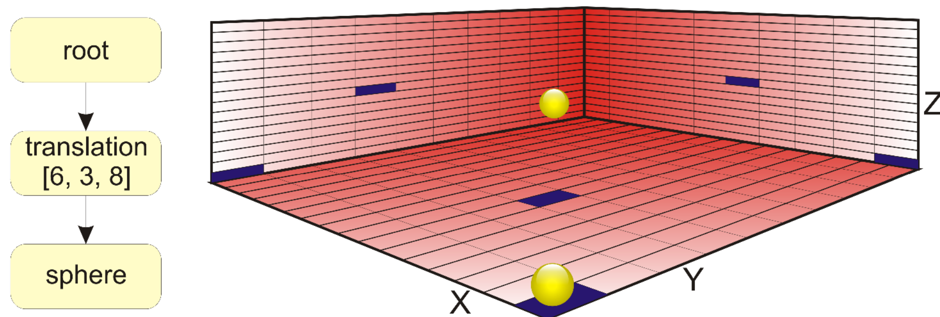
3.3 Základy OpenSceneGraph

3.3.1 Scene graph

`Scene Graph` predstavuje hierarchickú stromovú štruktúru, organizovanú do uzlov a hrán, ktoré dané uzly spájajú. To umožňuje ľahšie vytváranie výslednej scény, manipuláciu s objektmi, transformáciu objektov a v neposlednom rade zvyšuje výkon pri renderovaní.

Keďže sa jedná o stromovú štruktúru, tak na vrchole stromu sa musí vyskytovať koreňový uzol (*root node*). Pod ním sa nachádzajú synovské uzly, nazývané aj skupinové uzly (*group nodes*). Skupinové preto, lebo môžu mať pod sebou ďalšie synovské uzly alebo listy. Týmto potomkom potom určujú transformáciu a renderovacie stavy. Koreňový a skupinové uzly môžu mať 0 až n synovských uzlov alebo listov. Listami tohto stromu sú geometrické modely.

Vykreslovanie prebieha tak, že sa najprv ako začiatok zvolí koreňový uzol. Potom sa postupne prechádza nižšie pomocou algoritmu *depth-first search*[8] a vytvárajú sa modely, ku ktorým sa pridávajú transformačné a renderovacie atribúty. Týmto prechodom sa zobrazí celý vykresľovaný svet. Celý proces je znázornený na nasledujúcom obrázku.[4]



Obrázok 3.1: Prechod `Scene Graphom`

3.3.2 Zloženie OpenSceneGraph

S vývojom `OpenSceneGraph` sa postupne rozrastal aj počet knižníc, ktoré túto sadu nástrojov tvoria. Dnes môžeme knižnice `OpenSceneGraph` rozdeliť do piatich skupí:

1. Knižnice jadra OSG
    - **osg, osgUtil, osgDB**
2. Knižnice určené k zobrazovaniu
    - **osgViewer, osgGA**
3. Knižnice Nodekits

- **osgParticle, osgShadow, osgWidget, osgCharacter, . . .**

4. Knižnice zásuvných modulov

   - **OSGMaxExp, OSGMaya, . . .**

5. Knižnice podporujúce iné jazyky ako `ANSI/ISO C++` (*Komunitné projekty*)

   - **PyOSG, osgDotNet, JavaOSG. . .**

3.3.3 Knižnica `osg`

Je to základná knižnica `OpenSceneGraphu`, ktorá poskytuje triedy slúžiace na vytvorenie `Scene Graphu`. Sem patrí trieda **osg::Node**, ktorá je základom všetkých tried uzlov. Táto trieda je optimalizovaná na podporu vykresľovania a manažment stavu. Koreňový uzol je vlastne uzol triedy **osg::Node** bez rodiča. Ďalšou triedou, ktorú poskytuje osg je trieda **osg::Group**. Táto trieda je určená na vytváranie uzlov, ktoré môžu mať potomkov. Tým sa dostávame k listom `Scene Graphu`, ktoré sú reprezentovaná triedou **osg::Geode**. Ako bolo spomenuté predtým, tak listy tvoria geometrické objekty. Tie sú reprezentované práve triedou **osg::Geode**. Geometrické vlastnosti, ako je napríklad zoznam vrcholov objektu nesie trieda **osg::Geometry**. Okrem iného nesie aj informácie o *farbe*, *normálach* na vrcholoch a *koordinátoch textúry*. V knižnici osg sa nachádza aj trieda **osg::StateSet**, ktorá sa využíva na zmenu atribútov reprezentujúcich stavy v `OpenGL`. Každý uzol v `Scene Graphe` obsahuje referenciu na túto triedu.

K `OpenGL` patria aj `shadere`, preto poskytuje knižnica **osg** aj triedu **osg::Uniform**, ktorá slúži ako most medzi *aplikáciou* a *vertex/fragment procesorom*. Po tomto moste sa dáta od aplikácie predávajú procesorom. Existujú aj triedy poskytujúce vektory vo formáte, akom ich chápe `GLSL`. Tie sú sprístupnené pomocou tried **osg::Vec2**, **osg::Vec3**. . . . Samozrejme sú zastúpená aj matice. Tie sa sprístupnia pomocou triedy **osg::Matrixf** alebo **osg::Matrixd**.

V neposlednom rade netreba zabudnúť na triedu **Referenced**. Tá sa používa na manažment pamäte, ktorý má zamedziť pamäťovým únikom (*memory leaks*). Táto bázová trieda obsahuje čítač, ktorý má v sebe uložený počet referencií na objekt. Tento počet referencií sa pravidelne kontroluje a keď je nulový, tak sa zavolá deštruktor daného objektu. Takýto prístup sa v `OpenSceneGraph` označuje ako *referenced-counted memory scheme*. Túto vlastnosť vynútime v objektoch, ktoré ju podporujú pomocou triedy **osg::ref_ptr<>**.

Keďže celkový počet tried, ktoré obsahuje knižnica osg je 198, tak popis každej jednej z nich by bol nezmyselný. Ak sa chce čitateľ dozvedieť, aké triedy poskytuje knižnica **osg**, tak zoznam všetkých je k nahliadnutiu na [7]. V kapitole o implementácií budú popísane všetky potrebné triedy, ktoré neboli spomenuté tu.

3.3.4 Knižnica `osgViewer`

Táto knižnica poskytuje vysokú úroveň funkčnosti prehliadača výslednej scény. Jej úlohou je prejsť celý `Scene Graph` a vyrenderovať podľa zadaných uzlov a listov výslednú scénu. Vďaka sade ovládačov, ktoré poskytuje, je možné vytvoriť si svoj vlastný prehliadač šitý na mieru. A nakoniec môže byť tento výsledný prehliadač uspôsobený rôznym okenným manažérom.

3.3.5 Nodekits

Medzi Nodekity sa radia knižnice, ktoré rozširujú funkčnosť štandardných knižníc v poskytovaných typoch uzlov. Nodekity tvoria knižnice:

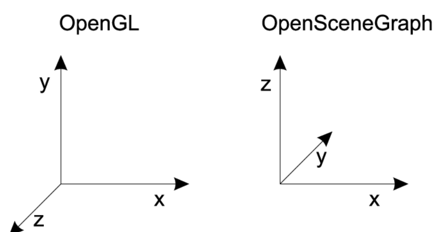
- **osgFX** – Špeciálne efekty ako bump mapping, cartooning, anisotropic lighting
- **osgParticle** – Časticové systémy
- **osgSim** – Vizuálna simulácia
- **osgManipulator** – Interaktívny manipulátor scény
- **osgShadow** – Tieňovanie
- **osgTerrain** – Rendering prostredia
- **osgText** – Renderovanie textu
- **osgWidget** – Podpora 2D (eventuálne 3D) GUI widgets
- **osgAnimation** – Animácie
- **osgVolume** – Objemové renderovanie

3.3.6 Zásuvné moduly

V základnej verzii `OpenSceneGraph` je 45 zásuvných modulov. Tieto moduly pridávajú do `OpenSceneGraph` podporu čítania a zapisovania štandardných formátov súborov ako aj súborov tretích strán. Okrem štandardných zásuvných modulov existujú aj komunitné zásuvné moduly.

3.3.7 Súradnicový systém

K rozdielom medzi `OpenSceneGraph` a `OpenGL` patrí aj orientácia súradnicového systému. Zatial čo OpenGL využíva súradnicový systém založený na zobrazení dát na obrazovke. Vychádza z toho, že obrazovka ako plocha má koordináty x a y, pričom vektor od obrazovky k oku pozorovateľa určuje hĺbku, teda koordinát z. OpenSceneGraph naproti tomu využíva súradnicový systém založený na fyzikálnom modely, kedy sa obrazovka neberie ako 2D plocha, ale ako keby tam ani nebola. Vtedy keď sa pozeráme na niečo z vrchu, tak sa pozeráme na plochu x/y a keď v perspektíve spredu, tak na teleso s výškou a šírkou na ploche x/z a hĺbkou v ploche y/z. Celú situáciu reprezentuje nasledujúci obrázok.



Obrázok 3.2: Rozdiel súradnicových systémov

Kapitola 4

Qt framework

4.1 Úvod

`Qt framework` poskytuje multiplatformové vývojové prostredie pre tvorbu aplikácií a užívateľských rozhraní. Vďaka vlastnému systému tvorby zostavovacích skriptov (*Makefile*), umožňuje vytváranie aplikácií na rôznych platformách bez nutnosti meniť zdrojový kód pri prechode na inú platformu. `Qt` využíva jazyk `ANSI/ISO C++`. Medzi hlavné výhody programovania v `Qt` patria:

- **Intuitívne názvy C++ tried** – Urýchľuje to proces učenia.
- **Prenositeľnosť medzi klasickými (*desktop*) a vstavanými systémami** – V dnešnej dobe je čoraz väčší dopyt po aplikáciách bežiacich jak na desktopoch, tak aj na mobilných zariadeniach. Tu sa uplatní koncept "píš raz, prelož kdekoľvek", ktorým sa `Qt` riadi.
- **Integrované vývojové nástroje s multiplatformovým rozhraním** – už v základe dostane záujemca všetky potrebné nástroje, aby mohol začať hneď tvoriť aplikáciu. Užívateľ je oslobodený od únavného hľadania potrebných nástrojov a ich zosúladením medzi sebou.
- **Vysoký výkon počas behu a nízke požiadavky na zdroje v stavaných systémoch**

V súčasnej dobe je Qt dostupné pre nasledujúce platformy:

Obrázok 4.1: Platformy

Avšak komunita vyvíja aj `Qt` pre platformy ako *Qt-iPhone*, *Qt for webOS* a iné. `Qt framework` podporuje aj programovanie v iných jazykoch ako je `ANSI/ISO C++`. Tie sú dostupné vďaka tzv. *language bindings*. Medzi najznámejšie patrí podpora pre `Javu`, `C#`, `.Net`, `Lisp`, `Python` a ďalšie. Kompletný zoznam spolu s podporou jednotlivých knižníc `Qt` v daných jazykoch je možné nájsť na [10].

4.2 Základy Qt

4.2.1 Qt moduly

`Qt framework` sa skladá z modulov, ktoré môžeme rozdeliť do štyroch kategórií:

1. Moduly na všeobecný vývoj softwaru
2. Moduly na prácu s Qt nástrojmi
3. Moduly pre vývojárov na platforme Windows
4. Moduly pre vývojárov na platforme Unix

Teraz si priblížime moduly na všeobecný vývoj softwaru, lebo s tými sa stretne najčastejšie. Do tejto skupiny patria dva základné moduly, a to *QtCore* a *QtGui*. Tieto dva moduly sú predvolenými modulmi pri projektoch.

QtCore je najzákladnejší modul, ktorý poskytuje základné, negrafické triedy využívané ostatnými modulmi. Na tomto module závisia všetky ostatné moduly. Medzi triedy, ktoré poskytuje patria napríklad QDebug, QString, QFile,... .

QtGui obsahuje triedy na tvorbu grafických prvkov užívateľského rozhrania. Medzi najpoužívanejšie triedy patria napríklad QMainWindow, QPushButton, QLineEdit, QBoxLayout, QMenubar, QStatusBar,... . Počet všetkých tried v tomto module je 406, preto budú v kapitole venujúcej sa implementácií stručne popísané len tie, ktoré boli použité.

Ďalším modulom je modul **QtOpenGL**, ktorý poskytuje triedy uľahčujúce použitie `OpenGL` v aplikácií. Medzi triedami, ktoré sú poskytované je aj trieda QGLWidget, ktorá slúži na renderovanie `OpenGL` grafiky. Táto trieda je využitá aj v demonštračnej aplikácií, keďže `OpenSceneGraph` je založené na `OpenGL`. Nachádza sa tu aj trieda QGLShader, ktorá má schopnosť skompilovať vertex a fragment shadere. V demonštračnej aplikácií boli využité len tieto tri moduly.

Zoznam všetkých modulov a tried nimi poskytovaných sa dá nájsť na [3]. Treba spomenúť, že dokumentácia `Qt frameworku` patrí k najlepším s akými som sa doteraz stretol. Všetky triedy a všetky metódy sú podrobne popísané a všade kde je to možné sú priložené aj demonštračné príklady.

4.2.2 Projektové súbory

Projektové súbory obsahujú všetky potrebné informácie, ktoré potrebuje aplikácia *qmake* na to, aby vygenerovala príslušný *Makefile*, alebo projektový súbor pre `Microsoft Visual Studio`.

Zdroje potrebné pre preklad sú deklarované jednoduchou postupnosťou deklarácií. Avšak nástroj *qmake* podporuje aj jednoduché programové konštrukcie, takže je možné vytvoriť jednoduché programy, ktoré zariadia bezproblémový preklad na rôznych platformách. Vytvorenie *Makefilu* je riadený pomocou premenných.

qmake definuje aj makrá ako `win32`, alebo `unix`. Pomocou týchto makier a podmienok typu `if`...`else`...je možné nastaviť preklad pre rôzne platformy.

Preklad prebieha zadaním príkazov:

1. **qmake -project** – Vytvorí nový projektový súbor
2. **qmake** – Vytvorí buď Makefile, alebo projektový súbor Visual Studia.

3. **make** – Spustí preklad

V nasledujúcich obrázkoch je výpis niektorých premenných a aj hodnoty ktoré môžu nadobúdať.

Premenná	Obsah
CONFIG	Hlavné konfiguračné nastavenia projektu.
DESTDIR	Adresár, do ktorého bude uložený spustiteľný súbor.
FORMS	Zoznam UI súborov vytvorených v Qt Designeri.
HEADERS	Zoznam hlavičkových súborov.
QT	Qt špecifické konfiguračné nastavenia.
RESOURCES	Zoznam zdrojov (.rc) súborov, ktoré budú zahrnuté do finálneho projektu.
SOURCES	Zoznam zdrojových súborov.
TEMPLATE	Šablóna použitá pre projekt.

Obrázok 4.2: Prehľad premenných

Šablóna	Popis výstupu z qmake
app(predvol.)	Vytvorí Makefile na zostavenie aplikácie.
lib	Vytvorí Makefile na zostavenie knižnice.
subdirs	Vytvorí Makefile, ktorý obsahuje pravidlá zo subdirs.
vcapp	Vytvorí projektový súbor pre MS Visual Studio(exe).
vclib	Vytvorí projektový súbor pre MS Visual Studio(lib).

Obrázok 4.3: Hodnoty pre premennú TEMPLATE

Nastavenie	Funkcia
core(predvol.)	QtCore modul.
gui(predvol.)	QtGui modul.
network	QtNetwork modul.
opengl	QtOpenGL modul.
sql	QtSql modul.
xml	QtXml modul.
xmlPatterns	QtXmlPatterns modul.
qt3support	Qt3Support modul

Obrázok 4.4: Hodnoty pre premennú QT

Niektoré konštrukcie v projektovom súbore budú predvedené a vysvetlené v časti zaoberajúcej sa implementáciou. Pre kompletný prehľad premenných a ich možných hodnôt odporúčam nasledujúci odkaz[6].

4.2.3 Signály a sloty

Signály a sloty sa v `Qt` používajú na komunikáciu medzi objektmi. Systém signálov a slotov je typickou vlastnosťou `Qt` a zároveň veľkou odlišnosťou od ostatných toolkitov. Pri programovaní GUI aplikácií často chceme, aby sa zmena v nejakom objekte prejavila

zavolaním metódy nejakého iného objektu, poprípade metódy daného objektu, v ktorom nastala zmena. Na toto sa práve používa systém signálov a slotov. V starších toolkitoch sa používal pre komunikáciu tohto typu systém využívajúci tzv. *callbacks*.

Callback je vlastne pointer na funkciu, ktorá sa má vykonať, keď sa niečo zmení v objekte. V praxi to vyzerá tak, že vykonávanej funkcií predáme ako parameter callback, a keď sa zmení stav objektu, tak vykonávaná funkcia predá callbacku parametre a zavolá ho. Tento prístup má dva problémy:

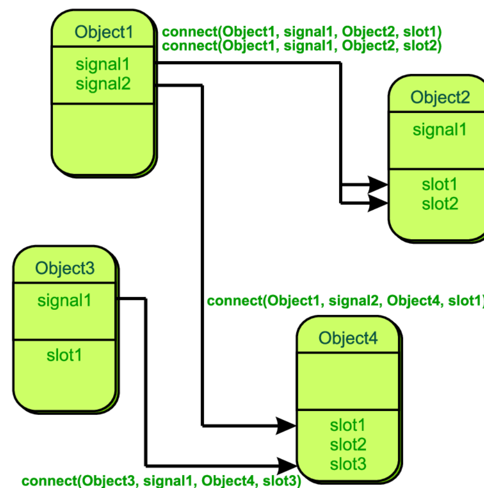
1. Nie je bezpečný typovo. Nikdy si nemôžeme byť istý, že vykonávaná funkcia zavolá callback so správnymi argumentmi.

2. Callback je pevne naviazaný na vykonávanú funkciu, pretože tá musí presne vedieť, ktorý callback má zavolať.

Na rozdiel od systému callbackov, je systém signálov a slotov typovo bezpečný. Túto vlastnosť zaručuje to, že signál aj prijímajúci slot musia mať rovnakú signatúru, resp. slot môže mať kratšiu signatúru, lebo môže ignorovať niektoré parametre signálu. Vďaka tejto vlastnosti môže aj kompilátor v dobe prekladu skontrolovať, či signatúry signálov a im priradených slotov súhlasia a v prípade potreby informovať užívateľa o chybe.

Vnútorný systém `Qt` zaručí to, že ak pripojíme signál k slotu, tak ten bude volaný *vždy so správnymi parametrami* a v *správny čas*. Všetky triedy, ktoré dedia vlastnosti z *QObject* alebo z jednej jeho podtriedy (*QWidget*) môžu obsahovať signály a sloty. Signál je emitovaný objektom vtedy, keď daný objekt zmení svoj stav tak, že je táto zmena zaujímavá pre ostatné objekty.

Na záver len toľko, že signál môže byť pripojený k viacerým slotom a naopak, jeden slot môže obsluhovať viacero signálov. Je tu aj možnosť napojiť signál na signál. Vtedy sa emituje druhý signál hneď po emitovaní prvého.



Obrázok 4.5: Systém signálov a slotov

4.2.4 Vývojové nástroje

Medzi vývojové nástroje `Qt frameworku` radíme nasledujúce:

- `Qt Creator`
- `Qt Assistant`
- `Qt Designer`
- `Qt Demo`

`Qt Creator` je multiplatformové vývojové prostredie určené na vývoj `Qt` aplikácií, ktoré je súčasťou `Qt SDK`. Obsahuje vizuálny debbuger a integrované prostredie na tvorbu užívateľského rozhrania. Editor kódu sa vyznačuje zvýrazňovaním syntaxe a automatickým doplňovaním príkazov. `Qt Creator` je akýmsi centralizovaným prostredím, kde sa dá písať kód aplikácie, debbugovať aplikácia, vytvárať vzhľad aplikácie alebo prezerať dokumentácia ku `Qt frameworku`.

`Qt Assistant` je aplikácia, ktorá zobrazuje kompletnú dokumentáciu `Qt frameworku`. Môže byť spustený priamo, alebo ako súčasť `Qt Creatoru`.

`Qt Designer` slúži na vytváranie grafického užívateľského rozhrania pomocou súčastí poskytovaných `Qt frameworkom`. V aplikácií má užívateľ možnosť vizuálne uspôsobiť prvky rozhrania pomocou tzv. *layouts*. Každému prvku potom môže priradiť jednoznačný názov (*Object name*). Prepájanie slotov a signálov sa deje pomocou jednoduchých a intuitívnych pravidiel, pričom sa pri každom novom pripojení slotu alebo signálu vygeneruje zdrojový kód metódy, a ten si môže užívateľ doupraviť. Alebo je možné priamo pripojiť už existujúci kód. Tvorba rozhrania pomocou tohto nástroja je naozaj rýchla a po čase veľmi efektívna. Nevýhodou je však niekedy dosť neprehľadný vygenerovaný výsledný zdrojový kód, čo môže spôsobiť problémy pri prepájaní s inými modulmi programu.

`Qt Demo` je aplikácia, ktorá je prístupná po skompilovaní príkladov z `Qt SDK`. Je určená na rýchle prezeranie príkladov, pričom obsahuje aj odkaz na dokumentáciu k nim. Tá sa spustí v aplikácií `Qt Assistant`.

Kapitola 5

Teória vybraných techník

5.1 Lambertov osvetľovací model

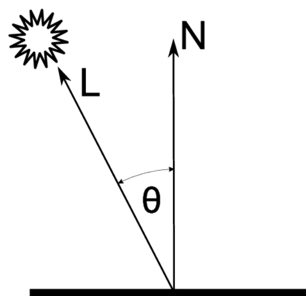
Je to empirický osvetľovací model, ktorý počíta len s difúznym odrazom.
Podľa Lambertovho pravidla:

> *Intenzita osvetlenia na difúznom povrchu je priamo úmerná kosínusu uhla medzi normálovým vektorom povrchu na ktorý dopadá svetelný lúč a zdrojom osvetlenia.*

Teda intenzita tohto difúzneho odrazu závisí na uhle dopadu svetla na povrch. Výpočet intenzity má potom nasledujúci tvar:

$$I_D = I_L \cdot r_D \cdot \cos\theta$$
$$I_D = I_L \cdot r_D \cdot (\vec{N} \cdot \vec{L})$$

Tým pádom je zrejmé, že sa jedná o v podstate veľmi jednoduchý osvetľovací model. Dôvodom, prečo som ho do práce zahrnul, je že dokáže pekne ukázať vplyv difúznej zložky.

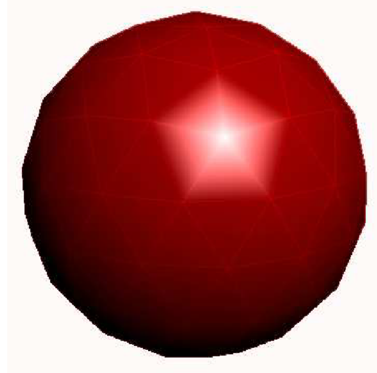


Obrázok 5.1: Lambertov osvetľovací model

5.2 Gouraudovo tieňovanie

Gouraudovo tieňovanie (angl. Gouraud shading) je metóda tieňovania, ktorú už v roku 1971 navrhol Henri Gouraud. Je založená na interpolácií svetelnej intenzity na hranách

plochy a jej interpoláciu pomocou scanline. Metóda vychádza z toho, že poznáme hodnoty normálových vektorov vo vrcholoch danej plochy. Preto je vhodná na tieňovanie telies, ktorých povrch je tvorený množinou mnohouholníkov. Táto metóda je *aproximatívna*, ale veľmi rýchla a používa sa v aplikáciách vyžadujúcich okamžitú (*real-time*) odozvu.



Obrázok 5.2: Gouraudovo tieňovanie

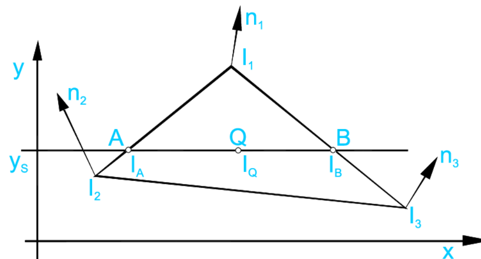
Pri tieňovaní potrebujeme mať informáciu o farbách vo všetkých vrcholoch daného mnohouholníka. Tie môžeme určiť podľa normály niektorým osvetľovacím modelom, napr. *Phongovým*. Potom môžeme vypočítať farebné odtiene (farbu) pre vnútorné body mnohouholníka.

Metóda na ich výpočet sa nazýva *bilineárna interpolácia*. Nasledujúci výpočet sa vzťahuje k obrázku 5.3:

$$
\begin{aligned}
I_A &= [I_1(y_S - y_2) + I_2(y_1 - y_S)]/(y_1 - y_2) \\
I_B &= [I_A(y_S - y_3) + I_3(y_1 - y_S)]/(y_1 - y_S) \\
I_Q &= [I_A(y_B - y_Q) + I_B(y_Q - y_A)]/(y_B - y_A)
\end{aligned}
$$

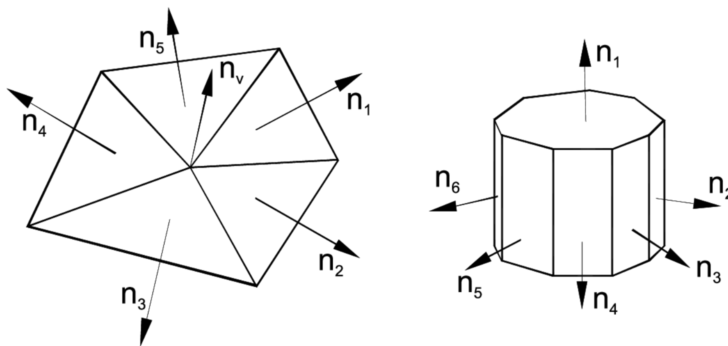
Iná varianta:

$$
\begin{aligned}
I_A &= I_1 + (I_2 - I_1) \cdot u \\
I_B &= I_A + (I_3 - I_1) \cdot w \\
I_Q &= I_A + (I_B - I_A) \cdot t \\
& u, w, t \in < 0, 1 >
\end{aligned}
$$



Obrázok 5.3: Bilineárna interpolácia

V prípade, že nemáme normálové vektory vo vrcholoch musíme ich určiť zo súčtu (alebo priemeru) normál *incidentných plôch*[1] tohto vrcholu.



Obrázok 5.4: Vektor vypočítame ako: $n_v = (n_1 + n_2 + n_3 + n_4 + n_5/5)$

V ďalšom kroku určíme intenzitu vrcholu niektorou z osvetľovacích metód. Nakoniec interpolujeme intenzity pre daný mnohouholník.

5.3 Phongov model

Túto metódu vyvinul **Bui Tuong Phong** počas svojho štúdia na univerzite v Utahu ako svoju doktorskú prácu v roku 1973. Phongov model v sebe zahrňuje model odrazu svetla (*Phong reflextion model*) od plôch a metódu tieňovania založené na určení výslednej farby pixelu pomocou interpolácie vrcholových normál pre každý pixel. Tu sa líši od *Gouraudovho tieňovania*, ktoré využíva vrcholové normály k interpolácií farby, pričom Phongov tieňovací model interpoluje tieto vrcholové normály k získaniu nových normál pre každý pixel v mnohouholníku. Takto je možné pri výpočte výslednej farby pixelu použiť jeho vlastnú normálu. Výsledkom je realisticky tieňovaný povrch, avšak za cenu veľkých výpočetných požiadaviek.

5.3.1 Phongov reflexný model

Phongov reflexný model je *empirický model* lokálneho osvetlenia. Popisuje spôsob ako plocha odráža svetlo. To ako sa svetlo odrazí závisí na nasledujúcich zložkách svetla:

- **Ambient** – Táto zložka reprezentuje okolité svetlo (*svetelný šum*), ktoré nie je vyžarované priamo nejakým zdrojom, ale je odrazené od okolitých objektov. Preto nemôžeme určiť jeho zdroj. Platí o ňom, že je všesmerové, teda nezávisí na pozícií iných svetelných zdrojov a objekt je ním osvetlený po celej svojej ploche. To zabraňuje tomu, aby odvrátené plochy objektu boli celkom čierne.

- **Diffuse** – Táto zložka je všesmerová a veľkosť intenzity odrazeného svetla je pre všetky smery rovnaká. Difúzne svetlo určuje farbu povrchu. Pomocou difúznej zložky sa vytvárajú matné povrchy.

- **Specular** – Zložka reprezentujúca odlesky lúčov svetla prichádzajúceho zo svetelného zdroja. Smer odrazu sa riadi *zákonom odrazu* a *Snellovým zákonom*. Len túto zložku

[1] Incidentnosť – vzájomná poloha dvoch útvarov so spoločnou časťou alebo prvkom.

má svetlo odrazené od dokonalého zrkadla. Odlesk môže mať inú farbu ako má povrch telesa.

Výpočet intenzity osvetlenia povrch vyjadruje nasledujúca rovnica:

$$I_V = k_a i_a + \sum_{m \in lights} (k_d(\vec{L}_m \cdot \vec{N}) i_d + k_s(\vec{R}_m \cdot \vec{V})^{\alpha} i_s)$$

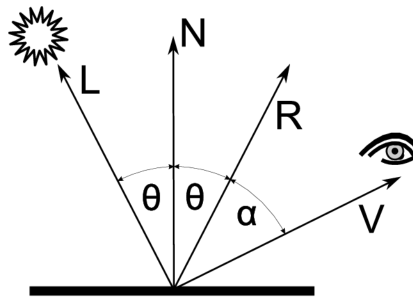
Kde sú vektory:

- $\vec{L}$ – Reprezentuje vektor od bodu na ploche smerom ku svetlu.
- $\vec{N}$ – Reprezentuje normálový vektor.
- $\vec{R}$ – Reprezentuje smer odrazu.
- $\vec{V}$ – Reprezentuje uhoľ pohľadu.

A kde pre každý materiál v scéne definujeme:

- k_a – Konštanta odrazu ambientnej zložky.
- k_d – Konštanta odrazu difúznej zložky.
- k_s – Konštanta odrazu specular zložky.
- α – Konštanta vyjadrujúca lesklosť materiálu.

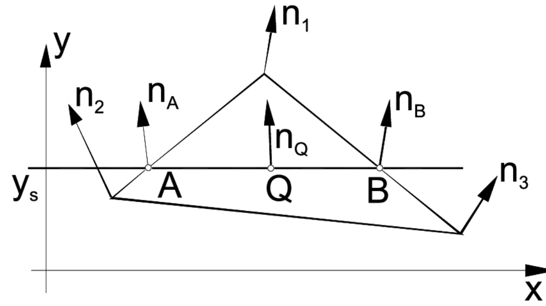
Pričom i_a, i_d a i_s sú zložky svetla(*ambien, diffuse, specular*). Celú situáciu demonštruje obrázok 5.5.



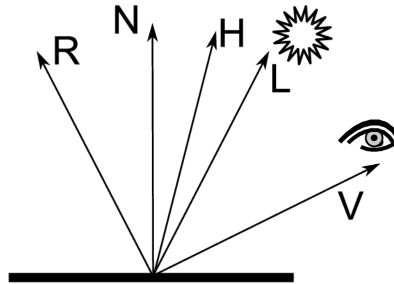
Obrázok 5.5: Phongov reflexný model

5.3.2 Phongova metóda tieňovania

Phongova metóda je oproti Gouraudovej metóde pomalšia, ale zato realistickejšia. Výsledná farba sa vypočíta za pomoci Phongovho reflexného modelu pre každý bod renderovanej plochy. Keďže Phongov reflexný model potrebuje k výpočtu farby bodu normálu v tomto bode, tak je potrebné ju dopočítať. To sa robí pomocou *lineárnej interpolácie normál z vrcholov*, pričom sa zohľadňuje zakrivenie plôch. To vede k tomu realistickému vzhľadu. Na nasledujúcom obrázku je znázornená lineárna interpolácia.



Obrázok 5.6: Lineárna interpolácia normálových vektorov



Obrázok 5.7: Blinn-Phong model

5.3.3 Blinn-Phongova metóda tieňovania

Blinn-Phongov model je modifikáciou (*aproximáciou*) Phongovho osvetľovacieho modelu. Používa sa v `OpenGL` ako predvolený osvetľovací model. Jeho výhodou je jeho nižšia náročnosť na výkon pri renderovaní. Pritom si zachováva pomerne svoju realistickosť. Rozdiel medzi Phongovým modelom a jeho modifikáciou (*Blinn-Phong*) spočíva v tom, že pri Phongovom modely musíme počítať uhol $R \cdot V$ medzi vektorom pohľadu (V) a lúčom zo zdroja (L), ktorý sa odrazí (R) od povrchu. Naproti tomu Blinn-Phongov model spočíta tzv. *halfway* vektor medzi vektorom pohľadu a vektorom k zdroju svetla. Ten sa spočíta nasledovne:

$$H = \frac{L + V}{|L + V|}$$

Tento uhol je síce menší ako uhol vypočítaný pomocou Phongovej metódy, ale po umocnení s exponentom α, sa približuje (*aproximuje*) k výsledku aký dosiahol Phongov model. Týmto získame modifikovanú rovnicu na výpočet intenzity osvetlenia povrchu:

$$I_V = k_a i_a + \sum_{m \in lights} (k_d(\vec{L}_m \cdot \vec{N}) i_d + k_s(\vec{N} \cdot \vec{H})^{\alpha} i_s)$$

Kapitola 6

Návrh a implementácia

6.1 Požiadavky

Požiadavkou bolo vytvoriť multiplatformovú aplikáciu, ktorá by demonštrovala základné techniky pokročilej real-time grafiky. Táto aplikácia má hlavne ukázať možnosti spojenia vývojového prostredia na tvorbu užívateľského rozhrania `Qt` a objektovo orientovanej knižnice na tvorbu real-time grafiky `OpenSceneGraph`.

Keďže `Qt toolkit` nemá vstavanú podporu pre renderovanie scény z `OpenSceneGraph`, ale iba podporu pre `OpenGL`, je nutné vytvoriť nové rozhranie medzi nimi. `OpenSceneGraph` je však objektovou nadstavbou nad `OpenGL` a sprístupňuje niektoré funkcie, ktoré nám vlastné prepojenie uľahčia.

Výsledná aplikácia bude z pohľadu `Qt` demonštrovať pokročilejšie vizuálne možnosti tohto prostredia. Tie však musia vytvárať vizuálne prívetivo vyzerajúcu aplikáciu. Tá bude poskytovať ovládacie prvky, ktoré budú manipulovať s objektmi v scéne renderovanej pomocou `OpenSceneGraph`. Taktiež bude aplikácia vytvorená tak, aby bol jej preklad na rôznych platformách bezchybný. To zabezpečí taktiež `Qt toolkit` a jeho systém projektových súborov spolu s utilitou *qmake*.

Zo strany `OpenSceneGraph` bude poskytnutá podpora renderovania pokročilej real-time grafiky.

Keďže požiadavkou je vytvoriť a demonštrovať prepojenie medzi `Qt` a `OpenSceneGraph`, tak využité techniky nebudú patriť k tým najzložitejším, ale budú zamerané skôr na svoj výkladový potenciál.

6.2 Vývojové prostredie

Ako bolo spomenuté predtým, výsledná aplikácia má byť multiplatformová. Z toho dôvodu je celá aplikácia napísaná v jazyku `ANSI/ISO C++`, ktorý je bez väčších problémov prenositeľný na rôzne platformy. Okrem iného poskytuje aj vysokú rýchlosť pri vykonávaní náročných výpočtov, kde renderovanie real-time grafiky určite patrí. `ANSI/ISO C++` je aj veľmi populárny medzi vývojármi, preto je pomerne ľahké vyhľadať riešenia vzniknutých problémov. Týmto by bola zodpovedaná otázka voľby implementačného jazyka a keďže oba toolkity majú podporu pre viacej programovacích jazykov, tak bolo nutné túto otázku zodpovedať.

`OpenSceneGraph` neposkytuje žiadne vývojové nástroje na prácu s touto knižnicou.

Existujú len pravidlá pre zvýrazňovanie syntaxe, ktoré sú určené pre Microsoft Visual Studio. Integrácia tejto knižnice do vývojového prostredia Microsoft Visual Studio nie je tiež jednou z najľahších.

Naproti tomu `Qt toolkit` prichádza kompletne s multiplatformovým vývojovým prostredím *Qt Creator*. Medzi jeho nesporné výhody patria najmä zvýrazňovanie syntaxe a doplňovanie kódu pri písaní na základe kontextu

K dispozícií je hneď od základu aj prehliadač on-line dokumentácie *Qt Assistant*, ktorý je v praxi neoceniteľnou pomôckou. Existuje aj prostredie *Qt Designer* na tvorbu užívateľského rozhrania pomocou vizuálnych prvkov. Toto prostredie však nevyužijeme.

6.3 Návrh aplikácie

Návrh aplikácie vychádza z požiadaviek, ktoré sme špecifikovali v predchádzajúcom texte. Hlavnou úlohou je vytvoriť aplikáciu demonštrujúcu spojenie `Qt` a `OpenSceneGraph` (ďalej už len OSG). Preto som sa rozhodol aplikáciu implementovať ako výučbový program, ktorý bude vysvetľovať a hneď aj názorne predvádzať osvetľovacie modely vyskytujúce sa v počítačovej grafike. Na začiatok je potrebné si vytvoriť hlavné okno.

V systéme `Qt` existuje trieda `QMainWindow`, ktorá je určená práve na tvorbu hlavného okna. Má vlastnú schému rozloženia prvkov v okne a dáva možnosť pridať vlastný panel nástrojov, lištu s menu, stavový riadok, alebo tzv. *plávajúce panely*. Ďalej musíme vyriešiť hlavnú otázku, a to je prepojenie `Qt` s `OSG`. `OSG` je založené na `OpenGL`, pre ktoré má `Qt` podporu v podobe `QGLWidget`. Problémom je ako prepojiť renderovacie okno `QGLWidget` s renderovacím oknom `OSG`. `OSG` poskytuje triedu `GraphicsWindowEmbedded`, ktorá umožňuje ako keby natiahnuť renderovacie okno `OSG` na `QGLWidget`. Takže využijeme túto možnosť.

Ďalšou časťou aplikácie bude prehliadač zdrojových súborov. Ten by mal byť schopný zvýrazniť syntax jazyka `ANSI/ISO C++`, ukázať čísla riadkov a zvýrazniť aktuálny riadok. K tomuto použijeme triedy ako:

- `QPlainTextEdit` – jedná sa o jednoduchý objekt zobrazujúci text.
- `QSyntaxHighlighter` – trieda poskytuje možnosť definovať pravidla zvýraznenia syntaxe
- `QPainter` – umožňuje „kresliť“ na objekty v systéme `Qt`.

Navrhovaným riešením je vytvoriť objekt `QPlainTextEdit`, keďže nepotrebujeme použiť tzv. *rich-text* a jemu definovať zvýraznenie syntaxe pomocou `QSyntaxHighlighter`. Čísla riadkov sa budú vykresľovať vedľa. Súčasťou aplikácie je aj jednoduchý návod (*tutoriál*) k daným technikám.

6.4 Implementácia

V tejto časti si teraz stručne popíšeme implementáciu výslednej aplikácie. K jej vývoju boli využité prostriedky spomenuté v podkapitole 6.2. Popis implementácie bude rozdelený na dve časti. Prvá sa bude zaoberať implementáciou v rámci `Qt` a druhá `OpenSceneGraph`.

6.4.1 Implementácia vrámci `Qt`

Tu spomeniem len významné a zaujímavé črty rozhrania v `Qt`, lebo popísať všetky implementované vlastnosti by bolo veľmi rozsiahle.

Základnou triedou, ktorá poskytuje vizuálne rozhranie je trieda je mnou definovaná trieda `Gui`. Tá dedí vlastnosti z triedy `QMainWindow`, ako je napríklad schéma rozloženia prvkov (*layout*). Ten mi umožňuje definovať plávajúce panely, ktoré sú jedným z pokročilejších vizuálnych prvkov Qt. Plávajúci panel sa v triede `Gui` definuje nasledovne:

```
optionsDock = new QDockWidget(tr("Options"), this);
options = new optionsWidget(this);
optionsDock->setFeatures(QDockWidget::DockWidgetMovable);
optionsDock->setAllowedAreas(Qt::LeftDockWidgetArea |
                             Qt::RightDockWidgetArea);
optionsDock->setWidget(options);
this->addDockWidget(Qt::LeftDockWidgetArea, optionsDock);
```

Najprv si vytvoríme objekt reprezentujúci samotný `QDockWidget`. Následne vytvoríme objekt, ktorý chceme umiestniť do `QDockWidget`. Nakoniec nastavíme vlastnosti novovzniknutému `QDockwidget` objektu, priradíme mu `QWidget`, ktorý chceme zobraziť a zaregistrujeme `QDockWidget` k nášmu oknu. Z vlastností som definoval len možnosť premiestnenia v rámci hlavného okna.

Keďže som si definoval triedu, ktorá dedí vlastnosti z QMainWindow, tak vytvoriť menu zobrazujúce sa hore v aplikácií je jednoduché.

```
fileMenu = menuBar()->addMenu(tr("&File"));
fileMenu->addAction(tr("&Exit"), this, \
        SLOT(close, QKeySequence(tr("Ctrl+E", "File|Exit")));
```

Definujeme si *fileMenu* ako *QMenu *fileMenu*. Potom ho priradíme hlavnému menu pod názvom „&File“. Ampersand určí, že k menu je možné pristúpiť klávesovou skratkou ALT + F. Nakoniec pridám položku *Exit*, ktorú pripojíme k uzatváraciemu slotu hlavnej aplikácie.

Ďalším rysom aplikácie je nastavenie štýlu jednotlivých častí `Qt` pomocou externého štýlového súboru. Ten má formát podobný *CSS*, čo robí štýlovanie aplikácie pomerne jednoduchým. Avšak je nutné presne vedieť, ktorý prvok vyžaduje akú formu popisu. Príklad popisu je nasledujúci:

```
QMenuBar::item { background: transparent;
                 color: deepskyblue;}
```

Štýl sa nastavuje pomocou príkazu *setStyleSheet(QString styleSheet)*. Tieto príklady predstavujú základ vizuálneho vzhľadu aplikácie.

Prvkom, ktorý je treba popísať je zvýrazňovač syntaxe pre prehliadač zdrojového kódu. Ten je implementovaný pomocou triedy Highlighter dediacej z triedy `QSyntaxHighlighter`. Nasledujúce úseky kódu reprezentujú len významné kroky:

```
QVector<HighlightingRule> highlightingRules
HighlightingRule rule;
QTextCharFormat keywordFormatOrange;
QStringList keywordPatternsOrange;
foreach (QString pattern, keywordPatternsOrange) {
        rule.pattern = QRegExp(pattern);
        rule.format = keywordPatternsOrange;
        highlightingRules.append(rule);
}
```

Úsek uvedeného kódu sa dá interpretovať ako:

1. Vytvor vektor pravidiel vyhľadávania.
2. Definuj pravidlo ako dvojicu regulárny výraz – formátovanie.
3. Pre každé pravidlo zo zoznamu pravidiel nastav príslušné formátovanie.

Premenná rule je vlastne štruktúra uchovávajúca regulárny výraz, pomocou ktorého s vyhľadáva a formátovanie nájdeného výrazu. Následne sa sada pravidiel aplikuje postupne na všetky bloky dokumentu.

Teraz sa dostávame k základu práce a tým je komunikácia medzi `Qt` a `OSG`. Tú v sebe zapúzdrujú triedy `AdapterWidget` a `ViewerQT`.

To pomyselné prepojenie vykonáva hlavne `AdapterWidget`, ktorý dedí z `QGLWidget`. Následne vytvorí objekt triedy `osgViewer::GraphicsWindowEmbedded`, ktorému nastaví geometriu na geometriu, ktorú dostal pridelenú ako QGLWidget. Takisto nastaví predanie riadenia po kliknutí a zabezpečí korektné predávanie signálov od myši, alebo klávesnice.

Trieda `ViewerQT` potom prevezme takto vytvorené renderovacie okno a nastaví jeho prekresľovanie pomocou časovača poskytovaného `Qt`. Nesmieme zabúdať, že ak vytvárame vlastnú geometriu okna, je potrebné definovať *afinnú* transformáciu súradníc x a y normalizovaného zariadenia na súradnice okna.

Ďalej je potrebné nastaviť *projekčnú maticu* a získať grafický kontext okna do ktorého vykresľujeme scénu. Nasledujúce kúsky kódu demonštrujú ako som dosiahol prepojenie `Qt` a `OSG`. Znovu budú uvedené len najhlavnejšie časti.

Triedy si nadefinujeme nasledovne:

```
class AdapterWidget : public QGLWidget
class ViewerQT : public osgViewer::Viewer, public AdapterWidget
```

Nastavíme geometriu okna `OSG` na geometriu, ktorú dostal `QGLWidget`. A nastavíme predanie riadenia po kliknutí myšou.

```
_gw = new \
       osgViewer::GraphicsWindowEmbedded(0,0,width(),height());
setFocusPolicy(Qt::ClickFocus);
```

Zabezpečíme aby sa korektne predalo riadenie z Qt do OSG.

```
_gw->getEventQueue()->keyPress((osgGA::GUIEventAdapter::KeySymbol)
                                    *(event->text().toAscii().data()));
switch(event->button()) {
  case(Qt::LeftButton): i = 1; break;
_gw->getEventQueue()->mouseButtonPress(event->x(),event->y(),i);
```

Toto všetko sa deje v `AdapterWidget`. Vo `ViewerQT` upravíme súradnice normalizované x, y na súradnice okna. Nastavíme projekčné maticu, predáme grafický kontext renderovacieho okna a nastavíme prekresľovanie po 10ms.

```
getCamera()->setViewport(\
                new osg::Viewport(0,0,   width(),height()));
getCamera()->setProjectionMatrixAsPerspective(30.0f, \
                              static_cast<double>(width()) \
      /static_cast<double>(height()), 1.0f, 10000.0f);
getCamera()->setGraphicsContext(getGraphicsWindow());

connect(&timer, SIGNAL(timeout()), this, SLOT(updateGL()));
timer.start(10);
```

Predchádzajúce informácie popisujú to najhlavnejšie, čo treba spraviť na prepojenie `OSG` a `Qt`.

6.4.2 Implementácia vrámci `OpenSceneGraph`

V nasledujúcej časti popíšem, ako boli realizované jednotlivé efekty pokročilého renderovania. Teoretické základy boli prebrané v kapitole 5. Nasledujúce riadky popisujú vytvorenie náhľadovej scény.

Ako už bolo predtým spomenuté, `OSG` renderuje scénu ako hierarchickú stromovú štruktúru, pričom začína v koreňovom uzle a postupuje nižšie. Preto je potrebné nadefinovať si koreňový uzol ako:

```
osg::ref_ptr <osg::Group> root;
root = new osg::Group;
```

Následne si vytvoríme uzol reprezentujúci objekt typu *Drawable*, a uzol na reprezentáciu osvetlenia.

```
osg::ref_ptr <osg::Geode> SphereGeode (new osg::Geode);
osg::ref_ptr <osg::LightSource> lightSource = new
    osg::LightSource;
```

Potom si vytvoríme samotný objekt, ktorým bude guľa (*Sphere*).

```
Sphere = new osg::Sphere(osg::Vec3(0.0f,0.0f,0.0f),5.0f);
SphereDrawable = new osg::ShapeDrawable(Sphere.get())
```

Keďže chceme, aby sa svetlo pohybovalo, tak využime možnosti uzla poskytujúceho transformáciu podľa transformačnej matice.

```
lightTransform = new osg::PositionAttitudeTransform();
```

Treba ešte vytvoriť vlastné osvetlenie a definovať mu jeho vlastnosti. Potom ho pripojiť k uzlu osvetlenia a ten zadať transformačnej matici, aby sme boli schopný vytvoriť pohybujúce sa svetlo. To dosiahneme pomocou ukážky nasledujúceho kódu.

```
myLight = new osg::Light;
myLight->setDiffuse(osg::Vec4(0.8f,0.8f,1.0f,1.0f));
lightSource->setLight(myLight.get());
lightTransform->addChild(lightSource.get());
```

Potom si zistíme stav vykresľovaného modelu a normalizujeme jeho normály. Nakoniec ho pripojíme k uzlu, ktorý ho bude reprezentovať.

```
sphereStateSet = SphereDrawable->getOrCreateStateSet();
sphereStateSet->setMode(GL_NORMALIZE,
                        osg::StateAttribute::ON);
SphereGeode->addDrawable(SphereDrawable.get());
```

Nasleduje vytvorenie a načítanie shaderov. Na vytvorenie *vertex* alebo *fragment* shaderu slúži trieda `osg::Shader`, pričom parametrom je typ použitého shaderu (*vertex*, *fragment*). Mimo iné poskytuje aj metódu na nahranie shaderu priamo zo súboru. Na načítanie shaderu pola využitá funkcia, ktorá je využitá v príkladoch pre `OSG`.

```
bool loadShaderSource(osg::Shader *shader,
                      const std::string &fileName)
```

Ďalšou triedou je trieda `osg::Program`, ktorá je abstrakciou nad *glProgram*. A teraz už k samotnému vytvoreniu a aplikovaniu shader programu. Najprv si nadefinujeme potrebné premenné a `osg::Program` objektu nastavíme názov.

```
ShadersPrograms = new osg::Program;
ShadersPrograms->setName("ShaderProg");

VertexShader = new osg::Shader(osg::Shader::VERTEX);
FragmentShader = new osg::Shader(osg::Shader::FRAGMENT);
```

Nasleduje nahranie vertex a shader programov pomocou vyššie spomenutej funkcie a pripojenie jednotlivých shaderov k `osg::Program` objektu. Nakoniec sa nastaví atribút koreňového uzla, ktorý zabezpečí, aby sa shader program aplikoval na celú scénu. Je však možné aplikovať shader program aj na jednotlivé časti scény. Potom už len pripojíme ku koreňovému uzlu transformačný uzol reprezentujúci naše osvetlenie a uzol reprezentujúci vykresľovaný objekt. Koreňový uzol potom zadáme do prehliadača scény, ktorý ho vyrenderuje.

```
loadShaderSource(VertexShader, "lambert.vert");
loadShaderSource(FragmentShader, "lambert.frag");
ShadersPrograms->addShader(VertexShader);
ShadersPrograms->addShader(FragmentShader);
rootStateSet->setAttributeAndModes(ShadersPrograms,
                                   osg::StateAttribute::ON);
root->addChild(SphereGeode);
root->addChild(lightTransform);
viewer->setSceneData(root);
```

Pri každom prekreslení scény sa nastaví nová poloha pre transformačný uzol.

```
lightTransform->setPosition(osg::Vec3(cos(this->count),
                            sin(this->count),  0.5) * 4);
```

Takto sme si vytvorili jednoduchý model a osvetlenie, aplikovali shader a vyrenderovali scénu, ktorá obsahuje animáciu.

6.4.3 Shader Programy

V tejto časti popíšem postupne všetky shader programy, ktoré boli implementované v rámci práce. Keďže teória zvolených techník bola prebraná v kapitole 5, uvediem už len priamo jednotlivé vertex a fragment programy.

Lambertov osvetľovací model

Je to asi najjednoduchšia implementovaná technika. *Lambertov model* je zameraný len na odraz difúznej zložky. Vo vertex shadery si uložíme vektor označujúci uhol pohľadu a normálový vektor. Keďže nemusíme transformovať vrcholy ručne, použijeme funkciu simulujúcu fixnú pipeline.

```
vec4 color = gl_FrontMaterial.diffuse;
vec4 lpos = gl_LightSource[0].position;
vec4 s = normalize(lpos - v);
vec3 light = s.xyz;
vec3 n = normalize(normal);

vec4 diffuse  = color * max(0.0, dot(n, s.xyz))
               * gl_LightSource[0].diffuse;
gl_FragColor = diffuse;
```

Celý výpočet sa uskutočňuje vo fragment shadery, preto dostávame vo výsledku *per-pixel* osvetlenie. V prvom rade si zistíme hodnotu difúznej zložky materiálu. To nám určí akú farbu odráža materiál. Následne už len spočítame potrebné vektory a pomocou *Lambertovho pravidla* vypočítame a zapíšeme fragmentu veľkosť intenzity difúznej zložky odrazeného lúča.

```
gl_FragColor = gl_Color;
```

Gouraudov tieňovací model

Táto technika sa využíva v mnohých aplikáciách ako náhrada podstatne náročnejšieho *Phongovho tieňovania*. Je to technika per-vertex osvetlenia, preto sa vo fragment procesore nevykonáva žiadny výpočet. Kód pre fragment procesor je nasledujúci:

```
gl_FragColor = gl_Color;
```

Jeho úlohou je simulovať fixnú pipeline. Celý výpočet prebieha vo vertex procesore. Najprv si zistíme normálu pre prichádzajúci vrchol, ktorú normalizujeme. Tu by som chcel poukázať na to, že je veľmi dôležité normalizovať vektory, ktoré si to vyžadujú, lebo potom môže dochádzať k vzniku nežiaducich fragmentov.

```
v3Normal = gl_NormalMatrix * gl_Normal;
v3Normal = normalize(v3Normal);
```

Následne môžeme pristúpiť k výpočtu uhla medzi pozorovateľom, vrcholom a zdrojom svetla.

```
vVertex = vec3(gl_ModelViewMatrix * gl_Vertex);
LightDir = gl_LightSource[0].position.xyz -vVertex;
LightDir = normalize(LightDir);
fAngle = max(0.0, dot(v3Normal, LightDir));
```

Dávame si pozor, aby uhol nebol menší ako 0 stupňov. Nakoniec spočítame žiarivosť vrcholu a hodnotu farby odrazeného lúča pomocou *Gouraudovho normálového výpočtu.*

```
fShininessFactor = pow(fAngle, gl_FrontMaterial.shininess);

gl_FrontColor = gl_LightSource[0].ambient *
                gl_FrontMaterial.ambient +
                gl_LightSource[0].diffuse *
                gl_FrontMaterial.diffuse * fAngle +
                gl_LightSource[0].specular *
                gl_FrontMaterial.specular * fShininessFactor;
```

Vrchol necháme prejsť bez zmeny jeho polohy.

```
gl_Position = ftransform();
```

Blinn-Phong model

Blinn-Phong je pre-pixel osvetlenie. Vo vertex shadery sa spočíta vektor k vrcholu, k svetlu a normálový vektor. Tieto vektory sa pošlú do fragment procesora.

```
P = vec3(gl_ModelViewMatrix * gl_Vertex);
L = normalize(gl_LightSource[0].position.xyz-P);
N = normalize(gl_NormalMatrix * gl_Normal);
```

Keďže sa jedná o per-pixel osvetlenie, tak celý výpočet prebieha práve vo fragment procesore. Tam si spočítame vektor k pozorovateľovi, ktorý normalizujeme. Potom spočítame *half-way* vektor podľa rovnice uvedenej v teoretickej časti.

```
P = vec3(gl_ModelViewMatrix * gl_Vertex);
L = normalize(gl_LightSource[0].position.xyz-P);
N = normalize(gl_NormalMatrix * gl_Normal);
```

A na záver spočítame jednotlivé časti zo vzorca umiestneného v teoretickej časti výsledné farbu fragmentu.

```
gl_FragColor = gl_FrontLightModelProduct.sceneColor
             + Iamb + Idiff + Ispec;
```

Phong model

V predchádzajúcej časti sme si Blinn-Phong model. Phongov model je rozdielny len v jednej veci. A to, že nepočíta tzv. *half-way vektor*, ale spočíta vektor odrazeného lúča od materiálu. Tento výpočet je však veľmi náročný, ale dosahuje reálnejší výsledok ako *Blinn-Phong*. `GLSL` má dokonca vstavanú funkciu na jeho výpočet, ktorú som využil aj vo fragment shadery. Zmenou teda oproti Blinn-Phong modelu je tento riadok:

```
vec3 R = normalize(-reflect(L,N));
```

Možný prepis funkcie reflect je nasledujúci:

```
R = 2 * ( N dot L) * N - L
```

Ten nám spočíta vektor odrazeného lúča. A zmení sa tým pádom aj použitá rovnica, ktorá je uvedená v teoretickej časti.

Implementované techniky patria k tým jednoduchším, ale vo výsledku sa na nich dá pekne ukázať spolupráca medzi `OSG` a `Qt`.

Kapitola 7

Záver

Výsledkom tejto práce je aplikácia, ktorej cieľom je ukázať prepojenie medzi knižnicami `OpenSceneGraph` a `Qt`. Aplikácia poskytuje užívateľovi informácie o najpoužívanejších osvetľovacích a tieňovacích modeloch. Zároveň mu dáva možnosť manuálne nastaviť niektoré vlastnosti osvetlenia a materiálu, pričom môže priamo pozorovať dopad týchto zmien.

Práca s knižnicou OpenSceneGraph je spočiatku zložitejšia, ale postupne ako si človek zvyká na systém práce s ňou, stáva sa jednoduchšou ako práca s tradičným `OpenGL`. Táto práca mi priniesla veľa nových poznatkov. Asi najväčší prínos bol z oblasti multiplatformových vývojových prostredí.

Multiplatformovosť je stále viac vyžadovanou vlastnosťou moderného software. Dávno sú preč časy, keď `Linux` malo nainštalovaný len pár odborníkov a bežný užívatelia o ňom nemali ani potuchy. Dnes je pomerne bežné sa stretnúť s užívateľmi, ktorý majú nainštalovanú nejakú z *user-friendly* distribúcií(*Ubuntu*, *Mint*).

Moja aplikácia vznikala pod systémom `Unix`, ale ak má dostupné potrebné knižnice, tak je bez problémov preložiteľná aj na iných platformách. Jediným problémom, s ktorým sa môže užívateľ stretnúť je, ako správne nainštalovať a skompilovať potrebné knižnice na inej platforme ako je Unix. Já osobne som aplikáciu spustil na systéme `Windows` asi po týždni snaženia. Pomohlo mi to uvedomiť si určité zákonitosti ohľadom multiplatformových vývojových prostredí.

Nepochybným prínosom bolo aj programovanie GUI v `Qt`. S toolkitom `Qt` som sa už predtým stretol v povinne voliteľnom predmete, ale až teraz som si prehĺbil svoje vedomosti o ňom.

Budúcnosť tejto práce by som videl v pridaní príkladov z oblasti pokročilého textúrovania alebo ďalších renderovacích techník. Mala by vzniknúť aplikácia vysvetľujúca jednotlivé techniky spolu s ich názornými ukážkami a možnosťou meniť ich parametre. Nepochybnou výhodou by bolo, ak by si užívateľ mohol sám nahrať shader programy a aj textúry. Aplikácií tohto typu na trhu moc nie je.

Pri vytváraní aplikácie ma nadchlo hlavne tvorenie užívateľského rozhrania pomocou knižnice `Qt`. V budúcnosti by som sa chcel zamerať skôr týmto smerom.

Literatúra

[1] KESSENICH, J., BALDWIN, D. and ROST, R.: The OpenGL Shading Language. In KESSENICH, J., editor. [online], 2010, ver. 1.1. rev.: 8. [cit. 11.05.2010].
URL <http://www.opengl.org/registry/doc/GLSLangSpec.4.00.8.clean.pdf>

[2] KHRONOS GROUP: OpenGL 4.0 & GLSL Quick Reference Guide. [online], 2010, rev.: 0210. [cit. 11.05.2010]. pp. 6-8.
URL <http://www.khronos.org/files/opengl4-quick-reference-card.pdf>

[3] WWW stránky: All Qt Modules. [online], [cit. 11.05.2010].
URL <http://doc.trolltech.com/4.6/modules.html>

[4] WWW stránky: An Introduction to OpenSceneGraph and osgART. [online], [cit. 11.05.2010].
URL <http://www.artoolworks.com/support/library/An_Introduction_to_OpenSceneGraph_and_osgART#Scene_Graphs>

[5] WWW stránky: OpenSceneGraph: History. [online], [cit. 11.05.2010].
URL <http://www.openscenegraph.org/projects/osg/wiki/Support/History>

[6] WWW stránky: qmake Project Files. [online], [cit. 11.05.2010].
URL <http://cartan.cas.suffolk.edu/qtdocs/qmake-project-files.html>

[7] WWW stránky: Reference Guides. [online], [cit. 11.05.2010].
URL <http://www.openscenegraph.org/projects/osg/wiki/Support/ReferenceGuides>

[8] WWW stránky: Wikipedia: Depth-first search. [online], [cit. 11.05.2010].
URL <http://en.wikipedia.org/wiki/Depth-first_search#Example>

[9] WWW stránky: Wikipedia: Graphics processing unit. [online], [cit. 11.05.2010].
URL <http://en.wikipedia.org/wiki/Graphics_processing_unit>

[10] WWW stránky: Wikipedia: Qt (framework). [online], [cit. 11.05.2010].
URL <http://en.wikipedia.org/wiki/Qt_(framework)>

[11] WWW stránky: OpenGL. [online], 2010, [cit. 11.05.2010].
URL <http://www.opengl.org/documentation/glsl/>

Dodatok A

Obsah CD

Na priloženom CD v adresáry SDK sú všetky potrebné knižnice pre preklad na systéme MS Windows. Potrebné informácie na preklad sú na CD v súbore README.txt. PDF verzia bakalárskej práce je priamo na CD s názvom ibp.pdf. Zdrojové súbory pre latex sú v adresári tex. Zdrojové súbory aplikácie sú v adresáry src.