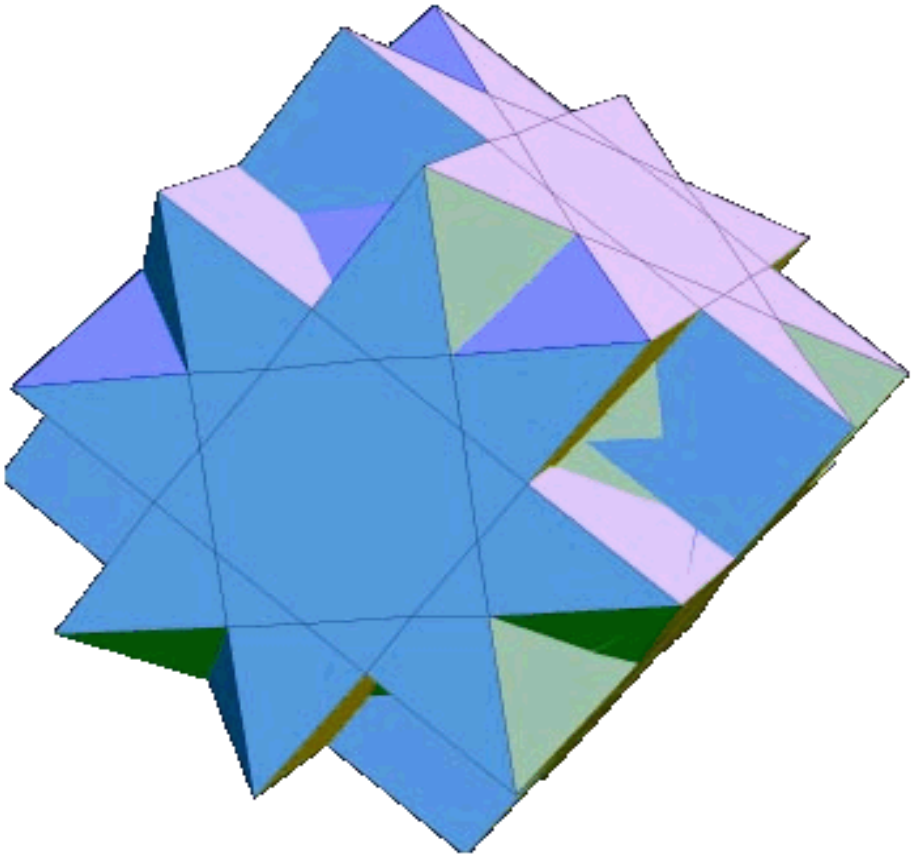


OpenGL

referát na praktikum z informatiky

Daniel Čech

Co je OpenGL

OpenGL (Open Graphics Library) je nízkoúrovňová knihovna pro práci s trojrozměrnou grafikou. Od doby svého uvedení na počátku devadesátých let se stala široce uznávaným a podporovaným standardem na poli grafických aplikací, CAD/CAM/CAE inženýrských systémů, virtuální reality a tvorby her. OpenGL představuje jednotné API (Application Programming Interface) mezi programem a grafickým hardware. Hlavními rysem OpenGL je nezávislost na cílové platformě a použitém programovacím jazyku.

Knihovna OpenGL vznikla v roce 1992 u společnosti Silicon Graphics Inc. (SGI). Jejímu uvedení předcházely výzkumy SGI v oblasti grafického modelování a realistické syntézy obrazu. V roce 1982 na Stanfordské univerzitě zrodil koncept grafického počítače, který na principu proudového zpracování obrazových elementů, vytvářel trojrozměrnou scénu. Na stejném základě se později u Silicon Graphics vyvíjely grafické stanice a knihovna IRIS GL, která představovala programátorské rozhraní právě pro podporu pipeline-renderingu. IRIS GL (IRIX je operační systém počítačů Silicon Graphics) lze považovat za přímého předka OpenGL. Obě tyto grafické knihovny mají mnoho společného. IRIS GL je souborem elementárních příkazů pro práci s 3D grafikou, které víceméně znamenají pouze volání služeb konkrétních grafických zařízení. Naproti tomu byla OpenGL od počátku koncipována jako hardwarově nezávislá. V průběhu času se IRIS GL zbavovala některých problematických rysů a dá se říci, že poslední verze IRIS GL je s OpenGL téměř kompatibilní.

Pokud se řekne OpenGL, má se na mysli standard API, který popisuje množinu funkcí a procedur s přesně specifikovaným chováním. Jak už plyne z názvu, OpenGL je otevřeným standardem. To znamená, že se autor specifikace nestojí nutně za každou realizací OpenGL na určité platformě. Dohlíží pouze na to, je-li implementace v souladu s jeho specifikací. Tuto roli převzalo konsorcium ARB (Architecture Review Board), do kterého patří firmy Digital Equipment Corporation (DEC), Hewlett-Packard, IBM, Intel, Intergraph, Microsoft, Silicon Graphics a Sun Microsystems. Implementací se zabývají společnosti působící v oblasti vývoje grafických zařízení. Efektivita implementace je potom předmětem konkurenčního soupeření. Každá implementace, která usiluje o označení OpenGL, musí projít sérií testů (Conformance Tests), které musí zcela splnit. Provádění testů patří rovněž pod pravomoci sdružení ARB. Další nezávislé uskupení, OpenGL Characterization Committee, se zabývá vytvářením a prováděním srovnávacích testů implementací. Důsledkem přijetí standardu je klid v duši především výrobců hardwaru. Jasné výhody z toho plynou i pro koncového uživatele.

Protože je OpenGL používána již osm let, nevyhnula se za dobu své existence určitým inovacím. Současná verze knihovny je 1.2 a je popisována standardem *OpenGL Graphics System 1.2 - A Specification* (Red Book). Podobně jako předchozí verze, i verze 1.2 do sebe pojala nové rysy, které byly dříve dostupné pouze jako systémová rozšíření. Přehled nových, doposud nezařazených vlastností a funkcí, si je možné prohlédnout na OpenGL stránce SGI.

Licenční podmínky OpenGL lze shrnout do několika bodů. Nejvýznamnější důsledek je přitom to, že vývojář, který ve svém programu využije služeb knihovny OpenGL, nemusí platit žádné poplatky. Oproti tomu autor určité implementace OpenGL podléhá jedné ze tří úrovní licenčního programu. Level 1 License se týká prodejců implementací OpenGL v binární podobě, a poplatky s ní spojené jsou nízké. Level 2 License je určena pro různé knihovní nadstavby konkrétních implementací (GLU, GLUT). Level 3 License řeší distribuci zdrojových textů implementací.

Existují freewarové implementace, např. MesaGL, MGL, atd., přičemž mnohé z nich neprošly atestací ARB, nejedná se proto o plnohodnotné OpenGL implementace. MesaGL je velmi populární na Linuxu a podléhá GPL licenci.

V současnosti najdeme OpenGL na téměř všech platformách, z nichž jmenujme ty nejvýznamnější – IRIX, Unixy, Windows 95/98/NT, DOS, Linux, Solaris, MacOS, OS/2, NeXTSTeP, OPENStep a BeOS. Trochu nečekaně lze OpenGL nalézt i na některých palmtopech. OpenGL využívá značné možství komerčních programů. Z vizualizačních programů například Kinetix 3D Studio MAX, Alias | Wavefront Maya, Caligari trueSpace nebo LightWave 3D. Z oblasti CAD potom MicroStation, CATIA nebo SolidWorks. Konečně z her využívá OpenGL mimojiné Quake 2, Unreal, Sin nebo Half Life.

První seznámení

Pro běžného uživatele je OpenGL standardem, díky kterému mohou aplikace plně využívat možností akcelerované grafiky. I v případě, že uživatel nemá v počítači zabudovánu výkonnou grafickou kartu, může provozovat grafické aplikace OpenGL podporující. Potom se ovšem všechny výpočty realizují softwarově, což je sice o poznání pomalejší, ale v praxi pořád ještě použitelné.

Z pohledu programátora je OpenGL soubor asi 150 funkcí, které umožňují specifikovat geometrické objekty ve dvou nebo třech dimenzích. Takto popsané objekty potom umožňuje dále zpracovávat. Autor programu přitom nemusí detailně znát konfiguraci počítače, na kterém jeho program poběží. Při značně zjednodušeném pohledu lze říci, že programátor pouze udává souřadnice vrcholů trojúhelníků společně s informací o mapování textury, vlastní vykreslování už za něj vyřídí OpenGL. Má přitom možnost zapínat a vypínat různé její dílčí funkce, a tak přímo ovlivňovat výslednou kvalitu vytvářeného obrazu. Komplexnější vykreslení scény znamená často velmi malý nárůst kódu.

Renderovací možnosti OpenGL jsou poměrně široké. Od zobrazování jednoduchých drátových modelů těles, přes ploché stínování (flat shading) až po komplexní zobrazení scény s definovanými světelnými zdroji, průhlednými či lesklými stěnami objektů, texturami, Goraudovým stínováním, mlhou atd.

Autoři OpenGL zvolili přístup, kde jsou objekty scény popsány ne svým objemem, ale plošnými útvary, které určují jejich povrch. Tato hraniční reprezentace (boundary representation) se ukazuje jako podstatně výhodnější tam, kde je třeba rychlé vykreslování.

Scéna se komponuje z elementárních objektů, což jsou právě a jenom body, čáry, polygony, výřezy obrázku (pixel rectangles) a bitmapy. Elementární objekty - primitiva jsou dány skupinou jednoho nebo více vrcholů. Vrcholy (vertexy) přitom znamenají body v trojrozměrném prostoru. Význam vrcholů v definici elementárního objektu může být různý. Podle typu potom může znamenat počáteční a koncový bod úsečky, vrchol mnohoúhelníku, apod. Veškeré další informace o scéně, jako například barva, normálový vektor nebo souřadnice textury jsou spjaty právě s vrcholy primitiv. Jednotlivé vrcholy, z nichž se elementární objekt skládá se dají zpracovávat nezávisle v daném pořadí. Právě na proudovém zpracování obrazových elementů staví OpenGL svůj grafický výkon.

Koncept zobrazování OpenGL s sebou nese i jistá omezení. Například je obtížné základními prostředky pracovat s objektem, který je oblý. Parametricky popsané plochy (Bézierovy plochy, NURBS) se nejprve musí převést na síť mnohoúhelníků, a teprve potom se mohou zobrazovat. To, že se jednotlivé komponenty obrazu zpracovávají odděleně, má vliv na výslednou kvalitu obrazu. Například výpočet osvětlení se vztahuje vždy lokálně k danému tělesu. Bez detailní simulace šíření světla ve scéně jsou výsledky ne zcela přesvědčivé. Kvalita obrazu výsledné scény se tak pouze blíží raytraycingu, tedy metodě, která nejlépe aproximuje fyzikální model. Tyto principiální nedostatky OpenGL jsou vyváženy rychlostí a jednoduchostí výpočtu. OpenGL sama o sobě nenabízí prostředky k objektovému popisu scény na hierarchickém principu. K vyřešení některých těchto nedostatků se používají pomocné knihovny:

GLU	OpenGL Utility Library
GLUT	OpenGL Utility Toolkit
GLAUX	OpenGL Auxiary Library

Lepší představa OpenGL než jako soubor funkcí je pohled na ní jako na stavový stroj. Mimo vlastní procedury má i velké množství proměnných, matic a zásobníků. K proměnným není přímý přístup z programovacího jazyka, ale operuje se s nimi k tomu určenými funkcemi. Vnitřní proměnná OpenGL (například aktuální barva) si uchovává svojí hodnotu od naplnění do té doby, než ji programátor změní. Takovýto přístup uspokojí jak programátora, tak i implementátora. Má totiž výraznou výhodu, umožnuje realizovat architekturu Client / Server. Uživatelský program - klient posílá žádost o provedení elementární operace OpenGL - serveru. Dovedeno do důsledku, složitý výpočet 3D scény může probíhat na jiném počítači, než vlastní zobrazování. Rozdělení úloh potom dává možnost velkého nárůstu distribuovaného výpočetního výkonu. Toho se s úspěchem využívá na unixových platformách.

Jak už bylo řečeno, OpenGL je koncipovaná jako systémově nezávislá. Programy napsané s jejím využitím by měly být přenositelné na úrovni zdrojového kódu. Přesto se ale použití na jednotlivých operačních systémech řídí určitými konvencemi. Pro podchycení různých specifických vlastností OS a zejména jejich okenních systémů byly vyvynuty rozšiřující knihovny.

AGL	Apple Macintosh OpenGL Extension
GLX	OpenGL Extension for X-Window System
PGL	OS/2 OpenGL Extension
WGL	OpenGL Extension for Windows 95/98/NT

Služby OpenGL jsou přístupné z většiny běžných programovacích jazyků a jejich vývojových prostředí, především C/C++, Javy, Pascalu/Delphi, Visual Basicu, Fortranu a Ady. Teoreticky není mnoho překážek, jak napasovat OpenGL téměř na libovolný jazyk, pouze je potřeba ze strany jazyka určitá podpora (knihovní moduly). Přesto ale zůstává ve vztahu k OpenGL nejrozumnější použití jazyků C a C++, protože je napojení nejpřímější a je popisováno už standardem.

Programátorské rozhraní OpenGL

Typickou distribuci OpenGL pro C/C++ tvoří knihovna *opengl.lib*, dále hlavičkové soubory *opengl.h*, *glu.h*, *glut.h*, popřípadě *glaux.h*. Inkludováním *opengl.h* do svého programu si zpřístupníme základní množinu funkcí OpenGL, datové typy a makra. OpenGL provádí veškeré své operace z důvodu zajištění kompatibility na speciálně definovaných datových typech. Pojmenování datových typů se řídí konvencí, že první dvě písmena jsou GL a potom následuje vlastní jméno typu malými písmeny. Norma udává minimální počet bytů na každý typ. Konkrétní hodnoty jsou v tabulce.

OpenGL typ	**odpovídající C typ**	**minimální počet bitů**	**přípona**
GLbyte	signed char	8-bit signed integer	b
GLshort	short	16-bit signed integer	s
GLint, GLsizei	int, long	32-bit signed integer	i
GLfloat, GLclampf	float	32-bit foating point	f
GLdouble, GLclampd	double	64-bit floating point	d
GLubyte, Glboolean	unsigned char	8-bit unsigned integer	ub
Glushort	unsigned short	16-bit unsigned short	us
GLuint, GLenum, GLbitfield	unsigned int, unsigned short	32-bit unsigned integer	ui

Podobnou konvencí jako u typů se řídí i pojmenování hodnot výčtového typu a pojmenování funkcí. Předpona funkce psaná malými písmeny udává knihovnu, z které pochází; ”gl” označuje základní funkci OpenGL, ”glu” potom funkci z OpenGL Utility Library (GLU), apod.

V OpenGL existují skupiny funkcí, které na venek dělají tutéž činnost a liší se pouze v počtu a typu argumentů. Tyto funkce jsou od sebe odlišeny příponou, která může mít až čtyři znaky. Prvním znakem může být číslice, která indikuje počet parametrů funkce. Druhý znak nebo dvojice znaků udává typ argumentů; uplatňují se přitom přípony uvedené v tabulce. Poslední znak, je-li přítomen je ”v”, a označuje, že se nebudou předávat parametry jednotlivě, ale že argumentem bude ukazatel na pole příslušného rozmětu a typu. Jako příklad lze uvést funkci `glVertex*`, která definuje bod v trojrozměrném prostoru. Funkce existuje v mnoha modifikacích, mimo jiné:

```
void glVertex3f(GLfloat x,GLfloat y, GLfloat z);
      // tři souřadnice ve floatech
void glVertex2sv(GLshort v[2]);
      // dvě souřadnice x,y zadány v poli,
void glVertex4d(GLdouble x, GLdouble y,  GLdouble z, GLdouble w);
      // čtyři souřadnice v double
```

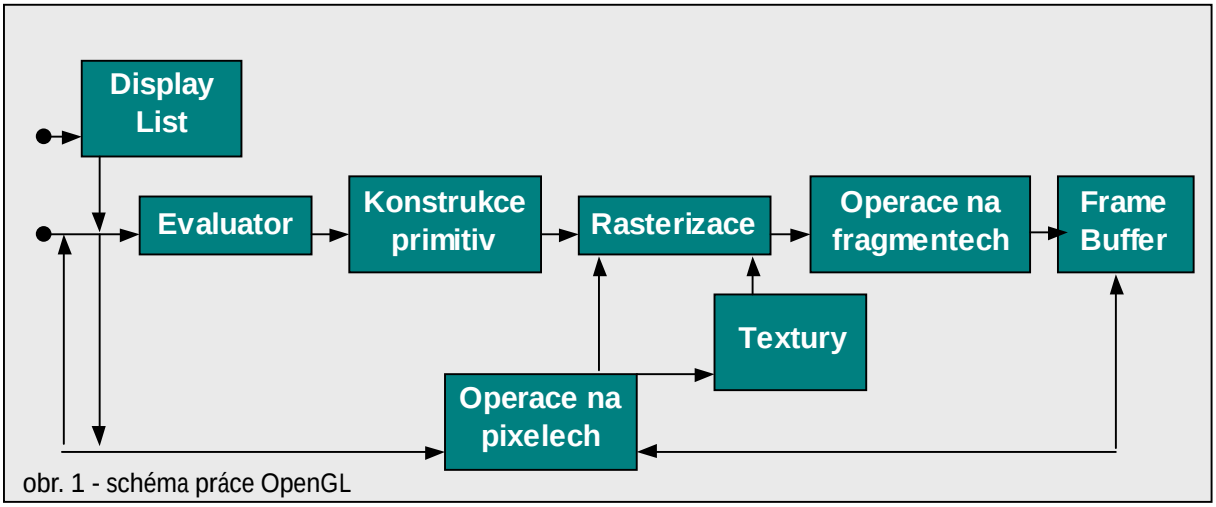
Princip práce OpenGL

Typický program, který využívá OpenGL začíná vytvořením okna, do kterého bude směřovat výstup. Vlastní vytvoření okna se děje voláním služeb API okenního systému a s OpenGL nemá nic společného. Poté se alokuje GL kontext (rendrovací kontext), což je jakýsi soubor stavových proměnných, který je unikátní pro každý grafický výstup. Jakmile se rendrovací kontext přiřadí k oknu, mohou se volat vlastní funkce OpenGL. Mimo rendrovacího kontextu se vytvoří i framebuffer, což je soubor zásobníků, ve kterých se uchovávají informace o scéně. Framebuffer samotný není součástí rendrovacího kontextu, ale rendrovací kontext obsahuje údaje o jeho konfiguraci. Výsledný efekt OpenGL příkazů je v konečné fázi kontrolován okenním systémem, který alokuje framebufferové zdroje. Je to právě okenní systém, který určuje, jak bude framebuffer strukturován, a jak se k němu bude přistupovat. Podobně výsledný obraz, zobrazovací část framebufferu, není adresovaný OpenGL, ale okenním systémem. Přitom se ještě provádí drobné úpravy, jako gamma korekce, apod. Konfigurace framebufferu se děje mimo OpenGL rendrovací kontext v souvislosti s okenním systémem a jeho inicializace probíhá při vytvoření okna.

Spolupráce OpenGL a operačního systému se týká navíc ošetření vstupů. Distribuce knihoven funkcí OpenGL umožňuje odchytávát události systému jako stisk klávesy nebo pohyb myší. Protože jsou tyto akce výrazně systémově závislé, řeší se v přídavných modulech, mimo základní množinu funkcí OpenGL.

OpenGL umožnuje nastavit režim práce pomocí množiny módů. Každý mód se dá měnit nezávisle; nastavení jednoho neovlivňuje nastavení druhého. Zpracování elementárních objektů může probíhat buď v *immediate* (bezprostředním) módu nebo *display list* módu. (pro označení těchto pracovních režimů jsem nenašel žádný ustálený český ekvivalent). Immediate mód spočívá v tom, že všechny akce s primitivy se provádějí přímo v ten okamžik, kdy jsou programem zavolány. Tento přístup je vhodný víceméně akorát pro statické scény, kde se objekty neopakují. V případě, že chceme vytvořit v OpenGL animaci, může být tento přístup neefektivní. V každém snímku, třebaže se od předchozího liší jen minimálně, bychom museli znovu specifikovat všechny objekty na scéně. Z toho důvodu je do OpenGL zařazen i Display list mód, který využívá pomocnou datovou strukturu display list. V té se uchovávají předdefinované objekty a na místě volání se potom vyvolávají pouze odkazy na tyto objekty.

Základní schéma práce OpenGL je na obrázku č. 1. Příkazy OpenGL přichází vlevo. Většina příkazů zůstává pro pozdější použití v Display listu, zbylá část příkazů je efektivně poslána přes linku proudového zpracování. V první, výpočetní fázi, která je zajišťována podpůrnou knihovnou GLU se složitější grafické



obr. 1 - schéma práce OpenGL

objekty jako křivky a polynomiální plochy převádí na jednodušší objekty, s kterými už umí OpenGL přímo zacházet. Druhá fáze zahrnuje zpracovávání bodů, úseček a polygonů, výpočet normálových vektorů a osvětlení. Provádí se také ořezávání na rozměry projekčního okna. Třetí fáze - rasterizace znamená vlastní projekci trojrozměrného obrazu na plochu. Rasterizér vytváří sérii adres do framebufferu s hodnotami, při použití dvojrozměrného popisu čar, úseček a polygonů. Nyní se každému bodu přiřazuje konkrétní barva v závislosti na definované barvě objektů, světelných zdrojích, normálovému vektoru plochy a použitém světelném modelu. Pokud má povrch objektu určenou texturu, ve fázi rasterizace se na objekt mapuje.

Výsledkem práce rasterizéru není ještě hotový obraz, ale množina tzv. fragmentů. Fragmentu odpovídá jeden pixel výsledného obrázku, který nese navíc informace o barvě, hloubce, popřípadě obsahuje i souřadnice textury. Fragmenty lze dále zpracovávat a docílit tak různým efektů. Můžeme například chtít vykreslovat polygony s vyhlazenými hranami *(antialiasing)*, ovlivňovat průhlednost nebo míchat barvy *(blending)*. Patří sem celá množina efektů *(depth cueing)*, které ovlivňují barvu bodu v závislosti na jeho vzdálenosti od pozorovatele. Tak lze jednoduše docílit mlhy a jiných atmosférických vlivů.

Výsledný obraz se uchovává ve framebufferu, kde vlastně proudové zpracování končí. Framebuffer už je přímo pod kontrolou operačního systému. Výsledek se zobrazí v okně. OpenGL ale umožňuje navíc tzv. *feedback mode* (situaci zachycuje obrácená šipka na obrázku č. 1), kdy se obsah framebufferu nevykresluje, ale posílá se zpět programu (klientovi). Tento mód se ukazuje jako velmi výhodný obvzlášť, když výpočet probíhá na několika počítačích najednou.

Definice primitiv

Jak už bylo řečeno, základními stavebními prvky, s nimiž OpenGL pracuje a vytváří scénu jsou body, čáry a polygony. Jednotlivé elementární objekty se v OpenGL definují v sekci uzavřené mezi funkce **glBegin** a **glEnd**. Například trojúhelník s vrcholy (1,0,0), (0,1,0) a (0,0,1) by se zapsal jako:

```
glBegin(GL_POLYGON);
  glVertex3i(1,0,0);
  glVertex3i(0,1,0);
  glVertex3i(0,0,1);
glEnd();
```

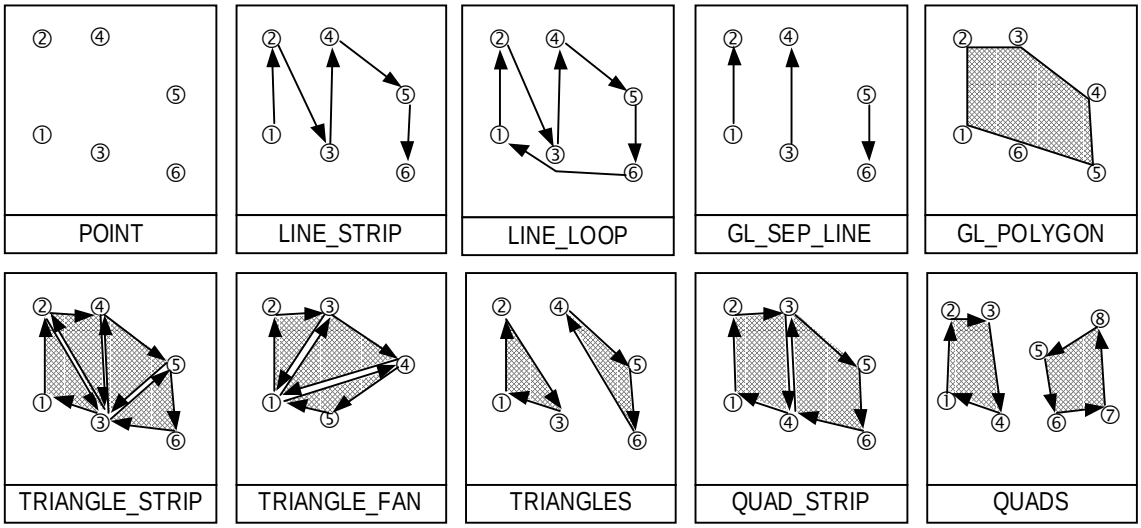
Prvním řádkem dávám najevo, že následující definice vrcholů specifikují vrcholy mnohoúhelníku. Konstanta GL_POLYGON je konkrétní hodnota výčtového typu GLenum. Následují definice vrcholů v prostoru.

Funkcí glVertex() mohu definovat bod v trojrozměrném prostoru udáním dvou, tří nebo čtyř souřadnic. Pokud zavolám glVertex s dvěma souřadnicemi, definuji přímo akorát x a y, souřadnice z se implicitně nastaví na nulu. Pro tři parametry jsou jasně dány všechny souřadnice. Zvláštní případ nastává tehdy, když zadám souřadnice bodu v 3D čtyřmi složkami. V tom případě znamenají jednotlivé složky x, y, z, w takzvané homogenní souřadnice. Homogenní souřadnice jsou výhodnou reprezentací, protože zjednodušují výpočty spojené s různými lineárními transformacemi scény (otočení, posunutí). Bod v n-rozměrném prostoru je zde popsán $n+1$ souřadnicemi. Souřadný systém se nazývá homogenní (stejnorodé), protože λ-násobek ($\lambda \neq 0$) vektoru (x, y, z, w) udává stále stejný bod ve 3D. Mezi kartézskými a homogenními souřadnicemi platí jednoduchá převodní pravidla:

$$(x, y, z) \Rightarrow (x, y, z, 1) \qquad (x, y, z, w) \Rightarrow (x/w, y/w, z/w).$$

Základní výhoda, která plyne z použití homogenních souřadnic, je, že možné realizovat i posunutí vynásobením vektoru maticí. Třebaže vrchol specifikujeme jiným počtem souřadnic, OpenGL si ho převede právě do homogenních, které používá jako vnitřní reprezentaci.

Mimo definice vrcholů může sekce mezi glBegin() a glEnd() obsahovat i volání dalších funkcí přímo spjatých s definicí objektu, jako například nastavení barev, nastavení normálového vektoru nebo specifikace souřadnic textury, volání display listů, atd. Tyto další informace se vážou k jednotlivým vrcholům tělesa a ne k tělesu samotnému. Sekce glBegin/glEnd by neměla mimo tyto funkce obsahovat nic jiného. Toto omezení je dáno tím, že některé implementace OpenGL se snaží optimalizovat elementární objekt pro průchod linkou proudového zpracování. Volání funkcí, které s tím bezprostředně nesouvisí, by mohlo zhoršit efektivitu programu. Naopak volání funkcí glVertex mimo sekci glBegin/glEnd nemá definované chování a může vést i ke kolapsu.



obr. 2 - módy zadávání primitiv

Při definici elementárního objektu funkce glBegin() očekává jako argument typ nově vytvářeného objektu. Podstatné je, jak se mají chápat následující definice vrcholů. Na obrázku č. 2 je znázorněno všech deset možností, jak se dají objekty zadat.

Když definujeme vrcholy mnohoúhelníku nebo lichoběžníků (quads) je třeba zaručit, že nově definovaný objekt bude rovinný. Další problém nastává tehdy, když zadáváme mnohoúhelník, který není konvexní. Takový mnohoúhelník se musí rozdělit na menší konvexní trojúhelníky (tessellation), protože OpenGL přímo pracuje pouze s konvexními. Vlastní rozdělení zajišťují funkce knihovy GLU.

Protože zadávání komplikovaných objektů může být tímto způsobem značně zdlouhavé, je v OpenGL možnost posílat údaje o vrcholech v poli. To se netýká pouze vlastních souřadnic vrcholů, ale i jejich barev, normál, atd. Pro zadávání hodnot přes pole jsou speciálně určeny tyto funkce:

```
GLvoid glEdgeFlagPointer( GLsizei stride, GLvoid *pointer);
    // pole indikátorů obvodových hran
GLvoid glColorPointer(GLint size, Glenum type,
        Glsizei stride, GLvoid *pointer);
    // pole barev
GLvoid glIndexPointer(GLint size, Glenum type,
        Glsizei stride, GLvoid *pointer);
    // pole indexů
GLvoid glNormalPointer(GLint size, Glenum type,
        Glsizei stride, GLvoid *pointer);
    // pole normálových vektorů
GLvoid glVertexPointer(GLint size, Glenum type,
        Glsizei stride, GLvoid *pointer);
    // pole souřadnic vrcholů
```

Tyto funkce mají společný způsob používání. Parametr *size* určuje počet položek pole, *type* udává typ položek pole a nabývá předdefinovaných hodnot výčtového typu. Parametr *stride* určuje prokládání mezi položkami, jejich zarovnání v paměti. Zarovnáním položek na určité adresy lze docílit významného urychlení přístupu do paměti. Jestliže je *stride* nula, položky následují bezprostředně za sebou. Parametr *pointer* je potom ukazatel na začátek pole.

Před vlastním vyvoláváním hodnot z pole je třeba nastavit OpenGL do příslušného pracovního módu příkazem glEnableClientState() s parametrem z hodnot:

GL_EDGE_FLAG_ARRAY, GL_TEXTURE_COORD_ARRAY, GL_COLOR_ARRAY, GL_INDEX_ARRAY, GL_NORMAL_ARRAY a GL_VERTEX_ARRAY.

Při práci s poli se uplatňují následující funkce:

```
void glArrayElement(GLint index);
```
vyvolává kontrétní hodnotu z pole

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```
zajišťuje zpracování celého pole

```
void glDrawRangeElements(GLenum mode, GLuint start, GLuint end,
                         GLsizei count, GLenum type, GLvoid *indices);
```
zpracovává hodnoty pole od indexu *start* do *end*

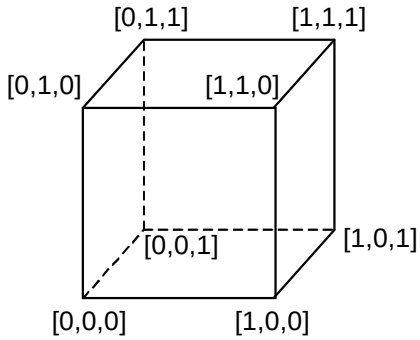
OpenGL umožňuje mimo to práci s prokládanými poli (interleaved arrays), kde prvky mohou být například zároveň souřadnice vrcholů, jejich barvy a souřadnice textury. Práce s prokládanými poli se příliš neliší od práce s jednoduchými poli.

```
void glInterleavedArrays(GLenum format, GLsizei stride,
                         GLvoid *pointer);
```
Funkce má podobný význam jako glDrawArrays; parametr *format* navíc specifikuje obsah pole, strukturu jeho položek, a může nabývat 14 hodnot, kterým odpovídají symbolické konstanty. Například konstanta GL_T2F_C4UB_V3F znamená to, že položka pole obsahuje nejdřív 2 čísla GLfloat, které udávají souřadnice textury, dále následuje specifikace barvy zadané jako čtyři složky typu GLubyte a nakonec jsou třemi GLfloaty udány souřadnice vrcholu.

Jako příklad zadávání primitiv se můžeme podívat na program, který vykreslí jednotkovou krychli, ve dvou provedeních. Poprvé klasickým zadáváním vrcholů, podruhé s využitím pole vrcholů.

```
void display(void)
{
   ...
   glBegin(GL_QUADS);
     glVertex3i(0,0,0);
     glVertex3i(1,0,0);
     glVertex3i(1,1,0);
     glVertex3i(0,1,0);

     glVertex3i(0,0,1);
     glVertex3i(1,0,1);
     glVertex3i(1,1,1);
     glVertex3i(0,1,1);

     glVertex3i(1,0,0);
     glVertex3i(1,0,1);
     glVertex3i(1,1,1);
     glVertex3i(1,1,0);

     glVertex3i(0,0,0);
     glVertex3i(0,0,1);
     glVertex3i(0,1,1);
     glVertex3i(0,1,0);


     glVertex3i(0,1,0);
     glVertex3i(1,1,0);
     glVertex3i(1,1,1);
     glVertex3i(0,1,1);

     glVertex3i(0,0,0);
     glVertex3i(0,1,0);
     glVertex3i(1,0,1);
     glVertex3i(0,0,1);
   glEnd();
   ...
}
```

```
GLint vertices =
 {0,0,0, 1,0,0, 1,1,0, 0,1,0,
  0,0,1, 1,0,1, 1,1,1, 0,1,1,
  1,0,0, 1,0,1, 1,1,1, 1,1,0,
  0,0,0, 0,0,1, 0,1,1, 0,1,0,
  0,1,0, 1,1,0, 1,1,1, 0,1,1,
  0,0,0, 1,0,0, 1,0,1, 0,0,1};

void display(void)
{  ...
   glDrawElements(GL_QUADS, 24,
            GL_INT, vertices);
}
```



obr. 3 - jednotková krychle

Manipulace se stavy

Řízením stavů se přímo ovlivňuje, které funkce OpenGL mají být v dané chvíli aktivní. Ze začátku je většina vlastností vypnuta a programátor je musí na začátku programu inicializovat, pokud je bude využívat. Týká se to například používání textur, odstraňování odvrácených stěn objektů nebo nastavení mlhy. Operace se stavy, které mohou nabývat pouze dvou hodnot, se provádí pomocí dvojice funkcí:

```
void glEnable(GLenum capability);        // zapíná vlastnost
void glDisable(GLenum capability);       // vypíná vlastnost
```

S inicializací vlastností OpenGL jsme se už setkali v minulém odstavci. Funkce glEnableClientState podobně jako glEnable zapíná dílčí vlastnost OpenGL. OpenGL totiž už od verze 1.1 dává možnost vyhodnocování některých stavů už na straně klienta.

Stavy obecně jsou skrytými systémovými proměnnými, které mohou mít různý význam. Stavovými proměnnými jsou i zásobníky framebufferu. Na zjišťování hodnot vnitřních proměnných OpenGL jsou v určeny následující funkce:

```
void glGetBooleanv(GLenum pname, GLboolean *params);
void glGetIntegerv(GLenum pname, GLint *params);
void glGetFloatv(GLenum pname, GLfloat *params);
void glGetDoublev(GLenum pname, GLdouble *params);
void glGetPointerv(GLenum pname, GLvoid *params);
```

Těmito funkcemi se můžeme odvolávat na obsah stavových proměnných jednotlivých typů. Parametr *pname* je symbolická konstanta, která označuje proměnnou, na níž se ptáme.

```
void glClear(GLbitfield mask);
```

Provádí mazání zásobníků. Parametr mask udává, kterých zásobníků se to má týkat. Přípustné jsou hodnoty: GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_ACCUM_BUFFER_BIT, GL_STENCIL_BUFFER_BIT. Funkci glClear() je možné volat s parametrem bitového součtu více hodnot. Mazání zásobníku je poměrně časově náročná operace, ale nové grafické karty umožňují mazání několika zásobníků najednou, proto například volání glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) může být mnohdy výrazně rychlejší než volání glClear jednotlivě.

OpenGL má vlastní ošetření chybových stavů. Kód chyby naposled prováděné operace lze také chápat jako stavovou proměnnou. Její hodnotu vrací funkce:

```
GLenum glGetError(void);
```

GL_NO_ERROR	operace proběhla bez chyby
GL_INVALID_VALUE	číselný argument je mimo rozsah
GL_INVALID_OPERATION	operace není povolena v aktuálním stavu
GL_STACK_OVERFLOW	došlo k přetečení zásobníku
GL_STACK_UNDERFLOW	došlo k podtečení zásobníku
GL_OUT_OF_MEMORY	nedostatek paměti ke spuštění příkazu
GL_TABLE_TOO_LARGE	příliš velká tabulka

Zobrazování plošných primitiv

Při projekci trojrozměrné scény do okna zůstává obraz složen z plošných objektů - bodů, čar a mnohoúhelníků. Jejich zobrazování může být dále určeno skupinou funkcí:

`void` **`glPointSize`**`(GLfloat size);`

Nastaví velikost bodu na obrazovce. Parametr *size* musí být větší než 0.0, implicitní hodnota je 1.0. Pokud je size rovno dvěma, bude jeden bod zobrazen jako čtvereček 2×2 pixely, a podobně.

`void` **`glLineWidth`**`(GLfloat width);`

Nastaví tloušťku čáry; význam parametru je obdobný jako u předchozí funkce. Význam tloušťky čáry se může změnit, pokud máme nastaveno vyhlazování hran - antialiasing volbou glEnable(GL_LINE_SMOOTH). Potom jsou povoleny i neceločíselné hodnoty. Čáry se potom mohou vykreslovat s menší barevnou intenzitou.

`void` **`glLineStripple`**`(GLint factor, GLushort pattern);`

Funkce umožňuje nastavit přerušovanou čáru. Parametr *pattern* určuje vzor, který se periodicky opakuje. Tam kde je v binárním obrazu *pattern* jednička se chápe čára plná, tam, kde je nula je čára přerušená. Argument *factor* udává délku periody.

`void` **`glPolygonStripple`**`(const GLubyte *mask);`

Definuje výplňový vzor pro mnohoúhelník. Argument *mask* představuje ukazatel na bitmapu rozměrů 32×32 bitů, která je uložena v poli 32×4 typu GLubyte. Podobně jako jinde, jednička v dané bitové pozici znamená vykreslení bodu, nula prázné místo.

Operace s mnohoúhelníky jsou podstatně složitější. Mnohoúhelník jako plošný útvar v prostoru má dvě strany - přední a zadní. Vykreslování polygonu se může lišit podle toho, jaká strana je přivrácena k pozorovateli. Toho se dá s výhodou použít u obyčejných těles, která mají svůj vnitřní objem a dá se jednoznačně určit, která strana stěny je vnější a která vnitřní. Významné urychlení výpočtu spočívá v tom, že se mohou při výpočtu odstranit odvrácené stěny těles, ještě před vlastním vykreslováním, protože by stejně nebyly vidět. Odvrácené stěny lze přitom poznat tak, že jejich normálové vektory směřující ven z tělesa svírají s vektorem kamery ostrý úhel.

Implicitně se obě strany mnohoúhelníku vykreslují stejně. Abychom popsaný mechanismus uvedli v život, je třeba zavolat funci glPolygonMode().

`void` **`glPolygonMode`**`(GLenum face, GLenum mode);`

Funkce určuje zobrazovací režim pro mnohoúhelník. Argument může nabývat hodnot GL_FRONT_AND_BACK, GL_FRONT nebo GL_BACK. Parametr *mode* potom určuje, jak se bude mnohoúhelník vykreslovat - GL_POINT (pouze vrcholy), GL_LINE (pouze obvod) nebo GL_FILL (vyplněný).

Je dáno konvencí, že mnohoúhelníky, kde pořadí vrcholů je proti směru hodinových ručiček se považují za přivrácené. Tak se dají vytvářet různá rozumná tělesa, která mají orientovatelný povrch. Výjimku tvoří třeba zvláštní matematické útvary jako Möbiova smyčka nebo Kleinova láhev, kde zobrazení z povrchu tělesa do normálového vektoru není spojité.

`void` **`glFrontFace`**`(GLenum mode);`

Funkce nastavuje, jak jsou popsány přivrácené strany mnohoúhelníků. Základní hodnota je GL_CCW, která odpovídá směru proti hodinovým ručičkám. Možná je také opačná hodnota GL_CW, tedy směr hodinových ručiček.

`void` **`glCullFace`**`(GLenum mode);`

Indikuje, které mnohoúhelníky mohou být odstraněny, dříve než se převedou na souřadnice obrazovky. Parametr *mode* je stejný jako u funkce glPolygonMode. Aby se odstraňování odvrácených stěn stalo aktivní, je ho třeba nejprve zapnout voláním glEnable s parametrem GL_CULL_FACE.

U zcela uzavřených objektů složených z mnohoúhelníků se stálou orientací je situace jasná. Je pouze třeba mít na paměti, že pokud zobrazujeme objekt zevnitř, například místnost, je lepší mít tuto službu neaktivní. Podobně pokud při zapnutém *face-cullingu* pracujeme s tělesem, které je otevřené, často je třeba definovat stejné stěny jednou jako vnitřní a podruhé jako vnější.

Geometrické transformace

Modelovací transformace

Modelovací transformace se používají, když chceme explicitně umístit definovaný objekt v prostoru nebo měnit jeho natočení. Obecněji můžeme souřadnice vrcholů objektu všemožně přetvořit násobením vektorů souřadnic vrcholů maticí. OpenGL pracuje s aktuální maticí, což je stavová proměnná. Na ní se realizují veškeré výpočty tohoto druhu. Transformační příkazy vynásobí aktuální matici danou maticí a výsledek znovu uloží do aktuální matice. Mezi jednotlivými transformacemi je třeba nastavit aktuální matici na jednotkovou funkcí glLoadIdentity(), aby nebyly zobrazení vzájemně ovlivněny.

Postup aplikací geometrických transformací je opačný vůči klasickému postupu v běžném smyslu. Nejprve se provede transformace a teprve potom se teprve objekt vykreslí. Například chceme-li vykreslit krychli se středem v *(x,y,z)* pootočenou podél vlastní osy, musíme provést následující kroky:

```
glMatrixMode(GL_MODELVIEW);    // nastavení aktuální matice
glLoadIdentity();              // jednotková matice
glTranslate(-x,-y,-z);         // posunutí o do počátku
glRotate(sigma,dx,dy,dz);      // otočení
glTranslate(x,y,z);            // posunutí zpět
draw_cube();                   // vykreslení krychle
```

Zobrazovací transformace

Zobrazovací transformace jsou analogií k umísťování a rotaci kamery, kterou získáváme obraz scény. Pozici kamery v prostoru určují její souřadnice a navíc pomocný vektor *up-vector*, který udává lokálně směr nahoru. Implicitně je *up-vector* nastaven na (0,1,0) tedy ve směru osy *y*.

Zobrazovací transformace jsou v přirozeném duálním vztahu k modelovacím transformacím. Pokud je scénu tvoří jediný objekt, potom posunutí objektu dozadu je ekvivalentní s posunutím kamery v opačném směru. V souvislosti se zobrazovacími a modelovacími transformacemi je výhodné hovořit o lokálním a globálním souřadném systému.

Projekční transformace

Konečně projekční transformace se již týkají promítání scény do okna. Nastavování vlastností projekční transformace má svou analogii ve vybírání objektivu ke kameře. To jak se bude scéna mapovat do roviny určuje matice projekčního zobrazení. OpenGL nabízí dva vestavěné nejdůležitější druhy projekce – kolmou a perspektivní projekci.

Při kolmé projekci nedochází ke zkreslení délek a rovnoběžky zůstávají i po aplikaci transformace rovnoběžné. Toto zobrazení však neodpovídá fyzikálnímu modelu. Jako transformační matice je použita jednotková matice. Kolmá (ortogonální) projekce nalézá uplatnění v různých technických aplikacích a CAD modelerech, kde je důležitější přesné zobrazení proporcí než realistický pohled. Oproti tomu perspektivní projekce dává věrné zobrazení. Nastavení parametrů perspektivy se provádí voláním funkce gluPerspective() z GLU. Obecné projekční zobrazení se definuje pomocí funkce glFrustum(), závisí ale na mnoha parametrech a její použití není zrovna intuitivní.

Pro práci s maticemi a modelovací transformace jsou v OpenGL následující funkce:

`void` **`glMatrixMode`**`(GLenum mode);`

Nastavuje, s kterou modelovací projekcí se bude pracovat. Parametr *mode* může nabývat hodnot GL_MODELVIEW, GL_PROJECTION a GL_TEXTURE. Zvolená matice bude dostupná jako aktivní matice. Z toho plyne, že zároveň je možné zpracovávat pouze jednu matici.

`void` **`glLoadIdentity`**`(void);`

Naplní aktuální matici jednotkovou maticí řádu 4×4.

`void` **`glLoadMatrix{fd}`**`(const TYPE *array);`

Do aktuální matice vloží prvky z vektoru délky 16. Argument ale není záměnný s polem definovaným jako *m[4][4]*, protože jsou prvky v matici číslovány po sloupcích, zatímco pole je číslované po řádcích. Na to je třeba si dát pozor, protože to bývá zdrojem těžko odhalitelných chyb v programu.

`void` **`glMultMatrix{fd}(`**`const TYPE *array);`

Vynásobí zprava aktuální matici maticí předanou jako argument a výsledek operace uloží jako aktuální matici.

`void` **`glTranslate{fd}`**`(TYPE x, TYPE y, TYPE z);`

Provede základní modelovací transformaci – posunutí. Z argumentů *x*, *y* a *z* vytvoří matici a tou vynásobí aktuální matici. Výsledek se uloží do aktuální matice.

`void` **`glRotate{fd}`**`(TYPE angle, TYPE x, TYPE y, TYPE z);`

Vynásobí aktuální matici maticí, která podle zadaných argumentů otočí lokální souřadný systém proti směru hodinových ručiček o úhel angle v jednotlivých osách v poměru daném parametry *x*, *y*, *z*. To, že se neotáčí objekt, ale pouze lokální souřadný systém vede k tomu, že objekt, který je vzdálen od počátku výrazně změní svou pozici aplikací transformace, zatímco objekt, který má těžiště v počátku se otočí jen kolem své vlastní osy.

`void` **`glScale{fd}`**`(TYPE x, TYPE y, TYPE z);`

Provádí změnu měřítka vynásobením aktuální matice. Pokud je jednotlivý parametr menší než jedna dojde ke krácení v dané ose, naopak číslo větší než jedna roztáhne lokální systém v daném směru. Jedničkové parametry znamenají identické zobrazení.

Protože není žádný principiální rozdíl mezi modelovou a zobrazovací transformací, podstatné je pořadí provádění rotací a posunutí. Například posloupnost příkazů posunutí, rotace a vykreslení dává jiný výsledek, než rotace, posunutí a vykreslení. Když se komponuje složitá dynamická scéna, je dobré postupovat od lokálních transformací ke globálním. Aby se zjednodušily zobrazovací transformace, je v knihovně GLU pro tyto účely určená funkce:

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
               GLdouble centerx,GLdouble centery,GLdouble centerz,
               GLdouble upx, GLdouble upy, GLdouble upz);
```

Definuje matici a zprava jí výnásobí aktuální maticí. Globální souřadný systém je potom dán souřadnicemi pozorovatele *eyex*, *eyey*, *eyez*, které se stanou novým počátkem. Parametry *centerx*, *centery*, *centerz* udávájí nějaký bod, který má být v novém výhledu uprostřed obrazovky, na které se bude dívat. Konečně *upx*, *upy*, *upz* udávájí *up-vector*, tedy směr, který bude znamenat směr nahoru. Přestože je gluLookAt() pouze složením předešlých modelovacích transformací, dává dojem, že se posunula kamera, ne objekt.

Pro projekční transformace se používá v OpenGL nejčastěji dvojice funkcí glOrtho() a gluPerspective().

```
void gluPerspective(GLdouble fovy,GLdouble aspect,
                    GLdouble near,GLdouble far);
```

Funkcí gluPerspective se nastavují vlastnosti perspektivy. Parametr *fovy* určuje zorný úhel a může být v intervalu <0, 180>, aspect určuje poměr mezi výškou a šířkou po aplikaci projekce. Nakonec parametry *near* a *far* určují ořezávací roviny hloubky. Vrcholy objektů ležící za *far* nebo před *near* budou ořezány – nevykreslí se.

```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,
             GLdouble top, GLdouble near, GLdouble far);
```

Vytvoří matici pro kolmou projekci a vynásobí s ní aktuální matici. Parametry *left*, *right*, *bottom*, *top*, *near*, *far* udávájí ořezávací roviny. Modelovaná scéna se tak redukuje na kvádr daných rozměrů.

```
void glViewport(GLint x, GLint y, GLsizei w, GLsizei h);
```

Funkce glViewPort je užitečná v případě, když potřebujeme ve svém programu explicitně zadat polohu a rozměry obdélníku na obrazovce, kam bude směřovat výstup. Pokud nic neuděláme, OpenGL nastaví svůj výstup na velikost okna. Implicitní velikost je tedy *[0, 0, winWidth, winHeight]*.

Barvy a světlo

Nastal čas krátce se zmínit o tom, jak OpenGL pracuje s barvami. OpenGL používá pro výpočty reprezentaci barev v RGBA. RGB jsou klasické barevné složky červená – zelená – modrá. Písmeno A označuje tzv. alfa kanál, který navíc určuje průhlednost barvy.

Dnes už snad téměř všechny grafické karty pracují v módu TrueColor nebo HighColor, kde není omezení na počet současně zobrazených barev. Protože ne vždy tomu tak bylo a z důvodu zajištění kompatibility umí OpenGL pracovat i v *color-index-mode*, kde se používají barevné palety. Použití indexovaných barev s OpenGL znamená komplikovanější spolupráci s operačním systémem, který spravuje paletu. Je také pomalejší, protože se musí provádět další přepočty.

OpenGL podporuje dithering, což je metoda, kterou lze dosáhnout uspokojivých výsledků i při použití indexovaných barev. Problém malé barevné hloubky se tak převede na menší prostorové rozlišení. Míchání barev se potom dociluje soutiskem různě barevných pixelů vedle sebe, což dodává na první pohled dojem nových odstínů. Dnes je používání indexovaných barev spíše historickou záležitostí.

Pro definování aktuální barvy je v OpenGL funkce glColor*() v několika verzích:

```
void glColor3{b s f d ub us ui}(TYPE r, TYPE g, TYPE b);
void glColor3{b s f d ub us ui}v(const TYPE *v);
void glColor4{b s f d ub us ui}(TYPE r, TYPE g, TYPE b, TYPE a);
void glColor4{b s f d ub us ui}v(const TYPE *v);
```

Funkce nastavuje RGB, popřípadě i alfa kanál pro aktuální barvu. Parametr může být navíc předán normálně nebo v poli. Pokud určím jenom tři parametry, alfa kanál se automaticky nastaví na hodnotu 1.0, která značí zcela krycí barvu.

Funkce glColor*() se týká výhradně barvy objektu. Pokud chceme specifikovat barvu pozadí, musíme k tomu použít funkci glClearColor(). Vlastní vymazání obrazovky se provede příkazem glClear() s argumentem GL_COLOR_BUFFER_BIT.

Pro zobrazování elementárních plošných objektů je podstatný zvolený stínovací model (shade model). Stínování objektu v tomto smyslu znamená změnu barvy nebo barevného jasu jeho stěn v závislosti na světelném zdroji. Nejedná se tedy o výpočet stínu vrženého tělesem. Nejjednodušší je ploché stínování (flat shading), kdy je stěna vykreslena pouze jednou barvou, bez vztahu ke světlům definovaným ve scéně. Lepší výsledky dává Goraudovo stínování (smooth shading). Při použití Goraudova stínování se pro každý vrchol elementárního objektu použije normálový vektor, jeho nastavená barva a provede se výpočet, který definitivně určí barvu s jakou se vy kreslí na obrazovce. Na hodnotách takto spočtených vrcholů mnohoúhelníku se provede bilineární interpolace podle plošné polohy. Vnitřním bodům polygonu se tak přiřadí barva smíchaná z barev vrcholů v poměru jejich vzdáleností od jednotlivých vrcholů. Výsledkem toho všeho jsou hladké barevné přechody, které dodávají lepší vizuální dojem (viz. obr. 4). Stínovací model se nastavuje funkcí glShadeModel().

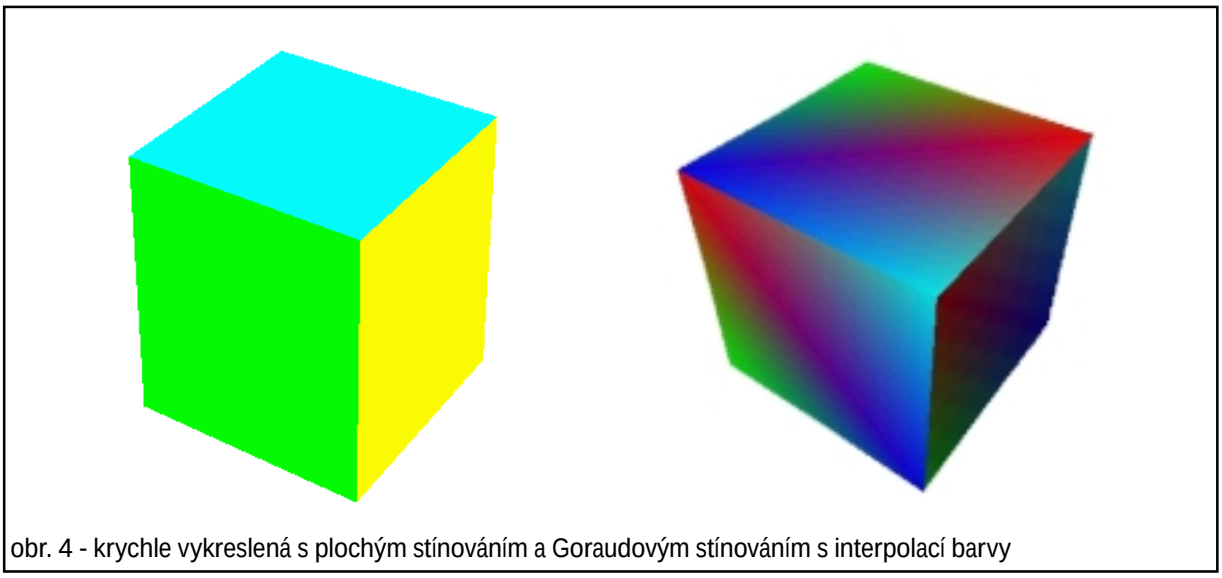
```
void glShadeModel(GLenum mode);
```

Funkce očekává jako argument hodnotu GL_SMOOTH nebo GL_FLAT.

OpenGL pracuje se světlem také v RGB složkách. Vlastnosti materiálu povrchu těles jsou potom dány shopností absorbovat jednotlivé světelné složky. To co se ze světla nepohltilo je odraženo. Rovnice šíření světla v OpenGL ne zcela odpovídají realitě, ale výpočty s nimi spjaté se dají realizovat velice rychle. Je třeba rozlišovat několik druhů světel, s kterými OpenGL pracuje.

Ambientní složka (ambient light)

Ambient je základní nesměrové osvětlení, které je ve scéně přítomné i tehdy, když není definován žádný zdroj světla. V realitě mu odpovídá to množství světla, které vzniká několikanásobným odrazem na stěnách těles. Protože nějaké přímé analytické výpočty byly nesmírně komplikované, OpenGL vystačí s tím, že ve scéně jednoduše předpokládá základní úroveň osvětlení. Ambientní světlo bývá asi 10 - 20% intenzity světelného zdroje.



obr. 4 - krychle vykreslená s plochým stínováním a Goraudovým stínováním s interpolací barvy

Rozptýlená složka (diffuse light)

Difusní složka světla přichází z konkrétního zdroje. Objekty blíž zdroji jsou osvětleny více a intenzita světla klesá se čtvercem vzdálenosti. Když se srazí se stěnou tělesa, rozptýlí se do všech směrů.

Směrová složka (specular light)

Přímé bodové osvětlení přichází z jednoho zdroje, podobně jako rozptýlené světlo. Na povrchu tělesa se odráží pod příslušným úhlem. Ideální zrcadlo odráží ze 100% právě směrové světlo díky dokonalé reflexi.

Pro výpočet odrazu světla na objektu je důležité znát normálové vektory jeho stěn. Informace o normálovém vektoru se sdružuje s jednotlivými vrcholy mnohoúhelníků. Pro přidávání světel do scény zaveden následující postup:

1. Definovat normálové vektory pro každý vrchol všech objektů.
2. Vytvořit, vybrat a umístit světelné zdroje
3. Vytvořit a zvolit světelný model, který definuje úroveň abientní složky a polohu pozorovatele (pro účely výpočtu osvětlení)
4. Definovat vlastnosti materiálu pro objekty ve scéně.

Definice normálových vektorů

Pokud ve svém programu použijeme knihovní funkce GLU, které vykreslují předdefinované objekty, jako krychli, kouli, válec nebo prstenec, jsou už většinou normálové vektory definovány. U nově vytvářených objektů je třeba tyto informace dodat.

S OpenGL je možné nastavit normálový vektor ke každému vrcholu mnohoúhelníku. Nejjednodušší případ je, když vrcholy polygonu mají stejný normálový vektor. Pokud ale určíme normálový vektor ke každému vrcholu zvlášť, můžeme dodat plošce optické vlastnosti zakřiveného povrchu. Vnitřním bodům polygonů se při rasterizaci přiřadí příslušné mezihodnoty. K nastavení aktuálního normálového vektoru je v OpenGL zavedena následující funkce glNormal* s těmito funkčními prototypy:

```
void glNormal3{bsidf}(TYPE nx, TYPE ny, TYPE nz);
void glNormal4{bsidf}v(const TYPE *vector);
```

U normálového vektoru je podstatný pouze jeho směr, na velikosti nezáleží. Proto je možné si zjednodušit jeho výpočet pomocí různých triků, namísto zdlouhavého výpočtu. S normálovým vektorem se často pracuje v tzv. normalizovaném tvaru, kdy je jeho délka rovna jedné. Normálové vektory zůstávají v normalizovaném tvaru i po aplikaci některých modelovacích transformací, třeba posunutí a otočení. Pokud se ale provede jiná transformace, jako změna měřítka nebo zkosení, je třeba normálové vektory znovu normalizovat. OpenGL může provádět automatickou normalizaci normálových vektorů po aplikaci modelovací transformace. Tato služba se zapíná voláním glEnable s parametrem GL_NORMALIZE.

Vytvoření a umístění světelných zdrojů

U světel můžeme nastavovat velké množství vlastností a parametrů. Pro definici světelného zdroje se používá funkce glLight*():

```
void glLight{if}(GLenum light, GLenum pname, TYPE param);
void glLight{if}v(GLenum light, GLenum pname, TYPE *param);
```

Funkce vytváří světelný zdroj, který se pojmenuje symbolickou konstantou uvedenou jako parametr *light*. Na světelné zdroje se potom odvolává pomocí výrazů výrazů GL_LIGHT0, GL_LIGHT1, ... , GL_LIGHT7. Parametr *pname* určuje, kterou vlastnost světla budeme nastavovat, vlastní hodnoty nese parametr *params*.

Definice světelného zdroje potom může vypadat v programu třeba takto:

```
GLfloat ambientni={0.0, 0.0, 0.0, 1.0};
GLfloat rozptylene={1.0, 1.0, 1.0, 1.0};
GLfloat smerove={1.0, 1.0, 1.0, 0.0};
GLfloat pozice={1.0, 1.0, 1.0, 0.0};

glLightfv(GL_LIGHT0, GL_AMBIENT, ambientni);
glLightfv(GL_LIGHT0, GL_DIFFUSE, rozptylene);
glLightfv(GL_LIGHT0, GL_SPECULAR, smerove);
glLightfv(GL_LIGHT0, GL_POSITION, pozice);
```

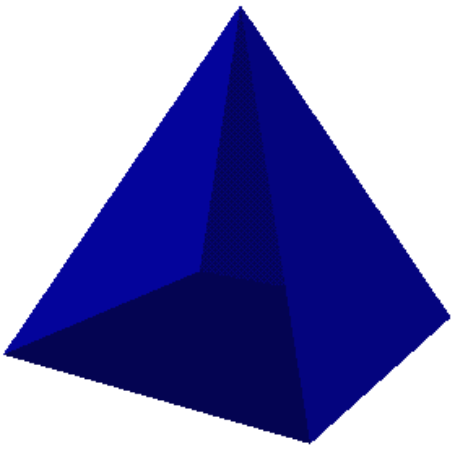
Význam parametru *param* v předchozí funkci může být různý, podle toho co se nastavuje. Pro barevné složky udává *param* hodnotu v RGBA, při zadávání pozice znamená souřadnice *(x, y, z, w)*, přičemž, je-li *w* nenulové, chápe se vektor jako homogenní souřadnice, pokud je nula, berou se hodnoty *x*, *y*, *z* jako souřadnice v trojrozměrném prostoru.

Před použitím světel musíme nejprve tuto vlastnost OpenGL uvést do chodu příkazem glEnable() s argumentem GL_LIGHTING. Konkrétní světelný zdroj je také třeba zapnout prostřednictvím volání glEnable s příslušnou symbolickou konstantou. Komplexní světelný model zahrnuje nastavení parametrů a určuje chování OpenGL ve vztahu k zobrazování osvětlené scény.

```
void glLightModel{if}(GLenum pname, TYPE param);
void glLightModel{if}v(GLenum pname, TYPE *param);
```

Nastavuje vlastnosti světelného modelu. Způsob použití funkce je v mnohém podobný s glLight*(). Parametr *pname*, určuje vlastnost, která má být měněna. Konkrétní hodnota je potom v *param*. Varianta funkce, která předává parametr hodnotou (ne přes ukazatel), může být použita pouze pro nastavení jednohodnotových položek, viz. dále.

Parametr *pname* může být buď GL_LIGHT_MODEL_AMBIENT, *param* potom znamená intenzitu ambientní složky světla v celé scéně v RGBA. Implicitní hodnota je 20% intenzita. Pokud má *pname* hodnotu GL_LIGHT_ MODEL_LOCAL_VIEWER, určuje se způsob, jak se bude vyhodnocovat směrová složka světla. Podle vzdálenosti objektu od pozorovatele může OpenGL provádět jednodušší výpočty odrazů. Tato možnost je zpočátku vypnutá. Přípustné hodnoty *param* jsou v tomto případě GL_TRUE / GL_FALSE. Konečně poslední vlastnost, která se funkcí glLightModel nastavuje, udává, má-li se rozdílně vykreslovat přední a zadní strany mnohoúhelníků, jak už bylo uvedeno. Příslušná hodnota pro *pname* je GL_LIGHT_MODEL_TWO_SIDE. Parametr *param* potom může nabývat hodnot GL_FALSE / GL_TRUE.



obr. 5 - jehlan s průhlednou barvou (kanál alfa na 50%)

Se zobrazovacími prostředky OpenGL lze provádět všemožná kouzla. Nastavení vlastností materiálu povrchu těles do značné míry přiblíží konečný obraz realitě. Některé objekty ve scéně navíc mohou vyzařovat světlo, dají se i zobrazit vržené stíny. Pokud se k tělesům na scéně ještě přiřadí textura, je výsledek skoro dokonalý. Tyto efekty už ovšem patří k pokročilejším praktikám při práci s OpenGL. Osvětlení v OpenGL je rozsáhlá kapitola, která přesahuje rámec tohoto textu. Proto se zatím spokojíme s tímto stručným nástinem vlastností.

Na co nezbyl čas

OpenGL nabízí široké spektrum služeb pro práci s trojrozměrnou grafikou. Proto je třeba chápat uvedený popis pouze jako úvod do problematiky s jistým zjednodušujícím pohledem. Z vlastností, které nebyly popsány, je určitě významné použití textur. OpenGL umožnuje práci s jendnorozměrnými, dvojrozměrnými i trojrozměrnými texturami. Základní, dvojrozměrné textury, jsou ve své podstatě bitmapy, které se promítají na povrch tělesa. Trojrozměrné textury naproti tomu zachycují přímo charakter materiálu a jakoby prostupují celým tělesem. Dá se tak napodobit dřevo nebo mramor. Jednorozměrné textury nejsou příliš obvyklé, odpovídají víceméně dvojrozměrným texturám, které mají výšku jeden pixel.

Mapování dvojrozměrných textur na objekty probíhá tak, že se každému vrcholu přiřadí souřadnice textury. Ty potom slouží jako kontrolní body; ve vnitřku mnohoúhelníků se potom nanese barva daná hodnotou uloženou v matici textury na souřadnicích mezi těmito kontrolními body. OpenGL podporuje i *mipmaping*, což je výhodný způsob práce s texturami, který umožňuje vykreslovat texturované objekty s různou úrovní detailu textury v závislosti na vzdálenosti od pozorovatele. Textura potom musí být pořízena v několika různých rozlišeních. Její rozměry textur by měly být mocniny 2. Když se má potom vykreslovat otexturovaný polygon, který má na obrazovce zabírat určitou plochu, OpenGL automaticky vybere dva nejbližší vzorky, které se jí velikostí blíží a vytvoří z nich povrch.

Doposud jsme v OpenGL pracovali v *immediate* módu, hlavní význam ale má použití *display listů*. Jak už bylo řečeno, *display listy* umožňují pohodlné uchování definovaných těles v paměti, na která se lze jednoduše odkazovat. Pomocí display listů se snadněji vytváří dynamická scéna s velkým počtem objektů.

Pokud chceme uvést scénu v OpenGL do pohybu, neznamená to žádné principiální komplikace. Pracuje se namísto s jedním bufferem, se dvěma. Zatímco je předchozí snímek na obrazovce, probíhají výpočty následujícího, v pravý okamžik se potom buffery prohodí a tak se zobrazí nový snímek. Toto dvojité bufferování je nezbytné, protože jinak by bylo vidět samotné překreslování, což by působilo rušivě. Největší rychlost překreslování, které lze dosáhnout je přitom 60 fps. Rychlost zobrazování kolísá podle míry komplexnosti scény po skocích 60/1, 60/2, 60/3, … To znamená, že mnohdy stačí jen o málo více zobrazovaných objektů a *frame-rate* dramaticky klesne. Úkolem programátora je potom využít hladinu maximálního výkonu.

Lze se setkat i s tím, že některé aplikace využívají OpenGL výhradně pro dvourozměrnou grafiku. Používání OpenGL jako základní grafické knihovny pro elementární kreslení však nelze příliš doporučit. Důvod je ten, že OpenGL nemá možnost přepínat mezi práci ve dvou a ve třech rozměrech. Kreslíme-li plošný útvar, automaticky se k němu přiřadí i nulová *z*-souřadnice. S objektem se dále nakládá jako s prostorovým. Podobné vedlejší účinky příkazů mohou mít následně negativní vliv na efektivitu programu.

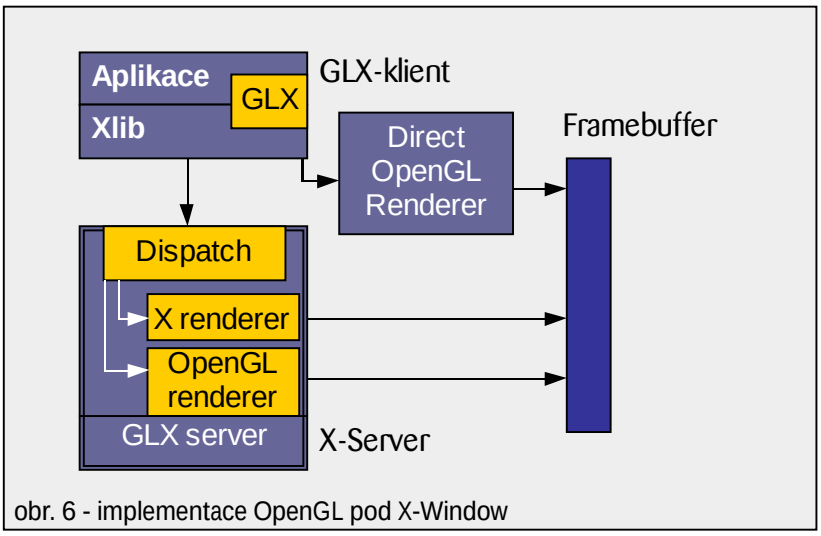
OpenGL pod Unixem a pod Windows

Zatímco za většinou implementací OpenGL nejsou vidět konkrétní lidé, portaci OpenGL do prostředí X-Window má z velké části na svědomí jediný člověk, Kurt Akeley. Dá se říci, že právě implementace OpenGL pod Unixem je nejvydařenější. Na Unixu byla nejdál dotažena koncepce klient/server.

X-Window řídí jednak procedurální interface tak i síťový protokol pro vytváření a manipulaci s framebufferovými okny. Do těchto oken vykresluje dvojrozměrné objekty. Integrace OpenGL do systému X-Window se realizuje rozšířením GLX. GLX využívá specifický síťový protokol pro rendrovací příkazy OpenGL a tento protokol je zapouzdřen v komunikačním protokolu X. GLX nabízí už poměrně komfortní služby jako například podporu PostScriptových fontů v OpenGL, apod.

OpenGL vyžaduje úsek framebufferu do kterého mají být rendrovány elementární objekty. V terminologii X-Window se tento úsek nazývá *drawable*. Aby mohl programátor používat OpenGL, společně s *drawable* vytvoří i OpenGL kontext. Jakmile je kontext, inicializuje se kopie OpenGL rendereru s informací o aktuálním výstupu. OpenGL renderer patří k GLX rozšířením X-Window. Protože je renderer koncepčně součástí X-Window, může se na něj odkazovat X-klient a úseky framebufferu mohou být sdílené. Mimo to je dostupný

režim přímého rendrování, který umožňuje GLX-klientovi vyšší grafický výkon. Tímto se obchází volání služeb X-Serveru, množství funkcí je však nepřístupných. Situaci přehledně zachycuje obr. 6.



obr. 6 - implementace OpenGL pod X-Window

Oproti tomu je situace u Windows podstatně jednoduší. Implementace OpenGL pod Windows pochází od samotného Microsoftu. Napojení OpenGL na Windows zajišťuje knihovní rozšíření Windows API - WGL. Pokud má aplikace vyžívat OpenGL, je třeba nejprve vytvořit okno, které OpenGL podporuje (DC). K tomu je třeba specifikovat formát pixelu (pixel format descriptor - PFD), tedy dohodnout se s operačním systémem, v jakém tvaru má očekávat grafický výstup. K nově vytvořenému oknu je třeba alokovat OpenGL kontext (renderovací kontext). Funkce Windows API pracují s kontextem zařízení (Device Context), který určuje zařízení, do nějž bude směřovat výstup volané funkce GDI. Je přitom třeba rozlišovat mezi kontextem zařízení (DC) a rendrovacím kontextem OpenGL (RC). OpenGL rendrovací kontext je soubor stavových proměnných, který je unikátní pro daný výstup. Pro práci s rendrovacím kontextem jsou ve Windows dostupné následující funkce:

```
wglCreateContext();        // vytvoří nový RC
wglMakeCurrent();          // nastaví nový RC jako aktuální
wglGetCurrentContext();    // vrací aktuální RC
wglGetCurrentDC();         // vrací DC asociované s aktuálním RC
```

Další fází je určení Pixel Format Deskriptoru (PFD). PFD je vrstva mezi OpenGL rutinami pro výstup a operacemi výstupu podporovanými Windows. PFD popisuje vlastnosti grafického výstupu - způsob zobrazování barev, barevnou hloubku, počet bitů Z-bufferu a další shopnosti daného zařízení. Ve Windows API mu odpovídá struktura PIXELFORMATDESCRIPTOR společně s několika podpůrnými funkcemi pro manipulaci s touto strukturou:

```
ChoosePixelFormat();       // vrací PFD, který nejvíce vyhovuje
SetPixelFormat();          // nastaví PFD danému DC
GetPixelFormat();          // vrací index akt. PFD v daném DC
DescribePixelFormat();     // vrací informace o PFD v daném DC
```

Implementace OpenGL pod Windows 95/NT podporuje 24 různých PFD, které jsou uspořádány podle svých vlastností. Hlavním kritériem je barevná hloubka (4, 8, 16, 24 a 32 bitů). Osm typů PFD je definováno podle barevné hloubky dané ovladačem zařízení. Těmto formátům se říká nativní formáty, protože jsou podporovány přímo hardware. Nenativní formáty jsou potom odvozeny pro ostatní barevné hloubky.

Univerzální postup je tedy asi následující. Jako první se nastaví PFD, na jeho základě se následně připraví okno, které by umožňovalo výstup z OpenGL (RC). Ve třetím kroku se právě vytvořený RC zvolí jako aktuální. Tímto jsme si zpřístupnili všechny funkce OpenGL. Při ukončování programu nastavíme RC jako neaktuální a nakonec ho zrušíme.

Se získanými vědomostmi bychom už měli být schopni napsat jeden kompletní program. Následující ukázka představuje jednoduchý program na vykreslení barevné krychle. S krychlí se dá myší rotovat a při stisknutém tlačítku myši i posouvat dopředu a dozadu. Program také demostruje některé vlastnosti OpenGL, které lze v průběhu zapínat a vypínat stiskem klávesy- Face Culling *(c,C)* a Depth Test *(z,Z)* popřípadě se dají prohodit osy otáčení *(o,O)*.

```
#include <windows.h>
#include <gl/glut.h>
#include <math.h>
#include <stdio.h>

#define PI 3.1415926535

GLfloat tz = -40.0;
GLfloat ry = 0.0;
GLfloat rx = 0.0;
GLboolean reverse= TRUE;                // nastavuje
GLboolean ZBUFF = TRUE;                 // Z-buffer
GLboolean CULL   = TRUE;                // a Face Culling
static int beforex, beforey;            // proměnné pro hladky pohyb

void doCube(void);                      // deklarace funkcí
void checkDepth(void);
void checkCulling(void);

////////////////////////////////////////////////////////////////////////

  static GLint v[24]=                   // pole vrcholů krychle
  {
    -5,  5,  5,  // 0      5,  5,  5,  // 1
     5, -5,  5,  // 2     -5, -5,  5,  // 3
    -5, -5, -5,  // 4     -5,  5, -5,  // 5
     5,  5, -5,  // 6      5, -5, -5,  // 7
  };
  static GLfloat colors[6][3]=          // pole barev stěn
  {
    {1.0, 0.0, 0.0},      {0.0, 1.0, 0.0},
    {0.0, 0.0, 1.0},      {0.0, 1.0, 1.0},
    {1.0, 0.0, 1.0},      {1.0, 1.0, 0.0}
  };
  static GLuint indices[24] =
  {
    0, 1, 2, 3, /*přední*/    1, 6, 7, 2,  // levá
    5, 0, 3, 4, /*pravá*/     5, 6, 1, 0,  // horní
    3, 2, 7, 4, /*zadní*/     6, 5, 4, 7   // dolní
  };

void doCube(void)                       // vykreslení krychle
{
  int count = 0;
  int col = 0;
  glBegin(GL_QUADS);                // začíná zadávání čtyřúhleníků
  for (count = 0; count < 24; count++)
  {
    col = (int)floor(count/4);
    glColor3fv (colors[col]);           // nastavení barvy stěně
    glArrayElement (indices[count]);    // vyvolání prvku z pole
  }
  glEnd();
}
```

```
void init(void)                          // inicializační procedura
{
  glShadeModel(GL_SMOOTH);               // Goraudovo stínování
  glClearColor(0.0, 0.0, 0.0, 1.0);      // výběr mazací barvy
  glEnableClientState(GL_VERTEX_ARRAY);// souřadnice vrcholů v poli
  glVertexPointer(3, GL_INT, 0, v);      // nastavení ukazatele na pole
}                                        // vrcholů

void checkDepth(void)                    // nastaví Z-bufffer do chodu
podle
{                                        // hodnoty proměnné ZBUFF
  if (ZBUFF) glEnable(GL_DEPTH_TEST);
  else glDisable(GL_DEPTH_TEST);
  glutPostRedisplay();                   // překreslení
}

void checkCulling(void)                  // nastaví funkci odstraňování
{                                        // odvrácených stěn, podle
  if (CULL)                              // hodnoty proměnné CULL
  {
    glEnable(GL_CULL_FACE);
    glCullFace(GL_FRONT);
    glFrontFace(GL_CCW);
  }
  else glDisable(GL_CULL_FACE);

  glutPostRedisplay();                   // překreslení
}

void rotate (int rotX, int rotY)         // provádí rotaci scény
{
  if (reverse)                           // podle parametru reverse
  {                                      // provede nejprve rotaci podle
    glRotatef(rotX, 1.0, 0.0, 0.0);      // x a potom podle y
    glRotatef(rotY, 0.0, 1.0, 0.0);      // nebo naopak
  }
  else
  {
    glRotatef(rotY, 0.0, 1.0, 0.0);
    glRotatef(rotX, 1.0, 0.0, 0.0);
  }
  glutPostRedisplay();                   // překreslení scény
}

void display(void)
{                                        //mazání dvou zásobníků zároveň
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();                      // jednotková matice

  checkDepth();                          // vlastnosti Z-Bufferingu
  checkCulling();                        // a Face-Cullingu
  glColor3f(1.0, 1.0, 1.0);              // aktuální barva bílá
  glPushMatrix();                        // uložení aktuální matice
    glTranslatef(0.0, 0.0, trans_z);     // posunutí dozadu o tz
```

```
    rotate (rx, ry);                    // rotace
    doCube();                           // vykreslení krychle
  glPopMatrix();                        // obnovení aktuální matice
  glutSwapBuffers();                    // prohození zásobníků,
}

void reshape(int width, int height)     // funkce volaná při změně
{                                       // velikosti okna
  GLfloat aspect = width/height;
  GLfloat left;
  GLfloat right;
  GLfloat top;
  GLfloat bottom;
  glViewport(0, 0, width, height);      // určení velikosti okna
  glMatrixMode(GL_PROJECTION);          // nastavení projekční matice
  glLoadIdentity();                     // naplnění jednotkovou maticí
  right = (width/100)/2;                // nastavení proporcí
  left = -right;
  top = (height/100)/2;
  bottom = -top;
  glFrustum(left, right, bottom, top, 10.0, 100.0); // projekce
  glMatrixMode(GL_MODELVIEW);           // aktuální matice je modelovací
  glLoadIdentity();                     // jednotková matice
}

void motion(int x, int y)               // funkce pro pohyb krychle
{                                       // dopředu a dozadu
  tz -= (beforey - y);
  beforey = y;
  if (tz <= -99.0) tz = -99.0;          // ohlídání mezí pro pohyb
  else if (tz >= -10.0) tz = -10.0;
  glutPostRedisplay();                  // překreslení
}

void rotation(int x, int y)             // funkce pro rotaci krychle
{
  rot_x += (beforey - y);
  beforey = y;                          // ohlídání mezí
  if (rx >= 360)rx -= 360.0;
  ry += (beforex - x);
  beforex = x;
  if (ry >= 360) ry -= 360.0;
  glutPostRedisplay();                  // překreslení
}

void mouse(int button, int state, int x, int y)
{                                       // obsluha myších událostí
  beforex = x;                          // aktualizace pomocných prom.
  beforey = y;
  switch(button)
  {
  case GLUT_LEFT_BUTTON:            // když je zmáčknuté levé tlačítko,
    if (state == GLUT_DOWN)         // proveď pohyb
      glutMotionFunc(motion);       // odkaz na funkci, která pohybuje
    if (state == GLUT_UP)           // když není stisklé tlačítko, rotuj
```

Předchozí program používal pro vytvoření okna funkci knihovny GLUT, která mnoho operací dělá za nás. Následující program je ukázkou popsaných postupů pro inicializaci OpenGL pod Windows přímo voláním služeb WIN API a základní množiny funkcí OpenGL. V programu by měly být ošetřeny všechny problematické situace, které mohou nastat.

```
#include <windows.h>
#include <GL/gl.h>            /* základní množina funkcí OpenGL */
#include <stdio.h>

HDC hDC;                      /* kontext zařízení - DC */
HPALETTE hPalette = 0;        /* paleta (pokud je vyžadována) */

HWND CreateOpenGLWindow(char* title, int x, int y,
                        int width, int height, BYTE type, DWORD flags)
{   int         n, pf;
    HWND        hWnd;
    WNDCLASS    wc;
    LOGPALETTE* lpPal;
    PIXELFORMATDESCRIPTOR pfd;
    static HINSTANCE hInstance = 0;  /* registrace nové třídy okna */
    if (!hInstance) {
      hInstance = GetModuleHandle(NULL);
      wc.style         = CS_OWNDC;
      wc.lpfnWndProc   = (WNDPROC)WindowProc;
      wc.cbClsExtra    = 0;
      wc.cbWndExtra    = 0;
      wc.hInstance     = hInstance;
      wc.hIcon         = LoadIcon(NULL, IDI_WINLOGO);
      wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
      wc.hbrBackground = NULL;
      wc.lpszMenuName  = NULL;
      wc.lpszClassName = "OpenGL";

      if (!RegisterClass(&wc)) {
        MessageBox(NULL, "nelze zaregistrovat třídu okna",
                   "Error", MB_OK);
      return NULL;
      }
    }                                  // vytvoření speciálního okna
    hWnd = CreateWindow("OpenGL", title, WS_OVERLAPPEDWINDOW |
        WS_CLIPSIBLINGS | WS_CLIPCHILDREN,
        x, y, width, height, NULL, NULL, hInstance, NULL);
    if (hWnd == NULL) {
      MessageBox(NULL, "nelze vytvořit okno", "Error", MB_OK);
      return NULL;
    }
    hDC = GetDC(hWnd);

    memset(&pfd, 0, sizeof(pfd));          // naplnění položek
    pfd.nSize        = sizeof(pfd);        // struktury PFD
    pfd.nVersion     = 1;
    pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | flags;
    pfd.iPixelType   = type;
    pfd.cColorBits   = 32;
```

```
    pf = ChoosePixelFormat(hDC, &pfd);
    if (pf == 0) {                          // formát nevyhovuje
  MessageBox(NULL, "Nelze nalézt odpovídající PFD", "Error", MB_OK);
  return 0;
    }
     if (SetPixelFormat(hDC, pf, &pfd) == FALSE) {
  MessageBox(NULL, "Chyba při změně PFD", "Error", MB_OK);
  return 0;
    }
    DescribePixelFormat(hDC, pf, sizeof(PIXELFORMATDESCRIPTOR), &pfd);
        // získání informací o PFD do struktury pfd
    if (pfd.dwFlags & PFD_NEED_PALETTE ||      // používání palety
        pfd.iPixelType == PFD_TYPE_COLORINDEX) {

      n = 1 << pfd.cColorBits;// zjištění počtu dostupných barev
  if (n > 256) n = 256;
  lpPal = (LOGPALETTE*)malloc(sizeof(LOGPALETTE) +
                              sizeof(PALETTEENTRY) * n);
  memset(lpPal, 0, sizeof(LOGPALETTE) + sizeof(PALETTEENTRY) * n);
  lpPal->palVersion = 0x300;
  lpPal->palNumEntries = n;
  GetSystemPaletteEntries(hDC, 0, n, &lpPal->palPalEntry[0]);

// pokud je formát pixelu RGBA, naplníme paletu po složkách
if (pfd.iPixelType == PFD_TYPE_RGBA) {
      int redMask = (1 << pfd.cRedBits) - 1;
      int greenMask = (1 << pfd.cGreenBits) - 1;
      int blueMask = (1 << pfd.cBlueBits) - 1;
      int i;

      for (i = 0; i < n; ++i) {        // naplnění barevné palety
        lpPal->palPalEntry[i].peRed =
                         (((i>>pfd.cRedShift)&redMask) * 255);
        lpPal->palPalEntry[i].peGreen =
                         (((i>>pfd.cGreenShift)&greenMask)*255);
        lpPal->palPalEntry[i].peBlue =
                     (((i >> pfd.cBlueShift)  & blueMask)  * 255);
        lpPal->palPalEntry[i].peFlags = 0;
      }
  } else {
      lpPal->palPalEntry[0].peRed = 0;
      lpPal->palPalEntry[0].peGreen = 0;
      lpPal->palPalEntry[0].peBlue = 0;
      lpPal->palPalEntry[0].peFlags = PC_NOCOLLAPSE;
      lpPal->palPalEntry[1].peRed = 255;
      lpPal->palPalEntry[1].peGreen = 0;
      lpPal->palPalEntry[1].peBlue = 0;
      lpPal->palPalEntry[1].peFlags = PC_NOCOLLAPSE;
      lpPal->palPalEntry[2].peRed = 0;
      lpPal->palPalEntry[2].peGreen = 255;
      lpPal->palPalEntry[2].peBlue = 0;
      lpPal->palPalEntry[2].peFlags = PC_NOCOLLAPSE;
      lpPal->palPalEntry[3].peRed = 0;
      lpPal->palPalEntry[3].peGreen = 0;
      lpPal->palPalEntry[3].peBlue = 255;
```

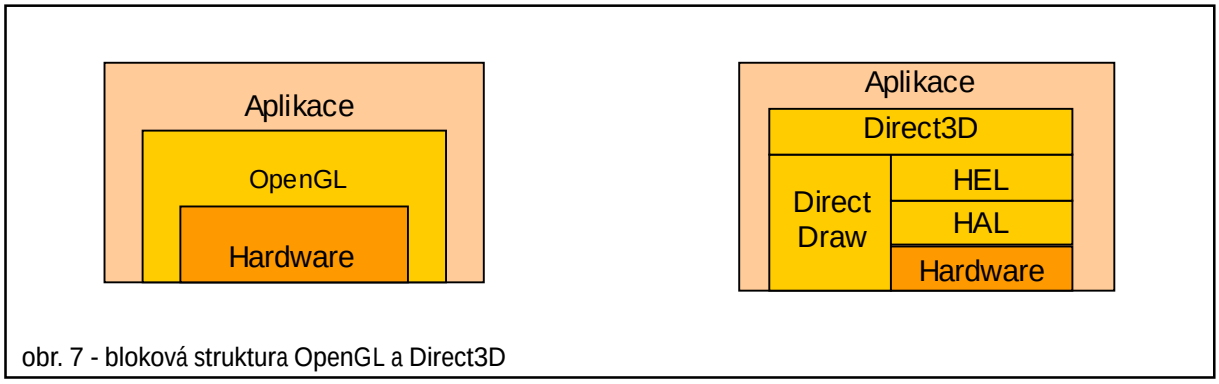
```
      lpPal->palPalEntry[3].peFlags = PC_NOCOLLAPSE;
    }
    hPalette = CreatePalette(lpPal);     // vytvoření barevné palety

  if (hPalette) {
      SelectPalette(hDC, hPalette, FALSE);
      RealizePalette(hDC);
    }
    free(lpPal);
     }
     ReleaseDC(hDC, hWnd);

     return hWnd;
}

int APIENTRY WinMain(HINSTANCE hCurrentInst, HINSTANCE hPreviousInst,
  LPSTR lpszCmdLine, int nCmdShow)
{
     HGLRC hRC;                    /* OpenGL kontext */
     HWND  hWnd;                   /* okno programu */
     MSG   msg;
     DWORD buffer = PFD_DOUBLEBUFFER;   /* způsob bufferování */
     BYTE  color  = PFD_TYPE_RGBA;      /* barevný režim */


     hWnd =
       CreateOpenGLWindow("animate", 0, 0, 256, 256, color, buffer);
     if (hWnd == NULL)             // při neúspěchu vytváření okna
   exit(1);                        // opusť program

     hDC = GetDC(hWnd);            // zjištění Device Contextu
     hRC = wglCreateContext(hDC);  // vytvoření rendrovacího kontextu
     wglMakeCurrent(hDC, hRC);     // nastavení RC na aktivní

     ShowWindow(hWnd, SW_SHOW);    // zobrazení okna
     UpdateWindow(hWnd);

     /*=================*/
     /* VLASTNÍ PROGRAM */
     /*=================*/

     wglMakeCurrent(NULL, NULL);   // na konci odoznač RC jako aktivní
     ReleaseDC(hDC, hWnd);         // uvolnění DC
     wglDeleteContext(hRC);        // zrušení RC
     DestroyWindow(hWnd);          // zrušení okna programu
     if (hPalette)                 // smazání palety
          DeleteObject(hPalette);

     return 0;
}
```

```
      glutPassiveMotionFunc(rotation);
    break;
  case GLUT_RIGHT_BUTTON: break;
  }
  glutPostRedisplay();            // překresli
}

void keys(unsigned char key, int x, int y) // ošetření klávesnice
{
  switch (key)
  {
  case 'O': case 'o':
    reverse=!reverse;
    break;
  case 'Z': case 'z':
    ZBUFF=!ZBUFF;
    break;
  case 'c': case 'C':
    CULL=!CULL;
    break;
  case 13:                        // Enter - výchozí poloha
    rot_y = 0.0;
    rot_x = 0.0;
    trans_z = -40.0;
    glutPostRedisplay();
    break;
  case 27:                        // Escape
    exit(0);
  }
}

int main(int argc, char** argv)       // hlavní funkce
{
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
  glutInitWindowPosition(50, 50);     // nastavení OpenGL v RGBA, s
  glutInitWindowSize(640, 480);       // dvojitým bufferováním a DB
  glutInit(&argc, argv);
  glutCreateWindow("Krychle");        // vytvoření okna
  init();                             // inicializační procedura
  glutDisplayFunc(display);           // přiřazení překreslovací
funkce
  glutReshapeFunc(reshape);           // funkce pro řešení změny okna,
  glutMouseFunc(mouse);               // zpracování myších a
klávesových
  glutKeyboardFunc(keys);             // událostí
  glutMotionFunc(NULL);               // zpočátku se ničím nehýbe
  glutPassiveMotionFunc(rotation);    // přiřazení funkce pro rotaci
  glutMainLoop();                     // spuštění správy událostí GLUT
  return 0;                           // konec programu
}
```

Srovnání OpenGL a Direct3D

V poslední době se významně prosazuje na poli trojrozměrné grafiky konkrurenční standard *Direct3D* firmy Microsoft. V oblasti, kde dříve jednoznačně kralovala OpenGL, je situace nejasná. Boj o pozice a o koncového zákazníka teprve v budoucnu ukáže, který standard přetrvá nebo jestli budou oba koexistovat vedle sebe zároveň

Direct3D je zcela nová knihovna, která byla navržena zejména pro tvorbu her. Jako součást multimediálního balíku DirectX má zajišťovat co nejlepší výkon grafických aplikací pod Windows. Programy napsané s využitím DirectX mohou prostřednictvím funkcí přímo přistupovat k systémovým zdrojům a využívat možností hardwarové akcelerace. Obchází se tak několikavrstvá struktura Windows API a grafické aplikace potom běží rychleji. Obrázek č. 7 zachychycuje blokovou strukturu OpenGL a Direct3D. Zatímco k OpenGL asi netřeba cokoliv dodávat, u Direct3D se uplatňuje několik vrstev. Vlastní Direct3D je odděleno od hardwaru, protože předem nelze říci, jaké funkce bude grafická karta přímo podporovat. Každá funkce, kterou aplikace volá je nejprve filtrována vrstvou HEL (Hardware Emulation Layer), která rozhodne, zda jí grafická karta umí provést nebo ne. Pokud ne, je příkaz vypočten softwarově pomocí této vrstvy. Jestliže funkci naopak grafická karta umí, použije se HAL (Hardware Activation Layer), která provede volání na konkrétním hardware. Mimo to Direct3D používá pro mapování textur pomocnou vrstvu *DirectDraw*, která je přístupná i přímo. Důvodem takto komplikovaného volání je možnost aplikace manipulovat s ovladači Direct3D a dotazovat se na to, co karta ve skutečnosti umí.



obr. 7 - bloková struktura OpenGL a Direct3D

Uvedením DirectX vyvolal Microsoft rozruch a to jak pozitivní, tak i negativní. Vývojáři her pod Windows dostali ucelený balík komponent, *DirectDraw*, *DirectSound*, *DirectPlay*, *Direct3D*, *DirectShow*, …, který dodával nové možnosti. Ze jmenovaných součástí je nejdiskutovanější právě Direct3D. Většina specialistů z oboru se shoduje v tom, že nevznikl důvod k zavedení nového standardu. Microsoft se pustil do návrhu vlastního 3D API, přitom OpenGL plně dostačovala všem aplikacím a byla prověřená časem. Programování v Direct3D se OpenGL příliš nepodobá. Direct3D přitom disponuje v zásadě stejnými funkcemi jako OpenGL.

Na rozdíl od Direct3D leží OpenGL přímo na hardwaru a volá přímo jeho služby. Ten, kdo OpenGL implementuje musí vědět, pro jaký typ grafické karty ho píše a co umí. Rozhodnutí, která služba bude vykonávána na procesoru je tedy na něm. Díky tomu, že se každá implementace z hlediska aplikace chová stejně, netestují se žádné ovladače. OpenGL předstírá, že umí úplně vše, záleží na implementaci, jakým způsobem a jak rychle bude služby poskytovat. OpenGL je přitom otevřený standard a implementace přitom musí projít náročnými testy, jak už zmíněno, aby se mohla honosit označením OpenGL. Direct3D naproti tomu vyvíjí jen Microsoft, a pouze on rozhoduje co se za touto značkou skrývá. Protože je OpenGL ve své koncepci striktně oddělena od jakéhokoliv okenního systému, běží téměř všude. Oproti tomu DirectX jsou pevně svázány s Windows, navíc donedávna neběhaly ani na Windows NT.

OpenGL v immediate módu umí všechno to co Direct3D a navíc poskytuje další funkce, jako třeba odstraňování zadních stěn před vykreslením (face-culling), správu šablon a paměti textur, trojrozměrné textury, akumulační buffer (používá se třeba k navození efektu rychlého pohybu s rozmazáním – motion blur), podporu parametrických křivek a ploch, atd. OpenGL i Direct3D používají Goraudovo stínování. Direct3D má teoretické předpoklady i pro Phongovo stínování, které by posululo grafický výstup blíže k realitě, tato funkce však nebyla doposud implementována.

Protějškem display-list módu OpenGL je u Direct3D tzv. retained mode. Tady poskytuje Direct3D podstatně lepší služby než OpenGL. Objekty na scéně je možné seskupovat do hierarchických struktur,

s kterými se dá pracovat na objektovém principu. Podobné funkce jsou u OpenGL dostupné pouze prostřednictvím určitých rozšíření. Za vymoženosti nového přístupu potom programátor zaplatí sníženým výkonem programu.

Přímé srovnání grafického výkonu OpenGL a Direct3D je docela komplikované, protože obě API mají trochu jiné konečné určení. OpenGL a Direct3D je navíc třeba srovnávat pouze pod Windows, protože jinde Direct3D nechodí. OpenGL poskytuje kvalitní zobrazení trojrozměrné scény, které je použitelné i pro seriózní aplikace (nehry). Direct3D přitom ani tolik nehledí na výsledek, ale je optimalizován na rychlost. Pro Direct3D zatím chybí rozsáhlejší výkonové testy, které by ukázaly, jak si vedou programy na různých konfiguracích počítače.

Významným rysem API je i to, jak dlouhý kód programu se musí napsat, aby se něco provedlo. Direct3D v tomto ohledu jasně vede. Lze se jen odkázat na stránku autora populární hry Quake, Johna Carmacka [6], který se svěřuje, že kód programu plnící stejnou funkci je v Direct3D čtyřikrát delší než v OpenGL. Navíc verze s Direct3D pod Windows, je podle něj pomalejší.

OpenGL se stala úspěšným průmyslovým standardem a její pozice zůstává do značné míry neotřesena. Významné je přitom, že za osm let se standard změnil pouze minimálně, hlavní vylepšení jsou v implementacích. U DirectX jsme svědky rychlého vývoje verzí, které často nejsou ani zpětně kompatibilní. Zatímco současná verze OpenGL je 1.2, dají se verze DirectX počítat na prstech obou rukou a i ty přestávají pomalu stačit.

Trochu překvapivě působí nedávno uzavřená těsná spolupráce mezi Microsoftem a Silicon Graphics na vývoji nového standardu, který by svázal přednosti obou. Projekt spojení OpenGL a DirectX se zatím označuje kódovým jménem *Fahrenheit*. Podle uveřejněných zpráv se má *Fahrenheit* zkládat ze tří vrstev, nejnižší *nízkoúrovňová* vyjde z OpenGL, Direct3D a DirectDraw, čímž má být zachována zpětná kompatibilita s aplikacemi a ovladači hardwaru napsanými pro DirectX a současně plná přenositelnost programů na úrovni zdrojového kódu s OpenGL. Druhá úroveň *Fahrenheit Scene Graph* má potom umožnovat vyšší programovou abstrakci. Konečně třetí vrstva *Large Model Visualization Extensions*, má sloužit pro manipulaci s rozsáhlou scénou a objekty o několika milionech trojúhelníků. Projekt Fahrenheit je teprve v první fázi vývoje, proto si budeme muset na konečné řešení teprve počkat.

Literatura:

❶	OpenGL Programming Guide	Addison Wesley 1997
❷	OpenGL 1.2 System Specification	Silicon Graphics Inc. 1999
❸	OpenGL Graphics with X-Window System	Silicon Graphics Inc. 1998
❹	OpenGL Reference Manual	Addison Wesley 1994
❺	Tomáš Holub: Jemný úvod do OpenGL	
❻	The OpenGL Graphics Interface	Silicon Graphics Inc.
❼	The OpenGL Graphics System Utility Library	Silicon Graphics Inc.
❽	Chip 3/98, 6/98	
❾	Computer 3/98	

Odkazy:

①	Základy počítačové grafiky	www.zcu.cz/..
②	Ceská stránka o OpenGL	www.opengl.cz
③	Domovská stránka OpenGL	www.opengl.com
④	OpenGL stránka Silicon Graphics	www.sgi.com/opengl
⑤	OpenGL Benchmark Tests	www.specbench.org/gpc/opc.static/index.html
⑥	Porovnání s Direct3D od autora Quake	www.opengl.org/developers/documentation/white_papers/direct-3d.html