VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

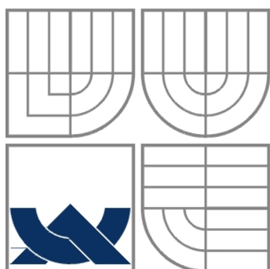
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SADA JEDNODUCHÝCH 3D HER

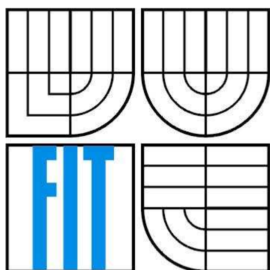
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE MICHAL VRÁNA
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

SADA JEDNODUCHÝCH 3D HER

SET OF SIMPLE 3D GAMES

BAKALÁŘSKÁ PRÁCE

BACHELOR´S THESIS

AUTOR PRÁCE — MICHAL VRÁNA

AUTHOR

VEDOUCÍ PRÁCE — ING. JAN PEČIVA

SUPERVISOR

BRNO 2007

Zadání bakalářské práce/402/2006/xvrana13

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Vrána Michal**
Obor: Informační technologie
Téma: **Sada jednoduchých 3D her**
Kategorie: Počítačová grafika

Pokyny:

1. Nastudujte si grafickou knihovnu Open Scene Graph a seznamte se s jejími základními koncepty.
2. Vytvořte sadu jednoduchých demonstračních aplikací používajících Open Scene Graph.
3. Vytvořte tutoriál k těmto aplikacím a publikujte je na internetu.
4. Na základě předchozích zkušeností s knihovnou Open Scene Graph navrhněte a implementujte rozsáhlou grafickou aplikaci dle domluvy s vedoucím, která bude využívat zmíněnou knihovnu a bude demonstrovat její možnosti.
5. Vyhodnoťte zkušenosti s knihovnou Open Scene Graph a její použitelnost v praxi.

Literatura:

- OpenSceneGraph dokumentace na http://www.openscenegraph.org/

Při obhajobě semestrální části projektu je požadováno:

- Vytvořit tutoriál a pokusit se ho publikovat.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Pečiva Jan, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Pavel Zemčík
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Michal Vrána**
Id studenta: 89007
Bytem: Kunčice nad Labem 118, 543 61 Kunčice nad Labem
Narozen: 10. 08. 1984, Jilemnice
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

....

(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP: Sada jednoduchých 3D her
Vedoucí/školitel VŠKP: Pečiva Jan, Ing.
Ústav: Ústav počítačové grafiky a multimédií
Datum obhajoby VŠKP: ....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

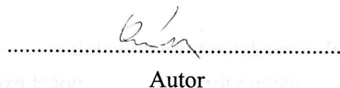
1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☒ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: ....



....

Nabyvatel Autor

Abstrakt

Předmětem tohoto bakalářského projektu je grafická knihovna Open Scene Graph a její použitelnost v praxi. Konkrétním cílem bylo vytvoření série demonstračních aplikací, tutoriálů k nim a jedné počítačové hry testující možnosti Open Scene Graph. Text této technické zprávy obsahuje popis Open Scene Graph, její srovnání s dalšími knihovnami pro práci s počítačovou grafikou, popis některých technik použitých v tutoriálech a v počítačové hře, a popis implementace.

Klíčová slova

Open Scene Graph, Počítačové hry, 3D grafika, Síťová hra, Detekce kolizí

Abstract

The objective of this thesis is Open Scene Graph graphic library and its usability in standard practice. More specifically, the goal of this project was creating set of demo applications, tutorials to them and one computer game testing possibilities of Open Scene Graph. Text of this technical report includes description of Open Scene Graph, its comparison to other graphic libraries, description of some techniques used in tutorials and in game, and description of implementation.

Keywords

Open Scene Graph, Computer games, 3D graphics, Multiplayer, Collision detection

Citace

Michal Vrána: Sada jednoduchých 3D her, bakalářská práce, Brno, FIT VUT v Brně, 2007

Sada jednoduchých 3D her

Prohlášení

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

……………………
Michal Vrána
14. 5. 2007

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu Janu Pečivovi za ochotu při konzultacích a za pomoc při snaze publikovat tutoriály na webu. Dále bych chtěl poděkovat Petru Valinovi za pomoc při testování síťové hry a za propůjčení hardwarových prostředků v době její tvorby.

© Michal Vrána, 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 ... 3
1 Úvod ... 6
2 Teorie ... 7
2.1 Základní informace o Open Scene Graphu ... 7
2.2 Scene graph ... 7
2.3 Součásti Open Scene Graphu ... 8
2.3.1 Producer ... 8
2.3.2 Open Threads ... 8
2.3.3 OsgDB ... 8
2.3.4 OsgGA ... 9
2.3.5 OsgUtil ... 9
2.3.6 OsgText ... 9
2.3.7 OsgTerrain ... 9
2.3.8 OsgParticle ... 9
2.4 Alternativní nástroje ... 10
2.4.1 Open Inventor ... 10
2.4.2 OpenSG ... 10
2.4.3 GLUT ... 11
2.4.4 OGRE 3D ... 11
2.4.5 Unreal Engine 3 ... 11
3 Grafické prvky ... 12
3.1 Grafické elementy v programových částech ... 12
3.1.1 Základní geometrie ... 12
3.1.2 Pokročilá geometrie ... 12
3.1.3 Nastavování stavů ... 13
3.1.4 Skybox ... 13
3.1.5 Billboardy ... 14
3.1.6 Stíny ... 15
3.1.7 Level of Detail ... 17
3.1.8 Vesmírný prach ... 18
3.1.9 Částicové systémy ... 18
3.2 Ostatní grafické prvky ... 19
3.2.1 Shadery ... 19
3.2.2 Terén ... 20

4 Implementace ... 22
4.1 Tutoriály ... 22
4.2 Počítačová hra ... 23
4.2.1 Základní koncept ... 24
4.2.2 Nastavení grafiky ... 24
4.2.3 Menu ... 25
4.2.4 Nahrávací obrazovka a práce s pamětí ... 25
4.2.5 Scéna ... 26
4.2.6 Planety ... 26
4.2.7 Asteroidy ... 27
4.2.8 Hvězdy ... 28
4.2.9 Obloha ... 28
4.2.10 Brány ... 29
4.2.11 Loď ... 29
4.2.12 Kamery ... 30
4.2.13 Vesmírný prach ... 30
4.2.14 HUD ... 30
4.2.15 Detekce kolizí ... 31
4.2.16 Síťová hra ... 32
5 Možnosti vylepšení ... 33
5.1 Tutoriály ... 33
5.2 Porovnání s ostatními knihovnami ... 33
5.3 Rozšíření počítačové hry ... 33
5.3.1 Grafika ... 33
5.3.2 Uživatelské rozhraní ... 34
5.3.3 Zvuky ... 34
5.3.4 Síťová hra ... 34
6 Závěr ... 35
Literatura ... 36
Příloha A: Grafické rozhraní ... 37
Příloha B: Tutoriály ... 38
B.1 Načítání modelu ... 38
B.2 Transformace a jednoduché tvary ... 41
B.3 Geometrie a Textury ... 43
B.4 Stavy ... 47
B.5 Text ... 48
B.6 Vstupy z klávesnice ... 50

B.7 Skybox 53
B.8 Částicové efekty 57
B.9 Billboardy 60
B.10 Shadow Texture 63
Seznam příloh 69

1 Úvod

V současné době představují počítačové hry spolu s databázovými aplikacemi největší hybnou sílu pro rozvoj výpočetní techniky. Spolu s jejich oblíbeností roste i jejich množství na trhu a s tím i konkurenční boj vyžadující stále rozsáhlejší, složitější a samozřejmě graficky vyspělejší tituly. Aby se mohla pozornost vývojářů zaměřit na to podstatné, tj. hra, jsou vyvíjeny stále nové knihovny a nástroje usnadňující tvorbu takovýchto her. Tato potřeba se samozřejmě netýká pouze počítačových her, ale počítačové grafiky obecně. Jednou z takovýchto knihoven je knihovna Open Scene Graph, které by se měla věnovat podstatná část této práce.

Cílem této práce bylo prozkoumání možností knihovny Open Scene Graph a zjištění zda je tato knihovna použitelná v praxi. K tomuto účelu jsem vytvořil sérii demonstračních aplikací a jednu počítačovou hru, což mě pomohlo získat zkušenosti s touto knihovnou. Kapitola 2 pojednává o knihovně Open Scene Graph a porovnává ji s jinými volně dostupnými i placenými nástroji. V kapitole 3 jsou rozebrány některé grafické prvky použité v tutoriálech a ve hře, a kapitola 4 popisuje implementaci obou programových částí tohoto projektu, tj. hra a série tutoriálů. Kapitola 5 obsahuje některé návrhy na vylepšení programových částí a této práce a kapitola 6 shrnuje poznatky získané díky tomuto bakalářskému projektu a snaží se vynést verdikt nad knihovnou Open Scene Graph. Zřejmě bych měl také objasnit nesrovnalost mezi názvem této práce a jejím zadáním. Tato práce se měla původně týkat sady jednoduchých 3D her, ale s vedoucím jsme se domluvili na vytvoření série tutoriálů pro knihovnu Open Scene Graph a zaměření této práce se poté ubíralo směrem k současnému tématu práce. Navíc založit bakalářskou práci na vytvoření několika jednoduchých her ve stylu asteroidy by nemělo význam, jelikož problematika jednoduchých her je spíše pro studenty středních škol a nároky pro studenta ve třetím ročníku vysoké školy by měly být větší. Bohužel jsme pozapomněli patřičně změnit název práce. Tímto se omlouvám za tuto nesrovnalost.

2 Teorie

V této kapitole se budu věnovat knihovně Open Scene Graph. Budu informovat o jejích principech, součástech a posléze alternativách. Cílem této kapitoly bude ukázat jak je to v současné době s nástroji pro tvorbu grafických aplikací a jaké jsou ostatní možnosti pro programátora. V tuto chvíli nechám zhodnocení, zda je Open Scene Graph vhodným nástrojem pro tvorbu 3D aplikací na Vás. Moje osobní zhodnocení se budete moci dočíst na konci tohoto dokumentu.

2.1 Základní informace o Open Scene Graphu

Knihovna Open Scene Graph je multi-platformním nástrojem pro vývoj aplikací jako jsou počítačové hry, virtuální realita, nástroje pro práci s 3D modely, různé vizualizace atd. Není to jen sbírka knihoven, ale zároveň jsou k ní přidruženy ukázkové programy, textury, modely, částicové systémy a další data usnadňující vývoj jakékoliv aplikace. Je to objektově orientovaný nástroj postavený nad OpenGL, který pomáhá programátorům k tomu, aby se mohli soustředit na zásadní otázky a nemuseli se starat o každou maličkost. Open Scene Graph je postaven na jazyce C++ a díky tomu je možné ho používat na většině platforem. Běží na Apple Mac OS X, Windows a na operačních systémech založených na Unixu. Existují pomůcky, které umožňují používat Open Scene Graph i v jiných jazycích než jen C++ jako jsou Python či Lua.

2.2 Scene graph

Open Scene Graph je jak již název napovídá založena na principu scene graph (graf scény). V této podkapitole bych chtěl vysvětlit, co to vlastně znamená.

Scene graph je ve své podstatě sbírka uzlů ve stromové struktuře. Převážnou má každý uzel právě jednoho rodiče, díky čemuž dědí jeho vlastnosti, a tudíž nemusí uživatel nastavovat pro každý uzel vlastnosti zvlášť. Na většinu uzlů se aplikuje transformační matice, jež posouvá, rotuje či mění velikost všech potomků. Toto jsou hlavní důvody, jenž dělají ze scene graphu velice užitečný nastroj pro práci s grafikou. Aby nedošlo k nedorozumění, tak pod pojmem grafika či počítačová grafika je v tomto textu myšlena vektorová 2D/3D grafika. Pokud jde o to, o čem jsem hovořil předtím, tak výhoda dědění je hlavně v tom, že pokud mátě několik jednotlivých grafických objektů tvořící jeden logický celek (například loď s pasažéry, nákladem atd.), tak změna polohy rodičovského objektu mění automaticky i polohu potomků a tím se šetří zbytečné výpočetní operace.

Další velikou výhodou dělení scény do stromových struktur je tzv. bounding volume hierarchies. Jedná se o stromovou strukturu obsahující koule čí osově zarovnané kvádry ohraničující

geometrii každého jednotlivého uzlu a jeho potomků. To velmi významně urychluje rozhodování, zda se má daný objekt vykreslit a rovněž usnadňuje detekci kolizí. Ono je totiž mnohem rychlejší zjistit, zda je ve výhledu koule, než takto kontrolovat každý polygon daného objektu. Zároveň pokud není v zorném poli rodičovský objekt, tak v něm nemůže být ani žádný z potomků, jelikož bounding box (či sphere) rodičovského objektu zahrnuje i potomky. To může při rozsáhlé scéně ušetřit tisíce výpočetních operací.

2.3 Součásti Open Scene Graphu

V této kapitole se budu věnovat součástem knihovny Open Scene Graph. Mezi ty patří například částicové systémy, producent, vlákna, freetype fonty atd.

2.3.1 Producer

Producer (výrobce) je jedna z externích částí Open Scene Graphu, což znamená, že není přímo součástí instalačního balíčku a musíte ho stáhnout zvlášť. Přitom tento modul je jednou z nejdůležitějších částí Open Scene Graphu, jelikož to je ta „věc“, která všechno zobrazuje. Producent obsahuje třídu viewer, které předá uživatel data scény (scene graph) a viewer v každém snímku aktualizuje jejich vlastnosti a zobrazí je. Zároveň tato třída umí nastavovat vlastnosti aplikace, jako jsou velikost okna, fullscreen(zobrazení na celou obrazovku), barevná hloubka obrazu atd. Tato součást by se neměla vyskytovat ve verzi Open Scene Graphu 2, kde by ji měla nahradit samostatná třída viewer.

2.3.2 Open Threads

Open Threads je knihovna pro vytváření a správu vláken. Popisovat vlákna v tomto dokumentu nebudu, jelikož to není předmětem mé bakalářské práce. Pokud chcete získat o vláknech nějaké informace, tak je můžete najít na [8].

2.3.3 OsgDB

OsgDB je modul starající se o načítání dat. Využívá externích pluginů, které rovněž nejsou součástí základního instalačního balíku Open Scene Graphu. Stejně jako Producer a Open Threads se dají stáhnout ze stránek Open Scene Graphu anebo se dají nalézt volně na webu. Data, která jsou načítána pomocí osgDB se dají rozdělit na data modelů, textur, fontů, shaderů a videa. Rovněž existují na webu pluginy pro načítání dat z archivů jako jsou RAR nebo ZIP.

2.3.4 OsgGA

Tento modul má na starosti ovládání pomocí myši nebo klávesnice. Používá zpráv k tomu, aby detekoval stisk klávesy či pohyb myši. Umožňuje uživateli vytvořit si vlastní třídu, která se pak stará o stisky jednotlivých kláves, což znamená, že aplikace správně reaguje na každou klávesu. Způsob, kterým se taková třída vytváří, je ukázán v tutoriálu 6 v příloze na konci dokumentu.

2.3.5 OsgUtil

Toto je velice užitečná knihovna, jež usnadňuje uživateli mnoho práce. Jsou v ní obsaženy funkce pro redukci modelů na modely s méně polygony, generování různých druhů textur, optimalizaci, získávání statistik o scéně jako je počet objektů, trojúhelníků, doba, kterou aplikace stráví vykreslováním či výpočty atd.

2.3.6 OsgText

Tato knihovna slouží pro práci s textem. Využívá knihovnu FreeType k převádění fontů na polygonové modely a textury. Pomocí této knihovny je možné vytvořit pro svou aplikaci prakticky jakýkoliv text. Lze nastavit, aby se text vykresloval zprava doleva, umí vytvářet stíny textu, lze nastavit zarovnání textu atd.

2.3.7 OsgTerrain

K praktickému využití této knihovny jsem se v této práci nedostal, nicméně jak je zřejmé z názvu a z dokumentace slouží tato knihovna k vytváření a práci s terénem. Dokáže vytvářet terén z heightmapy, což je pole hodnot reprezentující výšku v jednotlivých bodech. Je možné upravit vlastnosti terénu „za běhu“. To je vhodné zejména pokud uživatel chce, aby se terén deformoval například při výbuchu.

2.3.8 OsgParticle

OsgParticle je knihovna umožňující vytváření částicových efektů. K částicovým efektům se detailněji dostanu v kapitole 3, takže nyní se pouze zmíním o možnostech této knihovny. Knihovna dokáže definovat nejrůznější vlastnosti částic jako je barva, tvar, textura, velikost, délka života apod. Tyto vlastnosti (plus další, které jsem nezmiňoval) mají možnost být statické, dynamické a náhodné. Lze variabilně nastavit rozmisťování částic v prostoru, jejich směr, rychlost a zkrátka všechny vlastnosti částicového systému, díky čemuž je možné vytvořit prakticky libovolný částicový efekt. Navíc je tato knihovna velmi dobře optimalizována a proto i velmi rychlá.

2.4 Alternativní nástroje

V této kapitole se budu věnovat dalším nástrojům, které jsou k dispozici. Jen s některými z nich mám osobní zkušenosti a proto je většina informací, které zde naleznete získána z webu.

2.4.1 Open Inventor

Open Inventor vznikl z projektu IRIS Inventor, jehož vznik se datuje někdy mezi lety 1988-1989. Projekt prošel množstvím úprav a stal se z něj open source, díky čemuž se jej daří stále vyvíjet. Jedná se stejně jako Open Scene Graph o nadstavbu OpenGL a je založen na principu scene graphu. Je multi-platformní, podporuje PostScriptový tisk, ale jeho nevýhodou je, že nutí používat uživatele interní datový formát, což vede k potřebě psát konvertory.

Jako pomůcku pro vývoj 3D aplikací se mi zdá lepší Open Scene Graph, především díky množství pomůcek a podporovaných formátů, nicméně Open Inventor je i přes jeho stáří díky neustálému vývoji kvalitním nástrojem. Upozorňuji, že hodnocení není objektivní, jelikož informace jsou čerpány z webu a ne z osobních zkušeností.

2.4.2 OpenSG

OpenSG vznikl stejně jako většina scene graphů v době, kdy ztroskotal projekt Fahrenheit (druhá polovina 90. let) od společností Microsoft a SGI. V té době neexistoval žádný grafický nástroj, který by obsahoval vlastnosti, které se očekávali od tohoto projektu, a proto se mnoho týmů rozhodlo vytvořit vlastní podobu scene graphu. Mezi nimi je, pokud se nemýlím, kromě Open Inventoru i Open Scene Graph ačkoliv jeho verze 1.0 vyšla až v roce 2006. Pokud jde o nástroj samotný, tak jde v podstatě o podobný systém jakým je Open Scene Graph, což vychází zejména z faktu, že jde stejně jako u Open Inventoru o scene graph. Stejně jako oba doposud zmiňované systémy je OpenSG multi-platformní. Jako jeho hlavní výhodu bych uvedl podporu Clusteringu, což je způsob jak rozdělit výpočty na více počítačů a tím umožnit vykreslování velmi rozsáhlých, či výpočetně náročných scén v reálném čase. Díky této vlastnosti se z OpenSG stává mocný nástroj pro tvorbu filmových sekvencí, ale pro obyčejného uživatele moc velký význam nemá. Ono je totiž zbytečné mít podporu clusteringu, máte-li doma jedno PC. Na [6] naleznete srovnání Open Scene Graphu z pohledu podporovaných funkcí a vyplývá z něj, že jsou si to nástroje velmi podobné, které se liší jen v několika drobnostech, a proto si nedovolím porovnávat, který z těchto nástrojů je lepší. Zejména je to dáno tím, že jsem neměl možnost se s OpenSG seznámit osobně a nikdo by neměl hodnotit věci, které nezná.

2.4.3 GLUT

OpenGL Utility Toolkit neboli GLUT je sbírka knihoven a utilit pro práci s OpenGL. Je to spíše doplněk OpenGL a proto se nedá příliš srovnávat s Open Scene Graphem. Důvod proč ho zde uvádím, je zejména proto, že je používán na FIT jako nástroj pro tvorbu studentských projektů. GLUT poskytuje funkce, jež se starají o standardní rutiny systému a uživatel tudíž nemusí sám psát funkce pro vytváření okna, správu systémových volání atd., což usnadňuje přenositelnost kódu na ostatní platformy a hlavně nenutí uživatele ke znalosti systémových API. Další výhodou GLUTu je schopnost vykreslovat standardní tvary jako jsou koule, krychle, cylindr a další, což usnadňuje práci, protože uživatel nemusí vytvářet speciální rutiny pro jejich generování. Nicméně GLUT neposkytuje žádné další výhody oproti standardnímu OpenGL a proto si uživatel musí vše ostatní vytvářet sám. GLUT se hodí pro malé aplikace, které nevyžadují načítání modelů, či jiné složité operace, ale pro větší projekty je naprosto nevhodný. Ještě bych se rád zmínil o tom, že GLUT přestal být dále vyvíjen a vzhledem k licenci pod kterou je šířen není možné, aby jeho vývoj převzal někdo jiný nebo ho rozvíjela open source komunita. Proto vznikly nástroje, které GLUT vytvořily znovu od základu pod jinou licencí. Prvním z nich je freeglut, jež je klonem GLUT a OpenGLUT, který GLUT dále rozšiřuje o nové funkce.

2.4.4 OGRE 3D

Ogre 3D je především nástrojem pro tvorbu počítačových her, který má širokou komunitu starající se o jeho vývoj. Je objektově orientovaný a díky kvalitní architektuře se dobře rozšiřuje o nové funkce pomocí pluginů. Stejně jako Open Scene Graph, Open inventor i OpenSG je založen na konceptu scene graphu. Je multi-platformní, podporuje jak OpenGL tak i DirectX a zvládá všechny dnešní formáty shaderů, kterými jsou GLSL(OpenGL), HLSL(DirectX), Cg a asm(jazyk shaderů, který předcházel vzniku GLSL a HLSL). Pokud jde o kvalitu, tak tento nástroj dosahuje úrovně Open Scene Graphu, ale na rozdíl od něj je zaměřen čistě na tvorbu počítačových her. Jeho hlavní výhodou oproti Open Scene Graphu je podpora DirectX.

2.4.5 Unreal Engine 3

Unreal Engine zde neuvádím proto, že bych ho snad chtěl porovnávat s Open Scene Graphem, jelikož by tento produkt jednoznačně vyhrál, ale spíše proto, abych ukázal, kam až se dostal vývoj nástrojů proto tvorbu grafických aplikací. V současné době neexistuje na poli herních enginů nic kvalitnějšího než je právě Unreal Engine 3. Tento nástroj není pouze grafickým enginem, ale zároveň podporuje fyziku, audio, grafický editor, multiplayer, umělou inteligenci a další nástroje pro tvorbu grafických (především herních) aplikací. Podporuje všechny moderní grafické efekty, jako jsou dynamické stíny, HDR barvy, pixel a vertex shadery či pokročilé částicové systémy.

3 Grafické prvky

Tato kapitola se bude věnovat grafickým prvkům použitým v tutoriálech a ve hře. Jejím účelem bude popis těchto prvků a krátké vysvětlení, jakým způsobem se vytvářejí pomocí Open Scene Graph. Jednotlivé podkapitoly obsahují grafické prvky, které jsou obsaženy v tutoriálech, ve hře a některé z těch, které nejsou součástí programové části bakalářské práce.

3.1 Grafické elementy v programových částech

3.1.1 Základní geometrie

Geometrické elementy (trojúhelníky, čtyřúhelníky popř. polygony) jsou základním stavebním kamenem všech 3D aplikací, jelikož jsou na nich založeny architektury moderních grafických karet. Způsob jakým aplikace posílají informace o geometrii na grafickou kartu, zásadně ovlivňuje rychlost dané aplikace. Knihovna Open Scene Graph je jak již jsem psal v úvodu postavená nad OpenGL, takže samozřejmě používá architekturu OpenGL a posílá informace o geometrii pomocí tzv. Vertex Bufferů a Index Bufferů. Jedná se velmi rychlé řešení, jelikož se grafické kartě neposílají informace o každém bodě zvlášť, ale posílají se najednou.

Pokud jde o tvorbu geometrie jako takovou, tak řešení v Open Scene Graphu celkem zdlouhavé a nepřehledné. Ostatně to jak vypadá zdrojový kód pro vytvoření obyčejného čtverce, můžete vidět v tutoriálech. Naštěstí knihovna osgUtil obsahuje funkci pro generování čtverců, což ulehčí programování a zpřehlední kód. S jinými tvary je to ovšem problém.

3.1.2 Pokročilá geometrie

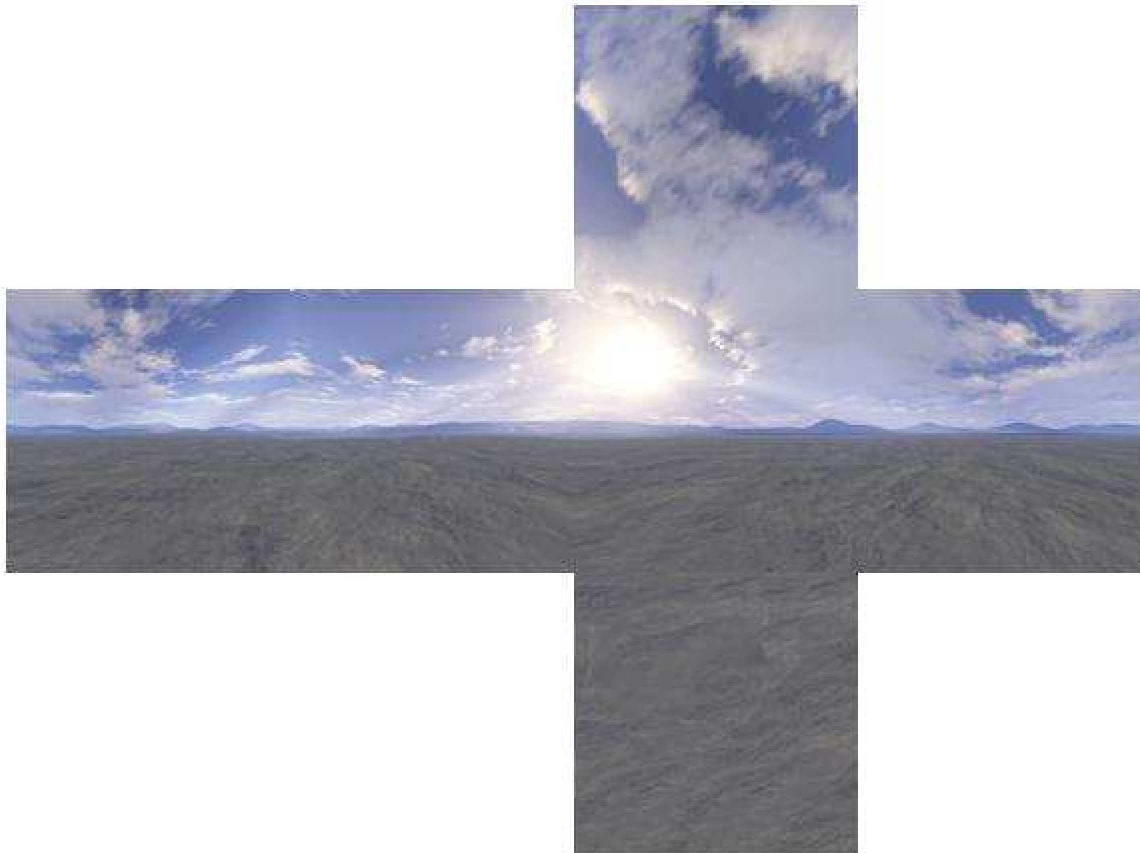
Pokud programujete nějakou aplikaci, tak si jen stěží vystačíte s obyčejnými čtverci či trojúhelníky a v tomto ohledu je Vám Open Scene Graph více než nápomocný. Jak bylo napsáno v kapitole 2.3.3, Open Scene Graph obsahuje funkce pro načítání modelů, a proto vložení složitého geometrického modelu do Vaší aplikace je záležitostí jednoho řádku. Model jako takový si samozřejmě musíte stáhnout, nebo si ho namodelovat v některém z mnoha dostupných modelovacích nástrojů. Pokud se spokojíte v objekty jako je koule, kvádr či jehlan, tak Open Scene Graph opět nabízí jednoduché řešení v podobě funkcí pro generování těchto objektů. Více informací a návod opět naleznete v tutoriálech.

3.1.3 Nastavování stavů

Nastavování stavů je velice důležitou úlohou při stavbě jakékoliv grafické aplikace. Pojmem stav je myšleno například to, zda je zapnuto osvětlení, zda je objekt texturován, či to zda je pro objekt používán Z-Buffer. Nastavování stavů v knihovně Open Scene Graph je snadnou záležitostí díky stromové struktuře, ve které je scéna uložena. Každý objekt dědí vlastnosti díky svému rodiči a proto pokud chce uživatel nastavit nějakou vlastnost pro celou větev objektů, tak stačí nastavit tuto vlastnost pro kořenový uzel této větve. Navíc pokud chce uživatel například vypnout světlo v celé scéně a některé objekty mají tuto vlastnost nastavenu jinak, tak lze zadat parametr OVERRIDE a tím se přepíše nastavení této vlastnosti pro všechny potomky. Pokud je přesto u některého z objektů nutné zachovat nastavení vlastnosti i při použití parametru OVERRIDE, tak lze použít parametr PROTECTED a nastavení se nezmění.

3.1.4 Skybox

Skybox je jedna z metod, pomocí které se vyobrazuje obloha či jakékoliv jiné pozadí aplikace. V podstatě se jedná o kostku, jež je obalená texturou, nicméně je zde několik komplikací, s kterými si musí tvůrce aplikace poradit. Tou první je, že uživatel nesmí ze skyboxu „vyjet“. To se řeší tak, že se v každém snímku nastaví pozice středu skyboxu na pozici pozorovatele. V Open Scene Graphu je nejjednodušším způsobem jak toto udělat, získání pozice z vieweru a její nastavení pro skybox. Dalším problémem je fakt, že skybox nesmí překreslovat žádný objekt ve scéně. Jednou z možností by bylo udělat krychli reprezentující skybox tak velkou, že by přesahovala rozměry scény, ale to by způsobilo problémy se Z-Bufferem. Druhou a zároveň tou správnou je vypnutí DEPTH_TESTu a zajištění, že se skybox bude zobrazovat jako první. K tomuto se dá použít jeden ze stavů nazývaný Render Bin, což je v zásadě pořadník vykreslování. V tuto chvíli má programátor vytvořenu krychli, která se chová tak jak má, nicméně na ni musí umístit textury reprezentující oblohu. U metody skybox je nicméně problém v tom, že tyto textury musí být vytvořeny tak, aby na nich nebylo vidět, že jsou umístěny na krychli. Toto už samozřejmě není problém programátora, ale grafiků. Jednou z metod, které odstraňují problémy s texturami je skysphere, kterou používám ve hře, ale rozepisovat se o ní budu až v kapitole 4.



Obrázek 3.1.1:Textury pro skybox použité v tutoriálech

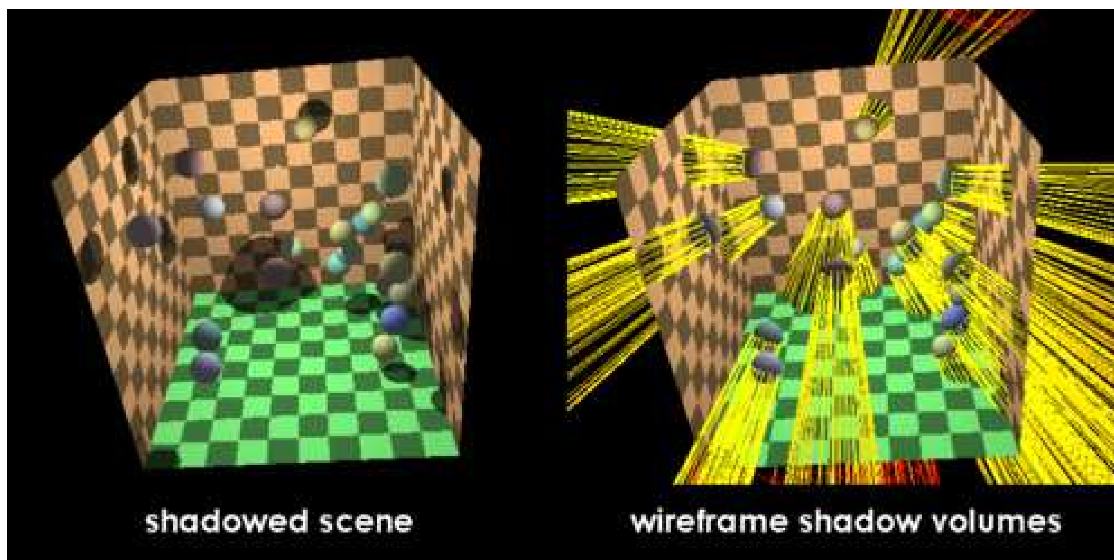
3.1.5 Billboardy

Billboardy jsou objekty, jejichž orientace se mění tak, aby byly stále natočeny směrem k pozorovateli. Je to technika vhodná pro simulaci objektů, které se mají tvářit trojrozměrně, přestože trojrozměrné nejsou. Například se tím celkem dobře simulují stromy či záře světel. Open Scene Graph nabízí třídu, která právě natáčení dělá za Vás. Tato třída se jmenuje jak jinak než Billboard. Uživateli stačí nastavit, po jaké ose se má billboard natáčet a pak přidat k billboardu grafické objekty jako jeho potomky. Vše ostatní se již odehrává automaticky.

Obrázek 3.1.2: Ukázka z tutoriálu č. 9 prezentující Billboardy

3.1.6 Stíny

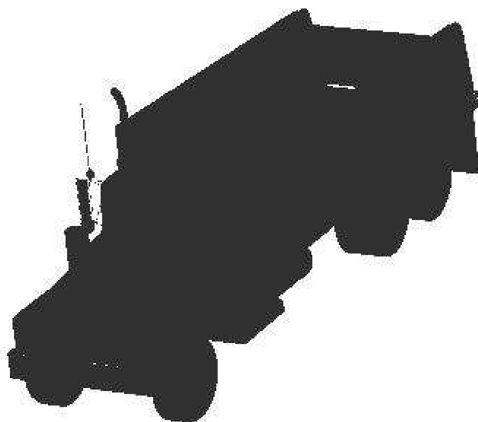
Tvorba stínů v počítačové grafice je velkým problémem. OpenGL i DirectX sice mají prostředky pro osvětlování objektů a tudíž i rozlišují osvětlené a neosvětlené polygony, nicméně již nemají schopnost stíny vrhat. Existuje několik metod pro vrhání stínů. Nejznámějšími metodami jsou *shadow volume* a *shadow mapping*. První metoda nejdříve vrhá paprsek z bodu světla na každý vrchol objektu a vytváří tak geometrii, ve které pixely nejsou osvětlovány. K rozlišování, zda bod je či není uvnitř takovéto geometrie, se používá pomocí *stencil bufferu*.



Obrázek 3.1.3: Vlevo je scéna stínovaná pomocí metody shadow volume. Vpravo je ukázána geometrie stínů. Zdroj obrázku HTTP://EN.WIKIPEDIA.ORG/WIKI/SHADOW_VOLUME

Druhou metodu zde popisovat nebudu, ale místo toho vysvětlím její upravenou verzi, kterou lze použít u knihovny Open Scene Graph. Pokud máte zájem o shadow mapping, tak podrobné informace můžete nalézt na HTTP://EN.WIKIPEDIA.ORG/WIKI/SHADOW_MAPPING.

Zde popisovaná metoda se nazývá Shadow Texture a můžete jí nalézt buď v ukázkových příkladech Open Scene Graphu a nebo v tutoriálech, které jsou umístěny na konci tohoto dokumentu. Tato metoda je dvouprůchodová, což znamená, že nejdříve musíte vykreslit stín do textury a teprve potom vykreslíte samotnou scénu. První věc, kterou je potřeba udělat je vykreslení objektu černou barvou (barvou stínu) do textury. Abyste to mohli provést, tak musíte vědět, kde je objekt umístěn a jak velkou plochu zabírá. K tomuto účelu se hodí bounding sphere, kterou vypočítává pro každý objekt Open Scene Graph(podrobnosti jsou k dispozici v tutoriálech). Pokud máte nastavenu kameru, tak je třeba objekt vykreslit a je nutné ho vykreslit před samotnou scénou. To lze zařídit tím, že pro kameru nastavíte parametr PRE_RENDER. V tuto chvíli dostanete černobílou texturu, kde je černou barvou vyjádřen stín.

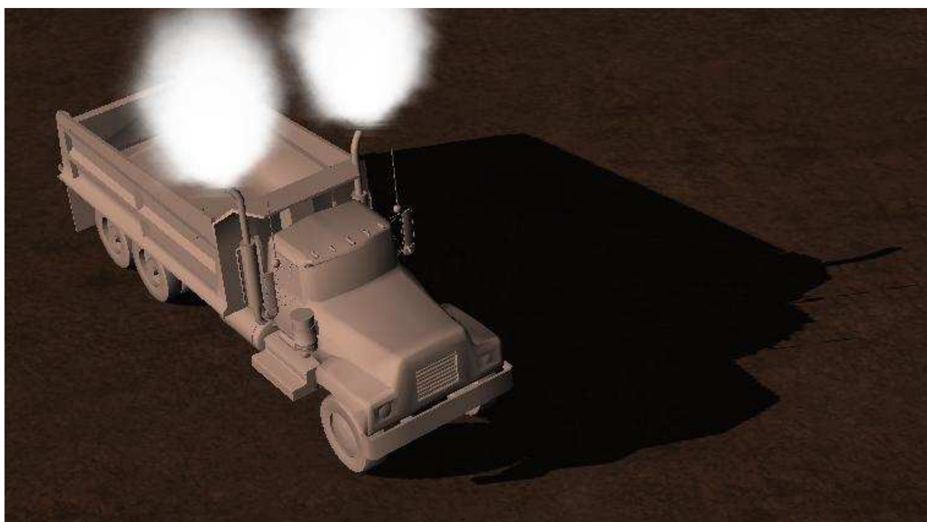


Obrázek 3.1.4 Textura obsahující stín získaná metodou Shadow Texture

Dalším problémem, který je nutné vyřešit je namapování textury na správné místo ve scéně. K tomuto má Open Scene Graph vynikající prostředek spočívající ve třídě TexGenNode. Jedná se o třídu, která se přidá do scény jako uzel, který automaticky generuje texturové koordináty. Jediné co musíte udělat je nastavení matice pro tento uzel. Tu získáte z matice „světelné kamery" převedením jejich koordinátů na texturové koordináty. To se udělá tak, že jí vynásobíte touto maticí.

$$\begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Tímto způsobem dostaneme texturu stínu namapovanou na správné místo ve scéně. Tato metoda je při správné volbě modelu vrhajícího stín velice rychlá. Její nevýhodou je, že čím kvalitnější stín chcete, tím detailnější musí být textura stínu a tím se zvyšují nároky na paměť. Na rozdíl od metody shadow volume však není náročná na výpočetní výkon procesoru.



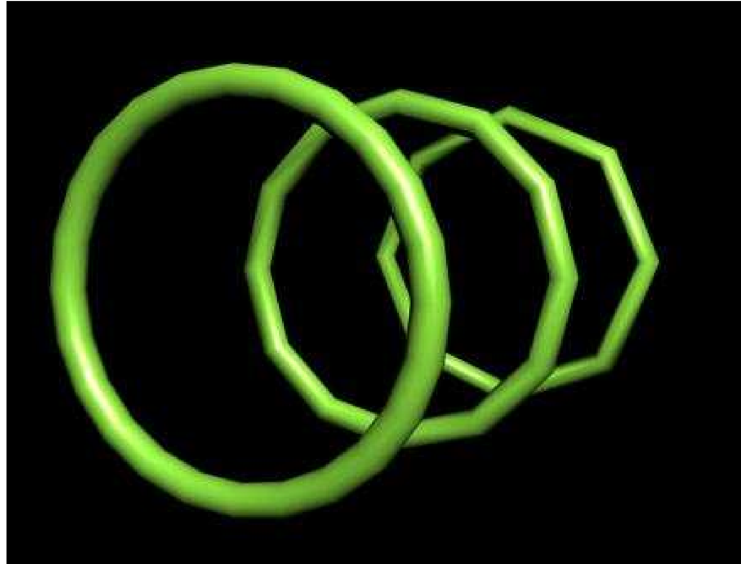
Obrázek 3.1.5 Screenshot z tutoriálu č. 10 demonstrující metodu Shadow Texture

3.1.7 Level of Detail

Level of Detail je technika pro snižování detailu objektů se zvyšující se vzdáleností od pozorovatele. To je důležité, především pokud máte rozsáhlou scénu plnou objektů, jelikož bez použití této techniky se s rostoucím množstvím zobrazovaných objektů rapidně snižuje počet snímků za vteřinu. Navíc pokud je objekt od pozorovatele dostatečně vzdálen, tak je přílišný detail zbytečný a občas i nežádoucí, jelikož to může způsobovat chyby v obraze. Těmito artefakty je myšleno například problikávání polygonů, které jsou blízko u sebe, v důsledku malého rozlišení Z-Bufferu. V Open Scene Graphu dva způsoby jak snižovat detail objektů.

První možností je „vzít“ detailní model a redukovat jeho detail. K tomuto účelu je v Open Scene Graph třída Simplifier, do které vložíte tento model, definujete pole vrcholů, které se nesmí změnit a necháte Simplifier snížit detail modelu. To jak moc se detail sníží, se definuje nastavením proměnné sampleratio. Je dokonce možné detail zvýšit, ale jediné čeho tím dosáhnete, bude zvýšení množství polygonů při stejně vypadajícím objektu.

Druhou možností je si různé detaily modelu vytvořit sám v nějakém modelovacím nástroji. Ve hře jsem použil pávě tuto možnost, ale je to v zásadě jedno, jelikož to pro techniku level of detail nehraje roli.



Obrázek 3.1.6: Obrázek ukazující 3 různé detaily bran použitých ve hře. Změny detailu jsou pro názornost trochu přehnané

Pokud máte vytvořeny různé detaily modelů, tak je stačí přidat do scény. V knihovně Open Scene Graph existuje třída LOD, která se chová jako standardní uzel s tím rozdílem, že zobrazeny jsou vždy jen ty potomci, kteří splňují rozsah velikosti či vzdálenosti. Tento rozsah může být definovaný buď velikostí objektu v pixelech na obrazovce anebo vzdáleností objektu od pozorovatele.

3.1.8 Vesmírný prach

Vesmírný prach je metoda zobrazování bodů či čar kolem pozorovatele dávající dojem, že uživatel prolítá různými nečistotami ve vesmíru. V případě hry jsem použil 200 bodů, které pokud se vzdálí od pozorovatele na určitou distanci, tak se náhodně generuje nová pozice v blízkosti pozorovatele. Více se tomu budu věnovat v kapitole 4. Důvodem, proč se o této problematice zmiňuji je fakt, že v knihovně Open Scene Graph není možné (nebo jsem tuto možnost nikde nenašel) v každém snímku prostě vykreslit body, kterým bych přímo definoval pozici. Vše je zde definováno jako objekt a tudíž pokud chcete, aby se body pohybovali nezávisle na sobě, tak musíte pro každý z nich definovat vlastní transformační matici. Funkce Open Scene Graphu automaticky počítající bounding sphere pro objekty a pro update objektů při velkém množství bodů značně zpomalují aplikaci.

3.1.9 Částicové systémy

Částicový systém je technika pro simulování dějů, které jsou normálním způsobem velmi těžko simulovatelné. Příkladem takovýchto dějů jsou například oheň, kouř, tekoucí voda, sníh či exploze. Základ každého částicového systému jsou částice. Většinou jsou to texturované čtyřúhelníky, ale v Open Scene Graphu je možné dát částicím i tvar bodu či čáry. Částice mají mnoho vlastností, které

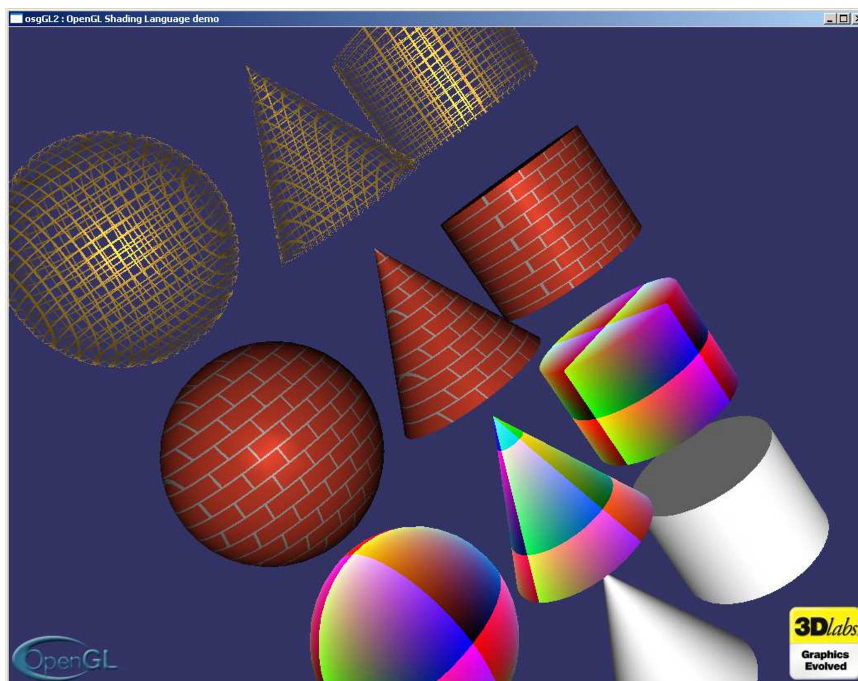
je definují. Jsou to délka života, barva, textura, velikost, hmotnost, pozice, rychlost a směr. Každý částicový systém obsahuje emitter, což je objekt „vyzařující“ částice. V Open Scene Graphu tuto úlohu tvoří třída Emitter, která se stará o všechny funkce s touto úlohou spojené. Těmito funkcemi je myšleno vystřelování částic v určitých intervalech z určitého místa a určitým směrem. Všechny tyto funkce mohou být pevně stanoveny či mohou mít náhodné parametry, které mají zadány meze. Další součástí každého částicového systému je updater, což je objekt, který se stará v každém snímku o aktualizaci vlastností částic. To znamená, že pohybuje částicemi, ničí částice pokud již vypršela doba jejich života a nastavuje ostatní vlastnosti v závislosti na tom, zda jsou tyto vlastnosti proměnlivé či ne. Poslední ze součástí je program, který mění vlastnosti částic. Program obsahuje operátory a pomocí nich se mění například směr či rychlost částice. Open Scene Graph obsahuje několik předpřipravených operátorů, jako je gravitační akcelerátor, pohyb ve větru či síla působící na částice v každém okamžiku. Nicméně je možné si definovat vlastní operátory.

Obrázek 3.1.7: Obrázky ukazující částicové systémy vytvořené knihovnou Open Scene Graph

3.2 Ostatní grafické prvky

3.2.1 Shadery

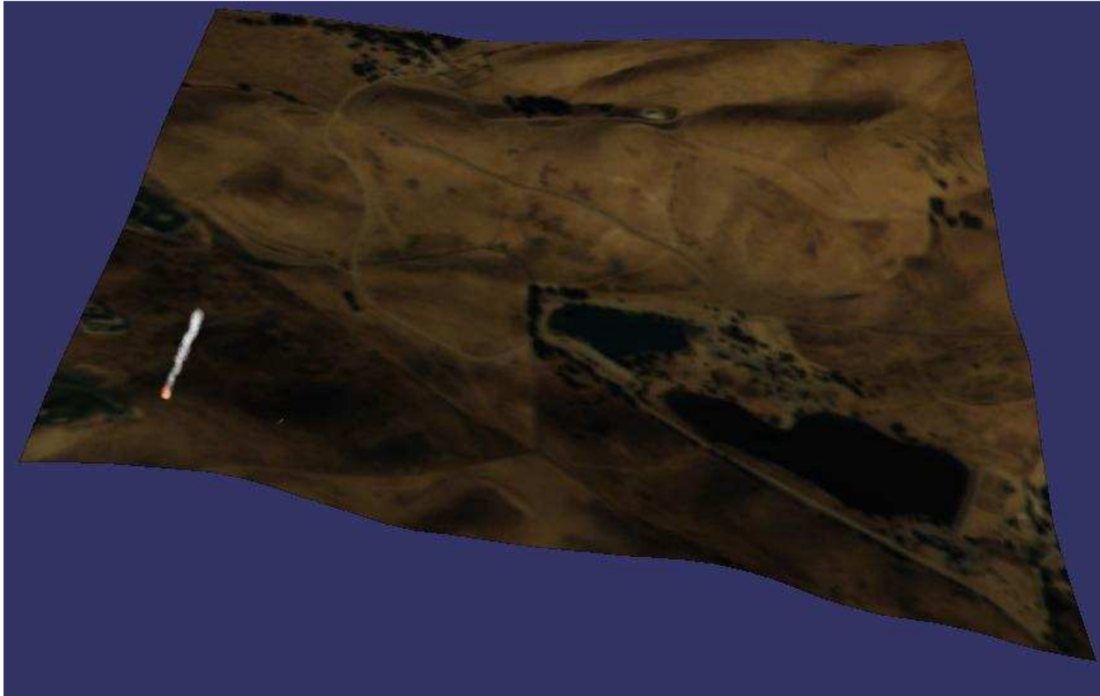
Shadery jsou programy pro grafické karty, pomocí kterých se vytváří většina moderních grafických efektů. Existuje několik verzí. První verze byly v zásadě assemblerem pro GPU a byly pro OpenGL i DirectX stejné. Dnes již existuje verze 4.0 a jedná se o tzv. vyšší shaderové jazyky. Pro DirectX je to HLSL (High Level Shading Language) a pro OpenGL je to GLSL (OpenGL Shading Language). Shadery se dělí na vertex a fragment (pixel) shadery. První pracuje s vrcholy a druhý s pixely. Pomocí těchto programů lze v dnešní době vytvořit prakticky jakýkoliv efekt. Open Scene Graph poskytuje funkce pro načítání, kompilaci a práci s těmito programy.



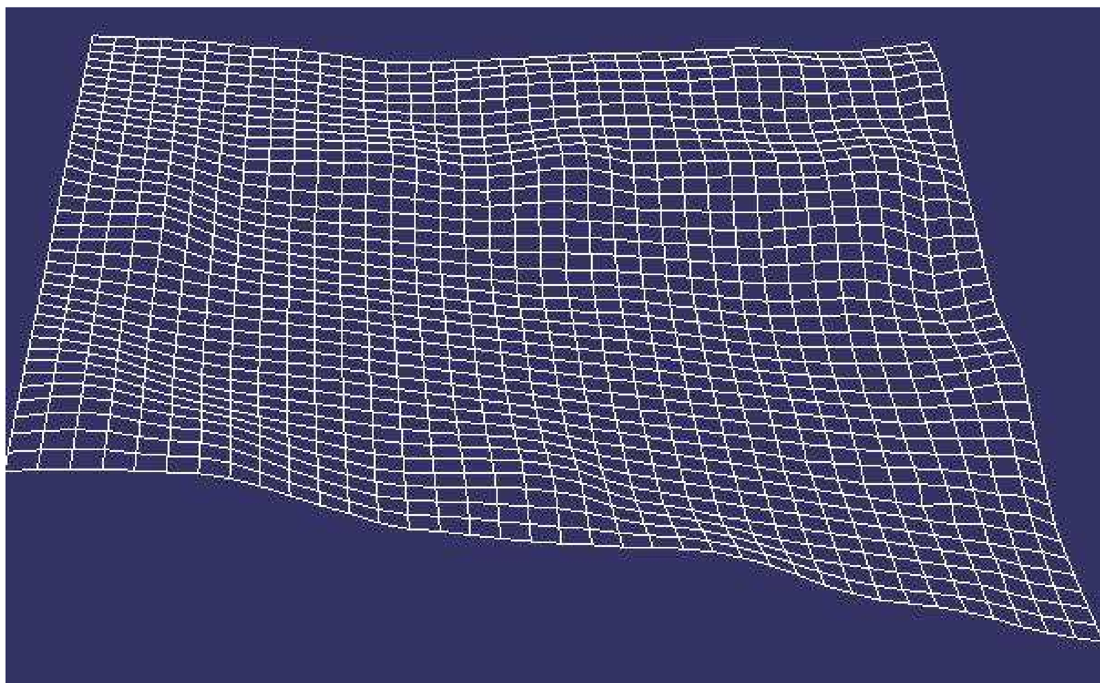
Obrázek 3.2.1: Obrázek zobrazuje 4 stejné série objektů zobrazených pomocí různých shader programů

3.2.2 Terén

Open Scene Graph umí, jak jsem psal v kapitole 2.3.7, vytvářet terén z heightmapy. Jedná se o nejjednodušší techniku pro vykreslování terénu, kdy se vytvoří síť polygonů, jejichž body mají stejnou vzdálenost a liší se pouze výškou. Tato metoda je vhodná pouze pro jednoduché aplikace a její použitelnost v praxi je minimální. Hlavním problémem je, že každé místo terénu má stejný detail a proto pro místa, která jsou naprosto rovná, se používá zbytečně velké množství polygonů a místa, která jich potřebují větší množství (vrcholky hor) jich mají málo. Metoda by byla relativně použitelná, pokud by se terén rozdělil na menší bloky a pro každý blok se používal level of detail. Toto bohužel Open Scene Graph nemá implementováno. Výhodou této funkce nicméně je zabudované vykreslování stínů mraků, což ale tuto funkci nedělá o nic použitelnější.



Obrázek 3.2.2: Terén vytvořený pomocí osgTerrain



Obrázek 3.2.3: Síťový model terénu je zde zobrazen pro názornost

4 Implementace

4.1 Tutoriály

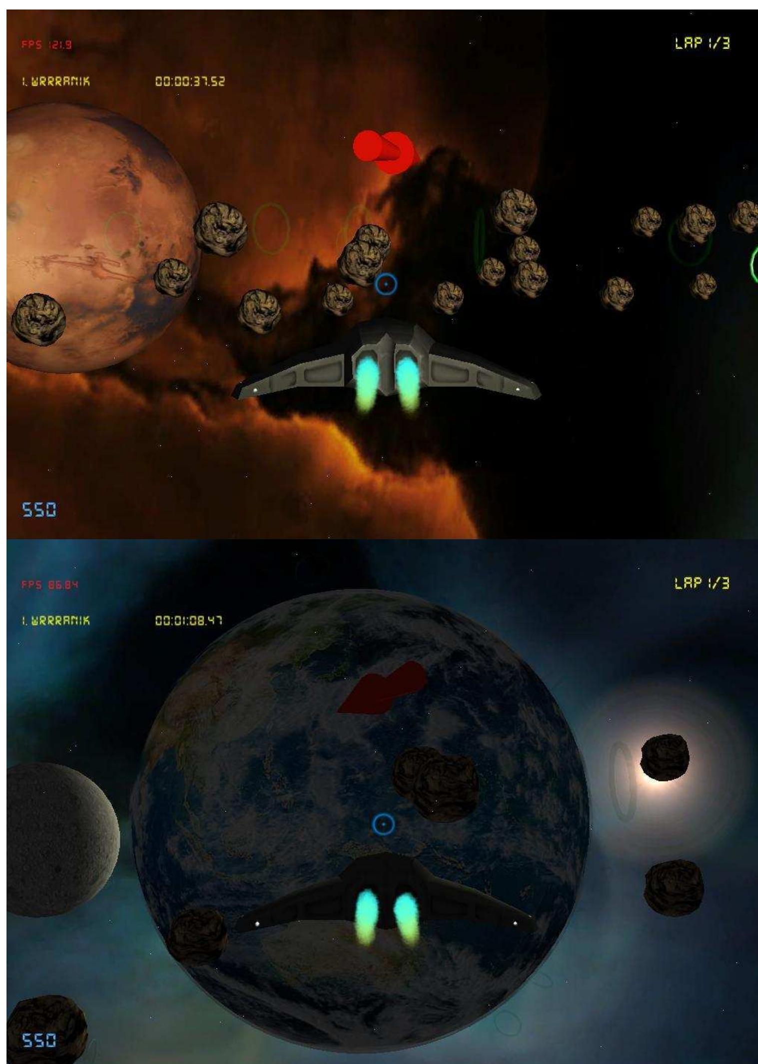
Tutoriály byly první programovou součástí této práce a byly tvořeny v zimním semestru. Z toho vyplývá, že jsem při jejich tvorbě získával zkušenosti s knihovnou Open Scene Graph a podle toho by se mělo přistupovat i ke zdrojovému kódu. Jedná se o sérii deseti aplikací, které na sebe navazují, a každý díl přidává nový prvek do finální aplikace. Touto finální aplikací je myšlena scéna nákladního automobilu, který je možno řídit v krajině. Jejich účelem, stejně jako účelem jakéhokoliv jiného tutoriálů, bylo postupně seznámit uživatele s knihovnou Open Scene Graph a s tvorbou aplikací v této knihovně. Z důvodu redundance textu se nebudu rozepisovat o tutoriálech více, jelikož jsou všechny přiloženy na konci tohoto textu a tam je jejich implementace dostatečně rozepsána. Z důvodu, že jednou z podmínek bakalářské práce bylo vystavení tutoriálů na web a jejich publikace na WWW.ROOT.CZ nebyla prozatím možná, jsou tutoriály k dispozici na webové stránce HTTP://EVA.FIT.VUTBR.CZ/~XVRANA13/TUTORIALS/

Obrázek 4.1.1: Screenshot z finální části tutoriálů

4.2 Počítačová hra

Tato kapitola se zabývá implementací hlavní programové části bakalářské práce, kterou je počítačová hra. Účelem této hry bylo použít zkušenosti získané při tvorbě tutoriálů a pokusit se vytvořit velkou aplikaci, která by vyzkoušela možnosti knihovny Open Scene Graph. Samozřejmě pro důkladné otestování této knihovny by bylo zapotřebí vytvořit takovýchto aplikací několik, ale čas pro tvorbu bakalářských projektů není neomezený a na něco takového by bylo zapotřebí mnohem více času, než je letní semestr.

Jedná se o závodní hru situovanou do vesmírného prostoru. Ve hře jde o to projíždět branami a přitom se vyhnout překážkám v podobě asteroidů a planet. Tato hra rovněž obsahuje možnost síťové hry pro dva hráče.



Obrázek 4.2.1

4.2.1 Základní koncept

Po spuštění se na obrazovce zobrazí hlavní nabídka, ve které si uživatel může vybrat mezi hrou pro jednoho hráče, vytvořením síťové hry, připojením do síťové hry a opuštěním aplikace. Ve hře pro jednoho hráče si uživatel vybere jednu z nabízených tratí (v současné době jsou k dispozici 2), počet kol a spustí hru. Při vytváření síťové hry jsou možnosti stejné s tím rozdílem, že je nutné nejprve vytvořit síťovou hru a poté počkat na připojení protihráče. Teprve poté je možné spustit hru. Pokud se uživatel k síťové hře připojuje, tak se mu zobrazí nabídka vytvořených her a k jedné z nich se připojí. Obrázky se vzhledem menu jsou k dispozici v příloze A.

Po spuštění se objeví nahrávací obrazovka, na které se ukazuje který soubor, se právě načítá. Po nahrání všech potřebných souborů se spustí hra. Pokud jde o síťovou hru, tak se čeká na zprávu od protivníka, že je připraven a pak přichází odpočítávání. Hra samotná spočívá v projíždění branami a odjetí tolika kol, kolik bylo nastaveno v menu. Ve hře je přítomno menu, které ve hře pro jednoho hráče zastavuje čas a umožňuje výběr restartu, pokračování nebo opuštění hry. V síťové hře není povolen restart a čas se nezastavuje, jelikož v tomto režimu slouží menu spíše jako možnost k opuštění hry. Po ukončení hry se program vrací do hlavní nabídky a přitom smaže data obsahující údaje o scéně, aby při vytvoření nové hry nedocházelo k zbytečnému nárůstu používané paměti.

4.2.2 Nastavení grafiky

Pro potřeby nastavení grafiky jsem vytvořil program, který ukládá nastavení do konfiguračního souboru. Ten je při spuštění hry přečten a grafika se nakonfiguruje podle toho. Pokud je hra spuštěna z přiloženého CD a není zkopírována na disk, tak změna konfigurace nebude možná, jelikož se konfigurační soubor načítá z adresáře, který bude rovněž na tomto CD a zápis na toto médium není samozřejmě možný. Možnosti nastavení jsou rozlišení obrazovky, barevná hloubka, rozlišení z-bufferu, to zda bude aplikace běžet ve fullscreenu a jméno hráče.

Obrázek 4.2.2: Obrázek ukazuje vzhled setup programu

4.2.3 Menu

Základními stavebními jednotkami jsou třídy button (tlačítko) a label (popisek). Smysl těchto objektů je zřejmý a proto ho nemá význam vysvětlovat. Pro jistotu uvedu, že význam popisku je zobrazovat informace, aby se menu stalo přehledným a tlačítko má význam ovládací.

Pro popisky je možné při běhu programu nastavovat pouze text, který vykreslují, a při vytváření je možné navíc nastavit pozici a rozměry.

Tlačítko má již možností více. Při vytváření lze rovněž nastavit pozici, rozměr a text, ale navíc je možné zvolit texturu, která se zobrazuje na pozadí a to zda je prvek aktivní či ne. Tlačítko obsahuje funkci update, která používá pozici myši k tomu, aby rozhodla, zda je možné tlačítko stisknout nebo ne.

Obrázek 4.2.3: Tři různé stavy tlačítka. Tlačítko je 1. deaktivováno 2. aktivováno 3. vybráno

Menu má vnitřní stavy, které určují, zda bylo konkrétní tlačítko zmáčknuto. Například pokud dojde ke stisku tlačítka restart, tak se nastaví stav restart a při dotazu v hlavní smyčce zda má dojít k restartu odpoví menu ano.

4.2.4 Nahrávací obrazovka a práce s pamětí

Nejdříve vysvětlím, jak aplikace pracuje s pamětí. Vzhledem k tomu, že velké množství objektů využívá ty samé modely, textury či kolizní modely, tak bylo nepřípustné, aby se pro každý objekt data

načítala zvlášť. Zaprvé by to nepříjemně prodloužilo dobu nahrávání a za druhé by to mělo vážné důsledky na množství využívané paměti. Z těchto důvodů jsem vytvořit třídu DataStorage, která funguje jako úložiště pro objekty. Aplikace vloží do tohoto úložiště informace o souboru, který se má načíst, tato třída tento objekt načte a ostatní objekty si poté pouze vyžádají ukazatel na data. Díky tomuto konceptu je každý datový objekt uložen v paměti pouze jednou a každý objekt pracuje pouze s referencí na originál.

K tomu, aby byla nějaká odezva od aplikace v době nahrávání, jsem vytvořil nahrávací obrazovku. Princip spočívá v tom, že se nejdříve načtou informace o mapě, což je časově zanedbatelná záležitost, poté se tyto informace vloží do seznamu a přidají se informace o menu, lodi atd. Ve chvíli, kdy je seznam naplněn všemi potřebnými informacemi začne nahrávání. To funguje tak, že se vždy přepíšou informace na nahrávací obrazovce a do úložiště se přidá jedna položka ze seznamu. Obrázek nahrávací obrazovky je opět k dispozici v příloze A.

4.2.5 Scéna

Scéna je definovaná souborem s příponou *map*, který je v textovém formátu a obsahuje všechny informace o mapě závodu. Jsou zde vypsány informace o branách, planetách, asteroidech, startovních pozicích, obloze a slunci. Každá z těchto informací začíná názvem informace jako například PLANET a končí slovem END. Mezi těmito slovy jsou pak uvedeny informace o pozici, velikosti atd.

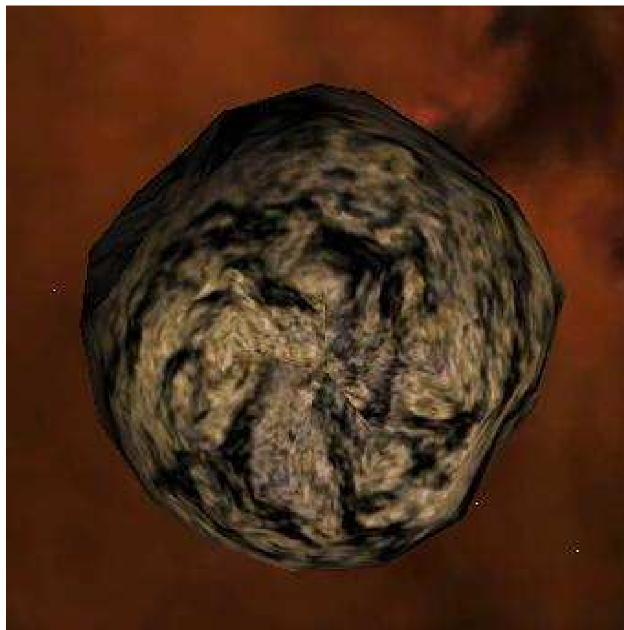
4.2.6 Planety

Planety jsou zde reprezentovány koulemi s použitím level of detail pro snížení detailu při větších vzdálenostech od uživatele. Každá planeta může a nemusí mít atmosféru. Pro detekci kolizí je použita bounding sphere vypočítaná Open Scene Graphem. Planety mají definovanou osu, po které rotují, pozici a rychlost rotace.

Obrázek 4.2.4: Ukázka vygenerované planety

4.2.7 Asteroidy

Asteroidy se v mnohém podobají planetám, tedy alespoň pokud jde o jejich tvorbu. Stejně jako planety mají rotační osu, rychlost, ale používají jiný 3D model. Stejně jako pro planety je použita bounding sphere pro detekci kolizí, což je řešeni rychlé, ale způsobuje to problém s přesností detekce. Občas se stane, že pokud letíte blízko asteroidu, tak do něj narazíte i přesto, že se ho Vaše loď ve skutečnosti nedotkla.



Obrázek 4.2.5: Ukázka asteroidu

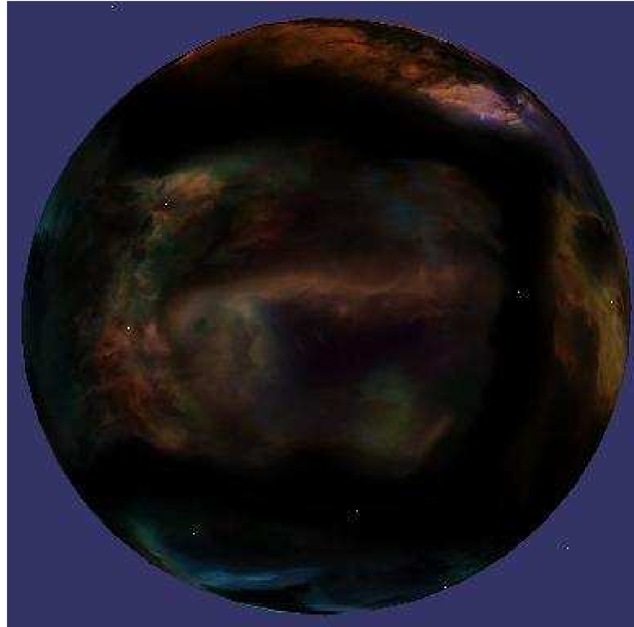
4.2.8 Hvězdy

Hvězda je ve scéně zdrojem světla. Je to billboard, který se neustále otáčí směrem k pozorovateli, takže to vypadá, že jde o trojrozměrný objekt. Vzhledem k tomu, že se nepředpokládá, že by uživatel měl potřebu se k hvězdě přibližovat, tak se na ní nevztahuje detekce kolizí.

Obrázek 4.2.6: Ukázka hvězdy

4.2.9 Obloha

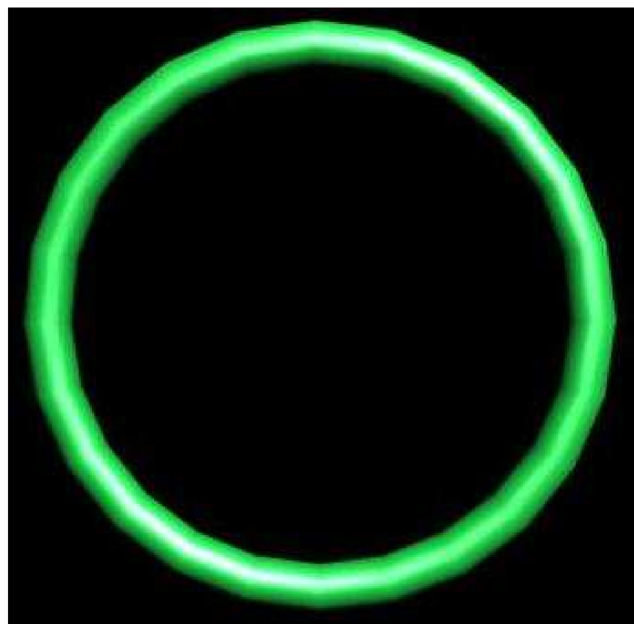
V kapitole 3.1.4 jsem popisoval jednu metodu pro tvorbu oblohy zvanou skybox. Metoda, kterou používám ve hře, řeší její problém s texturami a nazývá se skysphere. Je velice podobná metodě skydome, ale nepoužívá polokouli pro tvorbu oblohy, nýbrž celou kouli. Důvodem, proč zde není problém s texturami, je ten, že se textury nemapují na krychli, ale na kouli a tudíž nemusí sami o sobě dávat dojem kulatosti. Tato metoda je sice o něco náročnější na výkon, než skybox, ale dává přesvědčivý vzhled i bez použití speciálně vytvořených textur.



Obrázek 4.2.7: Nebe vytvořené metodou skysphere

4.2.10 Brány

Brány slouží ve hře jako místa, kterými musí uživatel prolétávat, aby se dostal k poslední z nich a tudíž do cíle závodu. Jako modely používají poloprůhledné kruhy a pouze ten, kterým má loď právě proletět není transparentní, aby byl dobře vidět. Každá brána používá k detekci kolizí kolizní model. Pokud střed lodi „koliduje" s aktuální bránou, stává se aktivní brána následující.



Obrázek 4.2.8: Ukázka brány

4.2.11 Loď

Loď je hlavní součástí hry. Jedná se o 3D model doplněný o dva částicové efekty. Prvním jsou trysky motoru a druhým jsou manévrovací trysky. Pro detekci kolizí se využívá kolizní model. Loď se

ovládá pomocí myši a klávesnice pro zrychlování, zpomalování a rotaci. V síťové hře je pozice lodi protivníka nastavována podle zpráv přijatých ze sítě. Obrázek lodi a jejích částicových systémů lze vidět na screenshotech v kapitole 4.2.12 a proto není nutné ji zde ukazovat.

4.2.12 Kamery

Pro kontrolu kamery jsem vytvořil třídu, jež se chová jako sledovací kamera. Jde v podstatě o objekt, který má nastavenu defaultní pozici vůči sledovanému objektu a tu neustále sleduje. Díky tomu se vytváří dojem jemnějšího pohybu. Je možnost nastavit kameru tak, aby se držela defaultní pozice, a zároveň lze vypnout zobrazení sledovaného objektu, což se hodí při pohledu z kokpitu. Pro hru existují tři nastavení kamery, z čehož jedna je zezadu, jedna z kokpitu a jedna sleduje loď zepředu.

Obrázek 4.2.9: Tyto tři obrázky ukazují jednotlivé kamery vytvořené pro hru

4.2.13 Vesmírný prach

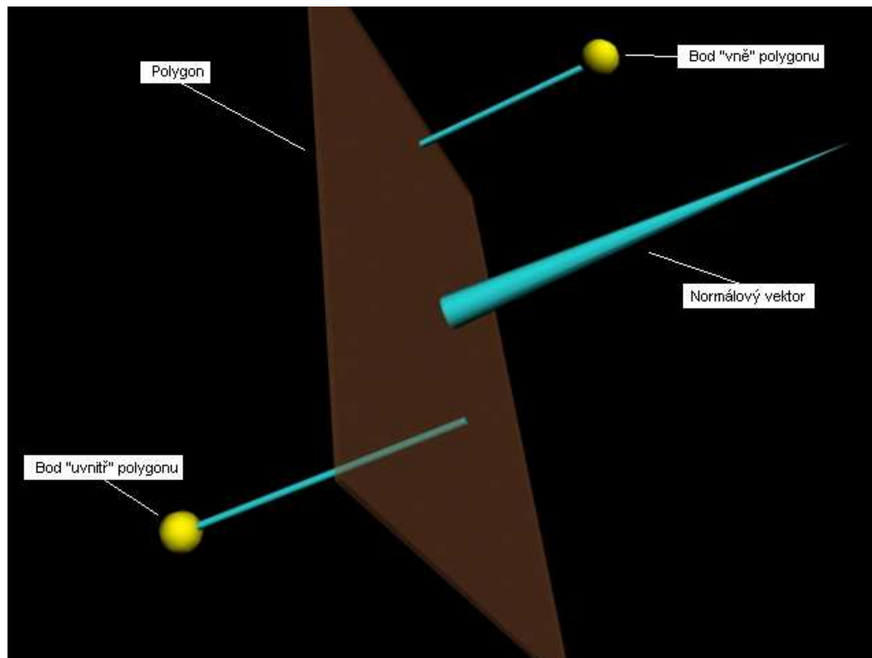
Jak již jsem psal v kapitole 3.1.8, tak vesmírný prach je skupina volně se pohybujících bodů. Kvůli problému výkonem jsou body ve dvojicích, které mají mezi sebou shodnou vzdálenost, a tudíž sdílí transformační matici. Uživatel si tohoto faktu nevšimne a značně to urychluje aplikaci. V případě, že se dvojice bodů dostane z určité vzdálenosti od pozorovatele, nastaví se pro ni nová pozice v blízkosti uživatele a navíc někde před ním. Exaktní pozici nelze popsat, jelikož je to náhodný jev. Směr pohybu se nastaví proti uživateli a tudíž pokud se uživatel nehýbe, tak se nestane, že by bod hned po vložení opustil prostor v blízkosti uživatele a tak musel být znovu umístěn.

4.2.14 HUD

HUD jsou informační data na obrazovce jako rychlost, čas, počet kol či doba závodu. Je implementován pomocí kamerového uzlu, který se nastaví tak, aby nemazal obraz a aby se vykresloval jako poslední. Pak už k tomuto uzlu pouze přidávají grafické entity. Těmito entitami jsou konkrétně textové popisky reprezentující informace uvedené na začátku. Významnější věcí ve třídě HUD je ukazatel směru. Je to model ve tvaru šipky, který ukazuje směr k další bráně.

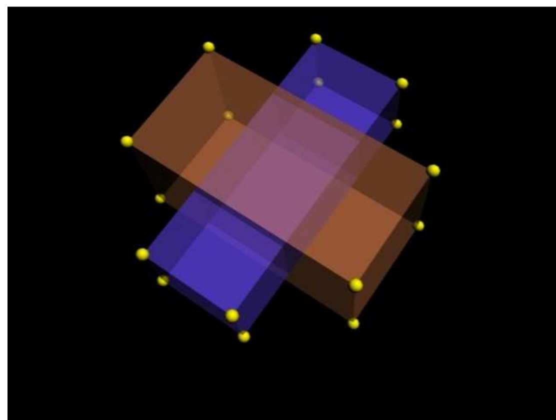
4.2.15 Detekce kolizí

Detekce kolizí slouží k tomu, aby hra správně reagovala například na dotyk lodi a planety. Algoritmus detekce kolizí síťových modelů využívá rovnici pro zjištění vzdálenosti bodu od plochy $ax + by + cz = -d$. Každý polygon totiž vytváří takovou plochu a jeho normálový vektor určuje, zda je bod „vně“ této plochy, nebo „uvnitř“. Pokud polygony vytváří uzavřený objekt, tak lze snadno určit, zda leží bod jiného objektu uvnitř takového objektu nebo ne.



Obrázek 4.2.10: Obrázek ukazuje princip detekce, zda je bod "uvnitř" plochy či ne.

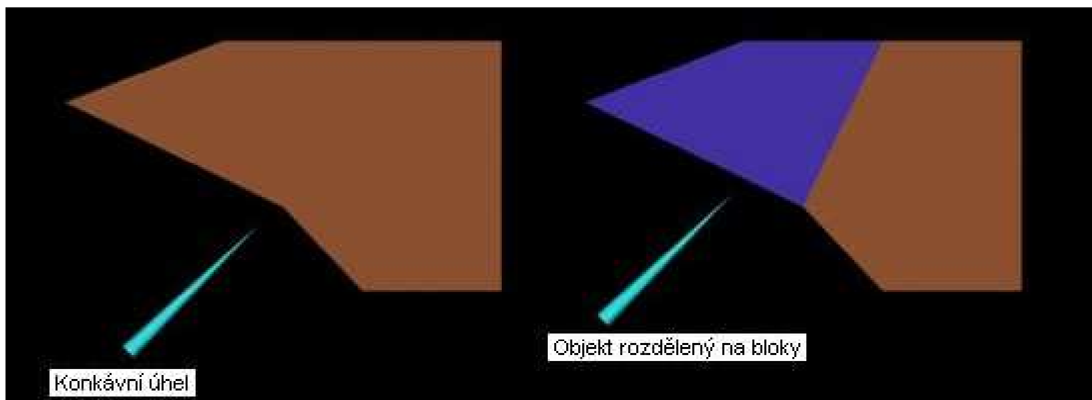
Tato metoda má dva problémy. Prvním je fakt, že dva objekty spolu mohou kolidovat a přesto nebude ležet žádný z jejich bodů uvnitř druhého objektu, a tudíž tato metoda nezjistí kolizi mezi těmito objekty. Tento problém jsem neřešil, jelikož jsou modely dostatečně detailní, aby tato situace nemohla reálně nastat.



Obrázek 4.2.11: Obrázek ukazuje stav, kdy dva objekty zřetelně kolidují, ale metoda pro detekci kolizí to neodhalí.

Druhým problémem je fakt, že žádný vnitřní úhel, který svírají polygony, nesmí být konkávní, neboli nesmí mít více než 180°. Důvod je to, že pokud je v polygonu takový úhel, tak některá z ploch

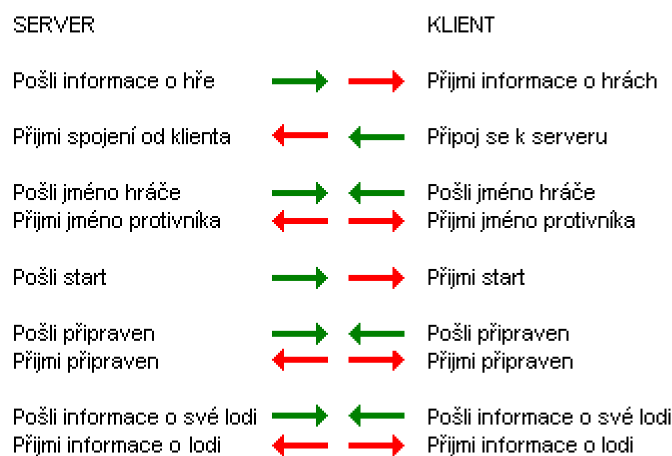
objektu bude ořezávat objekt uvnitř a ne zvenčí, což může mít za následek nedetekování kolize, přestože nastala. Řešením tohoto problému je rozdělení objektu na bloky, které nemají konkávní úhly, a kolize nastává v případě, že bod druhého objektu koliduje s některým z těchto bloků.



Obrázek 4.2.12: Řešení problému s konkávními úhly

4.2.16 Síťová hra

Hra nabízí možnost síťové hry pro dva hráče. Využívá při tom knihovnu winsock pro práci se sockety. Při založení hry jedním hráčem je v pravidelném intervalu tří vteřin vysílána multicastová zpráva, která obsahuje informace o nabízené hře. Hráč, který hledá dostupné hry, tuto zprávu přijme a hra se mu objeví v seznamu serverů. Pokud se hráč rozhodne připojit se, tak se pokusí navázat UDP spojení se serverem, který tuto zprávu vyslal a po připojení čeká na spuštění hry ze strany serveru. Poté co hráč, který hru zakládá, stiskne start, vyšle se druhému hráči zpráva, že má začít nahrávat mapu. Po nahrání mapy si klient i server pošlou informaci o tom, že jsou připraveni a poté se spustí hra stejně jako při hře jednoho hráče. Rozdílem je fakt, že si server a klient neustále vyměňují informace o jejich lodích a tudíž hráč nebojuje pouze s časem, ale zároveň se snaží být rychlejší než protivník. Více informací o multicastu je k dispozici na [3].



Obrázek 4.2.13: Postup při tvorbě síťové hry

5 Možnosti vylepšení

5.1 Tutoriály

V tutoriálech jsem se snažil získat zkušenosti s knihovnou Open Scene Graph, abych ji byl schopen porozumět a měl podklady k jejímu hodnocení. Nicméně množství těchto tutoriálů by mělo být větší, aby bylo moje hodnocení objektivnější. Jednou z věcí, které by měla být přidány je tutoriál o shaderech. Shadery v dnešní době hrají v počítačové grafice hlavní roli a proto schopnost knihovny Open Scene Graph s nimi pracovat je velmi důležitá. Druhou částí této knihovny, které by bylo zapotřebí věnovat větší pozornost je knihovna osgSim s kterou nemám žádné zkušenosti. Navíc jsem neprozkoumal všechny možnosti Open Scene Graphu jako například automatická simulace scény, kterou je vidět v některých příkladech. Tím jsou myšleny animace, kdy se například letadlu nastaví trasa, po které letí.

5.2 Porovnání s ostatními knihovnami

V kapitole 2.4 jsem se věnoval alternativám ke knihovně Open Scene Graph, ale tyto alternativy, jak je tam také napsáno, jsem osobně většinou nezkoumal, jelikož by časová náročnost takového zkoumání zabrala mnohem více času, než který je k dispozici pro tvorbu bakalářské práce. Pokud by se podařilo důkladně prozkoumat tyto alternativy, tak by konečné hodnocení bylo rovněž mnohem objektivnější.

5.3 Rozšíření počítačové hry

5.3.1 Grafika

Přestože jsem se snažil, aby hra vypadala co nejlépe, tak je vždy co vylepšovat. V práci na této hře budu pokračovat a jednou z cest dalšího rozvoje by mělo být použití shaderů pro tvorbu grafických efektů. Hra využívá pouze standardní vektorové osvětlení a bylo by potřeba použít pokročilých metod, aby osvětlení bylo na potřebné úrovni.

Rovněž jsem zvažoval použití stínů, jako v tutoriálech, ale nakonec jsem od toho upustil, vzhledem k tomu, že jsem to nepovažoval za dostatečný přínos pro kvalitu grafiky vzhledem k poklesu výkonu, který by to přineslo. Bylo by dobré použít nějakou metodu pro stínování sama sebe u lodi, ale metodu shadow texture jsem nedokázal tomuto efektu přizpůsobit.

Pro asteroidy by bylo vhodné použít více modelů, k čemuž je hra přizpůsobena, ale potřebovalo by to čas na modelování, který se mi nedostával. Bylo by také dobré přidat více objektů

do vesmíru, jelikož tak jak je mi připadá poněkud chudý. Třída planeta by potřebovala také rozšířit o nové možnosti. Například prstence by umožnily přidat planety typu Saturn.

5.3.2 Uživatelské rozhraní

Pro menu existuje pouze jeden ovládací prvek, kvůli kterému je menu značně omezené a hlavní menu se tomu muselo přizpůsobit. Minimálně by bylo potřeba přidat posuvnou lištu, která by umožnila mnohem více map. V tuto chvíli se při větším množství map nebo nabízených her zobrazují tyto hry mimo obrazovku. Navíc koncept ovládání menu je poněkud neohrabaný, což jsem zjistil v okamžiku, kdy jsem rozšiřoval funkčnost menu, ale v tu chvíli by bylo nutné vytvořit menu od začátku. Situace s menu způsobila značnou nepřehlednost zdrojového kódu.

5.3.3 Zvuky

Hra neobsahuje žádné zvuky, což by měla být jedna z prvních věcí, na které bych se měl zaměřit. Ve vesmíru sice skutečně žádné zvuky údajně slyšet nejsou, ale hra tak působí nevěrohodně a není možné člověka vtáhnout do hry. Navíc by to chtělo přidat nějakou hudbu do menu. Pro implementaci zvuků mám v plánu použít knihovnu OpenAL.

5.3.4 Síťová hra

Koncept síťové hry se mi zdá vcelku dobrý, ale bylo by potřeba předělat její implementaci, aby se chovala ve všech situacích korektně. Navíc by nebylo špatné přidat možnost hry více než jednoho hráče, což by znamenalo jen malé úpravy v konceptu. Původně jsem to měl v plánu, ale neměl jsem možnost to testovat a tak jsem od hry pro více hráčů upustil. Další věcí, o které jsem uvažoval, bylo kolidování hráčů mezi sebou, ale nebyl jsem schopen vytvořit odpovídající fyzikální model a proto jsem pojal hru ve stylu TrackManie, což je závodní hra, které se můžou účastnit stovky hráčů, kteří sice sdílí společnou trať, ale vzájemně se neovlivňují.

6 Závěr

Jak již bylo napsáno na začátku, tak účelem této práce bylo prozkoumání knihovny Open Scene Graph a zhodnocení její použitelnosti v praxi. Pokud jde o zkoumání knihovny, tak jsem udělal vše, co se dalo v čase určeném pro tuto práci stihnout. Až na několik věcí si myslím, že jsem knihovnu prostudoval dostatečně. Vytvořené tutoriály zkoumají podstatnou část možností knihovny, i když samozřejmě ne úplně vše.

Pokud se někdo rozhodne v této práci pokračovat, tak by se měl soustředit na prozkoumání částí uvedených v kapitolách 5.1 a 5.2.

Nyní přichází na řadu zhodnocení knihovny Open Scene Graph. Tato knihovna je prací mnoha lidí a bylo na ní odvedeno ohromné množství práce, která není u konce, a každým dnem se knihovna rozšiřuje o nové možnosti, a každá její funkce je vylepšována. Hodnotit práci takového množství lidí a navíc práci takto rozsáhlou je velmi obtížný úkol, ale vzhledem k získaným zkušenostem se tuto knihovnu zhodnotit pokusím.

Knihovna Open Scene Graph je jednoznačně vhodným nástrojem pro tvorbu 3d aplikací. Koncept scene graphu a snadný způsob přidávání datových objektů do scény velmi pomáhá při tvorbě takové scény. Rovněž fakt, že knihovna automaticky nevykresluje objekty, které nejsou v zorném poli kamery, či to, že sama řadí objekty při vykreslování podle toho, jak vzdáleny jsou od pozorovatele, značně usnadňuje práci. Navíc knihovna obsahuje mnoho pomocných funkcí pro tvorbu a správu scény. Díky systému pluginů, je možno importovat téměř jakýkoliv datový formát a uživatel se tak nemusí starat o případné převádění těchto data do jiného formátu. Samozřejmě, že knihovna Open Scene Graph není dokonalá a našel jsem několik věcí, které by potřebovaly vylepšit, jako například knihovna osgTerrain, ale dokonalé není nikdy nic a navíc se na knihovně stále pracuje, takže tato knihovna bude v budoucnu opět o něco kvalitnějším nástrojem pro tvorbu grafických aplikací.

Literatura

[1] Shreiner, D., *OpenGL : průvodce programátora*. 1.vyd, 2006. ISBN: 80-251-1275-6

[2] Černá, Bohumila., Matematika - lineární algebra. 2.vyd., přeprac., 2004. ISBN: 80-7157-784-7

[3] *Internet Protocol (IP) Multicast [online]*. 18. 1. 1999,
<http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ipmulti.htm>

[4] *Powered By Unreal Technology [online].,*
<http://www.unrealtechnology.com/html/technology/ue30.shtml>

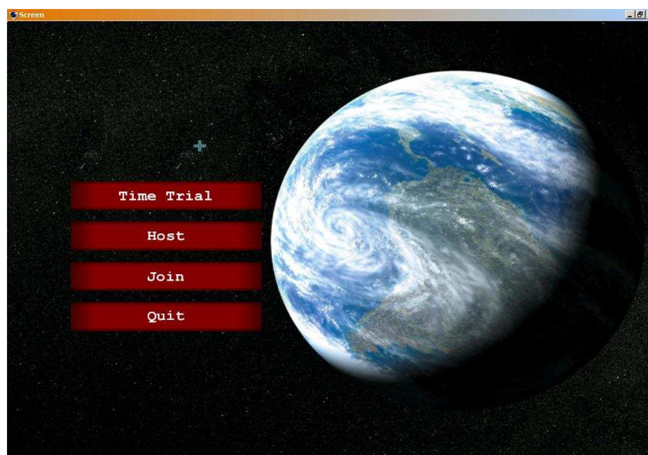
[5] *OGRE 3D - Wikipedia, the free encyclopedia [online].,*
<http://en.wikipedia.org/wiki/OGRE_Engine>

[6] *OSGComparison - OpenSG – Trac [online].,*
<http://opensg.vrsource.org/trac/wiki/OSGComparison>

[7] *Open Inventor - Wikipedia, the free encyclopedia [online].,*
< http://en.wikipedia.org/wiki/Open_Inventor>

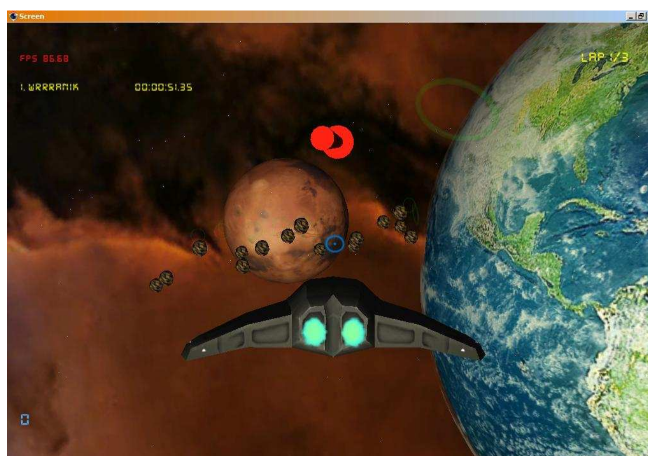
[8] *Vlákno (program) - Wikipedie, otevřená encyklopedie [online].,*
<http://cs.wikipedia.org/wiki/Vlákno_(program)>

Příloha A: Grafické rozhraní



Obrázek A.1: Hlavní menu

Obrázek A.2: Nahrávací obrazovka



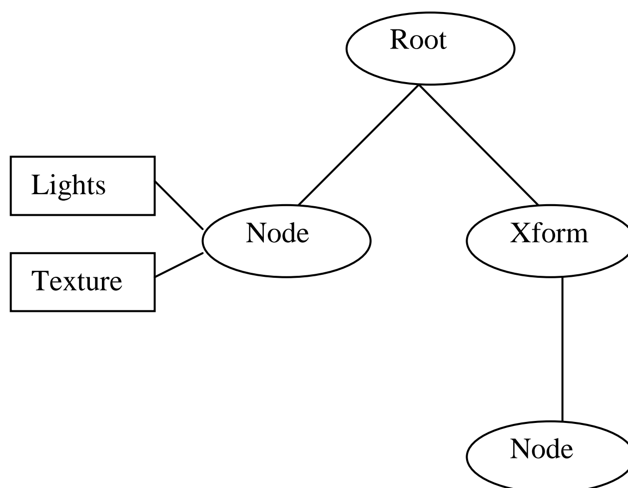
Obrázek A.3: Screenshot ze hry

Příloha B: Tutoriály

B.1 Načítání modelu

Tento článek je první ze série 10 článků, ve kterých se seznámíme s grafickou knihovnou Open Scene Graph (dále jen **osg**). Výsledkem tohoto seriálu by měla být celkem jednoduchá scéna, ve které budeme moci ovládat kamión jezdící po krajině. V každém díle přidáme do této scény jednu vlastnost (popř. prvek) a na konci této série byste měli být schopni vytvářet jednoduché scény či hry.

Co je to vlastně osg? Osg je grafická knihovna pracující s OpenGL. Je to něco jako jeho nadstavba jelikož dělá hodně věcí za Vás. Nemusíte se starat o načítání textur, modelů apod. Je zde docela kvalitní částicový systém, systém kamer, a spousta dalších věcí, které si ukážeme později v dalších dílech. O běh celé aplikace se stará třída producer), které pouze řeknete "Tohle je moje scéna a má takovéhle vlastnosti" a ona už sama scénu zobrazí. Scéna je v osg uspořádána v hierarchickém stromu, na jehož vrcholu je kořenový uzel **root.** Každý uzel může mít svoji transformaci, StateSet(sbírku vlastností jako osvětlení, textura atd.) a potomka. Pokud nemá uzel některou vlastnost definovanou, tak ji dědí od svého předka.



Obrázek B.1.1: Ukázka struktury scény

V tomto díle si ukážeme jak nainstalovat osg, popíšeme si jeho základní princip a nakonec přidáme do scény model kamiónu. K tomuto tutoriálu budete potřebovat znalost C++ a pokud jste už někdy dělali v OpengGL anebo aspoň v DirectX či jiné grafické knihovně, tak to pro Vás bude jen výhoda.

Nejdříve se podíváme na to jak nainstalovat osg a tyto tutoriály. Nejprve si musíte stáhnout osg z této http://www.openscenegraph.com/osgwiki/pmwiki.php/Downloads/Downloads. Máte možnost si vybrat mezi dvěma verzemi. Buďto si můžete stáhnout již zkompilované knihovny anebo zdrojové kódy a ty si potom zkompilovat. Já osobně dávám přednost první možnosti, jelikož Vám pak odpadá značné množství starostí. Po instalaci Vám už stačí pouze přidat složku **X:\OpenSceneGraph\bin** k systémovým cestám (PATH v nastavení systémových proměnných). Tutoriály si stáhněte z http://eva.fit.vutbr.cz/~xvrana13/Tutorials.zip. Tutoriály jsou implementovány v prostředí Microsoft Visual Studio 2005 a jediné co musíte udělat, aby Vám fungovaly je nastavení cest k include a k lib adresářům.

Teď se dostaneme k samotnému příkladu. Není úplně identický jako ten, který je ke stažení, ale rozdíly mezi nimi jsou pouze v rozmístění kódu do jednotlivých modulů.

```
// toto jsou nezbytné moduly pro tento tutorial
#include <osg/Group>
#include <osg/Node>
#include <osgProducer/Viewer>
#include <osgDB/ReadFile>
#include <osg/MatrixTransform>

int main()
{
	// uzel, který bude obsahovat nas model
	osg::Node* Node;
	// transformace, která bude obsahovat pozici naseho modelu
	osg::MatrixTransform* XForm;
	// hlavni uzel
	osg::Group* root = new osg::Group();
	// producent starajici se o beh aplikace
	osgProducer::Viewer viewer;

	// Nacteni modelu je velice jednoduchou zalezitosti, jelikoz
	// osg obsahuje veliké mnozstvi pluginu pro ruzne formaty
	// obrazku a modelu, takze staci pouze zavolat prislusnou
	// funkci a vse se uskutecni automaticky
	Node = osgDB::readNodeFile("dumptruck.osg");
	// Vytvorime si novou transformaci a nastavime Node (nas model)
	// jako jeji dite.
	XForm = new osg::MatrixTransform;
	XForm->addChild(Node);
```

```
        // Nastavime matici naší transformace
        // (v tomto pripade pozici (5,0,0))
        osg::Matrix matrix;
        matrix.makeTranslate(5,0,0);
        XForm->setMatrix(matrix);
        // a pridame model do sceny
        root->addChild(XForm);
        // toto je nastaveni naseho producenta. Timto mu nastavujeme
        // standardni nastaveni jako je zdroj svetla na pozici kamery,
        // manipulator kamery atd.
        viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
        // nastavime data sceny
        viewer.setSceneData( root );
        // a nechame ho provest všechny inicializacni operace
        viewer.realize();
        // hlavni smycka

        while( !viewer.done() )
        {
                viewer.sync();
                viewer.update();
                // vykresleni sceny
                viewer.frame();
        }
}
```

Obrázek B.1.2: Screenshot z prvního tutoriálu

B.2 Transformace a jednoduché tvary

V tomto tutoriálu se budeme zabývat transformacemi a vytvářením jednoduchých tvarů jako jsou kostka, koule či jehlan.

Co je to transformace? Předpokládám, že každý z Vás má matematický základ a proto to není třeba nijak složitě vysvětlovat, ale jednoduše je to matice, pomocí které můžeme rotovat, scalovat nebo posouvat objekt, na který tuto matici aplikujeme. Transformace se do scény přidává stejně jako model z minulého dílu. To znamená, že si vytvoříme objekt transformace a přidáme ho do hierarchického stromu.

```
osg::MatrixTransform * XForm = new osg::MatrixTransform;
root->addChild(Xform);
```

Teď aby tato transformace něco dělala, tak jí musíme přidělit nějakou hodnotu. Dejme tomu, že chceme, aby naše transformace byla translační (posun).

```
osg::Matrix mat;
mat.makeTranslate(10,0,0);
Xform->setMatrix(mat);
```

V tuto chvíli jsme dosáhli toho, že všechny objekty, které přidáme, jako potomky této transformace budou posunuty o 10 na Xové souřadnici. Transformace můžete dávat za sebou, ale musíte si uvědomit, že **záleží na pořadí,** v jakém je za sebe umístíte. To znamená, že rotace a následná translace bude mít zcela odlišný efekt než translace následovaná rotací.

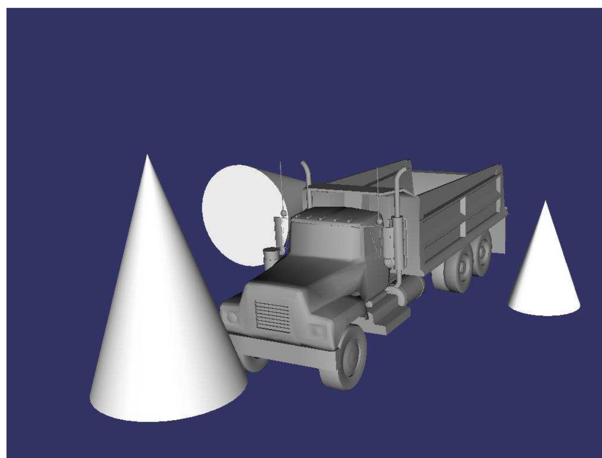
Nyní si ukážeme jak se do osg vkládají základní tvary. Řekněme, že chceme vytvořit kouli o poloměru 1. K tomu musíme vytvořit objekt třídy ShapeDrawable a říct mu, že naším tvarem bude koule.

```
osg::Sphere* sphere = new osg::Sphere( osg::Vec3(0,0,0), 1.0);
osg::ShapeDrawable* sphereDrawable = new osg::ShapeDrawable(sphere);
```

Nyní abychom mohli tuto kouli přidat do scény, tak ji musíme vložit do stromu, ale to nelze provést přímo. Budeme k tomu potřebovat objekt třídy Geode, což je jednoduše řečeno grafický uzel.

```
osg::Geode * geode = new osg::Geode();
geode->addDrawable(sphereDrawable);
root->addChild(geode);
```

Pomocí těchto jednoduchých tvarů můžete vytvořit hodně věcí, ale většinou se moc nepoužívají a uvedli jsme si je jen kvůli jejich jednoduchosti.



Obrázek B.2.1: Screenshot z druhého tutoriálu

B.3 Geometrie a Textury

Tento díl by Vás měl seznámit s tím jak vytvářet v osg geometrii a jak používat textury. Minule jsme si ukázali jak vytvořit jednoduché tvary, ale většinou Vám koule nebo kostka nestačí a proto si ukážeme jak udělat libovolnou geometrii. Navíc pokud si ještě pamatujete, tak jsem psal, že tento seriál by měl být komponován tak, že v každém díle přidáme něco nového k naší scéně s kamiónem a tentokrát tomu tak opravdu bude.

Takže dnes si k našemu kamiónu přidáme nějaký terén a toho docílíme pomocí dostatečně velkého čtverce, na který přidáme 2 textury, aby to vypadalo dobře.

Jak na geometrii v osg? Geometrie je tvořena množinou bodů, množinou normál, které určují, jakým směrem náš bod ukazuje (kvůli osvětlení), množinou texturových koordinátů a množinou ukazatelů, která odkazuje na jednotlivé body a určuje tím jednotlivé strany.

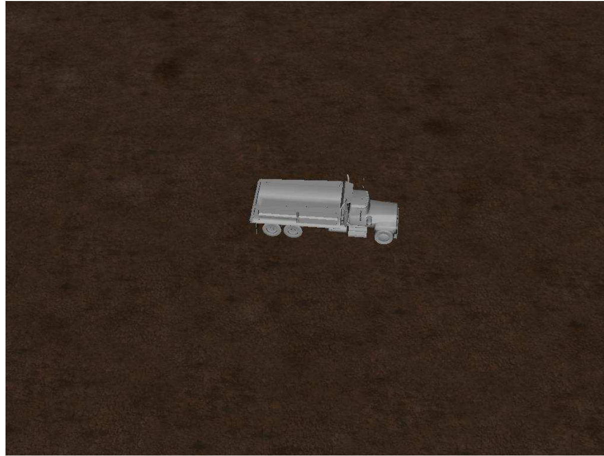
```
// nejdrive si vytvorime tridu geometry, ktera bude obsahovat nas
// teren
osg::Geometry* geometry = new osg::Geometry;

// nyni vytvorime mnozinu bodu (vertex array) a vlozime do nej
// souradnice (jednotlive body)
osg::Vec3Array* TerrainVerts = new osg::Vec3Array;
TerrainVerts->push_back( osg::Vec3( -256, 256, -6.0) );
TerrainVerts->push_back( osg::Vec3( -256, -256, -6.0) );
TerrainVerts->push_back( osg::Vec3( 256, -256, -6.0) );
TerrainVerts->push_back( osg::Vec3( 256, 256, -6.0) );

// priradime tuto mnozinu nasi geometrii
geometry->setVertexArray( TerrainVerts );

// nyni vztvorime mnozinu normal
// vzhledem k tomu, ze vsechny body ukazuji nahoru, tak staci
// vlozit jen jednu normalu a namapovat ji na vsechny body
osg::Vec3Array* TerrainNorms = new osg::Vec3Array;
TerrainNorms->push_back( osg::Vec3( 0, 0, 1) );

osg::TemplateIndexArray
	<unsigned int, osg::Array::UIntArrayType,3,4>
	*normalIndexArray;
normalIndexArray =
```

```
        new osg::TemplateIndexArray<unsigned int,
        osg::Array::UIntArrayType,3,4>;
normalIndexArray->push_back(0);

// priradime mnozinu do naší geometrie
geometry->setNormalArray( TerrainNorms );
geometry->setNormalIndices(normalIndexArray);
// a nastavime mapovani nasi normaly na všechny body
geometry->setNormalBinding(osg::Geometry::BIND_OVERALL);

// ted pridame mnozinu ukazatelu a urcime ze bodou representovany
// jako ctverce
osg::DrawElementsUInt* Terrain =
        new osg::DrawElementsUInt(osg::PrimitiveSet::QUADS, 0);
Terrain->push_back(0);
Terrain->push_back(1);
Terrain->push_back(2);
Terrain->push_back(3);

// a opet vlozime do naší geometrie
geometry->addPrimitiveSet(Terrain);
```

Nyní máme vytvořenou geometrii a budeme chtít přidat textury. Tyto textury budou 2 a to tak, že první bude méně detailní a bude určovat barvu a druhá se bude několikrát opakovat a dodávat terénu větší detail.

```
// vytvorime mnozinu texturovych koordinatu
osg::Vec2Array* texcoords = new osg::Vec2Array(4);
        (*texcoords)[0].set(0.0f,1.0f);
        (*texcoords)[1].set(0.0f,0.0f);
        (*texcoords)[2].set(1.0f,0.0f);
        (*texcoords)[3].set(1.0f,1.0f);

// a nastavime je pro obe textury
geometry->setTexCoordArray(0,texcoords);
geometry->setTexCoordArray(1,texcoords);

// vytvorime texturu
osg::Texture2D* texture = new osg::Texture2D;
texture->setDataVariance(osg::Object::DYNAMIC);
```

```
// a nacteme pro ni data
osg::Image* imData =
            osgDB::readImageFile("../DATA/Textures/sand.jpg");
texture->setImage(imData);

// to same pro druhou texturu
osg::Texture2D* texture2 = new osg::Texture2D;
texture2->setDataVariance(osg::Object::DYNAMIC);
// ale jeste u ni nastavime opakovani jelikoz se bude vyskytovat
// vicekrat
texture2->setWrap(osg::Texture::WRAP_S,osg::Texture::REPEAT);
texture2->setWrap(osg::Texture::WRAP_T,osg::Texture::REPEAT);
osg::Image* imData2 =
            osgDB::readImageFile("../DATA/Textures/detailmap.jpg");
texture2->setImage(imData2);

// nasledujicimi radky se budeme zabyvat podrobneji priste
osg::StateSet* stateOne = new osg::StateSet();

stateOne->setTextureAttributeAndModes(
            0,texture,osg::StateAttribute::ON);
stateOne->setTextureAttributeAndModes(
            1,texture2,osg::StateAttribute::ON);
// jenom zde uvedu, ze nasledujici 2 radky slouzi
// k multitexturingu
osg::TexEnv* blendTexEnv = new osg::TexEnv;
blendTexEnv->setMode(osg::TexEnv::MODULATE);

// a zde nastavime transformaci souradnic druhé textury tak, aby se
// opakovala 16krat
osg::Matrixf sMatrix;
sMatrix.makeScale(16,16,0);
osg::TexMat* texXForm = new osg::TexMat;
texXForm->setMatrix(sMatrix);

stateOne->setTextureAttribute(1,texXForm);
stateOne->setTextureAttribute(1,blendTexEnv);
geometry->setStateSet(stateOne);
```



Obrázek B.3.1: Screenshot z třetího tutoriálu

B.4 Stavy

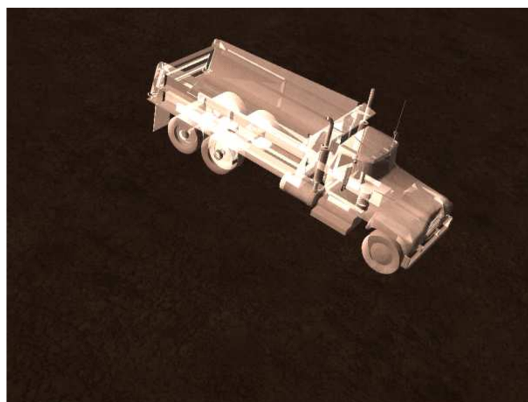
V této části se seznámíme s tím, jak se nastavují stavy v osg. Bude to velice krátký tutoriál, jelikož je to snadné, ale zato dosti důležité.

Takže co jsou to stavy a k čemu slouží? V minulém díle jsem Vám ukazoval jak vytvořit nějakou geometrii a jak na ní položit texturu. Aby osg vědělo, že tu texturu má zobrazit, musíme mu to nějak oznámit a k tomu právě slouží stavy. Pomocí stavů určujete nejenom, jestli se má zobrazit ta či ona textura, ale prakticky všechny informace o daném objektu.

K definici stavů slouží proměnná třídy osg::StateSet a následující kód Vám ukáže jak nastavit nějaký stav. V tomto případě to bude mlha.

```
// nejdrive si vytvorime StateSet
osg::StateSet* RootStateSet = new osg::StateSet();
// ten pak priradime nasemu uzlu (v tomto pripade korenovemu uzlu)
root->setStateSet(RootStateSet);
// nyní vytvorime objekt tridy mlha
osg::Fog* fog = new osg::Fog();
// a nastavime jeho vlastnosti
fog->setMode(osg::Fog::LINEAR); // druh mlhy
fog->setStart(300);            // zacatek
fog->setEnd(1500);             // konec
fog->setColor(osg::Vec4(0.4f,0.5f,0.72f,1.0f));// barva
// a priradime tuto mlhu nasemu StateSetu
RootStateSet->setAttributeAndModes(fog,osg::StateAttribute::ON);
```

Jak již jsem se zmínil v prvním díle, objekty jsou v osg uspořádány hierarchicky a tudíž dědí vlastnosti od svých předků. Pokud tedy nastavíme některou vlastnost kořenovému uzlu, budou ji mít i jeho potomci. Díky tomu můžeme vypnout například světla v celé scéně změnou stavu jediného objektu.



Obrázek B.4.1: Screenshot ze čtvrtého tutoriálu

B.5 Text

Tato část se zabývá vypisováním textu v osg. Zobrazení textu je velice důležité pokud chcete vytvořit menu, nebo popisky na obrazovce.

K tomu abychom vytvořili text, nám poslouží třída osgText::Text, která je odvozeninou od osg::Drawable a tudíž s ní můžeme nakládat jako s jakýmkoliv grafickým objektem. Problém je v tom, že musíme zajistit, aby se nám zobrazovala na správném místě. K tomu poslouží transformace, které jsem Vám předvedl v 2. tutoriálu. Tyto transformace musíme nastavit tak, aby byly ortogonální.

```
// Vytorime si graficky uzel, ktery bude obsahovat cely nas HUD
Geode = new osg::Geode();
// a matice které nastavime tak abychom měli orthogonalni zobrazeni
ProjectionMatrix = new osg::Projection;
ModelViewMatrix = new osg::MatrixTransform;

// Projekcni matici nastavime tak, aby byla orthogonalni a jeji
// souradnice odpovidali bodum na displeji
ProjectionMatrix->setMatrix(osg::Matrix::ortho2D(0,1024,0,768));
ModelViewMatrix->setMatrix(osg::Matrix::identity());

// zajistime aby ModelViewMatice nebyla ovlivnena zadnymi jinymy
// transformacemi
ModelViewMatrix->setReferenceFrame(osg::Transform::ABSOLUTE_RF);
ProjectionMatrix->addChild(ModelViewMatrix);
ModelViewMatrix->addChild(Geode);
```

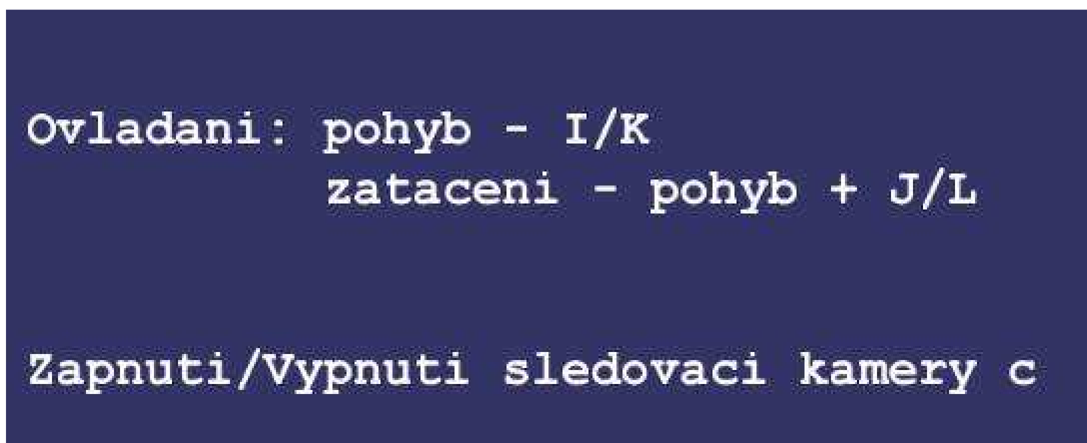
Teď potřebujeme zajistit, aby se nám text zobrazoval jako poslední věc, nebyl osvětlován a aby nebyl ničím překryt. To uděláme tím, že vypneme DEPTH_TEST, LIGHTING a nastavíme vysoké číslo v pořadníku vykreslování.

```
StateSet = new osg::StateSet();
// Text musi prekreslit vsechno ostatni
StateSet->setMode(GL_DEPTH_TEST,osg::StateAttribute::OFF);
// nesmi se osvetlovat
StateSet->setMode(GL_LIGHTING,osg::StateAttribute::OFF);
// a musi se vykreslit posledni
StateSet->setRenderBinDetails( 100, "RenderBin");
Geode->setStateSet(StateSet);
```

V tuto chvíli máme nastavený HUD, ale ještě musíme přidat text. K tomu nám poslouží následující funkce, která vždy přidá text do Geode na námi zadanou pozici.

```
void cHud::AddText(osg::Vec3 pos, char * s)
{
	// vytvorime text
	osgText::Text* text = new osgText::Text();
	// pridame ho do naší Geode
	Geode->addDrawable( text );

	// nastavime vlastnosti textu
	text->setCharacterSize(25); // velikost pisma
	text->setFont("C:/WINDOWS/Fonts/courbd.ttf"); // font
	text->setText(s); // znakovy retezec
	// nastavime orientaci textu, tak aby se otacel k obrazovce
	text->setAxisAlignment(osgText::Text::SCREEN);
	// jeho pozici
	text->setPosition(pos);
	// a barvu
	text->setColor( osg::Vec4f(1.0f, 1.0f, 1.0f, 1) );
};
```

V tuto chvíli byste měli být schopni vypisovat text na obrazovku.



Obrázek B.5.1: Výřez ze screenshotu pátého tutoriálu

B.6 Vstupy z klávesnice

V tomto díle se naučíme ovládat aplikaci pomocí vstupů z klávesnice. Vstup z klávesnice stejně jako například pohyb myši je zaznamenáván pomocí takzvaných událostí (dále jen event). Tento event se pošle pomocí zprávy aplikaci a ta na ni patřičně zareaguje. Asi každý, kdo kdy dělal nějaký program ve winapi, se již se zpracováváním zpráv setkal, a proto pro něj bude tento díl pouze formalitou, jelikož osg to funguje prakticky stejně.

Jako první věc si vytvoříme třídu cKeyEventHandler, která bude dědit z třídy osg::GUIEventHandler, což je třída, která se v osg stará o všechny eventy spojené s ovládáním (klávesnice, myš…). Tato třída bude obsahovat pole, ve kterém budou uloženy informace o jednotlivých klávesách a řídící funkce pro osg::GUIEventHandler.

```
class cKeyEventHandler : public osgGA::GUIEventHandler
{
public:
	struct cKeys
	{
		cKeys()	{	KeyDown = false;
				KeyPressed = false;
			};
		bool KeyDown;
		bool KeyPressed;
	};
	bool isKeyDown(int iKey);
	bool KeyPressed(int iKey);

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

	virtual void accept(osgGA::GUIEventHandlerVisitor& v)	{
		v.visit(*this); };

protected:
	cKeys Keys[65536];
};
```

Struktura cKeys obsahuje informace o tom, zda je daná klávesa stisknuta a jestli byla následně uvolněna.

Teď si pro tuto třídu musíme nadefinovat funkce handle, isKeyDown, KeyPressed.

```
// Tato funkce vraci informace o tom, zda je klavesa stisknuta
bool cKeyEventHandler::isKeyDown(int iKey)
{
      if (iKey < 0) return false;
      return Keys[iKey].KeyDown;
};

// Tato funkce vraci informace o tom, zda klavesa byla stisknuta a
nasledne
   uvolnena (je to vhodne pro prepinani stavu apod.)
bool cKeyEventHandler::KeyPressed(int iKey)
{
      if (iKey < 0) return false;
      if (Keys[iKey].KeyPressed)
      {
            Keys[iKey].KeyPressed = false;
            return true;
      }
      else return false;
};

// Tato funkce se stara o zpracovavani eventu a nastavuje stav ymacknuto u
// konkretni klavesy
bool cKeyEventHandler::handle(const osgGA::GUIEventAdapter&
                        ea,osgGA::GUIActionAdapter& aa)
{
// zjistime ktera klavesa byla stisknuta
      int iKey = ea.getKey();
      switch(ea.getEventType())
      {
            case(osgGA::GUIEventAdapter::KEYDOWN):
            {
                  if (iKey < 0) break;
                  Keys[iKey].KeyDown = true;
                  Keys[iKey].KeyPressed = true;
                  return true;
            };
            case(osgGA::GUIEventAdapter::KEYUP):
            {
```

```
                        if (iKey < 0) break;
                        Keys[iKey].KeyDown = false;
                        Keys[iKey].KeyPressed = false;
                        return true;
                };
        default:
                return false;
        }
        return false;
};
```

V tuto chvíli máme vytvořeno vše, co budeme potřebovat pro naší třídu cKeyEventHandler a zbývá nám ji pouze přidat do seznamu EventHandlerů. Upozorňuji, že pokud vložíte třídu cKeyEventHandler na začátek seznamu, tak ztratíte přístup k věcem jako je zapínání a vypínání světel pomocí klávesy „L“ apod.

```
        viewer.getEventHandlerList().push_back(keyEH);
```

A to je vše. Pokud máte například automobil a chcete, aby při držení klávesy „A“ zatočil do leva, tak stačí následující.

```
        if (keyEH->isKeyDown('a')) auto.turn_left();
```

B.7 Skybox

V tomto díle se naučíme vytvářet tzv. *skybox*. Nejdříve bych měl asi vysvětlit co to ten skybox vlastně je. Pokud děláte nějakou otevřenou scenerii a je jedno zda to je fotbalové hřiště na vesnici, nebo orbita naší planety, tak musíte nějakým způsobem ztvárnit oblohu (nebo hvězdy). V počítačové grafice existuje mnoho možností jak toho dosáhnout. Můžete vykreslit pozadí modrou barvou a zvlášť vykreslovat každý mráček a slunce nebo vykreslit statickou oblohu. První možnost je vhodná, pokud děláte dynamickou oblohu a chcete, aby se všechno hýbalo a měnilo, ale je to náročnější na výpočetní výkon, složitější a hlavně často zbytečné. Velmi často je vhodné použít možnost druhou, jelikož obloha je většinou pouze pro dokreslení prostředí a tudíž hraje až druhé housle. Možností jak ztvárnit statickou oblohu je více, ale nejpoužívanějšími jsou *skydome* a skybox. Já jsem se rozhodl popsat pávě druhou jmenovanou.

Takže abych se konečně dostal k tomu co to ten skybox je. Jak už název napovídá, jde v podstatě o krychli, na kterou se ze všech stran namapuje textura oblohy. Nevýhodou této metody je to, že musíte někde sehnat vhodnou 6tici textur, která bude po „nalepení" na krychli vytvářet dojem, že se díváte opravdu na oblohu. Abyste si udělali představu, jak to vlastně vypadá, tak je zde obrázek, který by Vám v tom měl pomoci.

Obrázek B.7.1: První obrázek ukazuje 6tici samostatných textur a druhý to, jak se skládají dohromady. Zdroj obrázků http://www.turbosquid.com

Teď je načase ukázat trochu té implementace. Jako první věc si vytvoříme krychli stejným způsobem, jako jsme to dělali v třetím tutoriálu. Abych zbytečně nenatahoval délku tohoto textu, tak jsem zde nevypisoval třídu cSkybox a ukážu zde pouze kód pro vytvoření jedné stěny krychle, jelikož pro ostatní stěny je kód totožný s tím, že jsou tam uvedené jiné hodnoty.

```
Geode = new osg::Geode;
geometry = new osg::Geometry;
```

```
Verts = new osg::Vec3Array;
Verts->push_back( osg::Vec3(-10, -10, -10));
Verts->push_back( osg::Vec3(-10, 10, -10));
Verts->push_back( osg::Vec3(-10, -10, 10));
Verts->push_back( osg::Vec3(-10, 10, 10));
Verts->push_back( osg::Vec3(10, -10, -10));
Verts->push_back( osg::Vec3(10, 10, -10));
Verts->push_back( osg::Vec3(10, -10, 10));
Verts->push_back( osg::Vec3(10, 10, 10));

geometry->setVertexArray( Verts );

side[0] = new osg::DrawElementsUInt(osg::PrimitiveSet::QUADS, 0);
side[0]->push_back(0);
side[0]->push_back(1);
side[0]->push_back(3);
side[0]->push_back(2);
geometry->addPrimitiveSet(side[0]);
```

Teď bych se měl asi zmínit o tom, jak zde pracuji s texturami. Pro potřeby skyboxu používám tzv. *cubetexture*, což není nic složitého. Jedná se o šestici textur, které po složení vytvářejí tvar krychle s tím, že pro ně platí jiná pravidla pro texturování. Používají se hned 3 texturové koordináty pro každý vertex a určují směr od středu krychle do její stěny. Z tohoto důvodu si texturové koordináty nastavíme tak, aby pro každý bod našeho skyboxu určovali vektor od středu.

```
texcoords = new osg::Vec3Array(8);
      (*texcoords)[0].set(-1.0f,1.0f,1.0f);
      (*texcoords)[1].set(-1.0f,1.0f,-1.0f);
      (*texcoords)[2].set(-1.0f,-1.0f,1.0f);
      (*texcoords)[3].set(-1.0f,-1.0f,-1.0f);
      (*texcoords)[4].set(1.0f,1.0f,1.0f);
      (*texcoords)[5].set(1.0f,1.0f,-1.0f);
      (*texcoords)[6].set(1.0f,-1.0f,1.0f);
      (*texcoords)[7].set(1.0f,-1.0f,-1.0f);
geometry->setTexCoordArray(0,texcoords);
```

Důležité je nastavit parametry textury na CLAMP_TO_EDGE, jinak se budou na hranách krychle objevovat artefakty v podobě čar.

```
texture = new osg::TextureCubeMap;
texture->setWrap(osg::Texture::WRAP_S, osg::Texture::CLAMP_TO_EDGE);
texture->setWrap(osg::Texture::WRAP_T, osg::Texture::CLAMP_TO_EDGE);
texture->setWrap(osg::Texture::WRAP_R, osg::Texture::CLAMP_TO_EDGE);
```

Pak už stačí jen načíst obrazová data pro každou stěnu textury.

```
texture->setDataVariance(osg::Object::DYNAMIC);
imData[0] = osgDB::readImageFile("../DATA/Textures/left.jpg");
imData[1] = osgDB::readImageFile("../DATA/Textures/right.jpg");
imData[2] = osgDB::readImageFile("../DATA/Textures/back.jpg");
imData[3] = osgDB::readImageFile("../DATA/Textures/front.jpg");
imData[4] = osgDB::readImageFile("../DATA/Textures/top.jpg");
imData[5] = osgDB::readImageFile("../DATA/Textures/surface.jpg");

texture->setImage(osg::TextureCubeMap::NEGATIVE_X, imData[0]);
texture->setImage(osg::TextureCubeMap::POSITIVE_X, imData[1]);
texture->setImage(osg::TextureCubeMap::NEGATIVE_Z, imData[2]);
texture->setImage(osg::TextureCubeMap::POSITIVE_Z, imData[3]);
texture->setImage(osg::TextureCubeMap::NEGATIVE_Y, imData[4]);
texture->setImage(osg::TextureCubeMap::POSITIVE_Y, imData[5]);
```

V tuto chvíli již máme vytvořený skybox jako takový, nicméně to rozhodně není vše co je potřeba udělat. Je důležité, abyste u jeho vykreslování vypnuli DEPTH_TEST a zařídili, že se bude vykreslovat jako poslední a vypnuli osvětlování.

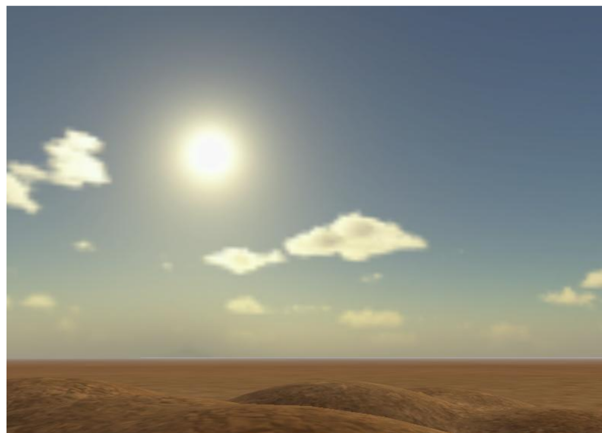
```
stateSet = new osg::StateSet();

// Zapneme texturu
StateSet->setTextureAttributeAndModes(0, texture,
      osg::StateAttribute::ON);

// Vypneme DEPTH_TEST a osvetlovani
stateSet->setMode(GL_DEPTH_TEST,osg::StateAttribute::OFF);
stateSet->setMode(GL_LIGHTING,osg::StateAttribute::OFF);

// a rekneme osg aby vykresloval skybox jako prvni
stateSet->setRenderBinDetails(-1, "RenderBin",
      osg::StateSet::USE_RENDERBIN_DETAILS);
```

```
        geometry->setStateSet(stateSet);
```

Nyní se chová skybox tak jak má s jednou výjimkou. V každém snímku, nebo jen při pohybu kamery (to záleží na Vás), musíme skybox umístit na pozici pozorovatele. Toho docílíme tak, že si vytvoříme funkci Update, která bude mít jako parametr viewmatrix (matici, která udává pozici a orientaci kamery). Tu získáme například od vieweru pomocí viewer.getViewMatrix(). Ještě je dobré se zmínit o tom, že funkci update musíte volat až po viewer.update().

```
void cSkybox::Update(osg::Matrixd viewmat)
{
        osg::Vec3 pos; // pozice kamery
        // pomocná proměnná, do které budeme „zahazovat" pro nás irelevantní
        // informace
        osg::Vec3 pom;
        // získáme pozici kamery
        viewmat.getLookAt(pos,pom,pom);
        // a nastavíme podle ní pozici skyboxu
        viewmat.makeTranslate(pos);
        XForm->setMatrix(viewmat);
};
```

Nyní byste měli být schopni vytvořit pro Vaší aplikaci vlastní skybox ne nepodobný tomuto.



Obrázek B.7.2: Ukázka nebe vytvořeného metodou skybox

B.8 Částicové efekty

V tomto díle bych Vám rád představil částicové efekty v osg. K čemu vlastně částicové efekty slouží? Pokud například chcete simulovat ve své aplikaci oheň, kouř, nebo například vodopád, tak to jen stěží budete dělat pomocí síťových modelů, jelikož by to bylo výpočetně neuvěřitelně náročné a navíc i zbytečně pracné. K těmto účelům právě slouží částicové efekty. Nyní k tomu, co to vlastně částicové efekty jsou. Jsou to simulátory stovek až tisíců jednotlivých částic, které vytvářejí dojem reálných systémů.

Každá částice je texturovaný objekt (nejčastěji čtverec) orientovaný na obrazovku a obsahuje údaje o pozici, směru, rychlosti a délce života atd. O všechny částice se pak stará částicový systém, jenž má následující podsystémy.

Placer – udává startovní pozici pro každou částici.

Shooter – udává počáteční rychlost a směr částic.

Counter – určuje počet částic, které jsou vytvářeny v každém snímku.

ModularEmitter – neboli vyzařovač. Je to třída, jejž obsahuje Placer, Shooter, Counter.

ParticleSystemUpdater – Poskytuje funkci update pro každou částici.

Operator – kontroluje změnu stavu (pozice, směr…) každé částice během jejího života.

ModularProgram – obsahuje operátory pro změnu stavů částice.

Nyní se dostáváme k samotnému postupu vytváření částicového systému. Jako první si vytvoříme instanci osg::ParticleSystem a přidáme ji do scény.

```
SmokeParticleSystem = new osgParticle::ParticleSystem;
// nastavime texturu castic a osvetlovani castic ponechame vypnute
SmokeParticleSystem->setDefaultAttributes(
      "../DATA/Textures/smoke.rgb", false, false);
SmokeGeode = new osg::Geode;
// pridame casticovy system do sceny
root->addChild(SmokeGeode);
SmokeGeode->addDrawable(SmokeParticleSystem);
```

Dále vytvoříme ParticleSystemUpdater a přiřadíme k němu částicový systém.

```
SmokeSystemUpdater = new osgParticle::ParticleSystemUpdater;
SmokeSystemUpdater->addParticleSystem(SmokeParticleSystem);
root->addChild(SmokeSystemUpdater);
```

Nastavíme defaultní hodnoty pro částice.

```
osgParticle::Particle SmokeParticle;
SmokeParticle.setSizeRange(osgParticle::rangef(0.1,5.0)); // velikost
SmokeParticle.setLifeTime(2); // delka zivota
SmokeParticle.setMass(0.01); // hmotnost
SmokeParticleSystem->setDefaultParticleTemplate(SmokeParticle);
```

Vytvoříme a nastavíme Emitter, Placer, Shooter a Counter.

```
SmokeEmitter = new osgParticle::ModularEmitter;
SmokeEmitter->setParticleSystem(SmokeParticleSystem);
SmokeRate = static_cast<osgParticle::RandomRateCounter *>(
      SmokeEmitter->getCounter());
SmokeRate->setRateRange(30,50 );

sectorPlacer = new osgParticle::SectorPlacer();
// posuneme startovni pozici vzhledem k objektu ke kteremu je
// casticovy system uchycen o vektor pos
sectorPlacer->setCenter(pos);
// nastavime rozsah moznych startovnich poloh
sectorPlacer->setRadiusRange(0.5,0.5);
sectorPlacer->setPhiRange(3.1415/8,3.1415/8);
SmokeEmitter->setPlacer(sectorPlacer);

SmokeShooter = new osgParticle::RadialShooter();
// nastavime smer vystrelovani cstic
SmokeShooter->setThetaRange(0.0, 3.14159/6);
// nastavime rychlost castic
SmokeShooter->setInitialSpeedRange(50,100);
SmokeEmitter->setShooter(SmokeShooter);
```

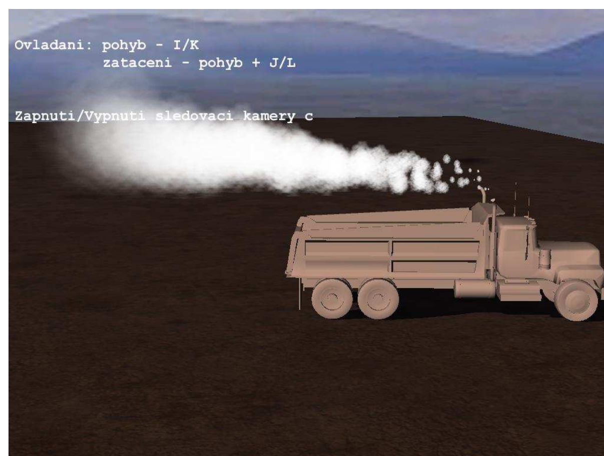
A nakonec vytvoříme ModularProgram a přidáme do něj operátory.

```
moveSmokeInAir = new osgParticle::ModularProgram;
moveSmokeInAir->setParticleSystem(SmokeParticleSystem);
// nastavime gravitaci (zapornou, aby castice stoupaly)
gravity = new osgParticle::AccelOperator;
gravity->setToGravity(-3);
moveSmokeInAir->addOperator(gravity);
// nastavime pohyb ve vzduchu
airFriction = new osgParticle::FluidFrictionOperator;
airFriction->setFluidToAir();
moveSmokeInAir->addOperator(airFriction);

root->addChild(moveSmokeInAir);
```

V tuto chvíli máme vytvořený částicový systém stoupajícího kouře a poslední věcí, která nám zbývá, je "připevnit" Emitter k objektu, z kterého chceme, aby kouř vycházel. To uděláme tak, že přiřadíme Emitter jako potomka k transformaci onoho objektu.

```
XForm->addChild(SmokeEmitter);
```

Výsledný efekt by měl být zhruba následující.



Obrázek B.8.1: Screenshot z osmého tutoriálů

B.9 Billboardy

Tento díl bude pojednávat o billboardech, ale vzhledem triviálnosti tohoto tématu bude celkem krátký. Billboardy jsou objekty, jejichž orientace se mění tak, aby byly stále natočeny směrem k pozorovateli. Je to technika vhodná pro simulaci objektů, jejž se mají tvářit trojrozměrně, přestože trojrozměrné nejsou. Například se tím celkem dobře simulují stromy, záře světel a v dobách dávno minulých například i nepřátele (vzpomeňte na Duke3D). V tomto díle si ukážeme jak vytvořit skupinku stromů.

Jako první si musíme vytvořit třídu osg::Billboard a přidat ji do scény.

```
TreeBillBoard = new osg::Billboard();
root->addChild(TreeBillBoard);

// nastavíme podle které osy se má billboard otacet a nastavíme
// normalovy vektor
TreeBillBoard->setMode(osg::Billboard::AXIAL_ROT);
TreeBillBoard->setAxis(osg::Vec3(0.0f,0.0f,1.0f));
TreeBillBoard->setNormal(osg::Vec3(0.0f,-1.0f,0.0f));
```

Následně nastavíme texturu, která bude reprezentovat tento billboard, průhlednost a vypneme světla.

```
// vytvorime texturu stromu
osg::Texture2D *texture = new osg::Texture2D;
texture->setImage(osgDB::readImageFile("Images/tree0.rgba"));

// nastavime, aby se mista s alphou mensi nez 0.05 nezobrazovala
osg::AlphaFunc* alphaFunc = new osg::AlphaFunc;
alphaFunc->setFunction(osg::AlphaFunc::GEQUAL,0.05f);

osg::StateSet* StateSet = new osg::StateSet;

// vypneme svetla
StateSet->setMode( GL_LIGHTING, osg::StateAttribute::OFF );
// zapneme texturu
StateSet->setTextureAttributeAndModes(0, texture,
                                    osg::StateAttribute::ON );
// nastavime pruhlednost
StateSet->setAttributeAndModes(new osg::BlendFunc( GL_ONE,
            GL_ONE_MINUS_SRC_ALPHA ), osg::StateAttribute::ON );
```

```
StateSet->setAttributeAndModes( alphaFunc, osg::StateAttribute::ON );
StateSet->setRenderingHint( osg::StateSet::TRANSPARENT_BIN );
TreeBillBoard->setStateSet(StateSet);
```

V tuto chvíli máme hotový billboard a jediné co musíme udělat je přidání několika stromů. To uděláme pomocí funkce AddTree, kterou Vám ukážu později.

```
AddTree(osg::Vec3(10,10,-6));
AddTree(osg::Vec3(100,32,-6));
AddTree(osg::Vec3(76,54,-6));
AddTree(osg::Vec3(45,67,-6));
AddTree(osg::Vec3(-42,54,-6));
AddTree(osg::Vec3(21,87,-6));
AddTree(osg::Vec3(67,98,-6));
AddTree(osg::Vec3(-54,-34,-6));
AddTree(osg::Vec3(-65,-72,-6));
AddTree(osg::Vec3(-75,25,-6));
AddTree(osg::Vec3(-34,-94,-6));
AddTree(osg::Vec3(23,15,-6));
AddTree(osg::Vec3(15,-64,-6));
AddTree(osg::Vec3(64,14,-6));
AddTree(osg::Vec3(41,-65,-6));
```

Zde je výše zmíněná funkce AddTree. Jedná se o primitivní funkci, jež vytváří na určených souřadnicích obdélník o zadané výšce a šířce.

```
void cTerrain::AddTree(osg::Vec3 pos)
{
	// rozměry stromu
	float width = 9.5f;
	float height = 18.0f;

	osg::Geometry* Tree = new osg::Geometry;

	// nastaveni vrcholu čtverce
	osg::Vec3Array* Verts = new osg::Vec3Array(4);
	(*Verts)[0] = osg::Vec3(-width/2.0f, 0, 0);
	(*Verts)[1] = osg::Vec3( width/2.0f, 0, 0);
	(*Verts)[2] = osg::Vec3( width/2.0f, 0, height);
	(*Verts)[3] = osg::Vec3(-width/2.0f, 0, height);
```

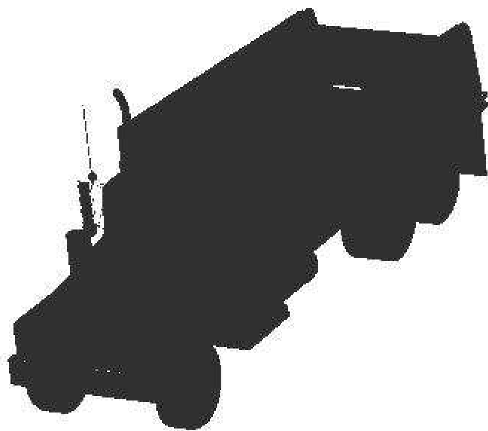
```
    Tree->setVertexArray(Verts);

    // nastaveni texturovych koordinatu
    osg::Vec2Array* TexCoords = new osg::Vec2Array(4);
    (*TexCoords)[0].set(0.0f,0.0f);
    (*TexCoords)[1].set(1.0f,0.0f);
    (*TexCoords)[2].set(1.0f,1.0f);
    (*TexCoords)[3].set(0.0f,1.0f);
    Tree->setTexCoordArray(0,TexCoords);

    Tree->addPrimitiveSet(new
          osg::DrawArrays(osg::PrimitiveSet::QUADS,0,4));

    // pridani stromu do billboardu na zadane pozici
    TreeBillBoard->addDrawable(Tree, pos);
};
```

A takhle to vypadá pokud se Vám to vše podaří.

Obrázek B.9.1: Screenshot z devátého tutoriálu.

B.10 Shadow Texture

V této poslední části tutoriálů o osg se budeme zabývat stíny. Metod pro vytváření stínů v počítačové grafice je povícero, ale ty opravdu důležité jsou shadow mapping a shadow volume. V tomto díle budu popisovat variantu shadow mappingu, která se jmenuje shadow texture. Princip této metody je v tom, že si vykreslíte objekt z pohledu světla do textury. Pozadí textury musí být bílé a objekt je vykreslen barvou, kterou má mít stín. Takto získanou texturu namapujete na scénu.



K vytvoření stínu ve scéně použijeme následující funkci. Je to funkce, do níž pošlete jako parametr ukazatel na objekt, který vrhá stín, ukazatel na kořenový uzel, na objekt na nějž je vrhán stín a pozici světla.

```
void CreateShadowedScene(osg::Group * root,osg::Node* shadower,osg::Geode*
shadowed,const osg::Vec3& lightPosition)
{
      // rozliseni textury
      unsigned int tWidth = 512;
      unsigned int tHeight = 512;

      // vytvoreni textury
      osg::Texture2D* texture = new osg::Texture2D;
      texture->setTextureSize(tWidth, tHeight);
      texture->setInternalFormat(GL_RGB);
      // nastaveni filtrovani
      texture->setFilter(osg::Texture2D::MIN_FILTER,
                                osg::Texture2D::LINEAR);
```

```
texture->setFilter(osg::Texture2D::MAG_FILTER,
                        osg::Texture2D::LINEAR);
texture->setWrap(osg::Texture2D::WRAP_S,
                        osg::Texture2D::CLAMP_TO_BORDER);
texture->setWrap(osg::Texture2D::WRAP_T,
                        osg::Texture2D::CLAMP_TO_BORDER);
texture->setBorderColor(osg::Vec4(1.0f,1.0f,1.0f,1.0f));

// nastaveni svetla
osg::Vec4 ambientLightColor(0.2,0.2f,0.2f,1.0f);

// vytvoreni kamerovaho uzlu, jenz bude slouzit ke snimani sceny
// z pohledu svetla
osg::CameraNode* camera = new osg::CameraNode;

// barva pozadi textury
camera->setClearColor(osg::Vec4(1.0f,1.0f,1.0f,1.0f));

// nastaveni viewportu na velikost textury
camera->setViewport(0,0,tWidth,tHeight);

// scena z pohledu svetla se bude vykreslovat před vykreslenim
// hlavni sceny
camera->setRenderOrder(osg::CameraNode::PRE_RENDER);

camera->setRenderTargetImplementation(
                              osg::CameraNode::FRAME_BUFFER_OBJECT);

// prirazeni obrazu z kamery do textury
camera->attach(osg::CameraNode::COLOR_BUFFER, texture);

// pridani objektu vrhajiciho stin
camera->addChild(shadower);

osg::StateSet* stateset = camera->getOrCreateStateSet();

// nastaveni materialu tak, aby vykresleny objekt vypadal jako stin.
osg::Material* material = new osg::Material;
material->setAmbient( osg::Material::FRONT_AND_BACK,
                                    osg::Vec4(0.0f,0.0f,0.0f,1.0f));
material->setDiffuse(osg::Material::FRONT_AND_BACK,
```

```
                                    osg::Vec4(0.0f,0.0f,0.0f,1.0f));
material->setEmission(osg::Material::FRONT_AND_BACK,
                              ambientLightColor);
material->setShininess(osg::Material::FRONT_AND_BACK,0.0f);
stateset->setAttribute(material,osg::StateAttribute::OVERRIDE);

root->addChild(camera);
```

TexGenNode je objekt, jenž nám umožňuje mapovat texturu z pohledu světla automaticky pomocí GL_TEXTURE_GEN.

```
osg::TexGenNode* texgenNode = new osg::TexGenNode;
// nasavíme texgen uzel na generovani tex. koordinatu pro texturu
// stinu
texgenNode->setTextureUnit(2);
root->addChild(texgenNode);
```

Nastavíme třídu, která bude updatovat kameru a TexGenNode. Tuto třídu si ukážeme později.

```
root->setUpdateCallback(new UpdateCameraAndTexGenCallback(
                         lightPosition, camera, texgenNode));
```

A jako poslední při vytváření stínu nastavíme stínovaný objekt na automatické generování texturových koordinátů, což díky TexGenNode správně umístí texturu stínu.

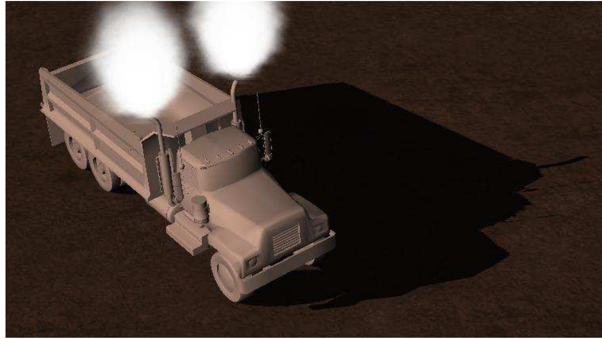
```
    {
      stateset = shadowed->getOrCreateStateSet();
      stateset->setTextureAttributeAndModes(2, texture,
                                    osg::StateAttribute::ON);
      stateset->setTextureMode(2, GL_TEXTURE_GEN_S,
                                    osg::StateAttribute::ON);
      stateset->setTextureMode(2, GL_TEXTURE_GEN_T,
                                    osg::StateAttribute::ON);
      stateset->setTextureMode(2, GL_TEXTURE_GEN_R,
                                    osg::StateAttribute::ON);
      stateset->setTextureMode(2, GL_TEXTURE_GEN_Q,
                                    osg::StateAttribute::ON);
    }
}
```

Tak v tuto chvíli máme vytvořenu funkci pro generování stínu v osg. Nicméně k tomu, aby se nám stín správně aktualizoval musíme ještě vytvořit update třídu, kterou jsme si před chvílí registrovali. Tato třída musí dědit ze třídy NodeCallBack a musí obsahovat operátor (), který obsahuje kód pro update.

```
class UpdateCameraAndTexGenCallback : public osg::NodeCallback
{
    public:

        UpdateCameraAndTexGenCallback(const osg::Vec3& position,
osg::CameraNode* cameraNode, osg::TexGenNode* texgenNode):
            _position(position),
            _cameraNode(cameraNode),
            _texgenNode(texgenNode)
        {
        }
        virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
        {
            // nejdrive presuneme potomky na sva mista
            traverse(node,nv);

            // nyni vypocitame matice kamery tak, aby ukazovali na shadower
            // (objekt vrhajici stin)
            osg::BoundingSphere bs;
            for(unsigned int i=0; i<_cameraNode->getNumChildren(); ++i)
            {
                bs.expandBy(_cameraNode->getChild(i)->getBound());
            }

            float centerDistance = (_position-bs.center()).length();

            float znear = centerDistance-bs.radius();
            float zfar  = centerDistance+bs.radius();
            float zNearRatio = 0.001f;
            if (znear<zfar*zNearRatio) znear = zfar*zNearRatio;

            float top   = (bs.radius()/centerDistance)*znear;
            float right = top;

            _cameraNode->setReferenceFrame(osg::CameraNode::ABSOLUTE_RF);
```

```
            _cameraNode->setProjectionMatrixAsFrustum( -right, right, -
top,
                                                                          top, znear, zfar);
            _cameraNode->setViewMatrixAsLookAt( _position, bs.center(),
                                                        osg::Vec3( 0.0f, 1.0f, 0.0f));

            // vypocitame matici, která prepocitava lokalni koordinaty na
            // koorinaty texury
            osg::Matrix MVPT = _cameraNode->getViewMatrix() *
                               _cameraNode->getProjectionMatrix() *
                               osg::Matrix::translate(1.0,1.0,1.0) *
                               osg::Matrix::scale(0.5f,0.5f,0.5f);

            // a nastavime TexGenNode pomoci této matice
            _texgenNode->getTexGen()->setMode(osg::TexGen::EYE_LINEAR);
            _texgenNode->getTexGen()->setPlanesFromMatrix(MVPT);

        }

    protected:

        virtual ~UpdateCameraAndTexGenCallback() {}

        osg::Vec3                      _position;
        osg::ref_ptr<osg::CameraNode> _cameraNode;
        osg::ref_ptr<osg::TexGenNode> _texgenNode;

};
```

Nyní stačí vložit do funkce CreateShadowedScene objekt nebo uzel, který chcete aby vrhal stín a objekt(uzel) na který se ten stín má promítat. Vše ostatní se již provede samo.


Obrázek B.10.1: Screenshot z posledního dílu tutoriálů

Seznam příloh

Příloha A. Grafické rozhraní

Příloha B. Tutoriály