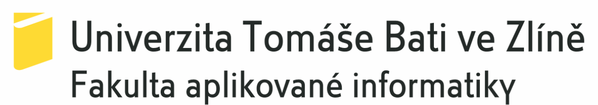
Vizualizační 3D grafický engine

Bc. Jan Štalmach

Diplomová práce
2006



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav aplikované informatiky
akademický rok: 2005/2006

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jan ŠTALMACH**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Vizualizační 3D grafický engine**

Zásady pro vypracování:

1. Vypracujte literární rešerši o 3D zobrazování v reálném čase.
2. Popište moderní, v současné době nejpoužívanější programové prostředky pro 3D vizualizaci (OpenGL, GLSL, Direct3D, HLSL, VRML, X3D).
3. Charakterizujte moderní hardwarové prostředky pro 3D vizualizaci (3D grafické akcelerátory a jejich architektura).
4. Navrhněte 3D vizualizační grafický engine s podporou grafové reprezentace scény, dynamického stínování, shaderingu, částicových efektů, detekce kolizí, maticové transformace a quaternionů.
5. Tento vizualizační engine podrobně charakterizujte a proveďte softwarovou realizaci s využitím knihoven OpenGL.
6. Vytvořte multiplatformní aplikaci pro 3D vizualizaci.

Rozsah práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. Kessenich, J., Baldwin, D., Rost, R. The OpenGL Shading Language. 3Dlabs, Inc. Ltd., 2004
2. Martišek, D. Matematické principy grafických systémů. Litera, Brno, 2002, ISBN 80-85763-19-2
3. Sagal, M., Akeley, K. The OpenGL Graphics System: A Specification (Version 2.0). Silicon Graphics, Inc., 2004
4. Virius, M. Programování v C++. ČVUT, Praha, 2001, ISBN 80-01-01874-1
5. Žára, J. a kol. Počítačová grafika - principy a algoritmy. Grada, Praha, 1992, ISBN 80-85623-00-5
6. Žára, J., Beneš, B, Sochor, J. a Felkel, P. Moderní počítačová grafika. Computer Press, Brno, 2004, ISBN 80-251-0454-0

Vedoucí diplomové práce: **Ing. Pavel Pokorný, Ph.D.**
Ústav aplikované informatiky

Datum zadání diplomové práce: **14. února 2006**

Termín odevzdání diplomové práce: **26. května 2006**

Ve Zlíně dne 14. února 2006

prof. Ing. Vladimír Vašek, CSc.
pověřený děkan

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

ABSTRAKT

Tato práce se zabývá principy a C++ implementací 3D grafického enginu v OpenGL. Engine je naprogramován jako multiplatformní aplikace pro interaktivní zobrazování a vizualizaci trojrozměrných dat. Základem reprezentace scény je hierarchická datová grafová struktura Oktalový strom s pohledovým ořezáváním. Technika stínových objemů umožňuje zobrazovat dynamické stíny. Programovací jazyk The OpenGL shading language dovoluje využívat moderní programovatelné vertex a fragment shadery.

Klíčová slova:

Počítačová grafika, 3D engine, Oktalový strom, Dynamické stíny, Shadering, OpenGL, GLSL

ABSTRACT

This thesis deals with the principles and C++ implementation of 3D graphics engine in OpenGL. Engine is programmed as multiplatform application for real time interactive rendering and visualization. Scenegraph is based on hierarchical data structure octal tree with frustum culling. Dynamic shadows are produced by algorithm shadow volumes. The OpenGL Shading Language provides support for vertex and fragment shaders.

Keywords:

Computer graphics, 3D engine, Octal tree, Dynamic shadows, Shadering, OpenGL, GLSL

Velký dík patří především vedoucímu diplomové práce Ing. Pavlu Pokornému, Ph.D., který mě umožnil realizovat tento projekt a věcně připomínkoval průběh tvorby softwarové aplikace. Díky Miriam Ševčíkové za pečlivé pročtení tohoto textu.

Děkuji také Ing. Miroslavu Topolánkovi a jeho nadaci Becario za poskytnutí prospěchového stipendia, které mě umožnilo financovat hardware, na kterém jsem engine vyvíjel.

Dále nesmím zapomenout poděkovat všem betatesterům, kteří mě pomáhali s odhalováním chyb a logických nepřesností v programu.

Můj velký obdiv a poklona patří také firmě Silicon Graphics za OpenGL, firmě 3Dlabs za The OpenGL Shading Language a dále všem na jejichž základě jsem mohl tento projekt realizovat.

Všechny zájemce rád uvítám na svých www stránkách http://stalmach.wz.cz nebo emailu stalmachjan@seznam.cz.

OBSAH

ÚVOD ... 8

I TEOERETICKÁ ČÁST ... 9

1 TROJROZMĚRNÉ POČÍTAČOVÉ ZOBRAZOVÁNÍ ... 10

1.1 GLOBÁLNÍ ZOBRAZOVACÍ METODY ... 10

1.2 ZOBRAZOVÁNÍ V REÁLNÉM ČASE ... 11

1.2.1 Plošková reprezentace dat ... 11

1.2.2 Phongův osvětlovací model ... 12

1.2.3 Zobrazování scény ... 13

2 PROGRAMOVÉ PROSTŘEDKY PRO 3D ZOBRAZOVÁNÍ ... 14

2.1 OPENGL 2.0 ... 14

2.2 DIRECT3D 9.0C ... 15

2.3 JAZYKY PRO PROGRAMOVATELNÉ GRAFICKÉ PROCESORY ... 16

2.3.1 The OpenGL Shading Language ... 16

2.3.2 High Level Shading Language ... 17

2.3.3 Cg ... 17

2.4 JAZYKY PRO POPIS VIRTUÁLNÍ REALITY ... 17

2.4.1 VRML ... 17

2.4.2 X3D ... 18

3 MODERNÍ GRAFICKÉ AKCELERÁTORY A JEJICH ARCHITEKTURA ... 19

3.1 HISTORICKÝ VÝVOJ 3D GRAFICKÝCH KARET PRO IBM PC ... 19

3.2 MODERNÍ PROGRAMOVATELNÝ GPU ... 20

3.3 ARCHITEKTURA PROGRAMOVATELNÉHO GRAFICKÉHO AKCELERÁTORU ... 20

3.4 VERTEXOVÝ PROCESOR ... 22

3.4.1 Vertexový procesor v pevném režimu ... 23

3.4.2 Vertexový procesor v programovatelném režimu ... 25

3.5 FRAGMENTOVÝ PROCESOR ... 25

3.5.1 Fragmentový procesor v pevném režimu ... 26

3.5.2 Fragmentový procesor v programovatelném režimu ... 29

3.6 VYUŽITÍ GRAFICKÉHO ČIPU PRO OBECNÉ VÝPOČTY ... 29

4 EFEKTIVNÍ REPREZENTACE SCÉNY ... 32

4.1 POMOCNÉ DATOVÉ STRUKTURY ... 32

4.2 HIERARCHIE OBÁLEK ... 33

4.3 DĚLENÍ PROSTORU ... 33

4.3.1 Pravidelná mřížka ... 34

4.3.2 BSP strom ... 34

4.3.3 k-D strom ... 34

4.3.4 Oktalový strom ... 35

5 DYNAMICKÉ STÍNY **39**

5.1 Základní pojmy 39

5.1.1 Vlastní a vržený stín 39

5.1.2 Měkký a ostrý stín 39

5.2 Metody pro zobrazování dynamických stínů 40

5.2.1 Projekční metoda 40

5.2.2 Stínová paměť hloubky 41

5.2.3 Stínové objemy 41

5.3 Technika stínových objemů 42

5.3.1 Určení siluety stínícího objektu 43

5.3.2 Vytvoření stínového objemu 44

5.3.3 Depth-pass algoritmus 46

6 SHADERING **48**

6.1 Shader Model 49

6.2 Shadering pomocí GLSL 50

II PRAKTICKÁ ČÁST **52**

7 REALIZACE 3D ENGINU V OPENGL **53**

7.1 Softwarové prostředky 54

7.1.1 wxWidgets 54

7.1.2 OpenGL 2.0 55

7.1.3 The OpenGL Shading Language 55

7.1.4 MS Visual Studio C++ .NET 55

7.2 Objektová struktura enginu 56

8 POPIS APLIKACE WXVIZUA 3D **57**

8.1 Dialogová okna 57

8.1.1 Dialog nového projektu 57

8.1.2 Dialog nastavení světel ve scéně 57

8.1.3 Dialog pro nastavení kamery 58

8.1.4 Dialog pro nastavení skyboxu 59

8.1.5 Dialog pro nastavení mlhy 59

8.1.6 Dialog pro nastavení vlastností objektu 60

8.2 ToolBar 61

8.3 Ovládání programu 61

8.4 Obsah CDROM 61

ZÁVĚR **62**

SEZNAM POUŽITÉ LITERATURY **63**

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK **66**

SEZNAM OBRÁZKŮ **67**

SEZNAM PŘÍLOH **69**

ÚVOD

Interaktivní trojrozměrná počítačová vizualizace probíhající v reálném čase zažívá v poslední době nebývalý rozvoj. Technologické možnosti dnešních programovatelných 3D grafických akcelerátorů dovolují u pokročilých grafických aplikací vytvářet komplexní trojrozměrné scény generované v reálném čase. Speciální grafická rozhraní zajišťují efektivní vývoj 3D enginu.

Práce se zabývá návrhem a implementací 3D grafického enginu v OpenGL 2.0. Základem aplikace je grafová hierarchická datová struktura oktalový strom s pohledovým ořezáváním (*octal tree with frustum culling*) umožňující efektivní zobrazování rozsáhlých scén. Engine disponuje podporou dynamických stínů za pomoci techniky stínových těles (*depth-pass stenciled shadows*), podporuje stínovací jazyk *The OpenGL Shading Language* (GLSL) a na pokročilých grafických akcelerátorech umožňuje vytváření speciálních efektů (*bump mapping, real glass,…*). Výsledná aplikace postavená na 3D enginu je multiplatformní software umožňující interaktivní vizualizaci architektonických řešení, prezentaci trojrozměrných dat a virtuální reality.

I. TEORETICKÁ ČÁST

1 TROJROZMĚRNÉ POČÍTAČOVÉ ZOBRAZOVÁNÍ

Téměř vše, s čím dnes uživatel na počítači pracuje, je grafické povahy. Grafická uživatelská rozhraní (GUI) se stala standardní komunikační součástí většiny počítačových systémů současnosti. Velký rozvoj dvourozměrné grafiky v osmdesátých a devadesátých letech minulého století přivedl grafická rozhraní téměř k dokonalosti. Nejnovějším trendem interaktivního zobrazování se stává trojrozměrná grafika. S příchodem cenově dostupných 3D grafických akcelerátorů a celkovým zvýšením výpočetního výkonu počítačů je tento krok logický. Nejočekávanější softwarový produkt současnosti, Microsoft Windows Vista, je již plně založen na interaktivní 3D počítačové grafice. Okna aplikací se kterými uživatel pracuje nejsou dvojrozměrné objekty, ale trojrozměrné polygonální modely, na které grafický akcelerátor mapuje textury.

Trojrozměrná počítačová grafika tvoří základ zobrazování CAD, virtuální reality, medicínských vizualizačních systémů, filmových a televizních triků, počítačových her, simulačních a vědeckých aplikací (molekulární modelování). Názorná vizualizace komplikovaných prostorových systémů je pomocí trojrozměrné interaktivní grafiky daleko lépe pochopitelná a vstřebatelná, než složitý systém dvourozměrných řezů a pohledů.

1.1 Globální zobrazovací metody

Z hlediska kvality grafického výstupu a rychlosti zobrazování trojrozměrné grafiky můžeme rozlišit dva základní směry, kterými se 3D počítačová grafika ubírá.

V první skupině je kladen maximální důraz na kvalitu zobrazované scény. Výpočetní složitost renderování takové scény je nesmírně náročná na strojový výkon. Nejpoužívanějšími postupy pro fotorealistické zobrazování jsou *globální zobrazovací metody* (*global illumination*).

Nejznámějším algoritmem je sledování paprsku (*ray tracing*) z roku 1980 a technika založená na řešení radiozitní rovnice – tzv. *radiozita* (1984-85). Tyto dvě metody (zvláště pak radiozitu) je v současnosti nemožné uspokojivě implementovat pro zobrazování v reálném čase. Výpočet je v případě složitých scén a velkém důrazu na kvalitu výstupu (filmové triky) řešen pomocí renderovací farmy, která představuje i stovky paralelně

propojených počítačů. Konečný výsledek je při použití kvalitních renderovacích rozhraní a standardů (např. *Pixar RenderMan*) nerozeznatelný od reálných fotografií nebo filmových záběrů. Více lze nalézt v [42].

Cílem globálních osvětlovacích technik, a ve své podstatě i cíl počítačové grafiky, je nalezení takového zobrazovacího algoritmu, který bude v reálném čase vytvářet scénu se všemi optickými a světelnými vlastnostmi jako tomu je ve skutečnosti. Dosažením tohoto stavu splní počítačová grafika své poslání.

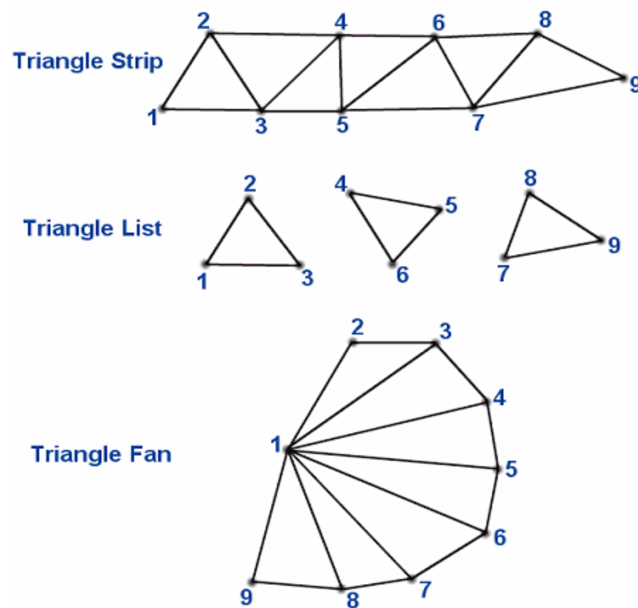
1.2 Zobrazování v reálném čase

Globální osvětlování dosahuje výborné grafické výsledky. Pro zobrazování v reálném čase se tyto postupy vzhledem k výpočetní náročnosti nehodí. Zobrazování scény v reálném čase se rozumí, generování alespoň 12 (25) kompletních scén za jednu sekundu (*frames per second*). Celá scéna musí být vygenerována a zaslána na zobrazovací jednotku za méně než 0.04 s.. Pro zobrazování v reálném čase nebo interaktivní grafiku proto používáme jiné renderovací techniky. Tyto algoritmy nejsou založeny na fyzikálním principu šíření a odrazu světla. Jednotlivé objekty ve scéně jsou vyhodnocovány nezávisle na ostatních. Každý objekt má individuálně (lokálně) vyhodnocen osvětlovací model tak, jako by byl ve scéně osamocen a přijímal světlo od všech světelných zdrojů. Ostatní objekty se berou v úvahu pouze při výpočtu viditelnosti.

1.2.1 Plošková reprezentace dat

Prostorová tělesa popisujeme nejčastěji pomocí ploškové reprezentace – polygonální sítě (*mesh*). Polygony volíme nejčastěji trojúhelníkové a to především díky vhodným vlastnostem. Trojúhelník je vždy konvexní, jeho vrcholy leží v rovině a výpočet průsečíků lze s jinými tělesy efektivně optimalizovat. Grafické akcelerátory mají hardwarové obvody navržené pro práci nad trojúhelníky. Trojúhelníky můžeme seskupovat do tzv. pásů (*triangle strip*) nebo vějířů (*triangle fan*), kdy pro renderování dalších trojúhelníků využíváme předchozí vypočítané vrcholy. Grafické akcelerátory a programová rozhraní podporují i práci se čtyřúhelníky (*quads*). Sítí myslíme takovou množinu jejíž polygony sdílí své hrany. Datová struktura popisující takovou síť je nejčastěji rozdělena do dvou částí – geometrické a topologické. V geometrické části jsou zaznamenány trojrozměrné

souřadnice tvořící vrcholy (*vertexes*); topologická část udržuje informace o tom, které vrcholy tvoří polygon.



Obr. 1. Pás (*strip*) a vějíř (*fan*) trojúhelníků

1.2.2 Phongův osvětlovací model

Pro výpočet odraženého světla v realtime generované počítačové grafice se nejčastěji používá empiricky odvozený systém nazvaný *Phongův osvětlovací model* navržený Bui-Tuong Phongem v roce 1977. Tento způsob výpočtu implementují OpenGL i Direct3D. Model rozlišuje tři druhy odrazu světla od materiálu. Složením těchto tří složek vzniká výsledný odraz. Rozlišujeme okolní (*ambient*), difúzní (*diffuse*) a odrazový (*specular*) odraz. Podrobnosti viz [42].

Phongův osvětlovací model je matematicky formulován:

$$I_V = I_a + \sum_{k=1}^{M} I_{L_k} \left[I_{s\,k} + I_{d\,k} \right], \tag{1}$$

kde

I_V	výsledná barva povrchu
I_a, I_d, I_s	okolní, difůzní a odrazová složka
M	počet světelných zdrojů

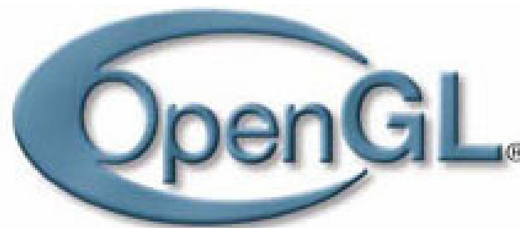
1.2.3 Zobrazování scény

Scénu reprezentovanou polygonální sítí nelze pouze jednoduše zaslat do zobrazovacího řetězce. Před samotným odesláním dat do akcelerátoru je třeba efektivně určit viditelné a neviditelné části a do dalšího zpracování propustit pouze potenciálně viditelné části scény. Scénu a její polygonální síť proto popisujeme hierarchickými datovými strukturami. Těmto algoritmům se věnuje kapitola 4. Na polygonech zaslaných do zobrazovacího řetězce jsou provedeny modelovací a projekční transformace. Poté jsou stínovány a mapovány texturou. Lokálně je vypočítán *Phongův osvětlovací model*. V poslední fázi jsou data rasterizována a zapsána do framebufferu.

2 PROGRAMOVÉ PROSTŘEDKY PRO 3D ZOBRAZOVÁNÍ

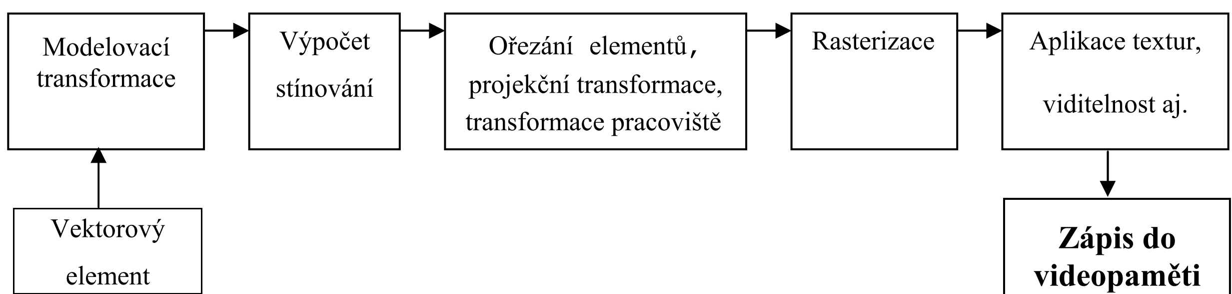
2.1 OpenGL 2.0

OpenGL [21], [22], [26] je grafické rozhraní pro definici a zobrazování 3D objektů. Knihovna OpenGL je na platformě nezávislá a lze ji provozovat na téměř jakémkoli operačním systému a počítačové architektuře současnosti. (Windows, Linux, UNIX, MacOS, SGI, …). Knihovna sama o sobě vznikla před více než dvaceti lety na počítačích Silicon Graphics (SGI) a od té doby se stále vyvíjí. Původní název na SGI byl IrisGL a pojmenování OpenGL dostala na počátku devadesátých let, kdy byla přenesena na platformu IBM PC.



Obr. 2. OpenGL

OpenGL pracuje na architektuře *klient-server* a využívá v maximální možné míře podporu akcelerátorů pracujících na principu *proudového zpracování dat* známého např. z architektury procesorů RISC. Proudové zpracování spočívá v rozdělení posloupnosti grafických operací do menších celků, které řeší speciálně navržený hardwarový obvod akcelerátoru. Výsledek je poté složen z těchto dílčích operací. Mezi typickou zpracovávací pipeline patří např. následují sekvence: *modelovací transformace → zobrazovací transformace → projekce a ořezání → transformace pracoviště → výsledný obraz*.



Obr. 3. Grafický zobrazovací řetězec OpenGL

Z programátorského hlediska se s OpenGL pracuje jako s množinou přibližně 200 funkcí. OpenGL se chová jako stavový automat, v daný moment může být celé renderovací rozhraní pouze ve stavu A nebo B (*např. syntaxe glEnable(GL_LIGHTING), glDisable(GL_LIGHTING) – buď se používá osvětlovací model nebo ne*). Pro pokročilou práci s OpenGL je nutné využívat tzv. *extensions* (*EXT_stencil_two_side, GL_ARB_multitexture,…*). Tyto rozšíření umožňují přistupovat ke speciálním grafickým hardwarovým funkcím akcelerátoru přes OpenGL. Funkcionalita těchto rozšíření je umístěna na grafickém akcelerátoru a pomocí speciální procedury je možné tyto rozšíření aktivovat. Počet extenzí grafické karty výrazně ovlivňuje možnosti zobrazování v OpenGL. Tyto extenze spravuje konsorcium *The OpenGL Architecture Review Board* (ARB), které vzniklo v roce 1992 a stará se o standardizaci těchto rozšíření. Současná verze OpenGL 2.0 je šestou reedicí původní specifikace OpenGL 1.0. OpenGL 2.0 přináší především podporu shaderingu pomocí The OpenGL Shading Language 1.10 [2.3.1], dále podporu pro více renderovacích cílů (*multiple render targets*) a separovatelný stencil buffer (využíváno v enginu).

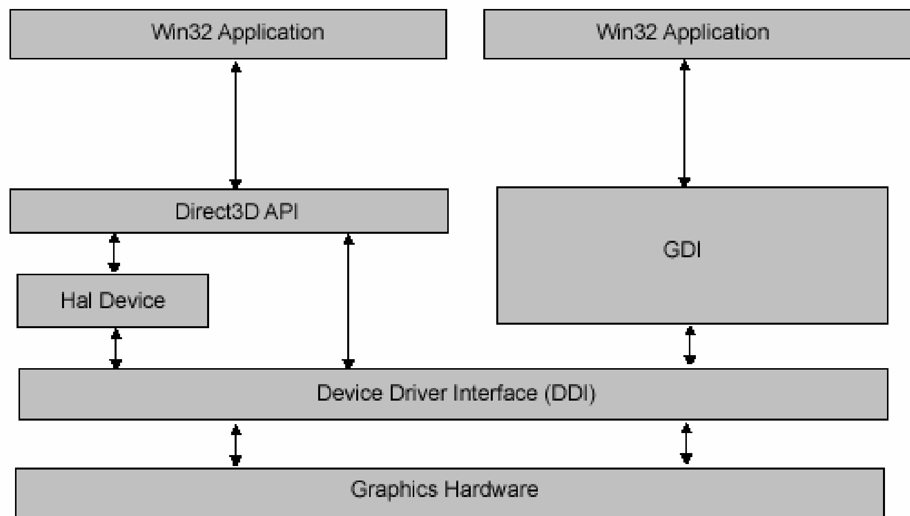
OpenGL 2.0 lze také provozovat na tzv. *embedded* systémech. Na této platformě hovoříme o OpenGL|ES (*OpenGL for Embedded Systems*).

2.2 Direct3D 9.0c

Direct3D [6], [36] (*DirectX Graphics*) je programové rozhraní pro vytváření trojrozměrné počítačové grafiky. Tento interface vyvinul Microsoft (první verze 1995), aby ulehčil vývojářům počítačových her a i ostatním programátorům vytváření aplikací. Direct3D je součástí balíku DirectX, který poskytuje ucelený API pro vývoj multimediálních aplikací pod operačním systémem Windows. V současné době je k dispozici verze DirectX 9.0c (*build April 2006*).

Operační systém Windows představuje z hlediska programátora několik vrstev funkcionalit, z nichž každá je reprezentována určitou knihovnou funkcí. Direct3D je další takovou vrstvou nacházející se mezi aplikací Windows a grafickým hardwarem podobně jako standardní grafická knihovna systému Windows, GDI (*Graphics Device Interface*). Aplikace v Direct3D dosahuje nezávislosti na vrstvě HAL (*Hardware Abstraction Layer*). Výrobci grafických karet dodávají svou vlastní vrstvu HAL, která implementuje ty funkce

definované v Direct3D, které je daný hardware schopný provádět. Funkce, které akcelerátor neumí vykonávat, je možné softwarově emulovat (značně pomalé).



Obr. 4. Hierarchie vrstev aplikace v Direct3D

Aplikace v Direct3D ve Windows (Obr. 4) komunikuje s vrstvou Direct3D, která komunikuje s vrstvou HAL nebo DDI (*Device Driver Interface*). DDI komunikuje přímo s grafickým hardwarem. Pro srovnání je v pravé části (Obr. 4) schéma komunikace Win32 aplikace přes standardní rozhraní GDI. Přehledný úvod do Direct3D lze najít v [36].

2.3 Jazyky pro programovatelné grafické procesory

2.3.1 The OpenGL Shading Language

The OpenGL Shading Language [18] (*GLSL, glslang*) je high level programovací jazyk, který je součástí specifikace standardu OpenGL 2.0. GLSL je určen pro přímé programování jádra moderního GPU. Tento programovací jazyk dovoluje obejít nativní funkci akcelerátoru v tzv. pevném režimu (*fixed functionality*) a nahradit ji režimem programovatelným, kdy se grafický procesor ovládá pomocí kódu napsaného v GLSL. Takovému zdrojovému kódu říkáme shader. GLSL navrhla společnost 3Dlabs, Inc. v roce 2004. Základní stavební kameny jazyka stojí na třech inspiracích. První je jazyk C/C++, druhá je grafický standard Pixar (Apple) RenderMan a třetí je OpenGL. Engine pro shadering používá právě tento jazyk.

2.3.2 High Level Shading Language

High Level Shading Language (HLSL) [6] je jazyk pro programovatelné grafické procesory běžící pod OS Windows. HLSL je nativně podporován API Direct3D 9. Syntaxe jazyka je odvozená od C/C++.

2.3.3 Cg

Jde o stínovací jazyk navržený firmou NVIDIA. *Cg* (*C for graphics*) pracuje pouze na grafických kartách NVIDIA. *Cg* shader lze spustit pod Direct3D i OpenGL.

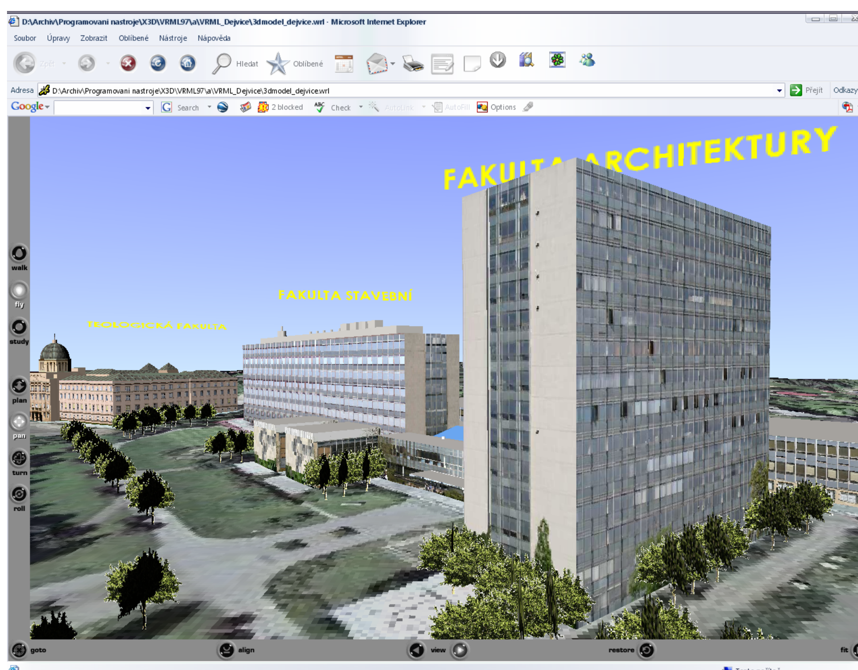
2.4 Jazyky pro popis virtuální reality

2.4.1 VRML

Jazyk VRML [34] (*Virtual Reality Modeling Language*) je mezinárodně uznávaný formát pro popis virtuální reality. Tento jazyk je v současné době standardizován jako VRML 97 (VRML 2.0) ISO/IEC 14772 [35]. Původní specifikaci VRML 1.0 navrhla v roce 1995 firma *Silicon Graphics* na základě knihovny pro práci s 3D objekty nazvané *OpenInventor*.

Obr. 5. VRML97

Prostorový virtuální svět, popisovaný jazykem VRML, je reprezentován hierarchickou datovou strukturou nazvanou zhroucený strom (*collapsed tree*). Virtuální svět je tvořen několika stromy – lesem. Stromová struktura je konstruována z uzlů, které jsou základními stavebními prvky jazyka VRML. Norma specifikuje 54 základních uzlů, další mohou být uživatelsky doprogramovány. Více o programování ve VRML lze nalézt v [10] a [40].



Obr. 6. VRML v internetovém prohlížeči

2.4.2 X3D

X3D [34], [9] (*eXtensible 3D*) je nástupce jazyka VRML, standard pro XML popis 3D objektů. X3D vytvořilo konsorcium Web3D, které sdružuje nejvýznamnější instituce a firmy z oblasti počítačové grafiky. Je založeno na komponentové technologii, která umožňuje vytváření tzv. profilů definující dále požadované vlastnosti.

Obr. 7. X3D

Reprezentace scény (*scene graph*) je architektura založená na hierarchické datové struktuře orientovaného acyklického grafu. X3D popisuje grafická 3D data, zvuková 3D data, video (MPEG-4) a hypertext. Základem je komponenta 3D runtime a postupným rozšiřováním o nové komponenty schopnosti X3D narůstají.

X3D je standardizován jako několik ISO/IEC norem. Nejnovější normy popisující některé části tohoto standardu jsou v současné době světové novinky pocházející teprve z dubna 2006 (*např. ISO/IEC 19775:2004/FDAM Am1:2006*).

3 MODERNÍ GRAFICKÉ AKCELERÁTORY A JEJICH ARCHITEKTURA

Postupným vývojem grafických akcelerátorů od nejjednodušších, které podporovaly pouze práci s plošnými rastrovými nebo vektorovými daty, jsou dnešní plně programovatelné grafické čipy nejvyspělejšími procesorovými architekturami současnosti. Masivní podpora paralelismu, rychlé datové sběrnice a paměti předurčují využití akcelerátorů nejenom pro grafiku, ale i pro obecné matematické výpočty.

3.1 Historický vývoj 3D grafických karet pro IBM PC

Na platformě IBM PC se první grafické karty s obvody podporujícími 3D výpočty objevují v polovině devadesátých let. Tyto akcelerátory však nelze zatím srovnávat s produkty pro počítače SGI. Výrobcem těchto 3D karet byla především firma ATI (ATI 3D Pro Turbo,…). Výraznější úspěch na platformě PC zaznamenala až karta firmy 3Dfx – Voodoo. Tato cenově dostupná karta zvládala na platformě PC jako první uspokojivě v reálném čase používat metodu *bilineárního filtrování* textur. Další vývoj grafických karet se poměrně dlouhou dobu nesl ve znamení 3Dfx. Kolem roku 2000 začíná nadvládu nad grafickými kartami získávat zpět ATI a firma NVIDIA (založená lidmi z SGI).

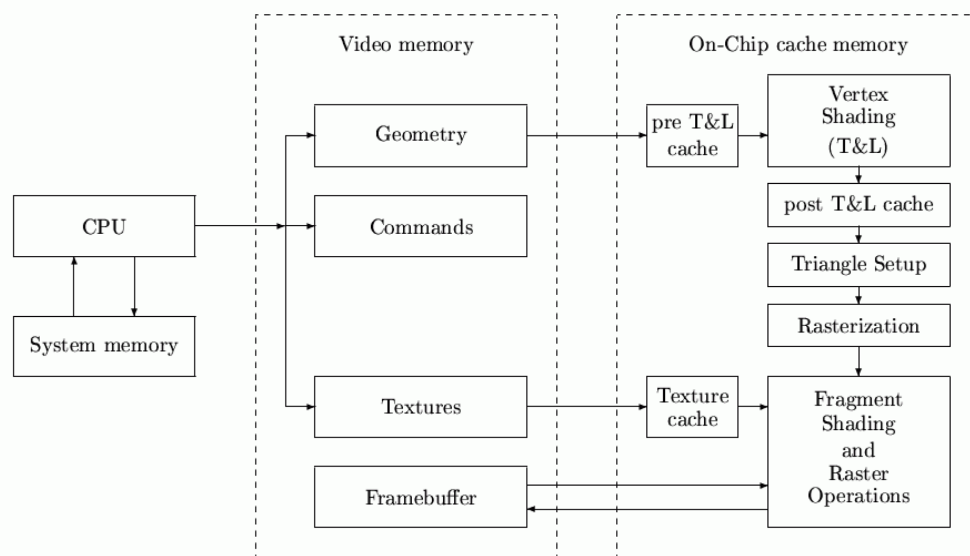
Uvedení grafických karet NVIDIA GeForce 1 a NVIDIA GeForce 2 můžeme považovat za začátek éry rozvoje programovatelných GPU. Až do té doby bylo možné komunikovat s grafickou kartou pouze pomocí programátorského rozhraní poskytující neměnnou sadu funkcí, kterou bylo možné programově využívat. Nebylo však možné funkčnost grafických akcelerátorů libovolně měnit či rozšiřovat jejich definované schopnosti. Postupným vývojem akcelerátorů se začala kromě jejich hrubého výpočetního výkonu zvyšovat i jejich flexibilita, která začala umožňovat měnit nativní funkci grafického procesoru pro operace, které nebyly součástí standardní výbavy. Tato první generace programovatelných GPU obsahovala operace pro úpravu vykreslovaných pixelů, resp. *texelů* skrze tzv. *registers-cambiners*, pomocí nichž bylo možné naprogramovat jednodušší rastrové operace prováděné se zapisovanými fragmenty.

3.2 Moderní programovatelný GPU

Současné moderní 3D grafické akcelerátory lze již plně programovat pomocí speciálních programových kódů, které můžeme do grafického procesoru nahrávat i za běhu programu. Podle typu zpracovávaných dat je možné rozlišit dvě základní programovatelné jednotky grafických karet: vrcholový procesor (*vertex processor*) a rasterizační procesor (*fragment processor*).

Obě jednotky pracují ve dvou režimech. Standardně beží v režimu pevném (*fixed functionality*), kdy se grafický procesor ovládá pomocí kódu napevno zapsaného v paměti akcelerátoru (např. stínování, transformace, texturování,…). Druhým módem, ve kterém dovedou tyto jednotky pracovat, je režim programovatelný, kdy se grafický procesor ovládá pomocí externě nahraného programu, který mění nativní funkci procesoru k libovolným uživatelsky definovaným operacím. V rámci jedné grafické aplikace lze tyto stavy libovolně kombinovat. Určitým problémem je pouze přepínání kontextu mezi pevným a programovatelným režimem. Výhledově se proto uvažuje pouze o práci v programovatelném režimu.

3.3 Architektura programovatelného grafického akcelerátoru



Obr. 8. Blokové schéma moderního grafického akcelerátoru [30]

Význam jednotlivých bloků uvedených na (Obr. 8) je následující:

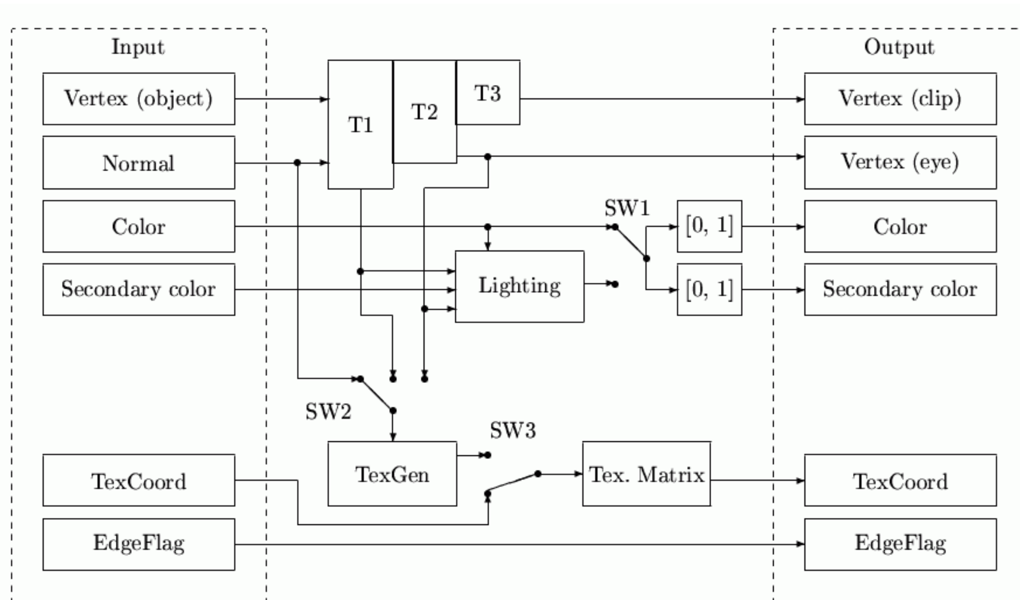
1. **CPU** – procesor
2. **System memory** - systémová paměť. Zdroj dat. Současné grafické akcelerátory nemohou přistupovat k jiným datovým zdrojům (disk, ethernet).
3. **Video memory** - paměť, která je umístěna na grafickém akcelerátoru
   3.1. **Geometry** - blok video paměti kde jsou uložena geometrická data těles
   3.2. **Commands** – část video paměti určené pro *vertex shader* a *fragment shader*.
   3.3. **Textures** - video paměť sloužící k uložení rastrových textur
   3.4. **Framebuffer** - do této oblasti video paměti ukládá grafický procesor jednotlivé vykreslované fragmenty. Část této paměti, která se nazývá barvový buffer (*color buffer*), může být vykreslena na obrazovku. Další části (například *depth buffer* či *stencil buffer*) nejsou pro uživatele přímo viditelné, ale slouží k realizaci různých grafických algoritmů.
4. **Vertex Shading** - blok, ve kterém se pomocí vertex shaderu provádí transformace geometrických dat (mezi geometrická data patří pozice vrcholů v prostoru, normály plošek, pozice i orientace světel a souřadnice v textuře). Standardní program provádí pohledovou transformaci, perspektivní projekci a následnou transformaci do obrazovkových souřadnic. Pro texturové koordináty je použita samostatná jednotka pro provádění transformací.
5. **Pre T&L Cache** - oblast paměti umístěné na grafickém procesoru, do níž se ukládají často používané geometrické informace, které vstupují do vertex shaderu. Organizace této paměti a způsob jejího využití je obdobný cache pamětem používaných u obecných mikroprocesorů.
6. **Post T&L Cache** - oblast paměti na grafickém procesoru, do které se ukládají geometrické informace již zpracované vertex shaderem, tj. po aplikaci naprogramovaných transformací.
7. **Texture Cache** - paměť, ve které jsou uloženy často používané textury či jejich části.

8. **Triangle setup** - pokud vykreslovaná oblast nemá tvar trojúhelníku (například vlivem ořezání tělesem záběru či ořezávacími rovinami), musí se provést *tesselace*, tj. opětovné rozložení složitějšího polygonu na trojúhelníky.
9. **Rasterization** - rasterizační procesor.
10. **Fragment shading and Raster Operations** - modul provádějící obarvení fragmentů. Vstupem jsou fragmenty reprezentující vykreslovaný polygon a rastrová data textury; výstupem je fragment pokrytý texturou, který je posléze uložen do *framebufferu*. Při některých operacích (blending, renderování do textury) lze z *framebufferu* jednotlivé fragmenty také číst, datový tok je oboustranný.

3.4 Vertexový procesor

Vertex shader neboli *vertexový procesor* je zpracováván v programovatelné jednotce, která je umístěna přímo na programovatelném jádru GPU. Pomocí této jednotky lze přeprogramovat transformační řetězec (*transformation pipeline*). Všechny geometrické údaje o jednotlivých vrcholech renderovaných polygonů procházejí před rasterizací tímto transformačním řetězcem. Mezi další funkce prováděné vertex shaderem patří operace se světly a jejich orientací, výpočet texturových koordinát, projekční a modelovací transformace.

3.4.1 Vertexový procesor v pevném režimu



Obr. 9. Transformace prováděné vertex shaderem v pevném režimu [30]

Význam jednotlivých bloků v pevném režimu zobrazených na (Obr. 9):

1. **Vertex (object)** - souřadnice vrcholu vykreslovaného polygonu, které jsou specifikovány ve světových souřadnicích
2. **Vertex (clip)** - souřadnice vrcholu vykreslovaného polygonu, které jsou v blocích **T1** až **T3** transformovány do normovaných souřadnic.
3. **Vertex (eye)** - souřadnice vrcholu plošky, které jsou transformovány do okna záběru.
4. **Normal** - normála vztažená k celému polygonu nebo ke každému vrcholu zvlášť. Řízení způsobu výpočtu texturovacích souřadnic naznačují přepínače **SW2** a **SW3**.
5. **Color** - barva vztažená k celému polygonu nebo každému vrcholu zvlášť. Barva je reprezentována čtveřicí složek R, G, B, A.
6. **Secondary color** - sekundární barva určená pro blending, modulaci textur a aplikaci efektu mlhy.

7. **TexCoord** - souřadnice v textuře. Tyto souřadnice jsou buď pro každý vrchol explicitně zadány, nebo mohou být automaticky vypočteny v bloku **TexGen**. Volba se provádí programovým přepínačem **SW3**.

8. **EdgeFlag** - příznak, kterým je specifikováno, zda se má příslušná hrana polygonu vykreslit.

9. **T1-T3** - bloky provádějící transformaci podle tří transformačních matic (pohledová transformační matice, projekční transformační matice a inverzní pohledová transformační matice).

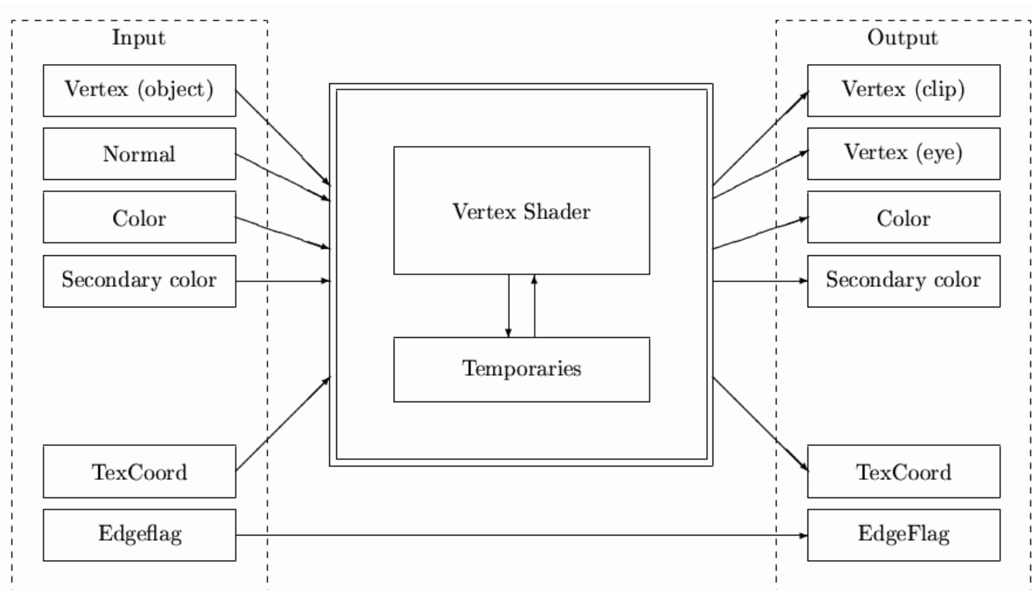
10. **Lighting** - v tomto bloku probíhá výpočet osvětlení, jejich souřadnic a normál. Přepínač **SW1** dovoluje tento krok vynechat.

11. **(0, 1)** - ořezání každé barevné složky v barvovém modelu RGBA do rozsahu 0..1. Ořezává se i hodnota průhlednosti. Tomuto postupu říkáme *clamping*.

12. **Texgen** - automatické generování souřadnic v textuře na základě souřadnic jednotlivých vrcholů a jejich normál (pokud je přepínač SW3 aktivní). Přepínač **SW2** určuje, zda se pro výpočet souřadnic použijí transformované nebo netransformované normálové vektory.

13. **Texture Matrix** - blok, který provádí transformace souřadnic v textuře na základě zadané transformační texturovací matice.

3.4.2 Vertexový procesor v programovatelném režimu



Obr. 10. Transformace prováděné vertex shaderem v programovatelném režimu [30]

V programovatelném režimu lze zpracování veškerých operací modifikovat pomocí programu ve *vertex shaderu*. *Vertex shader* současné generace neumí generovat nové nebo odstraňovat existující vrcholy (v souvislosti s Direct3D 10 a dalšími verzemi OpenGL se hovoří o nové jednotce, tzv. *Geometry shaderu*, který bude umět vrcholy přidávat i odebírat – využitelné při *Level of Detail* (LOD) renderování polygonálních modelů). Každá sekvence vstupních dat tedy generuje stejnou sadu výstupních dat.

Mezivýsledky výpočtů může *vertex shader* ukládat do pomocných proměnných (*temporaries*), které pracují na stejném principu, jako registry u univerzálních mikroprocesorů. Tyto proměné pak může využívat fragment shader nebo i samotná aplikace. Do *vertex shaderu* je možné parametry také posílat z aplikace.

3.5 Fragmentový procesor

Fragment shader neboli *rasterizační program* (někdy hovoříme o *pixel shaderu*) je implementován v samostatné programovatelné jednotce, která je, podobně jako výše popsaná jednotka pro vertex shader, taktéž umístěna na grafickém procesoru. Pomocí

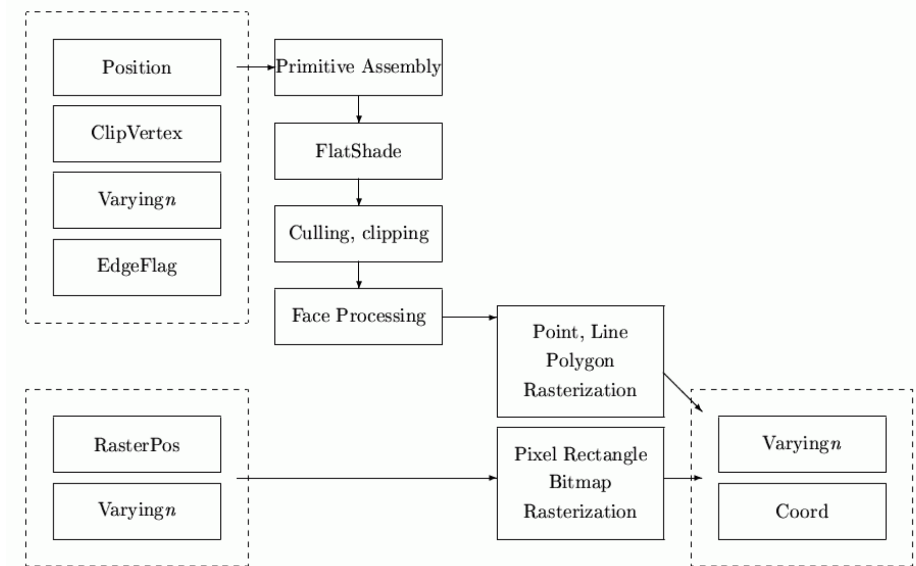
fragment shaderu je možné měnit původní standardní operace nad vykreslovanými fragmenty. Mezi tyto operace patří multitexturing, modulace textur, různé pixelové konvoluční filtry a další operace prováděné per pixel (pro každý jednotlivý pixel), nikoli tedy pouze per vertex (pouze pro každý vrchol plošky) jako tomu je v pevném režimu.

Na současných grafických akcelerátorech je z celého rasterizačního řetězce v současné době programovatelná pouze část jeho operací, která je na (Obr. 11) reprezentovaná blokem **Point, Line Polygon Rasterization**.

3.5.1 Fragmentový procesor v pevném režimu

Průběh rasterizace v pevném režimu:

1. Každý fragment se zpracovává samostatně, nezávisle na ostatních fragmentech.
2. Vstupem pro rasterizaci jsou souřadnice vykreslovaného pixelu z textury, primární barva, sekundární barva a parametry pro aplikaci efektu mlhy.
3. Barvu vykreslovaného fragmentu získáme postupnou kombinací texelů z příslušných textur. V této fázi se aplikují grafické efekty prováděné nad texturami. Tento bod zpracování se provádí pouze v případě, že je texturování povoleno. V případě, že je texturování zakázáno, tak se pro výpočet barvy fragmentu využije pouze primární a sekundární barva.
4. Barva získaná kombinací barev texelů se moduluje primární barvou a k výsledku je přičtena barva sekundární.
5. V dalším kroku je aplikován efekt mlhy (*fog*), což je ve skutečnosti opět operace míchání dvou barev. Barva mlhy je vypočtena pomocí směšovací funkce, jejíž výsledek závisí na hloubce fragmentu (tj. vzdálenosti od pozorovatele).
6. Následuje test na hloubku fragmentu oproti hloubce uložené v paměti hloubky (*depth buffer*). Pokud test proběhne úspěšně, je fragment poslán do dalšího zpracování; v opačném případě je z vykreslovacího řetězce odstraněn.
7. Na fragment může být následně aplikován *alfa-blending* s fragmentem uloženým ve *framebufferu*. Po provedení *alfa-blendingu* je výsledný fragment zapsán do *framebufferu*.



Obr. 11. Operace prováděné fragment shaderem v pevném režimu [30]

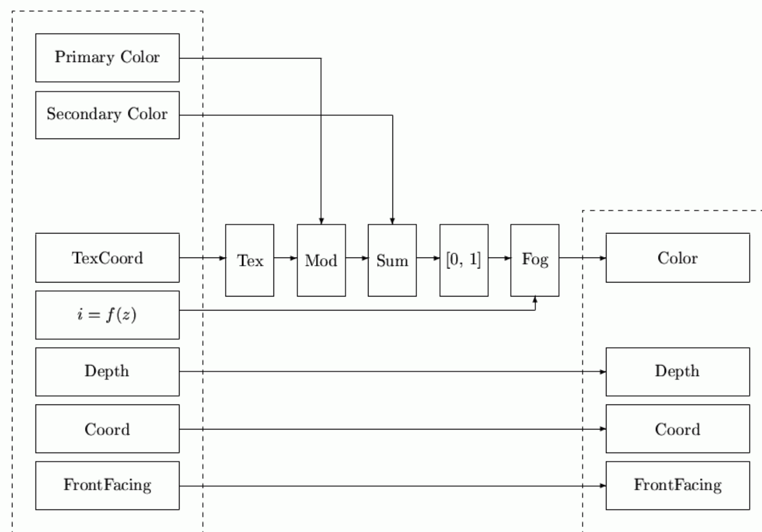
Význam jednotlivých bloků (Obr. 11) využívaných *fragment shaderem*:

1. **Position** - předem transformované pozice vrcholů bodů, úseček a polygonů. Tyto pozice reprezentují vrchol polygonu v obrazovkových souřadnicích.
2. **ClipVertex** - souřadnice vrcholů ořezávací roviny nebo ořezávacích rovin. Ořezání se provádí v bloku **Culling, Clipping**.
3. **Varyingn** - další parametry, které mohou být přiřazeny ke každému vrcholu.
4. **EdgeFlag** - příznak, zda je vykreslovaná hrana viditelná (tělesová), či nikoliv.
5. **RasterPos** - počáteční pozice bitmapy pro následné vykreslování.
6. **Primitive Assembly** - blok provádějící základní operace s grafickými primitivy
7. **Flatshade** - přiřazení konstantní barvy celé plošce – konstantní stínování
8. **Culling, clipping** - odstranění neviditelných plošek a ořezání plošek, které se nacházejí mimo těleso záběru.
9. **Face processing** - podle nastavení způsobu zobrazení polygonu se v tomto bloku může polygon převést na body nebo úsečky (*wireframe*).

10. **Point, Line, Polygon Rasterization** - hlavní část rasterizačního řetězce, převod geometrických dat na jednotlivé fragmenty.

11. **Pixel Rectangle, Bitmap Rasterization** - převod (*rasterizace*) bitmap na jednotlivé fragmenty s případnými operacemi zvětšování a zmenšování.

12. **Coord** - vypočtené souřadnice fragmentů, které jsou použity v dalším kroku při přístupu do framebufferu.



Obr. 12. Výkonná část fragment shaderu [30]

<u>Význam jednotlivých výkonných bloků vyobrazených na předchozím obrázku</u>:

1. **Tex** - výběr *texelu* z jedné či více textur. Pozice *texelu* v textuře se zjistí pomocí souřadnic zadaných do textur, které do rasterizačního řetězce vstupují z bloku **TexCoord**. Tyto souřadnice byly předem transformovány buď podle texturovací transformační matice, nebo pomocí programu spuštěného ve vertexovém procesoru.

2. **Mod** - modulace textur primární barvou fragmentu (použije se pouze v případě, že se kombinuje osvětlení spolu s texturováním).

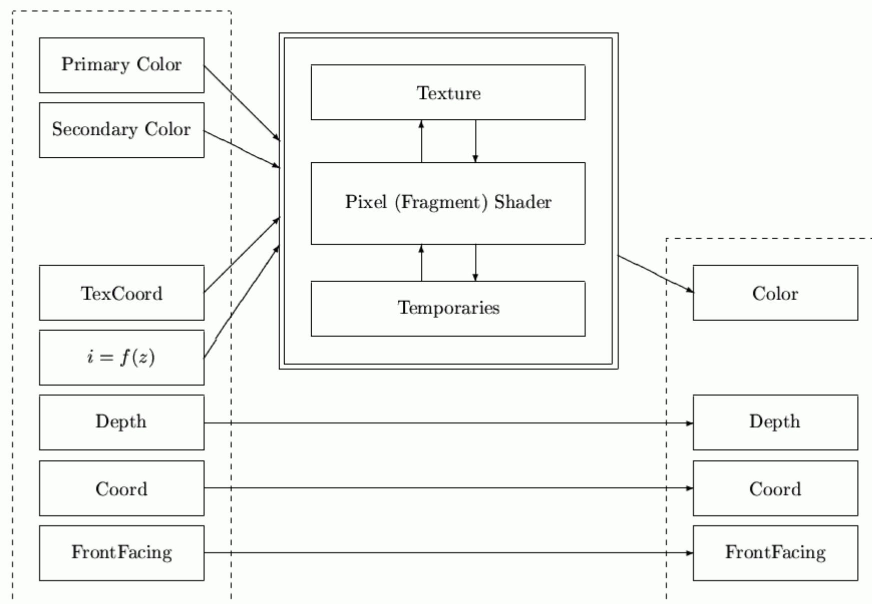
3. **Sum** - přičtení sekundární barvy k již vypočtené barvě fragmentu.

4. **[0,1]** - ořezání (clamp) hodnot barevných složek do rozsahu (0,1).

5. **Fog** - aplikace efektu mlhy, tj. smíchání vypočtené barvy s barvou mlhy, která je váhovaná dle výsledku funkce i=f(z), kde z je hloubka fragmentu.

3.5.2 Fragmentový procesor v programovatelném režimu

V programovatelném režimu probíhají veškeré operace s grafickými fragmenty podle schématu vyobrazeném na (Obr. 13). *Fragment shader* modifikuje zpracovávané fragmenty podle uživatelského programu. Výsledkem je barva fragmentu. Podobně jako *vertex shader* může i *fragment shader* pomocí speciálních proměnných komunikovat s aplikací.

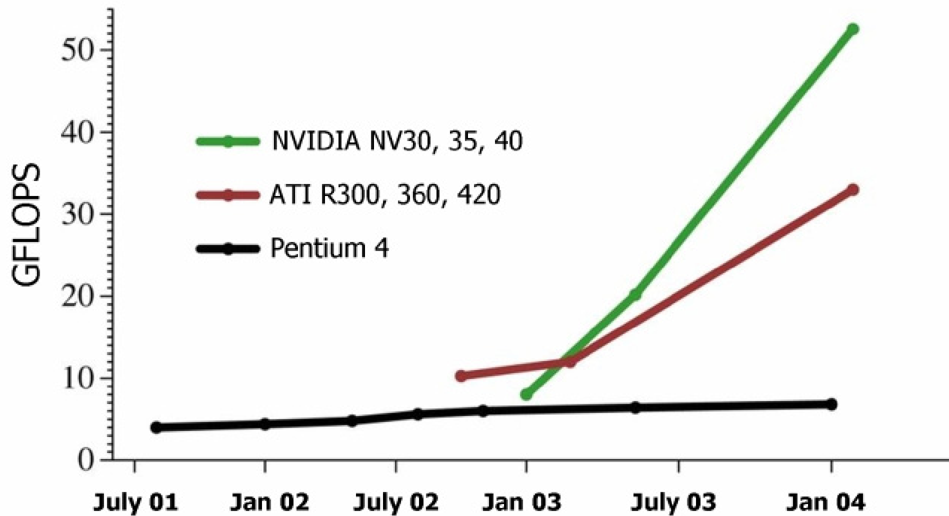


Obr. 13. Fragment shader v programovatelném režimu [30]

3.6 Využití grafického čipu pro obecné výpočty

Stále větší zájem o problematiku *vertex* a *fragment shaderů* není pouze ze strany programátorů grafických aplikací. Ukazuje se, že myšlenka, využít GPU pro obecné matematické výpočty (*GPGPU – General Purpose Computing on Graphics Processors*) otevírá zajímavé aplikační možnosti. Architektura dnešních moderních programovatelných akcelerátorů je totiž založena na vysoce paralelní struktuře, která svého výkonu dosahuje díky vektrorizovaným operacím zabudovaných přímo do jádra. Další výhodou je velký

stupeň integrace součástek a tím možnost rychlejší komunikace než je tomu u běžných PC. Z (Obr. 14) lze jasně vidět, že výkon (a hlavně trend křivky) grafických akcelerátorů je na zcela jiné úrovni než u konvenčního CPU.



Obr. 14. Výkon GPU a CPU [11]

Grafický procesor je ve své podstatě stream procesor a jeho architektura je dosti vzdálená od tradiční Von Neumanovy známé z běžných CPU. Síla velmi výkonné paralelizace (až 48 paralelních jáder) architektury SIMD (*Single Instruction Multiple Data*) spočívá v provedení jednoduché operace nad více daty najednou.

Základní postup obecného výpočtu na GPU je založen na následujících principech:

1. **Pole odpovídá texturám:** Klasické pole u grafického procesoru nelze využít, místo toho se logicky používá textura, do které lze díky fragment shaderu číst a zapisovat.
2. **Renderování do textury:** Výpočet (rastrový obraz) je uložen nikoli do framebufferu, ale do textury, která obsahuje výsledek.
3. **Smyčka nahrazená výpočetním jádrem:** Zpracování dat je silně paralelní, využití klasické výpočetní smyčky (*for pole [NxM] do…*) není proto možné.

Využijeme skutečnosti, že *fragment shader* představuje skupinu operací, která je aplikována na každý fragment rasterizovaného obrazu. Volbou vhodné velikosti rasterizovaného obrazu vzhledem k rozměru vstupních textur (polí), bude zpracován každý prvek vstupního pole, stejně jako kdybychom realizovali smyčku. Výpočet bude navíc díky implicitnímu paralelismu zpracován velmi rychle.

Uvedené principy jsou pouze základní myšlenky a náznaky převodu obecných výpočtů na GPU. Jako jazyk pro *shadery* se používá Cg, GLSL, HLSL nebo specializované metajazyky Brook [5] (Standfordská univerzita) a Sh [28] (University of Waterloo). Celá problematika provádění takových výpočtů není v dnešní době ještě zcela vyřešena, přesto je implementace některých klasických algoritmů pro výpočet na GPU několikanásobně rychlejší než na CPU. Použití GPU pro výpočty v sobě skrývá obrovský potenciál do budoucna.

V současnosti jsou na GPU úspěšně řešeny např. následující úlohy: řešení hustých soustav lineárních rovnic metodou LU dekompozice, řešení řídkých soustav lineárních rovnic metodami sdružených gradientů a Gauss-Seidelovou metodou, metoda pro třídění implementovaná v GPU (6-25 násobného zrychlení oproti CPU), projekt zabývající se implementací **paralelních genetických algoritmů na GPU**.

Celá problematika GPGPU je mimořádně zajímavá. Podrobnější informace lze nalézt v [8], [11] a [13].

4 EFEKTIVNÍ REPREZENTACE SCÉNY

Při zobrazování rozsáhlých scén je technicky nemožné zajistit dostatečnou rychlost renderování v reálném čase bez efektivní správy zobrazované scény. V zobrazovacím řetězci existuje standardně přímo v hardwaru podpora pro odstraňování od kamery odvrácených polygonů (*backface culling*) a ořezání scény mimo pohledový komolý jehlan perspektivy. Tyto prostředky jsou však pro zobrazování scén s větším počtem polygonů nedostatečné. Mnoho výpočtů je prováděno zbytečně, protože jsou v následujících krocích vyhodnoceny jako nepotřebné. Nelze pouze posílat polygony do renderovací pipeline a nezajímat se o jejich poloze v objektovém prostoru scény. Je potřeba uvědomit si, že podstatná část této geometrie bude mimo výhled kamery nebo bude zastíněná jinými tělesy. Také při detekci kolizí a testech interakce se scénou je vhodné provádět tyto testy jen pro takové skupiny polygonů, které má smysl uvažovat (např. avatar nemůže při svém pohybu kolidovat s geometrií, kterou má za zády). Další podrobnosti lze nalézt v [42].

4.1 Pomocné datové struktury

Pro zobrazování rozsáhlých scén existují pomocné datové struktury efektivně popisující scénu. Základní princip je v určení objektů nacházejících se v zadané prostorové oblasti. Vhodným uspořádáním těchto informací v prostoru velmi výrazně snížíme časové nároky při zobrazování rozsáhlé scény. Tyto struktury mají hierarchický charakter a popisují 3D prostor scény. Někdy také hovoříme o prostorové hierarchii.

Tyto struktury používáme většinou statické, tzn. že při zobrazování scény se data v těchto strukturách již nemění. Dynamické vytváření a modifikace struktury s ohledem na frekvenci využití prostorových dat je ale také možné.

Použití hierarchických datových struktur můžeme rozdělit do dvou kroků:

1. Prvotní inicializace a vytvoření datové struktury ze vstupních dat popisujících prostor scény
2. Operace s datovou strukturou při zobrazování scény v reálném čase, detekce kolizí, atd.

4.2 Hierarchie obálek

Prostorové hierarchie dělíme dvou skupin. První nazýváme *Hierarchie obálek* (*BVH – bounding volume hierarchies*) a vzniká postupným shlukováním objektů a obálek, které tyto objekty obklopují. Nejčastějšími typy obálek jsou koule (*sphere*), osově orientovaný kvádr (*AABB – axis aligned bounding box*), orientovaný kvádr (*OBB – oriented bounding box*) a orientovaný rovnoběžnostěn (*k-DOP – k-discrete orientation polytop*). Nelze jednoznačně říci, která obálka je lepší. Jejich volba záleží na řešeném problému. S jednoduchými obálkami se lépe pracuje, ale hůře se přizpůsobují různým tělesům – často obklopují i velkou část prázdného prostoru. Tyto obálky se poté seskupují do hierarchické stromové struktury. V jednotlivých listech stromu jsou pak obálky ohraničující objekty ve scéně.

Při hledání polohy bodu vůči objektům ve scéně procházíme strom obálek směrem od kořene a testujeme jeho polohu vůči obálce uložené v uzlu. Tento postup poté rekurzivně opakujeme. Pokud je strom vytvořen efektivně, tak se problém hledání polohy bodu transformuje z lineární složitosti na složitost logaritmickou. Tohoto stavu lze dosáhnout pouze tehdy, je-li strom vyvážený – má co nejmenší hloubku. Ne vždy je to ale možné.

4.3 Dělení prostoru

Druhou možností pro vytvoření hierarchické datové struktury popisující prostor scény je v systematickém dělení prostoru scény (*scene partitioning*). Nejznámější postupy

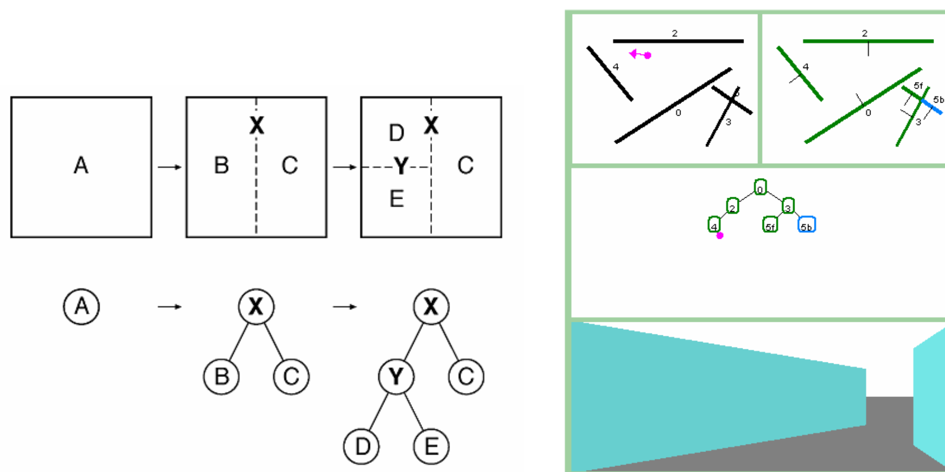
pro vytváření této datové struktury jsou: pravidelná mřížka (*grid*), oktalový strom (*octal tree*), strom BSP (*binary space partitioning tree*) a kD-strom (*k-dimensional tree*).

4.3.1 Pravidelná mřížka

Pravidelná mřížka nevytváří hierarchickou strukturu, pouze pravidelně dělí prostor rovinami kolmými na souřadnicové osy. Vzdálenost těchto rovin je ekvidistantní a vzniklé prostorové buňky (*cells*) mají tedy stejnou objemovou velikost. Mřížka nebere ohled na rozmístění geometrie objektů ve scéně.

4.3.2 BSP strom

BSP strom [6] je binární strom s obecně umístěnými řezy. Vytváří hierarchickou datovou strukturu a dělení prostoru a následné zobrazení je obdobné jako u oktalového stromu. Principielní změna je v prokládání prostoru rovinou rozdělující scénu na dva poloprostory. Tyto řezné roviny mohou mít libovolné umístění. V každé úrovni vznikají tedy dva potomci. *BSP strom* je úspornější varianta než oktalový strom, jeho implementace je složitější.



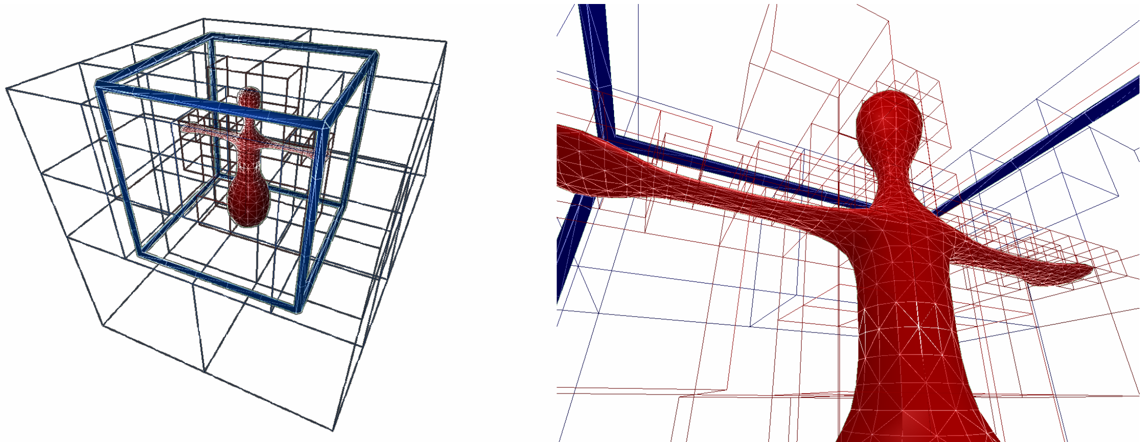
Obr. 15. BSP strom

4.3.3 k-D strom

Tato datová struktura je speciální případ stromu BSP určený k popisu libovolného k-rozměrného tělesa. Řezná rovina je vždy kolmá na jednu souřadnicovou osu a jejich orientace se cyklicky střídá.

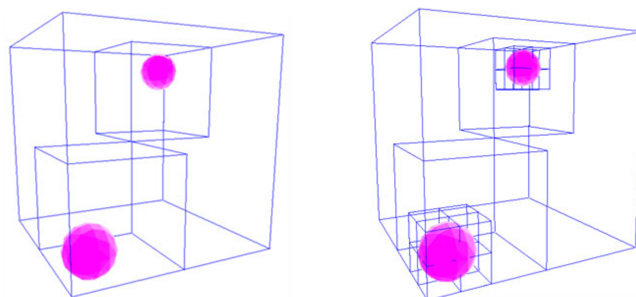
4.3.4 Oktalový strom

Technika *oktalového stromu* se zakládá na hierarchickém rekurzivním dělení prostoru scény pravidelnými řezy kolmými na roviny jednotlivých souřadnicových os. Při vytváření stromu je prostor scény s objekty rekurzivně dělen v každé úrovni na 8 vnitřních uzlů. Kořenem stromu je krychle obsahující celou geometrii scény. Každým krokem tedy vzniká 8 menších krychlí o poloměru $r_{new} = \frac{r_{old}}{8}$.



Obr. 16. Oktalový strom v enginu

V tomto stromu existují tři základní typy uzlů. Jesliže daný uzel (*oktant*) neobsahuje žádnou geometrii, je mu přiřazen příznak koncového listu (*void*). Obsahuje-li uzel geometrii a podmínka ukončení rekurze (viz dále) není splněna, je uzel označen jako smíšený (*mixed*) a následně je rekurzivně dále dělen. Obsahuje-li uzel geometrii a podmínka pokračování rekurze není splněna, je označen jako koncový list (*full*).



Obr. 17. Dělení prostoru oktalovým stromem

Podmínka určující zda-li je oktant označen jako prázdný nebo koncový, má nejčastěji 3 požadavky.

Uzel je označen jako koncový list:

1. Pokud je počet polygonů v tomto oktantu menší než stanovená mez maximálního počtu polygonů ve scéně
2. Přesáhne-li hloubka stromu maximální možnou úroveň
3. Přesáhne-li celkový počet koncových listů stromu maximální možnou hodnotu

Správným nastavením těchto tří statických parametrů lze řídit konstrukci *oktalového stromu*. Nastavíme-li maximální počet úrovní na nulu, podmínka rekurzivního dělení není splněna již v první úrovni a do kořene je přiřazená celá geometrie scény. Při příliš velkém počtu koncových listů se zvyšuje paměťová náročnost datové struktury a efektivita reprezentace postupně klesá.

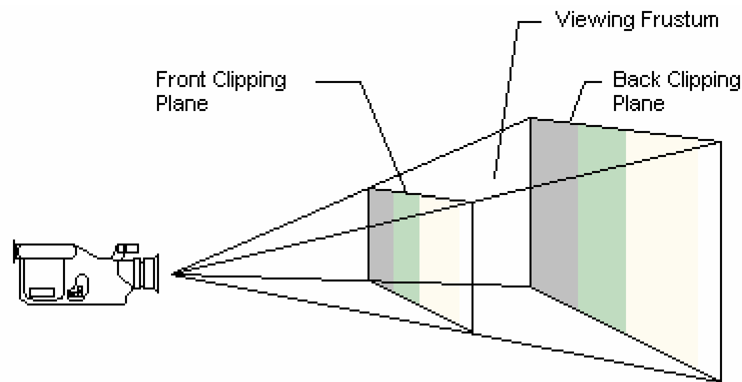
Při vytváření stromu často nastává situace, kdy jeden nebo více vrcholů polygonu leží v různých oktantech. Řešení tohoto problému spočívá buď v rozdělení daného polygonu na dva menší řezem vedeným hranicí mezi oktanty nebo v přiřazení polygonu do obou listů nebo v přiřazení polygonu pouze do jednoho listu. Rozřezávání polygonů nemusí být vždy ideální řešení, uvážíme-li že polygon může protínat i několik oktantů. Duplikování polygonů do všech listů, ke kterým tento polygon náleží, může způsobovat problémy při používání průhledných materiálů. V tomto případě bude průhlednost aplikována vícenásobně a výsledná plocha bude zobrazena nekorektně. Přiřazení polygonu pouze do jednoho listu může v určitých situacích zase způsobit nevykreslení polygonu – prostor (oktant), ve kterém část polygonu leží, byl vyhodnocen jako viditelný, ale fyzicky byl polygon přiřazen do jiného listu, vyhodnoceného jako neviditelný.

Ořezávání pohledovým objemem

Efektivita zobrazování scény pomocí *oktalového stromu* spočívá především v kombinaci s technikou ořezávání pohledovým objemem (*frustum culling*). Pohledový objem je část prostoru scény, která je v záběru kamery.

Tvar pohledového objemu záleží na projekční transformaci. Nejčastějším typem projekce je perspektiva. Pro perspektivní projekci je charakteristické, že objekty blíže ke kameře se jeví větší než vzdálenější. Takový pohledový objem lze vizualizovat jako jehlan omezený přední a zadní ořezávací rovinou (*front and back (near and far) clipping plane*).

Někdy proto hovoříme o komolém jehlanu perspektivy. Prostor mezi přední a zadní ořezávací rovinou tvoří pohledový objem.



Obr. 18. Komolý jehlan perspektivy

Pohledový objem lze získat jednoduchým algoritmem z transformační a modelovací matice. Myšlenka tohoto postupu spočívá ve vymezení pohledového objemu pomoci 6-ti rovin. Z rovnice roviny získáme informaci, zda-li testovaný bod leží před, za nebo v dané rovině. U každé roviny hledáme správný poloprostor ohraničující jednu stranu pohledu. Bod je vnitřní pokud leží vzhledem ke všem rovinám ve správném poloprostoru. Rovnice rovin určíme ze znalosti modelovací a projekční matice.

Tyto matice získáme z OpenGL příkazy:

```
1 glGetFloatv( GL_PROJECTION_MATRIX, proj );
2 glGetFloatv( GL_MODELVIEW_MATRIX, modl );
```

Celý princip zobrazování scény pomocí *oktalového stromu* je tedy v procházení jednotlivých větví stromu od kořene a testování zda-li koncový list obsahující geometrii scény leží ve výhledu kamery. Každý list stromu nese také informaci o velikosti strany krychle obklopující prostor scény přidělený tomuto listu. Test, zda-li je tato část prostoru ve výhledu kamery, je velmi rychlý. Úspora zobrazovacího času může být velká. Čím blíže ke kameře posuneme zadní ořezávací rovinu tím efektivnější zobrazování scény dosáhneme. Při větších scénách je ale vzdálenější geometrie oříznuta blízkou zadní ořezávací rovinou a scéna nevypadá zcela korektně. Jisté řešení může být použití mlhy jež od určité vzdálenosti od kamery scénu zahalí (lze nastavit enginu). Zadní ořezávací rovina by proto měla být volena s ohledem na charakter scény.

Použití pohledového ořezávání neumožňuje odstranit všechny polygony, které není třeba zobrazovat. V pohledovém objemu zobrazujeme veškerou geometrii a nebereme ohled na fakt, že ve scéně může existovat např. zeď zakrývající zbytek scény. Tato zeď nám zabraňuje vidět hlouběji položené objekty a zastiňuje nám zbytek scény. Tato část scény je renderována zbytečně. Řešení se nazývá *occlusion culling* a spočívá v umístění speciálního ořezávače (*occluder*), jež vytváří ořezávací masku pro zbytek scény.

5 DYNAMICKÉ STÍNY

Stíny jsou jedním z nejdůležitějších aspektů při prostorovém vnímání člověka. Stíny nám napovídají o rozmístění objektů, pomáhají nám uvědomit si velikost těles ve scéně a dávají nám představu o poloze světelných zdrojů. V počítačové grafice se proto techniky a algoritmy vytvářející stíny ve scénách používají především jako prostředek ke zvýšení reálnosti a věrohodnosti zobrazované scény. Při vytváření fotorealistických statických obrazů nebo při generování jednotlivých okének animace se používají tzv. globální zobrazovací metody (*radiozita* nebo *ray tracing*). Tyto metody vypočítávají scénu samozřejmě i se stíny. Vzhledem k nesmírné výpočetní náročnosti se tyto metody v realtime renderované nebo interaktivní počítačové grafice nepoužívají. Pro zobrazování dynamických scén se stíny pomocí grafických rozhraní OpenGL nebo Direct3D se proto tento problém transformuje obecně na problém geometrický. Podrobnější informace jsou uvedeny v [1] a [42].

5.1 Základní pojmy

5.1.1 Vlastní a vržený stín

Stíny rozlišujeme na *vlastní* (*self shadows*) a *vržené* (cast shadows). Vlastním stínem označujeme stín, který vrhá objekt sám na sebe. Ve vlastním stínu tedy leží všechny ke světlu odvrácené plochy tělesa a především všechny polygony zastíněné samotným tělesem. Vržené stíny jsou stíny, které objekt vrhá na ostatní tělesa ve scéně

5.1.2 Měkký a ostrý stín

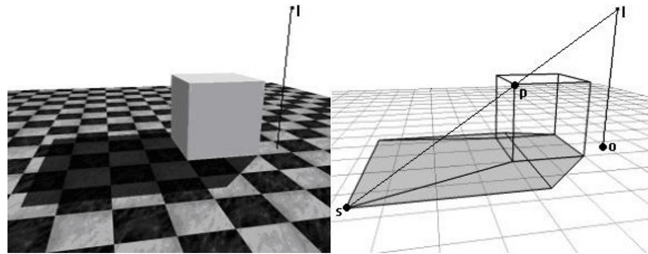
Bodové zdroje světla vytvářejí ve scéně ostré stíny (*hard shadows*). Pro každý bod prostoru scény můžeme jednoznačně určit, zda-li je tento bod přímo osvětlen bodovým světlem, nebo na spojnici se světlem leží další těleso. Měkké stíny (*soft shadows*) vytvářejí plošné zdroje světla. Tyto stíny mají rozostřenou hranici přechodu mezi hlavním stínem (*umbra*) a polostínem (*penumbra*). Polostín se ze zvětšující vzdáleností zdroje světla zvětšuje a naopak hlavní stín se zmenšuje. Měkké stíny vytvářejí oproti ostrým stínům kvalitnější vizuální zážitek, který je bohužel zaplacen velkou náročností výpočtu. V současné době se měkké stíny v realtime renderované počítačové grafice začínají teprve

prosazovat. Je to dáno především komplikovaností implementace takového algoritmu a také možnostmi dnešních grafických akcelerátorů. Na návrhu algoritmů pro výpočet měkkých stínů pracují v současné době především špičková pracoviště na univerzitách nebo vyspělé softwarové společnosti.

5.2 Metody pro zobrazování dynamických stínů

5.2.1 Projekční metoda

Tato jednoduchá metoda publikovaná v roce 1988 Jamesem Blinnem vypočítává stíny pomocí speciální transformační matice. Princip spočívá ve vytvoření zobrazení, které pro každou plochu, na kterou může dopadat stín, určí transformaci zobrazující do roviny této plochy libovolný 3D objekt jako 2D polygon. Tuto transformaci lze v homogenních souřadnicích popsat pouze jedinou maticí 4x4.



Obr. 19. Projekce 3D tělesa do 2D roviny [1]

Tato transformační matice má pro libovolnou plochu následující tvar:

$$M = \begin{bmatrix} nl + d - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & nl + d - l_y n_y & -l_y n_z & -l_x d \\ -l_z n_x & -l_z n_y & nl + d - l_z n_z & -l_y d \\ -n_x & -n_y & -n_z & nl \end{bmatrix} \qquad (2)$$

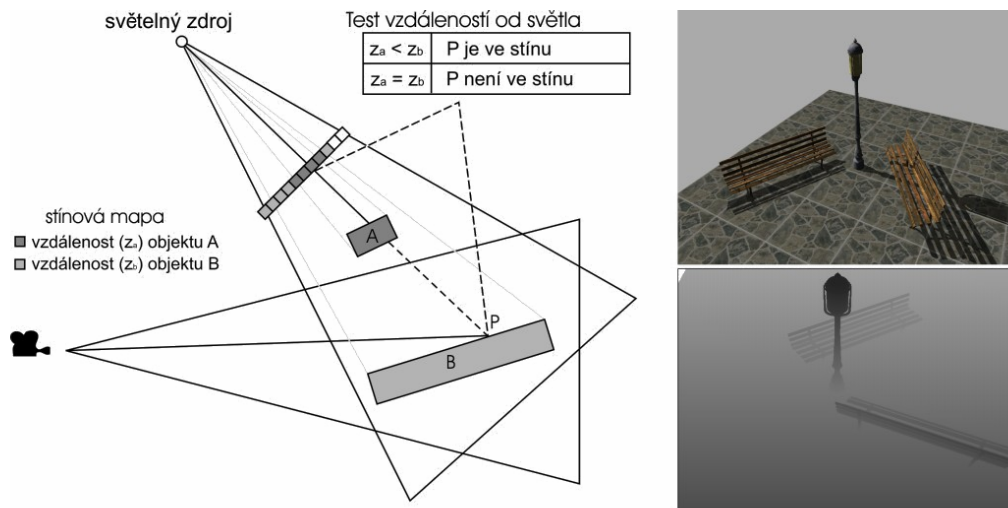
kde

l poloha světelného zdroje

n normálový vektor roviny $nx + d = 0$

5.2.2 Stínová paměť hloubky

Vytváření dynamických stínů pomocí stínových map [2] poprvé publikoval v roce 1978 Lance Williams. Tato metoda pracuje (na rozdíl od stínových objemů) v obrazovém prostoru scény a neklade požadavek na ploškovou (polygonální) reprezentaci scény. Další výhodou je velmi vysoká rychlost.



Obr. 20. Stínová paměť hloubky [1]

Algoritmus pracuje ve dvou krocích. Nejprve je scéna zobrazena z pohledu od světla a obsah této scény se uloží do stínové (hloubkové) mapy (*depth map*). Tato textura nenese informace o barvách, každý její pixel reprezentuje vzdálenost od světelného zdroje. Druhá fáze zobrazí scénu z pohledu kamery. Pixely hloubkové mapy tohoto pohledu se transformují do pohledu ze světla a výsledek se porovná se souřadnicí z v paměti hloubky. Nachází-li se bod získaný z pohledu kamery po transformaci dále, než při pohledu ze světla, je ve stínu a intenzita pixelu je přiměřeně modulována.

5.2.3 Stínové objemy

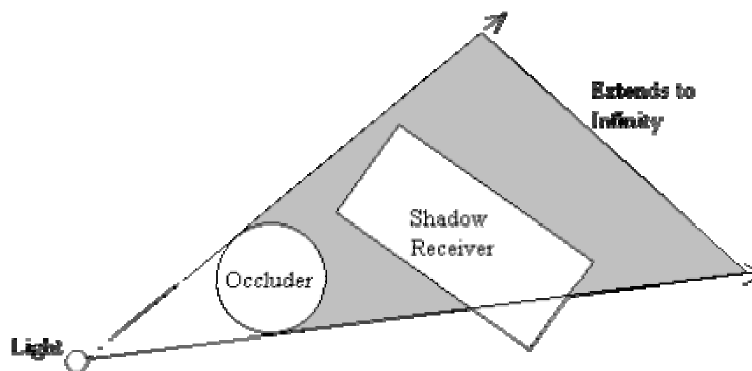
Metoda zobrazování stínů pomocí stínových objemů, přesněji algoritmus *depth-pass* je implementován v enginu. Tomuto algoritmu je věnována následující podkapitola 5.3.

5.3 Technika stínových objemů

Velmi rozšířenou metodou pro vytváření realtime dynamických ostrých stínů je technika založená na vytváření stínových objemů pomocí paměti šablon [20] (*stencil shadow volume*). Pro tento způsob výpočtu stínů disponují moderní akcelerátory několika specializovanými hardwarovými rozšířeními, které výpočet urychlují. Základní myšlenka algoritmu je ve vytvoření stínícího objemu pro každý polygon stínícího tělesa. Tento stínící objem reprezentuje zdrojem světla neosvětlený prostor scény.

Porovnáním vzájemné polohy stínícího objemu a povrchu objektů ve scéně může nastat některá z těchto situací:

1. Testovaná část objektu leží ve stínovém objemu, je tedy ve stínu
2. Testovaná část objektu neleží ve stínovém objemu, je tedy osvětlena světelným zdrojem



Obr. 21. Stínový objem

Základní variantu výpočtu stínů pomocí stínových objemů navrhl v roce 1977 Frank Crow. V roce 1991 implementoval Tim Heidmann tento algoritmus pomocí paměti šablon. Podle způsobu práce se *stencil bufferem* se tento algoritmus nazývá *depth-pass*. Vzhledem k hardwarovým možnostem tehdejších akcelerátorů a celkovému výpočetnímu výkonu počítačů na začátku devadesátých let minulého století se algoritmus praktického uplatnění dočkal až za několik let. V roce 2002 publikovala NVIDIA postup, který byl založen na opačném způsobu práce se *stencil bufferem*. Tento přístup dostal název *depth-fail* a v určitých situacích podává lepší a robustnější výsledky než původní varianta depth-pass. Tuto metodu proslavil především John Carmack ze společnosti id Software, která poprvé

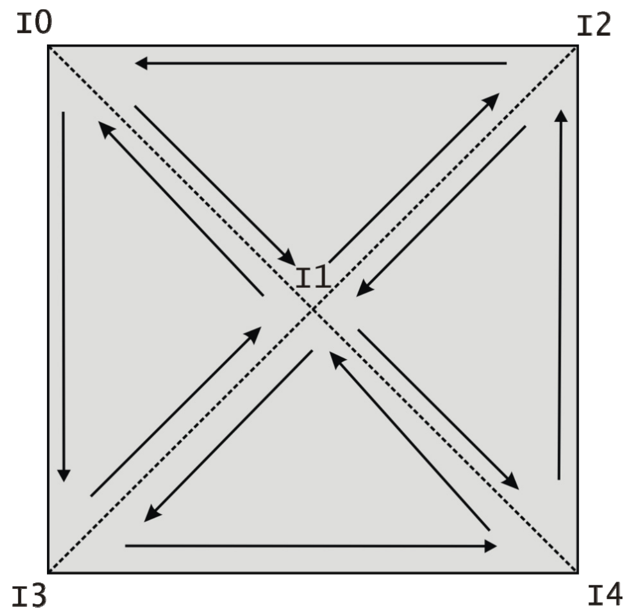
implementovala v počítačové hře (enginu) Doom3 dynamické stíny. Carmack postup odvodil nezávisle na ostatních pracovištích a nazval ho *Carmack's reverse*. Třetí způsob vytváření stínů pomocí *stencil bufferu* je tzv. *Exclusive-Or*. Reprezentuje nejjednodušší přístup, bohužel s řadou nedostatků. Výpočet stínů pomocí *depth-pass* i *depth-fail* probíhá v několika krocích. Exklusive-Or varianta má některé výpočetní kroky totožné, jiné vynechává.

Základní princip algoritmů lze shrnou do následujících kroků:

1. Renderování scény pouze v ambientním (okolním) osvětlení
2. Určení siluety stínícího objektu vzhledem k uvažovanému zdroji světla
3. Vytvoření stínínového objemu
4. Renderování stínového objemu a zápis informací do stencil bufferu
5. Renderování scény s difuzním a speculárním (odrazovým) osvětlením pouze v osvětlených částech scény

5.3.1 Určení siluety stínícího objektu

Základním problémem je určení siluety (*possible silhouette edges*) stínícího tělesa. Extrudováním této siluety do nekonečna získáme stínový objem. Siluetu tělesa určujeme především z důvodu snížení počtu polygonů ve stínícím tělese. Je zbytečné, a reálném výpočetním čase neoptimální, vrhat stín z vnitřních polygonů, které netvoří vnější plášť stínového objemu.



Obr. 22. Schéma hledání siluety objektu

Výpočet siluety tělesa vzhledem ke světelnému zdroji provádíme pouze pro ke světlu přivrácené polygony stínícího tělesa, tedy pokud skalární součin vektoru ze světelného zdroje a vektoru ležícího v rovině testovaného polygonu > 0. Pro každý takovýto polygon jsou do zásobníku hran tvořících siluetu tělesa postupně zapisovány jednotlivé hrany polygonu (hrana = 2 vrcholy I_A a I_B) a následně je testováno zda-li daná hrana již existuje v zásobníku (I_A-I_B vrcholy nebo I_B-I_A vrcholy). Pokud je takováto hrana v zásobníku nalezena, jsou testovaná i už existující hrana ze zásobníku vyřazeny a algoritmus pokračuje v testování dalšího polygonu stínového tělesa. Tímto způsobem se vytvoří seznam hran tvořící siluetu stínového tělesa.

5.3.2 Vytvoření stínového objemu

Stínový objem vytvoříme na základě siluety stínového tělesa. Princip vytvoření nekonečného stínového objemu (*infinitive shadow volume*) spočívá v protažení hran tvořící siluetu stínového tělesa do nekonečna. Při prodloužení stínového tělesa do nekonečna se využívá vlastností homogenních souřadnic. V homogenních souřadnicích popisujeme vrchol pomocí 4D vektoru $\vec{a} = (a_x, a_y, a_z, a_w)$. První tři složky vektoru určují pozici v prostoru a čtvrtá složka a_w určuje tzv. homogenní váhu. Standardně je přiřazena hodnota a_w= 1, při zobrazení v nekonečnu se přiřadí hodnota a_w= 0.

Pokud má nalezená obrysová hrana souřadnice vrcholů $\vec{a} = (a_x, a_y, a_z, a_w)$, $\vec{b} = (b_x, b_y, b_z, b_w)$ a souřadnice světelného zdroje jsou $\vec{l} = (l_x, l_y, l_z, l_w)$, pak boční stěna stínového objemu bude mít dva vrcholy totožné s vrcholy *a*, *b* tvořící nalezenou hranou a zbývající dva vrcholy vzniknou zobrazením vrcholů *a*, *b* do nekonečné vzdálenosti ve směru vektoru *al* resp. *bl*.

Vrcholy hraničního polygonu stínového tělesa:

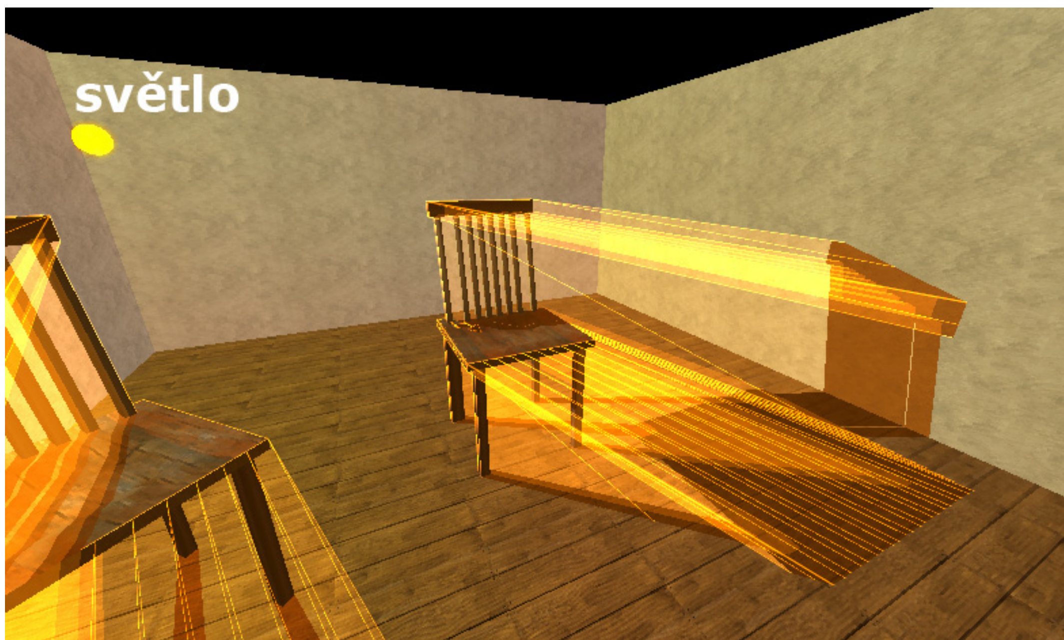
$$vertex_1 = (a_x - l_x, a_y - l_y, a_z - l_z, 0)$$

$$vertex_2 = (a_x, a_y, a_z, 1)$$

$$vetex_3 = (b_x - l_x, b_y - l_y, b_z - l_z, 0)$$

$$vertex_4 = (b_x, b_y, b_z, 1)$$

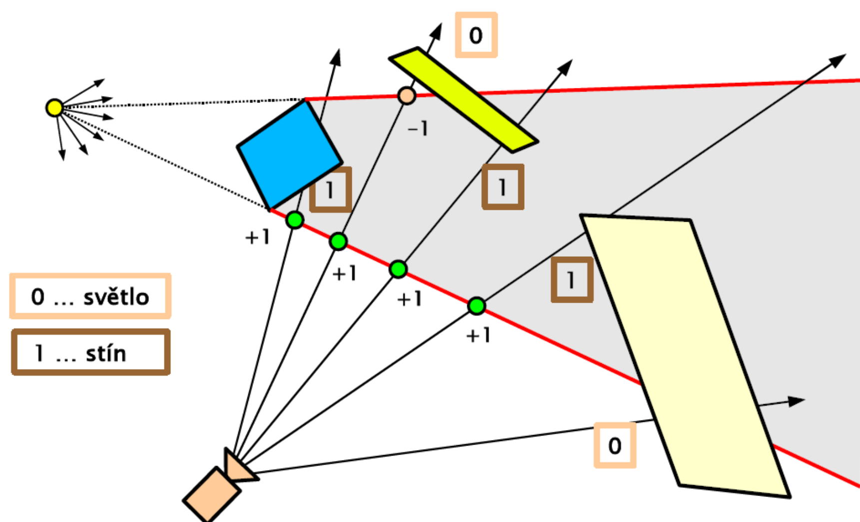
Tyto vrcholy se renderují buď jako čtvercové polygony (*quad*) nebo jako trojúhelníkové polygonální pásy (*triangle strip*). Poznamenejme, že výše uvedené určení stínového objemu je pro všesměrový světelný zdroj (*omni light*). V případě směrového zdroje světla (*directional light*) je výpočet stínového objemu jednodušší, protože rovnoběžné přímky se v homogenních systémech v nekonečnu protínají a polygon je tedy uvažován pouze trojúhelníkový (*triangle*).



Obr. 23. Vizualizace stínového objemu v enginu

5.3.3 Depth-pass algoritmus

Nejdůležitějším krokem při výpočtu stínů pomocí stínových těles je testování vzájemné polohy polygonu a stínového objemu. Myšlenka algoritmu spočívá ve vysílání testovacích paprsků ze středu promítání každým pixelem směrem k zobrazovanému povrch ve scéně. Během cesty paprsku se počítá kolikrát tento paprsek vstoupil a kolikrát vystoupil ze stínového tělesa. Pokud je rozdíl těchto hodnot nenulový, paprsek neopustil stínový objem a bod na povrchu tělesa, ke kterému dorazil, musí ležet ve stínu.



Obr. 24. Princip depth-pass algoritmu [23]

Celý systém zobrazování scény se děje v několika krocích. Nejprve je vyrenderována scéna a vyřešena viditelnost pomocí hloubkového testu (*depth test*). Poté každému pixelu přiřadíme čítač, který nastavíme na hodnotu odpovídající počtu stínových těles v nichž se kamera nachází (důležitý krok). Nyní provedeme hloubkový test polygonů stínových těles s geometrií ve scéně. Je-li test pro daný pixel úspěšný, tak pro ke kameře přivrácené plochy inkrementujeme čítač a pro odvrácené plochy ho *dekrementujeme*. Zůstane-li v čítači po zpracování všech ploch stínového tělesa nenulová hodnota je daný pixel ve stínu. Tento algoritmus popsal v roce 1991 Tim Heidman. Pole do kterého se ukládají čítače je hardwarový *stencil buffer* grafického akcelerátoru. Podle způsobu práce se *stencil bufferem* – čítač inkrementujeme jen tehdy, pokud byl hloubkový test úspěšný – dostala tato metoda pojmenování *depth-pass*.

Nejproblematičtější část uvedeného postupu nastává, pokud se kamera nachází uvnitř stínového objemu. Je třeba správně určit v kolika stínových tělesech se nachází (test kolize spojnice kamery a světelného zdroje se stínovými objemy) a hlavně dávat pozor na přední ořezávací rovinu pohledového objemu. Může nastat situace, kdy je sice kamera ve stínu, ale přední plocha tohoto objemu je ve stínu jen částečně. Konečné řešení tohoto problému ještě nebylo uspokojivě vyřešeno. Z výše uvedených důvodů, byl v letech 1999-2001 obrácen smysl algoritmu a navrhnuto inkrementovat hodnotu čítače pokud hloubkový test selže. Tento postup se již nedostává do konfliktu s přední ořezávací rovinou (problém přenáší na zadní). Postup se nazývá *depth-fail* (Bilodeau, Songy) [20] nebo také *Carmack's reverse* (Carmack).

Vzhledem k velké popularitě výpočtu stínů za pomoci paměti stencil buffer implementovali výrobci grafického hardwaru do akcelerátorů několik speciálních funkcí, které pomáhají zrychlovat práci se *stencil bufferem*. Jde zejména o rozšířeni *EXT_STENCIL_TWO_SIDE* umožňující provádět inkrementaci i dekrementaci čítače v *stencil bufferu* v jediném kroku nebo *EXT_STENCIL_WRAP* umožňující přetečení stencil bufferu při ukládání hodnot a tím zvýšení spolehlivosti metody. OpenGL 2.0 tyto rozšíření samozřejmě podporuje.

Depth pass algoritmus v OpenGL 2.0:

```
 1 glDisable(GL_CULL_FACE);
 2 glEnable(GL_STENCIL_TEST_TWO_SIDE_EXT);
 3 glActiveStencilFaceEXT(GL_BACK);
 4 glStencilFunc(GL_ALWAYS,0,~0);
 5 glStencilOp(GL_KEEP,GL_KEEP,GL_INCR_WRAP);
 6 glActiveStencilFaceEXT(GL_FRONT);
 7 glStencilFunc(GL_ALWAYS,0,~0);
 8 glStencilOp(GL_KEEP,GL_KEEP,GL_DECR_WRAP);
 9 RenderujStinovyObjem();
10 glDisable(GL_STENCIL_TEST_TWO_SIDE_EXT);
11 glEnable(GL_CULL_FACE);
```

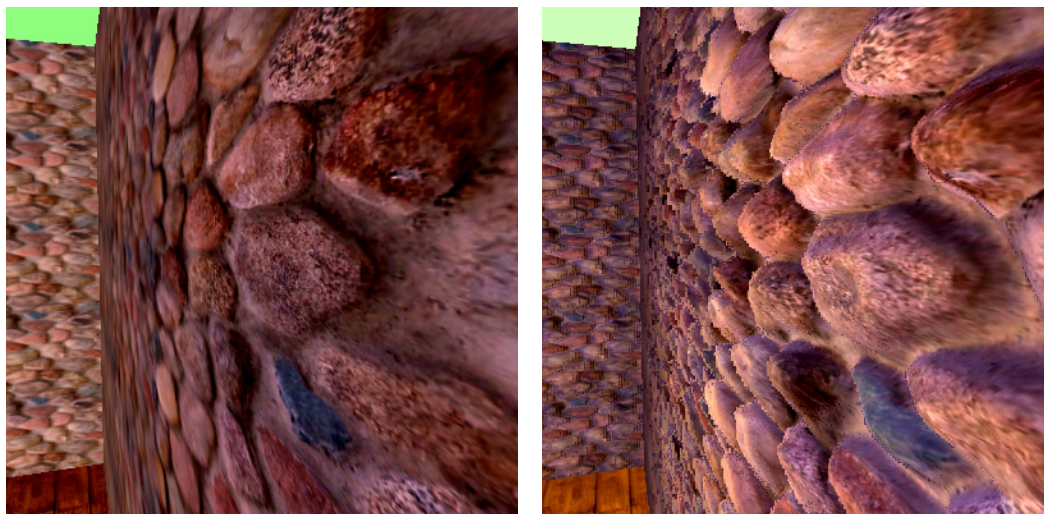
6 SHADERING

Rozvoj rychlých hardwarových akcelerátorů a zvyšování požadavků na kvalitu zobrazení vneslo do 3D grafiky nové programové řešení, pro které se vžil název *shadering* (*stínování*). Pojem *shadering* je staršího data a sahá až k tzv. standardu *RenderMan*, který zavedla firma PIXAR.

Princip práce shaderů spočívá ve změně nativní funkce programovatelného jádra GPU. Grafický procesor se chová dále normálně, tzn. že přijímá standardní data pro operace, ale díky pozměněné funkci pomocí kódu shaderu provádí další uživatelské operace. Tímto lze významně ovlivnit konečný výsledek renderování. Shadery se obecně dělí pro práci nad vrcholy (*vertex shader*) a pro práci nad pixely (*fragment shader, pixel shader*).

Zápis programového kódu shaderu se dříve zapisoval pseudojazykem podobným assembleru. To jeho použitelnost znatelně omezovalo a tak byly vytvořeny nové high level programovací jazyky, které ulehčují a zefektivňují práci s programovatelnými GPU. Nejpoužívanějšími stínovacími jazyky jsou *The OpenGL Shading Language* pro OpenGL, *High Level Shading Language* pro Direct3D a *Cg*. Základní informace lze nalézt v kapitole 2.3. Engine podporuje jazyk GLSL.

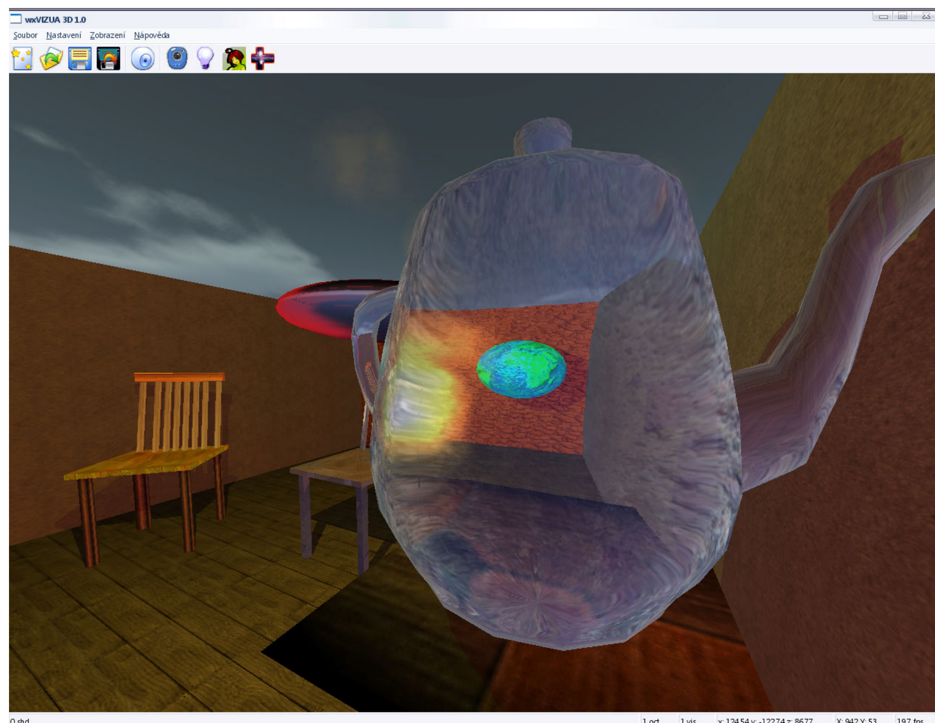
Velkou výhodou shaderingu je, že pracuje per pixel, tzn. že lze vypočítat barvu fragmentu přesně až na úroveň pixelu (základ per pixel osvětlení, bump mappingu, …). Klasický pevný režim umí zpracovávat fragmenty pouze per vertex, čili pouze s přesností vrcholů polygonální sítě.



Obr. 25. Engine: Vlevo bez shaderu, vpravo aplikován GLSL Relief shader

6.1 Shader Model

Shader Model (SM) specifikuje standard pro *vertex* a *fragment shadery*. Nejstarší verze 1.0 a 1.1 podporují v roce 2001 Direct3D 8.0 i OpenGL. SM popisuje počet instrukcí programovatelného GPU, počet registrů a další charakteristické prvky. V roce 2002 byl s příchodem akcelerátorů podporujících Direct3D 9.0 (OpenGL) uveden SM 2.0, který přinesl kvantitativní vylepšení (více instrukcí – 256, více konstantních registrů – 256). Poslední verze Shader Model 3.0, definuje již neomezený počet instrukcí a další významné prvky. Tento standard splňují např. grafické karty NVIDIA od GeForce 6xxx. Obecně tento model podporují akcelerátory s podporou Direct3D 9.0c a OpenGL 2.0.



Obr. 26. GLSL shader v enginu vytvářející efekt lomu světla na skle

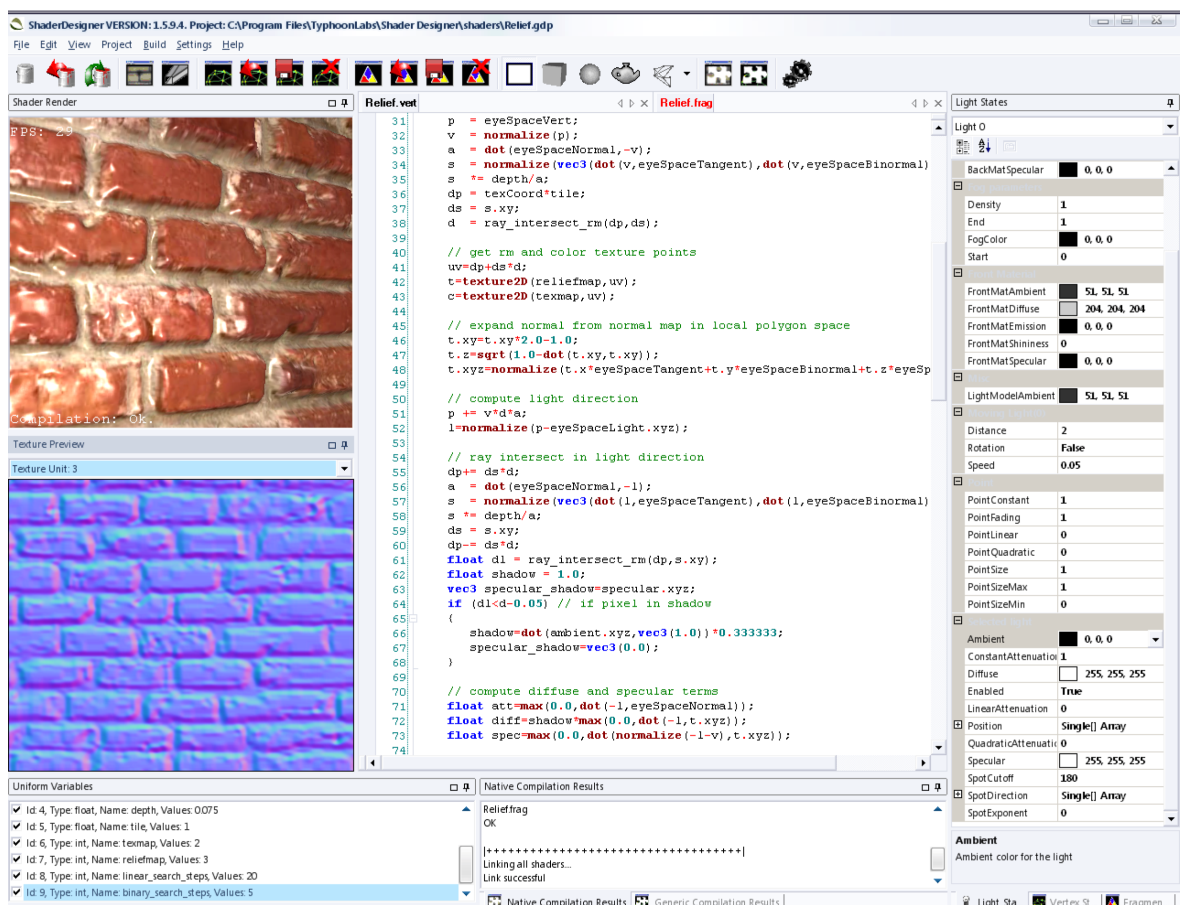
Základem SM 3.0 je podpora neomezeného počtu instrukcí a délky kódu shaderu, podpora dynamického větvení (*dynamic branching*), které umožňuje realizovat podmíněné skoky, smyčky a podprogramy. Standard podporuje výpočet oboustranného osvětlení polygonu v jednom výpočetním kroku, více renderovacích cílů (*multiple render targets*).

6.2 Shadering pomocí GLSL

Engine pro *shadering* využívá jazyk *The OpenGL Shading Language* (viz 7.1.3). Zdrojový kód shaderu je zkompilován a spuštěn až přímo v aplikaci (*principielní vlastnost GLSL*), tzn. že zdrojový kód lze kdykoli editovat a vylepšovat.

Pro vytváření *shaderů* se používají specializované vývojové editační nástroje. Nejznámější programy jsou ATI RenderMonkey, NVIDIA FX Composer a Shader Designer od společnosti TyphoonLabs 3D Production.

Pro vytváření a editaci *shaderů* pro wxVIZUA 3D byl použit nástroj Shader Designer (Obr. 27). Tento nástroj je velmi jednoduše ovladatelný a umožňuje pohodlné a efektivní programování GLSL *shaderů*.



Obr. 27. Shader Designer – pokročilý nástroj pro programování GLSL shaderů

Engine obsahuje zabudované *shadery* pro *Cel-Shading*, *Specular bump mapping*, *Relief shader*, *Real glass shader* a *Parallax shader*. Uvedené *shadery* lze případně modifikovat a upravit tak jejich vlastnosti (*adresář bin\wxVIZUA3D\data\shaders*).

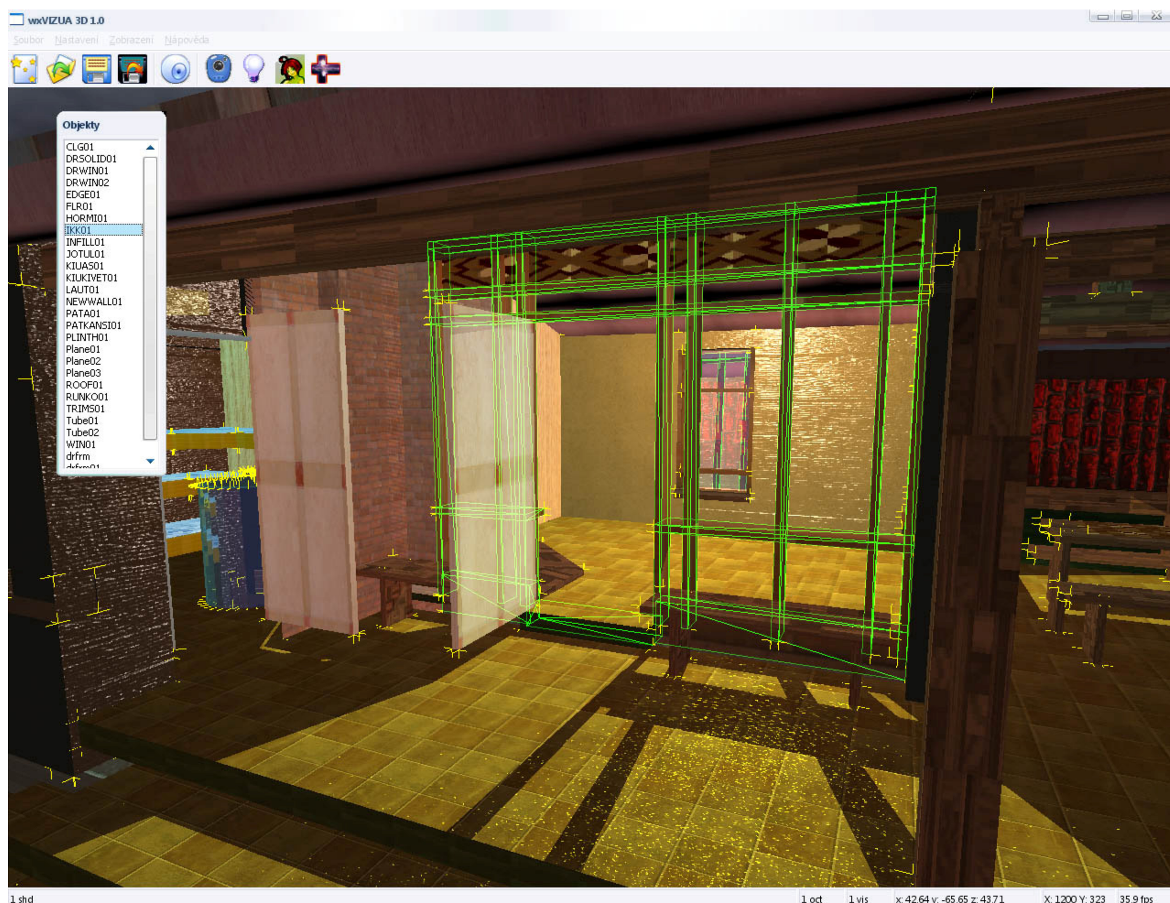
<u>Ukázka GLSL zápisu jednoduchého shaderu pro efekt Cel-Shading (cartoonová grafika):</u>

```
1 <vertex>
2
3 varying vec3 Normal;
4
5 void main(void)
6 {
7 Normal = normalize(gl_NormalMatrix * gl_Normal);
8 gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
9 }
10
11 <fragment>
12
13 uniform vec3 DiffuseColor;
14 uniform vec3 PhongColor;
15 uniform float Edge;
16 uniform float Phong;
17 varying vec3 Normal;
18
19 void main (void)
20 {
21 vec3 color = DiffuseColor;
22 float      f     = dot(vec3(0,0,1),Normal);
23 if (abs(f) < Edge)     color = vec3(0);
24 if (f > Phong) color = PhongColor;
25 gl_FragColor = vec4(color, 1);
26 }
```

II. PRAKTICKÁ ČÁST

7 REALIZACE 3D ENGINU V OPENGL

Praktická část této práce se zakládá na implementaci 3D grafického enginu v OpenGL 2.0. Základní algoritmické přístupy diskutované v teoretické části tvoří jádro aplikace. Software slouží pro interaktivní 3D realtime počítačovou vizualizaci. Aplikace byla pojmenována ***wxVIZUA 3D***. „*wx*“ zdůrazňuje, že software využívá multiplatformní API wxWidgets (viz 7.1). „*VIZUA*“ je zkrácenina slova vizualizace. Program představuje softwarovou architekturu založenou na hierarchické datové struktuře oktalovém stromu s pohledovým ořezáváním, dynamických stínech a *shaderingu*. Další vlastnosti: multitexturing, částicové efekty, rotace objektů a kamery pomocí quaternionů, detekce kolizí, skyboxing. Software byl vyvíjen 2 roky a rozsah napsaného C++ zdrojového kódu představuje 22 tříd na 15 000 řádcích.



Obr. 28. wxVIZUA 3D v editačním módu

Aplikace umožňuje importování 3DS modelu, na základě kterého je poté vytvořena prezentovaná scéna. Libovolnému objektu lze nastavit přibližně 30 různých parametrů,

které definují povrchové a výpočetní vlastnosti objektu. Celý projekt se ukládá do XML souboru. Aplikace má dva základní módy: editační (Obr. 28) a prezentační. První je určen pro nastavování parametrů scény a druhý je určen pro interaktivní prezentaci.

Možnosti nastavení:

1. Okolní (*ambient*), difůzní (*diffuse*), spekulární (*specular*) a emisivní (*emissive*) barva.
2. Odrazový exponent (*spot exponent*).
3. 4 texturové vrstvy (*multitexturing*) pro každý objekt, prolínání textur až na úroveň OpenGL parametrů.
4. 7 druhů zabudovaných shaderů (možnost i vlastní úpravy *shaderů*).
5. Průhlednost objektu (*blending*).
6. Průhledné pozadí stromů nebo bilbordů (*alpha channel masking*).
7. Generování texturových koordinát (*texture coordinate generation*).
8. Vrhání dynamických stínů (*dynamic shadow cas*t).
9. 8 bodových světel ve scéně; nastavení ambientního, difuzního a speculárního odrazu; nastavení kvadratického slábnutí světla; možnost vložit částicový efekt na pozici světla.
10. Nastavení mlhy (barva, hustota).
11. Nastavení kamery (úhel perspektivy, zadní ořezávací rovina, výška kamery).
12. Skybox (nastavení velkosti a libovolné textury na každou stranu této krychle).

7.1 Softwarové prostředky

7.1.1 wxWidgets

Multiplatformní open source C++ API pro vývoj GUI dektopových a mobilních aplikací [27], [38]. Toolkit podporuje OpenGL, síťové prvky, schránky, vícevláknové

programování, správu obrázků, databáze, HTML, práci ze soubory, XML, správu paměti a další užitečné funkce. Aplikaci napsanou pod wxWidgets lze zkompilovat pod Win32, MacOS X, Linux, GTK, Windows CE a pracuje se na podpoře PalmOS.

Obr. 29. wxWidgets

Počátky wxWidgets sahají do roku 1992, kdy Julian Smart na Univerzitě v Edinburgu vytvořil nástroj nazvaný Hardy. Nedlouho poté byla na FTP univerzity umístěna první verze wxWidgets (tehdy ještě pojmenována wxWindows). V roce 1997 vzniká verze 2.0. V roce 1999 se do projektu připojuje i čech Václav Slavík a implementuje wxHTML třídy (3D engine tyto objekty využívá). V současné době je wxWidgets ve verzi 2.6.3 (engine je napsán pod verzí 2.6.2). Ze známých aplikací využívá wxWidgets AVG AntiVirus nebo MojoWorld. Knihovnu používá také AMD, NASA, Xerox.

7.1.2 OpenGL 2.0

Grafické rozhraní je naprogramováno pod OpenGL 2.0. Podrobněji viz kapitola 2.1.

7.1.3 The OpenGL Shading Language

Pro *shadering* je použit jazyk GLSL. Podrobněji viz 2.3.1 a kapitola 6.

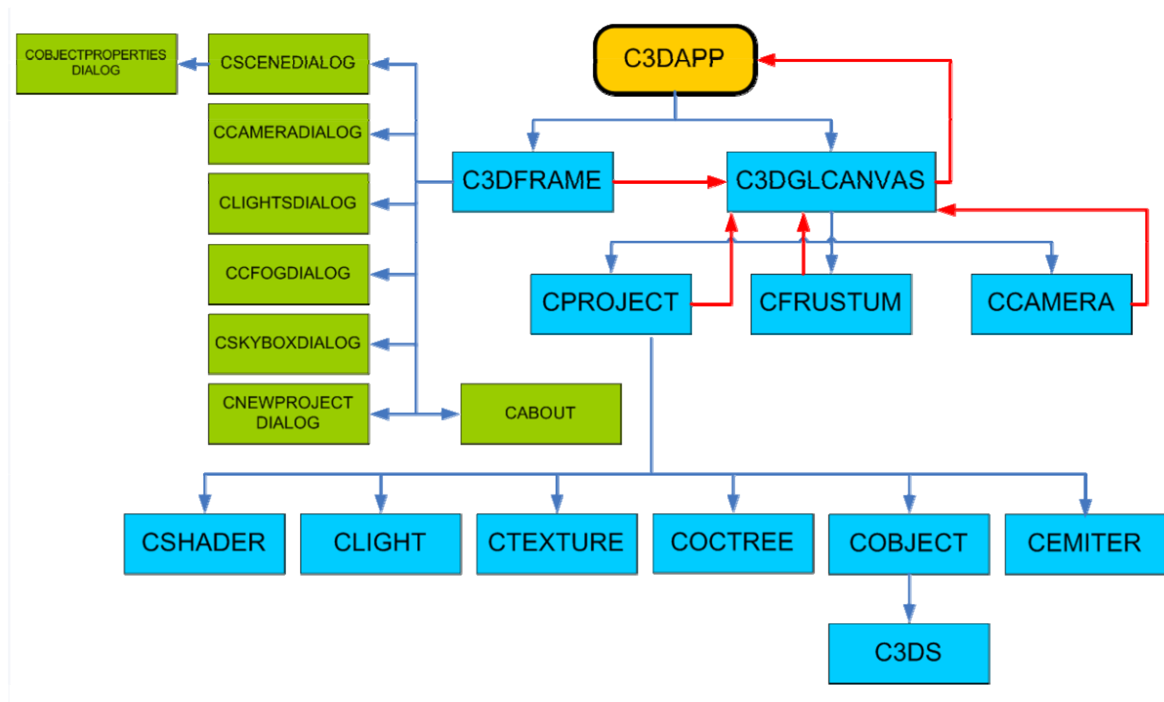
7.1.4 MS Visual Studio C++ .NET

Mateřská Windows verze Win32 aplikace je naprogramována pod vývojovým prostředím MS Visual Studio C++ .NET.

7.2 Objektová struktura enginu

Objektová struktura aplikace je uvedena na (Obr. 30). Základem je třída *C3DApp*, která obsahuje třídu *C3DFrame* a *C3DGLCanvas*. *C3DFrame* třída ovládá hlavní okno aplikace a všechny dialogy v aplikaci. *C3DGLCanvas* je objekt odvozený od wxWidgets třídy *WXGLCANVAS* a zpřístupňuje OpenGL renderovací kontext. Třída dále obsahuje *CProject* zajišťující správu vizualizačního projektu, *CFrustum* určuje pohledový objem a *CCamera* snímá scénu.

Objekt *CProject* drží informace o aktuálně editovaném objektu (geometrie, topologie, textury a parametry), ukládá a nahrává projekty do/z XML souboru.



Obr. 30. Objektová struktura aplikace

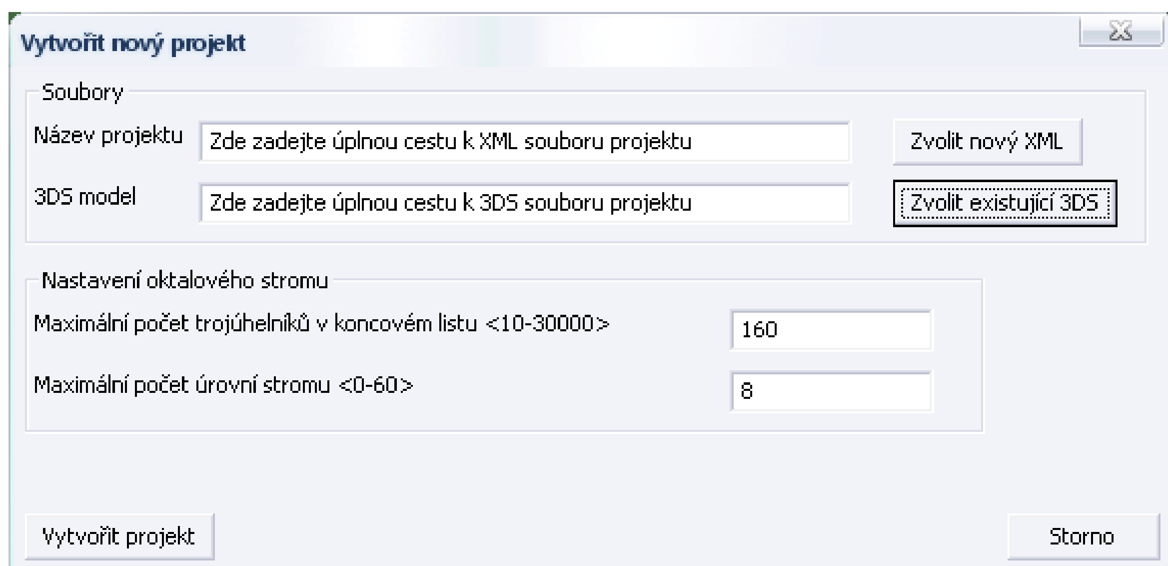
8 POPIS APLIKACE WXVIZUA 3D

8.1 Dialogová okna

Ke komunikaci s uživatelem slouží menu aplikace, toolbar, klávesové zkratky a tématická dialogová okna. Aplikace využívá i standardní dialogy poskytované API, především dialogy pro výběr barev a dialogy pro nahrání a uložení projektu. Následující text popisuje design tříd dialogových oken uvedených na (Obr. 30).

8.1.1 Dialog nového projektu

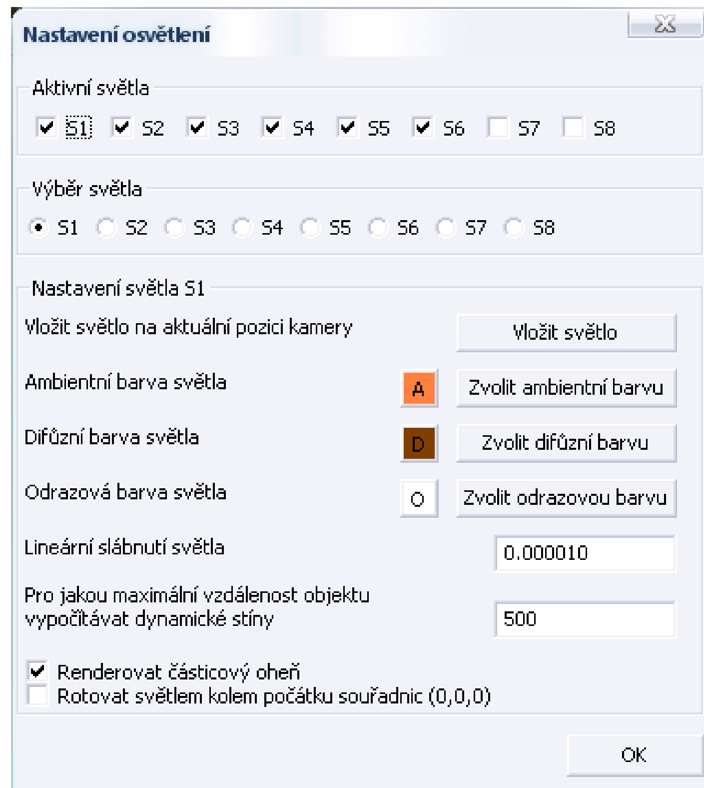
Třída *CNewProjectDialog* vytváří dialog pro vytvoření nového projektu. Okno umožňuje zvolit XML soubor projektu, cestu k 3DS modelu (musí být ve stejném adresáři). Dále lze nastavit parametry oktalového stromu.



Obr. 31. Dialog nového projektu

8.1.2 Dialog nastavení světel ve scéně

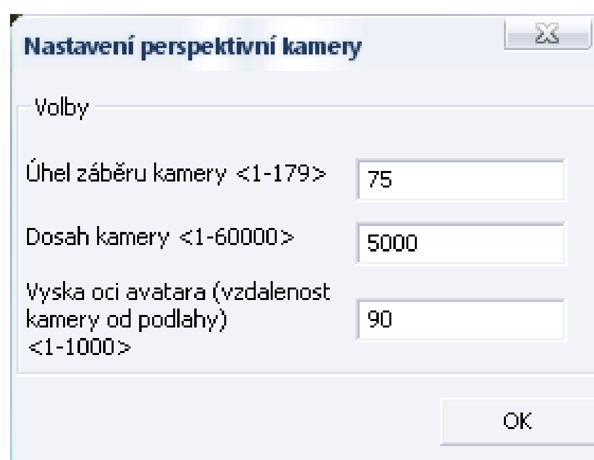
Nastavení parametrů světel ve scéně (Obr. 32) ovládá třída *CLightsDialog*. Lze modifikovat 8 světel. První světlo vrhá dynamický stín, ostatní světla tento stín nevrhají a slouží ke klasickému osvětlení objektů. Volba *Renderovat částicový oheň* aktivuje částicový efekt ohně v místě kamery. *Rotace světla* spustí rotování daného světla kolem počátku souřadnicového systému (rotace pomocí *quaternionů*).



Obr. 32. Dialog nastavení osvětlení

8.1.3 Dialog pro nastavení kamery

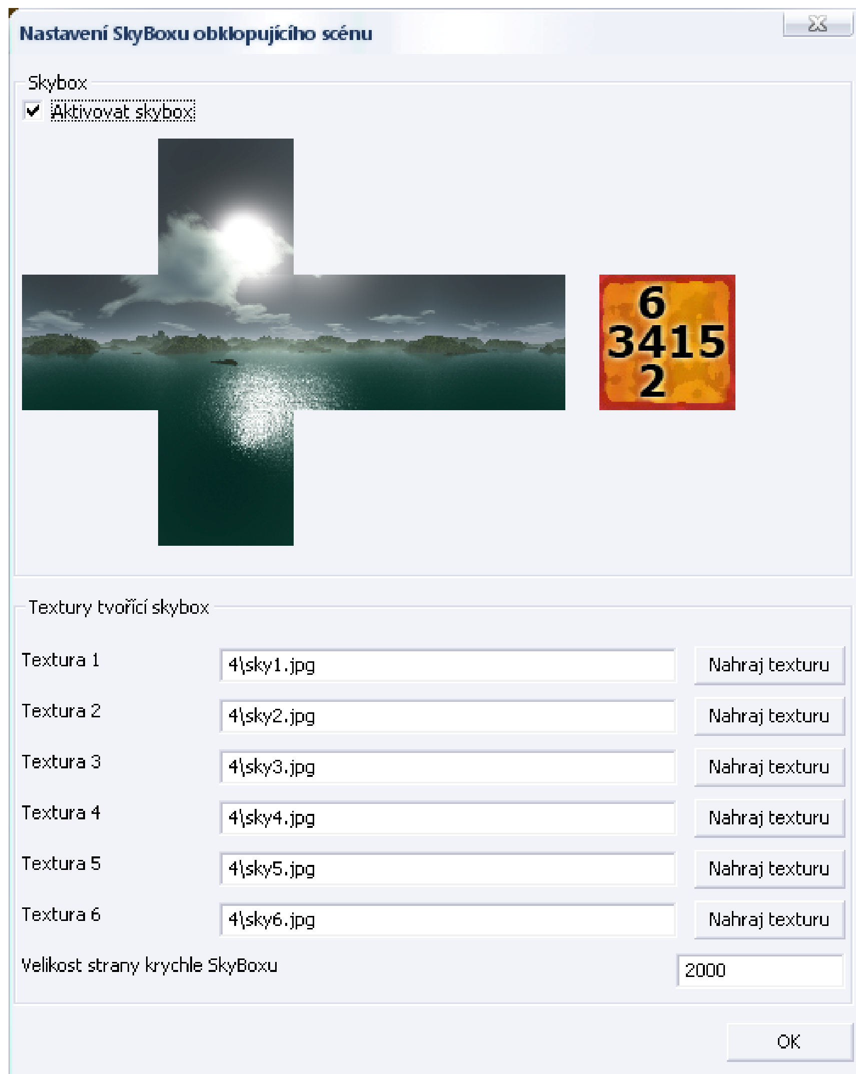
Třída CCameraDialog nastavuje perspektivní kameru snímající scénu. Lze nastavit záběr kamery, dosah kamery a výšku kamery nad povrchem (používáno pokud je aktivován pohyb avatara ve stylu počítačové hry).



Obr. 33. Dialog nastavení kamery

8.1.4 Dialog pro nastavení skyboxu

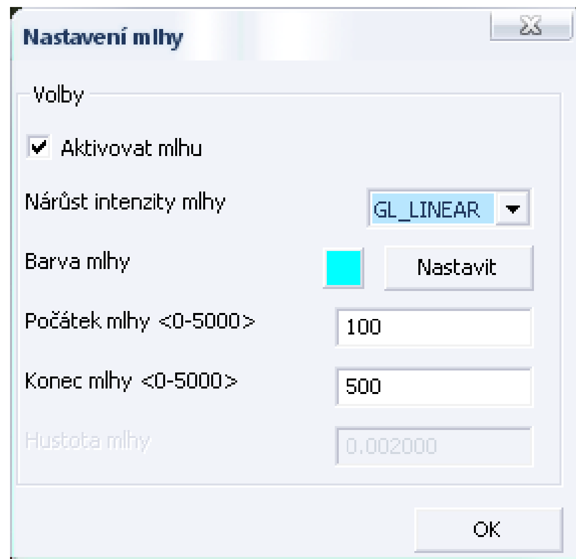
Celou scénu lze uzavřít do krychle na jejíž strany promítneme textury. Při vhodném umístění a volbě textur vznikne uživateli dojem, že ho obklopuje reálný svět. Toto nastavení umožňuje dialogová třída *CSkyBoxDialog*. Textbox *Velikost strany krychle SkyBoxu* určuje velikost této krychle.



Obr. 34. Dialog pro nastavení SkyBoxu obklopujícího scénu

8.1.5 Dialog pro nastavení mlhy

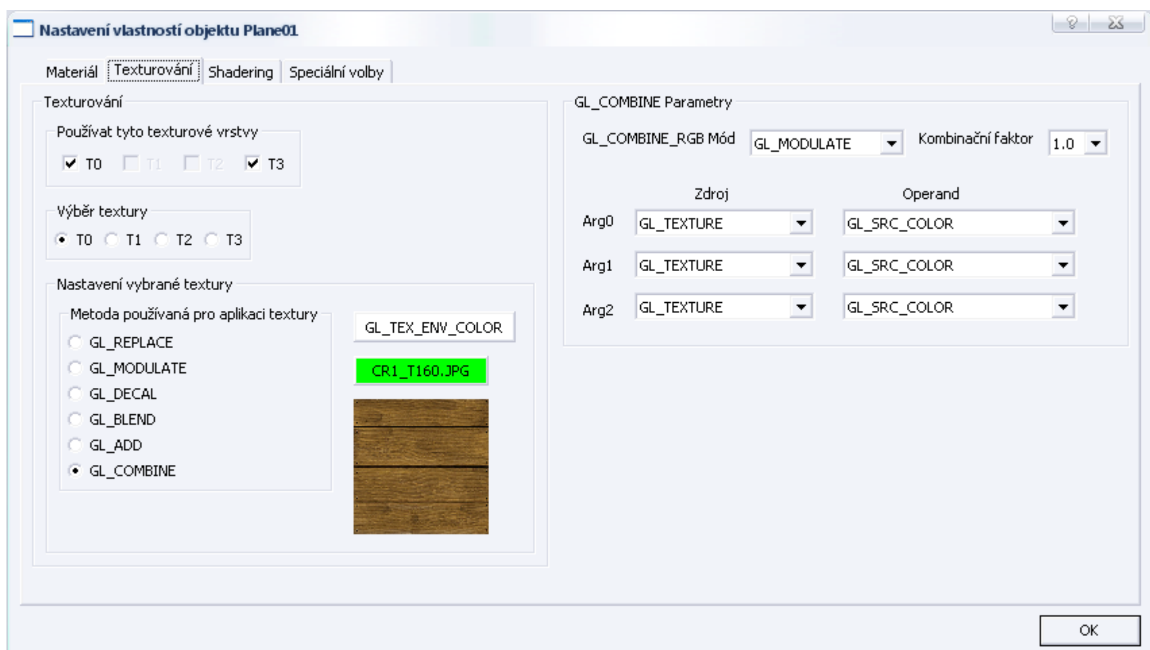
Ve scéně lze aktivovat mlhu. Pro nastavení parametrů slouží dialog *CFogDialog* (Obr. 35). Uživatel má na výběr barvu a tři typy mlhy: lineární a dva druhy exponenciální. Lze nastavit parametry určující intenzitu mlhy.



Obr. 35. Dialog nastavení mlhy

8.1.6 Dialog pro nastavení vlastností objektu

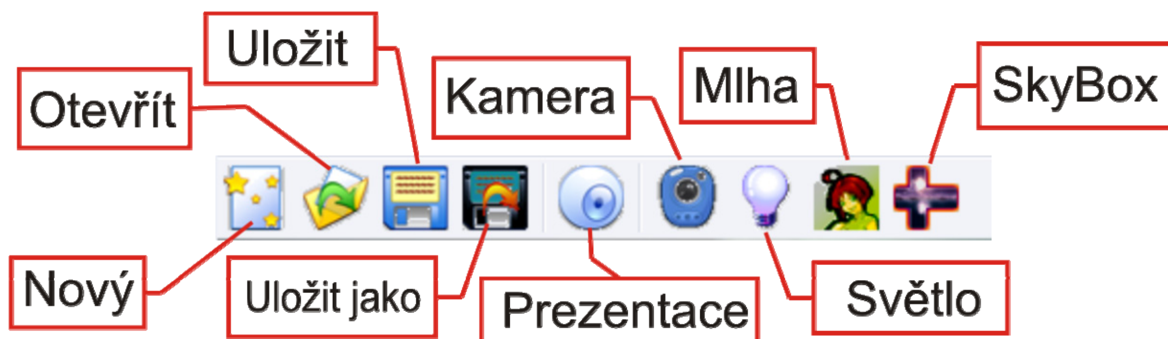
Třída *CObjectProperties* umožňuje nastavit materiálové, texturové s shaderové vlastnosti. Lze nastavit také další speciální parametry pro každý jednotlivý objekt ve scéně (průhlednost, masking,…). Většinu těchto nastavení lze upřesňovat až na úroveň parametrů OpenGL funkcí. Uživatel má velkou volnost pro specifikaci vlastností objektu. Na (Obr. 36) je zobrazena záložka Texturování, design ostatních záložek je v Příloze VI.



Obr. 36. Dialog nastavení texturových vlastností objektu

8.2 ToolBar

ToolBar umožňuje pohodlně ovládat aplikaci. Popis jednotlivých ikon je na (Obr. 37).



Obr. 37. Popis ToolBaru v aplikaci

8.3 Ovládání programu

Aplikace se ovládá pomocí klávesnice a myši. Aktivními prvky jsou hlavní menu aplikace, toolbar, klávesové zkratky a selekce objektů pomocí myši. Podrobnější popis ovládání lze najít v nápovědě programu (*menu Nápověda nebo klávesa F1*)

8.4 Obsah CDROM

1. V adresáři ***bin*** je umístěna aplikace wxVIZUA3D s několika ukázkovými projekty.
2. V adresáři ***text*** je umístěna tato textová verze diplomové práce ve formátu DOC a PDF.
3. V adresáři ***source*** se nachází kompletní zdrojový kód aplikace ve formě MS Visual C++ .NET projektu. (*3dvizualizace.vcproj*)
4. V adresáři ***tools*** lze najít nástroje použité při vytváření tohoto softwarového díla
5. V adresáři ***resources*** jsou umístěny elektronické materiály na jejichž základě jsem vytvářel diplomovou práci (dokumenty, prezentace, manuály)

ZÁVĚR

Cílem této práce bylo navrhnout a implementovat 3D grafický engine a analyzovat současný stav poznatků v oblasti moderní realtime počítačové grafiky a grafického hardware.

Aplikace je naprogramována v jazyku C++ a vzhledem k použitému API wxWidgets a grafickému rozhraní OpenGL je možné provozovat software nejenom na platformě Windows, ale také na Linux a MAC OS. Program a jeho jádro se zakládá na hierarchické acyklické grafové struktuře scény v podobě oktalového stromu jehož koncové listy jsou testovány na viditelnost pomocí metody *frustum culling*. Aplikace podporuje zobrazování ostrých dynamických stínů pomocí techniky *depth pass stenciled shadows volumes*. Engine podporuje shadery vytvořené v jazyku *The OpenGL Shading Language*. Aplikace uchovává vytvořené projekty v podobě XML souboru.

Výhledově je možné vylepšit jádro enginu o reprezentaci scény v podobě BSP stromu. Implementovat další techniky výpočtu stínů a celkově lépe optimalizovat zobrazovací část.

Software (engine) lze využít jako nástroj pro prezentaci trojrozměrných dat, vizualizaci architektonických a technologických řešení. Může sloužit jako základ počítačové hry nebo aplikace zaměřené na virtuální realitu.

SEZNAM POUŽITÉ LITERATURY

[1] AMBROŽ, D. *Dynamické stíny* [online]. ČVUT, Praha, 2006, [cit. 2006-05-17]. Dostupné z WWW: < http://www.shadowstechniques.com>

[2] AMBROŽ, D. *Shadow maps* [online]. ČVUT, Praha, 2006, [cit. 2006-05-17]. Dostupné z WWW: < http://www.shadowstechniques.com>

[3] *ARCHITECTURE REVIEW BOARD* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: <http://www.opengl.org/about/arb/>

[4] *ATI TECHNOLOGIES – Developer pages* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: < http://www.ati.com/developer>

[5] *BROOKGPU – Stream program language* [online]. 2006, [cit. 2006-05-17], Stanford University, California, USA, Dostupné z WWW: <http://graphics.stanford.edu/projects/brookgpu/>

[6] *BSP FAQ – BSP Frequently Asked Questions* [online]. 2000, [cit. 2006-05-17]. Silicon Graphics, Inc., Dostupné z WWW: <ftp://ftp.sgi.com/other/bspfaq/>

[7] *DIRECT X – Microsoft DirectX homepage* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: <http://www.microsoft.com/directx/>

[8] FILLER, T., VACATA J. *Základy GPGPU* [online]. FJFI-ČVUT, Praha,2006, [cit. 2006-05-17]. Dostupné z WWW: <http://kmlinux.fjfi.cvut.cz/~fillert1/paa/>

[9] FOUSEK, M. *X3D* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: <http://www.cgg.cvut.cz/~apg/apg-tutorials03/ch13s09.html>

[10] GERGELIŠOVÁ, Š. *VRML v příkladech.*. BEN – technická literatura, Praha, 2004, ISBN 80-7300-138-1

[11] *GPGPU – General Purpose Computation Using Graphics Hardware* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: <http://www.gpgpu.org>

[12] GRIB, G., HARTMANN, K. *Fast extraction of viewing frustum planes from the world-view-projection matrix.* 2001

[13] HARRIS, M. *Simulation on GPUs – simulation of natural phenomena using graphics hardware* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: <http://www.markmark.net/cml/>

[14] *NVIDIA CORPORATION – Software developer Kit* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: <http://developer.nvidia.com>

[15] NVIDIA CORPORATION. *NVIDIA GPU Programming guide version 2.4.0,* NVIDIA Corporation, Austin, Texas, 2005

[16] NYTRA, D., *Živé divadlo.* Diplomová práce FEL-ČVUT, Praha, leden 2005

[17] HUDEC, B. *Základy počítačové grafiky*. ČVUT, Praha,2001, ISBN 80-01-02290-0

[18] KESSENICH, J., BALDWIN, D., ROST, R. *The OpenGL Shading Language*. 3Dlabs, Inc. Ltd., 2004

[19] MARTŠEK, D. *Matematické principy grafických systémů*. Litera, Brno, 2002, ISBN 80-85763-19-2

[20] KILGARD, M., EVERITT, C. *Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering*. NVIDIA Corporation, Austin, Texas, 2002

[21] NEIDER, J., DAVIS, T. *The official guide to learning OpenGL, Version 1.1.* Adision Wesley Longman Inc., Silicon Graphics, Inc., USA, 1997

[22] *OPENGL – The Industry standard for High Performance Graphics* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: <http://www.opengl.org>

[23] PELIKÁN, J. *3D počítačová grafika na PC* [online]. 2003, [cit. 2006-05-17]. MFF UK, Praha, Dostupné z WWW: <http://cgg.ms.mff.cuni.cz/~pepca>

[24] PELIKÁN, J. *Hardware pro počítačovou grafiku (prezentace)* [online]. 2005, [cit. 2006-05-17]. MFF UK, Praha, Dostupné z WWW: <http://cgg.ms.mff.cuni.cz/~pepca>

[25] *PIXEL.* Vydává ATLANTIDA Publishing, s.r.o., 2006, č.3/2006, č.4/2006, č.5/2006, 12x ročně

[26] SEGAL, M., AKELEY, K. *The OpenGL Graphics System: A Specification (Version 2.0).* Silicon Graphics, Inc., 2004

[27] SMART, J., HOCK, K. *Cross platform GUI programming with wxWidgets*. Pearson Education, Inc., USA, 2006, ISBN 0-13-147381-6

[28] *SH – Embedded Metaprogramming Language* [online]. 2006, [cit. 2006-05-17], University of Waterloo, Canada, Dostupné z WWW: < http://www.libsh.org>

[29] TATARCHUK, N. *Practical Parallax Occlusion Mapping For Highly Detailed Surface Rendering.* 3D Application Research Group, ATI Research, Inc, Game Developers Conference, March 2006

[30] TIŠŇOVSKÝ, P. *Grafické karty a grafické akcelerátory* [online]. 2005, [cit. 2006-05-17]. Dostupné z WWW: <http://www.root.cz/clanky/graficke-karty-a-graficke-akceleratory-21/>

[31] TIŠŇOVSKÝ, P. *Grafická knihovna OpenGL* [online]. 2005, [cit. 2006-05-17]. Dostupné z WWW: <http://www.root.cz/clanky/opengl-33-souhrn-temat/>

[32] TOLAR, V. *X3D* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: <http://www.cgg.cvut.cz/~apg/apg-tutorials01/ch03s20.html>

[33] VIRIUS, M. *Programování v C++*. ČVUT, Praha, 2001. ISBN 80-01-01874-1

[34] *VRML & X3D CONSORCIUM* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: <http://www.web3d.org>

[35] *VRML SPECIFICATION* [online]. 1996, [cit. 2006-05-17]. Dostupné z WWW: <http://www.graphcomp.com/info/specs/sgi/vrml/spec/>

[36] WALNUM, C. *Programujeme grafiku v Microsoft Direct3D.* Computer press, Brno, 2004. ISBN 80-251-0136-3

[37] WROBLEWSKI, P. *Algoritmy, Datové struktury a programovací techniky.* Computer press, Brno, 2004. ISBN 80-251-0343-9

[38] *WXWIDGETS LIBRARY SDK KIT* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: <http://www.wxwidgets.org>

[39] ZAPRJAGAEV, A. *Frustum.org* [online]. 2006, [cit. 2006-05-17]. Dostupné z WWW: <http://www.frustum.org>

[40] ŽÁRA, J. *Jazyky pro popis virtuální reality.* ČVUT, Praha, 2000, ISBN 80-01-02100-9

[41] ŽÁRA, J. a kolektiv. *Počítačová grafika – principy a algoritmy*. Grada, Praha, 1992, ISBN 80-85623-00-5

[42] ŽÁRA, J., BENEŠ, B., FELKEL, P. *Moderní počítačová grafika*. Computer Press, Praha, 2004, ISBN 80-251-0454-0

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	**A**pplication **P**rogram **I**nterface. Aplikační programové rozhraní.
BSP	**B**inary **S**pace **P**artitioning. Způsob dělení prostoru scény.
BVH	**B**ounding **V**olume **H**ierarchy. Reprezentace scény pomocí hierarchie obálek.
Cg	**C** for **g**raphics. Jazyk pro programovatelné grafické karty firmy NVIDIA.
CPU	**C**entral **P**rocessing **U**nit. Procesor.
GDI	**G**raphical **D**evice **I**nterface. Grafické rozhraní ve Win32.
GLSL	The Open**GL** **S**hading **L**anguage. Jazyk pro programovatelné grafické karty.
GPGPU	**G**eneral **P**urpose Computing on **G**raphics **P**rocessors. Obecné výpočty na GPU.
GPU	**G**raphics **P**roccesing **U**nit. Grafický čip.
GUI	**G**raphical **U**ser **I**nterface. Grafické uživatelské rozhraní.
HAL	**H**ardware **A**bstraction **L**ayer. Hardwarová vrstva implementovaná v graf. kartě.
HLSL	**H**igh **L**evel **S**hading **L**anguage. Jazyk pro programovatelné grafické karty.
LOD	**L**evel **O**f **D**etail. Úroveň detailů v závislosti na vzdálenosti od kamery.
Pixel	**P**icture **El**ement. Základní jednotka rastrového obrazu.
Texel	**Tex**ture **El**ement. Základní jednotka maticové textury.
VRML	**V**irtual **R**eality **M**odelling **L**anguage. Jazyk pro popis virtuální reality
X3D	e**X**tensible **3D**. Jazyk pro popis virtuální reality.

SEZNAM OBRÁZKŮ

Obr. 1. Pás (strip) a vějíř (fan) trojúhelníků 12
Obr. 2. OpenGL 14
Obr. 3. Grafický zobrazovací řetězec OpenGL 14
Obr. 4. Hierarchie vrstev aplikace v Direct3D 16
Obr. 5. VRML97 17
Obr. 6. VRML v internetovém prohlížeči 18
Obr. 7. X3D 18
Obr. 8. Blokové schéma moderního grafického akcelerátoru [30] 20
Obr. 9. Transformace prováděné vertex shaderem v pevném režimu [30] 23
Obr. 10. Transformace prováděné vertex shaderem v programovatelném režimu [30] 25
Obr. 11. Operace prováděné fragment shaderem v pevném režimu [30] 27
Obr. 12. Výkonná část fragment shaderu [30] 28
Obr. 13. Fragment shader v programovatelném režimu [30] 29
Obr. 14. Výkon GPU a CPU [11] 30
Obr. 15. BSP strom 34
Obr. 16. Oktalový strom v enginu 35
Obr. 17. Dělení prostoru oktalovým stromem 35
Obr. 18. Komolý jehlan perspektivy 37
Obr. 19. Projekce 3D tělesa do 2D roviny [1] 40
Obr. 20. Stínová paměť hloubky [1] 41
Obr. 21. Stínový objem 42
Obr. 22. Schéma hledání siluety objektu 44
Obr. 23. Vizualizace stínového objemu v enginu 45
Obr. 24. Princip depth-pass algoritmu [23] 46
Obr. 25. Engine: Vlevo bez shaderu, vpravo aplikován GLSL Relief shader 48
Obr. 26. GLSL shader v enginu vytvářející efekt lomu světla na skle 49
Obr. 27. Shader Designer – pokročilý nástroj pro programování GLSL shaderů 50
Obr. 28. wxVIZUA 3D v editačním módu 53
Obr. 29. wxWidgets 55
Obr. 30. Objektová struktura aplikace 56
Obr. 31. Dialog nového projektu 57

Obr. 32. Dialog nastavení osvětlení 58
Obr. 33. Dialog nastavení kamery 58
Obr. 34. Dialog pro nastavení SkyBoxu obklopujícího scénu 59
Obr. 35. Dialog nastavení mlhy 60
Obr. 36. Dialog nastavení vlastností jednotlivého objektu 60
Obr. 37. Popis ToolBaru v aplikaci 61

SEZNAM PŘÍLOH

Příloha PI: Vizualizace zahradního domku

Příloha PII: Použití enginu pro počítačovou hru

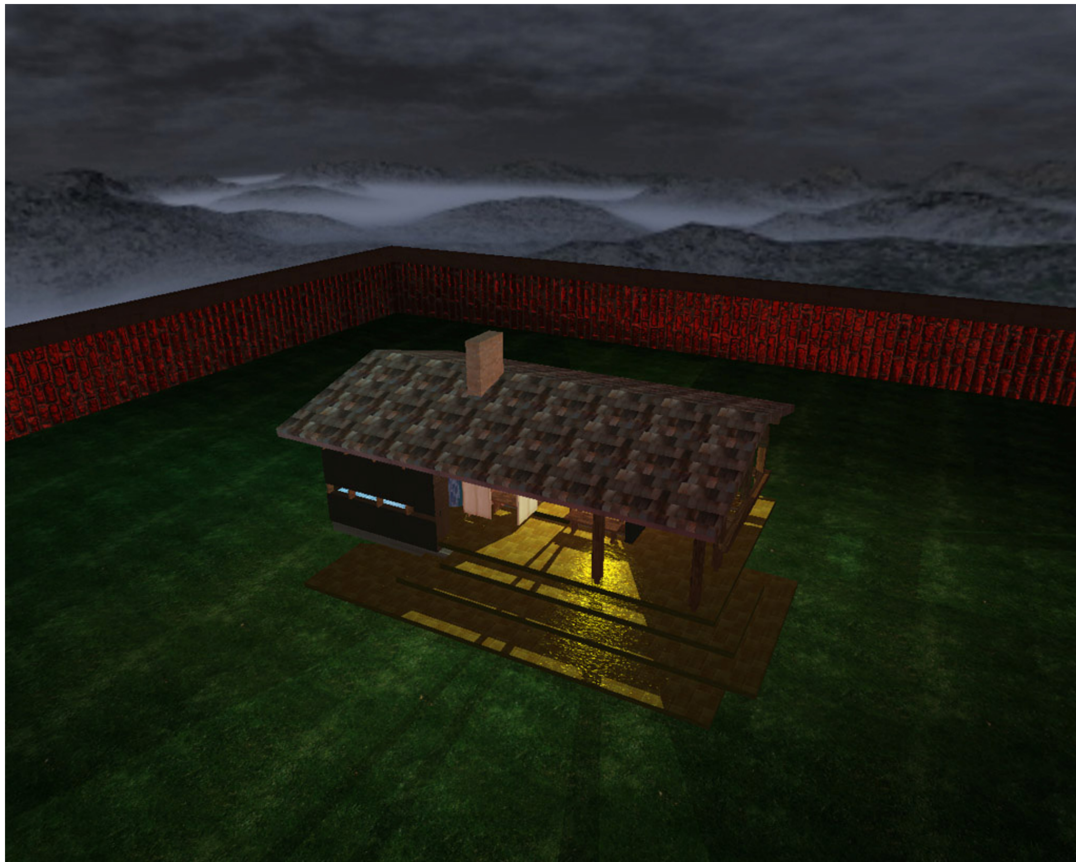
Příloha PIII: Fakulta Aplikované Informatiky

Příloha PIV: Rozdíl v kvalitě zobrazení

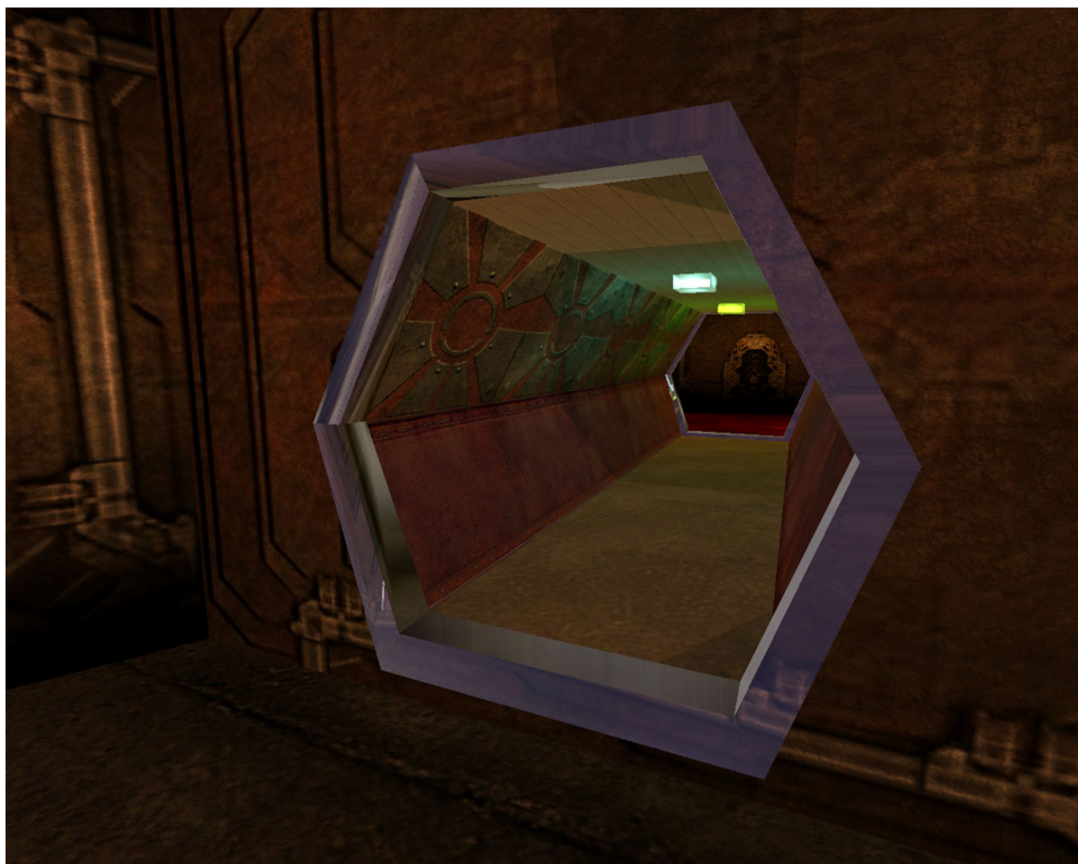
Příloha PV: Vizualizace velkoměsta

Příloha PVI: Dialogy nastavení objektu

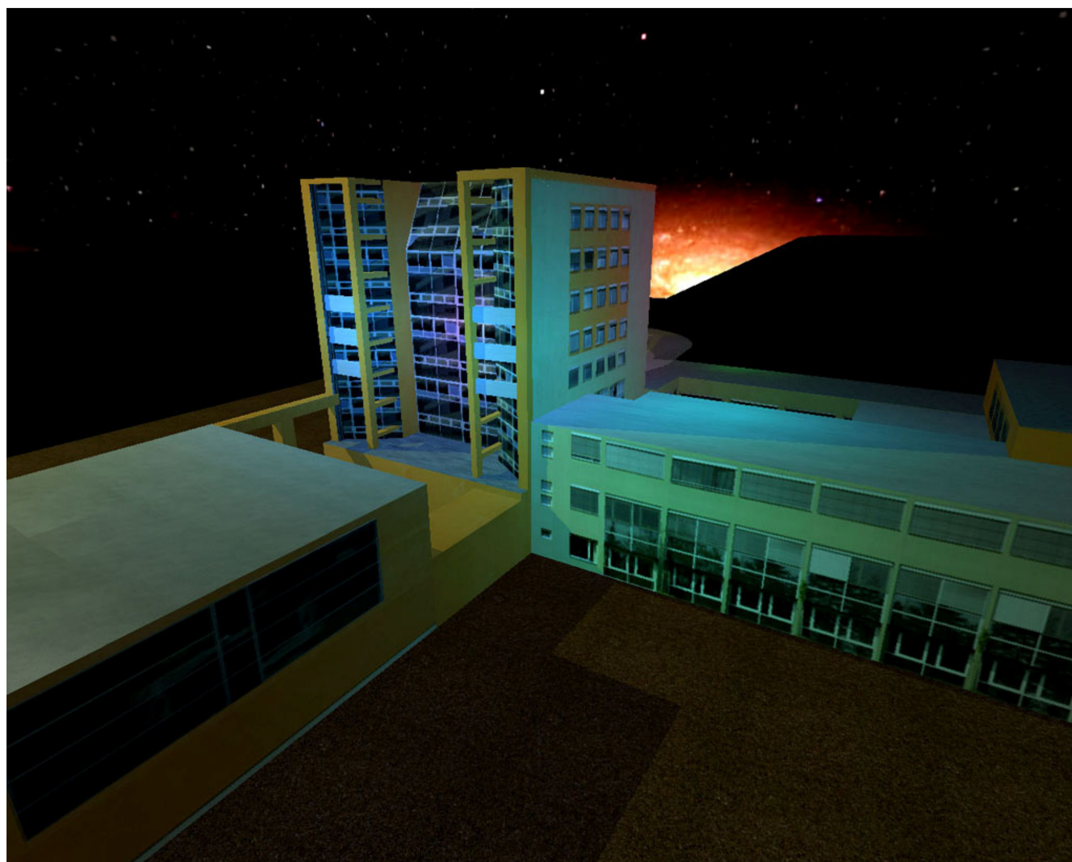
PŘÍLOHA P I: VIZUALIZACE ZAHRADNÍHO DOMKU



PŘÍLOHA P II: POUŽITÍ ENGINU PRO POČÍTAČOVOU HRU

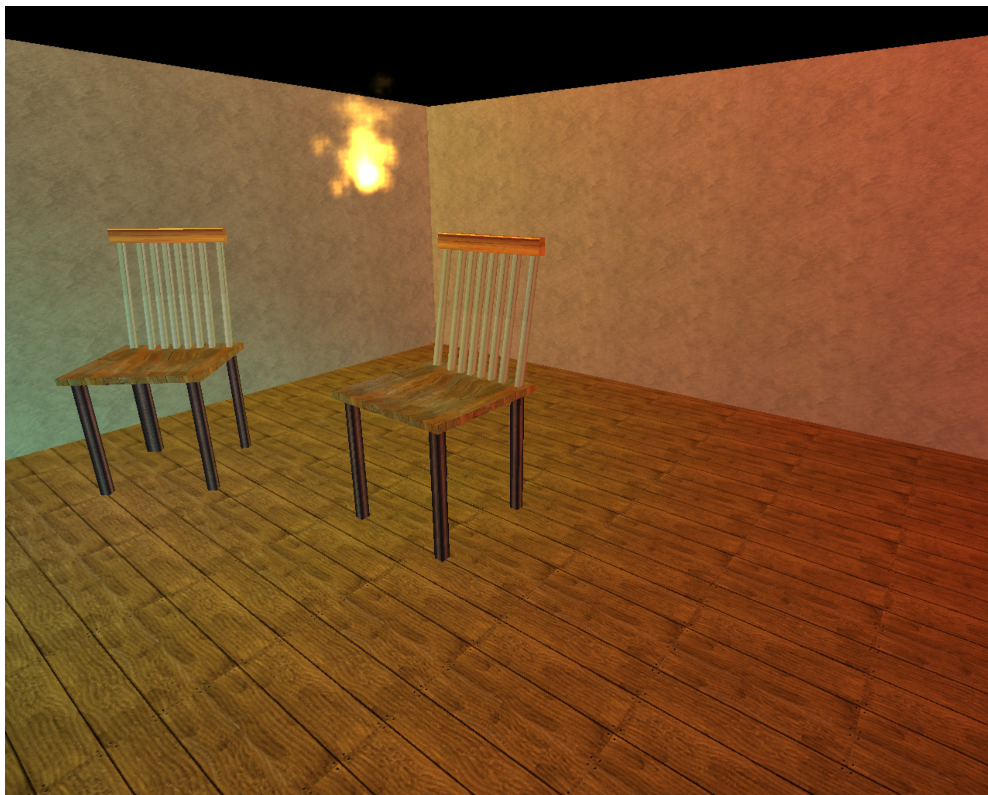


PŘÍLOHA P III: FAKULTA APLIKOVANÉ INFORMATIKY

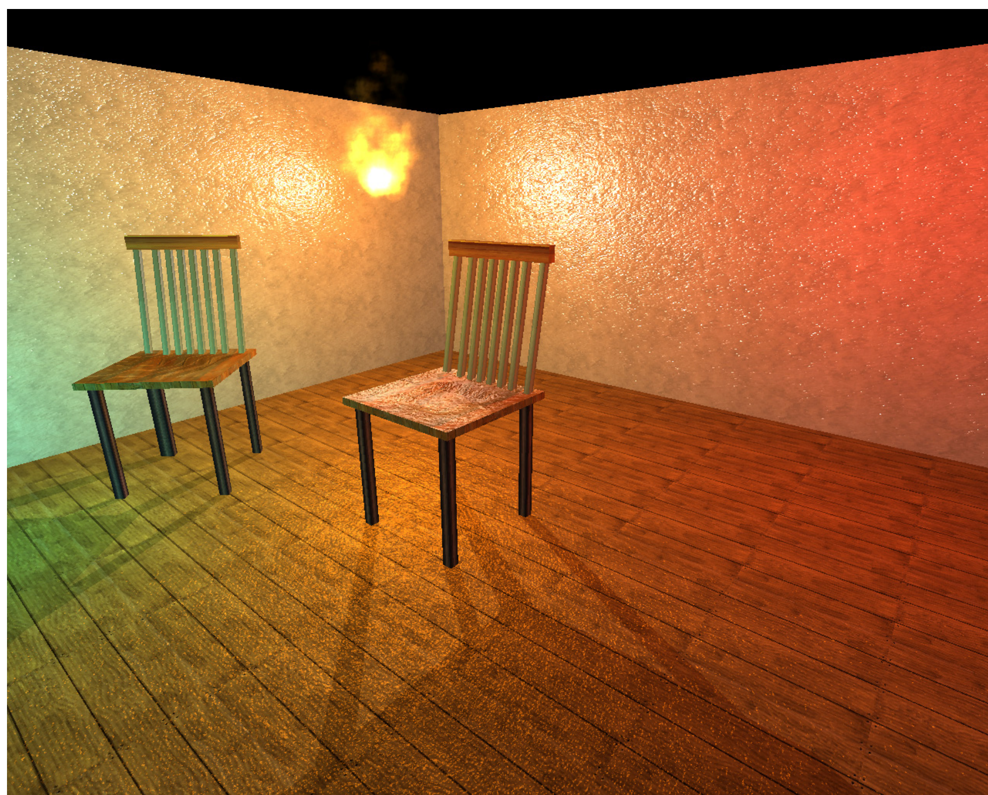


PŘÍLOHA P IV: ROZDÍL V KVALITĚ ZOBRAZENÍ

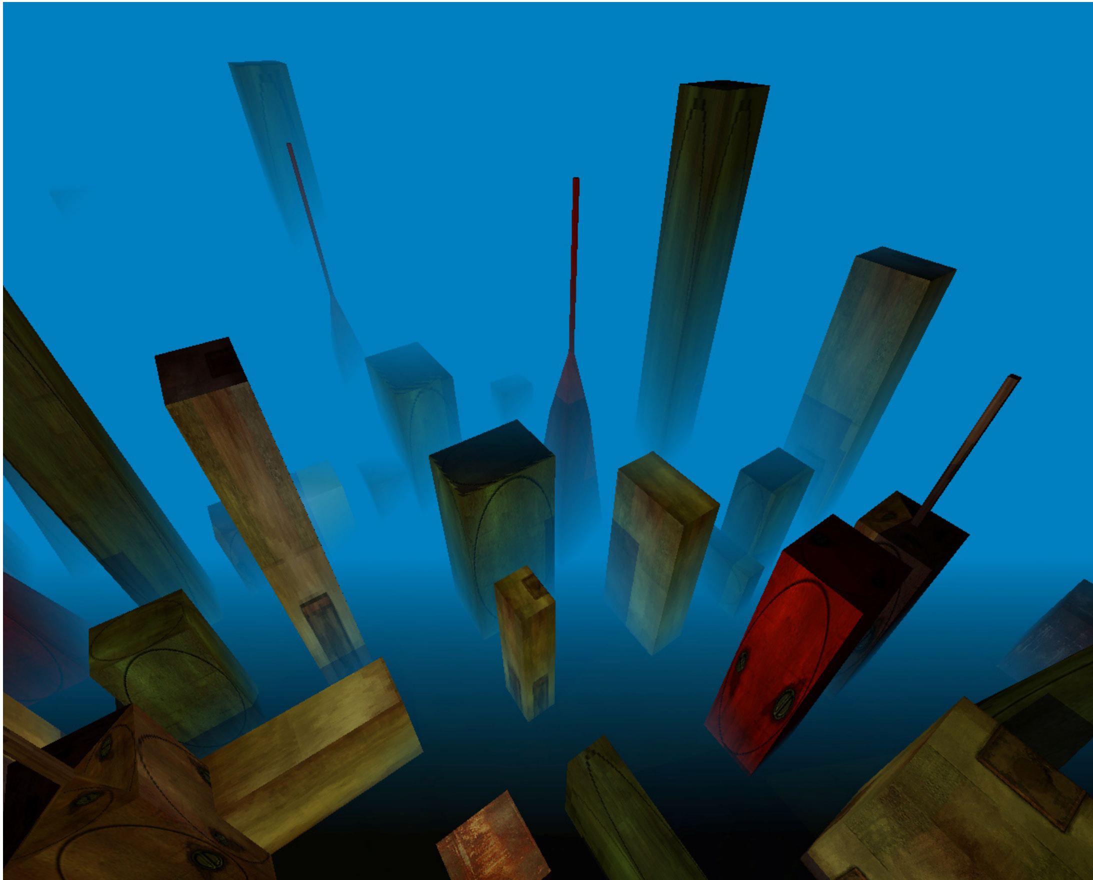
Scéna bez dynamických stínů a shader efektů

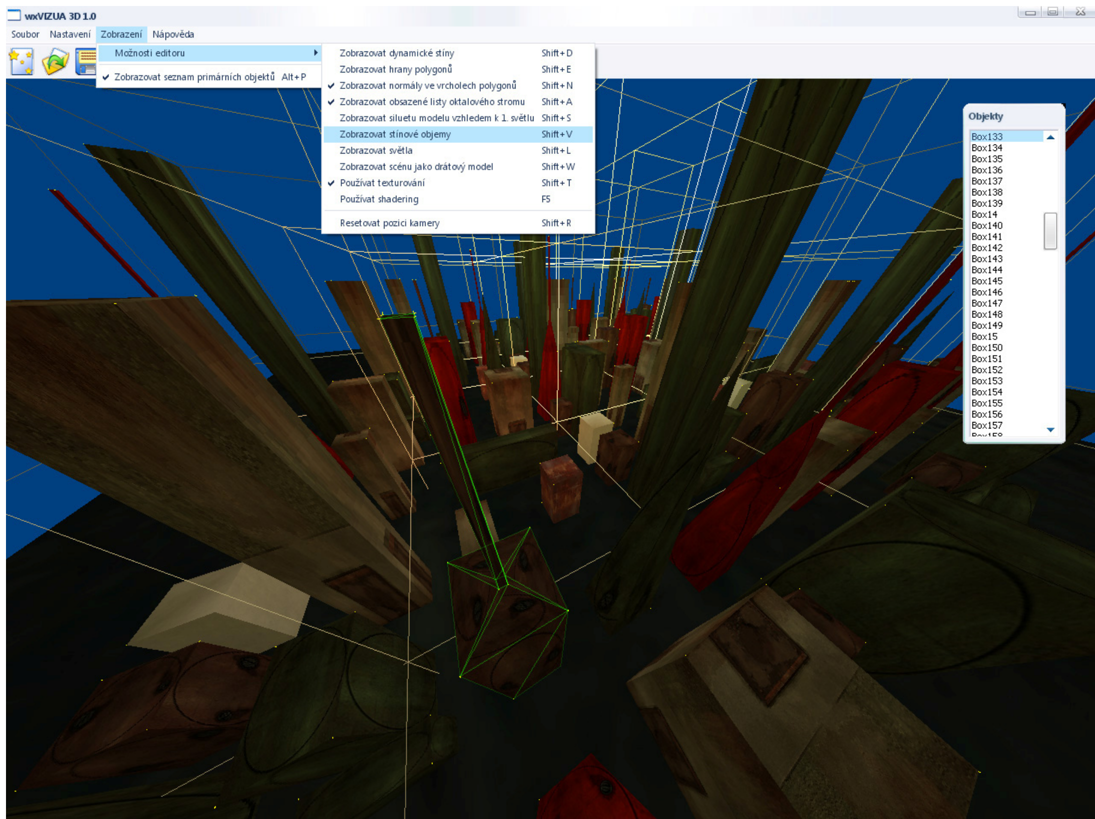


Scéna s dynamickými stíny a per pixel specular bump map shaderem



PŘÍLOHA P V: VIZUALIZACE VELKOMĚSTA





PŘÍLOHA P VI: DIALOGY NASTAVENÍ OBJEKTU