Anino BELAN

Knižnica OpenGL

učebný text pre oktávu osemročného gymnázia

BRATISLAVA
2007

Obsah

Úvod

Tento kurz práce s knižnicou OpenGL, podobne ako ostatné kurzy na našej škole, s ktorými ste sa mali možnosť stretnúť, nie je ani referenčná príručka, ani dokonalá učebnica. Jeho účelom je ukázať niektoré základné postupy a finty, aby človek, ktorý ním prejde, bol schopný samostatne experimentovať, skúšať a zisťovať, čo všetko sa musí naučiť poriadne (a z lepšej literatúry, než je táto), aby mohol napísať programy, ktoré napísať chce. Okrem toho je účelom tohto kurzu samozrejme aj ukázať, že OpenGL je pekná knižnica v ktorej sa dajú urobiť veľkolepé veci a prebudiť túžbu niečo pekné naprogramovať.

Pri čítaní toho kurzu sa predpokladá aspoň elementárna znalosť jazyka C++ a práce s grafickou knižnicou Allegro. Knižnica OpenGL ako taká nie je viazaná ani na jedno, ani na druhé. Rovnako, ako v jazyku C++ ju môžete použiť aj v Delphi, Pythone či Ruby a ako podpornú knižnicu použiť Glut, SDL, Qt alebo WinAPI. Voľba bola daná tým, že na našej škole sa učí C++ a Allegro a ten, kto chce použiť iný jazyk alebo knižnicu si niektoré detaily bude musieť naštudovať z iných zdrojov.

Chcel by som poďakovať človeku, ktorý sa volá Jeff Molofee (alias NeHe), ktorý s nejakými ďalšími ľuďmi spísal veľmi pekné OpenGL tutoriály, z ktorých som sa veľa naučil. Sú také dobré, že som váhal, či sa vôbec oplatí písať nejaký nový kurz. Nakoniec som sa k tomu odhodlal iba preto, že v tých tutoriáloch nie sú zadané žiadne úlohy – aj keď istú rolu zohrala aj túžba vyskúšať si to po svojom. Moja vďaka patrí aj Michalovi Turkovi, ktorý spravuje stránku s českým prekladom NeHe tutoriálov na adrese http://nehe.ceskehry.cz/ Každého, kto má záujem o ďalšie informácie alebo o podrobnejší výklad niečoho, čo som spravil prirýchlo, odkazujem na túto stránku.

Ďalšia základná literatúra k téme, ktorú vrelo odporúčam, ktorá mi veľmi pomohla a ktorú treba mať pri programovaní v OpenGL po ruke je knižka známa pod prezývkou „The Red Book“ od aurotov Jackieho Neidera, Toma Davisa a Masona Woo, ktorú môžete nájsť (v angličtine) napríklad na adrese http://fly.cc.fer.hr/~unreal/theredbook/ a ktorá na rozdiel od tohto kurzu spracovala tému naozaj podrobne.

A na záver by bola hanba nespomenúť stránku http://www.opengl.org/ na ktorej nájdete mnohé čerstvé informácie, najnovšie rozšírenia knižnice aj zaujímavé kusy kódu.

Prajem vám príjemnú zábavu.

Anino

1. lekcia
Úvod do OpenGL alebo „Štartujeme stroj“

Programátori sa už pred dávnym časom chytili rozumu a miesto toho, aby zakaždým všetko programovali od úplného začiatku, začali vyrábať knižnice funkcií, ktoré by sa dali znovu a znovu používať. Existujú knižnice, ktoré zjednodušujú prácu so súbormi, knižnice, s pomocou ktorých sa príjemne programuje komunikácia po sieti, knižnice obsahujúce matematické funkcie, knižnice určené na tvorbu používateľského rozhrania či knižnice na prácu s grafikou. Niektoré sú malé a šikovné, iné obsahujú kvantá rozličných funkcií z rôznych oblastí, niektoré sú vytvorené pre jeden konkrétny operačný systém, iné sú multiplatformné.

Knižnica, o ktorej bude reč v tomto kurze sa nazýva OpenGL (z anglického Open Graphics Library). Je to knižnica pomerne slávna a je určená výhradne na prácu s grafikou. Za jej vznik môže firma Silicon Graphics.[1] Začiatkom deväťdesiatych rokov dvadsiateho storočia sa táto firma prepracovala na absolútnu špičku, čo sa týka počítačovej grafiky. Jednak vyrábala vo svojej dobe najlepší a najrýchlejší hardvér, jednak mala vymyslenú veľmi príjemnú knižnicu na prácu s ním – táto knižnica sa volala IrisGL a stala sa v podstate priemyselným štandardom. V roku 1992 knižnicu prepracovali, vysekali z nej veci, ktoré sa netýkali grafiky a pod názvom OpenGL vydali ako otvorený štandard. To spätne ovplyvnilo výrobcov grafických kariet, ktorí pracovali na tom, aby hardvérovo zrýchlili práve funkcie z tohto rozhrania.

O aktuálnu verziu OpenGL 2.0, ktorá vyšla v roku 2004 sa zaslúžila hlavne firma 3Dlabs.[2] Dnes má na starosti špecifikáciu knižnice OpenGL konzorcium Khronos Group.[3] Knižnica je určená na prácu s 3D (ale aj 2D) grafikou. Využíva sa pri tvorbe projektovacích nástrojov, virtuálnej reality, vedeckých či informačných vizualizácii, letových simulátorov. Veľké využitie je tiež v oblasti herného priemyslu.[4]

Dobre. Toľko historický úvod. Teraz by bola na mieste otázka, prečo sa vlastne učíme pracovať s ďalšou grafickou knižnicou, keď už z jednou robiť vieme. Odpoveď je zrejmá – lebo je lepšia. Allegro je fajn na prácu s 2D grafikou, aj 3D sa s trochou snahy dá, ale tá 3D grafika s Allegrom žiaľ nemá priamu podporu grafickej karty, všetko si to treba počítať a je to pomalé. No fajn. A prečo sme sa potom neučili rovno OpenGL a nevykašľali sme sa na Allegro? Dôvody sú dva. Jeden je ten, že nie na všetko treba 3D grafiku. A na 2D je Allegro dobré. Druhý je ten, že OpenGL vie naozaj iba tú grafiku. Nevie nič o čítaní z klávesnice, o prehrávaní zvukov ani o timeroch. Nevie si prečítať obrázok zo súboru a keby sme mu tieto radosti nedodali z nejakej inej knižnice, museli by sme si ich naprogramovať sami. Takže v našich projektoch bude veselo spolunažívať OpenGL s Allegrom.

Toto spolužitie bude zrealizované s pomocou knižnice AllegroGL, ktorá vie sprístupniť Allegru funkcie z OpenGL a vie aj niektoré iné zaujímavé veci. OpenGL ale samozrejme nemusíte používať iba spolu s Allegrom. Existujú rozhrania, ktoré vám umožnia použiť OpenGL s mnohými inými knižnicami (napríklad s Windows API, SDL alebo Qt).

V dnešnej lekcii si povieme, ako naštartovať Allegro, aby to fungovalo aj s OpenGL.

1 O nej ste už počuli napríklad v súvislosti s knižnicou šablón STL.

2 3Dlabs je firma, ktorá vyrábala grafické karty a bola súčasťou CreativeLabs. Keď vo februári 2006 firma prestala grafické karty vyvíjať, väčšina zamestnancov prešla k Intelu a NVidii.

3 Údaj zo septembra 2006.

4 V oblasti hier OpenGL zdatného konkurenta v podobe knižnice Direct3D, ktorú Microsoft kúpil spolu s firmou RenderMorphics a ktorá je súčasťou DirectX. Tá však funguje len na operačných systémoch od Microsoftu.

Vytvoríme kostru programu, ktorú budete vo všetkých ďalších lekciach využívať a v ďalšom budeme meniť už iba detaily (takže v podstate všetku ťažkú robotu vybavíme dnes).

Takže tu je náš zdrojový kód (nazvime ho `01.cpp`) v celej svojej kráse:

```
#include <allegro.h>
#include <GL/glu.h>
#include <alleggl.h>


bool InitGL(GLvoid)
{
     int width = 640;
     int height = 480;
     glViewport(0,0,width,height);
     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();
     gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,0.1f,100.0f);
     glMatrixMode(GL_MODELVIEW);
     glLoadIdentity();

     return true;
}

bool DrawGLScene(GLvoid)
{
     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
     glLoadIdentity();

     glBegin(GL_TRIANGLES);
          glColor3ub(255, 0, 0);
          glVertex3f(-1.0f, -1.0f, -5.0f);
          glColor3ub(0, 255, 0);
          glVertex3f( 1.0f, -1.0f, -5.0f);
          glColor3ub(0, 0, 255);
          glVertex3f( 0.0f,  1.0f, -5.0f);
     glEnd();

     allegro_gl_flip();

     return true;
}

int main()
{
     allegro_init();

     install_allegro_gl();

     allegro_gl_clear_settings();
     allegro_gl_set(AGL_Z_DEPTH, 16);
     allegro_gl_set(AGL_DOUBLEBUFFER, 1);
     allegro_gl_set(AGL_COLOR_DEPTH, 16);

     allegro_gl_set(AGL_SUGGEST,
          AGL_Z_DEPTH | AGL_DOUBLEBUFFER | AGL_COLOR_DEPTH);

     set_color_depth(16);
     if (set_gfx_mode(GFX_OPENGL_FULLSCREEN, 640, 480, 0, 0) < 0)
     {
          allegro_message ("Neviem nastavit graficky mod:\n%s\n",
                                                    allegro_error);
          return -1;
     }

     install_keyboard();

     InitGL();
```

```
    while (DrawGLScene())
    {
        if (key[KEY_ESC])
            break;
    }

    return 0;
}
END_OF_MAIN();
```

Začnime funkciou `main()`. Najprv inicializujeme Allegro a potom volaním funkcie `install_allegro_gl` aj AllegroGL. Potom urobíme nastavenia, ktoré treba urobiť ešte pred inicializáciou grafického režimu. Kebyže používame iba Allegro, stačí nám nastaviť farebnú hĺbku. (To aj tak urobíme volaním funkcie `set_color_depth`.) Okrem toho ale ešte musíme nastaviť farebnú hĺbku pre OpenGL (16 znamená 2^{16} farieb rovnako ako sme to nastavili pre Allegro). Ďalej nastavíme Z-buffer (to je „zariadenie“, ktoré slúži na to, aby veci, ktoré sú vpredu prekrývali veci, ktoré sú vzadu bez ohľadu na to, v akom poradí boli vykreslené – čím väčšie číslo, tým lepšie to pracuje ale tým viac pamäte zožerie) a povieme, že budeme používať doublebuffer (1 = použiť, 0 = nepoužiť). Doublebuffer je dobrý na to, že počítač, zobrazuje jednu stránku, kým my kreslíme ďalšiu a keď dokreslíme, tak mu funkciou `allegro_gl_flip` povieme „tak a teraz ich vymeň“.

Všetky tieto nastavovačky vybavíme funkciou `allegro_gl_set`. Najprv s jej pomocou priradíme jednotlivým parametrom hodnoty a posledným volaním povieme, ktoré parametre má ponastavovať. Konštanta `AGL_SUGGEST` znamená, že sa s tým našim nastavením až tak veľmi nevnucujeme a keby to náhodou nebehalo, systém to môže vybrať aj inak. Iné zaujímavé parametre nájdete v dokumentácii k AllegroGL, ale tieto zatiaľ budú stačiť.

Potom inicializujeme grafický režim. Deje sa to v podstate klasicky až na ten detail, že používame režim `GFX_OPENGL_FULLSCREEN`. (Funguje aj `GFX_OPENGL_WINDOWED`.) Aby sme mohli nejako rozumne skončiť, nainštalujeme klávesnicu.

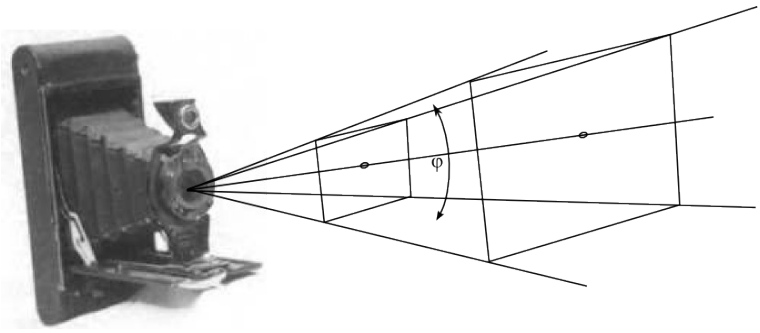
Zavoláme funkciu `InitGL()`, ktorú popíšeme neskôr a ktorá nám nastaví veci dôležité pre OpenGL a potom voláme stále znova a znova funkciu `DrawGLScene()`, ktorá obstaráva samotné kreslenie, až kým sa v nej niečo zlé nezomelie (čo sa v našom prípade nestane, funkcia vždy vráti `true`) alebo niekto nestlačí `Esc`.

A konečne sa dostávame k samotnému OpenGL. Funkcie z tejto knižnice spoznáme podľa toho, že začínajú na `gl` (podobne funkcie z „nadstavbovej“ knižnice GLU začínajú na `glu`). Najprv popíšeme, čo sa udeje vo funkcii `InitGL()`.

Funkcia `glViewport` povie knižnici, aký fyzický priestor má naozaj k dispozícii – na akom bode začína, akú má šírku a výšku v pixeloch. Ďalej ideme nastaviť metódu zobrazovania – v našom prípade to bude perspektíva. Chceme totiž, aby sa predmety, ktoré sú od nás ďalej, javili menšie. Príkazom `glMatrixMode(GL_PROJECTION)` povieme, že všetky nasledujúce nastavovačky sa budú týkať spôsobu, ako sa na scénu pozeráme. (Ak chcete, tak „kamery“.) Príkaz `glLoadIdentity` kameru resetne – uvedie ju do základného stavu. Samotné nastavenie je na príkaze `gluPerspective`. Kamera sa umiestni do počiatku súradnicovej sústavy a pozerá sa v smere osi `z` smerom, ktorým z-ová súradnica klesá. Prvé dva parametre určujú ihlan, ktorý má kamera v zábere. Prvý je uhol medzi hornou a dolnou rovinou a druhý je pomer medzi šírkou a výškou obrazu. (Tieto údaje stačia na to, aby bol ihlan určený.) Ďalšie dva parametre určujú, odkiaľ pokiaľ kamera vidí. V našom prípade teda nebude vidieť objekty so súradnicou `z` väčšou ako -0,1 alebo menšou ako -100.[5]

5 Kamera je otočená „do mínusu“ !!!

Kamera je teda nastavená. S pomocou funkcie `glMatrixMode(GL_MODELVIEW)` dáme vedieť, že všetky ďalšie nastavenia sa už budú týkať objektov, ktoré budeme do scény vkladať a nastavenie pre modely resetneme príkazom `glLoadIdentity`.



Poďme sa teraz venovať kresleniu scény, ktoré má na starosti funkcia `DrawGLScene`. Najprv všetko zmažeme príkazom `glClear`. Parameter hovorí, že sa má zmazať všetko, čo je nakreslené, aj všetky údaje o tom, ktoré veci sú viac vpredu (spomeňte na Z-buffer). Plocha sa štandardne premazáva na čierno, o tom, ako sa to dá zmeniť, si povieme neskôr. Znova všetko resetneme príkazom `glLoadIdentity`. Je fakt, že sme to spravili na konci funkcie `InitGL`, ale procedúra kreslenia sa volá mnohokrát a počas posledného kreslenia sa veci mohli zmeniť.

Konečne ideme kresliť. Pre začiatok nakreslíme jeden trojuholník. Že ideme kresliť práve trojuholníky, dáme vedieť príkazom `glBegin(GL_TRIANGLES)`. Po tomto príkaze budeme zadávať body a z každých troch bodov, ktoré určíme, urobí OpenGL trojuholník. V našom prípade sme zadali len tri body, takže trojuholník je len jeden. Keby sme ich zadali 10, trojuholníky by boli tri a posledný bod by bol odignorovaný. Každý bod sme zadali príkazom `glVertex3f`. Tá trojka v názve znamená, že nasledujú tri parametre – súradnice v priestore. To `f` znamená, že budú typu `float`.

Aby to nebola až taká nuda, každému vrcholu sme určili farbu – a pre istotu každému inú. Farby sme určovali príkazom `glColor3ub`, ktorý mal ako parametre tri čísla od 0 do 255 (`ub` znamená `unsigned byte`). Tento spôsob sme zvolili, pretože ste naň zvyknutí z Allegra. Bežnejšie sa v OpenGL používa funkcia `glColor3f` ktorá má parametre typu float, ktoré nadobúdajú hodnotu od 0 do 1. Takže napríklad zelená by bola `glColor3f(0.0f,1.0f,0.0f)` a oranžová by bola `glColor3f(1.0f,0.5f,0.0f)`. Ale používajte, čo sa vám páči.

Kreslenie trojuholníkov ukončíme príkazom `glEnd()` – pozor, tento príkaz nemá parametre. A to, čo sme práve nakreslili, pustíme na obrazovku príkazom `allegro_gl_flip`. Nie že by sa tam objavilo niečo nové, ale to je tým, že zatiaľ nič neanimujeme.

Dobre. Zdroják by bol, teraz to treba skompilovať. Pod linuxom sa to spraví príkazom

```
g++ -o 01 01.cpp -lagl -lalleg -lalleg_unsharable -lGL -lGLU
```

ak kompilujete s MinGW pod Windowsom, použite príkaz

```
g++ -o 01.exe 01.cpp-lagl -lalleg -luser32 -lgdi32 -lopengl32
     -lglu32
```

Pozor! Poradie, v akom zadávate tie knižnice na konci je dôležité.

Úloha č.1: Vyskúšajte.

Úloha č.2: Nakreslite modrý štvorček. (Farbu stačí určiť raz pred prvým vrcholom. Bude platiť, až kým nepoviete, že nastala zmena.)

Úloha č.3: Nakreslite koliesko a nejako zaujímavo ho vyfarbite. (Ak používate funkcie `sin` a `cos`, treba na začiatok zdrojáku pripísať `#include <math.h>` a pri kompilácii ešte na koniec prihodiť `-lm` aby sa prilinkovala matematická knižnica.)

Úloha č.4: Dorobte ovládanie klávesnicou. Pri stlačení šipky hore sa bude trojuholník vzďaľovať a pri stlačení šipky dole sa bude približovať.

2. lekcia
Transformácie
alebo „Zatočíme súradnicovou sústavou“

Keďže je knižnica OpenGL určená v prvom rade na tvorbu 3D grafiky, skúsime v tejto lekcii spraviť kocku. A pokúsime sa aj o to, aby sa trochu vrtela.

V predošlej lekcii sme nakreslili len jeden trojuholník, takže sme sa nemuseli starať o to, ktorá stena je vpredu, ktorá vzadu a o to, aby nám náhodou nejaká zadná stena neprekreslila prednú. Teraz si to už budeme musieť všímať – teda ani tak nie my, ako skôr počítač. V knižnici OpenGL sa o to, aby k takémuto neželanému prekresleniu nedochádzalo stará Z-buffer – finta, ktorú sme spomínali už v predošlej lekcii, keď sme štartovali AllegroGL a mazali obrazovku. Funguje to tak, že pre každý bod, ktorý sa vykreslí, si počítač pamätá nielen jeho farbu, ale aj vzdialenosť od kamery (súradnicu `z`). A keď má na dané miesto nakresliť ďalší bod, urobí to iba vtedy, ak je nový bod bližšie ku kamere. Knižnici OpenGL je ale treba povedať, že má túto fintu použiť.[6] Spravíme to tak, že do funkcie `InitGL`, v ktorej sme štartovali OpenGL pridáme nasledujúce riadky:

```
glClearDepth(1.0f);
glDepthFunc(GL_LESS);
glEnable(GL_DEPTH_TEST);
```

Funkcia `glClearDepth` nastaví hodnotu, ktorú priradí Z-buffer každému bodu pri mazaní. Parameter funkcie je číslo od 0 do 1, hodnota 1 znamená „najďalej, ako sa len dá“ – teda zadnú hranicu oblasti, ktorú kamera ešte sníma. Objekty za touto hranicou sa aj tak na obrazovke neobjavia a každý bližší sa vykreslí. Funkcia `glDepthFunc` zas nastaví, aký vzťah musí nový bod oproti údaju v bufferi spĺňať, aby bol vykreslený – v našom prípade musí byť ku kamere bližšie, než doteraz vykreslené body.[7] Funkcia `glEnable` zapína a vypína v knižnici OpenGL rôzne veci, v tomto prípade zapne používanie Z-buffra.

Takže Z-buffer je úspešne zapnutý, môžeme kresliť kocku. A to nie len takú hocijakú kocku, ale kocku rotujúcu.

Každá kocka bude reprezentovaná súradnicami v priestore (premenné `x`, `y` a `z`), natočením v smere osi `x` a osi `y` (premenné `xrot` a `yrot`) a rýchlosťou, ktorou sa bude kocka v každom zo zadaných smerov otáčať (premenné `dxr` a `dyr`). Kocka bude mať dve súkromné metódy: metódu `Pootoc`, ktorá podľa rýchlosti zmení natočenie kocky a metódu `KresliSteny`, ktorá sa, ako ste isto uhádli, postará o vykreslenie štvoruholníkov tvoriacich steny kocky. Okrem konštruktora bude mať dve verejné metódy: metódu `Nastav`, ktorá nastaví kocke novú polohu a metódu `Nakresli`, ktorá ju nakreslí. Hlavičkový súbor `kocka.h` bude preto vyzerať takto:

```
#ifndef _KOCKA_H_
#define _KOCKA_H_

class Kocka
{
public:
     Kocka(GLfloat nx = 0.0f, GLfloat ny = 0.0f, GLfloat nz = 0.0f);
     void Nastav(GLfloat nx = 0.0f, GLfloat ny = 0.0f, GLfloat nz = 0.0f);
     bool Nakresli();
```

6 Nie vždy je to vhodné. Keď napríklad budeme kresliť priesvitné objekty, budeme túto vlastnosť musieť vypnúť.

7 Hodnoty, ktoré sme nastavovali funkciami `glClearDepth` a `glDepthFunc` sú štandardné. Program by fungoval úplne rovnako, aj keby sme ich vynechali. Išlo nám iba o to ukázať, že takéto veci sa dajú nastaviť.

```
protected:
     void KresliSteny();
     void Pootoc();
     GLfloat x,y,z;        // suradnice
     GLfloat xrot, yrot;   // natocenie v smere osi x a y
     GLfloat dxr, dyr;     // rychlost otacania okolo x a y

};

#endif
```

Poďme sa teraz venovať súboru `kocka.cpp` a pozrieť sa na implementáciu jednotlivých metód. Najprv tie nudnejšie metódy. Metóda `Nastav` naozaj iba nastaví zadané súradnice (a ak nikto žiadne nezadá, použije štandardné hodnoty z hlavičkového súboru):

```
void Kocka::Nastav(GLfloat nx, GLfloat ny, GLfloat nz)
{
     x = nx;
     y = ny;
     z = nz;
}
```

Konštruktor okrem nastavenia súradníc (prípadne ich vynulovania, ak nie sú zadané) nastaví počiatočné natočenie na nulu a náhodne vygeneruje rýchlosť otáčania:

```
Kocka::Kocka(GLfloat nx, GLfloat ny, GLfloat nz): x(nx), y(ny), z(nz)
{
     xrot = yrot = 0.0f;
     dxr = (rand() % 10) / 10.0f - 0.5f;
     dyr = (rand() % 10) / 10.0f - 0.5f;
}
```

Metóda `Pootoc` zmení natočenie kocky o rýchlosť a v prípade, že natočenie v niektorom smere vylezie z intervalu 0 až 360 stupňov, prevedie patričnú korekciu.

```
void Kocka::Pootoc()
{
     xrot += dxr;
     xrot = xrot > 360.0f ? xrot - 360.0f : xrot;
     xrot = xrot < 0.0f ? xrot + 360.0f : xrot;
     yrot += dyr;
     yrot = yrot > 360.0f ? yrot - 360.0f : yrot;
     yrot = yrot < 0.0f ? yrot + 360.0f : yrot;
}
```

Technické veci máme za sebou. Poďme sa venovať OpenGL. Takže najprv metóda `KresliSteny`. Budeme ich kresliť ako štvoruholníky (`GL_QUADS`) a každú stenu zafarbíme nejakým iným odtieňom modrej:

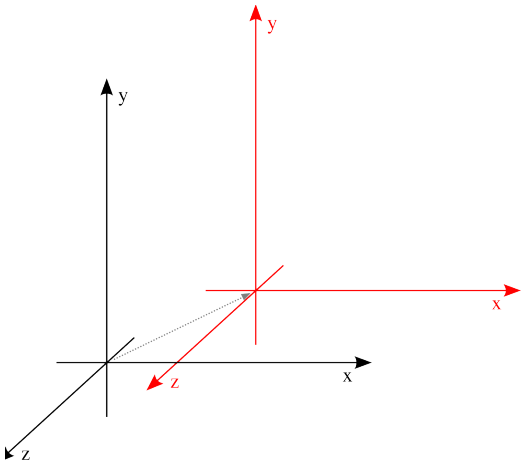
```
void Kocka::KresliSteny()
{
     glBegin(GL_QUADS);
          // Predna stena
          glColor3f(0.0f, 0.0f, 1.0f);
          glVertex3f(-1.0f, -1.0f, 1.0f);
          glVertex3f(-1.0f,  1.0f, 1.0f);
          glVertex3f( 1.0f,  1.0f, 1.0f);
          glVertex3f( 1.0f, -1.0f, 1.0f);
          // Prava stena
          glColor3f(0.0f, 0.0f, 0.6f);
          glVertex3f(1.0f, -1.0f, -1.0f);
          glVertex3f(1.0f, -1.0f,  1.0f);
          glVertex3f(1.0f,  1.0f,  1.0f);
          glVertex3f(1.0f,  1.0f, -1.0f);
          // Zadna stena
          glColor3f(0.6f, 0.6f, 1.0f);
          glVertex3f(-1.0f, -1.0f, -1.0f);
          glVertex3f(-1.0f,  1.0f, -1.0f);
```

```
            glVertex3f( 1.0f,  1.0f, -1.0f);
            glVertex3f( 1.0f, -1.0f, -1.0f);
            // Lava stena
            glColor3f(0.0f, 0.6f, 1.0f);
            glVertex3f(-1.0f, -1.0f, -1.0f);
            glVertex3f(-1.0f, -1.0f,  1.0f);
            glVertex3f(-1.0f,  1.0f,  1.0f);
            glVertex3f(-1.0f,  1.0f, -1.0f);
            // Horna stena
            glColor3f(0.2f, 0.0f, 0.8f);
            glVertex3f(-1.0f, 1.0f, -1.0f);
            glVertex3f(-1.0f, 1.0f,  1.0f);
            glVertex3f( 1.0f, 1.0f,  1.0f);
            glVertex3f( 1.0f, 1.0f, -1.0f);
            // Dolna stena
            glColor3f(0.2f, 0.2f, 1.0f);
            glVertex3f(-1.0f, -1.0f, -1.0f);
            glVertex3f(-1.0f, -1.0f,  1.0f);
            glVertex3f( 1.0f, -1.0f,  1.0f);
            glVertex3f( 1.0f, -1.0f, -1.0f);
      glEnd();
}
```

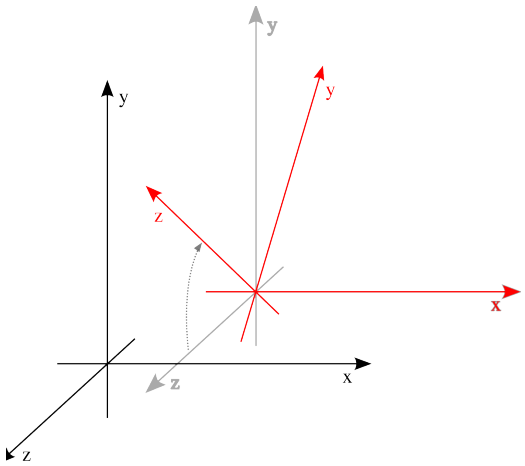
Všimnite si, že všetky súradnice bodov sú nastavené úplne napevno. Momentálne to vyzerá tak, že všetko, čo sme tu doteraz písali o polohe kocky a jej natočení, všetky tie premenné, ktoré sme kvôli tomu zriadili a celá príprava vyjde nazmar. Metóda `KresliSteny` nič z toho nepoužíva. Knižnica OpenGL ale posúvanie a natočenie objektov na patričné miesto rieši šikovnejšie, než tak, že by sme museli prepočítavať všetky súradnice každého objektu, ktorý použijeme. Ona totiž posunie celú súradnicovú sústavu. To je oveľa príjemnejší prístup. Ak by sme mali nejaký komplikovanejší objekt, ako je kocka, pri posúvaní a otáčaní by výpočty nemali konca kraja. Takto môžeme objekt vykresľovať stále rovnako a napriek tomu môžeme dosiahnuť, aby bol presne tam, kde potrebujeme. Takže sa pozrime, ako to bude vyzerať v praxi:

```
bool Kocka::Nakresli()
{
      glLoadIdentity();
      glTranslatef(x, y, z);
      glRotatef(xrot, 1.0f, 0.0f, 0.0f);
      glRotatef(yrot, 0.0f, 1.0f, 0.0f);
      KresliSteny();
      Pootoc();
      return true;
}
```

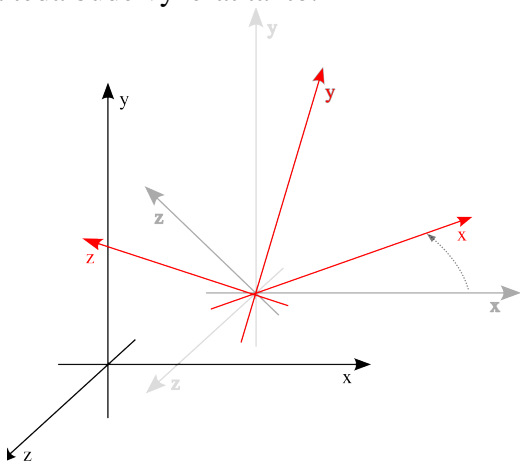
Najprv zavoláme funkciu `glLoadIdentity`. Tá spôsobí, že súradnicový systém sa vráti na začiatok (môže sa stať, že práve na začiatku nie je, pretože sme pred chvíľou niekde kreslili nejakú inú kocku). Potom zavoláme funkciu `glTranslate`, ktorá nám súradnicovú sústavu posunie do bodu `[x,y,z]`.



Funkcia `glRotatef` nám otočí súradnicovú sústavu o daný uhol okolo daného vektora. Toto sa deje už „z pohľadu“ novej súradnicovej sústavy. Takže po zavolaní `glRotatef(xrot, 1.0f, 0.0f, 0.0f)` sa súradnicová sústava otočí okolo vektora `(1,0,0)` o `xrot` a teda udeje sa toto:



Nakoniec ešte otočíme súradnicovú sústavu okolo najnovšej varianty osi `y` o uhol `yrot`. Výsledná súradnicová sústava teda bude vyzerať takto:



No a v tejto výslednej súradnicovej sústave nakreslíme našu kocku (zavoláme patričnú metódu). Žiaden div, že bude vyzerať otočená.

Pozor!!! Na poradí, v akom vykonávame jednotlivé transformácie veľmi záleží. Keby sme súradnicovú sústavu najprv otočili a potom posúvali, tak by sa miesto posunutia určovalo už v nových – teda otočených súradniciach. Súradnice by sa tým pádom posunuli nejakým neželaným smerom a pravdepodobne by nám objekt, ktorý chceme zobrazovať, úplne ušiel z dohľadu kamery.

Na záver vykresľovania ešte kockou pootočíme, aby pri ďalšom vykresľovaní vyzerala, že sa pohla a vrátime hodnotu, že všetko dobre dopadlo. Súbor `kocka.cpp` môžeme skúsiť skompilovať a ak sa budeme diviť, že prečo to nejde, pravdepodobne sme na začiatok zabudli pridať

```
#include <allegro.h>
#include <GL/glu.h>
#include <alleggl.h>
#include "kocka.h"
```

Poďme sa opäť venovať nášmu hlavnému súboru. Okrem zmien, ktoré sme vykonali na začiatku kvôli Z-bufferu by sa v ňom ešte patrilo nejako použiť kocky, ktoré sme práve vytvorili. A že nie sme žiadni trochári, dáme ich tam hneď desať.

Na začiatok súboru pridáme `#include "kocka.h"`. Dopredu (teda ešte pred prvú

funkciu) pridáme deklaráciu poľa kociek:

```
Kocka k[10];
```

Mimo funkcii, teda ako globálnu premennú, ho deklarujeme preto, aby sme ho mohli používať aj vo funkcii `main` – tam ho inicializujeme, aj vo funkcii `DrawGLScene` – tam ho budeme vykresľovať.

Tá časť funkcie `main`, ktorá nasleduje po `InitGL` bude vyzerať takto:

```
srand(time(NULL));
for(int i = 0; i < 10; i++)
     k[i].Nastav((rand() % 100) / 10.0f - 5.0f,
                 (rand() % 100) / 10.0f - 5.0f,
                 -15.0f - (rand() % 100) / 10.0f);

while (DrawGLScene())
{
     if (key[KEY_ESC])
          break;
}

return 0;
```

V podstate sa tam nič zvláštne nedeje. Najprv všetkým kockám nastavíme náhodnú polohu a potom až do odvolania voláme vykresľovaciu funkciu `DrawGLScene`. Tá bude vyzerať takto:

```
bool DrawGLScene(GLvoid)
{
     bool ret = true;

     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
     for(int i = 0; i < 10; i++)
          ret = ret && k[i].Nakresli();

     allegro_gl_flip();

     return ret;
}
```

Na začiatku kreslenia zmažeme plochu a nakreslíme všetkých desať kociek, pričom si v premennej `ret` pamätáme, či náhodou niektoré kreslenie neskončilo s hodnotou `false`. (Áno, ja viem, že neskončilo. V zložitejších prípadoch by ale mohlo.) Pošleme nakreslenú stránku na monitor a vrátime `true`, ak všetky kreslenia prebehli dobre a `false` inak.

Úloha č.1: Vyskúšajte.

Úloha č.2: Skúste, ako to bude vyzerať, ak nezapnete Z-buffer.

Úloha č.3: Vyskúšajte miesto rotácie okolo dvoch vektorov nechať rotovať kocku iba okolo jedného – okolo vektora `(1,1,1)`

Úloha č.4: Okrem funkcií na posúvanie a otáčanie má OpenGL aj funkciu `glScalef(x,y,z)`, ktorá vie súradnicové osi v určenom smere natiahnuť. Zmeňte kocku, aby okrem rotovania ešte aj pulzovala – v určitom rozmedzí opakovane rástla a zmenšovala sa.

Úloha č.5: Dorobte kocke verejné metódy `Brzda` a `Plyn`, ktoré dostanú ako parameter niektorú z osí a zvýšia resp. znížia rýchlosť otáčania okolo patričnej osi. Vo funkcii `main` ošetrite vstup z klávesnice tak, aby používateľ mohol niektorú z kociek ovládať.

3. lekcia
Svetlo a hmla alebo „Musíme si na to posvietiť“

V príklade z predošlej lekcie sa nám úspešne podarilo nakresliť desať v priestore rotujúcich kociek. Vyzerali celkom vierohodne. Malo to ale drobnú chybu. Totiž – predstavte si, že držíte v ruke jednofarebnú kocku zo stavebnice a len tak si ju otáčate v ruke. Stena kocky, ktorá je otočená k oknu sa vám bude zdať svetlejšia ako tá, ktorá je na strane od okna, pretože na prvú dopadá svetlo a druhá je v tieni. No a steny kociek nášho príkladu mali stále tú istú farbu, bez ohľadu na to, ktorým smerom boli otočené. Žiadne osvetľovanie sa nekonalo.

Čo sa osvetlenia týka, môžeme väčšinu starostí o správne zobrazenie objektov zveriť knižnici OpenGL. Treba ale ponastavovať množstvo vecí. A aby ste ich vedeli nastaviť správne, treba najprv povedať niekoľko slov k teórii.

Zdroje svetla sú rôzne, vydávajú rôzne druhy svetla a materiály na ne môžu tiež reagovať rôzne. Jednotlivé druhy svetla si opíšeme, aby sme vedeli, ako na ne budú naše objekty reagovať.

- Každý svetelný zdroj prispieva svojou troškou k **svetlu okolia** (ambient light). Svetlo okolia je svetlo, ktoré síce kedysi vyšlo z nejakého svetelného zdroja, ale medzitým sa už mnohokrát odrazilo a zmenilo smer, takže momentálne sa zdá, že prichádza zo všetkých smerov s rovnakou intenzitou. V miestnosti je väčšina svetla práve tohto pôvodu. Ak by niekde takéto svetlo nebolo, tie časti objektov, na ktoré by nedopadalo priame svetlo, by sa javili ako čierne. Ak by niekde bolo iba takéto svetlo, všetky steny jednofarebnej kocky by sa pozorovateľovi javili ako rovnaké a bolo by ťažké ich od seba rozoznať.
- Ak svetelný lúč dopadne na nejaký povrch, časť z neho sa pohltí – napríklad červené teleso odráža iba červené svetlo, všetky ostatné farebné zložky zožerie. Časť zo svetla sa odrazí. Matné telesá odrážajú svetlo všetkými smermi. Takéto svetlo sa volá **rozptýlené svetlo** (diffuse light). Napríklad stena kocky, ktorá je otočená k oknu, sa javí ako najjasnejšia. A v prípade, že sa kocka príliš neleskne, táto stena nemení farbu, keď sa na ňu pozeráme z rôznych strán. Rozptýlené svetlo je napríklad svetlo, ktoré prichádza z okna, keď slnko nesvieti priamo do miestnosti. Na objektoch je vidieť, že sú otočené ku oknu, ale ak máte v ruke zrkadielko, takýmto svetlom „prasiatko“ nevrhnete.
- Ak je ale povrch telesa lesklý a zdroj svetla dostatočne jasný, vtedy veľmi záleží na tom, kde sa práve pozorovateľ nachádza. Lesklým povrchom sa totiž dajú vrhať „prasiatka“. „Prasiatko“ je vrhnuté iba jedným smerom a ak niekto správnym smerom nestojí, bude sa mu síce zrkadielko javiť ako jasný obdĺžnik, ale nič ho oslepovať nebude. Svetlo, ktoré sa bude odrážať „prasiatkovským“ spôsobom sa nazýva **zrkadlové svetlo** (specular light). Je značnou zložkou svetla vyžiareného z lampy alebo zo slnka. (Tieto samozrejme vyžarujú aj ostatné druhy svetla.)[8]

Každý materiál môže reagovať na jednotlivé druhy svetla rôzne. Papier sa bude javiť ako jasný, keď ho otočíte ku svetlu – rozptýlené svetlo teda odráža dobre. Ale aj keby bol bodový zdroj akokoľvek jasný, papier zrkadlové svetlo odrážať nebude (nevrhá prasiatka). Vyleštená bronzová kľučka prasiatko síce vrhne, ale jeho farba nebude úplne biela – bude mať žltý odtieň. Z rozptýleného svetla aj svetla okolia bude kľučka odrážať iba zlatožltú farbu. Kapota auta s metalízou prasiatko možno vrhne, ale svetelný fľak, ktorý na nej bude pozorovateľ vidieť bude väčší a rozostrený. A fosforová hviezdička okrem toho, že môže rôzne druhy svetla odrážať, svieti aj vlastným svetlom. Z toho sa dá tušiť, že nastavenie materiálu bude trochu náročnejšie, ako keď sme

8 V blenderi je pri každom svetle možnosť vypnutia rozptýlenej a zrkadlovej zložky.

v predošlej lekcii iba nastavili farbu. Ale nie je to nijak zložité a keď človek vie, o čo ide, dosiahne oveľa lepšie efekty.

Takže poďme programovať. Do funkcie `InitGL` pridajte nasledujúci kus kódu (podľa možností ešte pred `return`):

```
GLfloat LightAmbient[]  = { 0.2f, 0.2f, 0.2f, 1.0f };
GLfloat LightDiffuse[]  = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightSpecular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightPosition[] = { 0.0f, 0.0f, 0.0f, 1.0f };

glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, LightSpecular);
glLightfv(GL_LIGHT1, GL_POSITION, LightPosition);
glEnable(GL_LIGHT1);

glEnable(GL_LIGHTING);
```

Na začiatku deklarujeme štyri polia a rovno ich naplníme hodnotami. Prvé tri opisujú farbu každej z troch opísaných zložiek svetla. Svetlo okolia je slabšie, rozptýlené aj zrkadlové svetlo svieti naplno. Posledné pole opisuje miesto, kde bude svetlo umiestnené. Môže sa zdať podivné, že každé pole neobsahuje tri, ale štyri čísla. V prípade farieb opisuje posledná zložka priesvitnosť (alfa kanál). V prípade súradníc sa štvrtá súradnica používa na niektoré triky pri počítaní transformácií. V prípade, že ju potrebujete použiť – ako napríklad v našej situácii, kedy funkcia `glLight` vo svojej verzii `glLightfv` očakáva pole štyroch premenných typu `float` (to `fv` znamená „float vector“), dajte tam jednotku.

Akékoľvek nastavenia svetelného zdroja sa dejú s pomocou funkcie `glLightfv`.[9] Prvý parameter označuje, ktoré svetlo sa ide nastavovať. Scéna môže v jednom momente obsahovať maximálne osem svetiel. Odvolávať sa na ne môžete s pomocou makier `GL_LIGHT1` až `GL_LIGHT8`. Druhý parameter hovorí, čo ideme nastavovať. Názvy makier v našej ukážke hovoria samy za seba. `GL_AMBIENT` nastavuje svetlo okolia, `GL_DIFFUSE` rozptýlené svetlo, `GL_SPECULAR` zrkadlové svetlo a `GL_POSITION` umiestnenie zdroja svetla.[10] Ako ďalší parameter (prípadne parametre) nasledujú samotné hodnoty, ktoré chceme nastaviť.

Keď je svetlo nastavené, môžete ho zapnúť. Robí sa to príkazom `glEnable`. Tým istým príkazom, len s parametrom `GL_LIGHTING` povieme OpenGL, aby venoval osvetľovaniu pozornosť. Jednotlivé svetlá aj celkové osvetlenie môžete podľa potreby zapínať a vypínať počas behu programu. Na vypínanie slúži príkaz `glDisable`.

Svetlo máme nastavené. Poďme sa teraz venovať samotnému vykresľovaniu. Budeme meniť iba funkciu `KresliSteny` z triedy `Kocka`. Prvá úprava bude, že odtiaľ vymažeme všetky nastavenia farieb s pomocou `glColor3f`. Keď sa ráta osvetlenie, toto nastavenie sa totiž ignoruje.

Potom spravíme nastavenie materiálu. Spraví sa to takto:

```
GLfloat mat_amb[] = { 0.0, 0.0, 0.3, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_amb);
GLfloat mat_diff[] = { 0.0, 0.0, 0.3, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diff);
GLfloat mat_spec[] = { 0.9, 0.9, 0.9, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_spec);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 127.0);
```

Znovu pre každý typ svetla nastavíme, ako sa k nemu kocka bude správať. Kocka bude lesklá

9 Alebo niektorej inej modifikácie funkcie `glLight`.

10 Je možné nastaviť niektoré ďalšie veci, napríklad to, že svetlo bude reflektor svietiaci určitým smerom s určitým uhlom záberu. Záujemcov odkazujeme na Red Book – klasickú učebnicu OpenGL od pánov Neidera, Davisa a Wooa zo Silicon Graphics. Je to na sieti a Google je kamarát.

a tmavomodrá, to znamená, že z rozptýleného svetla a zo svetla okolia bude odrážať iba tmavomodrú zložku, zrkadlové svetlo bude ale odrážať takmer úplne. Každý parameter materiálu sa nastavuje s pomocou niektorej z funkcií `glMaterial`. Prvý parameter funkcie (v našom prípade `GL_FRONT_AND_BACK`) hovorí, ktorej strany každej plôšky sa bude nastavenie týkať. Dajú sa spraviť materiály, v ktorých je predná strana plochy iná ako zadná. My to nastavujeme pre obe strany rovnako. Druhý parameter hovorí, čo v materiáli nastavujeme. Po ňom nasledujú konkrétne hodnoty, ktoré treba nastaviť.

Prvé tri nastavenia sú podobné, ako pri nastavovaní svetla. Makrá `GL_AMBIENT`, `GL_DIFFUSE` a `GL_SPECULAR` hovoria o tých istých zložkách svetla.[11] Posledné nastavenie je ale trocha odlišné. Vlastnosť `GL_SHININESS`, teda lesklosť hovorí, akú veľkú svetelnú škvrnu na ploche zrkadlové svetlo vyrobí. Prípustné hodnoty sú od 0.0 do 128.0, čím je číslo väčšie, tým je škvrna menšia a jasnejšia.[12]

Ďalšiu vec musíme zmeniť pri vykresľovaní samotných štvoruholníkov. Okrem toho, kde sú ich vrcholy je pri osvetlených objektoch dôležité poznať normálový vektor – to je vektor kolmý na danú plochu. Ten jednak určí, kde je predok a kde zadok plochy, jednak sa s jeho pomocou počíta natočenie plochy oproti zdroju svetla a teda množstvo odrazeného rozptýleného a zrkadlového svetla. Normálový vektor sa určuje s pomocou niektorej verzie funkcie `glNormal`. Samotné vykresľovanie teda bude vyzerať takto:

```
glBegin(GL_QUADS);
     // Predna stena
     glNormal3f( 0.0f,  0.0f, 1.0f);
     glVertex3f(-1.0f, -1.0f, 1.0f);
     glVertex3f(-1.0f,  1.0f, 1.0f);
     glVertex3f( 1.0f,  1.0f, 1.0f);
     glVertex3f( 1.0f, -1.0f, 1.0f);
     // Prava stena
     glNormal3f(1.0f,  0.0f,  0.0f);
     glVertex3f(1.0f, -1.0f, -1.0f);
     glVertex3f(1.0f, -1.0f,  1.0f);
     glVertex3f(1.0f,  1.0f,  1.0f);
     glVertex3f(1.0f,  1.0f, -1.0f);
     // Zadna stena
     glNormal3f( 0.0f,  0.0f, -1.0f);
     glVertex3f(-1.0f, -1.0f, -1.0f);
     glVertex3f(-1.0f,  1.0f, -1.0f);
     glVertex3f( 1.0f,  1.0f, -1.0f);
     glVertex3f( 1.0f, -1.0f, -1.0f);
     // Lava stena
     glNormal3f(-1.0f,  0.0f,  0.0f);
     glVertex3f(-1.0f, -1.0f, -1.0f);
     glVertex3f(-1.0f, -1.0f,  1.0f);
     glVertex3f(-1.0f,  1.0f,  1.0f);
     glVertex3f(-1.0f,  1.0f, -1.0f);
     // Horna stena
     glNormal3f( 0.0f, 1.0f,  0.0f);
     glVertex3f(-1.0f, 1.0f, -1.0f);
     glVertex3f(-1.0f, 1.0f,  1.0f);
     glVertex3f( 1.0f, 1.0f,  1.0f);
     glVertex3f( 1.0f, 1.0f, -1.0f);
     // Dolna stena
     glNormal3f( 0.0f, -1.0f,  0.0f);
     glVertex3f(-1.0f, -1.0f, -1.0f);
     glVertex3f(-1.0f, -1.0f,  1.0f);
     glVertex3f( 1.0f, -1.0f,  1.0f);
     glVertex3f( 1.0f, -1.0f, -1.0f);
glEnd();
```

11 V blenderi je farba `GL_DIFFUSE` farbou objektu a `GL_SPECULAR` farbou odlesku.

12 Čo sa týka ďalších vlastností, ktoré sa materiálu dajú nastaviť (napríklad vlastné vyžarovanie), opäť čitateľa odkazujeme na Red Book.

V tejto fáze je to hodné vyskúšania. Takže

Úloha č.1: Vyskúšajte.

Posledná vec, ktorou sa budeme v tejto lekcii zaoberať, je hmla. Nastavenie hmly je jednoduché, niečo budeme pridávať len do funkcie `InitGL`. A to niečo bude vyzerať takto:

```
GLfloat fogColor[]= {0.5f, 0.5f, 0.5f, 1.0f};
glClearColor(0.5f,0.5f,0.5f,1.0f);

glFogi(GL_FOG_MODE, GL_LINEAR);
glFogf(GL_FOG_START, 15.0f);
glFogf(GL_FOG_END, 25.0f);
glFogfv(GL_FOG_COLOR, fogColor);
glFogf(GL_FOG_DENSITY, 0.35f);
glHint(GL_FOG_HINT, GL_DONT_CARE);

glEnable(GL_FOG);
```

Do poľa si hodíme farbu pre hmlu (patrične šedivú) a farbu, ktorou sa obrázok maže nastavíme na rovnakú (nech nemáme zahmlené objekty na čiernom pozadí). Potom nastavíme spústu parametrov s pomocou rôznych odrôd funkcie `glFog` a hmlu zapneme volaním funkcie `glEnable(GL_FOG)`.

Prvý parameter `GL_FOG_MODE` sa týka typu hmly. Môže to byť buď najjednoduchšia `GL_LINEAR`, alebo reálnejšia, ale pomalšia `GL_EXP` alebo `GL_EXP2`. Ďalšie dva parametre `GL_FOG_START` a `GL_FOG_END` hovoria, v akej vzdialenosti od kamery hmla začína a končí. Pred začiatkom hmly vidno objekty rovnako, ako keby žiadna hmla nebola. Za koncom hmly sú objekty zahmlené úplne. Medzi týmito hranicami hmla postupne rastie. `GL_FOG_COLOR` nastaví farbu hmly, `GL_FOG_DENSITY` jej hustotu. Nastavovať hustotu má ale zmysel iba pri exponenciálnych hmlách. Pri lineárnej to bledne rovnomerne od začiatku do konca.

Funkcia `glHint` nastavuje niektoré detaily ohľadom správania OpenGL. Keď nastavujeme `GL_FOG_HINT`, môžeme ho nastaviť na `GL_NICEST` (vtedy to bude kresliť pekne, ale pomaly) alebo `GL_FASTEST` (vtedy to bude kresliť rýchlo, ale škaredšie). My sme to nastavili na `GL_DONT_CARE`. Vtedy to bude robiť nikto nevie čo. (Ešte, že sme to nastavili.)

Hmlu samozrejme možno kedykoľvek vypnúť – áno, hádate správne – s pomocou príkazu `glDisable(GL_FOG)`.

Úloha č.2: Vyskúšajte.

Úloha č.3: Urobte prepínanie medzi jednotlivými druhmi hmiel, nech ich môžete meniť počas behu programu a porovnať.

Úloha č.4: Vyskúšajte, ako to vyzerá, keď sa farba mazania pozadia nenastaví na šedú.

Ďalšie úlohy robte s vypnutou hmlou. (Stačí, ak riadok `glEnable(GL_FOG);` zakomentujete.)

Úloha č.5: Premiestnite zdroj svetla kdesi vľavo. Premiestnite zdroj svetla medzi kocky. Pozrite sa, ako sa kedy ktorá zložka svetla prejavuje. Skúste spraviť pohyblivý zdroj svetla.

Úloha č.6: Nastavte si biele pozadie a zmeňte materiál kociek tak, aby boli matné a oranžové.

4. lekcia
Textúry a transparentnosť alebo „Obrázky a vitráže“

Na začiatok sa na chvíľu vrátime k predošlým lekciám. V druhej lekcii bola jedna z úloh s pomocou funkcie `glScalef` kocky zväčšovať a zmenšovať. Niektorí z vás potom túto verziu programu používali na experimenty s materiálmi a osvetlením a zistili zaujímavú vec – keď sa kocky zmenšovali, ich farba bola čím ďalej, tým jasnejšia. A naopak, ak sa zväčšovali, tmavli. Efekt je to isto zaujímavý, ale ak človek pracne vyrobil materiál svojmu srdcu milý, často chce, aby tento materiál bol stále rovnaký bez ohľadu na to, aký veľký je objekt, na ktorý ho chce aplikovať.

Dôvod, prečo kocky menia farbu je jednoduchý – ako sme hovorili minule, pri počítaní osvetlenia hrá dôležitú rolu normálový vektor. A nie len jeho smer, ale aj jeho dĺžka. A transformácia `glScalef` spôsobí, že sa budú meniť nie iba súradnice vrcholov, ale sa budú zväčšovať a zmenšovať aj normálové vektory. Aby sme tomu zabránili, môžeme OpenGL povedať, že sa má na každý normálový vektor pozerať tak, ako keby mal dĺžku jedna. Spravíme to tak, že zavoláme funkciu `glEnable(GL_NORMALIZE)`. Potom sa už materiál zväčšovaním a zmenšovaním meniť nebude.

A teraz už k téme dnešnej lekcie. Textúry sú rastrové obrázky, ktoré môžeme použiť ako poťah na polygóny, ktoré vykresľujeme. Ich použitie často mnohé veci zjednoduší a dodá scéne na reálnosti. Napríklad sochu v pozadí chrámu sa môžeme pokúsiť vymodelovať z jednotlivých polygónov, ale na dosiahnutie rovnakého (alebo často aj lepšieho) efektu stačí otextúrovať patričným obrázkom obdĺžnik.

Na textúry, ktoré vie OpenGL použiť, platia isté obmedzenia: Prvé je, že ich rozmery musia byť mocniny dvojky, pričom výška a šírka nemusia byť rovnaké. Druhé je, že sa to musí vmestiť do pamäte. Totiž, ak použijete textúru s rozmermi 256 × 256 pixelov, ktorá má na každú farbu aj na alfa kanál (priesvitnosť) rezervovaný jeden bajt, zaberie to 256 × 256 × 4 bajtov = 256 kb. Aj keď pamäť súčasných grafických kariet býva značná, netreba to s množstvom a veľkosťou textúr preháňať.

Aby sme textúru mohli použiť, musíme ju najprv načítať. Všetky načítané textúry si OpenGL čísluje – každej priradí číslo, s pomocou ktorého sa potom na textúru, ktorú práve chceme použiť odvolávame. Na uchovanie tohto čísla zriadime objektu `Kocka` premennú `GLuint textura`. (`GLuint` je iný názov pre `unsigned int`.) Ďalej si zriadime premennú `mame_texturu` typu `bool`, ktorá nám povie, či sa podarilo textúru úspešne načítať. Okrem toho si ešte zriadime funkciu `NacitajTexturu`, ktorá prevedie samotné načítanie textúry. Nová verzia súboru `kocka.h` bude teda vyzerať takto:

```
#ifndef _KOCKA_H_
#define _KOCKA_H_

class Kocka
{
public:
     Kocka(GLfloat nx = 0.0f, GLfloat ny = 0.0f, GLfloat nz = 0.0f);
     void Nastav(GLfloat nx = 0.0f, GLfloat ny = 0.0f, GLfloat nz = 0.0f);
     bool Nakresli();
protected:
     void KresliSteny();
     bool NacitajTexturu();
     void Pootoc();
     GLfloat x,y,z;        // suradnice
     GLfloat xrot, yrot;   // natocenie v smere osi x a y
     GLfloat dxr, dyr;     // rychlost otacania okolo x a y
     bool mameTexturu;
```

```
        GLuint textura;
};

#endif
```

Metóda `NacitajTexturu` môže zas vyzerať takto (aby fungovala správne, musí byť samozrejme v adresári so spustiteľným programom súbor `hviezdicka.tga`):

```
bool Kocka::NacitajTexturu()
{
        BITMAP* b = load_bitmap("hviezdicka.tga",NULL);
        if (b == NULL)
                return false;

        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
        textura = allegro_gl_make_texture(b);

        destroy_bitmap(b);
        return true;
}
```

Najprv načítame bitmapu zo súboru bežným spôsobom, na aký sme zvyknutí z Allegra. Skontrolujeme, či sa podarilo načítať ju a ak nie, oznámime neúspech.

V ďalších dvoch riadkoch nastavíme, ako sa bude naša textúra správať. Ako sme už spomínali, textúra je rastrový obrázok a skladá sa z pixelov. A keď textúru natiahneme na polygón, ktorý je väčší, než ona, môže sa stať viacero vecí. Pre každý bod povrchu sa môže zvoliť farba najbližšieho pixelu textúry, ktorý mu zodpovedá. (Túto možnosť sme nastavili, keď sme použili hodnotu `GL_NEAREST`.) To je najjednoduchšia a najrýchlejšia metóda, má ale tú nevýhodu, že na na veľkom polygóne pri pohľade zblízka veľmi vidieť raster. Druhá možnosť je použiť namiesto `GL_NEAREST` metódu `GL_LINEAR`. Tá na výpočet farby použije lineárnu aproximáciu. Výsledok je krajší, ale trvá to dlhšie. Pri zmenšovaní máme dokonca ešte viac možností.[13] Pred vytvorením textúry je vhodné nastaviť, ako sa bude textúra pri zväčšovaní (`GL_TEXTURE_MAG_FILTER`) a pri zmenšovaní (`GL_TEXTURE_MIN_FILTER`) správať.

Keď všetko nastavíme, môžeme textúru vytvoriť. Slúži na to funkcia `allegro_gl_make_texture` alebo nejaká jej odroda. Táto zavolá patričné funkcie z OpenGL a vráti číslo vygenerovanej textúry. Keď už je textúra hotová, môžeme bitmapu vymazať, nech nezavadzia v pamäti.

Funkciu `NacitajTexturu` zavoláme z konštruktora – pridáme do neho riadok

```
mameTexturu = NacitajTexturu();
```

Momentálne sme si ale spôsobili trochu problémy. Totiž – v súbore s hlavným programom deklarujeme pole kociek pred všetkými funkciami, aby k nemu mali všetky funkcie prístup. Pre každú kocku sa zavolá konštruktor. Konštruktor spustí funkciu `NacitajTexturu` a tá sa pokúsi spustiť funkciu Allegra `load_bitmap`. A to ešte predtým, ako bolo Allegro inicializované. Program to nepredýcha a padne.

Situácia sa dá riešiť iba tak, že pole kociek deklarujeme vo vnútri funkcie `main` až potom, čo sme správne zinicializovali Allegro. To ale prinesie tú nevýhodu, že funkcia `DrawGLScene` prestane mať ku kockám prístup. Preto jej pole kociek odovzdáme ako parameter. Funkciu teda zmeníme takto:

```
bool DrawGLScene(Kocka* k)
```

Funkcia dostane ako parameter smerník, ktorý môže ukazovať na prvý člen poľa kociek.

13 Spomeňme napríklad `GL_LINEAR_MIPMAP_NEAREST` kde sa vygeneruje niekoľko zmenšených textúr a použije sa najvhodnejšia.

Vo funkcii `main` zmeníme riadok, z ktorého sme volali vykresľovanie takto:

```
while (DrawGLScene(k))
```

Odovzdáme tak funkcii `DrawGLScene` smerník na naše pole kociek a tá ich môže správne vykresliť.

Ako bude vyzerať samotné vykresľovanie? Takto:

```
void Kocka::KresliSteny()
{
     if (!mameTexturu)
     {
          GLfloat mat_amb[] = { 0.0, 0.0, 0.3, 1.0 };
          glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, mat_amb);
          GLfloat mat_diff[] = { 0.0, 0.0, 0.3, 1.0 };
          glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diff);
          GLfloat mat_spec[] = { 0.9, 0.9, 0.9, 1.0 };
          glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_spec);
          glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,127.0);
     }
     else
     {
          glBindTexture(GL_TEXTURE_2D, textura);
     }


     glBegin(GL_QUADS);
          // Predna stena
          glNormal3f( 0.0f,  0.0f, 1.0f);
          glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
          glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, 1.0f);
          glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, 1.0f);
          glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
          // Prava stena
          glNormal3f(1.0f,  0.0f,  0.0f);
          glTexCoord2f(0.0f, 0.0f); glVertex3f(1.0f, -1.0f, -1.0f);
          glTexCoord2f(0.0f, 1.0f); glVertex3f(1.0f, -1.0f,  1.0f);
          glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f,  1.0f,  1.0f);
          glTexCoord2f(1.0f, 0.0f); glVertex3f(1.0f,  1.0f, -1.0f);
          // Zadna stena
          glNormal3f( 0.0f,  0.0f, -1.0f);
          glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
          glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
          glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
          glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
          // Lava stena
          glNormal3f(-1.0f,  0.0f,  0.0f);
          glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
          glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
          glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
          glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
          // Horna stena
          glNormal3f( 0.0f, 1.0f,  0.0f);
          glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
          glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f,  1.0f);
          glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f,  1.0f);
          glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
          // Dolna stena
          glNormal3f( 0.0f, -1.0f,  0.0f);
          glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
          glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
          glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
          glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
     glEnd();
}
```

Zmeny oproti pôvodnému stavu sú dve. Prvá je tá, že na začiatku skontrolujeme, či sa podarilo načítať textúru. Ak nie, nastavíme materiál rovnako, ako v tretej lekcii. Ak sa to podarilo,

tak príkazom `glBindTexture` nastavíme, ktorú textúru ideme používať.

Druhá zmena je v tom, že pred každým volaním funkcie `glVetrex` voláme funkciu `glTexCoord`, ktorá určí, kde v textúre sa má daný bod nachádzať, pričom `[0,0]` znamená ľavý dolný roh textúry a `[1,1]` pravý horný roh textúry. Mimochodom – nie je nutné zadávať súradnice od 0 do 1. Keby sme všade vo funkcii `glTexCoord` namiesto `1.0f` použili hodnotu `2.0f`, textúra by sa na každej stene kocky dvakrát zopakovala.

Dobre. Teraz už stačí len pridať volanie funkcie `glEnable(GL_TEXTURE_2D)` do funkcie `InitGL` aby si OpenGL uvedomilo, že má používať textúry a môžete to vyskúšať.

Úloha č.1: Vyskúšajte.

Kód v tej podobe, v ktorej sme ho napísali, má jednu výraznú slabinu. Každá kocka si vygeneruje svoju vlastnú textúru. Všetky tie textúry sú ale úplne rovnaké a zbytočne zaberajú pamäť. Stačilo by vygenerovať jednu. Cesta k náprave vedie cez jednu sympatickú črtu C++ a to statické atribúty. Totiž, keď v deklarácii triedy `Kocka` napíšeme `GLuint textura;` zriadi sa premenná `textura` pre každý objekt triedy `Kocka`. Keď tam ale napíšeme `static GLuint textura;` tiež sa vyrobí premenná `textura`, ale bude iba jedna – spoločná pre všetky objekty triedy `Kocka`. A to je presne to, čo potrebujeme. Takže v hlavičkovom súbore upravíme patričné dva riadky na nasledujúcu podobu:

```
static bool mameTexturu;
static GLuint textura;
```

Premenné musíme inicializovať na počiatočné hodnoty. Otázka je, kde sa to má robiť. Keby sme to spravili v konštruktore, inicializovali by sa nanovo pri každom vzniku nového objektu a to nechceme. Statické premenné sa inicializujú podivne – kdekoľvek do súboru `kocka.cpp` ale tak, aby to nebolo vo vnútri žiadnej funkcie, pridajte tieto dva riadky:

```
bool Kocka::mameTexturu = false;
GLuint Kocka::textura = 0;
```

Vyzerá to ako nová deklarácia – ale nie je. Je to inicializácia statickej premennej. A kompilátor to pochopí.

Ešte treba zabezpečiť, aby sa metóda `NacitajTexturu` volala iba raz – vtedy keď sa vytvára prvý objekt triedy `Kocka`. To sa spraví jednoducho. Volanie z konštruktora zmeníme na

```
if (!mameTexturu)
     mameTexturu = NacitajTexturu();
```

Metóda `NacitajTexturu` sa bude volať iba vtedy, ak zatiaľ žiadna textúra nebola vygenerovaná.

Úloha č.2: Vyskúšajte.

Vieme teda otextúrovať kocku a vďaka finte so Z-bufferom vieme spôsobiť, aby boli vidieť vždy iba tie veci, ktoré nič nezakrýva. Niekedy to však nie je vec, po ktorej túžime. Niekedy naopak potrebujeme, aby bolo vidieť nejakú vec a okrem nej ešte nejakú inú vec, čo je za ňou. A to vtedy, ak tá vec, ktorá je vpredu, je priesvitná. Priesvitnosť sa v OpenGL rieši viacerými spôsobmi. Začneme tým jednoduchším.

Variant bez osvetlenia: V tomto prípade nebudeme používať osvetlenie. (Aj bez neho sa dajú spraviť mnohé zaujímavé efekty.) Vypnite preto svetlo zakomentovaním patričného riadku. Okrem toho, keďže ideme vykresľovať všetky steny, vypnite Z-buffer (zakomentujte zapnutie `GL_DEPTH_TEST`).

Používa sa tzv. blending. Po slovensky zmixovanie. Teda to, čo ideme nakresliť, zmixujeme s tým, čo už nakreslené je. Ako všetko, aj blending musíme v OpenGL zapnúť. A nečakane sa to

urobí príkazom `glEnable(GL_BLEND)`. (Môžete si ho pridať do `InitGL`.) Pointa je v tom, že ak je vypnuté počítanie osvetlenia, textúra sa dá ovplyvniť nastavenou farbou. Každý pixel textúry sa nastavenou farbou vynásobí. Násobí sa po zložkách, hodnoty sú od 0 do 1. Toto sa samozrejme dá využiť na dodatočnú farebnú úpravu textúry, ale hlavná výhoda je inde – tieto úpravy sa týkajú aj alfa kanálu, ktorý hovorí o tom, ako je ktorý pixel priesvitný. (Hodnotu má od nula do jedna. Nula znamená úplne priesvitný a neviditeľný, jedna znamená, že nie je vidieť nič za ním.) Takže ak nastavím farbu na `glColor4f(1.0f, 1.0f, 1.0f, 0.5f)`, farby textúry ostanú nezmenené, ale celá textúra sa stane priesvitnou s alfou 0,5. No a priesvitná textúra sa s pomocou už spomínaného blendingu dá zmixovať s pozadím. Na to ale treba ešte povedať, akým spôsobom sa to mixovať bude. Slúži na to funkcia `glBlendFunc`. V našom prípade použijeme variant `glBlendFunc(GL_SRC_ALPHA,GL_ONE)` čo znamená, že „farbu toho, čo ideš kresliť vynásob alfou (v našom prípade 0,5) a pripočítaj k tomu, čo už je nakreslené, vynásobenému jednotkou“.

Takže úplne najjednoduchšie riešená priesvitnosť – vypnite svetlo a Z-buffer, nastavte farbu s alfou a mixovaciu funkciu a priesvitnosť je na svete.

Úloha č.3: Vyskúšajte.

Variant s osvetlením: Problém je, že v momente, keď zapnete svetlo, textúra prestane byť ovplyvňovaná nastavenou farbou. Vtedy sa dá použiť metóda o kúsok silnejšia – textúry, ktoré priamo obsahujú alfa kanál.

Každý slušný bitmapový editor (aspoň teda Gimp určite) vie vyrobiť obrázok, ktorého časti sú priesvitné, dokonca rôzne časti rôzne priesvitné. Takýto obrázok môžete uložiť vo formáte `tga`, ktorý si také veci ako priesvitnosť vie zapamätať. A Allegro šťastnou náhodou formát `tga` podporuje. To znamená, že pri troške šikovnosti vieme takýto obrázok preonačiť na textúru, ktorá v sebe obsahuje aj alfa kanál a takúto textúru zmixovať s tým, čo už je nakreslené. Pri tom preonačovaní ale treba dať na niektoré veci pozor. Takže uveďte svoj projekt do toho stavu, v ktorom ste ho mali predtým, kým sme sa začali hrať s priesvitnosťou, vyskúšajte si, či sa vám to naozaj podarilo a ideme experimentovať.

Pokúsime sa do popredia našej scény vykresliť okennú mrežu (obrázok `mreza.tga`). Vzhľadom na to, že mreža sa kociek netýka, všetky úpravy budeme robiť v tom súbore, v ktorom máme `main` (u mňa `04.cpp`).

Najprv si deklarujeme premennú `GLuint mreza` (niekde pred funkciami, aby bola globálna) a urobíme funkciu, ktorá nám načíta textúru. Môže vyzerať napríklad takto:

```
bool NacitajTexturu()
{
    set_color_conversion(COLORCONV_NONE | COLORCONV_KEEP_TRANS);
    BITMAP* b = load_bitmap("mreza.tga",NULL);
    if (b == NULL)
        return false;

    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
    mreza = allegro_gl_make_texture_ex(AGL_TEXTURE_HAS_ALPHA,b,GL_RGBA);

    destroy_bitmap(b);
    return true;
}
```

Od podobnej funkcie, ktorú sme vytvorili pre triedu `Kocka` sa líši v dvoch drobných, zato však podstatných detailoch. Prvý je použitie funkcie `set_color_conversion`. Totiž – Allegro si pri načítavaní zo súboru bitmapu skonvertuje do formátu, ktorý práve používa. Keď máme nastavenú šestnásťbitovú farebnú hĺbku, aj načítanú bitmapu skreše na požadovaných šestnásť bitov. V tomto prípade je to však problém, pretože pri tom spoľahlivo oreže práve alfa kanál, ktorý je pre

nás podstatný. Preto mu parametrami funkcie `set_color_conversion` povieme „na nič nesiahaj, nič nekonvertuj a zvlášť si daj záležať na tom, aby si nezahodil alfa kanál“.

Druhý rozdiel je v použití funkcie `allegro_gl_make_texture_ex`. Funguje rovnako, ako `allegro_gl_make_texture`, ale máme šancu zdôrazniť jej, že tá bitmapa má alfakanál a že by sme ho radi videli aj vo výslednej textúre.

No a teraz, keď je textúra načítaná, do funkcie `DrawGLScene` za vykresľovanie kociek a pred výmenu stránok môžeme pridať tento kód:

```
glBindTexture(GL_TEXTURE_2D, mreza);
glLoadIdentity();

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
glBegin(GL_QUADS);
     glNormal3f( 0.0f,  0.0f, 1.0f);
     glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
     glTexCoord2f(0.0f, 10.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
     glTexCoord2f(10.0f, 10.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
     glTexCoord2f(10.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glEnd();
glDisable(GL_BLEND);
```

V ňom najprv povieme, že sa ideme hrať s textúrou `mreza`. Potom zrušíme všetky predošlé transformácie. Zapneme blending a funkciu nastavíme spôsobom „ak mám alfu nula, nenakresli zo mňa nič, ak jedna, všetko podo mnou vymaž (a medzitým to bude tiež dobre)“. Potom tesne pred kameru nakreslíme nahusto otextúrovaný štvorec. Na záver blending zas vypneme. Ak nezabudneme zavolať funkciu `NacitajTexturu` (napríklad z `InitGL`), malo by to fungovať.

Úloha č.4: Vyskúšajte. (Nejako sa tie úlohy opakujú.)

Úloha č.5: Načítajte si pre kocku dve textúry z toho istého obrázku, akurát, že jedna bude používať `GL_NEAREST` a druhá `GL_LINEAR`. Spravte prepínanie s pomocou klávesnice medzi oboma textúrami. Umiestnite jednu kocku tak, aby bolo dobre vidieť rozdiel.

Úloha č.6: Čo sa stane, ak najprv vykreslíme mrežu a až potom kocky?

Úloha č.7: Skúste si urobiť a použiť takú textúru, aby na stenách kocky boli iba hviezdičky a zvyšok bol priesvitný.

5. lekcia

Zoznamy inštrukcii a import z Blenderu alebo „Nie je všetko len kocka.“

Kocka je sympatické teleso. Jediné teleso, ktoré sa kedy v našich príkladoch vyskytovalo (ak vyhlásime, že ten trojuholník z prvej lekcie nebol teleso, ale rovinný útvar). Ale sú aj iné telesá. A niekedy snaha o to, aby scéna pôsobila čo najrealistickejšie vyžaduje, aby to neboli len kocky. Táto lekcia bude o tom, ako OpenGL primäť k tomu, aby zobrazilo aj niečo zložitejšie.

Najprv spomenieme jednu užitočnú fintu a to zoznam inštrukcii (po anglicky display list). Totiž pri práci s OpenGL sa často stane, že z trojuholníkov či štvorčekov vytvárate nejaké zložité teleso a popri tom robíte množstvo výpočtov či iných operácií. A robiť tieto výpočty zakaždým, keď vykresľujete scénu, je často veľmi neefektívne. Preto je v OpenGL zabudovaný mechanizmus, ktorým môžete povedať „teraz začni nahrávať“ a všetky príkazy knižnice OpenGL (ale iné nie!!!) sa budú ukladať do pamäte. A keď nahrávanie skončíte, môžete kedykoľvek povedať „teraz spusti tento zoznam inštrukcii“ a OpenGL vykoná – bez potreby opätovného počítania – všetky nahraté operácie.

Funguje to takto:

```
glNewList(1, GL_COMPILE);
/* Tu vykonavame nejake OpenGL a/alebo ine instrukcie */
glEndList();
```

Zoznam začína funkciou `glNewList`. Prvý parameter je identifikátor zoznamu. Práve vytvorený zoznam v našom príklade bude mať číslo 1. Druhý parameter je buď `GL_COMPILE`, ak chceme OpenGL príkazy iba zaznamenávať, alebo `GL_COMPILE_AND_EXECUTE`, ak ich chceme hneď aj vykonať. Zaznamenávanie inštrukcií do nášho zoznamu ukončíme príkazom `glEndList()`.

Keď chceme zoznam s číslom 1 spustiť, použijeme príkaz

```
glCallList(1);
```

Pri väčších projektoch môže samozrejme takýchto zoznamov existovať mnoho. A aby v číslovaní nebol chaos a aby sa náhodou nestalo, že pod tým istým číslom nahráme viacero zoznamov, poskytuje OpenGL funkciu, ktorá v tom udržiava poriadok – funkciu `glGenLists`. Funkcii ako parameter zadáme, koľko zoznamov ideme generovať a ona nám vyhradí patričný interval celých čísel. Pri jej ďalšom volaní už bude vedieť, že tento interval je obsadený a vráti iné čísla. Takže kompletné vytváranie zoznamu môže vyzerať približne takto:

```
GLuint mojeTeleso = glGenLists(1);
glNewList(mojeTeleso, GL_COMPILE);
/* Blabla... */
glEndList();
```

a jeho volanie

```
glCallList(mojeTeleso);
```

Môžeme si teda vytvoriť triedu `Teleso`, ktorá bude v podstate rovnaká, ako trieda `Kocka` z druhej lekcie až na to, že jej objekty môžu nadobúdať ľubovoľný tvar, ktorý jej predhodíme ako zoznam inštrukcií. Jej hlavičkový súbor bude vyzerať takto:

```
#ifndef _TELESO_H_
#define _TELESO_H_
```

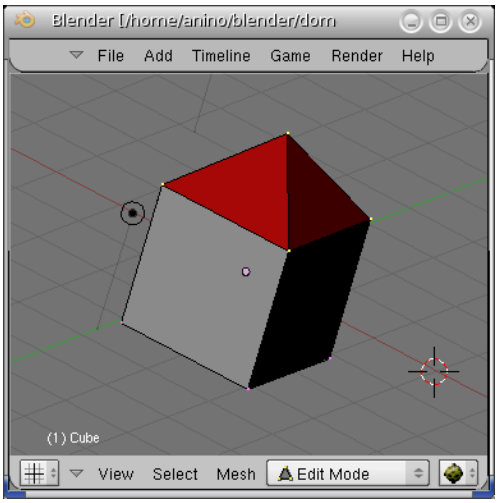
```
class Teleso
{
public:
     Teleso(GLuint displaylist = 0,
            GLfloat nx = 0.0f,
            GLfloat ny = 0.0f,
            GLfloat nz = 0.0f);
     void Nastav(GLuint displaylist = 0,
                 GLfloat nx = 0.0f,
                 GLfloat ny = 0.0f,
                 GLfloat nz = 0.0f);
     bool Nakresli();
protected:
     void Pootoc();
     GLfloat x,y,z;         // suradnice
     GLfloat xrot, yrot;    // natocenie v smere osi x a y
     GLfloat dxr, dyr;      // rychlost otacania okolo x a y
     GLuint displaylist;
};

#endif
```

Od triedy `Kocka` sa líši tým, že je mu možné nastavovať v konštruktore a vo funkcii `Nastav` zoznam inštrukcii, má premennú, do ktorej si to číslo uloží a chýba mu metóda `KresliSteny`. Výpis súboru `teleso.cpp` tu uviesť netreba, čitateľ si ľahko všetko potrebné domyslí. Zmeny oproti súboru `kocka.cpp` sú iba v tom, že do konštruktora a do funkcie `Nastav` treba pridať nastavovanie premennej `displaylist`, nebude sa tam nachádzať funkcia `KresliSteny` a namiesto jej volania do funkcie `Nakresli` vložíme kód

```
if (displaylist != 0)
     glCallList(displaylist);
```

(Okrem toho je tam samozrejme ešte jedna drobná zmena – trieda sa volá `Teleso` a nie `Kocka`.)

No dobre. Zoznamy inštrukcií používať vieme, aj náš objekt už máme prerobený podľa nich, poďme sa pozrieť, ako to vyzerá s tým exportom z Blenderu. Začneme niečim jednoduchým – domčekom, ktorý pozostáva z piatich štvorcov (dlážka a steny) a štyroch trojuholníkov (strecha). Farba stien je biela a farba strechy je červená.



Ak takýto domček uložíme ako `domcek.blend` a pokúsime sa ho otvoriť v editore, zistíme, že je v ňom množstvo binárneho balastu, ktorý nevieme čítať. Ono to určite nejako ide – nakoniec samotný Blender to nejako robiť musí, ale skúsime radšej niečo jednoduchšie.

V kolónke `File` zvolíme možnosť `Export`, z ponúknutých formátov vyberieme `Videoscape...` a uložíme. Tento exportný filter neukladá celú scénu, ale iba aktuálny objekt. Dajte preto pozor, aby bol v čase exportovania vybratý ten objekt, ktorý chcete exportovať a aby nebol (na rozdiel od nášho obrázku) v editovacom režime. Výsledný súbor má koncovku `.obj`. Poďme sa pozrieť, čo sa v ňom nachádza:

```
3DG1
9
1.000000 1.000000 -1.000000
1.000000 -1.000000 -1.000000
-1.000000 -1.000000 -1.000000
-1.000000 1.000000 -1.000000
```

```
1.000000 0.999999 1.000000
0.999999 -1.000001 1.000000
-1.000000 -1.000000 1.000000
-1.000000 1.000000 1.000000
0.000000 -0.000000 2.000000
4 0 1 2 3 0xffffff
4 0 4 5 1 0xffffff
4 1 5 6 2 0xffffff
4 2 6 7 3 0xffffff
4 4 0 3 7 0xffffff
3 7 6 8 0xff
3 6 5 8 0xff
3 4 7 8 0xff
3 5 4 8 0xff
```

Na začiatku je „čarodejné slovko“ `3DG1`. Súbory formátu Videoscape totiž okrem telies vedia ukladať aj svetlá a krivky a aj pre telesá majú dva rôzne formáty. Tento reťazec nás ubezpečí, že súbor je v tom správnom formáte. Potom nasleduje počet vrcholov (vrcholy kocky a špička strechy – dokopy deväť). Ďalších deväť riadkov popisuje súradnice jednotlivých vrcholov. Potom až do konca súboru sú popisované steny. Prvé číslo hovorí, či sa jedná o trojuholník, alebo o štvoruholník. Za ním nasleduje zoznam troch alebo štyroch čísel, ktoré hovoria o tom, ktoré vrcholy danú stenu tvoria (vrcholy sú číslované od nuly). A nakoniec príde farba steny.

Ako vidíte, tento formát je pomerne chudobný. Niet v ňom napríklad ani stopy po použitých textúrach. A pre jednotlivé steny ani nie sú zadané normálové vektory. Dokonca aj farby sú uvedené zvláštne. Červená je `0xff` namiesto toho, aby bola `0xff0000` – poradie farieb je BGR namiesto RGB.

S niektorými detailami ohľadom formátu si neporadíme. Textúry tento formát skrátka nespravuje. Ale niektoré veci sa vyriešiť dajú. Napríklad normálové vektory, bez ktorých žiadne rozumne osvetlené teleso nespravíme, sa dajú zrátať s pomocou vektorového súčinu. Vektorový súčin je finta, ktorá nám z dvoch vektorov v 3D vyrobí vektor, ktorý je kolmý na obidva. Ráta sa to tak, že ak máme vektory (x_1, y_1, z_1) a (x_2, y_2, z_2) tak ich vektorový súčin je vektor

$$\left(\begin{vmatrix} y_1 & z_1 \\ y_2 & z_2 \end{vmatrix}, \begin{vmatrix} z_1 & x_1 \\ z_2 & x_2 \end{vmatrix}, \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} \right)$$

Ak neviete, čo sú tie tabuľky s rovnými čiarami na bokoch (sú to determinanty), nezúfajte. Podstatné je, že sa to ráta takto: $x_n = y_1*z_2-y_2*z_1$, $y_n = z_1*x_2-z_2*x_1$, $z_n = x_1*y_2-x_2*y_1$. Spravíme si teda funkciu, ktorá na vstupe dostane smerník na vektor, do ktorého má uložiť výsledok a súradnice dvoch vektorov, ku ktorým má vypočítať kolmý. Funkcia bude vyzerať takto:

```
void NormalovyVektor(GLfloat* vektor,
                     GLfloat x1, GLfloat y1, GLfloat z1,
                     GLfloat x2, GLfloat y2, GLfloat z2)
{
     vektor[0] = y1 * z2 - y2 * z1;
     vektor[1] = z1 * x2 - z2 * x1;
     vektor[2] = x1 * y2 - x2 * y1;
}
```

Dobre. Všetko dôležité je pripravené, môžeme sa pustiť do výroby funkcie na čítanie súboru. Keďže v nej budeme používať reťazce a prúdy (spôsob, akým C++ číta súbory), nezabudneme na začiatok súboru pridať

```
#include <string>
#include <iostream>
#include <fstream>
```

Funkcia sa bude volať `ObjToDisplist`. Ako vstup dostane meno súboru, ktorý má prečítať, ako výstup dá číslo zoznamu inštrukcii, ktorý teleso nakreslí, alebo 0, ak niečo nebude fungovať. Budeme sa priebežne snažiť robiť kontrolu, či sa načítalo všetko, čo sa malo. Funkcia to

bude pomerne dlhá, takže budeme jej časti priebežne komentovať.

```
GLuint ObjToDisplist(string filename)
{
     ifstream vstup(filename.c_str());
     if (!vstup.is_open())
     {
          cerr << "Nepodarilo sa otvorit subor " << filename << endl;
          return 0;
     }
```

Na začiatku sme si vytvorili objekt triedy `ifstream` ktorá reprezentuje súbor otvorený na čítanie. Keďže konštruktor tejto triedy nechce na vstupe reťazec z C++ ale klasický C-čkový reťazec, zavolali sme metódu `c_str`, ktorá z C++-kového reťazca C-čkový reťazec vyrobila. Konštruktor súbor rovno aj otvorí. Ak sa otvorenie nepodarí, do chybového prúdu `cerr` (čo väčšinou znamená na konzolu) vypíšeme chybovú hlášku a vrátime 0. Všimnite si použitie výstupného operátora `<<` pri výstupe.

```
     string s;
     vstup >> s;
     if (s != "3DG1" || !vstup.good())
     {
          cerr << "Subor " << filename
               << " nema spravny format. :(" << endl;
          vstup.close();
          return 0;
     }
```

Súbor sme šťastne otvorili, môžeme začať čítať. Na jeho začiatku by sa malo nachádzať slovo `3DG1`. Deklarujeme si teda premennú typu `string`, načítame do nej hodnotu a skontrolujeme, či čítanie nezlyhalo (napríklad preto, že v súbore je niečo, čo sa nedá napchať do reťazca)[14], alebo či je tam naozaj to `3DG1`. Ak to tam nie je, vypíšeme informáciu, že ten súbor je nejaký divný, zavrieme ho a vrátime nulu.

```
     int pocetVrcholov;
     vstup >> pocetVrcholov;
     if (!vstup.good())
     {
          cerr << "Subor " << filename << " má poskodene data." << endl;
          vstup.close();
          return 0;
     }
```

Načítame počet vrcholov. Opäť skontrolujeme, či sa načítanie podarilo a či tam niekto namiesto čísla nepodhodil nejaké písmenká.

```
     GLfloat* vrchol_x = new GLfloat[pocetVrcholov];
     GLfloat* vrchol_y = new GLfloat[pocetVrcholov];
     GLfloat* vrchol_z = new GLfloat[pocetVrcholov];

     for(int i = 0; i < pocetVrcholov; i++ )
     {
          vstup >> vrchol_x[i] >> vrchol_y[i] >> vrchol_z[i];
          if (!vstup.good())
          {
               cerr << "Subor " << filename
                    << " má poskodene data." << endl;
               vstup.close();
               return 0;
          }
```

14 Na zistenie toho, že je všetko v poriadku slúži metóda `good`. Tá vráti `true`, ak je všetko v poriadku, všetky čítania sa podarili, súbor nie je na konci a ani nič podobné nekalé sa nestalo. Trieda `ifstream` má aj metódu `bad`, tá však nie je negáciou `good`. Tá je `true` iba keď dôjde k istým špecifickým chybám a nie je `true` napríklad vtedy, keď je súbor prečítaný až do konca.

```
}
```

Dynamicky alokujeme tri polia správnej veľkosti (jedno pre každú súradnicu) a načítame do nich súradnice vrcholov. Ak sa niečo pokazí, funkciu ukončíme.

```
GLuint teleso = glGenLists(1);
glNewList(teleso,GL_COMPILE);
```

Ak sme šťastne načítali vrcholy a dostali sa až sem, môžeme začať vytvárať zoznam inštrukcií. Necháme si priradiť nové číslo zoznamu a začneme s načítavaním OpenGL príkazov.

```
while (1)
{
     uint pocet, a, b, c, d, farba;
```

Steny budeme načítavať v nekonečnej slučke až dovtedy, kým sa niečo nepokazí alebo kým neprídeme na koniec súboru. Deklarujeme si premenné, do ktorých vieme načítať jeden riadok popisujúci stenu.

```
vstup >> pocet;
if (!vstup.good()) break;

if (pocet != 3 && pocet != 4)
{
     cerr << "Subor " << filename << " obsahuje "
          << pocet << "-uholnik." << endl;
     break;
}
```

Najprv načítame počet vrcholov. Ak už to nešlo, príkazom `break` vyskočíme z načítavacieho cyklu. Ak je počet vrcholov iný, než tri alebo štyri, podáme chybovú správu a tiež ukončíme cyklus.

```
if (pocet == 3)
     vstup >> a >> b >> c >> hex >> farba >> dec;
else
     vstup >> a >> b >> c >> d >> hex >> farba >> dec;

if (!vstup.good()) break;
```

Podľa počtu vrcholov načítame ostatné hodnoty z riadku. Predtým, než načítame hodnotu farby sa príkazom `>> hex` prepneme do šestnástkovej sústavy. Po načítaní sa opäť prepneme do desiatkovej príkazom `>> dec`. Znovu skontrolujeme, či sa všetko dobre načítalo.

```
GLfloat vektor[3];
NormalovyVektor(vektor,
                vrchol_x[b] - vrchol_x[a],
                vrchol_y[b] - vrchol_y[a],
                vrchol_z[b] - vrchol_z[a],
                vrchol_x[c] - vrchol_x[a],
                vrchol_y[c] - vrchol_y[a],
                vrchol_z[c] - vrchol_z[a]);
glNormal3f(vektor[0],vektor[1],vektor[2]);
```

Zoberieme vektor, ktorý ide z prvého do druhého vrchola a vektor, ktorý ide z prvého do tretieho vrchola a pošleme ich funkcii, ktorú sme si za tým účelom vyrobili a ktorá nám vyrobí vektor kolmý na obidva a uloží ho do poľa `vektor`. Vypočítaný vektor nastavíme s pomocou funkcie `glNormal` ako normálový.

```
GLubyte red = farba & 0xff;
GLubyte green = (farba & 0xff00) >> 8;
GLubyte blue = (farba & 0xff0000) >> 16;
GLfloat colorvector[4];
colorvector[0] = red / 255.0f;
colorvector[1] = green / 255.0f;
colorvector[2] = blue / 255.0f;
colorvector[3] = 1.0f;
```

```
        glColor3ub(red,green,blue);
        glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE,colorvector);
```

Spracujeme farbu. Použijeme na to fintu zvanú bitová maska. Keď chceme napríklad zistiť nastavenie zelenej, vypočítame `(farba & 0xff00)` čo spraví „bitové and“, čo v tomto prípade znamená, že zo šestnástkového zápisu farby ponechá iba tretiu a štvrtú cifru odzadu, čo sú práve tie cifry, ktoré hovoria, akú hodnotu má zelená zložka. Aby ale táto hodnota nebola uložená v tretej a štvrtej cifre, ale ako sa patrí v prvých dvoch, musíme ju posunúť o osem bitov doprava. Na to slúži príkaz `>> 8`.

Vyrobíme si pole, do ktorého napcháme zistené hodnoty farieb. Farbu nastavíme aj s pomocou `glColor`, aby sme mohli teleso použiť aj keď nepoužívame osvetlenie, aj s pomocou `glMaterial`, aby to fungovalo aj pri osvetlení. (Kvôli tomu sme robili to pole.)

```
        if (pocet == 3)
        {
             glBegin(GL_TRIANGLES);
             glVertex3f(vrchol_x[a],vrchol_y[a],vrchol_z[a]);
             glVertex3f(vrchol_x[b],vrchol_y[b],vrchol_z[b]);
             glVertex3f(vrchol_x[c],vrchol_y[c],vrchol_z[c]);
             glEnd();
        }
        else
        {
             glBegin(GL_QUADS);
             glVertex3f(vrchol_x[a],vrchol_y[a],vrchol_z[a]);
             glVertex3f(vrchol_x[b],vrchol_y[b],vrchol_z[b]);
             glVertex3f(vrchol_x[c],vrchol_y[c],vrchol_z[c]);
             glVertex3f(vrchol_x[d],vrchol_y[d],vrchol_z[d]);
             glEnd();
        }
```

Na záver cyklu konečne patričnú stenu vykreslíme.

```
    }

    delete [] vrchol_x;
    delete [] vrchol_y;
    delete [] vrchol_z;
    vstup.close();
    glEndList();
    return teleso;
}
```

Keď cyklus dobehne (najpravdepodobnejšie pretože skončil súbor), zmažeme alokované polia, zavrieme súbor, ukončíme načítavanie zoznamu inštrukcii a vrátime jeho číslo.

Všetky OpenGL inštrukcie vykonané počas tejto procedúry máme teraz uložené v zozname inštrukcií. To znamená, že kedykoľvek náš objekt budeme chcieť nakresliť, nemusíme už znovu otvárať súbor ani počítať normálové vektory a trápiť sa s farbami. Stačí zaznamenaný zoznam inštrukcií vykonať s pomocou `glCallList`.

Teraz nám už ostáva iba spraviť drobné zmeny vo funkcii `main`. Pred samotným vykresľovacím cyklom to bude vyzerať takto:

```
    GLuint displayList = ObjToDisplist("domcek.obj");
    if (displayList == 0)
        return -1;

    Teleso k[10];

    for(int i = 0; i < 10; i++)
        k[i].Nastav(displayList,
                    (rand() % 100) / 10.0f - 5.0f,
                    (rand() % 100) / 10.0f - 5.0f,
                    -15.0f - (rand() % 100) / 10.0f);
```

Necháme si načítať súbor do zoznamu inštrukcií a ak sa to nepodarí, skončíme. Deklarujeme pole desiatich telies a nastavíme im načítaný zoznam inštrukcií a polohu. Ďalej ako po starom.

Úloha č.1: Vyskúšajte.

Úloha č.2: Ručne narobte v načítavanom súbore nejakú paseku, alebo skúste načítať nejaký úplne iný súbor. Aké chyby to bude vyhadzovať?

Úloha č.3: Nejako pekne vyfarbite blenderovskú opicu (červené oči a tmavomodrá hlava je zvlášť podmanivá kombinácia), vyexportujte ju a načítajte do vášho programu.

Úloha č.4: (Ťažká, pre hackerov.) Načítajte teleso tak, aby malo vyhladené hrany. To sa dá tak, že normálové vektory nebudete nastavovať každej stene, ale každému vrcholu. Na to, aby sa vypočítal normálový vektor pre vrchol, ale najprv treba vypočítať normálový vektor pre každú stenu, s ktorou vrchol susedí a z týchto vektorov spraviť priemer. Aby sa to dalo rozumne počítať, treba ale najprv všetky vrcholy **aj steny** dostať do pamäte.

Úloha č.5: Pozrite sa po sieti, či neexistujú voľne prístupné knižnice, ktoré vedia aj iné typy dátových súborov načítať ako zoznamy inštrukcií do OpenGL.

6. lekcia
Testy a buffery alebo „na čo všetko dáva OpenGL pozor“

Už sme sa stretli s tým, že OpenGL nenakreslí celkom všetko, čo mu povieme. Nekreslí veci mimo dosahu kamery – a to nielen tie, ktoré sa strácajú za okrajmi, ale aj tie čo sú priďaleko. Ak máme zapnutý Z-buffer, nekreslí ani veci, ktoré sú už prekryté inými. A aj k veciam, ktoré sa nakoniec na obrazovke objavia, sa správa rôzne. Niektoré nakreslí tak, ako sú, niektoré zmixuje s pozadím, takže vyzerajú ako priehľadné. V tejto lekcii si povieme, akými presne testami musí polygón prejsť, aby sa šťastne dostal na obrazovku a aké buffery (kusy pamäte, ktoré môžeme využiť k tomu, aby sa veci vykresľovali tak, ako potrebujeme) máme k dispozícii.

Takže najprv k tým bufferom. Prvý a snáď najdôležitejší je **farebný buffer**. Do neho sa ukladajú farby, ktoré sa nakoniec ocitnú na monitore. Môžu tam byť uložené mnohými spôsobmi, ako indexy z palety alebo po jednotlivých farebných zložkách, s hodnotou alfa alebo bez nej. Do tohto buffera zapisujeme vždy, keď niečo kreslíme a čítame z neho napríklad vtedy, keď potrebujeme nakresliť priehľadný polygón, ktorý zmení farbu toho, čo je za ním, ale je dôležité, aby cez neho bolo vidno to, čo už v bufferi máme.

Ďalší je **hĺbkový buffer** alebo **Z-buffer**, s ktorým sme sa už tiež stretli. Ak chceme, aby sa nám OpenGL staralo o viditeľnosť, do tohto buffera sa s každým nakresleným pixelom zapíše jeho vzdialenosť od pozorovateľa a iný objekt ho môže prekresliť iba vtedy, ak sa nachádza bližšie ku kamere.

Buffer, ktorý bude pre nás nový, je **šablóna** (po anglicky **stencil buffer**). Funguje rovnako ako šablóny na ceruzky alebo sprej – s jeho pomocou môžeme niektoré miesta na monitore „prikryť“ a spôsobiť, že sa ich ďalšie kreslenie nebude týkať. Jeho využitie si ukážeme v dnešnej lekcii.

Posledný buffer je **akumulátor** (**accumulation buffer**). Podobá sa na farebný buffer a používa sa na ukladanie viacerých obrázkov, z ktorých sa nakoniec vyrobí výsledný. To je dobré napríklad na antialiasing alebo na vytvorenie rozmazania spôsobeného pohybom.

Buffery, ktoré pri práci používate je väčšinou treba pred začiatkom každého kreslenia vymazať. Je to ale pomerne náročná operácia – skúste si vypočítať, koľko pixelov má váš monitor. Preto môže byť úsporou, že sa nemaže každý buffer zvlášť, ale všetky naraz. V našich doterajších príkladoch sme väčšinou mazali farebný aj hĺbkový buffer súčasne príkazom `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);` Ak by ste chceli pridať aj mazanie šablóny resp. akumulátora, treba do parametra funkcie doplniť `GL_STENCIL_BUFFER_BIT`, prípadne `GL_ACCUM_BUFFER_BIT`.

Dobre. Toto sú teda buffery, do ktorých môžeme zapisovať. Poďme sa pozrieť, ako sa také zapisovanie polygónu vlastne deje a čo ho môže ovplyvniť.

Za prvé sa skontrolujú masky. Môžeme totiž dočasne zakázať zápis do niektorého buffera alebo farebnej hladiny. Napríklad príkazom `glColorMask(GL_TRUE, GL_FALSE, GL_GALSE, GL_TRUE)` spôsobíme, že zapisovať sa bude len do červenej vrstvy a alfa kanálu farebného buffera, príkazom `glDepthMask(GL_FALSE)` spôsobíme, že do hĺbkového buffera sa nebude písať vôbec. Maskovanie šablón je trochu komplikovanejšie, pretože parameter pre funkciu `glStencilMask()` nie je typu `Glboolean` ale typu `Gluint` a maskovanie sa deje bit po bite. Nič, čo neprešlo týmto výberom, sa už do buffera nezapíše. (Táto veta môže nasledovať za každým z ďalších bodov.)

Za druhé sa skontroluje orezanie obrazovky. Nemusíme totiž vždy chcieť zapisovať do celého okna, ale môžeme vybrať obdĺžnik, do ktorého chceme kresliť. Obdĺžnik sa nastavuje príkazom `glScissor(GLint x, GLint y, GLsizei sirka, GLsizei vyska)` a to, či chceme orezanie použiť, nastavujeme príkazmi `glEnable(GL_SCISSOR_TEST)` resp. `glDisable(GL_SCISSOR_TEST)`. Možnosť takto orezať výstup sa dá použiť napríklad na to, že si celú obrazovku rozdelíme na niekoľko častí a do každej budeme zobrazovať iný pohľad na tú istú scénu.

Za tretie sa skontroluje alfa. Môžeme nastaviť, že chceme vykresľovať iba tie pixely, ktoré majú istú určenú alfu. Nastavuje sa to príkazom `glAlphaFunc(funkcia,hodnota)`, kde funkcia môže byť napríklad `GL_LESS`, `GL_LEQUAL`, `GL_EQUAL` alebo `GL_GREATER`[15] a `hodnota` je hodnota, s ktorou sa porovnáva. Ak teda napríklad chceme kresliť iba priesvitné pixely, nastavíme `glAlphaFunc(GL_LESS,1.0)`. To, či toto filtrovanie práve chceme, alebo nechceme použiť, zapíname známym spôsobom (`glEnable`/`glDisable`) s parametrom `GL_ALPHA_TEST`.

S pomocou tohto filtrovania môžeme napríklad kresliť scény s komplikovanou viacnásobnou transparentnosťou. Scénu necháme nakresliť dvakrát. Prvýkrát si ale zapneme, že chceme kresliť len nepriehľadné veci (s alfou jedna). Pritom naplníme Z-buffer. Potom scénu necháme vykresliť ešte raz, pričom tentokrát vykreslíme len priehľadné veci (s alfou menšou ako jedna). Pri druhom kreslení zakážeme zápis do Z-buffra (funkciou `glDepthMask(GL_FALSE)`) aby sa nám nestalo, že nám predné okno zakryje zadné. Vďaka tomu, že testovanie Z-buffra je stále zapnuté, aj keď sa do buffra nepíše, obe okná budú zakryté prípadnými nepriehľadnými predmetmi z prvého kreslenia.

Filtrovanie na alfu sa dá využiť aj na vykreslenie mreže, ktoré bude oveľa jednoduchšie, než to, ktoré sme robili vo štvrtej lekcii.

Za štvrté sa kontroluje šablóna. Použitie šablóny zapíname/vypíname s pomocou makra `GL_STENCIL_TEST`. Šablónový buffer sa ovláda s pomocou dvoch funkcií. Prvá je funkcia `glStencilFunc(funkcia,hodnota,maska)`. Podobne ako pri alfe nastavuje spôsob testovania pixelu, má ale jeden parameter navyše – šablónový buffer totiž môže obsahovať viacero šablón a o ktorú šablónu sa jedná sa mu dá povedať práve bitovou maskou, ktorá sa zadá do posledného parametra. Ak sa budete hrať iba s jednou šablónou, môžete tam všade dať jednotku. Keď testovanie nastavíme na `glStencilFunc(GL_ALWAYS,1,1)`, vyhovie automaticky každý pixel. Keď ho nastavíme na `glStencilFunc(GL_EQUAL,1,1)`, vyhovejú tie pixely, ktoré majú v šablónovom buffri na prvom bite jednotku. (Pomerne časté je aj použitie funkcie `GL_NOTEQUAL`.)[16]

Druhou podstatnou funkciou pri používaní šablón je funkcia `glStencilOp(zlyhalo, zlyhal_z-buffer, preslo)`, ktorá hovorí, čo sa bude zapisovať **do šablóny** v závislosti na tom, ako dopadol test na šablónu a na Z-buffer. (Do farebného buffera sa zapisuje **iba** vtedy, ak test prešiel!!!) Miesto každej z položiek môžeme dať `GL_KEEP` (nemeň), `GL_REPLACE` (prepíš referenčnou hodnotou z `glStencilFunc`) alebo `GL_ZERO` (vymaž). Takže ak chceme do šablóny zapisovať na prvý bit jednotku vtedy, ak test prešiel a Z-buffer je tiež v poriadku, nastavíme to volaním `glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE)`.

Šablóny sú dobré na rôzne veci. Na orezávanie telies rovinou, spájanie nepresne nastavených priesvitných polygónov, aj tá mreža by tým išla. V našom príklade s pomocou šablóny orežeme odraz v zrkadle.

Za piate sa kontroluje Z-buffer. Zapína a vypína sa s pomocou makra `GL_DEPTH_TEST`. Už ste s ním robili, takže viete, o čom je reč. Pre úplnosť dodajme, že sa môže nastaviť funkciou

15 Možností je viacero, v prípade potreby si ich nájdete v dokumentácii.

16 So šablónami sa dajú robiť aj iné triky, toto použitie (viacero vrstiev rozlíšených maskou) je len jedno z možných.

`glDepthFunc(test)`, ktorá opisuje, ktoré pixely testu vyhoveli. `test` môže byť `GL_LESS`, `GL_LEQUAL`, `GL_GREATER`, `GL_ALWAYS` alebo niektorá iná z týchto funkcií. Porovnáva sa aktuálna vzdialenosť pixelu od kamery s hodnotou v Z-buffri.

Za šieste sa spracuje blending – teda ak je alfa menšia ako jedna, zmixuje sa to s farbou pozadia. Tiež viete, o čom je reč. Zapína a vypína sa s pomocou `GL_BLEND` a treba tomu správne nastaviť `glBlendFunc`.

Za siedme by sa mal spraviť dithering. To je taká finta, s pomocou ktorej sa dá simulovať, že grafická karta zvláda viac farieb, než je v skutočnosti pravda. Voľakedy to bolo celkom užitočné, ale dnes to už väčšinou nie je treba. Zapína sa to s pomocou `GL_DITHER`, ale pravdepodobne to nič robiť nebude vzhľadom na to, že 16777216 farieb, ktoré je schopná zobraziť väčšina súčasných grafických kariet už pomerne slušne pokrýva celé farebné spektrum.

A nakoniec za ôsme sa uplatnia logické operácie. Môžete si nastaviť, že novopríchodzia bitmapa má spraviť `XOR` s tým, čo už je vo farebnom buffri. Stačí zadať príkaz `glEnable(GL_COLOR_LOGIC_OP)` a nastaviť `glLogicOp(GL_XOR)`. Rovnako to funguje s `GL_AND`, `GL_OR` atď. Štandardná hodnota je `GL_COPY`.

Dosť bolo teórie, poďme programovať. Našim cieľom bude blenderovská opica, ktorá sa obzerá v zrkadle.

Prvú kozmetickú zmenu prekoná naša trieda `Teleso`. Doteraz sme ho vykresľovali tak, že sme zrušili všetky doterajšie transformácie (volaním funkcie `glLoadIdentity`) a potom sme teleso presunuli a natočili tak, ako sme potrebovali. Tento prístup ale nie je vždy práve najšťastnejší. Môže sa totiž stať, že sa budeme na celú vytváranú scénu chcieť pozrieť z nejakej inej strany. Skrátka z iného miesta ako s počiatku súradnicovej sústavy. Najľahšie sa to dosahuje tak, že pred začatím samotného vykresľovania transformujeme súradnicovú sústavu tak, ako sa nám hodí – nepohneme tak kamerou, ale celou scénou. Ale ak pri tom budeme vykresľovať objekt, ktorý zavolá funkciu `glLoadIdentity`, je to problém, pretože táto funkcia zruší všetky transformácie (včítane už spomínanej počiatočnej hlavnej transformácie).

Na druhú stranu potrebujeme, aby transformácie, ktoré použijeme pri kreslení nášho telesa neovplyvnili objekty, ktoré budú vykresľované až po ňom. Aby sme to mohli dosiahnuť, môžeme si aktuálnu transformačnú maticu uložiť príkazom `glPushMatrix()`. Potom môžeme nakresliť naše teleso – pričom môžeme aktuálnu transformačnú maticu ľubovoľne meniť – a keď už budeme hotoví, znovu vytiahnuť uloženú maticu z pamäte príkazom `glPopMatrix()`.

Okrem týchto zmien ešte zrušíme otáčanie okolo dvoch osí. Ak sa opica krúti pred zrkadlom, bohato postačuje aj jedna os. Takže v súbore `teleso.cpp` to bude vyzerať takto:

```
void Teleso::Pootoc()
{
     yrot += dyr;
     yrot = yrot > 360.0f ? yrot - 360.0f : yrot;
     yrot = yrot < 0.0f ? yrot + 360.0f : yrot;
}

bool Teleso::Nakresli()
{
     glDisable(GL_TEXTURE_2D);
     glPushMatrix();
     glTranslatef(x, y, z);
     glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);
     glRotatef(yrot, 0.0f, 0.0f, 1.0f);
     glCallList(displaylist);
     Pootoc();
     glPopMatrix();
     return true;
}
```

Keďže model importovaný z blenderu nie je textúrovaný, textúry vypneme. Otočenie modelu

o -90 stupňov okolo osi `x` si vyžiadala poloha modelu v blenderi. Ak si budete vytvárať svoj model, je možné, že budete musieť použiť inú transformáciu, alebo sa bez nej zaobídete úplne.

Ako bude vyzerať hlavný súbor? Najprv deklarácie niektorých dôležitých vecí:

```
Teleso op;
GLuint textura;
GLfloat LightAmbient[] = { 0.2f, 0.2f, 0.2f, 1.0f };
GLfloat LightDiffuse[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightSpecular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightPosition[] = { 0.0f, 0.0f, 0.0f, 1.0f };
GLfloat BielyMaterial[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLdouble zrkadlo[] = { -1.0f, 0.0f, 0.0f, -1.0f };
```

Teleso `op` reprezentuje opicu, `textura` bude obsahovať textúru zrkadla. Nastavenie svetla sa oproti predošlej lekcii nezmenilo, iba sme polia deklarovali ako globálne. Pole `BielyMaterial` obsahuje biely materiál. Zaujímavé je pole `zrkadlo`. Je v ňom totiž uložená rovnica roviny, v ktorej sa zrkadlo nachádza. Je to presne tá rovnica roviny, na akú ste zvyknutí z analytickej geometrie, teda $ax+by+cz+d=0$ a koeficienty a, b, c a d sú uložené v poli. Na čo nám bude táto rovnica dobrá, uvidíte neskôr.

Na načítanie textúry si urobíme funkciu:

```
bool nacitaj_texturu()
{
     BITMAP* b = load_bitmap("frame.tga",NULL);
     if (b == NULL)
          return false;

     glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
     glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
     textura = allegro_gl_make_texture(b);

     destroy_bitmap(b);
     return true;
}
```

Vo funkcii `InitGL` zmeníme jednak to, že z nej vysekáme všetko, čo sa týka osvetlenia s výnimkou funkcie `glEnable(GL_LIGHTING)`, pretože to budeme používať až v samotnej kresliacej funkcii, jednak jej koniec zmeníme na

```
return nacitaj_texturu();
```

Ďalej si urobíme funkciu na nakreslenie zrkadla. Funkcia bude mať jeden nepovinný parameter, ktorý bude hovoriť, ako sa má obraz zrkadla zväčšiť alebo zmenšiť.

```
void draw_frame(GLfloat scale = 1.0f)
{
     glPushMatrix();
     glEnable(GL_TEXTURE_2D);
     glColor3f(1.0f,1.0f,1.0f);
     glMaterialfv(GL_FRONT_AND_BACK,GL_DIFFUSE,BielyMaterial);
     glTranslatef(0.0f, 0.0f, -5.0f);
     glScalef(1.0f,scale,scale);
     glBindTexture(GL_TEXTURE_2D, textura);
     glBegin(GL_QUADS);
          glNormal3f(1.0f,  0.0f,  0.0f);
          glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -2.0f, -2.0f);
          glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, -2.0f,  2.0f);
          glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  2.0f,  2.0f);
          glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f,  2.0f, -2.0f);
     glEnd();
     glPopMatrix();
}
```

Uložíme maticu, zapneme textúrovanie, pre istotu nastavíme biely materiál, nech sa nám textúra s ničím nemieša, zrkadlo posunieme na správne miesto a ak je treba zmenšíme.

(Zmenšujeme len súradnice `y` a `z`. Súradnicu `x` meniť nechceme.) Nakreslíme zrkadlo a transformačnú maticu uvedieme do pôvodného stavu.[17]

Vo funkcii `main` sa udejú tiež nejaké zmeny. Jednak pri štartovaní Allegra mu treba povedať, že ideme používať aj šablónový buffer a nastaviť jeho hĺbku. Táto hĺbka na mojej grafickej karte musí byť 32 a rovnako hĺbka Z-buffra sa musí nastaviť na 32, inak mi to odmietlo fungovať. Čo to bude robiť na inom hardvéri netuším. Takže nastavovanie OpenGL v Allegre bude vyzerať takto:

```
allegro_gl_clear_settings();
allegro_gl_set(AGL_Z_DEPTH, 32);
allegro_gl_set(AGL_STENCIL_DEPTH, 32);
allegro_gl_set(AGL_DOUBLEBUFFER, 1);
allegro_gl_set(AGL_COLOR_DEPTH, 16);

allegro_gl_set(AGL_SUGGEST,
     AGL_Z_DEPTH | AGL_STENCIL_DEPTH |
     AGL_DOUBLEBUFFER | AGL_COLOR_DEPTH);
```

Keď všetky systémové veci inicializujete tak, ako máte, inicializácia premenných a hlavný cyklus budú vyzerať takto:

```
GLuint displayList = ObjToDisplist("opica.obj");

if (displayList == 0)
     return -1;

op.Nastav(displayList,0.5f,0.0f,-5.0f);

while (DrawGLScene())
{
     if (key[KEY_ESC])
          break;
}
```

No a na záver to, o čo nám išlo. Samotná funkcia `DrawGLScene()`:

```
bool DrawGLScene()
{
     glClear(GL_COLOR_BUFFER_BIT |
          GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

Vymazali sme farebný buffer, Z-buffer aj šablónu.

```
     glLoadIdentity();
     glTranslatef(0.0,0.0,-5.0);
     glRotatef(-45.0f,0.0f,1.0f,0.0f);
     glTranslatef(0.0,0.0,5.0);
```

Na úplnom začiatku sme vymazali všetky transformácie a celú scénu pootočili okolo bodu `[0,0,-5]` o 45 stupňov. Toto je transformácia, ktorá predchádza všetky ďalšie, tým pádom sa na to celé budeme pozerať trochu z boku.

```
     glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);
     glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);
     glLightfv(GL_LIGHT1, GL_SPECULAR, LightSpecular);
     glEnable(GL_LIGHT1);
```

Nastavíme svetlo (všetky veci okrem toho, kde sa vlastne nachádza) a zapneme ho.

```
     glColorMask(0,0,0,0);
     glEnable(GL_STENCIL_TEST);
     glStencilFunc(GL_ALWAYS, 1, 1);
     glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
     glDisable(GL_DEPTH_TEST);
     draw_frame(0.6f);
```

17 Všimnite si, že všetky vrcholy zrkadla ležia v rovine $(-1).x-1=0$

```
glEnable(GL_DEPTH_TEST);
```

Teraz začalo to zaujímavé a nové. Potrebujeme zabezpečiť, aby sa opica odrážala iba od lesklej časti zrkadla. Takže si do šablóny zakreslíme, ktorá časť obrazovky obsahuje odrazovú plochu. Najprv teda vypneme kreslenie do farebného bufferu, zapneme šablónu, a povieme (volaním funkcie `glStencilFunc(GL_ALWAYS, 1, 1)`) že všetko, čo sa bude kresliť, automaticky vyhovie testu a že sa teda na všetky pixely, na ktoré sa kreslilo má do šablóny zapísať `1`. Vypneme Z-buffer (jednak že nekreslíme nič viditeľné, jednak nechceme, aby náhodou zlyhal test na Z-buffer a nedošlo k možnosti `GL_KEEP`[18]) a nakreslíme zmenšené zrkadlo – zmenšené je preto, aby sme prekreslili iba lesklú časť.

```
glColorMask(1, 1, 1, 1);
glStencilFunc(GL_EQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

V šablóne máme zakreslenú odrážavú plochu zrkadla, ideme kresliť do farebného buffra. Najprv zapneme kreslenie farieb, potom šablónový test nastavím tak, že ním prejdú len tie body obrazovky, ktoré majú v šablóne na prvej vrstve jednotku (tým pádom nech budem kresliť do farebného buffera čokoľvek, bude sa kresliť iba na odrážavú plochu zrkadla) a nastavíme, že do šablóny sa už čmárať nemá.

```
glEnable(GL_CLIP_PLANE0);
glClipPlane(GL_CLIP_PLANE0,zrkadlo);
```

Teraz ešte zariadim takú drobnosť, že vykresľovať sa budú len tie veci, ktorých geometrická poloha je „za zrkadlom“. Je jasné, že kým je opica pred zrkadlom, jej obraz bude za zrkadlom, ale keby ste robili scénu, ako opica vstupuje do magického zrkadla, jej odraz by zo zrkadla vyliezal a to by nepôsobilo dobre. Na to, aby sme mohli povedať, že obraz chceme orezať nejakou rovinou, potrebujeme rovnicu tej roviny (máme ju uloženú v poli `zrkadlo`) a potom už len stačí nastaviť túto rovinu ako „clip plane“ – orezávaciu rovinu a zapnúť orezávanie.

Ako OpenGL vie, ktorú z polrovín má zobraziť a ktorú nie? Zobrazuje tie body, pre ktoré platí, že $ax+by+cz+d\geq 0$. Takže ak je rovnica našej roviny $(-1).x-1=0$, bod `[-2,0,0]` sa zobrazí a bod `[2,0,0]` nie.

```
glPushMatrix();
     glTranslatef(-3.0,0.0,0.0);
     glScalef(-1.0f,1.0f,1.0f);
     glLightfv(GL_LIGHT1, GL_POSITION, LightPosition);
     op.Nakresli();
glPopMatrix();
glDisable(GL_CLIP_PLANE0);
glDisable(GL_STENCIL_TEST);
```

Uložíme maticu, posunieme sa o 3 smerom za zrkadlo (opica je 1,5 pred zrkadlom, takže jej obraz bude 1,5 za zrkadlom), rafinovaným použitím `glScalef` vytvoríme zrkadlový obraz, umiestnime svetlo (aby bolo aj osvetlenie v zrkadle naopak) a opicu nakreslíme. Odraz opice je šťastne nakreslený, šablónu aj orezávaciu rovinu vypneme.

```
glLightfv(GL_LIGHT1, GL_POSITION, LightPosition);

glEnable(GL_BLEND);
glBlendFunc(GL_ONE,GL_ONE);
draw_frame(1.0f);
glDisable(GL_BLEND);
```

Znovu umiestnime svetlo (tentokrát bez zrkadlenia). Zapneme mixovanie a cez odraz opice nakreslíme zrkadlo. Potom mixovanie opäť vypneme.

18 Aj keď vzhľadom na to, že sme Z-buffer práve vymazali, táto možnosť nenastane.

```
        op.Nakresli();
```

Nakoniec nakreslíme opicu originál.

```
        allegro_gl_flip();
        return true;
}
```

Úloha č.1: Vyskúšajte a pokúste sa čo najlepšie pochopiť veci ohľadom šablón.

Úloha č.2: Spravte obchádzanie kamerou. Kamera by mala stále mieriť na zrkadlo. (Keď prídete na pointu, táto úloha je nečakane ľahká.)

Úloha č.3: Spravte vstup opice do magického zrkadla. Vypnite orezávanie rovinou a pozrite sa, čo to robí.

Úloha č.4: Spravte mrežu s pomocou testu na alfu.

7. lekcia
Tiene
alebo „matica univerzálna“

Vykresľovanie tieňov je zaujímavým spestrením scenérie a dá sa spraviť viacerými spôsobmi. My sme sa rozhodli pre spôsob, ktorý nie je veľmi náročný na hardvér – využíva šablónový buffer a trochu matematiky. Šablónový buffer pri novších grafických kartách nebýva problémom. Tá matematika by problémom byť mohla, ale o nej si povieme niečo v tejto lekcii a ak to náhodou nepochopíte (vzhľadom na to, že to nijak podrobne vysvetľovať nebudem, ani veľkú šancu nemáte), tiež sa až taká veľké škoda nestane. Nájdete tu totiž funkciu, ktorá dostatočne všeobecným spôsobom robí to, čo budete na vykresľovanie tieňov potrebovať.

Pointa je v tom, že každá transformácia, ktorú OpenGL vykoná, sa pre potreby zobrazenia zapíše v podobe matice. Matica je učený výraz pre tabuľku čísel. OpenGL vie každú transformáciu zakódovať do matice s rozmermi 4 × 4 čísla. A ak napríklad necháte vykresliť nejaký bod (nejakou odrodou funkcie `glVertex`), tak s pomocou takejto tabuľky vie veľmi rýchlo vypočítať pozíciu tohto bodu po všetkých možných otočeniach a posunutiach a ktovie čím ešte, čo ste s vašou súradnicovou sústavou spáchali.

Matice majú ešte jednu výhodu – môžu sa medzi sebou násobiť.[19] A keď dve matice vynásobíte, výsledná matica bude vyjadrovať transformáciu, ktorú dostanete, keď najprv použijete transformáciu, ktorá zodpovedá prvej matici a potom použijete transformáciu, ktorá zodpovedá druhej matici.

Všetky tieto na prvý pohľad zložité reči sme už používali a vôbec nám nevadilo, že sme nevedeli, čo sa za tým skrýva. Funkcia `glLoadIdentity` nastavila transformačnú maticu na jednotkovú – jednotková matica je taká, ktorá zodpovedá transformácii „nehýb ničím“. Funkcia `glRotate` vynásobila aktuálnu maticu maticou zodpovedajúcou otočeniu určenému parametrami funkcie. Funkcia `glTranslate` vynásobí aktuálnu maticu maticou posunutia. Funkcia `glPushMatrix` uložila transformačnú maticu do zásobníka, aby sme si ju odtiaľ mohli opäť vytiahnuť s pomocou funkcie `glPopMatrix`.

Ale naspäť k tieňom. Predstavte si, že máte lampu umiestnenú v bode $[S_x, S_y, S_z]$ a rovinu, na ktorú chcete vrhnúť tieň, ktorá má rovnicu $ax+by+cz+d=0$. Ako zistiť, kam presne vrhne tieň bod $[x, y, z]$? Ľudia, ktorí majú aké-také skúsenosti s analytickou geometriou, vedia. Spravíme si parametrické vyjadrenie priamky, ktoré prechádza bodmi $[S_x, S_y, S_z]$ a $[x, y, z]$ a zistíme, kde sa táto priamka pretne so zadanou rovinou. To sa síce ľahko povie, ale ťažšie urobí, zvlášť keď to má človek rátať ručne. Povzbudivé je, že takáto transformácia „premietni do roviny“ sa tiež dá napísať ako matica. Ale nebudem vás napínať – rátal som to tri hodiny, dva razy som sa pomýlil, ale nakoniec sa mi podarilo vyrobiť funkciu, ktorá spraví potrebnú maticu a vynásobí ňou aktuálnu transformačnú maticu. Vyzerá to takto:

```
void NastavProjekciuDoRoviny(GLdouble* rovina, GLfloat* svetlo)
{
     GLfloat matica[16];

     matica[0] = rovina[1]*svetlo[1]+rovina[2]*svetlo[2]+rovina[3];
     matica[1] = -rovina[0]*svetlo[1];
     matica[2] = -rovina[0]*svetlo[2];
     matica[3] = -rovina[0];
     matica[4] = -rovina[1]*svetlo[0];
```

19 Ako presne násobenie matíc funguje, sa dozviete v nejakom kurze lineárnej algebry.

```
        matica[5] = rovina[0]*svetlo[0]+rovina[2]*svetlo[2]+rovina[3];
        matica[6] = -rovina[1]*svetlo[2];
        matica[7] = -rovina[1];
        matica[8] = -rovina[2]*svetlo[0];
        matica[9] = -rovina[2]*svetlo[1];
        matica[10] = rovina[0]*svetlo[0]+rovina[1]*svetlo[1]+rovina[3];
        matica[11] = -rovina[2];
        matica[12] = -rovina[3]*svetlo[0];
        matica[13] = -rovina[3]*svetlo[1];
        matica[14] = -rovina[3]*svetlo[2];
        matica[15] = rovina[0]*svetlo[0]
                              +rovina[1]*svetlo[1]+rovina[2]*svetlo[2];

        glMultMatrixf(matica);
    }
```

Funkcia na vstupe dostane štvorprvkové pole `rovina`, v ktorom sú uložené koeficienty *a*, *b*, *c* a *d* ktoré opisujú rovinu, v ktorej je svetlo uložené a trojprvkové pole `svetlo`, v ktorom sú uložené súradnice svetelného zdroja. Ďalej funkcia vypočíta, čo treba dať do matice a aktuálnu maticu touto maticou transformácie vynásobí.

Ďalej je už situácia jednoduchá. Predpokladajme, že máme k dispozícii nasledujúce premenné:

```
Teleso op;
GLfloat LightAmbient[] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat LightDiffuse[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightSpecular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightPosition[] = { 0.0f, 2.0f, -5.0f, 1.0f };
GLfloat Light2Diffuse[] = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat Light2Position[] = { 0.0f, 0.0f, 0.0f, 1.0f };
GLfloat BielyMaterial[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat Tien[] = { 0.5f, 0.50f, 0.5f, 0.5f };
GLdouble dlazka[] = { 0.0f, 1.0f, 0.0f, 2.0f };
```

Pričom do premennej `op` načítame zvyčajným spôsobom opicu[20], veci ohľadom `Light` sa týkajú dvoch svetiel, ktoré použijeme (1. svetlo bude svietiť na opicu zhora a vrhať tieň, druhé bude na to celé svietiť spredu, aby na to bolo vidieť). Polia `BielyMaterial` a `Tien` opisujú použitý materiál pre dlážku a tieň a pole `dlazka` obsahuje rovnicu roviny, v ktorej sa dlážka nachádza. (Keďže je rovina vodorovná a od počiatku súradnicovej sústavy má vzdialenosť 2, jej rovnica bude $0x+1y+0z+2=0$.)

Samotná funkcia `DrawGLScene` bude vyzerať takto:

```
int DrawGLScene()
{
     glClear(GL_COLOR_BUFFER_BIT |
               GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

     glLoadIdentity();
```

Zmažeme všetko, čo treba zmazať a transformačnú maticu nastavíme na stav „zatiaľ sa nič netransformuje“.

```
     glLightfv(GL_LIGHT2, GL_AMBIENT, LightAmbient);
     glLightfv(GL_LIGHT2, GL_DIFFUSE, Light2Diffuse);
     glLightfv(GL_LIGHT2, GL_SPECULAR, LightSpecular);
     glLightfv(GL_LIGHT2, GL_POSITION, Light2Position);
     glEnable(GL_LIGHT2);
```

Nastavíme svetlo, ktoré bude scénu osvetľovať spredu.

20 Teda načítame DisplayList z blenderu a nastavíme objekt spôsobom
`op.Nastav(displayList,0.5f,0.0f,-5.0f);`

```
glTranslatef(0.0,0.0,-5.0);
```

Celú scénu posunieme o päť dozadu.

```
glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, LightSpecular);
glEnable(GL_LIGHT1);

glLightfv(GL_LIGHT1, GL_POSITION, LightPosition);
```

Nastavíme svetlo, ktoré bude vrhať tieň. Jeho umiestnenie by malo byť niekde nad opicou.

```
glBegin(GL_TRIANGLES);
     glVertex3f(LightPosition[0]-0.1f,
          LightPosition[1],LightPosition[2]);
     glVertex3f(LightPosition[0],
          LightPosition[1]+0.1f,LightPosition[2]);
     glVertex3f(LightPosition[0]+0.1f,
          LightPosition[1],LightPosition[2]);
glEnd();
```

Na mieste, kde máme svetlo, nakreslíme malý trojuholník, nech máme prehľad.

```
op.Nakresli();
```

Nakreslíme opicu.

```
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, BielyMaterial);
glBegin(GL_QUADS);
     glVertex3f(-5.0f, -2.0f, 0.0f);
     glVertex3f(5.0f, -2.0f, 0.0f);
     glVertex3f(5.0f, -2.0f, -10.0f);
     glVertex3f(-5.0f, -2.0f, -10.0f);
glEnd();
```

Nakreslíme dlážku.

```
glColorMask(0,0,0,0);
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glDisable(GL_DEPTH_TEST);

glPushMatrix();
NastavProjekciuDoRoviny(dlazka,LightPosition);
op.Nakresli();
glPopMatrix();
```

Začíname vykresľovať samotný tieň. Najprv vypneme kreslenie do farebného buffera a zapneme šablónu, nastavíme, že ideme kresliť do prvého bitu šablóny, že tam naozaj budeme kresliť všetko a vypneme hĺbkový test. Odložíme si aktuálnu transformačnú maticu a necháme si zrátať maticu premietania do roviny. Necháme si vykresliť opicu (vďaka správnej transformácii to vykreslí iba opičiu placku na správnom mieste, aj to iba do šablóny) a vrátime predošlú maticu transformácie.

```
glEnable(GL_DEPTH_TEST);

glColorMask(1, 1, 1, 1);
glStencilFunc(GL_EQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, Tien);
```

```
        glBegin(GL_QUADS);
             glVertex3f(-5.0f, -2.0f, 0.0f);
             glVertex3f(5.0f, -2.0f, 0.0f);
             glVertex3f(5.0f, -2.0f, -10.0f);
             glVertex3f(-5.0f, -2.0f, -10.0f);
        glEnd();
```

Zapneme testovanie hĺbky, aby sa nám tieň neprekresľoval cez veci, ktoré sú pred ním. Dajte si pozor, aby ste počas inicializácie nastavili `glDepthFunc(GL_LEQUAL)`, štandardná hodnota je totiž `GL_LESS` a keby ste chceli na dlážku prikresliť niečo ďalšie, pri zapnutej kontrole hĺbky by sa nič nekreslilo. Nastavíme šablónu tak, aby sa do farebného buffera kreslilo iba tam, kde je v nej záznam (teda iba tam, kde je tieň). Ďalej zapneme a nastavíme blending, aby tieň neprekryl dlážku úplne, pretože v zložitejšom príklade, než je ten náš, môže byť na dlážke textúra.

Keď máme tieto nastavenia hotové, nastavíme tieňový materiál (sivý, priesvitný) a znovu prekreslíme dlážku. Keďže je tieň v šablóne, nakreslí sa iba tam, kam sa má.

```
        glDisable(GL_STENCIL_TEST);
        glDisable(GL_BLEND);

        allegro_gl_flip();

        return( TRUE );
}
```

Vypneme šablónu a priesvitnosť a stránku odošleme na obrazovku.

Úloha č.1: Vyskúšajte.

Úloha č.2: Pridajte za opicu stenu a vykreslite na tieň od prvej lampy, ktorá svieti spredu. Kvôli lepšiemu efektu môžete lampu odsunúť z pozície [0,0,0].

Úloha č.3: Skúste interaktívne pohnúť lampou a sledujte, ako sa bude meniť tieň.

Úloha č.4: Skúste zakomentovať časti týkajúce sa šablóny a namiesto tieňa na dlážku vykreslite placatú opicu.

8. lekcia
Krivky
alebo „ako spraviť vlnenie“

Ľudia, ktorí sa hrajú s grafickým dizajnom, často prídu na to, že nejaké veci je užitočné vykresľovať s pomocou kriviek. Tento prístup sa používa prinajmenšom od čias, kedy pán Pierre Étienne Bézier, ktorý pracoval pre firmu Renault vymyslel Bézierove krivky – čo sa nakoniec odrazilo aj na ladných tvaroch vtedajších áut.[21] S Bézierovmi krivkami, prípadne s niektorými ich zovšeobecneniami, ako sú B-spline či NURBS krivky sa môžete stretnúť vo viacerých grafických prgramoch ako napríklad GIMP či Blender. V tejto lekcii si ukážeme niektoré prístupy, ako krivku, či plochu vykresliť v OpenGL.

Najbežnejší prístup, ktorý sa dá realizovať s pomocou metód každej rozumnej grafickej knižnice je jednoducho vypočítať jednotlivé body krivky a pospájať ich. Vlniacu sa sínusoidu môžeme vyrobiť napríklad týmto spôsobom:

```
int DrawGLScene()
{
     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

     glLoadIdentity();
     glTranslatef(0.0f, 0.0f, -3.0f);

     GLfloat bodiky[11][3] =
          {{ -1.0f, 0.0f, -1.0f},
           { -0.8f, 0.0f, -1.0f},
           { -0.6f, 0.0f, -1.0f},
           { -0.4f, 0.0f, -1.0f},
           { -0.2f, 0.0f, -1.0f},
           {  0.0f, 0.0f, -1.0f},
           {  0.2f, 0.0f, -1.0f},
           {  0.4f, 0.0f, -1.0f},
           {  0.6f, 0.0f, -1.0f},
           {  0.8f, 0.0f, -1.0f},
           {  1.0f, 0.0f, -1.0f}};

     int i;

     static float t = 0.0f;
     t += 0.01f;

     for (i = 0; i < 11; i++)
          bodiky[i][1] = (GLfloat) sin(t + PI * (i - 5) / 5.0f);

     glColor3f(1.0, 1.0, 0.0);

     glColor4f(0.7,0.7,1.0,0.5);
     glLineWidth(2.0f);
     glBegin(GL_LINES);
          for( i = 1; i < 11; i++)
          {
               glVertex3fv(&bodiky[i-1][0]);
               glVertex3fv(&bodiky[i][0]);
          }
     glEnd();

     glPointSize(5.0f);
     glColor3f(1.0, 1.0, 0.0);
     glBegin(GL_POINTS);
     for (i = 0; i < 11; i++)
          glVertex3fv(&bodiky[i][0]);
```

21 On tie krivky vymyslel tri roky pred Bézierom aj Paul de Casteljau, ktorý pracoval pre konkurenčnú firmu Citroën, ale volajú sa po Bézierovi.

```
    glEnd();

    allegro_gl_flip();
    return( TRUE );
}
```

Najprv sme si vymazali buffre a nastavili kameru. Potom sme si vyrobili pole bodov, ktoré majú súradnicu `z` nastavenú na `-1` a súradnica `x` sa mení od `-1` do `1` s krokom `0.2`. (Na čo na začiatku nastavujeme súradnicu `y` nie je podstatné – budeme to priebežne meniť.) Statická premenná `t` nám bude slúžiť na to, aby sme si zapamätali, o koľko sa má naša sínusoida v každom kroku posunúť. Pri každom volaní funkcie `DrawGLScene()` sa jej hodnota zmení o `0.01`. V nasledujúcom cykle vypočítame `y`-ové súradnice jednotlivých bodov krivky.[22]

Samotné vykresľovanie krivky sa udeje v ďalšom cykle. S pomocou `GL_LINES` jednotlivé body pospájame.

Na záver ešte vykreslíme samotné body, ktoré našu krivku určujú. Vykreslíme ich ako štvorčeky so stranou `5`, aby boli dobre viditeľné.

Úloha č.1: Vyskúšajte.

Tento prístup je použiteľný v mnohých bežných situáciach. Má ale drobnú vadu – naša krivka je pomerne hranatá. Samozrejme sa dostatočným zhustením bodov dá vyhladiť. Niekedy to ale nie je to, čo potrebujeme. Niekedy napríklad chceme používateľovi dať možnosť manipulovať s určitým malým počtom kontrolných bodov, ale pritom chceme, aby výsledná krivka bola hladká.

Kvôli takýmto situáciam poskytuje OpenGL možnosť kresliť NURBS krivky.[23] Tieto krivky vznikli ako zovšeobecnenie už spomenutých Bézierových kriviek a majú mnohé pekné vlastnosti – napríklad sa s ich pomocou dajú kresliť presné kružnice či elipsy a nevadí im, keď sa na ne použije nejaká transformačná matica systému OpenGL. Ich presný matematický zápis je relatívne komplikovaný, to nám ale nebráni, aby sme ich používali k našej spokojnosti. Ich použitie môže byť napríklad takéto:

Najprv si kdesi vytvoríme smerník na objekt `GLUnurbsObj`:

```
GLUnurbsObj* krivka;
```

Potom pri inicialicácii OpenGL (napríklad vo funkcii `InitGL`) tento objekt (krivku) vyrobíme:

```
krivka = gluNewNurbsRenderer();
gluNurbsProperty(krivka, GLU_SAMPLING_TOLERANCE, 0.25f);
```

Nastavenie `GLU_SAMPLING_TOLERANCE` na `0.25` spôsobí, že jednotlivé úsečky, z ktorých sa krivka skladá, nebudú dlhšie než `0.25` pixelu. A nakoniec nahradíme pôvodný vykresľovací cyklus týmto kódom:

```
GLfloat knots[14] = {0,0,0,1,2,3,4,5,6,7,8,9,9,9};

gluBeginCurve(krivka);
    gluNurbsCurve(krivka,       // objekt
                  14,           // pocet knotov
                  knots,        // pole s knotmi
```

22 Ako ste si iste všimli, `x`-ová súradnica `i`-teho bodu je `bodiky[i][0]`, `y`-ová súradnica je `bodiky[i][1]` a súradnica `z` je `bodiky[i][2]`.

23 NURBS znamená Non-uniform Rational B-splines.

```
                    3,              // kolko "floatov" su od seba daleko
                                    // jednotlive body v poli
                    &bodiky[0][0], // pole s kontrolnymi bodmi
                    3,              // rad krivky
                    GL_MAP1_VERTEX_3);
    gluEndCurve(krivka);
```

Prvá (a v podstate jediná) záhadná vec, s ktorou sa v tomto kóde stretávame, je pole `knots`. Je to niečo, čo sa vyskytuje v matematickom popise NURBS kriviek a o čom vám zatiaľ stačí vedieť nasledujúce veci:

- Postupnosť čísel v tomto poli musí byť neklesajúca.
- Malo by mať toľko prvkov, koľko je počet kontrolných bodov plus rád krivky. (Rád krivky hovorí niečo o tom, aká má byť krivka hladká, väčšinou stačí použiť hodnotu 3.)
- NURBS krivka je síce svojimi kontrolnými bodmi ovládaná, ale nemusí cez ne prechádzať. Ak potrebujete, aby krivka aspoň začínala v prvom a končila v poslednom kontrolnom bode, prvých `n` a posledných `n` hodnôt (kde `n` je rád krivky) musí byť rovnakých.

NURBS krivku nám potom vytvorí aj vykreslí funkcia `gluNurbsCurve`. Jej parametre sú celkom pochopiteľné z príkladu – prvý je smerník na samotný objekt krivky. Druhý je počet prvkov v poli `knots` a tretí je smerník na toto pole. Ďalší parameter označuje, ako ďaleko sú od seba v poli `bodiky` jednotlivé body. (Jeden bod v našom prípade určujeme troma `float`mi, takže ďalší bod začína „o tri `float`y ďalej“.) Potom nasleduje smerník na prvú hodnotu v poli. Ďalší parameter je rád krivky. A nakoniec sa krivke (parametrom `GL_MAP1_VERTEX_3`) povie, čo má generovať. V našom prípade sú to trojsúradnicové bodíky (`VERTEX_3`) pre krivku (`MAP1`).

A to je všetko. O zvyšok sa postará knižnica.

Úloha č.2: Vyskúšajte.

Úloha č.3: Skúste pomeniť hodnoty v poli `knots`. Čo to urobí?

NURBS mechanizmus, ktorý sme opísali, má ešte jednu zaujímavú výhodu – s jeho pomocou sa dajú kresliť nie len krivky, ale aj plochy. Opäť stačí vygenerovať niekoľko kontrolných bodov, nastaviť scénu, zavolať správnu funkciu a hotovo. Ako presne to vyzerá, sa môžete pozrieť v nasledujúcom príklade:

```
#define PI 3.14159265

GLUnurbsObj* plocha;
GLfloat bodiky[11][11][3];
```

Najprv sme (ako globálne premenné) deklarovali NURBS objekt pre plochu a pole s bodíkmi.

```
bool InitGL(GLvoid)
{
     glViewport(0,0,SCREEN_W,SCREEN_H);
     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();
     gluPerspective(45.0f,
                    (GLfloat)SCREEN_W/(GLfloat)SCREEN_H,
                    0.1f,
                    100.0f);
     glMatrixMode(GL_MODELVIEW);
     glLoadIdentity();

     GLfloat mat_diffuse[] = { 0.7, 0.7, 1.0, 1.0 };
     GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
     GLfloat mat_shininess[] = { 100.0 };
     glClearColor (0.0, 0.0, 0.0, 0.0);
     glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse);
```

```
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
        glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_AUTO_NORMAL);
        glEnable(GL_NORMALIZE);
        glDepthFunc(GL_LEQUAL);
        glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

        plocha = gluNewNurbsRenderer();
        gluNurbsProperty(plocha, GLU_SAMPLING_TOLERANCE, 20.0f);
        gluNurbsProperty(plocha, GLU_DISPLAY_MODE, GLU_FILL);

        for (int i = 0; i < 11; i++)
             for (int j = 0; j < 11; j++)
             {
                  bodiky[i][j][0] = ((float)i / 5.0) - 1.0;
                  bodiky[i][j][2] = ((float)j / 5.0) - 1.0;
             }
        return TRUE;
   }
```

V inicializačnej funkcii sme ponastavovali všetky bežné počiatočné nastavenia. Potom sme vytvorili NURBS objekt, nastavili mu, že hrany polygónov, z ktorých bude tvorený, môžu byť dlhé maximálne 20 pixelov a že polygóny majú byť vyplnené – použili sme hodnotu `GLU_FILL`. Ak by sme chceli vykresliť iba drôtený model, použili by sme namiesto nej hodnotu `GLU_OUTLINE_POLYGON`.

Na koniec sme nastavili `x`-ové a `y`-ové súradnice bodov v poli tak, aby tvorili mriežku bodov s rozostupom `0.2`, kde aj jedna aj druhá súradnica sa bude meniť od `-1` do `1`.

```
int DrawGLScene()
{
     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

     glLoadIdentity();
     glTranslatef(0.0f, 0.0f, -3.0f);
     glRotatef(-135.0,1.0,0.0,0.0);


     int i,j;

     static float t = 0.0f;
     t += 0.1f;

     for (i = 0; i < 11; i++)
          for (j = 0; j < 11; j++)
          {
               float dist = (float) sqrt(bodiky[i][j][0]*bodiky[i][j][0] +
                                         bodiky[i][j][2]*bodiky[i][j][2]);
               bodiky[i][j][1] = (GLfloat) sin(t + 2 * PI * dist) / 4.0;
          }

     GLfloat knots[16] = {0,0,0,0,0,1,1,1,2,2,6,7,7,7,7,7};

     gluBeginSurface(plocha);
          gluNurbsSurface(plocha,            // objekt
               16,         // pocet knotov v jednom smere
               knots,      // pole s knotmi
               16,         // pocet knotov v druhom smere
               knots,      // pole s knotmi
               11*3,            // kolko floatov su od seba
                           //zaciatky jednotlivych sekvencii
               3,          // kolko floatov su od seba daleko
                           // jednotlive body v poli
               &bodiky[0][0][0],          // pole s kontrolnymi bodmi
               5,          // rad krivky v jednom smere
```

```
                5,          // rad krivky v druhom smere
                GL_MAP2_VERTEX_3);
        gluEndSurface(plocha);

        glPointSize(5.0f);
        glColor3f(1.0, 1.0, 0.0);
        glBegin(GL_POINTS);
        for (i = 0; i < 11; i++)
             for (j = 0; j < 11; j++)
                  glVertex3fv(&bodiky[i][j][0]);
        glEnd();


        allegro_gl_flip();

        return( TRUE );
}
```

Samotné vykresľovanie už prebehne známym spôsobom. Scénu si posunieme a pootočíme. Každému bodu vypočítame aktuálnu tretiu súradnicu. Spravíme si pole knotov a zavoláme funkciu `gluNurbsSurface` (s patričným okolím), ktorá nám vytvorí plochu. Na rozdiel od vytvárania krivky treba opísať dva smery – v každom treba zadať pole knotov (použili sme to isté na oba smery), opísať usporiadanie poľa s bodmi, poslať smerník na začiatok poľa, povedať, aký rád to má mať v jednom a v druhom smere a na koniec povieme, že treba vyrábať trojsúradnicové vrcholy pre plochu.

Na záver ešte vykreslíme riadiace bodíky. Niektoré z nich síce nebude vidno, lebo budú za plochou, ale to nevadí.

Úloha č.4: Vyskúšajte.

Úloha č.5: Skúste namiesto plochy vykresliť jej drôtený model.