



Uvod u OpenGL

Šta je OpenGL?

OpenGL je softverski interfejs (Application Programming Interface - API) prema grafičkom hardveru. Ovaj interfejs čini nekoliko stotina procedura i funkcija, čiji je osnovni zadatak da omogući definisanje objekata i operacija za kreiranje visokokvalitetnih grafičkih slika i interaktivnih 3D aplikacija.

OpenGL je nastao početkom 90-tih godina prošlog veka, i zvanično se uzima 1. jul 1992. godine kao početak njegovog razvoja (prva specifikacija). Prva implementacija pojavila se tek 1993. godine i to pod imenom Iris GL, i bio je namenjen *Silicon Graphics* radnim stanicama. U narednim godinama *Silicon Graphics* nastavlja da razvija specifikaciju u saradnji sa drugim proizvođačima grafičkog hardvera, ali ne više kao svoju specifikaciju, već kao otvoreni standard, koji će se uskoro nazvati OpenGL (*Open Graphics Library*).

Godine 1996. specificirana je verzija 1.1 i to je ujedno i prva i poslednja verzija koju je *Microsoft* direktno uključio u Win32 API i podržao u svojim softverskim alatima za Windows operativne sisteme. Zapanjujuća je činjenica da čak i nakon instalacije *Microsoft Visual Studio 2005* razvojnog paketa, datoteke zaglavljene datiraju iz 1996. godine¹ i sadrže kopirajt *Silicon Graphics Inc.* *Microsoft Visual Studio 2008* potpuno je ukinuo podršku za OpenGL. To nikako ne znači da je OpenGL stagnirao u proteklih 12 godina. Naprotiv, doživeo je čak 10 revizija. Trenutno aktuelna verzija je 3.2. Pregled verzija i datuma objavljivanja dat je u tabeli 1.1.

Obzirom da je OpenGL otvoreni standard, činjenica da nije direktno podržan bibliotekama koje dolaze uz *Microsoft*-ove alate za programiranje ne predstavlja nikakvo ograničenje u primeni najnovijih funkcija implementiranih u grafičkom hardveru. Počev od verzije 1.2, OpenGL uvodi ekstenzije kao vrlo efikasan mehanizam za pristup najnovijim funkcijama koje još nisu postale deo standarda. Naime, sva funkcionalnost OpenGL-a implementirana je u drajverima, a ekstenzije nude interfejs ka tim funkcijama. Na adresi

¹ Pogledati datoteku GL.h koja se nalazi u direktorijumu ...\\Program Files\\Microsoft Visual Studio 8\\VC\\PlatformSDK\\Include\\gl\\

<http://www.opengl.org/registry/> mogu se naći specifikacije svih poznatih ekstenzija. U trenutku pisanja ovog praktikuma (septembar 2009. godine), bilo ih je 456. Potrebno je preuzeti sa odgovarajućeg web sajta potrebne *header* datoteke, da bi se znalo kako izgledaju zaglavlja novih funkcija, a samoj funkciji se pristupa preko pointera koji vraća drajver grafičkog adaptera. Ovo znači da odmah nakon nabavke novog hardvera možemo koristiti prednosti koje on donosi. Na Internetu postoje mnoge, potpuno besplatne, biblioteke, koje dodatno olakšavaju primenu ekstenzija. Osim toga, OpenGL podržavaju svi značajniji operativni sistemi: *Microsoft Windows*, *Apple MacOS*, sve verzije *Unix-a* (*SUN-Solaris*, *IBM -multiprocessor AIX*, *HP – UX*, *SGI IRIX*, *Compaq Tru64 Unix*, *FreeBSD*, ...), *Linux*, itd. Postoji i API za prenosne uređaje i ugrađene sisteme – OpenGL ES (*OpenGL for Embedded Systems*). Nokia 6630 smartphone, zasnovana na *S60 2nd Edition Feature Pack 2*, krajem 2004. godine bio je jedan od prvih mobilnih telefona sa 3D API-jem.

Tabela 1.1. Pregled verzija OpenGL-a

Verzija	Datum objavljivanja	Važna novina
OpenGL 1.0	1. jul 1992.	Iris GL
OpenGL 1.1	1996/97.	<i>Vertex Array</i> , <i>Texture Objects</i> , ...
OpenGL 1.2	16. mart 1998.	<i>3D Texturing</i> , <i>BGRA Pixel Formats</i> , ...
OpenGL 1.2.1	14. oktobar 1998.	Uvode se ARB ekstenzije i <i>multitexture</i>
OpenGL 1.3	14. avgust 2001.	<i>Compressed Textures</i> , <i>Cube Map Textures</i> , <i>Multisample</i> , <i>Multitexture</i> ...
OpenGL 1.4	24. jul 2002.	<i>Depth Textures and Shadows</i> , <i>Fog Coordinate</i> , <i>Multiple Draw Arrays</i> , ... usvojena nova ekstenzija <i>ARB vertex program</i> (najava novog doba)
OpenGL 1.5	29. jul 2003.	<i>Buffer Objects</i> , <i>Shadow Functions</i> , ..., i nove ekstenzije <i>ARB shader objects</i> , <i>ARB vertex shader</i> i <i>ARB fragment shader</i>
OpenGL 2.0	7. septembar 2004.	<i>OpenGL Shading Language</i> (verzije 1.10 i novije), teksture ne moraju biti dimenzija 2^N , <i>Point Sprites</i> ,...
OpenGL 2.1	30. jul 2006.	<i>OpenGL Shading Language</i> verzija 1.20, nekvadratne matrice, <i>Pixel Buffer Object</i> ...
OpenGL 3.0	11. avgust 2008.	<i>OpenGL Shading Language</i> verzija 1.30, uslovni rendering, novi <i>rendering context</i> ...
OpenGL 3.1	24. mart 2009.	<i>OpenGL Shading Language</i> verzija 1.40, raskidanje kompatibilnosti sa prethodnim verzijama...
OpenGL 3.2	3. avgust 2009.	<i>OpenGL Shading Language</i> verzija 1.50, geometry shader, blokovi parametara, profili...

Inicijalno, OpenGL funkcionalnost bila je dostupna jedino korišćenjem C/C++ programskog jezika. Od sredine 2003. godine SUN i SGI počinju saradnju u razvoju podrške za Javu, čime je i ovaj popularni programski jezik otvorio put 3D grafici. Danas je dostupno mnoštvo Java biblioteka sa podrškom za OpenGL (*Lightweight Java Game Library* - *LWJGL*, *Java 3D*, *Yet Another Java OpenGL Binding* – *YAJOGLB*, *Jogl*, *JavaOpenGL 1.0a3*, itd.). OpenGL je dostupan i za Fortran90 (u verziji 1.2), Perl, Pike, Python, Adu i C#.

Od verzije 7.0 Microsoft DirectX API postaje ozbiljan konkurent OpenGL-u, ali on je dostupan samo na Windows platformi. Štaviše, DirectX 10 dostupan je samo na Vista.

Šta je potrebno da bi se koristio OpenGL pod Windowsom?

Da bismo mogli početi sa programiranjem u OpenGL-u pod Windows-om, korišćenjem C++-a, najpre je potrebno proveriti da li su prisutne sve potrebne komponente, i to:

- skup biblioteka (*opengl32.lib*, *glu32.lib*, a poželjno je i *glaux.lib* i *glut.lib*),
- zaglavla (gl.h, glu.h, glaux.h, glut.h),
- *run-time* biblioteka (*OpenGL32.dll*) – mora biti iste verzije kao i biblioteke, ili novija, i
- klijentski drajver (*Installable Client Driver - IDC*) – dolazi uz drajver grafičke kartice i omogućuje hardversku akceleraciju i dodatne funkcije.

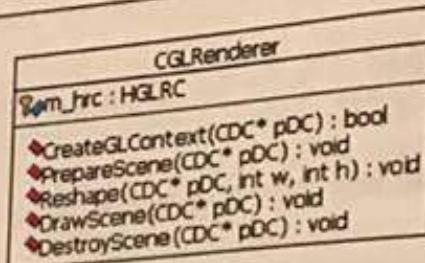
Naravno, podrazumeva se i postojanje odgovarajućeg okruženja za razvoj programa. Za sve primere koji slede korišćen je Microsoft Visual Studio, tačnije Visual C++ komponenta ovog okruženja. Svi prikazani primeri mogu se prevesti i izvršiti u svim verzijama Microsoft Visual Studio okruženja, počev od verzije 6.0 (VS 6.0), pa do verzije 2008 (VS 2008).

Priprema Visual C++ projekta za rad sa OpenGL-om

Obzirom na obilje korisnih klasa implementiranih u okviru Microsoft Foundation Class (MFC) biblioteke, kao i na činjenicu da su se studenti već susretali sa primenom MFC-a, povezivanje C++ projekta i OpenGL-a biće prikazano na primeru MFC aplikacije.

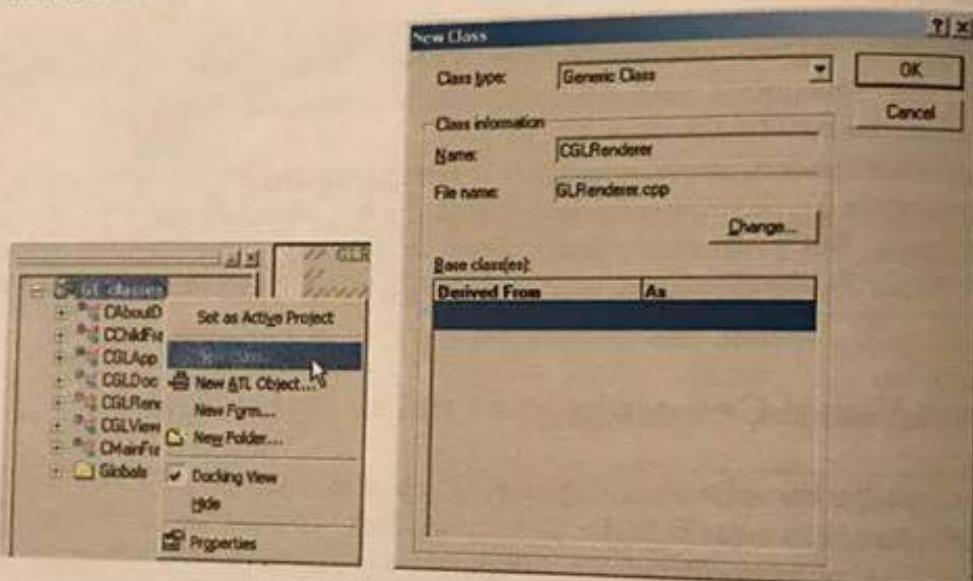
Sve funkcije vezane za OpenGL grupaćemo u jednu klasu, nazvanu **CGLRender**. Na slici 1.1 prikazana je ova klasa sa najbitnijim komponentama:

- atributom
 - HGLRC *m_hrc* – OpenGL Rendering Context (OpenGL server koji vrši iscrtavanje)
- metodama
 - *bool CreateGLContext(CDC* pDC)* – kreira OpenGL Rendering Context,
 - *void PrepareScene(CDC* pDC)* – inicijalizuje scenu,
 - *void DestroyScene(CDC* pDC)* – oslobada resurse alocirane u drugim funkcijama ove klase,
 - *void Reshape(CDC* pDC, int w, int h)* – kod koji treba da se izvrši svaki put kada se promeni veličina prozora ili pogleda i
 - *void DrawScene(CDC* pDC)* – iscrtava scenu



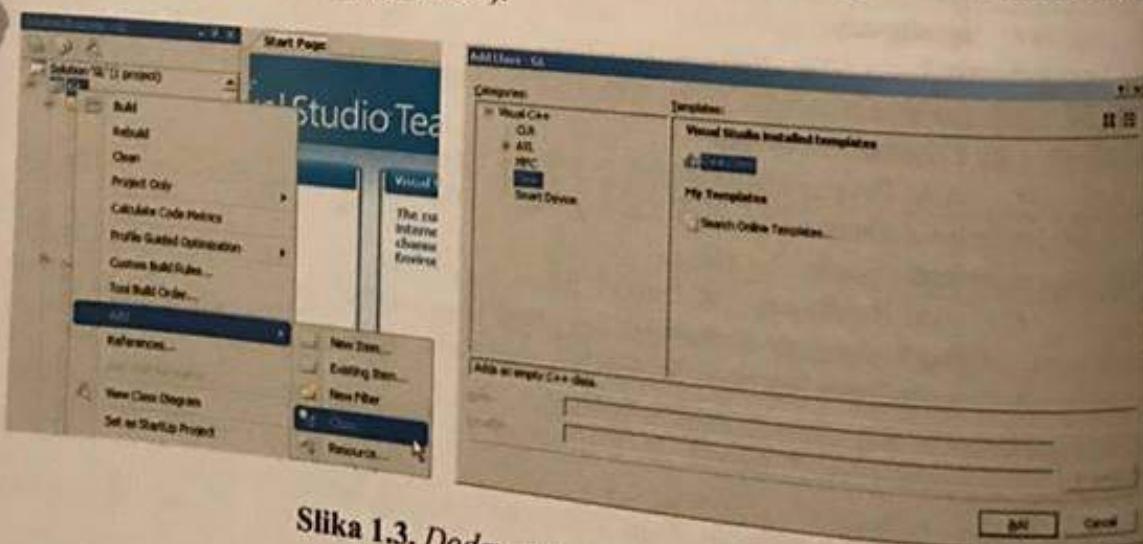
Slika 1.1. Klasa CGLRenderer

Dodavanje nove klase ostvaruje se izborom opcije **Insert/New Class...**, iz glavnog menija VS 6.0, ili desnim klikom na naziv projekta u prozoru *ClassView* i izborom opcije **New Class...** (slika 1.2). Za tip klase (*Class type*) izabraćemo **Generic Class**, a ime (*Name*) postaviti na **CGLRenderer**.



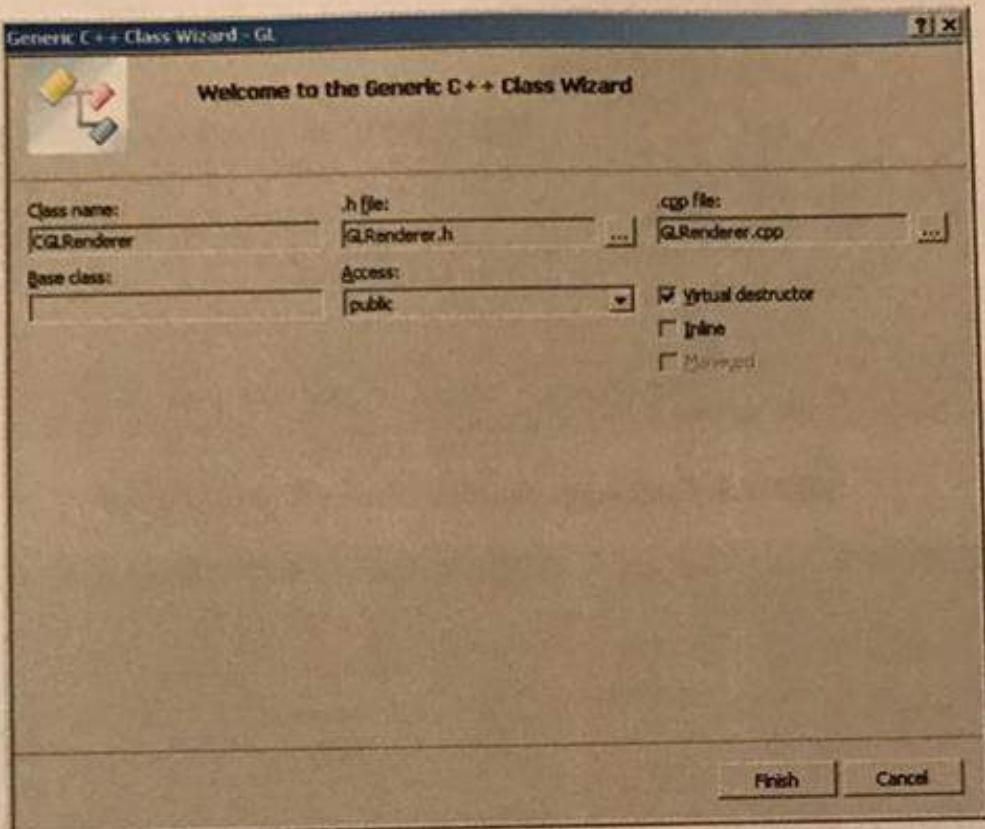
Slika 1.2. Dodavanje nove klase u VS 6.0

U VS2008 dodavanje nove klase u tekući projekat ostvaruje se izborom opcije **Project/Add Class...**, iz glavnog menija, ili desnim klikom na naziv projekta u prozoru *Solution Explorer* i izborom opcije **Add/Class...** (slika 1.3).



Slika 1.3. Dodavanje nove klase u VS 2008

Nakon klika na dugme Add u okviru *Add Class* dijaloga, otvara se novi dijalog za postavljanje parametara klase (slika 1.4).

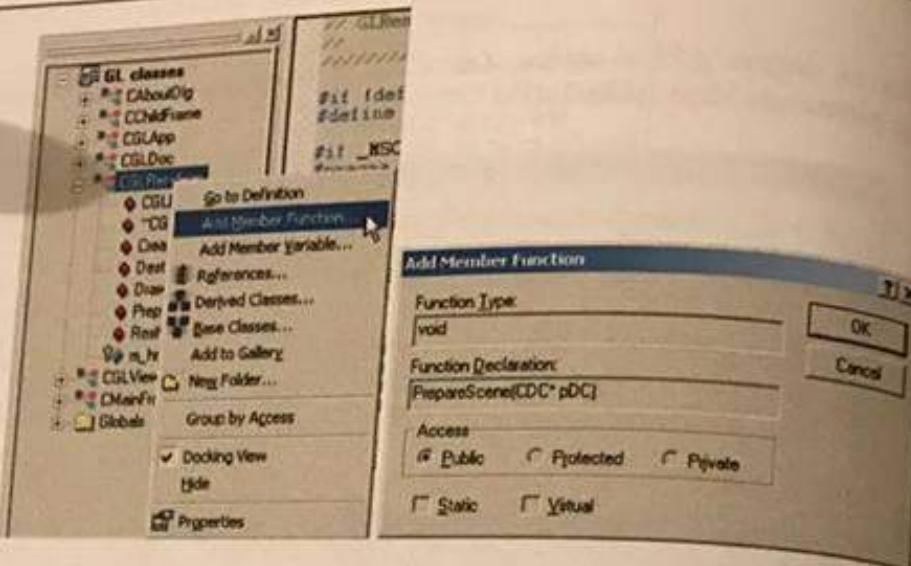


Slika 1.4. Dodavanje nove klase u VS 2008 - nastavak

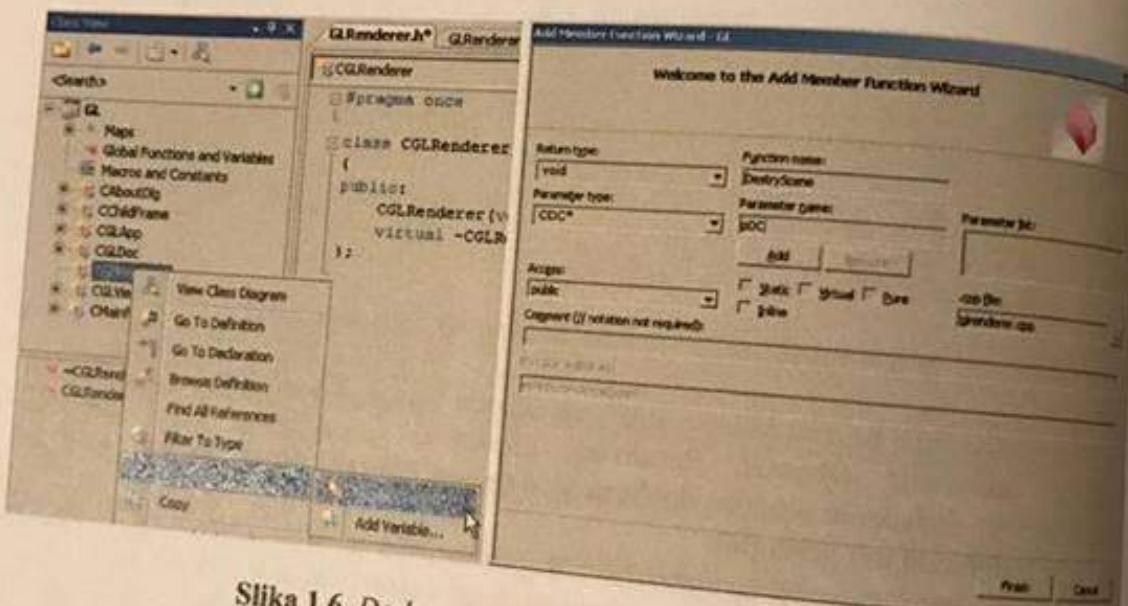
Nakon popunjavanja naziva klase i klika na dugme *Finish*, klasa je formirana. Sledeći korak je dodavanje *OpenGL Rendering Context*-a. U zaglavlju klase (datoteka *GLRenderer.h*) dodaćemo sledeću deklaraciju, i to kao zaštićeni atribut (jer van ove klase niko ne bi trebalo da mu pristupa):

```
protected:  
    HGLRC     m_hrc;    //OpenGL Rendering Context
```

Da bi klasa **CGLRenderer** bila funkcionalna, moramo dodati i prethodno pobrojane metode (funkcije članice). Dodavanje funkcije članice, u VS 6.0 ostvaruje se desnim klikom na naziv odgovarajuće klase u okviru prozora *ClassView*. Tom prilikom otvara se padajući meni (slika 1.5) iz koga treba izabrati opciju *Add Member Function...*. Time se otvara dijalog za unos tipa funkcije, njene deklaracije, pristupnog opsega itd. Na slici 1.5 prikazano je dodavanje funkcije *PrepareScene()* u VS 6.0, a na slici 1.6 u VS 2008.



Slika 1.5. Dodavanje funkcije članice klase u VS 6.0



Slika 1.6. Dodavanje funkcije članice klase u VS 2008

U dijalogu za dodavanje funkcije članice u VS 2008, potrebno je uneti tip povratne vrednosti funkcije (*Return type*), naziv funkcije (*Function name*), tip parametra (*Parameter type*) i naziv parametra (*Parameter name*). Klikom na dugme *Add*, parametar se dodaje u listu parametara i može se nastaviti sa dodavanjem novih. Nakon izbora pristup (*Access*), klika na dugme *Finish*, funkcija članica je dodata.

Prvu funkciju koju ćemo definisati biće `CreateGLContext()`. Da bi koristili OpenGL, najpre moramo da kreiramo OpenGL Rendering Context, a to upravo radi ova funkcija.

```
bool CGLRenderer::CreateGLContext(CDC* pDC)
{
    PIXELFORMATDESCRIPTOR pfd;
    memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));
    pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
    pfd.nVersion = 1;
```

```

    pfd.dwFlags      = PFD_DOUBLEBUFFER | PFD_SUPPORT_OPENGL
                      | PFD_DRAW_TO_WINDOW;
    pfd.iPixelFormat = PFD_TYPE_RGBA;
    pfd.cColorBits   = 32;
    pfd.cDepthBits   = 32;
    pfd.iLayerType   = PFD_MAIN_PLANE;

    int nPixelFormat = ChoosePixelFormat(pDC->m_hDC, &pfd);

    if (nPixelFormat == 0) return false;

    BOOL bResult = SetPixelFormat (pDC->m_hDC, nPixelFormat,
                                  &pfd);

    if (!bResult) return false;

    m_hrc = wglCreateContext(pDC->m_hDC);

    if (!m_hrc) return false;

    return true;
}

```

OpenGL Rendering Context (u daljem tekstu RC) se kreira tako što se najpre izabere željeni format piksela, popunjavanjem **PIXELFORMATDESCRIPTOR** strukture. Ova struktura se prosledjuje funkciji **ChoosePixelFormat()**, koja proverava da li prosledeni DC (*Device Context*) podržava taj format. Ukoliko podržava vraća celobrojni indeks tog formata, a ukoliko ne podržava, indeks najpričišnjeg formata piksela koji DC podržava. Ukoliko funkcija **ChoosePixelFormat()** vrati 0, postoji problem sa izborom formata piksela, pa treba prekinuti postupak formiranja RC-a.

Vrednost koju vrati **ChoosePixelFormat()** treba proslediti funkciji **SetPixelFormat()**, kako bi željeni format piksela bio izabran u datom DC-u. Tek sada se može kreirati RC korišćenjem funkcije **wglCreateContext()**.

Sve ono što treba da se izvrši samo jednom, pre prvog iscrtavanja scene, treba pozvati iz funkcije **PrepareScene()**. Da bi bilo koja OpenGL funkcija mogla da se izvrši, neophodno je selektovati i aktivirati RC koji će izvršiti iscrtavanje. Zbog toga se na početku mora pozvati funkcija **wglMakeCurrent()**, koja aktivira RC (drugi parametar u pozivu ove funkcije) i povezuje ga sa DC-om (prvi parametar funkcije). Sada možemo videti i razlog zašto skoro sve metode klase **CGLRender**er kao parametar imaju pokazivač na DC.

```

void CGLRender::PrepareScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClearColor (1.0, 1.0, 1.0, 0.0);
    //-----
    wglMakeCurrent(NULL, NULL);
}

```

Nakon završetka rukovanja OpenGL-om, poželjno je vratiti upravljanje iscrtavanjem prozora Windows-u, pa se deseletuje RC pozivom funkcije **wglMakeCurrent()**, pri čemu su oba parametra NULL. U kasnijim primerima, funkcija **PrepareScene()** sadržće mnogo više koda, ali za potrebe razvoja prve aplikacije sa podrškom za OpenGL, postavićemo

samo boju za brisanje prozora na belu, pozivom funkcije `glClearColor()` sa odgovarajućim parametrima. Detaljan opis OpenGL funkcija sledi u narednim poglavljima.

Iscrtavanje scene vrši se u funkciji `DrawScene()`. Sve što je potrebno da se izvrši prilikom svakog iscrtavanja prozora nalazi se u ovoj funkciji. Da se ne bi video proces formiranja slike (iscrtavanje jedne po jedne primitive), prilikom formiranja formata piksela, izabrano je dvostruko baferovanje (*double buffer mod*). Ovaj režim podrazumeva iscrtavanje slike u pozadinskom baferu, koji nije vidljiv. Tek kada slika bude gotova, i u trenutku kada se osvežava prikaz na monitoru, vrši se zamena pozadinskog (*back*) i prednjeg, tj. bafera koji se prikazuje na ekranu (*front*). Zamena prednjeg i zadnjeg bafera ostvaruje se funkcijom `SwapBuffers()`. Funkciju `SwapBuffers()` treba pozvati na samom kraju funkcije `DrawScene()`, a pre deseletovanja RC-a.

```
void CGLRenderer::DrawScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();

    glBegin(GL_TRIANGLES);
    glColor3f(1.0, 0.0, 0.0);
    glVertex3f(-1.0, -1.0, -5.0);
    glColor3f(0.0, 1.0, 0.0);
    glVertex3f(1.0, -1.0, -5.0);
    glColor3f(0.0, 0.0, 1.0);
    glVertex3f(1.0, 1.0, -5.0);
    glEnd();

    glFlush();
    //-----
    SwapBuffers(pDC->m_hDC);
    wglMakeCurrent(NULL, NULL);
}
```

Sve što je potrebno da se izvrši kada se menja veličina prozora, a tiče se iscrtavanja scene, smešteno je u funkciju `Reshape()`. Ukoliko se pravi *full screen* aplikacija, ili se veličina prozora, u kome OpenGL vrši iscrtavanje, ne menja, potreba za funkcijom `Reshape()` ne postoji. U tom slučaju kod koji je prikazan u ovoj funkciji može biti premešten na kraj funkcije `PrepareScene()`. Takođe, ovaj kod se može premestiti i na početak funkcije `DrawScene()`. Takođe, ovaj kod se poziva bilo koje druge OpenGL funkcije. Ovde je prikazana efikasnija verzija, jer se promena perspektive, koja je definisana u kodu, javlja samo u slučaju promene veličine i oblika prozora. Ova funkcija biće pozvana i pre prvog iscrtavanja prozora, tako da su validne vrednosti postavljenje i ukoliko prozor naknadno ne menja veličinu.

```
void CGLRenderer::Reshape(CDC *pDC, int w, int h)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(40, (double)w/(double)h, 1, 100);
    glMatrixMode(GL_MODELVIEW);
```

```
wglGetCurrent(NULL, NULL);
```

Na kraju, sve što je zauzeto (alocirano) treba i osloboditi, tj. dealocirati. To je zadatak funkcije **DestroyScene()**. Sav kod za dealokaciju treba da se nađe između dva poziva funkcije **wglGetCurrent()**. Zatim se „uništava“ i sam *OpenGL Rendering Context*, pozivom funkcije **wglDeleteContext()**.

```
void CGLRenderer::DestroyScene(CDC *pDC)
{
    wglGetCurrent(pDC->m_hDC, m_hrc);
    // ...
    wglGetCurrent(NULL, NULL);
    if(m_hrc)
    {
        wglDeleteContext(m_hrc);
        m_hrc = NULL;
    }
}
```

Sve pobrojane funkcije, kao i sama klasa za upravljanje iscrtavanjem preko OpenGL-a, mogu biti drugačije nazvane, i sasvim drugačije implementirane. Ovde je samo iznet skup preporuka.

Ako sada pokušate da prevedete kod, javiće se gomila grešaka, jer C++ prevodilac ne prepoznaje mnoge funkcije. Pažljivim pregledavanjem poruka o greškama može se uočiti da su sve vezane za OpenGL. Da bi se ovaj problem rešio potrebno je:

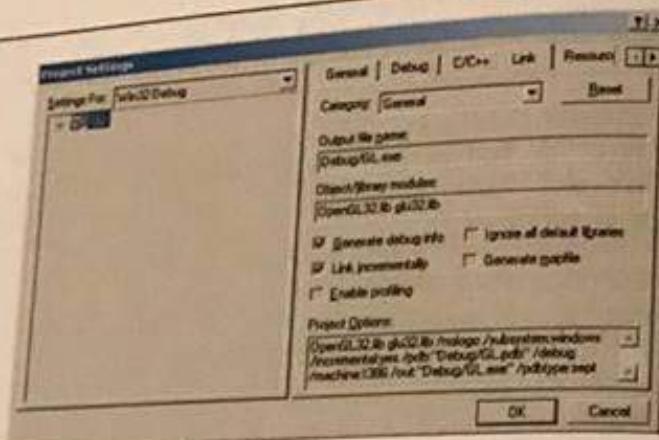
- ukazati na datoteke koje sadrže zaglavljaju tih funkcija i
- uključiti biblioteke koje sadrže samu implementaciju funkcija.

Deklaracije svih OpenGL funkcija nalaze se u dve datoteke: **gl.h** i **glu.h**. U prvoj se nalaze zaglavljaju funkcija iz samog jezgra OpenGL-a, dok su u drugoj zaglavljaju pomoćnih funkcija (*Utility Library*). Pomoćne funkcije samo kombinuju pozive funkcija iz jezgra, ali olakšavaju mnoge programerske zadatke, te se vrlo često koriste. Prvi deo rešenja problema je, dakle, dodavanje sledeća dva reda na početak **GLRenderer.cpp** datoteke:

```
#include <GL\gl.h>
#include <GL\glu.h>
```

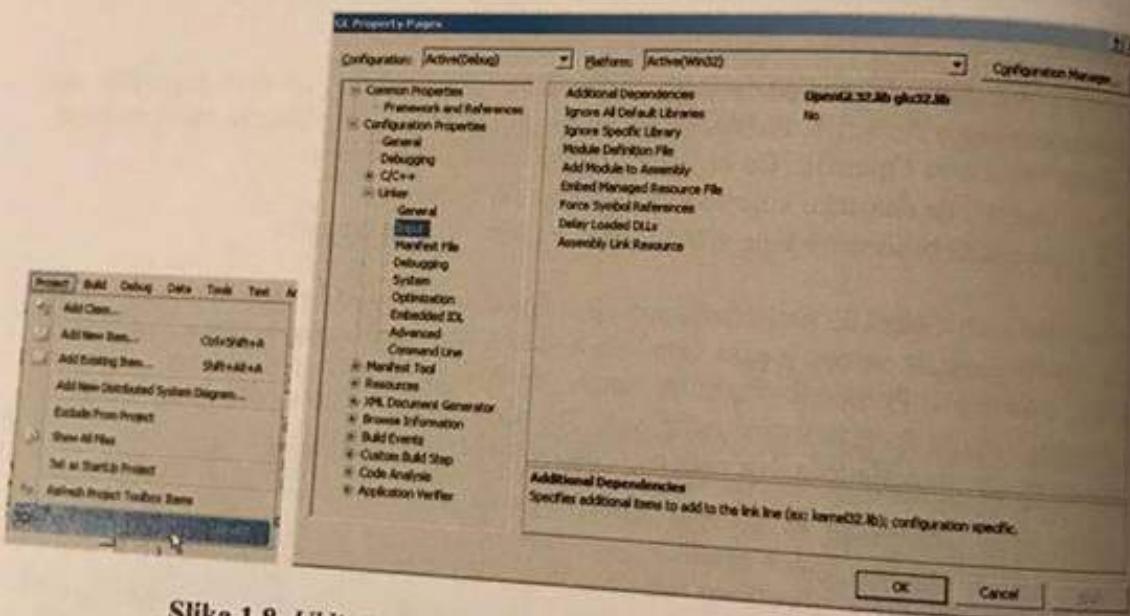
Obe datoteke zaglavljaju nalaze se u poddirektorijumu **GL** u okviru standardnog **Include** direktorijuma (\ Program Files \ Microsoft Visual Studio \ VC98 \ Include), kod VS 6.0. Slična je putanja i kod VS 2003 i VS 2005, dok kod VS 2008, ove datoteke **NE POSTOJE!** Ukoliko radite u VS 2008, kompletna podrška za OpenGL mora se naknadno dodati, najjednostavnije kopiranjem iz neke prethodne verzije VS-a.

Sledeći korak jeste dodavanje biblioteka. U VS 6.0 Uključivanje biblioteke ostvaruje se izborom opcije *Project/Settings...* iz glavnog menija, a zatim na kartici *Link* u polju *Object/Library modules* uneti ime biblioteke (u ovom slučaju **OpenGL32.lib** i **glu32.lib**).



Slika 1.7. Uključivanje staticke biblioteke u VS 6.0 projekat

U Visual Studio 2008 okruženju, staticka biblioteka se uključuje izborom opcije *Project/<ImeProjekta> Properties...*, zatim *Configuration Properties / Linker / Input*, i na kraju unosom imena biblioteke u polje *Additional Dependeces*.



Slika 1.8. Uključivanje staticke biblioteke u VS 2008 projekat

Moguće je i programski uključiti staticku biblioteku, i to pomoću direktive **#pragma**. Na primer, za dodavanje podrške za OpenGL u datoteci **StdAfx.h** treba dodati sledeće linije koda:

```
#pragma comment (lib, "OpenGL32.lib")
#pragma comment (lib, "glu32.lib")
```

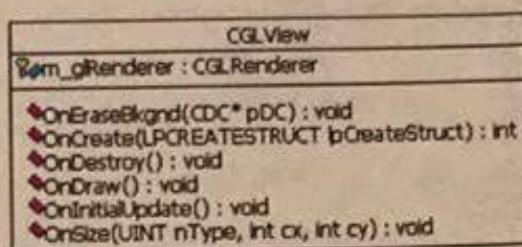
Ukoliko postoje različite verzije biblioteka (*release* i *debug*) uključivanje se obavlja na sledeći način:

```
#pragma message ("Automatsko povezivanje biblioteke")
#ifndef _DEBUG
```

```
#pragma comment( lib, "lib_release.lib")
#else
#pragma comment( lib, "lib_debug.lib")
#endif
```

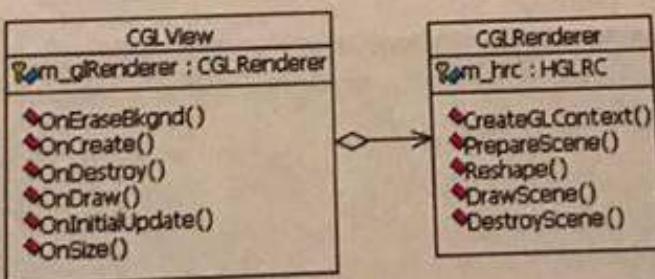
Sada se projekat može prevesti, ali ne daje nikakve rezultate. Tačnije, ne vrši se iscrtavanje definisano u okviru **CGLRenderer-a**. Razlog je vrlo jasan – klasa **CGLRenderer** nigde nije inicijalizovana, niti se njene funkcije zovu na izvršenje.

Sledeći korak je prilagodavanje klase pogleda (**CGLView**) za korišćenje **CGLRenderer-a**. Na slici 1.9 je prikazana klasa **CGLView** samo sa metodama koje su bitne za povezivanje sa **CGLRenderer-om**.



Slika 1.9. Klasa CGLView

Kao što se sa slike 1.9 može videti, formiraćemo instancu klase **CGLRender**er u okviru klase prozora. Tačnije, u duhu Objektnog projektovanja, formirali smo jednu agregaciju. Agregaciona veza prikazana je na slici 1.10.



Slika 1.10. Relacija klasa CGLView i CGLRender

Da bi se to ostvarilo, u zaglavlju klase **CGLView** dodati sledeći kod:

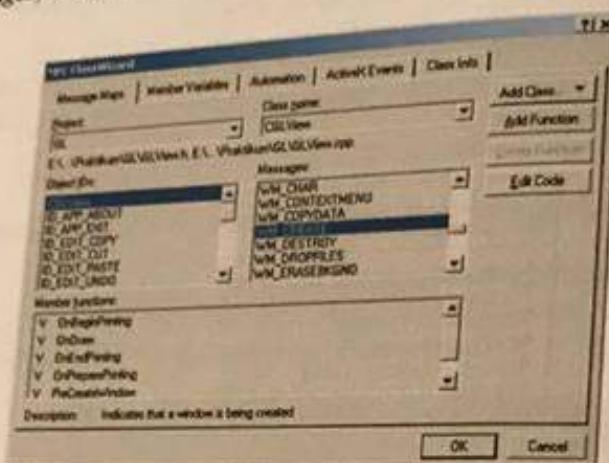
```
protected:
    CGLRender m_glRender;
```

a ispred deklaracije klase **CGLView**:

```
#include "GLRender.h"
```

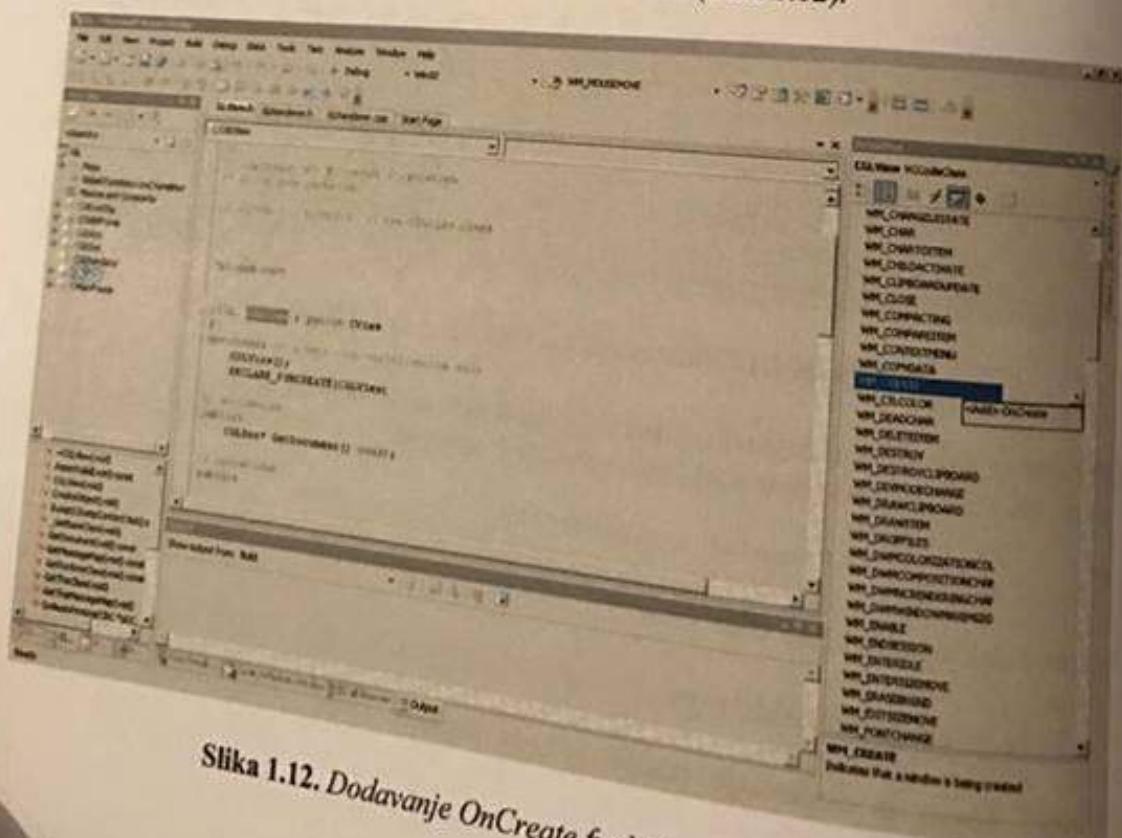
Sledeći korak je dodavanje poziva metoda instance klase **CGLRender**er iz metoda **CGLView**.

Kreiranje OpenGL RC-a treba izvršiti pre bilo kakvog iscrtavanja. Idealno mesto za to je funkcija `CGLView::OnCreate()`. Funkcija `OnCreate()` poziva se nakon što je prozor kreiran, ali pre nego što postane vidljiv. Inicijalno nije predefinisana, pa je u VS 6.0 potrebno dodati pozivanjem MFC *ClassWizard*-a. To se ostvaruje izborom opcije `View/ClassWizard` iz glavnog menija ili prečicom *Ctrl+W* sa tastature. Iz spiska ponudenih poruka (*Messages*) izabrati `WM_CREATE`, a zatim kliknuti na taster *Add Function* (slika 1.11).



Slika 1.11. Prozor MFC *ClassWizard*-a u VS 6.0

Ukoliko se koristi VS 2008, potrebno je u *ClassView*-u odabrat odgovarajuću klasu pogleda (u primeru *CGLView*), a zatim u prozoru *Properties* kliknuti na ikonu *Messages* i u spisku poruka izabrati `WM_CREATE` i <Add> `OnCreate` (slika 1.12).



Slika 1.12. Dodavanje *OnCreate* funkcije u VS 2008

Nakon što je *wizard* kreirao funkciju, u njeni telo dodajemo poziv funkcije **CGLRenderer::CreateGLContext()**. Obzirom da **CreateGLContext()** kao parametar zahteva DC za koji kreiramo RC, potrebno je pre poziva ove funkcije pribaviti pokazivač na DC. Postoji više načina za to, a jedan od njih je poziv funkcije **GetDC()**. **GetDC()** je funkcija članica prozora i vraća pokazivač na DC datog prozora. Po završetku korišćenja DC-ja, potrebno ga je oslobođiti. Za to se koristi poziv funkcije **ReleaseDC()**. Sve pozive funkcija članica klase CGLRenderer uokvirićemo parom poziva **GetDC/ReleaseDC**.

Drugi način pribavljanja DC-ja je kreiranje instance klase **CClientDC**, sa parametrom **this**. Konstruktor ove klase internu poziva **GetDC()**, a destruktur **ReleaseDC()**. Obzirom da se destruktur automatski poziva po izlasku iz funkcije gde je statički kreirana instanca klase **CClientDC**, dovoljno je dodati samo jednu liniju koda pre poziva funkcije u kojoj se koristi DC - **CClientDC dc(this);**

Sledi prikaz funkcije **OnCreate()** nakon dodavanja poziva funkcije za kreiranje RC-a.

```
int COGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    CDC* pDC = GetDC();
    m_glRenderer.CreateGLContext(pDC);
    ReleaseDC(pDC);

    return 0;
}
```

Funkcija **OnCreate()** je funkcija rukovalac (*handler function*) koja se aktivira na odgovarajuću Windows poruku – WM_CREATE. Na prethodno opisani način potrebno je dodati rukovaće i za sledeće poruke: WM_ERASEBKGND, WM_SIZE i WM_DESTROY.

Funkcija **OnEraseBkgnd()** aktivira se porukom WM_ERASEBKGND i rukuje brisanjem prozora pre svakog iscrtavanja. Naime, pre iscrtavanja prozora, vrlo često je potrebno obrisati njegov prethodni sadržaj, kako bi se uklonio neželjeni sadržaj. Boja kojom se „briše“ prozor podešava se u Windows-u, i podrazumevano je bela. Kako postoje slučajevi kada je to korisno, postoje slučajevi i kada to nije. Naša aplikacija je dobar primer za slučaj kada treba eliminisati brisanje. Ukoliko se to ne učini, pri svakom iscrtavanju javiće se neprijatan treptaj prozora, kada se za delić sekunde prozor oboji belom bojom, pre nego što se iscrti koristan sadržaj. Da bismo to sprečili dodaćemo rukovaoca za poruku WM_ERASEBKGND i zakomentarisati poziv funkcije **CView::OnEraseBkgnd(pDC)**, a umesto toga uvek vraćati TRUE.

```
BOOL CGLView::OnEraseBkgnd(CDC* pDC)
{
    return TRUE;
    //return CView::OnEraseBkgnd(pDC);
}
```

Funkcija **OnSize()** poziva se po prijemu poruke WM_SIZE, tj. kad god se promeni veličina prozora, ali i nakon inicijalnog kreiranja prozora. U ovoj funkciji smestićemo poziv odgovarajuće funkcije **CGLRenderer**-a koja treba da rukuje ovim dogadajem.

```
void CGLView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);

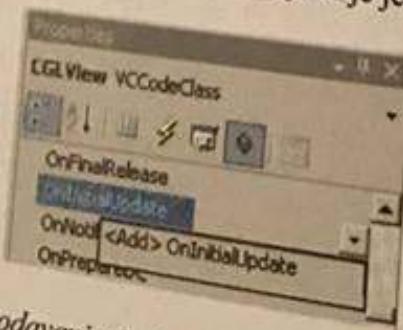
    CDC* pDC = GetDC();
    m_glRenderer.Reshape(pDC, cx, cy);
    ReleaseDC(pDC);
}
```

Funkcija **OnDestroy()** zove se neposredno pre „uništavanja“ prozora. Aktivira se po pristizanju poruke WM_DESTROY. U okviru ove funkcije trebalo bi „počistiti“ sve što je alocirano za potrebe prikaza, i uništiti *OpenGL Rendering Context*.

```
void CGLView::OnDestroy()
{
    CView::OnDestroy();

    CDC* pDC = GetDC();
    m_glRenderer.DestroyScene(pDC);
    ReleaseDC(pDC);
}
```

Inicijalizaciju scene obavićemo u funkciji **OnInitialUpdate()**. Ovo nije funkcija rukovalac dogadajem/porukom, već je članica klase pogleda (**CView**). Poziva se nakon što je pogled dodeljen dokumentu (pogledati *Document/View* arhitekturu u MSDN-u), a pre nego što je prvi put prikazan. Zbog toga je ovo idealno mesto za poziv inicijalizacije svega što je vezano za pogled. Obzirom da nije rukovalac dogadajem (*event handler*), ova funkcija se dodaje kao predefinisana funkcija (*override*) odgovarajuće nadklase (**CView**). U VS 6.0 dodavanje ovakvih funkcija ostvaruje se preko *ClassWizard*-a (slika 1.11). Potrebno je izabrati stavku **OnInitialUpdate** iz spiska *Messages*. U VS 2008 predefinisanje funkcija ostvaruje se izborom odgovarajuće klase (*ClassView*), a zatim u *Properties* prozoru kliknuti na ikonu *Overrides* kako bi se pojavio spisak funkcija koje je moguće predefinisati.



Slika 1.13. Dodavanje *OnInitialUpdate* funkcije u VS 2008

```
void CGLView::OnInitialUpdate()
{
    CView::OnInitialUpdate();

    CDC* pDC = GetDC();
    m_glRenderer.PrepareScene(pDC);
    ReleaseDC(pDC);

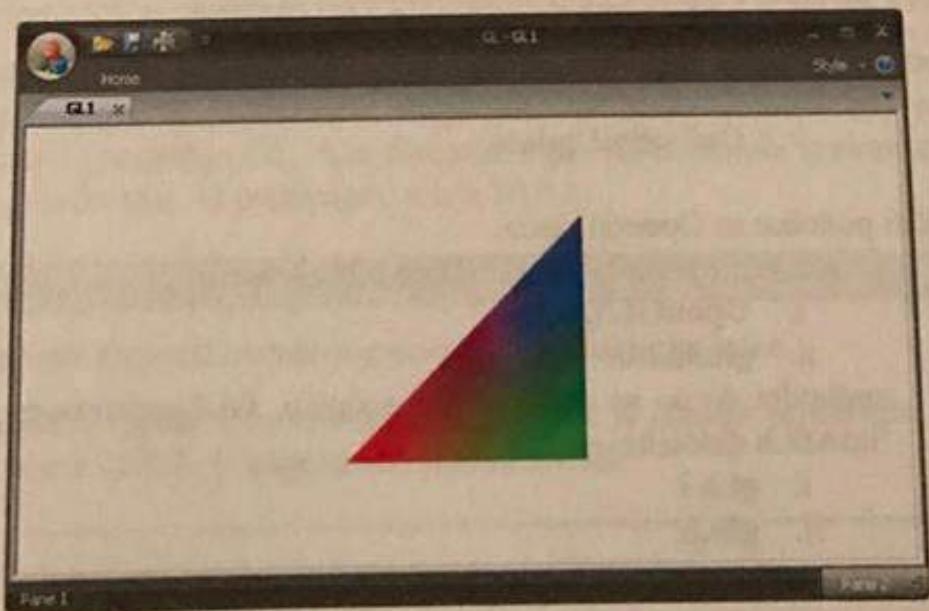
}
```

I, na kraju, treba dodati i poziv za iscrtavanje prozora – **CGLRenderer::DrawScene()**. Poziv funkcije **DrawScene()** umećemo u funkciju **CGLView::OnDraw()**. Ova funkcija već postoji u definiciji klase **CGLView**, a i već sadrži pokazivač na DC prozora, tako da je dovoljno dodati samo poziv odgovarajuće funkcije **CGLRenderer**-a. U VS 2008 parametar ove funkcije (**CDC* pDC**) je zakomentarisana. Da bi dobili pristup ulaznom parametru, potrebno je skinuti komentar (obrisati /* i */).

```
void CGLView::OnDraw(CDC* pDC)
{
    CGLDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    m_glRenderer.DrawScene(pDC);
}
```

Sada možemo prevesti aplikaciju. Ukoliko nema grešaka, trebalo bi dobiti pravougli trougao, sa svakim temenom obojenim drugom bojom, kao na slici 1.14.



Slika 1.14. Prva aplikacija u OpenGL-u

Kratak pregled gradiva

Da ukratko ponovimo korake za kreiranje podrške za OpenGL u VC++ MFC aplikaciji.

1. Kreirati MFC projekat u VS 6.0 ili VS 2003/2005/2008 okruženju.
2. Dodati klasu CGLRenderer koja objedinjuje sve funkcije za rad sa OpenGL-om, i okviru nje definisati
 - a. atribut
 - i. HGLRC **m_hrc**
 - b. funkcije
 - i. bool **CreateGLContext (CDC* pDC)**
 - ii. void **PrepareScene (CDC* pDC)**
 - iii. void **Reshape (CDC* pDC, int w, int h)**
 - iv. void **DrawScene (CDC* pDC)**
 - v. void **DestroyScene (CDC* pDC)**
3. Povezati klasu pogleda sa CGLRenderer-om
 - a. uključivanjem (agregacijom) CGLRenderer-a u klasu pogleda
 - b. dodavanjem odgovarajućih poziva CGLRenderer-a u funkcije pogleda, koje se dobijaju kao rukovaoci odgovarajućih poruka:
 - i. **WM_CREATE,**
 - ii. **WM_ERASEBKGND,**
 - iii. **WM_SIZE** i
 - iv. **WM_DESTROY,**
i predefinisanjem funkcije okvira
 - v. **OnInitialUpdate.**
4. Uključiti podršku za OpenGL kroz:
 - a. povezivanje projekta sa statičkim bibliotekama:
 - i. **OpenGL32.lib** i
 - ii. **glu32.lib.**
 - b. zaglavlja koja se dodaju na početku GLRenderer.cpp ili na kraju StdAfx.h datoteke
 - i. **gl.h** i
 - ii. **glu.h.**

Funkcije korišćene u ovoj glavi

```
int ChoosePixelFormat( HDC hdc,
                      CONST PIXELFORMATDESCRIPTOR * ppfd )
```

hdc – handle DC-a u kome se traži format piksela najpričližniji zadatom
ppfd – pokazivač na strukturu koja opisuje željeni format piksela

Funkcija vraća indeks formata piksela najpričližniji zadatom formatu. Ukoliko ne uspe, funkcija vraća 0.

```
BOOL SetPixelFormat( HDC hdc, int iPixelFormat,
                     CONST PIXELFORMATDESCRIPTOR * ppfd )
```

hdc – handle DC-a čiji se format piksela postavlja

iPixelFormat – indeks koji određuje format piksela koji se postavlja

ppfd – pokazivač na strukturu koja opisuje format piksela (uglavnom nema nikakvu ulogu pri ovom pozivu)

Funkcija vraća TRUE, ukoliko postavljanje formata piksela uspe. U protivnom, vraća FALSE.

```
HGLRC wglCreateContext( HDC hdc )
```

hdc – handle DC-a za koji se kreira odgovarajući OpenGL *rendering context*

Funkcija kreira OpenGL *rendering context* pogodan za iscrtavanje na uređaju čiji je DC prosleden kao parametar. OpenGL *rendering context* ima isti format piksela kao i prosledeni DC. Ako funkcija uspe, vraća handle kreiranog OpenGL *rendering context-a*. U protivnom, vraća NULL.

```
BOOL wglDeleteContext( HGLRC hglrc )
```

hglrc - handle OpenGL *rendering context* koji funkcija briše

Funkcija briše OpenGL *rendering context* čiji je handle prosleden. Ako uspe, funkcija vraća TRUE. U suprotnom vraća FALSE.

BOOL wglGetCurrent(HDC hdc, HGLRC hglrc)

hdc – handle DC-a uređaja na kome će OpenGL funkcije koje slede vršiti iscrtavanje

hglrc - handle OpenGL rendering context-a koji postaje tekući za nit koja je pozvala ovu funkciju

Funkcija postavlja tekući OpenGL rendering context za nit koja je pozvala. Ako je *hglrc* NULL, funkcija oslobađa DC koji je do tada koristio OpenGL rendering context, i nakon toga deselekтуje tekući rendering context.

BOOL SwapBuffers(HDC hdc)

hdc – handle DC-a čiji se baferi menjaju

Funkcija zamenjuje prednji (*front*) i zadnji (*back*) bafer prozora, ako format piksela uključuje *back*-bafer. Funkcija vraća TRUE, ako uspe. U protivnom, vraća FALSE.

CDC* CWnd::GetDC()

Funkcija članica prozora (CWnd). Vraća pokazivač na DC datog prozora.

int CWnd::ReleaseDC(CDC* pDC)

pDC – pokazivač na DC koji se oslobađa

Funkcija članica prozora (CWnd). Oslobađa DC, čiji se pokazivač prosleđuje kao parametar. Vraća 0 ako ne uspe. U protivnom vrednost različitu od 0.

Zadatak

Napraviti MFC aplikaciju koja podržava iscrtavanje korišćenjem OpenGL-a.

Rešenje

Prikazano je rešenje generisano pomoću VS 2008. Prikazan je sadržaj sledećih datoteka: StdAfx.h, GLView.h, GLView.cpp, GLRenderer.h i GLRenderer.cpp. Kod koji je automatski generisan prikazan je *kurzivom*, a dodati ili modifikovani – normalnim ispisom. Zbog bolje čitljivosti uklonjeni su veliki delovi koda koji ne utiču na razumevanje rešenja prethodnog zadatka. Na mestu uklonjenog koda javlja se sledeći komentar:

```
/* Uklonjeni deo koda */
```

StdAfx.h

```
// stdafx.h ...  

#pragma once  

/* Uklonjeni deo koda */
```

```
// OpenGL biblioteke  

#pragma comment (lib, "OpenGL32.lib")  

#pragma comment (lib, "glu32.lib")
```

GLView.h

```
// GLView.h : interface of the CGLView class  

//  

#pragma once  

// Zaglavije GLRenderer-a
```

```
#include "GLRenderer.h"
```

```
class CGLView : public CView  

{  

protected: // create from serialization only  

    CGLView();  

    DECLARE_DYNCREATE(CGLView)
```

```
// Attributes  

public:  

    CGLDoc* GetDocument() const;
```

```
protected:  

    CGLRenderer m_glRenderer; // GLRenderer
```

```

// Operations
public:
    // Overrides
public:
    virtual void OnDraw(CDC* pDC);
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

    // Implementation
public:
    virtual ~CGLView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
protected:
    // Generated message map functions
protected:
    afx_msg void OnFilePrintPreview();
    afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnContextMenu(CWnd* pWnd, CPoint point);
    DECLARE_MESSAGE_MAP()
public:
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg BOOL OnEraseBknd(CDC* pDC);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg void OnDestroy();
    virtual void OnInitialUpdate();
};

#ifndef _DEBUG // debug version in GLView.cpp
inline CGLDoc* CGLView::GetDocument() const
{
    return reinterpret_cast<CGLDoc*>(m_pDocument);
}
#endif

```

GLView.cpp

Napomena: Obavezno skinuti komentar sa parametra funkcije OnDraw()!

```

// GLView.cpp : implementation of the CGLView class
//
#include "stdafx.h"
#include "GL.h"

#include "GLDoc.h"
#include "GLView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#endif

```

```
// CGLView

IMPLEMENT_DYNCREATE(CGLView, CView)

BEGIN_MESSAGE_MAP(CGLView, CView)
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW,
&CGLView::OnFilePrintPreview)
    ON_WM_CREATE()
    ON_WM_ERASEBKGND()
    ON_WM_SIZE()
    ON_WM_DESTROY()
END_MESSAGE_MAP()

// CGLView construction/destruction

CGLView::CGLView()
{
    // TODO: add construction code here
}

CGLView::~CGLView()
{
}

BOOL CGLView::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CView::PreCreateWindow(cs);
}

// CGLView drawing

void CGLView::OnDraw(CDC* pDC)
{
    CGLDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

    m_glRenderer.DrawScene(pDC);
}

/* Uklonjeni deo koda */

// CGLView message handlers

int CGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    CDC* pDC = GetDC();
    m_glRenderer.CreateGLContext(pDC);
    ReleaseDC(pDC);

    return 0;
}
```

```

    BOOL CGLView::OnEraseBkgnd(CDC* pDC)
    {
        return TRUE;
    }

    void CGLView::OnSize(UINT nType, int cx, int cy)
    {
        CView::OnSize(nType, cx, cy);

        CDC* pDC = GetDC();
        m_glRenderer.Reshape(pDC, cx, cy);
        ReleaseDC(pDC);
    }

    void CGLView::OnDestroy()
    {
        CView::OnDestroy();

        CDC* pDC = GetDC();
        m_glRenderer.DestroyScene(pDC);
        ReleaseDC(pDC);
    }

    void CGLView::OnInitialUpdate()
    {
        CView::OnInitialUpdate();

        CDC* pDC = GetDC();
        m_glRenderer.PrepareScene(pDC);
        ReleaseDC(pDC);
    }
}

```

GLRenderer.h

```

#pragma once

class CGLRenderer
{
public:
    CGLRenderer(void);
    virtual ~CGLRenderer(void);
    bool CreateGLContext(CDC* pDC);
    void PrepareScene(CDC* pDC);
    void Reshape(CDC* pDC, int w, int h);
    void DrawScene(CDC* pDC);
    void DestroyScene(CDC* pDC);

protected:
    HGLRC      m_hrc; //OpenGL Rendering Context
};

```

GLRenderer.cpp

```
#include "StdAfx.h"
#include "GLRenderer.h"
#include <GL\gl.h>
#include <GL\glu.h>

CGLRenderer::CGLRenderer(void)
{
}

CGLRenderer::~CGLRenderer(void)
{
}

bool CGLRenderer::CreateGLContext(CDC* pDC)
{
    PIXELFORMATDESCRIPTOR pfd ;
    memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));
    pfd.nSize     = sizeof(PIXELFORMATDESCRIPTOR);
    pfd.nVersion  = 1;
    pfd.dwFlags   = PFD_DOUBLEBUFFER | PFD_SUPPORT_OPENGL |
                    PFD_DRAW_TO_WINDOW;
    pfd.iPixelType = PFD_TYPE_RGBA;
    pfd.cColorBits = 32;
    pfd.cDepthBits = 32;
    pfd.iLayerType = PFD_MAIN_PLANE;

    int nPixelFormat = ChoosePixelFormat(pDC->m_hDC, &pfd);

    if (nPixelFormat == 0) return false;

    BOOL bResult = SetPixelFormat (pDC->m_hDC,
                                   nPixelFormat, &pfd);

    if (!bResult) return false;

    m_hrc = wglCreateContext(pDC->m_hDC);

    if (!m_hrc) return false;

    return true;
}

void CGLRenderer::PrepareScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----

    //-----
    wglMakeCurrent(NULL, NULL);
}

void CGLRenderer::DrawScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----

    //-
    glFlush();
    SwapBuffers(pDC->m_hDC);
    wglMakeCurrent(NULL, NULL);
}
```

```
}

void CGLRenderer::Reshape(CDC *pDC, int w, int h)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----

    //-----
    wglMakeCurrent(NULL, NULL);
}

void CGLRenderer::DestroyScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----

    //-----
    wglMakeCurrent(NULL, NULL);
    if(m_hrc)
    {
        wglDeleteContext(m_hrc);
        m_hrc = NULL;
    }
}
```



Crtanje primitiva

U ovoj glavi upoznaćemo se sa sintaksom OpenGL funkcija, načinom definisanja boja i grafičkih primitiva.

Sintaksa komandi

OpenGL ima vrlo uniforman način definisanja imena konstanti, tipova podataka i funkcija.

Imena svih predefinisanih konstanti podležu sledećim pravilima:

- ispisana su isključivo velikim slovima,
- reči su povezane crticama za podvlačenje i
- imaju prefiks GL_.

Na primer, predefinisana konstanta koja ukazuje da se koristi niz povezanih trouglova, ima sledeći oblik:

`GL_TRIANGLE_STRIP`

Imena funkcija sastoje se od:

- prefiksa (gl – za funkcije iz jezgra OpenGL-a i glu – za funkcije iz *Utility Library*)
- smislenog naziva funkcije, pri čemu svaka reč počinje velikim slovom i
- sufiksa, koji se sastoji od:
 - o broja parametara,
 - o tipa parametara i
 - o indikatora vektora.

Primer poziva OpenGL funkcije:

`glColor3f(red, green, blue);`

Prethodni primer ilustruje definisanje boje. To je funkcija iz jezgra OpenGL-a, na što ukazuje prefiks `gl`, definiše boju, na što ukazuje reč `Color`, i prihvata 3 parametra, koja su tipa `float`. Prvo slovo tipa parametara dodaje se kao poslednje slovo sufiksa. Ukoliko se umesto zasebnih parametara prosleduje pokazivač na vektor koji sadrži parametre, posle tipa, u sufiks se dodaje slovo `v`, da ukaže na vektor.

```
float vec[3] = {red, green, blue};
glColor3fv(vec);
```

Upravo zbog različitog broja i tipa parametara, postoji čak 32 oblika funkcije `glColor`. Boja se može zadati pomoću tri ili četiri komponente. Ako ima tri komponente, onda svaka od njih određuje intenzitet crvene (`red`), zelene (`green`) ili plave (`blue`) komponente. Ukoliko se zada i četvrta komponenta, ona određuje providnost (`alpha`), tačnije intenzitet mešanja sa bojom pozadine. Vrednost komponenata kreće se u granicama od 0.0 do 1.0 za realne tipove (jednostrukе i dvostrukе tačnosti). Zadavanje vrednosti van ovog opsega rezultuje „zaokruživanjem“ na 1.0 svih brojeva većih od 1, a na 0.0 svih brojeva manjih od 0. Za neoznačene celobrojne tipove podataka, vrši se linearno mapiranje opsega vrednosti u interval [0.0, 1.0]. Označeni celi brojevi mapiraju se u interval [-1.0, 1.0], a nakon izračunavanja efekta osvetljenja, konačna vrednost se preslikava u interval [0.0, 1.0]. Negativne vrednosti postaju 0.0, a pozitivne veće od 1 postaju 1.0. Ako se ne navede `alpha`, podrazumeva se vrednost 1.0, tj. neprovidnost. Obzirom da OpenGL funkcioniše kao konačni automat, jednom postavljena boja ostaje aktivna sve dok se ne promeni. Sledi kompletan spisak svih oblika funkcije `glColor`.

```
void glColor3b( GLbyte red, GLbyte green, GLbyte blue );
void glColor3d( GLdouble red, GLdouble green, GLdouble blue );
void glColor3f( GLfloat red, GLfloat green, GLfloat blue );
void glColor3i( GLint red, GLint green, GLint blue );
void glColor3s( GLshort red, GLshort green, GLshort blue );
void glColor3ub( GLubyte red, GLubyte green, GLubyte blue );
void glColor3ui( GLuint red, GLuint green, GLuint blue );
void glColor3us( GLushort red, GLushort green, GLushort blue );
void glColor4b( GLbyte red, GLbyte green, GLbyte blue, GLbyte alpha );
void glColor4d( GLdouble red, GLdouble green, GLdouble blue, GLdouble alpha );
void glColor4f( GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha );
void glColor4i( GLint red, GLint green, GLint blue, GLint alpha );
void glColor4s( GLshort red, GLshort green, GLshort blue, GLshort alpha );
void glColor4ub( GLubyte red, GLubyte green, GLubyte blue, GLubyte alpha );
void glColor4ui( GLuint red, GLuint green, GLuint blue, GLuint alpha );
void glColor4us( GLushort red, GLushort green, GLushort blue, GLushort alpha );
void glColor3bv( const GLbyte *v );
void glColor3dv( const GLdouble *v );
void glColor3fv( const GLfloat *v );
void glColor3iv( const GLint *v );
void glColor3sv( const GLshort *v );
void glColor3ubv( const GLubyte *v );
void glColor3uiv( const GLuint *v );
void glColor3usv( const GLushort *v );
void glColor4bv( const GLbyte *v );
void glColor4dv( const GLdouble *v );
void glColor4fv( const GLfloat *v );
void glColor4iv( const GLint *v );
void glColor4sv( const GLshort *v );
```

```
void glColor4ubv( const GLubyte *v );
void glColor4uiv( const GLuint *v );
void glColor4usv( const GLushort *v );
```

Da bi se izbeglo vezivanje za određeni oblik funkcije, u daljem tekstu, osim kada se prikazuje konkretni kod, sufiks će biti zamjenjen zvezdicom (npr. glColor*). U tabeli 2 dat je pregled slova koja se mogu javiti u sufiku, a vezana su za tip argumenata funkcije.

Tabela 2. Tipovi argumenata funkcija

Slovo u sufiku	Duzina podatka b	Tip u jeziku C	Tip u OpenGL-u
b	8	char	GLbyte
s	16	short	GLshort
i	32	int/long	GLint, GLsizei
f	32	float	GLfloat, GLclampf
d	64	double	GLdouble, GLclampd
ub	8	unsigned char	GLubyte, GLboolean
us	16	unsigned short	GLushort
ui	32	unsigned int/long	GLuint, GLenum, GLbitfield

Temena

Čak i najfantastičnije scene generisane pomoću računarske grafike sastavljene su od jednostavnih oblika: tačaka, linija i poligona. Ove jednostavne oblike nazvaćemo **geometrijskim primitivama**. OpenGL sve geometrijske primitive definiše preko njihovih temena, a osnovna karakteristika svakog temena jesu njegove prostorne koordinate. One određuju položaj tačaka, krajeva linijskih segmenata i temena poligona u prostoru. Prostorne koordinate temena definišu se pozivom funkcije **glVertex***0.

U primeru

```
glVertex3f(-1.0, -2.0, -5.0);
```

definisano je teme sa sledećim koodinatama: x = -1.0, y = -2.0 i z = -5.0. Teme može biti zadato sa 2, 3 ili 4 koordinate. Ukoliko je zadato samo sa 2 kooinate, podrazumeva se da je z = 0.0. U tom slučaju definicija prostornih koordinata temena imala bi sledeći oblik:

```
glVertex2f(-1.0, -2.0);
```

Zbog uniformnosti izračunavanja transformacija, i olakšavanja nekih operacija, često se umesto trodimenzionalnih Euklidovih koordinata koriste **homogene koordinate**. U homogenom sistemu prostorna tačka ima četiri koordinate: x, y, z i w. Položaj tačke zadate homogenim koordinatama u Euklidovom prostoru dobija se deljenjem prve tri koordinate sa w, ukoliko je w ≠ 0. Ako je w = 0, koordinate definišu pravac (vektor), a ne tačku. Ako se w ne navede, podrazumeva se w = 1.

Funkcija **glVertex***0 definiše teme, ali ne određuje da li to teme predstavlja tačku, kraj linijskog segmenta ili ugao poligona. To određuju funkcije-zagrade: **glBegin()** i **glEnd()**.

Poziv funkcije `glVertex*`0 ima smisla samo ako se nalazi između poziva ove dve funkcije. Argument koji se prosledjuje kao parametar funkcije `glBegin()` određuje tip primitiva. Tabela 3 prikazuje sve vrednosti argumenta funkcije `glBegin()` i tipove primitiva koji definisu.

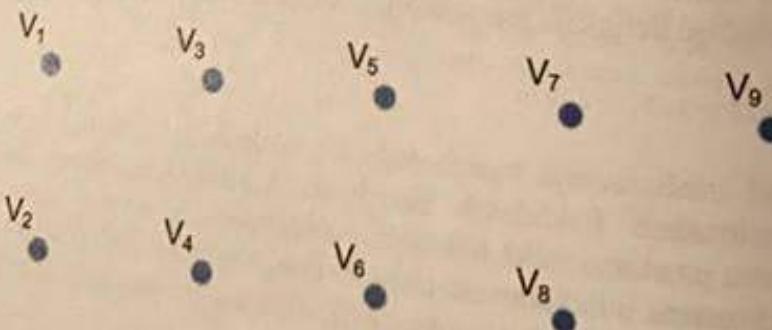
Tabela 3. Vrednosti parametra funkcije `glBegin`

Parametar	Tip primitive
GL_POINTS	Pojedinačne tačke
GL_LINES	Pojedinačne duži, definisane sa po dve tačke
GL_LINE_STRIP	Serijski medusobno povezanih duži
GL_LINE_LOOP	Serijski medusobno povezanih duži, pri čemu su povezane i prva i poslednja, tako da formiraju petlju
GL_TRIANGLES	Pojedinačni trouglovi, definisani grupama od po 3 temena
GL_TRIANGLE_STRIP	Traka sačinjena od trouglova koji dele po jednu stranicu
GL_TRIANGLE_FAN	Lepeza sačinjena od trouglova
GL_QUADS	Pojedinačni četvorouglovi
GL_QUAD_STRIP	Traka sačinjena od četvorouglova
GL_POLYGON	Jednostavan konveksan poligon (višegao)

Razmotrimo šta se dešava ako ista temena prosledimo različitim primitivama. Sledi primer koda koji definiše devet različitih temena.

```
glLineWidth(2.0);
glPointSize(10);
glBegin(GL_POINTS);
    glColor3f(0.5, 0.5, 1.0);
    glVertex2f(-2.0, 1.0); // V1
    glVertex2f(-2.0, 0.0); // V2
    glVertex2f(-1.0, 1.0); // V3
    glVertex2f(-1.0, 0.0); // V4
    glVertex2f( 0.0, 1.0); // V5
    glVertex2f( 0.0, 0.0); // V6
    glVertex2f( 1.0, 1.0); // V7
    glVertex2f( 1.0, 0.0); // V8
    glVertex2f( 2.0, 1.0); // V9
glEnd();
```

Obzirom da je u funkciji `glBegin` navedeno da se crtaju pojedinačne tačke, dobija se figura prikazana na slici 2.1.

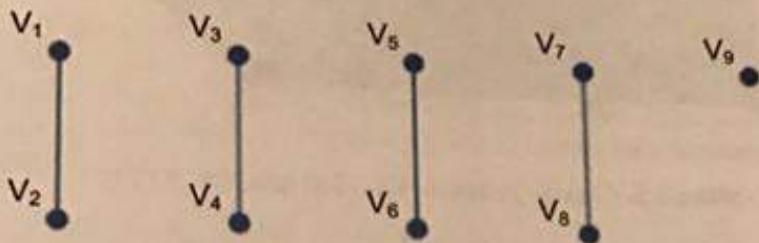


Slika 2.1. Crtanje primitive GL_POINTS

Ako umest...
drugačija f...
prikazane s...
koji je for...

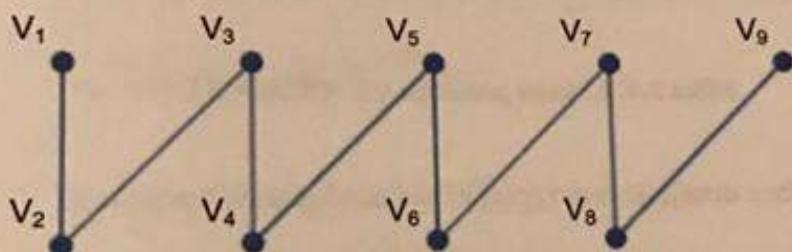
Na slika...
ona po...
GL_ QU...

Ako umesto `glBegin(GL_POINTS)` napišemo `glBegin(GL_LINES)`, dobija se sasvim drugačija figura. Uместо devet odvojenih tačaka, sada dobijamo četiri duži. Na slici 2.2 prikazane su i pozicije temena (koja se inače ne vide), zbog lakšeg prepoznavanja načina na koji je formirana figura.

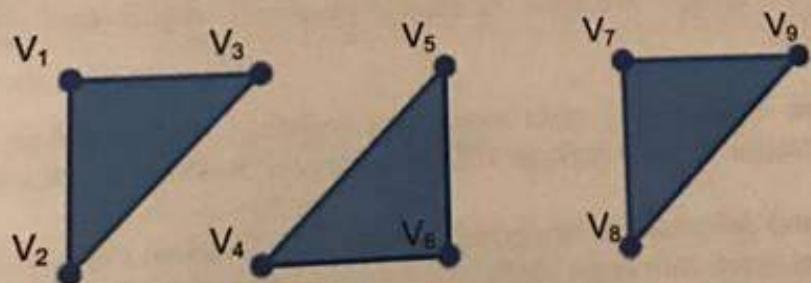


Slika 2.2. Crtanje primitive GL_LINES

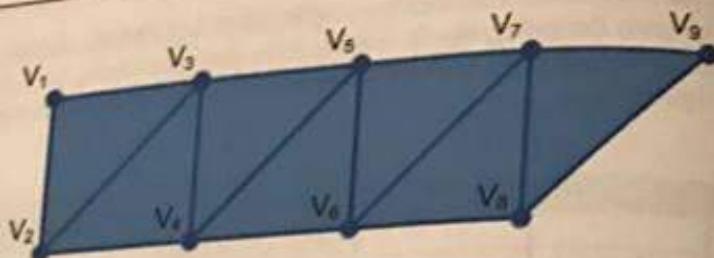
Na slikama 2.3 – 2.6 mogu se videti figure dobijene korišćenjem istih temena, ukoliko se ona povežu u: **GL_LINE_STRIP**, **GL_TRIANGLES**, **GL_TRIANGLE_STRIP** i **GL_QUAD_STRIP** primitive, respektivno.



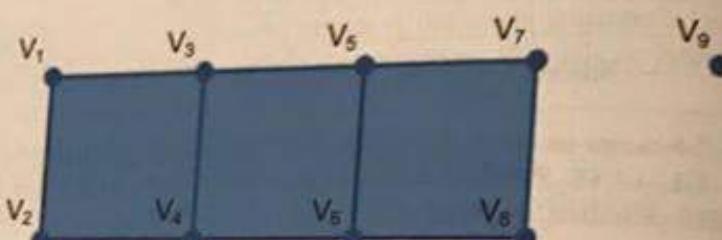
Slika 2.3. Crtanje primitive GL_LINE_STRIP



Slika 2.4. Crtanje primitive GL_TRIANGLES



Slika 2.5. Crtanje primitive GL_TRIANGLE_STRIP



Slika 2.6. Crtanje primitive GL_QUAD_STRIP

U nastavku ove glave detaljnije ćemo proučiti svaku od grafičkih primitiva.

Priprema projekta za crtanje primitiva

Kao što je već objašnjeno u prethodnoj glavi, da bismo mogli da koristimo OpenGL moramo uraditi određene pripreme. Te pripreme se sastoje u definisanju klase **CGLRender**, povezivanju ove klase sa klasom prozora i uključivanju odgovarajućih biblioteka i zaglavljia.

Projekat napravljen u prethodnoj glavi može nam koristiti i u ovoj, jedino je potrebno izmeniti implementaciju sledećih funkcija **CGLRender**-a: **Reshape()** i **DrawScene()**.

U funkciji **Reshape()** definisaćemo ortogonalnu projekciju. Detaljno objašnjenje projekcija sledi tek u narednoj glavi, zato ćemo sledeći kod preuzeti bez ikakve analize. Treba samo naglasiti da se koordinatni početak nalazi u središtu prozora, i da bi X i Y koordinate trebalo navoditi u opsegu od -2 do 2 da bi prikaz bio vidljiv na ekranu.

```
void CGLRender::Reshape(CDC *pDC, int w, int h)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //...
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    double aspect = (double)w/(double)h;
```

Funkcija I
pisali pre
DrawPrin
čitav kod
DrawPrin

Tačke

Tačke se
funkcije
funkciji g
na koordi
na sledeći

Nakon p
nije spek
Pažljiviji
pogrešili
je tri tač
ne nalož
gotovo p

```

    glOrtho(-H*aspect, H*aspect, -H, H, 0.5, 2.0);
    glMatrixMode(GL_MODELVIEW);
    //-----
    wglMakeCurrent(NULL, NULL);
}

```

Funkcija **DrawScene()** vrši samo iscrtavanje scene. Da ne bismo u okviru ove funkcije pisali proizvoljan kod i učinili je nepreglednom, uvešćemo novu funkciju – čitav kod vezan za crtanje primitiva (prikazan u ovoj glavi) biće implementacija funkcije **DrawPrimitives()**.

```

void CGLRenderer::DrawScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -1.0);

    DrawPrimitives();

    glFlush();
    //-----
    SwapBuffers(pDC->m_hDC);
    wglMakeCurrent(NULL, NULL);
}

```

Tačke

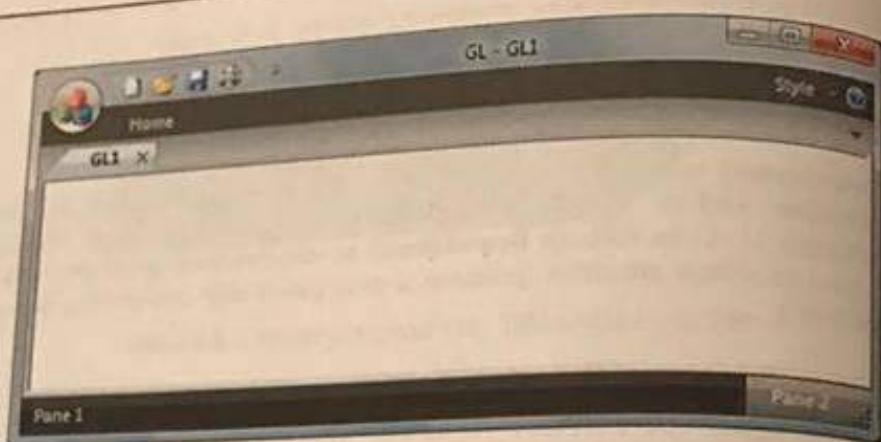
Tačke se u OpenGL-u crtaju definisanjem njihovih prostornih koordinata u pozivima funkcije **glVertex***0, unutar **glBegin()/glEnd()** bloka, a parametar koji se prosleđuje funkciji **glBegin()** je **GL_POINTS**. Na primer, ukoliko želimo nacrtati tri tačke plave boje na koordinatama (0,0), (1,0) i (0,1) definišimo implementaciju **DrawPrimitives()** funkcije na sledeći način:

```

void CGLRenderer::DrawPrimitives()
{
    glBegin(GL_POINTS);
        glColor3f(0.0, 0.0, 1.0);
        glVertex2f(0.0, 0.0);
        glVertex2f(1.0, 0.0);
        glVertex2f(0.0, 1.0);
    glEnd();
}

```

Nakon prevođenja i aktiviranja programa, dobija se prozor prikazan na slici 2.7. Rezultat nije spektakularan, jer se na prvi pogled ništa ne vidi na beloj radnoj površini prozora. Pažljivijim gledanjem mogu se uočiti tri vrlo male tačke u središtu prozora. Gde smo pogrešili? Odgovor je – nigde! OpenGL je uradio tačno ono što smo mu i naložili. Nacrtao je tri tačke. Matematički posmatrano, tačka je infinitezimalni entitet. Ako mu se drugačije ne naloži, OpenGL crta tačke veličine jednog piksela. Uz umekšavanje prikaza, tačke se gotovo potpuno stapanju sa pozadinom.



Slika 2.7. Prvi pokušaj crtanja tačaka

Da bismo omogućili crtanje većih tačaka, u pomoć ćemo pozvati funkciju `glPointSize`. Kao parametar ove funkcije prosleđuje se željena veličina tačke u pikselima.

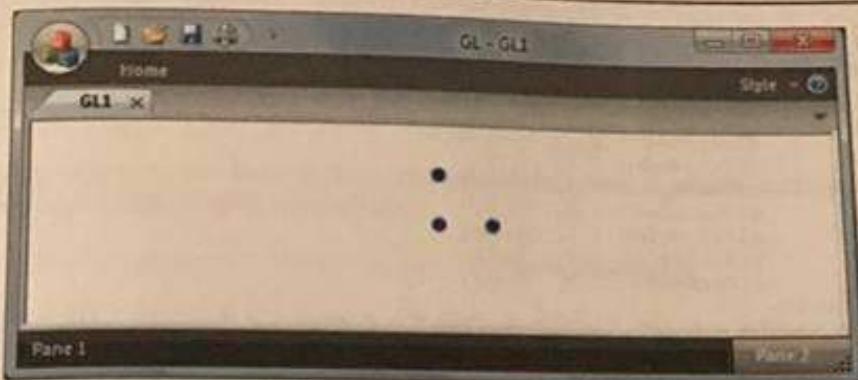
```
void glPointSize(GLfloat size)
```

Parametar je realnog tipa, i mora biti veći od nule. Podrazumevana vrednost je 1. Kolika će stvarna veličina tačke biti zavisi od toga da li je uključen *antialiasing*. *Antialiasing* je tehniku umekšavanja ivica, tako što se pikseli na rubovima primitiva mešaju sa bojom pozadine. Na taj način se postiže efekat daleko veće rezolucije od one koja je fizički prisutna. Ako je *antialiasing* isključen, parametar koji se prosleđuje zaokružuje se na najbliži ceo broj.

Modifikujmo sada crtanje, tako da „tačke“ budu veličine 10 piksela.

```
void CGLRenderer::DrawPrimitives()
{
    glPointSize(10);
    glBegin(GL_POINTS);
        glColor3f(0.0, 0.0, 1.0);
        glVertex2f(0.0, 0.0);
        glVertex2f(1.0, 0.0);
        glVertex2f(0.0, 1.0);
    glEnd();
}
```

Uместо tačaka sada se crtaju kružići, sa centrom na poziciji tačke i zadatim prečnikom (slika 2.8).



Slika 2.8. Crtanje tačaka veličine 10 piksela

Često postoji potreba da se očita vrednost neke promenljive koja definiše stanje u OpenGL-u. Za to se koriste funkcije: `glGetBooleanv()`, `glGetDoublev()`, `glGetFloatv()` i `glGetIntegerv()`. Prvi parametri ovih funkcija su nazivi promenljivih, a drugi – povratne vrednosti.

Ako želimo očitati tekuću veličinu tačke, izvršićemo sledeći kod:

```
float param;
glGetFloatv(GL_POINT_SIZE, &param);
```

Dakle, obzirom da je veličina tačke realna vrednost, poziva se funkcija `glGetFloatv()`. Prvi parametar je predefinisana konstanta `GL_POINT_SIZE`, a drugi promenljiva tipa `float` u koju se smešta vrednost veličine tačke.

Rezolucija sa kojom se može menjati veličina tačke dobija se sledećim pozivom:

```
glGetFloatv(GL_POINT_SIZE_GRANULARITY, &param);
```

a opseg u kojem se može menjati vrednost veličine tačke:

```
float params[2];
glGetFloatv(GL_POINT_SIZE_RANGE, params);
```

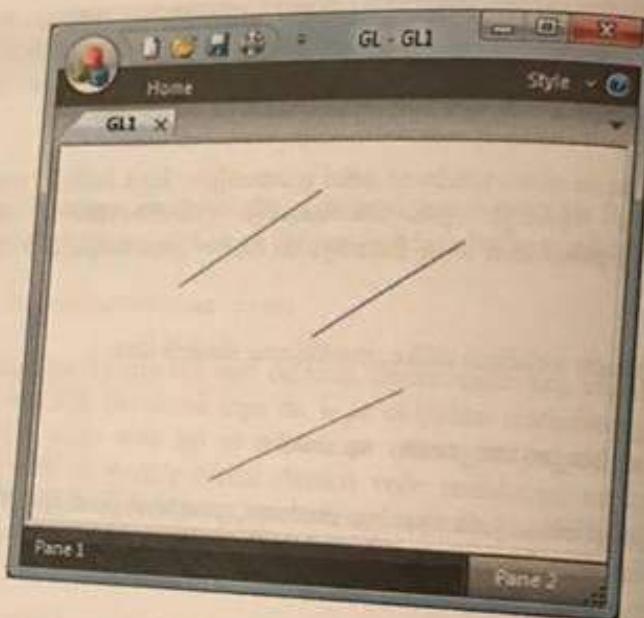
Opseg vrednosti ne može se zapisati u jednoj promenljivoj, pa se zato prosleduje pokazivač na vektor koga čine dva realna broja. Donja i gornja granica opsega zavise od implementacije. Tipične vrednosti su 1.0 za donju i 63.375 za gornju granicu, a rezolucija sa kojom se menjaju vrednosti je 0.125. To znači da se ne može formirati tačka, na primer, veličine 2.2, već će se ona „zaokružiti“ na 2.25.

Duži

Ukoliko kao parametar funkcije `glBegin()` prosledimo konstantu `GL_LINES`, temena čije se prostorne koordinate navode pozivima funkcija `glVertex*` definisu niz nepovezanih

duži. Prvo i drugo teme definišu prvu duž, treće i četvrto drugu, itd. Ako broj temenih paran, zadnje teme se odbacuje.

```
glBegin(GL_LINES);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2d( 0.0, 1.5);
    glVertex2d(-1.5, 0.5);
    glVertex2d(-1.0, -1.5);
    glVertex2d( 1.0, -0.5);
    glVertex2d( 1.5, 1.0);
    glVertex2d( 0.0, 0.0);
glEnd();
```



Slika 2.9. Crtanje linija

Linije ne moraju biti jedinične debljine, niti fiksne ispune. Debljina linija definise se pozivom funkcije `glLineWidth()`.

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

Jedini parametar ove funkcije je debljina linije u pikselima i mora biti veća od 0. Podrazumevana vrednost je 1. Parametri linije mogu se očitati funkcijom `glGetFloat()` posledivanjem sledećih parametara:

- `GL_LINE_WIDTH` – tekuća debljina linije,
- `GL_LINE_WIDTH_GRANULARITY` – korak sa kojim se može menjati debljina linije, i
- `GL_LINE_WIDTH_RANGE` – minimalna i maksimalna vrednost debljine linije.

Kao i na tačke, i na debljinu linija utiče to da li je uključen *antialiasing* ili ne. Ako nije debljina linija mora biti celobrojna i meri su u Y-pravcu, ako je apsolutna vrednost nagiba manja od 1, a u X-pravcu ako je veća od 1. Kada je uključen *antialiasing*, crtanje linije vrši se crtanjem pravougaonika čija širina odgovara debljini linije.

Umeto kons
Šablon se de
iscrtati, a nu
primer, ako j

Šablon linije
kojim se mn

voi

Faktor omog
fizičke piks
iscrtana, za
množenjem

Šablon koji
funkcije `glC`

- `GL`
- `GL`

Inicijalno je
je pozvati g

gl

Da li je
glGetBook
primeru:

Gl

g

Sledeći p
šablon. R

g

g

g

g

Uместо konstantne ispune, moguće je definisati i šablon koji se koristi za iscrtavanje linija. Šablon se definiše 16-bitnim nizom nula i jedinica. Jedinice označavaju piksele koje treba iscrtati, a nule one koje ne treba. Linija se iscrtava počinjući od bita najmanje težine. Na primer, ako je šablon 0xF0F0, prva 4 piksela neće biti iscrtana, zatim sledeća 4 hoće itd.

Šablon linije zadaje se pozivom funkcije `glLineStipple()`. Prvi parametar funkcije je faktor kojim se množi šablon, a drugi sam šablon.

```
void glLineStipple(GLint factor, GLushort pattern)
```

Faktor omogućuje da se šablon produži. Ukoliko je faktor 1, bitovi u šablonu označavaju fizičke piksele. Ako je, na primer, faktor 2 a šablon 0xF0F0, prvih 8 piksela neće biti iscrtana, zatim sledećih 8 hoće itd. Dakle, broj piksela koji se iscrtava ili ne dobija se množenjem broja jedinica i nula faktorom.

Šablon koji se koristi za iscrtavanje linije i odgovarajući faktor mogu se očitati pozivom funkcije `glGetIntegerv()` sa prosleđenim sledećim parametrima:

- `GL_LINE_STIPPLE_PATTERN` – tekući šablon i
- `GL_LINE_STIPPLE_REPEAT` – faktor množenja (ponavljanja) šablosa.

Inicijalno je iscrtavanje linija korišćenjem šablosa isključeno. Da bi se uključilo, potrebno je pozvati `glEnable()` sa parametrom `GL_LINE_STIPPLE`.

```
glEnable(GL_LINE_STIPPLE);
```

Da li je uključeno korišćenje šablosa možemo proveriti pozivom funkcije `glGetBooleanv()`, prosleđivanjem parametra `GL_LINE_STIPPLE`, kao u narednom primeru:

```
GLboolean value;
glGetBooleanv(GL_LINE_STIPPLE, &value);
```

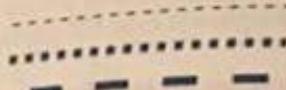
Sledeći programski kod demonstrira efekte promene debljine linije i faktora množenja šablosa. Rezultat je prikazan na slici 2.10.

```
glLineWidth(1.0);
glLineStipple(1, 0xF0F0);
glEnable(GL_LINE_STIPPLE);
glBegin(GL_LINES);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2f(-2.0, 0.0);
    glVertex2f( 2.0, 0.0);

glEnd();
glLineWidth(4.0);
glBegin(GL_LINES);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2f(-2.0, -0.5);
    glVertex2f( 2.0, -0.5);

glEnd();
glLineStipple(4, 0xF0F0);
glEnable(GL_LINE_STIPPLE);
glBegin(GL_LINES);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2f(-2.0, -1.0);
```

```
glVertex2f( 2.0, -1.0);
glEnd();
```



Slika 2.10. Crtanje linija korišćenjem šablonu

OPENGL - FIKSNA

Petlje

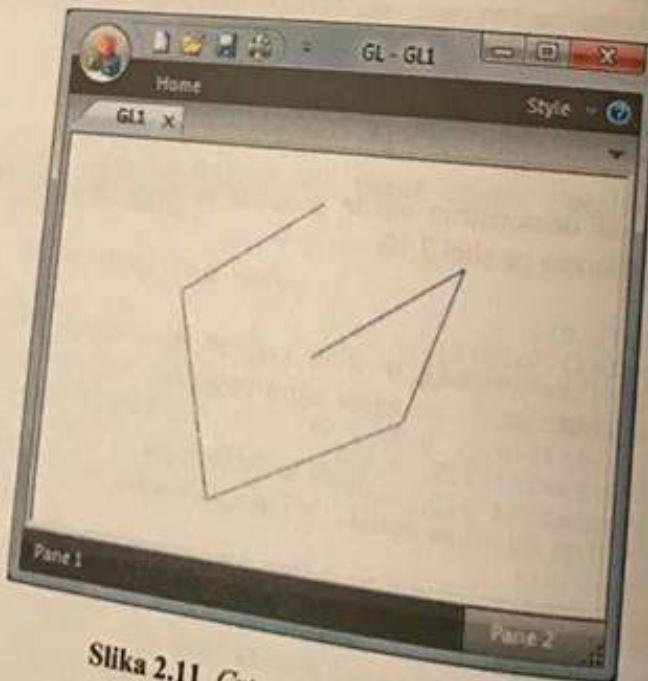
Petlje se formiraju na petlje iscrtava još jed funkciji `glBegin()` treb

```
glBegin(GL_LINES);
glColor3f(0.0, 0.0, 1.0);
glVertex2d( 0.0, 1.5);
glVertex2d(-1.5, 0.5);
glVertex2d(-1.0, -1.5);
glVertex2d( 1.0, -0.5);
glVertex2d( 1.5, 1.0);
glVertex2d( 0.0, 0.0);
glEnd();
```

Povezane duži

Temena zadata nakon poziva funkcije `glBegin(GL_LINE_STRIP)` definisu poligonsku liniju, odnosno niz duži koje su međusobno povezane. Prvo i drugo teme definisu prvu duž, drugo i treće definisu drugi duž, treće i četvrto – treću, itd. Navodenje samo jednog teme ne iscrtava ništa na ekranu.

```
glBegin(GL_LINE_STRIP);
glColor3f(0.0, 0.0, 1.0);
glVertex2d( 0.0, 1.5);
glVertex2d(-1.5, 0.5);
glVertex2d(-1.0, -1.5);
glVertex2d( 1.0, -0.5);
glVertex2d( 1.5, 1.0);
glVertex2d( 0.0, 0.0);
glEnd();
```



Slika 2.11. Crtanje povezanih duži

Sve što je rečeno za crtanje duži, tj. linija, važi i za povezane duži. Navodenje N temena formira N-1 duž. Duži se mogu presecati i/ili poklapati.

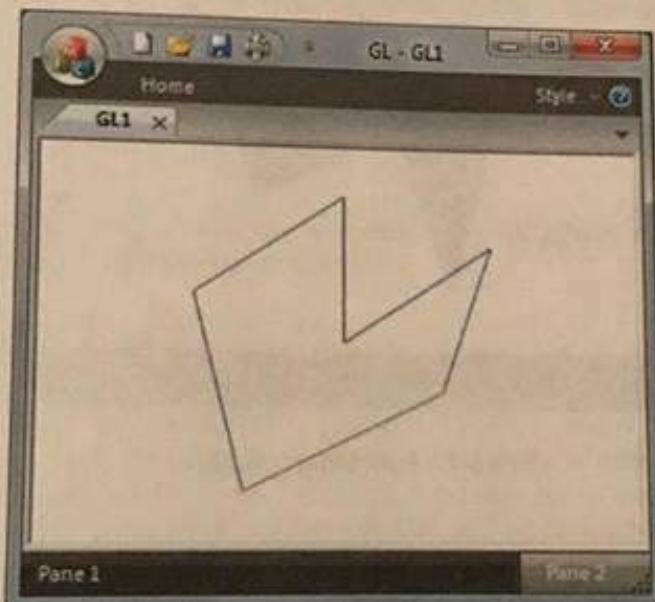
Trouglovi

Temena definisana nepovezanim trouglovi definisu drugi trouglovi (dva) se ignorise.

Petlje

Petlje se formiraju na isti način kao povezane duži. Jedina razlika je u tome što se u slučaju petlje iscrta još jedna duž koja spaja poslednje i prvo teme. Da bi se formirala petlja, funkciji **glBegin()** treba proslediti predefinisanu konstantu **GL_LINE_LOOP**.

```
glBegin(GL_LINE_LOOP);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2d( 0.0, 1.5);
    glVertex2d(-1.5, 0.5);
    glVertex2d(-1.0, -1.5);
    glVertex2d( 1.0, -0.5);
    glVertex2d( 1.5, 1.0);
    glVertex2d( 0.0, 0.0);
glEnd();
```



Slika 2.12. Crtanje petlje

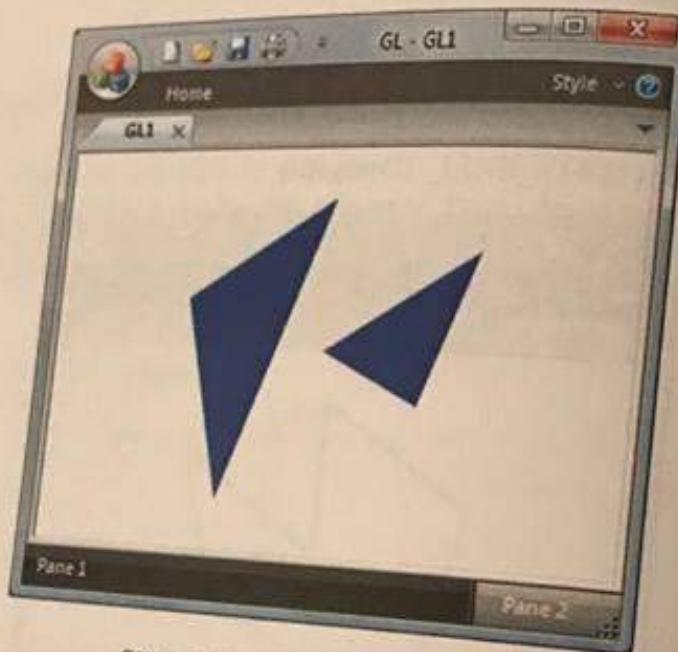
Trouglovi

Temena definisana nakon poziva funkcije **glBegin(GL_TRIANGLES)** formiraju niz nepovezanih trouglova. Prvo, drugo i treće teme definišu prvi trougao, četvrto, peto i šesto definišu drugi trougao, itd. Ukoliko broj temena nije deljiv sa tri, višak temena (jedno ili dva) se ignoriše.

```

glBegin(GL_TRIANGLES);
    glColor3f(0.0, 0.0, 1.0);
    glVertex2d( 0.0, 1.5);
    glVertex2d(-1.5, 0.5);
    glVertex2d(-1.0, -1.5);
    glVertex2d( 1.0, -0.5);
    glVertex2d( 1.5, 1.0);
    glVertex2d( 0.0, 0.0);
glEnd();

```



Slika 2.13. Crtanje trouglova

Trouglovi, i generalno svi poligoni, obično se iscrtavaju ispunjeno. Međutim, moguće je iscrtati samo ivice u obliku linija ili temena u obliku tačaka. Za definisanje načina iscrtavanja koristi se funkcija `glPolygonMode()`.

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

Prvi parametar je strana poligona za koju važi dati način iscrtavanja, a drugi parametar je sam način iscrtavanja. Strana poligona može biti: **GL_FRONT** (prednja), **GL_BACK** (zadnja) ili **GL_FRONT_AND_BACK** (i prednja i zadnja). Iscrtavanje može biti **GL_POINT** (tačke), **GL_LINE** (linije) ili **GL_FILL** (ispuna).

Šta se podrazumeva pod prednjom, a šta pod zadnjom stranom poligona? Strana na kojoj se temena pojavljuju u redosledu suprotnom od kretanja kazaljki na satu smatra se prednjom stranom. Ovo je pravilo „desnog zavrtnja“. Smer u kome napreduje zavrtanj, pri okretanju u smeru zadavanju temena odgovara prednjoj strani. Na slici 2.14 levi trougao vidi se sa prednje, a desni sa zadnje strane. Pretpostavka je da su temena zadata u redosledu: V1, V2, V3.

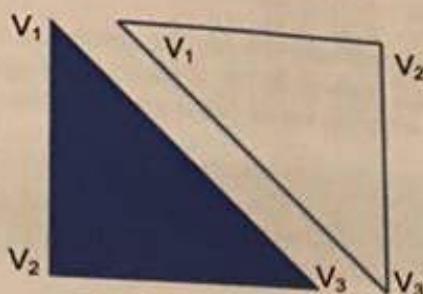
Način
„poziti
GL_C
časovn
orienta

Ako ž
okviri

Veoma
slučaje
strana,
vidi, ne
ta stran
izboron
prednje

Trake

Trake tr
zajednič
funkcije
čemu je



Slika 2.14. Prednja i zadnja strana poligona (ispunjena je prednja strana)

Način određivanja prednje strane može se promeniti. Funkcija `glFrontFace()` definiše „pozitivnu orientaciju“, tj. orientaciju prednje strane. Prosleđivanjem parametra `GL_CCW` prednja strana postaje ona koja ima orientaciju suprotnu od kazaljki na časovniku (ovo je podrazumevano stanje), a prosleđivanjem `GL_CW` prednja strana ima orientaciju u pravcu kazaljki.

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

Ako želimo da prednje strane iscrtanih poligona budu ispunjene, a zadnje korišćenjem okvirljih linija, koristimo sledeći programski kod:

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

Veoma često nema potrebe iscrtavati i prednju i zadnju stranu poligona. U takvim slučajevima značajno možemo ubrzati iscrtavanje čitave scene zabranjujući da se jedna strana, najčešće je to zadnja (`GL_BACK`), iscrtava. Obzirom da se zadnja strana obično ne vidi, nema potrebe trošiti procesorsko vreme grafičke kartice na izračunavanje kako izgleda ta strana. Ovo se postiže uključivanjem „odbacivanja stranica“ (`GL_CULL_FACE`), i izborom stranice koja se odbacuje (prednje – `GL_FRONT`, zadnje – `GL_BACK` ili i prednje i zadnje – `GL_FRONT_AND_BACK`).

```
 glEnable(GL_CULL_FACE);
 glCullFace(GL_BACK);
```

Trake trouglova

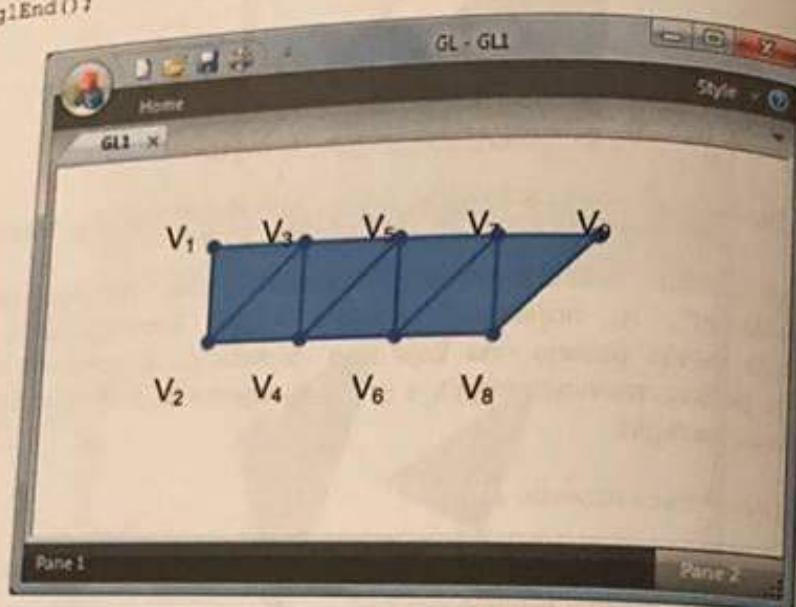
Trake trouglova predstavljaju nizove trouglova, takvih da po dva susedna trougla imaju dva zajednička temena. Zadaju se nizom temena čije prostorne koordinate slede nakon poziva funkcije `glBegin(GL_TRIANGLE_STRIP)`. Ako se traka definise pomoću n temena, pri čemu je $n \geq 3$, biće iscrtana $n-2$ trougla.

```
glBegin(GL_TRIANGLE_STRIP);
 glColor3f(0.5, 0.5, 1.0);
 glVertex2f(-2.0, 1.0); // V1
 glVertex2f(-2.0, 0.0); // V2
 glVertex2f(-1.0, 1.0); // V3
 glVertex2f(-1.0, 0.0); // V4
```

```

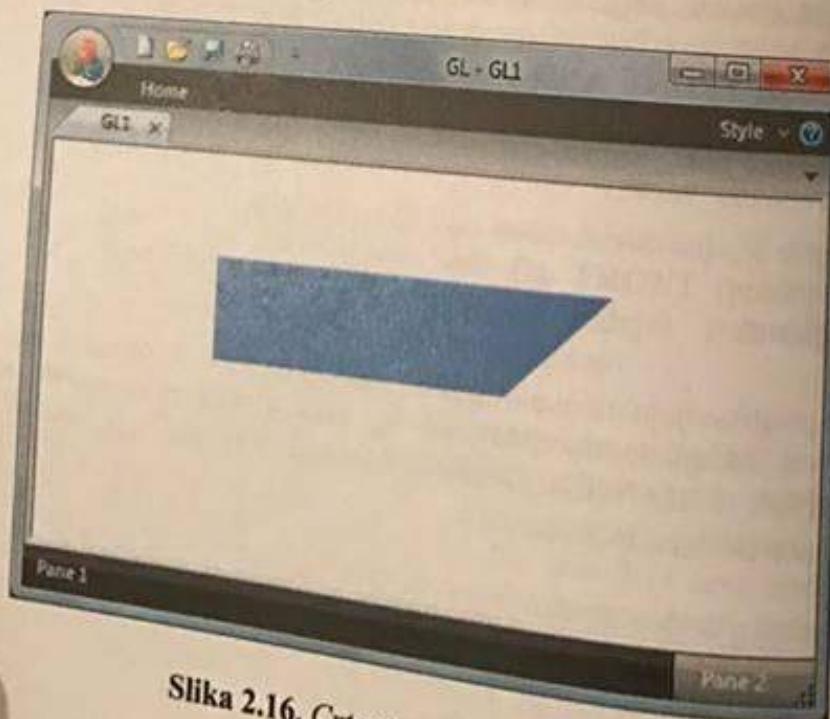
glVertex2f( 0.0, 1.0); // V5
glVertex2f( 0.0, 0.0); // V6
glVertex2f( 1.0, 1.0); // V7
glVertex2f( 1.0, 0.0); // V8
glVertex2f( 2.0, 1.0); // V9
glEnd();

```



Slika 2.15. Crtanje trake trouglova – proces formiranja trake

Prva tri temena (V_1 , V_2 i V_3) definišu prvi trougao, drugo (V_2), treće (V_3) i četvrti (V_1) teme definišu drugi, treće (V_3), četvrti (V_4) i peto (V_5) definišu treći trougao, itd. Minimalan broj temena koja se moraju zadati je tri, i tada se iscrtava samo jedan trougao. Svako sledeće teme dodaje po jedan trougao. Na slici 2.15 dodatno su prikazana temena V_6 do V_9 u svrhu označavanja ivica trouglova, kako bi proces formiranja trake trouglova bio jasaniji. Slika 2.16 daje realan prikaz iste trake trouglova.



Slika 2.16. Crtanje trake trouglova

Za razliku od pojedinačnog pozivanja funkcije `glBegin(GL_TRIANGLES)` i jednog novoupočetog pozivanja funkcije `glBegin(GL_TRIANGLE_STRIP)` u ovom slučaju se poziva funkcija `glBegin(GL_TRIANGLE_FAN)`.

Lepeze trouglova

Za razliku od trapeza, trouglovi su jednostavniji oblici. U OpenGL-u postoji nekoliko načina na koji se mogu izvoditi trouglovi. Neke od njih su:

`glBegin(GL_TRIANGLES)`

`glEnd()`

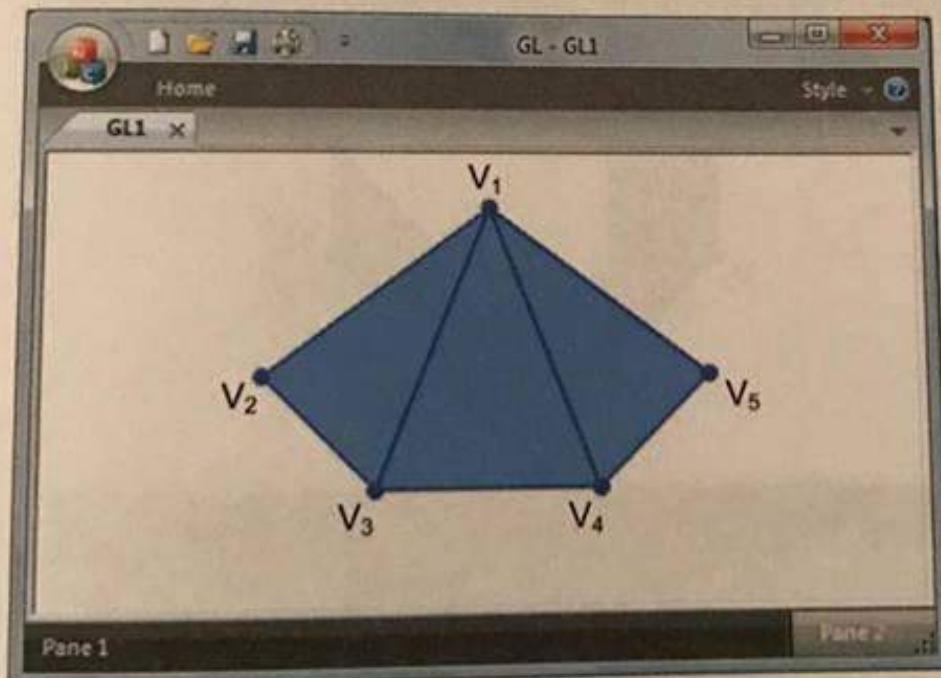
Kao i kod trapeza, u ovom slučaju se poziva funkcija `glBegin(GL_TRIANGLE_FAN)`. Tada se iscrtavaju temena (V_1 , V_2 , V_3 , V_4 , V_5 , V_6 , V_7 , V_8 , V_9) u redoslijedu definisanju, a posljednje dve temene (V_8 i V_9) definisu drugi trougao.

Za razliku od pojedinačnih trouglova, trouglovi u traci naizmenično menjaju orientaciju prednje strane. To omogućuje konzistentnost pri iscrtavanju, jer bi u protivnom svaki drugi trougao bio nevidljiv pri iscrtavanju samo jedne strane poligona.

Lepeze trouglova

Za razliku od trake trouglova, gde se novi trougao formira na osnovu dva prethodna temena i jednog novounetog, kod lepeze se novi trougao kreira od **prvog, prethodnog i novounetog temena**. Kreira se posledivanjem predefinisane konstante **GL_TRIANGLE_FAN** funkciji **glBegin()**.

```
glBegin(GL_TRIANGLE_FAN);
    glColor3f(0.5, 0.5, 1.0);
    glVertex2f( 0.0, 1.5); // V1
    glVertex2f(-2.0, 0.0); // V2
    glVertex2f(-1.0,-1.0); // V3
    glVertex2f( 1.0,-1.0); // V4
    glVertex2f( 2.0, 0.0); // V5
glEnd();
```



Slika 2.17. Crtanje lepeze trouglova

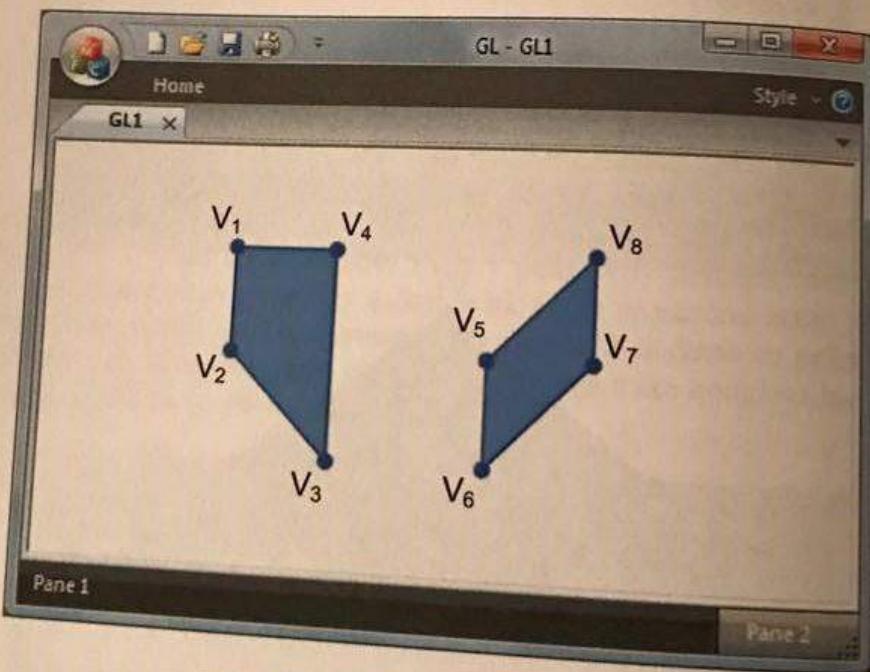
Kao i kod trake trouglova, i kod lepeze minimalan broj temena koja se moraju zadati je tri, i tada se iscrtava samo jedan trougao. Svako sledeće teme dodaje po jedan trougao. Prva tri temena (V1, V2 i V3) definišu prvi trougao, prvo (V1), treće (V3) i četvrto (V4) teme definišu drugi, prvo (V1), četvrto (V4) i peto (V5) definišu treći trougao, itd.

Četvorouglovi

Crtanje četvorouglova ostvaruje se navođenjem koordinata temena unutar bloka koji počinje pozivom funkcije `glBegin(GL_QUADS)`. Svaki četvorougao zadaje se sa po četiri zasebna temena. Ako broj temena između `glBegin()` i `glEnd()` funkcija nije deljiv sa četiri, višak od jednog do tri temena se odbacuje. Ukoliko se ne promeni podrazumevana orientacija, temena bi trebalo zadati u smeru suprotnom od kretanja kazaljki na časovniku.

```
glBegin(GL_QUADS);
    glColor3f(0.5, 0.5, 1.0);
    glVertex2f(-2.0, 1.0); // V1
    glVertex2f(-2.0, 0.0); // V2
    glVertex2f(-1.0,-1.0); // V3
    glVertex2f(-1.0, 1.0); // V4

    glVertex2f( 0.5, 0.0); // V5
    glVertex2f( 0.5,-1.0); // V6
    glVertex2f( 1.5, 0.0); // V7
    glVertex2f( 1.5, 1.0); // V8
glEnd();
```



Slika 2.18. Crtanje četvorouglova

Prilikom crtanja četvorouglova treba voditi računa da se stranice ne presecaju i da četvorougao bude konveksan. Ukoliko to nije ispunjeno, prikaz može biti vrlo nepredvidiv. Takođe, sva četiri temena jednog četvorougla ne moraju biti u istoj ravni. I ovo predstavlja neželjenu pojavu, jer se prilikom transformacija mogu dobiti nekonveksne figure, ili figure čije se ivice uzajamno presecaju. Zato je gotovo uvek poželjnije koristiti trouglove prilikom crtanja primitiva. Tri temena trougla uvek pripadaju jednoj ravni, i nikada se stranice trougla ne mogu uzajamno presecati. Mnoge grafičke kartice, zbog jednostavnosti i ubrzavanja iscrtavanja, mogu prikazivati samo trouglove. Svi složeniji oblici prevode se najpre u trouglove, pre finalnog prikazivanja.

OPENGL
Trake i
Trake će
posle po
četvorou
temena n

Trake četvorouglova

Trake četvorouglova su slične trakama trouglova. Prostorne koordinate temena navode se posle poziva **glBegin(GL_QUAD_STRIP)**, pri čemu prva četiri temena određuju prvi četvorougao, a svaki naredni dva prethodna i dva novododata temena. Ako je ukupan broj temena neparan, zadnje teme se ignoriše.

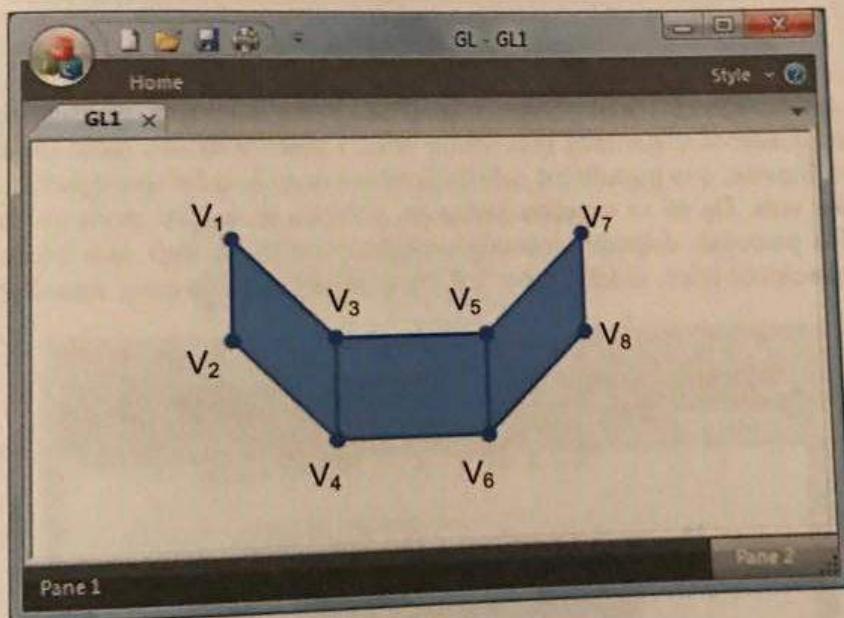
```
glBegin(GL_QUAD_STRIP);
    glColor3f(0.0, 0.0, 1.0);

    glVertex2f(-2.0, 1.0); // V1
    glVertex2f(-2.0, 0.0); // V2

    glVertex2f(-1.0, 0.0); // V3
    glVertex2f(-1.0,-1.0); // V4

    glVertex2f( 0.5, 0.0); // V5
    glVertex2f( 0.5,-1.0); // V6

    glVertex2f( 1.5, 1.0); // V7
    glVertex2f( 1.5, 0.0); // V8
glEnd();
```



Slika 2.19. Crtanje trake četvorouglova

Poligoni

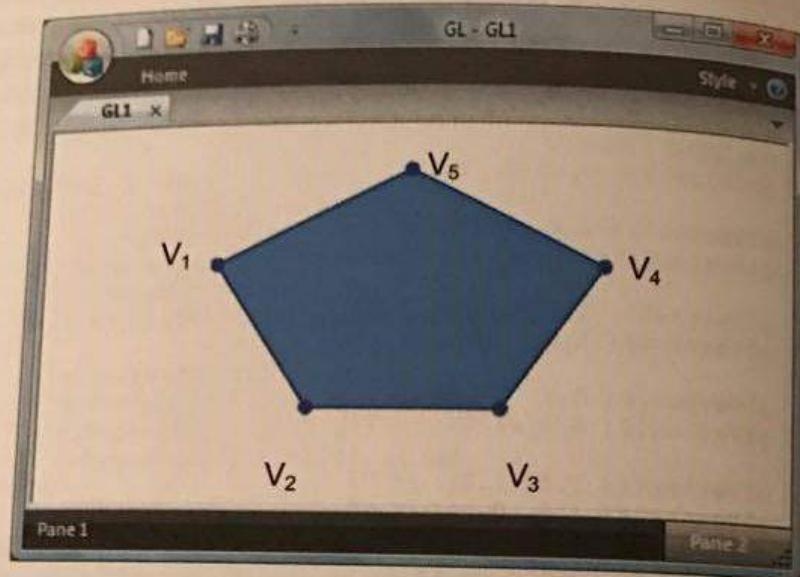
Poligoni se crtaju navođenjem najmanje tri temena nakon poziva funkcije **glBegin(GL_POLYGON)**. Ukoliko se navede manje od tri temena ništa se ne iscrtava.

```
glBegin(GL_POLYGON);
    glColor3f(0.1, 0.1, 0.5);
    glVertex2f(-2.0, 0.5); // v1
    glVertex2f(-1.0,-1.0); // v2
    glVertex2f( 1.0,-1.0); // v3
```

```

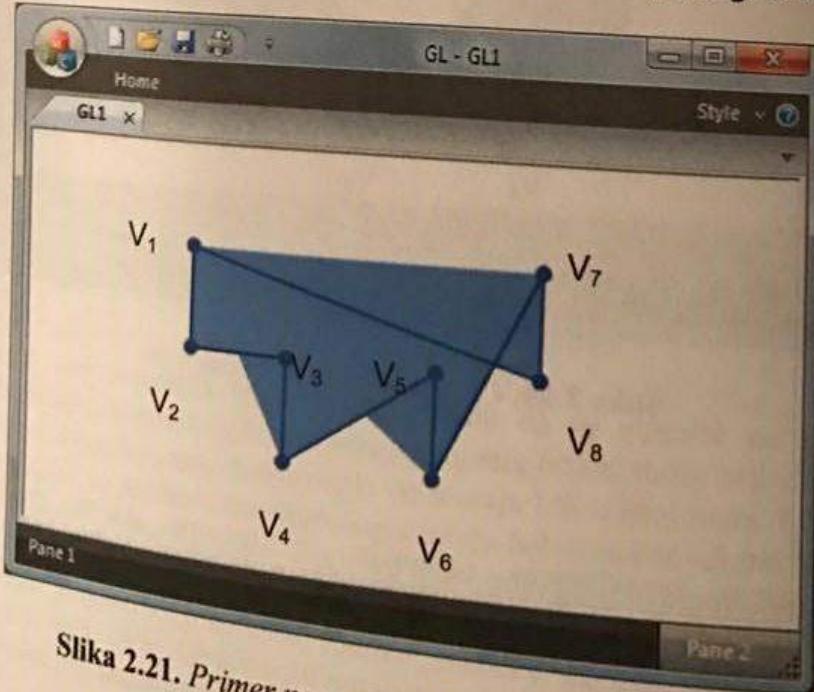
glVertex2f( 2.0, 0.5); // V4
glVertex2f( 0.0, 1.5); // V5
glEnd();

```



Slika 2.20. Crtanje poligona

Kao i kod četvorouglova i kod poligona važi ista priča o potrebi zadovoljenja: uslova konveksnosti, odsustvu uzajamnog presecanja ivica i potrebi da sve tačke poligona budu komplanarne. Štaviše, ovo postaju još ozbiljniji uslovi nego kod četvorouglova, obzirom da je broj tačaka veći. Da bi se uspešno prikazao, poligon se najpre mora prevesti u skup trouglova. Taj postupak deljenja složenog poligona (poligona koji nije konveksan, ima uzajamno presecajuće ivice, sadrži „rupe“ i sl.) naziva se teselacija (eng. *tessellation*).



Slika 2.21. Primer nepravilnog iscrtavanja poligona

Kombinovanje primitiva

Primitive, kao što i samo njihovo ime kaže, služe za izgradnju složenijih objekata. Kocka je primer jednostavnog objekta koji se sastoji od šest primitiva tipa GL_QUADS, tj. od šest kvadrata. Ako je stranica kocke dužine a , i želimo da je nacrtamo u koordinatnom početku, funkcija za iscrtavanje može da ima sledeći oblik:

```
void CGLRenderer::DrawCube(double a)
{
    glBegin(GL_QUADS);

    // Prednja stranica
    glVertex3d(-a/2, a/2, a/2);
    glVertex3d(-a/2, -a/2, a/2);
    glVertex3d( a/2, -a/2, a/2);
    glVertex3d( a/2, a/2, a/2);

    // Desna stranica
    glVertex3d( a/2, a/2, a/2);
    glVertex3d( a/2, -a/2, a/2);
    glVertex3d( a/2, -a/2, -a/2);
    glVertex3d( a/2, a/2, -a/2);

    // Zadnja stranica
    glVertex3d( a/2, a/2, -a/2);
    glVertex3d( a/2, -a/2, -a/2);
    glVertex3d(-a/2, -a/2, -a/2);
    glVertex3d(-a/2, a/2, -a/2);

    // Leva stranica
    glVertex3d(-a/2, a/2, -a/2);
    glVertex3d(-a/2, -a/2, -a/2);
    glVertex3d(-a/2, -a/2, a/2);
    glVertex3d(-a/2, a/2, a/2);

    // Gornja stranica
    glVertex3d(-a/2, a/2, a/2);
    glVertex3d( a/2, a/2, a/2);
    glVertex3d( a/2, a/2, -a/2);
    glVertex3d(-a/2, a/2, -a/2);

    // Donja stranica
    glVertex3d(-a/2, -a/2, -a/2);
    glVertex3d( a/2, -a/2, -a/2);
    glVertex3d( a/2, -a/2, a/2);
    glVertex3d(-a/2, -a/2, a/2);

    glEnd();
}
```

Uместо crtanja zasebnih kvadrata, proces iscrtavanja možemo ubrzati, uz istovremeno smanjenje broja linija koda potrebnih za opisivanje kocke, ukoliko četiri bočne stranice predstavimo GL_QUAD_STRIP primitivom, kao u sledećem primeru:

```
void CGLRenderer::DrawCube(double dSize)
{
    double a = dSize;
    glBegin(GL_QUAD_STRIP);
```

```

// Prednja leva vertikalna ivica
glVertex3d(-a/2, a/2, a/2);
glVertex3d(-a/2, -a/2, a/2);

// Prednja desna vertikalna ivica
glVertex3d( a/2, a/2, a/2);
glVertex3d( a/2, -a/2, a/2);

// Zadnja desna vertikalna ivica
glVertex3d( a/2, a/2, -a/2);
glVertex3d( a/2, -a/2, -a/2);

// Zadnja leva vertikalna ivica
glVertex3d(-a/2, a/2, -a/2);
glVertex3d(-a/2, -a/2, -a/2);

// Prednja leva vertikalna ivica
glVertex3d(-a/2, a/2, a/2);
glVertex3d(-a/2, -a/2, a/2);

glEnd();

glBegin(GL_QUADS);

    // Gornja stranica
    glVertex3d(-a/2, a/2, a/2);
    glVertex3d( a/2, a/2, a/2);
    glVertex3d( a/2, a/2, -a/2);
    glVertex3d(-a/2, a/2, -a/2);

    // Donja stranica
    glVertex3d(-a/2, -a/2, -a/2);
    glVertex3d( a/2, -a/2, -a/2);
    glVertex3d( a/2, -a/2, a/2);
    glVertex3d(-a/2, -a/2, a/2);

glEnd();
}

```

Efikasnost iscrtavanja se može značajno povećati, ako se umesto zadavanja pojedinačnih koriste polja temena.

Polja temena

Polja temena (eng. *vertex arrays*) omogućavaju bolje performanse programa smanjujući broj poziva funkcija i efikasnije prebacivanje podataka iz operativne memorije na grafičku karticu. Pogledajmo prethodni primer iscrtavanja kocke. Funkcija **DrawCube()** sadrži 22 poziva različitih OpenGL funkcija, od kojih svaka prenosi od jednog do tri parametra.

Osnovna ideja rada sa poljima temena jeste da se sva temena smeste u jedno ili više polja,² kada se javi potreba za iscrtavanjem da se jednim pozivom (ili pomoću više poziva) odgovarajuće funkcije aktivira brzi prenos bloka podataka (na primer korišćenjem DMA) iz operativne memorije računara u memoriju grafičke kartice.

OPENGL
Postupak
Korak 1:
Prostori
boju, ne
potzeline
čuvaćem
istovrem
Korak 2:
podaci,
gl*Point
primer, i

Prvi par
ili 4. I
GL_DOU
= 0, to z
istog po
atributa,
tipovi at
atributa
koordinat
(bajtova)
koordinata

Za svaki
podataka

Parametr
koristi o
ukoliko s
Sve na
glNorma
normale
na navod

Korak 3:
aktivirati

ide se i
koordinata

Postupak iscrtavanja odvija se u pet koraka:

Korak 1: Popunjavanje polja podacima. Već znamo da teme može imati više atributa. Prostorne koordinate temena su obavezni atributi, ali osim njih svako teme može imati: boju, normalu, teksturne koordinate, itd. Ukoliko se ovi atributi mogu nezavisno menjati, poželjno je da za svaki od tipova atributa imamo zasebno polje. Na primer, u jednom polju čuvaćemo samo prostorne koordinate, u drugom boje, itd. Ukoliko se svi atributi menjaju istovremeno, onda je mnogo efikasnije sve pomešati u jedno polje.

Korak 2: Definisanje pokazivača na polja. Da bi funkcije za crtanje znale gde se nalaze podaci, kako su organizovani i kako im pristupiti, potrebno je pozvati odgovarajući `gl*Pointer()` funkciju. Ovih funkcija ima onoliko koliko i tipova atributa temena. Na primer, za postavljanje pokazivača na prostorne koordinate temena koristi se funkcija:

```
void glVertexPointer(GLint size, GLenum type,
                     GLsizei stride, const GLvoid *pointer);
```

Prvi parametar `size` definiše broj koordinata po jednom temenu i može imati vrednosti: 2, 3 ili 4. Parametar `type` definiše tip podataka (`GL_SHORT`, `GL_INT`, `GL_FLOAT`, ili `GL_DOUBLE`), a `stride` rastojanje u bajtovima između dva sucesivna temena. Ako je `stride` = 0, to znači da su temena gusto pakovana, tj. da nema mešanja tipova atributa u okviru istog polja. Na primer, ako temena imaju tri koordinate tipa `float`, i nema mešanja tipova atributa, potpuno isti efekat imaće i `stride` = $3 * \text{sizeof}(float)$ i `stride` = 0. Kada se mešaju tipovi atributa u okviru istog polja `stride` mora imati vrednost jednaku ukupnoj veličini svih atributa samo jednog temena. Na primer, ako se boja prestavlja sa 3 podatka tipa `float` a koordinate sa 2 podatka tipa `float`, `stride` treba postaviti na $(3+2) * \text{sizeof}(float)$, tj. na 20 (bajtova). Poslednji parametar (`pointer`) je pokazivač na prvi podatak u polju koji sadrži koordinate temena.

Za svaki od tipova atributa postoji odgovarajuća funkcija za definisanje organizacije podataka i postavljanje pokazivača na njih. Za boje se koristi funkcija:

```
void glColorPointer(GLint size, GLenum type,
                    GLsizei stride, const GLvoid *pointer);
```

Parametri funkcije imaju isto značenje kao kod funkcije `glVertexPointer()`. Ukoliko se koristi osvetljenje, potrebno je definisati polje normala funkcijom `glNormalPointer()`, a ukoliko se koriste teksture, polje teksturnih koordinata, funkcijom `glTexCoordPointer()`. Sve navedene `gl*Pointer()` funkcije imaju iste parametre, osim funkcije `glNormalPointer()`, kojoj nedostaje parametar `size`. On je izostavljen obzirom da su normale vektori i moraju imati tačno tri komponente (koordinate), pa se to podrazumeva i na navodi se u listi parametara.

Korak 3: Aktiviranje polja. Da bi mogla da se koriste polja temena potrebno ih je aktivirati. To se postiže pozivom funkcije:

```
void glEnableClientState(GLenum array);
```

gde se kao parametar navodi: `GL_VERTEX_ARRAY` – ako želimo aktivirati polje koordinata, `GL_COLOR_ARRAY` – ako želimo aktivirati boje, itd.

Funkcije za crtanje koristiće sva aktivirana polja. Treba biti obazriv, jer ukoliko ostane aktivno neko polje koje nema validan pokazivač i nije pozvana odgovarajuća funkcija `gl*Pointer()` koja ga pravilno opisuje, to dovodi do sigurnog rušenja aplikacije.

Korak 4: Crtanje geometrije. Tek u ovom trenutku se polja temena prenose na serversku stranu, tj. na grafičku karticu. Crtanje se može ostvariti:

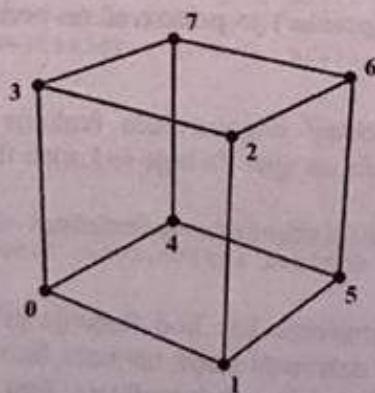
- proizvoljnim „skakanjem“ po polju i pristupom pojedinačnim elementima, pozivom funkcije `glArrayElement()`,
- sekvensijalnim pristupom elementima u polju, npr. pozivom funkcije `glDrawArrays()`, ili
- metodičkim pristupom elementima polja korišćenjem posebnog indeksnog bafera, npr. pozivom funkcije `glDrawElements()`.

Korak 5: Deaktiviranje polja. Ostvaruje se pozivima funkcije:

```
void glDisableClientState(GLenum array);
```

sa parametrima istim kao u koraku 3. Ovo je vrlo važan korak, jer ukoliko ostane aktivno neko od polja, a to nije željeno, sledeća funkcija za crtanje na osnovu polja temena može da izazove rušenje aplikacije.

Da bismo lakše shvatili funkcionisanje polja temena, prikazaćemo nekoliko primera crtanja kocke. Temena će biti smeštena u polja redosledom koji je prikazan na slici 2.22.



Slika 2.22. Redosled zadavanja temena kocke

Prvo ćemo videti kako se iscrtavanje vrši pomoću funkcije:

```
void glArrayElement(GLint i);
```

Ova funkcija prikuplja attribute na *i*-toj poziciji iz svih aktiviranih polja. Navodi se između `glBegin()` i `glEnd()` poziva, i njeno dejstvo ekvivalentno je pozivima funkcija `glColor*()`/`glNormal*()`/`glTexCoord*()`/`glVertex*()`.

```
// GLRenderer.h -----
float vert[24];
float col[24];
BYTE ind[24];
void PrepareVACube(float a);
void DrawVACube();

// GLRenderer.cpp -----
void CGLRenderer::PrepareScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClearColor (1.0, 1.0, 1.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    glLineWidth(2.0);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    PrepareVACube(2.0);
    //-----
    wglMakeCurrent(NULL, NULL);
}

void CGLRenderer::DrawScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity ();

    // Transformacija pogleda
    glTranslatef(0.0, 0.0, -10.0);
    glRotatef(20.0, 1.0, 0.0, 0.0);
    glRotatef(-30.0, 0.0, 1.0, 0.0);

    glColor3f(0.0,0.0,0.0);
    DrawVACube();
    //-----
    glFlush();
    SwapBuffers(pDC->m_hDC);
    wglMakeCurrent(NULL, NULL);
}

void CGLRenderer::PrepareVACube(float a)
{
    vert[0] = -a/2; vert[1] = -a/2; vert[2] = a/2; // vert0
    vert[3] = a/2; vert[4] = -a/2; vert[5] = a/2; // vert1
    vert[6] = a/2; vert[7] = a/2; vert[8] = a/2; // vert2
    vert[9] = -a/2; vert[10] = a/2; vert[11] = a/2; // vert3
    vert[12] = -a/2; vert[13] = -a/2; vert[14] = -a/2; // vert4
    vert[15] = a/2; vert[16] = -a/2; vert[17] = -a/2; // vert5
    vert[18] = a/2; vert[19] = a/2; vert[20] = -a/2; // vert6
    vert[21] = -a/2; vert[22] = a/2; vert[23] = -a/2; // vert7

    void CGLRenderer::DrawVACube()
    {
        glVertexPointer(3, GL_FLOAT, 0, vert);
        glEnableClientState(GL_VERTEX_ARRAY);
```

```

glBegin(GL_QUADS);

    glArrayElement(0);
    glArrayElement(1);
    glArrayElement(2);
    glArrayElement(3);

    glArrayElement(1);
    glArrayElement(5);
    glArrayElement(6);
    glArrayElement(2);

    glArrayElement(7);
    glArrayElement(6);
    glArrayElement(5);
    glArrayElement(4);

    glArrayElement(0);
    glArrayElement(3);
    glArrayElement(7);
    glArrayElement(4);

    glArrayElement(7);
    glArrayElement(3);
    glArrayElement(2);
    glArrayElement(6);

    glArrayElement(0);
    glArrayElement(4);
    glArrayElement(5);
    glArrayElement(1);

    glEnd();
    glDisableClientState(GL_VERTEX_ARRAY);
}

```

Ukoliko imamo više atributa temena smeštenih u zasebna polja, potrebno je aktivirati svako od datih polja. Na primer, ako za temena sem prostornih koordinata imamo i boju, funkcije **PrepareVACube()** i **DrawVACube()** izgledaju ovako:

```

void CGLRenderer::PrepareVACube(float a)
{
    vert[0] = -a/2; vert[1] = -a/2; vert[2] = a/2; // vert0
    vert[3] = a/2; vert[4] = -a/2; vert[5] = a/2; // vert1
    vert[6] = a/2; vert[7] = a/2; vert[8] = a/2; // vert2
    vert[9] = -a/2; vert[10] = a/2; vert[11] = a/2; // vert3
    vert[12] = -a/2; vert[13] = -a/2; vert[14] = -a/2; // vert4
    vert[15] = a/2; vert[16] = -a/2; vert[17] = -a/2; // vert5
    vert[18] = a/2; vert[19] = a/2; vert[20] = -a/2; // vert6
    vert[21] = -a/2; vert[22] = a/2; vert[23] = -a/2; // vert7

    col[0] = 0.0; col[1] = 0.0; col[2] = 0.0; // col0
    col[3] = 1.0; col[4] = 0.0; col[5] = 0.0; // col1
    col[6] = 1.0; col[7] = 1.0; col[8] = 0.0; // col2
    col[9] = 1.0; col[10] = 1.0; col[11] = 1.0; // col3
    col[12] = 0.0; col[13] = 1.0; col[14] = 0.0; // col4
    col[15] = 0.0; col[16] = 1.0; col[17] = 1.0; // col5
    col[18] = 0.0; col[19] = 0.0; col[20] = 1.0; // col6
    col[21] = 0.0; col[22] = 0.0; col[23] = 0.0; // col7
}

```

Broj pozvani
sekvensijalni
svako teme P
crtanje objekta

Parametar n
GL_TRIANGLES
od indeksa f

VOL

```

void CGLRenderer::DrawVACube()
{
    glVertexPointer(3, GL_FLOAT, 0, vert);
    glColorPointer(3, GL_FLOAT, 0, col);
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_COLOR_ARRAY);

    glBegin(GL_QUADS);
    glArrayElement(0);
    glArrayElement(1);
    glArrayElement(2);
    glArrayElement(3);

    glArrayElement(1);
    glArrayElement(5);
    glArrayElement(6);
    glArrayElement(2);

    glArrayElement(7);
    glArrayElement(6);
    glArrayElement(5);
    glArrayElement(4);

    glArrayElement(0);
    glArrayElement(3);
    glArrayElement(7);
    glArrayElement(4);

    glArrayElement(7);
    glArrayElement(3);
    glArrayElement(2);
    glArrayElement(6);

    glArrayElement(0);
    glArrayElement(4);
    glArrayElement(5);
    glArrayElement(1);

    glEnd();
    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_COLOR_ARRAY);
}

```

Broj pozvanih funkcija možemo smanjiti, ukoliko multipliciramo temena, tako da se sekvenčijalnim obilaskom polja iscrtava čitav objekat. U slučaju kocke to znači da će se svako teme ponoviti tačno tri puta (obzirom da učestvuje u formiranju tri stranice). Za crtanje objekta koristimo funkciju:

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

Parametar *mode* definiše koji tip primitiva se formira (GL_POINTS, GL_LINES, GL_TRIANGLES, itd.), a primitive se formiraju od elemenata svih aktivnih polja, počev od indeksa *first*, a zaključno sa indeksom *first+count-1*.

```

void CGLRenderer::PrepareVACube(float a)
{
    vert[0] = -a/2; vert[1] = -a/2; vert[2] = a/2; // vert0
    vert[3] = a/2; vert[4] = -a/2; vert[5] = a/2; // vert1
    vert[6] = a/2; vert[7] = a/2; vert[8] = a/2; // vert2
}

```

```

vert[9] = -a/2; vert[10] = a/2; vert[11] = a/2; // vert3
vert[12] = a/2; vert[13] = -a/2; vert[14] = a/2; // vert1
vert[15] = a/2; vert[16] = -a/2; vert[17] = -a/2; // vert5
vert[18] = a/2; vert[19] = a/2; vert[20] = -a/2; // vert6
vert[21] = a/2; vert[22] = a/2; vert[23] = a/2; // vert2
vert[24] = -a/2; vert[25] = a/2; vert[26] = -a/2; // vert7
vert[27] = a/2; vert[28] = a/2; vert[29] = -a/2; // vert0
vert[30] = a/2; vert[31] = -a/2; vert[32] = -a/2; // vert3
vert[33] = -a/2; vert[34] = -a/2; vert[35] = -a/2; // vert4
vert[36] = -a/2; vert[37] = -a/2; vert[38] = a/2; // vert0
vert[39] = -a/2; vert[40] = a/2; vert[41] = a/2; // vert3
vert[42] = -a/2; vert[43] = a/2; vert[44] = -a/2; // vert7
vert[45] = -a/2; vert[46] = -a/2; vert[47] = -a/2; // vert4
vert[48] = -a/2; vert[49] = a/2; vert[50] = -a/2; // vert7
vert[51] = -a/2; vert[52] = a/2; vert[53] = a/2; // vert3
vert[54] = a/2; vert[55] = a/2; vert[56] = a/2; // vert2
vert[57] = a/2; vert[58] = a/2; vert[59] = -a/2; // vert6
vert[60] = -a/2; vert[61] = -a/2; vert[62] = a/2; // vert0
vert[63] = -a/2; vert[64] = -a/2; vert[65] = -a/2; // vert4
vert[66] = a/2; vert[67] = -a/2; vert[68] = -a/2; // vert5
vert[69] = a/2; vert[70] = -a/2; vert[71] = a/2; // vert1
}

void CGLRenderer::DrawVACube()
{
    glVertexPointer(3, GL_FLOAT, 0, vert);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_QUADS, 0, 24);
    glDisableClientState(GL_VERTEX_ARRAY);
}

```

Zbog količine podataka potrebnih da budu definisani, prethodni primer sadrži samo polje prostornih koordinata temena. Ukoliko želimo i boju, dodali bismo još jedno polje od 72 elemenata, sa odgovarajućim bojama. Sa stanovišta broja poziva funkcija `glDrawArrays()` u prethodnom slučaju je mnogo bolje rešenje za crtanje kocke, jer zahteva samo jedan poziv (umesto 24), ali je sa stanovišta količine podataka koje je potrebno preneti i dalje neefiksnja, jer zahteva više podataka nego što je neophodno.

Da bi sprečili potrebu ponavljanja podataka, uvešćemo još jedno polje, koje će služiti za indeksiranje. Umesto da ponavljamo definicije temena, u polju za indeksiranje navešćemo samo indekse temena iz polja sa podacima u redosledu koji je potreban za iscrtavanje. Za samo iscrtavanje koristimo funkciju:

```

void glDrawElements(GLenum mode, GLsizei count, GLenum type,
                    const GLvoid *indices);

```

Iscrtavanje se vrši na osnovu polja indeksa, na koje ukazuje pokazivač `indices`. Broj elemenata u polju indeksa definisan je parametrom `count`, a njihov tip parametrom `type`. Kao i u prethodnom slučaju, `mode` određuje tip primitive koja se formira.

```

void CGLRenderer::PrepareVACube(float a)
{
    vert[0] = -a/2; vert[1] = -a/2; vert[2] = a/2; // vert0
    vert[3] = a/2; vert[4] = -a/2; vert[5] = a/2; // vert1
    vert[6] = a/2; vert[7] = a/2; vert[8] = a/2; // vert2
    vert[9] = -a/2; vert[10] = a/2; vert[11] = a/2; // vert3
    vert[12] = -a/2; vert[13] = -a/2; vert[14] = -a/2; // vert4
    vert[15] = a/2; vert[16] = -a/2; vert[17] = -a/2; // vert5
    vert[18] = a/2; vert[19] = a/2; vert[20] = -a/2; // vert6
    vert[21] = -a/2; vert[22] = a/2; vert[23] = -a/2; // vert7

    col[0] = 0.0; col[1] = 0.0; col[2] = 0.0; // col0
    col[3] = 1.0; col[4] = 0.0; col[5] = 0.0; // col1
    col[6] = 1.0; col[7] = 1.0; col[8] = 0.0; // col2
    col[9] = 1.0; col[10] = 1.0; col[11] = 1.0; // col3
    col[12] = 0.0; col[13] = 1.0; col[14] = 0.0; // col4
    col[15] = 0.0; col[16] = 1.0; col[17] = 1.0; // col5
    col[18] = 0.0; col[19] = 0.0; col[20] = 1.0; // col6
    col[21] = 0.0; col[22] = 0.0; col[23] = 0.0; // col7

    // Indeks
    ind[0] = 0; ind[1] = 1; ind[2] = 2; ind[3] = 3; // quad0
    ind[4] = 1; ind[5] = 5; ind[6] = 6; ind[7] = 2; // quad1
    ind[8] = 7; ind[9] = 6; ind[10] = 5; ind[11] = 4; // quad2
    ind[12] = 0; ind[13] = 3; ind[14] = 7; ind[15] = 4; // quad3
    ind[16] = 7; ind[17] = 3; ind[18] = 2; ind[19] = 6; // quad4
    ind[20] = 0; ind[21] = 4; ind[22] = 5; ind[23] = 1; // quad5
}

void CGLRenderer::DrawVACube()
{
    glVertexPointer(3, GL_FLOAT, 0, vert);
    glColorPointer(3, GL_FLOAT, 0, col);
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_COLOR_ARRAY);

    glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, ind);

    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_COLOR_ARRAY);
}

```

Moguće je i sve atribute temena smestiti u jedno polje. Onda je u funkcijama za definisanje pokazivača na podatke potrebno navesti i preskok koji se javlja kod pristupa sledećoj grupi podataka istog tipa. Da se podsetimo, treći parametar u funkcijama `gl*Pointer()` definiše rastojanje između istorodnih podataka. Ukoliko pomešamo boje i prostorne koordinate temena u okviru jednog polja, tako da, na primer, prvo navedemo boju prvog temena, zatim koordinate prvog temena, zatim boju drugog temena, pa koordinate drugog temena i tako dalje, tada je rastojanje između dve boje jednak zbiru veličine definicije boje i prostornih koordinata jednog temena. Obzirom da su i boja i prostorne koordinate definisane sa po 3 komponente tipa `float`, preskok treba da bude jednak $(3+3)*\text{sizeof}(\text{float})$, odnosno 24 bajta.

Četvrti parametar funkcija `gl*Pointer()` je pokazivač na početak odgovarajućeg vektora podataka. Ukoliko prvo smeštamo boje, početak „podvektora“ boja se poklapa sa početkom kombinovanog vektora, dok je početak „podvektora“ koordinata pomeren za veličinu jedne boje. Obzirom da se boja sastoji od 3 podatka tipa `float`, a indeksiranje počinje od nule,

adresa „podvektora“ koordinata se poklapa sa adresom trećeg elementa kombinovanog vektora. Sve ovo će biti mnogo jasnije ako pogledamo kako to izgleda na primeru.

```
void CGLRenderer::PrepareVACube(float a)
{
    // Zajedничko polje
    buf[0] = 0.0; buf[1] = 0.0; buf[2] = 0.0; // col0
    buf[3] = -a/2; buf[4] = -a/2; buf[5] = a/2; // ver0
    buf[6] = 1.0; buf[7] = 0.0; buf[8] = 0.0; // col1
    buf[9] = a/2; buf[10] = -a/2; buf[11] = a/2; // ver1
    buf[12] = 1.0; buf[13] = 1.0; buf[14] = 0.0; // col2
    buf[15] = a/2; buf[16] = a/2; buf[17] = a/2; // ver2
    buf[18] = 1.0; buf[19] = 1.0; buf[20] = 1.0; // col3
    buf[21] = -a/2; buf[22] = a/2; buf[23] = a/2; // ver3
    buf[24] = 0.0; buf[25] = 1.0; buf[26] = 0.0; // col4
    buf[27] = -a/2; buf[28] = -a/2; buf[29] = -a/2; // ver4
    buf[30] = 0.0; buf[31] = 1.0; buf[32] = 1.0; // col5
    buf[33] = a/2; buf[34] = -a/2; buf[35] = -a/2; // ver5
    buf[36] = 0.0; buf[37] = 0.0; buf[38] = 1.0; // col6
    buf[39] = a/2; buf[40] = a/2; buf[41] = -a/2; // ver6
    buf[42] = 0.0; buf[43] = 0.0; buf[44] = 0.0; // col7
    buf[45] = -a/2; buf[46] = a/2; buf[47] = -a/2; // ver7

    // Indeksi
    ind[0] = 0; ind[1] = 1; ind[2] = 2; ind[3] = 3; // quad0
    ind[4] = 1; ind[5] = 5; ind[6] = 6; ind[7] = 2; // quad1
    ind[8] = 7; ind[9] = 6; ind[10] = 5; ind[11] = 4; // quad2
    ind[12] = 0; ind[13] = 3; ind[14] = 7; ind[15] = 4; // quad3
    ind[16] = 7; ind[17] = 3; ind[18] = 2; ind[19] = 6; // quad4
    ind[20] = 0; ind[21] = 4; ind[22] = 5; ind[23] = 1; // quad5

    void CGLRenderer::DrawVACube()
    {
        glVertexPointer(3, GL_FLOAT, 6*sizeof(float), &buf[3]);
        glColorPointer(3, GL_FLOAT, 6*sizeof(float), &buf[0]);
        glEnableClientState(GL_VERTEX_ARRAY);
        glEnableClientState(GL_COLOR_ARRAY);
        glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, ind);
        glDisableClientState(GL_VERTEX_ARRAY);
        glDisableClientState(GL_COLOR_ARRAY);
    }
}
```

Postoji još mnoštvo funkcija za rad sa poljima temena. Neke od njih omogućuju kreiranje i manipulaciju baferima koji se nalaze u memoriji grafičke kartice, čime se postižu daleko veće brzine iscrtavanja, jer nema prenosa podataka iz operativne memorije ka grafičkoj kartici pri svakom pozivu funkcije za crtanje. Rad sa baferovanim poljima temena (*Vertex Buffer Object – VBO*) uveden je u OpenGL-u 1.5. U ovom poglavlju prikazane su samo funkcije podržane u svim verzijama OpenGL-a.

Izbor načina senčenja

OpenGL podržava dva načina senčenja:

- jednobojno (*flat*) senčenje i
- umekšano (*smooth*) senčenje,

a definše se pozivom funkcije:

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

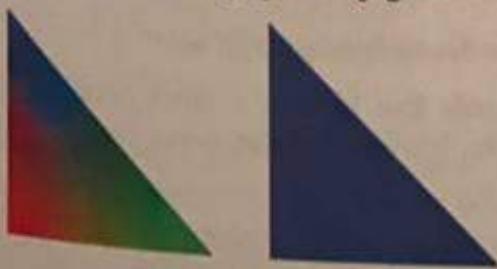
Ako se kao parametar ovoj funkciji prosledi konstanta GL_SMOOTH, svako teme primitive može imati drugačiju boju. Boja piksela koji ispunjuju unutrašnjost primitive biće određena linearnom interpolacijom boje temena. Ovo je podrazumevani način senčenja.

Jednobojno senčenje postiže se prosleđivanjem parametra GL_FLAT funkciji glShadeModel(). Kod ovog senčenja čitava površina primitive ispunjena je uniformnom bojom, i to bojom koja je bila aktivna u trenutku kada je zadavano poslednje teme primitive.

Razmotrimo naredni programski kod. Njime se crta pravougli trougao, čije je svako teme drugačje boje. Boja koja je zatećena kao „tekuća“ u trenutku kada se zadaju prostorne koordinate temena biće boja tog temena. U ovom primeru, teme sa koordinatama (0.0, -1.0, 0.0) je crvene boje, teme (2.0, -1.0, 0.0) je zeleno, a (0.0, 1.0, 0.0) je plavo. Pikseli u unutrašnjosti trougla imaju boju koja se dobija mešanjem boja temena u zavisnosti od rastojanja konkretnog piksela od temena trougla. I da nije navedeno glShadeModel(GL_SMOOTH), senčenje bi bilo identično. Rezultat senčenja prikazan je na slici 2.23a.

```
glShadeModel(GL_SMOOTH);
glBegin(GL_TRIANGLES);
    glColor3f (1.0, 0.0, 0.0);
    glVertex3f (0.0,-1.0, 0.0);
    glColor3f (0.0, 1.0, 0.0);
    glVertex3f (2.0,-1.0, 0.0);
    glColor3f (0.0, 0.0, 1.0);
    glVertex3f (0.0, 1.0, 0.0);
glEnd();
```

Ukoliko se umesto glShadeModel(GL_SMOOTH) navede glShadeModel(GL_FLAT), čitav trougao biće obojen u plavu boju, kao što je prikazano na slici 2.23b. Sve boje osim boje koja je navedena neposredno pre definicije poslednjeg temena biće ignorisane.



Slika 2.23. Dva načina senčenja: a) umekšano, b) jednobojno

Kod primitiva koje dele zajednička temena, kao što su: trake trouglova (GL_TRIANGLE_STRIP), lepeze trouglova (GL_TRIANGLE_FAN) i trake četvorouglova (GL_QUAD_STRIP), treba voditi računa da osim prostornih koordinata, primitivi dele i ostale attribute temena. Dakle, dele i boju. Zato je, na primer, nemoguće koristiti umekšane senčenja i traku trouglova, a da svaki trougao ima drugačiju boju.

Kratak pregled gradiva

Da ukratko ponovimo najvažnije:

- svi objekti sastavljeni su od grafičkih primitiva,
- grafičke primitive zadaju se temenima,
- u OpenGL-u prostorne koordinate definišu se funkcijom
 - o **glVertex***
- koja se navodi izmedju poziva funkcija **glBegin()** i **glEnd()**,
- tip primitive definiše se navođenjem parametra funkcije **glBegin()** i može biti:
 - o GL_POINTS – pojedinačne tačke,
 - o GL_LINES – pojedinačne duži (linije),
 - o GL_LINE_STRIP – serija povezanih duži,
 - o GL_LINE_LOOP – petlja,
 - o GL_TRIANGLES – pojedinačni trouglovi,
 - o GL_TRIANGLE_STRIP – traka trouglova,
 - o GL_TRIANGLE_FAN – lepeza trouglova,
 - o GL_QUADS – pojedinačni četvorouglovi,
 - o GL_QUAD_STRIP – traka četvorouglova i
 - o GL_POLYGON – poligon
- svi parametri figure vezani su za temena i moraju se postaviti pre definisanja prostornih koordinata datog temena,
- boja temena postavlja se funkcijom **glColor***,
- OpenGL funkcioniše kao konačni automat (jednom postavljena vrednost ostaje dok se ne promeni, i važi za sve primitive koje se iscrtavaju nakon postavljanja vrednosti),
- postavljene vrednosti pojedinih parametara mogu se pročitati korišćenjem funkcija **glGetBooleanv()**, **glGetDoublev()**, **glGetFloatv()** i **glGetIntegerv()**.

Funkcije korišćene u ovoj glavi

```
void glColor3{ubsid}(TYPE red, TYPE green, TYPE blue);
void glColor4{ubsid}(TYPE red, TYPE green, TYPE blue, TYPE alpha);
void glColor{34}{ubsid}v( const TYPE *v );
```

red – intenzitet crvene komponente

green – intenzitet zelene komponente

blue – intenzitet plave komponente

alpha – intenzitet *alpha* komponente (providnost)

v – pokazivač na polje koje sadrži RGB(A) komponente boje

Postavlja tekuću boju. Primjenjuje se na sve primitive koje se iscrtavaju nakon poziva ove funkcije.

```
void glVertex2{dfis}( TYPE x, TYPE y );
void glVertex3{dfis}( TYPE x, TYPE y, TYPE z );
void glVertex4{dfis}( TYPE x, TYPE y, TYPE z, TYPE w );
void glVertex{234}{dfis}v( const TYPE *v );
```

x, y, z, w – koordinate

v – pokazivač na polje sa koordinatama

Definiše prostorne koordinate temena.

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

mode – tip primitive koja se kreira na osnovu definicije prostornih koordinata.

Može imati jednu od sledećih vrednosti:

GL_POINTS – pojedinačne tačke

GL_LINES – pojedinačne duži

GL_LINE_STRIP – serija međusobno povezanih duži

GL_LINE_LOOP – petlja

GL_TRIANGLES – pojedinačni trouglovi

GL_TRIANGLE_STRIP – traka trouglova

GL_TRIANGLE_FAN – lepeza trouglova

GL_QUADS – pojedinačni četvorouglovi

GL_QUAD_STRIP – traka četvorouglova

GL_POLYGON – poligon

Ove dve funkcije uvek idu u paru. Između njih navode se definicije temena. Tip primitive koji se kreira zavisi od parametra prosledjenog funkciji `glBegin()`.

void glPointSize(GLfloat size);

size – veličina tačke

Definiše veličinu tačke (primitive GL_POINT) u pikselima.

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

pname – ime parametra koji je potrebno očitati. U ovom poglavlju korišćeni su sledeći parametri:

GL_POINT_SIZE – veličina tačke (GL_POINT) u pikselima,
 GL_POINT_SIZE_GRANULARITY – rezolucija sa kojom se može menjati veličina tačke,
 GL_POINT_SIZE_RANGE – opseg u kome se može menjati veličina tačke,
 GL_LINE_WIDTH – debljina linije u pikselima,
 GL_LINE_WIDTH_GRANULARITY – rezolucija sa kojom se može menjati debljina linije,
 GL_LINE_WIDTH_RANGE – opseg u kome se može menjati debljina linije,
 GL_LINE_STIPPLE_PATTERN – tekući šablon linije,
 GL_LINE_STIPPLE_REPEAT – faktor množenja (ponavljanja) šablonu,
 GL_LINE_STIPPLE – indikator da li je uključen šablon

params – očitana vrednost (ili vrednosti).

Navedene funkcije služe za očitavanje različitih parametara i stanja u kome se nalazi OpenGL *rendering context*. U OpenGL-u 1.1 mogu se na ovaj način očitati preko 200 različitih vrednosti.

void glLineWidth(GLfloat width);

width – debljina linije u pikselima

Postavlja debljinu linije na zadatu vrednost, pod uslovom da se nalazi u granicama koje su određene parametrom GL_LINE_WIDTH_RANGE i sa korakom GL_LINE_WIDTH_GRANULARITY.

void glLineStipple(GLint factor, GLushort pattern);

factor – faktor kojim se množi šablon,
pattern – sam šablon.

Definiše šablon linije.

void glEnable(GLenum cap) / void glDisable(GLenum cap)

cap – stanje koje se uključuje/isključuje. Može imati preko 40 različitih vrednosti, ne računajući podoblike. U ovom poglavlju koriste se samo:
GL_LINE_STIPPLE – uključuje/isključuje šablon za linije,
GL_CULL_FACE – uključuje/isključuje odbacivanje poligona koji se vide sa prednje ili zadnje strane.

Ovaj par funkcija uključuje/isključuje odgovarajuće stanje u OpenGL *rendering context-u*.

void glPolygonMode(GLenum face, GLenum mode);

face – strana poligona za koju važi dati način iscrtavanja. Može imati jednu od sledećih vrednosti:

GL_FRONT – prednja,
GL_BACK – zadnja,
GL_FRONT_AND_BACK – i prednja i zadnja.

mode – način iscrtavanja. Može imati jednu od sledećih vrednosti:

GL_POINT – tačke,
GL_LINE – linije,
GL_FILL – ispuna.

Definiše način iscrtavanja trouglova, četvorouglova i poligona. Moguće je iscrtavati samo tačke u temenima (GL_POINT), linije na ivicama (GL_LINE) ili spuniti čitavu površinu (GL_FILL). Podrazumevano iscrtavanje je GL_FILL.

void glFrontFace(GLenum mode)

mode – redosled zadavanja temena. Može imati jednu od sledeće dve vrednosti:

GL_CCW – prednja strana postaje ona koja ima orijentaciju suprotnu od kretanja kazaljki na časovniku (ovo je podrazumevano stanje),
GL_CW – prednja strana postaje ona koja ima orijentaciju u pravcu kretanja kazaljki na časovniku.

Definiše pozitivnu orijentaciju, tj. orijentaciju strane trougla, četvorouglja ili poligona koja se smatra prednjom.

void glCullFace(GLenum mode);

mode – definiše da li će poligoni koji se vide sa prednje ili zadnje strane biti odbačeni prilikom iscrtavanja. Može imati jednu od sledećih vrednosti:

GL_FRONT – odbacuju se poligoni koji se vide sa prednje strane,

GL_BACK – odbacuju se poligoni koji se vide sa zadnje strane,

GL_FRONT_AND_BACK – poligoni se odbacuju bez obzira sa koje se strane vide.

Definiše da li će poligon biti iscrtan zavisno od stane koja je vidljiva.

void glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);

size – broj koordinata po jednom temenu (2, 3 ili 4),

type – tip podataka u polju (GL_SHORT, GL_INT, GL_FLOAT, ili GL_DOUBLE),

stride – rastojanje u bajtovima između dva sucesivna temena,

pointer – pokazivač na polje prostornih koordinata temena.

Definiše memoriju organizaciju polja prostornih koordinata temena i pokazivač na dato polje.

void glColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer);

size – broj koordinata po jednom temenu (3 ili 4),

type – tip podataka u polju (GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, ili GL_DOUBLE),

stride – rastojanje u bajtovima između dva sucesivna temena,

pointer – pokazivač na polje boja temena.

Definiše memoriju organizaciju polja boja temena i pokazivač na dato polje.

void glEnableClientState(GLenum array);**void glDisableClientState(GLenum array) ;**

array – definiše polje atributa temena koje se aktivira/deaktivira. U ovom poglavlju korišćene su sledeće vrednosti:

GL_VERTEX_ARRAY – polje prostornih koordinata temena,

GL_COLOR_ARRAY – polje boja temena,

Aktivira/deaktivira odgovarajuće polje atributa temena.

void glDrawElements(GLint i);

i – indeks temena, u aktiviranim poljima temena, koje se iscrtava

Javlja se između poziva funkcija `glBegin()` i `glEnd()` i služi za iscrtavanje primitiva korišćenjem polja temena, proizvoljnim „skakanjem“ po polju i pristupom pojedinačnim elementima. Koriste se atributi iz svih aktiviranih polja, a tip primitive koja se formira zavisi od parametra funkcije `glBegin()`.

void glDrawArrays(GLenum mode, GLint first, GLsizei count);

mode – tip primitive koja se formira (GL_POINTS, GL_LINES, itd.)
first – indeks prvog temena kome se pristupa u svim aktivnim poljima
count – broj temena koja se koriste za formiranje primitiva

Iscrtava primitive definisane parametrom *mode*, sekvensijalnim pristupom elementima svih aktivnih polja temena i njihovih atributa, počev od indeksa *first*, a zaključno sa indeksom *first + count - 1*.

**void glDrawElements(GLenum mode, GLsizei count, GLenum type,
const GLvoid *indices);**

mode – tip primitive koja se formira (GL_POINTS, GL_LINES, itd.)
count – broj elemenata u polju indeksa
type – tip podataka smeštenih u polju indeksa
indices – pokazivač na polje indeksa

Iscrtava primitive definisane parametrom *mode*, pristupom elementima svih aktivnih polja temena i njihovih atributa na osnovu polja indeksa, na koje ukazuje pokazivač *indices*. Broj elemenata u polju indeksa definisan je parametrom *count*, a njihov tip parametrom *type*.

void glShadeModel(GLenum mode)

mode – način senčenja. Može imati jednu od sledećih vrednosti:

GL_FLAT – jednobojno senčenje,

GL_SMOOTH – umekšano senčenje (boje se interpoliraju na osnovu vrednosti zadatih u temenima).

Definiše način senčenja poligona. Podrazumevano je umekšano senčenje, kod koga se boja piksela u unutrašnjosti poligona izračunava interpolacijom boje temena. Kod jednobojnog senčenja, boja poslednjeg temena u primitivi definiše boju čitave primitive (osim kod poligona, gde prvo teme definiše boju primitive).

Zadatak

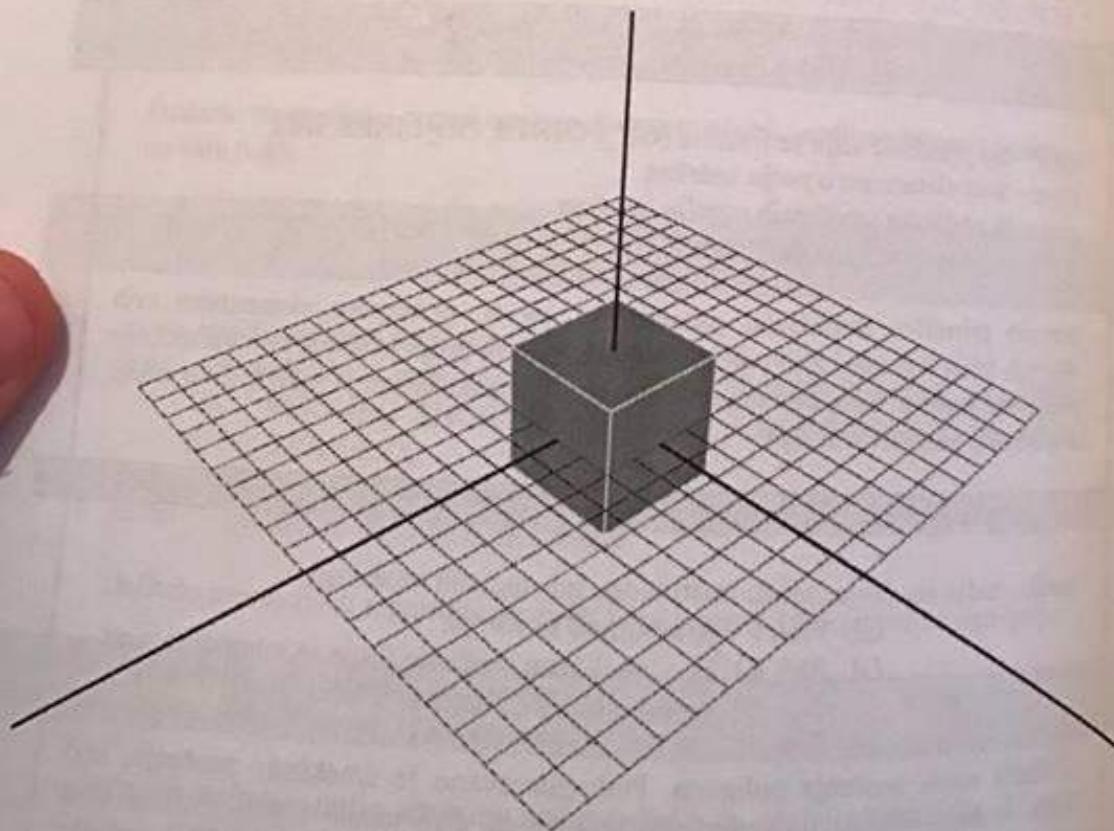
Napisati sledeće funkcije:

- void **DrawGrid(double dSize, int nSteps)** – koja iscrtava mrežu dimenzija $dSize \times dSize$ oko koordinatnog početka, u XZ-ravni sa $nSteps$ koraka,
- void **DrawCube(double dSize)** – koja iscrtava kocku stranice $dSize$ sa centrom u koordinatnom početku i
- void **DrawAxes(double len)** – koja iscrtava koordinatne ose dužine len .

U okviru funkcije **CGLRenderer::DrawScene()** formirati scenu koja se sastoji od:

- mreže dimenzija 5×5 , sa 20 koraka po svakoj osi, crne boje i debljine 1 piksel,
- kocke dimenzija $1 \times 1 \times 1$ sive bije sa ivicama bele boje debljine 2 piksela i
- koordinatnih osa dužina 10, crne boje i debljine 3 piksela.

Izgled scene prikazan je na slici 2.24.



Slika 2.24. Izgled scene

OPENGL - FIKSNA F
Rešenje

Napomena: U nastavku
zadruku. Obzirom da
poglavlju, funkcije Pre
zaključno sa pozivom
razmatranja. Cilj zadat
kao i njihov poziv u ob
iscrtavanja primitiva.

```
// GLPendere
#pragma once
class CGLRe
{
public:
    CGLR
    virt
    bool
    void
    void
    void
    void
    void
protected:
    HGLI
    ;
    // GLPendere
    /* Uklonjen
    void CGLRe
    {
        Wgl
        f
        glc
        gls
        f
        Wgl
        t
        void CGLRe
        Wgl
        g
    }

```

Rešenje

Napomena: U nastavku je prikazana klasa CGLRenderer dopunjena funkcijama traženim u zadatku. Obzirom da će postupak kreiranja 3D scene biti objašnjen tek u sledećem poglavljju, funkcije **PrepareScene()** i **Reshape()** u potpunosti, a funkciju **DrawScene()** zaključno sa pozivom **gluLookAt()** treba preuzeti onako kako su zadate, bez detaljnog razmatranja. Cilj zadatka je definisanje funkcija **DrawGrid()**, **DrawCube()** i **DrawAxes()**, kao i njihov poziv u okviru **DrawScene()** funkcije, uz promenu odgovarajućih parametara iscrtavanja primitiva.

```
// GLRenderer.h
#ifndef GLRENDERER_H
#define GLRENDERER_H

class CGLRenderer
{
public:
    CGLRenderer(void);
    virtual ~CGLRenderer(void);

    bool CreateGLContext(CDC* pDC);
    void PrepareScene(CDC* pDC);
    void Reshape(CDC* pDC, int w, int h);
    void DrawScene(CDC* pDC);
    void DestroyScene(CDC* pDC);

    void DrawGrid(double dSize, int nSteps);
    void DrawCube(double dSize);
    void DrawAxes(double len);

protected:
    HGLRC    m_hrc;
};

// GLRenderer.cpp
/* Uklonjeni deo koda */
void CGLRenderer::PrepareScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-
    glClearColor (1.0, 1.0, 1.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    //-
    wglMakeCurrent(NULL, NULL);
}

void CGLRenderer::DrawScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity ();
    gluLookAt(5.0, 5.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    glColor3f(0.5, 0.5, 0.5);
    glPolygonMode(GL_FRONT, GL_FILL);
}
```

```

        DrawCube(1.0);
        glColor3f(1.0, 1.0, 1.0);
        glPolygonMode(GL_FRONT, GL_LINE);
        glLineWidth(2.0);
        DrawCube(1.0);
        glColor3f(0.0, 0.0, 0.0);
        glLineWidth(1.0);
        DrawGrid(5.0, 20);
        glLineWidth(3.0);
        DrawAxes(10.0);
    //-----
    glFlush();
    SwapBuffers(pDC->m_hDC);
    wglMakeCurrent(NULL, NULL);
}

void CGLRenderer::Reshape(CDC *pDC, int w, int h)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective(40,(double)w/(double)h,1,100);
    glMatrixMode (GL_MODELVIEW);
    //-----
    wglMakeCurrent(NULL, NULL);
}

void CGLRenderer::DestroyScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----

    wglMakeCurrent(NULL,NULL);
    if(m_hrc)
    {
        wglDeleteContext(m_hrc);
        m_hrc = NULL;
    }
}

void CGLRenderer::DrawGrid(double dSize, int nSteps)
{
    double stepSize = dSize / nSteps;
    double Zmin = -dSize / 2.0;
    for(int i = 0; i < nSteps; i++)
    {
        double Xmin = -dSize / 2.0;
        glBegin(GL_QUAD_STRIP);
        for(int j = 0; j < nSteps; j++)
        {
            glVertex3d(Xmin, 0.0, Zmin);
            glVertex3d(Xmin, 0.0, Zmin+stepSize);
            Xmin += stepSize;
        }
        glEnd();
        Zmin += stepSize;
    }
}

```

```
void CGLRenderer::DrawCube(double dSize)
{
    double a = dSize;
    glBegin(GL_QUAD_STRIP);
    // Prednja leva vertikalna ivica
    glVertex3d(-a/2, a/2, a/2);
    glVertex3d(-a/2, -a/2, a/2);
    // Prednja desna vertikalna ivica
    glVertex3d( a/2, a/2, a/2);
    glVertex3d( a/2, -a/2, a/2);
    // Zadnja desna vertikalna ivica
    glVertex3d( a/2, a/2, -a/2);
    glVertex3d( a/2, -a/2, -a/2);
    // Zadnja leva vertikalna ivica
    glVertex3d(-a/2, a/2, -a/2);
    glVertex3d(-a/2, -a/2, -a/2);
    // Prednja leva vertikalna ivica
    glVertex3d(-a/2, a/2, a/2);
    glVertex3d(-a/2, -a/2, a/2);

    glEnd();

    glBegin(GL_QUADS);
    // Gornja stranica
    glVertex3d(-a/2, a/2, a/2);
    glVertex3d( a/2, a/2, a/2);
    glVertex3d( a/2, a/2, -a/2);
    glVertex3d(-a/2, a/2, -a/2);

    // Donja stranica
    glVertex3d(-a/2, -a/2, -a/2);
    glVertex3d( a/2, -a/2, -a/2);
    glVertex3d( a/2, -a/2, a/2);
    glVertex3d(-a/2, -a/2, a/2);

    glEnd();
}

void CGLRenderer::DrawAxes(double len)
{
    glBegin(GL_LINES);
    // X-osa
    glVertex3d(0.0, 0.0, 0.0);
    glVertex3d(len, 0.0, 0.0);

    // Y-osa
    glVertex3d(0.0, 0.0, 0.0);
    glVertex3d(0.0, len, 0.0);

    // Z-osa
    glVertex3d(0.0, 0.0, 0.0);
    glVertex3d(0.0, 0.0, len);

    glEnd();
}
```



Pogled u 3D

U ovoj glavi upoznaćemo se sa procesom formiranja trodimenzionalne (3D) scene u OpenGL-u. Definisaćemo osnovne transformacije kroz koje prolaze temena 3D objekata i način dobijanja željene slike na ekranu.

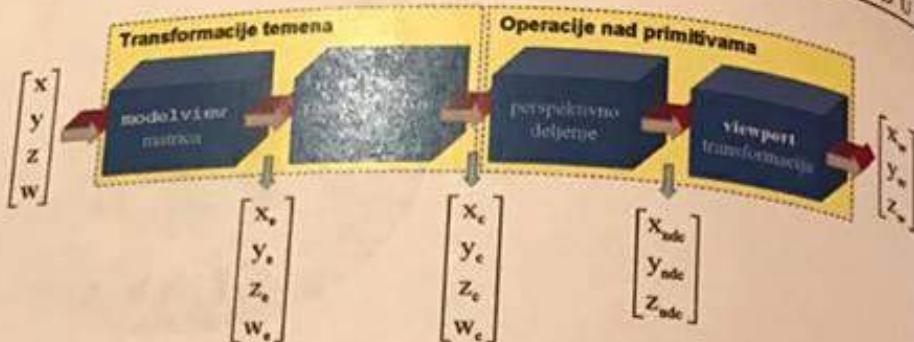
Postavljanje scene

Osnovni razlog korišćenja OpenGL-a jeste kreiranje fotorealističnih scena virtualnih svetova. Programer sada postaje umetnik, čije je jedino ograničenje njegova mašta. Da bi uspešno obavio svoj „umetnički“ zadatak, programer, baš kao i pravi umetnički fotograf, mora obaviti sledeće osnovne korake:

- napraviti ili pribaviti model koji će fotografisati i postaviti ga tako da se dobro uklapa u kompoziciju sa drugim objektima (skup svih objekata koji se vide i njihov uzajamni odnos nazvaćemo „scenom“),
- postaviti foto-aparat na odgovarajuću poziciju, tako da model bude dobro vidljiv i da se postigne željeni efekat cele kompozicije,
- izabrati sočivo na objektivu i podesiti *zoom*,
- nakon formiranja slike, razviti film i prilagoditi sliku određenom formatu (neke slike služe za Web prezentacije, dok druge služe za postere).

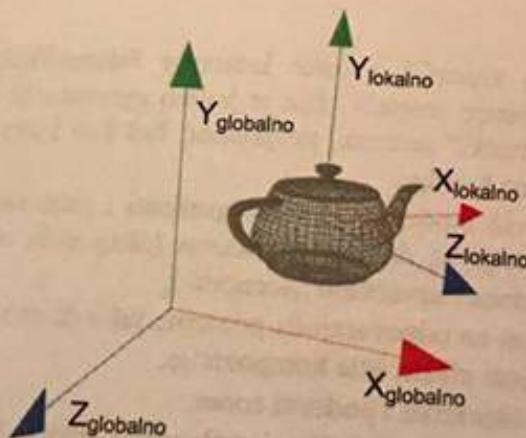
U digitalnom svetu, objekti su predstavljeni svojim matematičkim modelom, tj. koordinatama (ali i ostalim parametrima) svojih temena, a svaki od prethodnih koraka u procesu formiranja slike predstavlja jednu matematičku transformaciju nad tim temenima. Na slici 3.1 prikazan je proces sukcesivnih transformacija temena do dobijanja konačne slike.

Da se podsetimo, svako teme predstavljeno je sa svoje četiri (homogene) koordinate: x, y, z i w . Ove koordinate mogu se organizovati u vektor $[x \ y \ z \ w]^T$. To nam omogućuje da transformacije predstavimo matricama. Čitav postupak dobijanja 2D slike na osnovu 3D modela moguće je formalizovati i prevesti u množenje ovih matrica.



Slika 3.1. Proces transformacije temena

Prvi korak je kreiranje scene. Objekti se zadaju koordinatama svojih temena u lokalnom koordinatnom sistemu. **Lokalni koordinatni sistem** vezan je za objekat i najčešće se smešta u njegovo središte. Pomeranjem i rotiranjem lokalnog koordinatnog sistema u odnosu na **globalni (svetski) koordinatni sistem**, definiše se položaj objekta u sceni. Na slici 3.2 prikazan je primer odnosa lokalnog i globalnog koordinatnog sistema. Lokalni koordinatni sistem zarotiran je za ugao od 45° oko Y-ose globalnog koordinatnog sistema i transliran (pomeren) po sve tri koordinate.



Slika 3.2. Lokalni i globalni koordinatni sistemi

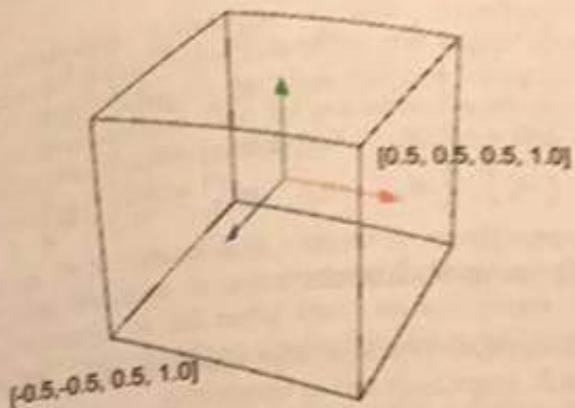
OpenGL koristi desni koordinatni sistem. To znači da X-osa raste udesno, Y-osa naviše, a Z-osa prema posmatraču. Tačnije, Z-osa raste u smeru napredovanja desne zavojnice, ako se ona rotira od X ka Y-osi. Inicijalno, posmatrač se nalazi u koordinatnom početku i gleda u pravcu negativne Z-ose. To znači da se bez primene transformacija mogu videti samo objekti koji imaju negativnu vrednost Z-koordinate.

Prvi zadatak prilikom stvaranja virtuelnog sveta jeste definisanje koordinata temena u lokalnom koordinatnom sistemu. Za slučaj jedinične kocke (kocka čije su stranice dužine 1), donje levo teme prednje stranice imajuće koordinate $[-0.5, -0.5, 0.5, 1.0]^T$ u lokalnom koordinatnom sistemu. Gornje desno teme prednje stranice biće na koordinatama $[0.5, 0.5, 0.5, 1.0]^T$.

Da bi se jedinične transformacije podrazumevala različite stvari, za primer jedinične Z-ose, da bi bila u tački $[0, 0, 1]^T$, i kocka posmatrač bi se razdvajati od udaljavanje kroz scenu.

Ipak, iako se napravićemo objekata na scenu (u održivoj scena neponavljanju).

Modelview transformacija objekata za sistem već označimo koordinata koja definira poziciju i rotaciju objekta.



Slika 3.3. Jedinična kocka

Da bi se jedinična kocka postavila na proizvoljno mesto u prostoru, potrebno je primeniti odgovarajuću transformaciju nad temenima kocke. Ta transformacija naziva se *modelview* transformacija, tj. transformacija modeliranja i pogleda. Pod pojmom modeliranja podrazumeva se razmeštanje objekata po sceni. Pogled predstavlja postavljanje i orijentaciju posmatrača, odnosno, kamere. Zašto su pomešane ove dve naizgled potpuno različite stvari? Zato što predstavljaju dva pogleda na istu transformaciju. Uzmimo ponovo za primer jediničnu kocku. Ako je posmatrač u koordinatnom početku i gleda duž negativne Z-ose, da bi bila vidljiva, kocku je potrebno pomeriti za, recimo, -5 jedinica u pravcu Z-ose. Umesto u tački $[-0.5, -0.5, 0.5, 1.0]^T$, jedno njen teme biće sada u tački $[-0.5, -0.5, -4.5, 1.0]^T$, i kocka će biti vidljiva. Potpuno identična slika dobila bi se ako bismo pomerili posmatrača za 5 jedinica u pravcu Z-ose. Dakle, pomeranje posmatrača ne može se razdvajati od pomeranja objekata u sceni. Transformaciju možemo posmatrati kao udaljavanje kocke od posmatrača, ili udaljavanje posmatrača od kocke. Rezultat je isti.

Ipak, iako se i modeliranje i pogled posmatrača predstavljaju istom transformacijom, npravljemo razliku između ova dva termina. Proces definisanja uzajamnog položaja objekata na sceni nazivaćemo **modeliranje**, a pomeranje svih objekata u sceni (tj. čitave scene) u odnosu na posmatrača, ili pomeranje samog posmatrača (pri čemu smatramo da je scena nepomična) nazivaćemo transformacijom **pogleda**.

Modelview matrica je dimenzija 4×4 i njenim množenjem sa vektorom koordinata temena objekata zadatih u lokalnom koordinatnom sistemu dobijamo vektor koordinata temena u sistemu vezanom za oko posmatrača (*eye* koordinate) - $[X_e \ Y_e \ Z_e \ W_e]^T$. Ako sa V označimo vektor koordinata temena u lokalnom koordinatnom sistemu, sa V_e vektor koordinata temena u koordinatnom sistemu vezanom za oko posmatrača, a sa M_{mv} maticu koja definiše *modelview* transformaciju, matrična jednačina ima sledeći oblik:

$$V_e = M_{mv} \cdot V$$

ili u razvijenom obliku:

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Osnovne operacije sa matricama

Pre nego što predemo na detaljno objašnjavanje transformacija, moramo se prvo upoznati sa načinom na koji OpenGL organizuje rad sa matricama.

Već smo videli da su sve transformacione matrice dimenzija 4×4 . Međutim, umesto da budu deklarisane na način standardan za C, ali i mnoge druge programske jezike, navođenjem dva indeksa, u OpenGL-u matrice su memorisane kao vektori. Na primer, umesto *double M[4][4]*, matrica se deklariše kao *double M[16]*. Dodatnu poteskuću za C/C++ programere predstavlja smeštanje elemenata **po kolonama**. Dakle, matrica ima sledeći oblik:

$$M = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

Postoje tri matrice koje definišu način iscrtavanja scene. To su:

- matrica modeliranja i pogleda (eng. *modelview matrix*),
- matrica projekcije (eng. *projection matrix*) i
- matrica texture (eng. *texture matrix*).

Matrica modeliranja i pogleda transformiše koordinate iz svetskog koordinatnog sistema u sistem vezan za oko posmatrača. Podrazumeva se da je posmatrač u koordinatnom početku i da je pogled usmeren u pravcu negativne Z-ose. Da bi se olakšalo čuvanje transformacija, umesto jedne *modelview* matrice, OpenGL implementira magacin koji može prihvatiti više *modelview* matrica. Koliko matrica može da se smesti u magacin zavisi od implementacije, a obično je to do 32 matrice. Samo matrica koja je na vrhu magacina utiče na transformaciju koordinata. Magacin olakšava formiranje složenih hijerarhijskih modела. Detaljnije o upotrebi magacina biće reči kasnije.

Matrica projekcije definiše „vidljivi“ prostor i transformiše 3D koordinate iz sistema vezanog za oko posmatrača u *clip* koordinate. U *clip* koordinatnom sistemu čitav vidljivi prostor ograničen je kockom koja se prostire od $-w$ do $+w$ po sve tri koordinate. Sve primitive van ovog prostora se odbacuju, a one koje presecaju kocku se „odsecaju“ na mestima preseka. Zato se ovaj koordinatni sistem i naziva sistem „odsecanja“ (eng. *clip*). I za projekcione matrice postoji magacin. Maksimalan broj elemenata ovog magacina je najčešće ograničen na 4.

Matrica *texture* omogućuje definisanje složenih transformacija nad teksturnim koordinatama. Teksturama je posvećena peta glava ove knjige. Tekture su obično slike koje se „lepe“ preko modela kako bi povećale njihovu realističnost. Način „lepljenja“ definisan je teksturnim koordinatama. Ono što *modelview* matrica predstavlja za prostorne koordinate temena, teksturna matrica predstavlja za teksturne koordinate temena. Dubina magacina teksturnih matrica je obično 10.

Postoji još jedna matrica. To je matrica boje, i njome se množi vektor koga čine R, G, B i A komponenta boje. Zgodna činjenica da postoje četiri komponente boja, omogućuje da se i na njih primeni transformacija na isti način kao i na koordinate. Međutim, primena ove matrice je vrlo „egzotična“ i nije podržana u svim implementacijama OpenGL-a.

Da bismo mogli da modifikujemo neku od datih matrica, najpre moramo da je selektujemo. Selekcijski se obavlja pozivom funkcije *glMatrixMode()* i prosleđivanjem jednog od 3 parametra: **GL_MODELVIEW**, **GL_PROJECTION** ili **GL_TEXTURE**.

Pre nego što izvršimo bilo kakvu transformaciju, moramo obrisati trenutnu vrednost matrice i postaviti neutralnu vrednost. Obzirom da se sve transformacije obavljaju množenjem matrica, neutralnu vrednost predstavlja **jedinična matrica**. Jedinična matrica je matrica koja ima nule za sve elemente osim elemenata na glavnoj dijagonali, koji su postavljeni na jedinice.

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Da bismo poništili trenutnu vrednost *modelview* matrice (tačnije matrice na vrhu odgovarajućeg magacina), potrebno je izvršiti sledeći kod:

```
float mat[16] = { 1.0, 0.0, 0.0, 0.0,
                  0.0, 1.0, 0.0, 0.0,
                  0.0, 0.0, 1.0, 0.0,
                  0.0, 0.0, 0.0, 1.0 };
glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(mat);
```

Funkcija *glMatrixMode()* selektuje odgovarajuću matricu, tako da sve transformacije koje sledi utiču baš na zadatu matricu, a funkcija *glLoadMatrix{fd}()* upisuje vrednost u zadatu matricu. Obzirom da je često potrebno učitati jediničnu matricu, postoji i posebna funkcija koja vrši ovu operaciju, bez potrebe da se definiše sama matrica. Primenom funkcije *glLoadIdentity()* prethodni kod se može napisati na sledeći način:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

Osnovne modelview transformacije

Postoje tri osnovne transformacije koje definišu položaj, orijentaciju i veličinu geometrijskih objekata u sceni. To su:

- translacija,
- rotacija i
- skaliranje.

Translacija

Translacija je geometrijska transformacija koja pomera temena objekata u pravcu X, Y i Z ose za zadate vrednosti. Ako sa (x, y, z) označimo koordinate temena pre transformacije, u a, b i c pomeraje duž X, Y i Z-ose, respektivno, a sa (x', y', z') koordinate nakon transformacije, translacija se može predstaviti na sledeći način:

$$\begin{aligned}x' &= x + a \\y' &= y + b \\z' &= z + c\end{aligned}$$

ili u matričnom obliku:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ z+c \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \wedge w' = w = 1$$

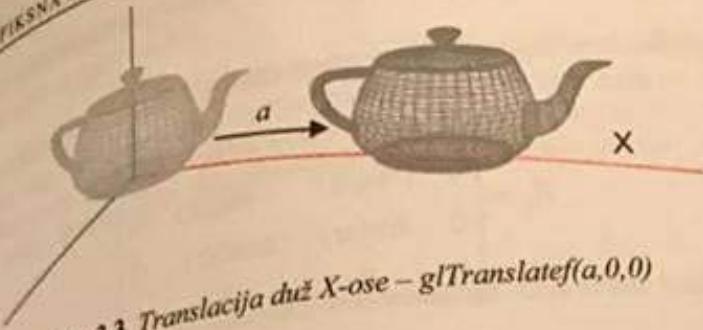
Da bi izvršili translaciju potrebno je definisati odgovarajuću matricu i pomnožiti tenu modelview matricu datom matricom.

```
double T[16] = { 1.0, 0.0, 0.0, 0.0,
                  0.0, 1.0, 0.0, 0.0,
                  0.0, 0.0, 1.0, 0.0,
                  a,      b,      c, 1.0 };
glMultMatrixd(T);
```

Naravno, postoji i mnogo lakši način da se ostvari translacija, a to je korišćenjem funkcije `glTranslate{fd}()`. Prethodni kod u potpunosti zamenjuje sledeća linija:

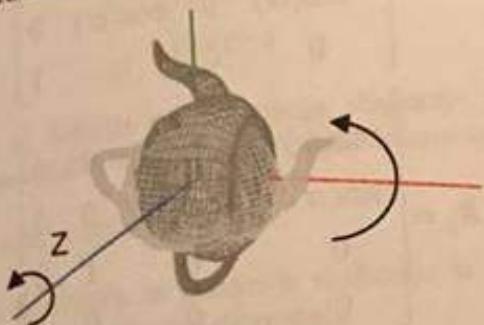
```
glTranslated(a, b, c);
```

Na slici 3.3 prikazana je translacija objekta duž X-ose.



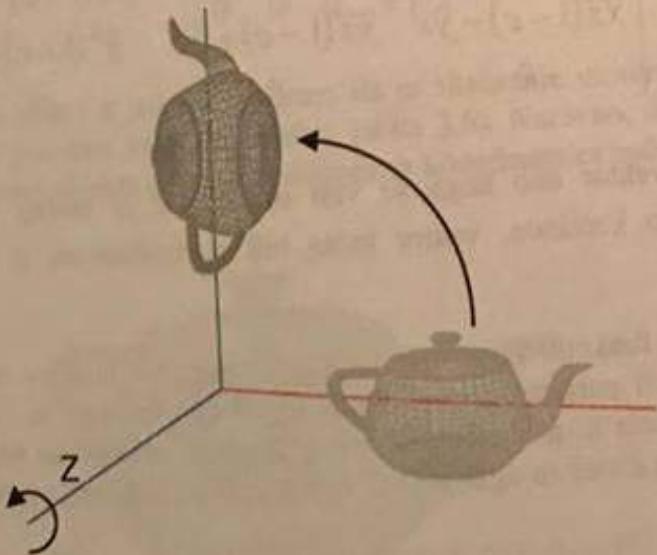
Slika 3.3. Translacija duž X-ose – `glTranslatef(a,0,0)`

Rotacija
Druge elementarnе geometrijske transformacije su rotacija oko koordinatnog početka za
zadan ugao. Na slici 3.4 prikazana je rotacija za 90° oko Z-ose.



Slika 3.4. Rotacija oko Z-ose za 90° – `glRotatef(90.0,0,0,1)`

Upoštevajte da rotacija meri se u pozitivnom smeru, tj. suprotno od kretanja kazaljki na časovniku.
Na primer, ukoliko rotiramo objekat oko Z-ose za zadati pozitivni ugao, i osa je okrenuta ka
nasu, objekat će se zarotirati ulevo. Ako je osa okrenuta od nas, tada će se objekat
zarotirati udesno.



Slika 3.5. Rotacija oko Z-ose za 90° kada objekat nije u koordinatnom početku

Ukoliko objekat nije u koordinatnom početku, rotacija će ne samo zarotirati objekat, već ga i pomeriti u prostoru (slika 3.5).

U matričnom obliku rotacije oko X, Y i Z ose za ugao α imaju sledeće oblike:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotacija oko proizvoljne ose još je komplikovanija, i ima sledeći oblik:

$$R = \begin{bmatrix} \tilde{x}^2(1-c)+c & \tilde{x}\tilde{y}(1-c)-\tilde{z}s & \tilde{x}\tilde{z}(1-c)+\tilde{y}s & 0 \\ \tilde{y}\tilde{x}(1-c)+\tilde{z}s & \tilde{y}^2(1-c)+c & \tilde{y}\tilde{z}(1-c)-\tilde{x}s & 0 \\ \tilde{x}\tilde{z}(1-c)-\tilde{y}s & \tilde{y}\tilde{z}(1-c)+\tilde{x}s & \tilde{z}^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Gde je $[\tilde{x} \tilde{y} \tilde{z}]$ vektor oko koga se vrši rotacija, c je $\cos(\alpha)$, a s je $\sin(\alpha)$. Da bi izračunavanje bilo korektno, vektor mora biti normalizovan, tj. mora imati jedinu dužinu.

Na sreću, postoji funkcija koja vrši izračunavanje ove matrice za nas. To je funkcija `glRotatef(fd)()`. Prvi parametar ove funkcije je ugao za koji se vrši rotacija, izražen u stepenima, a preostala tri parametra (x, y, z) definisu vektor oko koga se vrši rotacija. Na primer, rotacija oko Z-ose za ugao od 90° ostvaruje se sledećom naredbom:

```
glRotatef(90.0, 0.0, 0.0, 1.0);
```

Treba voditi računa da se ugao zadaje u stepenima, za razliku od trigonometrijskih funkcija iz `math` biblioteke koje zahtevaju uglove u radijanima. Vektor koji se prosledjuje ne mora

OPENGL - FIKSNA FUNKCIJA
normalizovan. Ukoliko
izračunavanja odgovarajućih
matrica.
Skaliranje
Skaliranje je treća elem
masti odgovarajućim k

OpenGL funkcija ko
primer, dvostruko uv
glScaled(2

Transformaciona m
koeficijente a, b i c

Ukoliko su koef
skaliranje koristi
deformacije obje

biti normalizovan. Ukoliko nije jedinične dužine, ova funkcija sama vrši normalizaciju pre izračunavanja odgovarajuće matrice.

I funkcija `glRotate*O`, kao i ostale transformacione funkcije (`glTranslate()`, `glScale()`, itd.) nakon izračunavanja transformacione matrice množi matricu koja se nalazi na vrhu trenutno selektovanog magacina, i to sa desne strane. Da bi uticale na promenu položaja objekata na sceni, geometrijske transformacije moraju se primenjivati na *modelview* matricu.

Skaliranje

Skaliranje je treća elementarna geometrijska transformacija. Ona svaku koordinatu temena množi odgovarajućim koeficijentom.

$$x' = a \cdot x$$

$$y' = b \cdot y$$

$$z' = c \cdot z$$

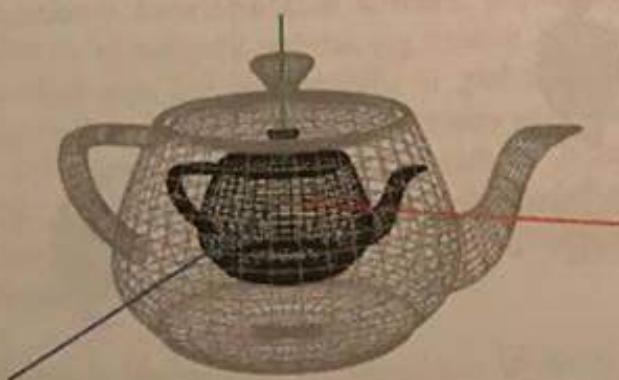
OpenGL funkcija kojom se definiše transformacija skaliranja je `glScale{fd}(a,b,c)`. Na primer, dvostruko uvećanje objekta ostvaruje se sledećim pozivom funkcije:

```
glScaled(2.0, 2.0, 2.0);
```

Transformaciona matrica **S**, kojom se ostvaruje skaliranje je vrlo jednostavna i sadrži koeficijente **a**, **b** i **c** raspoređene na glavnoj dijagonali.

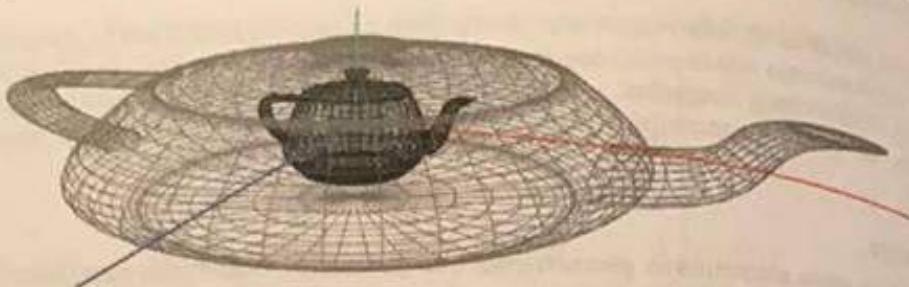
$$S = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ukoliko su koeficijenti **a**, **b** i **c** jednaki kažemo da je skaliranje uniformno. Uniformno skaliranje koristi se za promenu veličine objekta (slika 3.6). Naravno, da ne bi došlo do deformacije objekta, centar objekta mora se poklopiti sa koordinatnim početkom.



Slika 3.6. Uniformno skaliranje – `glScalef(a,a,a)`

Ukoliko su koeficijenti a , b i c različiti kažemo da je skaliranje neuniformno. Na slici 3.7 može se videti efekat neuniformnog skaliranja.



Slika 3.7. Neuniformno skaliranje – `glScalef(a,b,c)`

Neuniformno skaliranje, tj. zadavanje različitih faktora skaliranja po različitim osama nije poželjna transformacija jer narušava orientaciju normala i time osvetljenje čini „pogrešnim“. Detaljnije o normalama i osvetljenju biće reči u narednoj glavi. Za sada jedino treba znati da ukoliko postoji neuniformno skaliranje i aktivirano je osvetljenje, neophodno je aktivirati i automatsku normalizaciju, pozivom funkcije `glEnable()` i prosledivanjem parametra `GL_NORMALIZE`.

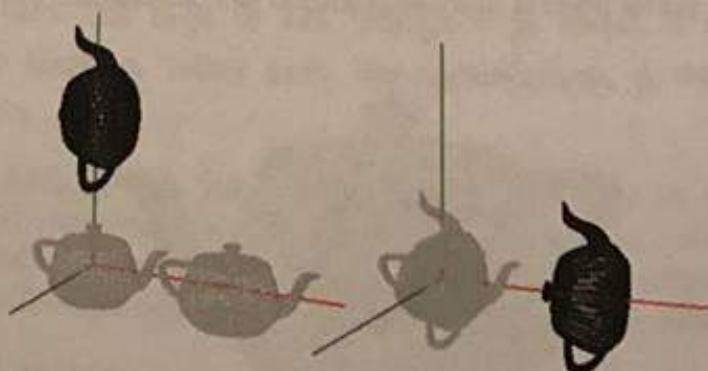
```
glEnable(GL_NORMALIZE);
```

Uključivanje automatske normalizacije može uticati na performanse programa, pa je poželjno izbegavati je.

Kombinovanje transformacija

Do sada smo videli efekte svake od geometrijskih transformacija pojedinačno. Međutim, da bi formirali scenu na način koji nama odgovara moramo kombinovati ove transformacije.

Redosled kojim se primenjuju transformacije vrlo je bitan. Na slici 3.8 prikazan je efekat translacije i rotacije u zavisnosti od redosleda kojim se izvršavaju ove transformacije. Ako se prvo izvrši translacija duž X-ose, pa zatim rotacija za 90° oko Z-ose, čajnik će biti na Y-osi zarotiran za 90° (slika 3.8.a). Međutim, ukoliko se prvo izvrši rotacija oko Z-ose, pa zatim translacija duž X-ose, čajnik će biti zarotiran, ali će ostati na X-osi (slika 3.8.b).



Slika 3.8. Različiti redosledi izvršavanja transformacija:
a) Translacija pre rotacije, b) Rotacija pre translacije

Kojim redosledom treba navoditi transformacije u programskom kodu da bi se dobio željeni efekat? Pogledajmo sledeći primer:

```
glTranslatef(1.5, 0.0, 0.0);
glRotatef(90.0, 0.0, 0.0, 1.0);
glutWireTeapot(0.5);
```

Prva linija koda definiše translaciju za 1.5 duž X-ose, druga rotaciju za 90° oko Z ose, a treća iscrtava čajnik. Šta mislite, gde će završiti čajnik? Na X ili na Y-osi? Na prvi pogled izgleda kao da će se čajnik naći na Y-osi (jer odgovara redosledu operacija na slici 3.8.a). Ali, to je pogrešno! Čajnik se nalazi na X-osi, zarotiran za 90° .

Da bismo ovo razjasnili, podsetimo se kako se vrše transformacije. Svaka od transformacija definiše svoju matricu, dimenzija 4×4 , kojom množi tekuću (selektovanu) matricu. Obzirom da vršimo modeliranje scene, to je *modelview* matrica. Ako sa C označimo tekuću matricu, nakon primene translacije njena vrednost je modifikovana i sada je:

$$C' = C \cdot T$$

Ovo je vrlo važno za razumevanje redosleda operacija – množenje se obavlja tako što se tekuća matrica pomnoži transformacionom matricom, ali sa **desne strane**.

Nakon translacije poziva se rotacija, dakle:

$$C'' = C' \cdot R = C \cdot T \cdot R$$

Matrica C'' množi vektor položaja (v) svakog od temena objekta da bi se odredio njihov konačni položaj u prostoru (v').

$$v' = C'' \cdot v = C \cdot T \cdot R \cdot v = (C \cdot (T \cdot (R \cdot v)))$$

Dakle, matrica R je prva matrica sa kojom se množi vektor v . Zato se čajnik prvo rotira oko Z-ose, pa tek nakon toga translira duž X-ose.

Da bi olakšali kombinovanje transformacija, uvećemo dva načina razmišljanja:

- razmišljanje u globalnom (svetskom) koordinatnom sistemu i
- razmišljanje u lokalnom koordinatnom sistemu.

Ako razmišljamo u **globalnom koordinatnom sistemu**, položaj objekta i transformacije uvek ćemo vršiti u odnosu na koordinatni početak tog sistema. Ovo je možda logičniji način razmišljanja, mada ponekad nije baš najzgodniji. U globalnom koordinatnom sistemu transformacije se u programskom kodu navode u redosledu **suprotnom** od onoga kojim se primenjuju. Odnosno, transformacije koje su bliže samom iscrtavanju objekta se pre izvršavaju od onih koje su dalje. Ako pogledamo prethodni primer, poziv funkcije `glRotatef()` je najbliži iscrtavanju čajnika, tako da se rotacija prva izvršava. Zatim se vrši translacija, jer nailazimo na funkciju `glTranslatef()`, itd.

Za razliku od razmišljanja u globalnom koordinatnom sistemu, gde transformacijama menjamo položaj objekta, razmišljanjem u **lokalnom koordinatnom sistemu** transformacijama zapravo pomeramo lokalni koordinatni sistem. Redosled izvršavanja

transformacija u tom slučaju je isti kao i redosled navodenja u programskom kodu. Dakle, u prethodnom primeru, prvo se javlja translacija, i lokalni koordinatni sistem biva translatovan duž X-ose globalnog koordinatnog sistema. Nakon toga vrši se rotacija lokalnog koordinatnog sistema. Sada je lokalni koordinatni sistem izmešten iz centra globalnog koordinatnog sistema i zarođivan za 90° . Objekat se iscrtava u lokalnom koordinatnom sistemu, a kako je ovaj sistem pomeren i zarođivan biće i sam objekat.

Bez obzira na koji način razmišljali, efekat transformacija je isti. U zavisnosti od situacije, izabraćemo jedan od ova dva načina da bismo lakše predstavili sebi šta se zapravo dešava sa objektom.

Iz prethodnog primera videli smo da su sve transformacije kumulativne. Ako smo nacrtali objekat koji je pomeren u odnosu na koordinatni početak, sledeći objekat biće nacrtan relativno u odnosu na prethodno zatećeno stanje. To je nekada i željeni efekat, ali nekada i nije. Da bismo sprečili kumulativno dodavanje transformacija, možemo jednostavno da učitamo jediničnu matricu (funkcijom `glLoadIdentity()`) i rešimo se svih prethodnih transformacija. Ovo, međutim, uglavnom nije dobro rešenje. Nakon svih transformacija modeliranja slede transformacije pogleda, koje orijentisu kameru. Na nesreću, transformacije pogleda i modeliranja su objedinjene u jednu matricu – *modelview* matricu. Obzirom da transformacije pogleda utiču na čitavu scenu, one se navode u kodu pre bilo koje druge transformacije modeliranja. To znači da množe tekuću *modelview* matricu pre ostalih Translate/Rotate/Scale funkcija. Učitavanjem jedinične matrice potpuno bi se izgubile transformacije pogleda.

Drugi razlog da ne obrišemo tekuću maticu transformacija je izgradnja hijerarhijskog modela, kod koga podobjekti zavise od položaja objekta koji je po hijerarhiji iznad njih. Na primer, ukoliko želimo da nacrtamo prst neke ruke, njegov položaj zavisi od položaja šake na kojoj je dati prst. Položaj šake zavisi od položaja podlaktice, a podlaktica od položaja nadlaktice. Čitava ruka vezana je za položaj tela u prostoru. Ovo je klasičan primer hijerarhije objekata, kod koje je trup u korenu hijerarhije, a prst na njenom kraju. Ako bismo obrisali transformacionu matricu nakon iscrtavanja samo jednog prsta, više ne bi mogli da nađemo pravi položaj za ostale prste.

Da bi se rešili svi prethodno navedeni problemi, umesto samo jedne transformacione matrice uvedeni su magacin matrica. Magacin omogućuje da se vrlo jednostavno i brzo zapamti trenutna transformaciona matrica, i polazeći od nje formira nova.

Funkcija `glPushMatrix()` pravi kopiju matice koja je na vrhu magacina, i novu matricu smešta u magacin. Dakle, odmah nakon poziva ove funkcije i na vrhu magacina i ispod njega nalaze se iste matrice. Zašto nam je potrebno da u magacina imamo dve iste matrice? Zato što će sve naredne transformacije menjati matricu koja je na vrhu, dok je ona ispod vrha potpuno bezbedna (nepromenljiva). Kada želimo da vratimo prethodno stanje dovoljno je samo da pozovemo funkciju `glPopMatrix()`. Matrica koja je na vrhu magacina biva odbačena, a ona ispod nje postaje tekuća jer dolazi na vrh magacina.

Razmotrimo primer kombinovanja transformacija na primeru robota koji se sastoji od tela, dve ruke sa po dva segmenta i glave.

```
void CGLRenderer::DrawRobot()
{
    glPushMatrix();
    // Telo
    glScalef(1.0, 2.0, 0.5);
    DrawCube(1.0);
    glPopMatrix();

    glPushMatrix();
    // Desna nadlaktica
    glTranslatef(0.65, 1.0, 0.0);
    glRotatef(-45.0, 1.0, 0.0, 0.0);
    glTranslatef(0.0, -0.5, 0.0);
    glPushMatrix();
    glScalef(0.3, 1.0, 0.5);
    DrawCube(1.0);
    glPopMatrix();

    // Desna podlaktica
    glTranslatef(0.0, -0.5, 0.0);
    glRotatef(-45.0, 1.0, 0.0, 0.0);
    glTranslatef(0.0, -0.5, 0.0);
    glPushMatrix();
    glScalef(0.3, 1.0, 0.5);
    DrawCube(1.0);
    glPopMatrix();

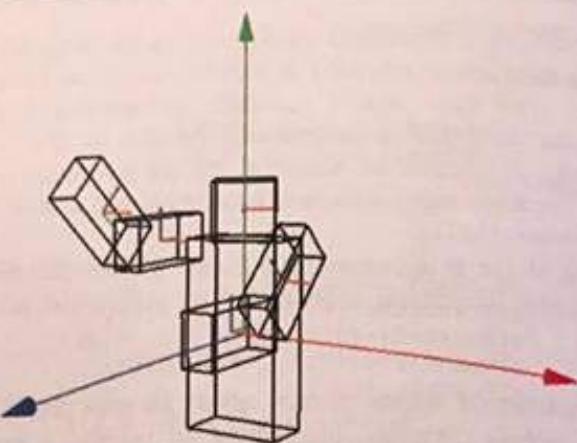
    glPopMatrix();

    glPushMatrix();
    // Leva nadlaktica
    glTranslatef(-0.65, 1.0, 0.0);
    glRotatef(-90.0, 1.0, 0.0, 0.0);
    glTranslatef(0.0, -0.5, 0.0);
    glPushMatrix();
    glScalef(0.3, 1.0, 0.5);
    DrawCube(1.0);
    glPopMatrix();

    // Leva podlaktica
    glTranslatef(0.0, -0.5, 0.0);
    glRotatef(-45.0, 1.0, 0.0, 0.0);
    glTranslatef(0.0, -0.5, 0.0);
    glPushMatrix();
    glScalef(0.3, 1.0, 0.5);
    DrawCube(1.0);
    glPopMatrix();

    glPopMatrix();

    glPushMatrix();
    // Glava
    glTranslatef(0.0, 1.3, 0.0);
    glRotatef(30.0, 0.0, 1.0, 0.0);
    glScalef(0.6, 0.6, 0.6);
    DrawCube(1.0);
    glPopMatrix();
}
```



Slika 3.9. Robot - primer kombinovanja transformacija

Za crtanje svih delova robota koristićemo kocku (funkcija **DrawCube()**) definisanu u prethodnom poglavlju. Obzirom da nam je za crtanje delova robota u većini slučajeva potreban kvadar a ne kocka, primenom skaliranja menjaćemo oblik i pretvorimo kocku u kvadar.

Centralni deo figure, i koren hijerarhije je telo. Pozivom funkcije **glScalef(1.0, 2.0, 0.5)**, od jedinične kocke dobijamo kvadar dimenzija $1.0 \times 2.0 \times 0.5$. Da ovo skaliranje nismo stavili unutar **glPushMatrix()/glPopMatrix()** funkcijskog para, svi naredni objekti bili bi modifikovani ovom transformacijom. Obzirom da to nije ono što želimo (namera je samo da pretvorimo kocku u odgovarajući kvadar), funkcijom **glPushMatrix()** čuvamo trenutnu transformaciju (pre poziva funkcije **DrawRobot()** sigurno postoje transformacije kojima se čitav robot postavlja u sceni).

Obzirom da l
 definiše polož
 ruke.

Zatim crtamo desnu nadlakticu (gornji deo ruke). Da bi je postavili na pravo mesto, potrebno je pomeriti nadlakticu tako da se njen vrh poklopi sa vrhom tela uz istovremeno pomeranje u stranu (tako da bude priljubljena uz bok tela). Naravno, potrebno je i da nadlaktica može da se rotira u ramenom zglobu. Kako da sve ovo postignemo? Prvo ćemo je pretvoriti u kvadar dimenzija $0.3 \times 1.0 \times 0.5$. Za to koristimo neuniformno skaliranje. Da to skaliranje ne bi uticalo na ostale objekte koji se iscrtavaju nakon nadlaktice, „uokvirićemo“ ovu transformaciju **glPushMatrix()/glPopMatrix()** parom. Sada je potrebno omogućiti rotaciju oko središta gornje stranice. Razmišljajmo, recimo, u globalnom koordinatnom sistemu. Kocka se crta oko koordinatnog početka. Da bi ostvarili datu rotaciju potrebno je spustiti objekat za polovicu njegove visine. U našem slučaju to je 0.5 . Dakle, potrebno je pozvati funkciju **glTranslatef(0.0, -0.5, 0.0)**. Zatim se vrši rotacija oko X-ose, recimo za 45° , što zahteva poziv **glRotatef(-45, 1.0, 0.0, 0.0)**. Nakon rotacije, nadlakticu je potrebno pomeriti tako da dodirne rame. Potrebno je, dakle, translirati po X-osi za zbir polovine širine tela i ruke ($0.5 + 0.15$), a po Y-osi dok ne dosegne rame (1.0), odnosno pozvati funkciju **glTranslatef(0.65, 1.0, 0.0)**. Obzirom da smo razmisljali u globalnom koordinatnom sistemu, redosled pisanja naredbi treba biti suprotan od redosleda izvršavanja transformacija.

Usmerava

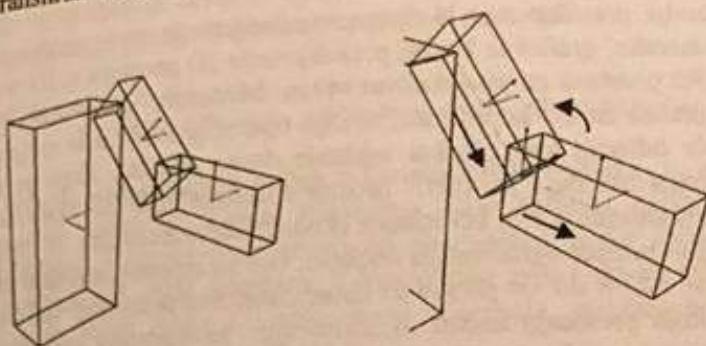
Već je rečeno
 negativne Z-
 dovesti objek
 čitavu scenu,
 modeliranja).

U opštem slu
 tako da ima
 olakšava post
 voj

Prva tri paran
 se gleda, a l
 posmatrač n
 transformacij

Prelazimo na crtanje desne podlaktice. Pošto je ona vezana za poziciju desne nadlaktice ne smemo poništiti translacije i rotacije postavljene crtanjem nadlaktice (tj. ne smemo pozvati

`glPopMatrix()`. Postupak je sličan crtanju nadlaktice. Podlaktica se pomera za pola dužine nadlaktice (u pravcu negativne Y-ose), rotira za zadati ugao, a zatim pomera za pola dužine nadlaktice (takođe u pravcu negativne Y-ose). Ovo je odličan primer za demonstraciju prednosti razmišljanja u lokalnom koordinatnom sistemu. Na slici 3.10 može se videti položaj i orientacija lokalnog koordinatnog sistema za svaki od podobjekata. Da bi se podlaktica postavila u pravilan položaj, lokalni koordinatni sistem je potrebno translirati za pola dužine nadlaktice u pravcu -Y-ose lokalnog koordinatnog sistema nadlaktice, zatim ga (kada se nalazi na donjoj stranici nadlaktice) zarotirati za dati ugao, a nakon toga translirati za pola dužine podlaktice, ponovo u pravcu -Y-ose lokalnog koordinatnog sistema (koji je u međuvremenu transliran i rotiran).



Slika 3.10. Pomeranje lokalnog koordinatnog sistema

Okizrom da leva ruka ne zavisi od položaja desne ruke, vraćamo se na transformaciju koja definije položaj tela (pozivom funkcije `glPopMatrix()`) i započinjemo proces crtanja druge ruke.

Usmeravanje pogleda

Već je rečeno da se u OpenGL-u posmatrač nalazi u koordinatnom početku i gleda u pravcu negativne Z-ose. Da bi čitava scena bila vidljiva, potrebno je translacijom i rotacijom dovesti objekte tako da budu „ispred“ posmatrača. Ove transformacije treba da utiču na čitavu scenu, što znači da se u kodu moraju naći na samom početku (pre transformacije modeliranja).

U opštem slučaju nije jednostavno odrediti transformacione parametre rotacija i translacija tako da imamo željeni pogled na scenu. Na sreću, postoji funkcija koja maksimalno olakšava postavljanje pogleda. To je funkcija `gluLookAt()`.

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

Tra tri parametra određuju položaj posmatrača (tj. oka), sledeća tri koordinate tačke u koju se gleda, a poslednja tri vektor upravan na pogled (*up* vektor). Na primer, ukoliko se posmatrač nalazi na poziciji (10, 5, 1), gleda u tačku (-5, 0, 5) i stoji uspravno, transformacija pogleda zadaje se na sledeći način:

```

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(10.0, 5.0, 1.0, -5.0, 0.0, 5.0, 0.0, 1.0, 0.0);

```

Funkcija `gluLookAt()` mapira položaj posmatrača u koordinatni početak, tačku u koje se gleda na negativnu Z-osu, a *up* vektor na pozitivnu Y-osu. *Up* vektor ne sme biti paralelni pravcu pogleda.

Projekcija

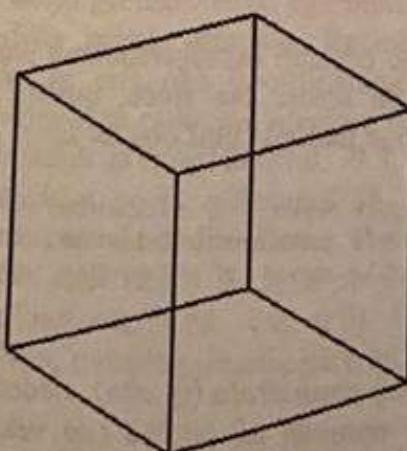
Projekcija predstavlja preslikavanje N-dimenzionalnog prostora u prostor sa manje od N dimenzija. U računarskoj grafici se koristi preslikavanje 3D prostora u 2D prostor, odnosno „projektovanje“ 3D prostora na projekcionu ravan. Međutim, ukoliko bi OpenGL već na ovom stadijumu prešao na 2D koordinate, mnoge operacije ne bi bile moguće. Na primer, ne bi bilo moguće odrediti koji objekat zaklanja druge objekte. Zato, je tačnije reći da **projekciona matrica** određuje „vidljivi“ prostor i transformiše 3D koordinate iz sistema vezanog za oko posmatrača u *clip* koordinate (koordinate odsecanja). U *clip* koordinatnom sistemu čitav vidljivi prostor ograničen je kockom, čije su strane poravnate sa koordinatnim osama, a prostire se od $-w$ do $+w$ po sve tri koordinate. Sve primitive van ovog prostora se odbacuju, a one koje presecaju kocku se „odsecaju“ na mestima preseka. Zato se ovaj koordinatni sistem i naziva sistem „odsecanja“ (eng. *clip*).

OpenGL omogućuje dva tipa projekcije:

- ortografsku i
- perspektivnu.

Ortografska projekcija

Ortografska projekcija je projekcija koja čuva dužine i paralelnost linija. Udaljeni objekti se ne smanjuju, a bliski objekti se ne deformišu. Površine koje su paralelne projekcionoj ravni zadržavaju odnos visine i širine, pa je moguće meriti delove tih površina na osnovu ortografske projekcije. Upravo zbog tih merenja, mnoge profesije zahtevaju izradu tehničkih crteža korišćenjem ortografske projekcije (npr. mašinstvo, građevina, itd.). Na slici 3.11 prikazan je žičani model kocke u ortografskoj projekciji.



Slika 3.11. Žičana kocka u ortografskoj projekciji

Ukoliko pažljivij
Zadnja stranica
paralelne. Razlog
perspektivu, a on

Da bismo dobili
kvadra, sa stranic
vidljivi prostor,
odsecanja. Prim
one koje se nala
unutrašnje a neki
OpenGL-u, ortog

void gl

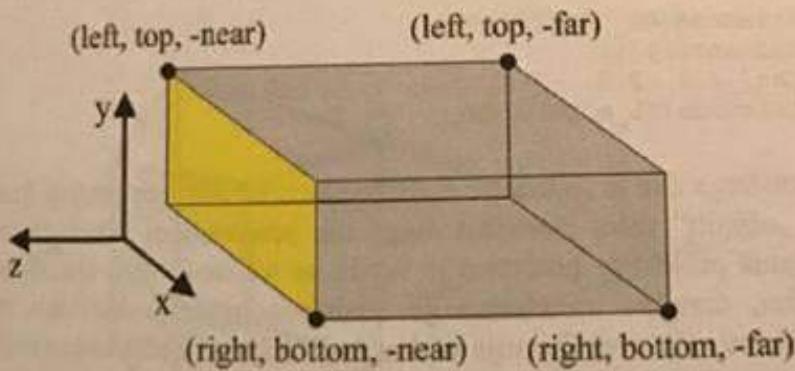
Kao što je već
(0,0,0), i gleda
odsecanja (istov
odsecanja. Para
bottom i top Y
označeni su svih

Ukoliko pažljivije pogledamo sliku 3.11, učiniće nam se da je nešto na njoj pogrešno. Zadnja stranica izgleda kao da je veća od prednje, odnosno kao da stranice nisu baš paralelne. Razlog za to je činjenica da je naš mozak navikao da tumači sliku koja poseduje perspektivu, a ona je u ovom slučaju eliminisana.

Da bismo dobili ortografsku projekciju, potrebno je da vidljivi prostor bude u obliku kvadra, sa stranicama poravnatim sa koordinatnim osama (slika 3.12). Ravni koje omeđuju vidljivi prostor, u ovom slučaju ravni koje sadrže stranice kvadra, nazivaju se **ravni odsecanja**. Primitive koje se nalaze sa „unutrašnje“ strane ravni odsecanja se prikazuju, one koje se nalaze sa spoljašnje strane se odbacuju, a one čija se neka temena nalaze sa unutrašnje a neka sa spoljašnje strane bivaju presečene. Odatle i potiče naziv ovih ravni. U OpenGL-u, ortografsku projekciju definisemo pozivom funkcije:

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

Kao što je već rečeno, posmatrač se nalazi u koordinatnom početku, tj. na koordinatama (0,0,0), i gleda duž negativne Z-ose. Parametar *near* definiše daljinu prednje ravni odsecanja (istovremeno i projekcione ravni) od posmatrača, a *far* daljinu zadnje ravni odsecanja. Parametri *left* i *right* određuju X-koordinatu leve i desne ravni odsecanja, a *bottom* i *top* Y-koordinatu donje i gornje ravni odsecanja, respektivno. Na slici 3.12 označeni su svi parametri funkcije *glOrtho*.



Slika 3.12. Vidljivi prostor ortografske projekcije

Matrica kojom se definiše ortografska projekcija ima sledeći oblik:

$$P_O = \begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{right + left}{right - left} \\ 0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & \frac{-2}{far - near} & -\frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Projekcionu matricu nije potrebno pamtiti, jer je funkcija `glOrtho()` sama formira osnovu zadatih parametara i množi sa vrhom trenutno aktivnog magacina matrica. Da bi ispunila svoju ulogu, funkcija `glOrtho()` treba biti pozvana tek nakon aktiviranja moda za definisanje projekcije, tj. nakon aktiviranja magacina projekcionih matrica. Sledi primer definisanja ortografske projekcije:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-2.0, 2.0, -2.0, 2.0, 1.0, 100.0);
glMatrixMode(GL_MODELVIEW);
```

Obzirom da množenje dve projekcione matrice nema smisla, pre poziva funkcije `glOrtho()` neophodno je „očistiti“ vršni elemenat magacina učitavanjem jedinične matrice. Takođe, nakon postavljanja projekcije potrebno je vratiti se na *modelview* transformacije. Ukoliko se ovo ne učini, naredne transformacije translacije/rotacije/skaliranja modifikovale bi projekcionu matricu, što svakako nije efekat koji želimo. Zbog efikasnosti koda, ukoliko nema nekog specijalnog razloga da se projekcija menja pri svakom iscrtavanju scene, definisanje projekcije se obično vrši u okviru funkcije `CGLRender::Reshape()`. Tada dolazi do promene veličine i oblika prozora, pa je potrebno preračunati parametre projekcije.

Perspektivna projekcija

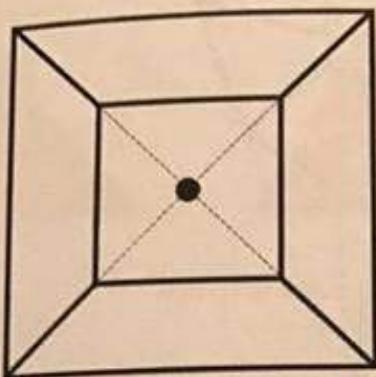
Perspektivna projekcija je mnogo bliža načinu na koje oko ili kamera sagledavaju svet oko sebe. Sa uvodenjem perspektive predmeti postaju sve manji kako se udaljavaju od posmatrača, a paralelne linije prestaju biti paralelne (osim ako se ne nalaze u ravni paralelnoj sa projekcionom ravnim). Svima je poznat primer prave deonice pruge i šina koje stvaraju privid da se, iako znamo da su potpuno paralelne, sve više približavaju jedna drugoj dok se konačno se spoje negde na horizontu. To je efekat perspektive.

Obzirom da živimo u svetu koji je u dobroj meri pravougaoni (zidovi zgrada su vertikalni, odnosno zaklapaju prav ugao sa ulicom, knjige, nameštaj, pa čak i računari obično imaju oblik kvadra, itd.) efekat perspektive je vrlo uočljiv, jer ona najviše ima uticaja na paralelne

Kada je linija pogodna jednu koordinatnu ivice kocke koje se ne sekut u jednom

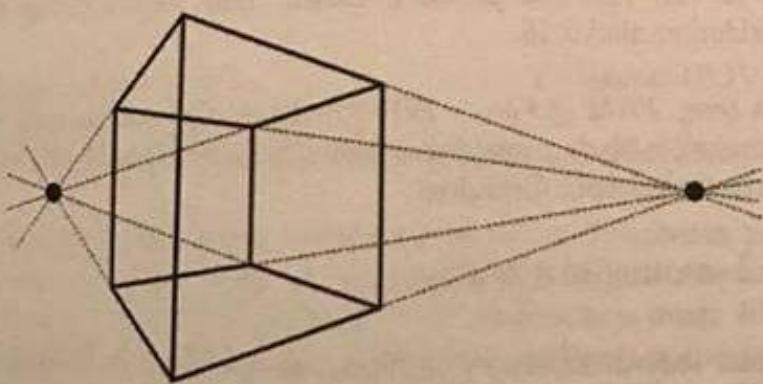
Ako se zarotiraju projekciju sa paralelne, a proc

linije i prave uglove. Zato ćemo i demonstrirati efekte perspektive na jediničnoj kocki čije su ivice poravnate sa koordinatnim osama.



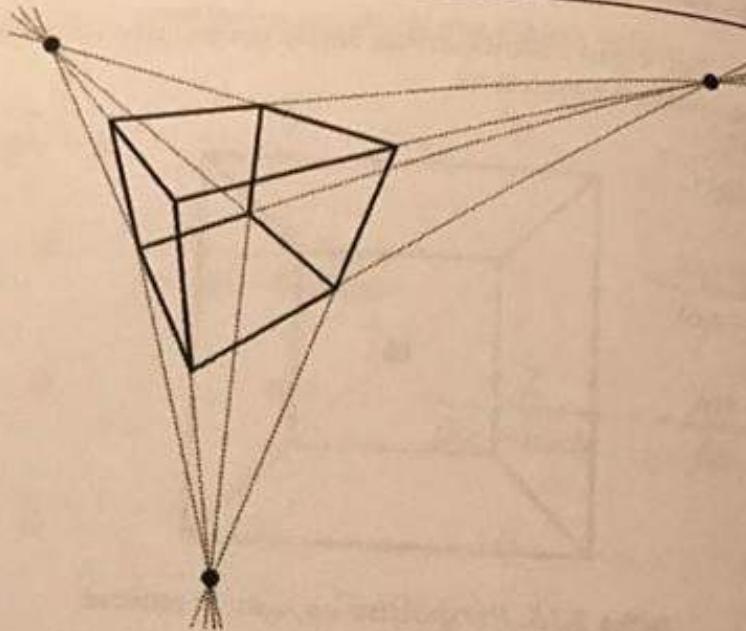
Slika 3.13. Perspektiva sa jednom tačkom

Kada je linija pogleda upravna na jednu stranicu kocke, tj. projekcionala ravan seče samo jednu koordinatnu osu, dobijamo **perspektivu sa jednom tačkom** (slika 3.13). Susedne ivice kocke koje pripadaju stranicama upravnim na pravac pogleda ostaju pod pravim uglovima, a ne gubi se ni paralelnost naspramnih ivica. Ukoliko produžimo bočne ivice, one se sekut u jednoj tački. Odатле potiče i naziv ove projekcije.



Slika 3.14. Perspektiva sa dve tačke

Ako se zarođivamo oko Y-ose, tj. ako projekcionala ravan seče dve koordinatne ose, dobijamo projekciju sa **dve tačke** (slika 3.14). Vertikalne ivice ostaju vertikalne i međusobno paralelne, a produžeci svih ostalih sekut se u dve tačke na horizontu.



Slika 3.15. Perspektiva sa tri tačke

Ukoliko projekciona ravan seče sve tri koordinatne ose, dobijamo projekciju sa tri tačke (slika 3.15).

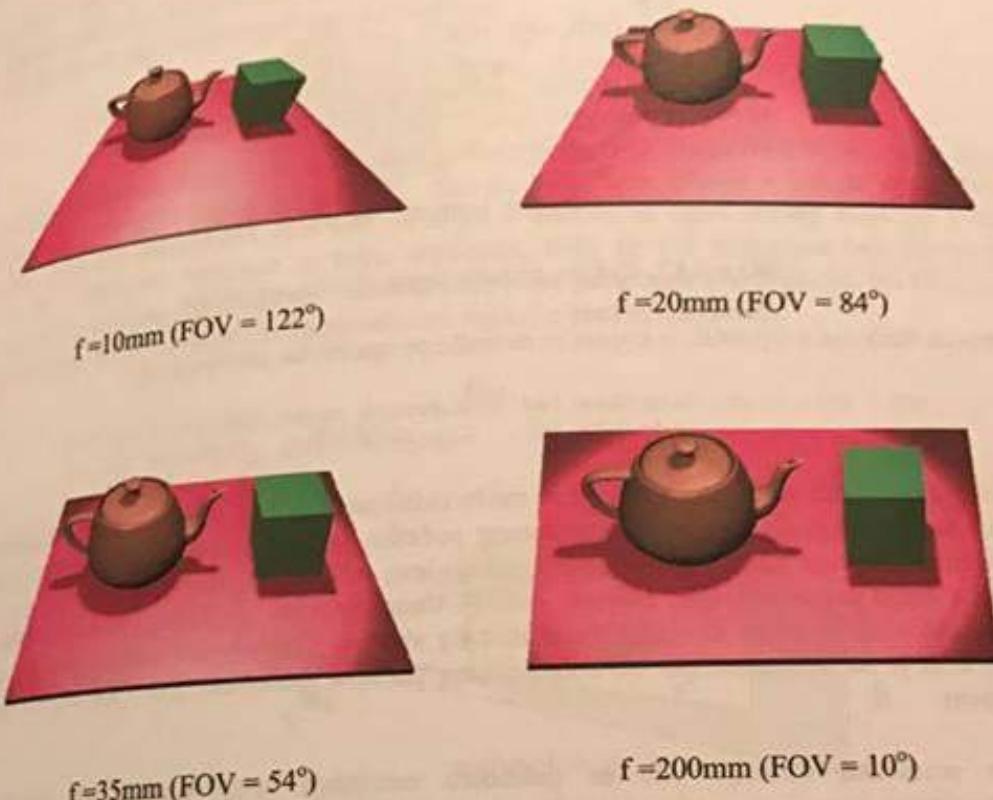
Efekat perspektive zavisi od žižne duljine sočiva. Što je ona manja, to su objekti deformisaniji, ali se vidi veći deo prostora. Efekat žižne duljine sočiva na perspektivu najbolje se može videti na slici 3.16.

Ugao vidnog polja (eng. *Field of View* – FOV) zavisi od tipa sočiva, njegovih dimenzija i žižne duljine. Uzimajući u obzir „standardno sočivo“, zavisnost ugla vidnog polja od žižne duljine može se izraziti sledećom formulom:

$$FOV = 2 \cdot \arctan(36 / (2 \cdot f))$$

gde je f žižna duljina sočiva izražena u milimetrima. Obzirom da funkcija \arctan obično izračunava vrednost u radijanima (u programskom jeziku C, to su funkcije atan i atan2), rezultat prethodne formule potrebno je pomnožiti sa 57.295779513.

Važna napomena: Sve OpenGL funkcije očekuju uglove izražene u stepenima, dok sve funkcije iz standardne matematičke biblioteke za jezik C (**math**) koriste radijane. Prilikom pisanja programskog koda obavezno izvršiti odgovarajuću konverziju.



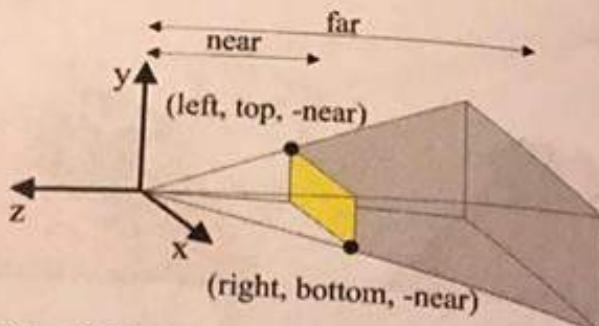
Slika 3.16. Efekat žižne daljine sočiva na perspektivu

Ljudsko oko ima ekvivalentnu žižnu daljinu od 48.24mm. Primenom prethodne formule dobijamo da to ogovara vidnom polju od 41° . Obično se zbog jednostavnosti pamćenja, a i lakšeg izračunavanja uglova, koristi vrednost od 45° . Sočiva koja imaju žižnu daljinu manju od ljudskog oka (često se koristi 50mm kao referentna vrednost) nazivaju se širokougaona sočiva. Standardna širokougaona sočiva imaju žižne daljine: 35mm ili 28mm. Širokougaona sočiva sa jako malom žižnom daljinom (obično 10-15mm) poznata su pod nazivom **riblje oko**. To su sferna sočiva koja imaju jako veliku distorziju, ali im je zato ugao vidnog polja jako veliki (preko 100°). Sočiva sa žižnom daljinom većom od 50mm nazivaju se **telefoto sočiva**. Za njih je karakteristično da uvećavaju prikaz i približavaju udaljene objekte. U tabeli 3.1 prikazane su žižne daljine i uglovi vidnog polja nekih standardnih sočiva.

Tabela 3.1. Pregled standardnih sočiva

$f[\text{mm}]$	15	20	24	28	35	50	85	135	200
$\text{FOV } [^\circ]$	100.39	83.974	73.74	65.47	54.432	39.598	23.913	15.189	10.286

Da bi se stvorio efekat perspektive, vidljivi prostor ne sme biti u obliku kvadra (kao kod ortografske projekcije), već ima oblik zarubljene piramide. Vrh piramide nalazi se u oku posmatrača, a ugao nagiba stranica određen je vidnim poljem.



Slika 3.17. Vidljivi prostor perspektivne projekcije

Osnovna funkcija u OpenGL-u kojom se definiše perspektivna projekcija je:

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

Značenje pojedinih parametara najbolje se može videti na slici 3.17. Prednja ravan odsečanja nalazi se na rastojanju *near* od koordinatnog početka (tj. oka posmatrača) i definisana je manjom osnovicom zarubljene piramide (gornja leva tačka je na koordinatama (*left, top, -near*), a donja desna na (*right, bottom, -near*)). Veća osnovica zarubljenje piramide dobija se projektovanjem zraka iz centra koordinatnog sistema kroz temena manje osnovice na ravan koja je na udaljenosti *far* od koordinatnog početka. Parametri *near* i *far* moraju biti pozitivni.

Poziv prethodne funkcije ima za posledicu množenje vršne matrice na trenutno selektovanom magacinu matrica (potrebno je da to bude magacin projekcionih matrica da bi dobili željeni efekat) sa matricom P_p koja ima sledeći oblik:

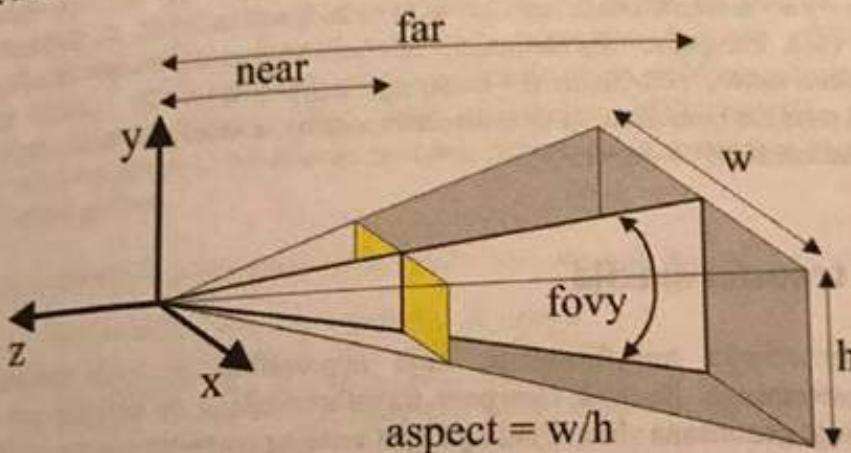
$$P_p = \begin{bmatrix} \frac{2 \cdot \text{near}}{\text{right} - \text{left}} & 0 & \frac{\text{right} + \text{left}}{\text{right} - \text{left}} & 0 \\ 0 & \frac{2 \cdot \text{near}}{\text{top} - \text{bottom}} & \frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} & 0 \\ 0 & 0 & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} & -\frac{2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Da bi određivanje duljina pojedinih primitiva od posmatrača bilo što preciznije, potrebno je da odnos *far/near* bude što je moguće manji. **Parametar *near* ne sme biti 0!** To je najčešća greška koju početnici prave, u želji da vide „sve što se nalazi ispred oka“. Ukoliko je parametra *near* jednak nuli, projekcija nije definisana.

funkcija `glFrustum()` često nije pogodna za rad jer se nigde eksplisitno ne definišu osnovni parametri perspektivne projekcije, kao što su FOV i aspekt. Zato se u praktičnim primenama mnogo češće koristi sledeće funkcija:

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

Prvi parametar ove funkcije definiše ugao vidnog polja (FOV) meren u YZ-ravni, a drugi odnos širine i visine projektovane slike. Ako je, na primer, *aspect* = 2.0, to znači da je video polje posmatrača dva puta šire po X-osi u odnosu na ugao vidnog polja po Y-osi. Ako je i *viewport* definisan sa istim aspektom, slika će biti prikazana bez distorzije. Preostala dva parametra imaju isto značenje kao i kod prethodne dve funkcije. Na slici 3.18 prikazana je fizička interpretacija parametara funkcije `gluPerspective()`.



Slika 3.18. Značenje parametara funkcije `gluPerspective()`

Funkcija `gluPerspective()` definiše projekcionu matricu na sledeći način:

$$P_p = \begin{bmatrix} \frac{f}{\text{aspect}} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2 \cdot far \cdot near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Ugle je $f = \text{ctan}(fovy/2) = 1/\tan(fovy/2)$.

Na osnovu svega što je do sada rečeno, definisanje perspektivne projekcije koja odgovara pogledu ljudskog oka (fov je u granicama od 40 do 50), može se izvršiti na sledeći način:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(45.0, aspect, 1.0, 100.0);
glMatrixMode(GL_MODELVIEW);
```

Vrednost promenljive *aspect* treba odrediti na osnovu odnosa širine i visine prozora u kome se prikazuje slika.

Perspektivno deljenje

Nakon projekcije, odnosno dobijanja *clip* koordinata (x_c, y_c, z_c, w_c) , prelazi se na normalizovane koordinate uredaja tako što se prve tri koordinate (x_c, y_c, z_c) podelje četvrtom koordinatom (w_c) . Perspektivnim deljenjem dobijaju se normalizovane koordinate uredaja (eng. *normalized device coordinates*) - $(x_{ndc}, y_{ndc}, z_{ndc})$. Čitav vidljivi prostor sveden je na kocku čije su stranice poravnate sa koordinatnim osama, a validne koordinate po sve tri ose kreću se u granicama od -1 do +1.

Viewport transformacija

Poslednja geometrijska transformacija, koja neposredno prethodi rasterizaciji, jeste **viewport** transformacija. Zadatak **viewport** transformacije je da definiše gde se u okviru prozora prikazuje generisana slika. Funkcija koja zadaje parametre ove transformacije je:

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

Parametri *x* i *y* određuju donji levi ugao, a *width* i *height* širinu i visinu slike, respektivno. Svi parametri ove funkcije zadate su u pikselima.

Matematički posmatrano, **viewport** transformacija preslikava normalizovane koordinate uredaja (x_{ndc}, y_{ndc}) u koordinate prozora (x_w, y_w) korišćenjem sledećih formula:

$$x_w = (x_{ndc} + 1) \left(\frac{width}{2} \right) + x$$

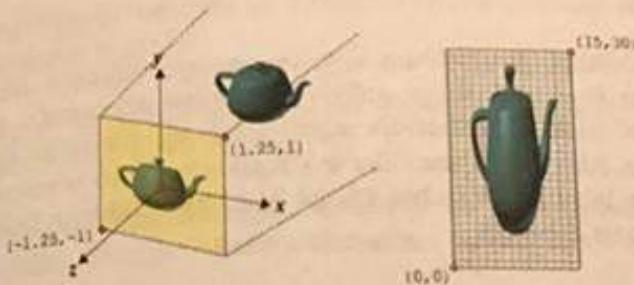
$$y_w = (y_{ndc} + 1) \left(\frac{height}{2} \right) + y$$

pri čemu su $x_{ndc} \in [-1, +1]$, $y_{ndc} \in [-1, +1]$, $x_w \in [x, x+width]$, a $y_w \in [y, y+height]$.

Prilikom definisanja širine i visine slike, potrebno je uskladiti ove vrednosti sa *aspect* parametrom funkcije *gluPerspective()*, tj. odnosom (right-left)/(top-bottom) kod funkcija *glFrustum()* i *glOrtho()*. Ukoliko vrednosti nisu uskladene dolazi do distorzije slike (slika 3.19).

Čitava pret
uspšte ne
dovoljno s
odgovaraju
oblika p
CGLRend
klijentsku

Rekli sm
projekcije



Slika 3.19. Uticaj promene aspekta na distorziju slike

Čitava prethodna priča može da deluje prilično komplikovano, ali na sreću u većini primena uopšte ne moramo da znamo ceo proces mapiranja 3D scene na prozor. Najčešće je dovoljno samo očitati veličinu prozora i postaviti parametre funkcije `glViewport()` na odgovarajuće vrednosti. Obzirom da se mapiranje menja samo prilikom promene veličine i oblika prozora, idealno mesto za poziv funkcije `glViewport()` je funkcija `CGLRenderer::Reshape()`. U narednom primeru prikazano je mapiranje scene na čitavu klijentsku površinu prozora.

```
void CGLRenderer::Reshape(CDC *pDC, int w, int h)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    double aspect = (double)w / (double)h;
    // definisanje projekciji ...
    //-----
    wglMakeCurrent(NULL, NULL);
}
```

Rekli smo već da je funkcija `CGLRenderer::Reshape()` idealna i za postavljanje projekcije. Sledi kompletan primer izgleda ove funkcije za slučaj perspektivne projekcije.

```
void CGLRenderer::Reshape(CDC *pDC, int w, int h)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    double aspect = (double)w / (double)h;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, aspect, 0.1, 100);
    glMatrixMode(GL_MODELVIEW);
    //-----
    wglMakeCurrent(NULL, NULL);
}
```

Z-bafer

Dubinski bafer, ili Z-bafer, je struktura koja omogućuje efikasno otkrivanje da li neki objekat zaklanja drugi objekat, tj. šta je vidljivo sa pozicije posmatrača, a šta nije. Najčešće je implementiran hardverski i predstavlja matricu čije su dimenzije odredene veličinom slike koja se prikazuje. Ako je veličina slike $w \times h$ piksela, toliko je potrebno i lokacija u Z-bafetu. Veličina jedne lokacije može biti 16, 24 ili 32 bita. Od toga koliko je velika jedna lokacija zavisi preciznost Z-bafera.

Nakon svih geometrijskih transformacija, vrši se rasterizacija primitiva, i one se pretvaraju u fragmente. Fragmenti su tačke koje imaju boju, koordinate, ali i neke druge atribute. Neki od fragmenata, koji nisu pokriveni drugim fragmentima, biće u vidu piksela prikazani na ekranu. Pre nego što se neki fragment pretvori u piksel, potrebno je odrediti njegovu udaljenost od posmatrača. Z-koordinata fragmenta je ta udaljenost, i ona se kreće od 0,0, ako se fragment nalazi na prednjoj ravni odsecanja, do 1,0, ako se fragment nalazi na zadnjoj ravni odsecanja. Z-koordinata se množi sa najvećim celim brojem koji može da se upiše u jednu lokaciju Z-bafera ($2^{16}-1$, $2^{24}-1$, ili $2^{32}-1$), a zatim se proverava da li na datom mestu u baferu (koje odgovara lokaciji piksela na ekranu) postoji vrednost koja je veća od date vrednosti. Ako jeste, to znači da je trenutno razmatrani fragment bliži oku od svih prethodnih, on se pretvara u piksel i upisuje u odgovarajući bafer za prikaz (framebuffer), a njegova udaljenost se upisuje u Z-bafer. Ako je vrednost u Z-bafetu manja od tekućeg, tekući fragment se odbacuje (jer se nalazi iza nekog drugog, koji je već upisan).

Da bismo omogućili Z-bafer, potrebno je aktivirati „proveru dubine“, pozivom funkcije:

```
glEnable(GL_DEPTH_TEST);
```

odrediti vrednost kojom se briše Z-bafer, pozivom funkcije:

```
void glClearDepth(GLclampd depth);
```

a pre svakog iscrtavanja scene obrisati Z-bafer:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Očigledno je da se prve dve funkcije pozivaju samo jednom, pa je najbolje njihove pozive smestiti u okviru funkcije **CGLRenderer::PrepareScene()**, dok se poziv treće funkcije mora naći u okviru funkcije **CGLRenderer::DrawScene()**, na samom početku, odmah nakon aktiviranja OpenGL rendering context-a.

Sada ćemo pokazati da preciznost Z-bafera veoma zavisi od načina na koji je definisana projekcija. Ako sa (x_e, y_e, z_e, w_e) označimo koordinate temena u sistemu vezanom za oko posmatrača (eye), nakon primene projekcije definisane pozivom **glFrustum(l, r, b, t, n, f)** dobijaju se *clip* koordinate, pri čemu je:

$$z_c = -z_e * (f+n)/(f-n) - w_e * 2*f*n/(f-n)$$

$$w_c = -z_e$$

Nakon perspektivnog deljenja, dobijaju se normalizovane koordinate uređaja. Nas trenutno zanima samo Z-koordinata, a ona je:

$$\begin{aligned} z_{\text{obj}} &= z_e / w_e = [-z_e \cdot (f+n) / (f-n) - w_e \cdot 2 \cdot f \cdot n / (f-n)] / -z_e \\ &= (f+n) / (f-n) + (w_e / z_e) \cdot 2 \cdot f \cdot n / (f-n) \end{aligned}$$

Nakon toga sledi **viewport** transformacija i množenje sa maksimalnim celim brojem koji može da se smesti u jednu lokaciju Z-bafera. Označimo taj broj sa s , i njegova vrednost je

$$s = 2^k - 1$$

gde je k preciznost Z-bafera (16, 24 ili 32). U koordinatnom sistemu prozora (*window*), Z-koordinata fragmenta računa se po formuli:

$$z_w = s \cdot [(w_e / z_e) \cdot f \cdot n / (f-n) + 0.5 \cdot (f+n) / (f-n) + 0.5]$$

Ako prethodni izraz preuređimo, tako da dobijemo zavisnost z_e/w_e u funkciji ostalih parametara, dobija se:

$$\begin{aligned} z_e / w_e &= f \cdot n / (f-n) / ((z_w / s) - 0.5 \cdot (f+n) / (f-n) - 0.5) \\ &= f \cdot n / ((z_w / s) \cdot (f-n) - 0.5 \cdot (f+n) - 0.5 \cdot (f-n)) \\ &= f \cdot n / ((z_w / s) \cdot (f-n) - f) \end{aligned}$$

Pogledajmo sada koje vrednosti ima z_e/w_e za tačke na prednjoj i zadnjoj ravni odsecanja:

$$\begin{aligned} z_w = 0 &\Rightarrow z_e / w_e = f \cdot n / (-f) = -n \\ z_w = s-1 &\Rightarrow z_e / w_e = f \cdot n / ((f-n) - f) = -f \end{aligned}$$

Obzirom da se kvantizacija obavlja na nivou celih brojeva, susedne dubine u Z-baferu su za dva prethodna slučaja 1 i $s-1$, respektivno.

$$\begin{aligned} z_w = 1 &\Rightarrow z_e / w_e = f \cdot n / ((1/s) \cdot (f-n) - f) \\ z_w = s-1 &\Rightarrow z_e / w_e = f \cdot n / (((s-1)/s) \cdot (f-n) - f) \end{aligned}$$

Ako prepostavimo da je $n = 0.01$, $f = 1000$, $s = 65535$ (tj. 16-to bitni Z-bafer) i zamenimo vrednosti u formule, dobićemo:

$$\begin{aligned} z_w = 1 &\Rightarrow z_e / w_e = -0.01000015 \\ z_w = s-1 &\Rightarrow z_e / w_e = -395.90054 \end{aligned}$$

Na osnovu prethodnog izračunavanja može se videti da Z-bafer nije linearan. Mnogo je veća „gustina“, tj. preciznost odmah iza prednje ravni odsecanja. U našem primeru, ukoliko se Z-koordinate fragmenata koji su veoma blizu prednje ravni odsecanja razlikuju za više od 0.01, biće tačno određeno koji se fragment nalazi ispred drugog. Ukoliko se Z-koordinate razlikuju za manje od 0.01, doći će do greške u izračunavanju, i može se fragment koji je dalje od oka iscrtati ispred onog koji je bliže.

Preciznost se sve više pogoršava kako prilazimo zadnjoj ravni odsecanja. U našem primjeru, dva fragmenta čije se Z-vrednosti razlikuju za skoro 400 jedinica, a nalaze se u zadnjoj trećini vidljive oblasti, biće smatrani da se nalaze na istom rastojanju od posmatrača. Ako su jedinice metri, to bi moglo značiti da se objekat koji je na 1km od posmatrača, zbog male preciznosti Z-bafera i loše postavljenih parametara projekcije vidi ispred objekta koji je na udaljenosti od 600m. Slične greške se javljaju i na manjim udaljenostima, ali sve više opadaju kako se približavamo prednjoj ravni odsecanja.

Zbog svega navedenog potrebno je prednju ravan odsecanja postaviti što dalje od posmatrača, ali tako da se vide svi objekti koji su bitni u sceni, a zadnja ravan odsecanja što bliže. Što je manji odnos **far/near**, to će preciznost biti veća.

Pregled transformacija

Do sada smo videli samo kako izgleda transformacija temena, ali kompletan proces formiranja slike na ekranu na osnovu ulaznih primitiva je mnogo komplikovaniji. Na kraju ove glave pokušaćemo da prikažemo čitav ovaj proces, odnosno najvažnije stepene (faze) u 3D grafičkom protočnom sistemu.

Čitav 3D grafički podsistem, radi lakše analize, podelićemo u sedam stepena, i to:

1. Transformacije temena
2. Sklapanje primitiva
3. Operacije nad primitivama
4. Rasterizacija
5. Transformacije fragmenata
6. Operacije nad fragmentima
7. Upis u framebuffer

Prvi stepen u protočnom sistemu je **transformacija temena**. U ovom stepenu vrši se:

- transformacija prostornih koordinata temena množenjem sa **modelview** i **projekcionom** matricom, respektivno,
- normale se množe inverznom matricom transponovane gornje leve 3×3 podmatrice **modelview** matrice,
- teksturne koordinate se transformišu (množe) **teksturnom** maticom (u ovom stepenu teksturne koordinate mogu biti i automatski generisane),
- primenjuju se izabrani materijal i definisani model osvetljenja kako bi se modifikovala osnovna boja temena i
- definiše se veličina tačaka.

Drugi stepen u protočnom sistemu je **sklapanje primitiva**. Zadatak ovog stepena je da na osnovu prosledenog tipa (GL_POINTS, GL_LINES, GL_TRIANGLES, itd.) nezavistno transformisana temena povezuje u primitive, koje se zatim prosleđuju sledećem stepenu. Mnoge operacije sledećeg stepena značajno zavise od toga kakve su primitive u pitanju.

Treći stepen, **operacije nad primitivama**, kao što i samo ime kaže, sastoji se od više zasebnih operacija. Prva operacija koja se izvršava u okviru ovog stepena je **odsecanje** (eng. *clipping*). Zadatak **odsecanja** je da ograniči primitive koje se prosleđuju dalje samo

OPENGL

na one koje
prostora.
odbacuju,
se samo u
oblasti. i

Druga op
zadate u
deljenje
na norma
su sada z
l.

Treća op
reći u pi
toga kak
culling,
glEnable
od para
prethodi

Četvrti
primitiv
Za razli
okviru
ekranske
fragmen
tip primit

Fragme
transfe
„mapir
slici),
boje fr
ovom
posma
sekund
sekund

Nakon
opera

na one koje se „vide“, odnosno, na „vidljive delove“ primitiva koje su na rubu vidljivog prostora. Sve primitive koje se u potpunosti nalaze van vidljive oblasti jednostavno se odbacuju, a primitive koje su u potpunosti unutar vidljive oblasti se prosledjuju dalje. Ako se samo deo primitive nalazi u vidljivoj oblasti, izračunava se presek primitive i vidljive oblasti, i na tom mestu dodaju nova temena.

Druga operacija nad primitivama je **perspektivno deljenje**. Sve koordinate su još uvek zadate u homogenom koordinatnom sistemu sa četiri komponente: x, y, z i w. Ovde se vrši deljenje prve tri komponente (x, y i z) sa w, i prelazi se sa koordinata odsecanja (eng. *clip*) na normalizovane koordinate uređaja (eng. *normalized device coordinates*). Sve koordinate su sada zadate samo sa tri komponente (x, y i z), a vrednosti se nalaze u granicama od -1 do 1.

Treća operacija nad primitivama je **viewport transformacija**. O njoj je već detaljno bilo reči u prethodnom poglavlju. Četvrta operacija vrši uklanjanje primitiva u zavisnosti od toga kako su okrenute prema posmatraču. Ova operacija poznata je pod engleskim nazivom *culling*, i opcionala je. Da bi se izvršila potrebno je omogućiti je pozivom funkcije `glEnable(GL_CULL_FACE)`, a koja strana poligona će biti smatrana za nevidljivu zavisi od parametra koji je prosledjen funkciji `glCullFace()`. O ovome je bilo više reči u prethodnoj glavi.

Četvrti stepen u protočnom sistemu je **rasterizacija**. Rasterizacija pretvara geometrijske primitive u fragmente. Fragmenti su „kandidati“ za piksele koji će biti prikazani na ekranu. Za razliku od piksela, koji imaju samo boju i 2D ekranske koordinate (tačnije, koordinate u okviru prozora), fragment ima: boju, teksturne koordinate, 3D koordinate (osim 2D ekranskih, poseduje i treću koordinatu koja definiše „dubinu“), itd. Vrednosti atributa fragmenata izračunavaju se interpolacijom odgovarajućih vrednosti atributa temena. Svaki tip primitiva ima svoja (drugačija) pravila za rasterizaciju.

Fragmenti se zatim prosledjuju sledećem stepenu u protočnom sistemu, koji vrši transformaciju **fragmenata**. Jedna od najvažnijih operacija koja se vrši u ovom stepenu je „*mapiranje teksture*“. Na osnovu teksturnih koordinata vrši se pristup teksturi (najčešće 2D slici), a zatim se na osnovu postavljenih parametara primene teksture vrši modifikovanje boje fragmenta. Teksture su detaljno obradene u petom poglavlju. Osim primene tekstura, u ovom stepenu se boja fragmenata modifikuje primenom magle (na osnovu udaljenosti od posmatrača, boja fragmenta se meša sa bojom magle) i kombinacijom primarne i sekundarne boje (da primena tekstura ne bi umanjila efekat osvetljenja, uvedena je i sekundarna boja koja se primenjuje tek nakon primene tekstura).

Nakon toga, fragmenti se prosledjuju sledećem stepenu, koji izvršava neke elementarne operacije nad **fragmentima**, i to:

- test pripadnosti piksela – utvrđuje da li je odredišni piksel vidljiv ili prekriven nekim drugim prozorom,
- test „makaza“ – vrši odbacivanje fragmenata ukoliko ne pripadaju oblasti definisanoj funkcijom `glScissor()`,
- test providnosti – određuje da li će fragment biti odbačen na osnovu *alpha* vrednosti (četvrta koordinata boje) i parametra prosledjenog funkciji `glAlphaFunc()`,

- test „zaklona“ – upoređuje vrednosti fragmenta sa vrednostima upisanim u stencil bafer, korišćenjem funkcija `glStencilFunc()` i `glStencilOp()`, i na osnovu toga određuje da li će biti odbačen ili ne (*stencil* bafer se koristi za pravljenje vrlo složenih maski, npr. definisanje unutrašnjosti automobila ili aviona dok se kroz prozor datog vozila prikazuju 3D scene, ali se može koristiti i za druge efekte, npr. senke i sl.),
- test dubine – korišćenjem parametra poređenja definisanog funkcijom `glDepthFunc()`, vrši poređenje „dubine“ fragmenta (tj. z-koordinate) sa dubinom zapisanom u odgovarajućem baferu (najčešće je poznat pod nazivom *depth*-bafer ili Z-bafer),
- mešanje boje fragmenta sa bojom koja je već smeštena u *framebuffer*, korišćenjem parametara definisanih funkcijama: `glBlendFunc()`, `glBlendColor()`, i `glBlendEquation()` i
- primena logičkih operacija definisanih funkcijom `glLogicOp`.

Sve operacije iz ovog stepena su vrlo jednostavne i u potpunosti hardverski implementirane, tako da se u jednoj sekundi mogu obraditi na milioni fragmenata. Fragmenti koji su „preživeli“ sve prethodne testove i transformacije bivaju upisani u *framebuffer* i vidimo ih na ekranu u okviru finalne slike.

Stepeni 1 i 5 (transformacija temena i fragmenata) počev od OpenGL-a 2.0 postaju programibilni. Programi koji upravljaju ovim stepenima nazivaju se – *shader*-i. Zaključno sa OpenGL-om 3.1 postojala su dva tipa *shader*-a:

- *vertex shader* (koji upravlja transformacijom temena) i
- *fragment shader* (koji upravlja transformacijom fragmenata).

Od OpenGL-a 3.2 dodat je i novi tip *shader*-a (istovremeno i novi stepen u protočnom sistemu, koji se nalazi odmah iza transformacije temena) nazvan *geometry shader*. On omogućuje dodavanje novih temena u geometriji, a time i upravljanje složenošću objekata.

Kratak pregled gradiva

Da bi dobili bojenu sliku na ekranu potrebno je:

- definisati objekte korišćenjem grafičkih primitiva u lokalnom koordinatnom sistemu,
- primeniti geometrijske transformacije `glTranslate()`/`glRotate()`/`glScale()` da bi se objekat postavio na odgovarajuće mesto u sceni, tj. da bi se izvršila transformacija modeliranja,
- ako je objekat složen i zahteva hijerarhijsku organizaciju transformacija koristiti `glPushMatrix()` i `glPopMatrix()` da bi se upravljalo magacinom *modelview* matrica,
- definisati transformaciju pogleda kao `glTranslate()`/`glRotate()` transformacije koje utiču na čitavu scenu, ili pozivom funkcije `gluLookAt()`,
- definisati projekciju jednom od funkcija: `glOrtho()`, za ortografsku, ili `glFrustum()` i `gluPerspective()`, za perspektivnu projekciju,
- definisati poziciju i veličinu rezultujuće slike u okviru prozora pozivom funkcije `glViewport()`.

Funkcije korišćene u ovom poglavlju

void glMatrixMode(GLenum mode);

mode – magacin matrica koji se selektuje, može imati jednu od sledećih vrednosti:

GL_MODELVIEW,
GL_PROJECTION i
GL_TEXTURE

Selektuje tekući magacin matrica.

void glLoadMatrix{fd}(const TYPE *m);

m – pokazivač na matricu dimezija 4×4 smeštene u vektor po kolonama

Zamenjuje matricu na vrhu tekućeg magacina matrica prosleđenom matricom.

void glLoadIdentity();

Zamenjuje matricu na vrhu tekućeg magacina matrica jediničnom matricom.

void glMultMatrix{fd}(const TYPE *m);

m – pokazivač na matricu dimezija 4×4 smeštene u vektor po kolonama

Množi matricu na vrhu tekućeg magacina matrica (C) prosleđenom matricom (M) sa desne strane. $C = CM$

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

x, y, z – koordinate vektora translacije

Pomera koordinatni početak lokalnog koordinatnog sistema u tačku (*x, y, z*).

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

angle – ugao rotacije zadat u stepenima

x, y, z – koordinate vektora translacije

Vrši rotaciju u smeru suprotnom od kazaljki na časovniku za ugao *angle* oko vektora [*x, y, z*].

`void glScalef(fd)(TYPE x, TYPE y, TYPE z);`

x, y, z – faktori skaliranja po X, Y i Z-osi, respektivno

Vrši skaliranje po X, Y i Z-osi za zadate vrednosti.

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

cap – stanje koje se uključuje/isključuje. Može imati preko 40 različitih vrednosti, ne računajući podoblike. U ovom poglavlju koriste se samo 3:

GL_NORMALIZE – uključuje/isključuje automatsku normalizaciju vektora normala

GL_DEPTH_TEST – uključuje/isključuje poređenje „dubine“ fragmenata i ažuriranje Z-bafera

GL_CULL_FACE – uključuje/isključuje odbacivanje primitiva u zavisnosti od orientacije

Ovaj par funkcija uključuje/isključuje odgovarajuće stanje u OpenGL rendering context-u

`void glPushMatrix() / void glPopMatrix()`

Ovaj par funkcija menja stanje selektovanog magacina matrica. Funkcija glPushMatrix() kopira matricu na vrhu i smešta je u magacin. Nakon poziva ove funkcije i na vrhu magacina i ispod njega nalaze se iste matrice. Funkcija glPopMatrix() uklanja matricu sa vrha magacina.

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

eyex, eyey, eyez – pozicija oka posmatrača

centerx, centery, centerz – tačka u koju je usmeren pogled

upx, upy, upz – vektor vertikale

Definiše transformaciju pogleda.

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

left, right – koordinate leve i desne vertikalne ravni odsecanja

bottom, top – koordinate donje i gornje horizontalne ravni odsecanja

near, far – udaljenost prednje i zadnje ravni odsecanja od posmatrača

Definiše ortografsku projekciju.

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

left, right – x-koordinate leve i desne ravni odsecanja na preseku sa prednjom ravni odsecanja
bottom, top – y-koordinate donje i gornje ravni odsecanja na preseku sa prednjom ravni odsecanja
near, far – udaljenost prednje i zadnje ravni odsecanja od posmatrača

Definiše perspektivnu projekciju.

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

fovy – ugao vidnog polja u YZ-ravni izražen u stepenima
aspect – odnos širine i visine vidnog polja
near, far – udaljenost prednje i zadnje ravni odsecanja od posmatrača

Definiše perspektivnu projekciju.

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

x, y – koordinate donjeg levog temena pravougaonika u kome se prikazuje slika
width, height – širina i visina pravougaonika u kome se prikazuje slika

Definiše **viewport** transformaciju, tj. deo prozora u kome se prikazuje slika.
Koordinata (0,0) odgovara donjem levom uglu prozora.

```
void glClearDepth( GLclampd depth );
```

depth – vrednost koja se koristi za „čišćenje“ Z-bafera

Definiše vrednost iz intervala [0, 1] koja se upisuje u sve lokacije Z-bafera prilikom poziva funkcije **glClear()** ukoliko je postavljen odgovarajući flag.

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

mask – polje flagova (indikatora) koji sve bafere trebaju biti obrisani pri pozivu ove funkcije. Koristi se bitska „ili“ kombinacija sledećih predefinisanih vrednosti:

GL_COLOR_BUFFER_BIT – bfer u kome se formira slika

za prikaz (*framebuffer*)

GL_DEPTH_BUFFER_BIT – Z-bfer

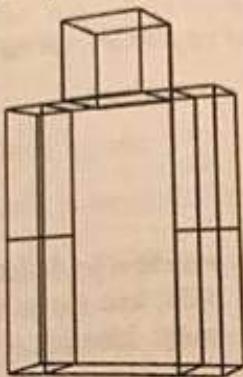
Briše specifičirane bafere, odnosno postavlja vrednosti definisane odgovarajućim funkcijama. Bfer slike ispunjuje se bojom definisanom funkcijom **glClearColor()**, a Z-bfer vrednošću postavljenom funkcijom **glClearDepth()**.

Zadatak

Napisati funkciju **CGLRenderer::DrawRobot()**, koja crta robota. Za crtanje svih delova robova koristiti samo jedinične kocke (pozivom funkcije **DrawCube(1.0)**, koja je definisana u prethodnom poglavljju). Pojedini delovi tela dobijaju se primenom odgovarajućih transformacija (skaliranje/translacija/rotacija) na jedinične kocke. Robota nacrtati kao član model sa sledećim dimenzijama delova:

- trup - $1.0 \times 2.0 \times 0.5$
- nadlaktice i podlaktice - $0.3 \times 1.0 \times 0.5$
- glava - $0.6 \times 0.6 \times 0.6$

Inicijalni izgled robota, sa međusobnim položajem svih delova, prikazan je na slici 3.20.



Slika 3.20. Inicijalni izgled robota

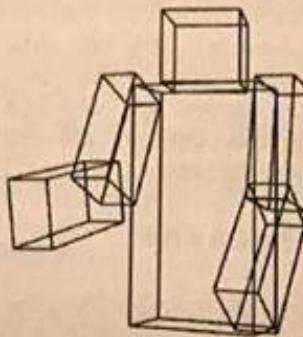
U klasi **CGLView** dodati funkciju koja će reagovati na pritisak tastera na tastaturi, odnosno rukovati porukom **WM_KEYDOWN**. Ova funkcija treba da pozove funkciju **CGLRenderer::OnKeyDown(UINT nChar)** i prosledi joj pritisnuti taster.

Funkcija **CGLRenderer::OnKeyDown()** treba da izvrši sledeće operacije:

- ako je pritisnut taster Q da rotira telo robota u levo,
- ako je pritisnut taster W da rotira telo robota u desno,
- ako je pritisnut taster E da rotira glavu robota u levo,
- ako je pritisnut taster R da rotira glavu robota u desno,
- ako je pritisnut taster A da rotira levu ruku robota u ramenu naviše,
- ako je pritisnut taster Z da rotira levu ruku robota u ramenu naniže,
- ako je pritisnut taster S da rotira desnu ruku robota u ramenu naviše,
- ako je pritisnut taster X da rotira desnu ruku robota u ramenu naniže,
- ako je pritisnut taster D da rotira levu ruku robota u laktu naviše,
- ako je pritisnut taster C da rotira levu ruku robota u laktu naniže,
- ako je pritisnut taster F da rotira desnu ruku robota u laktu naviše,
- ako je pritisnut taster V da rotira desnu ruku robota u laktu naniže.

Konak rotacije neka bude 5° .

Na slici 3.21 prikazan je robot sa pomerenim rukama i glavom. Pri definisanju rotacije nije uslovdno voditi računa o ograničavanju uglova do kojih mogu da se rotiraju ruke robota.



Slika 3.21. Robot sa pomerenim rukama i glavom

Rešenje

Napomena: Funkciju **OnKeyDown()** potrebno je dodati korišćenjem *ClassWizard-a* u VS 6.0, odnosno *Properties* panela u VS 2008, kao što je to prikazano u prvoj glavi. U ovom rešenju nisu prikazane funkcije koje imaju identičnu implementaciju kao u prethodnim glavama.

```
// GLView.cpp -----
void CGLView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    m_glRenderer.OnKeyDown(nChar);
    Invalidate(FALSE);

    CView::OnKeyDown(nChar, nRepCnt, nFlags);
}

// GLRenderer.h -----
#pragma once

class CGLRenderer
{
public:
    CGLRenderer(void);
    virtual ~CGLRenderer(void);

    bool CreateGLContext(CDC* pDC);
    void PrepareScene(CDC* pDC);
    void Reshape(CDC* pDC, int w, int h);
    void DrawScene(CDC* pDC);
    void DestroyScene(CDC* pDC);

    void OnKeyDown(UINT nChar);
    void DrawCube(double dSize);

protected:
    HGLRC    m_hrc; //OpenGL Rendering Context
```

```
void DrawRobot( double ugaoTrupa, double ugaoGlave,
                double ugaoLRamena, double ugaoDRamena,
                double ugaoLLakta, double ugaoDLakta );

// Ulazovi rotacije
double m_ugaoTrupa, m_ugaoGlave;
double m_ugaoLRamena, m_ugaoDRamena;
double m_ugaoLLakta, m_ugaoDLakta;

//-----  
// cLrenderer.cpp -----  
/* inicijeni deo koda */
CGLrenderer::CGLrenderer(void)
{
    m_ugaoTrupa = m_ugaoGlave = m_ugaoLRamena = m_ugaoDRamena =
    m_ugaoLLakta = m_ugaoDLakta = 0.0;
}

/* inicijeni deo koda */
void CGLrenderer::PrepareScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClearColor (1.0, 1.0, 1.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    glLineWidth(2.0);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    //-----
    wglMakeCurrent(NULL, NULL);
}

void CGLrenderer::DrawScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity ();

    // Transformacija pogleda
    glTranslatef(0.0, 0.0, -7.0);

    glColor3f(0.0,0.0,0.0);
    DrawRobot(m_ugaoTrupa, m_ugaoGlave,
              m_ugaoLRamena, m_ugaoDRamena,
              m_ugaoLLakta, m_ugaoDLakta);
    //-----
    glFlush();
    SwapBuffers(pDC->m_hDC);
    wglMakeCurrent(NULL, NULL);
}

void CGLrenderer::Reshape(CDC *pDC, int w, int h)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective(45,(double)w/(double)h,1,100);
    glMatrixMode (GL_MODELVIEW);
```

```

// wglGetCurrent(NULL, NULL);
}

void CGLRenderer::DrawRobot(double ugaoTrupa, double ugaoGlave,
                            double ugaoLRamena, double ugaoDRamena,
                            double ugaoLLakta, double ugaoDLakta)
{
    glPushMatrix();
    glRotatef(ugaoTrupa, 0.0, 1.0, 0.0);
    // Trup
    glPushMatrix();
    glScalef(1.0, 2.0, 0.5);
    DrawCube(1.0);
    glPopMatrix();

    // Desna nadlaktica
    glPushMatrix();
    glTranslatef(0.65, 1.0, 0.0);
    glRotatef(-ugaoDRamena, 1.0, 0.0, 0.0);
    glTranslatef(0.0, -0.5, 0.0);
    glPushMatrix();
    glScalef(0.3, 1.0, 0.5);
    DrawCube(1.0);
    glPopMatrix();

    // Desna podlaktica
    glTranslatef(0.0, -0.5, 0.0);
    glRotatef(-ugaoDLakta, 1.0, 0.0, 0.0);
    glTranslatef(0.0, -0.5, 0.0);
    glPushMatrix();
    glScalef(0.3, 1.0, 0.5);
    DrawCube(1.0);
    glPopMatrix();
    glPopMatrix();

    // Leva nadlaktica
    glPushMatrix();
    glTranslatef(-0.65, 1.0, 0.0);
    glRotatef(-ugaoLRamena, 1.0, 0.0, 0.0);
    glTranslatef(0.0, -0.5, 0.0);
    glPushMatrix();
    glScalef(0.3, 1.0, 0.5);
    DrawCube(1.0);
    glPopMatrix();

    // Leva podlaktica
    glTranslatef(0.0, -0.5, 0.0);
    glRotatef(-ugaoLLakta, 1.0, 0.0, 0.0);
    glTranslatef(0.0, -0.5, 0.0);
    glPushMatrix();
    glScalef(0.3, 1.0, 0.5);
    DrawCube(1.0);
    glPopMatrix();
    glPopMatrix();

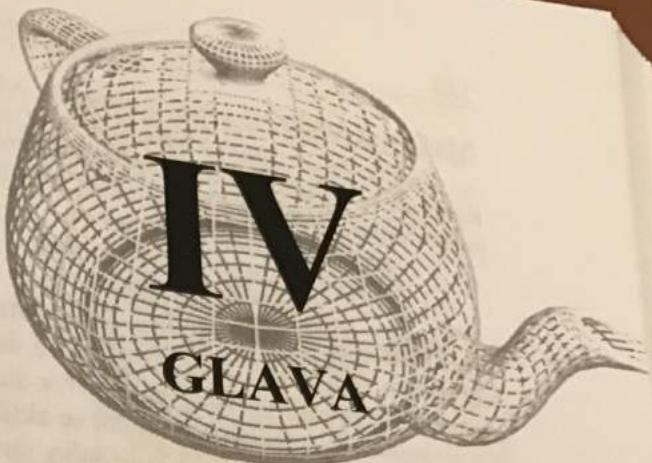
    // Glava
    glPushMatrix();
    glTranslatef(0.0, 1.3, 0.0);
    glRotatef(ugaoGlave, 0.0, 1.0, 0.0);
    glScalef(0.6, 0.6, 0.6);
    DrawCube(1.0);
}

```

OPENGL - FIKSNA FUNKCIONALNOST

```
glPopMatrix();
glPopMatrix();

void CGLRenderer::OnKeyDown(UINT nChar)
{
    if(nChar == 'Q')
        m_ugaoTrupa += 5.0;
    else if(nChar == 'W')
        m_ugaoTrupa -= 5.0;
    else if(nChar == 'E')
        m_ugaoGlave += 5.0;
    else if(nChar == 'R')
        m_ugaoGlave -= 5.0;
    else if(nChar == 'A')
        m_ugaoDRamena += 5.0;
    else if(nChar == 'Z')
        m_ugaoDRamena -= 5.0;
    else if(nChar == 'S')
        m_ugaoLRamena += 5.0;
    else if(nChar == 'X')
        m_ugaoLRamena -= 5.0;
    else if(nChar == 'D')
        m_ugaodLakta += 5.0;
    else if(nChar == 'C')
        m_ugaodLakta -= 5.0;
    else if(nChar == 'F')
        m_ugaoLLakta += 5.0;
    else if(nChar == 'V')
        m_ugaoLLakta -= 5.0;
}
```

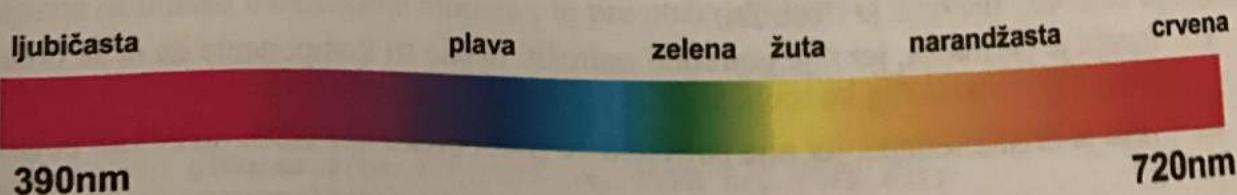


Osvetljenje

U prethodnoj glavi naučili smo kako da postavimo 3D scenu, ali je ona bila prilično „beživotna“. Objekti su jednobojni, nema sjajnih niti tamnih površina. Zato ćemo u ovoj glavi povećati realističnost naše scene uvođenjem osvetljenja.

Boje

Pre uvođenja osvetljenja, razmotrimo kako ljudi doživljavaju boju predmeta koji ih okružuju. Boja je zapravo osećaj koju svetlosna energija izaziva na mrežnjači (*retina*), koji se zatim prosleđuje mozgu na interpretaciju. Mozak, dakle, formira sliku na osnovu mešavine fotona različitih frekvencija, odnosno talasnih dužina. Ljudsko oko poseduje 3 tipa konusnih (kupastih) ćelija koje imaju maksimalnu osetljivost na svetlost određene talasne dužine. Jedne su najosetljivije na svetlost plave boje, druge na svetlost zelene, a treće na svetlost crvene boje. Ovakva građa ljudskog oka uticala je i na model boja. Zato se danas vrlo često koristi trokomponentni model – RGB. RGB je skraćenica od engleskih reči *Red* (crvena), *Green* (zelena) i *Blue* (plava). Da je oko drugačije građe, sigurno bi i model bio sasvim drugačiji. Talasne dužine vidljive svetlosti kreću se u granicama od 390nm (ljubičasta) do 720nm (crvena).



Slika 4.1. Talasne dužine vidljive svetlosti

Mešanje boja

Kako na osnovu samo tri komponente možemo dobiti sve ostale boje? Odgovor je vrlo jednostavan – mešanjem. Generalno, postoje dva modela:

- svetlosni (aditivni) model i
- pigmentni (subtraktivni) model.

Kod **svetlosnog modela**, boju doživljavamo kao energiju svetlosnog izvora. Ako izvor ima maksimum u crvenom delu spektra, mi ćemo ga videti kao crveni izvor svetlosti, jer će konusne ćelije, koje su najviše osetljive na talasnu dužinu koja odgovara crvenoj svetlosti, biti maksimalno pobuđene. Ukoliko se aktivira i zeleni izvor svetlosti, on će pobuditi drugu vrstu konusnih ćelija. Signali koje šalju „crvene“ i „zelene“ konusne ćelije biće „sabrani“, i mozak interpretira rezultat kao žutu svetlost. Maksimalna eksitacija svih konusnih ćelija stvara efekat bele svetlosti. Upravo zbog toga što se komponente ovog modela sabiraju, model je dobio naziv **aditivni**.

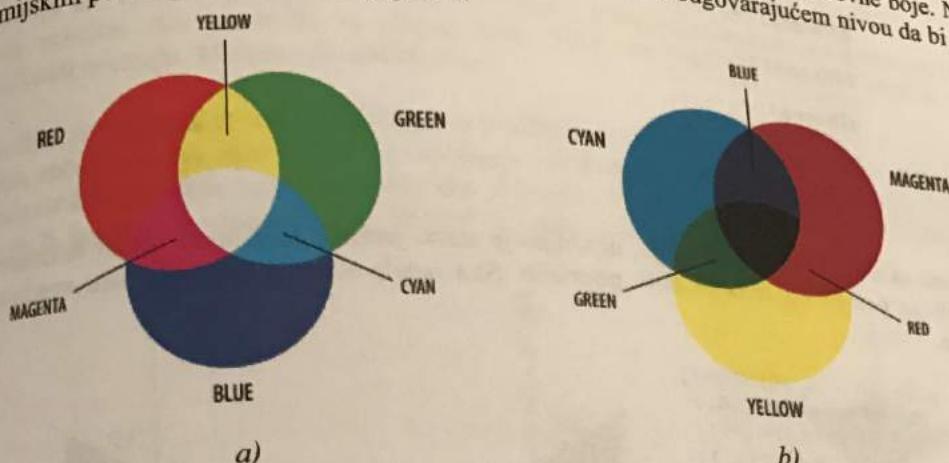
Pigmentni model doživljaj boje objekta definiše kao energiju svetlosti koja je reflektovana od datog objekta. Naime, svaki objekat izgrađen je od nekog materijala. Ukoliko je materijal neprovidan, na boju predmeta utiče talasna dužina svetlosti koja se reflektuje od površine. Ako predmet opažamo kao zeleni, to znači da je njegova površina apsorbovala fotone svih talasnih dužina, osim onih koji odgovaraju zelenoj boji. Reflektovani fotoni dolaze do našeg oka i utiču na opažaj boje.

Još od praistorije, setimo se pećina Lasko i Altamira, čovek je pokušavao umetnički da se iskaže, prikazujući svet koji ga okružuje. Prve površine koje su oslikavane bile su kamene. Kasnije je pronađen papirus i pergament. A najveća slikarska dela nastala su na platnu. Bilo gde da su crtane, slike su zahtevale nanošenje boja, tačnije pigmenata, na oslikanu površinu. Svaki od pigmenata „oduzimao“ je neku komponentu iz reflektovanog dela spektra i na taj način formirao doživljaj boje.

Koliko je minimalno pigmenata potrebno da bi se dobile sve boje, i koje boje oni predstavljaju kada su „čisti“. U osnovnoj školi učili smo da su osnovne boje: plava, crvena i žuta. Međutim, to nije tačno! Ako pomešamo ove tri boje, tj. pigmenta, oni bi trebalo da apsorbuju čitavu vidljivu svetlost i „mešavina“ bi trebalo da bude crne boje. Crna boja apsorbuje čitavu vidljivu svetlost, tj. fotone svih talasnih dužina. Ali, umesto crne, dobija se neka prljavo-braon boja. Ranije su umetnici ovu pojavu objašnjavali lošim mešanjem odgovarajućih pigmenata. Ali, ni to nije tačno. Razlog je što crvena, plava i žuta NISU primarne boje. Primarne boje pigmenata su: *Cyan* (plavo-zelena), *Magenta* (ljubičasta) i *Yellow* (žuta). Ovaj model poznat je kao CMY. Štampači najčešće koriste CMYB model. Poslednje slovo ukazuje na postojanje posebne komponente za crnu boju (*Black*). Razlog za postojanje zasebne crne boje je trostruki:

1. manja je potrošnja, jer nije potrebno potrošiti ostale tri komponente da bi se dobila crna,
2. veća je brzina štampe, jer nije potrebno $3 \times$ preći preko iste tačke različitim bojama i
3. ukoliko boje nisu pravi primari, njihovim mešanjem neće se dobiti crna boja.

Dobijanje čistih primarnih pigmenata je vrlo teško, jer priroda „ne voli“ osnovne boje. Na primer, tek početkom dvadesetog veka tehnologija je bila na odgovarajućem nivou da bi se hemijskim putem dobio čisto žuti pigment.



Slika 4.2. Mešanje boja: a) Svetlosni (aditivni) model, b) Pigmentni (subtraktivni) model

Dodavanje osvetljenja u scenu

Vratimo se na OpenGL. Da bi se formirala scena sa osvetljenjem, potrebno je uraditi sledeće:

- definisati normale u svakom od temena objekata,
- definisati svojstva materijala objekata u sceni,
- definisati model osvetljenja i
- kreirati, aktivirati i postaviti jedan ili više izvora svetlosti.

Definisanje normala

Da bi se moglo izračunati koliko je osvetljena neka površina objekta, potrebno je znati kako je ona orijentisana u odnosu na izvor svetlosti. Ako je površina okrenuta ka izvoru svetlosti, biće jako osvetljena, a ako je okrenuta od izvora, biće tamna. Orientacija površine određuje se vektorom normale u svakoj tački te površine.

Da ponovimo: u OpenGL-u sve parametre objekta možemo definisati samo u temenima, i sve ostaje na snazi od trenutka definisanja do promene. Imajući ovo u vidu, normale moramo definisati u temenima modela i to pre poziva `glVertex*0` funkcije.

Normala je trodimenzionalni vektor, i postavlja se jednom od `glNormal3*0` funkcija.

```
void glNormal3f(GLfloat nx, GLfloat ny, GLfloat nz);
void glNormal3fv(const GLfloat *v);
```

Parametri funkcije `glNormal3*0` su koordinate vektora normale. Ukoliko površina nije ravna, potrebno je izračunati normalu u svakom temenu i postaviti je na odgovarajuću vrednost, kao u sledećem primeru:

```

glBegin (GL_POLYGON);
    glNormal3fv(n0);
    glVertex3fv(v0);
    glNormal3fv(n1);
    glVertex3fv(v1);
    glNormal3fv(n2);
    glVertex3fv(v2);
    glNormal3fv(n3);
    glVertex3fv(v3);
glEnd();

```

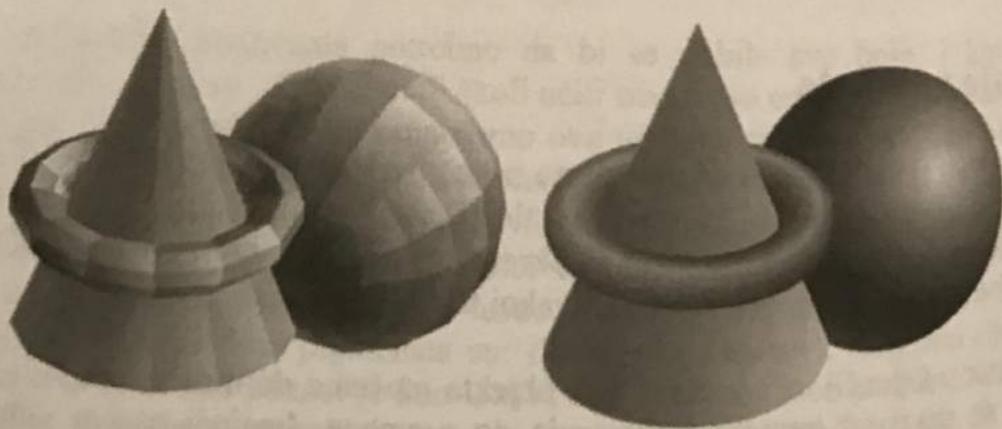
Međutim, ukoliko je površina ravna, dovoljno je samo jednom izračunati vektor normale i postaviti ga pre prvog temena date površine. Sva ostala temena preuzeće istu vrednost normale.

```

glBegin (GL_POLYGON);
    glNormal3fv(n0);
    glVertex3fv(v0);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glVertex3fv(v3);
glEnd();

```

Na slici 4.3 može se videti efekat normala na način senčenja objekata. Na slici 4.3.a sva temena koja čine jedan trougao ili četvorougao (objekat je izgrađen od mnoštva elementarnih trouglova ili četvorouglova) imaju istu orientaciju normale. To stvara efekat ravne površine, a objekti izgledaju „kockasto“. Na slici 4.3.b orijentacija normala u temenima jednog trougla ili četvorougla se razlikuju, ali se poklapaju za temena susednih primitiva. Naime, ako dva susedna trougla dele teme, deliće i normalu, pri čemu se normala dobija kao srednja vrednost normala susednih površina. Time se stvara efekat zaobljenosti površi.



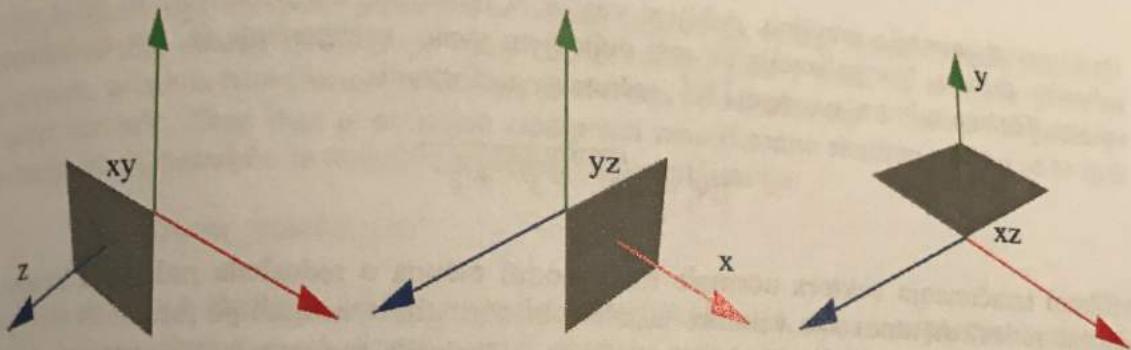
Slika 4.3. Primer primene normala u definisanju zaobljenosti figura:
a) oštре ivice, b) oble ivice

Izračunavanje normale

113

Funkcija `glNormal3*0` definiše parametre vektora normale, tačnije x, y i z komponentu ovog vektora. Ali, postavlja se pitanje kako dobiti taj vektor? OpenGL neće za nas izračunati normalu. Moramo to urediti sami.

Ukoliko je površina ravna, i ukoliko je paralelna nekoj od koordinatnih ravnih, onda čak i nema računanja. Na slici 4.4 može se jasno videti da je u tom slučaju normala zapravo jedna od koordinatnih osa. Na primer, ako je površ u ravni XY, normala na nju je paralelna Z-osi, tj. normala je vektor [0 0 1], ili [0 0 -1], zavisno od orientacije površi.



Slika 4.4. Normale ravnih površi paralelnih koordinatnim ravnima

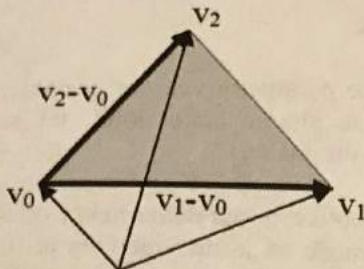
Ukoliko ravna površ nije paralelna koordinatnim ravnima, onda je izračunavanje neophodno. U opštem slučaju, najjednostavniji način za određivanje normale jeste računanje vektorskog proizvoda. Vektorski proizvod dva vektora je vektor koji je normalan na ravan koja sadrži prva dva vektora, a usmerenje se određuje po pravilu desne zavojnice. Potrebno je, dakle, odrediti dva vektora koji sigurno pripadaju površini koju osvetljavamo. Ako je ta površina trougao, a znamo da ivice trougla sigurno leže u ravni kojoj pripada trougao, onda vektori koji se poklapaju sa ivicama mogu da nam posluže za računanje normale. Neka su temena trougla V_0 , V_1 i V_2 . Znamo da su koordinate zapravo vektori položaja (V_0 je, dakle, vektor položaja odgovarajućeg temena u odnosu na koordinatni početak). Uzajamnim oduzimanjem ovih vektora dobijaju se vektori stranica, tj. ivica trougla (slika 4.5), pa se vektor normale može izračunati po formuli:

$$\mathbf{N} = [\mathbf{V}_1 - \mathbf{V}_0] \times [\mathbf{V}_2 - \mathbf{V}_0]$$

Ako znamo njihove koordinate, vektorski proizvod dva vektora $\mathbf{V}(v_x, v_y, v_z)$ i $\mathbf{W}(w_x, w_y, w_z)$, računa se po formuli:

$$\mathbf{V} \times \mathbf{W} = [v_x \ v_y \ v_z] \times [w_x \ w_y \ w_z] = [(v_y w_z - v_z w_y) \ (v_z w_x - v_x w_z) \ (v_x w_y - v_y w_x)]$$

Detaljnije o načinu izračunavanja vektorskog proizvoda, ali i drugim vektorskim operacijama, možete pročitati u dodatku A – Vektorska aritmetika.



Slika 4.5. Određivanje vektora koji pripadaju ravni trougla

Da bi osvetljenje bilo pravilno, dobijeni vektor \mathbf{N} mora biti normalizovan, tj. mora imati jediničnu dužinu. Normalizacija se vrši deljenjem svake komponente (x , y i z) dužinom vektora. Dužina vektora, u oznaci $|\mathbf{N}|$, računa se po formuli:

$$|\mathbf{N}| = \sqrt{x^2 + y^2 + z^2}$$

Prilikom izračunanja vektora normale treba voditi računa o redosledu zadavanja temena, odnosno redosledu množenja vektora. Ukoliko bi normalu izračunali po formuli $\mathbf{N} = [\mathbf{v}_2 - \mathbf{v}_0] \times [\mathbf{v}_1 - \mathbf{v}_0]$, a posmatrač vidi trougao na način prikazan na slici 4.5, trougao bi bio potpuno taman (uz pretpostavku da nije uključeno dvostrano osvetljenje), jer bi normala bila orientisana **od** posmatrača.

Ukoliko želimo odrediti normalu u nekoj tački zakrivljene površi koja je zadata analitički u eksplicitnom obliku $\mathbf{V}(s,t) = [X(s,t) \ Y(s,t) \ Z(s,t)]$, gde su X , Y i Z diferencijabilne funkcije po s i t , postupak je sledeći:

- najpre se odrede parcijalni izvodi po s i t , tj. $\partial\mathbf{V}/\partial s$ i $\partial\mathbf{V}/\partial t$,
- a zatim se izračuna vektorski proizvod ta dva parcijalna izvoda.

$$\mathbf{N} = \frac{\partial\mathbf{V}}{\partial s} \times \frac{\partial\mathbf{V}}{\partial t}$$

Parcijalni izvodi po s i t daju vektore tangentne na površ po s i t pravcu. Vektorski proizvod je upravan na ta dva vektora, pa time i na površ.

Na primer, ako je $\mathbf{V}(s,t) = [s^2 \ t^3 \ 3-st]$, tada je $\partial\mathbf{V}/\partial s = [2s \ 0 \ -t]$ i $\partial\mathbf{V}/\partial t = [0 \ 3t^2 \ -s]$, onda je vektor normale $\mathbf{N} = \partial\mathbf{V}/\partial s \times \partial\mathbf{V}/\partial t = [-3t^3 \ 2s^2 \ 6st^2]$. Za $s = 0.5$ i $t = 0.75$, tačka se nalazi na koordinatama $(0.25, 0.421875, 2.625)$. Vektor normale dobijen prethodnim izrazom je $\mathbf{N} = (-1.265625, 0.125, 1.6875)$, a dužina ovog vektora je $|\mathbf{N}| = 4.465087890625$. Dakle, nije jedinični vektor. Zato moramo da izvršimo normalizaciju, nakon koje dobijamo:

$$\mathbf{N}_{\text{norm}} = \mathbf{N} / |\mathbf{N}| = (-0.283449, 0.027995, 0.377932)$$

Ako je zakrivljena površ zadata u implicitnom obliku $\mathbf{F}(x, y, z) = 0$, problem je mnogo teži. Ukoliko je moguće izraziti jednu koordinatu preko ostale dve, npr $z = G(x, y)$, onda se problem svodi na prethodni, obzirom da je $\mathbf{V}(s,t) = [s \ t \ G(s,t)]$. Ako nije moguće dobiti eksplicitni oblik jednačine površi, normala se može izračunati preko gradijenta:

$$\nabla F = \begin{bmatrix} \frac{\partial F}{\partial x} & \frac{\partial F}{\partial y} & \frac{\partial F}{\partial z} \end{bmatrix}$$

Samo izračunavanje gradijenta nije teško, ali je komplikovano izračunati koordinate tačke koja leži na površi. Računanje normala zakrivljenih površina izlazi iz okvira ovog priručnika, tako da treba podrazumevati da su sve površine ravne, i osim ukoliko površina nije paralelna koordinatnim ravnima, normalu računati na osnovu vektorskog proizvoda ivica primitiva.

Prilikom zadavanja normale, potrebno je voditi računa da vektor normale bude jedinični. Ukoliko to nije slučaj, osvetljenje neće biti pravilno. Čak i kada su normale pravilno izračunate, primena neuniformnog skaliranja dovodi do narušavanja i dužine i orientacije vektora normale. Zbog toga je poželjno izbegavati neuniformno skaliranje. Ako se to ipak ne može izbeći, potrebno je uključiti automatsku normalizaciju:

```
 glEnable(GL_NORMALIZE);
```

Na ovaj način se ublažava efekat neuniformnog skaliranja, ali se usporava iscrtavanje scene, jer se zahteva normalizacija svih vektora normala. Orientacija vektora se ne „ispravlja“.

Primer definisanja normala za stranice kocke

Proširimo funkciju **DrawCube**, implementiranu u prethodnoj glavi, tako da sadrži definicije normala u temenima kocke i time omogući primenu osvetljenja.

```
void CGLRenderer::DrawCube(double a)
{
    glBegin(GL_QUADS);

        // Prednja stranica
        glNormal3d( 0.0, 0.0, 1.0);
        glVertex3d(-a/2, a/2, a/2);
        glVertex3d(-a/2,-a/2, a/2);
        glVertex3d( a/2,-a/2, a/2);
        glVertex3d( a/2, a/2, a/2);

        // Desna stranica
        glNormal3d( 1.0, 0.0, 0.0);
        glVertex3d( a/2, a/2, a/2);
        glVertex3d( a/2,-a/2, a/2);
        glVertex3d( a/2,-a/2,-a/2);
        glVertex3d( a/2, a/2,-a/2);

        // Zadnja stranica
        glNormal3d( 0.0, 0.0,-1.0);
        glVertex3d( a/2, a/2,-a/2);
        glVertex3d( a/2,-a/2,-a/2);
        glVertex3d(-a/2,-a/2,-a/2);
        glVertex3d(-a/2, a/2,-a/2);
```

```

    // Leva stranica
    glNormal3d(-1.0, 0.0, 0.0);
    glVertex3d(-a/2, a/2, -a/2);
    glVertex3d(-a/2, -a/2, -a/2);
    glVertex3d(-a/2, -a/2, a/2);
    glVertex3d(-a/2, a/2, a/2);

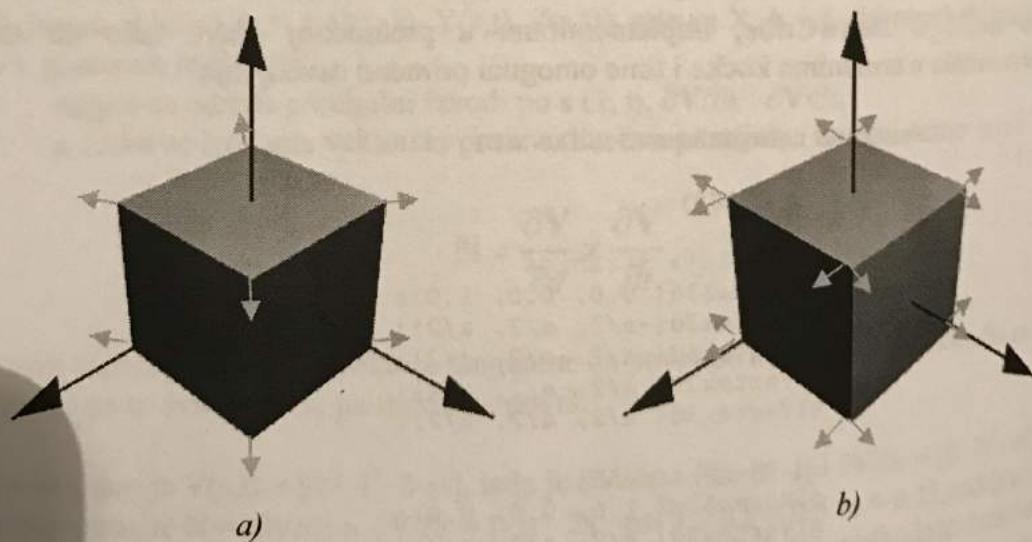
    // Gornja stranica
    glNormal3d( 0.0, 1.0, 0.0);
    glVertex3d(-a/2, a/2, a/2);
    glVertex3d( a/2, a/2, a/2);
    glVertex3d( a/2, a/2, -a/2);
    glVertex3d(-a/2, a/2, -a/2);

    // Donja stranica
    glNormal3d( 0.0,-1.0, 0.0);
    glVertex3d(-a/2,-a/2, -a/2);
    glVertex3d( a/2,-a/2, -a/2);
    glVertex3d( a/2,-a/2, a/2);
    glVertex3d(-a/2,-a/2, a/2);

    glEnd();
}

```

Iako se crtanje bočnih stranica efikasnije može ostvariti korišćenjem GL_QUAD_STRIP primitiva, da bi osvetljenje bilo pravilno moramo koristiti zasebne četvorouglove, tj. GL_QUADS. Razlog za to je potreba da u svakom temenu kocke definišemo više normala. Tačnije tri. Za svaku od stranica kocke koje se spajaju u tom temenu po jednu.



Slika 4.6. Osvetljenje kocke u slučaju da se koristi
a) GL_QUAD_STRIP i b) GL_QUADS

Kao što je ranije već rečeno, u trenutku kada se definišu prostorne koordinate temena, zatečena vrednost postavljena funkcijom `glNormal*0` postaje normala datog temena. GL_QUAD_STRIP primitiva deli definicije temena za susedne četvorouglove. To onemogućuje da se postave dve različite vrednosti za susedne strane, pa kocka prestaje da izgleda kao kocka. Gubi oštре ivice, i zavisno od tačke posmatranja i načina postavljanja normala, može izgledati kao valjak. Na slici 4.6 mogu se videti orientacije normala u

temenima bočnih strana za slučaj da se koristi GL_QUAD_STRIP primitiva (a) i GL_QUADS primitiva (b). Očigledno je da GL_QUAD_STRIP ne može imati dobro osvetljenje, čak i kada se uzimaju srednje vrednosti zajedničkih normala dve stranice (slika 4.6.a).

Definisanje materijala

Da bi se definisala interakcija objekta i svetlosti, osim normala, tj. orientacije površina u odnosu na izvore svetlosti, potrebno je definisati i materijal od koga je predmet napravljen.

Materijal određuje:

- koji deo vidljivog spektra objekat najviše reflektuje (pa time i boju objekta),
- da li je objekat sjajan ili mat,
- da li je i koliko objekat providan i
- da li emituje svetlost.

OpenGL ne definiše objekte materijala, tj. ne možemo imati više aktivnih materijala u jednom trenutku i po potrebi selektovati (aktivirati) neki od njih. Postoji samo jedan materijal koji se primenjuje na sve objekte nacrtane posle njegovog postavljanja. Postavljanje materijala podrazumeva više poziva funkcije `glMaterial*`(*0*), za svako svojstvo po jedan. Da bismo olakšali rad, definisaćemo klasu `CGLMaterial`, koja će sadržati sve atribute potrebne za postavljanje materijala, a kada je potrebno postaviti (aktivirati) dati materijal, pozvaćemo samo jednu funkciju te klase – `Select()`.

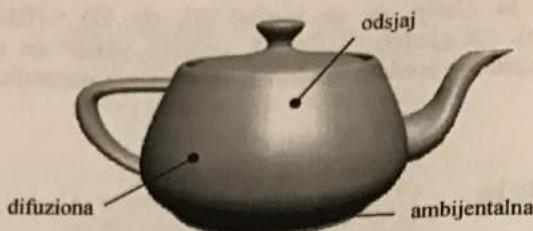
CGLMaterial	
<code>m_vAmbient[4]</code>	: float
<code>m_vDiffuse[4]</code>	: float
<code>m_vSpecular[4]</code>	: float
<code>m_vEmission[4]</code>	: float
<code>m_fShininess</code>	: float
<code>Select()</code>	
<code>SetAmbient()</code>	
<code>SetDiffuse()</code>	
<code>SetSpecular()</code>	
<code>SetEmission()</code>	
<code>SetShininess()</code>	

Slika 4.7. Klasa `CGLMaterial`

Materijal ima pet komponenti, i to:

- ambijentalnu boju (*ambient*),
- difuzionu boju (*diffuse*),
- boju odsjaja (*specular*),
- emisionu boju (*emission*) i
- sjaj (*shininess*).

Na slici 4.8 mogu se videti efekti pojedinih komponenti materijala.



Slika 4.8. Efekti pojedinih komponenti materijala

Sve komponente materijala postavljaju se funkcijom:

```
glMaterial{fi}[v] (GLenum face, GLenum pname, TYPE param)
```

Prvi argument ove funkcije (*face*) određuje stranu poligona na koju se primjenjuje dati materijal i može imati vrednost: GL_FRONT, GL_BACK ili GL_FRONT_AND_BACK. Ako se navede GL_FRONT, materijal se primjenjuje samo na prednju stranu, tj. na stranu na kojoj se temena pojavljuju u redosledu suprotnom kretanju kazaljki na satu (za detaljnije objašnjenje pogledati Glavu 2 – Crtanje primitiva).

Drugi argument (*pname*) je naziv parametra čija se vrednost modifikuje, i može imati jednu od sledećih vrednosti: GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS ili GL_AMBIENT_AND_DIFFUSE. Ukoliko se navede GL_AMBIENT_AND_DIFFUSE, ne menja se jedan, već dva parametra – ambijentalna i difuziona boja, i oba se postavljaju na istu vrednost.

Sama vrednost na koju se postavlja neka od komponenti prosleđuje se kao treći argument funkcije (*param*). Osim u slučaju definisanja sjaja, ovaj argument je vektor sa četiri komponente. Prva definiše crvenu, druga zelenu, treća plavu komponentu boje, a četvrta providnost. Providnost difuzione komponente materijala definiše i providnost čitavog objekta.

Ambijentalna komponenta materijala

Ambijentalna komponenta materijala predstavlja boju objekta kada se on nalazi u senci, tj. kada na njega ne pada direktna svetlost. Obično je to ista boja kao i boja koju vidimo kada je objekat osvetljen, ali zbog male količine svetlosti koja dopire do delova koji su u senci, obično je vidimo kao jako tamnu nijansu.

Ako svetlost od izvora svetlosti ne pada direktno na delove objekata koji su u senci, zašto onda uopšte pominjemo ovu komponentu? Obzirom da nema svetlosti, boja bi trebalo da bude mat crna. Osim ako nismo u svemiru, ovo nije tačno. Naime, svetlost se reflektuje od okolnih predmeta i osvetljava čak i objekte u senci. Što je veći broj objekata u sceni, to je efekat refleksije veći. Postupak izračunavanja prostorne distribucije energije, kao posledice refleksije svetlosti od okolnih predmeta, je vrlo složen i ne može se izračunati u konačnom vremenu. Zato se obično izračunavanje zaustavlja kada rekursija dostigne određenu dubinu, ili se postigne odgovarajući nivo konvergencije. OpenGL radi u realnom vremenu, i zbog

toga i ne pokušava da izračuna pravi efekat osvetljenja kao posledicu refleksije ili refrakcije (prelamanje prilikom prolaska kroz providne objekte). Umesto toga, efekat delovanja okoline (ambijenta) na dati objekat simulira se postavljanjem ove komponente materijala.

Prilikom postavljanja ambijentalne boje, postavlja se ili sama boja predmeta, ili njena zatamnjena varijanta. Korišćenje tamnijih tonova potencira senku. Objekti izgledaju oštije, a scena dobija na „dubini“. Ambijentalna boja, kao i sve druge komponente materijala, postavlja se funkcijom `glMaterial*()`.

```
GLfloat mat_amb[] = { 0.5, 0.5, 0.5, 1.0 };
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_amb);
```

Ako materijal ima samo ambijentalnu komponentu, a sve ostale su postavljene na 0, objekat izgleda dvodimenzionalno. Nema efekata senčenja, a boja objekta zavisi od intenziteta odgovarajuće komponente izvora svetlosti. Na slici 4.9 može se videti efekat samo ambijentalne komponente.



Slika 4.9. Efekat samo ambijentalne komponente materijala

Difuziona komponenta materijala

Difuziona komponenta materijala je ono što doživljavamo kao boju objekta. To je boja koja potiče direktno od izvora svetlosti i ravnomerno (difuziono) se reflektuje u svim prvcima. Intenzitet boje zavisi samo od kosinusa ugla koga formiraju normala na površinu objekta i pravac izvora svetlosti (Lambertov kosinusni zakon), a ne zavisi od položaja posmatrača.

I difuziona boja se postavlja funkcijom `glMaterial*()`, definisanjem vrednosti atributa `GL_DIFFUSE`.

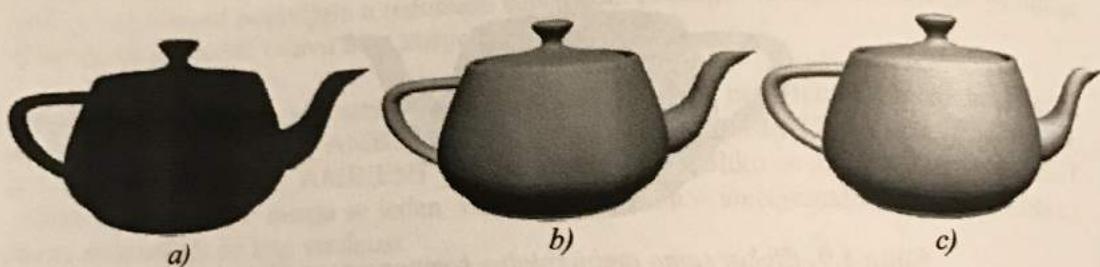
```
GLfloat mat_dif[] = { 0.8, 0.8, 0.8, 1.0 };
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_dif);
```

Efekat difuzione komponente može se videti na slici 4.10.



Slika 4.10. Efekat samo difuzione komponente materijala

Difuziona komponenta materijala dovoljna je da objekat dobije trodimenzionalni oblik, međutim, objekti izgledaju mnogo prirodnije (i svetlijе) ukoliko se kombinuje (sabere) efekat difuzione i ambijentalne komponente. Ambijentalna komponenta povećava svetlinu i delova koji su u potpunoj senci, čime simulira refleksiju od okolnih objekata, i direktno osvetljenih delova. Kombinovani efekat ambijentalne i difuzione komponente može se videti na slici 4.11.



Slika 4.11. Uporedni prikaz: a) samo ambijentalna, b) samo difuziona, c) kombinovani efekat ambijentalne i difuzione komponente materijala

Odsjaj materijala

Mnogi materijali iz realnog života nemaju sjaj. Takvi su na primer: kreda, gips, zemlja, drvo i mnogi drugi organski i neorganski materijali. Difuziona i ambijentalna komponenta u potpunosti opisuju ovakve materijale. Ali, postoji i mnoštvo materijala koji imaju sjaj. Na primer: metal, plastika, staklo, itd. Bitna karakteristika ovih materijala je da na površini objekata stvaraju oblasti svetlijе od difuzione boje, i da položaj tih oblasti zavisi od položaja svetlosnog izvora, orientacije površine objekta, ali i od položaja posmatrača. Ova pojava izazvana je direktnim reflektovanjem zraka od površine objekta ka oku posmatrača.

Sjaj materijala definiše se pomoću dva parametra: GL_SPECULAR и GL_SHININESS. GL_SPECULAR određuje boju površine sa koje se direktno reflektuje svetlost, a GL_SHININESS određuje veličinu te površine.

```
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat mat_shininess = 64.0;
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);
```

Prilikom postavljanja boje odsjaja, obično se bira bela boja, ili jako svetla nijansa difuzione boje. Na slici 4.12 može se videti efekat samo refleksione komponente materijala.



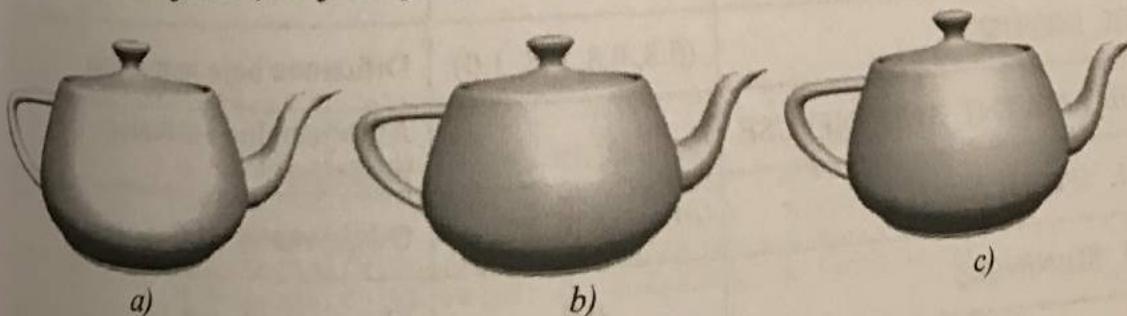
Slika 4.12. Efekat samo odsjaja materijala (refleksija)

Kombinovanjem ambijentalne, difuzione i refleksione komponente dobija se vrlo realističan izgled objekata (slika 4.13). Efekat svake od komponenti se sabira, a ukoliko zbimi intenzitet reflektovane svetlosti pređe vrednost 1.0, on se „zaokružuje“ na jedinicu. Ne postoji belja boja od bele!



Slika 4.13. Kombinovani efekat odsjaja, ambijentalne i difuzione komponente materijala

Samo boja reflektovane svetlosti nije dovoljna da stvori realističan efekat. Parametar `GL_SHININESS` definiše veličinu oblasti visokog sjaja. Može uzimati vrednosti od 0.0 do 128.0. Što je vrednost veća, to je površina manja. Ovaj parametar se još naziva i eksponent odsjaja, jer se osvetljaj računa kao kosinus ugla koji zaklapaju vektor reflektovanog zraka i položaja posmatrača u odnosu na tačku refleksije, podignut na dati stepen. Obzirom da je kosinus uvek manji od 1, što je eksponent veći, to vrednost brže pada ka nuli.



Slika 4.14. Uticaj eksponenta odsjaja:

$$a) s = 8, \quad b) s = 64, \quad c) \quad s = 128$$

Emisiona komponenta materijala

Emisiona komponenta materijala se postavlja ako želimo efekat iluminacije, tj. da izgleda kao da objekat isijava svetlost. Sam objekat neće uticati na okolne objekte i njihovo osvetljavanje. Jedino svetlosni izvori to mogu. Zbog toga, modeliranje uključene sijalice obuhvata kreiranje objekta sa visokom emisionom komponentom materijala i svetlosnog izvora u središtu tog objekta.

Emisiona komponenta materijala definiše se atributom `GL_EMISSION`, a efekat se može videti na slici 4.15. Čajnik „svetli“ kao da je neonska sijalica.

```
GLfloat mat_emission[] = {0.5, 0.5, 0.5, 1.0};
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```



Slika 4.15. Efekat emisione komponente materijala

Već je rečeno da OpenGL u jednom trenutku ima definisan samo jedan materijal. Parametri tog materijala postavljeni su i pre poziva `glMaterial*` funkcije. Pozivi ove funkcije zapravo samo modifikuju taj predefinisani materijal. Podrazumevane vrednosti parametara materijala prikazane su u tabeli 4.1.

Tabela 4.1. Podrazumevane vrednosti materijala

Parametar	Vrednost	Značenje
<code>GL_AMBIENT</code>	(0.2, 0.2, 0.2, 1.0)	Ambijentalna boja materijala
<code>GL_DIFFUSE</code>	(0.8, 0.8, 0.8, 1.0)	Difuziona boja materijala
<code>GL_AMBIENT_AND_DIFFUSE</code>	-	Ambijentalna i difuziona boja materijala
<code>GL_SPECULAR</code>	(0.0, 0.0, 0.0, 1.0)	Odsjaj materijala
<code>GL_SHININESS</code>	0.0	Eksponet odsjaja
<code>GL_EMISSION</code>	(0.0, 0.0, 0.0, 1.0)	Emisiona boja materijala

Primer implementacije klase CGLMaterial

123

Pre nego što predemo na ostale komponente osvetljenja, definišimo klasu **CGLMaterial** sa svim potrebnim atributima i metodama za postavljanje parametara materijala.

```

class CGLMaterial
{
public:
    CGLMaterial(void);
    virtual ~CGLMaterial(void);

    void Select(void);

    void SetAmbient (float r, float g, float b, float a);
    void SetDiffuse (float r, float g, float b, float a);
    void SetSpecular(float r, float g, float b, float a);
    void SetEmission(float r, float g, float b, float a);
    void SetShininess(float s);

protected:
    float m_vAmbient[4];
    float m_vDiffuse[4];
    float m_vSpecular[4];
    float m_vEmission[4];
    float m_fShininess;

};

#include "GLMaterial.h"
#include <GL\gl.h>

CGLMaterial::CGLMaterial(void)
{
    m_vAmbient[0] = 0.2; m_vAmbient[1] = 0.2;
    m_vAmbient[2] = 0.2; m_vAmbient[3] = 1.0;
    m_vDiffuse[0] = 0.8; m_vDiffuse[1] = 0.8;
    m_vDiffuse[2] = 0.8; m_vDiffuse[3] = 1.0;

    m_vSpecular[0] = 1.0; m_vSpecular[1] = 1.0;
    m_vSpecular[2] = 1.0; m_vSpecular[3] = 1.0;
    m_vEmission[0] = 0.0; m_vEmission[1] = 0.0;
    m_vEmission[2] = 0.0; m_vEmission[3] = 1.0;

    m_fShininess = 64.0;
}

CGLMaterial::~CGLMaterial(void)
{
}

void CGLMaterial::Select(void)
{
    glMaterialfv(GL_FRONT, GL_AMBIENT, m_vAmbient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, m_vDiffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, m_vSpecular);
    glMaterialfv(GL_FRONT, GL_EMISSION, m_vEmission);
    glMaterialf (GL_FRONT, GL_SHININESS, m_fShininess);
}

void CGLMaterial::SetAmbient(float r, float g, float b, float a)
{
    m_vAmbient[0] = r; m_vAmbient[1] = g;
    m_vAmbient[2] = b; m_vAmbient[3] = a;
}

```

```

}

void CGLMaterial::SetDiffuse(float r, float g, float b, float a)
{
    m_vDiffuse[0] = r; m_vDiffuse[1] = g;
    m_vDiffuse[2] = b; m_vDiffuse[3] = a;
}

void CGLMaterial::SetSpecular(float r, float g, float b, float a)
{
    m_vSpecular[0] = r; m_vSpecular[1] = g;
    m_vSpecular[2] = b; m_vSpecular[3] = a;
}

void CGLMaterial::SetEmission(float r, float g, float b, float a)
{
    m_vEmission[0] = r; m_vEmission[1] = g;
    m_vEmission[2] = b; m_vEmission[3] = a;
}

void CGLMaterial::SetShininess(float s)
{
    if(s < 0.0) m_fShininess = 0.0;
    else if(s > 128.0) m_fShininess = 128.0;
    else m_fShininess = s;
}

```

Definisanje modela osvetljenja

Pod definisanjem modela osvetljenja podrazumeva se

- određivanje jačine globalnog ambijentalnog osvetljenja
- izbor lokalnog ili udaljenog posmatrača,
- izbor jednostranog ili dvostranog osvetljenja i
- izbor da li se efekat refleksije primenjuje nezavisno od ambijentalne i difuzione boje i nakon primene teksture.

Sve što je vezano za definisanje modela osvetljenja postavlja se funkcijom:

```
void glLightModel{fi}[v]( GLenum pname, TYPE param )
```

Prvi argument funkcije definiše parametar čija se vrednost postavlja, a sama vrednost prosledjuje se kao drugi argument funkcije.

Globalno ambijentalno osvetljenje predstavlja osvetljenje koje ne potiče od postavljenih izvora svetlosti. Postavlja se zadavanjem RGBA vrednosti parametra `GL_LIGHT_MODEL_AMBIENT`. Što je vrednost veća, to je veća i globalna osvetljenost čitave scene, tačnije, globalno ambijentalno osvetljenje množi se sa ambijentalnom komponentom materijala objekata u sceni. Formula po kojoj se dobija efekat osvetljenosti data je na kraju ovog poglavlja. Podrazumevana vrednost globalnog ambijentalnog osvetljenja je (0.2, 0.2, 0.2, 1.0).

```
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

Izbor lokalnog ili udaljenog posmatrača utiče na način izračunavanja osvetljenosti objekata. Lokalni posmatrač zahteva mnogo složenije izračunavanje, jer je potrebno za svaku temu svakog od objekata izračunati ugao pod kojim ga vidi posmatrač. Kod posmatrača koji je beskonačno daleko, taj ugao se zanemaruje, pa je osvetljenje manje realistično, ali je zato izračunavanje mnogo brže. Izbor lokalnog ili udaljenog posmatrača ostvaruje se postavljenjem parametra GL_LIGHT_MODEL_LOCAL_VIEWER. Postavljanjem na GL_TRUE bira se lokalni, a GL_FALSE udaljeni posmatrač. Podrazumevana vrednost je GL_FALSE, tj. beskonačno udaljeni posmatrač. Ako želimo to da promenimo, potrebno je pozvati funkciju glLightModel*0 sa sledećim parametrima:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

Izbor jednostranog ili dvostranog osvetljenja vrši se parametrom GL_LIGHT_MODEL_TWO_SIDE. Postavljanjem ovog parametra na GL_FALSE, samo strana poligona okrenuta izvoru svetlosti biće osvetljena. Ona suprotna biće potpuno tamna. Već je rečeno da se difuziona komponenta materijala računa kao kosinus ugla koji zaklapaju vektor normale i vektor orijentisan ka svetlosnom izvoru. Ako je kosinus negativan, osvetljenje te strane je 0. Međutim, ako se parametar GL_LIGHT_MODEL_TWO_SIDE postavi na GL_TRUE, ukoliko je kosinus ugla negativan, vrši se inverzija normale, tako da i zadnja strana bude osvetljena. Ovo nije realan efekat, ali ponekad može biti koristan.

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
```

Podrazumevano stanje je – isključeno osvetljenje. Da nije tako, u prethodnim poglavljima ne bi mogli da zadajemo boju primitiva. Naime, boje definisane funkcijom glColor*0 imaju efekta samo ukoliko je svetlo isključeno. Kada se uključi osvetljenje, boju objekata definišu materijali i uključeni izvori svetlosti. Uključivanje osvetljenja ostvaruje se pozivom funkcije glEnable() i prosleđivanjem parametra GL_LIGHTING.

```
glEnable(GL_LIGHTING);
```

Isključivanje osvetljenja, bez obzira na to koliko izvora je aktivno, ostvaruje se pozivom:

```
glDisable(GL_LIGHTING);
```

Definisanje izvora svetlosti

OpenGL omogućava postavljanje do 8 izvora svetlosti. Inicijalno, svi izvori su isključeni. Uključivanje *i*-tog izvora ostvaruje se pozivom funkcije glEnable() i prosleđivanjem parametra GL_LIGHT*i*. Na primer, uključivanje nultog izvora svetlosti (numeracija počinje od 0) ostvaruje se na sledeći način:

```
glEnable(GL_LIGHT0);
```

Ako želimo izbeći delovanje izvora svetlosti na neki objekat, pozivamo funkciju glDisable() sa identifikatorom datog izvora. Na primer:

```
glDisable(GL_LIGHT0);
```

Postavljanje boje svetlosnih izvora

Osnovne karakteristike svakog izvora svetlosti su boja i intenzitet svetlosti koju emituje. Obzirom da materijal ima tri komponente na koje deluje svetlost (na emisionu komponentu ne utiče svetlost), i svetlosni izvori definišu tri komponente:

- ambijentalnu,
- difuzionu i
- refleksionu.

Sve komponente izvora svetlosti postavljaju se funkcijom:

```
void glLight{if}[v](GLenum light, GLenum pname, TYPE param)
```

gde argument *light* definiše izvor svetlosti čiji se parametar postavlja (može imati vrednost GL_LIGHT0 do GL_LIGHT7), *pname* naziv parametra, a *param* vrednost na koju se postavlja. Za postavljanje boje i intenziteta svetlosnih izvora koriste se parametri: GL_AMBIENT, GL_DIFFUSE i GL_SPECULAR.

```
float light_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
float light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
float light_specular[] = { 1.0, 1.0, 1.0, 1.0 };

glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

U prethodnom primeru definisan je svetlosni izvor bele boje. Pod dejstvom bele svetlosti objekti imaju svoju „pravu“ boju. Kako se zapravo dobija boja objekta? Svaka od komponenti izvora svetlosti množi se sa odgovarajućom komponentom materijala. Na primer, ako sa (M_{DR} , M_{DG} , M_{DB}) označimo intenzitet difuzione boje materijala (crvena, zelena i plava komponenta), a sa (L_{DR} , L_{DG} , L_{DB}) intenzitet difuzione komponente izvora svetlosti, difuziona boja objekta dobija se kao ($M_{DR} * L_{DR}$, $M_{DG} * L_{DG}$, $M_{DB} * L_{DB}$). Ista formula važi i za ostale dve komponente. Ukoliko postoji više izvora svetlosti, njihovo delovanje se sabira. Na primer, ukoliko postoje dva izvora svetlosti L^1 i L^2 , difuziona boja objekta dobija se kao ($M_{DR} * (L^1_{DR} + L^2_{DR})$, $M_{DG} * (L^1_{DG} + L^2_{DG})$, $M_{DB} * (L^1_{DB} + L^2_{DB})$). Pod dejstvom više izvora svetlosti, boja lako može da pređe u zasićenje. Vrednost veća od 1.0 zaokružuje se na 1.0.

Pozicija i slabljenje

Pozicija izvora svetlosti postavlja se funkcijom `glLightfv()`, definisanim vrednosti parametra GL_POSITION. Na primer:

```
float light_position[] = {1.0, 5.0, 5.0, 1.0};
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

U prethodnom primeru svetlo GL_LIGHT0 postavlja se na koordinate (1.0, 5.0, 5.0), ali u lokalnom koordinatnom sistemu. Gde se data pozicija nalazi u svetskom (globalnom) koordinatnom sistemu zavisi od trenutnog stanja *modelview* matrice.

Ako želimo da imamo osvetljenje vezano za pogled, na primer reflektor koji uvek prati kameru, onda se pozicija datog izvora svetlosti mora postaviti pre bilo koje *modelview* transformacije.

```

float light_position[] = { 0.0, 0.0, 1.0, 1.0 };
glLoadIdentity();
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
gluLookAt(5.0, 5.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

```

Obzirom da je nakon svih transformacija i normalizacija posmatrač u koordinatnom početku i gleda u pravcu negativne orientacije Z-ose, svetlo u prethodnom primeru uvek će biti iza njegovih leđa, pomereno za jednu jedinicu po Z-osi. U globalnom koordinatnom sistemu, sa zadatim položajem posmatrača, svetlosni izvor biće na koordinatama (5.57735, 5.57735).

Ako želimo da svetlo bude **na fiksnoj poziciji** u globalnom koordinatnom sistemu, onda se pozicija mora definisati odmah nakon *view* transformacije, tj. odmah nakon poziva *gluLookAt()* funkcije.

```

float light_position[] = { 0.0, 5.0, 0.0, 1.0 };
glLoadIdentity();
gluLookAt(5.0, 5.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);

```

U prethodnom primeru, svetlo je uvek na 5 jedinica iznad koordinatnog početka, bez obzira na položaj posmatrača i ostale transformacije.

Svetlo može biti i **pokretno**. Za to nije neophodno menjati poziciju svetla promenom *GL_POSITION* parametra. U sledećem primeru prikazano je svetlo koje rotira oko X-ose sa promenom *spin* parametra.

```

float light_position[] = { 0.0, 0.0, 5.0, 1.0 };
gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glPushMatrix();
    glRotated(spin, 1.0, 0.0, 0.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glPopMatrix();

```

Pomeranje svetla pomoću *modelview* transformacije, pogotovo kada se radi o rotacijama, brže je od pomeranja promenom *GL_POSITION* parametra, jer se izbegavaju trigonometrijske transformacije na centralnom procesoru (*Central Processing Unit* – CPU), a proces izračunavanja prenosi se na grafički procesor (*Graphical Processing Unit* – GPU). Izvršavanje transformacija koordinata na GPU su visoko optimizovane, i po brzini često prevazilaze i nekoliko redova veličina identične transformacije na CPU.

Prema vrednosti *w* koordinate pozicije, svi svetlosni izvori dele se na:

- direkcione (*w* = 0)
- pozicionе (*w* ≠ 0).

Već je ranije rečeno da se koordinate često zadaju u homogenom sistemu (pogledati **Glavu 2 – Crtanje primitiva za detaljniji opis**). Ukoliko je *w* ≠ 0, položaj tačke u Euklidovom 3D prostoru određuje se deljenjem ostale tri koordinate sa *w*. Međutim, ako je *w* = 0, deljenje se ne može obaviti. Tada kažemo da je „tačka u beskonačnosti“ u pravcu zadatom preostalim koordinatama. Ovakve koordinate koriste se za predstavljanje vektora.

Ukoliko se pozicija svetlosnog izvora zada u obliku:

```
float light_position[] = { 0.0, 0.0, 5.0, 1.0 };
```

radi se o pozicionom izvoru, postavljenom na Z-osi na 5 jedinica od koordinatnog početka.
Međutim, ako se pozicija zada kao:

```
float light_position[] = { 0.0, 0.0, 5.0, 0.0 };
```

svetlosni izvor je direkcioni, nalazi se u beskonačnosti, svi zraci koji dolaze od njega su paralelni i kreću se u pravcu Z-ose. Potpuno isti efekat za direkciono svetlo bio bi postignut i da je postavljena bilo koja druga pozitivna vrednost za z-koordinatu. Direkciono svetlo zahteva manje izračunavanja, jer svi zraci padaju na objekte pod istim uglom. Koristi se ili kada želimo simulirati osvetljenje koje potiče od Sunca ili nekog drugog udaljenog izvora, realističnosti.

Za poziciona svetla može se definisati i efekat slabljenja njihovog intenziteta sa povećanjem rastojanja. Faktor slabljenja računa se po formuli:

$$fa = \frac{1}{k_c + k_l d + k_q d^2}$$

gde su:

- d – rastojanje temena od izvora svetlosti,
- k_c – koeficijent konstantnog slabljenja (GL_CONSTANT_ATTENUATION),
- k_l – koeficijent linearног slabljenja (GL_LINEAR_ATTENUATION), i
- k_q – koeficijent kvadratnog slabljenja (GL_QUADRATIC_ATTENUATION),

Podrazumevane vrednosti su: $k_c = 1.0$ i $k_l = k_q = 0.0$, odnosno, slabljenje je isključeno. Treba biti obazriv prilikom definisanja ovih koeficijenata, pogotovo koeficijenta kvadratnog slabljenja, jer vrlo brzo intenzitet svetlosti može pasti na neželjeno nisku vrednost.

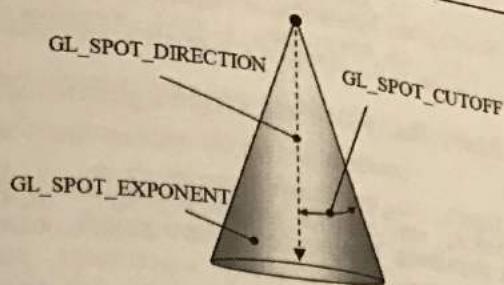
```
glLightf(GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5);
glLightf(GL_LIGHT1, GL_LINEAR_ATTENUATION, 0.5);
glLightf(GL_LIGHT1, GL_QUADRATIC_ATTENUATION, 0.1);
```

Usmereni izvori svetlosti

Pozicioni izvor svetlosti može se pretvoriti u usmereni, ograničavanjem osvetljenog prostora zapreminom kupe koju definišu sledeći parametri:

- GL_SPOT_DIRECTION – usmerenje ose granične kupe,
- GL_SPOT_CUTOFF – ugao između ose i omotača kupe, i
- GL_SPOT_EXPONENT – eksponent opadanja intenziteta svetlosti ka omotaču kupe.

Značenje svih pomenutih parametara ilustrovano je na slici 4.16.



Slika 4.16. Parametri usmerenog izvora svetlosti

Ključni parametar usmerenog izvora je GL_SPOT_CUTOFF. To je polovina ugla kupe koja definiše prostor osvetljen usmerenim izvorom svetlosti. Validne vrednosti GL_SPOT_CUTOFF parametra su od 0.0 do 90.0 i 180.0. Vrednost 180.0 koristi se za neusmereni izvor svetlosti (ovo je i podrazumevana vrednost), i u tom slučaju kupa se pretvara u sferu. Neusmereni izvor ravnomerno zrači u svim pravcima.

Da bi se dodatno povećala realističnost usmerenog izvora, parametrom GL_SPOT_EXPONENT definiše brzina opadanja intenziteta svetlosti od središnje ose ka omotaču kupe. Ukoliko je GL_SPOT_EXPONENT = 0.0, nema slabljenja. Za vrednosti veće od 0, osvetljenje se računa kao $\cos^{GL_SPOT_EXP}(\alpha)$, gde je α ugao između datog pravca i centralne ose kupe.

Ukoliko želimo da definišemo usmereni izvor svetlosti, koji je orijentisan u pravcu negativne Y-ose, ograničen kupom sa čiji je ugao pri vrhu 90° , a intenzitet opada sa kvadratom kosinusa ka omotaču kupe, to se može učiniti sledećim programskim kodom:

```
float spot_direction[] = { 0.0, -1.0, 0.0 };
glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, spot_direction);
glLightf (GL_LIGHT1, GL_SPOT_CUTOFF, 45.0);
glLightf (GL_LIGHT1, GL_SPOT_EXPONENT, 2.0);
```

Naravno, svetlosni izvor GL_LIGHT1 u prethodnom primeru mora imati konačnu poziciju, jer se parametri usmerenog izvora ne mogu definisati za izvor u beskonačnosti.

Primer definisanja svih parametara usmerenog izvora svetlosti

Da bi smo lakše sagledali šta sve možemo „podešavati“ kod izvora svetlosti, daćemo primer postavljanja jednog usmerenog izvora svetlosti uz definisanje svih njegovih parametara.

```
float light1_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
float light1_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
float light1_specular[] = { 1.0, 1.0, 1.0, 1.0 };
float light1_position[] = { -2.0, 2.0, 1.0, 1.0 };
float spot_direction[] = { -1.0, -1.0, 0.0 };
```

// Boja i intenzitet svetlosti

```

glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);
// Pozicija
glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
// Slabljjenje
glLightf (GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5);
glLightf (GL_LIGHT1, GL_LINEAR_ATTENUATION, 0.5);
glLightf (GL_LIGHT1, GL_QUADRATIC_ATTENUATION, 0.2);
// Usmerenje izvora
glLightf (GL_LIGHT1, GL_SPOT_CUTOFF, 45.0);
glLightf (GL_LIGHT1, GL_SPOT_EXPONENT, 2.0);
glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, spot_direction);
// Aktiviranje
 glEnable(GL_LIGHT1);

```

Izračunavanje osvetljenja

U svakom temenu objekata u sceni potrebno je izračunati efekat osvetljenosti prema sledećoj formuli:

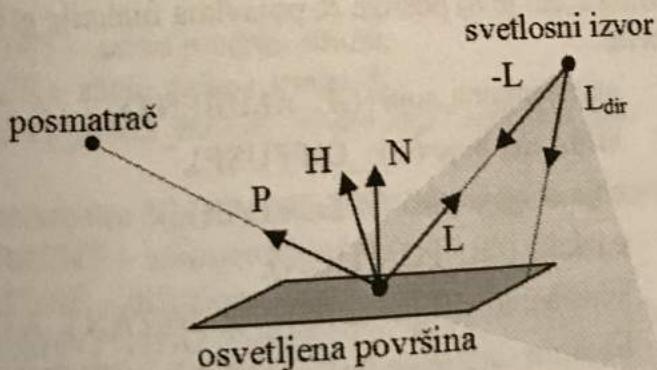
$$\begin{aligned}
 \mathbf{V}_{\text{color}} = & \mathbf{M}_{\text{emission}} + \mathbf{L}\mathbf{M}_{\text{ambient}} * \mathbf{M}_{\text{ambient}} \\
 & + \sum_i ((1 / (L[i]_{kc} + L[i]_{kl} * d + L[i]_{kq} * d^2)) * L_{\text{sl}} * \\
 & (\mathbf{L}[i]_{\text{ambient}} * \mathbf{M}_{\text{ambient}} + \max(\mathbf{L} \cdot \mathbf{N}, 0) * \mathbf{L}[i]_{\text{diffuse}} * \mathbf{M}_{\text{diffuse}} + \\
 & (\max(\mathbf{H} \cdot \mathbf{N}, 0))^s * \mathbf{L}[i]_{\text{specular}} * \mathbf{M}_{\text{specular}}))
 \end{aligned}$$

gde su:

- $\mathbf{V}_{\text{color}}$ – boja temena
- $\mathbf{M}_{\text{emission}}$ – emisiona boja materijala
- $\mathbf{M}_{\text{ambient}}$ – ambijentalna boja materijala
- $\mathbf{M}_{\text{diffuse}}$ – difuziona boja materijala
- $\mathbf{M}_{\text{specular}}$ – refleksiona boja materijala
- $\mathbf{L}\mathbf{M}_{\text{ambient}}$ – boja globalnog ambijentalnog osvetljenja
- $L[i]_{\text{ambient}}$ – ambijentalna komponenta i-tog izvora svetlosti
- $L[i]_{\text{diffuse}}$ – difuziona komponenta i-tog izvora svetlosti
- $L[i]_{\text{specular}}$ – refleksiona komponenta i-tog izvora svetlosti

$L[i]_{kc}$	– koeficijent konstantnog slabljenja i-tog izvora svetlosti
$L[i]_{kl}$	– koeficijent linear nog slabljenja i-tog izvora svetlosti
$L[i]_{kq}$	– koeficijent kvadratnog slabljenja i-tog izvora svetlosti
d	– rastojanje temena od izvora svetlosti
L_{sle}	– uticaj usmerenog izvora svetlosti ($L_{sle} = (-L \cdot L_{dir})^{slExp}$, ukoliko se teme nalazi u „kupi“ usmerenog izvora svetlosti, u protivnom je 0. L_{dir} je vektor usmerenja, a $slExp$ – vrednost GL_SPOT_EXPONENT parametra usmerenog izvora svetlosti).
L	– jedinični vektor orijentisan od temena ka izvoru svetlosti
N	– jedinični vektor normale u temenu
H	– jedinični <i>half</i> vektor (normalizovani zbir: $L + P$, gde je P jedinični vektor orijentisan od temena ka posmatraču)
*	– skalarni proizvod vektora
s	– pokomponentno množenje vektora
i	– eksponent odsjaja
i	– indeks izvora svetlosti (sumiranje se obavlja po svim aktiviranim izvorima, minimalno 0, maksimalno 7)

Orijentacija i uzajamni položaj vektora koji učestvuju u izračunavanju osvetljenosti temena prikazan je na slici 4.17.



Slika 4.17. Vektori koji učestvuju u izračunavanju osvetljenosti

Iz prethodne formule može se videti da je izračunavanje osvetljenja prilično komplikovano, te zbog toga značajno povećava vreme potrebno za iscrtavanje scene. Ovo vreme raste sa brojem izvora svetlosti i ukupnim brojem temena. Zbog toga treba biti obazriv prilikom projektovanja scene i broj izvora svetlosti smanjiti na minimum.

Kratak pregled gradiva

Da bismo omogućili osvetljenje scene, potrebno je:

- u temenima objekata definisati normale, pozivom funkcije `glNormal3*()`,
- pre iscrtavanja odgovarajućeg objekta postaviti svojstva materijala, pozivom funkcije `glMaterial*()`, i to
 - o ambijentalnu boju (`GL_AMBIENT`),
 - o difuzionu boju (`GL_DIFFUSE`),
 - o boju odsjaja (`GL_SPECULAR`) i odgovarajući eksponent sjaja (`GL_SHININESS`), i
 - o emisionu komponentu (`GL_EMISSION`)
- definisati model osvetljenja, pozivom funkcije `glLightModel*()` i postavljanjem:
 - o globalnog ambijentalnog osvetljenja (`GL_LIGHT_MODEL_AMBIENT`),
 - o lokalnog ili udaljenog posmatrača (`GL_LIGHT_MODEL_LOCAL_VIEWER`), i
 - o jednostranog ili dvostranog osvetljenja (`GL_LIGHT_MODEL_TWO_SIDE`)
- uključiti osvetljenje, pozivom funkcije `glEnable(GL_LIGHTING)`,
- postaviti parametre i aktivirati jedan ili više izvora svetlosti
 - o postavljanje parametra i -tog izvora svetlosti (maksimalan broj izvora svetlosti u sceni je 8) postiže se pozivima funkcije `glLight*()`, za sledeće parametre:
 - ambijentalna boja (`GL_AMBIENT`),
 - difuziona boje (`GL_DIFFUSE`),
 - boja odsjaja (`GL_SPECULAR`),
 - pozicija (`GL_POSITION`),
 - konstantno slabljenje (`GL_CONSTANT_ATTENUATION`),
 - linearno slabljenje (`GL_LINEAR_ATTENUATION`),
 - kvadratno slabljenje (`GL_QUADRATIC_ATTENUATION`),
 - pravac ose granične kupe (`GL_SPOT_DIRECTION`),
 - ugao između ose i omotača kupe (`GL_SPOT_CUTOFF`),
 - eksponent opadanja intenziteta svetlosti ka omotaču kupe (`GL_SPOT_EXPONENT`)
 - o aktiviranje se ostvaruje pozivom funkcije `glEnable(GL_LIGHTi)`, isključivanje pojedinih izvora svetlosti ostvaruje se pozivom funkcije `glDisable(GL_LIGHTi)`, a kompletног osvetljenja funkcijom `glDisable(GL_LIGHTING)`.

Funkcije korišćene u ovoj glavi

133

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

nx, ny, nz – koordinate vektora normale
v – koordinate vektora normale zadate kao vektor

Definiše normalu u temenima objekta.

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

cap – stanje koje se uključuje/isključuje. Može imati preko 40 različitih vrednosti, ne računajući podoblike. U ovoj glavi koriste se samo 3:
GL_NORMALIZE – uključuje/isključuje automatsku normalizaciju vektora normala
GL_LIGHTING – uključuje osvetljenje
GL_LIGHT*i* – uključuje *i*-ti izvor svetlosti (npr. **GL_LIGHT1**)

Ovaj par funkcija uključuje/isključuje odgovarajuće stanje u OpenGL *rendering context-u*

```
glMaterial{fi}[v] (GLenum face, GLenum pname, TYPE param)
```

face – stranu poligona na koju se primenjuje dati materijal i može imati vrednost:

GL_FRONT – samo prednja strana,
GL_BACK – samo zadnja strana ili
GL_FRONT_AND_BACK – i prednja i zadnja strana.

pname – naziv parametra čija se vrednost modifikuje, može biti:

GL_AMBIENT – ambijentalna boja,
GL_DIFFUSE – difuziona boja,
GL_AMBIENT_AND_DIFFUSE – ambijentalna i difuziona boja
GL_SPECULAR – reflektovana boja (odsjaj),
GL_EMISSION – emisiona boja, ili
GL_SHININESS – eksponent odsjaja

param – vrednost na koju se postavlja prethodni parametar

Funkcija postavlja sve parametre materijala.

void glLightModel{fi}[v] (GLenum pname, TYPE param)

pname – naziv parametra čija se vrednost modifikuje, a može biti:

GL_LIGHT_MODEL_AMBIENT – globalno ambijentalno osvetljenje,

GL_LIGHT_MODEL_LOCAL_VIEWER – lokalni ili udaljeni posmatrač,

GL_LIGHT_MODEL_TWO_SIDE – jednostrano ili dvostrano osvetljenje

param – vrednost na koju se postavlja prethodni parametar

Funkcija postavlja parametre modela osvetljenja.

void glLight{if}[v](GLenum light, GLenum pname, TYPE param)

light – izvor svetlosti čiji se parametar modifikuje, može biti: GL_LIGHT0,

GL_LIGHT1, GL_LIGHT2, GL_LIGHT3, GL_LIGHT4, GL_LIGHT5,

GL_LIGHT6, ili GL_LIGHT7

pname – naziv parametra čija se vrednost modifikuje, može biti:

GL_AMBIENT – ambijentalna komponenta,

GL_DIFFUSE – difuziona komponenta,

GL_SPECULAR – refleksiona komponenta,

GL_POSITION – pozicija,

GL_CONSTANT_ATTENUATION – konstantni faktor slabljenja,

GL_LINEAR_ATTENUATION – linearni faktor slabljenja,

GL_QUADRATIC_ATTENUATION – kvadratni faktor slabljenja,

GL_SPOT_DIRECTION – orijentacija usmerenog izvora svetlosti,

GL_SPOT_CUTOFF – ugao rasipanja svetlosti u odnosu na dati pravac,

GL_SPOT_EXPONENT – eksponent distribucije svetlosti.

param – vrednost na koju se postavlja prethodni parametar

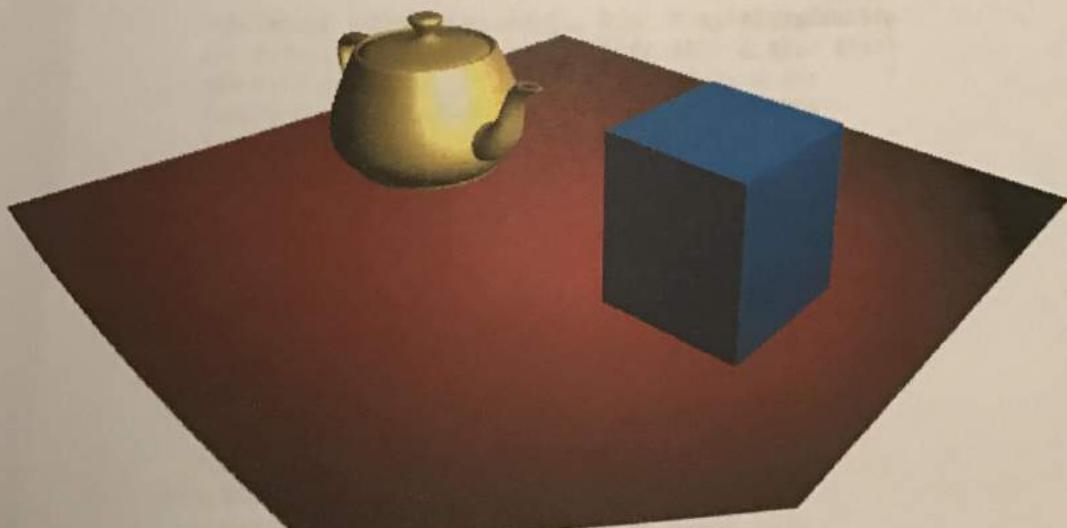
Funkcija postavlja parametre izvora osvetljenja.

Zadatak

Napisati sledeće funkcije:

- void **DrawSide**(double dSize, int nSteps) – koja crta kvadrat dimenzija $dSize \times dSize$ u XY-ravni, sa centrom u koordinatnom početku, podeljen na $nSteps \times nSteps$ podpovršina zbog pravilnijeg osvetljenja.
- void **DrawCube**(double dSize, int nSteps) – koja crta kocku čije se stranice iscrtavaju prethodnom funkcijom.
- void **PrepareLighting()** – koja postavlja parametre izvora svetlosti GL_LIGHT1, tako da ambientalna komponenta bude (0.2, 0.2, 0.2, 1.0), a sve ostale postavljene na belu boju. Izvor svetlosti treba da bude usmeren, sa uglom pri vrhu okvirne kupe od 80° i eksponentom slabljenja ka ivicama 7. U okviru ove funkciju uključiti osvetljenje.
- void **PrepareMaterials()** – koja postavlja parametre tri različita materijala:
 - o matTeapot – sjajni materijal mesingane boje,
 - o matCube – manje sjajni materijal metalno-plave boje i
 - o matTable – još manje sjajan materijal braon boje.

Funkcije **PrepareLighting()** i **PrepareMaterials()** pozvati u funkciji **CGLRender::PrepareScene()**. U funkciji **CGLRender::DrawScene()** iscrtati scenu koja se sastoji od čajnika, kocke i kvadratne površine na kojoj su postavljeni čajnik i kocka. Za crtanje čajnika koristiti funkciju **glutSolidTeapot(1.0)**, koja je deo **glut** biblioteke. Površinu na kojoj su postavljeni čajnik i kocka podeliti na 60×60 podsemenata pozivom funkcije **DrawSide()**. Dužina stranice kocke je 1.5. Svetlosni izvor GL_LIGHT1 postaviti na poziciju (5.0, 5.0, 0.0, 1.0) i usmeriti ka koordinatnom početku. Na čajnik primeniti materijal **matTeapot**, na kocku **matCube**, a na podlogu **matTable**. Izgled scene prikazan je na slici 4.18.



Slika 4.18. Izgled scene

Rešenje

Napomena: U nastavku su prikazani dodatni atributi i metode klase **CGLRenderer** traženi u zadatku. Navedene su samo funkcije čija se implementacija razlikuje u odnosu na prethodne glave.

```

// GLRenderer.h
CGLMaterial matTeapot, matCube, matTable;

void DrawSide(double dSize, int nSteps);
void DrawCube(double dSize, int nSteps);
void PrepareLighting();
void PrepareMaterials();
void PrepareScene(CDC* pDC);
void DrawScene(CDC* pDC);

// GLRenderer.cpp
void CGLRenderer::DrawSide(double dSize, int nSteps)
{
    double step = dSize/nSteps;
    glNormal3f(0.0, 0.0, 1.0);
    double y = dSize/2;
    for(int j = 0; j < nSteps; j++)
    {
        glBegin(GL_TRIANGLE_STRIP);
        double x = -dSize/2;
        for(int i = 0; i <= nSteps; i++)
        {
            glVertex2d(x, y);
            glVertex2d(x, y-step);
            x += step;
        }
        glEnd();
        y -= step;
    }
}

void CGLRenderer::DrawCube(double dSize, int nSteps)
{
    double angle = 90.0;
    int i;
    glPushMatrix();
    for(i = 0; i < 4; i++)
    {
        glRotatef(angle, 0.0, 1.0, 0.0);
        glPushMatrix();
        glTranslatef(0.0, 0.0, dSize/2.0);
        DrawSide(dSize, nSteps);
        glPopMatrix();
    }
    glPopMatrix();
    glPushMatrix();
    for(i = 0; i < 2; i++)
    {
        glRotatef(angle, 1.0, 0.0, 0.0);
        glPushMatrix();
        glTranslatef(0.0, 0.0, dSize/2.0);
        DrawSide(dSize, nSteps);
        glPopMatrix();
    }
    angle = 180.0;
}

```

OPENGL - FIKSNA FUNKCIONALNOST

137

```
    }
    glPopMatrix();
}

void CGLRenderer::PrepareLighting()
{
    float light1_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
    float light1_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    float light1_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    // Boja i intenzitet svetlosti
    glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
    glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);
    // Slabljene
    glLightf (GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.0);
    // Usmerenje izvora
    glLightf (GL_LIGHT1, GL_SPOT_CUTOFF, 40.0);
    glLightf (GL_LIGHT1, GL_SPOT_EXPONENT, 7.0);
    // Aktiviranje
    glEnable(GL_LIGHT1);
    glEnable(GL_LIGHTING);
}

void CGLRenderer::PrepareMaterials()
{
    matTeapot.SetAmbient(0.2, 0.1, 0.0, 0.0);
    matTeapot.SetDiffuse(0.7, 0.6, 0.0, 0.0);
    matTeapot.SetSpecular(1.0, 1.0, 1.0, 0.0);
    matTeapot.SetEmission(0.0, 0.0, 0.0, 0.0);
    matTeapot.SetShininess(16.0);

    matCube.SetAmbient(0.0, 0.0, 0.2, 0.0);
    matCube.SetDiffuse(0.1, 0.3, 0.7, 0.0);
    matCube.SetSpecular(1.0, 1.0, 1.0, 0.0);
    matCube.SetEmission(0.0, 0.0, 0.0, 0.0);
    matCube.SetShininess(32.0);

    matTable.SetAmbient(0.2, 0.0, 0.0, 0.0);
    matTable.SetDiffuse(0.5, 0.1, 0.0, 0.0);
    matTable.SetSpecular(0.8, 0.8, 0.8, 0.0);
    matTable.SetEmission(0.0, 0.0, 0.0, 0.0);
    matTable.SetShininess(64.0);
}

void CGLRenderer::PrepareScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClearColor (1.0, 1.0, 1.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    PrepareLighting();
    PrepareMaterials();
    //-----
    wglMakeCurrent(NULL, NULL);
}
```

```
void CGLRenderer::DrawScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(7.0, 5.0, 7.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    // Postavljanje svetla
    float light1_position[] = { 5.0, 5.0, 0.0, 1.0 };
    float spot_direction[] = {-1.0, -1.0, 0.0};
    glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
    glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, spot_direction);

    matTable.Select();
    glPushMatrix();
        glRotatef(-90.0, 1.0, 0.0, 0.0);
        DrawSide(8.0, 60);
    glPopMatrix();

    matTeapot.Select();
    glPushMatrix();
        glTranslatef(-2.0, 0.75, 0.0);
        glutSolidTeapot(1.0);
    glPopMatrix();

    matCube.Select();
    glTranslatef(2.0, 0.75, 0.0);
    DrawCube(1.5, 10);
    //-----
    glFlush();
    SwapBuffers(pDC->m_hDC);
    wglMakeCurrent(NULL, NULL);
}
```



Teksture

U ovoj glavi povećaćemo nivo realističnosti scene uvođenjem tekstura. Teksture su dvodimenzionalne slike „nalepljene“ na površine objekata i one mogu značajno povećati detaljnost prikaza, bez uticaja na složenost samih objekata. Štaviše, izgled drveta, mermera ili tekstila može se postići jedino korišćenjem tekstura. Kao teksture, vrlo često se koriste fotografije, čime scena dodatno dobija na fotorealističnosti.

Tipovi tekstura

Primena tekstura je jedna od najkompleksnijih oblasti u OpenGL-u, i oblast koja se najviše menjala sa svakom novom verzijom API-ja. Razlog za to je činjenica da u vreme kada je počeo da se razvija OpenGL hardver nije bio dovoljno „zreo“ za rukovanje teksturama.

Trenutno, OpenGL podržava šest osnovnih tipova tekstura, i to:

- jednodimenzionalne,
- dvodimenzionalne,
- trodimenzionalne
- kubne,
- jednodimenzionalna polja,
- dvodimenzionalna polja,
- teksturne bafere i
- pravougaone teksture.

Jednodimenzionalne (1D) teksture su jednodimenzionalna polja texsela. Tekselom ćemo nazivati piksele koji pripadaju teksturi, kako bi ih razlikovali od piksela koji se prikazuju na ekranu. Ove teksture imaju samo jednu koordinatu, koja se najčešće označava slovom *s*.

Dvodimenzionalne (2D) teksture su matrice teksela. To su slike koje imaju dužinu i visinu, i poput tapeta se „lepe“ na površinu objekata. Ove tekture imaju dve koordinate, koje se označavaju slovima s i t.

Trodimenzionalne (3D) tekture su polja matrica teksela. Osim širine i visine, ove tekture imaju i dubinu.

Kubne tekture sastoje se od šest dvodimenzionalnih tekstura. Mapiranje kubne tekture vrši se tako što se oko datog objekta projektuje opisana kocka, čije su strane poravnate sa koordinatnim osama. Svaka od šest 2D tekstura pripada jednoj stranici kocke. Da bi se odredila teksturna koordinata nekog temena, projektuje se zrak iz središta objekta koji prolazi kroz dato teme i preseca opisanu kocku. Na osnovu mesta preseka određuje se iz koje od šest 2D tekstura treba primeniti teksele, i koje su teksturne koordinate u datoj 2D teksturi.

Počev od verzije 1.3, OpenGL-a podržava i višestruko teksturisanje. To znači da se izgled površine objekta može definisati kao kumulativni efekat različitih tekstura, pri čemu koordinate temena u okviru različitih tekstura mogu biti različite.

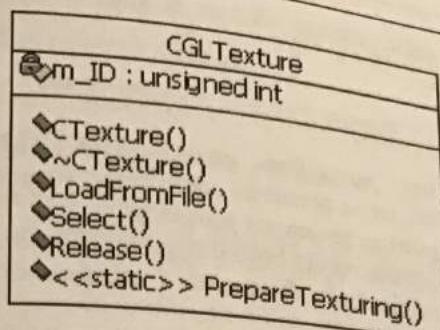
OpenGL 1.1 nema podršku za višestruko teksturisanje, a od tipova tekstura podržava samo jednodimenzionalne i dvodimenzionalne tekture. Zbog jednostavnosti i njihove najšire primene, u ovoj glavi razmotrićemo samo dvodimenzionalne tekture, i to bez višestrukog teksturisanja.

Klasa CGLTexture

Jedna od najvećih prednosti OpenGL-a 1.1 u odnosu na prethodnu verziju jeste uvođenje objekata tekstura. To su strukture koje omogućuju vezivanje atributa, na primer, izvora podataka, načina filtriranja i primene tekture, za određeni resurs. Na taj način stvara se infrastruktura za efikasno rukovanje teksturama, obzirom da nije potrebno postavljati sve parametre (kao, na primer, kod materijala) prilikom promene tekture sa kojom se radi, već samo selektovati odgovarajući objekat.

Da bismo dodatno olakšali rukovanje teksturama, uvešćemo i C++ klasu - **CGLTexture**, koja će sadržati odgovarajuće atribute i funkcije. Najvažniji atribut je **m_ID**, jedinstveni identifikacioni broj, na osnovu koga će OpenGL pristupati datoj teksturi. Od funkcija, implementiraćemo:

- **PrepareTexturing()** – za globalno podešavanje teksturisanja (nevezano za pojedine objekte),
- **LoadFromFile()** – za kreiranje tekture na osnovu datoteke, tj. slike u BMP formatu,
- **Select()** – za aktiviranje date tekture i
- **Release()** – za oslobođanje resursa koji zauzima tekstura u memoriji grafičke kartice.



Slika 5.1. Klasa CGLTexture

Implementacija pojedinih metoda biće prikazana u nastavku glave, uporedno sa prikazom odgovarajućih koncepata u radu sa teksturama.

Načini primene tekstura

Pre nego što pređemo na formiranje objekata tekstura, razmotrimo parametre koji nisu vezani za pojedinačne objekte. Ovi parametri utiču na sve teksture, i „na snazi“ su sve dok se eksplisitno ne promene.

Funkcija `glPixelStorei()` definiše način na koji je bitmapa (slika) smeštena u memoriji. Poznato je da se redosled bajtova u rečima i dvostrukim rečima različito tumači kod različitih računara. Takođe, vrlo često se, zbog brzine prenosa, bajtovi podataka organizuju u veće celine. Zbog toga, na primer, širina slike, merena u bajtovima, mora biti deljiva sa 2 (poravnanje po rečima), 4 (poravnanje po dvostrukim rečima), ili čak 8. Funkcija ima sledeći oblik:

```
void glPixelStore{if} (GLenum pname, TYPE param);
```

gde je *pname* naziv parametra koji se postavlja, a *param* njegova vrednost. Postoji mnoštvo parametara, i nekada je potrebno i više sukcesivnih poziva ove funkcije da bi se postavilo željeno stanje. Na sreću, u svim narednim primerima koristićemo nezapakovane podatke, i nećemo razmenjivati podatke između različitih platformi (što bi zahtevalo konverziju redosleda bajtova). Jedino na šta bi mogli da obratimo pažnju jeste poravnanje nezapakovanih podataka, tj. sa koliko bajtova treba da bude deljiva širina slike. Za to se koristi parametar `GL_UNPACK_ALIGNMENT`. Ukoliko ga postavimo na 1, to znači da će odmah nakon poslednjeg piksela jedne linije u slici, u sledećem bajtu započeti prvi piksel sledeće linije. Ovakav slučaj se najčešće javlja kada ručno generišemo sliku. Ako se slika učitava iz BMP datoteke, dužina svake linije mora biti deljiva sa 4. To znači da nakon poslednjeg piksela jedne linije, trebamo preskočiti od 0 do 3 bajta, da bismo preuzeli prvi piksel sledeće linije. Naravno, o ovom preskoku vodiće računa OpenGL. Potrebno je samo navesti kakvo je poravnanje. Podrazumevana je vrednost 4, tj.:

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
```

ali ćemo ovu naredbu ipak uključiti u implementaciju funkcije `CGLTexture::PrepareTexturing()`, da bismo ukazali na tekući način poravnjanja.

Funkcija kojom se definiše način primene teksture, tj. kako se vrednost pročitana iz tekture meša sa bojom fragmenata je:

```
void glTexEnv(if) ( GLenum target, GLenum pname, GLint param );
```

Parametar *target* mora biti postavljen na GL_TEXTURE_ENV, a *pname* na GL_TEXTURE_ENV_MODE, jer se postavlja „okruženje“ (eng. *environment*) za teksture. Vrednosti koje se mogu postaviti za parametar *param* su:

- GL_REPLACE – potpuno menja boju fragmenata podacima iz tekture,
- GL_DECAL – mešanje boje fragmenata i tekture definisano je providnošću tekture,
- GL_MODULATE – boja fragmenata je „modulisana“ teksturom i
- GL_BLEND – boja se dobija mešanjem boje fragmenata, tekture i posebno zadate boje.

Boja koja se koristi za mešanje u modu GL_BLEND, definiše se pozivom sledeće funkcije:

```
glTexEnv{if}v( GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, color );
```

gde je *color* polje sa četiri komponente koje sadrži RGBA boju.

Ako indeksom *t* označimo boju koja potiče od tekture, indeksom *f* boju fragmenata, a indeksom *c* boju definisanu parametrom GL_TEXTURE_ENV_COLOR, tada se konačna boja dobija primenom pravila zadatih u tabeli 5.1. Slovo **C** označava bilo koju od tri komponente boje (R, G ili B), a slovo **A** providnost (*alpha*). Kolona RGB odnosi se na slučaj kada tekstura nema providnost (tj. definisana je kao RGB), dok kolona RGBA definiše slučaj kada tekstura ima i providnost.

Tabela 5.1. Pravila za primenu tekstura

Mod	RGB	RGBA
GL_REPLACE	$C = C_t$ $A = A_f$	$C = C_t$ $A = A_t$
GL_DECAL	$C = C_t$ $A = A_f$	$C = C_f(1 - A_t) + C_t A_t$ $A = A_f$
GL_MODULATE	$C = C_f C_t$ $A = A_f$	$C = C_f C_t$ $A = A_f A_t$
GL_BLEND	$C = C_f(1 - C_t) + C_c C_t$ $A = A_f$	$C = C_f(1 - C_t) + C_c C_t$ $A = A_f A_t$

Dakle, ukoliko želimo realističnu scenu sa uticajem osvetljenja, onda svakako treba primeniti GL_MODULATE režim. U protivnom, koristimo GL_REPLACE ili GL_DECAL. Razlika između ova dva režima je u tome što GL_DECAL uzima u obzir providnost tekture pri mešanju sa bojom fragmenata, dok GL_REPLACE bezuslovno menja boju fragmenata teksturom. Režim GL_BLEND se koristi za posebne efekte.

Korišćenjem prethodnih funkcija, možemo da napišemo implementaciju funkcije **CGLRenderer::PrepareTexturing()**. Kao jedini parametar ove funkcije prosledićemo

logičku vrednost koja ukazuje na to da li želimo efekat osvetljenja, i na osnovu toga postaviti odgovarajući režim primena tekstura.

```
void CGLTexture::PrepareTexturing(bool bEnableLighting)
{
    glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
    if(bEnableLighting)
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                  GL_MODULATE);
    else
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                  GL_REPLACE);
}
```

Kreiranje objekata tekstura

Sledeći korak je kreiranje objekata tekstura. Za svaku BMP sliku koju budemo učitali iz datoteke kreiraćemo po jednu instancu klase **CGLTexture** i svaka od njih odgovaraće samo jednom OpenGL objektu tekstura.

Da bismo dobili validnu i jedinstvenu vrednost za identifikator teksture, potrebno je pozvati funkciju

```
void glGenTextures(GLsizei n, GLuint *textureNames);
```

Ova funkcija vraća *n* trenutno slobodnih identifikatora, u obliku neoznačenih celih brojeva, koje smešta u polje *textureNames*.

Za kreiranje i aktiviranje objekta tekstura, koristi se funkcija

```
void glBindTexture(GLenum target, GLuint textureName);
```

Parametar *target* određuje tip teksture, i može biti **GL_TEXTURE_1D**, **GL_TEXTURE_2D**, **GL_TEXTURE_3D** ili **GL_TEXTURE_CUBE_MAP**, a *textureName* je identifikator teksture, koji je vratila funkcija **glGenTextures()**.

Funkcija **glBindTexture()** ima više uloga. Kada je prvi put pozvana za dati identifikator, ona zapravo kreira objekat tekstura u memoriji grafičke kartice, i selektuje dati objekat. Sve kasnije modifikacije parametara odnosiće se na selektovani objekat. Kada se pozove za prethodno kreirani objekat, onda samo selektuje (aktivira) dati objekat. A kada se pozove za vrednost 0, isključuje rad sa objektima tekstura i aktivira neimenovanu „podrazumevanu“ tekstuру. Dakle, nula ne može biti identifikator objekta.

Pogledajmo na primeru kako izgleda kreiranje jedne 2D teksture:

```
unsigned int m_ID;
glGenTextures(1, &m_ID);
glBindTexture(GL_TEXTURE_2D, m_ID);
```

Nakon kreiranja, potrebno je tekstuру „napuniti“ podacima. Osnovna funkcija za prebacivanje podataka u tekstuру, uz istovremeno definisanje načina na koji su podaci smešteni jeste:

```
void glTexImage2D(GLenum target, GLint level, GLint internalFormat,
                  GLsizei width, GLsizei height, GLint border,
                  GLenum format, GLenum type, const GLvoid *texels);
```

Parametar *target* definiše tip teksture, i u svim primerima biće `GL_TEXTURE_2D`, obzirom da radimo sa 2D teksturama. Parametar *level* definiše koji nivo teksture popunjavamo. Nivo 0 je najdetaljniji nivo, i na njemu se postavlja tekstura najveće rezolucije. Parametar *internalFormat* definiše koliko komponenata ima jedan teksel i kako su one organizovane. Može imati vrednosti: 1, 2, 3, 4 ili jednu od 55 drugih predefinisanih (3) ili `GL_RGBA` (4), u zavisnosti od toga da li slika ima 3 ili 4 komponente po jednom pikselu.

Parametri *width* i *height* određuju širinu i visinu slike, respektivno, i moraju biti stepeni broja 2. Ovo ograničenje je ukinuto tek u OpenGL-u 2.0, ali korišćenje takvih tekstuera je dosta limitirano. Parametar *border* definiše širinu okvira oko slike. Dakle, čitava slika je dimenzija $(2^n + \text{border}) \times (2^n + \text{border})$, gde je *width* = 2^n , a *height* = 2^n . Okvir se koristi da bi se eliminisali „šavovi“ na mestu spoja dve ili više tekstuera. OpenGL, u zavisnosti od izabranog načina filtriranja tekstuera, obično meša više teksela da bi dobio jedan piksel. Na taj način je slika mekša i prijatnija za oko. Međutim, kada se teksel nalazi na rubu tekstuera, pored njega nema drugog teksela sa kojim bi mogao da bude pomešan. Zato se na rubovima javljaju artifakti poznati pod nazivom „šavovi“. Postojanje okvira sprečava ovu pojavu.

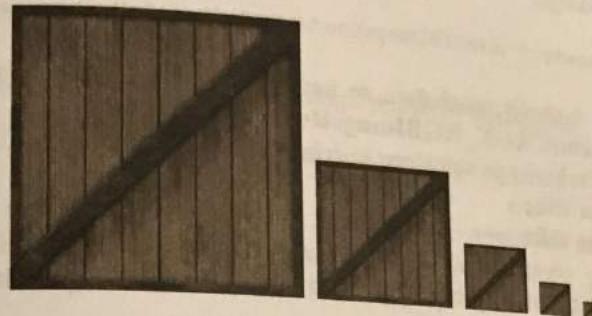
Parametri *format* i *type* definišu kako su organizovani pikseli u slici i kog su tipa pojedine komponente piksela. Na primer, *format* = `GL_RGB` i *type* = `GL_UNSIGNED_BYTE`, znači da piksel ima tri komponete u redosledu navedenom u formatu, pri čemu je svaka komponenta jedan neoznačeni bajt. Poslednji parametar je pokazivač na buffer iz koga se mogu pročitati sami podaci.

Ako, na primer, u strukturi `TextureImage` imamo zapamćene parametre slike koju želimo da koristimo kao tekstuру, i ukoliko je *sizeX* širina slike, *sizeY* visina slike, a *data* pokazivač na podatke, poziv funkcije `glTexImage2D()` za sliku bez providnosti mogao bi imati sledeći oblik:

```
glTexImage2D( GL_TEXTURE_2D, 0, 3,
               TextureImage->sizeX, TextureImage->sizeY,
               0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage->data );
```

Ovako definisana tekstura ima samo najveći nivo detalja. Neka je u našem primeru to 512×512 teksela. Ako je objekat jako udaljen, tekstura će se videti kao mala. Ukoliko se prilikom crtanja koristi, recimo, svaki 10-ti teksel, tekstura će izgledati iskrzano, i vrlo verovatno će biti potpuno neprepozнатljiva. Umesto fotorealistične slike, imaćemo samo skup raznoboјnih tačaka. Očigledno je da je potrebno smešati grupe od po 100 teksela (10×10) u jedan, i njima zameniti čitavu grupu. Ovo je nemoguće raditi u realnom vremenu za stotine ogromnih tekstuera, koliko može imati jedna scena. Zato se došlo na ideju da se unapred pripreme i umanjene verzije originalne tekstuera. Godine 1983 Lance Williams je u

svom radu „*Pyramidal Parametrics*“ prvi put spomenuo kovanicu *mipmap*. Ona je dobijena od latinske fraze *multum in parvo* (mnoštvo na jednom mestu). Osnovna ideja mipmape je da se u istom memorijskom prostoru, osim originalne mape, smeste i njene kopije, od kojih je prva $4 \times$ manja ($2 \times$ po svakoj dimenziji), druga $16 \times$ manja, treća $64 \times$ manja, itd. Sve dok se ne dođe do mape veličine 1×1 teksel. Na ovaj način, zauzima se samo za trećinu veći prostor, a dobici u kvalitetu i brzini su neverovatni. Na slici 5.2 dat je primer jedne mipmape.



Slika 5.2. Primer mipmape

Da bismo mogli da koristimo mipmape, potrebno je da izračunamo umanjene (i umekšane) verzije slika i prosledimo višim nivoima teksture. Na primer:

```
glTexImage2D( GL_TEXTURE_2D, 0, 3,
               TextureImage->sizeX, TextureImage->sizeY,
               0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage->data0 );

glTexImage2D( GL_TEXTURE_2D, 1, 3,
               TextureImage->sizeX/2, TextureImage->sizeY/2,
               0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage->data1 );

glTexImage2D( GL_TEXTURE_2D, 2, 3,
               TextureImage->sizeX/4, TextureImage->sizeY/4,
               0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage->data2 );

glTexImage2D( GL_TEXTURE_2D, 3, 3,
               TextureImage->sizeX/8, TextureImage->sizeY/8,
               0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage->data3 );

// itd.
```

pri čemu su *data0*, *data1*, *data2*, *data3*, itd. pokazivači na odgovarajuće podatke (slike).

Iz prethodnog primera vidimo da ovo može biti jako zahtevan posao. Na sreću, u pomoćnoj biblioteci postoji funkcija koja sve to može uraditi za nas, i to na vrlo kvalitetan način. To je funkcija:

```
int gluBuild2DMipmaps(GLenum target, GLint internalFormat,
                      GLint width, GLint height,
                      GLenum format, GLenum type, void *texels);
```

Parametri ove funkcije identični su odgovarajućim parametrima funkcije `glTexImage2D()`. Ova funkcija ima još jednu jako „lepu“ osobinu. Ukoliko joj se prosledi slika čije dimenzije nisu stepen dvojke, ona će automatski za prvi nivo detalja formirati po veličini najpribližniju sliku koja to zadovoljava.

Ostaje još problem kako učitati sliku sa hard diska. Ukoliko je slika u BMP formatu, za to možemo koristiti funkciju:

```
AUX_RGBImageRec* auxDIBImageLoad(LPCWSTR fileName);
```

Kao parametar ovoj funkciji prosleđuje se naziv datoteke, a ona kreira strukturu u koju učitava podatke. Struktura `AUX_RGBImageRec` ima tri atributa:

- **data** – bafer u kome je smeštena sadržina slike (pikseli),
- **sizeX** – širina slike i
- **sizeY** – visina slike.

Da bismo mogli koristiti ovu funkciju, potrebno je uključiti `glaux` biblioteku u naš projekat.

```
#include <GL\glaux.h>
#pragma comment (lib, "glaux.lib")
```

Struktura `AUX_RGBImageRec` se može uništiti (obrisati) odmah nakon poziva funkcije koja kreira teksturu na osnovu njenih atributa. Funkcija `glTexImage2D()` prenosi sve potrebne podatke u teksturu, pa nema potrebe za daljim zauzimanjem operativne memorije. Funkcija `gluBuild2DMipmaps()` interno poziva `glTexImage2D()`, pa sve prethodno važi i za nju.

Povežimo sve prethodno rečeno u vezi kreiranja tekstura u jednu celinu i implementirajmo funkciju `CGLTexture::LoadFromFile()`.

```
void CGLTexture::LoadFromFile(CString texFileName)
{
    if(m_ID != 0) Release();

    // Alokacija ID-a i kreiranje teksture
    glGenTextures(1, &m_ID);
    glBindTexture(GL_TEXTURE_2D, m_ID);

    // Ucitavanje bitmape
    AUX_RGBImageRec *TextureImage;
    TextureImage = auxDIBImageLoad(texFileName);

    // Kopiranje sadrzaja bitmape u teksturu
    glTexImage2D(GL_TEXTURE_2D, 0, 3,
                 TextureImage->sizeX, TextureImage->sizeY,
                 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage->data);

    // Brisanje bitmape
    if (TextureImage) // ako je ucitana slika
    {
        if (TextureImage->data) // i ako sadrzi podatke
        {
            free(TextureImage->data); // obrisati podatke
        }
        free(TextureImage); // obrisati samu strukturu
    }
}
```

Funkcija `LoadFromFile()` još nije sasvim kompletna, jer nismo definisali parametre teksture. No, za sada, možemo i bez njih. Ako se ne navedu, tekstura će biti kreirana sa podrazumevanim vrednostima.

Parametri tekstura

Postoji mnoštvo parametara tekstura koji se mogu postaviti funkcijom:

```
void glTexParameter(GLenum target, GLenum pname, TYPE param);
```

Obzirom da ćemo ograničiti rad samo na 2D teksture, `target` bi uvek trebalo da bude `GL_TEXTURE_2D`, a preostala dva argumenta funkcije su naziv parametra (`pname`) i njegova vrednost (`param`).

Četiri najvažnija parametra teksture su:

- filter koji se koristi pri uvećanju,
- filter koji se koristi pri umanjenju,
- način ponavljanja po S pravcu i
- način ponavljanja po T pravcu.

Filtriranje

U zavisnosti od toga koliko je objekat, na koji je primenjena tekstura, daleko od posmatrača, prilikom formiranja konačne slike, jedan teksel može uticati na formiranje više piksela, ili se više teksela može preslikati u samo jedan piksel. Moguće je i da obe pojave nastanu istovremeno na jednoj teksturi, ukoliko je objekat dugačak, i orijentisan u pravcu pogleda.

Prvi slučaj, kada jedan teksel utiče na formiranje više piksela, naziva se **uvećanje** teksture, i nastaje kada je posmatrač jako blizu objekta (slika 5.3). Kako će se tekstura ponašati u takvoj situaciji definišemo pozivom funkcije:

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, param);
```

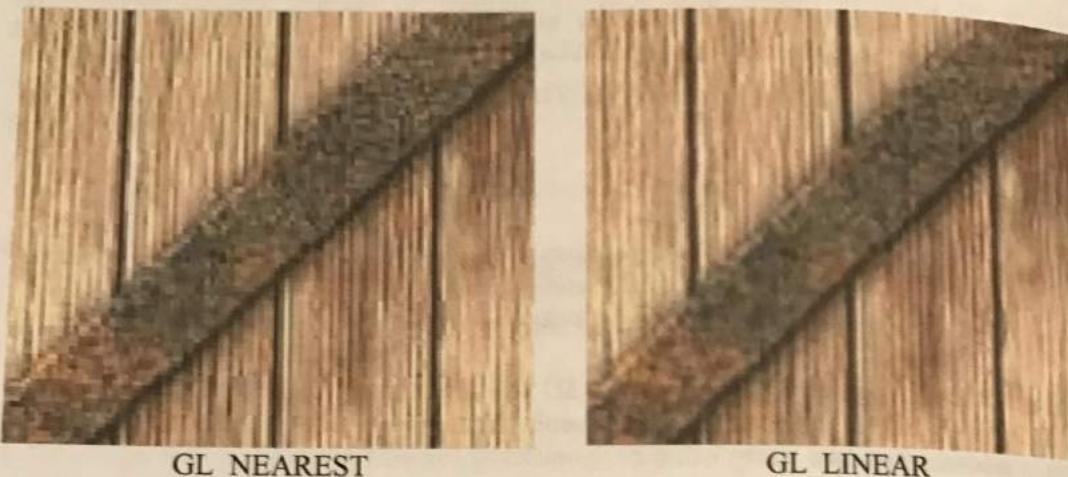
Parametar `param` može imati jednu od sledeće dve vrednosti:

- `GL_NEAREST`, ili
- `GL_LINEAR`.

Vrednost `GL_NEAREST` definiše da se za boju datog piksela treba uzeti boja teksela čiji je centar najbliži centru piksela. Ovo je najbrži metod za prikaz, ali je vizuelni efekat najmanje prihvatljiv. U levom delu slike 5.3 možemo videti efekat ovakvog filtriranja tekstura pri uvećanju.

Vrednost `GL_LINEAR` boju piksela formira usrednjavanjem boja četiri teksela najbliža centru datog piksela (odnosno, vrši težinsko linearno usrednjavanje polja 2×2 teksela).

Dobija se mekši prikaz, ali samo do određene granice. Ako smo preblizu površini koju posmatramo, artifakti su neizbežni.



Slika 5.3. Tipovi filtriranja tekstura prilikom uvećanja

Kada je objekat koji posmatramo daleko, više texsela stapaaju se u samo jedan piksel. Tada kažemo da nastaje **umanjenje** teksture. Kako će se tekstura ponašati pri umanjenju definišemo pozivom funkcije:

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, param);
```

Parametar *param* može imati jednu od sledećih vrednosti:

- GL_NEAREST,
- GL_LINEAR,
- GL_NEAREST_MIPMAP_NEAREST,
- GL_NEAREST_MIPMAP_LINEAR,
- GL_LINEAR_MIPMAP_NEAREST, ili
- GL_LINEAR_MIPMAP_LINEAR.

Efekti vrednosti GL_NEAREST i GL_LINEAR su identični kao kod filtera uvećanja. Ostale vrednosti moguće su samo ukoliko postoje mipmape. Ako mipmapa nije formirana kompletno (nedostaju neki nivoi), ili korektno (neki od nivoa nije zadat kako treba), a koristi se neki od GL_*_MIPMAP_* filtera, OpenGL automatski isključuje teksturisanje.

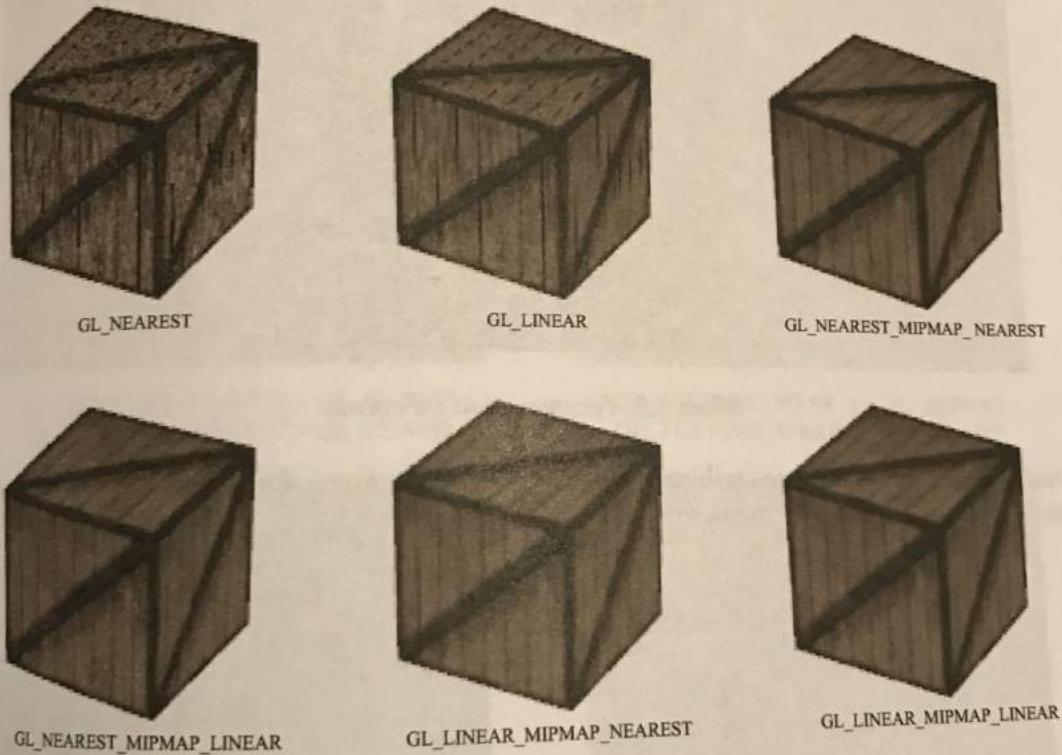
Vrednost GL_NEAREST_MIPMAP_NEAREST omogućuje da za boju piksela uzme teksel najbliži centru piksela iz odgovarajućeg nivoa mipmape (nivo koji sadrži odgovarajući teksel, koji je po veličini najbliži datom pikselu).

Vrednost GL_LINEAR_MIPMAP_NEAREST omogućuje da se boja piksela dobija linearnim usrednjavanjem polja 2×2 texsela iz nivoa mipmape koji po veličini texsela najviše odgovara datom pikselu.

Vrednost `GL_NEAREST_MIPMAP_LINEAR` vrši linearno usrednjavanje boja dva teksela, čiji su centri najbliži centru piksela, a uzimaju se iz dva susedna nivoa mipmape sa odgovarajućim veličinama teksela.

Vrednost `GL_LINEAR_MIPMAP_LINEAR` vrši linearno usrednjavanje boja dve grupe od po 2×2 teksela, iz dva susedna nivoa mipmape. Ovo je najsloženiji tip filtriranja i daje najmekši prikaz.

Na slici 5.4 prikazani su efekti svih filtera koji se koriste pri umanjenju tekstura. Složenost raste i kvalitet, tačnije umekšanost, prikaza. Razlike su najuočljivije za prva tri filtera: dodaju suptilne promene u kvalitetu, koje su teško uočljive na slici 5.4.



Slika 5.4. Tipovi filtriranja tekstura prilikom umanjenja

Ponavljanje

Teksture koje se primenjuju na neke objekte mogu biti takve strukture da omogućuju formiranje šablonu, koji se zatim mogu ponavljati više puta na površini samih objekata. Odličan primer za to je tekstura zida od cigala (slika 5.5). Ako bismo napravili jednu teksturu za čitav zid, ta tekstura bi morala biti prilično velika da bi sadržala dovoljan nivo detalja. Velike teksture zauzimaju mnogo prostora u memoriji grafičke kartice, sporo se učitavaju i prenose, i generalno je potrebno izbegavati njihovo korišćenje. Pogotovu zato što postoji ograničenje u maksimalnoj veličini tekstuра koja se može formirati za određenu

grafičku karticu. Ova vrednost može se očitati korišćenjem sledećeg programskog segmenta:

```
int param;
glGetIntegerv(GL_MAX_TEXTURE_SIZE, &param);
```

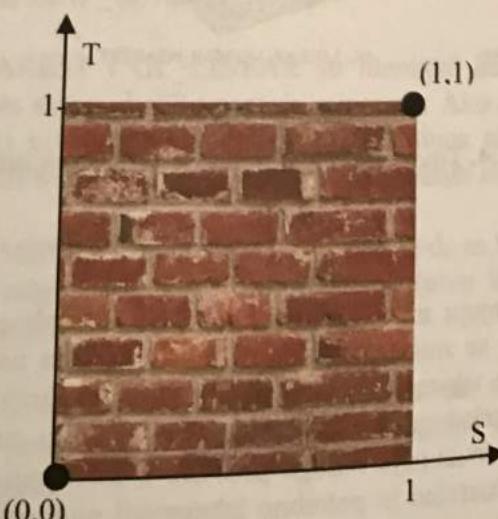
Nakon poziva funkcije `glGetIntegerv()`, u *param* je upisana maksimalna dimenzija teksture, i ona za savremene grafičke kartice sa dovoljno memorije ($\geq 512\text{MB}$) iznosi 8192. Ova vrednost je teoretski maksimum, jer u 512MB memorije može stati samo jedna nekompresovana 2D RGBA tekstura dimenzija 8192×8192 . U većini primena, maksimalna veličina teksture je 2048×2048 teksela, osim ukoliko se ne koriste atlasi tekstura, kada se više zasebnih slika smešta samo u jednu teksturu, kako bi se minimizovao broj aktivacija tekture (i time poboljšale performanse).



Slika 5.5. Tekstura zida od cigala

Način ponavljanja teksture, prilikom primene na odgovarajući objekat, može se definisati sledećim programskim segmentom:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, repParam);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, repParam);
```



Slika 5.6. Teksturne koordinatne ose

Parametar *repParam* određuje da li se vrši ponavljanje teksture po datom pravcu (S ili T), ili ne. S i T su koordinatne ose teksture koje odgovaraju X i Y osama (slika 5.6). Drugačije oznake su uvedene da bi se teksturne koordinate temena (s,t,p,q) razlikovale od prostornih koordinata (x,y,z,w).

Bez obzira na dimenzije teksture, koordinata (0,0) predstavlja donji levi ugao teksture, a koordinata (1,1) gornji desni. Sve dok koristimo koordinate u intervalu [0..1], ponavljanje ne igra nikakvu ulogu u primeni tekstura. Međutim, ukoliko izademo iz ovog intervala, parametar *repParam* određuje koji će tekseli biti iskorišćeni za teksturisanje objekta.



GL_TEXTURE_WRAP_S = GL_CLAMP
GL_TEXTURE_WRAP_T = GL_CLAMP



GL_TEXTURE_WRAP_S = GL_REPEAT
GL_TEXTURE_WRAP_T = GL_CLAMP



GL_TEXTURE_WRAP_S = GL_CLAMP
GL_TEXTURE_WRAP_T = GL_REPEAT



GL_TEXTURE_WRAP_S = GL_REPEAT
GL_TEXTURE_WRAP_T = GL_REPEAT

Slika 5.7. Ponavljanje tekstura

Ako je vrednost GL_REPEAT, tekstura se ponavlja po odgovarajućem pravcu. To se ostvaruje jednostavnim odbacivanjem celobrojne vrednosti teksturne koordinate, ukoliko je

koordinata pozitivna, ili oduzimanjem apsolutne vrednosti razlomljenog dela od jedinice, ako je koordinata negativna. Na primer, boje piksela sa teksturnim koordinatama 3.7 i -2.3, biće identične boji piksela sa teksturnom koordinatom 0.7.

Ako je vrednost `GL_CLAMP`, okvirni tekseli se koriste za sve koordinate izvan intervala [0..1]. To znači da će svi pikseli sa teksturnim koordinatama većim od 1, koristiti teksel sa koordinatom 1.0, a svi pikseli sa negativnim koordinatama teksel sa koordinatom 0.0. Slika 5.7 ilustruje prethodne primere.

Postavljanje načina filtriranja i ponavljanja tekstura (kao i ostalih parametara tekstura), može se ostvariti samo kada je tekstura aktivna, tj. nakon poziva funkcije `glBindTexture()` sa identifikatorom odgovarajuće teksture kao parametrom funkcije. Obzirom da se parametri tekstura obično ne menjaju u toku izvršenja programa, idealno mesto za njihovo postavljanje je funkcija u kojoj se i učitava sama tekstura. U trenutku njihovog postavljanja nije neophodno da bude učitana sama bitmapa, što se može videti i u sledećem primeru, već samo kreirana i aktivirana data tekstura.

```
void CGLTexture::LoadFromFile(CString texFileName)
{
    if(m_ID != 0) Release();

    // Alokacija ID-a i kreiranje teksture
    glGenTextures(1, &m_ID);
    glBindTexture(GL_TEXTURE_2D, m_ID);

    // Definisanje parametara teksture
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);

    // Ucitavanje bitmape
    AUX_RGBImageRec *TextureImage;
    TextureImage = auxDIBImageLoad(texFileName);
    // ...
}
```

Učitavanje tekstura

Učitavanje tekstura jeste vremenski zahtevna operacija, ali ukoliko su teksture nepromenljive, to se izvršava samo jednom. Kao što je već rečeno, pozive operacija koje se izvršavaju samo jednom poželjno je smestiti u funkciju `CGLRenderer::PrepareScene()`.

```
void CGLRenderer::PrepareScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClearColor (1.0, 1.0, 1.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    CGLTexture::PrepareTexturing();
    m_tex1.LoadFromFile(_T("Box.bmp"));
    //-----
```

Promenljiva `m_tex1`, iz prethodnog primera, predstavlja instancu klase `CGLTexture`, i deklarisana je u zaglavlju klase `CGLRender`. Ukoliko postoji više tekstura, njihovo učitavanje i inicijalizacija izvršilo bi se pozivom `LoadFromFile()` funkcije za svaki od objekata tekstura.

Teksturne koordinate temena

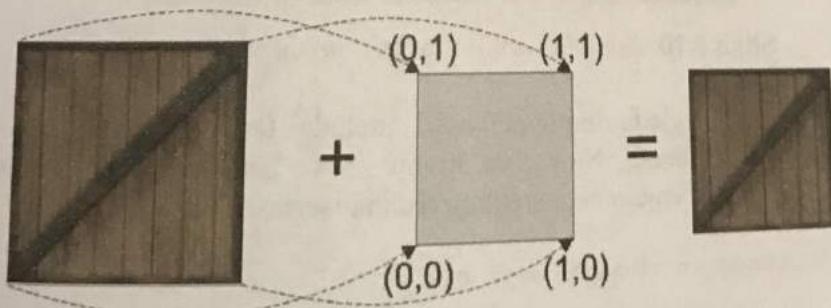
2D tekstura je slika koja se „lepi“ na površinu objekta, kako bi se povećala njegova realističnost bez usložnjavanja geometrije, na način sličan lepljenju tapeta na zid. Način lepljena zavisi isključivo od objekta, i mora biti prilagođen efektu koji želimo da postignemo.

Poznato je da se sve karakteristike objekata definišu u temenima (na primer, položaj, boja, materijal, normale, itd). Tako definisane karakteristike se zatim interpoliraju preko čitave površine odgovarajućih primitiva, kako bi se odredila boja svakog piksela. Ideničan je slučaj i sa teksturama. Svakom temenu treba zadati koordinate u okviru teksture, a tekstura će se poput beskonačno elastičnog materijala, „razvući“ po čitavoj površini primitive. Koordinate temena u okviru teksture zadaju se pozivom funkcije:

```
glTexCoord2f(coords, coordT);
```

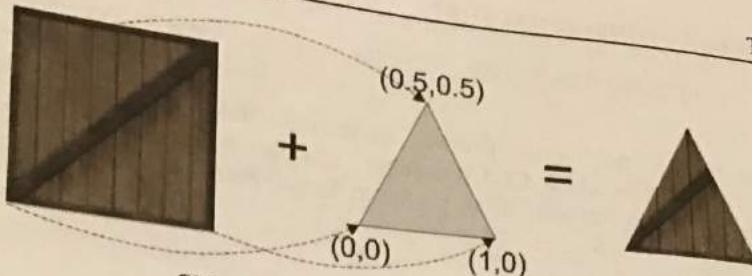
gde je `coordS` teksturna koordinata po S-pravcu, a `coordT` teksturna koordinata po T-pravcu. Funkcija `glTexCoord*` ima mnogo oblika, ali ćemo u nastavku poglavlja koristiti samo oblike vezane za 2D teksture.

Na slici 5.8 prikazan je najjednostavniji način primene teksture. Ukoliko je poligon na koji želimo primeniti čitavu teksturu zapravo kvadrat, polazeći od donjeg levog temena, i krećući se u pozitivnom smeru, temenima je potrebno dodeliti sledeće teksturne koordinate: (0,0), (1,0), (1,1) i (0,1).



Slika 5.8. Primer teksturisanja kvadrata

Na slici 5.9 prikazan je primer dodele teksture trouglu. Obzirom da tekstura uvek mora biti pravougaonog oblika, jedan od načina da se izbegne izobličenje slike jeste određivanje teksturnih koordinata tako da se zadrži odnos rastojanja između temena.



Slika 5.9. Primer teksturisanja trougla

Vrlo često se više tekstura smešta u jednu veću, kako bi se smanjio broj aktiviranja tekstura. Jedna velika texatura koja sadrži više manjih često se naziva **atlas tekstura**. Mnoge savremene grafičke kartice postižu veće brzine prikaza, prilikom upotrebe atlasa umesto mnoštva pojedinačnih tekstura. Na slici 5.10 prikazana je tekstura koja se može koristiti pri modeliranju nekog egipatskog hrama. Donji levi deo teksture pogodan je za kameni pod, gornji desni za stubove, gornji levi za različite ornamente na zidovima i predmetima, itd.



Slika 5.10. Kombinovanje više tekstura u okviru jedne

Pogledajmo sada kako izgleda implementacija metoda **DrawCube()**, koja uključuje i normale i teksturne koordinate. Na svaku stranu kocke primenićemo tekstuру na način prikazan na slici 5.8, korišćenjem neposrednog načina iscrtavanja.

```
void CGLRenderer::DrawCube(double a)
{
    glBegin(GL_QUADS);

    // Prednja stranica
    glNormal3d( 0.0, 0.0, 1.0);
    glTexCoord2f(0.0, 1.0);
    glVertex3d(-a/2, a/2, a/2);
    glTexCoord2f(0.0, 0.0);
    glVertex3d(-a/2,-a/2, a/2);
    glTexCoord2f(1.0, 0.0);
```

```

glVertex3d( a/2,-a/2, a/2);
glTexCoord2f(1.0, 1.0);
glVertex3d( a/2, a/2, a/2);
// Desna stranica
glNormal3d( 1.0, 0.0, 0.0);
glTexCoord2f(0.0, 1.0);
glVertex3d( a/2, a/2, a/2);
glTexCoord2f(0.0, 0.0);
glVertex3d( a/2,-a/2, a/2);
glTexCoord2f(1.0, 0.0);
glVertex3d( a/2,-a/2,-a/2);
glTexCoord2f(1.0, 1.0);
glVertex3d( a/2, a/2,-a/2);

// Zadnja stranica
glNormal3d( 0.0, 0.0,-1.0);
glTexCoord2f(0.0, 1.0);
glVertex3d( a/2, a/2,-a/2);
glTexCoord2f(0.0, 0.0);
glVertex3d( a/2,-a/2,-a/2);
glTexCoord2f(1.0, 0.0);
glVertex3d(-a/2,-a/2,-a/2);
glTexCoord2f(1.0, 1.0);
glVertex3d(-a/2, a/2,-a/2);

// Leva stranica
glNormal3d(-1.0, 0.0, 0.0);
glTexCoord2f(0.0, 1.0);
glVertex3d(-a/2, a/2,-a/2);
glTexCoord2f(0.0, 0.0);
glVertex3d(-a/2,-a/2,-a/2);
glTexCoord2f(1.0, 0.0);
glVertex3d(-a/2,-a/2, a/2);
glTexCoord2f(1.0, 1.0);
glVertex3d(-a/2, a/2, a/2);

// Gornja stranica
glNormal3d( 0.0, 1.0, 0.0);
glTexCoord2f(0.0, 0.0);
glVertex3d(-a/2, a/2, a/2);
glTexCoord2f(1.0, 0.0);
glVertex3d( a/2, a/2, a/2);
glTexCoord2f(1.0, 1.0);
glVertex3d( a/2, a/2,-a/2);
glTexCoord2f(0.0, 1.0);
glVertex3d(-a/2, a/2,-a/2);

// Donja stranica
glNormal3d( 0.0,-1.0, 0.0);
glTexCoord2f(0.0, 0.0);
glVertex3d(-a/2,-a/2,-a/2);
glTexCoord2f(1.0, 0.0);
glVertex3d( a/2,-a/2,-a/2);
glTexCoord2f(1.0, 1.0);
glVertex3d( a/2,-a/2, a/2);
glTexCoord2f(0.0, 1.0);
glVertex3d(-a/2,-a/2, a/2);

glEnd();
}

```

tekstura.
Mnoge
umesto
ristiti pri
eni pod,
, itd.

dajuće i
na način

Posredni način iscrtavanja podrazumeva korišćenje polja temena. Polja temena omogućuju znatno brži način iscrtavanja, pogotovo pri radu sa složenim objektima. Kocka ne spada u složene objekte, ali je pogodna za demonstraciju korišćenja polja temena.

Najpre je potrebno definisati polja koja će sadržati podatke. Obzirom da želimo imati prostorne koordinate, normale i teksturne koordinate, potrebna su nam tri polja:

- *vert* – polje prostornih koordinata,
- *norm* – polje normala i
- *texc* – polje teksturnih koordinata.

Svi atributi temena mogu biti smešani i samo u jedno polje, kao što je prikazano u glavi 2, ali ćemo zbog veće razumljivosti programskog koda ipak za svaki tip atributa alocirati zasebno polje.

Svako teme kocke učestvuje u formiranju tri stranice, i u svakoj stranici ima druge teksturne koordinate i normalu. Zbog toga ćemo umnožiti temena, tako da imamo zasebno teme za svako pojavljivanje u nekoj od stranica, i koristiti **metod sekvencijalnog pristupa elementima polja** (tj. poziv funkcije `glDrawArrays()`). Broj temena kocke je 8, svako učestvuje u formiranju 3 stranice, i definisano je sa po 3 koordinate. Dakle, potrebno je 72 *float* broja za specificiranje koordinata temena kocke. Normale takođe zahtevaju 3 komponente, pa je i vektor normala istih dimenzija kao vektor koordinata. Teksturne koordinate zadaju se sa po 2 *float* broja, tako da je vektor *texc* manji od prethodna dva, i ima 48 komponenti.

```
float vert[72];
float norm[72];
float texc[48];
```

Sva polja je potrebno popuniti pre iscrtavanja. Kako je kocka nepromenljive geometrije, popuna polja vrednostima biće obavljena u funkciji `PrepareVACube()`, koja će biti pozvana u okviru funkcije `CGLRenderer::PrepareScene()`.

```
void CGLRenderer::PrepareVACube(float a)
{
    // Prostorne koordinate temena
    vert[0] = -a/2; vert[1] = -a/2; vert[2] = a/2; // vert0
    vert[3] = a/2; vert[4] = -a/2; vert[5] = a/2; // vert1
    vert[6] = a/2; vert[7] = a/2; vert[8] = a/2; // vert2
    vert[9] = -a/2; vert[10] = a/2; vert[11] = a/2; // vert3
    vert[12] = a/2; vert[13] = -a/2; vert[14] = a/2; // vert4
    vert[15] = a/2; vert[16] = -a/2; vert[17] = -a/2; // vert5
    vert[18] = a/2; vert[19] = a/2; vert[20] = -a/2; // vert6
    vert[21] = a/2; vert[22] = a/2; vert[23] = a/2; // vert7
    vert[24] = -a/2; vert[25] = a/2; vert[26] = -a/2; // vert8
    vert[27] = a/2; vert[28] = a/2; vert[29] = -a/2; // vert9
    vert[30] = a/2; vert[31] = -a/2; vert[32] = -a/2; // vert10
    vert[33] = -a/2; vert[34] = -a/2; vert[35] = -a/2; // vert11
    vert[36] = -a/2; vert[37] = -a/2; vert[38] = a/2; // vert12
    vert[39] = -a/2; vert[40] = a/2; vert[41] = a/2; // vert13
    vert[42] = -a/2; vert[43] = a/2; vert[44] = -a/2; // vert14
    vert[45] = -a/2; vert[46] = -a/2; vert[47] = -a/2; // vert15
```

EKSTURE

omogućuju
ne spada u
elimo imaj
lo u glavu,
uta alocirati
ima druge
imo zasebno
og pristupa
je 8, svako
trebno je 72
zahtevaju 3
1. Teksturne
hodna dva, i
geometrije,
koja će biti
vert0
vert1
vert2
vert3
vert4
vert5
vert6
vert7
vert8
vert9
vert10
vert11
vert12
vert13
vert14
vert15
vert16
vert17
vert18
vert19
vert20
vert21
vert22
vert23
vert24
vert25
vert26
vert27
vert28
vert29
vert30
vert31
vert32
vert33
vert34
vert35
vert36
vert37
vert38
vert39
vert40
vert41
vert42
vert43
vert44
vert45
vert46
vert47
vert48
vert49
vert50
vert51
vert52
vert53
vert54
vert55
vert56
vert57
vert58
vert59
vert60
vert61
vert62
vert63
vert64
vert65
vert66
vert67
vert68
vert69
vert70
vert71

OPENGL - FIKSNA FUNKCIONALNOST

157

```

vert[48] = -a/2; vert[49] = a/2; vert[50] = -a/2; // vert7
vert[51] = -a/2; vert[52] = a/2; vert[53] = a/2; // vert3
vert[54] = a/2; vert[55] = a/2; vert[56] = a/2; // vert2
vert[57] = a/2; vert[58] = a/2; vert[59] = -a/2; // vert6
vert[60] = -a/2; vert[61] = -a/2; vert[62] = a/2; // vert0
vert[63] = -a/2; vert[64] = -a/2; vert[65] = -a/2; // vert4
vert[66] = a/2; vert[67] = -a/2; vert[68] = -a/2; // vert5
vert[69] = a/2; vert[70] = -a/2; vert[71] = a/2; // vert1

// Normale
norm[0] = 0.0; norm[1] = 0.0; norm[2] = 1.0;
norm[3] = 0.0; norm[4] = 0.0; norm[5] = 1.0;
norm[6] = 0.0; norm[7] = 0.0; norm[8] = 1.0;
norm[9] = 0.0; norm[10] = 0.0; norm[11] = 1.0;
norm[12] = 1.0; norm[13] = 0.0; norm[14] = 0.0;
norm[15] = 1.0; norm[16] = 0.0; norm[17] = 0.0;
norm[18] = 1.0; norm[19] = 0.0; norm[20] = 0.0;
norm[21] = 1.0; norm[22] = 0.0; norm[23] = 0.0;
norm[24] = 0.0; norm[25] = 0.0; norm[26] = -1.0;
norm[27] = 0.0; norm[28] = 0.0; norm[29] = -1.0;
norm[30] = 0.0; norm[31] = 0.0; norm[32] = -1.0;
norm[33] = 0.0; norm[34] = 0.0; norm[35] = -1.0;
norm[36] = -1.0; norm[37] = 0.0; norm[38] = 0.0;
norm[39] = -1.0; norm[40] = 0.0; norm[41] = 0.0;
norm[42] = -1.0; norm[43] = 0.0; norm[44] = 0.0;
norm[45] = -1.0; norm[46] = 0.0; norm[47] = 0.0;
norm[48] = 0.0; norm[49] = 1.0; norm[50] = 0.0;
norm[51] = 0.0; norm[52] = 1.0; norm[53] = 0.0;
norm[54] = 0.0; norm[55] = 1.0; norm[56] = 0.0;
norm[57] = 0.0; norm[58] = 1.0; norm[59] = 0.0;
norm[60] = 0.0; norm[61] = -1.0; norm[62] = 0.0;
norm[63] = 0.0; norm[64] = -1.0; norm[65] = 0.0;
norm[66] = 0.0; norm[67] = -1.0; norm[68] = 0.0;
norm[69] = 0.0; norm[70] = -1.0; norm[71] = 0.0;

// Teksturne koordinate
texc[0] = 0.0; texc[1] = 0.0;
texc[2] = 1.0; texc[3] = 0.0;
texc[4] = 1.0; texc[5] = 1.0;
texc[6] = 0.0; texc[7] = 1.0;
texc[8] = 0.0; texc[9] = 0.0;
texc[10] = 1.0; texc[11] = 0.0;
texc[12] = 1.0; texc[13] = 1.0;
texc[14] = 0.0; texc[15] = 1.0;
texc[16] = 0.0; texc[17] = 0.0;
texc[18] = 1.0; texc[19] = 0.0;
texc[20] = 1.0; texc[21] = 1.0;
texc[22] = 0.0; texc[23] = 1.0;
texc[24] = 0.0; texc[25] = 0.0;
texc[26] = 1.0; texc[27] = 0.0;
texc[28] = 1.0; texc[29] = 1.0;
texc[30] = 0.0; texc[31] = 1.0;

```

TEKSTURE

```
texc[32] = 0.0; texc[33] = 0.0;  
texc[34] = 1.0; texc[35] = 0.0;  
texc[36] = 1.0; texc[37] = 1.0;  
texc[38] = 0.0; texc[39] = 1.0;  
texc[40] = 0.0; texc[41] = 0.0;  
texc[42] = 1.0; texc[43] = 0.0;  
texc[44] = 1.0; texc[45] = 1.0;  
texc[46] = 0.0; texc[47] = 1.0;  
}
```

Samo iscrtavanje kocke biće definisano funkcijom **DrawVACube()**, tako što se najpre definišu pokazivači na podatke (**gl*Pointer()** funkcije), zatim aktiviraju odgovarajuća polja atributa temena (**glEnableClientState()** funkcije), i na kraju iscrta kocka pozivom samo jedne funkcije **glDrawArrays()**. Naravno, po završetku, poželjno je deaktivirati sva polja atributa temena koja nisu više potrebna.

```
void CGLRenderer::DrawVACube()  
{  
    glVertexPointer(3, GL_FLOAT, 0, vert);  
    glNormalPointer(GL_FLOAT, 0, norm);  
    glTexCoordPointer(2, GL_FLOAT, 0, texc);  
  
    glEnableClientState(GL_VERTEX_ARRAY);  
    glEnableClientState(GL_NORMAL_ARRAY);  
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);  
  
    glDrawArrays(GL_QUADS, 0, 24);  
  
    glDisableClientState(GL_VERTEX_ARRAY);  
    glDisableClientState(GL_NORMAL_ARRAY);  
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);  
}
```

OPENGL - F
teksture, Imp
glBindTextu
voi
{
}
Postupak akt
drawSceneC
voi
{
}

Kao rezulta

Aktiviranje tekstura

Inicijalno, teksturisanje je isključeno (isto kao i osvetljenje). Uključivanje 2D teksturisanja ostvaruje se pozivom funkcije:

```
glEnable(GL_TEXTURE_2D);
```

Poziv funkcije **glEnable()** može se ostvariti u okviru funkcije **CGLRenderer::PrepareScene()**, ukoliko su svi objekti u sceni teksturisani, ili u funkciji **CGLRenderer::DrawScene()**, neposredno pre crtanja teksturisanih objekata, ukoliko postoje i objekti bez tekstura.

Osim globalnog uključivanja tekstura (pozivom funkcije **glEnable()**), potrebno je aktivirati i odgovarajuću teksturu pre crtanja objekta koji je koristi. To se ostvaruje pozivom funkcije:

```
glBindTexture(GL_TEXTURE_2D, ID);
```

gde je *ID* identifikator date teksture. Da bismo olakšali rad sa teksturama, i bolje podržali objektno-orientisani model teksture, klasi **CGLTexture** dodaćemo metod za selekciju date

Brisan

Kada pre
oslobada

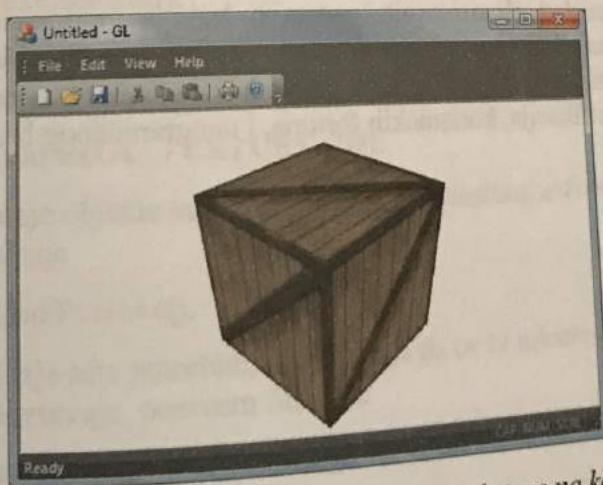
teksture. Implementacija metoda je vrlo jednostavna, i svodi se samo na poziv funkcije `glBindTexture()`, sa identifikatorom koji je već zapamćen kao atribut klase.

```
void CGLTexture::Select()
{
    if(m_ID)
        glBindTexture(GL_TEXTURE_2D, m_ID);
}
```

Postupak aktiviranja i primene tekture pogledajmo na primeru funkcije `CGLRenderer::DrawScene()`, koja iscrtava teksturisanu kocku.

```
void CGLRenderer::DrawScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(3.0, 3.0, 3.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glEnable(GL_TEXTURE_2D);
    m_text1.Select();
    DrawCube(2.0);
    glDisable(GL_TEXTURE_2D);
    //-----
    glFlush();
    SwapBuffers(pDC->m_hDC);
    wglMakeCurrent(NULL, NULL);
}
```

Kao rezultat dobija se scena prikazana na slici 5.11.



Slika 5.11. Konačni rezultat primene tekture na kocku

Brisanje tekstura

Kada prestane potreba za nekom teksturom, potrebno je datu teksturu obrisati. Na taj način oslobađa se prostor u memoriji grafičke kartice za prihvatanje drugih tekstura. Mada

OpenGL poseduje metod za realokaciju tekstura koje nisu skoro korišćene, resursi ostaju zauzeti sve dok se ne pozove funkcija:

```
void glDeleteTextures(GLsize n, const GLuint *textureNames);
```

Parametar *n* određuje koliko se tekstura briše, a identifikatori tih tekstura prosledjuju se u polju *textureNames*. Ako se pozove brisanje teksture koja je trenutno selektovana, aktivira se neimenovana podrazumevana tekstura, čiji je identifikator 0. A ako se pokuša brisati nepostojeće teksture ili teksture sa identifikatorom 0, odgovarajući poziv funkcije **glDeleteTextures()** biće ignoriran.

Dopunimo implementaciju klase **CGLTexture** dodavanjem metoda koji će vršiti brisanje date teksture.

```
void CGLTexture::Release()
{
    if (m_ID)
    {
        glDeleteTextures(1, &m_ID);
        m_ID = 0;
    }
}
```

Metod **CGLTexture::Release()** najpre proverava da li je data tekstura formirana i ima svoj identifikacioni broj. Ako je to ispunjeno, poziva se funkcija **glDeleteTextures()** samo za dati identifikator, a atribut **m_ID** postavlja se na nulu. Postavljanjem identifikatora na nulu olakšali smo ispitivanje postojanja teksture. Ukoliko je **m_ID** različit od nule, podrazumevaćemo da je tekstura regularno kreirana, a ako je jednak nuli, da još nije inicijalizovana.

Ovim je završen pregled elementarnih operacija koje se mogu izvesti u OpenGL-u korišćenjem fiksne funkcionalnosti. Međutim, to je samo vrh ledenog brega. Na vama je da nastavite dalje upoznavanje OpenGL-a, i generalno računarske grafike, kroz mnoštvo knjiga, zvaničnih specifikacija, korisničkih foruma, i neograničenog broja resursa dostupnih preko Interneta.

Kratak pregled gradiva

Prilikom rada sa teksturama, potrebno je:

- (opciono) definisati poravnanje teksture po bajtovima, pozivom funkcije
 - o `glPixelStorei()`,
- definisati način teksturisanja (kako tekstura utiče na promenu boje fragmenta), pozivom funkcije
 - o `glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, mode)`,
- za svaku od tekstura kreirati objekat
 - o određivanjem identifikatora – `glGenTextures()`,
 - o inicijalnim aktiviranjem (povezivanjem) teksture – `glBindTexture()`,
 - o postavljanjem parametara – `glTexParameter()`, i
 - o učitavanjem same teksture – `glTexImage2D()` ili `gluBuild2DMipmaps()`,
- za učitavanje podataka iz BMP datoteka, može se koristiti funkcija `auxDIBImageLoad()`, sadržana u pomoćnoj biblioteci (`glaux.lib`),
- prilikom definisanja temena, potrebno je zadati teksturne koordinate svakog od njih, pozivom funkcije
 - o `glTexCoord2f()`,
- pre iscrtavanja svih objekata koji koriste teksture, potrebno je uključiti teksturisanje, pozivom funkcije
 - o `glEnable(GL_TEXTURE_2D)`,
- a pre iscrtavanja objekta koji koristi određenu teksturu, aktivirati datu teksturu pozivom funkcije
 - o `glBindTexture()`,
- ako teksturisanje nije potrebno, isključiti ga da ne bi uticalo na objekte koji se nakon toga iscrtavaju, pozivom funkcije
 - o `glDisable(GL_TEXTURE_2D)`,
- po prestanku potrebe za nekom teksturom, oslobođiti resurse koje ona zauzima pozivom funkcije
 - o `glDeleteTextures()`.

Funkcije korišćene u ovoj glavi

void glPixelStore{if}(GLenum pname, TYPE param);

pname – naziv parametra (postoji 12 različitih parametara:

GL_PACK_SWAP_BYTES, GL_PACK_LSB_FIRST,
 GL_PACK_ROW_LENGTH, GL_PACK_SKIP_PIXELS,
 GL_PACK_SKIP_ROWS, GL_PACK_ALIGNMENT,
 GL_UNPACK_SWAP_BYTES, GL_UNPACK_LSB_FIRST,
 GL_UNPACK_ROW_LENGTH, GL_UNPACK_SKIP_PIXELS,
 GL_UNPACK_SKIP_ROWS, GL_UNPACK_ALIGNMENT)

param – vrednost parametra

Postavlja modove smeštanja piksela. Korišćen je samo parametar GL_UNPACK_ALIGNMENT za postavljanje poravnjanja nezapakovanih tekstura.

void glTexEnv{if}(GLenum target, GLenum pname, GLint param);

target – mora biti postavljen na GL_TEXTURE_ENV

pname – mora biti postavljen na GL_TEXTURE_ENV_MODE

param – način primene teksture, može imati sledeće vrednosti:

GL_REPLACE – potpuno menja boju fragmenta podacima iz tekture,
 GL_DECAL – mešanje boje fragmenta i tekture definisano je
 providnošću tekture,
 GL_MODULATE – boja fragmenata je „modulisana“ teksturom i
 GL_BLEND – boja se dobija mešanjem boje fragmenta, tekture i
 posebno zadate boje.

Definiše način primene teksture.

void glTexEnv{if}v(GLenum target, GLenum pname, GLint param);

target – mora biti postavljen na GL_TEXTURE_ENV

pname – mora biti postavljen na GL_TEXTURE_ENV_COLOR

param – definiše RGBA vrednost boje koja se koristi za mešanje.

Definiše boju koja se koristi u modu GL_BLEND.

void glGenTextures(GLsizei n, GLuint *textureNames);

n – broj identifikatora koji se vraćaju
textureNames – polje slobodnih identifikatora

Vraća *n* trenutno slobodnih identifikatora, u obliku neoznačenih celih brojeva, koje smešta u polje *textureNames*.

void glBindTexture(GLenum target, GLuint textureName);

163

target – tip teksture koja se aktivira. Može imati jednu od sledećih vrednosti:

GL_TEXTURE_1D – 1D tekstura,
 GL_TEXTURE_2D – 2D tekstura,
 GL_TEXTURE_3D – 3D tekstura,
 GL_TEXTURE_CUBE_MAP – kubna tekstura,

textureName – identifikator teksture koja se aktivira.

Kada se prvi put pozove za odgovarajući identifikator, dobijen pozivom funkcije **glGenTextures()**, objekat teksture se kreira i aktivira (selektuje). Sve kasnije modifikacije parametara odnosiće se na selektovani objekat. Kada se pozove za prethodno kreirani objekat, onda samo aktivira dati objekat. A kada se pozove za vrednost 0, isključuje rad sa objektima tekstura i aktivira neimenovanu „podrazumevanu“ teksturu.

**void glTexImage2D(GLenum target, GLint level, GLint internalFormat,
 GLsizei width, GLsizei height, GLint border,
 GLenum format, GLenum type, const GLvoid *texels);**

target – tip teksture (GL_TEXTURE_2D),

level – nivo teksture koji se popunjava,

internalFormat – broj komponenata jednog teksela (1, 2, 3, 4 ili jednu od 55 drugih predefinisanih vrednosti), najčešće koristimo GL_RGB (3) ili GL_RGBA (4)

width – širina u tekselima,

height – visina u tekselima,

border – širina okvira,

format – format teksela (GL_RGB, GL_RGBA, ...),

type – tip podataka koji formiraju teksel (GL_UNSIGNED_BYTE, ...),

texels – pokazivač na bafer sa podacima, koje treba preneti u teksturu.

Definiše format teksture i prenosi podatke.

**int gluBuild2DMipmaps(GLenum target, GLint internalFormat,
 GLint width, GLint height,
 GLenum format, GLenum type, void *texels);**

target – tip teksture (GL_TEXTURE_2D),

internalFormat – broj komponenti jednog teksela (vidi **glTexImage2D()**)

width, height – širina i visina u tekselima,

format – format teksela (GL_RGB, GL_RGBA, ...),

type – tip podataka koji formiraju teksel (GL_UNSIGNED_BYTE, ...),

texels – pokazivač na bafer sa podacima, koje treba preneti u teksturu.

Definiše format teksture, prenosi podatke i formira kompletну mipmapu. Automatski prilagođava najviši nivo detalja tako da dimenzije budu stepen broja 2.

TEKSTURE

AUX_RGBImageRec* auxDIBImageLoad(LPCWSTR fileName);

fileName – naziv (BMP) datoteke iz koje se učitava slika

Učitava sliku u strukturu *AUX_RGBImageRec* koja ima tri atributa:

- *data* – bafer u kome je smeštena sadržina slike (pikseli),
- *sizeX* – širina slike i
- *sizeY* – visina slike.

Ova funkcija pripada pomoćnoj biblioteci *glaux.lib*, koja se mora uključiti u projekat da bi funkcija bila dostupna.

void glTexParameteri(GLenum target, GLenum pname, TYPE param);

target – tip teksture (GL_TEXTURE_2D),

pname – naziv parametra koji se postavlja (GL_TEXTURE_MAG_FILTER, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_WRAP_S,

GL_TEXTURE_WRAP_T),

param – vrednost na koju se postavlja parametar (GL_NEAREST,

GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST,

GL_NEAREST_MIPMAP_LINEAR,

GL_LINEAR_MIPMAP_NEAREST,

GL_LINEAR_MIPMAP_LINEAR, GL_CLAMP, GL_REPEAT).

Definiše različite parametre tekstura.

void glGetIntegerv(GLenum pname, GLint * params);

pname – ime parametra koji je potrebno očitati. U ovom poglavlju korišćen je samo sledeći parametar:

GL_MAX_TEXTURE_SIZE – maksimalna dimenzija teksture u tekselima.

params – očitana vrednost (ili vrednosti).

Navedene funkcije služe za očitavanje različitih parametara i stanja u kome se nalazi OpenGL *rendering context*. U OpenGL-u 1.1 mogu se na ovaj način očitati preko 200 različitih vrednosti.

void glTexCoord{1234}{sifd}(TYPE coords);

void glTexCoord{1234}{sifd}v(const TYPE *coords);

coords – teksturne koordinate datog temena

Postavlja teksturne koordinate temena čija definicija sledi nakon poziva ove funkcije.

void glEnable(GLenum cap) / void glDisable(GLenum cap)

cap – stanje koje se uključuje/isključuje. Može imati preko 40 različitih vrednosti, ne računajući podoblike. U ovoj glavi koriste se samo: GL_TEXTURE_2D – uključuje/isključuje 2D teksture.

Ovaj par funkcija uključuje/isključuje odgovarajuće stanje u OpenGL *rendering context-u*.

void glDeleteTextures(GLsize n, const GLuint *textureNames);

n – broj tekstura koje se brišu

textureNames – polje identifikatora tekstura.

Briše teksture sa identifikatorima koji su navedeni u polju *textureNames*.



Zadatak

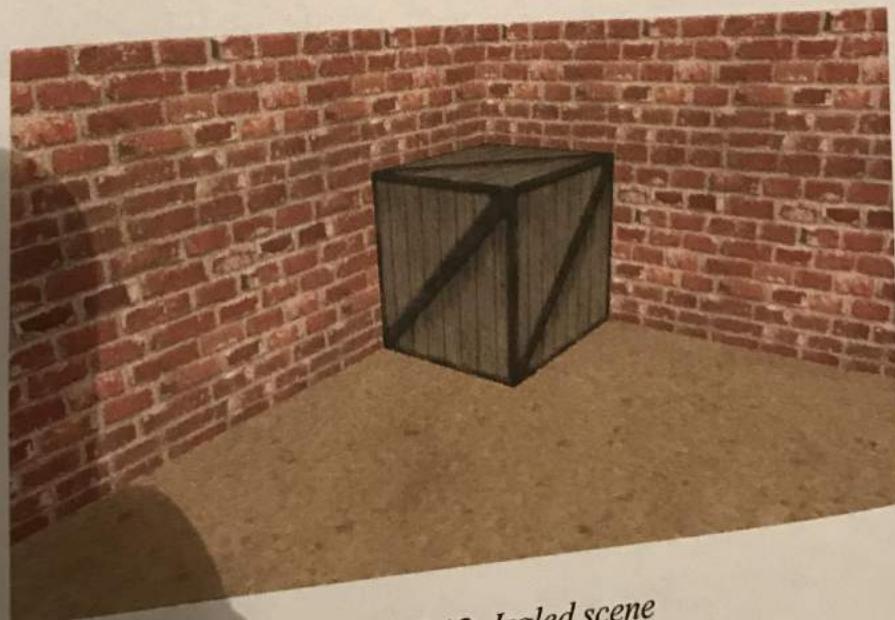
Napisati sledeće funkcije:

- void **DrawCube**(double dSize) – koja iscrtava kocku stranice $dSize$ sa centrom u koordinatnom početku (za sva temena definisati odgovarajuće prostorne koordinate, normale i teksturne koordinate) i
- void **DrawWall**(double sizeX, double sizeY, int repX, int repY) – koja iscrtava zid dimenzija $sizeX \times sizeY$, sa teksturnim koordinatama koje omogućuju da se tekstura ponavlja $repX$ puta po S-pravcu i $repY$ po T-pravcu.

U okviru funkcije **CGLRenderer::DrawScene()** formirati scenu koja se sastoji od:

- dva vertikalna zida dimenzija 8×8 , na kojima se tekstura cigala (**briks.bmp**) ponavlja $4 \times$ po svakom pravcu (zidovi se spajaju na Y-osi),
- horizontalnog zida dimenzija 8×8 , koji služi kao pod, sa teksturom zemlje (**gnd.bmp**) koja se ponavlja samo jednom (sa vertikalnim zidovima spaja se na X i Z-osi), i
- kocke, dimenzija 2×2 , sa teksturom **box.bmp**, čije se jedno donje teme nalazi na spoju tri zida (u koordinatnom početku).

Da bi se dobio pogled kao na slici 5.12, potrebno je udaljiti kameru za 10 jedinica od koordinatnog početka i zarotirati je za -20° oko X-ose i 40° oko Y-ose.



Slika 5.12. Izgled scene

Rešenje

Napomena: U rešenju koje sledi nedostaje implementacija metoda DrawCube(), obzirom da se već javlja u okviru glave.

```

void CGLRenderer::DrawWall(double sizeX, double sizeY,
                           int repX, int repY)
{
    glBegin(GL_QUADS);
        glTexCoord2f(0.0, 0.0);
        glVertex3f(-(sizeX/2), -(sizeY/2), 0.0);
        glTexCoord2f(repX, 0.0);
        glVertex3f( (sizeX/2), -(sizeY/2), 0.0);
        glTexCoord2f(repX, repY);
        glVertex3f( (sizeX/2), (sizeY/2), 0.0);
        glTexCoord2f(0.0, repY);
        glVertex3f(-(sizeX/2), (sizeY/2), 0.0);
    glEnd();
}

void CGLRenderer::PrepareScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClearColor (1.0, 1.0, 1.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    CGLTexture::PrepareTexturing(false);
    m_tex1.LoadFromFile(_T("Box.bmp"));
    m_tex2.LoadFromFile(_T("bricks.bmp"));
    m_tex3.LoadFromFile(_T("gnd.bmp"));
    //-----
    wglMakeCurrent(NULL, NULL);
}

void CGLRenderer::DrawScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity ();
    glTranslatef(0.0, 0.0, -10.0);
    glRotatef(20.0, 1.0, 0.0, 0.0);
    glRotatef(-40.0, 0.0, 1.0, 0.0);

    glEnable(GL_TEXTURE_2D);
    m_tex2.Select();
    glPushMatrix();
        glTranslatef(4.0, 4.0, 0.0);
        DrawWall(8.0, 8.0, 4, 4);
    glPopMatrix();

    glPushMatrix();
        glTranslatef(0.0, 4.0, 4.0);
        glRotatef(90.0, 0.0, 1.0, 0.0);
        DrawWall(8.0, 8.0, 4, 4);
    glPopMatrix();

    m_tex3.Select();
}

```

```
glPushMatrix();
    glTranslatef(4.0, 0.0, 4.0);
    glRotatef(-90.0, 1.0, 0.0, 0.0);
    DrawWall(8.0, 8.0, 1, 1);
glPopMatrix();

m_tex1.Select();
glPushMatrix();
    glTranslatef(1.0, 1.0, 1.0);
    DrawCube(2.0);
glPopMatrix();

glDisable(GL_TEXTURE_2D);
//-----
glFlush();
SwapBuffers(pDC->m_hDC);
wglMakeCurrent(NULL, NULL);

}

void CGLRenderer::Reshape(CDC *pDC, int w, int h)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
//-
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective(40, (double)w/(double)h, 1, 100);
    glMatrixMode (GL_MODELVIEW);
//-
    wglMakeCurrent(NULL, NULL);
}

void CGLRenderer::DestroyScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
//-
    m_tex1.Release();
    m_tex2.Release();
    m_tex3.Release();
//-
    wglMakeCurrent(NULL, NULL);
    if(m_hrc)
    {
        wglDeleteContext(m_hrc);
        m_hrc = NULL;
    }
}
```

Dodatak A – Vektorska aritmetika

Vektori su geometrijski objekti koji imaju intenzitet, pravac i smer. Položaj temena u Euklidovom prostoru predstavlja se vektorom položaja u odnosu na koordinatni početak. Obzirom da se svi objekti u računarskoj grafici definisu skupom svojih temena, vektori čine nezaobilaznu komponentu svake vizuelizacije. Vektor u n-dimenzionalnom prostoru zadat je poljem koordinata $[v.x \ v.y \ v.z]$. Npr. vektor v u 3D prostoru zadat je poljem koordinata $[v.x \ v.y \ v.z]$. Pravac vektora određen je pravom koja prolazi kroz koordinatni početak i tačku definisanu zadatim koordinata, a usmeren je od koordinatnog početka ka zadatoj tački. U ovom dodatku ukratko ćemo navesti osnovne operacije sa vektorima i prikazati implementaciju klase `vec3`, za manipulaciju trodimenzionalnim vektorima.

Množenje vektora skalarom

Množenjem vektora v skalarom a , u oznaci $a * v$, dobija se vektor u čije se koordinate računaju po sledećoj formuli:

$$u.x = a * v.x$$

$$u.y = a * v.y$$

$$u.z = a * v.z$$

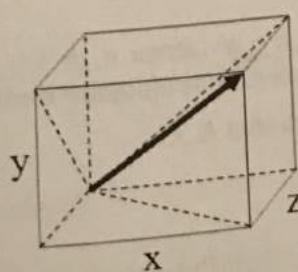
Množenjem vektora skalarom a , dužina vektora se povećava a puta.

Intenzitet vektora

Intenzitet (dužina) vektora v , u oznaci $|v|$, je skalar koji se računa po sledećoj formuli:

$$|v| = \sqrt{v.x^2 + v.y^2 + v.z^2}$$

Ovo je Pitagorina teorema primenjena na 3D prostor, što se može videti i na osnovu slike A.1.



Slika A.1. Dužina vektora

Normalizacija

Normalizacija je postupak svođenja vektora na jediničnu dužinu, a vrši se deljenjem vektora njegovim intenzitetom.

$$\mathbf{v}' = \mathbf{v} / |\mathbf{v}|$$

odnosno, tako što se svaka koordinata podeli dužinom vektora.

$$v.x = v.x / |\mathbf{v}|$$

$$v.y = v.y / |\mathbf{v}|$$

$$v.z = v.z / |\mathbf{v}|$$

Deljenje intenzitetom vektora je zapravo množenje skalarom, čija je vrednost obrnuto proporcionalna dužini vektora ($a = 1 / |\mathbf{v}|$).

Sabiranje i oduzimanje vektora

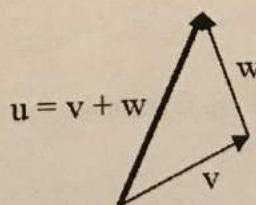
Zbir dva vektora \mathbf{v} i \mathbf{w} , u označi $\mathbf{v} + \mathbf{w}$, je vektor \mathbf{u} čije se koordinate računaju po sledećoj formuli:

$$u.x = v.x + w.x$$

$$u.y = v.y + w.y$$

$$u.z = v.z + w.z$$

Na slici A.2 grafički je prikazan rezultat sabiranja dva vektora \mathbf{v} i \mathbf{w} .

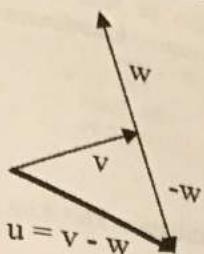


Slika A.2. Zbir dva vektora

Oduzimanje dva vektora \mathbf{v} i \mathbf{w} može se svesti na sabiranje, ukoliko invertujemo vektor \mathbf{w} , tj. pomnožimo skalarom -1. Naime, važi:

$$\mathbf{v} - \mathbf{w} = \mathbf{v} + (-1 * \mathbf{w}) = \mathbf{v} + (-\mathbf{w})$$

Grafički prikaz oduzimanja dat je na slici A.3.



Slika A.3. Razlika dva vektora

Skalarni proizvod

Skalarni (unutrašnji, tačkasti) proizvod dva vektora v i w , u oznaci $v \cdot w$, je skalar koji se dobija po sledećoj formuli:

$$v \cdot w = |v| * |w| * \cos(\alpha)$$

gde su $|v|$ i $|w|$ dužine, odnosno intenziteti, vektora v i w , respektivno, a α ugao koji oni zaklapaju. Ukoliko su poznate koordinate vektora, skalarni proizvod se računa po sledećoj formuli:

$$\text{dot}(v, w) = v.x * w.x + v.y * w.y + v.z * w.z$$

Svaka koordinata jednog vektora množi se odgovarajućom komponentom drugog vektora, a zatim se proizvodi saberi.

Skalarni proizvod se najčešće koristi za izračunavanje ugla koji zaklapaju dva vektora.

$$\alpha = \text{arc cos} (\text{dot}(v, w) / (\text{len}(v) * \text{len}(w)))$$

Takođe, služi za proveru ortogonalnosti vektora. Ukoliko je skalarni proizvod 0, to znači da su vektori uzajamno upravni, tj. ortogonalni.

Vektorski proizvod

Vektorski proizvod dva vektora v i w , u oznaci $v \times w$, je vektor u upravan na oba ova vektora. Pravac rezultujućeg vektora određuje se pravilom desne zavojnice, a intenzitet se računa po sledećoj formuli:

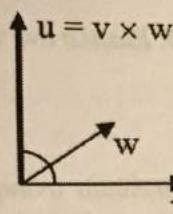
$$|u| = |v \times w| = |v| * |w| * \sin(\alpha)$$

Oznake u formuli identične su oznakama pri definiciji skalarnog proizvoda. Operacija nije komutativna, tj. $v \times w \neq w \times v$. Iz formule se vidi da se i vektorski proizvod može koristiti za izračunavanje ugla između dva vektora. Međutim, zbog veće složenosti izračunavanja u

odnosu na skalarni proizvod, daleko se ređe koristi za određivanje ugla. Koordinate rezultujućeg vektora računaju se na osnovu koordinate ulaznih vektora po sledećoj formuli:

$$\begin{aligned} u.x &= v.y * w.z - v.z * w.y \\ u.y &= v.z * w.x - v.x * w.z \\ u.z &= v.x * w.y - v.y * w.x \end{aligned}$$

Na slici A.4 dat je grafički prikaz rezultata vektorskog množenja dva vektora.



Slika A.4. Vektorski proizvod dva vektora

Implementacija klase 3D vektora

Sledi prikaz implementacije klase `vec3` za rad sa 3D vektorima. Klasa sadrži definicije operatora: dodele, sabiranja i oduzimanja vektora, kao i operator množenja vektora skalarom. Prikazane su i sve funkcije objašnjene u ovom dodatku.

```
// -----
// vec.h
// -----

class vec3
{
public:
    // Atributi
    double x;
    double y;
    double z;

    // Konstruktori
    vec3() {x = 0.0; y = 0.0; z = 0.0;}
    vec3(double xc, double yc, double zc) {x = xc; y = yc; z = zc;}

    // Operatori
    vec3& operator = (const vec3& v)
    {
        x = v.x; y = v.y; z = v.z; return *this;
    }
    vec3 operator + (const vec3& v)
    {
        return vec3(x + v.x, y + v.y, z + v.z);
    }
    vec3 operator - (const vec3& v)
    {
        return vec3(x - v.x, y - v.y, z - v.z);
    }
    vec3 operator * (const double val)
    {
        return vec3(x * val, y * val, z * val);
    }

    // Geometrijske funkcije
    double length();
    void normalize();
}
```

OPENGL - FIKSNA FUNKCIONALNOST

173

```
// Staticke geometrijske funkcije
static double distance(vec3 v, vec3 w);
static double dot(vec3 v, vec3 w);
static vec3 cross(vec3 v, vec3 w);
static double angle(vec3 v, vec3 w);

};

// -----
// vec.cpp
// -----

#include "StdAfx.h"
#include "vec3.h"
#include <math.h>

double vec3::length()
{
    return sqrt(x*x + y*y + z*z);
}

void vec3::normalize()
{
    double d = length();
    if(d!=0.0){
        x /= d;
        y /= d;
        z /= d;
    }
}

double vec3::distance(vec3 v, vec3 w)
{
    vec3 dv = v - w;
    return dv.length();
}

double vec3::dot(vec3 v, vec3 w)
{
    return (v.x*w.x + v.y*w.y + v.z*w.z);
}

vec3 vec3::cross(vec3 v, vec3 w)
{
    return vec3(    v.y * w.z - v.z * w.y,
                  v.z * w.x - v.x * w.z,
                  v.x * w.y - v.y * w.x );
}

double vec3::angle(vec3 v, vec3 w)
{
    double arg = dot(v,w) / ( v.length() * w.length() );
    if(arg > 1.0) arg = 1.0;
    if(arg < -1.0) arg = -1.0;
    return acos(arg);
}
```

Dodatak B – Korišćenje ekstenzija

Windows podrška za OpenGL je prilično stara. Direktno je podržana verzija 1.1, koja datira iz 1996. godine. Ne postoje novije biblioteke niti *header* fajlovi. Čak i *Visual Studio 2005* dolazi sa fajlovima iz sredine 90-tih, a *Visual Studio 2008* uopšte ne podržava OpenGL.

Kako onda uposlitи nove grafičke kartice i kako doći do novih efekata?

Odgovor je jednostavan - korišćenjem OpenGL ekstenzija!

Funkcionalnost koju koristi OpenGL ugrađena je u drajvere, a mehanizam ekstenzija nudi interfejs ka tim funkcijama. Da bi se dobila nova funkcionalnost potrebno je kupiti odgovarajući grafički akcelerator (karticu), instalirati drajvere i funkcionalnost je tu. Ne mora se čekati da se objavi SDK ili neka biblioteka.

Ekstenzija je zajednički naziv za:

- *name string* – jedinstveno ime koje je identificuje,
- funkcije – ne implementira svaka ekstenzija funkcije, ali većina da,
- enumeracije – ekstenzije mogu definisati i nove konstante i
- zavisnost – vrlo retko su ekstenzije samostalne, već zahtevaju prisustvo drugih ekstenzija, ili čak modifikuju ili proširuju druge.

Svaka ekstenzija mora imati jedinstveno ime. Ime se sastoji od prefiksa, koji definiše „proizvođača“ ekstenzije, i naziva funkcionalnosti ekstenzije. Na primer EXT_gpu_shader4. Prefiksi mogu biti: 3DFX, 3DL, 3Dlabs, AMD, APPLE, ARB, ATI, Autodesk, DIMD, EXT, EXTX, FGL, GL2, GREMEDY, HP, I3D, IBM, IMG, INGR, INTEL, KTX, MESA, MTX, NV, NVX, OES, OML, S3, SGI, SGIS, SGIX, SUN, SUNX, WGL i WIN.

Najznačajnije su ekstenzije koji počinju prefiksom ARB. To su ekstenzije koje su već ušle u zvaničnu specifikaciju OpenGL-a ili su kandidati za ulazak. Druge po značaju počinju prefiksom EXT. To su ekstenzije oko kojih se složilo više proizvođača. Generalni problem korišćenja ekstenzija je pitanje da li je proizvođač određenog grafičkog adaptora podržao datu ekstenziju. ARB ekstenzije se po pravilu implementiraju, ukoliko hardver može da ih podrži. Takođe, postoji velika verovatnoća i da su EXT ekstenzije podržane. Ali to nikako ne važi za ekstenzije vezane za samo jednog proizvođača. Ekstenzije vezane samo za jednog proizvođača grafičkog hardvera počinju prefiksom koji predstavlja datu firmu, i trenutno dve najznačajnije grupe ekstenzije počinju prefiksom NV (NVidia) i ATI (ATI/AMD).

Stanje na tržištu grafičkih akceleratora (kartica) je vrlo dinamično. Kako u takvom okruženju programer da zna koje su funkcije na raspolaganju? Odgovor je jednostavan. Na adresi <http://www.opengl.org/registry/> nalazi se spisak svih ekstenzija, njihov opis i potrebnih fajlovi.

Ponutan uključivanja podrške za ekstenzije prikazaćemo u 5 koraka, na primeru poziva funkcije `glGetUniformuivEXT` u okviru ekstenzije čiji je *name string* `GL_EXT_`

gpu_shader4. Name string dobija se kada se ispred naziva doda prefiks GL_ (sve konstante u OpenGL-u počinju ovako).

Korak 1. Posetiti adresu <http://www.opengl.org/registry/> i preuzeti odgovarajuće datoteke:

- gpu_shader4.txt (opis ove ekstenzije – ekstenzija br. 326)
- header datoteke – glext.h i wglext.h.

Korak 2. Uključiti odgovarajuća zaglavla u stdafx.h datoteku (ili u .cpp datotekama u kojima se koriste OpenGL funkcije).

```
// stdafx.h
// ...
#include "gl.h"
#include "glu.h"

#include "glext.h"
#include "wglext.h"

#pragma comment(lib, "opengl32.lib")
```

Korak 3. Deklarisati pokazivač na funkciju. U datoteci glext.h potražimo naziv odgovarajuće funkcije. U našem primeru to je glGetUniformuivEXT. Obzirom da je potrebno od drajvera pribaviti pokazivač na funkciju, ime glGetUniformuivEXT iskoristićemo za ime promenljive preko koje pristupamo toj adresi. Praksa je da se za pokazivač na funkciju koristi sledeći šablon: PFN<NAZIV_FUNKCIJE_ISPISAN_VELIKIM_SLOVIMA>PROC. Sledеći ovaj šablon, za funkciju glGetUniformuivEXT pokazivač je tipa:

PFNGLGETUNIFORMUIVEXTPROC

Proverimo da li odgovarajući tip postoji u datoteci glext.h. Ukoliko postoji, deklarišimo pokazivač na datu funkciju, u okviru našeg programa, na sledeći način:

PFNGLGETUNIFORMUIVEXTPROC glGetUniformuivEXT = NULL;

Ova deklaracija treba da bude „vidljiva“ iz svih funkcija u kojima se poziva. Može se dodati kao atribut neke klase, a može biti i globalna vrednost.

Korak 4. Proveriti da li je ekstenzija podržana i, ukoliko je podržana, pribaviti pokazivač na funkciju. Spisak svih podržanih ekstenzija u obliku niza karaktera (pri čemu su nazivi odvojeni jednim blankom znakom) dobija se pozivom funkcije glGetString(), sa parametrom GL_EXTENSIONS. Obzirom da se radi o OpenGL funkciji, neophodno je da u trenutku poziva postoji OpenGL rendering context i da je aktivran. Pokazivač na funkciju pribavlja se funkcijom wglGetProcAddress(), kojoj se prosleđuje naziv željene funkcije.

```
char* str = (char*)glGetString(GL_EXTENSIONS);

if(strstr(str, "GL_EXT_gpu_shader4 "))
{
    glGetUniformuivEXT = (PFNGLGETUNIFORMUIVEXTPROC)
        wglGetProcAddress("glGetUniformuivEXT");
}
```

Korak 5. Pozvati funkciju na izvršenje. Ukoliko ekstenzija nije podržana, a to se može odrediti samo u toku izvršenja na odgovarajućem hardveru, funkcija `wglGetProcAddress()` vraća NULL. Može se desiti, mada vrlo retko, da postoji „potpis“ ekstenzije u stringu koji vraća `glGetString()`, ali da ipak ne uspe pribavljanje pokazivača na funkciju. Ovo je karakteristično pre svega za „beta“ drajvere, ali je uvek poželjno proveriti da li je odgovarajući pokazivač NULL.

Ako je pokazivač validan, funkcija se poziva na isti način kao i bilo koja druga funkcija u C/C++u. U našem primeru poziv funkcije ima sledeći oblik:

```
glGetUniformivEXT(program, loc);
```

Osim funkcija koje čine jezgro OpenGL-a, ekstenzijama se mogu definisati i funkcije koje povezuju operativni sistem i OpenGL. U slučaju Windowsa to su wgl funkcije. Da bi se preuzeo spisak wgl ekstenzija potrebno je pozvati funkciju `wglGetExtensionsStringARB`, koja je i sama ekstenzija. Sledi primer preuzimanja spiska wgl ekstenzija.

```
PFNWGLGETEXTENSIONSSTRINGARBPROC wglGetExtensionsStringARB = NULL;
wglGetExtensionsStringARB = (PFNWGLGETEXTENSIONSSTRINGARBPROC)
    wglGetProcAddress("wglGetExtensionsStringARB");
char* wglStr = NULL;
if(wglGetExtensionsStringARB)
    wglStr = (char*)wglGetExtensionsStringARB(pDC->m_hDC);
```

Ako pogledamo sadržaj vraćenog stringa, videćemo da se i sama ekstenzija koja služi za pristup (WGL_ARB_extensions_string) nalazi u njemu.

```
WGL_ARB_buffer_region WGL_ARB_extensions_string
WGL_ARB_make_current_read WGL_ARB_multisample WGL_ARB_pbuffer
WGL_ARB_pixel_format WGL_ARB_pixel_format_float
WGL_ARB_render_texture WGL_ATI_pixel_format_float
WGL_EXT_extensions_string WGL_EXT_framebuffer_sRGB
WGL_EXT_pixel_format_packed_float WGL_EXT_swap_control
WGL_NV_float_buffer WGL_NV_multisample_coverage
WGL_NV_render_depth_texture WGL_NV_render_texture_rectangle
```

Ovo je ciklična zavisnost, pa je nemoguće ispitati da li je ova ekstenzija podržana (jer se ne nalazi u stringu koji vraća funkcija `glGetString`). Zbog toga se u prethodnom primeru bez ispitivanja da li je podržana ekstenzija zahteva pokazivač, a zatim ispituje da li je NULL.

Dodatak C – Pomoćne biblioteke

U prethodnom dodatku videli smo kako se direktno pristupa ekstenzijama. Ovaj postupak može biti mnogo lakši ukoliko se koristi neka od pomoćnih biblioteka-omotača. Jedini nedostatak korišćenja biblioteka je neophodnost čekanja na izdavanje nove verzije biblioteke. Nekada to znači čekanje i po godinu dana. Ekstenzije se na tržištu pojavljuju daleko brže. Međutim, ukoliko nije neophodno u softverski proizvod uključiti najnoviju funkcionalnost, ove biblioteke predstavljaju veoma korisno sredstvo.

OpenGL Extension Wrangler Library (GLEW)

OpenGL Extension Wrangler Library (GLEW) je *cross-platform open-source C/C++* biblioteka za pristup ekstenzijama. Implementira vrlo efikasan mehanizam za utvrđivanje koje su ekstenzije podržane i za pristup podržanoj funkcionalnosti. Ovo je najpopularnija biblioteka-omotač ekstenzija, o čemu govori i činjenica da je NVidia uključila ovu biblioteku u svoj OpenGL SDK.

Biblioteka se može preuzeti sa adrese <http://glew.sourceforge.net/>. U vreme pisanja ovog priručnika bila je aktuelna verzija 1.5.1, objavljena 3. novembra 2008. godine, sa podrškom za OpenGL 3.0.

Postupak uključivanja GLEW biblioteke odvija se u tri koraka.

Korak 1. Instalacija biblioteke ostvaruje se kopiranjem odgovarajućih datoteka u direktorijume gde ih *Visual Studio* i aplikacije mogu pronaći. Biblioteku čine:

- **glew32.dll** – run-time biblioteka, koju treba iskopirati u sistemski direktorijum (Windows/System32) ili u lokalni direktorijum aplikacije,
- **glew32.lib** – statička biblioteka sa implementacijom svih funkcija, koju treba iskopirati u *Visual Studio Lib* direktorijum, ili lokalni direktorijum projekta koji se razvija,
- **glew.h** i **wglew.h** – zaglavla, koja treba iskopirati u *Visual Studio Include/GL* direktorijum ili lokalni direktorijum projekta.

Korak 2. Povezati projekat sa GLEW bibliotekom navođenjem sledećeg koda u stdafx.h datoteku:

```
// stdafx.h  
  
#include "glew.h"  
#include "wglew.h"  
  
#pragma comment(lib, "glew32.lib")
```

Datoteka **glew.h** već sadrži **gl.h** i **glu.h** zaglavla, pa ih ne treba navoditi.

Korak 3. Inicijalizovati GLEW pozivom funkcije `glewInit()` u okviru `PrepareScene()` funkcije.

```
void CGLRenderer::PrepareScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    GLenum err = glewInit();
    if (err != GLEW_OK)
    {
        // Nastala je greska pri inicijalizaciji
        // ...
    }
    // ...
    //-----
    wglMakeCurrent(NULL, NULL);
}
```

Ukoliko je inicijalizacija prošla uspešno, funkcije sadržane u svim podržanim ekstenzijama mogu se direktno zvati. Ako se koristi OpenGL 3.0, inicijalizaciju GLEW-a treba prenesti u `CreateGLContext()`, jer je i OpenGL *rendering context* počev od verzije 3.0 ekstenzija. Naravno, da bi se pozivale OpenGL funkcije, potrebno je da postoji OpenGL *rendering context* i on mora biti aktivran. Zato se najpre formira „stari“ *rendering context*, aktivira se, inicijalizuje GLEW, napravi „novi“ *rendering context*, a zatim uništi stari. Detaljnije o formiranju OpenGL 3.0 *rendering context*-a biće reči u sledećem dodatku.

OpenGL Easy Extension library (GLEe)

OpenGL Easy Extension Library (GLEe) je besplatna *cross-platform* C/C++ biblioteka za pristup ekstenzijama. Može se preuzeti sa adrese <http://elf-stone.com/glee.php>. U vreme pisanja ovog priručnika bila je aktuelna verzija 5.4, objavljena 7. februara 2009. godine. GLee je dodao podršku za OpenGL 3.0 još u verziji 5.33, mesec dana pre GLEW-a (6. oktobra 2008. godine).

Postupak uključivanja GLee biblioteke odvija se u dva koraka.

Korak 1. Instalacija biblioteke ostvaruje se kopiranjem odgovarajućih datoteka u direktorijume, gde ih *Visual Studio* i aplikacije mogu pronaći. Biblioteku čine:

- **glee.lib** – statička biblioteka sa implementacijom svih funkcija, koju treba iskopirati u *Visual Studio Lib* direktorijum, ili lokalni direktorijum projekta koji se razvija,
- **glee.h** – zaglavlje, koja treba iskopirati u *Visual Studio Include/GL* direktorijum ili lokalni direktorijum projekta, i
- **glee.c** – implementacija biblioteke, koristi se kao zamena za **glee.lib** datoteku i uključuje se u projekat ukoliko biblioteka ne može da se linkuje (poveže) sa tekućim projektom.

Korak 2. Povezati projekat sa GLee bibliotekom navođenjem sledećeg koda u stdafx.h datoteci:

```
// stdafx.h
#include "glee.h"
#pragma comment(lib, "glee.lib")
```

Nema inicijalizacionog koda, i ekstenzije se učitavaju tek kada se pozovu na izvršenje.
Učitavanje određene ekstenzije može se forsirati pozivom funkcije:

```
GLint GLEForceLink(const char * extensionName)
```

a kompletna inicijalizacija može se ostvariti pozivom funkcije:

```
GLboolean GLEInit( void )
```

Postoji problem prilikom linkovanja GLee biblioteke u okviru *Visual Studio 2008* projekata. Datoteka **glee.lib** prevedena je sa starijim prevodiocem, što izaziva konflikt. Da bi GLee „proradio“ potrebno je uraditi sledeće:

- ukloniti sledeću liniju iz **stdafx.h** datoteke (dodata u koraku 2)
 - o `#pragma comment(lib, "glee.lib")`
- promeniti ekstenziju datoteke **glee.c** iz **.c** u **.cpp**
- na početku **glee.cpp** datoteke dodati sledeću liniju koda:
 - o `#include "StdAfx.h"`
- uključiti datoteku **glee.cpp** u projekat

Da bi proces inicijalizacije bio sličan GLEW biblioteci, u **PrepareScene()** možemo dodati inicijalizaciju GLee biblioteke.

```
void CGLRenderer::PrepareScene(CDC *pDC)
{
    wglMakeCurrent(pDC->m_hDC, m_hrc);
    //-----
    GLboolean err = GLEInit();
    if (err != GL_TRUE)
    {
        // Nastala je greska pri inicijalizaciji
        // ...
    }
    // ...
    //-----
    wglMakeCurrent(NULL, NULL);
}
```

Dodatak D – Kreiranje OpenGL 3.x Rendering Context-a

Samo tri dana nakon publikovanja specifikacije OpenGL-a 3.0, NVidia 14. avgusta 2008. godine objavljuje prve beta drajvere za Windows XP i Windows Vista operativne sisteme (ForceWare 178.25), namenjene prvenstveno programerima, koji podržavaju tek izašlu specifikaciju. Funkcionalnost nije bila direktno dostupna, već ju je bilo potrebno „otključati“ posredovanjem programa *NVemulate*.

Svoju dominaciju u oblasti računarske grafike i procesu standardizacije, NVidia je po drugi put dokazala 17. decembra 2008. godine, kada su objavljeni drajveri (ForceWare 181.00) koji u potpunosti podržavaju OpenGL 3.0.

Više od mesec dana nakon toga, 28. januara 2009. godine i ATI sa svojim *Catalyst* 9.1 drajverima uključuje podršku za OpenGL 3.0. Mnogo pre nego što se očekivalo, nova funkcionalnost je postala dostupna svima.

Da bi aktivirali OpenGL 3.0, najpre je potrebno kreirati odgovarajući *rendering context*. Sa zvaničnog sajta (<http://www.opengl.org/registry/>) treba preuzeti najnovije verzije fajlova: *glext.h* i *wglext.h*, i dodati u *stdafx.h* sledeće dve linije koda, kako bi uključili podršku za ekstenzije (vidi Dodatak B – Korišćenje ekstenzija):

```
#include "glext.h"
#include "wglext.h"
```

a zatim promeniti implementaciju funkcije **CreateGLContext()**, opisane u prvoj glavi – Uvod u OpenGL, tako da dobije sledeći oblik:

```
bool CGLRenderer::CreateGLContext(CDC* pDC)
{
    PIXELFORMATDESCRIPTOR pfd;
    memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));
    pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
    pfd.nVersion = 1;
    pfd.dwFlags = PFD_DOUBLEBUFFER | PFD_SUPPORT_OPENGL
                  | PFD_DRAW_TO_WINDOW;
    pfd.iPixelType = PFD_TYPE_RGBA;
    pfd.cColorBits = 32;
    pfd.cDepthBits = 32;
    pfd.iLayerType = PFD_MAIN_PLANE;
    int nPixelFormat = ChoosePixelFormat(pDC->m_hDC, &pfd);
    if (nPixelFormat == 0) return false;
    BOOL bResult = SetPixelFormat(pDC->m_hDC, nPixelFormat,
                                 &pfd);
    if (!bResult) return false;
    HGLRC tempContext = wglCreateContext(pDC->m_hDC);
    wglMakeCurrent(pDC->m_hDC, tempContext);
```

DODATAK D

```

int attribs[] = {
    WGL_CONTEXT_MAJOR_VERSION_ARB, 3,
    WGL_CONTEXT_MINOR_VERSION_ARB, 0,
    WGL_CONTEXT_FLAGS_ARB, 0,
};

PFNWGLCREATECONTEXTATTRIBSARBPROC wglCreateContextAttribsARB = NULL;
wglGetProcAddress("wglCreateContextAttribsARB");
if(wglCreateContextAttribsARB != NULL)
{
    m_hrc = wglCreateContextAttribsARB(pDC->m_hDC, 0, attribs);
    wglGetCurrent(NULL, NULL);
    wglDeleteContext(tempContext);
    if (!m_hrc) return false;
}
return true;
}

```

Nova verzija OpenGL-a uvodi nove funkcionalnosti i raskida kompatibilnost sa prethodnim verzijama što zahteva i novi način kreiranja *rendering context-a*. Počev od verzije 3.0, umesto funkcije **wglCreateContext()** koristi se funkcija **wglCreateContextAttribsARB()**.

```
HGLRC wglCreateContextAttribsARB(HDC hDC, HGLRC hShareContext,
                                const int *attribList)
```

Prvi parametar funkcije je *handle DC-a* za koji se kreira odgovarajući OpenGL *rendering context*, drugi parametar je *handle rendering context-a* sa kojim tekući može da deli podatke, a treći je lista atributa. Lista atributa definisana je parovima (ime_atributa, vrednost). Redosled navođenja atributa nije bitan, a najznačajniji su:

- **WGL_CONTEXT_MAJOR_VERSION_ARB** – glavna verzija,
- **WGL_CONTEXT_MINOR_VERSION_ARB** – podverzija,
- **WGL_CONTEXT_FLAGS_ARB** – ostali parametri konteksta.

Da bi se dobio OpenGL *rendering context* 3.0 potrebno je **WGL_CONTEXT_MAJOR_VERSION_ARB** postaviti na 3, a **WGL_CONTEXT_MINOR_VERSION_ARB** na 0. Ukoliko se postavi manja vrednost ili se uopšte ne navede verzija i podverzija, formira se OpenGL *context* 2.1 Kod najnovijih drajvera sa podrškom za OpenGL 3.2, nenađenjem verzije *rendering context-a* se podrazumeva verzija 1.0, što znači „najnovija verzija koja istovremeno podržava i verziju 1.0“, ili u prevodu OpenGL 3.2 *compatibility* profil (profil u kome su podržane i takozvane „zastarele“ funkcije). Počev od *ForceWare* 182.52 drajvera moguće je postaviti verziju 3.1, a sa 190.57 drajverima i verziju 3.2. Uporedo sa objavljivanjem specifikacije OpenGL-a 3.2, NVIDIA objavljuje i prve (beta) drajvere sa podrškom za novu verziju OpenGL-a – 190.56. Međutim, tek verzija 190.57 (objavljena samo tri nedelje nakon prethodne) otklanja sve bitne greške i zaista postaje funkcionalna.

Dodatne opcije, koje se mogu uključiti u okviru OpenGL 3.x *rendering context-a*, definisane su flegovima WGL_CONTEXT_FLAGS_ARB atributa:

- WGL_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB – isključuje kompatibilnost unazad (dostupne su samo funkcije koje nisu označene kao zastarele),
- WGL_CONTEXT_DEBUG_BIT_ARB – uključuje prikupljanje dodatnih podataka prilikom izvršenja i pomaže u otklanjanju grešaka.

Podrška za DEBUG *rendering context* još uvek ne postoji (ni u verziji 3.2). Čak ne postoji ni preliminarna specifikacija šta bi takav *rendering context* trebalo da radi, tako da WGL_CONTEXT_DEBUG_BIT_ARB trenutno nema nikakvu funkciju.

Model podrške zastarelim funkcijama dodatno je proširen u okviru OpenGL-a 3.2 uvođenjem profila. Trenutno postoje dva profila:

- core – sadrži samo „moderne“ funkcije i
- compatibility – sadrži sve funkcije iz prethodnog profila, ali i sve funkcije koje su dodavane u specifikaciju OpenGL-a od njegovog postanka do danas (potpuna kompatibilnost unazad).

Profil se postavlja atributom WGL_CONTEXT_PROFILE_MASK_ARB (dostupan tek u verziji 3.2), definisanim kao polje flegova (zbog budućih proširenja i mogućnosti da istovremeno budu aktivna više profila). Fleg WGL_CONTEXT_CORE_PROFILE_BIT_ARB aktivira *core* profil, a WGL_CONTEXT_COMPATIBILITY_PROFILE_BIT_ARB aktivira *compatibility* profil. Zbog različite prirode ova dva profila, ne mogu istovremeno oba da budu aktivna, tj. međusobno se isključuju. Podrazumeva se *core* profil, a postavljanje WGL_CONTEXT_COMPATIBILITY_PROFILE_BIT_ARB ga isključuje.

Polje atributa završava se nulom.

Obzirom da je funkcija **wglCreateContextAttribsARB()** takođe OpenGL ekstenzija, u trenutku njenog pozivanja potrebno je da postoji OpenGL *rendering context* i mora biti aktivan. Dakle, pre formiranja *rendering context-a* moramo imati *rendering context*. Ovaj paradox, odnosno cirkularnu zavisnost, možemo prevazići kreiranjem „starog“ *rendering context-a*, korišćenjem funkcije **wglCreateContext()**. Koraci formiranja „novog“ *rendering context-a* su sledeći:

- kreirati „stari“ *rendering context*,
 - o HGLRC tempContext = wglCreateContext(pDC->m_hDC);
- aktivirati ga,
 - o wglGetCurrent(pDC->m_hDC, tempContext);
- postaviti atribute i kreirati „novi“ *rendering context* (),
 - o int attribs[] = {WGL_CONTEXT_MAJOR_VERSION_ARB, 3,...};
 - o m_hrc = wglCreateContextAttribsARB(pDC->m_hDC, 0, attribs);
- deaktivirati i obrisati „stari“
 - o wglGetCurrent(NULL, NULL);

○ wglDeleteContext(tempContext);

Podršku za OpenGL 3.x trenutno nude samo NVidia (puna podrška za 3.1 i od grešaka prilično očišćena beta verzija drajvera za 3.2) i AMD/ATI (puna podrška za 3.0).