



FAKULTET ZA POSLOVNU INFORMATIKU

Miloš Vasić

**Programiranje kompjuterske grafike u
OpenGL-u**
- Diplomski rad -



FAKULTET ZA POSLOVNU INFORMATIKU

Programiranje kompjuterske grafike u OpenGL-u

- Diplomski rad -

Mentor:
Prof. dr Dragan Cvetković

Student:
Miloš Vasić

Br. indeksa:
142/2004



FAKULTET ZA POSLOVNU INFORMATIKU

UNIVERZITET SINGIDUNUM
FAKULTET ZA POSLOVNU INFORMATIKU
Beograd, Danijelova 32

Broj: _____/2009

Kandidat: Miloš Vasić

Broj indeksa 142/2004

Smer: Projektovanje i programiranje

Tema: Programiranje kompjuterske grafike u OpenGL-u

Zadatak: Objasniti upotrebu OpenGL-a i njegovih osnovnih funkcionalnosti i celina. Za svaku funkcionalnost dati osnovna teorijska objašnjenja i konkretne primere u kodu pisanom u C++ programskom jeziku.

MENTOR

Prof. dr Dragan Cvetković

Datum odobrenja teme:

Beograd

DEKAN

Prof. dr Milan Milosavljević

Abstract

ENG

OpenGL technology is the most used graphic programming technology of today. The most expensive and most powerful applications are developed under OpenGL.

OpenGL is used in many areas. It is used for programming visualisation graphics for weather applications, movie production, medicine, military etc.

Greatest advantage of using OpenGL is his simplicity. It is easy to be use in development and he offers great possibilities. Many companies are using OpenGL for development. Some of them are: Adobe, Autodesk, Pixologic etc.

SRB

OpenGL tehnologija je najkorišćenija tehnologija za programiranje grafike današnjice. Najskuplje i najmoćnije aplikacije su razvijene pod OpenGL-om.

OpenGL se koristi u mnogim oblastima. Koristi se za programiranje grafike za aplikacije u meteorologiji, filmskoj produkciji, u medicini, vojsci itd.

Najveća prednost upotrebe OpenGL-a je njegova jednostavnost. Lako se koristi u razvoju i nudi velike mogućnosti. Mnoge kompanije koriste OpenGL za razvoj. Neke od njih su: Adobe, Autodesk, Pixologic itd.

Sadržaj

1. Uvod	1
1.1 Šta je to OpenGL ?	1
1.2 Istorijat	1
1.3 OpenGL arhitektura	1
1.4 Interfejs OpenGL-a	1
1.5 Osnovne mogućnosti OpenGL-a	2
1.6 Arhitektura OpenGL-a	2
1.7 Funkcionisanje konvejera OpenGL-a	3
1.8 Sintaksa komandi	3
2. Razvojno okruženje i njegovo podešavanje	4
2.1 Linux i OpenGL biblioteke	4
2.2 Instalacija	4
3. Rad sa bibliotekama	6
3.1 MESA i GLUT	6
3.2 Pravljenje OpenGL aplikacija	6
3.3 KDevelop IDE	7
4. Prva OpenGL aplikacija	8
4.1 Crveni kvadrat	8
5. GLX - Rad sa X Windows interfejsom	11
5.1 Displeji i X	11
5.2 Upravljanje konfiguracijama i vizuelizacijom	11
5.3 Prozori i površine za renderovanje	12
5.4 Rad sa kontekstom (Context Management)	13
5.5 Sinhronizacija	13
5.6 Rezime	14
5.7 Prikaz na punom ekranu	18
6. OpenGL podešavanja i primitive	23
6.1 Pre nego što počnemo	23
6.2 Funkcije podešavanja	23
6.3 Izvlačenje numeričkih stanja - podešavanja	23
6.4 Paljenje i gašenje stanja - podešavanja OpenGL state machine-e	23
6.5 Detalji OpenGL implementacije	24
6.6 Pronalaženje grešaka	24
6.7 Davanje smernica OpenGL-u	24
6.8 Rad sa primitivama	24
6.9 Tačke	26
6.10 Menjanje veličine tače	26
6.11 Antialiasing tačaka	26
6.12 Efekat udaljenosti	27
6.13 Primer tačke	27
6.14 Linije	28
6.15 Modifikovanje debljine linije	28
6.16 Antialiasing linija	28
6.17 Primer linija	28
6.18 Crtanje poligona u 3 dimenzije	29

6.19 Polygon Face Culling	29
6.20 Sakrivanje ivica poligona	30
6.21 Antialiasing poligona	30
6.22 Trouglovi	30
6.23 Četvorouglovi	31
6.24 Mnogouglovi	31
6.25 Atributi	31
7. Geometrijske transformacije i matrice	32
7.1 Razumavanje transformacija	32
7.2 Koordinate oka	32
7.3 Viewing transformacija	32
7.4 Modeling transformacije	33
7.5 Modelview dualitet	34
7.6 Projection transformacija	34
7.7 Viewport transformacije	34
7.8 Rad sa matricama	34
7.9 Transformation pipeline	34
7.10 Modelview matrica	35
7.11 Translacija	35
7.12 Rotacija	36
7.13 Skaliranje	36
7.14 Identity matrica	37
7.15 Matrix stack	38
7.16 Projekcije	39
7.17 Ortographic projekcija	39
7.18 Perspective projekcija	39
7.19 Napredna manipulacija matricama	39
7.20 Hardverske transformacije	40
7.21 Učitavanje matrica	40
7.22 Množenje matrica	41
7.23 Podešavanje Viewport-a	41
7.24 Manipulacija pogledom (viewpoint manipulation)	41
7.25 Sastavimo sve zajedno	42
8. Rad sa senčenjem, svetlima i materijalima	43
8.1 Postavljanje Shading modela	43
8.2 Svetla	43
8.3 Ambient svetlo	43
8.4 Diffuse svetlo	44
8.5 Specular svetlo	44
8.6 Spojimo ih sve zajedno	44
8.7 Materijali	45
8.8 Svojstva materijala	45
8.9 Postavljanje svojstava materijala	45
8.10 Upotreba izvora svetla i materijala	46
8.11 Spot svetla	46
8.12 Kreiranje Spot svetla	47
9. Zaključak	48
10. Literatura	49

1. Uvod

1.1 Šta je to OpenGL ?

OpenGL daje programeru interfejs prema grafičkom hardveru. To je moćna, low-level rendering biblioteka, dostupna za sve platforme sa širokom hardverskom podrškom. Dizajniran je za razvoj od kompjuterskih igara do CAD aplikacija. Mnoge igre kao što je Doom 3 su zasnovane na OpenGL-u. Takođe Blizzard Entertainment-ov Starcraft II za MacOS će biti zasnovan na njemu. Auto CAD i MAYA softverski paketi su zasnovani na OpenGL-u. Kao paralela Aero okruženju za Windows razvijen je Compiz Fusion za Linux - kompletno radjen pod OpenGL-om.

OpenGL obezbeđuje low-level rutine omogućujući programeru veliku kontrolu i fleksibilnost. Tim rutinama je zatim moguće kreiranje high-level rutina. U suštini OpenGL Utility Library (GLU) koji se isporučuje u većini OpenGL distribucija čini baš to. Napomenimo i to da je OpenGL samo grafička biblioteka ! On ne sadrži podršku za zvuk, mrežu ili bilo šta što nije u direktnoj vezi sa grafikom. Za te potrebe moramo koristiti neku drugu biblioteku - npr. ukoliko radimo aplikaciju koja zahteva proračun koalizije objekata i fiziku: možemo koristiti Havok biblioteku uz OpenGL. Havok je integrisan u mnoge kompjuterske igre kao i programski paket Maya.

1.2 Istorijat

OpenGL je razvijen originalno od strane Silicon Graphics kompanije (Silicon Graphics, Inc. - SGI) kao nezavisan od platforme višefunkcionalan API. Od 1992. razvoj OpenGL-a je preuzeo OpenGL Architecture Review Board (ARB) koji čine vodeći grafički proizvođači i drugi industrijski lideri - na primer: 3Dlabs, ATI, DELL, Evans & Sutherland, Hewlett-Packard, IBM, Intel, Matrox, NVIDIA, SGI, Sun Microsystems.

U ovom trenutku je najšira upotreba OpenGL 2.x (2.1) standarda, a u toku je razvoj 3.x verzije koja će uskoro biti u punoj upotrebi i donosi pregršt novina kao i velike razlike u odnosu na OpenGL 2.x.

1.3 OpenGL arhitektura

OpenGL je kolekcija od nekoliko stotina funkcija koje daju pristup svim funkcijama grafičkog hardvera. Ponaša se kao kolekcija stanja - opcija (states) koje govore OpenGL-u šta da radi i na koj način. Upotrebom API-a možete ih menjati. Na primer: tekuća boja, svetlo, broj bitova za boju i slično. Kada renderujemo, sve što je prikazano je rezultat OpenGL State Machine-a tj. kolekcije stanja - opcija - atributa. Bitno je da znate kakve će rezultate proizvesti određene opcije.

1.4 Interfejs OpenGL-a

OpenGL se sastoji od skupa biblioteka. Sve bazne funkcije se drže u osnovnoj biblioteci, za čije se označavanje koristi skraćenica GL. Pored osnovne, OpenGL sadrži i nekoliko dopunskih biblioteka. Prva od njih je biblioteka pomoćnih alata GLU (GL Utility). Sve funkcije te biblioteke definisane su preko baznih funkcija GL. U sastav GLU ušla je realizacija složenijih funkcija, kao što su skup popularnih geometrijskih primitiva (kocka, sfera, cilindar, disk), funkcije za kreiranje splajnova, realizacija dopunskih operacija nad matricama itd.

OpenGL ne sadrži nikakve specijalne komande za rad sa prozorima ili prihvatanje ulaznih informacija od korisnika. Zato su napravljene specijalne prenosive (portabilne) biblioteke koje obezbeđuju često korišćene funkcije za međusobno delovanje računara i korisnika, kao i za prikaz informacija preko podsistema prozora. Najpopularnija biblioteka je GLUT (GL Utility Toolkit). Formalno, GLUT nije u sastavu OpenGL-a, ali se ukljuuje u sve njegove distribucije i postoje realizacije za različite platforme. GLUT predstavlja samo minimalni neophodni skup funkcija za pravljenje OpenGL aplikacija. Funkcionalno analogna biblioteka GL(AU)X manje je popularna.

Osim toga, funkcije, specifične za konkretan podsistem prozora, obično ulaze u njegov programski interfejs aplikacije (API). Tako funkcije, koje podržavaju izvršavanje OpenGL-a, ulaze u sastav Win32 API (Microsoft Windows) i X Window System (UNIX / Linux - MacOS).

1.5 Osnovne mogućnosti OpenGL-a

Opisivanje mogućnosti OpenGL-a može da se obavi preko funkcija njegove biblioteke. Sve funkcije mogu da se podele u pet kategorija:

- Funkcije za opisivanje primitiva definišu objekte nižeg nivoa hijerarhije (primitive), koje je grafički podsistem sposoban da prikaže. U OpenGL-u u primitive spadaju tačka, linija, poligon (mnogougao) itd.
- Funkcije za opisivanje izvora svetla služe za opisivanje položaja i parametara izvora svetla, koji su raspoređeni u trodimenzionalnoj sceni.
- Funkcije zadavanja atributa pomoću kojih programer definiše kako će na ekranu izgledati prikazani objekat. Drugim rečima, ako se pomoću primitiva definiše šta se pojavljuje na ekranu, to atributi definišu način prikazivanja na ekranu. Preko atributa OpenGL dozvoljava zadavanje boje, karakteristika materijala, teksture, parametre osvetljavanja itd.
- Funkcije vizuelizacije omogućavaju da se definiše položaj posmatrača u virtuelnom prostoru, kao i parametri objektiva kamere. Znajući te parametre, sistem može ne samo da pravilno napravi prikaz, već i da iseče (odseče) objekte koji su van vidnog polja.
- Skup funkcija za geometrijske transformacije omogućava programeru da obavi različite transformacije objekata - rotiranje, pomeranje (translacija), skaliranje (promena dimenzija).

Pri tome, OpenGL može da obavi dodatne operacije, takve kao što je korišćenje tzv. splajnova za kreiranje linija i površina, uklanjanje nevidljivih delova prikaza, rad sa prikazima na nivou piksela itd.

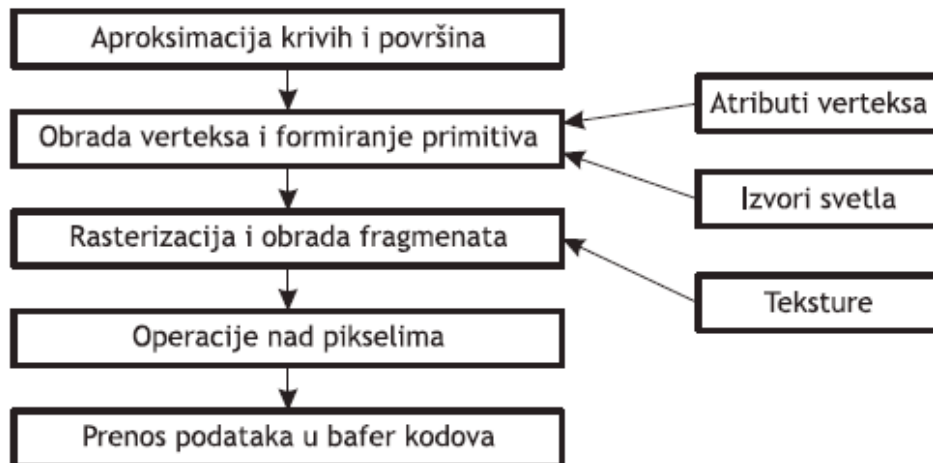
1.6 Arhitektura OpenGL-a

Funkcije OpenGL-a realizovane su u modelu klijent-server. Aplikacija nastupa u ulozi klijenta - ona produkuje komande, a server OpenGL ih interpretira i izvršava. Sam server može da se nalazi na tom istom računaru, na kom se nalazi i klijent (na primer, u obliku dinamički učitane biblioteke - DLL), tako i na drugom računaru (pri tome se može koristiti specijalni protokol prenosa podataka između računara). GL obrađuje i crta u baferu kadra (frame buffer) grafičke primitive uzimajući u obzir moguće izborne režime. Svaki režim može biti promenjen nezavisno od drugih.

Definisanje primitiva, izbor režima i druge operacije opisuju se pomoću komandi u obliku poziva funkcija prikladne biblioteke. Primitive se definišu skupom od jednog ili više verteksa (temena). Verteks definiše tačku, kraj odsečka ili ugao poligona. Svakom verteksu su pridruženi neki podaci (koordinate, boja, normala, koordinate teksture itd.), koji se nazivaju atributima. U većini slučajeva, svaki verteks se obrađuje nezavisno od drugih. Sa arhitektonske tačke gledišta, grafički sistem OpenGL je konvejer, koji se sastoji od sukcesivnih etapa obrade (tzv. protočna obrada) grafičkih podataka. Komande OpenGL uvek se obrađuju u poretku koji odgovara redosledu njihovog pojavljivanja mada se mogu pojaviti zadržke u proizvodnju njihovog efekta. U većini slučajeva, OpenGL daje neposredni interfejs, tj. definicija objekta izaziva njegovu vizuelizaciju u baferu kadra. Sa tačke gledišta programera, OpenGL je skup komandi koje upravljaju korišćenjem grafičkog hardvera. Ako se hardver sastoji samo od pomenutog bafera kadra, tada OpenGL mora potpuno da se realizuje korišćenjem resursa centralnog procesora. Obično grafički hardver daje različite nivoe ubrzavanja: od hardverske realizacije generisanja linija i poligona do moćnih grafičkih procesora sa podrškom za različite operacije nad geometrijskim podacima. OpenGL je međusloj između hardverskog i korisničkog nivoa, što omogućava da se dobije identičan interfejs na različitim platformama koristeći hardversku podršku. Osim toga, OpenGL može da se razmatra kao konačni automat, čije stanje je definisano mnoštvom vrednosti specijalnih promenljivih i vrednostima tekuće normale, boje, koordinata teksture i drugih atributa i obeležja. Sve te informacije će se koristiti pri ulasku u grafički sistem koordinata verteksa za pravljenje figure, u koju ona ulazi. Smena stanja obavlja se pomoću komandi koje se formiraju kao pozivi funkcija.

1.7 Funkcionisanje konvejera OpenGL-a

Date su redom etape obrade grafičkih podataka:



1.8 Sintaksa komandi

Definicije komandi GL nalaze se u fajlu `gl.h` i stoga ga je za njihovo uključanje potrebno koristiti:

```
#include <gl/gl.h> // <--- u Linuxu #include <GL/gl.h>
```

Za rad sa bibliotekom GLU potrebno je na analogan način uključiti fajl **glu.h**. Za razliku od standardnih biblioteka, paket GLUT treba instalirati i posebno priključiti.

Sve komande (procedure i funkcije) biblioteke GL počinju prefiksom `gl`, sve konstante - prefiksom `GL_`. Odgovarajuće komande i konstante biblioteka GLU i GLUT analogno imaju prefikse `glu` (`GLU_`) i `glut` (`GLUT_`). Osim toga, u nazive komandi ulaze i sufiksi, koji nose informaciju o broju i tipu parametara koji se prenose. U OpenGL-u pun naziv komande ima oblik:

```
type glCommand_name[1 2 3 4][b s i f d ub us ui][v]
(type1 arg1,...,typeN argN)
```

Naziv se sastoji od nekoliko delova:

- `gl` - naziv biblioteke, u kojoj je opisana ta funkcija: za bazne funkcije OpenGL, funkcije iz biblioteka GL, GLU, GLUT, GLAUX su `gl`, `glu`, `glut`, `aux`, respektivno.
- `Command_name` - naziv komande (procedure ili funkcije).
- `[1 2 3 4]` - broj argumenata komande.
- `[b s i f d ub us ui]` - tip argumenta: simbol `b` deklariše `GLbyte` (analogon `char` u C i C++ programskim jezicima), simbol `i` deklariše `GLint` (analogon `int`), simbol `f` deklariše `GLfloat` (analogon `float`) itd. Kompletan spisak tipova i njihovi opisi mogu da se vide u datoteci `gl.h`.
- `[v]` - postojanje tog simbola pokazuje da se kao parametar funkcije koristi pokazivač na niz vrednosti.

Simboli u kvadratnim zagradama u nekim nazivima se ne koriste. **Na primer: komanda `glVertex2i()` opisana je u biblioteci GL, i kao parametre koristi dva cela broja, a komanda `glColor3fv()` koristi kao parametar pokazivač na niz od tri realna broja.**

Korišćenje više varijanti svake komande može se delimično izbeći primenom preopterećenja funkcije jezika C++. Međutim, interfejs OpenGL nije predviđen za konkretan jezik programiranja i prema tome mora biti maksimalno univerzalan.

2. Razvojno okruženje i njegovo podešavanje

2.1 Linux i OpenGL biblioteke

OpenGL je već dugi niz godina podržan na Linux platformi (kao i na većini UNIX family operativnih sistema). Na UNIX-u je OpenGL podrška već integrisana u sistem. Za Linux, Mesa3D OpenGL biblioteka dolazi instalirana uz većinu X Server konfiguracija. Takođe većina proizvođača hardvera podržava hardversku akceleraciju za Linux.

X Window System je grafički korisnički interfejs za Linux. Sličan je grafičkom okruženju Windows-a i Mac OS-a. Međutim, X Window nije ograničen samo na jedan sistem na kome je instaliran ! Na primer: ukoliko se nalazite udaljeni od svog kompjutera, možete se konektovati na svoj X Server sa drugog računara, kao da sedite za sopstvenim.

Za X Window System postoje mnogi window manager-i. Najpopularniji su: KDE i Gnome. Svaki ima jedinstven vizuelni izgled i osećaj rada. Oni se nalaze na vrhu samog servera. Za Linux se najčešće koristi XFree86 open-source implementacija X Servera. Mi ćemo pisati OpenGL aplikacije za X Window System na Linuxu koji koristi XFree86.

2.2 Instalacija

U ovom radu ćemo programirati iz KDevelop IDE-a. Operativni sistem koji ćemo koristiti je RedHat-ova Fedora 10. Sve komponente (i sam KDevelop IDE) ćemo instalirati pomoću Fedorine aplikacije za dodavanje i otklanjanje komponenti.

Pre nego što instaliramo išta od softvera i biblioteka neophodno je dopuniti Fedorinu listu softverskih repozitorijuma. Sama Fedora ima default repozitorijum softvera, međutim u njemu se ne nalaze sve komponente koje je moguće instalirati na Linux distribucije. Dva najpoznatija opciona repozitorijuma su: RPM Fusion i RPM Livna.

Instalacija RPM Fusion-a na Fedora-i 10:

```
su -c 'rpm -Uvh http://download1.rpmfusion.org/free/fedora/rpmfusion-free-release-stable.noarch.rpm  
http://download1.rpmfusion.org/nonfree/fedora/rpmfusion-nonfree-release-stable.noarch.rpm'
```

Prethodnu komandu pokrenuti iz Linuxovog Terminala i po potrebi uneti administratorsku šifru kada sistem to zatraži. Kada smo instalirali RPM Fusion repozitorijum omogućeno nam je instaliranje NVIDIA Display drajvera (pošto se u ovom radu radi sa NVIDIA grafikom, slična procedura se može primeniti i sa drugim proizvođačima):

GeForce 6 ili jače:

```
yum install kmod-nvidia
```

GeForce 5 (FX serija):

```
yum install kmod-nvidia-173xx
```

GeForce 2 - 4:

```
yum install kmod-nvidia-96xx
```

Ukoliko instalirate NVIDIA display drajver sa NVIDIA oficijalnog installera (preuzet sa sajta) obavezno je instalirati i Kernel development komponente.

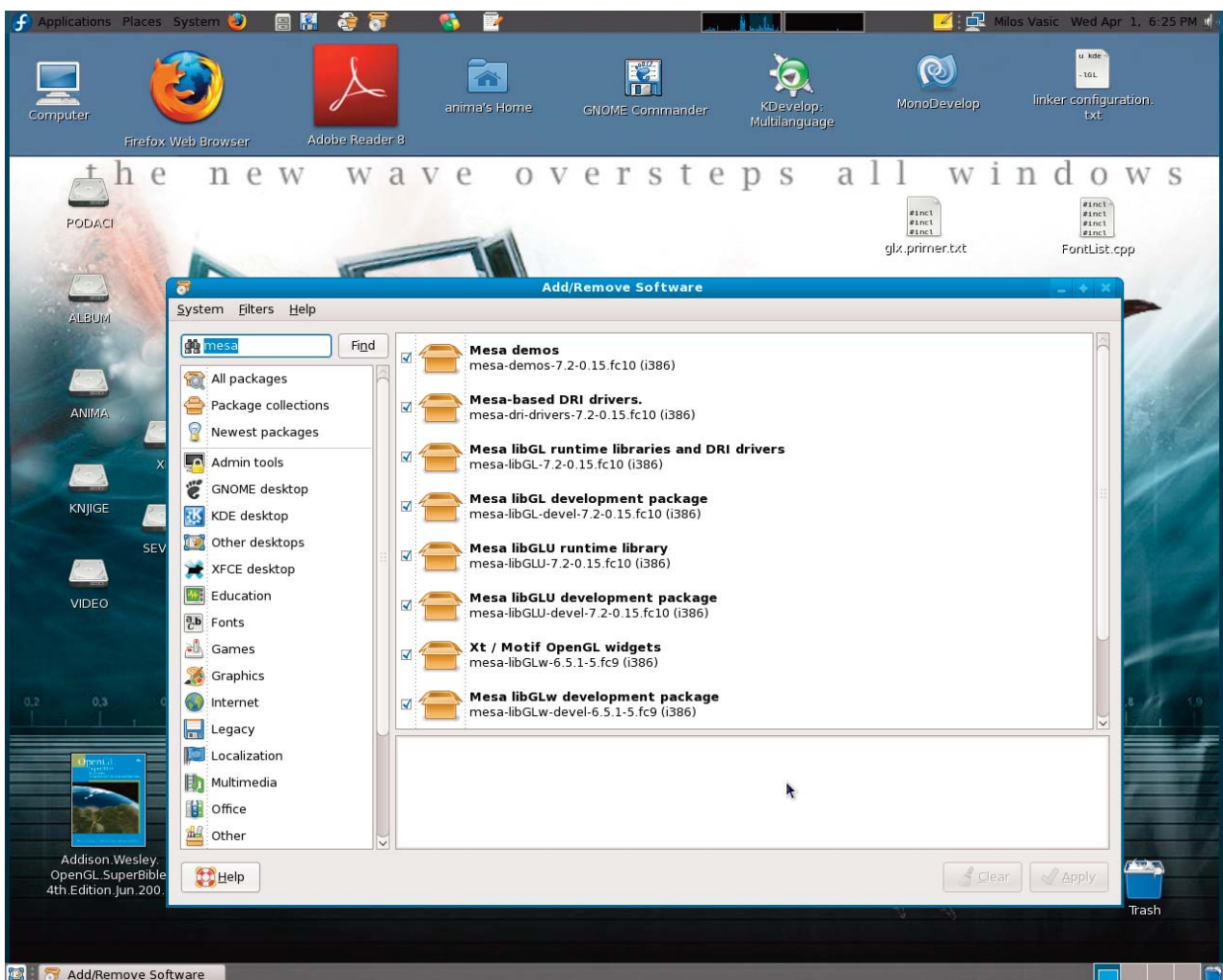
U svim aplikacijama je korišćena NVIDIA GeForce 7200 grafika.

Instalacija RPM Livna repozitorijuma na Fedora-i 10:

```
su -c "rpm -ivh http://rpm.livna.org/livna-release.rpm"
```

Po instalaciji RPM Livna repozitorijuma imamo sve neophodno za početak podizanja radnog okruženja (KDevelop IDE-a i razvojnih biblioteka).

Fedora 10 ima svoju aplikaciju za dodavanje i odstranjivanje komponenti sistema. Sve se svodi na search polje u koje unosimo ono što nas zanima. Dobijamo rezultate u vidu liste paketa koje možemo da dodajemo ili odstranjujemo. Takođe moguće im je pristupiti i putem hijerarhije komponenti aplikacije.



Na ovaj način ćemo instalirati sledeće komponente: Mesa3D, Free GLUT, GLX biblioteke, kao i sve što se tiče XFree86 servera i KDevelop IDE.

3. Rad sa bibliotekama

3.1 MESA i GLUT

Za rad sa OpenGL-om u Linuxu ćemo koristiti MESA i GLUT biblioteke. MESA je najčešće korišćena biblioteka za programiranje OpenGL-a u Linux okruženju, a GLUT tj. Free GLUT biblioteka je verzija GLUT-a za Linux.

3.2 Pravljenje OpenGL aplikacija

Upravo smo završili sa instalacijom IDE-a i neophodnih osnovnih biblioteka. Spremni smo da pravimo OpenGL aplikacije.

Da bismo napravili OpenGL aplikaciju neophodno je konfigurisanje Makefile-a. Na našu sreću KDevelop će to automatski raditi za nas. Makefile-ovi se koriste na Linuxu pri procesu build-ovanja za kompajliranje i spajanje source koda i konačno pravljenje egzekutabilnog fajla. **Oni sadrže instrukcije za kompajler i linker govoreći im gde šta da nađu i šta sa time da čine.**

Primer Makefile-a:

```
LIBDIRS = -L/usr/X11R6/lib -L/usr/X11R6/lib64 -L/usr/local/lib
INCDIRS = -I/usr/include -L/usr/local/include -L/usr/X11R6/include
COMPILERFLAGS = -Wall
CC = gcc
CFLAGS = $(COMPILERFLAGS) -g $(INCDIRS)
LIBS = -lX11 -lXi -lXmu -lglut -lGL -lGLU -lm
example : example.o
$(CC) $(CFLAGS) -o example $(LIBDIRS) example.c $(LIBS)
clean:
rm -f *.o
```

Prva linija kreira varijablu koja sadrži link parametre za biblioteke koje će biti uključene. U našem primeru LIBDIRS varijabla traži biblioteke u standardnim direktorijumima za biblioteke.

Druga linija traži header fajlove korišćene u kompajliranju. Zatim, slede flagovi kompajlera i za nas naj bitnija linija: LIBS = -lX11 -lXi -lXmu -lglut -lGL -lGLU -lm . Ona sadrži sve biblioteke koje treba linkovati u naš program.

Kao što smo već pomenuli KDevelop radi većinu koraka automatski. Međutim neophodno je u Compiler i Linker opcijama projekta dodati: -lX11 -lXi -lXmu -lglut -lGL -lGLU -lm. KDevelop će tu liniju koda postaviti u Makefile, čime smo zadali kompajleru da koristi OpenGL XFree86 biblioteke prilikom pravljenja našeg programa i da ih linkuje u naš program.

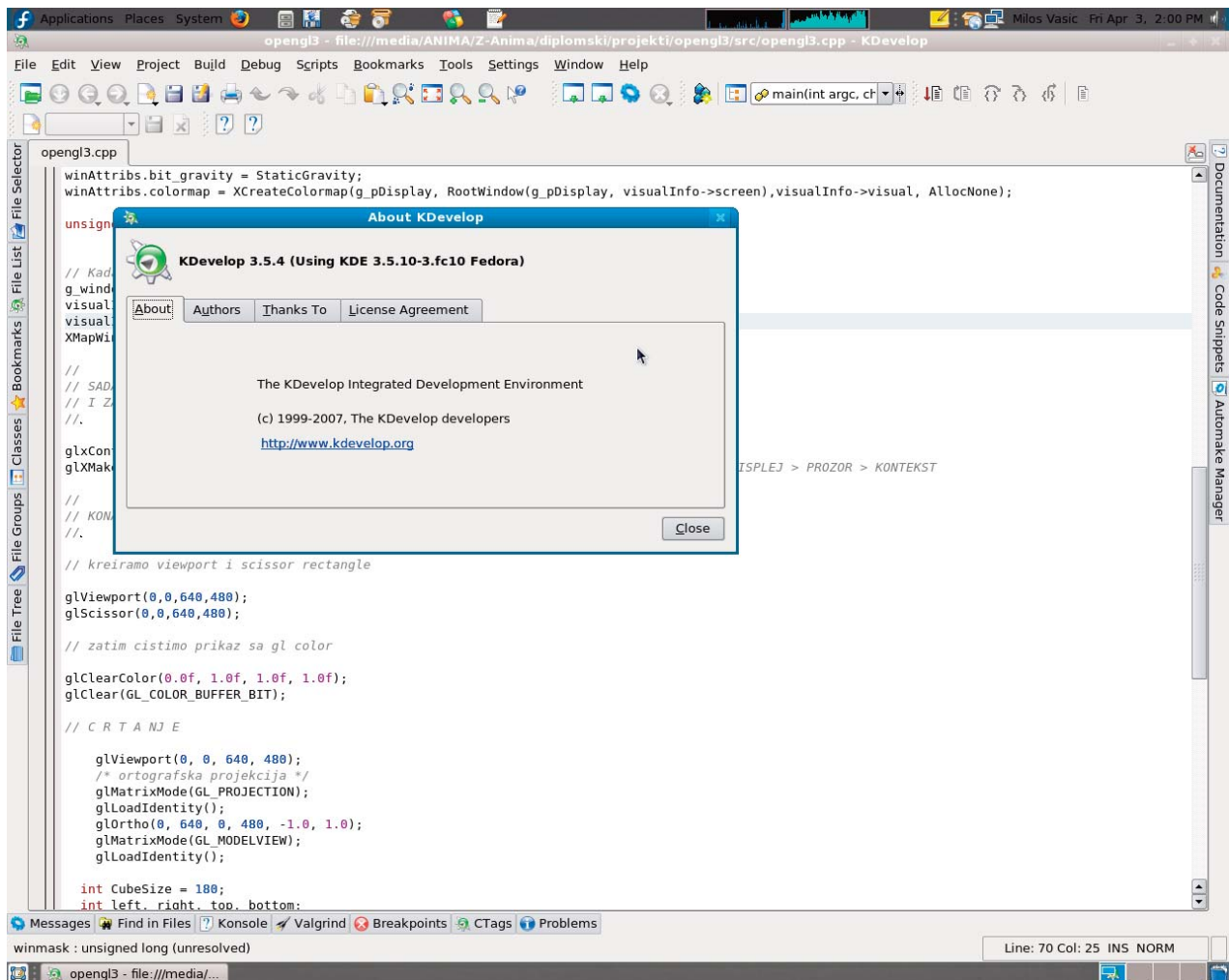
Sama sintaksa komande je prosta:

-l + Ime_biblioteke (-lGL)

3.3 KDevelop IDE

Za sve projekte tj. primere aplikacije koje budemo pisali koristićemo KDevelop (trenutna verzija: 3.5.4). Sledi niz komandi koje ćemo često koristiti u našem radu:

Project > Close // zatvaranje postojećeg projekta
 Project > New Project > C++ > Simple HelloWorld Program (obrišemo kod i pišemo naš)
 Project > Open project > ime_projekta.kdevelop // otvaranje postojećeg projekta
Project > Project Options > Configure options > Linker flags << uneti -l ime_biblioteke flagove redom
 Build > Build project (F8) // Pravljenje aplikacija
 Debug > Start (F9) // Pokretanje aplikacije
 Debug > Stop // Zaustavljanje aplikacije



KDevelop sadrži kompletan set alata neophodan za rad: menadžment projekta, gcc i g++ kompajler, korekciju sintakse... Interfejs je intuitivan pa nije neophodno posebno upoznavanje sa IDE-om.

4. Prva OpenGL aplikacija

4.1 Crveni kvadrat

Napravićemo novi projekat, a zatim napisati kod:

```
#include <stdlib.h>
/* uključivanje biblioteke GLUT */
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>

/* inicijalna sirina i visina prozora */
GLint Width = 512, Height = 512;

/* velicina kocke */
const int CubeSize = 200;

/* ova funkcija upravlja prikazom na ekranu */
void Display(void)
{
    int left, right, top, bottom;
    left = (Width - CubeSize) / 2;
    right = left + CubeSize;
    bottom = (Height - CubeSize) / 2;
    top = bottom + CubeSize;
    glClearColor(0, 0, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(255,0,0);
    glBegin(GL_QUADS);
    glVertex2f(left,bottom);
    glVertex2f(left,top);
    glVertex2f(right,top);
    glVertex2f(right,bottom);
    glEnd();
    glFinish();
}

/* Ova funkcija se poziva pri promeni dimenzija prozora */
void Reshape(GLint w, GLint h)
{
    Width = w;
    Height = h;
    /* postavljamo dimenzije oblasti prikazivanja, tj. vizir */
    glViewport(0, 0, w, h);
    /* ortografska projekcija */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, w, 0, h, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

```

/* Funkcija koja obradjuje dogadjaje od tastature */
void Keyboard( unsigned char key, int x, int y )
{
#define ESCAPE 's' // trebalo bi definisati karakter za esc

if(key==ESCAPE)
exit(0);
}

/* Glavna petlja aplikacije */
main(int argc, char *argv[])
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGB);
glutInitWindowSize(Width, Height);
glutCreateWindow("Primer crvenog kvadrata");
glutDisplayFunc(Display);
glutReshapeFunc(Reshape);
glutKeyboardFunc(Keyboard);
glutMainLoop();
}

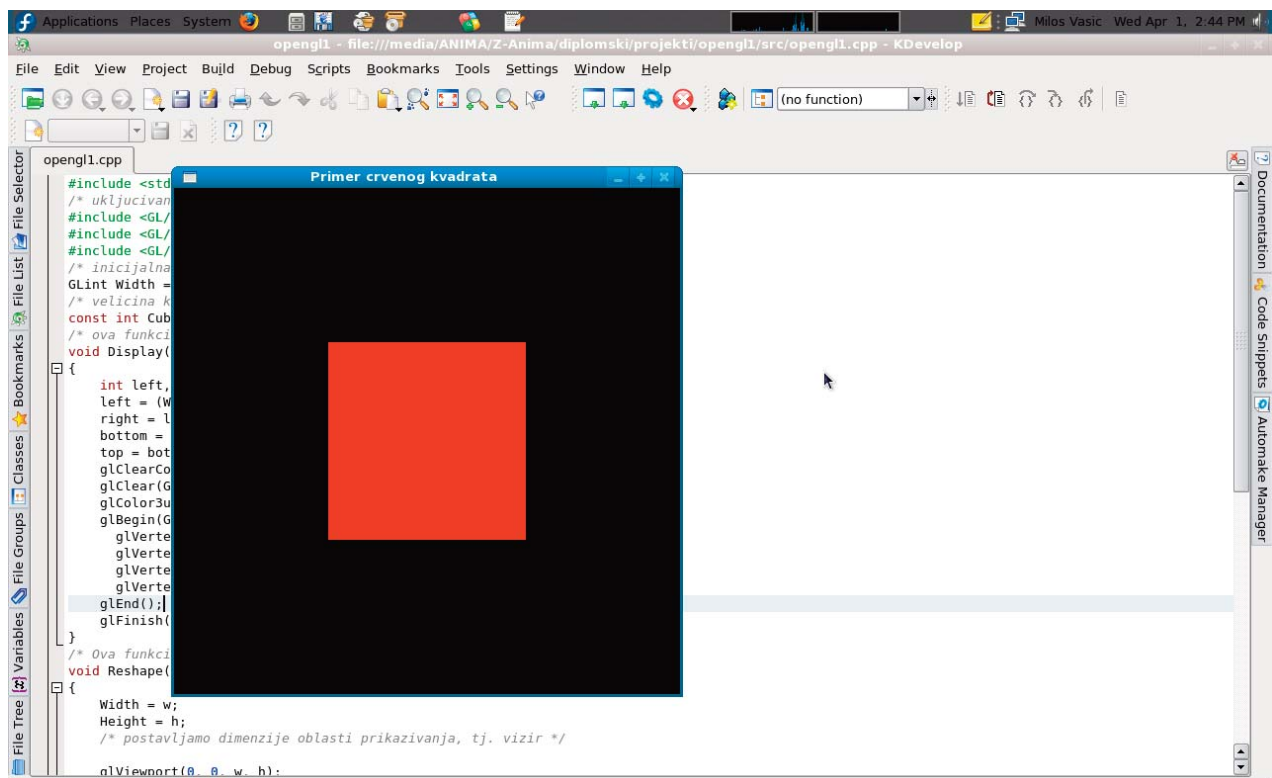
// KRAJ KODA !
//-----

```

Šta se tu zapravo zbiva ?

Tipičan program koji koristi OpenGL počinje sa definisanjem prozora u kome će se vršiti renderovanje (prikazivanje). Zatim se pravi kontekst (klijent) OpenGL i pridružuje se tom prozoru. Dalje programer može slobodno da koristi komande i operacije OpenGL API interfejsa.

Ovo je korišćenjem biblioteke GLUT - svojevrzni analogon klasičnog primera "Hello, World!". Sve što radi taj program je crtanje crvenog kvadrata u centru prozora.



Samim tim što je primer jednostavan, omogućava da se shvate osnovni principi programiranja pomoću OpenGL-a.

Bez obzira na veličinu, to je potpun program, koji može da se iskompajlira i izvršava na svakom sistemu koji podržava OpenGL i GLUT. Biblioteka GLUT podržava međusobno delovanje sa korisnikom preko tzv. funkcija sa povratnim pozivom (callback function). Ako je korisnik pomerio miša, pritisnuo taster na tastaturi ili promenio veličinu prozora, javlja se događaj i poziva se odgovarajuća korisnička funkcija - rukovalac događaja (funkcija sa povratnim pozivom). Trebalo bi razmotriti detaljnije funkciju main u datom primeru. Ova funkcija se sastoji od tri dela - inicijalizacije prozora, u kojem će crtati OpenGL, podešavanja funkcija sa povratnim pozivom i glavne petlje za obradu događaja.

Inicijalizacija prozora se sastoji od podešavanja odgovarajućih bafera kadra, početnog položaja i dimenzija prozora, kao i zaglavlja prozora.

Funkcija `glutInit(&argc, argv)` obavlja početnu inicijalizaciju same biblioteke GLUT.

Komanda `glutInitDisplayMode(GLUT_RGB)` inicijalizuje bafer kadra i postavlja režim RGB boja.

Komanda `glutInitWindowSize(Width, Height)` koristi se za zadavanje početnih dimenzija prozora.

Na kraju, `glut` `CreateWindow("Primer crvenog kvadrata")` zadaje zaglavlje prozora i vizuelizuje sam prozor na ekranu.

Zatim komande:

```
glutDisplayFunc(Display);  
glutReshapeFunc(Reshape);  
glutKeyboardFunc(Keyboard);
```

registruju funkcije `Display()`, `Reshape()` i `Keyboard()` kao funkcije koje će biti pozivane, respektivno, pri ponovnom prikazivanju prozora, promeni dimenzija prozora, pritiskanju tastera na tastaturi. Kontrola svih događaja i pozivanje potrebnih funkcija obavlja se unutar beskonačne petlje u funkciji `glutMainLoop()`.

Treba uočiti da biblioteka GLUT ne ulazi u sastav OpenGL-a i samo je prenosivi međusloj između OpenGL i podsistema prozora, tako da predstavlja minimalni interfejs. **OpenGL aplikacija za konkretnu platformu može da se napiše pomoću specifičnih API funkcija (Win32, X Window itd.), koje po pravilu daju šire mogućnosti. Mi ćemo u ovom radu, a od sledećeg poglavlja, raditi sa Linuxom i njegovom GLX bibliotekom za X Window System.**

Svi pozivi komandi OpenGL nastaju u rukovaocima događaja. Treba obratiti pažnju na funkciju `Display`, u koju je smešten kôd koji je neposredno odgovoran za crtanje na ekranu. Sledi sekvenca komandi funkcije `Display`:

```
glClearColor(0, 0, 0, 1);  
glClear(GL_COLOR_BUFFER_BIT);  
glColor3ub(255,0,0);  
glBegin(GL_QUADS);  
glVertex2f(left,bottom);  
glVertex2f(left,top);  
glVertex2f(right,top);  
glVertex2f(right,bottom);  
glEnd();
```

koja čisti prozor i prikazuje kvadrat na ekranu preko zadavanja četiri verteksa i boje.

5. GLX - Rad sa X Windows interfejsom

5.1 Displeji i X

Setite se da X Windows podržava klijent i server komponente koji mogu biti na potpuno odvojenim sistemima, što vam omogućuje da koristite vaš desktop odakle god želite.

Pre nego što kreiramo prozor moramo saznati na kom displeju će se izvršiti aplikacija. Displej će pomoći X serveru da shvati gde renderujemo. Koristićemo `XOpenDisplay()` funkciju kako bismo dobili tekući displej:

```
Display* dpy = XOpenDisplay(getenv("DISPLAY"));
```

Ovo će nam dati pokazivač na displej koji ćemo kasnije koristiti da bismo rekli X Serveru gde se nalazimo. Kada se naša aplikacija završi, potrebno je zatvoriti displej upotrebom `XCloseDisplay()` funkcije. Ovo govori X Serveru da smo završili i možemo zatvoriti konekciju:

```
XCloseDisplay(Display* display);
```

5.2 Upravljanje konfiguracijama i vizuelizacijom

Konfiguracije u Linuxu su veoma slične pixel formatima u Windowsu. Konfiguracije je nekada teško praviti pošto postoji mnogo faktora koje treba objediniti. Za početak možemo koristiti `glXGetFBConfigs` interfejs kako bismo dobili informacije o postojećim konfiguracijama:

```
GLXFBConfig* glXGetFBConfigs(Display* dpy, int screen, int* nelements);
```

Sledi lista konfiguracionih atributa:

`GLX_BUFFER_SIZE` Broj bitova color buffer-a.

`GLX_RED_SIZE` Broj bitova crvenog kanala color buffer-a.

`GLX_GREEN_SIZE` Broj bitova zelenog kanala color buffer-a.

`GLX_BLUE_SIZE` Broj bitova plavog kanala color buffer-a.

`GLX_ALPHA_SIZE` Broj bitova alfa kanala color buffer-a.

`GLX_DEPTH_SIZE` Broj bitova po pikselu depth buffer-a.

`GLX_STENCIL_SIZE` Broj bitova po pikselu stencil buffer-a.

`GLX_X_RENDERABLE` Podešeno je na `GLX_TRUE` ako X Server može renderovati na tu površinu.

`GLX_VISUAL_ID` Je `XID` vizuala.

`GLX_X_VISUAL_TYPE` Tip X vizuala ako konfiguracija podržava visual rendering.

`GLX_DRAWABLE_TYPE` Podržane validne površine za iscrtavanje. Može biti: `GLX_WINDOW_BIT`, `GLX_PIXMAP_BIT`, or `GLX_PBUFFER_BIT`.

`GLX_RENDER_TYPE` Tipovi konteksta koji mogu biti definisani. Može biti: `GLX_RGBA_BIT` ili `GLX_COLOR_INDEX_BIT`.

`GLX_FBCONFIG_ID` Je `XID` za `GLXFBConfig`.

`GLX_LEVEL` Frame buffer nivo..

`GLX_DOUBLEBUFFER` Je `GLX_TRUE` ako su kolor baferi double buffered.

`GLX_STEREO` Je `GLX_TRUE` ako kolor baferi podržavaju stereo buffering.

`GLX_SAMPLE_BUFFERS` Broj višestrukih bafera. Mora biti 0 ili 1.

`GLX_SAMPLES` Broj semplova po pikselu za multisample buffer. Biće 0 ako je `GLX_SAMPLE_BUFFERS` postavljen na 0.

`GLX_TRANSPARENT_TYPE` Određuje podršku za transparentiju. Vrednosti mogu biti `GLX_NONE`, `GLX_TRANSPARENT_RGB`, ili `GLX_TRANSPARENT_INDEX`.

`GLX_TRANSPARENT_RED_VALUE` Red vrednosti frame buffer piksela koje moraju biti postavljene da bi do transparentije došlo.

GLX_TRANSPARENT_GREEN_VALUE Green vrednosti frame buffer piksela koje moraju biti postavljene da bi do transparentije došlo.

GLX_TRANSPARENT_BLUE_VALUE Blue vrednosti frame buffer piksela koje moraju biti postavljene da bi do transparentije došlo.

GLX_TRANSPARENT_ALPHA_VALUE Alpha vrednosti frame buffer piksela koje moraju biti postavljene da bi do transparentije došlo.

GLX_TRANSPARENT_INDEX_VALUE Index vrednosti frame buffer piksela koje moraju biti postavljene da bi do transparentije došlo. Samo za index color konfiguracije.

GLX_MAX_PBUFFER_WIDTH Maksimalna širina koja može biti korišćena za pBuffer.

GLX_MAX_PBUFFER_HEIGHT Maksimalna visina koja može biti korišćena za pBuffer.

GLX_MIN_PBUFFER_PIXELS Najveća totalna veličina pBuffer-a, u pikselima.

GLX_AUX_BUFFERS Broj podržanih auxiliary buffer-a.

GLX_ACCUM_RED_SIZE Broj bitova u crvenom kanalu auxiliary buffer-a.

GLX_ACCUM_GREEN_SIZE Broj bitova u zelenom kanalu auxiliary buffer-a.

GLX_ACCUM_BLUE_SIZE Broj bitova u plavom kanalu auxiliary buffer-a.

GLX_ACCUM_ALPHA_SIZE Broj bitova u alpha kanalu auxiliary buffer-a.

Ako imate aplikaciju koja treba da renderuje u prozor potrebno joj proslediti konfiguraciju koja zadovoljava vaše potrebe za rendering u prozoru.

```
GLXFBConfig* glXChooseFBConfig(Display* dpy, int screen, const int* attrib_list, int* nelements);
```

Potrebno je proslediti ekran za koji ste zainteresovani. Takođe treba proslediti elemente koji su potrebni za konfiguraciju. Ti elementi su isti oni koje smo malopre nabrojali.

```
attrib_list = {atribut1, atribut1_vrednost,
               atribut2, atribut2_vrednost,
               atribut3, atribut3_vrednost,
               atribut4, atribut4_vrednost,
               0};
```

Ne zaboravite da upotrebite **XFree** kako biste očistili memoriju koja je popunjena povratnom vrednošću **glXChooseFBConfig** poziva.

Postoji nekoliko ključnih atributa na koje ćete obratiti pažnju kada kreirate config. Na primer: **GLX_X_RENDERABLE** treba da bude **GLX_TRUE** kako biste koristili OpenGL za rendering. **GLX_DRAWABLE_TYPE** treba da bude **GLX_WINDOW_BIT** ako renderujete u prozoru, **GLX_RENDER_TYPE** treba da bude **GLX_RGBA_BIT** ako želite da koristite RGB kolor mod i konačno, **GLX_COEFIG_CAVEAT** treba da bude podešen na **GLX_NONE** ili bar **GLX_SLOW_CONFIG**. Uz ova podešavanja možete podesiti i mnoga druga.

5.3 Prozori i površine za renderovanje

Vreme je da kreiramo prozor na displeju funkcijom **XCreateWindow()**.

```
Window XCreateWindow(Display* dpy, Window parent, int x, int y, unsigned int width, unsigned int height, unsigned int border_width, int depth, unsigned int class, Visual* visual, unsigned_long valuemask, XSetWindowAttributes* attributes);
```

Ova funkcija zahteva parent prozor, takođe možete koristiti glavni X prozor za to. Takođe je potrebno proslediti i koordinate pozicije i dimenzije. Klasa prozora može biti: **InputOnly**, **InputOutput**, **CopyFromParent**.

CopyFromParent će koristiti vrednosti parent prozora od koga se vrednosti preuzimaju. Atributi dodeljuju osobine prozoru, a valuemask polje govori X serveru na koje vrednosti treba da obrati pažnju.

Pošto smo kreirali prozor neophodno je kreirati i površinu za rendering u prozoru. Poziv funkcije za kreiranje oblasti mora primiti kompatibilne attribute konfiguracije kao i prozor u kome se polje kreira.

Upotrebićemo `glXCreateWindow()` funkciju.

```
GLXWindow glXCreateWindow(Display* dpy, GLXFBConfig config, Window win, const int* attrib_list);
```

`attrib_list` trenutno ne podržava nikakve parametre i služi za buduću ekspanziju. Prosledićemo NULL.

Funkcija će vratiti grešku ukoliko je jedan od uslova istinit: ako konfiguracija nije kompatibilna sa konfiguracijom prozora, ako konfiguracija ne podržava window rendering, ako je neki parametar ne-validan, ako je `GLXFBConfig` već pridružen nekom prozoru ili ako nije validan, ili ako postoji neka opšta greška pri kreiranju GLX prozora.

Pošto smo završili sa renderingom potrebno je uništiti oblast renderinga pa prozor:

```
glXDestroyWindow(Display* dpy, GLXWindow window); // uništavamo oblast renderinga
XDestroyWindow(Display* dpy, Window win); // uništavamo prozor u kome je renderovano
```

5.4 Rad sa kontekstom (Context Management)

Kontekst je set OpenGL iskaza koji su vezani za rendering površinu. Više konteksta može biti kreirano ali samo jedan može biti korišćen za rendering. Najmanje jedan kontekst je neophodan za rednering. Kontekst možete kreirati pomoću `glXCreateNewContext()` funkcije:

```
GLXContext glXCreateNewContext(Display * dpy, GLXFBConfig config, int render_type, GLXContext
share_list, bool direct);
```

`render_type` parametar prihvata `GLX_RGBA_TYPE` ili `GLX_COLOR_INDEX_TYPE`. Normalno, trebalo bi proslediti NULL u `share_list` parametar. Postavite `TRUE` za direct parameter zahteve tj. za vezu sa lokalnim X Severom ili `FALSE` za kontekst koji renderuje kroz X Server.

Da biste koristili kreirani kontekst upotrebite funkciju `glXMakeContextCurrent()`:

```
glXMakeContextCurrent(Display * dpy, GLXDrawable draw, GLXDrawable read, GLXContext ctx);
```

, a kada završite sa njegovom upotrebom potrebno ga je uništiti funkcijom: `glXDestroyContext()`:

```
glXDestroyContext(Display * dpy, GLXContext ctx);
```

Ako je kontekst vezan za bilo šta, neće biti uništen. Takođe je moguće kopirati kontekst iz jednog u drugi:

```
glXCopyContext(Display * dpy, GLXContext source, GLXContext dest, unsigned long mask);
```

5.5 Sinhronizacija

GLX ima nekoliko funkcija - komandi za sinhronizaciju:

Pozivom funkcije `glXWaitGL()` obezbeđujemo da će se završiti sav prikaz za prozor pre nego što se desi rendering. Na primer iscrtavanje dekoracije prozora od strane window manager-a pre nego što se desi rendering.

```
void glXWaitGL();
```

Isto tako, `glXWaitX()` omogućuje da se sav OpenGL rednering kompletira pre bilo kog OpenGL poziva

koji zatim sledi.

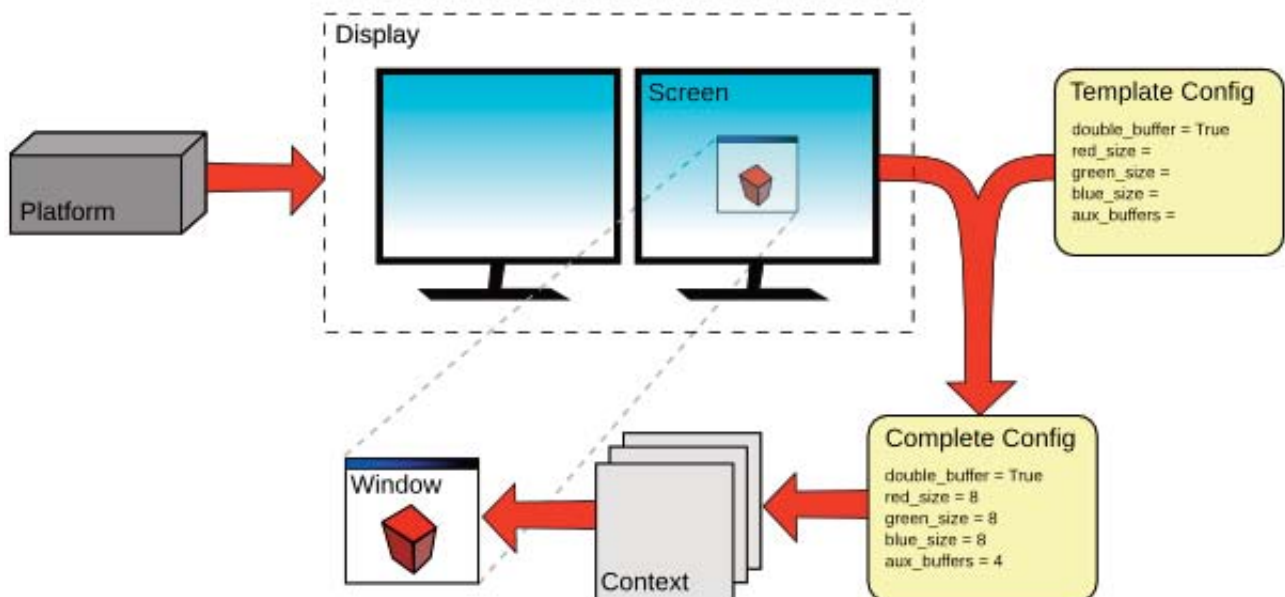
```
void glXWaitX();
```

5.6 Rezime

Dakle, upravo smo završili sa osnovnim funkcijama neophodnim za rad sa X Window System okruženjem i pozivanjem OpenGL renderinga u X prozor. Rezimiraćemo faze po redosledu:

- Definisanje displeja:
- Definisanje prozora:
- Definisanje površine za rendering
- Definisanje konteksta i odabir current konteksta
- Obavljanje OpenGL renderinga
- Uništavanje konteksta
- Uništavanje oblasti renderinga
- Uništavanje prozora
- Uništavanje displeja.

Na slici je prikazan odnos između displeja - prozora - konteksta :



Spajanje svega u celinu

Prva aplikacija je renderovala crveni kvadrat. Sada ćemo učiniti isto, međutim, kod je mnogo obimniji i komplikovaniji. Program koji sledi će upotrebom prethodno opisanih funkcija inicijalizovati X prozor i u njemu prikazati crveni kvadrat.

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glu.h>
```

```

Display* g_pDisplay = NULL; // DISPLEJ SA KOJIM SE RADI
Window g_window; // PROZOR SA KOJIM SE RADI
int main(int argc, char *argv[])
{
XVisualInfo* visualInfo = NULL; // ATRIBUTI DISPLEJA
XSetWindowAttributes winAttribs; // ATRIBUTI PROZORA
GLXContext glxContext;
// Open a connection to the X server
g_pDisplay = XOpenDisplay( NULL );
// glx verzija
// uslov je da se program izvrši ako je verzija glx veka od 1.2 ...
GLXFBConfig* fbConfigs; // iz niza prima konfiguraciju !

int numReturned;
int nMajorVer;
int nMinorVer;

glXQueryVersion(g_pDisplay, &nMajorVer, &nMinorVer);

std::cout << "Supported GLX version: " << nMajorVer << "." << nMinorVer;
if(nMajorVer == 1 && nMinorVer < 2)
{ std::cout << "ERROR: GLX 1.2 or greater is necessary! ";
XCloseDisplay(g_pDisplay);
exit(0); }

// kada smo ispitali verziju .... idemo dalje ...
// sada dolazi definisanje atributa. U windowsu se to radi
// iz pixel descriptora , a ovde to ide ovako:
// za single i double buffer ...

int singleBufferAttributess[] = {
GLX_DRAWABLE_TYPE, GLX_WINDOW_BIT,
GLX_RENDER_TYPE, GLX_RGBA_BIT,
GLX_RED_SIZE, 1, /* Request a single buffered color buffer */
GLX_GREEN_SIZE, 1, /* with the maximum number of color bits */
GLX_BLUE_SIZE, 1, /* for each component */
None
};

int doubleBufferAttributes[] = {
GLX_DRAWABLE_TYPE, GLX_WINDOW_BIT,
GLX_RENDER_TYPE, GLX_RGBA_BIT,
GLX_DOUBLEBUFFER, True, /* Request a double0buffered color buffer with */
GLX_RED_SIZE, 1, /* the maximum number of bits per component */
GLX_GREEN_SIZE, 1,
GLX_BLUE_SIZE, 1,
None
};

// ZADNJI ATRIBUT MORA BITI NONE !!!
// kada smo definisali atribut ubacujemo biramo ih za dalji rad :
// ps. dole izkomentarisani red je GLX 1.2 verzija funkcije ! - radi i iz 1.3 !!! i moze se tako koristiti...

//visualInfo = glXChooseVisual(g_pDisplay, DefaultScreen(g_pDisplay), attributess);

```

```
// ovo je verzija i nacin u 1.3:

fbConfigs = glXChooseFBConfig( g_pDisplay, DefaultScreen(g_pDisplay),
doubleBufferAttributes, &numReturned );

if ( fbConfigs == NULL ) { /* no double buffered configs available */
fbConfigs = glXChooseFBConfig( g_pDisplay, DefaultScreen(g_pDisplay),
singleBufferAttributes, &numReturned );
}

/* Create an X colormap and window with a visual matching the first
** returned framebuffer config */

visualInfo = glXGetVisualFromFBConfig( g_pDisplay, fbConfigs[0] );

// PRVO SMO DEFINISALI X DISPLAY
// PA MU DODALI ATRIBUTE
// SADA SLEDI ODREDJIVANJE TRIBUTA PROZORO
// I NJEGOVO KREIRANJE.

winAttribs.event_mask = ExposureMask | VisibilityChangeMask | KeyPressMask |
PointerMotionMask | StructureNotifyMask ;
winAttribs.border_pixel = 0;
winAttribs.bit_gravity = StaticGravity;
winAttribs.colormap = XCreateColormap(g_pDisplay, RootWindow(g_pDisplay,
visualInfo->screen),visualInfo->visual, AllocNone);
unsigned long winmask = CWBorderPixel | CWBitGravity | CWEventMask| CWColormap;

// Kada imamo atribut kreiramo prozor
g_window = XCreateWindow(g_pDisplay, DefaultRootWindow(g_pDisplay), 20, 20, 640,
480, 0,
visualInfo->depth, InputOutput,
visualInfo->visual, winmask, &winAttribs);
XMapWindow(g_pDisplay, g_window);

//
// SADA MORAMO KREIRATI CONTEXT
// I ZATIM GA POSTAVITI ZA CURRENT CONTEXT
//

glxContext = glXCreateContext(g_pDisplay, visualInfo, 0, True);
glXMakeCurrent(g_pDisplay, g_window, glxContext);
// SA LEVA NA DESNO PO HIJERARHIJI: DISPLEJ > PROZOR > KONTEKST

//
// KONACNO MOZEMO KORISTITI GL STEJTMENTE
//

// C R T A N J E

glViewport(0, 0, 640, 480);
/* ortografska projekcija */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, 640, 0, 480, -1.0, 1.0);
```

```

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
int CubeSize = 180;
int left, right, top, bottom;
left = (640 - CubeSize) / 2;
right = left + CubeSize;
bottom = (480 - CubeSize) / 2;
top = bottom + CubeSize;

glClearColor(0, 0, 0, 1);
glClear(GL_COLOR_BUFFER_BIT);
glColor3ub(255,0,0);
glBegin(GL_QUADS);
glVertex2f(left,bottom);
glVertex2f(left,top);
glVertex2f(right,top);
glVertex2f(right,bottom);
glEnd();

glXSwapBuffers(g_pDisplay, g_window);
sleep(2); // posle dve sekunde zatvara aplikaciju !

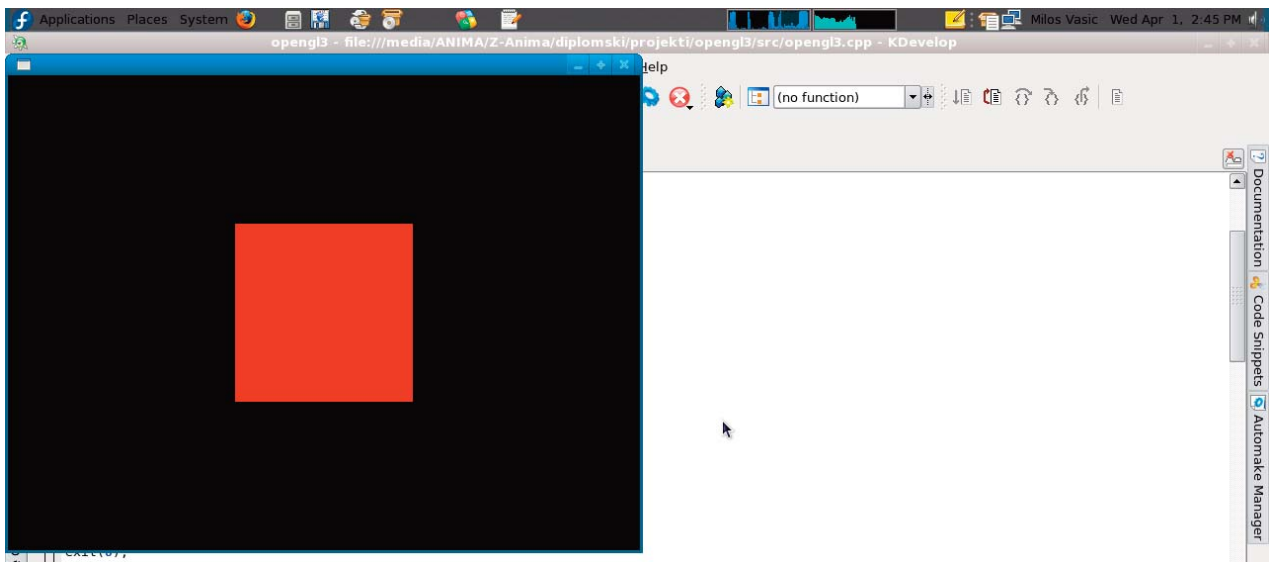
//
// A ZATIM REDOM UNISTAVAMO 0 KONTEKST, PROZOR I DISPLEJ
//

glXMakeCurrent(g_pDisplay, None, NULL);
glXDestroyContext(g_pDisplay, glxContext);
glxContext = NULL;
XDestroyWindow(g_pDisplay, g_window);
g_window = (Window)NULL;
XCloseDisplay(g_pDisplay);
g_pDisplay = 0;
return EXIT_SUCCESS;
}

// KRAJ KODA !

```

Rezultat pokretanja aplikacije:



5.7 Prikaz na punom ekranu

Konačno, vreme je da uradimo poslednju, takodje bitnu stvar, da omogućimo našoj aplikaciji pokretanje na punom ekranu. Ukoliko budemo programirali aplikaciju kao što je kompjuterska igra ili neka slična grafika u realnom vremenu ovo bi bilo neophodno uraditi. Kod je u suštini jednostavan i intuitivan. Sve dosad urađeno i dalje važi, samo sa jednom dopunom - upotrebom funkcija za promenu moda prikaza ekrana.

Sledi kod koji će do sada korišćeni crveni kvadrat na dve sekunde prikazati u takozvanom fullscreen modu, a zatim se vratiti u originalno stanje:

```
#include <X11/Xlib.h>
#include <X11/Xatom.h>
#include <X11/keysym.h>
#include <X11/extensions/xf86vmode.h>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glu.h>
#define WIDTH 1280
#define HEIGHT 1024
#define TITLE "OpenGL in X11 FULLSCREEN Mode"

/* stvari vezane za prozor grupisane zajedno */
typedef struct {
    Display *dpy;
    int screen;
    Window win;
    GLXContext ctx;
    XSetWindowAttributes attr;
    Bool fs;
    Bool doubleBuffered;
    XF86VidModeModeInfo deskMode;
    int x, y;
    unsigned int width, height;
    unsigned int depth;
} GLWindow;

/* naj vaznije varijable sadrze podatke o X serveru sa kojim radimo ! */

Display* display;
int screen;

/* instanca prozora */
Window window;
GLXContext context;
XSetWindowAttributes winAttr;
Bool fullscreen = True;
Bool doubleBuffered;
```

```
// OVO SU ORIGINALNI PODACI EKRANA ! BEZ NJIH NE MOZEMO VRATITI EKRAN U
// OSNOVNO PRVOBITNO STANJE !
```

```
XF86VidModeModeInfo desktopMode;
```

```
int x, y;
unsigned int width, height;
unsigned int depth;
```

```
/* attributes for a single buffered visual in RGBA format with at least
 * 4 bits per color and a 16 bit depth buffer */
```

```
static int attrListSgl[] =
{
    GLX_RGBA, GLX_RED_SIZE, 4,
    GLX_GREEN_SIZE, 4,
    GLX_BLUE_SIZE, 4,
    GLX_DEPTH_SIZE, 16,
    None
};
```

```
/* attributes for a double buffered visual in RGBA format with at least
 * 4 bits per color and a 16 bit depth buffer */
```

```
static int attrListDbl[] =
{
    GLX_RGBA, GLX_DOUBLEBUFFER,
    GLX_RED_SIZE, 4,
    GLX_GREEN_SIZE, 4,
    GLX_BLUE_SIZE, 4,
    GLX_DEPTH_SIZE, 16,
    None
};
```

```
GLWindow GLWin;
char* title;
```

```
/* PROTOTIPOVI DVE GLAVNE FUNKCIJE ZADUZENE ZA REALIZACIJU FULLSCREEN6A */
void createWindow();
void destroyWindow();
```

```
// create a window
```

```
void createWindow()
{
    XVisualInfo* vi;
    Colormap cmap;
    int i, dpyWidth, dpyHeight;
    int glxMajor, glxMinor, vmMajor, vmMinor;
    XF86VidModeModeInfo** modes;
    int modeNum, bestMode;
    Atom wmDelete;
    Window winDummy;
    unsigned int borderDummy;
    /* postavlja najbolji mod za tekuci */
    bestMode = 0;
```

```

/* uspostavljanje konekcije */
display = XOpenDisplay(0);
screen = DefaultScreen(display);
XF86VidModeQueryVersion(display, &vmMajor, &vmMinor);
printf("XF86 VideoMode extension version %d.%d\n", vmMajor, vmMinor);
XF86VidModeGetAllModeLines(display, screen, &modeNum, &modes);
// ODAVDE POCINJE GLAVNI DEO KODA ZA FULL SCREEN !
/* cuvamo trenutni desktop mod za kasniju upotrebu */
GLWin.deskMode = *modes[0]; // 0 JE TEKUCI MOD ! SADASNJA POSTAVKA EKRANA !
desktopMode = GLWin.deskMode; // <- KLJUCNI DEO SLAGALICE - VARIJABLA POSREDNIK
// UZIMA MOD KOJI TREBA VRATITI !
/* trzaimo mod sa rezolucijom koja nam je potrebna */
for (i = 0; i < modeNum; i++)
{
    if ((modes[i]->hdisplay == width) && (modes[i]->vdisplay == height))
        bestMode = i; // NASLI SMO MOD KOJI MOZEMO KORISTITI !
}
/* postavljamo vusal */
vi = glXChooseVisual(display, screen, attrListDbl);
if (vi == NULL)
{
    vi = glXChooseVisual(display, screen, attrListSgl);
    doubleBuffered = False;
    printf("singlebuffered rendering will be used, no doublebuffering available\n");
}
else
{
    doubleBuffered = True;
    printf("doublebuffered rendering available\n");
}
glXQueryVersion(display, &glxMajor, &glxMinor);
printf("GLXVersion %d.%d\n", glxMajor, glxMinor);
/* kreiramo GLX context */
context = glXCreateContext(display, vi, 0, GL_TRUE);
/* kreiramo color map */
cmap = XCreateColormap(display, RootWindow(display, vi->screen),
vi->visual, AllocNone);
winAttr.colormap = cmap;
winAttr.border_pixel = 0;
if (fullscreen)
{
    /* switch to fullscreen - KONACNO PREBACIVANJE U FULLSCREEN MODE
    - SA 0 NA BESTMODE */
    XF86VidModeSwitchToMode(display, screen, modes[bestMode]);
    XF86VidModeSetViewPort(display, screen, 0, 0);
    dpyWidth = modes[bestMode]->hdisplay;
    dpyHeight = modes[bestMode]->vdisplay;
    printf("resolution %dx%d\n", dpyWidth, dpyHeight);
    XFree(modes);
    /* atributi prozora */
    winAttr.override_redirect = True;
    winAttr.event_mask = ExposureMask | KeyPressMask | ButtonPressMask |
    StructureNotifyMask;
    window = XCreateWindow(display, RootWindow(display, vi->screen), 0, 0, dpyWidth, dpyHeight, 0,
    vi->depth, InputOutput, vi->visual, CWBorderPixel | CWColormap | CWEventMask |

```

```

CWOOverrideRedirect, &winAttr);
XWarpPointer(display, None, window, 0, 0, 0, 0, 0);
XMapRaised(display, window);
XGrabKeyboard(display, window, True, GrabModeAsync, GrabModeAsync, CurrentTime);
XGrabPointer(display, window, True, ButtonPressMask, GrabModeAsync,
GrabModeAsync, window, None, CurrentTime);
}
else
{
/* u slucaju da ne postoji trazeni mod kreira prikaz u prozoru - windowed mode */
winAttr.event_mask = ExposureMask | KeyPressMask | ButtonPressMask | StructureNotifyMask;
window = XCreateWindow(display, RootWindow(display, vi->screen),
0, 0, width, height, 0, vi->depth, InputOutput, vi->visual, CWBorderPixel | CWColormap |
CWEventMask, &winAttr);
/* only set window title and handle wm_delete_events if in windowed mode */
wmDelete = XInternAtom(display, "WM_DELETE_WINDOW", True);
XSetWMProtocols(display, window, &wmDelete, 1);
XSetStandardProperties(display, window, title, title, None, NULL, 0, NULL);
XMapRaised(display, window);
}
/* pripajamo glx kontekst prozoru */
glXMakeCurrent(display, window, context);
if (glXIsDirect(display, context))
printf("DRI enabled\n");
else
printf("no DRI available\n");
}

// destroy the window
void destroyWindow()
{
if(context)
{
if( !glXMakeCurrent(display, None, NULL))
{
printf("Could not release drawing context.\n");
}
/* destroy the context */
glXDestroyContext(display, context);
context = NULL;
}
/* vracamo nazad originalna podesavanja prikaza ! */
if(fullscreen)
{
XF86VidModeSwitchToMode(display, screen, &desktopMode);
XF86VidModeSetViewPort(display, screen, 0, 0);
}
XCloseDisplay(display);
}

// SVE SE ZATIM KORISTI U MAIN F-JI ...

```

```
int main(int argc, char ** argv)
{
    XEvent event;
    Bool done = False;
    width = WIDTH;
    height = HEIGHT;

    createWindow();

    // IZMEDJU KREIRANJA I UNISTENJA PROZORA DOLAZI KOD ZA ISCRTAVANJE

    // C R T A N J E

    glViewport(0, 0, 640, 480);
    /* ortografska projekcija */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, 640, 0, 480, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    int CubeSize = 180;
    int left, right, top, bottom;
    left = (640 - CubeSize) / 2;
    right = left + CubeSize;
    bottom = (480 - CubeSize) / 2;
    top = bottom + CubeSize;
    glClearColor(0, 0, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(255,0,0);
    glBegin(GL_QUADS);
    glVertex2f(left,bottom);
    glVertex2f(left,top);
    glVertex2f(right,top);
    glVertex2f(right,bottom);
    glEnd();
    glXSwapBuffers(display, window);
    sleep(2); // posle dve sekunde zatvara aplikaciju !

    // KRAJ OPENGL KODA ZA ISCRTAVANJE

    destroyWindow();

    return 0;
}
```

6. OpenGL podešavanja i primitive

6.1 Pre nego što počnemo

Konačno je došlo vreme da zaronimo u dubinu OpenGL-a. Pre nego što počnemo sa radom sa OpenGL primitivama neophodno je objasniti OpenGL podešavanja tj. tzv. OpenGL state machine.

OpenGL state machine se sastoji od stotina podešavanja koji utiču na mnoge aspekte renderinga. Pošto state machine ima ulogu u svemu što radimo veoma je bitno da znamo koja default podešavanja ona poseduje, kako da dodejmo do njih i kako da ih menjamo.

6.2 Funkcije podešavanja

OpenGL ima mnogo funkcija koje Vam omogućuju da saznate stanje nekog podešavanja u OpenGL state machine-i, većina ih počinje sa prefiksom: **glGet...**

NAPOMENA: Sve funkcije koje budemo koristili u radu sa OpenGL state machine-om zahtevaju da imate validan context kreiran !

6.3 Izvlačenje numeričkih stanja - podešavanja

Postoje četiri funkcije za generalnu upotrebu koje vam daju mogućnost da dobijete podešavanja koja sadrže numeričke ili boolean informacije. One su:

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

U svakoj od njih parametar **pname** određuje podešavanje koje tražimo, a params predstavlja niz dovoljno veliki da drži sve vrednosti vezane za to podešavanje koje ispitujemo.

Ne postoji funkcija tipa **glSet...** niti slična za podešavanje OpenGL state machine podešavanja. Umesto toga, postoji čitav set funkcija koje to rade na različite načine.

6.4 Paljenje i gašenje stanja - podešavanja OpenGL state machine-e

Dakle, kako onda izvršiti podešavanje ? Upotrebom **glEnable()** i **glDisable()** funkcija:

```
void glEnable(GLenum cap);  
void glDisable(GLenum cap);
```

Parametar **cap** pretstavlja OpenGL opciju koju želite da uključite ili isključite. Navešćemo neka od tih podešavanja: **GL_BLEND** (za blending operacije), **GL_TEXTURE_2D** (za 2d teksture), **GL_LIGHTING** (operacije sa svetlima).

glIsEnabled()

Često ćete želeći da saznate da li je neka opcija uključena ili ne. Iako to može biti urađeno pomoću **glGetBooleanv()** funkcije obično je mnogo lakše da to uradimo sa **glIsEnabled()**:

```
GLboolean glIsEnabled(GLenum cap);
```

Ova funkcija vraća **GL_TRUE** ukoliko je opcija tj. podešavanje uključeno ili **GL_FALSE** ukoliko nije.

6.5 Detalji OpenGL implementacije

Detalje Vaše OpenGL implementacije možete saznati pomoću sledeće funkcije:

```
const GLubyte *glGetString(GLenum name);
```

Vrednost koju će vratiti funkcija zavisi od parametra **name**. Neke od vrednosti koje on može imati su: **GL_VENDOR** (ime proizvođača hardvera), **GL_RENDERER** (model hardvera), **GL_VERSION** (verzija OpenGL-a) ...

6.6 Pronalaženje grešaka

Kada prosledite neispravan parametar funkciji error flag dobija određenu vrednost. Na primer kada funkcija nema ispravne parametre ona zapravo ne radi ništa, tada najčešće ne dobijate očekivane rezultate programa. Error flag će sadržati informacije o grešci koja je nastala. Te informacije ćemo izvući pomoću funkcije:

```
GLenum glGetError();
```

Vrednosti koje će Vam dati funkcija su sledeće: **GL_NO_ERROR** (default vrednost - nema nikakve greške u datom trenutku), **GL_INVALID_ENUM** (ukoliko prosledite vrednost f-ji. koju ona ne prima), **GL_INVALID_VALUE** (ukoliko prosledite vrednost van opsega koji prima f-ja), **GL_INVALID_OPERATION** (ukoliko prosledite parametre koji ne idu zajedno ili ukoliko za njih nemate odgovarajuću konfiguraciju npr. OpenGL state machine-e), **GL_STACK_OVERFLOW**, **GL_STACK_UNDERFLOW**, **GL_OUT_OF_MEMORY**, **GL_TABLE_TOO_LARGE** ...

6.7 Davanje smernica OpenGL-u

Prilikom razvoja aplikacija u OpenGL-u često ćete se sretati sa situacijama kada morate napraviti kompromis između kvaliteta i brzine. Pomoću funkcije **glHint()** moguće je OpenGL-u dati smernicu kako da tretira prikazivanje.

```
void glHint(GLenum target, GLenum hint);
```

Parametri funkcije su sledeći: **target** - određuje ponašanje koje želite da kontrolišete, **hint** parametar definiše smernicu koju prosleđujete OpenGL-u.

Ovo su neka od ponašanja koje biste mogli kontrolisati (parametar **target**): **GL_POINT_SMOOTH_HINT**, **GL_LINE_SMOOTH_HINT**, **GL_POLYGON_SMOOTH_HINT** - kvalitet glatkoće tačke, linije i poligona; **GL_FOG_HINT** - ukoliko je hint podešen na **GL_NICEST** izvršavaju se kalkulacije po pikselu, a ako je **GL_FASTEST** po vertex-u; **GL_PERSPECTIVE_CORRECTION_HINT** - određuje kvalitet boje i interpolacije;

Ovo su vrednosti koje možete dodeliti za **hint**: **GL_FASTEST**, **GL_NICEST**, **GL_DONT_CARE** - najbrži prikaz sa najlošijim kvalitetom, najbolji kvalitet sa sporijim prikazom i OpenGL sam određuje brzinu i kvalitet automatski.

6.8 Rad sa primitivama

Primitive su osnovni geometrijski elementi. U njih spadaju: tačke, linije i trouglovi.

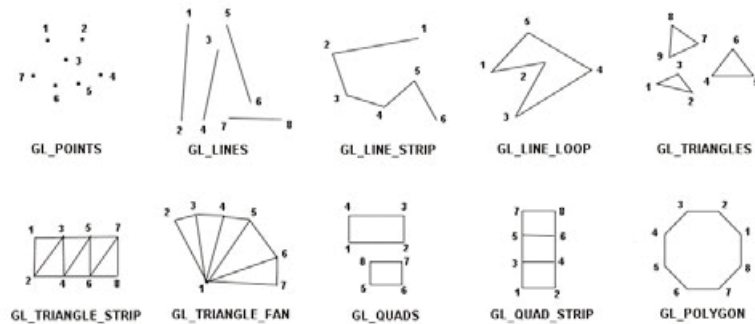
Da bismo krenuli sa njihovim crtanjem neophodno je pozvati **glBegin()** funkciju koja govori OpenGL-u da će biti započeto crtanje:

```
void glBegin (GLenum mode);
```

Funkcija prima argument **mode** koji predstavlja tip primitive:

GL_POINTS (tačke), **GL_LINES** (linije), **GL_LINE_STRIP** (serija vezanih linija), **GL_LINE_LOOP** (serija povezanih linija sa zatvorenim segmentom između prve i poslednje tačke), **GL_TRIANGLES** (trouglovi), **GL_TRIANGLE_STRIP** (serija povezanih trouglova), **GL_TRIANGLE_FAN** (serija trouglova oko zajedničkog centralnog vertex-a), **GL_QUADS** (četvorouglovi), **GL_QUAD_STRIP** (serija vezanih četvorouglova), **GL_POLYGON** (poligon).

Slika prikazuje osnovne primitive:



Svaki **glBegin()** poziv mora biti završen pozivom **glEnd()** funkcije koja ima sledeću formu:

```
void glEnd();
```

glEnd() nema nikakvih argumenata. Ona samo govori OpenGL-u da prikaže tražene primitive.

Između **glBegin()** / **glEnd()** funkcija ne mogu stajati sve OpenGL funkcije ! Ukoliko se upotrebi neka od funkcija koja nije predviđena generiše se **GL_INVALID_OPERATION** greška. Sledi lista funkcija koje mogu biti korišćene u bloku:

- glVertex*()** - postavlja koordinate vertex-a
- glColor*()** - postavlja trenutnu boju
- glSecondaryColor*()** - postavlja sekundarnu boju
- glIndex*()** - postavlja trenutnu index boju
- glNormal*()** - postavlja normal vector koordinate
- glTexCoord*()** - postavlja koordinate teksture
- glMultiTexCoord*()** - postavlja koordinate za multiztexturing
- glFogCoord*()** - postavlja koordinate magle
- glArrayElement()** - postavlja atribut za pojedinačni vertex na osnovu elemenata u nizu vertex-a
- glEvalCoord*()** - postavlja koordinate za bezier krive i površine
- glEvalPoint*()** - postavlja tačke za bezier krive i površine
- glMaterial*()** - postavlja svojstva materijala
- glEdgeFlag*()** - kontroliše prikaz ivica
- glCallList*()** - izvršava display listu
- glCallLists*()** - izvršava display liste.

Funkcija **glVertex()** ima brojne varijacije. Najčešće se koristi: **glVertex3f()** i prima tri float broja kao koordinate x, y i z.

Sledeće parče koda demonstrira upotrebu funkcije:

```
glVertex2i(5, 20); // crta vertex na (5, 20)
glVertex3f(1.5, 0.5, 10.0); // crta vertex na (1.5, 0.5, 10.0)
```

```
GLfloat v[3] = { 1.5, 0.5, 10.0 };  
glVertex3fv(v); // radi isto kao i prethodna dva primera samo sto cita iz niza
```

6.9 Tačke

Crtanje tačaka je veoma jednostavno. Sledeći primer koda to demonstrira:

```
// Jedna tacka  
glBegin(GL_POINTS);  
glVertex3f(0.0, 0.0, 0.0);  
glEnd();  
  
// Dve tacke - neefikasan nacin !!!  
glBegin(GL_POINTS);  
glVertex3f(0.0, 0.0, 0.0);  
glEnd();  
glBegin(GL_POINTS);  
glVertex3f(0.0, 1.0, 0.0);  
glEnd();  
  
// Prethodni kod sa dve tacke - efikasniji nacin  
glBegin(GL_POINTS);  
glVertex3f(0.0, 0.0, 0.0);  
glVertex3f(0.0, 1.0, 0.0);  
glEnd();
```

6.10 Menjanje veličine tače

Da biste promenili veličinu slike upotrebićete funkciju:

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

Rezultat je kvadrat dimenzija **size** x **size** pozicioniran na koordinate vertex-a. Default vrednost je 1.0. Ako je point antialiasing opcija ugašena (što i jeste dok sami to ne promenimo) veličina tačke će biti zaokružena na najbliži integer (sa minimalnom vrednošću od 1). Možete koristiti **glGet()** kako biste videli tekuću veličinu tačke. Sledi parče koda koje menja veličinu tačke:

```
// uzimamo trenutnu velicinu tacke  
GLfloat oldSize;  
glGetFloatv(GL_POINT_SIZE, &oldSize);  
// menjamo velicinu tacke po zelji  
if (oldSize < 1.0)  
    glPointSize(5.0);  
else  
    glPointSize(1.0);
```

6.11 Antialiasing tačaka

Iako možemo smeštati i kreirati primitive sa skoro beskonačnom preciznošću postoji ograničen broj piksela na ekranu. Ovo može prouzrokovati nazupčenost ivica. Antialiasing obezbeđuje glačanje kako bi one izgledale realističnije. Ako želite da koristite antialiasing morate ga uključiti. Za tačke treba proslediti **GL_POINT_SMOOTH** funkciji **glEnable()**:

```
//ako je antialiasing disabled - ukljuci ga !  
if (!glIsEnabled(GL_POINT_SMOOTH)) { glEnable(GL_POINT_SMOOTH); }
```

Kada smo to uradili ne mora da znači da smo dobili značajan broj piksela za prikaz ! Ograničenje i dalje postoji ! Ukoliko je dodeljena nepodržana vrednost za veličinu tačke, doćiće do zaokruživanja na prvi podržan broj. Pomoću funkcije `glGet()` možemo saznati opseg koji se može koristiti:

```
GLfloat sizes[2];
GLfloat granularity;
// uzimamo opseg
glGetFloatv(GL_POINT_SIZE_RANGE, sizes);
GLfloat minPointSize = sizes[0]; // najmanja dozvoljena velicina
GLfloat maxPointSize = sizes[1]; // najveća dozvoljena velicina
// a razmak između krajnjih velicina možemo dobiti ovako :
glGetFloatv(GL_POINT_SIZE_GRANULARITY, &granularity);
```

Važno je napomenuti da ukoliko nismo uključili blending opciju antialiasing ne radi !

6.12 Efekat udaljenosti

Logično tačke uvek zauzimaju isti prostor na ekranu bez obzira na to da li se nalaze dalje ili bliže od posmatrača. Za mnoge situacije, kao što su na primer sistemi čestica želećete da tačke postaju manje što su udaljenije. Ovo se može postići upotrebom `glPointParameter()` funkcije:

```
void glPointParameter{if}(enum pname, type param);
void glPointParameter{if}v(enum pname, const type *params);
```

Validni argumenti su ime parametra i vrednosti koje mu se pridružuju (`pname` i `params`).

Parametri koje je moguće podesiti:

`GL_POINT_SIZE_MIN` - minimalna veličina tačke, `GL_POINT_SIZE_MAX` - maksimalna veličina tačke, `GL_POINT_DISTANCE_ATTENUATION` - prima niz od tri vrednosti: a, b i c koji ulaze u formulu za izračunavanje faktora slabljenja - attenuation factor: $1/(a + b*d + c*d^2)$, gde d predstavlja udaljenost od tačke do oka. Default podešavanje je a=1, b=0 i c=0 što daje efekat - nema slabljenja, `GL_POINT_FADE_THRESHOLD` - koristi jednu vrednost koja određuje veličinu ispod koje OpenGL počinje da redukuje alpha vrednost tačke, dozvolivši Vam da postepeno nestane dok se smanjuje.

6.13 Primer tačke

Sledeći kod iscrtava tačke sa povećanjem njihove veličine kroz petlju:

```
float pointSize = 0.5;
// crta u liniji tacke - svaka sledeca tacka je veca od prethodne
for (float point = -4.0; point < 5.0; point+=0.5)
{
    // postavlja velicinu tacke
    glPointSize(pointSize);
    // crta tacku
    glBegin(GL_POINTS);
    glVertex3f(point, 0.0, 0.0);
    glEnd();
    // povecava velicinu tacke za sledecu tacku
    pointSize += 1.0;
}
```

6.14 Linije

Crtanje linija se ne razlikuje mnogo od crtanja tačaka:

```
glBegin(GL_LINES); // Ovoga puta koristimo GL_LINES
glVertex3f(-2.0, -1.0, 0.0); // I crtamo dva vertex-a za oba kraja linije
glVertex3f(3.0, 1.0, 0.0);
glEnd();
```

Kao i sa tačkama možemo nacrtati onoliko linija koliko to želimo. Za svaku liniju je potreban par vertex-a. Ukoliko ne dovršimo liniju drugim vertex-om (vertex-om kraja linije) ta linija neće biti nacrtana.

6.15 Modifikovanje debljine linije

Default vrednost za debljinu je 1.0. Da bismo videli tekuću debljinu koristimo:

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

Sledeće parče koda prikazuje konkretnu upotrebu:

```
// uzima trenutnu debljinu
GLfloat oldWidth;
glGetFloatv(GL_LINE_WIDTH, &oldWidth);
//ako je linija pretanka podebljava je ...
if (oldWidth < 1.0)
glLineWidth(5.0);
```

6.16 Antialiasing linija

Antialiasing linija funkcioniše slično tačkama. Sem uključivanja i gašenja antialiasing-a vodimo računa i o maksimalnoj i minimalnoj dozvoljenoj debljini linije:

```
// PRVO CEMO UKLJUCITI ANTIALIASING LINIJA !
glEnable(GL_LINE_SMOOTH); // KOD TACAKA SMO IMALI GL_POINT_SMOOTH
// A ZATIM ISPITATI OGRANICENJA ...
GLfloat sizes[2];
GLfloat granularity;
glGetFloatv(GL_LINE_WIDTH_RANGE, sizes);
GLfloat minLineWidth = sizes[0];
GLfloat maxLineWidth = sizes[1];
glGetFloatv(GL_LINE_WIDTH_GRANULARITY, &granularity);
```

6.17 Primer linija

Slično tačkama napravićemo kod koji će redom iscrtavati linije, svaka sledeća deblja od prethodne, odozgo na dole:

```
float lineWidth = 0.5;
// crta serije linije sa povecanjem njihove debljine kroz petlju
for (float line = 0.0; line < 7.0; line+=0.5)
{
// postavlja debljinu linije
glLineWidth(lineWidth);
// crta liniju
glBegin(GL_LINES);
```

```
glVertex3f(-5.0, 0.0, line-3.0);
glVertex3f(-1.0, 0.0, line-3.0);
glEnd();
// povecava debljinu za sledecu liniju
lineWidth += 1.0;
}
```

6.18 Crtanje poligona u 3 dimenzije

Poligone u trodimenzionalnom prostoru crtamo odredjivanjem koordinata njihovih vertex-a. Poligoni predstavljaju zatvorene celine koje je moguće popuniti bojom. Popunjavanje bojom je podrazumevano ponašanje. Pošto OpenGL state machine-a određuje kako će se rendering ponašati moguće je to ponašanje menjati. Kako bismo to izveli koristimo sledeću funkciju:

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

OpenGL razlikuje prednji i zadnji deo poligona odvojeno. Kao rezultat toga kada pozivamo `glPolygonMode()` moramo naznačiti da li se to odnosi na lice ili na zadnji deo poligona i zatim kao drugi parametar način na koji će biti obojen:

```
glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
```

`GL_FRONT` je prednji deo poligona, a `GL_BACK` zadnji deo. Oni mogu biti bojeni na tri načina: `GL_POINT` - prikazuju se samo tačke vertex-a, `GL_LINE` - prikazuju se samo linije, `GL_FILL` - unutrašnjost poligona je popunjena bojom.

Da biste videli koje podešavanje se trenutno koristi za bojenje poligona upotrebite kod:

```
glGet(GL_POLYGON_MODE);
```

6.19 Polygon Face Culling

Ako možemo da odredimo da je zadnja strana poligona okrenuta ka posmatraču (što bi bilo tačno za poligone na strani lopte suprotnoj od posmatrača) uštedeli bismo vreme na transformisanje i renderovanje pošto znamo da neke delove ne možemo videti. OpenGL to može učiniti za nas automatski kroz proces poznatiji kao **culling**. Da biste upotreбили culling morate ga prvo uključiti:

```
glEnable(GL_CULL_FACE);
```

Zatim, treba da naznačite koje strane treba da budu obrađene culling-om pomoću funkcije:

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

`mode` može biti `GL_FRONT` ukoliko želite da se prednji poligoni ne vide ili `GL_BACK` ukoliko želite da se ne vide zadnji poligoni - to je inače i default podešavanje. Takođe možete postaviti i `GL_FRONT_AND_BACK` što se u principu retko koristi pošto tada ne bi bilo ništa prikazano.

Sledeći korak je da kažemo OpenGL na koj način da odredi da li poligon okrenut spreda ili od pozadi. To se zove **polygon winding**. Gledajući poligon čelom napred možete odabrati bilo koji vertex koji će biti početak za njegovo opisivanje. Kako biste završili opisivanje morate ići ili u pravcu kazaljke na satu ili suprotno od ivice do ivice. OpenGL po default opcijama to radi suprotno od pravca kazaljke na satu. Način na koji ćete to raditi možete zadati pomoću funkcije:

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

mode može biti ili **GL_CCW** ili **GL_CW** (suprotno ili u pravcu kazaljke sata).

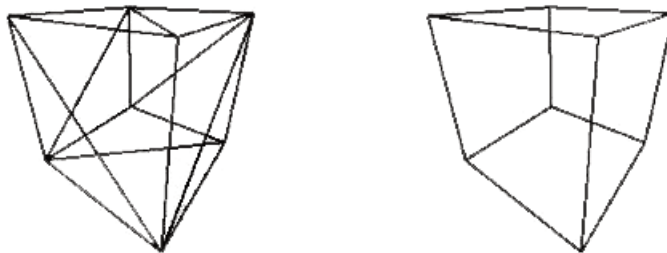
Winding se ne koristi samo kod culling-a već i u drugim delovima OpenGL-a kao što je rad sa svetlima.

6.20 Sakrivanje ivica poligona

Nije čest slučaj da ćete imati želju da nešto prikazete kao wireframe. Tada obično sklanjamo suvišne ivice poligona i ostavljamo samo glavne. To ćemo postići upotrebom funkcija:

```
void glEdgeFlag(GLboolean isEdge);
void glEdgeFlagv(const GLboolean *isEdge);
```

Jedina razlika među njima je što prva prima jednu boolean vrednost, a druga pokazivač na niz. Ako je prosledjena vrednost **GL_TRUE** sve ivice se prikazuju, a ako je **GL_FALSE** ne.



6.21 Antialiasing poligona

Evo primera koji će demonstrirati upotrebu antialiasing-a na poligonima. Upotreba je jako slična linijama i tačkama:

```
// ako je opcija ugašena uključujemo je ...
if (!glIsEnabled(GL_POLYGON_SMOOTH))
    glEnable(GL_POLYGON_SMOOTH);
```

6.22 Trouglovi

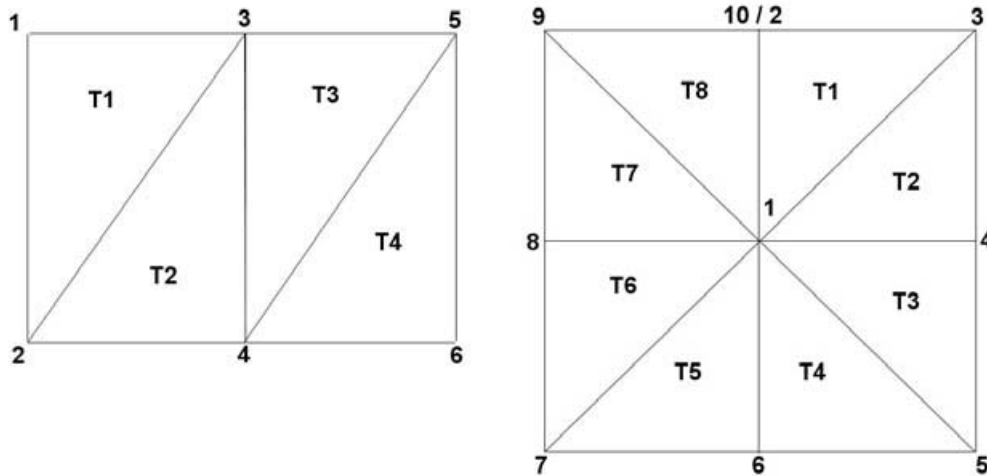
Trouglovi su najčešće korišćen tip poligona iz više razloga: stranice su uvek koplanarne, pošto tri tačke definišu ravan, trouglovi su uvek konveksni, trouglovi se ne mogu prekrstiti sami sa sobom. Ako pokušate da renderujete poligon koji krši bilo koje od ova tri pravila dobićete nepredvidljive rezultate. Crtanje trougla je jako jednostavno:

```
glBegin(GL_TRIANGLES);
glVertex3f(-2.0, -1.0, 0.0);
glVertex3f(3.0, 1.0, 0.0);
glVertex3f(0.0, 3.0, 0.0);
glEnd();
```

Kao i sa linijama možemo crtati više trouglova od jednom - tri vertex-a po tri vertex-a. Ukoliko zafali bilo koji vertex do tri taj trougao neće biti nacrtan.

Kada bi hteli da nacrtamo trouglove koji dele stranice upotrebili smo **GL_TRIANGLE_STRIP**. Tada se deljeni vertex-i ne unose dva puta pa možemo u bloku **glBegin()** / **glEnd()** imati broj **glVertex3f()** funkcija i da ne bude deljiv sa tri.

Kada bismo hteli da crtamo redom trouglove koji dele zajednički centralni vertex upotrebili bismo **GL_TRIANGLE_FAN**. Crtanje takvih trouglova takođe ne zahteva broj **glVertex3f()** funkcija deljiv sa tri. Trouglovi dele zajednički centralni vertex i nastavljaju se jedan na drugi.



Slika iznad prikazuje sa leva na desno redom: **GL_TRIANGLE_STRIP** i **GL_TRIANGLE_FAN**.

6.23 Četvorouglovi

Četvorouglovi ili **GL_QUADS** koriste se za crtanje kvadrata i pravougaonika. Radi se kao i sa prethodnim primitivama. Moguće je kreirati i **GL_QUAD_STRIP**.

6.24 Mnogouglovi

Mnogougao se crta definisanjem proizvoljnog broja vertex-a većeg od dva. Prvi i poslednji vertex se spajaju u jednu ivicu. Parametar za **glBegin()** je **GL_POLYGON**.

6.25 Atributi

Atribut grupa je set stanja varijabli koje OpenGL svrstava u grupu. Na primer, line grupa se sastoji od svih atributa vezanih za crtanje linija. Upotrebom **glPushAttrib()** i **glPopAttrib()** funkcija možete snimiti i vratiti sve informacije i podešavanja vezane za grupu:

```
void glPushAttrib(GLbitfield mask);
void glPopAttrib(void);
```

glPushAttrib() čuva sve atribute određene mask parametrom u atribut stack. **glPopAttrib()** vraća podešavanja iz stack-a. Ovo su neke moguće vrednosti mask paramtera:

- GL_ALL_ATTRIB_BITS** - sva OpenGL stanja i varijable
- GL_ENABLED_BIT** - enable-ovane state varijable
- GL_FOG_BIT** - varijable vezane za maglu
- GL_LIGHTING_BIT** - varijable vezane za svetlo
- GL_LINE_BIT** - linije
- GL_POINT_BIT** - tačke
- GL_POLYGON_BIT** - poligoni
- GL_TEXTURE_BIT** - teksture.

7. Geometrijske transformacije i matrice

7.1 Razumavanje transformacija

Došao je trenutak da prestanemo sa kreiranjem objekata i da se fokusiramo na to kako ih pomerati u virtuelnom svetu. OpenGL omogućuje programerima lako manipulisanje objektima u prostoru upotrebom različitih koordinatnih transformacija. Postoje tri glavna tipa transformacija koje se koriste između trenutka kada zadajete tačke objekata i trenutka kada se oni pojavljuju na ekranu: **viewing**, **modeling** i **projection**.

OpenGL terminologija transformacija obuhvata:

Viewing - određuje lokaciju kamere

Modeling - pomera objekte po sceni

Modelview - predstavlja dualitet viewing i modeling transformacija

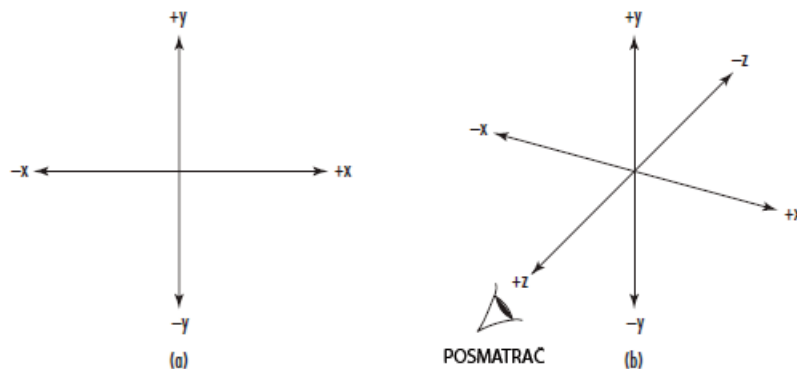
Projection - određuje oblik i veličinu volumena pogleda

Viewport - razvlači konačan pogled na prozor.

Sve transformacije se izvršavaju po redosledu !

7.2 Koordinate oka

Koordinate oka predstavljaju koordinate sa kojih se posmatra i bez obzira na transformacije koje se mogu dogoditi možemo ih posmatrati kao aposloutne kordinate ekrana. Zbog toga, koordinate oka predstavljaju virtuelni fiksirani kordinatni sistem koji se koristi kao osnovna referenca. Na sledećem prikazu možemo videti odnos posmatrača i koordinatnog sistema - pozitivna z osa Kartesienovog koordinatnog sistema koji se koristi u OpenGL-u se nalazi naspram posmatrača:



7.3 Viewing transformacija

Viewing transformacija je prva koja se primenjuje na Vašu scenu. Koristi se da odredi početnu tačku posmatranja scene. Po default-u, tačka posmatranja je 0, 0, 0 i gleda ka negativnoj z osi u dubinu monitora. Tačka posmatranja se pomera relativno u odnosu na koordinatni sistem oka kako bismo odredili početnu tačku posmatranja. Kada je tačka posmatranja locirana na početku koordinatnog sistema (0, 0, 0 - **origin**) gledano iz perspective projekcije pozitivne z vrednosti koordinatnog sistema su iza posmatrača. U orthographic projekciji pretpostavlja se da je posmatrač jako udaljen u pozitivnom smeru z ose i može videti sve u viewing volumenu.

Viewing transformacija omogućuje da postavite pogled bilo gde u sceni i u bilo kom pravcu. Određivanje pogleda je kao postavljanje kamere u scenu.

7.4 Modeling transformacije

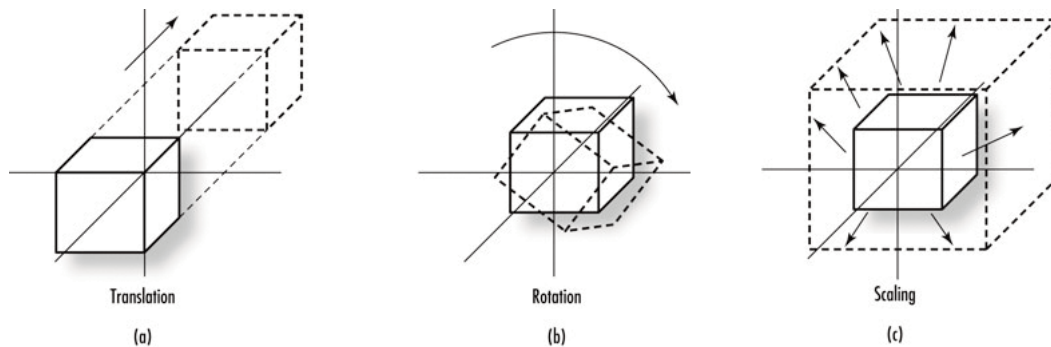
Modeling transformacije se koriste za manipulisanje modelima i objektima u njima. Ove transformacije pomeraju objekte u prostoru, rotiraju ih i razvlače i skupljaju po potrebi. Konačan prikaz u sceni zavisi od redosleda promena na objektu !

Translacija (a) - predstavlja pomeranje objekta po određenom vektoru.

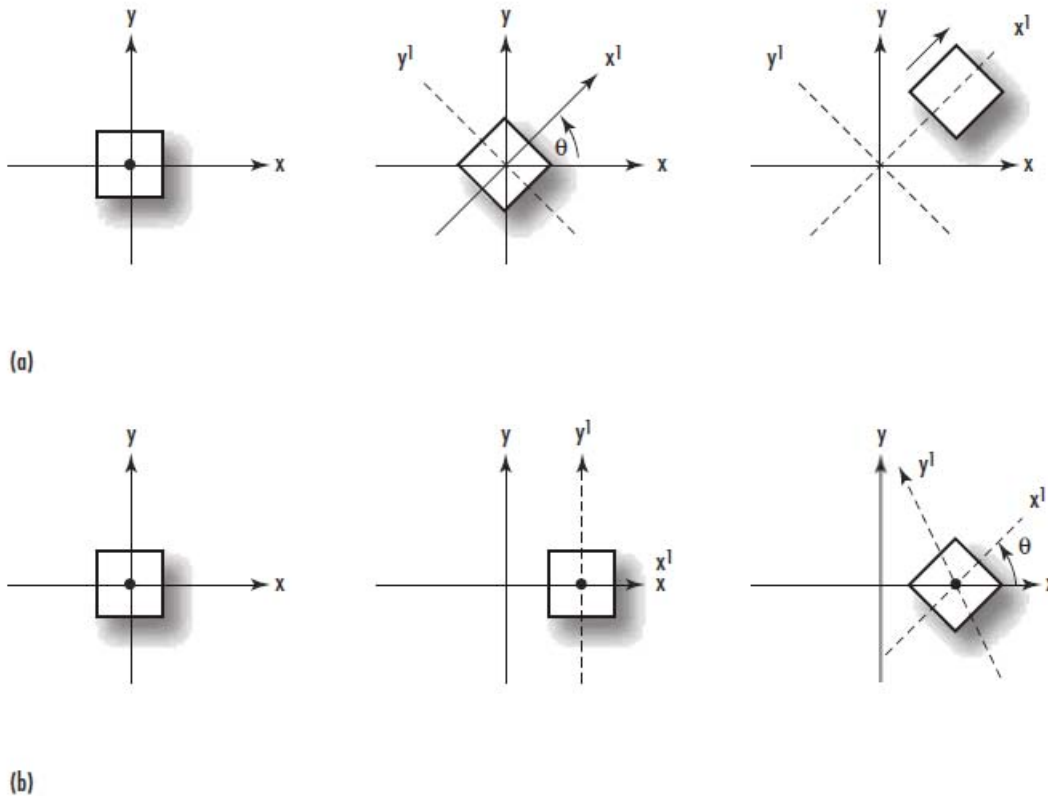
Rotacija (b) - rotiranje objekta oko vektora.

Skaliranje (c) - povećavanje i smanjivanje dimenzija objekta. Sa skaliranjem možete odrediti različite vrednosti po različitim osama čime možemo rastezati i skupljati objekte. Takođe upotrebom negativnih vrednosti moguće je postići mirroring efekat.

Modeling transformacije:



Efekat redosleda modeling transformacija (a - rotacija pa translacija)(b - translacija pa rotacija):



7.5 Modelview dualitet

Viewing i modeling transformacije su u suštini iste ako gledamo unutrašnji efekat i ako gledamo efekat na ekranu. Razlika između njih je u ubedjenju programera. Ne postoji prava vizuelna razlika između pomeranja objekta napred i pomeranju reference sistema napred. Viewing transformacije je jednostavno kao-modeling transformacija koja se primenjuje na celu scenu gde objekti u sceni često imaju svoje sopstvene modeling transformacije primenjene posle viewing transformacija. Termin modelview indicira da te dve transformacije su kombinovane u transformation pipeline-u u jedinstvenu matricu - modelview matricu.

Viewing transformacija je u suštini ništa nego transformacija primenjena na virtualni objekat - posmatrača pre crtanja drugih objekata. Transformacije daju reference na kojima se dalje yasnivaju druge transformacije.

7.6 Projection transformacija

Projection transformacija se primenjuje posle modelview transformacije. Ona u stvari utvrđuje viewing volumen i clipping ravni. Clipping ravni određuju granice vidljive geometrije koje posmatrač može videti. Ova transformacija određuje kako će se gotova scena prikazati na ekranu. Postoje dva tipa: **orthographic** i **perspective**.

Ortographic projekcija prikazuje 3d svet u njegovim pravim dimenzijama bez obzira na udaljenost kamere. **Perspective** projekcija deformiše objekte kao u realnom životu - prikazuje ih manje sa povećanjem udaljenosti i sve veće što smo im bliži.

7.7 Viewport transformacije

Kada je sve podešeno i urađeno završavamo sa dvodimenzionalnom projekcijom vaše scene koja će biti mapirana na prozor negde na ekranu. Mapiranje na koordinate fizičkog prozora je poslednja stvar koju treba uraditi i ostvaruje se upotrebom viewport transformacije.

7.8 Rad sa matricama

Svaka transformacija se postiže množenjem matrice koja sadrži koordinate matricom koja opisuje transformaciju. Tako se sve transformacije opisuju kao proizvod množenja jedne ili više matrica.

Jedan red brojeva se zove vektor. Matrice i vektori su veoma bitni u radu sa kompjuterskom grafikom. Pri radu ćemo često koristiti i termin skalar - skalar je jedna jedini broj - običan broj. Matrice se mogu međusobno množiti ali i vektorom ili skalarom. Množenje tačke (vektora) matricom (transformacija) daje novu transformisanu tačku (vektor).

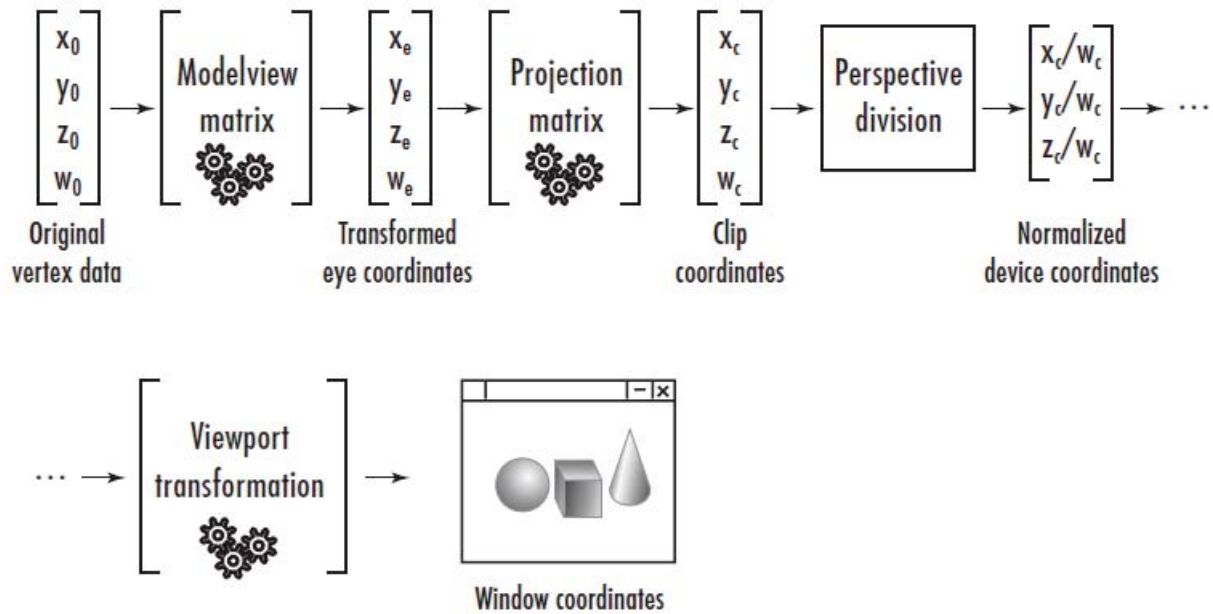
7.9 Transformation pipeline

Put od sirovih koordinata vertexa do ekrana je dug.

Prvo, koordinate vertexa se konvertuju u matricu 1x4. Prva tri broja su redom koordinate x, y i z, a četvrti broj scaling faktor - i obično je 1.0. Vertex se zatim množi sa modelview matricom koja drži koordinate oka. Dobijeni rezultat - matrica - se množi zatim projection matricom čime se dobija rezultat - clip koordinate. Četvrta - w - vrednost se može promeniti u zavisnosti od transformacija koje se izvrše.

Poslednji korak je mapiranje na 2d ravan tj množenjem viewport matricom čije vrednosti zavise od **glViewport()** funkcije.

Ceo proces je prikazan na slici:



7.10 Modelview matrica

Modelview matrica je matrica dimenzija 4x4 i predstavlja transformisani koordinatni sistem koji koristimo da bismo pozicionirali i orijentisali objekte. Koordinate vertexa koji predstavlja matrica 1x4 se množe modelview matricom i dobija se vertex predstavlja matricom 1x4 u novom koordinatnom sistemu.

7.11 Translacija

Pokušajmo da nacrtamo kocku, a zatim da je pomerimo za 10 jedinica po y osi:

```
// konstruisanje natrice za translaciju za 10 jedinica po y osi
...
// mnozenje modelview matricom
...
// crtanje kocke
glutWireCube(10.0f);
```

OpenGL ima funkciju koja će obaviti posao množenja matrica i pozicioniranje za 10 jedinica (u ovom slučaju):

```
void glTranslatef(GLfloat x, GLfloat y, GLfloat z);
```

dakle:

```
// Translacija po y osi za 10 jedinica
glTranslatef(0.0f, 10.0f, 0.0f);
// crtanje kocke
glutWireCube(10.0f);
```

Ova funkcija uzima parametra koji predstavljaju vrednosti translacije po x, y i z osi. Ona konstruiše matricu koja predstavlja transformaciju i množi je tekućom matricom iz stack-a.

7.12 Rotacija

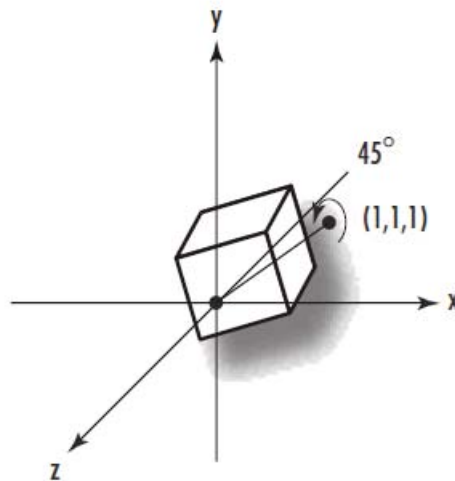
`glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);`

rotacija se postiže rotacijom oko vektora koji određuju **x**, **y** i **z** argumenti. Ugao rotacije je smer suprotan kazaljci sata meren u stepenima i određen argumentom **angle**. U najjednostavnijem slučaju rotacija se obavlja oko samo jedne ose: x, y ili z.

Da bismo videli osu rotacije moramo povući liniju od početka koordinatnog sistema (origin) do tačke koju predstavljaju argumenti **x**, **y** i **z**. Sledeće parče koda rotira kocku za 45 stepeni oko ose koju određuju vrednosti argumenata 1, 1, 1:

```
// Izvršavamo rotaciju
glRotatef(45.0f, 1.0f, 1.0f, 1.0f);
// Crtamo kocku
glutWireCube(10.0f);
```

Ono što zapravo dobijamo je:



7.13 Skaliranje

Skaliranje predstavlja širenje i skupljanje objekata širenjem ili skupljanjem njihovih stranica. Funkcija za rad sa skaliranjem je:

`glScalef(GLfloat x, GLfloat y, GLfloat z);`

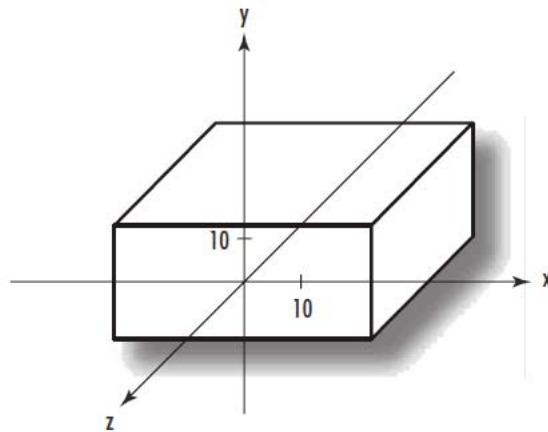
Ona množi x, y i z vrednosti faktorom koji je prosledjen funkciji za odgovarajuću koordinatu.

Primer kod skaliranja:

```
// Izvršavanje transformacije skaliranja
glScalef(2.0f, 1.0f, 2.0f);

// Crtanje kocke - u stvari sada vec kvadra !
glutWireCube(10.0f);
```

Dakle ono što dobijamo je kvadar, a ne kocka:

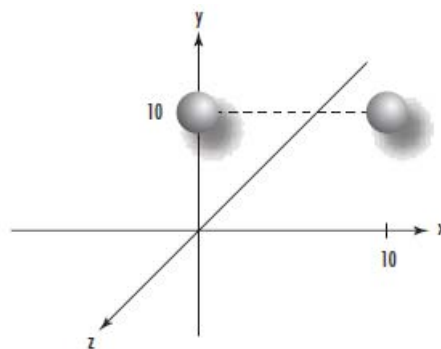


7.14 Identity matrica

Pretpostavimo da želite da nacrtate dve sfere. Jedna na 10 jedinica y ose, a druga na 10 x ose. Pretpostavimo da biste napisali ovakav kod:

```
// Idi na 10 jedinica y ose
glTranslatef(0.0f, 10.0f, 0.0f);
// Nacrtaj prvu sferu
glutSolidSphere(1.0f, 15, 15);
// Idi na 10 jedinica x ose
glTranslatef(10.0f, 0.0f, 0.0f);
// Nacrtaj drugu sferu
glutSolidSphere(1.0f);
```

Međutim, to ne daje baš sfere na mestima koja smo očekivali ! Šta dobijamo ? Rezultat ovakvog koda je na slici:



Svaki poziv `glTranslatef()` funkcije je kumulativan na modelview matricu ! Dakle, iz tog razloga druga sfera završava na poziciji 10, 10, 0. Mogli biste da pozivate redom funkciju - prvo se vratite nazad za deset jedinica, a onda postavite na novi položaj ... To bi pojele mnogo CPU i GPU resursa. Zato koristimo funkciju za promenu tekuće matrice: `glLoadIdentity();`

Ovime govorimo da ne treba da resetujete modelview matricu na origin (početak koordinatnog sistema).

Proces translacije sa identity matricom se matematički može prikazati ovako:

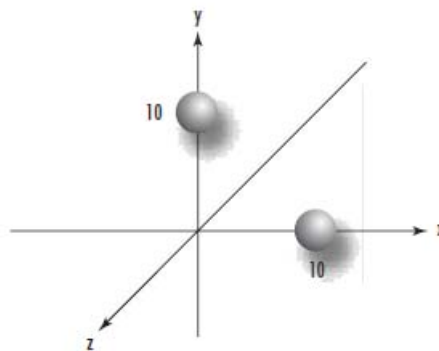
$$\begin{bmatrix} 8.0 \\ 4.5 \\ 2.0 \\ 1.0 \end{bmatrix} \begin{bmatrix} 1.0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 1.0 \end{bmatrix} = \begin{bmatrix} 8.0 \\ 4.5 \\ 2.0 \\ 1.0 \end{bmatrix}$$

Prva matrica 4x1 predstavlja translaciju a druga 4x4 identity matricu što za rezultat daje matricu 4x1 sa koordinatama koje su nam potrebne.

Dakle, dve sfere je trebalo crtati ovako:

```
// resetuje modelview
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
// ide na prvi položaj
glTranslatef(0.0f, 10.0f, 0.0f);
// crta prvu sferu
glutSolidSphere(1.0f, 15, 15);
// resetuje modelview
glLoadIdentity();
// ide na drugi položaj
glTranslatef(10.0f, 0.0f, 0.0f);
// crta drugu sferu
glutSolidSphere(1.0f, 15, 15);
```

Konačno dobijamo željeni prikaz :



7.15 Matrix stack

Resetovanje modelview-a pre crtanja nije baš uvek poželjno. Često ćete želeti da sačuvate trenutno stanje modelview-a kako biste ga kasnije ponovo pozvali i koristili. OpenGL poseduje **matrix stack**. U njega se mogu skladištiti matrice sa određenim stanjima. Možemo ga puniti tako što dodajemo matricu konadom **glPushMatrix()** ili je vratiti tj. prvu odozgo iz stacka komando **glPopMatrix()**. OpenGL ima matrix stack za modelview i matrix stack za projection matrice. Zavisno od operativnog sistema koji koristite moguće je u njega uneti ograničen broj matrica. U windows-u je ograničenje za modelview 32 matrice i 2 za projection. Podatak o ograničenju možete dobiti funkcijama:

```
glGet(GL_MAX_MODELVIEW_STACK_DEPTH);
ili
glGet(GL_MAX_PROJECTION_STACK_DEPTH);
```

Ako prekoračite ograničenje nastaje grška: **GL_STACK_OVERFLOW**, a u slučaju ako pokušate da povratite matricu iz praznog stack-a nastaje **GL_STACK_UNDERFLOW**.

7.16 Projekcije

Definisanjem projekcija biramo način prikaza i selekciju šta će biti prikazano u konačnom prikazu, a šta ne. Time se mogu uštedeti značajni resursi kompjutera.

Pre nego što izvršimo bilo koju projekcionu transformaciju potrebno je odabrati projection matricu:

```
glMatrixMode(GL_PROJECTION);
```

U većini slučaja ćete zatim identifikovati matricu pozivom **glLoadIdentity()** funkcije i resetovati je. Kada ste odabrali projection matricu spremni ste da podesite projekciju.

7.17 Ortographic projekcija

Funkcija za podešavanje je:

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

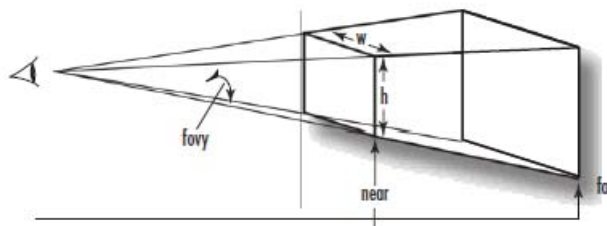
left i **right** određuju x-koordinate clipping ravni, **near** i **far** udaljenost z-koordinate clipping ravni i **top** i **bottom** y-koordinate clipping ravni. Tako se definiše prostor oblika kocke.

7.18 Perspective projekcija

Shodno prirodi perspective projekcije sasvim je logično da clipping volumen neće biti oblika kocke. Funkcija koja manipuliše ovakvom projekcijom je:

```
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);
```

Parametri **gluPerspective()** funkcije su nešto složeniji:



fovy - **field of view angle** - ugao posmatrača, **aspect** - odnos širine i visine prednje clipping ravni, **near** - udaljenost prednje clipping ravni i **far** udaljenost zadnje clipping racni.

7.19 Napredna manipulacija matricama

OpenGL predstavlja matrice kao nizove od 16 članova:

```
GLfloat matrix[16]; // odgovara matrici 4x4
```

Svaki član ima odgovarajuće mesto u matrici:

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix}$$

Prva tri stuba matrice su direkcioni vektori koji predstavljaju orijentaciju x, y i z osau prostoru. U većini slučajeva one su jedna u odnosu na drugu pod uglovima od 90 stepeni.

$$\begin{array}{c} \text{X axis direction} \\ \downarrow \\ x_x \\ x_y \\ x_z \\ 0 \end{array} \quad \begin{array}{c} \text{Y axis direction} \\ \downarrow \\ y_x \\ y_y \\ y_z \\ 0 \end{array} \quad \begin{array}{c} \text{Z axis direction} \\ \downarrow \\ z_x \\ z_y \\ z_z \\ 0 \end{array} \quad \begin{array}{c} \text{Translation/location} \\ \downarrow \\ t_x \\ t_y \\ t_z \\ 1 \end{array}$$

Poslednji stub predstavlja translaciju. Obično ga čine sve nule i poslednji član koji je 1.

Najimpresivnija stvar kod matrice 4x4 je da pošto sadrži podatke o poziciji i orijentaciji koordinatnih sistema je da kada njome pomnožimo bilo koji vertex on prelazi u taj koordinatni sistem. Odatle sledi da po tom principu možemo čitave objekte i scene prebacivati u druge koordinatne sisteme tj. iz koordinatnog sistema u koordinatni sistem.

7.20 Hardverske transformacije

Većina OpenGL implementacija u sebi imaju hardverske transformacije i svetla. To znači da transformisanje matrica - posebno kada imamo hiljade i hiljade vertexa - tj. hiljade i hiljade operacija množenja matrica naš specijalizovani hardver obavlja te operacije rasterećujući CPU putem hardverske akceleracije. GPU je inače mnogo brži od CPU-a !

7.21 Učitavanje matrica

Pošto sada znamo kako funkcioniše cela priča i kako radimo sa matricama možemo pokušati da za tekuću učitamo sopstvenu matricu. To ćemo učiniti `glLoadMatrix(GLfloat m)` funkcijom gde `m` predstavlja niz od 16 `GLfloat` članova.

Ukoliko umesto niza float-ova prosledimo niz double-ova možemo platiti skupu kaznu po ceni performansi zato što ćemo značajnu procesorsku snagu uložiti u konverziju float-ova !

Sledeći primer učitava matricu koju mi želimo. Efekat je isti kao upotreba funkcije `glLoadIdentity()` :

```
// učitava identity matricu
GLfloat m[] = { 1.0f, 0.0f, 0.0f, 0.0f, // X stub
0.0f, 1.0f, 0.0f, 0.0f, // Y stub
0.0f, 0.0f, 1.0f, 0.0f, // Z stub
0.0f, 0.0f, 0.0f, 1.0f }; // translacija
glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(m);
```

7.22 Množenje matrica

Da bismo trenutnu matricu pomnožili našom matricom koju smo sami definisali možemo upotrebiti funkciju:

```
glMultMatrixf(m);
```

gde je **m** matrica koju smo sami kreirali. Važno je napomenuti da upotrebom ove funkcije OpenGL koristi hardversku akceleraciju tj. proces množenja matrica obavlja procesor grafičke karte (GPU). Generalno, korišćenjem OpenGL funkcija iskoristićemo maksimum našeg grafičkog hardvera i rasteretiti CPU !

7.23 Podešavanje Viewport-a

Konačno viewport transformacija se dešava posle projection. Sada je vreme da pričamo o njoj. U suštini ona određuje 2d prostor (prozor) u kome će se renderovati. Sprovodi se funkcijom **glViewport()**:

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

x i **y** su koordinate donjeg levog ugla, a **width** i **height** širina i visina. Kada kreiramo rendering kontekst i pridružen prozoru viewport je automatski podešen prema dimenzijama prozora. Međutim, možemo pozivati funkciju kako bismo naknadno menjali njegove dimenzije - slučaj reshape prozora.

7.24 Manipulacija pogledom (viewpoint manipulation)

Kamera predstavlja posmatrača. OpenGL ima nekoliko funkcija koje omogućuju laku manipulaciju pogledom. Jedna od njih je **gluLookAt()**:

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

Prvi set od tri parametra predstavlja koordinate kamere, gde bi 0, 0, 0 bio na primer origin. Sledeća tri pravac u kom kamera gleda, a poslednja tri predstavljaju su vektor koji govori gde je pravac na gore.

Nedostatak ove funkcije je to da morate koristiti GLUT ! Šta raditi ako ne želimo da koristimo GLUT ?

Jedno rešenje je upotreba **glTranslate()** ili **glRotate()** funkcija:

```
void DisplayScene()
{
    glClear(GL_COLOR_BUFFER_BIT); // clear the color buffer
    glColor3f(1.0f, 0.0f, 0.0f); // postavlja crvenu boju za crtanje
    glLoadIdentity(); // cisti trenutnu matricu
    // zatim pomera kameru na zeljeno mesto
    glTranslatef(0.0f, 0.0f, -10.0f); // udaljujemo se za 10 jedinica
    // tek sada crta
    glBegin(GL_TRIANGLE);
    glVertexf(10.0f, 0.0f, 0.0f);
    glVertexf(0.0f, 10.0f, 0.0f);
    glVertexf(-10.0f, 0.0f, 0.0f);
    glEnd();
}
```

7.25 Sastavimo sve zajedno

Podestićemo se početka izlaganja ovog dela rada:

OpenGL terminologija transformacija obuhvata:

Viewing - određuje lokaciju kamere

Modeling - pomera objekte po sceni

Modelview - predstavlja dualitet viewing i modeling transformacija

Projection - određuje oblik i veličinu volumena pogleda

Viewport - razvlači konačan pogled na prozor.

Sve transformacije se izvršavaju po redosledu !

Sledeće parče koda skreće pažnju na korake pri radu sa transformacijama i matricama i kao što se može uočiti **redom se izvršavaju**:

```
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0f, 0.0f, 0.0f);
// definisanje pogleda
glLoadIdentity();
glTranslatef(0.0f, 0.0f, -10.0f); // udaljujemo se za 10 jedinica
// definisanje objekata
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0f, 10.0f, 0.0f);
glutSolidSphere(1.0f, 15, 15);
glLoadIdentity();
glTranslatef(10.0f, 0.0f, 0.0f);
glutSolidSphere(1.0f, 15, 15);
// definisanje projekcije
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, 800, 0, 600, -50.0, 50.0);
// definisanje viewport-a
glViewport(0, 0, 800, 600);
```

8. Rad sa senčenjem, svetlima i materijalima

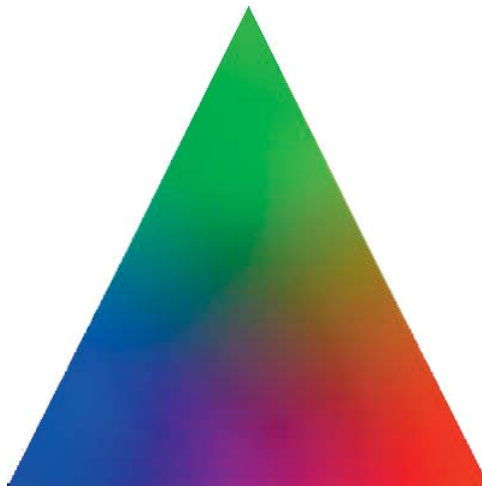
8.1 Postavljanje Shading modela

Da bismo objasnili čemu služi senčenje u OpenGL-u napisaćemo jedno malo parče koda koje ga koristi:

```
// Enable smooth shading
glShadeModel(GL_SMOOTH);
// Crtamo trougao
glBegin(GL_TRIANGLES);
// Crvena tacka
glColor3ub((GLubyte)255,(GLubyte)0,(GLubyte)0);
glVertex3f(0.0f,200.0f,0.0f);
// Zelena
glColor3ub((GLubyte)0,(GLubyte)255,(GLubyte)0);
glVertex3f(200.0f,-70.0f,0.0f);
// Plava
glColor3ub((GLubyte)0,(GLubyte)0,(GLubyte)255);
glVertex3f(-200.0f, -70.0f, 0.0f);
glEnd();
```

Prva linija zapravo uključuje OpenGL senčenje - Smooth Shading. Po default-u opcija je uključena, međutim nije loša ideja da je za svaki slučaj potvrdimo. Drugi mogući režim senčenja je **GL_FLAT** i znači da neće biti izvršena nikakva Shading kalkulacija.

OpenGL kao rezultat izvršavanja ovakvog koda prikazuje trougao isenčen od tačke do tačke prelaskom iz boje u boju.



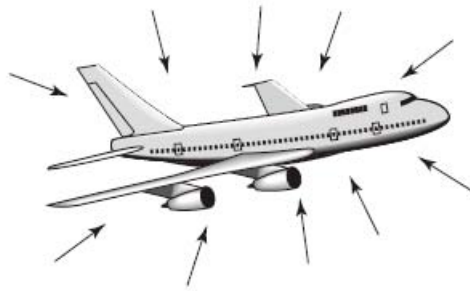
8.2 Svetla

U OpenGL-u moguć je rad sa svetlima koji objektima koje senčimo daju dodatnu realističnost. Postoje tri vrste svetla: **ambient**, **diffuse** i **specular**.

8.3 Ambient svetlo

Ambientno svetlo je svetlo koje ne dolazi sa bilo kog određenog pravca. Ima svoj izvor negde ali nje-

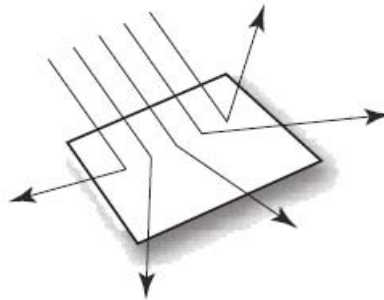
govi zraci padaju svuda. Objekti osvetljeni ovakvim svetlom su jednako osvetljeni na svim tačkama svoje površine.



8.4 Diffuse svetlo

Diffuse svetlo dolazi iz određenog pravca i udara u površinu objekta. Ovakvo svetlo se zatim odbija o površinu proporcionalno uglu pod kojim pada. Objekat je jače osvetljen što svetlo direktnije pada na površinu.

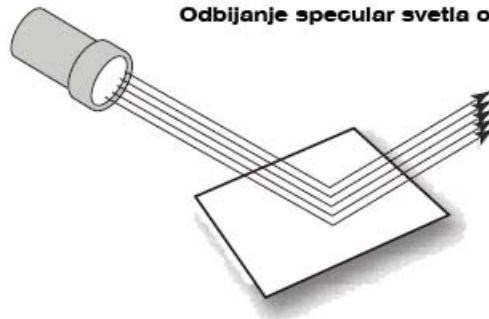
Odbijanje zraka diffuse svetla o površinu



8.5 Specular svetlo

Kao diffuse, specular je direkciono svetlo. Međutim, ono više utiče na površinu od diffuse svetla. Ono prouzrokuje svetle tače na mestima gde pada, zato se još zove i specular highlight. Zbog svoje direkcione prirode ono čak zavisi i od ugla pod kojim posmatrač posmatra scenu.

Odbijanje specular svetla o površinu



8.6 Spojimo ih sve zajedno

Ni jedno svetlo u OpenGL-u nije 100% ni jedan od tri pomenuta tipa, već se sa sastoji od osobina svakog koje variraju od potreba.

Zbog toga, svetlo se sastoji od tri lighting komponente: **ambient, diffuse i specular**. Baš kao i komponente boje ovi atributi imaju 4 osnovna parametra: R, G, B, A. Za svrhe boje svetla alpha komponenta se ignoriše !

Na primer svetlo crvenog lasera bi imalo sledeće parametre:

	Red	Green	Blue	Alpha
Specular	0.99	0.0	0.0	1.0
Diffuse	0.10	0.0	0.0	1.0
Ambient	0.05	0.0	0.0	1.0

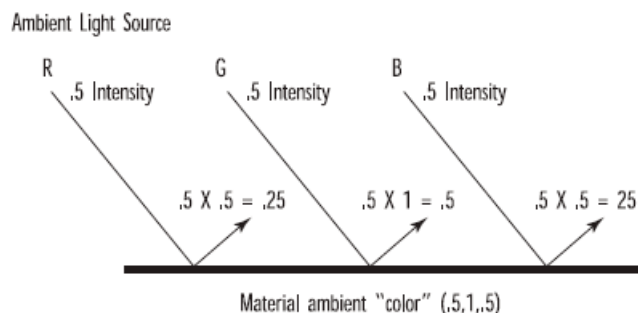
8.7 Materijali

Kada radimo sa svetlima obično poligonima ne dajemo samo boju već i određena svojstva kako bismo dobili napredne vizuelne performanse. Ova svojstva se zovu materijali.

8.8 Svojstva materijala

Kao i svetla materijali imaju tri osnovna svojstva: **ambient, diffuse i specular**. Zavisno od tri atributa svetla i tri atributa materijala pikseli dobijaju finalnu boju. Boja piksela se dobija složenim proračunima parametara. Uzmimo na primer svetlo koja ima tri vrednosti za ambient: 0.5, 0.5, 0.5 i materijal koji ima svojstva za ambient: 0.5, 1.0 i 0.5. Kako dobijamo konačnu ambient vrednost? Jednostavno:

$$(0.5 * 0.5, 0.5 * 1.0, 0.5 * 0.5) = (0.25, 0.5, 0.25)$$



Rezultat se dobija množenjem svakog ambient parametra svetla odgovarajućim ambient parametrom materijala. Kao rezultat se dobija konačna ambient boja piksela. Slično je i sa diffuse i specular atributima.

8.9 Postavljanje svojstava materijala

Sada je vreme da prethodno rečeno praktično primenimo - parče koda koje sledi prikazuje kako se definiše materijal i svetlo neophodno kako bi se on video:

```
// Jako belo svetlo – puni inteziteti RGB vrednosti
GLfloat ambientLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };
// Enable lighting
glEnable(GL_LIGHTING);
// Postavljamo light model da koristi ambient light koji odredjuje ambientLight[] array
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLight);
GLfloat gray[] = { 0.75f, 0.75f, 0.75f, 1.0f };
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray); // sencimo prednju stranu poligona
// a zatim vrednosti gray niza dodeljujemo i ambient i diffuse osobinama materijala
```

```
// zatim crtamo
glBegin(GL_TRIANGLES);
glVertex3f(-15.0f,0.0f,30.0f);
glVertex3f(0.0f, 15.0f, 30.0f);
glVertex3f(0.0f, 0.0f, -56.0f);
glEnd();
```

Dati primer je vrlo jednostavan, zadajemo paraneatre, a OpenGL za nas vrši proračun i konačan izlaz na ekran.

Međutim, postoji i drugi način zadavanja osobina materijala. Taj način se zove **color tracking**. Na prvi pogled zahteva više koda, a zapravo troši manje procesorske snage:

```
// Enable color tracking
glEnable(GL_COLOR_MATERIAL);
// Boju dodeljujemo pomocu glColor()
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
glColor3f(0.75f, 0.75f, 0.75f);
glBegin(GL_TRIANGLES);
glVertex3f(-15.0f,0.0f,30.0f);
glVertex3f(0.0f, 15.0f, 30.0f);
glVertex3f(0.0f, 0.0f, -56.0f);
glEnd();
```

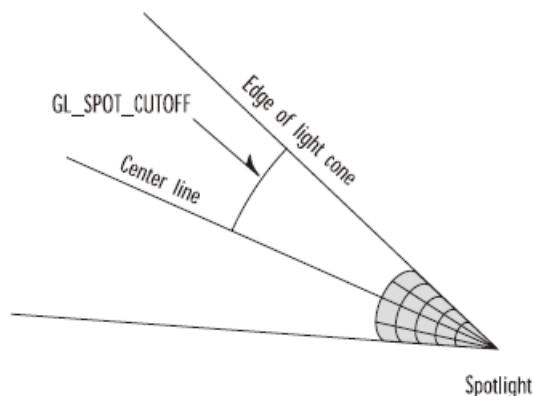
8.10 Upotreba izvora svetla i materijala

Sam OpenGL garantuje definisanje do 8 izvora svetla - mada vam u većini slučajeva nikada neće ni trebati više.

Pre nego što ispišemo kod za naš konačni primer upotrebe svetla i materijala neophodno je napomenuti i svrhu specular parametra i pojam spot svetla. Specular parametar se koristi za definisanje stepena svetlucavosti materijala i njenu boju - primer metala. Sam specular se zapravo definiše iz dve komponente: specular - boja svetlucavosti i shininess - intenzitet svetlucavosti i prihvata vrednosti od 1 do 128.

8.11 Spot svetla

Spot svetla su precizno usmereni izvori svetla. Definisani su lokacijom odakle se emituju, destinacijom - fokusom svetla i uglom njihove kupe (vrednost prosledjena kao parametar je polovina ugla kupe) - baš kao i prava spot svetla. Naravno postoje i mnogi drugi faktori koji utiču na spot svetla, međutim nećemo ih pominjati u ovom radu.



8.12 Kreiranje Spot svetla

Sledeće parče koda demonstrira praktičnu upotrebu spot svetla i senčenje - procedura je ista i za svako drugo:

```
// Definisemo vrednosti i koordinate
GLfloat lightPos[] = { 0.0f, 0.0f, 75.0f, 1.0f }; // pozicija odakle se emituje svetlost
GLfloat specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat specref[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat ambientLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat spotDir[] = { 0.0f, 0.0f, -1.0f }; // destinacija svetla
// zatim koristimo OpenGL komande
glEnable(GL_DEPTH_TEST); // Uklanjam sakrivene stranice
glFrontFace(GL_CCW); // Counterclockwise polygons face out
glEnable(GL_CULL_FACE); // Ne prikazujemo zadnju stranu poligona
// Enable lighting
glEnable(GL_LIGHTING);
// postavljamo i podesavamo light 0
// definisemo njegovu boju
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLight);
// Svetlo sastavljamo od diffuse i specular komponenti
glLightfv(GL_LIGHT0, GL_DIFFUSE, ambientLight);
glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
glLightfv(GL_LIGHT0, GL_POSITION, lightPos); // pozicija u prostoru
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spotDir); // destinacija
// Specifčni parametri za spot - usmereno svetlo
// Ugao od 60 stepeni za kupu svetla - ukupno 120
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 60.0f);
// Uključujemo svetlo
glEnable(GL_LIGHT0);
// Uključujemo color tracking
glEnable(GL_COLOR_MATERIAL);
// Postavljamo svojstva materijala
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
// Svetlu dodeljujemo visok stepen svetlucavosti
glMaterialfv(GL_FRONT, GL_SPECULAR, specref);
glMateriali(GL_FRONT, GL_SHININESS, 128);
// Postavljamo boju pozadine
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
// Zatim crtamo objekat
// + + + + + MESTO ZA KOD CRTANJA + + + + +
```

9. Zaključak

U ovom radu obradili smo osnove OpenGL-a. Ovo predstavlja samo deo njegovih mogućnosti i najelementarnije stvari neophodne za crtanje 3D objekata. Sam OpenGL se koristi u mnogim naćnim i industrijskim sferama - primer medicine ili rad aplikacija za vizualizaciju u meteorologiji. Iza ovih osnova krije se još mnogo toga. Na primer kako OpenGL prikazuje glatke površine, rad sa normalama pri osvetljenju, upotreba alfa kanala, fragment i vertex shaderi ...

Za razumevanje ovog materijala poželjno je poznavanje osnova 3D grafike i neke od aplikacija kao što je 3DSMAX ili Maya pošto je upućivanje ćitaoca u same osnove 3D grafike i osnovne pojmove koje podrazumevamo praktićno nemoguće smestiti na samo pedesetak strana.

Danas se OpenGL koristi i u segmentima računarstva gde na prvi pogled ne bismo predpostavili - savremenim korisničim interfejsim, prikazivanje prozora itd. Mnoge aplikacije ne bi bilo moguće napisati bez OpenGL-a. Neke od njig su i Adobe Photoshop, Adobe Premiere, Adobe After Effects

10. Literatura

1. [Begining OpenGL Game Programming] Dave Astel, Kevin Hawkins, THOMSON
2. [OpenGL Super Bible 4TH Edition]
Richard S. Wright Jr, Benjamin Lipchak, Nicholas Haemel, ADDISON WESLEY
3. [<http://www.opengl.org/sdk/docs/>] oficijalna OpenGL dokumentacija