Open Graphics Library (OpenGL) C#

OpenGL je standardni programski interfejs aplikacije (*API – Application Programming Interface*) namenjen razvoju aplikacija u oblasti dvodimenzionalne i trodimenzionalne računarske grafike.

OpenGL standard razvijen je i utvrđen 1992. godine od strane vodećih kompanija, koje čine *OpenGL Architecture Review Board*, kao efektivni hardversko-nezavisni interfejs, pogodan za realizaciju na različitim platformama. Za osnovu standarda iskorišćena je programska biblioteka *IRIS GL*, koju je razvila kompanija *Silicon Graphics Inc (SGI)*.

Većina proizvođača hardvera i softvera podržava OpenGL. Implementacije OpenGL standarda postoje za veliki broj platformi. Velika rasprostranjenost OpenGL standarda se bazira na sledećim osobinama:

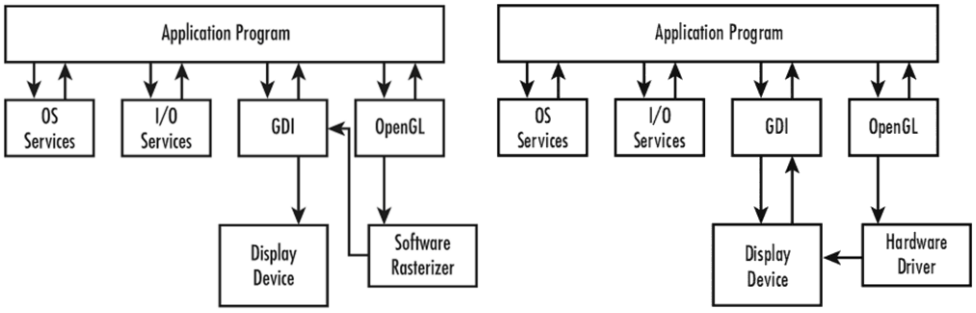
- **Stabilnost**. Dopune i izmene standarda održavaju kompatibilnost sa prethodnim verzijama.
- **Prenosivost.** OpenGL aplikacije garantuju istovetan vizuelni rezultat nezavisno od tipa korišćenog operativnog sistema i hardvera.
- **Lakoća upotrebe**. OpenGL poseduje intuitivan i jednostavan interfejs koji omogućava razvoj efektivnih aplikacija uz manji programerski napor. Obično aplikacije koje se oslanjanju na OpenGL sadrže manji broj redova programskog kôda nego kada se koriste druge grafičke biblioteke. Specifičnosti korišćenog hardvera kao i obezbeđivanje kompatibilnosti sa različitom opremom realizovane su na nivou OpenGL implementacije.

Verzija standarda koja će biti obrađena je OpenGL 2.1 verzija standarda.

Više informacija o OpenGL standardu se može pronaći na Internet adresi: http://www.opengl.org/.

1 Arhitektura OpenGL programske biblioteke

OpenGL se sastoji od: specifikacije i implementacije. Specifikacija propisuje funkcionalnost, a implementacija realizuje specifikovanu funkcionalnost. Implementacija OpenGL standarda može biti softverska ili hardverska, slika 1.1. OpenGL dozvoljava softversku emulaciju (softversku implementaciju) funkcionalnosti koju hardver ne implementira. Softverska implementacija koristi raspoloživi grafički sistem (npr. *GDI* na Windows plaftormi) za iscrtavanje, dok hardverska koristi OpenGL kompatibilni hardver za iscrtavanje. Obično grafički hardver daje različite nivoe ubrzavanja: od hardverske realizacije generisanja linija i poligona do moćnih grafičkih procesora sa podrškom za različite operacije nad geometrijskim podacima.



a) softverska b) hardverska

Slika 1.1 Tipovi implementacija OpenGL biblioteke za *Windows* platformu, preuzeto iz [1]

OpenGL metode realizovane su po modelu **klijent-server**. Aplikacija (klijent) poziva OpenGL metode, a OpenGL (server) ih interpretira i izvršava. Server se može nalaziti na tom istom računaru, na kom se nalazi i klijent ili na drugom računaru.

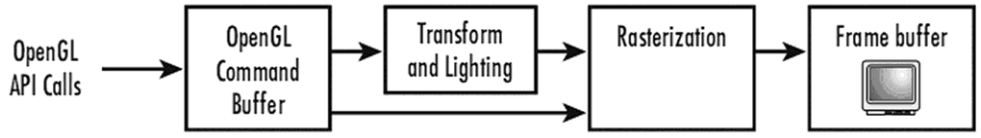
OpenGL iscrtava grafičke objekte u baferu kadra (*frame buffer*) uzimajući u obzir izabrane režime iscrtavanja. Svaki režim može se menjati nezavisno od drugih. Definisanje primitiva, izbor režima i druge operacije opisuju se pomoću **komandi** u obliku poziva OpenGL metoda.

Grafički objekti se definišu skupom temena (*vertex*). Temenu se pridružuju podaci (npr. koordinate, boja i dr.) koji se nazivaju **atributima**.

Uprošćeni proces prezentacije grafike, slika 1.2, se sastoji od sledećih koraka:

- poziva metoda *OpenGL API* od strane aplikacije (klijenta),
- skladištenja zadatih metoda – komandi u *Command* baferu,
- pražnjenja *Command* bafera (ili programski ili automatski),
- izvršavanja komandi, u redosledu njihovog pojavljivanja,
- primene transformacije i osvetljenja,
- rasterizacije – pretvaranja 3D scene u 2D projekciju,
- upis rasterizovane scene u bafer kadra grafičke kartice (iscrtavanje).

Pravi proces uključuje još neke dodatne korake, ali oni nisu neophodni za razumevanje načina rada OpenGL programske biblioteke. Čitaocima koje to zanima se preporučuje da pogledaju u literaturi [1].



Slika 1.2 Uprošćeni OpenGL proces prezentacije grafike, preuzeto iz [1]

1.1 Organizacija OpenGL C# implementacije

OpenGL omogućava prezentaciju računarske grafike nezavisno od konkretnog hardvera, i kao takav se može posmatrati kao softverski interfejs prema grafičkom hardveru. Pošto OpenGL vrši prezentaciju grafike, on ne sadrži u sebi nikakve specijalne komande za rad sa prozorima ili prihvat ulaznih podataka od korisnika. Oslanjanje na neku konkretnu platformu postaje problem ako se želi pisati prenosiv (*portable*) programski kôd, jer tada treba nekako apstrahovati platformske specifičnosti rukovanja prozorima, ulazno-izlaznim uređajima i dr. S toga su napravljene prenosive biblioteke koje obezbeđuju često korišćene metode za međusobno delovanje računara i korisnika, kao i za prikaz informacija preko podsistema prozora.

OpenGL implementacija koju ćemo koristiti je organizovana kao jedna programska biblioteka **SharpGL**. Sve bazne metode su implementirane u klasi **OpenGL** kao njene metode. Klasa pripada *Sharpgl* imenskom prostoru (*namespace*).

Funkcionalnost osnovne biblioteke na Windows platformi je implementirana u *SharpGL.dll* dinamičkoj programskoj biblioteci. Da bi se mogle koristiti OpenGL metode potrebno je u zaglavlju klase sa kojom radimo dodati sledeći programski kôd:

```
using SharpGL;
```

Veoma je bitno naglasiti da ceo SharpGL predstavlja samo C# *wrapper* odgovarajućih biblioteka implementiranih korišćenjem nativnog programskog jezika (konkretno za OpenGL u pitanju je C programski jezik).

Dave Kerr je njen originalni kreator, a trenutno je objavljena pod MIT licencom i njen source code je dostupan preko Github-a. Pored funkcionalnosti samog OpenGL-a, dostupne su nam i kontrole preko kojih možemo iscrtavati unutar WFA i WPF aplikacija.

U nastavku ove skripte **definicije** OpenGL funkcija će biti predstavljene onako kako bi se pozivale u programskom jeziku C, a ovo su osnovna pravila koja se koriste prilikom prebacivanja tih funkcija u C# jezik, koristeći SharpGL biblioteku i u tom jeziku će biti prikazani delovi izvornog koda.

- pozivi metoda koji počinju sa **gl ili glu** su member metode SharpGL.OpenGL objekta, pa se, ukoliko se referenci na taj objekat dodelilo ime **gl**, one mogu pozivati na sledeći način

  ```
  glLineWidth(3.0f);  ----->  gl.LineWidth(3.0f);
  ```

- nazivi funkcija koji unutar svog imena u C kodu za postfix imaju definisan tip parametra su unutar SharpGL biblioteke definisane bez tog postfixa, budući da unutar C#-a mogu postojati isti nazivi funkcija koje primaju različite parametre. Tako bi poziv menjanja boje izgledao

```
glColor3f(1f,1f,1f); ----> gl.Color3(1f,1f,1f);
```

- OpenGL konstante su definisane kao konstanti atributi SharpGL.OpenGL klase, i zadržale su identična imena

```
glPolygonMode(GL_FRONT, GL_FILL);  --->

     gl.PolygonMode(OpenGL.GL_FRONT,
OpenGL.GL_FILL);
                ili
     gl.PolygonMode(PolygonFace.Front,
PolygonMode.Fill);
```

1.2 Sintaksa OpenGL metoda

Nazivi OpenGL metoda su deskriptivni i obično pružaju informacije o klasi u kojoj su implementirane, broju i tipu parametara. Nomenklatura OpenGL metoda je (sa * su označeni opcioni elementi):

```
tip <prefiks><naziv><brojP*><tipP*><v>(tip1
param1,…,tipN paramN)
```

gde su:

`tip` – OpenGL tip podataka koji vraća metoda,

`<prefiks>` – karakter koji se želi prikazati,

`<naziv>` – naziv metode,

`<brojP>` – broj parametara dozvoljene vrednosti su: 1, 2, 3, 4. Ovaj element je opcion i pojavljuje se samo ako su tipovi parametara isti,

`<tipP>` – OpenGL tip podataka. Dozvoljene su sledeće vrednosti: *b, s, i, f, d, ub, us, ui*. Vrednosti predstavljaju skraćene oznake OpenGL tipova. Na primer, *b* odgovara *byte* tipu podataka, *i* odgovara *int* tipu podataka itd. Ovaj element je opcion i pojavljuje se samo ako su tipovi parametara isti,

`<v>` – oznaka da se kao parametar metode koristi pokazivač na niz vrednosti. Ovaj element je opcion i pojavljuje se samo ako su tipovi parametara isti,

`tip1, ..., tipN` – OpenGL tipovi podataka parametara 1 .. N,

`param1, ..., paramN` – nazivi parametara 1 .. N.

Primer metode sa oznakama prikazan je na slici 1.2.1. Ovakva nomenklatura u značajnoj meri olakšava pamćenje naziva metoda, kao i broja i tipa parametara.

Slika 1.2.1 Primer OpenGL funkcije

1.3 OpenGL tipovi podataka

Za razliku od C verzije, C# ne poseduje upotpunosti ekvivalent *#typedef* direktivi (koja se koristi u C verziji) pa zato se koriste C# tipovi podataka. Većina tipova je ekvivalentna njihovim C verzijama. C pokazivači se opisuju **IntPtr** tipom podataka. Razlika postoji i kod nizova koji su poseban tip u C#.

Sve konstante definisane u klasi *Gl* počinju prefiksom „*GL_*“. Odgovarajuće konstante klasa *Glu* analogno imaju prefikse „*GLU_*“.

1.4 OpenGL promenljive stanja

OpenGL specifikacija definiše OpenGL kao **automat stanja**. Definisan je veliki broj promenljivih stanja koje se mogu uključiti/isključiti pozivom metode **glEnable**/**glDisable**. Ako je promenljiva stanja uključena, ponovni pozivi uključenja iste promenljive se ignorišu. Analogno važi i za isključivanje promenljive. OpenGL razlikuje dva tipa promenjlivih stanja: klijentske (*client*) i serverske (*server*). Klijentske su vezane za stanja na strani klijenta, dok su serverske vezane za stanja na strani servera. Ako nije naglašeno, sve promenljive stanja opisane u daljem tekstu su serverske promenljive stanja.

```
public static void glEnable(int cap)
public static void glDisable(int cap)
```

gde je:

`cap` – simbolička konstanta promenljive stanja koja se uključuje/isključuje. Kompletan spisak promenljivih stanja i odgovarajućih konstanti može se videti u dokumentaciji.

Metoda **glIsEnabled** se koristi za ispitivanje da li određena promenljiva stanja uključena ili ne.

```
public static int glIsEnabled(int cap)
```

gde je:

`cap` – konstanta promenljive stanja za koju se ispituje stanje.

Često postoji potreba da se stanje sistema sačuva/restaurira. OpenGL specifikacija definiše *FIFO* stek stanja sistema, koji omogućava realizaciju gore navedenih metoda. Stekom se manipuliše korišćenjem metoda **glPushAttrib** i **glPopAttrib**.

```
void glPushAttrib(int mask)
void glPopAttrib()
```

gde je:

`mask` – *bitwise OR* kombinacija konstanti promenljivih stanja. Npr *glPushAttrib(GL_TEXTURE_BIT | GL_LIGHTING_BIT)* snima stanje rada sa teksturama i osvetljenjem.

1.5 OpenGL greške

Tokom izvršavanja OpenGL metoda mogu nastati greške. OpenGL registruje samo mali podskup mogućih grešaka. Ovakav pristup se bazira na činjenici da bi registrovanje i provera svih mogućih grešaka opasno ugrozila performansu. OpenGL signalizira greške preko numeričkih vrednosti koje se u OpenGL terminologiji nazivaju zastavicama (*flags*). OpenGL zastavice su prikazane u tabeli 1.5.1. Ako dođe do grešaka sistem nastavlja sa radom, a metode koje su izazvale grešku se ignorišu. Jedini izuzetak je zastavica *GL_OUT_OF_MEMORY* kada je rezultat metode nedefinisan.

Naziv	Opis
GL_INVALID_ENUM	*Enum* argument je van opsega.
GL_INVALID_VALUE	Numerički argument je van opsega.
GL_INVALID_OPERATION	Trenutno stanje sistema ne dozvoljava izvršenje metode.
GL_STACK_OVERFLOW	Izvršenje metode izaziva *stack overflow*.
GL_STACK_UNDERFLOW	Izvršenje metode izaziva *stack underflow*.
GL_OUT_OF_MEMORY	Za izvršenje OpenGL metode sistema nema dovoljno slobodno memorije.
GL_TABLE_TOO_LARGE	Zadata tabela je suviše velika.
GL_NO_ERROR	Nema greške – uspešno izvršena OpenGL metoda.

Tabela 1.5.1 Zastavice grešaka (*error flags*)

Za utvrđivanje koja zastavica je postavljena koristi se metoda **glGetError** koja vraća vrednost iz tabele 1.5.1. Za dobijanje deskriptivnog tekstualnog opisa greške koristi se metoda **gluErrorString**.

```
public static int glGetError()
public static string gluErrorString(int errorCode)
```

gde je:

`errorCode` – konstanta greške, videti tabelu 1.5.1

Ako se desilo više grešaka tada se one mogu dobiti sukcesivnim pozivima metode *glGetError* sve dok povratna vrednost metode ne bude vrednost: *GL_NO_ERROR*. S toga je uobičajno da se poziv ove metode nalazi unutar petlje.

2. OpenGL iscrtavanje objekata

OpenGL omogućava iscrtavanje sledećih grafičkih objekata u prostoru: tačaka, linija, trouglova, poligona, punih tela (sfera, kvadra i dr.) i *mesh* objekata. Ovo poglavlje se sastoji iz pet sekcija: sekcija o tačkama i linijama, sekcija o poligonima, sekcije o mesh objektima, sekcije o punim telima kao i sekcije koja je posvećena naprednijim mehanizmima iscrtavanja.

2.1 Crtanje tačaka i linija

Osnovna jedinica iscrtavanja u OpenGL standardu je tačka (*vertex*). Tačka je određena sa tri koordinate Dekartovog pravouglog koordinatnog sistema (pravilo desne ruke).

Tačka se definiše u OpenGL standardu pozivom neke od metode iz familije metoda **glVertex**. Često korišćene metode iz ove familije su:

```
void glVertex3f(float x, float y, float z)
void glVertex2f(float x, float y)
void glVertex3v(float[] vertex)
```

gde su:

`x,y,z` – koordinate tačke, *glVertex2* podrazumeva z = 0,

`vertex` – (x,y,z) kao niz.

Da bi se tačka, a i bilo koja primitiva, iscrtala na ekranu potrebno je poziv *glVertex* metode enkapsulirati između para metoda **glBegin** i **glEnd**. Ove dve metode definišu tip primitive i enkapsuliraju niz poziva metoda *glVertex.* Za iscrtavanje tačaka mora biti prosleđena vrednost *GL_POINTS* kao argument *glBegin* metode. Listing 2.1.1 prikazuje programski kod koji iscrtava tri tačke.

```
gl.Begin(OpenGL.GL_POINTS);
  gl.Vertex(0.0f, 0.0f);
  gl.Vertex(50.0f, 50.0f);
  gl.Vertex(-50.0f, 50.0f);
gl.End();
```

Listing 2.1.1 Programski kôd za crtanje tri tačke

Podrazumevana vrednost veličine tačke iznosi jedan piksel (kvadratnog oblika!). Ako se želi promeniti veličina tačke tada je potrebno pozvati metodu **glPointSize**.

```
void glPointSize(float size)
```

gde je:

`size` – veličina tačke.

Veličina tačke ne može biti proizvoljna i zavisi od OpenGL implementacije. Najmanja i najveća podržana veličina tačke je sadržana u promenljivoj stanja: *GL_POINT_SIZE_RANGE*, a korak – inkrement u *GL_POINT_SIZE_GRANULITY*. Listing 2.1.2 prikazuje programski kôd za utvrđivanje podržanih veličina tačke:

```
float[] min_max = new float[2];
float[] korak = new float[1];

gl.GetFloat(OpenGL.GL_POINT_SIZE_RANGE,
                min_max);
gl.GetFloat(OpenGL.GL_POINT_SIZE_GRANULARITY,
                korak);
```

Listing 2.1.2 Programski kôd za utvrđivanje podržanih veličina tačke

Na tačku ne deluje perspektiva tj. tačka je iste veličine bez obzira gde je pozicionirana u prostoru.

Crtanje linija se realizuje korišćenjem metode *glVertex* za definisanje temena linija enkapsulirane *glBegin*/*glEnd* metodama sa parametrima *GL_LINES, GL_LINE_STRIP* i *GL_LINE_LOOP.*

GL_LINES iscrtava niz odvojenih linija definisanih preko svojih temena. S obzirom da je linija definisana sa dve tačke, ukupan broj tačaka mora biti paran. Linija se povlači između dva susedna temena (po redosledu navođenja poziva *glVertex* metode). Listing 2.1.3 prikazuje primer crtanja dve linije.

```
gl.Begin(Gl.GL_LINES);
  gl.Vertex(0.0f, 0.0f);
  gl.Vertex(50.0f, 50.0f);
  gl.Vertex(20.0f, 10.0f);
  gl.Vertex(-50.0f, -50.0f);
gl.End();
```

Listing 2.1.3 Programski kôd za crtanje dve linije

GL_LINE_STRIP iscrtava otvorenu polilinju. Ne postoji ograničenje da broj temena mora biti paran broj, kao kod *GL_LINES*. Segment polilinije se povlači između dva susedna temena (po redosledu navođenja poziva *glVertex* metode). Listing 2.1.4 prikazuje primer crtanja slova M.

```
gl.Begin(OpenGL.GL_LINE_STRIP);
  gl.Vertex2(0.0f, 0.0f);
  gl.Vertex2(0.0f, 50.0f);
  gl.Vertex2(15.0f, 0.0f);
  gl.Vertex2(30.0f, 50.0f);
  gl.Vertex2(30.0f, 0.0f);
gl.End();
```

Listing 2.1.4 Programski kôd za crtanje polilinije u obliku slova M

GL_LINE_LOOP iscrtava zatvorenu polilinju. Razlika između *GL_LINE_STRIP* i *GL_LINE_LOOP* je u tome što za ista temena druga iscrtava i segment koji spaja prvo i poslednje teme. Listing 2.1.5 prikazuje primer crtanja trougla (polovine slova M).

```
gl.Begin(OpenGL.GL_LINE_LOOP);
  gl.Vertex(0.0f, 0.0f);
  gl.Vertex(0.0f, 50.0f);
  gl.Vertex(15.0f, 0.0f);
gl.End();
```

Listing 2.1.5 Primer crtanja zatvorene polilinije u obliku trougla

Podrazumevana debljina linije iznosi jedan piksel, a može se promeniti pozivom metode **glLineWidth**.

```
void glLineWidth(float width)
```

gde je:

`width` – debljina linije.

Slično kao i kod definisanja veličine tačke, i debljina linije ne može biti proizvoljna i zavisi od OpenGL implementacije. Najmanja i najveća podržana debljina linije je sadržana u promenljivoj stanja: *GL_LINE_WIDTH_RANGE*, a korak – inkrement u *GL_LINE_WIDTH_GRANULARITY*. Listing 2.1.6 prikazuje programski kôd za utvrđivanje podržanih debljina linije.

```
float[] min_max = new float[2];
float[] korak = new float[1];

gl.GetFloat(OpenGL.GL_LINE_WIDTH_RANGE,
                min_max);
gl.GetFloat(OpenGL.GL_LINE_WIDTH_GRANULARITY,
korak);
```

Listing 2.1.6 Programski kôd za utvrđivanje podržanih debljina linije

Linija se može iscrtavati prema šablonu (*pattern*), kao npr. isprekidana linija. Za to se koristi metoda **glLineStipple**, koja je oblika:

```
void glLineStipple(int factor,
                   short pattern)
```

gde su:

`factor` – faktor skaliranja šablona (koliko puta se povećava širina šablona, npr. ako je *factor* jednak pet tada jednom pikselu šablona odgovara pet piksela ispune linije).

`pattern` – šablon definisan kao niz bitova (1 – ima linije, 0 – nema linije. npr. 0101010101010101 crta svaku drugu tačku linije – tačkasta linija). Šablon se zadaje kao 16-bit integer i parsira se s desna na levo!

Za izlazak iz režima se koristi metoda *glDisable* sa argumentom *GL_LINE_STIPPLE*. Listing 2.1.7 prikazuje primer crtanja isprekidane linije korišćenjem šablona.

```
// sablon: isprekidana linija
ushort sablon = 0x5555;

// udji u rezim rada sa sablonima
gl.Enable(OpenGL.GL_LINE_STIPPLE);

gl.LineStipple(1, sablon);  // def. sablon

// crtanje linija
gl.Begin(OpenGL.GL_LINES);

  gl.Vertex2(-80.0f, y);

  gl.Vertex2(80.0f, y);

gl.End();

// izadji iz rezima rada sa sablonima
```

```
gl.Disable(OpenGL.GL_LINE_STIPPLE);
```

Listing 2.1.7 Primer crtanja isprekidane linije

Pomoću linija se može nacrtati bilo koje grafički objekat, ali tako nacrtan objekat ne može biti ispunjen bojom, niti mu se može pridružiti tekstura. Za grafički objekat iscrtan pomoću linija se kaže se da je *wireframe* objekat.

2.2 Crtanje poligona

U ovom poglavlju biće prezentovano crtanje poligona: trouglova, četvorouglova i n-touglova.

Crtanje trouglova se realizuje korišćenjem metode *glVertex* za definisanje temena trouglova enkapsulirane sa *glBegin*/*glEnd* metodama sa nekim od parametara: *GL_TRIANGLES, GL_TRIANGLE_STRIP* i *GL_TRIANGLE_FAN*.

GL_TRIANGLES iscrtava niz odvojenih trouglova definisanih preko svojih temena. S obzirom da je trougao definisan sa tri tačke, ukupan broj tačaka mora biti umnožak broja tri. Trougao određuju tri susedna temena (po redosledu navođenja poziva *glVertex* metode). Redosled navođenja temena trougla određuje orijentaciju strana trougla. Listing 2.2.1 prikazuje primer crtanja dva odvojena trougla.

```
gl.Begin(OpenGL.GL_TRIANGLES);
  gl.Vertex(0.0f, 0.0f);
  gl.Vertex(0.0f, 50.0f);
  gl.Vertex(15.0f, 0.0f);
  gl.Vertex(10.0f, 10.0f);
  gl.Vertex(25.0f, 50.0f);
  gl.Vertex(15.0f, 10.0f);
gl.End();
```

Listing 2.2.1 Programski kôd za crtanje dva odvojena trougla

GL_TRIANGLE_STRIP iscrtava niz trouglova koji imaju zajedničke ivice. Ova primitiva se najčešće koristi za iscrtavanje *mesh* objekata. Svaki trougao, osim prvog, u nizu je određen sa prethodna dva temena u nizu i jednim novim temenom. N trouglova se opisuju preko *GL_TRIANGLE_STRIP* sa N+2 temena. Listing 2.2.2 prikazuje primer crtanja kvadrata pomoću dva trougla.

```
gl.Begin(OpenGL.GL_TRIANGLE_STRIP);
  gl.Vertex(0.0f, 0.0f);
  gl.Vertex(0.0f, 50.0f);
```

```
    gl.Vertex(50.0f, 0.0f);
    gl.Vertex(50.0f, 50.0f);
  gl.End();
```

Listing 2.2.2 Programski kôd za crtanje kvadrata pomoću trouglova (*strip*)

GL_TRIANGLE_FAN iscrtava niz trouglova koji svi imaju jedno zajedničko teme. Svaki trougao, osim prvog, se definiše sa jednim temenom. Listing 2.2.3 prikazuje primer crtanja šestougla stranica dužine deset jedinica (*square_root_3* je konstanta koja sadrži vrednost kvadratnog korena iz broja tri).

```
gl.Begin(OpenGL.GL_TRIANGLE_FAN);
  gl.Vertex(0.0f, 0.0f);
  gl.Vertex(-10.0f, 0.0f);
  gl.Vertex(-5.0f, -5.0f * square_root_3);
  gl.Vertex(5.0f, -5.0f * square_root_3);
  gl.Vertex(10.0f, 0.0f);
  gl.Vertex(5.0f, 5.0f * square_root_3);
  gl.Vertex(-5.0f, 5.0f * square_root_3);
  gl.Vertex(-10.0f, 0.0f);
gl.End();
```

Listing 2.2.3 Primer crtanja pravilnog šestougla pomoću trouglova (*fan*)

Trouglovi uglavnom predstavljaju najbrži način za definisanje/crtanje objekata, jer je njihovo iscrtavanje hardverski ubrzano (*hardware acclerated*) kod svakog OpenGL kompatibilnog hardvera.

Crtanje četvorouglova se realizuje korišćenjem metode *glVertex* za definisanje temena četvorougla enkapsulirane sa *glBegin*/*glEnd* metodama sa nekim od parametara: *GL_QUADS* ili *GL_QUAD_STRIP*. Sva temena četvorougla moraju ležati u istoj ravni!

GL_QUADS crta niz planarnih četvrouglova. Četvorougao je definisan sa četiri temena, tako da ukupan broj tačaka mora biti umnožak broja četiri. Listing 2.2.4 prikazuje primer crtanja kvadrata stranice dimenzija pedeset jedinica.

```
gl.Begin(OpenGL.GL_QUADS);
  gl.Vertex2(0.0f, 0.0f);
  gl.Vertex2(0.0f, 50.0f);
  gl.Vertex2(50.0f, 50.0f);
  gl.Vertex2(50.0f, 0.0f);
gl.End();
```

Listing 2.2.4 Programski kôd za crtanje kvadrata pomoću *GL_QUADS*

GL_QUAD_STRIP crta niz četvorouglova koji imaju zajedničke ivice. U mnogome je ekvivalentan *GL_TRIANGLE_STRIP* primitivi. Za svaki četvorougao, osim prvog, potrebno je definisati dva temena. Listing 2.2.5 prikazuje primer crtanja pravougonika pomoću dva kvadrata.

```
gl.Begin(OpenGL.GL_QUAD_STRIP);
  gl.Vertex2(0.0f, 0.0f);
  gl.Vertex2(0.0f, 50.0f);
  gl.Vertex2(50.0f, 0.0f);
  gl.Vertex2(50.0f, 50.0f);
  gl.Vertex2(100.0f, 0.0f);
  gl.Vertex2(100.0f, 50.0f);
gl.End();
```

Listing 2.2.5 Programski kôd za crtanje pravougonika preko *GL_QUAD_STRIP*

Crtanje planaranih **poligona** se realizuje pozivom *glBegin* sa argumentom *GL_POLYGON* i navođenjem temena unutra *glBegin*/*glEnd* sekcije. Dozvoljeni su samo konveksni poligoni (zbog različitih optimizacija koje su moguće samo nad konveksim poligonima). Konkavni poligoni se dobijaju spajanjem više konveksnih poligona. Listing 2.2.6 prikazuje primer pravilnog šestougla.

```
gl.Begin(OpenGL.GL_POLYGON);
  gl.Vertex2(-10.0f, 0.0f);
  gl.Vertex2(-5.0f, -5.0f * square_root_3);
  gl.Vertex2(5.0f, -5.0f * square_root_3);
  gl.Vertex2(10.0f, 0.0f);
  gl.Vertex2(5.0f, 5.0f * square_root_3);
  gl.Vertex2(-5.0f, 5.0f * square_root_3);
gl.End();
```

Listing 2.2.6 Primer crtanja pravilnog šestougla pomoću *GL_POLYGON*

Ispuna poligona može biti *outline* ili *solid*. *Outline* (*wireframe*) ispuna poligona crta samo njegove ivice, dok *solid* dodatno ispunjava poligon zadatom bojom. Za izbor tipa ispune poligona koristi se metoda **glPolygonMode**, koja ima oblik:

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

gde su:

`face` – specifikuje da li se odnosi na lice (*front-facing polygons*) ili naličje (*back-facing polygons*).

`mode` – da li se iscrtavaju kao okvir (*GL_LINE*) ili sa ispunom (*GL_FILL*).

Ispuna poligona može biti prema nekom šablonu. U režim ispune poligona šablonom se ulazi pozivom *glEnable* sa argumentom *GL_POLYGON_STIPPLE*. Analogno sa šablonom za linije, šablon za ispunu poligona predstavlja bitmapa dimenzija 32x32 piksela. Za definisanje samog šablona koristi se metoda **glPolygonStipple**, koja ima oblik:

```
gl.PolygonStipple(byte[] bitmap)
gl.PolygonStipple(IntPtr bitmap)
```

gde je:

`bitmap` – bitmapa koja definiše šablon.

Listing 2.2.7 prikazuje primer korišćenja režima ispune poligona šablonom.

```
// udji u rezim ispune poligona sablonom
gl.Enable(OpenGL.GL_POLYGON_STIPPLE);
// definisi sablon
gl.PolygonStipple(sablon);
// kod koji iscrtava poligon
// izadji iz rezima ispune poligona sablonom
gl.Disable(OpenGL.GL_POLYGON_STIPPLE);
```

Listing 2.2.7 Primer ispune poligona šablonom

Prilikom crtanja objekata pomoću primitiva prikaz ivica (*GL_LINE* mode) unutar objekata uglavnom nije poželjan. OpenGL nudi metodu **glEdgeFlag** kojom se može definisati da li su unutrašnje ivice vidljive.

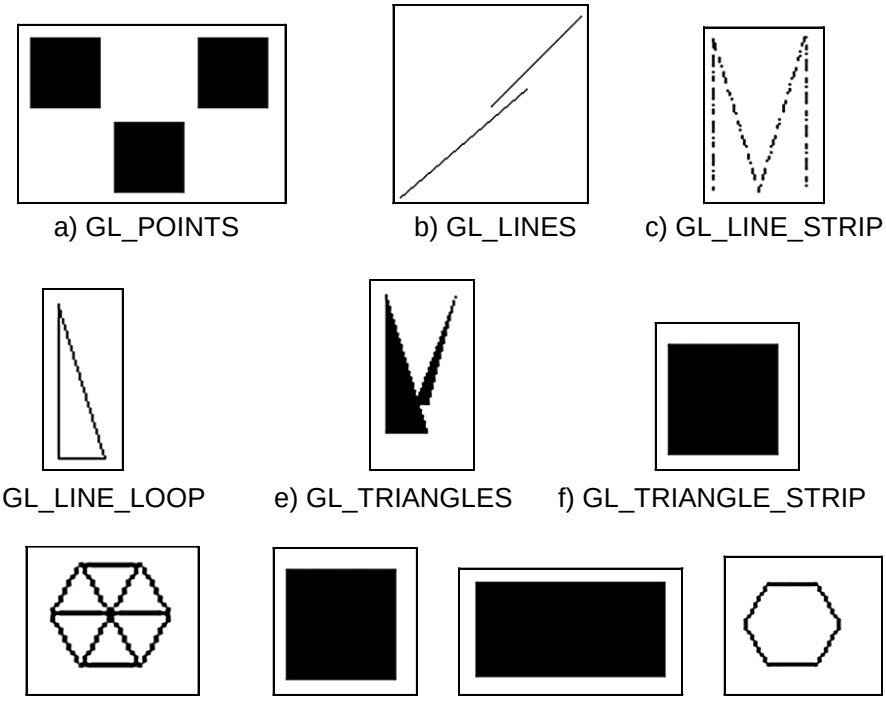
```
void glEdgeFlag(int flag)
```

gde je:

`flag` – vrednost iscrtavanja unutrašnjih linija objekta. Početna vrednost je *OpenGL.GL_TRUE.*

PREPORUKE ZA KORIŠĆENJE:

- koristiti trouglove umesto četvorouglova i poligona, jer:
  - uvek su planarni,
  - uvek su konveksni,
  - voditi računa da kombinovanje primitiva koje se iscrtavaju u suprotnim smerovima značajno usporava iscrtavan



Slika 6.3.2.1 Lista OpenGL primitiva

2.3 Crtanje mesh objekata

Za crtanje mesh objekata se najčešće koriste trouglovi, iz već navedenih razloga. Svi poligonalni objekti se mogu nacrtati preko trouglova. Kao jednostavan primer biće prezentovano crtanje kupe preko trouglova. Listing 2.3.1 prikazuje primer crtanja kupe preko trouglova. Primer je preuzet iz [2].

```
        /// <summary>
        ///  Iscrtavanje OpenGL kontrole.
        /// </summary>
        public void Draw(OpenGL gl)
        {
            // Ocisti sadrzaj kolor bafera i bafera dubine
            gl.Clear(OpenGL.GL_COLOR_BUFFER_BIT |
OpenGL.GL_DEPTH_BUFFER_BIT);

            // Ako je izabrano back face culling ukljuci ga i obratno
```

```
            if (m_culling == true)
            {
                gl.Enable(OpenGL.GL_CULL_FACE);
            }
            else
            {
                gl.Disable(OpenGL.GL_CULL_FACE);
            }

            // Ako je izabrano testiranje dubine ukljuci ga i obratno
            if (m_depthTesting == true)
            {
                gl.Enable(OpenGL.GL_DEPTH_TEST);
            }
            else
            {
                gl.Disable(OpenGL.GL_DEPTH_TEST);
            }

            // Ako je izabran rezim iscrtavanja objekta kao wireframe,
ukljuci ga i obratno
            if (m_outline == true)
            {
                // Iscrtati ceo objekat kao zicani model.
                gl.PolygonMode(OpenGL.GL_FRONT_AND_BACK,
OpenGL.GL_LINE);
            }
            else
            {
                gl.PolygonMode(OpenGL.GL_FRONT_AND_BACK,
OpenGL.GL_FILL);
            }

            double step = Math.Round(Math.PI / 8.0, 15); // Korak za
segment kruga/omotaca
            const double radius = 50.0; // radijus osnove kupe
            const double height = 75.0; // visina kupe
            double angle; // Ugao rotacije
            int pivot = 0; // Naizmenicna promena boje poligona

            // Sacuvaj stanje ModelView matrice i primeni rotacije
            gl.PushMatrix();
            gl.Rotate(m_xRotation, 1.0f, 0.0f, 0.0f);
            gl.Rotate(m_yRotation, 0.0f, 1.0f, 0.0f);

            // Zapocni iscrtavanje omotaca kupe
            gl.Begin(OpenGL.GL_TRIANGLE_FAN);

            // Da bismo dobili omotac a ne krug, postavljamo vrednost
z koordinate na visinu kupe
```

```
            gl.Vertex(0.0, 0.0, height);

            // Dodaj temena omotaca
            for (angle = 2 * Math.PI; angle > 0; angle -= step)
            {
                // Menjaj boju trouglova naizmenicno (zuta/plava)
                if ((pivot % 2) == 0)
                {
                    gl.Color(1.0f, 1.0f, 0.0f);
                }
                else
                {
                    gl.Color(0.0f, 0.0f, 1.0f);
                }

                // Azuriraj stanje boje trouglova
                ++pivot;

                gl.Vertex(radius * Math.Cos(angle), radius *
Math.Sin(angle));
            }

            // Gotovo iscrtavanje omotaca kupe
            gl.End();

            // Zapocni iscrtavanje osnove kupe
            gl.Begin(OpenGL.GL_TRIANGLE_FAN);

            // Centar osnove je u (0,0)
            gl.Vertex(0.0, 0.0);

            for (angle = 0.0; angle < 2 * Math.PI; angle += step)
            {
                // Menjaj boju trouglova (crvena/zelena)
                if ((pivot % 2) == 0)
                {
                    gl.Color(0.0f, 1.0f, 0.0f);
                }
                else
                {
                    gl.Color(1.0f, 0.0f, 0.0f);
                }

                // Azuriraj stanje boje trouglova
                ++pivot;

                // Specifikuj teme na osnovu izracunatog
                gl.Vertex(radius * Math.Cos(angle), radius *
Math.Sin(angle));
            }
```

```
            // Gotovo iscrtavanje osnove kupe
            gl.End();

            // Restauriraj stanje ModelView matrice na ono pre crtanja
kupe
            gl.PopMatrix();

            // Oznaci kraj iscrtavanja
            gl.Flush();
        }
```

Listing 2.3.1 Crtanje kupe kao *mesh* objekta

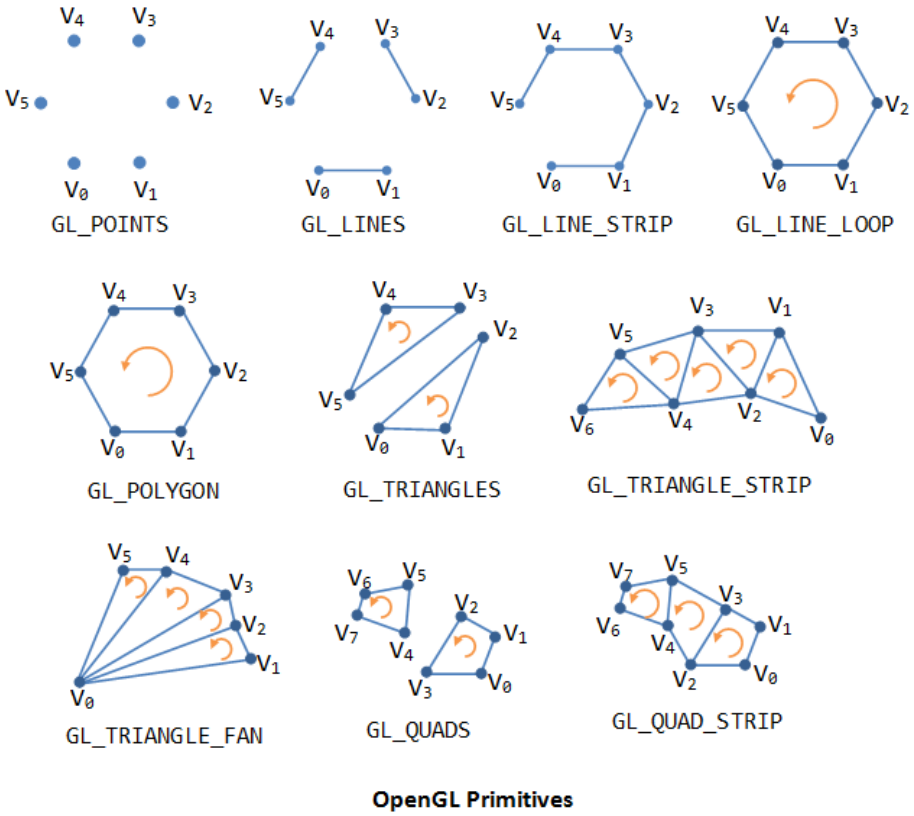
Gore navedeni primer demonstrira iscrtavanje kupe: aproksimacije kruga kao osnove (pomoću *GL_TRIANGLE_FAN*) i aproksimacije omotača (takođe *GL_TRIANGLE_FAN*). Susedni trouglovi se iscrtavaju različitim bojama radi lakšeg uočavanja.

FRONT poligoni (poligoni lica) su oni poligoni koji su na „spoljašnjosti“ objekta. **BACK poligoni** (poligoni naličja) su poligoni „unutrašnjosti“. Redosled navođenja temena određuje orijentaciju određene strane poligona, zbog čega on nije proizvoljan. Podrazumevana orijentacija iscrtavanja temena za *FRONT* stranu je *CCW* (counterclockwise), a za *BACK* stranu poligona je *CW* (clockwise). Izmena orijentacije lica poligona se postiže pozivom metode **glFrontFace** sa argumentom *GL_CW* ili*GL_CCW.* Na slici 2.3.1 je prikazan redosled navođenja temena u CCW orijentaciji.

```
public static void glFrontFace(int mode)
```

gde je:

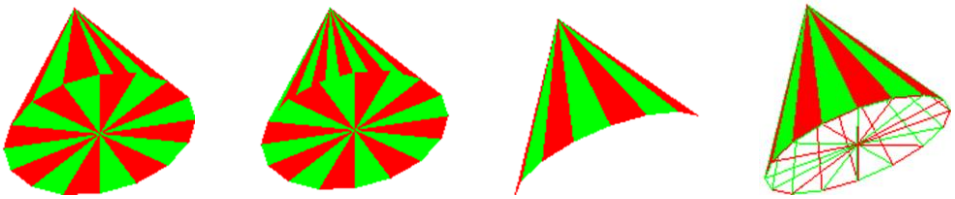
`mode` – vrednost orijentacije poligona lica. Moguće su dve vrednosti: *GL_CW* i *GL_CCW*. Početna vrednost je *GL_CCW*.



Slika 2.3.1 OpenGL primitive, iscrtane u CCW orijentaciji

Depth testing omogućava sakrivanje objekata koji su zaklonjeni nekim drugim objektom (sakrivanje po dubini). Ako nije uključen, tada je zaklonjen onaj objekat koji je ranije nacrtan (sakrivanje po vremenu iscrtavanja). U primeru se to može lepo uočiti, ako se kupa rotira oko ose da se vidi i omotač i osnova, ako nije uključen *depth testing* osnova će se videti preko omotača.

Back face culling (BFC) (sakrivanje nevidljivih površina) omogućava značajnu uštedu resursa pošto se nevidljive površine ne iscrtavaju. Utvrđivanje koja površina se iscrtava se određuje na osnovu orijentacije iscrtavanja temena poligona. Ako je BFC uključen tada se ne iscrtavaju ***BACK*** poligoni.



a) kupa sa testiranjem dubine | b) kupa bez testiranja dubine | c) kupa sa sakrivanjem nevidljivih površina | d) kupa sa osnovom iscrtanom kao žičani model

Slika 2.3.2 Rezultat rada aplikacije iz listinga 2.3.1

2.4 Crtanje punih tela i Quadric objekata

SharpGL programska biblioteka nudi metode za crtanje osnovnih geometrijskih tela, kao i nekih geometrijskih slika. Moguće je nacrtati: sferu, valjak, kupu, krug i prsten. Ovi objekti se nazivaju *quadric* objektima i SharpGL obezbeđuje posebne klase za svaki od njih. One se nalaze unutar SharpGL.SceneGraph.Quadrics imenskog prostora. Metode koje su opisane u nastavku ovog poglavlja obezbeđene su nam preko metoda klasa preko kojih u SharpGL-u radimo sa Quadric objektima. Bitna stvar za napomenuti jeste da nakon pozivanja konstruktora klase Quadric objekta koji želimo da iscrtamo, potrebno je taj objekat dodati u OpenGL kontekst preko quadricName.CreateInContext(gl).

Rad sa *quadric* objektima se razlikuje od ostalih primitiva. Definisanje parametara iscrtavanja se definiše za jedan „*template*“ objekat i dalje se primenjuje na sve objekte koji se kreiraju nakon tog „*template*“ objekta. Životni ciklus (programski kôd) tog objekta je sledeći:

- Instanciranje novog *quadrics* objekta sa:

```
Glu.GLUquadric object = Glu.gluNewQuadric();
```

- Podešavanje parametara iscrtavanja.
- Iscrtavanje objekata sa parametrima definisanim *object* promenljivom.
- Oslobađanje memorije sa:

```
Glu.gluDeleteQuadric(object);
```

Za podešavanje parametara iscrtavanja na raspolaganju su četiri metode: **gluQuadricDrawStyle, gluQuadricOrientation, gluQuadricNormals** i **gluQuadricTexture**.

Metoda **gluQuadricDrawStyle** definiše način iscrtavanja *quadric* objekta.

```
public static void gluQuadricDrawStyle(
                    GLUquadric obj, int drawStyle)
```

gde su:

`obj` – objekat na koji se metoda primenjuje,

`drawStyle` – način iscrtavanja može biti: *GL_FILL, GL_LINE, GL_POINTS i GLU_SILHOUETTE*. O prva dva stila je bilo reči kod metode *glPolygonMode*. *GL_POINTS* iscrtava objekat samo kao niz temena. *GLU_SILHOUETTE* slično kao i *GL_LINE*, ali se iscrtavaju samo određene ivice tako da se vidi samo silueta objekta.

Metode **gluQuadricNormals** i **gluQuadricOrientation** definišu tip normala, kao i smer – orijentaciju.

```
public static void gluQuadricNormals(
                    GLUquadric obj, int normals)
public static void gluQuadricOrientation(
                    GLUquadric obj, int orientation)
```

gde su:

`obj` – *quadric* objekat za koji se generišu normale,

`normals` – tip normala, može biti:

- *GLU_NONE* – ne generišu se normale,
- *GLU_FLAT* – normale su normalne na površinu (*facet*),
- *GLU_SMOOTH* – normale su tako postavljene da se dobija glatka površina objekta.

`orientation` – smer normala, može biti:

- *GLU_OUTSIDE* – smer normala je napolje (najčešće korišćeno),
- *GLU_INSIDE* – normale su usmerene u unutrašnjost *quadric* objekta. Koristi se ako želi prikazivati unutrašnjost objekta, npr. unutrašnjost kapsule svemirskog broda, koja je modelovana pomoću *quadric* objekta - sfere.

Metoda **gluQuadricTexture** omogućava automatsko generisanje koordinata teksture. Rad sa teksturama će biti detaljno objašnjen u drugom delu skripte.

```
public static void gluQuadricTexture(
                    GLUquadric obj, int textureCoords)
```

gde su:

`obj` – *quadric* objekat za koji se generišu koordinate teksture,

`textureCoords` – određuje da li se generišu koordinate teksture. Može biti:

- *GLU_TRUE* – generišu se koordinate teksture,
- *GLU_FALSE* – ne generišu se koordinate teksture.

Sfera se kreira pomoću klase **Sphere**. Centar sfere je u koordinatnom početku i sfera se iscrtava u pozitivnom smeru *z* ose.

```
public static void gluSphere(GLUQuadric obj,
              double radius, int slices, int stacks)
```

gde su:

`obj` – *quadric* objekat koji definiše parametre iscrtavanja sfere,

`radius` – radijus sfere,

`slices i stacks` – broj segmenata po x odnosno y osi. (sfera se iscrtava aproksimacijom pomoću trouglova ili četvorouglova (*quad stripes*))

Valjak se kreira pomoću klase **Cylinder**. Centar valjka osnove je u koordinatnom početku i valjak se iscrtava u pozitivnom smeru *z* ose. Ako je *topRadius* atribut klase jednak nuli tada se dobija **kupa**.

```
public static void gluCylinder(GLUquadric obj,
            double baseRadius, double topRadius,
            double height, int slices, int stacks)
```

gde su:

`obj` – objekat koji definiše parametre iscrtavanja,

`baseRadius` – radijus osnove,

`topRadius` – radijus vrha,

`height` – visina valjka,

`slices i stacks` – broj segmenata po x odnosno y osi. (valjak se iscrtava aproksimacijom pomoću trouglova ili četvorouglova (*quad stripes*))

Prsten se kreira pomoću klase **Disk**. Centar prstena je u koordinatnom početku i iscrtava se u *XY* ravni. Ako je *innerRadius* jednak nuli tada se dobija **krug**.

```
public static void gluDisk(GLUquadric obj,
            double innerRadius, double outerRadius,
            int slices, int loops)
```

gde su:

`obj` – objekat koji definiše parametre iscrtavanja,
`innerRadius` – unutrašnji radijus,
`outerRadius` – spoljašnji radijus,
`slices i loops` – broj segmenata po x odnosno y osi. (prsten se iscrtava aproksimacijom pomoću trouglova ili četvorouglova (*quad strips*))

Listing 2.4.1 prikazuje primer iscrtavanja *quadric* objekata.

```
/// <summary>
///  Iscrtavanje OpenGL kontrole.
/// </summary>
public void Draw(OpenGL gl)
{
    gl.Clear(OpenGL.GL_COLOR_BUFFER_BIT | OpenGL.GL_DEPTH_BUFFER_BIT);

    gl.LoadIdentity();

    // Podesi nacin iscrtavanja poligona
    gl.PolygonMode(m_selectedFaceMode, m_selectedPolygonMode);

    if (m_smooth)
    {
        // Podesi da tacka se iscrtava kao krug, a ne kao kvadrat
        gl.Enable(OpenGL.GL_POINT_SMOOTH);
    }
    else
    {
        gl.Disable(OpenGL.GL_POINT_SMOOTH);
    }


    switch (m_selectedPrimitive)
    {
        case OpenGLPrimitive.Cylinder:
            {
                gl.PushMatrix();
                gl.Rotate(-90f, 0f, 0f);
                Cylinder cil = new Cylinder();
                cil.CreateInContext(gl);
                cil.Render(gl, SharpGL.SceneGraph.Core.RenderMode.Render);
                gl.PopMatrix();
                break;
```

```
                    }
                case OpenGLPrimitive.Disk:
                    {
                        gl.PushMatrix();
                        gl.Translate(0f, 0f, -5f);
                        gl.Scale(0.3f, 0.3f, 0.3f);
                        gl.Rotate(30f, 0f, 0f);
                        Disk disk = new Disk();
                        disk.Loops = 120;
                        disk.Slices = 10;
                        disk.InnerRadius = 1.5f;
                        disk.OuterRadius = 2f;
                        disk.CreateInContext(gl);
                        disk.Render(gl, SharpGL.SceneGraph.Core.RenderMode.Render);
                        gl.PopMatrix();
                        break;
                    }
                case OpenGLPrimitive.Sphere:
                    {
                        gl.PushMatrix();
                        gl.Rotate(90f, 0f, 0f);
                        gl.Scale(0.5f, 0.5f, 0.5f);
                        Sphere sp = new Sphere();
                        sp.CreateInContext(gl);
                        sp.Render(gl, SharpGL.SceneGraph.Core.RenderMode.Render);
                        gl.PopMatrix();
                        break;
                    }
                default:
                    {
                        System.Console.Error.WriteLine("Greska! Odabrani mod
iscrtavanja ne postoji");
                        break;
                    }
            }
            gl.Flush();
        }
```

Listing 2.4.1 Crtanje *quadric* objekata

2.5 Crtanje objekata korišćenjem *Display List* i *Vertex Array* mehanizama

Veoma često postoji potreba za iscrtavanjem više instanci nekog objekta. Enkapsulacija programskog kôda u metode ne daje dovoljno dobre rezultate u aplikacijama gde je bitna performansa. Takođe, često se geometrija nekog objekta ne menja iz frejma u frejm.

OpenGL nudi mehanizam pomoću kojeg se može značajno ubrzati iscrtavanje složenih objekata – **Display List (DL)**-e. *DL* mehanizam omogućava enkapsulaciju i kompilaciju komandi za iscrtavanje objekata u komande hardvera. Ovaj mehanizam se bazira na strukturi OpenGL procesa prezentacije grafike (videti poglavlje 1). *DL* predstavlja niz prekompajliranih komandi na strani servera. Obično *DL* liste se kreiraju u toku inicijalizacije aplikacije.

Definisanje *DL* listi se realizuje enkapsulacijom programskog kôda za iscrtavanje objekata parom **glNewList**/**glEndList** komandi. Sledeći programski kôd prikazuje primer definisanja *DL* liste.

```
gl.NewList(list_id, OpenGL.GL_COMPILE);
    OpenGL programski kôd za isctavanje objekta
gl.EndList();
```

Svaka *DL* lista se jednoznačno identifikuje preko svog naziva koji se navodi kao prvi argument u pozivu *glNewList* (u primeru je to *list_id*). Ručna definicija identifikatora liste nije preporučljiva, jer je podložna grešci. OpenGL programska biblioteka nudi metodu **glGenLists** koja generiše jedinstvene identifikatore. Metoda *glGenLists* generiše niz uzastopnih identifikatora. Povratna vrednost iz metode je identifikator prvog objekta.

```
public static int glGenLists(int range)
```

gde je:

`range` – broj identifikatora koji se trebaju generisati.

Metoda **glDeleteLists** oslobađa zauzete identifikatore i alociranu memoriju za njihovo skladištenje.

```
public static void glDeleteLists(int list_id,
                                  int range)
```

gde su:

`list_id` – identifikator *DL* liste,

`range` – broj identifikatora koji se trebaju osloboditi.

Iscrtavanje DL liste se realizuje korišćenjem metoda **glCallList** i **glCallLists**. Razlika između metoda je u tome što poslednja omogućava izvršavanje više *DL* lista jednim pozivom.

```
public static void glCallList(int list_id)

public static void glCallLists(int n, int type,
                               IntPtr lists)
```

gde su:

`list_id` – identifikator *DL* liste,

`lists` – niz identifikatora *DL* lista,

`type` – tip podataka niza *lists,* obično *GL_UNSIGNED_BYTE*,

`n` – broj identifikatora niza *lists*.

Ugnježdavanje poziva iscrtavanja *DL* listi je dozvoljeno, ali ne i ugnježdavanje poziva kreiranja *DL* lista! Najviše može biti 64 nivoa hijerarhije.

Listing 2.5.1 prikazuje isečak programskog koda primera crtanja 1000 kopija aviona korišćenjem *DL* mehanizma.

```
using System;
using System.Collections;
using SharpGL;
using SharpGL.SceneGraph.Primitives;
using SharpGL.SceneGraph.Quadrics;
using SharpGL.Enumerations;
using SharpGL.SceneGraph.Core;
using System.Diagnostics;

//////////////////////////////////////////////////////////////////////
/
// Naziv:    DrawPlane
// Namena:   metoda iscrtava avion
// Napomena: kod preuzet od Richard S. Wright Jr., OpenGL SuperBible
//////////////////////////////////////////////////////////////////////
/
    ///<summary> Klasa koja enkapsulira OpenGL programski kod
</summary>
    class World
    {
        #region Atributi

        /// <summary>
        ///    Rotacija kocke
```

```
        /// </summary>
        private uint m_planeID = 0;


        /// <summary>
        ///    Ugao rotacije aviona
        private float m_planeRotation = 0.0f;

        /// <summary>
        ///    Referenca na OpenGL klasu unutar aplikacije
        private OpenGL gl;

        #endregion

        #region Metode

        /// <summary>
        /// Korisnicka inicijalizacija i podesavanje OpenGL
parametara
        /// </summary>
        public void Initialize(OpenGL gl)
        {
            this.gl = gl;
            gl.Enable(OpenGL.GL_DEPTH_TEST);
            gl.Enable(OpenGL.GL_CULL_FACE);
            gl.FrontFace(OpenGL.GL_CCW);
            gl.ClearColor(0.0f, 0.0f, 0.5f, 1.0f);
            // Kreiraj identifikator liste
            m_planeID = gl.GenLists(1);

            // Kreiraj listu
            gl.NewList(m_planeID, OpenGL.GL_COMPILE);
            DrawPlane(gl);
            gl.EndList();
        }

        /// <summary>
        /// Podesava viewport i projekciju za OpenGL kontrolu.
        /// </summary>
        public void Resize(OpenGL gl, int width, int height)
        {
            gl.MatrixMode(OpenGL.GL_PROJECTION);
            gl.LoadIdentity();
            gl.Perspective(45f, (double)width / height, 0.1f, 500f);
            gl.MatrixMode(OpenGL.GL_MODELVIEW);
        }

        /// <summary>
        ///  Iscrtavanje OpenGL kontrole.
        /// </summary>
```

```
        public void Draw(OpenGL gl)
        {
            gl.Clear(OpenGL.GL_COLOR_BUFFER_BIT |
OpenGL.GL_DEPTH_BUFFER_BIT);

            gl.LoadIdentity();
            gl.PushMatrix();
            gl.Rotate(m_planeRotation, 0.0f, 1.0f, 0.0f);
            // nacrtaj 1000 aviona
            for (int i = -5; i < 5; ++i)
                for (int j = -5; j < 5; ++j)
                    for (int k = -5; k < 5; ++k)
                    {
                        gl.PushMatrix();
                        gl.Translate(i * 10.0f, k * 10.0f, j *
10.0f);
                        gl.Scale(0.1f, 0.1f, 0.1f);
                        gl.CallList(m_planeID);
                        gl.PopMatrix();
                    }
            gl.PopMatrix();

            m_planeRotation += 2;
            if (m_planeRotation > 360f)
            {
                m_planeRotation = 0;

            }
        }

        static void DrawPlane(OpenGL gl)
        {
            gl.Color(1.0f, 1.0f, 0.0f);
            gl.Begin(OpenGL.GL_TRIANGLES);
            gl.Vertex(0.0f, 0.0f, 60.0f);
            gl.Vertex(-15.0f, 0.0f, 30.0f);
            gl.Vertex(15.0f, 0.0f, 30.0f);
            gl.Vertex(15.0f, 0.0f, 30.0f);
            gl.Vertex(0.0f, 15.0f, 30.0f);
            gl.Vertex(0.0f, 0.0f, 60.0f);
            gl.Vertex(0.0f, 0.0f, 60.0f);
            gl.Vertex(0.0f, 15.0f, 30.0f);
            gl.Vertex(-15.0f, 0.0f, 30.0f);
            gl.Vertex(-15.0f, 0.0f, 30.0f);
            gl.Vertex(0.0f, 15.0f, 30.0f);
            gl.Vertex(0.0f, 0.0f, -56.0f);
            gl.Vertex(0.0f, 0.0f, -56.0f);
            gl.Vertex(0.0f, 15.0f, 30.0f);
            gl.Vertex(15.0f, 0.0f, 30.0f);
            gl.Vertex(15.0f, 0.0f, 30.0f);
```

```
            gl.Vertex(-15.0f, 0.0f, 30.0f);
            gl.Vertex(0.0f, 0.0f, -56.0f);

            gl.Vertex(0.0f, 2.0f, 27.0f);
            gl.Vertex(-60.0f, 2.0f, -8.0f);
            gl.Vertex(60.0f, 2.0f, -8.0f);

            gl.Vertex(60.0f, 2.0f, -8.0f);
            gl.Vertex(0.0f, 7.0f, -8.0f);
            gl.Vertex(0.0f, 2.0f, 27.0f);

            gl.Vertex(60.0f, 2.0f, -8.0f);
            gl.Vertex(-60.0f, 2.0f, -8.0f);
            gl.Vertex(0.0f, 7.0f, -8.0f);

            gl.Vertex(0.0f, 2.0f, 27.0f);
            gl.Vertex(0.0f, 7.0f, -8.0f);
            gl.Vertex(-60.0f, 2.0f, -8.0f);

            gl.Vertex(-30.0f, -0.50f, -57.0f);
            gl.Vertex(30.0f, -0.50f, -57.0f);
            gl.Vertex(0.0f, -0.50f, -40.0f);

            gl.Vertex(0.0f, -0.5f, -40.0f);
            gl.Vertex(30.0f, -0.5f, -57.0f);
            gl.Vertex(0.0f, 4.0f, -57.0f);

            gl.Vertex(0.0f, 4.0f, -57.0f);
            gl.Vertex(-30.0f, -0.5f, -57.0f);
            gl.Vertex(0.0f, -0.5f, -40.0f);
            gl.Vertex(30.0f, -0.5f, -57.0f);
            gl.Vertex(-30.0f, -0.5f, -57.0f);
            gl.Vertex(0.0f, 4.0f, -57.0f);

            gl.Vertex(0.0f, 0.5f, -40.0f);
            gl.Vertex(3.0f, 0.5f, -57.0f);
            gl.Vertex(0.0f, 25.0f, -65.0f);

            gl.Vertex(0.0f, 25.0f, -65.0f);
            gl.Vertex(-3.0f, 0.5f, -57.0f);
            gl.Vertex(0.0f, 0.5f, -40.0f);

            gl.Vertex(3.0f, 0.5f, -57.0f);
            gl.Vertex(-3.0f, 0.5f, -57.0f);
            gl.Vertex(0.0f, 25.0f, -65.0f);
            gl.End(); // Of Jet
        }


        /// <summary>
```

```
        ///  Dispose metoda.
        /// </summary>
        public void Dispose()
        {
            this.Dispose(true);
            GC.SuppressFinalize(this);
        }

        /// <summary>
        ///  Destruktor.
        /// </summary>
        ~World()
        {
            this.Dispose(false);
        }

        #endregion

        #region IDisposable Metode

        /// <summary>
        ///  Implementacija IDisposable interfejsa.
        /// </summary>
        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {

                gl.DeleteLists(m_planeID, 1);
            }
        }

        #endregion
    }
```

Listing 2.5.1 Isečak programskog koda za crtanje aviona pomoću *DL* liste

Iako se *DL* liste prevashodno koriste za opis statične geometrije, programski kôd koji se enkapsulira može biti bilo koji skup OpenGL komandi sa izuzetkom *glNewList*/*glDeleteList*. Naime, nije dozvoljena rekurzija u definiciji *DL* lista. S obzirom da se OpenGL programski kôd prekompajlira *DL* liste nisu pogodne za opis geometrije koja se menja tokom izvršavanja. Pošto se *DL* liste smeštaju u memoriju servera nisu pogodne za objekte sa velikim brojem temena.

OpenGL nudi **Vertex Array** i **Indexed Vertex Array** mehanizme za efikasno iscrtavanje objekata čija se geometrija menja u toku izvršavanja i/ili koji imaju veliki broj temena. Ovi mehanizmi koriste nizove temena koji se prosleđuju kao jedini parametar. Ovim se značajno ubrzava iscrtavanje, jer se ne mora pozivati niz

OpenGL metoda za svako teme (specifikacija koordinata, koordinata teksture, normala i sl.).

Rad sa *vertex array* mehanizmom se realizuje kroz sledeće faze:

1. Uključenje klijentske promenljive stanja *GL_VERTEX_ARRAY* pozivom metode **glEnableClientState**. Ova metoda uključuje klijentske promenljive stanja koje su prosleđene kao argument. Suprotna akcija se realizuje pozivom **glDisableClientState**.

```
public static void glEnableClientState(
                        int cap)
```

gde je:

`cap` – simbolička konstanta klijentske promenljive stanja koja se uključuje.

2. Signalizacije koji niz temena treba da se iscrta pozivom metode **glVertexPointer**. Ova metoda signalizira OpenGL gde su smešteni podaci, kao i kog su tipa i koliko ih ima. Za specifikaciju normala i boja koriste se analogne metode: **glNormalPointer** i **glColorPointer** i sl.

```
public static void glVertexPointer(
                    int size, int type,
                    int stride, IntPtr pointer)
```

gde su:

`size` – broj koordinata po temenu. Inicijalna vrednost iznosi četiri,

`type` – tip koordinate. Dozvoljene vrednosti su: *GL_SHORT, GL_INT, GL_FLOAT* i *GL_DOUBLE.* Inicijalna vrednost je *GL_FLOAT,*

`stride` – razmak između susednih temena izražen u bajtima. Inicijalna vrednost iznosi nula,

`pointer` – niz temena.

3. Iscrtavanja niza temena pozivom metode **glDrawArrays**. Ova metoda iscrtava grafičku primitivu definisanu parametrom *mode* sa *count* temena počevši od temena sa indeksom *first*.

```
public static void glDrawArrays(int mode,
                                 int first,
                                 int count)
```

gde su:

`mode` – koju primitivu treba iscrtavati tj. kako tumačiti temena,

`first` – indeks početnog temena,

`count` – broj temena koja se iscrtavaju.

4. Isključivanje *vertex array* mehanizma pozivom **glDisableClientState** sa argumentom *GL_VERTEX_ARRAY.*

```
public static void glDisableClientState(
                        int cap)
```

gde je:

`cap` – simbolička konstanta klijentske promenljive stanja koja se isključuje.

U indeksnoj verziji postoje dva niza temena: niz indeksa i niz jedinstvenih temena. Ovaj pristup često obezbeđuje bolju performansu. Naime, u velikom broju slučaja kod *mesh* objekata broj temena se ponavlja jer se poligoni nastavljaju jedan na drugi. S toga umesto prosleđivanja niza svih temena od kojih se veliki broj ponavlja, prosleđuje se niz jedinstvenih (međusobno različitih) temena i niz indeksa tih temena. Rad sa indeksnom verzijom *vertex array* mehanizma se razlikuje od obične verzije u fazi tri gde se niz iscrtava pozivom metode **glDrawElements**.

```
public static void glDrawElements(int mode,
                int  count, int type, IntPtr indices)
```

gde su:

`mode` – koju primitivu treba iscrtavati tj. kako tumačiti temena,

`type` – tip podataka indeks niza. Dozvoljene vrednosti su: *GL_UNSIGNED_BYTE*, *GL_UNSIGNED_SHORT* i *GL_UNSIGNED_INT*,

`count` – broj temena koja se iscrtavaju,

`indices` – niz indeksa.

Listing 2.5.2 prikazuje isečak programskog kôda primera iscrtavanja 1000 *mesh* modela aviona korišćenjem *Vertex Array* i *Indexed Vertex Array* mehanizama.

```
        #region Atributi

        // indeksirana verzija, niz ima elemenata onoliko koliko je
temena = 51
        static uint[] indices = { 1, 10, 9,
       9, 4, 1,
```

```
        1, 4, 10,
        10, 4, 0,
        0, 4, 9,
        9, 10, 0,
        5, 16, 15,
        15, 8, 5,
        15, 16, 8,
        5, 8, 16,
        14, 13, 3,
        3, 13, 7,
        7, 14, 3,
        13, 14, 7,
        3, 11, 6,
        6, 12, 2,
        11, 12, 6};

         // razlicitih temena ima 17
         static float[] uniqueVertices = { 0.0f, 0.0f, -56.0f,    // 0
         0.0f, 0.0f, 60.0f,     // 1
         0.0f, 0.5f, -40.0f,    // 2
         0.0f, -0.5f, -40.0f,   // 3
         0.0f, 15.0f, 30.0f,    // 4
         0.0f, 2.0f, 27.0f,     // 5
         0.0f, 25.0f, -65.0f,   // 6
         0.0f, 4.0f, -57.0f,    // 7
         0.0f, 7.0f, -8.0f,     // 8
         15.0f, 0.0f, 30.0f,    // 9
         -15.0f, 0.0f, 30.0f,   // 10
         3.0f, 0.5f, -57.0f,    // 11
         -3.0f, 0.5f, -57.0f,   // 12
         30.0f, -0.5f, -57.0f,  // 13
         -30.0f, -0.5f, -57.0f, // 14
         60.0f, 2.0f, -8.0f,    // 15
         -60.0f, 2.0f, -8.0f};  // 16

         // neindeksirana verzija, 51 teme
         static float[] vertices = {
       0.0f, 0.0f, 60.0f, -15.0f, 0.0f, 30.0f, 15.0f, 0.0f, 30.0f,
       15.0f, 0.0f, 30.0f, 0.0f, 15.0f, 30.0f, 0.0f, 0.0f, 60.0f,
       0.0f, 0.0f, 60.0f, 0.0f, 15.0f, 30.0f, -15.0f, 0.0f, 30.0f,
       -15.0f, 0.0f, 30.0f, 0.0f, 15.0f, 30.0f, 0.0f, 0.0f, -56.0f,
       0.0f, 0.0f, -56.0f, 0.0f, 15.0f, 30.0f, 15.0f, 0.0f, 30.0f,
       15.0f, 0.0f, 30.0f, -15.0f, 0.0f, 30.0f, 0.0f, 0.0f, -56.0f,
       0.0f, 2.0f, 27.0f, -60.0f, 2.0f, -8.0f, 60.0f, 2.0f, -8.0f,
       60.0f, 2.0f, -8.0f, 0.0f, 7.0f, -8.0f, 0.0f, 2.0f, 27.0f,
       60.0f, 2.0f, -8.0f, -60.0f, 2.0f, -8.0f, 0.0f, 7.0f, -8.0f,
       0.0f, 2.0f, 27.0f, 0.0f, 7.0f, -8.0f, -60.0f, 2.0f, -8.0f,
       -30.0f, -0.5f, -57.0f, 30.0f, -0.5f, -57.0f, 0.0f, -0.5f, -
40.0f,
       0.0f, -0.5f, -40.0f, 30.0f, -0.5f, -57.0f, 0.0f, 4.0f, -57.0f,
```

```
      0.0f, 4.0f, -57.0f, -30.0f, -0.5f, -57.0f, 0.0f, -0.5f, -40.0f,
      30.0f, -0.5f, -57.0f, -30.0f, -0.5f, -57.0f, 0.0f, 4.0f, -57.0f,
      0.0f, 0.5f, -40.0f, 3.0f, 0.5f, -57.0f, 0.0f, 25.0f, -65.0f,
      0.0f, 25.0f, -65.0f, -3.0f, 0.5f, -57.0f, 0.0f, 0.5f, -40.0f,
      3.0f, 0.5f, -57.0f, -3.0f, 0.5f, -57.0f, 0.0f, 25.0f, -65.0f };


        private float yRot;

        // da li crtamo sa ili bez koriscenja indeksne verzije vertex
arrays
        private bool g_usingIndexedVertexArrays = true;

        #endregion

        #region Metode

        /// <summary>
        /// Korisnicka inicijalizacija i podesavanje OpenGL parametara
        /// </summary>
        public void Initialize(OpenGL gl)
        {
            gl.Enable(OpenGL.GL_DEPTH_TEST);
            gl.Enable(OpenGL.GL_CULL_FACE);
            gl.FrontFace(OpenGL.GL_CCW);
            gl.ClearColor(0.0f, 0.0f, 0.5f, 1.0f);
        }

        /// <summary>
        /// Podesava viewport i projekciju za OpenGL kontrolu.
        /// </summary>
        public void Resize(OpenGL gl, int width, int height)
        {
            gl.MatrixMode(OpenGL.GL_PROJECTION);
            gl.LoadIdentity();
            gl.Perspective(45f, (double)width / height, 0.1f, 500f);
            gl.MatrixMode(OpenGL.GL_MODELVIEW);
        }

        /// <summary>
        ///  Iscrtavanje OpenGL kontrole.
        /// </summary>
        public void Draw(OpenGL gl)
        {
            gl.Clear(OpenGL.GL_COLOR_BUFFER_BIT |
OpenGL.GL_DEPTH_BUFFER_BIT);
            gl.PushMatrix();
            gl.Translate(0f, 0f, -20f);
            gl.Rotate(yRot, 0.0f, 1.0f, 0.0f);
```

```
            // Ukljuci rad sa vertex array mehanizmom
            gl.EnableClientState(OpenGL.GL_VERTEX_ARRAY);

            // Nacrtaj 1000 aviona
            for (int i = -5; i < 5; ++i)
                for (int j = -5; j < 5; ++j)
                    for (int k = -5; k < 5; ++k)
                    {
                        gl.PushMatrix();
                        gl.Translate(i * 10.0f, k * 10.0f, j * 10.0f);
                        gl.Scale(0.1f, 0.1f, 0.1f);
                        // U zavisnosti od izbora crtamo sa ili bez DL
                        if (g_usingIndexedVertexArrays == false)
                        {
                            gl.VertexPointer(3, 0, vertices);
                            gl.DrawArrays(OpenGL.GL_TRIANGLES, 0, 51);
                        }
                        else
                        {
                            gl.VertexPointer(3, 0, uniqueVertices);
                            gl.DrawElements(OpenGL.GL_TRIANGLES, 51,
indices);
                        }
                        gl.PopMatrix();
                    }
            // iskljuci rad sa vertex array mehanizmom
            gl.DisableClientState(OpenGL.GL_VERTEX_ARRAY);
            gl.PopMatrix();

            yRot += 2;
            if (yRot > 360)
                yRot = 0;
        }
```

Listing 2.5.2 Isečak programskog kôda za crtanje aviona korišćenjem *Vertex Array* mehanizma

Pored VA mehanizma postoji sličan mehanizam - Vertex Buffer Object (VBO) koji nudi dodatna podešavanja i fleksibilnost i predstavlja fuziju DL i VA mehanizama. Za razliku od VA mehanizma, VBO mehanizam kreira tzv. bafer objekte (buffer objects) na serverskoj strani (što predstavlja sličnost sa DL listama). Metode za pristup i referenciranje podataka u baferima prilikom iscrtavanja koriste iste metode kao i kod VA mehanizma.

Ovaj mehanizam omogućava kontrolu nad transferom podataka. Podržane su tri mogućnosti: *GL_STREAM_DRAW*, *GL_STATIC_DRAW* i *GL_DYNAMIC-_DRAW.*

GL_STREAM_DRAW režim radi na isti način kao VA mehanizam – podaci se šalju prilikom svakog iscrtavanja (pozivi metoda: glDrawArrays, glDrawElements). Ovaj režim je pogodan za iscrtavanje animiranih objekata u sceni.

GL_STATIC_DRAW režim šalje podatke grafičkom kontroleru i oni se skladište u memoriji grafičke kartice. Ovaj režim najviše pogoduje za iscrtavanje nedeformabilnih (statičkih) objekata.

Za razliku od prethodnih u *GL_DYNAMIC_DRAW* režimu drajveri grafičke kartice određuju gde i kako će biti podaci uskladišteni.

Rad sa *VBO* mehanizmom se realizuje kroz sledeće faze:

Uključenje klijentske promenljive stanja *GL_VERTEX_ARRAY* pozivom metode **glEnableClientState**.

Kreiranje bafera metodom: **glGenBuffers.**

```
public static void glGenBuffers(
                int size, IntPtr buffers)
public static void glGenBuffers(
                int size, int[] buffers)
```

gde su:

`size` – broj bafera koji se alocira,

`buffers` – niz u koji se smešta niz identifikatora bafera koji se alociraju.

Proglasiti bafer aktivnim pozivom metode **glBindBuffer.**

```
public static void glBindBuffer(
                int target, int buffer)
public static void glBindBuffer(
                int target, uint buffer)
```

gde su:

`target` – određuje na koji tip bafera se vezuje. Moguće vrednosti su: GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER i GL_PIXEL_UNPACK_BUFFER.

`buffer` – identifikator aktivnog bafera.

Inicijalizacija bafera i specifikacija inicijalnih podataka sa **glBufferData**.

```
public static void glBufferData(
```

```
                int target, IntPtr size,
                IntPtr data, int usage)
public static void glBufferData(
                int target, IntPtr size,
                IntPtr data, int usage)
```

gde su:

`target` – određuje na koji tip bafera se vezuje. Moguće vrednosti su: GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_PIXEL_PACK_BUFFER i GL_PIXEL_UNPACK_BUFFER,

`size` – veličina u bajtima objekta u baferu,

`data` – podaci koji će inicijalno biti kopirani,

`usage` – specifikuje režim upotrebe: GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY, GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY, GL_DYNAMIC_DRAW, GL_DYNAMIC_READ i GL_DYNAMIC_COPY.

Deaktiviranje bafera, da ne bi pozivi metoda iscrtavanja sa **glBindBuffer** gde se umesto podataka šalje 0 (zero pointer).

Iscrtavanje sadržaja bafera se realizuje tako što se bafer proglasi aktivnim, a zatim pozivima *glVertexPointer* i *glDrawArrays* na sličan način iscrta. Na primer:

```
// Ukljuci rad sa vertex array mehanizmom
gl.EnableClientState(OpenGL.GL_VERTEX_ARRAY);
// Povezi vec kreirani bafer
gl.BindBuffer(OpenGL.GL_ARRAY_BUFFER, m_modelsVBO[0]);
// Neophodno zbog vertex array-a - iskljucuje postojeci
vertexpointer

gl.VertexPointer(VERTEX_COMPONENT_COUNT, OpenGL.GL_FLOAT, 0,
IntPtr.Zero);
// Iscrtaj sadrzaj bafera
gl.DrawArrays(OpenGL.GL_TRIANGLES, 0, Porsche.vertices.Length /
VERTEX_COMPONENT_COUNT);
// Unbind bafer
gl.BindBuffer(OpenGL.GL_ARRAY_BUFFER, 0);
// Iskljuci rad sa vertex array mehanizmom
gl.DisableClientState(OpenGL.GL_VERTEX_ARRAY);
```

Na kraju potrebno je osloboditi bafer pozivom metode **glDeleteBuffers**.

```
public static void glDeleteBufffers(int n,
                    int[] buffers)
public static void glDeleteBufffers(int n,
                    IntPtr buffers)
public static void glDeleteBufffers(int n,
                    ref int buffers)
public static void glDeleteBufffers(int n,
                    uint[] buffers)
public static void glDeleteBufffers(int n,
                    ref uint buffers)
```

gde su:

`n` – broj bafera koji se oslobađaju,

`buffers` – niz identifikatora bafera koji se oslobađaju.

Isključivanje *vertex array* mehanizma pozivom **glDisableClientState** sa argumentom *GL_VERTEX_ARRAY.*

```
public static void glDisableClientState(
                    int cap)
```

gde je:

`cap` – simbolička konstanta klijentske promenljive stanja koja se isključuje.

S obzirom da predstavlja kombinaciju najboljih osobina DL i VA mehanizma predstavlja preporuku za korišćenje.

3 Vrste projekcije

SharpGL obezbeđuje način da se definiše koja metoda treba da bude pozvana kada se promeni veličina prozora, tj. registruje povratni poziv za preračunavanje projekcije. Pored toga, ta metoda se poziva i kod inicijalnog kreiranja prozora. Ako se ne definiše ova metoda tada dolazi do deformacije objekata koji se iscrtavaju prilikom izmene dimenzija prozora. Ukoliko koristimo kontrolu unutar WPF aplikacije, imamo opciju definisanja funkcije koja će se pozvati prilikom okidanja Resized događaja.

```
        private void openGLControl_Resized(object
sender, OpenGLEventArgs args)
        {
m_world.Resize((int)openGLControl.ActualWidth,
(int)openGLControl.ActualHeight);
        }
```

gde su:

`openGLControl.ActualWidth` – širina WPF kontrole,
`openGLControl.ActualHeight` – visina WPF kontrole.

Metoda **Resize** klase **World** definiše kako će se skalirati prozor pri promeni veličine prozora od strane korisnika. U ovoj metodi koriste se metode *glMatrixMode* i *glLoadIdentity* koje će biti detaljno objašnjene u narednom poglavlju. Uobičajno je da metoda **Resize** ima sledeću formu:

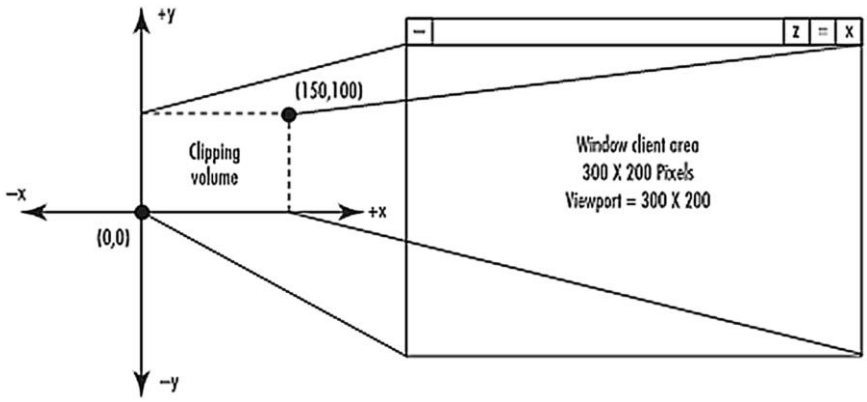
- Definisanje mapiranja logičkih koordinata u fizičke koordinate – definisanje *viewport*-a korišćenjem metode *glViewport*.
- Definisanje projekcije. Projekcijom se definiše koji deo 3D prostora će biti predstavljen unutar prozora. Postoje dve vrste projekcija: ortogonalna i projekcija u perspektivi. Obe projekcije će biti detaljno opisane kod odgovarajućih OpenGL metoda kojima se definišu.

Metoda **glViewport** definiše mapiranje kliping (*clipping*) prozora u fizički prozor aplikacije – mapiranje logičkog koordinatnog sistema u fizički (slika 3.1).

```
public static void glViewport(int x, int y,
                              int width, int height)
```

gde su:

`x, y` – donja leva koordinata, najčešće (0,0),
`width` – širina kliping prozora.,
`height` – visina kliping prozora.



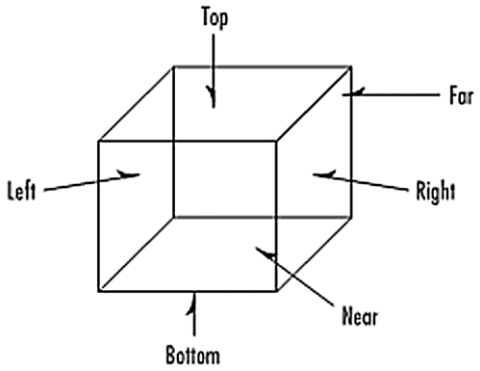
Slika 3.1 Mapiranje logičkih u fizičke koordinate prozora

Metoda **glOrtho** definiše kliping prozor sa **ortogonalnom projekcijom**, slika 3.2. Prostor je definisan preko šest parametara – dimenzija kvadra. U ortogonalnoj projekciji svi objekti koji su istih dimenzija prikazuju se u istoj veličini bez obzira gde se nalaze. Najčešće ova projekcija se koristi u *CAD* i arhitektonskom dizajnu.

```
public static void glOrtho(double left,
       double right, double bottom, double top,
       double near, double far)
```

gde su:

`left, right` – minimum/maksimum po x-osi,
`bottom, top` – minimum/maksimum po y-osi,
`near, far` – minimum/maksimum po z-osi.



Slika 3.2 Ortogonalna projekcija

Pored *glOrtho* metode u upotrebi je i metoda **gluOrtho2D**, koja definiše „2D“ projekciju. Metoda *gluOrtho2D* predstavlja specijalni slučaj gde su parametar *near* jednak -1, a parametar *far* jednak 1.

```
public static void gluOrtho2D(double left,
            double right, double bottom, double top)
```

gde su:

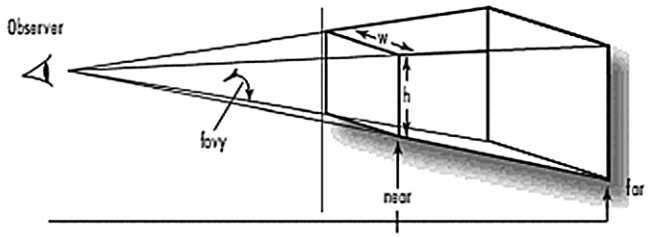
`left, right` – minimum/maksimum po x-osi,
`bottom, top` – minimum/maksimum po y-osi.

Za razliku od ortogonalne, koristi se još i **projekcija u perspektivi**, slika 3.3. U perspektivi objekti koji su dalji od posmatrača se prikazuju manjim i obratno. Ova projekcija omogućava stvaranje realističnog utiska sveta. Metoda **gluPerspective** definiše projekciju u perspektivi.

```
public static void gluPerspective(double fovy,
                                  double aspect,
                                  double zNear,
                                  double zFar)
```

gde su:

`fovy` – ugao tj. dubina vidnog polja (*field-of-view*),
`aspect` – odnos širine i visine,
`zNear, zFar` – dubina po z-osi.



Slika 3.3 Projekcija u perspektivi

S obzirom da se tekst ispisuje kao vektorska grafika, moguće je definisati debljinu linije korišćenjem metode *glLineWidth*.

4 Transformacije objekata

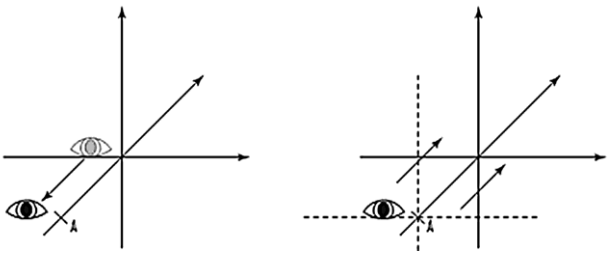
U OpenGL standardu transformacije se realizuju pomoću matrica: **Viewing**, **Modeling**, **Modelview**, **Projection** i **Viewport**. Redosled primena transformacija je fiksan i identičan redosledu navođenja matrica u prethodnoj rečenici.

Referentni koordinatni sistem je Dekartov pravougli koordinatni sistem, sa pozitivnim pravcem *x*-ose u desno, pozitivnim pravcem *y*-ose na gore i pozitivnim pravcem *z*-ose iz ravni ka korisniku (pravilo desne ruke).

Viewing matrica definiše tačku posmatranja – kameru (transformiše koordinatni sistem). **Modeling** matrica omogućava transformacije nad modelom i njegovim objektima.

Modelview matrica integriše transformacije koje vrše prethodne dve matrice i olakšava transformacije nad scenom. Ovo proizilazi iz činjenice da sa aspekta posmatrača je svejedno da li se kamera premestila ili su primenjene transformacije nad objektima ako je rezultujući pogled na scenu identičan u oba slučaja, slika 4.1.

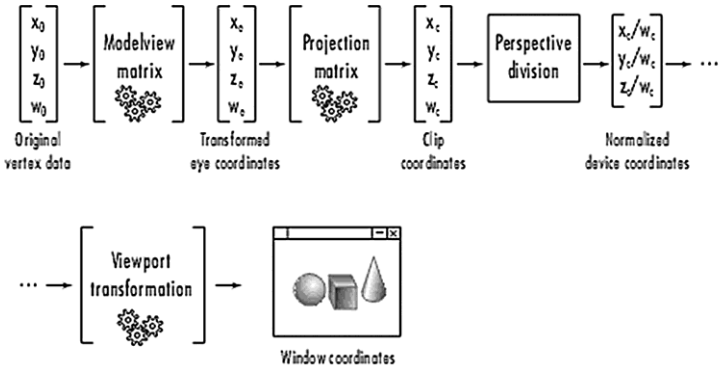
Nakon ovih transformacija primenjuju se transformacije projekcije. **Projection** matrica definiše vidljivi volumen, kao i tip projekcije scene na ekran (ortogonalna ili projekcija u perspektivi). **Viewport** matrica definiše mapiranje 2D projekcije scene u prozor u kojem se prikazuje. Najčešće se manipulacije vrši nad *Modelview* i *Projection* matricama. Grafička predstava procesa primena transformacija nad temenom je prikazana na slici 4.2.



a) pomeranje kamere

b) pomeranje koord. sistema

Slika 4.1 Ekvivalencija transformacija kamere i koord. sistema



Slika 4.2 Redosled primene transformacija nad temenom

Izbor matrice nad kojom se vrše transformacije se realizuje pomoću metode **glMatrixMode**.

```
public static void glMatrixMode(int mode)
```

gde je:

`mode` – koja matrica se selektuje. Između ostalih moguće su vrednosti: *GL_MODELVIEW, GL_PROJECTION* kojima se selektuje odgovarajuće matrice.

Translacija se realizuje pomoću familije metoda **glTranslate**, od kojih je najčešće korišćena metoda *glTranslatef.*

```
public static void glTranslatef(float x, float y,
                                 float z)
```

gde su:

`x, y, z` – pomeraj po x,y,z osama.

Rotacija se realizuje pomoću familije metoda **glRotate**, od kojih je najčešće korišćena metoda *glRotatef.*

```
public static void glRotatef(float angle, float x,
                             float y, float z)
```

gde su:

`angle` – ugao rotacije izražen u stepenima,

`x, y, z` – vektor u odnosu na koji se vrši rotacija (vektor je definisan sa (0,0,0) i (x,y,z)).

Skaliranje se realizuje pomoću familije metoda **glScale**, od kojih je najčešće korišćena metoda *glScalef.* Uniformno skaliranje se realizuje sa x = y = z.

```
public static void glScalef(float x, float y,
```

```
                                    float z)
```

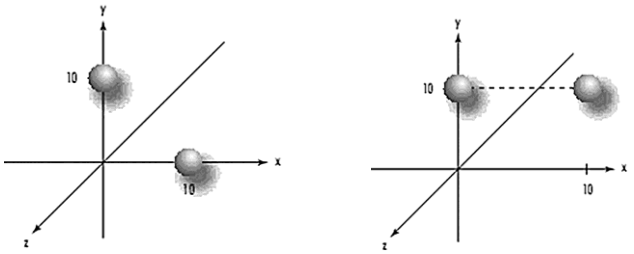
gde su:

`x, y, z` – faktori skaliranja po svakoj od osa.

Primena transformacija u OpenGL standardu je **kumulativna**. Samim tim, jednostavna primena transformacija dovodi do nepredvidivih rezultata. Npr. ako se žele iscrtati sfere pozicionirane kao na slici 4.3a, kao logična realizacija se nameće sledeći programski kôd:

```
// translacija za 10 jedinica po y-osi
gl.Translate(0.0f, 10.0f, 0.0f);
Sphere sfera = new Sphere();
sfera.CreateInContext(gl);
sfera.Radius = 10f;
// iscrtaj prvu sferu
sfera.Render(gl, RenderMode.Render);
// translacija za 10 jedinica po x-osi
gl.Translate(10.0f, 0.0f, 0.0f);
// iscrtaj drugu sferu
sfera.Render(gl, RenderMode.Render);
```

Problem nastaje što gore navedeni programski kôd zbog kumulativnosti primene transformacija daje rezultat prikazan na slici 6.4.3b.



a) željena pozicija b) rezultujuća pozicija

Slika 4.3 Kumulativnost transformacija

Da bi se izbegao problem kumulativnosti potrebno je imati mogućnost resetovanja matrice. U te svrhe se koristi metoda **glLoadIdentity**. Iako korišćenje ove metode rešava gore pomenute probleme sama metoda je zahtevna. Mnogo bi bilo lakše da je moguće nekako sačuvati privremeno stanje matrice, primeniti izmene i posle to stanje vrati. OpenGL definiše **matrični stek** (LIFO) za tu namenu

i metode **glPushMatrix** i **glPopMatrix**. Sve operacije matričnog steka se odnose na trenutno izabranu matricu (korišćenjem metode *glMatrixMode*). OpenGL nudi niz metoda za matrični račun kojima se direktno može menjati aktivna matrica (npr. *ModelView*).

Sledeći programski kôd realizuje poziciju sfera kao na slici 4.3a:

```
// sacuvaj stanje
gl.PushMatrix();
// translacija za 10 jedinica po y-osi
gl.Translate(0.0f, 10.0f, 0.0f);
Sphere sfera = new Sphere();
sfera.CreateInContext(gl);
sfera.Radius = 10f;
// iscrtaj prvu sferu
sfera.Render(gl, RenderMode.Render);
// ucitaj sacuvano stanje
gl.PopMatrix();
// translacija za 10 jedinica po x-osi
gl.Translate(10.0f, 0.0f, 0.0f);
// iscrtaj drugu sferu
sfera.Render(gl, RenderMode.Render);
```

Slika 4.4 Rezultat rada aplikacije iz listinga 6.4.1 u različitim vremenskim trenucima

Listing 4.1 i slika 4.4 prikazuje isečak programskog kôda (*DrawSolarSystem* metodu) simulacije revolucije Zemlje oko Sunca i Meseca oko Zemlje primenom transformacija. U primeru se koristi kumulativnost transformacija. Prvo se iscrta Sunce u centru prozora. Zatim se Zemlja zarotira za m_earthRotation ugao u odnosu na Sunce, a onda se Mesec zarotira za m_moonRotation ugao u odnosu na Zemlju. Ovo je postignuto upravo korišćenjem kumulativnosti transformacija.

```
        public void Draw(OpenGL gl)
        {
            gl.Clear(OpenGL.GL_COLOR_BUFFER_BIT |
OpenGL.GL_DEPTH_BUFFER_BIT);

            gl.LoadIdentity();

            DrawSolarSystem(gl);
        }

        public void DrawSolarSystem(OpenGL gl)
        {
            Sphere sfera = new Sphere();
            sfera.CreateInContext(gl);
            sfera.Radius = 20f;
            gl.Color(1f, 0f, 0f);
            gl.PushMatrix();
            // Transliraj scenu tako da se vidi unutar prozora
            gl.Translate(0.0f, 0f, -200.0f);
            sfera.Render(gl, RenderMode.Render);

            // Rotiraj koordinatni sistem da bi iscrtali Zemlju
            gl.Rotate(m_earthRotation, 0.0f, 1.0f, 0.0f);
            // Zemlja
            gl.Translate(100.0f, 0.0f, 0.0f);
            sfera.Radius = 10f;
            sfera.Render(gl, RenderMode.Render);

            // Rotiraj za ugao Meseca i nacrtaj Mesec
            gl.Rotate(m_moonRotation, 0.0f, 1.0f, 0.0f);
            gl.Translate(30.0f, 0.0f, 0.0f);
            sfera.Radius = 5f;
            sfera.Render(gl, RenderMode.Render);
            // Restauriraj stanje ModelView matrice pre crtanja
            gl.PopMatrix();

            m_earthRotation += 2.0f;
            if (m_earthRotation > 360.0f)
            {
                m_earthRotation = 0.0f;
            }
            m_moonRotation += 5.0f;
            if (m_moonRotation > 360.0f)
            {
                m_moonRotation = 0.0f;
            }
        }
```

Listing 4.1 Primer primene transformacija – Solarni sistem

LITERATURA

1. OpenGL specification, http://www.opengl.org/documentation/specs/
2. Wright R. Jr., Lipchak B., Haemel N., „OpenGL Super Bible“, 4th Ed.