UNIVERZA V LJUBLJANI
Fakulteta za strojništvo

Senčenje 3-D objekta z upodobitvijo

DIPLOMSKA NALOGA VISOKOŠOLSKEGA ŠTUDIJA

Leon Kos

Mentor: doc. dr. Jože Duhovnik, dipl. ing.

Ljubljana 1991

Op.: Namesto te strani vloži orginal oz. kopijo podpisane naloge

Za pomoč pri izdelavi diplomske naloge se zahvaljujem mentorju doc.dr. Jožetu Duhovniku.

(D) Tek. štev.: 3963 UDK:

SENČENJE 3-D OBJEKTA Z UPODOBITVIJO

Leon Kos

Ključne besede:

računalništvo, programiranje, obdelava slike, grafika, software, 3D, senčenje, Bézier, C jezik

Povzetek:

Diplomska naloga poskuša prikazati osnovne principe matematične formulacije masivnih objektov v tridimenzionalnem prostoru in njihovo čimbolj verno predstavitev na zaslonu terminala.

Izdelava ustrezne baze podatkov za 3D objekte je bistvena za zmožnosti prikaza raznovrstnih lastnosti površin objektov. Baza podatkov za senčenje in prikaz žičnega modela temelji na popisu mnogokotniške mreže. Osnovni popis objekta pa je parametričen z Bézierjevimi bikubičnimi zlepki. Topologija gradnikov baze je dvosmerna le tam, kjer je to potrebno.

V nalogi so obravnavani le osnovne lastnosti površin kot normala površine, barva, zrcalni odboj svetlobe in jakost ter barva svetlobnega izvora. Preizkušeni so inkrementalni algoritmi za senčenje (Polygon, Guarardov in Phongov), katere vključujejo tudi vsi znani komercialni programi za upodabljanje.

Senčenje je bilo do pred kratkim eksotična tema, ki je vključevala drago zaslonsko opremo, velike računalniške zmogljivosti in ustrezna programska orodja. Priljubljeni jezik F77 se pri razvoju grafov prej pokaže ovira kot pa prednost, zato je bil za izdelavo programa izbran jezik C.

Ključna informacijska dokumentacija

ŠD	Dn
UK	UDK
KG	računalništvo / programiranje / obdelava slike / grafika / software / 3D / senčenje / Bézier / C jezik

AV	KOS, Leon
SA	DUHOVNIK, Jože

KZ	61380 Cerknica, YU, Cesta 4. maja 11a
ZA	Univ. Ljubljana, Fakulteta za strojništvo
LI	1991

IN	SENČENJE 3-D OBJEKTA Z UPODOBITVIJO
TD	visokošolska diplomska naloga
OP	
IJ	Sl
JI	sl / en
AI	glej povzetek

Key words documentation

ND	Dn
UC	
CX	computer / programming / graphics / software / 3D / model visualisation / shading / Bézier / C programming language

AU	KOS, Leon
AA	DUHOVNIK, Jože
PL	61380 Cerknica, YU, Cesta 4. maja 11a
PB	Univ. Ljubljana, fac. of Mechanical Engineering

PY	1991

TI	3D OBJECT VISUALISATION WITH SHADING
DT	
NO	
AL	Sl
LA	sl / en

AB The paper deals with a problem of three-dimensional object visualisation with computer terminal. In the last decade, it became obvious that the use of straight line segments and planar polygons to approximate curved lines and surfaces has limited the sate-of-the-art in computer graphics. Bézier *patches* are the simplest way for parametric representation of complex objects.

As *benchmark* for incremental shading algorithms (Phong, Guarard) it was used well known teapot from Utah university. **C** was used as most promising and popular computer language today.

Vsebina

Kazalo slik

Pojasnila oznak

2D	dvodimenzionalno
3D	trodimenzionalno
C	Programski jezik, ki vsebuje dobre lastnosti različnih (*high level*) prevajalnikov in splošne ukaze makro zbirnika. Uporablja enostavne ukaze v ustrezni angleščini.
DC	Koordinatni sistem izhodne naprave (zaslona)
DECNET	Računalniški protokol za prenos podatkov po mreži
GKS	*Graphical Kernel System*, ISO standard za grafične aplikacije prirejen programskim jezikom ADA, C, FORTRAN in PACAL.
LUT	*Look-Up Table*, tabela barvnih indeksov zaslona
PHIGS	*Programmer's Hierarhical Interactive Graphics System*, ISO standard za grafične aplikacije
NDC	Normalizirane koordinate (po dogovoru enotski kvadrat v 2D ali enotska kocka vb 3D)
RCU	Računalniški center univerze v Ljubljani
TURBOC	Pravajalnik za jezik C na PC računalnikih
VAX	Večopravilni računalnik z operacijskim sistemom VMS
VAXC	Prevajalnik za C jezik na VAX-u
VWS	*Vax Workstation Software*, Programska oprema za delovne postaje
WC	Risarska ravnina v GKS koordinatnem sistemu ali svetovni koordinatni sistem v 3D

1 Uvod

Upodabljanje tridimenzionalnih objektov v računalniški grafiki postaja vsakdanja zahteva za vsak moderno usmerjeni konstrukcijski biro ne samo v strojništvu ampak tudi v drugih vejah industrije.

Večina algoritmov v 3D grafiki je stara manj kot 10 let, kar je povečalo zanimivost te diplomske naloge, saj so algoritmi v knjigah predstavljeni le z osnovami, kjer ni detajlne predstavitve problema. Velik del metod je predstavljen le v člankih tujih tiskovin. Programska izvedba algoritmov iz člankov in knjig lahko predstavlja težave, ki nastanejo zaradi nepopolnosti metode, slabe izbire programskega jezika, hardverskih in softverskih omejitev ciljnega računalnika, kakor tudi časovnih in umskih omejitev programerja.

Zahtevne tehnike upodabljanja (sledenje žarka, kompleksni modeli odboja) so časovno zahtevne, kar onemogoča hitro testiranje algoritmov. Ker je računalnška grafika izpeljala razne metode, od katerih je vsaka kompromis med vernostjo upodobitve in hitrostjo izračuna slike, ki jo dobimo s preslikavo objekta, popisanega v treh dimenzijah, na dvodimenzionalno površino (zaslon ali papir), sem se omejil na enostavne in s tem na časovno nezahtevne algoritme.

Doslej najbolj razširjeni grafični standard GKS je implementiran v vseh popularnih programskih jezikih kot so npr. FORTRAN, PASCAL in C. Fortran kot najstarejši programski jezik za numerične izračune se vse bolj umika novejšim jezikom (C^{++}, MODULA), ki imajo pri gradnji 3D baze podatkov najelegantnejši pristop k izdelavi strukture podatkov. Jezik, ki postaja v računalniški grafiki vse bolj popularen je prav C, katerega sem uporabil za izdelavo programa pri senčenju. C ima nekatere prednosti pred Pascalom, npr. zmogljiv predprocesor, modularna gradnja je standardna, določanje vidnosti spremenljivk in deklaracija npr. v vsakem bloku, možnost nadgradnje v objekte (C^{++} na UNIX sistemu) in velike knjižnice podprogramov. Koda, napisana v C-ju, je tako neodvisna od ciljnega računalnika. Za izris osenčene slike vsi algoritmi uporabljajo le najbolj osnovne gradnike, kot so določitev barve

točke na ekranu ali pa tudi višje gradnike (line, fillpolygon). Kompleksni algoritmi izračunavajo barvo vsake točke prikazane slike, tako da uporabljajo za prikaz le ukaz

putpixel(x,y,color_index);

Tako so metode senčenja neodvisne oziroma enostavno prilagodljive za različne krmilnike zaslonov. Najbolj popularne konfiguracije računalnikov: grafični terminali, povezani z glavnim večprocesnim računalnikov (host), lahko postanejo neuporabni, če za prenos podatkov uporabljajo le popularni serijski protokol RS232C.

Primer: Imamo grafični terminal (npr. RAMTEK serija 4000) z osmimi bitnimi ravninami (torej maksimalno 256 barvnih odtenkov), ločljivost zaslona je 1280x1024 točk. Za prenos uporabljamo RS232C protokol z maksimalno hitrostjo prenosa 19200BPS (Bits per Second), 8 bitne besede, 1 bit parnosti, 1 stop bit in 1 začetni bit (8+1+1+1=11bitov za besedo oz. byte). Če prenašamo podatke za vsako točko posebej, je to 1310720 bytov ali 14417920 bitov.

Čas potreben za prenos slike je:

(1280x1024x11)/19200BPS = minimalno 751 sekund ali 13 minut

Obstajajo tudi metode kompresije rasterskih podatkov RLE (Run Length Encoded) ali TIFF (Taged Image File Format), ki lahko količino podatkov zmanjšajo tudi za 90%, vendar je učinkovitost kompresije odvisna od kompleksnosti slike. Tako posamezni proizvajalci grafičnih terminalov poleg standardnega serijskega protokola vgrajujejo še paralelne povezave z glavnim računalnikom ali pa hitre serijske povezave, kot je ETHERNET ali celo povezave z optičnim kablom. Nakatere delovne postaje pa uporabljajo lastne procesorske zmogljivosti in tako nimajo težav s prenosom, kar je tudi najnovejši trend v računalništvu.

Učinkovitost je prav pri upodabljanju 3D objektov največja zahteva, tako so tudi različne metode kompromis med hitrostjo in vernostjo predstavitve. Cena za realističnost se tako vedno plača z računskim časom. Povečane zahteve v ločljivosti zaslona in števila barvnih odtenkov vodijo k zahtevam po večjih procesorskih zmogljivostih. Ker so računske operacije razmeroma enostavne, se je najbolj uveljavila arhitektura računalnika RISC (Reduced Instruction Set Computer), ki postopoma nadomešča konvencionalne procesorje s kompleksnim naborom ukazov. Trend nadaljnega razvoja računalnikov je usmerjn k večopravilnosti, vektorskim procesorjem in paralelnem procesiranju, kot pa k povečevanju

hitrosti, saj je splošno znano, da nič ne more biti hitrejša od svetlobe, vendar pa imajo današnji računalniki do magičnih hitrosti obdelave še veliko neizkoriščenih možnost.

Namen diplomske naloge je bil izbira najustreznejših tehnik upodabljanja, ki ne-bi bile računsko in časovno potratne, ter njihova izvedba za najbolj enostavne geometrijske oblike. V tekstu sem si bolj prizadeval opisati na način "kako je bilo narejeno" kot pa "kako naj bi bilo narejeno".

Glavna odločitev je bila na tem, kakšen naj bi bil popis baze podatkov 3D gradnikov. V praksi obstajata dva modela, ki se bistveno razlikujeta:

1. Površine so zgrajene iz mnogokotnikov. Točnost popisa površine je odvisna od gostote mreže vozlišč, ki gradi površino.
2. Površine so parametrično popisane. Točnost takega popisa je neodvisna od delitve mreže in tako lahko točnost prikaza vsakem primeru poljubno povečamo. Ta metoda lahko vedno generira bazo podatkov po prvi metodi. Baza podatkov pa je veliko bolj kompleksna.

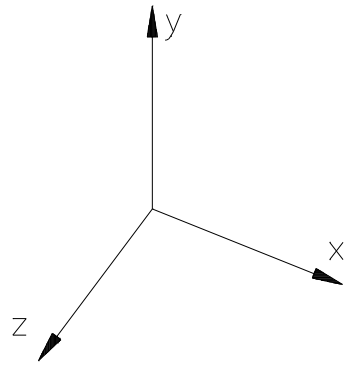
Čas in tudi vse praktične izvedbe današnjih[1] "post-procesorjev", so bili odločilni za izbiro za prve metode, druga metoda pa z današnjimi hitrostmi računalnikov predstavlja prej oviro kot pa prednost, saj so pridobitve v vernosti predstavitve scene zanemarljive [WATT90-139]. Za prvo metodo lahko druga vedno generira datoteko podatkov, obratno pa to praviloma ne velja. Čeprav je baza podatkov večine 3D modelirnikov parametrične narave, imajo le-ti konverter na mnogokotniško mrežo, katero lahko z različno zahtevnimi metodami nato osenčimo.

[1] EUCLID-IS firme Matra Datavision, ANVIL 5000

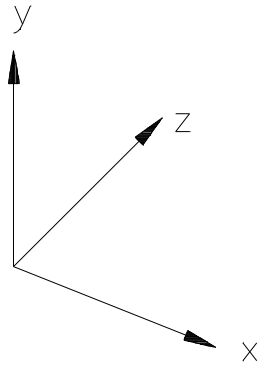
2 Osnove tridimenzionalne teorije

2.1 3D transformacije

Realni (svetovni) koordinatni sistem (WC), ki je po dogovoru desnosučen kartezijev koordinaten sistem, je osnova za popis objektov v treh dimenzijah. Tak koordinatni sistem krajše označimo z WC ("world coordinates"). V računalniški grafiki se je uveljavil tudi levosučni koordinatni sistem zaradi boljše analogije med 2D in 3D koordinatnega sistema.



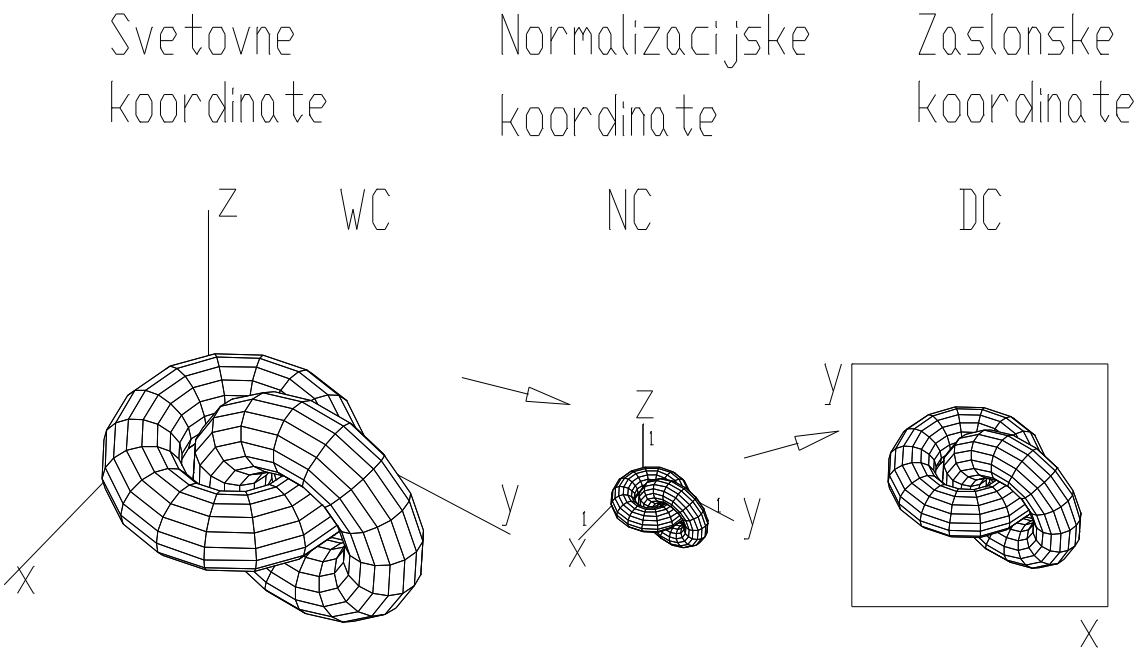
Slika 1 Desnosučni koordinatni sistem



Slika 2 Levosučni koordinatni sistem

Konverzija objektov iz treh dimenzij na dvodimenzionalni zaslon lahko poteka direktno s skaliranjem in translacijo v vidno polje. Tak način pa ni primeren zaradi slabe prenosljivosti aplikacij, saj imajo izhodne naprave praviloma različne ločljivosti in velikosti površine. Ravno zato se je v praksi kakor tudi v standarih uveljavila transformacija objektov preko vmesnega normalizacijskega prostora (NC), za katerega je znana velikost.

Po dogovoru je za 3D prostor to velikost kocke s stranicami velikosti 1 enote. Koordinate, ki jih iz WC pretvorimo v NC, lahko nato direktno transformiramo na 2D izhodno napravo. Prednost takega načina dela je tudi v tem, da imamo lahko na zaslonu več prikazov (oken) iste scene z razilčnimi pogledi. Lahko pa imamo več izhodnih naprav (terminal, risalnik, tiskalnik, kamera ...). V GKS standardu za 2D in 3D se vse te



Slika 3 Transformacija koordinatnih sistemov

transformacije avtomatsko izvajajo, tako uporabniku tega sistema ni potrebno skrbeti za preračunavanje koordinat, ampak samo nastavi posamezna okna oz. transformacijska načela, kot so vidno polje rezanje objektov in tekoče transformacije.

Včasih je objekt enostavneje popisati v lokalnih koordinatah in ga nato s transformacijo preslikati v globalni (svetovni) koordinatni sistem. Tak način popisa ima več prednosti, saj lahko objekt najprej npr. rotiramo v lokalnem koordinatnem sistemu, ga skaliramo, ali pa generiramo mrežo točk vrtenine, in šele po končanih operacijah postavimo na pravo mesto.

Transformacije pa morajo biti bijektivne (reverzibilne), da lahko naradimo kompozicijo preslikav. Navadne koordinate (x,y,z) in pripadajoče matrike dimenzije 3x3 nimajo teh lastnosti, zato se je v računalniški grafiki uveljavila kompozicija homogenih matrik transformacij, ki jih enostavno množimo med seboj, s tem pa sestavimo več transformacijskih matrik v eno.

Osnovne linearne transformacije so translacija, skaliranje in rotacija.

2.1.1 Translacija.

Točke se premaknejo z dodajanjem ustreznih vrednosti koordinatam točke.

Transformacija, ki translira točko (x,y,z) v novo točko, je

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{bmatrix},$$

kjer so Tx, Ty in Tz vrednosti, za katere se pomakne točka po osi *x*, osi *y* in osi *z*.

2.1.2 Skaliranje.

Točke se skalirajo z množenjem koordinat točke z vrednostmi skaliranja za posamezne točke.

Transformacija, ki skalira točko (x,y,z) v točko (x' y' z'), je

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1] \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

[ENQSTR86-204]
Za uniformno skaliranje (S=Sx=Sy=Sz) se matrika skaliranja poenostavi:

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/S \end{bmatrix}.$$

2.1.3 Rotacija okoli osi x

Transformacija, ki rotira točko (x,y,z) v novo točko (x',y',z') okoli osi *x* je:

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\vartheta & \sin\vartheta & 0 \\ 0 & -\sin\vartheta & \cos\vartheta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

kjer je ϑ kot rotacije v radianih okoli osi *x*.

2.1.4 Rotacija okoli osi y

Transformacija, ki rotira točko (x,y,z) v novo točko (x',y',z') okoli osi *y* je:

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1] \begin{bmatrix} 1 & \cos\vartheta & -\sin\vartheta & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \sin\vartheta & \cos\vartheta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

kjer je ϑ kot rotacije v radianih okoli osi *y*.

2.1.5 Rotacija okoli osi z

Transformacija, ki rotira točko (x,y,z) v novo točko (x',y',z') okoli osi *z*, je v levosučni notaciji koordinatnega sistema

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1] \begin{bmatrix} \cos\vartheta & -\sin\vartheta & 0 & 0 \\ \sin\vartheta & \cos\vartheta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

kjer je ϑ kot rotacije v radianih okoli osi *z*, za desnosučni sistem pa se menja le predznak pri obeh funkcijah sin ϑ.

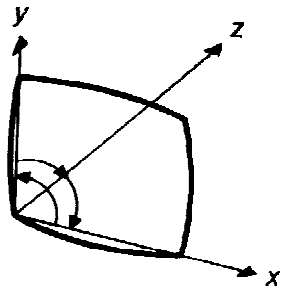
2.1.6 Koverzija iz homogenega v kartezijev koordinatni sistem

Svetovne koordinate točk izražene z (x,y,z,w) (homogene koordinate) se pretvorijo v 3D kartezijske koordinate (x',y',z') s sledečo transformacijo :

$$[x\ y\ z\ 1] = \left[\frac{x}{w}\ \frac{y}{w}\ \frac{z}{w}\ \frac{w}{w}\right] = [x'\ y'\ z'\ 1]$$

2.2 Alternativna formulacija transformacijskih matrik

Poleg predstavljenega seta matrik nekateri avtorji [FOLE90] raje formulirajo nabor homogenih transformacijskih matrik, katerih logika je bližje matematični predstavi vektorja v prostoru in njim ustreznih matrik. V tem načinu so matrike, v primerjavi s prejšnim, transponirane, vektorji pa so na desni strani homogene matrike.



Slika 4 Levoročni koordinatni sistem in analogija z zaslonom

Pri izbiri koordinatnega sistema sem se odločil za *desnoročni* koordinatni sistem, ker je to matematična konvencija, čeprav je udobneje v 3D grafiki uporabiti levoročni koordinatni sistem, ker daje naravnejšo predstavitev z koordinate, saj so večje z vrednosti dlje od opazovalca.

Homogene matrike po alternativni formulaciji so:

2.2.1 Matrika translacije

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Tako je $T(t_x, t_y, t_z)\cdot[x\ y\ z\ 1]^T = [x+t_x, y+t_y, z+t_z]^T$.

2.2.2 Matrika skaliranja,

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ki pa je enaka prvi formulaciji, le da je izračun vektorja

$$S(s_x, s_y, s_z)\cdot[x\ y\ z\ 1]^T = [x*s_x, y*s_y, z*s_z]^T.$$

2.2.3 Matrika rotacije okoli osi z v desnoročnem sistemu.

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

2.2.4 Matrika rotacije okoli osi x

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

2.2.5 Matrika rotacije okoli osi y

$$R\text{-}y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.2.6 Matrika pomika osi pravokotno na os z

Pri perspektivnih projekcijah se uporablja tudi matrika, ki premakne koordinate *x* in *y* točke pravokotno na os *z* ("shear matrix"), tako da točka $[x, y, z, 1]^T$ v prostoru dobi po transformaciji položaj $[x+z*sh_x, y+z*sh_y, z, 1]^T$.

$$SH_{xy}(sh_x, sh_y) = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.3 Kompozicija 3D transformacij

Homogene matrike imajo lastnost, da so reverzibilne in se jih vsled tega da sestaviti v eno samo matriko, ki predstavlja zmnožek vseh matrik, po funkciji pa je taka, kot če bi vektor postopoma množili z vsako transformacijsko matriko posebej.

Transformacija, ki naprej rotira točko okoli osi *x*, nato skalira in translira, je

$$[x' \; y' \; z' \; 1]^T = T(t_x,t_y,t_z)\cdot S(s_x,s_y,s_z)\cdot R_x(\vartheta)\cdot$$

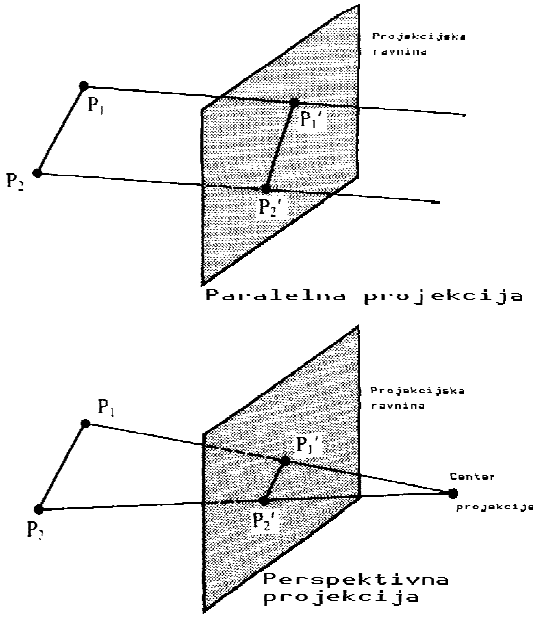
Matrike se množijo od leve proti desni[2]. Število kompozicij je poljubno, in je zato možno zmožiti matriko, ki izvaja zelo zahtevne transformacije v prostoru. Vektorji se na ta način množijo z matriko le enkrat, s tem pa se prihrani veliko računskega časa. Prav ta ugodnost se kasneje izkoristi za transformacijo objekta iz svetovnih (WC) ali normalizacijskih koordinat na ravnino izhodne naprave (DC) tako, da se zgradi sistem gledanja v eni od projekcij.

[2] Primer je podan alternativno formulacijo transformacijskih matrik

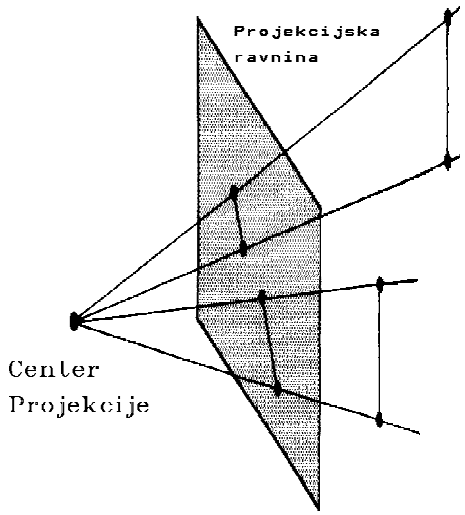
2.4 Projekcija tridimenzionalnih objektov na vidno ploskev

Projekcije v splošnem preslikajo točke iz *n* razsežnega koordinatnega sistema v koordinatni sistem z manj kot *n* koordinatami. Projekcija v 3D sistemu je definirana z ravnimi projekcijskimi žarki (projektorji). 2D ploskev je običajno ravnina in ne kakšna ukrivljena površina. Za preslikavo iz 3D v 2D moramo uporabiti eno od projekcij, ki jih delimo na dva osnovna razreda:

1. perspektivne
2. paralelne.



Slika 5 Projekcija dveh točk na ravnino z uporabo *paralelne* ali *perspektivne* projekcije



Slika 6 Pri perspektivni projekciji so bolj oddaljene linije manjše

Perspektivna projekcja je bolj priljubljena, ker vsebuje manjšanje objektov, ki se nahajajo na večji globini. Paralelne projekcije so hitrejše, saj v enostavnih primerih ni potrebna konverzija točk iz homogenih v kartezijeve koordinate.

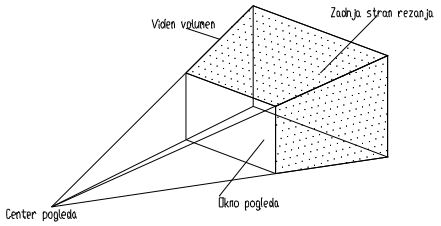
2.4.1 Paralelna projekcija [ENQSTR86-209]

$$[x'\ y'\ z'\ w] = [x\ y\ z\ 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & Z1 & 1 \end{bmatrix}$$

Ta projekcija preslika vse točke v prostoru na ravnino z=Z1. V praksi se za hitre projekcije sploh ne uporablja ta transformacija, ampak koordinate vektorjev *x* in *y*.

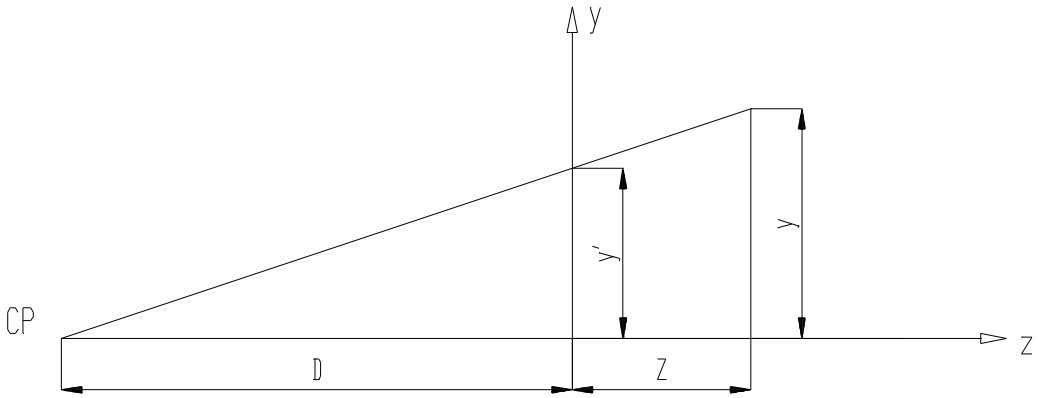
2.4.2 Perspektivna projekcija

Slika predstavlja perspektivno projekcijo. Perspektivni pogled je v smeri osi z v središču okna. Preden se izvede perspektivno projekcijo se reže vse črte, ki se nahajajo pred ekranom (Z<0) in črte, ki segajo preko ravnine Z=F. Nato se režejo vse črte, ki segajo izven piramide z robovi na oknu in ravnine F.



Slika 7 Perspektivna projekcija

Razdalja očesa definira oddaljenost točke gledanja (to je položaj očesa) od okna (Z=0). Ta



Slika 8 Perspektivna projekcija

razdalja določa globino in s tem efekt perspektive. Bližje kot je položaj očesa (centra projekcije) zaslonu, večji je efekt perspektive.

Iz slike se lahko iz podobnosti trikotnikov izračuna položaj točk, podanih v 3D, na ekranu oz. oknu gledanja.

$$\frac{y'}{D} = \frac{y}{D+Z}$$

Sledi, da je točka na oknu

$$y' = y\frac{D}{D+Z} = \frac{y}{1+\frac{Z}{D}}$$

in analogno

$$x' = x\frac{D}{D+Z} = \frac{x}{1+\frac{Z}{D}}$$

Iz zgornjih enačb lahko sklepamo, da je matrika perspektive[3] sledeča:

$$[x'\ y'\ z'\ 1] = [x\ y\ z\ 1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/D \\ 0 & 0 & 0 & 1 \end{bmatrix} = [x\ y\ z\ 1]\ P_{PZ}$$

Matrika P_{PZ} pa je produkt dveh matrik [ENCSTR86-211]

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

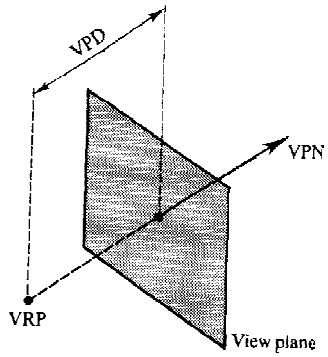
$$P_{PZ} = T_{PZ} \,.\, P_Z\ ,$$

od katerih je T_{PZ} matrika, ki opravlja centralno projekcijo in matrika P_Z, ki projicira vse točke na X-Y ravnino (Z=0). Projekcija je torej dvostopenjski proces sestavljen iz projekcije in transformacije vseh točk na ravnino Z=0. Transformacija, ki preslika vse točke na ravnino X-Y, ni reverzibilna, in se zato pri množenju uporablja samo matrika T_{PZ}.

[3]Podana je formulacija, ki je običajna v jeziku GKS. Alternativna formulacija ima matriki transponirano.

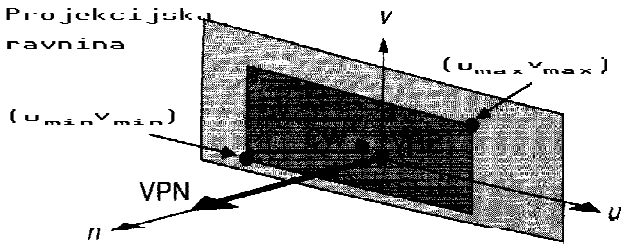
2.4.3 Operacija 3D gledanja

Sistem 3D gledanja je množica parametrov, ki opisujejo skupek operacij za transformacijo iz svetovnega koordinatnega sistema v koordinatni sistem 2D izhodne naprave. Tak set parametrov vključuje tudi eno od projekcij iz 3D v 2D. Grafična standarda PHIGS in GKS-3D imata sistem gledanja že vgrajen, tako da lahko programer uporabi že obstoječe podprograme za transformacijo v 2D, ali pa napiše svoje rutine za operacijo gledanja, če ni na voljo nobeden od grafičnih sistemov, ki omogočajo operacijo gledanja.



Slika 9 Projekcijska ravnina

Projekcijska ravnina (*projection plane, view plane*) je virtalni zaslon ustreznih dimenzij v svetovnem **WC** koordinatnem sistemu. Ta ravnina je definirana s točko **VPN** (*View Plane Normal*) na tej ravnini in smernim vektorjem **VPN** (*View Projection Normal*). PHIGS ima poleg omenjenih dveh parametrov še en parameter **VPD** (*View Projection Distance*), ki podaja oddaljenost projekcijske ravnine v smeri normale VPN. Projekcijska ravnina se lahko nahaja kjerkoli v svetovnem (WC) koordinatnem sistemu, lahko je pred objektom, za njim, lahko pa gre tudi skozi objekt.



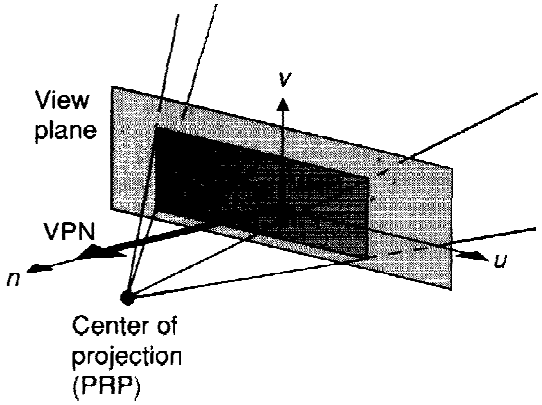
Slika 10 Okno na projekcijski ravnini

Na tako podani ravnini je potrebno definirati še okno, ki naj bi bi predstavljalo 2D zaslon. Definiranje okna je pomembno za samo predstavitev, saj se ravnina sicer razlega na vse strani. Na projekcijski ravnini se da definirati ortogonalni koordinatni osi in minimalne

ter maksimalne koordinate okna. Koordinatni osi *u* in *v* na projekcijski ravnini sta definirani s smernim vektorjem **VUP** (*View Up Vector*), ki je definiran v svetovnem koordinatnem sistemu (WC). Vektor VUP definira smer osi *v* na projekcijski ravnini. Smer osi *u* je definirana tako, da skupaj z vektorjem VPN in osjo *u* tvori desnosučni ortogonalni sistem, ki ga imenujemo koordinatni sistem gledanja **VRC** (*View Reference Coordinate system*).

Ko imamo VRC koordinatni sistem definiran, lahko na njemu definiramo minimalne in maksimalne koordinate okna (u_{min}, v_{min}, u_{max}, v_{max}). Tako okno lahko leži kjerkoli na projekcijski ravnini in ne samo v sredini koordinatnega sistema. Točka **CW** (*Center of Window*), ki leži na tej ravnini je definirana kot središče okna.

Center projekcije in smer projekcije sta definirani s točko **PRP** (*Projection Reference Point*), ki je za razliko od ostalih vektorjev definirana v VRC koordinatnem sistemu. Tako se relativni položaj centra projekcije glede na os *u* in *v* oz. definirano okno, ne spremeni s spremembo orientacije okna z vektorjem VUP ali položaja ravnine s spremembo VRP ali VPN vektorja.

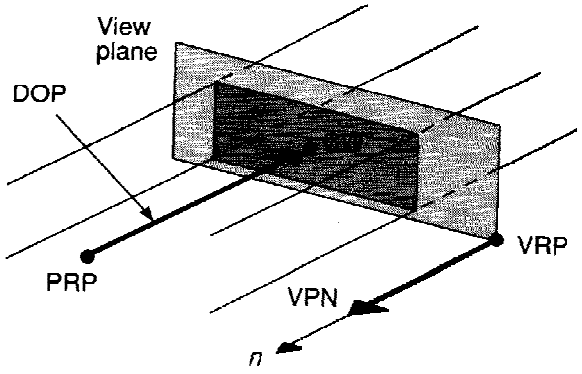


Slika 11 Konični polprostor perspektivne projekcije

Ta način formulacije torej ne spremeni tipa projekcije, kar se pokaže kot ugodnost pri posebnih tipih projekcij (npr. kabinetna in kavalirska).

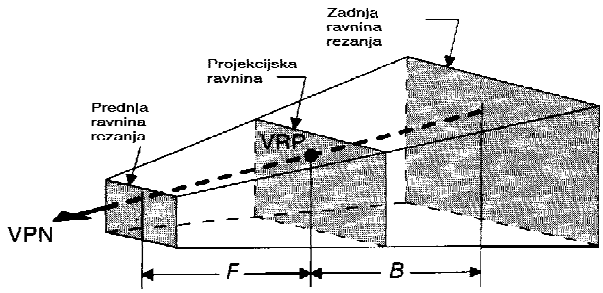
Volumen gledanja (*view volume*) je definiran s točko PRP in projektorskimi premicami. Pri paralelnih projekcijah ima volumen gledanja obliko neskončnega paralelopipeda, perspektivna projekcija pa ima polprostor v obliki neskončne piramide.

Kljub temu da so taki volumni gledanja omejeni s projektorskimi premicami, se pokaže potreba, da se viden volumen popolnoma omeji z dodatnima ravninama, od katerih se ena nahaja pred točko VRP in druga za njo v smeri vektorja VPN. Omejevanje volumna je lahko zelo koristno takrat, ko hočemo izločiti odvečne objekte ali dele njih iz operacije gledanja.



Slika 12 Neskončen paralelopiped paralelne projecije

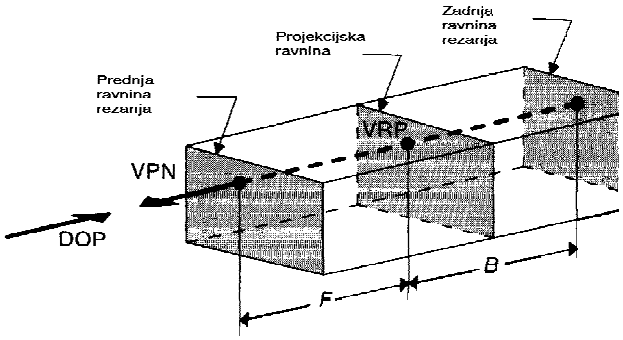
Pri perspektivni projekciji so to npr. objekti, ki so zelo oddaljeni od ravnine gledanja in jih na ta način enostavno izločimo iz scene. Če se žični model oddaljenega (beri majhnega) objekta riše na risalniku, lahko majhni premiki peresa risalnika naredijo luknjo v papir, pri vektorskih zaslonih pa premočno osvetljen delček zaslona.



Slika 13 Omejejen volumen gledanja pri perspektivni projekciji

Naslednji primer je, pri paralelni projekciji, ko hočemo prerezati objekt in nato prikazati objekt znotraj, pri čemer moramo rezati vse dele objekta, ki se nahajajo pred prednjo ravnino rezanja.

Izrezovanje (*clipping*) je standardni del vseh splošnih sistemov gledanja.



Slika 14 Izrezan volumen pri paralelni projekciji

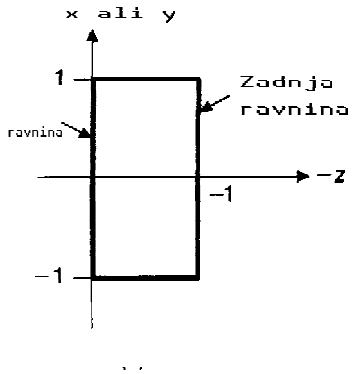
Poleg predstavljenega načina imamo lahko tudi izpeljane sisteme gledanja, ki so bolj enostavni za razumevanje. Eden takih je sistem kamere pri katerem se podajo položaj kamere, točka cilja oz. fokus, ter višina in širina okna kamere. Projekcija kamere je perspektivna, okno pa je simetrično (CW je v središču VRC koordinatnega sistema).

2.4.4 Praktična izvedba projekcij

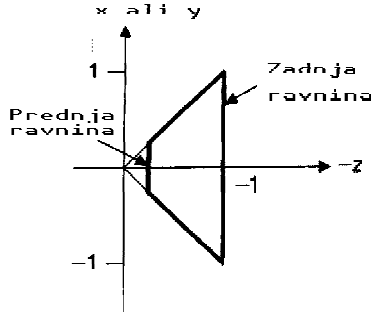
Za transformacijo 3D prostora na 2D ravnino je potrebno uporabiti eno od projekcij, s tem da ustvarimo čimbolj univerzalen sistem gledanja. Spoznali smo že, da je priporočljivo uporabiti t.i. normaliziran koordinatni sistem (NDC), iz katerega lahko enostavno transformiramo objekt na različne izhodne naprave. Normalno je to enotska kocka ali druga dogovorjena velikost in oblika normalizacijskega prostora.

Tako definiramo npr. normiran prostor s šestimi ravninami:

$$z=0,\ z=1,\ x=-1,\ x=1,\ y=-1,\ y=1\ .$$



Slika 15 Normiran volumen paralelne projekcije



Slika 16 Normiran prostor perspektivne projekcije

Definirane volumne lahko enostavno obrezujemo z znanimi mejami normiranih prostorov. Enostavna oblika kocke pri paralelnem normiranem prostoru ne predstavlja velikih težav za izrezovanje. Večji problem se pojavi pri perspektivni projekciji zaradi specifičnosti normiranega prostora, vendar se da tudi tu pomagati z izrezovanjem v homogenih koordinatah.

2.4.4.1 Paralelna projekcija

Potrebno je izračunati paralelno normalizacijsko projekcijo $\mathbf{N}_{par}$, ki transformira swtovne koordinate v normirane koordinate (WC -> NC).

Postopek transformacije v vidni volumen je sledeč: [FOLE90-261]

1. Translacija VRP v izhodišče svetovnega koordinatnega sistema (WC)
2. Vrtenje VRC koordinatnega sistema, tako da os *n* (VPN) postane z os, os *u* postane os *x* in *v* postane os *y*.
3. Premaknemo koordinati *x* in *y* tako vektorja DOP, da postane smer projekcije (DOP) enaka osi *z*.
4. Translacija in skaliranje v normirano velikost volumna.

Prvi korak je tranlsacija v izhodišče T(-PRP).

Drugi korak je rotacija v VRC koordinatnega sistema z matriko

$$R = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} & 0 \\ r_{1y} & r_{2y} & r_{3y} & 0 \\ r_{1z} & r_{2z} & r_{3z} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

pri tem je podmatrika rotacije vektorja VPN v os *z*

$$R_z = \frac{VPN}{|VPN|} .$$

Os *u* je pravokotna na VUP in VPN, in je zato vektorski produkt teh dveh vektorjev rotacija VRC sistema okoli osi *x*

$$R_x = \frac{VUP \times R_z}{|VUP \times R_z|} .$$

Podobno je tudi os *v*, ki je pravokotna na osi R_x in R_z, rotiran v os *y*

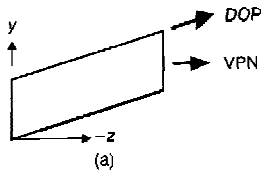
$$R_y = R_z \times R_x \ .$$

Tretji korak je premik volumna pravokotno na os *z* z matriko

$$SH_{par} = SH_{xy}(shx_{par}, shy_{par}) = \begin{bmatrix} 1 & 0 & shx_{par} & 0 \\ 0 & 1 & shy_{par} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

pri čemer je smer projekcije (DOP) merilo za linearen premik pravokotno v smeri osi *z*.

Smer projekcije se izračuna z odštevanjem središča okna (CW), ki leži zdaj na ravnini *z*=0 od projekcijske referenčne točke (PRP).



Slika 17 Premik volumna pravokotno *z* osi

$$DOP = CW - PRP = \left[\frac{u_{\max} + u_{\min}}{2} \quad \frac{v_{\max} + v_{\min}}{2} \quad 0 \quad 1 \right]^T - \left[prp_u \ prp_v \ prp_n \ 1 \right]^T$$

Intenziteta premika pravokotno na os *z* se izračuna z

$$shx_{par} = -\frac{dop_x}{dop_z} \ , \quad shy_{par} = -\frac{dop_y}{dop_z} \ .$$

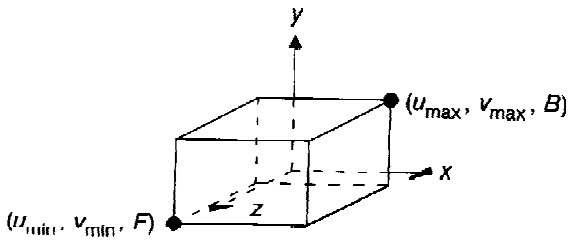
Pri ortografskih projekcijah (to ja takrat, ko je smer projekcije (DOP) enaka normali projekcijske ravnine VPN), je $dop_x = dop_y = 0$.

Po transformacijah ima volumen sledeče meje

$$u_{min} \le x \le u_{max}, \quad v_{min} \le y \le v_{max}, \quad B_{min} \le z \le F_{max}$$

in je tako potrebno le še transformirati viden volumen v ustrezen normaliziran prostor.

Tako lahko npr. uporabimo 2 x 2 x 1 velik normalizacijski prostor. Zadnji, četrti korak, je torej translacija centra okna (CW) v koordinatno izhodišče in inverzno skaliranje, da dobimo ustrezno velik NC prostor.



Slika 18 Viden volumen po transformaciskih korakih 1 do 3

Transformaciji sta:

$$T_{par} = T\left(-\frac{u_{max}+u_{min}}{2}, -\frac{v_{max}+v_{min}}{2}, -F\right),$$

$$S_{par} = S\left(\frac{2}{v_{max}-u_{min}}, \frac{2}{v_{max}-v_{min}}, \frac{1}{F-B}\right)$$

V jeziku GKS sta slednji transformaciji različni, saj je tam dogovorjen normalizacijski prostor v obliki kocke z ravninami

$$0 \le x \le 1,\ 0 \le y \le 1,\ 0 \le z \le 1.$$

Tako je v GKS-u translacija $T_{par} = T(-u_{min}, -v_{min}, -F)$ in rotacija

$$S_{par} = S\left(\frac{1}{u_{max}-u_{min}}, \frac{1}{v_{max}-v_{min}}, \frac{1}{F-B}\right)$$

Če se v sistemu gledanja ne uporablja izrezovanje, je potrebno vseeno zagotoviti, da so maksimalne vrednosti večje od maksimalnih. Če se npr. na zagotovi, da je $F > B$ pride v matriki S_{par} do negativnega tretjega diagonalnega elementa, kar povzroči nepredvideno zrcaljenje volumna preko osi z.

Po zmnožku vseh transformacijskih matrik dobimo matrika paralelne projekcije, ki je

$$N_{par} = S_{par} \cdot T_{par} \cdot SH_{par} \cdot R \cdot T(-\text{VRP}).$$

2.4.4.2 Perspektivna projekcija

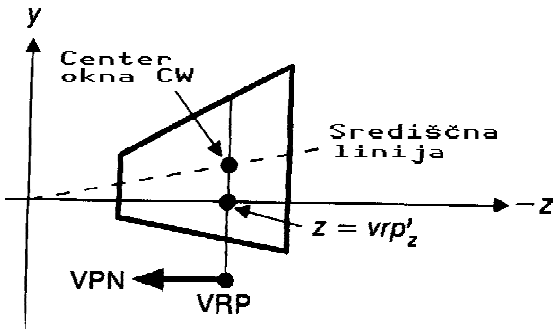
Izvedba perspektivne projekcije je podobna praralelni projekciji, le da moramo končni volumen ustrezno skalirati in tako dobiti volumen v obliki prisekane piramide.
Serija transformacij za perspektivno projekcijo je sledeča: [FOLE90-268]

1. Translacija VRP v koordintatno izhodišče
2. Vrtenje VRC koordinatnega sistema, tako da os *n* (VPN) postane os *z*, os *u* postane os *x* in os *v* postane os *y*.
3. Premik centra projekcije (COP) podanega z PRP točko v koordinatno izhodišče.
4. Premaknemo koordinati *x* in *y* točke tako, da postaneta smer projekcije (DOP) enaka osi *z*.
5. Skaliranje v normirano velikost volumna oblike prisekane piramide

Iz navedenega postopka je razvidno, da sta prva dva koraka enaka v pararalelni kakor tudi perspektivni projekciji. V PHIGS ali GKS-3D jezikih sta prva dva koraka združena v en podprogram, ki izračuna matriko transformacije svetovnih koordinat (WC) v VRC koordinatni sistem (*view orientation matrix*).

Tretji korak je enostavna translacija PRP točke v izhodišče *T*(-PRP), ker je le-ta točka definirana v VRC koordinatnem sistemu.

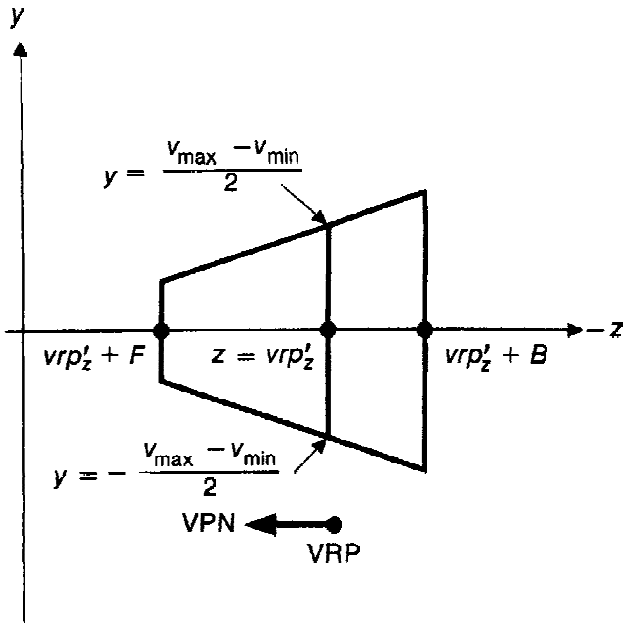
Četrti korak je podoben tretjemu koraku pri paralelni projekciji. Potrebno je premakniti center okna in z njim vse točke v sredino osi *z* tako, da se ne izgubijo razmerja med razdaljami. Center projekcije (COP) ostane v koordinatnem izhodišču. Po četrtem koraku sledi še skaliranje prostora v ustrezen normalizacijski prostor npr. velikosti 2 x 2 x 1.



Slika 19 Prerez volumna po transformacijah 1 do 3

Pri tem skaliranju je potrebno paziti, da zadnja stran rezanja ne presega maksimalnih podanih velikosti volumna. Izračunati je potrebno faktor skaliranja za os *x* in *y* z uporabo enačbe premice

$$y = kz + n = \frac{v_{max} - v_{min}}{2vrp'_z}(vrp'_z + B) + 0$$



Slika 20 Volumen persp. proj. pred normalizacijskim skaliranjem

Smerni koeficient *k* izračunamo iz točke

$$(z = vrp'_z\ ,\ y = (v_{max} - v_{min})/2).$$

Za koordinato *z* vzamemo zadnjo stran rezanja, ki se trenutno nahaja na poziciji

$$z = vrp'_z + B.$$

Podoben je izračun skaliranja *x* koordinate. kar nam da matriko skaliranja

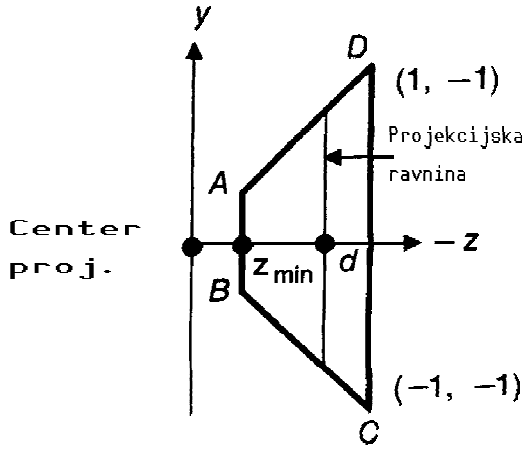
V celoti dobimo po zmnožku matriko perspektivne projekcije

$$N_{per} = S_{per} \cdot SH_{par} \cdot T(\text{-PRP}) \cdot R \cdot T(\text{-VRP}).$$

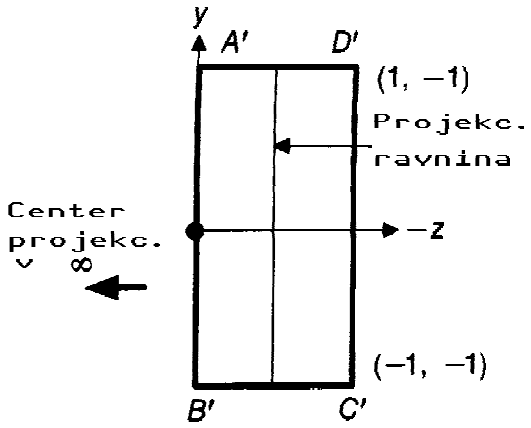
2.4.4.3 Rezanje v homogenih koordinatah [FOLE90-275]

Čeprav je možno izrezovanje objektov v 3D kartezijevih koordinatah po Liang-Barsky algoritmu [FOLE90-274], je hitreje in enostavneje uporabiti rezanje v homogenih koordinatah s pomočjo ene same matrike

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{min}} & \frac{-z_{min}}{1+z_{min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \quad z_{min} \neq -1$$



Slika 21 Normaliziran vidni volumen pred rezanjem z matriko *M*



Slika 22 NC prostor po množenju z matriki *M*

Matriko *M* množimo z perspektivno projekcijo in tako dobimo matriko N'_{per}, ki reže koordinate v homogenem prostoru

$$N'_{per} = M \cdot S_{per} \cdot SH_{par} \cdot T(\text{-PRP}) \cdot R \cdot T(\text{-VRP}).$$

Če koordinata točke v homogenih koordinatah po množenju dobi vrednost W < 0, potem izrezujemo točko glede na volumen

$$-W \le X \le W,\ -W \le Y \le W,\ -W \le Z \le 0,$$

ob vrednosti W > 0 pa izrezujemo glede na volumen

$$-W \ge X \quad W,\ -W \ge Y \ge W,\ -W \ge Z \ge 0.$$

2.4.4.4 Transformacija NC prostora na 2D ravnino izhodne naprave

Normaliziran prostor z mejami $0 \le x \le 1$, $0 \le y \le 1$, $0 \le z \le 1$ enostavno povečamo v koordinatni prostor izhodne naprave (DC), s tem da izračunamo velikost okna podanega s koordinatami okna v DC vrednostih.

Tako je skaliranje po osi *x* enako $(x_{v,max} - x_{v,min})/2$. Številka 2 je tu zaradi tega, ker je normalizacijski prostor velik dve enoti in ne eno kot npr. v PHIGS ali GKS-3D. Podobno je skaliranje po osi *y* le da tu podamo velikost okna $y_v=(y_{v,max} - y_{v,min})/2$.

Matrika skaliranja na velikost okna je

$$S\left(\frac{x_{v,\max}-x_{v,\min}}{2}, \frac{y_{v,\max}-y_{v,\min}}{2}, \frac{z_{v,\max}-z_{v,\min}}{1}\right).$$

Po skaliranju je potrebno premakniti okno na pravo mesto DC ravnine s translacijo minimalnih koordinat okna $T(x_{v,min}, y_{v,min}, z_{v,min})$.

2.4.4.5 Povzetek izvedbe projekcij

Postopek transformacije objektov iz 3D svetovnih koordinat na 2D polskev izhodne naprave je sledeč:

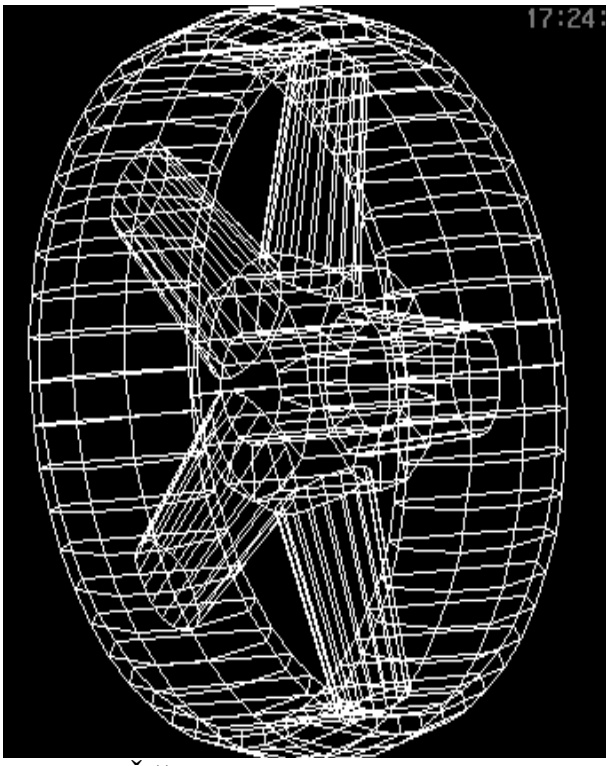
1. Razširimo 3D koordinate v homogene koordinate, s tem za dodamo še eno koordinato $w = 1$.
2. Uporabimo eno od normalizacijskih transformacij[4] N_{par} ali N'_{per}.
3. Po potrebi režemo v homogenem prostoru.
4. Skaliramo in transliramo na vidno okno (DC).
5. Točko v homogenih koordinatah delimo z *w*, da dobimo zaslonske koordinate pri perspektivni projekciji.

[4] Perspektivna normalizacijska matrika mora vsebovati matriko izrezovanja *M*.

3 Žični modeli

Najenostavnejšo predstavitev 3D objektov omogočajo žični modeli. To je množica točk povezanih z ravnimi črtami, tako da tvorijo obris površin objekta. V primerjavi z osenčenimi modeli se žični model zelo hitro izračuna in prikaže. Zato se ta način prikaza uporablja kot kontrola pred samim senčenjem in pa za samo konstruiranje, kjer igra hitrost odziva pomembno vlogo.

3D računalniške animacije uporabljajo tak model za kontrolo gibanja scene.



Slika 23 Žični model

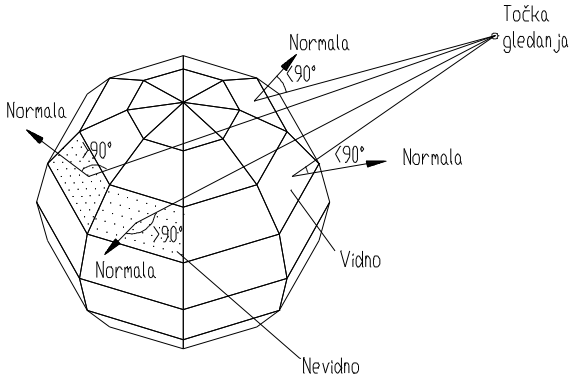
Gradnja objektov se sestoji iz množice mnogokotnikov, katere je priporočljivo združevati tako, da posamezna množica mnogokotnikov tvori površino. Čeprav to ni nujno, je pa to ugodno, ker se lahko taka struktura žičnega modela kasneje enostavneje aplicira za zapis osenčenega modela, ki ima mnogo bolj kompleksno strukturo. Slaba stran predstavitve objektov z žičnim modelom je ta, da se lahko izgubi orientacija položaja za simetrične objekte, in pa tudi ob množici črt postane slika nepregledna.

Da bi izboljšali predstavitev žičnih modelov, so se v praksi uveljavili različni algoritmi odstranjevanja nevidnih (skritih) robov in ploskev ([GUID88-347]). Tak način prikaza se v novejših časih opušča zaradi večjih zmogljivosti aparaturne opreme. Starejši prikazovalniki so bili po pravilu vektorski, in je zato bila medoda skrivanja nevidnih prevladujoča. Ne gre pa zanemariti uporabnost metode, če se kot izhodna naprava uporablja risalnik, ki je v osnovi vektorska naprava.

4 Eliminacija zadnje strani objekta [WATT90-36]

Na normalnih 3D objektih je običajno vidno le približno 50% vseh površin, iz katerih so sestavljeni. Tako je možno nekatere mnogokotnike v površini enostavno eliminirati, že če izračunamo skalarni produkt normale mnogokotnika in vektorja točke gledanja. Površina je vidna, če je kot med tema dvema vektorjema manjši od 90°. To je takrat, ko je

$$\text{vidnost} = N \cdot V > 0$$

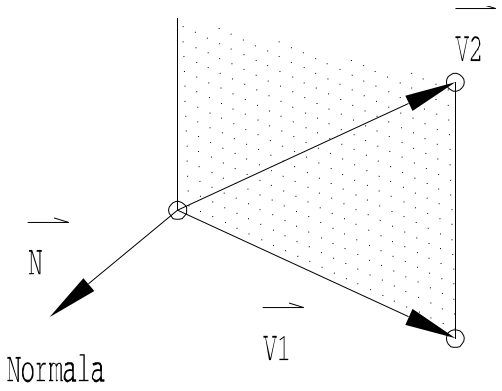


Slika 24 Eliminacija zadnje strani objekta

Normala ploskve se izračuna z vektorskim produktom dveh vektorjev, ki ležita na površini in tvorita desnosučni koordinatni sistem. Ta dva vektorja sta sestavljena iz treh vozlišč površine.

$$\vec{N} = \vec{V1} \times \vec{V2}$$

Eliminacija zadnje strani objekta (angl. culling) predstavlja enostavno operacijo pri eliminaciji odvečnih ploskev in se običajno uporablja kot predopreracija skrivanja nevidnih robov pri žičnih modelih ali pred senčenjem.



Slika 25 Normala ploskvice

5 Parametrična predstavitev površin

Uporaba gladkih krivulj in površin v računalniški grafiki postaja vse bolj pravilo, ki klasificira kvaliteto grafične aplikacije, še posebno v računalniško podprtem konstruiranju (CAD). Pomembnost parametrične predstavitve krivulj se kaže tudi v drugih branžah industrije (tiskarstvo: gladke oblike črk, alimacija: gladko gibanje kamere, itd.).

Parametrična predstavitev površin ima več prednosti pred predstavitvijo površin s končnim številom točk mnogokotniške mreže v 3D prostoru (x, y, z).

Uporabnik pri modeliranju novih oblik z računlnikom zahteva, da z majhnim številom parametrov popiše površino in da bi bilo to število dovolj veliko, da bi tak popis zajel tudi zahtevnejše oblike površin. Način popisa ne sme biti računalniško zahteven, saj se lahko kaj kmalu zapletemo v časovno zanko interaktivnega dela s programskim vmesnikom. Pomemben dejavnik je tudi spomin računalnika, katerega poraba naj bi bila čim manjša.

Predstavitev objekta z mnogokotniškimi mrežami se je najbolj uveljavila za prikaz na zaslonu, popis objekta pa je lahko raznovrsten:

- mnogokotniki popisani s seznamom vozlišč
- mnogokotniki s seznamom kazalcev na površine
- mnogokotniki s seznamom robov
- seznam parametričnih površin popisane na različne načine

Običajen pristop k parametrični predstavitvi se začne s popisom 3D krivuljami, ki jih nato posplošimo na 3D površine.

5.1 Bézierjevi zlepki

Ena najbolj popularnih parametričnih formulacij površin je Bézierjev zlepek katerega je uvedel leta že 1960 *Pierre Bézier*, vendar je bilo njegovo delo predstavljeno šele leta 1975[5].

Enostaven primer parametrične 3D krivulje je popis s kubičnimi polinomi

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x ,$$
$$y(u) = a_y u^3 + b_y u^2 + c_y u + d_y ,$$
$$z(u) = a_z u^3 + b_z u^2 + c_z u + d_z , \qquad 0 \le u \le 1.$$

Parameter u se običajno omeji na interval [0,1], ne da bi se pri tem izgubila splošnost uporabe. S polinomi višijh redov lahko popišemo zelo zahtevne krivulje, vendar le-te zahtevajo večje število koeficientov in imajo lahko nezaželene oscilacije na sami krivulji.

Z bolj jedrnato matrično formulacijo $U = [u^3\ u^2\ u\ 1]$ in

$$B = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & dz \end{bmatrix},$$

dobimo

$$P(u) = [x(u)\ y(u)\ z(u)] = B \cdot U .$$

Posebna oblika parametrične predstavitve krivulje je Bézierjeva krivulja, ki je izpeljana iz Bernsteinovih polinomov [GUID88-264]:

$$(1-u)^3 ,$$
$$3u(1-u)^2 ,$$

[5] [WATT90] st. 115

$3u^2(1-u)$,

u^3 ,

ki vključujejo množico štirih *kontrolnih točk*, iz katerih je izpeljana krivulja

$$P(u) = [(1-u)^3 \quad 3u(1-u)^2 \quad 3u^2(1-u) \quad u^3] \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}.$$

Polinomsko enačbo lahko zapišemo še kot

$$P(u) = \boldsymbol{UBP} = [u3 \quad u^2 \quad u \quad 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix},$$

pri čemer je **B** Bézierjeva transformacijska matrika.

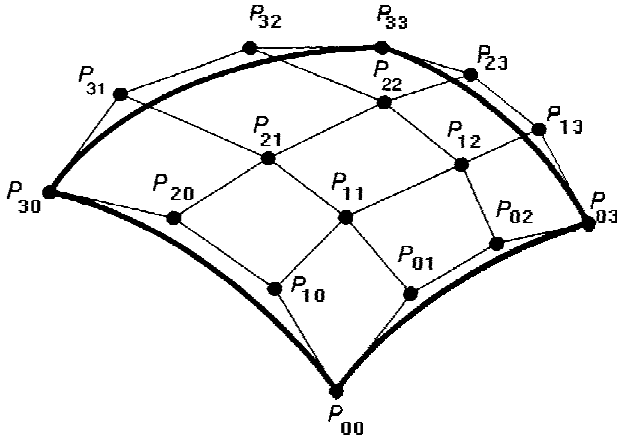
Obravnavanje parametričnih kubičnih krivulj se enostavno posploši na bi-parametrične kubične površine (krpe oz. zlepke), ki jih v računalniški grafiki sestavljamo v večje površine. Taka površina je podana z dvema parametroma *u* in *v*, ki sta ravno tako zaradi posplošitve omejena na interval [0,1].

x = x(u,v) ,

y = y(u,v) ,

z = z(u,v)

ali



Slika 26 Kontrolne točke in zlepek

$$P = [\, x(u,v)\ y(u,v)\ z(u,v)] \,.$$

Kubična Bézierjeva krpa pa je podana z

$$P(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} P_{i,j}\, B_{i,3}(u)\, B_{j,3}(v) \ ,$$

kar je kartezijski produkt dveh bikubičnih Bézierjevih krivulj. Tako imamo tu 16 kontrolnih točk (namesto 4 pri krivulji), od katerih ima vsaka točka svoj vektor (x y z) v 3D prostoru. Matrična formulacija ploskve je

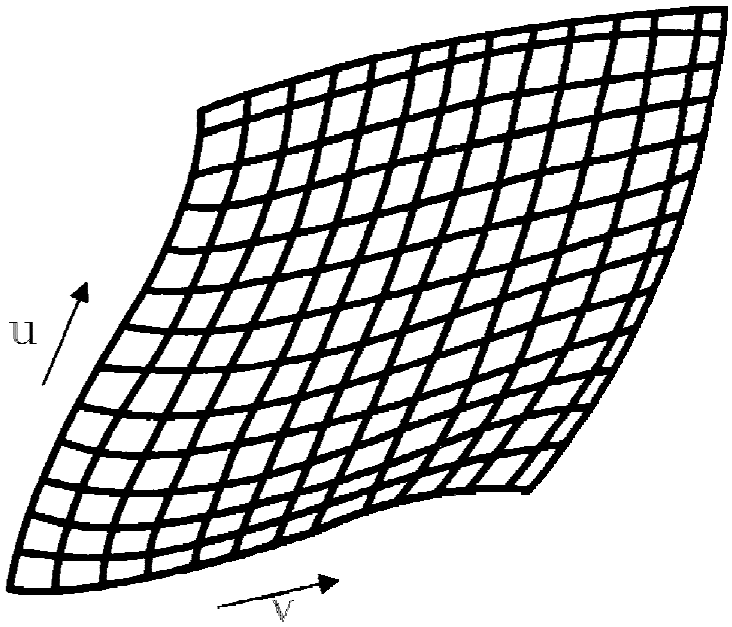
$$P(u,v) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{B}\boldsymbol{P}\boldsymbol{B}^T \end{bmatrix} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix},$$

kjer je

$$\boldsymbol{B} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix},$$

$$\boldsymbol{P} = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix}.$$

Če enemu od parametrov priredimo konstantno vrednost, dobimo enoparametrično družino krivulj. Dvoje parametrov predstavlja tudi dvoje družin enoparametričnih krivulj



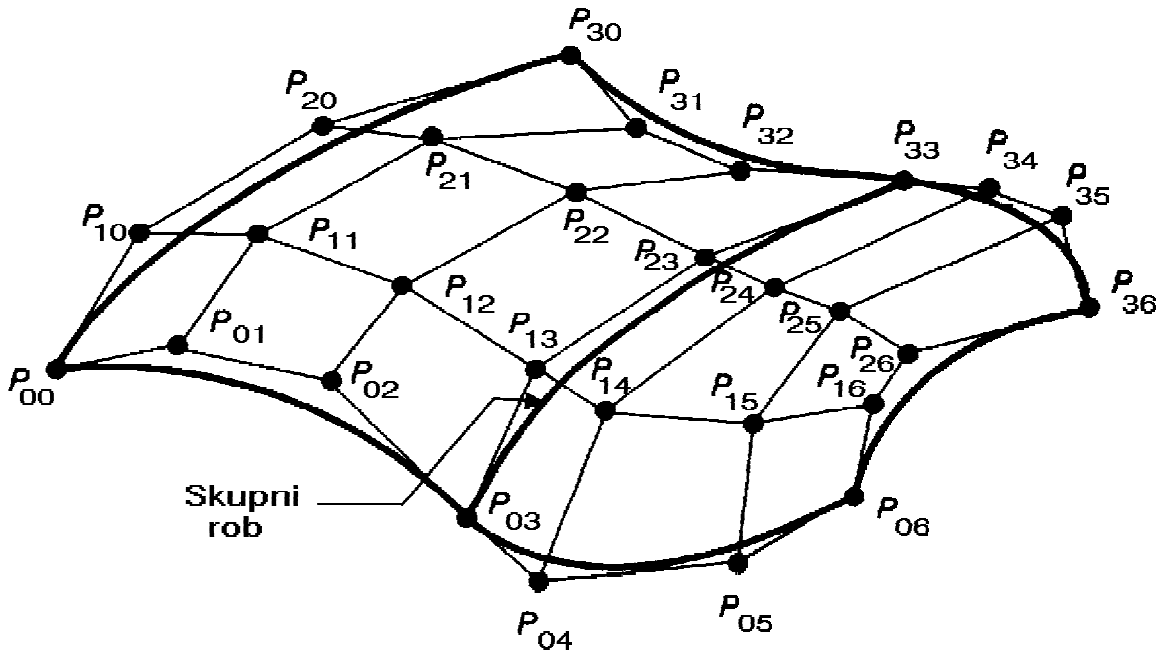
Slika 27 Zlepek z družinami krivulj *u* in *v*

P(u=konst, v) in **P**(u, v=konst).

5.2 Sestavljanje Bézierjevih krp

Če hočemo imeti kontinuiteto prvega reda preko dveh zlepkov, moramo imeti enake tangentne vektorje na robovih zlepkov. Pri Bézierjevih zlepkih je ohranjanje kontinuitete posebno lahko, saj je potrebno zagotoviti le, da so tri kontrolne točke dveh zlepkov kolinearne na preko robu, na katerem se stikata dva zlepka. Kljub enostavnosti in veliki prilagodljivosti na robovih se še vedno lahko pojavijo problemi pri sestavljanju in ena od rešiev je uporaba zlepkov višjega reda od kubičnih.

Tako so se razvile tudi druge oblike zlepkov, kot so B-zlepki, NURBS ("non uniform rational B-splines"), β-zlepki itd.



Slika 28 Sestavljanje Bézierjevih zlepkov

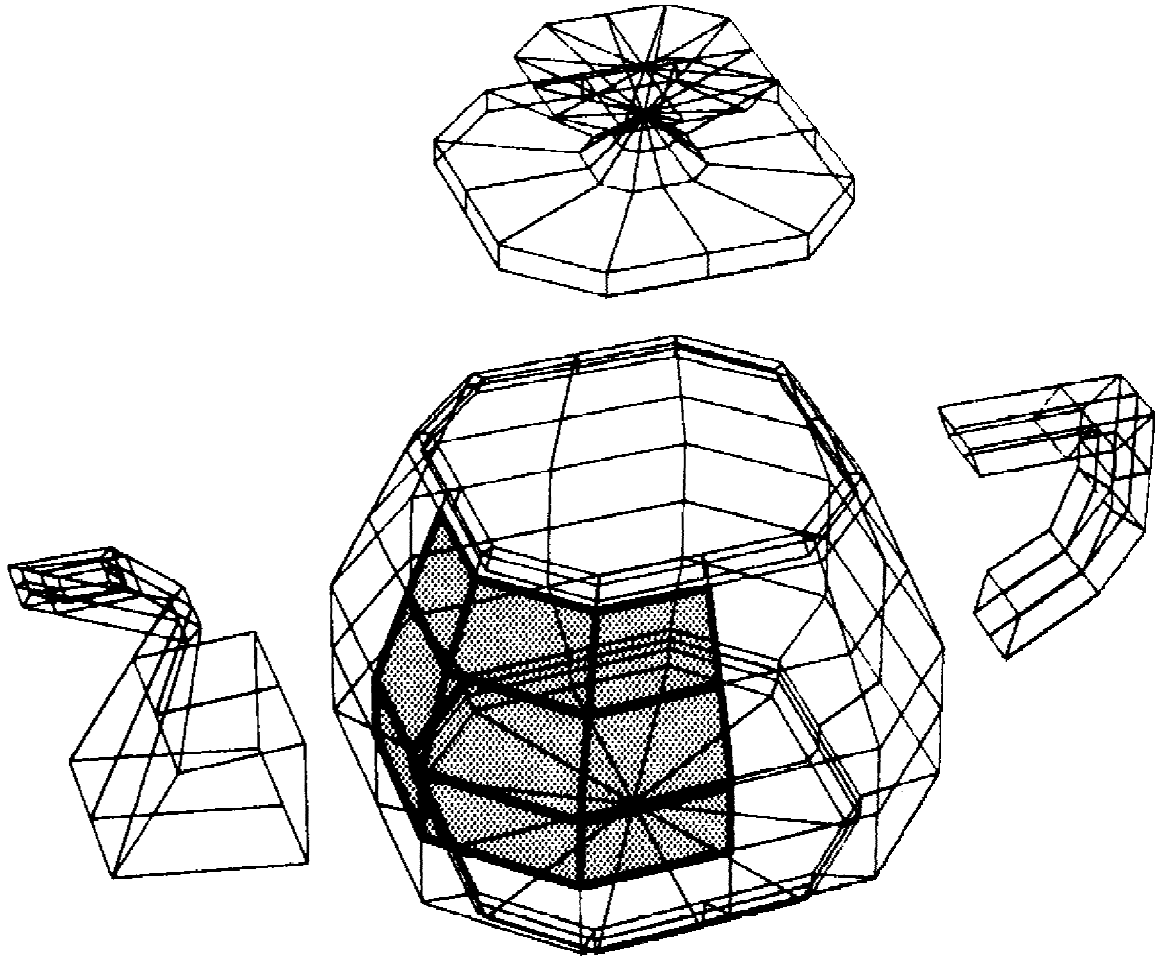
Poleg posameznih prednosti naštetih zlepkov imajo taki parametrični popisi zlepkov tudi številne slabosti, predvsem v izdelavi ustreznega uporabniškega vmesnika in zagotovitvi kontinuitete in enostavnosti delitve.

5.3 Objekti in bikubični parametrični zlepki

Uporaba bikubičnih parametričnih zlepkov teži k temu, da bo ostala stogo namenska le za področje CAD in raziskave. Tako predstavitev slike z mnogokotniškimi mrežami še vedno prevladuje, za kar je mnogo dobrih razlogov. Predvsem je izdelava baze podatkov z mnogokotniško mrežo neprimerno enostavnejša kot pa gradnja baze, ko uporabimo bikubične zlepke. Tako so zelo kompleksne 3D oblike objektov enostavno izmerljive s 3D digitizerjem in zapisane kot mreža poligonov. Brez pomoči obsežnega CAD paketa je težko zgraditi objekt, ki temelji na zlepkih.

y

Za primer predstavitve objekta, popisanega z Bézierjevimi bikubičnimi zlepki sem izbral čajnik z univerze Utah, ki je bila center za razvoj algoritmov računalniškega upodabljanja objektov v zgodnjih 70-ih letih. Leta 1975 je M. Newell[6] razvil čajnik, ki je postal nekaka primerjava ("benchmark") v računalniški grafiki, saj ga zasledimo v vseh boljših knjigah in prospektih proizvajalcev, ki se ukvarjajo z računalnško grafiko. Resnični čajnik se zdaj nahaja v računališkem muzeju v Bostonu poleg svojega računalniškega dvojnika na zaslonu.



Slika 29 Kontrolne točke čajnika sestavljenega iz 32 Bézierjevih bikubičnih zlepkov

[6]Vir [WATT90] stran 395.

6 Osnovni model odboja

6.1 Enostavni model odboja

Enostavni modeli odboja svetlobe skušajo vsebovati povezavo s površino. Standardni model v računalniški grafiki je kompromis med hitrostjo obdelave in realnostjo modela (Phong, 1975). Enostavni modeli odboja v računalniški grafiki so kompromis dveh stvari:

- aproksimacije geometrije objektov in
- aproksimacije dogodka na površini.

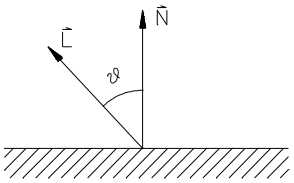
Očitna enačba dogodka na površini je:

dogodek = zrcalni odboj svetlobe (refleksija) +
+ svetloba, ki se raztrosi
+ absorbirana svetloba
+ svetloba, ki jo površina oddaja

Difuzna (raztresena) in reflektirana svetloba sta komponenti z iste površine. Reflektirana svetloba ima lastnosti odboja svetlobe na zrcalu, torej je vpadni in odbiti kot glede na normalo ploskve enak. Večina predmetov v naravi ne emitira svetlobe, ampak jo raje absorbira in reflektira. Dnevna svetloba, ki pade na površino, se delno absorbira in delno odbije.

Predmet običajno reflektira le del bele svetlobe določene valovne dolžine, tako da človeško oko to zaznava kot barvni odtenek določene intenzitete. Ker hrapave (mat) površine razpršujejo svetlobo v vseh smereh enako, lahko za take površine uporabimo Lambertov kosinusni zakon:

$$I_D = I_I K_D \cos\vartheta , \quad 0 \le \vartheta \le \pi/2$$



pri čemer je I_I moč svetlobnega izvora, ϑ kot med normalo površine in smerjo izvora svetlobe. Konstanta K_D popisuje lastnost površine (hrapavost oz. difuzna reflektivnost) in je v mejah [0..1].

$K_D = 0$ --> ploskev absorbira vso svetlobo
$K_D = 1$ --> ploskev odbije vso vpadno svetlobo.

Če sta normala ploskve in smerni vektor izvora svetlobe normirani, lahko računamo intenziteto ploskve Id s skalarnim produktom

$$I_D = I_I \, k_D \, (\mathbf{L} \cdot \mathbf{N}) \, .$$

V primeru, da imamo več svetlobnih izvorov, potem je

$$I_D = k_D \sum I_{I,N} \, (\mathbf{L_N} \cdot \mathbf{N}) \, .$$

V realnih razmerah je prisotna tudi ambientna svetloba kot rezultat večkratnih refleksij od sten in predmetov ter osvetljuje površino iz vseh smeri ne glede na njihovo orientacijo.

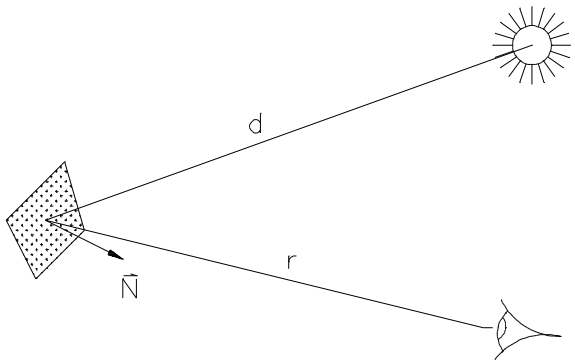
Ploskve, ki jih svetlobni izvor direktno ne zadene, so tako še vedno osvetljene. Če ta približek združimo z difuznim odbojem, dobimo enačbo za odbito svetlobo

$$I_D = I_A k_A + k_D \sum I_{I,N} \, (\mathbf{L_N} \cdot \mathbf{N}) \, ,$$

kjer k_A predstavlja delež ambientne svetlobe in zavzema vrednosti od 0 do 1.
Tu privzamemo, da objekti prosto plavajo v prostoru in tako ne obstaja nobena interakcija med njimi ali ozadjem.

Enačbo intenzitete površine lahko še izpopolnimo, če vključimo vanjo vpliv oddaljenosti ploskve od točke gledanja. Ker se energija svetlobe manjša s kvadratom oddaljenosti, [GUID88-396] zapišemo

R je vsota razdalje točke gledanja od ploskve in razdalje svetlobnega izvora od ploskve (r+D). Pri senčenju bi torej po tej enačbi morali izračunavati smer svetlobnega telesa za vsako ploskvico oziroma točko posebej in normirati smerni vektor, tako da bi odšteli od vektorja izvora vektor točke na



$$I = I_A k_A + I_P k_D \frac{(\vec{L} \cdot \vec{N})}{R^2}$$

ploskvici in izračunali normo. V vsakem primeru pa moramo izračunati normalo ploskve.

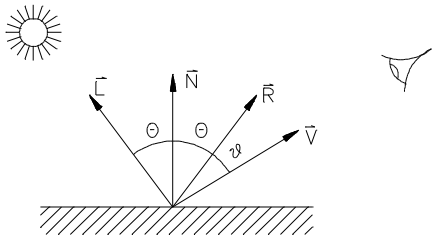
Metoda je že na prvi pogled zamudna, tako da se je v praksi veliko bolj uveljavila enačba, ki upošteva le linearno zmanjševanje intenzitete, za svetlobna telesa pa privzamemo, da se nahajajo v neskončnosti. Korigirana enačba je

$$I = I_A k_A + I_P k_D \frac{(\vec{L} \cdot \vec{N})}{(r + k)}$$

kjer je r oddaljenost ploskve od točke gledanja in k konstanta (k>0), ki je odvisna od aplikacije.

6.2 Zrcalni odboj

Večina površin nima samo mat zasnove ampak imajo tudi nekaj deleža sijaja ali leska. To so praviloma gladke površine. Blesk ni odvisen od barve površine ampak od barve svetlobnega izvora. Popolnoma gladka površina je zrcalo. Oko bo zaznalo lesk le takrat, ko bo odbiti žarek v smeri gledišča oziroma v ozkem pasu gledišča.



Bui-Tuong Phonh je leta 1975 modeliral ta efekt z empirično formulo $\cos^n\vartheta$. Za idealno zrcalo je n neskončen, v praksi pa se giblje vrednost od 1 do 200 [GUID88-398]. Bolj, ko postaja površina mat, bolj širok je spekter in manjša je intenzivnost, kar se popiše s konstanto k_b. Enačbo intenzitete lahko dopolnimo

$$I = I_A k_A + I_i \frac{k_d(\vec{L}\cdot N) + k_s \cos^n\vartheta}{r + k}$$

$$I = I_A k_A + I_i \frac{k_d(\vec{L}\cdot N) + k_s(R\cdot V)^n}{r + k}$$

6.3 Povzetek Phongovega modela [WATT89-55]

- Svetlobni izvori so točkovni. Vsaka intenziteta svetlobnih teles je ignorirana.
- Vsaka geometrija razen normale površine se ignorira (svetlobno telo in opazovalec sta postavljena v neskončnost)
- Empirični model se uporablja za zrcalni odboj.
- Barva deleža zrcalnega odboja je barva svetlobnega izvora.
- Ambientna svetloba je upoštevana kot konstanta po vsem prostoru.

7 Inkrementalne tehnike senčenja

Inkrementalne tehnike senčenja so bolj popularne od ostalih metod upodabljanja scen v računalniški grafiki. Zaradi enostavne in hitre aplikacije sem jih uporabil tudi v mojem programu za senčenje.

Gouraudova in Phongova metoda sta posebno priljubljeni in ju lahko najdemo v večini grafičnih sistemov upodabljanja. Gouraudova metoda je hitrejša od Phongove, ima pa zato tudi svoje slabosti, saj ne prikaže najbolje odbleske svetlobnega izvora. Gouraudova metoda je postala nekak standard v računalniški grafiki tako, da so letos izdelali integrirano vezje za grafični krmilnik, ki ima to metodo senčenja že vgrajeno[7].
Ravno tako je Phongova in Gouraudova metoda v predlaganem grafičnem standardu PHIGS+.

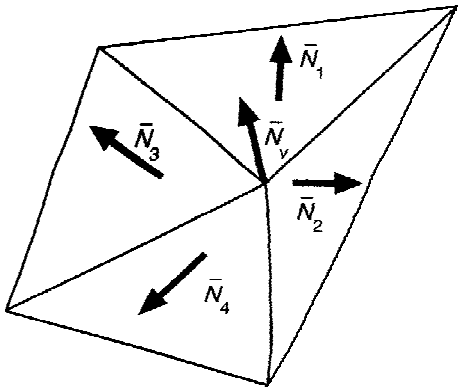
7.1 Gouraudovo senčenje

Prva metoda, ki je rešila problem senčenja celih mnogokotnikov je bila *Gouraudova* metoda (Gouraud, 1971). Pri tej metodi interpoliramo intenziteto površine za vsako rastrsko točko mnogokotnika, ki ga senčimo.

Gouraudova metoda zahteva, da so znane vse normale mnogokotnika, ki se senči, saj je mogoče le tako interpolirati vrednosti za ostale točke znotraj mnogokotnika. 3D modelirnik mora torej poleg koordinat točke za procesu senčenja podati tudi normalo v točki. Alternativni način pa je, izračun normale z izračunom poprečne vrednosti normal poligonov, ki vsebujejo skupno vozliščno točko.

---

[7] *Electronic Design*, vol. 39-1991, No. 6 - April 28, Penton Publications, Cleveland, OH, USA



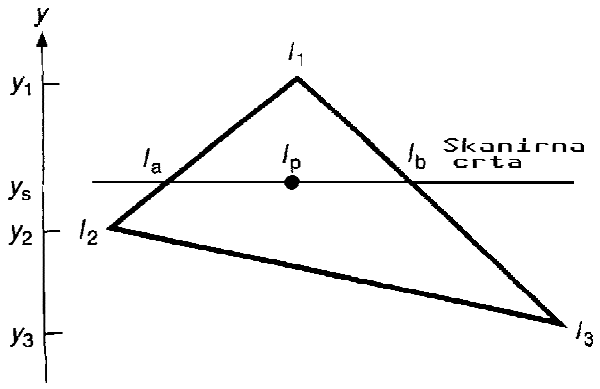
Slika 33 Izračun normale vozlišča z poprečenjem normal mnogokotnikov

Normala vozlišča po alternativnem izračunu je

$$N_v = \frac{\sum_{i=1}^{n} N_i}{\left|\sum_{i=1}^{n} N_i\right|} .$$

Za mnogokotnike se izračuna normala z vektorskim produktom treh zaporednih vozlišč. Običajno pričnemo z izračunom v prvem vozlišču mnogokotnika z uporabo naslednjih dveh. Če norma vektorskega produkta kaže bližino osnovne natančnosti realnih števil, se uporabi naslednji set treh vozlišč.

Interpolacija mnogokotnika poteka po rezinah osi *y*, za vsako točko te rastrske linije (angl. *scan line*).



Slika 34 Interpolacija intenzivosti

$$I_a = I_1 - (I_1 - I_2)\frac{y_1 - y_s}{y_1 - y_2} ,$$

$$I_b = I_1 - (I_1 - 1_3)\frac{y_1 - y_s}{y_1 - y_3} ,$$

$$I_p = I_b - (I_b - I_a)\frac{x_b - x_p}{x_b - x_a} .$$

Da povečamo hitrost izračuna rasterske črte, uporabimo inkrementalen način računanja intenzitete v vsakem pikslu z enačbo

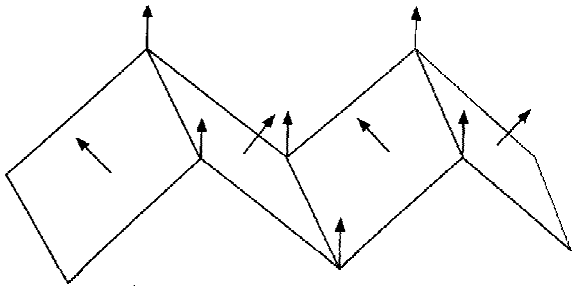
$$\Delta I_s = \frac{\Delta x}{x_b - x_a}(I_b - I_a) ,$$

in

$$I_{s,n} = I_{s,n-1} + \Delta I_s .$$

Posamezno povečanje po osi *x* se izračuna iz razmerja med svetovno in zaslonsko razdaljo med dvema točkama.

Gouraudova medoda ima sledeče slabosti:



Slika 35 Napaka povprečenja na narebreni površini

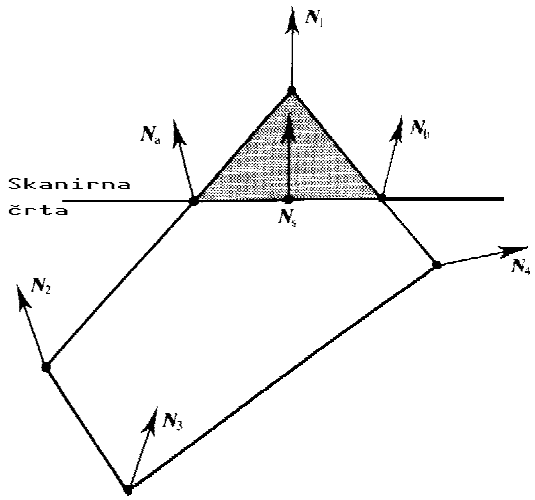
• Nastanejo lahko popačitve pri narebrenih površinah, ko imajo kljub narebrenosti normale vozlišč isto usmerjenost. Narebrenost površine po senčenju vseh mnogokotnikov ni vidna.

• Če se povpreči normale vozlišč iz normal mnogokotnikov, robne točke površine nimajo najpravilnejše usmeritve, kar se lepo pokaže pri ukrivljenih površinah.

7.2 Phongovo senčenje

Phongova inkrementalna metoda senčenja odpravi nekatere slabosti Gouraudovega senčenja. Odpravljena je slabost nezmožnosti prikazovanja odbleska Gouraudove metode. Namesto interpolacije intenzitete površine, se tu interpolirajo normale vozlišč. Tako je potrebno izračunati za vsak piksel svojo normalo. Po izračunu normale v pikslu uporabimo enega od refleksijskih modelov.

Interpolacija po robovih mnogokotnika je lahko tudi tu inkrementalna. Po skanirni črti pa je potrebno izračunane normale še normirati, ker interpolacija pokvari normo normale v pikslu.



Slika 36 Interpolacija normal mnogokotnika

Pri izračunu po Phongovi metodi je potrebno ravno tako, kot pri Gouraudovi metodi, izračunati normale izračunati normale v vozliščih. Sledi izračun normal na robovih mnogokotnika z interpolacijo normal vozlišč in nato še po skanirni črti. Po izračunu normale v pikslu se aplicira eden od refleksijskih modelov.
Podobno kot pri Gouraudovi metodi se tu izračuna normale z linerano enačbo

$$N_a = \frac{1}{y_1 - y_2}[N_1(y_s - y_2) + N_2(y_1 - y_s)] ,$$

$$N_b = \frac{1}{y_1 - y_4}[N_1(y_s - y_4) + N_2(y_1 - y_s)] ,$$

$$N_s = \frac{1}{x_b - x_z}[N_s(x_b - x_s) + N_b(x_s - x_a)] .$$

Inkrementalne enačbe se izvedejo podobno kot pri Gouraudovi metodi. Iz enačb je razvidno, da je potreben čas za senčenje mnogokotnika pri Phongovi metodi precej večji, saj namesto interpolacije ene vrednosti (intenzitete) tu interpoliramo tri koordinate normale, kar tudi porabi več pomnilnika računalnika pri rasterizaciji robov.

Za povečanje hitrosti senčenja nekateri avtorji predlagajo, da bi mnogokotnike, na katerih se ne pojavlja odblesk svetlobe, senčili po Gouraudovi metodi. Mnogokotnike, ki pa imajo odblesk pa naj bi senčili po Phongovi metodi. Za ugotovitev ali se na mnogokotnihu pojavi odblesk pa se uporabi *H-TEST* [WATT89-92].

7.3 Rasterizacija robov

Mnogokotnik, ki ga želimo senčiti z eno od naštetih metod je potrebno rasterizirati. To pomeni, da je potrebno poiskati vse točke mnogokotnika na zaslonu oz. podobni rasterski napravi. Pri prikazovanju robov v računalniški grafiki sta v veljavi dva načina, eden je rasterizacija za prikazovanje črt, drugi pa je rasterizacija za senčenje mnogokotnikov. Pri senčenju se uporablja drugi način, ki ima v eni rasterski črti vedno le en piksel enega rasteriziranega roba.

Če hočemo rasterizirati rob po pikslih ekrana, vzamemo začetno in končno točko na zaslonu, med katerima moramo izračunati še položaj ostalih točk, ki ležijo med njima.

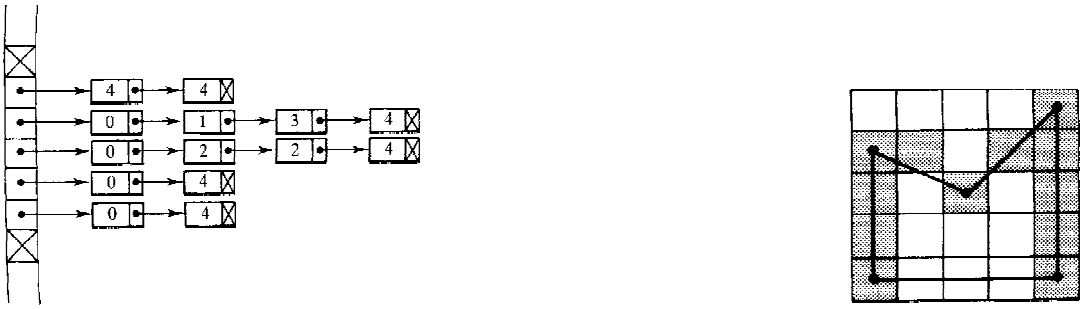
Najenostavnejši algoritem za rasterizacijo robov mnogokotnika je:

$x=x_{začetek}$
$korak=(x_{začetek}-x_{konec})/(y_{konec}-y_{začetek})$
ponavljaj za $y=y_{začetek}$ do y_{konec} {
    izpiši(zaokrožen(x), y)
    x=x+korak
}
.

Poleg omenjenega algoritma obstajajo še izboljšave [WATT89-99], ki pa bistveno ne povečajo hitrosti senčenja.

V praksi rasteriziramo ne rasteriziramo samo enega robu ampak vse robove mnogokotnika in nato senčimo notranje točka, ki jih omejujejo rasterizirane točke mnogokotnika. Rezultat rasterizacije se največkrat shranjuje v tabelo povezanih seznamov[8].

[8] F77 ne omogoča gradnje seznamov, zato je potrebno rezervirati ustrezno veliko tabelo za podatke.



Slika 37 Tabela seznamov robnih točk

Za pravilno rasterizacijo je potrebno uvesti naslednja pravila:

1. Robovi, ki so horizontalni se ne rasterizirajo
2. Rasterizacija naj poteka vedno v eni smeri za vse robove mnogokotnika, npr. od spodaj navzgor in od leve proti desni.
3. Rob, ki poteka v mejah od spodnje do zgornje točke mora generirati rasterske točke za kasnejšo interpolacijo tako, da za vsako rastrsko črto izračuna interpolirane vrednosti za eno od inkrementalnih metod. Pri rasterizaciji pa mora zadnjo (običajno je to zgornja točka) izpustiti iz rasterizacije.
4. Podobno kot pri pravilu 2 se tudi pri rasterizaciji po horizontali izpusti zadnja točka na ekranu (x_{leva} do x_{desna}-1). Če koordinata *x* dveh zaporednih pikslov enaka, se senčenje takega segmenta ignorira, sicer senčimo piksle po eni od inkrementalnih metod.

Izpuščanje zadnje točke rasteriziranega robu je potrebno zato da črta, ki se nadaljuje iz prejšnje črte nima rasterizirane isti piksel kot končna točka prve črte. Če je mnogokotnik konveksen, potem je ena rasterska črta v seznamu vedno vsebuje dve točki ali nobeno.

7.4 Vmesni pomnilnik za koordinate z

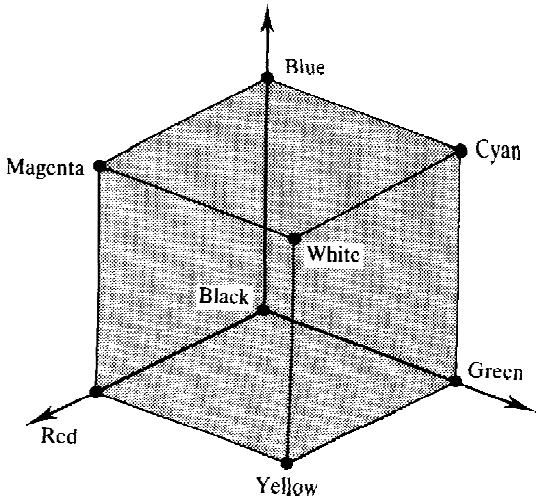
Če hočemo pravilno prikazovati objekte na zaslonu je potrebno skrivati dele površin, ki imajo z koordinato na zaslonu bolj oddaljeno od delov površin, ki so pred površino. Z uporabo inkrementalnega senčenja je osnovna ločljivost slike na zaslonu izražena v pikslih. Če hočemo torej prikazovati piksle, ki imajo koordinato *z* večjo kot ostali piksli na isti *x* in *y* poziciji, si moramo sproti za vsak piksel, ki ga izrišemo na zaslon, zapomniti kolikšna je globina točke. Če ima nova točka z isto pozicijo na zaslonu koordinato *z* bližjo kot točka, ki je bila že prej izrisana na tej poziciji, potem se ta točka ponovno izriše z novo vrednostjo (bližjo) koordinate *z*. Ta algoritem ("*z-buffer alggorithm*") torej poleg osnovnega pomnilnika za indekse barv zaslona zahteva ravno tolikšen pomnilnik še za koordinate *z* zaslona. Postopek uvedbe takega algoritma je pri rasterizaciji najenostavnejši, zahteva pa veliko količino pomnilniškega bazena. Običajno se za vmesni pomnilnik z koordinat uporabi celoštevilčno *x-y* polje, ki zavzame manj spomina kot pa polje realnih števil.

Pred uporabo vmesnega pomnilnika je potrebno vse vrednosti vmesnega pomnilnika nastaviti na največjo globino, tako da bo vsaka točka, ki se bo rasterizirala imela večjo koordinato *z* od začetne.

Mnogokotnike senčimo po poljubnem vrstnem redu, saj ni potrebno sortiranje mnogokotnikov po globini, kot je to potrebno pri senčenju s polnjenjem mnogokotnikov. Čas izvajanja je pri enostavnih scenah malo daljši kot pri ostalih prioritetnih algoritmih, vendar pa je za kompleksne scene z več objekti in zahtevnimi prekrivanji robov, to najhitrejši poznani algoritem. Tako imajo delovne postaje firme *Silicon Graphics* tak pomnilnik že vgrajen v osnovo grafičnega procesorja.

8 Barvni modeli

8.1 RGB barvni model



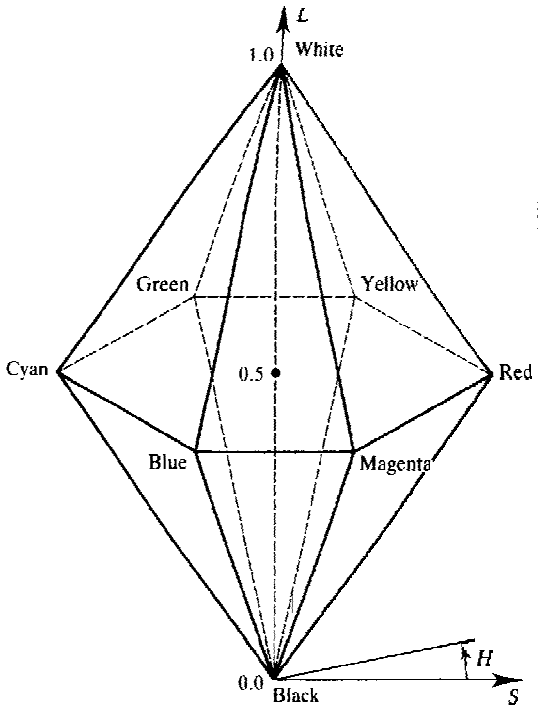
Slika 39 Barvni prostor RGB

RGB je standarden model za monitorje. Monitor računalnika je krmiljen s tremi barvnimi intezitetami (Red, Green, Blue). RGB model je lahko predstavljen kot kocka z robovi označenimi od 0 do 1, kar predstavlja količino posameznih primarnih barv (rdeča, zelena in modra) na stranicah, ki so ortogonalne.

Pomembno je poudariti, da RGB kocka predstavlja le del odtenkov, ki jih lahko vidi človeško oko. Ta model ne predstavlja povsem uniformen barvni prostor, kar se najbolj pozna pri malih intenzitetah svetlobe. Pri 8-bitnih D/A konverterjih za vsako osnovno barvo imamo po 256 odtenkov za vsako stranico barvne kocke (3x8=24 bitni barvni sistem ali 16,7 mio barvnih odtenkov). Za nekatera območja je potrebno napraviti 20 do 30 korakov, da bi oko sploh opazilo razliko.

Zaradi enostavnosti izdelave in dobre aproksimacije delovanja človeškega očesa je ta model osnova za druge barvne modele. Tako je ta model osnova za grafičnega standarda GKS.

8.2 Barvni model HLS (Hue, Lightness, Saturation)



Slika 40 HLS barvni prostor

HLS barvni model temelji za Ostwaldovem barvnem modelu (Ostwald, 1931) in ga uporablja firma TEKTRONIX kot osnovo za njihove barvne grafične aplikacije. Na terminalih te firme je poleg osnovnega modela RGB vgrajen še ta model, ker je v aplikacijah s senčenjem in sledenjem žarka, s takim modelom enostavneje popisati večbarvne površine in odtenke le-teh. Model je zgrajen iz dveh stožcev, spojenih na osnovni ploskvi. Barve ("hue") so predstavljene z vrtenjem za določen kot po simetrali modela.

Centralna os je označena z L ("lightness" ali svetlost) in predstavlja osvetlitev posamezne barve.

Če se gibljemo od centralne osi proti robu stožca, se povečuje nasičenost barve ("saturation"). Bolj ko gremo proti centru osi bolj se povečuje vsebnost sivih odtenkov.

[ENCWOLF86-367]
Konverzija barvnega modela RGB v barvni model HLS:

H - Hue ali barvni odtenek
L - Lightness ali svetlost
S - Saturation ali nasičenost

Vhodne vrednosti barvnega modela so intenzitete osnovnih barv RGB v obsegu od 0 do 1. Izhodne vrednosti pa so H, ki je na stožcu predstavljen z kotom izraženim v stopinjah in zavzame vrednosti od 0 do 360, L in S pa zavzameta vrednosti od 0 do 1.

Algoritem je sledeč:

1. max= Max(R,G,B);
   min= Min(R,G,B);
2. L=(min+max)/2;
3. if(min==max) {
   S=0;
   return;
   }
4. if(L<=0.5) S=(max-min)/(max+min);
   else S=(max-min)/(2-max-min);
5. r=(R-min)/(max-min);
   g=(G-min)/(max-min);
   b=(B-min)/(max-min);
6. if(R==max) H=g-b;
   else if(G==max) H=2+b-r;
   else H=4+r-g;
   if(H<0) H=H+6
7. H=H*60

Možna je tudi obratna konverzija, ki je tudi bolj primerna za uporabo ravno pri programiranju v grafičnem standardu GKS, saj se v GKS-u uporablja barvni model RGB. Barvne odtenke se nastavlja z ukazom Set Color Repesentation (GSCR).

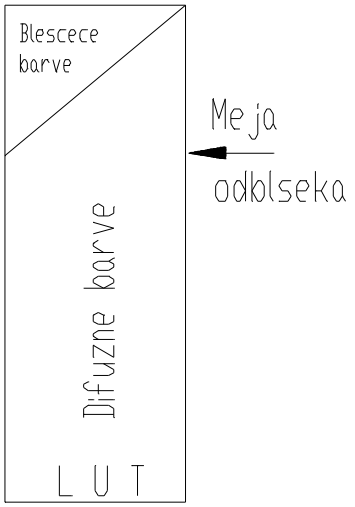
Algoritem konverzije iz HLS v RGB je sledeč:

1. H=H/60;
2. I=int(H);
   F=H-I;
3. if(L<=0.5) max=L*(1+S);
   else max=L+S-L*S;
   min=2*L-max;
4. DM=max-min;
5. if(S==0) (R,G,B)=(L,L,L)
   else
     switch(I) {
        case 0: (R,G,B)=(max, min+F*DM, min)
        case 1: (R,G,B)=(min+(1-F)*DM, max, min)
        case 2: (R,G,B)=(min, max, min+F*DM)
        case 3: (R,G,B)=(min, min+(1-F)*DM, max)
        case 4: (R,G,B)=(min+F*DM, min, max
        case 5: (R,G,B)=(max, min, min+(1-F)*DM)
      };

8.3 Uporaba tabele indeksov barv za izris objekta

Današnji barvni prikazovalniki imajo omejeno število barv, ki jih lahko prikažemo naenkrat na zaslonu. To število je pogojeno s t.i. številom bitnih ravnin zaslona. Terminali imajo možnost prikaza od 16 do 256 barrvnih odtenkov naenkrat, kar pomeni, da imajo od 4 do 8 bitnih ravnin (2^4=16 in 2^8=256 barvnih odtenkov). Za vsak barvni indeks pa lahko izberemo katerikoli barvni odtenek, ki ga podpira D/A konverter za vsako od RGB intenzitet. Če je, npr. za vsako od treh RGB intenzitet po en konverter s 4-bitno D/A konverzijo imamo na izbiro za vsak barvni indeks 4*4*4=64 barvnih odtenkov. Pri 8-bitnih konverterjih pa nam to število naraste na preko 16 mio. barvnih odtenkov, čeprav jih od tega števila lahko naenkrat uporabimo le 256.

Izračunavanje ustrezne intenzitete barve po Phongovem modelu odboja lahko vzame veliko časa računalniku in je zato ugodneje uporabiti funkcijo, ki korigira tabelo barvnih indeksov **LUT** in s tem se izognemo ponovnem izračunavanju scene. Posebej je važno, da je možno hitro spremeniti barvo objekta in barvo svetlobnega izvora. Pri Phongovem modelu je potrebno za vsako spremembo koeficientov ponovno izračunavati barvne indekse. Znano je, da je intenziteta v točki površine predvsem funkcija normale površine v tej točki.



Slika 41 Uporaba LUT za prikaz objektov

Ideja, ki se je v praksi pokazala kot zelo ugodna, je oporaba korigiranega profila barvnih indeksov. Tako se pri senčenju ne izračunava intenziteto po Phongovem modelu, ampak se izračuna le skalarni produkt med izvorom svetlobe in normalo v točki. Profil LUT se korigira z namenom, da lahko simuliramo odbleske na površini. S pravilno izbiro ustreznih vrednosti profila je mogoče simulirati barvo svetlobnega izvora, intenziteto odbleska in jakost ambientne svetlobe, ki ni nujno iste barve, kot je svetlobni izvor. Slabost te metode je, da je potrebno rezervirati za vsak tip površine rezervirati del LUT, kar pa je očitna omejitev pri številu in barvnem razponu površin.

9 Baza 3D podatkov

Sestava baze podatkov za 3D modeliranje je pomembna in vredna truda, če se namerava uporabiti v programu. Zgodnje napake v strukturi podatkov ostanejo in jih je kasneje praktično nemogoče odpraviti.
Baza naj bi imela sledeče značilnosti:

1. Hierarhičen koncept: objekt -⟶ površina -⟶ mnogokotnik
2. Vozlišča so zapisana v bazo samo enkrat in različni mnogokotniki si lahko delijo ista vozlišča, kar pa ne pomeni, da si lahko različne površine delijo ista vozlišča.
3. Normale mnogokotnikov so shranjene (za izločanje) kot tudi normale vozlišč (za senčenje).

Scena je lahko sestavljena iz večjega števila objektov in v bazi podatkov zapisana kot seznam objektov.
Zapis objekta je sestavljen:

- številka objekta
- število površin
- kazalec na začetno površino
- kazalec na začetno točko
- transformacijska matrike objekta za uporabila v transformaciji objekta (rotacija, skaliranje itd.)
- kazalec na naslednji objekt

Vsaka površina je sestavljena iz mreže mnogokotnikov.
Zapis površine je sestavljen:

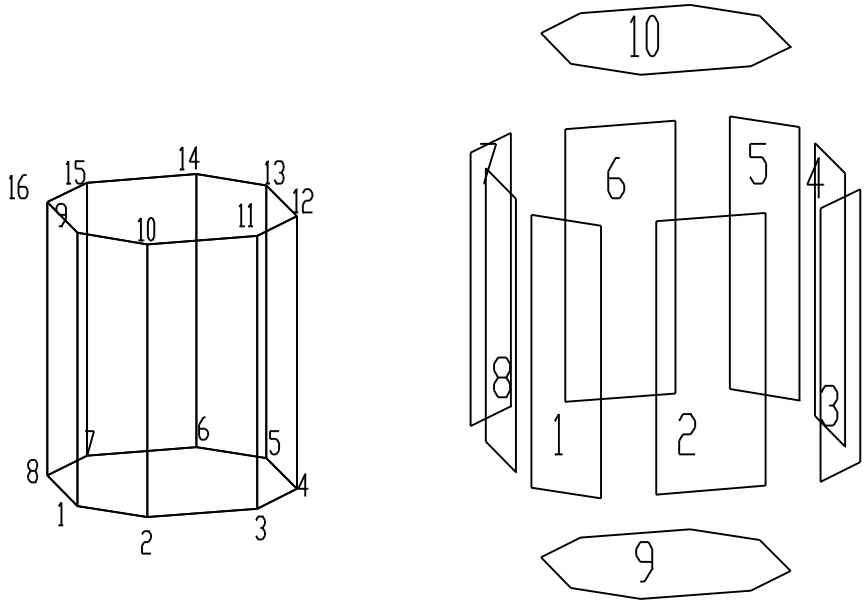
- število poligonov
- kazalec na začetni mnogokotnik
- lastnost površine (barva, prozornost, gladkost)
- kazalec na naslednjo površino

Zapis mnogokotnika vsebuje prostor za normalo in zastavico, ki kaže ali je mnogokotnik izločen zaradi tega, ker njegova normala kaže v nasprotno smer, kot pa je smer točke gledanja (eliminacija zadnje strani).

Mnogokotnik je sestavljen iz množice vozlišč, v katerih je za vsako vozlišče zapisano:

- lokalna pozicija vozlišča
- svetovna pozicija vozlišča
- zaslonska pozicija vozšča
- kazalec na seznam mnogokotnikov, ki vsebujejo tekoče vozlišče
- normala vozlišča
- kazalec na naslednje vozlišče

Povezave so med gradniki scene so torej lahko dvosmerne ali pa delno dvosmerne. Popolna dvosmernost je, da lahko iz informacije vozlišča dobimo seznam mnogokotnikov, iz mnogokotnikov dobimo površino, iz nje objekt in iz objekta sceno. To omogoča, da so koordinate vozlišč zapisane samo enkrat, pri čemer se prihrani pomnilnik računalnika in tudi omogoča, da se pri interaktivnem vnašanju in premikanju vozlišč premikajo vsi mnogokotniki, ki vsebujejo vozlišče.
Če se scene sestavljajo v enakomernih časovnih presledkih dobimo animacijo ali film.



Slika 42 Gradniki objekta

Primer datoteke s podatki o sceni s popolno dvosmerno vezavi za cilinder, ki ga aproksimiramo z osmerokotnikom:

```
 16 10 3 1        {število vozlišč, mnogokotnikov, površin, objektov}
 0  0.0 10.0 0.0  {št. vozlišča, koordinata x, y,  in z)
 1 10.0 0.00 0.0
 2 30.0 0.00 0.0
 3 40.0 10.0 0.0
 4 40.0 30.0 0.0
 5 30.0 40.0 0.0
 6 10.0 40.0 0.0
 7  0.0 30.0 0.0
 8  0.0 10.0 50.0
 9 10.0 0.00 50.0
10 30.0 0.00 50.0
11 40.0 10.0 50.0
12 40.0 30.0 50.0
13 30.0 40.0 50.0
14 10.0 40.0 50.0
15  0.0 30.0 50.0
0 0 4 0 1 9 8     {št. mnogokotnika, površine, št. vozlišč, indeksi vozlišč}
1 0 4 1 2 10 9
2 0 4 2 3 11 10
3 0 4 3 4 12 11
4 0 4 4 5 13 12
5 0 4 5 6 14 13
6 0 4 6 7 15 14
7 0 4 7 0 8 15
8 1 8 7 6 5 4 3 2 1 0
9 2 8 8 9 10 11 12 13 14 15
```

Bazo podatkov je najenostavneje graditi s programskimi jeziki, ki omogočajo strukture in dinamično alokacijo in dealokacijo novih spremenljivk. Za izdelavo je bil izbran jezik C, kot najbolj priljubljen in dovolj močan jezik za gradnjo baze grafičnih (numeričnih) podatkov.

Potrebno je bilo zagotoviti dvosmernost povezav na tistih delih baze, kjer je to potrebno. Dvosmernost je uvedena na relaciji mnogokotnik ↔ normala vozlišča ↔ vozlišče. Ostali del grafa (glej sliko) je hierarhično grajen tako, da je iz najvišjih gradnikov enostavno priti do osnovnih gradnikov baze (vozlišča in normale). Poleg hierarhičnih povezav pa imamo tudi hitro povezavo iz objekta na vsa vozlišča, kar omogoča hitro transformacijo vseg točk objekta in izračun normal vozlišč.

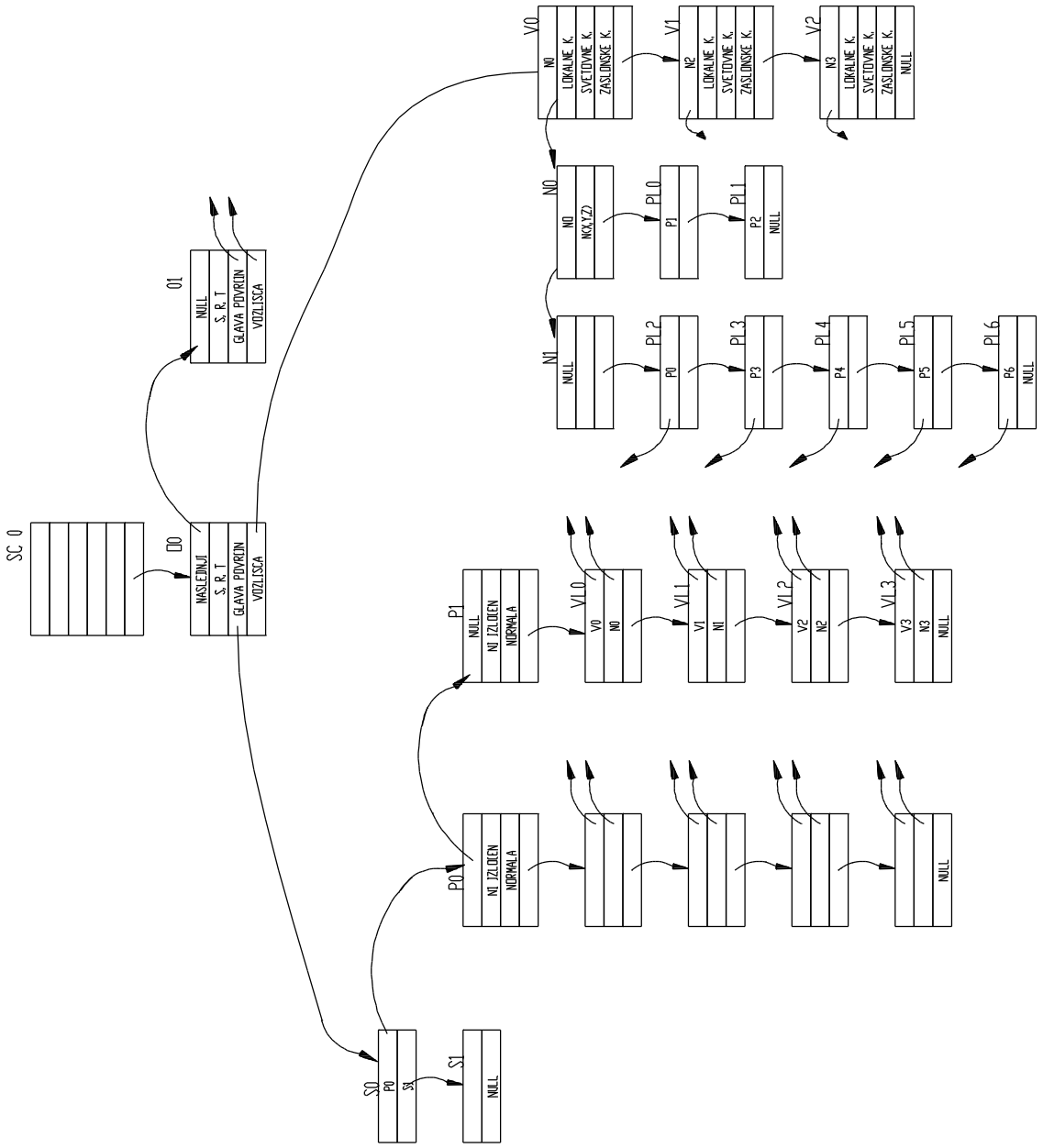
9.1 Algoritem izdelave 3D baze podatkov je sledeč:

1. Iz datoteke podatkov preberemo število vozlišč, poligonov, površin in objektov tako, da od začetka datoteke beremo prve številke v vrsticah sekvenčne datoteke.

2. Alociraj začasno tabelo kazalcev v ustrezni velikosti za:
   - vozlišča
   - poligone
   - tekoče normale vozlišč

4. Beri vozlišča:
   - naredi seznam vozlišč
   - nastavi vrednosti v tabeli kazalcev na vozlišča

5. Beri poligone:
   - naredi seznam poligonov
   - v tabelo kazalcev na poligone vstavi tekoč poligon
   - za vsak poligon naredi seznam kazalcev na vozlišča



Slika 43 Del baze podatkov, ki je uporabljena v programu

6. Beri površine in za vsako površino:

   - naredi seznam kazalcev na poligone

   - za vsako vozlišče alociraj novo celico in nastavi kazalce v tabeli tekočih normal, da kažejo na te novo alocirane celice.

   - za vsak seznam kazalcev na poligone:

- za vsako celico seznama nastavi kazalec na normalo vozlišča
- dodamo kazalec na poligon v seznam tekoče normale
- za vsako vozlišče dealociraj celico normale, če ne vsebuje seznama vozlišč

7. Po branju vseh objektov dealociraj vse začasne tabele

8. Če se pokaže potreba po realokaciji spominskega bazena zaradi prevelike fragmentacije, sprostimo ves spomin tako, da celotno bazo zapišemo v začasno datoteko in nato beremo nazaj v spomin.

9.2 Alternativni algoritem izdelave 3D baze:

Imamo več datotek z objekti.

1. preberemo št. vozlišč, št. poligonov in št. površin
   - alociramo tabele kazalcev na vozlisca in poligone

2. beremo vozlišča s tem, da sproti inicilaliziramo tabelo vozlišč

3. Beremo mnogokotnike (poligone) tako da za vsak mnogokotnik:
   - za poligon kreiramo novo celico za poligon in jo dodamo tekoči površini.
   - nastavimo kazalec v tabeli kazalcev na poligone
   - če se pojavi nova površina:
   - izločimo vse celice normal vozlišč, ki nimajo pripetega seznama poligonov.
   - kreiramo novo celico za površino
   - vsaki celici vozlišča dodamo novo celico za normalo vozlišča
   - beri stevilke vozlišč poligona in za vsako številko:
   - kreiraj novo celico seznama vozlišč
   - nastavi kazalec na vozlišče s pomočjo tabele vozlišč
   - poišči zadnjo celico normale vozlišča in tej celici dodaj v seznam, kazalc na poligon s pomočjo tabele poligonov
   - nastavi kazalec na normalo vozlišča

Alternativni algoritem izelave 3D baze je za senčenje bolj ugoden saj struktura podatkov in njene povezave niso tako zahtevne. Baza predstavljena v programu za senčenje je zgrajena na alternativnem algoritmu.

10 Zaključek

Diplomska naloga, je pokazala nekaj problemov, ki sem jih poskušal tudi praktično rešiti. Odločiti sem se moral za najosnovnejše algoritme, ki računalniku niso delale časovnih težav. Mnoga področja in metode pri upodabljanju so ostala neobdelana. Energijska metoda, sledenje žarka, gama-korekcija, teksture na površinah in ostale metode so področja, ki bi jih bilo potrebno obdelati v kasnejšem času. Ker je področje računalniške grafike razmeroma mlado, se osnovne raziskave vršijo predvsem na univerzah, ki dajo kaj na svoj sloves.

Računalniška grafika je sedanjem času nuja in pričakovati je, da bodo povpraševanja po takem znanju vse večja. To je bil tudi eden od razlogov, da sem se odločil ravno za tako tematiko diplomske naloge.

Junij, 1991

Leon Kos

11 Literatura

ANGEL81 Angel, I. O., *Grapische Datenverarbeitung*, Carl Hanser Verlag München Wien, 1981

BARSKY88 Barsky, Brian A., *Computer Graphics and Geometric Modeling Using Beta-splines*, Springer-Verlag Berlin Heidelberg 1988

ENCWOLF86 Encarnaçao J., Wolfang Straßer, *Computer Graphics: Gerätetchnik, Programmierung und Anwendung graphischer Systeme*, R. Oldenbourg Verlag München Wien 1986

FOLE90 Foley, James. D. ... [et al.], *Computer graphics: principles and practice*, -2nd ed., Addison-Wesley Publishing Company, 1990

GUID88 Guid, N., *Računalniška grafika*, Tehniška fakulteta v Mariboru, 1988

MORTEN85 Mortenson, Michale E., *Geometric modeling*, John Wiley & Sons, Inc., NY, 1985

ONOD90 Onodera, Tamiya, *A Formal Model of Visualizatization in Computer Graphics System*, Springer-Verlag Berlin Heidelberg, 1990

TURK87 Turk, Žiga, *Programski jezik C*, Zveza organizacij za tehnično kulturo slovenije, Ljubljana, Lepi pot 6, 1987

WATT90 Watt, Alan, *Fundaments of Three-Dimensional computer graphics*, Addison-Wesley Publishing Company, 1990

12 Dodatek

12.1 Fotografije

Nekateri rezultati obdelave, ki so bili izračunani na VAXstation 3100 in so bili fotografirani direktno z zaslona so predstavljeni v sledečih fotografijah. Čeprav je program splošne narave, lahko v okviru zmožnosti prikazuje različne kompozicije objektov, je bil uporabljen le za čajnik in nekatere enostavnejše scene. Podatki za objekt so podani v datotekah bikubičnih zlepkov, katere pa je potrebno ročno sestaviti z urejevalnikom besedil. Datoteke objektov se običajno izdeluje z 3D modelirnikom (*solid* ali *surface modeler*).

Fotografija 1 Čajnik zelene barve, osvetljen z desne strani in belo svetlobo izvora. Zadnji del čajnika je osvetljen le z ambientno svetlobo.

Fotografija 2 Rotacija, lončena barva in druga pozicija luči. Ročaj je videti skozi pokrov zato, ker so zadnje ploskve odstranjene zaradi hitrejšega računanja.

Fotografija 3 Čajnik sestavljen iz 32 Bézierjevih bikubičnih zlepkov, predstavljen z žičnim modelom in normalami v vozliščih mnogokotnikov.

Fotografija 4 Prebadanje dveh bikubičnih zlepkov

12.2 Program za senčenje

V tem delu je predstavljen celoten program, ki zgradi 3D bazo podatkov in osenči objekte. Program je pisan v C jeziku na PC računalniku v okolju jezika TURBO C^{++}. Potek programa je razumljiv že s pregledom funkcije main. Ker je program pisan v jeziku C, se bo nepoznavalcu morda zdel nerazumljiv. V tolažbo lahko povem le, da je vsak programski jezik nerazumljiv le do takrat, dokler se ga ne naučimo.

Po uspešnem preizkusu delovanja programa na PC računalniku sem s pomočjo protokola ZMODEM prekopiral izvorno kodo na računalnik VAX. Ker v laboratoriju LECAD nimamo še prevajalnika za C na VAX-u, je bilo potrebno vse te progrme preko mreže DECNET, prekopirati na univerzitetni računalnik RCU centra, ki se nahaja za Bežigradom. Na računalniku RCU so bili prevedeni vsi izvorni programi v C jeziku s prevajalnikom VAXC. Po prevajanju sem vse objektne datoteke kopiral iz RCU-ja nazaj v laboratorij LECAD, kjer sem jih z ustreznimi knjižnicami povezal v izvršni program, ki je v končni verziji tekel na delovni postaji VAXstation 3100 pod okoljem VWS pod operacijskim sistemom VMS. Za

VAXC je bilo potrebno prirediti tudi vse grafične ukaze, kar sem naredil s pogojnim prevajanjem na nivoju predprocesorja. Programe v C-ju na VAX-u nisem mogel razhroščevati, kar je povzročalo dodatne težave. Povsem možno je, da napake v programu še obstajajo, vendar pa do sedaj razen znanih napak ni bilo odkritih večjih "hroščev".

Pri razvijanju programa sem opazil nekatere nekompatibilnosti med prevajalnikoma, kar kaže na slabosti prevajalnika na VAXC.

Ugotovljene slabosti VAXC prevajalnika so:

- Predprocesor ne razume parametrov, ki so sestavljeni iz belega prostora. Primer:

```
#define ERROR(param1) printf(#param1)
    ERROR(Napaka alokacije);
```

- Nima razširjene knjižnice za dodeljevanje spomina <ALLOC.H>. Standardne funkcije za alokacijo so pri obeh prevajalnikih v <STDLIB.H>.

- #if in ostali pragmatični komentarji za preprocesor morajo biti na začetku vrstice, ka ni nujna zahteva v TURBOC^{++}

- glavni program na VAX-u (funkcija main) ne sme biti tipa *void*. Funkcija main lahko vrača *exit status*.

- knjižnica <STDLIB.H> nima vgrajenih makrojev za selekcijo dveh števil max(a,b) in min(a,b).

Celoten program za senčenje je sestavljen iz naslednjih modulov:

- R$RDDB.C bere datoteko scene in s pomočjo nje sestavi objekte, ki s zapisani v posameznih datotekah v lokalnem koordinatnem sistemu. Sproti, ko bere datoteke objektov, tudi gradi bazo podatkov in relacije med gradniki scene. Baza podatkov se gradi v spominskem bazenu računalnika (*heap memory* oz. *paged memory* na VMS sistemu).

- R$CANO.C izračuna normale površin v vozliščih in odstranjuje zadnje strani objektov. V tem modulu se nahaja tudi glavni program (*_main*).

- R$DRWF.C izriše žični model scene. Poleg izrisa ta modul vsebuje še vse funkcije, ki oprerirajo z grafiko. Tako je potrebno za implementacijo na drugo računalniško platformo prirediti le ta modul. Vgrajene ima ukaze za pogojno prevajane. Tako na prevajalniku VAXC generira drugačno kodo, kot pa prevajalnik za TURBOC. Uporabljene grafične rutine va VAX sistemu so iz GKS$ grafične knjižnice. Ta knjižnica ni povsem standardna, saj jo je možno uporabiti le na VMS ali ULTRIX operacijskih sistemih. Poleg te knjižnice obstajajo na VAX-u tudi t.i. *binding* knižnice za GKS v jeziku C, ki pa jih nisem mogel uporabiti, ker laboratorij LECAD nima licence za jezij C in s tem tudi ne za grafične knjižnice.

- R$TRDB.C transformira objekt popisan v lokalnih koordinatah v svetovne in nato celotno sceno v zaslonske koordinate izhodne naprave. Tu so vse rutine za transformacije v 3D prostoru in operacijo gledanja.

- R$PHONG.C izračuna indekse barv po Phongovi interpolacijski metodi z uporabo *z-buffer* algoritma in *LUT* korekcije barvnih indeksov. Vmesni izračun barvnih indeksov zapiše v t.i. *bitmap* datoteko, ki jo lahko kasneje uporabimo z LUT editorjem in ponovno prikažemo na ekranu.

- R$GUAR.C dela isto kot modul R$PHONG.C, le da je metoda tu Guaraudova.

- RENDER.H je t.i *header datoteka* in vsebuje skupne definicije za vse module projekta za senčenje. Tu so tudi nekatere makro funkcije in parametri celotnega programa.

Opisi posameznih spremenljivk so v izpisih programov. Vse nejasne spremenljivke imajo komentar. Običajno so spremenljivke, ki imajo za ime le eno črko, števci zanke, ali pa so enostavne spremenljivke, katerih pomen je viden že iz dela programa. Druge nedokumentirane spremenljike so izpisane s celotnim nazivom in je njihova funkcija takoj razumljiva že iz samega imena.

12.2.1 Modul za izračun normal

```
/*
    modul R$CANO.C      {CAlculate NOrmals}
    Izracuna normale poligonov in vozlisc ter izloci poligone, ki niso vidni.

    Projekt     RENDER
    Verzija     0.0
    Datum       13.5.1991
    Avtor       Leon Kos
    Jezik       TURBO C++ 1.0 ali VAXC 3.0
*/
#define MAIN

#include <math.h>
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include "render.h"

#if defined(__TURBOC__) && defined(MAIN)
extern unsigned _stklen = 20000;
/* za _stklen=10000 / 6 dobimo 1700 vozlov in povrsin */
#endif

/* Izracuna vektorski produkt dveh vektorjev podanih s tremi vektorji */
int calculate_normal(VECTOR wp1, VECTOR wp2, VECTOR wp3, VECTOR *normal)
{
  VECTOR a,b;
  a.x=wp2.x-wp1.x; b.x=wp3.x-wp2.x;
  a.y=wp2.y-wp1.y; b.y=wp3.y-wp2.y;
  a.z=wp2.z-wp1.z; b.z=wp3.z-wp2.z;
  normal->x=a.y*b.z-a.z*b.y;
  normal->y=a.z*b.x-a.x*b.z;
  normal->z=a.x*b.y-a.y*b.x;
  if(fabs(normal->x)+fabs(normal->y)+fabs(normal->z)>ALMOST_ZERO)
```

```
    return 1; /*vrne 1 ce je uspesno izracunal normalo mnogokotnika */
  else
    return 0;
}


void calculate_polygon_normal(POLYGON *polygon)
{
  int success=0;
  VERTEX_LIST *vertices=polygon->vertices;

  while(!success){
    if(vertices->rest->rest==NULL)
      ERROR("Ne morem izracunati normale poligona");
    success=calculate_normal(vertices->vertex->world_position,
                         vertices->rest->vertex->world_position,
                         vertices->rest->rest->vertex->world_position,
                         &polygon->normal);
    normalise(polygon->normal, &polygon->normal);
    vertices=vertices->rest;
  }
}

void calculate_polygon_normals(SCENE *scene)
{
  OBJECT *object=scene->object_head;
  SURFACE *surface;
  POLYGON *polygon;

  while(object){
    surface=object->surface_head;
    while(surface){
      polygon=surface->polygons;
      while(polygon) {
        calculate_polygon_normal(polygon);
        polygon=polygon->next;
      }
      surface=surface->next;
    }
    object=object->next;
  }
}


void calculate_vertex_normal(VERTEX_NORMAL *normal_cell)
{
  VECTOR totalvec={0.0, 0.0, 0.0};
  POLYGON_LIST *polygons=normal_cell->polygons;
  while(polygons){
    totalvec.x += polygons->polygon->normal.x;
    totalvec.y += polygons->polygon->normal.y;
    totalvec.z += polygons->polygon->normal.z;
    polygons=polygons->rest;
  }
  normalise(totalvec, &normal_cell->normal);
}

void calculate_vertex_normals(SCENE *scene)
{
  OBJECT *object=scene->object_head;
```

```
  VERTEX *vertex;
  VERTEX_NORMAL *normal_cell;

  while(object){
    vertex=object->vertex_head;
    while(vertex){
      normal_cell=vertex->normals;
      while(normal_cell){
        calculate_vertex_normal(normal_cell);
        normal_cell=normal_cell->next;
      }
      vertex=vertex->next;
    }
    object=object->next;
  }
}


void cull_polygons(SCENE *scene)
{
  OBJECT *object=scene->object_head;
  SURFACE *surface;
  POLYGON *polygon;
  VECTOR line_of_sight, view_point;

  normalise(scene->view_plane_normal, &scene->view_plane_normal);


  normalise(scene->view_plane_normal, &scene->view_plane_normal);
  view_point.x=scene->view_reference_point.x+
               scene->view_plane_normal.x*scene->projection_distance;
  view_point.y=scene->view_reference_point.y+
               scene->view_plane_normal.y*scene->projection_distance;
  view_point.z=scene->view_reference_point.z+
               scene->view_plane_normal.z*scene->projection_distance;

  while(object){
    surface=object->surface_head;
    while(surface){
      polygon=surface->polygons;
      while(polygon){
        if(scene->projection_type==PARALLEL){
          polygon->culled=
            dot_product(polygon->normal, scene->view_plane_normal)<0.0;
        }
        else {
          line_of_sight.x=view_point.x-
                polygon->vertices->vertex->world_position.x;
          line_of_sight.y=view_point.y-
                polygon->vertices->vertex->world_position.y;
          line_of_sight.z=view_point.z-
                polygon->vertices->vertex->world_position.z;
          polygon->culled=dot_product(polygon->normal, line_of_sight) <0.0;
        }
        polygon=polygon->next;
      }
      surface=surface->next;
    }
    object=object->next;
  }
}
```

```
#ifdef MAIN
main()
{
  int num_colors=16;
  SCENE *scene;
  scene=read_scene("SCENE1.DAT");
  transform_scene_to_world(scene);
  initialisation();
  calculate_device_coords(scene);
  calculate_polygon_normals(scene);
  calculate_vertex_normals(scene);
  cull_polygons(scene);
  draw_scene(scene);
  switch(scene->draw_operation){
    case PHONG :
      set_mono_palette(&num_colors);
      render_scene_phong(scene, scene->viewport.raster_width,
         scene->viewport.raster_height, num_colors,
         scene->shading_output_filename);
      show_bitmap_picture(scene->shading_output_filename);
      break;
    case GUARARD:
      set_mono_palette(&num_colors);
      render_scene_guarard(scene, scene->viewport.raster_width,
         scene->viewport.raster_height, num_colors,
         scene->shading_output_filename);
      show_bitmap_picture(scene->shading_output_filename);
      break;
  }
  terminate(1);

#ifdef __TURBOC__
  return 0;
#endif
}
#endif
```

12.2.2 Modul za branje 3D baze podatkov

```
/*
    modul R$RDDB.C      {Read Database}
    bere 3D bazo podatkov iz datoteke scene in datotek objektov.

    Projekt     RENDER
    Verzija     0.01
    Datum       4.5.1991
    Popravek    28.5.1991
    Avtor       Leon Kos
    Jezik       TURBO C++ 1.0 ali VAXC 3.0
*/

/* stikala pogojnega prevajanja modula */
#define LIST_FUNCTINOS            /* vkljucitev funkcij za izpis baze */
#define REMOVE_VOID_NORMAL       /* izlocanje neuporabljenih normal  */
#undef  MAIN                    /* vkljucitev main funkcije         */

#ifdef __TURBOC__
#include <alloc.h>
```

```
#endif


#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include "render.h"

static FILE *objf;              /* tekoca datoteka podatkov o objektu */
static char buffer[160];        /* medpomnilnik pri branju iz datoteke */

/* zacasne tabele pri gradnji baze */
static VERTEX   **vertex_map_array;
static POLYGON  **polygon_map_array;

/*bere eno vrstico iz datoteke objf v buffer*/
#define READBUF(file) if(fgets(buffer, sizeof(buffer), file) == NULL) \
  ERROR("Branje cele vrstice iz datoteke")


/* bere podatke iz datoteke in rekurzivno kreira seznam vozlisc */
VERTEX *read_vertices(int no_of_vertices)
{
  if(no_of_vertices==0)
    return NULL;
  else {
    static int n=0, count;
    VERTEX *current_vertex, *vertex_next, *vertex_head;
    float a;

    for(count=0; count<no_of_vertices; count++){
      if((vertex_next=malloc(sizeof(VERTEX))) == NULL)
        ERROR("Alokacija vozlisca");
      READBUF(objf);
      if(sscanf(buffer, "%d %f %f %f", &n,&vertex_next->local_position.x,
                        &vertex_next->local_position.y,
                        &vertex_next->local_position.z, &a) != 4)
      fprintf(stderr, "Napaka pri branju podatkov vozlisca #%d\n", n);
      vertex_next->normals=NULL;
      vertex_map_array[n]=vertex_next;
      vertex_next->next=NULL;
      if(n==0) {
        vertex_head=vertex_next;
        current_vertex=vertex_head;
      }
      else{
        current_vertex->next=vertex_next;
        current_vertex=vertex_next;
      }
    }
    return vertex_head;
  }
}

static char *white_space="\t\n ";

POLYGON_LIST *add_polygon_to_vertex_normal(POLYGON *polygon,
                POLYGON_LIST *polygon_list)
{
  if(polygon_list==NULL){
    if((polygon_list=malloc(sizeof(POLYGON_LIST)))==NULL)
```

```
      ERROR("Alokacija seznama normale vozlisca");
    polygon_list->rest=NULL;
    polygon_list->polygon=polygon;
    return polygon_list;
  }
  else {
    polygon_list->rest=add_polygon_to_vertex_normal(polygon,
                                                    polygon_list->rest);
    return polygon_list;
  }
}


VERTEX_LIST *create_vertex_list(POLYGON *polygon,
                int no_of_poly_vertices, char *spos)
{
  if(no_of_poly_vertices==0)
    return NULL;
  else {
    static int vno;
    VERTEX_LIST *vertex_list;
    VERTEX_NORMAL *normal;

    if((vertex_list=malloc(sizeof(VERTEX_LIST))) == NULL)
      ERROR("Alokacija seznama vozlisc mnogokotnika");
    if(sscanf(spos, "%d", &vno) !=1)
      ERROR("Branje vozlisca poligona");
    vertex_list->vertex=vertex_map_array[vno];

    /* v seznam zadnje normale vozlisca vstavimo tekoc poligon */
    normal=vertex_list->vertex->normals;
    while(normal->next) normal=normal->next;
    normal->polygons=add_polygon_to_vertex_normal(polygon,normal->polygons);

    vertex_list->normal=normal;

    vertex_list->rest=create_vertex_list(polygon, no_of_poly_vertices-1,
                                strtok(NULL, white_space));
    return vertex_list;
  }
}


static int last_surface, no_of_polygons, current_polygon;

/* bere podatke iz datoteke in rekurzivno kreira seznam poligonov */
POLYGON *read_polygons(char new_surface)
{
  static int  surface_no;
  static int no_of_poly_vertices;

  if(no_of_polygons==0)
    return NULL;

  if(!new_surface || !current_polygon) {
    if(fscanf(objf, "%d%d%d", &current_polygon, &surface_no,
                &no_of_poly_vertices) != 3){
      fprintf(stderr, "Napaka pri branju podatkov poligona #%d\n",
                current_polygon);
      ERROR("Branje poligonov iz datoteke");
    }
    READBUF(objf);
```

```
  }

  if(surface_no != last_surface) {
    last_surface=surface_no;
    return NULL;
  }
  else {
    POLYGON *polygon;
    if((polygon=malloc(sizeof(POLYGON))) == NULL)
      ERROR("Alokacija mnigokotnika");
    polygon->vertices=create_vertex_list(polygon, no_of_poly_vertices,
                                         strtok(buffer,white_space));
    polygon_map_array[current_polygon]=polygon;
    no_of_polygons--;
    polygon->next=read_polygons(0);
    return polygon;
  }
}

VERTEX_NORMAL *add_vertex_normal(VERTEX_NORMAL *head)
{
  if(head==NULL){
    if((head=malloc(sizeof(VERTEX_NORMAL)))==NULL)
      ERROR("Alokacija normale vozlisca");
    head->next=NULL;
    head->polygons=NULL;
    return head;
  }
  else {
    head->next=add_vertex_normal(head->next);
    return head;
  }
}

/* odstrani zadnjo normalo vozlisca v seznamu, ce ne vsebuje poligonov*/
VERTEX_NORMAL *remove_vertex_normal(VERTEX_NORMAL *head)
{
  if(head->next==NULL){
    if(head->polygons==NULL){
      free(head);
      return NULL;
    }
    else
      return head;
  }
  else {
    head->next=remove_vertex_normal(head->next);
    return head;
  }
}

/* bere podatke iz datoteke in rekurzivno kreira seznam povrsin */
SURFACE *read_surfaces(int no_of_surfaces, VERTEX *vertex_head)
{
  if(no_of_surfaces==0)
    return NULL;
  else {
    SURFACE *surface;
    static VERTEX *vertex;

    vertex=vertex_head;
    while(vertex){ /* dodamo po eno normalo vsakemu vozliscu */
```

```
      vertex->normals=add_vertex_normal(vertex->normals);
      vertex=vertex->next;
    }

    if((surface=malloc(sizeof(SURFACE)))==NULL) ERROR("Alokacija povrsine");

    surface->polygons=read_polygons(1);

    /* pocistimo vse neuporabljene normale vozlisc */
#ifdef REMOVE_VOID_NORMAL
    vertex=vertex_head;
    while(vertex){
       vertex->normals=remove_vertex_normal(vertex->normals);
       vertex=vertex->next;
    }
#endif
    surface->next=read_surfaces(no_of_surfaces-1, vertex_head);
    return surface;
  }
}



OBJECT *read_object(const char *object_filename)
{
  OBJECT *object;

  if((objf=fopen(object_filename, "r"))==NULL){
    perror(object_filename);
    exit(errno);
  }
  if((object=malloc(sizeof(OBJECT)))==NULL) ERROR("Alokacija objekta");

  READBUF(objf);
  if(sscanf(buffer, "%d %d %d", &object->no_of_vertices,
                                &object->no_of_polygons,
                                &object->no_of_surfaces) != 3)
        ERROR("Branje prve vrstice datoteke");

  no_of_polygons=object->no_of_polygons;

  vertex_map_array=malloc(sizeof(VERTEX *)*object->no_of_vertices);
  if(!vertex_map_array) ERROR("Alokacija tabele vozlisc");
  polygon_map_array=malloc(sizeof(POLYGON *)*object->no_of_polygons);
  if(!vertex_map_array) ERROR("Alokacija tabele poligonov");

  object->vertex_head=read_vertices(object->no_of_vertices);

  last_surface=current_polygon=0;
  object->surface_head=read_surfaces(object->no_of_surfaces,
               object->vertex_head);

  free(vertex_map_array);
  free(polygon_map_array);
  fclose(objf);

  return object;
}

OBJECT *read_objects(FILE *scf, int no_of_objects)
{
  if(no_of_objects==0)
```

```
    return NULL;
  else {
    OBJECT *object;
    fscanf(scf,"%s",buffer);
    object=read_object(buffer);
    READBUF(scf);       /*preberemo do konca vrstice*/
    READBUF(scf);
    if(sscanf(buffer,"%f%f%f",
                &object->scale.x,
                &object->scale.y,
                &object->scale.z) != 3)
        ERROR("Branje skaliranja objekta iz datoteke scene");
    READBUF(scf);
    if(sscanf(buffer,"%f%f%f",
                &object->rotate.x,
                &object->rotate.y,
                &object->rotate.z) != 3)
        ERROR("Branje rotacije objekta iz datoteke scene");
    READBUF(scf);
    if(sscanf(buffer,"%f%f%f",
                &object->translate.x,
                &object->translate.y,
                &object->translate.z) != 3)
        ERROR("Branje translacije objekta iz datoteke scene");

    object->next=read_objects(scf, no_of_objects-1);
    return object;
  }
}



SCENE *read_scene(char *scene_filename)
{
  FILE *scf;
  SCENE *scene;
  int no_of_objects;

  if((scf=fopen(scene_filename, "r"))==NULL){
    perror(scene_filename);
    exit(errno);
  }

  if((scene=malloc(sizeof(SCENE)))==NULL) ERROR("Alokacija scene");



  READBUF(scf);
  if(sscanf(buffer,"%d",&scene->projection_type) != 1)
    ERROR("Branje tipa projekcije iz datoteke scene");

  READBUF(scf);
  if(sscanf(buffer,"%f",&scene->view_plane_distance) != 1)
    ERROR("Branje razdalje vidne ravnine iz datoteke scene");

  READBUF(scf);
  if(sscanf(buffer,"%f",&scene->projection_distance) != 1)
    ERROR("Branje razdalje vidne ravnine iz datoteke scene");

  READBUF(scf);
  if(sscanf(buffer,"%f%f%f",
```

```
    &scene->view_reference_point.x,
    &scene->view_reference_point.y,
    &scene->view_reference_point.z) != 3)
      ERROR("Branje vektorja izhodisca (VRP) iz datoteke scene");

  READBUF(scf);
  if(sscanf(buffer,"%f%f%f",
    &scene->view_plane_normal.x,
    &scene->view_plane_normal.y,
    &scene->view_plane_normal.z) != 3)
      ERROR("Branje normale ravnine pogleda (VPN) iz datoteke scene");

  READBUF(scf);
  if(sscanf(buffer,"%f%f%f",
    &scene->view_up_vector.x,
    &scene->view_up_vector.y,
    &scene->view_up_vector.z) != 3)
     ERROR("Branje vektorja zavrtitve pogleda (VUP) iz datoteke scene");

  READBUF(scf);
  if(sscanf(buffer,"%f%f",
    &scene->viewport.width, &scene->viewport.height) != 2)
     ERROR("Branje sirine in visine okna iz datoteke scene");

  READBUF(scf);
  if(sscanf(buffer,"%d", &scene->viewport.raster_height) != 1)
     ERROR("Branje stevila rasterskih enot visine okna");

  READBUF(scf);
  if(sscanf(buffer,"%d",&scene->draw_operation) != 1)
    ERROR("Branje tipa operacije iz datoteke scene");

  switch(scene->draw_operation){
    case PHONG:
    case GUARARD:
       READBUF(scf);
       if(sscanf(buffer,"%f%f%f",
         &scene->light_position.x,
         &scene->light_position.y,
         &scene->light_position.z) != 3)
          ERROR("Branje vektorja luci iz datoteke scene");
       if(fscanf(scf, "%s", &scene->shading_output_filename) != 1)
         ERROR("Branje imena datoteke phongovega sencenja");
       READBUF(scf);
       break;
    default:
      READBUF(scf);
      READBUF(scf);
  }


  READBUF(scf);
  if(sscanf(buffer,"%d",&no_of_objects) != 1)
    ERROR("Branje stevila objektov iz datoteke scene");

  scene->object_head=read_objects(scf, no_of_objects);

  fclose(scf);
  return scene;
}
```

```
#ifdef LIST_FUNCTINOS
void list_vertex_normals(VERTEX_NORMAL *head)
{
  int n=0, pno;
  while(head){
    POLYGON_LIST *polygons=head->polygons;
    for(pno=0;polygons;pno++) polygons=polygons->rest;
    printf("\tNormala #%d (x:%5.3g y:%5.3g: z:%5.3g)\n\
\t\tvsebuje %d mnogokotnik%s na lokacij%s:\n\t\t", n++,
head->normal.x, head->normal.y, head->normal.z, pno,
(pno==1) ? "":"ov", (pno==1) ? "i":"ah");
    for(pno=0, polygons=head->polygons; polygons; pno++) {
      printf("%p\t", polygons->polygon);
      polygons=polygons->rest;
    }
    printf("\n");
    head=head->next;
  }
}

void list_vertices(VERTEX *head)
{
  int n=0;
  while(head) {
    printf("Vozel #%d: na lokaciji %p\n\
\tLokalna pozicija:\tx:%10.3g y:%10.3g z:%10.3g\n\
\tSvetovna:\t\tx:%10.3g y:%10.3g z:%10.3g\n\
\tZaslonska:\t\tx:%10.3g y:%10.3g z:%10.3g\n", n++, head,
head->local_position.x, head->local_position.y, head->local_position.z,
head->world_position.x, head->world_position.y, head->world_position.z,
head->screen_position.x, head->screen_position.y, head->screen_position.z);
    list_vertex_normals(head->normals);
    head=head->next;
  }
}

void list_polygons(POLYGON *head)
{
  int n=0, vno;
  while(head){
    VERTEX_LIST *vertices=head->vertices;
    for(vno=0;vertices;vno++) vertices=vertices->rest;
    printf("%d-kotnik #%d na lokaciji %p: %s\n\t\
normala(x:%5.3g y:%5.3g z:%5.3g)\n",
       vno, n++, head, head->culled ? "Izlocen":"\t",
       head->normal.x, head->normal.y, head->normal.z);
    vertices=head->vertices;
    for(vno=0;vertices;vno++) {
      printf("\tVozlisce #%2d na lokaciji %p\n", vno, vertices->vertex);
      vertices=vertices->rest;
    }
    head=head->next;
  }
}

void list_surfaces(SURFACE *head)
{
  int n=0;
  while(head){
    printf("\nPovrsina #%d ima %d mnogokotnikov.\n",n++, no_of_polygons);
    list_polygons(head->polygons);
    head=head->next;
```

```
  }
}

void list_objects(OBJECT *head)
{
  int n=0;
  while(head){
    printf("\nOBJEKT #%d:\n",n++);
    list_vertices(head->vertex_head);
    list_surfaces(head->surface_head);
    head=head->next;
  }
}



#endif


#ifdef MAIN
main()
{
  SCENE *scene;

#ifdef __TURBOC__
  printf("Velikost sklada je %u, velikost bazena pa %lu\n",
         _stklen, (long unsigned) coreleft());
#endif

  scene=read_scene("SCENE1.DAT");

#ifdef LIST_FUNCTINOS
  list_objects(scene->object_head);
#endif

#ifdef __TURBOC__
  printf("Velikost sklada je %u, velikost bazena pa %lu\n",
         _stklen, (long unsigned) coreleft());
#endif
 return 0;
}
#endif
```

12.2.3 Modul za 3D transformacije

```
/*
    modul R$TRDB.C      {Transform Database}
    transformira 3D bazo podatkov v svetovne in zaslonske koordinate.

    Projekt     RENDER
    Verzija     0.0
    Datum       4.5.1991
    Avtor       Leon Kos
    Jezik       TURBO C++ 1.0 ali VAXC 3.0
*/
#undef  MAIN
#define CANONICAL_VIEW
#define READ_DATABASE
#undef DEBUG

#include <math.h>
```

```
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <limits.h>
#include "render.h"

int **imatrix(int nrl, int nrh, int ncl, int nch)
{
  int i;
  int **m;

  m=(int **)malloc((unsigned) (nrh-nrl+1)*sizeof(int*));
  if(!m) ERROR("alokacijska napaka 1 v imatrix()");
  m -= nrl;

  for (i=nrl;i<=nrh;i++){
    m[i]=(int *)malloc((unsigned) (nch-ncl+1)*sizeof(int));
    if(!m[i]) ERROR("alokacijska napaka 2 v imatrix()");
    m[i] -= ncl;
  }
  return m;
}

void free_imatrix(int **m, int nrl,  int nrh, int ncl, int nch)
{
  int i;
  for(i=nrh;i>=nrl;i--)  free((void*) (m[i]+ncl));
  free((void*) (m+nrl));
}


float magnitude(VECTOR a){
  return sqrt(sqr(a.x)+sqr(a.y)+sqr(a.z));
}

void normalise(VECTOR a, VECTOR *b)
{
  float r=magnitude(a);
  if(r) {
    b->x=a.x/r;
    b->y=a.y/r;
    b->z=a.z/r;
    return;
  }
  else {
    b->x=a.x;
    b->y=a.y;
    b->z=a.z;
    return;
  }
}

int vec_prod(VECTOR a, VECTOR b, VECTOR *c)
{
  c->x=a.y*b.z-a.z*b.y;
  c->y=a.z*b.x-a.x*b.z;
  c->z=a.x*b.y-a.y*b.x;
  if(fabs(c->x)+fabs(c->y)+fabs(c->z) > ALMOST_ZERO)
    return 0;
  else
    return -1;
}
```

```
void return_identity_matrix(HMATRIX m)
{
  int i,j;
  for(i=0;i<4;i++)
    for(j=0;j<4;j++)
      if(i==j) m[i][j]=1.0;
      else m[i][j]=0.0;
}

void return_zero_matrix(HMATRIX m)
{
  int i,j;
  for(i=0;i<4;i++)
    for(j=0;j<4;j++)
      m[i][j]=0.0;
}

void matrix_mult(HMATRIX m1, HMATRIX m2, HMATRIX m)
{
  int i,j,element;
  return_zero_matrix(m);
  for(i=0; i<4; i++)
    for(j=0; j<4; j++)
      for(element=0; element<4; element++)
        m[i][j] += m1[i][element]*m2[element][j];
}

void matrix_vector(HMATRIX m, VECTOR v, VECTOR *mv)
{
  mv->x=m[0][0]*v.x+m[0][1]*v.y+m[0][2]*v.z+m[0][3];
  mv->y=m[1][0]*v.x+m[1][1]*v.y+m[1][2]*v.z+m[1][3];
  mv->z=m[2][0]*v.x+m[2][1]*v.y+m[2][2]*v.z+m[2][3];
}

void matrix_hvector(HMATRIX m, VECTOR v, HVECTOR *mv)
{
  mv->x=m[0][0]*v.x+m[0][1]*v.y+m[0][2]*v.z+m[0][3];
  mv->y=m[1][0]*v.x+m[1][1]*v.y+m[1][2]*v.z+m[1][3];
  mv->z=m[2][0]*v.x+m[2][1]*v.y+m[2][2]*v.z+m[2][3];
  mv->w=m[3][0]*v.x+m[3][1]*v.y+m[3][2]*v.z+m[3][3];
}

void matrix_hhvector(HMATRIX m, HVECTOR v, HVECTOR *mv)
{
  mv->x=m[0][0]*v.x+m[0][1]*v.y+m[0][2]*v.z+m[0][3];
  mv->y=m[1][0]*v.x+m[1][1]*v.y+m[1][2]*v.z+m[1][3];
  mv->z=m[2][0]*v.x+m[2][1]*v.y+m[2][2]*v.z+m[2][3];
  mv->w=m[3][0]*v.x+m[3][1]*v.y+m[3][2]*v.z+m[3][3];
}


void write_matrix(HMATRIX m)
{
  int i,j;
  for(i=0; i<4; i++){
    for(j=0; j<4; j++)
      printf("%10.3f",m[i][j]);
     printf("\n");
    }
  printf("\n");
}
```

```
void return_scale_matrix(HMATRIX m, float x, float y, float z)
{
  return_identity_matrix(m);
  m[0][0]=x;
  m[1][1]=y;
  m[2][2]=z;
}

void return_rotate_matrix(HMATRIX m, float x, float y, float z)
{
  HMATRIX mx, my, mz, mm;

  return_identity_matrix(m);
  x=rad(x);
  y=rad(y);
  z=rad(z);
  return_identity_matrix(mz);
  mz[0][0]= cos(z);  mz[0][1]= -sin(z);   /* desnosucni koordinatni sistem */
  mz[1][0]= sin(z);  mz[1][1]= cos(z);
  return_identity_matrix(my);
  my[0][0]= cos(y);  my[0][2]= sin(y);
  my[2][0]= -sin(y); my[2][2]= cos(y);
  return_identity_matrix(mx);
  mx[1][1]= cos(x);  mx[1][2]= -sin(x);
  mx[2][1]= sin(x);  mx[2][2]= cos(x);
  matrix_mult(my,mx,mm);
  matrix_mult(mz,mm,m);
}

void return_translate_matrix(HMATRIX m, float x, float y, float z)
{
  return_identity_matrix(m);
  m[0][3]=x;
  m[1][3]=y;
  m[2][3]=z;
}



void return_view_orientation_matrix(HMATRIX view_orientation_matrix,
                VECTOR view_reference_point,
                VECTOR view_plane_normal,
                VECTOR view_up_vector)
{
  HMATRIX mt, r;
  VECTOR rx, ry, rz;

  return_translate_matrix(mt, -view_reference_point.x,
                              -view_reference_point.y,
                              -view_reference_point.z);
  normalise(view_plane_normal, &rz);
  vec_prod(view_up_vector, rz, &rx);
  normalise(rx, &rx);
  vec_prod(rz, rx, &ry);
  return_identity_matrix(r);
  r[0][0]=rx.x; r[0][1]=rx.y; r[0][2]=rx.z;
  r[1][0]=ry.x; r[1][1]=ry.y; r[1][2]=ry.z;
  r[2][0]=rz.x; r[2][1]=rz.y; r[2][2]=rz.z;
  matrix_mult(r, mt, view_orientation_matrix);
}
```

```
void return_view_mapping_matrix(HMATRIX view_mapping_matrix,
                                int projection_type,
                                VECTOR projection_reference_point,
                                float umin, float umax,
                                float vmin, float vmax,
                                float front_plane, float back_plane)
{
  VECTOR direction_of_projection, center_of_window;
  HMATRIX Spar, Tpar, SHpar, temp_m;

  center_of_window.x=(umax+umin)/2.0;
  center_of_window.y=(vmax+vmin)/2.0;
  center_of_window.z=0.0;

  direction_of_projection.x=center_of_window.x-projection_reference_point.x;
  direction_of_projection.y=center_of_window.y-projection_reference_point.y;
  direction_of_projection.z=center_of_window.z-projection_reference_point.z;

  return_identity_matrix(SHpar);
#ifdef CANONICAL_VIEW
  SHpar[0][2]= -direction_of_projection.x/direction_of_projection.z;
  SHpar[1][2]= -direction_of_projection.y/direction_of_projection.z;
#endif
  if(projection_type==PARALLEL) {

    return_translate_matrix(Tpar, -center_of_window.x,
                                -center_of_window.y, -front_plane);
    return_scale_matrix(Spar, 2.0/(umax-umin), 2.0/(vmax-vmin),
                              1.0/(front_plane-back_plane));
    matrix_mult(Tpar, SHpar, temp_m);
    matrix_mult(Spar, temp_m, view_mapping_matrix);
  }
  else {
    VECTOR vrp, zero;
    HMATRIX trm, temp_m, temp_m2, Sper, Mclip;
    float zmin;

    return_translate_matrix(trm, -projection_reference_point.x,
                                 -projection_reference_point.y,
                                 -projection_reference_point.z);
    matrix_mult(SHpar, trm, temp_m);
    zero.x=zero.y=zero.z=0.0;

    matrix_vector(temp_m, zero, &vrp);

    return_scale_matrix(Sper, 2.0*vrp.z/((umax-umin)*(vrp.z+back_plane)),
                              2.0*vrp.z/((vmax-vmin)*(vrp.z+back_plane)),
                              -1.0/(vrp.z+back_plane));

    zmin= -(vrp.z+front_plane)/(vrp.z+back_plane);
    return_identity_matrix(Mclip);
    Mclip[2][2] /= 1+zmin;
    Mclip[2][3]  = -zmin/(1+zmin);
    Mclip[3][2] = -1;

    matrix_mult(Sper, temp_m, temp_m2);
    matrix_mult(Mclip, temp_m2, view_mapping_matrix);
  }
}


void return_viewport_mapping_matrix(HMATRIX viewport_mapping_matrix,
```

```
        int xvmin, int xvmax, int yvmin, int yvmax, int zvmin, int zvmax)
{
  HMATRIX s, t, t1, temp_m;

  return_translate_matrix(t, (float)xvmin, (float)yvmin, (float)zvmin);
  return_scale_matrix(s, (float)(xvmax-xvmin)/2.0, (float)(yvmax-yvmin)/2.0,
                      (float)(zvmax-zvmin));
   return_translate_matrix(t1, 1.0, 1.0, 0.0);
  matrix_mult(s, t1, temp_m);
  matrix_mult(t, temp_m, viewport_mapping_matrix);
}


void transform_object_to_world(OBJECT *object)
{
   HMATRIX ms, mr, mt, m, mm;
   VERTEX *vertex=object->vertex_head;

   return_scale_matrix(ms, object->scale.x, object->scale.y, object->scale.z);
   return_rotate_matrix(mr, object->rotate.x, object->rotate.y,
                        object->rotate.z);
   return_translate_matrix(mt, object->translate.x, object->translate.y,
                        object->translate.z);

   matrix_mult(ms, mr, mm);
   matrix_mult(mt, mm, m);

   while(vertex != NULL){
     matrix_vector(m, vertex->local_position, &vertex->world_position);
     vertex=vertex->next;
   }
}

void transform_scene_to_world(SCENE *scene)
{
  OBJECT *object=scene->object_head;

  while(object != NULL){
    transform_object_to_world(object);
    object=object->next;
  }
}



void calculate_device_coords(SCENE *scene)
{
  HMATRIX view_orientation_matrix, view_mapping_matrix,
                viewport_mapping_matrix, view_matrix;
  int xvmin, xvmax, yvmin, yvmax, zvmin = INT_MIN, zvmax = 0;
  OBJECT *object;
  VERTEX *vertex;

  return_view_orientation_matrix(view_orientation_matrix,
                scene->view_reference_point,
                scene->view_plane_normal,
                scene->view_up_vector);

  scene->projection_reference_point.x=0.0;
  scene->projection_reference_point.y=0.0;
  scene->projection_reference_point.z=scene->projection_distance;
```

```
  scene->viewport.umin= -scene->viewport.width/2.0;
  scene->viewport.umax=  scene->viewport.width/2.0;
  scene->viewport.vmin= -scene->viewport.height/2.0;
  scene->viewport.vmax=  scene->viewport.height/2.0;
  scene->viewport.front_plane=0.0;
  scene->viewport.back_plane=scene->view_plane_distance;


  return_view_mapping_matrix(view_mapping_matrix,
                scene->projection_type,
                scene->projection_reference_point,
                scene->viewport.umin, scene->viewport.umax,
                scene->viewport.vmin, scene->viewport.vmax,
                scene->viewport.front_plane, scene->viewport.back_plane);

  return_max_viewport(&xvmin, &xvmax, &yvmin, &yvmax);
  if(scene->viewport.raster_height <= max(yvmin, yvmax)+1){
    scene->viewport.raster_width=
      (int)((float)scene->viewport.raster_height*
        scene->viewport.width/scene->viewport.height);
    set_device_viewport(scene->viewport.raster_height,
                        scene->viewport.raster_width,
                        &xvmin, &xvmax, &yvmin, &yvmax);
  }
  else
     ERROR("Ne morem dobiti zelene rezolucije zaslona");

  return_viewport_mapping_matrix(viewport_mapping_matrix,
                xvmin, xvmax, yvmin, yvmax, zvmin, zvmax);

  matrix_mult(view_mapping_matrix, view_orientation_matrix, view_matrix);
  matrix_mult(viewport_mapping_matrix, view_matrix, scene->device_matrix);

  object=scene->object_head;
  while(object){
    vertex=object->vertex_head;
    while(vertex){
      matrix_hvector(scene->device_matrix, vertex->world_position,
                        &vertex->screen_position);
      if(vertex->screen_position.w != 1.0){
        vertex->screen_position.x /= vertex->screen_position.w;
        vertex->screen_position.y /= vertex->screen_position.w;
        vertex->screen_position.z /= vertex->screen_position.w;
        vertex->screen_position.w= 1.0;
      }
      vertex=vertex->next;
    }
    object=object->next;
  }
#ifdef DEBUG
  list_objects(scene->object_head);
#endif
}



#ifdef MAIN
main()
{
#ifdef READ_DATABASE
  SCENE *scene;
```

```
  scene=read_scene("SCENE1.DAT");
  transform_scene_to_world(scene);
  initialisation();
  calculate_device_coords(scene);
/*  list_objects(scene->object_head); */
  draw_scene(scene);



#else
  HMATRIX a,b,c;
  VECTOR va={5,6,7};

  return_translate_matrix(a,2,3,4);
  matrix_vector(a,va,&va);
  write_matrix(a);
  printf("%10.3f %10.3f %10.3f\n", va.x, va.y, va.z);
  normalise(va,&va);
#endif

  return 0;
}
#endif
```

12.2.4 Modul za izris na ekran

```
/*
    modul R$DRWF.C      {DRaw WireFrame}
    transformira 3D bazo podatkov v svetovne in zaslonske koordinate.
    Tu so shranjeni tudi vsi podprogrami, ki operirajo z grafiko.
    Za prilagoditev na drug graficen sistem je potrebno prirediti le
    ta modul.

    Projekt     RENDER
    Verzija     0.0
    Datum       13.5.1991
    Avtor       Leon Kos
    Jezik       TURBO C++ 1.0 ali VAXC 3.0
*/

#undef  DRAW_CULLED_POLYGONS /* Ali naj se izrisejo tudi izloceni poligoni */
#define NORMAL_DRAW_METHOD 2 /* 1 || 2 metoda izrisa vozlisc poligona */



#include <math.h>
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include "render.h"

#ifdef __TURBOC__
#include <graphics.h>
#include <dos.h>
#endif

#ifdef VAXC
#define GKS_DOLLAR
#endif
```

```
#ifdef GKS_DOLLAR
#include <gksdefs.h>
#include <descrip.h>

#define TNR 1           /* ena sama transformacija s stevilko TNR */
#define MAX_POLY_VERTICES 20 /*maksimalno stevilo vozlisc za gks$polyline */
#define MAX_STRING 80   /* najvecja dolzina stringa */

struct dsc$descriptor_a2 {
  unsigned short dsc$w_length;
  unsigned char dsc$b_type;
  unsigned char dsc$b_class;
  char *dsc$a_pointer;
  char dsc$b_scale;

  unsigned char dsc$b_digits;
  struct {
    unsigned                    :4;
    unsigned dsc$v_fl_redim     :1;
    unsigned dsc$v_fl_column    :1;
    unsigned dsc$v_fl_coeff     :1;
    unsigned dsc$v_fl_bounds    :1;
  } dsc$b_aflag;

  unsigned char dsc$b_dimct;
  unsigned long dsc$l_arsize;

  char *dsc$a_a0;
  long dsc$l_m[2];
  struct {
    long dsc$l_l;
    long dsc$l_u;
  }dsc$bounds[2];
};

#define DESC_ARRAY(name, length, ptr) struct dsc$descriptor_a        \
  name={4, DSC$K_DTYPE_L, DSC$K_CLASS_A, ptr, 0, 0, {0, 0, 0, 0, 0}, \
        1, length * 4}

#define DESC_ARRAY_2(name, dim1, dim2, ptr) struct dsc$descriptor_a2 \
  name={4, DSC$K_DTYPE_L, DSC$K_CLASS_A, ptr, 0, 0, {0, 0, 0, 1, 1}, \
        2, dim1 * dim2 * 4, ptr, {dim1, dim2}, {0, dim1 -1, 0, dim2-1}}


int  ws_id=1;   /* stevilka delovne postaje */
int ws_type;    /* tip delovne postaje */


setup_ws(int ws_id)
{
  int   error_status,
        category,
        inquire_okay,
        dummy_integer,
        def_mode,
        regen_flag;

  struct dsc$descriptor dummy_dsc;
  char dummy_string[MAX_STRING];

  $DESCRIPTOR(error_file, "sys$error:");
```

```
  /* Initialize variables */
  inquire_okay=0;
  dummy_dsc.dsc$a_pointer=dummy_string;
  dummy_dsc.dsc$w_length=(short) MAX_STRING;

  gks$open_gks(&error_file);
  gks$inq_ws_category(&GKS$K_WSTYPE_DEFAULT, &error_status, &category);

  /* Make sure that the workstation id type is valid */
  if((error_status != inquire_okay) ||
((category != GKS$K_WSCAT_OUTIN) && (category != GKS$K_WSCAT_MO))) {
    printf("The specified workstation type is invalid\n");
    printf("Error status: %d\n", error_status);
    return;
  }

  gks$open_ws(&ws_id, &GKS$K_CONID_DEFAULT, &GKS$K_WSTYPE_DEFAULT);
  gks$activate_ws(&ws_id);

  /* Make sure that the deferal mode and regeneration fag are proprely set */
  gks$inq_ws_type(&ws_id,    &error_status,    &dummy_dsc,    &ws_type,
&dummy_integer);
  gks$inq_def_defer_state(&ws_type, &error_status, &def_mode, &regen_flag);

  /* check the status of the inquiry function execution */
  if(error_status != inquire_okay) {
    printf("The deferal inquiry caused an error\n");
    printf("Error status: %d", error_status);
    return;
  }

  /* Defer output as long as possible and suppress implicit regenerations */
  if((def_mode != GKS$K_ASTI) && (regen_flag != GKS$K_IRG_SUPPRESSED))
    gks$set_defer_state(&ws_id, &GKS$K_ASTI, &GKS$K_IRG_SUPPRESSED);
} /* end setup */

clean_up(int ws_id)
{
  gks$update_ws(&ws_id, &GKS$K_PERFORM_FLAG);
  getchar();
  gks$deactivate_ws(&ws_id);
  gks$close_ws(&ws_id);
  gks$close_gks();
} /* end clean_up */



#endif


void draw_vertex_normals(VERTEX *vertex, float length, HMATRIX device_matrix)
{
  VERTEX_NORMAL *normal_list=vertex->normals;
  HVECTOR tvec;
#ifdef GKS_DOLLAR
#define LINE_POINTS 2
   float x_coordinates[LINE_POINTS],
         y_coordinates[LINE_POINTS];
#endif
  while(normal_list){
    tvec.x=vertex->world_position.x+normal_list->normal.x*length;
    tvec.y=vertex->world_position.y+normal_list->normal.y*length;
```

```
    tvec.z=vertex->world_position.z+normal_list->normal.z*length;
    matrix_hhvector(device_matrix, tvec, &tvec);
    tvec.x /= tvec.w; tvec.y /= tvec.w;
#ifdef __TURBOC__
    moveto( (int)tvec.x, (int)tvec.y);
    lineto( (int)vertex->screen_position.x,(int)vertex->screen_position.y);
#endif
#ifdef GKS_DOLLAR
    x_coordinates[0]=tvec.x;
    y_coordinates[0]=tvec.y;
    x_coordinates[1]=vertex->screen_position.x;
    y_coordinates[1]=vertex->screen_position.y;
    gks$polyline(&LINE_POINTS, &x_coordinates, &y_coordinates);
#endif
    normal_list=normal_list->next;
  }
}

void draw_vertex_normal(VERTEX *vertex, VERTEX_NORMAL *normal,
                float length, HMATRIX device_matrix)
{
  HVECTOR tvec;
#ifdef GKS_DOLLAR
#define LINE_POINTS 2
   float x_coordinates[LINE_POINTS],
         y_coordinates[LINE_POINTS];
#endif

    tvec.x=vertex->world_position.x+normal->normal.x*length;
    tvec.y=vertex->world_position.y+normal->normal.y*length;
    tvec.z=vertex->world_position.z+normal->normal.z*length;
    matrix_hhvector(device_matrix, tvec, &tvec);
    tvec.x /= tvec.w; tvec.y /= tvec.w;

#ifdef __TURBOC__
    moveto( (int)tvec.x, (int)tvec.y);
    lineto( (int)vertex->screen_position.x,(int)vertex->screen_position.y);
#endif
#ifdef GKS_DOLLAR
    x_coordinates[0]=tvec.x;
    y_coordinates[0]=tvec.y;
    x_coordinates[1]=vertex->screen_position.x;
    y_coordinates[1]=vertex->screen_position.y;
    gks$polyline(&LINE_POINTS, &x_coordinates, &y_coordinates);
#endif
}

void draw_polygon_normals(POLYGON *polygon,
                float length, HMATRIX device_matrix)
{
  VERTEX_LIST *vertices=polygon->vertices;
  while(vertices){
    draw_vertex_normal(vertices->vertex, vertices->normal,
                length, device_matrix);
    vertices=vertices->rest;
  }
}
```

```
void draw_scene(SCENE *scene)
{
  OBJECT *object=scene->object_head;
  SURFACE *surface;
  POLYGON *polygon;

  float normal_length=fabs(scene->viewport.umax-scene->viewport.umin)/30.0;


  while(object){
    surface=object->surface_head;
    while(surface){
      polygon=surface->polygons;
      while(polygon){
        VECTOR start_pos;
        VERTEX_LIST *vertex_list=polygon->vertices;
#ifdef DRAW_CULLED_POLYGONS
        {
#else
        if(!polygon->culled){
#endif
          start_pos.x=vertex_list->vertex->screen_position.x;
          start_pos.y=vertex_list->vertex->screen_position.y;
#ifdef __TURBOC__
          moveto( (int)start_pos.x, (int)start_pos.y);
   #if NORMAL_DRAW_METHOD == 1
          if(scene->draw_operation==DRAW_VERTEX_NORMALS)
            draw_vertex_normals(vertex_list->vertex, normal_length,
                                scene->device_matrix);
   #endif
          vertex_list=vertex_list->rest;
          while(vertex_list) {
            lineto( (int)vertex_list->vertex->screen_position.x,
                    (int)vertex_list->vertex->screen_position.y);
   #if NORMAL_DRAW_METHOD == 1
            if(scene->draw_operation==DRAW_VERTEX_NORMALS)
              draw_vertex_normals(vertex_list->vertex, normal_length,
                                  scene->device_matrix);
   #endif
            vertex_list=vertex_list->rest;
          }
          lineto( (int)start_pos.x, (int)start_pos.y);
#endif
#ifdef GKS_DOLLAR
          {
            float x_coordinates[MAX_POLY_VERTICES],
                  y_coordinates[MAX_POLY_VERTICES];
            int n;
            n=0;
            while(vertex_list){
              x_coordinates[n]=vertex_list->vertex->screen_position.x;
              y_coordinates[n]=vertex_list->vertex->screen_position.y;
              n++;
#if NORMAL_DRAW_METHOD == 1
              if(scene->draw_operation==DRAW_VERTEX_NORMALS)
                draw_vertex_normals(vertex_list->vertex, normal_length,
                                    scene->device_matrix);
#endif
              vertex_list=vertex_list->rest;
            }
            x_coordinates[n]=start_pos.x;
            y_coordinates[n]=start_pos.y;
```

```
            n++;
            gks$polyline(&n, &x_coordinates, &y_coordinates);
          }
#endif
        }
#if NORMAL_DRAW_METHOD == 2
        if(scene->draw_operation==DRAW_VERTEX_NORMALS)
          draw_polygon_normals(polygon, normal_length, scene->device_matrix);
#endif
        polygon=polygon->next;
      }
      surface=surface->next;
    }
    object=object->next;
  }
}



void initialisation(void)
{
#if !defined(VAXC) && !defined(__TURBOC__)
  ERROR("Nepoznan ciljni racunalnik");
#endif

#ifdef __TURBOC__
 int gdriver=DETECT, gmode, errorcode;

 detectgraph(&gdriver, &gmode);

 errorcode = graphresult();
 if (errorcode != grOk) {
    printf("Napaka v graficnem sistemu: %s\n", grapherrormsg(errorcode));
    ERROR("manjka graficna kartica ali pa je nepoznana");
 }
 initgraph(&gdriver, &gmode, "");
 errorcode = graphresult();

 if (errorcode != grOk)
 {
    printf("Napaka v graficnem sistemu: %s\n", grapherrormsg(errorcode));
    ERROR("");
 }
#endif
#ifdef GKS_DOLLAR
  setup_ws(ws_id);
#endif
}


void terminate(int wait_for_key)
{
  fflush(stdin);
#ifdef __TURBOC__
  if(wait_for_key){
    sound(523);
    delay(100);
    nosound();
    wait_for_key=getchar();
  }
  closegraph();
```

```
#endif
#ifdef GKS_DOLLAR
  if(wait_for_key){
    printf("END konca program >>");
    wait_for_key=getchar();
  }
  clean_up(ws_id);
#endif
}


void return_max_viewport(int *xvmin, int *xvmax, int *yvmin, int *yvmax)
{
#ifdef __TURBOC__
  *xvmin=0;
  *xvmax=getmaxx();
  *yvmin=getmaxy();
  *yvmax=0;
#endif
#ifdef GKS_DOLLAR
  int error_status;
  float device_coordinates_x, device_coordinates_y;
  int raster_units_x, raster_units_y;
  float minx, maxx, miny, maxy;

  gks$inq_max_ds_size(&GKS$K_WSTYPE_DEFAULT, &error_status,
        &GKS$K_METERS, &device_coordinates_x, &device_coordinates_y,
        &raster_units_x, &raster_units_y);
  *xvmin=0;
  *xvmax=raster_units_x;
  *yvmin=0;
  *yvmax=raster_units_y;

  minx=(float)*xvmin;
  maxx=(float)*xvmax;
  miny=(float)*yvmin;
  maxy=(float)*yvmax;

  gks$set_window(&TNR, &minx, &maxx, &miny, &maxy);
  gks$select_xform(&TNR);

#endif
}

void set_device_viewport(int width, int height,
        int *xvmin, int *xvmax, int *yvmin, int *yvmax)
{
#ifdef __TURBOC__
  setviewport(0, 0, width-1, height-1, 0);
  *xvmin=0;
  *xvmax=width-1;
  *yvmin=height-1;
  *yvmax=0;
#endif
#ifdef GKS_DOLLAR
  int error_status;
  float device_coordinates_x, device_coordinates_y;
  int raster_units_x, raster_units_y;
  float minx, maxx, miny, maxy, dev_raster_aspect;

  gks$inq_max_ds_size(&GKS$K_WSTYPE_DEFAULT, &error_status,
        &GKS$K_METERS, &device_coordinates_x, &device_coordinates_y,
```

```
      &raster_units_x, &raster_units_y);
  *xvmin=0;
  *xvmax=width-1;
  *yvmin=0;
  *yvmax=height-1;

  dev_raster_aspect=device_coordinates_x/(float)raster_units_x;


  minx=0.0;
  maxx=(float)*xvmax;
  miny=0.0;
  maxy=(float)*yvmax;


  gks$set_window(&TNR, &minx, &maxx, &miny, &maxy);
  gks$select_xform(&TNR);

  maxx *= dev_raster_aspect;
  maxy *= dev_raster_aspect;

  gks$set_ws_viewport(&ws_id, &minx, &maxx, &miny, &maxy);
#endif
}

void set_color_index(int coli, float red, float green, float blue)
{
#ifdef __TURBOC__
  /* na PC racunalnikih z EGA/VGA kartico je tezko nastavljati RGB*/
  printf("Set coli:%d R:%3.2f G:%3.2f B:%3.2f\n", coli, red, green, blue);
#endif
#ifdef GKS_DOLLAR
  float r, g, b;
  r=red; g=green; b=blue;
  gks$set_color_rep(&ws_id, &coli, &r, &g, &b);
#endif
}


void set_mono_palette(int *num_colors)
{
#ifdef __TURBOC__
  struct palettetype color_palette=
    {16, {0, 8, 1, 17, 41, 57, 9, 25, 11, 43, 59, 27, 31, 23, 54, 36}};
  if (getmaxcolor()<15)
    ERROR("Graficna kartica ni EGA/VGA");
  setallpalette(&color_palette);
  *num_colors=16; /* na PC racunalnikih z EGA/VGA kartico*/
#endif
#ifdef GKS_DOLLAR
  int coli, max_num_colors;
  float red=0.0, green=0.0, blue=0.0, delta_color;
  int error_status, color_flag, num_indexes;

  gks$inq_color_fac(&ws_type, &error_status, &max_num_colors, &color_flag,
                    &num_indexes);
  if(color_flag == GKS$K_MONOCHROME)
    ERROR("Postaja je monokromatska in ne omogoca odtenkov!");
  if(error_status){
    printf("GKS error status = %d\n", error_status);
    ERROR("EMERGENCY STOP");
  }
```

```
  delta_color=1.0/(float)num_indexes;

  for(coli=0; coli<num_indexes; coli++){
    gks$set_color_rep(&ws_id, &coli, &red, &green, &blue);
    red += delta_color;
    green += delta_color;
    blue += delta_color;
  }
  *num_colors=num_indexes;
#endif
}


void show_bitmap_picture(const char *filename)
{
  FILE *f;
  int x, y, xsize, ysize;

  f=fopen(filename, "rb");
  if(errno) {
    perror(filename); ERROR("Odpiranje bitmap datoteke");
  }
  xsize=getw(f);
  ysize=getw(f);
#ifdef __TURBOC__
  for(x=0; x<xsize; x++)
    for(y=0; y<ysize; y++)
      putpixel(x,y,getw(f));
#endif
#ifdef GKS_DOLLAR
  {
    int *cell_array_ptr, bitmap_offset_column=0, bitmap_offset_row=0;
    float minx, maxx, miny, maxy;
    DESC_ARRAY_2(bitmap_dsc2, 0,  0, cell_array_ptr);

    cell_array_ptr=malloc(xsize*ysize*sizeof(int));
    for(x=0; x<xsize; x++)
      for(y=0; y<ysize; y++)
        cell_array_ptr[x*xsize+y]=getw(f);

    /* nastavimo se vrednosti v opisniku tabele barv */
    bitmap_dsc2.dsc$a_pointer=cell_array_ptr;
    bitmap_dsc2.dsc$a_a0=cell_array_ptr;
    bitmap_dsc2.dsc$l_arsize=xsize*ysize*4;
    bitmap_dsc2.dsc$l_m[0]=xsize;
    bitmap_dsc2.dsc$l_m[1]=ysize;
    bitmap_dsc2.dsc$bounds[0].dsc$l_u=xsize-1;
    bitmap_dsc2.dsc$bounds[1].dsc$l_u=ysize-1;

    miny=minx=0.0;
    maxx=(float)xsize-1;
    maxy=(float)ysize-1;
    gks$set_window(&TNR, &minx, &maxx, &miny, &maxy);
    gks$select_xform(&TNR);

    gks$cell_array(&minx, &miny, &maxx, &maxy, &bitmap_offset_column,
        &bitmap_offset_row, &xsize, &ysize, &bitmap_dsc2);
  }
#endif
```

```
  fclose(f);
}
```

12.2.5 Modul za senčenje po Phongu

```
/*
    modul R$PHONG.C     {PHONGovo sencenje}
    izracuna vse piksle po Phongovi metodi

    Projekt     RENDER
    Verzija     0.0
    Datum       22.5.1991
    Avtor       Leon Kos
    Jezik       TURBO C++ 1.0 ali VAXC 3.0
*/

#define PHONG_MESSAGES
#define TDEBUG

#if defined(TDEBUG) && !defined(__TURBOC__)
#undef TDEBUG
#endif

#include <math.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <limits.h>
#include "render.h"

#ifdef TDEBUG
#include <graphics.h>
#endif

typedef struct EDGE_BOX_CELL {
  float x, z;
  VECTOR normal;
  struct EDGE_BOX_CELL *next;
} EDGE_BOX;

static EDGE_BOX **edge_list;
static int **z_buffer,          /* buffer z koordinat posameznih pikslov  */
       **color_buffer,          /* buffer kazalcev na vektor color_buffer */
       *color_buffer_head;      /* buffer barvnih indeksov */
static VECTOR light_vector;     /* normalizirani vektor luci */
static float last_color;        /* maksimalno stevilo barvnih indexov */
static int xsize, ysize;        /* velikost x in y koordinate
                                        rasterskega bufferja */


void list_edge_list(int ysize)
{
 int y;
 for(y=0; y<ysize; y++){
   if(edge_list[y])
     printf("y=%4d %p %p\n\tN1:(%3.2f,%3.2f,%3.2f), N2:(%3.2f,%3.2f,%3.2f)\n\
\tx1:%5.1f x2:%5.1f z1:%5.1f z2:%5.1f\n", y, edge_list[y], edge_list[y]->next,
edge_list[y]->normal.x, edge_list[y]->normal.y, edge_list[y]->normal.z,
edge_list[y]->next->normal.x, edge_list[y]->next->normal.y,
edge_list[y]->next->normal.z,   edge_list[y]->x, edge_list[y]->next->x,
```

```
edge_list[y]->z, edge_list[y]->next->z);
 }
}

static void initialise_buffers(void)
{
  int x,y;

  z_buffer=imatrix(0, xsize-1, 0, ysize-1);

  for(x=0; x<xsize; x++)
    for(y=0; y<ysize; y++)
      z_buffer[x][y]=INT_MIN;

  edge_list=calloc(ysize, sizeof(EDGE_BOX *));
  if(edge_list==NULL) ERROR("Ne morem alocirati tabele robov");

  color_buffer_head=calloc(xsize*ysize, sizeof(int));
  if(color_buffer_head==NULL) ERROR("Ne morem alocirati tabele robov");

  color_buffer=malloc(ysize*sizeof(int *));
  for(x=0; x<xsize; x++)
    color_buffer[x]=color_buffer_head+x*ysize;
}

static void free_buffers(void)
{

  free_imatrix(z_buffer, 0, xsize-1, 0, ysize-1);
  free(edge_list);
  free(color_buffer);
  free(color_buffer_head);
}



static void add_edge_to_list(VERTEX *vertex1, VERTEX *vertex2,
                VERTEX_NORMAL *normal1, VERTEX_NORMAL *normal2)
{
  float y1, y2;
  int iy1, iy2;

  if(vertex1->screen_position.y > vertex2->screen_position.y){
    VERTEX *swap_vertex;
    VERTEX_NORMAL *swap_normal;

    swap_vertex=vertex1;
    vertex1=vertex2;
    vertex2=swap_vertex;

    swap_normal=normal1;
    normal1=normal2;
    normal2=swap_normal;
  }

  y1=vertex1->screen_position.y;
  y2=vertex2->screen_position.y;

  iy1=(int)y1;
  iy2=(int)y2;
```

```
  if(iy1 != iy2) { /* to ni horizontalna linija */
    float x, z, dx, dz, nx, ny, nz, dnx, dny, dnz, _y2y1;
    int y;
    EDGE_BOX *box;

    _y2y1=1.0/(y2-y1);
    x=vertex1->screen_position.x;
    z=vertex1->screen_position.z;
    dx=(vertex2->screen_position.x-x)*_y2y1;
    dz=(vertex2->screen_position.z-z)*_y2y1;
    nx=normal1->normal.x;
    ny=normal1->normal.y;
    nz=normal1->normal.z;
    dnx=(normal2->normal.x-nx)*_y2y1;
    dny=(normal2->normal.y-ny)*_y2y1;
    dnz=(normal2->normal.z-nz)*_y2y1;

    for(y=iy1; y<iy2; y++){
      if(y>=0 && y<ysize){
        if((box=malloc(sizeof(EDGE_BOX)))==NULL)
          ERROR("Alokacija robu poligona");
        box->x=x; box->z=z;
        box->normal.x=nx; box->normal.y=ny; box->normal.z=nz;
        box->next=edge_list[y];
        edge_list[y]=box;
      }
      x += dx; z += dz; nx += dnx; ny += dny; nz += dnz;
    }
  }
}


static void render_segment(int y, EDGE_BOX *box1, EDGE_BOX *box2)
{
  int x, ix1, ix2, iz;

  if(box1->x > box2->x){ /* zamenjaj box1 in box2 */
    EDGE_BOX *temp_box;
    temp_box=box1;
    box1=box2;
    box2=temp_box;
  }

  ix1=(int)box1->x;
  ix2=(int)box2->x;

  if(ix1 != ix2) { /* segment ni nicte dolzine */
    float dx, z, dz;
    static VECTOR normal, delta_normal, normalised_normal;
    int coli;

    dx=box2->x-box1->x;
    z=box1->z;
    dz=(box2->z-z)/dx;
    memmove(&normal, &(box1->normal), sizeof(VECTOR));
    delta_normal.x=(box2->normal.x-normal.x)/dx;
    delta_normal.y=(box2->normal.y-normal.y)/dx;
    delta_normal.z=(box2->normal.z-normal.z)/dx;
    for(x=ix1; x<ix2; x++){     /* premaknemo iz koordinat v blizino z=0 */
      if( (0<=x) && (x<xsize) && ((iz=(int)z+INT_MAX) > z_buffer[x][y]) ) {
        z_buffer[x][y]=iz;
        normalise(normal, &normalised_normal);
```

```
        coli=(int)(dot_product(light_vector, normalised_normal)*last_color);
        if(coli<0) coli=0;
        color_buffer[x][y]= ++coli;
#ifdef TDEBUG
          putpixel(x,y,color_buffer[x][y]);
#endif
      }
      z += dz;
      normal.x += delta_normal.x;
      normal.y += delta_normal.y;
      normal.z += delta_normal.z;
    }
  }
}

static void render_polygon(POLYGON *polygon)
{
  int y;
  VERTEX_LIST *vertex0, *vertex1, *vertex2;

  vertex0=polygon->vertices;
  vertex1=vertex0;
  vertex2=vertex1->rest;
  do{
    add_edge_to_list(vertex1->vertex, vertex2->vertex,
                     vertex1->normal, vertex2->normal);

    vertex1=vertex2;
    vertex2=vertex2->rest;
  } while(vertex2 != NULL);
  add_edge_to_list(vertex1->vertex, vertex0->vertex,
                     vertex1->normal, vertex0->normal);


  for(y=0; y<ysize; y++){
    if(edge_list[y]){
      render_segment(y, edge_list[y], edge_list[y]->next);
      free(edge_list[y]->next); free(edge_list[y]);
      edge_list[y]=NULL;
    }
  }
}

static void render_object(OBJECT *object)
{
  SURFACE *surface=object->surface_head;
  POLYGON *polygon;
  while(surface){
    polygon=surface->polygons;
    while(polygon){
#ifndef RENDER_CULLED_POLYGONS
      if(!polygon->culled)
#endif
        render_polygon(polygon);
      polygon=polygon->next;
    }
    surface=surface->next;
  }
}


void render_scene_phong(SCENE *scene, int x_window_size, int y_window_size,
```

```
                        int max_num_of_colors, const char *output_filename)
{
  OBJECT *object=scene->object_head;

#ifdef PHONG_MESSAGES
    printf("Racunam indekse po Phongovi interpolacijski metodi...");
#endif

  xsize=x_window_size;
  ysize=y_window_size;
  last_color=max_num_of_colors-2; /* index 0 je ozadje */

  initialise_buffers();

  light_vector.x=scene->light_position.x;
  light_vector.y=scene->light_position.y;
  light_vector.z=scene->light_position.z;
  normalise(light_vector, &light_vector);

  while(object){
    render_object(object);
    object=object->next;
  }

  {/* zapisemo binarno datoteko na disk */
    FILE *f;
    int i, color_buffer_size=xsize*ysize;
    f=fopen(output_filename, "wb");
    if(errno){
      perror(output_filename);
      ERROR("Odpiranje ciljne datoteke za Phongovo sencenje");
    }
    putw(xsize, f); putw(ysize, f);
    for(i=0; i<color_buffer_size; i++)
      putw(color_buffer_head[i], f);
    fclose(f);
    free_buffers();
#ifdef PHONG_MESSAGES
    printf("\nRezultat Phongovega sencenja zapisan v datoteko %s.\n",
                output_filename);
#endif
  }
}
```

12.2.6 Modul za senčenje po Guaraudu

```
/*
    modul R$GUAR.C      {GUARaudovo sencenje}
    izracuna vse piksle po Guaraudovi metodi

    Projekt     RENDER
    Verzija     0.0
    Datum       1.6.1991
    Avtor       Leon Kos
    Jezik       TURBO C++ 1.0 ali VAXC 3.0
*/

#define GUARARD_MESSAGES
#define TDEBUG
```

```
#if defined(TDEBUG) && !defined(__TURBOC__)
#undef TDEBUG
#endif

#include <math.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <limits.h>
#include "render.h"

#ifdef TDEBUG
#include <graphics.h>
#endif

typedef struct EDGE_BOX_CELL {
  float x, z, i;
  struct EDGE_BOX_CELL *next;
} EDGE_BOX;

EDGE_BOX **edge_list;
static int **z_buffer,          /* buffer z koordinat posameznih pikslov  */
       **color_buffer,          /* buffer kazalcev na vektor color_buffer */
       *color_buffer_head;      /* buffer barvnih indeksov */
static VECTOR light_vector;     /* normalizirani vektor luci */
static float last_color;        /* maksimalno stevilo barvnih indexov */
static int xsize, ysize;        /* velikost x in y koordinate
                                        rasterskega bufferja */




static void initialise_buffers(void)
{
  int x,y;

  z_buffer=imatrix(0, xsize-1, 0, ysize-1);

  for(x=0; x<xsize; x++)
    for(y=0; y<ysize; y++)
      z_buffer[x][y]=INT_MIN;

  edge_list=calloc(ysize, sizeof(EDGE_BOX *));
  if(edge_list==NULL) ERROR("Ne morem alocirati tabele robov");

  color_buffer_head=calloc(xsize*ysize, sizeof(int));
  if(color_buffer_head==NULL) ERROR("Ne morem alocirati tabele robov");

  color_buffer=malloc(ysize*sizeof(int *));
  for(x=0; x<xsize; x++)
    color_buffer[x]=color_buffer_head+x*ysize;
}

static void free_buffers(void)
{

  free_imatrix(z_buffer, 0, xsize-1, 0, ysize-1);
  free(edge_list);
  free(color_buffer);
  free(color_buffer_head);
```

```
}



static void add_edge_to_list(VERTEX *vertex1, VERTEX *vertex2,
                      VERTEX_NORMAL *normal1, VERTEX_NORMAL *normal2)
{
  float y1, y2;
  int iy1, iy2;

  if(vertex1->screen_position.y > vertex2->screen_position.y){
    VERTEX *swap_vertex;
    VERTEX_NORMAL *swap_normal;

    swap_vertex=vertex1;
    vertex1=vertex2;
    vertex2=swap_vertex;

    swap_normal=normal1;
    normal1=normal2;
    normal2=swap_normal;
  }

  y1=vertex1->screen_position.y;
  y2=vertex2->screen_position.y;

  iy1=(int)y1;
  iy2=(int)y2;

  if(iy1 != iy2) { /* to ni horizontalna linija */
    float x, z, dx, dz, _y2y1, i, di;
    int y;
    EDGE_BOX *box;

    _y2y1=1.0/(y2-y1);
    x=vertex1->screen_position.x;
    z=vertex1->screen_position.z;
    i=dot_product(normal1->normal, light_vector);

    dx=(vertex2->screen_position.x-x)*_y2y1;
    dz=(vertex2->screen_position.z-z)*_y2y1;
    di=(dot_product(normal2->normal, light_vector)-i)*_y2y1;

    for(y=iy1; y<iy2; y++){
      if(y>=0 && y<ysize){
        if((box=malloc(sizeof(EDGE_BOX)))==NULL)
          ERROR("Alokacija robu poligona");
        box->x=x;
        box->z=z;
        box->i=i;
        box->next=edge_list[y];
        edge_list[y]=box;
      }
      x += dx; z += dz; i += di;
    }
  }
}


static void render_segment(int y, EDGE_BOX *box1, EDGE_BOX *box2)
{
```

```
  int x, ix1, ix2, iz;

  if(box1->x > box2->x){ /* zamenjaj box1 in box2 */
    EDGE_BOX *temp_box;
    temp_box=box1;
    box1=box2;
    box2=temp_box;
  }

  ix1=(int)box1->x;
  ix2=(int)box2->x;

  if(ix1 != ix2) { /* segment ni nicte dolzine */
    float dx, z, dz, i, di;

    dx=box2->x-box1->x;
    z=box1->z;
    i=box1->i;
    dz=(box2->z-z)/dx;
    di=(box2->i-i)/dx;
    for(x=ix1; x<ix2; x++){     /* premaknemo iz koordinat v blizino z=0 */
      if( (0<=x) && (x<xsize) && ((iz=(int)z+INT_MAX) > z_buffer[x][y]) ) {
        z_buffer[x][y]=iz;
        if(i<0.0)
          i=0.0;

        color_buffer[x][y]=i*last_color+1;  /*  index  0  je  rezerviran  za
ozadje*/
#ifdef TDEBUG
          putpixel(x,y,color_buffer[x][y]);
#endif
      }
      z += dz;
      i += di;
    }
  }
}

static void render_polygon(POLYGON *polygon)
{
  int y;
  VERTEX_LIST *vertex0, *vertex1, *vertex2;

  vertex0=polygon->vertices;
  vertex1=vertex0;
  vertex2=vertex1->rest;
  do{
    add_edge_to_list(vertex1->vertex, vertex2->vertex,
                     vertex1->normal, vertex2->normal);

    vertex1=vertex2;
    vertex2=vertex2->rest;
  } while(vertex2 != NULL);
  add_edge_to_list(vertex1->vertex, vertex0->vertex,
                     vertex1->normal, vertex0->normal);


  for(y=0; y<ysize; y++){
    if(edge_list[y]){
      render_segment(y, edge_list[y], edge_list[y]->next);
      free(edge_list[y]->next); free(edge_list[y]);
      edge_list[y]=NULL;
```

```
    }
  }
}

static void render_object(OBJECT *object)
{
  SURFACE *surface=object->surface_head;
  POLYGON *polygon;
  while(surface){
    polygon=surface->polygons;
    while(polygon){
#ifndef RENDER_CULLED_POLYGONS
      if(!polygon->culled)
#endif
        render_polygon(polygon);
      polygon=polygon->next;
    }
    surface=surface->next;
  }
}

void render_scene_guarard(SCENE *scene, int x_window_size,
        int y_window_size, int max_num_of_colors,
        const char *output_filename)
{
  OBJECT *object=scene->object_head;

#ifdef GUARARD_MESSAGES
    printf("Racunam indekse po Guarardovi interpolacijski metodi...");
#endif

  xsize=x_window_size;
  ysize=y_window_size;
  last_color=max_num_of_colors-2; /* index 0 je rezerviran za ozadje */

  initialise_buffers();

  light_vector.x=scene->light_position.x;
  light_vector.y=scene->light_position.y;
  light_vector.z=scene->light_position.z;
  normalise(light_vector, &light_vector);

  while(object){
    render_object(object);
    object=object->next;
  }

  {/* zapisemo binarno datoteko na disk */
    FILE *f;
    int i, color_buffer_size=xsize*ysize;
    f=fopen(output_filename, "wb");
    if(errno){
      perror(output_filename);
      ERROR("Odpiranje ciljne datoteke za Guarardovo sencenje");
    }
    putw(xsize, f); putw(ysize, f);
    for(i=0; i<color_buffer_size; i++)
      putw(color_buffer_head[i], f);
    fclose(f);
    free_buffers();
#ifdef GUARARD_MESSAGES
    printf("\nRezultat Guaroudovega sencenja zapisan v datoteko %s.\n",
```

```
                output_filename);
#endif
  }
}
```

12.2.7 Povezovalna *header* datoteka

```
enum PROJECTION_TYPE {PARALLEL, PERSPECTIVE};
enum DRAW_OPERATIONS {DRAW_WIREFRAME, DRAW_VERTEX_NORMALS, PHONG,
                        GUARARD};

typedef struct {float x, y, z;} VECTOR;
typedef struct {float x, y, z, w;} HVECTOR;
typedef float HMATRIX[4][4];
typedef char STRING32[32];
typedef struct {float x0, x1, y0, y1;} WINDOW;

typedef struct VIEWPORT_CELL{
  float umin, umax, vmin, vmax, front_plane, back_plane, height, width;
  int raster_height, raster_width;
} VIEWPORT;

/* Definiranje celic 3D seznama */
typedef struct VERTEX_LIST_CELL {
  struct VERTEX_CELL  *vertex;
  struct VERTEX_NORMAL_CELL *normal;
  struct VERTEX_LIST_CELL *rest;
} VERTEX_LIST;

typedef struct POLYGON_LIST_CELL {
  struct POLYGON_CELL *polygon;
  struct POLYGON_LIST_CELL *rest;
} POLYGON_LIST;

typedef struct VERTEX_NORMAL_CELL {
  struct POLYGON_LIST_CELL *polygons;
  struct VERTEX_NORMAL_CELL *next;
  VECTOR normal;
} VERTEX_NORMAL;

typedef struct OBJECT_CELL {
  struct OBJECT_CELL *next;
  struct SURFACE_CELL *surface_head;
  struct VERTEX_CELL *vertex_head;
  int number;
  int no_of_vertices, no_of_polygons, no_of_surfaces;
  HMATRIX transform;
  VECTOR scale, rotate, translate;
} OBJECT;

typedef struct SURFACE_CELL {
  int no_of_polygons;
  struct POLYGON_CELL *polygons;
  struct SURFACE_CELL *next;
} SURFACE;

typedef struct POLYGON_CELL {
  VERTEX_LIST *vertices;
  struct POLYGON_CELL *next;
  VECTOR normal;
```

```
  char culled;
} POLYGON;

typedef struct VERTEX_CELL {
  struct VERTEX_NORMAL_CELL *normals;
  struct VERTEX_CELL *next;
  VECTOR local_position,
         world_position;

  HVECTOR screen_position;
} VERTEX;

typedef struct SCENE_CELL {
  OBJECT *object_head;
  char projection_type, draw_operation;
  VECTOR view_reference_point,          /* VRC */
        view_plane_normal,              /* WC */
        view_up_vector,                 /* WC */
        projection_reference_point,     /* WC */
        light_position;                 /* WC */
  struct VIEWPORT_CELL viewport;
  float view_plane_distance,
        projection_distance;
  HMATRIX device_matrix;
  char shading_output_filename[80];
} SCENE;



/* Inline makro funkcije */
#define rad(deg) ((deg)*3.1415926536/180.0)
#define sqr(x)   ((x)*(x))
#define dot_product(a,b) (a.x*b.x+a.y*b.y+a.z*b.z)

/* sledeci makroji manjkajo v VAXC knjiznici */
#ifdef VAXC
#define max(a,b)        (((a) > (b)) ? (a) : (b))
#define min(a,b)        (((a) < (b)) ? (a) : (b))
#endif


#define ERROR(message) {\
  perror(message); \
  fprintf(stderr, "\tNapaka v datoteki %s\n\ \
Cas kreiranja datoteke : %s, %s\n\ \
na vrstici #%d\n", __FILE__, __DATE__, __TIME__, __LINE__); \
exit(errno);}

#define TRUE 1
#define FALSE 0
#define ALMOST_ZERO 1E-5

/* Prototipi funkcij */

SCENE *read_scene(char *scene_filename);        /* R$RDDB */
void list_objects(OBJECT *head);
void normalise(VECTOR a, VECTOR *b);            /* R$TRDB */
int **imatrix(int nrl, int nrh, int ncl, int nch);
void free_imatrix(int **m, int nrl,  int nrh, int ncl, int nch);
void matrix_hhvector(HMATRIX m, HVECTOR v, HVECTOR *mv);
void transform_scene_to_world(SCENE *scene);
```

```
void calculate_device_coords(SCENE *scene);
void initialisation(void);                          /* R$DRWF */
void terminate(int wait_for_key);
void return_max_viewport(int *xvmin, int *xvmax, int *yvmin, int *yvmax);
void set_device_viewport(int width, int height,
        int *xvmin, int *xvmax, int *yvmin, int *yvmax);
void draw_scene(SCENE *scene);
void show_bitmap_picture(const char *filename);
void set_color_index(int coli, float red, float green, float blue);
void set_mono_palette(int *num_colors);

/* R$PHONG */
void render_scene_phong(SCENE *scene, int x_window_size, int y_window_size,
                        int max_num_of_colors, const char *output_filename);
/* R$GUAR */
void render_scene_guarard(SCENE *scene, int x_window_size,
        int y_window_size, int max_num_of_colors,
        const char *output_filename);
```

12.2.8 Datoteka scene SCENE1.DAT

```
1               tip projekcije (0=paralelna, 1=perspektivna)
60.0            (VPD) oddaljenost vidne ravnine od referencne tocke v smeri
VPN
70.0            (COP ali PRP) oddaljenost centra projekcije od VRP
1.5 2 0.0       (VRP) referencna tocka pogleda
0 0 1.0 (VPN)   normala ravnine pogleda
0.0 1.0 0.0     (VUP) vektor rotacije vidne ravnine
8 8             (u,v) velikost okna
150             stevilo rasterskih enot visine okna
2               operacija {0=zicni model, 1=0+normale, 2=0+phong 3=0+guaraud}
0 1 1           vektor luci
phong.btm       ime datoteke rezultata sencenja
2               stevilo objektov
1.6 1 1         {skaliranje objekta
-60.0 2.0 0.0   {rotiranje objekta}
0 0 0           {translacija objekta}
house.dat
1.0 1.0 1.0     {skaliranje objekta}
0.0 0.0 0.0     {rotiranje objekta}
0.0 0.0 0.0     {translacija objekta}
```

12.2.9 Datoteka objekta HOUSE.DAT

```
17 13 13        { st. vozlisc, st. mnogokotnikov, st. povrsin}
0  0.0  0.0 54.0       {st., x, y, in z koordinate v lokalnem koor. sistemu}
1 16.0  0.0 54.0
2 16.0 10.0 54.0
3  8.0 16.0 54.0
4  0.0 10.0 54.0
5  0.0  0.0 30.0
6 16.0  0.0 30.0
7 16.0 10.0 30.0
8  8.0 16.0 30.0
9  0.0 10.0 30.0
10 0.0  0.0 58.0
11 4.0  0.0 58.0
12 4.0  0.0 54.0
13 0.0 14.0 54.0
14 0.0 14.0 58.0
```

```
15 4.0 14.0 58.0
16 4.0 14.0 54.0
0 0 5 0 1 2 3 4 {st. mnogokotnika, st. povrsine, st. vozlisc, indeksi vozlisc}
1 1 5 9 8 7 6 5
2 2 4 0 5 6 1
3 3 4 2 1 6 7
4 4 4 2 7 8 3
5 5 4 4 3 8 9
6 6 4 0 4 9 5
7 7 4 10 11 15 14
8 8 4 11 12 16 15
9 9 4 0 13 16 12
10 10 4 10 14 13 0
11 11 4 13 14 15 16
12 12 4 0 12 11 10
```

12.3 Editor LUT

V tem delu je predstavljen editor barvnih indeksov, s katerim lahko nastavljamo profil v RGB barvnem modelu. Editor prebere bitmap datoteko, ki smo jo popreje osenčili s programom za senčenje. Program omogoča izbiro ozadja okna, profila RGB, barve izvora svetlobe in nastavitev jakosti odbleska na površini. Osnovna pomoč je vgrajena v program.

```
/*
    modul LUTEDT.C      {LUT EDiTor}
    Prikaze bitmap datoteko z interaktivnim popravljanjem LUT tabele.

    Projekt     RENDER
    Verzija     0.0
    Datum       13.5.1991
    Avtor       Leon Kos
    Jezik       TURBO C++ 1.0 ali VAXC 3.0
    Oprema      DEC graficni terminali
    Moduli      LUTEDT, R$DRWF, R$TRDB.
*/

#include <stdio.h>
#include <stdlib.h>
#include "render.h"


#define MAXVERTICES 20

typedef struct {
  float r, g, b, x;
} LUT_PROFILE;


void scan_profile_vertex(LUT_PROFILE *lp)
{
```

```
  int ok;
  do{
    ok=1;
    printf("Vstavi R, G, B in X vrednost [0..1] >>");
    if(scanf("%f %f %f %f", &lp->r, &lp->g, &lp->b, &lp->x) != 4) ok=0;
#define check_bound(value) if(value<0.0 || value>1.0) ok=0
    check_bound(lp->r);
    check_bound(lp->g);
    check_bound(lp->b);
    check_bound(lp->x);
  } while(!ok);
}

void redraw_lut(int num_vertices, int num_colors, LUT_PROFILE lp[])
{
  int i;   float dx;
  LUT_PROFILE  *lp0, *lp1;

  lp0= &lp[0];
  lp1= &lp[1];

  dx=1.0/(float)(num_colors-2);


  for(i=1; i<num_vertices; i++){
    float x, kr, kg, kb, red, green, blue;
    int coli;


    kr=(lp1->r - lp0->r)/(lp1->x - lp0->x);
    kg=(lp1->g - lp0->g)/(lp1->x - lp0->x);
    kb=(lp1->b - lp0->b)/(lp1->x - lp0->x);
    coli=(int)(lp0->x*(float)(num_colors-2))+1;
    for(x=lp0->x; x<lp1->x+ALMOST_ZERO; x +=dx, coli++){
        red=kr*(x-lp0->x)+lp0->r;
      green=kg*(x-lp0->x)+lp0->g;
       blue=kb*(x-lp0->x)+lp0->b;
       set_color_index(coli, red, green, blue);
    }
    lp0=lp1;
    lp1= &lp[i+1];
  }
}

main(void)
{
  int num_colors, num_vertices=2, ok=0;
  char filename[160];
  FILE *f;

  LUT_PROFILE lp[MAXVERTICES]= { {0, 0.1, 0, 0}, {0, 1, 0, 1}};
  LUT_PROFILE bkg={1.0 , 1.0, 1.0};     /* barva ozadja */


  int width=50, height=50, xvmin, xvmax, yvmin, yvmax;

  initialisation();

  set_mono_palette(&num_colors);

  printf("Maksimalno stevilo odtenkov na tem terminalu je %d.\n", num_colors);
```

```
set_device_viewport(width, height, &xvmin, &xvmax, &yvmin, &yvmax);
set_color_index(0, bkg.r, bkg.g, bkg.b);

do{
  int i, prompt;
  printf("Trenutna nastavitev ima %d vozlov v RGB profilu z vrednostmi:\n",
              num_vertices);
  for(i=0; i<num_vertices; i++)
    printf("\t#%2d x:%3.2f    R:%3.2f G:%3.2f B:%3.2f\n",
              i, lp[i].x, lp[i].r, lp[i].g, lp[i].b);

  printf("\nZelis [R,V,P,B,L,X,Help] >>");
  fflush(stdin);
  prompt=getchar() & 0xDF;
  switch(prompt){
    case 'P':                                   /* PROFIL */
      printf("Stevilo vozlisc RGB profila>>");
      scanf("%d", &num_vertices);
      if(num_vertices<2){
        printf("Minimalno 2 vozlisca!");
        break;
      }
      printf("Potrebno je vnesti vse podatke za profil.\n");
      for(i=0; i<num_vertices; i++){
        printf("#%2d: ", i);
        scan_profile_vertex(&lp[i]);
      }
      break;

    case 'V':                                   /* VOZLISCE */
       printf("Stevilka vozlisca >>");
       scanf("%d", &i);
       if(i<0 || i>MAXVERTICES-1) break;
         scan_profile_vertex(&lp[i]);
       break;

    case 'R':
       redraw_lut(num_vertices, num_colors, lp);
       break;

    case 'B':
       printf("Vstavi R G B vrednosti ozadja [0..1] >>");
       scanf("%f %f %f", &bkg.r, &bkg.g, &bkg.b);
       set_color_index(0, bkg.r, bkg.g, bkg.b);
       break;

    case 'L':
       printf("Izvor>>>");
       scanf("%s", filename);
       f=fopen(filename, "rb");
       if(f==NULL || errno) {
         perror(filename);
         break;
       }
       width=getw(f);
       height=getw(f);
       fclose(f);
       set_device_viewport(width, height, &xvmin, &xvmax, &yvmin, &yvmax);
       show_bitmap_picture(filename);
       break;

    case 'X':
```

```
        ok++;
        break;

      case 'H':
         printf("Pomoc:\n\
P: nastavitev Profila\n\
V: nastavitev posameznega Vozla\n\
R: izracun LUT tabele glede na podan profil\n\
L: branje \"bitmap\" datoteke\n\
B: nastavitev ozadja okna\n\
H: ta pomoc\n\
X: izhod iz programa\n");
         break;
      default:
        printf("Kaj?\n");
        break;
    }
  } while(!ok);

  terminate(0);

#ifdef __TURBOC__
  return(0);
#endif
}
```

12.4 Program za konverzijo parametrične baze v žični model

Parametrični popis za senčenje po inkrementalnih metodah je potrebno pretvoriti v popis sestavljen iz mnogokotnikov. Sledeči program dela to konverzijo za parametrični popis površin sestavljenih iz Bézierjevih bikubičnih zlepkov. Ker so površine parametrično popisane je mogoče izdelati mrežo mnogokotnikov s poljubno natančnostjo površine. Deljenje (*subdivision*) enega zlepka je uniformno. V praksi se je izkazalo, da je polje 16x16 mnogokotnikov več kot dovolj za prikaz (glej fotografije). Program je optimiziran na velikost popisa 3D baze z mnogokotniki, zato je konverzija relativno počasna. Možne so še dodatne optimizacije na hitrost, ki pa se meni niso zdele pomembne.

```
/*
    modul BEZCNV.C konvertira parametricno datoteko objekta opisanega z
    bikubicnimi zlepki v objekt sestavljen iz poligonov

    Projekt     RENDER
    Verzija     0.1
    Datum       13.5.1991
    Popravek    18.5.1991 Vozlisca se isce nazaj in ne od glave
    Avtor       Leon Kos
```

```
    Jezik       TURBO C++ 1.0 ali VAXC 3.0
*/
#include <errno.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

typedef struct {float x, y, z;} VECTOR;
typedef struct {float x, y, z, w;} HVECTOR;
typedef float HMATRIX[4][4];
typedef int CONTROL_POINTS[16];

typedef struct POLYGON_CELL {
  int surface_no;
  int vertices[4];
  struct POLYGON_CELL *next;
} POLYGON;

typedef struct VERTEX_CELL {
  VECTOR position;
  struct VERTEX_CELL *next, *previous;
} VERTEX;


#define X 0
#define Y 1
#define Z 2
#define W 3
#define ERROR(message) {\
  perror(message); \
  fprintf(stderr, "\tNapaka v datoteki %s\n\ \
Cas kreiranja datoteke : %s, %s\n\ \
na vrstici #%d\n", __FILE__, __DATE__, __TIME__, __LINE__); \
exit(errno);}



float  *control_points[3];
POLYGON *polygon_head;
VERTEX *vertex_head, *last_vertex;
int last_vertex_index;
FILE    *f;
float   eps=1E-5;

void return_zero_matrix(HMATRIX m)
{
  int i,j;
  for(i=0;i<4;i++)
    for(j=0;j<4;j++)
      m[i][j]=0.0;
}

void matrix_mult(HMATRIX m1, HMATRIX m2, HMATRIX m)
{
  int i,j,element;
  return_zero_matrix(m);
  for(i=0; i<4; i++)
    for(j=0; j<4; j++)
      for(element=0; element<4; element++)
        m[i][j] += m1[i][element]*m2[element][j];
}
```

```
void read_control_points(int num_of_points)
{
  int i;
  for(i=0; i<num_of_points; i++){
    int count; float t;
    if(fscanf(f, "%d%f%f%f", &count, &control_points[X][i],
                  &control_points[Y][i], &control_points[Z][i], &t) != 4)
      ERROR("Napaka pri branju kontrolnih tock datoteke podatkov");
  }
}

void return_dbase_size(int *vertices, int *polygons, int *surfaces)
{
  VERTEX *vertex_ptr=vertex_head;
  POLYGON *polygon_ptr=polygon_head;

  *vertices=*polygons=0;

  while(polygon_ptr){
    *polygons+=1;
    *surfaces=polygon_ptr->surface_no;
    polygon_ptr=polygon_ptr->next;
  }
  while(vertex_ptr){
    *vertices+=1;
    vertex_ptr=vertex_ptr->next;
  }
  *surfaces+=1;
}

void list_dbase(FILE *target)
{
  VERTEX *vertex_ptr=vertex_head;
  POLYGON *polygon_ptr=polygon_head;
  int no_of_vertices, no_of_polygons, no_of_surfaces, vertex, polygon;

  return_dbase_size(&no_of_vertices, &no_of_polygons, &no_of_surfaces);

  fprintf(target, "%d %d %d\n", no_of_vertices,
                        no_of_polygons, no_of_surfaces);
  if(errno) ERROR("Pisanje cilja");

  vertex=polygon=0;

  while(vertex_ptr){
    fprintf(target, "%d %f %f %f\n", vertex++, vertex_ptr->position.x,
                vertex_ptr->position.y, vertex_ptr->position.z);
    vertex_ptr=vertex_ptr->next;
  }
  while(polygon_ptr){
    fprintf(target,    "%d    %d    4    %d    %d    %d    %d\n",    polygon++,
polygon_ptr->surface_no,
        polygon_ptr->vertices[0], polygon_ptr->vertices[1],
        polygon_ptr->vertices[2], polygon_ptr->vertices[3]);
    polygon_ptr=polygon_ptr->next;
  }
}


int add_vertex_to_list(VERTEX *vertex, float eps)
{
  VERTEX *current=last_vertex;
```

```
  float delta;
  int n=last_vertex_index;

  if(current==NULL){
    if((vertex_head=malloc(sizeof(VERTEX)))==NULL)
      ERROR("Alikacija prvega vozlisca")
    vertex_head->position.x=vertex->position.x;
    vertex_head->position.y=vertex->position.y;
    vertex_head->position.z=vertex->position.z;
    vertex_head->next=vertex_head->previous=NULL;
    last_vertex=vertex_head;
    return 0;
  }
  while(current->previous!=NULL){
    delta=fabs(vertex->position.x-current->position.x)+
          fabs(vertex->position.y-current->position.y)+
          fabs(vertex->position.z-current->position.z);
    if(delta<eps)
      return n;
    n--;
    current=current->previous;
  }
  current=last_vertex;
  if((current->next=malloc(sizeof(VERTEX)))==NULL)
    ERROR("Alokacija novega vozlisca");
  current->next->position.x=vertex->position.x;
  current->next->position.y=vertex->position.y;
  current->next->position.z=vertex->position.z;
  current->next->next=NULL;
  current->next->previous=current;
  last_vertex=current->next;
  return ++last_vertex_index;
}

void add_polygon_to_list(POLYGON *polygon)
{
  POLYGON *current=polygon_head;
  int i;

  if(current==NULL){
    if((polygon_head=malloc(sizeof(POLYGON)))==NULL)
      ERROR("Alokacija prvega mnogokotnika");
    polygon_head->surface_no=polygon->surface_no;
    for(i=0; i<4; i++) polygon_head->vertices[i]=polygon->vertices[i];
    polygon_head->next=NULL;
    return;
  }
  while(current->next!=NULL){
    current=current->next;
  }
  if((current->next=malloc(sizeof(VERTEX)))==NULL)
    ERROR("Alokacija novega stirikotnika");
  current->next->surface_no=polygon->surface_no;
  for(i=0; i<4; i++) current->next->vertices[i]=polygon->vertices[i];
  current->next->next=NULL;
}



void calculate_bpbt(CONTROL_POINTS cp_index, float bpbt[3][4][4])
{
  float b[4][4]={{-1,  3, -3, 1},
```

```
                { 3, -6,  3, 0},
                {-3,  3,  0, 0},
                { 1,  0,  0, 0}};

  float p[3][4][4], temp_matrix[3][4][4];

  int i, j, k;

  for(i=0; i<3; i++)
    for(j=0; j<4; j++)
      for(k=0; k<4; k++) {
        p[i][j][k]=control_points[i][cp_index[j*4+k]];
      }

  for(i=0; i<3; i++){
    matrix_mult(b, p[i], temp_matrix[i]);
    matrix_mult(temp_matrix[i], b, bpbt[i]);
  }
}


void calculate_vertex(float v, float u, float bpbt[3][4][4], VERTEX *vertex)
{                          /* float u, float v ??????? */
  int i, j, k;
  float uvec[4], vvec[4], tv[3][4];

  uvec[3]=vvec[3]=1.0;
  for(i=3; i>0; i--){
    uvec[i-1]=u*uvec[i];
    vvec[i-1]=v*vvec[i];
  }

  for(i=0; i<3; i++)
    for(j=0; j<4; j++){
     tv[i][j]=0.0;
     for(k=0; k<4; k++)
       tv[i][j]+=bpbt[i][j][k]*vvec[k];
    }
#ifdef CALCULATE_W
                    w=uvec[0]*tv[3][0]+uvec[1]*tv[3][1]+
                         uvec[2]*tv[3][2]+uvec[3]*tv[3][3];
#endif
  vertex->position.x=uvec[0]*tv[0][0]+uvec[1]*tv[0][1]+
                         uvec[2]*tv[0][2]+uvec[3]*tv[0][3];

  vertex->position.y=uvec[0]*tv[1][0]+uvec[1]*tv[1][1]+
                         uvec[2]*tv[1][2]+uvec[3]*tv[1][3];

  vertex->position.z=uvec[0]*tv[2][0]+uvec[1]*tv[2][1]+
                         uvec[2]*tv[2][2]+uvec[3]*tv[2][3];
}



void calculate_polygon(int n, int usteps, int vsteps,
                 float bpbt[3][4][4], POLYGON *polygon, float epsilon)
{

  VERTEX vertex;
  float u, v, delta_u, delta_v, eps;

  u=(float)(n%usteps)/(float)usteps;
```

```
  v=(float)(n/usteps)/(float)vsteps;
  delta_u=1.0/(float)usteps;
  delta_v=1.0/(float)vsteps;
  eps=(delta_u+delta_v)/2.0*epsilon;

  calculate_vertex(u, v, bpbt, &vertex);
  polygon->vertices[0]=add_vertex_to_list(&vertex, eps);

  calculate_vertex(u+delta_u, v, bpbt, &vertex);
  polygon->vertices[1]=add_vertex_to_list(&vertex, eps);

  calculate_vertex(u+delta_u, v+delta_v, bpbt, &vertex);
  polygon->vertices[2]=add_vertex_to_list(&vertex, eps);

  calculate_vertex(u, v+delta_v, bpbt, &vertex);
  polygon->vertices[3]=add_vertex_to_list(&vertex, eps);

#ifndef QUIET
  printf(".");
#endif
}



void read_patches(int no_of_patches, int usteps, int vsteps, float epsilon)
{
  int patch_no, surf_no, no_of_cp, i, n, patch_counter;
  static CONTROL_POINTS cp_index;
  static POLYGON polygon;
  static float bpbt[3][4][4];

  for(patch_counter=0; patch_counter<no_of_patches; patch_counter++){
    fscanf(f, "%d%d%d", &patch_no, &surf_no, &no_of_cp);
    for(i=0; i<no_of_cp; i++)
      fscanf(f, "%d", &cp_index[i]);

    calculate_bpbt(cp_index, bpbt);

    for (n=0; n<usteps*vsteps; n++){
      calculate_polygon(n, usteps, vsteps, bpbt, &polygon, epsilon);
      polygon.surface_no=surf_no;
      add_polygon_to_list(&polygon);
    }
  }
}


main()
{
  int noc, nop, nos, usteps, vsteps, i;
  char filename[80], target[80];
  float epsilon;
  FILE *target_file;

  printf("Konverter povrsin iz bikubicnih zlepkov v poligone objekta\n\n"
         "Izvor (teapot.dat)>>>");
  scanf("%s", filename);

  printf("Stevilo korakov v smeri u in v zlepka (4 4)>>>");
  scanf("%d %d", &usteps, &vsteps);

  printf("Maksimalno odstopanje blizjega vozlisca"
```

```
        " relativno na u in v delitev\n(0.05)>>>");
  scanf("%f", &epsilon);

  printf("Cilj (bez.dat)>>", target);
  scanf("%s", target);

  target_file=fopen(target, "w");
  if(errno) ERROR("Odpiranje datoteke cilja");

  f=fopen(filename, "r");
  if(errno) ERROR("Odpiranje datoteke izvora");
  fscanf(f, "%d%d%d", &noc, &nop, &nos);


  for(i=0; i<3; i++)
    if((control_points[i]=malloc(sizeof(float)*noc))==NULL)
      ERROR("Alokacija tabele kontrolnih tock")

  read_control_points(noc);

  last_vertex_index=0;
  vertex_head=last_vertex=NULL;
  polygon_head=NULL;

  read_patches(nop, usteps, vsteps, epsilon);
  list_dbase(target_file);

#ifdef __TURBOC__
  return 0;
#endif
}
```

12.4.1 Primer datoteke zlepka

```
Datoteka se imenuje PATCH.BEZ
16 1 1  {st. kontrolnih tock, st zlepkov, st. povrsin}
0  0 0 0 {stev. kontrolne tocke, koordinate x, y in z kontrolne tocke}
1  1 0 0
2  2 0 0
3  3 0 0
4  0 2 0
5  1 2 3
6  2 2 3
7  3 2 0
8  0 4 0
9  1 4 3
10 2 4 3
11 3 4 0
12 0 6 0
13 1 6 0
14 2 6 0
15 3 6 0
0 0 16 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 {zlepek, povrsina, st kontr.
tock,
                                                 indeksi kontrolnih tock}
```

12.4.2 Datoteka po konverziji

Po konverziji prejšne datoteke dobimo rezultat v ciljni datoteki. Primer datoteke je podan za štiri deljenja po *u* in *v* parametrih. Rezultat obdelave se je zapisal v datoteko PATCH.DAT. Pretvorjena oblika je taka, da jo sprejme program za senčenje.

```
25 16 1
0 0.000000 0.000000 0.000000
1 0.750000 0.000000 0.000000
2 0.750000 1.500000 0.949219
3 0.000000 1.500000 0.000000
4 1.500000 0.000000 0.000000
5 1.500000 1.500000 1.265625
6 2.250000 0.000000 0.000000
7 2.250000 1.500000 0.949219
8 3.000000 0.000000 0.000000
9 3.000000 1.500000 0.000000
10 0.750000 3.000000 1.265625
11 0.000000 3.000000 0.000000
12 1.500000 3.000000 1.687500
13 2.250000 3.000000 1.265625
14 3.000000 3.000000 0.000000
15 0.750000 4.500000 0.949219
16 0.000000 4.500000 0.000000
17 1.500000 4.500000 1.265625
18 2.250000 4.500000 0.949219
19 3.000000 4.500000 0.000000
20 0.750000 6.000000 0.000000
21 0.000000 6.000000 0.000000
22 1.500000 6.000000 0.000000
23 2.250000 6.000000 0.000000
24 3.000000 6.000000 0.000000
0 0 4 0 1 2 3
1 0 4 1 4 5 2
2 0 4 4 6 7 5
3 0 4 6 8 9 7
4 0 4 3 2 10 11
5 0 4 2 5 12 10
6 0 4 5 7 13 12
7 0 4 7 9 14 13
8 0 4 11 10 15 16
9 0 4 10 12 17 15
10 0 4 12 13 18 17
11 0 4 13 14 19 18
12 0 4 16 15 20 21
13 0 4 15 17 22 20
14 0 4 17 18 23 22
15 0 4 18 19 24 23
```

12.4.3 Parametrična datoteka čajnika

Tu so podane kontrolne točke čajnika, ki sem ga imel za testni primer za tertiranje tehnik senčenja. Datoteka je bila ročno vtipkana in nato urejena na računalniku.

	306	32	5
0	1.40000	0.00000	2.40000
1	1.40000	-0.78400	2.40000
2	0.78400	-1.40000	2.40000
3	0.00000	-1.40000	2.40000
4	1.33750	0.00000	2.53125
5	1.33750	-0.74900	2.53125
6	0.74900	-1.33750	2.53125
7	0.00000	-1.33750	2.53125
8	1.43750	0.00000	2.53125
9	1.43750	-0.80500	2.53125
10	0.80500	-1.43750	2.53125
11	0.00000	-1.43750	2.53125
12	1.50000	0.00000	2.40000
13	1.50000	-0.84000	2.40000
14	0.84000	-1.50000	2.40000
15	0.00000	-1.50000	2.40000
16	-0.78400	-1.40000	2.40000
17	-1.40000	-0.78400	2.40000
18	-1.40000	0.00000	2.40000
19	-0.74900	-1.33750	2.53125
20	-1.33750	-0.74900	2.53125
21	-1.33750	0.00000	2.53125
22	-0.80500	-1.43750	2.53125
23	-1.43750	-0.80500	2.53125
24	-1.43750	0.00000	2.53125
25	-0.84000	-1.50000	2.40000
26	-1.50000	-0.84000	2.40000
27	-1.50000	0.00000	2.40000
28	-1.40000	0.78400	2.40000
29	-0.78400	1.40000	2.40000
30	0.00000	1.40000	2.40000
31	-1.33750	0.74900	2.53125
32	-0.74900	1.33750	2.53125
33	0.00000	1.33750	2.53125
34	-1.43750	0.80500	2.53125
35	-0.80500	1.43750	2.53125
36	0.00000	1.43750	2.53125
37	-1.50000	0.84000	2.40000
38	-0.84000	1.50000	2.40000
39	0.00000	1.50000	2.40000
40	0.78400	1.40000	2.40000
41	1.40000	0.78400	2.40000
42	0.74900	1.33750	2.53125
43	1.33750	0.74900	2.53125
44	0.80500	1.43750	2.53125
45	1.43750	0.80500	2.53125
46	0.84000	1.50000	2.40000
47	1.50000	0.84000	2.40000
48	1.75000	0.00000	1.87500
49	1.75000	-0.89000	1.87500
50	0.98000	-1.75000	1.87500
51	0.00000	-1.75000	1.87500
52	2.00000	0.00000	1.35000
53	2.00000	-1.12000	1.35000
54	1.12000	-2.00000	1.35000
55	0.00000	-2.00000	1.35000
56	2.00000	0.00000	0.90000
57	2.00000	-1.12000	0.90000
58	1.12000	-2.00000	0.90000
59	0.00000	-2.00000	0.90000
60	-0.98000	-1.75000	1.87500
61	-1.75000	-0.98000	1.87500
62	-1.75000	0.00000	1.87500
63	-1.12000	-2.00000	1.35000
64	-2.00000	-1.12000	1.35000
65	-2.00000	0.00000	1.35000
66	-1.12000	-2.00000	0.90000
67	-2.00000	-1.12000	0.90000
68	-2.00000	0.00000	0.90000
69	-1.75000	0.98000	1.87500
70	-0.98000	1.75000	1.87500
71	0.00000	1.75000	1.87500
72	-2.00000	1.12000	1.35000
73	-1.12000	2.00000	1.35000
74	0.00000	2.00000	1.35000
75	-2.00000	1.12000	0.90000
76	-1.12000	2.00000	0.90000
77	0.00000	2.00000	0.90000
78	0.98000	1.75000	1.87500
79	1.75000	0.98000	1.87500
80	1.12000	2.00000	1.35000
81	2.00000	1.12000	1.35000
82	1.12000	2.00000	0.90000
83	2.00000	1.12000	0.90000
84	2.00000	0.00000	0.45000
85	2.00000	-1.12000	0.45000
86	1.12000	-2.00000	0.45000
87	0.00000	-2.00000	0.45000
88	1.50000	0.00000	0.22500
89	1.50000	-0.84000	0.22500
90	0.84000	-1.50000	0.22500
91	0.00000	-1.50000	0.22500
92	1.50000	0.00000	0.15000
93	1.50000	-0.84000	0.15000
94	0.84000	-1.50000	0.15000
95	0.00000	-1.50000	0.15000
96	-1.12000	-2.00000	0.45000
97	-2.00000	-1.12000	0.45000
98	-2.00000	0.00000	0.45000
99	-0.84000	-1.50000	0.22500
100	-1.50000	-0.84000	0.22500
101	-1.50000	0.00000	0.22500
102	-0.84000	-1.50000	0.15000
103	-1.50000	-0.84000	0.15000
104	-1.50000	0.00000	0.15000
105	-2.00000	1.12000	0.45000

106	-1.12000	2.00000	0.45000	168	3.10000	0.00000	0.82500
107	0.00000	2.00000	0.45000	169	2.30000	0.00000	2.10000
108	-1.50000	0.84000	0.22500	170	2.30000	-0.25000	2.10000
109	-0.84000	1.50000	0.22500	171	2.40000	-0.25000	2.02500
110	0.00000	1.50000	0.22500	172	2.40000	0.00000	2.02500
111	-1.50000	0.84000	0.15000	173	2.70000	0.00000	2.40000
112	-0.84000	1.50000	0.15000	174	2.70000	-0.25000	2.40000
113	0.00000	1.50000	0.15000	175	3.30000	-0.25000	2.40000
114	1.12000	2.00000	0.45000	176	3.30000	0.00000	2.40000
115	2.00000	1.12000	0.45000	177	1.70000	0.66000	0.60000
116	0.84000	1.50000	0.22500	178	1.70000	0.66000	1.42500
117	1.50000	0.84000	0.22500	179	3.10000	0.66000	0.82500
118	0.84000	1.50000	0.15000	180	2.60000	0.66000	1.42500
119	1.50000	0.84000	0.15000	181	2.40000	0.25000	2.02500
120	-1.60000	0.00000	2.02500	182	2.30000	0.25000	2.10000
121	-1.60000	-0.30000	2.02500	183	3.30000	0.25000	2.40000
122	-1.50000	-0.30000	2.25000	184	2.70000	0.25000	2.40000
123	-1.50000	0.00000	2.25000	185	2.80000	0.00000	2.47500
124	-2.30000	0.00000	2.02500	186	2.80000	-0.25000	2.47500
125	-2.30000	-0.30000	2.02500	187	3.52500	-0.25000	2.49375
126	-2.50000	-0.30000	2.25000	188	3.52500	0.00000	2.49375
127	-2.50000	0.00000	2.25000	189	2.90000	0.00000	2.47500
128	-2.70000	0.00000	2.02500	190	2.90000	-0.15000	2.47500
129	-2.70000	-0.30000	2.02500	191	3.45000	-0.15000	2.51250
130	-3.00000	-0.30000	2.25000	192	3.45000	0.00000	2.51250
131	-3.00000	0.00000	2.25000	193	2.80000	0.00000	2.40000
132	-2.70000	0.00000	1.80000	194	2.80000	-0.15000	2.40000
133	-2.70000	-0.30000	1.80000	195	3.20000	-0.15000	2.40000
134	-3.00000	-0.30000	1.80000	196	3.20000	0.00000	2.40000
135	-3.00000	0.00000	1.80000	197	3.52500	0.25000	2.49375
136	-1.50000	0.30000	2.25000	198	2.80000	0.25000	2.47500
137	-1.60000	0.30000	2.02500	199	3.45000	0.15000	2.51250
138	-2.50000	0.30000	2.25000	200	2.90000	0.15000	2.47500
139	-2.30000	0.30000	2.02500	201	3.20000	0.15000	2.40000
140	-3.00000	0.30000	2.25000	202	2.80000	0.15000	2.40000
141	-2.70000	0.30000	2.02500	203	0.00000	0.00000	3.15000
142	-3.00000	0.30000	1.80000	204	0.00000	-0.00200	3.15000
143	-2.70000	0.30000	1.80000	205	0.00200	0.00000	3.15000
144	-2.70000	0.00000	1.57500	206	0.80000	0.00000	3.15000
145	-2.70000	-0.30000	1.57500	207	0.80000	-0.45000	3.15000
146	-3.00000	-0.30000	1.35000	208	0.45000	-0.80000	3.15000
147	-3.00000	0.00000	1.35000	209	0.00000	-0.80000	3.15000
148	-2.50000	0.00000	1.12500	210	0.00000	0.00000	2.85000
149	-2.50000	-0.30000	1.12500	211	0.20000	0.00000	2.70000
150	-2.65000	-0.30000	0.93750	212	0.20000	-0.11200	2.70000
151	-2.65000	0.00000	0.93750	213	0.11200	-0.20000	2.70000
152	-2.00000	-0.30000	0.90000	214	0.00000	-0.20000	2.70000
153	-1.90000	-0.30000	0.60000	215	-0.00200	0.00000	3.15000
154	-1.90000	0.00000	0.60000	216	-0.45000	-0.80000	3.15000
155	-3.00000	0.30000	1.35000	217	-0.80000	-0.45000	3.15000
156	-2.70000	0.30000	1.57500	218	-0.80000	0.00000	3.15000
157	-2.65000	0.30000	0.93750	219	-0.11200	-0.20000	2.70000
158	-2.50000	0.30000	1.12500	220	-0.20000	-0.11200	2.70000
159	-1.90000	0.30000	0.60000	221	-0.20000	0.00000	2.70000
160	-2.00000	0.30000	0.90000	222	0.00000	0.00200	3.15000
161	1.70000	0.00000	1.42500	223	-0.80000	0.45000	3.15000
162	1.70000	-0.66000	1.42500	224	-0.45000	0.80000	3.15000
163	1.70000	-0.66000	0.60000	225	0.00000	0.80000	3.15000
164	1.70000	0.00000	0.60000	226	-0.20000	0.11200	2.70000
165	2.60000	0.00000	1.42500	227	-0.11200	0.20000	2.70000
166	2.60000	-0.66000	1.42500	228	0.00000	0.20000	2.70000
167	3.10000	-0.66000	0.82500	229	0.45000	0.80000	3.15000

230	0.80000	0.45000	3.15000
231	0.11200	0.20000	2.70000
232	0.20000	0.11200	2.70000
233	0.40000	0.00000	2.55000
234	0.40000	-0.22400	2.55000
235	0.22400	-0.40000	2.55000
236	0.00000	-0.40000	2.55000
237	1.30000	0.00000	2.55000
238	1.30000	-0.72800	2.55000
239	0.72800	-1.30000	2.55000
240	0.00000	-1.30000	2.55000
241	1.30000	0.00000	2.40000
242	1.30000	-0.72800	2.40000
243	0.72800	-1.30000	2.40000
244	0.00000	-1.30000	2.40000
245	-0.22400	-0.40000	2.55000
246	-0.40000	-0.22400	2.55000
247	-0.40000	0.00000	2.55000
248	-0.72800	-1.30000	2.55000
249	-1.30000	-0.72800	2.55000
250	-1.30000	0.00000	2.55000
251	-0.72800	-1.30000	2.40000
252	-1.30000	-0.72800	2.40000
253	-1.30000	0.00000	2.40000
254	-0.40000	0.22400	2.55000
255	-0.22400	0.40000	2.55000
256	0.00000	0.40000	2.55000
257	-1.30000	0.72800	2.55000
258	-0.72800	1.30000	2.55000
259	0.00000	1.30000	2.55000
260	-1.30000	0.72800	2.40000
261	-0.72800	1.30000	2.40000
262	0.00000	1.30000	2.40000
263	0.22400	0.40000	2.55000
264	0.40000	0.22400	2.55000
265	0.72800	1.30000	2.55000
266	1.30000	0.72800	2.55000
267	0.72800	1.30000	2.40000
268	1.30000	0.72800	2.40000
269	0.00000	0.00000	0.00000
270	1.50000	0.00000	0.15000
271	1.50000	0.84000	0.15000
272	0.84000	1.50000	0.15000
273	0.00000	1.50000	0.15000
274	1.50000	0.00000	0.07500
275	1.50000	0.84000	0.07500
276	0.84000	1.50000	0.07500
277	0.00000	1.50000	0.07500
278	1.42500	0.00000	0.00000
279	1.42500	0.79800	0.00000
280	0.79800	1.42500	0.00000
281	0.00000	1.42500	0.00000
282	-0.84000	1.50000	0.15000
283	-1.50000	0.84000	0.15000
284	-1.50000	0.00000	0.15000
285	-0.84000	1.50000	0.07500
286	-1.50000	0.84000	0.07500
287	-1.50000	0.00000	0.07500
288	-0.79800	1.42500	0.00000
289	-1.42500	0.79800	0.00000
290	-1.42500	0.00000	0.00000
291	-1.50000	-0.84000	0.15000
292	-0.84000	-1.50000	0.15000
293	0.00000	-1.50000	0.15000
294	-1.50000	-0.84000	0.07500
295	-0.84000	-1.50000	0.07500
296	0.00000	-1.50000	0.07500
297	-1.42500	-0.79800	0.00000
298	-0.79800	-1.42500	0.00000
299	0.00000	-1.42500	0.00000
300	0.84000	-1.50000	0.15000
301	1.50000	-0.84000	0.15000
302	0.84000	-1.50000	0.07500
303	1.50000	-0.84000	0.07500
304	0.79800	-1.42500	0.00000
305	1.42500	-0.79800	0.00000

0	0	16	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	16	3	16	17	18	7	19	20	21	11	22	23	24	15	25	26	27
2	0	16	18	28	29	30	21	31	32	33	24	34	35	36	27	37	38	39
3	0	16	30	40	41	0	33	42	43	4	36	44	45	8	39	46	47	12
4	0	16	12	13	14	15	48	49	50	51	52	53	54	55	56	57	58	59
5	0	16	15	25	26	27	51	60	61	62	55	63	64	65	59	66	67	68
6	0	16	27	37	38	39	62	69	70	71	65	72	73	74	68	75	76	77
7	0	16	39	46	47	12	71	78	79	48	74	80	81	52	77	82	83	56
8	0	16	56	57	58	59	84	85	86	87	88	89	90	91	92	93	94	95
9	0	16	59	66	67	68	87	96	97	98	91	99	100	101	95	102	103	104
10	0	16	68	75	76	77	98	105	106	107	101	108	109	110	104	111	112	113
11	0	16	77	82	83	56	107	114	115	84	110	116	117	88	113	118	119	92
12	1	16	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135
13	1	16	123	136	137	120	127	138	139	124	131	140	141	128	135	142	143	132
14	1	16	132	133	134	135	144	145	146	147	148	149	150	151	68	152	153	154
15	1	16	135	142	143	132	147	155	156	144	151	157	158	148	154	159	160	68
16	2	16	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176
17	2	16	164	177	178	161	168	179	180	165	172	181	182	169	176	183	184	173
18	2	16	173	174	175	176	185	186	187	188	189	190	191	192	193	194	195	196
19	2	16	176	183	184	173	188	197	198	185	192	199	200	189	196	201	202	193
20	3	16	203	203	203	203	206	207	208	209	210	210	210	210	211	212	213	214
21	3	16	203	203	203	203	209	216	217	218	210	210	210	210	214	219	220	221
22	3	16	203	203	203	203	218	223	224	225	210	210	210	210	221	226	227	228
23	3	16	203	203	203	203	225	229	230	206	210	210	210	210	228	231	232	211
24	3	16	211	212	213	214	233	234	235	236	237	238	239	240	241	242	243	244
25	3	16	214	219	220	221	236	245	246	247	240	248	249	250	244	251	252	253
26	3	16	221	226	227	228	247	254	255	256	250	257	258	259	253	260	261	262
27	3	16	228	231	232	211	256	263	264	233	259	265	266	237	262	267	268	241
28	4	16	269	269	269	269	278	279	280	281	274	275	276	277	92	119	118	113
29	4	16	269	269	269	269	281	288	289	290	277	285	286	287	113	112	111	104
30	4	16	269	269	269	269	290	297	298	299	287	294	295	296	104	103	102	95
31	4	16	269	269	269	269	299	304	305	278	296	302	303	274	95	94	93	92