VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

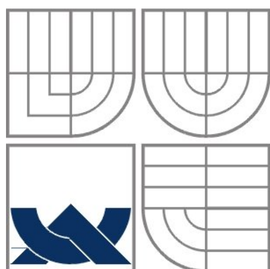
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

OPENSCENEGRAPH

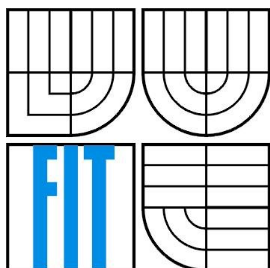
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE PETER JAVORSKÝ
AUTHOR

BRNO 2007



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

OPENSCENEGRAPH

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE PETER JAVROSKÝ
AUTHOR

VEDOUCÍ PRÁCE ING. ADAM HEROUT, PH.D.
SUPERVISOR

BRNO 2007

Abstrakt

OpenSceneGraph je voľne šíriteľný, multiplatformový interface pre programátorov aplikácií umožňujúci vytvorenie komplexného grafu scény, poskytujúceho vysokovýkonné 3D grafické zobrazovanie používané vývojármi aplikácií na vizuálne simulácie, virtuálnu realitu, hry, vedecké zobrazovanie a modelovanie.

Klíčová slova

OpenSceneGraph, 3D, grafické programové vybavenie, vizuálne zobrazovanie

Abstract

The OpenSceneGraph is an open source high performance 3D graphics toolkit, used by application developers in fields such as visual simulation, games, virtual reality, scientific visualization and modelling.

Keywords

OpenSceneGraph, 3D, graphics toolkit, visual rendering

Citace

Peter Javorský: OpenSceneGraph, bakalárska práca, Brno, FIT VUT v Brně, 2007

OpenSceneGraph

Prohlášení

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

……………………
Jméno Příjmení
Datum

Poděkování

Moje poďakovanie patrí Ing. Adamovi Heroutovi, Ph.D. za poskytnutie všetkých informácií týkajúcich sa mojej práce a podrobností jej vypracovania.

© Peter Javorský, 2007.
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

Obsah ... 5
Úvod ... 6
1 Základy ... 6
1.1 Hierarchia grafu scény ... 7
1.2 Typické zloženie grafu scény ... 8
1.3 Pozícia OSG pri tvorbe 3D aplikácie ... 9
2 História ... 10
3 Funkcie grafu scény ... 11
3.1 Ako grafy scény zobrazujú scénu ... 13
4 Prehľad OSG ... 14
4.1 Konvencie pomenovania súčastí OSG ... 14
4.2 Komponenty ... 15
5 Jadro OSG ... 16
5.1 Knižnica osg ... 17
5.1.1 Triedy grafu scény ... 18
5.1.2 Triedy geometrie ... 19
5.1.3 Triedy pre správu stavov ... 20
5.2 Knižnica osgDB ... 21
5.3 NodeKits ... 22
5.4 osgDB pluginy ... 22
6 Inštalácia OSG ... 23
6.1 Inštalácia na platforme Microsoft Windows ... 23
6.1.1 Kompilácia ... 24
6.2 Kompilácia na platforme GNU/Linux ... 26
7 Začíname s OSG ... 26
7.1 Jednoduché útvary ... 26
7.2 Pridanie textúry ... 28
7.3 Vytvorenie animačnej trasy ... 29
7.4 Vytváranie vlastných geometrických telies ... 32
7.5 Uloženie grafu scény do súboru ... 35
7.6 Implementácia kamery ktorá sleduje uzol ... 36
7.7 Výber objektov myšou ... 40
8 Záver ... 44
Literatúra ... 45
Zoznam príloh ... 46

Úvod

Táto práca je správa o tom, ako som detailne preštudoval knižnicu OpenSceneGraph a čo som o nej zistil. OpenSceneGraph je voľne šíriteľný, multiplatformový interface pre programátorov aplikácií (application programmer interface - API) umožňujúci vytvorenie komplexného grafu scény (scene graph).

OpenSceneGraph hrá v dnešnej dobe kľúčovú úlohu v množstve podobných 3D grafických API. V podstate je nadstavbou nad nízkoúrovňovou hardwarovo-abstrakčnou vrstvou[1] OpenGL, poskytujúcou rozsiahlu vysokoúrovňovú interpretáciu (zobrazovanie) 3D scén a podáva priestorovú funkčnosť 3D aplikáciám.

Počas niekoľkých rokov OpenSceneGraph úspešne prekvital s jedinou dostupnou dokumentáciou – jeho zdrojovým kódom. Distribúcia OSG zahŕňala viacero príkladov, ktoré ilustrovali rôzne zobrazovacie efekty a metódy umožňujúce integráciu OSG s aplikáciami určenými pre koncového užívateľa. Tieto ilustratívne príklady spolu so schopnosťou „kráčať" medzi jednotlivými riadkami kódu jadra OSG pri ladení výslednej aplikácie dali možnosť mnohým programátorom stať sa dokonalými a zbehlými odborníkmi na API OSG.

Hoci zdrojový kód v minulosti ako dokumentácia postačoval, v dnešnej dobe už ako náhrada „klasickej" formy dokumentácie nestačí. Rôzne príručky obsahujú rozličné formy ilustratívnych tabuliek a grafov, čo spôsobuje, že sa ťažko vkladajú do zdrojového kódu. Ako OSG rástol a stával sa stále komplexnejším, nedostatok akejkoľvek dokumentácie neprípustným spôsobom predĺžil krivku učenia (teda čas potrebný na naučenie) OSG pre nových užívateľov. Tento nedostatok spôsobil, že sa niektorí vývojári začali zaoberať otázkou, či je OSG už dostatočne zrelý a robustný na vývoj aplikácií na skutočne profesionálnej úrovni.

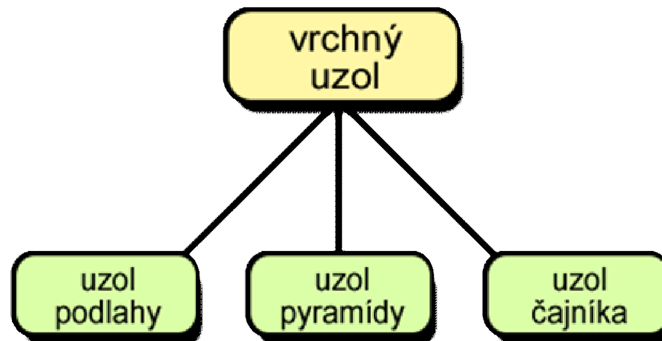
1 Základy

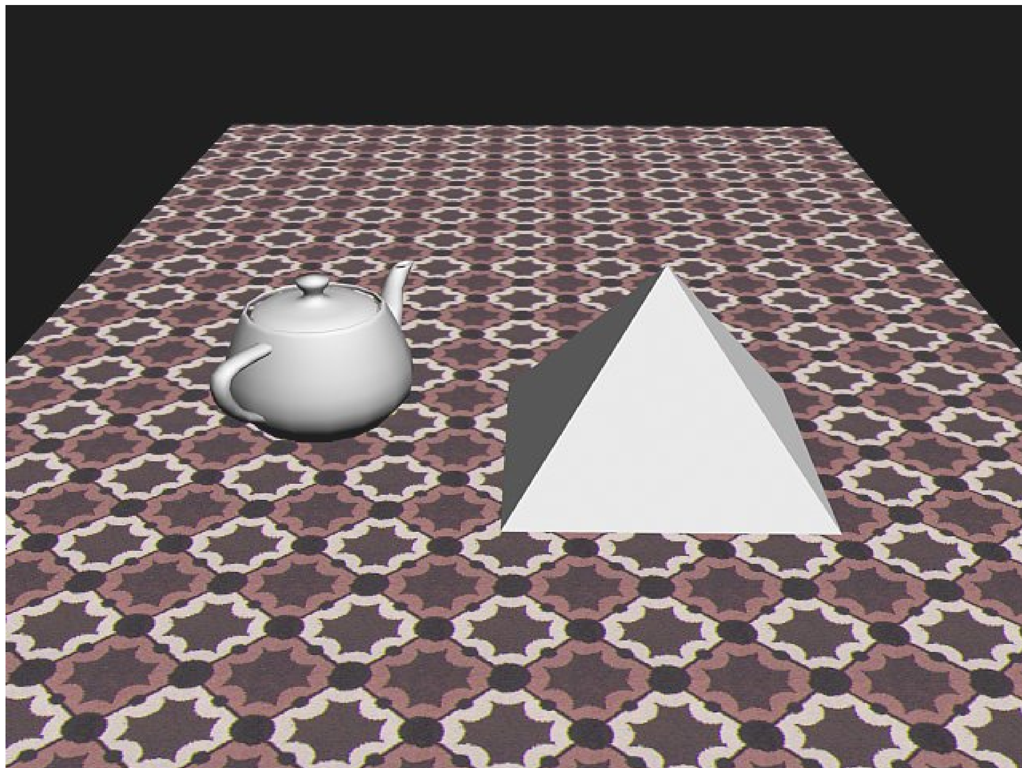
Ak čítate túto prácu, už ste pravdepodobne zbehlí v poli 3D grafických aplikácií. V tejto práci sa predpokladá dobrá znalosť programovacieho jazyka C++, ako aj OpenGL - štandardného nízkoúrovňového API.

[1] hardware abstraction layer - HAL

1.1 Hierarchia grafu scény

Graf scény je stromovo hierarchická štruktúra, ktorá organizuje priestorové geometrické dáta pre ich efektívne zobrazovanie. Obrázok č. 2 predstavuje abstraktný graf scény, zložený z podlahy, pyramídy a čajníka.



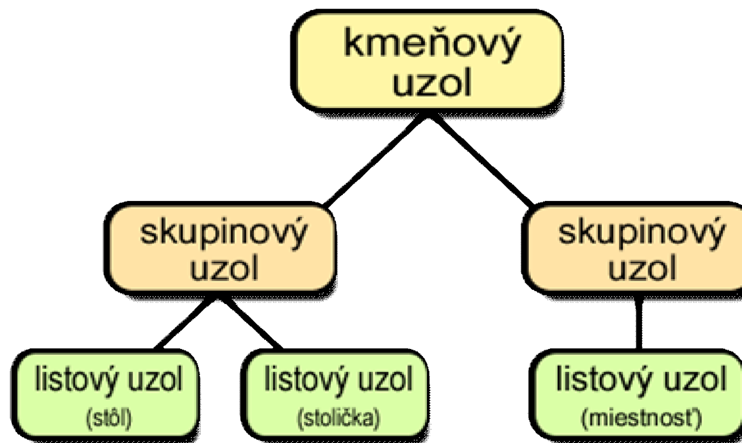


Obr. 2: Jednoduchý, abstraktný graf scény: na zobrazenie scény pozostávajúcej z podlahy, pyramídy a čajníka graf scény naberá formu „z vrchu dole", pričom uzly podlahy, čajníka a pyramídy sú potomkovia vrchného, nadradeného uzla

Na vrchu každého grafu scény sa nachádza kmeňový (tzv. root) uzol, ktorý je vždy na najvyššej úrovni. Pod ním sa väčšinou nachádzajú rôzne skupinové uzly, ktoré obsahujú ďalšie prvky prislúchajúce k danej skupine, väčšinou geometrické útvary, ktoré môžu mať priradené tzv. zobrazovacie stavy[2] – tieto určujú vzhľad danej skupiny, alebo uzla. Skupinové uzly môžu mať nula alebo viac potomkov, aj keď je samozrejmé, že skupinové uzly bez potomkov v grafe scény nemajú žiadny význam. Na spodku grafu sú tzv. listové uzly[3], ktoré obsahujú aktuálnu geometriu, a tá vytvára konkrétne objekty scény. Aplikácie využívajú skupinové uzly pre lepšiu organizáciu scény.

1.2 Typické zloženie grafu scény

Predstavme si 3D scénu obsahujúcu miestnosť so stolom a stoličkou. Mohli by sme graf tejto scény rozdeliť a usporiadať viacerými spôsobmi. Keby sme chceli do jednej skupiny umiestniť stôl a stoličku a do druhej celú miestnosť, hierarchia grafu scény by vyzerala približne ako na obrázku 3. Kmeňový uzol má dva skupinové uzly, jeden pre stôl a stoličku, druhý pre objekt miestnosti. Skupina „stôl a stolička" má dva listové uzly, prvý obsahujúci geometriu stola, druhý obsahujúci geometriu stoličky. Skupina „miestnosť" má ako potomka jediný listový uzol, ktorý obsahuje geometriu pre podlahu, steny a strop miestnosti.



Obr. 3: Typický graf scény: je zložený z kmeňového uzla, skupinových uzlov a listových uzlov. Listové uzly obsahujú geometriu a nemajú žiadne dcérske uzly. Skupinové uzly môžu mať viacero potomkov a umožňujú aplikáciám logicky organizovať geometrické dáta a zobrazovacie stavy. Kmeňový uzol je prvotný rodič všetkých uzlov grafu scény.

---

[2] anglicky **rendering states** – Priraďujú uzlom vlastnosti, akými sa má daný uzol zobraziť.
[3] anglicky **leaf nodes**

Grafy scény väčšinou poskytujú rozličné typy uzlov. Tieto poskytujú široké spektrum funkčnosti, ako napríklad prepínacie uzly (tzv. switch nodes), ktoré zobrazujú alebo skrývajú svojich potomkov, uzly kontrolujúce mieru detailu (LOD – level of detail), podľa vzdialenosti od pozorovateľa, alebo transformačné uzly, ktoré modifikujú alebo transformujú geometriu ich dcérskych uzlov.

Objektovo orientované grafy scény poskytujú túto univerzálnosť prostredníctvom dedenia. Všetky uzly zdieľajú spoločnú základnú triedu so špecializovanou funkčnosťou, definovanou v jej odvodených triedach.

Veľké množstvo typov uzlov a ich schopnosť priestorového organizovania dát poskytuje možnosti ktoré sú nedostupné v tradičných nízkoúrovňových zobrazovacích API.

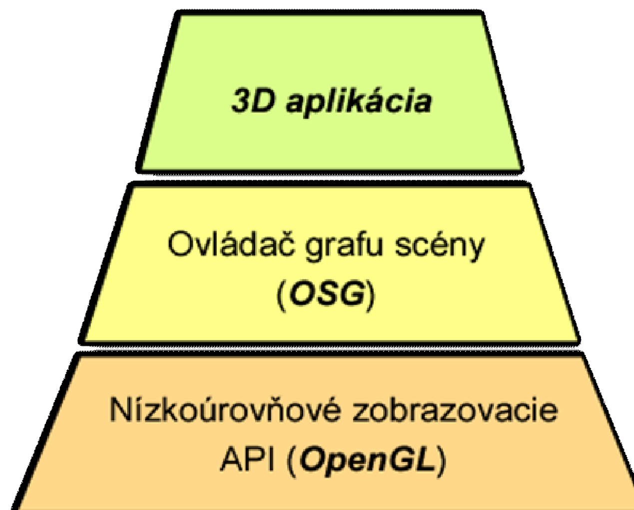
OpenGL a Direct3D sa zameriavajú predovšetkým na využívanie a abstrakciu funkcií, ktoré poskytuje grafický hardware. Aj keď grafický hardware umožňuje uskladnenie geometrických a stavových dát pre neskoršie zobrazenie (ako napr. zobrazovacie zoznamy[4] alebo zásobníkové objekty[5]), nízkoúrovňové možnosti API pre priestorovú organizáciu týchto údajov sú vo všeobecnosti minimálne a neadekvátne pre potreby značnej väčšiny 3D aplikácií.

1.3 Pozícia OSG pri tvorbe 3D aplikácie

Grafy scény sú tzv. middleware – prostredníkom medzi hardvérovým API a výslednou aplikáciou, čiže softvérom. Vybudované sú nad úrovňou nízkoúrovňového HW API, aby zabezpečili schopnosť priestorovej organizácie a iných vlastností vyžadovaných vysoko výkonnými 3D aplikáciami.

[4] tzv. **display lists** – používanie display listov na zobrazovanie komplexných scén je rýchlejšie, napr. objekt gule je zložený z polygónov v tvare trojuholníka, každý jeden polygón (trojuholník) sa uloží do display listu a pri zobrazovaní objektu sa iba určí, ktorý display list sa má vykresliť

[5] angl. **buffer objects**



Obr. 4: OSG je middleware: plní úlohu sprostredkovateľa medzi nízkoúrovňovým API a výslednou 3D aplikáciou, čím aplikácii sprostredkuje prídavné funkcie

Mnoho aplikácií vyžaduje miesto priameho pripájania sa k nízkoúrovňovému API pomocné „middleware" knižnice, akými je aj OpenSceneGraph. Tie jej poskytujú nielen prídavné funkcie, ktoré neuveriteľne uľahčujú prácu programátorovi pri písaní 3D aplikácií, ale aj zlepšujú organizáciu, prehľadnosť grafu scény a funkčnosť celej aplikácie.

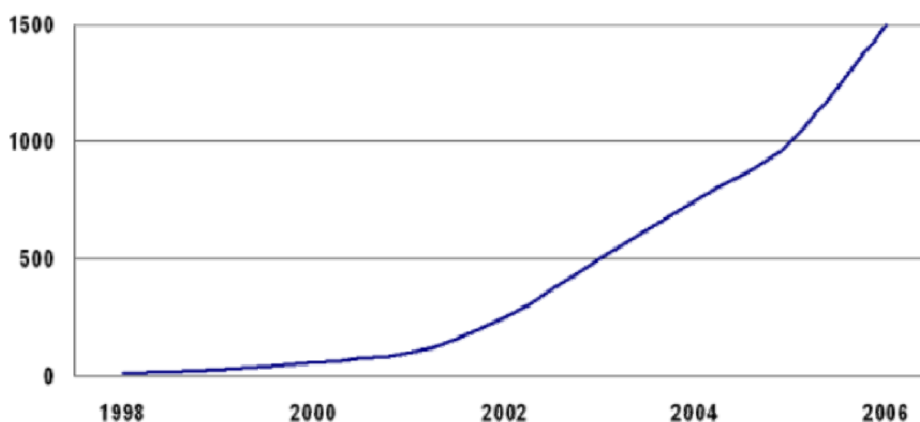
2 História

História OSG sa začala písať v roku 1997, keď bol Don Burns zamestnaný vo firme SGI a jeho záľuba v závesnom lietaní spolu s prístupom k high-end grafickému zobrazovaciemu hardware ho viedla k napísaniu programu simulujúceho závesné lietanie. Tento simulátor bežal pod systémom SGI Onyx, čo bol vysoko výkonný renderovací hardware.

V roku 1999 začal pomáhať Donovi s vývojom tohto simulátora Robert Osfield, ktorý prepísal prvky ovládajúce graf scény na platformu Windows. V septembri 1999 sa zdrojový kód stal voľne šíriteľným, vznikol názov OpenSceneGraph a stránka www.openscenegraph.org, pričom Osfield prevzal vedenie nad projektom OSG a Burns sa venoval naďalej vývoju svojho simulátora. V roku 2000 sa pre Osfielda tento koníček zmenil na „posadlosť", v roku 2001 sa z „posadlosti" stala profesia. V apríli 2001, ako odpoveď na rastúci svetový záujem sa Osfield začal plne angažovať v projekte OSG a založil spoločnosť

OpenSceneGraph Professional Services, poskytujúcu komerčnú podporu, konzultačné služby a výučbové tréningy zamerané na programovanie aplikácií využívajúcich OSG.

Funkcie a prídavné knižnice OSG boli od tohto momentu vyvíjané rapídnym tempom. V roku 2003 bola pre program Magic Earth vytvorená knižnica *Producer* (pôvodne OSGMP), ktorá poskytovala mnohovláknové zobrazovacie schopnosti pre OSG. V roku 2004 boli pridané funkcie pre stránkovanie veľkých databáz, pridaná bola taktiež podpora terénu a shaderov. Rok 2006 znamenal pre OSG kompletné prepracovanie pluginu *OpenFlight* a vznik *osgViewer*, integrovanej knižnice ovládajúcej kameru.



Obr. 1: Rast počtu registrovaných užívateľov diskusnej skupiny *osg-users*

3 Funkcie grafu scény

Grafy scény interpretujú geometriu a spravujú stavové údaje funkciami ktoré sa dajú nájsť v nízkoúrovňových API, ale taktiež poskytujú dodatočné funkcie a schopnosti, ktoré rozširujú možnosti grafu scény, ako sú napríklad:

- **Priestorová organizácia** – štruktúra stromu grafu scény sama o sebe poskytuje intuitívnu priestorovú organizáciu, ktorá môže byť počas behu aplikácie ľubovoľne preusporiadaná alebo menená.

- **Culling** – Zobrazovací ihlan, alebo šírka zobrazenia a tzv. *backface culling*[6] redukuje celkové zaťaženie CPU tým, že sa neberie do úvahy geometria objektov, ktorá v konečnej zobrazenej scéne nebude viditeľná.

- **LOD** – Výpočet vzdialenosti pozorovateľa od objektu umožňuje efektívnejšie zobrazovanie objektov použitím nižšej úrovne detailu pre príliš vzdialené objekty. Prvky grafu scény môžu byť dokonca pri dosiahnutí určitej vzdialenosti od pozorovateľa načítané z disku do pamäti a pridané do grafu scény, alebo naopak vymazané, ak sa nachádzajú za hranicou tejto vzdialenosti.

- **Priehľadnosť** – Zobrazovanie geometrie priehľadných objektov vyžaduje, aby všetky priesvitné telesá boli pri zobrazovaní scény vyrenderované až po tom, ako boli vyrenderované pevné (nepriehľadné) telesá. Priehľadná geometria musí byť dokonca triedená podľa hĺbky a vykreslená v poradí zozadu dopredu. Tieto operácie sú bežne podporované grafmi scén.

- **Minimalizácia zmien stavov zobrazovania** – Pre maximalizáciu výkonu aplikácie by sa malo predchádzať prebytočným a nepotrebným zmenám stavov zobrazovania. Je bežné, že grafy scén triedia geometriu podľa jej nastaveného stavu zobrazovania a správa OSG zmien stavov takéto redundantné zmeny eliminuje.

- **Súborový vstup/výstup** – Grafy scén sú efektívnymi nástrojmi na čítanie a zapisovanie uzlov alebo celých scén do súborov. Po načítaní do pamäti umožňuje vnútorná štruktúra grafu scény aplikácii jednoducho manipulovať s dynamickými 3D dátami alebo ich prevádzať do iných typov súborov.

- **Dodatočná vysokoúrovňová funkčnosť** – knižnice grafov scén poskytujú vysokoúrovňovú funkčnosť, ktorá je oveľa pokročilejšia ako tá, ktorú môžeme nájsť v nízkoúrovňových API. Je ňou napríklad plnohodnotná podpora práce

[6] v počítačovej grafike pojem **backface culling** určuje, či sú polygóny daného objektu zobrazené, alebo nie – v závislosti na pozícii kamery od objektu

s textom a nápismi, zobrazovacie efekty (časticové systémy, tiene, atď.) a multiplatformový prístup k vstupným zariadeniam a zobrazovacím plochám.

Prinajmenšom niektoré z týchto funkcií sú v dnešnej dobe potrebné pre všetky 3D aplikácie. Výsledok je ten, že vývojári ktorí stavajú svoje aplikácie priamo na nízkoúrovňovom API sú skôr či neskôr prinútení implementovať v ich aplikácií mnoho z týchto nízkoúrovňových funkcií, čo zvyšuje čas a náklady potrebné na vývoj aplikácie. Použitie verejne dostupného grafu scény, akým je napríklad OSG, razantne urýchli vývoj aplikácie, pretože tieto funkcie už plne podporuje.

3.1 Ako grafy scény zobrazujú scénu

Uvažujme triviálnu implementáciu grafu scény, ktorá by umožňovala vkladanie geometrie a zobrazovanie výslednej scény. Všetka geometria uložená v grafe scény by bola pri zobrazovaní scény posielaná do hardvéru v podobe OpenGL príkazov. Akokoľvek, takejto implementácii by chýbali mnohé funkcie popísané v predchádzajúcej časti. Pre podporu dynamickej aktualizácie geometrie scény, cullingu, triedenia a efektívneho zobrazovania grafy scény poskytujú viac než iba jednoduchý *traverzálny priechod*[7] scénou pri procese zobrazovania. Vo všeobecnosti existujú 3 typy traverzálnych priechodov:

- **Aktualizačný** (*Update*) – aktualizačný traverzálny priechod umožňuje aplikácii modifikovať graf scény, čo poskytuje podporu pre dynamické scény. Aktualizácie sú dosiahnuté buď priamo aplikáciou, alebo takzvanými *callback funkciami* priradenými k uzlom grafu scény.

- **Vyraďovací** (*Cull*) – počas vyraďovacieho priechodu knižnica grafu scény testuje hraničné objemy všetkých uzlov pre zahrnutie v scéne. Ak tento test indikuje, že by listový uzol mal byť zobrazený, odkaz na geometriu tohto listového uzlu je pridaný do zoznamu pre konečné vyobrazenie.

[7] **traverzálny (krížový) priechod** znamená proces navštívenia každého uzla v stromovej dátovej štruktúre, akou je napríklad graf scény, pričom každý uzol je systematickým spôsobom navštívený práve raz

- **Zobrazovací** (*Render*) – v zobrazovacom priechode (niekedy označovaný ako vykresľovací priechod) graf scény prejde zoznam geometrie určenej na vykreslenie (vytvorený vyraďovacím priechodom) a interpretuje ju poslaním nízkoúrovňových volaní API do hardvéru (grafickej karty), ktoré túto geometriu vyobrazia.

Tieto priechody sú normálne vykonané raz pre každý nový vyobrazený snímok. Pre aplikácie ktoré vyžadujú súčasné vyobrazenie viacero rôznych pohľadov na tú istú scénu musia byť tieto priechody (okrem aktualizačného) vykonané pre každý pohľad zvlášť. Vyraďovací priechod musí ostať operáciou len na čítanie aby mohla byť využitá podpora viacvláknového prístupu.

4 Prehľad OSG

OSG je skupina voľne šíriteľných knižníc, ktoré poskytujú správu scény a optimalizáciu grafického zobrazovania aplikáciám. Sú napísané v prenositeľnom ANSI C++ a na zobrazovanie používajú štandardné nízkoúrovňové grafické OpenGL API. Výsledkom je, že OSG je multiplatformové a beží na platformách Windows, Max OS X a na väčšine UNIXových a Linuxových operačných systémoch. Prevažná časť OSG pracuje nezávisle na miestnom uživateľskom prostredí, ktoré spravuje systém okien. OSG je dostupné pod modifikovanou LGPL softvérovou licenciou.

4.1 Konvencie pomenovania súčastí OSG

Nasledujúci zoznam definuje jednotlivé zvyklosti pomenúvania používané v zdrojovom kóde OSG:

- **Menné priestory** – menný priestor OSG začína vždy prefixom malými písmenami. Príklady: **osg**, **osgSim**, **osgFX**.

- **Triedy** – mená tried OSG začínajú veľkým písmenom. Ak je meno triedy zložené z viacerých slov, každé nové slovo začína tiež veľkým písmenom. Príklady: **MatrixTransform**, **NodeVisitor**, **Optimizer**.

- **Metódy tried** – mená metód v rámci OSG začínajú malým písmenom. Ak je meno metódy zložené z viacerých slov, každé ďalšie slovo začína tiež veľkým písmenom. Príklady: **addDrawable()**, **getNumChildren()**, **setAttributeAndModes()**.

- **Mená členských premenných** – V rámci tried používajú mená členských premenných rovnaké konvencie ako mená metód

- **Šablóny** – mená šablón OSG sú malými písmenami, viacslovné názvy sú oddelené podčiarnikom. Príklady: **ref_ptr<>**, **graph_array<>**, **observer_ptr<>**.

- **Statické premenné** – Statické premenné a funkcie začínajú s prefixom „**s_**", ale inak používajú rovnaké konvencie pomenovania, ako mená členských premenných a metódy tried. Príklady: *s_applicationUsage, a_ArrayNames()*.

- **Globálne premenné** – Globálne premenné začínajú prefixom „**g_**". Príklady: *g_NotifyLevel, g_readerWriter_BMP_Proxy*.

4.2 Komponenty

OSG funguje ako sada dynamicky nahrávaných knižníc (alebo zdieľaných objektov) a spúšťateľných pomocných utilít. Tieto knižnice spadajú do piatich konceptuálnych kategórií:

- Knižnice jadra OSG poskytujú najzákladnejšiu, esenciálnu funkčnosť grafu scény a zobrazovania, ako aj dodatočnú funkčnosť typicky vyžadovanú 3D aplikáciami.

- Tzv NodeKits („Uzlové súpravy“) rozširujú funkcie tried jadra OSG a poskytujú typy uzlov vyššej úrovne a špeciálne efekty.

- Pluginy OSG sú knižnice, ktoré čítajú a zapisujú súbory 2D obrazov a 3D modelov.

- Knižnice pre podporu interoperability OSG umožňujú jednoduchú integráciu OSG do iných prostredí, akými sú napríklad dynamické skriptovacie jazyky - Python alebo Lua.

- Rozširujúca zbierka aplikácií a príkladov, ktorá je súčasťou balíka stiahnuteľného z webu OSG, poskytuje užitočné funkcie a demonštruje správne používanie OSG.

5 Jadro OSG

Jadro OSG poskytuje základné funkcie operujúce s grafom scény a dodatočné funkcie vyžadované 3D aplikáciami, ako napríklad prístup k pluginom pre načítanie 2D/3D súborov. Skladá sa z týchto knižníc:

- Knižnica **osg** – Táto knižnica obsahuje triedy, ktoré vaša aplikácia použije na vytvorenie grafu scény. Okrem toho obsahuje triedy pre geometriu, vektorové a maticové výpočty a správu stavov zobrazovania. Ostatné triedy obsiahnuté v *osg* poskytujú napríklad analýzu argumentov, správu dráh animácie a komunikáciu chybových hlásení.

- Knižnica **osgUtil** – Obsahuje triedy a funkcie na prácu s grafom scény a jeho obsahom. Okrem iného poskytuje funkcie na zhromažďovanie štatistík, optimalizáciu grafov scény a vytváranie finálneho (zobrazovacieho – render) grafu. Ďalej sú tu triedy pre geometrické operácie, napríklad Delaunayova triangulácia, obtiahnutie trojuholníkov alebo generovanie koordinátov textúr.

- Knižnica **osgDB** – Obsahuje triedy a na vytváranie a zobrazovanie 3D databáz, čiže súpis OSG pluginov pre čítanie a zapisovanie 2D a 3D súborov, ako aj triedy pre prístup k týmto pluginom. *osgDB* podporuje dynamické nahrávanie (a uvoľňovanie) veľkých databázových segmentov.

- Knižnica **osgViewer** – Táto knižnica obsahuje triedy, poskytujúce podporu pre zobrazovanie scén. Umožňuje rôznym zobrazovacím plochám priradiť rôzne kamery a zabezpečuje OSG integráciu do rôznych okenných prostredí, ktoré sú v rôznych operačných systémoch.

- V súčasnom vydaní OSG sa nachádza aj knižnica **osgGA** pre adaptáciu udalostí užívateľského inerface-u. V blízkej budúcnosti bude však táto súčasť OSG predizajnovaná tak, aby bola časť funkcií *osgGA* prenesená do knižnice *osgViewer* a samotná *osgGA* bude ako samostatná knižnica z OSG vyradená.

5.1 Knižnica osg

Srdce OpenSceneGraph-u tvorí knižnica *osg*. Definuje jadrové uzly, ktoré vytvárajú základ grafu scény, ako aj niekoľko prídavných tried zameraných na manažment grafu scény a vývoj aplikácií. Niektoré z nich budú popísané v nasledujúcom texte.

5.1.1 Triedy grafu scény

Triedy grafu scény sú zamerané na konštrukciu grafu scény. V OSG sú všetky triedy grafu odvodené od *osg::Node*, a z konceptuálneho hľadiska sú kmeňové, skupinové a listové uzly všetky iné typy uzlov. Tieto sú napokon aj tak zdedené od *osg::Node* a špecializované triedy sa líšia funkciami, ktoré grafu scény poskytujú. Kmeňový uzol v OSG je taktiež špeciálny typ uzlu; je to jednoducho typ *osg::Node*, ktorý nemá žiadny rodičovský uzol.

- **Node** – Trieda *osg::Node* je základná trieda pre všetky typy uzlov v grafe scény. Obsahuje triedy, ktoré napomáhajú traverzálnym priechodom, vyraďovacím priechodom, spätnými volaniami (callbackmi) a manažmentom stavov zobrazenia.

- **Group** – Trieda *osg::Group* reprezentuje skupinový uzol. Je to v podstate základná trieda pre akýkoľvek uzol ktorý môže mať potomkov. Je to kľúčová trieda pre lepšiu priestorovú organizáciu v grafe scény.

- **Geode** – Trieda *osg::Geode* (odvodené od Geometry Node – uzol geometrie) reprezentuje listové uzly s geometriou v grafe scény. Nemá žiadnych potomkov, ale obsahuje objekty typu *osg::Drawable* (viz nižšie) ktoré obsahujú geometriu určenú na vykreslenie.

- **LOD** – Trieda *osg::LOD* zobrazuje svojich potomkov s rozličnou mierou detailu alebo ostrosti na základe ich vzdialenosti od pozorovateľa. Toto je využité na vytvorenie objektov s rôznou mierou detailu v scéne. Objekty ďalej od pozorovateľa sú zobrazované s menšími detailmi, čo má za následok skrátenie času potrebného na vygenerovanie aktuálneho snímku.

- **MatrixTransform** – Trieda osg::MatrixTransform obsahuje maticu, ktorá transformuje geometriu jej potomkov a umožňuje geometriu otáčať, meniť jej veľkosť, posúvať ju a podobne.

- **Switch** – Trieda *osg::Switch* implementuje systém prepínača, ktorý booleovskou maskou zapína alebo vypína viditeľnosť geometrie jeho potomkov.

Takto by vyzeral príklad, ako vytvoriť jednoduchú hierarchiu grafu scény s kmeňovým uzlom (skupinovým – typu osg::Group) a jednou geodou, ktorú by sme kmeňovému uzlu priradili.

```
// kmenovy uzol
osg::Group *rootNode = new osg::Group;
// vytvor geodu, v ktorej sa bude nachadzat geometria
osg::Geode* geode = new osg::Geode();
rootNode->addChild(geode); // pridaj kmenovemu uzlu geodu ako potomka
```

Tento príklad ukazuje, ako sa dá použiť transformačný uzol MatrixTransform na pootočenie geometrie geody o 90 stupňov v x-ovej osi, posunutie 10 jednotiek v smere y-ovej osi a zdvojnásobenie jej veľkosti.

```
osg::MatrixTransform* mt = new osg::MatrixTransform;
mt->setMatrix(osg::Matrix::translate(osg::Vec3d(0.0f,10.0f,0.0f))*
              osg::Matrix::scale(2.0f,2.0f,2.0f)*
              osg::Matrix::rotate(osg::inDegrees(90.0f),0.0f,0.0f,1.0f));
mt->addChild(geode);
root->addChild(mt);
```

5.1.2 Triedy geometrie

Trieda *osg::Geode* je v OSG listovým uzlom, ktorý obsahuje geometriu na vykreslenie. Nasledujúce triedy sa používajú na ukladanie geometrických dát v *Geode*:

- **Drawable** – Trieda *osg::Drawable* („vykresliteľná“) je základnou triedou pre ukladanie geometrie. *osg::Geode* udržiava zoznam pozostávajúci z jednotlivých *osg::Drawable*. *Drawable* je čisto virtuálna trieda a jej inštancie nemôžu byť vytvárané priamo. Musíte použiť jednu z odvodených tried, ako napríklad *osg::Geometry* alebo osg::ShapeDrawable, ktoré umožňujú vašej aplikácii zobrazovať preddefinované tvary (napr. gule, kužele, kocky a podobne)

- **Geometry** – Trieda *osg::Geometry* v spolupráci s triedou *osg::PrimitiveSet* sa správajú ako vysokoúrovňové obálky na funkčnosť vertexových polí[8] OpenGL. Trieda *Geometry* uchováva vertexové polia, koordináty textúr, farby a polia normál.

- **PrimitiveSet** – Trieda *osg::PrimitiveSet* poskytuje vysokoúrovňovú podporu pre OpenGL príkazy vertexových polí. Použite túto triedu na špecifikovanie typu primitív, ktoré sa majú vykresliť z dát uložených v asociovanej triede *osg::Geometry*.

- Vektory (**Vec2**, **Vec3**, ...) – OSG poskytuje skupinu preddefinovaných dvoj-, troj- a štvorprvkových vektorov typu float a double. Používajú sa na špecifikáciu bodov v priestore, farieb, normál a koordinátov textúr.

- Vektorové polia (**Vec2Array**, **Vec3Array**, ...) OSG definuje niekoľko bežne používaných typov vektorových polí, ako napríklad Vec2Array pre koordináty textúr. Pri zadávaní dát vertexových polí vaša aplikácia ukladá dáta v týchto poliach pred tým ako ich predá objektom *osg::Geometry*.

Zosumarizovať by sa to dalo asi takto: Geometrické uzly – čiže Geody (Geodes – Geode Nodes) sú v grafe scény listové uzly, ktoré ukladajú vykresliteľnú geometriu v podobe takzvaných Drawables (*osg::Drawable*). Geometria (osg::Geometry) ako jeden typ Drawable ukladá v sebe dáta vertexových polí ako aj príkazy potrebné na vykreslenie týchto vertexových polí.

5.1.3 Triedy pre správu stavov

OSG poskytuje mechanizmus na ukladanie požadovaného stavu zobrazenia (StateSet) v grafe scény. Počas vyraďovacieho priechodu je zoskupená geometria, ktorá má byť vykreslená

[8] **vertex** je pojem označujúci bod v priestore, vertexové polia sú polia bodov v priestore

s rovnakým stavom (napr. všetka geometria so zapnutým GL_LIGHTNING) a vykreslená je naraz pre minimalizáciu zmien OpenGL stavov.

- **StateSet** – OSG ukladá kolekciu hodnôt OpenGL stavov (tzv. režimy a atribúty) v triede **StateSet**. Akémukoľvek uzlu v grafe scény môže byť priradený StateSet.

- Režimy – Analogicky k OpenGL príkazom **glEnable()** a **glDisable()** umožňujú zapínanie a vypínanie vstavaných funkcií OpenGL na ovládanie osvetľovania, miešania alebo hmly. Na uloženie režimu do StateSet-u sa používa metóda **osg::StateSet::setMode()**

- Atribúty – Umožňujú určiť stavové parametre, medzi ktoré patrí hlavne zmiešavacie funkcie, vlastnosti materiálov a farbu hmly. Na uloženie atribútu do StateSet-u sa používa metóda **osg::StateSet::setAttribute().**

- Textúrové atribúty a režimy – Tieto sa používajú hlavne pri multitexturingu. Na nastavenie týchto parametrov sa používajú funkcie **setTextureMode()** a **setTextureAttribute()**.

Tento systém na správu stavov sa ukázal byť ako veľmi flexibilný. Všetky nové stavy použiteľné v OpenGL, vrátane OpenGL Shading Language boli do správy stavov OSG ľahko implementované. Ako používať StateSet si ukážeme v demopríkladoch.

5.2 Knižnica osgDB

Knižnica osgDB umožňuje aplikáciám otvárať, používať a zapisovať 3D databázy. Poskytuje podporu pre široké spektrum bežne používaných formátov 2D a 3D súborov. Využíva pritom technológiu pluginov, pričom na každý formát je jeden plugin. osgDB spravuje register týchto pluginov.

Napríklad na načítanie 3D súboru glider.osg by sme použili funkciu `readNodeFile()` z knižnice osgDB:

```
osg::Node* glider = osgDB::readNodeFile("glider.osg");
```

5.3 NodeKits

NodeKits, tzv. „Uzlové súpravy“ rozširujú koncept uzlov, Drawable objektov a StateSetov a sú rozšírením jadra OSG.

OSG verzia 1.3 má 6 NodeKitov:

- Knižnica osgFX – Tento NodeKit poskytuje dodatočné uzly grafu scény pre zobrazovanie špeciálnych efektov ako anisotropické osvetľovanie, bump mapping alebo „rozprávkové“ tieňovanie (cartoon shading).

- Knižnica osgParticle – Tento NodeKit poskytuje časticové špeciálne efekty, ako napríklad výbuchy, oheň a dym.

- Knižnica osgSim – Tento NodeKit poskytuje špeciálne zobrazovacie požiadavky simulačných systémov a OpenFlight databáz.

- Knižnica osgText – poskytuje výkonný nástroj na pridávanie textu do scény. Plne podporuje všetky TrueType fonty.

- Knižnica osgTerrain – poskytuje podporu na vykresľovanie výškových dát.

- Knižnica osgShadow – Tento NodeKit poskytuje framework na podporu vykresľovania tieňov.

5.4 osgDB pluginy

Jadrové knižnice OSG podporujú súborový vstup/výstup mnohých 2D a 3D formátov. *osgDB::Registry* spravuje načítanie knižníc pluginov automaticky. Aplikácia jednoducho

zavolá funkciu na načítanie alebo zápis súboru a pokiaľ je potrebný plugin dostupný, *Registry* ho vyhľadá a použije.

OSG v1.3 podporuje výber bežne používaných 2D obrázkových formátov, vrátane .bmp, .dds, .gif, .jpeg, .pic, .png, .rgb, .tga, a .tiff. Je tu taktiež QuickTime plugin na načítanie súborov videa.

Podpora OSG pre 3D súbory je všeobecne obsiahla a zahŕňa nasledujúce súborové formáty: Collada (.dae), ESRI Shapefile (.shp), OpenFlight (.flt), Carbon Graphics' Geo (.geo), 3D Studio Max (.3ds), Quake (.md2), Alias Wavefront (.obj) a Terrex TerraPage (.txp).

Ďalej sú to natívne 3D formáty, vlastné pre OSG, a tými sú textový vstavaný formát .osg a binárny .ive. Do týchto formátov je možné uložiť celý graf scény, vrátane všetkých jeho uzlov.

6 Inštalácia OSG

V tejto kapitole popíšem ako nainštalovať a začať používať OSG.

Dôležitým krokom je všetky závislosti OSG ako aj ukážkový Sample DataSet a takzvané 3rd-party knižnice (knižnice tretích strán) umiestniť správne hierarchicky do adresárov. V nasledujúcej kapitole tento postup podrobne vysvetlím.

Ďalej sa budem venovať vysvetleniu postupu, ako OSG skompilovať aby bol pripravený na používanie. Vysvetlený postup je pre platformy Linux (GCC) a Microsoft Windows a Visual Studio 2005 .NET, ale Visual Studio vo verzii 7.x (2003) som tiež skúšal a je ho taktiež možné použiť. Odporúčam však verziu 2005 .NET, kvôli chybám vo verzii 7.x.

6.1 Inštalácia na platforme Microsoft Windows

Prvým krokom, ktorý treba vykonať je stiahnutie aktuálneho stabilného vydania zdrojových kódov zo stránky OSG. Konkrétne sa jedná o adresu http://www.openscenegraph.com/osgwiki/pmwiki.php/Downloads/CurrentRelease. Ja som mal dostupnú verziu 1.2.

Ďalej je potrebné stiahnutie dvoch hlavných závislostí, bez ktorých nie je možné OpenSceneGraph skompilovať. Sú nimi OpenThreads a Producer. Tieto sú dostupné na adresách http://www.andesengineering.com/Producer/download.html (Producer) a http://openthreads.sourceforge.net/ (OpenThreads).

Taktiež budeme potrebovať 3rd-party knižnice, ktoré sú dostupné pod jedným uceleným .zip archívom na adrese:
http://openscenegraph.org/downloads/dependencies/3rdParty_Win32binaries_2005_05_10.zip
Teraz je potrebné všetko rozbaliť a hierarchicky umiestniť do adresárov. Postup je nasledovný:

- Rozbaliť zdrojový kód OSG
- Rozbaliť OpenThreads, Producer a 3rd-party binárne knižnice do adresárov ..\OpenThreads, ..\Producer a ..\3rdParty

- Stiahnuť Sample Dataset, ktorý obsahuje dátové súbory používané priloženými príkladmi k zdrojovým kódom OSG. Tento Sample Dataset sa dá stiahnuť na adrese http://www.openscenegraph.com/osgwiki/pmwiki.php/Downloads/SampleDataset
  Rozbalíme ho do adresára ..\OpenSceneGraph-Data

Teraz máme všetky potrebné zdrojové kódy a závislosti pripravené k používaniu a môžeme sa pustiť do kompilácie.

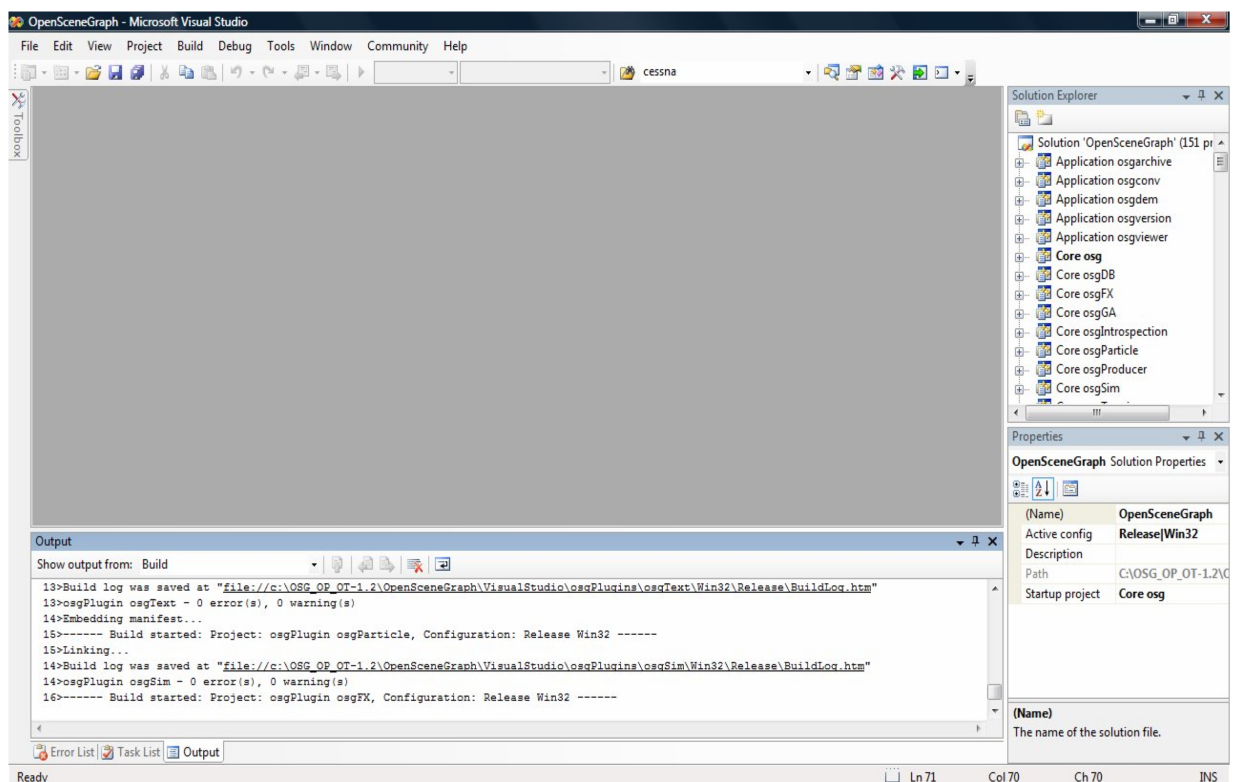
6.1.1 Kompilácia

Ako prvé treba skompilovať OpenThreads. Vo VisualStudiu otvoríme súbor ..\OpenThreads\win32_src\OpenThreads.dsw. Pri otázke, či súbory skonvertovať do novej verzie VisualStudia zvolíme Yes To All. Z horného menu výberu konfigurácie vyberieme požadovanú konfiguráciu (odporúčam už bez ladiacich informácií, teda Release a nie Debug) a stlačíme F6 pre kompiláciu. Pri úspešnej kompilácii sa nám v adresári

..\OpenThreads\bin\win32\ objaví skompilovaná dynamická knižnica OpenThreadsWin32.dll.

Ako druhý treba skompilovať Producer. Otvoríme Producer.sln a postupujeme analogicky. Ak všetko prebehlo úspešne, v adresári ..\Producer\bin\win32\ sa nám objaví dynamicky linkovateľná knižnica Producer.dll.

Po tom, ako sme dokončili kompiláciu Producera, sa konečne môžeme pustiť do kompilácie OpenSceneGraphu. Otvoríme si súbor ..\OpenSceneGraph\VisualStudio\OpenSceneGraph.dsw a opäť potvrdíme „Yes To All“ pre konverziu na novšiu verziu VisualStudia. Zvolíme typ konfigurácie na Release a tlačítkom F6 spustíme build. Táto kompilácia trvá pomerne dlhú dobu. Po jej dokončení by sme v adresári ..\OpenSceneGraph\bin\win32\ mali vidieť všetky výsledné skompilované .exe a .dll súbory.



Obr. 5: Kompilácia OpenSceneGraphu vo VisualStudiu

6.2 Kompilácia na platforme GNU/Linux

Na platforme Linux netreba sťahovať 3rd-party knižnice, stiahnutý archív OSG je pripravený na kompiláciu. Treba vojsť do adresára ../OpenSceneGraph a napísať: **make**. Až sa OSG skompiluje, treba sa zalogovať ako root a nainštalovať ho príkazom: **make install**.

7 Začíname s OSG

Teraz, keď máme OSG skompilovaný, môžeme sa začať venovať tvorbe grafu scény.

Všetky ukážkové príklady, ktoré tu rozoberiem sa nachádzajú na priloženom disku DVD.

7.1 Jednoduché útvary

Teraz rozoberiem prvý demopríklad, v ktorom ukážem ako vytvoriť jednoduchý graf scény so štyrmi objektmi, jednoduchými útvarmi akými sú kocka, cylinder, kužeľ a guľa.

Najprv je potrebné vytvoriť tzv. „prehliadač", inštanciu triedy `osgProducer::Viewer`, ktorý bude náš graf scény zobrazovať. Príkazom `viewer.setUpViewer` nastavíme náš viewer na štandardné nastavenia.

```
#include <osg/ShapeDrawable>
#include <osgProducer/Viewer>
#include <osgDB/ReadFile>

int main( int, char **)
{

// vytvor instanciu triedy osgProducer::Viewer
osgProducer::Viewer viewer;

// nastav na standardne nastavenia
viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
```

Teraz vytvoríme kmeňový skupinový uzol `rootNode` a geodu `geode`. Do nej budeme pridávať geometrické objekty, ktoré sa budú mať vykresliť.

```
osg::Group *rootNode = new osg::Group; // kmenovy uzol
osg::Geode *geode = new osg::Geode(); // vytvor geodu do ktorej budeme vkladat
objekty


// vloz 4 rozne geometricke objekty typu ShapeDrawable
geode->addDrawable(new osg::ShapeDrawable(
      new osg::Sphere(osg::Vec3(-2.0f,0.0f,0.0f),1.1f))); // gula
```

```
geode->addDrawable(new osg::ShapeDrawable(
      new osg::Cone(osg::Vec3(0.0f,0.0f,-0.5f),1.0f,2.0f))); // kuzel

geode->addDrawable(new osg::ShapeDrawable(
      new osg::Cylinder(osg::Vec3(2.0f,0.0f,0.0f),1.0f,2.0f))); // cylinder

geode->addDrawable(new osg::ShapeDrawable(
      new osg::Box(osg::Vec3(4.0f,0.0f,0.0f),1.8f))); // kocka
```

Vytvoríme ďalší uzol do scény (typu `osg::Node*`), nazveme ho `listovy_uzol` a priradíme mu nech ukazuje na našu geodu s geometriou. Následne ho pomocou funkcie `addChild` priradíme kmeňovému uzlu ako potomka. Ďalším krokom je nastavenie zobrazovaných dát viewera na kmeňový uzol, ktorý však treba pretypovať na `osg::Node*`.

```
osg::Node *listovy_uzol = geode; // listovy uzol bude obsahovat geometriu z geode
rootNode->addChild(listovy_uzol); // pridaj kmenovemu uzlu listovy uzol ako potomka

viewer.setSceneData((osg::Node *)rootNode); // na zobrazenie nastav kmenovy uzol
```

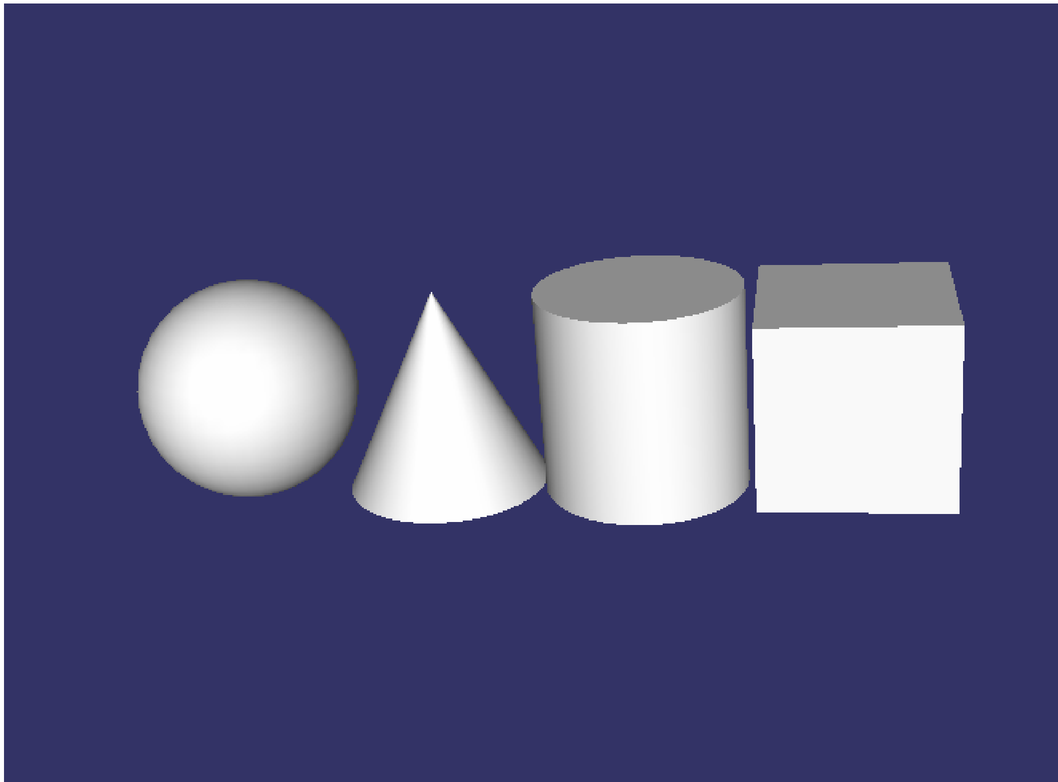
Teraz môžeme náš viewer spustiť, na to použijeme príkaz `viewer.realize()` a jednoduchý cyklus:

```
viewer.realize();

    while( !viewer.done() )              // hlavny cyklus
    {
        viewer.sync();
        viewer.update();
        viewer.frame();
    }
```

Po ukončení grafu scény ešte treba zavolať tieto 3 príkazy na korektné ukončenie práce viewera.

```
    viewer.sync();
    viewer.cleanup_frame();
    viewer.sync();

    return 0;
}
```

Hotovo. Takto napísaný program nám vytvorí graf scény so štyrmi geometrickými útvarmi. Výsledok by mal vyzerať približne takto:



Obr. 6: Jednoduchý graf scény

7.2 Pridanie textúry

V tejto časti si popíšeme, akým spôsobom pridať textúru našim štyrom geometrickým útvarom.

Prvé, čo treba spraviť je vytvoriť objekt textúry, ktorý bude typu `osg::Texture2D` a načítať do neho obrázok. Pre svoje druhé tutoriálové demo som zvolil textúru loga FIT.

```
// vytvor novy objekt textury
osg::Texture2D* texture = new osg::Texture2D;
// nacitaj obrazok loga FIT
texture->setImage(osgDB::readImageFile("Images/fit_logo_cz.gif"));
```

Keď máme vytvorený objekt textúry a načítaný do neho obrázok, treba vytvoriť StateSet z našej geody v ktorej máme geometriu. Toto spravíme pomocou príkazu `getOrCreateStateSet()`. Tomuto StateSetu už len treba textúru priradiť príkazom `setTextureAttributeAndModes()` a zobrazí sa nám na našich geometrických útvaroch.

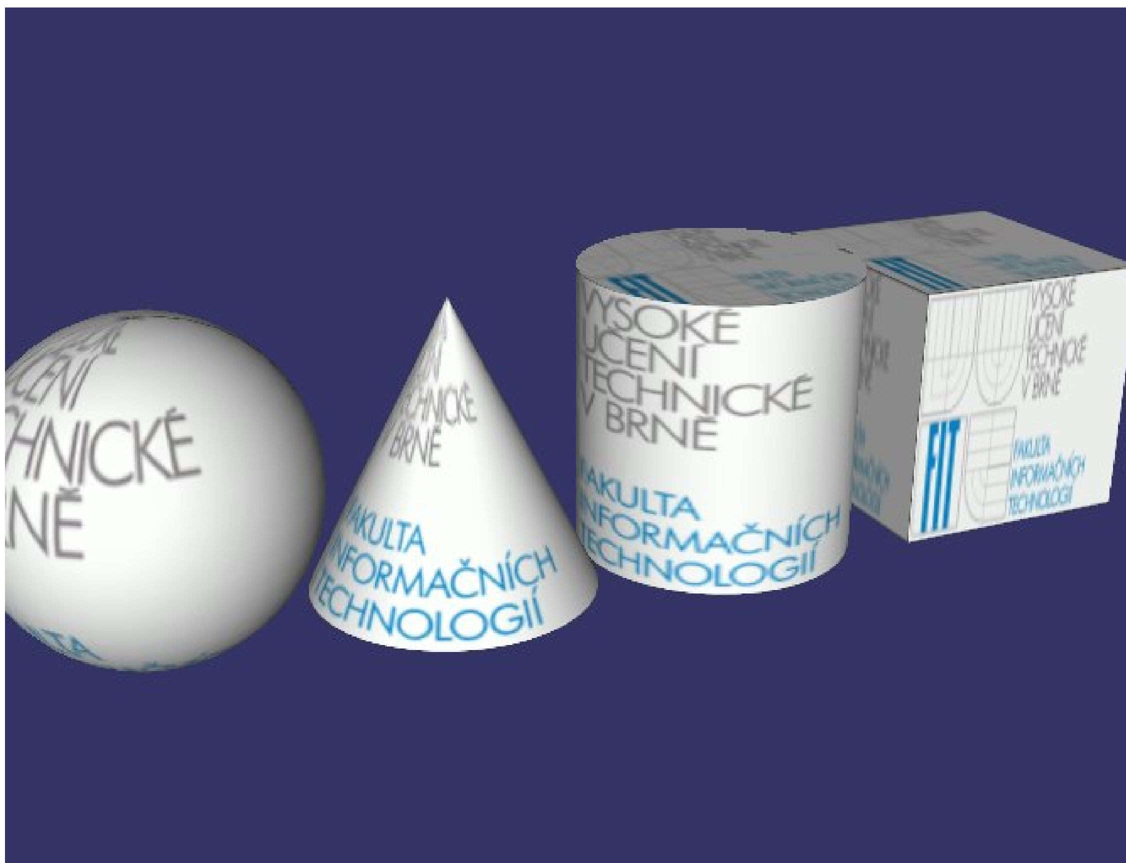
```
// vytvor stateset z geody
osg::StateSet* stateset = geode->getOrCreateStateSet();
```

```
// prirad mu texturu
stateset->setTextureAttributeAndModes(0,texture,osg::StateAttribute::ON);
```

Teraz už ďalej postupujeme rovnako ako v prvom deme – listový uzol priradíme ako potomka kmeňovému uzlu `rootNode` a ten nastavíme na zobrazenie.

```
osg::Node *listovy_uzol = geode; // listovy uzol bude obsahovat geometriu z geode
rootNode->addChild(listovy_uzol); // pridaj kmenovemu uzlu listovy uzol ako potomka

viewer.setSceneData((osg::Node *)rootNode); // na zobrazenie nastav kmenovy uzol
```

Ak všetko funguje správne, výsledok by mal vyzerať ako na nasledujúcom obrázku.



Obr. 7: Pridanie textúry geometrickým objektom

7.3 Vytvorenie animačnej trasy

Teraz si ukážeme, ako vytvoriť jednoduchú animačnú trasu po kružnici a priradiť ju objektu.

Nasledujúca funkcia vracia animačnú trasu typu *osg::AnimationPath** po kružnici vrátane rotácie objektu, ktorá spraví dojem letu. Prvý treba vytvoriť objekt

osg::AnimationPath, ktorý bude funkcia vracať a nastaviť jeho opakovanie na nekonečnú slučku funkciou *setLoopMode()*.

```
osg::AnimationPath* vytvorAnimacnuTrasu(const osg::Vec3& center,float radius,double
looptime)
{
    // nastav trasu na animaciu
    osg::AnimationPath* animationPath = new osg::AnimationPath;
    animationPath->setLoopMode(osg::AnimationPath::LOOP);
```

Nasleduje cyklus, v ktorom budeme v každom kroku vkladať aktuálnu pozíciu a rotáciu telesa. Všimnime si, že do animačnej trasy sa okrem času vkladajú „control pointy", teda ovládacie body (typu `osg::AnimationPath::ControlPoint`) s dvomi vstupnými hodnotami, pozíciou, ktorá je typu *osg::Vec3* a rotáciou, ktorá je typu *osg::Quat*.

```
    int numSamples = 40;
    float yaw = 0.0f;
    float yaw_delta = 2.0f*osg::PI/((float)numSamples-1.0f);
    float roll = osg::inDegrees(30.0f);

    double time=0.0f;
    double time_delta = looptime/(double)numSamples;

    for(int i=0;i<numSamples;++i)
    {

    osg::Vec3 position(center+ osg::Vec3(sinf(yaw)*radius,cosf(yaw)*radius,0.0f));
    osg::Quat rotation (osg::Quat(roll,osg::Vec3(0.0,1.0,0.0))*osg::Quat(-
(yaw+osg::inDegrees(90.0f)),osg::Vec3(0.0,0.0,1.0)));

  // vloz control point do animacnej trasy
  animationPath->insert(time,osg::AnimationPath::ControlPoint(position,rotation));

        yaw += yaw_delta;
        time += time_delta;
    }
    return animationPath;
}
```

Teraz môžeme vytvoriť animovaný model, pričom vracať budeme *osg::MatrixTransform*, ktorý je tiež zdedený od typu osg::Node. Funkciou `vytvorAnimacnuTrasu` vytvoríme animačnú trasu so zadaným stredom, polomerom a časom, a funkciou `osgDB::readNodeFile` načítame do uzlu `glider` 3D model závesného lietadla.

```
osg::Node* vytvorLetiaciModel(const osg::Vec3& center, float radius) //
vytvor uzol s lietajucim zavesnym lietadlom
{
    float animationLength = 10.0f;

    osg::AnimationPath* animationPath =
vytvorAnimacnuTrasu(center,radius,animationLength);

    osg::Node* glider = osgDB::readNodeFile("glider.osg");
```

Teraz vytvoríme dva `osg::MatrixTransform*`. Ten prvý s názvom `positioned` poslúži na správne umiestnenie telesa do grafu scény. Druhý s názvom `xform` je určený na samotnú animáciu – jeho update callback nastavíme na našu vytvorenú trasu, čím spôsobíme, že každým aktualizačným priechodom grafu scény sa naše závesné lietadlo posunie o jeden control point dopredu. Na zistenie hraničného polomeru lietadla, tzv. BoundingSphere použijeme funkciu `glider->getBound()`. MatrixTransform-u `positioned` pridáme ako potomka naše závesné lietadlo a MatrixTransform-u `xform` pridáme ako potomka `positioned`.

```
const osg::BoundingSphere& bs = glider->getBound();

float size = radius/bs.radius()*0.3f;
osg::MatrixTransform* positioned = new osg::MatrixTransform;
positioned->setMatrix(osg::Matrix::translate(-bs.center())*
                      osg::Matrix::scale(size,size,size)*
                      osg::Matrix::rotate(osg::inDegrees(-90.0f), 0.0f,0.0f,1.0f));

        positioned->addChild(glider);

        osg::MatrixTransform* xform = new osg::MatrixTransform;
        xform->setUpdateCallback(new
osg::AnimationPathCallback(animationPath,0.0f,2.0));
        xform->addChild(positioned);

    return xform;
}
```

Potrebujeme už len pridať kmeňovému uzlu ako potomka naše lietadlo, aby sa nám zobrazilo v scéne.

```
// pridaj animovane zavesne lietadlo
rootNode->addChild(vytvorLetiaciModel(osg::Vec3(0.0f,0.0f,0.0f), 7.0f));
```

Ak všetko funguje správne, výsledok by mal vyzerať približne ako na nasledujúcom obrázku:

Obr. 8: Vytvorenie animovaného závesného lietadla

7.4 Vytváranie vlastných geometrických telies

Teraz si ukážeme, ako vytvoriť geometrický útvar pyramídy priamo vkladaním priestorových bodov v programe a nafarbením jeho vertexov každý na inú farbu. Najprv musíme vytvoriť prehliadač `viewer`, kmeňový skupinový uzol `root`, geodu pre pyramídu `pyramidGeode` a uzol typu *osg::Geomerty*, v ktorom bude uložená samotná geometria pyramídy.

```
osgProducer::Viewer viewer;
osg::Group* root = new osg::Group();
osg::Geode* pyramidGeode = new osg::Geode();
osg::Geometry* pyramidGeometry = new osg::Geometry();
```

Geometrický uzol pyramídy priradíme geode pyramídy a tú priradíme ako potomka kmeňovému uzlu.

```
   pyramidGeode->addDrawable(pyramidGeometry);
   root->addChild(pyramidGeode);
```

Teraz deklarujeme pole priestorových bodov. Každý priestorový bod bude reprezentovaný trojicou – inštanciou triedy *Vec3*. Na uloženie týchto trojíc bude použitá trieda *osg::Vec3Array*. Na vkladanie prvkov môžeme použiť metódu push_back, pretože *osg::Vec3Array* je odvodený od štandardného typu *std::vector*. Z-ovou osou smerom hore vložíme 5 bodov potrebných na vytvorenie pyramídy, pričom prvý bude mať index 0 a posledný piaty index 4.

```
osg::Vec3Array* pyramidVertices = new osg::Vec3Array;
   pyramidVertices->push_back( osg::Vec3( 0, 0, 0) ); // vpedu vlavo
   pyramidVertices->push_back( osg::Vec3(10, 0, 0) ); // vpredu vpravo
   pyramidVertices->push_back( osg::Vec3(10,10, 0) ); // vzadu vpravo
   pyramidVertices->push_back( osg::Vec3( 0,10, 0) ); // vzadu vlavo
   pyramidVertices->push_back( osg::Vec3( 5, 5,10) ); // vrchol
```

Priradíme túto skupinu bodov geometrickému uzlu `pyramidGeometry` typu *osg::Geometry*, ktorý sme pridali do scény.

```
pyramidGeometry->setVertexArray( pyramidVertices );
```

Ďalším krokom je vytvorenie PrimitiveSetu a jeho pridanie geometrii pyramídy. Použitím prvých štyroch bodov vytvoríme základňu. Na to nám poslúži trieda `osg::DrawElementsUInt`. Táto trieda je tiež odvodená od typu *std::vector*, takže na vkladanie bodov použijeme metódu push_back. Na zaistenie správnej viditeľnosti musia byť body polygónu vkladané v poradí proti smeru hodinových ručičiek.

```
osg::DrawElementsUInt* pyramidBase =
      new osg::DrawElementsUInt(osg::PrimitiveSet::QUADS, 0);
pyramidBase->push_back(3);
pyramidBase->push_back(2);
pyramidBase->push_back(1);
pyramidBase->push_back(0);
pyramidGeometry->addPrimitiveSet(pyramidBase);
```

Zopakujeme rovnaký postup pre všetky 4 steny pyramídy.

```
osg::DrawElementsUInt* pyramidFaceOne =
      new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
pyramidFaceOne->push_back(0);
pyramidFaceOne->push_back(1);
pyramidFaceOne->push_back(4);
pyramidGeometry->addPrimitiveSet(pyramidFaceOne);

osg::DrawElementsUInt* pyramidFaceTwo =
      new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
```

```
pyramidFaceTwo->push_back(1);
pyramidFaceTwo->push_back(2);
pyramidFaceTwo->push_back(4);
pyramidGeometry->addPrimitiveSet(pyramidFaceTwo);

osg::DrawElementsUInt* pyramidFaceThree =
      new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
pyramidFaceThree->push_back(2);
pyramidFaceThree->push_back(3);
pyramidFaceThree->push_back(4);
pyramidGeometry->addPrimitiveSet(pyramidFaceThree);
osg::DrawElementsUInt* pyramidFaceFour =
      new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
pyramidFaceFour->push_back(3);
pyramidFaceFour->push_back(0);
pyramidFaceFour->push_back(4);
pyramidGeometry->addPrimitiveSet(pyramidFaceFour);
```

Deklaruj pole prvkov *Vec4* do poľa `colors` na uloženie farieb.

```
osg::Vec4Array* colors = new osg::Vec4Array;
colors->push_back(osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f) ); //index 0 cervena
colors->push_back(osg::Vec4(0.0f, 1.0f, 0.0f, 1.0f) ); //index 1 zelena
colors->push_back(osg::Vec4(0.0f, 0.0f, 1.0f, 1.0f) ); //index 2 modra
colors->push_back(osg::Vec4(1.0f, 1.0f, 1.0f, 1.0f) ); //index 3 biela
```

Teraz treba deklarovať vektor, ktorý priradí farbu bodovým elementom. Tento vektor by mal mať rovnaký počet prvkov, aký je počet priestorových bodov a slúži ako spojovateľ medzi poľom bodov a poľom farieb. V tomto prípade priraďujeme 5 priestorových bodov len štyrom farbám, pretože prvky 0 (dole vľavo) a 4 (vrchol) sú obe priradené farbe s indexom 0 (červená farba).

```
osg::TemplateIndexArray
  <unsigned int, osg::Array::UIntArrayType,4,4> *colorIndexArray;
colorIndexArray =
  new osg::TemplateIndexArray<unsigned int,osg::Array::UIntArrayType,4,4>;
colorIndexArray->push_back(0); // bod 0 priradeny farbe 0
colorIndexArray->push_back(1); // bod 1 priradeny farbe 1
colorIndexArray->push_back(2); // bod 2 priradeny farbe 2
colorIndexArray->push_back(3); // bod 3 priradeny farbe 3
colorIndexArray->push_back(0); // bod 4 priradeny farbe 0
```

Ďalším krokom je priradenie poľa farieb geometrii a priradenie indexov farieb, ktoré sme zadefinovali vyššie. Taktiež treba nastaviť farebné spájanie na _PER_VERTEX, teda na viazanie na body.

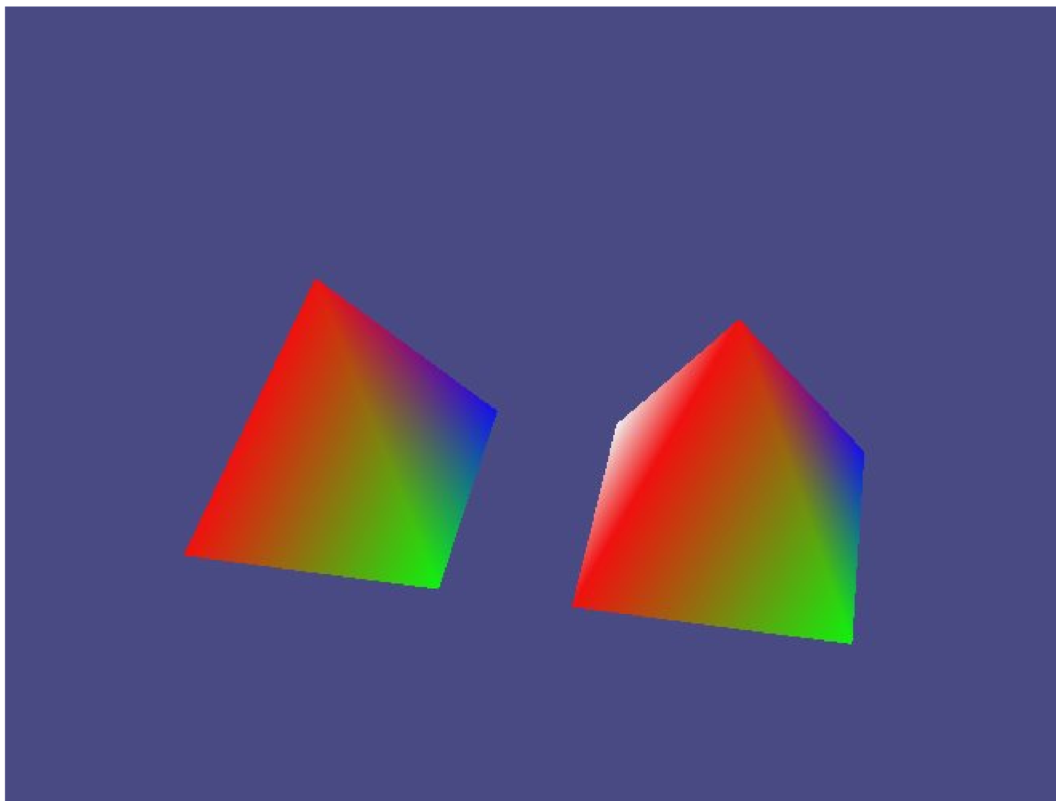
```
pyramidGeometry->setColorArray(colors);
pyramidGeometry->setColorIndices(colorIndexArray);
pyramidGeometry->setColorBinding(osg::Geometry::BIND_PER_VERTEX);
```

Teraz keď už máme vytvorený uzol geometrie `pyramidGeometry` a je pridaný do scény, môžeme túto geometriu znovu použiť ľubovolný počet krát. Napríklad, keby sme ju chceli mať vytvorenú druhý krát a posunutú o 15 jednotiek doprava, pridali by sme jej geodu

transformačnému uzlu *osg::PositionAttitudeTransform*, ktorý by ju posunul. Vytvorme teda transformačný uzol `druhaPyramida`, ktorému pridáme ako potomka geodu pyramídy a funkciou `setPosition()` ho posunieme o 15 jednotiek doprava. Tento uzol nakoniec pridáme kmeňovému uzlu a ten nastavíme na zobrazovanie

```
osg::PositionAttitudeTransform* druhaPyramida =
      new osg::PositionAttitudeTransform();
druhaPyramida->addChild(pyramidGeode);
druhaPyramida->setPosition(osg::Vec3(15,0,0));
root->addChild(druhaPyramida);

viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
viewer.setSceneData( root );
```

Výsledok by mal vyzerať približne ako na nasledujúcom obrázku:



Obr. 9: Ukážka vytvorenia geometrických telies

7.5 Uloženie grafu scény do súboru

Ľubovolný uzol grafu scény môžeme uložiť do natívneho textového .osg súboru, ktorý podporuje ukladanie všetkých typov uzlov v OSG. Napríklad keby sme chceli naše dve pyramídy uložiť do súboru, použijeme nasledujúci príkaz:

```
osgDB::writeNodeFile(*root, "pyramidy.osg");
```

Týmto sa uloží kmeňový skupinový uzol `root` do .osg súboru, ktorý môžeme na inom mieste znova načítať. Použili by sme na to tento príkaz:

```
osg::Node* root = osgDB::readNodeFile("pyramidy.osg");
```

7.6 Implementácia kamery ktorá sleduje uzol

Teraz si ukážeme, ako ľahko naprogramovať kameru, ktorá by bola pripojená k pohyblivému uzlu a sledovala by ho. Postup, ktorý tu bude vysvetlený je založený na obnovovaní matice ktorá umiestňuje kameru na základe súradníc uzla v grafe scény. Touto metódou môže byť uzlu priradená kamera ktorá bude uzol sledovať. Na získanie súradníc určitého uzla implementujeme triedu, ktorá využíva trasu „navštevovateľa“ uzlov. Táto trieda nám umožní priradiť samu seba uzlu a získať súradnice tohto uzla vzhľadom na okolný svet (v podobe matice). Táto matica (ako reprezentácia súradnice ľubovoľného uzlu v grafe) bude použitá inštanciou triedy *osgGA::MatrixManipulator* na umiestnenie kamery.

Najprv vytvoríme triedu, ktorá bude poskytovať nahromadenie všetkých transformačných uzlov nad ním. Nie je to také zložité, pretože každý node visitor („navštevovateľ“) má prístup k ceste ku kmeňovému uzlu v hierarchii. Ak máme túto cestu k dispozícii, môžeme použiť funkciu OSG *computeWorldToLocal(osg::NodePath)* na získanie matice, ktorá bude reprezentovať súradnice daného uzla.

Jadro tejto triedy bude update callback ktorý bude aktualizovať maticu, ktorá reprezentuje nahromadenie všetkých matíc nad daným uzlom. Celá trieda bude vyzerať asi takto

```
// tato trieda umozni pristup k matici, ktora reprezentuje
// nahromadenie vsetkych transformacnych matic v specifikovanom uzle grafu
struct updateAccumulatedMatrix : public osg::NodeCallback
{
   virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
   {
      matrix = osg::computeWorldToLocal(nv->getNodePath() );
      traverse(node,nv);
   }
   osg::Matrix matrix;
};
```

Ďalej musíme zaistiť aby sa tento callback spúšťal počas aktualizačného priechodu grafom scény. Na to musíme vytvoriť triedu, ktorá obsahuje uzol *osg::Node*. Zaistíme, aby update callback tohto uzla bola naša funkcia `updateAccumulatedMatrix`. Na umožnenie prístupu k matici reprezentujúcej súradnice uzla ku ktorému sú inštancie pripojené, potrebujeme pre našu maticu poskytnúť „get“ metódu. Deklarácia triedy bude nasledujúca:

```
struct transformAccumulator
{
   transformAccumulator();
   bool attachToGroup(osg::Group* g);
   osg::Matrix getMatrix();
protected:
   osg::ref_ptr<osg::Group> parent;
   osg::Node* node;
   updateAccumulatedMatrix* mpcb;
};
```

A implementácia bude vyzerať takto:

```
osg::Matrix transformAccumulator::getMatrix()
{
   return mpcb->matrix;
}
transformAccumulator::transformAccumulator()
{
   parent = NULL;
   node = new  osg::Node;
   mpcb = new updateAccumulatedMatrix();
   node->setUpdateCallback(mpcb);
}

bool transformAccumulator::attachToGroup(osg::Group* g)
{
   bool success = false;
   if (parent != NULL)
   {
      int n = parent->getNumChildren();
      for (int i = 0; i < n; i++)
      {
         if (node == parent->getChild(i) )
         {
            parent->removeChild(i,1);
            success = true;
         }
      }
      if (! success)
      {
         return success;
      }
   }
   g->addChild(node);
   return true;
}
```

Teraz keď máme triedu, ktorá poskytuje maticu ktorá reprezentuje súradnice uzla v scéne, použijeme túto maticu na umiestnenie kamery. Zdedíme triedu od *osgGA::MatrixManipulator* ktorá použije maticovú reprezentáciu súradníc na umiestnenie

kamery. Na to aby sme toto dosiahli, bude naša trieda potrebovať dátový člen, ktorý bude spracovávať inštanciu našej triedy `transformAccumulator`, ktorú sme vytvorili vyššie. Trieda bude taktiež potrebovať člena na uloženie matice na umiestnenie kamery.

Jadro triedy `followNodeMatrixManipulator` je metóda `handle`. Táto metóda kontroluje jednotlivé GUI udalosti a odpovedá na ne. Pre našu triedu, jediná GUI udalosť na ktorú budeme odpovedať je „FRAME“. Pri každom snímku musíme uložiť kópiu našej matice, ktorú použijeme na umiestnenie kamery. Na základe tohto bude deklarácia našej triedy vyzerať takto:

```
class followNodeMatrixManipulator : public osgGA::MatrixManipulator
{
public:
   followNodeMatrixManipulator( transformAccumulator* ta)
{worldCoordinatesOfNode = ta; theMatrix = osg::Matrixd::identity();}
   bool handle (const osgGA::GUIEventAdapter&ea,
osgGA::GUIActionAdapter&aa);
   void updateTheMatrix();
   virtual void setByMatrix(const osg::Matrixd& mat);
   virtual void setByInverseMatrix(const osg::Matrixd&mat);
   virtual osg::Matrixd getInverseMatrix() const;
   virtual osg::Matrixd getMatrix() const;
protected:
   ~followNodeMatrixManipulator() {}
   transformAccumulator* worldCoordinatesOfNode;
   osg::Matrixd theMatrix;
};
```

Implementácia triedy bude vyzerať nasledujúco:

```
void followNodeMatrixManipulator::setByMatrix(const osg::Matrixd& mat)
{
   theMatrix = mat;
}
void followNodeMatrixManipulator::setByInverseMatrix(const osg::Matrixd&
mat)
{
   theMatrix = mat.inverse(mat);
}
void followNodeMatrixManipulator::updateTheMatrix()
{
   theMatrix = worldCoordinatesOfNode->getMatrix();
}
osg::Matrixd followNodeMatrixManipulator::getInverseMatrix() const
{
   osg::Matrixd m;
   m = theMatrix * osg::Matrixd::rotate(-M_PI/2.0, osg::Vec3(1,0,0) );
   return m;
}

osg::Matrixd followNodeMatrixManipulator::getMatrix() const
{
   return theMatrix;
}
```

```
bool followNodeMatrixManipulator::handle(const osgGA::GUIEventAdapter&ea,
osgGA::GUIActionAdapter&aa)
{
   switch(ea.getEventType())
   {
   case (osgGA::GUIEventAdapter::FRAME):
      {
         updateTheMatrix();
         return false;
      }
   }
   return false;
}
```

Teraz už len stačí zadefinovať inštanciu triedy `transformAccumulator`. Táto inštancia bude prichytená k niektorému z uzlov v grafe scény. Ďalej potrebujeme zadefinovať inštanciu triedy `followNodeMatrixManipulator`, pričom konštruktor tohto manipulátoru si berie ako parameter ukazateľ na inštanciu našej triedy `transformAccumulator`. Posledným krokom je pridanie tohto manipulátora do zoznamu manipulátorov viewera. Toto dosiahneme nasledujúcimi príkazmi:

```
transformAccumulator* tankWorldCoords = new transformAccumulator();
tankWorldCoords->attachToGroup(followerPAT);
followNodeMatrixManipulator* followTank = new
followNodeMatrixManipulator(tankWorldCoords);
osgGA::KeySwitchMatrixManipulator *ksmm =
viewer.getKeySwitchMatrixManipulator();
   if (!ksmm)
      return -1;
// pridaj ako manipulator transformacny uzol ktory sleduje tank.
// prirad tlacitku 'm' funkciu prepnutia prehliadania na tento manipulator
   ksmm->addMatrixManipulator('m',"tankFollower",followTank);
   ksmm->selectMatrixManipulator(ksmm->getNumMatrixManipulators()-1);
```

Výsledok tohto demopríkladu vyzerá nasledovne:

Obr. 10: Implementácia kamery ktorá sleduje uzol

7.7 Výber objektov myšou

V tomto demopríklade ukážem, ako spracovať v OSG výber objektov myšou. Podobne ako v predchádzajúcom príklade, na spracovanie udalostí použijeme zdedenú triedu, v tomto prípade budeme dediť od triedy *osgGA::GUIEventHandler* a reagovať budeme znova na udalosť „FRAME", pri ktorej vykonáme metódu ktorá spracuje výber myšou a vypíše, aký objekt bol vybraný. Metóda sa bude volať `PickHandler` a jej jadrom bude metóda `handle`, ktorá bude spracuvávať udalosti a reagovať na „FRAME". Definícia tredy bude vyzerať takto:

```
// trieda ktora spracuje vyber mysou
class PickHandler : public osgGA::GUIEventHandler {
public:

    PickHandler(osgProducer::Viewer* viewer,osgText::Text* updateText):
        _viewer(viewer),
        _updateText(updateText) {}
```

```
    ~PickHandler() {}

    bool handle(const osgGA::GUIEventAdapter& ea,osgGA::GUIActionAdapter& us);

    virtual void pick(const osgGA::GUIEventAdapter& ea);

    void setLabel(const std::string& name)
    {
        if (_updateText.get()) _updateText->setText(name);
    }

protected:

    osgProducer::Viewer* _viewer;
    osg::ref_ptr<osgText::Text>  _updateText;
};
```

Implementácia metódy handle, ktorá bude reagovať na „FRAME" bude nasledujúca:

```
bool PickHandler::handle(const osgGA::GUIEventAdapter& ea,osgGA::GUIActionAdapter&)
{
    switch(ea.getEventType())
    {
    case(osgGA::GUIEventAdapter::FRAME):
        {
            pick(ea);
        }
        return false;

    default:
        return false;
    }
}
```

V implementácii metódy `pick` musíme najprv zadefinovať inštanciu triedy osgUtil::IntersectVisitor::HitList, do ktorej sa uložia geody, ktoré sa pretnú s kurzorom myši. Toto dosiahneme pomocou metódy `computeIntersections` s parametrami x-ovej a y-ovej súradnici kurzora myši a našou inštanciou triedy osgUtil::IntersectVisitor::HitList. Ďalej si zadefinujeme reťazec `gdlist`, do ktorého budeme ukladať informácie o geodách, nad ktorými sa práve nachádza kurzor myši.

```
void PickHandler::pick(const osgGA::GUIEventAdapter& ea)
{
    osgUtil::IntersectVisitor::HitList hlist;

    std::string gdlist="";
    if (_viewer->computeIntersections(ea.getX(),ea.getY(),hlist))
    {
        for(osgUtil::IntersectVisitor::HitList::iterator hitr=hlist.begin();
            hitr!=hlist.end();
            ++hitr)
        {
            std::ostringstream os;
            if (hitr->_geode.valid() && !hitr->_geode->getName().empty())
            {
                // geody su identifikovane menom
                os<<"Objekt \""<<hitr->_geode->getName()<<"\""<<std::endl;
            }
            else if (hitr->_drawable.valid())
```

```
            {
                os<<"Objekt \""<<hitr->_drawable->className()<<"\""<<std::endl;
            }

            os<<"        lokalne suradnice("<< hitr-
>getLocalIntersectPoint()<<")"<<"  normala("<<hitr-
>getLocalIntersectNormal()<<")"<<std::endl;
            os<<"        globalne suradnice("<< hitr-
>getWorldIntersectPoint()<<")"<<"  normala("<<hitr-
>getWorldIntersectNormal()<<")"<<std::endl;
            osgUtil::Hit::VecIndexList& vil = hitr->_vecIndexList;
            for(unsigned int i=0;i<vil.size();++i)
            {
                os<<"        "<<i<<". index bodu = "<<vil[i]<<std::endl;
            }

            gdlist += os.str();
        }
    }
    setLabel(gdlist);
}
```

Teraz ešte pridáme kód na vytvorenie siedmych obdĺžnikov typu GL_QUAD, každý vo vlastnej geode indentifikovaný špecifickým menom. Dajú sa použiť napríklad na vytvorenie menu. Kód vyzerá nasledujúco:

```
    for (int i=0; i<7; i++) {
        osg::Vec3 dy(0.0f,-30.0f,0.0f);
        osg::Vec3 dx(120.0f,0.0f,0.0f);
        osg::Geode* geode = new osg::Geode();
        osg::StateSet* stateset = geode->getOrCreateStateSet();
            const char *opts[]={"Prva", "Druha", "Tretia", "Stvrta", "Piata",
"Siesta", "Siedma"};
        osg::Geometry *quad=new osg::Geometry;
        stateset->setMode(GL_LIGHTING,osg::StateAttribute::OFF);
        stateset->setMode(GL_DEPTH_TEST,osg::StateAttribute::OFF);
            std::string name="Volba";
        name += " ";
        name += std::string(opts[i]);
            geode->setName(name);
        osg::Vec3Array* vertices = new osg::Vec3Array(4); // 1 quad
        osg::Vec4Array* colors = new osg::Vec4Array;
        colors = new osg::Vec4Array;
        colors->push_back(osg::Vec4(0.8-0.1*i,0.1*i,0.2*i, 1.0));
        quad->setColorArray(colors);
        quad->setColorBinding(osg::Geometry::BIND_PER_PRIMITIVE);
        (*vertices)[0]=position;
        (*vertices)[1]=position+dx;
        (*vertices)[2]=position+dx+dy;
        (*vertices)[3]=position+dy;
        quad->setVertexArray(vertices);
        quad->addPrimitiveSet(new osg::DrawArrays(osg::PrimitiveSet::QUADS,0,4));
        geode->addDrawable(quad);
        hudCamera->addChild(geode);

        position += delta;
    }
```

Konečný výsledok demopríkladu vyzerá nasledujúco:

Obr. 11: Výber objektov myšou

8 Záver

V mojej práci som sa zaoberal knižnicou OpenSceneGraph. Dielo, ktoré sa pôvodne vyvinulo z koníčka Dona Burnsa sa postupom času vypracovalo na všestranné open-sourcové riešenie 3D grafických požiadaviek programátora.

Vzhľadom na jeho open-sourcový charakter je zdrojový kód dostupný pre každého voľne na webe. V priebehu rokov vďaka tomu za prispenia mnohých autorov postupne dozrieva do podoby profesionálneho nástroja, ktorý odbremeňuje programátora od programovania nízkoúrovňových OpenGL volaní a poskytuje priestorovú organizáciu a funkčnosť 3D aplikáciám. Zároveň s komplexným riešením grafickej scény významne odľahčuje nároky na hardvér. Jeho najväčšou výhodou je organizovanie priestorových geometrických dát pre ich efektívne zobrazovanie a zjednodušenie správy zobrazovacích stavov.

Môj vlastný prínos do tejto práce je vytvorenie demopríkladov a ich spracovanie do formy tutoriálov, ktoré som uviedol v tejto práci, čo môže byť prínosné pre ľudí ktorí by s knižnicou OpenSceneGraph chceli začať pracovať. Všetky demopríklady uvedené v tejto práci sa nachádzajú aj na priloženom disku DVD, vrátane skompilovaného jadra OSG, priložených originálnych príkladov, aplikácií a zdrojových kódov dodávaných s distribúciou OSG.

Ďalším pokračovaním tohto projektu by mohlo byť napríklad vytvorenie hry za použitia OSG, či už leteckého simulátora alebo napríklad automobilových pretekov.

Iné praktické využitie OSG vidím napríklad vo vedeckých simuláciách rôznych procesov, vojenských simuláciách, nesporné je taktiež využitie v priemysle a vo výrobe. Možné je tiež vytvorenie projektu virtuálnej reality so skĺbením funkcií OSG a špeciálneho hardvéru.

Literatúra

[1] Skew Matrix Software, OpenSceneGraph Quick Start Guide, 2007
[2] www.openscenegraph.com
[3] http://www.nps.navy.mil/cs/sullivan/osgtutorials/
[4] http://www.devmaster.net/engines/engine_details.php?id=58
[5] http://www.3drealtimesimulation.com/osg/osg_faq_1.htm

Zoznam príloh

Príloha 1. DVD so skompilovaným OSG, uvedenými demopríkladmi vrátane všetkých zdrojových kódov