

Interaktivna računalna grafika kroz primjere u OpenGL-u

Marko Čupić

Željka Mihajlović

16. veljače 2011.

Sadržaj

Sadržaj	i
Predgovor	1
1 Osnove OpenGL-a	3
1.1 Što je OpenGL	3
1.2 Prvi program	3
1.2.1 Priprema prevodioca	4
1.2.2 Priprema pomoćne biblioteke	4
1.2.3 Priprema strukture direktorija	4
1.2.4 Program	5
1.3 Anatomija GLUT aplikacije	6
1.3.1 Metoda main	7
1.3.2 Događaji	7
1.3.3 IsCRTavanje scene	10
1.3.4 Promjena veličine prozora	10
1.3.5 Crtanje točaka i linija	11
1.4 OpenGL primitivi za crtanje	11
1.5 Animacija i OpenGL	13
1.5.1 Animacija temeljena na metodi idle	17
1.5.2 Animacija temeljena na uporabi timera	20
1.6 Ponavljanje	21
2 Matematičke osnove u računalnoj grafici	23
2.1 Način označavanja	23
2.2 Točka i vektor	23
2.3 Pravac	25
2.3.1 Jednadžba pravca	25
2.3.2 Posebni slučajevi jednadžbe pravca	28
2.4 Homogeni prostor. Homogene koordinate.	30
2.4.1 Ideja	30
2.4.2 Jednadžba 2D pravca u homogenom prostoru	31
2.4.3 Jednadžba 3D pravca u homogenom prostoru	31
2.4.4 Alternativni oblik jednadžbe pravca u 2D u homogenom prostoru	32
2.5 Ravnina	34
2.5.1 Jednadžba ravnine	34
2.5.2 Jednadžba ravnine kroz tri točke	37
2.6 Dodatni često korišteni pojmovi	37
2.6.1 Skalarni i vektorski produkt	37
2.6.2 Baricentrične koordinate	40
2.6.3 Izračun baricentričnih koordinata	40
2.6.4 Odnos trokuta i točke preko baricentričnih koordinata	44
2.7 Česti zadatci	44

2.7.1	Probodište pravca i ravnine	44
2.7.2	Probodište pravca i sfere	45
2.7.3	Probodište pravca i trokuta	46
2.8	Ponavljanje	46
3	Crtanje linija i poligona na rasterskim prikaznim jedinicama	49
3.1	Uvod	49
3.1.1	Bresenhamov postupak crtanja linije	49
3.1.2	Izvod Bresenhamovog algoritma s decimalnim brojevima	51
3.1.3	Izvod Bresenhamovog algoritma s cijelim brojevima	52
3.1.4	Kutevi od 0° do 90°	54
3.1.5	Kutevi od 0° do -90°	55
3.1.6	Konačan kod za sve kuteve	56
3.2	Konveksni poligon	56
3.2.1	Matematički opis poligona	56
3.2.2	Orijentacija vrhova poligona	58
3.2.3	Odnos točke i poligona	60
3.2.4	Bojanje konveksnog poligona	60
3.2.5	Funkcije za rad s poligonima	63
4	Osnovne geometrijske transformacije	67
4.1	Vrste transformacija	69
4.2	2D transformacije	71
4.2.1	Uvod	71
4.2.2	Translacija	71
4.2.3	Rotacija	72
4.2.4	Skaliranje	74
4.2.5	Smik	76
4.2.6	Primjer	78
4.3	3D transformacije	79
4.3.1	Uvod	79
4.3.2	Translacija	80
4.3.3	Rotacija	80
4.3.4	Skaliranje	81
4.3.5	Smik	82
4.3.6	Primjer	82
4.4	<i>OpenGL</i> i transformacije	82
4.4.1	Translacija	83
4.4.2	Skaliranje	83
4.4.3	Rotacija	83
4.5	Transformacije normala	84
5	Projekcije i transformacije pogleda	87
5.1	Projekcije	87
5.1.1	Paralelna projekcija	87
5.1.2	Prespektivna projekcija	90
5.2	Transformacije pogleda	92
5.3	Primjena na poopćenje perspektivne projekcije	96
5.3.1	Korak 1	97
5.3.2	Korak 2	97
5.3.3	Korak 3	98
5.3.4	View-up vektor	99
5.4	Transformacije pogleda na drugi način	101
5.4.1	Izvod	101

5.4.2	Primjena na perspektivnu projekciju	102
5.5	Transformacija pogleda i projekcije u OpenGL-u	102
5.5.1	Korak 1. Transformacije modela i pogleda	104
5.5.2	Korak 2. Projekcija	106
5.5.3	Korak 3. Normalizacija koordinata	108
5.5.4	Korak 4. <i>viewport</i> transformacija	109
5.5.5	Izgradnja projekcijske matrice naredbe glFrustum	111
5.5.6	Izgradnja projekcijske matrice naredbe glOrtho	112
6	Krivulje	115
6.1	Uvod	115
6.1.1	Načini zadavanja krivulja	115
6.1.2	Klasifikacija krivulja i poželjna svojstva	116
6.1.3	Svojstvo neprekinitosti krivulja	116
6.2	Krivulje zadane parametarskim oblikom	117
6.2.1	Uporaba parametarskog oblika	117
6.2.2	Primjer crtanja kružnice	118
6.2.3	Primjer crtanja elipse	119
6.2.4	Konstrukcija krivulje s obzirom na zadane točke	119
6.2.5	Ponavljanje	121
6.3	Bezierove krivulje	122
6.3.1	Aproksimacijska Bezierova krivulja	122
6.3.2	Interpolacijska Bezierova krivulja	127
6.4	Prikaz krivulja pomoću razlomljenih funkcija	129
6.4.1	Prikaz krivulja pomoću kvadratnih razlomljenih funkcija	129
6.4.2	Parametarske derivacije u homogenom prostoru	130
6.4.3	Prikaz krivulja pomoću kubnih razlomljenih funkcija	132
6.4.4	Parametarske derivacije u homogenom prostoru	132
6.4.5	Veza između parametarskih derivacija u radnom i homogenom prostoru	133
6.4.6	Primjer	134
6.4.7	Određivanje matrice A	135
6.4.8	Hermitova krivulja	136
6.4.9	Određivanje matrice A - primjer	137
7	Uklanjanje skrivenih linija i površina	139
7.1	Uvod	139
7.2	Uklanjanje stražnjih poligona – provjera normale	141
7.3	Minimaks provjere	142
7.4	Postupak Watkinsa	145
7.5	Z-spremnik	156
7.6	Postupak Warnocka	162
7.7	Četvero i oktalno stablo	164
7.8	Algoritam Cohen Sutherlanda	165
7.9	Algoritam Cyrus Beck	170
7.10	Binarna podjela prostora BSP	171
8	Osvjetljavanje	177
8.1	Osvjetljavanje	177
8.1.1	Uvod	177
8.1.2	Fizikalni model svjetlosti	177
8.2	Phongov model osvjetljenja	179
8.2.1	Općenito o modelu	179
8.2.2	Ambijentna komponenta	179
8.2.3	Difuzna komponenta	180

8.2.4	Zrcalna komponenta	180
8.2.5	Ukupan utjecaj	181
8.2.6	Primjer	182
8.3	Gouraudovo sjenčanje poligona	183
8.4	Phongovo sjenčanje	186
9	Globalni modeli osvjetljavanja	189
9.1	Uvod	189
9.2	Algoritam bacanja zrake	189
9.2.1	Matematički tretman algoritma	190
9.3	Algoritam praćenja zrake	192
9.3.1	Jednostavno preslikavanje tekstura	195
10	Boje	199
10.1	Uvod	199
10.2	Sheme za prikaz boja – prostori boja	199
10.2.1	Modeli	199
10.2.2	RGB-model	199
10.2.3	CMY/CMYK-model	201
10.2.4	HLS-model	203
10.3	Gamma korekcija	203
10.4	Pamćenje boja na računalu	206
11	Fraktali	209
11.1	Uvod	209
11.2	Samoponavljavajući fraktali	209
11.3	Mandelbrotov fraktal	213
11.3.1	Bojanje Mandelbrotovog fraktala	216
11.4	Julijev fraktal i Julijeva krivulja	219
11.5	IFS fraktali	221
11.5.1	Kako crtamo IFS fraktal	222
11.5.2	Konstrukcija IFS fraktala	224
11.6	Lindermayerovi sustavi	226
11.7	Opseg i površina fraktala. Fraktalna dimenzija.	233
A	Prevodenje programa koji koriste GLUT	239
A.1	Operacijski sustav Windows i primjeri u jeziku C++	239
A.1.1	Microsoft Visual Studio	239
A.1.2	Gcc	239
A.1.3	Bcc	241
A.2	Operacijski sustav Linux i primjeri u jeziku C++	242
A.3	Primjeri u jeziku Java	243
A.4	Primjeri u jeziku Python	243

Popis slika

1.1	Prvi OpenGL program	5
1.2	primitiv GL_POINT	13
1.3	primitiv GL_LINES	13
1.4	primitiv GL_LINE_STRIP	14
1.5	primitiv GL_LINE_LOOP	14
1.6	primitiv GL_TRIANGLES	14
1.7	primitiv GL_TRIANGLE_STRIP	15
1.8	primitiv GL_TRIANGLE_FAN	15
1.9	primitiv GL_QUADS	15
1.10	primitiv GL_QUAD_STRIP	16
1.11	primitiv GL_POLYGON	16
2.1	Vektor između dviju točaka	24
2.2	Pronalazak reflektiranog vektora	24
2.3	Pravac dobiven skaliranjem vektora	26
2.4	Ravnina određena točkom i dvama vektorima	34
2.5	Ravnina i njezina normala	34
2.6	Vektorski produkt	39
2.7	Baricentrične koordinate preko omjera površina trokuta	42
3.1	Prikaz linije na rasterskoj prikaznoj jedinici	50
3.2	Razlika između konveksnog i konkavnog poligona	57
3.3	Poligon	57
3.4	Orijentacija vrhova poligona – u smjeru kazaljke na satu	58
3.5	Orijentacija vrhova poligona – u smjeru suprotnom od smjera kazaljke na satu	58
3.6	Odnos točke i poligona	61
3.7	Podjela bridova na lijeve i desne kod konveksnog poligona	61
3.8	Sjedišta vodoravne zrake i pravaca određenih bridovima konveksnog poligona	62
4.1	Translacija točke	72
4.2	Rotacija točke	73
4.3	Rotacija točke: utjecaj na pojedine komponente	73
4.4	Skaliranje točke	75
4.5	Skaliranje točke: djelovanje po komponentama	75
4.6	Smik	77
4.7	Smik: djelovanje po komponentama	77
4.8	Rotacija oko zadane točke	78
4.9	Rotacija oko zadane točke: nakon translacije	79
5.1	Paralelnna projekcija	88
5.2	Paralelnna projekcija, analiza x -koordinate	88
5.3	Perspektivna projekcija	90
5.4	Perspektivna projekcija, analiza x -koordinate	90
5.5	Perspektivna projekcija, analiza za x -os	91
5.6	Translatirani koordinatni sustavi	92

5.7	Rotacija oko z -osi	93
5.8	Rotacija oko y -osi	94
5.9	Rotacija oko z -osi	95
5.10	Poklopljeni sustavi	96
5.11	Pronalaženje y -osi temeljem <i>view-up</i> vektora	100
5.12	Proces obrade vrhova pri generiranju konačne slike	103
5.13	Model koji koristi naredba <i>glFrustum</i>	107
5.14	Dvije kocke	107
5.15	Model koji koristi naredba <i>gluPerspective</i>	108
5.16	Model koji koristi naredba <i>glOrtho</i>	109
5.17	Utjecaj odabranog <i>viewport</i> -a na konačni prikaz slike	110
6.1	Utjecaj broja točaka na prikaz kružnice	118
6.2	Primjer težinske funkcije zvonolikog oblika	120
6.3	Težinske funkcije uz tri točke krivulje	120
6.4	Konstrukcija Bezierove krivulje	125
7.1	Primjer scene s puno objekata na kojoj je prisutna velika količina objekata (npr. stupovi u pozadini) i poligona koji se ne vide ili su djelomično zaklonjeni.	140
7.2	Uklanjanje poligona na osnovi kuta između vektora normale poligona i vektora prema promatraču	141
7.3	Ortografska i perspektivna projekcija	142
7.4	Primjer min-maks provjere na dvije dužine	143
7.5	Primjer kada se dvije dužine potencijalno preklapaju	144
7.6	Poligoni se potencijalno preklapaju, ali niti jedan brid prvog objekta sigurno se ne preklapa s niti jednim bridom drugog objekta	144
7.7	Linija pretrage (lijevo) i rasponi uzorka (desno) u postupku Watkinsa	146
7.8	Podatkovne strukture u postupku Watkinsa	146
7.9	Watkinsov postupak: dva disjunktna trokuta	151
7.10	Watkinsov postupak: preklapajući trokuti	151
7.11	Watkinsov postupak: jednostavniji slučaj	152
7.12	Watkinsov postupak: trokuti koji se probadaju	153
7.13	Dvije kugle prikazane uporabom z-spremnika	157
7.14	Warnockov postupak: rekurzivna podjela prostora na četiri dijela	163
7.15	Warnockov postupak mogući slučajevi: poligon je izvan prozora (1), poligon je u prozoru ili siječe prozor (2), poligon prekriva prozor (3)	163
7.16	Warnockov postupak - primjer podjele za dva poligona gdje su označeni mogući slučajevi.	164
7.17	Oktalno stablo. Prostor dijelimo u oktatnte ovisno o sadržaju pojedinih oktanata.	166
7.18	Prostor podijeljen na ćelije koje mogu biti prazne granične ili neprozirne. Iz crvene točke se promatra scene, žute su neprozirne ćelije koje zaklanjaju plave ćelije (Schaufler).	167
7.19	Podjela prostora na devet dijelova kod algoritma Cohen Sutherlanda.	167
7.20	Primjer odsijecanja dužine kod algoritma Cohen Sutherlanda.	168
7.21	Primjer odsijecanja poligona kod algoritma Cohen Sutherlanda.	169
7.22	Primjer odsijecanja obzirom na volumen pogleda kod algoritma Cohen Sutherlanda.	169
7.23	Određivanje probodišta pravca na kojem je dužina P_0P_1 i poligona čija je normala \vec{n}_i	171
7.24	Razvrstavanje sjecišta na <i>PE</i> i <i>PL</i>	171
7.25	Izgradnja BSP stabla. Postupak nije dovršen, brid 3 još treba podijeliti i smjestiti u stablo http://www.zemris.fer.hr/predmeti/irg/BSP/	172
7.26	Traženje pozicije točke (očišta) u BSP stablu i pripadne pozicije u sceni http://www.zemris.fer.hr/predmeti/irg/BSP/	174
7.27	Primjer 3D scene sačinjene od niza 'zidova' koji predstavljaju prostor scene. Izvor: http://symbolcraft.com/graphics/bsp/index.php	174
7.28	Sortiranje poligona obzirom na udaljenost od očišta.	175
7.29	Tipični problematični slučajevi kod postupaka uklanjanja skrivenih linija i površina.	175

8.1	Refleksija i refrakcija svjetlosti na površini	178
8.2	Primjer učinka kaustike na površini vode	178
8.3	Određivanje difuzne komponente svjetlosti	180
8.4	Na glatkoj površini dominira zrcalna komponenta (lijevo), a na hrapavoj površini dominira difuzna komponenta (desno)	180
8.5	Određivanje zrcalne komponente svjetlosti	181
8.6	Utjecaj faktora n na zrcalnu komponentu svjetlosti	181
8.7	Prostorna distribucija ambijentne, difuzne, zrcalne i ukupne sume svih komponenti (s lijeva na desno)	182
8.8	Utjecaj parametra n na osvjetljavanje kugle	184
8.9	Izračun srednje normalne u vrh tijela	185
8.10	Interpolacija intenziteta kod Gouraudovog sjenčanja	186
8.11	Interpolacija normala kod Phongovog sjenčanja	187
8.12	Interpolacija normala kod Phongovog sjenčanja, detaljniji primjer	187
9.1	Model bacanja zrake	190
9.2	Definiranje ravnine ekrana	191
9.3	Primjer scene prikazan algoritmom bacanja zrake	192
9.4	Izračun reflektirane i lomljene zrake	193
9.5	Primjer scene prikazan algoritmom praćenja zrake	195
9.6	Primjer teksture za sferu	196
9.7	Primjer praćenja zrake uz preslikavanje teksture na sferni objekt	197
10.1	Aditivno miješanje boja	200
10.2	Komponente RGB potrebne za prikaz svih boja	200
10.3	Komponente XYZ potrebne za prikaz svih boja	201
10.4	Kromatski dijagram	202
10.5	GAMUT boja	202
10.6	CMY-model boja	202
10.7	HLS-model boja	203
10.8	Linearna raspodjela intenziteta	204
10.9	Linearna raspodjela intenziteta (2)	204
10.10	Eksponencijalna raspodjela intenziteta	205
10.11	Pamćenje boja u paleti boja	206
10.12	Direktna pohrana boja	207
11.1	Kochina krivulja – početak konstrukcije	209
11.2	Kochina krivulja – drugi korak rekurzije	210
11.3	Kochina krivulja – rezultat uz dubinu rekurzije 6	210
11.4	Kochina pahuljica - početak	211
11.5	Kochina pahuljica - drugi korak rekurzije	212
11.6	Kochina pahuljica - kraj	212
11.7	Ispitivanje pripadnosti točke Mandelbrotovom skupu	214
11.8	Crtanje Mandelbrotovog skupa	215
11.9	Mandelbrotov skup	216
11.10	Mandelbrotov skup uz bojanje prema brzini divergencije	217
11.11	Mandelbrotov fraktal uz različita uvećanja	218
11.12	Julijev skup za funkciju $z_{n+1} = z_n^2$	219
11.13	Povezan i nepovezan Julijev skup	220
11.14	Primjeri Julijevog fraktala	221
11.15	Primjeri lista paprati generiranog ITS-om	223
11.16	Trokat Sierpinskog	224
11.17	Konstrukcija ITS fraktala	224
11.18	Rezultat konstrukcije ITS fraktala – tepih Sierpinskog	226

11.19L-sustav za Fibonaccijeve brojeve	227
11.20Kochina krivulja generirana L-sustavom	227
11.21Kochina pahuljica generirana L-sustavom	230
11.22L-sustavom s grananjem	232
11.23Opseg Kochine krivulje	233
11.24Površina i opseg trokuta Sierpinskog	234
11.25Površina Kochine krivulje	234
11.26Određivanje Hausdorffove dimenzije segmenta linije	235
11.27Određivanje Hausdorffove dimenzije segmenta kvadrata	236
11.28Određivanje Hausdorffove dimenzije Kochine krivulje	236
11.29Određivanje Hausdorffove dimenzije trokuta Sierpinskog	237

Popis tablica

1.1 OpenGL primitivi za crtanje	12
10.1 Odabir boja kod HLS-modela	203
10.2 Primjer palete boja	207
11.1 IFS za primjer vrste paprati	222
11.2 IFS za trokut Sierpinskog	223
11.3 Tablica za IFS za konstruirani fraktal	225

Izvorni kodovi programa

1.1	Jednostavan primjer	5
1.2	Primjer animacije	17
1.3	Primjer animacije	20
7.1	Crtanje dviju kugli uz z-spremnik	157
7.2	Crtanje dviju kugli uz z-spremnik OpenGL-a	160

Predgovor

Ovaj dokument predstavlja radnu verziju skripte iz računalne grafike: *Interaktivna računalna grafika kroz primjere u OpenGL-u*. Molimo sve pogreške, komentare, nejasnoće te sugestije dojaviti na Marko.Cupic@fer.hr ili Zeljka.Mihajlovic@fer.hr.

Verzija dokumenta: 0.1.2011-02-16.

Poglavlje 1

Osnove OpenGL-a

1.1 Što je OpenGL

Kada govorimo o računalnoj grafici na modernim računalnima, o vizualizaciji ili jednostavno o igranju igara, dva pojma koja se odmah pojavljuju su *OpenGL* i *DirectX*. Oba pojma odnose se na specifikacije (norme) koje danas omogućavaju rad s grafičkim karticama, a u svrhu crtanja 2D i 3D scena. Kako je od te dvije norme *OpenGL* široko prihvaćena i višeplatformska specifikacija, u ovoj knjizi primjeri će biti ilustrirani upravo kroz *OpenGL*, čije je ime kratica od *Open Graphics Library*.

Za *OpenGL* možemo reći da je to višeplatformska specifikacija s podrškom za niz programskih jezika, a čiji je cilj omogućiti pisanje aplikacija koje rade s 2D i 3D grafikom. Uz *OpenGL* možemo vezati još atributa, poput sklopovski neovisna specifikacija, specifikacija neovisna o operacijskom sustavu te specifikacija koja nije vezana uz jednog pojedinačnog proizvođača. *OpenGL* specifikacija definira niz primitiva koji se koriste za izgradnju složenih scena (poput točke, linije i poligona). Također, *OpenGL* nudi mogućnost primjene različitih transformacija nad objektima u sceni, nudi različite vrste projekcija, nudi odbacivanje dijelova objekata koji su promatraču nevidljivi, primjenu tekstura i još niz drugih mogućnosti. Sa stanovišta programera, *OpenGL* je jedan veliki stroj stanja. To znači da, primjerice, jednom kada definiramo boju kojom se crta, svi objekti koji se pošalju na crtanje koristit će upravo navedenu boju – tako dugo dok je ne promijenimo. Isto vrijedi i za sve ostale dijelove *OpenGL-a*.

Međutim, *OpenGL* specifikacija ne miješa se u rad s prozorima, što je danas jedan od temeljnih zadataka operacijskih sustava s grafičkim korisničkim sučeljem. Stoga se uz *OpenGL* tipično koriste još dvije pomoćne biblioteke: *GLU* i *GLUT*.

Biblioteka *GLU* (što je kratica od *OpenGL Utility Library*) obogaćuje skup naredbi koje pruža *OpenGL* i uvodi kompleksnije primitive koji je moguće koristiti u opisu scena; jedan od primjera su NURBS krivulje i površine te sfere, cilindri i stošci. Ova biblioteka uobičajeno dolazi sa svim instalacijama *OpenGL-a*, i nije ju potrebno zasebno instalirati.

Biblioteka *GLUT* (što je kratica od *OpenGL Utility Toolkit*) dodatno pojednostavljuje izradu aplikacija koje koriste *OpenGL* uvođenjem platformski neovisne podrške za stvaranje i rad s prozorima, za hvatanje i obradu niza događaja (poput događaja vezanih uz miša i tipkovnicu), za izradu izborničkih struktura (engl. *menus*), i sl. Biblioteka donosi i definicije još nekih složenih objekata koje programeru stavlja na raspolaganje, poput torusa i čajnika. Ova biblioteka nije standardni dio *OpenGL* instalacija, i bit će je potrebno doinstalirati.

1.2 Prvi program

U ovom poglavlju opisat ćemo što je potrebno napraviti kako bismo preveli i pokrenuli naš prvi program koji će koristiti *OpenGL*. Program ćemo napisati u programskom jeziku C++, na operacijskom sustavu Windows Vista. Ovo međutim neće biti nikakvo ograničenje, jer na gotovo identičan način program možemo prevesti i na operacijskom sustavu Linux. Detaljniji opis kako napraviti pripremu dostupan je u dodatku A. Stoga se preporuča svima koji rade u primjerice u razvojnim okolinama *Microsoft Visual Studio* i drugima, da najprije prouče dodatak A. U nastavku ove sekcije pogledat ćemo to za

jedan konkretni primjer: program pisan u jeziku C++ koji želimo prevesti i pokrenuti na operacijskom sustavu *Microsoft Windows Vista*.

1.2.1 Priprema prevodioca

Za potrebe ove knjige pretpostaviti ćemo da ste instalirali gcc prevodioc (primjerice, *MinGW*). U konkretnom primjeru, prevodioc je instaliran u D:\usr\MinGW. Ovaj prevodioc može se skinuti kao ZIP arhiva, i to je u ovom slučaju i napravljeno. Direktorij D:\usr\MinGW\bin dodan je u varijablu okruženja PATH. Ako to niste napravili, tada je prije poziva prevodioca potrebno zadati naredbu:

```
SET "PATH=%PATH%;D:\usr\MinGW\bin"
```

Staza koja se pri tome koristi mora odgovarati direktoriju u koji ste raspakirali MinGW.

1.2.2 Priprema pomoćne biblioteke

S obzirom da je OpenGL biblioteka temeljena na prilično niskoj razini, razvijene su pomoćne biblioteke koje olakšavaju njegovu uporabu. Najprije je napravljena biblioteka *glut*, a nakon nje i biblioteka *freeglut* koja nudi jednostavnu zamjenu za ovu prvu. Razlog razvoja biblioteke *freeglut* leži u problemima oko licence te činjenici da se biblioteka *glut* više ne razvija, što u današnje doba predstavlja problem. Stoga ćemo u okviru ove knjige sve primjere raditi koristeći biblioteku *freeglut* (uz napomenu da će primjeri raditi bez većih problema i s bibliotekom *glut*).

Biblioteku *freeglut* možete skinuti sa adrese¹. Na njoj ćete naći link na stranicu *Martin Payne's Windows binaries* gdje je potrebno skinuti ZIP arhivu *freeglut 2.6.0 MSVC Package*. U toj arhivi se nalaze dva direktorija: *include* i *lib*, te biblioteka *freeglut.dll*.

Ako ipak želite koristiti biblioteku *glut*, do nje možete doći na stranici *Nate Robins Main OpenGL Chronicles Allies*². Tamo potražite ZIP arhivu *glut-3.7.6-bin.zip* (ili noviju) koja sadrži sve potrebne datoteke.

Glut je prilično korisna biblioteka (*OpenGL Utility Toolkit*) koja olakšava interaktivni rad i općenito pisanje OpenGL aplikacija jer sam OpenGL ne daje podršku za rad s prozorima. Načinjene aplikacije time su neovisne o platformi na kojoj se izvode a istovremeno koriste operacije kao što su zadavanje koordinata mišem u prozoru, pomicanje objekata interaktivno mišem i slično. Time će načinjena aplikacija biti izvediva na različitim operacijskim sustavima.

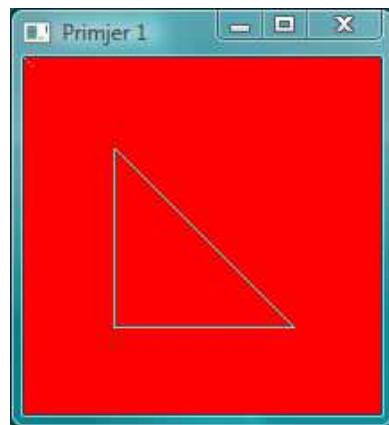
1.2.3 Priprema strukture direktorija

Nakon što ste nabavili ZIP arhivu biblioteke *freeglut*, spremni smo za izradu našeg prvog programa. Na disku ćemo napraviti direktorij **primjer1**. U taj direktorij ćemo iz ZIP arhive iskopirati direktorije *include* i *lib* i biblioteku *freeglut.dll*. Ako ste ovo dobro napravili, trebali biste dobiti strukturu direktorija kako je prikazano u nastavku.

```
primjer1
  include
    GL
      freeglut.h
      freeglut_ext.h
      freeglut_std.h
      glut.h
  lib
    freeglut.lib
    freeglut.dll
```

¹<http://freeglut.sourceforge.net/>

²<http://www.xmission.com/~nate/glut.html>



Slika 1.1: Prvi OpenGL program

Ako ste se odlučili na korištenje biblioteke *glut*, pripadna ZIP arhiva nema direktorije `include` i `lib`, već sve datoteke sadrži na jednom mjestu. U tom slučaju naprije sami u direktoriju `primjer1` napravite poddirektorije `include\GL` i `lib`. U direktorij `include\GL` iskopirajte `glut.h` a u direktorij `lib` iskopirajte `glut32.lib`. Biblioteku `glut32.dll` iskopirajte direktno u direktorij `primjer1`. Ako ste ovo dobro napravili, trebali biste dobiti strukturu direktorija kako je prikazano u nastavku.

```
primjer1
  include
    GL
      glut.h
  lib
    glut32.lib
  glut32.dll
```

1.2.4 Program

Nakon što smo pripremili strukturu direktorija za projekt, dodajmo još i naš prvi program. Program ćemo nazvati `prvi.cpp`, i smjestiti ćemo ga izravno u direktorij `primjer1`. Izvorni kod prikazan je na ispisu 1.1.

Da bismo preveli program, iskoristit ćemo prevodioc `gcc`.

```
gcc -Iinclude -Llib -o prvi.exe prvi.cpp -lfreenglut -lopengl32
```

Argumentom `-Iinclude` direktorij `include` dodajemo u popis direktorija u kojima `gcc` traži zaglavne datoteke. Ovo je potrebno jer smo tamo smjestili zaglavnu datoteku `GL/glut.h` koju koristimo u programu. Argumentom `-Llib` direktorij `lib` dodajemo u popis direktorija u kojima `gcc` traži biblioteke. Ovo je potrebno jer prilikom prevođenja koristimo datoteku `freenglut.lib` koja je tamo smještena. Biblioteka `opengl32.lib` nalazi se među datotekama koje su došle zajedno s *MinGW*-om. Arhument `-o prvi.exe` nalaže prevodiocu da izvršnu datoteku nazove `prvi.exe`. Konačno, argumenti koji započinju s `-l` uključuju pojedine biblioteke.

Uočimo da nam za prevođenje programa nisu potrebne prevedene biblioteke (konkretno, `freenglut.dll`) već samo njihovi opisi (`*.lib` datoteke). Međutim, da bismo program uspješno pokrenuli, prevedene biblioteke moraju biti ili u direktoriju iz kojeg pokrećemo program, ili u direktoriju koji na razini operacijskog sustava čuva sve dijeljene biblioteke. U našem konkretnom slučaju, biblioteku `freenglut.dll` smjestili smo u trenutni direktorij, pa će biti automatski korištena.

Nakon što smo program uspješno preveli, njegovim pokretanjem dobit ćemo prozor s tri točkice i jednim trokutom, kao što je prikazano na slici 1.1.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <windows.h>
4 #include <GL/glut.h>
5
6 void reshape(int width, int height);
7 void display();
8 void renderScene();
9
10 int main(int argc, char **argv) {
11     glutInit(&argc, argv);
12     glutInitDisplayMode(GLUT_DOUBLE);
13     glutInitWindowSize(200, 200);
14     glutInitWindowPosition(0, 0);
15     glutCreateWindow("Primjer 1");
16     glutDisplayFunc(display);
17     glutReshapeFunc(reshape);
18     glutMainLoop();
19 }
20
21 void display() {
22     glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
23     glClear(GL_COLOR_BUFFER_BIT);
24     glLoadIdentity();
25     // crtanje scene:
26     renderScene();
27     glutSwapBuffers();
28 }
29
30 void reshape(int width, int height) {
31     glDisable(GL_DEPTH_TEST);
32     glViewport(0, 0, (GLsizei)width, (GLsizei)height);
33     glMatrixMode(GL_PROJECTION);
34     glLoadIdentity();
35     glOrtho(0, width-1, height-1, 0, 0, 1);
36     glMatrixMode(GL_MODELVIEW);
37 }
38
39 void renderScene() {
40     glPointSize(1.0f);
41     glColor3f(0.0f, 1.0f, 1.0f);
42     glBegin(GL_POINTS);
43     glVertex2i(0, 0);
44     glVertex2i(2, 2);
45     glVertex2i(4, 4);
46     glEnd();
47     glBegin(GL_LINE_STRIP);
48     glVertex2i(50, 50);
49     glVertex2i(150, 150);
50     glVertex2i(50, 150);
51     glVertex2i(50, 50);
52     glEnd();
53 }
```

1.3 Anatomija GLUT aplikacije

Sada kada smo uspjeli prevesti i pokrenuti prvi OpenGL program, vrijeme je da se upoznamo s anatomijom same aplikacije, i pogledamo od čega se sve sastoji program prikazan na ispisu 1.1.

Krenimo redom. Linije 1-4 definiraju sve zaglavne datoteke koje su nam potrebne. Radimo li na operacijskom sustavu Linux, liniju 3 možemo preskočiti. Slijede linije 6-8 koje definiraju prototipove funkcija koje su kasnije korištene u kodu, i uskoro ćemo ih objasniti. Linije 10-19 čine metodu main() - mjesto od kuda započinje izvođenje samog programa. Linije 21-28 čine pomoćnu metodu display, linije 30-37 čine pomoćnu metodu reshape, a linije 39-53 čine pomoćnu metodu renderScene.

1.3.1 Metoda main

Metoda main predstavlja početnu točku programa. U toj metodi potrebno je inicijalizirati biblioteku *GLUT*, podesiti potrebne parametre, i zatim kontrolu izvođenja predati samoj biblioteci. Inicijalizacija biblioteke *GLUT* započinje u retku 11 pozivom: `glutInit(&argc, argv);`, gdje se metodi predaju dva argumenta: adresa varijable koja sadrži broj argumenata iz komandne linije te polje tih argumenata.

U retku 12 pozivom metode `glutInitDisplayMode` slijedi podešavanje načina iscrtavanja scene. Ako se kao argument preda vrijednost `GLUT_SINGLE`, sve metode za iscrtavanje scene će crtati direktno u grafičkom spremniku koji se istovremeno i prikazuje. U tom slučaju nakon što završimo s crtanjem scene, potrebno je pozvati metodu `glFlush();`. Ovaj način prikaza nikako nije prikladan kada se OpenGL koristi za prikazivanje animacija, jer će rezultirati pojmom titranja (engl. *flicker*). Razlog pojave titranja jest taj što se kod animacije stalno ponavlja petlja *obriši scenu – nacrtaj scenu*. Kako se sadržaj grafičkog spremnika u određenim trenutcima čita u svrhu prikaza na ekranu, često se dogodi da se pročita i prikaže do pola obrisana scena ili nedovršena scena.

Rješenje navedenog problema jest uporaba dva grafička spremnika: jednog čiji se sadržaj čita u svrhu prikaza na zaslonu te drugog u koji se crta sljedeća scena. Jednom kada je nova scena nacrtana, pozivom metode `glutSwapBuffers();` mijenja se uloga spremnika: slika se na ekranu počinje prikazivati iz spremnika u kojem smo završili s crtanjem nove scene, a stari spremnik sada postaje spremnik u kojem započinjemo s crtanjem sljedeće scene. Kako se na ovaj način za potrebe prikaza na zaslonu uvijek čita spremnik koji sadrži gotovu sliku scene, neće se javljati titranje. Ovakav način rada podešavamo uporabom parametra `GLUT_DOUBLE`.

Metodi `glutInitDisplayMode` osim specificiranja jednostrukog ili dvostrukog spremnika možemo predati i zastavicu koja određuje kako se radi s bojom (koristi li se RGB ili paleta boja), koristi li se z-spremnik i sl. Primjerice, ako želimo dvostruki spremnik te paletu boja, naredbu ćemo pozvati na sljedeći način:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_INDEX);
```

Ako se ne izjasnimo o načinu prikaza boja, `GLUT_RGBA` smatra se pretpostavljenim načinom, uz koji će se za svaku točku komponente crvene, zelene i plave boje pamtitи nezavisno, što će omogućiti uporabu izuzetno velikog broja različitih boja.

U retku 13 pozivom metode `glutInitWindowSize(200, 200);` podešavamo dimenzije prozora u kojem ćemo iscrtavati scenu. Prvi argument određuje širinu prikaza, a drugi visinu. Oba argumenta zadaju se kao broj slikovnih elemenata (piksela).

U retku 14 pozivom metode `glutInitWindowPosition(0, 0);` određujemo poziciju gornjeg lijevog ugla prozora koji će prikazivati našu scenu. Prvi argument je vrijednost koordinate x a drugi vrijednost koordinate y, u pikselima.

U retku 15 pozivom metode `glutCreateWindow("Primjer 1");` zahtjeva se stvaranje prozora u kojem će se iscrtavati scena. Kao argument se predaje naziv prozora – taj će tekst biti isписан u naslovnoj traci prozora. Metoda `glutCreateWindow` kao rezultat vraća podatak tipa `int` – identifikator prozora koji možemo zapamtitи, i kasnije koristiti za uništavanje prozora pozivom metode `glutDestroyWindow(windowID);`.

Da bismo objasnili sljedeće tri naredbe, moramo se upoznati s načinom na koji GLUT korisniku dojavljuje što se sve događa s prikazom.

1.3.2 Događaji

Grafičke aplikacije najčešće su interaktivne; prate pomake miša korisnika, prate pritiske tipaka i shodno tome, mijenjaju prikazanu scenu. Pomak miša, pritisak te otpuštanje neke od tipaka miša, te pritisak i otpuštanje neke od tipaka na tipkovnici primjeri su onoga što jednostavno zovemo – *događaji* (engl. *events*). Međutim, osim navedenih, GLUT će korisniku dojavljivati još neke događaje. Primjerice, nakon što stvorimo novi prozor, potrebno je u njemu nacrtati scenu. Ako se u nekom trenutku iznad tog prozora otvori i potom zatvori drugi prozor, scenu će ponovno trebati nacrtati. Ako korisnik odluči promijeniti dimenzije prozora, scenu će ponovno trebati nacrtati, a možda i dodatno prilagoditi. Stoga je programski model koji GLUT nudi korisniku model temeljen na događajima: kada se god dogodi nešto što zahtjeva intervenciju korisnika, GLUT će to dojaviti generiranjem određenog događaja.

Da bi korisnik (programer) mogao reagirati na događaj, potrebno je obaviti registraciju – korisnik mora najaviti GLUT-u da želi da se u slučaju pojave događaja X izvede korisnikova metoda `obradiX`.

Dakako, ovisno o vrsti događaja, metode će imati različit broj i vrstu argumenata. Nakon što je i ovaj korak obavljen, kontrolu izvođenja potrebno je predati biblioteki GLUT, koja će ući u beskonačnu petlju osluškivanja, generiranja događaja i pozivanja registriranih metoda koje će potom obraditi te događaje. Sve ovo događa se pozivom metode glutMainLoop();, što je prikazano u retku 18. Pseudokod ove metode mogli bismo prikazati na sljedeći način.

```
void glutMainLoop () {
    while(1) {
        // cekaj na promjenu
        // utvrdi o cemu se radi
        // pozovi registriranu metodu
    }
}
```

Iz ove metode više se nikada ne izlazi, tako da je prije njenog poziva potrebno završiti kompletну inicijalizaciju programa, i registrirati sve željene metode. Upoznajmo se sada redom i s događajima koji nam stoje na raspolaganju.

Potreba za iscrtavanjem scene

Svaki puta kada se prikazana scena na neki način uništi (prekrivanjem nekim drugim prozorom) ili pak promjenom dimenzija prozora, ili kada je scenu potrebno prikazati po prvi puta, GLUT generira događaj *display*. Da bi se korisnik registrirao za taj događaj, treba napisati metodu čiji je prototip prikazan u nastavku.

```
void nazivMetode();
```

Primjer ovakve metode prikazan je u nastavku.

```
void display() {
    // obavi crtanje scene
}
```

Naziv same metode uopće nije bitan. Naime, jednom kada smo metodu napisali, prilikom inicijalizacije biblioteke GLUT potrebno je tu metodu registrirati kao metodu koja se poziva u svrhu iscrtavanja scene. To se radi pozivom metode: glutDisplayFunc(metoda); kojoj kao argument predajemo pokazivač na samu metodu. U ovoj knjizi koristit ćemo konvenciju koja će takvu metodu uvijek zвати *display*.

U primjeru prikazanom na ispisu 1.1, prototip metode *display* naveden je u retku 7, registracija te metode obavljena je u retku 16, a sama metoda definirana je u retcima 21 do 28.

Osim opisanog događaja *display* u primjeru 1.1. mogući su i razni drugi događaji, te su neki od njih opisani u nastavku.

Tipka je pritisnuta

Svaki puta kada korisnik pritisne neku od *običnih* tipki (slovo, znamenku i sl.), GLUT generira događaj *keyboard*. Funkcija koju korisnik može registrirati za obradu ovog događaja treba imati prototip kako slijedi.

```
void nazivMetode(unsigned char key, int x, int y);
```

Takvu funkciju registriramo pozivom metode glutKeyboardFunc(metoda); Primjer takve funkcije prikazan je u nastavku. Prilikom poziva te funkcije, GLUT će nam odmah dostaviti i x i y koordinate na kojima se je u trenutku pritiska tipke nalazio pokazivač miša.

```
void keyPressed(unsigned char key, int x, int y) {
    // obradi; primjerice, zapamti u nekom polju da je
    // tipka key pritisnuta.
}
```

Tipka je otpuštena

Svaki puta kada korisnik otpusti neku od *običnih* tipki (slovo, znamenku i sl.), GLUT generira događaj *keyboardUp*. Funkcija koju korisnik može registrirati za obradu ovog događaja treba imati prototip kako slijedi.

```
void nazivMetode(unsigned char key, int x, int y);
```

Takvu funkciju registriramo pozivom metode glutKeyboardUpFunc(metoda); Primjer takve funkcije prikazan je u nastavku. Prilikom poziva te funkcije, GLUT će nam odmah dostaviti x i y koordinate na kojima se u trenutku otpuštanja tipke nalazio pokazivač miša.

```
void keyReleased(unsigned char key, int x, int y) {
    // obradi; primjerice, zapamti u nekom polju da tipka
    // key nije pritisnuta.
}
```

Posebna tipka je pritisnuta

Svaki puta kada korisnik pritisne neku od *posebnih* tipki (kursorska tipka gore, funkcija tipka i sl.), GLUT generira događaj *special*. Funkcija koju korisnik može registrirati za obradu ovog događaja treba imati prototip kako slijedi.

```
void nazivMetode(int key, int x, int y);
```

Takvu funkciju registriramo pozivom metode glutSpecialFunc(metoda); Primjer takve funkcije prikazan je u nastavku. Prilikom poziva te funkcije, GLUT će nam odmah dostaviti x i y koordinate na kojima se je u trenutku pritiska tipke nalazio pokazivač miša.

```
void keySpecial(unsigned char key, int x, int y) {
    // obradi; primjerice, zapamti u nekom polju da je
    // posebna tipka key pritisnuta.
}
```

Posebna tipka je otpuštena

Svaki puta kada korisnik otpusti neku od *posebnih* tipki (kursorska tipka gore, funkcija tipka i sl.), GLUT generira događaj *specialUp*. Funkcija koju korisnik može registrirati za obradu ovog događaja treba imati prototip kako slijedi.

```
void nazivMetode(int key, int x, int y);
```

Takvu funkciju registriramo pozivom metode glutSpecialUpFunc(metoda); Primjer takve funkcije prikazan je u nastavku. Prilikom poziva te funkcije, GLUT će nam odmah dostaviti x i y koordinate na kojima se je u trenutku otpuštanja tipke nalazio pokazivač miša.

```
void keySpecialUp(unsigned char key, int x, int y) {
    // obradi; primjerice, zapamti u nekom polju da
    // posebna tipka key nije pritisnuta.
}
```

Promjenjena je veličina prozora

Svaki puta kada se promijeni veličina prozora, GLUT generira događaj *reshape*. Funkcija koju korisnik može registrirati za obradu ovog događaja treba imati prototip kako slijedi.

```
void nazivMetode(int width, int height);
```

Takvu funkciju registriramo pozivom metode glutReshapeFunc(metoda); Primjer takve funkcije prikazan je u nastavku.

```
void reshape(int width, int height) {
    // obradi; promijeni potrebne parametre scene
}
```

U primjeru prikazanom na ispisu 1.1, prototip metode reshape naveden je u retku 6, registracija te metode obavljena je u retku 17, a sama metoda definirana je u retcima 30 do 37.

1.3.3 IsCRTavanje scene

Funkciju `display()` u ispisu 1.1 registrirali smo kod GLUT-a kao funkciju koju treba pozvati svaki puta kada se pojavi potreba za iscrtavanjem scene u prozoru. Implementacija te funkcije prikazana je u retcima 21 do 28. Metoda počinje pozivom kojim se postavlja vrijednost boje koja će biti korištenja za brisanje površine platna, odnosno boja pozadine, na kojem radimo iscrtavanje. Radi se o pozivu u retku 22: `glClearColor(1.0f, 0.0f, 0.0f, 1.0f);`. Prva tri argumenta su redom RGB vrijednosti crvene boje, zelene boje i plave boje. Vrijednost 0 kod pojedine komponente znači da ta komponenta nije uključena u stvaranje konačne boje dok vrijednost 1 znači da je uključena maksimalnim intenzitetom. Četvrta komponenta predstavlja α vrijednost boje, odnosno vrijednost koja definira prozirnost te boje. Vrijednost $\alpha = 0$ označava da je boja potpuno prozirna (engl. *transparent*) dok vrijednost $\alpha = 1$ označava da je boja potpuno neprozirna (engl. *opaque*). Prema postavljenim argumentima u našem primjeru, boja pozadine bit će crvena.

U retku 23 slijedi poziv metode `glClear(GL_COLOR_BUFFER_BIT);` kojom se obavlja brisanje, tj. popunjavanje čitave površine prethodno definiranom bojom za brisanje.

Redak 24 sadrži poziv kojim se poništavaju sve prethodno definirane transformacije nad aktivnom transformacijskom matricom, a to je u konkretnom primjeru nad matricom `MODEL_VIEW`, odnosno matricom koja regulira operacije nad kamerom kojom se "snima" scena. To se obavlja tako što se kao matrica koja djeluje nad modelom postavlja matrica identiteta, odnosno jedinična matrica: `glLoadIdentity();`. Ovo na prvi pogled djeluje možda malo zbumujuće, no lagano je za objasniti ako prihvativimo da OpenGL sve transformacije nad scenom radi kroz nekoliko matrica. Također, važno je zapamtiti da se OpenGL ponaša kao jedan veliki automat - svaka naredba koju zadamo mijenja stanje tog automata te ostaje djelovati sve dok to nekom drugom naredbom ne poništimo. Inicijalno, OpenGL se nalazi u stanju u kojem sve operacije djeluju nad matricom modela. Stoga, budući da još ništa nismo mijenjali, a to ćemo kasnije raditi naredbom `glMatrixMode(koja_matrica);`, naredba `glLoadIdentity();` djelovat će upravo nad matricom modela i resetirati će je na jediničnu matricu. U metodi `reshape()` pogledat ćemo primjer u kojem se najprije prebacujemo u stanje u kojem manipuliramo projekcijskom matricom, nakon čega se vraćamo u način koji manipulira matricom modela.

Napomenimo još i da OpenGL uvijek radi s 3D koordinatama točaka. Zapravo, interno radi s 3D-koordinatama kojima je pridružena još i homogena koordinata; više o ovome biti će riječi u kasnijim poglavljima. Za sada je dovoljno razmišljati o tim točkama kao 3D točkama. U ovim početnim primjerima ograničit ćemo se na 2D prikaz, na način da OpenGL-u naložimo uporabu projekcije svih točaka na ravninu $x-y$. Dodatno, pozivat ćemo funkcije koje će točke ili zadavati kao 2D objekte (pa će time z -koordinata implicitno biti postavljena na 0), ili ćemo u slučaju uporabe funkcije koje primaju 3D koordinate treću koordinatu eksplicitno postaviti na 0.

Jednom kada smo sve pripremili za crtanje same scene (čitav spremnik popunili smo pozadinskom bojom, poništili smo sve eventualno zaostale transformacije od prethodnog crtanja), u retku 26 pozivamo našu metodu kojoj je cilj u aktivnom grafičkom spremniku nacrtati samu scenu. Sjetimo se da smo prilikom inicijalizacije OpenGL-a podesili uporabu dvostrukog spremnika - sve što smo do sada radili, radili smo nad spremnikom koji se trenutno ne prikazuje, i služi za pripremu nove slike. Metoda `renderScene();` scenu će nacrtati upravo u tom spremniku.

I konačno, nakon što je slika nacrtana, u retku 27 pozivom metode `glutSwapBuffers();` tražimo od OpenGL-a da na zaslon počne iscrtavati sliku koju smo upravo pripremili, te da spremnik koji je do tada čuvao staru sliku počne koristiti za iscrtavanje nove.

1.3.4 Promjena veličine prozora

Funkciju `reshape()` u ispisu 1.1 (retci 30-37) prilikom inicijalizacije GLUT-a registrirali smo kao funkciju koju GLUT treba pozvati svaki puta kada se promjeni veličina prozora. Argumenti te funkcije su nova širina (`width`) te nova visina (`height`) prozora. Kako u ovom primjeru ne želimo raditi s 3D scenom, prvi korak je isključivanje uporabe *z-spremnika* – strukture koja se koristi kako bi u 3D sceni otkrila između više točaka koje leže na pravcu gledanja, koju točku zapravo vidimo (odnosno koja točka skriva ostale). To se obavlja pozivom metode `glDisable(GL_DEPTH_TEST);` u retku 31. Nakon toga, u retku 32 pozivom funkcije `glViewport(0, 0, (GLsizei)width, (GLsizei)height);` definiramo koji dio prozora prikazuje

samu scenu. Argumenti su redom `x`, `y`, `width` i `height`, što znači da se scena iscrtava počev od piksela na koordinatama $(0,0)$ i proteže `width` piksela u širinu i `height` piksela u visinu – dakle, preko čitavog prozora.

U retku 33 pozivom metode `glMatrixMode(GL_PROJECTION);` govorimo OpenGL-u da će se rad s matricama koji slijedi odnositi na podešavanje matrice projekcije. Više o projekciji bit će objašnjeno u kasnijim poglavljima ove knjige. Slijedi redak 34 s naredbom `glLoadIdentity();` koja kao projekcijsku matricu učitava matricu identiteta – matricu koja ne obavlja ništa, čime efektivno poništava sve prethodno definirane transformacije vezane uz projekciju točaka. U retku 35 potom definiramo novu projekcijsku matricu, koja obavlja ortogonalnu projekciju; naredba je `glOrtho(0, width-1, height-1, 0, 0, 1);`. Semantika ove naredbe bit će detaljno objašnjena nakon što se upoznamo s pojmom projekcije i vrstama koje se koriste. Za sada, bit će dovoljno reći da ovako definirana projekcija prikazuje sve slikovne elemente koji po `x`-u idu od 0 do `width-1`, te koji po `y`-u idu od 0 do `height-1` a ishodište im je u gornjem lijevom kutu. Želimi li ishodište u donjem lijevom kutu, to možemo načiniti jednostavnom promjenom `glOrtho(0, width-1, 0, height-1, 0, 1);`. Svaka 3D točka oblika (x,y,z) kod koje su `x` i `y` koordinate unutar propisanog raspona naprsto se preslikala u 2D prostor u točku (x,y) .

Nakon što smo podesili parametre koji definiraju što se točno prikazuje, OpenGL prebacujemo u stanje u kojem daljnje zadavanje transformacija ponovno djeluje nad matricom modela, što u retku 36 radimo pozivom metode `glMatrixMode(GL_MODELVIEW);`. Važno je ovaj korak ne preskočiti. Naime, sjetimo se da je OpenGL stroj stanja. Kako se funkcija `reshape` tipično zove neposredno prije metode `display` (kada dođe do promjene veličine, ili prvi puta kada stvorimo novi prozor), preskočimo li ovaj korak, poziv metode `glLoadIdentity();` u retku 24 ponovno bi resetirao projekcijsku matricu, a ne transformacije nad kamerom – kako smo to prethodno objasnili.

1.3.5 Crtanje točaka i linija

I konačno, metoda `renderScene()` (retci 39-53) radi konkretno crtanje. Prisjetimo se, ta je metoda pozvana iz metode `display` nakon što je grafički spremnik bio popunjen s pozadinskom bojom. Pa krenimo redom. U retku 40 pozivom metode `glPointSize(1.0f);` podešava se veličina *točke* koju će OpenGL koristiti prilikom crtanja. Primjerice, ako veličinu točke postavimo na 1, naredba koja točku crta na mjestu $(10,10)$ doista će i upaliti samo jedan piksel. Međutim, stavimo li primjerice kao veličinu točke 10, naredba koja točku crta na mjestu $(10,10)$ nacrtat će popunjeno krug čije je središte na zadanom mjestu.

U retku 41 naredbom `glColor3f(0.0f, 1.0f, 1.0f);` definiramo boju koja će se koristiti za crtanje točaka. Argumenti su vrijednosti R, G i B komponenti boje. Nakon ovoga slijede još dva bloka naredbi koji započinju naredbom `glBegin(vrstaPrimitiva);` i završavaju naredbom `glEnd();`

Prvi primjer koji se proteže od retka 42 do retka 46 koristi primitiv `GL_POINTS`. Ovaj primitiv crta slijed točaka koje mu se zadaju – jednu po jednu točno kako su zadane. U ovom primjeru, palimo piksele smještene na koordinatama $(0,0)$, $(2,2)$ i $(4,4)$. Svaka točka zadaje se naredbom `glVertex2i(x, y);`. U toj naredbi broj 2 označava da se radi o 2D koordinati, a slovo *i* označava da su koordinate cijeli brojevi.

Sljedeći primjer koji se proteže od retka 47 do retka 52 demonstrira uporabu primitiva `GL_LINE_STRIP`. Ovaj primitiv niz točaka koje navedemo spaja linijama, i to onim redoslijedom kojim su zadane točke. Primjerice, ako nakon `glBegin(GL_LINE_STRIP);` zadamo četiri točke: t_1 , t_2 , t_3 i t_4 , naredba će povući linije t_1-t_2 , t_2-t_3 i t_3-t_4 . S obzirom da se u primjeru crta trokut, zadani su redom prvi vrh trokuta, drugi vrh trokuta, treći vrh trokuta i konačno još jednom je ponovljen prvi vrh trokuta (čime je povučena linija od trećeg vrha natrag do prvog vrha i tako je zatvoren trokut).

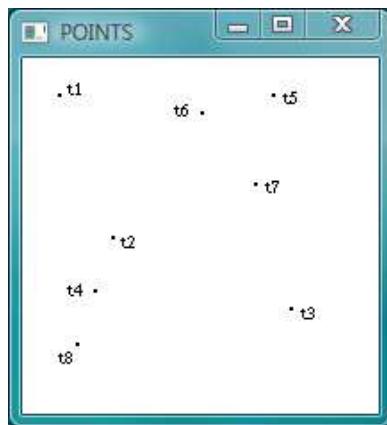
1.4 OpenGL primitivi za crtanje

U prethodnom poglavlju spomenuli smo već da se crtanje u OpenGL-u radi uporabom bloka naredbi koji započinje naredbom `glBegin(vrstaPrimitiva);` i završava naredbom `glEnd();`. Između ta dva graničnika, zadaje se jedna ili više točaka, koje se potom obrađuju, ovisno o odabranu primitivu `vrstaPrimitiva`. U nastavku ćemo pogledati što nam sve stoji na raspolaganju.

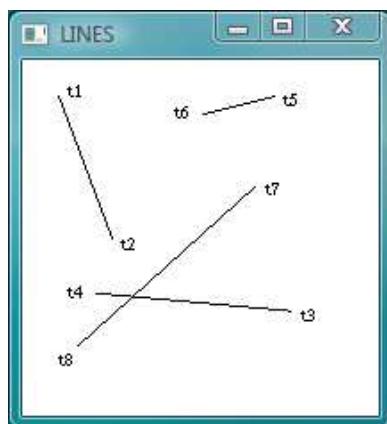
Tablica 1.1: OpenGL primitivi za crtanje

Primitiv	Opis
GL_POINTS	Svaka zadana točka crta se zasebno. Zadamo li n točaka, toliko će ih biti i nacrtano.
GL_LINES	Svake dvije točke tumače se kao jedan segment linije, i tako se crtaju. Zadamo li n točaka, nacrtat će se $n/2$ linija: $t_1-t_2, t_3-t_4, \dots, t_{n-3}-t_{n-2}, t_{n-1}-t_n$.
GL_LINE_STRIP	Zadane točke tumače se kao vrhovi poligona koji treba nacrtati. Zadamo li n točaka, nacrtat će se $n - 1$ linija: $t_1-t_2, t_2-t_3, \dots, t_{n-2}-t_{n-1}, t_{n-1}-t_n$.
GL_LINE_LOOP	Zadane točke tumače se kao vrhovi zatvorenog poligona koji treba nacrtati. Zadamo li n točaka, nacrtat će se n linija: $t_1-t_2, t_2-t_3, \dots, t_{n-2}-t_{n-1}, t_{n-1}-t_n, t_n-t_1$.
GL_TRIANGLES	Zadane točke grupiraju se u grupe po tri, i tumače kao vrhovi trokuta koje treba nacrtati. Zadamo li n točaka, nacrtat će se $n/3$ trokuta: trokut $t_1-t_2-t_3$, trokut $t_4-t_5-t_6$, itd. Trokuti se popunjavaju aktivnom bojom.
GL_TRIANGLE_STRIP	Također služi za crtanje trokuta, ali uz prepostavku da susjedni trokuti dijele jednu zajedničku stranicu, pa se time štedi na broju točaka koje treba zadati. Zadane točke tumače se kao vrhovi trokuta. Zadamo li n točaka, nacrtat će se $n - 2$ trokuta. Za neparni vrh k crta se trokut $t_k-t_{k+1}-t_{k+2}$, dok se za parni vrh k crta trokut $t_{k+1}-t_k-t_{k+2}$. Trokuti se popunjavaju aktivnom bojom.
GL_TRIANGLE_FAN	Također služi za crtanje trokuta, ali uz prepostavku da svi trokuti dijele jedan zajednički vrh, pa se time štedi na broju točaka koje treba zadati. Zadane točke tumače se kao vrhovi trokuta. Zadamo li n točaka, nacrtat će se $n - 2$ trokuta. Redom se crtaju trokuti $t_1-t_2-t_3, t_1-t_3-t_4, t_1-t_4-t_5$, itd. Trokuti se popunjavaju aktivnom bojom.
GL_QUADS	Zadane točke grupiraju se u grupe po četiri, i tumače kao vrhovi četverokuta koje treba nacrtati. Zadamo li n točaka, nacrtat će se $n/4$ četverokuta: četverokut $t_1-t_2-t_3-t_4$, četverokut $t_5-t_6-t_7-t_8$, itd. Četverokuti se popunjavaju aktivnom bojom.

Nastavlja se na sljedećoj stranici



Slika 1.2: primitiv GL_POINT



Slika 1.3: primitiv GL_LINES

Tablica 1.1 – nastavak s prethodne stranice

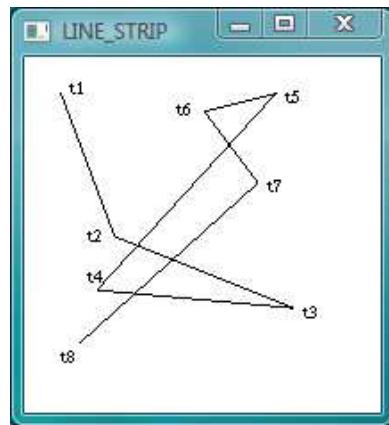
Primitiv	Opis
GL_QUAD_STRIP	Zadane točke tumače kao vrhovi povezanih četverokuta koje treba nacrtati. Zadamo li n točaka, nacrtat će se $n/2 - 1$ četverokuta: četverokut $t_0-t_1-t_3-t_2$, četverokut $t_2-t_3-t_5-t_4$, itd. Četverokuti se popunjavaju aktivnom bojom.
GL_POLYGON	Crta se jedan konveksan poligon koji je određen zadanim točkama.

Primjeri rada ovih primitiva prikazani su na slikama 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10 i 1.11.

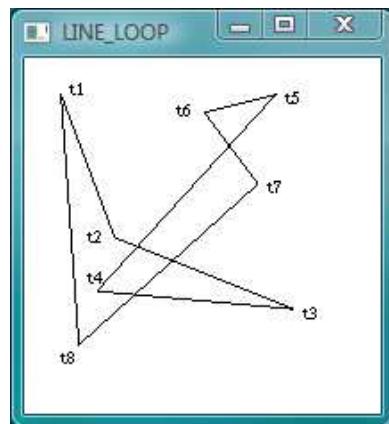
1.5 Animacija i OpenGL

U prvom dijelu ovog poglavlja već smo spomenuli da *OpenGL* ne nudi direktno podršku za rad s prozorima; u tu svrhu koristili smo biblioteku *GLUT*. Unutar iste biblioteke nalazi se i osnovna podrška za izradu animacija. Osnovna ideja animacije jest prikazati promjenu kroz vrijeme. A što se može mijenjati? Tipično, govorimo o:

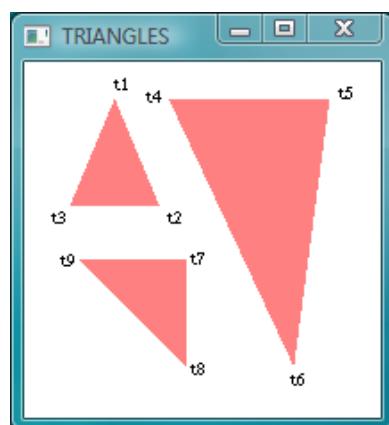
- *svojstvima objekata u sceni* – pri čemu se može mijenjati oblik, tekstura, položaj i orientacija u prostoru, te čak i sam broj objekata,



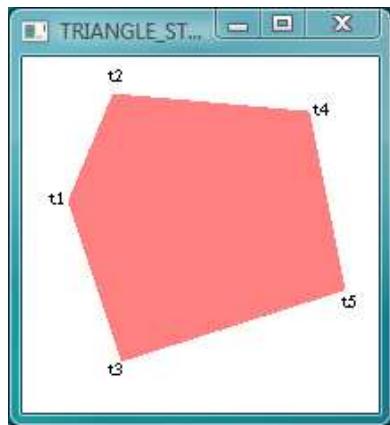
Slika 1.4: primitiv GL_LINE_STRIP



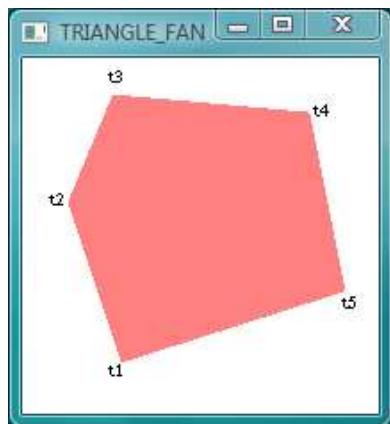
Slika 1.5: primitiv GL_LINE_LOOP



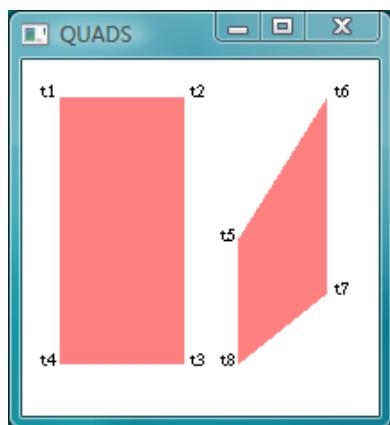
Slika 1.6: primitiv GL_TRIANGLES



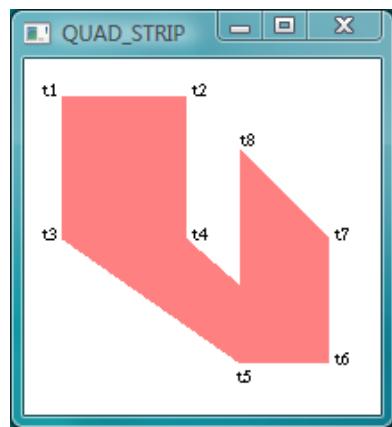
Slika 1.7: primitiv GL_TRIANGLE_STRIP



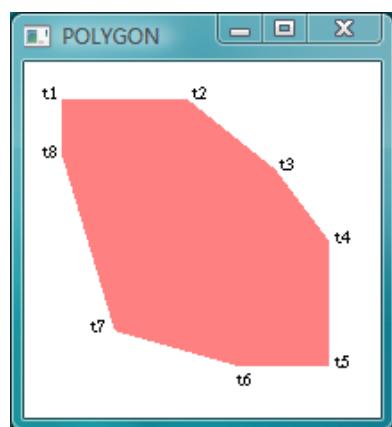
Slika 1.8: primitiv GL_TRIANGLE_FAN



Slika 1.9: primitiv GL_QUADS



Slika 1.10: primitiv GL_QUAD_STRIP



Slika 1.11: primitiv GL_POLYGON

- *položaju promatrača* – što nam omogućava da kameru pomičemo kroz prostor (ali ne mijenjamo točku u koju je kamera usmjerena) te
- *točki pogleda* – što prepostavlja da je položaj kamere fiksiran, ali se mijenja točka u koju kamera gleda.

Dakako, u okviru animacije mogući su i složeniji slučajevi. Primjerice, kamera se može pomicati kroz scenu pri čemu se istovremeno može mijenjati i točka u koju je kamera usmjerena. Ovdje ćemo se prvenstveno fokusirati na općenitu podršku animaciji koja dozvoljava sve navedene slučajeve. Međutim, kako još nismo govorili o 3D scenama, primjeri će se ilustrirati animaciju koja prikazuje promjenu svojstava objekata.

1.5.1 Animacija temeljena na metodi idle

Prvi primjer prikazan je u ispisu 1.2.

Ispis 1.2: Primjer animacije

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <GL/glut.h>

void reshape(int width, int height);
void display();
void renderScene();
void idle();
void drawSquare();
int kut = 0;

int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE);
    glutInitWindowSize(600, 300);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Primjer animacije");
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(idle);
    glutMainLoop();
}

void idle() {
    kut++;
    if(kut>=360) kut = 0;
    glutPostRedisplay();
}

void display() {
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    renderScene();
    glutSwapBuffers();
}

void reshape(int width, int height) {
    glDisable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, width-1, 0, height-1, 0, 1);
    glViewport(0, 0, (GLsizei)width, (GLsizei)height);
    glMatrixMode(GL_MODELVIEW);
}
```

```

void drawSquare() {
    glBegin(GL_QUADS);
    glVertex2f( 0.0f, 0.0f );
    glVertex2f(100.0f, 0.0f );
    glVertex2f(100.0f, 100.0f );
    glVertex2f( 0.0f, 100.0f );
    glEnd();
}

void renderScene() {
    glPointSize(1);
    glColor3f(1.0f, 0.0f, 0.3f);

    glPushMatrix();
    glTranslatef(-150.0f, 160.0f, 0.0f);
    glScalef(1.5f, 1.5f, 1.0f);
    glRotatef((float)kut, 0.0f, 0.0f, 1.0f);
    glTranslatef(-50.0f, -50.0f, 0.0f);
    drawSquare();
    glPopMatrix();

    glPushMatrix();
    glTranslatef( 400.0f, 160.0f, 0.0f);
    glScalef(1.5f, 1.5f, 1.0f);
    glRotatef(-(float)kut, 0.0f, 0.0f, 1.0f);
    glTranslatef(-50.0f, -50.0f, 0.0f);
    drawSquare();
    glPopMatrix();
}

```

Ovaj program prikazuje dva kvadrata koja se rotiraju; prvi u smjeru kazaljke na satu, drugi u suprotnom smjeru. U prikazanom primjeru iskorištena je *GLUT*-ova funkcija *glutIdleFunc(...)* koja nam omogućava registraciju funkcije koju treba pozvati svaki puta kada *GLUT* više nema nikakvog posla (obrađeni su svi događaji). Registracija je obavljena u metodi *main(...)* gdje se traži pozivanje naše funkcije **void idle() ...**. U tijelu funkcije *idle* povećavamo globalnu varijablu *kut* (za 1 stupanj) te pozivom metode *glutPostRedisplay()* dojavljujemo *GLUT*-u da bi sadržaj prozora trebalo ponovno nacrtati. Uočimo da poziv *glutPostRedisplay()* ne pokreće odmah iscrtavanje prozora – interno, metoda samo postavlja zastavicu da bi prozor ponovno trebalo nacrtati. Višestruki pozivi te metode također neće uzrokovati višestruka iscrtavanja. Fizičko iscrtavanje dogodit će se tek nakon što se kontrola izvođenja vrati metodi *glutMainLoop()*, koja će ustanoviti da postoji potreba za ponovnim iscrtavanjem prozora, pa će pozvati u tu svrhu registriranu funkciju (u našem primjeru to će biti metoda *display()*).

Posljedica ovakvog pristupa animaciji jest da će se animacija odvijati maksimalnom brzinom. Naime, čim se završi obrada svih događaja, *GLUT* će pozvati našu funkciju *idle()* koja će tražiti ponovno iscrtavanje prozora. Povratkom kontrole metodi *glutMainLoop()* pozvat će se funkcija za crtanje, i kako nakon toga više neće biti nikakvog posla, pozvat će se naša registrirana metoda *idle()*, koja će opet zatražiti novo crtanje. Brzina animacije bit će, dakle, određena brzinom centralnog procesora računala te brzinom grafičke kartice.

Pogledajmo još kako je izvedeno samo crtanje. Najprije, definirana je metoda *drawSquare()* koja crta jedan kvadrat, i to uvijek na istoj poziciji: lijevi donji ugao je ishodište – točka (0,0) – a desni gornji ugao je točka (100,100). Kvadrat crtamo uporabom primitiva *GL_QUADS*. Metoda *renderScene()* zadužena je za crtanje scene, nakon što je metoda *display* podesila početne postavke. Prve dvije naredbe metode *renderScene()* podešavaju veličinu točke te boju kojom će se popunjavati likovi. Potom slijede dva bloka naredbi omeđena pozivima *glPushMatrix()*; i *glPopMatrix()*; koji crtaju rotirane kvadrate.

Objasnjimo najprije kako se postiže efekti rotacije, da bi nam bilo jasno čemu taj par naredbi. Metoda *drawSquare()* uvijek crta kvadrat stranice 100 čiji je jedan ugao fiksiran u ishodište. S druge pak strane, *OpenGL* ima na raspolaganju funkcije koje koje obavljaju rotaciju, ali uvijek oko ishodišta. Razmislimo li malo, doći ćemo do zaključka da nam ovo baš i ne odgovara. Naime, mi kvadrat želimo rotirati oko njegovog središta, a ne oko vrha koji je fiksiran u ishodište. Stoga *OpenGL*-u trebamo reći da napravi nekoliko koraka, kako slijedi.

1. Kvadrat želimo pomaknuti za -50 po osi x te za -50 po osi y . Ovime će se kvadrat protezati od (-50,-50) pa do (50,50), a njegov centar (sjecište dijagonala) bit će smješten u ishodište.
2. Sada možemo tako pomaknuti kvadrat zarotirati za željeni broj stupnjeva oko osi z (ta je os okomita na ravninu $x-y$ u kojoj se radi crtanje).
3. Nastali kvadrat želimo uvećati za 50% (tj. duljinu stranica pomnožiti faktorom 1.5).
4. Konačno, prikladno zarotirani kvadrat (čije je središte rotacijom i dalje ostalo u ishodištu) pomaknut ćemo do mjesta gdje ga stvarno želimo nacrtati.

Za sve ove operacije *OpenGL* nam nudi odgovarajuće funkcije. Pomak (translacijsku) ćemo obaviti uporabom funkcije `glTranslatef` a rotaciju uporabom funkcije `glRotatef`. Međutim, ove funkcije ne primaju kao argument točku i ne vraćaju transformiranu točku. Prisjetimo se – *OpenGL* je stroj stanja, koji za sve transformacije koristi matrični račun. Pozivanjem navedenih metoda malo po malo modificira se matrica koja će u konačnici obaviti čitav slijed transformacija u samo jednom koraku. Detaljnije o tome kako se ovo radi bit će objašnjeno u jednom od sljedećih poglavlja. Pogledajmo za sada samo jedan jednostavan primjer. U metodi `display` matricu smo resetirali na jediničnu matricu. Ako potom redom pozovemo tri metode `m1()`, `m2()` i `m3()` koje obavljaju neku transformaciju, svaka metoda će trenutnu matricu pomnožiti matricom koja obavlja traženu transformaciju, i rezultat postaviti kao trenutnu matricu. Neka je trenutna matrica označena s \mathbf{M} . Možemo pisati:

$$\begin{aligned} \text{glLoadIdentity}(); &\rightarrow \mathbf{M} = \mathbf{I} \\ \text{m1}(); &\rightarrow \mathbf{M} = \mathbf{I} \cdot \mathbf{M}_1 \\ \text{m2}(); &\rightarrow \mathbf{M} = \mathbf{I} \cdot \mathbf{M}_1 \cdot \mathbf{M}_2 \\ \text{m3}(); &\rightarrow \mathbf{M} = \mathbf{I} \cdot \mathbf{M}_1 \cdot \mathbf{M}_2 \cdot \mathbf{M}_3 \end{aligned}$$

U konačnici, kada će krenuti u obavljanje transformacija, *OpenGL* će trenutnom matricom pomnožiti svaku od točaka \vec{t} koje treba transformirati kako bi dobio transformirane točke \vec{t}' , pa možemo pisati:

$$\vec{t}' = \mathbf{M} \cdot \vec{t} = \mathbf{I} \cdot \mathbf{M}_1 \cdot \mathbf{M}_2 \cdot \mathbf{M}_3 \cdot \vec{t}.$$

Ovaj mali izvod bio nam je potreban kako bismo utvrdili kojim redoslijedom treba pozivati metode *OpenGL*-a. Prisjetimo se svojstava matričnog množenja: ono nije komutativno ali jest asocijativno. Stoga gornju transformaciju možemo zapisati ovako:

$$\vec{t}' = \mathbf{M} \cdot \vec{t} = \mathbf{I} \cdot (\mathbf{M}_1 \cdot (\mathbf{M}_2 \cdot (\mathbf{M}_3 \cdot \vec{t}))).$$

Sada je jasno vidljivo: zadnja definirana transformacija na točku djeluje prva! Potom se nad tako transformiranom točkom primjenjuje predzadnja transformacija, itd. Sjetimo se što smo u našem primjeru htjeli postići: kvadrat smo najprije htjeli pomaknuti po osima x i y za -50 – sada je jasno da će ovo morati posljednji poziv, a ne prvi. Potom smo kvadrat htjeli zarotirati za zadani kut, skalirati s faktorom 1.5 po x - i y -osima, i konačno tako zarotirani i uvećani kvadrat pomaknuti na konačnu poziciju. Imajući to u vidu, korektan redoslijed naredbi prikazan je u sljedećem isječku.

```
glTranslatef( 150.0f, 160.0f, 0.0f );
glScalef( 1.5f, 1.5f, 1.0f );
glRotatef(( float )kut, 0.0f, 0.0f, 1.0f );
glTranslatef(-50.0f, -50.0f, 0.0f );
drawSquare();
```

Crtanje drugog kvadrata, međutim, zahtjeva drugačije transformacije; tamo rotiramo u smjeru suprotnom od smjera kazaljke na satu, a i konačna je pozicija drugačija. Stoga nam treba mehanizam kojim bismo pojedinim transformacijama mogli ograničiti doseg. Upravo u tu svrhu, *OpenGL* ima na raspolaganju matrični stog. Evo koja je ideja. Nakon što sve podesimo u metodi `display()`, kopiju trenutne matrice gurnut ćemo na stog (čime ćemo je sačuvati), promjenit ćemo što treba, nacrtati

prvi kvadrat, i kada smo gotovi, sa stoga ćemo izvaditi očuvanu vrijednost i postaviti je kao trenutnu – efektivno vraćajući stanje na ono koje je bilo prije transformacija prvog kvadrata. Priču ponavljamo i za drugi kvadrat: kopiju trenutne matrice pohranjujemo na stog, radimo crtanje i kada smo gotovi, sa stoga restauriramo pohranjenu vrijednost.

1.5.2 Animacija temeljena na uporabi timera

U prethodnom potpoglavlju uočili smo loše svojstvo uporabe metode idle – brzina animacije ovisna je o brzini sklopolja. Radimo li računalnu igru, ovo nam ponašanje nikako ne odgovara. Naime, ne želimo da se na bržem računalu neprijateljske jedinice kreću brže! Kako bismo riješili taj problem, ažuriranje stanja sustava želimo raditi u dobro definiranim vremenskim trenutcima. U tu svrhu, *GLUT* nam na raspolaganje stavlja mehanizam *timer-a*. Umjesto da koristimo poziv `glutIdleFunc(idle);`, prilikom inicijalizacije programa registrirat ćemo našu funkciju koju će *GLUT* pozvati nakon što istekne određeni vremenski period. Zadatak te funkcije bit će isti kao i zadatak funkcije idle – ažurirati podatke u stanju sustava (konkretno, kut rotacije), zatražiti ponovno iscrtavanje i podesiti *timer* tako da opet pozove tu funkciju.

Metoda koju nam *GLUT* nudi za rad s *timer-ima* zove se `glutTimerFunc`, i prima tri parametra: vrijeme u milisekundama koje govori nakon koliko vremena treba pozvati funkciju; drugi parametar je pokazivač na funkciju koju treba pozvati; konačno, treći argument je broj koji će se predati registriranoj funkciji u trenutku poziva – u ovom primjeru, to nećemo koristiti. *Timer* koji smo dobili na ovaj način poznat je pod nazivom *one-shot timer*, odnosno *timer* koji okida samo jednom. *GLUT* nam ne nudi mogućnost stvaranja *timer-a* koji periodički poziva registriranu funkciju. Stoga je zadatak registrirane funkcije (u našem primjeru metode `animate(...)`) da osim izmjene stanja sustava ponovno zatraži pozivanje uporabom *timer-a*. Čitav kod prikazan je na ispisu 1.3.

Funkcija koju registriramo za poziv iz *timer-a* mora primati jedan argument: cijeli broj (tipa `int`) kojim *timer* predaje vrijednost koja je bila zadana prilikom registracije.

Ispis 1.3: Primjer animacije

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <GL/glut.h>

void reshape(int width, int height);
void display();
void renderScene();
void animate(int value);
void drawSquare();
int kut = 0;

int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE);
    glutInitWindowSize(600, 300);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Primjer animacije");
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutTimerFunc(20, animate, 0);
    glutMainLoop();
}

void animate(int value) {
    kut++;
    if (kut >= 360) kut = 0;
    glutPostRedisplay();
    glutTimerFunc(20, animate, 0);
}

void display() {
```

```

glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
glLoadIdentity();
renderScene();
glutSwapBuffers();
}

void reshape(int width, int height) {
    glDisable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, width-1, 0, height-1, 0, 1);
    glViewport(0, 0, (GLsizei)width, (GLsizei)height);
    glMatrixMode(GL_MODELVIEW);
}

void drawSquare() {
    glBegin(GL_QUADS);
    glVertex2f(-0.0f, 0.0f);
    glVertex2f(100.0f, 0.0f);
    glVertex2f(100.0f, 100.0f);
    glVertex2f(-0.0f, 100.0f);
    glEnd();
}

void renderScene() {
    glPointSize(1);
    glColor3f(1.0f, 0.0f, 0.3f);

    glPushMatrix();
    glTranslatef(-150.0f, -160.0f, 0.0f);
    glScalef(1.5f, 1.5f, 1.0f);
    glRotatef((float)kut, 0.0f, 0.0f, 1.0f);
    glTranslatef(-50.0f, -50.0f, 0.0f);
    drawSquare();
    glPopMatrix();

    glPushMatrix();
    glTranslatef(-400.0f, -160.0f, 0.0f);
    glScalef(1.5f, 1.5f, 1.0f);
    glRotatef(-(float)kut, 0.0f, 0.0f, 1.0f);
    glTranslatef(-50.0f, -50.0f, 0.0f);
    drawSquare();
    glPopMatrix();
}
}

```

Osvojimo se još na dobru praksi rada s *GLUT*-om (i sličnim bibliotekama): nije dobro direktno pozivati metodu za crtanje. Umjesto toga, kada se nešto promjeni stanje "svijeta" koji se prikazuje, bolje je to dojaviti *GLUT*-u pozivom metode `glutPostRedisplay()`, i osloniti se na to da sam *GLUT* pokrenuti ponovno iscrtavanje. Naime, možda se je još nešto promjenilo zbog čega će *GLUT* ionako pokrenuti ponovno crtanje (npr. uništen je dio prozora, došlo je do promjene veličine prozora i sl.).

Drugi detalj na koji bi trebalo paziti jest mjesto gdje se mijenja stanje "svijeta" koji se prikazuje. Ažuriranje toga u metodi koja istovremeno crta nije dobra – naime, mi nemamo kontrolu kada će se i koliko često ta metoda crtati. Čak i kada koristimo mehanizam *timer*-a, *GLUT* zbog različitih razloga metodu za crtanje može pozivati i češće. Kako je najčešće važno da se stanje svijeta mijenja zadanom brzinom, metoda za crtanje je loše mjesto za takve izmjene.

1.6 Ponavljanje

- Što je OpenGL? Što označavaju pojmovi: OpenGL, GLU te GLUT?

2. Koja je razlika između poziva naredbe glutInitDisplayMode(...); s argumentima GLUT_SINGLE odnosno GLUT_DOUBLE?
3. Kako se radi s dvostrukim spremnikom, i što se dobiva njegovom uporabom?
4. Kako izgleda osnovna struktura programa koji koristi GLUT?
5. Ako program pisan uporabom GLUT-a treba reagirati na pritiske tipki na tipkovnici, što je potrebno napraviti?
6. Koje nam primitive za crtanje nudi OpenGL, tj. što može doći kao argument naredbe glBegin (...); ?
7. Koja je razlika između primitiva GL_LINE_LOOP i GL_POLYGON?
8. Koja nam dva tipična scenarija omogućava GLUT kada govorimo o potpori za animaciju?
9. Na koji se način može tražiti od GLUT-a da ponovno pokrene crtanje scene?

Poglavlje 2

Matematičke osnove u računalnoj grafici

Naše upoznavanje s reačunalnom grafikom, kako statičkom tako i interaktivnom, započet ćemo kroz matematički pogled na računalnu grafiku. U okviru ovog poglavlja najprije ćemo se upoznati s notacijom koju ćemo koristiti kroz ovu knjigu. Zatim slijedi upoznavanje s pojmovima *točka*, *vektor*, *pravac* i *ravnina*. Konačno, pred kraj poglavlja upoznat ćemo se s alternativnim načinom prikaza točaka koji se često koristi u računalnoj grafici: prikaz *homogenim* koordinatama.

2.1 Način označavanja

Točka će se obično označavati kao T_X , pri čemu će slovo X biti zamjenjeno u S ako se govori o početnoj točki pravca, u E ako se govori o završnoj točki pravca, odnosno u P ukoliko se govori o proizvoljnoj točki pravca. (Termini početna točka pravca i završna točka pravca biti će jasniji nakon poglavlja 2.3).

Zapis točke T_X po komponentama bit će $(T_{X_1}, T_{X_2}, \dots, T_{X_n})$ pri čemu je T_{X_i} i -ta komponenta točke.

Vektori će se označavati slično kao i točke. Vektor pravca bit će označen kao \vec{v}_p , odnosno po komponentama $(v_{p_1}, v_{p_2}, \dots, v_{p_n})$.

Matrice ćemo označavati velikim štampanim i podebljanim slovima. Npr. karakteristična matrica pravca nosit će oznaku \mathbf{L} .

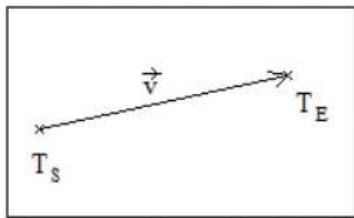
2.2 Točka i vektor

Točka je matematički pojam koji se u pravilu ne definira. No potrebno je uvesti neke osnovne pojmove. Točka je svojstvo prostora (element prostora; osnovna građevna nedjeljiva cjelina). Zbog toga označavanje točke ovisi o prostoru u kojem se ta točka nalazi. Ono osnovno što određuje način označavanja točke jest dimenzionalnost prostora. Svaka točka bit će označena svojim koordinatama, i to s toliko koordinata kolika je dimenzija prostora. Tako će točka u 2D prostoru biti označena pomoću dvije koordinate: x i y . U 3D prostoru točka će biti označena pomoću tri koordinate: x , y i z . U nastavku ćemo točke označavati kao uređene n -torke koordinata, npr. (x, y, z) ili kao matricu $[x \ y \ z]$.

Vektor ćemo promatrati kao usmjerenu dužinu, iako riječ "dužina" možda i nije baš najprikladnija. Naime, vektor će nam obično služiti kao gradijent, tj. pokazatelj koji govori za koliko se nešto mijenja. Vektore ćemo označavati pomoću strelice iznad imena što je uobičajena matematička notacija. Zapis vektora, isto kao i točke ovisi o prostoru u kojem taj vektor opisuje, te će imati onoliko komponenti kolika je dimenzionalnost prostora. Zbog toga ćemo vektore također zapisivati kao uređene n -torke, npr. $\vec{v} = (x, y, z)$. Kako ćemo u nastavku govoriti o računalnoj grafici, vektore ćemo obično razapinjati između dvije točke, i to na slijedeći način (slika 2.1): vektor razapet između točke T_S i točke T_E kreće iz točke T_S , i završava u točki T_E . Ovime je određen smjer vektora, a njegov iznos se računa pomoću relacije:

$$\vec{v} = T_E - T_S \tag{2.1}$$

Pri tome se točke promatraju kao radij-vektori, pa je oduzimanje upravo ono vektorsko: i -ta komponenta rezultata dobije se tako da se oduzmu i -te komponente točaka. Treba imati na umu da



Slika 2.1: Vektor između dviju točaka

vektore možemo zbrajati i oduzimati i da će rezultat biti vektor. Razlika dviju točaka je vektor, dok zbroj dviju točaka matematički nije definiran.

Primjer: 1

Zadane su dvije točke u 3D prostoru: $T_S = (3 \ 1 \ 7)$ i $T_E = (5 \ 0 \ 11)$. Izračunajte vektor koji one razapinju (uz prethodno utvrđenu konvenciju da je T_S početna točka).

Rješenje:

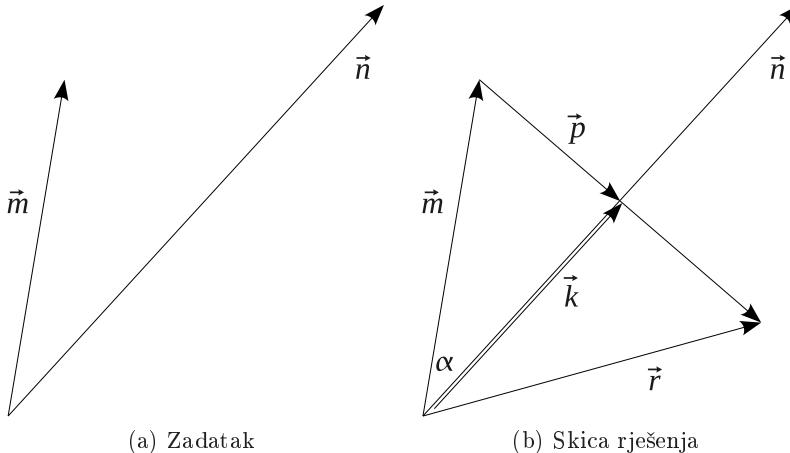
$$\begin{aligned}\vec{v} &= T_E - T_S \\ &= (5 \ 0 \ 11) - (3 \ 1 \ 7) \\ &= (5 - 3 \ 0 - 1 \ 11 - 7) \\ &= (2 \ -1 \ 4)\end{aligned}$$

Primjer: 2

Zadani su proizvoljni vektori \vec{m} i \vec{n} . Pronaći vektor \vec{r} koji čini reflektirani vektor vektora \vec{m} s obzirom na vektor \vec{n} . Provjerite rezultat na konkretnom primjeru $\vec{n} = (3 \ 3)$ i $\vec{m} = (2 \ 3)$.

Rješenje:

Prvi korak u rješavanju ovog zadatka bit će izrada kvalitetne skice. Problem je prikazan na slici 2.2a. Pomoćni vektori kao i traženi vektor \vec{r} prikazani su na slici 2.2b.



Slika 2.2: Pronalazak reflektiranog vektora

Pogledamo li sliku 2.2b, vidimo da možemo pisati:

$$\vec{r} = \vec{m} + 2 \cdot \vec{p}.$$

Stoga ćemo se najprije fokusirati na pronalazak vektora \vec{p} . Vektor \vec{m} projiciran na vektor \vec{n} označen je s \vec{k} . To je vektor koji je kolinearan s vektorom \vec{n} , pri čemu mu je magnituda određena izrazom $||\vec{m}|| \cdot \cos(\alpha)$, pa možemo pisati:

$$\vec{k} = \frac{\vec{n}}{||\vec{n}||} \cdot ||\vec{m}|| \cdot \cos(\alpha).$$

Kosinus kuta možemo izračunati direktno iz vektora \vec{m} i \vec{n} . Kako ti vektori nisu normirani, sjetimo se da vrijedi opći izraz:

$$\cos(\alpha) = \frac{\vec{m} \cdot \vec{n}}{\|\vec{m}\| \cdot \|\vec{n}\|}.$$

Uvrštanjem dalje slijedi:

$$\vec{k} = \frac{\vec{n}}{\|\vec{n}\|} \cdot \|\vec{m}\| \cdot \frac{\vec{m} \cdot \vec{n}}{\|\vec{m}\| \cdot \|\vec{n}\|} = \frac{\vec{n}}{\|\vec{n}\|} \cdot \frac{\vec{m} \cdot \vec{n}}{\|\vec{n}\|}.$$

Sada možemo doći do traženog vektora \vec{p} . Iz slike 2.2b slijedi da je $\vec{m} + \vec{p} = \vec{k}$, pa je stoga $\vec{p} = \vec{k} - \vec{m}$. Uvrštanjem u izraz za \vec{r} slijedi:

$$\begin{aligned} \vec{r} &= \vec{m} + 2 \cdot \vec{p} \\ &= \vec{m} + 2 \cdot (\vec{k} - \vec{m}) \\ &= \vec{m} + 2 \cdot \vec{k} - 2 \cdot \vec{m} \\ &= 2 \cdot \vec{k} - \vec{m} \\ &= 2 \cdot \frac{\vec{n}}{\|\vec{n}\|} \cdot \frac{\vec{m} \cdot \vec{n}}{\|\vec{n}\|} - \vec{m} \end{aligned}$$

Primjenimo to na konkretnom primjeru. Kao reflektirani vektor morali bismo dobiti vektor $\vec{r} = (3 \ 2)$, pa provjerimo to. Norma vektora \vec{n} je $\sqrt{3^2 + 3^2} = \sqrt{18}$.

$$\begin{aligned} \vec{r} &= 2 \cdot \frac{\vec{n}}{\|\vec{n}\|} \cdot \frac{\vec{m} \cdot \vec{n}}{\|\vec{n}\|} - \vec{m} \\ &= 2 \cdot \frac{(3 \ 3)}{\sqrt{18}} \cdot \frac{(3 \ 2) \cdot (3 \ 3)}{\sqrt{18}} - (2 \ 3) \\ &= 2 \cdot \frac{(3 \ 3)}{\sqrt{18}} \cdot \frac{3 \cdot 2 + 3 \cdot 3}{\sqrt{18}} - (2 \ 3) \\ &= 2 \cdot \frac{(3 \ 3)}{18} \cdot 15 - (2 \ 3) \\ &= (5 \ 5) - (2 \ 3) \\ &= (3 \ 2) \end{aligned}$$

2.3 Pravac

Pravac je također pojam koji se ne definira, no najopćenitije govoreći može se reći da je pravac skup točaka. U tom smislu, točke koje pripadaju pravcu možemo matematički opisati na nekoliko načina.

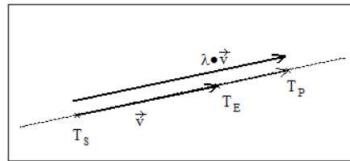
2.3.1 Jednadžba pravca

Kada govorimo o jednadžbi pravca, ideja je pronaći takav matematički izraz (jednadžbu) koji će vrijediti za sve točke koje pripadaju pravcu. Broj načina na koji ovo možemo napraviti ovisi o dimenzionalnosti prostora u kojem promatramo pravac. Ako se radi o 2D prostoru, uobičajeno možemo napisati *implicitni oblik* jednadžbe pravca, najčešće možemo napisati i *eksplicitni oblik* jednadžbe pravca, a uvjek možemo napisati jednadžbu pravca u *parametarskom obliku*. U 3D prostoru, broj mogućnosti se smanjuje, i tu uvjek koristimo parametarski oblik jednadžbe pravca. Pa upoznajmo se najprije s tim oblikom.

Parametarski oblik jednadžbe pravca

Ideja parametarskog zapisa krivulje – pa tako i pravca kao jedne specifične vrste krivulja – jest pomoću konačnog broja parametara opisati sve točke koje pripadaju dotičnoj krivulji. Za opis pravca dovoljan je samo jedan parametar, s obzirom da su promjene po svim koordinatama linearno vezane. Jednadžbu možemo dobiti iz slike 2.3.

$$T_P = (T_E - T_S) \cdot \lambda + T_S = \vec{v}_p \cdot \lambda + T_S \quad (2.2)$$



Slika 2.3: Pravac dobiven skaliranjem vektora

Formula zapravo kaže: vektor \vec{v}_p skaliran parametrom λ dodaj točki T_S i dobit ćeš jednu točku pravca. Ovo napravi za svaki $\lambda \in \mathbb{R}$ i dobit ćeš sve točke pravca.

Pojam "skalirati" parametrom λ znači svaku komponentu vektora \vec{v}_p pomnožiti realnim brojem λ . Time se norma vektora \vec{v}_p povećava λ puta, te vrijedi:

$$\|\lambda \cdot \vec{v}_p\| = \lambda \cdot \|\vec{v}_p\|$$

Npr. prema slici 2.3, ako točki T_S dodamo \vec{v}_p uz $\lambda = 1$ doći ćemo do točke T_E . No ako je λ veći od jedan, tada ćemo doći do neke točke koja se krenuvši od točke T_S nalazi iza točke T_E . Ako je $\lambda \in <0, 1>$, dolazimo do točaka koje su između točke T_S i točke T_E , dok za $\lambda < 0$ dolazimo do točaka koje su ispred točke T_S . Primjerice, za $\lambda = 0.5$, dobit ćemo točku koja se nalazi točno na pola puta između T_S i T_E , a za $\lambda = -1$ dolazimo do točke koja je ispred točke T_S i to na jednakoj udaljenosti od točke T_S kao što je točka T_E udaljena od točke T_S .

U jednadžbi pravca koju smo prethodno napisali krije se zapravo onoliko jednadžbi koliko točke imaju komponenata. Važno je za uočiti da ovakav zapis pravca donekle i usmjerava pravac. Naime, porastom parametra λ točka T_P giba se u smjeru vektora \vec{v}_p odnosno od točke T_S prema točki T_E i dalje. Ovakav oblik zapisa pravca pogodan je i za određivanje gdje se neka zadana točka pravca nalazi u odnosu na točke T_S i T_E . Vrijedi slijedeće. Ako je za zadalu točku T_P parametar λ :

- negativan, točka T_P je ispred točke T_S ;
- nula, točka T_P se upravo poklapa s točkom T_S ;
- pozitivan i manji od 1, točka T_P je između točaka T_S i T_E ;
- jednak 1, točka T_P se upravo poklapa s točkom T_E ; te
- pozitivan i veći od 1, točka T_P je iza točke T_E .

Raspisemo li jednadžbu (2.2) po komponentama, dobivamo:

$$(T_{P_1}, T_{P_2}, \dots, T_{P_n}) = (v_{p_1}, v_{p_2}, \dots, v_{p_n}) \cdot \lambda + (T_{S_1}, T_{S_2}, \dots, T_{S_n}) \quad (2.3)$$

što možemo prikazati i u matričnom obliku:

$$(T_{P_1}, T_{P_2}, \dots, T_{P_n}) = [\lambda \ 1] \cdot \begin{bmatrix} v_{p_1} & v_{p_2} & \dots & v_{p_n} \\ T_{S_1} & T_{S_2} & \dots & T_{S_n} \end{bmatrix} = [\lambda \ 1] \cdot \mathbf{L} \quad (2.4)$$

gdje slovo \mathbf{L} stoji za *karakterističnu matricu* pravca. Kako jednadžba mora biti zadovoljena za svaku komponentu zasebno, gornji zapis jednadžbe raspada se na n jednadžbi. Matrični oblik zapisa u računalnoj grafici iznimno je važan jer je pogodan za zapis na računalu. Matematičke pojmove koje ovdje obrađujemo trebat ćemo implementirati na računalu. Kako pri toj implementaciji nije pogodno koristiti nizove varijabli, već polje tako i u primjeru karakteristične matrice pravca, pravac \mathbf{L} će biti zapisan kao 2D polje podataka.

Računanje jednadžbe pravca kroz dvije točke

Neka su zadane dvije točke kroz koje prolazi pravac. Točka T_S neka bude početna točka pravca, a točka T_E neka bude završna točka pravca. Jednadžbu pravca kroz te dvije točke možemo izračunati temeljem izraza (2.2):

$$T_P = (T_E - T_S) \cdot \lambda + T_S = \vec{v}_p \cdot \lambda + T_S.$$

Jedina nepoznanica je vrijednost vektora smjera \vec{v}_p , no taj vektor možemo odrediti direktno iz izraza (2.2), čime dobivamo:

$$v_p = T_E - T_S. \quad (2.5)$$

Primjer: 3

Pravac u 2D prostoru zadan je dvjema točkama: $T_S = (1 \ 1)$ i $T_E = (3 \ 2)$. Kako glasi parametarski oblik jednadžbe tog pravca?

Rješenje:

Prema (2.5) računamo vektor smjera pravca:

$$\begin{aligned} \vec{v} &= T_E - T_S \\ &= (3 \ 2) - (1 \ 1) \\ &= (3 - 1 \ 2 - 1) \\ &= (2 \ 1) \end{aligned}$$

Označimo li prvu koordinatu s x a drugu s y , izračunati vektor smjera nam govori da svaki puta kada se y poveća za 1, x se poveća za 2. Iskoristimo ovaj zaključak da nabrojimo nekoliko točaka koje pripadaju pravcu. Znamo da točka $(1 \ 1)$ pripada pravcu. Prema uočenoj pravilnosti, povećamo li y za 1 i x za 2, dolazimo do točke $(3 \ 2)$ – uočimo da je ovo dodavanje zapravo odgovaralo dodavanju čitavog vektora v_p , odnosno izraza $v_p \cdot \lambda$ uz $\lambda = 1$ točki T_S , čime smo dobili upravo točku T_E , što je u skladu s prethodno utvrđenim ponašanjem. Uvećamo li ponovno koordinate točke, stižemo do točke $(5 \ 3)$ koja također pripada pravcu.

Traženi parametarski oblik jednadžbe pravca tada je:

$$T_P = (2 \ 1) \cdot \lambda + (1 \ 1).$$

Evo još jednog načina kako se može doći do parametarske jednadžbe pravca ako znamo dvije točke koje mu pripadaju. Krenimo iz parametarskog oblika jednadžbe pravca zapisanog matrično. Prisjetimo se izraza (2.4):

$$(T_{P_1}, T_{P_2}, \dots, T_{P_n}) = [\lambda \ 1] \cdot \mathbf{L}$$

U ovom izrazu jedina je nepoznanica matrica \mathbf{L} koju treba odrediti. Budući da pravac kojeg tražimo za neku vrijednost parametra λ prolazi kroz točku T_S , a za neku drugu vrijednost parametra λ prolazi kroz točku T_E , ostavljena nam je sloboda izbora da sami odlučimo za koje će se to vrijednosti parametra λ dogoditi. Zbog toga ćemo se odlučiti za uobičajeni izbor: neka pravac prođe kroz točku T_S za $\lambda = 0$, a kroz točku T_E za $\lambda = 1$. Tada možemo pisati:

$$T_S = [0 \ 1] \cdot \mathbf{L}$$

$$T_E = [1 \ 1] \cdot \mathbf{L}$$

gdje sada T_S i T_E promatramo kao jednoretčane matrice s onoliko stupaca koliko točke imaju koordinata. Ove dvije jednadžbe možemo stopiti u jednu matričnu jednadžbu:

$$\begin{bmatrix} T_S \\ T_E \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \mathbf{L}$$

Treba uočiti da T_S i T_E na ljevoj strani jednadžbe predstavljaju matrice $1 \times n$ pa matrica na ljevoj strani nije 2×1 već $2 \times n$. Kako je u jednadžbi sve poznato osim matrice \mathbf{L} , množeći jednadžbu s lijeva inverznom matricom uz \mathbf{L} dobiva se:

$$\mathbf{L} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} T_S \\ T_E \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} T_S \\ T_E \end{bmatrix} = \begin{bmatrix} T_E - T_S \\ T_S \end{bmatrix} \quad (2.6)$$

pri čemu je matrica \mathbf{L} dimenzija $2 \times n$.

Primjer: 4

Pravac u 2D prostoru zadan je dvjema točkama: $T_S = (1 \ 1)$ i $T_E = (3 \ 2)$. Kako glasi parametarski oblik jednadžbe tog pravca u matričnom zapisu? Za koji se vrijednost parametra dobiva točka $(-3 \ -1)$?

Rješenje:

Matricu \mathbf{L} izračunat ćemo prema izrazu (2.6):

$$\mathbf{L} = \begin{bmatrix} T_E - T_S \\ T_S \end{bmatrix} = \begin{bmatrix} 3 - 1 & 2 - 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

Traženi oblik jednadžbe pravca tada glasi:

$$(x \ y) = [\lambda \ 1] \cdot \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

Kako znamo da tražena točka $(-3 \ -1)$ leži na pravcu, možemo pisati:

$$(-3 \ -1) = [\lambda \ 1] \cdot \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

Slijedi:

$$[\lambda \ 1] = [-3 \ -1] \cdot \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}^{-1} = [-3 \ -1] \cdot \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} = [-2 \ 1].$$

Dakle, tražena vrijednost parametra je $\lambda = -2$.

Ovim korakom smo u osnovi rješavali sustav od dvije jednadžbe s jednom nepoznanicom, kojeg možemo dobiti i izravno iz prethodnog koraka, odnosno:

$$-3 = 2\lambda + 1$$

$$-1 = \lambda + 1$$

Ako u obje jednadžbe dobijemo istu vrijednost za nepoznanicu, znači da točka leži na pravcu, u protivnom točka ne leži na pravcu, odnosno ne postoji jedinstveni parametar λ za koji to vrijedi. Valja napomenuti da u jednadžbi (2.4) točka na pravcu T_S može biti bilo koja točka, bitno je da je ta točka na pravcu. Vektor smjera pravca \vec{v}_p isto tako može biti bilo koji vektor koji ima smjer dotičnog pravca. Drugim riječima, isti pravac možemo zapisati na beskonačno mnogo načina. Radi li se upravo o nekom zadanom pravcu provjerit ćemo tako da provjerimo kolinearnost vektora pravaca koji ih određuju, te pripadnosti točke promatranoj pravca zadanom pravcu.

2.3.2 Posebni slučajevi jednadžbe pravca

2D prostor

U 2D prostoru svaka točka ima po dvije komponente, pa izraz (2.3):

$$(T_{P_1}, T_{P_2}, \dots, T_{P_n}) = (v_{p_1}, v_{p_2}, \dots, v_{p_n}) \cdot \lambda + (T_{S_1}, T_{S_2}, \dots, T_{S_n})$$

prelazi u:

$$(T_{P_1}, T_{P_2}) = (v_{p_1}, v_{p_2}) \cdot \lambda + (T_{S_1}, T_{S_2}) \quad (2.7)$$

Umjesto oznaka T_{P_1} i T_{P_2} mogli smo koristi i oznake x_p, y_p , a umjesto oznaka T_{S_1} i T_{S_2} mogli smo koristi i oznake x_s, y_s , a umjesto oznaka v_{p_1} i v_{p_2} mogli smo koristi i oznake v_x, v_y . Tada bismo dobili prepoznatljiv oblik izraza:

$$(x_p, y_p) = (v_x, v_y) \cdot \lambda + (x_s, y_s).$$

Jednadžbu (2.7) možemo raspisati po komponentama i tako dobiti dvije jednadžbe:

$$T_{P_1} = v_{p_1} \cdot \lambda + T_{S_1}$$

$$T_{P_2} = v_{p_2} \cdot \lambda + T_{S_2}$$

Eliminacijom parametra λ dobiva se *eksplicitni oblik* jednadžbe pravca:

$$T_{P_2} - T_{S_2} = \frac{v_{p_2}}{v_{p_1}} (T_{P_1} - T_{S_1}) \quad (2.8)$$

Vektor smjera \vec{v}_p definiran je izrazom (2.5), što po komponentama možemo raspisati kao:

$$v_{p_1} = T_{E_1} - T_{S_1},$$

$$v_{p_2} = T_{E_2} - T_{S_2}.$$

Uvrštavanjem ovih izraza u eksplisitni oblik – izraz (2.7) – dobiva se:

$$T_{P_2} - T_{S_2} = \frac{T_{E_2} - T_{S_2}}{T_{E_1} - T_{S_1}} (T_{P_1} - T_{S_1}), \quad (2.9)$$

što nakon promjene imena odgovarajućih varijabli u "školska" imena daje:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1} (x - x_1). \quad (2.10)$$

Pomnožimo li izraz (2.9) sa zajedničkim nazivnikom i potom malo preuređimo, dobit ćemo *implicitni oblik* jednadžbe pravca:

$$(T_{E_2} - T_{S_2}) \cdot T_{P_1} - (T_{E_1} - T_{S_1}) \cdot T_{P_2} - (T_{E_2} - T_{S_2}) \cdot T_{S_1} + (T_{E_1} - T_{S_1}) \cdot T_{S_2} = 0. \quad (2.11)$$

Koristimo li opet uobičajenu školsku notaciju, izraz prelazi u:

$$(y_2 - y_1) \cdot x - (x_2 - x_1) \cdot y - (y_2 - y_1) \cdot x_1 + (x_2 - x_1) \cdot y_1 = 0. \quad (2.12)$$

Općenito implicitni oblik zapisujemo kao:

$$a \cdot T_{P_1} + b \cdot T_{P_2} + c = 0, \quad (2.13)$$

odnosno u uobičajenoj notaciji, prepoznatljiv nam je oblik:

$$a \cdot x + b \cdot y + c = 0. \quad (2.14)$$

Od zanimljivijih oblika vrijedno je još spomenuti i *segmentni oblik* jednadžbe pravca, koji direktno daje odsječke pravca na obje osi. Dobije se svođenjem implicitnog oblika na oblik:

$$\frac{T_{P_1}}{l_x} + \frac{T_{P_2}}{l_y} = 1. \quad (2.15)$$

Pri tome vrijednosti l_x i l_y predstavljaju odsječke na obje osi. Uočimo međutim da ovaj oblik ne postoji za sve pravce u 2D prostoru – primjerice, pokušajte ga napisati za pravac koji je paralelan s osi x ili paralelan s osi y . Iz eksplisitnog se oblika dobije:

$$\frac{T_{P_1}}{\frac{T_{S_2}v_{p_1}-T_{S_1}v_{p_2}}{-v_{p_2}}} + \frac{T_{P_2}}{\frac{T_{S_2}v_{p_1}-T_{S_1}v_{p_2}}{v_{p_1}}} = 1. \quad (2.16)$$

3D prostor

U 3D prostoru pravac nije moguće prikazati jednadžbom u eksplisitnom obliku. Naime, parametarski oblik jednadžbe pravca definiran izrazom (2.3):

$$(T_{P_1}, T_{P_2}, \dots, T_{P_n}) = (v_{p_1}, v_{p_2}, \dots, v_{p_n}) \cdot \lambda + (T_{S_1}, T_{S_2}, \dots, T_{S_n})$$

u tom slučaju prelazi u:

$$(T_{P_1}, T_{P_2}, T_{P_3}) = (v_{p_1}, v_{p_2}, v_{p_3}) \cdot \lambda + (T_{S_1}, T_{S_2}, T_{S_3}) \quad (2.17)$$

pri čemu se svaka točka (ili vektor) opisuje pomoću tri koordinate. Ova jednadžba ekvivalent je tri jednadžbe (po jedna za svaku koordinatu), te je parametar λ nemoguće eliminirati tako da se dobije jedna jednadžba. No eliminacijom parametra λ iz dva para jednadžbi moguće je dobiti implicitni oblik jednadžbe pravca u tri dimenzije:

$$\frac{T_{P_1} - T_{S_1}}{v_{p_1}} = \frac{T_{P_2} - T_{S_2}}{v_{p_2}} = \frac{T_{P_3} - T_{S_3}}{v_{p_3}} \quad (2.18)$$

odnosno:

$$\frac{T_{P_1} - T_{S_1}}{T_{E_1} - T_{S_1}} = \frac{T_{P_2} - T_{S_2}}{T_{E_2} - T_{S_2}} = \frac{T_{P_3} - T_{S_3}}{T_{E_3} - T_{S_3}}. \quad (2.19)$$

Uz školske oznake točaka, izraz glasi:

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} = \frac{z - z_1}{z_2 - z_1}.$$

Poteškoće

Kada zadajemo jednadžbu pravca, želimo biti u stanju pomoći nje prikazati sve moguće pravce. Krenemo li od segmentnog oblika (uz ograničenje razmatranja na zapisa pravca samo u 2D prostoru), vidimo da on ne može prikazati niti jedan pravac paralelan s bilo kojom osi. Eksplisitni oblik ima poteškoća s pravcima paralelnim s ordinatom. Jedini oblik koji nema problema je implicitni oblik. Međutim, implicitni oblik pravca za primjenu u računalima nije praktičan. Zbog toga se kao relativno dobar oblik pokazuje parametarski oblik. Dodatan "plus" ovom obliku nosi i mogućnost matričnog zapisa, koji je idealan za primjenu u računalima, i obilno se koristi u grafičkom sklopolju.

2.4 Homogeni prostor. Homogene koordinate.

2.4.1 Ideja

Dijeljenje s nulom jedan je od čestih problema pri geometrijskim proračunima. Npr. traženje sjecišta dva paralelna pravca na uobičajeni način rezultirat će dijeljenjem s nulom budući da se sjecište nalazi u beskonačnosti. Da bi se ovakvi problemi izbjegli, uvodi se *homogeni prostor* i homogene koordinate. Evo razmišljanja. Ako je neka od koordinata jednakonačno, to znači da se može dobiti dijeljenjem s nulom. Da bi se ovo izbjeglo, potrebno je sve koordinate napisati u obliku razlomka:

$$T_{P_1} = \frac{T_{Ph_1}}{h}, T_{P_2} = \frac{T_{Ph_2}}{h}, \quad (2.20)$$

pri čemu su T_{P_1} i T_{P_2} standardne koordinate (kažemo još i koordinate u radnom prostoru), a T_{Ph_1} i T_{Ph_2} su homogene koordinate. Uvrštavanjem ovih izraza u implicitni oblik jednadžbe pravca dobivamo:

$$a \cdot \frac{T_{Ph_1}}{h} + b \cdot \frac{T_{Ph_2}}{h} + c = 0$$

odnosno

$$a \cdot T_{Ph_1} + b \cdot T_{Ph_2} + c \cdot h = 0. \quad (2.21)$$

U ovom slučaju koordinate T_{Ph_1} i T_{Ph_2} nikada neće biti jednake beskonačno. Općenito se može reći ovako. Zadana je neka proizvoljna točka T u n -dimenzijskom prostoru. Ta točka T u homogenom će se prostoru opisivati pomoću $n+1$ koordinate. Pri tome će jednoj konkretnoj točki radnog prostora biti pridruženo beskonačno homogenih točaka. Npr. točka $(12, 24, 12)$ u radnom prostoru u tri dimenzije imat će u homogenom prostoru zapise: $(12, 24, 12, 1)$, $(6, 12, 6, 0.5)$, $(24, 48, 24, 2)$, ... Naime, uočite da se svaka od tih homogenih točaka nakon dijeljenja s homogenim parametrom pretvara u istu točku radnog prostora: $(12, 24, 12)$.

Poseban slučaj je zapis beskonačnosti: ako je homogena koordinata h točke T jednaka 0, točka leži u beskonačnosti. Ako dobijemo da se dva pravca sijeku u homogenoj točki $(6, 6, 4, 2)$, to znači da se u radnom prostoru sijeku u točki $(3, 3, 2)$. Ako se pak sijeku u homogenoj točki $(10, 10, 3, 0)$, tada su ti pravci u radnom prostoru paralelni i nemaju sjecišta. Valja napomenuti da kombinacija $(0, 0, 0, 0)$ nije dozvoljena.

Ponovimo ukratko najvažnije. Ako je točka u radnom prostoru zadana kao:

$$T_P = (T_{P_1}, T_{P_2}, \dots, T_{P_n})$$

tada ta točka u homogenom prostoru ima zapis:

$$T_Ph = (T_{Ph_1}, T_{Ph_2}, \dots, T_{Ph_n}, h)$$

pri čemu je definirana veza:

$$T_{P_i} = \frac{T_{Ph_i}}{h}, \text{ odnosno } T_{Ph_i} = T_{P_i} \cdot h. \quad (2.22)$$

2.4.2 Jednadžba 2D pravca u homogenom prostoru

U poglavlju 2.3.2 dana je jednadžba pravca u 2D u radnom prostoru (izraz (2.7)):

$$(T_{P_1}, T_{P_2}) = (v_{p_1}, v_{p_2}) \cdot \lambda + (T_{S_1}, T_{S_2})$$

Uvrštavanjem izraza (2.22) u tu jednadžbu dobiva se:

$$(T_{P_1}, T_{P_2}, h_P) = (v_{p_1}, v_{p_2}, v_p) \cdot \lambda + (T_{S_1}, T_{S_2}, h_S). \quad (2.23)$$

Pri tome treba obratiti pažnju na h_P i h_S komponente točaka. Nepažljiva uporaba gornje relacije može dovesti do nepoželjnih posljedica (vidi zadatak ??).

2.4.3 Jednadžba 3D pravca u homogenom prostoru

U poglavlju 2.3.2 dana je jednadžba pravca u 3D u radnom prostoru (izraz (2.17)):

$$(T_{P_1}, T_{P_2}, T_{P_3}) = (v_{p_1}, v_{p_2}, v_{p_3}) \cdot \lambda + (T_{S_1}, T_{S_2}, T_{S_3}).$$

Proširivanjem tog izraza uz izraz (2.22) dolazi se do:

$$(T_{P_1}, T_{P_2}, T_{P_3}, h_P) = (v_{p_1}, v_{p_2}, v_{p_3}, v_p) \cdot \lambda + (T_{S_1}, T_{S_2}, T_{S_3}, h_s) \quad (2.24)$$

pri tome treba obratiti pažnju na h_P i h_S komponente točaka. Nepažljiva uporaba gornje relacije može dovesti do nepoželjnih posljedica (vidi zadatak ??).

2.4.4 Alternativni oblik jednadžbe pravca u 2D u homogenom prostoru

Krenemo li od implicitnog oblika jednadžbe pravca u 2D (izraz 2.13):

$$a \cdot T_{P_1} + b \cdot T_{P_2} + c = 0$$

i uvrstimo li u jednadžbu izraz za homogene koordinate 2.22

$$T_{P_i} = \frac{T_{Ph_i}}{h}, \text{ odnosno } T_{Ph_i} = T_{P_i} \cdot h$$

dolazimo do jednadžbe:

$$a \cdot \frac{T_{Ph_1}}{h} + b \cdot \frac{T_{Ph_2}}{h} + c = 0$$

odnosno nakon sređivanja:

$$a \cdot T_{Ph_1} + b \cdot T_{Ph_2} + c \cdot h = 0 \quad (2.25)$$

Ovaj se izraz može prikazati i u matričnom obliku, što je dosta čest slučaj. Tada on glasi:

$$\begin{bmatrix} T_{Ph_1} & T_{Ph_2} & h \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = 0 \quad (2.26)$$

Uvođenjem uobičajenih oznaka

$$T_P = [T_{Ph_1} \ T_{Ph_2} \ h], \quad \mathbf{G} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \quad (2.27)$$

dolazi se do jednadžbe:

$$T_P \cdot \mathbf{G} = 0. \quad (2.28)$$

Ovdje se može razmotriti još jedan interesantan slučaj. Ako odaberemo točku T_P takvu da leži na pravcu, tada će gornja relacija biti zadovoljena. No što ako ubacimo proizvoljnu točku u jednadžbu pravca? Gornja relacija očito neće biti jednak nuli, i na temelju tog rezultata možemo definirati odnos točke i pravca prema slijedećem kriteriju:

$$T_P \cdot \mathbf{G} \begin{cases} > 0, & T_P \text{ je iznad pravca,} \\ = 0, & T_P \text{ je na pravcu,} \\ < 0, & T_P \text{ je ispod pravca.} \end{cases} \quad (2.29)$$

Ovakva interpretacija vrijedi ako smo matricu \mathbf{G} izračunali prema izrazu (2.27). Da bismo dobili bolji osjećaj što nam ovo pravilo zapravo govori, zamislite da se fizički krećete po zadanom pravcu, i to na način da ste krenuli od točke T_S i hodate prema točki T_E . U tom će slučaju za sve točke ravnine koje ne pripadaju pravcu a nalaze Vam se s lijeve strane vrijediti $T_P \cdot \mathbf{G} > 0$; za te točke kažemo da su iznad pravca po kojem se krećete. Za sve točke ravnine koje ne pripadaju pravcu a nalaze Vam se s desne strane vrijediti će $T_P \cdot \mathbf{G} < 0$; za te točke kažemo da su ispod pravca po kojem se krećete. Određivanje odnosa točke i pravca na ovaj način koristit ćemo kasnije u nizu algoritama (primjerice, kod ispitivanja odnosa točke i poligona), pa ja važno razumjeti o čemu se tu radi.

Poznavanjem dviju točaka T_A i T_B na jednostavan se način može doći do pravca ako se napravi \times -proizvod tih točaka. Neka su točke T_A i T_B zadane na sljedeći način:

$$T_A = (T_{A_1} \ T_{A_2} \ h_A) \text{ i } T_B = (T_{B_1} \ T_{B_2} \ h_B).$$

Njihov \times produkt tada je:

$$T_A \times T_B = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ T_{A_1} & T_{A_2} & h_A \\ T_{B_1} & T_{B_2} & h_B \end{vmatrix} = \vec{i} \cdot (T_{A_2}h_B - T_{B_2}h_A) - \vec{j} \cdot (T_{A_1}h_B - T_{B_1}h_A) + \vec{k} \cdot (T_{A_1}T_{B_2} - T_{A_2}T_{B_1}).$$

Ovo možemo dalje raspisati kao:

$$\begin{aligned} T_A \times T_B &= \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} \cdot \begin{bmatrix} T_{A_2}h_B - T_{B_2}h_A \\ -(T_{A_1}h_B - T_{B_1}h_A) \\ T_{A_1}T_{B_2} - T_{A_2}T_{B_1} \end{bmatrix} \\ &= \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} \cdot \mathbf{G}. \end{aligned}$$

Potrebno je skrenuti pažnju da je bitan redoslijed točaka pri izračunu \times produkta. Ako zamijenimo redoslijed točaka tako da načinimo produkt $T_A \times T_B$, dobit ćemo isti pravac, no kriterij koji određuje što je iznad, a što ispod postaje upravo oburnuti. Isti učinak postići ćemo ako jednadžbu pravca pomnožimo s (-1) . Slično se poznavanjem dvaju pravaca može odrediti točka u kojoj se oni sijeku ako se napravi \times produkt. Neka su pravci određeni s:

$$\begin{aligned} \mathbf{G}_1 &= \begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix}, \mathbf{G}_2 = \begin{bmatrix} a_2 \\ b_2 \\ c_2 \end{bmatrix}. \\ G_1^T \times G_2^T &= \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{vmatrix} = \vec{i} \cdot (b_1c_2 - b_2c_1) - \vec{j} \cdot (a_1c_2 - a_2c_1) + \vec{k} \cdot (a_1b_2 - a_2b_1). \end{aligned}$$

Ovo možemo dalje raspisati kao:

$$\begin{aligned} G_1^T \times G_2^T &= \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} \cdot \begin{bmatrix} b_1c_2 - b_2c_1 \\ -(a_1c_2 - a_2c_1) \\ a_1b_2 - a_2b_1 \end{bmatrix} \\ &= T_P^T. \end{aligned}$$

Operacija transponiranja nužna je ako se žele napisati jednadžbe u skladu s pravilima matematike. Ovdje se može prodiskutirati opravdanost uvođenja homogenih koordinata. Naime, zahvaljujući homogenim koordinatama, matrični račun ne daje niti jedno dijeljenje. Ovo znači da će i paralelni pravci imati sjecište. Doista, ako su pravci paralelni, tada su im koeficijenti smjera jednaki i treća komponenta točke T_P iznosit će nula, što prema definiciji znači točku u beskonačnosti. Evo jednostavnog dokaza. Jednadžba pravca u 2D s homogenim koordinatama glasi:

$$a \cdot T_{Ph_1} + b \cdot T_{Ph_2} + c \cdot h = 0$$

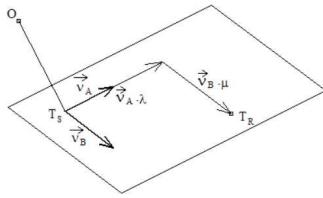
što se može napisati kao:

$$T_{Ph_2} = -\frac{a}{b} \cdot T_{Ph_1} - \frac{c}{b} \cdot h = -k \cdot T_{Ph_1} - \frac{c}{b} \cdot h$$

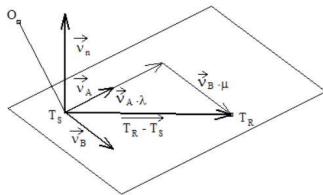
gdje je k koeficijent smjera pravca. Ako pravci G_1 i G_2 imaju jednake koeficijente smjera, to prema gornjoj relaciji znači da vrijedi:

$$-\frac{a_1}{b_1} = -\frac{a_2}{b_2} \Rightarrow a_1b_2 = a_2b_1 \Rightarrow a_1b_2 - a_2b_1 = 0$$

kako je poslijednja relacija upravo jednadžba zadnje koordinate točke sjecišta, slijedi da je uz paralelne pravce zadnja koordinata (tj. homogeni faktor) točke sjecišta jednaka 0.



Slika 2.4: Ravnina određena točkom i dvama vektorima



Slika 2.5: Ravnina i njezina normala

2.5 Ravnina

2.5.1 Jednadžba ravnine

Ravnina se kao pojam u općem smislu može definirati u n -dimenzijskom prostoru. Međutim, kako se u računalnoj grafici koriste jedino ravnine u 3D prostoru, razmatranja ćemo ograničiti na taj tip. U 3D prostoru ravnina je određena s dva vektora koji leže u njoj i nisu kolinearni, te jednom točkom ravnine, koja fiksira te vektore. Krenuvši ovakvom definicijom dolazimo vrlo jednostavno do jednadžbe ravnine u parametarskom obliku:

$$T_R = \vec{v}_A \cdot \lambda + \vec{v}_B \cdot \mu + T_S. \quad (2.30)$$

ili riječima:

krenuvši od točke T_S svaka točka ravnine može se dobiti pomakom za $\vec{v}_A \cdot \lambda$ i $\vec{v}_B \cdot \mu$, pri čemu su λ i μ realni parametri. Pri tome točka T_R predstavlja bilo koju točku ravnine. Slika 2.3 ovo jasno ilustrira.

Iraz (2.30) ekvivalentan je sustavu od tri jednadžbe:

$$T_{R_1} = v_{A_1} \cdot \lambda + v_{B_1} \cdot \mu + T_{S_1},$$

$$T_{R_2} = v_{A_2} \cdot \lambda + v_{B_2} \cdot \mu + T_{S_2},$$

$$T_{R_3} = v_{A_3} \cdot \lambda + v_{B_3} \cdot \mu + T_{S_3}.$$

Ovo se može prikazati i matričnim zapisom:

$$T_R = [\begin{array}{ccc} T_{R_1} & T_{R_2} & T_{R_3} \end{array}] = [\begin{array}{ccc} \lambda & \mu & 1 \end{array}] \cdot [\begin{array}{ccc} v_{A_1} & v_{A_2} & v_{A_3} \\ v_{B_1} & v_{B_2} & v_{B_3} \\ T_{S_1} & T_{S_2} & T_{S_3} \end{array}]$$

No, ovaj se oblik može i pojednostavniti. Prepostavimo da imamo vektor \vec{v}_n koji je okomit i na vektor \vec{v}_A i na vektor \vec{v}_B . Takav vektor zovemo *normala ravnine*, i ilustriran je na slici 2.5. U parametarskoj jednadžbi T_S prebacimo na lijevu stranu:

$$T_R - T_S = \vec{v}_A \cdot \lambda + \vec{v}_B \cdot \mu$$

te razliku točaka na lijevoj strani proglašimo vektorom \vec{v}_R :

$$\vec{v}_R = \vec{v}_A \cdot \lambda + \vec{v}_B \cdot \mu.$$

Pomnožimo sada sve vektorom \vec{v}_n :

$$\vec{v}_R \cdot \vec{v}_n = 0. \quad (2.31)$$

Cijela desna strana jednadžbe je nestala jer su vektori \vec{v}_A i \vec{v}_n kao i vektori \vec{v}_B i \vec{v}_n međusobno okomiti, pa im je skalarni produkt jednak nuli. Ovo nas vodi na zapis ravnine pomoću njezine normale:

$$(T_R - T_S) \cdot \vec{v}_n = 0. \quad (2.32)$$

Vektor \vec{v}_n naziva se vektorom normale (kraće normala) ravnine iz očitih razloga. Ako znamo vektore \vec{v}_A i \vec{v}_B , vektor \vec{v}_n možemo dobiti na različite načine, i taj vektor nije jednoznačan. Naime, postoji beskonačno mnogo vektora koji su okomiti na zadani ravninu; međutim, svi su međusobno kolinearni i razlika između njih je isključivo u njihovoj duljini (normi) te smjeru, gdje su moguća dva smjera. Jedan od načina dobivanja ovakvog vektora je i \times produkt. Možemo odabratи:

$$\vec{v}_n = \vec{v}_A \times \vec{v}_B. \quad (2.33)$$

Ako zapis ravnine pomoću njezine normale – izraz (2.31) – dalje razriješimo raspisivanjem skalarnog produkta, dobivamo:

$$(T_{R_1} - T_{S_1}) \cdot v_{n_1} + (T_{R_2} - T_{S_2}) \cdot v_{n_2} + (T_{R_3} - T_{S_3}) \cdot v_{n_3} = 0.$$

Nakon množenja i grupiranja slijedi:

$$T_{R_1} v_{n_1} + T_{R_2} v_{n_2} + T_{R_3} v_{n_3} - (T_{S_1} v_{n_1} + T_{S_2} v_{n_2} + T_{S_3} v_{n_3}) = 0 \quad (2.34)$$

ili:

$$A \cdot T_{R_1} + B \cdot T_{R_2} + C \cdot T_{R_3} + D = 0. \quad (2.35)$$

Uočimo odmah jedan važan detalj: koeficijenti uz komponente T_{R_1} , T_{R_2} i T_{R_3} direktno odgovaraju komponentama vektora normale ravnine. Ovo je zgodno svojstvo koje se često zaboravi. Dodatno, ako se prethodni izraz podijeli izrazom $\sqrt{A^2 + B^2 + C^2}$ – čime se vektor normale normira (svodi na vektor čija je norma jednaka 1) – apsolutna vrijednost slobodnog koeficijenta tada će odgovarati udaljenost ravnine do ishodišta koordinatnog sustava.

Primjer: 5

Pokažite da tvrdnja iz prethodnog odlomka doista vrijedi: ako je vektor normale ravnine prisutan u implicitnoj jednadžbi ravnine normiran, tada apsolutna vrijednost slobodnog člana predstavlja udaljenost ishodišta do te ravnine.

Rješenje:

Krenut ćemo od implicitnog oblika jednadžbe ravnine, zapisane na uobičajeni način.

$$a \cdot x + b \cdot y + c \cdot z + d = 0$$

Normala ove ravnine je vektor $\vec{n} = [\begin{array}{ccc} a & b & c \end{array}]$. Ako je taj vektor normiran, znači da mu je norma jednaka 1:

$$\|\vec{n}\| = \sqrt{a^2 + b^2 + c^2} = 1 \Rightarrow a^2 + b^2 + c^2 = 1$$

Ova jednakost brzo će nam zatrebatи. Sljedeći korak: kako se definira udaljenost proizvoljne točke od ravnine? Iz te točke spusti se okomica na ravninu. Udaljenost je duljina te okomice. Posljedica je interesantna: iz ishodišta se trebamo pomaknuti upravo u smjeru normale ravnine – samo što ne znamo koliko točno. Zapšimo ono što znamo:

$$T_R = [\begin{array}{ccc} 0 & 0 & 0 \end{array}] + k \cdot [\begin{array}{ccc} a & b & c \end{array}] = [\begin{array}{ccc} ka & kb & kc \end{array}]$$

Pri tome točka T_R leži u ravnini. Iz ishodišta smo se pomaknuli za vektor $k \cdot \vec{n}$, gdje je k u ovom trenutku još nepoznata konstanta koja skalira vektor normale. Također, ako točka T_R leži u ravnini, onda ona zadovoljava implicitni oblik jednadžbe ravnine. Uvrstimo stoga $x = ka$, $y = kb$ i $z = kc$:

$$\begin{aligned} a \cdot (ka) + b \cdot (kb) + c \cdot (kc) + d &= 0 \\ \Rightarrow ka^2 + kb^2 + kc^2 + d &= 0 \\ \Rightarrow k \cdot (a^2 + b^2 + c^2) + d &= 0 \end{aligned}$$

Iskoristimo sada činjenicu da je vektor normale normiran, odnosno da vrijedi: $a^2 + b^2 + c^2 = 1$:

$$k \cdot (a^2 + b^2 + c^2) + d = 0 \Rightarrow k \cdot 1 + d = 0 \Rightarrow k = -d.$$

I ovime smo praktički gotovi s tvrdnjom. Naime, iz ishodišta smo se do ravnine pomaknuli za vektor $k \cdot \vec{n}$, pa je upravo njegova norma tražena udaljenost ishodišta do ravnine. Norma tog vektora je:

$$\begin{aligned} \|k \cdot \vec{n}\| &= \sqrt{(k \cdot a)^2 + (k \cdot b)^2 + (k \cdot c)^2} \\ &= \sqrt{k^2 \cdot (a^2 + b^2 + c^2)} \\ &= \sqrt{(-d)^2 \cdot (1)} \\ &= \sqrt{d^2} \\ &= |d| \end{aligned}$$

Ako koeficijenti normale nisu normirani, dovoljno je podijeliti čitavu implicitnu jednadžbu s normom normale – i slobodni koeficijent automatski će odgovarati udaljenosti ishodišta od ravnine.

Za vježbu se uvjerite da vrijedi i sljedeća tvrdnja: ako su koeficijenti normale ravnine u jednadžbi ravnine normirani, tada se uvrštavanjem proizvoljne točke T u lijevu stranu jednadžbe ravnine, dakle u izraz $a \cdot x + b \cdot y + c \cdot z + d$, direktno dobiva udaljenost točke T od te ravnine (uz eventualnu korekciju predznaka dobivenog broja, jer je udaljenost po definiciji nenegativan broj). Dokaz se radi na sličan način kao što smo to dokazali za udaljenost ishodišta od ravnine – uvjerite se!

Ako u jednadžbu (2.34) ubacimo homogene koordinate, dobiva se:

$$A \cdot \frac{T_{Rh_1}}{h_R} + B \cdot \frac{T_{Rh_2}}{h_R} + C \cdot \frac{T_{Rh_3}}{h_R} + D = 0.$$

ili nakon sređivanja:

$$A \cdot T_{Rh_1} + B \cdot T_{Rh_2} + C \cdot T_{Rh_3} + D \cdot h_R = 0. \quad (2.36)$$

Izraz se dalje može prikazati i matrično:

$$\begin{bmatrix} T_{Rh_1} & T_{Rh_2} & T_{Rh_3} & h_R \end{bmatrix} \cdot \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = 0. \quad (2.37)$$

Ova oblik koristi se vrlo često i korisno ga je poznavati. No, bitan je iz još jednog razloga. Kod jednadžbe pravca na temelju slične relacije definirali smo odnos točke i pravca. Pomoću ove jednadžbe definirat ćemo odnos točke i ravnine. Jednadžbu (2.37) simbolički možemo zapisati:

$$T_{Rh} \cdot \mathbf{R} = 0. \quad (2.38)$$

Dakle, za svaku točku koja je na ravni jednadžba je zadovoljena. No ako uvrstimo za T_R neku točku koja ne pripada ravni, gornja jednadžba neće biti zadovoljena. Na temelju tog rezultata definira se odnos točke i ravnine:

$$T_{Rh} \cdot \mathbf{R} \begin{cases} > 0, & T_{Rh} \text{ je iznad ravnine,} \\ = 0, & T_{Rh} \text{ je na ravni,} \\ < 0, & T_{Rh} \text{ je ispod ravnine.} \end{cases} \quad (2.39)$$

Ovaj odnos moguće je pojasniti ako se prisjetimo da je jedan od načina definiranja ravnine bio upravo putem normale ravnine. Svaka ravnina može imati beskonačno mnogo normala. Pola ih je orijentirano u jednom smjeru, pola u drugom smjeru. Međutim, jednom kada napišemo jednadžbu ravnine (ili očitamo matricu \mathbf{R}), fiksirali smo smjer u kojem je normala okrenuta. Nakon što je to fiksirano, za sve točke koje ne pripadaju ravni a do kojih se dolazi tako da se s ravni pomicemo u smjeru normale kažemo da su iznad ravni; za njih će umnožak $T_{Rh} \cdot \mathbf{R}$ biti pozitivan. S druge pak strane, za sve točke koje ne pripadaju ravni a do kojih se dolazi tako da se s ravni pomicemo u smjeru suprotnom od smjera normale umnožak $T_{Rh} \cdot \mathbf{R}$ će biti negativan. Za takve točke kažemo da su ispod ravni.

Ispitivanje odnosa točke i ravnine bit će nam vrlo važno kod crtanja tijela u 3D prostoru, gdje će tijelo uobičajeno biti zadano dijelovima ravnina koje čine njegovo oplošje. Tamo će nas zanimati koje su nam površine vidljive a koje nisu, i do tih informacija dijelomično ćemo dolaziti i temeljem ispitivanja odnosa točke i ravnine.

2.5.2 Jednadžba ravnine kroz tri točke

Svaka ravnina jednoznačno je određena s tri točke koje ne leže na pravcu. Da bismo odredili jednadžbu ravnine kroz tri točke, krenut ćemo od parametarskog oblika ravnine u homogenom prostoru:

$$T_{Rh} = [\begin{array}{cccc} T_{Rh_1} & T_{Rh_2} & T_{Rh_3} & h_R \end{array}] = [\begin{array}{ccc} \lambda & \mu & 1 \end{array}] \cdot [\begin{array}{cccc} v_{A_1} & v_{A_2} & v_{A_3} & h_A \\ v_{B_1} & v_{B_2} & v_{B_3} & h_B \\ T_{S_1} & T_{S_2} & T_{S_3} & h_S \end{array}]$$

što kraće možemo zapisati kao:

$$T_{Rh} = [\begin{array}{ccc} \lambda & \mu & 1 \end{array}] \cdot \mathbf{G}$$

U ovom slučaju ravnina je određena s dva vektora i točkom. Pri tome su vektori i točke četverokomponentni jer radimo u tri dimenzije (3 komponente) i u homogenom prostoru (još jedna komponenta).

Kao i kod određivanja jednadžbe pravca, i ovdje možemo raditi na isti način. Jednadžba će kroz točku T_A proći za neki λ i neki μ , kroz točku T_B će proći za neki drugi λ i neki drugi μ , i kroz točku T_C će proći za neki treći λ i neki treći μ . Idemo stoga odabratи za koje će se to λ i μ dogoditi. Neka ravnina prolazi kroz T_A za $\lambda = 0$ i $\mu = 0$, kroz T_B za $\lambda = 1$ i $\mu = 0$ te kroz T_C za $\lambda = 1$ i $\mu = 1$. Tada vrijedi:

$$T_{Ah} = [\begin{array}{cccc} T_{Ah_1} & T_{Ah_2} & T_{Ah_3} & h_A \end{array}] = [\begin{array}{ccc} 0 & 0 & 1 \end{array}] \cdot \mathbf{G}$$

$$T_{Bh} = [\begin{array}{cccc} T_{Bh_1} & T_{Bh_2} & T_{Bh_3} & h_B \end{array}] = [\begin{array}{ccc} 1 & 0 & 1 \end{array}] \cdot \mathbf{G}$$

$$T_{Ch} = [\begin{array}{cccc} T_{Ch_1} & T_{Ch_2} & T_{Ch_3} & h_C \end{array}] = [\begin{array}{ccc} 1 & 1 & 1 \end{array}] \cdot \mathbf{G}$$

što nakon stapanja u jednu matričnu jednadžbu daje:

$$[\begin{array}{c} T_{Ah} \\ T_{Bh} \\ T_{Ch} \end{array}] = [\begin{array}{ccc} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{array}] \cdot \mathbf{G}.$$

Sređivanjem ovog izraza dobivamo:

$$\mathbf{G} = [\begin{array}{ccc} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{array}]^{-1} \cdot [\begin{array}{c} T_{Ah} \\ T_{Bh} \\ T_{Ch} \end{array}] = [\begin{array}{ccc} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 1 & 0 & 0 \end{array}] \cdot [\begin{array}{c} T_{Ah} \\ T_{Bh} \\ T_{Ch} \end{array}] = [\begin{array}{c} T_{Bh} - T_{Ah} \\ T_{Ch} - T_{Bh} \\ T_{Ah} \end{array}]. \quad (2.40)$$

2.6 Dodatni često korišteni pojmovi

2.6.1 Skalarni i vektorski produkt

Skalarni i vektorski produkt dva su temeljna operatora linearne algebre, i kao takvi čine nezaobilazan alat računalne grafike. Stoga ćemo se ovdje samo ukratko podsjetiti kako su definirani, i kako ih najčešće koristimo.

Skalarni produkt

Skalarni produkt dvaju n -dimenzijskih vektora $\vec{A} = (a_1, a_2, \dots, a_n)$ i $\vec{B} = (b_1, b_2, \dots, b_n)$ definiran je kao suma umnožaka i -tih parova komponenti obaju vektora. Dakle:

$$\vec{A} \cdot \vec{B} = (a_1, a_2, \dots, a_n) \cdot (b_1, b_2, \dots, b_n) = a_1 \cdot b_1 + a_2 \cdot b_2 + \dots + a_n \cdot b_n$$

Od zanimljivijih svojstava skalarnog produkta spomenimo sljedeća dva svojstva.

- Ako je skalarni produkt dvaju vektora čija je norma različita od nule jednak nula, vektori su međusobno okomiti. Možemo pisati:

$$\vec{A} \cdot \vec{B} = 0 \rightarrow \vec{A} \perp \vec{B}.$$

- Skalarni produkt dvaju vektora \vec{A} i \vec{B} mjeri je kosinusa kuta koji oni zatvaraju. Točnije, vrijedi sljedeće:

$$\cos(\angle(\vec{A}, \vec{B})) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \cdot \|\vec{B}\|}.$$

Primjer: 6

Pronadite vektor koji je okomit na vektor $\vec{m} = (1, 2, 4)$.

Rješenje:

Poslužit ćemo se skalarnim produktom. Uočimo da je zadani vektor trodimenzijski, pa će i traženi vektor \vec{n} biti trodimenzijski. Možemo pisati:

$$(n_x, n_y, n_z) \cdot (1, 2, 4) = 0 \quad \Rightarrow \quad n_x \cdot 1 + n_y \cdot 2 + n_z \cdot 4 = 0$$

Osim ove jednadžbe – jednadžbe od tri nepoznanice – nemamo niti jedan drugi izraz kojim bismo mogli više ograničiti traženi vektor. Evo i zašto. Zamislimo da smo u koordinatnom sustavu vektor \vec{m} fiksirali u ishodište. Taj vektor tada je okomit na ravninu koja je također fiksirana u ishodištu. Svaki vektor koji se nalazi u toj ravnini bit će jedno moguće rješenje. Postavimo li u tu ravninu novi dvodimenzijski koordinatni sustav, svaki vektor u toj ravnini bit će određen s dva parametra: svojom koordinatom vezanom uz prvu os, te svojom koordinatom vezanom uz drugu os. Tek kada ta dva parametra fiksiramo, gornja će nam jednadžba dati preostalu treću vrijednost. Zaključak ovog misaonog eksperimenta jest uočiti da imamo dva stupnja slobode, pa možemo sami dodati dvije jednadžbe koje će rezultirati jednoznačnim rješenjem.

Primjerice, gledajući gornji izraz, možemo tražiti da je $n_x = 1$. Tada ostajemo na jednadžbi:

$$1 \cdot 1 + n_y \cdot 2 + n_z \cdot 4 = 0 \quad \Rightarrow \quad n_y \cdot 2 + n_z \cdot 4 = -1.$$

Dalje, možemo poželjeti da je $n_y = 2$, čime se jednadžba dalje reducira na:

$$2 \cdot 2 + n_z \cdot 4 = -1 \quad \Rightarrow \quad n_z \cdot 4 = -5.$$

Pa slijedi:

$$n_z = \frac{-5}{4}.$$

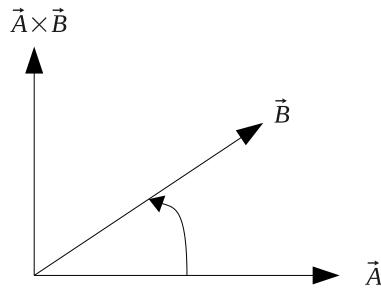
Provjerimo da su vektori $(1, 2, 4)$ i pronadjeni vektor $(1, 2, \frac{-5}{4})$ doista okomiti:

$$(1, 2, \frac{-5}{4}) \cdot (1, 2, 4) = 1 \cdot 1 + 2 \cdot 2 + \frac{-5}{4} \cdot 4 = 1 + 2 + (-5) = 0$$

Ako je potrebno pronaći bilo koji okomit vektor, postupak prikazan u prethodnom primjeru je sasvim prihvatljiv. Međutim, u praksi se češće umjesto ovakvih "proizvoljnih" ograničenja kao jedno od ograničenja nameće da je norma tog vektora jednaka 1 (dakle, da je pronađeni vektor normiran). I uz takvo ograničenje, ostaje nam još jedan stupanj slobode koji je potrebno fiksirati. U praksi, okomiti nam vektori najčešće trebaju kao normale ravnine. Kako je ravnina u parametarskom obliku zadana preko dva vektora koji leže u njoj, naš je zadatak pronaći vektor koji je okomit na oba – čime smo ograničili početni jednadžbu na samo jedan stupanj slobode, pa je dovoljno zadati još jedno ograničenje kojim ćemo sam vektor fiksirati.

Treba međutim paziti kako zadajemo to treće ograničenje. Primjerice, ako tražimo da vrijedi:

$$n_x + n_y + n_z = 1$$



Slika 2.6: Vektorski produkt

dobit ćemo jednoznačno definiran vektor. Međutim, ako tražimo da norma tog vektora bude 1, dakle:

$$\sqrt{n_x \cdot n_x + n_y \cdot n_y + n_z \cdot n_z} = 1 \quad \Rightarrow \quad n_x \cdot n_x + n_y \cdot n_y + n_z \cdot n_z = 1$$

bitno smo zakomplificirali izračun (jer imamo kvadratnu jednadžbu), i dodatno, opet nam je ostao jedan stupanj slobode – rekli smo da želimo vektor norme 1, ali takva postoje dva: onaj koji gleda na gornju stranu ravnine, te onaj koji gleda na donju stranu ravnine, pa i to treba dodatno specificirati. Stoga se u slučaju da na dva zadana vektora želimo na što jednostavniji način pronaći treći okomit (i ne zanimaju nas daljnja "svojstva" tog vektora), često koristi i vektorski produkt opisan u nastavku.

Vektorski produkt

Vektorski produkt tipično razmatramo u trodimenzijskom prostoru. Vektorski produkt dvaju 3-dimenzijskih vektora $\vec{A} = (a_1, a_2, a_3)$ i $\vec{B} = (b_1, b_2, b_3)$ definiran je kao:

$$\vec{A} \times \vec{B} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix},$$

što dalje možemo razriješiti kao:

$$\vec{A} \times \vec{B} = \vec{i} \cdot (a_2 \cdot b_3 - a_3 \cdot b_2) - \vec{j} \cdot (a_1 \cdot b_3 - a_3 \cdot b_1) + \vec{k} \cdot (a_1 \cdot b_2 - a_2 \cdot b_1).$$

Od zanimljivijih svojstava vektorskog produkta spomenimo sljedeća četiri svojstva.

- Vektorski produkt \vec{n} dvaju vektora \vec{A} i \vec{B} okomit je i na \vec{A} i na \vec{B} (slika 2.6).

$$(\vec{A} \times \vec{B}) \perp \vec{A} \quad \wedge \quad (\vec{A} \times \vec{B}) \perp \vec{B}.$$

U ovo se možete jednostavno uvjeriti tako da pogledate skalarne produkte $\vec{n} \cdot \vec{A}$ i $\vec{n} \cdot \vec{B}$.

- Smjer vektora \vec{n} koji predstavlja vektorski produkt dvaju vektora \vec{A} i \vec{B} određen je pravilom desne ruke. Ako prsti desne ruke pokazuju od vektora \vec{A} prema vektoru \vec{B} , palac pokazuje smjer vektora \vec{n} .
- Norma vektorskog produkta $\|\vec{n}\|$ dvaju vektora \vec{A} i \vec{B} jednaka je:

$$\|\vec{A} \times \vec{B}\| = \|\vec{A}\| \cdot \|\vec{B}\| \cdot \sin(\angle(\vec{A}, \vec{B}))$$

pri čemu se kao kut gleda onaj manji. Geometrijska interpretacija ove činjenice jest da je norma vektorskog produkta vektora \vec{A} i \vec{B} jednaka površini paralelograma što ga razapinju vektori \vec{A} i \vec{B} , odnosno jednaka je dvostrukoj vrijednosti površine trokuta koji razapinju ti vektori. Ovu činjenicu kasnije ćemo koristiti na više mesta, a jedan od primjera će biti i izračun baricentričnih koordinata.

- Posljedica prethodnog svojstva jest da je vektorski produkt dvaju kolinearnih vektora jednak nul-vektor. Dakle, ako su vektori paralelni, vektorski produkt ima normu nula.

2.6.2 Baricentrične koordinate

Baricentrične koordinate često su korištene u računalnoj grafici pri radu s trokutima. Trokut je dio ravnine omeđen trima točkama koje zovemo vrhovima trokuta. Označimo ih A , B i C . Prisjetimo se: tri točke koje ne leže na istom pravcu u prostoru određuju ravninu, i u toj ravnini leži trokut ABC . Također, već znamo da svaku točku ravnine možemo zapisati kao linearnu kombinaciju dvaju nekolinearnih vektora koji leže u toj ravnini, i koja "kreće" iz neke proizvoljne fiksne točke u ravnini. Napravimo sada jedan specifični odabir: neka fiksna točka bude sam vrh A , neka prvi vektor bude onaj razapet između vrhova B i A : $\vec{v}_1 = \vec{B} - \vec{A}$, te neka drugi vektor bude onaj razapet između vrhova C i A : $\vec{v}_2 = \vec{C} - \vec{A}$. Tada se proizvoljna točka T koja leži u toj ravnini može zapisati kao:

$$\vec{T} = \vec{A} + \lambda \cdot (\vec{B} - \vec{A}) + \mu \cdot (\vec{C} - \vec{A}).$$

Krenuvši od ovog izraza, možemo to dalje raspisati i svesti na drugačiji oblik:

$$\begin{aligned}\vec{T} &= \vec{A} + \lambda \cdot \vec{B} - \lambda \cdot \vec{A} + \mu \cdot \vec{C} - \mu \cdot \vec{A} \\ &= (1 - \lambda - \mu) \vec{A} + \lambda \cdot \vec{B} + \mu \cdot \vec{C}\end{aligned}$$

čime dolazimo do konačnog zapisa:

$$\vec{T} = (1 - \lambda - \mu) \vec{A} + \lambda \cdot \vec{B} + \mu \cdot \vec{C}. \quad (2.41)$$

Ako parametre uz svaku točku zamijenimo novom oznakom (koristit ćemo oznake t_i), dobivamo izraz:

$$\vec{T} = t_1 \vec{A} + t_2 \cdot \vec{B} + t_3 \cdot \vec{C}. \quad (2.42)$$

gdje su (t_1, t_2, t_3) baricentrične koordinate točke T . Uočimo dakle: baricentrične koordinate su drugačiji način za prikaz točaka koje leže u istoj ravnini. Kada koristimo baricentrične koordinate, moramo znati s obzirom na koje vrhove su te koordinate definirane. Baricentrične koordinate postoje za svaku točku ravnine koju definiraju tri zadana vrha trokuta. Međutim, uporabom baricentričnih koordinata na jednostavan ćemo način moći utvrditi u kakvom je odnosu bilo koja promatrana točka ravnine i zadani trokut. Prije no što pokažemo, uočimo i da vrijedi sljedeća jednakost:

$$t_1 + t_2 + t_3 = 1. \quad (2.43)$$

Naime, ako iskoristimo (2.41), i uočimo da vrijedi: $t_1 = 1 - \lambda - \mu$, $t_2 = \lambda$ i $t_3 = \mu$, dokaz je trivijalan.

$$t_1 + t_2 + t_3 = (1 - \lambda - \mu) + \lambda + \mu = 1.$$

2.6.3 Izračun baricentričnih koordinata

Baricentrične koordinate možemo izračunati na nekoliko načina. Prvi način jest direktno uporabom izraza (2.42). Naime, taj izraz dovoljan je da napišemo sustav od tri jednadžbe s tri nepoznanice (u slučaju 3D točaka), kao što je ilustrirano u primjeru u nastavku.

Primjer: 7

Trokut je zadan vrhovima $A = (1, 1, 0)$, $B = (6, 11, 2)$ i $C = (11, 1, 0)$. Izračunati baricentrične koordinate točke $T = (6, 6, 1)$ primjenom izraza (2.42).

Rješenje:

Prema (2.42) vrijedi:

$$\vec{T} = t_1 \vec{A} + t_2 \cdot \vec{B} + t_3 \cdot \vec{C}$$

Vektore ćemo prikazati kao trostupčaste matrice, pa možemo pisati:

$$\begin{bmatrix} 6 \\ 6 \\ 1 \end{bmatrix} = t_1 \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} + t_2 \cdot \begin{bmatrix} 6 \\ 11 \\ 2 \end{bmatrix} + t_3 \cdot \begin{bmatrix} 11 \\ 1 \\ 0 \end{bmatrix}$$

Ovo je sustav od tri jednadžbe s tri nepoznanice:

$$\begin{aligned} 6 &= 1 \cdot t_1 + 6 \cdot t_2 + 11 \cdot t_3 \\ 6 &= 1 \cdot t_1 + 11 \cdot t_2 + 1 \cdot t_3 \\ 1 &= 0 \cdot t_1 + 2 \cdot t_2 + 0 \cdot t_3 \end{aligned}$$

Rješavanjem dobivamo $t_1 = \frac{1}{4}$, $t_2 = \frac{1}{2}$ i $t_3 = \frac{1}{4}$, pa su baricentrične koordinate točke T upravo $(\frac{1}{4}, \frac{1}{2}, \frac{1}{4})$.

Drugi način jest uočiti da sve tri baricentrične koordinate nisu potpuno slobodne (vidi izraz (2.43)), i posegnuti za matričnim računom. Naime, koristeći izraze (2.42) i (2.43) za proizvoljnu točku ravnine T možemo pisati:

$$\begin{aligned} \vec{T} &= t_1 \cdot \vec{A} + t_2 \cdot \vec{B} + t_3 \cdot \vec{C} \\ &= (1 - t_2 - t_3) \cdot \vec{A} + t_2 \cdot \vec{B} + t_3 \cdot \vec{C} \\ &= \vec{A} + t_2 \cdot (\vec{B} - \vec{A}) + t_3 \cdot (\vec{C} - \vec{A}) \end{aligned}$$

Sređivanjem ovog izraza dobivamo:

$$t_2 \cdot (\vec{B} - \vec{A}) + t_3 \cdot (\vec{C} - \vec{A}) = \vec{T} - \vec{A}$$

U ovom izrazu imamo dvije nepoznanice: t_2 i t_3 , i tri jednadžbe. Zanemarimo stoga na tren treću jednadžbu, i promatrajmo samo prve dvije. Možemo pisati:

$$\begin{aligned} t_2 \cdot (B_x - A_x) + t_3 \cdot (C_x - A_x) &= T_x - A_x \\ t_2 \cdot (B_y - A_y) + t_3 \cdot (C_y - A_y) &= T_y - A_y \end{aligned}$$

Ovo možemo zapisati matrično:

$$\begin{bmatrix} B_x - A_x & C_x - A_x \\ B_y - A_y & C_y - A_y \end{bmatrix} \cdot \begin{bmatrix} t_2 \\ t_3 \end{bmatrix} = \begin{bmatrix} T_x - A_x \\ T_y - A_y \end{bmatrix}$$

Označimo li ovu lijevu matricu kao \mathbf{M} , možemo pisati:

$$\mathbf{M} \cdot \begin{bmatrix} t_2 \\ t_3 \end{bmatrix} = \begin{bmatrix} T_x - A_x \\ T_y - A_y \end{bmatrix}$$

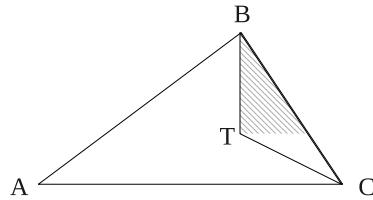
Množenjem tog izraza s lijeva inverznom matricom od \mathbf{M} direktno dobivamo tražene baricentrične koordinate:

$$\begin{bmatrix} t_2 \\ t_3 \end{bmatrix} = \mathbf{M}^{-1} \cdot \begin{bmatrix} T_x - A_x \\ T_y - A_y \end{bmatrix} \quad (2.44)$$

Preostalu baricentričnu koordinatu dobit ćemo preko izraza (2.43). Uočimo da se u ovom slučaju radi o matrici \mathbf{M} dimenzija 2×2 , čiji je inverz trivijalno pronaći. Također, kako su stupci te matrice upravo vektori između vrhova trokuta, matrica je sigurno invertibilna, budući da su ti vektori nekolinearni. Najjednostavniji način brzog pronalaska matričnog inverza matrice dimenzija 2×2 jest uporabom matrice kofaktora \mathbf{C} , te jednakosti:

$$\mathbf{A}^{-1} = \frac{1}{\det \mathbf{A}} \cdot \mathbf{C}^T \quad (2.45)$$

gdje je \mathbf{A} matrica koju želimo invertirati, \mathbf{C} njezina matrica kofaktora a \mathbf{C}^T transponirana matrica kofaktora.



Slika 2.7: Baricentrične koordinate preko omjera površina trokuta

Primjer: 8

Trokut je zadan vrhovima $A = (1, 1, 0)$, $B = (6, 11, 2)$ i $C = (11, 1, 0)$. Izračunati baricentrične koordinate točke $T = (6, 6, 1)$ primjenom izraza (2.44).

Rješenje:

Matrica \mathbf{M} definirana je kao dvoretčana matrica čiji su stupci $\vec{B} - \vec{A}$ i $\vec{C} - \vec{A}$. Ona dakle glasi:

$$\mathbf{M} = \begin{bmatrix} 6-1 & 11-1 \\ 11-1 & 1-1 \end{bmatrix} = \begin{bmatrix} 5 & 10 \\ 10 & 0 \end{bmatrix}$$

Idemo izračunati njezin inverz, koristeći izraz (2.45). Determinantu lagano izračunamo:

$$\det \mathbf{M} = 5 \cdot 0 - 10 \cdot 10 = -100$$

Matrica kofaktora matrice \mathbf{M} je matrica (označimo je s \mathbf{C}) jednakih dimenzija kao i \mathbf{M} . Pri tome se element na poziciji $c_{i,j}$ dobije tako da se u originalnoj matrici \mathbf{M} izbriše i -ti redak i j -ti stupac, pa se od tako dobivene matrice koja je za jednu dimenziju manja izračuna determinanta. Ako je $i+j$ parno, $c_{i,j}$ jednak je tako izračunatoj determinanti. Ako je $i+j$ neparno, $c_{i,j}$ jednak je minus vrijednosti izračunate determinante. Zvući komplikirano, ali za matricu dimenzija 2×2 postupak je izuzetno jednostavan – brisanjem jednog retka i stupca ostaje matrica dimenzija 1×1 , čija je determinanta upravo jednaka jedinom elementu te matrice.

Izračunajmo elemente matrice kofaktora. Pogledajmo naprije element $c_{1,1}$. Brisanjem prvog retka i prvog stupca matrice \mathbf{M} ostaje nam matrica [0] čija je determinanta 0, pa je $c_{1,1} = 0$. Pogledajmo element $c_{2,1}$. Brisanjem drugog retka i prvog stupca matrice \mathbf{M} ostaje nam matrica [10] čija je determinanta 10. Kako je $2+1=3$ neparno, uzimamo minus vrijednost determinante, pa je $c_{2,1} = -10$. Na taj način dobivamo matricu kofaktora:

$$\mathbf{C} = \begin{bmatrix} 0 & -10 \\ -10 & 5 \end{bmatrix}$$

Transponirana matrica kofaktora tada je:

$$\mathbf{C}^T = \begin{bmatrix} 0 & -10 \\ -10 & 5 \end{bmatrix}$$

Prema (2.45) slijedi da je:

$$\mathbf{M}^{-1} = \frac{1}{\det \mathbf{M}} \cdot \mathbf{C}^T = \frac{1}{-100} \cdot \begin{bmatrix} 0 & -10 \\ -10 & 5 \end{bmatrix}$$

Uvrštavanjem u izraz (2.44) slijedi:

$$\begin{bmatrix} t_2 \\ t_3 \end{bmatrix} = \frac{1}{\det \mathbf{M}} \cdot \mathbf{C}^T = \frac{1}{-100} \cdot \begin{bmatrix} 0 & -10 \\ -10 & 5 \end{bmatrix} \cdot \begin{bmatrix} 6-1 \\ 6-1 \end{bmatrix} = \frac{1}{-100} \cdot \begin{bmatrix} 0 & -10 \\ -10 & 5 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 5 \end{bmatrix}$$

Od tuda direktno čitamo:

$$t_2 = \frac{1}{-100} \cdot (0 \cdot 5 + -10 \cdot 5) = \frac{1}{-100} \cdot (-50) = \frac{1}{2}$$

$$t_3 = \frac{1}{-100} \cdot (-10 \cdot 5 + 5 \cdot 5) = \frac{1}{-100} \cdot (-25) = \frac{1}{4}$$

Preostala baricentrična kordinata tada je:

$$t_1 = 1 - t_2 - t_3 = 1 - \frac{1}{2} - \frac{1}{4} = \frac{1}{4}$$

Tražene baricentrične koordinate su: $(\frac{1}{4}, \frac{1}{2}, \frac{1}{4})$.

Spomenimo još jedan način kako možemo izračunati baricentrične koordinate. Prepostavimo da nas zanimaju baricentrične koordinate točke T koja se nalazi u trokutu ABC . Da bismo izračunali vrijednost baricentrične koordinate t_1 (koja je uz vrh A), promotrimo trokut koji točka T definira s preostalim vrhovima trokuta (dakle, B i C). Slika 2.7 ilustrira ovu situaciju. Omjer površine trokuta TBC i ABC odgovara baricentričnoj koordinati t_1 . Na jednak način definiraju se i ostale baricentrične koordinate.

Možda nije odmah očito kako jednostavno doći do površine promatranih trokuta. Međutim, prijetimo se samo svojstava vektorskog produkta. Norma vektorskog produkta odgovara površini paralelograma što ga razapinju promatrani vektori, što je točno jednako dvostrukoj površini trokuta što ga razapinju ti vektori. Dakle, površinu trokuta ABC dobit ćemo kao:

$$P(ABC) = \frac{\|(B - A) \times (C - A)\|}{2}$$

Površina trokuta TBC po istom principu je:

$$P(TBC) = \frac{\|(B - T) \times (C - T)\|}{2}$$

Njihov omjer odgovara baricentričnoj koordinati t_1 :

$$t_1 = \frac{P(TBC)}{P(ABC)} = \frac{\frac{\|(B - T) \times (C - T)\|}{2}}{\frac{\|(B - A) \times (C - A)\|}{2}} = \frac{\|(B - T) \times (C - T)\|}{\|(B - A) \times (C - A)\|} \quad (2.46)$$

Na sličan način dobiju se i izrazi za preostale baricentrične koordinate:

$$t_2 = \frac{P(TAC)}{P(ABC)} = \frac{\|(A - T) \times (C - T)\|}{\|(B - A) \times (C - A)\|} \quad (2.47)$$

$$t_3 = \frac{P(TAB)}{P(ABC)} = \frac{\|(A - T) \times (B - T)\|}{\|(B - A) \times (C - A)\|} \quad (2.48)$$

Primjer: 9

Trokut je zadan vrhovima $A = (1, 1, 0)$, $B = (6, 11, 2)$ i $C = (11, 1, 0)$. Izračunati baricentrične koordinate točke $T = (6, 6, 1)$ primjenom omjera površina.

Rješenje:

Trebat će nam vektori $\vec{A} - \vec{T}$, $\vec{B} - \vec{T}$ i $\vec{C} - \vec{T}$, kao i $\vec{B} - \vec{A}$ i $\vec{C} - \vec{A}$. Izračunajmo ih.

$$\vec{A} - \vec{T} = (1, 1, 0) - (6, 6, 1) = (-5, -5, -1)$$

$$\vec{B} - \vec{T} = (6, 11, 2) - (6, 6, 1) = (0, 5, 1)$$

$$\vec{C} - \vec{T} = (11, 1, 0) - (6, 6, 1) = (5, -5, -1)$$

$$\vec{B} - \vec{A} = (6, 11, 2) - (1, 1, 0) = (5, 10, 2)$$

$$\vec{C} - \vec{A} = (11, 1, 0) - (1, 1, 0) = (10, 0, 0)$$

Prema (2.42) vrijedi:

Vektorski produkt $(\vec{B} - \vec{A}) \times (\vec{C} - \vec{A})$ jednak je $(0, -20, 100)$ pa je njegova norma $\sqrt{400 + 10000} = \sqrt{10400}$. Vektorski produkt $(\vec{B} - \vec{T}) \times (\vec{C} - \vec{T})$ jednak je $(0, 5, 25)$ pa je njegova norma $\sqrt{25 + 625} = \sqrt{650}$.

$$t_1 = \frac{\sqrt{650}}{\sqrt{10400}} = \sqrt{\frac{650}{10400}} = \sqrt{0.0625} = 0.25 = \frac{1}{4}$$

Vektorski produkt $(\vec{A} - \vec{T}) \times (\vec{C} - \vec{T})$ jednak je $(0, -10, 50)$ pa je njegova norma $\sqrt{100 + 2500} = \sqrt{2600}$.

$$t_2 = \frac{\sqrt{2600}}{\sqrt{10400}} = \sqrt{\frac{2600}{10400}} = \sqrt{0.25} = 0.5 = \frac{1}{2}$$

Vektorski produkt $(\vec{A} - \vec{T}) \times (\vec{B} - \vec{T})$ jednak je $(0, 5, 25)$ pa je njegova norma $\sqrt{25 + 625} = \sqrt{650}$.

$$t_3 = \frac{\sqrt{650}}{\sqrt{10400}} = \sqrt{\frac{650}{10400}} = \sqrt{0.0625} = 0.25 = \frac{1}{4}$$

Baricentrične koordinate točke T su dakle $(\frac{1}{4}, \frac{1}{2}, \frac{1}{4})$.

2.6.4 Odnos trokuta i točke preko baricentričnih koordinata

Baricentrične koordinate mogu nam poslužiti kako bismo utvrdili nalazi li se proizvoljna točka T ravnine unutar trokuta, na rubu trokuta ili pak izvan trokuta. Vrijedi sljedeće. Neka točka T ima baricentrične koordinate (t_1, t_2, t_3) .

$$\begin{cases} \text{ako } \forall i.t_i \in (0, 1) & \Rightarrow T \text{ je unutar trokuta,} \\ \text{ako } \forall i.t_i \in [0, 1] \wedge \exists j.t_j = 1 & \Rightarrow T \text{ je na rubu trokuta,} \\ \text{ako } \exists i.t_i \notin [0, 1] & \Rightarrow T \text{ je izvan trokuta.} \end{cases} \quad (2.49)$$

Drugim riječima, ako su sve baricentrične koordinate strogo između 0 i 1 (0 i 1 su isključeni iz intervala), točka se nalazi unutar trokuta. Ako su sve baricentrične koordinate između 0 i uključivo 1 te ako je barem jedna baricentrična koordinata jednaka 1, tada je točka na rubu trokuta. Ako ništa od ovoga nije zadovoljeno, točka je izvan trokuta.

Primjer: 10

Trokat je zadan vrhovima $A = (1, 1, 0)$, $B = (6, 11, 2)$ i $C = (11, 1, 0)$. Odrediti položaj točaka $T_1 = (6, 6, 1)$ i $T_2 = (8.5, 8.5, 1.5)$ uporabom baricentričnih koordinata.

Rješenje:

Baricentrične koordinate točke T_1 već smo odredili: $(\frac{1}{4}, \frac{1}{2}, \frac{1}{4})$. Kako su sve baricentrične koordinate unutar intervala $(0, 1)$, točka se nalazi unutar trokuta.

Baricentrične koordinate točke T_2 su: $(-\frac{1}{8}, \frac{3}{8}, \frac{3}{4})$ (izračunajte ih za vježbu). Kako je koordinata t_1 manja od 0, točka se nalazi izvan trokuta.

2.7 Česti zadatci

U nastavku ove knjige često ćemo se susretati s nekim tipičnim problemima poput pronalaska sjecišta pravca i ravnine, pravca i sfere, razmatranjem prolazi li pravac kroz trokut u 3D prostoru i sl. Samo neki od slučajeva gdje će nam trebati odgovori na ova pitanja su algoritmi bacanja zrake (engl. *Ray Casting*) i praćenja zrake (engl. *Ray Tracing*). Stoga ćemo se ovdje pozabaviti rješavanjem upravo tih problema.

Prepostavit ćemo da je pravac zadan točkama T_S i T_E . Pri tome će se promatrač nalaziti u točki T_S i gledat će prema točki T_E . Takav pravac u 3D prostoru zapisivat ćemo u vektorskom parametarskom obliku:

$$\vec{t}(\lambda) = \vec{T}_S + \lambda \cdot \vec{d}$$

pri čemu je $\vec{t}(\lambda)$ točka na pravcu a \vec{d} predstavlja normirani (jedinični) vektor pravca:

$$\vec{d} = \frac{\vec{T}_E - \vec{T}_S}{\|\vec{T}_E - \vec{T}_S\|}.$$

Držimo li se analogije promatrača koji se nalazi u T_S i gleda u smjeru T_E , tada su sve točke $\vec{t}(\lambda)$ takve da je $\lambda > 0$ ispred promatrača, a sve točke $\vec{t}(\lambda)$ takve da je $\lambda < 0$ iza promatrača (pa ih promatrač ne vidi).

2.7.1 Probodište pravca i ravnine

Neka je ravnina zadana u implicitnom obliku: $A \cdot x + B \cdot y + C \cdot z + D = 0$. Prisjetimo li se da koeficijenti A , B i C zapravo predstavljaju koeficijente normale ravnine, jednadžbu možemo zapisati i ovako: $\vec{n} \cdot \vec{t} + D = 0$, gdje je \vec{n} normala ravnine a \vec{t} točka koja leži u ravnini, i čije su komponente (x, y, z) . Uvrštavanjem parametarskog oblika jednadžbe pravca umjesto \vec{t} slijedi:

$$\vec{n} \cdot \vec{t} + D = 0$$

$$\vec{n} \cdot (\vec{T}_S + \lambda \cdot \vec{d}) + D = 0$$

$$\vec{n} \cdot \vec{T}_S + \lambda \cdot \vec{n} \cdot \vec{d} + D = 0$$

$$\lambda = \frac{-(\vec{n} \cdot \vec{T}_S + D)}{\vec{n} \cdot \vec{d}}$$

Ako je $\vec{n} \cdot \vec{d} = 0$, normala ravnine i pravac su okomiti, što znači da je pravac paralelan s ravninom. Postoje dvije mogućnosti - ili pravac leži u ravnini, pa ima beskonačno sjecišta (svaka točka je jedno sjecište), ili pravac ne leži u ravnini a kako je paralelan, nema niti jedno sjecište. U praksi ćemo slučaj kada je $\vec{n} \cdot \vec{d} = 0$ najčešće zanemariti (odnosno, tada ćemo reći da se pravac i ravnina nemaju probodište).

Slučaj kada je $\vec{n} \cdot \vec{d} \neq 0$ je slučaj koji će nas zanimati. Ako je dodatno $\lambda > 0$, postoji probodište ispred promatrača, a dobit ćemo ga tako da izračunati λ uvrstimo u parametarski oblik jednadžbe pravca. Ako je dodatno $\lambda < 0$, postoji probodište no kako je ono iza promatrača, slučaj nas neće zanimati.

2.7.2 Probodište pravca i sfere

Sfera radijusa r i centra \vec{C} definirana je jednadžbom $(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$. Poslužimo se opet istim oznakama kao u prethodnoj sekciji; neka je \vec{t} točka koja leži na sferi, $\vec{t} = (x, y, z)$. Prethodnu jednadžbu tada možemo pisati i ovako: $(\vec{t} - \vec{C}) \cdot (\vec{t} - \vec{C}) = r^2$. Uvrstimo sada u jednadžbu parametarski oblik jednadžbe pravca:

$$(\vec{t} - \vec{C}) \cdot (\vec{t} - \vec{C}) = r^2$$

$$(\vec{T}_S + \lambda \cdot \vec{d} - \vec{C}) \cdot (\vec{T}_S + \lambda \cdot \vec{d} - \vec{C}) = r^2$$

$$(\lambda \cdot \vec{d} + \vec{T}_S - \vec{C}) \cdot (\lambda \cdot \vec{d} + \vec{T}_S - \vec{C}) = r^2$$

$$\lambda^2 \vec{d} \cdot \vec{d} + 2\lambda \vec{d} \cdot (\vec{T}_S - \vec{C}) + (\vec{T}_S - \vec{C}) \cdot (\vec{T}_S - \vec{C}) - r^2 = 0.$$

Uočimo da smo dobili kvadratnu jednadžbu oblika $a\lambda^2 + b\lambda + c = 0$, uz:

$$a = \vec{d} \cdot \vec{d} = 1$$

$$b = 2\vec{d} \cdot (\vec{T}_S - \vec{C})$$

$$c = (\vec{T}_S - \vec{C}) \cdot (\vec{T}_S - \vec{C}) - r^2.$$

Koefficijent a je jedan jer smo prethodno tražili da vektor \vec{d} bude normiran; skalarni produkt tog vektora sa samim sobom jednak kvadratu norme, no to u tom slučaju upravo 1. Rješenja jednadžbe su:

$$\lambda_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \Rightarrow \lambda_{1,2} = \frac{-b \pm \sqrt{b^2 - 4c}}{2}.$$

Sada imamo sljedeće mogućnosti.

- λ_1 i λ_2 su dva različita realna broja i oba su veća od 0. Neka je $\lambda_1 < \lambda_2$ (ako ovo ne vrijedi, zamijenite vrijednosti). Postoje dva sjecišta sa sferom i oba su ispred promatrača. λ_1 određuje bliže sjecište a λ_2 dalje sjecište (ako je sfera neprozirna, to se sjecište onda ne vidi). Konkretnе točke dobivamo uvrštavanjem λ_1 i λ_2 u parametarski oblik jednadžbe pravca.
- λ_1 i λ_2 su dva različita realna broja, jedan je pozitivan, drugi nije. Neka je $\lambda_1 \leq 0$ i $\lambda_2 > 0$ (ako ovo ne vrijedi, zamijenite vrijednosti). Postoje dva sjecišta sa sferom. Jedno je iza promatrača (određeno s λ_1) a drugo je ispred promatrača (određeno s λ_2). Promatrač se, dakle, nalazi u sferi.
- $\lambda_1 = \lambda_2 = \lambda$ i to je realan broj. Pravac je tangenta na sferu i postoji samo jedna točka dodira. Ako je $\lambda > 0$, točka je ispred promatrača, ako je $\lambda < 0$ točka je iza promatrača.
- λ_1 i λ_2 su dva različita kompleksna broja. U tom slučaju pravac i sfera nemaju sjecišta.

2.7.3 Probodište pravca i trokuta

Neka je trokut zadan u prostoru s tri točke A , B i C . Zanima nas probada li pravac taj trokut. Problem ćemo rastaviti na dva jednostavnija problema.

Prvi problem je pronalazak probodišta ravnine u kojoj leži trokut i pravca. Prisjetimo se, ako s \vec{t} označimo točku ravnine, a s \vec{n} normalu ravnine, implicitni oblik jednadžbe ravnine možemo pisati kao $\vec{n} \cdot \vec{t} + D = 0$. Normalu ćemo odrediti kao vektorski produkt između vektora koje razapinju točke A i B te A i C : $\vec{n} = (\vec{B} - \vec{A}) \times (\vec{C} - \vec{A})$. Jednom kada znamo \vec{n} , u implicitnom obliku jednadžbe pravca jedina je nepoznanica D , no za njega tada vrijedi: $D = -\vec{n} \cdot \vec{t}$. Kako i A i B i C leže u ravnini, možemo uzeti bilo koju od tih točaka kako bismo dobili D ; npr. $D = -\vec{n} \cdot \vec{A}$.

Jednom kada znamo implicitni oblik jednadžbe ravnine, odredit ćemo sjecište (već smo opisali kako). Ako sjecište ne postoji, znamo da pravac ne probada trokut i gotovi smo. U suprotnom, neka je to sjecište točka t . Ostaje nam odrediti nalazi li se točka t unutar trokuta ili ne. Jedan od jednostavnijih načina jest izračunati baricentrične koordinatne točke t s obzirom na trokut A , B i C (i to smo već obradili – napravite to primjerice omjerom odgovarajućih površina trokuta koje ćete izračunati kao polovine normi odgovarajućih vektorskog produkata). Neka su baricentrične koordinate t_1 , t_2 i t_3 . Točka se nalazi unutar trokuta ako je $t_1 \geq 0$ i $t_2 \geq 0$ i $t_3 \geq 0$. U tom slučaju probodište trokuta i pravca postoji, i to je upravo točka t .

2.8 Ponavljanje

1. Kako se računa reflektirani vektor?
2. Napišite jednadžbu pravca u 3D prostoru u parametarskom obliku.
3. Je li u parametarskom obliku jednadžbe pravca u 3D prostoru vektor pravca jedinstven? Objasnite.
4. Kod parametarskog oblika jednadžbe pravca u 3D prostoru vektor pravca određen je razlikom krajnje i početne točke nekog segmenta. Objasnite što nam tada za neku promatrano točku govori parametar λ koji pripada toj točki?
5. Zašto se uvodi homogeni prostor? Kako se točke iz radnog preslikavaju u homogeni prostor? Kako se točke iz homogenog prostora preslikavaju u radni prostor? Je li zapis točke iz radnog prostora u homogenom prostoru jedinstven? Objasnite zašto.
6. Objasnite na koji se način definira odnos točke i pravca u 2D prostoru.
7. Napišite parametarski oblik jednadžbe ravnine u 3D prostoru, i to u vektorskom obliku i u matričnom obliku.
8. Kako se iz parametarskog oblika jednadžbe ravnine dolazi do implicitnog oblika jednadžbe ravnine?
9. U kakvoj su vezi vektor normale ravnine te implicitni oblik jednadžbe ravnine?
10. Objasnite na koji se način definira odnos točke i ravnine u 3D prostoru.
11. Kod definiranja odnosa točke i ravnine u 3D prostoru, u kakvoj su vezi smjer normale ravnine i klasifikacija "točka je iznad" odnosno "točka je ispod" ravnine?
12. Kako se računa skalarni produkt dvaju vektora? Navedite svojstva skalarnog produkta.
13. Kako se računa vektorski produkt dvaju vektora? Koja je njegova interpretacija? Navedite svojstva vektorskog produkta.
14. Kako uporabom vektorskog produkta možemo izračunati površinu trokuta?
15. Kako možete provjeriti jesu li dva pravca u 3D prostoru okomiti?

16. Kako možete provjeriti jesu li dva pravca u 3D prostoru paralelni?
17. Kako možete provjeriti sijeku li se dva pravca u 3D prostoru ili su mimoilazni?
18. Objasnite kako se dolazi do baricentričnih koordinata.
19. Objasnite kako za neku točku možemo izračunati njezine baricentrične koordinate rješavanjem sustava jednadžbi.
20. Objasnite kako za neku točku možemo izračunati njezine baricentrične koordinate uporabom omjera površina trokuta.
21. Pokažite kako se računa probodište pravca i ravnine.
22. Pokažite kako se računa probodište pravca i sfere.
23. Pokažite kako se računa probodište pravca i trokuta.

Poglavlje 3

Crtanje linija i poligona na rasterskim prikaznim jedinicama

3.1 Uvod

Kada na monitoru želimo nacrtati liniju, za to obično pozivamo ugrađenu funkciju jezika u kojem pišemo program. Pri tome zadajemo koordinate početne točke i završne točke, i očekujemo da će funkcija u što kraćem vremenu nacrtati tu liniju. No na koji način se izvodi to brzo crtanje linije? Slično pitanje vrijedi i za složenije likove – poligone, kod kojih pri tome možemo iscrtavati samo obrub, ili ih možemo i popunjavati. Oba ovog zadatka pri tome rezultiraju osvjetljavanjem određenog broja slikovnih elemenata na zaslonu, odnosno na rasterskoj prikaznoj jedinici. Naime, slika koju prikazuje monitor sastoji od niza elementarnih djelića slike - piksela (engl. *picture element*), što znači da je slika koju prikazuje monitor diskretna. Moguće je osvijetliti piksel na poziciji npr. $(0,0)$, no nije moguće osvijetliti pixel na poziciji $(1.13, 2.52)$ jer takav ne postoji. Posljedica ovakve diskretizacije slike je da prilikom crtanja linija dolazi do nazubljenosti istih (slika 3.1 ovo jasno ilustrira). Više o ovoj pojavi reći ćemo u nastavku. Pa krenimo najprije s metodom crtanja linije.

3.1.1 Bresenhamov postupak crtanja linije

Postupak kojim se danas uobičajeno crtaju linije jest implementacija Bresenhamovog postupka. Pa pogledajmo o čemu se tu radi.

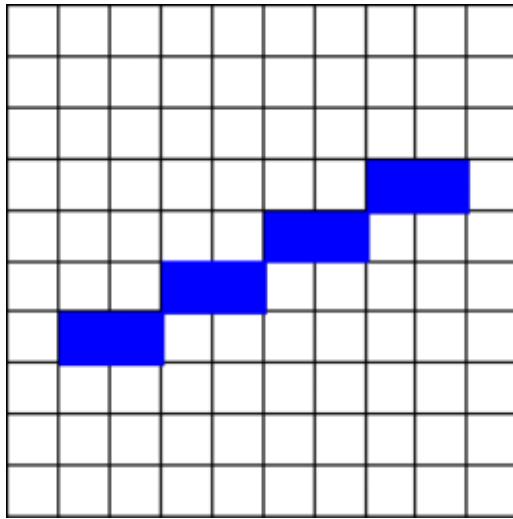
Prilikom crtanja linije poznate su nam koordinate početne i završne točke. Isto tako znamo da liniju crtamo u ravnini. Zaboravimo na tren da liniju crtamo na zaslonu. Zamislimo da imamo idealnu prikaznu jedinicu koja može prikazati sve što poželimo s beskonačno finom preciznošću. Prva stvar koju moramo napraviti jest izračunati koje sve točke moramo osvijetliti. Da bismo si olakšali razmatranje, zadajmo točke na slijedeći način:

- Početna točka T_S ima koordinate (x_s, y_s) .
- Završna točka T_E ima koordinate (x_e, y_e) .
- Vrijedi: $x_s < x_e$, $y_s < y_e$.
- Pravac je pod kutom manjim ili jednakim 45° .

Kasnije ćemo se oslobođiti ovih ograničenja no za početak neka ograničenja vrijede. Koordinate točaka T_S i T_E u ovom slučaju nisu označene standardnim oznakama $(T_{S,1}, T_{S,2})$ i $(T_{E,1}, T_{E,2})$ već su iskorištene oznake (x_s, y_s) i (x_e, y_e) budući da ćemo cijeli postupak izvoditi imajući na umu računala gdje su zasloni dvodimenzionalni sa standardnim oznakama x i y za pojedine osi.

Jednadžba pravca kroz dvije točke može se napisati prema relaciji u obliku:

$$y - y_s = \frac{y_e - y_s}{x_e - x_s} (x - x_s)$$



Slika 3.1: Prikaz linije na rasterskoj prikaznoj jedinici

ili nakon sređivanja i uvodenja oznake a za koeficijent smjera pravca (odnosno tangens kuta) te oznake b za odsječak na osi y :

$$y = a \cdot x + b$$

pri čemu je

$$a = \frac{y_e - y_s}{x_e - x_s}, \quad b = -a \cdot x_s + y_s.$$

a i b su izdvojeni kao zasebne konstante jer se mogu izračunati samo jednom na početku, s obzirom da ovise samo o konstantama. Umjesto oznaka a i b često su u uporabi i oznake k za koeficijent smjera i l za odsječak na osi y .

Uz ovu posljednju formulu pravac se može nacrtati sljedećim trivijalnim algoritmom:

```
void nacrtaj(int xs, int ys, int xe, int ye) {
    double a = (ye-ys)/(double)(xe-xs);
    double b = -a*xs+ys;

    for (int x=xs; x<=xe; x++) {
        int y = zaokruzi(a*x + b);
        osvijetli_pixel(x,y);
    }
}
```

S obzirom na diskretiziranost monitora potrebno je za svaki x pronaći odgovarajući y i tu točku osvijetliti. No ovaj algoritam, iako radi, ima jedan veliki nedostatak: sporost. Osnovni problem ovog algoritma je množenje u petlji. Za svaki x odgovarajući y računa se množenjem i to u aritmetici pomicnog zareza. Algoritam vapi za poboljšanjem!

Pogledamo još jednom što prethodno opisani algoritam radi. Varijabla x mijenja se u petlji od vrijednosti x_s do vrijednosti x_e po jedan i za svaki x se računa odgovarajući y uz uporabu operacije množenja. Pogledajmo malo bolje što to program zapravo računa:

$$\begin{aligned} y|_{x=x_s} &= a \cdot x_s + b = y_s \\ y|_{x=x_s+1} &= a \cdot (x_s + 1) + b = a \cdot x_s + b + a = y|_{x=x_s} + a \\ y|_{x=x_s+2} &= a \cdot (x_s + 2) + b = a \cdot x_s + b + 2a = y|_{x=x_s+1} + a \\ y|_{x=x_s+3} &= a \cdot (x_s + 3) + b = a \cdot x_s + b + 3a = y|_{x=x_s+2} + a \end{aligned}$$

Pogledamo li desne strane u svakom retku, možemo vidjeti da se svaki redak može dobiti tako da se prethodnom doda vrijednost varijable a . To je izvrstan rezultat jer nam ukazuje na to da nam više

ne treba vremenski zahtjevno množenje. Možda bi bio problem s prvim retkom, jer po našoj filozofiji on nema prethodnika pa bismo tu trebali koristiti množenje, no nije tako. Vrijednost prvog retka već nam je zadana i iznosi y_s . Pogledajmo sada što smo dobili.

```
void nacrtaj(int xs, int ys, int xe, int ye) {
    int x,y_pom;
    double a,y;

    a = (ye-ys)/(double)(xe-xs);

    y=ys;
    for (x=xs; x<=xe; x++) {
        y_pom = zaokruzi(y);
        osvijetli_pixel(x,y_pom);
        y = y + a;
    }
}
```

Poboljšanje je već značajno. Izvan petlje vrijednost varijable y postavlja se na početnu vrijednost. U petlji je uvedena pomoćna varijabla y_pom koja pamti vrijednost varijable y nakon zaokruživanja jer varijablu y ne smijemo dirati budući da nam treba i u nastavku.

Pogledajmo što još ne valja. Prečesto zovemo funkciju $zaokruzi(y)$. Bilo bi jako zgodno kada bismo mogli i bez zaokruživanja znati koja je zaokružena vrijednost y koordinate, ili ako ništa drugo, pokušati funkciju zvati rjeđe. Naravno, i to se može. Evo kako.

3.1.2 Izvod Bresenhamovog algoritma s decimalnim brojevima

Ideja je sljedeća. Početnu zaokruženu vrijednost y koordinate znamo: to je y_s , tj. za $x = x_s$ imamo $y = y_s$. Trebamo dakle osvijetliti piksel na poziciji (x_s, y_s) . Kada se pomaknemo za jedan piksel udesno, vrijednost y koordinate poveća se za koeficijent a . Kako smo postavili ograničenje da je pravac pod kutom manjim ili jednakim 45° , to znači da se vrijednost varijable a kreće između 0 i 1. Uzmimo za primjer da a iznosi 0.2. Tada nakon jednog pomaka u desno y se je povećao za 0.2 te iznosi $y = y_s + 0.2$. Zaokruživanjem ove vrijednosti dolazi se opet do vrijednosti y_s te se y koordinata točke koju trebamo osvijetliti ne razlikuje od prethodnog koraka. Znači, trebamo osvijetliti piksel na poziciji $(x_s + 1, y_s)$. Pomaknimo se za još jedan piksel u desno. y -koordinatu opet treba uvećati za vrijednost varijable a . Sada to iznosi $y = y_s + 0.2 + 0.2 = y_s + 0.4$. Zaokruživanjem se opet dolazi do vrijednosti $y = y_s$, pa osvjetljavamo piksel na poziciji $(x_s + 2, y_s)$. Idemo za još jedan piksel u desno: $y = y_s + 0.4 + 0.2 = y_s + 0.6$. Konačno, zaokruživanjem ove vrijednosti penjemo se za jedan piksel prema gore: $y = y_s + 1$, i osvjetljavamo piksel na poziciji $(x_s + 2, y_s + 1)$.

Označimo cjelobrojnu vrijednost koordinate y s y_c , a pridodani decimalni dio s y_f . Tada gornji postupak možemo opisati ovako: osvjetljavamo točke na poziciji $(x_s + k, y_c)$. Na početku je $y_c = y_s$ a $y_f = 0$. Svakim pomakom u desno y_f uvećavamo za koeficijent a . Onog trenutka kada y_f pređe (ili dođe na) 0.5, y_c uvećavamo za jedan.

U ovom trenutku možemo nastaviti na dva načina.

1. Možemo nastaviti prilikom pomaka u desno s dodavanjem vrijednosti varijable a varijabli y_f . U tom slučaju sljedeće uvećavanje y_c -a za jedan biti će kada y_f prekorači 1.5, pa sljedeće kada prekorači 2.5 itd. No ova ideja i nije baš najbolja jer opet dozvoljava da y_f postane veći od jedan pa moramo voditi računa da svaki puta cijelobrojni dio "zaboravimo" i promatramo samo decimalni ostatak, što je opet vremenski zahtjevno.
2. Možemo od y_f oduzeti 1 i time poništiti nagomilanu pogrešku te y_f zadržati u opsegu od -0.5 do 0.5 čime nam je usporedba znatno olakšana.

Budući da je drugi postupak bolji, odlučit ćemo se za njega. No u tom slučaju treba još pojasniti zašto se od y_f oduzima baš 1. Razmislimo malo što nam predstavlja y_f . Crtajmo liniju iz početne točke s y koordinatom jednakom 0 i ponovimo ukratko gore opisani postupak. Prvo smo bili u $y = 0$, odnosno $y_c = 0$, $y_f = 0$. Zatim smo došli u $y = 0.2$, tj. $y_c = 0$, $y_f = 0.2$. Zatim $y = 0.4$, tj. $y_c = 0$, $y_f = 0.4$, i konačno $y = 0.6$, tj. $y_c = 1$, $y_f = 0.6$. Vrijednost pohranjena u y_f nam zapravo govori

koliko smo pobjegli od cijelobrojne koordinate. Npr. za $y = 0.4$, tj. $y_c = 0$, $y_f = 0.4$, osvjetlit ćemo piksel s y -koordinatom 0, dok smo realno gledajući već 0.4 piksela iznad. Onog trena kada od piksela pobjegnemo za 0.5 ili više (npr. za $y_f = 0.6$), prema dogovoru uvećavamo y_c za jedan. No tada više nismo u točki s y -koordinatom 0, već s y -koordinatom 1. A to za pogrešku y_f znači da više nismo 0.6 piksela iznad, nego 0.4 piksela ispod trenutnog piksela određenog s y_c , što znači da pogreška od $y_f = 0.6$ prelazi u $y_f = 0.6 - 1 = -0.4$.

Implementacija ovog algoritma dana je u nastavku.

```
void nacrtaj(int xs, int ys, int xe, int ye) {
    int x, yc;
    double a, yf;

    a = (ye - ys) / (double)(xe - xs);

    yc = ys; yf = 0.0;
    for (x = xs; x <= xe; x++) {
        osvijetli_pixel(x, yc);
        yf = yf + a;
        if (yf >= 0.5) {
            yf = yf - 1.0;
            yc = yc + 1;
        }
    }
}
```

Uvedemo li još samo jednu minornu modifikaciju, a to je da inicijalno za y_f uzmemmo vrijednost -0.5 , te poslije usporedbu radimo s $0.5 - 0.5 = 0$ (dakle nulom) dobili smo Bresenhamov algoritam!

```
void bresenham_nacrtaj(int xs, int ys, int xe, int ye) {
    int x, yc;
    double a, yf;

    a = (ye - ys) / (double)(xe - xs);

    yc = ys; yf = -0.5;
    for (x = xs; x <= xe; x++) {
        osvijetli_pixel(x, yc);
        yf = yf + a;
        if (yf >= 0.0) {
            yf = yf - 1.0;
            yc = yc + 1;
        }
    }
}
```

Može li bolje? Naravno...

3.1.3 Izvod Bresenhamovog algoritma s cijelim brojevima

Do osnovnog Bresenhamovog postupka došli smo krenuvši iz samo jednog zahtjeva – brzine. Pogledom na sam algoritam postavlja se pitanje može li još bolje? Trn u oku u opisanom algoritmu svakako su brojevi s pomičnim zarezom. Pokazuje se da se cijeli algoritam može prebaciti u cijelobrojnu domenu, a opće je poznato da su operacije s cijelim brojevima daleko brže od aritmetike u pomičnom zarezu. Stoga ćemo se u nastavku pozabaviti prilagodbom osnovnog Bresenhamovog algoritma tako da proradi s cijelim brojevima.

Osnovni element koji je uveo decimalne brojeve u igru bio je koeficijent smjera koji smo računali prema formuli:

$$a = \frac{y_e - y_s}{x_e - x_s}.$$

Ovaj koeficijent koristili smo prilikom izračuna pogreške (varijabla y_f). Pri tome smo, nakon svakog pomaka po x -u za jedan, pogrešku računali prema formuli:

$$y_f = y_f + a.$$

Idemo to malo raspisati. Uvrštavanjem izraza za a dobiva se:

$$y_f = y_f + \frac{y_e - y_s}{x_e - x_s} = \frac{y_f \cdot (x_e - x_s) + y_e - y_s}{x_e - x_s}.$$

Množenjem čitave jednadžbe nazivnikom razlomka s desne strane dobivamo:

$$y_f \cdot (x_e - x_s) = y_f \cdot (x_e - x_s) + y_e - y_s.$$

Uvođenjem $y'_f = y_f \cdot (x_e - x_s)$ izraz prelazi u:

$$y'_f = y'_f + y_e - y_s.$$

Ovaj posljednji redak nam govori da umjesto dosadašnje pogreške možemo pamtitи pogrešku pomnoženu s koeficijentom $x_e - x_s$. Kako su to sve cijeli brojevi, ovdje smo se riješili sporih decimalnih brojeva. Sljedeći korak je rješavanje inicijalnih uvjeta. Naime, inicijalno vrijednost pogreške postavljamo na $-\frac{1}{2}$. To opet možemo raspisati:

$$\begin{aligned} y_f &= -\frac{1}{2} \quad | \cdot (x_e - x_s) \quad \Rightarrow \quad y_f \cdot (x_e - x_s) = -\frac{x_e - x_s}{2} \\ y'_f &= -\frac{x_e - x_s}{2} \end{aligned}$$

Skoro pa dobro. Još da se riješimo dvojke u nazivniku i svi naši problemi su riješeni. Dvojke ćemo se jednostavno riješiti tako da sve pomnožimo s 2. No tada na lijevoj strani umjesto nove pogreške stoji njezina dvostruka vrijednost. To će nas još prisiliti da kod izvoda nove pogreške sve pomnožimo s dvojkom, te će se dobiti redom izrazi opisani u nastavku.

- Za pogrešku ćemo koristiti izraz:

$$\begin{aligned} y_f &= y_f + \frac{y_e - y_s}{x_e - x_s} = \frac{y_f \cdot (x_e - x_s) + y_e - y_s}{x_e - x_s} | \cdot 2(x_e - x_s) \\ 2 \cdot y_f \cdot (x_e - x_s) &= 2 \cdot y_f \cdot (x_e - x_s) + 2 \cdot (y_e - y_s). \end{aligned}$$

Uvođenjem $y'_f = 2 \cdot y_f \cdot (x_e - x_s)$ izraz prelazi u:

$$y'_f = y'_f + 2 \cdot (y_e - y_s).$$

- Inicijalno dodjeljivanje tada prelazi u

$$\begin{aligned} y_f &= -\frac{1}{2} \quad | \cdot 2 \cdot (x_e - x_s) \quad \Rightarrow \quad 2 \cdot y_f \cdot (x_e - x_s) = -2 \cdot \frac{x_e - x_s}{2} = -(x_e - x_s) \\ y'_f &= -(x_e - x_s). \end{aligned}$$

- uz ove označke oduzimanje jedinice od pogreške može se zapisati kao:

$$\begin{aligned} y_f &= y_f - 1 \quad | \cdot 2 \cdot (x_e - x_s) \quad \Rightarrow \quad 2 \cdot y_f \cdot (x_e - x_s) = 2 \cdot y_f \cdot (x_e - x_s) - 2(x_e - x_s) \\ y'_f &= y'_f - 2(x_e - x_s). \end{aligned}$$

Imajući u vidu ove izmjene, može se napisati Bresenhamov algoritam s cijelim brojevima. Pri tome će se umjesto označke y'_f koristiti standardna označka y_f podrazumijevajući da se pri tome govori o novo-definiranoj pogreški. Tako napisana varijanta Bresenhamovog algoritma sa cijelim brojevima prikazana je u nastavku.

```

void bresenham_nacrtaj_cjelobrojni1(int xs, int ys, int xe, int ye) {
    int xc, korekcija;
    int a, yf;

    a = 2*(ye-ys);

    xc=ys; yf=-(xe-xs); korekcija=-2*(xe-xs);
    for( x = xs; x <= xe; x++ ) {
        osvijetli_pixel(x,xc);
        yf=yf+a;
        if(yf>=0) {
            yf=yf+korekcija;
            xc=xc+1;
        }
    }
}

```

3.1.4 Kutevi od 0° do 90°

Sada kada smo se riješili decimalnih brojeva, vrijeme da se riješimo i ograničenja vezanih uz kutove. Pa idemo postupno. Razmatranje ćemo raditi za postupak s cijelim brojevima, ali ćemo imati stalno u vidu kako smo do tog postupka zapravo došli. Pravac pod kutom od 90° za algoritam s decimalnim brojevima (osnovna izvedba) bio je neizvediv. Naime, tangens kuta je tada beskonačno i imamo dijeljenje s nulom (i vjerojatno nasilni prekid programa). No, algoritam s cijelim brojevima nema dijeljenja pa ovo više nije problem.

Za kuteve od 0° do 45° algoritam smo već izveli. No zašto smo se ograničili na to područje? Odgovor opet treba tražiti u iznosu tangensa kuta. Naime, na tom intervalu on se kreće u granicama od nula do jedan, *osiguravajući pri tome da ćemo se pomakom za jedan piksel u desno pomaknuti maksimalno za jedan piksel prema gore*. A što ako dopustimo da tangens postane veći od 1? To znači da bismo se jednim pomakom u desno mogli pomaknuti i za više piksela prema gore (vodeći računa o tome da ih sve treba osvijetliti). Međutim, pojavljuje se problem u određivanju koje sve piksele pri tom penjanju treba osvijetliti. Da ne ulazimo dublje u prazne rasprave o ovome, rješenje ćemo potražiti na drugi način.

Ako je kut između pravca i apscise veći od 45° , tada je očito kut između pravca i ordinate manji od 45° . Znači, ako sada jednostavno zamijenimo osi prilikom postupka crtanja, umjesto problematičnog višepikselnog pomaka prema gore sa svakim pomakom u desno dobivamo maksimalno jednopikselski pomak u desno sa svakim pomakom prema gore. I to je rješenje. Sve što treba napraviti jest ispitati je li tangens kuta veći od 1, tj. je li kut veći od 45° . Ako nije, koristimo već poznati algoritam; ako je mijenjamo uloge osima i kopiramo algoritam. Evo implementacije:

```

void bresenham_nacrtaj_cjelobrojni2(int xs, int ys, int xe, int ye) {
    int xc, korekcija;
    int a, yf;

    if(ye-ys <= xe-xs) {
        a = 2*(ye-ys);
        xc=ys; yf=-(xe-xs); korekcija=-2*(xe-xs);
        for( x = xs; x <= xe; x++ ) {
            osvijetli_pixel(x,xc);
            yf=yf+a;
            if(yf>=0) {
                yf=yf+korekcija;
                xc=xc+1;
            }
        }
    } else {
        // zamijeni x i y koordinate
        xe=ye; ye=x;
        xs=ys; ys=x;
        a = 2*(ye-ys);
    }
}

```

```

y c=ys; yf=-(xe-xs); korekcija=-2*(xe-xs);
for( x = xs; x <= xe; x++ ) {
    osvijetli_pixel(yc,x);
    yf=yf+a;
    if(yf>=0) {
        yf=yf+korekcija;
        yc=yc+1;
    }
}
}
}

```

Funkcija kreće s ispitivanjem je li razlika po y -u manja od razlike po x -u. Ako je, pravac je pod kutom manjim od 45° i koristi se već izvedeni algoritam. Ako je razlika po y -u veća od razlike po x -u, koristeći pomoćnu varijablu x zamjenjuju se x i y koordinate točaka i nastavlja se s crtanjem. U petlji se u ovom slučaju može uočiti još jedna razlika: funkciji `osvijetli_piksela` predajemo na prvi pogled zamijenjene koordinate. No imajući u vidu da smo već prethodno napravili jednu zamjenu, ono što se mijenja u varijabli yc zapravo je vrijednost x komponente točke i obrnuto.

3.1.5 Kutevi od 0° do -90°

Osnovna razlika od prethodnog slučaja je okomito gibanje koje sada ide prema dolje, umjesto prema gore. Zbog toga yc treba umanjavati, a ne uvećavati za jedan. Isto tako, postupak pri računanju pogreške se ponešto modificira, no modifikacije su čisto kozmetičke prirode. Evo algoritma:

```

void bresenham_nacrtaj_cjelobrojni3(int xs, int ys, int xe, int ye) {
    int x, yc, korekcija;
    int a, yf;

    if(-(ye-ys) <= xe-xs) {
        a = 2*(ye-ys);
        yc=ys; yf=(xe-xs); korekcija=2*(xe-xs);
        for( x = xs; x <= xe; x++ ) {
            osvijetli_pixel(x,yc);
            yf=yf+a;
            if(yf<=0) {
                yf=yf+korekcija;
                yc=yc-1;
            }
        }
    } else {
        x=xe; xe=ys; ys=x;
        x=xs; xs=ye; ye=x;
        a = 2*(ye-ys);
        yc=ys; yf=(xe-xs); korekcija=2*(xe-xs);
        for( x = xs; x <= xe; x++ ) {
            osvijetli_pixel(yc,x);
            yf=yf+a;
            if(yf<=0) {
                yf=yf+korekcija;
                yc=yc-1;
            }
        }
    }
}

```

Od izmjena u odnosu na kod koji crta pravce pod kutovima od 0° do 90° mogu se navesti sljedeće:

- inicijalna vrijednost pogreške promijenila je predznak,
- y -komponenta se ne uvećava za jedan već se umanjuje za jedan te
- kako je $y_s > y_e$ ispitivanje $(y_e - y_s) \leq (x_e - x_s)$ pretvoreno je u $-(y_e - y_s) \leq (x_e - x_s)$ da bi se poništio negativan predznak rezultata na lijevoj strani.

3.1.6 Konačan kod za sve kuteve

Do sada smo obradili slučajeve kada smo se pri crtaju pravca gibali od lijeva u desno. Ostalo nam je obraditi crtanje pravca kojeg bismo trebali crtati s desna u lijevo. No za ovo nam ne trebaju posebni algoritmi: ukoliko je pravac zadan tako da se crta s desna u lijevo, zamjenom početne i završne koordinate dobivamo pravac koji se crta s lijeva u desno.

```
void bresenham_nacrtaj_cjelobrojni(int xs, int ys, int xe, int ye) {
    if( xs <= xe ) {
        if( ys <= ye ) {
            bresenham_nacrtaj_cjelobrojni2(xs,ys,xe,ye);
        } else {
            bresenham_nacrtaj_cjelobrojni3(xs,ys,xe,ye);
        }
    } else {
        if( ys >= ye ) {
            bresenham_nacrtaj_cjelobrojni2(xe,ye,xs,ys);
        } else {
            bresenham_nacrtaj_cjelobrojni3(xe,ye,xs,ys);
        }
    }
}
```

Funkcija ovisno o predanim točkama poziva odgovarajuću funkciju i pri tome po potrebi zamjenjuje početnu i krajnju točku. Prisjetimo se što smo već implementirali:

- funkcija `bresenham_nacrtaj_cjelobrojni2` crta linije pod kutovima od 0° do 90° , uz $x_s < x_e$,
- funkcija `bresenham_nacrtaj_cjelobrojni3` crta linije pod kutovima od 0° do -90° , uz $x_s < x_e$ te
- funkcija `bresenham_nacrtaj_cjelobrojni` kombinirajući prethodne dvije crta linije pod svim kutovima, uz proizvoljan odnos koordinata x_s i x_e .

3.2 Konveksni poligon

Poligon je lik koji nastaje slijednim povezivanjem zadanih točaka koje predstavljaju vrhove poligona. Pri tome se povezuje i završna točka s početnom. Točke koje predstavljaju vrhove poligona označavati ćemo oznakom T_i , gdje je i indeks vrha. Točke ćemo zadavati u radnom prostoru i redoslijed zadavanja bit će točno definiran, jer taj redoslijed određuje kako ćemo povezivati točke poligona. Povezivanje vrhova poligona obavlja se linijama (segmentima pravaca), koje čine bridove poligona. Bridove ćemo označavati oznakom b_i , gdje je i indeks brida. Može se uočiti da je broj bridova uvijek jednak broju vrhova poligona. Poligon je konveksan ako ne postoji spojnica dviju proizvoljnih točaka poligona koja bi prolazila izvan poligona. Jasnije tumačenje daje slika 3.2.

3.2.1 Matematički opis poligona

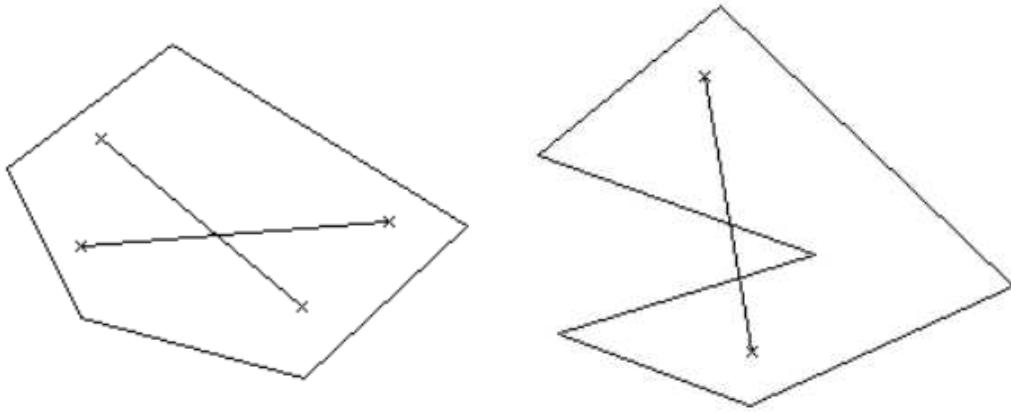
Osnovno što će nas kod poligona interesirati, i pomoću čega ćemo donositi razne zaključke jesu jednadžbe bridova poligona. U ovom razmatranju ograničit ćemo se na poligone u dvije dimenzije. Prema slici 3.3, brid b_i određen je vrhovima:

$$b_i \dots \begin{cases} T_i, T_{i+1} & 0 < i < n \\ T_i, T_1 & i = n \end{cases}$$

Pri tome ćemo bridove prikazivati u homogenom prostoru. Sve koordinate poligona prije uporabe proširit ćemo homogenim parametrom 1, budući da vrhove poligona zadajemo u radnom prostoru.

Parametarska jednadžba pravca (u radnom prostoru) glasi:

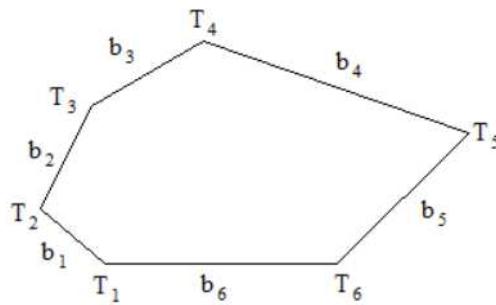
$$T_b = \begin{cases} (T_{i+1} - T_i) \cdot \lambda + T_i & 0 < i < n \\ (T_1 - T_i) \cdot \lambda + T_i & i = n \end{cases}$$



(a) Konveksni poligon

(b) Konkavni poligon

Slika 3.2: Razlika između konveksnog i konkavnog poligona



Slika 3.3: Poligon

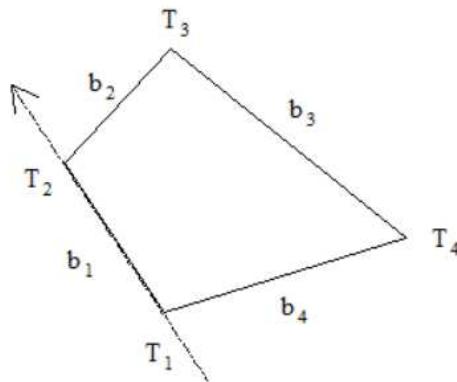
gdje je T_b proizvoljna točka pravca.

Jednadžba pravca u homogenom prostoru može se dobiti kao \times -prodotku vrhova pravca:

$$\begin{aligned} T_i \times T_{i+1} &= \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \\ T_{i,1} & T_{i,2} & 1 \\ T_{i+1,1} & T_{i+1,2} & 1 \end{bmatrix} \\ &= \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} \cdot \begin{bmatrix} T_{i,2} - T_{i+1,2} \\ -(T_{i,1} - T_{i+1,1}) \\ T_{i,1}T_{i+1,2} - T_{i,2}T_{i+1,1} \end{bmatrix} \\ &= \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} \cdot b_i \quad \text{za } 0 < i < n \end{aligned}$$

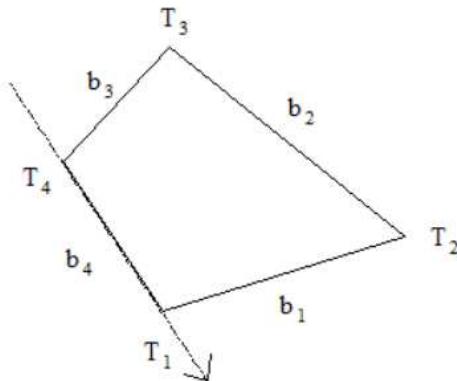
$$\begin{aligned} T_i \times T_1 &= \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \\ T_{i,1} & T_{i,2} & 1 \\ T_{1,1} & T_{1,2} & 1 \end{bmatrix} \\ &= \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} \cdot \begin{bmatrix} T_{i,2} - T_{1,2} \\ -(T_{i,1} - T_{1,1}) \\ T_{i,1}T_{1,2} - T_{i,2}T_{1,1} \end{bmatrix} \\ &= \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} \cdot b_i \quad \text{za } i = n \end{aligned}$$

Oznaka $T_{i,k}$ predstavlja k -tu komponentu točke T_i . Kako smo se ogradiili na 2D prostor, umjesto $T_{i,1}$ mogli smo pisati $T_{i,x}$ a umjesto $T_{i,2}$ mogli smo pisati $T_{i,y}$. Iz razloga koji će kasnije biti objašnjeni, posebno je važno poštivati redoslijed vrhova, odnosno prilikom izračuna jednadžbi bridova vrhove uzimati uvijek na isti način (u smjeru kazaljke na satu ili obrnuto, ali konzistentno).



b₂ je ispod b₁
(uz označeni smjer kretanja nalazi se desno)

Slika 3.4: Orijentacija vrhova poligona – u smjeru kazaljke na satu



b₂ je iznad b₁
(uz označeni smjer kretanja nalazi se lijevo)

Slika 3.5: Orijentacija vrhova poligona – u smjeru suprotnom od smjera kazaljke na satu

3.2.2 Orijentacija vrhova poligona

Poligon može imati vrhove zadane tako da se njihovim obilaskom gibamo u smjeru kazaljke na satu ili u smjeru suprotnom od smjera kazaljke na satu. Orijentacija vrhova bit će nam bitna u dalnjim razmatranjima, pa je nužno objasniti kako se ista može ispitati. U prethodnom podpoglavlju izračunali smo jednadžbe bridova. Na temelju tih jednadžbi može se ustanoviti u kojem su odnosu neka promatrana točka i zadani brid, odnosno pravac na kojem brid leži. Mogući odgovori su pri tome: točka se nalazi iznad pravca, točka se nalazi na pravcu te točka se nalazi ispod pravca. Pametnim odabirom ispitne točke jednostavno ćemo doći do spoznaje o orientaciji vrhova. Pogledajmo sliku 3.4.

Brid b_1 određen je vrhovima T_1 i T_2 . U kakvom je odnosu taj brid s točkom T_3 ? Točka T_3 nalazi se ispod brida. Isto vrijedi i za brid b_2 određen vrhovima T_2 i T_3 i točku T_4 . Točka T_4 je ispod brida b_2 . Obidemo li tako sve bridove u krug ispitujući odnos tog brida i prvog sljedećeg vrha poligona, rezultat je uvijek isti. Pogledamo li kako su zadani vrhovi našeg poligona, vidimo da su zadani u smjeru kazaljke na satu. Ova spoznaja daje naslutiti kriterij koji bi se mogao koristiti, no prije nego što proglašimo kriterij ispravnim, pogledajmo i drugi slučaj. Na slici 3.5 prikazan je poligon s vrhovima zadanim u smjeru suprotnom od smjera kazaljke na satu.

Pogledamo sada odnos brida b_1 određenog vrhovima T_1 i T_2 i točke T_3 . Točka T_3 nalazi se iznad brida b_1 . Na sličan način opet se može vidjeti da ovo vrijedi za svaki brid i prvi sljedeći vrh poligona. Dakle, kriterij je ispravan. Evo načina kako provjeriti orientaciju vrhova poligona.

- Vrhovi poligona zadani su u smjeru kazaljke na satu ako vrijedi:

$$(\forall i) T_j \cdot b_i \leq 0 \quad j = \begin{cases} j = i + 2, & \text{za } 0 < i < n - 1 \\ j = i - n + 2, & \text{za } n - 1 \leq i \leq n \end{cases}.$$

- Vrhovi poligona zadani su u smjeru suprotnom od smjera kazaljke na satu ako vrijedi:

$$(\forall i) T_j \cdot b_i \geq 0 \quad j = \begin{cases} j = i + 2, & \text{za } 0 < i < n - 1 \\ j = i - n + 2, & \text{za } n - 1 \leq i \leq n \end{cases}.$$

Ako ustanovimo da je poligon zadan uz jednu orijentaciju vrhova, a nama treba suprotna orijentacija vrhova, tada se jednostavno može zamijeniti redoslijed vrhova poligona i ponovno preračunati jednadžbe bridova.

Imajući u vidu da radimo s konveksnim poligonom kod kojeg su vrhovi zadani nekim konzistentnim redoslijedom, može se jednostavno pokazati sljedeće:

- ako vrijedi da

$$(\exists i) T_j \cdot b_i < 0 \quad j = \begin{cases} j = i + 2, & \text{za } 0 < i < n - 1 \\ j = i - n + 2, & \text{za } n - 1 \leq i \leq n \end{cases}$$

dakle da postoji bar jedan vrh (prvi iza dva koja određuju brid) takav da se nalazi ispod brida, tada mora vrijediti:

$$(\forall i) T_j \cdot b_i \leq 0 \quad j = \begin{cases} j = i + 2, & \text{za } 0 < i < n - 1 \\ j = i - n + 2, & \text{za } n - 1 \leq i \leq n \end{cases}$$

odnosno da to vrijedi za svaki vrh. Ovo proizlazi iz činjenice da je poligon konveksan. Isto tako se može pokazati i da

- ako vrijedi

$$(\exists i) T_j \cdot b_i > 0 \quad j = \begin{cases} j = i + 2, & \text{za } 0 < i < n - 1 \\ j = i - n + 2, & \text{za } n - 1 \leq i \leq n \end{cases}$$

dakle da postoji bar jedan vrh (prvi iza dva koja određuju brid) takav da se nalazi iznad brida, tada mora vrijediti:

$$(\forall i) T_j \cdot b_i \geq 0 \quad j = \begin{cases} j = i + 2, & \text{za } 0 < i < n - 1 \\ j = i - n + 2, & \text{za } n - 1 \leq i \leq n \end{cases}$$

odnosno da to vrijedi za svaki vrh.

Prethodna dva uvjeta mogu se složiti u *kriterij koji određuje tip poligona*: konveksan ili konkavan. Možemo reći ovako. Poligon je konveksan ako vrijedi:

$$(\forall i) T_j \cdot b_i \leq 0 \quad j = \begin{cases} j = i + 2, & \text{za } 0 < i < n - 1 \\ j = i - n + 2, & \text{za } n - 1 \leq i \leq n \end{cases}$$

ili

$$(\forall i) T_j \cdot b_i \geq 0 \quad j = \begin{cases} j = i + 2, & \text{za } 0 < i < n - 1 \\ j = i - n + 2, & \text{za } n - 1 \leq i \leq n \end{cases}.$$

Konveksnost zahtijeva da ukoliko za neki vrh T_{i+2} utvrdimo da je ispod (iznad) brida b_i (određenog vrhovima T_i i T_{i+1}) tada i svi vrhovi T_{j+2} moraju biti ispod (iznad) svih bridova b_j poligona. Ukoliko pak utvrdimo da su neki vrhovi iznad a neki ispod odgovarajućeg brida, tada je poligon sigurno konkavan.

Ovdje izrečeni kriterij možemo iskoristiti i na drugi način. Ako sigurno znamo da je poligon konveksan, tada se određivanje orijentacije vrhova svodi na jedno jedino ispitivanje. Npr. uzmemo točku T_3 i brid b_1 (određen točkama T_1 i T_2). Ako je T_3 ispod b_1 , orijentacija je u smjeru kazaljke na satu; ako je T_3 iznad b_1 , orijentacija je u smjeru suprotnom od smjera kazaljke na satu. Jedino što se ovdje nepredviđenoga može dogoditi jest da je poligon zadan "čudno" pa da vrh T_3 leži na bridu b_1 . No tada se jednostavno pomaknemo na sljedeći brid i slijedeću točku.

3.2.3 Odnos točke i poligona

Želimo utvrditi u kakvom je odnosu proizvoljna točka T i poligon – je li točka unutar poligona ili je izvan. To možemo napraviti sličnim postupkom kao i kod provjere orijentacije bridova. Potrebno je pogledati odnos svakog brida i zadane točke T . Isto tako je potrebno poznavati orijentaciju bridova poligona. Prema slikama 3.6a, 3.6b i 3.6c, kriterij glasi:

- Točka T je unutar poligona ako su vrhovi poligona zadani u smjeru kazaljke na satu i ako vrijedi:

$$(\forall i) T \cdot b_i \leq 0 \quad \text{za } 1 \leq i \leq n.$$

Dakle, točka je unutar poligona ako su vrhovi poligona zadani u smjeru kazaljke na satu i ako točka je ispod *svakog* brida.

- Točka T je unutar poligona ako su vrhovi poligona zadani u smjeru suprotnom od smjera kazaljke na satu i ako vrijedi:

$$(\forall i) T \cdot b_i \geq 0 \quad \text{za } 1 \leq i \leq n.$$

Znači, točka je unutar poligona ako su vrhovi poligona zadani u smjeru suprotnom od smjera kazaljke na satu i točka je iznad *svakog* brida.

- I matematički "pametno" za kraj: točka T je izvan poligona ako nije unutra.

3.2.4 Bojanje konveksnog poligona

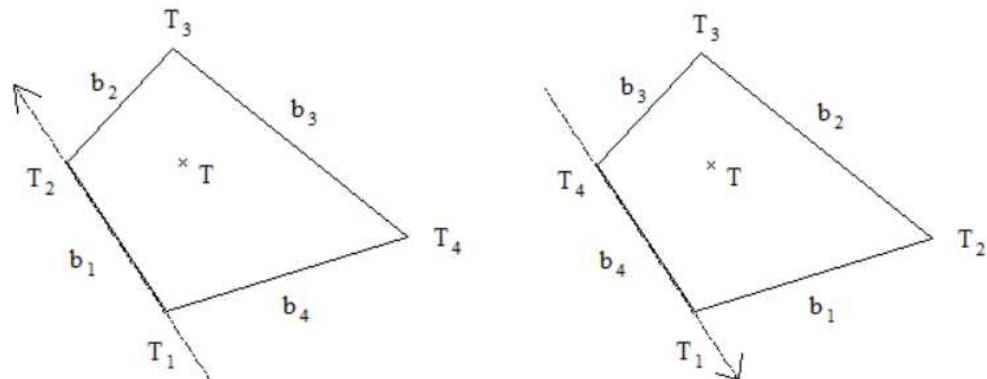
Postoji nekoliko načina za bojanje poligona. U nastavku ćemo opisati postupak s putujućom vodoravnim zrakom (engl. *scan-line*). Ideja je sljedeća: za svaki y pustit ćemo vodoravnu zraku i pratiti gdje se ona siječe s poligonom (tj. njegovim bridovima). Uz pretpostavku da je poligon konveksan, dobit ćemo ili nula ili dva sjecišta. Ako sjecišta postoje, spojiti ćemo ih vodoravnom linijom zadane boje. Da bismo osigurali da za svaku zraku sjecišta uvijek postoje, prije bojanja proći ćemo kroz sve vrhove poligona i zapamtiti najveću i najmanju y -koordinatu. S ovim podatkom vodoravnu zraku ćemo puštati za sve y -e od y_{min} do y_{max} . Budući da je zraka zapravo pravac $y = \text{konst}$, traženjem sjecišta sa svim zrakama čiji se y kreće od y_{min} do y_{max} proći ćemo cijeli poligon. Prilikom prolaska kroz točke korisno će biti zapamtiti i najmanju i najveću x -koordinatu.

Opisana ideja čini se vrlo jednostavnom, no računski je dosta zahtjevna. Stoga ćemo uvesti dodatne olakšice i iskoristiti činjenicu da bojimo konveksan poligon. U tom slučaju ideja je sljedeća. Poligon možemo podijeliti na lijevi dio (gdje vrhovi rastu prema većim y vrijednostima) i desni dio (gdje vrhovi padaju prema nižim y vrijednostima). To je lijepo ilustrirano na slici 3.7.

Tražit ćemo sjecišta sa pravcima koje definiraju bridovi poligona, kako je to prikazano na slici 3.8, i pamtitи njihove x koordinate. y -koordinate već znamo – sve su iste i odgovaraju y -vrijednosti za koju promatramo vodoravnu zraku. Uočimo pri tome da, formalno govoreći, doista ne tražimo sjecišta zrake i bridova (na slici 3.8 brid b_3 i zraka nemaju sjecišta) već tražimo sjecišta pravaca koji su bridovi dio i zrake (na slici 3.8 pravac koji odgovara bridu b_3 i zraka imaju sjecišta; to je L_3). Uočimo da, ako brid nije vodoravan, odgovarajući pravac i zraka uvijek će imati sjecište. Međutim, promatramo li samo sjecišta koja odgovaraju pravcima lijevih bridova, te ako analiziramo samo zrake $y_{min} \leq y \leq y_{max}$, tada najdesnije sjecište (tj. sjecište koje ima maksimalnu x -koordinatu) sigurno odgovara sjecištu poligona i zrake. Isto tako, promatramo li samo sjecišta koja odgovaraju pravcima desnih bridova, te ako analiziramo samo zrake $y_{min} \leq y \leq y_{max}$, tada najlijevije sjecište (tj. sjecište koje ima minimalnu x -koordinatu) sigurno odgovara sjecištu poligona i zrake. Razmislite zašto je tome tako.

Iz prethodnog opisa sada je jasno što treba pamtiti:

- za lijeve bridove pamtit ćemo ono sjecište koje ima najveću x -koordinatu i taj x označit ćemo s L , a
- za desne bridove pamtit ćemo ono sjecište koje ima najmanju x -koordinatu, i taj x označit ćemo s D .

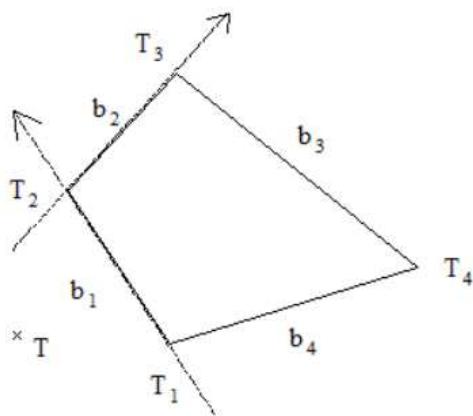


T je ispod b_1
(uz označeni smjer kretanja nalazi se desno)

(a) Slučaj 1

T je iznad b_1
(uz označeni smjer kretanja nalazi se lijevo)

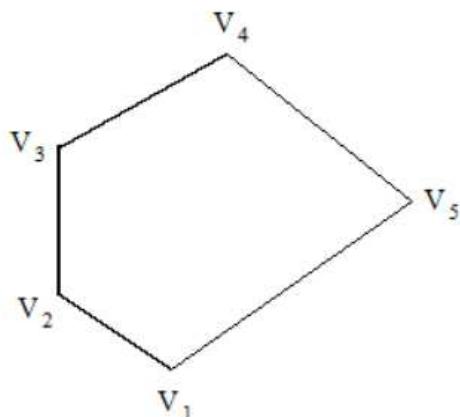
(b) Slučaj 2



T je iznad b_1 ali je ispod b_2

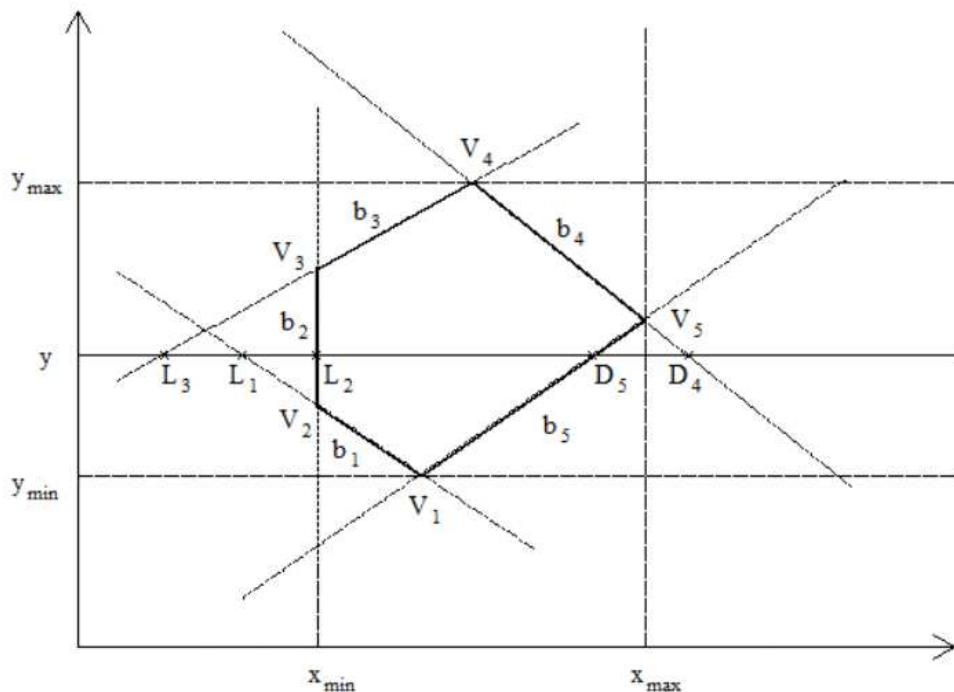
(c) Slučaj 3

Slika 3.6: Odnos točke i poligona



Lijevi bridovi označeni su podebljano,
dok su desni povučeni običnom linijom.

Slika 3.7: Podjela bridova na lijeve i desne kod konveksnog poligona



Slika 3.8: Sjecišta vodoravne zrake i pravaca određenih bridovima konveksnog poligona

Nakon što nađemo sjecište sa svim bridovima, iscrtat ćemo liniju između L i D koordinata na visini y .

Formalno bridove možemo podijeliti na lijeve i desne na sljedeći način:

- brid b_i određen vrhovima T_i i T_{i+1} je lijevi ako je $T_{i,y} < T_{i+1,y}$, a
- brid b_i određen vrhovima T_i i T_{i+1} je desni ako je $T_{i,y} > T_{i+1,y}$.

Struktura podataka za pamćenje konveksnog poligona

Pri opisu poligona rekli smo da poligon ima onoliko bridova koliko ima vrhova. Stoga za elementarni dio strukture koja pamti podatke o poligoni možemo uzeti pamćenje jednog vrha i jednog brida. Poligon će se tada sastojati od n ovakvih struktura. Kako vrhove poligona zadajemo u pikselima, za pamćenje koordinata mogu poslužiti cijeli brojevi. Isto tako s obzirom da jednadžbe bridova računamo bez dijeljenja, svi koeficijenti mogu biti cjelobrojni.

Za pamćenje točke u 2D s cjelobrojnim koordinatama koristit ćemo sljedeću strukturu.

```
typedef struct {
    int x;
    int y;
} iTocka2D;
```

Za pamćenje koeficijenata brida u 2D s cjelobrojnim koeficijentima koristit ćemo sljedeću strukturu.

```
typedef struct {
    int a;
    int b;
    int c;
} iBrid2D;
```

Za pamćenje elementarnog dijela poligona koristit ćemo sljedeću strukturu.

```
typedef struct {
    iTocka2D Vrh;
    iBrid2D Brid;
    int lijevi;
} iPolyElem;
```

Elementu lijevi pridružen je tip **int** u nedostatku prikladnijeg tipa. Tip koji bi najbolje odgovarao bio bi boolean koji u C-u ne postoji.

3.2.5 Funkcije za rad s poligonima

Krenimo s najjednostavnijom funkcijom čiji je zadatak nacrtati poligon na zaslonu. Za crtanje bridova koristit ćemo prethodno napisanu funkciju koja taj postupak radi uporabom Bresenhamovog algoritma.

```
void CrtajPoligonKonv( iPolyElem *polel , int n) {
    int i , i0 ;

    i0 = n-1;
    for( i = 0; i < n; i++ ) {
        bresenham_nacrtaj_cjelobrojni(
            polel[ i0 ].Vrh.x, polel[ i0 ].Vrh.y,
            polel[ i ].Vrh.x, polel[ i ].Vrh.y );
        i0 = i;
    }
}
```

Funkcija jednostavno spaja dva po dva vrha u smjeru kako su zadani. Spaja se i završni i početni vrh kako bi se poligon zatvorio. Argumenti funkcije su polje elemenata poligona te broj elemenata polja.

Funkcija koja računa koeficijente poligona prikazana je u nastavku.

```
void RacunajKoefPoligonKonv( iPolyElem *polel , int n) {
    int i , i0 ;

    i0 = n-1;
    for( i = 0; i < n; i++ ) {
        polel[ i0 ].Brid.a = polel[ i0 ].Vrh.y-polel[ i ].Vrh.y;
        polel[ i0 ].Brid.b = -(polel[ i0 ].Vrh.x-polel[ i ].Vrh.x);
        polel[ i0 ].Brid.c = polel[ i0 ].Vrh.x*polel[ i ].Vrh.y
                            - polel[ i0 ].Vrh.y*polel[ i ].Vrh.x;
        polel[ i0 ].lijevi = polel[ i0 ].Vrh.y < polel[ i ].Vrh.y;
        i0 = i;
    }
}
```

Funkcija računa koeficijente bridova radeći pri tome \times -proizvod dva po dva vrha. Jednadžba se dobjije u homogenom prostoru, a točke se u homogene pretvaraju proširivanjem s homogenim parametrom iznosa 1. Dodatno se brid klasificira kao lijevi ako je zadan vrhovima od kojih je prvi ispod drugoga. Ova klasifikacija bit će ispravna samo ako je poligon zadan tako da su mu vrhovi u smjeru kazaljke na satu. Ako su vrhovi zadani suprotno, tada će postavljena zastavica lijevi zapravo označavati da je brid desni.

Funkcija čiji je zadatak obijati konveksni poligon prikazana je u nastavku.

```
void PopuniPoligonKonv( iPolyElem *polel , int n) {
    int i , i0 , y;
    int xmin, xmax, ymin, ymax;
    double L,D,x;

    /* Trazenje minimalnih i maksimalnih koordinata */
    xmin = xmax = polel[ 0 ].Vrh.x;
    ymin = ymax = polel[ 0 ].Vrh.y;
    for( i = 1; i < n; i++ ) {
        if( xmin > polel[ i ].Vrh.x ) xmin = polel[ i ].Vrh.x;
        if( xmax < polel[ i ].Vrh.x ) xmax = polel[ i ].Vrh.x;
        if( ymin > polel[ i ].Vrh.y ) ymin = polel[ i ].Vrh.y;
        if( ymax < polel[ i ].Vrh.y ) ymax = polel[ i ].Vrh.y;
    }

    /* Bojanje poligona: za svaki y izmedu ymin i ymax radi... */
    for( y = ymin; y<=ymax; y++ ) {
```

```

/* Pronadi najveće lijevo i najmanje desno sjecište... */
L = xmin; D = xmax;
i0=n-1;
/* i0 je pocetak brida, i je kraj brida */
for( i = 0; i < n; i0=i++ ) {
    /* ako je brid vodoravan*/
    if( polel[i0].Brid.a==0.) {
        if( polel[i0].Vrh.y == y) {
            if( polel[i0].Vrh.x<polel[i].Vrh.x) {
                L = polel[i0].Vrh.x;
                D = polel[i].Vrh.x;
            } else {
                L = polel[i].Vrh.x;
                D = polel[i0].Vrh.x;
            }
            break;
        }
    } else { /* inace je regularan brid, nadi sjecište */
        x = (-polel[i0].Brid.b*y-polel[i0].Brid.c)/(double) polel[i0].Brid.a;
        if( polel[i0].lijevi) {
            if( L < x ) L = x;
        } else {
            if( D > x ) D = x;
        }
    }
}
bresenham_nacrtaj_cjelobrojni( zaokruzi(L),y,zaokruzi(D),y);
}
}

```

Funkcija je implementacija prethodno opisanog postupka. Funkcija očekuje da su koeficijenti bridova već izračunati. Pogledajmo trenutak kako se računaju sjecišta s bridovima. Postoje dva slučaja opisana u nastavku.

- Brid je vodoravan (i zraka je vodoravna) te postoji opasnost da se brid i zraka sijeku u puno točaka. Brid je vodoravan ako mu je koeficijent a jednak nuli. Tada opet imamo dva slučaja: ili se brid i zraka uopće ne sijeku jer su na različitim y -ima ili se sijeku u svim točkama jer su na istim y -ima. Ako se ne sijeku, tada jednostavno preskačemo daljnju analizu i nastavljamo s obradom sljedećeg brida. Ako se sijeku, tada kao lijevo sjecište uzimamo x -koordinatu onog vrha kod kojeg je ona manja, a kao desno sjecište uzimamo x -koordinatu onog vrha kod kojeg je ona veća. Zatim prestajemo s danjom analizom sjecišta (jer kod konveksnog poligona ona ionako ne postoje) i crtamo liniju određenu pronađenim lijevim i desnim sjecištem.
- Brid nije vodoravan te postoji točno jedno sjecište. Tada pronalazimo to sjecište i ovisno o tome je li brid lijevi ili desni, pamtimo to sjecište kao lijevo ili desno (odnosno pamtimo ga samo ako je lijevo i veće od trenutno zapamćenog lijevog, odnosno ako je desno i manje od trenutno zapamćenog desnog).

Evo za kraj još i funkcije čiji je zadatak utvrditi je li poligon konveksan te koja je orijentacija vrhova poligona.

```

void ProvjeriPoligonKonv(iPolyElem *polel, int n, int *konv, int *orij) {
    int i,i0,r;
    int iznad, ispod, na;

    ispod = iznad = na = 0;
    i0 = n-2;
    for( i = 0; i < n; i++,i0++ ) {
        if( i0>=n) i0=0;
        r = polel[i0].Brid.a*polel[i].Vrh.x + polel[i0].Brid.b*polel[i].Vrh.y + polel[i0].Brid.c;
        if( r == 0 ) na++;
        else if( r > 0 ) iznad++;
        else ispod++;
    }
}

```

```
    }
    *konv = 0; *orij = 0;
    if( ispod == 0 ) {
        *konv = 1;
    } else if( iznad == 0 ) {
        *konv = 1; *orij = 1;
    }
}
```

Funkcija se zasniva na brojanju koliko vrhova leži iznad odgovarajućih bridova, koliko ispod a koliko na njima. Podatak koliko vrhova leži na odgovarajućim bridova redundantan je i trebao bi biti nula. Zaključivanje je sljedeće:

- $\text{ispod} = 0$

Poligon je konveksan jer su tada svi vrhovi iznad odgovarajućih bridova. Orientacija je u smjeru suprotnom od smjera kazaljke na satu.

- $\text{iznad} = 0$

Poligon je konveksan jer su tada svi vrhovi ispod odgovarajućih bridova. Orientacija je u smjeru kazaljke na satu.

- $\text{iznad} != 0 \&& \text{ispod} != 0$

Neki su vrhovi iznad a neki ispod odgovarajućih bridova. Poligon je konkavan i informaciju o orientaciji u varijabli orij treba ignorirati.

Poglavlje 4

Osnovne geometrijske transformacije

Matrični račun temeljni je alat linearne algebre – velikog područja matematike koje je svoju primjenu našlo i u računalnoj grafici. Prilikom izrade kompleksnih scena, često smo u situaciji da neko tijelo želimo pomaknuti u prostoru, promijeniti mu veličinu ili ga zasuvati oko neke točke. Uporabom matričnog računa i homogenih koordinata svaka se od spomenutih operacija može prikazati kao jedna matrica. Izvođenje takve operacije svodi se na matrično množenje – točku koju želimo pomaknuti u prostoru jednostavno pomnožimo matricom translacije. Iz ovog razloga matrični je prikaz ovih operacija izuzetno pogodan u grafičkim aplikacijama, i toliko je čest, da zapravo čini temelj izvođenja grafičkih operacija i jezgru svih popularnih biblioteka za rad s grafikom, uključivo *OpenGL* i *DirectX*.

U ovom poglavlju najprije ćemo pogledati kako se definiraju spomenute operacije u 2D prostoru, nakon čega ćemo rezultate poopćiti na 3D prostor. Pri tome ćemo čitavo vrijeme raditi s homogenim koordinatama točaka. U 2D slučaju to znači da će koordinate imati 3 komponente, i pripadne matrice bit će kvadratne, dimenzija 3×3 . U 3D slučaju koordinate će imati 4 komponente, i pripadne matrice bit će kvadratne, dimenzija 4×4 .

Djelovanje nekog operatora na točku T_h rezultirat će točkom T'_h . Ovo djelovanje možemo iskazati na dva jednakovrijedna načina. Prvi način jest množenjem jednoretčane matrice točke i matrice operatora Ψ , kako je prikazano izrazom (4.1).

$$T'_h = T_h \cdot \Psi. \quad (4.1)$$

Drugi način jest množenjem matrice operatora Ω i jednostupčane matrice promatrane točke, kao što to prikazuje izraz (4.2).

$$T'_h = \Omega \cdot T_h. \quad (4.2)$$

Ovisno o konvenciji s kojom radimo (množimo li točku s matricom operatora ili matricu operatora s točkom), matrice operatora bit će različite, iako se poznavanjem jedne može doći do druge. Naime, vrijedit će:

$$\Omega^T = \Psi.$$

No, u računalnoj grafici je izuzetno rijetka situacija da koristimo samo jednu transformaciju. Puno je češća situacija primjene slijeda transformacija koji generira konačnu poziciju točke. Primjerice, točku A najprije želimo translirati čime dobivamo točku B . Potom, tu točku želimo rotirati, čime dobivamo točku C . Potom tako dobivenu točku želimo još jednom translirati, čime ćemo dobiti točku D , i konačno točku još želimo skalirati čime ćemo dobiti konačnu točku E . Ovaj postupak preslikavanja točke A u točku E skiciran je u nastavku.

$$A \xrightarrow{\text{oper}_1} B \xrightarrow{\text{oper}_2} C \xrightarrow{\text{oper}_3} D \xrightarrow{\text{oper}_4} E$$

Koristimo li konvenciju prikazanu izrazom (4.1), možemo pisati:

$$\begin{aligned}B &= A \cdot \Psi_1 \\C &= B \cdot \Psi_2 = A \cdot \Psi_1 \cdot \Psi_2 \\D &= C \cdot \Psi_3 = A \cdot \Psi_1 \cdot \Psi_2 \cdot \Psi_3 \\E &= D \cdot \Psi_4 = A \cdot \Psi_1 \cdot \Psi_2 \cdot \Psi_3 \cdot \Psi_4\end{aligned}$$

Zbirni operator koji obavlja sve navedene transformacije tada možemo prikazati matricom Ψ koja je naprsto umnožak matrica koje obavljaju željene transformacije:

$$\Psi = \Psi_1 \cdot \Psi_2 \cdot \Psi_3 \cdot \Psi_4.$$

U slučaju da se odlučimo za konvenciju definiranu izrazom (4.2), možemo pisati:

$$\begin{aligned}B &= \Omega_1 \cdot A \\C &= \Omega_2 \cdot B = \Omega_2 \cdot \Omega_1 \cdot A \\D &= \Omega_3 \cdot C = \Omega_3 \cdot \Omega_2 \cdot \Omega_1 \cdot A \\E &= \Omega_4 \cdot D = \Omega_4 \cdot \Omega_3 \cdot \Omega_2 \cdot \Omega_1 \cdot A\end{aligned}$$

Zbirni operator koji obavlja sve navedene transformacije tada možemo prikazati matricom Ω koja je umnožak matrica koje obavljaju željene transformacije, ali reverznim redoslijedom. Dakle,

$$\Omega = \Omega_4 \cdot \Omega_3 \cdot \Omega_2 \cdot \Omega_1.$$

Pogledajmo sada jednostavan primjer. Neka nam je na raspolaganju biblioteka koja za svaku od ovih transformacija nudi prikladnu naredbu koja modificira trenutnu matricu na način da ju pomnoži želenom matricom (s desna) i rezultat postavi kao novu trenutnu matricu. Stavimo li se u ulogu programera, pogledajmo čime će rezultirati program u kojem naredbe zadajemo redoslijedom kojim želimo da se obave opisane transformacije.

Naredba	Djelovanje
<code>identity();</code>	$M = I$
<code>translate(x0,y0);</code>	$M *= M1 \Rightarrow M = M1$
<code>rotate(angle);</code>	$M *= M2 \Rightarrow M = M1 * M2$
<code>translate(x1,y1);</code>	$M *= M3 \Rightarrow M = M1 * M2 * M3$
<code>scale(k1,k2);</code>	$M *= M4 \Rightarrow M = M1 * M2 * M3 * M4$

Ako koristimo konvenciju množenja točke s matricom, rezultat će biti upravo u skladu s očekivanim. Naime, predanu točku najprije će množiti matrica M_1 , potom će rezultat pomnožiti matrica M_2 , taj rezultat pomnožit će matrica M_3 i na kraju će sve pomnožiti matrica M_4 . Koristimo li međutim konvenciju množenja matrice s točkom, rezultat više neće biti korektan. Naime, u tom slučaju prvo se s točkom množi matrica M_4 ; potom matrica M_3 tako dobiveni rezultat, i tako sve do matrice M_1 . Koji je zaključak? Ako se koristi konvencija množenja matrice s točkom, prilikom pisanja programa naredbe koje obavljaju željene transformacije moramo pisati obrnutim redoslijedom od onoga kojim želimo da se te transformacije doista i obave.

U nastavku ovog poglavlja (a i knjige) koristit ćemo upravo konvenciju množenja točke matricom, kako je definirano izrazom (4.1). Međutim, biblioteka *OpenGL* koja nudi programsko sučelje koje radi baš kao u prethodnom jednostavnom primjeru koristi konvenciju definiranu izrazom (4.2) – matricu operatora množi točkom. Kod početnika u *OpenGL*-u ovo može rezultirati problemima i frustracijom, jer početnici često zaborave da naredbe koje obavljaju željene transformacije moraju pisati obrnutim redoslijedom. U tom smislu, program koji obavlja navedene transformacije morao bi biti napisan kako slijedi u nastavku.

Naredba	Djelovanje
<code>identity();</code>	$M = I$
<code>scale(k1,k2);</code>	$M *= M4 \Rightarrow M = M4$
<code>translate(x1,y1);</code>	$M *= M3 \Rightarrow M = M4 * M3$
<code>rotate(angle);</code>	$M *= M2 \Rightarrow M = M4 * M3 * M2$
<code>translate(x0,y0);</code>	$M *= M1 \Rightarrow M = M4 * M3 * M2 * M1$

Osvrnamo se i na svojstva kompozicije transformacija. Budući da se djelovanje operatora opisuje matričnim množenjem, lako se može doći do sljedećeg zaključka: ako na jednu točku djeluje više

transformacija, tada je bitan redoslijed djelovanja transformacija. Dokaz ove tvrdnje je trivijalan. Naime, matrično množenje nije komutativno pa se za različite redoslijede množenja matrica dobivaju različiti rezultati.

Djelovanje više transformacija može se zapisati:

$$T_h' = T_h \cdot \Psi_1 \cdot \Psi_2 \cdots \Psi_n = T_h \cdot \Psi'$$

$$\Psi' = \Psi_1 \cdot \Psi_2 \cdots \Psi_n$$

pri čemu na točku prvo djeluje operator Ψ_1 , pa Ψ_2 itd.

Postoji i iznimka gornjem pravilu: unutar jedne vrste transformacija, redoslijed djelovanja pojedine transformacije nije bitan. Ovo se može i matematički dokazati, no ostanimo na logičkom dokazu: rotiramo li točku za kut α pa za kut β , dobit ćemo isti rezultat kao i da rotiramo točku najprije za kut β pa onda za kut α .

Pojasnimo još nekoliko pojmove koji se vežu uz transformacije, kao što su Euklidske transformacije, afine transformacije te projektivne transformacije.

4.1 Vrste transformacija

Kako su različite vrste transformacija od velikog značaja za računalnu grafiku, u ovom odjeljku upoznat ćemo se s podjelom transformacija. Rigorozan matematički tretman transformacija spada u područje linearne algebre; stoga ćemo se ovdje pokušati zadržati na manje formalnoj razini.

U računalnoj grafici jedan od osnovnih pojmoveva je točka, koju tipično poistovjećujemo s pripadnim radij-vektorom. Pri tome točke promatramo ili u dvodimenzijском radnom prostoru ili u trodimenzijском radnom prostoru, pa govorimo o dvo- ili trokomponentnim vektorima. Nad tim vektorima radit ćemo niz transformacija poput translacije, rotacije, povećanja, sažimanja, smika itd. Djelovanje same transformacije opisivat ćemo pripadnom kvadratnom matricom, kako smo već opisali u ovom poglavlju. Pa krenimo redom.

Promatrajmo n -dimenzijski radni prostor \mathbf{V} . *Linearna transformacija* T je transformacija koja zadovoljava sljedeća svojstva.

1. Čuvanje zbrajanja vektora. Vrijedi: $T(v_1 + v_2) = T(v_1) + T(v_2)$ i to $\forall v_1, v_2 \in \mathbf{V}$.
2. Čuvanje množenja sa skalarom. Vrijedi: $T(\alpha \cdot v) = \alpha \cdot T(v)$ i to $\forall v \in \mathbf{V}, \forall \alpha \in \mathbf{R}$.

Označimo li s \mathbf{M} kvadratnu matricu ranga n koja predstavlja promatrano transformaciju, možemo pisati:

- $(v_1 + v_2) \cdot \mathbf{M} = v_1 \cdot \mathbf{M} + v_2 \cdot \mathbf{M}$, te
- $(\alpha \cdot v) \cdot \mathbf{M} = \alpha \cdot (v \cdot \mathbf{M})$.

Transformacije rotacije, skaliranja i smika pripadaju u linearne transformacije. Translacija se, međutim, ne može zapisati kao linearna transformacija u radnom prostoru. Naime, iz prethodna dva zahtjeva slijedi da se nul-vektor (ishodište) uvijek preslikava u nul-vektor, tj $T(0) = 0$ (provjerite). Stoga transformacija koja vektor pomiče za konstantan pomak ne može biti linearna.

Afne transformacije su transformacije koje čuvaju kolinearnost i omjere udaljenosti, i čine općenitiju kategoriju od linearnih transformacija. Ovo prvo (čuvanje kolinearnosti) ima za posljedicu da će tri različite točke A , B i C koje su u originalnom prostoru kolinearne i nakon transformacije u točke A' , B' i C' ostati kolinearne (ležat će na istom pravcu). Drugo svojstvo nam govori i da će omjeri udaljenosti tih točaka ostati nepromijenjeni, tj: $d(A, B)/d(A, C) = d(A', B')/d(A', C')$, gdje je $d(X, Y)$ funkcija koja vraća udaljenost točaka X i Y . Afne transformacije čuvaju i paralelnost. Linije koje su u originalnom prostoru paralelne ostat će paralelne i nakon transformacije. Međutim, afne transformacije ne čuvaju kuteve niti stvarne duljine. Naime, afinom transformacijom bilo koji trokut

moguće je preslikati u bilo koji drugi trokut – posljedica je da se kutevi u trokutu kao niti duljine stranica ne moraju očuvati.

Svaka se afina transformacija može zapisati u obliku:

$$T(v) = L(v) + p$$

gdje je L linearna transformacija a p vektor pomaka. Kompozicija afinskih transformacija opet se može prikazati kao jedna afina transformacija. Afine transformacije obuhvaćaju sve linearne transformacije kao i neke druge, primjerice pomak.

Euklidske transformacije su posebna vrsta afinskih transformacija. Naime, Euklidska transformacija je afina transformacija oblika $E(v) = L(v) + p$, gdje je L ortogonalna linearna transformacija. Euklidska transformacija kao posebna vrsta afine transformacije ima sva svojstva afine transformacije. Dodatno, kako je $E(u - v) = L(u - v)$ i kako je L ortogonalna linearna transformacija, E čuva duljine segmenata i kuteve između dva linijska segmenta. Drugim riječima, pravokutni trokut duljina stranica 9, 12 i 15 i nakon Euklidske transformacije ostat će pravokutan, s upravo tim duljinama stranice. Kako bi ova svojstva bila zadovoljena, na opći oblik afine transformacije već smo postavili jedno ograničene – L mora biti ortogonalna linearna transformacija. Postavljanjem dalnjih ograničenja na L i p dolazi se do tri moguće vrste Euklidske transformacije.

1. Translacija – dobije se ako je $L = I$ (tj. kada je matrica linearne translacije jedinična matrica). Tada ostaje $T(v) = v + p$, što odgovara pomaku točke fiksni iznos.
2. Rotacija – dobije se ako je $p = 0$ i ako je dodatno $\det(L) = 1$. Svaka ortonormalna matrica čija je determinanta jednaka 1 predstavlja transformaciju rotacije.
3. Refleksija – dobije se ako je $p = 0$ a $\det(L) = -1$. Primjerice, u 2D prostoru matrica

$$\begin{vmatrix} \cos(\phi) & \sin(\phi) \\ \sin(\phi) & -\cos(\phi) \end{vmatrix}$$

zrcali svaki vektor oko pravca određenog jednadžbom $y = x \cdot \operatorname{tg}\left(\frac{\phi}{2}\right)$.

Kako smo se ovdje oslonili na pojam ortogonalne (tj. ortonormalna) transformacije, recimo da je to transformacija čija je matrica ortonormalna, a to znači da za takvu matricu A vrijedi:

- $A \cdot A^T = I$,
- $A^{-1} = A^T$,
- ortonormalne matrice čuvaju skalarni produkt: $v \cdot w = Av \cdot Aw$ te
- determinanta od A je 1 ili -1.

Konačno, s obzirom da je izvorni zapis afine transformacije:

$$T(v) = L \cdot v + p$$

nepraktičan za direktnu primjenu na računalu, sve transformacije najčešće se provode u *homogenom* prostoru. U tom slučaju čitava se afina transformacija može prikazati kao jedna kvadratna matrica. Uz gore prikidanu konvenciju množenja matrice i točke gdje je matrica L kvadratna ranga n i gdje je točka v zapravo $n+1$ -komponentni vektor (jednostupčana matrica) koji u homogenom prostoru predstavlja točku radnog n -dimenzijskog prostora, odgovarajuća matrica afine transformacije je oblika:

$$\begin{vmatrix} L & p \\ 0 \cdots 0 & 1 \end{vmatrix}.$$

U slučaju da se koristi konvencija množenja točke s matricom, $T(v) = v \cdot L + p$, tj. ako je točka v zapravo $n+1$ -komponentni vektor (jednoretčana matrica) koji u homogenom prostoru predstavlja točku radnog n -dimenzijskog prostora, odgovarajuća matrica afine transformacije je oblika:

$$\begin{vmatrix} L & 0 \\ p & 1 \end{vmatrix}.$$

konačno, osvrnimo se i na još općenitiji razred transformacija – *projektivne transformacije*. Projektivne transformacije opisujemo matricama u homogenom prostoru, pa je tako opći oblik projektivne matrice koja radi projekciju u 3D radnom prostoru opisan kvadratnom matricom ranga 4:

$$\begin{vmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \\ p_{41} & p_{42} & p_{43} & p_{44} \end{vmatrix}.$$

Ovo nije afna transformacija jer nema ograničenja koje smo postavili nad afne matrice. U nastavku ovog poglavlja obraditi ćemo najčešće korištene afne transformacije, a u sljedećem poglavlju govorit ćemo o projekcijama.

4.2 2D transformacije

4.2.1 Uvod

Postoji nekoliko elementarnih transformacija koje djeluju nad točkom. Kada se ta točka nalazi u 2D prostoru, govorimo o 2D-transformacijama. U nastavku ćemo obraditi sljedeće elementarne transformacije:

- translaciju,
- rotaciju,
- skaliranje te
- smik.

Koristit ćemo točke u homogenom prostoru, a transformacije ćemo prikazivati u matričnom obliku. Svaka elementarna transformacija bit će predstavljena kao jedan operator, kome će odgovarati kvadratna matrica. U 2D prostoru ovi operatori imaju kvadratne matrice reda 3, budući da se točke zapisuju kao jednoretčane matrice s tri stupca $[x, y, h]$.

4.2.2 Translacija

Translacija je transformacija koja svakoj komponenti točke u radnom prostoru dodaje određeni pomak. Primjer translacije prikazan je na slici 4.1. Točka T translatira se u točku T' .

Postupak translacije može se opisati sljedećim jednadžbama u radnom prostoru:

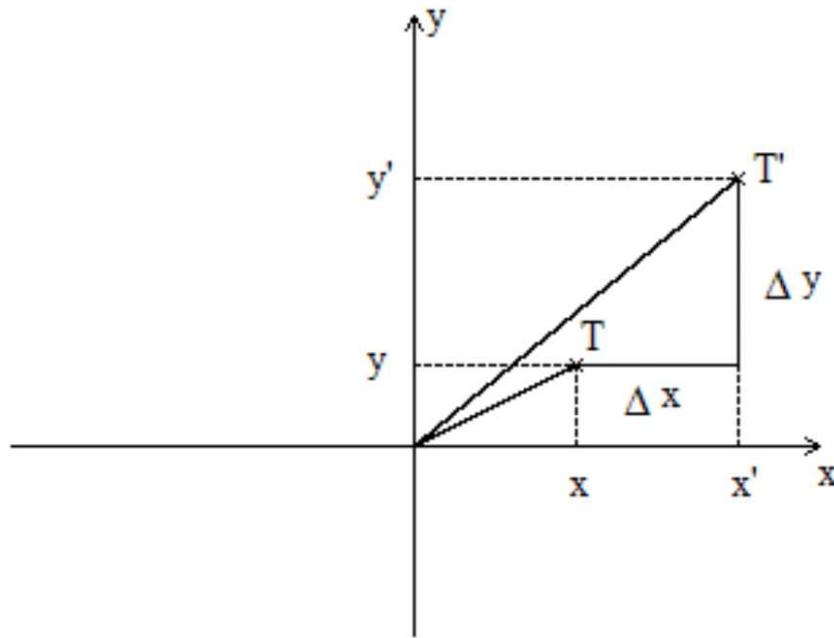
$$\begin{aligned} T'_x &= T_x + \Delta_x \\ T'_y &= T_y + \Delta_y \end{aligned}$$

gdje su T_x i T_y komponente točke T a T'_1 i T'_2 komponente točke T' . Iskoristimo li vezu između radnih i homogenih koordinata:

$$T_x = \frac{T_{h,x}}{T_{h,h}} \quad T_y = \frac{T_{h,y}}{T_{h,h}}$$

dobiva se:

$$\begin{aligned} T'_{h,x} &= T_{h,x} + \Delta_x \cdot T_{h,h} \\ T'_{h,y} &= T_{h,y} + \Delta_y \cdot T_{h,h} \end{aligned}$$



Slika 4.1: Translacija točke

pri čemu su $T_{h,x}$ i $T_{h,y}$ komponente homogenog zapisa točke T (odnosno tada T_h), dok je $T_{h,h}$ homogeni parametar.

Prethodne relacije pokazuju da se točka T'_h može dobiti matričnim množenjem točke T_h operatorom translacije:

$$\begin{bmatrix} T'_{h,x} & T'_{h,y} & T'_{h,h} \end{bmatrix} = \begin{bmatrix} T_{h,x} & T_{h,y} & T_{h,h} \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \Delta_x & \Delta_y & 1 \end{bmatrix}$$

pa se operator translacije matrično zapisuje kao:

$$\Psi_{tr} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \Delta_x & \Delta_y & 1 \end{bmatrix}.$$

Inverzni operator ovom operatoru je operator translacije za negativan pomak:

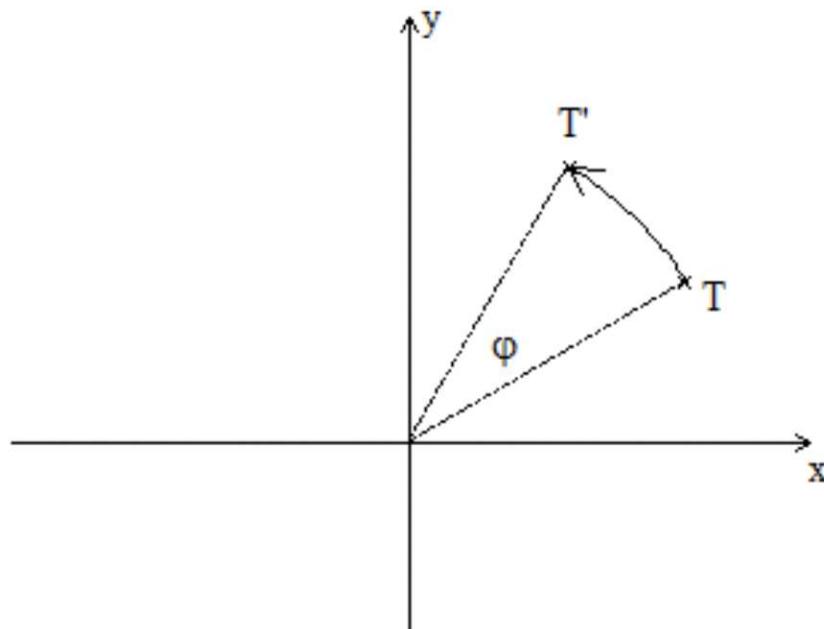
$$\Psi_{tr}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\Delta_x & -\Delta_y & 1 \end{bmatrix},$$

jer vrijedi:

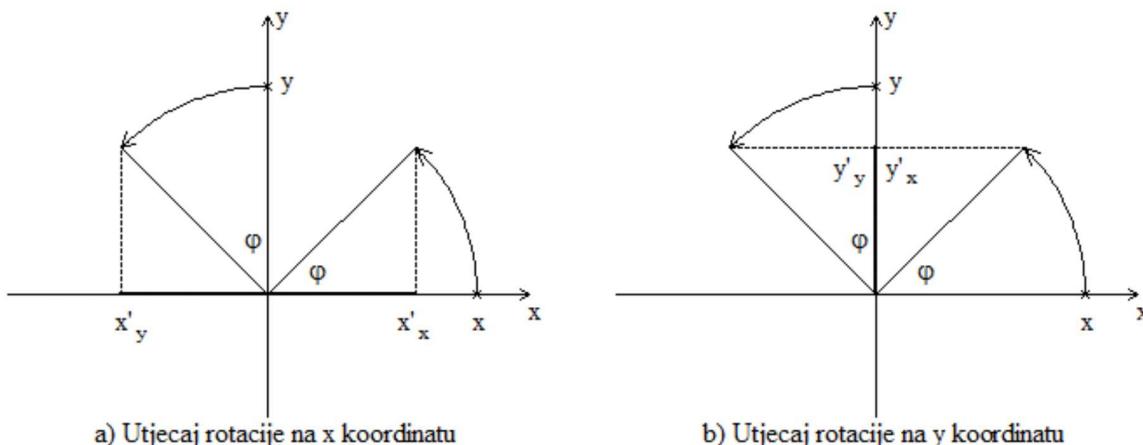
$$\Psi_{tr} \cdot \Psi_{tr}^{-1} = \Psi_{tr}^{-1} \cdot \Psi_{tr} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

4.2.3 Rotacija

Rotacija je transformacija koja točku rotira oko ishodišta za zadani kut ϕ . Smjer rotacije može biti podudaran sa smjerom kazaljke na satu, ili suprotan od smjera kazaljke na satu. Kako je matematički "pozitivan" smjer rotacije definiran kao smjer suprotan od smjera kazaljke na satu, tako će u nastavku biti izведен operator koji rotira točku u smjeru suprotnom od kazaljke na satu. Ako se želi rotirati u smjeru kazaljke na satu, dovoljno je umjesto kuta ϕ rotirati za smjer $-\phi$. Primjer rotacije prikazan je na slici 4.2.



Slika 4.2: Rotacija točke



Slika 4.3: Rotacija točke: utjecaj na pojedine komponente

Da bismo izveli maticu operatora rotacije, potrebno se je prisjetiti linearne algebre, gdje smo naučili da se djelovanje operatora može zapisati kao zbroj djelovanja operatora na svaku komponentu baze prostora pojedinačno (naravno, ako operator zadovoljava određene uvjete koji su ovdje zadovoljeni). Budući da izvodimo rotaciju u ravnini, imamo dvije komponente baze: komponentu u smjeru osi x , i komponentu u smjeru osi y . Djelovanje operatora na te dvije komponente prikazano je na slici 4.3.

Pogledajmo najprije sliku 4.3.a. Ako točku koja leži na osi x zarotiramo za kut ϕ , dobiti ćemo točku čija je x' koordinata jednaka $x'_x = x \cdot \cos\phi$. Ako točku koja leži na osi y zarotiramo za kut ϕ , dobiti ćemo točku čija je x'_y koordinata jednaka: $x'_y = -y \cdot \sin\phi$. Tada djelovanje operatora na točku daje x' komponentu jednaku sumi ova dva djelovanja:

$$x' = x \cdot \cos\phi - y \cdot \sin\phi.$$

Da bismo utvrdili kako se tvori y' komponenta točke, pogledajmo djelovanje operatora opet na točke $(x, 0)$ i $(0, y)$. Međutim, sada moramo pratiti što se događa s y -projekcijama nastalih točaka, kao što je prikazano na slici 4.3.b. Djelovanjem na $(x, 0)$ dobiva se $y'_x = x \cdot \sin\phi$ a djelovanjem na $(0, y)$ dobiva se $y'_y = y \cdot \cos\phi$. Ukupno djelovanje daje:

$$y' = x \cdot \sin\phi + y \cdot \cos\phi.$$

Zapisano u matričnom obliku u homogenim koordinatama dobiva se:

$$\begin{bmatrix} T'_{h,x} & T'_{h,y} & T'_{h,h} \end{bmatrix} = \begin{bmatrix} T_{h,x} & T_{h,y} & T_{h,h} \end{bmatrix} \cdot \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

pa se operator rotacije zapisuje:

$$\Psi_{rot} = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Ako želimo operator koji rotira u smjeru kazaljke na satu, dovoljno je umjesto kuta ϕ rotirati za kut $-\phi$, pa se uvrštavanjem u matricu operatora i uzimanjem u obzir parnosti funkcije \cos i neparnosti funkcije \sin dobiva operator:

$$\Psi'_{rot} = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Korištenjem ovog operatora uz pozitivne vrijednosti kuta ϕ dobiti ćemo rotaciju u smjeru kazaljke na satu.

Odmah se može uočiti da se operatori Ψ_{rot} i Ψ'_{rot} međusobno poništavaju u djelovanju te su inverzi jedan drugome. Naime, ako pogledamo operator koji opisuje djelovanje oba operatora, dobiva se:

$$\Psi_{rot} \cdot \Psi'_{rot} = \Psi'_{rot} \cdot \Psi_{rot} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

iz čega je vidljivo da slijedno djelovanje jednog pa drugog operatora vraća točku u prvobitni položaj.

4.2.4 Skaliranje

Skaliranje je transformacija koja "skalira" (rasteže ili steže) svaku komponentu točke. Pri tome je skaliranje svake komponente određeno faktorom skaliranja, pri čemu faktori ne moraju biti isti za sve komponente. Ako je to slučaj, tada govorimo o neproporcionalnom skaliranju. Ako su faktori skaliranja jednaki za sve komponente, tada govorimo o proporcionalnom skaliranju. Primjer skaliranja prikazan je na slici 4.4.

Djelovanje operatora po komponentama prikazano je na slici 4.5.

Vrijedi:

$$x' = k_1 \cdot x$$

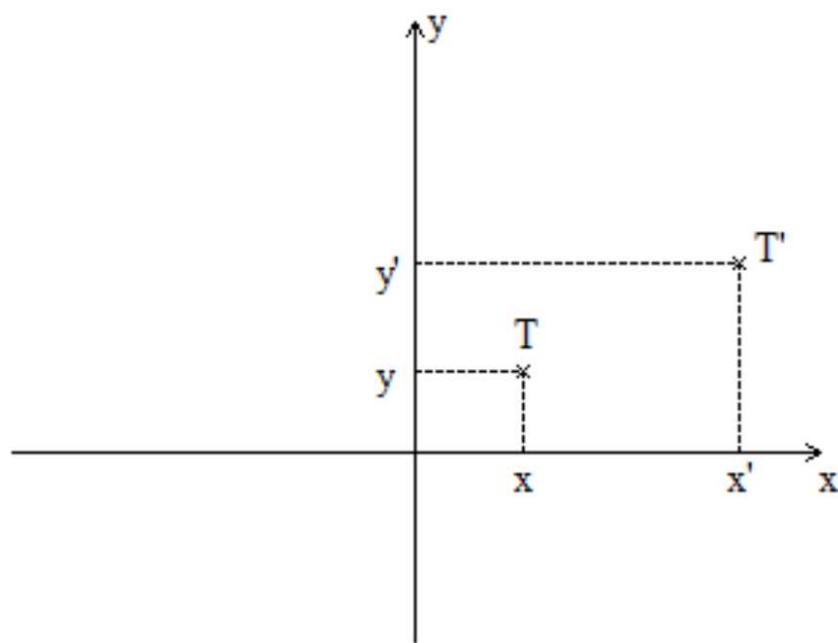
$$y' = k_2 \cdot y$$

odnosno prelaskom na homogene koordinate:

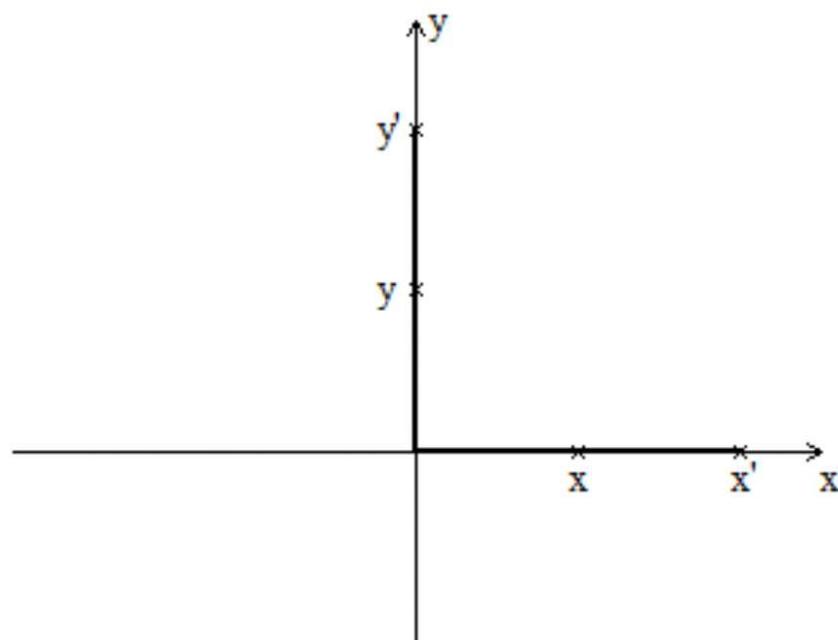
$$\begin{bmatrix} T'_{h,x} & T'_{h,y} & T'_{h,h} \end{bmatrix} = \begin{bmatrix} T_{h,x} & T_{h,y} & T_{h,h} \end{bmatrix} \cdot \begin{bmatrix} k_1 & 0 & 0 \\ 0 & k_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

pa se operator skaliranja zapisuje:

$$\Psi_{skal} = \begin{bmatrix} k_1 & 0 & 0 \\ 0 & k_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$



Slika 4.4: Skaliranje točke



Slika 4.5: Skaliranje točke: djelovanje po komponentama

Ako želimo proporcionalno skaliranje, tada će k_1 biti jednak k_2 što možemo nazvati k , pa operator poprima oblik:

$$\Psi_{pskal} = \begin{bmatrix} k & 0 & 0 \\ 0 & k & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

što se još može zapisati i u obliku:

$$\Psi_{pskal} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \frac{1}{k} \end{bmatrix}.$$

Inverzni operator operatoru skaliranja je skaliranje recipročnim koeficijentima, što za neproporcionalno skaliranje daje:

$$\Psi_{skal}^{-1} = \begin{bmatrix} \frac{1}{k_1} & 0 & 0 \\ 0 & \frac{1}{k_2} & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

a za proporcionalno skaliranje:

$$\Psi_{pskal}^{-1} = \begin{bmatrix} \frac{1}{k} & 0 & 0 \\ 0 & \frac{1}{k} & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

ili

$$\Psi_{pskal}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & k \end{bmatrix},$$

ovisno o matrici kojom je vršeno proporcionalno skaliranje.

4.2.5 Smik

Smik je "uzdužna" deformacija čije se djelovanje najbolje vidi sa slike 4.6. Sastoji se od deformacije uzduž osi x i deformacije uzduž osi y te se opisuje kutovima α i β .

Djelovanje operatora na pojedine osi prikazano je na slici 4.7.

Za pojedine komponente proizlazi:

$$x' = x'_x + x'_y = x + y \cdot \operatorname{tg}\beta$$

$$y' = y'_y + y'_x = y + x \cdot \operatorname{tg}\alpha$$

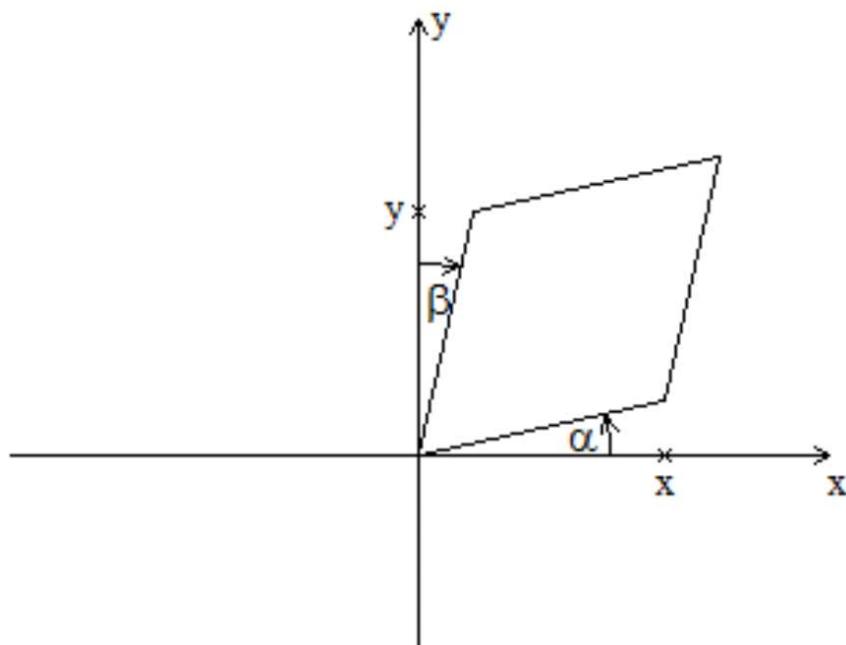
pa se prelaskom na homogene koordinate dobiva:

$$\begin{bmatrix} T'_{h,x} & T'_{h,y} & T'_{h,h} \end{bmatrix} = \begin{bmatrix} T_{h,x} & T_{h,y} & T_{h,h} \end{bmatrix} \cdot \begin{bmatrix} 1 & \operatorname{tg}\alpha & 0 \\ \operatorname{tg}\beta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

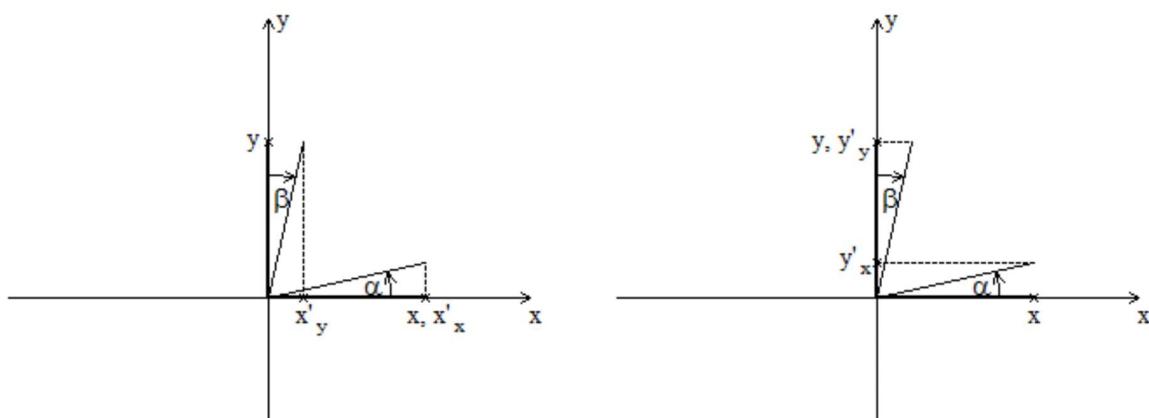
pa se operator smika zapisuje:

$$\Psi_{smik} = \begin{bmatrix} 1 & \operatorname{tg}\alpha & 0 \\ \operatorname{tg}\beta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

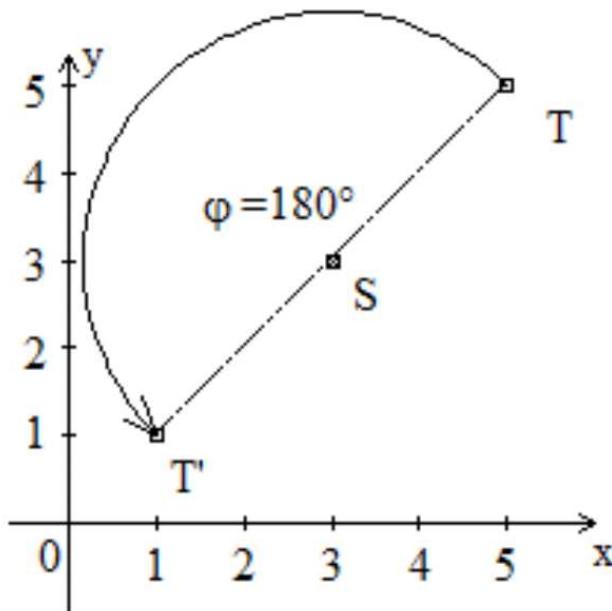
Potražimo li inverz ovog operatora iz uvjeta da umnožak operatora smika i inverznog operatora daje jediničnu matricu dobiva se:



Slika 4.6: Smik



Slika 4.7: Smik: djelovanje po komponentama



Slika 4.8: Rotacija oko zadane točke

$$\Psi_{smik} = \begin{bmatrix} \frac{1}{1-tg\alpha \cdot tg\beta} & \frac{-tg\alpha}{1-tg\alpha \cdot tg\beta} & 0 \\ \frac{-tg\beta}{1-tg\alpha \cdot tg\beta} & \frac{1}{1-tg\alpha \cdot tg\beta} & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

što nije definirano ukoliko je umnožak tangensa jednak 1.

4.2.6 Primjer

U nastavku ćemo pokazati jednostavan primjer na kojem se mogu uočiti interesantni detalji. Potrebno je rotirati točku $T(5,5)$ u smjeru suprotnom od smjera kazaljke na satu za kut ϕ oko točke $S(3,3)$. Slika 4.8. pokazuje što želimo učiniti. Na slici je prikazan primjer rotacije za 180° .

Pokušamo li direktno primijeniti operator rotacije, rezultat neće dati očekivani rezultat (zapravo, ovisi što ste očekivali). Naime, operator rotacije izведен je tako da rotira točku oko ishodišta. Želimo li rotirati točku oko nekog drugog središta, morat ćemo primijeniti kompoziciju operatora translacije, rotacije i ponovno translacije. Evo koraka koje treba napraviti.

1. Točku T potrebno je translatirati istom translacijom koja bi točku S (željeno središte rotacije) dovela u ishodište koordinatnog sustava. To će biti translacija za $\Delta_x = -S_x$ i $\Delta_y = -S_y$, uz pretpostavku da je točka S dana komponentama $(S_x, S_y, 1)$. Nazovimo ovu transformaciju Ψ_1 :

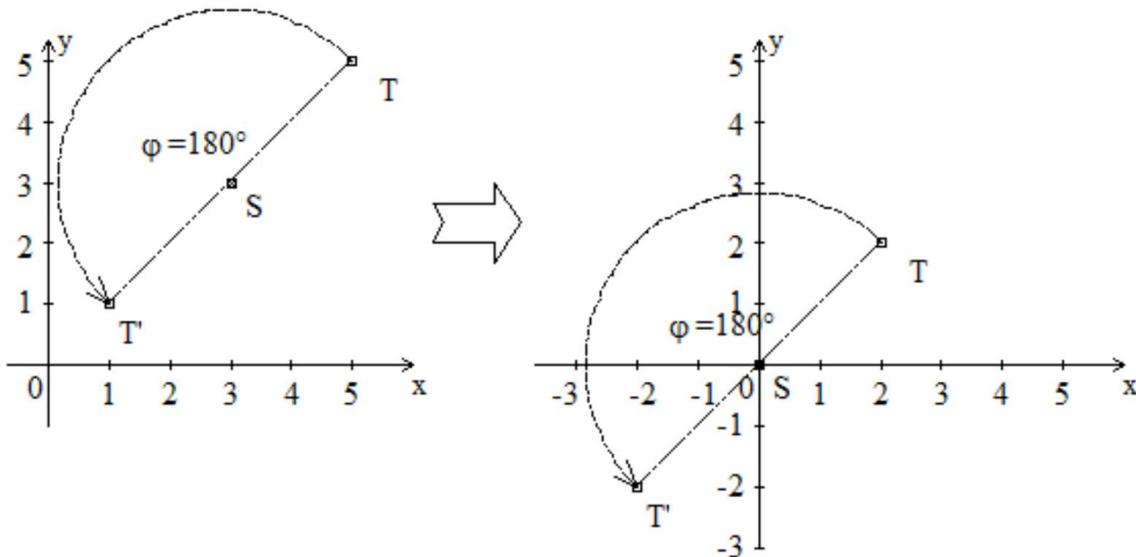
$$\Psi_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -S_x & -S_y & 1 \end{bmatrix}.$$

2. Na ovako dobivenu točku možemo primijeniti operator rotacije jer se sada središte rotacije nalazi u ishodištu. Tada je Ψ_2 :

$$\Psi_2 = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

3. Konačno, nakon što smo točku zarotirali, moramo je natrag translatirati inverznom translacijom od one iz prvog koraka:

$$\Psi_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ S_x & S_y & 1 \end{bmatrix}.$$



Slika 4.9: Rotacija oko zadane točke: nakon translacije

Što se je dogodilo nakon prvog koraka, opisuje slika 4.9.

U svakom koraku dobili smo po jedan operator. Zbirni operator Ψ koji će napraviti zadanu operaciju dobije se kao umnožak sva tri operatora i to upravo redoslijedom kojim su djelovali:

$$\Psi = \Psi_1 \cdot \Psi_2 \cdot \Psi_3$$

Da je ovo ispravno, možemo se lako uvjeriti pokusom. Ako rotiramo točku T za 180° oko točke S , trebali bismo dobiti točku $T'(1, 1)$. Ako izračunamo vrijednost operatora uz zadani kut dobit ćemo:

$$\Psi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & -3 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 3 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 6 & 6 & 1 \end{bmatrix}$$

Primijenimo li operator na točku $T(5, 5)$ dobivamo:

$$\begin{aligned} \begin{bmatrix} T'_{h,x} & T'_{h,y} & T'_{h,h} \end{bmatrix} &= \begin{bmatrix} T_{h,x} & T_{h,y} & T_{h,h} \end{bmatrix} \cdot \Psi \\ &= \begin{bmatrix} 5 & 5 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 6 & 6 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \end{aligned}$$

Dakle, dobili smo točku koju smo i očekivali.

Sličan postupak koji je naveden u prethodna tri koraka često se provodi jer izvedene elementarne transformacije djeluju obzirom na ishodište, pa treba dobro paziti što se želi postići, a što transformacije zapravo daju.

4.3 3D transformacije

4.3.1 Uvod

3D transformacije su transformacije nad točkom u 3D prostoru. U nastavku ćemo obraditi iste transformacije koje smo obradili u 2D prostoru. Jedina novost koju donosi 3D prostor su tri operatora rotacije umjesto jednog u 2D prostoru. Tako ćemo obraditi:

- translaciju,

- rotaciju oko osi x ,
- rotaciju oko osi y ,
- rotaciju oko osi z ,
- skaliranje te
- smik.

Podsjetimo se još jednom: redoslijed djelovanja transformacija bitan je za krajnji rezultat, osim ako se ne radi o uzastopnom djelovanju iste transformacije kada je redoslijed nebitan. Naravno, rotacija oko osi x i rotacija oko osi y nisu iste transformacije.

U nastavku će biti dani izrazi za ove transformacije, budući da se oni izvode identično kao i izrazi za 2D transformacije koje smo već obradili u prethodnom podpoglavlju.

4.3.2 Translacija

Translacija pomiciće točku tako da svakoj koordinati točke doda određeni pomak. Vrijedi:

$$\begin{aligned} T'_x &= T_x + \Delta_x \\ T'_y &= T_y + \Delta_y \\ T'_z &= T_z + \Delta_z \end{aligned}$$

ili nakon prelaska u homogene koordinate:

$$\begin{bmatrix} T'_{h,x} & T'_{h,y} & T'_{h,z} & T'_{h,h} \end{bmatrix} = \begin{bmatrix} T_{h,x} & T_{h,y} & T_{h,z} & T_{h,h} \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \Delta_x & \Delta_y & \Delta_z & 1 \end{bmatrix}$$

što daje operator translacije u 3D:

$$\Psi_{tr} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \Delta_x & \Delta_y & \Delta_z & 1 \end{bmatrix}.$$

Inverz je translacija za negativne pomake:

$$\Psi_{tr}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\Delta_x & -\Delta_y & -\Delta_z & 1 \end{bmatrix}.$$

4.3.3 Rotacija

Razlikujemo tri rotacije: rotaciju oko osi x , rotaciju oko osi y te rotaciju oko osi z .

Rotacija oko osi x

Rotacija oko osi x rotira točku u y - z ravnini, pri čemu x -koordinata točke ostaje nepromijenjena.

Operator rotacije oko osi x u smjeru suprotnom od smjera kazaljke na satu glasi:

$$\Psi_{rotx} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Njemu inverzni operator je operator rotacije oko osi x u smjeru kazaljke na satu:

$$\Psi'_{rotx} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Rotacija oko osi y

Rotacija oko osi y rotira točku u x - z ravnini, pri čemu y -koordinata točke ostaje nepromijenjena.

Operator rotacije oko osi y u smjeru suprotnom od smjera kazaljke na satu glasi:

$$\Psi_{roty} = \begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Njemu inverzni operator je operator rotacije oko osi y u smjeru kazaljke na satu:

$$\Psi_{roty}^{-1} = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Rotacija oko osi z

Rotacija oko osi z rotira točku u x - y ravnini, pri čemu z -koordinata točke ostaje nepromijenjena.

Operator rotacije oko osi z u smjeru suprotnom od smjera kazaljke na satu glasi:

$$\Psi_{rotz} = \begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Njemu inverzni operator je operator rotacije oko osi z u smjeru kazaljke na satu:

$$\Psi_{rotz}^{-1} = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Usporedimo li ovu matricu s matricom dobivenom za 2D rotaciju, vidimo da je matrica građena identično. Ovo je posljedica činjenice da smo rotaciju u 2D upravo izveli kao rotaciju u x - y ravnini gdje smo rekli da točku rotiramo oko ishodišta; ovo bi se slobodno moglo proširiti pa reći da rotaciju izvodimo oko osi okomite na x - y ravninu koja prolazi ishodištem $\Rightarrow z$ -osi!

4.3.4 Skaliranje

Skaliranje svaku koordinatu "skalira" (rasteže ili steže) množeći je sa odgovarajućim koeficijentom. Kako imamo tri koordinate radnog prostora, imati ćemo i tri koeficijenta. Ako su ti koeficijenti međusobno različiti, govorimo o neproporcionalnom skaliranju; inače govorimo o proporcionalnom skaliranju.

$$\Psi_{sk} = \begin{bmatrix} k_1 & 0 & 0 & 0 \\ 0 & k_2 & 0 & 0 \\ 0 & 0 & k_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Inverz je translacija za negativne pomake:

$$\Psi_{sk}^{-1} = \begin{bmatrix} \frac{1}{k_1} & 0 & 0 & 0 \\ 0 & \frac{1}{k_2} & 0 & 0 \\ 0 & 0 & \frac{1}{k_3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Proporcionalno skaliranje dobije se za $k_1 = k_2 = k_3 = k$, no tada se operator proporcionalnog skaliranja može zapisati na još jedan način (osim već prikazanog općeg, pa uvrštavanjem k za sve koeficijente):

$$\Psi_{psk} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{k} \end{bmatrix}.$$

U tom slučaju inverz ovom operatoru je operator:

$$\Psi_{psk}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & k \end{bmatrix}.$$

4.3.5 Smik

Smik je uzdužna deformacija koja deformira položaj točke i opisuje se kutom deformacije prema svakoj koordinatnoj osi. U 3D prostoru imamo 3 koordinatne osi i tri kuta: α , β i γ .

Operator smika u 3D glasi:

$$\Psi_{sm} = \begin{bmatrix} 1 & \operatorname{tg}\alpha & \operatorname{tg}\alpha & 0 \\ \operatorname{tg}\beta & 1 & \operatorname{tg}\beta & 0 \\ \operatorname{tg}\gamma & \operatorname{tg}\gamma & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Izvođenje inverznog operatora i pronalaženje uvjeta njegove egzistencije ostavlja se čitateljima za zabavu.

4.3.6 Primjer

U nastavku knjige trebat će nam rotacija oko zadano središta, pa ćemo se ovdje još jednom podsjetiti na primjer koji smo već pokazali kod 2D transformacija. Potrebno je točku T rotirati oko točke S . Naravno, kako u 3D prostoru imamo 3 vrste rotacija, potrebno je zadati i koju rotaciju želimo primijeniti. Jednom kada imamo sve podatke, postupak je sljedeći:

1. translatirati točku S u ishodište koordinatnog sustava,
2. izvršiti rotaciju te
3. primijeniti inverznu translaciju od translacije iz prvog koraka.

4.4 OpenGL i transformacije

Prisjetimo se zaključka s početka ovog poglavlja: djelovanje operatora na točku možemo iskazati ili množenjem točke matricom operatora, ili množenjem matrice operatora točkom. Kroz ovu knjigu mi ćemo se držati prvog načina, dok *OpenGL* koristi drugi način. Prisjetimo se još i veze između matrica istog operatora uz te dvije konvencije. Neka uz prvu konvenciju operatoru odgovara matrica Ψ , a uz drugu konvenciju matrica Ω . Vrijedi:

$$\Omega^T = \Psi.$$

Pogledajmo sada koje nam naredbe stoje na raspolaganju u *OpenGL*-u, i kako izgledaju pripadne matrice.

4.4.1 Translacija

Za potrebe translacije *OpenGL* nam nudi naredbu: `glTranslate*(dx,dy,dz);` koja trenutnu matricu množi matricom Ω_{tr} :

$$\Omega_{tr} = \begin{bmatrix} 1 & 0 & 0 & \Delta_x \\ 0 & 1 & 0 & \Delta_y \\ 0 & 0 & 1 & \Delta_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Uvjericite se da je to upravo transponirana matrica od matrice Ψ_{tr} koju smo prethodno izveli. Inverz ove matrice je:

$$\Omega_{tr}^{-1} = \begin{bmatrix} 1 & 0 & 0 & -\Delta_x \\ 0 & 1 & 0 & -\Delta_y \\ 0 & 0 & 1 & -\Delta_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Zjvezdica u imenu zapravo nam govori da postoji porodica takvih funkcija koje primaju argumente različitog tipa. Primjerice, ako kao vrijednosti dx , dy i dz predajemo **float**-ove, pozvat ćemo naredbu `glTranslatef(1.0f,2.4f,-0.7f);`.

4.4.2 Skaliranje

Za potrebe skaliranja *OpenGL* nam nudi naredbu `glScale*(sx,sy,sz);` koja trenutnu matricu množi matricom Ω_{sk} :

$$\Omega_{sk} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Uvjericite se da je to upravo transponirana matrica od matrice Ψ_{sk} koju smo prethodno izveli. Inverz ove matrice je:

$$\Omega_{sk}^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 & 0 \\ 0 & 0 & \frac{1}{s_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

4.4.3 Rotacija

Za potrebe rotacije *OpenGL* nam nudi naredbu: `glRotate*(a,x,y,z);`. Ova naredba trenutnu matricu množi matricom koja obavlja rotaciju za kut α (prvi argument naredbe) oko proizvoljne osi određene vektorom (x, y, z) . Pri tome se i dalje rotacija obavlja oko ishodišta. Kada vektor (x, y, z) predstavlja koordinatne osi, matrice će odgovarati već prethodno izvedenima.

Tako naredba `glRotate(a,1,0,0);` računa matricu rotacije oko osi x , pa će trenutnu matricu pomnožiti matricom Ω_{rotx} :

$$\Omega_{rotx} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Naredba `glRotate*(a,0,1,0);` računa matricu rotacije oko osi y , pa će trenutnu matricu pomnožiti matricom Ω_{rot_y} :

$$\Omega_{rot_y} = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Naredba `glRotate*(a,0,0,1);` računa matricu rotacije oko osi z , pa će trenutnu matricu pomnožiti matricom Ω_{rot_z} :

$$\Omega_{rot_z} = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Konačno, za proizvoljnu os određenu vektorom (x, y, z) , naredba `glRotate*(a,x,y,z);` računa matricu rotacije oko osi određene tim vektorom, pa će trenutnu matricu pomnožiti matricom Ω_{rot} :

$$\begin{bmatrix} u_x^2 + (1 - u_x^2)\cos\alpha & u_xu_y(1 - \cos\alpha) - u_z\sin\alpha & u_xu_z(1 - \cos\alpha) + u_y\sin\alpha & 0 \\ u_xu_y(1 - \cos\alpha) + u_z\sin\alpha & u_y^2 + (1 - u_y^2)\cos\alpha & u_yu_z(1 - \cos\alpha) - u_x\sin\alpha & 0 \\ u_xu_z(1 - \cos\alpha) - u_y\sin\alpha & u_yu_z(1 - \cos\alpha) + u_x\sin\alpha & u_z^2 + (1 - u_z^2)\cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Pri tome su u_x , u_y i u_z komponente vektora \vec{u} koji je dobiven normiranjem zadano vektora $\vec{v} = (x, y, z)$, tj:

$$\vec{u} = \frac{\vec{v}}{\|\vec{v}\|}.$$

4.5 Transformacije normala

Prilikom rada s objektima u prostoru scene često ćemo trebati normale (primjerice, normale poligona). Štoviše, kako bismo uštedjeli na vremenu, normale ćemo računati prilikom učitavanja objekta i kasnije ih samo koristiti. Da bismo to međutim mogli, pogledajmo kakav utjecaj transformacije imaju na normale. Pogledajmo to na jednostavnom primjeru u 2D prostoru. Neka je zadan segment pravca određen točkama $T_1 = (0, 0)$ i $T_2 = (5, 1)$. Vektor pravca na kojem leži taj segment je $\vec{v} = (5, 1) - (0, 0) = (5, 1)$. Normala \vec{n} koja pripada tom segmentu tada je vektor $\vec{n} = (-1, 5)$ (uvjerite se da je $\vec{v} \cdot \vec{n} = 0$). Normala je dobivena iz zahtjeva da skalarni produkt bude nula ($v_x \cdot n_x + v_y \cdot n_y = 0$) te da suma komponenti vektora \vec{n} bude 1 ($n_x + n_y = 1$).

Neka je sada zadana afina transformacija koja obavlja skaliranje po osi y s faktorom 2. Pripadna matrica \mathbf{M} je:

$$\mathbf{M} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Ova transformacija točke T_1 i T_2 preslikava u $T_3 = T_1\mathbf{M} = (0, 0)$ te $T_4 = T_2\mathbf{M} = (5, 2)$, pri čemu se segment određen točkama T_1 i T_2 preslikava u segment određen točkama T_3 i T_4 . Odgovarajući vektor pravca sada je $\vec{v}' = (5, 2) - (0, 0) = (5, 2)$.

Provjerimo je li vektor \vec{n} i dalje korektna normala? Ako je, mora vrijediti da je skalarni produkt jednak nuli, no to više nije slučaj:

$$v'_x \cdot n_x + v'_y \cdot n_y = 5 \cdot -1 + 2 \cdot 5 = -5 + 10 = 5 \neq 0.$$

Pokušamo li normalu transformirati na isti način kao i segment, opet nećemo dobiti korektnu normalu. Evo i dokaza. Izračunajmo najprije transformiranu normalu matricom \mathbf{M} (u homogenom prostoru).

$$n' = n \cdot \mathbf{M} = (-1, 5, 1) \cdot \begin{vmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{vmatrix} = (-1, 10, 1)$$

Skalarni produkt "nove" normale i vektora pravca daje:

$$v'_x \cdot n'_x + v'_y \cdot n'_y = 5 \cdot -1 + 2 \cdot 10 = -5 + 20 = 15 \neq 0.$$

Kako onda doći do potrebne transformacije? Vratimo se na početak. Imali smo vektor \vec{v} i normalu \vec{n} . Ta dva vektora jesu bili okomiti, i zadovoljavali su izraz:

$$v \cdot n = 0.$$

Uvažavajući da vektore prikazujemo kao jednoretčane matrice, prethodni se izraz za skalarni produkt može zamijeniti matričnim množenjem, pri čemu drugi član treba transponirati:

$$v \ n^T = 0.$$

Transformacijom vektora \vec{v} dobiva se vektor $\vec{v} \mathbf{M}$ čime u prethodni izraz između v i n^T dolazi matrica \mathbf{M} zbog čega jednakost prestaje vrijediti. Da bismo je vratili, matricu \mathbf{M} treba neutralizirati – množeći je njezinim inverzom. Evo ideje:

$$v \mathbf{M} \mathbf{M}^{-1} n^T = 0.$$

Grupiranjem slijedi:

$$(v \mathbf{M}) (\mathbf{M}^{-1} n^T) = v' n'^T = 0.$$

Sada je očito da je:

$$n'^T = \mathbf{M}^{-1} n^T \Rightarrow n' = (\mathbf{M}^{-1} n^T)^T = n \mathbf{M}^{-1 T}$$

Iz ovog razmatranja došli smo do konačnog zaključka: normale se transformiraju množenjem s transponiranim inverzom originalne transformacije. Provjerimo to još i na radnom primjeru.

$$\mathbf{M}^{-1} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{vmatrix}, \quad \mathbf{M}^{-1 T} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{vmatrix}.$$

$$n' = n \mathbf{M}^{-1 T} = \left(-1, \frac{5}{2}, 1\right)$$

$$v'_x \cdot n'_x + v'_y \cdot n'_y = 5 \cdot -1 + 2 \cdot \frac{5}{2} = -5 + 5 = 0.$$

Poglavlje 5

Projekcije i transformacije pogleda

5.1 Projekcije

Područje računalne grafike osim rada u 2D prostoru obuhvaća i rad u 3D prostoru. Štoviše, ljudima je koncepcija 3D prostora daleko bliža nego 2D prostor. Razloga tome je mnogo, a jedan je i taj što živimo u 3D prostoru pa su pojmovi poput iznad, ispod, lijevo, desno, ispred ili iza sasvim prirodni jasni. No u 2D prostoru ograničeni smo na samo dva smjera. U 2D prostoru ne možemo prikazati tijela; moguće je isključivo prikaz likova. S druge strane, prikazne jedinice danas su još uvek dominantno dvodimenzijske. Zaslon monitora nudi nam mogućnost prikazivanja u jednoj ravnini. S druge strane, čovjek nastoji i u svijet računala uvesti 3D prostor. Dakako, to je jednim dijelom moguće. Sjetimo se samo fotoaparata. Kada slikamo, slikamo objekte u 3D prostoru. Kao rezultat slikanja dobivamo sliku, dakle komad papira na kojem je "slika" smještena u ravninu – u 2D prostor. Ipak, pogledom na sliku dobivamo jasnú predodžbu o onome što je slikano; imamo privid 3D prostora. Naravno, u tom 3D prostoru ne možemo pogledati kako bi objekti izgledali kada bismo ih pogledali malo desnije, ili pak kada bismo otišli iza njih. Dobivena slika jednostavno je zamrznut prikaz onoga što smo vidjeli točno sa mjesta s kojeg smo gledali i točno u smjeru u kojem smo gledali. I to je mjesto na koje uskače računalna grafika. Tu se dakle pruža mogućnost da i na računalu kreiramo takve "snimke" koje nam pokazuju što bismo vidjeli od 3D prostora kada bismo stajali u jednoj točki u prostoru i gledali prema nekoj drugoj točki.

Metode koje opisuju na koji način "gledamo" i što bismo zapravo vidjeli zovu se – **projekcije**. Naziv "projekcije" dolazi od činjenice da objekte 3D prostora "projiciramo" na ravninu u 2D prostor. Projekcije su matematički modeli koji nam govore na koji način treba točke 3D prostora preslikati u ravninu u točke 2D prostora. Postoji više vrsta projekcija, a mi ćemo u nastavku obraditi dvije:

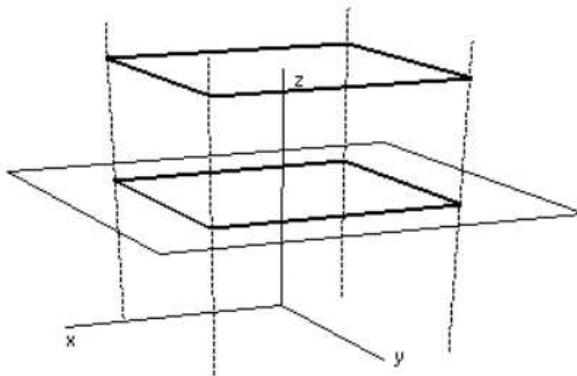
- paralelna projekcija te
- perspektivna projekcija.

Evo jednostavnog primjera što znači projicirati objekte 3D prostora u 2D prostor. Uzmimo list papira i stavimo ga na stol. Iznad njega (ali ne na njega) postavimo nekakav objekt, primjerice olovku, a iznad postavimo uključenu svjetiljku. Rezultat je sjena olovke na papiru; 3D objekt preslikao se je u ravninu.

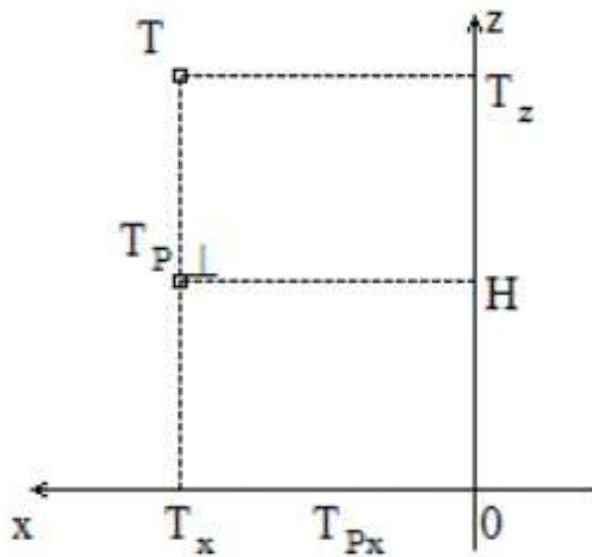
Ovaj jednostavan pokus pokazao nam je još nešto – projekcije su destruktivne. U 3D prostoru znamo i duljinu olovke, i debljinu olovke, promjer, udaljenost do papira, kut pod kojim je olovka nagnuta s obzirom na ravninu papira i sl. Projiciranjem u 2D prostor dobili smo sliku iz koje više ne možemo doznati te informacije.

5.1.1 Paralelna projekcija

Jedan od najjjednostavnijih modela projekcija jest model paralelne (kažemo još i ortografske) projekcije. Taj model podrazumijeva da je izvor svjetlosti smješten u beskonačnosti, pa su sve zrake svjetlosti koje dolaze do objekata okomite na ravninu projiciranja i samo takve zrake tvore sliku. Ovakvu projekciju



Slika 5.1: Paralelni projekciji



Slika 5.2: Paralelni projekciji, analiza x-koordinate

daje dakle točkasti izvor smješten iznad ravnine projiciranja i to na beskonačnoj udaljenosti. U tom slučaju projiciranjem objekta širokog 10 cm dobili bismo i sliku široku točno 10 cm.

Za opisivanje projekcije poslužiti ćemo se slijedećim primjerom. Želimo dobiti paralelnu projekciju točaka na ravninu $z = H$ pri čemu je H je proizvoljan broj (to je dakle projekcija u xy -ravninu na visini H). Slika 5.1 prikazuje problem.

Na slici je prikazan lik koji se projicira. Ispod lika smještena je ravnina projekcije s likom koji se dobije projiciranjem. Na slici su također prikazane i karakteristične zrake projiciranja kroz sva četiri vrha lika. Zrake su okomite na ravninu projekcije.

Ovaj model uzet je zbog jednostavnosti, kako bismo se lakše upoznali s idejom paralelne projekcije. Rješenje zadalog problema je jednostavno. Ako proizvoljnu točku T projiciramo, dobiti ćemo točku T_P i pri tome će za komponente točke T_P vrijediti:

$$\begin{aligned}T_{P,x} &= T_x \\T_{P,y} &= T_y \\T_{P,z} &= H\end{aligned}$$

To se jasno vidi sa slike 5.2. Na slici je prikazana situacija za x -komponentu, no isto vrijedi i za y -komponentu. Projicira se točka T , a projekcija je točka T_P .

Sve točke koje ćemo na ovaj način projicirati, imati će z -koordinatu jednaku H ; dakle, sve će ležati u ravnini $z = H$. Ovdje prikazano paralelno projiciranje može se opisati matricom paralelne projekcije:

$$\pi_{par} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & H & 1 \end{bmatrix}$$

Tada se projekcija opisuje umnoškom točke i matrice projekcije:

$$\begin{aligned} T_{Ph} &= [T_{Ph,x} \quad T_{Ph,y} \quad T_{Ph,z} \quad T_{Ph,h}] \\ &= T_h \cdot \pi_{par} \\ &= [T_x \quad T_y \quad T_z \quad 1] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & H & 1 \end{bmatrix} \\ &= [T_x \quad T_y \quad H \quad 1] \end{aligned}$$

Kada dobivene točke prikazujemo u ravnini, tada kao x - i y -koordinatu koristimo prve dvije komponente točke, dok ostale komponente zanemarujemo.

Prilikom crtanja točaka u ravnini pri tome treba voditi računa o još jednom problemu: ako se dvije točke 3D prostora preslikavaju u istu točku ravnine (jedna zelena a druga plava), koju ćemo točku tada prikazati u ravnini? Da bismo odgovorili na ovo pitanje, treba se prisjetiti koja je zapravo svrha projekcije – želimo "vidjeti" 3D prostor na zaslonu. Ako je to istina, tada se dvije različite točke 3D prostora koje se preslikavaju u istu točku 2D prostora mogu tumačiti kao dva objekta smještena jedan iza drugoga. Koji ćemo od tih objekata vidjeti, ovisi o tome gdje se nalazi promatrač. Pretpostavimo stoga da se u 3D prostoru u nekoj točki nalazi promatrač, potom se na određenoj udaljenosti nalazi ravnina projekcije i konačno, svi se objekti scene nalaze dalje iza te ravnine. U tom slučaju, promatrač će vidjeti onaj objekt koji je bliži ravnini projekcije, i to je uobičajeni scenarij koji ćemo dalje koristiti. Želimo li sačuvati informaciju o udaljenosti, matricu projekcije potrebno je modificirati. Iskoristit ćemo z -koordinatu projicirane točke za pohranu udaljenosti projicirane točke od same ravnine. Tada vrijedi:

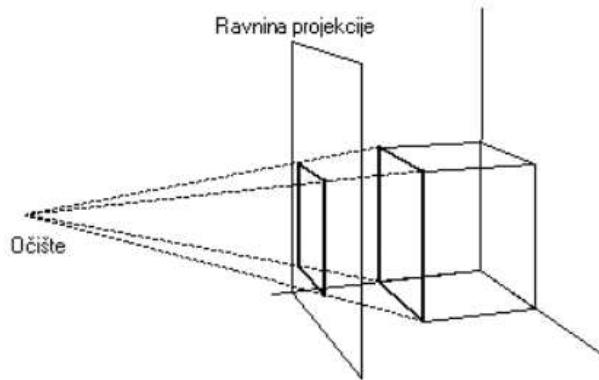
$$\begin{aligned} T_{P,x} &= T_x \\ T_{P,y} &= T_y \\ T_{P,z} &= T_z - H \end{aligned}$$

dok se matrica paralelne projekcije koja čuva udaljenost točke od ravnine modificira u:

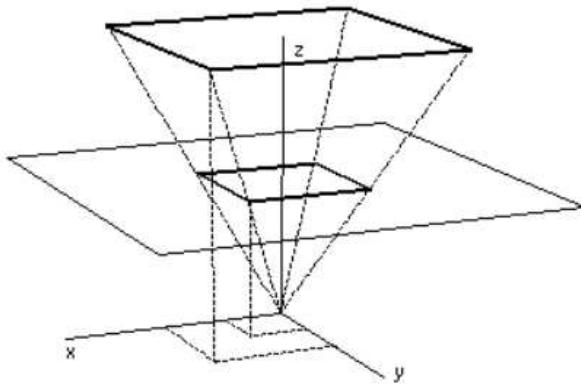
$$\pi'_{par} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -H & 1 \end{bmatrix}.$$

Rezultat koji se dobije množenjem točke koju projiciramo i ove matrice treba tumačiti na sljedeći način.

- Prve dvije komponente točke odgovaraju komponentama točke u ravnini projekcije.
- z -koordinata točke, budući da točka pripada ravnini $z = H$ iznosi upravo H .
- Treća komponenta točke odgovara udaljenosti točke koju smo projicirali od ravnine projekcije, i može poslužiti kao kriterij za određivanje koju točku treba prikazati u slučaju da se više točaka projicira u istu točku ravnine. Ovu informaciju možemo iskoristiti uz podatkovnu strukturu z -spremnika za određivanje konačne vidljive točke.



Slika 5.3: Perspektivna projekcija

Slika 5.4: Perspektivna projekcija, analiza x -koordinate

Matrica paralelne projekcije ispala je ovako jednostavna zbog toga što smo odabrali vrlo jednostavan primjer projekcije: projekciju na ravninu $z = H$. Razumno bi bilo pitati se a kako bi izgledala matrica paralelne projekcije na proizvoljnu ravninu u 3D prostoru. No odgovor na ovo glasi: takvu matricu (na svu sreću) ne moramo tražiti jer bi ispala krajnje komplikirana i nepregledna. Umjesto toga, primijeniti ćemo postupak *transformacije pogleda* o kojem će biti više riječi u nastavku, i zatim iskoristiti upravo ovu jednostavnu matricu paralelne projekcije. Prije no što se upustimo u postupak transformacija pogleda, pogledajmo još i drugi tip projekcije.

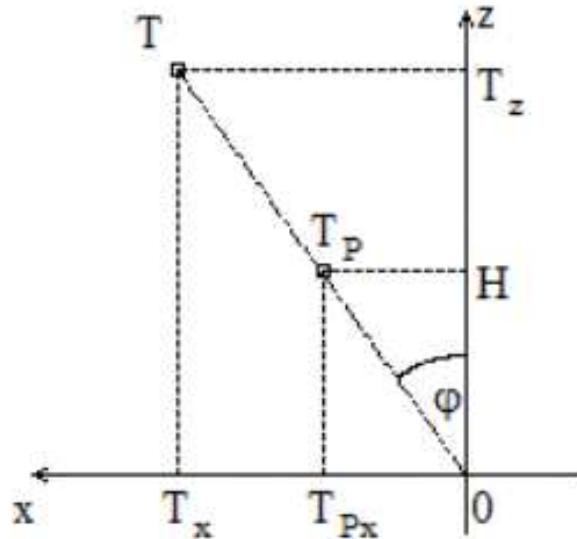
5.1.2 Prespektivna projekcija

Perspektivna projekcija je model koji je čovjeku bliži. Naime, model uvodi dvije točke: očište i gledište. Očište je točka u kojoj se nalazi promatrač. Gledište određuje smjer prema kojem promatrač gleda. U točki gledišta stvara ravnina projekcije koja je okomita na spojnici očište-gledište. Točka gledišta pripada toj ravnini i ona tipično postaje ishodište lokalnog dvodimenzijskog koordinatnog sustava koji razapinjemo u ravnini projekcije.

Projekcija proizvoljne točke T dobije se tako da se očište spoji pravcem sa zadanim točkom T . Kao projekcija točke uzima se točka u kojoj tako dobiveni pravac probada ravninu projekcije. Općeniti primjer perspektivne projekcije prikazan je na slici 5.3.

Za matematičku analizu problema pokušajmo odrediti perspektivnu projekciju proizvoljne točke T , za slučaj da je očište smješteno u ishodište koordinatnog sustava, a gledište na z -os na visini $z = H$. Ovakvim odabirom očišta i gledišta postiže se da je ravnina projekcije upravo ravnina $z = H$ (dakle, xy -ravnina podignuta od ishodišta za H). Slika 5.4 pokazuje perspektivnu projekciju lika uz tako zadano očište i gledište.

Pogledajmo što se događa s pojedinim točkama. Očište (koje je smješteno u ishodištu) i vrhovi lika (promatrano sliku 5.3) spojeni su spojnicama koje probadaju i ravninu projekcije $z = H$. Na x -

Slika 5.5: Perspektivna projekcija, analiza za x -os

i y -koordinatnim osima označene su koordinate vrhova i probodišta ravnine projekcije. Analizu treba provesti za svaku koordinatu zasebno, no kako je rezultat identičan, u nastavku je na slici 5.5 prikazan slučaj za jedan vrh i to za x -koordinatu.

Trokuti $0 - H - T_P$ i $0 - T_z - T$ su slični trokuti jer dijeli jednak kut ϕ . Tada vrijedi:

$$\frac{T_{P,x}}{H} = \frac{T_x}{T_z}$$

odakle slijedi:

$$T_{P,x} = T_x \cdot \frac{H}{T_z}.$$

Sličnom analiza za y -koordinatu projekcije slijedi:

$$\frac{T_{P,y}}{H} = \frac{T_y}{T_z} \Rightarrow T_{P,y} = T_y \cdot \frac{H}{T_z}.$$

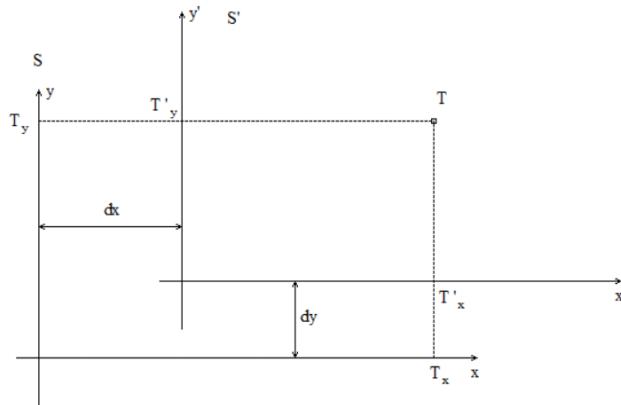
Ovdje izvedeni odnosi mogu se zapisati i matrično, ako za sve točke iskoristimo njihove homogene inačice. Dobije se:

$$\pi_{persp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{H} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

U točnost se možemo uvjeriti pomnožimo li točku i matricu projekcije:

$$\begin{aligned} T_{Ph} &= [T_{Ph,x} \ T_{Ph,y} \ T_{Ph,z} \ T_{Ph,h}] \\ &= T_h \cdot \pi_{persp} \\ &= [T_x \ T_y \ T_z \ 1] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{H} \\ 0 & 0 & 0 & 0 \end{bmatrix} \\ &= [T_x \ T_y \ 0 \ \frac{T_z}{H}] \end{aligned}$$

Prelaskom u radni prostor nakon dijeljenja sa homogenim parametrom dobiju se upravo izrazi od kojih smo krenuli.



Slika 5.6: Translatirani koordinatni sustavi

z -koordinata točke koja se dobije perspektivnom projekcijom preko prethodno izvedene matrice iznosi 0. Naime, kako točka leži u ravnini projekcije logično je za z -koordinatu uzeti vrijednost nula. No ponekad projekciju nećemo koristiti kao prijelaz iz 3D prostora u 2D prostor. U tom slučaju potrebno je projekciji svake točke ostaviti sve tri koordinate ispravnima: znači ako projiciramo na ravninu $z = H$, tada sve projicirane točke imaju z -koordinatu jednaku H . Matrica koja će raditi ovakvo projiciranje dobije se modifikacijom prethodne matrice:

$$\pi'_{persp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \frac{1}{H} \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Kao i kod paralelne projekcije, i ovdje ćemo se zaustaviti i nećemo ići u komplikiranije slučajeve, odnosno u opći slučaj (što bismo vidjeli kada bi očiše bilo negdje, a gledište negdje drugdje). Razlog tome je komplikiranost cijelog postupka ukoliko bismo išli ovako direktno, grubom silom. Umjesto toga, dovoljno je reći da se takav opći slučaj može postupkom transformacije pogleda svesti upravo na ovaj jednostavan. I stoga, pogledajmo što nam nudi transformacija pogleda...

5.2 Transformacije pogleda

Transformacije pogleda su postupci kojima se točke iz jednog koordinatnog sustava preslikavaju u drugi koordinatni sustav. Ideja je, kao i uvijek, krajnje jednostavna. Treba pronaći matricu 4×4 takvu se množenjem proizvoljne točke i te matrice dobijete točku s koordinatama koje bi originalna točka imala kada bismo je promatrali iz nekog drugog sustava. Postoji više načina kako se može izgraditi ovakva matrica. Najprije ćemo pogledati "intuitivni" način – primjenjivat ćemo elementarne transformacije koje smo već obradili kako bismo napravili traženo preslikavanje. Pred kraj ovog poglavlja pokazat ćemo i drugi način koji se temelji direktno na spoznajama vektorskih prostora linearne algebre. Krenimo s prvim načinom, i to kroz primjere.

Neka u koordinatnom sustavu S imamo točku T . Koje bi koordinate ta točka imala u sustavu S' koji bismo dobili kada bismo sustav S lagano trknuli tako da mu ishodište otklizi u točku O' ? U sustavu S koordinate točke O' su (dx, dy) . Problem je prikazan na slici 5.6.

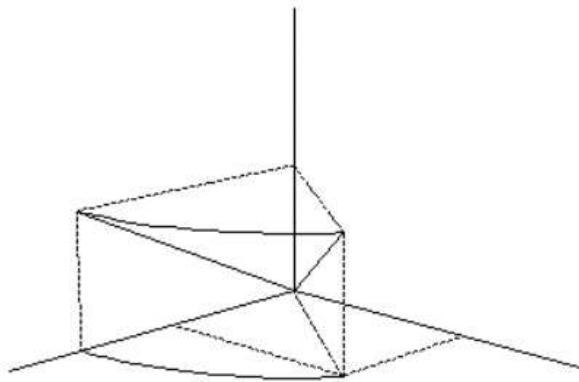
Problem je radi jednostavnost prikazan u 2D prostoru. Sa 5.6 jasno se vidi veza između koordinata u oba sustava:

$$T_x = T'_x + dx, \quad T_y = T'_y + dy$$

pa slijedi:

$$T'_x = T_x - dx, \quad T'_y = T_y - dy.$$

Proširenjem na koordinatni sustav u 3D prostoru dobije se:

Slika 5.7: Rotacija oko z -osi

$$T'_x = T_x - dx, \quad T'_y = T_y - dy, \quad T'_z = T_z - dz.$$

Uočimo sada da je primjenjena transformacija upravo translacija za $(-dx, -dy, -dz)$. Stoga je prva matrica upravo matrica translacije:

$$\Theta_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -dx & -dy & -dz & 1 \end{bmatrix}.$$

Ovim razmatranjem naučili smo kako preslikati koordinate iz jednog koordinatnog sustava u drugi, ako je ishodište drugog koordinatnog sustava samo pomaknuto za neki vektor (dx, dy, dz) . No što ukoliko je drugi koordinatni sustav umjesto pomaka uslijed našeg udarca iz gornjeg primjera doživio rotaciju? Radi jednostavnosti pretpostaviti ćemo da mu je ishodište bilo učvršćeno u referentni sustav pa nije moglo doći do pomaka, već samo do rotacije. Ovaj rotirani sustav označiti ćemo sa S' . Kako će tada izgledati koordinate zadane točke T u tom sustavu?

U 2D sustavu svaka rotacija jednoznačno je određena kutom rotacije – jednim kutom. U 3D prostoru opis rotacije zahtjeva uporabu prostornog kuta – kuta koji se može u pravokutnom koordinatnom sustavu opisati pomoću dvije komponente kuta: kutom koji projekcija rotirane točke u xy -ravninu čini s x -osi koordinatnog sustava, te kutom koji projekcija rotirane točke u xz -ravninu čini sa z -osi koordinatnog sustava (zapravo, ovo nije jedina mogućnost; mogu se koristiti bilo koja dva para ravnina i u njima odgovarajući kutovi). No unatoč ovako složenom opisu prostornog kuta, ideja rješenja je ista kao i kod prethodnog problema. U prethodnom slučaju rješenje smo dobili tako da smo pogledali što trebamo učiniti da bismo oba koordinatna sustava ponovno poklopili jedan preko drugoga; rješenje je bilo ishodištu novog sustava oduzeti vektor pomaka i time su se sustavi poklopili.

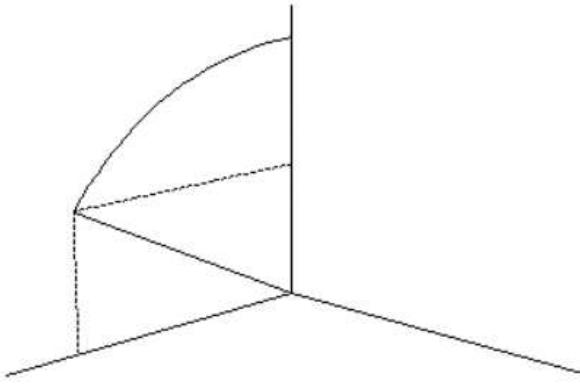
U ovom primjeru ishodišta se već poklapaju. Ono što se ne poklapa su osi. Ono što ćemo pokušati da bismo poklopili naše sustave jest sljedeće: uhvatit ćemo z -os novog sustava, i zarotirati je tako da se poklopi sa z -osi starog sustava. Pri tome ćemo najprije z -os zarotirati za kut α u xy -ravnini i to u smjeru kazaljke na satu (dakle u matematički negativnom smjeru) tako da padne u xz -ravninu. Slika 5.7 prikazuje postupak.

Uzmimo da se točka $G = (G_x, G_y, G_z)$ nalazi na z -osi zarotiranog sustava S' . Kut α što ga projekcija te točke u xy -ravninu originalnog sustava zatvara s x -osi određen je izrazima:

$$\cos\alpha = \frac{G_x}{\sqrt{G_x^2 + G_y^2}}, \quad \sin\alpha = \frac{G_y}{\sqrt{G_x^2 + G_y^2}}.$$

Rotiranjem za kut α točka G preslikat će se u točku G' koja leži u xz -ravnini. Pri tome su komponente točke G' određene izrazima:

$$G'_x = \sqrt{G_x^2 + G_y^2}$$

Slika 5.8: Rotacija oko y -osi

$$G'_y = 0$$

$$G'_z = G_z$$

Rotacijom u xy -ravnini z -koordinata točke ostala je očuvana. y -koordinata pala je na nulu jer je točka rotacijom stigla u xz -ravninu. Kako se ovdje radi o rotaciji točke oko z -osi početnog koordinatnog sustava u *smjeru kazaljke na satu* (što je matematički negativan smjer), pripadnu matricu rotacije glasi:

$$\Theta_2 = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Ovdje treba spomenuti i specijalni slučaj koji može nastupiti. Ako je $G_x = 0$ i $G_y = 0$, tada se ova rotacija preskače, jer su z -osi obaju sustava već kolinearne, te kut α nije definiran. Krenimo dalje.

Zajedno s točkom G koja se je preslikala u G' , z -os našeg rotiranog sustava također je pala u xz -ravninu referentnog sustava. Nova situacija prikazana je na slici 5.8.

Još nam je preostalo da točku G' odvučemo na z -os referentnog koordinatnog sustava rotacijom u xz -ravnini opet u smjeru kazaljke na sat. Kut β koji predstavlja kut u xz -ravnini što ga točka G' (odnosno precizno govoreći njezina projekcija na xz -ravninu) zatvara s z -osi određen je izrazima:

$$\cos\beta = \frac{G'_z}{\sqrt{(G'_x)^2 + (G'_z)^2}} \quad \sin\beta = \frac{G'_x}{\sqrt{(G'_x)^2 + (G'_z)^2}}.$$

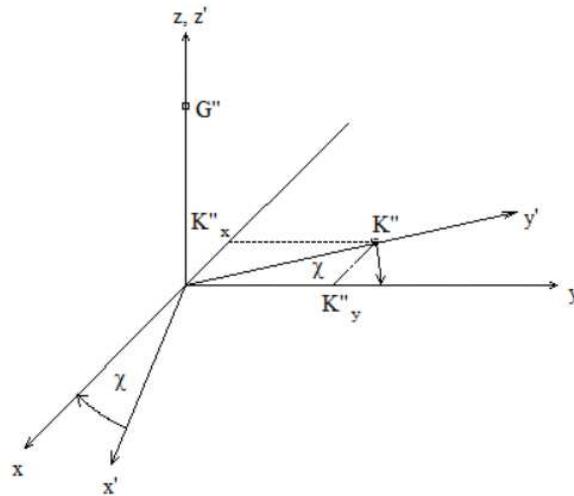
Rotacijom točke G' dobiva se točka G'' koja leži i na z -osi referentnog sustava, te za njezine komponente vrijedi:

$$\begin{aligned} G''_x &= 0, \\ G''_y &= 0, \\ G''_z &= \sqrt{(G'_x)^2 + (G'_z)^2} = \sqrt{G_x^2 + G_y^2 + G_z^2}. \end{aligned}$$

Matrični zapis ove transformacije glasi:

$$\Theta_3 = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Ovim postupkom postigli smo poklapanje z -osi referentnog sustava s rotiranim sustavom. Pri izvođenju izraza za $\sin\beta$ i $\cos\beta$ specijalnog slučaja nema. Naime, pogreška bi nastupila samo ako bi



Slika 5.9: Rotacija oko \$z\$-osi

\$G'_x\$ i \$G'_z\$ bili istodobno jednaki nuli, no to se ne može dogoditi osim u slučaju da se je polazna točka \$G\$ poklapala s ishodištem referentnog sustava (što ne smije jer tada nismo niti definirali \$z\$-os).

Rotacijama \$\Theta_2\$ i \$\Theta_3\$ postigli smo ispravno poklapanje \$z\$-osi referentnog sustava i \$z\$-osi zarotiranog sustava. No jesmo li tim rotacijama uspjeli poklopiti i \$x\$- i \$y\$- osi? Generalno govoreći, i jesmo i nismo. Odgovor "jesmo" možemo pravdati ovako: rotirani sustav zadali smo samo jednom točkom – točkom na \$z\$-osi. Zbog toga smo položaj \$x\$- i \$y\$- osi rotiranog sustava ostavili nedefinirane. Zatim smo rotacijama postigli poklapanje \$z\$-osi obaju sustava.

Sada možemo razmišljati dalje ovako. Što napraviti s \$x\$- i \$y\$-osima? Budući da ih nikako nismo vezali, možemo se praviti da smo htjeli da budu baš takve da se sada upravo poklapaju s \$x\$- i \$y\$- osima referentnog sustava. Iako malo "nategnuta", pretpostavka se ne može pobiti tako dugo dok rotirani sustav ne definiramo malo bolje, pa tu u igru ulazi odgovor "nismo".

Trodimenijski sustav jednoznačno je zadan ako je poznato npr. ishodište, točka na \$z\$-osi, točka na \$y\$-osi, te ako je poznato da je sustav primjerice desni. Točku ishodišta znamo – to je ista točka koja je i ishodište referentnog sustava. Točku na \$z\$-osi isto znamo: zadali smo je kao točku \$G\$. Da bismo sustav definirali do kraja, zadajmo još i točku \$K\$ koja leži na \$y\$-osi sustava, i riješimo problem do kraja.

Na sustav \$S'\$ do ovog trenutka već smo djelovali rotacijama \$\Theta_2\$ i \$\Theta_3\$. Točka \$K\$ uslijed tih rotacija prešla je u točku \$K''\$:

$$K''_h = K_h \cdot \Theta_2 \cdot \Theta_3$$

pri čemu je \$K_h\$ homogena inačica točke \$K\$. Uslijed obavljenih rotacija \$z\$-komponenta točke \$K''\$ postala je nulom, dok su \$x\$- i \$y\$- komponente općenito različite od nule (barem jedna od njih). Situaciju prikazuje slika 5.9.

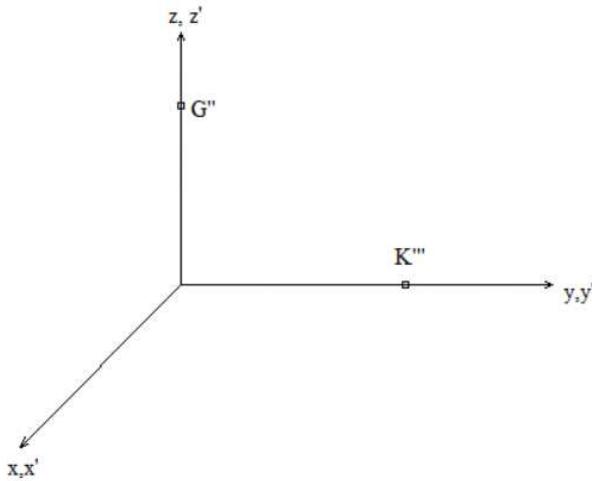
Kut \$\chi\$ koji određuje nepoklapanje između \$y\$- i \$y'\$- osi definiran je izrazima:

$$\cos \chi = \frac{K''_y}{\sqrt{(K''_y)^2 + (K''_x)^2}}, \quad \sin \chi = \frac{-K''_x}{\sqrt{(K''_y)^2 + (K''_x)^2}}.$$

Rotiranjem sustava \$S'\$ oko osi \$z\$ za kut \$\chi\$ postići ćemo potpuno poklapanje referentnog sustava i sustava \$S'\$. Matrica kojom se izvodi rotacija oko \$z\$-osi za kut \$\chi\$ glasi:

$$\Theta_4 = \begin{bmatrix} \cos \chi & -\sin \chi & 0 & 0 \\ \sin \chi & \cos \chi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Rezultat ove rotacije prikazan je na slici 5.10.



Slika 5.10: Poklopljeni sustavi

Ako se ipak odlučimo na nezadavanje točke K , može se uzeti jednostavna pretpostavka: kut χ iznosi 90° . Tada će matrica rotacije oko z -osi izgledati:

$$\Theta'_4 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Pogledamo li koje smo sve transformacije primijenili za rješavanje problema rotiranog sustava, vidjeti ćemo da smo iskoristili transformacije Θ_2 , Θ_3 i Θ_4 ili Θ'_4 . To znači da se cijeli posao može obaviti jednom transformacijom $\Theta_2 \cdot \Theta_3 \cdot \Theta_4$ (odnosno $\Theta_2 \cdot \Theta_3 \cdot \Theta'_4$). Dakle, ukoliko imamo zadane koordinate točke T u referentnom sustavu, i imamo zadanu jednu točku G koja leži na z -osi našeg rotiranog sustava, te točku K koja se nalazi na y -osi našeg rotiranog sustava, tada ćemo koordinate točke T u rotiranom sustavu S' dobiti množenjem točke T i transformacijskih matrica.

Još nam je preostalo odgovoriti na najopćenitiji slučaj: kako bi glasile koordinate točke T u sustavu koji je uslijed "udarca" pretrpio i rotaciju, i translaciju? Tada mu se niti ishodišta, niti osi više ne poklapaju. Odgovor na ovo pitanje dati će nam kompozicija prethodnih slučajeva: prvo treba ishodište rotiranog sustava vratiti u ishodište referentnog sustava (matricom Θ_1), a zatim sustav treba još dodatno zarotirati matricom $\Theta_2 \cdot \Theta_3 \cdot \Theta_4$. Pri tome treba обратити pažnju na točke koje će se uzeti kao G točka i K točka za određivanje Θ_2 , Θ_3 i Θ_4 . Naime, kada matricom Θ_1 ishodište rotiranog sustava vratimo u ishodište referentnog sustava, tada i izvorne točke G i K treba translatirati istom matricom Θ_1 i tako dobivene točke treba uzeti za određivanje matrica Θ_2 , Θ_3 i Θ_4 .

Sa slike 5.10 vidljivo je da se orientacija sustava prilikom transformacija ne mijenja: polazni sustav bio je desni – završni sustav također je desni. Postoji transformacija pogleda koja nam omogućava i tu promjenu. Ukoliko je referentni sustav lijevi, a odredišni sustav desni (ili obrnuto), tada će se koordinate između sustava transformirati jednostavnim okretanjem predznaka na x -osi što čini matrica:

$$\Theta_5 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Ovime završavamo pregled transformacija pogleda.

5.3 Primjena na poopćenje perspektivne projekcije

U sekciji 5.1.2 pokazali smo osnove perspektivne projekcije. Uveli smo točku očišta kao onu točku gdje stavljamo naše oko, te točku gledišta kao onu točku koja predstavlja točku koja pripada ravnini

projekcije, i koja ujedno tvori ishodište 2D koordinatnog sustava u toj ravnini. Dodatno smo rekli da je ravnina projekcije određena jednom pripadnom točkom (gledištem) i normalom (vektor očište-gledište). No tu smo stali. Kao primjer perspektivne projekcije uzeli smo najjednostavniji mogući slučaj: očište je u ishodištu koordinatnog sustava, a gledište je na z -osi i to točka $(0, 0, H)$ gdje je H udaljenost očišta od gledišta.

Kako bismo došli do općeg slučaja, razmotrit ćemo situaciju kada su očište i gledište proizvoljne točke u prostoru. Kako tada naći perspektivnu projekciju? Pogledajmo još jednom što znamo. Znamo naći perspektivnu projekciju ako se očište nalazi u ishodištu, a gledište na z -osi. Međutim, u općem slučaju, točke očišta i gledišta su proizvoljne točke čije koordinate znamo u globalnom koordinatnom sustavu scene. Ono što nas zanima jest koje su koordinate objekata gledano iz novog koordinatnog sustava čije je ishodište u točki očišta te čija z -os prolazi kroz očište i gledište. Pogodili ste – trebamo transformaciju pogleda. Pa krenimo redom.

Očište ćemo u nastavku označavati s Q a gledište s R .

5.3.1 Korak 1

Prisjetimo li se transformacija pogleda, tada je prvi korak vraćanje novog ishodišta u ishodište referentnog sustava. Dakle, koristimo matricu Θ_1 , uz pomake koji odgovaraju upravo koordinatama očišta. Naime, očište (ishodište novog koordinatnog sustava) trebamo vratiti u ishodište globalnog koordinatnog sustava. Transformacije ćemo raditi nad proizvoljnom točkom T .

$$\Theta_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Q_x & -Q_y & -Q_z & 1 \end{bmatrix}.$$

Ova transformacija očište će preslikati u ishodište, dok će gledište preslikati u:

$$\begin{aligned} R'_h &= R_h \cdot \Theta_1 \\ &= [R_{h,x} \ R_{h,y} \ R_{h,z} \ 1] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Q_x & -Q_y & -Q_z & 1 \end{bmatrix} \\ &= [R_{h,x} - Q_x \ R_{h,y} - Q_y \ R_{h,z} - Q_z \ 1]. \end{aligned}$$

Prema tome, za pojedine komponente vrijedi:

$$R'_x = R_x - Q_x$$

$$R'_y = R_y - Q_y$$

$$R'_z = R_z - Q_z$$

5.3.2 Korak 2

Sada kada se ishodišta obaju sustava poklapaju, treba izvršiti rotaciju sustava tako da im se z -osi poklope. Prvi korak je rotacija u xy -ravnini matricom Θ_2 .

$$\Theta_2 = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

pri čemu vrijede izrazi koje smo već prethodno izveli:

$$\cos\alpha = \frac{G_x}{\sqrt{G_x^2 + G_y^2}}, \quad \sin\alpha = \frac{G_y}{\sqrt{G_x^2 + G_y^2}}.$$

Kako je G točka upravo jednaka točki gledišta nakon prve transformacije: $G = R'$, pa se može pisati:

$$\cos\alpha = \frac{R'_x}{\sqrt{(R'_x)^2 + (R'_y)^2}}, \quad \sin\alpha = \frac{R'_y}{\sqrt{(R'_x)^2 + (R'_y)^2}}.$$

Transformacija Θ_2 djelovanjem na točku R' daje točku R'' :

$$\begin{aligned} R''_h &= R'_h \cdot \Theta_2 \\ &= [\begin{array}{cccc} R'_x & R'_y & R'_z & 1 \end{array}] \cdot \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= [\begin{array}{cccc} R'_x \cdot \cos\alpha + R'_y \cdot \sin\alpha & -R'_x \cdot \sin\alpha + R'_y \cdot \cos\alpha & R'_z & 1 \end{array}]. \end{aligned}$$

Uvrštavanjem izraza za $\sin\alpha$ i $\cos\alpha$ dobiva se:

$$\begin{aligned} R''_x &= \sqrt{(R'_x)^2 + (R'_y)^2} \\ R''_y &= 0 \\ R''_z &= R'_z \end{aligned}$$

5.3.3 Korak 3

Sada je potrebno primijeniti i rotaciju u xz -ravnini transformacijom Θ_3 .

$$\Theta_3 = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Pri tome su:

$$\cos\beta = \frac{R''_z}{\sqrt{(R''_x)^2 + (R''_z)^2}} \quad \sin\beta = \frac{R''_x}{\sqrt{(R''_x)^2 + (R''_z)^2}}.$$

Transformacija Θ_3 djelovanjem na točku R'' daje točku R''' , što po komponentama daje:

$$\begin{aligned} R''''_x &= 0 \\ R''''_y &= 0 \\ R''''_z &= \sqrt{(R''_x)^2 + (R''_z)^2} = \sqrt{(R'_x)^2 + (R'_y)^2 + (R'_z)^2} \end{aligned}$$

Primjenimo li još i transformaciju Θ'_4 (a po potrebi i Θ_5 ako su sustavi različito orijentirani), učinili smo sve potrebne transformacije kako bi sustav s ishodištem u očištu i točkom gledišta na njegovoj z -osi preveli u sustav kojemu se ishodište i z -os poklapaju s referentnim sustavom. Što smo time dobili? Ako znamo točku T u referentnom sustavu, tada su njezine koordinate u sustavu s ishodištem u očištu jednake:

$$T' = T \cdot \Theta_1 \cdot \Theta_2 \cdot \Theta_3 \cdot \Theta'_4 \cdot \Theta_5.$$

Budući da su nam time poznate koordinate u sustavu u kojem je očište jednako ishodištu, a gledište se nalazi na z -osi, tada točku T' znamo prethodno izvedenom jednostavnom matricom perspektivno projicirati, te možemo pisati:

$$T'_p = T' \cdot \phi_{persp} = T \cdot \Theta_1 \cdot \Theta_2 \cdot \Theta_3 \cdot \Theta'_4 \cdot \Theta_5.$$

pri čemu je matrica perspektivne projekcije definirana kao:

$$\pi_{persp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{H} \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

H je pri tome udaljenost od očišta do gledišta pa se može izračunati iz tih podataka, ili se može primijetiti da smo tu udaljenost već izračunali prilikom transformacije pogleda, te se udaljenost nalazi kao z -komponenta točke R''_z . Dakle, za H vrijedi:

$$\begin{aligned} H &= \sqrt{(Q_x - R_x)^2 + (Q_y - R_y)^2 + (Q_z - R_z)^2} \\ &= \sqrt{(R'_x)^2 + (R'_y)^2 + (R'_z)^2} \\ &= \sqrt{(R''_x)^2 + (R''_z)^2} \\ &= \sqrt{(R'''_z)^2} \\ &= R'''_z \end{aligned}$$

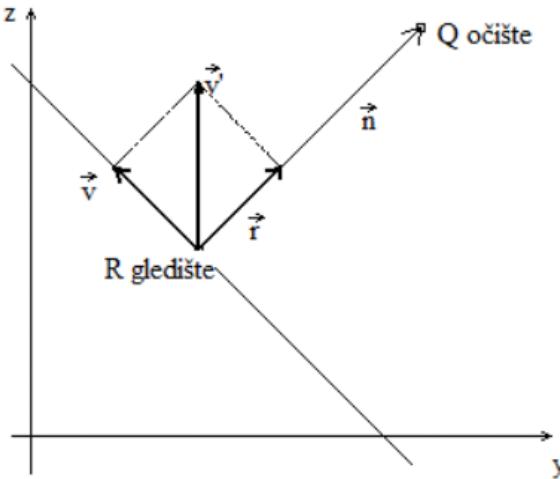
Da smo željeli dobiti opću paralelnu projekciju, postupak bi bio identičan, samo bismo umjesto posljednje matrice perspektivne projekcije uzeli matricu paralelne projekcije.

U postupku se koristi matrica Θ'_4 budući da prilikom zadavanja parametara za perspektivnu projekciju nismo zadali niti jedan podatak koji bi nam omogućio identificiranje položaja osi y . Ukoliko se želi potpuna kontrola nad slikom koju generira perspektivna projekcija, tada je potrebno zadati još i jednu točku K koja se nalazi na y -osi sustava u kojem radimo projekciju. Podsjetimo se još jednom zašto je to tako.

Kod opće perspektivne projekcije zadajemo dvije točke: očište Q i gledište R . Pri tome se stvara novi sustav s ishodištem u točci Q , i s pozitivnom z' -osi kroz točku R . Okomito na spojnici $R-Q$ u točki R stvara se ravnina projekcije, i lokalni dvodimenzionalni koordinatni sustav čije su osi x' i y' . Problem u svemu tome jest što specificiranjem samo točaka Q i R nemamo kontrolu nad x' - i y' -osima, tj. ne možemo nikako zadati: "želim da y' -os gleda baš ovako...". Jedna od mogućnosti da se dobije kontrola nad tim osima objašnjena je u poglavlju o transformacijama pogleda pomoću dodatne točke K . Ako prilikom zadavanja parametara perspektivne projekcije zadamo i točku K , tada ćemo umjesto matrice Θ'_4 koristiti matricu Θ_4 čime ćemo dobiti kontrolu nad svim osima sustava. Implementacija ove jednostavne modifikacije ostavlja se čitateljima za vježbu. No postoji i drugi pristup.

5.3.4 View-up vektor

View-up vektor popularan je naziv za jedan od načina "kroćenja" svih osi sustava (dakle osi x' i y'). Evo ideje. Korisnik će prilikom zadavanja parametara npr. perspektivne projekcije zadati i vektor koji će pokazivati smjer y' osi. Ovo može biti dosta problematično jer jedinični vektor y' osi leži u ravnini projekcije, a korisnik prilikom zadavanja "što želi dobiti" ne mora sam računati sve – tome služe računala. Stoga je problem slijedeći: korisnik programa želi reći primjerice neka "y' os gleda prema gore". U prirodi "prema gore" obično označava u smjeru pozitivne z -osi, čiji je reprezentant npr. vektor $\vec{v}_{up} = [0 \ 0 \ 1]$. Pretpostavimo sada da su očište i gledište zadani tako da ravnina projekcije leži pod kutem od 45° prema ravnini xz , i okomita je na yz -ravninu. Vektor koji će u tom slučaju gledati "prema gore" biti će npr. $\vec{v} = [0 \ -1 \ 1]$ jer on leži u ravnini projekcije (dok vektor \vec{v}_{up} očito ne leži). Postavlja se pitanje kako uz približni vektor \vec{v}_{up} koji zadaje korisnik pronaći pravi

Slika 5.11: Pronalaženje y -osi temeljem *view-up* vektora

vektor \vec{v} koji će pokazivati u smjeru osi y' . Problem je ilustriran na slici 5.11 (vektor \vec{v}_{up} označen je kao \vec{v}').

Rješenje se dobiva jednostavnim računom. Treba uočiti da vrijedi:

$$\vec{v}_{up} = \vec{r} + \vec{v}.$$

Neka je vektor $\vec{h} = Q - R$. Uočimo da je taj vektor ujedno i normala na ravninu projekcije. Kut između vektora \vec{h} i vektora \vec{v}_{up} definiran je izrazom:

$$\cos(\vec{h}, \vec{v}_{up}) = \frac{\vec{h} \cdot \vec{v}_{up}}{\|\vec{h}\| \cdot \|\vec{v}_{up}\|}.$$

Norma vektora \vec{r} slijedi iz pravokutnog trokuta:

$$\|\vec{r}\| = \|\vec{v}_{up}\| \cdot \cos(\vec{h}, \vec{v}_{up}) = \frac{\vec{h} \cdot \vec{v}_{up}}{\|\vec{h}\|},$$

dok je sam vektor jednak svojoj normi puta jedinični vektor u smjeru \vec{h} :

$$\vec{r} = \|\vec{r}\| \cdot \frac{\vec{h}}{\|\vec{h}\|} = \frac{\vec{h} \cdot \vec{v}_{up}}{\|\vec{h}\|^2} \cdot \vec{h}.$$

Sada iz jednakosti $\vec{v}_{up} = \vec{r} + \vec{v}$ možemo izvući traženi vektor \vec{v} :

$$\vec{v} = \vec{v}_{up} - \vec{r} = \vec{v}_{up} - \frac{\vec{h} \cdot \vec{v}_{up}}{\|\vec{h}\|^2} \cdot \vec{h}.$$

Iraz se da još pojednostavniti ako se vektor \vec{h} prije uporabe normira, pa koristimo:

$$\vec{h} = \frac{Q - R}{\|Q - R\|}.$$

U tom slučaju izraz za \vec{v} se pojednostavljuje u:

$$\vec{v} = \vec{v}_{up} - (\vec{h} \cdot \vec{v}_{up}) \cdot \vec{h}.$$

Koristeći dobiveni izraz točka K može se izračunati trivijalno:

$$K = R + \vec{v}.$$

Nakon izračunavanja točke K može se krenuti u izgradnju matrica za opću perspektivnu projekciju koristeći pri tome matricu Θ_4 . Uočimo još jednom: *view-up* vektor je vektor koji zadaje korisnik kako bi definirao smjer y -osi lokalnog 2D koordinatnog sustava razapetog u ravnini projekciji s ishodištem u gledištu. Tipično, *view-up* vektor ne mora ležati u ravnini projekcije; ravnina projekcije uvijek je okomita na spojnicu očište-gledište. Projekcija *view-up* vektora u ravninu projekcije daje vektor kolinearan jediničnom vektoru y -osi.

5.4 Transformacije pogleda na drugi način

5.4.1 Izvod

U prethodnom dijelu ovog poglavlja vidjeli smo jedan način prelaska između dva koordinatna sustava. U nastavku ćemo dati prikaz još jedne mogućnosti. Zanima nas koje će koordinate imati točka T u sustavu S' ako znamo koordinate točke T u referentnom sustavu S . Neka je referentni sustav S naš poznati osnovni sustav s osima x , y i z koje su u smjeru vektora \vec{i} , \vec{j} i \vec{k} . Sustav S' možemo zadati na slijedeći način: ishodište mu je u točki C (čije koordinate znamo u sustavu S), pozitivna z -os zadana je vektorom \vec{N} , pozitivna y -os zadana je vektorom \vec{V} a pozitivna x -os zadana je vektorom \vec{U} .

Prilikom zadavanja vektora \vec{N} , \vec{V} i \vec{U} treba paziti da su vektori međusobno okomiti i jedinični (znači, moraju biti ortonormirani). Ako zadani vektori nisu jedinični, treba ih prije uporabe normirati. Kako sada doći do koordinata točke T u sustavu S' ? Prema već navedenom receptu, najprije treba ishodište sustava S' dovesti u ishodište sustava S . To ćemo postići maticom Θ_1 uz pomak koji odgovara negativnim koordinatama ishodišta sustava S' mjereno iz sustava S :

$$\Theta_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -C_x & -C_y & -C_z & 1 \end{bmatrix}.$$

Ovime smo postigli poklapanje ishodišta obaju sustava. Sada još treba očitati koordinate. No prije nego što se upustimo u to, prisjetimo se, što su zapravo koordinate? Koordinatni sustav zadan je svojim baznim vektorima. Ako od ishodišta krenemo malo po jednom vektoru, pa nakon toga krenemo po drugom vektoru, i tako po svim vektorima, došli smo u neku točku prostora. Koordinate te točke su upravo one "malo po jednom vektoru"-komponente. Točka u koju smo stigli može se predočiti vektorom: projekcije tog vektora na svaki bazni vektor daje upravo odgovor "koliko malo" smo se pomaknuli po tom vektoru. Dakle, svaku koordinatu točke možemo dobiti tako da vektor točke projiciramo na odgovarajući bazni vektor. A projekcija jednog vektora na drugi je upravo njihov skalarni produkt (ako je vektor na koji projiciramo jediničan; inače treba rezultat podijeliti s njegovom normom). Pogledajmo to na primjeru. U referentnom sustavu zadan je točka $T = (5, 2, 1)$. Pretvorbom u vektor dobivamo $5\vec{i} + 2\vec{j} + 1\vec{k}$ jer je referentni sustav upravo zadan vektorima \vec{i} , \vec{j} i \vec{k} . Koliko iznosi x -komponenta točke? Množimo $(5\vec{i} + 2\vec{j} + 1\vec{k})$ skalarno s $(1\vec{i} + 0\vec{j} + 0\vec{k})$ i rezultat je upravo 5. y -komponenta dobije se množenjem $(5\vec{i} + 2\vec{j} + 1\vec{k})$ s $(0\vec{i} + 1\vec{j} + 0\vec{k})$ i rezultat je 2. z -koordinata dobije se množenjem $(5\vec{i} + 2\vec{j} + 1\vec{k})$ s $(0\vec{i} + 0\vec{j} + 1\vec{k})$ i rezultat je 1. Po ovoj analogiji može se pokazati da komponente točke T u sustavu S' odgovaraju upravo projekcijama vektora T na bazne vektore sustava S' . Kako je sustav S' zadan s tri bazna vektora: \vec{U} , \vec{V} i \vec{N} (koji su jedinični), vrijedi:

$$\begin{aligned} T'_x &= \vec{T} \cdot \vec{U} = T_x \cdot U_x + T_y \cdot U_y + T_z \cdot U_z, \\ T'_y &= \vec{T} \cdot \vec{V} = T_x \cdot V_x + T_y \cdot V_y + T_z \cdot V_z, \\ T'_z &= \vec{T} \cdot \vec{N} = T_x \cdot N_x + T_y \cdot N_y + T_z \cdot N_z. \end{aligned}$$

Ovo se može i matrično zapisati, pa matrica Θ_2 glasi:

$$\Theta_2 = \begin{bmatrix} U_x & V_x & N_x & 0 \\ U_y & V_y & N_y & 0 \\ U_z & V_z & N_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Ukupna matrica transformacije pogleda tada glasi: $\Theta = \Theta_1 \cdot \Theta_2$.

U postupku smo pretpostavili da su zadana sva tri vektora: \vec{U} , \vec{V} i \vec{N} . No lako je pokazati da ne moraju biti zadani baš svi vektori. Npr. ukoliko su zadani vektori \vec{V} i \vec{N} , vektor \vec{U} može se dobiti kao \times -proizvod vektora \vec{V} i \vec{N} (što će rezultirati desnom orijentacijom sustava S') ili kao \times -proizvod vektora \vec{N} i \vec{V} (što će rezultirati lijevom orijentacijom sustava S').

Nadalje, vektor \vec{V} mora biti zadan tako da bude okomit na vektor \vec{U} i \vec{N} . Ukoliko se korisniku dopusti da zada vektor koji je približno okomit, tada se pravi okomit vektor može izračunati iz poznatog *view-up* vektora.

Najslobodniji pristup dopustit će korisniku zadavanje trodimenzijske točke C , vektora \vec{N} kao po-kazivača pozitivnog smjera osi z' , vektor smjera \vec{v}_{up} koji i ne mora biti baš okomit na vektor \vec{N} te podatak želi li se lijevi ili desni sustav S' . U tom slučaju prikladan postupak računanja je sljedeći: prvo treba normirati vektor \vec{N} , zatim izračunati pravi vektor \vec{V} , te iz podatka o orijentaciji sustava treba izračunati vektor \vec{U} kao \times -proizvod. Zatim se slože matrice Θ_1 i Θ_2 te ukupna matrica transformacije glasi: $\Theta = \Theta_1 \cdot \Theta_2$.

5.4.2 Primjena na perspektivnu projekciju

Točka C bit će očište. Gledište možemo zadati na dva načina.

1. Eksplicitnim zadavanjem točke gledišta R ; tada se vektor \vec{N} računa kao $R - C$ uz naknadno normiranje. Tada se za H može uzeti norma od $R - C$, ili se točka R može koristiti samo za određivanje vektora \vec{N} , dok se H i dalje zadaje proizvoljno.
2. Zadavanjem vektora \vec{N} (koji po potrebi treba normirati), te zadavanjem udaljenosti H (u ovom slučaju H se mora zadati).

Uz ove podatke potrebno je zadati i *view-up* vektor, te željenu orijentaciju sustava.

5.5 Transformacija pogleda i projekcije u OpenGL-u

Nakon što smo se upoznali s matematičkim osnovama transformacija u 2D i 3D prostoru, njihovom uporabom kod transformacije pogleda i na kraju s projekcijama, pogledajmo kako je to sve organizirano u OpenGL-u. U ovoj sekciji naučit ćemo na koji način OpenGL radi transformacije vrhova i koje nam naredbe stoje na raspolaganju kojima možemo upravljati tim procesom. Pa krenimo redom.

Prisjetimo se – kada nešto crtamo u OpenGL-u, unutar bloka `glBegin(...); /glEnd();` zadajemo jedan ili više vrhova kojima definiramo objekt koji postoji u sceni. Koordinate koje zadajemo su pri tome koordinate u trodimenzijskom koordinatnom sustavu scene. Primjer funkcije koja crta žičani model kocke sa stranicom duljine w i s centrom smještenim u ishodištu koordinatnog sustava dat je u nastavku. Tu funkciju koristit ćemo u ostatku ove sekcije u primjerima.

```
void kocka( float w ) {
    float wp = w/2.0f;
    // gornja stranica
    glBegin(GL_LINE_LOOP);
    glVertex3f(-wp, -wp, wp);
    glVertex3f(wp, -wp, wp);
    glVertex3f(wp, wp, wp);
    glVertex3f(-wp, wp, wp);
    glEnd();

    // donja stranica
    glBegin(GL_LINE_LOOP);
    glVertex3f(-wp, wp, -wp);
    glVertex3f(wp, wp, -wp);
    glVertex3f(wp, -wp, -wp);
    glVertex3f(-wp, -wp, -wp);
    glEnd();
```

```

// desna stranica
glBegin(GL_LINE_LOOP);
glVertex3f(wp, wp, -wp);
glVertex3f(-wp, wp, -wp);
glVertex3f(-wp, wp, wp);
glVertex3f(wp, wp, wp);
glEnd();

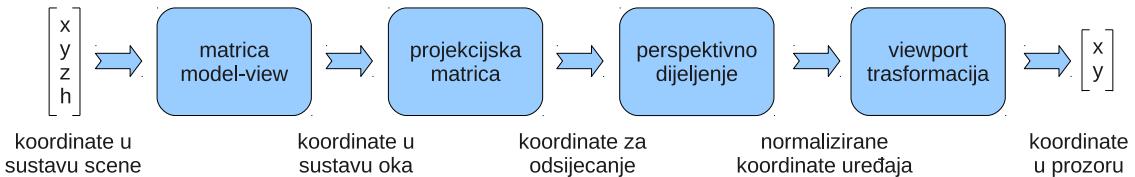
// lijeva stranica
glBegin(GL_LINE_LOOP);
glVertex3f(wp, -wp, wp);
glVertex3f(-wp, -wp, wp);
glVertex3f(-wp, -wp, -wp);
glVertex3f(wp, -wp, -wp);
glEnd();

// prednja stranica
glBegin(GL_LINE_LOOP);
glVertex3f(wp, -wp, -wp);
glVertex3f(wp, wp, -wp);
glVertex3f(wp, wp, wp);
glVertex3f(wp, -wp, wp);
glEnd();

// straznja stranica
glBegin(GL_LINE_LOOP);
glVertex3f(-wp, -wp, wp);
glVertex3f(-wp, wp, wp);
glVertex3f(-wp, wp, -wp);
glVertex3f(-wp, -wp, -wp);
glEnd();
}

```

Nakon što OpenGL-u zadamo vrhove, nad svakim vrhom obavlja se niz transformacija kako bi se utvrdio njegov konačan položaj na ekranu. Proces je prikazan na slici 5.12.



Slika 5.12: Proces obrade vrhova pri generiranju konačne slike

Pojednostavljeno obradu možemo promatrati kao proces koji se odvija u četiri koraka. U prvom koraku nad vrhovima zadanim u prostoru scene primjenje se transformacija *model-view* matricom, čime se dobivaju koordinate u sustavu oka (tj. promatrača). Potom se primjenjuje projekcijska matrica, kako bi se dobole *clip*-koordinate. Ova transformacija ujedno definira i *volumen pogleda*, pa se svi objekti izvan tog volumena odsijecaju. Slijedi dijeljenje koordinata s homogenim parametrom kako bi se točke vratile u 3D radni prostor i preslikale na interval $[-1, 1]$; ovako dobivene koordinate zovemo normalizirane koordinate uređaja (engl. *normalized device coordinates*). Konačno, točke se primjenom *viewport*-transformacije prevode u koordinate prozora. U ovom koraku nacrtana se slika može još rastegnuti ili smanjiti kako bi stala u dio prozora odabran za njezin prikaz. Detaljnije ćemo se svakim od ovih koraka pozabaviti u nastavku.

Matrice koje se koriste za prva dva koraka OpenGL čuva zasebno, i omogućava manipuliranje svakom od njih nizom naredbi koje djeluju nad matricama. Kako je OpenGL stroj stanja, ta se činjenica koristi kako bi se smanjio broj parametara koji se predaju naredbama za manipuliranje s matricama. Naime, te naredbe ne primaju parametar koji bi rekao nad kojom se matricom djeluje; umjesto toga, OpenGL koristi pojam *odabrane matrice*, i sve matrične operacije djeluju nad tom matricom. Da bismo odabrali *model-view* matricu, potrebno je zadati naredbu:

```
glMatrixMode(GL_MODELVIEW);
```

Sve matrične naredbe koje zadamo nakon tog poziva modificirat će matricu *model-view*. Da bismo odabrali *projekcijsku* matricu, potrebno je zadati naredbu:

```
glMatrixMode(GL_PROJECTION);
```

To pak znači da će tipična konceptualna struktura OpenGL programa biti kako slijedi.

```
void crtanje() {
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity(); // ponisti trenutne transformacije
    // dalje podesi projekcijsku matricu
    // ...
    // definiraj viewport ...
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity(); // ponisti trenutne transformacije
    // dalje podesi model-view matricu
    // ...
    nacrtajObjekteUScenu();
}
```

U stvarnosti, ovaj kod će biti raspodijeljen između više metoda. Podešavanje projekcijske matrice tipično će zajedno s definiranjem *viewport-a* biti smješteno u metodu *reshape(...)* jer ćemo tu trebati trenutne dimenzije prozora u kojem prikazujemo sliku, a promjene će se događati samo kada se mijenjaju dimenzije prozora. Podešavanje *model-view* matrice bit će pak smješteno u metodu *display()* jer se od trenutka do trenutka mogu mijenjati pozicije promatrača, smjer gledanja, položaji objekata u sceni i slično, pa tu matricu treba podešavati svaki puta kada se kreće u crtanjenu sliku.

Pogledajmo sada korake malo detaljnije.

5.5.1 Korak 1. Transformacije modela i pogleda

Nad koordinatama vrhova obavljuju se transformacije modela te transformacija pogleda. Transformacije modela su transformacije koje primjenjujemo kako bismo objekte pozicionirali na željeno mjesto u prostoru scene. Npr. pretpostavimo da trebamo nacrtati dva žičana modela kocke. Jedna kocka duljine stranice 10 mora se nalaziti u ishodištu. Druga kocka duljine stranice 5 mora biti zarotirana za 30° i pomaknuta za 10 u smjeru osi *x*. Jedna mogućnost jest ručno izračunati koordinate vrhova obaju točaka, i potom zadati niz *glVertex3f(...)* naredbi koje će OpenGL-u proslijediti te vrhove na crtanjenu. Bolja mogućnost jest iskoristiti našu postojeću funkciju *kocka(...)* koja uvijek crta kocku u ishodištu i podesiti OpenGL tako da vrhove koje zada ta funkcija transformira na željeni položaj (primjenom 3D transformacija). Za potrebe primjera pretpostavimo čak i da nemamo onako generičku funkciju *kocka(w)*; koja može crtati kocku proizvoljne duljine stranice, već da imamo samo funkciju *kocka1()*; koja crta kocku duljine stranice 1. Implementacija te funkcije ovom općenitijom je trivijalna, i prikazana je u nastavku.

```
void kocka1() {
    kocka(1.0f);
}
```

Tražena dva objekta u scenu možemo dodati sljedećim isječkom koda.

```
void renderScene() {
    glColor3f(1.0f, 0.2f, 0.2f);
    glPushMatrix();
    glScalef(10.0f, 10.0f, 10.0f);
    kocka1();
    glPopMatrix();

    glColor3f(0.0f, 0.2f, 1.0f);
    glPushMatrix();
    glTranslatef(10.0f, 0.0f, 0.0f);
```

```

    glRotatef(30.0f, 0.0f, 0.0f, 1.0f);
    glScalef(5.0f, 5.0f, 5.0f);
    kocka1();
    glPopMatrix();
}

```

Metoda se sastoji od dva bloka koda. prvi blok nakon definiranja nijanse crvene boje na stog pohranjuje trenutnu *model-view* matricu (koja je u ovom trenutku još uvijek jednaka *view-matrici*). Naredbe koje slijede dio su definiranja *model*-matrice, no te se transformacije odmah primjenjuju na cjelokupnu *model-view* matricu. Matrica se modificira tako da se pomnoži matricom skaliranja s faktorom 10. Slijedi poziv funkcije *kocka1()*; koja stvara niz vrhova koji odgovaraju jediničnoj kocki. Međutim, te vrhove množi trenutna *model-view* matrica što rezultira $10 \times$ većom kockom od jedinične. Nakon toga, sa stoga se vraća originalna *model-view* matrica i prvi blok je gotov.

Crtanje druge kocke zahtjeva malo više posla. Postupak započinje definiranjem nijanse plave boje koja će se koristiti za crtanje, te pohranom trenutne *model-view* matrice na stog. Naš je zadatak sada transformirati kocku duljine stranice 1 u kocku duljine stranice 5 koja je još i zarotirana te pomaknuta po *x*-osi za 10. Za to trebamo tri naredbe: najprije naredbom *glScalef (...)* obavljamo skaliranje kocke. Potom primjenjujemo naredbu *glRotatef (...)* koja za 30° rotira kocku oko osi *z* (središte kocke je još uvijek u ishodištu pa je ovo korektno). Konačno, nakon što smo točke zarotirali, naredbom *glTranslatef (...)* ih pomičemo na konačnu poziciju – za deset po osi *x*. I tek tada pozivamo funkciju *kocka1()*; koja generira niz vrhova koji odgovaraju kocki duljine stranice 1 – na njih redom djeluje skaliranje, rotacija i zadnje translacija čime smo efektivno dobili kocku kakvu smo izvorno i htjeli. Uočimo još da je redoslijed naredbi kako je napisan u izvornom kodu upravo obrnut od redoslijeda koji smo upravo naveli. Ali to je u redu. Sjetimo se da zadnje definirana transformacija na točke djeluje prva, zbog toga što trenutnu transformacijsku matricu množi s desna.

U ovom trenutku trebali bismo već razlikovati dva tipa koordinata. Objekti su tipično zadani *koordinatama u prostoru objekta*. Tipično, ishodište tog koordinatnog sustava nalazi se u centru objekta a koordinate po svih komponentama idu od -1 do 1. Primjer je naša jedinična kocka koja je omeđena ravninama paralelnim s *xy*-, *xz*- i *yz*-ravninama i s udaljenošću 0.5 do ishodišta. *model*-matricom koordinate tako zadanih objekata preslikavamo u *koordinate scene*; termin koji se u engleskom koristi jest *world-coordinates*. Zadatak *model*-matrice jest objekt pozicionirati na njegov stvarni položaj u sceni.

view-matrica zadužena je za transformaciju pogleda i tipično se definira prije poziva metode *renderScene()* – u metodi *display()*. Pojam *model-view*-matrica označava kompoziciju ovih dvaju matrica, i u stvarnosti, OpenGL i radi samo s jednom – konačnom – matricom. Definiranje *view*-matrice obavlja se u metodi *display()*, kako je prikazano u nastavku.

```

void display() {
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, -20.0f);
    renderScene();
    glutSwapBuffers();
}

```

Nakon čišćenja pozadine, kao trenutna se matrica učitava matrica identiteta. Važno je pri tome uočiti da je u trenutku poziva metode *display()* odabran rad s matricom *model-view*. Naime, pred kraj metode *reshape(...)* poziva se *glMatrixMode(GL_MODELVIEW)*; i nakon toga više u kodu nigdje ne mijenjamo odabranu matricu.

Transformacijom pogleda koju sada moramo napraviti trebamo definirati gdje se nalazi oko promatrača, te u kamo je pogled usmjeren. Inicijalno, OpenGL stavlja promatrača u ishodište koordinatnog sustava scene, okreće ga tako da gleda u smjeru *-z* osi te smjer "gore" definira tako da se podudara s pozitivnom *y*-osi. Ako tu ostavimo promatrača, on će se naći usred prve kocke koju smo crtali, a to ne želimo. Stoga ćemo promatrača pomaknuti u točku (0,0,20), tako da bude iznad obje kocke. Ovime je definiran koordinatni sustav oka (tj. promatrača) čije su osi kolinearne s odgovarajućim osima koordinatnog sustava scene, a ishodište je translatirano za (0,0,20). U tom sustavu, sve *x* i

y koordinate objekata ostaju očuvane, a z koordinate manje su za 20 – i time je upravo definirana potrebna transformacija pogleda, koju obavlja naredba `glTranslatef(0.0f, 0.0f, -20.0f);`.

Problem možemo riješiti i formalno. Naime, želimo da promatrač bude u točki $(0, 0, 20)$ te da gleda prema $-z$ osi. To znači da je očište $(0, 0, 20)$ a gledište, primjerice, $(0, 0, 0)$. Dodatno, *view-up* vektor je $(0, 1, 0)$, odnosno "gore" treba biti u smjeru porasta y -osi. Provedemo li postupak transformacije pogleda uz ove podatke, konačna matrica koju ćemo dobiti mora odgovarati upravo matrici translacije za -20 po osi z – provjerite.

Umjesto ručnog izračunavanja matrice transformacije pogleda, ili uporabe niza naredbi kojima opisujemo željenu translaciju i rotacije (koje nam trebaju za transformaciju pogleda), u biblioteci GLU definirana je pomoćna naredba koju možemo iskoristiti za sve navedene izračune. Radi se o naredbi `gluLookAt` čiji je prototip dan u nastavku.

```
void gluLookAt(
    GLdouble eyex, GLdouble eyey, GLdouble eyez,
    GLdouble centerx, GLdouble centery, GLdouble centerz,
    GLdouble vupx, GLdouble vupy, GLdouble vupz);
```

Metoda `gluLookAt` prima očište, gledište i *view-up* vektor (za sve se zadaju svaka od koordinata zasebno), i temeljem toga izračunava matricu kojom množi trenutnu *model-view* matricu. Uporabom metode `gluLookAt` metodu `display` mogli smo napisati ovako:

```
void display() {
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0.0f, 0.0f, 20.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);
    renderScene();
    glutSwapBuffers();
}
```

Uporabom `gluLookAt` možemo vrlo jednostavno zadati proivoljnu lokaciju očišta, gledišta te *view-up* vektor, što znači da sami ne trebamo raditi kompleksan izračun matrica potrebnih za definiranje transformacije pogleda – ova će naredba to učiti za nas.

5.5.2 Korak 2. Projekcija

OpenGL kroz postojeće naredbe direktno podržava dvije vrste projekcija: perspektivnu i paralelnu. Iako je složenija, perspektivna se projekcija češće koristi, pa ćemo krenuti od nje.

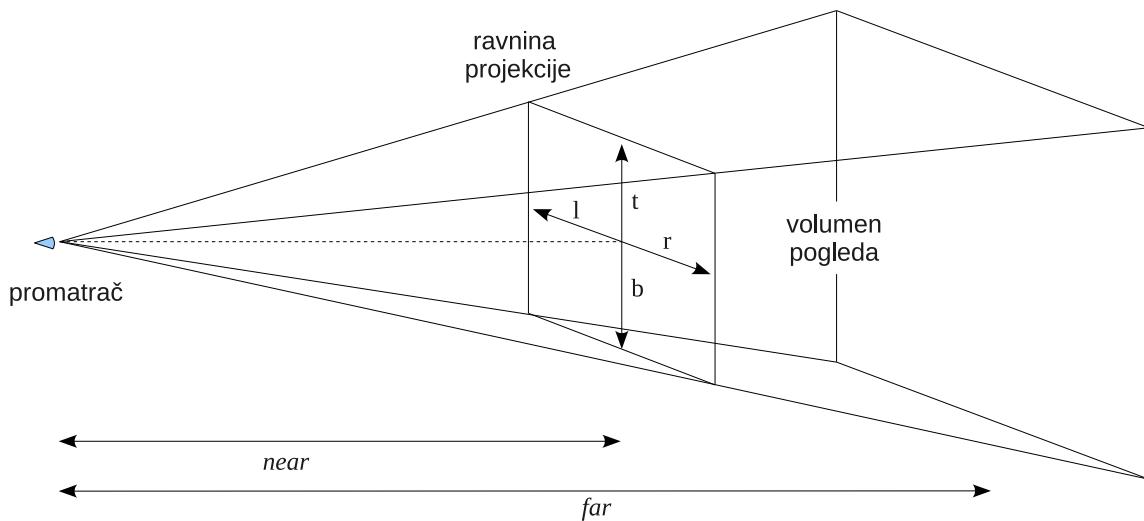
Podrška za perspektivnu projekciju

Ugrađena naredba kojom se stvara matrica perspektivne projekcije je `glFrustum`. Njezin prototip prikazan je u nastavku.

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

Naredba definira ravninu projekcije koja je smještena ispred promatrača na udaljenosti `near` (mora biti pozitivan broj). Ravnina projekcije koja se stvara okomita je na poveznici očište-gledište (slika 5.13). Mjereno od probodišta ravnine projekcije i pravca očište-gledište, stvara se pravokutno područje na kojem će se stvoriti slika. To se područje od probodišta proteže `left` u lijevo, `right` u desno, `bottom` prema dolje i `top` prema gore. Sve što bi palo u ravni projekcije ali izvan tog područja bit će odsiječeno. Konačno, na udaljenosti `far` od promatrača (pri čemu mora biti `far > near`) stvara se nova ravnina. Spojnice očište i rubovi područja u ravni projekcije probadaju i tu drugu ravninu i time zatvaraju volumen pogleda – dio prostora koji je sprijeda ogradien ravninom projekcije, straga ravninom na udaljenosti `far` te uokolo navedenim spojnicama.

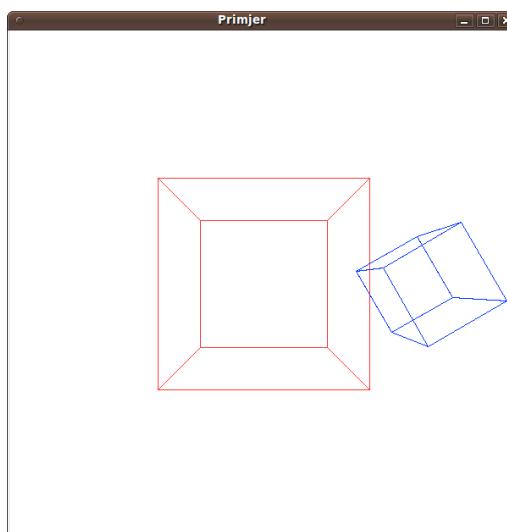
Uporabom ove naredbe možemo sada riješiti dio metode `reshape()`.

Slika 5.13: Model koji koristi naredbu *glFrustum*

```
void reshape(int width, int height) {
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1.2f, 1.2f, -1.2f, 1.2f, 1.5f, 30.0f);
    glMatrixMode(GL_MODELVIEW);
    glViewport(0, 0, (GLsizei)width, (GLsizei)height);
}
```

Ulaskom u metodu `reshape()`; najprije odabiremo matricu projekcije. Potom je resetiramo na jediničnu matricu, i zatim pozivom metode `glFrustum(...)`; definiramo površinu u ravnini projekcije koja je od promatrača udaljena 1.5. Ta se ravnina od probodišta sa spojnicom očište-gledište proteže 1.2 u svim smjerovima (prisjetimo se, promatrač je u točki $z = 20$ na osi z , i gleda prema ishodištu). Do prednje stranice veće kocke je udaljen za 15 a do njezine stražnje stranice za 25 (ta kocka ima stranice duljine 10). Stoga je kao *far* vrijednost odabранo 30, tako smo sigurni da je čitav objekt unutar volumena pogleda, i da će, shodno tome, biti i prikazan. Slika kojom rezultira ovaj program prikazana je na slici 5.14.

Biblioteka GLU nudi nam još jedan način definiranja matrice perspektivne projekcije – naredbu `gluPerspective`. Prototip naredbe prikazan je u nastavku.



Slika 5.14: Dvije kocke

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

Naredba definira ravninu projekcije koja je smještena ispred promatrača na udaljenosti $near$ (mora biti pozitivan broj). Ravnina projekcije koja se stvara okomita je na poveznicu očište-gledište (slika 5.15). Od probodišta ravnine projekcije tom spojnicom pravokutno područje proteže jednako i gore i dolje za $near \cdot \tan(\frac{fovy}{2})$, pri čemu je $fovy$ (engl. field-of-view) kut u stupnjevima. Ukupna visina tada je određena izrazom $h = 2 \cdot near \cdot \tan(\frac{fovy}{2})$. Širina područja ne zadaje se direktno, već se određuje iz zadanog omjera širine i visine (što je dano parametrom $aspect$): $w = aspect \cdot h$. Konačno, volumen pogleda definiran je još i stražnjom ravninom koja se od promatrača nalazi na udaljenosti far .

Podrška za paralelnu projekciju

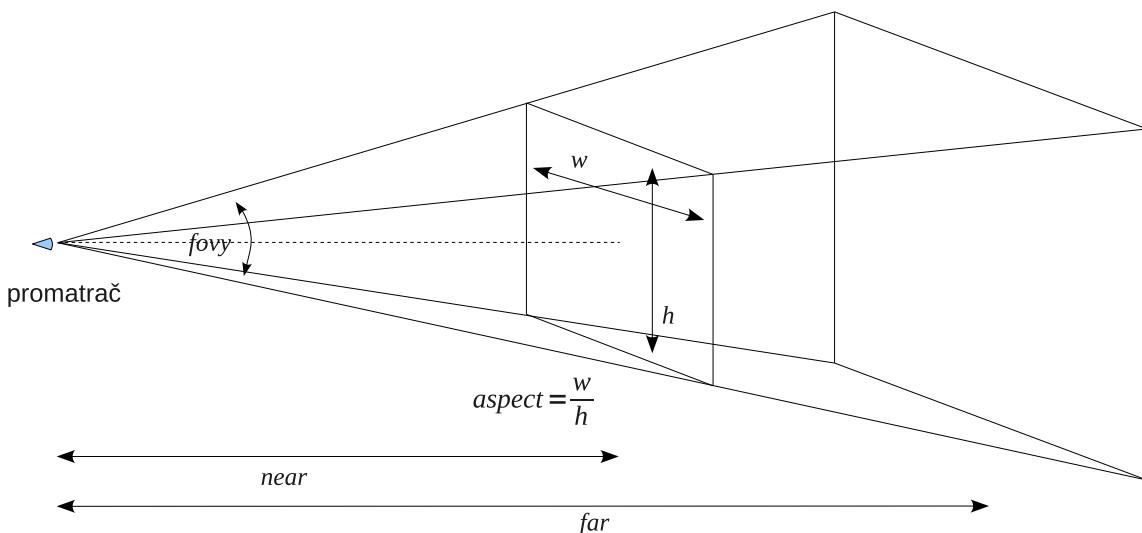
Ugrađena naredba kojom se stvara matrica paralelne projekcije je `glOrtho`. Njezin prototip prikazan je u nastavku.

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

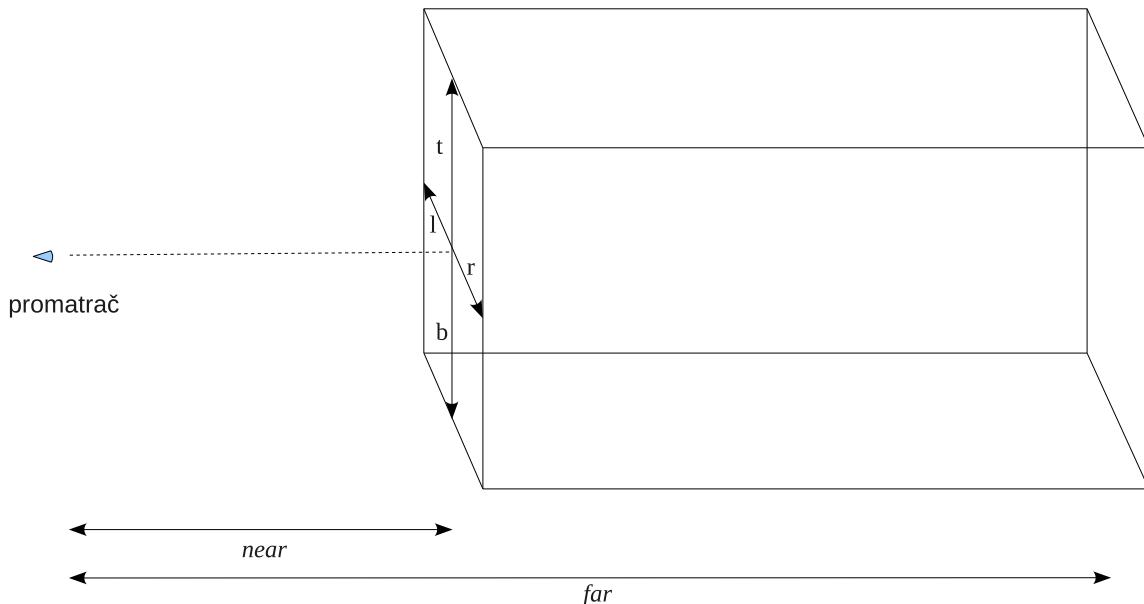
Naredba definira ravninu projekcije koja je smještena ispred promatrača na udaljenosti $near$ (mora biti pozitivan broj). Ravnina projekcije koja se stvara okomita je na poveznicu očište-gledište (slika 5.16). Mjereno od probodišta ravnine projekcije i pravca očište-gledište, stvara se pravokutno područje na kojem će se stvoriti slika. To se područje od probodišta proteže $left$ u lijevo, $right$ u desno, $bottom$ prema dolje i top prema gore. Sve što bi palo u ravninu projekcije ali izvan tog područja bit će odsijećeno. Konačno, na udaljenosti far od promatrača (pri čemu mora biti $far > near$) stvara se nova ravnina. Spojnice rubova područja u ravnini projekcije paralelne sa z -osi probadaju i tu drugu ravninu i time zatvaraju volumen pogleda koji je u ovom slučaju kvadar.

5.5.3 Korak 3. Normalizacija koordinata

Nakon što su točke projicirane u ravninu projekcije, postupak normalizacije koordinata pretvara *clip*-koordinate x_c , y_c i z_c koordinate u raspon $[-1, 1]$. Nakon projekcije točke su još uvijek u homogenom prostoru. Konceptualno, to znači najprije sve komponente točke podijeliti homogenim parametrom h_c , čime se koordinate vraćaju u radni prostor. Potom se svaka od koordinata linearno mapira na interval $[-1, 1]$. Primjerice, ako je korištena naredba `glFrustum`, prednja ravnina nalazi se na udaljenosti $near$ od promatrača, i proteže lijevo, desno, dolje i gore za $left$, $right$, $bottom$ i top . Pogledajmo najprije što se događa s x_c -koordinatom. Ona se iz intervala $[l, r]$ preslikava u $[-1, 1]$. Stoga možemo pisati:



Slika 5.15: Model koji koristi naredba `gluPerspective`

Slika 5.16: Model koji koristi naredbu *glOrtho*

$$\begin{aligned}
 x_{nd} &= \frac{x_c - l}{r - l} \cdot (1 - (-1)) - 1 \\
 &= \frac{x_c - l}{r - l} \cdot 2 - 1 \\
 &= \frac{2x_c - 2l}{r - l} - \frac{r - l}{r - l} \\
 &= \frac{2x_c}{r - l} - \frac{r + l}{r - l}
 \end{aligned}$$

Analogno imamo izraz za dobivanje normalizirane y_{nd} koordinate.

$$y_{nd} = \frac{2y_c}{t - b} - \frac{t + b}{t - b}$$

Problem koji možda ovdje nije odmah očit jest u našoj prepostavci – ako smo koristili naredbu *glFrustum*, tada normalizaciju radimo prema gornjim izrazima. Međutim, u ovom trenutku OpenGL više ne zna kako smo konstruirali matricu projekcije: je li to bila naredba *glFrustum*, ili naredba *glProjection* ili smo pak mi sami zadali direktno matricu element po element. Stoga nas ovakav pristup neće nigdje dovesti. OpenGL u ovom koraku radi samo jednu stvar: točke iz homogenog prostora pretvara u točke radnog prostora dijeljenjem s homogenim parametrom. Nakon toga očekuje se da su koordinate normalizirane. Da bi to bilo moguće, projekcijska matrica mora biti prikladno konstruirana tako da zadovolji taj zahtjev. To pak znači da će naredba koju koristimo za stvaranje projekcijske matrice morati o tome voditi računa. Sekcija 5.5.5 opisuje na koji naredba *glFrustum* gradi projekcijsku matricu, dok istu stvar za naredbu *glOrtho* sadrži sekcija 5.5.6. Nakon ovog koraka, sve točke koje imaju bilo koju koordinatu veću od 1 ili manju od -1 se odbacuju – takve točke predstavljaju točke koje su izvan volumena pogleda.

Zapamtimo: *clip*-koordinate kojima rezultira primjena projekcijske matrice iz prethodnog koraka su takve da povratkom u radni prostor (dijeljenjem s homogenim parametrom) rezultiraju normaliziranim koordinatama uređaja. Normalizirane koordinate uređaja imaju sve komponente u intervalu $[-1, 1]$.

5.5.4 Korak 4. *viewport transformacija*

Posljednji korak u crtanju scene jest odrediti gdje će se ta scena iscrtati u prozoru, odnosno uporaba transformacija koja točke iz normaliziranih koordinata uređaja preslikava u ekranske koordinate. Kako

bi bilo jasnije o čemu se radi, zamislite na tren da je kompletna slika nakon projekcije već nacrtana na pravokutnom području koje određuje bliža ravnina (engl. *near-plane*), tj. ravnina koja je od promatrača udaljena za *near*. Tu sliku sada treba negdje prekopirati na površinu prozora. I tu možemo birati – primjerice, hoćemo li sliku iskopirati preko čitave površine prozora, ili ćemo je nacrtati u gornjoj lijevoj četvrtini prozora, ili možda u donjoj desnoj četvrtini? Pravokutno područje unutar prozora u koje ćemo iskopirati sliku (uz eventualno potrebno rastezanje ili sažimanje) zove se *viewport*. U OpenGL-u *viewport* se definira naredbom `glViewport(...)`, čiji je prototip prikazan u nastavku.

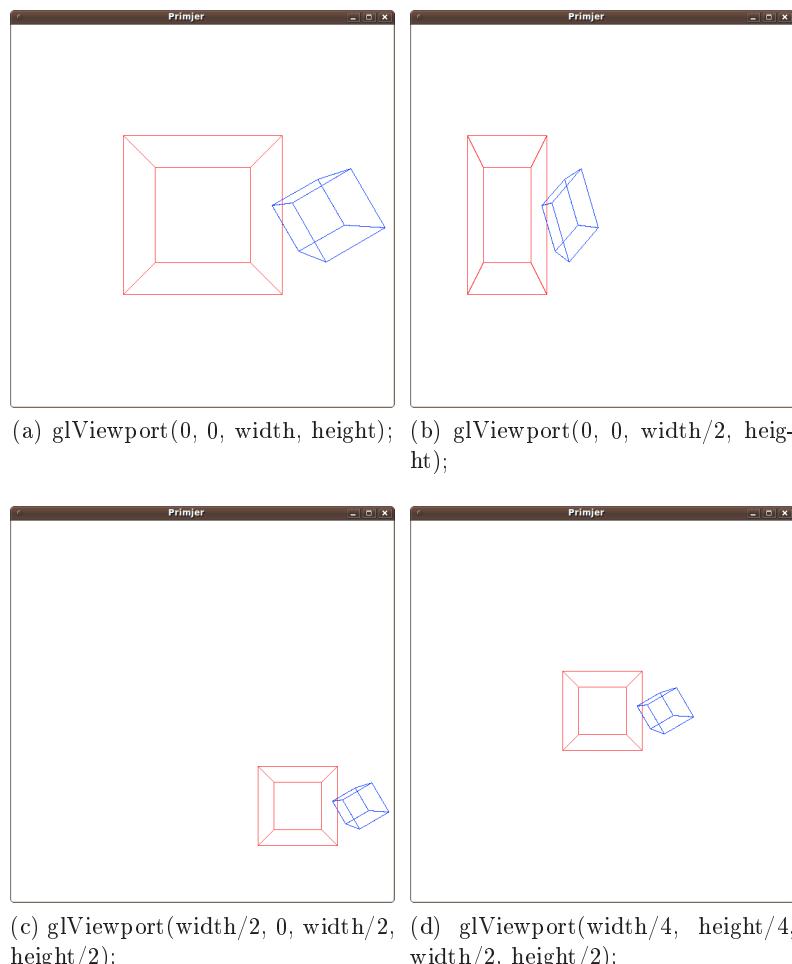
```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

Pri tome su *x* i *y* koordinate relativne unutar prozora. *y* = 0 pri tome predstavlja dno prozora. Ovu naredbu tipično pozivamo unutar metode *reshape* jer se tada mijenjaju i dimenzije prozora, pa je potrebno prilagoditi i *viewport*. Primjerice, da bismo sliku preslikali na čitavu površinu prozora, pozvat ćemo naredbu:

```
glViewport(0, 0, width, height);
```

Detaljniji primjer pokazuje slika 5.17. Tako je na slici 5.17a prikazan rezultat kada se kao *viewport* odabere čitava površina ekrana. Na primjeru prikazanom na slici 5.17b, kao *viewport* je odabrana lijeva polovica prozora, pa je čitava slika komprimirana po *x*-osi tako da stane u taj dio. Na primjeru prikazanom na slici 5.17c, kao *viewport* je odabrana donja desna četvrtina prozora, pa je čitava slika komprimirana i po *x*-osi i po *y*-osi. Konačno, na primjeru prikazanom na slici 5.17d, *viewport* je širok i visok kao pola prozora, i smješten je za četvrtinu širine prozora od lijevog ruba prozora te za četvrtinu visine prozora od donjeg ruba prozora.

Transformacije koje se ovdje primjenjuju prikazane su u nastavku. *x_w* i *y_w* su koordinate u prozoru, *x_{nd}* i *y_{nd}* su normalizirane koordinate točke iz prethodnog koraka a *x*, *y*, *width* i *height* su parametri



Slika 5.17: Utjecaj odabranog *viewport*-a na konačni prikaz slike

naredbe `glViewport(...)`.

$$x_w = (x_{nd} + 1) \frac{w}{2} + x, \quad y_w = (y_{nd} + 1) \frac{h}{2} + y.$$

5.5.5 Izgradnja projekcijske matrice naredbe `glFrustum`

U ovoj podsekciji osvrnut ćemo se na prethodno postavljeni zahtjev da projekcijska matrica generira takve točke koje dijeljenjem homogenim parametrom odmah postaju normalizirane. Ovo ćemo pogledati na modelu zadavanja projekcije kakav koristi naredba `glFrustum`, a prikazan je na slici 5.13. Argumente naredbe `glFrustum` kraće ćemo pisati l, r, b, t, n, f . Oznake x_e, y_e i z_e koristit ćemo za koordinate točke u sustavu promatrača (dakle, nakon transformacije pogleda). Oznake x_p, y_p i z_p koristit ćemo za koordinate točke u ravnini projekcije (koordinate odsijecanja, tj. *clip*-koordinate). Konačno, oznake x_{nd}, y_{nd} i z_{nd} koristit ćemo za normalizirane koordinate. Pa krenimo redom.

Nakon transformacije pogleda pretpostavka je da je ishodište koordinatnog sustava podudarno s promatračem, te da promatrač gleda u smjeru $-z$ osi. Međutim, parametri n i f zadaju se kao udaljenosti od promatrača, a z -koordinate dobivenih projiciranih točaka također želimo koristiti kao mjeru udaljenosti točke do promatrača. Ovo implicira koordinatni sustav u kojem udaljavanjem u smjeru pogleda z koordinate rastu, a ne padaju, pa ćemo u formulama po potrebi ubaciti promjenu predznaka. Ako je promatrač u ishodištu a ravnina projekcije na udaljenosti n od njega, točka (x_e, y_e, z_e) perspektivno se transformira prema izrazima:

$$x_p = \frac{n \cdot x_e}{-z_e}, \quad y_p = \frac{n \cdot y_e}{-z_e}.$$

Kako x_p i y_p treba dijeliti s $-z_e$, odabrat ćemo da je homogeni parametar w_p upravo jednak $-z_e$ pa točke sada nećemo dijeliti, već će se to dogoditi u trenutku kada krenemo vraćati točke u radni prostor. Međutim, ostavimo za sada formule takve kako su napisane. Normirane koordinate dobit ćemo preslikavanjem x_p na interval $[l, r]$ i y_p na interval $[b, t]$. Već smo izveli izraze koji to opisuju:

$$x_{nd} = \frac{2x_p}{r-l} - \frac{r+l}{r-l}, \quad y_{nd} = \frac{2y_p}{t-b} - \frac{t+b}{t-b}.$$

Uvrštavanjem prethodnih izraza za x_p i y_p u ove dobije se:

$$x_{nd} = \frac{\frac{2 \cdot n \cdot x_e}{-z_e} + \frac{r+l}{r-l} \cdot z_e}{-z_e}, \quad y_{nd} = \frac{\frac{2 \cdot n \cdot y_e}{-z_e} + \frac{t+b}{t-b} \cdot z_e}{-z_e}.$$

clip-koordinate x_c i y_c definirane su kao brojnih razlomaka prethodnih izraza:

$$x_c = \frac{2 \cdot n \cdot x_e}{r-l} + \frac{r+l}{r-l} \cdot z_e, \quad y_c = \frac{2 \cdot n \cdot y_e}{t-b} + \frac{t+b}{t-b} \cdot z_e.$$

Kako je $h_c = h_p = -z_e$, povratkom iz *clip*-koordinata u radni prostor računamo $\frac{x_c}{h_c}$ što je upravo x_{nd} , te $\frac{y_c}{h_c}$ što je upravo y_{nd} . Ostalo je još izračunati z_c . Kod klasične perspektivne projekcije, pisali bismo $z_p = -n$. Međutim, z_p želimo iskoristiti za mjeru udaljenosti točke do promatrača, kako bismo kasnije mogli odbacivati točke koje su iza točaka koje smo već nacrtali (kada ćemo raditi uklanjanje skrivenih površina). Kod perspektivne projekcije znamo da $z_p = z_c$ ne ovisi o x_e i y_e . To znači da može ovisiti još samo o z_e (ako prepostavimo da točka T_e ima homogeni parametar jednak 1). Pretpostavimo da je ta ovisnost linearna, tj. da možemo uvesti neke dvije konstante A i B , i pisati $z_c = A \cdot z_e + B$. Normalizirana z_{nd} koordinata (kao i sve druge) dobije se dijeljenjem s homogenim parametrom h_c , a njega smo odabrali da je $-z_e$. Stoga možemo pisati:

$$z_{nd} = \frac{A \cdot z_e + B}{-z_e}.$$

Želimo da vrijednost normalizirane z_{nd} koordinate bude jednaka -1 ako je $z_e = -n$ (ako leži na bližoj ravnini), odnosno da vrijednost normalizirane z_{nd} koordinate bude jednaka $+1$ ako je $z_e = -f$ (ako leži na daljoj ravnini). Iz ovoga dobivamo sustav jednadžbi:

$$\begin{aligned}-1 &= \frac{A \cdot (-n) + B}{-(-n)} = \frac{-A \cdot n + B}{n}, \\ +1 &= \frac{A \cdot (-f) + B}{-(-f)} = \frac{-A \cdot f + B}{f}.\end{aligned}$$

čije je rješenje:

$$A = -\frac{f+n}{f-n}, \quad B = \frac{-2 \cdot f \cdot n}{f-n}.$$

Uvrštavanjem u izraz za z_{nd} konačno dobivamo:

$$z_{nd} = \frac{-\frac{f+n}{f-n} \cdot z_e + \frac{-2 \cdot f \cdot n}{f-n}}{-z_e}.$$

Brojnik ovog izraza proglašit ćemo z_c . Time smo definirali izraze za sve komponente *clip*-koordinata u homogenom prostoru. Evo ih još jednom:

$$x_c = \frac{2 \cdot n \cdot x_e}{r-l} + \frac{r+l}{r-l} \cdot z_e, \quad y_c = \frac{2 \cdot n \cdot y_e}{t-b} + \frac{t+b}{t-b} \cdot z_e, \quad z_c = -\frac{f+n}{f-n} \cdot z_e + \frac{-2 \cdot f \cdot n}{f-n}, \quad h_c = -z_e.$$

Tada je projekcijska matrica π_{persp} koja točku T_e projicira u točku T_c na način $T_c = T_e \cdot \pi_{persp}$ definirana na sljedeći način.

$$\pi_{persp} = \begin{bmatrix} \frac{2 \cdot n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2 \cdot n}{t-b} & 0 & 0 \\ \frac{r+l}{r-l} & \frac{t+b}{t-b} & -\frac{f+n}{f-n} & -1 \\ 0 & 0 & \frac{-2 \cdot f \cdot n}{f-n} & 0 \end{bmatrix}$$

Prikazana matrica je upravo matrica koju prema dokumentaciji i gradi metoda `glFrustum`¹. Naravno, s obzirom da OpenGL množi matricu točkom, u stvarnosti koristi transponiranu matricu od ove koju smo izveli.

5.5.6 Izgradnja projekcijske matrice naredbe `glOrtho`

U ovoj podsekciji osvrnut ćemo se na prethodno postavljeni zahtjev da projekcijska matrica generira takve točke koje dijeljenjem homogenim parametrom odmah postaju normalizirane. Ovo ćemo pogledati na modelu zadavanja projekcije kakav koristi naredba `glOrtho`, a prikazan je na slici 5.16. Argumente naredbe `glOrtho` kraće ćemo pisati l, r, b, t, n, f . Oznake x_e, y_e i z_e koristit ćemo za koordinate točke u sustavu promatrača (dakle, nakon transformacije pogleda). Oznake x_p, y_p i z_p koristit ćemo za koordinate točke u ravni projekcije (koordinate odsijecanja, tj. *clip*-koordinate). Konačno, oznake x_{nd}, y_{nd} i z_{nd} koristit ćemo za normalizirane koordinate. Pa krenimo redom.

Nakon transformacije pogleda pretpostavka je da je ishodište koordinatnog sustava podudarno s promatračem, te da promatrač gleda u smjeru $-z$ osi. Međutim, parametri n i f zadaju se kao udaljenosti od promatrača, a z -koordinate dobivenih projiciranih točaka također želimo koristiti kao mjeru udaljenosti točke do promatrača. Ovo implicira koordinatni sustav u kojem udaljavanjem u smjeru pogleda z koordinate rastu, a ne padaju, pa ćemo u formulama po potrebi ubaciti promjenu predznaka. Ako je promatrač u ishodištu a ravnina projekcije na udaljenosti n od njega, točka (x_e, y_e, z_e) paralelno se transformira prema izrazima:

$$x_p = x_e, \quad y_p = y_e, \quad z_p = -z_e.$$

Krenimo u normiranje ovih koordinata. x_p želimo preslikati iz raspona $[l, r]$ u $[-1, 1]$ a y_p želimo preslikati iz raspona $[b, t]$ u $[-1, 1]$. Izraze za ovo već smo izveli u prethodnoj sekciji. Uvažavanjem činjenice da je $x_p = x_e$ te $y_p = y_e$ dobivamo:

¹Dokumentacija s adresom <http://www.opengl.org/sdk/docs/man/xhtml/glFrustum.xml>.

$$x_{nd} = x_c = \frac{2 \cdot x_e}{r-l} - \frac{r+l}{r-l}$$

$$y_{nd} = y_c = \frac{2 \cdot y_e}{t-b} - \frac{t+b}{t-b}$$

z_p želimo preslikati iz raspona $[n, f]$ u $[-1, 1]$, a to odgovara preslikavanju z_e iz raspona $[-n, -f]$ u $[-1, 1]$. Trivijalno je pokazati da se to postiže izrazom:

$$z_{nd} = z_c = \frac{-2 \cdot z_e}{f-n} - \frac{f+n}{f-n}.$$

Homogeni parametar h_c postaviti ćemo na 1. Time je u potpunosti definirana i projekcijska matrica π_{par} .

$$\pi_{par} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{-2}{f-n} & 0 \\ -\frac{r+l}{r-l} & -\frac{t+b}{t-b} & -\frac{f+n}{f-n} & 1 \end{bmatrix}$$

Poglavlje 6

Krivulje

6.1 Uvod

Pojam krivulje promatrati ćemo u najširem obliku: krivulja je niz točaka (uz dodatne uvjete koji pobliže opisuju specifični tip krivulje). U ovom dijelu teksta pozabaviti ćemo se načinima zadavanja krivulja, klasifikacijama krivulja te poželjnim svojstvima krivulja.

6.1.1 Načini zadavanja krivulja

Krivulje obično zadajemo analitički – formulom (ili formulama). Tako imamo tri osnovne kategorije zapisa krivulje.

- *Eksplicitnom jednadžbom* $y = f(x)$. Problemi koji se pri tome javljaju svakom su poznati. Primjerice, jednadžba pravca u eksplicitnom obliku glasila je $y = ax + b$. Međutim, ovaj oblik zapisivanja imao je i ozbiljnih problema; najbanalniji jest nemogućnost prikaza pravca paralelnog s y -osi. Drugi primjer je jednadžba kružnice, koja se uobičajeno zapisuje u obliku: $y = \pm\sqrt{R^2 - x^2}$ – ne iz razonode, već jednostavno iz činjenice da se eksplicitnim oblikom ne mogu prikazati funkcije s višestrukim vrijednostima. Kod kružnice je to još nekako prolazilo dodavanjem znaka \pm ispred korijena i prešutnim prihvaćanjem da tako zapravo radimo s dvije krivulje a ne s jednom.
- *Implicitnom jednadžbom* $f(x, y) = 0$. Ovakav oblik jednadžbe može prikazati višestruke vrijednosti, no ne može dati djelomičan prikaz. Npr. implicitni oblik jednadžbe kružnice je $x^2 + y^2 = R^2$. Ovime su obuhvaćene sve točke koje pripadaju kružnici, no sada riječ *sve* počinje predstavljati problem. Ponekad želimo prikazati samo npr. četvrt kružnice, što na ovaj način ne možemo specificirati.
- *Parametarskim zapisom*. Pri tome se uvodi jedan ili više parametara, te se sve koordinate opisuju kao eksplicitne funkcije tih parametara. Npr. za kružnicu možemo pisati ovako: parametar je t a koordinate su zadane izrazima $x = \cos(t)$, $y = \sin(t)$ uz $0 \leq t \leq 2\pi$. Ovaj oblik također nudi i mogućnost zadavanja samo određenog dijela krivulje. Tako primjerice, da bismo definirali samo prvu četvrtinu kružnice, parametar t umjesto na interval $0 \leq t \leq 2\pi$ treba ograničiti na interval $0 \leq t \leq \frac{\pi}{2}$.

Neovisno o analitičkom prikazu krivulje, krivulje možemo zadavati prema sljedećim kriterijima:

- tako da prolaze kroz zadane točke,
- tako da im definiramo vrijednosti prve derivacije u pojedinim točkama (tangente) te
- tako da im definiramo vrijednosti viših derivacija.

Pri tome se mogu ravnopravno koristiti i kombinacije navedenih kriterija; npr. možemo tražiti da krivulja prolazi kroz n zadanih točaka te da tangenta u prvoj točki bude pod kutom $\frac{\pi}{4}$. Prilikom ovakvog zadavanja treba imati u vidu da ponekad nije jednostavno dobiti sve potrebne parametre krivulje koja će ispunjavati ovakve miješane kriterije.

6.1.2 Klasifikacija krivulja i poželjna svojstva

Krivulje možemo klasificirati prema različitim kriterijima. Najčešće su klasifikacije sljedeće:

- periodičke \leftrightarrow neperiodičke,
- racionalne \leftrightarrow neracionalne,
- otvorene \leftrightarrow zatvorene te
- interpolacijske \leftrightarrow aproksimacijske.

U nastavku nabrojimo još i svojstva koja želimo kod krivulja.

- Mogućnost prikaza višestrukih vrijednosti (za jedan x više mogućih y vrijednosti).
- Nezavisnost od koordinatnog sustava.
- Svojstvo lokalnog nadzora (promjena jedne točke uzrokuje promjenu oblika krivulje samo u okolini te točke).
- Smanjenje varijacije (zbog visokog stupnja polinoma krivulje ponekad se umjesto glatke krivulje pojavljuje nepoželjno istitravanje).
- Red neprekinutosti što veći (više o ovome u nastavku).
- Mogućnost opisa osnovnih geometrijskih oblika poput kružnice, elipse i sl.

6.1.3 Svojstvo neprekinutosti krivulja

Pojam *red neprekinutosti* dolazi iz matematičke analize i odnosi se na funkcije. Kako se krivulje zadaju također kao funkcije, treba pogledati na što se odnose redovi neprekinutosti u ovom slučaju, odnosno koju imaju interpretaciju. Razmotrit ćemo dvije vrste neprekinutosti: C -kontinuitete i G -kontinuitete.

C -kontinuiteti

- C^0 -kontinuitet (čita se "ce nula", ne "ce na nultu"). Zahtjeva se neprekinutost u koordinatama. Ovo je, funkcionalno gledano, zahtjev na funkciju da nema diskontinuitete. Npr. funkcija $abs(x)$ je funkcija C^0 -kontinuiteta.
- C^1 -kontinuitet. Zahtjeva se neprekinutost tangente u svim točkama (što je zapravo zahtjev na neprekinutost prve derivacije). Drugim riječima, ne smije biti šiljaka, već krivulja mora biti glatka. Tako funkcija $abs(x)$ nije funkcija C^1 -kontinuiteta, iako ima C^0 -kontinuitet. Razlog je skok u ishodištu.
- C^2 -kontinuitet. Zahtjeva se neprekinutost druge derivacije u svim točkama.
- C^3 -kontinuitet. Zahtjeva se neprekinutost treće derivacije u svim točkama. Može se opisati kao neprekinutost u izvijanju.

G -kontinuiteti

G -kontinuiteti postavljaju nešto blaže zahtjeve. Da bi vrijedio G -kontinuitet, zahtjeva se da derivacije budu proporcionalne (dok C -kontinuiteti zahtjevaju da budu iste).

Traži se da vrijedi $f'_1|_T = k \cdot f'_2|_T$.

6.2 Krivulje zadane parametarskim oblikom

U prethodnom podpoglavlju opisani su načini zapisivanja funkcija. U ovom podpoglavlju posvetit ćemo se najzanimljivijem obliku – parametarskom. Ovaj oblik specifičan je po tome što uvodi dodatne parametre zahvaljujući kojima omogućava opisivanje i "pravih" funkcija (u matematičkom smislu) i funkcija s višestrukim vrijednostima. Ideja je da se sve koordinate izraze kao funkcija tih novih parametara. Radimo li s jednim parametrom, njega ćemo tipično označavati oznakom t . U tom se slučaju ovisnost koordinata o parametru t može izraziti funkcijски kao:

$$x = f(t), \quad y = g(t).$$

Funkcije f i g mogu biti različitih oblika. Neke od oblika upoznati ćemo u sljedećem podpoglavlju.

Parametarski oblik za svaku vrijednost parametra određuje po jednu točku krivulje. Zamislimo li da parametar t predstavlja vrijeme a krivulja putanju sićušne čestice, tada parametarski oblik za svaki trenutak određuje položaj čestice. Pustimo li vrijeme da teče kontinuirano, tada će se čestica gibati po zadanoj krivulji. Dakako, pri tome se moramo oslobođiti fizikalnih zakonitosti koje idu uz ovu sliku; naime, kako će izgledati ta putanja ovisi jedino o funkcijama koje određuju ovisnosti koordinata o parametru. Tako je moguće da se čestica u jednom trenutku giba ulijevo, pa već u sljedećem zakrene za 90° i nastavi gibanje, ili pak prekine gibanje na tom mjestu i nastavi ga na nekom sasvim drugom.

Mijenjamo li parametar t po svim dozvoljenim vrijednostima (za koje su funkcije definirane), dobit ćemo cijelu krivulju. No mijenjamo li taj parametar unutar manjeg intervala $[t_0, t_1]$, čestica će opisati samo segment krivulje.

6.2.1 Uporaba parametarskog oblika

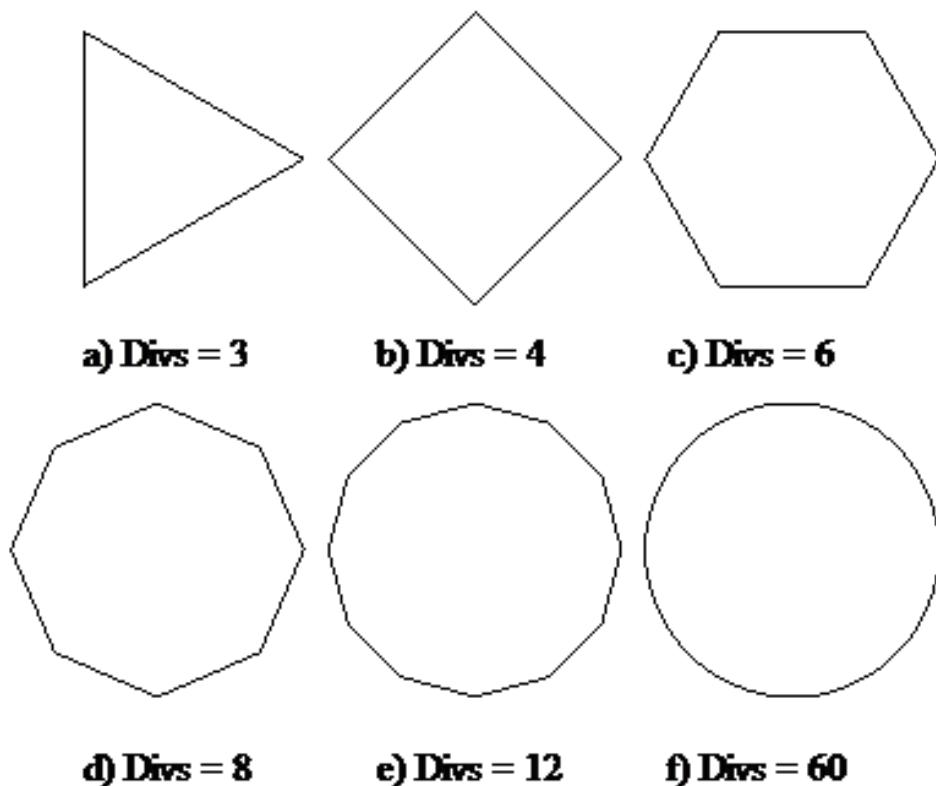
Krivulja je po definiciji skup točaka – beskonačan skup točaka. Parametarski oblik nam ovisno o parametru t daje za svaki t po jednu točku krivulje. Ako želimo nacrtati cijelu krivulju (dakle pokupiti sve točke), posao nećemo obaviti nikada. Krivulje obično prikazujemo na zaslonu, koji je rasterska jedinica. Zaslon se sastoji od niza elemenata koji mogu biti ili upaljeni ili ugašeni. Elementi zaslona adresiraju se cijelobrojnom adresom. Proizlazi da zaslon ima konačno mnogo elemenata i krivulja će na zaslonu biti aproksimirana konačnim brojem elemenata. No kako ćemo odrediti te elemente? Jedan od načina je određivanje konačnog broja točaka krivulje, te njihovo povezivanje linijskim segmentima. Za primjer ćemo uzeti prikaz kružnice. Kružnica se u parametarskom obliku može napisati:

$$x = R \cdot \cos(2\pi t) + centerX$$

$$y = R \cdot \sin(2\pi t) + centerY.$$

Sve točke kružnice pokupit ćemo ako parametar t mijenjamo po intervalu $[0, 1]$. Središte kružnice nalazi se u točki ($CenterX$, $CenterY$) a radijus kružnice je R . Prema navedenom receptu, da bismo nacrtali krivulju (kružnicu), odredit ćemo D_{i+1} točaka kružnice te susjedne točke spojiti linijama. Postavlja se pitanje koliko točaka treba odrediti? Pokušajmo to utvrditi eksperimentom. Na slici 6.1 prikazani su rezultati za različite vrijednosti broja točaka.

Iz slike je jasno vidljivo kada je prikaz bolji; što više točaka odredimo računski, prikaz na zaslonu je uvjerljiviji. Što nas opet vodi na zaključak da treba uzeti beskonačno finu podjelu. Nasreću, to i nije istina. Neka je kružnica radijusa 100 elemenata. I pretpostavimo da smo izračunali upravo onoliko točaka koliko je potrebno da bi svaka izračunata točka imala i susjednu koja je isto izračunata (drugim riječima, linije kojima spajamo te točke dugačke su točno jedan element – svaka spojnica degenerirala je u točku). U tom slučaju finijom podjelom nećemo ništa postići osim da na zaslonu više puta osvjetlimo iste točke. Koliko smo točaka osvjetlili? Pa gruba računica kaže da smo osvjetlili cijeli opseg kružnice, dakle $2R\pi = 628$ elemenata. U praksi ćemo, međutim, ipak ići na malo više točaka. Evo obrazloženja. Mijenjamo li parametar t od 0 do 0.25, opisat ćemo prvu četvrtinu kružnice. Pogledajmo kako naše funkcije raspodjeljuju točke u toj četvrtini. Prvih 45° prolazimo s t između 0 i 0.125. Drugih 45° prolazimo s t između 0.125 i 0.25. Oba ova segmenta imaju isti broj izračunatih točaka. Da smo umjesto kružnice crtali elipsu dosta izduženu po x -osi, tada bi u prvih 45° segment bio puno duži od segmenta u drugih 45° , a oba bi dobila isti broj točaka. Ovo vodi na zaključak da funkcije, s iznimkom



Slika 6.1: Utjecaj broja točaka na prikaz kružnice

posebnih slučajeva, već inherentno neravnomjerno raspoređuju točke te se može dogoditi da na jednom segmentu imamo puno izračunatih točaka a na drugom malo. Na segmentu na kojem ima malo točaka jasnije bi se vidjelo spajanje linijama a to nije poželjno. Stoga je potrebno napraviti već u startu gušću podjelu da se ovi efekti učine zanemarivim.

6.2.2 Primjer crtanja kružnice

U nastavku ćemo dati implementaciju funkcije koja crta kružnicu na zaslonu, oslanjajući se na parametarski zapis krivulje. Funkciji se predaju koordinate centra, radijus kružnice te željeni broj točaka koje će se računati.

```

typedef struct {
    double x;
    double y;
} Point2D;

void circle(Point2D center, double radius, int divs) {
    double t;
    Point2D p;

    if (divs < 2) return;

    glBegin(GL_LINE_STRIP);
    for (int n=0; n<=divs; n++) {
        t = 2.0*PI/divs*n;
        p.x = radius*cos(t)+center.x;
        p.y = radius*sin(t)+center.y;
        glVertex2f(p.x, p.y);
    }
    glEnd();
}

```

Ovdje primijenjen postupak za izračunavanje točaka kružnice temelji se direktno na jednadžbi kružnice (parametarskom obliku), te kao takav ne spada u grupu optimalnih. Za iscrtavanje kružnice postoje i brži postupci, a jedan od njih je i Bresenhamov postupak za kružnice.

6.2.3 Primjer crtanja elipse

Prethodni primjer može se vrlo jednostavno modificirati tako da umjesto kružnice crta elipsu. Elipsa za razliku od kružnice ima dva radijusa – jedan određuje širinu elipse a drugi visinu. Funkcija koju ćemo napisati prihvatiće koordinate pravokutnika kojemu treba upisati elipsu.

```
void ellipse(Point2D p1, Point2D p2, int divs) {
    double t, rx, ry;
    Point2D p, center;

    if(divs<2) return;
    if(p1.x>p2.x) {
        double tmp = p1.x;
        p1.x = p2.x;
        p2.x = tmp;
    }
    if(p1.y>p2.y) {
        double tmp = p1.y;
        p1.y = p2.y;
        p2.y = tmp;
    }

    rx = (p2.x-p1.x)/2.0;
    ry = (p2.y-p1.y)/2.0;
    center.x = p1.x + rx;
    center.y = p1.y + ry;

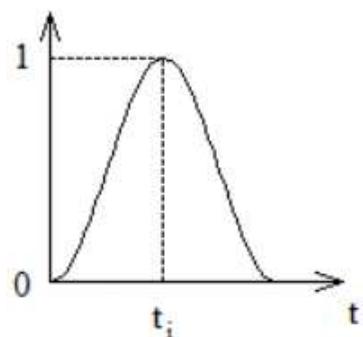
    glBegin(GL_LINE_STRIP);
    for(int n=0; n<=divs; n++) {
        t = 2.0*PI/divs*n;
        p.x = rx*cos(t)+center.x;
        p.y = ry*sin(t)+center.y;
        glVertex2f(p.x, p.y);
    }
    glEnd();
}
```

6.2.4 Konstrukcija krivulje s obzirom na zadane točke

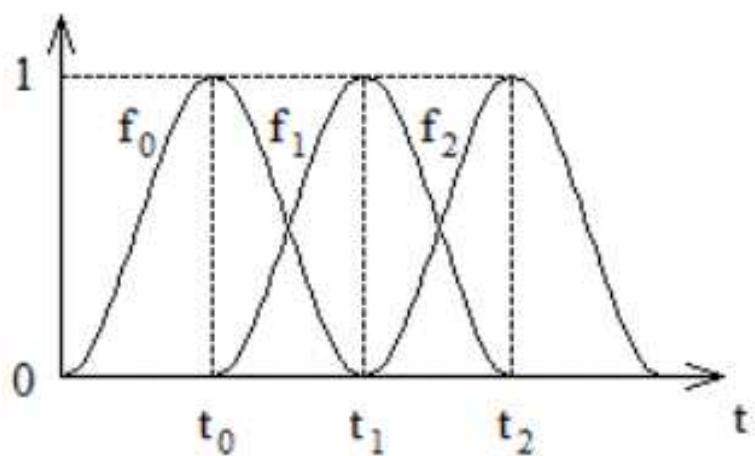
Jedan od češćih problema koji se veže uz krivulje jest sljedeći: "zadano je $n+1$ točka; potrebno je povući krivulju s obzirom na te točke". Što točno znači "s obzirom", ovisi o potrebama; to može značiti "kroz točke" pa tada govorimo o *interpolaciji*, ili može značiti "tako da krivulja prolazi negdje u blizini tih točaka" pa govorimo o *aproksimaciji*. Također, mogu se postavljati različiti zahtjevi i na oblik krivulje. Primjerice, možemo tražiti da krivulja bude glatka i slično.

Jedno od mogućih rješenja je uporaba težinskih funkcija. Ideja je sljedeća. Želimo konstruirati krivulju čiji se početak dobije za vrijednost parametra $t = t_s$ a kraj za $t = t_e$. Pri tome krivulja prolazi kraj točke T_i za vrijednost parametra $t = t_i$. Iskoristiti ćemo težinske funkcije zvonolikog oblika koje imaju maksimum za $t = t_i$, kao na slici 6.2. Težinskih funkcija bit će onoliko koliko je zadano točaka. Svaka točka obliku krivulje doprinositi će svojim koordinatama pomnoženim s težinskom funkcijom. Jednadžbu krivulje tada ćemo zapisati kao sumu svake točke pomnožene težinskom funkcijom. Težinske funkcije označavat ćeemo oznakom f_i , pri čemu indeks i govori kojoj točki pripada ta težinska funkcija. Tako jednadžbu krivulje možemo zapisati:

$$T_K = \sum_{i=0}^n f_i(t) \cdot T_i$$



Slika 6.2: Primjer težinske funkcije zvonolikog oblika



Slika 6.3: Težinske funkcije uz tri točke krivulje

gdje je T_K točka krivulje a T_i i -ta zadana točka (kako je zadano $n + 1$ točaka, indeks ide od 0 do n). Uzmimo kao primjer da su zadane tri točke. Težinske funkcije moguće bi izgledati kao na slici 6.3. Ako krivulju crtamo od $t = t_0$ do $t = t_2$, što možemo reći o krivulji gledajući ove težinske funkcije?

Za $t = t_0$ je $f_0 = 1, f_1 = 0, f_2 = 0$, pa je:

$$T_K(t = t_0) = \sum_{i=0}^2 f_i(t_0) \cdot T_i = f_0(t_0) \cdot T_0 + f_1(t_0) \cdot T_1 + f_2(t_0) \cdot T_2 = 1 \cdot T_0 + 0 \cdot T_1 + 0 \cdot T_2 = T_0.$$

Za $t = t_1$ je $f_0 = 0, f_1 = 1, f_2 = 0$, pa je:

$$T_K(t = t_1) = \sum_{i=0}^2 f_i(t_1) \cdot T_i = f_0(t_1) \cdot T_0 + f_1(t_1) \cdot T_1 + f_2(t_1) \cdot T_2 = 0 \cdot T_0 + 1 \cdot T_1 + 0 \cdot T_2 = T_1.$$

Za $t = t_2$ je $f_0 = 0, f_1 = 0, f_2 = 1$, pa je:

$$T_K(t = t_2) = \sum_{i=0}^2 f_i(t_2) \cdot T_i = f_0(t_2) \cdot T_0 + f_1(t_2) \cdot T_1 + f_2(t_2) \cdot T_2 = 0 \cdot T_0 + 0 \cdot T_1 + 1 \cdot T_2 = T_2.$$

Krivulja dakle prolazi kroz sve zadane točke, a međutočke se aproksimiraju. Možemo li pogledom na sliku 6.3. reći kako krivulja ovisi o promjeni pojedine točke? Na slici je jasno vidljivo da istovremeno na položaj točke djeluju najviše dvije težinske funkcije dok su ostale nula; naime, na intervalu $[t_0, t_1]$ djeluju f_0 i f_1 dok je f_2 jednaka nuli. Na intervalu $[t_1, t_2]$ djeluju f_1 i f_2 dok je f_0 jednaka nuli. To znači da oblik krivulje na intervalu $[t_0, t_1]$ određuju isključivo točke T_0 i T_1 dok točku T_2 možemo pomicati kamo god želimo bez da utječemo na ovaj segment krivulje. Oblik krivulje na intervalu $[t_1, t_2]$ određuju pak točke T_1 i T_2 dok točku T_0 možemo pomicati kamo god želimo bez da utječemo na ovaj segment krivulje. Ovakvo svojstvo krivulje kada promjena položaja jedne točke utječe na promjenu oblika krivulju isključivo u okolini te točke naziva se *svojstvo lokalnog nadzora*. Neke krivulje nemaju ovo svojstvo, pa pomaknemo li jednu točku krivulje, mijenja se cijela krivulja. Svojstvo lokalnog nadzora poželjno je svojstvo.

Pogledajmo još malo sliku 6.3. Na slici su sve težinske funkcije jednake. Takve funkcije nazivamo *uniformnima*. Kod težinskih funkcija možemo mijenjati dva parametra: visinu težinske funkcije te širinu težinske funkcije. Pogledajmo kakvog utjecaja ima na oblik krivulje.

Ako mijenjamo visinu težinske funkcije, fizikalno to možemo protumačiti kao promjenu privlačenja dotične točke i krivulje. Što je težinska funkcija viša, to će odgovarajuća točka više utjecati na oblik krivulje; što je težinska funkcija niža, to će točka manje mijenjati oblik krivulje. Ovo nam omogućava da osim samih točaka odredimo i njihove "težine" te na temelju toga mijenjamo oblik krivulje.

Promjena širine težinske funkcije može se tumačiti kao promjena dosega privlačne sile između točke i krivulje. Što je težinska funkcija šira, to će odgovarajuća točka imati utjecaj na veći komad krivulje; što je težinska funkcija uža, to će točka utjecati na krivulju na manjem segmentu. Na slici 6.3 odabrane su težinske funkcije tako da svaka točka utječe na krivulju na segmentu od prethodnog susjeda točke pa do sljedećeg susjeda točke. Ovime smo postigli svojstvo lokalnog nadzora jer utjecaj jedne točke nije dopirao dalje od njezinih susjeda. Ako bismo širine težinskih funkcija udvostručili, tada bi utjecaj točke T_0 dopirao sve do točke T_2 ; a kako krivulju crtamo upravo od točke T_0 do točke T_2 , tada bi točka T_0 utjecala na cijelu krivulju. Time bismo izgubili svojstvo lokalnog nadzora.

Ako se dozvoljava da se pojedine težinske funkcije razlikuju po širini i po visini, tada za takve težinske funkcije kažemo da su *neuniformne*.

6.2.5 Ponavljanje

U ovom podpoglavlju upoznali smo se sa značajem parametarskog oblika, načinima crtanja funkcija zadanih na taj način, te idejom konstrukcije krivulje s obzirom na zadane točke, što smo ilustrirali uporabom težinskih funkcija. Kao primjer smo pokazali zvonolike težinske funkcije. Primjer nam

je poslužio i za upoznavanje s utjecajima pojedinih parametara samih težinskih funkcija na oblik krivulje. Situacija u praksi, što se tiče ovog dijela je vrlo šarolika. Ne samo da se koriste svakakvi oblici težinskih funkcija, nego se i izvode novi, ovisno o zahtjevima koji se postavljaju na krivulje. Kao primjer krivulja koje nude dosta slobode a koriste zvonolike funkcije navesti ću *NURBS* (neuniformni racionalni b-spline). Primjer često korištenih krivulja koje ne koriste zvonolike težinske funkcije su Bezierove krivulje.

U nastavku ćemo se upoznati s nekim od čestih oblika krivulja.

6.3 Bezierove krivulje

U računalnoj grafici jedan od čestih zadataka je provlačenje glatke krivulje između zadanog niza točaka (aproksimacija krivulje). Jedno od rješenja problema nude nam Bezierove krivulje. Do krivulja su nezavisno došli 1962. Bezier koji je radio za tvrtku Renault, te 1959. De Casteljau koji je radio za tvrtku Citroen. Da je riječ o istim krivuljama, 1970. godine dokazao je Robert Forest. Za matematički opis Bezierovih krivulja postoje dvije metode: *gibanje vrha sastavljenog otvorenog poligona* koja vodi na Bezierove težinske funkcije, te *parametarsko dijeljenje vektora* koja vodi na Bernsteinove težinske funkcije. Za praktičnu primjenu u računalima Bernsteinove težinske funkcije su daleko pogodnije. Isto tako ćemo razlikovati dva tipa Bezierovih krivulja: aproksimacijska i interpolacijska. Aproksimacijska Bezierova krivulja prolazi kroz početnu i završnu točku, dok kroz ostale točke ne prolazi. Zato se i kaže da aproksimira zadanu krivulju. Interpolacijska Bezierova krivulja prolazi kroz sve zadane točke; ona dakle interpolira krivulju po segmentima između zadanih točaka, a prolazi kroz zadane točke.

Aproksimacijsku Bezierovu krivulju lagano je odrediti, kao što ćemo vidjeti u nastavku. S druge strane, interpolacijsku Bezierovu krivulju dobivamo trikom. Naime, budući da jednostavno provlačimo aproksimacijsku krivulju, a zadane su nam točke kroz koje krivulja mora proći, te točke ćemo iskoristiti tako da izračunamo nove točke između kojih ćemo provući aproksimacijsku krivulju, uz uvjet da ta krivulja prolazi kroz stare zadane točke. Vidimo dakle da je osnovno što moramo naučiti – kako računati aproksimacijsku krivulju.

6.3.1 Aproksimacijska Bezierova krivulja

Prilikom rada s Bezierovim krivuljama te njihovog matematičkog tretmana koristit ćemo sljedeće označake.

- Točke - vrhovi poligona bit će označeni oznakom T_i , gdje je i indeks točke. Bezierova krivulja bit će zadana točkama T_0, T_1, \dots, T_n . Pri tome n označava stupanj krivulje. Tako će krivulja trećeg stupnja ($n = 3$) biti zadana s četiri točke: T_0, T_1, T_2 i T_3 .
- Oznakom \vec{r}_i označavat će se radij-vektori točaka vrhova poligona. Radij vektor je vektor koji spaja ishodište i zadanu točku T_i . Zbog toga su mu sve komponente jednake kao i kod same točke T_i . Dakle, radij-vektor koji spaja točku $(1 \ 2)$ je $(1 \ 2)$.
- Oznakom \vec{a}_i označavat će se vektori između točaka T_{i-1} i T_i ; kako točka T_0 nema prethodnika, vektor \vec{a}_0 definirat ćemo da je jednak radij-vektoru te točke, tj $\vec{a}_0 = \vec{r}_0$. Općenito se može zapisati:

$$\vec{a}_i = \begin{cases} T_i - T_{i-1}, & i > 0 \\ T_0 & i = 0 \end{cases} .$$

- Težinske funkcije bit će označavane oznakom $\psi_{i,n}$, pri čemu će kod Bezierovih težinskih funkcija umjesto ψ stajati f , a kod Bernsteinovih težinskih funkcija b . Pri tome i označava indeks funkcije i taj parametar će se mijenjati između 0 i n , dok n označava stupanj krivulje.

Bezierove težinske funkcije

Do ovih se funkcija dolazi metodom gibanja vrha sastavljenog otvorenog poligona, kao što je već spomenuto u uvodu. Sastavljeni otvoreni poligon označava poligon koji se dobije kada se svi vektori

\vec{a}_i izvornog poligona pomnože s određenim težinskim funkcijama i zatim zbroje. Kod nas će, dakako, vektori biti pomnoženi Bezierovim težinskim funkcijama. Slijedi da se može pisati:

$$\vec{p}(t) = \sum_{i=0}^n \vec{a}_i f_{i,n}(t)$$

pri čemu je $\vec{p}(t)$ radij-vektor točke koja pripada krivulji (vrh sastavljenog otvorenog poligona), a t je parametar kojim određujemo sve točke krivulje. Za $t = 0$ dobiva se početna točka krivulje a za $t = 1$ dobiva se završna točka krivulje. Sve ostale točke dobivaju se za $t \in (0, 1)$.

Do analitičkih izraza za funkcije $f_{i,n}$ dolazi se iz sljedećih zahtjeva.

1. Krivulja za $t = 0$ prolazi kroz prvu zadalu točku: $\vec{p}(0) = \vec{a}_0$, odakle slijedi da mora vrijediti $f_{0,n} = 1$ te $f_{i,n} = 0, i = 1, 2, \dots, n$.
2. Krivulja za $t = 1$ prolazi kroz zadnju zadalu točku: $\vec{p}(1) = \sum_{i=0}^n \vec{a}_i$, odakle slijedi da mora vrijediti $f_{i,n} = 1, i = 0, 1, \dots, n$.
3. U početnoj točki nagib krivulje jednak je nagibu vektora \vec{a}_1 , što povlači $f'_{1,n}(0) = 1$ te $f'_{i,n}(0) = 0, i \neq 1$ (ovo ćemo pokazati u nastavku).
4. U završnoj točki nagib krivulje jednak je nagibu vektora \vec{a}_n , što povlači $f'_{n,n}(1) = 1$ te $f'_{i,n}(1) = 0, i \neq n$.
5. Postavlja se zahtjev na druge derivacije u početnoj točki: $f''_{1,n}(0) \neq 0, f''_{2,n}(0) \neq 0$ te $f''_{i,n}(0) = 0, i \neq 1, 2$.
6. Postavlja se zahtjev na druge derivacije u završnoj točki: $f''_{n-1,n}(1) \neq 0, f''_{n,n}(1) \neq 0$ te $f''_{i,n}(1) = 0, i \neq n-1, n$.
7. Zahtjeva se simetričnost, odnosno traži se da redoslijed točaka ne utječe na izgled krivulje (zamjena početne i krajnje točke nema utjecaja), što vodi na zahtjev $f_{i,n}(t) = 1 - f_{n-i+1,n}(1-t), i = 1, 2, \dots, n$.

Težinske funkcije koje proizlaze iz ovih zahtjeva nazivaju se Bezierove težinske funkcije i zadane su izrazom:

$$f_{i,n}(t) = \frac{(-t)^i}{(i-1)!} \frac{d^{(i-1)} \Phi_n(t)}{dt^{(i-1)}}, \quad i = 1, 2, \dots, n$$

$$f_{0,n}(t) = 1$$

pri čemu je funkcija $\Phi_n(t)$ zadana jednadžbom:

$$\Phi_n(t) = \frac{1 - (1-t)^n}{-t}.$$

Zbog potrebe za deriviranjem i vrlo složenih izraza vidljivo je da ovakav zapis nije baš prikladan za uporabu u računalima. No ovi se izrazi mogu napisati u, za računala, puno prihvatljivijem obliku: kao rekurzivne funkcije.

$$f_{i,n}(t) = (1-t)f_{i,n-1}(t) + t \cdot f_{i-1,n-1}(t),$$

$$f_{0,0}(t) = 1, \quad f_{-1,k}(t) = 1, \quad f_{k+1,k}(t) = 0.$$

Pokažimo još kako smo došli do izraza iz točke 3 u zahtjevima na Bezierove težinske funkcije. Vektor \vec{a}_1 je razlika radij-vektora $\vec{a}_1 = \vec{r}_1 - \vec{r}_0$, te svojim komponentama određuje nagib $tg\delta = \frac{a_{1,y}}{a_{1,x}}$. Krivulja je zadana u parametarskom obliku sumom $\vec{p}(t) = \sum_{i=0}^n \vec{a}_i f_{i,n}(t)$. Ova jednadžba vrijedi i za svaku

komponentu zasebno. Nagib krivulje određen je derivacijom $\frac{dy}{dx}$, što se iz dane jednadžbe ne možemo dobiti direktno već malim trikom. Pomnožimo traženu derivaciju s prikladno napisanom jedinicom:

$$\begin{aligned}\frac{dy}{dx} &= \frac{dy}{dx} \cdot \frac{dt}{dt} = \frac{\frac{dy}{dt}}{\frac{dx}{dt}} \\ &= \frac{\frac{d}{dt} \left(\sum_{i=0}^n a_{i,y} f_{i,n}(t) \right)}{\frac{d}{dt} \left(\sum_{i=0}^n a_{i,x} f_{i,n}(t) \right)} \\ &= \frac{a_{0,y} f'_{0,n}(t) + a_{1,y} f'_{1,n}(t) + \cdots + a_{n,y} f'_{n,n}(t)}{a_{0,x} f'_{0,n}(t) + a_{1,x} f'_{1,n}(t) + \cdots + a_{n,x} f'_{n,n}(t)}.\end{aligned}$$

Zahtjev je da u početnoj točki ($t = 0$) nagib bude $\frac{a_{1,y}}{a_{1,x}}$, odakle odmah slijedi:

$$\frac{dy}{dx}(t = 0) = \frac{a_{0,y} f'_{0,n}(t) + a_{1,y} f'_{1,n}(t) + \cdots + a_{n,y} f'_{n,n}(t)}{a_{0,x} f'_{0,n}(t) + a_{1,x} f'_{1,n}(t) + \cdots + a_{n,x} f'_{n,n}(t)} = \frac{a_{1,y}}{a_{1,x}}$$

što je moguće samo ako su derivacije svih težinskih funkcija u toj točki jednake nuli, osim derivacije funkcije $f_{1,n}$ jer ona množi tražene komponente vektora \vec{a}_1 .

Slično se dobije i za zahtjev 4, gdje se traži jednakost nagiba u završnoj točki krivulje ($t = 1$), i vektora \vec{a}_n , ako se u gornju formulu uvrsti $t = 1$ i $\operatorname{tg}\delta = \frac{a_{n,y}}{a_{n,x}}$. Tada se izjednačavanjem dobije da derivacije svih težinskih funkcija osim $f_{n,n}$ moraju biti nula, a derivacija od $f_{n,n}$ mora biti 1.

Bernsteinove težinske funkcije

Do funkcija se dolazi na sljedeći način. Potrebno je odrediti točku krivulje za fiksni t . Npr. za $t = \frac{1}{4}$. Sve stranice izvornog poligona podijele se novom točkom koja odgovara zadanim parametru t (duljina cijele stranice je 1). Novodobivene točke spoje se spojnicama, i zatim se na spojnicama određuju nove točke opet zadane parametrom t . Postupak se ponavlja sve dok ne ostane samo jedna spojnica i na njoj ovako određena točka. Ta točka ujedno je i točka krivulje.

Postupak ćemo najbolje ilustrirati na primjeru. Neka su zadane točke prema slici 6.4. Na slici 6.4(a) točke su spojene u otvoreni poligon. Odredimo točku za $t = \frac{1}{4}$. Točka $b_{1,0}$ označava točku na četvrtini spojnici između točaka T_1 i T_0 . Točka $b_{1,1}$ je na četvrtini spojnici između točaka T_2 i T_1 . Točka $b_{1,2}$ je na četvrtini spojnici između točaka T_3 i T_2 . Točaka $b_{1,0}$, $b_{1,1}$ i $b_{1,2}$ spojene su uporabom dviju novih spojnika. Na slici 6.4(b) te su spojnice podijeljene tako da se točka $b_{2,0}$ nalazi na četvrtini puta između $b_{1,1}$ i $b_{1,0}$, a točka $b_{2,1}$ nalazi se na četvrtini puta između $b_{1,2}$ i $b_{1,1}$. Ove dvije točke opet su spojene spojnicom. Na slici 6.4(c) ta je spojnica podijeljena tako da se točka $b_{3,0}$ nalazi na četvrtini puta između $b_{2,1}$ i $b_{2,0}$. Time je postupak gotov jer više nemamo novih spojnika i točka $b_{3,0}$ predstavlja točku Bezierove krivulje za $t = \frac{1}{4}$.

Postupak možemo opisati na sljedeći način. Krenuli smo od točaka poligona:

$$\begin{aligned}b_{1,0} &= (T_1 - T_0) \cdot t + T_0, \\ b_{1,1} &= (T_2 - T_1) \cdot t + T_1, \\ b_{1,2} &= (T_3 - T_2) \cdot t + T_2.\end{aligned}$$

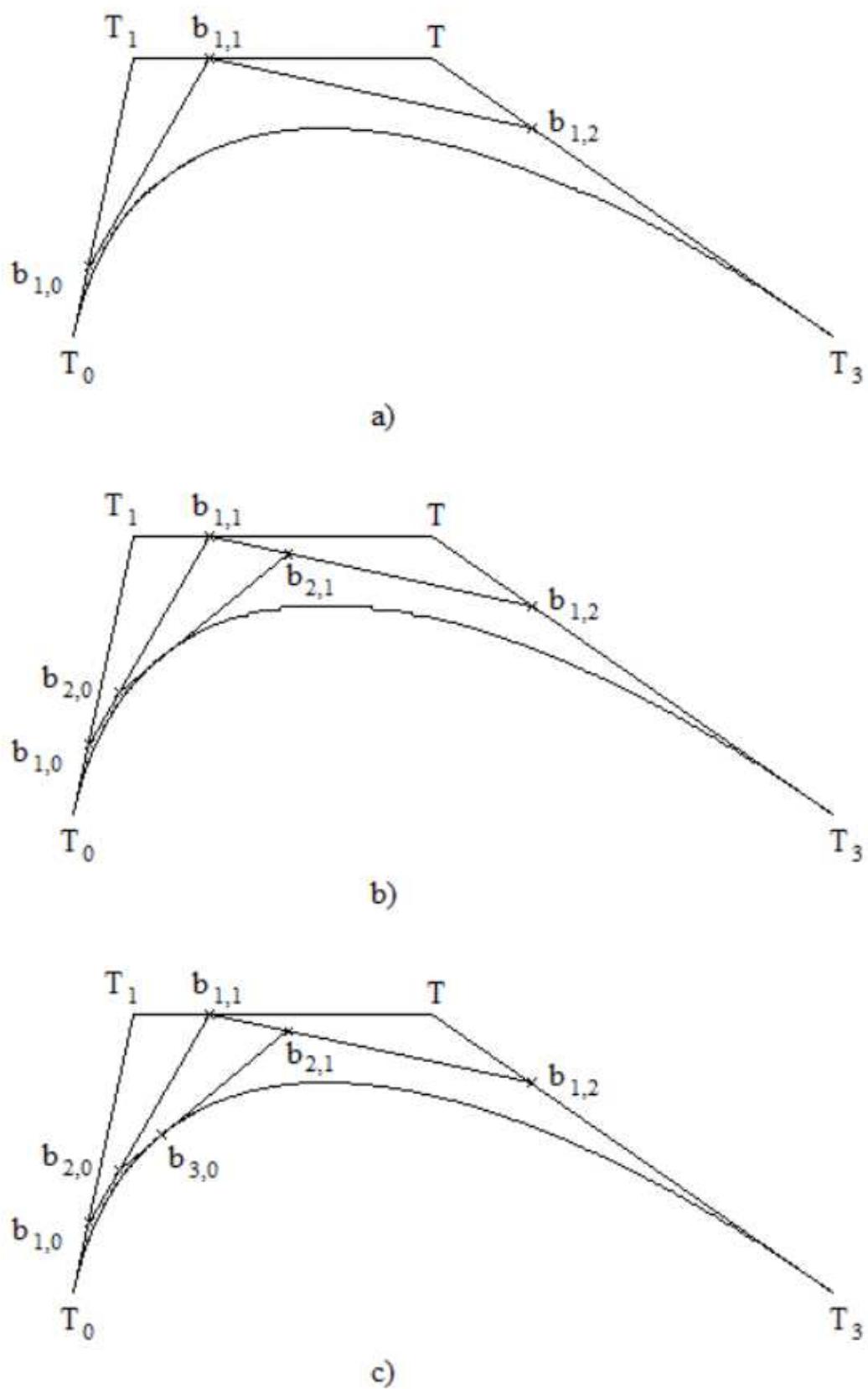
Zatim smo spojnice dijelili:

$$\begin{aligned}b_{2,0} &= (b_{1,1} - b_{1,0}) \cdot t + b_{1,0}, \\ b_{2,1} &= (b_{1,2} - b_{1,1}) \cdot t + b_{1,1}.\end{aligned}$$

I konačno smo i te spojnice podijelili:

$$b_{3,0} = (b_{2,1} - b_{2,0}) \cdot t + b_{2,0}.$$

Ako sada $b_{3,0}$ izrazimo samo preko točaka, dobit ćemo:



Slika 6.4: Konstrukcija Bezierove krivulje

$$b_{3,0} = (1-t)^3 \cdot T_0 + 3t(1-t)^2 \cdot T_1 + 3t^2(1-t) \cdot T_2 + t^3 \cdot T_3$$

ili da to napišemo korektno preko radij-vektora točaka:

$$b_{3,0} = (1-t)^3 \cdot \vec{r}_0 + 3t(1-t)^2 \cdot \vec{r}_1 + 3t^2(1-t) \cdot \vec{r}_2 + t^3 \cdot \vec{r}_3.$$

Ova točka pripada krivulji, pa konačno možemo pisati:

$$\vec{p}(t) = (1-t)^3 \cdot \vec{r}_0 + 3t(1-t)^2 \cdot \vec{r}_1 + 3t^2(1-t) \cdot \vec{r}_2 + t^3 \cdot \vec{r}_3.$$

Dobivena je formula koja opisuje sve točke Bezierove krivulje zadane preko četiri točke, dakle krivulje trećeg reda. Pogleda li se formula malo bolje, vidi se da faktori ispred radij-vektora neobično podsjećaju na binomnu formulu. Da ima istine u tome, govori nam i sljedeći zapis ovdje prikazanih težinskih funkcija poznat kao Bernsteinove težinske funkcije:

$$\vec{p}(t) = \sum_{i=0}^n \vec{r}_i \cdot b_{i,n}(t)$$

pri čemu su težinske funkcije dane relacijom:

$$b_{i,n}(t) = \binom{n}{i} \cdot t^i \cdot (1-t)^{n-i} = \frac{n!}{i!(n-i)!} \cdot t^i \cdot (1-t)^{n-i}.$$

Ovo je najjednostavniji oblik za uporabu u računalima, i mi ćemo ga koristiti u nastavku. Pojava faktorijela također ne komplicira stvari.

Veza između Bezierovih i Bernsteinovih težinskih funkcija

U uvodu smo već rekli da se i pomoću Bezierovih i pomoću Bernsteinovih težinskih funkcija opisuje ista krivulja; to je dokazao Robert Forest. Mi ćemo u nastavku samo dati tu vezu:

$$f_{i,n}(t) = \sum_{j=i}^n b_{j,n}(t).$$

Crtanje aproksimacijske krivulje pomoću Bernsteinovih težinskih funkcija

Kod koji crta Bezierovu krivulju struktrom će biti sličan onome za kružnicu ili elipsu. Metoda `draw_bezier` kao argumente će primiti polje točaka kontrolnog poligona, broj tih točaka te parametar `divs` koji govori koliko ćemo gusto "uzorkovati" Bezierovu krivulju kako bismo ostatak spajali linijskim segmentima. Metoda na početku stvori pomoćno polje faktora u koje će se pohraniti vrijednost: $\binom{n}{i}$ kako se ne bi nepotrebno računala za svaki uzorak. To popunjava metoda `compute_factors` i to u linearnoj složenosti.

```
void compute_factors(int n, int *factors) {
    int i, a=1;

    for (i = 1; i <= n+1; i++) {
        factors[i-1] = a;
        a = a * (n-i+1) / i;
    }
}

void draw_bezier(Point2D *points, int points_count, int divs) {
    Point2D p;
    int n = points_count - 1;
    int *factors = (int*) malloc(sizeof(int)*points_count);
    double t, b;

    compute_factors(n, factors);
```

```

glBegin(GL_LINE_STRIP);
for(int i = 0; i <= divs; i++) {
    t = 1.0 / divs * i;
    p.x = 0; p.y = 0;
    for(int j = 0; j <= n; j++) {
        if(j==0) {
            b = factors[j]*pow(1-t, n);
        } else if(j==n) {
            b = factors[j]*pow(t, n);
        } else {
            b = factors[j]*pow(t, j)*pow(1-t, n-j);
        }
        p.x += b * points[j].x;
        p.y += b * points[j].y;
    }
    glVertex2f(p.x, p.y);
}
glEnd();
free(factors);
}

```

Linearnu složenost kojom metoda `compute_factors` računa vrijednost funkcije *povrh* možemo jednostavno objasniti. Neka je zadan stupanj n . Tada su faktori redom:

$$\begin{aligned}
 factors[0] &= 1 \\
 factors[1] &= \frac{n}{1} = factors[0] \cdot \frac{n}{1} \\
 factors[2] &= \frac{n \cdot (n-1)}{1 \cdot 2} = factors[1] \cdot \frac{n-1}{2} \\
 factors[3] &= \frac{n \cdot (n-1) \cdot (n-2)}{1 \cdot 2 \cdot 3} = factors[2] \cdot \frac{n-2}{3} \\
 \dots \\
 factors[n] &= \frac{n \cdot (n-1) \cdot (n-2) \dots 1}{1 \cdot 2 \cdot 3 \dots n} = factors[n-1] \cdot \frac{n-(n-1)}{n}
 \end{aligned}$$

pa se svaki faktor može dobiti direktno poznavanjem samo prethodnog člana.

Svojstva aproksimacijskih Bezierovih krivulja

- Postoji konveksna ljska i krivulja je unutar ljske.
- Krivulja nema više valova od kontrolnog poligona (ili drugim riječima, ravnina kojom presiječemo krivulju nema više sjecišta s tom ravninom no što ih ima kontrolni poligon s tom ravninom).
- Krivulja nema svojstvo lokalnog nadzora. Naime, ukoliko pomaknemo samo jednu točku kojom smo zadali krivulju, cijela će krivulja promijeniti oblik. Poželjno bi bilo kada bi se krivulja promijenila samo u okolini pomaknute točke, no ovo kod Bezierovih krivulja ne vrijedi.
- Broj točaka u direktnoj je vezi sa stupnjem krivulje. Npr. krivulja četvrtog stupnja zadana je pomoću pet kontrolnih točaka.
- Neovisnost o afnim transformacijama (translacije, rotacije, skaliranja). Perspektivne transformacije nisu afine transformacije pa za njih ovo ne vrijedi.
- Simetričnost. Krivulja ima isti izgled ako zamijenimo redoslijed kontrolnih točaka (tako da točka koja je bila prva postane zadnja, točka koja je bila druga postane predzadnja, itd).

6.3.2 Interpolacijska Bezierova krivulja

Interpolacijska krivulja prolazi svim zadanim točkama. Ideja izračuna interpolacijskih krivulja već je opisan u uvodu, no neće škoditi da je ponovimo. Zadat ćemo točke (odnosno radij-vektore točaka) kroz koje želimo da krivulja prođe za neki parametar t . Zatim ćemo na temelju tih točaka izračunati kontrolne točke aproksimacijske krivulje, koja će proći kroz naše tražene točke.

Problem se općenito može zapisati ovako. Zadani su radij-vektori:

$$\vec{p}_0 = \vec{p}(t_0), \vec{p}_1 = \vec{p}(t_1), \dots, \vec{p}_i = \vec{p}(t_i), \dots, \vec{p}_n = \vec{p}(t_n).$$

Kako je krivulja zadana s $n + 1$ radij-vektorom, krivulja je n -tog stupnja. Za opis krivulje koristiti ćemo Bezierove težinske funkcije. Neka je kontrolni poligon aproksimacijske krivulje zadan vektorima $\vec{a}_0, \dots, \vec{a}_n$ (koje ne znamo). Tada je svaka točka aproksimacijske krivulje dana sumom:

$$\vec{p}(t) = \sum_{i=0}^n \vec{a}_i f_{i,n}(t).$$

Postavimo sada zahtjev da ta krivulja *mora* proći kroz zadane radij-vektore \vec{p}_i .

$$\vec{p}(t_i) = \sum_{j=0}^n \vec{a}_j f_{j,n}(t_i) = \vec{p}_i.$$

Uočimo da smo ovime definirali $(n + 1)$ jednadžbu, jer i ide od 0 do n . To pak možemo zapisati i matrično:

$$\begin{bmatrix} \vec{p}_0 \\ \vec{p}_1 \\ \vdots \\ \vec{p}_n \end{bmatrix} = \begin{bmatrix} f_{0,n}(t_0) & f_{1,n}(t_0) & \cdots & f_{n,n}(t_0) \\ f_{0,n}(t_1) & f_{1,n}(t_1) & \cdots & f_{n,n}(t_1) \\ \vdots & \vdots & \ddots & \vdots \\ f_{0,n}(t_n) & f_{1,n}(t_n) & \cdots & f_{n,n}(t_n) \end{bmatrix} \cdot \begin{bmatrix} \vec{a}_0 \\ \vec{a}_1 \\ \vdots \\ \vec{a}_n \end{bmatrix}$$

ili kraće:

$$\mathbf{P} = \mathbf{F} \cdot \mathbf{A} \quad \Rightarrow \quad \mathbf{A} = \mathbf{F}^{-1} \cdot \mathbf{P}.$$

Iz ovih jednadžbi potrebno je odrediti matricu \mathbf{A} . Nakon toga su nam poznati svi vektori \vec{a}_i te se iz njih može direktno crtati krivulja, ili se mogu izračunati radij-vektori točaka kontrolnog poligona aproksimacijske krivulje i zatim crtati krivulju pomoću Bernsteinovih težinskih funkcija.

Vrlo često se u praksi zadaju samo točke kroz koje želimo provući krivulju, ali se pri tome ne specificira za koju vrijednost parametra t krivulja mora proći kroz koju točku. Tada se za parametar može odabrat i vrlo jednostavan oblik:

$$t_i = \frac{i}{n}$$

pa se prethodni matrični račun pojednostavljuje:

$$\begin{bmatrix} \vec{p}_0 \\ \vec{p}_1 \\ \vdots \\ \vec{p}_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & f_{1,n}\left(\frac{1}{n}\right) & \cdots & f_{n,n}\left(\frac{1}{n}\right) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & f_{1,n}\left(\frac{n}{n}\right) & \cdots & f_{n,n}\left(\frac{n}{n}\right) \end{bmatrix} \cdot \begin{bmatrix} \vec{a}_0 \\ \vec{a}_1 \\ \vdots \\ \vec{a}_n \end{bmatrix}$$

Kao primjer možemo uzeti krivulju trećeg stupnja koju želimo provući kroz četiri točke: $\vec{p}_0, \vec{p}_1, \vec{p}_2$ i \vec{p}_3 . Budući da vrijednosti parametara nisu zadane, uzet ćemo parametre prema relaciji:

$$t_i = \frac{i}{n}$$

čime dobivamo:

$$\vec{p}_0 = p\left(\frac{0}{3}\right), \vec{p}_1 = p\left(\frac{1}{3}\right), \vec{p}_2 = p\left(\frac{2}{3}\right), \vec{p}_3 = p\left(\frac{3}{3}\right).$$

Bezierove težinske funkcije za krivulju trećeg reda glase:

$$\begin{aligned} f_{0,3}(t) &= 1 \\ f_{1,3}(t) &= 3t - 3t^2 + t^3 \\ f_{2,3}(t) &= 3t^2 - 2t^3 \\ f_{3,3}(t) &= t^3 \end{aligned}$$

Uvrstimo li ovo u matricu, dobivamo:

$$\begin{bmatrix} \vec{p}_0 \\ \vec{p}_1 \\ \vec{p}_2 \\ \vec{p}_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & \frac{19}{27} & \frac{7}{27} & \frac{1}{27} \\ 1 & \frac{26}{27} & \frac{20}{27} & \frac{8}{27} \\ 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \vec{a}_0 \\ \vec{a}_1 \\ \vec{a}_2 \\ \vec{a}_3 \end{bmatrix}$$

Slijedi da je:

$$\mathbf{A} = \frac{1}{18} \cdot \begin{bmatrix} 18 & 0 & 0 & 0 \\ -33 & 54 & -27 & 6 \\ 21 & -81 & 81 & -21 \\ -6 & 27 & -54 & 33 \end{bmatrix} \cdot \mathbf{P}$$

odnosno po komponentama:

$$\begin{bmatrix} \vec{a}_0 \\ \vec{a}_1 \\ \vec{a}_2 \\ \vec{a}_3 \end{bmatrix} = \frac{1}{18} \cdot \begin{bmatrix} 18 & 0 & 0 & 0 \\ -33 & 54 & -27 & 6 \\ 21 & -81 & 81 & -21 \\ -6 & 27 & -54 & 33 \end{bmatrix} \cdot \begin{bmatrix} \vec{p}_0 \\ \vec{p}_1 \\ \vec{p}_2 \\ \vec{p}_3 \end{bmatrix}$$

Sada kada smo izračunali tražene vektore, jednadžba aproksimacijske Bezierove krivulje (koja je interpolacijska s obzirom na početno zadane točke \vec{p}_i) glasi:

$$\vec{p}(t) = \sum_{i=0}^3 \vec{a}_i f_{i,3}(t) = 1 \cdot \vec{a}_0 + (3t - 3t^2 + t^3) \cdot \vec{a}_1 + (3t^2 - 2t^3) \cdot \vec{a}_2 + t^3 \cdot \vec{a}_3.$$

U ovom primjeru bile su zadane $n + 1$ točka. No interpolacijska se krivulja može provlačiti i na temelju drugih podataka. Za izračun aproksimacijske krivulje potreban nam je $n + 1$ uvjet. Neki od načina zadavanja su sljedeći:

- $n + 1$ poznata točka (kao u primjeru) te
- n poznatih točaka i poznata tangenta u nekoj od točaka (obično prva ili zadnja točka).

Krivulju je moguće računati i na temelju poznavanja viših derivacija i slično.

6.4 Prikaz krivulja pomoću razlomljenih funkcija

Jedan od načina zapisivanja krivulja koji do sada nismo spomenuli jest pomoću razlomljenih funkcija. Pri tome se krivulje opisuju parametarski uz jedan parametar t , u homogenom prostoru, a svaka koordinata točke polinomna funkcija parametra t . Kako ćemo krivulje opisivati općenito u $3D$ -prostoru, svaka točka T_K imat će svoje tri koordinate $T_{K,1}$ ili x , $T_{K,2}$ ili y i $T_{K,3}$ ili z u radnom prostoru, odnosno naziv T_{Kh} i četiri koordinate $T_{Kh,1}$, $T_{Kh,2}$, $T_{Kh,3}$ i $T_{Kh,h}$ u homogenom prostoru.

6.4.1 Prikaz krivulja pomoću kvadratnih razlomljenih funkcija

Kvadratne razlomljene funkcije omogućavaju nam jednostavan prikaz koničnih krivulja (krivulje koje nastaju kao presjecište stošca i ravnine, npr. kružnice, elipse i sl.). Kako je riječ o kvadratnim funkcijama, funkcije ovisnosti svih koordinata u homogenom prostoru bit će izražene kvadratnim polinomom, pa možemo pisati:

$$\begin{aligned} T_{Kh,1} &= a_1 \cdot t^2 + b_1 \cdot t + c_1 \\ T_{Kh,2} &= a_2 \cdot t^2 + b_2 \cdot t + c_2 \\ T_{Kh,3} &= a_3 \cdot t^2 + b_3 \cdot t + c_3 \\ T_{Kh,h} &= a \cdot t^2 + b \cdot t + c \end{aligned}$$

Povratkom u radni prostor dobiva se:

$$\begin{aligned} T_{K,1} &= \frac{a_1 \cdot t^2 + b_1 \cdot t + c_1}{a \cdot t^2 + b \cdot t + c} \\ T_{K,2} &= \frac{a_2 \cdot t^2 + b_2 \cdot t + c_2}{a \cdot t^2 + b \cdot t + c} \\ T_{K,3} &= \frac{a_3 \cdot t^2 + b_3 \cdot t + c_3}{a \cdot t^2 + b \cdot t + c} \end{aligned}$$

Razlog za naziv "razlomljene kvadratne" funkcije trebao bi biti jasan iz prethodnih jednadžbi. Zadržimo li se u homogenom prostoru, tada se jednadžbe po koordinatama mogu spojiti u jedan matrični zapis:

$$T_{Kh} = [T_{Kh,1} \ T_{Kh,2} \ T_{Kh,3} \ T_{Kh,h}] = [t^2 \ t \ 1] \cdot \begin{bmatrix} a_1 & a_2 & a_3 & a \\ b_1 & b_2 & b_3 & b \\ c_1 & c_2 & c_3 & c \end{bmatrix} = [t^2 \ t \ 1] \cdot \mathbf{K}.$$

Matrica \mathbf{K} naziva se *karakteristična matrica krivulje*. Za određivanje matrice \mathbf{K} dovoljno je poznavati tri točke kroz koje krivulja mora proći. Npr. neka prođe kroz točku T_A za $t = t_1 = 0$, kroz točku T_B za $t = t_2 = 0.5$ i kroz točku T_C za $t = t_3 = 1$. Tada vrijedi:

$$\begin{aligned} T_{Ah} &= [T_{Ah,1} \ T_{Ah,2} \ T_{Ah,3} \ T_{Ah,h}] = [t_1^2 \ t_1 \ 1] \cdot \mathbf{K} \\ T_{Bh} &= [T_{Bh,1} \ T_{Bh,2} \ T_{Bh,3} \ T_{Bh,h}] = [t_2^2 \ t_2 \ 1] \cdot \mathbf{K} \\ T_{Ch} &= [T_{Ch,1} \ T_{Ch,2} \ T_{Ch,3} \ T_{Ch,h}] = [t_3^2 \ t_3 \ 1] \cdot \mathbf{K} \end{aligned}$$

što možemo zapisati i matrično:

$$\begin{bmatrix} T_{Ah} \\ T_{Bh} \\ T_{Ch} \end{bmatrix} = \begin{bmatrix} T_{Ah,1} & T_{Ah,2} & T_{Ah,3} & Ah, h \\ T_{Bh,1} & T_{Bh,2} & T_{Bh,3} & Bh, h \\ T_{Ch,1} & T_{Ch,2} & T_{Ch,3} & Ch, h \end{bmatrix} = \begin{bmatrix} t_1^2 & t_1 & 1 \\ t_2^2 & t_2 & 1 \\ t_3^2 & t_3 & 1 \end{bmatrix} \cdot \mathbf{K}.$$

Da bismo odredili matricu \mathbf{K} , cijelu jednadžbu potrebno je pomnožiti s inverznom matricom koeficijenata i to s lijeve strane:

$$\mathbf{K} = \begin{bmatrix} t_1^2 & t_1 & 1 \\ t_2^2 & t_2 & 1 \\ t_3^2 & t_3 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} T_{Ah} \\ T_{Bh} \\ T_{Ch} \end{bmatrix} = \begin{bmatrix} t_1^2 & t_1 & 1 \\ t_2^2 & t_2 & 1 \\ t_3^2 & t_3 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} T_{Ah,1} & T_{Ah,2} & T_{Ah,3} & Ah, h \\ T_{Bh,1} & T_{Bh,2} & T_{Bh,3} & Bh, h \\ T_{Ch,1} & T_{Ch,2} & T_{Ch,3} & Ch, h \end{bmatrix}.$$

Uvrstimo li zadane t -ove u izraz, dobiva se:

$$\mathbf{K} = \begin{bmatrix} 0^2 & 0 & 1 \\ 0.5^2 & 0.5 & 1 \\ 1^2 & 1 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} T_{Ah} \\ T_{Bh} \\ T_{Ch} \end{bmatrix} = \begin{bmatrix} 2 & -4 & 2 \\ -3 & 4 & -1 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} T_{Ah} \\ T_{Bh} \\ T_{Ch} \end{bmatrix}.$$

Zadamo li sada i točke T_A , T_B i T_C , matrica \mathbf{K} može se odrediti u potpunosti.

6.4.2 Parametarske derivacije u homogenom prostoru

Svim koordinatama definirali smo funkciju ovisnost o parametru t . U ovom slučaju ta je funkcija ovisnost definirana kvadratnim polinomom. No to znači da se te funkcije daju i derivirati. Pogledajmo kako bi izgledala prva derivacija po parametru t . Funkcijske ovisnosti koordinata definirane su relacijama:

$$\begin{aligned} T_{Kh,1} &= a_1 \cdot t^2 + b_1 \cdot t + c_1 \\ T_{Kh,2} &= a_2 \cdot t^2 + b_2 \cdot t + c_2 \\ T_{Kh,3} &= a_3 \cdot t^2 + b_3 \cdot t + c_3 \\ T_{Kh,h} &= a \cdot t^2 + b \cdot t + c \end{aligned}$$

Deriviranjem svake relacije po t dobiva se:

$$\begin{aligned}\frac{dT_{Kh,1}}{dt} &= a_1 \cdot 2t + b_1 \\ \frac{dT_{Kh,2}}{dt} &= a_2 \cdot 2t + b_2 \\ \frac{dT_{Kh,3}}{dt} &= a_3 \cdot 2t + b_3 \\ \frac{dT_{Kh,h}}{dt} &= a \cdot 2t + b\end{aligned}$$

Označimo li derivaciju u točki T_{Kh} po parametru t oznakom T'_{Kh} tada možemo derivaciju raspisati po komponentama u matričnom obliku:

$$T'_{Kh} = \begin{bmatrix} T'_{Kh,1} & T'_{Kh,2} & T'_{Kh,3} & T'_{Kh,h} \end{bmatrix} = \begin{bmatrix} 2t & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & a_3 & a \\ b_1 & b_2 & b_3 & b \\ c_1 & c_2 & c_3 & c \end{bmatrix} = \begin{bmatrix} 2t & 1 & 0 \end{bmatrix} \cdot \mathbf{K}.$$

Možda da još jednom naglasimo što predstavlja oznaka T'_{Kh} . To nije derivacija krivulje u točki (što bismo trebali poistovjetiti s nagibom krivulje u toj točki). To je derivacija funkcija kojima su definirane ovisnosti o parametru t u nekoj proizvoljnoj točki T_{Kh} . Kako tih funkcija ima četiri, tako je i struktura T'_{Kh} četverokomponentni vektor.

Drugu derivaciju dobivamo deriviranjem prve derivacije; dobiva se:

$$\begin{aligned}\frac{d^2T_{Kh,1}}{dt^2} &= a_1 \cdot 2 \\ \frac{d^2T_{Kh,2}}{dt^2} &= a_2 \cdot 2 \\ \frac{d^2T_{Kh,3}}{dt^2} &= a_3 \cdot 2 \\ \frac{d^2T_{Kh,h}}{dt^2} &= a \cdot 2\end{aligned}$$

ili matrično zapisano:

$$T''_{Kh} = \begin{bmatrix} T''_{Kh,1} & T''_{Kh,2} & T''_{Kh,3} & T''_{Kh,h} \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & a_3 & a \\ b_1 & b_2 & b_3 & b \\ c_1 & c_2 & c_3 & c \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \end{bmatrix} \cdot \mathbf{K}.$$

Sve više derivacije daju:

$$\left. \begin{aligned}\frac{d^nT_{Kh,1}}{dt^n} &= 0 \\ \frac{d^nT_{Kh,2}}{dt^n} &= 0 \\ \frac{d^nT_{Kh,3}}{dt^n} &= 0 \\ \frac{d^nT_{Kh,h}}{dt^n} &= 0\end{aligned} \right\} \quad n > 2$$

odnosno matrično:

$$T^{(n)}_{Kh} = \begin{bmatrix} T^{(n)}_{Kh,1} & T^{(n)}_{Kh,2} & T^{(n)}_{Kh,3} & T^{(n)}_{Kh,h} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & a_3 & a \\ b_1 & b_2 & b_3 & b \\ c_1 & c_2 & c_3 & c \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \cdot \mathbf{K}, \quad n > 2.$$

Iz ovog jednostavnog izvoda jasno se vidi da su sve derivacije određene upravo karakterističnom maticom krivulje \mathbf{K} . Također se vidi da se sve derivacije mogu direktno dobiti samo deriviranjem elemenata matrice parametra t . Tako smo krenuli od:

$$T_{Kh} = \begin{bmatrix} t^2 & t & 1 \end{bmatrix} \cdot \mathbf{K}.$$

Prva derivacija je bila:

$$T'_{Kh} = \begin{bmatrix} 2t & 1 & 0 \end{bmatrix} \cdot \mathbf{K}.$$

Druga derivacija:

$$T''_{Kh} = \begin{bmatrix} 2 & 0 & 0 \end{bmatrix} \cdot \mathbf{K}.$$

I sve ostale više:

$$T^{(n)}_{Kh} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \cdot \mathbf{K}, \quad n > 2.$$

6.4.3 Prikaz krivulja pomoću kubnih razlomljenih funkcija

Ove funkcije omogućavaju jednostavan prikaz koničnih krivulja (krivulje koje nastaju kao presjecište stoča i ravnine, npr. kružnice, elipse i sl.), no može dati i infleksije što kvadratne razlomljene funkcije nisu mogle. Kako je riječ o kubnim funkcijama, funkcijeske ovisnosti svih koordinata u homogenom prostoru biti će izražene kubnim polinomom:

$$\begin{aligned} T_{Kh,1} &= a_1 \cdot t^3 + b_1 \cdot t^2 + c_1 \cdot t + d_1 \\ T_{Kh,2} &= a_2 \cdot t^3 + b_2 \cdot t^2 + c_2 \cdot t + d_2 \\ T_{Kh,3} &= a_3 \cdot t^3 + b_3 \cdot t^2 + c_3 \cdot t + d_3 \\ T_{Kh,h} &= a \cdot t^3 + b \cdot t^2 + c \cdot t + d \end{aligned}$$

Povratkom u radni prostor dobiva se:

$$\begin{aligned} T_{K,1} &= \frac{a_1 \cdot t^3 + b_1 \cdot t^2 + c_1 \cdot t + d_1}{a \cdot t^3 + b \cdot t^2 + c \cdot t + d} \\ T_{K,2} &= \frac{a_2 \cdot t^3 + b_2 \cdot t^2 + c_2 \cdot t + d_2}{a \cdot t^3 + b \cdot t^2 + c \cdot t + d} \\ T_{K,3} &= \frac{a_3 \cdot t^3 + b_3 \cdot t^2 + c_3 \cdot t + d_3}{a \cdot t^3 + b \cdot t^2 + c \cdot t + d} \end{aligned}$$

Jednadžbe dane za homogenih prostora opet vode na matrični zapis:

$$T_{Kh} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} a_1 & a_2 & a_3 & a \\ b_1 & b_2 & b_3 & b \\ c_1 & c_2 & c_3 & c \\ d_1 & d_2 & d_3 & d \end{bmatrix} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \mathbf{A}.$$

gdje je \mathbf{A} karakteristična matrica krivulje.

6.4.4 Parametarske derivacije u homogenom prostoru

U ovom su slučaju sve funkcijeske ovisnosti kubne:

$$\begin{aligned} T_{Kh,1} &= a_1 \cdot t^3 + b_1 \cdot t^2 + c_1 \cdot t + d_1 \\ T_{Kh,2} &= a_2 \cdot t^3 + b_2 \cdot t^2 + c_2 \cdot t + d_2 \\ T_{Kh,3} &= a_3 \cdot t^3 + b_3 \cdot t^2 + c_3 \cdot t + d_3 \\ T_{Kh,h} &= a \cdot t^3 + b \cdot t^2 + c \cdot t + d \end{aligned}$$

Deriviranjem po parametru t dobiva se:

$$\begin{aligned} \frac{dT_{Kh,1}}{dt} &= a_1 \cdot 3t^2 + b_1 \cdot 2t + c_1 \\ \frac{dT_{Kh,2}}{dt} &= a_2 \cdot 3t^2 + b_2 \cdot 2t + c_2 \\ \frac{dT_{Kh,3}}{dt} &= a_3 \cdot 3t^2 + b_3 \cdot 2t + c_3 \\ \frac{dT_{Kh,h}}{dt} &= a \cdot 3t^2 + b \cdot 2t + c \end{aligned}$$

što se matrično može zapisati kao:

$$T'_{Kh} = \begin{bmatrix} T'_{Kh,1} & T'_{Kh,2} & T'_{Kh,3} & T'_{Kh,h} \end{bmatrix} = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \cdot \mathbf{A}.$$

Više derivacije iznose:

$$\begin{aligned} T''_{Kh} &= \begin{bmatrix} 6t & 2 & 0 & 0 \end{bmatrix} \cdot \mathbf{A}. \\ T'''_{Kh} &= \begin{bmatrix} 6 & 0 & 0 & 0 \end{bmatrix} \cdot \mathbf{A}. \\ T^{(n)}_{Kh} &= \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \cdot \mathbf{A}, \quad n > 3. \end{aligned}$$

6.4.5 Veza između parametarskih derivacija u radnom i homogenom prostoru

Veza između radnih i homogenih koordinata već nam je poznata:

$$T_{Kh,i} = \frac{T_{Kh,h}}{T_{Kh,h}}, i \in 1, 2, 3$$

gdje je T_K točka u radnom prostoru, a T_{Kh} točka u homogenom prostoru. Ovu ovisnost možemo zapisati matrično:

$$T_{Kh} = T_{Kh,h} \cdot [T_{K,1} \ T_{K,2} \ T_{K,3} \ 1].$$

Prepostavimo da su nam poznate parametarske derivacije koordinata u radnom prostoru, odnosno da znamo:

$$T'_{K,1} = \frac{dT_{K,1}}{dt}, \quad T'_{K,2} = \frac{dT_{K,2}}{dt}, \quad T'_{K,3} = \frac{dT_{K,3}}{dt}$$

te da znamo funkciju kojom ćemo vršiti prebacivanje u homogeni prostor $T_{Kh,h}$ i njezinu derivaciju $T'_{Kh,h}$.

Kako vrijede veze:

$$T_{Kh,1} = T_{K,1} \cdot T_{Kh,h}, \quad T_{Kh,2} = T_{K,2} \cdot T_{Kh,h}, \quad T_{Kh,3} = T_{K,3} \cdot T_{Kh,h},$$

parametarske derivacije u homogenom prostoru mogu se primjenom pravila derivacije umnoška prikazati kao:

$$\begin{aligned} T'_{Kh,1} &= (T_{K,1} \cdot T_{Kh,h})' = T'_{K,1} \cdot T_{Kh,h} + T_{K,1} \cdot T'_{Kh,h}, \\ T'_{Kh,2} &= (T_{K,2} \cdot T_{Kh,h})' = T'_{K,2} \cdot T_{Kh,h} + T_{K,2} \cdot T'_{Kh,h}, \\ T'_{Kh,3} &= (T_{K,3} \cdot T_{Kh,h})' = T'_{K,3} \cdot T_{Kh,h} + T_{K,3} \cdot T'_{Kh,h}. \end{aligned}$$

Zapisano matrično dobije se:

$$\begin{aligned} T'_{Kh} &= [T'_{Kh,1} \ T'_{Kh,2} \ T'_{Kh,3} \ T'_{Kh,h}] \\ &= [T'_{K,1} \cdot T_{Kh,h} + T_{K,1} \cdot T'_{Kh,h} \ T'_{K,2} \cdot T_{Kh,h} + T_{K,2} \cdot T'_{Kh,h} \ T'_{K,3} \cdot T_{Kh,h} + T_{K,3} \cdot T'_{Kh,h} \ T'_{Kh,h}] \\ &= [T'_{Kh,h} \ T_{Kh,h}] \cdot [T_{K,1} \ T_{K,2} \ T_{K,3} \ 1] \\ &= [T'_{Kh,h} \ T_{Kh,h}] \cdot [\begin{matrix} T_K & 1 \\ T'_K & 0 \end{matrix}] \end{aligned}$$

Na ovaj način dobili smo direktnu vezu između traženih parametarskih derivacija u homogenom prostoru i poznatih parametarskih derivacija u radnom prostoru.

Druga parametarska derivacija u homogenom prostoru dobije se deriviranjem prve parametarske derivacije u homogenom prostoru; nakon što se deriviraju izrazi i nakon ubacivanja u matricu dobiva se:

$$\begin{aligned} T''_{Kh} &= [T''_{Kh,1} \ T''_{Kh,2} \ T''_{Kh,3} \ T''_{Kh,h}] \\ &= [(T'_{K,1} \cdot T_{Kh,h} + T_{K,1} \cdot T'_{Kh,h})' \ (T'_{K,2} \cdot T_{Kh,h} + T_{K,2} \cdot T'_{Kh,h})' \ (T'_{K,3} \cdot T_{Kh,h} + T_{K,3} \cdot T'_{Kh,h})' \ T''_{Kh,h}] \\ &= [T''_{Kh,h} \ 2 \cdot T'_{Kh,h} \ T_{Kh,h}] \cdot [\begin{matrix} T_{K,1} & T_{K,2} & T_{K,3} & 1 \\ T'_{K,1} & T'_{K,2} & T'_{K,3} & 0 \\ T''_{K,1} & T''_{K,2} & T''_{K,3} & 0 \end{matrix}] \\ &= [T''_{Kh,h} \ 2 \cdot T'_{Kh,h} \ T_{Kh,h}] \cdot [\begin{matrix} T_K & 1 \\ T'_K & 0 \\ T''_K & 0 \end{matrix}] \end{aligned}$$

I više derivacije mogu se izvesti na sličan način.

6.4.6 Primjer

Pomoću razlomljene kvadratne funkcije želimo odrediti krivulju koja će prolaziti sljedećim točkama:

- za $t = t_1 = 0$ kroz točku $[R \ 0 \ 0 \ 1]$,
- za $t = t_2 = 0.5$ kroz točku $[0 \ R \ 0 \ 1]$ te
- za $t = t_3 = 1$ kroz točku $[-R \ 0 \ 0 \ 1]$.

Na prvi pogled popis točaka odgovara kružnici, no pogledajmo što ćemo dobiti. Izračunajmo matricu \mathbf{K} temeljem izračuna koji smo već prethodno napravili.

$$\mathbf{K} = \begin{bmatrix} 2 & -4 & 2 \\ -3 & 4 & -1 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} R & 0 & 0 & 1 \\ 0 & R & 0 & 1 \\ -R & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -4R & 0 & 0 \\ -2R & 4R & 0 & 0 \\ R & 0 & 0 & 1 \end{bmatrix}.$$

Posljednji stupac $[0 \ 0 \ 1]^T$ nam govori da je riječ o paraboli, a ne o koničnoj krivulji. Sve konične krivulje imaju posljednji stupac jednak $[1 \ 0 \ 1]^T$.

Svaka točka ove naše krivulje određena je jednostavnom relacijom:

$$T_{Kh} = [t^2 \ t \ 1] \cdot \mathbf{K} = [t^2 \ t \ 1] \cdot \begin{bmatrix} 0 & -4R & 0 & 0 \\ -2R & 4R & 0 & 0 \\ R & 0 & 0 & 1 \end{bmatrix}.$$

Ako želimo prikazati kružnicu, morat ćemo se poslužiti trikom. Znamo da kod kružnice vrijedi veza:

$$x = R\cos\phi$$

$$y = R\sin\phi$$

Znamo i da postoji veza između sinusa i kosinusa:

$$t = tg\frac{\phi}{2} \quad x = R\frac{1-t^2}{1+t^2} \quad y = R\frac{2t}{1+t^2}$$

Kako je nazivnih isti i za x i za y -koordinatu, njega ćemo uzeti kao funkciju koja opisuje homogeni parametar. Tada možemo pisati:

$$T_{Kh,x} = -Rt^2 + R$$

$$T_{Kh,y} = 2tR$$

$$T_{Kh,z} = 0$$

$$T_{Kh,h} = t^2 + 1$$

iz čega direktno slijedi matrica \mathbf{K} :

$$\mathbf{K} = \begin{bmatrix} -R & 0 & 0 & 1 \\ 0 & 2R & 0 & 0 \\ R & 0 & 0 & 1 \end{bmatrix}$$

pa krivulju možemo zapisati kao:

$$T_{Kh} = [t^2 \ t \ 1] \cdot \begin{bmatrix} -R & 0 & 0 & 1 \\ 0 & 2R & 0 & 0 \\ R & 0 & 0 & 1 \end{bmatrix}.$$

Zadnji stupac nam govori da smo dobili koničnu krivulju.

6.4.7 Određivanje matrice A

U nastavku ćemo odrediti matricu **A** za krivulju koja je zadana pomoću početne i završne točke u radnom prostoru, te prve derivacije u tim točkama; početna točka dobiva se za parametar $t = t_0 = 0$ dok se završna točka dobiva za parametar $t = t_1 = 1$. Dakle, poznato nam je sljedeće:

$$\begin{aligned} t = t_0 = 0 &\Rightarrow T_0 = [T_{0,1} \quad T_{0,2} \quad T_{0,3}] \\ t = t_1 = 1 &\Rightarrow T_1 = [T_{1,1} \quad T_{1,2} \quad T_{1,3}] \\ t = t_0 = 0 &\Rightarrow T'_0 = [T'_{0,1} \quad T'_{0,2} \quad T'_{0,3}] \\ t = t_1 = 1 &\Rightarrow T'_1 = [T'_{1,1} \quad T'_{1,2} \quad T'_{1,3}] \end{aligned}$$

Kako prva točka leži na krivulji, tada zadovoljava jednadžbu krivulje. Zato možemo pisati:

$$T_{0h} = [T_{0,1} \cdot T_{0h,h} \quad T_{0,2} \cdot T_{0h,h} \quad T_{0,3} \cdot T_{0h,h} \quad T_{0h,h}] = [t_0^3 \quad t_0^2 \quad t_0 \quad 1] \cdot \mathbf{A}.$$

Ovo se kraće može zapisati sažimanjem prva tri stupca u jedan, u kojem ćemo tada pisati trokomponentnu točku radnog prostora (čime dimenzija matrice ostaje očuvana):

$$T_{0h} = [T_0 \cdot T_{0h,h} \quad T_{0h,h}] = [t_0^3 \quad t_0^2 \quad t_0 \quad 1] \cdot \mathbf{A}.$$

Za prvu derivaciju u homogenom prostoru izveli smo vezu s derivacijom u radnom prostoru:

$$T'_{0h} = [T'_0 \cdot T_{0h,h} + T_0 \cdot T'_{0h,h} \quad T'_{0h,h}] = [3t_0^2 \quad 2t_0 \quad 1 \quad 0] \cdot \mathbf{A}.$$

Slično se može pisati i za drugu točku, te se dobiju četiri jednadžbe:

$$T_{0h} = [T_0 \cdot T_{0h,h} \quad T_{0h,h}] = [t_0^3 \quad t_0^2 \quad t_0 \quad 1] \cdot \mathbf{A}.$$

$$T_{1h} = [T_1 \cdot T_{1h,h} \quad T_{1h,h}] = [t_1^3 \quad t_1^2 \quad t_1 \quad 1] \cdot \mathbf{A}.$$

$$T'_{0h} = [T'_0 \cdot T_{0h,h} + T_0 \cdot T'_{0h,h} \quad T'_{0h,h}] = [3t_0^2 \quad 2t_0 \quad 1 \quad 0] \cdot \mathbf{A}.$$

$$T'_{1h} = [T'_1 \cdot T_{1h,h} + T_1 \cdot T'_{1h,h} \quad T'_{1h,h}] = [3t_1^2 \quad 2t_1 \quad 1 \quad 0] \cdot \mathbf{A}.$$

Sustav se može zapisati kao jedna poveća matrica, te se dobiva:

$$\begin{bmatrix} T_0 \cdot T_{0h,h} & T_{0h,h} \\ T_1 \cdot T_{1h,h} & T_{1h,h} \\ T'_0 \cdot T_{0h,h} + T_0 \cdot T'_{0h,h} & T'_{0h,h} \\ T'_1 \cdot T_{1h,h} + T_1 \cdot T'_{1h,h} & T'_{1h,h} \end{bmatrix} = \begin{bmatrix} t_0^3 & t_0^2 & t_0 & 1 \\ t_1^3 & t_1^2 & t_1 & 1 \\ 3t_0^2 & 2t_0 & 1 & 0 \\ 3t_1^2 & 2t_1 & 1 & 0 \end{bmatrix} \cdot \mathbf{A}$$

Matricu na desnoj strani označiti ćemo oznakom **B** (matrica parametara). Uvrštavanjem vrijednosti za parametre dobiva se:

$$\mathbf{B} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

Matricu na lijevoj strani prethodne jednadžbe možemo malo preureediti tako da je napišemo kao umnožak dviju matrica, kako slijedi:

$$\begin{bmatrix} T_0 \cdot T_{0h,h} & T_{0h,h} \\ T_1 \cdot T_{1h,h} & T_{1h,h} \\ T'_0 \cdot T_{0h,h} + T_0 \cdot T'_{0h,h} & T'_{0h,h} \\ T'_1 \cdot T_{1h,h} + T_1 \cdot T'_{1h,h} & T'_{1h,h} \end{bmatrix} = \begin{bmatrix} T_{0h,h} & 0 & 0 & 0 \\ 0 & T_{1h,h} & 0 & 0 \\ T'_{0h,h} & 0 & T_{0h,h} & 0 \\ 0 & T'_{1h,h} & 0 & T_{1h,h} \end{bmatrix} \cdot \begin{bmatrix} T_{01} \\ T_{11} \\ T'_0 \\ T'_1 \end{bmatrix} = \mathbf{H} \cdot \mathbf{V}$$

Prva matrica u umnošku naziva se matrica \mathbf{H} (matrica ovisna samo o homogenom parametru i funkciji po kojoj se on mijenja), dok se druga matrica naziva matrica \mathbf{V} (i ona je ovisna samo o zadanim točkama i derivacijama u njima). Sada se početna jednadžba može pisati:

$$\mathbf{H} \cdot \mathbf{V} = \mathbf{B} \cdot \mathbf{A}.$$

Za matricu \mathbf{A} se dobiva:

$$\mathbf{A} = \mathbf{B}^{-1} \cdot \mathbf{H} \cdot \mathbf{V} = \mathbf{M} \cdot \mathbf{H} \cdot \mathbf{V}.$$

Matrica \mathbf{M} naziva se *univerzalna transformacijska matrica* i jednaka je inverzu matrice \mathbf{B} . Kako smo matricu \mathbf{B} već odredili, matrica \mathbf{M} iznosi:

$$\mathbf{M} = \mathbf{B}^{-1} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Ako su nam poznate točke krivulje i derivacije u njima (\mathbf{V}), te homogeni parametri i njihove derivacije za obje točke (\mathbf{H}), na temelju izvedene relacije možemo odrediti matricu \mathbf{A} kao umnožak univerzalne transformacijske matrice, matrice homogenog parametra i matrice točaka.

6.4.8 Hermitova krivulja

Hermitova krivulja specijalan je slučaj kubne razlomljene krivulje. Kod Hermitove krivulje funkcija po kojoj se mijenja homogeni parametar nije kubna, već je konstanta i iznosi 1. Drugim riječima, za točku krivulje T_{Kh} u homogenom prostoru vrijedi:

$$T_{Kh,1} = P_3(t) \quad T_{Kh,2} = Q_3(t) \quad T_{Kh,3} = R_3(t) \quad T_{Kh,h} = 1$$

gdje su P_3 , Q_3 i R_3 polinomi trećeg stupnja po parametru t .

Iskoristimo sada izraz za matricu \mathbf{A} :

$$\mathbf{A} = \mathbf{M} \cdot \mathbf{H} \cdot \mathbf{V}.$$

Matrica \mathbf{M} je poznata, matricu \mathbf{V} lako odredimo ako znamo dvije točke i derivacije u njima. Ostaje nam još matrica \mathbf{H} . Ona u sebi sadrži homogene parametre (koji su u ovom slučaju za sve točke jednaki 1 – Hermitova krivulja), i njihove derivacije. Kako su funkcije homogenih parametara konstante, njihova je derivacija 0. Time matrica \mathbf{H} postaje jedinična.

$$\begin{bmatrix} T_{0h,h} & 0 & 0 & 0 \\ 0 & T_{1h,h} & 0 & 0 \\ T'_{0h,h} & 0 & T_{0h,h} & 0 \\ 0 & T'_{1h,h} & 0 & T_{1h,h} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{I}$$

Između Hermitove krivulje i Bezierove krivulje za kubni slučaj postoji veza. Već smo rekli da Hermitova krivulja traži da je vrijednost homogenog parametra u svim točkama jednaka jedan. To znači da su prve tri homogene koordinate jednake radnim koordinatama. U tom je slučaju za prelazak iz homogenih u radne koordinate dovoljno u izrazu:

$$\mathbf{A} = \mathbf{M} \cdot \mathbf{V}.$$

matricu \mathbf{V} zamjeniti samo s njena prva tri stupca. Dobije se:

$$\vec{p}(t) = [t^3 \quad t^2 \quad t \quad 1] \cdot \mathbf{M} \cdot \begin{bmatrix} \vec{p}_0 \\ \vec{p}_1 \\ \vec{p}_0 \\ \vec{p}_1 \end{bmatrix} = [t^3 \quad t^2 \quad t \quad 1] \cdot \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \vec{r}_0 \\ \vec{r}_1 \\ \vec{r}_2 \\ \vec{r}_3 \end{bmatrix}$$

Ljeva strana jednadžbe dolazi od Hermitove krivulje; desna strana jednadžbe je Bezierova krivulja. Zaključak je da to mogu biti iste krivulje. Pri tome se Hermitova krivulja zadaje početnom i krajnjom točkom i derivacijama u njima, dok se Bezierova krivulja zadaje preko četiri točke: početnom, dvjema kontrolnima i završnom.

6.4.9 Određivanje matrice A - primjer

Zadano je sljedeće:

$$\begin{aligned} T_{0h} &= [\begin{array}{cccc} 0 & 0 & 0 & 1 \end{array}] & t = t_0 = 0, \\ T_{1h} &= [\begin{array}{cccc} 1 & 0 & 0 & 1 \end{array}] & t = t_1 = 1, \\ T'_{0h} &= [\begin{array}{cccc} 1 & 1 & 0 & 0 \end{array}] & t = t_0 = 0, \\ T'_{1h} &= [\begin{array}{cccc} 1 & -1 & 0 & 0 \end{array}] & t = t_1 = 1. \end{aligned}$$

Treba odrediti matricu **A**. Za matricu **A** smo izveli izraz:

$$\mathbf{A} = \mathbf{M} \cdot \mathbf{H} \cdot \mathbf{V}.$$

M je poznata:

$$\mathbf{M} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

H možemo odrediti:

$$\mathbf{H} = \begin{bmatrix} T_{0h,h} & 0 & 0 & 0 \\ 0 & T_{1h,h} & 0 & 0 \\ T'_{0h,h} & 0 & T_{0h,h} & 0 \\ 0 & T'_{1h,h} & 0 & T_{1h,h} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & 1 & 0 \\ 0 & b & 0 & 1 \end{bmatrix}.$$

Uočimo da smo u matricu uveli dvije pomoćne vrijednosti: a i b . Poznavanje iznosa homogenog parametra u nekoj točki ne govori nam ništa o derivaciji u toj točki. Da bismo doznali derivaciju, moramo znati funkciju po kojoj se mijenja homogeni parametar. Zato ostavljamo dvije nepoznanice: a i b koje ćemo izračunati kasnije.

Matrica **V** nam je također poznata:

$$\mathbf{V} = \begin{bmatrix} T_0 & 1 \\ T_1 & 1 \\ T'_0 & 0 \\ T'_1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix}.$$

Množenjem ove tri matrice dobiva se:

$$\mathbf{A} = \mathbf{M} \cdot \mathbf{H} \cdot \mathbf{V} = \begin{bmatrix} b & 0 & 0 & a+b \\ -b & -1 & 0 & -2a-b \\ 1 & 1 & 0 & a \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

No da bismo odredili matricu **A** jednoznačno, vidimo da nam nedostaje još jedan uvjet. Tako možemo tražiti sljedeće: neka krivulja za $t = t_2 = \frac{1}{2}$ prođe kroz točku radnog prostora $(1/2 \ 1/2 \ 0)$. Ovaj podatak pomoći će nam da odredimo traženu matricu. No računu treba pristupiti oprezno. Zadali smo točku u radnom prostoru i tražimo da krivulja prođe kroz nju. Jednadžba krivulje, međutim, daje sve točke u homogenom prostoru. I to za svaku točku radnog prostora daje samo jednu točku homogenog prostora (iako tih točaka za svaku točku radnog prostora ima beskonačno). Ukoliko ovu činjenicu ignoriramo, mogli bismo reći sljedeće: točka leži na krivulji, pa vrijedi:

$$\begin{aligned} \left[\begin{array}{cccc} \frac{1}{2} & \frac{1}{2} & 0 & 1 \end{array} \right] &= \left[\begin{array}{cccc} t_2^3 & t_2^2 & t_2 & 1 \end{array} \right] \cdot \mathbf{A} \\ &= \left[\begin{array}{cccc} \frac{1}{8} & \frac{1}{4} & \frac{1}{2} & 1 \end{array} \right] \cdot \left[\begin{array}{cccc} b & 0 & 0 & a+b \\ -b & -1 & 0 & -2a-b \\ 1 & 1 & 0 & a \\ 0 & 0 & 0 & 1 \end{array} \right] \end{aligned}$$

Ovaj sustav predstavlja četiri jednadžbe s četiri nepoznanice, i pri tome je nerješiv (zapravo, rješiv je: rješenje ne postoji). Naime, već izjednačavanjem po drugoj komponenti dobiva se:

$$\frac{1}{2} = -\frac{1}{4} + \frac{1}{2} = \frac{1}{4}$$

što je očito besmisleno. Jedno od loših tumačenja ovog rješenja je da krivulja jednostavno ne može proći kroz tu točku uz zadane parametre. I ovo je mjesto na kojem treba razmisliti. Krivulja (očito) ne može proći kroz tu točku homogenog prostora, no može li možda proći kroz neku drugu točku homogenog prostora a da pri tome prolazi kroz istu točku radnog prostora? Odgovor na ovo je potvrđan! Naime, sustav je ispravno napisati i rješavati po komponentama samo ukoliko su sve komponente u potpunosti nezavisne - što ovdje nisu. Da bismo dobili naše rješenje, potrebno je primijeniti zavisnosti koje znamo i raditi jednačenje po stvarno-nezavisnim komponentama: komponentama radnog prostora! Tada ćemo dobiti sustave:

$$\begin{aligned} \frac{\frac{1}{8}b - \frac{1}{4}b + \frac{1}{2}}{\frac{1}{8}(a+b) + \frac{1}{4}(-2a-b) + \frac{1}{2}a + 1} &= \frac{\frac{1}{2}}{1} \\ \frac{-\frac{1}{4} + \frac{1}{2}}{\frac{1}{8}(a+b) + \frac{1}{4}(-2a-b) + \frac{1}{2}a + 1} &= \frac{\frac{1}{2}}{1} \\ \frac{0}{\frac{1}{8}(a+b) + \frac{1}{4}(-2a-b) + \frac{1}{2}a + 1} &= \frac{0}{1} \end{aligned}$$

Sada umjesto četiri "nezavisne" jednadžbe imamo samo tri, i to rješive. Treća jednadžba je identitet pa otpada. Prve dvije daju:

$$\frac{-b+4}{a-b+8} = \frac{1}{2} \quad \frac{2}{a-b+8} = \frac{1}{2} \quad \Rightarrow \quad a = -2, b = 2.$$

Tražena matrica \mathbf{A} glasi:

$$\mathbf{A} = \left[\begin{array}{cccc} 2 & 0 & 0 & 0 \\ -2 & -1 & 0 & 2 \\ 1 & 1 & 0 & -2 \\ 0 & 0 & 0 & 1 \end{array} \right].$$

Ostalo nam je još da pogledamo kroz koju je točku homogenog prostora krivulja prošla za traženu točku radnog prostora:

$$T_{Kh} = \left[\begin{array}{cccc} \left(\frac{1}{2}\right)^3 & \left(\frac{1}{2}\right)^2 & \frac{1}{2} & 1 \end{array} \right] \cdot \left[\begin{array}{cccc} 2 & 0 & 0 & 0 \\ -2 & -1 & 0 & 2 \\ 1 & 1 & 0 & -2 \\ 0 & 0 & 0 & 1 \end{array} \right] = \left[\begin{array}{cccc} \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{2} \end{array} \right]$$

a odgovarajuća točka radnog prostora je doista tražena:

$$T_K = \left[\begin{array}{ccc} \frac{1}{2} & \frac{1}{2} & 0 \end{array} \right] = \left[\begin{array}{ccc} \frac{1}{2} & \frac{1}{2} & 0 \end{array} \right].$$

Poglavlje 7

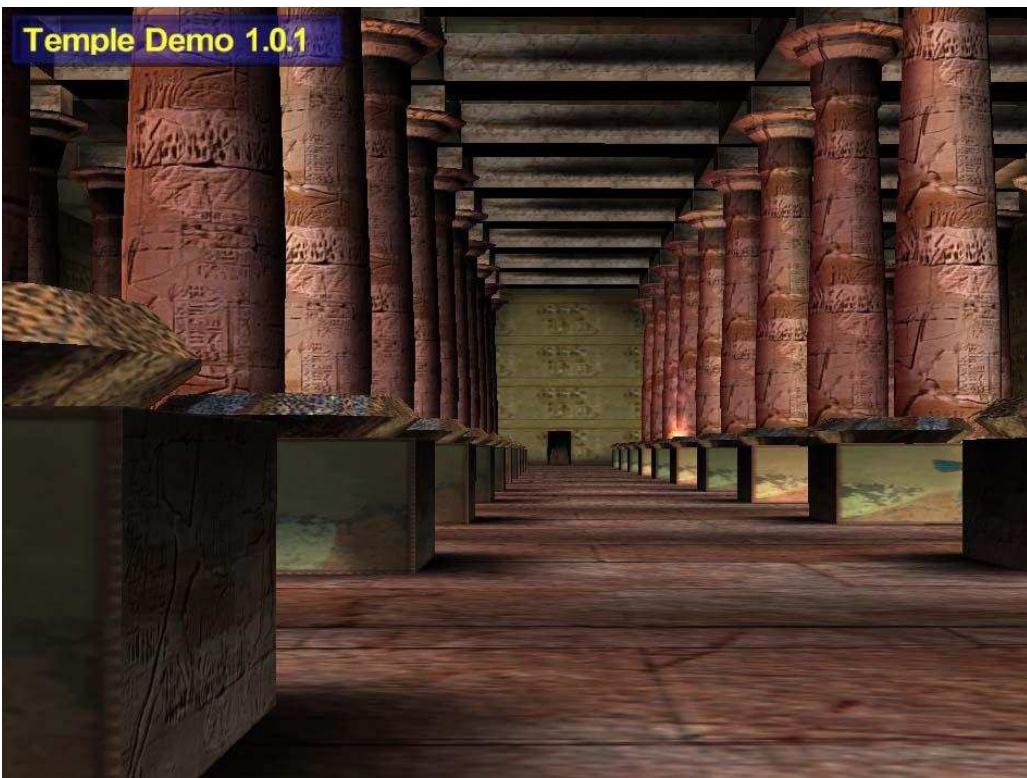
Uklanjanje skrivenih linija i površina

7.1 Uvod

Objekte scene opisujemo matematičkim primitivama: vrhovima, bridovima, poligonima, funkcijama i sl. Rad s ovakvim opisom postaje računski zahtjevan kako scena raste i kako detaljnost pojedinih objekata raste. Iako broj poligona koji u jedinici vremena grafička kartica može prikazati, kroz nekoliko posljednjih godina, raste nevjerojatnom brzinom, rastu i naša očekivanja na vjernost prikaza i različite učinke koje nam prikaz omogućava. Na ovaj način scena postaje sve složenija a time i vrijeme potrebno da se ostvari njen prikaz. Da bi se iscrtavanje scene maksimalno ubrzalo, treba iz postupka iscrtavanja izbaciti sve radnje koje su nepotrebne (a takvih ima). Npr. ukoliko u scenu stavimo kocku, gdjegod da se nalazio promatrač, nikada mu neće biti vidljive sve stranice kocke. "Nevidljive" stranice zapravo su stražnje stranice kocke gledano iz smjera promatrača. Stražnje stranice ne treba niti sjenčati niti preslikavati teksturu na njih jer ionako nisu vidljive i trebaju biti uklonjene (engl. *backface culling*). Kocka predstavlja trivijalan primjer, no treba imati na umu da će objekt koji ima nekoliko milijuna poligona, uz uklanjanje samo stražnjih poligona, imati oko pedeset posto manje poligona koje treba prikazati, čime ćemo bitno rasteretiti grafički protočni sustav. Scenu također treba promatrati u cjelini, a iscrtavanje svih objekata i dijelova objekata – vidljivih i nevidljivih – nepotrebno će preopteretiti prikaz, kao na slici 7.1. Na slici je vidljivo da je prisutno puno stražnjih poligona, za svaki stup barem pola, koji nisu vidljivi. Stupovima je zaklonjen i veliki dio scene koji je iza njih i nije vidljiv. A ako pogledamo i same stupove, međusobno se zaklanjavaju, pa sagledavši sve skupa imamo veliku količinu poligona koja će se iscrtavati, a neće biti vidljiva ili će se djelomično zaklanjati.

No, treba biti oprezan u odlučivanju kada su zaista pojedine plohe nevidljive, odnosno kada se smiju ukloniti. Ako u sceni imamo zrcala ili prozirne objekte tada se može desiti da neke dijelove objekata vidimo indirektno i u tom slučaju te poligone ne smijemo ukloniti. To može biti odraz na podu ili na površini vode, pa će neki dijelovi, iako su stražnji za promatrača, biti vidljivi na zrcalnoj površini ili kroz proziran objekt. Drugo na što treba paziti je ako poligone smijemo ukloniti, na kojem mjestu u protočnom sustavu to treba učiniti. Proračun normala u vrhovima zahtjeva poznavanje normala susjednih poligona kako bi ih mogli odrediti. Ako uklonimo nevidljive poligone za promatrača uklonit ćemo i poligone koji diraju vrhove na silueti objekta. Učinimo li to prije izračunavanja normala u vrhovima, za točke na rubnim dijelovima objekta nećemo imati potrebne poligone, odnosno njihove normale, potrebne za određivanje normala u vrhovima.

Općenito, cilj algoritama za uklanjanje skrivenih linija i površina (engl. *hidden surface removal*) je ukloniti što je moguće prije i što je moguće više toga što je nepotrebno u grafičkom protočnom sustavu. Stoga se različite provjere rade na raznim mjestima i u pravilu na više mjesta u protočnom sustavu. Provjere koje nisu složene i nemaju zahtjevan izračun, a uklanjuju dijelove koji sigurno nisu vidljivi, obavljaju se ranije, kako bi do vremenski zahtjevnijih algoritama došao što manji broj nepotrebnih poligona. Uz sve provjere koje načinimo do konačnog koraka iscrtavanja, još uvijek dolazi u pravilu više pristupa iscrtavanju slikovnih elemenata nego što će u konačnici biti vidljivo. Tada promatramo koliko tog "viška" dolazi do kraja protočnog sustava (engl. *overdraw*). Procjenjuje se da je taj broj u aplikacijama kao što su računalne igre, otprilike tri, uz sve prethodne provjere i uklanjanje skrivenih dijelova. Odnosno, za iscrtavanje svakog slikovnog elementa na zaslonu bit će tri pokušaja iscrtavanja



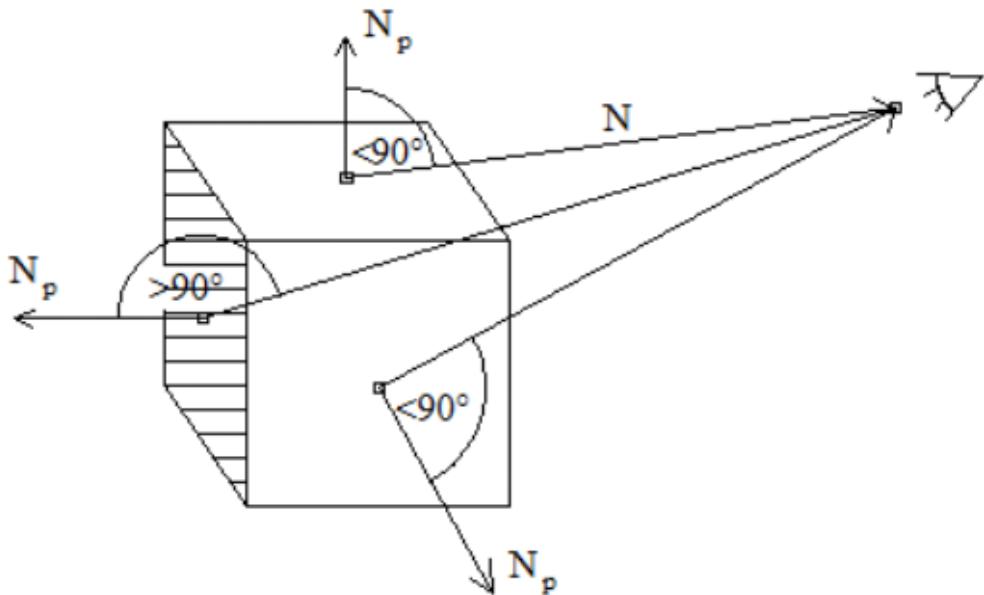
Slika 7.1: Primjer scene s puno objekata na kojoj je prisutna velika količina objekata (npr. stupovi u pozadini) i poligona koji se ne vide ili su djelomično zaklonjeni.

kako bi se iscrtao slikovni element koji će se u konačnici vidjeti. To se, naravno, obavlja prije konačnog osvježavanja iz slikovne memorije na zaslon.

Neke provjere koje ćemo ovdje objasniti imaju sklopovsku podršku jer značajno pridonose rassterećenju prikaza i k tome se obavljaju vrlo brzo, a neke su dane kao ideje koje se mogu koristiti i u različitim drugim kontekstima. Na primjer, kod detekcije kolizije javljaju se slični problemi kao i kod uklanjanja skrivenih linija i površina. Kod detekcije kolizije potrebno je odrediti prodire li jedan objekt u drugi. Kada i gdje dolazi do kontakta, potrebno je odrediti kako bi mogli načiniti međusobnu reakciju pri sudaru dva objekata. Ovi problemi su slični, pa su i algoritmi koji se koriste za uklanjanje skrivenih linija i površine, odnosno načini razmišljanja primjenjivi. Postupci odsijecanja koji se obavljaju obzirom na volumen pogleda ili otvor prikaza, također daju važne ideje i upotrebivi su u raznim drugim kontekstima. Bačene sjene, ako ih želimo realizirati u stvarnom vremenu također vuku bitne koncepte koji su se izvorno primjenjivali na problematiku uklanjanja skrivenih linija i površina. Vrijedi, naravno, i obrnuto nove ideje koje se javljaju pri razvoju algoritama detekcije sudara ili ostvarivanja sjene mogu biti upotrebive u postupcima uklanjanja skrivenih linija i površina. Primjer je ideja koja je razvijena za postupak ostvarivanja sjene. Objekti se dijele na one koji su zaklanjavajući i objekte koji su zaklonjeni, odnosno objekte u području sjene i nisu vidljivi za izvor. Način određivanja zaklonjenih objekata primjenjiv je i u postupcima određivanja skrivenih linija i površina.

S obzirom da se radi o cijelom nizu algoritama koji se koriste u ove svrhe, možemo ih podijeliti na algoritme koji rade u prostoru scene – trodimenzijskom prostoru – i algoritme koji rade u prostoru projekcije, odnosno u dvodimenzijskom prostoru. Valja primijetiti i da, iako ćemo načiniti ovu podjelu, pojedini algoritmi koji su definirani u dvodimenzijskom prostoru proširivi su i na trodimenzijski prostor. Također, možemo primijetiti da su neki postupci vrlo jednostavnii, no mogu ukloniti ne mali broj objekata ili njihovih dijelova, pa ih je stoga poželjno što prije i češće koristiti.

Sve skupa, zajedno gledano, algoritmi uklanjanja skrivenih linija i površina troše neko vrijeme, no njihova primjena utječe na uštedu zbog smanjenja broja objekata ili njihovih dijelova u konačnom prikazu. U ovom poglavlju način ćemo pregled kako vrlo jednostavnih, tako i složenih algoritama koji troše vrijeme i memoriju za pohranu struktura podataka. Utrošak vremena na različite provjere mora biti isplativ prema učinku koji se time postiže. Funkcija cilja je povećati broj iscrtanih slika u



Slika 7.2: Uklanjanje poligona na osnovi kuta između vektora normale poligona i vektora prema promatraču

jedinici vremena (*fps*), pa je to obično glavni kriteriju u ocjeni koji će algoritam i gdje biti korišten, uz prihvativ utrošak za strukture podataka.

7.2 Uklanjanje stražnjih poligona – provjera normale

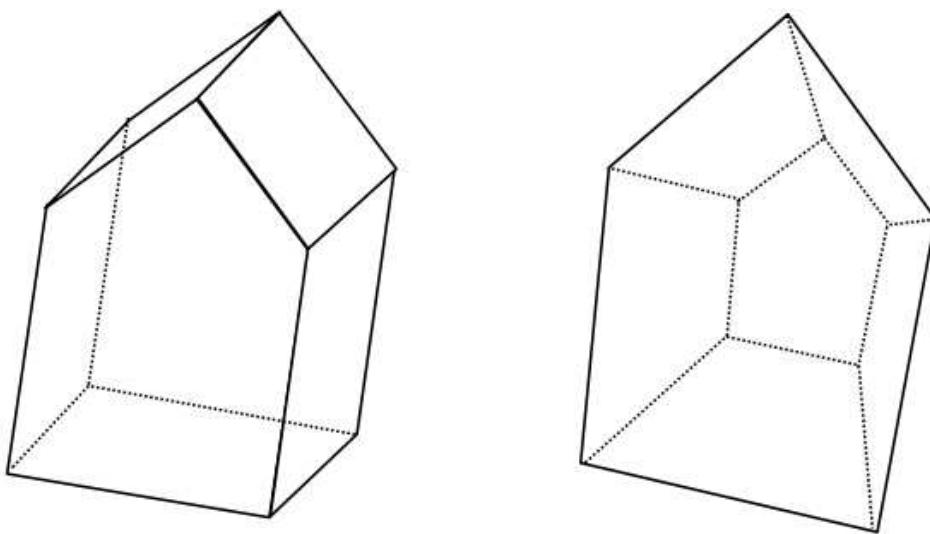
Ovaj postupak pomoći će nam da otkrijemo koji su poligoni tijela stražnji u odnosu na promatrača. Pretpostavka je da je tijelo opisano nizom poligona, i da je tijelo konveksno. Nadalje, pretpostavlja se da su vrhovi poligona zadani u smjeru suprotnom od smjera kazaljke na satu ako gledamo poligon iz točke koja se nalazi izvan tijela. Ovo će za posljedicu imati da sve normale poligona gledaju iz tijela prema van. Tada se odluka je li poligon prednji ili stražnji može donijeti na temelju kuta što ga *normala poligona* zatvara s *vektorom usmjerenim prema gledatelju*. Slika 7.2. demonstrira takvo zaključivanje.

Potrebno je promatrati kut što ga zatvara vektor normale poligona i vektor iz središta poligona prema promatraču. Tada vrijedi:

- poligon je stražnji, ako je $\vec{N}_P \cdot \vec{N} < 0$,
- poligon je prednji (vidljiv), ako je $\vec{N}_P \cdot \vec{N} > 0$, te
- poligon je degenerirao u liniju (za promarača) ako je $\vec{N}_P \cdot \vec{N} = 0$.

Uklanjanje stražnjih poligona možemo načiniti korištenjem još jedne jednostavnije provjere. Položaj promatrača možemo uvrstiti u jednadžbu ravnine koja sadrži pojedini poligon. Ako je točka promatrača "iznad" taj poligon je vidljiv, a ako je "ispod" poligon nije vidljiv. Ova provjera ima manje računskih operacija, a u osnovi radimo istu stvar kao i kod provjere kuta između vektora normale i vektora prema očištu.

Pri uklanjanju stražnjih poligona važno je obratiti pažnju na vrstu projekcije koja se obavlja. Ako se obavlja ortografska projekcija, neće biti problema, no ako se koristi perspektivna projekcija treba biti oprezan. Na slici 7.3 prikazan je objekt uz ortografsku projekciju lijevo i perspektivnu desno, za isti položaj promatrača. Crtkano su prikazane linije koje čine bridove stražnjih poligona i ne bi trebale biti vidljive. Treba primjetiti da su poligoni koji su skriveni u jednom i drugom slučaju različiti. Ako ne vodimo računa o transformaciji normala poligona pri perspektivnoj projekciji krov kućice i



Slika 7.3: Ortografska i perspektivna projekcija

lijeva strana u projekciji perspektivnom projekcijom bit će prikazani iako nisu vidljivi u toj projekciji. Znači, prilikom korištenja perspektivne projekcije moramo voditi računa da transformaciji podvrgnemo vektore normala koje smo izračunali na početku ili da nakon obavljanja projekcije u transformiranom prostoru računamo jednadžbe ravnina na osnovi transformiranih vrhova. Kako često želimo mijenjati položaj očišta, u prvom pristupu imat ćemo manji broj računskih operacija.

Postupak uklanjanja stražnjih poligona može raditi i u prostoru projekcije. Ovaj pristup temelji se na provjeri orijentacije poligona u 2D prostoru projekcije. Možemo primijetiti da je, za poligone koji u trodimenzijskom prostoru imaju redoslijed obilaženja vrhova u smjeru suprotnom od smjera kazaljke na satu gledano izvan objekta, u prostoru projekcije ta orijentacija sačuvana za poligone okrenute prema promatraču, a obrnuta je za promatraču skrivene poligone. Ako koristimo ovu provjeru ne moramo paziti na vrstu projekcije jer će i kod ortografske i kod perspektivne projekcije navedeno svojstvo vrijediti.

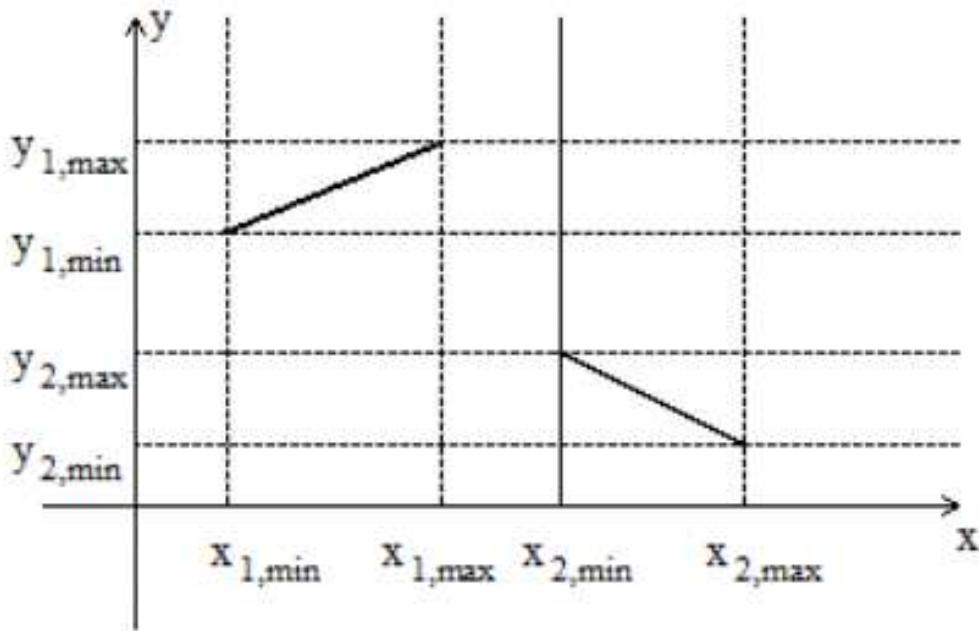
Postupak uklanjanja stražnjih poligona obično je sklopovski podržan. Standardom *OpenGL* ova je operacija podržana, i kako bi je aktivirali, potrebno je zadati niz naredbi prikazanih u kodu u nastavku.

```
glFrontFace(GL_CCW);      // ili GL_CW
glEnable(GL_CULL_FACE);   // glDisable(GL_CULL_FACE);
glCullFace(GL_BACK);     // ili GL_FRONT ili GL_FRONT_AND_BACK
```

Prvo moramo odrediti koju orijentaciju imaju poligoni koje smatramo prednjima. Nakon toga moramo omogućiti uklanjanje skrivenih poligona, te odabrati koju orijentaciju želimo koristiti pri uklanjanju. Uporabom konstante *GL_FRONT_AND_BACK* možemo ukloniti i prednje i stražnje poligone. Ova operacija ima smisla ako u sceni imamo i linijske segmente koji na ovaj način neće biti uklonjeni pa će biti istaknuti.

7.3 Minimaks provjere

Minimaks provjera služi nam kao brza provjera da li se dva objekta *ne preklapaju*. Prethodna rečenica zvuči malo čudno, ali ispravno karakterizira postupak. Naime, minimaks provjerama možemo utvrditi je li sigurno da se dva objekta ne preklapaju. Ukoliko pak utvrdimo da postoji mogućnost preklapanja objekata, tada daljnje provjere treba obavljati drugim (tipično kompleksnijim i sporijim) algoritmima. Evo ideje. Svaki objekt u sustavu prikaza ima svoje minimalne i maksimalne koordinate kroz koje se proteže. Usporedbom tih vrijednosti možemo donijeti neke zaključke. Primjer je prikazan na slici 7.4. Provjerava se preklapanja dvaju segmenata linije. Linije su uzete radi jednostavnosti, no postupak je identičan za bilo kakve druge objekte.



Slika 7.4: Primjer min-maks provjere na dvije dužine

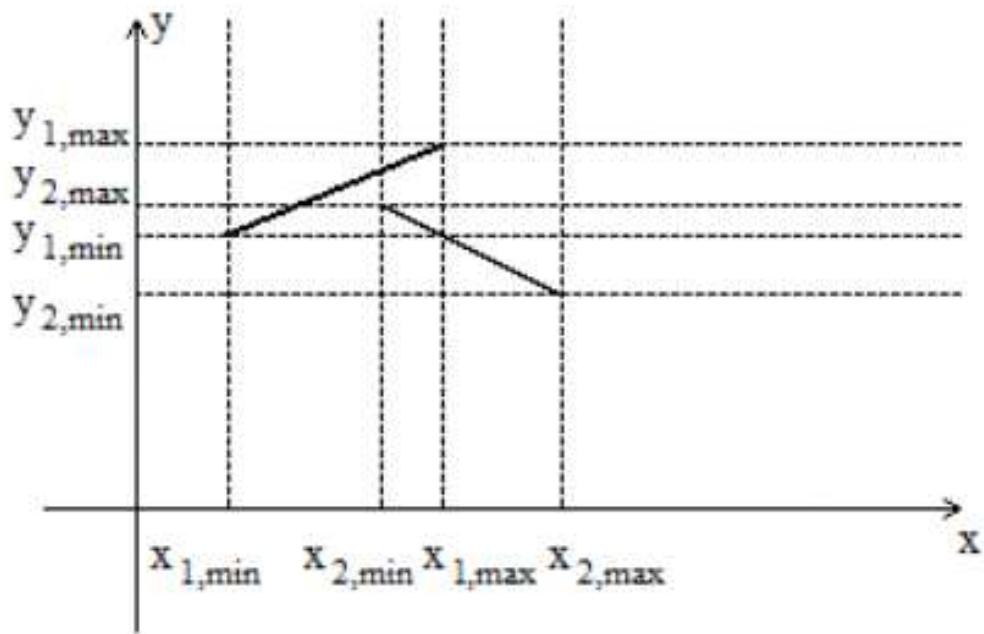
Postavlja se pitanje, kada možemo biti sigurni da se objekti ne preklapaju? Četiri su takva slučaja. Objekti se sigurno ne preklapaju, ako je:

- $x_{1,\max} < x_{2,\min}$ ili
- $x_{2,\max} < x_{1,\min}$ ili
- $y_{1,\max} < y_{2,\min}$ ili
- $y_{2,\max} < y_{1,\min}$.

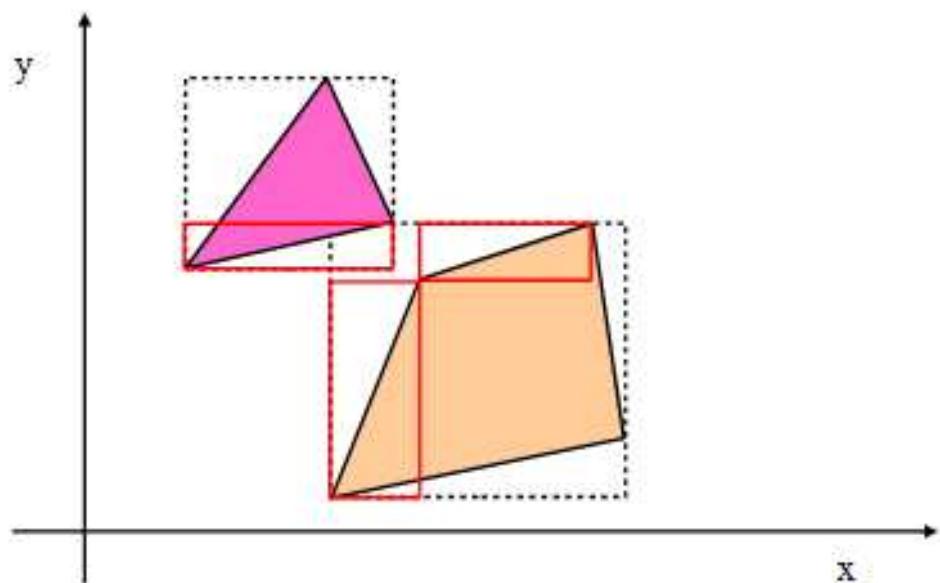
Ako je zadovoljen bilo koji od ova četiri uvjeta, tada smo sigurni da se objekti ne preklapaju. Ako niti jedan od ta četiri uvjeta nije ispunjen, tada treba izvršiti daljnje provjere. Naime, i dalje je moguće da se objekti ne preklapaju, a da niti jedan od prethodnih uvjeta nije ispunjen. Ovakav primjer ilustriran je na slici 7.5, gdje se objekti i dalje ne preklapaju, no minimaks provjerom to ne možemo utvrditi. Minimaks provjera u ovom slučaju rezultirat će potrebom za dodatnim provjerama, jer se kvadratna područja u kojima leže objekti preklapaju.

Valja primjetiti kako je osnova minimaks provjere zapravo provjera koja gleda preklapaju li se projekcije objekata na os x , ili na os y . U općem slučaju to može biti bilo koja os, odnosno pravac. Koordinatne osi posebno su pogodne zbog jednostavnosti primjene. Kada provjeravamo preklapanje po obje osi x i y u stvari provjeravamo da li se preklapaju pravokutnici koji obuhvaćaju promatrane objekte. Ako se pravokutnici ne preklapaju tada zaključujemo da se objekti sigurno ne preklapaju, odnosno da sigurno postoji pravac koji je između promatranih objekata. Kod složenijih objekata postupak možemo hijerarhijski primijeniti. Na primjer, provjeravamo li za dva poligona da li se preklapaju te utvrđimo da se potencijalno preklapaju, provjeru možemo primijeniti na međusoban odnos svaka dva brida koji sačinjavaju poligon. Ovaj slučaj je prikazan na slici 7.6 gdje se poligoni potencijalno preklapaju, ali usporedbom bridova potom zaključujemo da se niti jedan brid prvog objekta sigurno ne preklapa niti s jednim bridom drugog objekta.

Opisana ideja proširiva je na trodimenijski prostor gdje se provjerava preklapanje omedujućih pravokutnika koji obuhvaćaju objekte. Ovaj postupak koristi se pri detekciji kolizije dva objekta i naziva se *provjera omedujućih kvadara poravnatih s koordinatnim osima* (engl. *AABB axes aligned bounding box*). Nedostatak ovog postupka je u tome što ako je objekt uzak i dugačak, kvadar koji



Slika 7.5: Primjer kada se dvije dužine potencijalno preklapaju



Slika 7.6: Poligoni se potencijalno preklapaju, ali niti jedan brid prvog objekta sigurno se ne preklapa s niti jednim bridom drugog objekta

ga obuhvaća zahvaća velik dio prostora, posebno kada je objekt položen dijagonalno, pa je procjena preklapanja objekata u tom slučaju vrlo loša.

Želimo li napraviti ispitivanje složenijih objekata uporabom ovog algoritma, najprije ćemo složenije objekte obuhvatiti kvadrima. Provjeru jesu li objekti u koliziji najprije ćemo obaviti nad kvadrima jer je to bitno jednostavnije. Tek ako se objekti potencijalno preklapaju idemo u vremenski zahtjevnije, odnosno skuplje provjere. Objekti kojima obuhvaćamo naše složene objekte mogu biti i kugle, koje se često korste zbog jednostavnosti provjere. Za dvije kugle dovoljno je provjeriti udaljenost od njihovih središta; kako znamo radijuse tih kugli, trivijalno je provjeriti preklapaju li se potencijalno objekti koje te kugle obuhvaćaju.

7.4 Postupak Watkinsa

Postupak koji je osmislio Watkins često se referencira i kao *algoritam linije pretrage* (engl. *scan-line algorithm*), odnosno nastavlja se na ovaj algoritam. Algoritam radi u prostoru projekcije. Osnovna ideja proizlazi iz postupka rasterizacije poligona liniju po liniju, tako da je cilj iskoristiti informaciju koja čini kontekst i proširiti je duž linije pretrage, a isto tako i duž bridova koji čine poligone.

Poligone promatramo u ravnini projekcije; neka to za potrebe primjera bude xy -ravnina. Krenemo li promatrati za jednu liniju pretrage, potrebno je odrediti koji su dijelovi vidljivi. Na slici 7.7 prikazana su dva poligona koja se po dubini sijeku. Trenutno promatrana linija pretrage je označena s y_i . Za promatranu liniju želimo odrediti vidljivost pojedinih poligona duž te linije. Desno je prikazan presjek kroz promatrane poligone po dubini, odnosno po z -osi. Za trenutno promatranu liniju pretrage y_i desna slika predstavlja presjek dva promatrana poligona ravninom koja je okomita na xy -ravninu. U presjeku vidimo da se poligoni sijeku po dubini, odnosno duž z -osi. Ovdje je cilj odrediti raspone uzorka. *Raspon uzorka* definiramo kao dio linije na kojoj se ne može dogoditi promjena vidljivosti. Raspon uzorka je dio linije koji zadovoljava sljedeće uvjete:

1. broj segmenata u rasponu uzorka konstantan je i veći od jedan, te
2. projekcije presjeka za $y = y_i$ unutar raspona uzorka u ravnini xz ne sijeku se unutar raspona uzorka (svaki presjek u projekciji označava novi raspon uzorka).

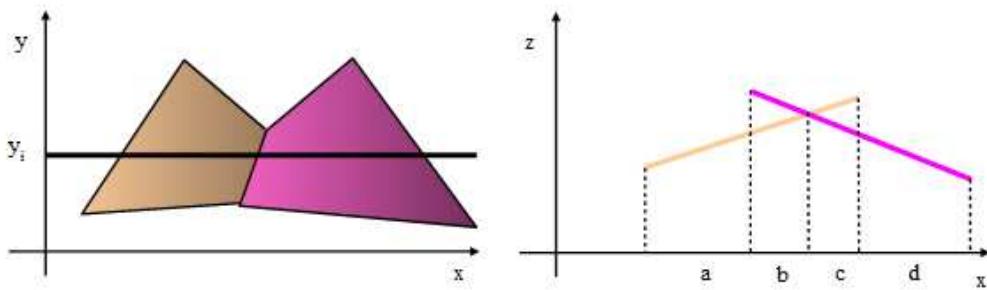
U primjeru na slici 7.7 desno označeni su rasponi uzorka. Promatrajući s lijeva na desno označen je raspon uzorka (a) dobiven na osnovi uvjeta (1) jer tu počinje prvi poligon promatrano s lijeva. U daljnjem dijelu vidimo da se poligoni po dubini sijeku pa su prema uvjetu (2) dobiveni rasponi uzorka (b) i (c), nakon toga završava prvi poligon s lijeva što određuje početak raspona uzorka označenog s (d). Kraj drugog poligona određuje završetak raspona uzorka (d).

Nakon određivanja raspona uzorka potrebno je odrediti vidljivost pojedinog raspona. Vidljivost se određuje na osnovi udaljenosti od promatrača. Ako je promatrač na slici 7.7 desno postavljen iz negativnog smjera z -osi na rasponu uzorka (a) i (b) bit će vidljiv prvi poligon, a na rasponu uzorka (c) i (d) drugi poligon.

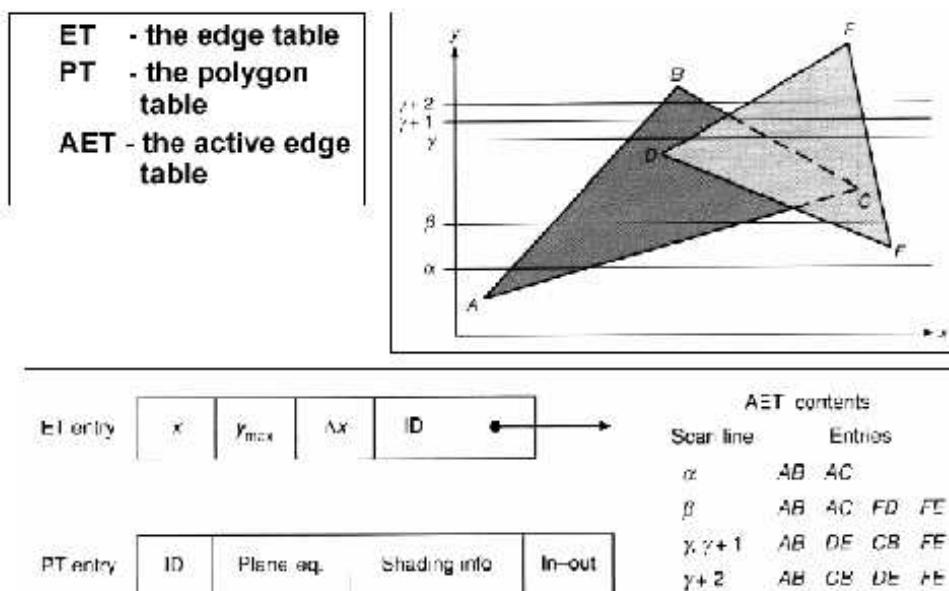
Kada nije dostupna sklopovska grafička podrška za uklanjanje skrivenih linija i površina, ovaj algoritam je često bio korišten u programskoj izvedbi ostvarivanja prikaza (engl. *software renderers*). Vidimo da je osnovna ideja u tome da se ispitivanje ne provodi točku po točku, već je cilj iskoristiti kontekst, odnosno odrediti raspone za koje vrijedi svojstvo vidljivosti. Prilikom određivanja točku po točku dolazi do dodatnog opterećivanja dohvatom podataka, određivanja vidljivosti, slanjem podataka za svaku točku. Na starijim sustavima gdje nije postojala sklopovska podrška za grafiku, to je bilo toliko izraženo da je grupiranje podataka u veće cjeline (raspone) i grupiranje linija pretrage u blokove imalo drastičan učinak na brzinu ostvarivanja prikaza. Danas, na ugređenim sustavima ovi koncepti ponovo imaju smisla, ali i proširenje koncepta na više dimenzija.

Sljedeći korak u razvoju algoritma je proširenje konteksta na susjedne linije pretrage i poligone koji čine prikaz. Stvara se tablica bridova (ET, engl. *Edge Table*), za sve ne-horizontalne bridove (slika 7.8). Pojedini zapis brida, sortiran po rastućoj y -koordinati, sadrži informaciju o:

- x -koordinati kraja koji ima manju y -koordinatu,



Slika 7.7: Linija pretrage (lijevo) i rasponi uzorka (desno) u postupku Watkinsa



Slika 7.8: Podatkovne strukture u postupku Watkinsa

- y -koordinati drugog kraja brida,
- promjena po x , odnosno Δx promjena pri prelasku u sljedeću liniju pretrage, (recipročna vrijednost nagiba brida), te
- identifikator koji određuje kojem poligonom brid pripada.

Tablica poligona (PT, engl. *Polygon Table*) sadrži informaciju o:

- koeficijentima jednadžbe ravnine,
- zastavica koja određuje je li poligon aktivan (je li brid aktivan), te
- druge podatke, odnosno attribute kao što je boja, tekstura i sl.

Na slici 7.8 prikazana su dva projicirana poligona, a skriveni dijelovi prikazani su crtkano. Gradi se tablica aktivnih bridova (AET, engl. *Active Edge Table*) koja je prikazana na slici. Na dijelu gdje je linija pretrage u području označenom s α , aktivni su bridovi AB i AC jer tim redom slijede x -koordinate sjecišta s tim bridovima. Nailaskom na brid AB postavlja se zastavica poligona ABC , a prolaskom kroz brid AC se spušta. Gledano po y -koordinati to se ne mijenja sve do točke F . Zatim slijedi područje β u kojem su aktivni AB , AC , FD i FE . U ovom dijelu uvjek je aktivan u jednom trenutku samo jedan poligon i u prvom dijelu algoritma gdje određujemo raspone uzorka nije potrebo

ispitivati vidljivost već možemo koristiti informaciju s prethodne linije pretrage. Na slici nije posebno istaknut prijelaz kada se bridovi AC i FD sijeku, i time zamijene redoslijed u AET. Na tom dijelu kao i na γ , u jednom dijelu raspona bit će aktivna oba poligona istovremeno, gdje će onda ispitivanje z -koordinate odlučiti što je bliže. Ako se poligoni probadaju, to se može utvrditi na početku i linija koja određuje njihov presjek uvodi se kao poseban brid.

U nastavku ćemo dati malo i nešto detaljniji pseudokod ovog algoritma. Krenimo s potrebnim podatkovnim strukturama. Pri razmišljanju o ovom algoritmu potrebno je zapamtiti da algoritam scenu iscrtava jednu po jednu vodoravnu liniju, počevši od $y = y_{min}$ pa do $y = y_{max}$.

```

1 struct triangle_data_str;
2
3 // struktura za pamcenje jednog brida
4 typedef struct {
5     point3d *v1; // prvi vrh brida
6     point3d *v2; // drugi vrh brida
7     struct triangle_data_str *p_trokut; // pokazivac na trokut
8     int y; // minimalna y koordinata brida
9     int dy; // koliko puta ga scan-linija sijece
10    double x, dx; // presjek sa scan-linijom te prirast x-a
11 } edge_data;
12
13 // struktura za pamcenje trokuta
14 typedef struct triangle_data_str {
15     int y; // minimalna y koordinata poligona
16     int dy; // koliko puta ga scan-linija sijece
17     double a,b,c,d; // koeficijenti ravnine
18     f_rgb_color boja; // boja trokuta
19     bool aktivan; // je li aktivan
20     edge_data bridovi[3]; // bridovi
21 } triangle_data;
22
23 // podatci potrebni za Watkinsov algoritam
24 typedef struct {
25     list aktivni_trokuti; // popis aktivnih trokuta
26     list aktivni_bridovi; // popis aktivnih bridova
27     list **buckets; // polje lista trokuta koji pocinju na y-u
28     double intervalL, intervalD; // Interval koji se ispituje
29     int xmin, xmax; // raspon x koordinate ekrana
30     int ymin, ymax; // raspon y koordinate ekrana
31     f_rgb_color boja_pozadine; // boja pozadine
32 } watkins_data;

```

Struktura `edge_data` služi za pamćenje podataka o jednom bridu. Brid je određen točkama `v1` i `v2`. Element `p_trokut` je pokazivač na trokut kojemu ovaj brid pripada (za potrebe algoritma pretpostavimo da se bridovi ne dijele). Kako bismo definirali sadržaj varijabli `x`, `y`, `dx` i `dy`, promotrimo točke `v1` i `v2`, i pretpostavimo da vrijedi da nakon projekcije u xy -ravninu `v1` ima manju y -koordinatu od `v2`. U slučaju da obje točke imaju jednaku y -koordinatu, tada vrijedi da je x -koordinata točke `v1` nakon projekcije u xy -ravninu manja ili jednaka x -koordinati točke `v2`. Ako ovo nije zadovoljeno, dovoljno je zamijeniti točke `v1` i `v2`. U tom slučaju element `y` predstavlja y -koordinatu točke `v1` (niže točke), element `x` predstavlja x -koordinatu točke `v1`. `dx` je tada $(v_{2,x} - v_{1,x})/(v_{2,y} - v_{1,y})$, odnosno što govori kako treba promjeniti sadržaj varijable `x` ako se `y` poveća za jedan (pri tome su `x` i `y` koordinate one koje se dobiju nakon projekcije tih točaka u xy -ravninu). Element `dy` postavlja se na vrijednost $v_{2,y} - v_{1,y}$, što nam govori kroz koliko se različitim y -a (dakle, kroz koliko scan-linija) proteže sam brid.

Struktura `triangle_data` sadrži podatke o jednom trokutu (algoritam je trivijalno proširiv na konveksne poligone). Element `y` sadrži y -koordinatu dna poligona, a element `dy` predstavlja visinu poligona (tj. broj scan-linija koje prelaze preko poligona); ovaj podatak dobijemo kao razliku maksimalne i minimalne y -koordinate iz skupa vrhova poligona nakon projekcije u xy -ravninu. Elementi `a`, `b`, `c` i `d` predstavljaju koeficijente implicitnog oblika jednadžbe ravnine u kojoj leži trokut. Element `boja` predstavlja boju kojom je potrebno iscrtati trokut. Element `aktivan` koristit će se prilikom rada algoritma pa će biti pojašnjen uskoro. Konačno na kraju strukture nalazi se polje podataka o bridovima trokuta (prethodno opisana struktura `edge_data`).

Konačno, struktura `watkins_data` sadrži se što potrebno za rad algoritma. Element `aktivni_trokuti` je lista trokuta preko kojih prelazi trenutna scan-linija. Slično, element `aktivni_bridovi` je lista bridova trokuta iz liste `aktivni_trokuti` preko kojih prelazi trenutna scan-linija. Naime, kako bismo izbjegli potrebu da neprestano prolazimo kroz kompletan popis trokuta i bridova, i provjeravamo da li ih trenutna scan-linija $y = y_i$ siječe ili ih možemo zanemariti, koristi se pametniji pristup. Već smo objasnili da se za svaki trokut pamti y -koordinata dna trokuta, a za svaki brid y -koordinata "nižeg" vrha. Da bismo ovo iskoristili, definirali smo pomoćno polje `buckets` koje ima onoliko elemenata koliko ima različitih y -vrijednosti između y_{min} i y_{max} (dakle, po jedan element za svaku scan-liniju). i -ti element tada je lista svih trokuta koji počinju na $y = i$. Jednom kada se trokut doda u listu aktivnih trokuta, svakim podizanjem scan-linije vrijednost njegove varijable `dy` umanjujvat ćeemo za jedan – trokut se izbacuje iz liste aktivnih trokuta onog trenutka kada mu `dy` padne na 0. Slično, za svaku novu scan-liniju proći ćeemo kroz popis bridova aktivnih trokuta i one bridove koji započinju na trenutnoj scan-liniji dodat ćeemo u listu aktivnih bridova. Svakim podizanjem scan-linije bridovima iz liste aktivnih bridova umanjujvat ćeemo njihov `dy` i kada on padne na 0, brid ćeemo izbaciti iz popisa aktivnih bridova.

Uočimo da uporabom ovakvog pristupa nikada ne moramo računati sjedište brida i trenutne scan-linije. Naime, kada scan-linija prvi puta dođe do brida, znamo da ga siječe u pohranjenoj točki `x` (koju čuva struktura brida – to je bila x -koordinata nižeg od vrhova brida). Svakim podizanjem scan-linije `x` brida povećat ćeemo za `dx`, i time će nam `x` opet predstavljati korektnu x -koordinatu sjedišta. Na ovaj način štedi se na broju operacija računskih izračuna.

Varijable `intervalL` i `intervalD` predstavljaju pomoćne varijable – x -koordinate lijevog i desnog segmenta na scan-liniji koji treba iscrtati. `xmin`, `xmax`, `ymin` i `ymax` predstavljaju raspon x i y -koordinata zaslona a `boja_pozadine` čuva boju kojom će se crtati segmenti koji ne pripadaju niti jednom trokutu.

```

1 void initialize_data(watkins_data *w, tablica *svijet) {
2     // Isprazni liste aktivnih trokuta i bridova
3     list_init(&w->aktivni_trokuti);
4     list_init(&w->aktivni_bridovi);
5
6     // definiraj velicinu ekrana i boju pozadine
7     w->ymin = 0; w->ymax = 299;
8     w->xmin = 0; w->xmax = 299;
9     w->boja_pozadine.r = 1.0f;
10    w->boja_pozadine.g = 1.0f;
11    w->boja_pozadine.b = 0.0f;
12
13    // stvori i inicijaliziraj buckets :
14    int broj_scan_linija = w->ymax-w->ymin+1;
15    w->buckets = (list **) malloc(broj_scan_linija * sizeof(list *));
16    for(int i = 0; i < broj_scan_linija; i++) {
17        w->buckets[i] = NULL;
18    }
19
20    // Za svaki trokut :
21    for(Trokut t : svijet->trokuti()) {
22        // Popuni podatke o trokutu :
23        triangle_data *td = fillTriangleData(t);
24        addToBucket(td, w->buckets, td->y);
25    }
26}

```

Metoda `initialize_data` resetira liste aktivnih trokuta i bridova i stvara polje `buckets`. Za svaki trokut stvara se njegov opisnik (podatak tipa `triangle_data`). Računaju se koeficijenti implicitnog oblika jednadžbe ravnine u kojoj poligon leži, utvrđuju se minimalna i maksimalna y -koordinata vrhova poligona temeljem kojih se postavljaju `y` i `dy`. Element aktivovan postavlja se na `false`. Za svaki brid popunjava se struktura `edge_data`. Konačno, trokut se dodaje u odgovarajući pretinac polja `buckets` prema svojoj najmanjoj y -koordinati.

```

1 void watkins_algorithm(tablica *svijet) {
2     int y;
3     watkins_data w;

```

```

4
5 // obavi inicijalizaciju potrebnih podataka
6 initialize_data(&w, svijet);
7
8 // Idemo za svaku scan-liniju odozdo prema gore
9 for(y = w.ymin; y <= wymax; y++) {
10    // Dodaj trokute koji počinju u buckets[y] u aktivne
11    list * l = w.buckets[y - w.ymin];
12    if( !l ) {
13        for( int i = 0; i < l->size; i++ ) {
14            list_add(&w.aktivni_trokuti, list_get(l, i));
15        }
16    }
17
18    // Dodaj sve bridove aktivnih trokuta koji počinju na
19    // y u aktivne bridove
20    for( int i = 0; i < w.aktivni_trokuti.size; i++ ) {
21        triangle_data *td = list_get(&w.aktivni_trokuti, i);
22        for( int j = 0; j < 3; j++ ) {
23            if(td->bridovi[j].y == y) {
24                list_add(&w.aktivni_bridovi, &td->bridovi[j]);
25            }
26        }
27    }
28
29    // Sortiraj aktivne bridove po x-u
30    for( int i = 0; i < w.aktivni_bridovi.size; i++ ) {
31        for( int j = i+1; j < w.aktivni_bridovi.size; j++ ) {
32            int xj = list_get(&w.aktivni_bridovi, j)->x;
33            int xi = list_get(&w.aktivni_bridovi, i)->x;
34            if(xj < xi) {
35                list_swap(&w.aktivni_bridovi, i, j);
36            }
37        }
38    }
39
40    // Obradi aktivne bridove
41    process_active_edges(&w, y);
42    // Azuriraj aktivne bridove
43    update_active_edges(&w);
44    // Azuriraj aktivne trokute
45    update_active_triangles(&w);
46}
47 free_watkins_data(&w);
48}

```

Metoda `watkins_algorithm` predstavlja glavnu metodu Watkinsovog algoritma. Nakon što se napravi inicijalizacija u kojoj se stvore sve potrebne podatkovne strukture, algoritam pomiče scan-liniju od najmanjeg y -a do najvećeg dozvoljenog y -a. Za svaku scan-liniju, u listu aktivnih trokuta dodaju se trokuti koji počinju na toj scan-liniji (uporabom informacija iz polja `buckets`). Potom se prolazi kroz bridove aktivnih trokuta, i svi bridovi koji počinju na scan-liniji dodaju se u listu aktivnih bridova. Lista aktivnih bridova potom se sortira prema trenutnoj x -koordinati sjecišta sa scan-linijom (pohranjeno u varijabli x svakog brida). Ovo je važno jer se podizanjem scan-linije može promijeniti redoslijed sjecišta bridova i scan-linije. Nakon što su aktivni bridovi sortirani, prelazi se na njihovu obradu (tu se obavlja crtanje vodoravnih segmenata), te potom na ažuriranje liste aktivnih bridova i trokuta. Krenimo od ove posljednje dvije metode.

```

1 void update_active_edges(watkins_data *w) {
2     for( int i = 0; i < w->aktivni_bridovi.size; i++ ) {
3         edge_data *e = list_get(&w->aktivni_bridovi, i);
4         e->dy = e->dy - 1;
5         if( e->dy <= 0 ) {
6             list_delete(&w->aktivni_bridovi, i);
7             i--;
8         } else {

```

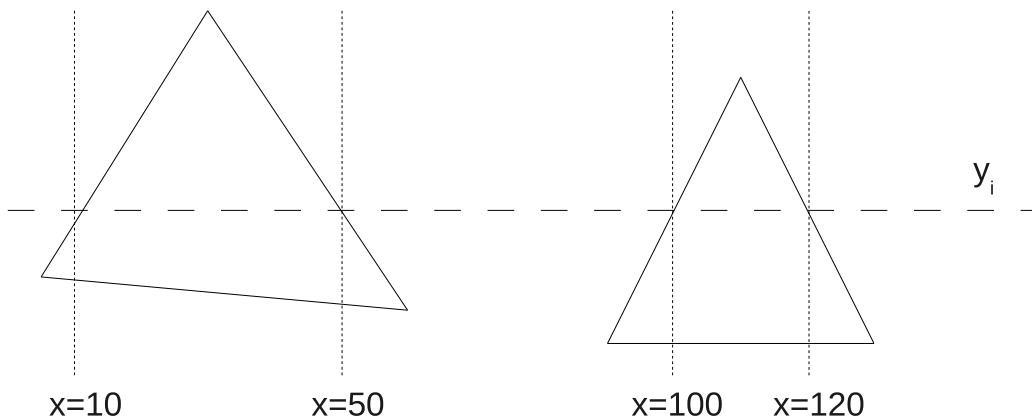
```

9         e->x = e->x + e->dx;
10    }
11 }
12 }
13
14 void update_active_triangles(watkins_data *w) {
15     for(int i = 0; i < w->aktivni_trokuti.size(); i++) {
16         triangle_data *t = list_get(&w->aktivni_trokuti, i);
17         t->dy = t->dy - 1;
18         if(t->dy <= 0) {
19             list_delete(&w->aktivni_trokuti, i);
20             i--;
21         }
22     }
23 }
```

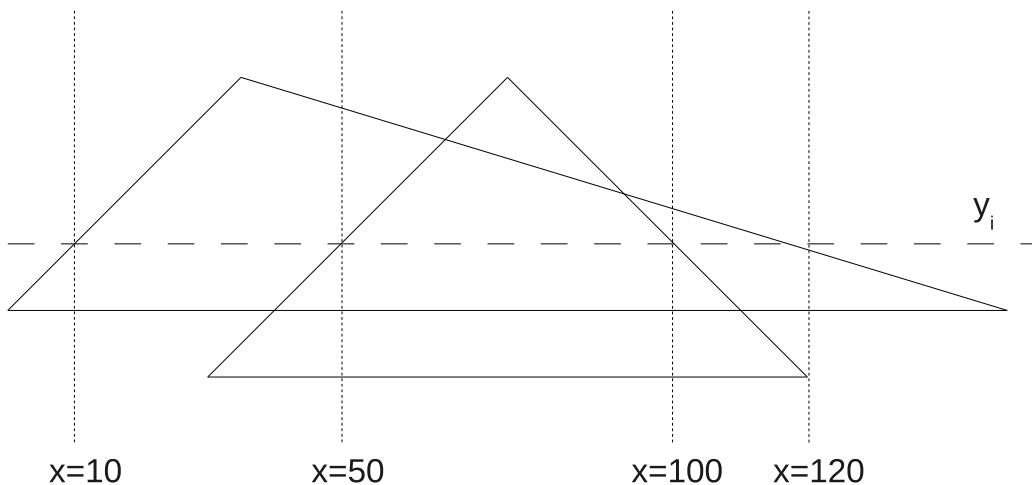
Obje metode, update_active_edges i update_active_triangles već smo načelno opisali: umanjuju dy za jedan, i ako se dode na nulu, brid odnosno trokut izbacuju se iz liste aktivnih.

```

1 void process_active_edges(watkins_data *w, int y) {
2     f_rgb_color seg_boja; // boja intervala
3     int broj_trokuta; // koliko je trokuta u segmentu?
4     edge_data *e; // brid
5     triangle_data *t; // trokut
6
7     broj_trokuta = 0;
8     w->intervalL = w->xmin; // pocetak intervala je na xmin
9
10    // Srusi zastavicu aktivan svih aktivnih trokuta
11    for(int i = 0; i < w->aktivni_trokuti.size(); i++) {
12        t = list_get(&w->aktivni_trokuti, i);
13        t->aktivan = false;
14    }
15
16    // Idemo za svaki brid (sortirani su prema x-u sjecista
17    // sa scan-linijom)
18    for(int i = 0; i < w->aktivni_bridovi.size(); i++) {
19        // Dohvati brid
20        e = (edge_data*) list_get(&w->aktivni_bridovi, i);
21        // Zatvorim interval njegovim sjecistem
22        w->intervalD = e->x;
23        // Koliko je trokuta u području tog segmenta?
24        switch(broj_trokuta) {
25            // ako nula, popuni pozadinom
26            case 0:
27                seg_boja = w->boja_pozadine;
28                break;
29            // ako je jedan, uzmi boju aktivnog trokuta
30            case 1:
31                seg_boja = color_of_active_triangle(w);
32                break;
33            // ako je vise, utvrdi boju najblizeg
34            default:
35                seg_boja = color_for_segment(w, y);
36                break;
37        }
38
39        // Promjeni zastavicu aktivan pripadnog trokuta
40        t = e->p_trokut;
41        t->aktivan = !t->aktivan;
42        // i azuriraj broj aktivnih trokuta
43        if(t->aktivan) {
44            broj_trokuta = broj_trokuta + 1;
45        } else {
46            broj_trokuta = broj_trokuta - 1;
47        }
48 }
```



Slika 7.9: Watkinsov postupak: dva disjunktna trokuta

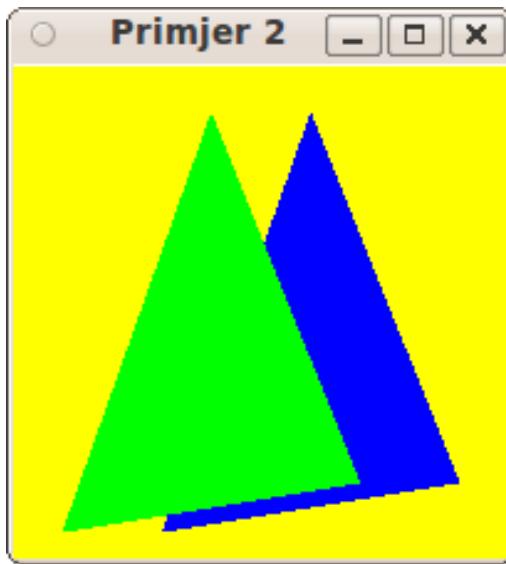


Slika 7.10: Watkinsov postupak: preklapajući trokuti

```

49     // nacrtaj vodoravni segment scan-linije
50     display_interval(w->intervalL, w->intervalD,
51                     y, &seg_boja);
52
53     // zapocni novi interval s krajem trenutnog
54     w->intervalL = w->intervalD;
55 }
56
57     // ako je ostao neiscrtan kraj scan-linije:
58     if(w->intervalL < w->xmax) {
59         display_interval(w->intervalL, w->xmax,
60                           y, &w->boja_pozadine);
61     }
62 }
```

Metoda `process_active_edges` zadužena je za iscrtavanje vodoravnih segmenata jedne scan-linije. Metoda promatra sjecišta s aktivnim bridovima i temeljem toga utvrđuje veličinu segmenta i njegovu boju. Primjerice, pretpostavimo da za trenutnu scan-liniju postoje 4 aktivna brida, čija su sjecišta sa scan-linijom $x = 10$, $x = 50$, $x = 100$ i $x = 120$. Uz dimenzije ekrana po x -osi od $x = 0$ do $x = 299$, očekujemo 5 segmenata: prvi S_1 od $x = 0$ do $x = 10$, drugi S_2 od $x = 10$ do $x = 50$, treći S_3 od $x = 50$ do $x = 100$, četvrti S_4 od $x = 100$ do $x = 120$ te konačno peti S_5 od $x = 120$ do $x = 299$. Segment S_1 bojat ćemo pozadinskom bojom - naime, on se proteže od ruba ekrana, pa do sjecišta scan-linije s prvim aktivnim brdom; varijabla `broj_trokuta` u tom segmentu bit će 0. Nailaskom na prvo sjecište (dakle, obradom prvog aktivnog brda), njegov trokut također postaje aktivan – sljedeći segment prelazit će preko tog trokuta. Zbog toga ćemo i `broj_trokuta` povećati za 1. Obradom drugog brda, odnosno



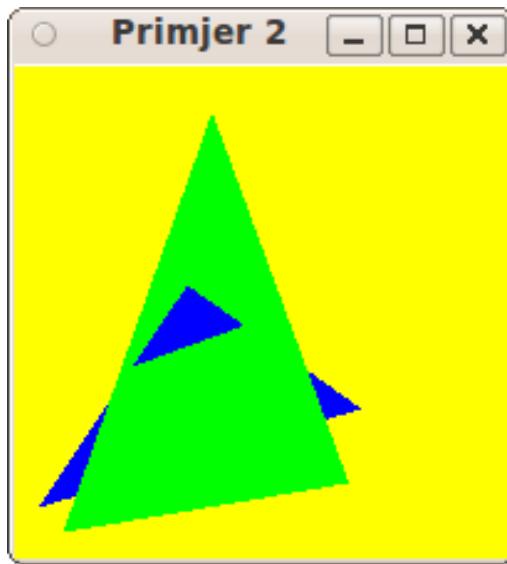
Slika 7.11: Watkinsov postupak: jednostavniji slučaj

dolaskom do sjecišta $x = 50$, razmatramo novi interval: S_2 . Ovdje su već moguća dva slučaja: ako taj brid pripada istom trokutu (kao i maloprije, vidi sliku 7.9), znači da tim sjecištem segment izlazi iz područja tog trokuta. U tom slučaju zastavicu aktivan treba spustiti i vrijednost varijable broj_trokuta smanjiti za 1. Međutim, ako brid pripada nekom drugom trokutu (vidi sliku 7.10), tada ili ulazimo i u njegovo područje – a već jesmo na području onog starog trokuta, čime i njegovu zastavicu aktivan treba postaviti na **true** i broj_trokuta uvećati za jedan, ili izlazimo iz njegovog područja, pa mu zastavicu aktivan treba postaviti na **false** i broj_trokuta umanjiti za jedan. U općem slučaju, dakako, moguće je da se scan-linija nalazi na području niti jednog, jednog ili proizvoljno mnogo trokuta. Ako se scan-linija nalazi na području primjerice dva trokuta, tada su opet moguća dva slučaja. U jednostavanom slučaju, jedan se trokut nalazi ispred drugog trokuta, i segment će biti bojan bojom onog koji je promatraču bliži. Međutim, moguć je slučaj da se ta dva trokuta u dijelu promatranog segmenta probadaju, pa segment treba podijeliti na onaj dio u kojem je vidljiv jedan trokut, te na ostatak u kojem je vidljiv drugi trokut. U općem slučaju, s više trokuta, ovih podjela može biti još i više. Pa imajući to u vidu, objasnimo što se događa u metodi `process_active_edges`. Slike 7.11 i 7.12 prikazuju rezultat iscrtavanja Watkinsovim postupkom.

Metoda započinje postavljanjem lijeve granice segmenta na x -koordinatu početka ekrana. Svim trokutima iz liste aktivnih trokuta zastavica aktivan se postavlja na **false**. broj_trokuta postavlja se na 0. Potom se gleda brid po brid iz liste aktivnih bridova. x -koordinata sjecišta promatranog brida sa scan-linijom zatvara trenutni segment. Promatra se broj_trokuta koji se nalazi ispod segmenta. Ako je to 0, bira se pozadinska boja za popunjavanje. Ako je to 1, pretražuje se lista aktivnih poligona, i vraća boja onog (jedinog) koji ima postavljenu zastavicu aktivan na **true**. Konačno, ako je to 2 ili više (slučaj **default**), poziva se pomoćna metoda koja će proanalizirati koji je slučaj nastupio, po potrebi će segment podijeliti na manje podsegmente i sve ih iscrtati osim zadnjeg, čiju će boju vratiti. Nakon toga, mijenjamo zastavicu aktivan trokuta kojemu pripada trenutni brid, te ako smo ga aktivirali, povećavamo, a ako smo ga deaktivirali, smanjujemo broj_trokuta. Potom crtamo segment utvrđenom bojom, i započinjemo novi segment od kraja trenutnog segmenta.

```

1 f_rgb_color color_of_active_triangle(watkins_data *w) {
2     for( int i = 0; i < w->aktivni_trokuti.size(); i++) {
3         triangle_data *t = list_get(&w->aktivni_trokuti, i);
4         if(t->aktivan) {
5             return t->boja;
6         }
7     }
8     // Ovdje ne smijemo sticati!
9     return w->boja_pozadine;
10 }
```



Slika 7.12: Watkinsov postupak: trokuti koji se probadaju

```

11 f_rgb_color color_for_segment(watkins_data *w, int y) {
12     // Stog dovoljno velik da primi sva sjecista ...
13     int broj_sjecista =
14         w->aktivni_trokuti.size*w->aktivni_trokuti.size;
15     float* stog = (float*) malloc( broj_sjecista*sizeof( float ) );
16     int stog_size;
17     bool imam_presjek;
18     float x_presjeka;
19     f_rgb_color interval_boja;
20
21     x_presjeka = w->intervalL;
22     stog_size = 0;
23     while(1) {
24         checkForIntersectionsInSegment(
25             w, &imam_presjek, &x_presjeka, y);
26         if(imam_presjek) {
27             stog[stog_size++] = w->intervalD;
28             w->intervalD = x_presjeka;
29         } else {
30             interval_boja =
31                 color_of_min_active_z_value_triangle(
32                     w, (w->intervalL+w->intervalD)/2.0, y);
33             if(stog_size==0) {
34                 free(stog);
35                 return interval_boja;
36             }
37             display_interval(w->intervalL, w->intervalD,
38                             y, &interval_boja);
39             w->intervalL = w->intervalD;
40             w->intervalD = stog[--stog_size];
41         }
42     }
43 }
44 }
```

Metodu `color_of_active_triangle` vraća iz liste aktivnih trokuta boju trokuta koji ima postavljenu zastavicu aktivan na `true`. Pretpostavka je da takav postoji samo jedan. Metoda `color_for_segment` koristi se za analizu složenijih slučajeva, gdje se u području segmenta nalazi više trokuta. Ideja metode jest skraćivati segment koji je početno zadan varijablama `w->intervalL` i `w->intervalD` na manji segment, tako dugo dok se ne dođe do situacije da se u promatranom segmentu više ne događa probadanje trokuta – pa točno možemo utvrditi koji je trokut u tom segmentu iznad kojeg trokuta. Evo kako se to

radi. Metoda ulazi u petlju i traži postoji li u trenutnom segmentu (što je na početku onaj originalni, no u sljedećem prolazu to već može biti i skraćeni segment) sjecište bridova aktivnih trokuta. Ako postoji, trenutni kraj segmenta gura na stog (kako bi se kasnije moglo nastaviti), a kao kraj trenutnog segmenta uzima to sjecište. Postupak se ponavlja s tako skraćenim segmentom.

Kada se utvrdi da u promatranom segmentu više nema sjecišta, kao boja za segment se uzima boja onog aktivnog trokuta iz liste aktivnih trokuta koji je u tom segmentu najbliži promatraču. Ovo se jednostavno ispita: naime, budući da znamo jednadžbu ravnine u kojoj trokut leži, kao x koordinatu uzmemo sredinu segmenta, y je zadan scan-linijom, i iz jednadžbe ravnine očitamo pripadnu z -koordinatu. Ako je u tom trenutku stog prazan, vraćamo tu boju, i puštamo pozivatelja da iscrta taj segment. Ako stog nije prazan, tada metoda crta pronađeni podsegment, te započinje novi segment koji počinje na mjestu gdje je trenutni završio, a kraj se vadi sa stoga. S tim novim segmentom postupak se ponavlja. Metoda koja pronalazi boju najbližeg aktivnog trokuta prikazana je u nastavku.

```

1 f_rgb_color color_of_min_active_z_value_triangle(
2     watkins_data *w, float x, int y) {
3
4     f_rgb_color *boja = NULL;
5     double z = 0.0;
6
7     for (int i = 0; i < w->aktivni_trokuti.size; i++) {
8         triangle_data *t = list_get(&w->aktivni_trokuti, i);
9         if (t->aktivan) {
10             double z_trenutni = -(t->a*x+t->b*y+t->d)/t->c;
11             // gledam izishodista prema -z; veci z mi je blizi
12             if (boja == NULL || z_trenutni > z) {
13                 boja = &t->boja;
14                 z = z_trenutni;
15             }
16         }
17     }
18     if (boja == NULL) {
19         printf("Greska: nema boje.\n");
20     }
21     return boja == NULL ? w->boja_pozadine : *boja;
22 }
```

Konačno, metoda koja traži sjecišta opisana je u nastavku, u malo naglašenijem pseudokodu.

```

1 void checkForIntersectionsInSegment(
2     watkins_data *w, bool *imam_presjek,
3     float *x_presjeka, int y) {
4
5     za_svaki_aktivni_trokut t {
6         za_svaki_aktivni_brid t.b {
7             (x, z) = brid_point(t.b, y);
8         }
9     }
10
11    // sada (potencijalno) za svaki trokut imamo segment određen
12    // s dvije tocke koje se nalaze na dva aktivna brida kroz
13    // koje prolazi scan-linija
14
15    // pogledaj sjecista tih segmenata (voditi računa da to nisu
16    // pravci, vec dijelovi pravaca
17
18    // ako postoji sjeciste cija je x-koordinata x_sjec i ako za
19    // nju vrijedi: w->intervalL < x_sjec < w->intervalD
20    if (postoji_opisano_sjeciste) {
21        *imam_presjek = true;
22        *x_presjeka = x_sjec;
23    } else {
24        *imam_presjek = false;
25        *x_presjeka = w->intervalL;
26    }
27 }
```

Metoda checkForIntersectionsInSegment programski je najzahtjevnija, no konceptualno je vrlo jednostavna. Zato je odabran opis koji je više kroz komentare a manje programerski čist. Ideja je najprije pronaći za sve aktivne bridove točke u kojima ih scan-linija siječe. Kako znamo y -koordinatu, x i z -koordinate možemo dobiti direktno iz parametarskog oblika jednadžbe brida. Za svaki trokut, općenito govoreći, postojat će ili nula, ili dva takva sjecišta. Ako radimo samo s aktivnim bridovima, onda ćemo sigurno imati dva sjecišta (oba uz isti y), i te dvije točke u xz -ravnini određuju segment pravca. Nakon što pronađemo sve segmente, kod za svaki par segmenata provjerava sijeku li se, i ako da, pada li x -koordinata u granice trenutnog segmenta koji želimo iscrtati. Ako takvo sjecište postoji, može se vratiti odmah prvo; naime, sjetimo se da će pozivatelj to sjecište iskoristiti da skrati interval, i potom na kraćem intervalu opet pozvati ovu provjeru. Za kraj je ostalo još prikazati i najjednostavniju metodu - onu koja uporabom *OpenGL-a* radi crtanje vodoravnog segmenta.

```

1 void display_interval( float x0, float x1,
2   float y, f_rgb_color *boja) {
3
4   glColor3f( boja->r, boja->g, boja->b );
5   glBegin(GL_LINE_STRIP);
6   glVertex2f(x0, y);
7   glVertex2f(x1, y);
8   glEnd();
9 }
```

Primjer: 11

Zadana su dva trokuta u 3D prostoru. Vrhovi prvog (zelenog) su: $A = (20 \ 10 \ -10)$, $B = (80 \ 180 \ -20)$ i $C = (135 \ 30 \ -15)$; vrhovi drugog (plavog) su: $D = (10 \ 20 \ -30)$, $E = (70 \ 110 \ -5)$ i $F = (140 \ 60 \ -40)$. Boja pozadine je žuta. Provedite Watkinsov postupak za scan-liniju $y=80$. Pretpostavite da se slika iscrtava na pravokutnom području (0,199) po obje koordinate.

Rješenje:

Kako se u primjeru zahtjeva izračun samo za jednu liniju, ne znamo kakvo je stanje u tablicama aktivnih bridova i trokuta – naime, postupak nismo provodili od početka. Pa idemo najprije rekonstruirati to stanje. Bridovi i trokuti s kojima radi algoritam, te početne vrijednosti koje se za njih pamte prikazane su u tablicama u nastavku.

Brid	y	dy	x	dx	Trokut	
AB	10	170	20	0.352941	T1	
BC	30	150	135	-0.366667	T1	
CA	10	20	20	5.75	T1	
DE	20	90	10	0.666667	T2	
EF	60	50	140	-1.4	T2	
FD	20	40	10	3.25	T2	
Trokut	y	dy	a	b	c	d
T1	10	170	-650	-850	-18350	-162000
T2	20	90	-1900	3850	-9300	-337000

Za $y = 80$ vidimo da će oba trokuta biti u listi aktivnih trokuta (prvi počinje na $y = 10$ i proteže se sljedećih 170 scan-linija, a drugi počinje na $y = 20$ i proteže se sljedećih 90 scan-linija; dakle, na $y = 80$ oba su u listi aktivnih trokuta). Od bridova, u listi aktivnih bridova bit će AB, BC, DE i EF. Sjecišta tih bridova sa scan-linijom $y = 80$ prikazana su u sljedećoj tablici.

Brid	Pripada trokutu	$x_{sjec} = b.x + (y - b.y) \cdot b.dx$
AB	T1	$20 + (80 - 10) \cdot 0.352941 = 44.7059$
BC	T1	$135 + (80 - 30) \cdot -0.366667 = 116.6667$
DE	T2	$10 + (80 - 20) \cdot 0.666667 = 50.0000$
EF	T2	$140 + (80 - 60) \cdot -1.4 = 112.0000$

Alternativno, sjecišta x -koordinatu sjecišta mogli smo direktno računati iz parametarskog oblika jednadžbi bridova, tako da najprije temeljem poznatog y -a odredimo parametar, a zatim za taj parametar odredimo preostale koordinate. Nakon što smo definirali sadržaj liste aktivnih bridova, bridove je potrebno sortirati prema x -koordinati sjecišta. Tada dobivamo tablicu prikazanu u nastavku.

Brid	Pripada trokutu	x_{sjec}
AB	T1	44.7059
DE	T2	50.0000
EF	T2	112.0000
BC	T1	116.6667

Algoritam kreće s broj_trokuta=0 te intervalL=0.

Korak 1. Gleda se prvi brid (AB), i kraj segmenta se postavlja na intervalD=44.7059. Kako je broj_trokuta=0, za popunjavanje segmenta bira se pozadinska boja (žuta). Postavlja se T1.aktivovan=true i broj_trokuta=1. Boja se interval (0, 44.7059) žutom. Početak intervala postavlja se na intervalL=44.7059.

Korak 2. Gleda se drugi brid (DE), i kraj segmenta se postavlja na 50.0000. Kako je broj_trokuta=1, traži se jedini trokut koji ima postavljenu zastavicu aktivovan (to će biti T1), i za popunjavanje se bira njegova boja (zelena). Postavlja se T2.aktivovan=true i povećava se broj_trokuta=2. Boja se interval (44.7059, 50.0000) zelenom. Početak intervala postavlja se na intervalL=50.0000.

Korak 3. Gleda se treći brid (EF). Kako je broj_trokuta=2, provjerava se postoji li u intervalu $x \in (50, 112)$ kakvo sjecište. Presjek prvog trokuta i ravnine $y = 80$ je segment pravca u xz -ravnini $(44.705883, -14.117647) - (116.666664, -16.666666)$. Presjek drugog trokuta i ravnine $y = 80$ je segment pravca u xz -ravnini $(50.000000, -13.333333) - (112.000000, -26.000000)$. Ova dva segmenta sijeku se u točki koja ima $x = 55.7547$. Kako je ovo unutar promatranog intervala, vraća se pronadeno sjecište. Zbog toga se aktualni kraj intervala gura na stog (112.0000), a kao novi kraj se postavlja pronadeno sjecište 55.754688. Ponovno se provjerava postoji li kakvo sjecište unutar tog skraćenog intervala, i odgovor je ne. Tada se kao testna točka uzima sredina intervala $(50 + 55.7547)/2 = 52.8773$. Za tu točku ($x = 52.8773$, $y = 80$) računa se pripadna z -koordinata u ravnini svakog od aktivnih trokuta, i vraća se boja onog koji je najbliži promatraču (trokut čija točka ima najveću z -koordinatu – promatrač je u ishodištu). U ovom slučaju to će biti plava boja. Kako stog nije prazan, crta se trenutni interval $(50, 55.7547)$ plavom bojom. Početak intervala postavlja se na trenutni kraj intervalL=55.7547, a kraj se vadi sa stoga intervalL=112.0000. Za zadani interval provjerava se postoji li koje sjecište u xz -ravnini koje pada unutar tog intervala (rubovi su isključeni). Kako takvo sjecište ne postoji (bridovi se sijeku u $x = 55.7547$), traži se trokut za koji je središnja točka intervala $x = (55.7547+112)/2 = 83.8774$, $y = 80$ najbliža promatraču (ima najveću z -koordinatu) i uzima se njegova boja i vraća (to će biti zelena). Ovime smo ponovno u glavnoj metodi koja sada vidi skraćeni segment $(55.7547, 112)$ i ima za njega zelenu boju. Kako trenutni brid pripada trokutu T2, mijenja se njegova zastavica T2.aktivivan=false i umanjuje broj_trokuta=1. Boja se interval $(55.7547, 112.0000)$ zelenom. Početak intervala postavlja se na intervalL=112.0000.

Korak 4. Gleda se četvrti brid (BC), i kraj segmenta se postavlja na 116.6667. Kako je broj_trokuta=1, traži se jedini trokut koji ima postavljenu zastavicu aktivavan (to će biti T1), i za popunjavanje se bira njegova boja (zelena). Mijenja se T1.aktivivan=false i umanjuje se broj_trokuta=0. Boja se interval $(112.0000, 116.6667)$ zelenom. Početak intervala postavlja se na intervalL=116.6667.

Korak 5. Kako više nema bridova, izlazi se iz petlje. Kraj segmenta postavlja se na desni rub ekrana (199), i segment $(116.6667, 199)$ boja se pozadinskom bojom (žutom). Ovime završava obrada te scan-linije.

7.5 Z-spremnik

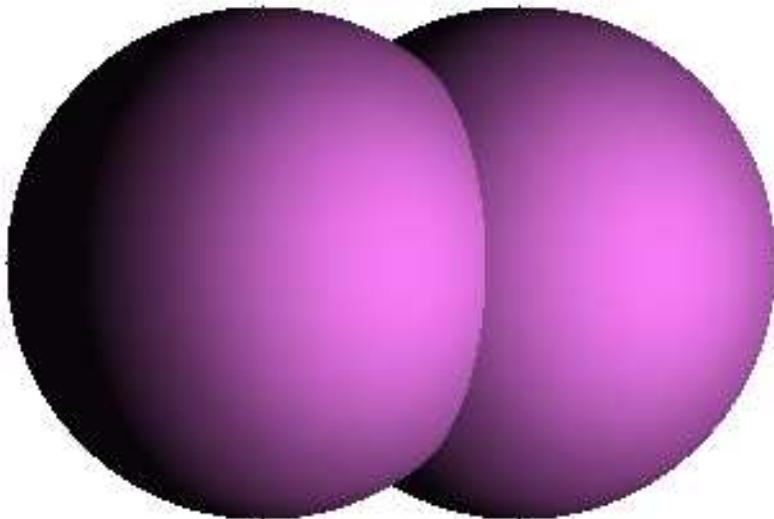
Z-spremnik (engl. *z-buffer*) je postupak koji provjeru preklapanja radi na razini točaka. Ideja je sljedeća. Slika se iscrtava na zaslon konačnih dimenzija. Tada se prilikom iscrtavanja točke na zaslon u z-spremniku pamti udaljenost izvorne točke od promatrača (z -koordinata točke nakon projekcije). Npr. potrebno je iscrtati točku (x,y,z) . Radi se projekcija točke i dobije se točka (x',y',z') . Na zaslon treba iscrtati točku na poziciji (x',y') . No prije nego što nacrtamo tu točku, potrebno je u z-spremniku pogledati je li na tu poziciju već nacrtana neka druga točka, i ako je, koliko je ona daleko bila od promatrača. Ako je naša točka bliža, tada ona skriva prethodno iscrtanu točku (jer je bliže promatraču), pa je crtamo i u z-spremnik upisujemo na mjesto (x',y') udaljenost naše točke. Ako je naša točka dalja, tada je ne crtamo, i sadržaj z-spremnika ne mijenjamamo.

U praktičnim izvedbama z-spremnik implementira se kao polje dimenzija $\text{rezolucija}_x \cdot \text{rezolucija}_y$. Hoće li to biti polje tipa **short**, **int** ili nešto treće, ovisi o postavkama. Inicijalno se svi elementi polja postave na najveće vrijednosti (u smislu razmišljanja: do sada smo "iscrtavali" jedino točke u beskonačnosti koje su iza svih drugih). Prostor kojeg smo odredili prednjom i stražnjom ravninom odsijecanja određuje raspon koji će se preslikati na raspon z-spremnika. Ovdje treba pripaziti na moguće pogreške. Ako na primjer odredimo prednju ravninu odsijecanja na 1, stražnju na 1000, a naš objekt ima z koordinatu u rasponu od 1, 2 - 1, 3 očito je da nismo dobro odredili raspon spremnika i da će pogreške uslijed zaokruživanja utjecati na rezultat. Sklopovska podrška za z-spremnik već je dulje vrijeme je nezaobilazna na računalima i svakako ju treba koristiti. U OpenGL-u prvo je potrebno inicijalizirati spremnik, pa ako koristimo i dvostruki spremnik opisan u prvom poglavljiju potrebno je:

```
1 glutInitDisplayMode(GLUT_DOUBLE|GLUT_DEPTH);
```

Dalje je potrebno omogućiti korištenje spremnika dubine i odabrati uvjet ispitivanja. Ako je inicijalno u spremniku najveća vrijednost tada ćemo provjeravati je li nešto bliže od te postavljene vrijednosti. Z-spremnik možemo koristiti i za razne druge namjene pa su i različiti uvjeti koji mogu biti postavljeni pri navedenoj provjeri. Treba još obratiti pažnju da prilikom brisanja spremnika s bojom obrišemo i z-spremnik. Ako to ne učinimo iscrtavanje pomaknutog objekta obavljat će se uz stare neispravne vrijednosti u z-spremniku pa će i prikaz animacije biti vrlo neobičan.

```
1 glEnable(GL_DEPTH_TEST);
2 glDepthFunc(GL_LESS);
3 ...
4 void display()
5 {
6     ...
7     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
8 }
```



Slika 7.13: Dvije kugle prikazane uporabom z-spremnika

Najveći nedostatak ovakvog pristupa je u zahtjevima za memorijom. Ako radimo na nekom ugrađenom sustavu ili sustavu na kojem imamo memorija ograničenja posebnu pažnju morat ćemo posvetiti zadanim ograničenjima. Npr. odlučimo li se na prikaz u rezoluciji 1024×768 piksela, uz maksimalnu dubinu scene takvu da možemo koristiti tip **short** (2 okteta po podatku), potrebna količina memorije iznosi $1024 \cdot 768 \cdot 2 = 1572864$ okteta, što je poprilici 1.5 MB. Zahtjevi za memorijom mogu još porasti ako u z-spremnik odlučimo upisivati više informacija. Naime, ako ga koristimo za spremanje dubine, a točke iscrtavamo na zaslonu, tada ćemo kod vrlo kompleksnih scena kojima treba dosta vremena za iscrtavanje uočiti kako se neki pikseli pojavljuju, pa ih zamjenjuju drugi pikseli dobiveni od bližih objekata. Da bi se ovo izbjeglo, z-spremnik možemo koristiti tako da pamti i dubinu pojedine točke, i njezinu boju (samo se iscrtavanje u ovom koraku ne radi na zaslon). Jednom kada smo čitavu scenu iscrtali u z-spremnik, čitavu sliku možemo u jednom prolazu iz z-spremnika osvježavati na zaslonu. No ovakav postupak za svaku točku z-spremnika zahtjeva 2 okteta za dubinu + 3 okteta za boju što daje 5 okteta po točki, ili uz rezoluciju 1024×768 iznos od 3932160 okteta odnosno 3.75 MB.

Ove značajne memorijske zahtjeve možemo ublažiti višeprolaznim iscrtavanjem scene: npr. u prvom prolazu iscrtati ćemo gornju polovicu zaslona, a u drugom prolazu donju polovicu. Tada nam treba z-spremnik veličine za pola zaslona. Općenito, n -prolazno iscrtavanje zahtjeva samo n -ti dio z-spremnika koji bi trebao za jednoprolavno iscrtavanje, ali zato postupak traje n puta dulje. Razvijen je i *scan line z-buffer* algoritam koji iscrtava liniju po liniju zaslona. Međutim, efikasna primjena ovog algoritma zahtjeva velike izmjene u načinu pamćenja objekata.

Kod Phongovog modela osvjetljavanja prikazali smo funkciju koja je nacrtala kuglu i osjenčala je. U nastavku ćemo najprije pokazati dio programa koji će osjenčati dvije kugle koje se međusobno probadaju koristeći Phongov model i našu vlastitu implementaciju z-spremnika. Rezultat rada prikazan je na slici 7.13 a izvorni kod naveden je u nastavku. Centar desne kugle nalazi se nešto ispod centra lijeve kugle. Konkretno, lijeva kugla ima centar u točki $(150, 150, 0)$, dok desna kugla ima centar u točki $(250, 150, -50)$.

Ispis 7.1: Crtanje dviju kugli uz z-spremnik

```
1 // Struktura tocke / 3D vektora
2 typedef struct {
```

```

3     double x;
4     double y;
5     double z;
6 } Point3D;
7
8 // Normiranje vektora
9 void normalize( Point3D *pnt ) {
10    double norm = sqrt(
11        pnt->x*pnt->x + pnt->y*pnt->y + pnt->z*pnt->z );
12    pnt->x = pnt->x / norm;
13    pnt->y = pnt->y / norm;
14    pnt->z = pnt->z / norm;
15 }
16
17 // Skalarni produkt dvaju vektora
18 double scalarProduct( Point3D *a, Point3D *b ) {
19    return a->x*b->x + a->y*b->y + a->z*b->z ;
20 }
21
22 // Pomice predanu tocku za trazen i vektor
23 double add( Point3D *p, Point3D *delta ) {
24    p->x += delta->x;
25    p->y += delta->y;
26    p->z += delta->z;
27 }
28
29 // Struktura z-spremnika
30 typedef struct {
31     int w;
32     int h;
33     short *buffer;
34 } z_buffer;
35
36 // Konstruktor z-spremnika
37 z_buffer *new_z_buffer( int w, int h ) {
38     z_buffer *z_buf = ( z_buffer* ) malloc( sizeof( z_buffer ) );
39     if( z_buf==NULL ) return NULL;
40     z_buf->buffer = ( short* ) malloc( w*h*sizeof( short ) );
41     if( z_buf->buffer==NULL ) {
42         free( z_buf );
43         return NULL;
44     }
45     z_buf->w = w;
46     z_buf->h = h;
47     return z_buf;
48 }
49
50 // Oslobananje z-spremnika
51 void free_z_buffer( z_buffer *z_buf ) {
52     free( z_buf->buffer );
53     free( z_buf );
54 }
55
56 // Brisanje sadrzaja z-spremnika
57 void clear_z_buffer( z_buffer *z_buf ) {
58     int x,y;
59
60     for( y=0; y<z_buf->h; y++ ) {
61         for( x=0; x<z_buf->w; x++ ) {
62             z_buf->buffer[ y*z_buf->w+x ] = -32000;
63         }
64     }
65 }
66
67 // Provjera smije li se iscrtati predana tocka; ako da, njezina
68 // z-koordinata odmah ce se upisati i u sam spremnik.

```

```

69  bool can_update_z_buffer( z_buffer *z_buf, Point3D *p) {
70      int pomak = z_buf->w*zaokruzi(p->y) + zaokruzi(p->x);
71      int z = zaokruzi(p->z);
72      bool result = z > z_buf->buffer [pomak];
73      if(result) {
74          z_buf->buffer [pomak] = z;
75      }
76      return result;
77  }
78
79 // Metoda za crtanje kugle , uz pretpostavku da se promatra iz
80 // smjera pozitivne z-osi.
81 void bojajKuglu( const int R, Point3D *cntr , Point3D *l ,
82                   Point3D *v, z_buffer *z_buf) {
83     int x, y; // indeksi petlje
84     double Id, Is; // difuzna i reflektirajuca komponenta
85     float I; // ukupni intenzitet
86     double ln_scalar; // pomocna varijabla
87     Point3D p; // tocka na kugli
88     Point3D n; // normala u tocki p
89     Point3D r; // vektor reflektirane komponente
90
91     glBegin(GL_POINTS);
92     for( x=-R; x<=R; x++ ) {
93         for( y=-R; y<=R; y++ ) {
94             // odredi tocku ishodisne kugle
95             p.x = x; p.y = y;
96             p.z = R*R - (p.x*p.x + p.y*p.y);
97             if( p.z < 0 ) continue;
98             p.z = sqrt(p.z);
99             // odredi normalu
100            n = p;
101            normalize(&n);
102            // racunaj difuznu komponentu
103            ln_scalar = scalarProduct(l, &n);
104            Id = ln_scalar;
105            if( Id > 0.0 ) Id = 200*Id; else Id = 0.0;
106            // racunaj reflektirajuca zraku i komponentu
107            r.x=2*ln_scalar*n.x-l->x;
108            r.y=2*ln_scalar*n.y-l->y;
109            r.z=2*ln_scalar*n.z-l->z;
110            normalize(&r);
111            Is = scalarProduct(&r, v);
112            if( Is > 0.0 ) {
113                Is = 45*pow(Is, 1.1);
114            } else Is=0.0;
115            // ukupni intenzitet
116            I = (float)( (10.0 + Id + Is)/255.0 );
117            // translatiraj tocku obzirom na pravi centar
118            add(&p, cntr);
119            // nacrtaj ako smijes
120            if(can_update_z_buffer(z_buf, &p)) {
121                glColor3f((float)I, (float)(I/2), (float)I);
122                glVertex2i(zaokruzi(p.x), zaokruzi(p.y));
123            }
124        }
125    }
126    glEnd();
127 }
128
129 void renderScene() {
130     const int R = 100;
131     Point3D l = {1, 0, 1}; // vektor prema izvoru
132     Point3D v = {0, 0, 1}; // vektor prema gledatelju
133     Point3D c0 = {150, 150, 0}; // centar prve kugle
134     Point3D c1 = {250, 150, -50}; // centar druge kugle

```

```

135
136     // normiranje vektora
137     normalize(&l);
138     normalize(&v);
139
140     // zauzmi spremnik
141     z_buf = new_z_buffer(400, 300);
142     if( z_buf==NULL ) {
143         printf("Nema dovoljno memorije.");
144         return;
145     }
146
147     // očisti z-spremnik
148     clear_z_buffer(z_buf);
149
150     // nacrtaj obje kugle
151     glPointSize(1);
152     bojajKuglu(R, &c0, &l, &v, z_buf);
153     bojajKuglu(R, &c1, &l, &v, z_buf);
154
155     // osloboodi spremnik
156     free_z_buffer(z_buf);
157 }
```

Program je potrebno pratiti od metode renderScene. U toj metodi najprije definiramo vektore \vec{L} (prema izvoru svjetla) i \vec{V} (prema gledatelju), te centre obje kugle. Potom stvaramo strukturu z-spremnika, čistimo ga, pozivamo crtanje obje kugle i konačno oslobađamo z-spremnik. Svaki od ovih poslova obavlja po jedna pomoćna metoda koja je također prikazana u izvornom kodu. Pri implementaciji z-spremnika u ovoj funkciji točka je bliža promatraču što joj je z -koordinata veća (jer je promatrač na z -osi u pozitivnoj beskonačnosti). Zbog toga je inicijalno z-spremnik popunjen s negativnim vrijednostima.

Isti program u nastavku je prikazan uporabom OpenGL-ove implementacije z-spremnika. Rezultat toga je značajno pojednostavljenje koda. Međutim, uključivanje OpenGL-ove implementacije unijelo je promjene kroz čitav program, pa je zbog toga u ispisu 7.2 prikazan čitav kod. Promjene započinju od metode main gdje se može vidjeti promjena u inicijalizaciji, uključivanje podrške za z-spremnik te definiranje načina provođenja usporedbe z -koordinata. Kako bi se dobila ista slika kao i u ona generirana ispisom 7.1, smjer gledanja je trebalo okrenuti iz $+z$ u $-z$. Stoga je u metodi display dodana matrica m koja okreće predznak z -koordinati, i nakon što je učitana jedinična matrica, ona je pomnožena tom matricom. U istoj metodi ažuriran je i poziv funkcije brisanja, koja sada briše i stanje z-spremnika. Promijenjena je i metoda reshape, točnije poziv metode glOrtho, koji sada sa zadnjima dva parametra specificira z -koordinate bližeg i daljeg kraja kojim se definira volumen pogleda (vrijednosti su postavljene na 300 – bliži kraj, te -300 – dalji kraj).

Ispis 7.2: Crtanje dviju kugli uz z-spremnik OpenGL-a

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <GL/glut.h>
5
6 void reshape(int width, int height);
7 void display();
8 void renderScene();
9
10 int main(int argc, char **argv) {
11     glutInit(&argc, argv);
12     glutInitDisplayMode(GLUT_DOUBLE|GLUT_DEPTH);
13     glutInitWindowSize(400, 300);
14     glutInitWindowPosition(0, 0);
15     glutCreateWindow("Primer 9");
16     glutDisplayFunc(display);
17     glutReshapeFunc(reshape);
18     glEnable(GL_DEPTH_TEST);
```

```

19     glDepthFunc(GL_LESS);
20     glutMainLoop();
21 }
22
23 void display() {
24     GLdouble m[] = {1, 0, 0, 0, 0, 1, 0, 0, 0, 0, -1, 0, 0, 0, 0, 1};
25     glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
26     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
27     glLoadIdentity();
28     glMultMatrixd(m);
29     renderScene();
30     glutSwapBuffers();
31 }
32
33 void reshape(int width, int height) {
34     glMatrixMode(GL_PROJECTION);
35     glLoadIdentity();
36     glOrtho(0, width-1, 0, height-1, 300, -300);
37     glViewport(0, 0, (GLsizei)width, (GLsizei)height);
38     glMatrixMode(GL_MODELVIEW);
39 }
40
41 int zaokruzi(double d) {
42     if(d>=0) return (int)(d+0.5);
43     return (int)(d-0.5);
44 }
45
46 // Struktura tocke / 3D vektora
47 typedef struct {
48     double x;
49     double y;
50     double z;
51 } Point3D;
52
53 // Normiranje vektora
54 void normalize(Point3D *pnt) {
55     double norm = sqrt(pnt->x*pnt->x + pnt->y*pnt->y
56                         + pnt->z*pnt->z);
57     pnt->x = pnt->x / norm;
58     pnt->y = pnt->y / norm;
59     pnt->z = pnt->z / norm;
60 }
61
62 // Skalarni produkt dvaju vektora
63 double scalarProduct(Point3D *a, Point3D *b) {
64     return a->x*b->x + a->y*b->y + a->z*b->z;
65 }
66
67 // Pomice predanu tocku za trazenji vektor
68 double add(Point3D *p, Point3D *delta) {
69     p->x += delta->x;
70     p->y += delta->y;
71     p->z += delta->z;
72 }
73
74 // Metoda za crtanje kugle, uz pretpostavku da se promatra iz
75 // smjera pozitivne z-osi.
76 void bojajKuglu(const int R, Point3D *cntr, Point3D *l,
77                  Point3D *v) {
78     int x, y; // indeksi petlje
79     double Id, Is; // difuzna i reflektirajuca komponenta
80     float I; // ukupni intenzitet
81     double ln_scalar; // pomocna varijabla
82     Point3D p; // tocka na kugli
83     Point3D n; // normala u tocki p
84     Point3D r; // vektor reflektirane komponente

```

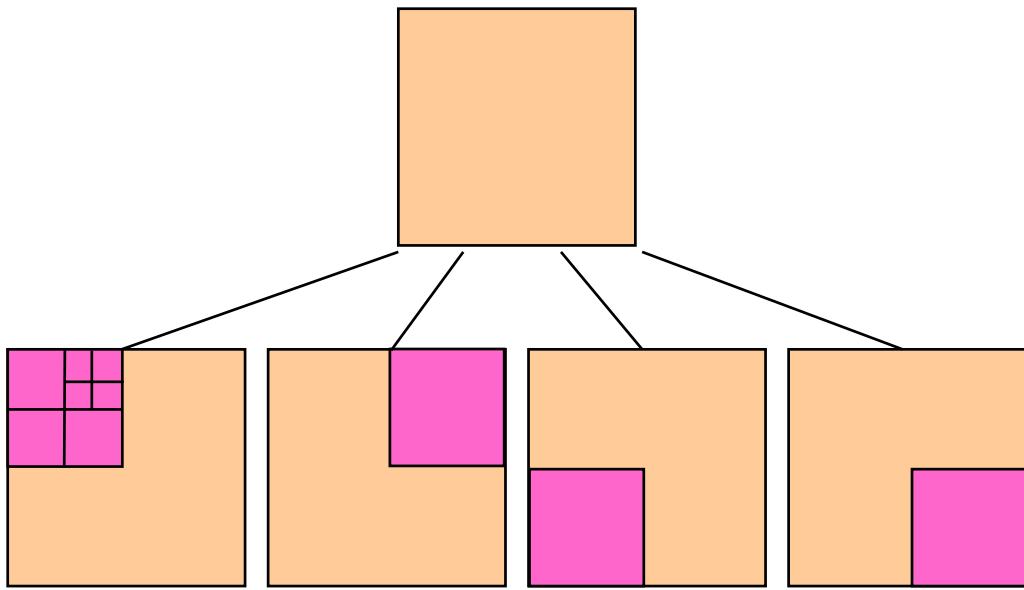
```

85     glBegin(GL_POINTS);
86     for( x=-R; x<=R; x++ ) {
87         for( y=-R; y<=R; y++ ) {
88             // odredi tocku ishodisne kugle
89             p.x = x; p.y = y;
90             p.z = R*R - (p.x*p.x + p.y*p.y);
91             if( p.z < 0 ) continue;
92             p.z = sqrt(p.z);
93             // odredi normalu
94             n = p;
95             normalize(&n);
96             // racunaj difuznu komponentu
97             ln_scalar = scalarProduct(1, &n);
98             Id = ln_scalar;
99             if( Id > 0.0 ) Id = 200*Id; else Id = 0.0;
100            // racunaj reflektiraju cu zraku i komponentu
101            r.x=2*ln_scalar*n.x-1->x;
102            r.y=2*ln_scalar*n.y-1->y;
103            r.z=2*ln_scalar*n.z-1->z;
104            normalize(&r );
105            Is = scalarProduct(&r , v);
106            if( Is > 0.0 ) {
107                Is = 45*pow(Is , 1.1);
108            } else Is=0.0;
109            // ukupni intenzitet
110            I = (float)( (10.0 + Id + Is)/255.0 );
111            // translatiraj tocku obzirom na pravi centar
112            add(&p, cntr);
113            // nacrtaj; z-spremnik ce provjeriti je li to OK
114            glColor3f((float)I, (float)(I/2), (float)I);
115            glVertex3i(zaokruzi(p.x), zaokruzi(p.y),
116                         zaokruzi(p.z));
117        }
118    }
119    glEnd();
120 }
121
122 void renderScene() {
123     const int R = 100;
124     Point3D l = {1, 0, 1}; // vektor prema izvoru
125     Point3D v = {0, 0, 1}; // vektor prema gledatelju
126     Point3D c0 = {150, 150, 0}; // centar prve kugle
127     Point3D c1 = {250, 150, -50}; // centar druge kugle
128
129     // normiranje vektora
130     normalize(&l);
131     normalize(&v);
132
133     // nacrtaj obje kugle
134     glPointSize(1);
135     bojajKuglu(R, &c0, &l, &v);
136     bojajKuglu(R, &c1, &l, &v);
137 }
138 }
```

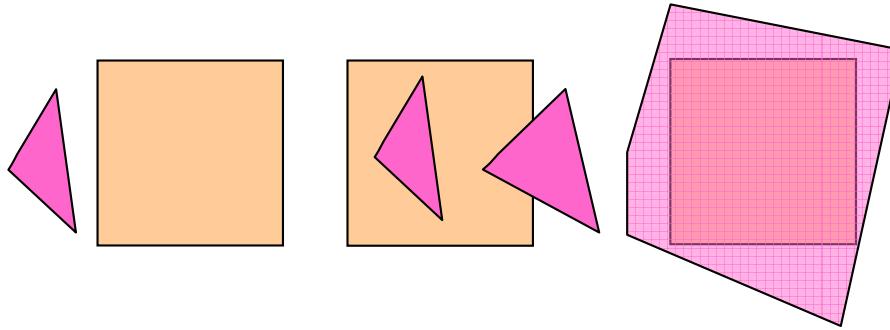
7.6 Postupak Warnocka

Postupak Warnocka teži tome da se područje ispitivanja udaljenosti objekta do promatrača poveća. Kod postupka ispitivanja z-spremnika provjerava se jedan slikovni element, kod Watkinsovog postupak provjerava se ispitna linija, dok u postupku Warnocka područje želimo povećati na kvadrat za koji ćemo moći odlučiti što je vidljivo, odnosno općenito što se nalazi u njemu.

Inicijalno, u postupku Warnocka, u početni čvor postavimo listu svih poligona u sceni. Početno područje zaslona, koje ćemo zvati prozor, dijelimo na četiri pravilna dijela stvarajući nove prozore i



Slika 7.14: Warnockov postupak: rekurzivna podjela prostora na četiri dijela



Slika 7.15: Warnockov postupak mogući slučajevi: poligon je izvan prozora (1), poligon je u prozoru ili siječe prozor (2), poligon prekriva prozor (3)

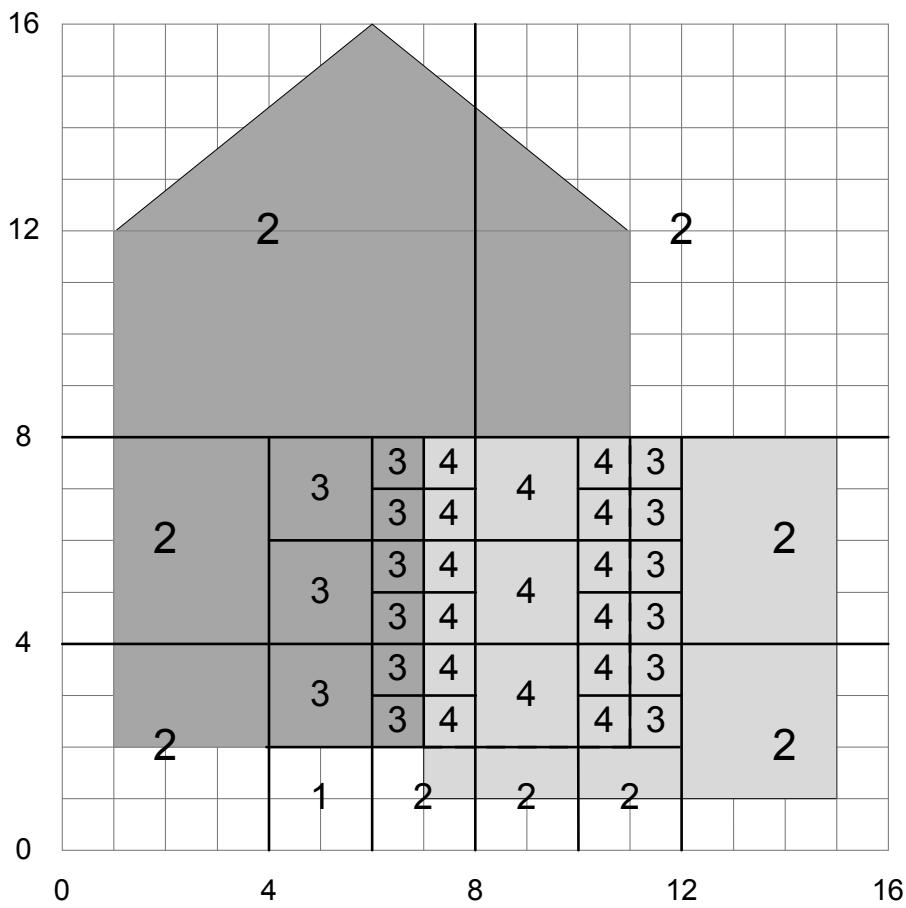
pripadne liste poligona. Podjelu rekuzivno dalje ponavljamo dok ne ostvarimo neki od postavljenih slučajeva. Na slici 7.14 prikazan je osnovni prozor i podjela na četiri jednakaka dijela. Ako je pojedini dio dalje potrebno dijeliti tada to i učinimo sve do unaprijed zadane dubine rekurzije. U prikazanom primjeru gornji lijevi prozor se dalje dijeli, dok ostale nismo dijelili, a nakon te podjele gornji desni smo podijelili i time završili podjelu. Za dio prozora gdje smo rekurzijom došli do kraja koristit ćemo neki drugi algoritam odluke u prikazu poligona.

Mogući slučajevi koji određuju hoće li dalje ostvarivati podjelu su:

- (1) poligon je izvan prozora pa ga uklonimo iz liste,
- (2) poligon siječe prozor ili je u prozoru,
- (3) poligon (jedan) prekriva prozor,
- (4) više poligona prekriva ili siječe prozor,
- (0) scena nije jednostavna - dijelimo dalje.

Znači, ako je poligon izvan prozora kojeg smo dobili podjelom, jasno je da ćemo ga ukloniti iz liste (1) 7.15. Ako imamo samo jedan poligon koji siječe prozor ili je u prozoru (2), tada je scena jednostavna i možemo ju prikazati za taj prozor, pa ćemo dani poligon i prikazati u prozoru. Treći slučaj je također jednostavan, a to je kada jedan poligon prekriva prozor, pa i njegova boja u konačnici daje prikaz. Četvrti slučaj je složeniji i potrebna je detaljnija analiza. U četvrtom slučaju, kada više poligona prekriva ili siječe prozor, potrebno je provjeriti postoji li jedan koji je najbliži promatraču i koji prekriva ostale. U tom slučaju prikazat ćemo taj najbliži, a ako to nemožemo jednoznačno utvrditi prelazimo na zadnji slučaj (0), kada scena nije jednostavna i moramo ju dalje dijeliti.

Primjer jednostavne scene u kojoj su prikazana dva poligona prikazana je na slici 7.16. Prvi je u



Slika 7.16: Warnockov postupak - primjer podjele za dva poligona gdje su označeni mogući slučajevi.

obliku kućice, a drugi je kvadratan. Nakon prve podjele gornji lijevi prozor ćemo morati dijeliti dalje i nakon te podjele, ponovo za gornji lijevi možemo zaključiti da je poligon van prozora pa pridjeljujemo oznaku (1). Za gornji desni i doljni lijevi jedan poligon je u prozoru pa pridjeljujemo oznaku (2) dok donji desni dalje dijelimo prema zadanim slučajevima. Postupak nastavljamo i za ostala tri kvadranta koja smo dobili u prvoj podjeli.

Idjea koja je ovdje prikazana i korištena za određivanje koji poligon je najbliži promatraču, može se na sličan način upotrijebiti i za stvaranje strukture četvero stabla te oktalnog stabla.

7.7 Četvero i oktalno stablo

Strukturu stabla čini niz povezanih čvorova od kojih je korijen osnovni čvor, a grane ga povezuju s ostalim čvorovima koji se dalje mogu granati ili mogu biti završni čvorovi. U strukturi četvero stabla (engl quadtree) primitive scene na početku dijelimo u četiri grane stvarajući strukturu stabla ovisno u kojem kvadrantu se primitive nalaze. Podjelu rekurzivno ponavljamo do unaprijed zadane dubine rekursije. Uvjet zaustavljanja podjele je i ako je broj primitiva u nekom čvoru manji od unaprijed zadano broja. Primitivie u općem slučaju mogu biti točke, linije, poligoni, cijeli objekti ili dijelovi scene. Ako se primitiva nađe na rubu između dva susjedna dijela prostora tada se primitiva zapisuje u oba podčvora ili se takve primitive čuvaju u čvoru kojeg dijelimo, te se dalje ne razvrstavaju, ovisno već o tome za koju varijantu izgradnje stabla se odlučimo. Izgrađeno stablo služi za brzu pretragu prostora. Zanima li nas, na primjer, za zadanu točku je li u nekom poligonu, iz koordinata točke brzo ćemo odrediti u kojem podprozoru se nalazi i s kojim poligonima upoće trebamo raditi provjeru. Oktalno stablo (engl. Octree) analogno je četverostablu, osim što svaki čvor sarži podjelu na osam grana koje dalje rekurzivno pratimo. Osim x i y koordinata promatramo i cijeli volumen scene odnosno z koordinatu. Slika 7.17 prikazuje primjer podjele prostora pri izgradnji oktalnog stabla. U prikazanom primjeru donji lijevi oktant se rekurzivno dijeli. Interesantna informacija može biti li

nešto izvan objekta, na rubu objekta ili u unutrašnjosti te će i pripadna informacija u oktalom stablu biti pohranjena.

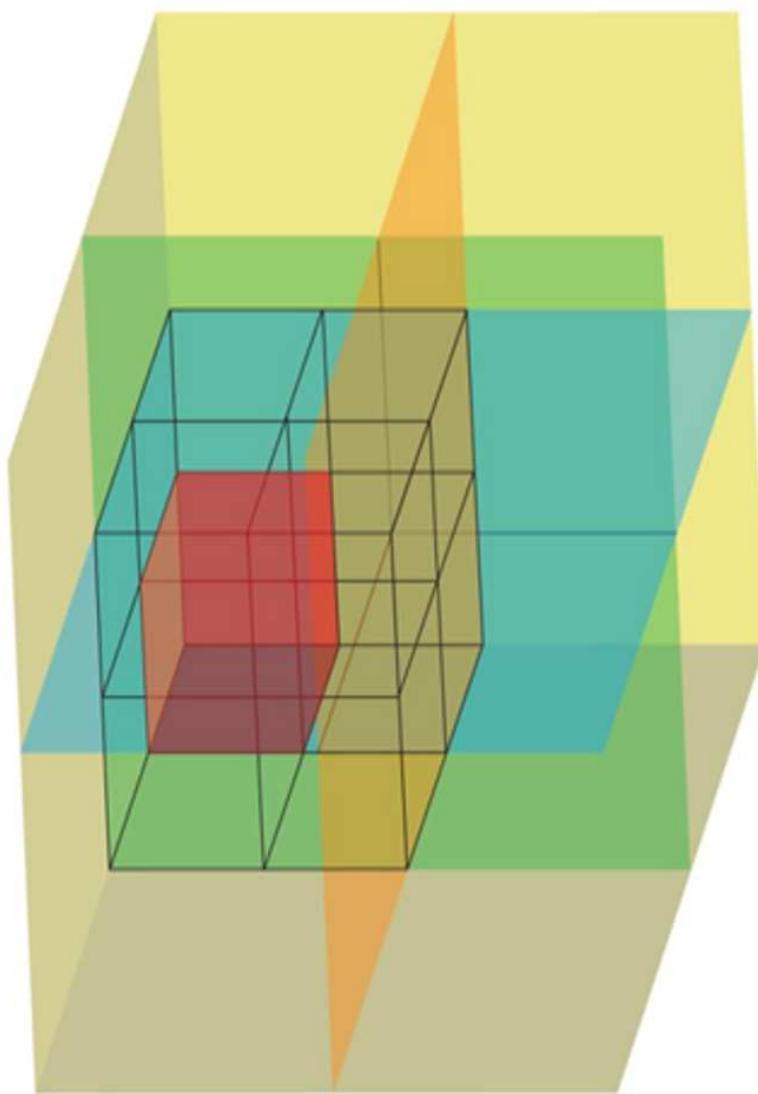
Pogledajmo jedan primjer gdje je pogled na scenu odozgo. Prostor je podijeljen u četverostablo i pojedini čvorovi su nazvani ćelijama (Slika 7.17). Ćelije mogu biti prazne, granične ili neprozirne, a pripadna polja su na slici svjetlosiva, siva i crna. Crveno je označena točka iz koje se promatra scena, a žuto ćelije koje su u smjeru pogleda i neprozirne su, dok su plavom označene ćelije koje su smjeru pogleda i zaklonjene su neprozirnim ćelijama. Ovdje možemo lako uočiti da kod složenih scena, uz dobru podijelu možemo vrlo veliki dio scene ukloniti tako da ga uopće ne šaljemo grafičkom protočnom sustavu i time bitno ubrzamo izvođenje aplikacije. U prvom redu to su sve ćelije koje su van piramide pogleda (engl frustum), a dalje to su i plave ćelije obzirom da su zaklonjene. Znači u konačnici, samo će ćelije koje su u pramidi pogleda i nisu zaklonjene, biti proslijedene protočnom sustavu. Cijela scena se na početku izvođenja aplikacije iz glavne memorije računala prebacuje u memoriju grafičke kartice. Danas uobičajeno ima dovoljno memorije na grafičkoj kartici za ovakav način rada. Ako mijenjamo scenu onda obično pričekamo prebacivanje objekata i tekstura neke druge scene. U OpenGLu, kada želimo objekte jednokratno prebaciti u memoriju grafičke kartice i onda ih višekratno koristiti upotrijebit ćemo VBO (Vertex Buffer Object), odnosno više takvih objekata. Određivanje geoemtijskih podataka koji su potrebni pri iscrtavanju (obično se obavlja na CPU) i slanje svaki puta nanovo u memoriju grafičke kartice ne bi bilo dobro riješenje. Izračunavanje koji dijelovi su zaklonjeni na osnovi sagrađenog stabla obično se radi na CPU, pa se samo određeni dijelovi koji su vidljivi, proslijeduju na prikazivanje ali iz memorije grafičke kartice. U protočnom sustavu, dalje, niz poligona možemo ukloniti kao stražnje za što smo vidjeli da je potrebno svega nekoliko naredbi u OpenGLu, a obično pola poligona jesu stražnji. Programi za sjenčanje geometrije osim što omogućuju stvaranje nove geometrije omogućuju i uklanjanje postojeće geometrije iz protočnog sustava, pa ovdje onda možemo primijeniti razne algoritme koji će zaključiti što se još može ukloniti prije faze rasterizacije, čime smanjujemo opterećenje i iscrtavanje zaklonjenih poligona (engl. *overdraw*).

Stvorene stukture podataka o sceni u obliku četvero ili oktalog stabla koriste se u raznim primjenama gdje je potrebna pretraga ili analiza prostora scene. To je, na primjer, u postupcima detekcije kolizije, gdje je potrebno utvrditi koji objekt s kojim je potencijalno u koliziji. Za na primjer n objekata bit će potrebno provjeriti svaki objekt sa svakim je li u koliziji, odnosno biće potreno $n(n - 1)/2$ provjere i to će biti potrebno izračunavati za svaki okvir animacije koji se prikazuje, odnosno nekoliko desetaka puta u sekundi. Podjela svih objekata u skupine po oktantima koje su potencijalno u koliziji drastično će smanjiti potreban broj ispitivanja. U postupku praćenja zrake, gdje je izraziti problem određivanja probodišta zrake i tijela bitno je što je moguće više smanjiti broj nepotrebnih ispitivanja, a organizacija prostora i podataka u tome znatno doprinosi.

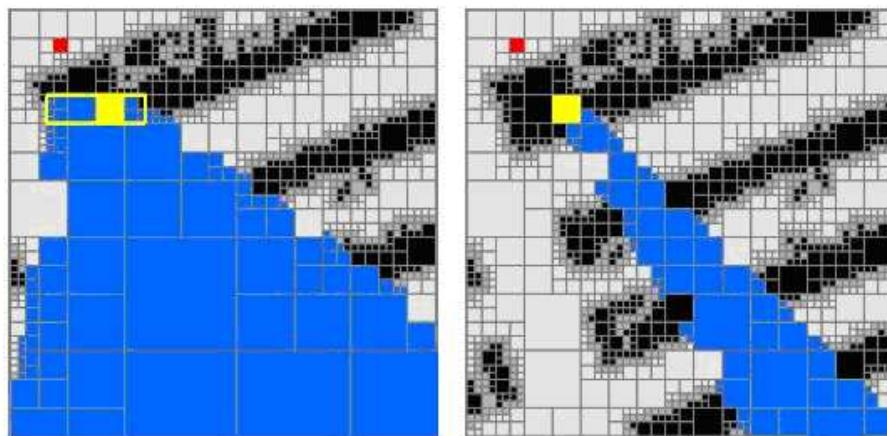
7.8 Algoritam Cohen Sutherlanda

Osnovna primjena ovog algoritma je odsijecanje linija, odnosno poligona obzirom na područje zaslona na kojem se ostvaruje prikaz. Prvi korak algoritma je podjela prostora na devet dijelova i pridjeljivanje četverobitnog koda. Točkama koje se nađu u pojedinom od devet dijelova pridjelujmo četverobitni kod. To znači da ćemo prvi bit s lijeva postaviti u jedinicu ako je nešto iznad y_{max} , a inače u nulu. Drugi bit postavljamo u jedinicu ako smo u dijelu prostora ispod y_{min} . Slijedeći bit postavljamo u jedinicu ako smo desno od x_{max} i zadnji bit je jedan ako smo lijevo od x_{min} (Slika 7.19). Na ovaj način imat ćemo za svaku točku u prostoru jednoznačno pridijeljen kod, a središnji dio koji čini naš zaslon ima pridijeljen kod 0000. Za promatranoj točku $T(x, y)$ dovoljno je ispitati predznak $y_{max} - y$ da bi postavili najznačajniji bit koda, što je sklopovski jednostana operacija. U ovom algoritmu posebice treba razmišlati o sklopovskoj implementaciji obzirom da je temeljni algoritam svaki sustav ga treba imati i sklopovski se implementira.

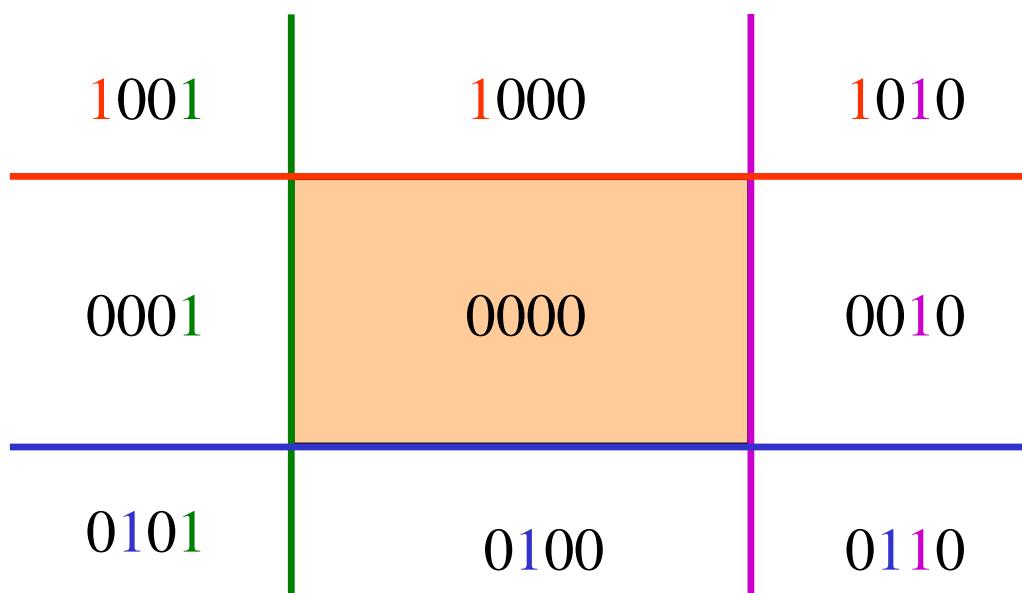
Nakon što točkama promatrane dužine V_1 , V_2 pridijelimo četverobitne kodove c_1 i c_2 krećemo s ispitivanjem. Prvi trivijalan slučaj je ako su obje točke unutar središnjeg prozora $c_1 = 0000$, $c_2 = 0000$ i odsijecanje nije potrebno. Dalje, provjerit ćemo logičku operaciju *AND* između postavljenih bitova koda. U slučaju kada je taj rezultat različit od 0000 dužina trivijalno nije vidljiva. Možemo primjetiti da ako su na primjer obje točke iznad y_{max} očito je i cijela dužina iznad y_{max} pa će četverobitni kod



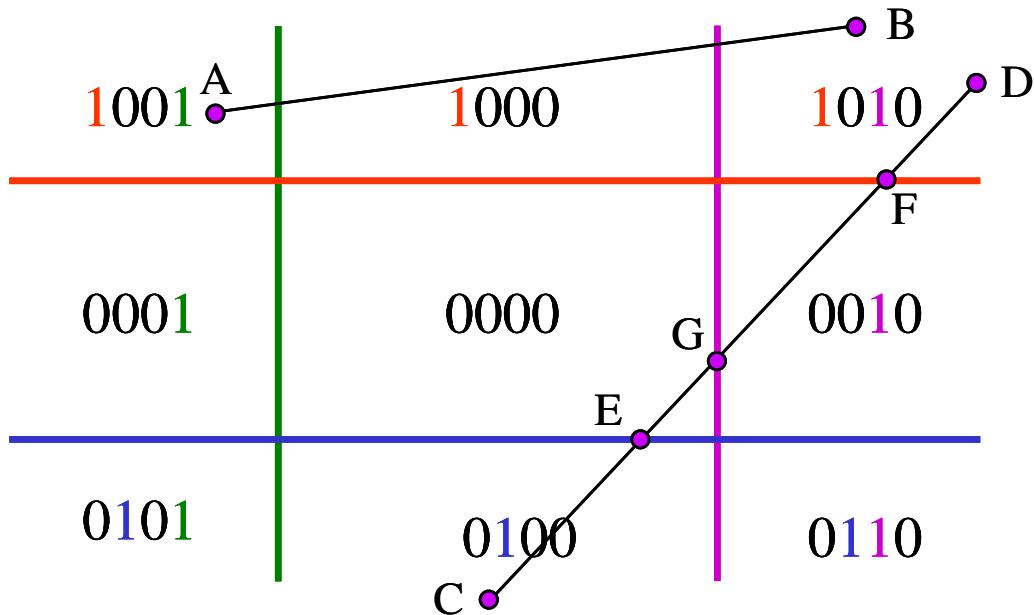
Slika 7.17: Oktalno stablo. Prostor dijelimo u oktatnte ovisno o sadržaju pojedinih oktanata.



Slika 7.18: Prostor podijeljen na ćelije koje mogu biti prazne granične ili neprozirne. Iz crvene točke se promatra scene, žute su neprozirne ćelije koje zaklanjaju plave ćelije (Schaufler).



Slika 7.19: Podjela prostora na devet dijelova kod algoritma Cohen Sutherlanda.

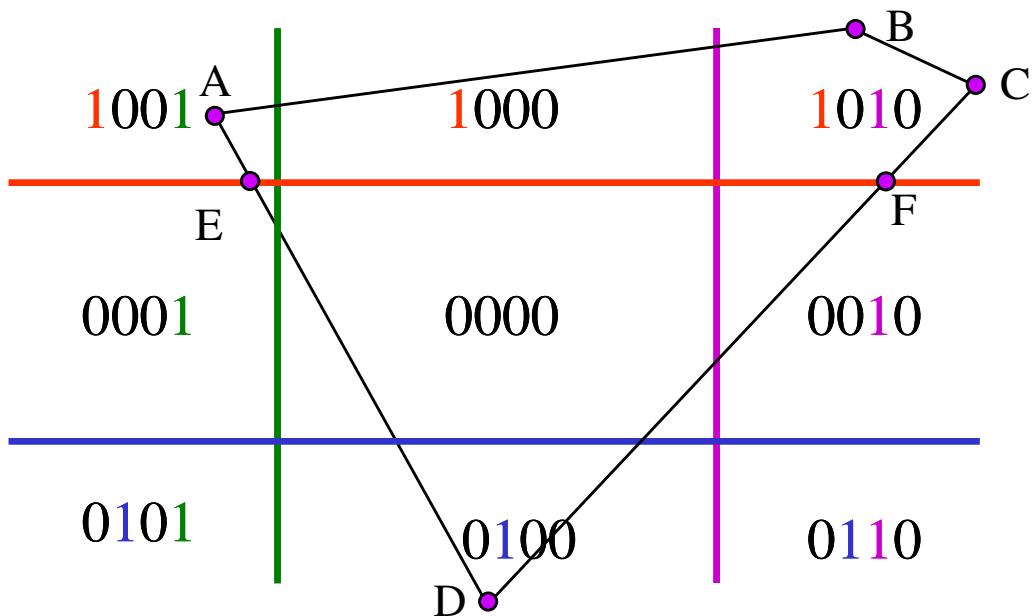


Slika 7.20: Primjer odsijecanja dužine kod algoritma Cohen Sutherlanda.

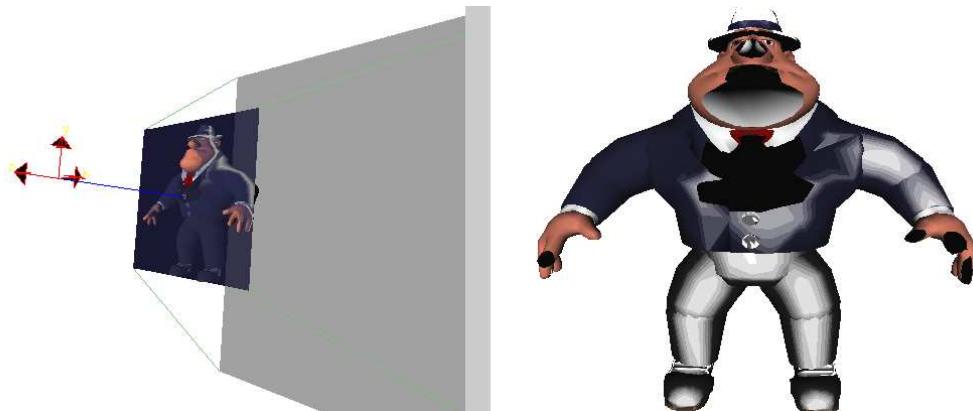
biti $1000 \text{ AND } 1000 = 1000$, kao što je slučaj za dužinu AB na slici 7.20. Ako ni to nije ispunjeno, potrebo je odsijecanje. Odsijecanje se provodi prema unaprijed zadanim redoslijedu ispitivanja. Na primjer, prvo provjeravamo bitove postavljene u 1 za točku V_1 pa za točku V_2 i to s lijeva na desno. Redoslijed ispitivanja može biti proizvoljni zadan. Promotrimo primjer na slici 7.20 za dužinu CD . Točka C ima pridijeljen kod $c_1 = 0100$, a točka D kod $c_2 = 1010$. Kako je pri provjeri prvi bit koji je postavljen i na koji ćemo naići drugi bit s lijeva u c_1 tako je prva provjera s pravcem $y = y_{min}$. Potrebno je odrediti sjecište pravca određenog točkama CD i pravca $y = y_{min}$. Označimo to sjecište s E . Točki E ćemo ponovno pridijeliti kod, a kako smo upravo načinili ispitivanje s y_{min} , i točka se nalazi upravo na liniji y_{min} jedinica koju smo provjeravali se briše. Segment CE ćemo odbaciti, dok novu liniju koju dalje provjeravamo čini linija ED . Nova linija za provjeru je ED , svi bitivi točke E su nula, pa prelazimo na točku D . Postavljen je prvi bit s lijeva pa se obavlja provjera, odnosno traži se sjecište s y_{max} , odnosno točka F i konačno zadnji bit koji određuje provjeru s x_{max} daje točku G . Gdje je konačni segment koji prikazujemo u našem središnje dijelu EG . Iako tijekom algoritma pojedine dijelove odbacujemo i dobivamo nove segmente izračun sjecišta uvijek obavljamo s linijom koja je odredene izvornim točkama, u našem primjeru CD . Parametri pravca određenog točkama CD se ionako ne mijenjaju, a ponovni izračun bi mogao uzrokovati odstupanja zbog numeničke pogreške. Možemo primjetiti da je moguće nepotreban izračun nekih točaka. U našem primjeru točka F ne čini konačno riješenje i izračun nije potreban, no to je nedostatak algoritma. Za drugačije odabira redoslijed točaka, na primjer DC i redoslijed dobivenih sjecišta bit će različit. Pa tako za segment DC dobit ćemo redom segmente: FC , GE , GC .

Odsijecanje poligona obzirom na promatrani prozor provodi se slično. Uspoređivanje pojedinih brodova poligona provodi se redom obzirom na pojedine pravce i čuva se lista bridova koji nakon takvog odsijecanja ostaju. Na slici 7.21 prikazano je odsijecanje poligona $ABCD$ obzirom na gornji pravac y_{max} , nakon čega ostaje poligon FDE koji se dalje sječe s ostalim linijama odsijecanja.

Opisani algoritam lako je proširiv na tri dimenzije, odnosno na volumen pogleda, koji može biti kvadar ili krnja piramida. Četverobitni kod potrebno je proširiti još s dva bita koji određuju je li nešto ispred prednje ravnine odsijecanja ili je iza stražnje ravnine odsijecanja. Postupak se provodi sklopovski i rezultat odsijecanja vidljiv je na slici 7.22. Lijevo je prikazan objekt, položaj kamere te prednja i stražnja ravnina odsijecanja. Prednja ravnina odsijecanja zahvaća dio objekta, odnosno siječe ga te će nakon provedenog postupka ostati rupa u objektu kroz koju se vidi stražnja strana objekta što je vidljivo na slici 7.22 desno.



Slika 7.21: Primjer odsijecanja poligona kod algoritma Cohen Sutherlanda.



Slika 7.22: Primjer odsijecanja obzirom na volumen pogleda kod algoritma Cohen Sutherlanda.

7.9 Algoritam Cyrus Beck

Algoritam Cyrus Becka omogućuje određivanje sjecišta linije, odnosno dužine s proizvoljnim konveksnim tijelom. Algoritam je vrlo sličan algoritmu koji se koristi u laboratorijskim vježbama za popunjavanje konveksnog poligona, osim što je proširen tako da radi u 3D prostoru. Pojmovi koji se koriste u vježbi popunjavanja konveksnog poligona "lijevo sjecište" i "desno sjecište" ovdje su poopćeni u "potencijalno ulaznu" točku i "potencijalno izlaznu" točku.

Prvo je potrebno odrediti izraz za određivanje sjecišta pravca i poligona. Neka je pravac određen točkama dužine P_0P_1 . Parametarska jednačba pravca kroz te dvije točke za parametar t je: $P(t) = t(P_1 - P_0) + P_0$. Neka je normala poligona \vec{n}_i usmjerena prema vanjštini objekta. Odaberimo sada jednu točku poligona i označimo ju P_{Ei} . Obično se za ovu točku odabire proizvoljan vrh poligona. Sada promatramo vektor između odabranog vrha P_{Ei} i bilo koje točke na pravcu. Za kut između promatranog vektora i vektora normale poligona možemo zaključiti da je taj kut manji od 90° ako je točka na pravcu van promatranog poligona, odnosno skalarni produkt tih vektora je pozitivan. Na slici 7.23 prikazan je pogled okomito na ravninu u kojoj je poligon, pa nam je poligon zapravo degenerirao u jednu dužinu na koju je označena normala \vec{n}_i . Očito je da je kut između vektora normale tog poligona \vec{n}_i i vektora između točaka P_{Ei} i na primjer P_0 biti manji od 90° . Promatramo li točku koja je u ravnini poligona, produkt promatralih vektora bit će nula, dok za točke koje su ispod ravnine poligona, kao što je na slici P_1 , produkt će biti negativan. Nas zanima točka probodišta, odnosno slučaj kada je promatrani produkt nula, odnosno:

$$\vec{n} \cdot (P(t) - P_{Ei}) = 0$$

Uvrstimo li u navedenu formulu parametarsku jednačbu pravca $P(t) = t(P_1 - P_0) + P_0$, za parametar t ćemo dobiti:

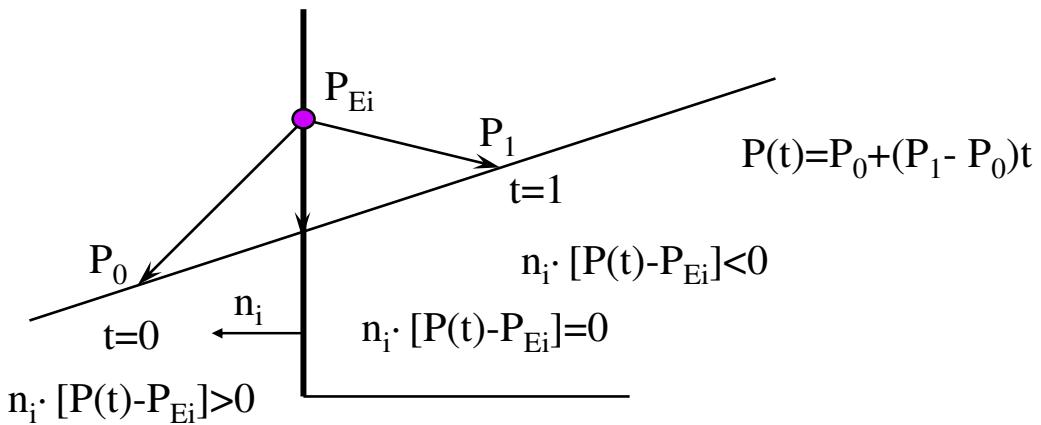
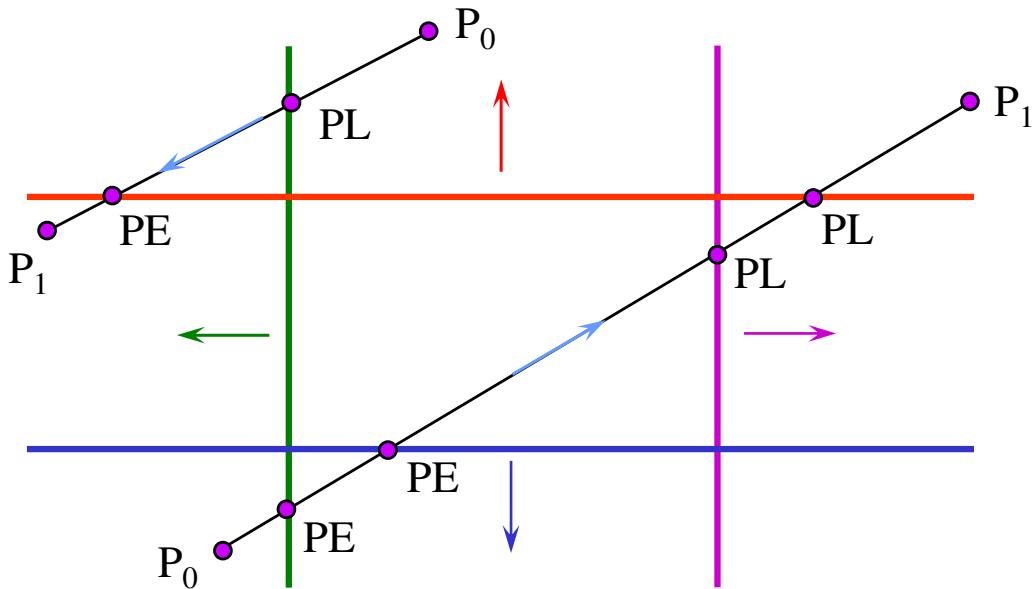
$$t = \frac{\vec{n}_i \cdot (P_0 - P_{Ei})}{-\vec{n}_i \cdot D},$$

gdje je $D = P_1 - P_0$. I to je parametar t koji određuje probodište pravca i ravnine u kojoj leži promatrani poligon. Promotrimo detaljnije izraz za parametar koji smo upravo dobili. U brojniku i u nazivniku imamo skalarni produkt dva vektora, od kojih je jedan vektor normale \vec{n}_i . Drugim riječima imamo omjer projekcija vektora $P_0 - P_{Ei}$ i vektora $P_1 - P_0$ na vektor normale. Naveden omjer, odnosno parametar t može biti pozitivan nula i negativan, te nam određuje točku probodišta.

U izvedenoj formuli za t moramo još pripaziti da nema nepoželjnih slučajeva, odnosno dijeljenja s nulom. Nazivnik može biti nula u nekoliko slučajeva. To je ako je vektor normale nula, što nije za očekivati, no moramo provjeriti. Mogući su slučajevi da zbog različitih transformacija bridovi poligona čiju normalu izračunavamo degeneriraju u pravac čime vektor normale postaje nula. Isto tako ako poligoni postanu sićušni, zbog numeričke preciznosti moguće su degeneracije vektora normale u nul vektor. U zapisu objekta možemo imati neregularnih poligona koji degeneriraju u liniju, što obično u prikazu u nećemo ni primijetiti, a normala će, naravno, biti nula. Slijedeći vektor koji nam može stvoriti probleme je $P_1 - P_0$. On će biti nula ako te dvije točke degeneriraju u jednu. Lijepo bi bilo da to ne učine, ali znamo da stari u životu nisu idealne pa zbog prethodno opisanih razloga upravo to se može dogoditi. Treća je mogućnost da oba promatrana vektora nisu nul vektori, ali da njihov skalarni produkt je. To će biti u slučaju kada je promatrani pravac paralelan s ravninom u kojoj leži poligon, a time je sjecište u beskonačnosti i ne možemo ga odrediti. Dijeljenje s nulom u tom slučaju moramo spriječiti tako da sjecište ne određujemo. Još je zanimljiv slučaj ako je brojnik nula, odnosno ako je vektor $P_0 - P_{Ei}$ nula. To znači da su se odabrana točka poligona i prva točka promatrane dužine poklopili. Parametar t je tada nula, što je sasvim u redu jer je prva točka ujedino i sjecište, pa je u tom slučaju sve regularno.

Određivanje sjecišta pravca zadanog točkama i poligona može se svesti i na 2D slučaj gdje promatramo sjecište 2D pravca i poligona. Sve ostaje potpuno isto kao i u opisanom slučaju u 3D, osim što normala koju smo promatrali \vec{n}_i je sada normala brida poligona. Sliku 7.23 sada možemo promatrati u 2D i po istom postupku odrediti sjecište s pravcem.

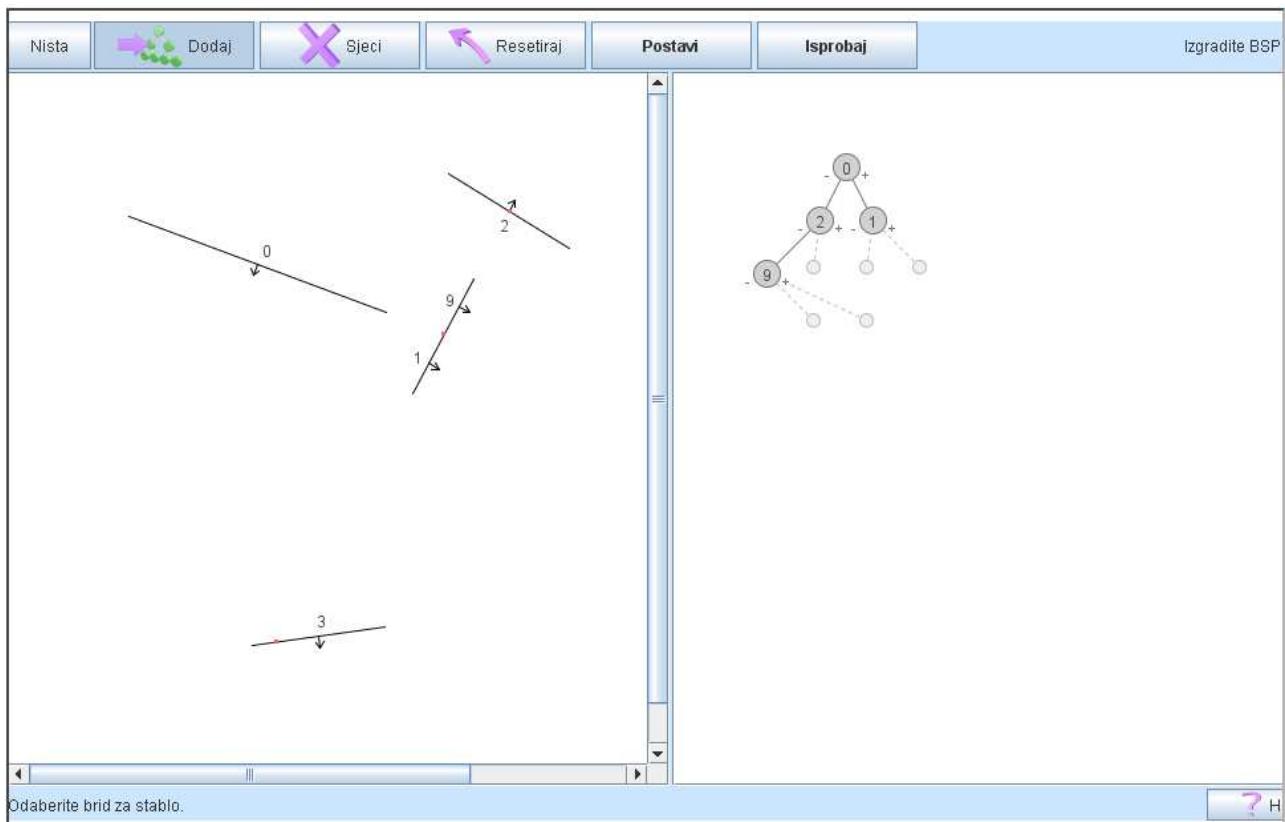
Slijedeći korak je određivanje potencijalno ulaznih i potencijalno izlaznih točaka. Označimo potencijalno ulazno sjecište s PE , a potencijalno izlazno s PL . Postavimo inicijalne vrijednosti $PE = 0$ i

Slika 7.23: Određivanje probodišta pravca na kojem je dužina P_0P_1 i poligona čija je normala \vec{n}_i .Slika 7.24: Razvrstavanje sjecišta na PE i PL .

$PL = 1$. Je li neko sjecište potencijalno ulazno određeno je kutem između vektora normale brida (promatramo li 2D slučaj) i vektora između točaka koje određuju pravac $P_1 - P_0$. Zapazimo da smo ovaj produkt ionako već računali pri određivanju parametra t , pa se izračun pokazuje višestruko korisnim. Ako je ovaj produkt negativan, odnosno kut između vektora je veći od 90° , sjecište je potencijalno ulazno PE , dok je u suprotnom potencijalno izlazno PL . Na ovaj način sva sjecišta imamo razvrstana kao PE ili kao PL . Izbor najvećeg PE i najmanjeg PL dat će traženo odsijecanje. Kako smo na početku inicijalizirali $PE = 0$ i $PL = 1$, imat ćemo odsiječenu dužinu obzirom na zadani konveksni poligon, čija početna točka ima $t = 0$, a krajnja $t = 1$. Ako želimo načiniti odsijecanje pravca, a ne dužine, vrijednosti ćemo inicijalizirati na $PE = VrloMaliBroj$ i $PL = VrloVelikiBroj$.

7.10 Binarna podjela prostora BSP

Izgradnja stabla kroz koje organiziramo scenu u pojedine djelove značajno može doprinjeti ubrzaju postupaka kojima pretražujemo takve scenu. Jedna od najzančajnijih takvih organizacija je BSP (engl. Binary Space Partitioning) odnosno Binarna podjela prostora. Osnovna ideja izgradnje BSP stabla je rekursivna binarna podjela prostora. Pogledajmo prvo problem u 2D prostoru. Neka početna scena sarži proizvoljan broj bridova ili geometrijskih likova koji se sastoje od bridova. Početni korak je da odaberemo bilo koji brid i pravcem na kojem leži taj brid binarno podijelimo prostor na dio koji je iznad promatranih pravaca i dio koji je ispod. Slučaj kada je točka točno na promatranoj pravcu

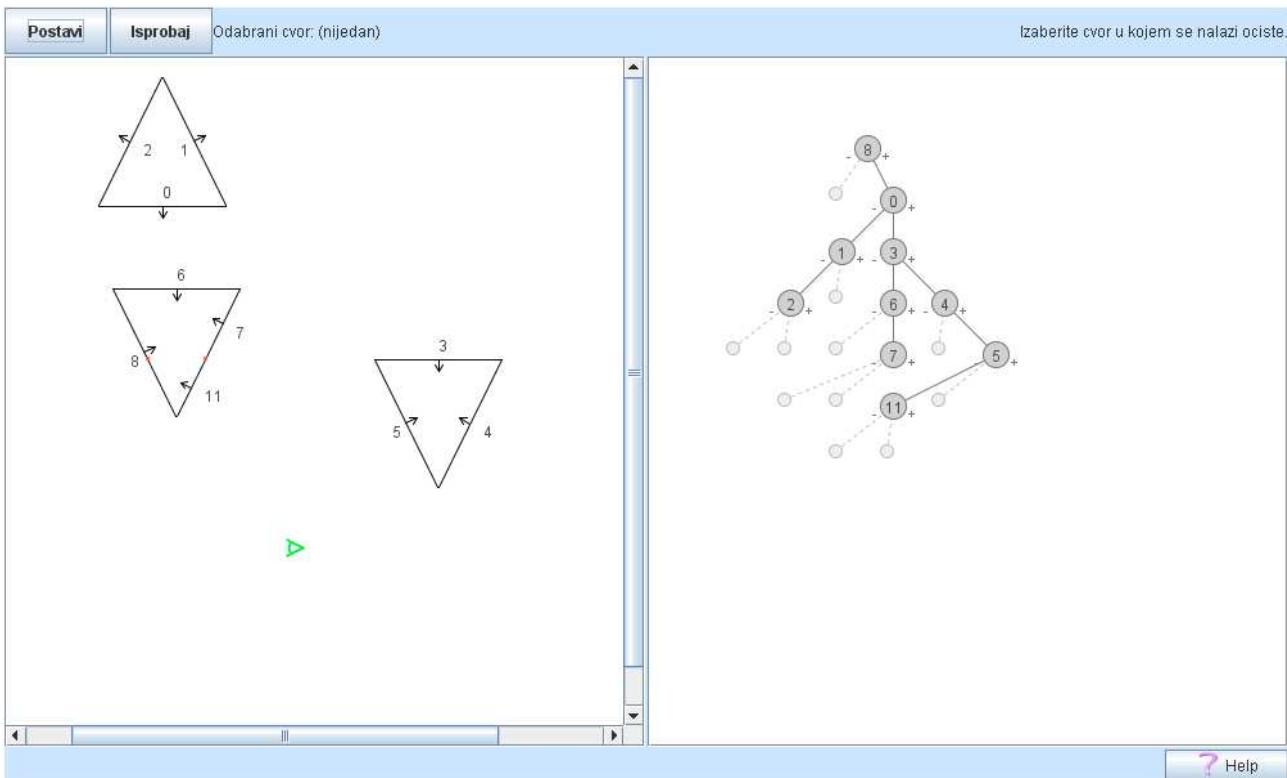


Slika 7.25: Izgradnja BSP stabla. Postupak nije dovršen, brid 3 još treba podijeliti i smjestiti u stablo
<http://www.zemris.fer.hr/predmeti/irg/BSP/>.

može se pridružiti jednom od ova dva osnovna slučaja. Istovremeno s ovom podjelom gradimo stablo i definiramo početni čvor (korijen) stabla. U taj čvor, i bilo koji čvor koji ćemo kasnije napraviti, spremimo označku brida kojeg promatramo (ID brida), jednadžbu pravca na kojem je promatrani brid kojim smo ostvarili podjelu, te načinimo dva pokazivača. Jedan pokazivač pokazuje na dio stabla koji je vezan uz prostor 'iznad', a drugi uz dio prostora 'ispod' promatranog pravca. U shemi koju koristimo za prikaz stabla prvi pokazivač označimo s +, a drugi s -. Ako smo ovom početnom podjelom presjekli bilo koji drugi brid tada od takvog brida u stvari radimo dva nova od kojih je jedan dio iznad a drugi ispod početnog pravca. Ovom početnom podjelom razvrstamo i sve ostale bridove scene uključujući i ove nove dobivene dijeljenjem presječenih bridova u dvije skupine. To su skupine bridova iznad početnog pravca i skupina bridova ispod početnog pravca.

Na slici 7.25 možemo vidjeti da je odabran prvo brid 0, te je postavljan desno na istoj slici kao korijen stabla. Brid koji se nalazi neposredno desno-dolje od promatranog brida presječen je na dva dijela od kojih je jedan označen s 1, a drugi s 9. Kako će se ovi bridovi naći na različitim mjestima u stablu i njihova će jednadžba pravca biti zapravo dva puta zapisana. Već nakon ove prve podjele znamo da će se bridovi 2 i 9 nalaziti s lijeve strane stabla zato što su 'ispod' brida 0, a bridovi 1 i 3 bit će s desne strane, gdje je i oznaka podstabla +. Brojevi pridruženi pojedinim oznakama bridova ovdje su proizvoljni (nisu bitni). Ovaj osnovni princip dijeljenja se dalje nastavlja s tim da sada više ne promatramo cijelu ravninu, već promatramo odvojeno dvije poluravnine koje smo dobili prvom podjelom. Za lijevo podstablo i daljnju podjelu odabran je brid 2, a brid 9 se nalazi s njegove 'negativne strane', pa su tako i smješteni u stablu. U desnom podstabalu odabran je brid 1, za koji vidimo da će pravac na kojem leži presjeći brid 3 u crvenoj točkici na bridu. U ovom primjeru postupak nije dovršen. Potrebno bi bilo još ostvariti podjelu brida 3 na dva dijela i jedan od njih postaviti s pozitivne, a drugi s negativne strane brida 1. Time bi izgrađeno stablo bilo potpuno.

Možemo primijetiti da će odabir redoslijeda bridova utjecati na izgradnju stabla. Biramo li bridove koji se nalaze posred scene, stablo će biti balansirano, dok ako bridove biramo s rubnih dijelova, stablo će postati lista. Naravno, poželjno je da stablo bude balansirano.



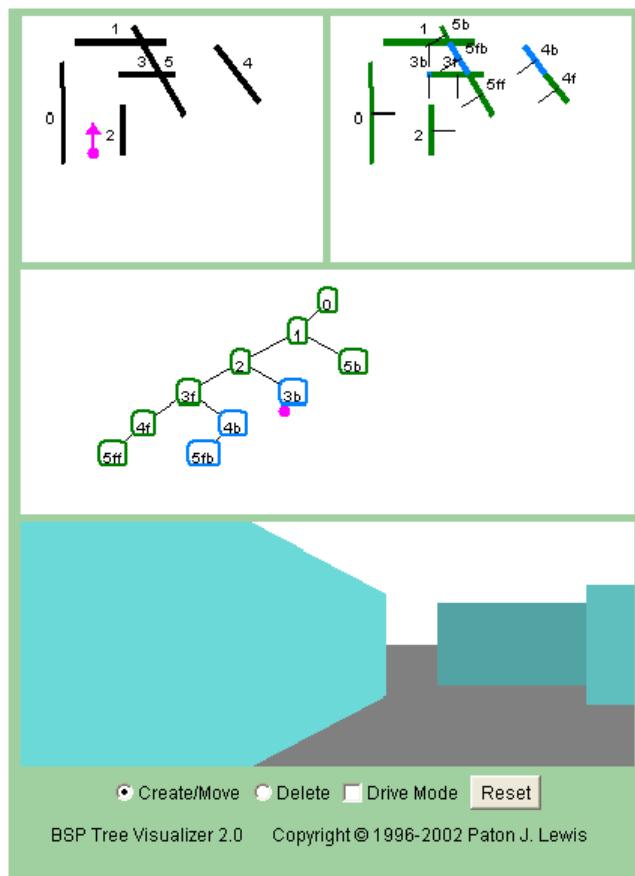
Slika 7.26: Traženje pozicije točke (očista) u BSP stablu i pripadne pozicije u sceni <http://www.zemris.fer.hr/predmeti/irg/BSP/>.

Kada jednom scenu podijelimo i načinimo pripadno stablo u prostoru smo dobili niz konveksnih podprostora koji pripadaju završnim čvorovima stabla. Za odbir pravaca kojima obavljamo podjelu nije nužan uvjet da na njima jesu bridovi, već možemo odabrati i proizvoljne pravce u prostoru. Bridove biramo jer obično uzrokuju najmanje dodatnih presijecanja. Specijalan slučaj može nastupiti ukoliko je brid kojeg trebamo razvrstati već na pravcu koji nam dijeli prostor, no tada je njegova jednadžba već ionako u zapisana u promatranoj čvoru, pa je potrebno zapisati i oznaku toga brida.

Čemu sad služi stablo koje smo napravili? Jedna o osnovnih primjena je za određivanje u kojem od podprostora se nalazi točka $V(x, y, z)$. Na prvi pogled, ovo djeluje malo zbumnjuće, pa naravno da vidimo gdje se nalazi točka. No da to objasnimo računalu i nije tako jednostavno, posebno za veće scene. Ispitivanje gdje se nalazi točka korištenjem stabla je jednostavno. Umnožak točke i pohranjene jednadžbe pravca u pojedinom čvoru dat će odgovor je li točka 'ispod' ili 'iznad' pojedinih pravaca pa ovisno o tom rezultatu krećemo na lijevu ili desnu stranu podstabla, sve dok ne dođemo do podprostora gdje se točka doista nalazi.

Na slici 7.26 točka čiji položaj ispitujemo je očište i označena je malim zelenim na slici. Potrebno je redom pomnožiti poziciju točke s jednadžbama bridova zapisanim u stablu. Pa prvo pomožimo koordinate promatrane točke s jednadžbom pravca zapisanom u korijenu stabla, odnosno s jednadžbom u čvoru '8' i zaključimo da je desno, pa s '0', i opet je desno, pa s '3', i opet je desno, pa s '4', opet desno, pa s '5', e sada je lijevo i s '11' i tu smo u konačnom čvoru. Na slici bi mogli i iscrtati konveksni poligon koji predstavlja cijelo područje koje pripada tom čvoru i vidjeli bi da je doista promatrana točka u tom poligону. U općem slučaju, ako imamo stotinjak poligona kakvi su na slici 7.26 rezultat će biti neki od listova stabla, no nas obično u konačnici zanima je li to neki od poligona ili se radi o prostoru oko poligona, što nas obično zanima kod detekcije kolizije. U tom slučaju u završnim čvorovima stabla čuvamo informaciju pripada li taj čvor praznom prostoru ili nekom poligonu. Općenito gledano, broj ispitivanja ovisit će o tome koliko je dobro balansirano stablo, i naravno kolika je scena.

Prelazak u 3D prostor samo proširuje razmatranje na poligone umjesto bridova i pripadne ravnine umjesto pravaca, koje opet prostor dijeli na dva dijela. Pripadno stablo će biti binarno stablo, a dijelovi prostora će biti konveksni poliedri. Primjer je prikazan na slici 7.27 gdje je scena sačinjena od niza 'zidova' prikazanih na slici gore lijevo. Na istoj slici gore desno su prikazani podijeljeni bridovi. Treba



Slika 7.27: Primjer 3D scene sačinjene od niza 'zidova' koji predstavljaju prostor scene. Izvor: <http://symbolcraft.com/graphics/bsp/index.php>

primijetiti da redoslijed odabira bridova pri izgradnji stabla određuje koji će bridovi biti dijeljeni. Pripadno stablo je prikazano ispod, a 3D prikaz prostora koji u konačnici izgleda kao labirint, je prikazan dolje. POMicanjem očišta bit će moguća interaktivna šetnja kroz scenu. Osnovni koncept izgradnje i primjene BSP stabla proširiv je i dalje na više dimenzije.

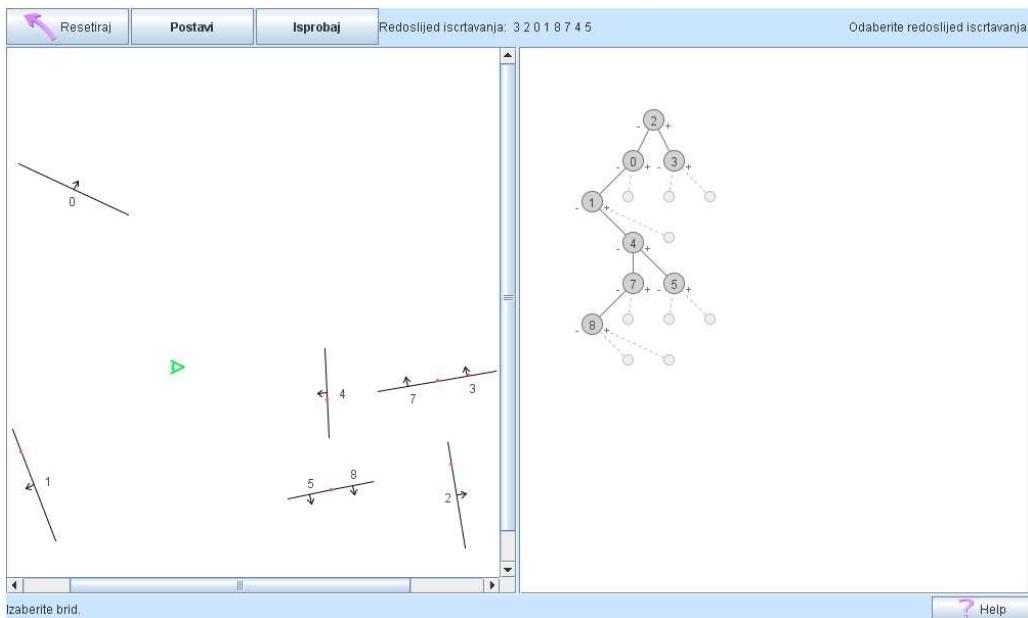
Slijedeća primjena sagradenog BSP stabla je za sortiranje poligona s obzirom na udaljenost od očišta od najdaljeg prema najbližem BTF (Back To Front). Pokušate li ovo napraviti korištenjem nekog od uobičajenih načina sortiranja, vrlo brzo ćete spoznati da je problem vrlo nezgodan. Promatrano u 2D prostoru, jasno je da svaki brid ima po dvije točke od kojih svaka ima po dvije koordinate što treba uzeti u razmatranje, a i da se promatrani bridovi mogu sjeći. Čak i uspješno rješenje sortiranja bridova na klasičan način u pravilu će dati prilično spora rješenja. Zato je za primjenu sortiranja bridova (poligona) upotreba BSP stabla vrlo značajna. Postupak sortiranja se temelji na provjeri određivanja položaja očišta u stablu, slično kao kod upravo opisanog postupka određivanja točke u stablu (Slika 7.26), i postupanja prema sljedećem algoritmu:

Ako je očište s

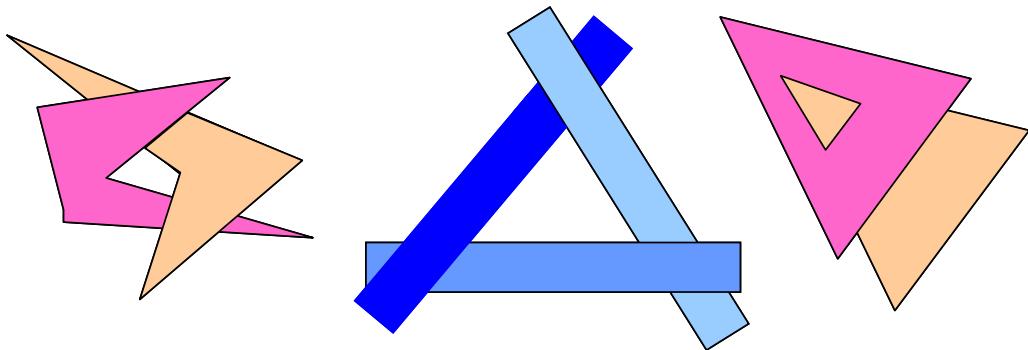
1. pozitivne strane obidi negativnu starnu, pa čvor, pa pozitivnu stranu,
 2. negativne strane obidi pozitivnu starnu, pa čvor, pa negativnu stranu,
- gdje obilaženje znači i iscratvanje pripadnih bridova.

Detaljnije algoritam izgleda ovako:

```
BSP_display(BSP_tree tree) {
    if (!EMPTY(tree)) {
        if (observer is located on front of root) {
            BSP_display(tree->backChild);
            displayPolygon(tree->root);
            BSP_display(tree->frontChild); }
        else {
            BSP_display(tree->frontChild); }
```



Slika 7.28: Sortiranje poligona obzirom na udaljenost od očišta.



Slika 7.29: Tipični problematični slučajevi kod postupaka uklanjanja skrivenih linija i površina.

```
displayPolygon(tree->root);
BSP_display(tree->backChild); } } }
```

Primjer sortiranja poligona prikazan je na slici 7.28. Krećemo s pozicijom očišta i korjenom stabla, te zaključujemo da se očište nalazi s negativne strane brida 2, te moramo prvo obići suprotnu stranu, znači čvor 3 pa čvor 2, pa tek onda negativnu stranu. Time će iscrtani bridovi biti 3 pa 2. Dalje posjećujemo čvor 0, a očište je s negativne strane. Obilazimo pozitivnu stranu koja je prazna, čvor 0 i pripadni brid, te dalje negativnu stranu gdje je čvor 1. Slično kao maloprije očište je s negativne strane pa je slijedeći iscrtani brid 1, a posjećeni čvor 4. Za čvor 4 očište je s pozitivne strane pa idemo do čvora 7 gdje je očište isto s pozitivne strane što nas vodi na iscrtavanje čvora 8, povratak na 7 i njegovo iscrtavanje, povratak na 4 i njegovo iscrtavanje, te konačno do čvora 5 čime je postupak završen. Konačan redoslijed iscrtanih bridova je 32018745.

Zanimljivo je da će navedeni algoritam uspješno prikazati i slučajeve koji su problematični za većinu ostalih algoritama za uklanjanje skrivenih linija i površina prikazanih na slici 7.29. Kada bi htjeli ove poligone iscrtati jedan po jedan od najdaljeg prema najbližem, ne bi mogli odrediti redoslijed kojim je to potrebno učiniti, a da scena bude ispravno prikazana. BSP algoritam, će upravo zbog dijeljenja bridova, odnosno poligona koje je sastavni dio algoritma ispravno prikazati bilo koju od prikazanih scena.

Poglavlje 8

Osvjetljavanje

8.1 Osvjetljavanje

8.1.1 Uvod

U prethodnim poglavljima naučili smo dovoljno da znamo kako iscrtati žičani model objekta na zaslonu. Sada je došlo vrijeme da se upoznamo s načinima kako u scenu uvesti svjetlosne izvore i analizirati njihov utjecaj na objekte. U prirodi, interakcija svjetlosti s površinom izuzetno je složena i stvaranje modela koji bi u potpunosti obuhvatio sve aspekte te interakcije ne bi bilo izvedivo. Stoga su razvijeni različiti modeli koji obuhvaćaju pojedine aspekte u većoj ili manjoj mjeri.

8.1.2 Fizikalni model svjetlosti

Postoje različite interpretacije, razvijane tijekom stoljeća, koje opisuju propagaciju svjetlosti kroz medij i interakciju s površinom. Tri su osnovne interpretacije koje se temelje na: geometrijskoj optici, fizikalnoj optici i kvantnoj optici.

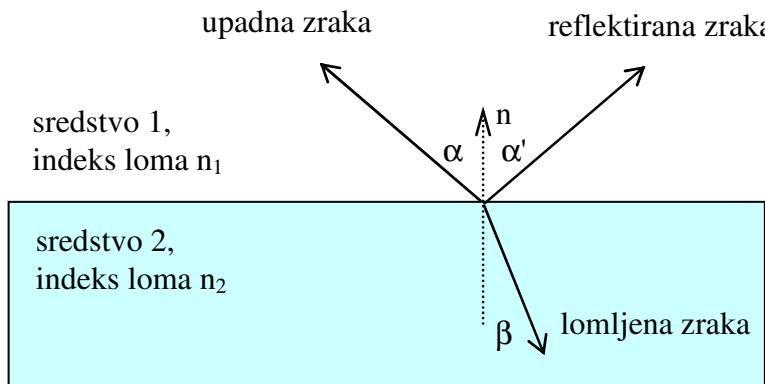
U okviru **geometrijske optike** svjetlost se predstavlja skupom svjetlosnih zraka. Geometrijska optika se temelji na četiri osnovni zakona, a to su zakoni o pravocrtnom širenju svjetlosti, neovisnost svjetlosnih snopova, zakon obijanja (refleksije) i zakon loma (refrakcije) svjetlosti (Slika 8.1). Zakon refleksije kaže da je kut između upadne zrake i normale n jednak kutu između reflektirane zrake i normale ($\alpha = \alpha'$). A zakon refrakcije (Snellov zakon) se odnosi na kut između lomljene zrake i normale (β) obzirom na upadni kut zrake (α) i vrijedi:

$$\frac{\sin(\alpha)}{\sin(\beta)} = \frac{v_1}{v_2} = \frac{n_2}{n_1}, \quad (8.1)$$

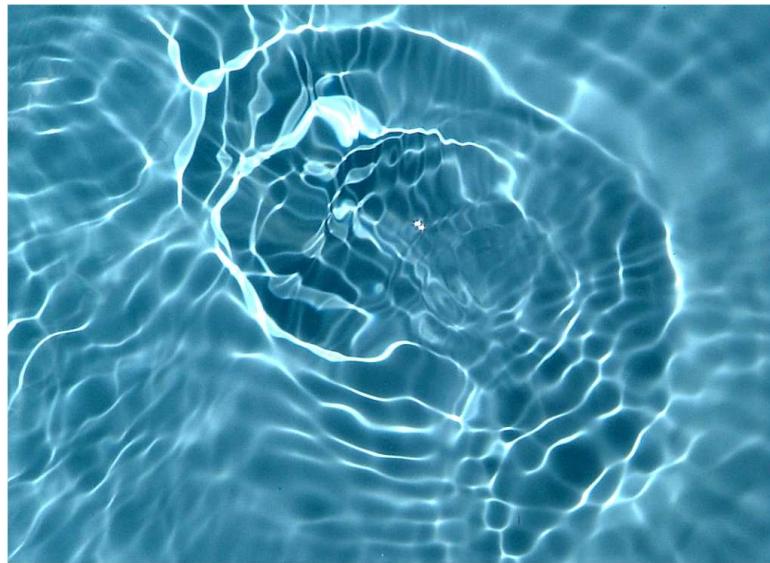
gdje su n_1 i n_2 indeksi loma pojedinih sredstava a v_1 i v_2 brzine širenja vala u pojedinim sredstvima. Prelaskom iz jednog sredstva u drugo mijenja se brzina širenja vala, a kako je indeks loma definiran kao omjer brzine svjetlosti naprema brzini širenja vala u sredstvu ($n = c/v$), vrijedi dani zakon.

Ove osnove se koriste u proučavanju ponašanja leća, zrcala i općenito optičkih sustava. Navedeni zakoni nam predstavljaju osnovu u svakom modelu kojim opisujemo ponašanje svjetlosti.

Svetlost u okviru **fizikalne optike** je shvaćena kao elektromagnetsko zračenje, odnosno širenje svjetlosti shvaća se kao širenje progresivnog, odnosno transverzalnog vala. Ovo znači da se kod svjetlosti električno i magnetsko polje vala mijenjaju periodički u svim smjerovima okomitim na smjer gibanja vala koji se širi. Pojave kao što su difrakcija (ogib svjetlosti) i interferencija ne mogu se objasniti geometrijskom optikom, već se objašnjavaju fizikalnom optikom. Osnovni princip rada LCD monitora također se zasniva na polarizirajućim svojstvima materijala i svjetlosti kao elektromagnetskom valu. Sunčeva svjetlost je val polariziran u svim smjerovima, odnosno titranje vala je prisutno u svim smjerovima, dok polarizirajući slojevi u LCD-u propuštaju samo horizontalno ili vertikalno polariziranu komponentu. Propuštanjem ili blokiranjem polarizirane svjetlosti postiže se vidljivost pojedinog slikovnog elementa. Šareni uzorci na mjehuriću sapunice također su primjer iz stvarnog života gdje valna priroda i interferencija svjetlosti dolazi do izražaja. Lijep vizualni učinak koji na određenom



Slika 8.1: Refleksija i refrakcija svjetlosti na površini.



Slika 8.2: Primjer učinka kaustike na površini vode.

mjestu stvara fronta svjetlosnog vala zove se kaustika. Učinak kaustike vidljiv je, na primjer, na dnu bazena ili mora za sunčanog vremena (Slika 8.2).

Treći aspekt ponašanja svjetlosti temelji se na čestičnoj prirodi svjetlosti, odnosno **kvantnoj optici**. Ovaj aspekt ponašanja svjetlosti vidljiv je kada čestična priroda svjetlosti temeljena na česticama zvanim *fotoni* dolazi do izražaja. Pojedina čestica nosi kvant energije koji je jednak hf , gdje je h Planckova konstanta, a f frekvencija svjetlosti. Ovaj koncept posebno nam je zanimljiv kod razumijevanja subtraktivnog sustava boja, o kojem će biti riječ kasnije. U ovom sustavu boja iz skupa frekvencija prisutnih u dolaznoj svjetlosti na neku obojenu površinu, neke će frekvencije biti absorbirane ("oduzete"), dok će ostale biti reflektirane. Boja površine, odnosno atomi koji ju čine sposobni su absorbirati samo određene frekvencije te je time određeno i što će biti reflektirano, odnosno koju ćemo boju u končnici vidjeti na danoj plohi.

Ovo su samo neki primjeri zanimljivih učinaka koje ćemo primijetiti promatrujući svjetlost, no i na vrlo jednostavnim površinama svjetlost koja dolazi, ovisno o valnim duljinama, raspršit će se različito (anizotropno) u pojedinim smjerovima, prodrijet će do neke dubine u površinu i tek će se onda reflektirati što je tipično za ljudsku kožu. Tako da postavljeni fizikalni zakoni daju tek osnovu ponašanja svjetlosti a u izgradnji modela kojim želimo vjerno reproducirati svu pojavnost njena ponašanja u svijetu koji nas okružuje susrećemo se s vrlo izazovnim zadatkom. Problem u izgradnji modela ne stvara samo interakcija svjetlosti s jednom površinom, već svaka površina u prostoriji svjetlost koju dolazi do nje i apsorbira i reflektira do bilo koje druge površine u prostoriji bilo direktno bilo indirektno, te se tu susrećemo s rekurzivnim problemom. Međusobna interakcija svjetlosti nad objektima prisutna je ukoliko u sceni imamo više od jednog objekta ili ako je objekt konkavan. Naime, dio svjetlosti

koji osvjetljava objekt A dolazi do objekta B (bilo refleksijom, bilo raspršenjem i sl.) i osvjetljava ga. Objekt B opet dio te svjetlosti odbija prema objektu A; objekt A opet dio šalje objektu B, i – vjerojatno ste shvatili. Zbog ove kompleksnosti uvode se modeli koji u određenoj mjeri uvažavaju pojedine fizikalne zakone i interakcije objekata, te daju koliko-toliko zadovoljavajuće rezultate.

Modeli kojima se aproksimira ponašanje svjetlosti u računalnog grafici temelje se na iznesenim fizikalnim osnovama i u većoj ili manjoj mjeri obuhvaćaju navedena fizikalna ponašanja. Vremenom su razvijani sve složeniji modeli kojima se aproksimira ponašanje svjetlosti, a oni koji nisu u početku bili ostvarivi u stvarnom vremenu razvojem računalne opreme to postaju.

No, odnekud moramo početi, pa to je najjednostavniji model i prva aproksimacija složenog ponašanja koje smo opisali. Općenito, **modelom osvjetljenja** određujemo kako izračunati intenzitet svjetlosti u promatranoj točki, a **postupkom sjenčanja** određujemo kako osjenčati površine uz odbarani model osvjetljenja.

Promatrano s aspekta fizikalne optike svjetlost je samo jedan dio spektra zračenja, te i za nju vrijedi općenito:

$$\text{Upadnasvjetlost} = \sum \begin{array}{l} \text{reflektirana} \\ \text{rasprsena} \\ \text{apsorbirana} \\ \text{transmitirana} \end{array}$$

Svaka od ovih komponenti ovisna je o samim fizikalnim svojstvima materijala, o gruboći površine, o valnoj duljini same svjetlosti i još mnoštvu faktora. Među najjednostavnije modele spada i Phongov model osvjetljavanja koji ćemo obraditi u nastavku. Ovaj model predstavlja prvu aproksimaciju i još uvijek je najkorišteniji model, a zove se još i empirijski model.

8.2 Phongov model osvjetljenja

8.2.1 Općenito o modelu

Model prepostavlja da se cijelokupno osvjetljenje objekta, i svakog njegovog djelića, može opisati kao linearna kombinacija triju komponenata:

- ambijentna komponenta,
- difuzna komponenta, te
- zrcalna komponenta.

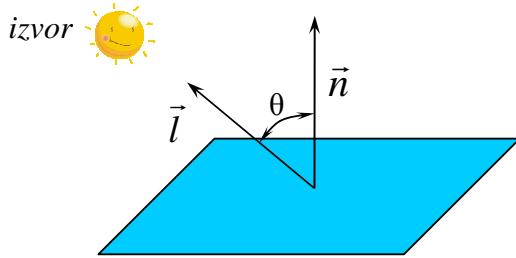
Uz to, svjetlosni izvori s kojima ovaj model barata uglavnom su točkasti. Kako u takvom scenariju sva svjetlost dopire iz jedne ili više točaka, govorimo o jednom ili više točkastih izvora.

8.2.2 Ambijentna komponenta

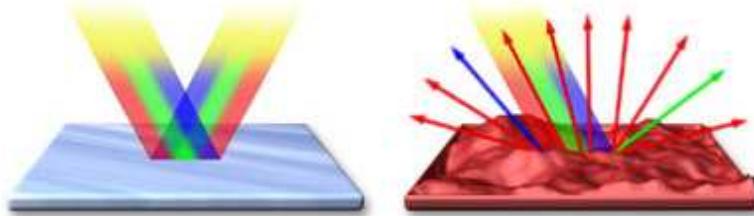
Ambijentna komponenta rezultat je interakcije svjetlosti među svim objektima u sceni opisane u uvodu. Znači, prepostavlja se da je reflektirana svjetlost koja dolazi do neke površine od svih okolnih površina konstantnog iznosa. Površine koje nisu pod izravnim utjecajem svjetlosti bit će u stvarnosti razločito osvijetljene. Ako zavirimo ispod stola i promotrimo donju stranu radne plohe, sigurno će biti manje rasvijetljena nego na primjer bočna strana stola, uz uvijet da obje plohe nisu pod izravnim utjecajem svjetlosti. No u ovom modelu, za ambijentnu komponentu uzima se konstantna:

$$I_g = k_a \cdot I_a, \quad (8.2)$$

gdje je k_a koeficijent između 0 i 1 ($0 \leq k_a \leq 1$) i daje za ovu komponentu koliko svjetlosti I_a će biti apsorbirano, a koliko reflektirano. Usprkos ovako gruboj aproksimaciji, rezultati su prihvatljivi. Ambijenta komponenta osigurava da površine koje su stražnje u odnosu na izvor svjetlosti ne budu potpuno crne, kakve bi inače bile da ove komponente nema.



Slika 8.3: Određivanje difuzne komponente svjetlosti.



Slika 8.4: Na glatkoj površini dominira zrcalna komponenta (lijevo), a na hrapavoj površini dominira difuzna komponenta (desno).

8.2.3 Difuzna komponenta

Iz fizike je poznato da intenzitet svjetlosti koji reflektira elementarna površina tijela ovisi o kutu pod kojim upada svjetlost u odnosu na normalu te elementarne površine (Lambetov zakon). Pri tome, prema slici 8.3 vrijedi:

$$I_d = I_i \cdot k_d \cdot \cos(\theta), \quad (8.3)$$

gdje je I_i intenzitet točkastog izvora, a k_d empirijski koeficijent refleksije ovisan o valnoj duljini upadne svjetlosti ($0 \leq k_d \leq 1$). θ je kut između normale površine i vektora od promatrane točke prema izvoru.

Pogledajmo sliku 8.3 i na njoj označene vektore. Vektor \vec{l} predstavlja jedinični vektor iz točke koju promatramo na površini i usmjeren je prema izvoru. Vektor \vec{n} je jedinični vektor koji predstavlja normalu u promatranoj točki. Uz te oznake, kosinus kuta θ možemo izraziti skalarnim produktom tih vektora, pa relacija (8.3) prelazi u:

$$I_d = I_i \cdot k_d \cdot (\vec{l} \cdot \vec{n}). \quad (8.4)$$

Ako I_d ispadne negativan zbog $\vec{l} \cdot \vec{n} < 0$, tada se za I_d uzima 0, drugim riječima promatrana točka za taj slučaj nije vidljiva iz izvora i zato nema doprinosa ove komponente. Stoga se uzima $\max(\vec{l} \cdot \vec{n}, 0)$.

Ako u sceni ima više točkastih izvora, tada se za ukupni intenzitet u promatranoj točki može pisati:

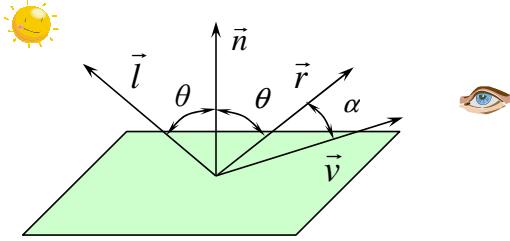
$$I_d = k_d \cdot \sum_m I_{i,m} \cdot \max(\vec{l}_m \cdot \vec{n}, 0). \quad (8.5)$$

Valja primijetiti da difuzna komponenta ne ovisi o polažaju promatrača, već samo o polažaju izvora prema promatranoj površini. Njezin utjecaj dominira na hrapavim površinama. Svjetlost koja dolazi do hrapave površine, raspršuje se podjednako u svim smjerovima kao što je prikazano na slici 8.4 (desno). Znači, mikrostruktura, odnosno hrapavost površine utjecat će na ovu komponentu.

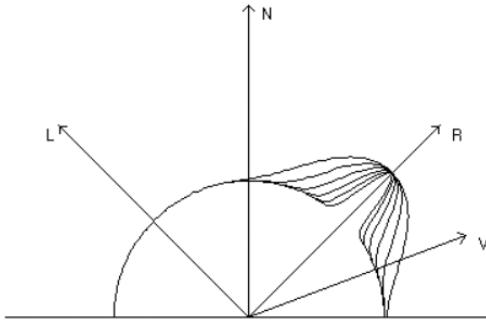
8.2.4 Zrcalna komponenta

Zrcalna komponenta, prema slici 8.5, funkcija je kuta α između reflektirane zrake \vec{r} i zrake prema promatraču \vec{v} .

$$I_s = I_i \cdot k_s \cdot \cos^n \alpha \quad (8.6)$$



Slika 8.5: Određivanje zrcalne komponente svjetlosti.

Slika 8.6: Utjecaj faktora n na zrcalnu komponentu svjetlosti.

pri čemu je I_i intenzitet izvora, k_s koeficijent ovisan o materijalu ($0 \leq k_s \leq 1$). Koeficijent n u eksponentu je indeks koji opisuje gruboću površine i njezina reflektirajuća svojstva i ne treba ga pomiješati s vektorm normale.

Prema slici 8.5 vektor \vec{r} predstavlja vektor reflektirane zrake i nalazi se u ravnini koju tvore vektori \vec{l} i \vec{n} , pod istim je kutem prema \vec{n} kao što ga zatvaraju vektori \vec{l} i vektor \vec{n} . Vektor \vec{v} predstavlja vektor usmjerjen iz točke površine prema promatraču (oku). Vektori \vec{r} i \vec{v} također su jedinični vektori. U tom slučaju kut α može se opisati skalarnim produktom $\vec{r} \cdot \vec{v}$, pa izraz (8.6) prelazi u:

$$I_s = I_i \cdot k_s \cdot (\vec{r} \cdot \vec{v})^n \quad (8.7)$$

Na slici 8.5 svi vektori su prikazani u istoj ravnini. U općem slučaju, vektor prema promatraču \vec{v} neće ležati u istoj ravnini s ostalim vektorima. Primjer numeričkog izračuna vektora \vec{r} imali smo u poglavljiju 2. u primjeru s vektorima.

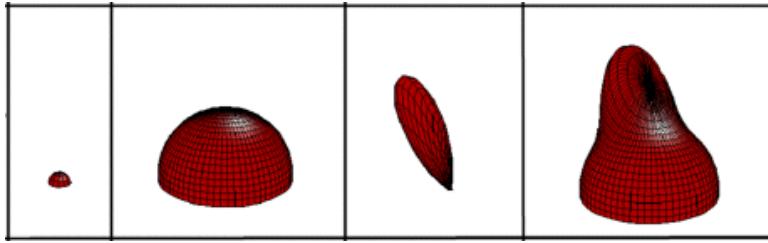
Zrcalna komponenta omogućava nam postizanje efekta blještavila. Naime, ovisno o faktoru n kut između promatrača i reflektirane zrake imat će različiti utjecaj na konačan intenzitet promatrane točke. Ako n teži u beskonačnost, površina se ponaša kao zrcalo. Naime, tada će ova komponenta postojati samo u smjeru reflektirane zrake – a to je upravo karakteristika idealnog zrcala. Ako je pak površina nešto grublja, tada ćemo imati rasipanje svjetlosti i u malim kutevima oko reflektirane zrake. To opisujemo konačnim n -om. Slika 8.6 pokazuje utjecaj indeksa n na intenzitet ove komponente uvezši u obzir položaj vektora \vec{v} .

Slika je generirana za $n = 10, 20, 40, 80, 160$ i 320 . Uz $n = 10$ dobivena je vanjska krivulja što pokazuje da je za male vrijednosti indeksa n ova komponenta prisutna u širokom spektru kuteva oko reflektirane zrake. Najveći n (320) generirao je najužu krivulju. Iz ovoga proizlazi da se spektar kuteva u kojima ova komponenta ima utjecaja smanjuje i teži prema kutu $\alpha = 0$, u kojem bi slučaju komponenta postojala samo u smjeru reflektirane zrake.

8.2.5 Ukupan utjecaj

Ukupan utjecaj iskazuje se kao linearna kombinacija sve tri komponente:

$$I = I_g + I_d + I_s = I_a \cdot k_a + I_i \cdot \left(k_d \cdot (\vec{l} \cdot \vec{n}) + k_s \cdot (\vec{r} \cdot \vec{v})^n \right) \quad (8.8)$$



Slika 8.7: Prostorna distribucija ambijentne, difuzne, zrcalne i ukupne sume svih komponenti (s lijeva na desno).

Radi jednostavnosti, model može pretpostaviti da se svjetlosni izvor, kao i promatrač, nalaze u beskonačnosti. Ova pretpostavka rezultira konstantnim vrijednostima vektora \vec{l} i \vec{v} kroz čitavu scenu, čime se izbjegava potreba za računanjem istih u svakoj promatranoj točki, pa se cijeli postupak ubrzava. No ovo povlači za posljedicu da će proizvoljno velikoj površini u svakoj točki biti pridjeljen isti vektor prema izvoru i promatraču. U model se još uvodi ovisnost intenziteta o udaljenosti izvora do površine. Fizikalno intenzitet svjetlosti opada s kvadratom udaljenosti, no eksperimentalno se to pokazalo u grafičkim primjenama kao prejak utjecaj, te se obično uzima linearna ovisnost opadanja utjecaja s udaljenošću. Tada se relacija (8.8) modificira u:

$$I = I_g + \frac{I_d + I_s}{r + k} = I_a \cdot k_a + \frac{I_i \cdot (k_d \cdot (\vec{l} \cdot \vec{n}) + k_s \cdot (\vec{r} \cdot \vec{v})^n)}{d + k} \quad (8.9)$$

pri čemu je d udaljenost promatrane točke od točke izvora, a konstanta k u nazivniku je veća od nule i sprječava dijeljenje s nulom u slučaju da je $d = 0$.

Ako se za opis svjetlosti koriste R, G i B komponente, tada izraz (8.9) treba primijeniti za sve tri komponente, pri čemu treba uočiti da k_s nije ovisan o komponenti svjetlosti. Možemo pisati:

$$I_r = I_{a,r} \cdot k_{a,r} + \frac{I_{i,r} \cdot (k_{d,r} \cdot (\vec{l} \cdot \vec{n}) + k_s \cdot (\vec{r} \cdot \vec{v})^n)}{d + k} \quad (8.10)$$

$$I_g = I_{a,g} \cdot k_{a,g} + \frac{I_{i,g} \cdot (k_{d,g} \cdot (\vec{l} \cdot \vec{n}) + k_s \cdot (\vec{r} \cdot \vec{v})^n)}{d + k} \quad (8.11)$$

$$I_b = I_{a,b} \cdot k_{a,b} + \frac{I_{i,b} \cdot (k_{d,b} \cdot (\vec{l} \cdot \vec{n}) + k_s \cdot (\vec{r} \cdot \vec{v})^n)}{d + k} \quad (8.12)$$

Prostorna distribucija pojedinih komponenti, te njihov ukupni doprinos prikazan je na slici 8.7. Vidljivo je da je doprinos ambijentne komponente relativno malen, difuzna komponenta obično dominira a nju upotpunjuje prostorno pozicionirana oko reflektirane zrake zrcalna komponenta. Karakteristike same površine utjecat će onda na doprinos pojedinih komponenti, a nama ostaje priličan broj različitih koeficijenata za podešavanje i postizanje željenog izgleda površine.

8.2.6 Primjer

Pogledajmo kako se ovaj model ponaša u najjednostavnijim slučajevima. U scenu ćemo postaviti kuglu u ishodište 3D sustava. Za prijelaz u 2D sustav koristit ćemo paralelnu projekciju, uz promatrača koji se nalazi u $+∞$ na z -osi i gleda prema ishodištu. Koristit ćemo jedan svjetlosni izvor, koji će se također nalaziti u beskonačnosti, usmjeru vektora $[1\ 0\ 1]$). Budući da gledamo odozgo, crtati ćemo samo gornji plasti kugle ($z \geq 0$).

Uz paralelnu projekciju preslikavanje iz 3D u 2D sustav vrlo je jednostavno: $x' = x$, $y' = y$, $z' = 0$. Crtanje kugle nije riješeno najefikasnije, no poslužit će za demonstraciju. Kako se cijela kugla projicirana u $z = 0$ ravninu svede na krug upisan kvadratu $-R \leq x \leq R$ i $-R \leq y \leq R$, funkcija provjerava za svaku točku te površine pripada li krugu; ako ne, ide se na sljedeću točku, a ako pripada, računa se intenzitet. Funkcija koja ovo računa prikazana je na sljedećem ispisu.

```

1 void __fastcall TForm1::Button3Click(TObject *Sender)
2 {
3     int x,y;
4     double z, I, Id, Is;
5     const int R = 100;
6     double nx, ny, nz;
7     double lx, ly, lz, rx, ry, rz, l_n_2, vx, vy, vz, naz;
8
9     lx = 1; ly = 0; lz = 1;
10    naz = sqrt(lx*lx + ly*ly + lz*lz);
11    lx = lx / naz; ly = ly / naz; lz = lz / naz;
12
13    vx = 0; vy = 0; vz = 1;
14    vz = sqrt(vx*vx + vy*vy + vz*vz);
15    vx = vx / vz; vy = vy / vz; vz = vz / vz;
16
17    for( x=-R; x<=R; x++ ) {
18        for( y=-R; y<=R; y++ ) {
19            z = R*R - (x*x + y*y);
20            if( z < 0 ) continue;
21            z = sqrt(z);
22            nz = sqrt(x*x + y*y + z*z);
23            nx = x / nz; ny = y / nz; nz = z / nz;
24            Id = lx*nx + ly*ny + lz*nz;
25            l_n_2 = 2*Id;
26            if( Id > 0. ) Id = 200*Id; else Id = 0.;
27            rx=l_n_2*nx-lx; ry=l_n_2*ny-ly; rz=l_n_2*nz-lz;
28            naz = sqrt(rx*rx + ry*ry + rz*rz);
29            rx = rx / naz; ry = ry / naz; rz = rz / naz;
30            Is = rx*vx + ry*vy + rz*vz;
31            if( Is > 0. ) {
32                Is = 45*pow(Is,80);
33            } else Is=0.;
34            I = 10. + Id + Is;
35            Image1->Canvas->Pixels[x+150][-y+150]=RGB(I,I/2,I);
36        }
37    }
38 }
```

Svaki izračunati piksel crta se na zaslonu, a budući da se dio kruga dobiva uz negativne vrijednosti x i y koordinata, ishodište je pomaknuto u točku (150, 150). Dodatno se mijenja i orientacija y -osi prema gore (na zaslonima se pozitivni y proteže prema dolje).

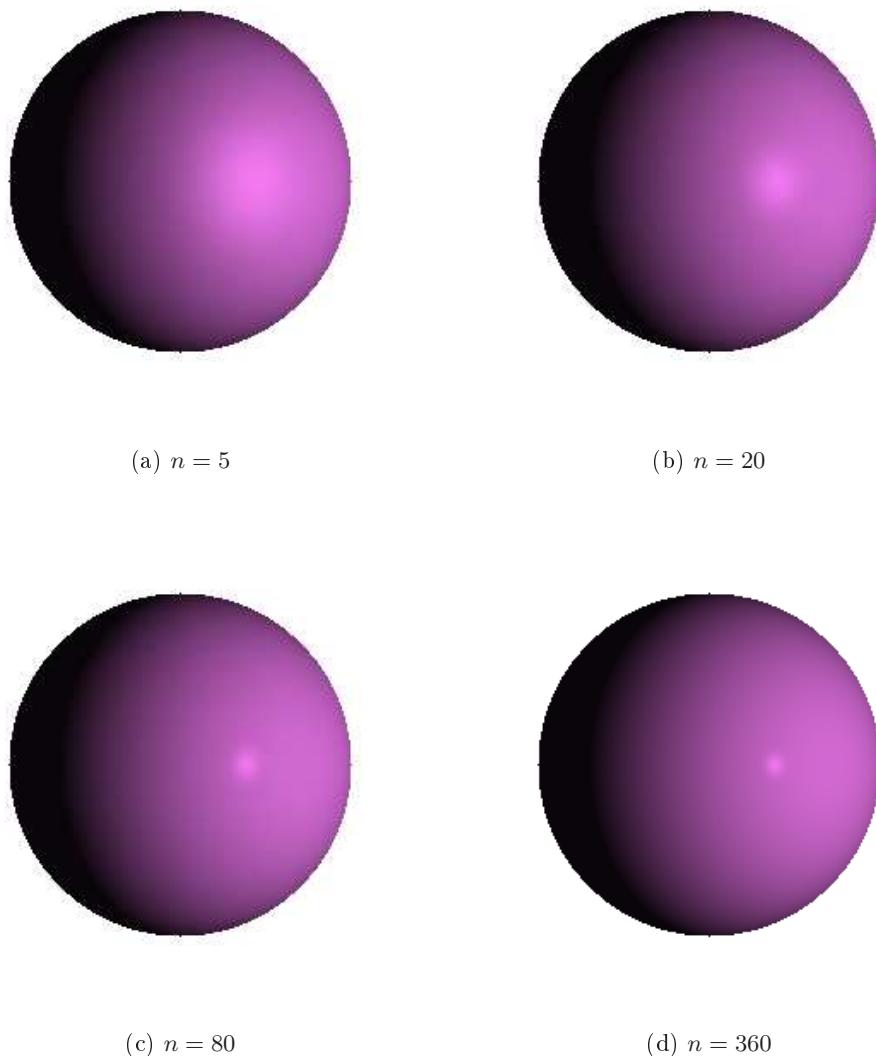
Funkcija pretpostavlja slijedeće vrijednosti:

- $I_i = 256$,
- $k_d = 0.78125 \Rightarrow I_d = I_i \cdot k_d = 200$,
- $k_s = 0.17578125 \Rightarrow I_s = I_i \cdot k_s = 45$,
- $I_g = I_a \cdot k_a = 10$, te
- $n = 80$.

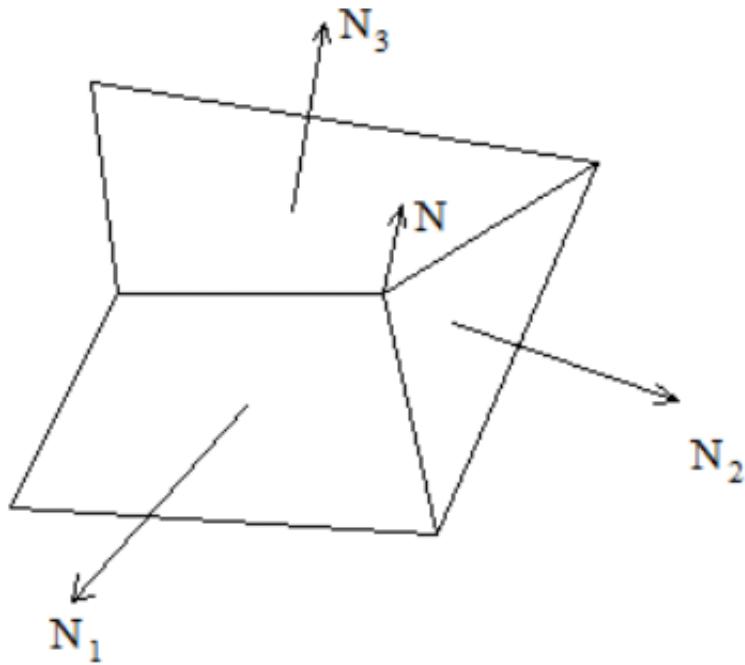
Četiri slike uz različite vrijednosti faktora n prikazane su u nastavku.

8.3 Gouraudovo sjenčanje poligona

U sekciji 8.2 opisan je Phongov model. Jedan od postupaka koji sjenčanje poligona temelje na tom modelu je Gouraudov postupak. Prema izrazu (8.4) difuzna komponenta u svakoj točki ovisi o skalarnom produktu vektora \vec{L} i vektora \vec{N} . Budući da je izvor smješten u beskonačnosti, vektor \vec{L} konstantnog je iznosa u cijeloj sceni. Kako sjenčamo poligon, a poligon je dio ravnine, tada je po cijeloj površini



Slika 8.8: Utjecaj parametra n na osvjetljavanje kugle



Slika 8.9: Izračun srednje normalne u vrh tijela

poligona vektor \vec{N} konstantan. Slijedi da je skalarni produkt tih vektora, a time i difuzna komponenta, po cijeloj površini poligona konstantna.

Tijela (zapravo njihovo oplošje) u 3D prostoru obično modeliramo nizom poligona. Sjenčanje tijela tada se svodi na sjenčanje poligona. Svaki poligon dijeli pojedine vrhove s drugim poligonima. Potrebno je u svakom vrhu pronaći srednju normalu, i pomoću nje izračunati intenzitet u tom vrhu. Srednja normala računa se kao aritmetička sredina normala svih poligona koji dijele taj vrh. Slika 8.9 prikazuje primjer.

Uz sliku vrijedi relacija $\vec{N} = \frac{1}{3} (\vec{N}_1 + \vec{N}_2 + \vec{N}_3)$, dok općenito vrijedi:

$$\vec{N} = \frac{1}{n} \sum_{i=1}^n \vec{N}_i \quad (8.13)$$

Intenzitet u vrhu računa se prema izrazu (8.8), pri čemu zanemarujemo komponentu I_s , pa slijedi:

$$I = I_a \cdot k_a + I_d \cdot k_d \cdot (\vec{L} \cdot \vec{N}) \quad (8.14)$$

Nakon što ovo izračunamo za sve vrhove, krećemo na sjenčanje poligona. Sada svaki poligon, općenito govoreći, ima u svojim vrhovima različite intenzitete. Ove intenzitete treba najprije interpolirati uzduž svih bridova (npr. postupkom DDA). Nakon ovoga poznati su intenziteti u svakoj točki svakog brida poligona. Sada još treba te intenzitete interpolirati od lijevih bridova poligona do desnih bridova poligona (opet postupkom DDA). Ideja je pokazana na slici 8.10.

Za točku (x_a, y_s) intenzitet se dobije interpolacijom između I_1 i I_4 :

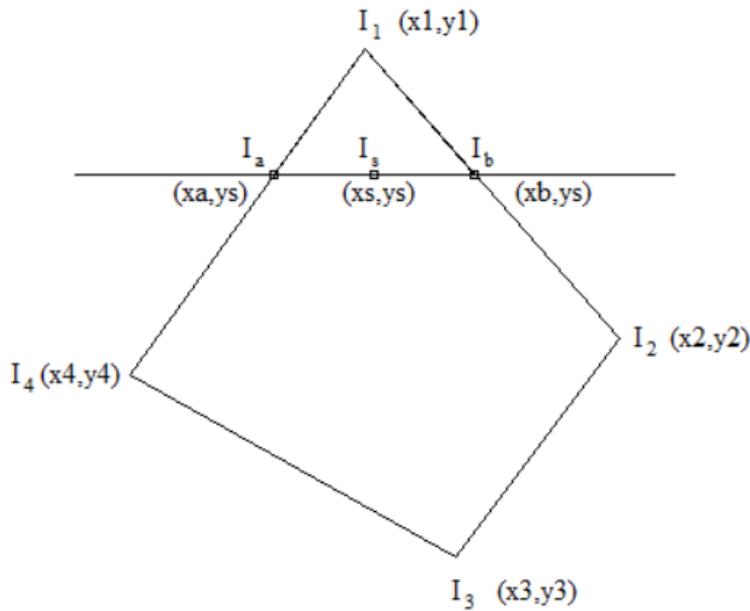
$$I_a = \frac{1}{y_4 - y_1} (I_1(y_4 - y_s) + I_4(y_s - y_1)).$$

Za točku (x_b, y_s) intenzitet se dobije interpolacijom između I_1 i I_2 :

$$I_b = \frac{1}{y_2 - y_1} (I_1(y_2 - y_s) + I_2(y_s - y_1)).$$

Intenzitet točke (x_s, y_s) dobije se interpolacijom intenziteta I_a i I_b :

$$I_s = \frac{1}{x_b - x_a} (I_a(x_b - x_s) + I_b(x_s - x_a)).$$



Slika 8.10: Interpolacija intenziteta kod Gouraudovog sjenčanja

Ovakav postupak računanja naziva se bilinearna interpolacija (naime, najprije se linearno interpoliraju intenziteti uzduž y -osi, a potom se radi interpolacija tih interpoliranih vrijednosti uzduž x -osi).

Budući da se intenzitet I_s računa za svaki piksel scan-linije koji se nalazi unutar poligona, postupak se može malo modificirati da bi se dobilo na brzini. Uvede se prirast ΔI_s :

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

pa se intenzitet može računati prema:

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

Pri tome Δx označava korak po x -osi. Ako popunjavamo svaki piksel, što je uobičajeno, Δx iznosi 1.

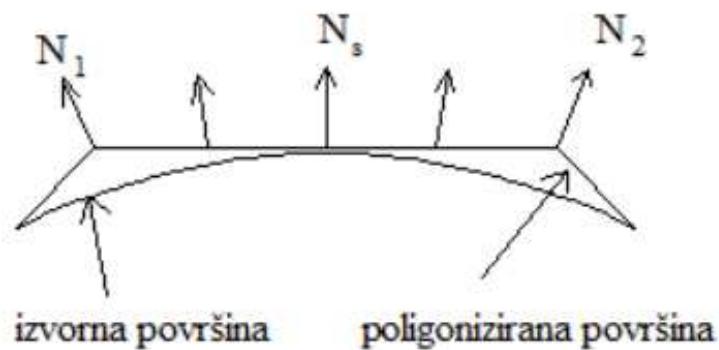
8.4 Phongovo sjenčanje

Ovo je još jedan postupak koji se zasniva na Phongovom refleksijskom modelu. Postupak je za računalo zahtjevniji, ali daje bolje rezultate od Gouraudovog sjenčanja. Postupak zadržava bilinearnu interpolaciju, ali više se ne interpoliraju intenziteti, već same normale, da bi se na kraju za svaki piksel na temelju dobivene vrijednosti za normalu izvelo računanje intenziteta na temelju izraza (8.14). Sada je vidljivo zašto je postupak puno zahtjevniji. Međutim, dobra strana postupka jest privid da interpolirana normala slijedi zakriviljenosti površine. Slika 8.11 pokazuje primjer.

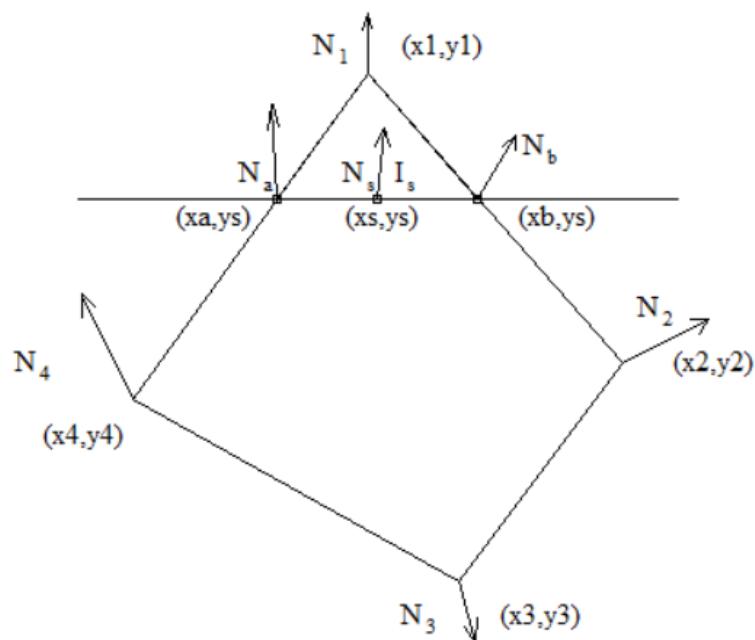
Zakriviljenu površinu pokušali smo prikazati poligonima. U tu svrhu generirano je nekoliko poligona tako da "slijede" zakriviljenost površine. Naravno, mi smo uzeli svega tri poligona (dok bi za koliko toliko realnu i glatku zakriviljenost trebalo uzeti puno više). Normale u zajedničkim vrhovima su N_1 i N_2 . Ako normalu N_s dobijemo (bi)linearnom interpolacijom, dobiva se dojam da normala slijedi zakriviljenost izvorne površine, pa očekujemo i bolje rezultate od sjenčanja.

U postupak krećemo kao i kod Gouraudovog sjenčanja: potrebno je izračunati normale u svim vrhovima prema izrazu (8.13). Zatim se izvodi bilinearna interpolacija normala, tako da u točki (x_s, y_s) dobijemo normalu N_s . Posljednji korak je izračun intenziteta u toj točki prema izrazu (8.14) gdje za N uzimamo N_s . Slika 8.12 pokazuje postupak.

Za točku (x_a, y_s) normala se dobije interpolacijom između N_1 i N_4 :



Slika 8.11: Interpolacija normala kod Phongovog sjenčanja



Slika 8.12: Interpolacija normala kod Phongovog sjenčanja, detaljniji primjer

$$N_a = \frac{1}{y_4 - y_1} (N_1(y_4 - y_s) + N_4(y_s - y_1))$$

Za točku (x_b, y_s) normala se dobije interpolacijom između N_1 i N_2 :

$$N_b = \frac{1}{y_2 - y_1} (N_1(y_2 - y_s) + N_2(y_s - y_1))$$

Normala točke (x_s, y_s) dobije se interpolacijom normala N_a i N_b :

$$N_s = \frac{1}{x_b - x_a} (N_a(x_b - x_s) + N_b(x_s - x_a))$$

Budući da se normala N_s računa za svaki piksel scan-linije koji se nalazi unutar poligona, postupak se može malo modificirati da bi se dobilo na brzini. Uvede se prirast ΔN_s :

$$\Delta N_s = \frac{\Delta x}{x_b - x_a} (N_b - N_a)$$

pa se konačna normala u promatranoj točki može računati prema:

$$N_{s,n} = N_{s,n-1} + \Delta N_s.$$

Pri tome Δx označava korak po x -osi. Ako popunjavamo svaki piksel, što je uobičajeno, Δx iznosi 1. Treba uočiti da ovdje svi izrazi opisuju vektore, što znači da se svaki izraz pri računanju raspada na tri izraza (jer se računanje provodi za svaku komponentu vektora posebno). To je još jedan razlog daleko veće računske zahtjevnosti na sklopolje koje izvodi ove kalkulacije.

Poglavlje 9

Globalni modeli osvjetljavanja

9.1 Uvod

U ovom poglavlju razmotrit ćemo globalne modele osvjetljavanja. Globalni modeli osvjetljavanja za određivanje boje svakog elementa scene u obzir uzimaju primarne svjetlosne izvore prisutne u sceni kao i utjecaj drugih objekata (sekundarnih izvora svjetlosti). Prisjetimo se, tijelo koje je osvjetljeno izvorom svjetlosti dio te svjetlosti reflektira (postajući tako sekundarni izvor svjetlosti) i time dodatno utječe na osvjetljenje drugih objekata u sceni. Phongov model osvjetljavanja ove efekte nije uzimao u obzir. Kod Phongovog modela osvjetljavanja boja pojedinih elemenata scene bila je određena samo sumom utjecaja primarnih točkastih izvora svjetlosti – tada smo govorili o lokalnom modelu osvjetljavanja. U ovom ćemo se poglavlju pozabaviti globalnim modelima. Najprije ćemo krenuti s modelom poznatim kao *bacanje zrake* (engl. *Ray Casting*), čijim ćemo proširenjem doći do prvog globalnog modela poznatog pod nazivom *praćenja zrake* (engl. *Ray Tracing*). Na kraju poglavlja opsat ćemo još jedan od računski dosta intenzivnih globalnih modela: *Radiosity*.

9.2 Algoritam bacanja zrake

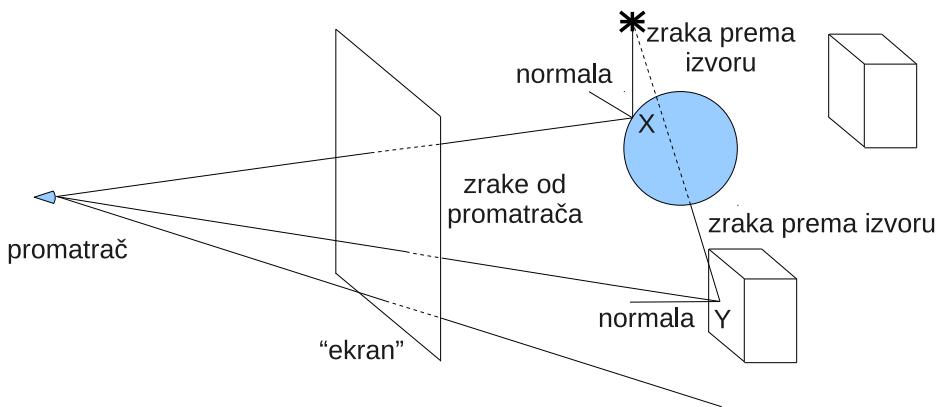
Temeljna pretpostavka modela bacanja zrake jest da se svjetlost širi pravocrtno – kao svjetlosne zrake; valna priroda svjetlosti u ovom se modelu zanemaruje. Iz točkastog izvora svjetlosti kreće beskonačno mnogo svjetlosnih zraka u svim smjerovima. Neke zrake pogađaju objekte u sceni, reflektiraju se od njih, moguće ponovno pogađaju druge objekte, reflektiraju se od njih, i u konačnici, jedan mali dio tih zraka pogada oko promatrača dok preostale zrake odlaze u beskonačnost. Dio zraka koji pogada promatračevo oko definira sliku koju promatrač vidi.

Krenemo li direktno u implementaciju ove ideje, pojavit će se vrlo neugodan problem: većina zraka koje ćemo pratiti od izvora pa u scenu neće stići do promatrača. Umjesto toga, nakon niti jedne, jedne ili nekoliko refleksija, zrake će završiti u beskonačnosti (pretpostavka je da je scena otvorena). Ovakav postupak stoga je računski vrlo skup – najveći dio zraka koje ćemo slijediti neće ni na koji način doprinositi slici koju vidi promatrač.

Vratimo se stoga na početak: pretpostavimo da se svjetlost doista širi pravocrtno, i da smo pronašli jednu zraku koja je nakon refleksije od objekta stigla do promatračevog oka. Put te zrake puno efikasnije možemo rekonstruirati obrnemo li put zrake, tako da se pretvaramo da je zraka krenula od oka promatrača – takve nas zrake uvijek zanimaju, jer one doprinose konačnoj slici koju vidi promatrač.

Ideja postupka ilustrirana je na slici 9.1. Prikazana je scena koja se sastoji od jednog izvora svjetlosti, jedne kugle te dva kvadra. Promatrač se na slici nalazi skroz lijevo. Između promatrača i objekata scene postavili smo ravninu projekcije, i u toj ravnini odabrali smo pravokutno područje koje predstavlja "ekran". Ako sliku renderiramo za prikaz na ekranu razlučivosti 1024×768 , to pravokutno područje podijelit ćemo u mrežu kvadratića koji će upravo odgovarati svakom od slikovnom elementu zaslona (pikselu). Potom ćemo za svaki kvadratič izračunati zraku koja kreće iz promatračeva oka i prolazi kroz taj kvadratič. Pratit ćemo zraku kroz scenu i pratiti koji od objekata ta zraka prvoga pogada. Dva su moguća slučaja.

Ako zraka ne probada niti jedan objekt u sceni, slikovnom elementu dodijelit ćemo intenzitet koji



Slika 9.1: Model bacanja zrake

odgovara ambijentnoj komponenti (najdonja zraka na slici 9.1). Ako zraka probada jedan ili više objekata, postupak mora pronaći promatraču najblže sjecište – to je ono što promatrač vidi. Primjeri su najgornja zraka na slici 9.1, gdje zraka probada i kuglu i kvadar (no probodište s kuglom označeno slovom X je bliže, pa se ono promatra) te srednja zraka koja probada samo donji kvadar (probodište je označeno slovom Y). Jednom kada je pronađeno najblže probodište zrake i objekta, slikovnom elementu treba dodijeliti određenu boju. Prvi korak jest provjera je li sjecište možda u sjeni. Iz sjecišta se prema svakom izvoru prisutnom u sceni stvara nova zraka (engl. *shadow rays*). Za svaku zraku sjene traže se sjecišta s objektima u sceni, kako bi se utvrdilo blokira li koji objekt svjetlost tog izvora. Ako je odgovor potvrđan, odnosno postoji sjecište zrake sjene i nekog objekta koji se nalazi između početnog sjecišta i izvora, tada izvor ne doprinosi intenzitetu sjecišta (kažemo da je točka u sjeni obzirom na promatrani izvor). Taj je slučaj prikazan za srednju zraku na slici 9.1. Ako točka nije u sjeni (primjer s gornjom zrakom na slici 9.1), računa se doprinos tog izvora i to uporabom lokalnog Phongova modela osvjetljavanja: računa se normala na objekt i temeljem nje intenzitet difuzne i zrcalne komponente. U konačnici, ako je promatrano sjecište u sjeni s obzirom na sve izvore u sceni, njegov intenzitet bit će jednak upravo ambijentnoj komponenti. U konačnici, intenzitet sjecišta bit će jednak zbroju ambijentne komponente te doprinosa difuzne i zrcalne komponente svakog izvora za koji to sjecište nije u sjeni.

Pseudokod algoritma bacanja zrake prikazan je u nastavku.

```

za svaki slikovni element (x,y)
izračunaj zraku od oka do slikovnog elementa (x,y)
izračunaj sjecišta zrake sa svim objektima u sceni
pronađi sjecište (i objekt) koje je najbliže
dodijeli boju sjecištu (klasični model osvjetljavanja)
zapiši tu boju slikovnom elementu (x,y)

```

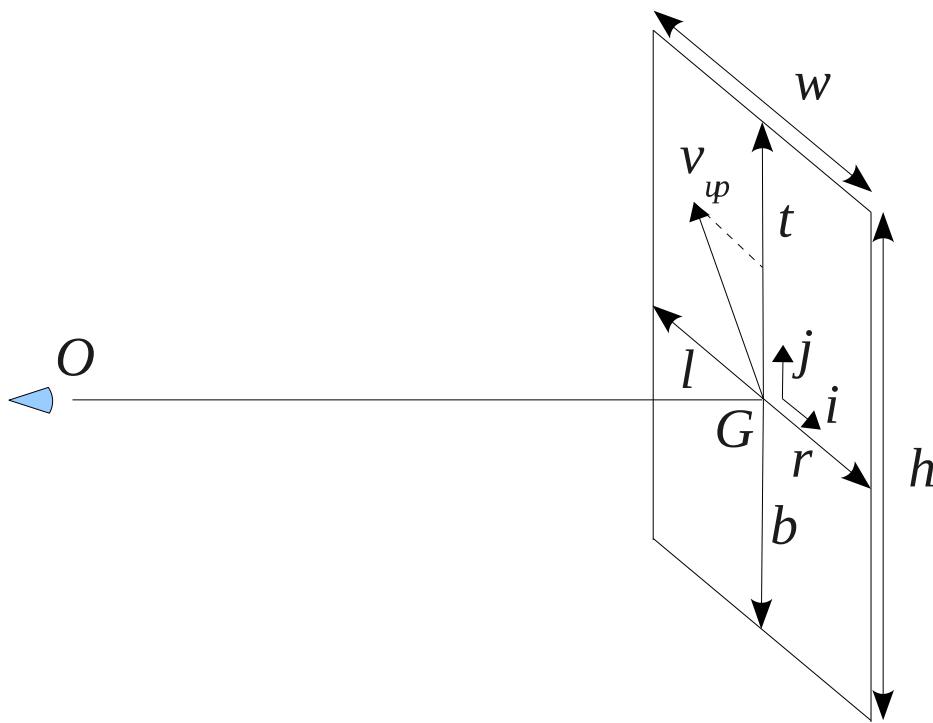
9.2.1 Matematički tretman algoritma

U sekciji 2.7 već smo dali matematički tretman problema koje je potrebno riješavati prilikom provođenja algoritma bacanja zrake, pa je sada pravo vrijeme za pažljivije čitanje te sekcije. Prisjetimo se samo najvažnijih detalja. Zraku ćemo uvijek prikazivati izrazom:

$$\vec{t}(\lambda) = \vec{T}_S + \lambda \cdot \vec{d}$$

pri čemu će \vec{d} biti normirani vektor smjera pravca dobiven kao $\vec{d} = \vec{T}_E - \vec{T}_S$. Za zraku koja iz oka promatrača prolazi kroz slikovni element (x, y) točka T_S odgovarat će očištu. Točka T_E bit će točka koja leži u ravnini "ekrana" i koja odgovara slikovnom elementu na poziciji (x, y) .

Ravninu u kojoj će ležati naš "ekran", te konkretno pravokutno područje unutar te ravnine moguće je zadati na više načina. Jedan je primjer prikazan na slici 9.2. Ravnina se zadaje preko očišta O , gledišta G te *view-up* vektora \vec{v}_{up} koji ne leži nužno u ravnini. Točka G pri tome leži u ravnini.



Slika 9.2: Definiranje ravnine ekrana

Projekcija *view-up* vektora \vec{v}_{up} u ravninu i normiranje daje vektor \vec{j} koji pokazuje pozitivan smjer y -osi. Normiranjem vektorskog produkta vektora \vec{j} i $O - G$ dobiva se vektor \vec{i} koji pokazuje pozitivan smjer x -osi.

Vektore \vec{i} i \vec{j} možemo dobiti na drugi način. Vektor \vec{i} dobit ćemo normiranjem vektorskog produkta vektora \vec{v}_{up} i $O - G$. Jednom kada imamo vektor \vec{i} , vektor \vec{j} dobit ćemo normiranjem vektorskog produkta vektora $O - G$ i \vec{i} .

Jednom kada smo utvrdili vektore \vec{i} i \vec{j} , trebamo definirati koliko je (oko gledišta) ecran širok i visok. To je moguće definirati uporabom 4 parametra:

- l - koliko se ecran proteže u lijevo od gledišta (mjereno u duljinama jediničnog vektora \vec{i}),
- r - koliko se ecran proteže u desno od gledišta (mjereno u duljinama jediničnog vektora \vec{i}),
- t - koliko se ecran proteže iznad gledišta (mjereno u duljinama jediničnog vektora \vec{j}) te
- b - koliko se ecran proteže ispod gledišta (mjereno u duljinama jediničnog vektora \vec{j}).

Uz ove podatke, još trebamo znati željenu rezoluciju ekrana, i ona je zadana parametrima w i h . Ishodište ekranskog koordinatnog sustava pri tome je smješteno dolje lijevo – točka s koordinatama $(0,0)$ odgovara donjem lijevom uglu promatranog pravokutnog područja. Označimo sada s v točku u prostoru koja leži u ravnini i odgovara slikovnom elementu na koordinatama (x,y) . Koordinate te točke su:

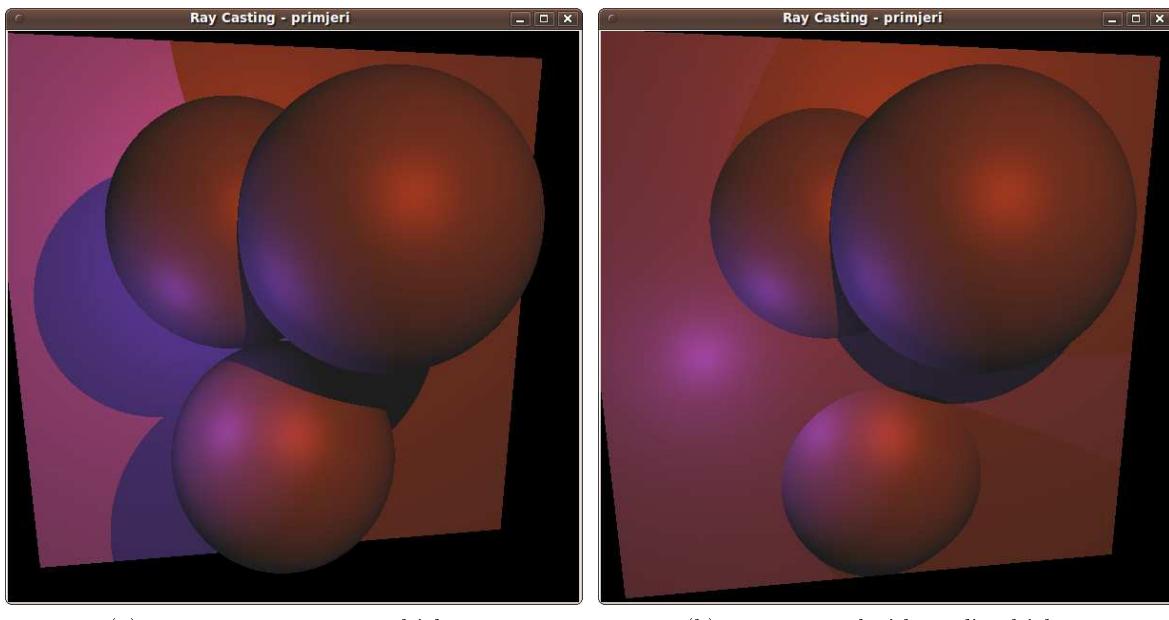
$$v = G + \vec{i} \cdot \left(-l + \frac{x}{w} \cdot (l + r) \right) + \vec{j} \cdot \left(-b + \frac{y}{h} \cdot (t + b) \right)$$

Jednom kada smo izračunali točku v krećemo sa zrakom koja polazi iz v i ima vektor smjera:

$$\vec{d} = \frac{v - O}{\|v - O\|}.$$

Nabrojimo još i neka od svojstava algoritma bacanja zrake.

- Model spada u lokalne modele osvjetljavanja.



Slika 9.3: Primjer scene prikazan algoritmom bacanja zrake

- Vidljivost se razriješava provjerom sjecišta zrake i objekata; nema potrebe za uporabom z-spremnika i sličnih podatkovnih struktura.
- Koristi je jednostavan Phongov model osvjetljavanja za određivanje intenziteta točke.
- Podržano je dobivanje grubih sjena.
- Računski vrlo zahtjevan model.

Rezultat rada ovog algoritma koji jasno prikazuje navedena svojstva prikazan je na slici 9.3.

9.3 Algoritam praćenja zrake

Algoritam praćenja zrake dodaje neke od fizikalnih zakonitosti koje smo kod algoritma bacanja zrake zanemarili, i time postaje globalni model osvjetljavanja. Naime, kada zraka svjetlosti udari u neku površinu, ovisno o svojstvima te površine mogu nastati dvije nove zrake: reflektirana zraka te lomljena zraka. Algoritam praćenja zrake stoga će za promatrano sjecište uzeti u obzir intenzitet koji definira lokalni Phongov model te dodatno još i doprinose od reflektirane zrake i od lomljene zrake. Da bi izračunao doprinos od reflektirane zrake, algoritam započinje postupak praćenja te zrake (isto vrijedi i za doprinos lomljene zrake). Prirodan način za implementaciju ovakvog algoritma jest rekurzija. Naime, kada se kreće pratiti reflektiranu zraku, ako se pronađe njezino sjecište s nekim drugim objektom, ta se zraka u tom sjecištu opet dijelom reflektira a dijelom lomi, pa je svaku od takо nastalih zraka opet potrebno pratiti kako bi se utvrdio njihov doprinos intenzitetu. Detaljniji pseudokod algoritma prikazan je u nastavku.

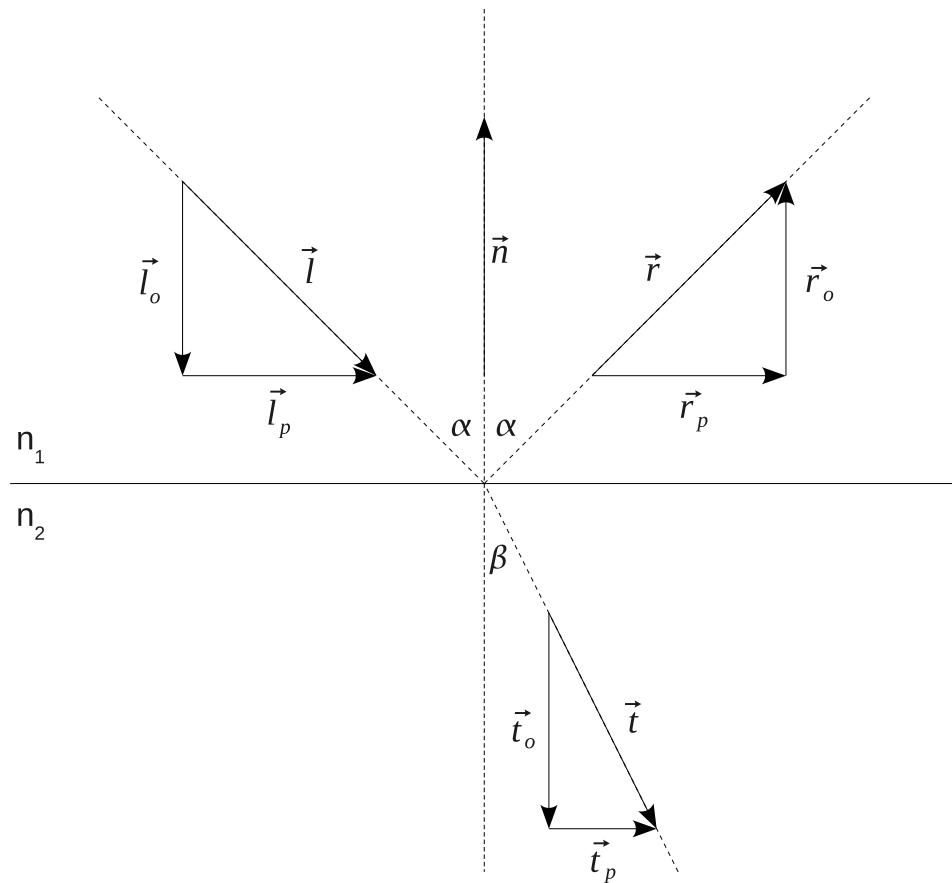
```

void raytrace()
    za svaki piksel ekrana (x,y)
        postaviBoju(x,y,slijedi(izračunaj_zraku_od_oka_do(x,y)))

rgbColor slijedi(zraka r)
    najmanji_t = infinity

    za svaki objekt o

```



Slika 9.4: Izračun reflektirane i lomljene zrake

```
t = izračunaj_sjecište(r, o)
najmanji_t = MIN(najmanji_t, t)

ako je pronađeno sjecište
    return utvrди_boju(o, r, najmanji_t)
inače
    return pozadinska_boja

rgbColor utvrdi_boju(objekt o, zraka r, double t)
point x = r(t)
rgbColor color = black

za svaki izvor svjetlosti L
    ako je najbliže_sjecište(shadow_ray(x, L)) >= udaljenost(x,L)
        color += obojaj_prema_phongu(o, x)

color += k_specular * slijedi(reflektirana_zraka(o,r,x))
color += k_transmit * slijedi(lomljena_zraka(o,r,x))

return color
```

Izračun reflektirane i lomljene zrake pojasnit ćemo u nastavku (iako je izračun reflektiranog vektora već bio obrađen u poglavljju 2). Poslužimo se slikom 9.4.

Vektori \vec{n} (vektor normale na površinu), \vec{l} (vektor zrake od izvora svjetlosti od površine), \vec{r} (vektor reflektirane zrake) i \vec{t} (vektor lomljene zrake) pri tome predstavljaju normirane vektore, i račun koji

slijedi bit će rađen uz tu pretpostavku. Za proizvoljan vektor \vec{v} oznaka \vec{v}_o predstavljat će komponentu vektora \vec{v} koja je okomita na promatranoj površini (tj. komponentu koja je kolinearna vektoru normale na slici 9.4). Oznaka \vec{v}_p predstavljat će komponentu vektora \vec{v} koja je paralelna promatranoj površini na slici 9.4.

Riješimo najprije vektor reflektirane zrake. Vektor \vec{l}_o je kolinearan vektoru \vec{n} ali je suprotnog smjera. Norma mu odgovara kosinusu kuta α (sjetimo se da je \vec{l} normiran), pa možemo pisati:

$$\vec{l}_o = -\vec{n} \cdot \cos(\alpha) = -\vec{n} \cdot ((-\vec{l}) \cdot \vec{n}) = \vec{n} \cdot (\vec{l} \cdot \vec{n}),$$

$$\vec{l}_p = \vec{l} - \vec{l}_o = \vec{l} - \vec{n} \cdot (\vec{l} \cdot \vec{n}).$$

Uočimo sada da je reflektirana zraka pod istim kutem (α) s obzirom na normalu kao i upadna zraka. Tada vrijedi:

$$\vec{r}_o = -\vec{l}_o \quad \vec{r}_p = \vec{l}_p.$$

Slijedi:

$$\vec{r} = \vec{r}_o + \vec{r}_p = -\vec{l}_o + \vec{l}_p = -\vec{n} \cdot (\vec{l} \cdot \vec{n}) + \vec{l} - \vec{n} \cdot (\vec{l} \cdot \vec{n}) = \vec{l} - 2\vec{n} \cdot (\vec{l} \cdot \vec{n}).$$

Da bismo dobili vektor lomljene zrake, prisjetimo se fizikalnih osnova loma svjetlosti koje opisuje *Snellov zakon*. Prema njemu, ako svjetlost putuje medijem čiji je indeks loma n_1 i dolazi na granicu s drugim medijem čiji je indeks loma n_2 , događa se ili totalna refleksija, ili je kut upadne i lomljene zrake određen je izrazom:

$$\frac{\sin(\alpha)}{\sin(\beta)} = \frac{n_2}{n_1}.$$

Poznavanjem upadnog kuta te indeksa lomova dolaznog i odlaznog medija n_1 i n_2 moguće je dobiti sinus kuta lomljene zrake:

$$\sin(\beta) = \sin(\alpha) \frac{n_1}{n_2}.$$

Uočimo: ako je $n_1 > n_2$, omjer $\frac{n_1}{n_2} > 1$, pa se iznos $\sin(\alpha)$ množi s brojem koji je veći od 1, čime bi se moglo dogoditi da čitav umnožak $\sin(\alpha) \frac{n_1}{n_2}$ postane veći od 1. ovo bi pak značilo da je sinus kuta β veći od jedan, što je nemoguće. Na sreću, od trenutka kada izraz $\sin(\alpha) \frac{n_1}{n_2}$ poraste do 1 (ili preko), događa se totalna refleksija: sva svjetlost koja dolazi iz izvora reflektira se, a lomljene zrake nema. Prilikom izrade algoritma praćenja zrake, ovaj uvjet treba uzeti u obzir, i lomljenu zraku slijediti samo ako ona doista postoji.

Uz pretpostavku da se lom doista događa, izvedimo sada izraz i za vektor lomljene zrake. Uočimo da vrijedi:

$$\vec{t} = \vec{t}_p + \vec{t}_o.$$

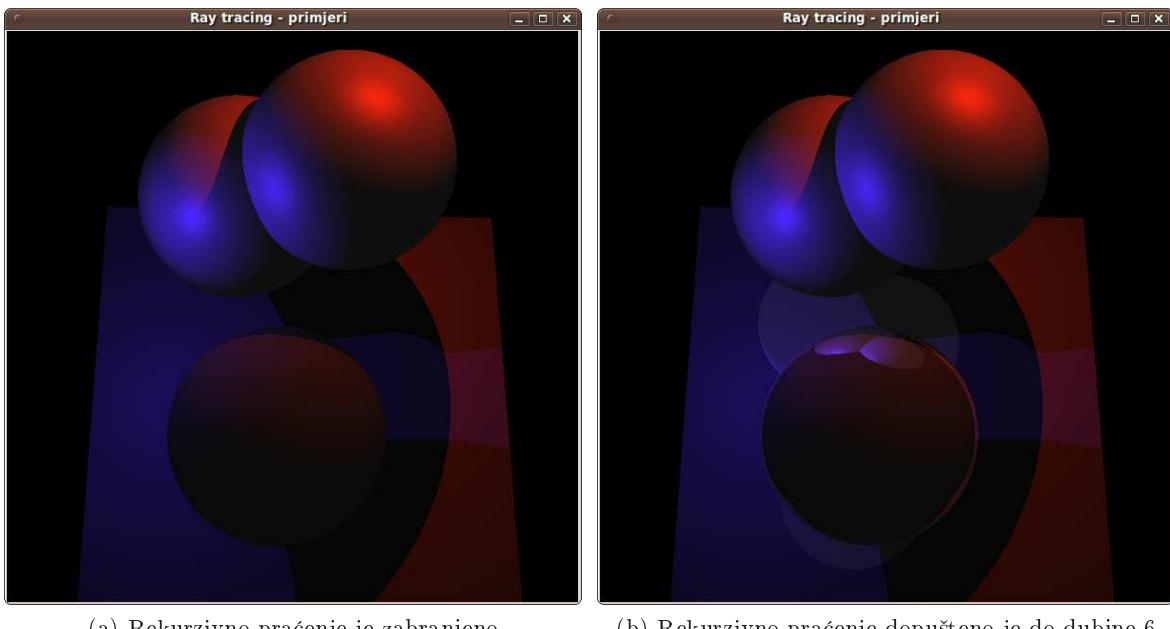
Kako je \vec{t} normiran, vrijedi $\|\vec{t}_p\| = \sin(\beta)$. Prisjetimo se i da vrijedi $\|\vec{t}_o\| = \sin(\alpha)$. Kako su $\sin(\alpha)$ i $\sin(\beta)$ povezani Snellovim zakonom, možemo pisati:

$$\sin(\beta) = \frac{n_1}{n_2} \sin(\alpha) \Rightarrow \|\vec{t}_p\| = \frac{n_1}{n_2} \|\vec{t}_o\|.$$

No, s obzirom da su \vec{l}_p i \vec{t}_p kolinearni i istog smjera, tada vrijedi:

$$\vec{t}_p = \frac{n_1}{n_2} \vec{l}_p = \frac{n_1}{n_2} \left(\vec{l} - \vec{n} \cdot (\vec{l} \cdot \vec{n}) \right).$$

Kako je komponenta \vec{t}_o kolinearna vektoru \vec{n} ali je suprotnog smjera, te s obzirom da joj je norma određena izrazom $\|\vec{t}_o\| = \cos(\beta)$ vrijedi:



Slika 9.5: Primjer scene prikazan algoritmom praćenja zrake

$$\begin{aligned}
 \vec{t}_o &= -\vec{n} \cdot \cos(\beta) \\
 &= -\vec{n} \cdot \sqrt{1 - \sin^2(\beta)} \\
 &= -\vec{n} \cdot \sqrt{1 - \frac{n_1^2}{n_2^2} \sin^2(\alpha)} \\
 &= -\vec{n} \cdot \sqrt{1 - \frac{n_1^2}{n_2^2} (1 - \cos^2(\alpha))} \\
 &= -\vec{n} \cdot \sqrt{1 - \frac{n_1^2}{n_2^2} (1 - (\vec{l} \cdot \vec{n})^2)}.
 \end{aligned}$$

Konačno, traženi vektor lomljene zrake tada je:

$$\begin{aligned}
 \vec{t} &= \vec{t}_p + \vec{t}_o \\
 &= \frac{n_1}{n_2} \left(\vec{l} - \vec{n} \cdot (\vec{l} \cdot \vec{n}) \right) - \vec{n} \cdot \sqrt{1 - \frac{n_1^2}{n_2^2} (1 - (\vec{l} \cdot \vec{n})^2)} \\
 &= \frac{n_1}{n_2} \vec{l} - \vec{n} \left(\frac{n_1}{n_2} \cdot (\vec{l} \cdot \vec{n}) + \sqrt{1 - \frac{n_1^2}{n_2^2} (1 - (\vec{l} \cdot \vec{n})^2)} \right)
 \end{aligned}$$

Prilikom implementacija algoritma praćenja zrake, nužno je uvesti još jedno ograničenje: maksimalnu dubinu rekurzije do koje smo spremni slijediti zrake. Ovo je posebice važno uzmememo li u obzir da se svaka zraka koja pogodi objekt razbija u dvije nove (reflektiranu i lomljenu), čime je porast broja zraka s dubinom eksponencijalan.

Rezultat rada ovog algoritma koji jasno prikazuje navedena svojstva prikazan je na slici 9.5.

9.3.1 Jednostavno preslikavanje tekstura

U ovom ćemo se poglavlju još ukratko osvrnuti na najjednostavniji način dodavanja tekstura na sferne objekte. Prepostavimo za početak da imamo pripremljenu teksturu, kao što je to prikazano na slici 9.6.



Slika 9.6: Primjer teksture za sferu

Prepostavimo sada da smo praćenjem zrake pogodili sferu koja ima definiranu teksturu, i neka je točka probodišta T . Za točku T koja leži na površini sfere potrebno je izračunati sferne koordinate. Označimo s β kut koji zatvaraju vektori razapeti između sjevernog pola sfere i centra sfere, te između točke T i centra sfere. Ako se točka T nalazi na sjevernom polu, taj će kut biti 0; ako se točka nalazi na ekvatoru kut će biti 90 a ako se točka nalazi na južnom polu, kut će biti 180 stupnjeva. Uočimo, dakle, da je raspon kuta β od 0 do 180 stupnjeva.

Označimo sada s α kut koji zatvara vektor razapet između projekcije točke T u ravninu u kojoj leži ekvator i centar sfere te vektor koji iz centra sfere pokazuje u smjeru nultog kuta (u analogiji sa Zemljom, to bi bio vektor koji iz središta Zemlje probada ekvator točno kroz meridijan koji prolazi kroz Greenwich). Ovaj kut treba izmjeriti pazeći na smjer, tako da se sud o kutu ne može donijeti samo temeljem skalarnog produkta tih vektora. Ovaj kut kreće se od 0 do 360 stupnjeva. U analogiji sa Zemljom, kut β predstavlja latitudu a kut α longitudu.

Jednom kada imamo kuteve α i β , potrebno je utvrditi koju to točku predstavlja u teksturi. No to je trivijalno. Neka je tekstura dimenzije $w \times h$. Treba pogledati piksel na koordinatama (x,y) i preuzeti odgovarajuće RBG komponente kao intenzitet probodišta. Koordinate (x,y) određene su izrazima:

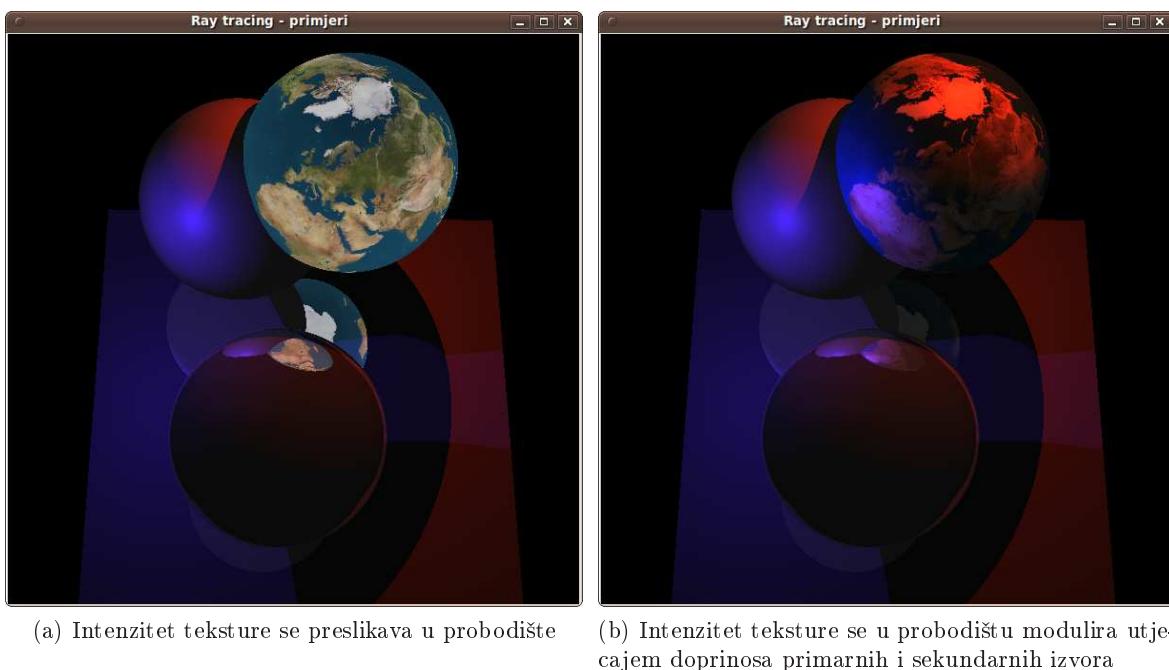
$$x = \frac{\alpha}{360} * w \quad y = \frac{\beta}{180} * h.$$

Kutevi α i β mogu se odrediti poznavanjem vektora normale \vec{n} u točki probodišta prema izrazima:

$$\alpha = \arctan(n_y, n_x) \quad \beta = \arccos(n_z).$$

Funkcija \arctan koju ovdje koristimo je funkcija koja prima dva argumenta, i temeljem toga može odrediti korektan kut od 0 do 360 stupnjeva. Primjer slike nastao ovakvom primjenom teksture prikazan je na slici 9.7a.

Konačno, kako bi se dobio utjecaj okolnog osvjetljenja, umjesto da se točki probodišta dodijeli intenzitet preuzet iz tekture možemo posegnuti za malo kompleksnjim načinom. Najprije za točku na klasičan način izračunamo pripadni intenzitet (označimo ga s I_1). Potom iz tekture dohvatimo intenzitet pripadne točke (označimo ga s I_2). Prepostavimo također da su I_1 i I_2 zapravo trokomponentni (imaju r, g i b komponente čiji je iznos od 0 do 1). Konačni intenzitet izračunat ćemo kao $I_1 \cdot I_2$ i to ćemo uzeti kao intenzitet probodišta. Primjer slike nastao ovakvom modulacijom intenziteta teksture prikazan je na slici 9.7b.



(a) Intenzitet teksture se preslikava u probodište

(b) Intenzitet teksture se u probodištu modulira utjecajem doprinosa primarnih i sekundarnih izvora

Slika 9.7: Primjer praćenja zrake uz preslikavanje teksture na sferni objekt

Poglavlje 10

Boje

10.1 Uvod

Što su boje? Kako čovjek vidi boje? Kako možemo izmjeriti "boje"? Ovo su samo neka od pitanja na koje ćemo pokušati dati odgovor kroz ovaj tekst. Boje su prvenstveno vezane uz pojam svjetlosti. Čovjek vidi predmete oko sebe zahvaljujući razvijenim senzorima za elektromagnetske valove u području valnih duljina koje nazivamo vidljiva svjetlost. Ovo područje obuhvaća valne duljine od 390 do 760 nm, odnosno frekvencijski spektar između 394.7 i 769.2 THz. Kod čovjeka postoje dvije vrste senzora koji su se specijalizirali za dvije različite namjene, vjerojatno kao prilagodba na čovjekovu okolinu. Naime, čovjek živi, kada o svjetlosti govorimo, u dva različita okruženja: danju i noću.

Noć je karakteristična po tome što svjetlosti ima vrlo malo, odnosno vrlo je slabog intenziteta. U takvom okruženju bitno je raspoznavati osnovne elemente okoline. Bitno je razaznavati različite intenzitete svjetlosti, i na temelju toga raspoznavati objekte koji ga okružuju. U ovakvoj okolini nema dovoljno informacija, a niti potrebe, za raspoznavanjem boje. I upravo za ovakvo okruženje razvila se je prva vrsta osjetila – *štapići*. To su osjetila koja raspoznaju različite intenzitete svjetlosti, i mogu "raditi" već pri vrlo malim količinama svjetlosti.

Za razliku od noći, tijekom dana raspoloživa količina svjetlosti je vrlo velika. Toliko velika, i na toliko različitim valnim duljinama, da se na temelju te količine svjetlosti može dobiti puno više informacije od običnog intenziteta – možemo raspoznavati boju. Za ovaj režim rada razvijen je drugi tip osjetila – *čunjići*. Pomoću tih osjetila čovjek je sposoban primati informaciju o boji predmeta iz okoline. U čovjekovom oku postoji tri tipa ovih osjetila: osjetilo za crvenu boju, osjetilo za zelenu boju te osjetilo za plavu boju. Naravno, odmah se postavlja pitanje kako onda vidimo npr. žutu boju, kada za nju nemamo osjetila. Odgovor na ovo pitanje dat će u nastavku.

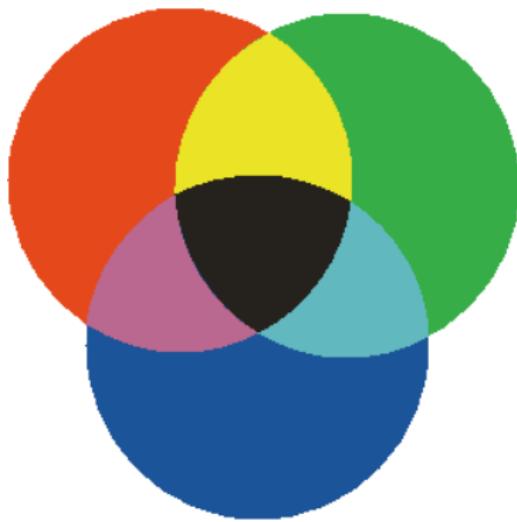
10.2 Sheme za prikaz boja – prostori boja

10.2.1 Modeli

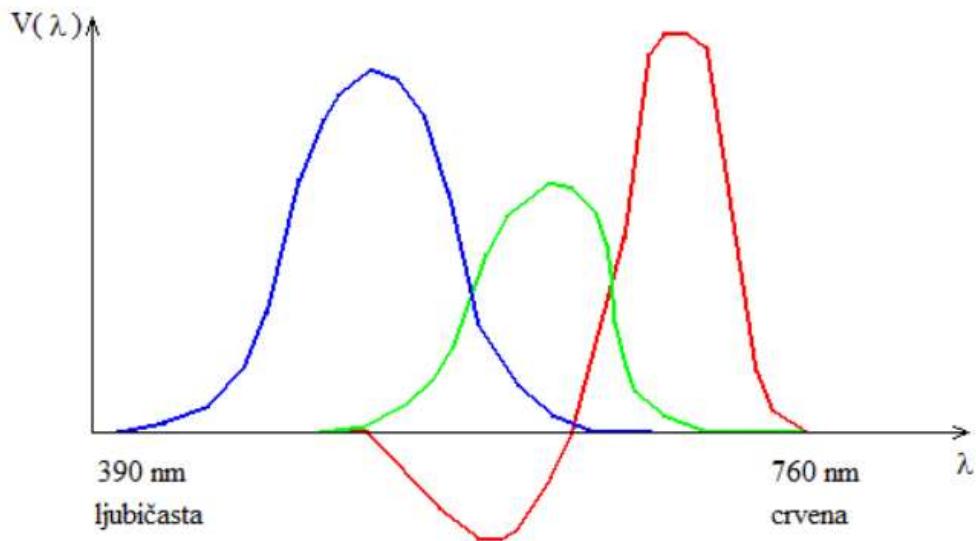
Tijekom proučavanja svjetlosti i problematike vezane uz boje razvijeno je nekoliko modela kojima se pokušava dobiti kontrola nad bojama. Prvi i osnovni model proizašao je iz otkrića kako čovjek vidi boje. Tako je nastao *RGB-model*. Slova u nazivu modela su početna slova od engleskih naziva boja: *Red* (crvena), *Green* (zelena) te *Blue* (plava). Teorijskim razmatranjima razvijen je *XYZ-model*. Iz tehničkih razloga razvijen je i *CMY-model* (Cyan, Magenta, Yellow), a potom i *CMYK-model* (Cyan, Magenta, Yellow, black). Radi lakoće odabiranja boja nastao je *HSV-model* (Hue, Saturation, Value). Razvijeni su još i modeli razreda *Y* (engl. *class Y models*) kao rezultat razvitka TV i video tehnike, i drugi.

10.2.2 RGB-model

RGB-model proizašao je iz pokušaja da se imitira način na koji čovjek vidi boje. Naime, čovjek ima osjetila za tri osnovne (primarne) boje: crvenu, zelenu i plavu. Uvezši tu činjenicu u obzir, prepostavilo se da se sve boje mogu prikazati kao linearna kombinacija ovih triju osnovnih boja. Na ovaj način, da



Slika 10.1: Aditivno miješanje boja

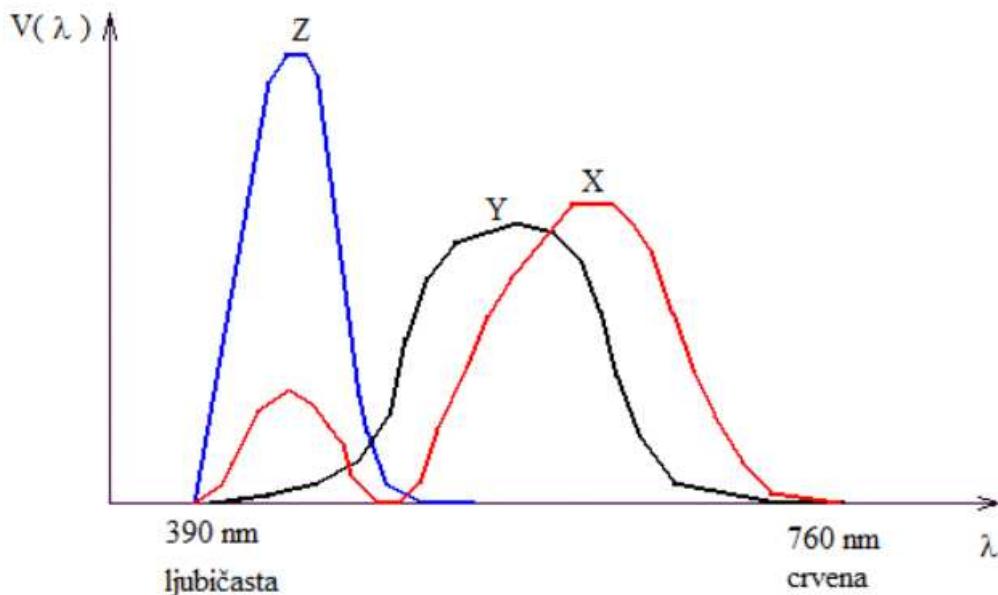


Slika 10.2: Komponente RGB potrebne za prikaz svih boja

bismo pamtili boju jedne točke, trebamo pamtiti tri broja: koliko imamo crvene, koliko imamo zelene, te koliko imamo plave. Ako sve tri komponente iznose nula, dobiti ćemo crnu. Ako sve tri komponente iznose maksimum, dobiti ćemo bijelu. Ako su pojedine komponente prisutne s različitim udjelima, dobiti ćemo različite boje. Nijanse sivih boja dobivat ćemo ako sve tri osnovne boje držimo jednakе po iznosu, i taj iznos variramo. Fizikalno si ovaj proces možemo zamisliti na slijedeći način: imamo na raspolaganju tri svjetlosna izvora: izvore crvene, zelene i plave. Sva tri izvora usmjerena su u istu točku, i ne postoji niti jedan drugi izvor svjetlosti. Ako su sva tri izvora ugašena, točka je crna jer nema nikakve svjetlosti koja bi je obasjala. Počnemo li zatim paliti pojedine izvore, točka će poprimati različite boje jer će doći do miješanja boja. Zbog ovog svojstva da se pojedine komponente svjetlosti zbrajaju, ovo miješanje zovemo *aditivno miješanje*. Proses je prikazan na slici 10.1.

Opisani model vrlo je jednostavan. Međutim, ima jednu manu. Sve boje ne mogu se opisati pomoću ovih triju primarnih boja, barem ne ako imamo u vidu fizikalnu sliku miješanja boja. Naime, pokazuje se da bi se sve boje mogle prikazati kada bi crveni izvor, osim emitiranja crvene svjetlosti mogao i oduzimati crvenu svjetlost – ni iz čega. Naime, za prikaz svih boja, koeficijent crvene boje protezao bi se i u pozitivnom području, i u negativnom području, kao što to prikazuje slika 10.2.

Ovaj nedostatak rezultirao je sastankom CIÉ (Commission Internationale de l'Éclairage) gdje se je pokušao definirati model koji bi također imao samo tri primarne boje X, Y i Z, i koji bi opisivao sve



Slika 10.3: Komponente XYZ potrebne za prikaz svih boja

boje uz pozitivne koeficijente. Rezultat je prikazan na slici 10.3. CIÉ je također definirala kromatski dijagram koji opisuje sve boje. Dijagram je prikazan na slici 10.4.

Ovaj dijagram ima veliku važnost, utoliko što omogućava da bolje shvatimo našu ograničenost u prikazivanju boja. Naime, ukoliko imamo na raspolaganju tri izvora boja (npr. izvore crvene, zelene i plave boje), boje koje možemo prikazati u kromatskom se dijagramu nalaze unutar trokuta koji zatvaraju te tri boje (slika 10.5). Ovaj trokut naziva se *GAMUT*.

Boje koje se nalaze izvan tog trokuta, ne mogu se prikazati kombinacijom tih triju boja. Ako pokušamo umjesto tri izvora uporabiti četiri – ništa se ne mijenja; lik sada neće biti trokut, već četverokut, i površina će biti nešto veća. No opet će biti boje koje ne možemo prikazati.

Model RGB koristi se u uređajima u kojima je moguće ostvariti tri izvora boja i njihovo miješanje, na način kako smo to opisali. To u praksi znači da će se model koristiti prvenstveno u monitorima. Općenito vrijedi: trokut koji razapinju boje topova u monitoru naziva se *GAMUT uredaja*.

10.2.3 CMY/CMYK-model

Model CMY slijedi upravo obratnu filozofiju od RGB-modela. Pretpostavka je da bez ikakvih utjecaja na točku boja točke mora biti bijela. Boje ćemo dobiti tako da od bijele oduzimamo pojedine komponente u različitim iznosima. Osnovne boje koje se ovdje koriste su *Cyan* (cijan), *Magenta* (neka verzija ljubičaste) i *Yellow* (žuta). Ovo je primjer subtraktivnog modela, i to je komplementarni model RGB-modelu. Štoviše, ukoliko normiramo pojedine komponente RGB-modela (dakle, svake boje ima u iznosu od 0 do maksimalno 1), tada vrijedi relacija:

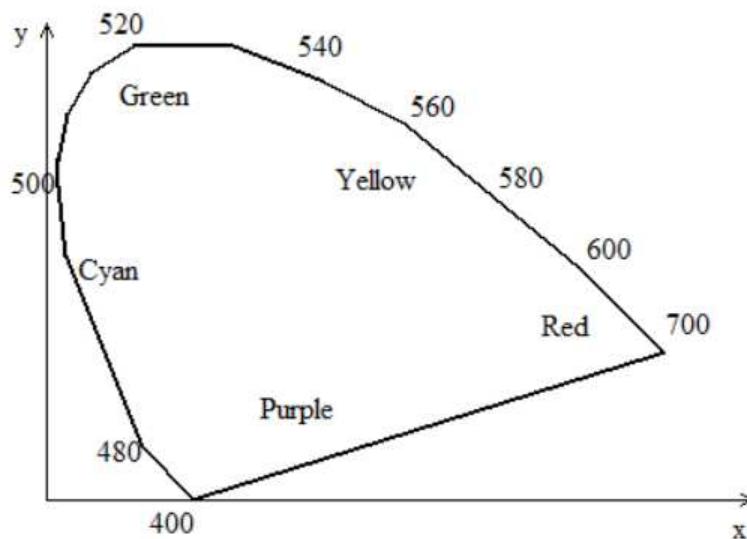
$$C = 1 - R$$

$$M = 1 - G$$

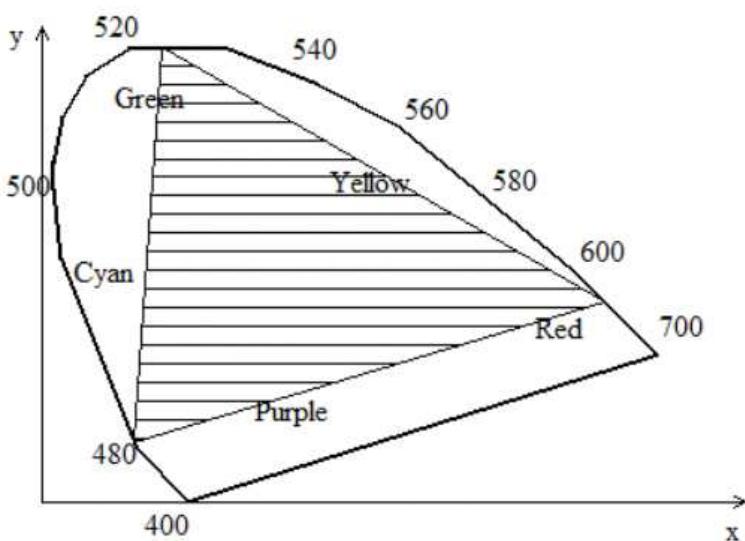
$$Y = 1 - B$$

Slika 10.6 prikazuje miješanje ovih osnovnih boja.

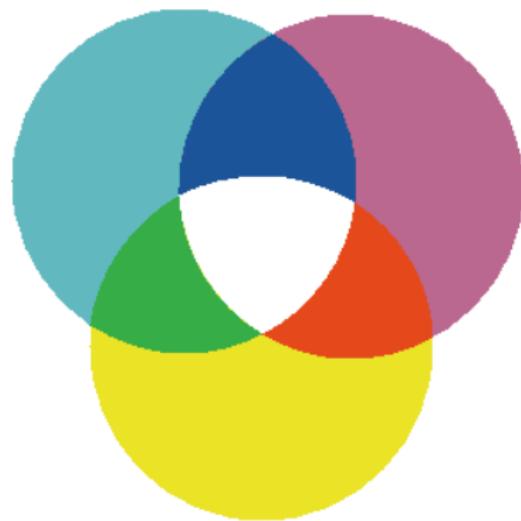
Uvjet da je točka sama po sebi bijela ukoliko na nju ne djeluje ništa nalazimo upravo kod bijelog papira – zbog toga se ovaj model koristi upravo kod printer-a. Zapravo, kod printer-a se koristi CMYK model, koji je ekonomiziran CMY model. Naime, teorijske pretpostavke i praksa obično se baš i ne slažu najbolje, pa tako maksimalnim miješanjem boja CMY dobivamo crnu koja baš i nije onako crna kako bismo to željeli. S druge strane, većina dokumenata koja se danas još uvijek ispisuje obilato koristi upravo crnu boju, što za ovaj model znači maksimalnu potrošnju svih boja – a uopće ne ispisujemo u



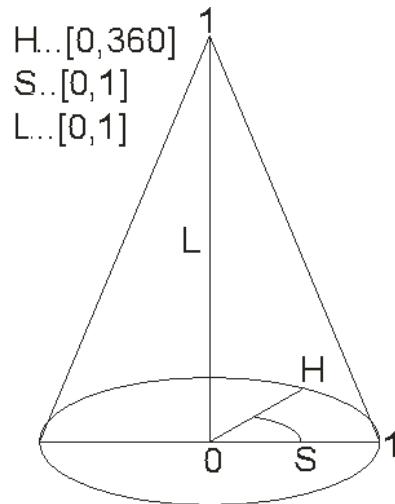
Slika 10.4: Kromatski dijagram



Slika 10.5: GAMUT boja



Slika 10.6: CMY-model boja



Slika 10.7: HLS-model boja

H vrijednost	Odgovarajuća boja
0°	Plava
60°	Magenta
120°	Crvena
180°	Žuta
240°	Zelena
300°	Cijan

Tablica 10.1: Odabir boja kod HLS-modela

boji. Zbog tih razloga, u model je uvedena još i crna boja, te je model dobio u oznaci slovo K od riječi blacK. Tako se uvode slijedeće relacije:

$$K = \min(C, M, Y)$$

$$C = C - K$$

$$M = M - K$$

$$Y = Y - K$$

10.2.4 HLS-model

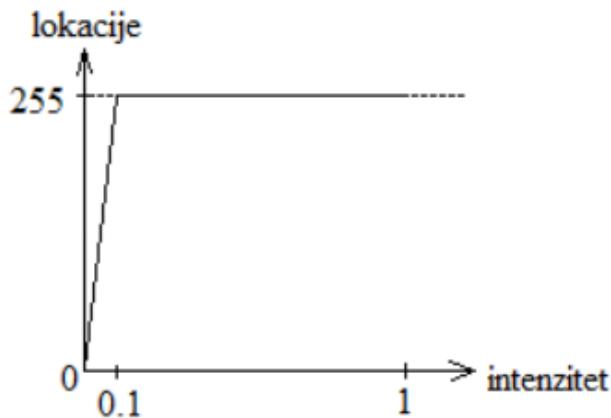
HLS model (Hue, Lightnes, Saturation) nastao je iz potrebe da se olakša odabir boje. Naime, prethodno opisani modeli vrlo su nezgrapni kada je riječ o odabiru boje. Zamislite da ste našli kombinaciju koja vam daje neku nijansu narančaste boje, i sada želite dobiti samo malo zasićeniju boju, ili pak malo svjetliju. Što učiniti da bi se to postiglo? Problem je u tome što treba mijenjati sve tri komponente RGB-modela, a to znači smrt jednostavnoj uporabi. Ideja je da promjenom vrijednosti H obilazimo sve boje. Kada pronađemo odgovarajuću boju, s L odredimo željenu svjetlinu, te sa S njezinu zasićenost. Pri tome komponente H , L i S čine stožac, prikazan na slici 10.7.

H je predstavljen kutom u odnosu na pozitivnu x -os, i njime se biraju boje. Vrijedi slijedeće:

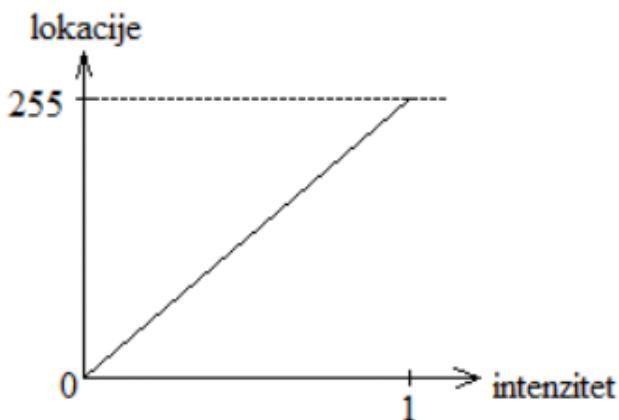
Ako zasićenost stavimo na nulu ($S = 0$), tada promjenom svjetline (L) od 0 do 1 dobivamo sve nijanse sive boje, uključujući crnu ($L = 0$), i bijelu ($L = 1$).

10.3 Gamma korekcija

Zamislimo da se nalazimo pred slijedećim problemom: imamo na raspolaganju 256 lokacija na kojima možemo pamtitи intenzitet boje. Intenzitet je u rasponu od 0 do 1 (0 je minimalni, 1 maksimalni).



Slika 10.8: Linearna raspodjela intenziteta



Slika 10.9: Linearna raspodjela intenziteta (2)

Potrebno je odrediti koliki intenzitet ćemo pohraniti u pojedinu lokaciju, a da pri tome dobijemo takvu raspodjelu koja će biti uočljiva ljudskom oku. Npr. intenzitete možemo raspodijeliti prema slici 10.8.

Lokacija 0 sadržavati će intenzitet 0, lokacija 255 intenzitet 0.1, dok će intenziteti između 0 i 0.1 biti linearno pridjeljeni lokacijama 1 do 254. Međutim, ovakva raspodjela nije dobra jer ćemo cijelu sliku nacrtanu pomoću ovakvih intenziteta jedva vidjeti – naime, intenziteti su raspodijeljeni kao da je na snazi opća opasnost i metode zamračivanja. To nije dobro. Možda je bolja raspodjela prikazana na slici 10.9.

Tu je intenzitet također linearno raspodijeljen, i to tako da se u pojedinim lokacijama nalaze i najmanji, a u pojedinim i najveći intenziteti. To je, dakako, bolje. Međutim... Ljudsko oko intenzitetu ne raspoznaće baš na ovaj način. Naime, da bi oko primijetilo da se je intenzitet povećao, to povećanje mora biti za potenciju veće. Naime, karakteristika ljudskog oka je logaritamska. To znači da oko primijeti intenzitet koji je veći za 1 tek kada logaritam tog intenziteta poraste za jedan, a logaritam poraste za jedan samo ako potencija poraste za jedan. Dakle, umjesto linearne raspodjele intenziteta sa slike 10.9:

$$I_0 = I_0$$

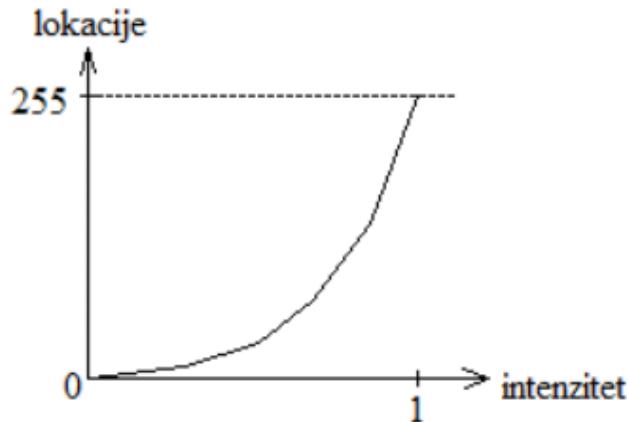
$$I_1 = r \cdot I_0$$

$$I_2 = (r + r) \cdot I_0$$

...

$$I_n = (r \cdot n) \cdot I_0$$

treba napraviti eksponencijalnu raspodjelu intenziteta:



Slika 10.10: Eksponencijalna raspodjela intenziteta

$$\begin{aligned}
 I_0 &= I_0 \\
 I_1 &= r \cdot I_0 \\
 I_2 &= r^2 \cdot I_0 \\
 &\dots \\
 I_n &= r^n \cdot I_0
 \end{aligned} \tag{10.1}$$

Ujedno znamo da je I_n upravo jednak 1, pa možemo izračunati faktor r :

$$I_n = r^n \cdot I_0 = 1 \Rightarrow r = \left(\frac{1}{I_0} \right)^{\frac{1}{n}} \tag{10.2}$$

U općem slučaju se, dakle, može napisati formula:

$$I_j = r^j \cdot I_0 = \left(\left(\frac{1}{I_0} \right)^{\frac{1}{n}} \right)^j \cdot I_0 = (I_0)^{1 - \frac{j}{n}} \tag{10.3}$$

Kako u našem slučaju imamo 256 razina, odnosno razine od 0 do 255, vrijedi:

$$r = \left(\frac{1}{I_0} \right)^{\frac{1}{255}}, \quad I_j = (I_0)^{1 - \frac{j}{255}}, \quad j \in \{0, 1, \dots, 255\}$$

Ovakva raspodjela prikazana je na slici 10.10.

Pogledajmo sada situaciju sa klasičnim CRT monitorom. Osvjetljenje neke točke na zaslonu monitora rezultat je pogađanja te točke (odnosno fosfora u toj točci) zrakom elektrona. Intenzitet tako dobivene svjetlosti to je veći, što više elektrona pogađa tu točku u jedinici vremena. Matematički se ovisnost intenziteta može opisati relacijom:

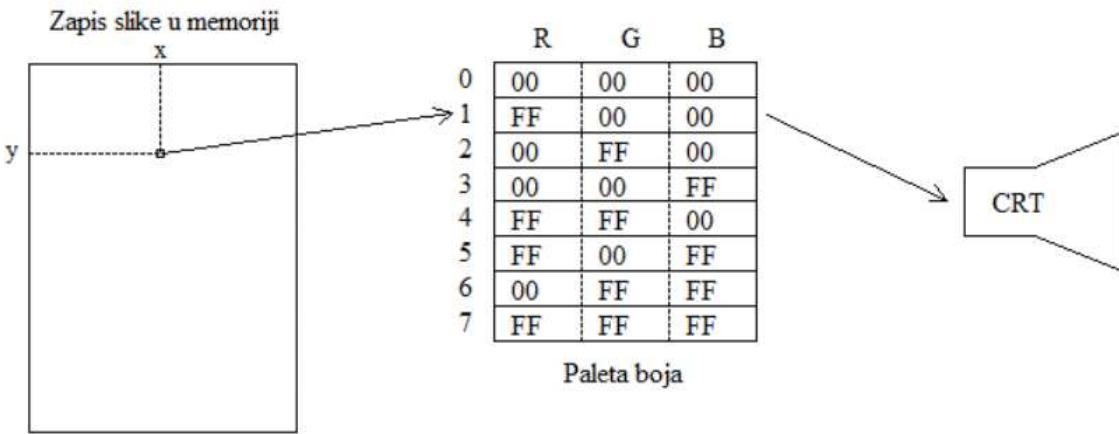
$$I = k \cdot N^\gamma \tag{10.4}$$

gdje je I intenzitet koji emitira fosfor, a k i γ konstante. Broj elektrona proporcionalan je naponu koji uzrokuje pojavu, te vrijedi:

$$I = k \cdot (k_1 \cdot V)^\gamma = k \cdot k_1^\gamma \cdot V^\gamma = K \cdot V^\gamma \tag{10.5}$$

gdje je I intenzitet koji emitira fosfor a k , k_1 , K i γ konstante. Ako je poznat intenzitet koji želimo dobiti, napon koji je potreban za njegov prikaz dobiti ćemo izjednačavanjem relacija (10.1) i (10.5):

$$K \cdot V_j^\gamma = r^j \cdot I_0 \Rightarrow I_j = \left(r^j \cdot \frac{I_0}{K} \right)^{\frac{1}{\gamma}} \tag{10.6}$$



Slika 10.11: Pamćenje boja u paleti boja

Treba uočiti da je γ karakteristika samog monitora, pa će zbog toga na različitim monitorima odgovarajući naponi biti različiti – ili pak isti, ali tada će prikazane slike biti različite, a to ipak ne želimo.

10.4 Pamćenje boja na računalu

Rad s bojama na računalima izведен tako da se može prilagoditi potrebama i memorijskim zahtjevima, odnosno raspoloživim memorijskim resursima samog računala. Osnovni način prezentacije boje na računalu je putem RGB-sustava. Dakle, za svaku boju pamte se tri komponente: crvena, zelena i plava. Međutim, tu postoje dva moda rada:

- uporaba paleta boja, te
- direktna reprezentacija boje.

Prva metoda razvijena je zbog uštede memorije, i time naravno nameće određena ograničenja. Glavni faktor koji određuje koliko ćemo različitih boja moći prikazati naziva se dubina. Ovisnost dubine boja i broja mogućih različitih boja dana je relacijom (10.7):

$$n = 2^d \quad (10.7)$$

gdje je n broj raspoloživih boja a d dubina.

Dubina se mjeri u broju bitova. Ideja je da se formira tablica koja ima onoliko elemenata koliko mi to odredimo preko dubine. Svaki taj element sadržavat će tri komponente: količinu crvene, količinu zelene te količinu plave boje. Boja pojedinog piksela (npr. boja j) određivat će se brojem od 0 do $n-1$, i odgovarat će onoj boji čija se definicija nalazi u odgovarajućem retku tablice (dakle, j -ti redak). Slika 10.11 prikazuje kako to radi.

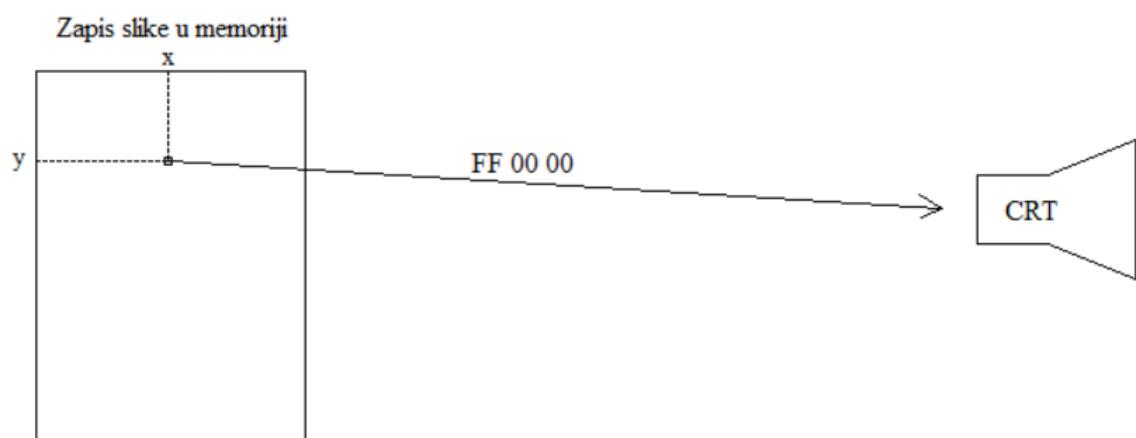
Paleta boja na slici 10.11 definira sljedeće boje:

U navedenom primjeru, na koordinata (x,y) nalazi se boja 1. Prije iscrtavanja na zaslonu, gleda se u paletu boja i pronalazi pod brojem 1 definicija crvene boje. Na zaslonu se crta crvena točka. U ovom primjeru tablica je imala 8 elemenata jer je zadana dubina iznosila 3 bita. Ukoliko odaberemo dubinu od 8 bitova, imat ćemo na raspolaganju 256 mogućih boja. Ova metoda dobra je kada nemamo puno memorije na raspolaganju, a niti ne želimo prikazivati *true-color* slike.

Povećavamo li dubinu do veličine od 24 bita, paleta boja nam se više ne isplati jer paletom boja možemo definirati jednaki iznos boja kao i da direktno definiramo boju. Naime, uz 24 bita po jednom pikselu, bitove možemo grupirati u grupe po 8: 8 bitova za crvenu, 8 bitova za zelenu i 8 bitova za plavu. U tom slučaju više ne koristimo palete, već se za svaki piksel direktno definira boja. Mana ove metode je što zahtjeva veliku količinu memorije za pohranu iole većih slika. Metoda je prikazana na slici 10.12. Sada na lokaciji (x,y) piše kompletan zapis boje: FF 00 00 što odgovara crvenoj boji.

Indeks u paleti boja	Boja
0	crna
1	crvena
2	zelena
3	plava
4	žuta
5	magenta
6	cijan
7	bijela

Tablica 10.2: Primjer palete boja



Slika 10.12: Direktna pohrana boja

Poglavlje 11

Fraktali

11.1 Uvod

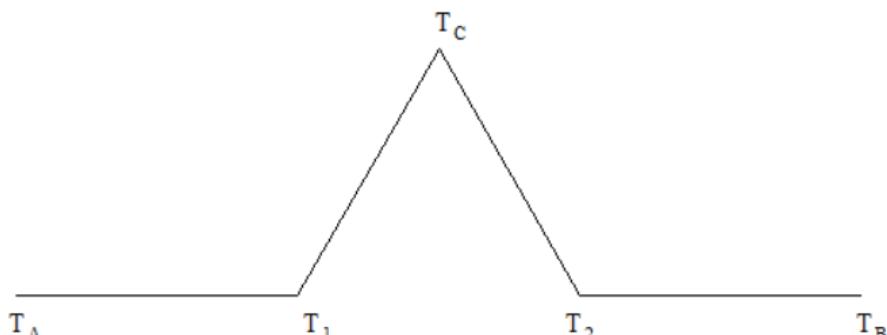
Jedno od ljepših područja računalne grafike pokriva i matematičke umotvorine – fraktale. U nastavku ćemo dati prikaz nekoliko karakterističnih fraktala i pojasniti postupak kako se do njih dolazi. Treba napomenuti da ovdje prikazani fraktali čine vrlo mali dio do sada otkrivenih fraktala, a generalno govoreći, skup fraktala je beskonačan skup tako da ih nikada niti nećemo upoznati sve. Inače, trebamo malo korigirati uvodnu rečenicu: fraktali zapravo i nisu matematičke, već su prirodne tvorevine. Matematičari su samo našli način kao baratati s njima. Pa krenimo od najjednostavnije vrste.

11.2 Samoponavljujući fraktali

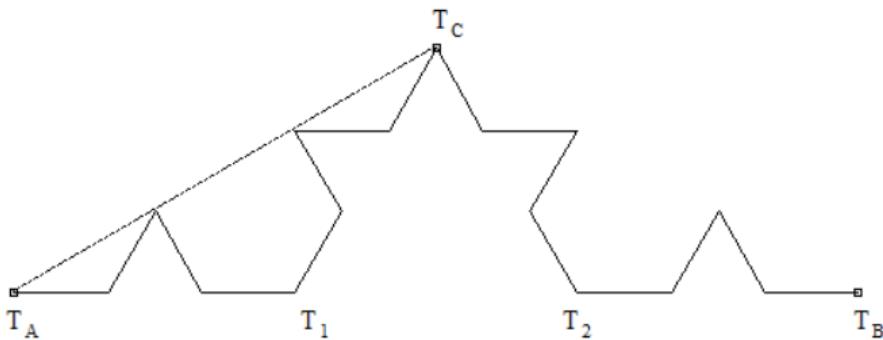
Samoponavljujući fraktali su tvorevine koje na svim skalama umanjenja zadržavaju isti karakteristični oblik. Kao primjer ćemo uzeti *Kochinu krivulju* – fraktal koji je ime dobio prema švedskoj matematičarki Helge von Koch. Kochina krivulja dobije se sljedećim jednostavnim algoritmom.

1. Krenimo od osnovnog oblika prikazanog na slici 11.1.
2. Svaki od četiri segmenta prepravi se tako da se umjesto linije umetne čitav lik iz točke 1. Dobiti će se slika 11.2. Točke T_A i T_C spojene su spojnicom da se pokaže kako novi vrhovi leže na njoj; inače spojnica ne spada u proces generiranja fraktala.
3. Svaki segment nastao u prethodnom koraku opet se zamijeni oblikom iz koraka 1; proces se ponavlja u beskonačnost.

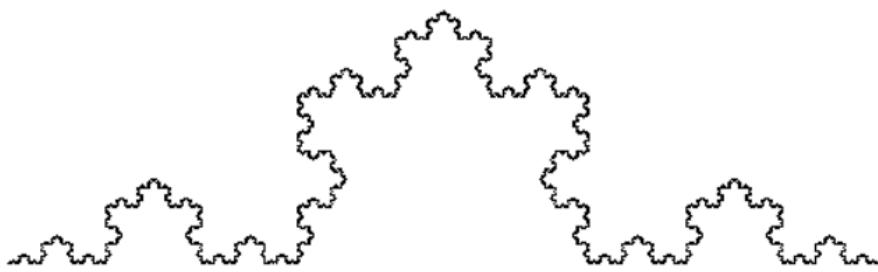
Praktična implementacija gornjeg algoritma neće naravno ići u beskonačnost, već do neke zadane dubine. Npr. ako algoritam pustimo do dubine 6, dobit ćemo sliku 11.3. Kochina krivulja dobije se kada se rekurziju pusti u beskonačnost. Dakako, na računalima obično prikazujemo aproksimaciju



Slika 11.1: Kochina krivulja – početak konstrukcije



Slika 11.2: Kochina krivulja – drugi korak rekurzije



Slika 11.3: Kochina krivulja – rezultat uz dubinu rekurzije 6

Kochine krivulje. Nitko još nije uspio nacrtati sasvim točnu krivulju. Uočimo i neka zanimljiva svojstva Kochine krivulje:

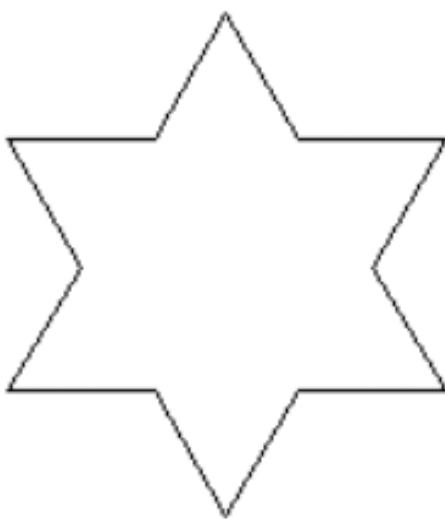
1. duljina Kochine krivulje je beskonačna,
2. duljina bilo kojeg segmenta Kochine krivulje (ma kako mali segment gledali), opet je beskonačna,
3. Kochina krivulja je kontinuirana krivulja,
4. niti u jednoj točki Kochine krivulje derivacija ne postoji.

Funkcija koja iscrtava krivulju prema prethodnom algoritmu može se napisati kao vrlo jednostavna rekurzivna funkcija, prikazana u nastavku.

```

1 void DrawFractal1Rek (T2DRealPoint A, T2DRealPoint B,
2                     T2DRealPoint C, int depth)
3 {
4     T2DRealPoint Ap,Bp,Cp;
5
6     if( depth > 5 ) {
7         MoveTo(fround(A.x),fround(A.y));
8         LineTo(fround(A.x+(B.x-A.x)/3.),
9                 fround(A.y+(B.y-A.y)/3.));
10        LineTo(fround(C.x),fround(C.y));
11        LineTo(fround(A.x+2.*(B.x-A.x)/3.),
12                 fround(A.y+2.*(B.y-A.y)/3.));
13        LineTo(fround(B.x),fround(B.y));
14    }
15
16    depth++;
17
18    Ap.x = A.x; Ap.y = A.y;
19    Bp.x = A.x+(B.x-A.x)/3.; Bp.y = A.y+(B.y-A.y)/3.;
20    Cp.x = A.x+(C.x-A.x)/3.; Cp.y = A.y+(C.y-A.y)/3.;
21
22    DrawFractal1Rek( Ap, Bp, Cp, depth );

```



Slika 11.4: Kochina pahuljica - početak

```

23
24 Ap.x = Bp.x; Ap.y = Bp.y;
25 Bp.x = C.x; Bp.y = C.y;
26 Cp.x = A.x+2.* (C.x-A.x) / 3.; Cp.y = A.y+2.* (C.y-A.y) / 3. ;
27 DrawFractal1Rek( Ap, Bp, Cp, depth );
28
29 Ap.x = Bp.x; Ap.y = Bp.y;
30 Bp.x = A.x+2.* (B.x-A.x) / 3.; Bp.y = A.y+2.* (B.y-A.y) / 3. ;
31 Cp.x = B.x+2.* (C.x-B.x) / 3.; Cp.y = B.y+2.* (C.y-B.y) / 3. ;
32 DrawFractal1Rek( Ap, Bp, Cp, depth );
33
34 Ap.x = Bp.x; Ap.y = Bp.y;
35 Bp.x = B.x; Bp.y = B.y;
36 Cp.x = B.x+(C.x-B.x) / 3.; Cp.y = B.y+(C.y-B.y) / 3. ;
37 DrawFractal1Rek( Ap, Bp, Cp, depth );
38 }

```

Funkcija prima točke A , B i C koje na prethodnim slikama odgovaraju točkama T_A , T_B i T_C , te uz njih i malo elementarne geometrije računa točke T'_A , T'_B i T'_C za svoj svaki segment te izračunate točke predaje u novi rekurzivni poziv. Maksimalna dubina rekurzije određena je prvim ispitivanjem i postavljena je na prvu veću od 5 (dakle 6). Uz ovu funkciju, još je potrebna i nerekurzivna funkcija koja će pozvati svoju rekurzivnu inačicu, a prikazana je u nastavku.

```

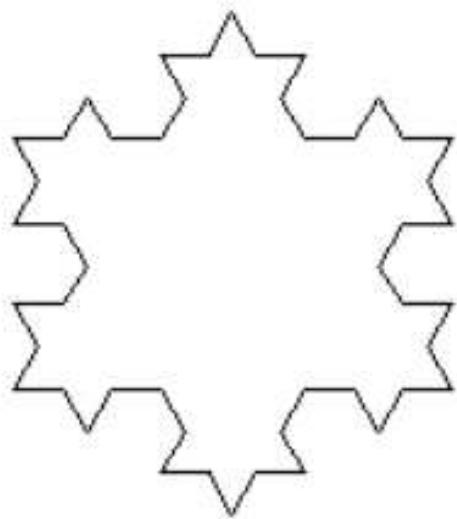
1 void DrawFractal1()
2 {
3     T2DRealPoint A,B,C;
4
5     A.x = 10; A.y = 200 - 10;
6     B.x = 320 - 10; B.y = 200 - 10;
7     C.x = ( A.x + B.x ) / 2. ;
8     C.y = A.y - sqrt(3)/2.* ( B.x - A.x ) / 3. ;
9     DrawFractal1Rek( A, B, C, 1 );
10 }

```

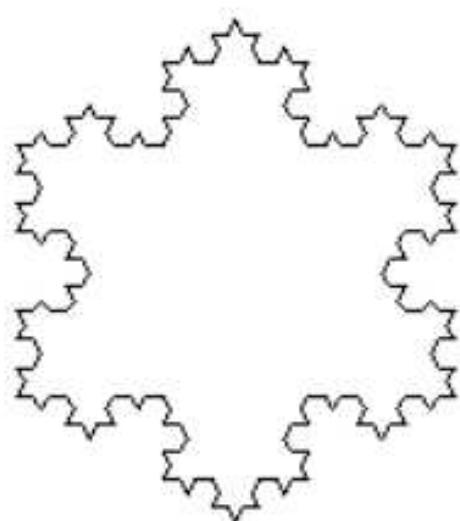
Funkcija pretpostavlja zaslon razlučivosti 320×200 i ostavlja po 10 praznih pixela lijevo, desno i ispod krivulje.

Jednostavnim proširenjem gornjeg algoritma dobiti ćemo poznatu *Kochinu pahuljicu* (engl. *Koch's Snowflake*). Umjesto od jedne linije krenut ćemo od trokuta čije su stranice segmenti prikazani u prethodnom algoritmu pod brojem 1. Dobiti ćemo lik prikazan na slici 11.4.

Sada svaku liniju iterativno zamijenimo tim segmentima. Razvoj slike fraktala prikazan je na slikama 11.5 i 11.6.



Slika 11.5: Kochina pahuljica - drugi korak rekurzije



Slika 11.6: Kochina pahuljica - kraj

Iz opisanog postupka može se vidjeti da se Kochina pahuljica može dobiti kao jednakostranični trokut čije su stranice Kochine krivulje. Matematički gledano, ovaj fraktal zanimljiv je po još nečemu: duljina njegova opsega je beskonačna, iako je površina koju lik zauzima konačna. Algoritam za crtanje pahuljice također se sastoјi od dva dijela: jedne rekurzivne funkcije koja je već navedena kod Kochine krivulje (DrawFractal1Rek), i nerekurzivne funkcije koja poziva istu, koja je prikazana u nastavku.

```

1 void DrawFractalSnowflake()
2 {
3     T2DRealPoint A,B,C;
4     double visina ,d;
5
6     visina = 200 - 20;
7     d = sqrt(3.)*visina /2.;
8
9     A.x = 10; A.y = 200 - 10 - visina /4.;
10    B.x = A.x + d; B.y = A.y;
11    C.x = ( A.x + B.x )/2.;
12    C.y = A.y + sqrt(3)/2.*d /3.;
13    DrawFractal1Rek(A, B, C, 1);
14
15    B.x = A.x + d /2.;
16    B.y = A.y - d*sqrt(3.) /2.;
17    C.x = (A.x+B.x)/2.-visina/4.*sqrt(3) /2.;
18    C.y = (A.y+B.y)/2.-visina/4./2.;
19    DrawFractal1Rek(A, B, C, 1);
20
21    A.x = A.x + d;
22    C.x = (A.x+B.x)/2.+visina/4.*sqrt(3) /2.;
23    C.y = (A.y+B.y)/2.-visina/4./2.;
24    DrawFractal1Rek(A, B, C, 1);
25 }
```

I ova funkcija podrazumijeva ekran 320×200 pixela. Funkcija se sastoјi od tri cjeline: poziva za iscrtavanje donje Kochine krivulje, poziva za iscrtavanje lijeve Kochine krivulje te poziva za iscrtavanje desne Kochine krivulje.

Spomenimo još i zanimljiva svojstva Kochine pahuljice:

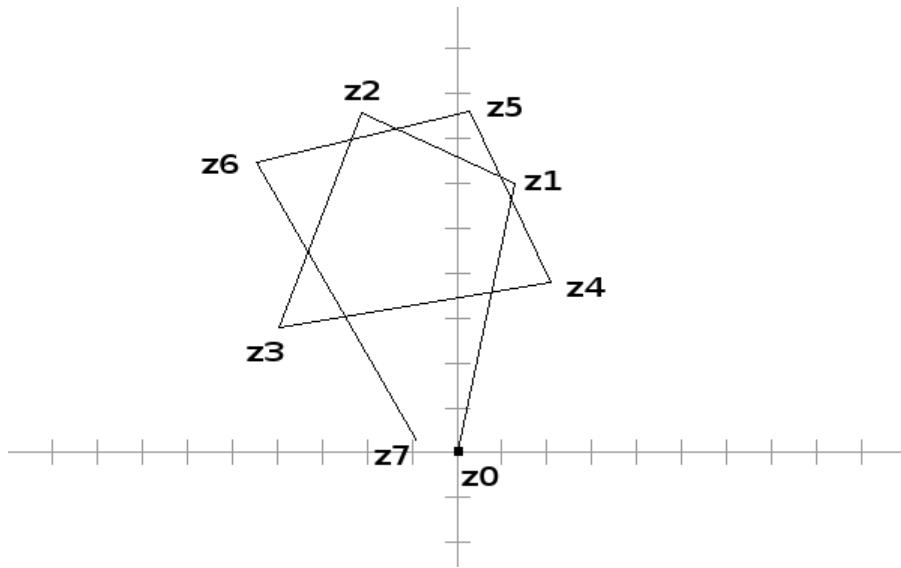
1. opseg Kochine pahuljice je beskonačan,
2. površina Kochine pahuljice je konačna.

11.3 Mandelbrotov fraktal

Kako bismo opisali postupak nastanka Mandelbrotovog fraktala, morat ćemo se najprije upoznati s temeljnijim pojmom – Mandelbrotovim skupom. Mandelbrotov skup je skup točaka u kompleksnoj ravnini, koji zadovoljava određeni uvjet s kojim ćemo se upoznati u nastavku, i čiju granicu nazivamo Mandelbrotovim fraktalom (fraktal je ime dobio prema matematičaru Benoîtu Mandelbrotu). Pa kada za neku točku c iz kompleksne ravnine kažemo da pripada Mandelbrotovom skupu? Da bismo na to odgovorili, definirajmo naprije kompleksnu rekurzivnu funkciju z_{n+1} :

$$z_{n+1} = z_n^2 + c \quad \text{uzpoetniuvjet } z_0 = 0 \quad (11.1)$$

Promotrimo niz kompleksnih brojeva koje ova rekurzija generira za jedan konkretan kompleksni broj c . Primjerice, ako uzmemo $c = 0.13 + 0.6i$, dobit ćemo sljedeći niz kompleksnih brojeva: $z_0 = 0$, $z_1 = 0.13 + 0.6i$, $z_2 = -0.2131 + 0.756i$, $z_3 = -0.3961 + 0.2778i$, $z_4 = 0.2097 + 0.3799i$, $z_5 = 0.0297 + 0.7594i$, $z_6 = -0.4458 + 0.6450i$, $z_7 = -0.0874 + 0.0249$, itd. Brojevi su prikazani na slici 11.7, pri čemu je njihova trajektorija naznačena linijama. Ono što nas zanima jest je li modul ovih kompleksnih brojeva koje generira promatrana rekurzivna funkcija ograničen, ako pustimo da n teži u beskonačnost. Riječ *ograničen* u ovom kontekstu znači da je moguće pronaći konačan broj ϵ takav da je $\forall n. z_n < \epsilon$. U geometrijskom smislu, zanima nas ako oko ishodišta kompleksne ravnine (točke $0 + 0i$)



Slika 11.7: Ispitivanje pripadnosti točke Mandelbrotovom skupu

nacrtamo kružnicu radijusa ϵ , hoće li svi kompleksni brojevi koje generira ova rekurzivna jednadžba ostati unutar te kružnice. Ako je odgovor *da*, tada kompleksni broj c za koji smo radili ispitivanje pripada Mandelbrotovom skupu. Ako je odgovor *ne*, odnosno ako niz divergira, kompleksni broj c ne pripada Mandelbrotovom skupu.

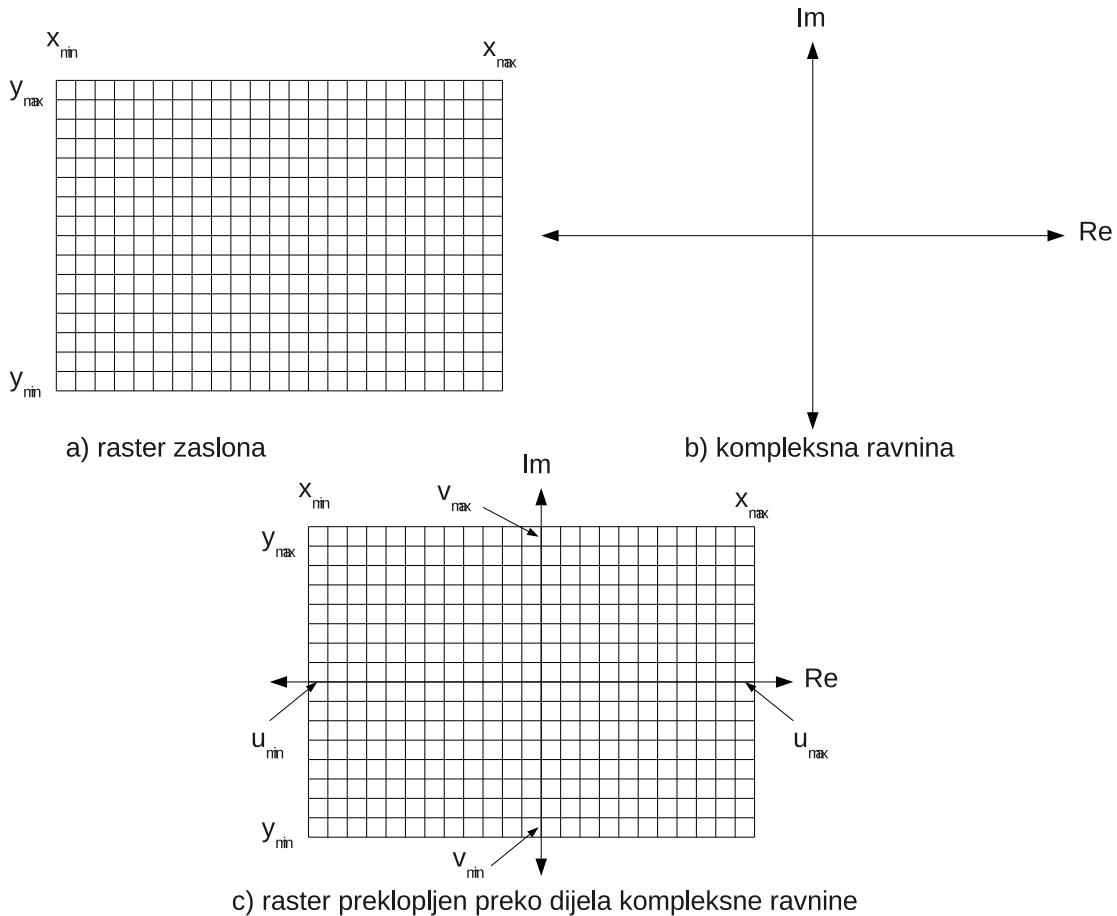
Pitanje na koje još treba odgovoriti jest koliki treba biti ϵ ? Prema modernim spoznajama, za sve točke c Mandelbrotovog skupa, elementi pripadnog niza z_n nikada ne izlaze iz kruga radijus 2 povučenog oko ishodišta kompleksne ravnine. Ako se dogodi da u bilo kojem trenutku modul elementa niza z_n postane veći od 2, niz će divergirati, i pripadna točka c ne pripada Mandelbrotovom skupu.

Programski isječak koji ilustrira ispitivanje divergencije prikazan je u nastavku. Prilikom implementacije provjere divergira li niz očitno nije moguće raditi provjeru do beskonačnosti. Stoga metoda `divergence_test` kao drugi parametar prima dozvoljenu dubinu rekurzije. Ako se generira i ispita limit točaka, i njihov modul je i dalje unutar ϵ kružnice, predana točka c proglašit će se točkom koja pripada Mandelbrotovom skupu (metoda će vratiti -1). Ako se u nekom koraku utvrđi divergencija, metoda kao povratnu vrijednost vraća korak u kojem se je to utvrdilo. Metoda kao radijus koristi $\epsilon = 2$, odnosno ispituje je li kvadrat modula veći od $2 \cdot 2 = 4$, kako bi se izbjeglo vađenje korijena.

```

1 typedef struct {
2     double re;
3     double im;
4 } complex;
5
6 int divergence_test(complex c, int limit) {
7     complex z;
8     z.re = 0; z.im = 0;
9     for (int i = 1; i <= limit; i++) {
10         double next_re = z.re*z.re - z.im*z.im + c.re;
11         double next_im = 2*z.re*z.im + c.im;
12         z.re = next_re;
13         z.im = next_im;
14         double modul2 = z.re*z.re + z.im*z.im;
15         if(modul2 > 4) return i;
16     }
17     return -1;
18 }
```

No kako nacrtati Mandelbrotov fraktal? Uočimo da Mandelbrotov skup, baš kao i Mandelbrotov fraktal (granica tog skupa) imaju beskonačno točaka. S druge pak strane, ekran zaslona omogućava nam isključivo prikaz rastera - konačnog diskretnog niza slikovnih elemenata. Stoga ćemo za potrebe crtanja napraviti uzorkovanje točaka kompleksne ravnine, i samo ćemo za njih provjeriti pripadaju li



Slika 11.8: Crtanje Mandelbrotovog skupa

one Mandelbrotovom skupu ili ne. To ćemo napraviti tako da ćemo raster zaslona preklopiti preko pravokutnog područja u kompleksnoj ravnini, i potom za svaki slikovni element definiran ekranским koordinatama (x, y) izračunati u koju se on točku kompleksne ravnine c preslikava (slika 11.8). Pri tome lijevi rub ekrana poravnavamo s vrijednosti u_{min} na realnoj osi, desni rub ekrana poravnavamo s vrijednosti u_{max} na realnoj osi, donji rub ekrana poravnavamo s vrijednosti v_{min} na imaginarnoj osi, te gornji rub ekrana poravnavamo s vrijednosti v_{max} na imaginarnoj osi. Točke kompleksne ravnine čiji je realni dio manji od u_{min} ili veći od u_{max} , ili čiji je imaginarni dio manji od v_{min} ili veći od v_{max} uopće nećemo ispitivati.

Dio kompleksne ravnine koji je preklopljen s rasterom uzorkovat ćemo u skladu s gustoćom rastara. Promotrimo neku točku ekrana (x, y) . Neka se ona u preslikava u kompleksnoj ravnini u točku (u, v) . Uočimo da vrijedi:

$$\frac{x - x_{min}}{x_{max} - x_{min}} = \frac{u - u_{min}}{u_{max} - u_{min}},$$

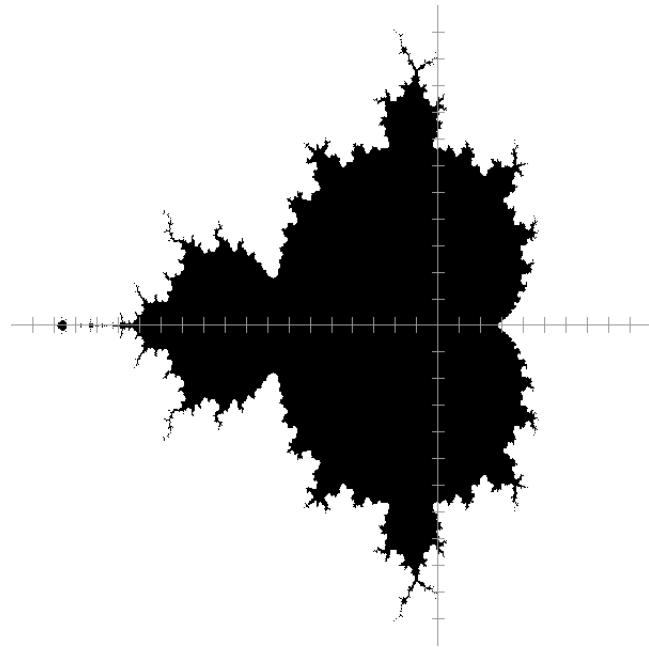
$$\frac{y - y_{min}}{y_{max} - y_{min}} = \frac{v - v_{min}}{v_{max} - v_{min}}.$$

Temeljem tih izraza za zadatu ekransku točku (x, y) možemo očitati koordinate pripadne točke kompleksne ravnine (u, v) :

$$u = \frac{x - x_{min}}{x_{max} - x_{min}} \cdot (u_{max} - u_{min}) + u_{min} \quad (11.2)$$

$$v = \frac{y - y_{min}}{y_{max} - y_{min}} \cdot (v_{max} - v_{min}) + v_{min} \quad (11.3)$$

Crtanje Mandelbrotovog frakta sada se svodi na ugniježđene **for**-petlje: jedna koja ide po retcima i druga koja ide po stupcima. Za svaki slikovni element ispita se pripada li Mandelbrotovom skupu, i



Slika 11.9: Mandelbrotov skup

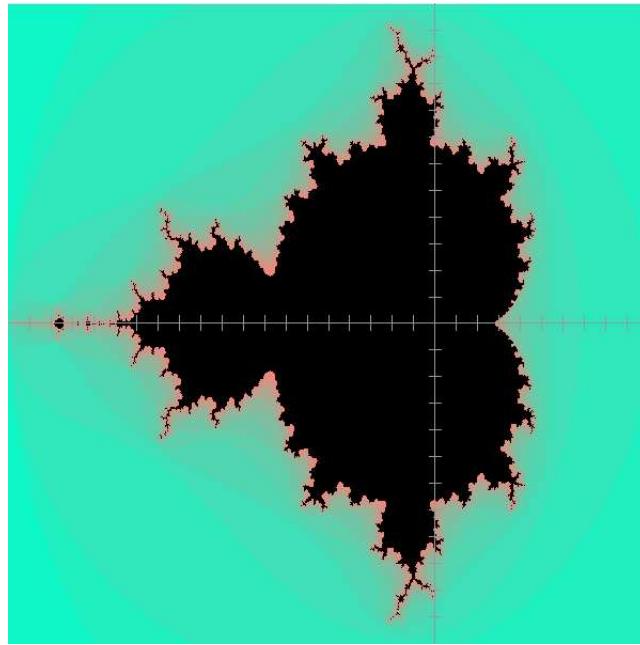
ako pripada, točka se nacrtava crnom bojom, a ako ne pripada, nacrtava se bijelom bojom. Algoritam je prikazan u nastavku. Slika 11.9 prikazuje rezultat za parametre $u_{min} = -2$, $u_{max} = 1$, $v_{min} = -1.2$, $v_{max} = -1.2$ (koordinatni sustav dodan je kako bi se vidjela pozicija točaka u kompleksnoj ravnini).

```

1  double xmin = 0;
2  double xmax = 599;
3  double ymin = 0;
4  double ymax = 599;
5  double umin = -2;
6  double umax = 1;
7  double vmin = -1.2;
8  double vmax = 1.2;
9
10 void renderScene() {
11     glPointSize(1);
12     glBegin(GL_POINTS);
13     for(int y = ymin; y <= ymax; y++) {
14         for(int x = xmin; x <= xmax; x++) {
15             complex c;
16             c.re = (x-xmin)/(double)(xmax-xmin)*(umax-umin)+umin;
17             c.im = (y-ymin)/(double)(ymax-ymin)*(vmax-vmin)+vmin;
18             int n = divergence_test(c, 16);
19             if(n == -1) {
20                 glColor3f(0.0f, 0.0f, 0.0f);
21             } else {
22                 glColor3f(1.0f, 1.0f, 1.0f);
23             }
24             glVertex2i(x, y);
25         }
26     }
27 }
28 glEnd();
29 }
```

11.3.1 Bojanje Mandelbrotovog fraktala

Kada govorimo o Mandelbrotovom fraktalu, obično u glavi imamo viziju slike žarkih boja – međutim, naš prvi pokušaj rezultirao je crno bijelom slikom. Takva slika nastala je stoga što smo gledali pripada



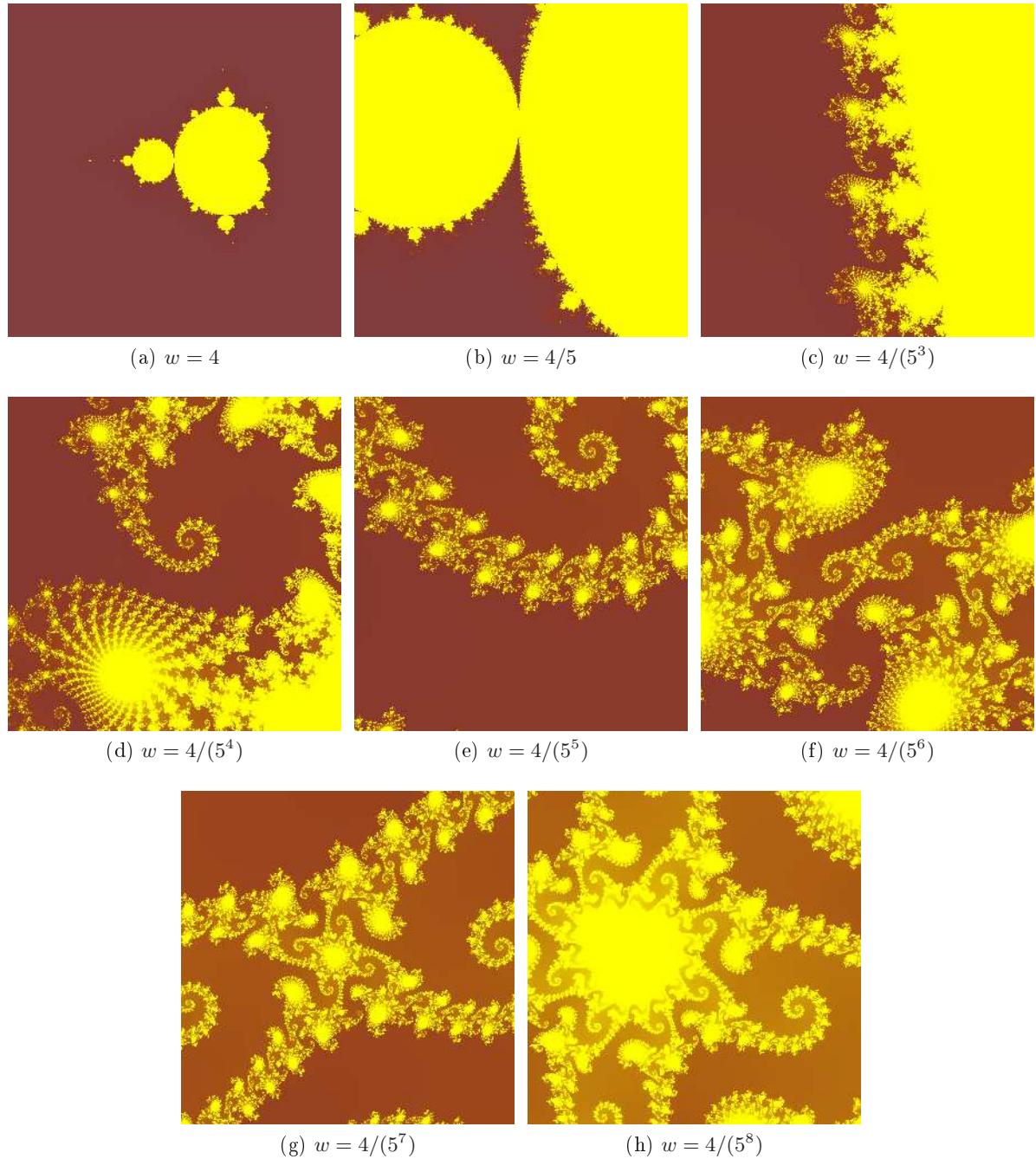
Slika 11.10: Mandelbrotov skup uz bojanje prema brzini divergencije

li promatrana točka Mandelbrotovom skupu ili ne. Potrebna je jednostavna modifikacija kako bismo uveli malo više boja u sliku: umjesto da gledamo čistu pripadnost točke Mandelbrotovom skupu, idemo iskoristiti informaciju koju nam vraća metoda divergence_test – brzinu divergencije. Prisjetimo se, ta nam je metoda vraćala -1 ako divergencija nije utvrđena, a ako je, vraćala nam je korak u kojem je ona utvrđena. Iskoristimo to tako da temeljem brzine divergencije točke odaberemo prikladnu boju. Modifikacija je prikazana u programskom isječku u nastavku, a rezultat prikazuje slika 11.10.

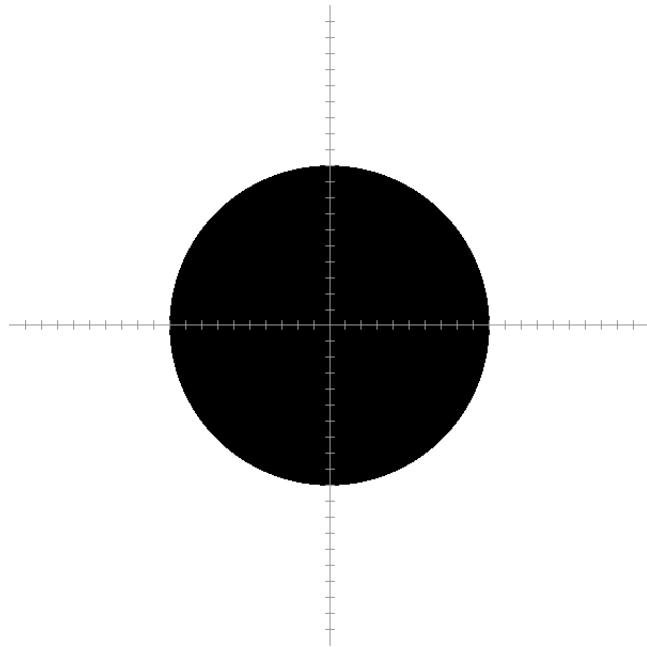
```

1 void renderScene() {
2     glPointSize(1);
3     glBegin(GL_POINTS);
4     for (int y = ymin; y <= ymax; y++) {
5         for (int x = xmin; x <= xmax; x++) {
6             complex c;
7             c.re = (x-xmin)/(double)(xmax-xmin)*(umax-umin)+umin;
8             c.im = (y-ymin)/(double)(ymax-ymin)*(vmax-vmin)+vmin;
9             int n = divergence_test(c, 16);
10            if (n== -1) {
11                glColor3f(0.0f, 0.0f, 0.0f);
12            } else {
13                glColor3f((double)n/limit,
14                          1.0-(double)n/limit/2.0,
15                          0.8-(double)n/limit/3.0);
16            }
17            glVertex2i(x, y);
18        }
19    }
20    glEnd();
21 }
```

Umjesto da Mandelbrotov fraktal gledamo na područje od popriliči ± 2 , zanimljive slike dobit ćemo i ako promatramo neko uže područje (što zapravo odgovara uvećavanju tog dijela slike i prikazivanju na zaslonu). Slika 11.11 sadrži niz pogleda na različite dijelove Mandelbrotovog fraktala, koji su dobiveni promatranjem dijela kompleksne ravnine koji odgovara pravokutnom području ukupne širine i visine w , čiji je centar točka $(C_x, C_y) = (-0.7454265, 0.1130090)$. Uz ovu konvenciju, vrijedi: $u_{min} = C_x - w/2$, $u_{max} = C_x + w/2$, $v_{min} = C_y - w/2$, $v_{max} = C_y + w/2$.



Slika 11.11: Mandelbrotov fraktal uz različita uvećanja



Slika 11.12: Julijev skup za funkciju $z_{n+1} = z_n^2$

11.4 Julijev fraktal i Julijeva krivulja

Da bismo korektno definirali Julijev skup, morali bismo se pozabaviti teorijom dinamičkih sustava i pojmovima poput atraktora, orbita i sl. Umjesto toga, ovdje ćemo pokušati definirati definirati Julijev skup jednostavnije (i možda ne baš skroz matematički korektno). Pogledajmo jedan jednostavan primjer kompleksne funkcije kompleksne varijable:

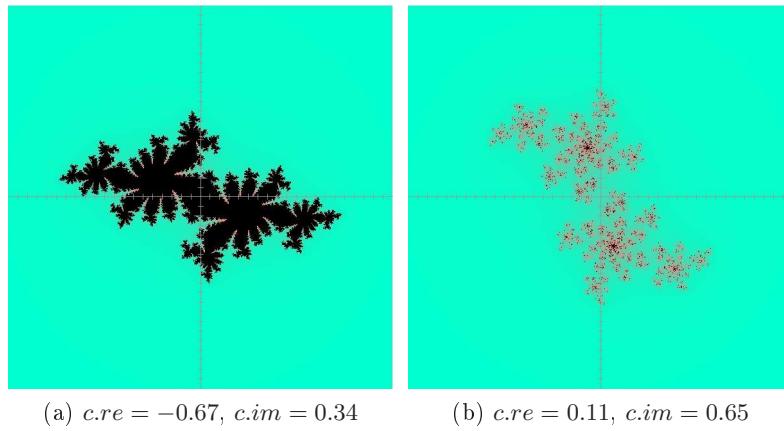
$$f(z) = z^2$$

Zanima nas slična stvar kao i kod Mandelbrotovog skupa: ako odaberemo neki početni kompleksni broj z_0 , izračunamo vrijednost funkcije $z_1 = f(z_0)$, potom tu vrijednost uzmemo kao argument i ponovo izračunamo vrijednost funkcije $z_2 = f(z_1) = f(f(z_0))$, i tako postupak nastavimo u beskonačnost, što će se dogoditi s nizom brojeva koje dobivamo? U slučaju promatrane funkcije, lako je pokazati da su moguća tri scenarija. Ako je modul od z_0 bi manji od 1, niz će konvergirati u točku $0 + 0i$ (to je jedan atraktor). Ako je modul od z_0 bio veći od 1, niz će divergirati u beskonačnost. Konačno, ako je modul od z_0 bio točno jednak 1, niz će se sastojati od brojeva koji i sami imaju modul jednak 1. Ovo ispitivanje možemo napraviti za sve brojeve kompleksne ravnine. Prema rezultatu tog ispitivanja, sve brojeve kompleksne ravnine $z \in \mathbb{C}$ podijelit ćemo u dva skupa. Sve točke $z \in \mathbb{C}$ za koje prethodni niz divergira zvat ćemo *escape set*. Sve točke koje konvergiraju prema atraktoru (u našem slučaju prema ishodištu) tvore *trapping set*. *Granica ovih dvaju skupova čini Julijev skup*, i za promatranoj funkciji prikazana je na slici 11.12. Crnom bojom prikazan je *trapping set*, bijelom *escape set* – granica (točke na jediničnoj kružnici) čini Julijev skup, i uočimo da je to u ovom slučaju jednostavna krivulja – kružnica (nije fraktal). Francuski matematičar Gaston Julia zaslужan je za otkriće Julijeva skupa, koji je usko vezan sa skupom koji je otkrio također francuski matematičar Pierre Fatou – Fatouovim skupom, koji čini komplement Julijevog skupa.

Uočimo da promatranoj funkciji možemo i poopćiti. Kada se danas govori o Julijevom skupu, uobičajeno se misli barem na funkciju oblika:

$$f(z) = z^2 + c$$

gdje su $z, c \in \mathbb{C}$ (iako je moguće koristiti i druge funkcije). Uočimo da je ovo isti izraz koji smo promatrali i kod Mandelbrotovog fraktaala. Za Mandelbrotov skup točka c bila je funkcija promatrane točke ekrana (nju smo mijenjali) dok je z_0 bio 0. Kod Julijevog skupa parametar c se fiksira na početku, a svaka točka ekrana (x, y) odredit će zasebnu početnu točku $z_0 = (u, v)$. Uzmemo li kao $c = -2$,



Slika 11.13: Povezan i nepovezan Julijev skup

Julijev skup bit će skup točaka koje leže na realnoj osi i protežu se od -2 do 2 (točke čine segment linije). Interesantno, za sve točke koje se ne nalaze na tom segmentu, iterativno preslikavanje će divergirati u beskonačnost. U ovom slučaju Julijev skup opet čini krivulju i to krivulju koja nije fraktal.

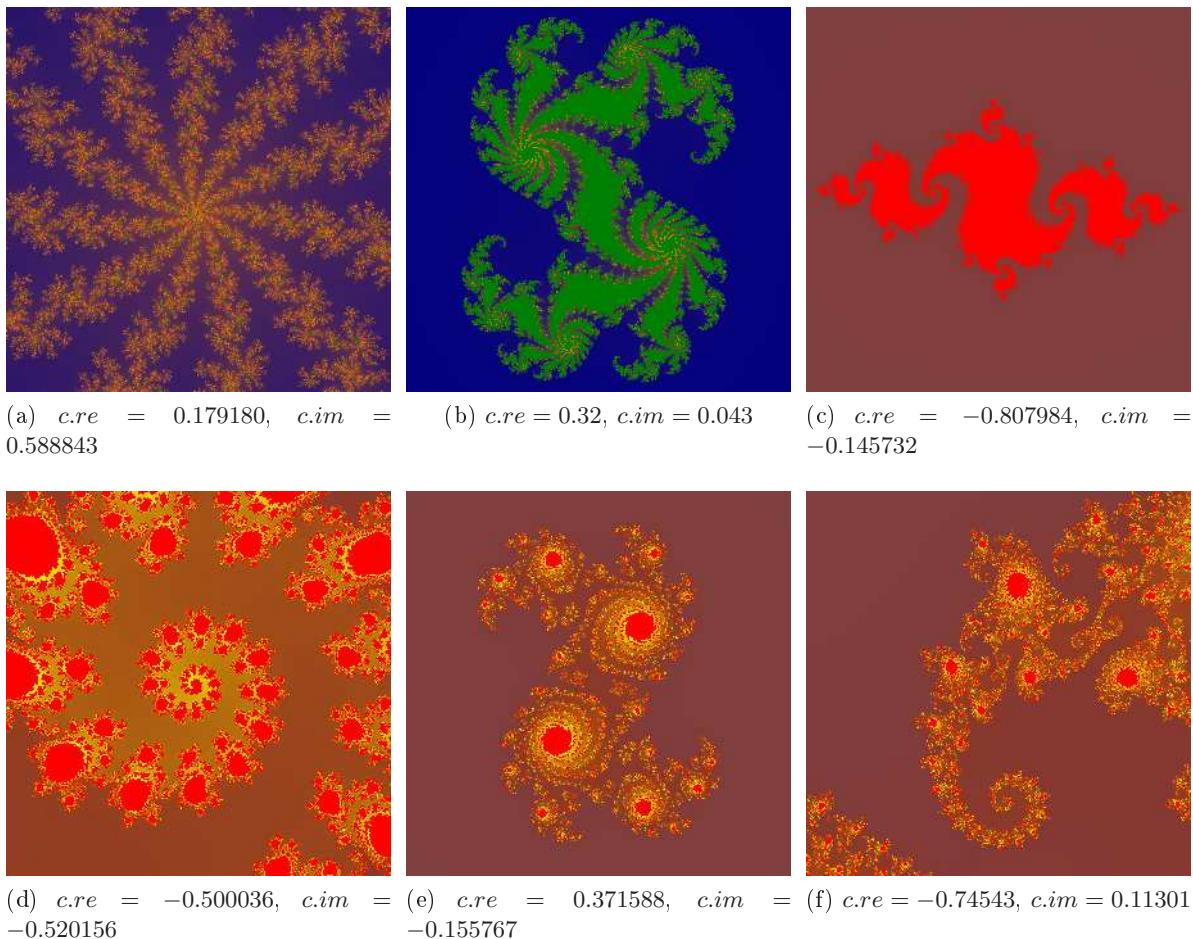
Konačno, za različite druge vrijednosti $c \in \mathbb{C}$ dobit ćemo različite Julijeve skupove. Svi Julijevi skupovi mogu se podijeliti u dvije grupe: *povezani* i *nepovezani* (slika 11.13). Povezani Julijev skup nastaje kada kao vrijednost parametra c odaberemo vrijednost koja je u Mandelbrotovom skupu. U tom slučaju, od svake točke koja pripada Julijevu skupu do svake druge točke koja pripada Julijevu skupu moguće je doći direktno prateći točke koje su također u Julijevu skupu (otuda pojam *povezan* skup). Ako kao c odaberemo vrijednost koja nije u Mandelbrotovom skupu, rezultat će biti nepovezan Julijev skup. Za takav skup kažemo da predstavlja *prašinu* (engl. *dust*) – to je skup sastavljen od beskonačnog broja *izoliranih* točaka koje su kondenzirane u grupu. Taj je skup također beskonačno gust: za svaki konačno mali radijus oko svake točke Julijevog skupa postoji još barem jedna točka koja također pripada Julijevom skupu.

Kod koji uvažava ove razlike prikazan je u nastavku, a nekoliko dodatnih primjera slika zajedno s korištenim parametrom c prikazano je na slici 11.14.

```

1 int divergence_test(complex z0, complex c, int limit, int epsilonSquare) {
2     complex z;
3     z.re = z0.re; z.im = z0.im;
4     double modul2 = z.re*z.re + z.im*z.im;
5     if(modul2 > epsilonSquare) return 0;
6     for(int i = 1; i <= limit; i++) {
7         double next_re = z.re*z.re - z.im*z.im + c.re;
8         double next_im = 2*z.re*z.im + c.im;
9         z.re = next_re;
10        z.im = next_im;
11        double modul2 = z.re*z.re + z.im*z.im;
12        if(modul2 > epsilonSquare) return i;
13    }
14    return -1;
15 }
16
17 int max(int a, int b) {
18     return (b>a) ? b : a;
19 }
20
21 void renderScene() {
22     int limit = 64;
23     complex c;
24     // Ovdje je odabran parametar 'c':
25     c.re = 0.11; c.im = 0.65;
26     double epsilon = max(c.re*c.re + c.im*c.im, 2.0);
27     double epsilonSquare = epsilon * epsilon;
28     glPointSize(1);
29     glBegin(GL_POINTS);

```



Slika 11.14: Primjeri Julijevog fraktala

```

30    for (int y = ymin; y <= ymax; y++) {
31        for (int x = xmin; x <= xmax; x++) {
32            complex z0;
33            z0.re = (x-xmin)/(double)(xmax-xmin)*(umax-umin)+umin;
34            z0.im = (y-ymin)/(double)(ymax-ymin)*(vmax-vmin)+vmin;
35            int n = divergence_test(z0, c, limit, epsilonSquare);
36            if (n== -1) {
37                glColor3f(0.0f, 0.0f, 0.0f);
38            } else {
39                glColor3f((double)n/limit,
40                          1-(double)n/limit/2.0,
41                          0.8-(double)n/limit/3.0);
42            }
43            glVertex2i(x, y);
44        }
45    }
46    glEnd();
47 }
```

11.5 IFS fraktali

IFS u imenu ove vrste fraktala dolazi od engleskog naziva *Iterated Function System*, što opisuje postupak kojim se dolazi do fraktala. Ideja je sljedeća. Neka je dana afina transformacija:

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \quad (11.4)$$

Ova transformacija točku $T_i = (x_i, y_i)$ preslikava u točku $T_{i+1} = (x_{i+1}, y_{i+1})$, pri čemu matričnu

a	b	c	d	e	f	p_i
0.00	0.00	0.00	0.16	0.00	0.00	0.01
0.85	0.04	-0.04	0.85	0.00	1.60	0.85
0.20	-0.26	0.23	0.22	0.00	1.60	0.07
-0.15	0.28	0.26	0.24	0.00	0.44	0.07

Tablica 11.1: IFS za primjer vrste paprati

jednadžbu (11.4) možemo raspisati po komponentama kako slijedi.

$$x_{i+1} = a \cdot x_i + b \cdot y_i + e \quad (11.5)$$

$$y_{i+1} = c \cdot x_i + d \cdot y_i + f \quad (11.6)$$

Fraktal ćemo dobiti tako da definiramo n različitih transformacija i za svaku transformaciju definiramo vjerojatnost p_i da će ta transformacija biti primijenjena (mora vrijediti $\sum_{i=1}^n p_i = 1$). Ovakav sustav transformacija tipično ćemo prikazivati u tabličnom obliku, gdje će retci tablice biti transformacije, a stupci tablice koeficijenti odnosno pripadna vjerojatnost. Primjer takvog zapisa prikazan je u tablici 11.1. IFS se u prikazanom primjeru sastoji od 4 funkcije, pri čemu je vjerojatnost primjene prve jednaka 1%, vjerojatnost primjene druge jednaka je 85%, te su vjerojatnosti primjene treće i četvrte jednaka 7% za svaku.

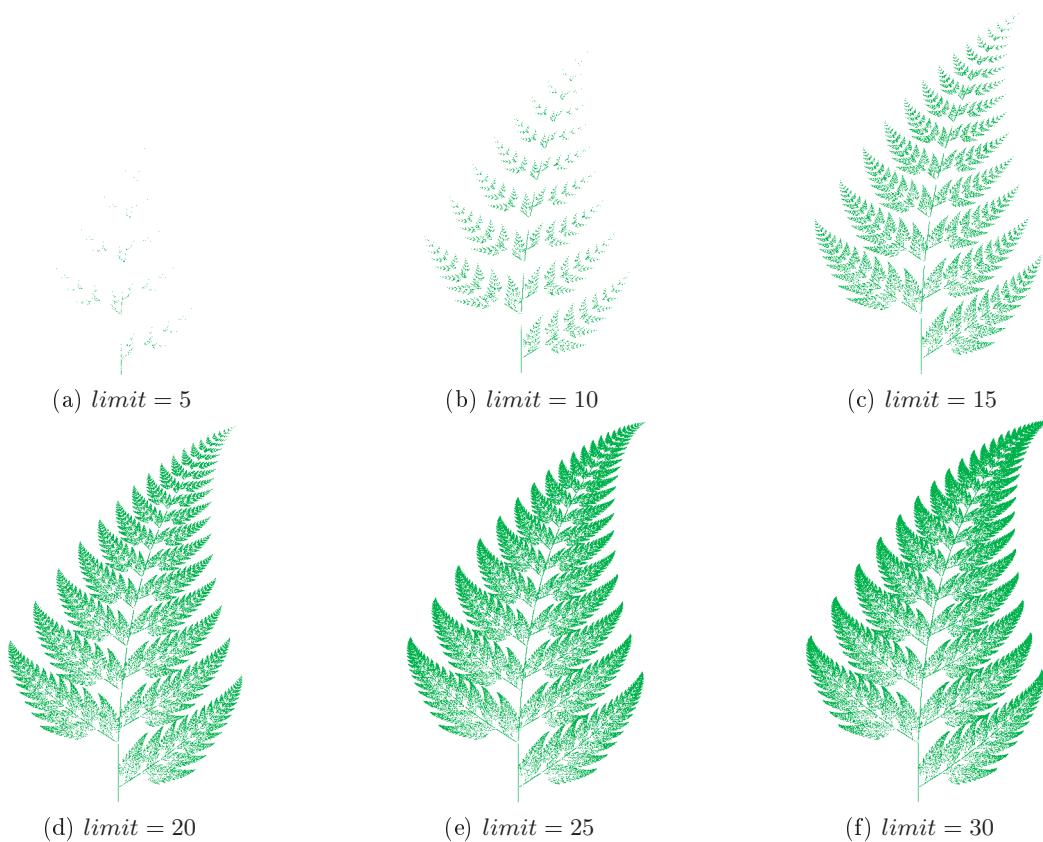
11.5.1 Kako crtamo IFS fraktal

IFS fraktal nastaje kao rezultat iterativne primjene definiranih transformacija na neku početnu točku. Pri tome se u svakom koraku posredstvom slučajnog mehanizma odabere koja će transformacija biti korištena u tom koraku (dakako, proporcionalno vjerojatnostima primjene definiranima uz svaku transformaciju). Postupak se ponovi određen broj puta (do zadane dubine), i potom se nacrtava samo konačni piksel. Čitav postupak potom se ponavlja veliki broj puta. Izvorni kod programa koji crta fraktal u skladu s IFS-om definiranim u tablici 11.1 prikazan je u nastavku.

```

1 int zaokruzi(double d) {
2     if(d>=0) return (int)(d+0.5);
3     return (int)(d-0.5);
4 }
5
6 void renderScene() {
7     int limit = 25;
8     glPointSize(1);
9     glColor3f(0.0f, 0.7f, 0.3f);
10    glBegin(GL_POINTS);
11    double x0, y0;
12    for(int brojac = 0; brojac < 200000; brojac++) {
13        // pocetna tocka:
14        x0 = 0;
15        y0 = 0;
16        // iterativna primjena:
17        for(int iter = 0; iter < limit; iter++) {
18            double x,y;
19            int p = rand() % 100;
20            if(p<1) {
21                x = 0;
22                y = 0.16*y0;
23            } else if(p<8) {
24                x = 0.2*x0-0.26*y0+0;
25                y = 0.23*x0+0.22*y0+1.6;
26            } else if(p<15) {
27                x = -0.15*x0+0.28*y0+0;
28                y = 0.26*x0+0.24*y0+0.44;
29            } else {

```



Slika 11.15: Primjeri lista paprati generiranog ITS-om

	a	b	c	d	e	f	p_i
30	0.5	0.0	0.0	0.5	0.0	0.0	$1/3$
31	0.5	0.0	0.0	0.5	1.28	0.0	$1/3$
32	0.5	0.0	0.0	0.5	0.64	0.8	$1/3$

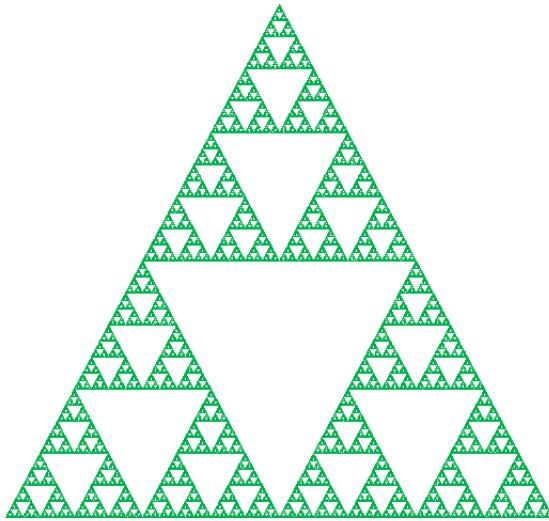
Tablica 11.2: IFS za trokut Sierpinskog

```

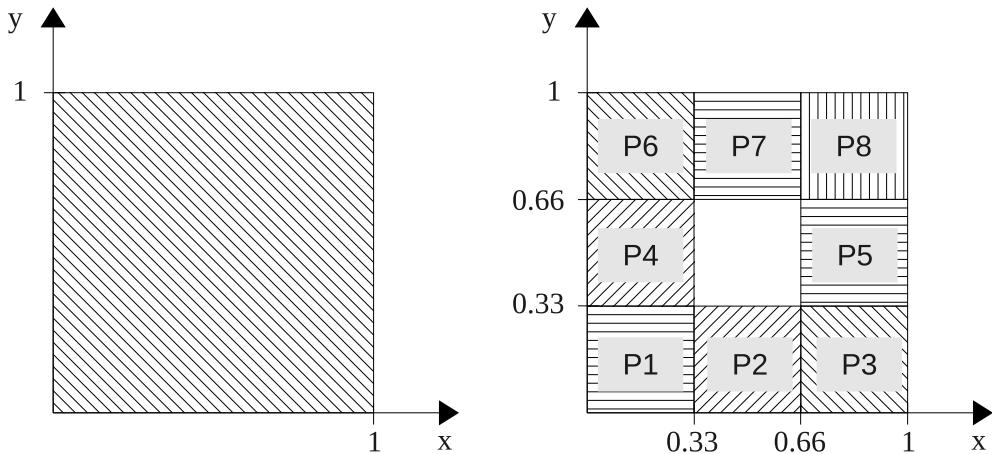
30      x = 0.85*x0+0.04*y0+0;
31      y = -0.04*x0+0.85*y0+1.6;
32  }
33  x0 = x; y0 = y;
34  }
35 // crtanje konacne tocke
36 glVertex2i( zaokruzi(x0*80+300), zaokruzi(y0*60));
37
38 glEnd();
39 }
```

Iterativna primjena transformacija ovog IFS-a daje x -koordinate iz raspona popriliči ± 2.5 , te y -koordinate iz raspona od 0 do popriliči 10. Kako je prikazani izvorni kod korišten za crtanje na prozoru dimenzija 600×600 , konačna točka prije crtanja dodatno se transformira tako da se x skalira s 80 (čime mu raspon postaje ± 200) i zatim translatira za 300 piksela (prema centru prozora). y -koordinata samo se skalira sa 60, čime joj raspon postaje od 0 do popriliči 600. Primjena ovog algoritma uz različite vrijednosti parametra $limit$ prikazana je na slici 11.15. Taj fraktal poznat je pod nazivom *Barnsley-eva paprat*.

Različitim IFS-ovima moguće je dobiti niz različitih fraktala. Primjerice, tablica 11.2 prikazuje transformacije koje generiraju poznati *trokut Sierpinskog*, koji je potom nacrtan prilagodbom prethodnog programa, i prikazan je na slici 11.16. Spomenimo samo da je kao konačna transformacija prije crtanja piksela korišteno $x \leftarrow x \cdot 200 + 50$, $y \leftarrow y \cdot 300$.



Slika 11.16: Trokut Sierpinskog



Slika 11.17: Konstrukcija ITS fraktala

11.5.2 Konstrukcija IFS fraktala

U ovoj sekciji pokušat ćemo približiti način na koji konstruiramo IFS fraktale, ali bez ulazeњa u detaljna matematička razmatranja koja su podloga čitavog procesa. Proses je ilustriran na slici 11.17. Prisjetimo se najprije – ovdje govorimo o fraktalima koji su samoslični; uvećamo li neki dio fraktala, opet ćemo uočiti jednaku građu. Imajući to u vidu, zamislimo da je čitav fraktal obuhvaćen jediničnim pravokutnikom (pravokutnik koji je smješten u ishodište, i ima visinu i širinu jednaku 1).

Građu fraktala obuhvaćenog tim pravokutnikom želimo prikazati pomoću više manjih pravokutnika – od kojih će svaki ponovno biti umanjena kopija tog istog fraktala. U ovom trenutku važno je samo zapamtiti da ti pravokutnici moraju biti manji od polaznog pravokutnika, da bi proces konvergirao. Primjerice, odlučili smo se da će fraktal imati imati 8 umanjenih kopija, koje su na slici prikazane kao pravokutnici P_1 do P_8 . Središnji dio bit će prazan. Naš je zadatak sada pronaći affine transformacije w_i koje će originalni pravokutnik preslikati u svaki od pravokutnika P_1 do P_8 . Krenimo redom: svaki od tih umanjenih pravokutnika velik je upravo $1/3$ originalnog – što znači da trebamo skaliranje po obje osi i to s faktorom $1/3$. Samo pomoću te transformacije originalni pravokutnik preslikava se direktno u P_1 , pa je prva transformacija w_1 jednaka:

a	b	c	d	e	f	p_i
0.33	0.0	0.0	0.33	0.0	0.0	1/8
0.33	0.0	0.0	0.33	0.33	0.0	1/8
0.33	0.0	0.0	0.33	0.67	0.0	1/8
0.33	0.0	0.0	0.33	0.0	0.33	1/8
0.33	0.0	0.0	0.33	0.67	0.33	1/8
0.33	0.0	0.0	0.33	0.0	0.67	1/8
0.33	0.0	0.0	0.33	0.33	0.67	1/8
0.33	0.0	0.0	0.33	0.67	0.67	1/8

Tablica 11.3: Tablica za IFS za konstruirani fraktal

$$w_1 : \begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 0 & 1/3 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Pravokutnik P_2 dobit ćemo na sličan način: originalni pravokutnik skalirat ćemo s $1/3$, i potom translatirati po osi x za $1/3$. Za P_3 radimo isto, samo što je pomak za $2/3$. Pravokutnici P_4 i P_5 nakon skaliranja translatirani su po osi y za $1/3$, a P_5 je dodatno translatiran i po osi x za $2/3$. Konačno, pravokutnici P_6 , P_7 i P_8 nakon skaliranja translatirani su po osi y za $2/3$, te je P_7 dodatno translatiran i po osi x za $1/3$ a P_8 za $2/3$. Ovime smo dobili i ostatak transformacija:

$$w_2 : \begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 0 & 1/3 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} 1/3 \\ 0 \end{bmatrix}$$

$$w_3 : \begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 0 & 1/3 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} 2/3 \\ 0 \end{bmatrix}$$

$$w_4 : \begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 0 & 1/3 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} 0 \\ 1/3 \end{bmatrix}$$

$$w_5 : \begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 0 & 1/3 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} 2/3 \\ 1/3 \end{bmatrix}$$

$$w_6 : \begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 0 & 1/3 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} 0 \\ 2/3 \end{bmatrix}$$

$$w_7 : \begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 0 & 1/3 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} 1/3 \\ 2/3 \end{bmatrix}$$

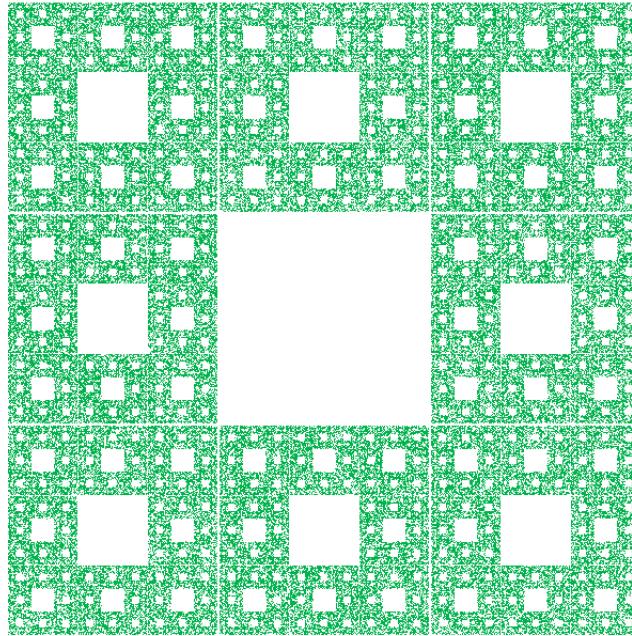
$$w_8 : \begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} 1/3 & 0 \\ 0 & 1/3 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} 2/3 \\ 2/3 \end{bmatrix}$$

Tablični prikaz ovih transformacija uobičajen kod IFS-fraktala dan je u tablici 11.3. Svim transformacijama dodjelili smo jednaku vjerojatnost ($1/8$).

Ako temeljem podataka iz tablice 11.3 preradimo izvorni kod koji je crtao trokut Sierpinskog, rezultat koji ćemo dobiti prikazan na slici 11.18. Uočite na slici položaj svakog od pravokutnika P_1 do P_8 , i njihov sadržaj.

Pokušajte temeljem ovog razmatranja objasniti kako je nastao trokut Sierpinskog, koji smo prikazali na slici 11.16. Kako su tamo pravokutnici bili složeni, odnosno koje su affine transformacije korištene?

Spomenimo još i da nam osim skaliranja i translacije na raspolaganju stoji i rotacija, čijom se primjenom mogu dobiti interesantni fraktalni oblici. A linijski segmenti (primjerice, peteljka kod paprati) dobiju se tako da se pravokutnik po jednoj dimenziji skaliranjem degenerira u segment linije (primjerice, transformacijom koja x skalira s 0); upravo je takva transformacija prikazana u prvom retku tablice 11.1.



Slika 11.18: Rezultat konstrukcije ITS fraktala – tepih Sierpinskog

11.6 Lindermayerovi sustavi

Lindermayerove sustave, ili kraće, *L-sustave* predložio je 1968. godine mađarski biolog Aristid Lindenmayer. L-sustav je formalni jezik koji se temelji na sustavima s prepisivanjem (engl. *rewriting systems*), a razvijen je kako bi se modeliralo ponašanje biljnih stanica. Osnovna ideja je vrlo jednostavna. Zamislimo sustav koji radi nad znakovnim nizovima, pri čemu znakovi dolaze iz ograničenog alfabetra. Za sustav se definira početni uzorak (koji još zovemo i *axiom*), te niz pravila koja govore kako se dijelovi postojećeg uzorka zamjenjuju novim (tipično složenijim) dijelovima. U svakom se koraku svi znakovi postojećeg niza temeljem definiranih pravila zamjenjuju novim nizovima – paralelno. Ovaj paralelizam motiviran je ponašanjem živih stanica koje se u tkivu ne dijele slijedno, već se mnoštvo dioba i promjena događa paralelno. Postoji više vrsta L-sustava, a mi ćemo se u nastavku upoznati s determinističkim kontekstno-neovisnim L-sustavom poznatim kao *DOL-sustav*.

Formalno, DOL-sustav definiran je kao uređena trojka:

$$\mathbf{G} = (\mathbf{V}, \omega, \mathbf{P})$$

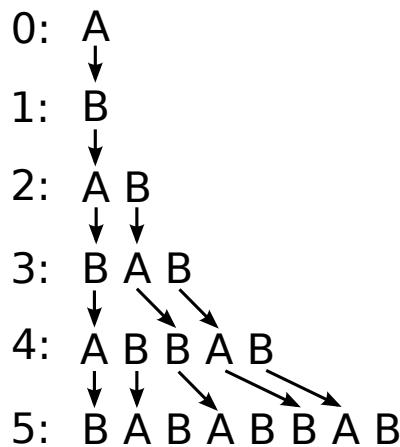
pri čemu je \mathbf{V} skup znakova alfabetra, ω početni niz odnosno aksiom te \mathbf{P} skup produkcijskih pravila.

Neka je alfabet $\mathbf{V} = A, B$, aksiom $\omega = A$ te neka je skup produkcijskih pravila $\mathbf{P} = \{A \rightarrow B, B \rightarrow AB\}$. Prvo produkcionsko pravilo kaže nam da svaki znak A trebamo zamijeniti znakom B ; drugo produkcionsko pravilo kaže nam da svaki znak B trebamo zamijeniti novim nizom AB . Prepisivanje do dubine 5 prikazano je na slici 11.19. Pogledamo li duljine znakovnih nizova nastalih u svakom koraku, uočit ćemo da ovaj sustav generira slijed Fibonaccijevih brojeva.

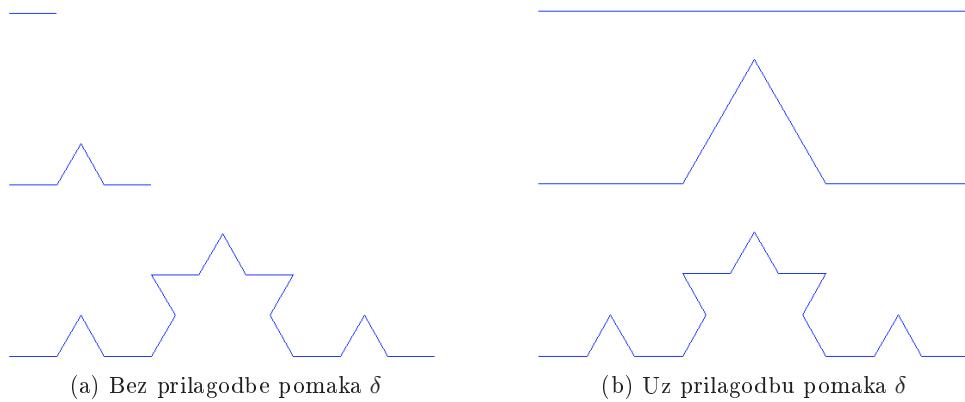
Kako bismo mogli dalje, potrebno je još nizove koje sustav generira povezati s računalnom grafikom. Odnosno, s popularno nazvanom grafikom kornjače (engl. *turtle graphics*). Evo ideje. Zamislimo malu kornjaču koju smo stavili u lijevi donji ugao ekrana i okrenuli tako da gleda prema desno. Svaki simbol generiranog niza kornjača će tumačiti kao jednu naredbu koju treba napraviti. Izvođenjem tih naredbi kornjača će se pomicati po ekranu ostavljajući za sobom trag čime će u konačnici nastati slika koja odgovara promatranom nizu. Modelirajmo kornjaču jednostavnom podatkovnom strukturu prikazanom u nastavku. Kornjača pamti svoje x i y -koordinate na ekranu, te smjer u kojem gleda. Smjer opisujemo kutem u stupnjevima, pri čemu 0 odgovara pogledu u desno.

```

1 typedef struct {
2     double x;
3     double y;
```



Slika 11.19: L-sustav za Fibonaccijeve brojeve



Slika 11.20: Kochina krivulja generirana L-sustavom

```

4     double angle;
5 } Turtle;

```

Definirajmo sada naredbe koje kornjača razumije.

Simbol	Značenje
F	Kornjača se pomiče za iznos δ u smjeru u kojem gleda. Ovo će rezultirati novom pozicijom: $x \leftarrow x + \delta \cdot \cos(\text{angle})$ te $y \leftarrow y + \delta \cdot \sin(\text{angle})$. Prilikom pomicanja na novu poziciju kornjača će na ekranu ostaviti trag.
f	Isto kao i F samo što prilikom pomicanja na novu poziciju kornjača neće na ekranu ostaviti trag.
+	Kornjača se okreće za kut ψ u smjeru suprotnom od smjera kazaljke na satu. Pozicija se ne mijenja.
-	Kornjača se okreće za kut ψ u smjeru kazaljke na satu. Pozicija se ne mijenja.

Definirajmo sada naredbe koje kornjača razumije.

Simbol	Značenje
F	Kornjača se pomiče za iznos δ u smjeru u kojem gleda. Ovo će rezultirati novom pozicijom: $x \leftarrow x + \delta \cdot \cos(\text{angle})$ te $y \leftarrow y + \delta \cdot \sin(\text{angle})$. Prilikom pomicanja na novu poziciju kornjača će na ekranu ostaviti trag.
f	Isto kao i F samo što prilikom pomicanja na novu poziciju kornjača neće na ekranu ostaviti trag.
+	Kornjača se okreće za kut ψ u smjeru suprotnom od smjera kazaljke na satu. Pozicija se ne mijenja.
-	Kornjača se okreće za kut ψ u smjeru kazaljke na satu. Pozicija se ne mijenja.

Uz ovako definirane naredbe spremni smo za crtanje Kochine krivulje uporabom L-sustava:

$$\mathbf{G} = (\{F, +, -\}, F, \{F \rightarrow F + F - -F + F\}).$$

Slika koju ovaj sustav generira prikazana je na slici 11.20a. Pri tome je kao parametar δ korištena fiksna vrijednost 55 piksela, a kut δ bio je 60. Tri slike prikazane na slici 11.20a pri tome odgovaraju aksiomu F (gornja slika), nizu $F + F - -F + F$ koji odgovara prvoj iteraciji primjene produkcijskih

pravila na aksiom (srednja slika) te nizu $F + F --F + F + F + F --F + F --F + F + F + F --F + F$ koji odgovara drugoj iteraciji primjene produkcijskih pravila na aksiom (donja slika). Budući da tijekom crtanja pomak δ držimo fiksним, slika koju dobivamo svakom sljedećom iteracijom sve je veća i veća.

Problemu s rastom slike možemo doskočiti tako da se prisjetimo koja je karakteristika Kochine krivulje: prilikom supstitucije svaki segment mijenja se umanjenom kopijom kod koje su duljine segmentata jednake trećini duljine promatranog segmenta. To će za posljedicu imati eksponencijalan pad duljine segmenta s iteracijama. Označimo s δ_0 početnu duljinu jednog segmenta (dakle, ono što se crta aksiomom). Tada ćemo kao iznos pomaka koji ćemo koristiti ako crtamo niz dobiven L-sustavom uz ograničenje dubine na d koristiti $\delta = \delta_0 \cdot \left(\frac{1}{3}\right)^d$. Slike dobivene uz takvu korekciju prikazane su na slici 11.20b, gdje se može uočiti da su uz sve tri dubine (0, 1 i 2) generirane slike jednakih dimenzija.

Isječak koda koji crta Kochinu krivulju na opisani način prikazan je u nastavku. Pri tome su prikazane samo metode ključne za crtanje navedene slike. Dijelovi koda koji inicijaliziraju GLUT i pripremaju opće postavke isti su kao i svim do sada prikazanim primjerima za 2D slike. Metoda pronadiPravilo(...) u skupu pravila traži pravilo koje je primjenjivo na zadani simbol. Ako takvo pravilo ne postoji, metoda vraća NULL. Metoda moveTurtle(...) ažurira poziciju kornjače zadanom pomakom u smjeru u kojem kornjača trenutno gleda.

Metoda lSustav(...) predstavlja implementaciju sustava prepisivanja do zadane dubine. Metoda prima *aksiom* te zadanu dubinu, i generira konačni niz znakova uporabom definiranih pravila. U svakoj iteraciji ova metoda prolazi kroz dotada generirani niz i za svaki znak niza traži pravilo koje kaže čime se znak mijenja. Ako se ne pronađe odgovarajuće pravilo, metoda prepisuje taj znak (ponaša se kao da je pronađeno pravilo oblika $x \rightarrow x$). Metoda vraća konačni niz znakova.

Konačno, metoda renderScene() stvara početni niz (postavlja ga na aksiom), poziva metodu lSustav (...) i potom crta prikaz koji odgovara tumačenju generiranog niza znakova preko prethodno definirane tablice značenja simbola. Ovaj programski isječak još koristi i nešto pomoćnog koda vezanog uz strukturu CharSequence koji nije prikazan i ostavlja se čitatelju da ga implementira za vježbu. Navedena struktura kao i korištene metode omogućavaju nam rad sa znakovnim nizom koji sam povećava svoj kapacitet onom dinamikom kojom mu nadodajemo znakove. Na tu se strukturu može gledati kao na polje koje se samo povećava kako mu nadodajemo elemente – pa je stoga vrlo zgodno za uporabu u L-sustavima.

```

1 #define PI (3.141592653589793)
2
3 char *RULE1 = "F>F+F--F+F";
4 char *rules [] = {RULE1,NULL};
5
6 char *pronadiPravilo(char simbol) {
7     int i = 0;
8     while(true) {
9         if(rules[i]==NULL) return NULL;
10        if(rules[i][0]==simbol) return rules[i]+2;
11        i++;
12    }
13 }
14
15 void moveTurtle(Turtle *turtle, double pomak) {
16     turtle->x += pomak * cos(turtle->angle*PI/180.0);
17     turtle->y += pomak * sin(turtle->angle*PI/180.0);
18 }
19
20 CharSequence *lSustav(CharSequence *axiom, int dubina) {
21     CharSequence *niz = copyOfCharSequence(axiom);
22     for(int i = 0; i < dubina; i++) {
23         CharSequence *novi = allocCharSequence();
24         for(int i = 0; i < niz->n; i++) {
25             char simbol = niz->data[i];
26             char *pravilo = pronadiPravilo(simbol);
27             if(pravilo==NULL) {
28                 charSequenceAdd(novi, simbol);
29             } else {

```

```

30             charSequenceAddString( novi , pravilo );
31         }
32     }
33     freeCharSequence ( niz );
34     niz = novi;
35 }
36 return niz ;
37 }

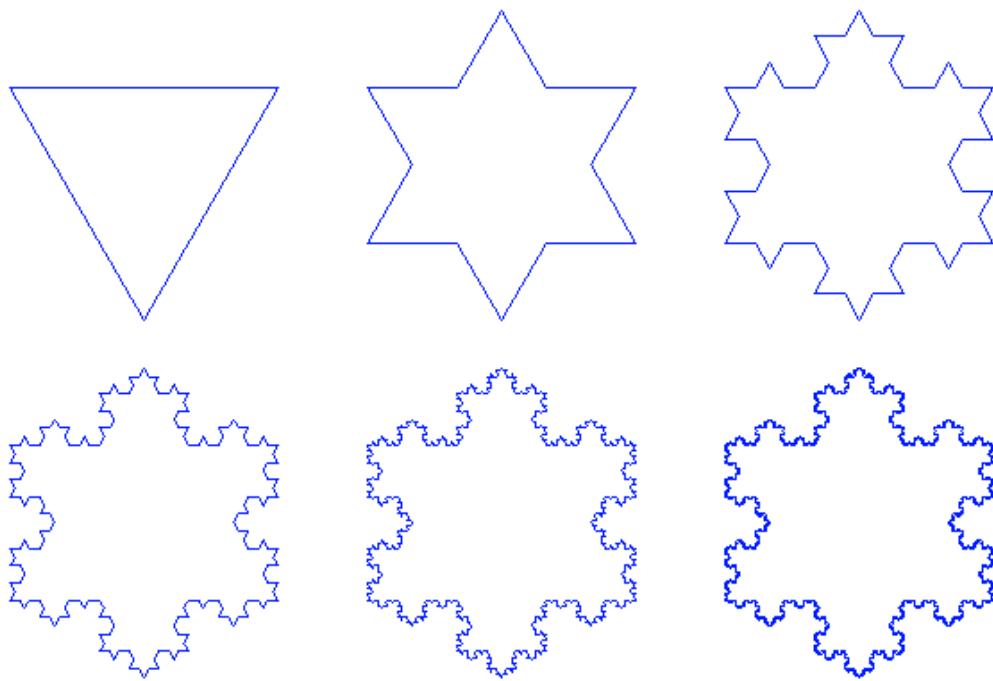
38 void renderScene() {
39
40     int dubina = 5;
41
42     CharSequence *axiom = allocCharSequenceFromString ( "F" );
43     CharSequence *niz = lSustav(axiom, dubina);
44
45     double pomak = 500 * pow(1.0/3.0 ,dubina );
46     double angle = 60;
47
48     Turtle turtle;
49     turtle.angle = 0;
50     turtle.x = 50;
51     turtle.y = 150;
52
53     glColor3f ( 0.0f , 0.1f , 1.0f );
54     glPointSize ( 1 );
55
56     for ( int i = 0; i < niz->n; i++ ) {
57         glBegin(GL_LINE);
58         char simbol = niz->data[ i ];
59         if ( simbol== 'F' ) {
60             // zapamti staru poziciju
61             double x0 = turtle.x;
62             double y0 = turtle.y;
63             moveTurtle(&turtle , pomak);
64             // zapamti novu poziciju
65             double x1 = turtle.x;
66             double y1 = turtle.y;
67             // nacrtaj liniju stara-nova
68             glVertex2i( zaokruzi(x0) , zaokruzi(y0));
69             glVertex2i( zaokruzi(x1) , zaokruzi(y1));
70         } else if ( simbol== '+' ) {
71             turtle.angle += angle;
72         } else if ( simbol== '-' ) {
73             turtle.angle -= angle;
74         }
75     }
76     glEnd();
77 }
78
79     freeCharSequence ( niz );
80     freeCharSequence ( axiom );
81 }
```

Kochinu pahuljicu možemo nacrtati uporabom sljedećeg L-sustava:

$$\mathbf{G} = (\{F, +, -\}, F --F --F, \{F \rightarrow F + F --F + F\}).$$

Uočimo da se je u odnosu na prethodni primjer promjenio samo aksiom, koji crta jednakostranični trokut. Rezultat je prikazan na slici 11.21.

Zajednička karakteristika svih dosad prikazanih L-sustava je bila generiranje neprekinute krivulje. Međutim, često to nam nije dovoljno. Primjerice, stablo je struktura koja se grana, i problem koji se tu javlja jest kojim putem krenuti kod grananja? Očit odgovor je: svim putevima – trebamo nacrtati sve grane. Međutim, kako je kornjača ograničena na jednu poziciju u jednom trenutku, to očito nije moguće direktno postići. Jedno moguće rješenje bi bilo da kornjača dolaskom na grananje negdje zapamti svoju trenutnu poziciju, i potom krene jednom granom. Nakon što granu nacrtata, kornjača se



Slika 11.21: Kochina pahuljica generirana L-sustavom

teleportira na prethodno zapamćenu poziciju (u redu je, problem je složen pa je opravdana posudba *Star Trek* tehnologije), i kreće u drugu granu. Kako će stablaste strukture tipično imati više od jednog grananja, potreban nam je stog kornjačinih stanja. Za rad sa stogom definirat ćemo dva nova simbola "[" i "]". Također, kornjača u svom stanju osim trenutne pozicije i smjera gledanja može pamtitи druge parametre, poput boje kojom crta, debljine linije i slično. Definirajmo stoga dodatni simbol "w" koji će kornjači naložiti da smanji korištenu debljinu linije kojom crta za određeni postotak, te simbol "s" koji će kornjači naložiti da smanji duljinu segmenta koji crta. Evo tabličnog prikaza novih simbola.

Simbol	Značenje
[Kornjača trenutno stanje zapisuje na stog.
]	Kornjača trenutno stanje restaurira sa stoga.
w	Debljinu linije potrebno je smanjiti za propisani faktor.
s	Duljinu segmenta potrebno je skratiti za propisani faktor.

Pogledajmo nekoliko primjer sustava s grananjima. Prvi sustav čija je slika prikazana na slici 11.22a definiran je kao:

$$\mathbf{G} = (\{F, +, -, s, w, [,]\}, FX, \{X \rightarrow sw[-FX] + FX\}).$$

Metoda `renderScene()` koja prikazuje sliku koju generira ovaj sustav prikazana je u nastavku. Ujedno je i redefinirana podatkovna struktura koja pamti stanje kornjače dodavanjem podataka o trenutnoj debljini linije te trenutnoj duljini segmenta koji se isrtava pod djelovanjem simbola "F". Kut za koji se mijenja smjer kornjače pod djelovanjem simbola "+" i "-" je 50. Simbol "s" trenutnu duljinu segmenta množi s 0.6 dok simbol "w" trenutnu širinu linije množi s 0.7. U kodu se može vidjeti i uporaba podatkovne strukture `TurtleStack` – stoga na koji se stanje kornjače potiskuje svaki puta kada se nađe na simbol "[", odnosno s kojeg se stanje vadi svaki puta kada se nađe na simbol "]". Programski kod koji opslužuje ovu strukturu nije prikazan i ostavlja se čitateljima za vježbu.

```

1 typedef struct {
2     double x;
3     double y;
4     double angle;
5     double penSize;
6     double segmentSize;
7 } Turtle;
```

```

8
9 void renderScene() {
10
11     int dubina = 7;
12
13     CharSequence *axiom = allocCharSequenceFromString ("FX");
14     CharSequence *niz = lSustav(axiom, dubina);
15
16     double angle = 50;
17
18     double penFactor = 0.7;
19     double segmentFactor = 0.6;
20
21     Turtle turtle;
22     turtle.angle = 90;
23     turtle.x = 300;
24     turtle.y = 10;
25     turtle.penSize = 10;
26     turtle.segmentSize = 220;
27
28     glColor3f(0.0f, 0.1f, 1.0f);
29     glPointSize(1);
30
31     TurtleStack *stack = newTurtleStack();
32     for(int i = 0; i < niz->n; i++) {
33         char simbol = niz->data[i];
34         if(simbol=='F') {
35             glLineWidth((float)turtle.penSize);
36             glBegin(GL_LINE);
37             double x0 = turtle.x;
38             double y0 = turtle.y;
39             moveTurtle(&turtle, turtle.segmentSize);
40             double x1 = turtle.x;
41             double y1 = turtle.y;
42             glVertex2i(zaokruzi(x0), zaokruzi(y0));
43             glVertex2i(zaokruzi(x1), zaokruzi(y1));
44             glEnd();
45         } else if(simbol=='+') {
46             turtle.angle += angle;
47         } else if(simbol=='-') {
48             turtle.angle -= angle;
49         } else if(simbol=='s') {
50             turtle.segmentSize = max(1.0, turtle.segmentSize*segmentFactor);
51         } else if(simbol=='w') {
52             turtle.penSize = max(1.0, turtle.penSize*penFactor);
53         } else if(simbol=='[') {
54             pushTurtleStack(stack, &turtle);
55         } else if(simbol==']') {
56             popTurtleStack(stack, &turtle);
57         }
58     }
59     freeTurtleStack(stack);
60
61     freeCharSequence(niz);
62     freeCharSequence(axiom);
63 }
```

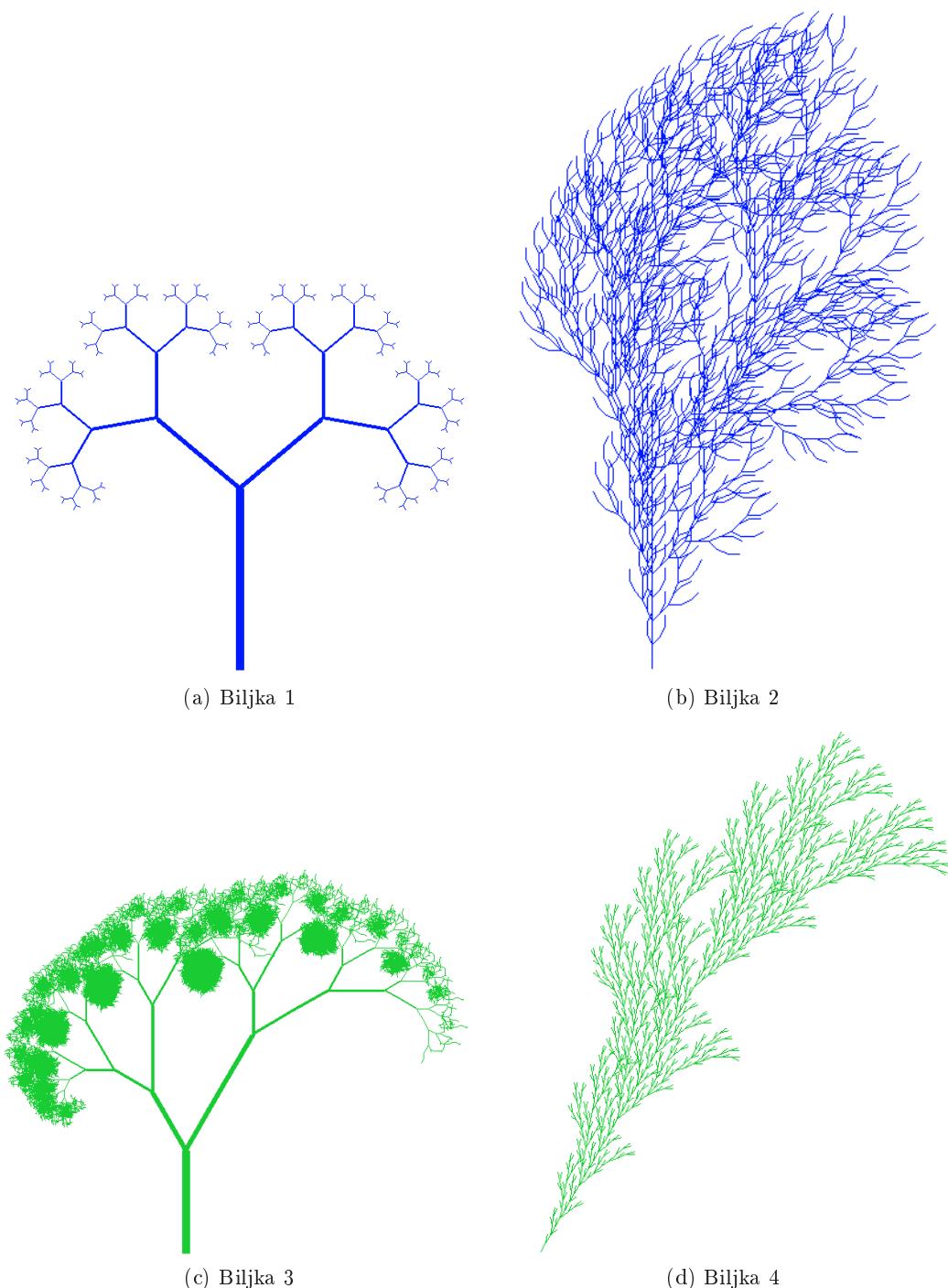
L-sustav čija je slika prikazana na slici 11.22b definiran je kao:

$$\mathbf{G} = (\{F, X, Y, +, -, [,]\}, F, \{F \rightarrow FF - [-F + F + F] + [+F - F - F]\}).$$

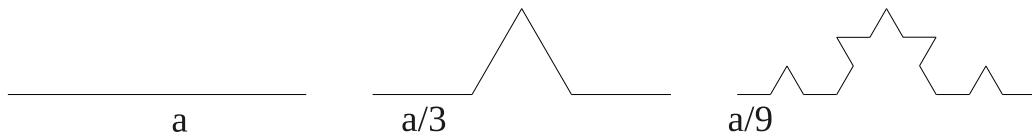
Kut za koji se mijenja smjer kornjače pod djelovanjem simbola "+" i "-" je 22.5.

L-sustav čija je slika prikazana na slici 11.22c definiran je kao:

$$\mathbf{G} = (\{F, +, -, s, w, [,]\}, + + FX, \{X \rightarrow sw[-FY] + FX, Y \rightarrow FX + FY - FX\}).$$



Slika 11.22: L-sustavom s grananjem



Slika 11.23: Opseg Kochine krivulje

Kut za koji se mijenja smjer kornjače pod djelovanjem simbola "+" i "-" je 30. Simbol "s" trenutnu duljinu segmenta množi s 0.65 dok simbol "w" trenutnu širinu linije množi s 0.7.

L-sustav čija je slika prikazana na slici 11.22d definiran je kao:

$$\mathbf{G} = (\{F, +, -, [,]\}, F, \{F \rightarrow F[+F]F[-F][F]\}).$$

Kut za koji se mijenja smjer kornjače pod djelovanjem simbola "+" i "-" je 20.

L-sustavi, općenito govoreći, mogu biti puno kompleksniji od ovdje opisanih. Primjerice, jedna modifikacija su stohastički L-sustavi, kod kojih za pojedine simbole može biti definirano više produkcija te svaka produkcija može imati vjerojatnost da će baš ona biti odabrana; u tom slučaju, simbol se zamjenjuje prema produkciji koja se odabere posredstvom slučajnog mehanizma i pridijeljenih vjerojatnosti. Druga modifikacija jest uporaba kontekstno ovisnih produkcija, koje s lijeve strane nemaju samo simbol X već mogu imati i niz α koji se mora nalaziti prije tog simbola i niz β koji se mora nalaziti nakon tog simbola. Tada će produkcija "paliti" i obaviti zamjenu samo ako se u promatranom nizu neposredno lijevo od simbola X nalazi niz α te ako se od njega neposredno desno nalazi niz β . Također, osim što mogu generirati 2D slike, L-sustavi mogu raditi i u 3D prostoru generirajući različite trodimenzijske objekte. U ovom poglavlju stoga je predstavljena ideja L-sustava, te je dan samo osnovni pregled DOL-sustava.

11.7 Opseg i površina frakta. Fraktalna dimenzija.

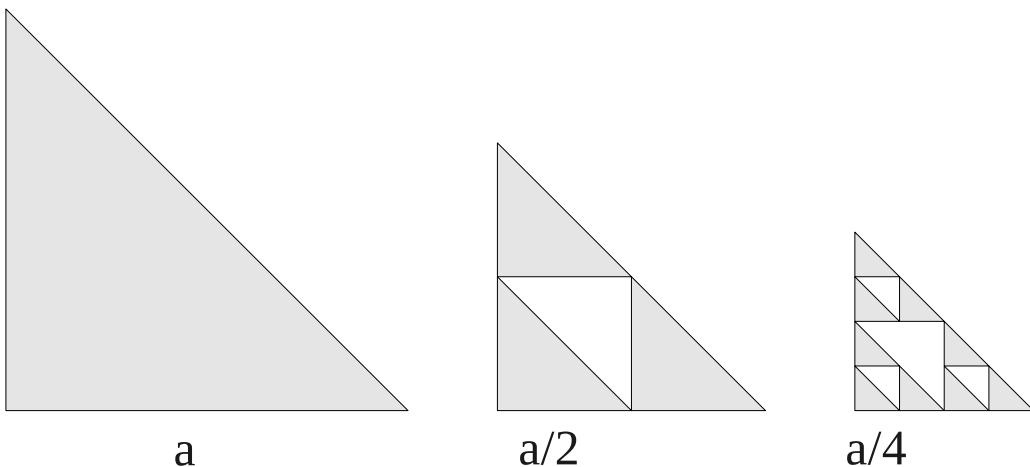
Započnimo najprije s izračunom opsega Kochine krivulje. Neka je početni segment Kochine krivulje segment linije duljine a (slika 11.23). Taj segment mijenjamo s 4 segmenta, svaki duljine $a/3$, pa je nakon prvog koraka duljina tako dobivene krivulje $4 \times (a/3) = a \cdot \frac{4}{3}$. Promotrimo sada jedan mali segment te krivulje. U sljedećem koraku taj segment duljine $a/3$ zamjenit ćemo s nova 4 segmenta čija je pojedinačna duljina trećina promatrane: $(a/3)/3$. Kako ih ima 4, segment duljine $a/3$ zamjenili smo konstrukcijom duljine $4 \times \frac{a}{9}$. Krivulja je imala 4 segmenta duljine $a/3$, pa je duljina krivulje nakon drugog koraka jednaka $4 \times 4 \times \frac{a}{9} = a \cdot \frac{4^2}{3}$. Sada možemo uočiti uzorak koji se pojavljuje:

$$a \rightarrow a \cdot \frac{4}{3} \rightarrow a \cdot \left(\frac{4}{3}\right)^2 \rightarrow a \cdot \left(\frac{4}{3}\right)^3 \rightarrow \dots \rightarrow a \cdot \left(\frac{4}{3}\right)^\infty = \infty$$

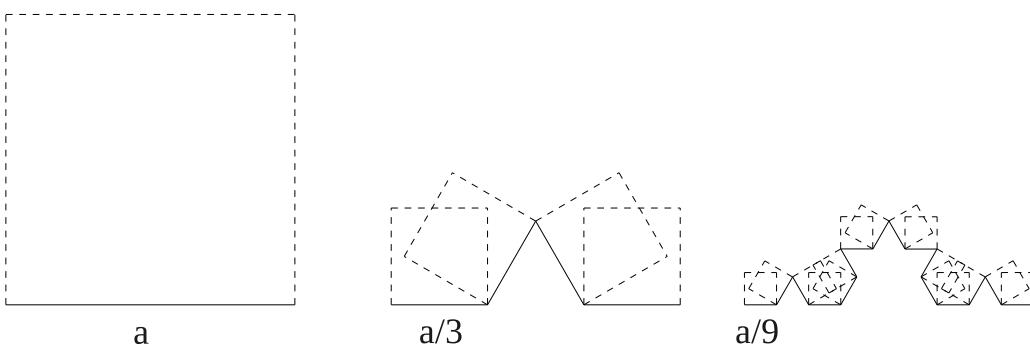
Možemo dakle zaključiti da je duljina Kochine krivulje doista beskonačna. Pogledajmo sada primjer fraktalnog lika – trokut Sierpinskog, pa izračunajmo njegov opseg i površinu (slika 11.24). Krenimo od jednakostaničnog trokuta koji ima stranicu duljine a . Opseg tog trokuta je $3 \cdot a$, površina $a^2/2$. Početni trokut sada zamjenimo s tri trokuta čija je stranica jednaka polovici početne stranice. Ukupni opseg takvog lika je suma triju opsega manjih trokuta: $3 \cdot a/2 + 3 \cdot a/2 + 3 \cdot a/2 = 3 \cdot a \cdot \frac{3}{2}$. Površina je jednaka sumi površina triju manjih trokuta: $(\frac{a}{2})^2/2 + (\frac{a}{2})^2/2 + (\frac{a}{2})^2/2 = \frac{a^2}{2} \cdot \frac{3}{4}$. U sljedećem koraku konstrukcije svaki bi trokut stranice $a/2$ bio zamijenjen s 3 trokuta upola manje stranice, dakle $a/4$. Time bi ukupni opseg tog lika bio $3 \cdot a \cdot (\frac{3}{2})^2$, a ukupna površina $\frac{a^2}{2} \cdot (\frac{3}{4})^2$. Opet možemo uočiti uzorak koji se pojavljuje:

$$3a \rightarrow 3a \cdot \frac{3}{2} \rightarrow 3a \cdot \left(\frac{3}{2}\right)^2 \rightarrow 3a \cdot \left(\frac{3}{2}\right)^3 \rightarrow \dots \rightarrow 3a \cdot \left(\frac{3}{2}\right)^\infty = \infty$$

$$a^2/2 \rightarrow a^2/2 \cdot \frac{3}{4} \rightarrow a^2/2 \cdot \left(\frac{3}{4}\right)^2 \rightarrow a^2/2 \cdot \left(\frac{3}{4}\right)^3 \rightarrow \dots \rightarrow a^2/2 \cdot \left(\frac{3}{4}\right)^\infty = 0$$



Slika 11.24: Površina i opseg trokuta Sierpinskog



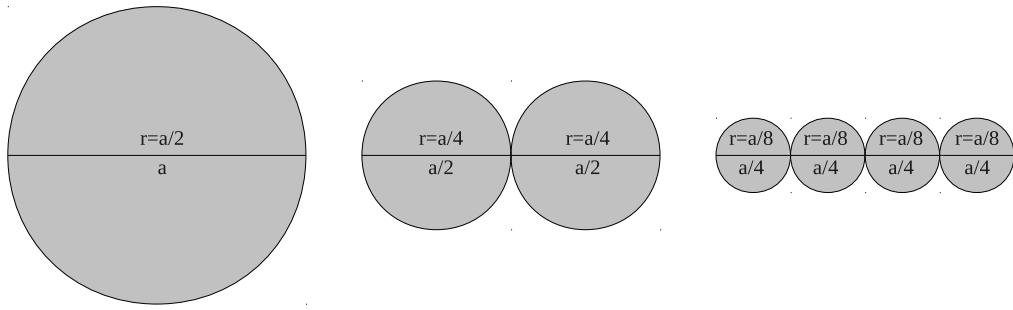
Slika 11.25: Površina Kochine krivulje

Trokut Sierpinskog ima dakle beskonačan ukupni opseg te površinu jednaku 0.

Pogledajmo malo rezultate prethodnih izračuna. Krenimo s Kochinom krivuljom. Promotrimo skup točaka koji čini tu krivulju. Zanima nas koje je dimenzionalnosti taj objekt? Je li to 1D objekt? Kada bi bio, teško bismo mogli objasniti kako to da je njegova duljina beskonačna – a opet, objekt je smješten na konačnoj površini! Možda je to dvodimenzijijski objekt? U tom slučaju, mogli bismo pokušati procijeniti njegovu površinu; dapače, ne točnu, već barem naći nekakvu gornju ogralu. Evo eksperimenta. Aproksimirajmo površinu ovog objekta sumom površina kvadrata koje postavimo iznad svakog segmenta krivulje (slika 11.25). Ovo će očito biti vrlo gruba procjena, i puno prevelika. No, rezultat će biti zanimljiv.

Konstrukciju Kochine krivulje započinjemo jednim segmentom linije duljine a . U ovom koraku, kako imamo samo jedan segment, površinu aproksimiramo površinom pripadnog kvadrata: a^2 . Napravimo sada jedan korak izgradnje krivulje: liniju zamijenimo s 4 segmenta, svaki duljine $a/3$. Nam njima možemo podignuti 4 kvadrata stranice $a/3$, pa je suma njihovih površina jednaka $4 \cdot (a/3)^2 = a^2 \cdot \frac{4}{9}$. Napravimo li sada sljedeći korak u konstrukciji, svaki segment duljine $a/3$ zamijenit ćemo s 4 segmenta duljine $(a/3)/3$. Suma površina svih kvadrata nad tim segmentima bit će tada: $a^2 \cdot \frac{4^2}{3^4}$. Sljedeći će korak dati površinu od $a^2 \cdot \frac{4^3}{3^6}$. Iz ovoga je lagano uočiti da je procjena površine u koraku n dana formulom: $a^2 \cdot \frac{4^n}{3^{(2n)}}$ što je zapravo $a^2 \cdot (\frac{4}{9})^n$. Kada n pustimo da teži u beskonačnost, površina teži k nuli. Iz ovog eksperimenta možemo zaključiti da objekt nije niti dvodimenzijijski – nema površine. Stoga je zaključak da je dimenzija ove krivulje negdje između 1 i 2.

Hausdorffova dimenzija nastala je kao pokušaj da se definira dimenzionalnost i ovakvih objekata. Bez ulazeњa u formalnu definiciju Hausdorffove dimenzije, pogledajmo samo konačan rezultat: kako se računa? Ideja je sljedeća: promatrani skup točaka koji čini promatrani objekt potrebno je obuhvatiti kuglama radijusa r . Očito je da ako smanjujemo radijus r , trebat ćemo više kugli; označimo broj



Slika 11.26: Određivanje Hausdorffove dimenzije segmenta linije

kugli potreban za prekrivanje objekta Θ s $N_\Theta(r)$. Hausdorffova dimenzija definirana je kao limes od negativnog omjera logaritma potrebnog broja kugli i logaritma radiusa tih kugli, kada radijus teži ka nuli. Pri tome pojam *kugla* treba shvatiti u kontekstu n -dimenzijskih hiperkugli (što primjerice, za $n = 2$ odgovara kružnici). Možemo pisati:

$$d_H(\Theta) = - \lim_{r \rightarrow 0} \frac{\log N_\Theta(r)}{\log r} \quad (11.7)$$

Hausdorffova mjera odgovara našem uobičajenom poimanju dimenzije. Pogledajmo dva primjera.

Primjer: 12

Neka je kao objekt zadan segment linije. Odredite Hausdorffovu dimenziju tog objekta.

Rješenje:

Neka je duljina segmenta linije jednaka a (slika 11.26). Čitav segment moguće je obuhvatiti jednom kružnicom čiji je centar u središtu segmenta, a radijus $a/2$. Međutim, treba vidjeti što se dogada ako smanjujemo r . Podijelimo stoga naš segment na dva jednakog dugačka dijela: lijevi i desni. Svaki od njih ima duljinu $a/2$. Svaki segment moguće je obuhvatiti jednom kružnicom radiusa $a/4$ čiji je centar smješten u središtu tog segmenta. Broj takvih kugli potreban da se prekrije čitav objekt bit će 2. Ako nastavimo sa smanjivanjem, početnu liniju možemo rekurzivno podijeliti na 4 jednakog dijela. Tada će nam trebati 4 kugle radijusa $a/8$, kako bismo prekrili čitav objekt. Trend je prikazan u sljedećoj tablici:

Dubina rekurzije	0	1	2	...	n
Broj kugli H_Θ	1	2	4	...	2^n
Radijus kugle r	$a/2$	$a/4$	$a/8$...	$\frac{a}{2^{n+1}}$

Očito je da puštanjem $n \rightarrow \infty$ radijus r teži ka nuli. Pogledajmo stoga kamo teži omjer broja kugli i radijusa:

$$\begin{aligned} d_H(\Theta) &= - \lim_{r \rightarrow 0} \frac{\log N_\Theta(r)}{\log r} \\ &= - \lim_{n \rightarrow \infty} \frac{\log 2^n}{\log \frac{a}{2^{n+1}}} \\ &= - \lim_{n \rightarrow \infty} \frac{n \cdot \log 2}{\log \frac{a}{2} - n \log 2} \\ &= 1 \end{aligned}$$

Zaključak: segment linije je jednodimenzijski objekt, što je u skladu s našim očekivanjima.

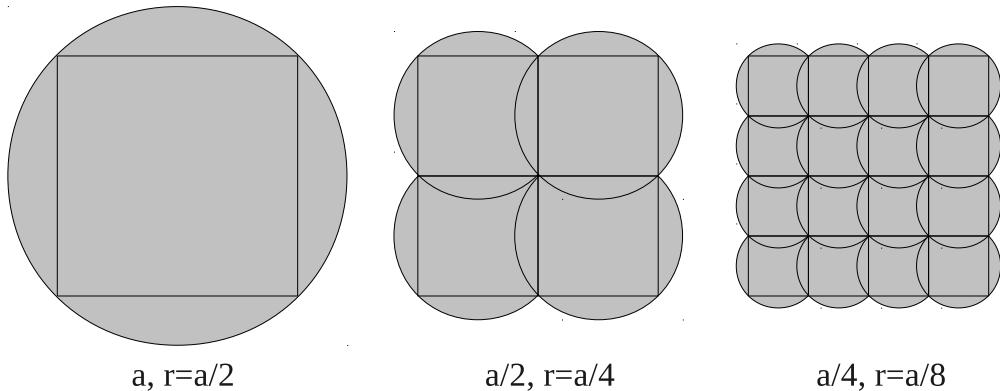
Primjer: 13

Neka je kao objekt zadan kvadrat. Odredite Hausdorffovu dimenziju tog objekta.

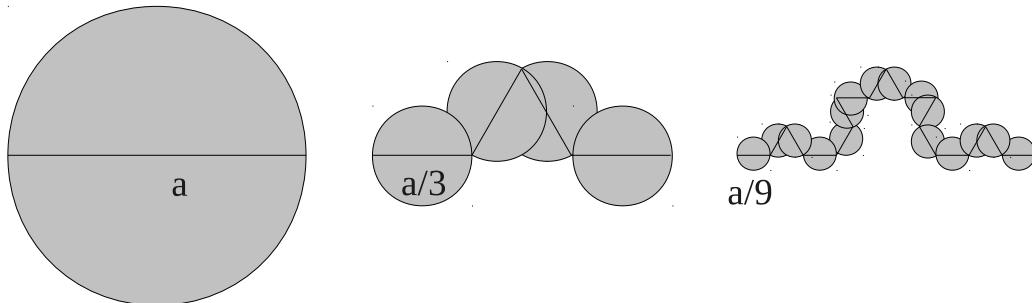
Rješenje:

Neka je duljina stranice kvadrata jednaka a (slika 11.27). Čitav kvadrat moguće je obuhvatiti jednom kružnicom čiji je centar u središtu kvadrata, a radijus $a/\sqrt{2}$. Podijelimo sada kvadrat kvadrate čija je stranica $a/2$. Početni kvadrat ovime će se raspasti na 4 manja kvadrata. Svaki od njih ima stranicu $a/2$, pa ćemo ga moći obuhvatiti u kružnicu radijusa $\frac{a/2}{\sqrt{2}}$. Za pokriti čitav lik trebat ćemo 4 kružnice. Ako postupak nastavimo tako da svaki od manjih kvadrata rekurzivno podijelimo, dobit ćemo još manje kvadratne duljine stranice $a/4$. Čitav lik tada će se sastojati od $4 \cdot 4 = 16$ manjih kvadrata, a svaki ćemo moći obuhvatiti jednom kružnicom radijusa $\frac{a/4}{\sqrt{2}}$. Trend je prikazan u sljedećoj tablici:

Dubina rekurzije	0	1	2	...	n
Broj kugli H_Θ	1	4	16	...	4^n
Radijus kugle r	$a/\sqrt{2}$	$\frac{a/2}{\sqrt{2}}$	$\frac{a/4}{\sqrt{2}}$...	$\frac{a}{2^n \sqrt{2}}$



Slika 11.27: Određivanje Hausdorffove dimenzije segmenta kvadrata



Slika 11.28: Određivanje Hausdorffove dimenzije Kochine krivulje

Očito je da puštanjem $n \rightarrow \infty$ radijus r teži ka nuli. Pogledajmo stoga kamo teži omjer broja kugli i radiusa:

$$\begin{aligned} d_H(\Theta) &= -\lim_{r \rightarrow 0} \frac{\log N_\Theta(r)}{\log r} \\ &= -\lim_{n \rightarrow \infty} \frac{\log 4^n}{\log \frac{a}{2^n \sqrt{2}}} \\ &= -\lim_{n \rightarrow \infty} \frac{2n \cdot \log 2}{\log \frac{a}{\sqrt{2}} - n \log 2} \\ &= 2 \end{aligned}$$

Zaključak: kvadrat je dvodimenzinski objekt, što je opet u skladu s našim očekivanjima.

Nakon što smo se uvjerili da Hausdorffova mjera doista odgovara onome što smo i očekivali za 1D i 2D objekte, pokušajmo sada na isti način izračunati Hausdorffovu dimenziju Kochine krivulje kao jednog od najjednostavnijih samosličnih fraktala.

Primjer: 14

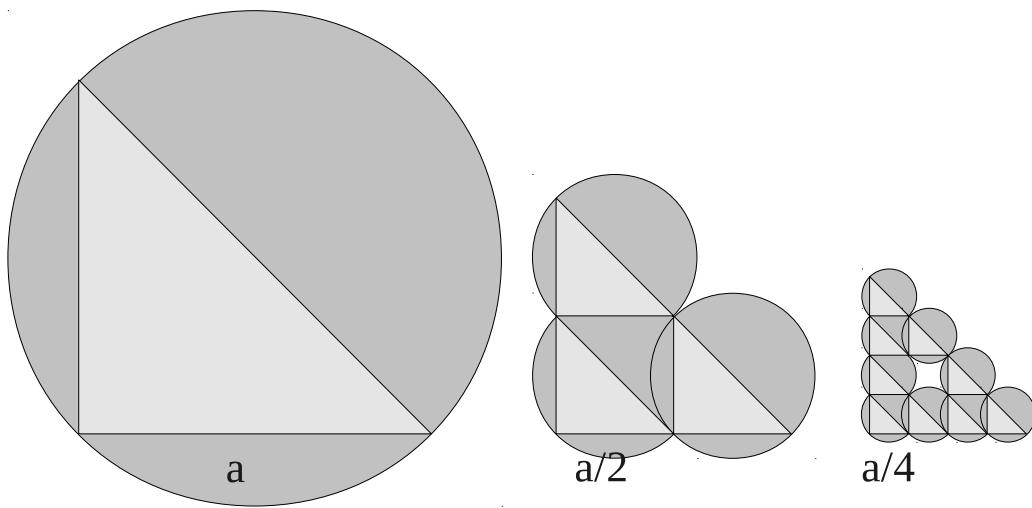
Neka je kao objekt zadana Kochina krivulja. Odredite Hausdorffovu dimenziju tog objekta.

Rješenje:

Neka je duljina početnog segmenta linije iz kojeg kreće konstrukcija jednaka a (slika 11.28). Čitav segment moguće je obuhvatiti jednom kružnicom čiji je centar u središtu segmenta, a radijus $a/2$. Napravimo li sljedeći korak konstrukcije, jedan segment duljine a zamijenit ćemo s 4 segmenta, svaki duljine $a/3$. Taj objekt moći ćemo obuhvatiti s 4 kružnice, svaka radijusa $a/6$. Napravimo li sljedeći korak konstrukcije, jedan segment duljine $a/3$ zamijenit ćemo s 4 segmenta, svaki duljine $a/9$. Čitav objekt tada ćemo moći obuhvatiti s $4 \cdot 4 = 16$ kružnicama, svaka radijusa $a/18$. Nastavimo li dalje, trend je prikazan u sljedećoj tablici:

Dubina rekurzije	0	1	2	...	n
Broj kugli H_Θ	1	4	16	...	4^n
Radijus kugle r	$a/2$	$\frac{a/2}{3}$	$\frac{a/2}{9}$...	$\frac{a}{3^n}$

Očito je da puštanjem $n \rightarrow \infty$ radijus r teži ka nuli. Pogledajmo stoga kamo teži omjer broja kugli i radiusa:



Slika 11.29: Određivanje Hausdorffove dimenzije trokuta Sierpinskog

$$\begin{aligned}
 d_H(\Theta) &= -\lim_{r \rightarrow 0} \frac{\log N_\Theta(r)}{\log r} \\
 &= -\lim_{n \rightarrow \infty} \frac{\log 4^n}{\log \frac{a}{2 \cdot 3^n}} \\
 &= -\lim_{n \rightarrow \infty} \frac{n \cdot \log 4}{\log \frac{a}{2} - n \log 3} \\
 &= \frac{\log 4}{\log 3} \\
 &= 1.2618595...
 \end{aligned}$$

Zaključak: Hausdorffova dimenzija Kochine krivulje je poprilici 1.26, što je između 1 i 2, baš kao što smo to i očekivali.

Izračunajmo i Hausdorffovu dimenziju trokuta Sierpinskog.

Primjer: 15

Neka je kao objekt zadan trokut Sierpinskog. Odredite Hausdorffovu dimenziju tog objekta.

Rješenje:

Neka je duljina stranice početnog pravokutnog jednakokračnog trokuta iz kojeg kreće konstrukcija jednakna a (hipotenuza je tada $a\sqrt{2}$) – slika 11.29. Čitav trokut moguće je obuhvatiti jednom kružnicom čiji je centar u središtu hipotenuze trokuta, a radijus $\frac{a\sqrt{2}}{2}$. Napravimo li sljedeći korak konstrukcije, jedan trokut kraka a zamijenit ćemo s 3 trokuta, svaki kraka $a/2$. Taj objekt moći ćemo obuhvatiti s 3 kružnice, svaka radijusa $\frac{a\sqrt{2}}{4}$. Napravimo li sljedeći korak konstrukcije, svaki trokut kraka duljine $a/2$ zamijenit ćemo s 3 nova trokuta, svaki duljine kraka $a/4$. Čitav objekt tada ćemo moći obuhvatiti s 9 kružnica, svaka radijusa $\frac{a\sqrt{2}}{8}$. Nastavimo li dalje, trend je prikazan u sljedećoj tablici:

Dubina rekurzije	0	1	2	...	n
Broj kugli H_Θ	1	3	9	...	3^n
Radijus kugle r	$\frac{a\sqrt{2}}{2}$	$\frac{a\sqrt{2}}{2 \cdot 2}$	$\frac{a\sqrt{2}}{2 \cdot 2^2}$...	$\frac{a\sqrt{2}}{2 \cdot 2^n}$

Očito je da puštanjem $n \rightarrow \infty$ radijus r teži ka nuli. Pogledajmo stoga kamo teži omjer broja kugli i radijusa:

$$\begin{aligned}
 d_H(\Theta) &= -\lim_{r \rightarrow 0} \frac{\log N_\Theta(r)}{\log r} \\
 &= -\lim_{n \rightarrow \infty} \frac{\log 3^n}{\log \frac{a\sqrt{2}}{2 \cdot 2^n}} \\
 &= -\lim_{n \rightarrow \infty} \frac{n \cdot \log 3}{\log \frac{a\sqrt{2}}{2} - n \log 2} \\
 &= \frac{\log 3}{\log 2} \\
 &= 1.5849625...
 \end{aligned}$$

Zaključak: Hausdorffova dimenzija trokuta Sierpinskog iznosi poprilici 1.58, što je između 1 i 2.

Izračunajte za vježbu Hausdorffovu dimenziju tepiha Sierpinskog, prikazanog na slici 11.18.

Spomenimo da osim Hausdorfove dimenzije postoji još sličnih mjera, koje se u velikom broju slučaju podudaraju s Hausdorffovom dimenzijom (no postoje i objekti za koje se te mjere razlikuju u odnosu na Hausdorffovu dimenziju). Jedan od poznatijih primjera je Minkowski-Bouligand-ova dimenzija, poznata još i pod nazivom *box-counting* dimenzija, koja je uvijek veća ili jednaka Hausdorffovoj dimenziji. Prilikom izračuna ove dimenzije objekt se ne prekriva kružnicama, već se prekriva pravilnom kvadratičnom rešetkom. Neka su dimenzije jednog kvadratiča takve rešetke $\epsilon \times \epsilon$. Potrebno je izbrojati preko koliko se kvadratiča proteže fraktal, ili rečeno drugačije, koliko nam kvadratiča treba da bismo prekrili fraktal. Potom pustimo duljinu stranice da teži ka nuli (rešetka postaje sve finija i finija), i gledamo limes omjera potrebnog broja kvadratiča i stranice kvadratiča. Konkretno, za Minkowski-Bouligand-ovu dimenziju nekog skupa S možemo pisati:

$$d_M(S) = -\lim_{\epsilon \rightarrow 0} \frac{\log N_S(\epsilon)}{\log \epsilon} = \lim_{\epsilon \rightarrow 0} \frac{\log N_S(\epsilon)}{\log \frac{1}{\epsilon}} \quad (11.8)$$

Za korektan izračun ove mjere nije nužno da kvadratiči budu pravilno raspoređeni u rešetki. Umjesto toga, moguće je naprsto koristiti potreban broj kvadratiča (koji mogu biti i zarotirani) i ručno ih rasporediti tako da pokriju čitav fraktal.

Za vježbu se uvjerite da i ovakav izračun fraktalne dimenzije za Kochinu krivulju daje isti broj kao i Hausdorffova dimenzija (naputak: za pokrivanje koristite četiri pravokutnika – dva "normalna" i dva zarotirana).

Dodatak A

Prevodenje programa koji koriste GLUT

U ovom poglavlju osvrnut ćemo se na različite prevodioce i različite operacijske sustave, te dati naputak kako najjednostavnije doći do koda koji se može prevesti i pokrenuti. Cilj ovog poglavlja je prevesti program prikazan u ispisu 1.1 prikazanom u poglavlju 1.

A.1 Operacijski sustav Windows i primjeri u jeziku C++

Prvi korak je nabavka biblioteke *glut*¹ ili biblioteke *freeglut*² (preporučamo ovu posljednju). U oba slučaja potrebno je skinuti ZIP arhivu koja sadrži potrebne zaglavne datoteke, opis biblioteke (*.lib) te samu biblioteku (*.d11).

Potom je potrebno nabaviti prevodioc. Pogledat ćemo slučajeve uporabe alata *Microsoft Visual Studio*, zatim *gcc*-a (skinite i raspakirajte *MinGW – Minimalist GNU for Windows*³), te konačno *Free Borland C++ Compiler*⁴.

A.1.1 Microsoft Visual Studio

TBD.

Priprema

TBD.

Uporaba biblioteke *glut*

TBD.

Uporaba biblioteke *freeglut*

TBD.

A.1.2 Gcc

Priprema

Prepostaviti ćemo da ste skinuli *MinGW* distribuciju, te da ste je raspakirali u direktorij D:\usr\MinGW. Takoder, prepostaviti ćemo da nam je cilj prevesti program *prvi.cpp* za koji smo napravili direktorij D:\primjeri\primjer1. Unutar direktorija *primjer1* napravite poddirektorije *include\GL* i *lib*. Otvorite *Command Prompt* i pozicionirajte se u taj direktorij.

¹<http://www.xmission.com/~nate/glut.html>

²<http://freeglut.sourceforge.net/>

³<http://sourceforge.net/projects/mingw/>

⁴<http://edn.embarcadero.com/article/20633>

D:

```
cd D:\primjeri\primjer1
```

Ako to već niste napravili u varijablama okruženja samog operacijskog sustava, dodajte `gcc` u PATH.

```
SET "PATH=%PATH%;D:\usr\MinGW\bin"
```

Uporaba biblioteke *glut*

Iz ZIP arhive *glut*-a u direktorij `include\GL` iskopirajte `glut.h`, u direktorij `lib` iskopirajte `glut32.lib`, a u direktorij `primjer1` iskopirajte samu biblioteku `glut32.dll`. Trebali biste dobiti sljedeću strukturu direktorija.

```
primjer1
  include
    GL
      glut.h
  lib
    glut32.lib
  glut32.dll
  prvi.cpp
```

Konačno, pokrenite prevođenje programa `prvi.cpp`.

```
gcc -Iinclude -Llib -o prvi.exe prvi.cpp -lglut32 -lopengl32
```

Program ćete potom pokrenuti sljedećim pozivom.

```
prvi.exe
```

Uporaba biblioteke *freeglut*

Iz ZIP arhive biblioteke *freeglut* iskopirati direktorije `include` i `lib` te biblioteku `freeglut.dll` u direktorij `primjer1`. Trebali biste dobiti strukturu direktorija kako je prikazano u nastavku.

```
primjer1
  include
    GL
      freeglut.h
      freeglut_ext.h
      freeglut_std.h
      glut.h
  lib
    freeglut.lib
  freeglut.dll
  prvi.cpp
```

Pokrenite prevođenje programa `prvi.cpp`.

```
gcc -Iinclude -Llib -o prvi.exe prvi.cpp -lfreeglut -lopengl32
```

Program ćete potom pokrenuti sljedećim pozivom.

```
prvi.exe
```

A.1.3 Bcc

Prepostavite da ste skinuli Free Borland C++ Compiler distribuciju, te da ste je raspakirali u direktorij D:\usr\BCC55. Također, prepostavite da nam je cilj prevesti program prvi.cpp za koji smo napravili direktorij D:\primjeri\primjer1. Unutar direktorija primjer1 napravite poddirektorije include\GL i lib. Otvorite Command Prompt i pozicionirajte se u taj direktorij.

```
D:  
cd D:\primjeri\primjer1
```

Ako to već niste napravili u varijablama okruženja samog operacijskog sustava, dodajte bcc32 u PATH.

```
SET "PATH=%PATH%;D:\usr\BCC55\bin"
```

Uporaba biblioteke glut

Iz ZIP arhive glut-a u direktorij include\GL iskopirajte glut.h, u direktorij lib iskopirajte glut32.lib, a u direktorij primjer1 iskopirajte samu biblioteku glut32.dll. Trebali biste dobiti sljedeću strukturu direktorija.

```
primjer1  
  include  
    GL  
      glut.h  
  lib  
    glut32.lib  
    glut32.dll  
  prvi.cpp
```

Opis biblioteke koji dolazi u ZIP arhivi nažalost nije kompatibilan s Borlandovim, pa je datotku glut32.lib potrebno nanovo generirati. Taj posao napraviti ćemo uporabom naredbe **implib** koja dolazi u Borlandovom paketu. Evo što treba napraviti.

```
cd lib  
implib glut32.lib ..\glut32.dll  
cd ..
```

Konačno, pokrenite prevođenje programa prvi.cpp.

```
bcc32 -Iinclude;D:\usr\bcc55\include  
     -Llib;D:\usr\bcc55\lib;D:\usr\bcc55\lib\psdk  
     -tWM glut32.lib prvi.cpp
```

Program ćete potom pokrenuti sljedećim pozivom.

```
prvi.exe
```

Uporaba biblioteke freeglut

Iz ZIP arhive biblioteke freeglut iskopirati direktorije include i lib te biblioteku freeglut.dll u direktorij primjer1. Trebali biste dobiti strukturu direktorija kako je prikazano u nastavku.

```
primjer1
include
GL
  freeglut.h
  freeglut_ext.h
  freeglut_std.h
  glut.h
lib
  freeglut.lib
freeglut.dll
```

Opis biblioteke koji dolazi u ZIP arhivi nažalost nije kompatibilan s Borlandovim, pa je datotku `freeglut.lib` potrebno nanovo generirati. Taj posao napraviti ćemo uporabom naredbe `implib` koja dolazi u Borlandovom paketu. Evo što treba napraviti.

```
cd lib
implib freeglut.lib ..\freeglut.dll
cd ..
```

Konačno, pokrenite prevođenje programa `prvi.cpp`.

```
bcc32 -Iinclude;D:\usr\bcc55\include
-Llib;D:\usr\bcc55\lib freeglut.lib prvi.cpp
```

Program ćete potom pokrenuti sljedećim pozivom.

```
prvi.exe
```

A.2 Operacijski sustav Linux i primjeri u jeziku C++

Sve potrebno prikazat ćemo na primjeru danas relativno popularne distribucije Ubuntu (konkretno, verzija 9.10). Ovdje nećemo niti razmatrati uporabu osnovne biblioteke `glut` jer se u paketnom sustavu proglašena zamjenjenom bibliotekom `freeglut3`. Stoga ćemo sve primjere raditi upravo korištenjem biblioteke `freeglut3`. Nakon instalacije Ubuntu-a trebalo je instalirati 3 paketa:

1. *prevodilac za c++, naredbom*

```
sudo apt-get install g++
```

2. *biblioteku freeglut, naredbom*

```
sudo apt-get install freeglut3
```

3. *zaglavne datoteke biblioteke freeglut, naredbom*

```
sudo apt-get install freeglut3-dev
```

Potom napravite direktorij u koji ćemo smjestiti izvorni kod programa (primjerice `primjer1`), te u njega smjestite izvorni kod. Trebali biste dobiti sljedeću strukturu datoteka:

```
primjer1
  prvi.cpp
```

Sada je dovoljno ući u direktorij `primjer1`, i pokrenuti prevodenje, kako je prikazano u nastavku.

```
cd primjer1  
g++ -o prvi prvi.cpp -lglut
```

Napomena: u izvornom kodu programa uključena je i zaglavna datoteka `windows.h`. Prije prevodenja programa na Linux-u taj je redak potrebno izbrisati. Nakon što prevodenje završi, program ćemo pokrenuti kako je opisano u nastavku.

```
./prvi
```

Tim pozivom prikazat će se novi prozor u kojem će biti nacrtan trokut i tri točke.

A.3 Primjeri u jeziku Java

TBD

A.4 Primjeri u jeziku Python

TBD

Indeks

Homogeni prostor, 30

Pravac, 25

2D, 28

 eksplicitni oblik, 29

 implicitni oblik, 29

 parametarska jednadžba, 28

 segmentni oblik, 29

3D, 30

 implicitni oblik, 30

 parametarski oblik, 30

Homogeni prostor

 2D, 31

 3D, 31

 matrični prikaz, 32

 sjecište dvaju pravaca, 33

 karakteristična matrica, 26, 28

 kroz dvije točke, 27

 odnos točke i pravca, 32

Parametarski oblik jednadžbe, 25

Ravnina, 34

 matrični zapis, 34

 matrični zapis u homogenom prostoru, 36

 odnos točke i ravnine, 36

 parametarski oblik, 34

 zapis pomoću normale, 35

Točka, 23

Vektor, 23