HÁROMDIMENZIÓS GRAFIKA, ANIMÁCIÓ ÉS JÁTÉKFEJLESZTÉS

SZIRMAY-KALOS LÁSZLÓ, ANTAL GYÖRGY, CSONKA FERENC

Tartalomjegyzék

1. Bevezetés **1**
1.1. A modellezés 1
1.2. A képszintézis 3
1.2.1. Mi a fény és hogyan érzékeljük? 4
1.2.2. A képszintézis lépései 5

2. Grafikus hardver és szoftver **7**
2.1. A grafikus hardverek felépítése 7
2.2. A grafikus szoftverek felépítése 10
2.3. Programvezérelt és eseményvezérelt interakció 10
2.3.1. Programvezérelt interakció 10
2.3.2. Eseményvezérelt interakció 11
2.4. Programozás Windows környezetben 12
2.5. A grafikus hardver illesztése és programozása 18
2.5.1. OpenGL 21
2.5.2. GLUT 26
2.5.3. Ablakozó rendszer független OpenGL 31

3. Geometriai modellezés **33**
3.1. Pontok, vektorok és koordinátarendszerek 33
3.1.1. A Descartes-koordinátarendszer 34
3.1.2. Program: Descartes-koordinátákkal definiált vektor 35
3.1.3. Síkbeli polár és térbeli gömbi koordinátarendszer 36
3.1.4. Baricentrikus koordináták 36
3.1.5. Homogén koordináták 38
3.2. Geometriai transzformációk 39
3.2.1. Eltolás 42
3.2.2. Skálázás a koordinátatengely mentén 42
3.2.3. Forgatás a koordinátatengelyek körül 42

3.2.4. Általános tengely körüli forgatás 44
3.2.5. A transzformációk támpontja 45
3.2.6. Az elemi transzformációk homogén koordinátás megadása . . . 46
3.2.7. A középpontos vetítés 47
3.2.8. Koordinátarendszer-váltó transzformációk 50
3.2.9. Transzformáció-láncok 51
3.2.10. Program: transzformációs mátrixok 51
3.2.11. Nemlineáris transzformációk 52
3.3. Görbék 54
3.3.1. A töröttvonal 55
3.3.2. Bézier-görbe 56
3.3.3. B-spline 58
3.3.4. B-spline görbék interpolációs célokra 66
3.3.5. Nem egyenletes racionális B-spline: NURBS 67
3.3.6. A görbék tulajdonságai 69
3.4. Felületek 70
3.4.1. Poligonok 71
3.4.2. Poligon modellezés 75
3.4.3. Felosztott felületek 76
3.4.4. Progresszív hálók 80
3.4.5. Implicit felületek 82
3.4.6. Parametrikus felületek 84
3.4.7. Kihúzott felületek 88
3.4.8. Forgásfelületek 90
3.4.9. Felületillesztés görbékre 90
3.5. Testek 92
3.5.1. Konstruktív tömörtest geometria alapú modellezés 92
3.5.2. Funkcionális reprezentáció 94
3.5.3. Cseppek, puha objektumok és rokonaik 94
3.6. Térfogati modellek 98
3.7. Modellek poligonhálóvá alakítása: tesszelláció 99
3.7.1. Sokszögek háromszögekre bontása 99
3.7.2. Delaunay-háromszögesítés 100
3.7.3. Paraméteres felületek és magasságmezők tesszellációja 101
3.7.4. CSG modellek tesszellációja 104
3.7.5. Funkcionális és térfogati modellek tesszellációja 105
3.7.6. Mérnöki visszafejtés 106

4. Színek és anyagok 109
4.1. A színérzet kialakulása . . . 109
4.2. A színillesztés . . . 110
4.3. A színek definiálása . . . 111
4.4. Színleképzés a háromdimenziós grafikában . . . 113
4.5. A hétköznapi életben előforduló anyagok . . . 114
4.6. Anyagok a háromdimenziós grafikában . . . 115
4.6.1. Fényforrások . . . 115
4.6.2. A kétirányú visszaverődés eloszlási függvény . . . 116
4.7. Spektrális képszintézis . . . 117
4.8. Anyagmodellek . . . 118
4.8.1. Lambert-törvény . . . 118
4.8.2. Ideális visszaverődés . . . 119
4.8.3. Ideális törés . . . 121
4.8.4. A spekuláris visszaverődés Phong-modellje . . . 121
4.8.5. A spekuláris visszaverődés Phong – Blinn modellje . . . 122
4.8.6. Cook – Torrance modell . . . 123
4.8.7. Összetett anyagmodellek . . . 124
4.8.8. Az árnyalási egyenlet egyszerűsített változata . . . 125
4.8.9. Anyagon belüli szóródás . . . 126
4.9. Textúrák . . . 126
4.9.1. Paraméterezés . . . 127
4.9.2. Közvetítő felületek használata . . . 130

5. Virtuális világ 133
5.1. Hierarchikus adatszerkezet . . . 133
5.1.1. A színtérgráf . . . 134
5.1.2. A Java3D színtérgráf . . . 135
5.1.3. A VRML színtérgráf . . . 138
5.1.4. Maya hipergráf . . . 140
5.1.5. CSG-fa . . . 142
5.2. A geometriai primitívek . . . 142
5.2.1. A geometria és a topológia szétválasztása . . . 142
5.2.2. Poligonhálók . . . 143
5.2.3. Parametrikus felületek . . . 147
5.3. Világmodellek fájlokban . . . 147
5.3.1. Formális nyelvek . . . 148
5.3.2. Wavefront OBJ fájlformátum beolvasása . . . 152
5.3.3. A VRML 2.0 fájlformátum beolvasása . . . 158
5.4. Világmodellek felépítése a memóriában . . . 160

6. Sugárkövetés **165**
6.1. Az illuminációs modell egyszerűsítése 166
6.2. A tükör- és törési irányok kiszámítása 169
6.3. Metszéspontszámítás felületekre 171
6.3.1. Háromszögek metszése 171
6.3.2. Implicit felületek metszése 174
6.3.3. Paraméteres felületek metszése 175
6.3.4. Transzformált objektumok metszése 175
6.3.5. CSG modellek metszése 176
6.4. A metszéspontszámítás gyorsítási lehetőségei 177
6.4.1. Befoglaló keretek 178
6.4.2. Az objektumtér szabályos felosztása 178
6.4.3. Az oktális fa 179
6.4.4. A kd-fa 181
6.5. Program: rekurzív sugárkövetés 185

7. Inkrementális képszintézis **193**
7.1. Nézeti csővezeték 195
7.2. Nézeti transzformáció 197
7.3. A perspektív transzformáció 198
7.3.1. Perspektív transzformáció a normalizált nézeti gúlából 200
7.4. Vágás 202
7.4.1. Vágás homogén koordinátákkal 202
7.5. Képernyő transzformáció 205
7.6. A takarási feladat megoldása 205
7.6.1. Triviális hátsólap eldobás 206
7.6.2. Z-buffer algoritmus 206
7.7. Árnyalás 209
7.7.1. Fényforrások 210
7.7.2. Anyagok 211
7.7.3. Árnyalási módok 212
7.8. Program: Egyszerű színtér megjelenítése 215
7.9. Stencil buffer 216
7.10. Átlátszóság 217
7.11. Textúra leképzés 218
7.12. Textúra leképzés az OpenGL-ben 221
7.12.1. Textúra definíció 221
7.12.2. Textúrák és a megvilágítás kombinálása 223
7.12.3. Paraméterezés 223
7.13. A textúrák szűrése 224

7.13.1. Határsáv 227
7.14. Multitextúrázás 227
7.15. Fénytérképek 229
7.16. Bucka leképzés 230
7.17. Környezet leképzés 232
7.18. Árnyékszámítás 232
7.18.1. Síkra vetített árnyékok 233
7.18.2. Árnyéktestek 236
7.18.3. Árnyékszámítás z-buffer segítségével 239
7.19. A 3D grafikus hardver 244
7.19.1. Csúcspont-árnyalók 245
7.19.2. Pixel-árnyalók 246
7.19.3. Magasszintű árnyaló nyelvek 246

8. Globális illumináció 249
8.1. Pont és irányhalmazok 250
8.1.1. A fényerősség alapvető mértékei 251
8.1.2. A fotometria alaptörvénye 252
8.2. A fény–felület kölcsönhatás: az árnyalási egyenlet 253
8.3. Térfogati fényjelenségek 256
8.4. A képszintézis feladat elemei 257
8.4.1. BRDF-modellek 258
8.4.2. Mérőműszerek 262
8.5. Az árnyalási egyenlet megoldása 267
8.6. Monte-Carlo integrálás 271
8.6.1. Kvázi Monte-Carlo módszerek 273
8.6.2. A fontosság szerinti mintavételezés 276
8.7. Az árnyalási egyenlet megoldása véletlen gyűjtősétákkal 278
8.8. Az árnyalási egyenlet megoldása véletlen lövősétákkal 280
8.9. Fontosság szerinti mintavételezés a véletlen bolyongásnál 282
8.9.1. BRDF mintavételezés 283
8.9.2. A fényforrás mintavételezése 286
8.9.3. Orosz rulett 288
8.9.4. BRDF mintavételezés összetett anyagmodellekre 289
8.9.5. Fontosság szerinti mintavételezés színes terekben 290
8.10. Véletlen bolyongási algoritmusok 290
8.10.1. Inverz fényútkövetés 292
8.10.2. Fénykövetés 294
8.10.3. Kétirányú fényútkövetés 295
8.10.4. Metropolis-fénykövetés 298

8.10.5. Foton térkép 303
8.11. A globális illuminációs feladat iterációs megoldása 306
8.11.1. Végeselem-módszer 306
8.11.2. Párhuzamos sugárköteg módszer 309
8.11.3. Perspektív sugárköteg módszer 311
8.11.4. Sugárlövés módszer 311

9. Animáció 313
9.1. Folyamatos mozgatás különböző platformokon 315
9.2. Dupla bufferelés 317
9.3. Valószerű mozgás feltételei 318
9.4. Pozíció–orientáció mátrixok interpolációja 320
9.5. Az orientáció jellemzése kvaternióval 322
9.5.1. Interpoláció kvaterniókkal 329
9.6. A mozgásgörbék megadási lehetőségei 331
9.7. Képlet animáció 333
9.8. Kulcskeret animáció 336
9.8.1. Animációs spline-ok 338
9.9. Pálya animáció 345
9.10. Fizikai animáció 348
9.10.1. Kiterjedt testek haladó mozgása és forgása 350
9.10.2. Merev testek mozgásegyenletei 352
9.10.3. A tehetetlenségi mátrix tulajdonságai 357
9.10.4. Ütközésdetektálás 359
9.10.5. Ütközésválasz 363
9.10.6. A merev testek mozgásegyenleteinek megoldása 367
9.11. A hierarchikus mozgás 371
9.11.1. Program: a primitív ember 373
9.12. Deformációk 378
9.13. Karakteranimáció 379
9.13.1. Előremenő kinematika 381
9.13.2. Inverz kinematika 381
9.13.3. Bőrözés 385
9.14. Mozgáskövető animáció 386
9.15. Valós és virtuális világok keverése 388

10. Számítógépes játékok 393
10.1. A felhasználói beavatkozások kezelése 395
10.1.1. A billentyűzet és az egér kezelése GLUT környezetben 396
10.1.2. A billentyűzet és az egér kezelése Ms-Windows környezetben . 399

10.2. A játékmotor 400
10.2.1. A Camera osztály 400
10.2.2. A GameObject osztály 401
10.2.3. A Member osztály 404
10.2.4. Az Avatar osztály 408
10.2.5. A TexturedObject osztály 409
10.2.6. Plakátok: a Billboard osztály 411
10.2.7. Részecskerendszerek: a ParticleSystem osztály 415
10.2.8. A játékmotor osztály 420
10.3. Az űrharc játék 421
10.3.1. A bolygók 423
10.3.2. Az űr 426
10.3.3. Az űrhajó 427
10.3.4. A fotonrakéta 433
10.3.5. A robbanás 434
10.3.6. Az avatár 436
10.3.7. Az űrhajós játék főosztálya 437
10.4. Hierarchikus szereplők 438
10.5. Mozgó karakterek 442
10.6. Terepek 448
10.7. A hegyivadász játék 452
10.7.1. Az ég 453
10.7.2. A hegyvidék 453
10.7.3. Az ellenségek 454
10.7.4. A lövedék 459
10.7.5. Az avatár 460
10.7.6. A hegyivadász játék főosztálya 461
10.8. A teljesítmény növelése 461
10.8.1. Megjelenítési listák 462
10.8.2. Részletezettségi szintek 463
10.8.3. Láthatatlan részek eldobása 463
10.8.4. Térparticionáló adatstruktúrák 464

11. DirectX **467**
11.1. Program: HelloDirectX alkalmazás 469
11.2. Program: VRML színtér megjelenítése 474
11.3. OpenGL kontra DirectX 477

Előszó

A szemünk az egyik legfontosabb érzékszervünk. Hétköznapi tevékenységeink során túlnyomórészt a szemünkkel követjük környezetünk változásait, és ennek megfelelően döntünk saját cselekedeteinkről. A képek, a film és a televízió ezt a folyamatot kiterjesztették mind térben, mind pedig időben, hiszen segítségükkel olyan dolgokat is érzékelhetünk, amelyek tőlünk távol, vagy a valóságban sokkal korábban zajlottak le. A *számítógépes grafika* még tovább megy ezen az úton, és olyan világokba enged bepillantani, amelyek a valóságban sohasem léteztek. A nem létező, virtuális világokat a matematika nyelvén, számokkal adhatjuk meg. Számítógépünk a számokkal leírt virtuális világmodellt fényképezi le, azaz kiszámítja az ugyancsak számokat tartalmazó képet. A modellben szereplő számokat a kép számaira nagyon sokféleképpen alakíthatjuk át, amely végtelen sokféle lehetőséget ad grafikus rendszerek kialakítására. Ezek közül azokban mozgunk otthonosan, amelyek a mindennapjaink megszokott képeihez hasonlatosakkal kápráztatnak el bennünket, ezért célszerű a grafikus rendszert a természettől ellesett elvek szerint, azok analógiájára megalkotni. Amennyiben modellünk háromdimenziós térben elhelyezkedő tárgyakat tartalmaz, a fényképezés pedig a fénytan (*optika*) alapján működik, akkor *háromdimenziós grafikáról* beszélünk. Az optikai analógia nem feltétlenül jelenti azt, hogy az optika törvényszerűségeit pontosan be is akarjuk tartani, csupán a számunkra legfontosabbakat tartjuk tiszteletben, a többit pedig szükség szerint egyszerűsítjük. A kiszámított kép leggyakrabban a számítógép monitoráról jut a felhasználó szemébe. Különleges alkalmazásokban azonban a képet a felhasználót körülvevő szoba falára, vagy akár a szemüvegének a felületére vetíthetjük úgy, hogy a felhasználó mozgásának megfelelően a képet mindig az új virtuális nézőpontnak megfelelően frissítjük. A szemüveges megoldásban a felhasználó a két szemével kissé eltérő képeket érzékelhet, így tényleges háromdimenziós élményhez juthat.

A valós életben már megszoktuk, hogy a környezetünk nem állandó, hanem szereplői mozognak, tulajdonságaik időben változnak. A virtuális világunk mozgatását *animáció*nak nevezzük. A felhasználó a virtuális világ passzív szemlélőjéből annak részesévé válhat, ha megengedjük, hogy a térben mozogjon, és a tér objektumait átren-

dezze (*interakció*). Az ilyen *virtuális valóság* rendszerek megpróbálják a felhasználóval minél jobban elhitetni, hogy valós környezet veszi körül. Innen már csak egyetlen lépésre vagyunk a *számítógépes játékok*tól, amelyekben a virtuális világ objektumai is figyelemmel kísérik a felhasználó mozgását, és ennek megfelelően alakítják saját viselkedésüket, azaz túlélési stratégiájukat.

Ez a könyv a háromdimenziós számítógépes grafikával, animációval, virtuális valósággal és a számítógépes játékokkal foglalkozik, ismerteti azokat az elméleti alapokat és algoritmusokat, amelyekkel magunk is grafikus, illetve animációs rendszereket hozhatunk létre.

Kinek készült ez a könyv?

Szándékaink szerint minden informatikusnak és leendő informatikusnak, aki maga is szeretne grafikus rendszereket fejleszteni, illetve grafikusoknak és animátoroknak, akik eszközeik lelkébe kívánnak látni. A számítógépes grafika egyszerre tudomány, mérnöki-informatikai szakma és művészet. Nem vettük a bátorságot ahhoz, hogy a grafika művészeti oldalához hozzászóljunk, így a könyv csak a tudományos és technikai elemeket tekinti át. Igyekeztük az elméleti alapokat úgy összefoglalni, hogy a témakörök nagy részének megértéséhez a középiskolai matematika és fizika is elegendő legyen. Kivételek persze vannak, ilyen például a globális illuminációról szóló fejezet, illetve az animáció egyes részei, de reméljük, hogy ezek a részek sem veszik el az Olvasó kedvét a könyvtől. Azt ajánljuk, hogy ha a kedves Olvasónak egy-egy rész első olvasásra nehéznek tűnik, akkor nyugodtan ugorja át, és inkább a példaprogramokat próbálja megérteni. Az elmélethez ráér később is visszatérni.

A könyv szinte minden fontosabb témakörét programokkal demonstráljuk, amelyeket az Olvasó a saját programjaiba is átvehet. A könyvben részleteiben, a CD-n pedig teljességükben megjelenő programok bemutatják az algoritmusok implementálási fortélyait. Másrészt, talán azt is sikerül velük megmutatni, hogy a számítógépes grafika egyáltalán nem olyan nehéz, mint amilyennek talán első pillantásra látszik, hiszen rövidke programokkal valóban csodálatos eredményeket érhetünk el.

A programok készítése során az áttekinthetőségre és az egyszerűségre törekedtünk, nem bonyolítottuk a kódot optimalizálással, hibakezeléssel, sőt helyenként még a memória felszabadításával sem. Így a megoldások biztosan nem optimálisak, és nem is robusztusak, de hitünk szerint könnyen követhetőek.

A programokat C++ nyelven adjuk közre, és általában az OpenGL, a GLU és a GLUT könyvtárakat használjuk fel. Röviden kitérünk még a Windows eseményvezérelt programozási felületének, és a DirectX könyvtárnak az ismertetésére is. Ezek közül csak a C++ nyelv és az alapvető objektum-orientált programozási elvek ismeretét tételezzük fel, a többi könyvtár használatába lépésről-lépésre vezetjük be az Olvasót.

Hogyan készült ez a könyv?

A könyv a BME Irányítástechnikai és Informatika Tanszékén sok éve folyó kutatási és oktatási munkának az egyik eredménye. A könyvben található magyarázatok követhetőségét az informatika és villamosmérnöki karok hallgatóin teszteltük több éven keresztül. A hallgatók türelméért és kitartásáért most is hálásak vagyunk, sikereik megerősítettek bennünket, a kudarcaikból pedig tanultunk és a tanulságok alapján módosítottunk egyes részeken. A kutatási munkát az OTKA (T042735), az IKTA (00159/2002) és a TÉT alapítvány, valamint az Alias|Wavefront és az Intel támogatta.

A könyv szerkesztési munkáit LaTeX szövegszerkesztővel végeztük, amelyhez dr. Horváth Tamás írt segédprogramokat. A vonalas ábrákat a szabadon hozzáférhető TGIF rajzolóprogrammal Sün Cecil készítette el. A borítót Tikos Dóra és Tüske Imre Maya programmal alkotta meg. A címlapon és a könyvben nagyon sok helyen felbukkanó számítógépes sün karakter is a kezük munkáját és a Maya lehetőségeit dicséri. A sün felosztott felület (3.4.3. fejezet), amelyet egy csontvázra húztak rá (9.13. fejezet), és a fotontérképes globális illuminációs algoritmussal fényképeztek le (8.10.5. fejezet). A modell modelljéért Keszthelyi Máriát illeti köszönet. A képeket részben a CD mellékletben is megtalálható programokkal, részben Maya-val és RenderX-szel számítottuk ki. Másrészt felhasználtuk kollegáink és barátaink — Szirmay-Kalos Barnabás (Maya), Marcos Fajardo (Arnold), Alexander Pasko (HyperFun), Henrik Wann Jensen (Mental Ray), Czuczor Gergely, Aszódi Barnabás (3D Studio Max), dr. Csébfalvi Balázs (saját térfogatvizualizációs program), Szécsi László (RenderX), Deák Szabolcs (saját autószimulátor), Szíjártó Gábor és Koloszár József (pixel árnyaló program), Jakab Gábor és Balogh Zoltán (saját játék), Tüske Imre (Mental Ray) és a Blackhole Ltd. — műveit is. A könyv lektora dr. Tamás Péter volt, akinek véleményét és megjegyzéseit felhasználtuk a végső változat kialakításában. A könyvet nagyon sokan átolvasták, és megjegyzéseikkel segítettek a fejezetek csiszolgatásában. Köszönetképpen felsoroljuk a nevüket: dr. Sparing László, Lőrincz József, Polyák Tamás, Czuczor Szabolcs, Benedek Balázs, Lazányi István, Szécsi László és Vass Gergely. A szerzők „magyarszerű” kéziratát Megyeri Zsuzsa igazította ki és fordította az irodalmi magyar nyelvre. Ha ezek után is maradt hiba a könyvben, az csak a szerzők gondatlanságának tulajdonítható.

Ficzek Mária segítségével és tanácsai alapján a kéziratot Jenny (SGI) és Bagira (SUN) Postscript formában állította elő és a színes oldalakat CMYK alapszínekre bontotta, amely alapján a BME Kiadó készítette el a nyomda számára levilágított filmeket.

Miért érdemes elolvasni ezt a könyvet?

Szándékaink szerint az Olvasó, miután végigrágta magát ezen a könyvön, érteni fogja, hogy hogyan készülnek a háromdimenziós grafikák, az animációk és a játékok, ismerni fogja azokat az elveket és szoftver eszközöket, amelyeket ilyen rendszerek készítéséhez felhasználhat.

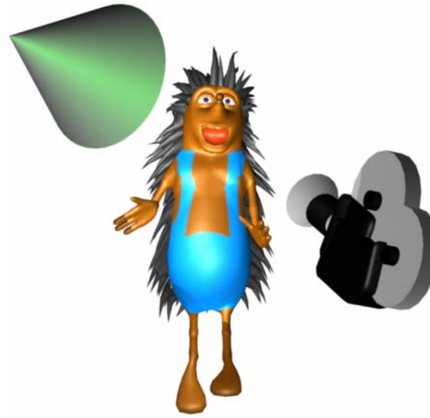
A témakör fontosságát talán csak néhány közismert ténnyel támasztanánk alá. A mai harminc alatti korosztály elsődleges szórakozási formája a számítógépes játék. Az emberek nem azért vesznek két-három évenként új számítógépeket, hogy még gyorsabban tudjanak levelezni, szöveget szerkeszteni, interneten böngészni stb., hanem azért, hogy a legújabb, egyre valószerűbb grafikus játékok is élvezhetőek legyenek. Alig készül olyan mozifilm, amelyben legalább egyes jeleneteket nem számítógépes grafikával hoztak volna létre. Mindamellett a gyártók az új processzorok architektúrájának kialakításánál alapvető szempontnak tartják, hogy a grafikus algoritmusok nagyon gyorsan fussanak rajtuk, és ezért ezeket a műveleteket külön utasításkészlettel valósítják meg (Intel/SSE2, AMD/3Dnow!+).

Ráadásul ezek a tények elhanyagolhatók ahhoz képest, hogy ha az Olvasónak gusztusa támad rá, maga is készíthet grafikus, illetve animációs programokat, amelyek a semmiből új világot teremtenek, sőt akár háromdimenziós játékokat is, amelyekben fantasztikus világokban legyőzhetetlennek tűnő ellenfelek ellen küzdhet, és következmények nélkül veszíthet vagy győzhet. Közülünk valószínűleg kevesen fogják megízlelni az űrutazás élményét, kevesen fognak vadászrepülőt vezetni, és a köztünk megbújó leendő kommandósok, páncélos lovagok és dzsungelharcosok száma is csekély. A számítógépes játékok segítségével azonban egy kis időre bárkiből bármi lehet. Talán még nagyobb bizonyossággal mondhatjuk, hogy senki sem fog a fénysebesség közelében repülni. A számítógépes grafika számára ez sem lehetetlen, csupán a programunkba a relativitáselmélet néhány alapképletét kell beépíteni.

Foglaljuk el a helyünket a számítógépünk előtt! Dőljünk kényelmesen hátra és engedjük el a fantáziánkat, a többi már jön magától. Kellemes olvasást, programozást, izgalmas játékot és virtuális öldöklést mindenkinek!

Budapest, 2003.

a szerzők



1. fejezet

Bevezetés

A *számítógépes grafika* segítségével *virtuális világ*ot teremthetünk, amely létező vagy nem létező tárgyak modelljeit tartalmazza. A világ leírását *modellezés*nek nevezzük. A modellt a *képszintézis* eljárás lefényképezi és az eredményt a számítógép képernyőjén megjeleníti.

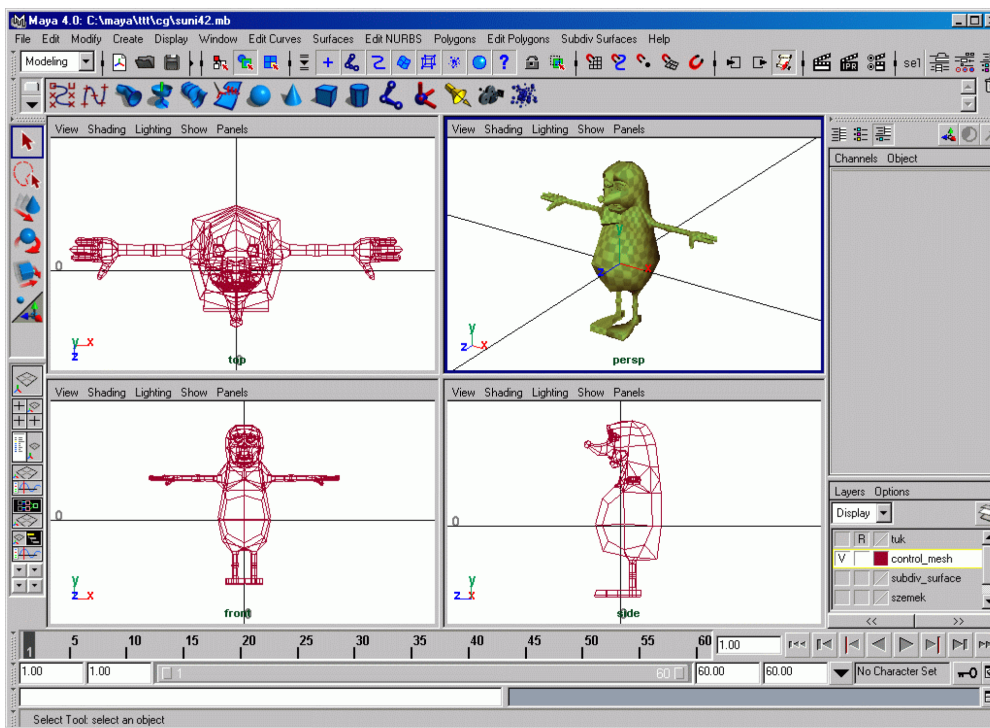
1.1. A modellezés

A *modellezés* során egy virtuális világot írunk le a modellező program eszközeivel. A virtuális világ tartalmazza a tárgyak nagyságát, helyét, alakját, más szóval geometriáját, valamint a megjelenítési tulajdonságaikat, mint például a színt, az átlátszóságot stb. A tárgyakon kívül még fényforrásokat és kamerát is elhelyezünk a virtuális világban, hogy az egy fényképész műterméhez hasonlítson, és hogy a tárgyakat le tudjuk fényképezni. A tárgyak, a fényforrások és a kamera tulajdonságai az időben nem feltétlenül állandók, amit úgy kezelhetünk, hogy a változó tulajdonságokhoz (például a helyhez, nagysághoz stb.) egy-egy időfüggvényt rendelünk. Így minden pillanatban más képet készíthetünk, amelyek a mozgást bemutató képsorozattá, azaz *animáció*vá állnak össze.

A modellezés terméke a *virtuális világ*, amelyet a felhasználó módosíthat és a képszintézis programmal megjeleníthet. A virtuális világot a számítógép számok formájában tárolja. A geometria számokkal történő leírásához egy *világ-koordinátarendszer*t veszünk fel, amelyben az alakzatok pontjainak koordinátáit adjuk meg. Az alakzatok általában végtelen sok pontból állnak, így egyenkénti felsorolásuk lehetetlen. A pontok egyenkénti azonosítása helyett inkább egy szabályrendszert adunk meg, amely alapján eldönthető, hogy egy pont az alakzathoz tartozik-e vagy sem. Dolgozhatunk például matematikai *egyenlet*ekkel, amikor azon pontokat tekintjük egy-egy alakzat részének, amelyek x, y, z Descartes-koordinátái egy-egy adott egyenletet kielégítenek. Például az

$$\left(R - \sqrt{x^2 + y^2}\right)^2 + z^2 = r^2$$

egyenletet megoldó pontok egy olyan úszógumi (tórusz) felületét formázzák, amelynél az r sugarú hengert egy R sugarú körben hajlították meg. Mint ahogyan a példából is látható, a bonyolult alakzatok egyenletei kevéssé szemléletesek. Aki egy úszógumiról ábrándozik, ritkán szokott ezzel az egyenlettel álmodni, ezért egy modellezőprogram nem is várhatja el, hogy a felhasználók közvetlenül a tárgyak egyenleteit adják meg. Egy kényelmesen használható modellező program felhasználói felületén a tervező a virtuális világot szemléletes, interaktív műveletek sorozatával építi fel, amiből a matematikai egyenleteket a program maga határozza meg (az 1.1. ábra a *Maya*[1] felhasználói felületét mutatja be).



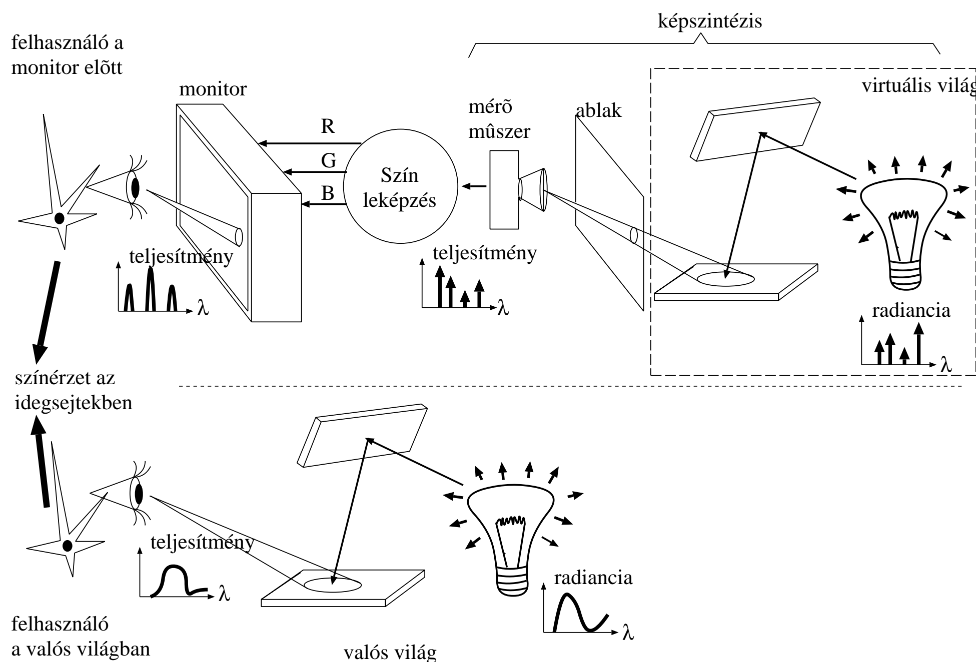
1.1. ábra. *Egy modellezőprogram (Maya) felhasználói felülete*

A műveletsorozat alkalmazása azt jelenti, hogy a virtuális világ sok állapoton keresztül éri el a végső formáját. Az interaktív modellezőprogram a modell aktuális állapotáról alkotható képeket több nézetben mutatja, a képen pedig a felhasználó pontokat, görbéket, felületeket, vagy akár testeket választhat ki, és azokat egyenként módosíthatja.

[1] A Maya modellezőprogram tanulóváltozata a www.aliaswavefront.com oldalról letölthető, az ismerkedéshez pedig a [10, 80] könyveket ajánljuk

1.2. A képszintézis

A *képszintézis* (*rendering* vagy *image synthesis*) a virtuális világot „lefényképezi” és az eredményt a számítógép képernyőjén megjeleníti annak érdekében, hogy a számítógép előtt ülő felhasználóban a valóság szemlélésének illúzióját keltse (1.2. ábra). A képet a *virtuális világ* alapján, egy fényképezési folyamatot szimuláló számítási eljárás segítségével kapjuk meg. A fényképezés során többféle „látásmódot” követhetünk. Az egyik legkézenfekvőbb módszer az optika törvényszerűségeinek szimulálása. A keletkező képek annyira fognak hasonlítani a valódi fényképekre, amennyire a szimuláció során betartottuk a fizikai törvényeket.



1.2. ábra. *A képszintézis célja a valós világ illúziójának keltése*

A kép akkor lesz teljesen valószerű, ha a számítógép monitora által keltett színérzet a valós világéval azonos. Az emberi szem színérzékelése a beérkező fényenergiától és a szem működésétől függ. A fényenergiát a látható pontok fényessége határozza meg, amely a virtuális világ objektumainak geometriája, optikai tulajdonságai és a fényforrások alapján számítható ki. Ezen bonyolult jelenség megértéséhez mind a fény fizikájával, mind pedig a szem működésével meg kell ismerkednünk.

1.2.1. Mi a fény és hogyan érzékeljük?

A „mi a fény?” kérdésre a tudomány eddig több részleges választ adott. Az egyes válaszok modelleket jelentenek, amelyekkel a fénynek csak bizonyos tulajdonságai magyarázhatóak. Az egyik modell szerint a fény elektromágneses hullámjelenség, amelyben az elektromos és mágneses tér egymást pumpálva hullámzik. Emlékszünk ugye az indukcióra? Ha a mágneses tér megváltozik, akkor elektromos tér jön létre (dinamó), illetve, ha az elektromos tér változik, akkor mágneses tér keletkezik (elektromágnes). A fényben a változó elektromos tér a mágneses teret is módosítja, ami visszahat az elektromos tér változására. Ennek a körforgásnak köszönhetjük azt a folyamatos lüktetést, amit hullámnak nevezünk. A hullámokat a maximális magasságukkal (*amplitúdó*), és a hullámcsúcsok távolságával (*hullámhossz*), illetve a hullámhossz reciprokával (*frekvencia*) jellemezzük. A hullámok energiát továbbítanak, amelyet más objektumoknak átadhatnak. Ezt az energiát érezzük, amikor a tavon úszó hajónkat a hullámok ringatják. A környezetünkben előforduló fényforrások nem csupán egyetlen hullámhosszon bocsátanak ki fényt, hanem egyszerre nagyon sok hullámhosszon, azaz a fény általában különböző hullámhosszú hullámok keveréke. Az emberi szem a 300-800 nm hullámhosszú tartományba eső hullámokat képes érzékelni, ezért az ilyen elektromágneses hullámokat nevezzük fénynek.

Egy másik modell szerint a fény „részecskékből”, úgynevezett *foton*okból áll. Egy foton $\hbar/\lambda$ energiát szállít, ahol $\hbar$ a *Planck-állandó* ($\hbar = 6.6 \cdot 10^{-34}$ Joule másodperc), λ pedig a fény hullámhossza. A fotont mint kis golyót képzelhetjük el, amely a felületekkel ütközhet, azokról visszaverődhet, illetve elnyelődhet. Elnyelődéskor a foton energiáját átadja az eltalált testnek.

A *fény*nek az emberi érzékekre gyakorolt hatása a *szín*. Az emberi szemben különböző érzékelők találhatók, amelyek más és más hullámhossz tartományokban képesek a fényt elnyelni, és annak energiáját az idegpályák jeleivé átalakítani. Így a színérzetet az határozza meg, hogy a látható fény milyen hullámhosszokon, mekkora energiát szállít a szembe. Az energia hullámhosszfüggvényét *spektrum*nak nevezzük. A szem a beérkező energiát három, részben átlapolódó sávban képes mérni. Ennek következtében a monitoron nem szükséges (és nem is lehetséges) a számított spektrumot tökéletesen reprodukálni, csupán olyat kell találni, amely a szemben ugyanolyan, vagy legalábbis hasonló színérzetet ad. Ez a *színleképzés* (*tone mapping*).

A számítógépes grafika a fizika törvényeit szimulálja úgy, hogy eközben az emberi szem tulajdonságait is figyelembe veszi. A fizikai törvények alapján ki kell számítani, hogy a különböző felületi pontokból a különböző irányokba milyen spektrumú fény lép ki. Az emberi szem korlátozott képességeinek köszönhetően a számítások során jelentős elhanyagolásokat tehetünk. Az elhanyagolásokra, egyszerűsítésekre annál is inkább szükségünk van, mert a bonyolult fizikai feladat megoldásához roppant kevés idő áll rendelkezésre.

1.2.2. A képszintézis lépései

A képszintézishez a fény által szállított energiát kell kiszámítanunk legalább három hullámhosszon, amely a monitor miatt általában a vörös, a zöld és a kék színnek felel meg. Tekintsünk egy felületet elhagyó fénysugarat. A fénysugár erősségét a *sugársűrűség*gel (*radiancia*) jellemezzük és általában L-lel jelöljük. A sugársűrűség arányos a fénysugár által szállított energiával, azaz a szállított fotonok számával, illetve az elektromágneses hullámzás intenzitásával. A tapasztalat azt mutatja, hogy levegőben vagy légüres térben a sugársűrűség két felület között nem változik. Az Olvasó ezen könyv fehér lapját éppen olyan fehérnek látja, ha a könyvet a szeméhez közelebb emeli vagy távolabb tartja. A közelünkben lévő papírlapról, falról, tárgyakról visszavert fény intenzitása látszólag nem változik a távolsággal, a távoli, pontszerű objektumoké viszont a távolsággal csökken, hiszen a távolabbi csillagok fényét is egyre halványabbnak látjuk. A közeli és kiterjedt, illetve a távoli és pontszerű fényforrások eltérő viselkedésének magyarázata a következő: a fizikai törvények alapján egy pontszerű test által sugárzott energia sűrűsége a távolság négyzetével arányosan csökken, mivel a kisugárzott energia egyre nagyobb felületen oszlik szét. Ha azonban egy közeli, kiterjedt tárgyra tekintünk, akkor szemünk egy-egy „mérőműszere” nem egyetlen pont fényét érzékeli, hanem egy kicsiny felületdarab teljes sugárzását. Ezen kicsiny felületdarab mérete viszont a távolság négyzetével arányosan nő. A két hatás, a pontsugárzó távolsággal csökkenő energiasűrűsége, és a pontszerűnek látszó terület mérete kioltja egymást, azaz a sugársűrűség a közeli tárgyakra állandó. Messzi, illetve pontszerű tárgyak esetén az egy mérőműszer által lefedett terület nem nő a távolsággal, így semmi sem tudja kompenzálni a fényerő csökkenését.

A sugársűrűség megváltozik, ha a fotonok ütköznek a felületeken, így pályájuk módosul (köd, fényelnyelő anyagok esetén ütközés nemcsak a felületeken, de a felületek közötti térben is bekövetkezhet). A fénynyaláb fotonjai, a felület anyagával kölcsönhatásba lépve vagy visszaverődnek a felületről, vagy behatolnak a felület határolta testbe.

A testről visszavert fény intenzitása a megvilágítás irányától, a felület állásától, a nézeti iránytól és a felület optikai tulajdonságaitól függ. A felület állását — más szóval orientációját — az adott pontban a felület normálvektorával jellemezzük. A felület optikai tulajdonságait a *kétirányú visszaverődés eloszlási függvény*, röviden *BRDF* (*Bidirectional Reflection Distribution Function*) írja le. A BRDF minden felületi ponthoz — a hullámhossz, a normálvektor, a megvilágítási és a nézeti irányok alapján — megadja a pont visszaverő képességét.

A virtuális világ leírja a felületek geometriáját és az anyagjellemzőket. A virtuális világot fényforrásokkal és egy virtuális kamerával egészítjük ki. A kamera egy általános helyzetű téglalap alakú *ablak*ból, valamint egy *szem*ből áll, és a világnak az ablakon keresztül látható részét fényképezi le. Mivel a fényintenzitás a felületek és a szem között nem változik, a fényképezésnek az ablak egyes pontjain keresztül látható

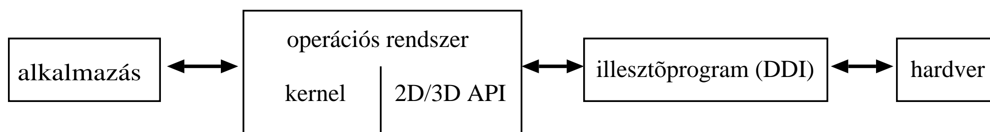
felületi pontokat kell azonosítani, majd a szem irányú sugársűrűséget kiszámítani. Előfordulhat, hogy több objektum is vetíthető az ablak ugyanazon pontjára. Ilyenkor el kell döntenünk, hogy melyiket jelenítsük meg, azaz melyik takarja a többi objektumot az adott pontban (nyilván az, amely a kamerához a lehető legközelebb van). Ezt a lépést *takarás*nak, vagy *takart felület elhagyás*nak (*hidden surface elimination*) nevezzük.

A látható pontban a szem irányába visszavert sugársűrűség számítása az *árnyalás*. A megvilágítási viszonyok ismeretében a BRDF modelleket használhatjuk a számítás elvégzésére. Az árnyalás eredményét a grafikus kártya memóriájába írva megjeleníthetjük a képet.

2. fejezet

Grafikus hardver és szoftver

A háromdimenziós grafikában alkalmazott eljárások, módszerek tárgyalásához tisztában kell lennünk azzal a számítógépes környezettel, amelyben a grafikus alkalmazásaink futnak. A számítógépes környezet szoftver és hardver komponensekből áll.



2.1. ábra. *A számítógépes környezet felépítése*

A 2.1. ábra egy operációs rendszeren futó grafikus alkalmazás környezetét mutatja be. A grafikus programok futtatásához szükség van célhardverekre. A *grafikus hardver*eket az operációs rendszer a hozzá kapcsolódó illesztőprogramok interfészein (DDI: Device Driver Interface) keresztül kezeli.

A hardvereket a modern operációs rendszerek biztonságos és ellenőrzött interfészek mögé „rejtik" el. Számunkra ez azt jelenti, hogy a grafikus alkalmazások közvetlenül nem kezelhetik a grafikus hardvereket, csak az operációs rendszer által biztosított interfészeken, illetve az ezekre épülő könyvtárakon keresztül érhetik el azokat.

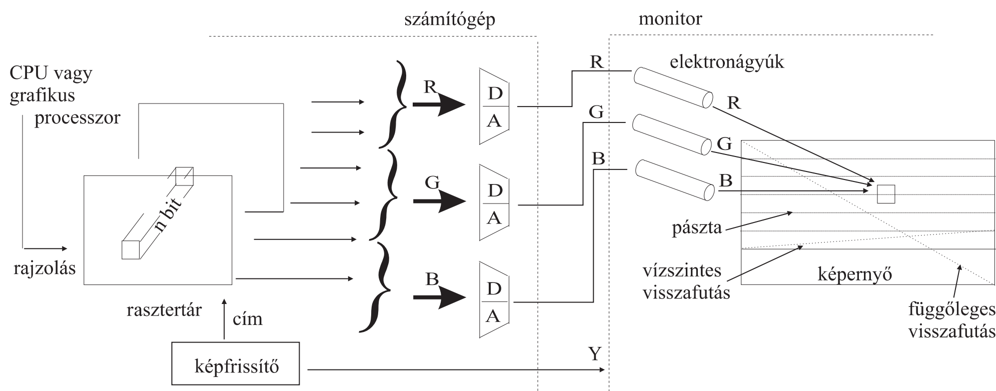
2.1. A grafikus hardverek felépítése

A grafikus megjelenítő eszközöknek két típusa létezik:

A *vektorgrafikus rendszerek* az elektronsugár mozgatásával a képet vonalakból és görbékből építik fel. A módszer előnye, hogy a kép tetszőlegesen nagyítható. Ez a típus a 60-as és 70-es évek elterjedt számítógépes megjelenítő eszköze volt.

A *rasztergrafikus rendszerek*nél a kép szabályos négyzetrácsba szervezett pixelekből áll össze. Maga a *pixel* szó is erre utal, hiszen az a *picture* (kép) és *element* (elem)

angol szavak összeragasztásával keletkezett. Nagy vizuális komplexitású színes képek megjelenítéséhez a módszer ideálisabb, mint a vektorgrafikus megjelenítők. A pixelek színét meghatározó értéket egy speciális memóriába, a *rasztertár*ba kell beírni. A 2.2. ábra egy számítógépből és egy monitorból álló egyszerű rasztergrafikus rendszert mutat be. A megjeleníteni kívánt színinformáció a rasztertárban van, amelyet a *grafikus processzor* ír a rajzolási műveletek (vonalrajzolás, területszínezés stb.) megvalósítása során. A legegyszerűbb rendszerekben a grafikus processzor el is maradhat, ilyenkor a számítógép központi processzora hajtja végre a rajzolási műveleteket és tölti fel a rasztertárat.



2.2. ábra. *Rasztergrafikus rendszerek felépítése (valós szín mód)*

A rasztertár olyan nagy kapacitású, speciális szervezésű memória, amely minden egyes pixel színét egy memóriaszóban tárolja. A szó szélessége (n) a legegyszerűbb rendszerekben 8, grafikus munkaállomásokban 16, 24, sőt 32 vagy 48 bit. A pixel színének kódolására két módszer terjedt el:

1. **Valós szín mód** esetén a szót általában három részre osztjuk, ahol az egyes részek a vörös, zöld és kék színkomponensek színintenzitását jelentik. Ha minden komponenst 8 biten tárolunk, akkor a pixel 24 biten kódolható. Ha ehhez még egy átlátszóságot definiáló úgynevezett *alfa* értéket is hozzáveszünk, akkor egy pixelt 32 bittel adhatunk meg. Ha a rasztertárban egy pixelhez n színintenzitás bit tartozik, akkor valós szín módban a megjeleníthető színek száma 2^n. Például a legjellemzőbb $n = 24$ beállítás esetén 16.7 millió különböző színt tudunk megadni.

2. **Indexelt szín mód** esetén a memóriaszó tartalma valójában egy index a *színpaletta* (*lookup tábla (LUT)*) megfelelő elemére. A tényleges vörös, zöld és kék színintenzitásokat a színpaletta tartalmazza. A módszer előnye a mérték-

letes memóriaigény, hátránya pedig a színpaletta adminisztrációs költsége, a programkomplexitás növekedése, valamint az, hogy az egyszerre megjeleníthető színek száma kevesebb, mint valós szín mód esetén. Ha a rasztertárban egy pixelhez n bit tartozik, akkor indexelt szín módban az egyszerre megjeleníthető színek száma 2^n, de, hogy melyek ezek a színek, az már a paletta tartalmától függ. Ha a palettában egy színt m biten ábrázolunk, akkor a lehetséges színek száma 2^m. Az indexelt szín módnál a képszintézis előtt tudnunk kell, hogy milyen színek bukkannak fel a képen, és a színpalettát ennek megfelelően kell kitölteni. A háromdimenziós grafikában egy tárgy látható színe az optikai tulajdonságainak, a fényforrásoknak, a kamerának, sőt a többi tárgy tulajdonságainak bonyolult függvénye, így a legritkább esetben tudjuk előre megmondani a lehetséges színeket. Ha már ismertek a megjelenítendő színek, ezekből olyan palettát kell készíteni, amellyel jól közelíthető minden pixel színe. Az optimális paletta megtalálása sem egyszerű és gyors algoritmus. A fentiekből kifolyólag a háromdimenziós grafikában elsősorban a valós szín módot alkalmazzák.

A színinformációt a videokártya memóriájából a képernyőre kell varázsolni. Két eltérő elven működő rasztergrafikus megjelenítővel találkozunk a számítógépes grafikában: a *katódsugárcsöves monitor*ral (röviden CRT, az angol *Catode Ray Tube* kifejezés rövidítése nyomán) és a vékonyfilm tranzisztorra (TFT, a *Thin Film Transistors* után) épülő *folyadékkristályos képernyő*vel (LCD, az angol *Liquid Crystal Display* rövidítése).

A 2.2. ábrán a rasztertár tartalmát egy katódsugárcsöves monitor jeleníti meg. A katódsugárcsöves monitorok képének stabilizálásához a rasztertár tartalmát rendszeresen (legalább másodpercenként 50-100-szor) ki kell olvasni, és a képernyőre a képet újra fel kell rajzolni. A kirajzolás során 3 elektronsugárral[1] végigpásztázzuk a képernyő felületét. Az elektronsugarak intenzitását a rasztertár tartalmával moduláljuk. A pixelek egymás utáni kiolvasását a *képfrissítő egység* vezérli, amely szinkronizációs jeleket biztosít a monitor számára annak érdekében, hogy az elektronsugár a pixelsor végén fusson vissza a képernyő bal oldalára. A katódsugárcsöves monitorok számára a digitális színinformációt három D/A átalakító analóg jellé alakítja.

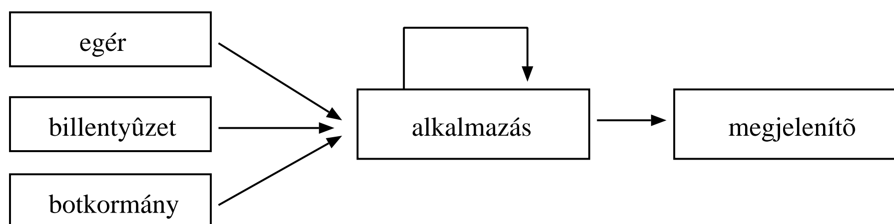
A folyadékkristályos monitorok másképp működnek, itt az elektronsugár visszafutásának lépései hiányoznak. Az LCD megjelenítőknél a VGA (*Video Graphics Array*) interfész mellett lehetőség van DVI (*Digital Visual Interface*) csatlakozásra is, azaz a képinformáció mindvégig digitális marad és a minőségét nem rontják a digitális-analóg átalakítások.

A grafikus rendszer *felbontás*át a pixel sorok és oszlopok száma definiálja. Egy olcsóbb rendszerben a tipikus felbontás 800×600 vagy 1024×768, a professzionális grafika pedig 1280×1024, 1600×1200 vagy még nagyobb felbontást használ.

[1] a vörös(R), zöld(G) és kék(B) színkomponenseknek megfelelően

2.2. A grafikus szoftverek felépítése

A 2.3. ábra egy interaktív program struktúráját mutatja be.



2.3. ábra. *A grafikus szoftver felépítése*

A felhasználó a *grafikus beviteli eszköz*ök (billentyűzet, egér, botkormány, fényceruza stb.) segítségével avatkozhat be a program működésébe. A beviteli eszközöket az operációs rendszer illeszti a programhoz. Az eseményekre való reakciók hatása általában a képernyő tartalmának frissítését eredményezi.

2.3. Programvezérelt és eseményvezérelt interakció

A felhasználói beavatkozások kezelésére alapvetően két programozási technikát használhatunk.

2.3.1. Programvezérelt interakció

A *programvezérelt interakció*ban a program tölti be az irányító szerepet, a felhasználó pedig válaszol a feltett kérdésekre. Amikor a számítások során új bemeneti adatra van szükség, a program erről értesítést küld a felhasználónak, majd addig várakozik, amíg választ nem kap a kérdésre. A jól ismert `printf`-`scanf` C függvénypár ennek tipikus megvalósítása. Ebben az esetben a begépelt karakterek értelmezéséhez szükséges állapotinformációt (például a „347” karaktersorozat valakinek a neve, személyi száma, vagy fizetése) az határozza meg, hogy pontosan hol tartunk a program végrehajtásában. A programvezérelt interakció alapvető hiányosságai:

- Egyszerre csak egy beviteli eszköz kezelésére képes: ha ugyanis az alkalmazás felteszi a kérdését a felhasználónak, akkor addig a program nem lép tovább, amíg a `scanf` függvény vissza nem tér a kérdésre adott válasszal, így ezalatt rá sem nézhet a többi beviteli eszközre.
- Nincsenek globális felhasználói felületet kezelő rutinok, ezért nagyon nehéz szép és igényes felhasználói felületet készíteni.

- A felhasználói kommunikáció és a program feldolgozó része nem válik el egymástól, amíg a program felhasználói bevitelre vár, addig semmi más számítást (például animációt) nem futtathat.

2.3.2. Eseményvezérelt interakció

Az *eseményvezérelt interakció*ban a felhasználó tölti be az irányító szerepet, az alkalmazás pedig passzívan reagál a felhasználói beavatkozásokra. A program nem vár egyetlen eszközre sem, hanem periodikusan teszteli, hogy van-e feldolgozásra váró esemény. Az eseményeket reprezentáló adatok egy *eseménysor*ba kerülnek, ahonnan a program a beérkezési sorrendben kiolvassa és feldolgozza azokat.

A beviteli eszközök (egér, billentyűzet) működtetése megszakítást (*interrupt*) eredményez. A megszakítást kezelő rutin az esemény adatot az eseménysorban tárolja. Eseményt nemcsak a beviteli eszközök, hanem az operációs rendszer és az alkalmazások is kiválthatnak. Az eseményvezérelt programok magja tehát az eseménysor feldolgozása, azaz az *üzenethurok*, amelynek szerkezete általában a következő:

```
while (message != ExitMessage) {          // amíg az üzenet nem a kilépés
    GetMessageFromMessageQueue(&message); // üzenet lekérése
    Process(message);                     // üzenet feldolgozása
}
```

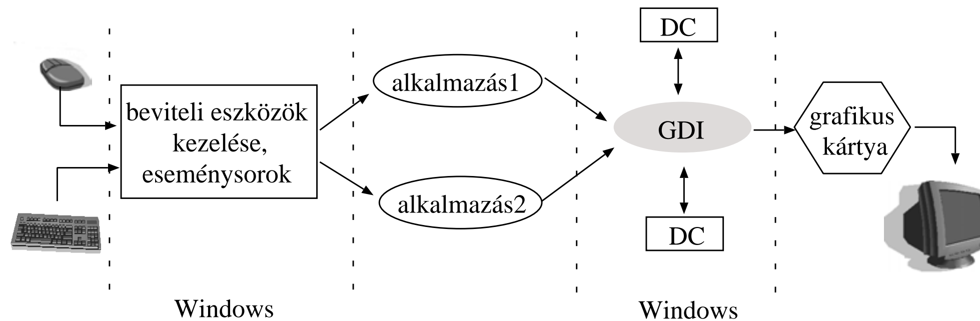
Az eseményvezérelt program felhasználója minden pillanatban szabadon választhat, hogy melyik beviteli eszközt használja. A reakció az eseménytől és a program állapotától is függhet, hiszen például egy egér kattintás esemény egy nyomógomb lenyomását, egy rádiógomb bekapcsolását vagy az ablak bezárását is jelentheti. Az események értelmezéséhez szükséges állapotinformációt ezért explicit módon, változókban kell tárolni.

Vegyük észre, hogy az eszközök tesztelése és az eseménysor kezelése, sőt bizonyos alapvető eseményekre elvárt reakció (például az egér mozgatásakor az egérkurzort is mozgatni kell) az alkalmazástól független, ezért azt csak egyszer kell megvalósítani és egy könyvtárban elérhetővé tenni. A grafikus alkalmazás tehát már csak az egyes eseményekre reagáló rutinok gyűjteménye. Ez egyrészt előnyös, mert a fejlesztőt megkíméljük az interakció alapvető algoritmusainak a megírásától.[2] Másrészt viszont felborul az a jól megszokott világképünk, hogy a program egy jól meghatározott, átlátható, lineáris szálon fut végig, és még azt sem mindig mondhatjuk meg, hogy a program az egyes részeket milyen sorrendben hajtja végre. Eseményvezérelt programot tehát nehezebb írni, mint programvezéreltet.

[2] például karakterek leütése esetén a szöveg megjelenítése, vagy egy nyomógomb lenyomáskor a lenyomott állapotnak megfelelő kép kirajzolása

2.4. Programozás Windows környezetben

Felhasználói szemszögből a *Windows* operációs rendszer az asztalon heverő könyvek metaforáját használja. Egy könyvet ki lehet nyitni, illetve be lehet csukni, ha tartalma többé már nem érdekes számunkra. A könyveket egymásra helyezhetjük, amelyek így részlegesen vagy teljesen eltakarják az alattuk lévőket. Mindig a legfelül lévő könyvet olvassuk.



2.4. ábra. *Ablakozó felhasználói felület*

Windows operációs rendszer alatt az asztalon (*Desktop*) alkalmazások futnak (2.4. ábra). Az alkalmazások ablakkal rendelkeznek, amelyek részlegesen vagy teljesen takarhatják egymást. Minden időpillanatban létezik egy kitüntetett, *aktív alkalmazás*, amely a többi program ablaka előtt helyezkedik el. A felhasználói események ennek az ablaknak az eseménysorába kerülnek.

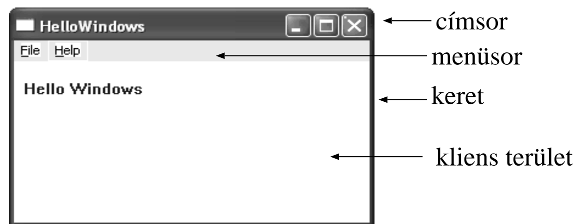
Az alkalmazások a számítógép erőforrásait (képernyő, memória, processzor, merevlemez) megosztják egymás között. Az ablakokat az Ms-Windows *GDI* (*Graphics Device Interface*) alegysége jeleníti meg. Az operációs rendszer minden ablakhoz hozzárendel egy *DC* (*Device Context*) *eszköz kontextus*t, amely a *rajzolási attribútum*okat (például rajzolás színe, vonal vastagsága, betű stílusa stb.) tartalmazza.

Ms-Windows alkalmazások készítésére[3] számos programozási nyelv használható: Visual Basic, Pascal, Delphi, Java, C, C++ [85], mostanában pedig még a C#[4] is. A választásunkat megkönnyíti az a tény, hogy a grafika, különösen a háromdimenziós grafika gyors programokat kíván. Eléggé bosszantónak találnánk, ha kedvenc játékunkban a fejünket egy gránát azért robbantaná szét apró darabokra, mert a program túl lassan reagált arra a billentyűzet eseményre, amellyel fedezékbe ugrottunk. Sebességkritikus programok fejlesztéséhez a rendszerközeli C, illetve a C++ program-

[3]a Windows operációs rendszer Visual Studioval történő programozásához a magyar nyelvű [143] könyvet ajánljuk

[4]kiejtése: szí sárp, jelentése pedig a cisz zenei hang

nyelv ajánlható.



2.5. ábra. *A Windows alkalmazás felülete*

Ebben a fejezetben egy egyszerű HelloWindows alkalmazást fogunk készíteni, amely csak arra képes, hogy kiírja a kliens területre a „Hello Windows” üzenetet (2.5. ábra). Ez lesz az alapja a továbbiakban az OpenGL, GLUT és DirectX alkalmazásoknak. Legegyszerűbben a Visual Studio fejlesztőeszköz varázslójával készíthetünk Win32 projektet. Egy `hpp` (fejléc, *header*), egy `cpp` (program) és egy `rc` (erőforrás, *resource*) fájlt kell létrehozni. Az *erőforrás fájl* tartalmazza a program által használt menük, ikonok, egérkurzorok és sztringek leírását.

Kezdjünk egy kis terminológiával! A 2.5. ábrán egy Windows alkalmazás felülete látható. Az ablak *címsor*ral kezdődik, amely az alkalmazás nevét mutatja. A *menüsor* szintén az ablak tetején, míg az *állapotsor* (*status bar*) általában az ablak alján található. A HelloWindows alkalmazásunk nem tartalmaz állapotsort. A *kliens terület* az ablakkereten belüli maradék rész. Az alkalmazás 2D és 3D rajzolófüggvényei általában erre a területre vannak hatással, ide lehet vonalakat, háromszögeket rajzolni, nyomógombot kitenni, vagy sztringet kiíratni.

Az alkalmazás sohasem foglalkozik közvetlenül a beviteli eszközökkel. A felhasználói interakcióról — például az egér mozgatásáról — az operációs rendszer értesíti az alkalmazást. Tolvajnyelven ezt úgy mondják, hogy a „Windows egy üzenetet küld” az alkalmazásnak. Az üzenet átvételéhez az alkalmazásnak egy speciális függvényt (eseménykezelő) kell megvalósítani. Egér mozgás esetén például ezt a függvényt a Windows a `WM_MOUSEMOVE` paraméterrel hívja meg, míg a bal egérgomb lenyomása esetén a `WM_LBUTTONDOWN` paraméterrel.

Minden Windows alkalmazás belépési pontja a `WinMain()` függvény. A `WinMain()` a konzolos C program `main()` függvényéhez hasonlít: az alkalmazás futtatása a `WinMain()` első utasításával kezdődik. A `WinMain()` szerkezete általában eléggé kötött:

```
//-----------------------------------------------------------------
int WINAPI  WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPTSTR lpCmdLine, int nCmdShow) {
//-----------------------------------------------------------------
    MyRegisterClass(hInstance);             // inicializálás
```

```
    if (!MyCreateInstance(hInstance, nCmdShow)) return FALSE;

    MSG msg;                                  // a fő üzenethurok
    while (GetMessage(&msg, NULL, 0, 0)) {   // amíg van üzenet
        TranslateMessage(&msg);               // billentyű üzenetek átalakítása
        DispatchMessage(&msg);                // üzenet elosztása
    }
    return (int)msg.wParam;
}
```

Az első paraméter (`hInstance`) a HelloWindows program aktuális példányát azonosítja. Ha két HelloWindows alkalmazást indítunk, akkor két különböző példányunk lesz. A második paraméter (`hPrevInstance`) szerepe a 16 bites operációs rendszerek esetén az volt, hogy megadta ugyanabból az alkalmazásból előzőleg elindított alkalmazáspéldányt. A 32 és 64 bites operációs rendszereken azonban ez a paraméter mindig `NULL`, mert itt elviekben minden alkalmazás úgy működik, mintha belőle csak egyetlen példány lenne. A harmadik paraméter a parancssor paramétereit tartalmazza. (Például a „HelloWindows.exe /?” hívás esetén a „/?” sztringet.) Az `nCmdShow` paraméter azt jelenti, hogy az alkalmazás ablakát milyen módon kell megjeleníteni (`SW_SHOWNORMAL`, `SW_SHOWMINIMIZED`, `SW_SHOWMAXIMIZED`, `SW_HIDE`).

A függvények nevében a „My” előtaggal jelezzük (például `MyRegisterClass()`), hogy nem könyvtári, hanem általunk írt függvényről van szó.

Az inicializálást egy pillanatra átugorva tanulmányozzuk az üzenethurok működését! A `GetMessage()` függvény addig vár, amíg egy feldolgozatlan üzenet meg nem jelenik az üzenetsorban, és `WM_QUIT` üzenet esetén hamis, egyébként mindig igaz értékkel tér vissza. A `TranslateMessage()` a billentyűzetről érkező virtuális billentyűkódokat — a SHIFT billentyű állapotát is figyelembe véve — karakterkódokká alakítja (például a #65-ös kódot az ’a’ karakterré), és erről egy új üzenetet helyez el az üzenetsorban. Az ’a’ billentyű lenyomásakor tehát először egy `WM_KEYDOWN` üzenet keletkezik a 65 virtuális billentyűkóddal, majd egy `WM_CHAR` üzenet a 97 (ASCII ’a’) kóddal. Az ASCII kód nélküli billentyűkről (például iránybillentyűk) nem érkezik `WM_CHAR` üzenet, csak `WM_KEYDOWN`. Egy billentyű felengedésekor `WM_KEYUP` üzenet keletkezik. Az üzenethurokban a `DispatchMessage()` függvény küldi el az üzenetet az általunk megadott `WindProc()` függvénynek (lásd később). Az üzenethurok ilyen megvalósítása mellett létezik egy másik — számunkra különösen fontos — módszer is:

```
    while (msg.message != WM_QUIT) {          // amíg nem jön kilépés üzenet
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) { // lekérés
            TranslateMessage(&msg);           // billentyű üzenetek átalakítása
            DispatchMessage(&msg);            // üzenet elosztása
        } else {
            Animate();                        // szabadidőben animáció
            Render();                         // és kirajzolás
        }
    }
```

Említettük, hogy a `GetMessage()` csak akkor tér vissza, ha valamilyen esemény érkezik az üzenetsorba. Valós idejű alkalmazásokban azonban ezt a holt időt is fel kell használni. Az ellenfél katonái ugyanis akkor is mozognak, ha hozzá sem érünk a billentyűzethez. Ilyen esetekben alkalmazható a `PeekMessage()` függvény, amely hamis értékkel tér vissza, ha nincs feldolgozásra váró üzenet. Ebben az esetben hívható például az objektumok mozgatását, majd felrajzolását elvégző `Animate()` és `Render()` függvény, amelyet mi fogunk megvalósítani. A `PM_REMOVE` paraméter azt jelzi, hogy a kiolvasás után az üzenetet az üzenetsorból törölni kell.

Térjünk vissza a Windows alkalmazás inicializálásához (lásd `WinMain()` függvény). Ez két részből áll. Először a `MyRegisterClass()` segítségével egy ablakosztályt regisztrálunk.

```
//-----------------------------------------------------------------
ATOM MyRegisterClass(HINSTANCE hInstance) {
//----------------------------------------------------------------
    WNDCLASSEX wcex;
    wcex.cbSize        = sizeof(WNDCLASSEX);
    wcex.style         = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc   = (WNDPROC)WndProc;        // eseménykezelő függvény
    wcex.cbClsExtra    = 0;
    wcex.cbWndExtra    = 0;
    wcex.hInstance     = hInstance;
    wcex.hIcon         = LoadIcon(hInstance,(LPCTSTR)IDI_HELLOWINDOWS);
    wcex.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground= (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName = (LPCTSTR)IDC_HELLOWINDOWS;
    wcex.lpszClassName= myWindowClass;
    wcex.hIconSm       = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);
    return RegisterClassEx(&wcex);
}
```

Az ablakosztály definiálása a `RegisterClassEx()` függvénnyel történik, amelyhez egy `WNDCLASSEX` struktúrát kell helyesen kitölteni. A megfelelő mezők jelentése a következő:

- `cbSize`: a struktúra mérete. Kötelezően `sizeof(WNDCLASSEX)`.
- `style`: a `CS_HREDRAW | CS_VREDRAW` stílus hatására a horizontális és vertikális mozgatás, illetve átméretezés esetén az ablak tartalma érvénytelen lesz.
- `lpfnWndProc`: az ablak eseménykezelő függvénye. A Windows ezt hívja meg (a `DispatchMessage()` rutinon belül), ha az ablakkal kapcsolatos esemény bekövetkezett.
- `hInstance`: az aktuális alkalmazás példányát azonosító leíró.
- `hIcon`: az ablak ikonját jellemző leíró.
- `hCursor`: az egérkurzor definíciója.
- `hbrBackground`: a háttérkitöltő minta vagy szín.
- `lpszMenuName`: a menü azonosítója az erőforrás fájlban.
- `lpszClassName`: az ablakosztály neve.

- `hIconSm`: az ablak kis ikonját azonosítja.

Az ablakosztály egy példányát a `MyCreateInstance()` rutinban a `CreateWindow()` függvény hívja életre.

```
//-----------------------------------------------------------------
BOOL MyCreateInstance(HINSTANCE hInstance, int nCmdShow) {
//-----------------------------------------------------------------
   HWND hWnd = CreateWindow(myWindowClassName,  // az ablak típus neve
                         myWindowTitle,      // a címsor
                         WS_OVERLAPPEDWINDOW,// stílus
                         CW_USEDEFAULT, 0,   // kezdőpozíció (x,y)
                         300, 200,           // szélesség, magasság
                         NULL,               // a szülőablak
                         NULL,               // menü
                         hInstance,          // alkalmazás példány
                         NULL);              // üzenet adat
   if (hWnd == NULL) return FALSE;
   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);
   return TRUE;
}
```

Az első paraméter annak az ablakosztálynak a neve, amelyet az előbb definiáltunk, a második pedig a címsorban megjelenő szöveg. A `WS_OVERLAPPEDWINDOW` stílus azt jelenti, hogy olyan ablakot készítünk, amelynek kerete, címsora, valamint a címsorban minimalizáló és maximalizáló nyomógombja van. A kezdőpozíciót `CW_USEDEFAULT` esetén az operációs rendszer a képernyő zsúfoltságát figyelembe véve határozza meg. Ablakunknak nincs szülőablaka, és `NULL` menü megadása esetén az ablakosztály menüje lesz az alapértelmezett. Az inicializálási üzenetben egy olyan adatot is megadhatunk, amelyet az ablak az elkészítése során a `WM_CREATE` üzenetben fog megkapni. Ekkor az ablak még rejtőzködik, amit orvosolhatunk a `ShowWindow()` függvény meghívásával, amely láthatóvá teszi az ablakot. Az ezek után hívott `UpdateWindow()` az ablaknak egy újrarajzolás (`WM_PAINT`) üzenetet küld.

Végül megírjuk a `WndProc()` függvényt, amellyel az alkalmazás az eseményekre fog válaszolni. Tekintsük a következő példát:

```
//-----------------------------------------------------------------
LRESULT CALLBACK WndProc(HWND hWnd,      // ablak azonosítója
                         UINT message,   // üzenet típusa
                         WPARAM wParam,  // üzenet egyik paramétere
                         LPARAM lParam){ // üzenet másik paramétere
//-----------------------------------------------------------------
    PAINTSTRUCT ps;                      // rajzolási attribútumok táblázata
    HDC hdc;
    RECT rect = {0,0,150,50};            // 150x50 pixeles terület az üzenetnek
    switch (message) {                   // az eseménynek megfelelő elágazás
    case WM_COMMAND:                     // menü esemény
        switch (LOWORD(wParam)) {        // menü események feldolgozása
        case IDM_EXIT: DestroyWindow(hWnd); return 0; // kilépés menüpont
```

```
        }
        break;
    case WM_PAINT:  // az ablak tartalma érvénytelen, újrarajzolás
        hdc = BeginPaint(hWnd, &ps);    // rajzolás kezdés
        SetTextColor(hdc, 0x00ff0000);  // kék szín (RGBA)
        DrawText(hdc,"HelloWindows",-1,&rect,DT_LEFT|DT_VCENTER|DT_SINGLELINE);
        EndPaint(hWnd, &ps);            // rajzolás befejezés
        return 0;
    case WM_KEYDOWN:                    // billentyűzet események
        if ((int)wParam == VK_RIGHT) {  // jobb iránybillentyű
           MessageBox(hWnd,"A jobb billentyűt lenyomták.","Info",MB_OK);
           return 0;
        }
        break;
    case WM_CHAR:                       // ASCII billentyűzet események
        if ((int)wParam == 'a') {       // 'a' karakter leütése
            MessageBox(hWnd,"Az 'a' billentyűt lenyomták.",Info",MB_OK);
            return DefWindowProc(hWnd, message, wParam, lParam);
        }
        break;
    case WM_DESTROY:
        PostQuitMessage(0);  // WM_QUIT üzenet küldése az üzenetsorba
        return 0;
    }
    return DefWindowProc(hWnd,message,wParam,lParam); //alapértelmezett kezelő
}
```

A `WndProc()` függvény első paramétere az ablakpéldány azonosítója. Ezt követi az üzenet kódja (*message*) és két paramétere (*wParam*, *lParam*). Mindenfajta üzenet paraméterének kódolhatónak kell lenni ebben a két változóban. Az üzenetek nevében a „`WM_`” prepozíció a *Windows Message* angol szavakból származik.

Az eseménykezelő függvény egy `switch-case` struktúra, amely az üzenet típusa szerint ágazik el. Ha egy üzenettel nem szeretnénk foglalkozni, a `DefWindowProc()` függvénnyel meghívhatjuk az operációs rendszer alapértelmezett eseménykezelőjét. Így az olyan feladatokat, mint például az ablak egérrel történő mozgatása, automatikusan elvégeztethetjük. A `DefWindowProc()` függvény — mint ahogy a példánkból látszik — természetesen akkor is meghívható, ha az eseményt már feldolgoztuk. A leggyakoribb eseménykódok a következők:

- `WM_COMMAND`: menüesemény történt.
- `WM_PAINT`: az ablak egy részének tartalma érvénytelen, újra kell rajzolni.
- `WM_KEYDOWN`: egy billentyűt leütöttek.
- `WM_KEYUP`: egy billentyűt felengedtek.
- `WM_CHAR`: ASCII karakter billentyű leütése történt.
- `WM_MOUSEMOVE`: az egér mozog az ablakban.
- `WM_LBUTTONDOWN`: a bal egérgombbal kattintottak.
- `WM_LBUTTONUP`: a bal egérgombot felengedték.
- `WM_MBUTTONDOWN`: a középső egérgombbal kattintottak.
- `WM_MBUTTONUP`: a középső egérgombot felengedték.

- `WM_RBUTTONDOWN`: a jobb egérgombbal kattintottak.
- `WM_RBUTTONUP`: a jobb egérgombot felengedték.
- `WM_DESTROY`: kérés az alkalmazás befejezésére.
- `WM_QUIT`: kilépés üzenet.

A `WM_QUIT` üzenettel a `WndProc()` eseménykezelőben nem fogunk találkozni, mivel az üzenethurok `GetMessage()` függvénye `WM_QUIT` esetén hamis értékkel tér vissza. Ez pedig a `DispatchMessage()` meghívása helyett az üzenethurokból való kilépést jelenti.

A `MessageBox()` függvénnyel egy üzenetablakot hozhatunk létre. Az `MB_OK` paraméter azt jelenti, hogy az üzenetablakban az üzenet, és a fejléc szövege mellett egy `OK` nyomógomb is megjelenik.

Reméljük a HelloWindows alkalmazással sikerült betekintést nyújtani a Windows programozásába. Minden Windows alkalmazás, még a bonyolult programok is, a fent ismertetett elveken alapulnak. Egy komolyabb alkalmazás megírása azonban rengeteg időt emészthet fel. A fejlesztés megkönnyítésére használható például az *MFC* (*Microsoft Foundation Classes*), az *ATL* (*Active Template Library*), a *COM* (*Common Object Model*) és a *.NET* keretrendszer. A könyvtárak használatának elsajátításához szükséges idő ugyan aránylag nagy, azonban hosszabb távon mindenképpen kifizetődő.

2.5. A grafikus hardver illesztése és programozása

A program a grafikus hardver szolgáltatásait *grafikus könyvtár*ak segítségével érheti el. A grafikus könyvtárak általában hierarchikus rétegeket képeznek és többé-kevésbé szabványosított interfésszel rendelkeznek. A grafikus könyvtárak kialakításakor igyekeznek követni az *eszközfüggetlenség* és a *rajzolási állapot* elveit.

Az *eszközfüggetlenség* (*device independence*) azt jelenti, hogy a műveletek paraméterei nem függnek a hardver jellemzőitől, így az erre a felületre épülő program hordozható lesz. A koordinátákat például a megjelenítőeszköz felbontásától függetlenül, a színt pedig az egy képponthoz tartozó rasztertárbeli bitek számától[5] elvonatkoztatva célszerű megadni.

A *rajzolási állapot* (*rendering state*) használatához az a felismerés vezet, hogy már az olyan egyszerűbb grafikus primitívek rajzolása is, mint a szakasz, igen sok jellemzőtől, úgynevezett *attribútum*tól függhet (például a szakasz színétől, vastagságától, mintázatától, a szaggatási közök sűrűségétől, a szakaszvégek lekerekítésétől stb.). Ezért, ha a primitív összes adatát egyetlen függvényben próbálnánk átadni, akkor a függvények paraméterlistáinak nem lenne se vége, se hossza. A problémát a *rajzolási állapot* koncepció bevezetésével oldhatjuk meg. Ez azt jelenti, hogy a könyvtár az érvényes attribútumokat egy belső táblázatban tartja nyilván. Az attribútumok hatása mindaddig

[5] például 8 bit az indexelt szín módban, 32 bit a valós szín módban

érvényben marad, amíg meg nem változtatjuk azokat. Az attribútumok kezelése a rajzolóparancsoktól elkülönített állapotállító függvényekkel lehetséges.

A programozó munkájának megkönnyítésére számos grafikus könyvtár létezik. Az Ms-Windows Win32 API (*Application Programming Interface*) az ablakot, a menüt és az egeret kezeli. A rajzoláshoz az Ms-Windows környezet *GDI*, *GDI+* és *DirectDraw* könyvtárát használhatjuk. Hasonlóan használhatók az XWindow környezet *Xlib*, *Motif*, *QT*, *GNOME* és *KDE* függvénykönyvtárai. Ezek a csomagok 2D grafika programozásában nyújtanak segítséget.

A 3D grafikai könyvtárak közül az *OpenGL* és a *DirectX* a legnépszerűbbek. Jelentőségük az, hogy ezeken keresztül érhetjük el a grafikus kártyák hardverben implementált szolgáltatásait. Tehát egy OpenGL-t használó program fut egy 3D kártyát nem tartalmazó rendszerben is[6], de grafikus gyorsító kártyával sokkal gyorsabban. Egyszerűsége és könnyen tanulhatósága miatt tárgyaljuk a GLUT könyvtárat is, amely az OpenGL szolgáltatásait platformfüggetlen ablak és eseménykezelő szolgáltatásokkal egészíti ki. A következő alfejezetekben a 3D megjelenítést támogató függvénykönyvtárakkal foglalkozunk. Egy grafikus alkalmazás felépítése általában a következő négy séma (2.6. ábra) egyikére épül:

a, A megjelenítést az OpenGL, a billentyűzet és egér eseményeit az Ms-Windows operációs rendszer kezeli.

b, A megjelenítést az OpenGL, az eseménykezelést az XWindow operációs rendszer végzi.

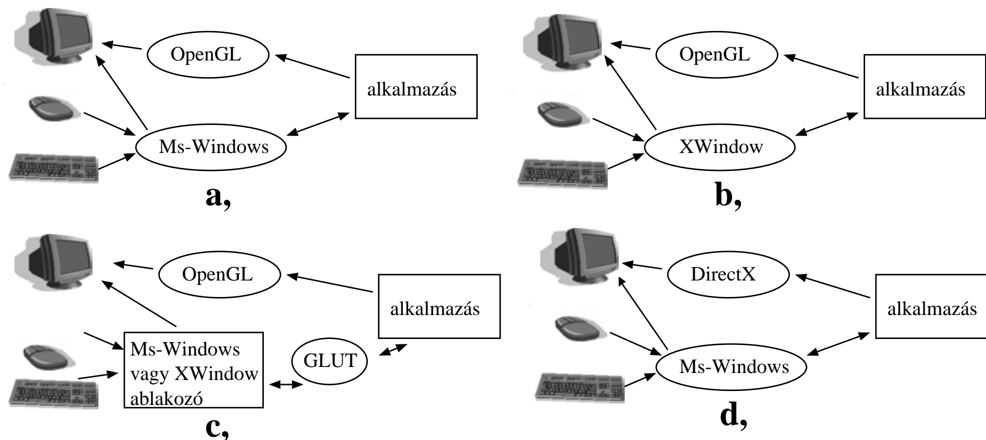
c, A megjelenítést az OpenGL, az eseménykezelési és ablakozó feladatokat pedig egy platformfüggetlen alrendszer, a GLUT valósítja meg. A GLUT valójában szintén Ms-Windows vagy XWindow rutinokat használ, az alkalmazásnak azonban nem kell tudnia erről.

d, DirectX megjelenítés használata esetén csak az Ms-Windows ablakozó és eseménykezelő rendszer használható.

Mielőtt belevágnánk a programozás részleteibe, összefoglaljuk az OpenGL és a DirectX közös jellemzőit. Mindkét rendszerben lehetőség van *dupla bufferelés*re (lásd 9.2. fejezet). Ez a kép villogásának elkerülésére kifejlesztett technika. A takarási feladat megoldására mindkét könyvtár a *z-buffer* (7.6.2. fejezet) módszert használja. Mindkét grafikus könyvtár esetén az attribútumok állapotukat (a megváltoztatásig) megtartják. Ez azt jelenti, hogy teljesen felesleges például a rajzolási színt (ha az statikus) minden egyes képkocka megjelenítésekor újra beállítani.

A megvilágítási viszonyok beállításához lehetőség van absztrakt fényforrások (4.6.1. fejezet) definiálására. Mindkét API irány, pontszerű, szpot valamint ambiens fényforrásokat kezel. OpenGL-ben azonban legfeljebb 8 lámpa definiálására van lehetőség.

[6]ilyenkor a 3D rajzolás szoftveresen történik



2.6. ábra. *Grafikus alkalmazás felépítése*

Különbség van a két API között a koordinátarendszer választásban. Az OpenGL jobbsodrású, a DirectX pedig balsodrású koordinátarendszert használ.

A Windows operációs rendszerben mindkét API megtalálható. Linux operációs rendszeren a DirectX nem elérhető[7]. A Silicon Graphics munkaállomásokon futó IRIX operációs rendszer alatt az OpenGL áll rendelkezésre. Macintosh platformon is lehetőség van az OpenGL használatára, a Mac OS X operációs rendszertől kezdve pedig az operációs rendszer is ezt a könyvtárat használja a rajzoláshoz.

Az OpenGL és a DirectX programozásának bemutatására egy Windows operációs rendszeren futó HelloOpenGL és egy HelloDirectX alkalmazást készítettünk el. Mivel az OpenGL-t Ms-Windows-tól függetlenül, GLUT-tal is lehet programozni, egy HelloGLUT alkalmazással a GLUT API-t is ismertetjük. Ezeket a kedves Olvasó megtalálja a könyvhöz mellékelt CD-n.

Először egy `Application` osztályt definiálunk:

```
//===============================================================
class Application {
//===============================================================
    virtual void    Init(void);   // inicializálás
    virtual void    Exit(void);   // kilépés
    virtual void    Render(void); // ablakozó rendszerfüggetlen rajzolás
};
```

Az `Init()` az alkalmazás indulásakor azonnal lefut és inicializálja a megjelenítő programot. A `Render()` függvény rajzolja ki a képet az ablakba. Végül az alkalmazás

[7]Linux operációs rendszeren az OpenGL programozására az ingyenesen beszerezhető Mesa API-t [8] javasoljuk

leállítása esetén az `Exit()` szabadítja fel a megjelenítő program által használt erőforrásokat.

2.5.1. OpenGL

Az *OpenGL*[8] egy C alapú interfésszel rendelkező 2D és 3D grafikus alkalmazásfejlesztő rendszer. 1992-ben a Silicon Graphics műhelyében GL néven látta meg a napvilágot. A nevében az Open szó a platformfüggetlenségre, a GL pedig a grafikus nyelv (*Graphics Language*) szavakra utal. A csomag manapság már minden fontosabb platformon elérhető. Az OpenGL 1.1 futtatható verziója a Windows XP, Windows 2000, Windows 98, és Windows NT operációs rendszerek része. Napjainkban az 1.4-es verzió a legfrissebb, amelynek fejlesztői verziója ingyenesen elérhető, végfelhasználói verzióját pedig a videokártya gyártók illesztőprogramjaikkal együtt adják. A régóta várt OpenGL 2.0 verzió e könyv írásakor még csak tervezési stádiumban van.

Az OpenGL-hez tartozik egy szabványos segédeszköz csomag, a *GLU* (*OpenGL Utilities*), amely programozást könnyítő hasznos rutinokat tartalmaz, például a transzformációs mátrix beállítására vagy a felületek tesszellálására. A függvények névkonvenciója az OpenGL esetén a `gl` (például `glEnable()`), a *GLU* esetén a *glu* (például `gluPerspective()`) előtag. A függvények utótagja a paraméterek számára és típusára utal. Például egy csúcspont megadása történhet a `glVertex3f()` esetén 3 `float`, a `glVertex3d()` esetén 3 `double`, a `glVertex3i()` esetén pedig 3 `int` értékkel. Hasonló meggondolásokkal egy homogén koordinátás térbeli pont a `glVertex4f()` függvény 4 `float` paraméterével adható meg.

Az OpenGL programozásához a korábban készített HelloWindows alkalmazást fogjuk továbbfejleszteni. Első lépésben fel kell venni a `gl.h` és `glu.h` fejléc (*header*) fájlokat a kódba. A hozzájuk tartozó `OpenGL32.lib` és `glu32.lib`[9] könyvtár fájlokat pedig hozzá kell szerkeszteni a programhoz (*link*).

```
#include <gl/gl.h>
#include <gl/glu.h>
```

Az OpenGL inicializálása a következőképpen történik:

```
//-----------------------------------------------------------------
void Application::Init(void) {
//-----------------------------------------------------------------
    // 1. Windows inicializálás
    MyRegisterClass(hInstance);
    if (!MyCreateInstance(hInstance, nCmdShow)) return;

    // 2. OpenGL inicializálás
    HDC hDC = GetDC(g_hWnd);
```

[8] OpenGL programozásához a [2], [3] és [6] könyveket ajánljuk.

[9] ezek a fájlok általában a gépen vannak, illetve a http://www.opengl.org címről letölthetők

```
    if (!MySetWindowPixelFormat(hDC)) return;
    gGLContext = wglCreateContext(hDC);
    if (gGLContext == NULL) return;
    if (!wglMakeCurrent(hDC, gGLContext)) return;
}
```

Az Ms-Windows inicializálásával a 2.4. fejezetben már foglalkoztunk. Az OpenGL inicializálásához először a `GetDC()`-vel az ablakhoz tartozó *eszköz kontextust* (*Device Context*) kérdezzük le. Egy kontextus a korábban ismertetett *rajzolási állapot* fogalom megvalósítása. A `GetDC()` függvény annak a táblázatnak az azonosítóját adja vissza, amelyben a rajzolási állapot aktuális attribútumai találhatók. Feladatunk az, hogy ennek alapján, a `wglCreateContext()` függvénnyel egy *OpenGL kontextus*t (*OpenGL Rendering Context*) hozzunk létre. Ha ez sikerrel járt, akkor a `wglMakeCurrent()` függvénnyel mondjuk meg, hogy ez legyen az ablak alapértelmezett kontextusa.

Az OpenGL — a színtér megjelenítése során — számos segédtárolóval dolgozik. Ezek a *színbuffer*, a *z-buffer*, az árnyékvetéshez használt *stencil buffer* (7.9. fejezet) és a képek kombinálásához használt *akkumulációs buffer* (*accumulation buffer*). Minden buffer pixel adatokat tartalmaz. Például egy memóriaszó a színbufferben indexelt színmód esetén egy 8 bites indexet, valós színmód esetén egy 24 bites RGB, 32 bites RGBA[10], vagy 16 bites $R_5G_5B_5A_1$ értéket tartalmaz, de a memóriaszó bitjei a színkomponensek között tetszőleges arányban feloszthatók. A felosztást az OpenGL-nek a pixelformátumot leíró struktúra (*PixelFormat*) mondja meg. Mindezek ismeretében a `MySetWindowPixelFormat()` függvény a következőképpen néz ki:

```
//-----------------------------------------------------------------
bool Application::MySetWindowPixelFormat(HDC hDC) {
//-----------------------------------------------------------------
    PIXELFORMATDESCRIPTOR pixelDesc;
    ZeroMemory(&pixelDesc, sizeof(pixelDesc));      // struktúra törlése
    pixelDesc.nSize       = sizeof(pixelDesc);      // a struktúra mérete
    pixelDesc.nVersion    = 1;                      // verziószám
    pixelDesc.dwFlags     = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
            PFD_DOUBLEBUFFER | PFD_STEREO_DONTCARE; // tulajdonságok
    pixelDesc.iPixelType  = PFD_TYPE_RGBA;          // RGBA vagy indexelt mód
    pixelDesc.cColorBits  = 32;         // színbuffer egy szavának mérete
    pixelDesc.cRedBits    = 8;          // vörös komponens mérete
    pixelDesc.cGreenBits  = 8;          // zöld komponens mérete
    pixelDesc.cBlueBits   = 8;          // kék komponens mérete
    pixelDesc.cBlueShift  = 0;          // vörös komponens kezdőbitje
    pixelDesc.cGreenShift = 8;          // zöld komponens kezdőbitje
    pixelDesc.cRedShift   = 16;         // kék komponens kezdőbitje
    pixelDesc.cDepthBits  = 16;         // z-buffer szavának mérete
    pixelDesc.cStencilBits= 0;          // stencil buffer szavának mérete
    pixelDesc.cAccumBits  = 0;          // akkumulációs buffer szavának mérete

    int pixelFormatIndex = ChoosePixelFormat(hDC, &pixelDesc);
    if (!SetPixelFormat(hDC, pixelFormatIndex, &pixelDesc)) return false;
```

[10]Red: vörös; Green: zöld; Blue: kék; Alpha: átlátszóság színkomponens

```
    return true;
}
```

A struktúra `dwFlags` mezőjét úgy állítottuk be, hogy a képet egy ablakba (nem a teljes képernyőre) kérjük (`PFD_DRAW_TO_WINDOW`), ide OpenGL-lel fogunk rajzolni (`PFD_SUPPORT_OPENGL`), dupla bufferelést szeretnénk (`PFD_DOUBLEBUFFER`) és nem sztereó[11] monitoron fog megjelenni a kép (`PFD_STEREO_DONTCARE`). A struktúra feltöltése után meghívott `ChoosePixelFormat()` egy olyan pixel formátum indexet ad vissza, amely az aktuális eszköz kontextusban legjobban hasonlít az általunk megadott pixel formátumra. Az ablak kliens területéhez tartozó tényleges pixel formátum váltást a `SetPixelFormat()` végzi el.

A színtér kirajzolását a `Render()` metódus végzi. Az alábbi példában különböző színű oldalakkal egy egységkockát rajzolunk ki a képernyőre.

```
//-----------------------------------------------------------------
void Application::Render(void) {
//-----------------------------------------------------------------
    HDC hDC = GetDC(g_hWnd);            // eszköz kontextus
    wglMakeCurrent(hDC, gGLContext);    // kontextus aktualizálás

    // 1. lépés: OpenGL állapot-attribútumok beállítása
    glClearColor(0.0, 0.0, 0.9, 0.0);   // törlési szín
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDrawBuffer(GL_BACK);              // a hátsó bufferbe rajzoljon
    glEnable(GL_DEPTH_TEST);            // z-bufferel bekapcsolása
    glEnable(GL_LIGHTING);              // megvilágításszámítás engedélyezése
```

Az OpenGL hívások előtt az aktuális *végrehajtási szál*hoz (*thread*) tartozó OpenGL kontextust a `wglMakeCurrent()` függvénnyel jelöljük ki. Ezután az aktuális rajzolási állapotok már megváltoztathatók. A `glEnable()` függvénnyel bekapcsolunk, a `glDisable()` hívásával pedig kikapcsolunk egy bizonyos megjelenítési attribútumot. A fenti példában a z-buffert és a megvilágítás számítását engedélyezzük. A színbuffert és a z-buffert a `glClear()` függvény törli. A `glClearColor()`-ral adjuk meg a háttérszínt RGBA formátumban.

A következő részben a fényforrást írjuk le:

```
  // 2. lépés: a fényviszonyok beállítása
  float globalAmbient[]={0.2, 0.2, 0.2, 1.0};       // globális ambiens szín
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, globalAmbient);

  float LightAmbient[] = {0.1, 0.1, 0.1, 1.0};      // ambiens fényforrás
  float LightDiffuse[] = {0.5, 0.5, 0.5, 1.0};      // diffúz fényforrás
  float LightPosition[] = {5.0, 5.0, 5.0, 0.0};     // pozíció, de itt irány
  glLightfv(GL_LIGHT0,GL_AMBIENT,LightAmbient);     // ambiens szín
  glLightfv(GL_LIGHT0,GL_DIFFUSE,LightDiffuse);     // diffúz szín
```

[11] a sztereó képszintézis a jobb és a bal szemhez különböző képeket készít, amelyeket egy erre alkalmas szemüveggel nézve térhatású látványt kaphatunk

```
glLightfv(GL_LIGHT0,GL_POSITION,LightPosition);   // pozíció v. irány
glEnable(GL_LIGHT0);
```

Az OpenGL maximum 8 fényforrásából (lásd 4.6.1. fejezet) a példában a 0. fényforrás paramétereit adtuk meg. Ha a megadott pozíciót tartalmazó vektor negyedik koordinátája 0, akkor irányfényforrásról, egyébként pontszerű fényforrásról beszélünk. A globális ambiens színt (lásd 4.6.1. fejezet) szintén megadtuk.

A rajzolás következő lépése a kamera transzformáció beállítása:

```
// 3. lépés: kamera beállítása: projekciós mátrix
RECT rect;
GetClientRect(g_hWnd, &rect);       // ablakméret lekérdezése
float width     = rect.right - rect.left;
float height    = rect.bottom - rect.top;
float aspect    = (height == 0) ? width : width / height;
glViewport(0, 0, width, height);    // a rajzolt kép felbontása
glMatrixMode(GL_PROJECTION);        // projekciós mátrix mód
glLoadIdentity();
gluPerspective(45,                  // 45 fokos látószög,
               aspect,              // magasság-szélesség arány
               1, 100);             // első és hátsó vágósík távolsága

// 4. lépés: kamera beállítása: modell-nézeti mátrix
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(2.0, 3.0, 4.0,            // szem pozíció
          0.0, 0.0, 0.0,            // nézett pont
          0.0, 1.0, 0.0);           // felfele irány
```

A kamera beállítása az OpenGL egy belső állapotának, egy mátrixnak a megváltoztatását jelenti. Az OpenGL három transzformáció típust (`glMatrixMode()`) ismer, a modell–nézeti, a projekciós és a textúra transzformációt. Most persze azt sejtjük, hogy minden transzformációhoz egy-egy mátrix tartozik. Sokat nem is tévedtünk. A helyzet azonban az, hogy nem egy mátrix, hanem egy mátrixokból álló verem tartozik egy transzformációhoz. A verembe `glPushMatrix()`-szal tehetünk be elemeket. A verem tetején keletkező új elem ilyenkor még megegyezik az alatta lévővel. A verem tetejéről a `glPopMatrix()` vesz le egy elemet. A verem kezdetben 1 mátrixot tartalmaz, maximális elemszáma a modell–nézeti transzformációra 32, a projekciós és textúra transzformációra pedig 2. A mátrixműveletek mindig az aktuális transzformáció típushoz tartozó verem tetején található mátrixra vonatkoznak. Például egy `glRotatef()` függvénnyel a legfelső mátrixra még egy tengely körüli forgatást adhatunk meg. A műveletet az OpenGL „hozzáfűzi" az aktuális transzformációhoz. A kezdeti egységtranszformációt a `glLoadIdentity()` függvénnyel állíthatjuk be. Mindig a verem tetején található transzformáció az érvényes.

Végül a színtér objektumait átadjuk az OpenGL-nek:

```
// 5. lépés: színtér felépítése, az objektum létrehozása
```

```
    const float RedSurface[]   = {1, 0, 0, 1};
    const float GreenSurface[] = {0, 1, 0, 1};
    const float BlueSurface[]  = {0, 0, 1, 1};
    float   v[8][3] = {             // a csúcspontok
        { 0.5,  0.5,  0.5}, {-0.5,  0.5,  0.5},
        {-0.5, -0.5,  0.5}, { 0.5, -0.5,  0.5},
        { 0.5,  0.5, -0.5}, {-0.5,  0.5, -0.5},
        {-0.5, -0.5, -0.5}, { 0.5, -0.5, -0.5}};

    glBegin(GL_QUADS);      // rajzolás megkezdése: négyszögek következnek
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, RedSurface);
    glNormal3f(0.0, 0.0, 1.0);              // előlap normálvektora
    glVertex3fv(v[0]); glVertex3fv(v[1]);   // előlap
    glVertex3fv(v[2]); glVertex3fv(v[3]);
    glNormal3f(0.0, 0.0, -1.0);             // hátlap
    glVertex3fv(v[6]); glVertex3fv(v[5]);
    glVertex3fv(v[4]); glVertex3fv(v[7]);
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, GreenSurface);
    glNormal3f(0.0, 1.0, 0.0);              // tetőlap
    glVertex3fv(v[0]); glVertex3fv(v[4]);
    glVertex3fv(v[5]); glVertex3fv(v[1]);
    glNormal3f(0.0, -1.0, 0.0);             // alsólap
    glVertex3fv(v[6]); glVertex3fv(v[7]);
    glVertex3fv(v[3]); glVertex3fv(v[2]);
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, BlueSurface);
    glNormal3f(1.0, 0.0, 0.0);              // jobb oldallap
    glVertex3fv(v[0]); glVertex3fv(v[3]);
    glVertex3fv(v[7]); glVertex3fv(v[4]);
    glNormal3f(-1.0, 0.0, 0.0);             // bal oldallap
    glVertex3fv(v[6]); glVertex3fv(v[2]);
    glVertex3fv(v[1]); glVertex3fv(v[5]);
    glEnd();            // rajzolás befejezése

    SwapBuffers(wglGetCurrentDC()); // a dupla-buffer szerepcseréje
    wglMakeCurrent(NULL, NULL);
}
```

Egy primitív rajzolása a `glBegin()` utasítással kezdődik, és a `glEnd()` utasítással fejeződik be. A `GL_QUADS` paraméterrel azt jelezzük, hogy a csúcspontok négyesével definiálnak egy-egy négyszöget. A használandó anyag a `glMaterialfv()` segítségével állítható be. Egy négyszöget a normálvektorával (`glNormal3f()`) és négy csúcspontjával (`glVertex3fv()`[12]) adhatunk meg.

A kirajzolás végén a `SwapBuffers()` megcseréli az első és a hátsó színbuffer szerepét. Ha nem használunk dupla bufferelést, akkor hívjuk meg a `glFlush()` függvényt, amely megvárja amíg a videokártya a még éppen folyamatban lévő OpenGL utasításokat is végrehajtja. Hibás működés esetén a legutolsó OpenGL utasítás hibakódja a `glGetError()` függvénnyel kapható meg.

Az alkalmazás leállítása során az `Exit()` függvény indul el, amelynek feladata az OpenGL kontextus törlése. A törlés előtt a `wglMakeCurrent(NULL, NULL)`-lal

[12]a függvény 3 karakter hosszú utótagja utal a paraméter típusára, amely most egy 3 elemű float vektor

kérjük, hogy az ablak szüntesse meg az éppen beállított kontextus érvényességét:

```
//-----------------------------------------------------------------
void Application::Exit(void) {
//-----------------------------------------------------------------
    if (wglGetCurrentContext() != NULL) wglMakeCurrent(NULL, NULL);
    if (gGLContext != NULL) wglDeleteContext(gGLContext);
}
```

2.5.2. GLUT

Az OpenGL nem ablakozó rendszer és nem kezeli a felhasználói eseményeket, ehhez az adott operációs rendszer szolgáltatásait kell igénybe venni. Ez általában bonyolult, és nem hordozható. A megoldás a GLUT API[13], amely egy platformfüggetlen ablakozó és eseménykezelő rendszer is [72]. Az akronim az OpenGL Utility Toolkit angol szavakból származik. Legfontosabb funkciója egy nagyon egyszerű ablakozó rendszer és felhasználói felület megvalósítása. A GLUT független az operációs rendszertől (Macintosh, Windows, Linux, SGI), az ablakozó rendszertől (Motif, Gnome, Windows) és a programozási nyelvtől (C, C++, FORTRAN, Ada). Hagyományos Windows vagy Motif alkalmazásokhoz képest a GLUT-ot nagyon egyszerű használni, éppen ezért elsősorban kezdő programozók számára hasznos. Kis méretű, egyszerű programok írására tervezték, és amíg az OpenGL megtanulására ideális eszköz, komolyabb (például gördítősávot, párbeszédablakot, menüsort használó) alkalmazásokhoz már nem alkalmas. A GLUT jellemzői:

- egyszerre több ablak kezelése.
- visszahívó (*callback*) függvény alapú eseménykezelés.
- időzítők (*timer*) és üresjárat (*idle*) kezelés.
- számos előre definiált tömör és drótváz test (például a `glutWireTeapot()` egy teáskanna drótvázát rajzolja).
- többféle betűtípus.

A GLUT programozását nem a HelloWindows példaprogram továbbfejlesztésével, hanem egy üres konzol alkalmazás megírásával kezdjük. A HelloGLUT program teáskannát, illetve kockát jelenít meg.

Első lépésben fel kell venni a `glut.h` fejléc (*header*) fájlt a forráskódba, majd a hozzá tartozó `glut32.lib` könyvtár fájlt hozzá kell szerkeszteni a programhoz (*link*). Ezek általában nincsenek a gépünkön, így ezeket a GLUT programozás megkezdése előtt le kell tölteni, vagy a CD-ről fel kell telepíteni.

Természetesen biztosítani kell azt is, hogy a futás során a `glut32.dll`-t megtalálja az operációs rendszer. A program `main()` függvénye a következőképpen néz ki:

[13] a GLUT hivatalos honlapjáról, a http://www.opengl.org/developers/documentation/glut/index.html címről a fejlesztőeszköz és a dokumentáció is ingyenesen letölthető

```
#include <gl/glut.h>
//-----------------------------------------------------------------
int main(int argc, char* argv[]) {
//-----------------------------------------------------------------
    // inicializálás
    Application application;    // saját applikáció objektumunk
    application.Init(NULL, argc, argv);
    onexit(ExitFunc);           // kilépéskor hívott függvény
    glutMainLoop();             // a fő üzenethurok
    return 0;
}
```

Az üzenethurok a `glutMainLoop()` függvényben található. Valójában soha nem tér vissza, azaz az utána következő „`return 0`” utasításra soha nem fut rá a program. Továbbá, mivel a GLUT-nak nincs az alkalmazás leállásához rendelhető visszahívó függvénye sem, az erőforrások korrekt felszabadítására egyetlen lehetőségünk az ANSI C `onexit()` függvénye. Az ezzel regisztrált `ExitFunc()` függvényt hívja meg a rendszer a program leállása esetén.

Az alkalmazás ablakot az `Init()` függvény készíti el:

```
//-----------------------------------------------------------------
void Application::Init(void) {
//-----------------------------------------------------------------
    glutInit(&argc, argv);                  // GLUT inicializálás
    // 1. lépés: az ablak inicializálása
    glutInitWindowPosition(-1, -1);         // alapértelmezett ablak hely
    glutInitWindowSize(600, 600);           // ablak mérete
                                            // dupla buffer + RGB + z-buffer
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutWindowID = glutCreateWindow("Hello GLUT");  // ablak készítése
    glutSetWindow(glutWindowID);

    // 2. lépés: visszahívó függvények beállítása
    glutDisplayFunc(Render);                // rajzoláshoz
    glutIdleFunc(IdleFunc);                 // semmittevés esetén
    glutMouseFunc(MouseFunc);               // egérgomb lenyomás
    glutMotionFunc(MouseMotionFunc);        // egér mozgatás
    glutKeyboardFunc(KeyboardFunc);         // billentyűzet
    glutSpecialFunc(SpecialKeysFunc);       // nem ASCII billentyűk

    // 3. lépés: legördülő menü készítése
    int submenu = glutCreateMenu(MenuFunc); // almenü visszahívó fg.
    glutAddMenuEntry("Solid Teapot" , SolidPotMenuID);
    glutAddMenuEntry("Wire Teapot"  , WirePotMenuID);
    glutAddMenuEntry("Solid Cube"   , SolidCubeMenuID);
    glutAddMenuEntry("Wire Cube"    , WireCubeMenuID);
    glutCreateMenu(MenuFunc);               // főmenü visszahívó fg.
    glutAddSubMenu("Type", submenu);
    glutAddMenuEntry("Exit", ExitMenuID);
    glutAddMenuEntry("About...", AboutMenuID);
    glutAttachMenu(GLUT_RIGHT_BUTTON);      // jobb kattintásra aktiválódik
}
```

A `glutInit()` inicializálja a GLUT-ot. Az ablak pozícióját és méretét a `glut-InitWindowPosition()` és a `glutInitWindowSize()` függvények adják meg. Ha még emlékszünk az OpenGL inicializálására, akkor észrevehetjük, hogy a `glutInit-DisplayMode()` valójában az ottani PixelFormat regisztrációját végzi el. Használhatunk valós szín módot (`GLUT_RGB`) vagy indexelt szín módot (`GLUT_INDEX`). Bekapcsolható a z-buffer (`GLUT_DEPTH`) vagy a stencil buffer (`GLUT_STENCIL`). Alkalmazhatunk egy vagy két színbuffert (`GLUT_SINGLE`, `GLUT_DOUBLE`). Mivel a GLUT egyszerre több ablak kezelésére is képes, a `glutSetWindow()`-val kell megmondani, hogy éppen melyik ablakkal szeretnénk dolgozni.

Az ablak elkészítése után beállítjuk a visszahívó függvényeket, amelyeket nekünk kell megírni, és amelyeket a fő üzenethurok (`glutMainLoop()`) a megfelelő események bekövetkezése esetén fog meghívni. Paraméterként `NULL`-t adva törölhetjük az adott üzenethez korábban regisztrált függvényt. Az eseménykezelésről a későbbiekben még lesz szó.

A GLUT-ban menüsort nem, csupán az egér valamelyik gombjával előhívható legördülő (*popup*) menüt készíthetünk. A példa menüje egy „Type” almenüből (*subMenu*), egy „Exit” és egy „About” menüpontból áll. Az almenü „Solid Teapot”, „Wire Teapot”, „Solid Cube” és „Wired Cube” menüpontokat tartalmaz. A menüpontokat egy-egy egész számmal azonosítjuk.

```
const short SolidPotMenuID  = 0;    // tömör teáskanna
const short WirePotMenuID   = 1;    // drótváz teáskanna
const short SolidCubeMenuID = 2;    // tömör kocka
const short WireCubeMenuID  = 3;    // drótváz kocka
const short ExitMenuID      = 12;   // kilépés
const short AboutMenuID     = 13;   // program névjegye
```

A legördülő menüt GLUT-ban valamelyik egérgomb kattintásával lehet elővarázsolni. A bal (`GLUT_LEFT_BUTTON`), a középső (`GLUT_MIDDLE_BUTTON`) vagy a jobb (`GLUT_RIGHT_BUTTON`) egérgomb egyikéhez a `glutAttachMenu()` hívással rendelhetünk menüt. A `glutCreateMenu()` segítségével adjuk meg a menükezelő függvényt. Esetünkben ezt a szerepet mind a főmenü, mind az almenü esetén a `MenuFunc()` tölti be, amely paraméterként a menüpont azonosítóját kapja meg:

```
//-----------------------------------------------------------------
void MenuFunc(int menuItemIndex) {
//-----------------------------------------------------------------
    switch (menuItemIndex) {
    case SolidPotMenuID:  gShowedItemType = SolidPotMenuID;  break;
    case WirePotMenuID:   gShowedItemType = WirePotMenuID;   break;
    case SolidCubeMenuID: gShowedItemType = SolidCubeMenuID; break;
    case WireCubeMenuID:  gShowedItemType = WireCubeMenuID;  break;
    case ExitMenuID:      exit(0); // Exit: kilépés
    case AboutMenuID:     MessageBox(NULL,"Hello GLUT.","About",MB_OK); break;
    }
}
```

A gShowedItemType mező a megjelenített objektum típusát jelenti, amely esetünkben teáskanna vagy kocka lehet.

Az onexit()-tel beállított ExitFunc() az Application osztály Exit() metódusát hívja, amely elvégzi az alkalmazás által lefoglalt erőforrások felszabadítását.

```
//-----------------------------------------------------------------
void Application::Exit(void) {
//-----------------------------------------------------------------
    glutDestroyWindow(glutWindowID);
}
```

Az alkalmazás három függvénye közül az Init()-et és az Exit()-et már megadtuk. A Render() függvényben az Ms-Windows-os OpenGL esethez képest csak az ott használt wglMakeCurrent() és SwapBuffers() változik meg a GLUT-os megfelelőre:

```
//-----------------------------------------------------------------
void Render(void) {
//-----------------------------------------------------------------
    glutSetWindow(glutWindowID);    // az ablak aktualizálása
    ...// natív OpenGL hívások (lásd HelloOpenGL program)

    // 4. színtér felépítése
    float GreenSurface[] = {1.0, 0.0, 0.0, 1.0};
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, GreenSurface);
    switch (gShowedItemType) {
    case SolidPotMenuID:    glutSolidTeapot(1.0);   break;
    case WirePotMenuID:     glutWireTeapot(1.0);    break;
    case SolidCubeMenuID:   glutSolidCube(1.0);     break;
    case WireCubeMenuID:    glutWireCube(1.0);      break;
    }
    glutSwapBuffers();
}
```

A GLUT képességeinek bemutatása céljából — a gShowedItemType változó aktuális értékének megfelelően — a Render() függvényben egy tömör vagy egy drótváz teáskannát (glutSolidTeapot(), glutWireTeapot()), illetve egy tömör vagy egy drótváz kockát (glutSolidCube(), glutWireCube()) jelenítünk meg. A buffereket a glutSwapBuffer() cseréli ki, így az eredmény megjelenik a képernyőn.

A színtér megjelenítése ezzel készen is volna. Foglalkozzunk egy keveset a felhasználói interakciók kezelésével! Egy egérgomb lenyomására (ha nem rendeltünk hozzá legördülő menüt) a glutMouseFunc() függvénnyel beállított MouseFunc() eljárást hívja a rendszer:

```
//-----------------------------------------------------------------
void MouseFunc(int button, int state, int x, int y) {
//-----------------------------------------------------------------
    if (button != GLUT_LEFT_BUTTON) return; // csak a bal egérgombra reagálunk
    bool isAltKeyDown   = (glutGetModifiers() == GLUT_ACTIVE_ALT);
    if (state == GLUT_DOWN && isAltKeyDown) {   // ha az ALT is lenyomva
```

```
        lastMousePos.x = x;                 // eltároljuk az egér pozícióját
        lastMousePos.y = y;
        bMouseButtonDown = true;            // eltároljuk a gomb állapotát
    } else
        bMouseButtonDown = false;           // eltároljuk a gomb állapotát
}
```

Példánkban célunk az, hogy az Alt billentyű és a bal egérgomb lenyomására, majd az egér mozgatására a kamerát fel, le, jobbra, balra tudjuk mozgatni. Az Alt, a Shift és a Control billentyű állapota a `glutGetModifiers()` függvénnyel kérdezhető le. Ez a metódus csak a `MouseFunc()`, `KeyboardFunc()`, `SpecialKeysFunc()` visszahívó függvényekből hívható.

Az egér mozgatásakor a `glutMotionFunc()` által beállított `MouseMotionFunc()` függvényt hívja a rendszer:

```
//-----------------------------------------------------------------
void MouseMotionFunc(int x, int y) {
//-----------------------------------------------------------------
    if (bMouseButtonDown) {
        if (x != lastMousePos.x) CameraStrafe();     // oldalirányú mozgás
        if (y != lastMousePos.y) CameraMoveUpDown(); // vertikális mozgás
        glutPostRedisplay();
    }
    lastMousePos.x = x;  lastMousePos.y = y;
}
```

Ha az egérgomb lenyomásakor az egér aktuális és régebbi pozíciója között különbség van, akkor elvégezzük a kamera mozgatását. A `CameraStrafe()` az oldalirányú, a `CameraMoveUpDown()` pedig a vertikális mozgatást valósítja meg. Az ablak újrarajzolását a `glutPostRedisplay()` hívással kérjük. Ennek hatására a GLUT elindítja a `glutDisplayFunc()`-ban megadott függvényünket.

A billentyűzet esemény feldolgozására két visszahívó függvény szolgál. Egyrészt a `glutKeyboardFunc()` segítségével beállított `KeyboardFunc()` az ASCII kóddal rendelkező karaktereket kezeli. Másrészt a `glutSpecialFunc()` paramétereként megadott `SpecialKeysFunc()` a speciális karaktereket dolgozza fel. Ilyenek az F1,...,F12, a fel-le-jobbra-balra irány, a PageUp, PageDown, Home, End és az Insert[14] billentyűk. A Ctrl, Alt, Shift billentyűk lenyomásáról a GLUT nem küld üzenet.

A HelloWindows program billentyűzet kezelése a HelloGLUT példában a következőképpen néz ki:

```
//-----------------------------------------------------------------
void KeyboardFunc(unsigned char asciiCode, int x, int y) {
//-----------------------------------------------------------------
     if (asciiCode == 'a')   //'a' karakter leütése
        MessageBox(NULL, "Az 'a' billentyűt lenyomták.", "Info", MB_OK);
```

[14] a Backslash és a Delete billentyűket a GLUT ASCII karakternek tekinti

```
}

//-----------------------------------------------------------------
void SpecialKeysFunc(int key, int x, int y) {
//-----------------------------------------------------------------
    if (key == GLUT_KEY_RIGHT)  // jobb irány billentyű
        MessageBox(NULL, "A Jobb billentyűt lenyomták.", "Info", MB_OK);
}
```

Egy animáció során a képernyőt rendszeresen újra kell rajzolni. A `glutIdleFunc()` segítségével beállított `IdleFunc()` függvényt a GLUT a szabadidejében folyamatosan hívogatja. Az animációs számítások elvégzése után akár azonnal újrarajzolhatjuk a képernyő tartalmát, azonban lehetőség van késleltetett rajzolásra is. Ilyenkor a `glutPostRedisplay()` hívással helyezünk el egy rajzolási eseményt az üzenetsorban. Ezzel a módszerrel a különböző események (például ablak átméretezés, billentyűzet leütés, animáció) miatt bekövetkező többszörös újrarajzolást spórolhatjuk meg.

```
//-----------------------------------------------------------------
void IdleFunc(void) { // animációhoz szükséges
//-----------------------------------------------------------------
    ... // animációs számítások elvégzése
    glutPostRedisplay(); // újrarajzolás esemény küldése
}
```

2.5.3. Ablakozó rendszer független OpenGL

Könyvünkben a programozási példák során nem szeretnénk azzal foglalkozni, hogy Ms-Windows vagy GLUT környezetben használjuk-e az OpenGL rajzolási rutinokat. Ennek érdekében egy *Application* ősosztályt definiálunk, amely elfedi az API-k közti különbségeket:

```
enum ApplicationType {GlutApplication, WindowsApplication}; // alkalmazás típus
//===============================================================
class Application {
//===============================================================
public:
static  Application*    gApp;                    // globális alkalmazáspéldány
static  void CreateApplication();                // globális alkalmazás készítő

    ApplicationType applicationType;             // ablakozó rendszer típusa
    char            windowTitle[64];             // ablak címsora
    short           windowWidth, windowHeight;   // ablak mérete

    // a származtatott osztályban átdefiniálandó metódusok
    virtual void Init(void) {}                   // megjelenés után hívják
    virtual void Render(void) {}                 // színtér kirajzolása

    virtual void MousePressed(int x,int y) {}    // üzenet az egér lenyomásáról
    virtual void MouseReleased(int x,int y) {}   // üzenet az egér elengedéséről
    virtual void MouseMotion(int x,int y) {}     // üzenet az egér mozgatásáról
```

```
    Application(char* windowTitle, int width, int height);
};
```

Az `Init()` és a `Render()` metódusok virtuális függvények, tehát ezeket a származtatott osztályban újradefiniálhatjuk. Egy `Application` objektumot az `Init()` függvény hozza kezdőállapotba. A színteret a `Render()` metódus rajzolja ki.

Egy ablakozó rendszerfüggetlen alkalmazást úgy írunk, hogy az `Application` osztályból egy `MyApp` osztályt származtatunk:

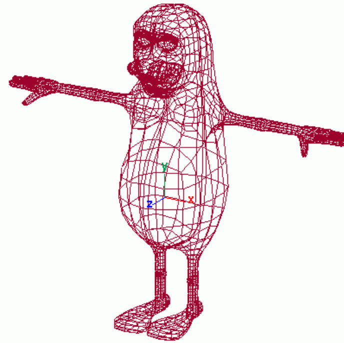
```
//===============================================================
class MyApp : public Application {
//===============================================================
public:
    MyApp(char* windowTitle, int width, int height) :
        Application(windowTitle, width, height) { ... }

    void Init(void)   { ... }
    void Render(void) { ,,ablakozófüggetlen OpenGL rajzolás'' }
};
```

Az alkalmazás belépési pontjait, tehát a `main()` és a `WinMain()` függvényeket, a keretrendszerhez tartozó `OpenGLFramework.cpp` már tartalmazza, ezért ezeket a saját alkalmazásunkban már nem kell definiálni. GLUT alkalmazás esetén a `main()`, Ms-Windows alkalmazás esetén a `WinMain()` a program belépési pontja. Ezt például a `/SUBSYSTEM:WINDOWS` illetve a `/SUBSYSTEM:CONSOLE` paraméterekkel kell a program összeszerkesztésénél (*link*) megadnunk.

A `main()` és a `WinMain()` függvény először a `CreateApplication()`-t hívja, amelyben az alkalmazás létrehozza a saját példányát:

```
void Application::CreateApplication( ) {
    new MyApp("MyApplication Title", 600, 600);
}
```

A példaprogramot a kedves Olvasó a CD-n `OpenGLFramework` néven találja meg. A későbbi fejezetekben ezt a keretrendszert fogjuk kiterjeszteni (például az animáció témakörében időzítéssel és billentyűzetkezeléssel).



3. fejezet

Geometriai modellezés

A virtuális világ definiálását *modellezés*nek nevezzük. A modellezés során megadjuk a világban szereplő objektumok geometriáját (alak, kiterjedés, pozíció, orientáció) és megjelenítési attribútumait (szín, optikai tulajdonságok). Ebben a fejezetben a geometria létrehozásával foglalkozunk.

3.1. Pontok, vektorok és koordinátarendszerek

Egy *pont* a tér egyetlen eleme, amelynek semmiféle kiterjedése nincs. Az *alakzat*ok pontok halmazai. A *vektor* egy eltolás, aminek iránya és hossza van, és amely a tér egy pontjához azt a másik pontot rendeli hozzá, amelyik tőle a megadott irányban és a vektor hosszának megfelelő távolságban van. A vektor hosszát gyakran a vektor *abszolút értékének* is mondjuk és $|\vec{v}|$-vel jelöljük. A vektorokon értelmezzük az *összeadás* műveletet, amelynek eredménye egy újabb vektor, amely az összeadandó eltolások egymás utáni végrehajtását jelenti. A továbbiakban az összeadásra a $\vec{v}_1 + \vec{v}_2 = \vec{v}$ jelölést alkalmazzuk. Beszélhetünk egy vektor és egy szám szorzatáról, amely ugyancsak vektor lesz ($\vec{v}_1 = \lambda \cdot \vec{v}$), és ugyanabba az irányba tol el, mint a $\vec{v}$ szorzandó, de a megadott λ szám arányában kisebb vagy nagyobb távolságra. Egy vektort nemcsak számmal „szorozhatunk”, hanem egy másik vektorral is, ráadásul ezt a műveletet két eltérő módon is definiálhatjuk (félrevezető a vektor–szám szorzást, és a kétféle vektor–vektor szorzást is mind a „szorzás” névvel illetni, hiszen ezek a műveletek különbözőek, de a matematikusok nem mindig jó névadók). Két vektor *skaláris szorzat*a egy szám, amely egyenlő a két vektor hosszának és a bezárt szögük koszinuszának a szorzatával:

$$\vec{v}_1 \cdot \vec{v}_2 = |\vec{v}_1| \cdot |\vec{v}_2| \cdot \cos\alpha, \quad \text{ahol } \alpha \text{ a } \vec{v}_1 \text{ és } \vec{v}_2 \text{ vektorok által bezárt szög.}$$

A skaláris szorzást még szokás belső szorzatnak is nevezni, az angol nyelvi kifejezés pedig a műveleti jelre utal: *dot product*.

Másrészt, két vektor *vektoriális szorzat*a (más néven *keresztszorzat*a, *cross product*) egy vektor, amely merőleges a két vektor síkjára, a hossza pedig a két vektor hosszának és a bezárt szögük szinuszának a szorzata:

$$\vec{v}_1 \times \vec{v}_2 = \vec{v}, \quad \text{ahol } \vec{v} \text{ merőleges } \vec{v}_1,\ \vec{v}_2\text{-re, és } |\vec{v}| = |\vec{v}_1| \cdot |\vec{v}_2| \cdot \sin\alpha.$$

Ez még nem adja meg a vektor irányát egyértelműen, hiszen a fenti szabályt egy vektor és az ellentettje is kielégítené. A két lehetséges eset közül azt az irányt tekintjük vektoriális szorzatnak, amelybe a jobb kezünk középső ujja mutatna, ha a hüvelykujjunkat az első vektor irányába, a mutatóujjunkat pedig a második vektor irányába fordítanánk (*jobbkéz szabály*).

Az elemi vektorműveletekből összetett műveleteket, úgynevezett *transzformáció*kat is összeállíthatunk, amelyek egy $\vec{v}$ vektorhoz egy $\mathcal{A}(\vec{v})$ vektort rendelnek hozzá. Ezek közül különösen fontosak a *lineáris transzformáció*k, amelyek a vektor összeadással és a számmal szorzással felcserélhetők, azaz fennállnak a következő azonosságok:

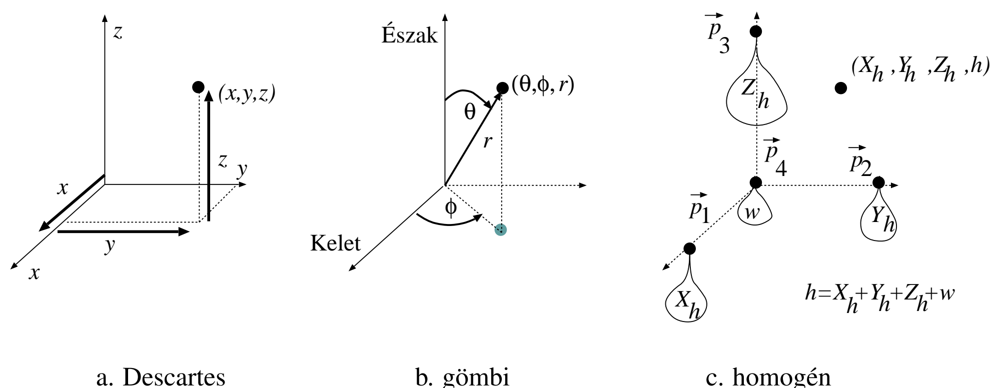
$$\mathcal{A}(\vec{v}_1 + \vec{v}_2) = \mathcal{A}(\vec{v}_1) + \mathcal{A}(\vec{v}_2), \qquad \mathcal{A}(\lambda \cdot \vec{v}) = \lambda \cdot \mathcal{A}(\vec{v}). \tag{3.1}$$

Egy pontot gyakran vektorral adunk meg úgy, hogy megmondjuk, hogy az a tér egy kitüntetett pontjához, az *origó*hoz képest milyen irányban és távolságra van. Hasonlóképpen egy pont is meghatározza azt a vektort, ami az origót éppen ide tolja. Ezen kapcsolat miatt, különösen a programkódokban a pont és vektor fogalma gyakran keveredik. Érdemes azonban hangsúlyozni, hogy ez nem jelenti azt, hogy a vektorok pontok volnának és viszont. Két vektort — azaz két eltolást — például össze lehet adni, két pont összeadása viszont értelmetlen. Ha ebben a könyvben pontok átlagáról beszélünk, akkor ezen azt a pontot értjük, amit a pontoknak megfelelő vektorok átlagaként kapott vektor jelöl ki (de ezt nem mindig írjuk le ilyen körülményesen).

Egy pont, és hasonlóképpen egy vektor egy alkalmasan választott *koordinátarendszer*ben a *koordináták* megadásával definiálható. Ez azért fontos, mert programjainkban kizárólag számokkal dolgozhatunk, a koordinátarendszerek pedig lehetőséget adnak arra, hogy egy geometriai elemet számokkal írjunk le. A megfeleltetésre több lehetőségünk van, így különböző típusú és elhelyezkedésű koordinátarendszerek léteznek. A koordinátarendszerek közös tulajdonsága, hogy a térben referenciaként geometriai elemeket vesznek fel, a pontot pedig ezekhez a geometriai elemekhez mérik.

3.1.1. A Descartes-koordinátarendszer

A *Descartes-koordinátarendszer*ben a viszonyítási rendszer három, egymásra merőleges, egymást az origóban metsző tengely. Egy tetszőleges pontot a tengelyekre vetített távolságokkal jellemzünk (3.1/a. ábra). Síkban ez egy $[x, y]$ számpárt, térben pedig egy $[x, y, z]$ számhármast jelent.



a. Descartes b. gömbi c. homogén

3.1. ábra. *Pontok azonosítása háromdimenziós koordinátarendszerekben*

A Descartes-koordinátarendszer működését vektorokkal is leírhatjuk. Vegyünk fel három, egységnyi hosszú, a koordinátatengelyek irányába mutató $\vec{\mathbf{i}}$, $\vec{\mathbf{j}}$, $\vec{\mathbf{k}}$ *bázisvektor*t. Egy $[x,y,z]$ számhármassal a következő vektort azonosítjuk:

$$\vec{v}[x,y,z] = x\cdot\vec{\mathbf{i}} + y\cdot\vec{\mathbf{j}} + z\cdot\vec{\mathbf{k}}.$$

3.1.2. Program: Descartes-koordinátákkal definiált vektor

A programjainkban a vektorokat (illetve a pontokat) tehát három számmal adhatjuk meg, amelyeket célszerű egy struktúrában vagy C++ osztályban összefoglalni. Egy `Vector` osztály, amely a vektor műveleteket is megvalósítja, a következő:

```
//===============================================================
class Vector {
//===============================================================
   float x, y, z;   // a Descartes-koordináták
   Vector operator+(const Vector& v) { // két vektor összege
      return Vector(x + v.x, y + v.y, z + v.z);
   }
   Vector operator*(float f) {         // vektor és szám szorzata
      return Vector(x * f, y * f, z * f);
   }
   float operator*(const Vector& v) {  // két vektor skaláris szorzata
      return (x * v.x + y * v.y + z * v.z);
   }
   Vector operator%(const Vector& v) { // két vektor vektoriális szorzata
      return Vector(y * v.z - z * v.y, z * v.x - x * v.z, x * v.y - y * v.x);
   }
   float Length() {                    // vektor abszolút értéke
      return (float)sqrt(x * x + y * y + z * z);
   }
   float * GetArray() { return &x; }   // struktúra kezdőcíme
};
```

3.1.3. Síkbeli polár és térbeli gömbi koordinátarendszer

A síkbeli *polár-koordinátarendszer* lényegében egy referencia pontból induló félegyenes. Egy tetszőleges pontot a referencia ponttól vett távolságával és azzal a szöggel adunk meg, amely a félegyenes, valamint a referencia pontra és az adott pontra illeszkedő egyenes között mérhető. Az eljárás könnyen általánosítható a háromdimenziós térre is. Ekkor két, az origóból induló, egymásra merőleges félegyenesre és két szögre van szükségünk. Nevezzük az első félegyenes irányát keleti iránynak, a másodikét pedig északinak (3.1/b. ábra). A két szöggel egy tetszőleges irányt azonosíthatunk. Az adott irány és az északi irány közötti szöget jelöljük θ-val. Képezzük az adott iránynak az északi irányra merőleges (és így a keleti irányt tartalmazó) síkra vett vetületét. Ezen vetület és a keleti irány közötti szög jele legyen ϕ. A két szög alapján a pont irányát már tudjuk, még az origótól mért r távolságot kell megadnunk, tehát a pontot jellemző *gömbi-koordináták* a (θ, ϕ, r) számhármas (3.1/b. ábra). Ez a koordinátarendszer azért érdemelte ki a „gömbi” nevet, mert ha a harmadik, az origótól mért távolságot kifejező koordinátát rögzítjük, akkor a másik két koordináta változtatásával egy gömbfelület mentén mozoghatunk.

Egy pont Descartes-féle és gömbi koordinátarendszerben egyaránt kifejezhető, tehát a két rendszer koordinátái kapcsolatban állnak egymással. A kapcsolat annak függvénye, hogy a két rendszer viszonyítási elemeit egymáshoz képest hogyan helyeztük el. Tegyük fel, hogy gömbi és a Descartes-koordinátarendszerünk origója egybeesik, a Descartes-koordinátarendszer z tengelye az északi iránynak, az x tengely pedig a keleti iránynak felel meg. Ebben az esetben a (θ, ϕ, r) gömbi koordinátákkal jellemzett pont Descartes-koordinátái:

$$x = r \cdot \cos\phi \cdot \sin\theta, \quad y = r \cdot \sin\phi \cdot \sin\theta, \quad z = r \cdot \cos\theta.$$

3.1.4. Baricentrikus koordináták

A Descartes-koordinátarendszerben a viszonyítás alapja három egymást metsző, és egymásra merőleges tengely, a gömbi koordinátarendszerben pedig egy pont és két itt kezdődő félegyenes. Egy pont azonosítását a Descartes-koordinátarendszer hosszúság mérésre, a polár-koordinátarendszer pedig hosszúság és szögmérésre vezette vissza. A jelenlegi és a következő fejezet koordinátarendszereiben térbeli pontokat választunk viszonyítási alapként, egy tetszőleges pont azonosításához pedig egy mechanikai analógiát használunk.

A mechanikai analógia érdekében tegyünk egy kis kitérőt a fizikába, és helyezzük el gondolatban az $m_1, m_2, \ldots, m_n$ tömegeket az $\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_n$ pontokban (3.2. ábra). A rendszer *tömegközéppont*ját a következő kifejezéssel definiáljuk, amely az egyes pontok

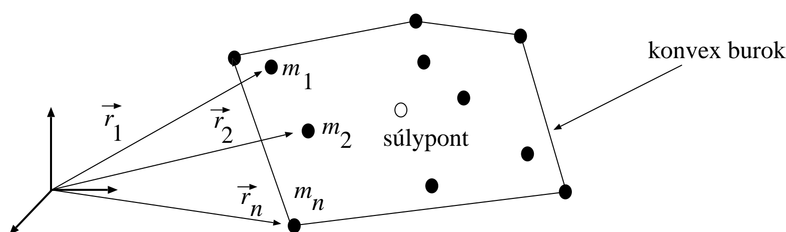
helyvektorait az ott található tömegekkel súlyozva átlagolja:

$$\vec{r}_c = \frac{\sum_{i=1}^{n} m_i \cdot \vec{r}_i}{\sum_{i=1}^{n} m_i}. \tag{3.2}$$

Pongyolán fogalmazva a tömegközéppontot gyakran *súlypont*nak nevezzük. A súlypont a rendszer azon pontja, amelyen felfüggesztve nem billen ki a nyugalmi állapotából. Szigorúan nézve a tömegközéppont csak akkor egyezik meg a súlyponttal, ha a nehézségi gyorsulás minden pontra megegyezik. Mivel ezt Földünk felszínén igen jó közelítéssel elfogadhatjuk, a továbbiakban a rövidebb súlypont kifejezést fogjuk használni.

Az m_i súlyok megváltoztatásával rendszerünk súlypontja megváltozik. Úgy is elképzelhetjük, hogy rögzített $\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_n$ referencia pontok mellett a súlyok variálásával jelölünk ki súlypontokat a térben. Ebben az esetben a referencia pontokat egy *baricentrikus koordinátarendszer*nek, az m_i súlyokat és az összsúly hányadosát pedig a pontot meghatározó *baricentrikus koordinátá*knak tekinthetjük.

A súlypont a mechanikai rendszer (test) „közepén” van. Ez a közép nem feltétlenül esik a test belsejébe, egy úszógumi (*tórusz*) közepe a belső kör közepén van, ott ahova az úszni nem tudó bújik be. Az viszont biztosan nem fordulhat elő, hogy a súlyponthoz képest a test pontjai csak egy irányban legyenek, azaz a test pontjai és a súlypont egy sík két oldalán helyezkedjenek el.



3.2. ábra. *Ponthalmazok, konvex burkok és súlypontok*

Pontosabban a súlypont mindig a test konvex burkán belül van. Egy ponthalmaz *konvex burka* (*convex hull*) az a minimális konvex halmaz, amely a ponthalmazt tartalmazza (3.2. ábra). Egy ponthalmazt akkor mondunk *konvex*nek, ha bármely két pontját összekötő szakasz teljes egészében a halmazban van.

Konvex burokkal például az ajándékok csomagolásakor találkozhatunk, hiszen a szépen kifeszített csomagolópapír éppen a tárgyak konvex burkára simul rá. Hasznos lehet a konvex burok fogalom ismerete akkor is, ha a sivatagban egy csapat alvó oroszlán

közé kerülünk, és egyetlen „fegyverünk” egy tekercs drótkerítés, amivel az oroszlánokat bekeríthetjük mielőtt felébrednek. Ebben az esetben a kerítést az oroszláncsapat konvex burka mentén kell kifeszíteni, ugyanis ekkor lesz a legrövidebb, így ekkor végzünk a leghamarabb.

3.1.5. Homogén koordináták

A fizikai és afrikai kalandok után térjünk vissza a pontok azonosításához, és a súlypont-analógia megtartása mellett pontosítsuk a mechanikai rendszerünket (3.1/c. ábra). A homogén koordináták alkalmazásakor a pontjainkat mechanikai rendszerek súlypontjaiként írjuk le. Egyetlen pont azonosításához egy $\vec{p}_1$ referencia pontban X_h súlyt helyezünk el, egy $\vec{p}_2$ referencia pontban Y_h súlyt, egy $\vec{p}_3$ pontban Z_h súlyt és végül egy $\vec{p}_4$ pontban w súlyt. A mechanikai rendszer súlypontja:

$$\vec{r}_c = \frac{X_h \cdot \vec{p}_1 + Y_h \cdot \vec{p}_2 + Z_h \cdot \vec{p}_3 + w \cdot \vec{p}_4}{X_h + Y_h + Z_h + w}.$$

Vezessük be az összsúly fogalmát a $h = X_h + Y_h + Z_h + w$ egyenlettel! Definíciószerűen az (X_h, Y_h, Z_h, h) négyest a súlypont *homogén koordinátái*nak nevezzük. A „homogén” elnevezés abból származik, hogy ha az összes súlyt ugyanazzal a skalárral szorozzuk, a súlypont, azaz a négyes által definiált pont nem változik, tehát minden nem zérus λ-ra a $(\lambda X_h, \lambda Y_h, \lambda Z_h, \lambda h)$ négyesek ugyanazt a pontot azonosítják.

Ebben az esetben is érdemes kapcsolatot keresni a homogén és a Descartes-koordináták között. Egy ilyen összefüggés felállításához a két koordinátarendszer viszonyát (a Descartes-koordinátarendszer tengelyeinek és a homogén koordinátarendszer referencia pontjainak viszonyát) rögzíteni kell. Tegyük fel például, hogy a referencia pontok a Descartes-koordinátarendszer [1,0,0], [0,1,0], [0,0,1] és [0,0,0] pontjaiban vannak. A mechanikai rendszerünk súlypontja (ha a h összsúly nem zérus) a Descartes-koordinátarendszerben:

$$\vec{r}(X_h, Y_h, Z_h, h) = \frac{1}{h} \cdot (X_h \cdot [1,0,0] + Y_h \cdot [0,1,0] + Z_h \cdot [0,0,1] + w \cdot [0,0,0]) = \left[\frac{X_h}{h}, \frac{Y_h}{h}, \frac{Z_h}{h}\right].$$

Tehát az (X_h, Y_h, Z_h, h) homogén koordináták és az (x, y, z) Descartes-koordináták közötti összefüggés ($h \neq 0$):

$$x = \frac{X_h}{h}, \qquad y = \frac{Y_h}{h}, \qquad z = \frac{Z_h}{h}. \tag{3.3}$$

A negyedik koordinátával történő osztást *homogén osztás*nak nevezzük.

A Descartes-koordinátákat többféleképpen alakíthatjuk homogén koordinátákká, mert a homogén koordináták egy skalárral szabadon szorozgathatóak. Ha az x, y, z Descartes-koordináta hármas ismert, akkor bármely $(x \cdot h, y \cdot h, z \cdot h, h)$ négyes megfelel ($h \neq 0$), hiszen ezek mindegyike kielégíti a 3.3. egyenletet. A lehetséges megoldások közül

gyakran célszerű azt kiválasztani, ahol a negyedik koordináta 1 értékű, ugyanis ekkor az első három homogén koordináta a Descartes-koordinátákkal egyezik meg:

$$X_h = x, \qquad Y_h = y, \qquad Z_h = z, \qquad h = 1. \tag{3.4}$$

Descartes-koordinátákat tehát úgy alakíthatunk homogén koordinátákká, hogy hozzájuk csapunk egy negyedik 1 értékű elemet.

Az ismertetett összerendelésnek messzemenő következményei vannak. Például, ez bizonyíték arra, hogy minden Descartes-koordinátákkal kifejezhető pontot (az *euklideszi tér* pontjait) megadhatunk homogén koordinátákkal. Ha a homogén koordináták súlypontot használó bevezetése miatt az Olvasó idáig ebben kételkedett volna, abban semmi meglepő sincs. Megszoktuk ugyanis, hogy a súlypont a test belsejében (egész pontosan a test konvex burkán belül) van, tehát most is azt várnánk, hogy a homogén koordinátákkal megadott pont a négy referencia pont „között” helyezkedik el, és például nem kerülhet az ezen kívül levő $[2,0,0]$ pontra. Nézzük akkor most meg erre a pontra az összerendelésből következő homogén koordinátákat: $X_h = 2$, $Y_h = 0$, $Z_h = 0$, $h = 1$, azaz az $[1,0,0]$ pontba 2 súlyt, a $[0,1,0]$ pontba és $[0,0,1]$ pontba 0 súlyt, végül a $[0,0,0]$ pontba $w = h - X_h - Y_h - Z_h = -1$ súlyt kell tennünk. Itt van tehát a kutya elásva! Azért tudunk a referencia pontok konvex burkából kilépni, és azért vagyunk képesek az összes pontot leírni, mert a súlyok lehetnek negatívak is.

3.2. Geometriai transzformációk

A számítógépes grafikában geometriai alakzatokkal dolgozunk. Az alakzatok megváltoztatását *geometriai transzformáció*nak nevezzük.

Mivel a számítógépekben mindent számokkal jellemzünk, a geometriai leírást is számok megadására vezetjük vissza. A pontokat és a vektorokat, egy alkalmas koordinátarendszer segítségével, számhármasokkal vagy számnégyesekkel írjuk le. A transzformáció pedig a vektorok koordinátáin értelmezett matematikai művelet. A 3.1. egyenlet felírásával már kijelöltük ezen matematikai műveletek egy fontos csoportját, a *lineáris transzformáció*kat.

Tekintsük először a Descartes-koordinátarendszerben megadott vektorok lineáris transzformációit. Egy $\mathcal{A}$ lineáris transzformáció a 3.1. egyenlet értelmében az összeadással és a számmal való szorzással felcserélhető, így az $[x,y,z]$ vektor transzformáltját a következőképpen is írhatjuk:

$$\mathcal{A}(x \cdot \vec{\mathbf{i}} + y \cdot \vec{\mathbf{j}} + z \cdot \vec{\mathbf{k}}) = x \cdot \mathcal{A}(\vec{\mathbf{i}}) + y \cdot \mathcal{A}(\vec{\mathbf{j}}) + z \cdot \mathcal{A}(\vec{\mathbf{k}}).$$

Az $\vec{\mathbf{i}}, \vec{\mathbf{j}}, \vec{\mathbf{k}}$ bázisvektorok transzformáltjai is vektorok, amelyeket az adott Descartes-koordinátarendszerben koordinátákkal azonosíthatunk. Jelöljük az $\mathcal{A}(\vec{\mathbf{i}})$ vektor koordinátáit $[a_{1,1}, a_{1,2}, a_{1,3}]$-mal, az $\mathcal{A}(\vec{\mathbf{j}})$ koordinátáit $[a_{2,1}, a_{2,2}, a_{2,3}]$-mal, az $\mathcal{A}(\vec{\mathbf{k}})$ koordi-

nátáit pedig $[a_{3,1}, a_{3,2}, a_{3,3}]$-mal. Összefoglalva, az $[x,y,z]$ vektor transzformáltjának $[x',y',z']$ koordinátái:

$$[x',y',z'] = [x \cdot a_{1,1} + y \cdot a_{2,1} + z \cdot a_{3,1},\; x \cdot a_{1,2} + y \cdot a_{2,2} + z \cdot a_{3,2},\; x \cdot a_{1,3} + y \cdot a_{2,3} + z \cdot a_{3,3}].$$

Ezt a műveletet szemléletesebben, táblázatos formában is felírhatjuk:

$$[x',y',z'] = [x,y,z] \cdot \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}.$$

A kifejezésben szereplő számtáblázat egy mátrix [108]. A *mátrix* számoknak egy kétdimenziós, $n \times m$-es, azaz n sorból és m oszlopból álló táblázata, amelyen különböző műveletek hajthatók végre. Két azonos szerkezetű, azaz megegyező számú sorból és oszlopból álló *mátrix összeg*e egy ugyanilyen szerkezetű mátrix, amelynek az elemei, a két összeadandó ugyanezen a helyen lévő elemeinek az összege:

$$\begin{bmatrix} a_{1,1} & \dots & a_{1,m} \\ a_{2,1} & \dots & a_{2,m} \\ \vdots & & \\ a_{n,1} & \dots & a_{n,m} \end{bmatrix} + \begin{bmatrix} b_{1,1} & \dots & b_{1,m} \\ b_{2,1} & \dots & b_{2,m} \\ \vdots & & \\ b_{n,1} & \dots & b_{n,m} \end{bmatrix} = \begin{bmatrix} a_{1,1}+b_{1,1} & \dots & a_{1,m}+b_{1,m} \\ a_{2,1}+b_{2,1} & \dots & a_{2,m}+b_{2,m} \\ \vdots & & \\ a_{n,1}+b_{n,1} & \dots & a_{n,m}+b_{n,m} \end{bmatrix}.$$

Amikor egy mátrixot egy számmal szorzunk, akkor a mátrix elemeire a szorzást egyenként végezzük el:

$$\lambda \cdot \begin{bmatrix} a_{1,1} & \dots & a_{1,m} \\ a_{2,1} & \dots & a_{2,m} \\ \vdots & & \\ a_{n,1} & \dots & a_{n,m} \end{bmatrix} = \begin{bmatrix} \lambda \cdot a_{1,1} & \dots & \lambda \cdot a_{1,m} \\ \lambda \cdot a_{2,1} & \dots & \lambda \cdot a_{2,m} \\ \vdots & & \\ \lambda \cdot a_{n,1} & \dots & \lambda \cdot a_{n,m} \end{bmatrix}.$$

Az egyik legizgalmasabb mátrixművelet a *mátrixszorzás*. Nem szorozható össze két tetszőleges szerkezetű mátrix, a szorzást csak akkor értelmezzük, ha az első mátrix oszlopainak a száma megegyezik a második mátrix sorainak a számával. Ha az első mátrix $n \times K$ elemű, a második pedig $K \times m$ elemű, akkor az eredmény egy $n \times m$ elemű mátrix lesz, amelyben az i, j elem az első mátrix i-edik sorában és a második mátrix j-edik oszlopában lévő elemek szorzatainak az összege:

$$\begin{bmatrix} a_{1,1} & \dots & a_{1,K} \\ a_{2,1} & \dots & a_{2,K} \\ \vdots & & \\ a_{n,1} & \dots & a_{n,K} \end{bmatrix} \cdot \begin{bmatrix} b_{1,1} & \dots & b_{1,m} \\ b_{2,1} & \dots & b_{2,m} \\ \vdots & & \\ b_{K,1} & \dots & b_{K,m} \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{K} a_{1,k} \cdot b_{k,1} & \dots & \sum_{k=1}^{K} a_{1,k} \cdot b_{k,m} \\ \sum_{k=1}^{K} a_{2,k} \cdot b_{k,1} & \dots & \sum_{k=1}^{K} a_{2,k} \cdot b_{k,m} \\ \vdots & & \\ \sum_{k=1}^{K} a_{n,k} \cdot b_{k,1} & \dots & \sum_{k=1}^{K} a_{n,k} \cdot b_{k,m} \end{bmatrix}.$$

Szemléletesen, ha az eredménymátrix $c_{i,j}$ elemére vagyunk kíváncsiak, akkor tegyük bal kezünk mutatóujját az első szorzandó mátrix i-edik sorának első, azaz legbaloldalibb elemére, a jobb kezünk mutatóujját pedig a második mátrix j-edik oszlopának első, azaz legfelső elemére. Szorozzuk össze a két számot, majd csúsztassuk a bal kezünket jobbra, a következő elemre, a jobb kezünket pedig lejjebb! Ezeket a számokat ismét szorozzuk össze, és az eredményt adjuk hozzá az első két szám szorzatához, majd ismételgessük az eljárást, amíg a két kezünk a sor illetve az oszlop végére ér! Az a feltétel, hogy az első mátrix oszlopainak száma meg kell, hogy egyezzen a második mátrix sorainak a számával, azt jelenti, hogy a jobb és bal kezünk alatt éppen egyszerre fogynak el a számok. A mátrixszorzás és összeadás programszintű megvalósítását 4×4-s mátrixokra a 3.2.10. fejezetben adjuk meg.

A mátrixszorzásban a tényezők sorrendje nem cserélhető fel ($\mathbf{A} \cdot \mathbf{B} \neq \mathbf{B} \cdot \mathbf{A}$, a művelet nem *kommutatív*), viszont a zárójelezés áthelyezhető ($\mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{C}) = (\mathbf{A} \cdot \mathbf{B}) \cdot \mathbf{C}$, a művelet *asszociatív*).

Ha a sorok és az oszlopok száma megegyezik a mátrix *négyzetes*. A négyzetes mátrixok között fontos szerepet játszik az *egységmátrix* (*identity matrix*, **E**), amelynek a főátlójában csupa 1 érték található, a főátlón kívül lévő értékek pedig nullák (*főátló*nak azokat az $a_{i,j}$ elemeket nevezzük, ahol $i = j$). Egy négyzetes *mátrix inverze* az a négyzetes mátrix, amellyel szorozva eredményül az egységmátrixot kapjuk. Az **A** mátrix inverzét $\mathbf{A}^{-1}$-gyel jelöljük, így $\mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{E}$. Csak négyzetes mátrixoknak lehet inverzük, de azok közül sincs mindegyiknek. Például a csupa zéruselemet tartalmazó *zérusmátrix*nak nincs inverze, hiszen nullával szorozva sohasem kaphatunk egyet eredményként.

Egy n elemű vektort tekinthetünk egy $n \times 1$ elemű, egyetlen oszlopból álló mátrixnak, vagy akár egy $1 \times n$ elemű, egyetlen sorból álló mátrixnak. Így a mátrixszorzás szabályainak megfelelően, beszélhetünk vektorok és mátrixok szorzatáról is, mégpedig kétféleképpen. A vektort $1 \times n$ elemű mátrixnak tekintve, megszorozhatjuk egy $n \times m$ elemű mátrixszal. Másrészt a vektort $n \times 1$ elemű mátrixnak tekintve, egy $m \times n$ elemű mátrixot megszorozhatjuk a vektorunkkal. Az első esetben egy m elemű sorvektort, a másodikban pedig egy m elemű oszlopvektort kapunk, amelynek elemei általában nem lesznek ugyanazok, mivel a mátrixszorzás nem kommutatív.

A matematikában gyakrabban alkalmazzák az első megközelítést, amikor a vektorok oszlopvektorok, mi azonban főleg a második formát fogjuk előnyben részesíteni. A számítógépes grafikában ennek főleg történelmi hagyományai vannak, amit tiszteletben tartunk. Hangsúlyozzuk, ez csak egy jelöléstechnika, ami a lényeget nem érinti. Mégis fontos, hogy pontosan tisztában legyünk azzal, hogy éppen melyik esettel dolgozunk, mert a mátrixokat ennek megfelelően tükrözni kell. Ha tehát a kedves Olvasó más irodalmakban olyan mátrixokra lel, amelyek az ebben a könyvben szereplő változatok tükörképei, akkor a különbség az eltérő értelmezésben keresendő.

A mátrixelméleti kitérő után kanyarodjunk vissza a geometriai transzformációkhoz.

A bevezető példa tanulsága, hogy tetszőleges *lineáris transzformáció* felírható egy 3×3-as mátrixszorzással. Ennek a mátrixnak a sorai a bázisvektorok transzformáltjai. Mint látni fogjuk, a lineáris transzformációk sokféle fontos geometriai műveletet foglalnak magukban, mint a nagyítást, a forgatást, a nyírást, a tükrözést, a merőleges vetítést stb. Egy alapvető transzformáció, az eltolás, azonban kilóg ebből a családból. Az eltolást és a lineáris transzformációkat is tartalmazó bővebb családot *affin transzformáció*knak nevezzük. Az affin transzformációkra az jellemző, hogy a párhuzamos egyeneseket párhuzamos egyenesekbe viszik át. A következőkben először elemi affin transzformációkkal ismerkedünk meg, majd ezt a családot is bővítjük a *projektív transzformáció*k körére, amely a középpontos vetítést is tartalmazza. Végül az utolsó alfejezetben általános, nemlineáris transzformációkkal foglalkozunk.

3.2.1. Eltolás

Az *eltolás* (*translation*) egy konstans $\vec{v}$ vektort ad hozzá a transzformálandó $\vec{r}$ ponthoz:

$$\vec{r}\,' = \vec{r} + \vec{v}.$$

Descartes-koordinátákban:

$$x' = x + v_x, \qquad y' = y + v_y, \qquad z' = z + v_z.$$

3.2.2. Skálázás a koordinátatengely mentén

A *skálázás* (*scaling*) a távolságokat és a méreteket a különböző koordinátatengelyek mentén függetlenül módosítja. Például egy $[x, y, z]$ pont skálázott képének koordinátái:

$$x' = S_x \cdot x, \qquad y' = S_y \cdot y, \qquad z' = S_z \cdot z.$$

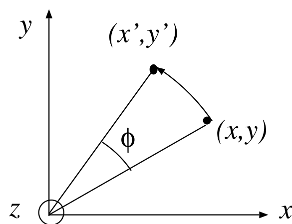
Ezt a transzformációt mátrixszorzással is leírhatjuk:

$$\vec{r}\,' = \vec{r} \cdot \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix}. \tag{3.5}$$

3.2.3. Forgatás a koordinátatengelyek körül

A z tengely körüli ϕ szöggel történő *forgatás* (*rotation*) az x és y koordinátákat módosítja, a z koordinátát változatlanul hagyja. Az elforgatott pont x és y koordinátái a következőképpen fejezhetők ki (3.3. ábra):

$$x' = x \cdot \cos\phi - y \cdot \sin\phi, \qquad y' = x \cdot \sin\phi + y \cdot \cos\phi. \tag{3.6}$$



3.3. ábra. *Forgatás a z tengely körül*

A továbbiakban a szinusz és koszinusz függvényekre az $S_\phi = \sin\phi$, $C_\phi = \cos\phi$ rövid jelölést alkalmazzuk. A forgatás mátrix művelettel is kifejezhető:

$$\vec{r}\,'(z,\phi) = \vec{r} \cdot \begin{bmatrix} C_\phi & S_\phi & 0 \\ -S_\phi & C_\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \qquad (3.7)$$

Az x és y tengelyek körüli forgatásnak hasonló alakja van, csupán a koordináták szerepét kell felcserélni:

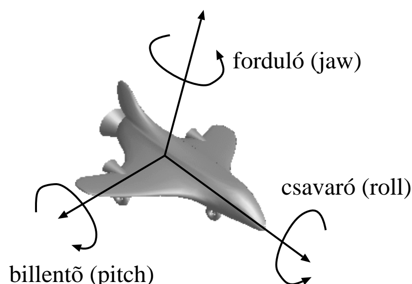
$$\vec{r}\,'(x,\phi) = \vec{r} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & C_\phi & S_\phi \\ 0 & -S_\phi & C_\phi \end{bmatrix}, \quad \vec{r}\,'(y,\phi) = \vec{r} \cdot \begin{bmatrix} C_\phi & 0 & -S_\phi \\ 0 & 1 & 0 \\ S_\phi & 0 & C_\phi \end{bmatrix}.$$

Bármely orientáció előállítható három egymás utáni forgatással. Először a z tengely körül forgatunk α szöggel, majd az új, elfordult y' tengely körül β szöggel, végül pedig a második forgatást is elszenvedő x'' tengely körül γ szöggel. Mivel az elfordulás szögét mindig a korábbi lépésekben már elforgatott koordinátarendszerben értelmezzük, a forgatási tengelyek sorrendje nem cserélhető fel. Az α, β, γ szögeket rendre *csavaró* (*roll*), *billentő* (*pitch*) és *forduló* (*yaw*) szögeknek vagy röviden *RPY szögek*nek nevezik (3.4. ábra).

Az (α, β, γ) csavaró–billentő–forduló szögekkel megadott orientációba a következő mátrix visz át:

$$\vec{r}\,'(\alpha,\beta,\gamma) = \vec{r} \cdot \begin{bmatrix} C_\alpha & S_\alpha & 0 \\ -S_\alpha & C_\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} C_\beta & 0 & -S_\beta \\ 0 & 1 & 0 \\ S_\beta & 0 & C_\beta \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & C_\gamma & S_\gamma \\ 0 & -S_\gamma & C_\gamma \end{bmatrix}.$$

Az ilyen *orientációs mátrix*ok sorvektorai egymásra merőleges egységvektorok (úgynevezett *ortonormált mátrix*ok), amelyeket egyszerűen invertálhatunk úgy, hogy az elemeket tükrözzük a főátlóra, azaz a mátrixot *transzponáljuk*.



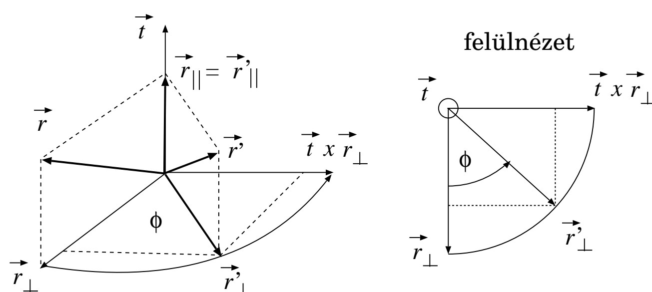
3.4. ábra. *Csavaró (roll), billentő (pitch) és forduló (yaw) szögek*

3.2.4. Általános tengely körüli forgatás

Most vizsgáljuk meg azt az általános esetet, amikor egy, a koordinátarendszer origóján átmenő tengely körül ϕ szöggel forgatunk. Jelöljük a forgástengelyt $\vec{t}$-vel és tegyük fel, hogy a $\vec{t}$ vektor egységnyi hosszú (ezen vektor hossza nyilván nem befolyásolja a forgatást). Az eredeti $\vec{r}$ és az elforgatott $\vec{r}\,'$ vektorokat felbontjuk egy-egy, a forgástengellyel párhuzamos $\vec{r}_{\parallel}$, illetve $\vec{r}_{\parallel}\,'$, és egy-egy, a forgástengelyre merőleges $\vec{r}_{\perp}$, illetve $\vec{r}_{\perp}\,'$ komponensre. Az eredeti vektor párhuzamos komponensét, a forgástengelyre vett vetületként, a merőlegest pedig az eredeti vektor és a vetület különbségeként állíthatjuk elő:

$$\vec{r}_{\parallel} = \vec{t}(\vec{t}\cdot\vec{r}), \qquad \vec{r}_{\perp} = \vec{r} - \vec{r}_{\parallel} = \vec{r} - \vec{t}(\vec{t}\cdot\vec{r}).$$

Mivel a forgatás a párhuzamos komponenst változatlanul hagyja: $\vec{r}_{\parallel}\,' = \vec{r}_{\parallel}$.



3.5. ábra. *A $\vec{t}$ tengely körüli ϕ szögű forgatás*

Az $\vec{r}_{\perp}$ és $\vec{r}_{\perp}\,'$ vektorok, valamint a $\vec{t}\times\vec{r}_{\perp} = \vec{t}\times\vec{r}$ vektor a $\vec{t}$ tengelyre merőleges síkban vannak és ugyanolyan hosszúak. A $\vec{r}_{\perp}$ és $\vec{t}\times\vec{r}_{\perp}$ egymásra merőlegesek (3.5. ábra). A z tengely körüli forgatáshoz hasonlóan, az $\vec{r}_{\perp}$ és $\vec{t}\times\vec{r}_{\perp}$ merőleges vektorok kombinációjaként felírhatjuk az elforgatott vektor $\vec{r}_{\perp}\,'$ merőleges komponensét:

$$\vec{r}_{\perp}\,' = \vec{r}_{\perp}\cdot C_{\phi} + \vec{t}\times\vec{r}_{\perp}\cdot S_{\phi}.$$

Az elforgatott $\vec{r}\,'$ vektor a merőleges és párhuzamos komponenseinek az összege:

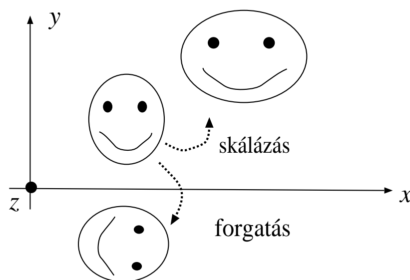
$$\vec{r}\,' = \vec{r}\,'_{\|} + \vec{r}\,'_{\perp} = \vec{r}\cdot C_\phi + \vec{t}\times\vec{r}\cdot S_\phi + \vec{t}(\vec{t}\cdot\vec{r})(1-C_\phi).$$

Ez az egyenlet *Rodrigues-képlet* néven ismeretes, amelyet ugyancsak megadhatunk mátrixos formában is:

$$\vec{r}\,' = \vec{r}\cdot\begin{bmatrix} C_\phi(1-t_x^2)+t_x^2 & t_xt_y(1-C_\phi)+S_\phi t_z & t_xt_z(1-C_\phi)-S_\phi t_y \\ t_yt_x(1-C_\phi)-S_\phi t_z & C_\phi(1-t_y^2)+t_y^2 & t_xt_z(1-C_\phi)+S_\phi t_x \\ t_zt_x(1-C_\phi)+S_\phi t_y & t_zt_y(1-C_\phi)-S_\phi t_x & C_\phi(1-t_z^2)+t_z^2 \end{bmatrix}.$$

3.2.5. A transzformációk támpontja

Az idáig megismert forgatási és skálázási transzformációk az origón átmenő tengely körül forgatnak és az origóhoz viszonyítva skáláznak. Más szemszögből, a transzformációk az origót változatlanul hagyják, a többi pontot pedig az origóhoz képest változtatják meg. A transzformációk helyben maradó, viszonyítási pontját *fixpont*nak vagy *támpont*nak (*pivot point*) nevezzük. A támpont origóhoz rögzítése nem mindig ad kielégítő eredményt (3.6. ábra), hiszen ekkor a skálázás nemcsak az alakzat méretét változtatja meg, hanem távolabbra is viszi, a forgatás pedig ugyancsak elmozdítja az eredeti helyéről. Könnyen elképzelhető, hogy sok esetben az alakzatot „helyben” szeretnénk felnagyítani illetve elforgatni, azaz a transzformáció helyben maradó támpontját egy általunk kijelölt $\vec{p}$ pontra kívánjuk beállítani.



3.6. ábra. *Skálázás és forgatás az origót tekintve támpontnak*

Az általános támpontú skálázást és forgatást visszavezethetjük az origó támpontú esetre, ha a transzformáció előtt az objektumot eltoljuk úgy, hogy a támpont az origóba kerüljön, elvégezzük az origó támpontú transzformációt, végül pedig visszatoljuk az eredményt úgy, hogy az origó ismét a támpontba menjen át. Formálisan egy $\vec{p}$ támpontú 3×3-as $\mathbf{A}$ mátrixú forgatás illetve skálázás képlete:

$$\vec{r}\,' = (\vec{r}-\vec{p})\cdot\mathbf{A}+\vec{p}.$$

3.2.6. Az elemi transzformációk homogén koordinátás megadása

Az idáig megismert transzformációk, az eltolást kivéve, *lineáris transzformáció*k, ezért mátrixszorzással is elvégezhetők. Ez azért hasznos, mert ha egymás után több ilyen transzformációt kell végrehajtani, akkor a transzformációs mátrixok szorzatával (más néven *konkatenált*jával) való szorzás egyszerre egy egész transzformáció sorozaton átvezeti a pontot, így egyetlen transzformáció számítási munkaigényét és idejét felhasználva tetszőleges számú transzformációt elvégezhetünk (erre a mátrixszorzás asszociativitása miatt van lehetőségünk). Sajnos az eltolás ezt a szép képet eltorzítja, ezért az eltolást és a lineáris transzformációkat magába foglaló *affin transzformáció*kat már egy kicsit körülményesebben kell kezelnünk. Az affin transzformációkat azzal a tulajdonsággal definiálhatjuk, hogy a transzformált koordináták az eredeti koordináták lineáris függvényei [51], tehát általános esetben:

$$[x', y', z'] = [x, y, z] \cdot \mathbf{A} + [p_x, p_y, p_z], \tag{3.8}$$

ahol $\mathbf{A}$ egy 3×3-as mátrix, amely az elmondottak szerint jelenthet forgatást, skálázást stb., sőt ezek tetszőleges kombinációját is. A különálló $\vec{p}$ vektor pedig az eltolásért felelős.

Az eltolás és a többi transzformáció egységes kezelésének érdekében szeretnénk az eltolást is mátrixművelettel leírni. Egy háromdimenziós eltolást sajnos nem erőltethetünk be egy 3×3-as mátrixba, mert ott már nincs erre hely. Azonban, ha a Descartes-koordináták helyett homogén koordinátákkal dolgozunk, és ennek megfelelően a mátrixot 4×4-esre egészítjük ki, akkor már az eltolást is mátrixszorzással kezelhetjük. Emlékezzünk vissza, hogy a Descartes-homogén koordináta váltáshoz a vektorunkat is ki kell bővíteni egy negyedik, 1 értékű koordinátával. Ebben az esetben a 3.8. egyenlet, azaz az általános affin transzformáció, a következő formában is felírható:

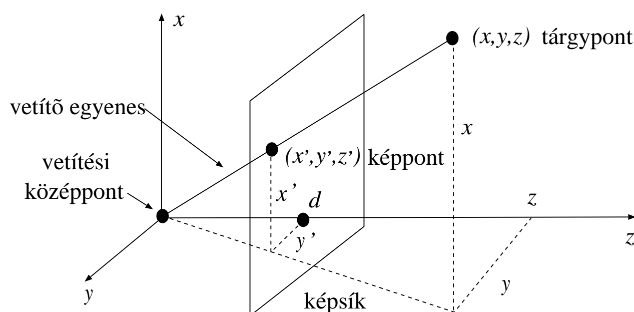
$$[x', y', z', 1] = [x, y, z, 1] \cdot \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 \\ A_{21} & A_{22} & A_{23} & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ p_x & p_y & p_z & 1 \end{bmatrix} = [[x, y, z] \cdot \mathbf{A} + \vec{p}, 1]. \tag{3.9}$$

A homogén koordinátákra a forgatás, skálázás, eltolás mind hasonlóan, egy mátrixszorzással megadható. Az affin transzformációkban a mátrix negyedik oszlopa mindig $[0, 0, 0, 1]$, tehát a pont negyedik 1 értékű koordinátáját a transzformáció nem rontja el. Egy affin transzformáció, egy Descartes-koordinátákban adott pontból egy másik, Descartes-koordinátákkal adott pontot készít, csak a negyedik 1-es koordinátát kell figyelmen kívül hagynunk.

Ha a mátrix negyedik oszlopában nem ragaszkodunk a $[0, 0, 0, 1]$ értékekhez, akkor egy még általánosabb transzformáció típushoz, a *projektív transzformáció*khoz jutunk. Ekkor persze a transzformált vektor negyedik koordinátája nem feltétlenül lesz 1 értékű,

azaz az eredmény nem Descartes-koordinátákban, hanem homogén koordinátákban áll elő. A következő fejezetben tárgyalt középpontos vetítés a *projektív transzformáció*khoz tartozik, és kilóg az affin és a lineáris transzformációk köréből. Mivel a projektív transzformációkban a transzformált pont homogén koordinátáit az eredeti pont homogén koordinátáiból egy mátrix szorzással kapjuk, gyakran *homogén lineáris transzformáció*nak is nevezzük ezt a családot. A homogén lineáris transzformációk a számítógépes grafikában szerzett rendkívüli népszerűségüket annak köszönhetik, hogy az affin transzformációknál bővebbek, de azokhoz hasonlóan továbbra is pontot pontba, egyenest egyenesbe, síkot síkba visznek át[1]. Ez a tulajdonság azért fontos, mert ekkor szakaszok és sokszögek esetén elegendő a csúcspontjaikra kiszámítani a transzformációt. Ráadásul, mivel homogén koordinátákkal a párhuzamosok metszéspontjaként értelmezhető végtelen távoli pontok is leírhatók véges számokkal, a középpontos vetítés elvégzése során nem kell kizárni az euklideszi geometriában nem kezelhető pontokat.

3.2.7. A középpontos vetítés



3.7. ábra. *Középpontos vetítés*

Vizsgáljuk meg az origó középpontú *középpontos vetítés* műveletét, egy x,y síkkal párhuzamos, a $[0,0,d]$ ponton keresztülmenő képsíkot feltételezve! A középpontos, más néven *perspektív vetítés* egy x,y,z *tárgypont*hoz azt a képsíkon lévő $[x',y',z']$ *képpont*ot rendeli hozzá, ahol a *vetítési középpont*ot és a tárgypontot összekötő *vetítő egyenes* metszi a képsíkot. A 3.7. ábra jelölései alapján, a hasonló háromszögeket felismerve kifejezhetjük a képpont Descartes-koordinátáit:

$$x' = \frac{x}{z} \cdot d, \quad y' = \frac{y}{z} \cdot d, \quad z' = d.$$

Descartes-koordinátákra a művelet nemlineáris, hiszen osztásokat tartalmaz. Írjuk fel a

[1]Elfajulások előfordulhatnak, amikor síkból egyenes vagy pont, illetve egyenesből pont keletkezik.

tárgy- és a képpontot homogén koordinátákban:

$$\begin{aligned}[X_h, Y_h, Z_h, h] &= [x, y, z, 1], \\ [X'_h, Y'_h, Z'_h, h'] &= [\frac{x}{z} \cdot d, \frac{y}{z} \cdot d, d, 1].\end{aligned}$$

A homogén koordináták által azonosított pont nem változik, ha mindegyiküket ugyanazzal az értékkel megszorozzuk. Ha ez az érték éppen z/d, akkor a képpont homogén koordinátái:

$$[X'_h, Y'_h, Z'_h, h'] = [x, y, z, \frac{z}{d}].$$

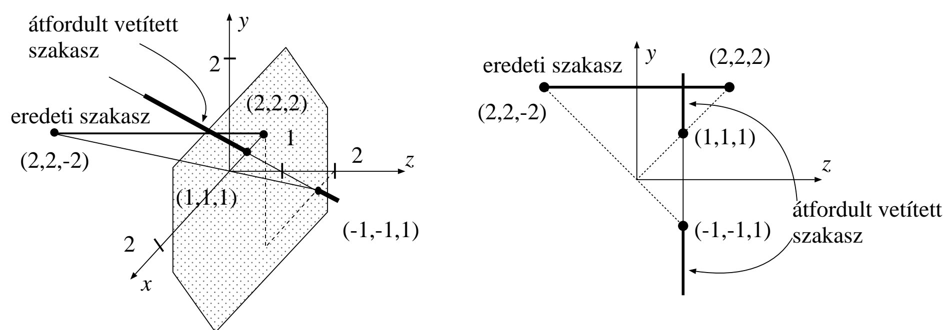
Vegyük észre, hogy a képpont homogén koordinátái valóban lineárisan függenek a tárgypont homogén koordinátáitól, és így kifejezhetők a következő mátrixművelettel is:

$$[X'_h, Y'_h, Z'_h, h'] = [x, y, z, 1] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 0 \end{bmatrix}. \qquad (3.10)$$

Ez még nem minden! Próbáljuk meg alkalmazni ezt az összefüggést egy, az x, y síkon levő $[x, y, 0]$ pontra, azaz próbáljuk vetíteni ezt a pontot az x, y síkkal párhuzamos képsíkra. Ebben az esetben az origón átmenő vetítőegyenesek ugyancsak az x, y síkban lesznek, azaz párhuzamosak a képsíkkal, így nem is metszik azt (a párhuzamosok csak a végtelenben találkoznak, amit Karinthy rossz tanulója így képzelt el: „Látja a végtelent ... nagy, kék valami ... oldalt egy kis házikó is van, amire fel van írva: Bejárat a negyedik végtelenbe. A házban fogasok vannak, ahol a párhuzamos vonalak leteszik a kalapjukat, aztán átmennek a szobába, leülnek a padba, és örömmel üdvözlik egymást"). A 3.10. egyenlet szerint ezen végtelenben lévő pont homogén koordinátái:

$$[X'_h, Y'_h, Z'_h, h'] = [x, y, 0, 0],$$

tehát homogén koordinátákban ezeket a pontokat is megadhatjuk véges számokkal. Az ilyen végtelen távoli és az euklideszi térben nem létező pontokat *ideális pont*oknak nevezzük. Az euklideszi tér pontjait az ideális pontokkal kiegészítő tér a *projektív tér*. Figyeljük meg, hogy nem csupán egyetlen végtelent tudunk így leírni, ugyanis az $[x, y, 0, 0]$ ideális pontok különböznek, ha az első két koordinátát nem arányosan változtatjuk meg! Ebben az esetben az x és az y aránya egy irányt azonosít, amerre az ideális pont van. Egy egyenes mentén mindkét irányban ugyanazt az ideális pontot találjuk a „világ peremén". Az ideális pont tehát összeragasztja az egyenesünk végeit, így, legalábbis topológiai szempontból, körszerűvé teszi azt. A projektív tér egyenesén tehát elmehetünk a világ végére, majd azon is túl, és előbb-utóbb visszajutunk oda, ahonnan elindultunk. Ugye érdekes, de miért kell ezt tudnunk a számítógépes grafikához? Vegyünk egy példát!



3.8. ábra. *A* $[2,2,2]$ *és* $[2,2,-2]$ *pontok közötti szakasz vetülete*

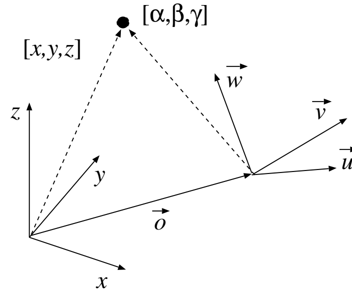
Tegyük fel, hogy a képsík origótól vett távolsága $d=1$, és vetítsük az x,y síkkal párhuzamos képsíkra a $[2,2,2]$ és $[2,2,-2]$ pontok közötti szakaszt (3.8. ábra)! Azt várjuk, hogy elegendő a szakasz két végpontját vetíteni, és a vetületeket összekötni. A 3.10. egyenletbe behelyettesítve a két pont vetületének homogén koordinátái $[2,2,2,2]$ illetve $[2,2,-2,-2]$. Ha Descartes-koordinátákban szeretnénk megkapni az eredményt, el kell végezni a homogén osztást, tehát a vetületek $[1,1,1]$ és $[-1,-1,1]$. A két pontot összekötve már indulnánk is tovább, de mégse tegyük, mert az eredmény rossz! A 3.8. ábrán látható, hogy ha a szakasz minden pontjára külön-külön végeznénk el a vetítést, akkor nem az $[1,1,1]$ és a $[-1,-1,1]$ pontok közötti szakasz pontjait kapnánk meg, hanem azt a két félegyenest, amely az $[1,1,1]$ és a $[-1,-1,1]$ közötti szakaszt egy teljes egyenessé egészítenék ki. A vetítés során a szakasz kifordult magából, és két félegyenes keletkezett belőle. A jelenség neve *átfordulás* (*wrap-around*). A szakasz két végpontját még helyesen vetítettük, a hibát akkor követtük el, amikor a két végpontot összekötöttük, és nem ismertük fel az átfordulást.

A projektív egyenes tulajdonságainak ismeretében az átfordulásban semmi meglepőt sem találhatunk. A projektív egyenes ugyanis olyan, mint egy kör, azaz körbejárható. Miként egy körön felvett két pont sem azonosít egyértelműen egy ívet, a projektív egyenes két pontja közé is két szakasz húzható, amelyek egymás kiegészítései. Az átfordulási probléma azt jelenti, hogy rosszul tippeltük meg a szakaszt. Ilyen nehézségekkel akkor találkozunk, ha a vetített szakasz valamely pontja a vetítés során egy ideális pontra kerül. A Descartes-koordinátarendszerbe visszatérve úgysem tudunk mit kezdeni ezekkel a végtelen távoli pontokkal, ezért a problémát úgy oldhatjuk meg, hogy a vetítés előtt a tárgyból eltávolítjuk azokat a pontokat, amelyek ideális pontra kerülhetnének. A $[2,2,2]$ és $[2,2,-2]$ közötti szakaszt például egy $[2,2,2]$ és $[2,2,\varepsilon]$ pontok közötti szakaszra és egy $[2,2,-2]$ és $[2,2,-\varepsilon]$ pontok közötti szakaszra bontjuk, ahol ε egy elegendően kis érték. A hiányzó, a $[2,2,\varepsilon]$ és $[2,2,-\varepsilon]$ pontok közötti tartomány pedig nagyon messzire (majdnem végtelenbe) vetül, így figyelmen kívül hagyhatjuk.

3.2.8. Koordinátarendszer-váltó transzformációk

Egy adott koordinátarendszerben felírt alakzatra más koordinátarendszerben is szükségünk lehet. Tekintsünk két Descartes-koordinátarendszert és vizsgáljuk meg, hogy a két rendszerben felírt koordináták milyen összefüggésben állnak egymással! Tegyük fel, hogy a régi rendszer bázisvektorai és origója az új koordinátarendszerben rendre $\vec{u}$, $\vec{v}$, $\vec{w}$ és $\vec{o}$:

$$\vec{u} = [u_x, u_y, u_z], \quad \vec{v} = [v_x, v_y, v_z], \quad \vec{w} = [w_x, w_y, w_z], \quad \vec{o} = [o_x, o_y, o_z].$$



3.9. ábra. *Koordinátarendszer-váltó transzformációk*

Vegyünk egy $\vec{p}$ pontot, amelyet az új rendszerben az x, y, z koordináták, a régi $\vec{u}, \vec{v}, \vec{w}$ rendszerben pedig az α, β, γ koordináták azonosítanak! Az új rendszer origójából a $\vec{p}$ pontba két úton is eljuthatunk. Vagy az új rendszer bázisai mentén gyaloglunk x, y, z távolságot, vagy pedig először az $\vec{o}$ vektorral a régi rendszer origójába megyünk, majd innen a régi rendszer bázisai mentén α, β, γ lépést teszünk meg. Az eredmény a $\vec{p}$ pont mindkét esetben:

$$\vec{p} = [x, y, z] = \alpha \cdot \vec{u} + \beta \cdot \vec{v} + \gamma \cdot \vec{w} + \vec{o}.$$

Ezt az egyenletet szintén felírhatjuk homogén lineáris transzformációként:

$$[x, y, z, 1] = [\alpha, \beta, \gamma, 1] \cdot \mathbf{T_c}, \qquad \mathbf{T_c} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ o_x & o_y & o_z & 1 \end{bmatrix}.$$

Ha az $\vec{u}, \vec{v}, \vec{w}$ vektorok is ortonormált rendszert alkotnak, tehát egymásra merőlegesek, és hosszuk egységnyi, akkor a $\mathbf{T_c}$ koordinátarendszer-váltó transzformáció mindig invertálható (az új rendszerből mindig visszatérhetünk a régibe), azaz az $[\alpha, \beta, \gamma]$ hármas is kifejezhető az $[x, y, z]$ segítségével:

$$[\alpha, \beta, \gamma, 1] = [x, y, z, 1] \cdot \mathbf{T_c}^{-1}. \tag{3.11}$$

A $\mathbf{T_c}$ mátrix inverze könnyen előállítható, hiszen ekkor a bal-felső minormátrix *ortonormált mátrix* (a sorvektorai egymásra merőleges egységvektorok), tehát annak inverze egyenlő a transzponáltjával, így:

$$\mathbf{T_c}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -o_x & -o_y & -o_z & 1 \end{bmatrix} \cdot \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

3.2.9. Transzformáció-láncok

A gyakorlatban egy alakzatot nem csupán egyetlen elemi transzformáció módosít, hanem egymást követő transzformációk sorozata. Az egymás utáni transzformációkat a $\mathbf{T_1}, \mathbf{T_2}, \ldots, \mathbf{T_n}$ 4×4-es mátrixok sorozatával írjuk le. Egy $[\vec{r}, 1]$ pontot az első transzformáció az $[\vec{r}, 1] \cdot \mathbf{T_1}$ pontra képez le, amiből a második transzformáció az $([\vec{r}, 1] \cdot \mathbf{T_1}) \cdot \mathbf{T_2}$ pontot állítja elő. Ezt a lépést ismételgetve felírhatjuk a transzformációs lánc kimenetén megjelenő $[X_h', Y_h', Z_h', h]$ pontot:

$$[X_h', Y_h', Z_h', h] = (\ldots(([\vec{r}, 1] \cdot \mathbf{T_1}) \cdot \mathbf{T_2}) \cdot \ldots \cdot \mathbf{T_n}).$$

Mivel a mátrixszorzás asszociatív $((\mathbf{A} \cdot \mathbf{B}) \cdot \mathbf{C} = \mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{C}))$, tehát a zárójelek áthelyezhetők, az eredmény más formában is előállítható:

$$[X_h', Y_h', Z_h', h] = [\vec{r}, 1] \cdot (\mathbf{T_1} \cdot \mathbf{T_2} \cdot \ldots \cdot \mathbf{T_n}) = [\vec{r}, 1] \cdot \mathbf{T},$$

ahol $\mathbf{T}$ az egyes mátrixok szorzata, más szóval *konkatenált*ja. Ennek az egyszerű összefüggésnek óriási jelentősége van, hiszen ez azt jelenti, hogy tetszőlegesen hosszú és bonyolult transzformáció-sorozat helyettesíthető egyetlen transzformációval, azaz a műveletsor egyetlen vektor–mátrix szorzással (16 skalár szorzás és 12 skalár összeadás) megvalósítható.

3.2.10. Program: transzformációs mátrixok

Az affin és lineáris transzformációkat is magában foglaló projektív transzformációkat egy 4×4-es mátrixszal, azaz 16 számmal írhatjuk le. A következő `Matrix` osztály a legfontosabb mátrixműveleteket valósítja meg. A Descartes-koordinátákban megadott `Vector`-ok transzformálásához a három koordinátát egy negyedik, 1 értékű koordinátával egészíti ki, a műveletet homogén koordinátákban számolja, majd az eredményt Descartes-koordinátákká alakítva adja vissza.

```
//===============================================================
class Matrix {
//===============================================================
public:
   float m[4][4];

   void Clear() {  // a mátrixelemek törlése
      memset(&m[0][0], 0, sizeof(m));
   }
   void LoadIdentity() { // a mátrix legyen egységmátrix
      Clear();
      m[0][0] = m[1][1] = m[2][2] = m[3][3] = 1;
   }
   Matrix operator+(const Matrix& mat) { // mátrix összeadás
      Matrix result;
      for(int i = 0; i < 4; i++)
         for(int j = 0; j < 4; j++) result.m[i][j] = m[i][j]+mat.m[i][j];
      return result;
   }
   Matrix operator*(const Matrix& mat) { // mátrixok szorzása
      Matrix result;
      for(int i = 0; i < 4; i++)
         for(int j = 0; j < 4; j++) {
            result.m[i][j] = 0;
            for(int k = 0; k < 4; k++) result.m[i][j] += m[i][k] * mat.m[k][j];
         }
         return result;
   }
   Vector operator*(const Vector& v) {  // Vektor-mátrix szorzás
      float Xh = m[0][0] * v.x + m[0][1] * v.y + m[0][2] * v.z + m[0][3];
      float Yh = m[1][0] * v.x + m[1][1] * v.y + m[1][2] * v.z + m[1][3];
      float Zh = m[2][0] * v.x + m[2][1] * v.y + m[2][2] * v.z + m[2][3];
      float  h = m[3][0] * v.x + m[3][1] * v.y + m[3][2] * v.z + m[3][3];
      return Vector(Xh/h, Yh/h, Zh/h);
   }
};
```

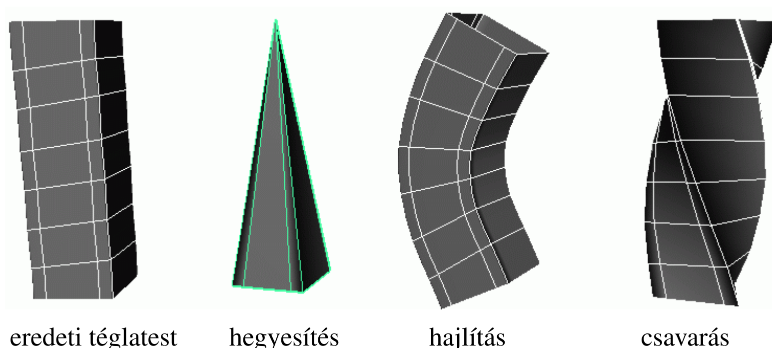
3.2.11. Nemlineáris transzformációk

Az idáig ismertetett transzformációkban az új koordináták a régi koordináták lineáris függvényei voltak. Más oldalról, a transzformációs mátrixok nem függtek a koordinátáktól, így azokat csak konstans értékekkel szorozhatták meg és konstans értéket adhattak hozzá. Ha megengedjük, hogy a transzformációs mátrix elemeiben maguk a koordináták is megjelenjenek, akkor hasznos *nemlineáris transzformáció*kat kapunk. A következőkben néhány példát mutatunk be. A transzformációk hatását a 3.10. ábrán láthatjuk.

- z irányú *hegyesítés* (*tapering*):

  $$x' = \frac{z_{\max} - z}{z_{\max} - z_{\min}} \cdot x, \qquad y' = \frac{z_{\max} - z}{z_{\max} - z_{\min}} \cdot y, \qquad z' = z,$$

  ahol $z_{\max}$ és $z_{\min}$ a tárgy maximális és minimális z koordinátái.



3.10. ábra. *Nemlineáris transzformációk*

- x-tengely körüli, θ szögű, y_0, z_0 középpontú, $[z_0, z_1]$ kiterjedésű *hajlítás* (*bending*):

$$
\begin{aligned}
x' &= x,\\
y' &= \begin{cases} y, \text{ ha } z < z_0,\\ y_0 - (y_0 - y)\cos\left(\frac{z-z_0}{z_1-z_0}\cdot\theta\right), \text{ ha } z_0 \le z \le z_1,\\ y_0 - (y_0 - y)\cos\theta + (z - z_1)\sin\theta, \text{ ha } z > z_1, \end{cases}\\
z' &= \begin{cases} z, \text{ ha } z < z_0,\\ z_0 + (y_0 - y)\cos\left(\frac{z-z_0}{z_1-z_0}\cdot\theta\right), \text{ ha } z_0 \le z \le z_1,\\ z_0 + (y_0 - y)\cos\theta + (z - z_1)\sin\theta, \text{ ha } z > z_1. \end{cases}
\end{aligned}
$$

- z-tengely körüli *csavarás* (*twisting*):

$$
\begin{aligned}
x' &= x\cdot\cos\left(\frac{2\pi(z-z_{\min})}{z_{\max}-z_{\min}}\cdot k\right) - y\cdot\sin\left(\frac{2\pi(z-z_{\min})}{z_{\max}-z_{\min}}\cdot k\right),\\
y' &= x\cdot\sin\left(\frac{2\pi(z-z_{\min})}{z_{\max}-z_{\min}}\cdot k\right) + y\cdot\cos\left(\frac{2\pi(z-z_{\min})}{z_{\max}-z_{\min}}\cdot k\right),\\
z' &= z,
\end{aligned}
$$

ahol a test a teljes z irányú kiterjedése mentén k-szor csavarodik meg.

Ezeket a transzformációkat például akkor érdemes bevetni, ha az alakzatot valamilyen erő hatására, vagy a mozgás hangsúlyozására deformáljuk.

3.3. Görbék

Görbén folytonos vonalat értünk. Egy görbe egy olyan egyenlettel definiálható, amelyet a görbe pontjai elégítenek ki. A 3D görbéket paraméteres formában adhatjuk meg:

$$x = x(t), \quad y = y(t), \quad z = z(t), \quad t \in [0,1]. \tag{3.12}$$

A paraméteres egyenletet a következőképpen értelmezhetjük. Ha egy $[0,1]$ intervallumbeli t értéket behelyettesítünk az $x(t)$, $y(t)$, $z(t)$ egyenletekbe, akkor a görbe egy pontjának koordinátáit kapjuk. A t paraméterrel végigjárva a megengedett intervallumot az összes pontot meglátogatjuk.

Például egy $\vec{r}_1 = [x_1, y_1, z_1]$-tól $\vec{r}_2 = [x_2, y_2, z_2]$-ig tartó *3D szakasz* egyenlete:

$$x = x_1 \cdot (1-t) + x_2 \cdot t, \quad y = y_1 \cdot (1-t) + y_2 \cdot t, \quad z = z_1 \cdot (1-t) + z_2 \cdot t, \quad t \in [0,1],$$

illetve vektoros formában:

$$\vec{r}(t) = \vec{r}_1 \cdot (1-t) + \vec{r}_2 \cdot t.$$

A szakasz egyenlete, egyszerűsége ellenére, alkalmat teremt általános következtetések levonására. Figyeljük meg, hogy a szakasz pontjait úgy állítottuk elő, hogy a szakasz végpontjait a paraméterértéktől függően súlyoztuk, majd a részeredményeket összeadtuk:

$$\vec{r}(t) = \vec{r}_1 \cdot B_1(t) + \vec{r}_2 \cdot B_2(t),$$

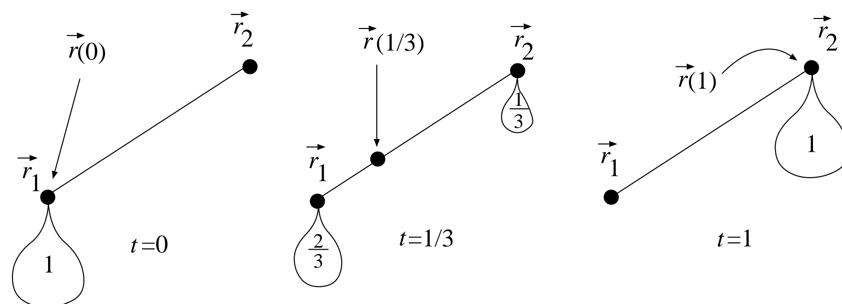
ahol $B_1(t) = 1 - t$ és $B_2(t) = t$. A súlyokat t szerint a B_1, B_2, függvények adják meg, így, a vezérlőpontok mellett, ezek felelősek a görbe alakjáért. Fontosságukat a nevük is kifejezi, ők a *bázisfüggvény*ek. A görbét úgy is elképzelhetjük, hogy az $\vec{r}_1$ pontba $B_1(t)$, az $\vec{r}_2$ pontba pedig $B_2(t)$ súlyt teszünk, és tekintjük ezen mechanikai rendszer súlypontját (lásd a 3.2. egyenletet):

$$\vec{r}_c(t) = \frac{\vec{r}_1 \cdot B_1(t) + \vec{r}_2 \cdot B_2(t)}{B_1(t) + B_2(t)}.$$

Mivel a B_1, B_2 bázisfüggvények összege mindig 1, a tört nevezője eltűnik, a súlypont pedig éppen a görbe adott pontját azonosítja:

$$\vec{r}_c(t) = \vec{r}_1 \cdot B_1(t) + \vec{r}_2 \cdot B_2(t) = \vec{r}(t),$$

Ahogy a t végigfut a $[0,1]$ intervallumon, az első végpont súlya ($B_1(t) = 1 - t$) egyre csökken, mialatt a másiké egyre növekszik ($B_2(t) = t$), és így a súlypont szépen átsétál az egyik végpontból a másikba (3.11. ábra). Mivel a $t = 0$ értéknél a teljes súly az egyik végpontban van ($B_1(0) = 1, B_2(0) = 0$), a szakasz itt átmegy az első végponton, hasonlóképpen a $t = 1$-nél átmegy a másik végponton is.



3.11. ábra. *A szakasz egyenlete és a súlypont analógia*

A szakasznál megismert elveket általános görbék előállításához is használhatjuk. A felhasználó a görbe alakját néhány *vezérlő pont*tal (*control point*) definiálja, amelyekből tényleges görbét úgy kapunk, hogy a vezérlőpontokba t paramétertől függő súlyokat teszünk és adott t értékre a mechanikai rendszer súlypontját tekintjük a görbe adott pontjának. A súlyokat úgy kell megválasztani, hogy más és más t értékekre más vezérlőpontok domináljanak, így a görbe meglátogatja az egyes pontok környezetét. Ha egy pontba nagyobb súlyt teszünk, a rendszer súlypontja közelebb kerül az adott ponthoz. Így a vezérlőpontokat kis mágneseknek képzelhetjük el, amelyek maguk felé húzzák a görbét.

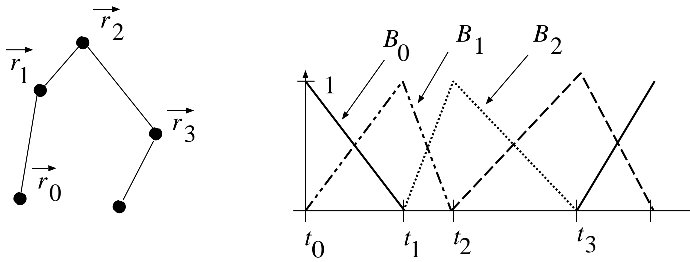
Ha van olyan t paraméterérték, ahol az egyik vezérlőpontot kivéve a többi mind zérus súlyt kap, akkor ennél a t értéknél a görbe átmegy az adott vezérlőponton. Ha ilyen paraméterérték minden vezérlőpontra található, a görbénk mindegyiken átmegy, és a görbe *interpolációs*. Ha nincs, akkor a görbe általában csak megközelíti a vezérlő pontokat, a görbénk tehát csupán *approximációs*.

3.3.1. A töröttvonal

Az első „általános” görbénket, a *töröttvonal*at (*polyline*) a szakasz fogalom kiterjesztésével alkotjuk meg. A legkézenfekvőbb megoldás ugyanis, ha a tervező által megadott $\vec{r}_0, \ldots, \vec{r}_{m-1}$ vezérlőpont-sorozatot szakaszokkal kötjük össze (3.12. ábra).

Ezt az ötletet a következőképpen fordíthatjuk le a bázisfüggvények nyelvére. Rendeljünk az $\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_{m-1}$ vezérlőpontokhoz egy $t_0 \leq t_1 \leq \ldots \leq t_{m-1}$ paraméter sorozatot (*knot point*) és tűzzük ki célul azt, hogy a görbe t_i értéknél az $\vec{r}_i$ pontot interpolálja, t_i és t_{i+1} között pedig $\vec{r}_i$ és $\vec{r}_{i+1}$ közötti szakaszon fusson végig! A szakasz példáján láttuk, hogy ez akkor következik be, ha a t_i és t_{i+1} között $\vec{r}_i$ súlya 1-ről lineárisan csökken zérusra, az $\vec{r}_{i+1}$ súlya pedig éppen ellentétesen nő, mialatt a többi vezérlőpont súlya zérus, így nem szólhatnak bele a görbe alakulásába. A t_{i+1} paraméterértéken túl az $\vec{r}_{i+1}$ és $\vec{r}_{i+2}$ bázisfüggvényei lesznek zérustól különbözők, az összes többi pont súlya pedig

zérus.



3.12. ábra. *A töröttvonal és bázisfüggvényei*

A 3.12. ábrán ennek megfelelően ábrázoltuk a görbét és a bázisfüggvényeket. Az egyes bázisfüggvények „sátor” alakúak, egy $\vec{r}_i$ pontnak csak a $[t_{i-1}, t_i]$ intervallumban van egyre növekvő súlya, valamint a $[t_i, t_{i+1}]$ intervallumban egyre csökkenő súlya. Más oldalról, egy pont, a töröttvonal kezdetét és végét kivéve, két szakasz kialakításában vesz részt, az egyiknek a végpontja, a másiknak pedig a kezdőpontja.

A töröttvonal folytonos, de meglehetősen szögletes, ezért alkatrészek tervezéséhez, vagy például egy hullámvasút sínjének kialakításához nem használható (az alkatrész ugyanis eltörne, a sín törési pontjában pedig az utasok kirepülnének a kocsikból). Simább görbékre van szükségünk, ahol nem csupán a görbe, de annak magasabb rendű deriváltjai is folytonosak.

3.3.2. Bézier-görbe

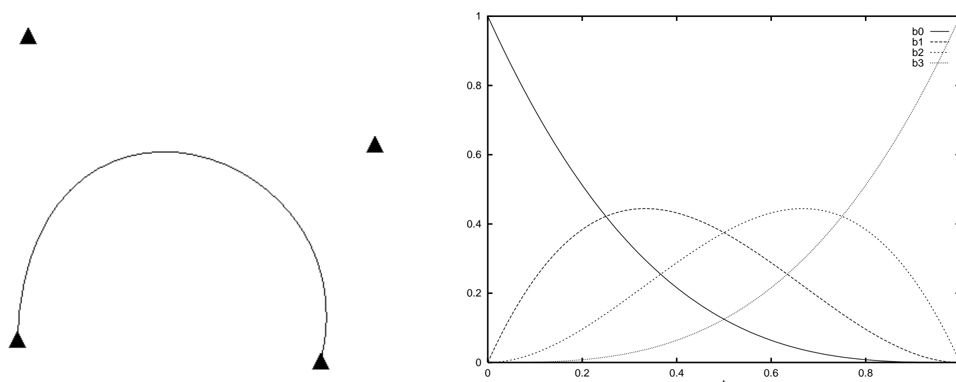
Sima görbék előállításához magasabb rendben folytonos bázisfüggvényeket kell alkalmazni [41]. Mivel a bázisfüggvényekkel súlyozott vektorok összege akkor jelöli ki a rendszer súlypontját, ha az összsúly 1, ilyen függvényosztályokban érdemes keresgélni. A Renault gyár Pierre Bézier nevű konstruktőre az 1960-as években, a *Bernstein-polinom*okat javasolta bázisfüggvényeknek, amelyeket az $1^m = (t + (1-t))^m$ *binomiális tétel* szerinti kifejtésével kapunk:

$$(t + (1-t))^m = \sum_{i=0}^{m} \binom{m}{i} t^i \cdot (1-t)^{m-i}.$$

A *Bézier-görbe* bázisfüggvényei ezen összeg tagjai ($i = 0, 1, \ldots, m$):

$$B_{i,m}^{\text{Bezier}}(t) = \binom{m}{i} t^i \cdot (1-t)^{m-i}. \tag{3.13}$$

A definícióból rögtön adódik, hogy $\sum_{i=0}^{m} B_{i,m}^{\text{Bezier}}(t) = 1$, és ha $t \in [0, 1]$, akkor $B_{i,m}^{\text{Bezier}}(t)$ nem negatív. Mivel $B_{0,m}^{\text{Bezier}}(0) = 1$ és $B_{m,m}^{\text{Bezier}}(1) = 1$, a görbe átmegy az első és utolsó



3.13. ábra. *Bézier-approximáció és bázisfüggvényei ($m = 3$)*

vezérlőponton, de általában nem megy át a többi vezérlőponton. Mint az könnyen igazolható, a görbe kezdete és vége érinti a vezérlőpontok által alkotott sokszöget (3.13. ábra).

A Bézier-görbe bázisfüggvényei között fennáll a következő rekurzív összefüggés:

$$B^{\text{Bezier}}_{i+1,m}(t) = t \cdot B^{\text{Bezier}}_{i,m-1}(t) + (1-t) \cdot B^{\text{Bezier}}_{i+1,m-1}(t),$$

amelyet a Bernstein-polinomokkal történő helyettesítéssel igazolhatunk:

$$t \cdot B^{\text{Bezier}}_{i,m-1}(t) + (1-t) \cdot B^{\text{Bezier}}_{i+1,m-1}(t) =$$

$$t \cdot \binom{m-1}{i} t^i (1-t)^{m-i-1} + (1-t) \cdot \binom{m-1}{i+1} t^{i+1} (1-t)^{m-i-2} =$$

$$\left(\binom{m-1}{i} + \binom{m-1}{i+1}\right) \cdot t^{i+1}(1-t)^{m-i-1} = \binom{m}{i+1} \cdot t^{i+1}(1-t)^{m-i-1} = B^{\text{Bezier}}_{i+1,m}(t).$$

Ez azt jelenti, hogy a bázisfüggvények lineáris átlagolásával a magasabb fokú bázisfüggvényeket kapjuk meg. Az átlagolást akár geometriai módszerrel is elvégezhetjük, ami a Bézier-görbe *de Casteljau-módszer*rel történő felrajzolásához vezet. Tegyük fel, hogy a Bézier-görbe t paraméterértéknél felvett pontját szeretnénk megszerkeszteni (3.14. ábra). Hacsak két vezérlőponttal rendelkezne a görbe, akkor a megfelelő pontot úgy kaphatjuk meg, hogy a két $\vec{r}_0, \vec{r}_1$ pontot összekötő szakaszon megkeressük a

$$\vec{r}^{\,(1)}_0 = \vec{r}_0 \cdot (1-t) + \vec{r}_1 \cdot t = \vec{r}_0 \cdot B^{\text{Bezier}}_{0,1}(t) + \vec{r}_1 \cdot B^{\text{Bezier}}_{1,1}(t)$$

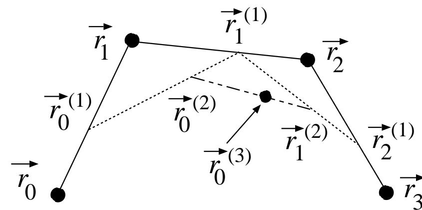
pontot. Végezzük el ezt a műveletet az összes egymást követő vezérlőpont párra, amelynek eredményeként $m-1$ újabb pont adódik. Ezeket megint összeköthetjük szakaszokkal, amelyeken kijelölhetjük a $t:(1-t)$ aránypárnak megfelelő pontot. A súlyfüggvények imént levezetett tulajdonságai alapján, az első így előálló pont:

$$\vec{r}_0^{(2)} = \vec{r}_0^{(1)} \cdot (1-t) + r_1^{(1)} \cdot t =$$

$$\vec{r}_0 \cdot (1-t) \cdot B_{0,1}^{\text{Bezier}}(t) + \vec{r}_1 \left(t \cdot B_{0,1}^{\text{Bezier}}(t) + (1-t) \cdot B_{1,1}^{\text{Bezier}}(t)\right) + \vec{r}_1 t \cdot B_{1,1}^{\text{Bezier}}(t) =$$

$$\vec{r}_0 B_{0,2}^{\text{Bezier}}(t) + \vec{r}_1 B_{1,2}^{\text{Bezier}}(t) + \vec{r}_2 B_{2,2}^{\text{Bezier}}(t).$$

Az eljárást rekurzív módon folytatva az m. lépésben éppen a Bézier-görbe t értéknél felvett pontjához jutunk.



3.14. ábra. *A de Casteljau-algoritmus*

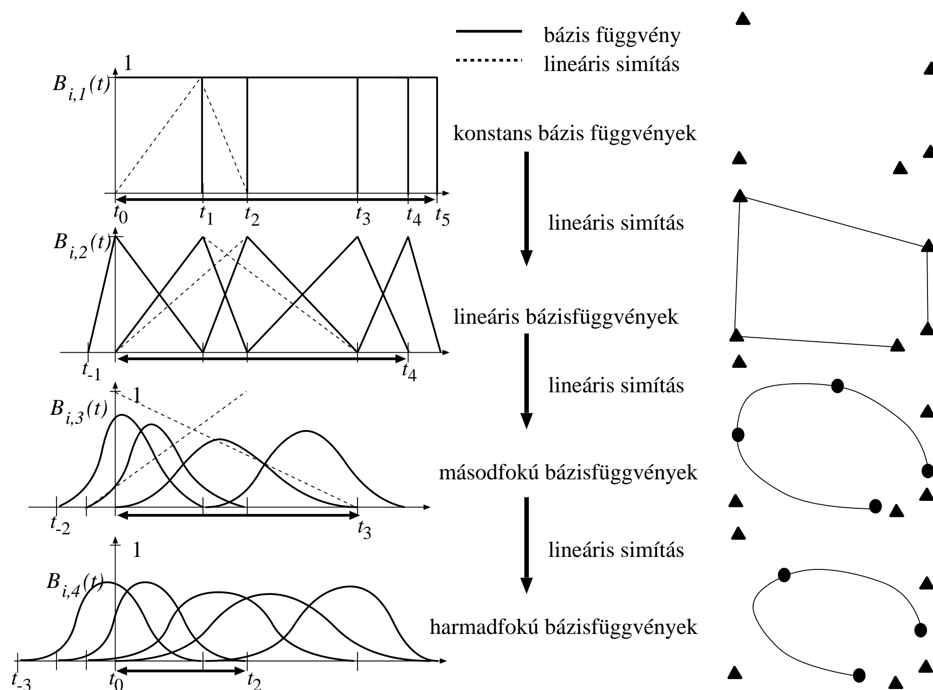
A Bézier-görbe szép görbült, m-szer deriválható, amiért viszont komoly árat kell fizetnünk. Az egyes bázisfüggvények, a végpontokat kivéve, a teljes paramétertartományban pozitívak (3.13. ábra), azaz egy vezérlőpont szinte minden helyen érezteti a hatását. A görbe tehát nem vezérelhető lokálisan, ami nehézkessé teszi a finomhangolását, hiszen egy vezérlőpont módosítása nemcsak a vezérlőpont környezetében, hanem attól messze is megváltoztatja a görbét. Másrészt, ha sok vezérlőpontunk van, a görbénk egyre erősebben approximációs jellegű lesz, azaz egyre kevésbé fogja megközelíteni a vezérlőpontokat.

3.3.3. B-spline

A töröttvonallal szemben a szögletességet, a Bézier-görbével szemben pedig az erősen approximációs jelleget és a lokális vezérelhetőség hiányát hánytorgattuk fel. A görbültség és a lokális vezérelhetőség nyilván ellentmondó követelmények, amelyek közül egyet-egyet a töröttvonal a görbültség, a Bézier-görbe pedig a lokális vezérelhetőség figyelmen kívül hagyásával elégített ki.

Ezen fejezet görbéje, a *B-spline*, mindkét elvárást szem előtt tartja, és közöttük észszerű kompromisszumot köt. Célunk tehát az, hogy a töröttvonal szögletességén javítsunk anélkül, hogy a lokális vezérelhetőségről teljesen lemondanánk. A teljesség

kedvéért még egy szintet visszalépünk és nem is a töröttvonalról, hanem az $\vec{r}_0,\ldots,\vec{r}_{m-1}$ vezérlőpontokról indulunk el. A vezérlőpontok maguk is egy véges számú pontból álló paraméteres „görbeközelítésnek” tekinthetők, ha feltételezzük, hogy bármely $t_i \leq t < t_{i+1}$ paraméterértékre a görbe éppen az $\vec{r}_i$ vezérlőpontban van. A paraméterértékek $[t_0,t_1,\ldots,t_{m-2},t_{m-1}]$ sorozatát *csomópontvektor*nak nevezzük.



3.15. ábra. *A B-spline bázisfüggvények létrehozása*

Ez a közelítés azonban nem is ad folytonos görbét, hiszen a „görbe” diszkrét paraméterpontokban ugrik az egyik vezérlőpontról a következőre, amin úgy segíthetünk, hogy két egymást követő bázisfüggvényt lineáris súlyozással összeadunk (3.15. ábra). Az első bázisfüggvényt a $t_i \leq t < t_{i+1}$ értelmezési tartományában a lineárisan növekvő $(t-t_i)/(t_{i+1}-t_i)$ kifejezéssel szorozzuk, így a hatását lineárisan zérusról egyre növeljük. A következő bázisfüggvényt pedig annak $t_{i+1} \leq t < t_{i+2}$ értelmezési tartományában lineárisan csökkenő $(t_{i+2}-t)/(t_{i+2}-t_{i+1})$ függvénnyel skálázzuk. Az így súlyozott bázisfüggvényeket összeadva kapjuk a magasabb rendű változat sátorszerű bázisfüggvényeit, amelyek a töröttvonalnál megismertekkel egyeznek meg. Figyeljük meg, hogy míg az eredeti bázisfüggvények egy-egy intervallumon voltak pozitívak, a sátorszerű, simított változatban ez már két-két intervallumra igaz.

A szomszédos bázisfüggvények összesimítása azonban felvet egy gondot. Ha kez-

detben m darab vezérlőpontunk és így m darab állandó bázisfüggvényünk volt, a szomszédos párok száma csak $m-1$ lesz, és a két szélen lévő állandó bázisfüggvényt a belső függvényektől eltérően csak egyszer tudjuk átlagolni. Mivel m vezérlőpontunk van, m darab bázisfüggvényre van szükségünk a simítás után is. A hiányzó bázisfüggvényt megkaphatjuk, ha gondolatban még egy t_{-1} paraméterértéket veszünk fel a t_0 elé, mert ekkor a legelső állandó bázisfüggvényt is kétszer tudjuk átlagolni. Ezzel a görbe eleje rendben is volna, de mit tegyünk a legutolsó bázisfüggvénnyel. Ha oda is egy újabb t_m paraméterértéket tennénk, akkor már $m+1$ darab bázisfüggvényünk lenne, ami éppen eggyel több a szükségesnél. A gordiuszi csomót úgy vágjuk el, hogy a görbét ezentúl csak a $[t_0, t_{m-2}]$ tartományban értelmezzük, szemben a korábbi $[t_0, t_{m-1}]$ tartománnyal. A $[t_0, t_{m-2}]$ tartományban ugyanis elmondhatjuk, hogy éppen m darab bázisfüggvényt találunk, amelyek mindegyikét a két szomszédos konstans bázisfüggvény összemosásával kaptunk meg. Az új t_{-1} csomóértéknek egyelőre nincs hatása a görbére.

A töröttvonal tehát előállítható a vezérlőpontok lineáris „simításával”. A fő problémánk a töröttvonallal az volt, hogy szögletes, tehát érdemes a simítási lépést újból megismételni. Vegyünk ismét két egymást követő lineáris, sátorszerű bázisfüggvényt, és az elsőt szorozzuk meg az értelmezési tartományában lineárisan növekvő $(t-t_i)/(t_{i+2}-t_i)$ függvénnyel, a következőt pedig annak az értelmezési tartományában lineárisan csökkenő $(t_{i+3}-t)/(t_{i+3}-t_{i+1})$ függvénnyel! Felhívjuk a figyelmet, hogy ez nem ugyanaz a simítás, amit az állandó bázisfüggvényekre végeztünk, hiszen a sátorszerű bázisfüggvények már két intervallumra terjednek szét, így a lineáris súlyozófüggvények is két intervallumon keresztül érik el a zérusról az 1 értéket. Az eredményeket összeadva megkapjuk a még simább bázisfüggvényeket. A lineáris függvényekből ezzel már három intervallumra kiterjedő másodfokú bázisfüggvényeket készíthetünk, amelyek nemcsak folytonosak, de folytonosan deriválhatók is lesznek. Most is felmerül az első és utolsó bázisfüggvény különleges helyzete, hiszen azok nem tudnak tovább terjeszkedni, mert az idáig tekintett paramétertartomány a t_{-1}-nél illetve a t_m-nél véget ér. Az egységes kezelés érdekében ezért egy újabb t_{-2} paraméterértékeket veszünk fel a t_{-1} elé, a görbe hasznos tartományát pedig a $[t_0, t_{m-3}]$ intervallumra korlátozzuk.

A t_i csomóértékekről idáig semmit sem mondtunk azon túl, hogy egy nem csökkenő sorozatot alkotnak. Ez nem is véletlen, hiszen a lineáris közelítésig ezek értéke nem befolyásolja a görbe alakját. A másodfokú görbeközelítés alakjára azonban már hatnak a csomóértékek. Tételezzük fel, hogy a $[t_i, t_{i+1}]$ tartomány lényegesen kisebb a $[t_{i+1}, t_{i+2}]$ tartománynál. Amikor a $[t_i, t_{i+2}]$-beli sátorszerű bázisfüggvényt lineárisan súlyozzuk, akkor az egyik lineáris súlyozógörbe a t_i-ben 1 értékű és t_{i+2}-ig zérusra csökken. Mivel a t_i és a t_{i+1} közel van egymáshoz, a lineáris súlyozógörbe még a t_{i+1}-nél, azaz a sátor 1 értékű csúcsánál is 1-hez közeli. Ezért a súlyozott bázisfüggvény is 1-hez közeli értéket vesz itt fel. Az approximációs görbénk tehát az $\vec{r}_{i+1}$ vezérlőpont közelében halad el. Ezt a jelenséget általában is megfogalmazhatjuk. Ha azt akarjuk, hogy egy vezérlőpont erősen magához rántsa a görbét, a vezérlőpont legközelebbi paramétertartományát ki-

csire kell venni. A szélsőséges esetben az intervallumot választhatjuk zérusra is, amikor a lineáris súlyozás maximuma éppen a degenerált sátor csúcsával esik egybe, ezért a súlyozott bázisfüggvény is 1 értékű lesz, tehát még a másodfokú görbe is interpolálja a megfelelő vezérlőpontot. Óvatosan kell bánnunk a zérus hosszúságú intervallumokkal, hiszen itt a bázisfüggvények lineáris súlyozása 0/0 jellegű eredményt ad. Az elmondottak szerint akkor járunk el helyesen, ha a számítások során keletkező 0/0 törteket 1-nek tekintjük.

Ha még a másodfokú közelítés simaságával sem vagyunk elégedettek, a két egymás utáni másodfokú bázisfüggvényt lineáris súlyozás után ismét összevonhatjuk, amely harmadfokú, kétszer folytonosan deriválható eredményt ad. A határokat megint úgy kezeljük, hogy egy t_{-3} paramétert veszünk fel, a görbét pedig a $[t_0, t_{m-4}]$ tartományban használjuk. Szükség esetén a simítási lépés a harmadfokú görbéken túl is tetszőleges szintig folytatható.

Figyeljük meg, hogy az első (konstans) közelítésben a görbe egyetlen pontjára az m vezérlőpontból csupán egyetlen hatott, mégpedig úgy, hogy a felelős pont szerepét az intervallum határokon mindig más vezérlőpont vette át. A második (lineáris) közelítésben már két vezérlőpont uralkodott a görbe felett úgy, hogy az intervallum határokon a pár első tagja kikerült a szerepéből, a második tag elsővé lépett elő, és a következő vezérlőpont kapta meg a második tag szerepét. A harmadik közelítésben már vezérlőpont hármasok határozzák meg a görbét egy adott paraméterértékre (általában a k. szinten pedig k elemű vezérlőpont csoportok). A görbe azon részét, amit ugyanazon vezérlőpontok uralnak, *szegmens*nek nevezzük. Mivel a vezérlőpontok az intervallum határoknál cserélnek szerepet, egy szegmens egyetlen intervallumhoz tartozik. A 3.15. ábrán a görbéken pontok mutatják a szegmenshatárokat. A szintek növelésével, a hasznos tartomány intervallumai, és így a szegmensek száma csökken.

A tárgyalt módszerben megengedtük, hogy az egymást követő vezérlőpont párok közötti paramétertartomány eltérő legyen, ezért a kapott görbét *nem egyenletes B-spline*-nak (*Non-Uniform-B-spline*) vagy röviden *NUBS*-nak nevezzük (egyesek azt állítják, hogy a NUBS inkább a „Nobody Understands B-Splines” rövidítése).

A *k-ad fokú B-spline* bázisfüggvényeinek előállításakor egy $(k-1)$-ed fokú bázisfüggvényt kétszer használunk fel, egyszer lineárisan csökkenő, egyszer pedig lineárisan növekvő súllyal. Mivel kezdetben a bázisfüggvények összege minden t paraméterértékre egységnyi, és a két lineáris súlyozás összege is egy, mindvégig érvényben marad az a tulajdonság, hogy a bázisfüggvények összege egy. Ez valóban fontos feltétel, hiszen ez biztosítja, hogy a görbe a vezérlőpontok konvex burkában halad, tehát görbénk valóban arra megy, amerre a vezérlőpontok kijelölik. Ha nem vettünk volna fel minden simítási lépésnél újabb csomóértékeket, és nem szűkítettük volna a hasznos tartományt egy intervallummal, akkor ez a követelmény a görbe elején és végén sérült volna, hiszen a legelső és legutolsó bázisfüggvényeket nem tudtuk volna kétszer átlagolni. Tehát éppen a t_{-1}, t_{-2} stb. csomóértékeknek köszönhetjük azt, hogy a t_0 és t_{m-k} közötti hasznos

tartományban a bázisfüggvények összege mindig 1. A t_0 előtt, illetve a t_{m-k} után ez a feltétel nem teljesül, de ez nem is fontos, ugyanis a görbe rajzolásához csak a t_0 és t_{m-k} intervallumot vesszük figyelembe. A kiegészítő t_{-1}, t_{-2} stb. illetve a t_{m-k+1}, t_{m-k+2} csomóértékek persze megjelennek a bázisfüggvények képleteiben (az első és utolsó $k-2$ darab bázisfüggvényre vannak hatással, ahol k a szintek száma), ezért a megválasztásuk módosítja a görbét, pontosabban annak kezdeti és befejező szakaszát.

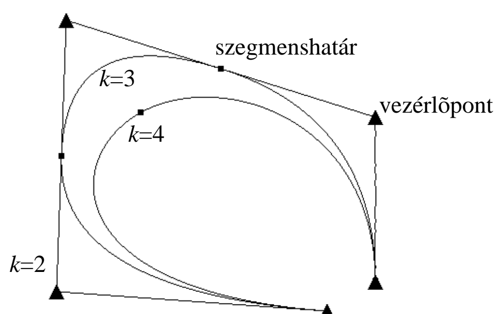
A programnyelvi implementációban nehézséget jelenthet az, hogy a paraméterértékeket negatív indexszel láttuk el. Ezért a görbe végső szintszámának megfelelően a paramétereket átsorszámozzuk úgy, hogy az index mindig zérusról induljon. A *Cox-deBoor rekurzió*s formulák ezen feltételezéssel élnek, és a k-adik szint i-edik bázisfüggvényeit az előző szint bázisfüggvényeiből fejezik ki:

$$B_{i,1}^{\text{NUBS}}(t) = \begin{cases} 1, & \text{ha } t_i \leq t < t_{i+1}, \\ 0, & \text{különben}, \end{cases}$$

$$B_{i,k}^{\text{NUBS}}(t) = \frac{(t-t_i)B_{i,k-1}^{\text{NUBS}}(t)}{t_{i+k-1}-t_i} + \frac{(t_{i+k}-t)B_{i+1,k-1}^{\text{NUBS}}(t)}{t_{i+k}-t_{i+1}}, \qquad \text{ha } k>1, \quad \left(\tfrac{0}{0}=1\right).$$

A következő osztály egy általános NUBS görbét valósít meg:

```
//===============================================================
class NUBSCurve {
//===============================================================
   Vector * r;          // vezérlőpontok tömbje
   float *  t;          // csomóvektor
   int      m;          // pontok száma
   int      K;          // szintek száma (fokszám + 1)
public:
   float B(int i, int k, float tt) { // k-szintű i. bázisfüggvény
      if (k == 1)  {                 // triviális eset vizsgálata
         if (i < m - 1) {
            if (t[i] <= tt && tt < t[i+1])  return 1; else return 0;
         } else {
            if (t[i] <= tt)  return 1; else return 0;
         }
      }
      float b1, b2;
      if (t[i+k-1]-t[i] > 0.00001) b1 = (tt-t[i]) / (t[i+k-1]-t[i]);
      else                         b1 = 1.0;       // Itt: 0/0 = 1
      if (t[i+k]-t[i+1] > 0.00001) b2 = (t[i+k]-tt) / (t[i+k]-t[i+1]);
      else                         b2 = 1.0;       // Itt: 0/0 = 1
      return (b1 * B(i, k-1, tt) + b2 * B(i+1, k-1, tt) ); // rekurzió
   }

   Vector Curve(float t) {              // a görbe egy adott pontja
      Vector rt(0,0,0);
      for(i = 0; i < m; i++) rt += r[i] * B(i, K, t);
      return rt;
   }
};
```



3.16. ábra. *A NUBS magasabb szinten kevesebb szegmensből áll és jobban görbül*

Egy k szintű NUBS bázisfüggvényei $(k-1)$-ed fokú polinomok, amelyek k intervallumra terjeszkednek szét. Vegyük észre, hogy mialatt a fokszámot növeljük, a görbe simasága nő, de a lokális vezérelhetősége romlik. Amíg a töröttvonal sátras bázisfüggvényei két intervallumban zérustól különbözőek, addig a másodfokú bázisfüggvények már három, a harmadfokúak pedig már négy intervallumban lesznek zérustól különbözőek, tehát egy vezérlőpont egyre nagyobb részén érezteti a hatását (3.16. ábra). Amíg a szintek száma kisebb, mint a vezérlőpontok száma, a bázisfüggvények a görbe egy részére hatnak csupán, tehát a görbénk *lokálisan vezérelhető* lesz. A görbe egynél nagyobb fokszámú bázisok esetén elveszti interpolációs jellegét. Egy vezérlőpontra az interpolációs feltételt kierőszakolhatjuk, ha a hozzá tartozó csomóértékek távolságát zérusra választjuk. Ehhez a másodfokú bázis esetén egy, a harmadfokúnál pedig két egymást követő intervallumot kell zérus hosszúságúra venni.

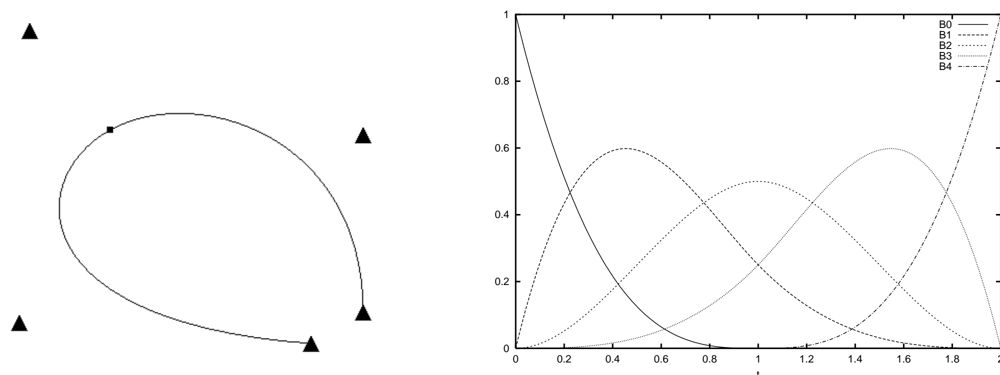
Idáig nem beszéltünk arról, hogy hogyan kell a kiegészítő t_{-1}, t_{-2} stb. illetve a hasznos tartományból kicsúszó t_{m-k}, t_{m-k+1} csomóértékeket felvenni, habár elismertük, hogy azok a görbe alakjára hathatnak [66]. A következőkben a gyakorlatban leggyakoribb harmadfokú esettel és három csomóérték választási eljárással foglalkozunk.

A végpontokon átmenő NUBS

Az első eljárás a pótlólagosan felvett csomóértékeket az első, illetve az utolsó csomóértékkel megegyezően veszi fel, azaz az újabb intervallumok hossza mindig zérus, tehát a NUBS csomóvektora a következő lesz:

$$[t_0, t_0, t_0, t_0, t_1, \ldots, t_{m-4}, t_{m-4}, t_{m-4}, t_{m-4}].$$

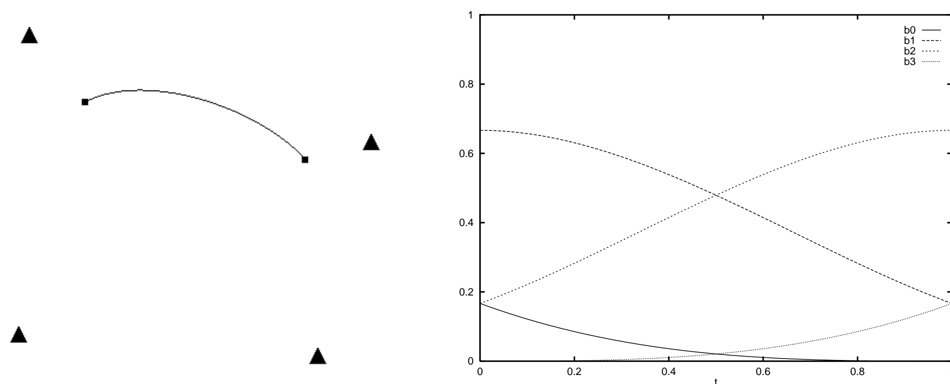
Mivel a harmadfokú NUBS interpolálja azokat a vezérlőpontokat, amelyeknek megfelelő két legközelebbi csomóintervallum hossza zérus (azaz három csomóérték közös), az így kialakított görbe mindig átmegy a legelső és a legutolsó vezérlőponton.



3.17. ábra. *A végpontokon átmenő harmadfokú NUBS és bázisfüggvényei*

Egyenletes B-spline

Az *egyenletes B-spline*ban a csomóértékek közötti távolság egységnyi.



3.18. ábra. *B-spline approximáció és bázisfüggvények*

A harmadfokú egyenletes B-spline csomóvektora a

$$[-3,-2,-1,0,1,2,3,4],$$

a görbe hasznos tartománya pedig a $[0,1]$. Ebben a tartományban a bázisfüggvényeket a Cox-deBoor formulákból analitikusan is meghatározhatjuk. Az első közelítésben $B_{3,1}(t)=1$ a $t=[0,1]$-ben, az összes többi bázisfüggvény zérus. A második közelítést

az első lineáris súlyozásával kapjuk, azaz $B_{3,2}(t)$-höz a $B_{3,1}(t)$-t t-vel, a $B_{2,2}(t)$-höz pedig a $B_{3,1}(t)$-t $(1-t)$-vel kell súlyozni, amiből azt kapjuk, hogy

$$B_{2,2}(t) = 1-t, \qquad B_{3,2}(t) = t.$$

A harmadik szintű változatokhoz újabb lineáris interpolációt végzünk, de most már a lineáris súlyozófüggvények két intervallumot fognak át, tehát képletük: $(1-t)/2$, $(1+t)/2$, $(2-t)/2$ és $t/2$. A másodfokú súlyfüggvények a $[0,1]$-ben tehát a következő alakúak:

$$\begin{aligned}
B_{1,3}(t) &= \frac{(1-t)^2}{2}, \\
B_{2,3}(t) &= \frac{(1-t)(1+t)+t(2-t)}{2} = \frac{1+2t(1-t)}{2}, \\
B_{3,3}(t) &= \frac{t^2}{2}.
\end{aligned}$$

Végül a negyedik szintű görbében az újabb lineáris súlyozás súlyfüggvényeinek már három intervallumra van szükségük, hogy 0-ról 1-re emelkedjenek vagy süllyedjenek: $(1-t)/3$, $(2+t)/3$, $(2-t)/3$, $(1+t)/3$, $(3-t)/3$ és $t/3$. Ezek alkalmazásával a harmadfokú egyenletes B-spline bázisfüggvényeihez jutunk:

$$\begin{aligned}
B_{0,4}(t) &= \frac{(1-t)^3}{6}, \\
B_{1,4}(t) &= \frac{(1-t)^2\cdot(2+t)+(1+2t(1-t))\cdot(2-t)}{6} = \frac{1+3(1-t)+3t(1-t)^2}{6}, \\
B_{2,4}(t) &= \frac{1+2t(1-t)\cdot(t+1)+t^2\cdot(3-t)}{6} = \frac{1+3t+3t^2(1-t)}{6}, \\
B_{3,4}(t) &= \frac{t^3}{6}.
\end{aligned}$$

Bézier-görbe mint a NUBS speciális esete

Végül vizsgáljuk meg, hogy milyen harmadfokú NUBS görbe tartozik a

$$[0,0,0,0,1,1,1,1]$$

csomóvektorhoz. Harmadfokú esetben a görbe hasznos tartománya a $[0,1]$. Az előző alfejezethez hasonlóan a Cox-deBoor formulákat alkalmazzuk. Az első közelítésben $B_{3,1}(t) = 1$ a $t = [0,1]$-ben, az összes többi bázisfüggvény zérus. A második közelítést az elsőrendű lineáris súlyozásával kapjuk:

$$B_{2,2}(t) = 1-t, \qquad B_{3,2}(t) = t.$$

A harmadik változathoz újabb lineáris interpolációt végzünk. Mivel most a csomóértékek távolsága zérus, a két intervallumra szétterülő lineáris súlyozó függvények továbbra is a t illetve az $(1-t)$ lesznek:

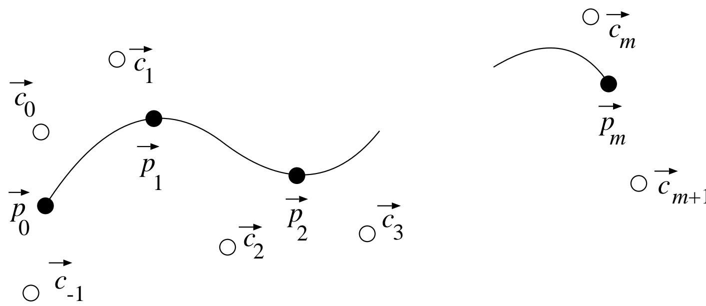
$$B_{1,3}(t) = (1-t)^2, \quad B_{2,3}(t) = (1-t)t + t(1-t) = 2t(1-t) \quad B_{3,3}(t) = t^2.$$

Végül a negyedik szinten ugyanilyen súlyozásra van szükségünk:

$$\begin{aligned} B_{0,4}(t) &= (1-t)^3, \\ B_{2,4}(t) &= (1-t)^2 t + 2t(1-t)^2 = 3t(1-t)^2, \\ B_{3,4}(t) &= 2t^2(1-t) + t^2(1-t) = 3t^2(1-t), \\ B_{3,4}(t) &= t^3. \end{aligned}$$

Ezek pedig a jól ismert Bernstein-polinomok, tehát éppen a Bézier-görbéhez jutottunk. A fenti konstrukció során erre már akkor gyanakodhattunk, amikor megállapítottuk, hogy az előző szint bázisfüggvényeit mindig a t illetve az $(1-t)$ függvényekkel kell simítani, ugyanis ez éppen a *de Casteljau-algoritmus*.

3.3.4. B-spline görbék interpolációs célokra



3.19. ábra. *A B-spline interpoláció*

A harmadfokú B-spline csupán approximálja a vezérlőpontjait, de ez nem jelenti azt, hogy interpolációs célra ne lenne használható. Tegyük fel, hogy egy olyan görbét keresünk, amely a $t_0 = 0, t_1 = 1, \ldots, t_m = m$ paraméterértékeknél éppen a $\vec{p}_0, \vec{p}_1, \ldots, \vec{p}_m$ pontokon megy át (3.19. ábra). Ehhez a görbénk $[\vec{c}_{-1}, \vec{c}_0, \vec{c}_1 \ldots \vec{c}_{m+1}]$ vezérlőpontjait úgy kell kitalálni, hogy a következő interpolációs feltétel teljesüljön:

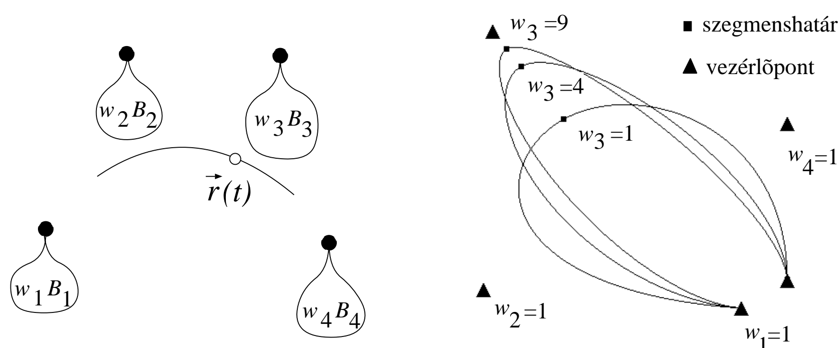
$$\vec{r}(t_j) = \sum_{i=-1}^{m+1} \vec{c}_i \cdot B_{i,k}^{\mathrm{BS}}(t_j) = \vec{p}_j.$$

Ez egy $m+2$ ismeretlenes lineáris egyenletrendszer m egyenletét határozza meg, tehát több megoldás is lehetséges. A feladatot teljesen meghatározottá tehetjük, ha még két járulékos feltételt felveszünk, azaz megadjuk például a görbénk deriváltját a kezdő- és végpontban.

3.3.5. Nem egyenletes racionális B-spline: NURBS

A NUBS bázisfüggvényei, az első és az utolsó néhány bázisfüggvényt kivéve hasonlóak, tehát az egyes vezérlőpontok egyenlő eséllyel küzdenek a görbe alakjának befolyásolásáért. A paraméter függvényében más és más vezérlőpont kerül ki győztesen, amelynek közelében a görbe elhalad. Előfordulhat, hogy bizonyos vezérlőpontok a többieknél fontosabbak, és ezért azt szeretnénk, hogy a görbe őket a többiek kárára is pontosabban közelítse. A NUBS görbénél erre az ad lehetőséget, hogy a megfelelő paramétertartományok hosszát zérusra választjuk, így egy vezérlőpontot kétszeresen, háromszorosan stb. veszünk figyelembe. Ekkor azonban a vezérlőpont fontossága túl nagy ugrásokban változik.

A *nem egyenletes racionális B-spline* (*Non-Uniform Rational B-Spline* vagy röviden *NURBS*) ezen egy új w_i vezérlőpont paraméter, a fontosságot kifejező *súly* (*weight*) bevezetésével segít.



3.20. ábra. *A NURBS görbe és a súly változtatásának a hatása*

A szokásos mechanikai analógiánkban a NURBS-nél egy vezérlőpontba $w_iB_i(t)$ súlyt teszünk, tehát a NUBS-hoz képest az egyes vezérlőpontok hatását még az új súlyértékükkel skálázzuk. A rendszer súlypontja továbbra is a görbe adott t paraméterű

pontja:

$$\vec{r}(t) = \frac{\sum_{i=0}^{m-1} w_i B_i^{\text{NUBS}}(t) \cdot \vec{r}_i}{\sum_{j=0}^{m-1} w_j B_j^{\text{NUBS}}(t)} = \sum_{i=0}^{m-1} B_i^{\text{NURBS}}(t) \cdot \vec{r}_i.$$

A fenti képlet alapján a NUBS és NURBS bázisfüggvények közötti kapcsolat a következő:

$$B_i^{\text{NURBS}}(t) = \frac{w_i B_i^{\text{NUBS}}(t)}{\sum_{j=0}^{m-1} w_j B_j^{\text{NUBS}}(t)}.$$

Mivel a NUBS bázisfüggvények polinomok, a NURBS bázisfüggvények két polinom hányadosaként írhatók fel. Polinomok hányadosát *racionális törtfüggvény*nek nevezzük, ezért jelenik meg a NURBS nevében az R (*Rational*) betű.

A NURBS a többi görbetípushoz képest a járulékos súlyoknak köszönhetően szabadabban vezérelhető. Ráadásul a másodfokú implicit egyenlettel megadható görbéket, az úgynevezett *kúpszelet*eket (kör, ellipszis, parabola, hiperbola stb.) a legalább harmadfokú NURBS-ök segítségével tökéletesen pontosan leírhatjuk, a többi görbével viszont csak közelíthetjük [106]. A jó tulajdonságok ára az, hogy a bázisfüggvények nem polinomok és kiszámításuk osztást is igényel. Ez az ár is látszólagos csupán, ha homogén koordinátákkal dolgozunk. Emlékezzünk vissza, hogy egy homogén koordinátás alakot úgy kapunk meg, hogy a Descartes-koordinátákat egy 1 értékű negyedik koordinátával kiegészítjük:

$$[\vec{r}(t), 1] = \left[\frac{\sum_{i=0}^{m-1} w_i B_i^{\text{NUBS}}(t) \cdot \vec{r}_i}{\sum_{j=0}^{m-1} w_j B_j^{\text{NUBS}}(t)}, \ 1 \right].$$

A homogén koordináták által meghatározott pont nem változik, ha a négyes minden elemét ugyanazzal az értékkel szorozzuk meg. Legyen ez az érték éppen a tört nevezője, így a NURBS homogén koordinátákban:

$$[X_h(t), Y_h(t), Z_h(t), h(t)] =$$

$$\left[\sum_{i=0}^{m-1} w_i B_i^{\text{NUBS}}(t) \cdot x_i, \sum_{i=0}^{m-1} w_i B_i^{\text{NUBS}}(t) \cdot y_i, \sum_{i=0}^{m-1} w_i B_i^{\text{NUBS}}(t) \cdot z_i, \sum_{j=0}^{m-1} w_j B_j^{\text{NUBS}}(t) \right].$$

A dolgunk tehát csak annyival nehezebb, hogy most nem három, hanem négy koordinátával kell számolnunk. Ennyit a szabadabb vezérelhetőség pedig mindenképpen megér.

A NURBS görbeosztályt a NUBS osztályból származtatjuk, azt csak a súlyozással kell kiegészíteni:

```
//===============================================================
class NURBSCurve : public NUBSCurve {
//===============================================================
   float * w;              // súlyok tömbje
public:
   Vector Curve(float t) { // a görbe egy adott pontja
      float total = 0;     // a nevező
      for(int i = 0; i < m; i++) total += B(i, K, t) * w[i];
      Vector rt(0,0,0);
      for(i = 0; i < m; i++) rt += r[i] * (B(i, K, t) * w[i] / total);
      return rt;
   }
};
```

3.3.6. A görbék tulajdonságai

Az előző fejezetekben különböző bázisfüggvényekkel jellemzett görbéket ismertünk meg. A görbéket osztályozhatjuk aszerint, hogy interpolációs vagy approximációs típusúak. A töröttvonal esetében a $t = t_i$ paraméterértéknél az $\vec{r}_i$ súlya 1, az összes többi vezérlőpont súlya zérus, így a töröttvonal interpolációs. Ezzel szemben a Bézier-görbe és a legalább másodfokú B-spline csupán approximációs.

A töröttvonal bázisfüggvényeinek alakjára tekintve azt is megállapíthatjuk, hogy az $\vec{r}_i$ pont csak a (t_{i-1}, t_{i+1}) intervallumban nem zérus súlyú, az ezen kívüli paraméterértéknél a pontnak semmiféle hatása nincs. Más oldalról, ha egy vezérlőpontot megváltoztatunk, az csak a görbe egy kis részét módosítja, a vezérlőponttól távolabb eső tartományokat változatlanul hagyja. Az ilyen tulajdonságokkal rendelkező görbét *lokálisan vezérelhető*nek nevezzük.

A Bézier-görbe bázisfüggvényei a teljes paraméter-intervallumon zérustól különbözők, tehát a Bézier-görbe csak *globálisan vezérelhető*. A B-spline-nál az a tartomány, amelyben egyetlen bázisfüggvény nem zérus, a fokszámmal nő, csak akkor fogja át a teljes paramétertartományt, ha a fokszám eggyel kevesebb a vezérlőpontok számánál.

Ha a bázisfüggvények mindegyike folytonos, akkor a bázisfüggvények lineáris kombinációjával előállított görbe is folytonos. A *folytonosság* (*continuity*) szemléletesen azt jelenti, hogy a görbét le tudjuk rajzolni anélkül, hogy a ceruzánkat a papírról fel kellene emelni. A folytonos görbéket C^0 típusúnak mondjuk.

Mint a töröttvonalnál láttuk, a folytonosság még nem elegendő, ettől a görbe még meglehetősen szögletes lehet. Simább görbéknél nem csupán a görbe, de annak magasabb rendű deriváltjai is folytonosak. Általánosan, ha a görbén belül a deriváltak az n. szintig bezárólag folytonosak, akkor a görbét a C^n *osztály*hoz soroljuk. Ha a vezérlőpontok különbözőek, akkor a nagyobb *folytonossági szint* simább görbét eredményez.

A NUBS és a NURBS fokszámának emelésével együtt nő a folytonossági szint is (3.16. ábra).

Ezek után az alapvető kérdés az, hogy milyen szintű folytonosságot értelmes megkövetelnünk. Vegyünk két példát! Ha egy meghajlított rúd alakja az $y(x)$ függvény, akkor a mechanika törvényei szerint a rúd belsejében ébredő feszültség arányos az $y(x)$ második deriváltjával. Ha azt szeretnénk, hogy a rúd ne törjön el, a feszültség nem lehet végtelen, aminek elégséges feltétele, ha a rúd alakja C^2 folytonos. A második példánkban gondoljunk az animációra, amikor a t paraméter az időt képviseli, a görbe pedig a pozíció vagy orientáció valamely koordinátáját. A mozgás akkor lesz valószerű, ha kielégíti a fizikai törvényeket, többek között Newton második törvényét, miszerint a pozícióvektor második deriváltja arányos az erővel. Mivel az erő valamilyen rugalmas mechanizmuson keresztül hat, nem változhat ugrásszerűen, így a görbe szükségképpen C^2 folytonos. A két példa alapján kijelenthetjük, hogy ha a természet törvényeit szeretnénk követni, akkor C^2 folytonos görbéket kell használnunk. A szakirodalom a *spline* elnevezést gyakran csak a C^2 folytonos görbékre alkalmazza. A Bézier-görbe és a legalább harmadfokú polinomokat használó B-spline kétszeresen folytonosan deriválható.

A megismert görbék bázisfüggvényeinek összege 1, és a bázisfüggvények nem negatívak, ezért használhattuk közvetlenül a súlypont analógiát. A súlypont a vezérlőpontok konvex burkán belül van, így valóban azt várhatjuk a görbénktől, hogy arra megy, amerre a vezérlőpontok vannak.

3.4. Felületek

Eddig görbékkel foglalkoztunk, amelyek egydimenziós alakzatok, azaz az egyenletük az egydimenziós számegyenes egy intervallumát képezte le a háromdimenziós tér pontjaira. A *felület*ek kétdimenziósak, tehát a sík egy tartományát, célszerűen egy téglalapját vagy egy egységoldalú négyzetét feleltetik meg a 3D tér pontjainak. A *felület*ek a görbékhez hasonlóan definiálhatók *paraméteres egyenlet*ekkel, amelyek ezek szerint kétváltozósak:

$$x = x(u,v), \qquad y = y(u,v), \qquad z = z(u,v), \qquad u,v \in [0,1],$$

vagy vektoros alakban: $\vec{r} = \vec{r}(u,v)$. A felületeket *implicit egyenlet*tel is megadhatjuk (a paraméteres egyenlettel szemben az implicit egyenletben nincsenek szabad változók, csak az x,y,z koordináták):

$$f(x,y,z) = 0, \tag{3.14}$$

amit ugyancsak felírhatunk vektoros formában is: $f(\vec{r}) = 0$.

Például egy (x_0, y_0, z_0) középpontú, R sugarú *gömbfelület* paraméteres egyenletei:

$$x = x_0 + R \cdot \cos 2\pi u \cdot \sin \pi v, \qquad y = y_0 + R \cdot \sin 2\pi u \cdot \sin \pi v, \qquad z = z_0 + R \cdot \cos \pi v,$$

$$u, v \in [0, 1],$$

illetve implicit egyenlete:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - R^2 = 0.$$

Az implicit forma előnye, hogy könnyen el tudjuk dönteni, hogy egy pont rajta van-e a felületen vagy sem. Ehhez csupán be kell helyettesíteni a pont koordinátáit az implicit egyenletbe és ellenőrizni, hogy zérust kapunk-e eredményül. A paraméteres forma viszont remekül használható olyan esetekben, amikor megfelelő sűrűséggel pontokat kell előállítanunk a felületen. Az u, v paramétertartományban paraméter párokat veszünk fel, és ezeket az explicit egyenletekbe helyettesítve a felületi pontokhoz jutunk.

3.4.1. Poligonok

A legegyszerűbb felület a sík, illetve annak korlátozásával kapott *háromszög*, *négyszög* vagy általános *sokszög*, más néven *poligon*.

A háromszög

Tekintsük a *háromszög*et, amelyet az $\vec{r}_1, \vec{r}_2, \vec{r}_3$ csúcspontjaival definiálhatunk! A háromszög belső pontjaihoz a baricentrikus koordináták elve szerint juthatunk el. Tegyünk az első csúcspontba u, a másodikba v, a harmadikba pedig $1 - u - v$ súlyt, és nézzük a rendszer súlypontját! A súlypont akkor lesz a három pont konvex burkán, azaz a háromszögön belül, ha a súlyok nem negatívak, tehát ezt a feltételt is beépítjük a *háromszög paraméteres egyenlet*ébe:

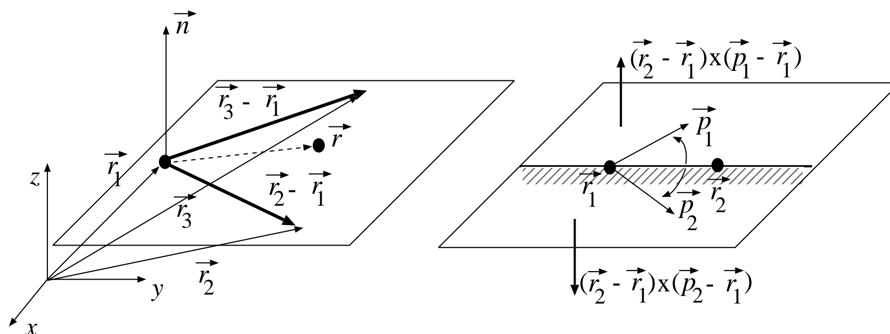
$$\vec{r}(u, v) = \vec{r}_1 \cdot u + \vec{r}_2 \cdot v + \vec{r}_3 \cdot (1 - u - v), \quad u, v \geq 0, \quad u + v \leq 1. \tag{3.15}$$

A *háromszög implicit egyenlet*éhez két lépésben juthatunk el (3.21. ábra). Először a háromszög tartósíkjának egyenletét írjuk fel, majd feltételeket adunk arra, hogy egy síkbeli pont a háromszög belsejében van-e. A háromszög síkjának normálvektora merőleges az élekre, így a két élvektor vektoriális szorzataként számítható:

$$\vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1).$$

A sík egy helyvektora $\vec{r}_1$, ezért a sík $\vec{r}$ pontjaira az $\vec{r} - \vec{r}_1$ vektorok a síkkal párhuzamosak, tehát a normálvektorra merőlegesek, azaz kielégítik a következő *sík egyenlet*et:

$$\vec{n} \cdot (\vec{r} - \vec{r}_1) = 0. \tag{3.16}$$



3.21. ábra. *A háromszög belső pontjai*

A sík pontjai közül nem mindegyik van a háromszög belsejében. Egy $\vec{r}$ pont akkor van a háromszögön belül, ha a háromszög mind a három oldalegyeneséhez viszonyítva a háromszöget tartalmazó félsíkban van. Tekintsük az $\vec{r}_1$ és $\vec{r}_2$ csúcsokon átmenő egyenest és egy tetszőleges $\vec{p}$ pontot! A *vektoriális szorzat*tal kapott vektor hosszának kifejezésében a két vektor abszolút értékei és a közöttük lévő szög szinusza szerepel. Mivel a szinusz 0–180 fok között pozitív, 180–360 fok között pedig negatív, a

$$(\vec{r}_2 - \vec{r}_1) \times (\vec{p} - \vec{r}_1)$$

vektoriális szorzat az egyenes egyik oldalán lévő $\vec{p}$ pontra a normálvektor irányába, a másik oldalán lévő $\vec{p}$ pontra viszont éppen ellentétesen fog mutatni (3.21. ábra). Ha tehát ezt a vektort a normálvektorral skalárisan szorozzuk, akkor az egyik oldalon pozitív, a másik oldalon pedig negatív eredményt kapunk. A vizsgálatot mindhárom oldalra elvégezve a következő feltételrendszerhez jutunk:

$$\begin{aligned} ((\vec{r}_2 - \vec{r}_1) \times (\vec{r} - \vec{r}_1)) \cdot \vec{n} &\geq 0, \\ ((\vec{r}_3 - \vec{r}_2) \times (\vec{r} - \vec{r}_2)) \cdot \vec{n} &\geq 0, \\ ((\vec{r}_1 - \vec{r}_3) \times (\vec{r} - \vec{r}_3)) \cdot \vec{n} &\geq 0. \end{aligned} \tag{3.17}$$

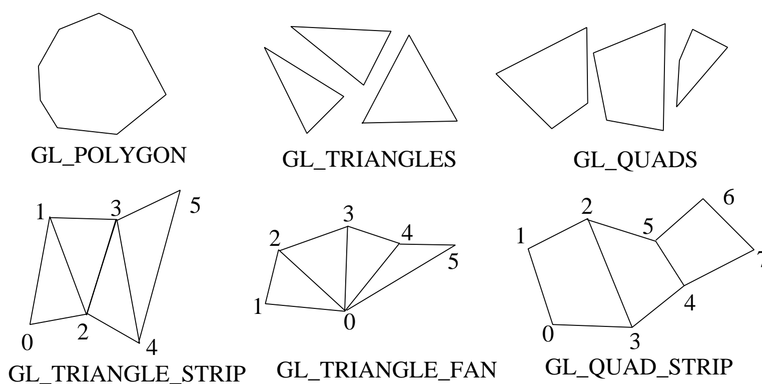
A háromszög $\vec{r}$ pontjai kielégítik a 3.16. egyenletet és a 3.17. egyenlőtlenségeket.

A négyszög

A négyszöget célszerű mindig két olyan háromszögnek tekinteni, amelyek két-két csúcsát összeragasztottuk. A számítógépes grafikában nem kell ragaszkodnunk ahhoz, hogy a négy csúcs egy síkban legyen, ezért négyszögnek nevezhetünk minden pontnégyest.

Hálók

Bonyolultabb felületekhez több három- vagy négyszöget kell alkalmaznunk. A több egymáshoz illeszkedő, nem feltétlenül egy síkban lévő sokszöget tartalmazó felületet *háló*nak (*mesh*) nevezzük. A hálókban a csúcspontok koordinátáin kívül a lapok, az élek és a csúcsok illeszkedési viszonyait (*topológia*) is nyilván kell tartani, hiszen enélkül nem tudnánk, hogy egy csúcspont megváltoztatása vagy él törlése mely lapokat érinti. Ismernünk kell például, hogy egy csúcsban mely élek és mely lapok találkoznak, egy élnek melyek a végpontjai és melyik két lapot választja el, valamint azt is, hogy egy lapnak melyek az élei és csúcsai. A kapcsolódási információk ismerete több szempontból is hasznos. Ha egy háló szerkesztésénél egy csúcspontot módosítunk, akkor az összes, a csúcspontra illeszkedő háromszög alakja megváltozik, tehát az alakot anélkül változtathatjuk, hogy a topológiát elrontanánk. Másrészt, mivel a hálókban egy csúcspont sok háromszögben vesz részt, lényegesen kevesebb csúcspontra van szükségünk, mintha a háromszögeket egyenként sorolnánk fel, így az adatszerkezet kisebb helyen elfér és a transzformációkat is gyorsabban elvégezhetjük.



3.22. ábra. *OpenGL hálók*

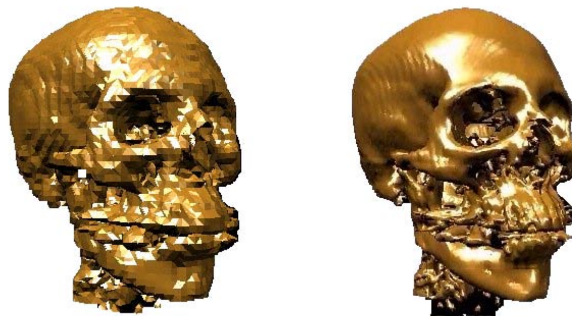
Az illeszkedési viszonyokat leíró adatszerkezetekkel a 5.2.2. fejezetben foglalkozunk. Most néhány olyan fontos speciális esetet tárgyalunk, amikor a csúcspontok felsorolási sorrendjéből kideríthető, hogy hol vannak élek és lapok (3.22. ábra). Az ilyen hálókat tehát nagyon egyszerűen, a csúcspontok tömbjével reprezentálhatjuk. Ezen hálók jelentőségét tovább növeli, hogy az *OpenGL* csak ilyen formában átadott hálókat hajlandó megjeleníteni. Az alábbi felsorolásban a hálók OpenGL nevét is megadjuk:

- *Egyetlen különálló poligon* (`GL_POLYGON`).
- *Háromszög lista* (`GL_TRIANGLES`): Háromszögek felsorolása, amelyben minden egymást követő ponthármas egy háromszöget azonosít.

- *Négyszög lista* (`GL_QUADS`): Négyszögek felsorolása, amelyben minden egymást követő pontnégyes egy különálló négyszöget ír le.
- *Háromszög szalag* (`GL_TRIANGLE_STRIP`): Egymáshoz mindig egy-egy élben kapcsolódó háromszögek. Az i-edik háromszög csúcsai az i-edik, az $(i+1)$-edik és az $(i+2)$-tedik pont.
- *Háromszög legyező* (`GL_TRIANGLE_FAN`): Egy csúcsot közösen birtokló és páronként közös élre illeszkedő háromszögek. Az i-edik háromszög csúcsai az első, az $(i+1)$-edik és az $(i+2)$-tedik pont. Az első csúcspont minden háromszögben szerepel.
- *Négyszög szalag* (`GL_QUAD_STRIP`): Egymáshoz mindig egy-egy élben kapcsolódó négyszögek. Az i-edik négyszög csúcsai a $2i$-edik, a $(2i+1)$-edik, a $(2i+2)$-tedik és az $(2i+3)$-adik pont.

Árnyalási normálisok

A poligonok síklapokra illeszkednek, ezért minden pontjukban ugyanaz a normálvektoruk. Ez rendben is lenne akkor, ha a tervezett felület valóban ilyen szögletes. A poligonhálókat azonban gyakran valamilyen mérési vagy közelítési feladat eredményeként kapjuk, amikor szó sincs arról, hogy a célfelület szögletes, csupán nincs jobb közelítő eszköz a kezünkben, mint például egy háromszög háló.



3.23. ábra. *Saját normálvektorok (bal) és az árnyalási normálisok (jobb)*

Mivel ekkor a normálvektor ugrásszerűen változik a háromszögek határán, az így megjelenített képekről ordít, hogy a görbült felületet háromszögekkel közelítettük (3.23. ábra bal oldala). Ezen úgy segíthetünk, ha a visszavert fény intenzitásának számításakor nem a háromszögek normálvektoraival, hanem a háromszög belsejében folyamatosan

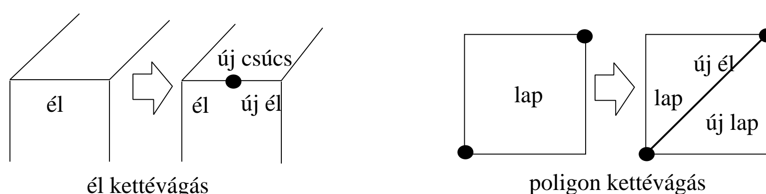
változó „normálvektorral” dolgozunk. A folyamatosan változó normálvektor az eredeti, görbült felület normálvektorának közelítése. A modellben tehát a normálvektorokat a háló csúcsaihoz rendeljük, a háromszög belső pontjaiban pedig a három csúcspontban található normálvektorokból lineáris interpolációval számoljuk ki az úgynevezett *árnyalási normálvektor* értékét. Mivel ekkor két érintkező háromszög határán mindkét háromszögben ugyanaz a normálvektor, a felület, legalábbis látszólag, sokkal simább lesz (3.23. ábra jobb oldala).

3.4.2. Poligon modellezés

A poligon modellezés elemi lépései poligonhálókat módosítanak. A poligonháló által meghatározott testet *poliéder*nek nevezzük. Ha a poligonokat egymástól függetlenül adjuk meg, akkor nem lehetünk biztosak abban, hogy azok hézagmentesen illeszkednek egymáshoz és egy érvényes 3D testet fognak közre. Ezért olyan műveletekkel kell építkeznünk, amelyek a test *topológiai helyesség*ét nem rontják el. Az egyszerűség kedvéért csak az egyetlen darabból álló, lyukakat nem tartalmazó *poliéder*ek létrehozásával foglalkozunk. Egy ilyen poliéder érvényességének szükséges feltétele, hogy, ha l lapot, c csúcsot és e élt tartalmaz, akkor fennáll az *Euler-tétel*:

$$l + c = e + 2. \tag{3.18}$$

Például egy téglatestnek 6 lapja, 8 csúcsa és 12 éle van, így kielégíti az Euler-egyenletet. Azokat az elemi műveleteket, amelyek a lapok, csúcsok és élek számát úgy változtatják meg, hogy közben az Euler-egyenlet egyensúlyát nem borítják fel, *Euler-művelet*eknek nevezzük. Most csak a leghasznosabb Euler-műveletekkel, az *él kettévágás*sal, a *poligon kettévágás*sal, az *élzsugorítás*sal és a *poligon kihúzás*sal foglalkozunk.

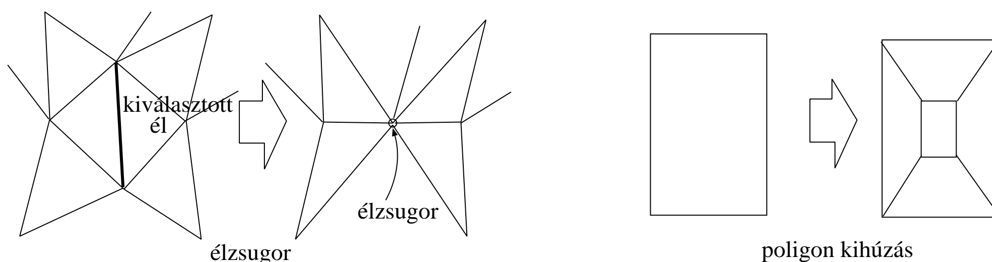


3.24. ábra. *Él és poligon kettévágás*

Az *él kettévágás*hoz (*edge split*) egy pontot jelölünk ki az élen, és itt az élt kettévágjuk. A művelet az Euler-egyenlet mindkét oldalát eggyel növeli (3.24. ábra bal oldala).

A *poligon kettévágás*hoz (*polygon split*) a poligon két csúcsát jelöljük ki, amelyek között egy új élt veszünk fel, amely az eredeti poligont kettébontja (3.24. ábra jobb oldala). Ezzel egy új él és egy új lap keletkezik, az Euler-egyenlet bal és jobb oldala

tehát egyaránt eggyel növekszik. Megjegyezzük, hogy egyes modellező eszközök az él és poligon kettévágást egy műveletté vonják össze.



3.25. ábra. *Élzsugorítás és poligon kihúzás*

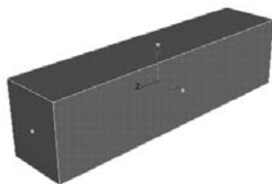
Az *élzsugorítás* (*edge collapse*) vagy más néven *csúcspont összevonás* (*vertex merge*) egy él két végpontját egyesíti, mialatt az él eltűnik (3.25. ábra). Négyszöghálónál a hatás csupán ennyi, háromszög hálónál viszont az a két lap is megszűnik, amelyek a zsugorított élre illeszkedtek. Az élzsugorítás is Euler-művelet, hiszen az élek számát eggyel, a lapok számát kettővel, a csúcsok számát pedig eggyel csökkenti, így az Euler-egyenlet két oldalát az egyensúly betartásával változtatja meg.

A *poligon kihúzás*hoz (*polygon extrude*) a poliéder egy lapját kijelöljük, majd azt a lapot elmozdítva, skálázva, esetleg elforgatva egy új poligont hozunk létre. Az eredeti poligon eltűnik, viszont az eredeti poligon és az új poligon élei között összekötő négyszögek jelennek meg (3.25. és 3.26. ábra). Ha a kiválasztott poligonnak e_p éle van, akkor a művelet során $2e_p$ új él, $e_p + 1$ új lap és e_p új csúcs keletkezik, mialatt egyetlen lap szűnik meg. Az új poliéder $e' = e + 2e_p$ élt, $l' = l + e_p + 1 - 1$ lapot, és $c' = c + e_p$ csúcsot tartalmaz, tehát továbbra is fennáll az $l' + c' = e' + 2$ Euler-összefüggés, ha a műveletet megelőzően fennállt.

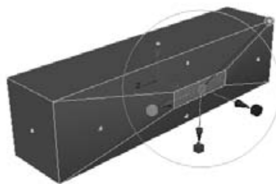
3.4.3. Felosztott felületek

A poligon modellek meglehetősen szögletesek. A paraméteres (például NURBS) felületek viszont szép simák, még a többszörös deriváltjaik is folytonosak. Ha a felületre mechanikai számítások miatt van szükségünk, akkor a magasabb rendű folytonosságnak nagy jelentősége van, így ebben az esetben a poligon modellezés önmagában nem használható. Ha viszont a felületmodellt megjelenítésre használjuk, akkor a mégoly sima NURBS felületeket is poligonhálókkal közelítjük, hiszen a képszintézis algoritmusok zöme csak ilyen modelleket képes megjeleníteni.

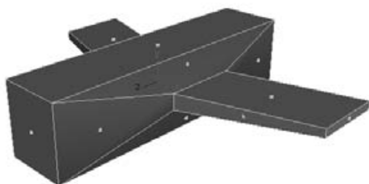
Itt álljunk meg egy pillanatra! A poligon modellt túl szögletesnek tartottuk, ezért paraméteres felületmodellekre tértünk át, amelyeket a megjelenítés előtt megint sokszögekre bontunk. Rögtön adódik a kérdés, hogy nem lehetne-e kikerülni a paramé-



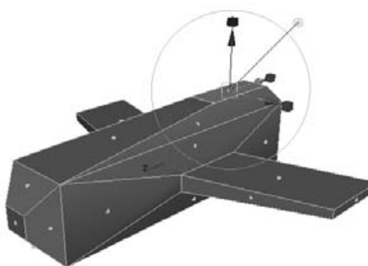
1. A kiindulási alakzat egy téglatest



2. Az oldallapok kihúzása



3. A kihúzott oldallapok újbóli kihúzása



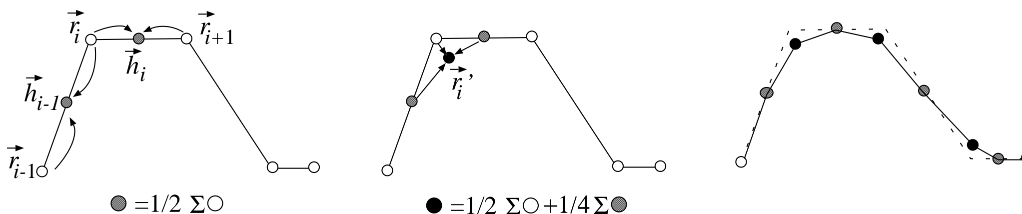
4. Az első és felső lapok kihúzása

5. A kihúzott felső lap újbóli kihúzása

6. Simítás felosztással

3.26. ábra. *Egy űrhajó létrehozásának lépései poligon modellezéssel*

teres felületeket, és helyettük a szögletes poligon modelleket úgy simítgatni, illetve felosztani, hogy azok kevésbé szögletesnek látszó hálókat eredményezzenek? A válasz szerencsére igen, az eljárást pedig *felosztott felület* (*subdivision surface*) módszernek nevezzük.



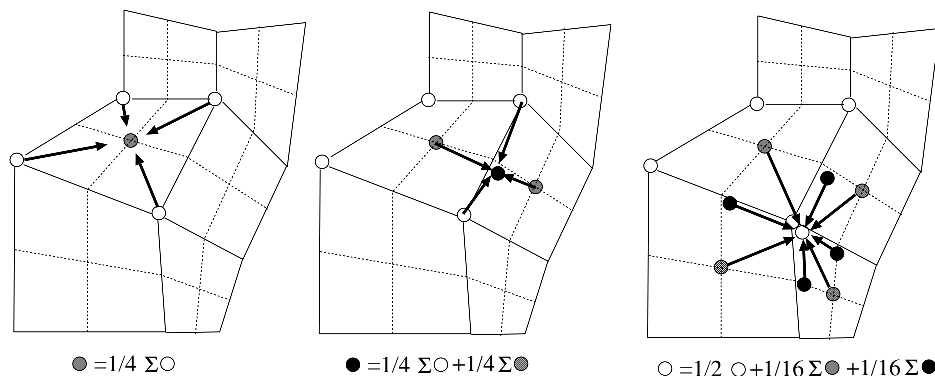
3.27. ábra. *Felosztott görbe létrehozása*

A felosztott felületek elvének megértéséhez először vegyük elő régi ismerősünket a töröttvonalat, amely igazán szögletes, hiszen a megadott $\vec{r}_0, \ldots, \vec{r}_{m-1}$ pontsorozatot szakaszokkal köti össze! Egy látszólag simább töröttvonalhoz jutunk a következő, a vezérlőpontokat megduplázó eljárással (3.27. ábra). Minden szakaszt megfelezünk és ott egy-egy új $\vec{h}_0, \ldots, \vec{h}_{m-2}$ vezérlőpontot veszünk fel. Bár már kétszer annyi vezérlőpontunk van, a görbénk éppen annyira szögletes, mint eredetileg volt. A régi vezérlőpontokat ezért úgy módosítjuk, hogy azok a régi helyük és a két oldalukon lévő felezőpontok közé kerüljenek, az alábbi súlyozással:

$$\vec{r}_i\,' = \frac{1}{2}\vec{r}_i + \frac{1}{4}\vec{h}_{i-1} + \frac{1}{4}\vec{h}_i = \frac{3}{4}\vec{r}_i + \frac{1}{8}\vec{r}_{i-1} + \frac{1}{8}\vec{r}_{i+1}.$$

Az új töröttvonal valóban sokkal simábbnak látszik. Ha még ez sem elégít ki bennünket, az eljárást tetszőleges mélységig ismételhetjük. Ha végtelen sokszor tennénk meg, akkor éppen a B-spline-t állítanánk elő.

Az eljárás közvetlenül kiterjeszthető háromdimenziós hálókra, amelynek eredménye a *Catmull – Clark felosztott felület* (*Catmull – Clark subdivision surface*) [28]. Induljunk ki egy háromdimenziós négyszöghálóból (3.28. ábra) (az algoritmus nemcsak négyszögeket képes felosztani, de a létrehozott lapok mindig négyszögek). Első lépésként minden él közepén felveszünk egy-egy *élpont*ot, mint az él két végpontjának az átlagát, és minden lap közepén egy-egy *lappont*ot, mint a négyszög négy csúcspontjának az átlagát. Az új élpontokat a lappontokkal összekötve ugyanazt a felületet négyszer annyi négyszöggel írtuk le. A második lépésben kezdődik a simítás, amikor az élpontokat módosítjuk az élhez illeszkedő lapok lappontjai alapján úgy, hogy az új élpont éppen a két lappont és az él két végén levő csúcspont átlaga legyen. Ugyanezt az eredményt úgy is megkaphatjuk, hogy az élpontot a két, az élre illeszkedő lap négy-négy eredeti sarokpontjának, valamint az él két végpontján található pontnak az átlagát képezzük (azaz



3.28. ábra. *Catmull – Clark felosztás egy lépése*

az él végpontjait háromszor szerepeltetjük az átlagban). A simítás utolsó lépésében az eredeti csúcspontok új helyét súlyozott átlaggal határozzuk meg, amelyben az eredeti csúcspont $1/2$ súlyt, az illeszkedő élek összesen 4 db módosított élpontja és illeszkedő lapok összesen 4 db lappontja pedig $1/16$ súlyt kap. Az eljárást addig ismételjük, amíg a felület simasága minden igényünket ki nem elégíti (3.29. ábra).

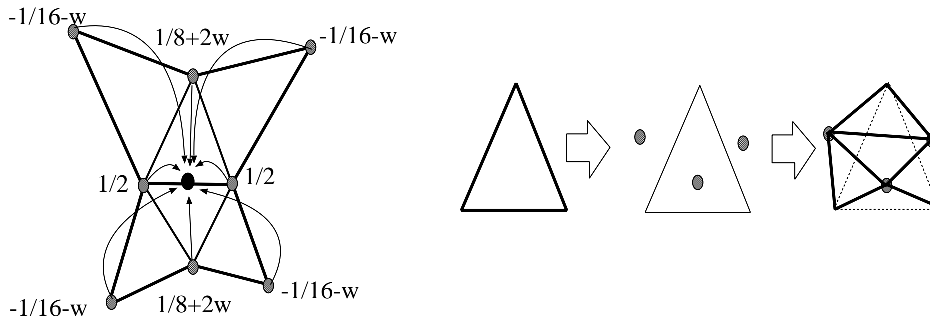
3.29. ábra. *Az eredeti háló valamint egyszer és kétszer felosztott változatai*

Ha a háló egyes éleinek és csúcsainak környezetét nem szeretnénk simítani, akkor a megőrzendő éleken túl lévő pontokat nem vonjuk be az átlagolási műveletekbe.

A felosztott felületeknek a simításon kívül még van egy fontos alkalmazási területe. A 3.28. ábrára nézve megállapíthatjuk, hogy a felosztás egyrészt új csúcsokat hoz létre, másrészt pedig a már meglévő csúcsokat a környéken lévő új csúcsok és a régi csúcs átlagára állítja be. Ezt úgy is tekinthetjük, mintha egyszerre két hálónk lenne, az eredeti felosztatlan, és az új dupla felbontású, a végső felületet pedig két háló átlaga jelenti. A két hálót egyaránt változtathatjuk, az eredeti háló a nagyvonalú alakításokat, a második pedig a finomhangolást jelenti. Ha nem állunk meg egyetlen felosztás után, akkor akár sok különböző részletezettségű hálót kapunk. A hálók hierarchiájában mindig az

elvégzendő változtatás kiterjedése szerint választunk.

A Catmull-Clark felosztás approximációs, azaz az eredmény csak közelíti az eredeti háló csúpontjait. Ezt a hátrányt küszöböli ki a háromszöghálókon működő *pillangó felosztás* (*butterfly subdivision*) [37].



3.30. ábra. *Az új élpont meghatározása és a háromszög pillangó felosztása*

A pillangó felosztás a háromszögek élfelező pontjainak közelébe egy-egy új élpontot helyez, majd az eredeti háromszögeket négy új háromszöggel váltja fel. Az új háromszögek csúcsai egyrészt az eredeti háromszög csúcsai, másrészt az élfelező pontjai (3.30. ábra). Az élpontok kialakításában az élre illeszkedő háromszögek csúcspontjai és ezen két háromszöggel közös élt birtokló még további négy háromszög vesz részt. Az élpontra ható háromszögek elrendezése egy pillangóra emlékeztet, ami magyarázza az eljárás elnevezését. Az élpont koordinátáit az él végpontjainak koordinátáiból számítjuk $1/2$-es súlyozással, az élre illeszkedő két háromszög harmadik csúcsaiból $1/8+2w$ súlyozással, valamint az élre illeszkedő két háromszöghöz tapadó négy háromszögnek az illeszkedő háromszögön kívül lévő csúcsaiból $-1/16-w$ súlyozással. A w a művelet paramétere, amellyel azt állíthatjuk be, hogy az eljárás mennyire görbítse meg a felületet az élek környezetében. A $w=-1/16$-os beállítás megtartja a háló szögletességét, a $w=0$-t használó felosztás pedig erősen legömbölyíti az eredeti éleket.

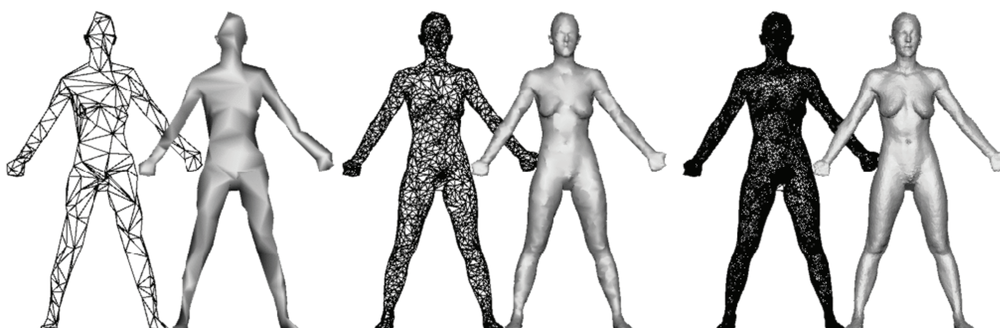
3.4.4. Progresszív hálók

A felosztott felületek egy poligonhálót finomítanak, ezzel annak méretét növelik. Szükségünk lehet egy ellentétes folyamatra is, amikor a poligonháló túlságosan nagy, ezért kevesebb poligont tartalmazó hálóval szeretnénk közelíteni, esetleg azon az áron is, hogy az eredmény szögletesebb lesz.

A Hoppe-féle *progresszív háló* [57] *élzsugorítás*ok (*edge collapse*) sorozatával dolgozik (3.25. ábra). Az élzsugorítás kiválaszt egy élt, és azt eltávolítja a modellből, minek következtében az élre illeszkedő két háromszög is eltűnik, az él két végpontjából pedig egyetlen pont lesz. A két csúcspontot felváltó új csúcspont például a két csúcspont

koordinátáinak az átlagaként számítható. Nyilván azt az élt érdemes az egyes fázisokban zsugorítani, amelyik a legkisebb mértékben módosítja a poliéder alakját, hiszen azt szeretnénk, hogy az egyszerűsített modell kevesebb háromszöggel, de lehetőség szerint pontosan írja le az eredeti alakzatot. Minden élhez egy-egy prioritásértéket rendelünk, amely kifejezi, hogy ha ezt az élt zsugorítjuk, akkor milyen mértékben változik meg a modellünk. Az egyszerűsítés egyes lépéseiben mindig a legkisebb prioritású éltől, és az erre illeszkedő lapoktól szabadulunk meg.

A prioritásfüggvény definiálására nincsenek bombabiztos módszerek, leginkább heurisztikus eljárások jöhetnek szóba. Egy szokásos heurisztika azokat az éleket tartja meg, amelyek hosszúak, és az itt találkozó lapokról kevéssé mondható el, hogy egy síkban lennének, azaz a normálvektoraik által bezárt szög nagy. A prioritásfüggvény ebben az esetben az él hosszának és a normálvektorok skalárszorzatának a hányadosa. Ez a kritérium azonban nem garantálja, hogy az egyszerűsítések során a test topológiája megmarad, előfordulhat, hogy az több különálló részre esik szét.



3.31. ábra. *Egy geometriai modell három változatban (795, 6375 és 25506 lap)*

Az egyszerűsítésnek több előnye is van. A nagy poligonszám több képszintézis időt emészt fel, így valós idejű képszintézis rendszerekben (például játékokban) szükségünk van egyszerűbb (*low-poly*) modellekre. Az eljárásnak különösen nagy jelentősége van akkor, ha az eredeti hálót nem kézi modellezéssel, hanem mérési vagy konverziós eljárásból kaptuk. Ilyen esetekben ugyanis könnyen előfordulhat, hogy a háló kezelhetetlenül sok (akár több millió) háromszöget tartalmaz. A 3.31. ábra jobb oldalán például egy hölgy[2] 3D scanner segítségével mért felülete látható. A középső modellben a lapok számát 25%-ra, a bal oldaliban pedig 3%-ra csökkentettük.

Másrészt nagy segítséget adnak az egyszerűsített modellek a több *részletezettségi szint*et alkalmazó geometriai modelleknél (*level of detail* vagy *LOD*). Gondoljunk arra, hogy egy tárgyat a virtuális térbeli barangolásunk során néha egészen közelről, máskor pedig meglehetősen távolról szemlélünk. Ha a tárgyat közelről látjuk, részletes modellt

[2] http://www.3DCafe.com

kell megjelenítenünk, különben a szögletesség csúnya hatást kelt. Ha viszont a tárgy távolban van, és ezért csupán néhány pixelt foglal el a képen, akkor feleslegesnek tűnik a tárgyat sok ezer, pixelnél kisebb méretű poligonnal modellezni. Ilyen környezetekben érdemes ugyanannak a geometriának különböző részletezettségű modelljeivel dolgozni, és a szemlélő távolsága alapján mindig a legmegfelelőbbet kiválasztani.

Utoljára hagytuk azt a felhasználást, ami megmagyarázza a progresszív háló elnevezést. Az egyszerűsítési sorozat megfordítható, ha minden zsugorított élhez eltároljuk inverzének — azaz egy *csúcspont kettévágás* (*vertex split*) műveletnek — paramétereit. A paraméterek megadják, hogy az élzsugor alatt milyen változásokat szenvedtek el az illeszkedő lapok, élek és csúcsok. Egy erősen egyszerűsített modellváltozat, és a csúcspont kettévágás műveletek paraméterei alapján bármely kevésbé egyszerűsített változathoz lépésről lépésre visszatérhetünk. Képzeljük el, hogy a bonyolult modellünket a hálózaton keresztül szeretné valaki megvizsgálni! A leegyszerűsített változat még a lassabb kapcsolaton is gyorsan odaér, tehát a türelmetlen felhasználó rögtön kap egy közelítő modellt, amely az időben fokozatosan finomodik, ahogy a folyamatosan érkező paraméter rekordok progresszív módon tökéletesítik azt.

3.4.5. Implicit felületek

Az implicit felületekhez az $f(x,y,z) = 0$ implicit egyenlettel leírható alakzatok tartoznak.

Kvadratikus felületek

Egy fontos felületosztályhoz juthatunk, ha az olyan implicit egyenleteket tekintjük, ahol bármely változó legfeljebb másodfokú alakban szerepelhet. Az összes ilyen egyenlet megadható egy általános, *homogén koordinátás alak*ban:

$$[x,y,z,1] \cdot \mathbf{Q} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0, \qquad (3.19)$$

ahol $\mathbf{Q}$ egy 4×4-es konstans együttható mátrix. A *kvadratikus felület*ek speciális típusai az *ellipszoid*, a *hengerpalást*, a *kúp*, a *paraboloid*, a *hiperboloid* stb. A 3.32. ábrán egy

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} - 1 = 0$$

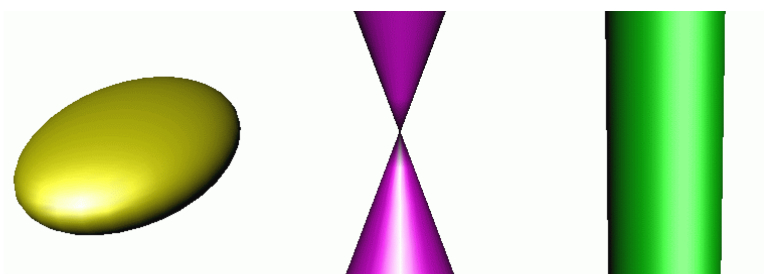
egyenletű ellipszoidot, egy

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - z^2 = 0$$

egyenletű ellipszis alapú végtelen kúpot, és egy

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - 1 = 0$$

egyenletű ellipszis alapú hengerfelületet láthatunk. A végtelenbe nyúló változatok helyett a megszokott változatokat kapjuk, ha a koordinátákat például a $0 \leq z \leq z_{\max}$ egyenlőtlenségekkel korlátozzuk.



3.32. ábra. *Kvadratikus felületek*

Magasságmezők

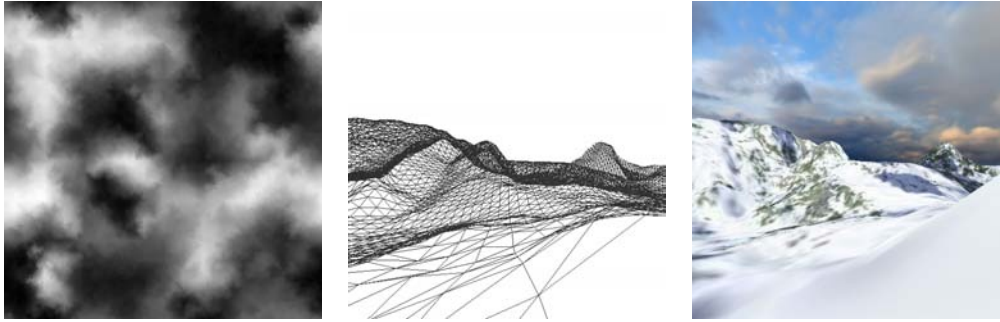
A *magasságmező*k olyan implicit felületek, ahol az $f(x,y,z) = 0$ implicit egyenlet a

$$z = h(x,y)$$

alakra hozható. Erre természetesen csak akkor van lehetőség, ha egy x,y koordináta mellett pontosan egy z érték elégíti ki az implicit egyenletet. A magasságmező elnevezés abból a felismerésből ered, hogy ezeket a felületeket elképzelhetjük úgy is, hogy a tengerszinthez (x,y sík) képest megadjuk a terep z magasságát. Például a következő egyenlet egy az origóból induló, elhaló körhullámot ír le:

$$z = \frac{1}{\sqrt{x^2 + y^2 + 1}} \cdot \sin\left(\sqrt{x^2 + y^2}\right).$$

A magasságmezők egyesítik az paraméteres és implicit egyenletek előnyeit. Ugyanis, hasonlatosan az paraméteres egyenletekhez, a magasságmezőben könnyű pontokat felvenni, csupán x,y koordinátapárokat kell választani, majd őket a h magasságfüggvénybe behelyettesíteni. Másrészt, miként az implicit egyenleteknél, behelyettesítéssel egyszerűen eldönthetjük, hogy egy x,y,z pont rajta van-e felületen. A magasságmezőket gyakran alkalmazzák *terepmodellezés*ére. A magasságértékek származhatnak mérésekből, vagy egy fraktális felosztó algoritmus eredményéből [118].



3.33. ábra. *Magasságmező mint szürkeárnyalatos kép, és a belőle származó felület*

Amennyiben az x, y értelmezési tartománya az $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ téglalap, a magasságmezőket kétdimenziós tömbökben is tárolhatjuk úgy, hogy az x tartományt egyenletesen felosztjuk N, az y tartományt pedig M részre, és az így kapott $N \times M$ méretű rács csúcspontjaiban adjuk meg a magasság értékét. A tömb i, j eleme

$$z_{ij} = h(x_i, y_j) = h\left(x_{\min} + \frac{i}{N} \cdot (x_{\max} - x_{\min}),\ y_{\min} + \frac{j}{M} \cdot (y_{\max} - y_{\min})\right).$$

A rácspontok között lineárisan interpolálunk. Egy kétdimenziós, skalárértékeket tartalmazó tömb egy fekete-fehér képnek is tekinthető, tehát a magasságmező létrehozásához egy ilyen képet kell megalkotni (3.33. ábra).

3.4.6. Parametrikus felületek

A parametrikus felületek kétváltozós függvények:

$$\vec{r}(u, v), \qquad u, v \in [0, 1].$$

A parametrikus görbékhez képest az egyetlen különbség, hogy most nem a számegyenes egy intervallumát, hanem az egységnégyzetet képezzük le az alakzat pontjaira, ezért a parametrikus függvényben két független változó szerepel.

Miként a parametrikus görbéknél láttuk, a függvény közvetlen megadása helyett véges számú $\vec{r}_{ij}$ vezérlőpontot veszünk fel, amelyeket a bázisfüggvényekkel súlyozva kapjuk meg a felületet leíró függvényeket:

$$\vec{r}(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \vec{r}_{ij} \cdot B_{ij}(u, v). \tag{3.20}$$

A bázisfüggvényektől továbbra is elvárjuk, hogy összegük minden paraméterre egységnyi legyen, azaz $\sum_{i=0}^{n}\sum_{j=0}^{m} B_{ij}(u,v) = 1$ mindenütt fennálljon. Ekkor ugyanis a súlypont analógia szerint most is elképzelhetjük úgy, mintha a vezérlőpontokba u,v-től függő $B_{ij}(u,v)$ súlyokat helyezünk, és a rendszer súlypontját tekintjük a felület ezen u,v párhoz tartozó pontjának.

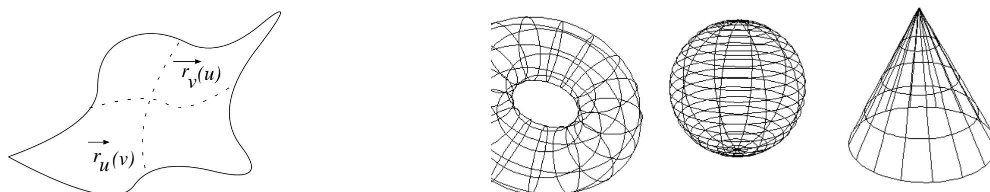
Szorzatfelületek

A $B_{ij}(u,v)$ bázisfüggvények definíciójánál visszanyúlhatunk a görbéknél megismert eljárásokra. Rögzítsük gondolatban a v paraméter értéket. Az u paraméterértéket szabadon változtatva egy $\vec{r}_v(u)$ görbét kapunk, amely a felületen fut végig (3.34. ábra). Ha a NURBS vagy a Bézier-görbe tulajdonságai megfelelnek, akkor keressük a felületet olyan alakban, hogy ez a görbe ugyancsak ilyen típusú legyen, tehát:

$$\vec{r}_v(u) = \sum_{i=0}^{n} B_i(u)\vec{r}_i, \tag{3.21}$$

ahol a $B_i(u)$ a kívánt görbe bázisfüggvénye. Természetesen, ha más v értéket rögzítünk, akkor a felület más görbéjét kell kapnunk. Mivel egy adott típusú görbét a vezérlőpontok egyértelműen definiálnak, az $\vec{r}_i$ vezérlő pontoknak függeniük kell a rögzített v paramétertől. Ahogy a v változik, az $\vec{r}_i = \vec{r}_i(v)$ ugyancsak egy görbén fut végig, amit érdemes ugyanazon görbetípussal a $\vec{r}_{i,0}, \vec{r}_{i,2}, \ldots, \vec{r}_{i,m}$ vezérlőpontok segítségével felvenni:

$$\vec{r}_i(v) = \sum_{j=0}^{m} B_j(v)\vec{r}_{ij}.$$



3.34. ábra. *Egy paraméteres felület paramétervonalai*

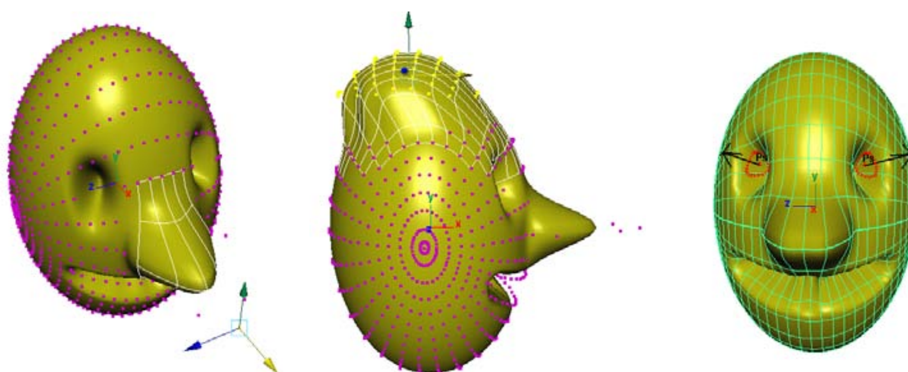
Ezt behelyettesítve a 3.21. egyenletbe, a felület paraméteres függvénye a következő lesz:

$$\vec{r}(u,v) = \vec{r}_v(u) = \sum_{i=0}^{n} B_i(u)\left(\sum_{j=1}^{m} B_j(v)\vec{r}_{ij}\right) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i(u)B_j(v)\cdot\vec{r}_{ij}.$$

A görbékkel összehasonlítva most a vezérlőpontok egy 2D rácsot alkotnak, a kétváltozós bázisfüggvényeket pedig úgy kapjuk, hogy a görbéknél megismert bázisfüggvények u-val és v-vel parametrizált változatait összeszorozzuk.

NURBS felület, felületszobrászat

Ha a $B_i(u)$ és $B_j(v)$ függvényeket a NURBS bázisfüggvényeknek választjuk, akkor *NURBS felület*hez jutunk. A legalább harmadfokú NURBS segítségével a másodfokú implicit egyenlettel leírható felületeket (gömb, ellipszoid, henger stb.) tökéletesen elő tudjuk állítani.



3.35. ábra. *A vezérlőpontok módosítása és felületszobrászat*

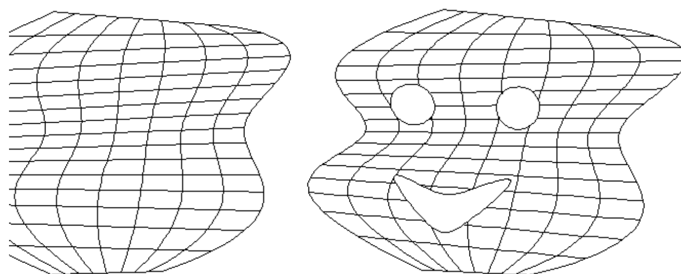
A NURBS felületet a vezérlőpontok mozgatásával, illetve a vezérlőpontokhoz rendelt súlyok állítgatásával alakíthatjuk. Minden vezérlőpont egy kis mágnes, amely maga felé húzza a felület közeli részét. A mágnes hatását a paramétertartomány korlátozza. Például egy harmadfokú NURBS összesen 16 tartományra hat, azon kívül nem (lásd a 3.35. ábra első fejét). Egy-egy mágnes erejét, a többi rovására a súlyok növelésével fokozhatjuk.

A vezérlőpontok egyenkénti vagy csoportos áthelyezésénél szemléletesebb a *felületszobrászat* (*sculpting*), amely a vezérlőpontokat nem közvetlenül, hanem egy természetes formaalakító műveletet beiktatva változtatja meg (3.35. ábra harmadik feje). Ez a formaalakító művelet leginkább az agyagszobrászathoz hasonlít, amikor az anyagot simogatva, nyomogatva mélyedéseket hozhatunk létre. A virtuális szobrászathoz a kurzor által kijelölt felületi pont környezetében lévő vezérlőpontokat az itteni normális irányában elmozdítjuk egy kicsit és ezt periodikusan ismételgetjük, amíg a kurzor éppen itt tartózkodik. Ugyanezzel a módszerrel nemcsak befelé, hanem kifelé is elmozdíthatjuk a felületet.

Trimmelt felületek

A parametrikus felületek az egységnégyzetet vetítik a háromdimenziós tér egy részhalmazára, ami meg is látszik az eredményen. A felületek négyszögszerűek lesznek, amelyekben nincsenek lyukak, és a határukon felismerhetők a paraméternégyzet sarkai. Például egy arc modelljénél nem tudjuk kialakítani a szájat, orrlyukakat illetve a szemüreget, legfeljebb itt benyomhatjuk a felületet. Minél erősebben nemlineáris $\vec{r}(u,v)$ függvényeket használunk, a négyszög alap egyre kevésbé lesz jellemző, sőt, ha a függvény nem folytonos akkor akár lyukakat is készíthetünk. Az erősen nemlineáris és szakadásos függvények azonban nehezen kezelhetőek, nem véletlen, hogy az idáig tárgyalt megoldások folytonos, legfeljebb harmadfokú polinomokkal dolgoznak. A szakadásos függvények helyett egy másik megoldást érdemes alkalmazni, amely továbbra is egyszerű, legfeljebb harmadfokú polinomokkal definiált felületekkel dolgozik, viszont kivágja belőlük a lyukakat és levágja róluk a felesleges részeket. Ezt a vágási eljárást nevezzük *trimmelés*nek.

A trimmeléshez egy görbét veszünk fel a felületen és azt mondjuk, hogy mindazon felületrészletet eltávolítjuk, amelyek a görbe által határolt rész belsejében (lyukak) vagy külsejében (levágás) található. Igen ám, de hogyan biztosítjuk, hogy egy térbeli görbe a felületre illeszkedjen, és hogyan döntjük el, hogy egy pont most a határolt rész belsejében vagy azon kívül van-e? Mindkét dolog rendkívül egyszerű, ha a trimmelő görbét nem közvetlenül a felületen, hanem abban az egységnégyzetben vesszük fel, amelyet a felületegyenletek a háromdimenziós térbe vetítenek.



3.36. ábra. *Eredeti felület és a trimmelt változata*

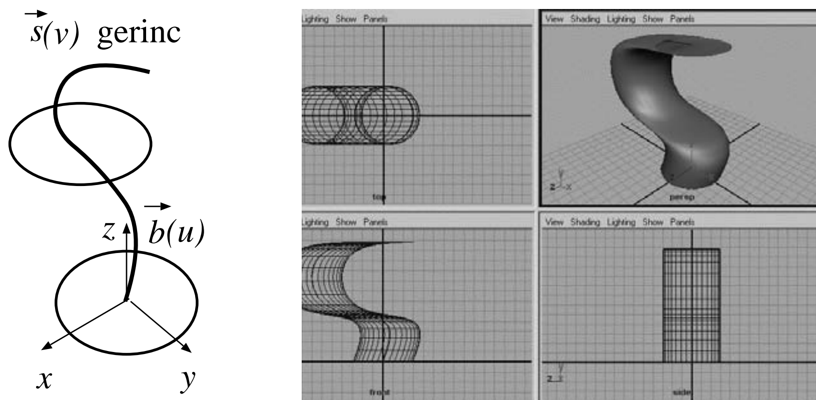
Jelöljünk ki az egységnégyzet belsejében vezérlőpontokat és azokra illesszünk egy $u(t), v(t)$ önmagában zárt síkgörbét a görbetervezésnél megismert eljárások bármelyikével (akár úgy, hogy az egymást követő vezérlőpontokat szakaszokkal kötjük össze)! Az $u(t), v(t)$ síkgörbét a felület egyenletébe helyettesítve a felületen futó térgörbét kapunk:

$$\vec{r}(t) = \vec{r}(u(t), v(t)).$$

A felület egy adott pontjáról úgy dönthetjük el, hogy az áldozatául esett-e a trimmelésnek, ha meghatározzuk a pontnak megfelelő u,v paraméterpárt, majd megvizsgáljuk, hogy az a trimmelő görbe által határolt tartomány belsejében vagy azon kívül van-e. A vizsgálathoz egy félegyenest indítunk a pontból egy tetszőleges irányba és megszámoljuk, hogy hányszor metszettük a határgörbét. Páratlan számú metszés esetén a tartomány belsejében, páros számú metszéskor pedig azon kívül vagyunk.

3.4.7. Kihúzott felületek

A háromdimenziós felületek létrehozását visszavezethetjük görbék megadására. Az egyik ilyen eljárás a *kihúzás* (*extruding*), amely egy profilgörbét és egy gerincgörbét használ, és az eljárás azon pontokat tekinti a felülethez tartozónak, amit a profilgörbe söpör, mialatt végighúzzuk a gerincgörbe mentén. Egy rúd párizsinál a profilgörbe kör, a gerincgörbe pedig egy, a kör síkjára merőleges szakasz.



3.37. ábra. *Állandó profil kihúzásával kapott felület négy nézetben*

Jelöljük a profilgörbét $\vec{b}(u)$-val, a gerincgörbét pedig $\vec{s}(v)$-vel! A két görbe paraméterezéséhez két különböző változót használtunk, hiszen ezeket egymástól függetlenül változtathatjuk. Az u,v paraméterpárhoz tartozó ponthoz ekkor úgy jutunk el, hogy elsétálunk a gerincgörbe $\vec{s}(v)$ pontjára, majd innen a profilgörbe síkjával párhuzamosan megtesszük még a profilgörbének megfelelő távolságot. A felületünk $\vec{r}(u,v)$ pontja tehát:

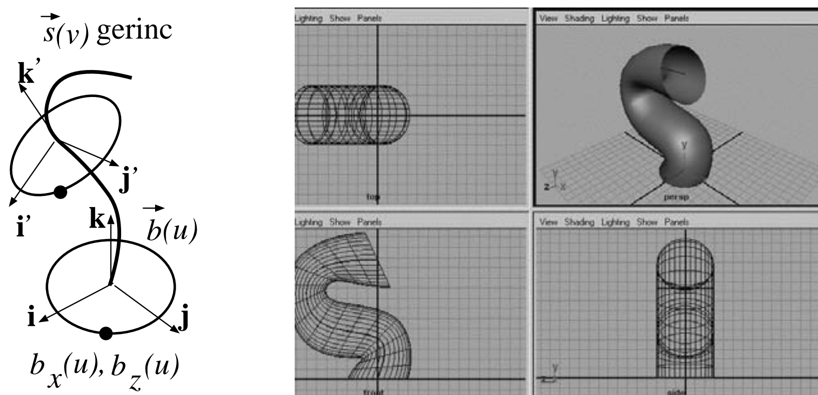
$$\vec{r}(u,v) = \vec{b}(u) + \vec{s}(v).$$

Nehézséget jelent az, hogy az eredményt nem szorzatfelület alakban kapjuk, ami akkor kínos, ha a kihúzott felületet a vezérlőpontok változtatásával még tovább szeretnénk alakítgatni. A megoldást az jelenti, hogy a profilgörbének a gerincgörbével történő kihúzása helyett a profilgörbe vezérlőpontjait a gerincgörbe vezérlőpontjaival húzzuk

ki. Tekintsük a gerincgörbe $\vec{s}_1, \ldots, \vec{s}_n$ és a profilgörbe $\vec{b}_1, \ldots, \vec{b}_m$ vezérlőpontjait. A gerincgörbe közelében lévő $\vec{s}_j$ vezérlőpontot a profilgörbe $\vec{b}_i$ vektorával eltolva, az a

$$\vec{r}_{ij} = \vec{b}_i + \vec{s}_j$$

pontba kerülne. A műveletet minden i, j párra végrehajtva a vezérlőpontok rendszerét kapjuk, amelyhez már tetszőleges szorzatfelületet — célszerűen NURBS-öt — illeszthetünk.



3.38. ábra. *A gerincre merőlegesen tartott profil kihúzása*

Amint a 3.37. ábrán látható, az ezzel a módszerrel előállított felület ellapul olyan helyeken, amikor a gerincgörbe a profilgörbe síkjára nem merőleges. Ezen úgy segíthetünk, hogy a profilgörbe adott pontjának megfelelő helyre nem a profilgörbe eredeti síkján megyünk, hanem egy olyan síkon, amely a gerincgörbére merőleges (3.38. ábra). Tegyük fel, hogy a profilgörbe az x, y síkon van, és koordinátái $b_x(u), b_y(u)$, azaz

$$\vec{b}(u) = \vec{\mathbf{i}} b_x(u) + \vec{\mathbf{j}} b_y(u),$$

ahol $\vec{\mathbf{i}}, \vec{\mathbf{j}}, \vec{\mathbf{k}}$ a Descartes-koordinátarendszer három bázisvektora. Miután a gerincgörbén az $\vec{s}(v)$ pontig eljutottunk, a $b_x(u)$ és $b_y(u)$ távolságokat egy olyan koordinátarendszer tengelyei mentén kell megtenni, amelyben $\vec{\mathbf{i}}'$ és $\vec{\mathbf{j}}'$ merőleges a gerincgörbére ebben a pontban. Egy, a görbét érintő $\vec{\mathbf{K}}'$ vektort a görbe deriváltjaként állíthatjuk elő:

$$\vec{\mathbf{K}}'(v) = \frac{d\vec{s}(v)}{dv}.$$

Az erre, és egymásra merőleges $\mathbf{I}'$ és $\mathbf{J}'$ vektorokat úgy érdemes megválasztani, hogy a felület ne csavarodjon. Ez a következőképpen lehetséges:

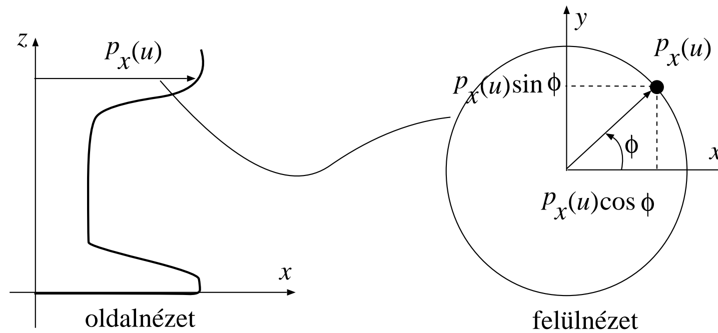
$$\vec{\mathbf{I}}'(v) = \mathbf{j} \times \vec{\mathbf{K}}'(v), \quad \vec{\mathbf{J}}'(v) = \vec{\mathbf{K}}'(v) \times \vec{\mathbf{I}}'(v).$$

Az új $\vec{\mathbf{i}}'(v)$ és $\vec{\mathbf{j}}'(v)$ egységvektorokat az $\vec{\mathbf{I}}'(v)$ és $\vec{\mathbf{J}}'(v)$ vektorok normalizálásával számíthatjuk ki. A felület u, v paraméterhez tartozó pontja pedig:

$$\vec{r}(u,v) = \vec{\mathbf{i}}'(v) b_x(u) + \vec{\mathbf{j}}'(v) b_y(u) + \vec{s}(v).$$

Ez a művelet is elvégezhető csak a vezérlőpontokra, azaz ez a felület is közelíthető egyetlen NURBS felülettel.

3.4.8. Forgásfelületek



3.39. ábra. *A forgatás paramétereinek a megadása*

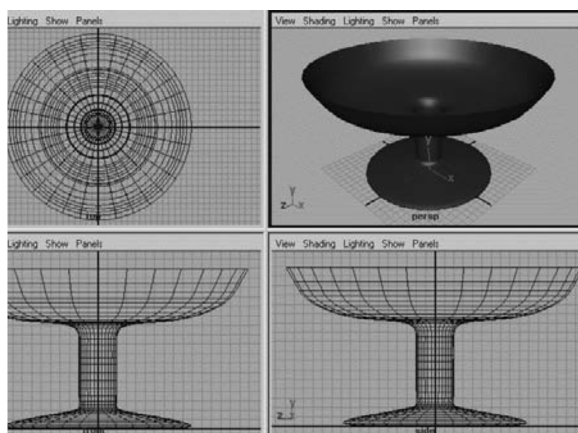
A forgásfelületek létrehozását a kihúzáshoz hasonlóan ugyancsak görbetervezésre vezetjük vissza (3.39. ábra). Most a profil a felületnek és a szimmetriatengelyén átmenő síknak a metszésvonala. A profilon kívül a forgástengelyt kell megadni. Tegyük fel, hogy a forgástengely a koordinátarendszer z tengelye, a profilgörbe pedig az x, z síkban van és paraméteres egyenlete a $[p_x(u), 0, p_z(u)]$. A $[p_x(u), 0, p_z(u)]$ pontot a z tengely körül ϕ szöggel elforgatva a $[p_x(u)\cos\phi,\ p_x(u)\sin\phi,\ p_z(u)]$ ponthoz jutunk. Ha a teljes forgásfelületet szeretnénk előállítani, a v paraméter változtatásával a teljes $[0, 2\pi]$ szögtartományon végig kell futni, így a felület pontjai:

$$\vec{r}(u,v) = [p_x(u)\cos 2\pi v,\ p_x(u)\sin 2\pi v,\ p_z(u)].$$

A szinusz és koszinusz helyett használhatunk bármilyen olyan $[c_x(v), c_y(v), 0]$ paraméteres görbét, amely a kört állítja elő. Emlékezzünk vissza, hogy a NURBS erre kompromisszumok nélkül képes. A NURBS alkalmazása ezen a helyen azért hasznos, mert így az elforgatott felületet közvetlenül NURBS szorzatfelület alakban kapjuk meg.

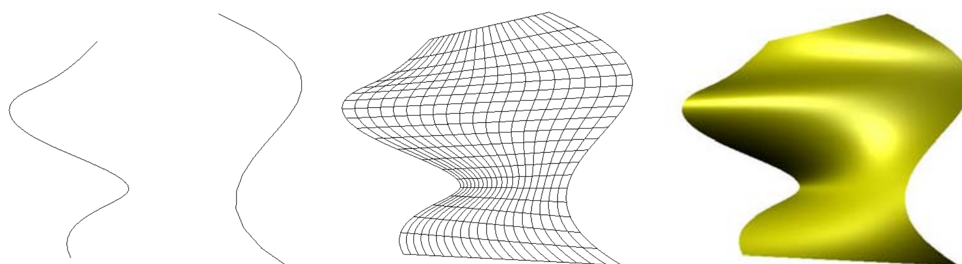
3.4.9. Felületillesztés görbékre

Az utolsó felületmodellezési módszerünk két görbe pontjainak összekötögetésével állítja elő a kívánt felületet. Az eljárást, amelyet a szakma *lofting* néven ismer, a ha-



3.40. ábra. *Forgatott felület négy nézetben*

jóépítésből örökölte a számítógépes grafika. Vegyünk fel két, ugyanazzal a változóval paraméterezett görbét ($\vec{r}_1(u)$, $\vec{r}_2(u)$), és kössük össze a két görbe azonos paraméterű pontjait szakaszokkal! A szakaszok összessége a modellezett felületet adja meg.



3.41. ábra. *Egy felület mint két paraméteres görbe pontjainak összekötögetése*

A felület egyenletének felírásához a szakasz egyenletében a paramétert v-vel jelöljük. A két egyenes u paramétereit összekötő szakasz egyenlete:

$$\vec{r}_u(v) = \vec{r}_1(u) \cdot (1 - v) + \vec{r}_2(u) \cdot v, \qquad v \in [0, 1].$$

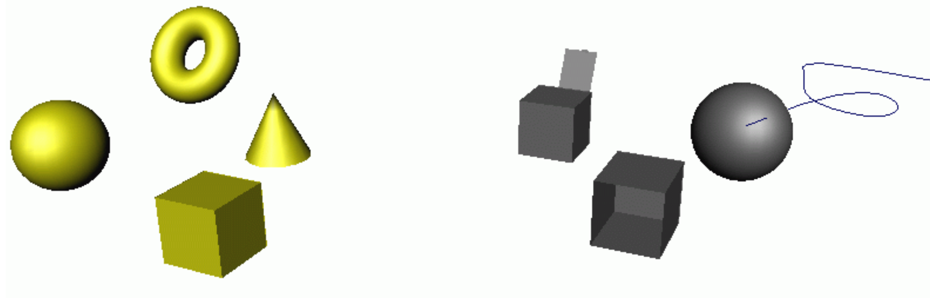
A szakaszok összességét jelentő felület egyenletét megkaphatjuk, ha a szakaszt azonosító u változót felszabadítjuk, azaz tetszőleges 0 és 1 közötti értéket megengedünk:

$$\vec{r}(u, v) = \vec{r}_1(u) \cdot (1 - v) + \vec{r}_2(u) \cdot v, \qquad u, v \in [0, 1].$$

Az összekötő szakaszok tekinthetők NURBS görbéknek, amelyeket legalább két-két vezérlőpont határoz meg. Így, ha az $\vec{r}_1(u)$ és $\vec{r}_2(u)$ görbék ugyanannyi vezérlőpontból álló NURBS görbék, akkor a keletkezett felület is NURBS szorzatfelület lesz. A szorzatfelület vezérlőpontjai az $\vec{r}_1(u)$ és $\vec{r}_2(u)$ görbék vezérlő pontjait páronként összekötő szakaszok vezérlőpontjai lesznek.

3.5. Testek

*Test*nek a 3D tér egy olyan korlátos részhalmazát nevezzük, amelyben nincsenek alacsonyabb dimenziós elfajuló részek. Egy téglatest, gömb stb. nyilván testek, de nem érdemli meg a test nevet az a ponthalmaz, amelynek egy része egy 3D kiterjedés nélküli síkot vagy vonalat formáz, vagy elszórt pontok gyűjteménye (3.42. ábra). Háromnál alacsonyabb dimenziós ponthalmazok ugyanis a valós világban nem léteznek (még a legvékonyabb papírlapnak is van valamennyi vastagsága). Az olyan ponthalmazokat, amelyek testnek tekinthetők, *reguláris halmaz*oknak nevezzük. A folyamat pedig, amelyben az elfajult részektől megszabadulunk, a *regularizáció*.



3.42. ábra. *Testek és testnek nem nevezhető 3D ponthalmazok*

A következőkben olyan testmodellezési eljárásokkal ismerkedünk meg, amelyeknél a kapott test érvényességét maga az eljárás garantálja.

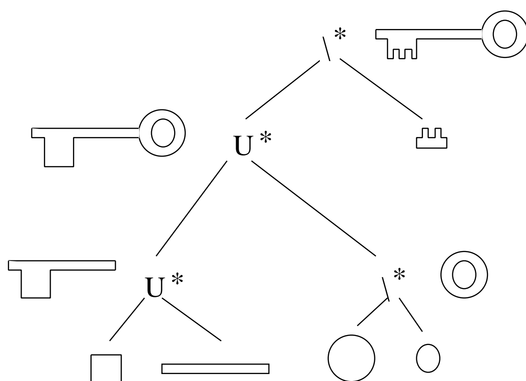
3.5.1. Konstruktív tömörtest geometria alapú modellezés

A *konstruktív tömörtest geometria* (*Constructive Solid Geometry*, *CSG*) az összetett testeket *primitív testek*ből halmazműveletek (egyesítés, metszet, különbség) alkalmazásával építi fel (3.43. ábra).

Annak érdekében, hogy a keletkező test mindig kielégítse a testekkel szemben támasztott követelményeinket — azaz ne tartalmazzon alacsonyabb dimenziójú elfajult részeket — nem a közönséges halmazműveletekkel, hanem azok regularizált változataival

3.43. ábra. *A három alapvető halmazművelet egy nagy gömbre és 6 kis gömbre*

dolgozunk. A *regularizált halmazművelet*et úgy képzelhetjük el, hogy az eredményből minden alacsonyabb dimenziójú elfajulást kiirtunk. Például két, csak egy lapban vagy élben illeszkedő kocka metszete a közös lap vagy él, amit a regularizált metszet művelet eltávolít, tehát a két illeszkedő kocka regularizált metszete az üreshalmaz lesz.



3.44. ábra. *Összetett objektum felépítése halmazműveletekkel*

Bonyolult objektumok nem állíthatók elő a primitív testekből valamely reguláris halmazművelet egyszeri alkalmazásával, hanem egy teljes műveletsorozatot kell végrehajtani. Mivel az egyes műveleteket primitív testeken, vagy akár primitív testekből korábban összerakott összetett testeken is elvégezhetjük, a felépítési folyamat egy bináris fával szemléltethető. A fa csúcsán áll a végleges objektum, levelein a primitív objektumok, közbenső csúcspontjain pedig a műveletsor részeredményei láthatók (3.44. ábra).

3.5.2. Funkcionális reprezentáció

A *funkcionális reprezentáció* (*functional representation*, *F-Rep*[3]) a testmodellezés és az implicit felületek házasságának a gyümölcse. A felületmodellezésnél egy $f(x,y,z)=0$ egyenlettel azonosítottuk a felület pontjait, most viszont egy egyenlőtlenséget használunk 3D ponthalmazok megadására, és a testhez tartozónak tekintünk minden olyan x,y,z pontot, amely kielégíti az

$$f(x,y,z) \geq 0$$

egyenlőtlenséget. Az $f(x,y,z)=0$ egyenletnek is megfelelő pontok a test határpontjai, az $f(x,y,z)<0$ pontok pedig a testen kívül vannak.

test	$f(x,y,z)$ funkcionális reprezentáció
R sugarú *gömb*	$R^2-x^2-y^2-z^2$
$2a,2b,2c$ élű *téglatest*	$\min\{a-\lvert x\rvert, b-\lvert y\rvert, c-\lvert z\rvert\}$
z tengelyű, r (hurka) és R (lyuk) sugarú *tórusz*	$r^2-z^2-(R-\sqrt{x^2+y^2})^2$

3.1. táblázat. *Néhány origó középpontú test funkcionális reprezentációja*

3.5.3. Cseppek, puha objektumok és rokonaik

A szabadformájú, amorf testek létrehozását — a parametrikus felületekhez hasonlóan — vezérlőpontok megadására vezetjük vissza. Rendeljünk minden $\vec{r}_i$ vezérlőponthoz egy $h(R_i)$ hatásfüggvényt, amely kifejezi a vezérlőpont hatását egy tőle $R_i=|\vec{r}-\vec{r}_i|$ távolságban lévő pontban! Az összetett testnek azokat a pontokat tekintjük, ahol a teljes hatás egy alkalmas T küszöbérték felett van (3.45. ábra):

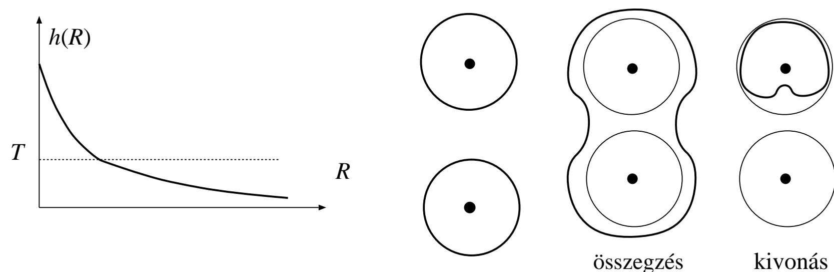
$$f(\vec{r}) = \sum_{i=1}^{n} h_i(R_i) - T, \qquad \text{ahol } R_i = |\vec{r}-\vec{r}_i|.$$

Egy hatásfüggvénnyel egy gömböt írhatunk le, a gömbök pedig cseppszerűen összeolvadnak (3.46. ábra). A kevés hatásfüggvényt tartalmazó modellek még erősen gömbszerűek, de kellő türelemmel és elengedő hatásfüggvénnyel ez a jelenség is eltüntethető. A 3.46. ábra jobb oldalán feltűnő gyilkosbálna egy japán diák 2–3 heti munkája [97].

Blinn [21] a következő hatásfüggvényeket javasolta a *csepp* (*blob*) módszerében:

$$h_i(R) = a_i \cdot e^{-b_i R^2}.$$

[3] http://cis.k.hosei.ac.jp/ F-rep



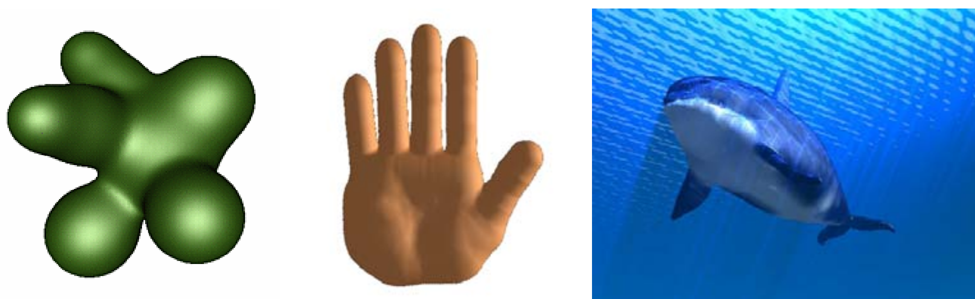
3.45. ábra. *Hatásfüggvény és hatásösszegzés*

Az a, b paraméterek vezérlőpontonként változhatnak, így egyes vezérlőpontokhoz nagyobb hatást rendelhetünk.

Nishimura[4] *metalabdái* (*metaballs*) a következő függvényt használják:

$$h(R) = \begin{cases} b(1 - 3R^2/d^2), & \text{ha } 0 < R \leq d/3, \\ 1.5b(1 - R/d)^2, & \text{ha } d/3 < R \leq d, \\ 0, & \text{ha } R > d. \end{cases}$$

A metalabda hatásfüggvénye másodfokú, tehát egy ilyen felület elmetszéséhez másodfokú egyenletet kell megoldani, szemben a cseppek által megkövetelt transzcendens egyenletekkel (transzcendens függvénynek azt nevezzük, amelynek a pontos kiértékelését nem lehet a négy alapművelet véges számú alkalmazására visszavezetni).



3.46. ábra. *Csepp és metalabda modellek*

[4]egy metalabda szerkesztő Java applet, számos más érdekes applet társaságában a http://www.eml.hiroshima-u.ac.jp/member/jrs/nis/javaexampl/demoBclp.htm címen található.

Wyvill [141] a *puha objektum*ait (*soft object*) a küszöbre alkalmazott $T = 0$ feltétellel, és az alábbi hatásfüggvényekkel építette fel:

$$h(R) = 1 - \frac{22R^2}{9d^2} + \frac{17R^4}{9d^4} - \frac{4R^6}{9d^6}.$$

Figyeljük meg, hogy a cseppek, a metalabdák és a puhaobjektumok mind gömbszimmetrikus, a távolsággal csökkenő hatásfüggvényeket adnak össze, így a modellezésben való használatuk nagyon hasonló! A függvények tényleges algebrai formájának a keletkezett objektumok megjelenítésénél és feldolgozásakor van jelentősége.

Modellezés F-Rep objektumokkal

A funkcionális reprezentáció nagy előnye, hogy geometriai alakzatok transzformációja helyett függvényeket kell változtatgatnunk, amely egyrészt egyszerűbb, másrészt sokkal rugalmasabb. Először is vegyük észre, hogy a szokásos eltolás, skálázás, elforgatás a függvényeken is elvégezhető, csak a változókon a művelet inverzét kell végrehajtani! Most azonban nem kell csak a lineáris függvényekre gondolnunk, hanem tetszőleges $\vec{r}\,' = \vec{D}(\vec{r})$ invertálható, a teret deformáló függvényeket használhatunk. A deformált alakzat funkcionális reprezentációja:

$$f^D(\vec{r}) = f(\vec{D}^{-1}(\vec{r})).$$

Például egy f objektum s_x, s_y, s_z-vel skálázott majd a p_x, p_y, p_z pontra eltolt változata:

$$f^*(x,y,z) = f\left(\frac{x}{s_x} - p_x, \frac{y}{s_y} - p_y, \frac{z}{s_z} - p_z\right).$$

A CSG *halmazművelet*eit ugyancsak leírhatjuk F-Rep műveletekkel:

- f és g metszete: $\min(f,g)$.
- f és g egyesítése: $\max(f,g)$.
- f komplemense: $-f$.

A normál metszettel és egyesítéssel kapott test felszíne csak C^0 folytonos, amin *simító-metszet* (*blending-intersection*) illetve *simító-egyesítés* (*blending-union*) alkalmazásával segíthetünk:

- f és g simító-metszete:

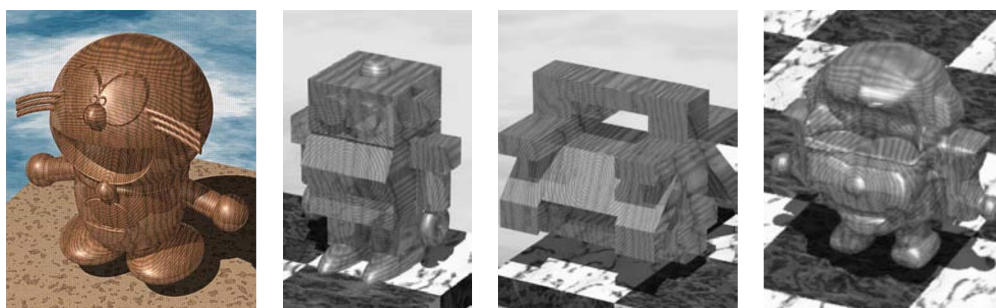
$$f + g + \sqrt{f^2 + g^2} + \frac{a}{1 + (f/b)^2 + (g/c)^2},$$

3.47. ábra. *Funkcionális reprezentációval modellezett tárgyak*

- f és g simító-egyesítése:

$$f + g - \sqrt{f^2 + g^2} - \frac{a}{1 + (f/b)^2 + (g/c)^2},$$

ahol az a, b, c paraméterekkel szabályozhatjuk a művelet eredményét és a kapott test simaságát.



3.48. ábra. *A macska, a robot és a „Japán" kanji metamorfózisa [42]*

Az F-Rep modellezés során két test közötti *átmenet* (*morph*) is könnyen kezelhető, ami pedig más modellezési módszerekben nem kevés gondot okoz. Tegyük fel, hogy két testünk van, például egy kocka és egy gömb, amelyek F-Rep alakjai f_1 és f_2. Ebből egy olyan testet, amely t részben az első objektumhoz, $(1-t)$ részben pedig a második objektumhoz hasonlít, az

$$f^{morph}(x,y,z) = t \cdot f_1(x,y,z) + (1-t) \cdot f_2(x,y,z)$$

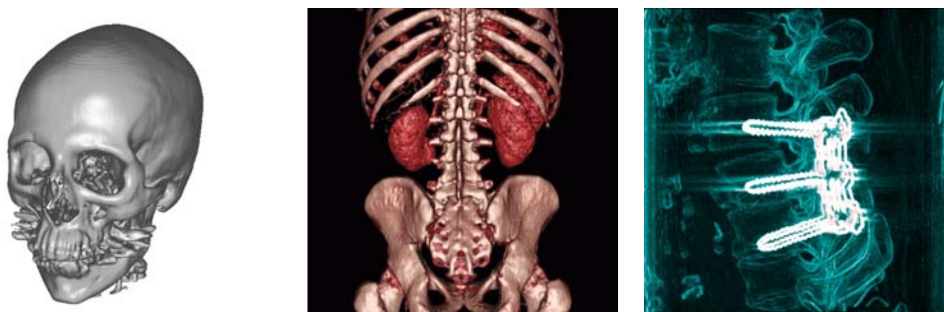
egyenlettel állíthatunk elő (3.48. ábra). Ha a t paramétert időben változtatjuk, érdekes animációt hozhatunk létre.

3.6. Térfogati modellek

Egy *térfogati modell* (*volumetric model*) a 3D tér egyes pontjaihoz rendelt $v(x,y,z)$ sűrűségfüggvény. Az egyetlen különbség a 3D testeket leíró F-Rep modell és a sűrűségfüggvény között, hogy most a függvény értelmezési tartományát, azaz a 3D teret nem osztjuk önkényesen a testhez tartozó (nem negatív értékű) és külső (negatív) tartományra, azaz nem csupán a függvény előjelét, hanem az abszolút értékét is felhasználjuk.

A térfogati modellnek tehát nincs éles határa, hanem a sűrűsége pontról-pontra változik, amit például egy ködfelhőként képzelhetünk el. A gyakorlatban térfogatmodellekre vezetnek a 3D térben elvégzett mérések (hőmérséklet- illetve sűrűségmérés), vagy a mérnöki számítások (pl. egy elektromágneses térben a potenciál-eloszlás). Az orvosi diagnosztikában használt *CT* (számítógépes *tomográf*) és *MRI* (mágneses rezonancia mérő) a céltárgy (tipikusan emberi test) sűrűségeloszlását méri, így ugyancsak térfogati modelleket állít elő [55, 33].

A térfogati modellt általában szabályos ráccsal mintavételezzük, és az értékeket egy 3D mátrixban tároljuk. Úgy is tekinthetjük, hogy egy mintavételi érték a térfogat egy kicsiny kockájában érvényes függvényértéket képvisel. Ezen elemi kockákat térfogatelemnek, vagy a *volume* és *element* szavak összevonásával *voxel*nek nevezzük.



3.49. ábra. *CT berendezéssel mért térfogati adatok megjelenítése [33]*

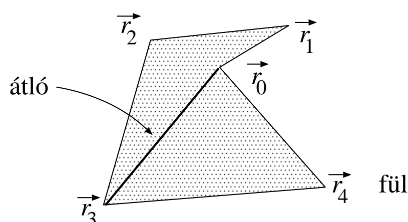
3.7. Modellek poligonhálóvá alakítása: tesszelláció

A korábbi fejezetekben olyan módszereket ismertünk meg, amelyek a 3D testeket, illetve a testek felületeit különféleképpen adják meg. Természetesen felmerülhet az igény arra, hogy egy reprezentációból a felület más módszer szerinti modelljét is előállítsuk. A különböző modellkonverziók között különösen nagy jelentősége van azoknak, amelyek tetszőleges modellt háromszög- vagy négyszöghálóvá alakítanak át, mert a képszintézis algoritmusok jelentős része csak ilyeneket képes megjeleníteni. Ezt a folyamatot *tesszelláció*nak nevezzük.

3.7.1. Sokszögek háromszögekre bontása

A célként megfogalmazott háromszögsorozathoz a sokszögek állnak a legközelebb, ezért először ezek *háromszögesítés*ével foglalkozunk. Konvex sokszögekre a feladat egyszerű, egy tetszőleges csúcspontot kiválasztva, és azt az összes többivel összekötve, a felbontás elvégezhető.

Konkáv sokszögeknél azonban ez az út nem járható, ugyanis előfordulhat, hogy a két csúcsot összekötő él nem a sokszög belsejében fut, így ez az él nem lehet valamelyik, a sokszöget felbontó háromszög oldala. A következőkben egy olyan algoritmust ismertetünk, amely egy konvex vagy konkáv $\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_n$ sokszöget háromszögekre oszt fel.



3.50. ábra. *A sokszög diagonálja és füle*

Kezdjük két alapvető definícióval:

- Egy sokszög *diagonál*ja egy, a sokszög két csúcsát összekötő olyan szakasz, amely teljes egészében a háromszög belsejében van (3.50. ábra). A diagonál tulajdonság egy szakaszra úgy ellenőrizhető, ha azt az összes oldallal megpróbáljuk elmetszeni és megmutatjuk, hogy metszéspont csak a végpontokban lehetséges, valamint azt is, hogy a diagonáljelölt egy tetszőleges belső pontja a sokszög belsejében van. Ez a tetszőleges pont lehet például a jelölt középpontja. Egy pontról úgy dönthető el, hogy egy sokszög belsejében van-e, hogy a pontból egy tetszőleges irányban egy félegyenest indítunk és megszámláljuk, hogy az hányszor metszi

a sokszög éleit. Ha a metszések száma páratlan, a pont belül van, ha páros, akkor kívül.

- A sokszög egy csúcsa *fül*, ha az adott csúcsot megelőző és követő csúcsokat összekötő szakasz a sokszög diagonálja. Nyilván csak azok a csúcsok lehetnek fülek, amelyekben a belső szög 180 foknál nem nagyobb. Az ilyen csúcsokat *konvex csúcs*oknak nevezzük, a nem konvexeket pedig *konkáv csúcs*oknak.

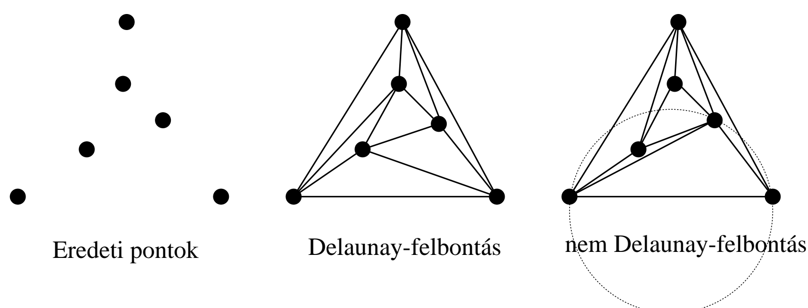
A háromszögekre bontó algoritmus füleket keres, és azokat levágja addig, amíg egyetlen háromszögre egyszerűsödik az eredeti sokszög. Az algoritmus az $\vec{r}_2$ csúcstól indul. Amikor az algoritmus az i. csúcsnál van, először ellenőrzi, hogy az előző $\vec{r}_{i-1}$ csúcspont fül-e. Ha az nem fül, a következő csúcspontra lépünk ($i = i+1$). Ha a megelőző csúcs fül, akkor az $\vec{r}_{i-2}, \vec{r}_{i-1}, \vec{r}_i$ háromszöget létrehozzuk, és az $\vec{r}_{i-1}$ csúcsot töröljük a sokszög csúcsai közül. Ha az új csúcsot megelőző csúcspont éppen a 0 indexű, akkor a következő csúcspontra lépünk. Az algoritmus minden lépésében egy háromszöget vág le a sokszögből, amely így előbb-utóbb elfogy, és az eljárás befejeződik.

3.7.2. Delaunay-háromszögesítés

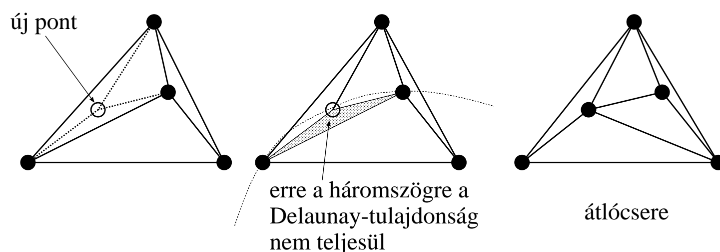
Tegyük fel, hogy egy sereg, egy síkban lévő pontot kapunk, amelyek közé éleket kell felvennünk úgy, hogy az élek nem metszik egymást, és a síktartományt háromszögekre bontják fel! A háromszögek csúcsai tehát a megadott pontok (ez alól csak az algoritmus első lépésében adunk felmentést). Ezt a feladatot nagyon sokféleképpen meg lehet oldani, ezért a lehetséges megoldások közül valamilyen szempont szerint a legjobbat kell kiválasztani. Általában előnyös, ha a háromszögek „kövérek”, nem pedig hosszan elnyúltak. A feladat tehát egy olyan illeszkedő háromszög háló előállítása, amely nem tartalmaz hosszú keskeny háromszögeket. Ezt pontosabban úgy fogalmazhatjuk meg, hogy semelyik háromszög körülírt köre sem tartalmazhat más háromszög csúcspontot. Az ilyen tulajdonságú felbontást *Delaunay-felbontás*nak nevezzük (3.51. ábra).

A Delaunay háromszögesítés inkrementális megvalósítása a [49, 50, 87] cikkekből származik. Az algoritmus egy olyan háromszögből indul, amelynek az összes kapott pont a belsejében található. Előfordulhat, hogy a megadott pontok közül nem választható ki három úgy, hogy a keletkező háromszög az összes többi pontot tartalmazza. Ilyenkor a kapott adathalmazhoz önkényesen felveszünk még további pontokat is.

Az algoritmus egy adatszerkezetet épít fel lépésenként, amely a feldolgozott pontokat, és a háromszögeket tartalmazza. A kapott pontokat egyenként adjuk hozzá az adatszerkezethez úgy, hogy a Delaunay-tulajdonság minden lépés után megmaradjon. Először az új pontot tartalmazó háromszöget azonosítjuk (3.52. ábra), majd új éleket



3.51. ábra. *Egy poligon Delaunay-felbontása (bal) és nem Delaunay-felbontása*



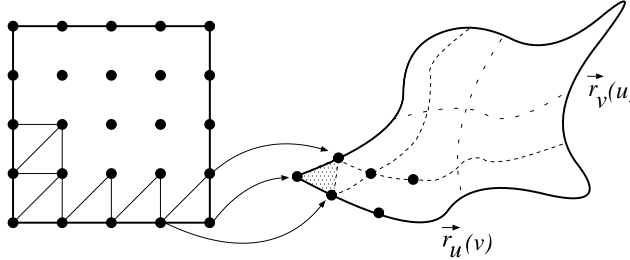
3.52. ábra. *Egy újabb pont felvétele Delaunay-hálóba*

hozunk létre az új pont és a pontot tartalmazó háromszög csúcspontjai között (a tartalmazó háromszöget ezzel három kis háromszögre bontjuk). Egy kis háromszög egy élt a tartalmazó háromszögtől örökölt, kettő pedig most született. A keletkező kis háromszögekre ellenőrizni kell, hogy nem sértik-e meg a Delaunay-tulajdonságot, azaz tartalmaznak-e a körülírt köreik más, az adatszerkezetben található pontot. Ha a háromszög nem teljesíti ezt az elvárást, akkor a kis háromszögnek a tartalmazó háromszögtől örökölt élét töröljük, és felváltjuk a törölt élre korábban illeszkedő két háromszög távolabbi csúcsait összekötő éllel (az örökölt élt egy négyszög egy átlójának tekinthetjük, amit most a négyszög másik átlójával váltunk fel). Ezzel két másik háromszög keletkezik, amelynek eredeti oldalait rekurzívan ellenőrizni kell. Belátható, hogy a rekurzív cserélgetés általában hamar véget ér, és egy új pont beszúrása többnyire csak néhány él áthelyezését igényli. Az algoritmus implementációja a [87]-ban található.

3.7.3. Paraméteres felületek és magasságmezők tesszellációja

A paraméteres felületek a paraméter tartomány egy $[u_{\min}, u_{\max}] \times [v_{\min}, v_{\max}]$ téglalapját képezik le a felület pontjaira. A magasságmezőknél pedig az $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ tartományhoz tárolunk magasság (z) értékeket. Ilyen értelemben a magasságmező egy paraméteres felületnek tekinthető, ahol az x, y koordináták közvetlenül a paramétereket

jelentik. Ezért elegendő csak a paraméteres felületek felbontásával foglalkozni, a kapott algoritmusok a magasságmezőkre is alkalmazhatók.



3.53. ábra. *Paraméteres felületek tesszellációja*

A tesszelláció elvégzéséhez a paraméter téglalapot háromszögesítjük. A paraméter háromszögek csúcsaira alkalmazva a paraméteres egyenletet, éppen a felületet közelítő háromszöghálóhoz jutunk. A legegyszerűbb felbontás az u tartományt N részre, a v-t pedig M részre bontja fel, és az így kialakult

$$[u_i, v_j] = \left[u_{\min} + (u_{\max} - u_{\min})\frac{i}{N},\ v_{\min} + (v_{\max} - v_{\min})\frac{j}{M}\right],$$

párokból kapott pontok közül az $\vec{r}(u_i, v_j)$, $\vec{r}(u_{i+1}, v_j)$, $\vec{r}(u_i, v_{j+1})$ ponthármasokra, illetve az $\vec{r}(u_{i+1}, v_j)$, $\vec{r}(u_{i+1}, v_{j+1})$, $\vec{r}(u_i, v_{j+1})$ ponthármasokra háromszögeket illeszt.

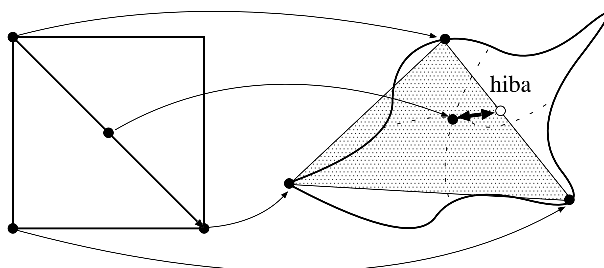
A tesszelláció lehet *adaptív* is, amely csak ott használ kis háromszögeket, ahol a felület gyors változása ezt indokolja. Induljunk ki a paraméter tartomány négyzetéből és bontsuk fel két háromszögre! A háromszögesítés pontosságának vizsgálatához a paramétertérben lévő háromszög élfelezőihez tartozó felületi pontokat összehasonlítjuk a közelítő háromszög élfelező pontjaival, azaz képezzük a következő távolságot (3.54. ábra):

$$\left|\vec{r}\left(\frac{u_1+u_2}{2}, \frac{v_1+v_2}{2}\right) - \frac{\vec{r}(u_1, v_1) + \vec{r}(u_2, v_2)}{2}\right|,$$

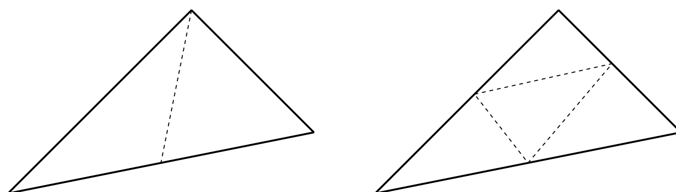
ahol (u_1, v_1) és (u_2, v_2) az él két végpontjának a paramétertérbeli koordinátái.

Ha ez a távolság nagy, az arra utal, hogy a paraméteres felületet a háromszög rosszul közelíti, tehát azt fel kell bontani kisebb háromszögekre. A felbontás történhet úgy, hogy a háromszöget két részre vágjuk a legnagyobb hibával rendelkező felezőpont és a szemben lévő csúcs közötti súlyvonal segítségével, vagy pedig úgy, hogy a háromszöget négy részre vágjuk a három felezővonala segítségével (3.55. ábra).

Az adaptív felbontás nem feltétlenül robosztus, ugyanis előfordulhat, hogy a felezőponton a hiba kicsi, de a háromszög mégsem közelíti jól a paraméteres felületet. Ebbe

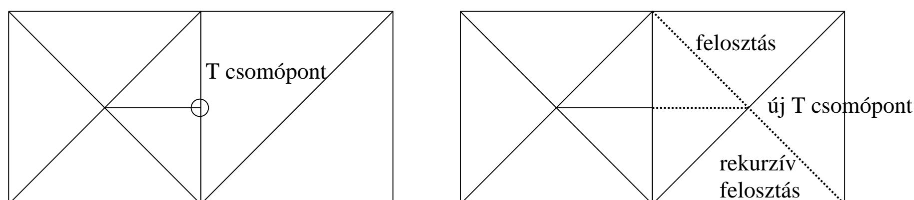


3.54. ábra. *A tesszellációs hiba becslése*



3.55. ábra. *A háromszögek felbontásának lehetőségei*

vagy beletőrődünk azzal nyugtatva a lelkiismeretünket, hogy ennek a valószínűsége azért elég csekély, vagy valamilyen robosztusabb módon döntjük el, hogy a háromszög megfelelő közelítésnek tekinthető, vagy sem. Az adaptív felbontásnál előfordulhat, hogy egy közös élre illeszkedő két háromszög közül az egyiket az élfelező ponton átmenő súlyvonallal felbontjuk, de a másikat nem, így a felbontás után az egyik oldalon lévő háromszög nem illeszkedik a másik oldalon lévő két másikhoz, azaz a felületünk kilyukad. Az ilyen problémás élfelező pontokat *T csomópont*nak nevezzük (3.56. ábra).



3.56. ábra. *T csomópontok és kiküszöbölésük erőszakos felosztással*

Amennyiben a felosztást mindig csak arra az élre végezzük el, amelyik megsérti az előírt, csak az él tulajdonságain alapuló hibamértéket, a T csomópontok nem jelenhetnek meg. Ha a felosztásban az él tulajdonságain kívül a háromszög egyéb tulajdonságai

is szerepet játszanak, akkor viszont fennáll a veszélye a T csomópontok feltűnésének, amit úgy háríthatunk el, hogy ekkor arra az illeszkedő háromszögre is kierőszakoljuk a felosztást, amelyre a saját hibakritérium alapján nem tettük volna meg.

A *trimmelt felület*ek esetén a paramétertér háromszögesítése egy kicsit bonyolultabb, ugyanis a felbontásnak illeszkednie kell a trimmelőgörbére (kényszervezérelt háromszögesítés). Első lépésben tehát a trimmelőgörbét bontjuk fel egyenes szakaszokra úgy, hogy a t paramétertartományában pontokat veszünk fel, azokat behelyettesítjük az $(u(t), v(t))$ görbeegyenletekbe, és az egymás utáni pontokat szakaszokkal kötjük össze. A keletkezett trimmelősokszögek és a paraméternégyszög határa (hacsak nem dobattuk el egy trimmelőgörbével), együttesen általában konkáv tartományt jelölnek ki. Ezt a konkáv tartományt az előző fejezet algoritmusával (fülek levágása) háromszögekre bontjuk, majd a háromszögeket a megismert hibaellenőrzéses eljárás segítségével mindaddig finomítjuk, amíg a közelítés elfogadható nem lesz.

3.7.4. CSG modellek tesszellációja

A CSG test felülete valamelyik felépítő primitív test felületéből származhat. Ez az állítás visszafelé nem igaz, ugyanis egy primitív felületének egy része nem feltétlenül jelenik meg a test felületében, mert azt egy halmazművelet eltüntethette, vagy egy tartalmazó objektumba olvaszthatta bele. A CSG modellek tesszellációját a primitív testek felületének a tesszellációjával kezdjük, majd az így kapott háromszögeket a három alábbi osztályhoz soroljuk:

1. A háromszög a CSG test határán van, tehát a tesszellált felületének része.
2. A háromszög egyetlen pontja sem tartozik a CSG test felületéhez.
3. A háromszög nem sorolható az előző két csoportba, azaz vannak olyan pontjai, amelyek a felületen vannak, de az összes pontja nem ilyen.

Nyilván az első kategóriába tartozó felületek a CSG test határát írják le, így megtartandók. A második csoport háromszögei nem lehetnek a CSG test határán, ezért eldobandók. A harmadik, bizonytalan kategóriát pedig visszavezethetjük az első kettőre úgy, hogy a bizonytalan háromszöget kisebbekre daraboljuk fel, majd megismételjük az osztályozást. A feldarabolás történhet a testek metszésvonala mentén, vagy pedig egyszerűen az élek felezésével. A felosztást addig folytatjuk, amíg minden háromszöget az első vagy második csoporthoz tudjuk sorolni, vagy pedig a háromszög mérete olyan kicsi lesz, hogy önkényes osztályozása nem befolyásolja a végeredményt.

Az algoritmus kritikus pontja annak eldöntése, hogy egy primitív felületének egy háromszöge teljes egészében a CSG test belsejében, azon kívül, vagy éppenséggel annak határán van. A CSG testet elemi primitívekből, halmazműveletekkel építjük fel,

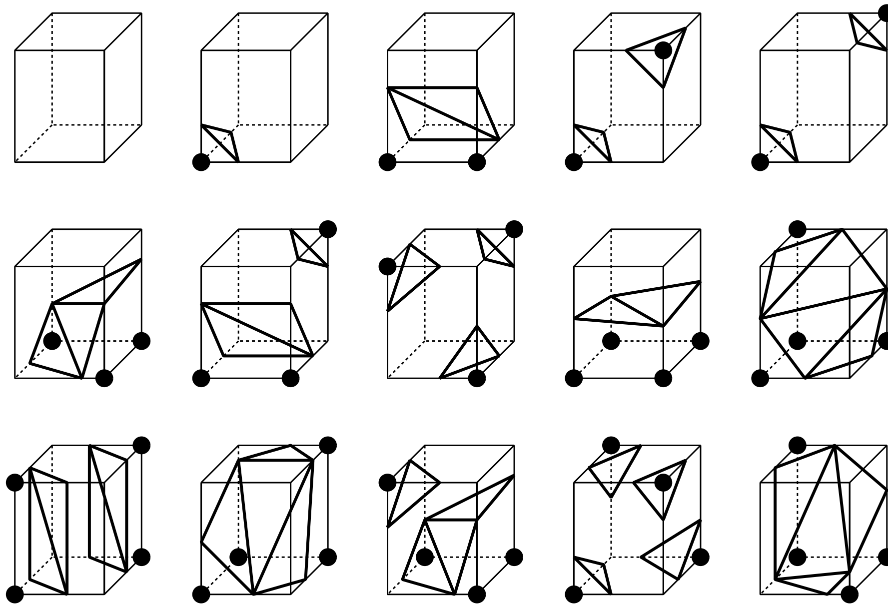
azaz egyetlen primitív halmazműveletek sorozatán megy át. Vegyük kézbe a háromszögünket és menjünk végig azokon a halmazműveleteken, amelyeken a háromszög szülőprimitíve is átesik! Az unió és a különbség akkor tartja meg a felületi háromszöget, ha az a másik testen kívül foglal helyet, a metszet pedig éppen ellenkezőleg, akkor őrzi meg a háromszöget, ha az a másik testen belül van. Ha valamikor kiderül, hogy a háromszöget teljes egészében el kell dobni, akkor megállhatunk, hiszen a háromszögünk nem tartozik a felülethez. Hasonlóképpen, ha a háromszög egyes pontjai megtartandónak, míg más pontok eldobandónak minősülnek, ugyancsak megállunk, mert egy bizonytalan esettel állunk szemben. A végső felület számára csak akkor tartjuk meg a háromszöget, ha minden halmazműveleten sikeresen túljutott, és mindenhol megtartandónak találtatott.

3.7.5. Funkcionális és térfogati modellek tesszellációja

Egy térfogati modellből elvileg úgy nyerhetünk felületeket, hogy azonosítjuk a 3D térfogat *szintfelület*eit, azaz azon 2D ponthalmazokat, ahol a $v(x,y,z)$ megegyezik a megadott szintértékkel. A funkcionális reprezentációnál a felületet definíciószerűen a zérus szintérték képviseli, tehát a zérushoz tartozó szintfelületet kell előállítani. A térfogati modellek általában mintavételezett formában egy 3D tömbben, úgynevezett *voxeltömb*ben állnak rendelkezésre, a funkcionális modellből pedig mintavételezéssel hozhatunk létre voxeltömböt. A továbbiakban a voxeltömbben két rácspontot szomszédosnak nevezünk, ha két koordinátájuk páronként megegyezik, a harmadik koordináták különbsége pedig éppen az adott tengely menti rácsállandó. A rács pontjaiban ismerjük a függvény pontos értékét, a szomszédos rácspontok közötti változást pedig általában lineárisnak tekintjük. A voxeltömb alkalmazása azt jelenti, hogy az eredeti függvény helyett a továbbiakban annak egy voxelenként *tri-lineáris* közelítésével dolgozunk (a tri-lineáris jelző arra utal, hogy a közelítő függvényben bármely két koordinátaváltozó rögzítésével a harmadik koordinátában a függvény lineáris). A lineáris közelítés miatt két szomszédos rácspont közötti él legfeljebb egyszer metszheti a közelítő felületet, hiszen a lineáris függvénynek legfeljebb egyetlen gyöke lehet.

A felületet háromszöghálóval közelítő módszer neve *masírozó kockák algoritmus* (*marching cubes algorithm*). Az algoritmus a mintavételezett érték alapján minden mintavételezési pontra eldönti, hogy az a szintértéknél kisebb vagy nagyobb-e. Ha két szomszédos mintavételezési pont eltérő típusú, akkor közöttük felületnek kell lennie. A határ helyét és az itt érvényes normálvektort a szomszédos mintavételezési pontok közötti élen az értékek alapján végzett lineáris interpolációval határozhatjuk meg.

Végül az éleken kijelölt pontokra háromszögeket illesztünk, amelyekből összeáll a közelítő felület. A háromszögillesztéshez figyelembe kell venni, hogy a tri-lineáris felület a szomszédos mintavételezési pontokra illeszkedő kocka éleinek mindegyikét legfeljebb egyszer metszheti. A kocka 8 csúcsának típusa alapján 256 eset lehetséges,



3.57. ábra. *Egy voxelenkénti tri-lineáris implicit függvényű felület és egy voxel lehetséges metszetei. Az ábrán az azonos típusú mintavételezett értékeket körrel jelöltük.*

amiből végül 15 topológiailag különböző — azaz egymásból elforgatással nem létrehozható — konfiguráció különíthető el (3.57. ábra).

Az algoritmus sorra veszi az egyes voxeleket és megvizsgálja a csúcspontok típusát. Rendeljünk a szintérték alatti csúcspontokhoz 0 kódbitet, a szintérték felettiekhez pedig 1-et. A 8 kódbit kombinációja egy 0–255 tartományba eső kódszónak tekinthető, amely éppen az aktuális metszési esetet azonosítja. A 0 kódszavú esetben az összes sarokpont a testen kívül van, így a felület a voxelt nem metszheti. Hasonlóan, a 255 kódszavú esetben minden sarokpont a test belsejében található, ezért a felület ekkor sem mehet át a voxelen. A többi kódhoz pedig egy táblázatot építhetünk fel, amely leírja, hogy az adott konfiguráció esetén mely kockaélek végpontjai eltérő előjelűek, ezért metszéspont lesz rajtuk, valamint azt is, hogy a metszéspontokra miként kell háromszöghálót illeszteni. Az algoritmus részletei és programja a [118]-ban megtalálható.

3.7.6. Mérnöki visszafejtés

A fejezet végén megemlítjük, hogy a geometriai modellt mérésekkel is előállíthatjuk. A különböző, mérésen alapuló módszereket összefoglaló néven *mérnöki visszafejtés*nek (*reverse engineering*) nevezzük. Az eljárás általában kiválasztott felületi pontok helyének a meghatározásával kezdődik, amelyhez lézeres távolságmérőt, vagy sztereolátáson ala-

puló eszközöket használhatunk (9.14. fejezet). A pontfelhőhöz, például a legközelebbi szomszédok megkeresésével háromszöghálót rendelünk, a háromszöghálót pedig egyszerűsítjük, esetleg a felületeket paraméteres felületekkel közelítjük. A további részletekhez a [126, 125, 29, 19, 77] tanulmányozását ajánljuk.

4. fejezet

Színek és anyagok

Az 1. fejezetben már említettük, hogy a háromdimenziós grafikában külön definiáljuk a megjelenítendő virtuális világ geometriáját, és külön a virtuális világban található anyagok jellemzőit, majd egy későbbi fázis során rendeljük össze őket. Ezt az elvet úgy könnyű megérteni, ha a kifestő könyvekre gondolunk, hisz ezekben csak az alakzatokat rajzolják meg előre, azaz megadják a geometriát. Miután megvesszük a kifestő könyvet, színes ceruzákat választunk, azaz az anyagok lehetséges jellemzőit definiáljuk. Majd az ábrákat kiszínezzük, azaz a geometriai tulajdonságokhoz anyagjellemzőket rendelünk. A 3. fejezetben megismerkedtünk a geometria leírásának alapvető módszereivel, ebben a fejezetben pedig ceruzákat választunk a színezéshez.

4.1. A színérzet kialakulása

Amikor egy fénysugár a szembe érkezik, megtörik a szemlencsén, majd a retinára vetül. A retina kétféle fényérzékeny sejtből épül fel: a *pálcikák*ból (*rod*) és a *csapok*ból (*cone*). Míg a csapok elsődleges feladata a színek érzékelése, a pálcikákra csak a fény erőssége, intenzitása van hatással.

Amikor a fénysugár elér egy csaphoz, vagy egy pálcikához, a sejt fényérzékeny anyaga kémiai reakciót indít be, amely egy neurális jelet eredményez. Ez a jel az idegrendszeren keresztül az agyba jut, ahol a beérkezett jelekből kialakul a *színérzet*. A kémiai reakcióért felelős anyagot *fotopigment*nek nevezzük.

Az emberi szemben három különböző típusú csapot különböztetünk meg attól függően, hogy milyen hullámhosszú beérkező fénysugár indítja be a fentebb leírt kémiai folyamatot. Miután a reakció beindult, mind a pálcikák, mind a csapok csak annyit üzennek az agynak, hogy „ehhez a sejthez fény érkezett”. Tehát a fény hullámhossza ezen a szinten elvész, csupán a különböző típusú csapsejtek jelentései alapján lehet a beérkezett fénysugár spektrumára — korlátozottan — következtetni [47].

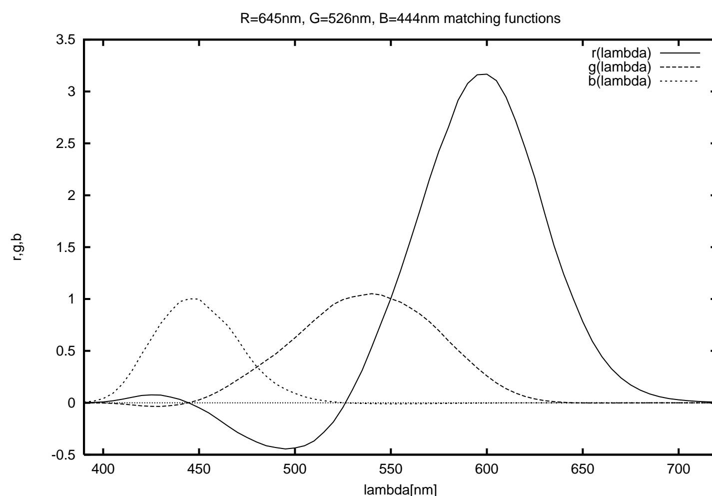
4.2. A színillesztés

Mivel az emberi szem a beérkező fényenergiát három különböző, kissé átlapolódó tartományban összegzi, ezért az agyban kialakuló színérzet három skalárral, úgynevezett *tristimulus értékek*kel is megadható. Ennek következtében a monitoron nem szükséges a számított spektrumot pontosan visszaadni, csupán egy olyant kell találni, amely a szemben ugyanolyan színérzetet kelt. Ezt a lépést nevezzük *színleképzés*nek (*tone mapping*) vagy *színillesztés*nek (*color matching*).

A lehetséges színérzetek — az elmondottak szerint — egy háromdimenziós térben képzelhetők el. A térben kijelölhető egy lehetséges koordinátarendszer oly módon, hogy kiválasztunk három elég távoli hullámhosszt, majd megadjuk, hogy három ilyen hullámhosszú fénynyaláb milyen keverékével kelthető az adott érzet. A komponensek intenzitásait *tristimulus koordinátáknak* nevezzük. Az alábbi egy megfelelő készlet, amely az önmagukban vörös (*red*), zöld (*green*) és kék (*blue*) színérzetet okozó hullámhosszakból áll:

$$\lambda_{\text{red}} = 645 \text{ nm}, \quad \lambda_{\text{green}} = 526 \text{ nm}, \quad \lambda_{\text{blue}} = 444 \text{ nm}.$$

Egy tetszőleges λ hullámhosszú fénynyaláb keltette ekvivalens színérzetet ezek után az $r(\lambda)$, $g(\lambda)$ és $b(\lambda)$ *színillesztő függvény*ekkel adunk meg, amelyeket fiziológiai mérésekkel vehetünk fel (4.1. ábra). Tehát ha például egy 500 nm hullámhosszú, egységnyi teljesítményű fénysugár érkezik a szembe, akkor az agyban a 4.1. ábráról leolvasható ($r(500)$, $g(500)$, $b(500)$) hármassal hasonló színérzet kelthető.



4.1. ábra. *Az $r(\lambda)$, $g(\lambda)$ és $b(\lambda)$ színillesztő függvények*

Amennyiben az érzékelt fénynyalábban több hullámhossz is keveredik (a fény *nem monokromatikus*), az R,G,B tristimulus koordinátákat az alkotó hullámhosszak által keltett színérzetek összegeként állíthatjuk elő. Ha a fényenergia spektrális eloszlása $\Phi(\lambda)$, akkor a megfelelő koordináták:

$$R = \int_{\lambda} \Phi(\lambda) \cdot r(\lambda)\, d\lambda, \qquad G = \int_{\lambda} \Phi(\lambda) \cdot g(\lambda)\, d\lambda, \qquad B = \int_{\lambda} \Phi(\lambda) \cdot b(\lambda)\, d\lambda.$$

Két eltérő spektrumhoz is tartozhat ugyanaz a színérzet, hisz két függvénynek is lehet ugyanaz az integrálja. A hasonló színérzetet keltő fénynyalábokat *metamer*eknek nevezzük.

Figyeljük meg a 4.1. ábrán, hogy az $r(\lambda)$ függvény (kisebb mértékben a $g(\lambda)$ is) az egyes hullámhosszokon negatív értékeket vesz fel! Ez azt jelenti, hogy van olyan monokromatikus fény, amelynek megfelelő színérzetet nem lehet előállítani a megadott hullámhosszú fénynyalábok keverékeként, csak úgy, ha előtte az illesztendő fényhez vörös komponenst keverünk. Tekintve, hogy a monitorok szintén a fenti hullámhosszú fénynyalábok nemnegatív keverékével készítenek színes képet, lesznek olyan színek, amelyeket a számítógépünk képernyőjén sohasem reprodukálhatunk.

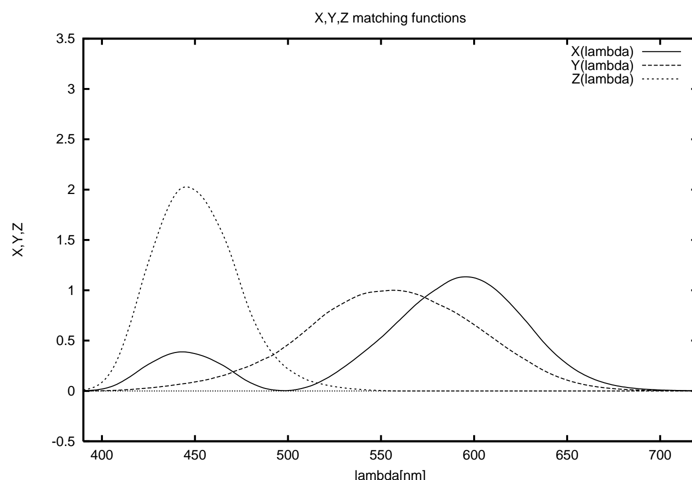
4.3. A színek definiálása

Mint megállapítottuk, a színérzetek terét egy háromdimenziós térként képzelhetjük el, amelyben a tér pontjainak azonosításához egy koordinátarendszert kell definiálnunk. Mivel a koordinátarendszerek számtalan különböző módon megadhatók, így a színek is többféleképpen definiálhatók.

A színillesztés során egy színérzetet a vörös, a zöld és a kék színillesztő függvényekkel adtunk meg, így a színeket az úgynevezett *RGB színrendszer*ben határoztuk meg. Az alábbi `Color` osztály RGB színrendszerrel dolgozik:

```
//============================================
class Color {
//============================================
public:
    float r, g, b; // az R, G, B színkomponensek

    Color(float rr, float gg, float bb) { r = rr; g = gg; b = bb; }
    Color  operator+(const Color& v)    { return Color(r+v.r, g+v.g, b+v.b); }
    Color  operator*(float s)           { return Color(r*s, g*s, b*s); }
    Color  operator*(const Color& v)    { return Color(r*v.r, g*v.g, b*v.b); }
    float  Luminance(void)              { return (r+g+b)/3.0; }
 }
```

Az RGB színrendszerben negatív értékek is lehetségesek, amelyek problémákat okozhatnak. Ezen okokból kifolyólag gyakran használjuk az *XYZ színrendszer*t is, amelyet 1931-ben a CIE (*Commission Internationale de l'Eclairage*) definiált. Az XYZ színrendszert az $X(\lambda)$, $Y(\lambda)$ és $Z(\lambda)$ színillesztő függvények adják meg, amelyek már nem vehetnek fel negatív értékeket (4.2. ábra).



4.2. ábra. *Az $X(\lambda)$, $Y(\lambda)$ és $Z(\lambda)$ színillesztő függvények*

Az XYZ színrendszer a látható színek pusztán matematikai leírása, ugyanis az $X(\lambda)$, $Y(\lambda)$ és $Z(\lambda)$ színillesztő függvények hullámhosszhoz nem köthetők. Mivel a legtöbb megjelenítő az RGB színrendszerrel dolgozik, ezért minden egyes eszközre külön meg kell adni a szabványos XYZ rendszerből az eszköznek megfelelő RGB-be átvivő transzformációt. Ezt például katódsugárcsöves megjelenítő esetében a foszforrégetek által kisugárzott fény X, Y, Z koordinátáinak és a monitor *fehér pont*jának[1] ismeretében tehetjük meg. Az alábbiakban a szükséges transzformációt szabványos NTSC foszforrétegek és fehér pont esetén adjuk meg [47]:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.967 & -0.548 & -0.297 \\ -0.955 & 1.938 & -0.027 \\ 0.064 & -0.130 & 0.982 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}.$$

Az RGB és az XYZ színrendszereken felül még számos modell létezik (például HSV, HLS, YUV, CMYK stb.). Ezekre azonban könyvünk keretein belül nem térünk ki, ám a kedves Olvasó a [43, 118] könyvekben részletesen olvashat róluk.

[1]A fehér pont a monitor fehér fényének *színhőmérséklet*ét jelenti, azaz azt a hőfokot, amelyre egy ideális fekete testet hevítve, a sugárzó a monitor fehér fényével azonos színt bocsát ki magából.

4.4. Színleképzés a háromdimenziós grafikában

A hardver által megengedett (R,G,B) értékek pozitívak és általában a [0,255] tartományba esnek. Tehát nem állítható elő valamennyi valós szín, egyrészt a színillesztő függvények negatív tartományai, másrészt pedig a korlátozott színdinamika miatt. A mai monitorok által létrehozható színek intenzitásainak aránya kb. a százas nagyságrendbe esik, míg az emberi szem akár 10^{10} nagyságú tartományt is át tud fogni úgy, hogy az egyszerre érzékelt fényességhez adaptálódik. Ezért látunk jól a vakító napsütésben és a pislákoló csillagok fényénél is.

A megjelenítés érdekében a számított színt skálázni, illetve torzítani kell. Ezt a torzítást *színleképző operátor*nak (*tone-mapping operator*) nevezzük. A következőkben összefoglaljuk a legfontosabb skálázási lehetőségeket [89].

Jelöljük a számított színintenzitást I-vel, amely most a vörös, a zöld és a kék komponensek bármelyikét jelentheti, a rasztertárba írt és a monitor által ténylegesen megjelenített fizikai értéket pedig D-vel! A feladat tehát egy olyan $I \to D$ leképzést találni, amely a számított színeket hűen visszaadja, de figyelembe veszi a monitor illetve a hardver lehetőségeit és az emberi látórendszer tulajdonságait is.

A legegyszerűbb leképzés a *lineáris skálázás*, amely a maximális számított színintenzitást a hardver által előállítható maximális színintenzitásra képezi le:

$$D = \frac{D_{\max}}{I_{\max}} \cdot I.$$

A lineáris skálázás használhatatlan eredményt ad, ha a fényforrás is látszik a képen, hiszen a kép túlságosan sötét lesz. Ezen úgy segíthetünk, hogy az $I_{\max}$ értéket a képen levő azon pixelek színértékeinek maximumaként keressük meg, amelyben nem fényforrás látszik. A látható fényforrás-értékek színe ennek következtében $D_{\max}$-ot meghaladhatja, tehát a színértéket $D_{\max}$-ra kell vágni[2].

Ismert tény, hogy az emberi érzékelés logaritmikus, amely nemlineáris skálázásnak is létjogosultságot ad. Az egyik legegyszerűbb nemlineáris modell a *Schlick-leképzés*:

$$D = D_{\max} \cdot \frac{p \cdot I}{p \cdot I - I + I_{\max}},$$

ahol p egy alkalmasan választott paraméter. Legyen G a legsötétebb nem fekete szürke szín, N pedig a fizikai eszköz által megjeleníthető intenzitások száma (tipikusan 255)! Ekkor az ismeretlen p paraméter:

$$p = \frac{G \cdot I_{\max} - G \cdot I_{\min}}{N \cdot I_{\min} - G \cdot I_{\min}}.$$

[2] A jelenséget a fényképészetben „beégésnek” nevezik.

4.5. A hétköznapi életben előforduló anyagok

A háromdimenziós grafikában egy pixelen keresztül a kamerába jutó spektrum függ a felület optikai tulajdonságaitól, amelyek pedig a felület anyagának jellegzetességeire vezethetők vissza. Ahhoz, hogy ezt a területet jobban megérthessük, először a hétköznapi életben előforduló anyagok tulajdonságait érdemes rendszereznünk.

A fényt (pontosabban energiát) kibocsátó felületek *emittáló* anyaggal rendelkeznek. Ilyen például a Nap, a villanykörte izzószálja, de a gyerekek foszforeszkáló játékfigurái is. A háromdimenziós grafikában ezeket *fényforrás*oknak nevezzük, hisz az általuk kibocsátott fény világítja meg a virtuális világot, miattuk látunk a „virtuális sötétben”.

Ha egy frissen meszelt falra nézünk, akkor az minden irányból ugyanolyan színűnek tűnik, de ugyanezt tapasztalhatjuk például a homoknál is. A *diffúz* felület anyaga a beérkező fénysugár energiáját minden lehetséges irányba azonos intenzitással veri vissza. Erre a köznyelvben gyakran a *matt* jelzőt használjuk.

Ha tükörbe nézünk, magunkat és a környezetünket látjuk benne. Tudjuk azt is, hogy a tükrökön kívül számos olyan anyag létezik, amely többé-kevésbé tükröz. Az ilyen anyaggal rendelkező felületeken inkább csak a fényforrások fedezhetők fel. A *spekuláris* vagy más néven *tükröző* felületek anyaga a beérkező fénysugár energiájának legnagyobb részét az ideális visszaverődési irány környezetébe veri vissza. A köznyelvben gyakran a *csillogó* vagy *polírozott* jelzőt használjuk rájuk. Felhívjuk a figyelmet a spekuláris anyagok speciális esetére, az *ideális tükör*re, amely bár a való életben nem létezik, de a háromdimenziós grafika gyakran használja. Az ideális tükör felületére teljesül a *geometriai optika visszaverődési törvénye*, amely azt mondja ki, hogy a beérkező és a visszaverődő fénysugár, valamint a felületi ponthoz tartozó normálvektor egy síkban helyezkedik el, ráadásul a beesési szög megegyezik a visszaverődési szöggel.

Télen jó bent ülni a meleg szobában, ám előszeretettel nézünk ki az ablakon, hogy megcsodáljuk a csillogó, hófödte tájat. Ilyenkor fel sem merül bennünk, hogy ha az ablak anyaga nem eresztené át a hóról visszaverődő fénysugarakat, akkor ezért a látványért bizony ki kellene mennünk a hidegbe. Ugyanígy már elég korán megtanuljuk, hogy ha egy nagyítót teszünk egy papírlap fölé, akkor a nagyító összegyűjti a Nap fényét és egy idő múlva a papír meggyullad. Ez is annak köszönhető, hogy a nagyítóban levő lencse anyaga átereszti a fényt, sőt úgy töri meg a beérkező fénysugarakat, hogy azokat a papírlapon egy pontba gyűjti. Az *átlátszó* (*transparent*) felületek a beérkező fénysugár energiáját elhanyagolható, vagy minimális veszteséggel eresztik át.

Számos olyan anyaggal találkozunk, amely a beérkező fénysugár egy jó részét magába engedi, ám csak kis része jut át az anyagon, nagyobb része elenyészik vagy a belépés oldalán lép ki. Ilyen például a tej, a márvány, de az emberi bőr is. Összefoglaló néven ezeket *áttetsző* (*translucent*) anyagoknak nevezzük. Jellemző rájuk, hogy csak „homályosan” látunk keresztül rajtuk, a túloldalukon levő objektumoknak csak a

körvonalát tudjuk kivenni.

Ha egy CD-t, egy szatén ruhát vagy egy csiszolt fémfelületet a tengelye körül forgatunk, akkor változik a színe. Az *anizotróp* felületek anyaga olyan, hogy a felületet tengelye körül forgatva hiába tartjuk meg a beeső és a visszaverődési szögeket, a felületek más színt mutatnak. A legtöbb felület azonban *izotróp*, azaz ha a felületet a tengelye körül úgy forgatjuk, hogy a beeső és a visszaverődési szögek állandók, akkor a felületet mindig ugyanolyan színűnek látjuk.

A hétköznapi életben előforduló anyagok legtöbbje nem sorolható csak az egyik vagy csak a másik fenti kategóriába, hanem ezek valamilyen keverékeként áll elő, ezért ezeket *összetett* anyagoknak nevezzük.

4.6. Anyagok a háromdimenziós grafikában

A körülöttünk levő anyagok tulajdonságai hihetetlenül széles skálán mozognak: lehetnek színesek, érdesek, kicsit fényesek, mattok, tükrösek, áttetszőek stb. Mindezeket a hatásokat a háromdimenziós grafikának is meg kell tudnia jeleníteni, ráadásul úgy, hogy a számítási időt ne növeljük meg túlságosan. Ennek érdekében számos egyszerűsítést végzünk. Például a testek anyagjellemzőit csak egész felületi elemekre vonatkozóan adjuk meg, nem pedig pontonként. Ekkor ugyanis az egész feladat az alábbi két kérdésre vezethető vissza:

- A felület képes-e magából „fényt” kibocsátani?
- Ha egy fénysugár a felület egy pontjához érkezik, akkor a felület anyaga hogyan reagál rá?

A következő két alfejezetben ezeket a kérdéseket vizsgáljuk meg.

4.6.1. Fényforrások

A lámpagyártók katalógusaiban minden fényforráshoz megadják az általuk kibocsátott fény színét (spektrális eloszlását), erősségét (intenzitását), különböző irányokba való eloszlását. A lámpagyártók a fényforrások ezen fotometriai tulajdonságait általában az *IES*, a *CIBSE* és az *EULUMDAT* szabványos formátumokban írják le.

A háromdimenziós grafikában leginkább a globális illumináció (8. fejezet) igényli a fényforrások pontos geometriai és fotometriai tulajdonságainak megadását. A játékokban és a valós idejű képszintézis programokban azonban jellemzően *absztrakt fényforrás*okat használunk, amelyekkel sokkal könnyebb számolni.

Az absztrakt fényforrások legfontosabb típusai a következők:

- A *pontszerű fényforrás* (*point light*) a háromdimenziós világ egy pontjában található, kiterjedése nincs. A háromdimenziós tér egy tetszőleges $\vec{p}$ pontjában a sugárzási irány a $\vec{p}$ pontot és a fényforrás helyét összekötő vektor. Az intenzitás a távolság négyzetének arányában csökken. Az elektromos izzó jó közelítéssel ebbe a kategóriába sorolható.
- Az *irány-fényforrás* (*directional light*) végtelen távol levő sík sugárzónak felel meg. Az irány-fényforrás iránya és intenzitása a tér minden pontjában azonos. A Nap irány-fényforrásnak tekinthető — legalábbis a Földről nézve.
- Az *ambiens fényforrás* (*ambient light*) minden pontban és minden irányban azonos intenzitású.
- A *szpotlámpa* (*spotlight*) a virtuális világ egy pontjában található, iránnyal és kúpos hatóterülettel rendelkezik. A zseblámpa szpotlámpának tekinthető.
- Az *égbolt fény* (*skylight*) akár irányfüggő is lehet, és akkor jelentkezik, ha az adott irányban semmilyen tárgy sincsen.

4.6.2. A kétirányú visszaverődés eloszlási függvény

Térjünk át a felület–fény kölcsönhatására! Legyen L^{in} a beérkező, L pedig a visszavert fényintenzitás! Továbbá jelölje az $\vec{L}$ a fényforrás irányába mutató egységvektort, a $\vec{V}$ a nézőirányba mutató egységvektort, az $\vec{N}$ a felület normálvektorát, a θ' pedig a normálvektor és a megvilágítási irány közötti szöget (4.3. ábra) ! Tekintsük az

$$f_r(\vec{L},\vec{V}) = \frac{L}{L^{in} \cdot \cos\theta'} \tag{4.1}$$

hullámhossztól függő elsődleges anyagjellemzőt, amelyet *kétirányú visszaverődés eloszlási függvény*nek vagy röviden *BRDF*-nek (*Bi-directional Reflection Distribution Function*) nevezünk! A „kétirányú” jelző abból származik, hogy ez rögzített felületi normálvektor mellett a megvilágítási és a nézeti irányoktól függ.

Felmerülhet a kérdés, hogy miért nem a visszavert és a beérkező intenzitások hányadosát használjuk anyagjellemzőként, és miért osztunk a megvilágítási szög koszinuszával. Ennek egyrészt történeti okai vannak, másrészt ekkor a függvény a valós anyagokra szimmetrikus lesz, amelynek jelentőségét a 8. fejezetben ismerjük majd meg (lásd Helmholtz-féle reciprocitás törvény).

Az anyagok BRDF adataihoz többféle módon juthatunk:

- az adatokat már megmérték helyettünk és elérhetővé tették azokat [81, 99],
- valakit megbízunk ezzel a mérési feladattal: egy komoly méréssorozat általában több tízezer dollárba kerül, így ezt a megoldást nem sokan alkalmazzák,
- otthon összerakunk egy kevésbé precíz, de azért megbízható BRDF mérőt [136].

Vegyük azonban észre, hogy például egy tégla esetében annak minden felületi pontjához az összes lehetséges megvilágítási és nézeti irányra, valamint néhány reprezentatív hullámhosszra hozzá kellene rendelni egy függvényértéket! Ez olyan óriási adatmennyiséget jelentene, amelynek tárolására egy mai személyi számítógép valószínűleg nem lenne képes. A képszintézis szempontjából a legtöbb esetben ez a nagyfokú pontosság nem fontos. Például egy játékprogramban, ahol rakétákkal támadó katonákkal kell küzdenünk, a harc hevében nem is vesszük észre, hogy a katona ruhája a fényt fizikailag teljesen pontosan veri-e vissza. Ezért a mért BRDF-ek helyett legtöbbször erősen approximált, ám könnyen számítható *anyagmodell*eket alkalmazunk.

4.7. Spektrális képszintézis

A színillesztésnél elmondottak alapján akkor járunk el helyesen, ha a képszintézis során a teljes spektrumot, azaz az egyes pixeleken áthaladó energiát hullámhosszonként számítjuk ki, majd az eredményhez hasonló színérzetet keltő vörös–zöld–kék komponenseket keresünk. Ezt az eljárást *spektrális képszintézis*nek nevezzük.

Egy objektumról a kamerába jutó fény spektrumát a térben lévő anyagok optikai tulajdonságai és a fényforrások határozzák meg. Jelöljük a fényforrások által kibocsátott spektrumfüggvényt $\Phi_{light}(\lambda)$-val (ez a hullámhosszon kívül a kibocsátási ponttól és az iránytól is függhet)! Egy P pixelen keresztül a kamerába jutó spektrum a fényforrások és a BRDF spektrumának függvénye:

$$\Phi_P(\lambda) = \mathcal{L}(\Phi_{light}(\lambda), f_r(\lambda)).$$

Az $\mathcal{L}$-t a felületi geometria, az optikai tulajdonságok és a kamera állása határozza meg. A pixel R, G, B értékeit a színillesztő függvényekkel súlyozott integrálokkal számíthatjuk ki. Az integrálokat numerikus módszerekkel becsüljük. Például a vörös komponens:

$$R_P = \int_\lambda \Phi_P(\lambda) \cdot r(\lambda)\, d\lambda =$$

$$\int_\lambda \mathcal{L}(\Phi_{light}(\lambda), f_r(\lambda)) \cdot r(\lambda)\, d\lambda \approx \sum_{i=1}^{l} \mathcal{L}(\Phi_{light}(\lambda_i), f_r(\lambda_i)) \cdot r(\lambda_i) \cdot \Delta\lambda. \tag{4.2}$$

Ezt azt jelenti, hogy a fényforrások intenzitását és a felületek anyagi tulajdonságait l különböző hullámhosszon kell megadni (az l szokásos értékei 3, 8, 16). A reprezentatív hullámhosszokon a pixelen keresztüljutó teljesítményt egyenként számítjuk ki, majd a 4.2. képlet alkalmazásával meghatározzuk a megjelenítéshez szükséges R, G, B értékeket.

Gyakran azonban közelítéssel élünk, és a térben elhelyezkedő fényforrásokat és anyagokat úgy tekintjük, mintha azok csak a vörös, zöld és kék fény hullámhosszain sugároznának, illetve csak ezeken a hullámhosszokon vernék vissza a fényt. Ekkor ugyanis megtakaríthatjuk a pixelenkénti színleképzést. Bár ezáltal a kiszámítandó spektrumnak csak egy durva közelítéséhez juthatunk, a játékok és a legtöbb grafikus alkalmazás is ezzel a megoldással dolgozik.

4.8. Anyagmodellek

A továbbiakban bemutatjuk a legjellemzőbb anyagmodelleket a következő jelölések alkalmazásával: $\vec{L}$ továbbra is a vizsgált felületi pontból a fényforrás irányába mutató egységvektor, míg $\vec{V}$ a nézőirányba mutató egységvektor, $\vec{N}$ pedig a normálvektor. Az $\vec{L}$ megvilágítási irány és az $\vec{N}$ normálvektor közötti szöget továbbra is θ' jelölje, míg az $\vec{N}$ és a $\vec{V}$ nézőirány közötti szöget θ! Továbbá $\vec{R}$ legyen az $\vec{L}$ tükörképe az $\vec{N}$-re vonatkoztatva, $\vec{H}$ pedig az $\vec{L}$ és $\vec{V}$ közötti felező egységvektor!

Minden anyagmodellnél megadjuk, hogyan kell egy felületi ponthoz tartozó sugársűrűséget meghatározni, ha spektrális képszintézist alkalmazunk, illetve ha a képet csak a vörös–zöld–kék hullámhosszakon számítjuk ki.

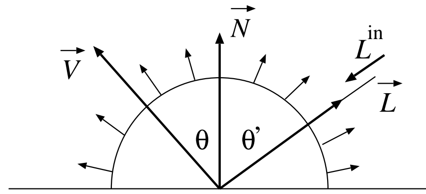
4.8.1. Lambert-törvény

Az optikailag nagyon durva, diffúz anyagok esetén a visszavert sugársűrűség független a nézeti iránytól: matt felületre nézve ugyanazt a hatást érzékeljük, ha merőlegesen nézünk rá, mintha élesebb szögben vizsgálódnánk.

A beérkező sugárnyaláb azonban nagyobb területen oszlik szét, ha a felületre nem merőleges, hanem lapos szögben érkezik. A felületnagyobbodás, és így az intenzitáscsökkenés a θ' belépési szög koszinuszával arányos, tehát a diffúz felületekről visszavert intenzitás egyetlen hullámhosszon:

$$L_\lambda = L_\lambda^{in} \cdot k_{d,\lambda} \cdot \cos\theta'.$$

Ezt az összefüggést *Lambert-törvény*nek nevezzük, amelynek időjárásra gyakorolt hatását éves ciklusokban magunk is tapasztalhatjuk. Nyáron ugyanis azért van meleg, mert a Nap a „diffúz” földet közel függőlegesen, azaz kis θ' szögben világítja meg, amelynek koszinusza egyhez közeli. Télen viszont a θ' szög nagy, amelynek koszinusza kicsi, ezért a visszavert sugársűrűség ugyancsak kicsiny.



4.3. ábra. *Diffúz visszaverődés*

A k_d visszaverődési tényező a λ hullámhossz függvénye, és alapvetően ez határozza meg, hogy fehér megvilágítás esetén a tárgy milyen színű. Ha a képet a vörös–zöld–kék hullámhosszakon számítjuk ki, akkor a visszaverődéshez három egyenletet kell felírni, a három hullámhossznak megfelelően:

$$L_R = L_R^{in} \cdot k_{d,R} \cdot \cos\theta', \quad L_G = L_G^{in} \cdot k_{d,G} \cdot \cos\theta', \quad L_B = L_B^{in} \cdot k_{d,B} \cdot \cos\theta'.$$

A fentiek és a 4.1. definíció alapján a diffúz felületek BRDF modellje:

$$f_{r,\lambda}(\vec{L},\vec{V}) = k_{d,\lambda}.$$

4.8.2. Ideális visszaverődés

Mint megállapítottuk, az *ideális tükör* a geometriai optika által kimondott visszaverődési törvény szerint veri vissza a fényt, miszerint a beesési irány, a felületi normális és a kilépési irány egy síkban van, és a θ' beesési szög megegyezik a θ visszaverődési szöggel (4.4. ábra). Az ideális tükör tehát csak a megvilágítási iránynak a normálvektorra vett tükörirányába ver vissza fényt, egyéb irányokba nem. A tükörirányban a visszavert sugársűrűség arányos a bejövő sugársűrűséggel (minden más irányban $L_\lambda = 0$):

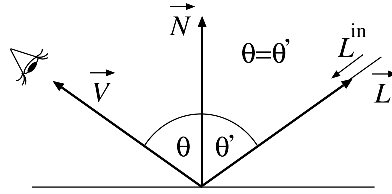
$$L_\lambda = L_\lambda^{in} \cdot k_{r,\lambda}.$$

Ha a képet csak a vörös–zöld–kék hullámhosszakon számítjuk ki, akkor a tükörirányra vonatkozó sugársűrűségre az alábbi három egyenletet kell felírni:

$$L_R = L_R^{in} \cdot k_{r,R}, \quad L_G = L_G^{in} \cdot k_{r,G}, \quad L_B = L_B^{in} \cdot k_{r,B},$$

különben pedig $L_R = L_G = L_B = 0$.

A k_r azt fejezi ki, hogy még a tökéletes tükrök is elnyelik a beérkező fény egy részét. A visszaverődési együttható a felület anyagjellemzőitől, a hullámhossztól és a megvilágítási szögtől függhet. Műanyagoknál a hullámhossz és a megvilágítási irányfüggés elhanyagolható, egyes fémeknél azonban jelentős lehet.



4.4. ábra. *Az ideális visszaverődés*

A visszavert és a beeső energia hányadát az anyag *Fresnel-együtthatója* fejezi ki, amely az anyag *törésmutatójá*ból számítható. A törésmutató komplex szám, de nemfémes anyagoknál a képzetes rész többnyire elhanyagolható. Jelöljük a törésmutató valós részét ν-vel, amely a fény vákumbeli és az anyagban mutatott sebességének arányát fejezi ki! A κ-val jelölt képzetes rész a fény csillapítását mutatja a tárgy anyagában. A *Fresnel-egyenlet*ek a visszavert és a beérkező fénynyalábok energiahányadát fejezik ki külön arra az esetre, amikor a fény polarizációja[3] párhuzamos a felülettel, és külön arra, amikor a polarizáció merőleges a felületre:

$$F_{\|}(\lambda,\theta') = \left|\frac{\cos\theta' - (\nu+\kappa j)\cdot\cos\theta_t}{\cos\theta' + (\nu+\kappa j)\cdot\cos\theta_t}\right|^2, \qquad F_{\perp}(\lambda,\theta') = \left|\frac{\cos\theta_t - (\nu+\kappa j)\cdot\cos\theta'}{\cos\theta_t + (\nu+\kappa j)\cdot\cos\theta'}\right|^2,$$

ahol $j = \sqrt{-1}$, θ_t pedig a *Snellius – Descartes törvény* által kijelölt törési szög, azaz

$$\frac{\sin\theta'}{\sin\theta_t} = \nu.$$

Ezen egyenleteket a *Maxwell-egyenletek*ből [75] származtathatjuk, amelyek az elektromágneses hullámok terjedését írják le. Nem polarizált fény esetében a párhuzamos ($\vec{E}_{\|}$) és merőleges ($\vec{E}_{\perp}$) mezőknek ugyanaz az amplitúdója, így a visszaverődési együttható:

$$k_r = F(\lambda,\theta') = \frac{|F_{\|}^{1/2}\cdot\vec{E}_{\|} + F_{\perp}^{1/2}\cdot\vec{E}_{\perp}|^2}{|\vec{E}_{\|} + \vec{E}_{\perp}|^2} = \frac{F_{\|} + F_{\perp}}{2}.$$

A Fresnel-együtthatót jól közelíthetjük a Lazányi – Schlick féle képlettel:

$$F(\lambda,\theta') \approx \frac{(\nu(\lambda)-1)^2 + (\kappa(\lambda))^2 + (1-\cos\theta')^5\cdot 4\nu(\lambda)}{(\nu(\lambda)+1)^2 + (\kappa(\lambda))^2}.$$

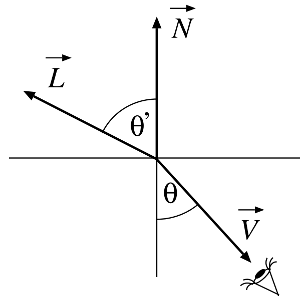
[3] Ha a fény elektromos mezővektora egy síkban változik, akkor *polarizált fény*ről beszélünk, a jelenséget pedig *polarizáció*nak nevezzük.

4.8.3. Ideális törés

Az ideális törés során a fény útját a *Snellius – Descartes törvény* írja le, miszerint a beesési irány, a felületi normális és a törési irány egy síkban van, és a beesési és törési szögekre fennáll a következő összefüggés:

$$\frac{\sin\theta'}{\sin\theta} = \nu,$$

ahol ν az anyag relatív törésmutatójának valós része (4.5. ábra). A törés azért következik be, mert a fény sebessége a törésmutató arányában megváltozik, midőn belép az anyagba.



4.5. ábra. *Az ideális törés*

A törési irányban a sugársűrűség arányos a bejövő sugársűrűséggel (minden más irányban viszont zérus):

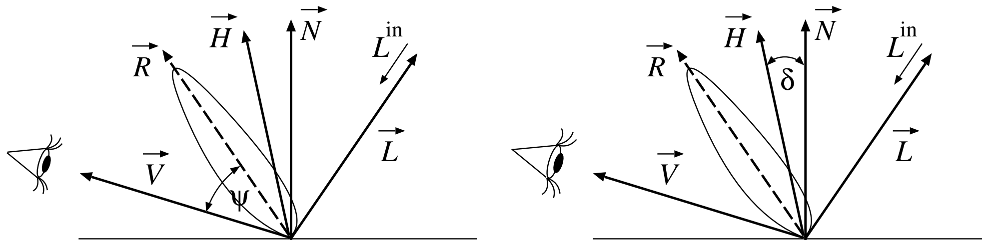
$$L_\lambda = L_\lambda^{in} \cdot k_{t,\lambda},$$

illetve csak vörös–zöld–kék hullámhosszakon számított képek esetén:

$$L_R = L_R^{in} \cdot k_{t,R}, \quad L_G = L_G^{in} \cdot k_{t,G}, \quad L_B = L_B^{in} \cdot k_{t,B}.$$

4.8.4. A spekuláris visszaverődés Phong-modellje

A körülöttünk található fényes tárgyak nem írhatók le az eddig ismertetett modellekkel, sőt azok kombinációival sem. A fényes tárgyakra az jellemző, hogy a fényt minden irányban visszaverhetik, de nem egyenletesen, mint a diffúz modellben, hanem főleg az elméleti visszaverődési irány környezetében. Ebben az esetben a visszaverődést általában két tényezőre bontjuk: egyrészt a diffúz visszaverődésre, amelyet a Lambert-törvénnyel írunk le, másrészt a tükörirány körüli csúcsért felelős spekuláris visszaverődésre, amelyre külön modellt állítunk fel.



4.6. ábra. *A spekuláris visszaverődés Phong és Phong – Blinn modellje*

Azt a jelenséget, hogy a spekuláris felületek a beérkező fény jelentős részét a tükörirány környezetébe verik vissza, modellezhetjük bármely olyan függvénnyel, amely a tükörirányban nagy értékű, és attól távolodva rohamosan csökken. Phong [101] a nézeti irány és a tükörirány közötti szöget ψ-vel jelölve a $k_s \cdot \cos^n \psi$ függvényt javasolta erre a célra, így a modelljében a spekulárisan visszavert sugársűrűség:

$$L_\lambda = L_\lambda^{in} \cdot k_{s,\lambda} \cdot \cos^n \psi,$$

míg ha a képet csak a vörös–zöld–kék hullámhosszakon számítjuk ki:

$$L_R = L_R^{in} \cdot k_{s,R} \cdot \cos^n \psi, \quad L_G = L_G^{in} \cdot k_{s,G} \cdot \cos^n \psi, \quad L_B = L_B^{in} \cdot k_{s,B} \cdot \cos^n \psi.$$

Az n hatvány a felület *fényességé*t (*shininess*) határozza meg. Ha az n nagy, akkor spekuláris visszaverődés csak a tükörirány szűk környezetében jelenik meg. A k_s faktort az elektromos áramot nem vezető anyagoknál tekinthetjük hullámhossz- és beesési szög függetlennek (egy műanyagon a fehér fény által létrehozott tükrös visszaverődés fehér), fémeknél azonban a hullámhossztól és a belépési szögtől függ (ezért látunk különbséget az arany és a réz gyűrű között, holott mindkettő sárga).

A fentiek alapján a spekuláris felületek Phong BRDF modellje:

$$f_{r,\lambda}(\vec{L},\vec{V}) = k_{s,\lambda} \cdot \frac{\cos^n \psi}{\cos \theta'}.$$

4.8.5. A spekuláris visszaverődés Phong – Blinn modellje

A tükörirány és a nézeti irány közötti „távolságot” nemcsak a szögükkel fejezhetjük ki, hanem a normálvektor, valamint a nézeti és megvilágítási irányok felezővektora közötti szöggel is (4.6. ábra). Figyeljük meg, hogy ha a nézeti irány éppen a tükörirányban van, akkor a normálvektor a felezőirányba mutat, ha pedig a nézeti irány eltávolodik a tüköriránytól, akkor a felezővektor is eltávolodik a normálvektortól!

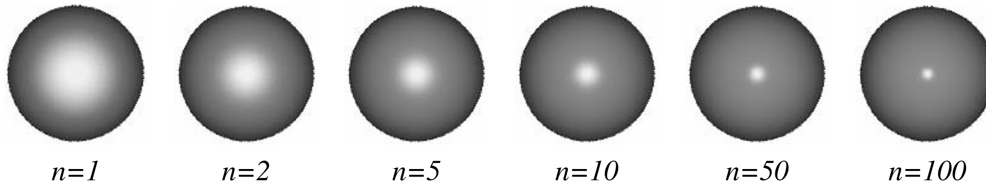
Jelöljük a normálvektor és a felezővektor közötti szöget δ-val! Ekkor a spekulárisan visszavert fény a Blinn által javasolt változatban:

$$L_\lambda = L_\lambda^{in} \cdot k_{s,\lambda} \cdot \cos^n \delta,$$

illetve csak vörös–zöld–kék hullámhosszakon számított képek esetén:

$$L_R = L_R^{in} \cdot k_{s,R} \cdot \cos^n \delta, \quad L_G = L_G^{in} \cdot k_{s,G} \cdot \cos^n \delta, \quad L_B = L_B^{in} \cdot k_{s,B} \cdot \cos^n \delta.$$

Megjegyezzük, hogy a Phong és a Phong – Blinn modell k_s és n tényezői nem ugyanazok, ha a két modellt azonos tényezőkkel használjuk, akkor nem kapunk azonos eredményt. Abban viszont hasonlóak, hogy amint az n-t növeljük, a felület mindkét modellben egyre „polírozottabbá” válik.



n=1 n=2 n=5 n=10 n=50 n=100

4.7. ábra. *Diffúz-spekuláris gömbök különböző n fényességértékekkel*

Összefoglalva a spekuláris felületek Phong – Blinn BRDF modellje:

$$f_{r,\lambda}(\vec{L}, \vec{V}) = k_{s,\lambda} \cdot \frac{\cos^n \delta}{\cos \theta'}.$$

4.8.6. Cook – Torrance modell

A *Cook – Torrance* BRDF [31] a spekuláris visszaverődés fizikai alapú modellje, amely a felületet véletlen orientációjú, azonos S területű, ideális tükör jellegű mikrofelületek halmazának tekinti. A feltételezés szerint a mikrofelületek egyszeres visszaverődése a spekuláris taghoz járul hozzá. A többszörös visszaverődés, illetve a fotonok elnyelése és későbbi emissziója viszont a diffúz tagot erősíti. A Cook – Torrance BRDF alakja a következő:

$$f_{r,\lambda}(\vec{L}, \vec{V}) = \frac{P_{\vec{H}}(\vec{H})}{4(\vec{N} \cdot \vec{L})(\vec{N} \cdot \vec{V})} \cdot G(\vec{N}, \vec{L}, \vec{V}) \cdot F(\lambda, ang(\vec{H}, \vec{L})),$$

ahol $P_{\vec{H}}(\vec{H})$ annak a valószínűség-sűrűsége, hogy a mikrofelület normálisa a $\vec{H}$ felezővektor irányába esik, a

$$G(\vec{N}, \vec{L}, \vec{V}) = \min\{2 \cdot \frac{(\vec{N} \cdot \vec{H}) \cdot (\vec{N} \cdot \vec{V})}{(\vec{V} \cdot \vec{H})}, 2 \cdot \frac{(\vec{N} \cdot \vec{H}) \cdot (\vec{N} \cdot \vec{L})}{(\vec{L} \cdot \vec{H})}, 1\}$$

geometria faktor pedig annak a valószínűségét fejezi ki, hogy a mikrofelületet a foton akadálytalanul megközelíti, és a visszaverődés után nem találkozik újabb mikrofelülettel, végül az $F(\lambda, ang(\vec{H}, \vec{L}))$ Fresnel-együttható annak a valószínűsége, hogy a foton az eltalált, ideális tükörnek tekintett mikrofelületről visszaverődik.

A $P_{\vec{H}}(\vec{H})$ mikrofelület orientációs sűrűségfüggvényt több különböző megközelítéssel definiálhatjuk. Az elektromágneses hullámok szóródását leíró elmélet szerint a *Beckmann-eloszlás* [18] használandó:

$$P_{\vec{H}}(\vec{H}) = \frac{1}{m^2 \cos^4 \delta} \cdot e^{-\left(\frac{\tan^2 \delta}{m^2}\right)}.$$

Sajnos ez az eloszlás nem alkalmas fontosság szerinti mintavételre (lásd a 8.9.1. fejezetet). Ezt a hiányosságot küszöböli ki az egyszerűbb, de fizikailag kevésbé megalapozott Ward-féle változat:

$$P_{\vec{H}}(\vec{H}) = \frac{1}{m^2 \pi \cos^3 \delta} \cdot e^{-\left(\frac{\tan^2 \delta}{m^2}\right)}.$$

4.8.7. Összetett anyagmodellek

A valódi anyagok általában nem sorolhatók be egyértelműen az eddigi osztályokba, hanem egyszerre több visszaverődési modell tulajdonságait is hordozzák. Például egy szépen lakkozott asztal a fény egy részét a felületéről ideális tükörként veri vissza. A fény másik része viszont behatol a lakkrétegbe és azon belül spekuláris jelleggel változtatja meg az irányát, végül lesznek olyan fotonok is, amelyek egészen a fáig jutnak, amelynek felületén diffúz módon változtatnak irányt. A lakk a beeső fény színét nem módosítja, viszont a fa a fehér fényből csak a „barna” részt veri vissza. Az ilyen anyagokat az eddigi visszaverődési modellek összegével jellemezhetjük:

$$\begin{aligned}
L_R &= L_R^{in} \cdot k_{d,R} \cdot \cos\theta' + L_R^{in} \cdot k_{s,R} \cdot \cos^n \psi + L_R^{in} \cdot k_{r,R}, \\
L_G &= L_G^{in} \cdot k_{d,G} \cdot \cos\theta' + L_G^{in} \cdot k_{s,G} \cdot \cos^n \psi + L_G^{in} \cdot k_{r,G}, \\
L_B &= L_B^{in} \cdot k_{d,B} \cdot \cos\theta' + L_B^{in} \cdot k_{s,B} \cdot \cos^n \psi + L_B^{in} \cdot k_{r,B}.
\end{aligned}$$

Természetesen az eddig bemutatott anyagmodelleken felül még számos létezik. A háromdimenziós grafikában alkalmazott anyagmodellek egyik legrészletesebb ismertetését a Siggraph konferencia kurzus anyagai között találjuk [12]. Az egyes modellek tulajdonságainak, paramétereinek megismerésére a legjobb módszer egy közös programcsomagban való implementálásuk lenne [54, 109].

A következő oldalon látható `Material` osztály a diffúz-spekuláris anyagmodellek egy lehetséges implementációját mutatja be. Mivel egy diffúz felület akkor veri vissza a teljes fényenergiát, ha a k_d tényező értéke $1/\pi$, illetve spekuláris felület esetén akkor, ha a $k_s = (n+2)/2\pi$, ezért a `SetDiffuseColor` és a `SetSpecularColor` metódusok a

visszaverődési tényezőket úgy számítják ki, hogy a k_d illetve a k_s maximális értékét a paraméterként kapott értékkel súlyozzák.

```
//=============================================================
class Material {
//=============================================================
public:
    Color kd;    // diffúz visszaverődési tényező
    Color ks;    // spekuláris visszaverődési tényező
    float n;     // fényesség

    Material();
    void  SetDiffuseColor(Color& Kd)  { kd = Kd / M_PI; }
    void  SetSpecularColor(Color& Ks) { ks = Ks * (n + 2) / M_PI / 2.0; }
    Color Brdf(Vector& inDir, Vector& norm, Vector& outDir);
};

//-------------------------------------------------------------
Color Material::Brdf(Vector& inDir, Vector& norm, Vector& outDir) {
//-------------------------------------------------------------
    double cosIn = -1.0 * (inDir * norm);
    if (cosIn <= EPSILON) return Color(); // ha az anyag belsejéből jövünk
    Color  retColor = kd;         // diffúz BRDF
    Vector reflDir = norm * (2.0 * cosIn) + inDir; // tükörirány
    double cosReflOut = reflDir * outDir;          // tükörirány-nézeti szöge
    if (cosReflOut > EPSILON)     // spekuláris BRDF
        retColor += ks * pow(cosReflOut, n) / cosIn;
    return retColor;
}
```

4.8.8. Az árnyalási egyenlet egyszerűsített változata

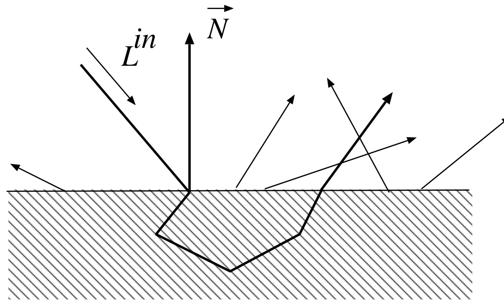
Eddig feltételeztük, hogy a felületi pontot csak egy beérkező fénysugár világítja meg. A valóság szimulációjához azonban a fényforrásokból közvetlenül, és a visszaverődések miatt közvetetten sugárzott teljes fénymennyiséget figyelembe kellene venni. A közvetlen (direkt) és a közvetett (indirekt) megvilágítás hatását az *árnyalási egyenlet* (*rendering equation*) írja le. Mivel az árnyalási egyenlettel a 8. fejezetben részletesen fogunk foglalkozni, ezért itt csak az egyszerűsített változatát adjuk meg, amely a fényforrások felületi pontban jelentkező direkt megvilágítását számítja ki, az indirekt megvilágításból pedig csak a tükör és a törési irányból érkező fénysugarakat veszi figyelembe:

$$L = L^e + k_a \cdot L^a + \sum_l \left[k_d \cdot \cos\theta'_l \cdot L_l^{\text{in}} + k_s \cdot \cos^n \psi_l \cdot L_l^{\text{in}} \right] + k_r \cdot L_r^{\text{in}} + k_t \cdot L_t^{\text{in}},$$

ahol L^e a felületi pont által kibocsátott intenzitás, $k_a \cdot L^a$ pedig az *ambiens tag*, amely a többszörös visszaverődések elhanyagolásának kompenzálására szolgál. A képlet harmadik tagja az absztrakt fényforrásokból érkezett, majd a felület által a kamera irányába vert fényerősséget határozza meg. Az árnyalási egyenlet negyedik tagja a tükörirányból érkező L_r^{in} intenzitás hatását adja meg, míg a $k_t \cdot L_t^{\text{in}}$ az ideális törésre vonatkozik.

Az egyszerűsített árnyalási egyenletet használó módszereket *lokális illuminációs algoritmus*oknak, a többszörös visszaverődéseket nem elhanyagolókat pedig *globális illuminációs algoritmus*oknak hívjuk.

4.8.9. Anyagon belüli szóródás



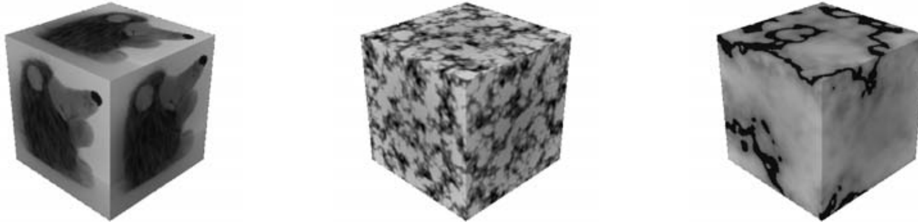
4.8. ábra. *Anyagon belüli szóródás*

A fémek kivételével minden anyag egy bizonyos szintig áttetsző, azaz a fény a felület belsejébe be tud jutni. Magán az anyagon azonban a fény csak egy kis része jut át, nagyobb része elenyészik, vagy a belépés oldalán lép ki. Ezt a jelenséget *anyagon belüli szóródás*nak (*subsurface scattering*) nevezzük (4.8. ábra). Az anyagon belüli szóródást a széleskörben elterjedt BRDF modellekkel nem lehet szimulálni, ám a legtöbb anyagnál ez nem is lényeges. Viszont a márvány, a gránit vagy az emberi bőr valószerű megjelenítésénél nem tekinthetünk el tőle (**??**. ábra). A teljes szimulációhoz a *kétirányú szóró felületi visszaverődési eloszlásfüggvény* vagy röviden *BSSRDF* (*Bi-directional Scattering Surface Reflectance Distribution Function*) alkalmazása szükséges. Sajnos az anyagon belüli szóródás szimulációja még közelítések alkalmazása mellett is rengeteg számítást igényel.

4.9. Textúrák

A fürdőszobában felrakott csempe, az üvegpohár, vagy a tűzhely anyagjellemzőit az eddig ismertetett módszerekkel könnyen megadhatjuk. Ám elég csak felidézni egy perzsaszőnyeg bonyolult mintázatát és máris gondban vagyunk. Mivel ezek a tárgyak jóval összetettebb, változatosabb anyagtulajdonságokkal rendelkeznek, felületük számos pontján kellene a BRDF modelleket különböző paraméterekkel használnunk. Ez egyrészt a modellezési folyamatot rettentően meghosszabbítaná, másrészt a képszintézist is jelentősen lelassítaná.

A problémát a textúrák segítségével oldhatjuk meg. A *textúra* fogalom először csak egy olyan kétdimenziós képet jelentett, amelyet egy felülethez lehetett rendelni, a benne szereplő adatok pedig a felület színét írták le. Tehát a perzsaszőnyeget egy téglalapra ráfeszített képként kell elképzelni. Mivel ezek a textúrák valamilyen képi információt tárolnak, *bittérképes textúrák*nak is nevezzük őket.

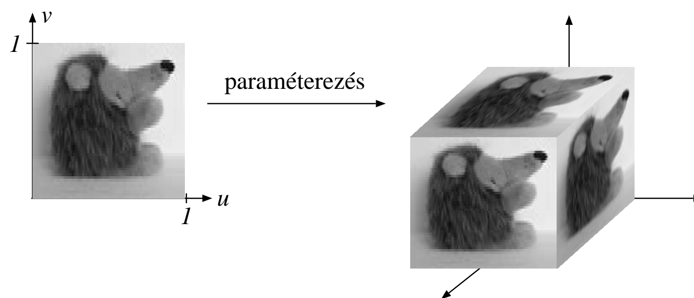


4.9. ábra. *Bittérképes, procedurális és 3D textúrák*

Később megjelentek a *procedurális textúrák* és a *3D textúrák* (4.9. ábra), illetve a már jólismert 2D textúra felhasználási területét is jelentősen kibővítették. A procedurális és a 3D textúrákkal a továbbiakban nem foglalkozunk, ám ha a kedves Olvasó többet szeretne megtudni róluk, akkor Alan Watt könyveit javasoljuk [137, 138]. A bittérképes textúrák lehetséges alkalmazási területeit a 7. fejezetben részletesen tárgyaljuk.

4.9.1. Paraméterezés

A bittérképes textúra egy kép, a *paraméterezés* során pedig azt a leképzést adjuk meg, amely a 2D textúra értelmezési tartományát, azaz az $(u,v) \in [0,1]^2$ egységnégyzet pontjait hozzárendeli a háromdimenziós tárgy (x,y,z) felületi pontjaihoz.



4.10. ábra. *Paraméterezés*

A továbbiakban a legjellemzőbb felületek paraméterezésével foglalkozunk. A képszintézis módszerek ismertetésénél látni fogjuk, hogy a valószerű képek előállításakor legtöbbször nem is erre a leképzésre van szükségünk, hanem ennek az inverzére. Tehát gyakran szükséges az a leképzés is, amely az (x,y,z) felületi ponthoz hozzárendel egy (u,v) egységnégyzetbeli pontot. Ezért minden paraméterezésnél megadjuk az inverz leképzést is.

Gömbfelületek paraméterezése

Az origó középpontú, r sugarú gömbfelület egy lehetséges paraméterezését úgy kapjuk meg, hogy a felület pontjait gömbi koordinátarendszerben (3.1.3. fejezet) fejezzük ki:

$$x(\theta,\phi) = r \cdot \sin\theta \cdot \cos\phi, \quad y(\theta,\phi) = r \cdot \sin\theta \cdot \sin\phi, \quad z(\theta,\phi) = r \cdot \cos\theta,$$

ahol a θ a $[0,\pi]$, a ϕ pedig a $[0,2\pi]$ tartományból kerülhet ki.

4.11. ábra. *Gömbi és cilindrikus leképzés*

Azonban nekünk a háromdimenziós test (x,y,z) pontját nem θ-val és ϕ-vel kell paraméterezni, hanem az egységintervallumba eső u-val és v-vel. Ezért a textúra koordinátákat kifejezzük a gömbi koordinátákkal:

$$u = \frac{\phi}{2\pi}, \qquad v = \frac{\theta}{\pi}.$$

Tehát egy gömbfelület paraméterezése:

$$x(u,v) = r \cdot \sin v\pi \cdot \cos 2\pi u, \quad y(u,v) = r \cdot \sin v\pi \cdot \sin 2\pi u, \quad z(u,v) = r \cdot \cos v\pi.$$

Egy gömbfelület paraméterezésének inverz leképzése:

$$u = \frac{1}{2\pi} \cdot (\mathrm{atan2}(y,x) + \pi), \quad v = \frac{1}{\pi} \cdot \arccos\left(\frac{z}{r}\right),$$

ahol az atan2(y,x) azt a C könyvtári függvényt jelenti, amely egy tetszőleges (y,x) koordinátapárhoz hozzárendeli a polárszöget a $[-\pi,\pi]$ tartományban.

Hengerfelületek paraméterezése

A H magasságú, r sugarú z-tengely körüli forgásfelület alsó alapkörének középpontja legyen az origó (4.11. ábra) ! Az így kialakuló hengerfelület implicit egyenlete:

$$x^2 + y^2 = r^2, \quad 0 \le z \le H.$$

Ezen hengerfelület egy lehetséges paraméterezését úgy kapjuk meg, hogy a felület pontjait cilindrikus koordinátarendszerben fejezzük ki:

$$x(\theta, h) = r \cdot \cos\theta, \quad y(\theta, h) = r \cdot \sin\theta, \quad z(\theta, h) = h,$$

ahol a θ a $[0, 2\pi]$, a h pedig a $[0, H]$ tartományból kerülhet ki.

Természetesen a háromdimenziós test (x, y, z) pontját itt sem a θ-val és a h-val kell paraméterezni, hanem u-val és v-vel. Ezért a textúra koordinátákat kifejezzük a cilindrikus koordinátákkal:

$$u = \frac{\theta}{2\pi}, \quad v = \frac{h}{H}.$$

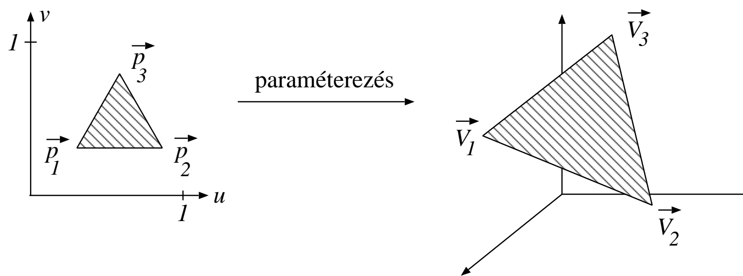
Tehát egy hengerfelület paraméterezése:

$$x(u, v) = r \cdot \cos(2\pi u), \quad y(u, v) = r \cdot \sin(2\pi u), \quad z(u, v) = v \cdot H.$$

Egy hengerfelület paraméterezésének inverz leképzése:

$$u = \frac{1}{2\pi} \cdot (\text{atan2}(y, x) + \pi), \quad v = \frac{z}{H}.$$

Háromszögek paraméterezése



4.12. ábra. *Háromszögek paraméterezése*

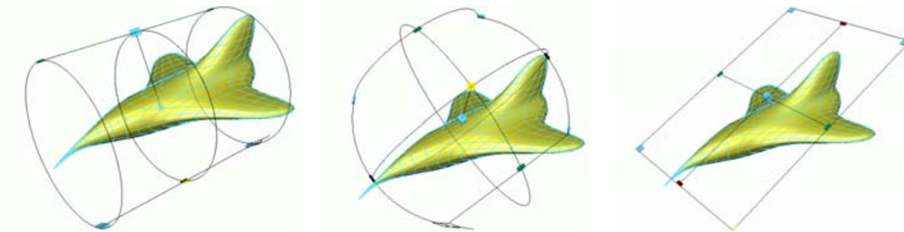
Ebben az esetben a paraméterezés egy, a textúratérben adott 2D háromszöget képez le egy előre megadott térbeli háromszögre. A leképzés megadására lineáris függvényt

alkalmazunk, amely a linearitása miatt nemcsak a csúcspontokat, hanem a teljes háromszöget megőrzi:

$$x = A_x \cdot u + B_x \cdot v + C_x, \quad y = A_y \cdot u + B_y \cdot v + C_y, \quad z = A_z \cdot u + B_z \cdot v + C_z. \qquad (4.3)$$

Ha a 4.3. képletbe behelyettesítjük a háromszög $\vec{V_1} = (x_1, y_1, z_1)$, $\vec{V_2} = (x_2, y_2, z_2)$ és $\vec{V_3} = (x_3, y_3, z_3)$ pontjait, illetve a textúratérbeli háromszög $\vec{p_1} = (u_1, v_1)$, $\vec{p_2} = (u_2, v_2)$ és $\vec{p_3} = (u_3, v_3)$ csúcsait, akkor egy 9 egyenletből álló, 9 ismeretlenes lineáris egyenletrendszerhez jutunk. Ezt megoldva az ismeretlen A_x, B_x, C_x, A_y, B_y, C_y, A_z, B_z, C_z értékek, és ezáltal a leképzés is meghatározható.

4.9.2. Közvetítő felületek használata



4.13. ábra. *Közvetítő felületek: henger, gömb, téglalap*

A virtuális világunkban elég ritkán szerepelnek gömbök és hengerek, egy bonyolultabb test pedig túl sok háromszögből épül fel, ezért nagyon ritka, hogy valaki minden térbeli háromszöghöz egyesével rendeli hozzá a textúratér egy-egy háromszögét. Ezért a paraméterezésnél gyakran egy közvetítő tárgy felületét is használjuk a következő módon:

1. A textúrázni kívánt objektumhoz hozzárendelünk valamilyen egyszerű geometriájú közvetítő alakzatot (4.13. ábra),
2. a közvetítő felület (x', y', z') pontjait a textúratér (u, v) koordinátáival paraméterezzük (*S-leképzés*),
3. az (x', y', z') hármashoz hozzárendeljük a textúrázni kívánt objekum (x, y, z) pontját (*O-leképzés*).

Az O-leképzés a textúrázni kívánt felületnek a közvetítő felületre történő *vetítés*ét jelenti. A vetítősugarak a közvetítő felületre mindig merőlegesek. Az (x', y', z') vetületet

az (x, y, z)-n átmenő vetítősugár és a közvetítő felület metszéspontjaként határozhatjuk meg.

Ha a közvetítő felület henger, a vetítősugarak a hengerpalástra merőlegesek és a henger középvonalában találkoznak. Ha a közvetítő felület gömb, akkor a vetítősugarak a gömb középpontjában futnak össze. Ha azonban a közvetítő felület sík, akkor viszont párhuzamos vetítés történik.

5. fejezet

Virtuális világ

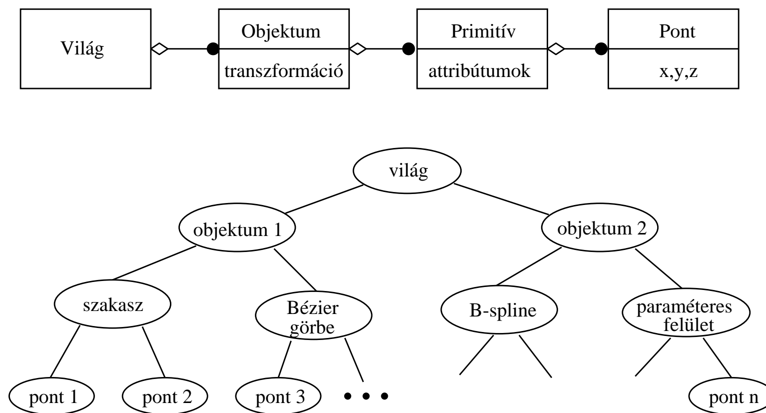
A modellezés során a számítógépbe bevitt információt a program a memóriában adatszerkezetekben, illetve a merevlemezen fájlokban tárolja. Az adatszerkezetek és a fájl többféleképpen is kialakítható. A modellezési folyamathoz használt optimális adatstruktúra nem feltétlenül hatékony a képszintézishez, és ez fordítva is igaz. A különböző adatszerkezetek közti választás ezért mindig az adott feladat függvényében történik.

A színtérben szereplő objektumok (alakzatok, fényforrások, kamera) a *világ-koordinátarendszer*ben találkoznak. Az alakzatok geometriáját azonban nem mindig célszerű közvetlenül ebben a térben definiálni. Sokkal egyszerűbb az a megközelítés, amikor az objektumokat a saját lokális koordinátarendszerükben (*modellezési-koordinátarendszer*) készítjük el[1], majd ehhez egy *modellezési transzformáció*t is megadunk, amely az objektumot a modellezési-koordinátarendszerből a világ-koordinátarendszerbe transzformálja. Ennek a megközelítésnek nagy hasznát vesszük animáció esetén, hiszen a tárgyak mozgatásakor — a geometriát érintetlenül hagyva — csak a modellezési transzformációt kell változtatnunk.

5.1. Hierarchikus adatszerkezet

A modell tárolásához legkézenfekvőbb a virtuális világ hierarchikus szerkezetéből kiindulni. A világ objektumokat tartalmaz, az objektumok pedig *primitív* objektumokat. Geometriai primitív például a gömb és a gúla, valamint a téglatest vagy poliéder, amely lapokból (*face*), azaz poligonokból áll. A poligont élek építik fel, az élek pedig térbeli pontokat kapcsolnak össze. A hierarchikus felépítésnek megfelelő objektummodell az 5.1. ábrán látható.

[1]Vessük össze gondolatban egy téglatest definiálásának nehézségeit akkor, ha a téglatest a világ-koordinátarendszerben általános helyzetű, illetve akkor, ha a saját modellezési-koordinátarendszerében az egyik sarka az origó, és az oldalai párhuzamosak a koordinátatengelyekkel!



5.1. ábra. *A világleírás osztály és objektum diagramja*

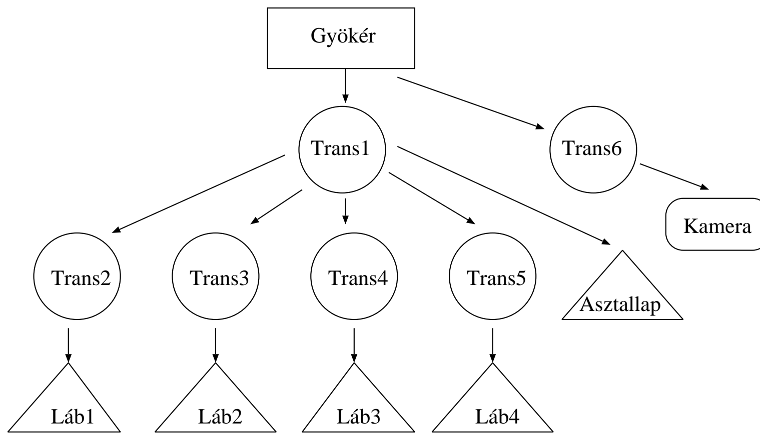
Egy *objektum* szokásos *attribútum*ai: az objektum neve, a modellezési transzformációja, a képszintézis gyorsítását szolgáló *befoglaló doboz* stb. A *primitív*eknek többféle típusa lehetséges, úgy mint szakasz, görbe, felület, poligon stb. A primitívek attribútumai a primitív típusától függnek. Gyakran előfordul, hogy egy objektum más objektumokat is magában foglal. A tartalmazott objektumok a tartalmazóhoz képes mozoghatnak. Gondoljunk például egy autóra, amely a karosszériából és négy forgó kerékből áll! A karosszéria transzformációja (haladás) a kerekekre is vonatkozik. Az emberi test is bonyolult hierarchikus rendszer (9. fejezet).

5.1.1. A színtérgráf

A *színtérgráf* egy olyan adatszerkezet, amely a színtér különböző jellemzőit és az elemek alá- és fölérendeltségi viszonyait tartalmazza. Az adatstruktúra tulajdonképpen egy irányított körmentes gráf, ahol a csomópontok a következők lehetnek: geometria, anyagjellemzők, fényviszonyok, kamera, transzformációk. Egy színtérgráf implementáció lehet egy fájl formátum (VRML), egy programozási API (Java3D), vagy mindkettő egyszerre (OpenInventor).

Egy egyszerű színtérgráf látható az 5.2. ábrán, amely egy asztalt és egy kamerát tartalmaz. Az asztal elhelyezkedését a világban a `Trans1` transzformáció adja meg. Az asztal négy lába négy különböző helyen szerepel a gráfban. Ezeket az asztalhoz képest a `Trans2`, `Trans3`, `Trans4`, `Trans5` transzformációk adják meg. Az asztalláb helyzetét a virtuális világban tehát a csomópontból kiindulva, a gráf csúcspontjáig meglátogatott transzformációk szorzata határozza meg. Egy adott transzformáció alá korlátlan számú objektum szúrható be. A kamera helyét a `Trans6` transzformáció definiálja.

A színtérgráf nemcsak a geometriát tartalmazza, hanem minden olyan attribútu-



5.2. ábra. *Színtérgráf*

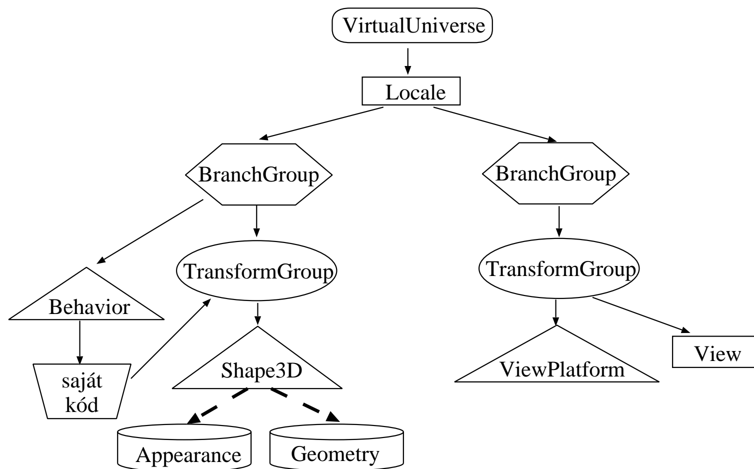
mot, amelyre a modellezés vagy a megjelenítés során szükség lehet. Egy objektumhoz anyagjellemzők (szín, textúra) és viselkedési minták is tartozhatnak. Egy viselkedési minta előírhatja például azt, hogy egy ajtó felé közeledve az ajtó kinyílik, vagy hogy Bodri harcikutya fogait csattogtatva járőrözik a ház körül. A színtérgráfokban általában absztrakt fényforrásokat (pontszerű, szpot, irány stb.) is elhelyezhetünk. Ilyen színtérgráf megvalósítások az OpenInventor, a VRML és a Java3D környezetek. Ilyent használnak a Maya és Houdini alkalmazások is. Az egyik legfiatalabb és legrobusztusabb közülük a Java3D, ezért ezt mutatjuk be először.

5.1.2. A Java3D színtérgráf

A *Java3D*-t a Java programozási nyelv [40] háromdimenziós kiterjesztéseként vezették be. A Java3D valójában egy Java osztálykönyvtár (API), a színtérgráf felépítése Java osztályok példányosításával és Java metódusok meghívásával történik.

Egy egyszerűsített Java3D színtérgráf séma látható az 5.3. ábrán. Egy virtuális univerzum (`VirtualUniverse`) egy vagy több (általában csak egy) `Locale`-t tartalmazhat. A `Locale` egy saját középponttal (origó) és koordinátarendszerrel rendelkező galaxist szimbolizál az univerzumban. A színtérgráf a galaxisban két fő ágra bomlik: az egyik ág tartalmazza a testek és a fényforrások leírását, a másik ág pedig a kamera paramétereit.

A `Group` csomópont egy tároló (konténer), amelynek tetszőleges számú gyermeke lehet. A `Group`-ból származik a `BranchGroup` és a `TransformGroup`. A `BranchGroup` az elágazásokért felelős. `Locale` alá csak `BranchGroup`-ot lehet beszúrni.



5.3. ábra. *Java3D színtérgráf sablon*

A `TransformGroup` csomópont egy olyan transzformációt definiál, amelyet a csomóponthoz tartozó részgráf összes objektumára végre kell hajtani. Több transzformáció egymásba ágyazása esetén az a megállapodás, hogy a mélyebben levő transzformációkat hajtjuk végre először, majd innen a csúcs felé haladva látogatjuk meg a transzformációs csomópontokat.

A `Shape` csomópont egy színtérbeli elemnek a geometriai (`Geometry`) és a megjelenítési (`Appearance`) jellemzőit definiálja. Geometriai adatok a háromdimenziós koordináták, a normálvektorok, a textúra koordináták stb. A geometria leírható pontokkal, szakaszokkal, négyszöglapokkal, háromszöglapokkal, *háromszög szalag*okkal (`TriangleStrips`) vagy *háromszög legyező*kkel (`TriangleFan`) (3.4.1. fejezet). A színtérgráf megadja az objektumok dinamikus viselkedését is. Erre a `Behavior` csomópont alkalmas, amelyhez a programozó a viselkedést megvalósító rutinokat írhat. Ezek a rutinok megváltoztathatják magát a színtérgráfot is. Egy ilyen viselkedés lehet egy transzformációs mátrix periodikus változtatása (például egy kocka egyik tengelye körüli forgatása).

A színtérgráf másik ága a képszintézishez szükséges kamerát adja meg. Az itt található `TransformGroup` az *avatár*[2] pozícióját, *nézeti irány*át stb. határozza meg, a `ViewPlatform` pedig egy gömb alakú tartományt ír le, amelyen belül az avatár és a színtér objektumai közötti interakció lehetséges. Például egy hangforrás csak akkor hallható az avatár számára, ha a `Sound` csomópont hatástartománya — amely szintén egy gömb — metszi az avatár tartományát. Hasonlóan, az objektumok csak akkor léphetnek kapcsolatba az avatárral, ha az objektum az avatár tartományán belül tartózkodik.

[2]a virtuális világban a felhasználót képviselő objektum

A `View` objektum tartalmazza a képszintézishez szükséges egyéb információkat: például a csipkézettség csökkentés (anti-aliasing) módját [118], a vágósíkokat, a sztereó vagy monó beállítást stb. Egy `Locale`-ban egyszerre több — különböző transzformációjú — `ViewPlatform` és `View` is definiálható, és így egyszerre több képernyőre is kerülhet különböző beállításokból készített kép.

A színtérgráfot a Java3D-ben metódushívásokkal, alulról felfelé építjük fel. Ebben a könyvben ugyan nem célunk a Java programozási nyelv bemutatása, azonban a nyelvet ismerők kedvéért egy kis ízelítőt adunk az ilyen programokból. A Java3D program vázát a következő utasítássorozat alkotja:

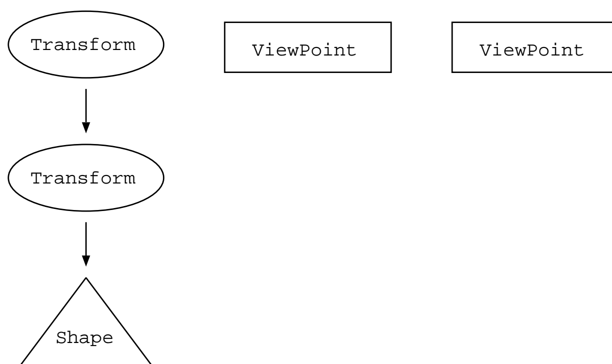
```
//=================================================================
public class HelloUniverse extends Applet {
//=================================================================
    universe = new VirtualUniverse();   // univerzum
    locale = new Locale(universe);      // ez egy világ koordinátarendszer

    // 1. készítsük el a kamerát definiáló részgráfot
    // készítsük el a kamera transzformációt
    Transform3D transform = new Transform3D();
    transform.set(new Vector3f(0.0, 0.0, 2.0)); // ez egy eltolás
    TransformGroup viewTransformGroup = new TransformGroup(transform);

    // állítsuk össze a kamera részgráfot a viewTransformGroup gyermekeként
    ViewPlatform viewPlatform = new ViewPlatform();
    viewTransformGroup.addChild(viewPlatform);

    Canvas3D canvas; // erre a vászonra rajzolunk
    View view = new View();
    view.addCanvas3D(canvas);
    view.attachViewPlatform(viewPlatform);

    BranchGroup viewBranch = new BranchGroup();
    viewBranch.addChild(viewTransformGroup);

    // a kamera ág hozzáadásával a színtérgráf "élővé válik"
    locale.addBranchGraph(viewBranch);

    // 2. készítsük el a modellt definiáló részgráfot
    BranchGroup objBranch = new BranchGroup();  // elágazás csomópont

    TransformGroup objTransform = new TransformGroup(); // transzformáció
    objTransform.addChild(new ColorCube().getShape());  // Shape3D hozzáadás
    objBranch.addChild(objTransform);

    locale.addBranchGraph(objBranch);
}
```

5.1.3. A VRML színtérgráf

A *VRML* (*Virtual Reality Modeling Language*) [134] egy szöveges[3] fájlformátum. Létrehozásának célja az volt, hogy egy kereskedelmi termékektől, vállalatoktól független szabvány szülessen, amely elősegítheti a világhálón a hagyományos tartalom (*HTML*) mellett a háromdimenziós információ terjedését. Létezik a VRML-nek egy 1.0-s verziója is, amely nem kompatibilis a VRML 2.0-val. A VRML újabb, 2.0 verzióját az (ISO/IEC 14772-1:1997) szabvány elfogadási évére utalva szokás még VRML97-nek is nevezni. A továbbiakban VRML alatt mindig a 2.0 verziót értjük.

A VRML számos jellemzőjét az Open Inventor .iv fájlformátumából örökölte. A VRML tapasztalatait pedig az előző fejezetben bemutatott Java3D színtérgráf kialakításakor használták fel. Azidőtájt ugyanis a VRML már egy sikeres és elfogadott szabvánnyá vált. Érdekességképpen megemlítjük, hogy a Web3D konzorcium közreműködésével 2003-ban elkészült a VRML következő generációja, amelyet az XML-lel való kapcsolat miatt X3D-nek neveztek el.

A VRML ismertetésére álljon itt egy egyszerű színtér. Az 5.4. ábrán a VRML fájl szerkezetét látjuk. Az ábrára tekintve a legszembetűnőbb különbség a Java3D-hez képest (5.3. ábra) a színtérgráf gyökerének (`VirtualUniverse`) hiánya.



5.4. ábra. *VRML színtérgráf*

A továbbiakban egy kockát tartalmazó színteret írunk le. A Java3D-hez képest különbség, hogy míg a Java3D a színtérgráf csomópontjait mellérendelő viszonyban, egyesével adta meg, és ezek a csomópontok mutatók segítségével hivatkoztak egymásra, addig a VRML a csomópontokat egymásba ágyazza. A különbség abból adódik, hogy míg az előbbi egy programozási nyelv, addig az utóbbi egy adat leíró nyelv.

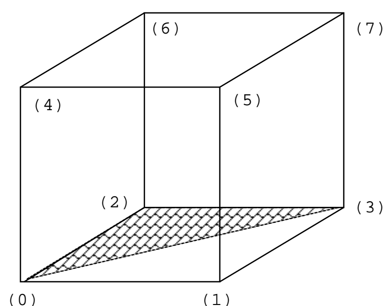
[3] a hálózati letöltések felgyorsításához ezt a szöveges fájlt bináris formába (.wrz) szokták tömöríteni

```
#VRML V2.0 utf8

DEF Box01 Transform {
  translation 6 0 -4
  children [
    Transform {
        translation 0 8.959 0
        children [
            Shape {
                appearance Appearance {
                    material Material { diffuseColor 0.89 0.6 0.72 }
                }
                geometry Box { size 24.04 17.92 39.49 }
            }
        ]
    }
  ]
}

DEF Camera01 Viewpoint {
  position -26.82 0 12.84       # pozíció
  orientation 1 0 0 -1.571      # orientáció
  fieldOfView 0.6024            # látószög
  description "Camera01"        # leírás
}
DEF Camera02 Viewpoint {
  position 77.3 0 -13.6
  orientation 1 0 0 -1.571
  fieldOfView 0.6024
  description "Camera02"
}
```

A szöveges formátumú VRML fájl kötelezően a `#VRML V2.0 utf8` megjegyzéssel kezdődik. Ezt egy eltolást ($(6,0,-4)$ vektorral) tartalmazó transzformációs csomópont követ. A `DEF` (*definition*) kulcsszóval ennek a transzformációnak (és a tartalmazott részgráfnak) a `Box01` nevet adtuk. A `USE` kulcsszóval lehetne a továbbiakban ezt a részgráfot a színtérgráf tetszőleges szintjére újra beszúrni. Nekünk azonban most elegendő egyetlen példány ebből a részgráfból. Egy `Transform` csomópontnak tetszőleges számú gyermeke (`children`) lehet, és a transzformációk tetszőleges mélységben egymásba ágyazhatók. A test (`Shape`) egy `appearance` és egy `geometry` mezőt tartalmaz. A `Viewpoint` kulcsszóval tetszőleges számú kamerát definiálhatunk, amelyeket pozícióval, orientációval és látószöggel adunk meg.

A kézigránátot felénk hajító terrorista, vagy a birodalmi lépegető elég nehezen írható le csak dobozok, gömbök és hengerek segítségével. Ezért szükség van egy olyan elemre, amellyel tetszőleges poliéder megadható. A leggyakrabban használt VRML csomópont az `IndexedFaceSet`, amelynek a `coord` adattagjában találhatók a geometriát leíró pontok Descartes-koordinátái. A `coordIndex` mező definiálja a poligonokat, azaz a topológiát. A `coordIndex` indexeket tartalmaz a `coord` mező pontjaira. A -1 index azt jelenti, hogy ott új poligon kezdődik. Egy kocka VRML leírása a következő:

```
Shape {
    appearance Appearance {
        material Material { diffuseColor 0.55 0.027 0.22 }
    }
    geometry DEF Box01-FACES IndexedFaceSet {
        ccw TRUE    # óramutató járásával ellentétes körüljárás
        solid TRUE  # tömör test
        coord DEF Box01-COORD Coordinate { point [
          -2 -2 2, 2 -2 2, -2 -2 -2, 2 -2 -2, -2 2 2, 2 2 2, -2 2 -2, 2 2 -2]
        }
        coordIndex [
          0, 2, 3, -1, 3, 1, 0, -1, 4, 5, 7, -1, 7, 6, 4, -1,
          0, 1, 5, -1, 5, 4, 0, -1, 1, 3, 7, -1, 7, 5, 1, -1,
          3, 2, 6, -1, 6, 7, 3, -1, 2, 0, 4, -1, 4, 6, 2, -1]
    }
}
```



5.5. ábra. *IndexedFaceSet kocka csúcsainak sorrendje és az első háromszöge*

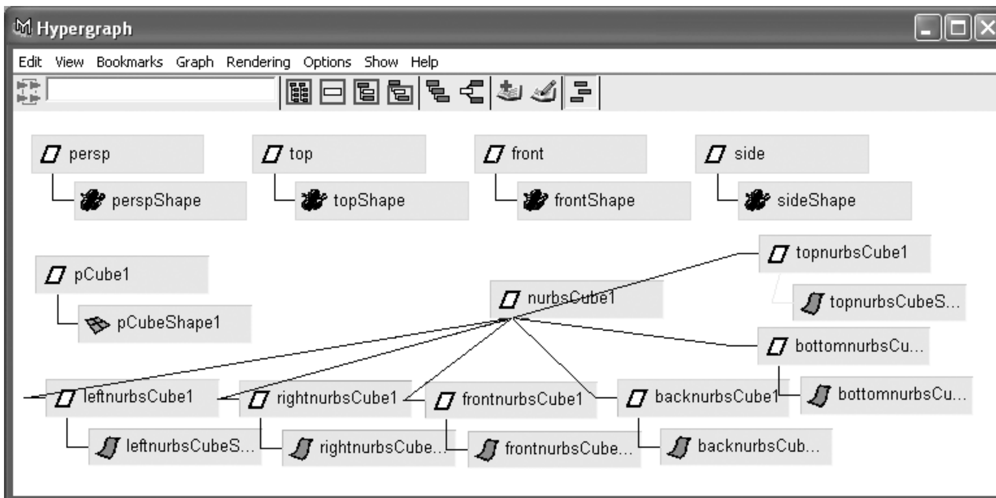
A csúcspontokat és az első háromszög helyét az 5.5. ábra szemlélteti. Hangsúlyozzuk, hogy a poligonok megadásakor fontos a csúcspontok sorrendje, ugyanis ez alapján számítjuk a lap *normálvektor*át. Megállapodás szerint a normálvektor irányából (azaz a testet kívülről) nézve a csúcsok sorrendje az óramutató járásával ellentétes körüljárást követ (a fenti VRML részletben ezt a `ccw TRUE` sorban állítottuk be).

A VRML fájlokra és azok beolvasására az 5.3.3. fejezetben még visszatérünk.

5.1.4. Maya hipergráf

A *Maya* [10] modellezőprogram *hipergráf*ja (*Hypergraph*) a színtér komponensei közötti kapcsolatokat mutatja. Kétféle hipergráf létezik: a *színtér hierarchia gráf* és a *függőségi gráf*.

A színtér hierarchia gráf (5.6. ábra) csomópontjai az objektumok, a fényforrások, a kamerák és az egyéb színtér építő elemek. A `Shape` típusú csomópontok (például `pCubeShape1`) tartalmazzák az objektumok geometriáját. A transzformációs csomópontok (`pCube1`) elhelyezik az objektumokat a térben. Az 5.6. ábrán lévő színtér két kockát

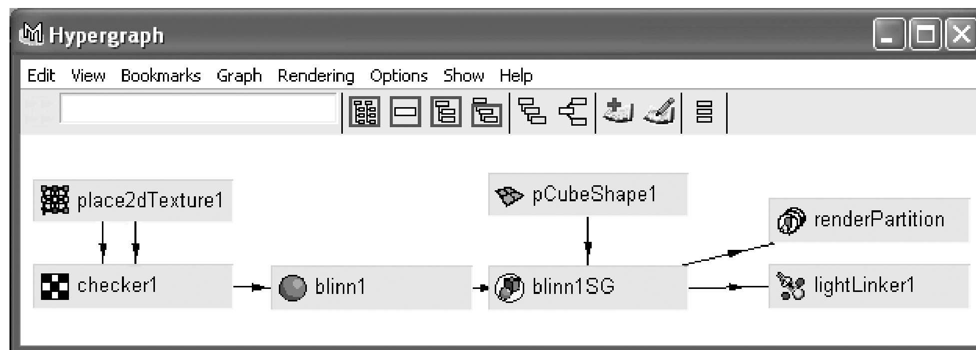


5.6. ábra. *Színtér hierarchia gráf Maya-ban*

tartalmaz. A `pCube1` egy poligon kocka, a `nurbsCube1` egy NURBS felületekből álló kocka transzformációs csomópontja. A NURBS kocka az éles élek miatt nem adható meg egyetlen NURBS felülettel, ezért a Maya a test 6 oldallapjához különálló felületeket rendel. Minden lap egy saját transzformációval rendelkező NURBS felület. Ezeket fogja össze a `nurbsCube1` transzformáció. A gráf ezenkívül tartalmazza azokat a kamerákat és transzformációjukat, amelyek a Maya felhasználói felületén az oldal-, elöl- és a felülnézetet, valamint a perspektív nézetet szolgáltatják.

A függőségi gráf (5.7. ábra) a Maya építő elemek közötti kapcsolatokat mutatja. Az építő elemek értékeket kapnak és értékeket szolgáltatnak más elemek számára. Az egész olyan, mint egy gép, mint egy automata, amelynek működése hozza létre a végeredményt, a képet vagy az animációt. Az adatáramlás irányát nyilak jelzik. Az ábrán például a `place2dTexture1` textúratranszformáció `outUV` mezője a `checker1` textúra `uvCoord` inputjához, a `checker1 outColor`-ja a `blinn1.ambientColor`-hoz van rendelve. A `blinn1SG` egy `ShadingGoup`, amely az adott `blinn1` anyaghoz tartozó objektumokat fogja össze.

Minden elem, amely a színtér hierarchia gráfban (5.6. ábra) megtalálható, szerepelhet a függőségi gráfban (5.7. ábra) is, azonban ez fordítva nem teljesül. A függőségi gráf mutatja például a képszintézis során felhasznált optikai elemeket (textúra, Phong BRDF stb.). Ezek az anyagjellemzők a Maya színtér hierarchia gráfjában nem jelennek meg.



5.7. ábra. *Függőségi gráf Maya-ban*

5.1.5. CSG-fa

A hierarchikus modell általánosításához juthatunk, ha a színtérgráf egy szintjén nem csupán az alatta lévő objektumok (mint például az 5.2. ábra asztallábai, asztallapja) egyesítését, hanem bármilyen halmazműveletet megengedünk. Mivel a halmazműveletek (unió, metszet, különbség) kétváltozósak, a keletkező modell egy bináris fa, amelynek levelei primitív testeket, a többi csomópontja pedig a gyermekobjektumokon végrehajtott halmazműveletet (3.44. ábra) képviselnek. Ezen modell különösen jól illeszkedik a konstruktív tömörtest geometriához, ezért az ilyen bináris fa szokásos elnevezése a *CSG-fa*, amelyet a modellezésről szóló 3.5.1. fejezetben már tárgyaltunk.

5.2. A geometriai primitívek

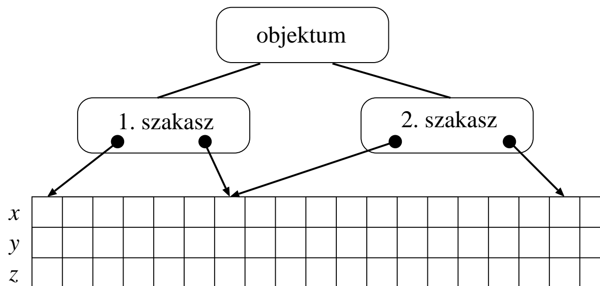
A geometriai alapelemekről részletesen a 3. fejezetben a modellezés témakörben olvashatunk. Ebben a fejezetben a geometriai primitívek adatszerkezeteivel és tárolási lehetőségeivel foglalkozunk.

5.2.1. A geometria és a topológia szétválasztása

Az 5.1. és az 5.2. ábra tisztán hierarchikus modelljével szemben több kifogás emelhető. A hierarchikus modell a különböző primitívek közös pontjait többszörösen tárolja, azaz nem használja ki, hogy a különböző primitívek általában illeszkednek egymáshoz, így a pontokat közösen birtokolják. Ez egyrészt helypazarló, másrészt a transzformációkat feleslegesen sokszor kell végrehajtani. Ráadásul, ha az interaktív modellezés során a felhasználó módosít egy pontot, akkor külön figyelmet kíván valamennyi másolat korrekt megváltoztatása. Ezt a problémát megoldhatjuk, ha a pontokat eltávolítjuk az objektumokból és egy közös tömbben fogjuk össze őket. A test leírásában csupán mu-

tatókat vagy indexeket (fájl adatszerkezetben a mutatók természetesen nem jöhetnek szóba) helyezünk el a pontok azonosítására (5.8. ábra).

A javított modellünk tehát két részből áll. A pontokat tartalmazó tömb lényegében a geometriát határozza meg. Az adatstruktúra többi része pedig a részleges topológiát írja le, azaz azt, hogy egy objektum mely primitívekből áll és a primitíveknek melyek a definíciós pontjai.



5.8. ábra. *A világleírás kiemelt geometriai információval*

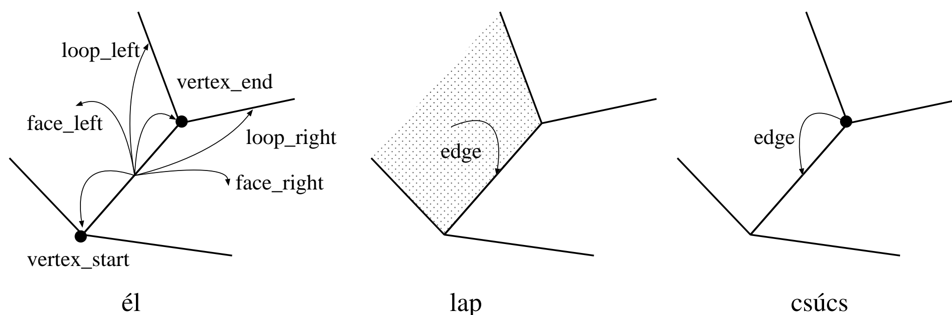
A hierarchikus modellel szemben a következő kifogásunk az lehet, hogy az adatstruktúrából nem olvasható ki közvetlenül a teljes *topológia*i információ. Például nem tudhatjuk meg, hogy egy pontra mely primitívek illeszkednek, illetve egy primitív mely objektumokban játszik szerepet. Ilyen topológiai információra azért lehet szükségünk, hogy eldöntsük, hogy a virtuális világ csak érvényes 2D illetve 3D objektumok gyűjteménye, vagy elfajult, háromnál alacsonyabb dimenziós „korcsok” is az objektumaink közé keveredtek. A beteg objektumok kiszűrése nem a képszintézis miatt fontos, hanem azért, mert a modell alapján geometriai műveleteket kívánunk végezni, esetleg szeretnénk térfogatot számítani, vagy egy NC szerszámgéppel legyártatni a tervezett objektumot. Elsőként a poligonokat tartalmazó adatszerkezeteket vizsgáljuk.

5.2.2. Poligonhálók

A teljes topológiai információ az illeszkedéseket kifejező mutatók beépítésével reprezentálható. Egy ilyen modell a 3D felületi modellek tárolására kifejlesztett *szárnyas él adatstruktúra* [17] (5.9. ábra), amelyben minden illeszkedési relációt mutatók fejeznek ki.

Az adatszerkezet központi eleme az él, amelyben mutatókkal hivatkozunk a két végpontra (`vertex_start`, `vertex_end`), az él jobb illetve bal oldalán lévő lapra (`face_left`, `face_right`), valamint ezen a két lapon a következő élre (`loop_left`, `loop_right`). Az éleket egy láncolt listában tartjuk nyilván, a `next` mutató a lista láncolásához kell és semmiféle topológiai jelentése sincsen.

```
//===============================================================
class Edge {
//===============================================================
   Vertex *vertex_start, *vertex_end; // kezdő és végpont
   Face   *face_left, *face_right;    // bal és jobb lap
   Edge   *loop_left, *loop_right;    // bal és jobb hurok
   Edge   *next;                      // láncoló mutató
public:
   Edge(Vertex* v1, Vertex* v2, Edge* np) {
      vertex_start = v1; vertex_end = v2; next = np;
      face_left = face_right = NULL;
      loop_left = loop_right = NULL;
      if (v1->edge == NULL) v1->edge = this;
      if (v2->edge == NULL) v2->edge = this;
   }
   void SetFace(Face* f, LineOrient o);
   bool HasFace(Face* f) { return (face_right == f || face_left == f); }
};
```



5.9. ábra. *Szárnyas él adatstruktúra*

Az élhez tartozó mutatók egy részét a konstruktorban töltjük fel. A másik részük pedig akkor kap tényleges jelentést, amikor a `SetFace` függvénnyel az élhez egy lapot rendelünk hozzá. Az élnek két „szárnya” van, amelyek közül az `orient` változóval választhatunk. A változó tartalmát úgy is értelmezhetjük, hogy az éleket irányítottnak tekintjük, és a jobb oldali lapra akkor mutatunk a jobb kezünkkel, ha az él irányába fordulunk, a bal oldali lapra pedig akkor, ha hátat fordítunk az él irányának.

```
//-----------------------------------------------------------------
void Edge::SetFace(Face* face, LineOrient orient) {
//-----------------------------------------------------------------
   switch (orient) {
   case FORWARD:  face_right = face;
                  if (face->edge->loop_right != NULL &&
                     face->edge->loop_right->HasFace(face)) {
                     loop_right = face->edge->loop_right;
                     face->edge->loop_right = this;
                  } else {
                     loop_right = face->edge->loop_left;
                     face->edge->loop_left = this;
                  }
                  face->edge = this;
                  return;
   case BACKWARD: face_left = face;
                  if (face->edge->loop_left != NULL &&
                     face->edge->loop_left->HasFace(face)) {
                     loop_left = face->edge->loop_left;
                     face->edge->loop_left = this;
                  } else {
                     loop_left = face->edge->loop_right;
                     face->edge->loop_right = this;
                  }
                  face->edge = this;
                  return;
   }
}
```

A csúcspontok tartalmazzák a Descartes-koordinátákat (`point`) és hivatkoznak az egyik illeszkedő élre, amelyből mutatókon keresztül már minden topológiai kapcsolat előállítható.

```
//===========================================================
struct Vertex {    // csúcspont
//===========================================================
   Vector     point;     // koordináták
   Edge*      edge;      // a csúcsot tartalmazó él
};
```

Hasonlóképpen a lapok is hivatkoznak egyik élükre, amelyből az összes határgörbe származtatható:

```
//===========================================================
struct Face {
//===========================================================
   Edge* edge;   // egy él
   Face* next;   // láncoló mutató
};
```

Ezek alapján egy poliéder geometriáját és topológiáját leíró adatszerkezet a következőképpen néz ki:

```
//=============================================
class Mesh {
//=============================================
protected:
   Vertex   *vertexarray;  // csúcsokat tartalmazó tömb
   int      nvertices, vertex_iterator;
   Edge     *edgelist, *edge_iterator, *edge_of_face_iterator;
   Face     *facelist, *face_iterator;
   int      nedges, nfaces;
public:
   Mesh( );
   Vertex*  AddVertex(Vector& point);
   Vertex*  GetVertex(int i) { return &vertexarray[i]; }
   Edge*    AddEdge(Vertex* v1, Vertex* v2);
   Face*    AddFace(Vertex* v1, Vertex* v2);
   void     LinkEdgeToFace(Face* face, Vertex* v1, Vertex* v2);

   Face*    GetNextFace();            // lapok egyenkénti visszaolvasása
   Edge*    GetNextEdge();            // élek egyenkénti visszaolvasása
   Vertex*  GetNextVertex();          // csúcsok egyenkénti visszaolvasása

   void     GetVerticesOfEdge(Edge* e, Vertex*& v1, Vertex*& v2); // él csúcsai
   void     GetFacesOfEdge(Edge* e, Face*& v1, Face*& v2);        // él lapjai
   Edge*    GetNextEdgeOfFace(Face* face, LineOrient& orient);    // lap élei
   Vertex*  GetNextVertexOfFace(Face* p);                         // lap csúcsai
};
```

Az élhez tartozó lapok és csúcsok az él struktúrából könnyen megkereshetők. A lap éleinek és csúcsainak megkereséséhez viszont már be kell járnunk az adatszerkezetet.

Egy lap éleinek előállításához először arra az élre lépünk, amelyre a lap hivatkozik, majd a lapok következő éleit azonosító mutatók mentén körbejárjuk a lapot. Az alábbi függvény újabb hívásakor mindig egy következő élt állít elő, és az `orient` változóban azt is megmondja, hogy a lapunk az él melyik oldalán található:

```
//-------------------------------------------------------------------
Edge* Mesh::GetNextEdgeOfFace(Face* face, LineOrient& orient) {
//-------------------------------------------------------------------
   if ( edge_of_face_iterator->loop_right->HasFace(face)) {
      edge_of_face_iterator = edge_of_face_iterator->loop_right;
   } else {
      edge_of_face_iterator = edge_of_face_iterator->loop_left;
   }
   if (edge_of_face_iterator->face_right == face) orient = FORWARD;
   else                                           orient = BACKWARD;
   return edge_of_face_iterator;
}
```

A lap csúcsait a lap éleiből úgy kaphatjuk meg, hogy vesszük az élek kezdőpontját. Az élek kezdőpontját az élek irányítottságának megfelelően jelöljük ki:

```
//-----------------------------------------------------------------
Vertex* Mesh::GetNextVertexOfFace(Face* face) {
//-----------------------------------------------------------------
   LineOrient orient;
   Edge* nextedge = GetNextEdgeOfFace(face, orient);
   switch (orient) {
   case FORWARD:  return nextedge->vertex_start;
   case BACKWARD: return nextedge->vertex_end;
   }
}
```

A szárnyas él adatstruktúrát általában a topológiai helyességet hangsúlyozó *B-rep modellező*k (*Boundary Representation*) használják. Vannak azonban olyan szituációk, amikor nincs szükségünk a teljes topológiai információra. Például egy *sugárkövetés*en alapuló algoritmusban nem fogunk a testek éleire hivatkozni, így az élek kapcsolódását a pontokhoz és a poligonokhoz felesleges és pazarló lenne tárolni. Az ilyen esetekben általában elég egy olyan adatszerkezet, amelyben a poligonokat egy tömbbe szervezzük, és minden poligonhoz a körüljárási iránynak megfelelően egy mutatótömb tartozik. A tömbben tárolt mutatók a csúcspontokra mutatnak.

5.2.3. Parametrikus felületek

A parametrikus felületeket vezérlőpontokkal definiáljuk, amelyeket egy kétdimenziós tömbben tárolhatunk. NURBS felületeknél a vezérlőpontok nem csak a koordinátákat tartalmazzák, hanem a vezérlőpont súlyát is. Másrészt a NURBS felületekhez a csomóértékek kétdimenziós tömbjét is meg kell adni, amelyben több elem van, mint a vezérlőpontok száma.

5.3. Világmodellek fájlokban

Az állományokban tárolt virtuális világ szerkezetére számos, széles körben elfogadott megoldás ismeretes. Ezek egy része valóban termékfüggetlen és szabványnak tekinthető (VRML (*.wrl)[4], IGES (*.ige, *.igs), MGF (*.mgf) stb.). Másik részük elterjedt modellező, vagy képszintézis programok leíró nyelvei (POVRAY (*.pov), Maya ASCII és bináris (*.ma, *.mb), 3D Studio (*.3ds), 3ds max (*.max), AutoCAD (*.dxf, *.dwg), Wavefront (*.obj), Open Inventor (*.iv) stb.). Amennyiben magunk írunk grafikus rendszert, akkor azt is célszerű felkészíteni valamely elterjedt formátum megértésére, mert ebben az esetben könnyen átvehetjük a mások által sok fáradtság árán létrehozott modelleket. Elegendő egy gyakori formátum értelmezését beépíteni a programba, hiszen

[4] http://www.web3d.org

léteznek olyan konverziós programok (PolyTrans[5], Crossroads 3D[6]), amelyek a szabványos formátumokat egymásba átalakítják.

A fájlok lehetnek binárisak, vagy szövegesek egyaránt. A bináris fáljok a memória adatszerkezetek leképzései, így viszonylag könnyen beolvashatók. A szöveges fájlok viszont emberi fogyasztásra is alkalmasak, ilyen leírásokat ugyanis akár egy szövegszerkesztővel is előállíthatunk, illetve módosíthatunk. Ezen kétségtelen előny mellett, a szöveges fájlokat sokkal nehezebb beolvasni, mint a binárisakat. A következőkben a szöveges fájlformátumok gépi értelmezésével foglalkozunk, egy bináris fájlformátummal pedig a 10.5. fejezetben fogunk megismerkedni.

5.3.1. Formális nyelvek

A szöveges fáljformátumok a színtér elemeit *formális nyelv*en írják le, ezért egy kis kitérőt kell tennünk a természetes és formális nyelvek világába. A természetes nyelvek legszebbike a magyar, a formális nyelvekhez pedig például a programozási nyelvek sorolhatók. A tárolt információ megismeréséhez tehát ezt a nyelvet kell megértenünk. A formális nyelvek [60] a természetes nyelvekhez hasonlóan szavakból és speciális jelekből állnak, amelyek a nyelv nyelvtani szabályai szerinti sorrendben követhetik egymást. A szavak betűkből épülnek fel. Több szót nem szabad egymás után írni, hanem szóközökkel kell őket elválasztani. Egy speciális jel egyetlen betű, és szemben a szavakkal, ezeket egymás után és a szavak után akár szóközök nélkül is leírhatjuk.

Vegyünk példaként egy nagyon egyszerű természetes nyelvet! A nyelv magyarnak hangzik, de természetesen nem vállalkozunk arra, hogy a magyar nyelv teljes szókészletét és nyelvtanát áttekintsük, ezért a példanyelvünk a természetes nyelvnél lényegesen egyszerűbb. A nyelv szavai főnevekből (például *„Józsi”*, *„Sör”*) és igékből (például *„iszik”*, *„kedveli”*) állnak. Német hatásra, a főneveket az igéktől úgy különböztetjük meg, hogy a főnevek mindig nagybetűvel, az igék pedig mindig kisbetűvel kezdődnek. A nyelv speciális jelei a mondatvégi pont („.”) és a tárgyrag (*„t”*). A nyelv szavai nem tartalmaznak sem pontot, sem *„t”* betűt, így nem kell azon tanakodnunk, hogy ha ilyen jelet találunk, akkor az vajon mondatvégi pont illetve tárgyrag, vagy pedig egy szó része. A szavakat a szóköz (*space*) karakter választhatja el.

Egy szöveg tehát főnevekből, igékből, tárgyragokból és mondatvégi pontokból állhat, amelyeket összefoglalóan a nyelv *terminális szimbólumai*nak, vagy *token*jeinek nevezünk.

Egy nyelv tokenjeit, azaz szavait és speciális jeleit nem használhatjuk tetszőleges sorrendben. A példanyelvünk szókincsével a *„Jani Sört iszik.”* helyesnek hangzik, de a *„Sör Jani Vali.”* már meglehetősen furcsa. A szavak és speciális jelek lehetséges sorrendjét a *nyelvtan* definiálja. A nyelvtan kimondhatja, hogy egy mondat alannyal kez-

[5] http://www.okino.com/conv/conv.htm

[6] http://home.europa.com/~keithr/

dődik, amelyet tárgy követhet, végül mindig állítmánnyal fejeződik be, és a mondatot pont zárja. Az alany helyén főnév állhat, a tárgy helyén ugyancsak főnév, amelyet a *„t”* tárgyrag egészít ki, az állítmány viszont csak ige lehet.

Figyeljük meg, hogy a nyelvtani szabályok új fogalmakat vezetnek be (*mondat, alany, állítmány* stb.) és megkötik, hogy ezeket a fogalmakat hogyan lehet helyettesíteni újabb fogalmakkal illetve a nyelv szavaival. Azokat a fogalmakat, amelyeket más fogalmak fejtenek ki, *nem terminális szimbólum*oknak nevezzük. A nyelv tokenjeit, azaz szavait és speciális jeleit már semmivel sem lehet helyettesíteni, így ezek a *terminális szimbólum*ok. Ahhoz, hogy a nyelv egy lehetséges szövegét előállítsuk, a *Szöveg* nem terminális szimbólumra az összes lehetséges helyettesítést el kell végezni és meg kell vizsgálni, hogy valamelyik eredményeként a vizsgált szöveget kapjuk-e. A helyettesítések eredményeként újabb nem terminális szimbólumok keletkezhetnek, amelyekre ismét az összes lehetséges helyettesítést megcsináljuk. Az eljárást addig kell folytatni, amíg már csak terminális szimbólumok sorozataival állunk szemben.

A helyettesítési szabályokhoz egy formális jelölésrendszert is megadhatunk. Itt a bal oldalon a nem terminális szimbólumok állnak, a jobb oldalon pedig azon terminális vagy nem terminális szimbólumok sorozata, amely a bal oldalon lévő szimbólumot helyettesíti. Az előbb vázolt egyszerű nyelv nyelvtanát az alábbi szabályok definiálják:

$$
\begin{aligned}
\langle \textbf{Szöveg} \rangle &\rightarrow \{\langle \textbf{Mondat} \rangle\} \\
\langle \textbf{Mondat} \rangle &\rightarrow \langle \textbf{Alany} \rangle + \langle \textbf{Cselekvés} \rangle + \langle . \rangle \\
\langle \textbf{Cselekvés} \rangle &\rightarrow \langle \textbf{Állítmány} \rangle \\
\langle \textbf{Cselekvés} \rangle &\rightarrow \langle \textbf{Tárgy} \rangle + \langle \textbf{Állítmány} \rangle \\
\langle \textbf{Alany} \rangle &\rightarrow \langle \textit{Főnév} \rangle \\
\langle \textbf{Tárgy} \rangle &\rightarrow \langle \textit{Főnév} \rangle + \langle t \rangle \\
\langle \textbf{Állítmány} \rangle &\rightarrow \langle \textit{Ige} \rangle
\end{aligned}
$$

A formális szabályok között új jelölések is felbukkantak. A nem terminális és a terminális szimbólumokat is ⟨ ⟩ jelek közé tesszük, azonban a nem terminálisokat vastag betűvel szedjük. A terminális szimbólumok konkrét helyettesítését „ ” jelek közé tesszük, és ezeket nem szedjük vastag betűvel.

A { } kapcsos zárójel az ismétlésre utal, tehát az első szabály szerint a ⟨**Szöveg**⟩ $0, 1, 2, \ldots$ darab ⟨**Mondat**⟩ból állhat. A + összeadásjel az egymás utáni felsorolást jelenti, azaz a második szabály szerint a ⟨**Mondat**⟩ ⟨**Alany**⟩nyal kezdődik, amelyet ⟨**Cselekvés**⟩ követhet, végül pedig a mondatformát pont zárja.

Most már mindent tudunk egyszerű nyelvtanunkról, tehát arra is választ adhatunk, hogy helyes-e a *„Józsi Sört iszik.”* mondat. Fejlett emberi intelligenciával szinte helyettesítések nélkül azonnal megállapítjuk, hogy a vizsgált szöveg a következő tokenekből áll: ⟨*Főnév*⟩+⟨*Főnév*⟩+⟨*t*⟩+⟨*Ige*⟩+⟨.⟩. Egy számítógép számára azonban az értelmezést algoritmizálni kell. Azt kell ellenőrizni,

hogy ez a sorozat levezethető-e a ⟨**Szöveg**⟩ből. A ⟨**Szöveg**⟩re egyetlen helyettesítési szabályt ismer a nyelvtan:

$$\langle \textbf{Szöveg} \rangle \quad \rightarrow \quad \{\langle \textbf{Mondat} \rangle\}$$

Tehát a *„Józsi Sört iszik."*-et ⟨**Mondat**⟩oknak kell megfeleltetni. A ⟨**Mondat**⟩hoz ugyancsak egyetlen helyettesítési szabály tartozik:

$$\langle \textbf{Mondat} \rangle \quad \rightarrow \quad \langle \textbf{Alany} \rangle + \langle \textbf{Cselekvés} \rangle + \langle . \rangle$$

Ezek szerint a *„Józsi Sört iszik."*, csak akkor lehet helyes, ha ⟨**Alany**⟩nyal kezdődik. Az ⟨**Alany**⟩ra vonatkozó szabályok szerint, az ⟨**Alany**⟩ csak ⟨*Főnév*⟩ lehet:

$$\langle \textbf{Alany} \rangle \quad \rightarrow \quad \langle \textit{Főnév} \rangle$$

A *„Józsi"* ⟨*Főnév*⟩. A helyettesítések sorozatával tehát sikerül egy terminális szimbólumhoz jutnunk, amely megegyezik az éppen vizsgált szövegünk első tokenjével. Idáig tehát rendben vagyunk, a szövegünk pedig pontosan akkor helyes, ha a maradékra is el tudjuk végezni ezt a műveletet. Vágjuk le tehát a vizsgált szövegből a felismert terminális szimbólumot, a *„Sört iszik."* (szerkezetét tekintve ⟨*Főnév*⟩ + ⟨*t*⟩ + ⟨*Ige*⟩ + ⟨.⟩) mondatrészlettel pedig térjünk vissza oda, ahol a vizsgálatot abbahagytuk, azaz a második szabályhoz:

$$\langle \textbf{Mondat} \rangle \quad \rightarrow \quad \langle \textbf{Alany} \rangle + \langle \textbf{Cselekvés} \rangle + \langle . \rangle$$

Az ⟨**Alany**⟩t már megtaláltuk, most már csak azt kell ellenőrizni, hogy a *„Sört iszik."* helyettesíthető-e egy ⟨**Cselekvés**⟩sel és a mondatvégi ponttal. A ⟨**Cselekvés**⟩re két szabály is alkalmazható, hiszen a ⟨**Cselekvés**⟩ állhat csak ⟨**Állítmány**⟩ból vagy pedig ⟨**Tárgy**⟩ból és ⟨**Állítmány**⟩ból. Először az első szabályt alkalmazzuk, és ⟨**Állítmány**⟩ra helyettesítünk. Ezt azonban csak egyféleképpen tudjuk folytatni:

$$\langle \textbf{Állítmány} \rangle \quad \rightarrow \quad \langle \textit{Ige} \rangle$$

A *„Sör"* azonban nem ⟨*Ige*⟩, ez az ág tehát kudarcba fulladt, ezért lépjünk vissza egy szintet. Próbálkozzunk a második lehetséges helyettesítéssel:

$$\langle \textbf{Cselekvés} \rangle \quad \rightarrow \quad \langle \textbf{Tárgy} \rangle + \langle \textbf{Állítmány} \rangle$$

Alkalmazzuk a ⟨**Tárgy**⟩ra az egyetlen lehetséges helyettesítést:

$$\langle \textbf{Tárgy} \rangle \quad \rightarrow \quad \langle \textit{Főnév} \rangle + \langle t \rangle$$

A szövegünk: *„Sört iszik."* Örömmel állapíthatjuk meg, hogy ismét sikerült két terminális szimbólumot felismernünk. Vágjuk le a vizsgált szövegből ezeket a szimbólumokat! Így már csak az *„iszik."* szekvencia képezi a vizsgálódásunk tárgyát. Ha ezután elvégezzük az

$$\langle \textbf{Állítmány} \rangle \quad \rightarrow \quad \langle \textit{Ige} \rangle$$

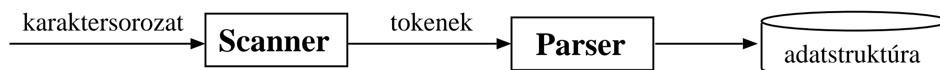
helyettesítést, és a felismert $\langle$*Ige*$\rangle$-t kivágjuk a szövegből, akkor már csak a „." maradt. Ez pedig éppen redukálható a $\langle$**Mondat**$\rangle$-ra vonatkozó, már alkalmazott helyettesítés utolsó szimbólumával ($\langle . \rangle$). Így a „." is eltűnik, és eredményként egy üres sztringet kapunk. Ez azt jelenti, hogy az elemzés sikerült, a *„Józsi Sört iszik."* egy helyes mondata a definiált nyelvnek.

Az ismertetett elemzési stratégiában érdemes néhány tulajdonságot kiemelni. Annak érdekében, hogy megállapítsuk, hogy egy szöveg levezethető-e, a $\langle$**Szöveg**$\rangle$ nem terminális szimbólumból indultunk ki, és a nyelvtani szabályok bal oldalán álló nem terminális szimbólumokat helyettesítettük rekurzívan a nyelvtani szabályok jobb oldalaival. Ezt a megközelítést *balelemzés*nek nevezzük. Eljárhatnánk úgy is, hogy magából az elemzett szövegből indulunk ki, és a szabályok jobb oldalát cserélgetjük a bal oldalon álló nem terminális szimbólumra egészen addig, amíg az elemzett szövegből a $\langle$**Szöveg**$\rangle$ szimbólumig el nem jutunk. Ekkor *jobbelemzés*t végeznénk.

A másik fontos észrevétel az, hogy a helyettesítés nem mindig egyértelmű. Például egy $\langle$**Cselekvés**$\rangle$ nem terminálist kicserélhetünk $\langle$**Tárgy**$\rangle$ra és $\langle$**Állítmány**$\rangle$ra vagy pedig csak $\langle$**Állítmány**$\rangle$ra. Elvileg megkísérelhetnénk mind a két utat, és ha valamelyik kudarchoz vezetne, akkor csak a másik úton haladnánk tovább. A kudarcot senki sem szereti, nem beszélve a felesleges munkáról. A kérdés tehát, hogy elkerülhetjük-e a kudarcélményt úgy, hogy a több lehetőség közül szerencsés kézzel mindig mellőzzük azokat, amelyek kudarchoz vezetnek. Úgy teszünk mint a türelmetlen Harry Potter olvasó, aki izgalmában előrelapoz, és megnézzük az elemzendő szövegünk következő szavát. Midőn azon elmélkedünk, hogy a *„Sört iszik."* mondatrész elemzésekor ezt a $\langle$**Cselekvés**$\rangle$t $\langle$**Tárgy**$\rangle$ra és $\langle$**Állítmány**$\rangle$ra, vagy csak $\langle$**Állítmány**$\rangle$ra kell-e bontani, a *„Sör"* szövegrészt vesszük górcső alá (a tárgyragot már nem, mert az már a következő szimbólum). A nyelvtan szabályai szerint a *„Sör"*-ből sohasem lehet $\langle$**Állítmány**$\rangle$, $\langle$**Tárgy**$\rangle$ viszont igen, tehát a $\langle$**Cselekvés**$\rangle$ két lehetséges helyettesítési szabályából csak azt alkalmazhatjuk, amelyikben a $\langle$**Cselekvés**$\rangle$t $\langle$**Tárgy**$\rangle$ra és $\langle$**Állítmány**$\rangle$ra bontjuk. Egy szó előreolvasása tehát feloldotta a gordiuszi csomót. Természetesen nem lehetünk biztosak abban, hogy bármilyen nyelvtannál ezt ilyen egyszerűen elintézhetjük, de egyszerű nyelvünknél, sőt a programozási nyelvek döntő többségénél is igen. Az olyan nyelvtani szabályrendszert, ahol a balelemzés során fellépő többértelműséget egyetlen következő szó ismeretében feloldhatjuk, *LL(1)* nyelvnek nevezzük. A továbbiakban ilyen nyelvekkel foglalkozunk.

A nyelvtani helyesség ellenőrzése kritikus lépés a szöveg megértésében és feldolgozásában. Ha a szöveg nyelvtanilag helytelen, nem tudunk vele mit kezdeni és visszadobjuk (fordítási hiba). Ha viszont helyes, a nyelvtani elemzés során azonosítjuk a szöveg egységeit, amivel összekapcsolhatjuk a megértés és a fordítás lépéseit is. Gondoljunk arra, hogy egyszerű nyelvünket angolra szeretnénk fordítani! A mondat elején álló alany felismerése után ezt a szót rögtön fordíthatjuk, a tárgy és állítmány párt pedig akkor, amikor a mondat végére értünk. Ez azt jelenti, hogy a nyelvtani, úgynevezett *szintaktikai elemzés*t nem csupán a helyesség eldöntéséhez használjuk, hanem a megértést is ezzel vezéreljük.

Egy általános beolvasó felépítését szemlélteti az 5.10. ábra. A bemeneti állomány karakterekből áll. Az értelmezés első lépése a szavak és egyéb lexikális szimbólumok felismerése, valamint a lényegtelen részek (megjegyzések, üres karakterek) eldobása. Ezt a műveletet egy *lexikális elemző* (*Scanner*) végzi el. A lexikális elemző kimenete az azonosított egység típusa (valamelyik terminális szimbólum) és tartalma. A típusokat *token*eknek is hívják. A típus egy speciális jelet egyértelműen azonosít, egy $\langle$*Főnév*$\rangle$ további feldolgozásához azonban a tartalmat is jó tudni, azaz, hogy éppen *„Józsi”*-ról vagy a *„Sör”*-ről van-e szó. Vannak egyértelműbb megfeleltetések is, például a $\langle . \rangle$ terminális szimbólumhoz mindig a „.” konkrét helyettesítés tartozik. A tokeneket az *értelmező* (*Parser*) dolgozza fel, amely ez alapján elvégzi a nyelvtani helyettesítéseket és megállapítja, hogy a mondat helyes eleme-e a nyelvnek.



5.10. ábra. *Egy általános beolvasó felépítése*

A lexikális elemző (*Scanner*) és az értelmező (*Parser*) elkészítését egy konkrét nyelv értelmezőjének megvalósításával mutatjuk be. A nyelv egyszerű, talán nem is kellene a formális nyelvek teljes fegyvertárát bevetni a beolvasójának elkészítéséhez. Mégis ezt az utat követjük, mert ez az eljárás tetszőlegesen bonyolult nyelveknél is alkalmazható.

5.3.2. Wavefront OBJ fájlformátum beolvasása

A Wavefront *OBJ* fájlformátumával ebben a témakörben azért foglalkozunk, mert ez az egyik legkönnyebben érthető és elemezhető világmodell leíró, szöveges fájlformátum. Egy egyszerű színtér beolvasásával és a hozzá tartozó elemző megírásával szeretnénk a gyakorlatban is kamatoztatni formális nyelvekről elsajátított ismereteinket. Először magát a Wavefront fájlformátumot ismertetjük. A példa színterünk egyetlen négyszöget definiál:

```
v 0.0 0.1 0.0
v 1.0 0.0 0.0
v 1.0 1.0 0.0
v 0.0 1.0 0.0

vt 0.0 0.0
vt 1.0 0.0
vt 1.0 1.0
vt 0.0 1.0

vn  1.0  0.0  0.0

f 1/1/1 2/2/1 3/3/1 4/4/1
```

A fájl először a csúcspontokat sorolja fel a `v` kulcsszóval, majd textúra pontokat a `vt` kulcsszóval és a normál vektorokat `vn` kulcsszóval adja meg, végül pedig mindezeket az `f` utasítás egymáshoz és lapokhoz rendeli. A lap utasításban például a lapot négy csúcsponttal adtuk meg (ez egy négyszög), és `/` jellel elválasztva minden csúcshoz közöltük a csúcspont, a textúrapont és a normálvektor sorszámát.

A beolvasó programhoz a tokeneket előállító `Scanner`-t és a nyelvtani szabályokat értelmező `Parser`-t kell megírnunk. A `Scanner` osztály feladata a bemeneti karaktersorozat összetartozó elemeinek, az úgynevezett *token*eknek az azonosítása. Például a C nyelvben egy token lehet egy speciális jel (például `*`), egy kulcsszó (`if`, `for` stb.), egy konstans (`123`), vagy akár egy változó vagy függvény neve. Az OBJ fáljformátumban a kulcsszavak a következők: `v` = csúcspont, `vn` = normál vektor, `vt` = textúra koordináta, `f` = lap. Egyetlen speciális karaktert találunk a `/` elválasztó jelet. A számok számjegyeket, előjelet és tizedespontot tartalmazhatnak. Végül a nem kulcsszó és nem szám karaktersorozatok a változók, ilyen a fenti példában nem szerepel. A `Scanner`-nek tehát ezeket az elemeket kell szétválogatni.

A lehetséges tokeneket egy felsorolás típussal adjuk meg, a kulcsszavakat és a speciális karaktereket pedig táblázatok segítségével kapcsoljuk a tokenazonosítókhoz:

```
//-----------------------------------------------------------------
enum Tokens {  // az OBJ nyelv tokenjei
//-----------------------------------------------------------------
   VERTEX_TOKEN,          // ,,v''
   VERTEX_NORMAL_TOKEN,   // ,,vn''
   VERTEX_TEXTURE_TOKEN,  // ,,vt''
   FACE_TOKEN,            // ,,f''
   SEPARATOR_TOKEN,       // ,,/''
   NUMBER_TOKEN,          // egész szám
   REAL_TOKEN,            // lebegőpontos szám
   NAME_TOKEN             // szöveg
};
//-----------------------------------------------------------------
SpecialChar specials[] = { // speciális karakterek táblázata
//-----------------------------------------------------------------
      { '/',       SEPARATOR_TOKEN }
};
```

```
//-----------------------------------------------------------------
Keyword keywords[] = {    // kulcsszavak táblázata
//-----------------------------------------------------------------
      { "v",       VERTEX_TOKEN },
      { "vn",      VERTEX_NORMAL_TOKEN   },
      { "vt",      VERTEX_TEXTURE_TOKEN  },
      { "f",       FACE_TOKEN   }
};
```

A `Scanner` mindenekelőtt az elválasztó jelekig (szóköz, tabulátor, új sor) gyűjti az egymás utáni karaktereket, majd megvizsgálja, hogy az kulcsszó-e vagy pedig a programozó által megadott név. A `Scanner` mindig az aktuális karaktert tárolja, illetve előresandít a fájlban és megnézi, hogy mi a következő karakter. Erre azért van szüksége, mert csak a következő karakter alapján ismerheti fel, hogy az aktuális karakter a token utolsó karaktere-e. Amíg a token nem áll össze, a hozzá tartozó karakterek a `token_buffer` karaktertömbbe kerülnek. A `Scanner` osztályhoz a karakterek osztályozása (elválasztó, szám, betű stb.) tartozik:

```
//-----------------------------------------------------------------
class Scanner : public InputFile {
//-----------------------------------------------------------------
   char          curr_char, next_char; // aktuális és következő karakter
   TokenBuffer   token_buffer;         // aktuális tokenhez tartozó sztring
   Token         current_token;        // aktuális token
   char   Read();                // beolvassa a következő karaktert és lép
   char   InspectNext();         // bekéri a következő karaktert, de nem lép
   void   Advance() { Read(); }  // lép
protected:
   int  IsEOF(char c)    { return (int)(c == EOF); }
   bool IsWhite(char c)  { return (c == ' ' || c == '\t' || c == '\r'); }
   bool IsLetter(char c) { return (('a'<=c && c<='z') || ('A'<=c && c<='Z'));
   int  IsDecimal(char c){ return (int)('0'<= c && c <= '9');  }
public:
   Scanner(char* filename) : InputFile( filename ) { }
   Token GetToken(void);       // következő token megkeresése
   Token GetCurrentToken(void) { return current_token; } // aktuális token
   int   GetNumber(void);      // egész szám illesztése és lekérése
   float GetReal(void);        // lebegőpontos szám illesztése és lekérése
   void  Match(Token t) {      // egy tetszőleges token illesztése
      if (current_token == t) GetToken(); // illeszkedés, jöhet a következő
      else exit(-1);                      // nem a várt token, hiba
   }
};
```

A `Match()` eljárás az aktuális tokent a várt tokennel veti össze, illeszkedés esetén a következő tokenre lép, eltéréskor viszont hibát érzékelve leáll. A `GetNumber()` és `GetFloat()` az illeszkedésvizsgálat speciális formái, amelyek egész, illetve lebegőpontos számokat várnak, és a számként értelmezhető karaktersorozatot számmá alakítják át. Elemző programunk `Scanner` osztályának lelke a `GetToken()` függvény, amely a fájlból a következő azonosítható karaktersorozattal, valamint az annak megfelelő tokennel tér vissza:

```
//-----------------------------------------------------------------
Token Scanner::GetToken( ) {
//-----------------------------------------------------------------
   token_buffer.Clear( );      // a tokenhez tartozó karaktertömb ürítése
   while (!IsEOF(curr_char)) { // addig olvass, amig a token nem teljes
      curr_char = Read( );     // aktuális karakter
      if (IsWhite(curr_char)) continue;         // szóköz->eldob
      for(int i = 0; i < sizeof specials; i++)  // speciális karakter?
         if ( specials[i].c == curr_char ) {
            current_token = specials[i].token;
            return current_token;               // speciális karakter!
      }
      if (curr_char == '-') {                   // - jel?
         token_buffer.Put(curr_char);           // be a bufferbe
         curr_char = Read();
      }
      if (IsDecimal(curr_char)) {               // számjegy?
         token_buffer.Put(curr_char);           // bufferbe
         bool real = FALSE;
         for( ; ; ) {                           // további számjegyek
            next_char = InspectNext( );
            if (IsDecimal(next_char)) {
               token_buffer.Put(next_char);
               Advance( );
            } else if (next_char =='.' && !real) { // ha . akkor lebegőpont
               real = TRUE;
               token_buffer.Put(next_char);
               Advance( );
            } else {
               current_token = (real) ? REAL_TOKEN : NUMBER_TOKEN;
               return current_token;
            }
         }
      }
      if (IsLetter(curr_char)) {                // szó
         token_buffer.Put(curr_char);           // bufferbe
         for( ; ; ) {                           // további betűk
            next_char = InspectNext( );
            if (!IsLetter(next_char)) {
               for(int i = 0; i < sizeof keywords; i++)  // kulcsszó?
                  if (strcmp(keywords[i].key, token_buffer) == 0) {
                  current_token = keywords[i].token;
                  return current_token;                  // kulcsszó!
               }
               return NAME_TOKEN;                        // név
            } else {
               token_buffer.Put(next_char);
               Advance( );
            }
         }
      }
   }
   current_token = EOF_TOKEN;
   return current_token;
}
```

Az eljárás addig olvas, amíg egy összetartozó egységet fel nem ismer. A szóközök átlépése után, először a speciális karaktereket vizsgáljuk. Ha nem találunk ilyet, akkor fel kell készülnünk arra, hogy a többi elem több karakterből állhat össze, ezért a felismerésükig a `token_buffer`-ben gyűjtögetjük a karaktereket. A mínuszjel egyszerűen a bufferbe kerül, a számoknál azonban figyelünk arra, hogy a számjegyek közé már nem vegyülhetnek elválasztó karakterek, és a szám akkor fejeződik be, ha nem szám, vagy egy második tizedespont érkezik. A betűvel induló elemek karaktereit addig gyűjtjük, amíg nem betűt kapunk (például szóközt), és ekkor megvizsgáljuk, hogy az idáig összeálló karaktersorozat vajon azonos-e valamelyik kulcsszóval. Ha nem, csakis változónév lehet.

A `Scanner` kimenete a tokenek sorozata, amelyet a `Parser` a nyelvtani szabályoknak megfelelően dolgoz fel. Az OBJ fájlformátum formális nyelvében a $\langle v \rangle$ (*vertex*) terminális szimbólum pontokat vezet be, amelyeket 3 koordinátájukkal (x, y, z) adunk meg. A pontok sorrendje lényeges, hiszen a későbbiekben a sorszámukkal hivatkozunk az egyes csúcspontokra. A $\langle vt \rangle$ (*vertex texture*) a textúratérben azonosít pontokat, a $\langle vn \rangle$ (*vertex normal*) pedig normálvektorokat vezet be. Az OBJ fájlformátum sokszögeket definiál. Egy sokszög az $\langle f \rangle$ (*face*) kulcsszóval indul, amelyet a sokszög csúcspontjai követnek. Minden csúcspontban a pont, a textúra koordináta és a normálvektor sorszámára hivatkozunk. A textúra koordináta és normálvektor sorszám opcionális. Egyetlen csúcspont, textúrapont és normálvektor sorszámait „/” karakterrel választjuk el egymástól.

Összefoglalva, az OBJ formális nyelv kulcsszavakból ($\langle v \rangle$, $\langle vt \rangle$, $\langle vn \rangle$, $\langle f \rangle$), speciális karakterekből ($\langle / \rangle$) és számokból ($\langle Float \rangle$, $\langle Integer \rangle$,) épül fel.

Az OBJ nyelv nyelvtanát az alábbi LL(1) szabályokkal adhatjuk meg:

$$\begin{aligned}
\langle \textbf{OBJFile} \rangle &\rightarrow \{\langle \textbf{Vertex} \rangle\} + \{\langle \textbf{VertexTexture} \rangle\} + \{\langle \textbf{VertexNormal} \rangle\} + \{\langle \textbf{Face} \rangle\} \\
\langle \textbf{Vertex} \rangle &\rightarrow \langle v \rangle + \langle Float \rangle + \langle Float \rangle + \langle Float \rangle \\
\langle \textbf{VertexTexture} \rangle &\rightarrow \langle vt \rangle + \langle Float \rangle + \langle Float \rangle \\
\langle \textbf{VertexNormal} \rangle &\rightarrow \langle vn \rangle + \langle Float \rangle + \langle Float \rangle + \langle Float \rangle \\
\langle \textbf{Face} \rangle &\rightarrow \langle f \rangle + \{\langle \textbf{VertexOfFace} \rangle\} \\
\langle \textbf{VertexOfFace} \rangle &\rightarrow \langle Integer \rangle + [\langle / \rangle + [\langle Integer \rangle] + [\langle / \rangle + [\langle Integer \rangle]]]
\end{aligned}$$

A [ ] szögletes zárójel az opcionalitás jele, azaz a benne foglalt fogalom egyszer vagy egyszer sem jelenik meg.

Az értelmezőt rekurzív ereszkedő stratégiával készítjük el. Ez azt jelenti, hogy minden nyelvtani szabályhoz egy függvényt írunk, amely megpróbálja a jobb oldal elemeit illeszteni. Egy terminális illesztése a `Scanner`-től kapott token és a nyelvtani szabály alapján várható token összehasonlításából áll. Ha megegyeznek, minden rendben van, lépünk tovább. Ha nem egyeznek meg, a fájl nem felel meg a nyelvtani szabályoknak.

Amennyiben a jobb oldalon nem terminális is feltűnik, akkor lennie kell olyan szabálynak, amelyben ez a nem terminális éppen a bal oldalon szerepel, tehát léteznie kell ezt a szabályt illesztő függvénynek is. Meghívjuk tehát ezt a függvényt, és rábízzuk a további illesztést. Az eljárás azért kapta az ereszkedő nevet, mert először a teljes fájlnak megfelelő szabályt próbáljuk illeszteni, majd annak jobb oldalát, aztán a jobb oldalon álló nem terminálisok feloldását stb. A rekurzív jelző arra utal, hogy előfordulhat, hogy egy nem terminális szabályának feloldása során előbb-utóbb újból ugyanezen típusú, nem terminális szimbólumot kell illeszteni. A programunk tehát rekurziót végezhet.

Először az első nyelvtani szabály elemzőrutinját írjuk meg. Egy OBJ fájlban pontokat, normálvektorokat, textúrapontokat és lapokat sorolhatunk fel. A kérdés csak az, hogy honnan vesszük észre, hogy a csúcsok, normálvektorok stb. elfogytak, így olvasásukat be kell fejezni? Ehhez használhatjuk az LL(1) egy tokent előreolvasó stratégiáját. Figyeljük meg, hogy a nyelvtani szabályokban a csúcsok a $\langle v \rangle$ terminálissal kezdődnek, a lapok pedig az $\langle f \rangle$ terminálissal! Ha például a pontok feldolgozása során előreolvasunk, és már nem a $\langle v \rangle$ tokent látjuk, akkor befejezhetjük a pontolvasást. Hasonlóképpen a lapolvasást csak addig kell erőltetni, amíg előrelapozva $\langle f \rangle$ tokent látunk.

```
//-----------------------------------------------------------------
void ObjParser::ParseFile() { // {Vertex}+{VertexTexture}+{VertexNormal}+{Face}
//-----------------------------------------------------------------
   GetToken();
   while(GetCurrentToken() == VERTEX_TOKEN)         ParseVertex();
   while(GetCurrentToken() == VERTEX_TEXTURE_TOKEN) ParseVertexTexture();
   while(GetCurrentToken() == VERTEX_NORMAL_TOKEN)  ParseVertexNormal();
   while(GetCurrentToken() == FACE_TOKEN)           ParseFace();
}
```

Amikor arra a következtetésre jutunk, hogy egy $\langle v \rangle$ (`VERTEX_TOKEN`)-nek kell jönnie, akkor a `Scanner` osztály `Match()` eljárásával ellenőrizzük, hogy valóban az jött-e, és rögtön a következő token feldolgozásába kezdünk. A `ParseVertex()` nem csupán nyelvtani elemzést végez, hanem a beolvasott fájl tartalmának megfelelően építgeti a geometriai adatstruktúrát is, és amikor egy csúcspont előáll, az 5.2.2. fejezetben megismert `Mesh` típusú adatstruktúrába írja a beolvasott információt:

```
//-----------------------------------------------------------------
void ObjParser::ParseVertex() { // v + Float + Float + Float
//-----------------------------------------------------------------
   Match(VERTEX_TOKEN);  // kulcsszó illesztés
   float x = GetReal(), y = GetReal(), z = GetReal();
   mesh->AddVertex(Vector(x, y, z));
}
```

Az OBJ lapleírásában találjuk az alakzat teljes topológiai információját, így ebben a fázisban hozzuk létre a szárnyas él adatstruktúra éleit és lapjait:

```
//-----------------------------------------------------------------
void ObjParser::ParseFace() { // f + { VertexOfFace }
//-----------------------------------------------------------------
   Match(FACE_TOKEN);                          // kulcsszó illesztés
   Vertex* vertex_start = ParseVertexOfFace();  // első csúcs
   Vertex* vertex = ParseVertexOfFace();        // második csúcs
   Edge* edge = mesh->AddEdge(vertex_start, vertex); // él
   Face* face = mesh->AddFace(vertex_start, vertex); // lap
   mesh->LinkEdgeToFace(face, vertex_start, vertex);
   Vertex* vertex_prev;
   while(GetCurrentToken() == NUMBER_TOKEN) {   // további csúcsok
      vertex_prev = vertex;
      vertex = ParseVertexOfFace();
      edge = mesh->AddEdge(vertex_prev, vertex);
      mesh->LinkEdgeToFace(face, vertex_prev, vertex);
   }
   edge = mesh->AddEdge(vertex, vertex_start);
   mesh->LinkEdgeToFace(face, vertex, vertex_start);
}

//-----------------------------------------------------------------
Vertex* ObjParser::ParseVertexOfFace() { // Integer+[/+[Integer]+[/+[Integer]]]
//-----------------------------------------------------------------
   int texture_idx, normal_idx;
   int vertex_idx = GetNumber();                 // csúcspont index
   if (GetCurrentToken() == SEPARATOR_TOKEN) {    // lehet textúraindex is
      Match(SEPARATOR_TOKEN);
      if (GetCurrentToken() == NUMBER_TOKEN) texture_idx = GetNumber();
      if (GetCurrentToken() == SEPARATOR_TOKEN) { // lehet normálindex is
         Match(SEPARATOR_TOKEN);
         if (GetCurrentToken() == NUMBER_TOKEN) normal_idx = GetNumber();
      }
   }
   return mesh->GetVertex(vertex_idx - 1);       // a hivatkozott csúcs
}
```

5.3.3. A VRML 2.0 fájlformátum beolvasása

Az OBJ fájlok értelmezéséhez képest a VRML színterek beolvasása — a szabályok számának és komplexitásának növekedése miatt — sokkal nehezebb feladat. Rengeteg programozási munkától kímélhetjük meg magunkat, ha keresünk egy szabadon felhasználható szoftver csomagot, és ezt építjük be a programunkba.

A VRML színterek beolvasásához egy ilyen szabad szoftvert, az OpenVRML-t [4] fogjuk felhasználni. Az elérhető VRML elemzők közül ez a legelterjedtebb, és gyakorlatilag teljesen megfelel a VRML97 specifikációnak. Az OpenVRML-t úgy készítették, hogy a legelterjedtebb platformokon (Windows, Linux, Macintosh) használható legyen. A csomagot a kedves Olvasó megtalálja a könyvhöz mellékelt CD-n, a legfrissebb verzió pedig a *http://www.openvrml.org* címről mindig letölthető. A SourceForge-ról [7] letölthető forrásfájlokból először egy *DLL*-t kell készíteni. A saját alkalmazásunkból később ezt a *DLL*-t fogjuk meghívni. (A Windows operációs rendszer alatt használható

könyvtár OpenVrmlWin.dll néven a CD-n megtalálható. Ha megfelel egy ilyen „nem a legfrissebb” verzió (0.12.4-es), akkor a következő pár sort átugorhatjuk.)

Az OpenVrmlWin.dll elkészítésének lépései:

- Hozzunk létre egy Win32 *DLL* projektet[7]!
- Az OpenVRML forrás fájlokat és a lib/antlr könyvtár fájljait tegyük a projektbe (Vrml97Parser.cpp-t és Vrml97Parser.g-t kivéve), és tegyük megjegyzésbe, vagy töröljük a #line sorokat a Vrml97Parser.cpp-ből!
- A vrml97node.cpp annyira nagy fájl, hogy a fordításához a /gz kapcsolót be kell állítani.
- A VRML csomópontok futás közbeni azonosításához a Run-Time Type Info-t be kell kapcsolni.
- Fordítsuk le a *DLL*-t!

Az OpenVRML használatához az OpenVRMLWin.dll és az OpenVRMLWin.lib fájlokra is szükségünk lesz (ezeket állítottuk elő az előző lépésben). A munka megkönnyítésére ezeket a CD-n az OpenVRMLDll/ könyvtárba gyűjtöttük össze. Ha egy olyan alkalmazást szeretnénk készíteni, amely felhasználja az OpenVRML.dll-t, akkor a következőt kell tennünk:

- Készítsük el a Win32 alkalmazás projektet!
- Vegyük fel a VRMLScene.h és field.h fejléc (*header*) fájlokat a kódba, és az elérési útjukat szerepeltessük a fordítási paraméterek között (-I "elérési út")!
- Szerkesszük hozzá az OpenVRMLWin.lib könyvtárat a programhoz (*link*)!
- Másoljuk az OpenVRMLWin.dll-t az alkalmazás futtatható (*.exe) programja mellé, vagy az elérési útját tegyük be a PATH környezeti változóba!

Ezek után az OpenVRMLWin.dll-t az alkalmazásban így tudjuk használni:

```
 // fájlnév alapján a színtér gráf felépítése
OpenVRML::VrmlScene* vrmlScene = new OpenVRML::VrmlScene(pathOrUrl);
// a gráf gyökéréhez tartozó csomópontok lekérdezése
const MFNode& rootNodes = vrmlScene->getRootNodes();
// ha a csomópontok száma nulla, hiba történhetett
if (rootNodes.getLength() == 0) throw "Hiba történt az olvasásban.";
else ::MessageBox(NULL, "A betöltés sikerült.", "Üzenet", MB_OK);
```

Az OpenVRML a Node osztályából öröklődéssel származtatja a színtérgráf csomópontjait. Többféle módszer létezik annak eldöntésére, hogy egy Node* pointer milyen dinamikus típussal rendelkezik. A legegyszerűbb a dinamikus (dynamic_cast) típuskonverzió :

[7] legegyszerűbben a Visual Studio varázslójával készíthetünk Win32 projektet

```
if (dynamic_cast<Vrml97Node::Shape*>(pNode) != NULL)
   HandleShape(dynamic_cast<Vrml97Node::Shape*>(pNode));
```

Használható még a C++ Run-Time Type Information (RTTI) típusazonosítása is:

```
if (typeid(*pNode) == typeid(Vrml97Node::Shape));   // RTTI
   HandleShape(dynamic_cast<Vrml97Node::Shape*>(pNode));
```

Végül igénybe vehetjük a `Node` osztály publikus `nodeType` adattagját, amelynek `id` mezője az osztálynevet tartalmazó sztring.

```
if (pNode->nodeType.id == std::string("Shape"))
   HandleShape(dynamic_cast<Vrml97Node::Shape*>(pNode));
```

A VRML97 specifikáció `Anchor`, `Billboard`, `Collision`, `Transform` és `Group` gyűjtőcsomópontokat definiál. Ezeknek gyermekei lehetnek. Az OpenVRML ezt úgy valósítja meg, hogy a `Group` ősosztály származtatott osztályai az `Anchor`, `Billboard`, `Collision`, `Transform` csomópontok. Gyakori feladat, hogy egy csomópontról el kell dönteni, hogy az gyűjtőcsomópont-e. Ilyenkor ahelyett, hogy mind az 5 csomóponttal megpróbáljuk a `dynamic_cast` műveletet, a `pNode->toGroup()` metódust is alkalmazhatjuk , amely pontosan ezt csinálja.

Az OpenVRML a VRML csomópontok adattagjaihoz egy meglehetősen szokatlan lekérdezési módszert használ. Például egy `Shape` osztály privát `appearance` mezőjét a következőképpen lehet lekérdezni:

```
const SFNode& pApp     = (SFNode&)pShape->getField("appearance");
```

5.4. Világmodellek felépítése a memóriában

Mindig a feladat nagysága és nehézsége határozza meg, hogy milyen adatszerkezetet építünk fel a memóriában. Ha az adatszerkezet tömör, akkor nagyobb valószínűséggel találhatók a kért adatok a gyorsítómemóriában. Ezért programunk annál gyorsabb lesz, minél kompaktabb adatstruktúrákat és minél kevesebb memóriát használ. Nem célszerű például az anyagok törésmutatóját, vagy a csúcsok normálvektorát tárolni, ha azokat a program nem használja. Ebben a fejezetben egy olyan saját adatszerkezetet építünk fel, amely jól illeszkedik az OpenGL (2.5.1. fejezet) vagy a DirectX (11. fejezet) képszintézishez. Egy sugárkövető algoritmushoz, egy animációtervező programhoz vagy egy CAD modellezőhöz más-más adatstruktúrákat használunk.

A világmodell felépítését egy VRMLViewer példaprogramon keresztül fogjuk illusztrálni. Az OpenVRML által a memóriában felépített színtérgráfot járjuk be, majd az adatokból egy saját színtér adatszerkezetet építünk fel, végül az OpenVRML adatszerkezetét eldobjuk. A konvertálás során azonban csak azokkal a csomópontokkal

foglalkozunk, amelyeket fontosnak ítélünk. A legfontosabb VRML elem az `IndexedFaceSet`, erre mindenképpen fel kell készülni.

Színterünk a kamerából (`camera`), 3D pontokból (`gVertices`), háromszögekből (`gPatches`) és anyagdefiníciókból (`gMaterials`) áll. Egy egyszerű VRML megjelenítő alkalmazásban elegendő az is, hogy a `Patch` objektum csak háromszöget tárol. Ebben az esetben a háromszögekre tesszellálást (3.7. fejezet) a beolvasáskor el kell végezni. A tömböket a *Standard Template Library* (*STL*) `vector` tárolójával valósítjuk meg. Az értelmezés során a transzformációk egymásba ágyazottan is előfordulhatnak. Ennek kezelésére az OpenGL-hez hasonlóan egy vermet (`gMatrixStack`) készítünk, amelyben transzformációs mátrixokat helyezünk el:

```
Camera                   camera;         // kamera
std::vector <Vector>     gVertices;      // csúcsok vektora
std::vector <Patch>      gPatches;       // felületelemek vektora
std::vector <Material>   gMaterials;     // anyagok vektora
std::stack<VrmlMatrix>   gMatrixStack;   // transzformációs verem
```

Az elemzés folyamán a transzformációs verem tetején (`gMatrixStack.top()`) található mátrixot használjuk. Egy transzformációs csomópont felismerése esetén a verembe egy új elemet teszünk:

```
//-----------------------------------------------------------------
void Reader::TransformBegin(Vrml97Node::Transform* pVrmlNodeTransform) {
//-----------------------------------------------------------------
   VrmlMatrix transformMx;  // új mátrix = veremtető * transzformáció
   pVrmlNodeTransform->getMatrix(transformMx);
   VrmlMatrix newMx = gMatrixStack.top().multLeft(transformMx);
   gMatrixStack.push(newMx);   // az új mátrix kerül a verem tetejére
}
```

A transzformációs csomópont feldolgozása után a verem tetején található transzformációt eldobjuk:

```
//-----------------------------------------------------------------
void Reader::TransformEnd(Vrml97Node::Transform* pVrmlNodeTransform) {
//-----------------------------------------------------------------
   gMatrixStack.pop();      // veremtető törlése
}
```

A `Camera`, a `Material` és a `Patch` osztály megvalósítása a következő:

```
//=================================================================
class Camera {
//=================================================================
public:
   Vector  eyep;              // pozíció
   Vector  lookp;             // hova néz
   Vector  updir;             // felfele irány
   float   viewdist;          // fókusztávolság

   float   fov, hfov, vfov;    // látószögek radiánban
   float   nearClip, farClip;  // közeli és távoli vágósik
   int     hres, vres;         // szélesség, magasság pixelben

   // kamera koordinátarendszer: X=jobbra, Y=le, Z=nézeti irány
   Vector  X, Y, Z;
   float   pixh, pixv;         // egy pixel szélessége, magassága

   Camera();
   void CompleteCamera();
};

//=================================================================
class Material { // anyagjellemzők
//=================================================================
public:
   Color diffuseColor;  // diffúz szín
};

//=================================================================
class Patch { // csak háromszögek
//=================================================================
public:
   Vector         *a, *b, *c;     // a három csúcspont
   Vector         normal;         // a síklap normálisa
   Vector         *Na, *Nb, *Nc;  // a csúcspontok normálvektorai
   Material       *pMaterial;     // anyagjellemző
public:
   void FinishPatch(void);
};
```

Elemzéskor először a kamerával (`HandleCamera()`), majd a színtér többi részével (`HandleNodes()`) foglalkozunk, végül töröljük az OpenVRML színtérgráfot.

```
   gVertices.clear();              // adatszerkezetek inicializálása
   gPatches.clear();
   gMaterials.clear();

   VrmlMatrix  identityMx;         // alapértelmezett eset az identitás mátrix
   gMatrixStack.empty();           // mátrix verem törlése
   gMatrixStack.push(identityMx);  // és az identitással feltöltése

   Viewpoint* pView = vrmlScene->bindableViewpointTop(); //kamera adat
   HandleCamera(pView);            // kamera feldolgozása
   HandleNodes(rootNodes);         // geometria feldolgozása
   delete vrmlScene;               // színtérgráf törlése
```

A `HandleNodes()` függvény bejárja a színteret. Ha egy csoport (`Group`) típusú csomópontot dolgoz fel, akkor egy rekurzív függvényhívással a gráfbejárást a gyermekekre is elvégzi. Ha ez a `Group` csomópont egyben transzformációs csomópont is, akkor a transzformációs verem egy új elemmel bővül, és az akkumulált transzformáció kerül a verem tetejére. Ha éppen egy `Shape` csomópontot látogattunk meg, akkor meghívjuk a `HandleShape()` metódust.

```
//-----------------------------------------------------------------
void Reader::HandleNodes(const MFNode& nodes) {
//-----------------------------------------------------------------
   for(size_t i = 0; i < nodes.getLength(); i++) {
      Node* pNode = nodes.getElement(i).get();
      if (pNode->toGroup() != NULL) {
         Vrml97Node::Group* pGroup   = pNode->toGroup();

         if (pNode->nodeType.id == std::string("Transform"))
             TransformBegin(dynamic_cast<Vrml97Node::Transform*>(pGroup));
         HandleNodes(pVrmlNodeGroup->getChildren());
         if (pNode->nodeType.id == std::string("Transform"))
         TransformEnd(dynamic_cast<Vrml97Node::Transform*>(pGroup));
      } else {
         if (pNode->nodeType.id == std::string("Shape"))
             HandleShape(dynamic_cast<Vrml97Node::Shape*>(pNode));
         ...
         ... // itt értelmezzük még a számunkra fontos VRML elemeket
         ...
      }   // if group
   }   // for
}

//-----------------------------------------------------------------
void Reader::HandleShape(Vrml97Node::Shape* pShape) {
//-----------------------------------------------------------------
   const SFNode& pApp      = (SFNode&)pShape->getField("appearance");
   const SFNode& pGeom     = (SFNode&)pShape->getField("geometry");
   const Vrml97Node::Appearance* pApperance =
        dynamic_cast<Vrml97Node::Appearance*>(pApp.get().get());
   HandleMaterial(pApperance);     // anyagok kezelése

   Vrml97Node::AbstractGeometry* pGeometry =   // geometria kezelése
        dynamic_cast<Vrml97Node::AbstractGeometry*>(pGeom.get().get());
   if (pGeometry->nodeType.id == std::string("IndexedFaceSet"))
     HandleIFaceSet(dynamic_cast<Vrml97Node::IndexedFaceSet*>(pGeometry));
}
```

A `Shape` csomópont `appearance` mezőjét a `HandleMaterial()` dolgozza fel. Ha a `geometry` mező éppen `IndexedFaceSet`, akkor a `HandleIFaceSet()` metódust hívjuk:

```
//-----------------------------------------------------------------
void Reader::HandleMaterial(const Vrml97Node::Appearance* pApperance){
//-----------------------------------------------------------------
   if (!pApperance->getMaterial().get()) throw "Nincs anyaga a csomópontnak!";

   MaterialNode* pMaterial = pApperance->getMaterial().get()->toMaterial();
   const SFColor& color    = pMaterial->getDiffuseColor();

   Material material; // egy új anyag felvétele
   material.diffuseColor.Set(color.getR(), color.getG(), color.getB());
   gMaterials.push_back(material);
}
```

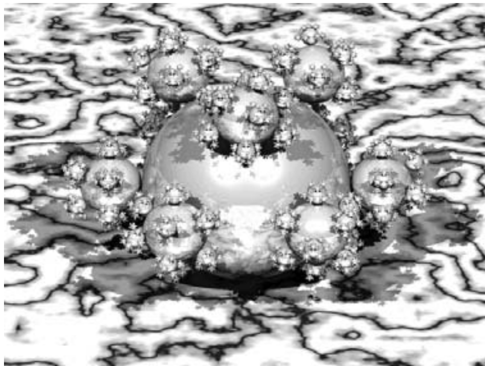
A `HandleIFaceSet()` függvény először az `IndexedFaceSet` példány `coord` mezőjében található csúcspontokat teszi a `gVertices` tömbbe. A poligonok csúcspontjait a `coordIndex` mező tartalmazza, amely egy indexekből álló vektor. A „-1”-es index jelzi, hogy a poligon itt befejeződött. Az így kapott (lehetőleg konvex) sokszöglapokat háromszögekre tesszelláljuk.

```
//-----------------------------------------------------------------
void Reader::HandleIFaceSet(const Vrml97Node::IndexedFaceSet* pIFaceSet) {
//-----------------------------------------------------------------
   float   transformedVertex[3];              // transzformált csúcspont
   VrmlMatrix& trMatrix = gMatrixStack.top(); // aktuális transzformáció

   int vertexIndexBefore       = gVertices.size();
   const SFNode&   pCoordinate = (SFNode&)pIFaceSet->getField("coord");
   const MFVec3f& point = (MFVec3f&)pCoordinate.get().get()->getField("point");

   int             nCoord      = point.getLength();
   const float*    pCoord      = &point.getElement(0)[0];
   for(int i = 0; i < nCoord; i++) { // csúcspontok feldolgozása
      const float* pCoordItem = pCoord + i*3;
      gTransformMx.multMatrixVec(pCoordItem, transformedVertex);
      gVertices.push_back(Vector(transformedVertex[0],
                        transformedVertex[1], transformedVertex[2]));
   }

   const MFInt32&  coordIndex = (MFInt32&)pIFaceSet->getField("coordIndex");
   int nCoordIndex            = coordIndex.getLength();
   int poligonStartIndex      = 0; // az coordIndex feldolgozásában itt tartunk
   for(i = 0; i < nCoordIndex; i++) { // a csúcspont koordinátákon megy végig
      if (coordIndex.getElement(i) != -1) continue;// -1 jelzi a poligon végét
      int nTriangles = i-poligonStartIndex-2;// ennyi háromszögre bontható
      for(int k = 0; k < nTriangles; k++) {  // háromszögekre tesszellálás
         Patch patch;
         patch.a = &gVertices[coordIndex.getElement(poligonStartIndex)];
         patch.b = &gVertices[coordIndex.getElement(poligonStartIndex+k+1)];
         patch.c = &gVertices[coordIndex.getElement(poligonStartIndex+k+2)];
         patch.pMaterial = &gMaterials[gMaterials.size() - 1];
         gPatches.push_back(patch);
      }
      poligonStartIndex = i + 1;
   }  // for
}
```

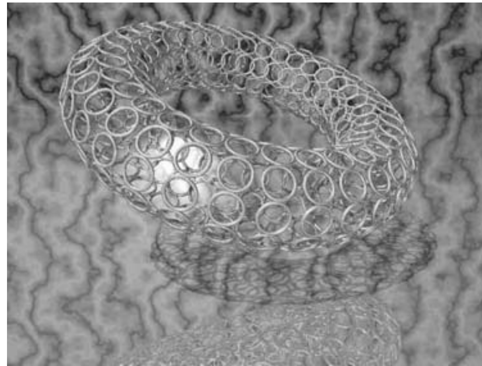
6. fejezet

Sugárkövetés

A *sugárkövetés*[1] (*raytracing*) születése az 1980-as évek elejére tehető. Ez az algoritmus — szemben az inkrementális képszintézissel (lásd 7. fejezet) — tükrök, átlátszó illetve áttetsző felületek, valamint árnyékok automatikus megjelenítésére is képes. „Életének” 20 éve alatt a sugárkövetés számos fejlesztésen és finomításon ment keresztül. A különböző optimalizációs technikák a kép minőségét lényegesen nem javították, az amúgy eléggé időigényes képszintézis folyamatot jelentősen felgyorsították.



500 000 gömbből álló fraktális test



Tórusz arany gyűrűkből

6.1. ábra. *Sugárkövetéssel készített képek (Henrik W. Jensen)*

A sugárkövetés a képernyő pixeleire egymástól függetlenül oldja meg a takarási és árnyalási feladatokat. A módszer elnevezése abból ered, hogy az algoritmus megpróbálja a színtérben a fény terjedését, a fénysugarak és a felületek ütközését szimulálni.

[1] A sugárkövetés első részletes összefoglalóját Andrew S. Glassner [48] készítette 1987-ben. Megjelenése óta már többször átdolgozták, ezért még mindig aktuális.

A sugárkövetés elnevezés egy kicsit megtévesztő, ugyanis azt sugallja, hogy a fotonok követése a fényforrásnál kezdődik, és a szemnél fejeződik be. Ez a módszer azonban — tekintve, hogy például egy izzó fényének csak egy töredéke jut a szembe — rengeteg felesleges számítást igényelne. Tehát csak azokkal a fotonokkal érdemes foglalkozni, amelyek ténylegesen a szembe jutnak. Ezért a „fotonkövetés” a szemből indul, és innen — mivel a fény útja megfordítható — a fény által megtett utat rekurzívan visszafelé követve jutunk el a fényforrásig.

6.2. ábra. *Pov-Ray sugárkövető programmal készített képek*

Ha a kedves Olvasó egy professzionális és ingyenes sugárkövető programmal szeretne a 6.2. ábrához hasonló képeket készíteni, akkor a *http://www.povray.org* honlapra érdemes ellátogatnia, ahonnan a Pov-Ray[2] (*Persistence of Vision Raytracer*) programot töltheti le.

6.1. Az illuminációs modell egyszerűsítése

A *sugárkövetés* a lokális illuminációs algoritmusokhoz hasonlóan, de kevésbé durván egyszerűsíti az árnyalási egyenletet (lásd 8.2. fejezet). A lehetséges visszaverődésekből és törésekből elkülöníti a geometriai optikának megfelelő ideális (úgynevezett *koherens*) eseteket, és csak ezekre hajlandó a többszörös visszaverődések és törések követésére. A többi, úgynevezett *inkoherens komponens*re viszont — a lokális illuminációs módszerekhez hasonlóan — elhanyagolja az indirekt megvilágítást és csak az absztrakt fényforrások direkt hatását veszi figyelembe.

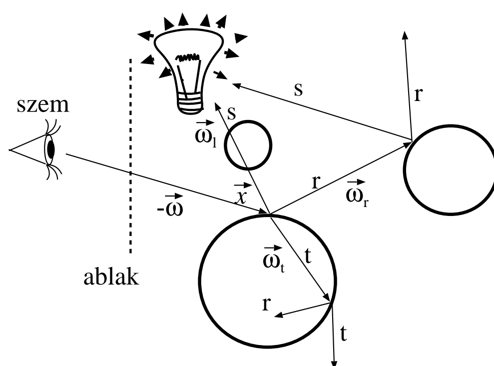
[2] a Pov-Ray forráskódja is ingyenesen elérhető

A 4.8.8. fejezetben már volt szó az *árnyalási egyenlet* egyszerűsített alakjáról, amelyet itt kicsit átdolgozva megismétlünk:

$$L(\vec{x},\vec{\omega}) = L^e(\vec{x},\vec{\omega}) + k_a \cdot L^a + \sum_l f_r(\vec{\omega}'_l,\vec{x},\vec{\omega}) \cdot \cos\theta'_l \cdot L^{\text{in}}(\vec{x},\vec{\omega}'_l) +$$

$$k_r \cdot L^{\text{in}}(\vec{x},\vec{\omega}_r) + k_t \cdot L^{\text{in}}(\vec{x},\vec{\omega}_t), \tag{6.1}$$

ahol $\vec{\omega}_r$ az $\vec{\omega}$ tüköriránya, $\vec{\omega}_t$ a fénytörésnek megfelelő irány, $f_r(\vec{\omega}'_l,\vec{x},\vec{\omega})$ a diffúz és a spekuláris visszaverődést jellemző BRDF, $L^{\text{in}}(\vec{x},\vec{\omega}'_l)$ pedig az l-edik absztrakt fényforrásból, az $\vec{\omega}'_l$ irányból az $\vec{x}$ pontba érkező sugársűrűség (radiancia). A $k_a \cdot L^a$ a 4.6.1. fejezetben bevezetett *ambiens tag*. A k_r a tükör, a k_t pedig a fénytörés visszaverődési hányadosa.



6.3. ábra. *Rekurzív sugárkövetés*

Egy pixel színének számításához mindenekelőtt a pixelben látható felületi pontot kell megkeresnünk. Ehhez először a szempozícióból a pixel középpontján keresztül egy félegyenest, úgynevezett sugarat indítunk. A sugár és a felületek metszéspontja az illuminációs képletben (6.1. egyenlet) szereplő $\vec{x}$ pont, az $\vec{x}$-ből a szembe mutató irányvektor pedig az $\vec{\omega}$ lesz. Ezekkel a paraméterekkel kiértékeljük az illuminációs képletet, és a pixelt ennek megfelelően kiszínezzük.

Az illuminációs képlet kiszámításához a következőket kell elvégezni:

- Az $\vec{x}$ felületi pont és $\vec{\omega}$ nézeti irány ismeretében kiértékeljük a saját sugárzást és az ambiens fényvisszaverődést ($L^e(\vec{x},\vec{\omega}) + k_a \cdot L^a$).

- A tükörirányból érkező fény visszaveréséhez kiszámítjuk a tükörirányt, és meghatározzuk az innen érkező sugársűrűséget ($L^{\text{in}}(\vec{x},\vec{\omega}_r)$), amelyet a látható színben k_r súllyal veszünk figyelembe. Vegyük észre, hogy a tükörirányból érkező sugársűrűség kiszámítása pontosan ugyanarra a feladatra vezet, mint amelyet a pixel

színének a számításakor oldunk meg, csupán a vizsgált irányt most nem a szem és a pixel középpont, hanem a vizsgált $\vec{x}$ pont és a tükörirány határozza meg! Az implementáció szintjén ebből nyilván egy rekurzív program lesz.

- A törési irányból érkező fény töréséhez — szintén rekurzív módon — egy új sugarat indítunk a törési irányba, majd az onnan visszakapott sugársűrűséget ($L^{\text{in}}(\vec{x}, \vec{\omega}_t)$) a k_t tényezővel megszorozzuk.

- Az inkoherens visszaverődések kiszámításához minden egyes fényforrásról eldöntjük, hogy az az adott pontból látszik-e vagy sem. A képen így *árnyék*ok is megjelenhetnek. Ha tehát az l. pontszerű fényforrás teljesítménye Φ_l, pozíciója pedig $\vec{y}_l$, akkor a beérkező sugársűrűség:

$$L^{\text{in}}(\vec{x}, \vec{\omega}'_l) = v(\vec{x}, \vec{y}_l) \cdot \frac{\Phi_l}{4\pi|\vec{x} - \vec{y}_l|^2},$$

  ahol a $v(\vec{x}, \vec{y})$ a *láthatósági indikátor*, amely azt mutatja meg, hogy az $\vec{x}$ pontból látható-e ($v = 1$) a fényforrás, vagy sem ($v = 0$). Amennyiben a fényforrás és a pont között átlátszó vagy áttetsző objektumok vannak, a v 0 és 1 közötti értéket is felvehet.

  A láthatósági indikátor értelmezése miatt a fényforrás felé tartó *árnyék sugár* (*shadow ray*) és a színtér metszéspontjának számításakor elegendő csak a fényforrásig vizsgálni a geometriai elemeket, az ennél távolabb levő objektumokat már nem kell figyelembe venni.

  A láthatósági indikátor előállításához tehát első lépésben egy árnyék sugarat indítunk az $\vec{x}$ pontból a fényforrás felé, majd a metszett objektumok k_t átlátszósági tényezőit összeszorozva meghatározzuk v értékét. Az átlátszósági tényezők összeszorzásának mellőzése esetén az átlátszó objektumok is ugyanolyan árnyékot vetnek, mint az átlátszatlanok. Valójában ilyenkor a fény törését is figyelembe kellene venni, de ez meglehetősen bonyolult lenne, ezért nagyvonalúan eltekintünk tőle.

Az illuminációs képlet paraméterei elvileg hullámhossztól függőek, tehát a sugár által kiválasztott felület sugársűrűségét minden reprezentatív hullámhosszon (R, G, B) tovább kell adnunk.

A sugárkövető programunk az egyes pixelek színét egymás után és egymástól függetlenül számítja ki:

```
for (minden p pixelre) {
    r = szemből a pixel középpontjába mutató sugár;
    pixel színe = Trace(r, 0);
}
```

A **Trace**(r, d) szubrutin az r sugár irányából érkező sugársűrűséget határozza meg rekurzív módon. A d változó a rekurzió mélységét tartalmazza:

```
Color Trace(r, d) {
    if (d > d_max) return L^a;                                   // rekurzió korlátozása
    (q, x⃗) = Intersect(r);          // q: sugárral eltalált objektum, x⃗: felületi pont
    if (nincs metszéspont) return L^a;                           // saját emisszió + ambiens
    ω⃗ = r irányvektora;                                          // direkt megvilágítás
    c = L^e_q(x⃗, ω⃗) + k_a · L^a;
    for (minden l. fényforrásra) {
        r_s = x⃗-ből induló, y⃗_l felé mutató sugár;                // árnyék sugár
        (q_s, x⃗_s) = Intersect(r_s);
        if (nincs metszéspont vagy |x⃗_s − x⃗| > |y⃗_l − x⃗|)   // a fényforrás nem takart
            c += f_r(ω⃗'_l, x⃗, ω⃗) · cos θ'_l · Φ_l/|x⃗ − y⃗_l|^2/4/π;
    }
    if (k_r(x⃗) > 0) {                          // indirekt megvilágítás a tükörirányból
        r_r = az r tükörirányába mutató sugár;
        c += k_r(x⃗)·Trace(r_r, d + 1);
    }
    if (k_t(x⃗) > 0) {                          // indirekt megvilágítás a törési irányból
        r_t = az r törési irányába mutató sugár;
        c += k_t(x⃗)· Trace(r_t, d + 1);
    }
    return c;
}
```

A szubrutin kezdetén a rekurzió mélységének korlátozására egyrészt azért van szükség, hogy a tükörszobában fellépő végtelen rekurziót elkerüljük, másrészt pedig azért, hogy az elhanyagolható sokadik visszaverődések kiszámítására ne pazaroljuk drága időnket. Az algoritmus sebessége szempontjából kritikus pont az `Intersect()` függvény, amely egy sugár és a színtér metszéspontját számítja ki.

6.2. A tükör- és törési irányok kiszámítása

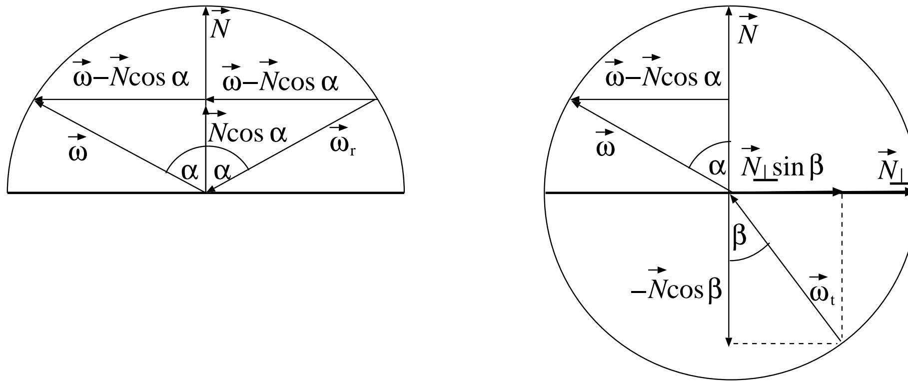
A *tükörirányt* a 6.4. ábra alapján a következőképpen számíthatjuk ki:

$$\vec{\omega}_r = (\vec{\omega} - \cos\alpha \cdot \vec{N}) - \cos\alpha \cdot \vec{N} = \vec{\omega} - 2\cos\alpha \cdot \vec{N}. \tag{6.2}$$

ahol α a beesési szög, melynek koszinusza a $\cos\alpha = (\vec{N} \cdot \vec{\omega})$ skalárszorzattal állítható elő, feltéve, hogy $\vec{N}$ és $\vec{\omega}$ egységvektorok.

A *törési irány* meghatározása egy kicsit bonyolultabb. Ha a törés szöge β, akkor a törés irányába mutató egységvektor:

$$-\vec{\omega}_t = -\cos\beta \cdot \vec{N} + \sin\beta \cdot \vec{N}_\perp.$$



6.4. ábra. *A tükörirány és a törési irány kiszámítása*

ahol $\vec{N}_\perp$ a normálvektorra merőleges, a normálvektor és a beesési vektor síkjába eső egységvektor:

$$\vec{N}_\perp = \frac{\cos\alpha \cdot \vec{N} - \vec{\omega}}{|\cos\alpha \cdot \vec{N} - \vec{\omega}|} = \frac{\cos\alpha \cdot \vec{N} - \vec{\omega}}{\sin\alpha}.$$

Ezt behelyettesítve és felhasználva a *Snellius – Descartes törvény*t (4.8.3. fejezet), miszerint

$$\frac{\sin\alpha}{\sin\beta} = \nu$$

(ν a relatív törésmutató), a következő összefüggéshez[3] jutunk:

$$\vec{\omega}_t = \cos\beta \cdot \vec{N} - \frac{\sin\beta}{\sin\alpha} \cdot (\cos\alpha \cdot \vec{N} - \vec{\omega}) = \frac{\vec{\omega}}{\nu} - \left(\frac{\cos\alpha}{\nu} - \cos\beta\right) \cdot \vec{N} =$$

$$\frac{\vec{\omega}}{\nu} - \left(\frac{\cos\alpha}{\nu} - \sqrt{1 - \sin^2\beta}\right) \cdot \vec{N} = \frac{\vec{\omega}}{\nu} - \left(\frac{\cos\alpha}{\nu} - \sqrt{1 - \frac{(1 - \cos^2\alpha)}{\nu^2}}\right) \cdot \vec{N}.$$

A képletben szereplő ν relatív törésmutató értéke attól függ, hogy éppen belépünk-e az anyagba, vagy kilépünk belőle (a két esetben ezek az értékek egymásnak reciprokai). Az aktuális helyzetet a sugárirány és a felületi normális által bezárt szög mondja meg. A programban elegendő meghatározni a fenti vektorok skalárszorzatának előjelét.

Ha a négyzetgyök jel alatti tag negatív, akkor a *teljes visszaverődés* esete áll fenn, tehát az optikailag sűrűbb anyagból a fény nem tud kilépni a ritkább anyagba. Ilyenkor a tört fénymennyiség is a visszaverődéshez adódik hozzá[4].

[3] a képletet koszinuszos alakban adjuk meg, ennek kiszámítása ugyanis (szemben a szinusszal) skalárszorzattal gyorsan elvégezhető

[4] így működik például az üvegszálas jeltovábbító kábel

6.3. Metszéspontszámítás felületekre

A sugárkövetés legfontosabb részfeladata az, hogy meghatározza, hogy a sugár milyen felületet, és ezen belül melyik felületi pontot találja el. Erre a célra egy `Intersect()` függvényt készítünk , amely az r sugár és a legközelebbi felület metszéspontját keresi meg! A gyakorlati tapasztalatok szerint a sugárkövető programunk a futás során az idő 65–90%-át az `Intersect()` rutinban tölti, ezért ennek hatékony implementációja a gyors sugárkövetés kulcsa. A sugarat általában a következő egyenlettel adjuk meg:

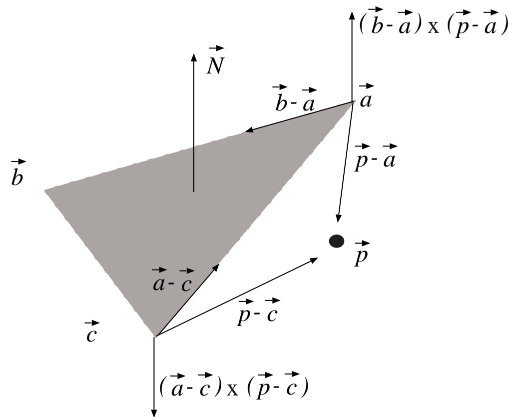
$$\vec{r}(t) = \vec{s} + t \cdot \vec{d}, \qquad (t > 0), \tag{6.3}$$

ahol $\vec{s}$ a kezdőpont, $\vec{d} = -\vec{\omega}$ a sugár iránya, a t *sugárparaméter* pedig a kezdőponttól való távolságot jelenti. Ha a t negatív, akkor a metszéspont a szem mögött helyezkedik el. A következőkben áttekintjük, hogy a különböző primitív típusokra hogyan számíthatjuk ki a sugár és a felület metszéspontját.

6.3.1. Háromszögek metszése

A *háromszög*ek metszése a 3.4.1. fejezet alapján két lépésben történik. Először előállítjuk a sugár és a háromszög síkjának metszéspontját, majd eldöntjük, hogy a metszéspont a háromszög belsejében van-e. Legyen a háromszög három csúcsa $\vec{a}$, $\vec{b}$ és $\vec{c}$! Ekkor a háromszög síkjának normálvektora $\vec{N} = (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})$, egy helyvektora pedig $\vec{a}$, tehát a sík $\vec{p}$ pontjai kielégítik a sík normálvektoros egyenletét:

$$\vec{N} \cdot (\vec{p} - \vec{a}) = 0. \tag{6.4}$$



6.5. ábra. *A háromszög metszés szemléltetése*

A sugár és a sík közös pontját megkaphatjuk, ha a sugár egyenletét (6.3. egyenlet) behelyettesítjük a sík egyenletébe (6.4. egyenlet), majd a keletkező egyenletet megoldjuk az ismeretlen t paraméterre. Ha a kapott t^* érték pozitív, akkor visszahelyettesítjük a sugár egyenletébe, ha viszont negatív, akkor a metszéspont a sugár kezdőpontja mögött helyezkedik el, így nem érvényes. A sík metszése után azt kell ellenőriznünk, hogy a kapott $\vec{p}$ pont vajon a háromszögön kívül vagy belül helyezkedik-e el. A $\vec{p}$ metszéspont akkor van a háromszögön belül, ha a háromszög mind a három oldalegyeneséhez viszonyítva a háromszöget tartalmazó félsíkban van (3.4.1. fejezet):

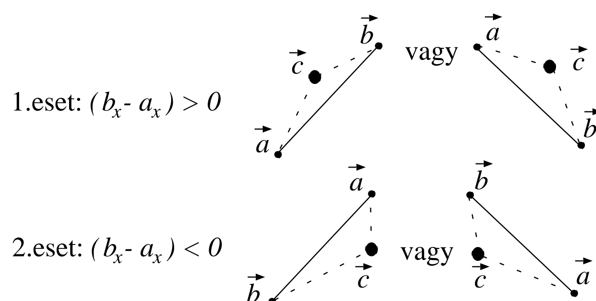
$$\begin{aligned} ((\vec{b}-\vec{a})\times(\vec{p}-\vec{a}))\cdot\vec{N} &\geq 0, \\ ((\vec{c}-\vec{b})\times(\vec{p}-\vec{b}))\cdot\vec{N} &\geq 0, \\ ((\vec{a}-\vec{c})\times(\vec{p}-\vec{c}))\cdot\vec{N} &\geq 0. \end{aligned} \quad (6.5)$$

A 6.5. ábra azt az esetet illusztrálja, amikor a síkon levő $\vec{p}$ pont az $\vec{ab}$ és $\vec{bc}$ egyenesektől balra, a $\vec{ca}$ egyenestől pedig jobbra helyezkedik el, azaz nincs bent a háromszög belsejében. Az ábrán berajzolt vektorok hossza a jobb áttekinthetőség végett nem pontos, irányuk azonban igen.

A 6.5. egyenlőtlenségrendszer kiértékelése (mivel a skaláris és vektoriális szorzatok aránylag sok szorzást tartalmaznak) elég számításigényes feladat. Ha szeretnénk gyorsítani a sugárkövető algoritmusunkat, akkor háromdimenzió helyett érdemesebb kétdimenzióban dolgozni. Miután megvan a sík és az egyenes metszéspontja, vetítsük le a pontot, és vele együtt a háromszöget valamelyik koordinátasíkra, és ezen a síkon végezzük el a háromszög három oldalára a tartalmazás vizsgálatot! Nem lehet azonban a projekció síkját mindig ugyanúgy kijelölni, hiszen például az XZ síkban elhelyezkedő háromszög YZ síkbeli képe csak egy vonal, amivel sajnos nem lehet tovább dolgozni. Pontosabb numerikus számítások miatt érdemes azt a síkot választani, amelyiken a vetített háromszögnek a legnagyobb a területe. Ezt a síkot a *domináns sík*nak nevezzük. A háromszög domináns síkjának meghatározása a sík normálvektorának vizsgálatával kezdődik. Mivel a normálvektor nem változik, ezért ezt egy előfeldolgozási lépésben is kiszámíthatjuk. A következő kis rutin megadja, hogy az `n` normálvektor `x`, `y` vagy `z` irányú komponensei közül melyik (`X_DOMINANT_NORMAL`, `Y_DOMINANT_NORMAL`, `Z_DOMINANT_NORMAL`) a legnagyobb.

```
//-----------------------------------------------------------------
DominantType GetDominance(Vector n) {
//-----------------------------------------------------------------
    if (fabs(n.x) > fabs(n.y)) {
        if (fabs(n.x) > fabs(n.z))   return X_DOMINANT_NORMAL;
                                else return Z_DOMINANT_NORMAL;
    } else {
        if (fabs(n.y) > fabs(n.z))   return Y_DOMINANT_NORMAL;
                                else return Z_DOMINANT_NORMAL;
    }
}
```

Ha a normálvektor például Z domináns, akkor a háromszög domináns síkja az XY sík. Az egyszerűség kedvéért a továbbiakban csak ezen a síkon dolgozunk.



6.6. ábra. *A gyors háromszög metsző algoritmus*

A gyors algoritmusunk két részből áll. Egy előfeldolgozási lépésben átalakítjuk a csúcsok sorrendjét úgy, hogy $\vec{a}$-ból $\vec{b}$-be haladva a $\vec{c}$ pont mindig a bal oldalon helyezkedjen el. Ehhez először vizsgáljuk meg az XY síkra vetített $\vec{ab}$ egyenes egyenletét:

$$\frac{b_y - a_y}{b_x - a_x} \cdot (x - b_x) + b_y = y.$$

A 6.6. ábra segítségével értelmezzük a fenti egyenletet. A $\vec{c}$ akkor van az egyenes bal oldalán, ha $x = c_x$-nél c_y az egyenes felett van:

$$\frac{b_y - a_y}{b_x - a_x} \cdot (c_x - b_x) + b_y < c_y.$$

Mindkét oldalt $(b_x - a_x)$-szel szorozva:

$$(b_y - a_y) \cdot (c_x - b_x) < (c_y - b_y) \cdot (b_x - a_x).$$

A második esetben a meredekség nevezője negatív. A $\vec{c}$ akkor van az egyenes bal oldalán, ha $x = c_x$-nél c_y az egyenes alatt van:

$$\frac{b_y - a_y}{b_x - a_x} \cdot (c_x - b_x) + b_y > c_y.$$

A negatív nevezővel, a $(b_x - a_x)$-szel való szorzás miatt a relációs jel megfordul:

$$(b_y - a_y) \cdot (c_x - b_x) < (c_y - b_y) \cdot (b_x - a_x),$$

azaz mindkét esetben ugyanazt a feltételt kaptuk. Ha ez a feltétel nem teljesül, akkor $\vec{c}$ nem az $\vec{ab}$ egyenes bal oldalán, hanem a jobb oldalán helyezkedik el. Ez pedig azt

jelenti, hogy $\vec{c}$ a $\vec{ba}$ egyenes bal oldalán található, tehát az $\vec{a}$ és $\vec{b}$ sorrendjének cseréjével biztosítható, hogy $\vec{c}$ az $\vec{ab}$ egyenes bal oldalán tartózkodjon. Fontos észrevenni, hogy ebből következik az is, hogy az $\vec{a}$ a $\vec{bc}$ egyenes, valamint a $\vec{b}$ a $\vec{ca}$ egyenes bal oldalán helyezkedik el.

A módszer második része már a metszéspontszámításhoz kapcsolódik. Itt lényegében ugyanazt kell megismételnünk, mint az előbb. A különbség egyrészt annyi, hogy most nem a $\vec{c}$ csúcsot, hanem a $\vec{p}$ pontot kell megvizsgálni. Másrészt a háromszög mindhárom oldalára el kell végezni a vizsgálatot. A 6.5. egyenlőtlenségekkel ekvivalens vizsgálatok kódja a következő:

```
if (Z_DOMINANT_NORMAL) {
    px = ray->origin.x + t * ray->dir.x;
    py = ray->origin.y + t * ray->dir.y;

    if ((by - ay) * (px - bx) > (py - by) * (bx - ax)) return false;
    if ((cy - by) * (px - cx) > (py - cy) * (cx - bx)) return false;
    if ((ay - cy) * (px - ax) > (py - ay) * (ax - cx)) return false;
    return true;
}
```

Méréseink alapján a kétdimenziós módszer kétszer olyan gyors, mint a háromdimenziós.

6.3.2. Implicit felületek metszése

Vegyünk először példaként egy egyszerű felületet, egy gömböt! A síkmetszéshez hasonlóan egy *gömb*re úgy kereshetjük a metszéspontot, ha a sugár egyenletét behelyettesítjük a gömb egyenletébe:

$$|(\vec{s}+t\cdot\vec{d})-\vec{c}|^2 = R^2,$$

majd megoldjuk t-re az ebből adódó

$$(\vec{d})^2\cdot t^2 + 2\cdot\vec{d}\cdot(\vec{s}-\vec{c})\cdot t + (\vec{s}-\vec{c})^2 - R^2 = 0$$

egyenletet, ahol $(\vec{d})^2 = (\vec{d}\cdot\vec{d})$ a skalárszorzást jelenti. Csak a pozitív valós gyökök érdekelnek bennünket, ha ilyen nem létezik, az azt jelenti, hogy a sugár nem metszi a gömböt. Ez a módszer bármely más kvadratikus felületre használható. A kvadratikus felületeket különösen azért szeretjük a sugárkövetésben, mert a metszéspontszámítás másodfokú egyenletre vezet, amelyet a megoldóképlet alkalmazásával könnyen megoldhatunk.

Általánosan egy $F(x,y,z) = 0$ implicit egyenlettel definiált felület metszéséhez a sugáregyenletnek az implicit egyenletbe történő behelyettesítésével előállított

$$f(t) = F(s_x + d_x\cdot t, s_y + d_y\cdot t, s_z + d_z\cdot t) = 0$$

nemlineáris egyenletet kell megoldani, amelyhez numerikus gyökkereső eljárásokat használhatunk [123].

6.3.3. Paraméteres felületek metszése

Az $\vec{r} = \vec{r}(u,v)$, $(u,v \in [0,1])$ paraméteres felület és a sugár metszéspontját úgy kereshetjük meg, hogy először az ismeretlen u,v,t paraméterekre megoldjuk a

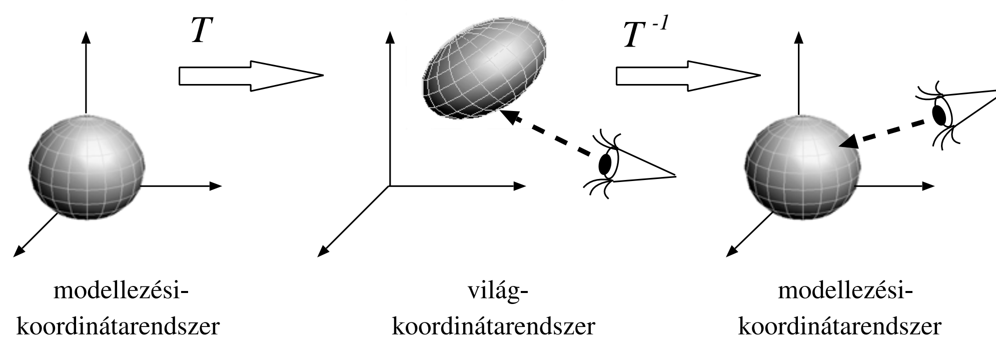
$$\vec{r}(u,v) = \vec{s} + t \cdot \vec{d}$$

háromváltozós, nemlineáris egyenletrendszert, majd ellenőrizzük, hogy a t pozitív, és az u,v paraméterek valóban a $[0,1]$ tartomány belsejében vannak-e.

A gyakorlatban a nemlineáris egyenletrendszerek megoldása helyett inkább azt az utat követjük, hogy a felületeket poligonhálóval közelítjük (emlékezzünk vissza, hogy ez a *tesszelláció*s folyamat különösen egyszerű paraméteres felületekre), majd a poligonhálót próbáljuk a sugárral elmetszeni. Ha sikerül metszéspontot találni, az eredményt úgy lehet pontosítani, hogy a metszéspont környezetének megfelelő paramétertartományban egy finomabb tesszellációt készítünk, és a metszéspontszámítást újra elvégezzük.

6.3.4. Transzformált objektumok metszése

Az előző fejezetben ismertetett módszer ellenére, a sugárkövetés nem igényel tesszellációt, azaz az objektumokat nem kell poligonhálóval közelíteni, mégis implicit módon elvégzi a nézeti transzformációs, vágási, vetítési és takarási feladatokat.



6.7. ábra. *Transzformált objektumok metszése*

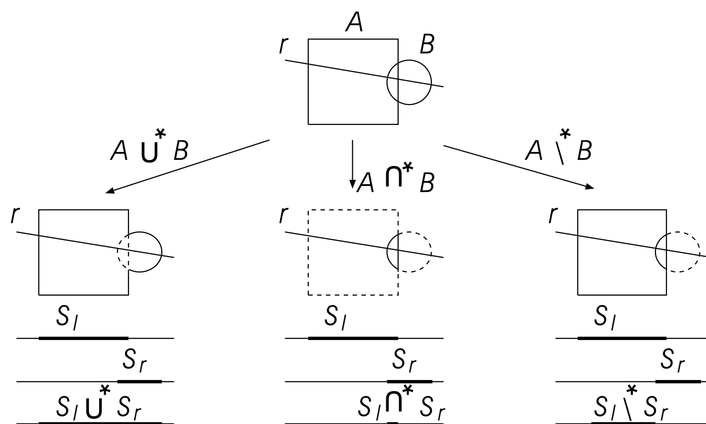
Ha egy objektumot közvetlenül a világ-koordinátarendszerben írunk le, akkor — mivel a szemből indított sugár is ebben a koordinátarendszerben található — a metszéspont egyszerűen meghatározható. Ha viszont az objektum a különálló modellezési-koordinátarendszerben adott, és innét egy **T** modellezési transzformáció viszi át a világ-koordinátarendszerbe, akkor a feladat már nem is olyan egyszerű. Ez ugyanis ahhoz a problémához vezet, hogy hogyan is kell transzformálni például egy gömböt ellipszoiddá. Szerencsére ezt a kérdést megkerülhetjük, ha nem az objektumot, hanem — a

$\mathbf{T}^{-1}$ inverztranszformációval — a sugarat transzformáljuk. Ezek után a modellezési-koordinátarendszerben meghatározzuk a transzformált sugár és az objektum metszetét, majd a $\mathbf{T}$ alkalmazásával a világ-koordinátarendszerbe képezzük a metszéspontokat (6.7. ábra).

6.3.5. CSG modellek metszése

A *konstruktív tömörtest geometria* (*CSG*) a modelleket egyszerű primitívekből (kocka, henger, kúp, gömb stb.) reguláris halmazműveletek ($\cup^*, \cap^*, \setminus^*$) segítségével állítja elő. Egy objektumot egy bináris fa adatstruktúra ír le, amelyben a levelek a primitíveket azonosítják, a belső csomópontok pedig a két gyermeken végrehajtandó geometriai transzformációkat, és az eredmény előállításához szükséges halmazműveletet. A fa gyökere magát az objektumot képviseli, a többi csomópont pedig a felépítéshez szükséges egyszerűbb testeket.

Ha a fa egyetlen levélből állna, akkor a sugárkövetés könnyen megbirkózna a sugár és az objektum közös pontjainak azonosításával. Tegyük fel, hogy a sugár és a primitív felületelemeinek metszéspontjai a $t_1 \leq t_2 \ldots \leq t_{2k}$ sugárparamétereknél találhatók. Ekkor a sugár az $(\vec{s}+t_1 \cdot \vec{d},\ \vec{s}+t_2 \cdot \vec{d})$, ..., $(\vec{s}+t_{2k-1} \cdot \vec{d},\ \vec{s}+t_{2k} \cdot \vec{d})$ pontpárok közötti szakaszokon (*ray-span*, úgynevezett belső szakaszok) a primitív belsejében, egyébként a primitíven kívül halad. A szemhez legközelebbi metszéspontot úgy kaphatjuk meg, hogy ezen szakaszvégpontok közül kiválasztjuk a legkisebb pozitív paraméterűt. Ha a paraméter szerinti rendezés után a pont paramétere páratlan, a szem az objektumon kívül van, egyébként pedig az objektum belsejében ülve nézünk ki a világba. Az esetleges geometriai transzformációkat a 6.3.4. fejezetben javasolt megoldással kezelhetjük.



6.8. ábra. *Belső szakaszok és a kombinálásuk*

Most tegyük fel, hogy a sugárral nem csupán egy primitív objektumot, hanem egy CSG fával leírt struktúrát kell elmetszeni! A fa csúcsán egy halmazművelet található, amely a két gyermekobjektumból előállítja a végeredményt. Ha a gyermekobjektumokra sikerülne előállítani a belső szakaszokat, akkor abból az összetett objektumra vonatkozó belső szakaszokat úgy kaphatjuk meg, hogy a szakaszok által kijelölt ponthalmazra végrehajtjuk az összetett objektumot kialakító halmazműveletet. Emlékezzünk vissza, hogy a CSG modellezés regularizált halmazműveleteket használ, hogy elkerülje a háromnál alacsonyabb dimenziójú elfajulásokat. Tehát, ha a metszet vagy a különbség eredményeképpen különálló pontok keletkeznek, azokat el kell távolítani. Ha pedig az egyesítés eredménye két egymáshoz illeszkedő szakasz, akkor azokat egybe kell olvasztani.

Az ismertetett módszer a fa csúcsának feldolgozását a részfák feldolgozására és a belső szakaszokon végrehajtott halmazműveletre vezette vissza. Ez egy rekurzív eljárással implementálható, amelyet addig folytatunk, amíg el nem jutunk a CSG-fa leveleihez. Az algoritmus pszeudokódja az alábbi:

```
CSGIntersect(ray, node) {
    if (node nem levél) {
        left span = CSGIntersect(ray, node bal gyermeke);
        right span = CSGIntersect(ray, node jobb gyermeke);
        return CSGCombine(left span, right span, operation);
    } else                    // node primitív objektumot reprezentáló levél
        return PrimitiveIntersect(ray, node);
}
```

6.4. A metszéspontszámítás gyorsítási lehetőségei

Egy naiv sugárkövetés algoritmus minden egyes sugarat minden objektummal összevet, és eldönti, hogy van-e köztük metszéspont. A módszer jelentősen gyorsítható lenne, ha az objektumok egy részére kapásból meg tudnánk mondani, hogy az adott sugár biztosan nem metszheti őket (mert például azok a sugár kezdőpontja mögött, vagy nem a sugár irányában helyezkednek el), illetve miután találunk egy metszéspontot, akkor ki tudnánk zárni az objektumok egy másik körét azzal, hogy ha a sugár metszi is őket, akkor azok biztosan ezen metszéspont mögött helyezkednek el. Ahhoz, hogy ilyen döntéseket hozhassunk, ismernünk kell az objektumteret. A megismeréshez egy előfeldolgozási fázis szükséges, amelyben a metszéspontszámítás gyorsításához szükséges adatstruktúrát építjük fel.

6.4.1. Befoglaló keretek

A legegyszerűbb gyorsítási módszer a *befoglaló keret*ek (*bounding volume*) alkalmazása. A befoglaló keret egy egyszerű geometriájú objektum, tipikusan gömb vagy téglatest, amely egy-egy bonyolultabb objektumot teljes egészében tartalmaz. A sugárkövetés során először a befoglaló keretet próbáljuk a sugárral elmetszeni. Ha nincs metszéspont, akkor nyilván a befoglalt objektummal sem lehet metszéspont, így a bonyolultabb számítást megtakaríthatjuk.

A befoglaló keretet úgy kell kiválasztani, hogy a sugárral alkotott metszéspontja könnyen kiszámítható legyen, és ráadásul kellően szorosan körbeölelje az objektumot. A könnyű metszéspontszámítás követelménye feltétlenül teljesül a gömbre, hiszen ehhez csak egyetlen másodfokú egyenletet kell megoldani.

A *Cohen – Sutherland szakaszvágó algoritmus* (7.4.1. fejezet) bevetésével a koordinátatengelyekkel párhuzamosan felállított befoglaló dobozokra ugyancsak hatékonyan dönthetjük el, hogy a sugár metszi-e őket. A vágási tartománynak a dobozt tekintjük, a vágandó objektumnak pedig a sugár kezdőpontja és a maximális sugárparaméter[5] által kijelölt pontja közötti szakaszt. Ha a vágóalgoritmus azt mondja, hogy a szakasz teljes egészében eldobandó, akkor a doboznak és a sugárnak nincs közös része, következésképpen a sugár nem metszhet semmilyen befoglalt objektumot.

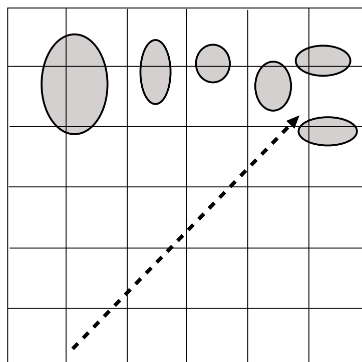
A befoglaló keretek hierarchikus rendszerbe is szervezhetők, azaz a kisebb keretek magasabb szinteken nagyobb keretekbe foghatók össze. Ekkor a sugárkövetés során a befoglaló keretek által definiált hierarchiát járjuk be.

6.4.2. Az objektumtér szabályos felosztása

Tegyünk az objektumtérre egy szabályos 3D rácsot (6.9. ábra) és az előfeldogozás során minden cellára határozzuk meg a cellában lévő, vagy a cellába lógó objektumokat! A sugárkövetés fázisában egy adott sugárra a sugár által metszett cellákat a kezdőponttól való távolságuk sorrendjében látogatjuk meg. Egy cellánál csak azon objektumokat kell tesztelni, amelyeknek van közös része az adott cellával. Ráadásul, ha egy cellában az összes ide tartozó objektum tesztelése után megtaláljuk a legközelebbi metszéspontot, be is fejezhetjük a sugár követését, mert a többi cellában esetlegesen előforduló metszéspont biztosan a megtalált metszéspontunk mögött van.

Ennek a módszernek előnye, hogy a meglátogatandó cellák könnyen előállíthatók egy 3D szakaszrajzoló (*DDA*) algoritmus [45] segítségével, hátránya pedig az, hogy gyakran feleslegesen sok cellát használ. Két szomszédos cellát ugyanis elég lenne csak akkor szétválasztani, ha azokhoz az objektumok egy más halmaza tartozik. Ezt az elvet követik az adaptív felosztó algoritmusok.

[5] t_{max} = a kamerával együtt értendő színtér átmérője



6.9. ábra. *Az objektumtér szabályos felosztása*

6.4.3. Az oktális fa

Az objektumtér adaptív felosztása rekurzív megközelítéssel lehetséges. A fa építésének folyamata a következő:

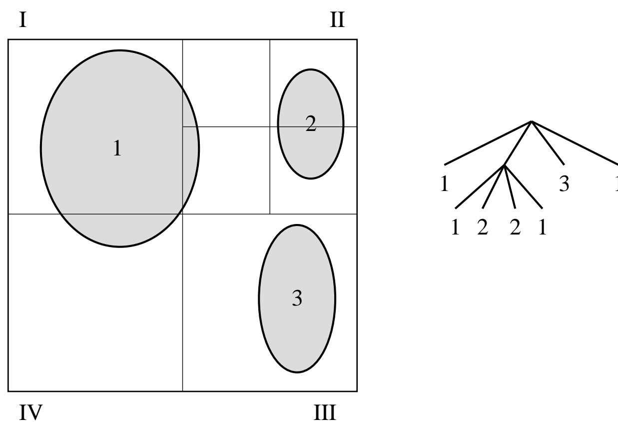
- Kezdetben foglaljuk az objektumainkat egy koordinátatengelyekkel párhuzamos oldalú dobozba, majd határozzuk meg a színtér befoglaló dobozát is! Ez lesz az oktális fa gyökere, és egyben a rekurzió kiindulópontja.
- Ha az aktuális cellában a belógó befoglaló dobozok száma nagyobb, mint egy előre definiált érték, akkor a cellát a felezősíkjai mentén 8 egybevágó részcellára bontjuk, majd a keletkező részcellákra ugyanazt a lépést rekurzívan megismételjük.
- A gráfépítő folyamat egy adott szinten megáll, ha az adott cellához vezető út elér egy előre definiált maximális mélységét, vagy az adott cellában az objektumok száma egy előre definiált érték alá esik.

Az eljárás eredménye egy *oktális fa* (6.10. ábra). A fa levelei azon elemi cellák, amelyekhez a belógó objektumokat nyilvántartjuk. Az adaptív felosztás kétségkívül kevesebb memóriát igényel, mint a tér szabályos felosztása.

A metszéspontszámítás során végig kell menni a fa levelein:

```
IntersectOctree(Ray ray) {
    Q = ray.origin;
    do { // végigmegy a cellákon
        cella = findnode(Q);
        for (minden ojektumra a cellában) Intersect(ray, ojektum);
        if (nincs metszéspont)
            Q = a ray olyan pontja amely már a következő cellában van;
    } while (nincs metszéspont és Q a színtérben van);
}
```

Töprengjünk el egy kicsit az algoritmus azon lépésén, amely a következő cellát határozza meg! A szabályos felosztás rácsán szakaszrajzoló algoritmusok segítségével kényelmesen sétálhattunk, azaz könnyen eldönthettük, hogy egy cella után melyik lesz a következő, amely a sugár útjába kerül. Az adaptív felosztásoknál egy cella után következő cella meghatározása már nem ilyen egyszerű. A helyzet azért nem reménytelen, és a következő módszer elég jól megbirkózik vele.



6.10. ábra. *A síkot felosztó négyes fa, amelynek a 3D változata az oktális fa*

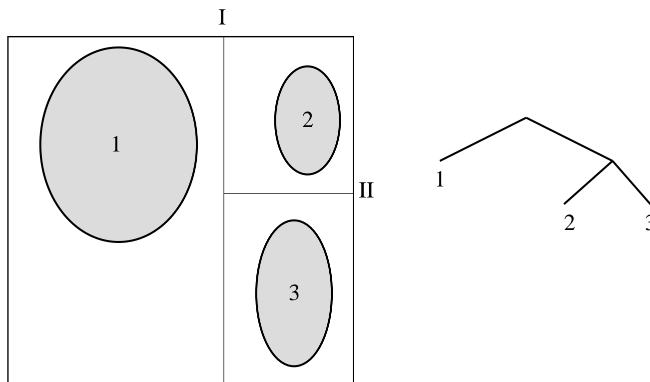
Az aktuális cellában számítsuk ki a sugár kilépési pontját, azaz a sugárnak és a cellának a metszéspontját, majd adjunk hozzá a metszéspont sugárparaméteréhez egy „kicsit”! A kicsivel továbblendített sugárparamétert visszahelyettesítve a sugáregyenletbe, egy, a következő cellában lévő pontot (az algoritmusban a `Q` pont) kapunk. Azt, hogy ez melyik cellához tartozik, az adatstruktúra bejárásával (`findnode(Q)`) dönthetjük el. Kézbe fogván a pontunkat a fa csúcsán belépünk az adatstruktúrába. A pont koordinátáit a felosztási feltétellel (oktális fánál a doboz középpontjával) összehasonlítva eldönthetjük, hogy melyik úton kell folytatni az adatszerkezet bejárását. Előbb-utóbb eljutunk egy levélig, azaz azonosítjuk a pontot tartalmazó cellát.

6.4.4. A kd-fa

Az oktális fa adaptálódik az objektumok elhelyezkedéséhez. A felbontás azonban mindig felezi a cellaoldalakat, tehát nem veszi figyelembe, hogy az objektumok hol helyezkednek el, így az adaptivitás nem tökéletes. Ennél jobb algoritmust akkor tudunk csak készíteni, ha észrevesszük, hogy egy oktális fa bejárási ideje a fa átlagos mélységével arányos. Az oktális fa építésének pedig nagy valószínűséggel egy kiegyensúlyozatlan fa az eredménye.

Tekintsünk egy olyan felosztást, amely egy lépésben nem mind a három felezősík mentén vág, hanem egy olyan síkkal, amely az objektumteret a lehető legigazságosabban felezi meg! Ez a módszer egy bináris fához vezet, amelynek neve *bináris térparticionáló fa*, vagy *BSP-fa* (az angol Binary Space Partition kifejezés nyomán).

Ha a felezősík a koordinátarendszer valamely tengelyére merőleges, akkor *kd-fa* adatszerkezetről beszélünk. Az elnevezés onnan ered, hogy a módszer egy általános k dimenziós teret egy $k-1$ dimenziós hipersíkkal vág két térfélre.



6.11. ábra. *kd-fa*

A felezősík elhelyezése és iránya a kd-fában

A kd-fában a felezősíkot többféleképpen elhelyezhetjük. A *térbeli középvonal módszer* a befoglaló keretet mindig két egyforma részre osztja. Mivel a felezés eredménye mindig két egyforma nagyságú cellát eredményez, ezért ezeknek a részeknek a fa mélységével arányosan egyre kisebbeknek kell lennie.

A *test középvonal módszer* úgy osztja fel a teret, hogy annak bal és jobb oldalán egyforma számú test legyen. Néhány test ebben az esetben mind a jobb, mind a bal oldali ágba kerülhet, hiszen a felezősík akár testeket is metszhet.

A harmadik módszer valamilyen *költség modellt* használ, azaz a *kd-fa* felépítése során becsli azt az átlagos időt, amelyet egy sugár a kd-fa bejárása során felhasznál és ennek minimalizálására törekszik. Ez az eljárás teljesítményben felülmúlja mind a térbeli középvonal, mind a test középvonal módszert. Egy megfelelő költségmodell szerint úgy felezzük a cellát, hogy a két gyermek cellában lévő testek összes felülete megközelítőleg megegyezzen, így a metszés ugyanakkora valószínűséggel következik be a gyermek cellákban [52].

A felezősík irányát a fa építésekor a mélység növekedésével ciklikusan változtathatjuk (X,Y,Z,X,Y,Z,X...). Az elmondottak alapján egy általános kd-fa építő rekurzív algoritmust mutatunk be. A `node` paraméter az aktuális cellát, a `depth` a rekurzió mélységét, a `currentSubdividingAxis` pedig az aktuális vágósík orientációját jelenti:

```
void Subdivide(node, depth, currentSubdividingAxis) {
    if (node.object száma < MaxObjectsInCell vagy depth > dMax) return;

    child[0] és child[1] befoglalódoboza = node befoglalódoboza;
    if (subdividingAxis = X) {
        child[1].min.x = Node cella középpontja X irányban;
        child[0].max.x = Node cella középpontja X irányban;
    } else if (subdividingAxis = Y) {
        child[1].min.y = Node cella középpontja Y irányban;
        child[0].max.y = Node cella középpontja Y irányban;
    } else if (subdividingAxis = Z) {
        child[1].min.z = Node cella középpontja Z irányban;
        child[0].max.z = Node cella középpontja Z irányban;
    }
    for (Node objektumaira) {
        if (ha az objektum a child[0] befoglaló dobozában van)
            adjuk az objektumot a child[0] listájához;
        if (ha az objektum a child[1] befoglaló dobozában van)
            adjuk az objektumot a child[1] listájához;
    }
    Subdivide(child[0], depth + 1, RoundRobin(currentSubdividingAxis));
    Subdivide(child[1], depth + 1, RoundRobin(currentSubdividingAxis));
}
```

A kd-fa bejárása

A kd-fa felépítése után egy olyan algoritmusra is szükségünk van, amely segítségével egy adott sugárra nézve meg tudjuk mondani annak útját a fában, és meg tudjuk határozni a sugár által elsőként metszett testet is. A továbbiakban két algoritmust mutatunk be ennek az adatstruktúrának a bejárására: a *szekvenciális sugárbejárási algoritmus*t (*sequential ray traversal algorithm*) és a *rekurzív sugárbejárási algoritmus*t (*recursive ray traversal algorithm*) [52] [122].

A *szekvenciális sugárbejárási algoritmus* a sugár mentén lévő celláknak a kd-fában történő szekvenciális megkeresésén alapul. Legelső lépésként a kezdőpontot kell meghatározni a sugár mentén, ami vagy a sugár kezdőpontja, vagy pedig az a pont, ahol a sugár belép

a befoglaló keretbe[6]. A pont helyzetének meghatározása során azt a cellát kell megtalálnunk, amelyben az adott pont van. Megint kézbe fogván a pontunkat a fa csúcsán belépünk az adatstruktúrába. Az adott pont koordinátáit a sík koordinátájával összehasonlítva eldönthetjük, hogy melyik úton kell folytatni az adatszerkezet bejárását. Előbb-utóbb eljutunk egy levélig, azaz azonosítjuk a pontot tartalmazó cellát.

Ha ez a cella nem üres, akkor megkeressük a sugár és a cellában lévő illetve a cellába belógó testek metszéspontját. A metszéspontok közül azt választjuk ki, amelyik a legközelebb van a sugár kezdőpontjához. Ezután ellenőrizzük, hogy a metszéspont a vizsgált cellában van-e (mivel egy test több cellába is átlóghat, előfordulhat, hogy nem ez a helyzet). Ha a metszéspont az adott cellában van, akkor megtaláltuk az első metszéspontot, így befejezhetjük az algoritmust. Ha a cella üres, vagy nem találtunk metszéspontot, esetleg a metszéspont nem a cellán belül van, akkor tovább kell lépnünk a következő cellára. Ehhez a sugár azon pontját határozzuk meg, ahol elhagyja a cellát. Ezután ezt a metszéspontot egy kicsit előre toljuk, hogy egy a következő cellában lévő pontot kapjunk. Innentől az algoritmus a tárgyalt lépéseket ismételi.

Ennek az algoritmusnak hátránya, hogy mindig a fa gyökerétől indul, pedig nagyban valószínűsíthető, hogy két egymás után következő cella esetén a gyökérből indulva részben ugyanazon cellákat járjuk be. Ebből adódóan egy csomópontot többször is meglátogatunk.

A *rekurzív sugárbejárási algoritmus* (*recursive ray traversal algorithm*) [52] [113] a szekvenciális sugárbejárási algoritmus hátrányait igyekszik kiküszöbölni, és minden belső pontot és levelet csak egyetlen egyszer látogat meg. Amikor a sugár egy olyan belső csomóponthoz ér, amelynek két gyermekcsomópontja van, eldönti hogy a gyermekeket milyen sorrendben látogassa meg. A gyermekcsomópontokat „közeli” és „távoli” gyermekcsomópontként osztályozzuk aszerint, hogy azok milyen messze helyezkednek el a sugár kezdetétől, a felezősíkhoz képest. Ha a sugár csak a „közeli” gyermekcsomóponton halad keresztül, akkor a sugár ennek a csomópontnak az irányába mozdul el, és az algoritmus rekurzívan folytatódik. Ha a sugárnak mindkét gyermekcsomópontot meg kell látogatnia, akkor az algoritmus egy veremtárban megjegyzi az információkat a „távoli” gyermekcsomópontról, és a „közeli” csomópont irányába mozdul el, majd rekurzívan folytatódik az algoritmus. Ha a „közeli” csomópont irányában nem találunk metszéspontot, akkor a veremből a „távoli” gyermekcsomópontot vesszük elő, és az algoritmus rekurzívan fut tovább, immár ebben az irányban.

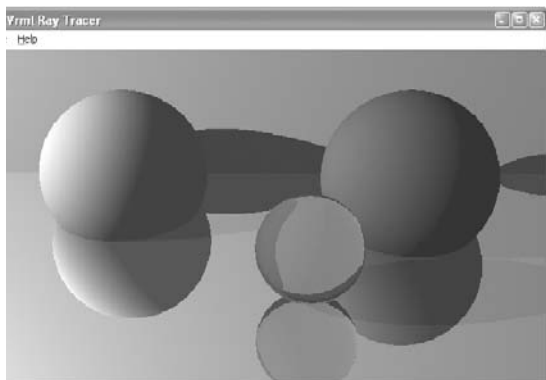
Az algoritmus kódja a következő:

[6] attól függően, hogy a sugár kezdőpontja a befoglaló dobozon belül van-e vagy sem

```
enum Axes {X_axis, Y_axis, Z_axis}; // X, Y, Z tengelyek

//===============================================================
struct KDTreeNode {           // a kd-fa cellája
//===============================================================
   Point3d min, max;          // a cella kiterjedése
   GeomObjlist* objlist;      // a cellához tartozó objektumok listája
   struct KDTreeNode *left, *right;  // bal és jobb gyerek
   Axis axis;                 // a felezősík orientációja
};
//===============================================================
struct StackElem {            // a verem egy eleme
//===============================================================
   KDTreeNode* node;
   float a, b;                // a be- és kilépés előjeles távolsága
};

//---------------------------------------------------------------
Object RayTravAlg(KFTreeNode *roorNode, Ray ray) { // rekurzív bejárás
//---------------------------------------------------------------
   float a, b;                 // a belépés/kilépés előjeles távolsága
   float t;                    // a felezősík távolsága
   StackElem stack[MAXDEPTH]; // verem
   int stackPtr = 0;           // mutató a veremre
   KDTreeNode *farChild, *nearChild, *currNode; //gyerekek, aktuális cella

   RayBoxIntersect(ray, rootNode, &a, &b); // metszés a befoglalódobozzal
   if ( "nincs metszéspont" ) return ["Nincs metszéspont"];
   "Tedd a (rootNode, a, b)-t a verem tetejére"

   while ( "a verem nem üres" ) {  // amíg a fát be nem jártuk
      "Vedd ki a (currNode, a, b)-t a veremből"
      while ("currNode nem levél") {
         float diff = currNode->right.min[axis] - ray.origin[axis]
         t = diff / ray.dir[axis];
         if (diff > 0.0) {
              nearChild = currNode->left; farChild = currNode->right;
         } else {
              nearChild = currNode->right; farChild = currNode->left;
         }
         if ( (t > b) || (t < 0.0) ) currNode = nearChild;
         else {
            if (t < a) currNode = farChild;
            else {
               "Tedd a (farNode, t, b)-t a verem tetejére";
               currNode = nearChild;
               b = t;
            }
         }
      }
     // ha az aktuális csomópont egy levél
      "a listában lévő objektumokkal metszéspontszámítás"
      "ha egy metszéspont nem a és b között van --> eldobjuk"
      if (létezik metszéspont) return ["legközelebbi metszéspont"]
   }
   return ["Nincs metszéspont"];
}
```

6.5. Program: rekurzív sugárkövetés

A sugárkövetés algoritmust C++ környezetben valósítottuk meg. A *VRML* beolvasó programból (lásd 5.3.3. fejezet) indulunk ki, és ebből készítünk egy VRMLViewerRT (6.12. ábra) alkalmazást. Sajnos — mint később látni fogjuk — a hiányos anyagmodell leírás miatt egy VRML-ben megadott színtérből nem kapjuk meg a sugárkövetéshez szükséges összes adatot. A VRML kiterjesztésére[7] már vannak ígéretes kezdeményezések, elterjedésükig azonban megpróbálunk a jelenlegi VRML leírással dolgozni.



6.12. ábra. *Sugárkövetés mintaprogrammal készített kép*

A VRML színtér előkészítésénél a következőkre kell ügyelnünk. Lehetőleg pontszerű fényforrásaink legyenek (hiszen a sugárkövetés ezekre hatékony). Ha tükör anyagot szeretnénk, állítsuk az anyag fényesség (`nu`) paraméterét 100-ra. Az átlátszóság (`transparency`) paraméter a VRML-ben hullámhosszfüggetlen skalár érték. VRML-ben törésmutatót az anyagokra nem tudunk megadni, ezért ezt „beégetjük” a programba. A példaprogram az egyszerűség kedvéért csak indexelt háromszöglistával és gömbökkel birkózik meg. Ezek után nézzük a program osztályait!

Egy sugár (`Ray`) kezdőponttal (`origin`) és irányvektorral (`dir`) jellemezhető:

```
//===============================================================
class Ray {
//===============================================================
public:
    Vector  origin;      // kezdőpont
    Vector  dir;         // irány
    Ray(const Vector& newOrigin,  const Vector& newDir);

};
```

[7] például a Philippe Bekaert munkája, a PhBRML fájlformátum

Az objektumok anyagi jellemzőit a `Material` osztály tartalmazza, amelyet az anyagokkal foglalkozó 4. fejezetben adtunk meg. A `FinishMaterial()` függvény a sugárkövetés elindítása előtt előfeldolgozást végez az objektumon, és beállítja azokat az értékeket, amelyek a VRML fájlból hiányoznak:

```
//-------------------------------------------------------------------
void Material::FinishMaterial(void) {
//-------------------------------------------------------------------
    if (n >= 100.0) {   // 100-as shine esetén tükörnek tekintjük
        kr  = ks;                   // tükör együttható feltöltése
        ks  = Color(0.0, 0.0, 0.0); // spekuláris együttható törlése
    }
    nu = 1.2;           // mert a törésmutatót VRML-ben nem lehet megadni
}
```

Egy ideális tükör csak az elméleti visszaverődési irányba veri vissza a fényt. A BRDF tehát egyetlen irányban végtelen értékű, másutt pedig zérus, így nem reprezentálható közvetlenül. Ahol erre a programban szükség van, a következő programsorral állítjuk elő a fénysugár tükörirányát (lásd 6.2. egyenlet):

```
Vector reflDir = normal * (-2.0 * (inDir * normal)) + inDir;
```

Az ideális fénytörő anyag szintén csak egyetlen irányba adja tovább a fényt, amelyet a `RefractionDir()` függvénnyel számíthatunk ki az anyag törésmutatójából (`nu`). A bejövő irány és a normálvektor segítségével meg lehet határozni, hogy a fénytörő felületet kívülről vagy belülről közelítjük-e meg. Ha belülről jövünk, akkor a törésmutató reciprokát kell használni. A függvény a visszatérési értékében jelzi, ha *teljes visszaverődés* miatt nem létezik törési irány.

```
//-------------------------------------------------------------------
bool Material::RefractionDir(const Vector& inDir,
                             const Vector& normal, Vector* outDir) {
//-------------------------------------------------------------------
    double cosIn = -1.0 * (inDir * normal);
    if (fabs(cosIn) <= EPSILON) return false;

    float cn = nu;                  // törésmutató
    Vector useNormal = normal;
    if (cosIn < 0) {                // ha az anyag belsejéből jövünk
        cn          = 1.0 / nu;
        useNormal   = -normal;      // a törésmutató reciprokát kell használni
        cosIn       = -cosIn;
    }

    float disc = 1 - (1 - cosIn * cosIn) / cn / cn;// Snellius-Descartes törv.
    if (disc < 0) return false;
    *outDir = useNormal * (cosIn / cn - sqrt(disc)) + inDir / cn;
    return true;
}
```

A színtér kamerából, anyagokból, objektumokból és fényforrásokból épül fel.

```
//===============================================================
class Scene {
//===============================================================
public:
    Camera                  camera;     // kamera
    std::vector <Material>  materials;  // anyagok vektora
    std::vector <Object*>   objects;    // objektumok
    std::vector <Light*>    lights;     // fényforrások

    bool      Intersect        (const Ray& ray, HitRec* hitRec);
    Color     Trace            (const Ray& ray, short depth);
    Color     DirectLightsource(const Vector& inDir, const HitRec& hitRec);
};
```

Egy objektumhoz két művelet tartozik. Az objektum és egy sugár metszéspontját az `Intersect()` függvény számítja ki, amely egy `HitRec` adatstruktúrát ad vissza. Az objektum anyagát a metszéspontban a `GetMaterial()` metódussal kaphatjuk meg.

```
//===============================================================
class Object {
//===============================================================
public:
    virtual bool Intersect(const Ray& ray, HitRec* hitRec) { return false; };
    virtual Material* GetMaterial(const HitRec& hitRec) {return NULL; };
};
```

Az `Object` ősosztály virtuális metódusait valamilyen alapértelmezett működéssel definiáljuk.

A fenti metódusokban paraméterként szereplő metszéspontleíró `HitRec` osztály a következő:

```
//===============================================================
class HitRec {
//===============================================================
public:
    int     objectInd;      // objektum index
    int     primitiveInd;   // primitív index
    Vector  point;          // metszéspont
    Vector  normal;         // normálvektor az adott pontban
    float   t;              // sugárparaméter

    HitRec() { objectInd = primitiveInd = -1; t = 0.0; }
};
```

A mintaprogramunkban kétféle objektumtípus létezhet: gömb (`Sphere`) és háromszögháló (`Mesh`). A gömböt definiáló osztály az `Object` származtatott osztálya, amely újraértelmezi annak virtuális függvényeit. A gömb geometriáját egy középponttal (`origin`) és egy sugárral (`radius`) írjuk le:

```
//===============================================================
class Sphere : public Object {
//===============================================================
public:
    Vector      origin;         // gömb középpontja
    float       radius;         // sugara
    Material*   pMaterial;      // anyaga

    bool        Intersect(const Ray& ray, HitRec* hitRec);
    Material*   GetMaterial(const HitRec& hitRec) { return pMaterial; };
};
```

Az `Intersect()` metódus egy sugár és a gömb metszéspontját adja meg:

```
//-----------------------------------------------------------------
bool Sphere::Intersect(const Ray& ray, HitRec* hitRec) {
//-----------------------------------------------------------------
    Vector dist = ray.origin - origin;
    double b = (dist * ray.dir) * 2.0;   // másodfokú egyenlet együtthatói
    double a = (ray.dir * ray.dir);      // a > 0, ezért t1 > t2 lesz
    double c = (dist * dist) - radius * radius;

    double discr = b * b - 4.0 * a * c;  // diszkrimináns
    if (discr < 0) return false;         // ha negatív --> nincs megoldás
    double sqrt_discr = sqrt(discr);
    double t1 = (-b + sqrt_discr)/2.0/a; // az egyik sugárparaméter
    double t2 = (-b - sqrt_discr)/2.0/a; // a másik sugárparaméter

    if (t1 < EPSILON) t1 = -EPSILON;     // ha túl közel --> érvénytelen
    if (t2 < EPSILON) t2 = -EPSILON;     // ha túl közel --> érvénytelen
    if (t1 < 0.0 && t2 < 0.0) return false;

    float t;    // a kisebbik pozitív sugárparaméter kiválasztása
    if (t1 < 0.0) return false; // ekkor t2 is kisebb, hiszen t1 > t2
    if (t2 > 0.0) t = t2;       // t2 a kisebb a kettő közül
    else t = t1;

    hitRec->t       = t;    // hitRec feltöltése
    hitRec->point   = ray.origin + ray.dir * t;
    hitRec->normal  = (hitRec->point - origin) / radius;
    return true;
}
```

A háromszöghálót definiáló `Mesh` osztály is az `Object` származtatott osztálya:

```
//===============================================================
class Mesh : public Object {
//===============================================================
public:
    std::vector <Vector>    vertices;   // csúcspontok
    std::vector <Patch>     patches;    // háromszögek

    bool        Intersect(const Ray& ray, HitRec* hitRec);
    Material*   GetMaterial(const HitRec& hitRec)
           { return patches[hitRec.primitiveInd].pMaterial; };
};
```

A `Mesh` háromszögekből (`patches`) áll. A háromszögek pointerekkel hivatkoznak a `Mesh` osztályban definiált csúcspontokra (`vertices`). A metszéspontszámítást az `Intersect()` metódus végzi el:

```
//-----------------------------------------------------------------
bool Mesh::Intersect(const Ray& ray, HitRec* hitRec) {
//-----------------------------------------------------------------
    hitRec->primitiveInd = -1;
    float mint = FLT_MAX; // minimumkeresés
    HitRec hitRecLocal;
    for(int i = 0; i < patches.size(); i++) {
        if (!patches[i].Intersect(ray, &hitRecLocal)) continue;
        if (hitRecLocal.t < mint) { // ha új minimumot találunk
            mint = hitRecLocal.t;
            hitRec->primitiveInd = i;
            hitRec->t       = hitRecLocal.t;
            hitRec->point   = hitRecLocal.point;
            hitRec->normal  = patches[i].normal;
        }
    }
    return hitRec->primitiveInd != -1;
}
```

A `Mesh::Intersect()` függvény a `Patch::Intersect()` függvényt használja fel. A háromszög metszésére a korábban ismertetett két algoritmus közül a háromdimenziós kódját részletezzük. A CD-n, a mintaprogramban azonban a hatékonyabb kétdimenziós módszer forrása is megtalálható.

```
//-----------------------------------------------------------------
bool  Patch::Intersect(const Ray& ray, HitRec* hitRec) {
//-----------------------------------------------------------------
    double cost = ray.dir * normal;
    if (fabs(cost) <= EPSILON) return false;

    double t = ((*a - ray.origin) * normal)/cost; // sugárparaméter
    if(t < EPSILON) return false; // ha túl közel --> érvénytelen

    Vector ip = ray.origin + ray.dir * t; // a metszéspont
    hitRec->point   = ip;
    hitRec->t       = t;

    double c1 = (((*b - *a) % (ip - *a)) * normal); // vektoriális szorzat
    double c2 = (((*c - *b) % (ip - *b)) * normal); // vektoriális szorzat
    double c3 = (((*a - *c) % (ip - *c)) * normal); // vektoriális szorzat

    if (c1>=0 && c2>=0 && c3>=0) return true; // a háromszög belsejében van
    if (c1<=0 && c2<=0 && c3<=0) return true; // ellentétes körüljárás esetén
    return false;
}
```

Az objektumok definícióján kívül a sugárkövetés algoritmusnak még egy olyan függvényre van szüksége, amely egy képernyőn lévő (x, y) ponthoz megkeresi azt a sugarat, amely a szemből indul, és éppen ezen a ponton megy keresztül:

```
//-----------------------------------------------------------------
Ray GetRay(int x, int y) {
//-----------------------------------------------------------------
    float   h = scene.camera.pixh;  // pixel horizontális mérete
    float   v = scene.camera.pixv;  // pixel vertikális mérete

    // az aktuális pixel középpontja
    float   pixX = -h * scene.camera.hres / 2.0 + x * h + h / 2.0;
    float   pixY = -v * scene.camera.vres / 2.0 + y * v + v / 2.0;

    Vector rayDir = scene.camera.Z + pixX*scene.camera.X + pixY*scene.camera.Y;
    rayDir.Normalize();
    return Ray(scene.camera.eyep, rayDir); // a sugár a szemből
}
```

A színtér `Intersect` tagfüggvénye megkeresi azt az objektumot, és azon belül azt a primitívet, amelyet a sugár legközelebb metsz, és visszaadja a metszéspont attribútumait tartalmazó `hitRec` adatstruktúrát:

```
//-----------------------------------------------------------------
bool Scene::Intersect(const Ray& ray, HitRec* hitRec) {
//-----------------------------------------------------------------
    hitRec->objectInd = -1;
    float mint = FLT_MAX;
    HitRec hitRecLocal;
    for(int i = 0; i < objects.size(); i++) {          // min. keresés
        if (!objects[i]->Intersect(ray, &hitRecLocal)) continue;
        if (hitRecLocal.t < mint) {
            mint = hitRecLocal.t;
            *hitRec = hitRecLocal;
            hitRec->objectInd = i;
        }
    }
    return (hitRec->objectInd != -1);
}
```

Az egyszerűsített illuminációs képlet kiértékeléséhez — egy adott `hitRec` metszéspontban és egy `inDir` bejövő irány esetén — az absztrakt fényforrások direkt megvilágítását is ki kell számítani.

A direkt megvilágítást a `DirectLightsource()` tagfüggvény határozza meg:

```
//-----------------------------------------------------------------
Color Scene::DirectLightsource(const Vector& inDir, const HitRec& hitRec) {
//-----------------------------------------------------------------
    Color sumColor = Color(0,0,0); // akkumulált sugársűrűség
    for(short i = 0; i < lights.size(); i++) {  // minden fényforrásra
                                // pontszerű fényforrások kezelése
        PointLight* pLight = dynamic_cast<PointLight*>(lights[i]);
                                // sugár a felületi pontból a fényforrásig
        Ray     rayToLight(hitRec.point, pLight->location - hitRec.point);
        float   lightDist = rayToLight.dir.Norm();
        rayToLight.dir.Normalize();
                                // az árnyalási normális az adott pontban
        float   cost = rayToLight.dir * hitRec.normal;
        if (cost <= 0) continue;    // a test belsejéből jövünk

        HitRec  hitRecToLight;
        bool isIntersect = Intersect(rayToLight, &hitRecToLight);
        bool meetLight = !isIntersect;
        if (isIntersect) {  // a metszéspont távolabb van, mint a fényforrás
            Vector distIntersect = pLight->location - hitRecToLight.point;
            if (distIntersect.Norm() > lightDist) meetLight = true;
        }
        if (!meetLight) continue;       // árnyékban vagyunk

        Color brdf = objects[hitRec.objectInd]->GetMaterial(hitRec)->
                                Brdf(inDir,rayToLight.dir, hitRec.normal);
        sumColor += brdf * lights[i]->emission * cost;
    }
    return sumColor;
}
```

A program legfontosabb része a sugarat rekurzívan követő `Trace` függvény, amely az egyszerűsített illuminációs egyenletnek megfelelően négy összetevőből állítja elő a sugár által kijelölt pont színét.

```
//-----------------------------------------------------------------
Color Scene::Trace(const Ray& ray, short depth) {
//-----------------------------------------------------------------
    if (depth > MaxDepth) return gAmbient; // rekurzió korlátozása
    HitRec hitRec;
    // ha nincs metszéspont kilépünk
    if (!Intersect(ray, &hitRec)) return gAmbient;

    // 1. ambiens rész
    Color ambientColor = objects[hitRec.objectInd]->
                                GetMaterial(hitRec)->ka * gAmbient;
    // 2. fényforrások közvetlen hatása
    Color directLightColor = DirectLightsource(ray.dir, hitRec);
```

```
    // 3. ideális tükör rész
    Material* pMaterial = objects[hitRec.objectInd]->GetMaterial(hitRec);
    Color idealReflector = Color(0,0,0);
    Color kr = pMaterial->kr;
    if (kr.Average() > EPSILON) {
        Vector reflDir = hitRec.normal * (-2.0 * (ray.dir * hitRec.normal))
                                        + ray.dir;
        idealReflector = kr * Trace(Ray(hitRec.point, reflDir), depth + 1);
    }

    // 4. ideális fény törés rész
    Color idealRefractor = Color(0,0,0);
    Color kt = pMaterial->kt;
    if (kt.Average() > EPSILON) {
        Vector refrDir; //törésmutató függő
        if (pMaterial->RefractionDir(ray.dir, hitRec.normal, &refrDir))
            idealRefractor = kt * Trace(Ray(hitRec.point, refrDir), depth + 1);
    }

    return ambientColor + directLightColor + idealReflector + idealRefractor;
}
```

A képszintézis végrehajtása során minden egyes pixelközépponton keresztül egy sugarat indítunk az objektumtérbe, majd a sugárkövetés által számított színnek megfelelően kifestjük a pixelt.

```
//-----------------------------------------------------------------
void RayTracingApplication::Render(void) {
//-----------------------------------------------------------------
    for(int y = 0; y <= scene.camera.vres; y++) {
        for(int x = 0; x <= scene.camera.hres; x++) {
            Ray r = GetRay(x, y);
            Color color = scene.Trace(r, 0);
            SetPixel(x, y, color);
        }
    }
}
```

7. fejezet

Inkrementális képszintézis

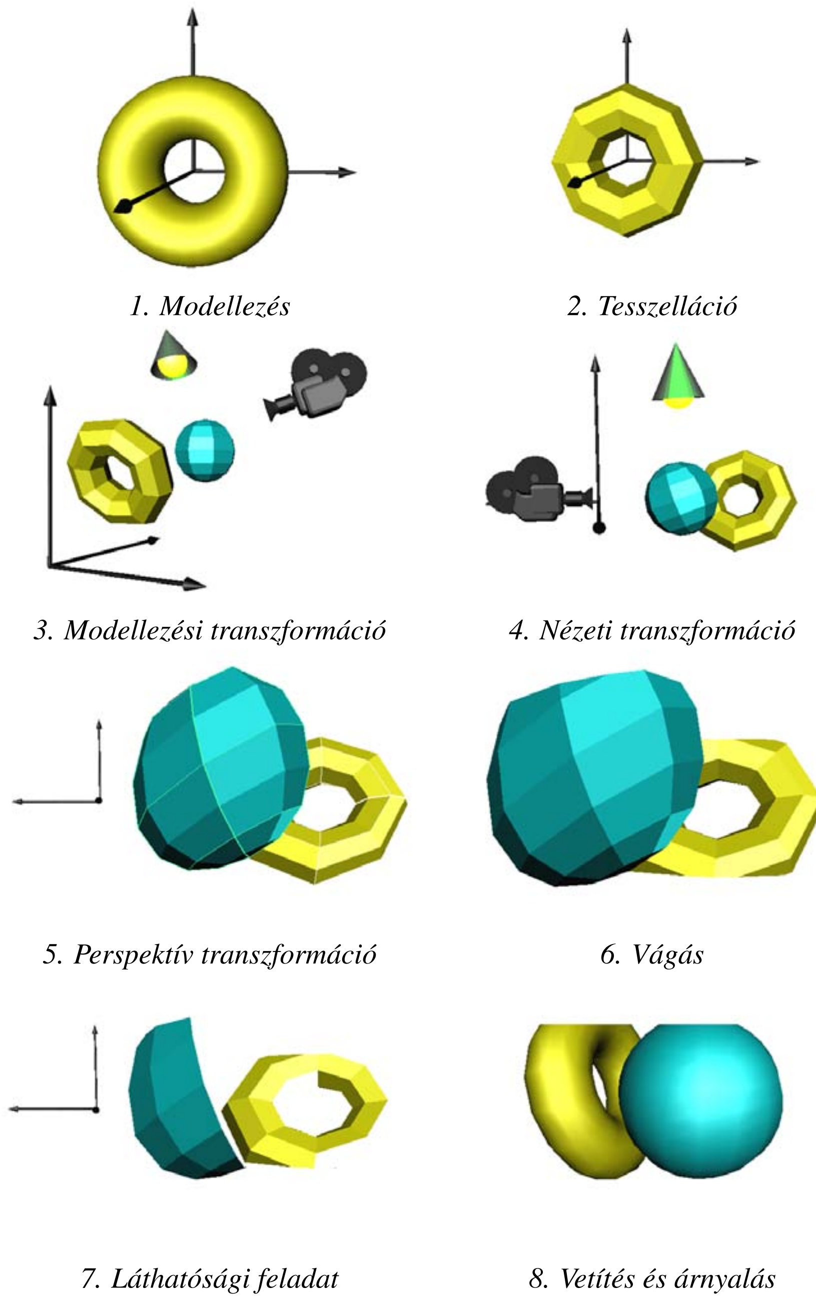
A képszintézis során a virtuális világról fényképet készítünk és azt a monitor képernyőjén megjelenítjük. A világban szereplő objektumokat vagy a saját modellezési-koordinátarendszerükben, vagy közvetlenül a világ-koordinátarendszerben definiáljuk. Míg a modellből monitoron megjeleníthető kép lesz, számos feladatot (például takarás és árnyalás) kell megoldani. A sugárkövetés ezeket a feladatokat pixelenként egymástól függetlenül hajtja végre, azaz nem használja fel újra az egyszer már nagy nehezen megszerzett takarási és árnyalási információkat, így egy interaktív program számára nem elég gyors. Az *inkrementális képszintézis* néhány egyszerű elv alkalmazásával az alapfeladatok végrehajtási idejét jelentősen lerövidíti:

1. A feladatok egy részének elvégzése során elvonatkoztat a pixelektől, és az objektumtér nagyobb részeit egységesen kezeli.

2. Ahol csak lehet, kihasználja az *inkrementális elv* nyújtotta lehetőségeket. Az inkrementális elv alkalmazása azt jelenti, hogy egy pixel takarási és árnyalási információinak meghatározása során jelentős számítási munkát takaríthatunk meg, ha a megelőző pixel hasonló adataiból indulunk ki, és nem kezdjük a számításokat elölről.

3. Minden alapfeladatot a hozzá optimálisan illeszkedő koordinátarendszerben végez el, azok között pedig homogén lineáris geometriai transzformációkkal vált. Ezt könnyedén akkor teheti meg, ha a virtuális világban csak sokszögek találhatók, ezért a modellben levő szabadformájú elemeket (például felületeket) sokszögekkel közelítjük. Ezt a műveletet *tesszelláció*nak hívjuk (3. fejezet).

4. Feleslegesen nem számol, ezért a *vágás* során eltávolítja azon geometriai elemeket, illetve azoknak bizonyos részeit, amelyek a képen nem jelennének meg.



7.1. ábra. *Az inkrementális képszintézis lépései*

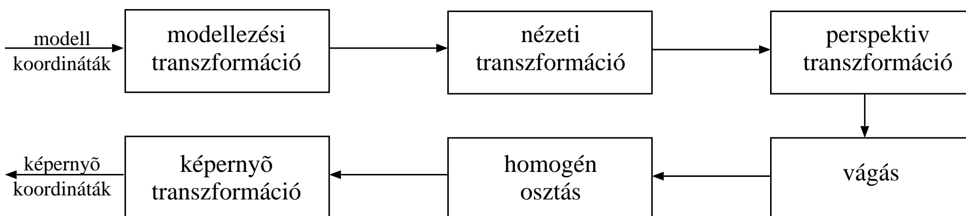
Az inkrementális képszintézis ezen elvek betartása miatt jóval gyorsabb mint a sugárkövetés. Leginkább ennek köszönhető, hogy a valós idejű alkalmazások az inkrementális képszintézist használják. A könyvünkben bemutatott grafikus könyvtárak, az OpenGL (2. fejezet) és a DirectX (11. fejezet) is ezeket az elveket követik. A továbbiakban az inkrementális képszintézis lépéseit az OpenGL szemszögéből mutatjuk be, és feltesszük, hogy a tesszelláció megtörtént, így „csak” a sokszögekkel adott modellt kell lefényképeznünk.

7.1. Nézeti csővezeték

A háromdimenziós grafikában a szempozícióból, egy téglalap alakú ablakon keresztül látható képet szeretnénk előállítani. A képszintézis során el kell dönteni, hogy az objektumok hogyan takarják egymást, és csak a látható objektumokat kell megjeleníteni. Ezen műveleteket közvetlenül a világ-koordinátarendszerben is el tudnánk végezni, azonban ekkor egy pont vetítése egy általános helyzetű egyenes és az ablak metszéspontjának kiszámítását igényelné, a takarás pedig az általános pozíciójú szemtől való távolsággal dolgozna.

Sokkal jobban járunk, ha ezen műveletek előtt áttranszformáljuk a teljes objektumteret egy olyan koordinátarendszerbe, ahol a vetítés és a takarás triviálissá válik. Ezt a rendszert *képernyő-koordinátarendszer*nek nevezzük, amelyben az X,Y koordináták azon pixelt jelölik ki, amelyre a pont vetül, a Z koordináta alapján pedig eldönthetjük, hogy két pont közül melyik van a szemhez közelebb. A képernyő-koordinátarendszerbe átvivő transzformációt egy koordinátarendszereken átvezető transzformáció sorozattal definiáljuk.

A modellezési-koordinátarendszertől a képernyőig tartó transzformáció sorozatot *nézeti csővezeték*nek (*viewing pipeline*) nevezzük, amelyen a virtuális világmodell pontjai „végigfolynak”. A csővezeték végén a primitívek a képernyő-koordinátarendszerben „csöpögnek ki”:



7.2. ábra. *Nézeti csővezeték*

Ha az objektumok a saját modellezési-koordinátarendszereikben állnak rendelkezésre, akkor a képszintézis során a közös világ-koordinátarendszerbe kell átvinni őket. A *modellezési transzformáció* ezt a célt szolgálja. A *nézeti transzformáció* egyrészt elhelyezi a kamerát a virtuális világban, másrészt a színtérre a kamera szemszögéből tekint. A *vágás*t a *perspektív transzformáció* eredményén, a homogén koordinátás alakban kifejezett pontokon hajtjuk végre. A perspektív transzformáció eredményét Descartes-koordinátákban a homogén osztással kapjuk meg, amelyeket ezek után már csak a képernyő-koordinátarendszerbe kell transzformálnunk.

Az OpenGL nézeti csővezetékének első két fokozatában egy-egy mátrixot találunk. A rendszer a modellezési és nézeti transzformációkat együttesen kezeli és a modell–nézeti (MODELVIEW) mátrixban „gyűjti” őket össze, míg a perspektív torzítást a projekciós (PROJECTION) mátrixszal írja le. Ez azt jelenti, hogy ha például egy $\mathbf{T}_{rot}$ elforgatást szeretnénk végrehajtani, akkor az OpenGL a $\mathbf{T}_{modelview}$ modell–nézeti mátrix aktuális állapotát megszorozza a forgatási mátrixszal.

Itt álljunk meg egy pillanatra és vizsgáljuk meg jobban az OpenGL mátrixkezelését! Az OpenGL 4×4 elemű mátrixokat használ, amelyekkel a pontok homogén koordinátás alakját transzformálja. A 3.2. fejezetben már elmélkedtünk azon, hogy a pontokat, illetve a vektorokat tekinthetjük 4×1 elemű, egyetlen oszlopból álló mátrixnak (oszlopvektornak), vagy akár egy 1×4 elemű, egyetlen sorból álló mátrixnak (sokvektornak). Ha az egyik megközelítés helyett a másikat alkalmazzuk, akkor a mátrixainkat a főátlóra tükrözni kell, ezért fontos, hogy pontosan értsük, hogy az OpenGL melyik értelmezést használja.

A dolgot még tovább bonyolítja, hogy a szokásos programozási nyelvek a kétdimenziós tömböket egydimenziós tömbként tárolják a memóriában, amit ugyancsak kétféleképpen tehetnek meg. A C és a C++ nyelv a „sorfolytonos” megoldást követi, amikor a mátrix sorai egymást követik a memóriában, azaz egy $m[4][4]$ mátrix elemei a következő sorrendben foglalnak helyet: $m[0][0]$, $m[0][1]$, $m[0][2]$, $m[0][3]$, $m[1][0]$, $m[1][1]$, $\ldots$, $m[4][3]$, $m[4][4]$. A mátrixok elvileg tárolhatók „oszlopfolytonosan” is, amikor az egyes oszlopok követik egymást, azaz a mátrixelemek sorrendje: $m[0][0]$, $m[1][0]$, $m[2][0]$, $m[3][0]$, $m[0][1]$, $m[1][1]$, $\ldots$, $m[3][4]$, $m[4][4]$. A sor- és oszlopfolytonos értelmezés hasonló eredményre vezet, ha a mátrixot a főátlóra tükrözzük.

Ha az OpenGL dokumentációt olvasgatjuk, akkor azzal a kijelentéssel találkozunk, hogy az OpenGL oszlopvektorokkal dolgozik, és a mátrixokat is oszlopfolytonosan várja. Ebben a könyvben a pontokat és vektorokat sorvektornak tekintettük, ezért az OpenGL oszlopvektoros megközelítése miatt az OpenGL dokumentációban szereplő mátrixok a mi mátrixaink tükörképei. Viszont, ha a C nyelvben szokásos sorfolytonos mátrixokat szeretnénk átadni, akkor az OpenGL dokumentációban szereplő mátrixokat tükrözni kell. A tükrözött mátrix viszont sorvektorokra írja le helyesen a transzformációt, így ha ragaszkodunk a C kétdimenziós tömb tárolási módszeréhez, akkor a mátrixaink pontosan olyan formában szerepelnek, ahogyan az OpenGL-nek át kell adni.

Nem szabad tehát elbizonytalanodnunk, a könyv mátrixai és a sorvektoros értelmezés tökéletesen illeszkedik az C/OpenGL filozófiához, a mátrixokat tükrözni csak fejben, a könyv és az OpenGL dokumentáció párhuzamos olvasásakor kell.

Az OpenGL-ben — a rajzolási állapot elvét (2. fejezet) követve — nem kell minden transzformációnál külön megadni, hogy az a modell–nézeti vagy a projekciós mátrixra vonatkozik-e. Ehelyett a rendszer a megadott transzformációt mindig az aktuális mátrixszal szorozza meg jobbról. A `glMatrixMode()` függvénnyel tudjuk kiválasztani, hogy a modell–nézeti vagy a projekciós mátrixot kívánjuk módosítani. Ha a modell–nézeti mátrixot szeretnénk kijelölni, akkor a `glMatrixMode(GL_MODELVIEW)` függvényhívást kell alkalmaznunk, míg a `glMatrixMode(GL_PROJECTION)` utasítással a projekciós mátrixot jelöljük ki. A kijelölés a `glMatrixMode` legközelebbi hívásáig marad érvényben. A következő alfejezetekben a nézeti csővezeték lépéseit és azok programozását tekintjük át [1].

7.2. Nézeti transzformáció

A képszintézis során általában egy kameraállásból látható látványra vagyunk kíváncsiak, ahol a *szempozíció* határozza meg a kamera helyét ($\vec{eye}$), irányát pedig a nézeti referencia pont ($\vec{lookat}$) és az $\vec{eye}$ vektor különbsége definiálja. A függőleges irányt jelölő $\vec{up}$ egységvektor a kamera billentő szögét adja meg.

A kamerához egy koordinátarendszert, azaz három egymásra merőleges egységvektort rendelünk. Az $\vec{u} = (u_x, u_y, u_z)$ vízszintes, a $\vec{v} = (v_x, v_y, v_z)$ függőleges és a $\vec{w} = (w_x, w_y, w_z)$ nézeti irányba mutató egységvektorokat a következő módon határozhatjuk meg:

$$\vec{w} = \frac{\vec{eye} - \vec{lookat}}{|\vec{eye} - \vec{lookat}|}, \qquad \vec{u} = \frac{\vec{up} \times \vec{w}}{|\vec{up} \times \vec{w}|}, \qquad \vec{v} = \vec{w} \times \vec{u}.$$

Az inkrementális képszintézis számos későbbi lépését akkor könnyű elvégezni, ha a kamera az origóban helyezkedik el és a $-Z$ irányába néz. Itt megint érdemes egy pillanatra elidőzni! A világ-koordinátarendszer jobbsodrású, míg a képernyő-koordinátarendszer balsodrású, ezért a nézeti csővezeték egyik lépésénél mindenképpen váltanunk kell. Az inkrementális elveket követő képszintézis megközelítések legtöbbje ezt már itt, a nézeti transzformáció során megteszi, így a kamerát a Z irányába állítja. Az OpenGL azonban csak a perspektív transzformáció során vált sodrást, ezért forgatja a kamerát a $-Z$ irányába.

A $\mathbf{T}_{view}$ *nézeti transzformáció* a világ-koordinátarendszerből a *kamera-koordinátarendszer*be vált (3.2.8. fejezet), így a kamera szemszögéből néz a világra:

$$[x', y', z', 1] = [x, y, z, 1] \cdot \mathbf{T}_{view} = [x, y, z, 1] \cdot \mathbf{T}_{tr} \cdot \mathbf{T}_{rot}, \tag{7.1}$$

[1] Ebben a fejezetben az OpenGL nézeti csővezetékét mutatjuk be, ám léteznek más megközelítések (PHIGS, GKS) is, amelyekről a [38, 46, 118, 78]-ben olvashat a kedves Olvasó.

ahol a $\mathbf{T}_{tr}$ a világot úgy tolja el, hogy a kamera az origóba kerüljön, míg a $\mathbf{T}_{rot}$ úgy forgat, hogy a kamera bázisvektorai a világ-koordinátarendszer bázisvektoraival essenek egybe:

$$\mathbf{T}_{tr} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -eye_x & -eye_y & -eye_z & 1 \end{bmatrix}, \qquad \mathbf{T}_{rot} = \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

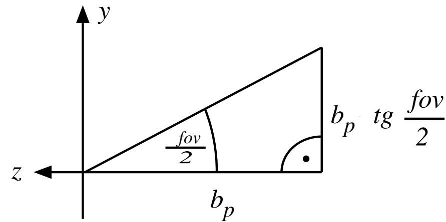
Az OpenGL-ben a nézeti transzformációt a `gluLookAt()` függvénnyel adhatjuk meg, amelynek egy lehetséges implementációját az alábbiakban mutatjuk be:

```
//-----------------------------------------------------------------------
void  gluLookAt(double eye_x, double eye_y, double eye_z,
                double lookat_x, double lookat_y, double lookat_z,
                double up_x, double up_y, double up_z) {
//-----------------------------------------------------------------------
    double w_x = eye_x - lookat_x;            // w vektor komponensei
    double w_y = eye_y - lookat_y;
    double w_z = eye_z - lookat_z;
    double wnorm = sqrt(w_x*w_x+w_y*w_y+w_z*w_z); // w vektor normalizálása
    if (wnorm > EPSILON) { w_x /= wnorm; w_y /= wnorm; w_z /= wnorm; }
    else                 { w_z = -1.0; w_x = w_y = 0.0; }

    double u_x = up_y * w_z - up_z * w_y;     // u vektor komponensei
    double u_y = up_z * w_x - up_x * w_z;
    double u_z = up_x * w_y - up_y * w_x;
    double unorm = sqrt(u_x*u_x+u_y*u_y+u_z*u_z); // u vektor normalizálása
    if (unorm > EPSILON) { u_x /= unorm; u_y /= unorm; u_z /= unorm; }
    else                 { u_x = 1.0; u_y = u_z = 0.0; }

    double v_x = w_y * u_z - w_z * u_y;       // v vektor komponensei
    double v_y = w_z * u_x - w_x * u_z;
    double v_z = w_x * u_y - w_y * u_x;

    double m[4][4];
    m[0][0]=u_x; m[0][1]=v_x; m[0][2]=w_x; m[0][3]=0.0;
    m[1][0]=u_y; m[1][1]=v_y; m[1][2]=w_y; m[1][3]=0.0;
    m[2][0]=u_z; m[2][1]=v_z; m[2][2]=w_z; m[2][3]=0.0;
    m[3][0]=0.0; m[3][1]=0.0; m[3][2]=0.0; m[3][3]=1.0;
    glMultMatrixd((double*)m);            // a kamera a -Z irányába néz
    glTranslated(-eye_x, -eye_y, -eye_z); // a szem origóba mozgatása
}
```

7.3. A perspektív transzformáció

A perspektív transzformáció célja, hogy a modellezési és a nézeti transzformációval elhelyezett virtuális világot az ablak síkjára vetítse. Az OpenGL-ben a perspektív transzformációt a `gluPerspective(`$fov, aspect, f_p, b_p$`)` függvénnyel lehet definiálni. A

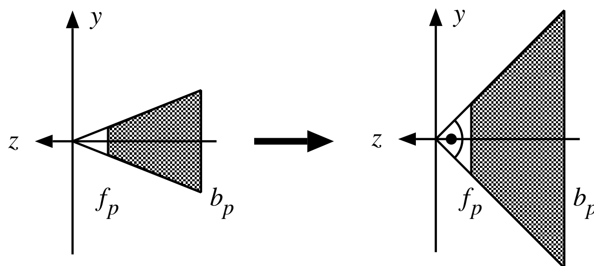
fov a kamera függőleges irányú látószögét, az $aspect$ az ablak szélességének és magasságának arányát, az f_p és a b_p pedig az úgynevezett első és hátsó vágósíkok szemtől mért távolságát jelenti (7.3. ábra).



7.3. ábra. *A* `gluPerspective()` *függvény geometriai értelmezése*

Itt álljunk meg egy pillanatra és vegyük észre, hogy az objektumtérnek csak azon részei láthatók a képen, amelyek a szem előtt, a képernyő téglalapja által meghatározott gúlában találhatók! Ez azt jelenti, hogy a szem mögötti objektumokat a képszintézis során vágással el kell távolítani. A vágási tartományt azonban az első és hátsó vágósík bevezetésével tovább korlátozhatjuk, így a képszintézisben csak azon objektumok vesznek részt, amelyek a két vágósík között helyezkednek el.

A nézeti transzformáció után a képszintézisben résztvevő pontok tartománya egy szimmetrikus csonka gúla (7.4. ábra). A további műveletekhez normalizáljuk ezt a gúlát oly módon, hogy a csúcsában a nyílásszög 90 fok legyen!



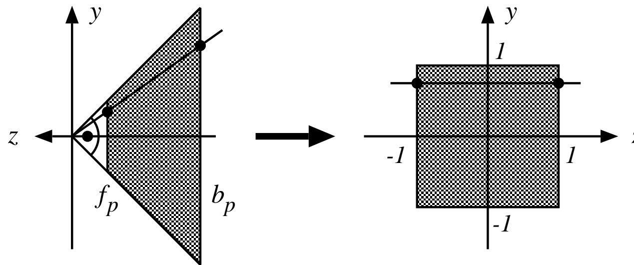
7.4. ábra. *Normalizáló transzformáció*

A normalizálás egy egyszerű skálázás:

$$
\mathbf{T}_{norm} = \begin{bmatrix} 1/(tg\,\frac{fov}{2} \cdot aspect) & 0 & 0 & 0 \\ 0 & 1/tg\,\frac{fov}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.
$$

7.3.1. Perspektív transzformáció a normalizált nézeti gúlából

A következő lépésben a csonka gúlát a perspektív vetítés szerint torzítjuk (7.5. ábra).



7.5. ábra. *Perspektív transzformáció*

A perspektív transzformációnak pontot pontba, egyenest egyenesbe kell átvinnie, ám a gúla csúcsát, azaz a szempozíciót, a végtelenbe kell elhelyeznie. Ez azt jelenti, hogy a perspektív transzformáció nem lehet az euklideszi tér lineáris transzformációja. Szerencsére a homogén lineáris transzformációkra is igaz az, hogy pontot pontba, egyenest egyenesbe visznek át, viszont képesek az ideális pontokat is kezelni. Ezért keressük a perspektív transzformációt a homogén lineáris transzformációk között a következő alakban:

$$\mathbf{T}_{persp} = \begin{bmatrix} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{34} \\ t_{41} & t_{42} & t_{43} & t_{44} \end{bmatrix}.$$

A 7.5. ábrán berajzoltunk egy egyenest és annak a transzformáltját. Jelöljük m_x-szel és m_y-nal az egyenes x- illetve y-tengely szerinti meredekségét. Ekkor a normalizált nézeti gúlában a $[-m_x \cdot z, -m_y \cdot z, z]$ egyenesből[2] a transzformáció után egy, a $[m_x, m_y, 0]$ ponton átmenő, z-tengellyel párhuzamos („vízszintes”) egyenest kapunk. Vizsgáljuk meg ezen egyenes vágósíkokkal való metszéspontjait, azaz a z helyébe helyettesítsük az $-f_p$-t illetve a $-b_p$-t! Ekkor az $[m_x, m_y, -1]$, illetve az $[m_x, m_y, 1]$ transzformált pontokhoz jutunk. Írjuk fel a perspektív transzformációt például az első vágósíkon levő metszéspontra:

$$[m_x \cdot f_p, m_y \cdot f_p, -f_p, 1] \cdot \mathbf{T}_{persp} = [m_x, m_y, -1, 1] \cdot a,$$

ahol a tetszőleges szám lehet, hisz a homogén koordinátákkal leírt pont nem változik, ha a koordinátákat egy konstanssal megszorozzuk. Az a konstanst f_p-nek választva:

$$[m_x \cdot f_p, m_y \cdot f_p, -f_p, 1] \cdot \mathbf{T}_{persp} = [m_x \cdot f_p, m_y \cdot f_p, -f_p, f_p]. \qquad (7.2)$$

[2] A negatív előjel oka az, hogy a transzformáció előtt a szem a $-z$ irányba néz.

Vegyük észre, hogy a transzformált pont első koordinátája megegyezik a metszéspont első koordinátájával tetszőleges m_x, m_y és f_p esetén! Ez csak úgy lehetséges, ha a $\mathbf{T}_{persp}$ mátrix első oszlopa $[1,0,0,0]$. Hasonló okokból következik, hogy a mátrix második oszlopa $[0,1,0,0]$. Ráadásul a 7.2. egyenletben jól látszik, hogy a vetített pont harmadik és negyedik koordinátájára a metszéspont első két koordinátája nem hat, ezért $t_{13} = t_{14} = t_{23} = t_{24} = 0$. A harmadik és a negyedik homogén koordinátára tehát a következő egyenleteket állíthatjuk fel:

$$-f_p \cdot t_{33} + t_{43} = -f_p, \qquad -f_p \cdot t_{34} + t_{44} = f_p.$$

Az egyenes hátsó vágósíkkal vett metszéspontjára ugyanezt a gondolatmenetet alkalmazva két újabb egyenletet kapunk:

$$-b_p \cdot t_{33} + t_{43} = b_p, \qquad -b_p \cdot t_{34} + t_{44} = b_p.$$

Ezt az egyenletrendszert megoldva kapjuk a perspektív transzformáció mátrixát:

$$\mathbf{T}_{persp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -(f_p+b_p)/(b_p-f_p) & -1 \\ 0 & 0 & -2\cdot f_p \cdot b_p/(b_p-f_p) & 0 \end{bmatrix}.$$

Miként behelyettesítéssel meggyőződhetünk róla, ez a transzformáció az eredetileg a szempozícióban találkozó vetítősugarakból párhuzamosakat csinál, hiszen a $[0,0,0,1]$ szempozíciót valóban a $[0,0,-2\cdot f_p \cdot b_p/(b_p-f_p),0]$ ideális pontba viszi át.

Mivel a perspektív transzformáció az euklideszi tér nemlineáris transzformációja, így a keletkező homogén koordinátanégyes negyedik koordinátája nem lesz 1 értékű. Ezért, ha a transzformáció eredményét Descartes-koordinátákban szeretnénk megkapni, akkor a negyedik homogén koordinátával végig kell osztani a többi koordinátát. A homogén osztást követően a pontok az *eszköz-koordinátarendszer*ben állnak elő.

Az alábbiakban a `gluPerspective()` függvény egy lehetséges implementációját mutatjuk be, amelyben a $\mathbf{T}_{norm}$ és a $\mathbf{T}_{persp}$ transzformációkat összevontuk:

```
//-----------------------------------------------------------------------------
void  gluPerspective(double fov, double aspect, double fp, double bp) {
//-----------------------------------------------------------------------------
    double slopey = tan(fov * M_PI / 180.0);
    double m00 = 1 / slopey / aspect, m11 = 1 / slopey;
    double m22 = -(fp + bp) / (bp - fp), m32 = -2.0 * fp * bp / (bp - fp)

    double m[4][4];
    m[0][0] = m00; m[0][1] = 0.0; m[0][2] = 0.0; m[0][3] = 0.0;
    m[1][0] = 0.0; m[1][1] = m11; m[1][2] = 0.0; m[1][3] = 0.0;
    m[2][0] = 0.0; m[2][1] = 0.0; m[2][2] = m22; m[2][3] = -1.0;
    m[3][0] = 0.0; m[3][1] = 0.0; m[3][2] = m32; m[3][3] = 0.0;
    glMultMatrixd((double*)m); // perspektív transzformáció végrehajtása
}
```

7.4. Vágás

A *vágás* célja az összes olyan objektumrészlet eltávolítása, amely nem vetülhet az ablakra, vagy amely nem az első és a hátsó vágósíkok között van. Az *átfordulási probléma* (lásd 3.2.7. fejezet) kiküszöbölése miatt, a vágást a homogén osztás előtt kell végrehajtani. A leghatékonyabb és egyben a legizgalmasabb a homogén osztást közvetlenül megelőző pillanat megragadása. Ekkor már szoroztunk a perspektív transzformációs mátrixszal, tehát a pontjaink homogén koordinátákban adottak.

A homogén koordinátás vágási határokat a képernyő-koordinátarendszerben megfogalmazott feltételek visszatranszformálásával kaphatjuk meg. A homogén osztás után a vágási határok a következők (7.5. ábra):

$$X_{\min} = -1, \;\; X_{\max} = 1, \;\; Y_{\min} = -1, \;\; Y_{\max} = 1, \;\; Z_{\min} = -1, \;\; Z_{\max} = 1.$$

A belső pontok tehát kielégítik a következő egyenlőtlenségeket:

$$-1 \leq X_h/h \leq 1, \qquad -1 \leq Y_h/h \leq 1, \qquad -1 \leq Z_h/h \leq 1. \tag{7.3}$$

Másrészt a szem előtti tartományok — a kamera-koordinátarendszerben — negatív Z_{kamera} koordinátákkal rendelkeznek, és a perspektív transzformációs mátrixszal való szorzás után a 4. homogén koordináta $h = -Z_{kamera}$ lesz, amely mindig pozitív. Tehát további követelményként megfogalmazzuk a $h > 0$ feltételt. Ekkor viszont szorozhatjuk a 7.3. egyenlőtlenségeket h-val, így eljutunk a vágási tartomány homogén koordinátás leírásához:

$$-h \leq X_h \leq h, \qquad -h \leq Y_h \leq h, \qquad -h \leq Z_h \leq h, \qquad h > 0. \tag{7.4}$$

7.4.1. Vágás homogén koordinátákkal

Pontok vágása triviális feladat, hisz a homogén koordinátás alakjukra csak ellenőrizni kell, hogy teljesülnek-e a 7.4. egyenlőtlenségek. A pontoknál összetettebb primitívekre (szakaszok, sokszögek stb.) azonban ki kell számítani a vágási tartomány határoló lapjaival való metszéspontokat, a primitívnek pedig csak azt a részét kell meghagyni, amely pontjaira a 7.4. egyenlőtlenségek fennállnak.

Szakaszok vágása

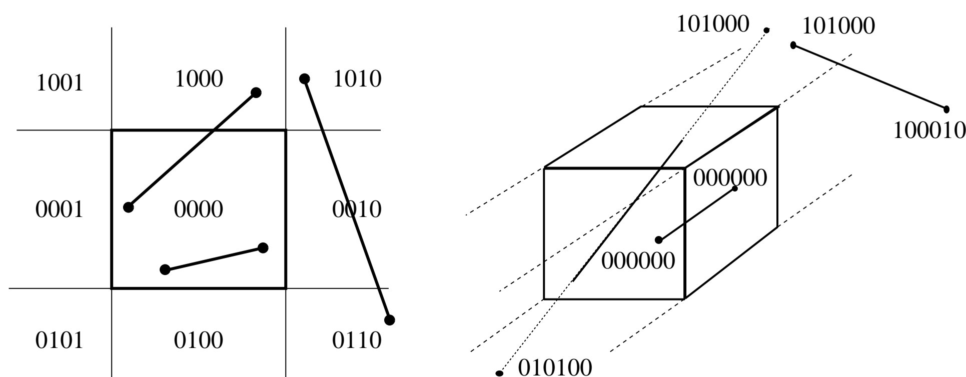
Az egyik legegyszerűbb módszer a *Cohen – Sutherland szakaszvágó algoritmus*. A módszer azon az észrevételen alapul, hogy egy vágási tartományt határoló sík a teret két féltérre osztja, így könnyen eldönthető, hogy egy adott pont és a vágási tartomány azonos, vagy ellentétes féltérben helyezkednek-e el.

Jelöljük 1-gyel, ha a pont nem a vágási tartomány félterében helyezkedik el, míg 0-val, ha azonos féltérben található! Mivel 6 határoló sík létezik, 6 darab 0 vagy 1 értékünk lesz, amelyeket egymás mellé téve egy 6-bites kódot kapunk (7.6. ábra). Egy pont $C[0], \ldots, C[5]$ kódbitjei:

$$C[0] = \begin{cases} 1, & X_h \leq -h, \\ 0, & \text{egyébként.} \end{cases} \qquad C[1] = \begin{cases} 1, & X_h \geq h, \\ 0, & \text{egyébként.} \end{cases}$$

$$C[2] = \begin{cases} 1, & Y_h \leq -h, \\ 0, & \text{egyébként.} \end{cases} \qquad C[3] = \begin{cases} 1, & Y_h \geq h, \\ 0, & \text{egyébként.} \end{cases}$$

$$C[4] = \begin{cases} 1, & Z_h \leq -h, \\ 0, & \text{egyébként.} \end{cases} \qquad C[5] = \begin{cases} 1, & Z_h \geq h, \\ 0, & \text{egyébként.} \end{cases}$$



7.6. ábra. *A tér pontjainak 6-bites kódjai*

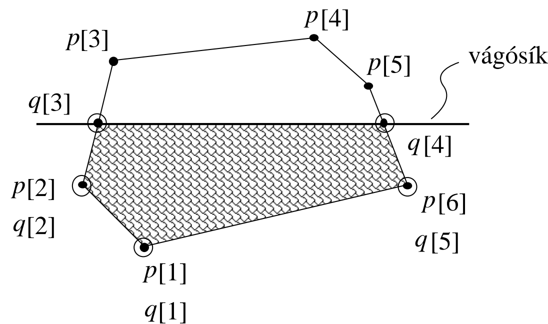
Nyilvánvalóan a 000000 kóddal rendelkező pontok a vágási tartományban, a többi pedig azon kívül találhatók (7.6. ábra). Alkalmazzuk ezt a szakaszok vágására! Legyen a szakasz két végpontjához tartozó kód C_1 és C_2! Ha mindkettő 0, akkor mindkét végpont a vágási tartományon belül van, így a szakaszt nem kell vágni. Ha a két kód ugyanazon a biten 1, akkor egyrészt egyik végpont sincs a vágási tartományban, másrészt ugyanabban a „rossz” féltérben találhatók, így az őket összekötő szakasz is ebben a féltérben helyezkedik el. Ez pedig azt jelenti, hogy nincs a szakasznak olyan része, amely „belelógna” a vágási tartományba, így az ilyen szakaszokat a további feldolgozásból ki lehet zárni. Ezt a vizsgálatot legegyszerűbben úgy végezhetjük el, hogy a C_1 és C_2 kódokon végrehajtjuk a bitenkénti ÉS műveletet, és ha az eredményül kapott kód nem nulla, akkor az azt jelenti, hogy a két kód ugyanazon a biten 1, azaz ezzel a szakasszal a továbbiakban nem kell foglalkoznunk. Egyéb esetekben van olyan vágósík, amelyre nézve az egyik végpont a belső, a másik pedig a külső („rossz”) tartományban van, így a szakaszt erre a síkra vágni kell.

Ezek alapján a Cohen – Sutherland szakaszvágó algoritmus:

```
C1 = a P1 végpont kódja;
C2 = a P2 végpont kódja;
for (;;) {
    if (C1 == 0 és C2 == 0) return true;              // a szakaszt nem kell eldobni
    if (C1 & C2 ≠ 0) return false;                     // a szakaszt el kell eldobni
    f = a legelső bit indexe, amelyen a C1 és a C2 különbözik;
    P* = a (P1, P2) szakasz és az f indexű sík metszéspontja;
    C* = a P* metszéspont kódja;
    if (C1[f] == 1) { P1 = P*; C1 = C*; }             // P1 az f. sík rossz oldalán van
    else            { P2 = P*; C2 = C*; }             // a P2 pont esik az f. sík rossz oldalára
}
```

Poligonok vágása

A poligonok vágását is 6 egymás után végrehajtott félsíkra történő vágással valósítjuk meg. A vágás során egyrészt az egyes csúcspontokat kell megvizsgálni, hogy azok belső pontok-e vagy sem. Ha egy csúcspont belső pont, akkor a vágott poligonnak is egyben csúcspontja. Ha viszont a csúcspont külső pont, nyugodtan eldobhatjuk. Másrészt vegyük észre, hogy az eredeti poligon csúcsain kívül a vágott poligonnak lehetnek új csúcspontjai is, amelyek az élek és a félsík határolóegyenesének a metszéspontjai! Ilyen metszéspont akkor keletkezhet, ha két egymást követő csúcs közül az egyik belső, míg a másik külső pont. A csúcsok egyenkénti vizsgálata mellett tehát arra is figyelni kell, hogy a következő pont a félsík tekintetében ugyanolyan típusú-e (7.7 ábra).



7.7. ábra. *Poligonvágás*

Tegyük fel, hogy az eredeti poligonunk pontjai a $p[0], \ldots, p[n-1]$ tömbben érkeznek, a vágott poligon csúcsait pedig a $q[0], \ldots, q[m-1]$ tömbbe kell elhelyezni. A vágott poligon csúcsait az m változóban számoljuk. Az implementáció során apró kellemetlenséget okoz, hogy általában az i-edik csúcsot követő csúcs az $(i+1)$-edik, kivéve az

utolsó, az $(n-1)$-edik csúcs esetében, hiszen az ezt követő a 0-adik. Ezt a kellemetlenséget elháríthatjuk, ha a p tömböt kiegészítjük még egy $(p[n] = p[0])$ elemmel, amely még egyszer tárolja a 0-adik elemet.

Ezek alapján a *Sutherland – Hodgeman poligonvágás* [112] egyetlen vágósíkra:

```
for (i = 0; i ≤ n − 1; i++) {
    if (p[i] belső pont) {
        q[m++] = p[i]                                  // az i-edik csúcs része a vágott poligonnak
        if (p[i + 1] külső pont)
            q[m++] = Intersect((p[i], p[i + 1]), félsík)   // vágással kapott új csúcspont
    } else if (p[i + 1] belső pont)
            q[m++] = Intersect((p[i], p[i + 1]), félsík)   // vágással kapott új csúcspont
}
```

A teljes vágáshoz ezt a programrészletet hatszor meg kell ismételni.

7.5. Képernyő transzformáció

A homogén osztást követően a képszintézisben résztvevő pontokat az eszköz-koordinátarendszerben kapjuk meg, amelyeket a nézeti csővezeték utolsó lépéseként a $\mathbf{T}_{viewport}$ *képernyő transzformációval* a képernyő-koordinátarendszerbe visszük át. Ha a képernyő bal-alsó sarkát (V_x, V_y) koordinátájú ponttal, méreteit V_{sx}-szel és V_{sy}-nal, a lehetséges minimális mélység értéket Z_{min}-nel, a maximálisat pedig Z_{max}-szal jelöljük, akkor az $[X_d, Y_d, Z_d]$ pontra alkalmazott képernyő transzformáció eredményeképpen a következő képernyő pontot kaphatjuk meg:

$$\begin{aligned} [X_w, Y_w] &= [(X_d + 1) \cdot \frac{V_{sx}}{2} + V_x,\ (Y_d + 1) \cdot \frac{V_{sy}}{2} + V_y], \\ Z_w &= \frac{(Z_{max} - Z_{min})}{2} \cdot Z_d + \frac{(Z_{min} + Z_{max})}{2}. \end{aligned}$$

Az OpenGL-ben a képernyő transzformációt a `glViewport(`V_x`,`V_y`,`V_{sx}`,`V_{sy}`)` és a `glDepthRange(`Z_{min}`,`Z_{max}`)` függvényekkel definiálhatjuk.

7.6. A takarási feladat megoldása

A *takarási feladat*ot megoldó algoritmusok a képernyő-koordinátarendszerben működnek, ahol két pont akkor takarja egymást, ha az (X, Y) koordinátáik megegyeznek, és az takarja a másikat, amelynek a Z koordinátája kisebb.

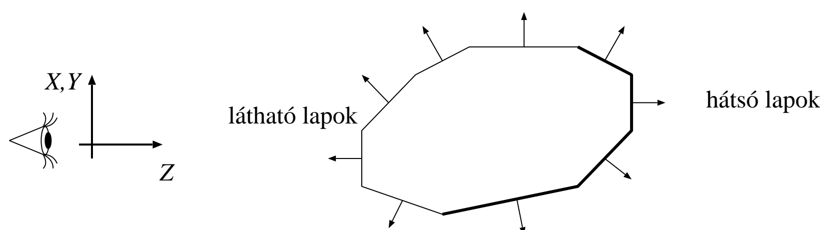
Gyakran feltételezzük, hogy a tesszelláció eredményeként az objektumok felületét alkotó sokszögek háromszögek. Ez a feltételezés nem jelent különösebb korlátozást, hiszen minden sokszög háromszögekre bontható. Feltételezzük továbbá azt is, hogy

kívülről nézve a testre, a háromszögek csúcsainak sorrendje az óramutató járásával ellentétes bejárású. Ekkor az

$$\vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1)$$

formulával minden háromszögre kiszámítható egy olyan normálvektor, amely a testből kifelé mutat.

7.6.1. Triviális hátsólap eldobás



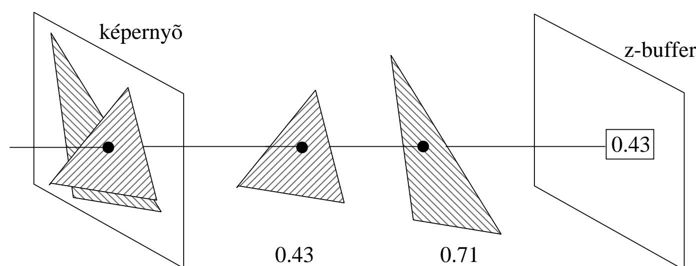
7.8. ábra. *Normálvektorok és hátsólapok*

A *triviális hátsólap eldobás* azon a felismerésen alapszik, hogy ha a képernyő-koordinátarendszerben egy lap normálvektorának pozitív Z koordinátája van, akkor ez a lap a test hátsó, nem látható oldalán foglal helyet, így eldobható. Ha az objektumtér egyetlen konvex testet tartalmaz, akkor ezzel a takarási feladatot meg is oldottuk. Bonyolultabb esetekben, azaz amikor a test nem konvex, vagy a tér több testet is tartalmaz, az első lapok is takarhatják egymást, ezért nem ússzuk meg a takarási feladatot ilyen egyszerűen. A triviális hátsólap eldobást ekkor is érdemes alkalmazni, mert ez a takarási algoritmusok által kezelendő lapok számát átlagosan a felére csökkenti.

Az OpenGL-ben a `GL_CULL_FACE` állapotváltozó jelzi, hogy eldobjuk-e a hátsólapokat vagy sem. Ez egy kétértékű változó, így a `glEnable(GL_CULL_FACE)` függvénnyel lehet bekapcsolni, a `glDisable(GL_CULL_FACE)` függvénnyel pedig kikapcsolni.

7.6.2. Z-buffer algoritmus

A *z-buffer algoritmus* a takarási feladatot az egyes pixelekre oldja meg oly módon, hogy minden pixelre megkeresi azt a sokszöget (általában háromszöget), amelynek a pixelen keresztül látható pontjának a Z koordinátája minimális (7.9. ábra). A keresés támogatására minden pixelhez, a feldolgozás adott pillanatának megfelelően tároljuk az abban látható felületi pontok közül a legközelebbi Z koordinátáját. Ezt a Z értékeket tartalmazó tömböt nevezzük *z-buffer*nek vagy *mélység-buffer*nek.



7.9. ábra. *Z-buffer algoritmus*

A sokszögeket egyenként dolgozzuk fel, és meghatározzuk az összes olyan pixelt, amely a sokszög vetületén belül van. Ehhez egy 2D sokszögkitöltő algoritmust kell végrehajtani. Amint egy pixelhez érünk, kiszámítjuk a felületi pont Z koordinátáját és összehasonlítjuk a z-bufferben lévő, az adott pixelhez tartozó mélységértékkel. Ha az ott található érték kisebb, akkor a már feldolgozott sokszögek között van olyan, amelyik az aktuális sokszöget ebben a pontban takarja, így az aktuális sokszög ezen pontját nem kell megrajzolni. Ha viszont a z-bufferbeli érték nagyobb, akkor ebben a pontban az aktuális sokszög az eddig feldolgozott sokszögeket takarja, ezért ennek a színét kell beírni az aktuális pixelbe és egyúttal a Z értékét a z-bufferbe.

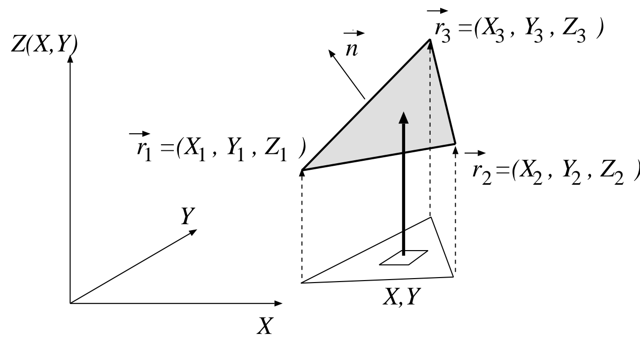
A z-buffer algoritmus tehát:

```
raszter memória = háttér szín;
zbuffer = ∞;
for (minden o sokszögre) {
    for (o sokszög vetületének minden p pixelére) {
        if (az o sokszög p-ben látható pontjának Z koordinátája < zbuffer[p]) {
            p színe = o színe ebben a pontban;
            zbuffer[p] = o sokszög p pixelben látható pontjának Z koordinátája;
        }
    }
}
```

A z-buffer ∞-nel történő inicializálása ténylegesen a lehetséges legnagyobb Z érték használatát jelenti. Az algoritmus részleteinek bemutatása során feltesszük, hogy az objektumok háromszögek, és az adott pillanatban az

$$\vec{r}_1 = [X_1, Y_1, Z_1], \qquad \vec{r}_2 = [X_2, Y_2, Z_2], \qquad \vec{r}_3 = [X_3, Y_3, Z_3]$$

csúcspontokkal definiált háromszöget dolgozzuk fel. A raszterizációs algoritmusnak elő kell állítania a háromszög vetületébe eső X, Y pixel címeket a Z koordinátákkal együtt (7.10. ábra).



7.10. ábra. *Egy háromszög a képernyő-koordinátarendszerben*

Az X,Y pixel címből a megfelelő Z koordinátát a háromszög síkjának egyenletéből származtathatjuk, azaz a Z koordináta az X,Y koordináták valamely lineáris függvénye. A háromszög síkjának az egyenlete:

$$\vec{n} \cdot [X,Y,Z] = C, \qquad \text{ahol } \vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1), \qquad C = \vec{n} \cdot \vec{r}_1.$$

A normálvektor koordinátáit $[n_X, n_Y, n_Z]$-vel jelölve, ebből a $Z(X,Y)$ függvény:

$$Z(X,Y) = \frac{C - n_X \cdot X - n_Y \cdot Y}{n_Z}. \tag{7.5}$$

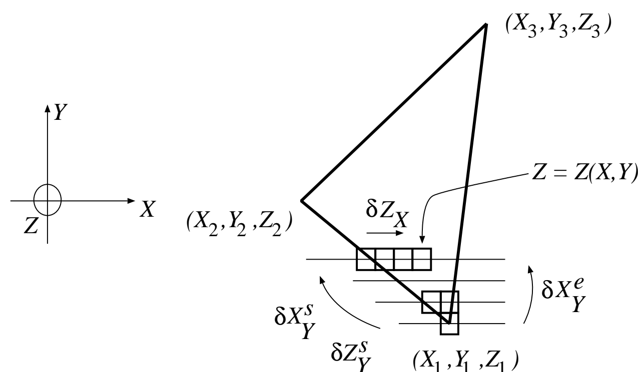
Az inkrementális elv felhasználásával ezen képlet jelentősen egyszerűsíthető:

$$Z(X+1,Y) = Z(X,Y) - \frac{n_X}{n_Z} = Z(X,Y) + \delta Z_X. \tag{7.6}$$

A δZ_X paraméter állandó az egész háromszögre, ezért csak egyszer kell kiszámítani. Egyetlen pásztán belül a Z koordináta kiszámítása tehát egyetlen összeadást igényel.

Mivel a Z és az X lineárisan változik a háromszög bal és jobb éle között, ezért a határvonal mentén a pászták kezdeti Z és X koordinátája is egyetlen összeadással számítható a megelőző pászta kezdeti Z és X koordinátájából. Sőt, a pászták végső X koordinátája is hasonlóan számítható (7.11. ábra):

$$\begin{aligned}
X_{\text{start}}(Y+1) &= X_{\text{start}}(Y) + \frac{X_2 - X_1}{Y_2 - Y_1} = X_{\text{start}}(Y) + \delta X_Y^s, \\
X_{\text{end}}(Y+1) &= X_{\text{end}}(Y) + \frac{X_3 - X_1}{Y_3 - Y_1} = X_{\text{end}}(Y) + \delta X_Y^e, \\
Z_{\text{start}}(Y+1) &= Z_{\text{start}}(Y) + \frac{Z_2 - Z_1}{Y_2 - Y_1} = Z_{\text{start}}(Y) + \delta Z_Y^s.
\end{aligned}$$



7.11. ábra. *Inkrementális elv a z-buffer számításoknál*

Ezek alapján a z-buffer algoritmus inkrementális megvalósítása egy háromszög alsó felére (a felső részre hasonló program készíthető):

```
X_start = X_1 + 0.5; X_end = X_1 + 0.5; Z_start = Z_1 + 0.5;
for (Y = Y_1; Y ≤ Y_2; Y++) {
    Z = Z_start;
    for (X = (int)(X_start); X ≤ (int)(X_end); X++) {
        z = (int)(Z);
        if (z < zbuffer[X][Y]) { SetPixel(X, Y, szín); zbuffer[X][Y] = z; }
        Z += δZ_X;
    }
    X_start += δX^s_Y; X_end += δX^e_Y; Z_start += δZ^s_Y;
}
```

Az OpenGL-ben a z-buffer használatát a `GL_DEPTH_TEST` állapotváltozó beállításával irányíthatjuk. Mivel ez is egy kétértékű változó, ezért a z-buffer használatát szintén a `glEnable()` és a `glDisable()` függvénypárossal engedélyezhetjük, illetve tilthatjuk.

7.7. Árnyalás

A takarási algoritmusok a képernyő-koordinátarendszerben minden pixelre meghatározzák az ott látható sokszöget. A hátralévő feladat az adott pixelben látható felületi pont színének kiszámítása, amelyet az OpenGL a következő *árnyalási egyenlet*tel (4.8.8. fejezet) határoz meg:

$$L = L^e + k_a \cdot L^a + \sum_l a_l \cdot s_l \left[k_a \cdot L_l^a + k_d \cdot \cos\theta_l' \cdot L_l^d + k_s \cdot \cos^n \delta_l \cdot L_l^s \right],$$

ahol L^e a felületi pont által kibocsátott intenzitás, $k_a \cdot L^a$ az *ambiens tag*, amely a többszörös visszaverődések elhanyagolásának kompenzálására szolgál, az illuminációs képlet utolsó tagja pedig az absztrakt fényforrásokból érkezett, majd a felület által a kamera irányába vert fényerősséget adja meg. Az OpenGL az árnyalási egyenletet a kamerakoordinátarendszerben értékeli ki.

Az ambiens tagot a `glLightModel()` függvénnyel írhatjuk elő, amelynek első paraméterként a `GL_LIGHT_MODEL_AMBIENT` konstanst kell megadni, míg a második paraméterként a virtuális világra jellemző ambiens színt kell RGBA színrendszerben definiálni. Az árnyalási egyenlet további tagjaival a következő alfejezetekben ismerkedünk meg.

7.7.1. Fényforrások

Az OpenGL-ben egy fényforrás lehet pontszerű, iránnyal adott és szpotlámpa. Az a_l próbálja kifejezi azt a tényt, hogy ha egy pontszerű fényforrástól vagy egy szpotlámpától távolodunk, akkor az általa kibocsátott fény erőssége csökken, amelynek mértékét a következő képlettel számolhatjuk ki:

$$a_l = \frac{1}{(k_{0,l} + k_{1,l} \cdot d_l + k_{2,l} \cdot d_l^2)}, \tag{7.7}$$

ahol d_l a felületi pont és az l. absztrakt fényforrás távolsága, $k_{0,l}$ a konstans, $k_{1,l}$ a lineáris, $k_{2,l}$ pedig a négyzetes lecsengési tényező. Irány-fényforrás esetén az a_l lecsengési tényező értéke 1, tehát a fény erőssége nem csökken.

A szpotlámpa által kibocsátott fény erőssége nemcsak a távolsággal, hanem a lámpa fő sugárzási irányától mérhető eltéréssel is csökken, hatása pedig a hatóterületén kívül teljesen megszűnik. Az s_l tényező a szpotlámpák esetén ezt a szögtől függő lecsengést szabályozza, amely a lámpa hatóterületén belül a $\cos^m \alpha$ képlettel számolható ki, ahol α a szpotlámpa fő iránya és a kibocsátott fény iránya közötti szög, m pedig a lecsengés „sebessége”. Az s_l tényező értéke a lámpa hatóterületén kívül 0, azonban pontszerű és irány-fényforrás esetén mindentől függetlenül konstans 1.

Az OpenGL az l. fényforrás által kibocsátott fényt az L_l^a ambiens, az L_l^d diffúz és az L_l^s spekuláris komponensekre bontja[3]. Ennek megfelelően a $k_a \cdot L_l^a$ a felületi pont által a kamera felé vert ambiens, a $k_d \cdot \cos\theta_l' \cdot L_l^d$ a diffúz, a $k_s \cdot \cos^n \delta_l \cdot L_l^s$ pedig a spekuláris fény intenzitását jelenti. A θ' a beérkező fénysugár és a felületi ponthoz tartozó normálvektor által bezárt szöget, míg a δ a normálvektor és a felezővektor közötti szöget jelöli (4.6. ábra).

[3]Megjegyezzük, hogy a fényforrások által kibocsátott fény ambiens, diffúz és spekuláris komponensekre bontásának nincs fizikai alapja. Az OpenGL tervezőinek ezzel a megoldással valószínűleg az lehetett a célja, hogy a programozó a létrehozható hatásokat szabadabban állíthassa be.

Az OpenGL 8 fényforrást kezel, amelyekre a GL_LIGHT0, ..., GL_LIGHT7 neveken hivatkozhatunk. Egy fényforrás tulajdonságait a glLight() függvénycsaláddal definiálhatjuk. A fényforrás által kibocsátott fény három komponensét a GL_AMBIENT, a GL_DIFFUSE és a GL_SPECULAR paraméterekkel állíthatjuk be. A pontszerű fényforrás és a szpotlámpa pozícióját a GL_POSITION paraméter után homogén koordinátás alakban kell megadnunk. Ugyanezzel a paraméterrel írhatjuk le az irány-fényforrás sugárzási irányát is, mégpedig úgy, hogy a homogén koordinátás alak negyedik koordinátájának a 0 értéket adjuk. A 7.7. képlet $k_{0,l}$ konstans lecsengési tényezőjét a GL_CONSTANT_ATTENUATION, a $k_{1,l}$ lineárist a GL_LINEAR_ATTENUATION, míg a fényforrás $k_{2,l}$ négyzetes lecsengési sebességét a GL_QUADRATIC_ATTENUATION paraméter után állíthatjuk be. A szpotlámpa fő sugárzási irányát a GL_SPOT_DIRECTION, a hatóterületének szögét a GL_SPOT_CUTOFF, míg a lámpa szögtől függő lecsengésének „sebességét" a GL_SPOT_EXPONENT paraméter után határozhatjuk meg.

A következő példában egy pontszerű fényforrás tulajdonságai jelennek meg:

```
float LightAmbient[] = {0.1, 0.1, 0.1, 1.0};    // ambiens RGBA
float LightDiffuse[] = {0.5, 0.2, 0.3, 1.0};    // diffúz RGBA
float LightPosition[] = {1.0, 2.0, 3.0, 1.0};   // pozíció (x,y,z,h)

glLightfv(GL_LIGHT0, GL_AMBIENT, LightAmbient);   // az ambiens komponens
glLightfv(GL_LIGHT0, GL_DIFFUSE, LightDiffuse);   // a diffúz komponens
glLightfv(GL_LIGHT0, GL_POSITION, LightPosition); // a lámpa pozíciója
glEnable(GL_LIGHT0);                              // bekapcsoljuk a lámpát
```

7.7.2. Anyagok

A felületek anyagtulajdonságait a glMaterial() függvénycsaláddal definiálhatjuk. A glMaterial() első paramétereként azt kell megadni, hogy a definiálandó anyagot a felület elülső (GL_FRONT), hátsó (GL_BACK) vagy mindkét (GL_FRONT_AND_BACK) oldalára kívánjuk-e alkalmazni. Ez azt jelenti, hogy ha a felületre szemből nézünk, akkor az anyagtulajdonság hatását a felületnek csak a felénk eső (elülső), csak a másik (hátsó), vagy mindkét oldalán szeretnénk érzékelni. A függvény második és harmadik paramétere pedig egy anyagtulajdonságot ír le. A GL_EMISSION paraméter után a felület által kibocsátott fény intenzitását (L^e) határozhatjuk meg. A felület ambiens fényvisszaverő képességét (k_a) a második paraméterként megadott GL_AMBIENT, a diffúzt (k_d) a GL_DIFFUSE, a spekuláris tényezőt (k_s) a GL_SPECULAR, a fényességét (n) pedig a GL_SHININESS után írhatjuk le. Lehetőség van az ambiens és a diffúz fényvisszaverő képességet egyszerre állítani a GL_AMBIENT_AND_DIFFUSE paraméter segítségével. A következő példaprogramban egy piros anyagot adunk meg:

```
const float RedSurface[] = {1.0, 0.0, 0.0, 1.0}; // (R=1, B=G=0, A=1)
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, RedSurface);
```

7.7.3. Árnyalási módok

Az árnyalási egyenlet megoldása során gyakran érdemes a feladatot pixeleknél nagyobb egységekben kezelni, azaz kihasználni, hogy ha a szomszédos pixelekben ugyanazon felület látszik, akkor ezen pixelekben látható felületi pontok optikai paraméterei, normálvektora, megvilágítása, sőt, végső soron akár a látható színe is igen hasonló. Tehát vagy változtatás nélkül használjuk a szomszédos pixelekben végzett számítások eredményeit, vagy pedig az inkrementális elv alkalmazásával egyszerű formulákkal tesszük azokat aktuálissá az új pixelben. A következőkben ilyen módszereket ismertetünk.

Saját színnel történő árnyalás

A *saját színnel történő árnyalás* a háromdimenziós képszintézis árnyalási módszerének direkt alkalmazása. Előnye, hogy nem igényel semmiféle illuminációs számítást, viszont a keletkezett képeknek sincs igazán háromdimenziós hatásuk (7.28/2. ábra).

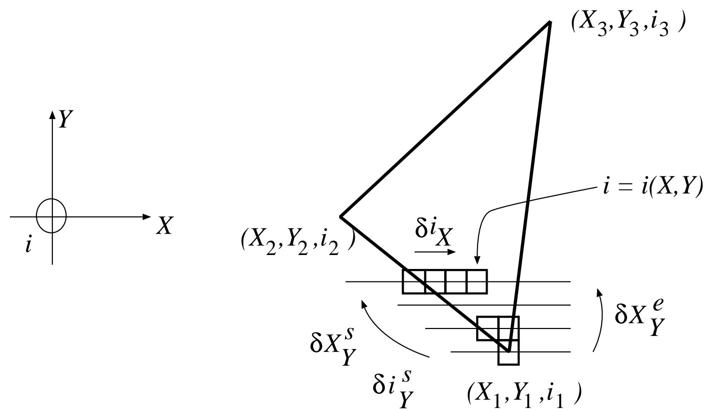
Konstans árnyalás

A *konstans árnyalás* a sokszögekre csak egyszer számítja ki az absztrakt fényforrások hatását. Amennyiben valamelyik pixelben a sokszög látszik, akkor mindig ezzel a konstans színnel jelenítjük meg. Az eredmény általában elég lesújtó, mert a képről ordít, hogy a felületeket sík sokszögekkel közelítettük (7.28/3. ábra).

Gouraud-árnyalás

A *Gouraud-árnyalás* a háromszögek csúcspontjaiban értékeli ki a fényforrásokból odajutó fény visszaverődését. Az illuminációs képlet alkalmazásánál az eredeti felület normálvektorával dolgozik, azaz a tesszellációs folyamat során a kiadódó pontokban a normálvektort is meg kell határozni, amelyet a sokszögháló visz magával a modellezési transzformációk során. Ezután a Gouraud-árnyalás a háromszög belső pontjainak színét a csúcspontok színéből lineárisan interpolálja (7.28/4. ábra).

A z-buffer inkrementális megvalósításánál bemutatott levezetést (7.6.2. fejezet) a Gouraud-árnyalás esetében is alkalmazhatjuk, így az árnyalni kívánt háromszög határvonala mentén a pászták kezdeti R, G, B színértéke és X koordinátája is egyetlen összeadással számítható a megelőző pászta kezdeti színéből és X koordinátájából. Ráadásul itt is érvényes, hogy a pászták végső X koordinátája is hasonlóan számítható (7.12. ábra).



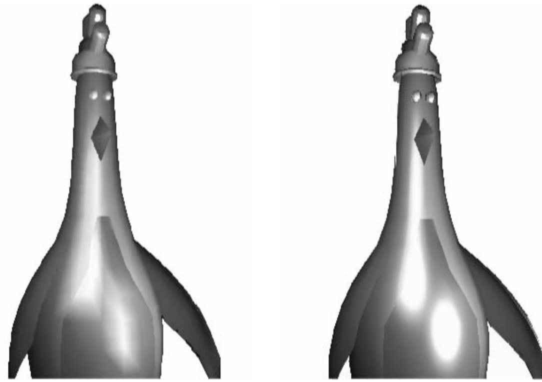
7.12. ábra. *Inkrementális elv a Gouraud-árnyalásnál*

Ezek alapján a Gouraud-árnyalás programja, amely egy háromszög alsó felét színezi ki (a felső részre hasonló program készíthető):

```
Xstart = X1 + 0.5;   Xend = X1 + 0.5;
Rstart = R1 + 0.5;   Gstart = G1 + 0.5;   Bstart = B1 + 0.5;
for (Y = Y1; Y ≤ Y2; Y++) {
      R = Rstart;   G = Gstart;   B = Bstart;
      for (X = (int)(Xstart); X ≤ (int)(Xend); X++) {
            SetPixel(X,Y,(int)(R),(int)(G),(int)(B));
            R += δRX;   G += δGX;   B += δBX;
      }
      Xstart += δX^s_Y;   Xend += δX^e_Y;
      Rstart += δR^s_Y;   Gstart += δG^s_Y;   Bstart += δB^s_Y;
}
```

Az OpenGL grafikus könyvtárban a `glShadeModel()` függvénnyel lehet az árnyalás módját beállítani. A konstans árnyalást a `glShadeModel(GL_FLAT)`, a Gouraud-árnyalást pedig a `glShadeModel(GL_SMOOTH)` függvényhívással lehet bekapcsolni.

A Gouraud-árnyalás akkor jó, ha a háromszögön belül a szín valóban közelítőleg lineárisan változik. Ez nagyjából igaz diffúz visszaverődésű objektumokra, de elfogadhatatlan tükrös, illetve spekuláris visszaverődésű felületekre. A lineáris interpoláció ezekben az esetekben egyszerűen kihagyhatja vagy szétkenheti a fényforrás tükröződő foltját (7.13. ábra). Ezt a problémát a következő fejezet algoritmusával, a Phong-árnyalással oldhatjuk meg.



7.13. ábra. *Egy csirke Gouraud- (balra) és Phong-árnyalással (jobbra)*

Phong-árnyalás

A *Phong-árnyalás* az árnyalási egyenletben szereplő, a fényforrás és a kamera irányába mutató egységvektorokat, illetve a normálvektort interpolálja a háromszög csúcspontjaiban érvényes adatokból, az árnyalási egyenletet pedig minden pixelre külön értékeli ki (7.28. ábra). A műveleteket ezen vektorok világ-koordinátarendszerbeli koordinátáin kell végrehajtani. Az alábbiakban a Phong-árnyalás egyszerűsített programját mutatjuk be, amely csak a normálvektort interpolálja:

```
X_start = X_1 + 0.5;  X_end = X_1 + 0.5;
for (Y = Y_1; Y ≤ Y_2; Y++) {
      N⃗ = N⃗_start;
      for (X = (int)(X_start); X ≤ (int)(X_end); X++) {
            (R,G,B) = ShadingModel(Normalize(N⃗));            // árnyalási egyenlet
            SetPixel(X,Y,(int)(R),(int)(G),(int)(B));
            N⃗ += δN_X;
      }
      X_start += δX_Y^s,   X_end += δX_Y^e;
      N⃗_start += δN⃗_Y^s;
}
```

A Phong-árnyalás a színtérben nemlineáris interpolációnak felel meg, így nagyobb sokszögekre is megbirkózik a tükrös felületek gyorsan változó sugársűrűségével (7.13. ábra). A Phong-árnyalás a Gouraud-árnyalás olyan határeseteként is elképzelhető, amikor a tesszelláció finomításával a sokszögek vetített területe a pixelek méretével összevethető.

7.8. Program: Egyszerű színtér megjelenítése

Az eddig elmondottakat „váltsuk aprópénzre” és jelenítsünk meg egy piros golyót, amelyhez használjuk fel a 2. fejezetben bemutatott keretrendszert! Első lépésként származtassunk az `Application` osztályból egy újat és `RedBallRenderer`nek nevezzük el! Ez az osztály vezérli az alkalmazást. Az `Init()` metódusban engedélyezzük a z-buffert, beállítjuk az árnyalási módot és fényforrást veszünk fel:

```
//-----------------------------------------------------------------
void RedBallRenderer::Init() {
//-----------------------------------------------------------------
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // bufferek törlése
  glEnable(GL_DEPTH_TEST); // a z-buffer algoritmus bekapcsolása
  glEnable(GL_LIGHTING);   // a megvilágítás számításának bekapcsolása

  // a 0. indexű absztrakt fényforrás megadása és üzembe helyezése
  float LightAmbient[] = {0.1, 0.1, 0.1, 1.0};
  float LightDiffuse[] = {0.5, 0.5, 0.5, 1.0};
  float LightPosition[] = {5.0, 5.0, 5.0, 0.0};
  glLightfv(GL_LIGHT0, GL_AMBIENT, LightAmbient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, LightDiffuse);
  glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
  glEnable(GL_LIGHT0);
}
```

A `Render()` metódusban megadjuk a perspektív transzformációhoz szükséges információkat, illetve a világ-koordinátarendszerben elhelyezzük a kamerát és a gömböt:

```
//-----------------------------------------------------------------
void RedBallRenderer::Render() {
//-----------------------------------------------------------------
  glViewport(0, 0, windowWidth, windowHeight); // képernyő transzformáció
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();          // a perspektív transzformáció alaphelyzete

  float aspect = (windowHeight == 0) ? 1.0 : windowWidth / windowHeight;
  gluPerspective(45, aspect, 1, 100); // perspektív transzformáció

  glMatrixMode(GL_MODELVIEW);   // a kamera elhelyezése, nézeti transzformáció
  glLoadIdentity();
  gluLookAt(2.0, 3.0, 4.0,      // szem pozíció
            0.0, 0.0, 0.0,      // nézeti referencia pont
            0.0, 1.0, 0.0);     // felfelé irány

  // modellezési transzformáció: (-2.0, -2.0, -3.0) vektorral való eltolás
  glTranslatef(-2.0, -2.0, -3.0);

  const float RedSurface[] = {1.0, 0.0, 0.0, 1.0}; // piros anyag
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, RedSurface);

  // egységnyi sugarú, 40x40 síklappal közelített gömb létrehozása
  GLUquadricObj*  sphere = gluNewQuadric();
  gluSphere(sphere, 1.0, 40, 40);
}
```

7.9. Stencil buffer

A későbbiekben számos példát fogunk látni, amelyben a virtuális világot egymás után többször is le kell fényképeznünk. Az OpenGL lehetőséget ad arra, hogy egy ilyen megjelenítési menet közben a képernyő felbontásával azonos méretű *stencil buffer*ben megjelöljük, hogy a következő menetekben pontosan melyik pixeleket akarjuk használni. Például a repülőgép-szimulátorokban a műszerfal képe nem függ a pillanatnyi pozíciótól, ezért az ahhoz tartozó pixeleket nem érdemes minden egyes megjelenítési menetben újra kiszámolni és kirajzolni.

A stencil-teszt a z-buffer használata előtt történik meg. A stencil buffer működését — a z-bufferhez hasonlóan — ki és be lehet kapcsolni. A teszt eredménye megmondja, hogy egy adott pixelben egy sokszöget használhatunk-e a z-buffer algoritmus során vagy sem. Ha egy adott pixelnél egy sokszögre a teszt nem sikerül, akkor a sokszöget a pixelhez tartozó további műveletekből kizárjuk. Az OpenGL-ben a stencil-teszt bekapcsolását a `glEnable(GL_STENCIL_TEST)` függvényhívással lehet elérni.

A teszt egy adott pixelhez tartozó stencil bufferbeli számot egy referencia értékkel hasonlít össze. Ezenkívül megadható egy bitmaszk is, amellyel kijelöljük, hogy a tesztnek az összehasonlításkor mely biteket kell kezelnie. Ha r_v a referencia érték, s_v a stencil bufferben levő érték, b_m pedig a bitmaszk, akkor például a `GL_LESS` konstanssal hívott összehasonlító függvény csak akkor sikerül, ha $(r_v \ \& \ b_m) < (s_v \ \& \ b_m)$. Ezen az elven működnek a `GL_LEQUAL`, a `GL_EQUAL`, a `GL_GEQUAL`, a `GL_GREATER` és a `GL_NOTEQUAL` paraméterrel megadott stencil függvények is, csak a két érték összehasonlításakor a $\leq$, az $=$, a $\geq$, a $>$, illetve a $\neq$ műveleteket hajtják végre. A `GL_NEVER` paraméter segítségével előírható, hogy a teszt sohase sikerüljön, míg a `GL_ALWAYS` konstans használatakor az összehasonlítás mindig pozitív válasszal tér vissza.

A stencil-teszt során alkalmazandó összehasonlító stratégiát a `glStencilFunc()` függvénnyel adhatjuk meg. A következő példában a referencia értéket 1-re, a bitmaszkot pedig `0xff`-re állítjuk be, és az összehasonlításkor akkor kérünk pozitív választ, ha a két érték azonos, azaz ha a stencil bufferben is 1-es van:

```
glStencilFunc(GL_EQUAL, 0x1, 0xff);
```

A z-buffer algoritmus használatakor kétféle döntés születhet: a sokszöget az adott pixelben a Z értéke alapján nem rajzoljuk ki, vagy pedig mélységértékét beírjuk a z-bufferbe, színértékét pedig a rasztertárba. Mivel a stencil-teszt megelőzi a z-buffer használatát, így a két buffer minden pixelre együttesen háromféle eredményt adhat:

- a stencil-teszt sikertelen,
- a stencil-teszt sikerül, de a z-buffer nem engedi a rajzolását,
- a stencil és a z-buffer alapú teszt is sikerül.

Az OpenGL-ben — a `glStencilOp()` függvény segítségével — mindhárom eredményhez megmondhatjuk, hogy a stencil bufferbeli értékkel mit tegyen. Ha egy kimeneti eseményhez a `GL_KEEP` módszert rendeljük, akkor ezen esemény bekövetkezésekor a stencil bufferbeli érték változatlanul marad. Ha a `GL_ZERO` eljárást használjuk, akkor az esemény hatására a stencil bufferbeli érték nullázódik. A `GL_REPLACE` módszer esetében a stencil érték helyébe a referencia értéket írjuk. A `GL_INCR` eljárás a stencil értéket eggyel növeli, míg a `GL_DECR` eggyel csökkenti. A `GL_INVERT` módszer az esemény bekövetkezésekor a stencil bufferben levő értéket bitenként invertálja.

A következő példában arról rendelkezünk, hogy ha a stencil-teszt sikertelen, akkor a buffer értékét lenullázzuk. Ha a stencil buffer alapú teszt sikerül, de a z-buffer alapú nem, akkor eggyel csökkentjük, egyébként pedig az értéket változatlanul hagyjuk:

```
glStencilOp(GL_ZERO, GL_DECR, GL_KEEP);
```

7.10. Átlátszóság

Az OpenGL képes átlátszó objektumokat is kezelni. Az anyag tulajdonságait négy színcsatornán írjuk le: a szokásos R,G,B színhármast még egy negyedik, A-val jelölt, „alfa"-taggal egészítjük ki, amelynek jelentése *átlátszatlanság* (*opacitás*). Ennek 1 értéke teljesen átlátszatlan színt, 0 értéke pedig teljesen átlátszó színt jelöl. Az átlátszó színekkel való rajzolás azt jelenti, hogy amikor az OpenGL egy új pixelértéket írna be a rasztertárba, akkor nem egyszerűen felülírja a korábbi, tárolt értéket, hanem az új érték és a tárolt érték valamilyen súlyozott átlagát képezi, és az eredményt teszi vissza a rasztertárba. Ezt az *összemosás* (*blending*) műveletet a `glEnable(GL_BLEND)` hívással lehet engedélyezni, és a `glDisable(GL_BLEND)` hívással tiltani. Jelöljük az új (*forrás* illetve *source*) színnégyest R_s, G_s, B_s, A_s-sel, a rasztertárban tárolt (*cél* illetve *destination*) értéket pedig R_d, G_d, B_d, A_d-vel! Az összemosó művelet az

$$R = r_s \cdot R_s + r_d \cdot R_d, \quad G = g_s \cdot G_s + g_d \cdot G_d, \quad B = b_s \cdot B_s + b_d \cdot B_d, \quad A = a_s \cdot A_s + a_d \cdot A_d$$

eredményt a $[0,1]$ intervallumra vágás után írja a rasztertárba. Az r_s, g_s, b_s, a_s és az r_d, g_d, b_d, a_d súlyozó-tényezők a `glBlendFunc(source,destination)` első és második paraméterével állíthatók be. A forrásra alkalmazható súlyozási lehetőségeket a 7.1. táblázatban foglaltuk össze.

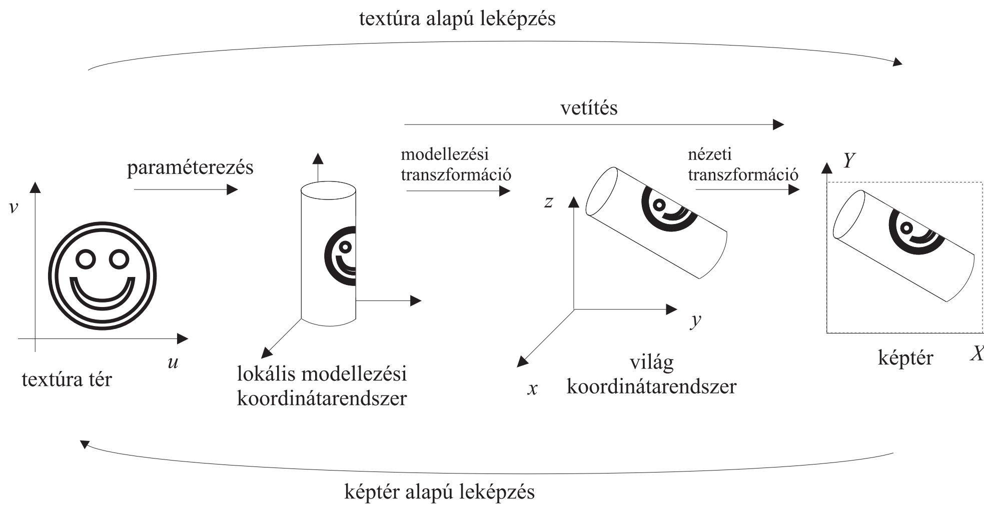
A célértékre (*destination*) a lehetőségek hasonlóak, a képletekben a forrásra és a célra vonatkozó indexek és a DST, illetve SRC szavak értelemszerűen szerepet cserélnek.

`glBlendFunc()` paraméter	hatás
GL_ZERO	$r = g = b = a = 0,$
GL_ONE	$r = g = b = a = 1,$
GL_DST_COLOR	$r = R_d,\ \ g = G_d,\ \ b = B_d,\ \ a = A_d$
GL_ONE_MINUS_DST_COLOR	$r = 1 - R_d,\ \ g = 1 - G_d,\ \ b = 1 - B_d,\ \ a = 1 - A_d$
GL_SRC_ALPHA	$r = g = b = a = A_s,$
GL_ONE_MINUS_SRC_ALPHA	$r = g = b = a = 1 - A_s,$
GL_DST_ALPHA	$r = g = b = a = A_d,$
GL_ONE_MINUS_DST_ALPHA	$r = g = b = a = 1 - A_d,$
GL_SRC_ALPHA_SATURATE	$r = g = b = a = \min(A_s, 1 - A_d),$

7.1. táblázat. *A `glBlendFunc()` lehetséges `source` paraméterei*

7.11. Textúra leképzés

Mivel az árnyalási egyenletben szereplő BRDF nem szükségképpen állandó a felületen, hanem pontról pontra változhat, ezért a finom részletek megjelenítéséhez textúrákat használunk, ahelyett, hogy a felületek geometriáját túlságosan bonyolítanánk. A textúrákat általában a textúratérben adjuk meg, amelyet a felület pontjaival a paraméterezés kapcsol össze (4.9.1. fejezet).



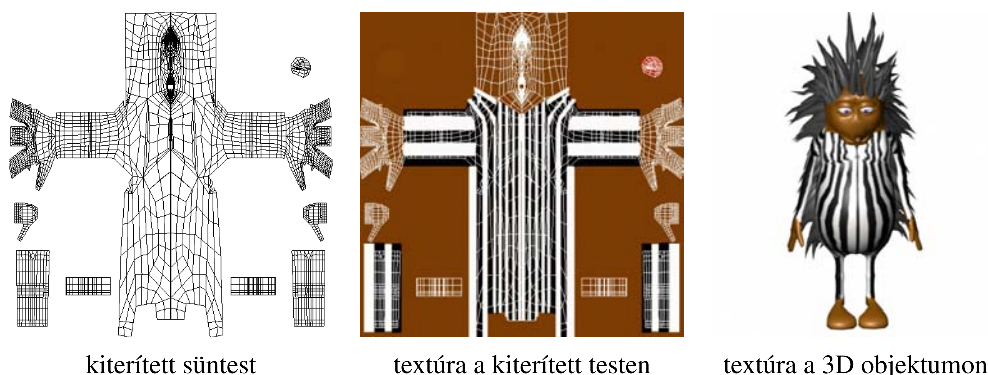
7.14. ábra. *Textúra leképzés*

Bemutattuk, hogy a modellezési transzformáció a modellezési-koordinátarendszert a világ-koordinátarendszerbe viszi át. Innen az inkrementális képszintézis módszerek továbblépnek a képernyő-koordinátarendszerbe a takarási probléma egyszerűsítésének érdekében. A világ-koordinátarendszerből a pixelekhez vezető transzformációt *vetítés*nek nevezzük (7.14. ábra).

A pixelek és a textúratérbeli pontok közötti kapcsolat bejárására két lehetőségünk van:

1. A *textúra alapú leképzés* a textúratérben lévő ponthoz keresi meg a hozzá tartozó pixelt.
2. A *képtér alapú leképzés* a pixelhez keresi meg a hozzá tartozó textúra elemet.

A textúra alapú leképzés általában hatékonyabb, de alapvető problémája, hogy nem garantálja, hogy textúra térben egyenletesen kijelölt pontok képei a képernyőn is egyenletesen helyezkednek el. Így előfordulhat, hogy nem minden érintett pixelt színezünk ki, vagy éppenséggel egy pixel színét feleslegesen túl sokszor számoljuk ki. A képtér alapú leképzés jól illeszkedik az inkrementális képtér algoritmusok működéséhez, viszont használatához elő kell állítani a paraméterezési és a vetítési transzformációk inverzét, ami korántsem könnyű feladat.



kiterített süntest — textúra a kiterített testen — textúra a 3D objektumon

7.15. ábra. *A textúrázás a modell kiterítése*

Mivel a paraméterezés és a vetítés is homogén lineáris transzformáció, a textúrateret a képtérrel összekötő szorzatuk is az. A paraméterezést mátrixokkal felírva:

$$[x\cdot h, y\cdot h, z\cdot h, h] = [u, v, 1]\cdot \mathbf{P}_{3\times 4}. \quad (7.8)$$

A modellezési és nézeti transzformáció együttesen a következő transzformációt jelenti:

$$[X\cdot q, Y\cdot q, Z\cdot q, q] = [x\cdot h, y\cdot h, z\cdot h, h]\cdot \mathbf{T}_{V(4\times 4)}. \quad (7.9)$$

Ha a vetítést a képtérben hajtjuk végre, akkor az a Z koordinátát egyszerűen csak elhagyja, így ezt nem is érdemes a textúra leképzéshez kiszámítani. Mivel a $\mathbf{T}_{V(4\times4)}$ harmadik oszlopa felelős a Z koordináta kiszámításáért, a mátrix ezen oszlopát törölhetjük, így $\mathbf{T}_{V(4\times3)}$-mal is dolgozhatunk:

$$[X\cdot q, Y\cdot q, q] = [x\cdot h, y\cdot h, z\cdot h, h]\cdot \mathbf{T}_{V(4\times3)} = [u,v,1]\cdot \mathbf{P}_{3\times4}\cdot \mathbf{T}_{V(4\times3)}. \quad (7.10)$$

Jelöljük a $\mathbf{P}_{3\times4}\cdot \mathbf{T}_{V(4\times3)}$ szorzatot $\mathbf{C}_{3\times3}$-mal, így a paraméterezés és a vetítés kompozíciója:

$$[X\cdot q, Y\cdot q, q] = [u,v,1]\cdot \mathbf{C}_{\mathbf{3}\times\mathbf{3}}. \quad (7.11)$$

Az egyes koordinátákra (c_{ij} a $\mathbf{C}_{3\times3}$ mátrix (i,j)-edik eleme):

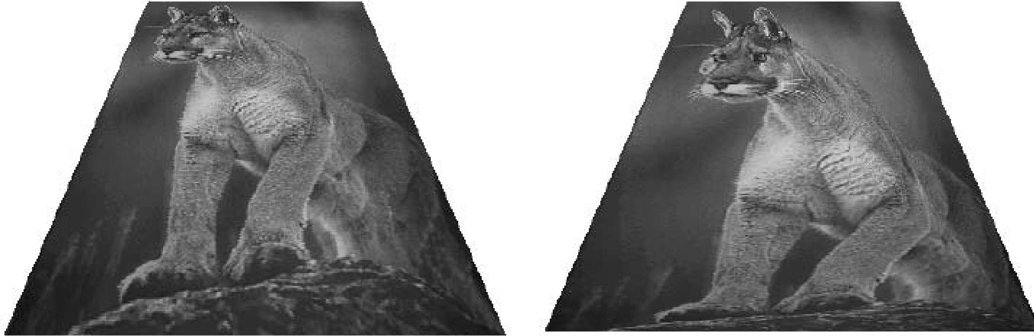
$$X(u,v) = \frac{c_{11}\cdot u + c_{21}\cdot v + c_{31}}{c_{13}\cdot u + c_{23}\cdot v + c_{33}}, \qquad Y(u,v) = \frac{c_{12}\cdot u + c_{22}\cdot v + c_{32}}{c_{13}\cdot u + c_{23}\cdot v + c_{33}}.$$

Az inverz transzformáció pedig (C_{ij} a $\mathbf{C}^{-1}_{3\times3}$ mátrix (i,j)-edik eleme)

$$[u\cdot w, v\cdot w, w] = [X,Y,1]\cdot \mathbf{C}^{-1}_{3\times3}.$$

$$u(X,Y) = \frac{C_{11}\cdot X + C_{21}\cdot Y + C_{31}}{C_{13}\cdot X + C_{23}\cdot Y + C_{33}}, \quad v(X,Y) = \frac{C_{12}\cdot X + C_{22}\cdot Y + C_{32}}{C_{13}\cdot X + C_{23}\cdot Y + C_{33}}.$$

A fenti egyenletek nevezője a perspektív transzformáció homogén osztásának elvégzéséből adódik. Ez azt jelenti, hogy a képernyő-koordinátarendszer pontjait a textúra térre perspektív korrekcióval sikerült leképezni. Ha a képletet egyszerűsítjük, és a nevezőt egy konstans értékkel közelítjük, akkor ugyan nem kell pixelenként osztani, de a textúra nem a perspektívának megfelelően fog az objektumra illeszkedni (7.16. ábra).



7.16. ábra. *Textúra leképzés perspektív korrekcióval (balra) és anélkül (jobbra)*

A képtér algoritmusok során alkalmazott inverz textúra leképzést az inkrementális elv alkalmazásával tehetjük még hatékonyabbá. Adott Y koordináta esetén legyen az $u(X)$ tényezőt meghatározó hányados számlálója $uw(X)$, a nevezője pedig $w(X)$! Az $u(X+1)$-et az $u(X)$-ből két összeadással és egyetlen osztással számíthatjuk a következő képlet alkalmazásával:

$$uw(X+1) = uw(X) + C_{11}, \quad w(X+1) = w(X) + C_{13}, \quad u(X+1) = \frac{uw(X+1)}{w(X+1)}.$$

Hasonló összefüggések érvényesek a v koordinátára is. Az inkrementális elvet azonban nemcsak a textúra koordináták pásztán belüli számítására lehet használni, hanem a képtérbeli háromszög kezdőélére is.

A textúra leképzés inkrementális megvalósítása a Phong-árnyalásnál bemutatott sémát követi:

```
X_start = X_1 + 0.5;  X_end = X_1 + 0.5;  N⃗_start = N⃗_1;
uw_s = uw_1;  vw_s = vw_1;  w_s = w_1;
for (Y = Y_1; Y ≤ Y_2; Y++) {
    N⃗ = N⃗_start;  uw = uw_s;  vw = vw_s;  w = w_s;
    for (X = (int)(X_start); X ≤ (int)(X_end); X++) {
        u = uw/w;  v = vw/w;
        (R, G, B) = ShadingModel(Normalize(N⃗), u, v);
        SetPixel(X, Y, (int)(R), (int)(G), (int)(B));
        N⃗ += δN⃗_X;  uw += C_11;  vw += C_12;  w += C_13;
    }
    X_start += δX^s_Y;  X_end += δX^e_Y;  N⃗_start += δN⃗^s_Y;
    uw_s += δuw^s_Y;  vw_s += δvw^s_Y;  w_s += δw^s_Y;
}
```

7.12. Textúra leképzés az OpenGL-ben

Először az OpenGL 1.1-es specifikációjában jelentek meg a *textúra objektum*ok, amelyek a textúrákat és azok tulajdonságait tárolják. A textúra objektumok bevezetésével a textúra leképzés az OpenGL-ben két feladatra, a textúra definiálásra, és annak paraméterezésére bontható.

7.12.1. Textúra definíció

Ahhoz, hogy egy textúrát képszintézisre tudjunk használni, először egy azonosító számot kell kérni a `glGenTextures()` függvénnyel. Az OpenGL terminológiájában ez a pozitív szám a textúra objektum „neve”, ugyanis ezzel tudjuk „megnevezni”, hogy melyik textúra objektumhoz szeretnénk hozzáférni.

A *k* azonosítójú textúra objektumot a `glBindTexture()` függvény hozza létre, amikor először hívjuk meg a *k* értékkel. Ha a *k* egy már létező textúra objektum indexe, akkor az OpenGL — követve a rajzolási állapot elvét — a *k* azonosítójú textúra objektumot teszi aktívvá. A textúrák tulajdonságmódosító függvényei mindig az aktív textúra objektumra vannak hatással. Egy textúra objektumot a `glDeleteTextures()` függvénnyel lehet megszüntetni.

A textúra definíció következő állomása a paraméterezésnél használt ismétlési, illetve a megjelenítésnél fontos szűrési információk megadása a `glTexParameter()` függvény segítségével.

Végül a `glTexImage2D(target,level,internalFormat,width,height, border,format,type,pixels)` függvénnyel a bittérképes textúrát hozzárendeljük a textúra objektumhoz. A függvény első paramétere — bittérképes textúrák esetén — mindig `GL_TEXTURE_2D`. A `level` paramétert a textúrák szűrésénél használjuk (részletesebben lásd 7.13. fejezet). Az `internalFormat` kijelöli, hogy a textúra *R*,*G*,*B*,*A* komponensei közül melyeket kívánjuk használni. A textúra szélességét a `width`, míg a magasságát a `height` segítségével adhatjuk meg. A `border` paraméterrel a határsáv használatát engedélyezhetjük, illetve tilthatjuk (részletesebben lásd 7.13.1. fejezet). A `format` és a `type` a textúra tárolási mechanizmusát adják meg. Végül a `pixels` tömbben adjuk át a textúra képpontjait, az úgynevezett *texel*eket.

Az OpenGL csak olyan négyzet alakú textúrákat tud kezelni, amelyek szélessége és magassága 2 hatvány. Ha a textúraként használni kívánt kép nem ilyen méretekkel rendelkezik, akkor a `gluScaleImage()` függvénnyel átméretezhetjük azt.

A következő `Texture` osztály létrehoz egy bittérképes textúrát:

```
//===============================================
class Texture {
//===============================================
public:
  unsigned int texture_id;
  void MakeTexture(int width, int height, const GLvoid* pixels) {
    // generáltatunk egy új textúra objektum indexet
    glGenTextures(1, &texture_id);

    // létrehozzuk a textúra objektumot, amelyhez 2D textúrát fogunk rendelni
    glBindTexture(GL_TEXTURE_2D, texture_id);

    // bekapcsoljuk a vízszintes és függőleges irányú ismétlést, arra az esetre,
    // ha a textúra a textúrázandó tárgyhoz képest túl kicsi lenne
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    // definiáljuk a bittérképes textúrát
    glTexImage2D(GL_TEXTURE_2D, 0, 3,
                 width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, pixels);
  }
};
```

7.12.2. Textúrák és a megvilágítás kombinálása

Eddig a textúra leképzés azt jelentette, hogy egy felület színét helyettesítettük a textúrából érkező színinformációval. Lehetőség van arra is, hogy a textúrában tárolt színeket összemossuk a felület megvilágításból adódó színével. Ehhez a `glTexEnv()` függvénnyel a környezeti paramétereket kell megfelelően definiálnunk (7.2. táblázat). A függvény első paraméterének kötelezően a `GL_TEXTURE_ENV` értéket kell adni, míg a második paramétere `GL_TEXTURE_ENV_MODE` és `GL_TEXTURE_ENV_COLOR` is lehet. Utóbbi esetén a harmadik paraméter az összemosásnál az R,G,B,A értékekkel adott C_c színt definiálja. A `GL_TEXTURE_ENV_MODE` esetén a függvény utolsó paramétere azt adja meg, hogy az OpenGL a textúrákat és a megvilágításból adódó színeket hogyan kombinálja. A 7.2. táblázatban ezen kombinációkat foglaltuk össze az R,G,B és az R,G,B,A színértékekkel adott textúrák esetén. A táblázatban C_t a textúrából, C_f pedig a megvilágításból származó szín, míg C a kombinált színérték. Hasonlóan A_t a textúrából, A_f a megvilágításból származó átlátszatlanság, míg A a kombinált érték.

`glTexEnv()` paraméter	RGB textúra esetén	RGBA textúra esetén
GL_BLEND	$C = C_f \cdot (1 - C_t) + C_c \cdot C_t$ $A = A_f$	$C = C_f \cdot (1 - C_t) + C_c \cdot C_t$ $A = A_f \cdot A_t$
GL_MODULATE	$C = C_f \cdot C_t$ $A = A_f$	$C = C_f \cdot C_t$ $A = A_f \cdot A_t$
GL_REPLACE	$C = C_t$ $A = A_f$	$C = C_t$ $A = A_t$

7.2. táblázat. *A textúrák és a megvilágítás lehetséges kombinálásai*

7.12.3. Paraméterezés

A paraméterezés során azt a leképzést definiáljuk, amely a 2D textúra értelmezési tartományát, azaz az $(u,v) \in [0,1]^2$ pontjait hozzárendeli a 3D tárgy (x,y,z) felületi pontjaihoz. A 4.9.1. fejezetben ismertetett paraméterezési módszerek közül az OpenGL a sokszögek paraméterezését és a gömbi vetítést ismeri. Az OpenGL-ben az u koordinátát s-sel, a v komponenst pedig t-vel jelöljük. A paraméterezés előtt a textúrázást engedélyezzük, az árnyalást kikapcsoljuk és az adott textúrát kiválasztjuk:

```
glEnable(GL_TEXTURE_2D); // bekapcsoljuk a 2D textúrákat

// a képszintéziskor csak a textúrából érkező színinformációt
// használjuk, az objektumok saját színét nem vesszük figyelembe
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

// a paraméterezésnél használni kívánt textúra objektum kijelölése
glBindTexture(GL_TEXTURE_2D, texName);
```

Sokszögek paraméterezése

Az OpenGL-ben a virtuális világ objektumait általában sokszögekkel definiáljuk, amelyek háromszögek vagy négyszögek lehetnek. Egy sokszög paraméterezését a csúcspontjaihoz tartozó textúra koordinátákkal adjuk meg, az OpenGL pedig a sokszög belsejében lineáris interpolációt alkalmaz. A csúcspontokhoz tartozó textúra koordinátákat a `glTexCoord2...()` függvényekkel írjuk le, mindig a kapcsolódó csúcspont előtt. A következő példában egy háromszöget paraméterezünk:

```
glBegin(GL_TRIANGLES);
   glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
   glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
glEnd();
```

Gömbfelületek paraméterezése

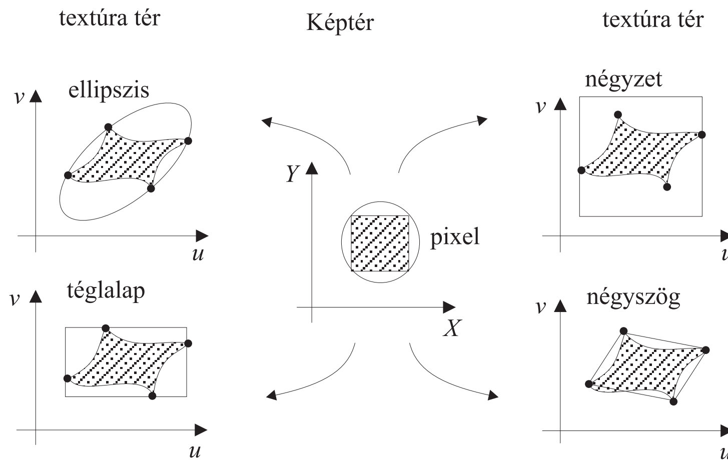
Gömbfelületek paraméterezése esetén a textúra koordináták egyenkénti megadása igen körülményes feladat, ám elkerülhető, ha az s és t koordináták automatikus számítását a `glTexGen...()` függvénycsalád segítségével bekapcsoljuk. Az alábbi példában egy golyót teszünk ki, amelyre a bittérképes textúrát gömbi vetítéssel helyezzük rá:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);

GLUquadricObj*  sphere = gluNewQuadric();
gluSphere(sphere, 1.0, 40, 40);
```

7.13. A textúrák szűrése

A textúratér és a képernyő-koordinátarendszer közötti leképzés a textúratér egyes részeit nagyíthatja, más részeit pedig összenyomhatja. Az előbbi esetben *nagyítás*ról, az utóbbiban pedig *kicsinyítés*ről beszélünk. Ez azt jelenti, hogy a képernyőtérben egyenletes sűrűséggel kiválasztott pixel középpontok igen egyenlőtlenül mintavételezhetik a textúrát, amely végső soron problémákat okozhat. Ezért a textúra leképzésnél a mintavételezési problémák elkerülését célzó szűrésnek különleges jelentősége van.

A textúra szűrés nehézsége abból fakad, hogy a textúratér és a képtér közötti leképzés nemlineáris. Például ha doboz szűrést szeretnénk alkalmazni, azaz a pixel textúratérbeli képében kívánjuk a texeleket átlagolni, akkor szabálytalan, általános görbék által határolt területtel kell dolgoznunk. A szokásos eljárások ezt az általános területet egyszerű területekkel, például ellipszissel, négyszöggel, téglalappal vagy négyzettel közelítik (7.17. ábra).



7.17. ábra. *A pixel ősképének közelítése*

Négyzettel történő közelítés esetén egyetlen pixel színét úgy határozhatjuk meg, hogy megkeressük a pixel sarokpontjainak megfelelő textúratérbeli pontokat, előállítjuk a négy pontot tartalmazó legkisebb négyzetet, majd átlagoljuk a négyzetben lévő texelek színeit.

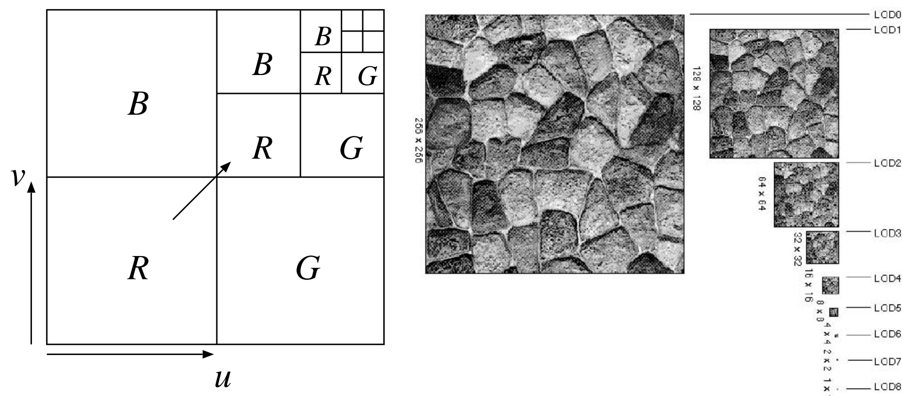
Az OpenGL lehetőséget ad arra, hogy a `glTexParameter()` függvénnyel más szűrő eljárást definiáljunk a nagyítás (`GL_TEXTURE_MAG_FILTER`) esetére, mint a kicsinyítésre (`GL_TEXTURE_MIN_FILTER`). Az OpenGL szűrésre két módszert biztosít. Ha a `GL_NEAREST` eljárást választjuk, akkor a megjelenítésre csak a pixel középpontjához legközelebbi texelt használja. Ha azonban a `GL_LINEAR` módszert alkalmazzuk, akkor a rendszer a pixel középpontjához legközelebbi 2×2 texel súlyozott átlagát jeleníti meg. Természetesen az első módszer gyorsabb, ám csak a legközelebbi texel figyelembevétele észrevehető mintavételezési problémákat okozhat. Ezzel szemben a `GL_LINEAR` módszer nagyobb ráfordítással szebb végeredményt ad.

A következő példában a szűrő eljárások használatát mutatjuk be:

```
// nagyításnál a GL_LINEAR eljárást használjuk
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

// kicsinyítésnél pedig a GL_NEAREST módszert alkalmazzuk
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Ha egy textúrázott objektumhoz közelebb megyünk, akkor gyakran megjelenítési hibákat vehetünk észre. Ezek a problémák abból fakadnak, hogy csak egy rögzített méretű textúrát használunk, amelyet a mozgás hatására az OpenGL az objektum méreteinek megfelelően próbál megjeleníteni. Például ha egy falra egy kis textúrát teszünk fel, majd a falhoz nagyon közel megyünk, akkor a texeleket külön-külön felismerhetjük. Ha a

falhoz közeledve a rögzített méretű textúrát egyre nagyobb felbontásúval cserélnénk le, akkor a textúra kép véges felbontása kevésbé lenne észrevehető. Ezt a problémát a *piramis*ok használatával oldhatjuk meg, amelyek a textúrát (általános esetben egy képet) több felbontásban tárolják. Két egymást követő textúra felbontásának aránya egy a kettőhöz. Az egymást követő képeket egy képpiramisként is elképzelhetjük, amelyben a legnagyobb felbontású kép a piramis alján, a legkisebb felbontású pedig a piramis tetején foglal helyet. A textúraképeket általában *mip-map adatstruktúrába* szervezik[4] [140] (7.18. ábra).



7.18. ábra. *A textúratár mip-map szervezése*

A képpiramis használatához az OpenGL-nek az összes kettő hatvány méretű textúrát át kell adni, a legnagyobb felbontásútól az 1×1-es méretűig. Tehát ha a legnagyobb felbontású textúránk 16×16 texelből áll, akkor elérhetővé kell tenni a textúra 8×8-as, 4×4-es, 2×2-es és 1×1-es méretű változatait is. A képpiramis különböző szintjein levő textúrákat a `glTexImage2D()` függvénnyel definiáljuk. A következő példában egy képpiramist adunk meg:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 16, 16, 0, // a 16x16-os textúra
             GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage16);
glTexImage2D(GL_TEXTURE_2D, 1, GL_RGBA, 8, 8, 0,   // a 8x8-as textúra
             GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage8);
glTexImage2D(GL_TEXTURE_2D, 2, GL_RGBA, 4, 4, 0,   // a 4x4-es textúra
             GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage4);
glTexImage2D(GL_TEXTURE_2D, 3, GL_RGBA, 2, 2, 0,   // a 2x2-es textúra
             GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage2);
glTexImage2D(GL_TEXTURE_2D, 4, GL_RGBA, 1, 1, 0,   // az 1x1-es textúra
             GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage1);
```

[4]A mip-map kifejezés mip prefixe a latin *multim im parvo* rövidítése, amely magyarul kb. annyit tesz, hogy „sok dolog kis helyen”, a map pedig angolul térképet jelent.

A kisebb felbontású képeket a legnagyobból szűréssel hozzuk létre úgy, hogy veszünk 4 szomszédos texelt és átlagoljuk őket. Mivel a képpiramis létrehozás igen fontos feladat, a GLU könyvtárban találunk is rá megoldást: egy nagyfelbontású textúrából a `gluBuild2DMipmaps()` függvénnyel a kívánt képsorozat elkészíthető. A következő példában egy 32×32 felbontású képből hozzuk létre a képpiramishoz szükséges textúrákat:

```
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, 32, 32,
                  GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage32);
```

7.13.1. Határsáv

Az OpenGL-ben a textúrák maximális mérete korlátozott. Ha mégis ennél nagyobb textúrát szeretnénk használni, akkor azt — egy puzzle darabjaihoz hasonlóan — több elegendően kisfelbontású részre kell szétszedni, és azokat egymás mellé téve, de külön kell megjeleníteni. Ám a dolog nem ennyire egyszerű, mert ilyenkor textúra szűrés a darabkák széleinél nem venné figyelembe a szomszédos darabkák színinformációit. A probléma megoldása, hogy minden textúra darabkához megadunk egy pixelnyi szélességű határsávot is, amelyben a színek megegyeznek a kapcsolódó darabkák szélein levő színekkel. Ezen határsávok alapján az OpenGL a textúra szűrést már a kívánalmaknak megfelelően végzi el. Egy textúra határsávját a `glTexImage2D()` függvény 6. paraméterével jelölhetjük ki. Ha ennek értéke 0, akkor a textúrának nincs határsávja. Ha viszont a 6. paraméter 1, akkor a textúrához tartozik egy pixel szélességű határsáv. A 7.12.1. fejezetben szereplő `Texture` osztály által létrehozott textúrák nem rendelkeznek határsávval.

7.14. Multitextúrázás

A textúra leképzésben egy sokszöghöz akár több textúrát is rendelhetünk. A helikopter-szimulátorokban általában hegyes-völgyes terep fölött kell az ellenfelekkel csatáznunk. Ha egy ilyen játékban változatos színvilágú terepet szeretnénk létrehozni, akkor az egyik lehetőségünk az, hogy rengeteg textúrával dolgozunk, amelyeket a terep különböző részeihez rendelünk. A másik lehetőség az, hogy csak néhány textúrát használunk a terep alapszínének megadására, a változatosság eléréséhez pedig úgynevezett *részlet térkép*ek (*detail map*) segítségével kicsit módosítunk a terep alapszínein.

A legegyszerűbb megoldás, hogy ha egy objektumot n különböző textúrával szeretnénk megjeleníteni, akkor a textúrák leképzését n megjelenítési menetben egyesével definiáljuk, majd a keletkezett képeket összemossuk. Természetesen ekkor a képszintézis időigénye jelentősen megnő. A következő oldalon található példában ezt a stratégiát követve rendelünk egy négyzethez két textúrát.

```
// kiválasztjuk az első textúrához tartozó textúra objektumot
glBindTexture(GL_TEXTURE_2D, texName[0]);
glBegin(GL_QUADS);
   glTexCoord2d(0.0, 0.0); glVertex3d(0.0, 0.0, 0.0);
   glTexCoord2d(1.0, 0.0); glVertex3d(1.0, 0.0, 0.0);
   glTexCoord2d(1.0, 1.0); glVertex3d(1.0, 1.0, 0.0);
   glTexCoord2d(0.0, 1.0); glVertex3d(0.0, 1.0, 0.0);
glEnd();

glEnable(GL_BLEND); // összeszorozzuk a két textúrát
glBlendFunc(GL_ZERO, GL_SRC_COLOR);

// kiválasztjuk a második textúrához tartozó textúra objektumot
glBindTexture(GL_TEXTURE_2D, texName[1]);

// a négyzetet a második textúrával paraméterezzük
glBegin(GL_QUADS);
   glTexCoord2d(0.0, 0.0); glVertex3d(0.0, 0.0, 0.0);
   glTexCoord2d(1.0, 0.0); glVertex3d(1.0, 0.0, 0.0);
   glTexCoord2d(1.0, 1.0); glVertex3d(1.0, 1.0, 0.0);
   glTexCoord2d(0.0, 1.0); glVertex3d(0.0, 1.0, 0.0);
glEnd();

glDisable(GL_BLEND); // kikapcsoljuk az összemosást
```

Míg a fenti példában a két textúra használatához két megjelenítési menet szükséges, a *multitextúrázás* több textúrát egyetlen lépésben képes megjeleníteni. Az OpenGL 1.2.1-es verziójában jelent meg a `GL_ARB_multitexture` kiegészítés, amely interfészt adott a multitextúrázáshoz. Mivel a kiegészítéseket általában nem minden OpenGL implementáció ismeri, ezért a tervezők a multitextúrázást az 1.3-as verziótól kezdve a grafikus könyvtár részévé tették.

Multitextúrázás esetén egyszerre több textúrakezelő egység dolgozik párhuzamosan és a kimeneti képeik megfelelő összemosása adja meg a képszintézis végeredményét. A textúrakezelő egységekből összesen `GL_MAX_TEXTURE_UNITS_ARB` darab használható, amelyek között a `glActiveTextureARB()` függvénnyel válthatunk. A `GL_MAX_TEXTURE_UNITS_ARB` konstans értéke implementációtól függő, ám legalább kettőnek kell lennie. A `glMultiTexCoord2...ARGB()` függvényekkel lehet a textúra koordinátákat definiálni.

A következő példában megint két textúrát rendelünk ugyanahhoz a négyzethez, ám most multitextúrázással:

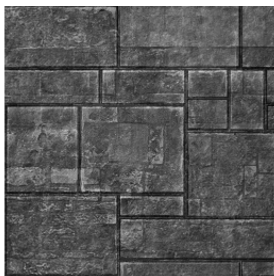
```
// a multitextúrázásnál az első textúrát is használjuk
glActiveTextureARB(GL_TEXTURE0_ARB);
glBindTexture(GL_TEXTURE_2D, texName[0]);

// a multitextúrázásnál a második textúrát is használjuk
glActiveTextureARB(GL_TEXTURE1_ARB);
glBindTexture(GL_TEXTURE_2D, texName[1]);


// a négyzetet mindkét textúrával paraméterezzük
```

```
glBegin(GL_QUADS);
   glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 0.0);
   glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0, 0.0);
   glVertex3d(0.0, 0.0, 0.0);
   glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 0.0);
   glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0, 0.0);
   glVertex3d(1.0, 0.0, 0.0);
   glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 1.0);
   glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0, 1.0);
   glVertex3d(1.0, 1.0, 0.0);
   glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 1.0);
   glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0, 1.0);
   glVertex3d(0.0, 1.0, 0.0);
glEnd();
```

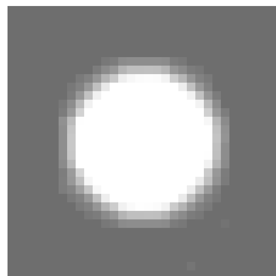
7.15. Fénytérképek

A megvilágítás kiszámítása jóval több időt igényel, mint egy textúra leképzés. Ha a virtuális világban csak diffúz felületek és statikus fényforrások találhatók, akkor a nem mozgó objektumok megvilágítását egy előfeldolgozási lépésben is kiszámolhatjuk és sokszögenként egy-egy 2D textúrában el is tárolhatjuk azt. Az ilyen textúrákat *fénytérkép*eknek nevezzük.

Egy adott fénytérkép a hozzá rendelt sokszögről visszavert fényt mintavételezi és tárolja. Ekkor az árnyaláskor elegendő a fénytérkép megfelelő indexén levő intenzitás értéket megjeleníteni. Ha az előfeldolgozási lépésben elég pontos módszert használunk (például valamilyen globális illuminációs eljárást (8. fejezet)), akkor a fénytérképekkel megjelenített világ valószerűbbnek hat, mint egy Gouraud-árnyalással készített kép.



fal textúra



fénytérkép

multitextúrázott eredmény

7.19. ábra. *Fénytérképek alkalmazása*

Ha a fénytérképek mellett a sokszögekhez további textúrákat szeretnénk hozzárendelni, akkor a 7.14. fejezetben ismertetett módszereket használhatjuk. Sőt, azt is mondhatjuk, hogy a fénytérképek és más textúrák közös megjelenítése a multitextúrázás talán legjellemzőbb felhasználási területe (7.19. ábra).

A módszer egyik legfőbb hibája abból adódik, hogy a megvilágítást csak kétdimenzióban tároljuk. Így ha egy felületre különböző irányokból nézünk rá, mindig ugyanazt a megvilágítást látjuk. Ez csak a diffúz felületek jellegzetessége, tehát spekuláris vagy tükrös felületek esetében a fénytérképek hibás eredményt adnak. Először a Quake 3-ban jelentek meg az úgynevezett *irányított fénytérkép*ek, amelyek a sokszögek különböző irányokba visszavert fényét tárolják.

7.16. Bucka leképzés

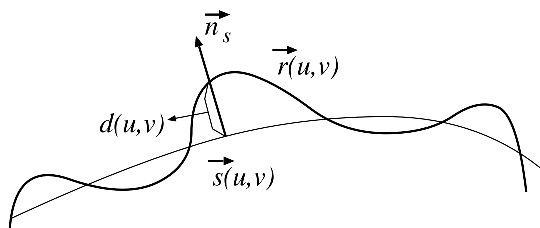
A felületi normálvektor alapvető szerepet játszik a BRDF definíciókban. Hepehupás felületek, mint például a kráterekkel tarkított bolygók, sötétebb, illetve világosabb foltokkal rendelkeznek amiatt, hogy a buckákon a normálvektor és a fényforrás által bezárt szög eltérhet az átlagos megvilágítási szögtől. A hepehupás felületek geometriai modellel történő leírása igen nehéz és keserves feladat lenne, nem beszélve a bonyolult geometrián dolgozó takarási feladat megoldásának szörnyűségeiről. Szerencsére létezik egy módszer, amely lényegesen egyszerűbb, ugyanakkor távolról szemlélve a hatás tekintetében a geometriai modellekétől nem marad el lényegesen. A módszer, amelyet *bucka leképzés*nek (*bump-mapping*) nevezünk, a textúra leképzéshez hasonló, de most nem a BRDF valamely elemét, hanem a normálvektornak a geometriai normálvektortól való eltérését tároljuk külön táblázatban. A transzformációs, takarási stb. feladatoknál egyszerű geometriával dolgozunk (a Holdat például gömbnek tekintjük), de az árnyalás során a geometriából adódó normálvektort még perturbáljuk a megfelelő táblázatelemmel [22]. Tegyük fel, hogy a buckákat is tartalmazó felület az $\vec{r}(u,v)$ egyenlettel, míg az egyszerű geometriájú közelítése az $\vec{s}(u,v)$ egyenlettel definiálható! Az $\vec{r}(u,v)$-t kifejezhetjük úgy is, hogy a sima felületet a normálvektorának irányában egy kis $d(u,v)$ eltolással, azaz egy mikro magasságmezővel módosítjuk (7.20. ábra).

Az $\vec{s}(u,v)$ felület $\vec{n}_s$ normálvektorát a felület $(\vec{s}_u, \vec{s}_v)$ parciális deriváltjainak vektoriális szorzataként is kifejezhetjük, azaz $\vec{n}_s = \vec{s}_u \times \vec{s}_v$, amiből az egységnyi hosszú $\vec{n}_s^0$ normálvektorhoz normalizálás után jutunk. Így a buckás felület egyenlete:

$$\vec{r}(u,v) = \vec{s}(u,v) + d(u,v) \cdot \vec{n}_s^0.$$

A buckás felület normálvektorához először az $\vec{r}(u,v)$ parciális deriváltjait képezzük:

$$\vec{r}_u = \vec{s}_u + d_u \cdot \vec{n}_s^0 + d \cdot \frac{\partial \vec{n}_s^0}{\partial u}, \qquad \vec{r}_v = \vec{s}_v + d_v \cdot \vec{n}_s^0 + d \cdot \frac{\partial \vec{n}_s^0}{\partial v}.$$



7.20. ábra. *A buckák leírása*

Az utolsó tagok elhanyagolhatók, hiszen mind a $d(u,v)$ eltolás, mind pedig a sima felület normálvektorának változása kicsiny:

$$\vec{r}_u \approx \vec{s}_u + d_u \cdot \vec{n}_s^0, \qquad \vec{r}_v \approx \vec{s}_v + d_v \cdot \vec{n}_s^0.$$

A buckás felület normálvektorát a két derivált vektoriális szorzataként kapjuk:

$$\vec{n}_r = \vec{r}_u \times \vec{r}_v = \vec{s}_u \times \vec{s}_v + d_u \cdot \vec{n}_s^0 \times \vec{s}_v + d_v \cdot \vec{s}_u \times \vec{n}_s^0 + d_u d_v \cdot \vec{n}_s^0 \times \vec{n}_s^0.$$

Az utolsó tagban $\vec{n}_s^0$ önmagával vett vektoriális szorzata szerepel, ami azonosan zérus. Ezen kívül használhatjuk a következő helyettesítéseket:

$$\vec{s}_u \times \vec{s}_v = \vec{n}_s, \qquad \vec{n}_s^0 \times \vec{s}_v = \vec{t}, \qquad \vec{s}_u \times \vec{n}_s^0 = \vec{b}.$$

A $\vec{t}$ és $\vec{b}$ vektorok a felület érintősíkjában vannak. A buckás felület normálvektora:

$$\vec{n}_r = \vec{n}_s + d_u \cdot \vec{t} + d_v \cdot \vec{b}.$$

A $d(u,v)$ eltolásfüggvényt fekete-fehér képként (magasságmezőként) tároljuk, amelyet *bucka térkép*nek nevezünk. A buckás felület normálvektora az eltolásfüggvény deriváltjait tartalmazza, amelyet véges differenciákkal közelíthetünk. Ha a B bucka tábla egy $N \times N$ méretű kép, akkor a közelítő deriváltak:

$U = (\text{int})(u * (N-3) + 1); \quad V = (\text{int})(v * (N-3) + 1);$
$d_u(u,v) = (B[U+1,V] - B[U-1,V]) \cdot N/2;$
$d_v(u,v) = (B[U,V+1] - B[U,V-1]) \cdot N/2;$

A fenti módszer hibája, hogy a buckák látszólagos relatív mérete változik, ha a felületet skálázzuk, vagy nyíró transzformációval módosítjuk. Ezen segíthetünk, ha $\vec{n}_s$, $\vec{t}$ és $\vec{b}$ vektorok helyett egységnyi hosszú, és egymásra merőleges $\vec{n}_s^0$, $\vec{T} = -\vec{s}_u^0$ és $\vec{B} = n_s^0 \times \vec{T}$ vektorokkal dolgozunk:

$$\vec{n}_r = \vec{n}_s^0 + d_u \cdot \vec{T} + d_v \cdot \vec{B}.$$

A $\vec{T}$ *érintő vektor*, $\vec{B}$ *binormális* és az $\vec{n}_s^0$ normálvektor egy derékszögű koordinátarendszert alkot. Bucka leképzéssel készült kép látható a 7.21. ábrán.

7.17. Környezet leképzés

A textúra leképzésnek egy szellemes alkalmazása az ideális tükrök szimulációja az inkrementális képszintézis keretein belül, amelyet *környezet leképzés*nek (*environment mapping*) nevezünk [92]. Ennek az a lényege, hogy külön képszintézis lépéssel meghatározzuk, hogy mi látszik a tükörirányban, majd a képet textúraként rátapétázzuk a tükröző objektumra (7.21. ábra).

Az OpenGL-lel a környezet textúráját az ábrán látható gömb felülethez legegyszerűbben gömbi vetítéssel rendelhetjük hozzá (7.12.3. fejezet). Ekkor a környezet textúráját úgy kell elkészíteni, hogy az illeszkedjen a gömbi vetítéshez, így halszem optikát érdemes használni.

7.21. ábra. *Bucka és környezet leképzés*

7.18. Árnyékszámítás

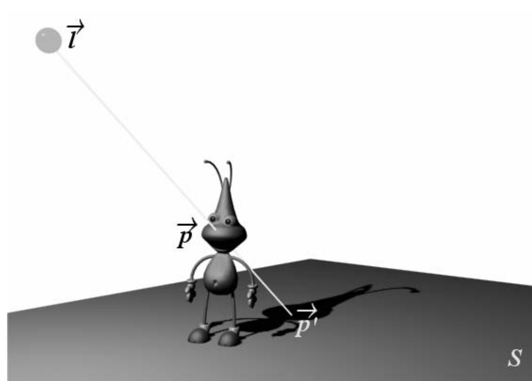
Ha a fénysugarak egy objektumon nem jutnak keresztül, akkor a fény az objektum mögötti térrészbe csak más utakon juthat el, ezért ott részben vagy teljesen sötét lesz. Az ilyen sötét térrészekben levő felületeken alakulnak ki az *árnyék*ok, amelyek fontos szerepet játszanak a virtuális világ valószerű megjelenítésében. Ha egy játékban a főhőst ábrázoló ember árnyék nélkül jelenik meg, akkor olyan hatást kelt, mintha a karakter a föld felett lebegne. Az árnyékok az animációkban talán még fontosabbak. Gondoljunk csak egy földön pattogó labdára, ahol az árnyék visszajelzést ad arról, hogy a labda éppen milyen magasan van! A következő alfejezetekben néhány jellemző árnyékszámító módszert ismertetünk, amelyeknél feltételezzük, hogy a virtuális világban csak egy fényforrás található, ugyanis több fényforrás esetén egyszerűen csak többször kell alkalmazni a bemutatott eljárásokat.

7.18.1. Síkra vetített árnyékok

A legegyszerűbb módszer *síkra vetített árnyékok*at jelenít meg [23]. Ezt az eljárást számos játék alkalmazza, amely a főszereplő földre vetülő árnyékát akarja szimulálni, vagy például az autóversenyzős játékoknál is gyakran használják az úton a „csodajárgányok" alatt megjelenő árnyékok kiszámítására.

Ismert a fényforrás $\vec{l} = [l_x, l_y, l_z]$ pozíciója és adott az S sík normálvektoros egyenlete, amelyen az árnyékot ki szeretnénk számolni (7.22. ábra):

$$S(\vec{r}) = \vec{n} \cdot (\vec{r} - \vec{r}_0) = 0, \quad \vec{n} = [A, B, C], \quad D = -\vec{n} \cdot \vec{r}_0. \tag{7.12}$$



7.22. ábra. *A síkra vetített árnyékok geometriája*

Az $\vec{l}$ fényforrást a $\vec{p} = [p_x, p_y, p_z]$ ponttal összekötő egyenes egyenlete:

$$\vec{p}' = \vec{l} + \alpha \cdot (\vec{p} - \vec{l}). \tag{7.13}$$

A 7.13. összefüggést a sík 7.12. egyenletébe helyettesítve kifejezhetjük a metszéspontot, azaz a $\vec{p}$ vetített $\vec{p}'$ képének megfelelő α értéket:

$$\alpha = \frac{\vec{n} \cdot \vec{r}_0 - \vec{n} \cdot \vec{l}}{\vec{n} \cdot (\vec{p} - \vec{l})}.$$

Az α-t a 7.13. egyenletbe visszahelyettesítve megkapjuk a $\vec{p}'$ síkra vetített pontot:

$$\vec{p}' = \vec{l} + \frac{\vec{n} \cdot (\vec{r}_0 - \vec{l})}{\vec{n} \cdot (\vec{p} - \vec{l})} \cdot (\vec{p} - \vec{l}),$$

amelyet a $\gamma = \vec{n} \cdot (\vec{l} - \vec{r}_0)$ (γ a fényforrás–sík távolság) és a $h = -\vec{n} \cdot (\vec{p} - \vec{l})$ jelölésekkel is felírhatunk:

$$\vec{p}' = \vec{l} + \frac{\gamma}{h} \cdot (\vec{p} - \vec{l}).$$

Az egyenlet mindkét oldalát h-val szorozva, majd azt átrendezve kapjuk:

$$\vec{p}\,' \cdot h = \vec{l} \cdot h + \gamma \cdot (\vec{p} - \vec{l}) = \gamma \cdot \vec{p} + \vec{l} \cdot (\vec{n} \cdot (\vec{p} - \vec{r}_0)).$$

Vegyük észre, hogy $\vec{p}\,' \cdot h$ és h a $\vec{p}$ lineáris függvénye, így a leképzés egy projektív transzformációval is leírható! Ezt a projektív transzformációt a $\mathbf{T}_{shadow}$ 4×4-es mátrixszal szorozva végezzük el:

$$[p'_x \cdot h, p'_y \cdot h, p'_z \cdot h, h] = [p_x, p_y, p_z, 1] \cdot \mathbf{T}_{shadow},$$

ahol az objektumokat az S síkra vetítő *árnyék mátrix*:

$$\mathbf{T}_{shadow} = \begin{bmatrix} \gamma - l_x \cdot A & -l_y \cdot A & -l_z \cdot A & -A \\ -l_x \cdot B & \gamma - l_y \cdot B & -l_z \cdot B & -B \\ -l_x \cdot C & -l_y \cdot C & \gamma - l_z \cdot C & -C \\ -l_x \cdot D & -l_y \cdot D & -l_z \cdot D & \gamma - D \end{bmatrix}.$$

A vetítés után kapott felületi pont Descartes-koordinátáit homogén osztással számíthatjuk ki. Ha a síkra vetített, azaz „kilapult” objektumokat sötét színnel jelenítjük meg, akkor olyan hatást érünk el, mintha azok árnyékait is kiszámoltuk volna. Ezzel a módszerrel tehát a virtuális világ árnyékokkal együttes lefényképezéséhez az összes objektumot kétszer meg kell jeleníteni (egyszer vetítés nélkül, egyszer pedig vetítve, árnyékként).

A síkra vetített kilapult objektumok és a sík „alattuk lévő” pontjai ugyanolyan mélységértékkel rendelkeznek. Ez azonban problémákat okoz a z-buffer algoritmus használatakor, hisz a z-buffer nem tudja megállapítani, hogy a kilapított objektum a sík „előtt” van. Ezért az OpenGL-lel az árnyéksokszögeket a szempozícióhoz közelebb kell hozni, így az árnyékok mindig láthatók lesznek:

```
// bekapcsoljuk a sokszögek mélység értékének eltolását
glEnable(GL_POLYGON_OFFSET_FILL);

// a sokszögek mélység értékéből 3 egységnyit levonunk
glPolygonOffset(1.0, -3.0);
```

Ha a sík sokszögeihez textúrát rendelünk, akkor az árnyékokat úgy kellene megjeleníteni, hogy a szükséges helyeken a textúra „elsötétül”. Ez megoldható az OpenGL textúra összemosó függvényeivel, ám „csak” az a probléma vele, hogy ha egy pixelre több „árnyéksokszög” is vetül, akkor azokat többször fogjuk összemosni, így az árnyékban sötét foltok jönnek létre, amelyek a valószerű kép hatását jelentősen rontják. Ráadásul ha a földet szimbolizáló sík véges kiterjedésű, előfordulhat, hogy a vetített árnyék egy része „kilóg a semmibe”. Mindezen problémák orvoslására a módszert stencil buffer használatával bővítjük ki.

Stencil buffer használata

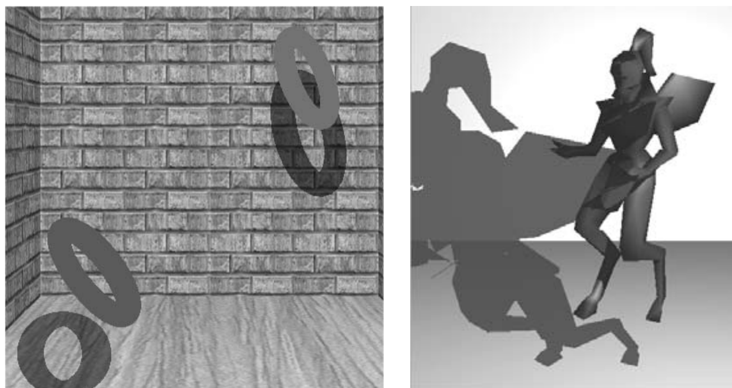
Először töröljük a stencil buffer tartalmát, majd a földet szimbolizáló sík megjelenítésekor a buffer megfelelő pixeleibe egy pozitív számot írunk. Az objektumok második felrajzolásakor csak azokat a pixeleket módosítjuk, amelyeknél a stencil bufferben nem nulla áll. Ezzel a „lelógó" árnyékok problémáját megoldottuk. Ráadásul, ha a bufferbe nullát írunk be azon pixelek esetében, amelyeknél a színt módosítjuk, akkor meg tudjuk különböztetni a már árnyékolt pixeleket, azaz a többszörös textúra összemosás problémáját is kiküszöbölhetjük. Mindezeken felül a stencil buffer használatával a sokszögek mélységértékének változtatását is elhagyhatjuk.

Ezek alapján a stencil buffert alkalmazó árnyékvető algoritmus:

```
RenderObjects();           // az objektumok első felrajzolása
glClearStencil(0);         // kitöröljük a stencil buffer tartalmát
glClear(GL_STENCIL_BUFFER_BIT);
glEnable(GL_STENCIL_TEST); // a stencil-teszt bekapcsolása


// a stencil-teszt eredményétől függetlenül minden sokszöget
// továbbengedünk, referencia értékként pedig 1-et állítunk be
glStencilFunc(GL_ALWAYS, 1, ~0); // ~0 = 0xff

// a stencil buffer tartalmát csak azokban a pixelekben változtatjuk,
// azaz 1-et írunk a 0 helyébe, ahol a sík egy darabkája megjelenik
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
RenderPlane(); // kirajzoljuk a földet szimbolizáló síkot
glDisable(GL_LIGHTING);   // kikapcsoljuk a világítást
glDisable(GL_DEPTH_TEST); // kikapcsoljuk a z-buffert

// az árnyék sötétségét az OpenGL összemosó lehetőségével érjük el
glEnable(GL_BLEND);
glBlendFunc(GL_DST_COLOR, GL_ZERO);
glColor3f(0.5, 0.5, 0.5); // összemosáskor a színt 50%-kal csökkentjük

// csak azokat a pixeleket módosítjuk, ahol a stencil bufferben 1 áll
glStencilFunc(GL_EQUAL, 1, ~0);

// a módosított pixelekhez tartozó stencil bufferbeli
// értéket kinullázzuk, így  elkerüljük a többszörös összemosást
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);

glPushMatrix();              // elmentjük a transzformációs mátrixot
glMultMatrixf(shadowMatrix); // az objektumokat a síkra vetítjük
RenderObjects();             // az objektumok másodszori rajzolása

glPopMatrix();               // alaphelyzetbe állunk vissza
glDisable(GL_STENCIL_TEST);
glDisable(GL_BLEND);
glEnable(GL_DEPTH_TEST);
```

A stencil buffert alkalmazó módszer gyors, viszont egy kép előállításához az összes objektumot kétszer kell felrajzolnia, csak sík felületeken tud árnyékot megjeleníteni, és

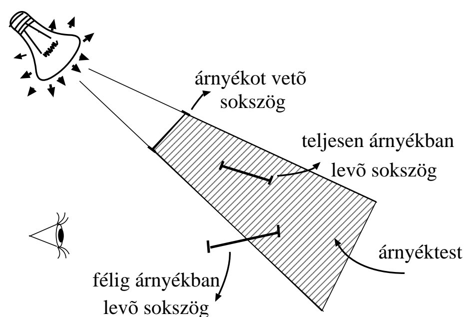
feltételezi, hogy tudjuk, melyik felületen szeretnénk az árnyékot kiszámolni.



7.23. ábra. *Síkra vetített (balra) és árnyéktesteket alkalmazó (jobbra) árnyékok*

7.18.2. Árnyéktestek

Az előző módszer feltételei a legtöbb háromdimenziós grafikai feladatnál nem teljesülnek, ezért azoknál más megoldást kell alkalmazni. A hétköznapi életben, ha egy objektum árnyékot vet, akkor az árnyék nem csupán egy sík kétdimenziós darabkája, hanem valójában egy térrész, amelyet *árnyéktest*nek hívunk. Ha egy objektum egy árnyéktestbe esik, akkor az félig vagy teljesen árnyékban van (7.24. ábra).



7.24. ábra. *Árnyéktest*

Az árnyéktest egy térrészt jelent, amelyet sokszögekből felépített határfelületével adhatunk meg, ezért a továbbiakban az árnyéktesten a határoló felületet értjük. Az

árnyéktestekkel dolgozó algoritmusok [32] az árnyékokat két menetben jelenítik meg. Az első menetben kiszámítják az árnyéktestek elhelyezkedését, míg a másodikban minden felületi pontról megállapítják, hogy árnyékban, azaz egy árnyéktesten belül van-e vagy sem. Ez utóbbi feladat hatékony megoldását stencil buffer segítségével végezhetjük el [71].

Az árnyéktestek meghatározása

Az árnyéktesteket a pontszerű vagy irány-fényforrás és az árnyékot vető objektumok definiálják. Általában az árnyékot vető objektumokat nem ismerjük előre, ezért első lépésben ezeket kell megkeresnünk. Pontosabban elegendő az árnyékot vető objektumok körvonalát alkotó sokszögeket megkeresni, hisz azok az árnyéktesteket egyértelműen meghatározzák.

Ha a fényforrásból tekintünk a virtuális világra, akkor pontosan azokat az objektumokat látjuk, amelyek árnyékokat vetnek. Ezért a fényforrásból készítünk egy konstans árnyalással készült képet, amelyen az objektumok lapjainak színeként az indexüket adjuk meg, így az elkészített képből meg tudjuk határozni, hogy a fényforrásból mely felületek láthatók.

Az árnyékot vető objektumok körvonalai a meghatározott felületek élei közül kerülnek ki. Azokra az élekre nincs szükségünk, amelyek a látható felületek közül kettőhöz is tartoznak, hisz ezek a körvonalak szempontjából „belső” élek, csak azokra, amelyek pontosan egyszer tűnnek fel. Tehát a fölösleges belső élek elhagyásával a körvonalakat a következő egyszerű algoritmussal határozhatjuk meg:

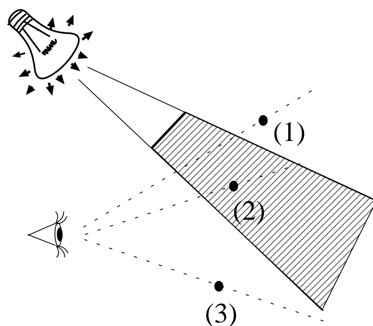
```
l éllista kiürítése;
for (minden o látható sokszögre) {
    for (o sokszög minden e élére) {
        if (e él nem létezik az l éllistában)
            Beszúrjuk az e élt az l éllistába;
        else
            Kitöröljük a már létező élet az l éllistából;
    }
}
```

Ez az algoritmus egy élt beszúr, ha nincs a listában, és eltávolít, ha van, így végül csak azok az élek maradnak benne, amelyeket páratlan sokszor, azaz nem kétszer próbáltunk feldolgozni.

Az árnyékok meghatározása

Miután a virtuális világot árnyékok nélkül megjelenítettük és kiszámoltuk az árnyéktestet határoló négyszögeket, meg kell határozni, hogy a látható felületi pontok közül

melyik van árnyékban és melyik nincs. Tegyük fel, hogy a szempozíció az árnyéktesten kívül helyezkedik el! Egy felületi pont akkor és csak akkor van árnyékban, ha a pontot a szempozícióval összekötő szakasz az árnyéktestet határoló sokszögeket páratlan sokszor metszi (7.25. ábra). Ha viszont a szempozíció épp egy árnyéktest belsejében van, akkor a felületi pont pontosan akkor van árnyékban, ha a szakasz a sokszögeket páros sokszor metszi.



7.25. ábra. *Árnyékok meghatározása metszéspontok száma alapján*

Ezt a megfigyelést felhasználva az árnyéktestet határoló sokszögeket is „küldjük végig" a nézeti csővezetéken, úgy, hogy a z-buffer és a képernyő tartalmát ne módosítsuk, de a z-ellenőrzést azért használjuk. Ezzel állapítjuk meg, hogy melyik pixeleket kellene frissíteni, majd a stencil bufferben ezekhez tartozó értékeket invertáljuk. Ha a műveletet megelőzően a stencil bufferben mindenhol 0 állt, és a szempozíció az árnyéktesten kívül van, akkor az árnyéktest sokszögeinek megjelenítése után a bufferben pontosan azokon a helyeken lesz 0, ahol az invertálást páros sokszor (akár 0 alkalommal) alkalmaztuk, 1 pedig azokban a pixelekben, ahol páratlan sokszor hajtottuk végre azt. Ez azt jelenti, hogy a 0-tól különböző stencil bufferbeli értékek azokat a pixeleket jelzik, amelyekben árnyékban levő felületi pontok láthatóak. Tehát, ha szigorúan csak ezekben a pixelekben a virtuális világot kikapcsolt fényforrással is lefényképezzük, akkor a készített képen az árnyékok is rajta lesznek.

Ezek alapján az árnyékban levő pixelek meghatározása:

```
glEnable(GL_DEPTH_TEST); // bekapcsoljuk a z-buffer algoritmust
glDepthFunc(GL_LESS);    // a szokásos z-buffer algoritmus működik,
glDepthMask(0);          // de a buffer tartalmát nem írja felül
glColorMask(0, 0, 0, 0); // letiltjuk a színbuffer módosítását
glClearStencil(0);       // kitöröljük a stencil buffer tartalmát
glClear(GL_STENCIL_BUFFER_BIT);
glEnable(GL_STENCIL_TEST); // bekapcsoljuk a stencil buffer használatát

// a stencil-teszten minden sokszöget átengedünk,
```

```
// referencia értékként pedig 0-t használunk
glStencilFunc(GL_ALWAYS, 0, 0);
glStencilMask(0x1);

// az árnyéktest minden sokszögét ki kell rajzolni, ezért a megjelenítésük
// idejére a 'triviális hátsólap eldobást' kikapcsoljuk
glDisable(GL_CULL_FACE);
RenderShadowVolumePolygons();  // az árnyéktestek kirajzolása
glEnable(GL_CULL_FACE);        // kérjük a hátsó lapok eldobását

// kikapcsoljuk a fényforrást, így alakítjuk ki az árnyékok színét
glDisable(GL_LIGHT0);

// egy adott pixelben csak azt a sokszöget használjuk, amelyet a virtuális
// világ első kirajzolásakor a pixelben legközelebb levőként jelöltünk meg
glDepthFunc(GL_EQUAL);
glDepthMask(0);

// csak azokat a pixeleket engedjük frissíteni, amelyek stencil bufferbeli
// értéke 1, azaz azt jelzik, hogy a látható felületi pont árnyékban van
glStencilFunc(GL_EQUAL, 0x1, 0x1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

glColorMask(1, 1, 1, 1); // engedélyezzük a kép bufferének átírását
RenderObjects();         // a virtuális világ második megjelenítése
```

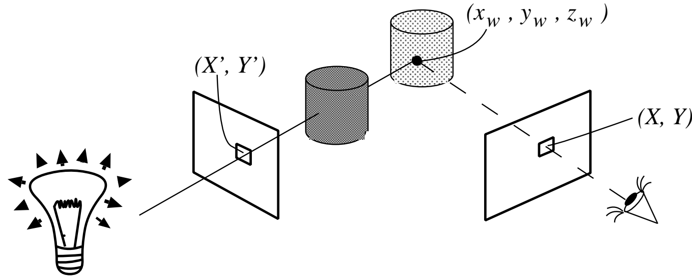
Ha az árnyéktestet határoló sokszögek helyét pontosan határozzuk meg, akkor a módszer az árnyékokat a megfelelő helyen jeleníti meg, ráadásul a stencil buffert tartalmazó videokártyák esetén a megjelenítés gyors lesz. Azonban egy kép előállításához az összes objektumot kétszer kell felrajzolni, ami hardveresen gyorsított stencil buffer nélkül nagyon lassú. Ráadásul ha az első vágósík egy árnyéktestet határoló sokszögbe belevág, akkor a stencil buffer használatakor problémák merülhetnek fel.

Az árnyékvető módszerek stencil bufferrel való implementálásával az nVidia honlapjáról letölthető [71] cikk igen részletesen foglalkozik. Az árnyéktestek implementációs nehézségeivel számos kutató és programozó, köztük John Carmack (a Doom és a Quake fő fejlesztője) is foglalkozott [27]. Ezen felül ajánljuk Hun Yen Kwoon gyűjteményes munkáját, amely rengeteg problémás esetet felvillant, és megoldást is mutat rájuk, illetve igen jó irodalomjegyzékkel rendelkezik [142].

7.18.3. Árnyékszámítás z-buffer segítségével

Williams 1978-ban egy olyan árnyékvető módszert javasolt, amely a z-buffer algoritmust használja [139]. A módszer egy előfeldolgozási lépésben a fényforrásból nézve meghatározza a z-buffer tartalmát. Ezzel megkapjuk a fényforrásból nézve legközelebbi sokszögek távolságértékeit. Ezután eldöntjük, hogy a képernyő (X,Y) pixelében látható $(x_w,\ y_w,\ z_w)$ felületi pont a fényforrásból készített kép melyik $(X',\ Y')$ pixelében milyen Z' távolságra lenne látható. Ha a Z' nagyobb, mint a meghatározott $(X',\ Y')$ koordinátán levő z-bufferbeli mélységérték, akkor ez azt jelenti, hogy a fényforrásból

nézve az (x_w, y_w, z_w) felületi pontnál van közelebbi is, amely a pontot takarja. Ebből pedig az következik, hogy a felületi pont árnyékban van (7.26. ábra). Ha a két z érték egy igen kicsi ε sugarú környezetben van, akkor a felületi pont mind a kamerából, mind az éppen vizsgált fényforrásból látható. Ez azt jelenti, hogy a pont nincs árnyékban.



7.26. ábra. *Árnyékszámítás z-buffer segítségével*

Legyen $\mathbf{T}_l$ a virtuális világ objektumait a fényforrások „képernyőjére” vetítő transzformáció, a kamera esetében pedig $\mathbf{T}_c$! Ekkor egy (X, Y) pixelben látható, képernyő-koordinátarendszerben Z mélységértékű felületi pont világ-koordinátarendszerben adott koordinátáit úgy határozzuk meg, hogy a felületi pontra végrehajtjuk a $\mathbf{T}_c^{-1}$ inverz transzformációt. A kiszámított (x_w, y_w, z_w) koordinátákból a fényforrás képernyő-koordinátarendszerbeli értékét pedig úgy kaphatjuk meg, hogy a $\mathbf{T}_l$ transzformációt hajtjuk végre. A következő oldalakon bemutatjuk az algoritmus egy lehetséges implementációját, amelyben csak egy fényforrással dolgozunk, ráadásul az is csak egy irányba világít. A megvalósításhoz felhasználjuk a 2. fejezetben bemutatott keretrendszert, ezért első lépésként az `Application` osztályból származtatunk egy újat, amely az alkalmazást vezérli.

```
//============================================
class ShadowZBuffer : public Application {
//============================================
   float Tc_inv[4][4];  // eszköz->világ-koordinátarendszer transzformáció
   float Tl[4][4];      // világ->fényforrás-koordinátarendszer transzformáció
   float Tcl[4][4];     // a két transzformáció szorzata
   float lightDepth[IMAGE_SIZE*IMAGE_SIZE];  // fényforráshoz tartozó z-buffer
public:
   ShadowZBuffer();
   void CalcCameraTransf(Vector& eye, Vector& lookAt, Vector& up);
   void CalcLightTransf(Vector& light, Vector& lookAt, Vector& up);
   bool ShadowZCheck(float X, float Y, float Z);
};
```

A `CalcCameraTransf()` metódusban kiszámítjuk a kamera eszköz-koordinátarendszeréből a világ-koordinátarendszerbe átvivő `Tc_inv` transzformációt:

```
//-----------------------------------------------------------------
void ShadowZBuffer::CalcCameraTransf(Vector& eye,Vector& lookAt,Vector& up) {
//-----------------------------------------------------------------
    glMatrixMode(GL_PROJECTION); // perspektív transzformáció
    glLoadIdentity();
    gluPerspective(45, 1, 1, 100);
    glMatrixMode(GL_MODELVIEW);  // nézeti transzformáció
    glLoadIdentity();
    gluLookAt(eye.X(), eye.Y(), eye.Z(),
              lookAt.X(), lookAt.Y(), lookAt.Z(), up.X(), up.Y(), up.Z());
    float Tproj[4][4], Tmodview[4][4];
    glGetFloatv(GL_PROJECTION_MATRIX, &Tproj[0][0]);   // projekciós mátrix
    glGetFloatv(GL_MODELVIEW_MATRIX, &Tmodview[0][0]); // modell-nézeti mátrix
    MatrixConcat(Tmodview, Tproj, Tc_inv);
    MatrixInvert(Tc_inv, Tc_inv); // kamera eszköz -> világ transzformáció
}
```

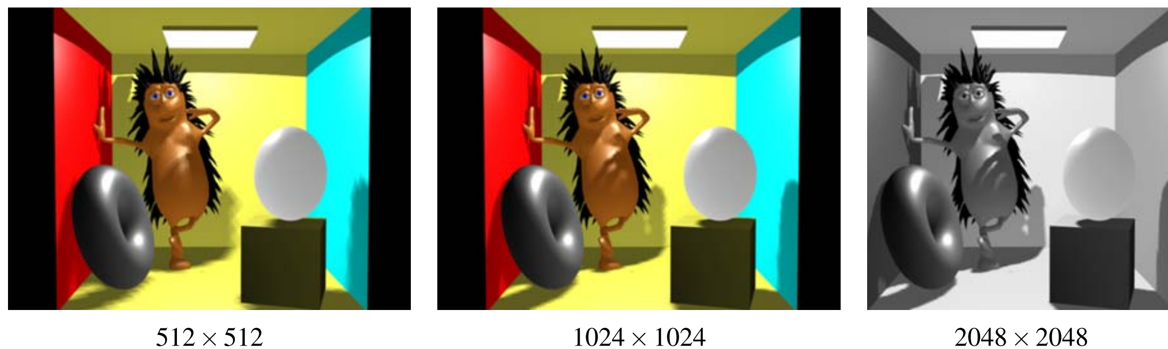
A `CalcLightTransf()` metódusban kiszámítjuk a világ-koordinátarendszerből a fényforrás eszköz-koordinátarendszerébe átvívő `Tl` transzformációt, majd a fényforrás szemszögéből fényképezzük le a virtuális világot, azért, hogy az OpenGL-től lekérhessük a z-buffer tartalmát. Pontszerű fényforrásból hat képet kell készíteni: egyet felfelé, egyet lefelé, egyet balra, egyet jobbra, egyet előre és egyet hátra. Ez azt jelenti, hogy a metódust hatszor kell hívni.

```
//-----------------------------------------------------------------
void ShadowZBuffer::CalcLightTransf(Vector& light,Vector& lookAt,Vector& up) {
//-----------------------------------------------------------------
    glMatrixMode(GL_PROJECTION); // perspektív transzformáció
    glLoadIdentity();
    gluPerspective(90, 1, 1, 100);
    glMatrixMode(GL_MODELVIEW);  // nézeti transzformáció
    glLoadIdentity();
    gluLookAt(light.X(), light.Y(), light.Z(),
              lookAt.X(), lookAt.Y(), lookAt.Z(), up.X(), up.Y(), up.Z() );
    float Tproj[4][4], Tmodview[4][4];
    glGetFloatv(GL_PROJECTION_MATRIX, &Tproj[0][0]);
    glGetFloatv(GL_MODELVIEW_MATRIX, &Tmodview[0][0]);

    // transzformáció: világból a fényforrás eszköz koordinátarendszerébe
    MatrixConcat(Tmodview, Tproj, Tl);
    // transzformáció: kamerából a fényforrás eszköz-koordinátarendszerébe
    MatrixConcat(Tc_inv, Tl, Tcl);
    RenderObjects(); // az objektumok felrajzolása
    glReadPixels (0, 0, IMAGE_SIZE, IMAGE_SIZE,  // z-buffer elmentése
                  GL_DEPTH_COMPONENT, GL_FLOAT, &lightDepth[0]);
}
```

A `ShadowZCheck()` metódusban egy (X,Y,Z) pontról eldöntjük, hogy árnyékban van-e vagy sem. Ennek során a pontot először a kamera eszköz-koordinátarendszerében határozzuk meg, majd végrehajtjuk a `CalcLightTransf` metódusban kiszámolt, a fényforrás eszköz-koordinátarendszerébe átvivő transzformációt. Ebből pedig az (X',Y',Z') pont már könnyen előállítható.

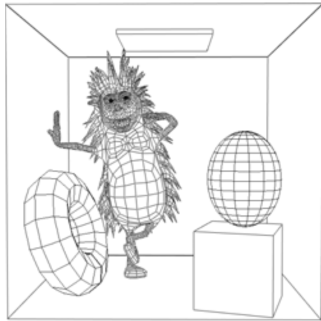
```
//-----------------------------------------------------------------
bool ShadowZBuffer::ShadowZCheck(float X, float Y, float Z) {
//-----------------------------------------------------------------
    float x = X * 2.0 / IMAGE_SIZE - 1; // X eszköz koordinátában adva
    float y = Y * 2.0 / IMAGE_SIZE - 1; // Y eszköz koordinátában adva
    float z = Z * 2.0 - 1.0;            // Z eszköz koordinátában adva

    // az (x,y,z)-t a fényforrás eszköz-koordinátarendszerébe transzformáljuk
    float xl = x * Tcl[0][0] + y * Tcl[1][0] + z * Tcl[2][0] + Tcl[3][0];
    float yl = x * Tcl[0][1] + y * Tcl[1][1] + z * Tcl[2][1] + Tcl[3][1];
    float zl = x * Tcl[0][2] + y * Tcl[1][2] + z * Tcl[2][2] + Tcl[3][2];
    float wl = x * Tcl[0][3] + y * Tcl[1][3] + z * Tcl[2][3] + Tcl[3][3];
    xl /= wl; yl /= wl; zl /= wl;

    int Xl = (xl + 1) * IMAGE_SIZE/2 + 0.5; // X': xl képernyő koordinátában
    int Yl = (yl + 1) * IMAGE_SIZE/2 + 0.5; // Y': yl képernyő koordinátában
    if (Xl<0 || Xl>IMAGE_SIZE-1 || Yl<0 || Yl>IMAGE_SIZE-1) return false;

    // a fényforrás z-bufferének (X',Y') pixeléhez tartozó mélységérték
    float z = lightDepth[(int)(Yl * IMAGE_SIZE + Xl)] * 2 - 1;
    if (z + EPSILON >= zl) return false; // a felületi pont nincs árnyékban
    return true; // a felületi pont árnyékban van
}
```

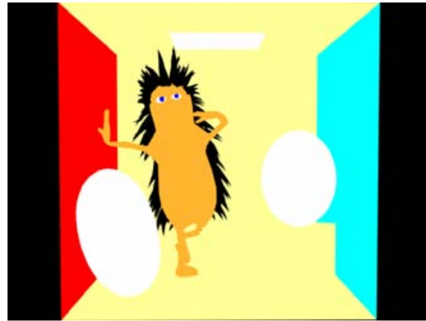
A módszer előnye, hogy a grafikus kártyák z-bufferével hardveresen gyorsítható. Hátránya viszont, hogy az árnyékok széle kisfelbontású árnyéktérképek esetén csipkézett, másrészt minden esetben megfelelő ε választása szinte lehetetlen, így az árnyékokban lyukak jelenhetnek meg, amelyet *árnyék kiütés*nek (*shadow acne*) nevezünk.



512 × 512 1024 × 1024 2048 × 2048

7.27. ábra. *Különböző felbontású árnyéktérképekkel számolt árnyékok*


1. huzalváz


2. saját színnel árnyalás

3. konstans árnyalás

4. Gouraud-árnyalás

5. Gouraud-árnyalás
finom tesszellációval

6. Phong-árnyalás

7. textúra leképzés

8. z-buffer árnyékok

9. egyszerű sugárkövetés

7.28. ábra. *Megjelenítés árnyalás nélkül és lokális illuminációs modellel*

7.19. A 3D grafikus hardver

A 7.29. ábra egy tipikus *grafikus processzor* (*GPU*) felépítését mutatja be. A grafikus API (például OpenGL) hívások a kártya parancsfeldolgozójához kerülnek, amely a *csúcspont-árnyaló* (*vertex shader*) modullal áll kapcsolatban. A csúcspont-árnyaló a háromszögek csúcspontjait homogén koordinátás, normalizált képernyő-koordinátarendszerbe (7.5. ábra jobb oldala) transzformálja, és megváltoztathatja a csúcsokhoz kapcsolódó tulajdonságokat (textúra koordináta, szín, normálvektor stb.), például a színt kicserélheti az illuminációs számítás eredményével. A csúcspont-árnyalást követően a vágás azon háromszög részeket tartja meg, ahol a homogén $[x,y,z,w]$ koordináták teljesítik a $-w \leq x \leq w$, $-w \leq y \leq w$, $-w \leq z \leq w$ egyenlőtlenségeket. A vágás után a kártya elvégzi a homogén osztást, majd a Descartes-koordinátákra alkalmazza a képernyő transzformációt (7.5. fejezet), amely után egy pont x,y koordinátái éppen azt a pixelt jelölik ki, amelyre az vetül. A *raszterizáló egység* három egymás utáni csúcspontot bevárva, a csúcsokra háromszöget illeszt, és annak x,y vetületét kitölti, azaz egyenként meglátogatja azokat a pixeleket, amelyek a vetület belsejébe esnek. A kitöltés során a hardver a csúcspont-árnyaló kimeneteként előállított tulajdonságokból (szín, textúra koordináta, mélységérték stb.) lineárisan interpolációval pixel tulajdonságokat számít ki. A *pixel-árnyaló* (*pixel shader*) a pixel tulajdonságokból előállítja a pixel színét, amelyhez általában a textúratárból színinformációt olvas ki. A GPU takarási feladatot általában ezt követően, jellemzően z-buffer alkalmazásával oldja meg. Végül a hardver a z-buffer által átengedett pixeleket színét a rasztertárba írja, vagy ha engedélyeztük az átlátszóság számítását, akkor a rasztertárban az ebben a pixelben található színnel összemossa.

7.29. ábra. *Egy tipikus grafikus processzor felépítése*

A modern grafikus processzorok lehetőséget adnak arra, hogy ezen működési modellbe két ponton is „belenyúljunk”. A csúcspont-árnyaló működését, azaz a csúcspontok koordinátáinak és tulajdonságainak az átalakítását, valamint a *pixel-árnyaló* műveleteit, azaz a pixelszín textúra leképzéssel történő számítását, saját programmal cserélhetjük le. A csúcspont és pixel-árnyalók programozására asssembly jellegű nyelvet, vagy magasszintű árnyaló nyelvet használtunk, mint például a DirectX grafikus alrendszer (11.

fejezet) *HLSL-nyelv*ét (*High Level Shader Language*), és az OpenGL-ből és DirectX-ből egyaránt elérhető *Cg-nyelv*et. A továbbiakban a Cg-nyelvből szeretnénk egy kis ízelítőt adni[5]. A Cg-nyelv alapja a C programozási nyelv, amelyben nyelvi szintre emelték a vektorok és mátrixok kezelését. Például a `float4` egy négyelemű `float` vektort, a `float4x4` pedig egy 4x4-es mátrixot jelent. A 4×32 bites `float4` változók egyes elemeit a változó neve után tett `.x`, `.y`, `.z`, `.w`, vagy `.r`, `.g`, `.b`, `.a` utótagokkal kaphatjuk meg.

7.19.1. Csúcspont-árnyalók

Egy csúcspont-árnyaló az aktuális csúcspont bemeneti és kimeneti tulajdonságait definiáló, `float4` típusú regiszterekkel rendelkezik, mint a pozíció (`POSITION`), a színek (`COLOR0`, `COLOR1`), a normálvektor (`NORMAL`), a textúra koordináták (`TEXCOORD0`,..., `TEXCOORD8`) stb. A bemeneti regiszterekbe az OpenGL felületén átadott csúcspont tulajdonságok kerülnek. A `glVertex` a paramétereit a lebegőpontos konverzió után a `POSITION` regiszterbe, a `glColor` a `COLOR0` regiszterbe, a `glNormal` a `NORMAL` regiszterbe, a `glTexCoord` a `TEXCOORD0` regiszterbe teszi. A csúcspont-árnyaló program a bemeneti regiszterekből kiszámítja a kimeneti regiszterek értékét. Az árnyaló a számításokhoz a bemeneti regisztereken kívül még egységes (*uniform*) bemeneti változókat is felhasználhat, amelyek nem változhatnak csúcspontonként, hanem egy `glBegin` és `glEnd` pár között állandók. Tipikus egységes paraméterek a transzformációs mátrixok, anyagtulajdonságok és fényforrás adatok, de a programozó akár saját maga is definiálhat ilyeneket.

A következő csúcspont-árnyaló a szokásos modell-nézeti és perspektív transzformációt végzi el, azaz a pontot homogén koordinátás alakban a normalizált képernyőkoordinátarendszerben fejezi ki, a kapott színt és textúra koordinátákat pedig változtatás nélkül továbbadja:[6]

```
struct outputs { // kimeneti regiszterek elnevezése
   float4 hposition : POSITION;     // transzformált pont homogén koordinátákban
   float3 color     : COLOR0;       // a csúcspont színe
   float2 texcoord  : TEXCOORD0;    // a csúcspont textúra koorinátája
};

outputs main(                       // ki: outputs-ban felvett regiszterek
   in float4 position  : POSITION,  // be: pozíció a glVertex-ből a POSITION-ban
   in float3 color     : COLOR0,    // be: szín a glColor-ból a COLOR0-ban
   in float2 texcoord  : TEXCOORD0, // be: textúra koordináta a glTexCoord-ból
   uniform float4x4 modelviewproj ) // be: modell-nézeti * perspektív transzf.
{
   outputs OUT;
   OUT.hposition = mul(modelviewproj, position); // képernyő koordinátákba
```

[5]A HLSL kísértetiesen hasonlít a Cg-re.

[6]Az illuminációval most nem foglalkoznunk, azaz `glDisable(GL_LIGHTING)` beállítást feltételezünk.

```
   OUT.texcoord = texcoord;
   OUT.color = color;
   return OUT;
}
```

Figyeljük meg, hogy a bemeneti és kimeneti regisztereknek tetszés szerinti változóneveket adhattunk! A bemeneti regiszterek és egységes paraméterként a modellezési, nézeti és perspektív transzformációk szorzatát jelentő `modelviewproj` 4×4-es mátrix a csúcspont-árnyaló `main` függvényének bemeneti paraméterei, az eredmény regisztereket összefogó struktúra pedig a visszatérési értéke.

7.19.2. Pixel-árnyalók

A pixel-árnyalók a vágott, vetített háromszögeket kitöltő pixelekre futnak le, és a pixel a saját tulajdonságai, valamint egységes (uniform) paraméterek alapján a pixel színét (és esetleg mélység értékét) számíthatják ki. A saját pixel tulajdonságokat a raszterizáló egység a csúcspont tulajdonságokból a háromszög belsejében lineárisan interpolációval állítja elő. Ugyanaz a program fut le az összes pixelre, de minden pixel a bemeneti regisztereiben csak a saját tulajdonságait kapja meg. A pixel árnyaló szokásos egységes paraméterei a felhasználandó textúrák azonosítói. A pixel árnyaló a szín számítása során a textúratárból adatokat olvashat ki.

A következő pixel-árnyaló programban, a csúcspont-árnyaló által előállított, majd a raszterizáló egység által interpolált színt (`COLOR0` regiszter) és textúra koordinátákat (`TEXCOORD0` regiszter) kapjuk meg, valamint egységes paraméterként a textúra azonosítót (`texture`). A pixel-árnyaló kiolvassa a textúrából a textúra koordinátákkal címzett texelt, és azt a kapott színnel szorozva (modulálva) állítja elő a pixel végleges színét:

```
float4 main( in float2 texcoord  : TEXCOORD0,  // be: textúra koordináta
             in float3 color     : COLOR0,     // be: szín
             uniform sampler2D texture )       // be: textúra azonosító
                : COLOR  // ki: az float4 eredmény a COLOR regiszterbe
{
   return tex2D(texture, texcoord) * color;        // moduláció
}
```

7.19.3. Magasszintű árnyaló nyelvek

Végül nézzük meg, hogy az OpenGL programunkból hogyan bírhatjuk rá a grafikus kártyát, hogy az általunk megírt csúcspont és pixel-árnyaló programot hajtsa végre, és hogyan állíthatjuk be az árnyaló programok paramétereit! Mindenekelőtt szükségünk van a Cg könyvtárra[7], amelynek deklarációit a `cgGL.h` fejléc fájlban találhatjuk.

[7] A Cg könyvtár, a nyelv leírása és fordítója a http://developer.nvidia.com/object/cg_toolkit.html címről ingyenesen letölthető

Tekintsük először az inicializáló részt, amely betölti a Cg forrásnyelvű programokat, lefordítja azokat, majd átadja a grafikus kártyának, végül pedig meghatározza, hogy az egységes paraméterekre milyen névvel hivatkozzunk a CPU-n futó, valamint a csúcspont és pixel-árnyaló programjainkban:

```
#include <Cg/cgGL.h>                // cg függvények deklarációi
CGparameter MVPT, textureMap;       // egységes (uniform) paraméterek

void InitCg( ) {
   CGprofile VP = CG_PROFILE_ARBVP1, PP = CG_PROFILE_ARBFP1;  // 1.0 utasítások
   cgGLEnableProfile(VP); cgGLEnableProfile(PP);

   CGcontext shaderContext = cgCreateContext();      // árnyaló környezet
        // a forrásnyelvű csúcspont-árnyalót az árnyaló környezetbe töltjük
   CGprogram vertexProg = cgCreateProgramFromFile(shaderContext,
                              CG_SOURCE, "myvertex.cg", VP, NULL, NULL);
   cgGLLoadProgram(vertexProg);      // áttöltjük a GPU-ra
   cgGLBindProgram(vertexProg);      // ez fusson
        // a forrásnyelvű pixel-árnyalót az árnyaló környezetbe töltjük
   CGprogram pixelProg = cgCreateProgramFromFile(shaderContext,
                              CG_SOURCE, "mypixel.cg", PP, NULL, NULL);
   cgGLLoadProgram(pixelProg);       // áttöltjük a GPU-ra
   cgGLBindProgram(pixelProg);       // ez fusson
                // egységes (uniform) paraméterek név-összerendelése
   MVPT = cgGetNamedParameter(vertexProg, "modelviewproj");
   textureMap = cgGetNamedParameter(pixelProg, "texturemap");
}
```

Az inicializálás a csúcspont és pixel árnyaló-utasításkészletének kijelölésével kezdődik, ahol az 1.0-ás szabványú utasítások elfogadását kértük. Az árnyaló környezet (`shaderContext`) felépítése egy táblázatot hoz létre a Cg könyvtárban, amely az árnyaló programjainkat és tulajdonságaikat tartalmazza. Ebbe a táblázatba programokat tölthetünk be a `cgCreateProgramFromFile` függvény segítségével, amelynek megadjuk a program forrását tartalmazó fájl nevét (`myvertex.cg`), közöljük azt tényt, hogy ez forrásnyelvű, tehát a fordításáról a betöltéssel párhuzamosan gondoskodni kell, valamint kijelöljük a fordításnál megengedhető utasítások körét (1.0-ra állított `VP`). A `cgGLLoad-Program` az árnyaló környezetből a grafikus kártyára másolja a lefordított programot, a `cgGLBindProgram` pedig ezt jelöli ki futásra az áttöltött programok közül. A pixel-árnyaló kártyára töltése ugyanilyen lépéseket igényel. Végül a csúcspont és pixel-árnyaló programokhoz egységes paramétereket definiálunk. Például a csúcspont-árnyaló egységes paraméterére a CPU-n `MVPT` névvel, a GPU-n pedig `modelviewproj` névvel hivatkozunk. Az ismertetett inicializálási lépések után a fájlokban leírt csúcspont és pixel-árnyaló programok váltják fel a megszokott OpenGL megjelenítési csővezeték két programozható fázisát. A megjelenítési csővezetéket a megszokott módon, a `glBegin` és `glEnd` hívások közé elhelyezett csúcspont adatokkal táplálhatjuk. Újdonságot csak az egységes paraméterek beállítása jelent.

```
void Render(void) {
```

```
   cgGLSetStateMatrixParameter(MVPT,     // modell-nézeti-perspektív transzf.
                            CG_GL_MODELVIEW_PROJECTION_MATRIX,
                            CG_GL_MATRIX_IDENTITY);
   glBindTexture(GL_TEXTURE_2D, texture_id);
   cgGLSetTextureParameter(textureMap, texture_id); // textureMap beállítása
   cgGLEnableTextureParameter(textureMap);          // textureMap engedélyezése
   ...
   glBegin( GL_TRIANGLES ); // nem uniform paramétereknek értékadás
   for( ... ) {
      ...                   // csúcspont tulajdonságok számítása
      glColor3f(r, g, b);   // (r,g,b) a COLOR0 regiszterbe
      glTexCoord2f(u, v);   // (u,v)   a TEXCOORD0 regiszterbe
      glVertex3f(x, y, z);  // (x,y,z,1) a POSITION regiszterbe
   }
   glEnd();
   cgGLDisableTextureParameter(textureMap);          // textureMap tiltása
}
```

Először a `MVPT` egységes paramétert, a modell-nézeti-perspektív transzformációs mátrixot állítjuk be úgy, hogy az OpenGL-t megkérjük, hogy az általa kiszámított mátrixot (`CG_GL_MODELVIEW_PROJECTION_MATRIX`) még egy egységmátrixszal való szorzás után (`CG_GL_MATRIX_IDENTITY`), azaz változtatás nélkül, adja át a grafikus kártyának. Beállíthatunk még inverz számítást, vagy akár mátrix transzponálást is. A pixelárnyalóban felhasznált `textureMap` egységes paramétert a textúra azonosító értékére állítjuk, amit a textúra létrehozásánál a `glGenTextures` függvénnyel kaptunk. A textúra paramétereket ezenkívül még engedélyezni is kell.

8. fejezet

Globális illumináció

Idáig a fény–anyag kölcsönhatás leegyszerűsített modelljét használtuk. Nem vettük figyelembe azt, hogy a fény többszörös visszaverődés után is a szemünkbe juthat, a fényforrásokat pontszerű, szpot, irány és ambiens kategóriákba[1] osztottuk, amelyek közül a valóságban egyik sem létezik. Feltételeztük, hogy a fény csak a vörös, a zöld és a kék szín hullámhosszain terjed, holott a valóságban a fényforrások a teljes látható hullámhossztartományban kibocsátanak energiát, amelynek bármely komponensét érzékelhetjük is. Ezekre az egyszerűsítésekre elsősorban azért volt szükségünk, hogy a képeket a valós idejű megjelenítés sebességével ki tudjuk számolni. Az elhanyagolásokért azonban nagy árat kell fizetnünk. Mivel a számítások során használt összefüggések nem felelnek meg a természet törvényeinek, a keletkezett képek sem fognak a mindennapjaink során megszokott látványhoz hasonlítani. A nyilvánvaló csalás és az ebből adódó pontatlanság ellenére képeink még nagyon szépek lehetnek, de egyes mérnöki alkalmazásokban (például világítástervezésben) teljesen hasznavehetetlenek. Ebben a fejezetben olyan algoritmusokat mutatunk be, amelyek nem élnek durva egyszerűsítésekkel, és így a képet a valóságnak megfelelően számítják ki.

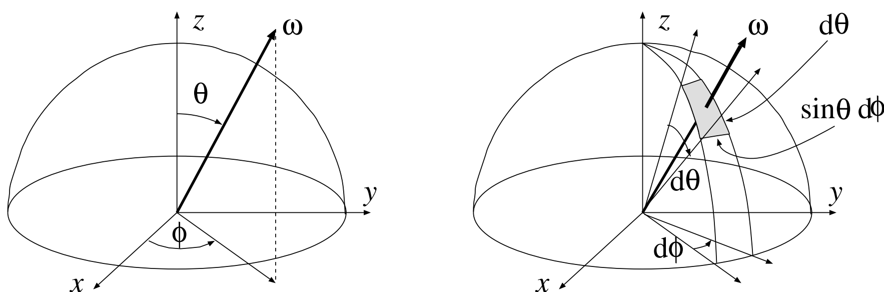
A bevezetőben beszéltünk a fény kettős természetéről, amely szerint a fényt egyrészt elektromágneses hullámnak, másrészt pedig fotonok gyűjteményének tekinthetjük. Mindkét értelmezésben közös, hogy a fény a látható hullámhossztartomány frekvenciáin energiát szállít a fényforrásoktól visszaverődéseken és töréseken keresztül az emberi szemig. A színérzetet a szembe érkező spektrum határozza meg.

Mivel a fényerősség pontról pontra és irányról irányra változhat, még mielőtt belevágnánk a fényerősség mértékeinek tárgyalásába, szenteljünk egy kis időt a pontok és irányok halmazainak!

[1] absztrakt fényforrások

8.1. Pont és irányhalmazok

Miként már a modellezésről szóló fejezetben megállapítottuk, a pontokat egy alkalmas koordinátarendszer, például Descartes-koordinátarendszer segítségével számokkal adhatjuk meg. Egy felületelem pontok halmaza. A halmaz nagyságát (mértékét) a felület ΔA területével jellemezhetjük. Ha azt a határesetet vizsgáljuk, amikor a terület végtelenül kicsi lesz, és egyetlen pontra zsugorodik, akkor a differenciális felületelemet dA-val jelöljük. Ha külön hangsúlyozni akarjuk, hogy a felületelem az $\vec{x}$ vagy az $\vec{y}$ pontokra zsugorodik, akkor a dx, illetve a dy jelölést alkalmazzuk.



8.1. ábra. *Az irányok (bal) és a differenciális térszög (jobb)*

A térbeli irányok bevezetése előtt érdemes felidézni, hogy a síkban az irányokat szögekkel jellemezhetjük. Egy síkszög az egységkör egy ívével adható meg, értéke pedig ezen ív hossza. A szög mértékegysége a *radián*, vagy annak $180/\pi$-szerese, a *fok*. A szögtartomány azon irányokat foglalja magába, amelyek a szög csúcsából az ív valamely pontjába mutatnak.

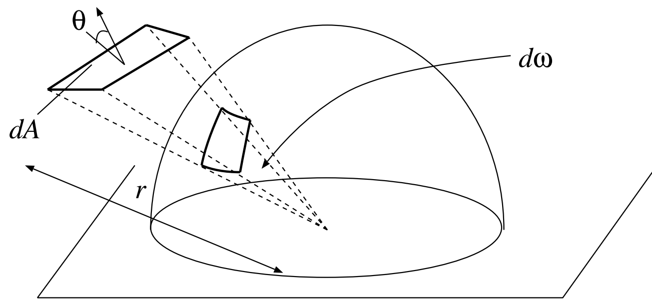
Az egységkör és a síkbeli szög fogalmának általánosításával juthatunk el az illuminációs gömb és a térszög fogalmához. A térbeli irányokat a 2D egységkör mintájára az úgynevezett *illuminációs gömb* segítségével definiálhatjuk egyértelműen. Az illuminációs gömböt Ω-val jelöljük. Ha a felület nem átlátszó, akkor csak a felület fölötti félgömbből érkezhet fény, ezért ekkor *illuminációs félgömb*ről beszélünk és az Ω_H jelölést alkalmazzuk.

Egy irány lényegében az origó középpontú egységsugarú gömb egyetlen pontja. Az irány tehát ugyancsak egy egységvektor, amit $\vec{\omega}$-val jelölünk. Az irányokat kényelmesebb Descartes-koordinátarendszer helyett gömbi koordinátákban megadni, hiszen ekkor az egységnyi hosszra vonatkozó megkötés nem igényel további számításokat (8.1. ábra). A *gömbi-koordinátarendszer*ben egy irányt két szög ír le, a θ az irány és a z-tengely közötti szöget, a ϕ pedig az irány x,y síkra vett vetülete és az x-tengely közötti szöget

jelenti (3.1.3. fejezet). A felületekhez hasonlóan, az irányhalmazok nagyságát a gömbi területek méretével adhatjuk meg. A térszög mértékegysége a *szteradián* $[sr]$. A gömbi terület méretét *térszög*nek (*solid angle*) nevezzük. A véges térszöget $\Delta\omega$-val, a kicsiny (differenciális) térszögeket $d\omega$-val jelöljük. Egy térszög azon irányokat tartalmazza, amelyek a gömb középpontjából a felületrész valamely pontjába mutatnak.

A $d\omega$ *differenciális térszög*et a θ,ϕ polárszögekkel is kifejezhetjük. Tegyük fel, hogy a θ szög $d\theta$-val a ϕ szög pedig $d\phi$-vel megváltozik! A változás alatt az irányvektor egy kicsiny téglalapot söpör végig, amelynek délkör menti mérete $d\theta$, szélességi kör menti mérete pedig $\sin\theta \cdot d\phi$ (a 8.1. ábra jobb oldala), így a differenciális térszög

$$d\omega = \sin\theta \cdot d\phi d\theta. \tag{8.1}$$



8.2. ábra. *A dω differenciális térszögben látható dA felületelem nagysága*

A számítások során gyakran szükségünk van arra a térszögre, amelyben egy adott felület egy pontból látszik. Tekintsünk egy infinitezimális felületelemet, hiszen az erre vonatkozó eredményekből tetszőleges felületre megoldást adhatunk integrálással! Egy dA felületelem egy $\vec{p}$ pontból

$$d\omega = \frac{dA \cdot \cos\theta}{r^2} \tag{8.2}$$

térszög alatt látszik, ahol r a $\vec{p}$ pont és dA felületelem távolsága, θ pedig a dA felület normálisa és a $\vec{p}$ iránya közötti szög (8.2. ábra). Ezzel az összefüggéssel a felület szerinti integrálást térszög szerinti integrálással válthatjuk fel.

8.1.1. A fényerősség alapvető mértékei

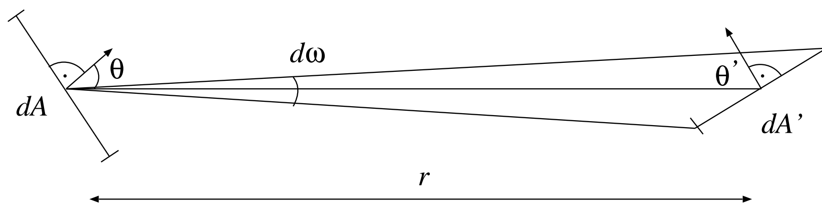
Egy adott felületen egységnyi idő alatt átlépő energiát *teljesítmény*nek, vagy *fluxus*nak nevezzük. A fluxus mértékegysége a watt $[W]$. Ha csupán egy kicsiny hullámhossz tartományt tekintünk, például a $[\lambda, \lambda + d\lambda]$-t, akkor a részecskemodell szerint a teljesítmény az átlépő fotonok számával arányos. A fluxus hullámhosszról-hullámhosszra változhat, tehát a spektrum meghatározásához hullámhosszfüggvényekkel kellene dolgoz-

nunk. A számítások során azonban a folytonos függvények helyett a látható tartományban néhány (3, 8, 16 stb.) reprezentatív hullámhosszt választunk ki, és a tényleges számításokat csak ezeken végezzük el. A reprezentatív hullámhosszok között a kapott értékekből interpolálunk. A vizsgálatunkat a továbbiakban λ hullámhosszú, *monokromatikus* (azaz csak azonos hullámhosszú hullámokat tartalmazó) fényre végezzük el, mivel a teljes spektrumban történő analízis több ilyen elemzésre vezethető vissza. Az anyagjellemzők nyilván függhetnek a megadott hullámhossztól.

A fluxus értéke önmagában nem mond semmit, mert mindig tisztázni kell, hogy pontosan milyen felületen átlépő energiát vizsgálunk. Egy nagy fluxusérték tehát lehet egyrészt annak a következménye, hogy erős sugárzó van a közelben, másrészt annak is, hogy nagy felületet tekintünk. Ezért a számítógépes grafikában a fluxus helyett általában annak sűrűségét, a sugársűrűséget használjuk. A *sugársűrűség*, *radiancia*, vagy *intenzitás* (L), egy dA felületelemet $d\omega$ térszögben elhagyó $d\Phi$ infinitezimális fluxus osztva a kilépési irányból látható differenciális területtel ($dA \cdot \cos\theta$) és a térszöggel:

$$L = \frac{d\Phi}{dA \cdot d\omega \cdot \cos\theta}. \tag{8.3}$$

8.1.2. A fotometria alaptörvénye



8.3. ábra. *Két infinitezimális felületelem között átadott fluxus*

Miután megismerkedtünk az alapvető mennyiségekkel, nézzük meg, hogy miként határozhatók meg egy olyan elrendezésben, ahol egy dA felületelem kibocsátott fényteljesítménye egy másik dA' felületelemre jut (8.3. ábra)! Ha a felületelemek látják egymást, és a dA intenzitása a dA' irányába L, akkor a 8.3. egyenlet szerint az átadott fluxus:

$$d\Phi = L \cdot dA \cdot d\omega \cdot \cos\theta.$$

A 8.2. egyenlet felhasználásával a térszöget kifejezhetjük a látható felületelem dA' területével. Ezzel egy alapvető egyenlethez jutunk, amely a *fotometria alaptörvénye*:

$$d\Phi = L \cdot \frac{dA \cdot \cos\theta \cdot dA' \cdot \cos\theta'}{r^2}. \tag{8.4}$$

Ezen egyenlet szerint az átadott fluxus egyenesen arányos a forrás sugársűrűségével, a forrás és az antenna látható területével és fordítottan arányos a távolságukkal. Vegyük észre, hogy a 8.2. egyenlet alkalmazásával az átadott teljesítmény a következő alakban is felírható:

$$d\Phi = L \cdot dA' \cdot \frac{dA \cdot \cos\theta}{r^2} \cdot \cos\theta' = L \cdot dA' \cdot d\omega' \cdot \cos\theta', \qquad (8.5)$$

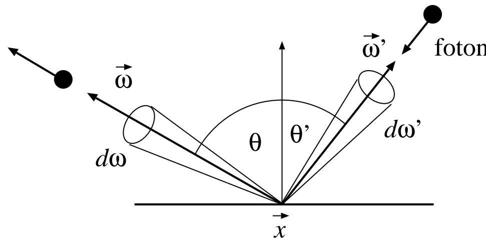
amely szerint ugyanolyan képlet vonatkozik a sugárzó felületelemre (8.3. egyenlet), mint a sugárzást felfogó antennára. Ez az egyenlet az egyik oka annak, hogy a számítások során a fénysugarak megfordíthatók, azaz ahelyett, hogy a fénysugarakat a fényforrásból a szem irányába követnénk, a szemből is közeledhetünk a fényforrások felé.

8.2. A fény–felület kölcsönhatás: az árnyalási egyenlet

A megvilágított felület a beérkező fényteljesítmény egy részét különböző irányokba visszaveri, míg másik részét elnyeli. Az optikailag tökéletesen sima felületekre a visszaverődést a *visszaverődési törvény*, a fénytörést pedig a *Snellius – Descartes törvény* írja le (4. fejezet). A felületi egyenetlenségek miatt azonban a valódi felületek bármely irányba visszaverhetik, illetve törhetik a fényt. Az ilyen „kiszámíthatatlan” hatásokat a valószínűségszámítás eszközeivel írhatjuk le. Tegyük fel, hogy az $\vec{\omega}'$ irányból egy foton érkezik a felület $\vec{x}$ pontjába! A foton az $\vec{\omega}$ irányban a következő *visszaverődési-sűrűségfüggvény* szerinti valószínűséggel halad tovább:

$$w(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot d\omega = \Pr\{\text{a foton az } \vec{\omega} \text{ körüli } d\omega \text{ térszögben megy} \mid \vec{\omega}' \text{ irányból jön}\}.$$

Erősen tükröző felületeknél nagy a valószínűsége annak, hogy a foton az elméleti visszaverődési irány közelében halad tovább. Matt felületeknél viszont a különböző irányokban történő kilépés hasonló valószínűségű.



8.4. ábra. *Az $\vec{\omega}'$ irányból érkező fotonok visszaverődése az $\vec{\omega}$ körüli $d\omega$ térszögbe*

Most térjünk rá annak vizsgálatára, hogy a felület egy adott irányból milyen fényesnek látszik! Az $\vec{\omega}$ irány körüli $d\omega$ térszögbe visszavert vagy tört fluxust megkaphatjuk, ha tekintjük az Ω *illuminációs gömb* összes lehetséges $\vec{\omega}'$ bejövő irányát, és az ezekből

érkező fluxusok hatását összegezzük. Egy $\vec{\omega}'$ iránybeli $d\omega'$ differenciális térszögből az $\vec{x}$ pontra illeszkedő dA felületre érkező fluxus, a fotometria alaptörvényének 8.5. egyenlet szerinti alakja szerint, következőképpen írható fel:

$$\Phi^{\rm in}(\vec{x}, dA, \vec{\omega}', d\omega') = L^{\rm in}(\vec{x}, \vec{\omega}') \cdot dA \cdot \cos\theta' \cdot d\omega',$$

ahol $L^{\rm in}(\vec{x}, \vec{\omega}')$ az $\vec{x}$ pontból az $-\vec{\omega}'$ irányba látható pontnak az $\vec{x}$ irányú sugársűrűsége, a θ' pedig a $-\vec{\omega}'$ irány és a felületi normális közötti szög (8.4. ábra). Egy rögzített hullámhosszon a fluxus arányos a beérkező fotonok számával. Annak valószínűsége, hogy egyetlen foton az $\vec{\omega}$ iránybeli $d\omega$ térszögbe verődik vissza a visszaverődési valószínűség-sűrűségfüggvény definíciója szerint $w(\vec{\omega}', \vec{x}, \vec{\omega})\; d\omega$. Ennek értelmében a visszavert fluxus várhatóan:

$$w(\vec{\omega}', \vec{x}, \vec{\omega})\; d\omega \cdot L^{\rm in}(\vec{x}, \vec{\omega}') \cdot dA \cdot \cos\theta' \cdot d\omega'.$$

Az $\vec{\omega}$ iránybeli $d\omega$ térszögbe visszavert teljes Φ^r fluxust megkapjuk, ha az összes bemeneti irányt tekintjük, és az onnan kapott fluxusokat összegezzük (integráljuk):

$$\Phi^r(\vec{x}, dA, \vec{\omega}, d\omega) = \int\limits_{\vec{\omega}' \in \Omega} w(\vec{\omega}', \vec{x}, \vec{\omega})\; d\omega \cdot L^{\rm in}(\vec{x}, \vec{\omega}') \cdot dA \cdot \cos\theta' d\omega'.$$

Amennyiben a felület maga is fényforrás, a visszavert fény fluxusán kívül a

$$\Phi^e(\vec{x}, \vec{\omega}) = L^e(\vec{x}, \vec{\omega}) \cdot dA \cdot \cos\theta \cdot d\omega$$

kisugárzott fénymennyiség is hozzájárul a kimeneti fluxushoz ($\Phi^{\rm out} = \Phi^e + \Phi^r$). A kimeneti fluxus képletében a θ szög az $\vec{\omega}$ irány és a felületi normális közötti szög. A kimeneti fluxus és a radiancia közötti 8.3. összefüggést felhasználva:

$$\Phi^{\rm out}(\vec{x}, dA, \vec{\omega}, d\omega) = L(\vec{x}, \vec{\omega}) \cdot dA \cdot \cos\theta \cdot d\omega.$$

A kimeneti fluxust, mint a kisugárzott és a visszavert fluxusok összegét, a sugársűrűségek segítségével is felírhatjuk:

$$L(\vec{x}, \vec{\omega}) \cdot dA \cdot \cos\theta \cdot d\omega = L^e(\vec{x}, \vec{\omega}) \cdot dA \cdot \cos\theta \cdot d\omega + \int\limits_{\Omega} w(\vec{\omega}', \vec{x}, \vec{\omega})\; d\omega \cdot L^{\rm in}(\vec{x}, \vec{\omega}') \cdot dA \cdot \cos\theta' d\omega'.$$

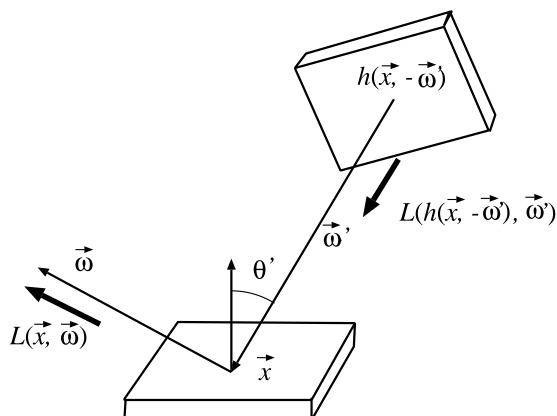
Osszuk el az egyenlet mindkét oldalát $dA \cdot d\omega \cdot \cos\theta$-val:

$$L(\vec{x}, \vec{\omega}) = L^e(\vec{x}, \vec{\omega}) + \int\limits_{\Omega} L^{\rm in}(\vec{x}, \vec{\omega}') \cdot \cos\theta' \cdot \frac{w(\vec{\omega}', \vec{x}, \vec{\omega})}{\cos\theta}\; d\omega'. \tag{8.6}$$

A foton haladását leíró valószínűség-sűrűségfüggvény és a kimeneti szög koszinuszának hányadosa, az optikai anyagmodellek egy alapvető mennyisége, amelynek neve *kétirányú visszaverődés eloszlási függvény*, vagy röviden *BRDF* (*Bi-directional Reflection Distribution Function*):

$$f_r(\vec{\omega}', \vec{x}, \vec{\omega}) = \frac{w(\vec{\omega}', \vec{x}, \vec{\omega})}{\cos\theta}. \tag{8.7}$$

A BRDF mértékegysége 1 per szteradián $[sr^{-1}]$. A BRDF első paramétere a fény bejövő irányát, a második paramétere a felületi pontot, a harmadik paramétere pedig a kilépő irányt azonosítja.



8.5. ábra. *Az árnyalási egyenlet geometriája*

A 8.6. egyenlet szerint az $L^{\text{in}}(\vec{x},\vec{\omega}')$ bejövő sugársűrűség egyenlő az $\vec{x}$ pontból a $-\vec{\omega}'$ irányba látható $\vec{y}$ pont $\vec{\omega}'$ irányú sugársűrűségével. Vezessük be az

$$\vec{y} = h(\vec{x},\vec{\omega}')$$

*láthatóság függvény*t, amely megmondja, hogy egy pontból egy adott irányba milyen másik felületi pont látszik! Ezzel végre eljutottunk a fényátadás alapvető integrálegyenletéhez, az *árnyalási egyenlet*hez (*rendering equation*) [67]:

$$L(\vec{x},\vec{\omega}) = L^e(\vec{x},\vec{\omega}) + \int_{\Omega} L(h(\vec{x},-\vec{\omega}'),\vec{\omega}') \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \, d\omega'. \qquad (8.8)$$

Az árnyalási egyenlet, bár bonyolultnak látszik, valójában rendkívül egyszerűen értelmezhető. Egy felületi pont adott irányú sugársűrűsége ($L(\vec{x},\vec{\omega})$) megegyezik a felületi pont ilyen irányú saját emissziójának ($L^e(\vec{x},\vec{\omega})$) és a különböző irányokból ide jutó ($L(h(\vec{x},-\vec{\omega}'),\vec{\omega}')$) sugársűrűségnek az adott irányba történő visszaverődésének az összegével. A visszaverődést az $f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \, d\omega'$ tag jellemzi, amely lényegében annak a fényútnak a valószínűségét határozza meg, amely a nézeti irányt a visszaverődésen keresztül a $d\omega'$ elemi térszöggel köti össze. Minden egyes árnyalási feladat annyi árnyalási egyenlettel adható meg, ahány reprezentatív hullámhosszon dolgozunk. Az L^e emisszió és az $f_r(\vec{\omega}',\vec{x},\vec{\omega})$ BRDF a hullámhossztól függenek.

Vezessük be a fény–felület kölcsönhatást leíró $\mathcal{T}_{f_r}$ integráloperátort:

$$(\mathcal{T}_{f_r}L)(\vec{x},\vec{\omega}) = \int_{\Omega} L(h(\vec{x},-\vec{\omega}'),\vec{\omega}') \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \, d\omega'.$$

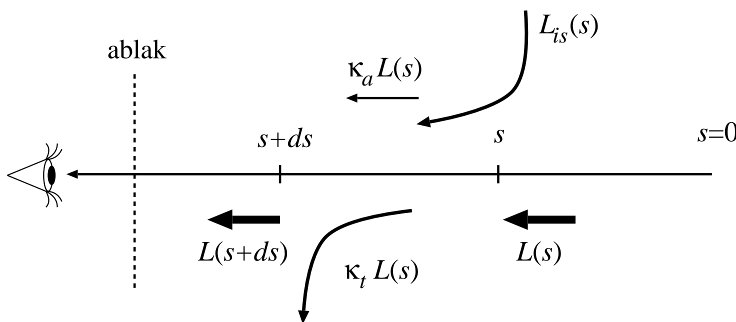
Ez az integráloperátor egy sugársűrűség-függvényből kiszámítja annak egyszeres visszaverődését. A fényátadás operátor felhasználásával felállíthatjuk az árnyalási egyenlet rövid alakját:

$$L = L^e + \mathcal{T}_{f_r} L. \tag{8.9}$$

Az egyenlet ismeretlene az L sugársűrűség-függvény.

8.3. Térfogati fényjelenségek

Az árnyalási egyenlet származtatása során feltételeztük, hogy a felületek között a fényintenzitás nem csökken, azaz a térben nincsenek fényelnyelő és szóró anyagok (*participating media*). Ha *felhő*t, tüzet, *füst*öt, *köd*öt stb. szeretnénk megjeleníteni, akkor a korábbi feltételezésekkel alkotott modellek elégtelennek bizonyulnak, tehát általánosítani kell őket.



8.6. ábra. *A sugár intenzitásának változása fényelnyelő közegben*

Tekintsünk egy fényelnyelő, fényszóró, sőt akár fényemittáló anyagon áthaladó sugarat! Egy ds elemi szakaszon a sugár L intenzitásának megváltozása több tényező függvénye:

- A fény a pálya mentén elnyelődik, illetve az eredetileg sugárirányú fotonok más irányba szóródnak az anyag molekuláival bekövetkező ütközések során. Ezen hatás következménye egy $-\kappa_t \cdot L \cdot ds$ mértékű változás, ahol κ_t annak valószínűsége, hogy egy foton az egységnyi intervallumon ütközik az anyag részecskéivel, amely az anyag sűrűségétől, illetve átlátszóságától függ (*outscattering*).
- A fényintenzitás az anyag saját emissziójával növekedhet: $\kappa_a \cdot L^e \cdot ds$.
- Az eredetileg más irányú fotonok a molekulákba ütközve éppen a sugár irányában folytatják az útjukat (*inscattering*). Ha az $\vec{\omega}'$ irányból az elemi ds szakasz környezetébe

$L_i(s,\vec{\omega}')$ radiancia érkezik, az $\vec{\omega}$ sugárirányban történő visszaverődés valószínűség-sűrűségfüggvénye pedig $f(\vec{\omega}',\vec{\omega})$, akkor ez a hatás az intenzitást

$$L_{\rm is}(s)\cdot ds = \left(\int_{\Omega} L_i(s,\vec{\omega}')\cdot f(\vec{\omega}',\vec{\omega})\; d\omega'\right)\cdot ds$$

mennyiséggel növeli.

A fenti változásokat összefoglalva, és a változást az intervallum ds hosszával osztva a sugár intenzitására a következő egyenletet állíthatjuk fel:

$$\frac{dL(s,\vec{\omega})}{ds} = -\kappa_t(s)\cdot L(s,\vec{\omega}) + \kappa_a(s)\cdot L^e(s,\vec{\omega}) + L_{\rm is}(s,\vec{\omega}) =$$

$$-\kappa_t(s)\cdot L(s,\vec{\omega}) + \kappa_a(s)\cdot L^e(s,\vec{\omega}) + \int_{\Omega} L_i(s,\vec{\omega}')\cdot f(\vec{\omega}',\vec{\omega})\; d\omega'. \qquad (8.10)$$

Ebben az egyenletben az ismeretlen sugársűrűség több helyen is szerepel, megtalálható derivált formában, normál alakban, sőt az L_i mögé rejtve még integrálva is. Mivel a feladat sokkal egyszerűbb lenne, ha az L_i független lenne az ismeretlen sugársűrűségtől, és valaki megsúgná nekünk az $L_{\rm is}$ értékét, a gyakorlatban sokszor olyan egyszerűsítő feltételezéseket teszünk, amelyek ehhez az esethez vezetnek. Ekkor a fénynek csak az egyszeres szóródását (*single scattering*) számítjuk, a többszörös szóródást (*multiple scattering*) elhanyagoljuk. Az egyszeres szóródást leíró

$$\frac{dL(s,\vec{\omega})}{ds} = -\kappa_t(s)\cdot L(s,\vec{\omega}) + \kappa_a(s)\cdot L^e(s,\vec{\omega}) + L_{\rm is}(s,\vec{\omega})$$

egyszerűsített differenciálegyenlet ismeretlen L függvényét már a differenciálegyenletek szokásos megoldási módszereivel is kifejezhetjük. A következő megoldás helyességéről behelyettesítéssel is meggyőződhetünk:

$$L(s,\vec{\omega}) = e^{-\int_0^s \kappa_t(\tau)\, d\tau}\cdot L(0,\vec{\omega}) + \int_0^s (\kappa_a(t)\cdot L^e(t,\vec{\omega}) + L_{\rm is}(t,\vec{\omega}))\cdot e^{-\int_t^s \kappa_t(\tau)\, d\tau}\, dt.$$

8.4. A képszintézis feladat elemei

A képszintézis feladatban a láthatóságfüggvénybe elbújtatva megtaláljuk a felületek geometriáját, az anyagtulajdonságokat leíró BRDF-et, az emissziót, amelyet a fényforrásmodellekből kapunk meg, valamint a kamerát. Ezeket nevezzük a képszintézis feladat elemeinek. A geometriával már a 3. fejezetben foglalkoztunk, most pedig a többi elemet vizsgáljuk részleteiben.

8.4.1. BRDF-modellek

Valósághű képek előállítása során olyan BRDF-modelleket kell használnunk, amelyek nem sértik az alapvető fizikai törvényeket, mint például a BRDF-k szimmetriáját kimondó *Helmholtz-törvény*t, vagy az *energiamegmaradás* törvényét.

A Helmholtz-féle szimmetria, vagy *reciprocitás* [93] szerint a fénysugár megfordítható, azaz a BRDF-ben a bejövő és kimenő irányok felcserélhetőek:

$$f_r(\vec{\omega},\vec{x},\vec{\omega}') = f_r(\vec{\omega}',\vec{x},\vec{\omega}). \tag{8.11}$$

Ez a tulajdonság az, amely miatt a valószínűség-sűrűségfüggvényekkel szemben a BRDF-eket részesítjük előnyben az optikai anyagmodellek megadásánál.

Az energiamegmaradás elve értelmében, egy önállóan nem sugárzó felületelem nem adhat ki több fotont (nagyobb fluxust), mint amit maga kapott, vagy másképpen, a tetszőleges irányú visszaverődés teljes valószínűsége nyilván nem lehet egynél nagyobb. A tetszőleges irányú visszaverődés valószínűségét *albedó*nak nevezzük. Az albedó definíciója:

$$a(\vec{x},\vec{\omega}') = \int\limits_{\Omega_H} f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta \, d\omega \leq 1. \tag{8.12}$$

Az energiamegmaradás elvének következménye, hogy a fényátadás operátor egymás utáni alkalmazása során a visszavert sugársűrűség zérushoz tart. Miként a megoldási módszerek ismertetésénél látni fogjuk, ez a tulajdonság biztosítja, hogy a megoldások konvergálnak. A reciprocitást és az energiamegmaradás elvét nem sértő BRDF-eket *fizikailag plauzibilis*nek nevezzük [86].

A bevezetett BRDF és albedó nem csupán absztrakt fogalmak, hanem adott megvilágítási körülmények között ezek láthatóvá is válnak. A BRDF, pontosabban a BRDF és a bejövő szög koszinuszának szorzata, az anyag pontszerű megvilágításra adott válaszát írja le. Ha a pontszerű fényforrásból a felületre az $\vec{\omega}'$ irányból egységnyi sugársűrűség érkezik, akkor az árnyalási egyenlet szerint a visszavert sugársűrűség:

$$L = \int\limits_{\Omega_H} L^{in}(\vec{x},\vec{\omega}') \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \, d\omega' = f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta'.$$

Figyeljük meg, hogy az integrálási tartományban egyetlen irányban nem zérus a bejövő sugársűrűség, így az integrálból az integrandus egyetlen értéke marad (matematikailag ez a *Dirac-delta* integrálását jelenti)!

Ha a bejövő sugársűrűség ugyancsak egységnyi, de minden irányban homogén (égboltszerű), akkor a visszavert sugársűrűség:

$$L = \int\limits_{\Omega_H} 1 \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \, d\omega' = \int\limits_{\Omega} 1 \cdot f_r(\vec{\omega},\vec{x},\vec{\omega}') \cdot \cos\theta' \, d\omega' = a(\vec{x},\vec{\omega}),$$

amennyiben a BRDF szimmetrikus. Az albedó tehát a homogén égbolt fény megvilágítás mellett látható.

A 4. fejezetben már megismerkedtünk a legfontosabb BRDF-modellekkel. Most ismét áttekintjük őket és ellenőrizzük a fizikai érvényességüket. A BRDF-modellek bemutatása során a következő jelöléseket használjuk: $\vec{N}$ a felületelemre merőleges egységvektor, $\vec{L}$ a fényforrás irányába mutató egységvektor, $\vec{V}$ a nézőirányba mutató egységvektor, $\vec{R}$ az $\vec{L}$ tükörképe az $\vec{N}$-re vonatkoztatva, $\vec{H}$ az $\vec{L}$ és $\vec{V}$ közötti felező egységvektor.

Diffúz visszaverődés

A *diffúz* anyagokról visszavert sugársűrűség független a nézeti iránytól. A Helmholtz-féle reciprocitás értelmében a BRDF ekkor a bejövő iránytól sem függhet, azaz a BRDF irányfüggetlen konstans:

$$f_r(\vec{L}, \vec{V}) = k_d. \tag{8.13}$$

Az energiamegmaradás miatt az albedó diffúz visszaverődés esetén sem lehet 1-nél nagyobb, így a k_d diffúz visszaverődési együtthatóra a következő korlát állítható fel:

$$a(\vec{L}) = \int_{\Omega_H} k_d \cdot \cos\theta \, d\omega = k_d \cdot \pi \qquad \Longrightarrow \qquad k_d \leq \frac{1}{\pi}. \tag{8.14}$$

A valódi diffúz felületeknél, a visszaverési együttható tehát legfeljebb $1/\pi \approx 0.3$ lehet. Ezzel szemben a lokális illuminációs számításoknál nem ritkán 0.8-nál is nagyobb értékeket adunk meg. Bár ezzel vétünk a fizikai törvények ellen, mentségünkre szolgáljon, hogy a lokális illuminációs számításokban úgyis jelentős elhanyagolásokat teszünk (például a többszörös visszaverődéseket figyelmen kívül hagyjuk), ezért a hiányzó energiát az irreálisan nagy visszaverődési tényezők segítségével lopjuk vissza. A globális illuminációs számítások során viszont 0.3-nál nagyobb diffúz visszaverődési tényezőket ne használjunk!

Spekuláris visszaverődés

A Phong-BRDF a spekuláris visszaverődés egyszerű empirikus modellje [101], amely a visszavert és beérkező sugársűrűség arányának a tükörirány és a valódi nézeti irány közötti szögtől való függését egy $\cos^n$ függvénnyel írta le, ahol az n a felület optikai simaságát, „polírozottságát” fejezi ki. Ebből a BRDF-re a következő képlet adódik:

$$f_{r,\text{Phong}}(\vec{L}, \vec{V}) = k_s \cdot \frac{(\vec{R} \cdot \vec{V})^n}{(\vec{N} \cdot \vec{L})} \tag{8.15}$$

ahol $\vec{R}$ az $\vec{L}$ vektor tükörképe a felületi normálisra. A k_s faktor a Fresnel-együtthatóval arányos, de annál kisebb, hiszen a felület most nem ideális tükör.

Az eredeti Phong-modell fizikailag nem plauzibilis, mert nem szimmetrikus. Ezért a globális illuminációs számításokban ehelyett a következő változatokat használják [59]:

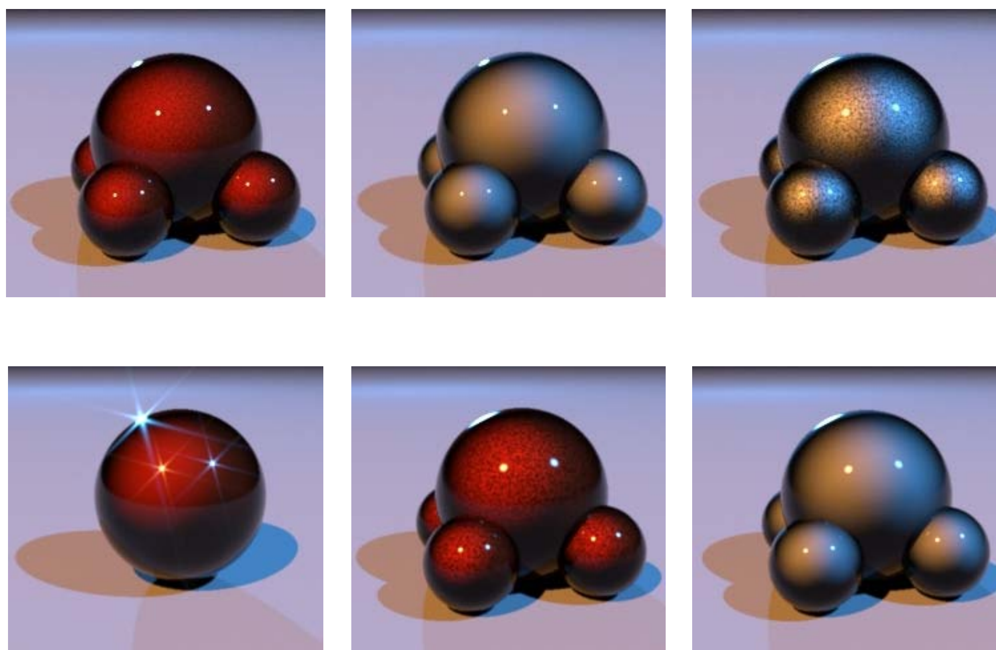
$$f_{r,\text{reciprocalPhong}}(\vec{L},\vec{V}) = k_s \cdot (\vec{R}\cdot\vec{V})^n \tag{8.16}$$

Az ilyen modell által visszavert sugársűrűség nagy beesési szögekre zérushoz tart, ami nem felel meg a gyakorlati tapasztalatainknak. Ezt a hiányosságot küszöböli ki a következő, az Arnold (8.9., 8.10. és 8.8. képek) és RenderX (8.13. ábra) programokban is használt *max-Phong* változat [95]:

$$f_{r,\text{maxPhong}}(\vec{L},\vec{V}) = k_s \cdot \frac{(\vec{R}\cdot\vec{V})^n}{\max\left((\vec{N}\cdot\vec{V}),(\vec{N}\cdot\vec{L})\right)}. \tag{8.17}$$

Az energiamegmaradáshoz a következő feltételt kell betartani [83]:

$$k_s \leq \frac{n+2}{2\pi}. \tag{8.18}$$



8.7. ábra. *Max-Phong BRDF-ek keverése (Marcos Fajardo)*

8.8. ábra. *Max-Phong BRDF és inverz fénykövetés (modell: Adam York (NewKat Studios), program: Arnold (Marcos Fajardo))*

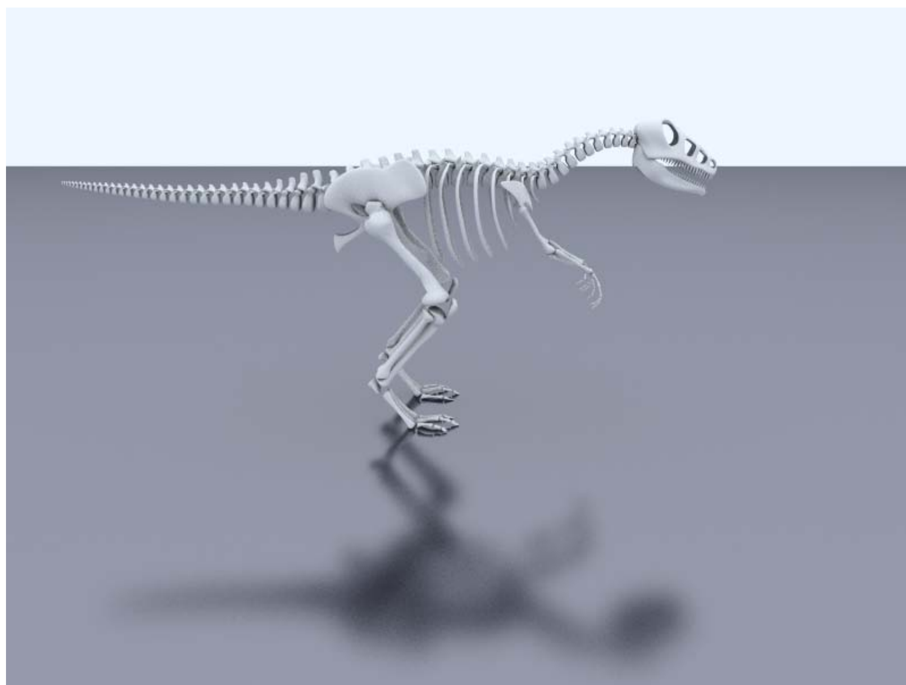
Ha a k_s paramétert a Fresnel-együttható alapján határozzuk meg, akkor gondot jelent az, hogy milyen beesési szögre tekintsük annak az értékét. A felületi normális és a fényvektor szöge most nem megfelelő, egyrészt azért, mert ekkor a BRDF nem lesz szimmetrikus, másrészt azért, mert a felületi egyenetlenségek következtében egy pontban a tényleges normálvektor nem állandó, hanem valószínűségi változó. Ha a felületet kis, véletlenszerűen orientált ideális tükrök gyűjteményének tekintjük, akkor azon felületelemek, amelyek $\vec{L}$-ből $\vec{V}$ irányba vernek vissza, a visszaverődési törvénynek megfelelően $\vec{H} = (\vec{L}+\vec{V})/2$ normálvektorral rendelkeznek. A beesés szögének koszinuszát a $(\vec{H} \cdot \vec{L})$ skalárszorzatból számolhatjuk ki.

8.9. ábra. *Max-Phong BRDF (program: Arnold/Marcos Fajardo)*

8.4.2. Mérőműszerek

Az árnyalási egyenlet megoldása után a sugársűrűséget minden felületi pontban és irányban ismerjük. A képelőállításhoz viszont azt kell tudnunk, hogy egy fényérzékeny eszköz (retina vagy film) egyes részein milyen teljesítményű fény halad keresztül. Egy *kamera* elemi kamerák, vagy mérőeszközök gyűjteményeként fogható fel, ahol minden elemi kamera egyetlen mennyiséget mér. Egy elemi kamera általában egy pixelen átjutó fényt detektál, de mérheti a felületelemet adott térszögben elhagyó fényteljesítményt is. Rendeljünk minden elemi kamerához egy $W^e(\vec{y}, \vec{\omega})$ *érzékenységfüggvény*t, amely megmutatja, hogy az $\vec{y}$ pontból az $\vec{\omega}$ irányba kibocsátott egységnyi energiájú foton mekkora hatást kelt a műszerünkben. Ha az elemi kamera a pixelen átjutó teljesítményt méri, akkor nyilván az érzékenységfüggvény valamilyen pozitív *C skálafaktor* azon pontokra és irányokra, amelyeket a szempozícióval összekötve éppen az adott irányt kapjuk, és minden más esetben zérus.

Az összes pont és irány hatását az elemi hatások összegeként írhatjuk fel. Egy L sugársűrűségű, az $\vec{y}$ pontból az $\vec{\omega}$ irányba kilépő nyaláb fluxusa $L(\vec{y}, \vec{\omega}) \cos\theta dy d\omega$, tehát

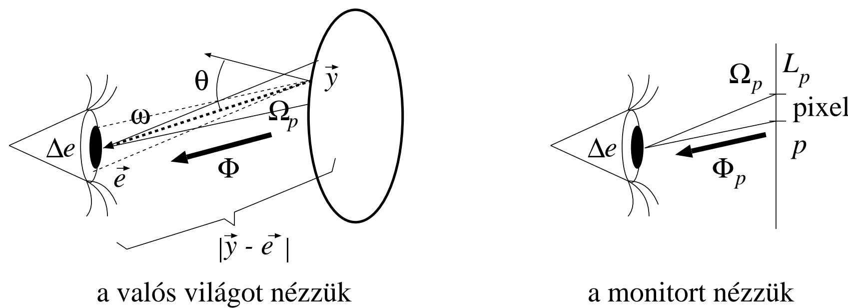


8.10. ábra. *Max-Phong BRDF (Gonzalo Rueda, program: Arnold/Marcos Fajardo)*

a mérőműszerben $W^e(\vec{y},\vec{\omega})$-szer ekkora hatást kelt. A teljes hatáshoz az S teljes felületet és az Ω illuminácós gömb összes irányát figyelembe kell venni:

$$\int_{\Omega}\int_{S} L(\vec{y},\vec{\omega})\cos\theta \cdot W^e(\vec{y},\vec{\omega})\; dy\, d\omega = \mathcal{M}L, \tag{8.19}$$

ahol $\mathcal{M}$ a *sugársűrűségmérő operátor*. A képletet a következőképpen értelmezhetjük. Ahhoz, hogy egy pixelen keresztül a szembe jutó teljesítményt meghatározzuk, számba kell venni a szemből a pixelen keresztül látható felületi pontok szemirányú sugársűrűségét ($L(\vec{y},\vec{\omega})$). A szemből látható pontokat és az innen a szembe mutató irányokat az érzékenységfüggvény jelöli ki ($W^e(\vec{y},\vec{\omega})$), amely csak akkor különbözik zérustól, ha $\vec{y}$ a pixelben látható, és $\vec{\omega}$ éppen a szem felé mutat.



8.11. ábra. *Az emberi szem modellje*

A *kameramodell* megalkotásához vizsgáljuk meg, hogy hogyan reagál az *emberi szem* a monitorból és a valós világból érkező ingerekre! Az emberi szemben egy lencse, ún. *pupilla* található, amelynek mérete Δe (8.11. ábra). A továbbiakban feltételezzük, hogy a pupilla a monitorhoz és a tárgyakhoz képest kicsiny. Amikor a szem a monitortól kap ingereket, a p pixelt Ω_p térszögben látjuk. Annak érdekében, hogy a monitorból érkező gerjesztés a valós gerjesztéssel egyezzen meg, a pixel által kibocsátott és a pupillára érkező Φ_p teljesítménynek a valós világból, a Ω_p térszögből a pupillára jutó Φ teljesítménynek kell megfelelnie. Amennyiben a pixel sugárzási intenzitása L_p, a 8.5. egyenlet szerint a pixelből a pupillára jutó teljesítmény:

$$\Phi_p = L_p \cdot \Delta e \cdot \cos\theta_e \cdot \Omega_p,$$

ahol θ_e a pupilla felületi normálisa és a pixel iránya által bezárt szög.

A kameramodellnek olyan P mért értéket kell előállítani, amelyet a *rasztertárba* írhatunk, és amellyel a monitort vezérelhetjük. Tételezzük fel, hogy ha a rasztertárba P értéket írunk, akkor a monitoron kibocsátott sugársűrűség éppen $L_p = P$ lesz. A monitor esetleges nem egységnyi erősítését, vagy nem linearitását kompenzálhatjuk úgy,

hogy a *lookup tábla (LUT)* segítségével a P értékeket előtorzítjuk a nemlineáris átviteli függvény inverzével (az eljárás *gamma-korrekció* néven vonult be a köztudatba).

Mivel elvárásunk szerint a pixelről érkező Φ_p fluxusnak meg kell egyeznie a valós világból, a pixelnek megfelelő térszögből érkező Φ fluxussal, a kameramodellnek a következő mért értéket kell szolgáltatnia:

$$P = L_p = \frac{\Phi_p}{\Delta e \cdot \cos\theta_e \cdot \Omega_p} = \frac{\Phi}{\Delta e \cdot \cos\theta_e \cdot \Omega_p}.$$

Rendeljünk egy mérőműszert ehhez a pixelhez! Műszerünk a pixelben — azaz az Ω_p térszögben — látható pontokra és azokra az irányokra érzékeny, amelyek a látható pontokat a pupillával összekötik. Formálisan ez a következő érzékenységfüggvénnyel adható meg:

$$W^e(\vec{y}, \vec{\omega}) = \begin{cases} C, & \text{ha } \vec{y} \text{ látható az } \Omega_p \text{ térszögben és } \vec{\omega} \text{ az } \vec{y}\text{-ból a pupillára mutat,} \\ 0, & \text{egyébként,} \end{cases} \tag{8.20}$$

ahol

$$C = \frac{1}{\Delta e \cdot \cos\theta_e \cdot \Omega_p}.$$

A 8.19. egyenlet szerint a műszer a következő mért értéket mutatja:

$$P = \mathcal{M}L = \int_{\Omega} \int_{S} L(\vec{y}, \vec{\omega}) \cdot W^e(\vec{y}, \vec{\omega}) \cdot \cos\theta \, dy d\omega. \tag{8.21}$$

Jelöljük a pixelen keresztül látható pontok halmazát S_p-vel! Ha a pupilla kicsiny, az érzékenységfüggvény csak egy kicsiny térszögben különböző zérustól, amelynek mérete, a térszög és a benne látható felület nagysága közötti 8.2. egyenlet szerint a következő:

$$\Delta\omega = \Delta e \cdot \frac{\cos\theta_e}{|\vec{y} - \vec{\text{eye}}|^2}$$

ahol $\vec{\text{eye}}$ a pupilla helye. Ha $\vec{y}$ látható, azaz az S_p-ben van, és az $\vec{\omega}$ irány éppen a pupilla felé mutat, akkor az érzékenységfüggvény értéke C, tehát a P mért értéket a következőképpen közelíthetjük:

$$\int_{\Delta\vec{\omega}} \int_{S_p} L(\vec{y}, \vec{\omega}_{\vec{y} \to \vec{\text{eye}}}) \cdot W^e(\vec{y}, \vec{\omega}) \cdot \cos\theta \, dy d\omega = \int_{S_p} L(\vec{y}, \vec{\omega}_{\vec{y} \to \vec{\text{eye}}}) \cdot C \cdot \cos\theta \cdot \Delta e \cdot \frac{\cos\theta_e}{|\vec{y} - \vec{\text{eye}}|^2} \, dy.$$

A C skálatényező értékét behelyettesítve:

$$P = \int_{S_p} L(\vec{y}, \vec{\omega}_{\vec{y} \to \vec{\text{eye}}}) \cdot \frac{\cos\theta}{\Omega_p \cdot |\vec{y} - \vec{\text{eye}}|^2} \, dy. \tag{8.22}$$

A pixelnek megfelelő térszöget ugyancsak a térszög és a benne látható felület nagysága közötti 8.2. egyenlettel kaphatjuk meg. Ha a pixel helye $\vec{p}$, területe A_p és a pixel normálvektora valamint a nézeti irány közötti szög θ_p, akkor

$$\Omega_p \approx \frac{A_p \cdot \cos\theta_p}{|\vec{p} - \vec{\mathrm{eye}}|^2}.$$

A szem és az ablak síkja közötti távolságot *fókusztávolság*nak nevezzük és f-fel jelöljük. A fókusztávolság felhasználásával, a 8.12. ábra szerint $|\vec{p} - \vec{\mathrm{eye}}| = f/\cos\theta_p$, amely alapján a pixel a szemből a következő térszögben látszik:

$$\Omega_p \approx \frac{A_p \cdot \cos\theta_p^3}{f^2}.$$

A 8.22. integrált a felület helyett a nézeti irányok halmazán, sőt a pixelen is kiszámolhatjuk. A 8.2. egyenlet felhasználásával, a differenciális felületet a nézeti irányok differenciális térszögével válthatjuk fel:

$$dy = \frac{|\vec{y} - \vec{\mathrm{eye}}|^2}{\cos\theta} \cdot d\omega_p,$$

ahol $d\omega_p$ azon térszög, amelyben a szemből a dy differenciális felület látható. A mért érték ez alapján:

$$P = \int_{\Omega_p} L(\vec{y}, \vec{\omega}) \cdot \frac{\cos\theta}{\Omega_p \cdot |\vec{y} - \vec{\mathrm{eye}}|^2} \cdot \frac{|\vec{y} - \vec{\mathrm{eye}}|^2}{\cos\theta}\, d\omega_p = \int_{\Omega_p} L(h(\vec{\mathrm{eye}}, -\vec{\omega}_p), \vec{\omega}_p) \cdot \frac{1}{\Omega_p}\, d\omega_p. \tag{8.23}$$

Vegyük észre, hogy a mért érték független mind a látható pont távolságától, mind pedig a látható felület orientációjától! Ez megfelel annak a tapasztalatnak, hogy egy objektumra (például a falra) ránézve ugyanolyan fényesnek érezzük akkor is, ha közelebb megyünk hozzá, vagy ha eltávolodunk tőle. A jelenséget azzal magyarázhatjuk, hogy amikor távolodunk a felülettől, bár az egységnyi felület által kibocsátott és a szembe jutó teljesítmény csökken a távolság négyzetével, az adott térszögben látható felület nagysága ugyanezen sebességgel nő.

Az Ω_p azon irányokat tartalmazza, amelyek keresztülmennek a pixelen. A mért értéket adó integrál az Ω_p térszög helyett az S_p területű pixelen is kiértékelhető (8.12. ábra). Ha a pixel a *fókusztávolság*hoz, azaz a szem és az ablak távolságához képest kicsiny, akkor élhetünk a következő közelítéssel:

$$\frac{d\omega_p}{\Omega_p} \approx \frac{dp}{A_p},$$

8.12. ábra. *A mért érték kiszámítása a pixelen (bal) és a felületen integrálva (jobb)*

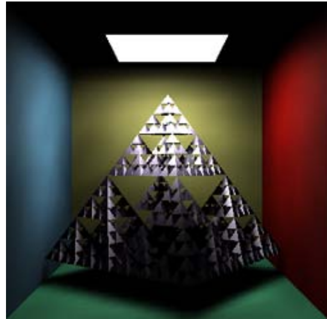
ahol A_p a pixel területe. Ez az összefüggés a térszögek szerinti integrálást a pixel felületén végrehajtott integrálással cseréli fel:

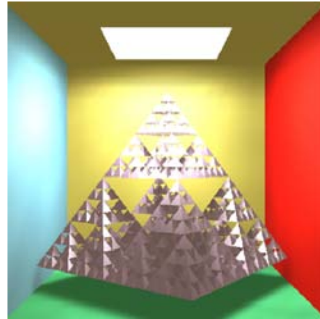
$$P = \int_{A_p} L(h(\vec{p}, -\vec{\omega}_{\vec{p}}), \vec{\omega}_{\vec{p}}) \cdot \frac{1}{A_p} \, dp. \tag{8.24}$$

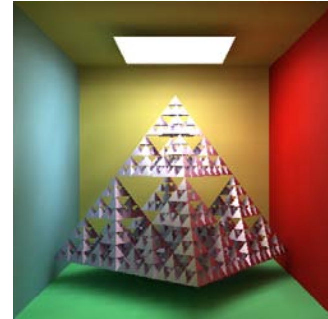
8.5. Az árnyalási egyenlet megoldása

Matematikai szempontból az árnyalási egyenlet (8.8. egyenlet) egy másodfajú *Fredholm-féle integrálegyenlet*, amelyben az ismeretlen L sugársűrűség-függvényt kell meghatározni. Ez a sugársűrűség-függvény egyrészt megjelenik önállóan a bal oldalon, másrészt az integrálon belül is. Azt is mondhatjuk, hogy az egyenletben az integrál és az azon kívüli részek között csatolás van, mert mindkettő függ az ismeretlen sugársűrűségtől. Szemléletesen, egy felületi pont sugárzása a visszaverődések miatt függhet a többi pont intenzitásától, azok sugárzása viszont akár éppen a kérdéses felületi pont fényességétől. Ez a kölcsönös függés kapcsolja össze a különböző pontok sugársűrűségét. Ilyen integrálegyenletek megoldása általában meglehetősen időigényes. Ha gyorsabban szeretnénk képet kapni, akkor a megoldandó feladat egyszerűsítéséhez folyamodhatunk, elfogadva azt is, hogy a fizikai modell egyszerűsítése a valósághűség romlásához vezethet. A rendelkezésre álló eljárásokat három nagy csoportba sorolhatjuk, amelyek a gyorsaság–valósághűség ellentmondó követelményeit különböző kompromisszummal elégítik ki.

A *lokális illuminációs algoritmus*ok az árnyalási egyenlet drasztikus egyszerűsítésével kiküszöbölnek mindenféle csatolást, azaz egy felület fényességének meghatározásához nem veszik figyelembe a többi felület fényességét. Megvilágítás csak a képen közvetlenül nem látható absztrakt fényforrásokból érkezhet. A csatolás megszűntetésével az árnyalási egyenletben az integrálból eltűnik az ismeretlen függvény, így az integrálegyenlet megoldása helyett csupán egy egyszerű integrált kell kiértékelnünk.


lokális illumináció


lokális illumináció
ambiens fényforrással


globális illumináció

rekurzív sugárkövetés

rekurzív sugárkövetés
területi fényforrással

globális illumináció

8.13. ábra. *Lokális illumináció, sugárkövetés és globális illumináció összehasonlítása*

A *rekurzív sugárkövetés* illuminációs algoritmusa a csatolást csak véges számú ideális visszaverődésre és törésre követi (6. fejezet).

A *globális illuminációs algoritmus*ok az árnyalási egyenletet a benne lévő csatolás elhanyagolása nélkül oldják meg, ily módon képesek a többszörös visszaverődések pontos kezelésére. A globális illuminációs algoritmusok azonosítják az összes olyan *fényutat*, amelyek a fényforrásokat — akár visszaverődéseken vagy töréseken keresztül — összekötik a szemmel, majd ezen fényutak képhez való hozzájárulását összegzik. Mivel a fényutak tere folytonos és sokdimenziós, az összegzés egy sokdimenziós integrál kiszámítását jelenti.

Mielőtt belemerülnénk a matematikai részletekbe, érdemes megvizsgálni, hogy a természet hogyan oldja meg ugyanezt a feladatot. Egy 100 wattos égő például másodpercenként körülbelül 10^{42} darab fotont bocsát ki, a természet pedig a fotonok útját fénysebességgel és egymással párhuzamosan „számítja” ki, sőt a számítási időre még a térben felhalmozott tárgyak száma sincs hatással. A kibocsátott fotonok az eltalált felületeken véletlenszerűen visszaverődnek vagy elnyelődnek, végül egy kis részük a megfigyelő szemébe jut, kialakítva a képet. Sajnos, amikor a globális illuminációt számítógépes szimulációval valósítjuk meg, nem áll rendelkezésünkre ilyen óriási számú, fénysebességgel működő számítógép, így a képet sokkal kevesebb, maximum néhány tízmillió fényút vizsgálatával kell előállítanunk. Akkor van esélyünk egyáltalán arra, hogy a természet 10^{42} darab fényútjához képest nevetségesen kevésnek tűnő néhány millió mintával is a valóságosnak megfelelő képet számítsuk ki, ha a fényutakat nagyon gondosan válogatjuk ki.

Három lényeges szempontot kell kiemelni:

- *Egyenletesen sűrű minták*: A fényútmintákat mindenütt elegendően sűrűn kell felvenni, különben fontos részek kimaradnának. Mint látni fogjuk, sokdimenziós terekben a szabályos rácsok nagyon egyenetlenek, ezért érdemes a mintákat inkább véletlenszerűen előállítani, amely a *Monte-Carlo módszer*ekhez vezet.

- *Fontosság szerinti mintavételezés*: Azokra az utakra kell összpontosítani, amelyek mentén jelentős fényteljesítmény halad, és nem érdemes olyan utak számítására időt vesztegetni, amelyekre legfeljebb néhány foton téved.

- *Koherencia*: Érdemes kihasználni, hogy a fényviszonyok nagyjából állandóak a felületeken, ezért ahelyett, hogy a pontokat egymástól teljesen függetlenül kezelnénk, nagyobb egységekre egyszerre kell a számításokat elvégezni.

Tekintsük először az egyenletesen sűrű minták előállítását, és egyelőre tételezzük fel, hogy az $\int_0^1 f(z)\,dz$ egydimenziós integrál kiszámításához használjuk őket! A legismertebb lehetőség a $z_1,\ldots,z_M$ minták *szabályos rács*on történő elhelyezése ($z_i = i/M$),

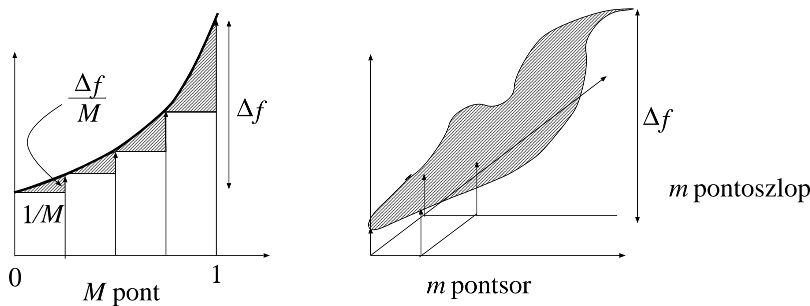
ami konstans súlyozással az integrál *téglányszabály* szerinti kiértékeléséhez vezet:

$$\int_0^1 f(z)\,dz \approx \frac{1}{M}\cdot\sum_{i=1}^{M} f(z_i).$$

A téglányszabály a görbe alatti területet téglalapok sorozatával közelíti, amelyek területe $f(z_i)\Delta z = f(z_i)/M$ (8.14. ábra). A közelítés hibája a téglalapok és a függvény közötti derékszögű „háromszögek” teljes területe, amelyek alapja $1/M$, darabszáma M, átlagos magassága pedig $\Delta f/2/M$, ahol Δf a függvény teljes megváltozása, így az integrálbecslés hibája:

$$\left|\int_0^1 f(z)\,dz - \frac{1}{M}\cdot\sum_{i=1}^{M} f(z_i)\right| \approx \frac{\Delta f}{2M}.$$

A hiba a mintaszámmal arányosan csökken, azaz tized akkora hibához tízszer annyi mintára van szükség, ami meglehetősen méltányosnak tűnik.



8.14. ábra. *A téglányszabály hibája egy- és kétdimenziós esetben*

A klasszikus integrálási szabályok a magasabb dimenziós integrálok becslését egydimenziós integrálok kiszámítására vezetik vissza.

Tekintsünk egy kétdimenziós $f(\mathbf{z}) = f(x,y)$ függvényt:

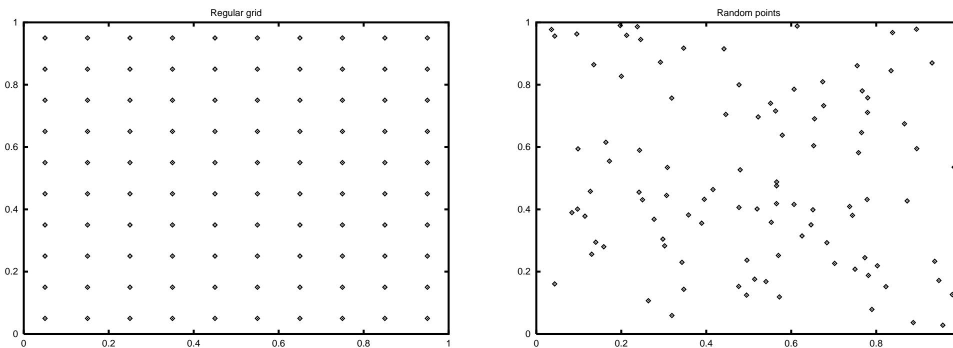
$$\int_{[0,1]^2} f(\mathbf{z})\,d\mathbf{z} = \int_0^1\int_0^1 f(x,y)\,dydx = \int_0^1\left(\int_0^1 f(x,y)\,dy\right)dx = \int_0^1 F(x)\,dx,$$

ahol $F(x)$ a belső függvény integrálja! Az $F(x)$ integráljának becsléshez az x tartományában felveszünk m darab $x_1,\ldots,x_j,\ldots,x_m$ pontot, és alkalmazzuk az egydimenziós becslést. Ehhez persze tudni kell az $F(x_j)$ értékét, ami maga is egy integrál, így az x_j mellett az y tartományában is ki kell jelölnünk még m darab y_k pontot. Az (x_j,y_k) kétdimenziós minták száma $M = m^2$. Az integrálbecslés pedig formailag az egydimenziós

becslésre hasonlít:

$$\int\limits_{[0,1]^2} f(\mathbf{z})\, d\mathbf{z} \approx \frac{1}{m^2} \cdot \sum_{j=1}^{m} \sum_{k=1}^{m} f(x_j, y_k) = \frac{1}{M} \cdot \sum_{i=1}^{M} f(\mathbf{z}_i).$$

Vizsgáljuk meg, hogy hogyan alakul a hiba! Mivel az F függvényt, mint egy egydimenziós integrált m mintával becsüljük, ennek hibája a korábbi eredmény alapján m-mel fordítottan arányos. Hasonlóképpen az F integrálásához megint m mintát használunk, így itt is m-mel fordítottan arányos hibát vétünk. A kétdimenziós integrálás hibája tehát $m = \sqrt{M}$-mel fordítottan arányos. Ez azt jelenti, hogy a hiba tizedére szorításához százszor annyi mintát, azaz százszor annyi számítási időt kell felhasználnunk, ami már nem tűnik nagyon kedvezőnek.



8.15. ábra. *100 mintapont szabályos rácson (bal) és véletlenszerűen (jobb)*

A gondolatmenetet tetszőleges számú dimenzióra kiterjeszthetjük és megállapíthatjuk, hogy egy D-dimenziós integrál klasszikus közelítésének hibája M^{-D}-vel arányos. A dolog egészen tragikussá válik magasabb dimenziókban. Például, ha a dimenzió 8, a hiba tizedére csökkentéséhez 10^8, azaz százmilliószor több mintát kell felhasználnunk. A globális illuminációs feladatnál pedig akár 20-dimenziós integrálok is előfordulhatnak. A szükséges minták száma és így a számítási idő a tartomány dimenziójával exponenciálisan, azaz robbanásszerűen nő. A jelenség magyarázata az, hogy magas dimenziókban a szabályos rács sorai és oszlopai között nagy űrök tátonganak, ezért a mintapontok nem töltik ki elegendően sűrűn az integrálási tartományt (8.15. ábra).

8.6. Monte-Carlo integrálás

A klasszikus integrálszabályok *dimenzionális robbanás*át elkerülhetjük, ha a mintapontokat nem egy szabályos rács mentén, hanem véletlenszerűen választjuk ki. Tekintsünk

egy D-dimenziójú $\mathbf{z} = [z_1, \ldots, z_D]$ pontokat tartalmazó V tartományt, és a tartomány felett integrálandó $f(\mathbf{z})$ függvényt! Szorozzuk be, és osszuk is el az $f(\mathbf{z})$ integrandust egy $p(\mathbf{z})$ valószínűség-sűrűségfüggvénnyel:

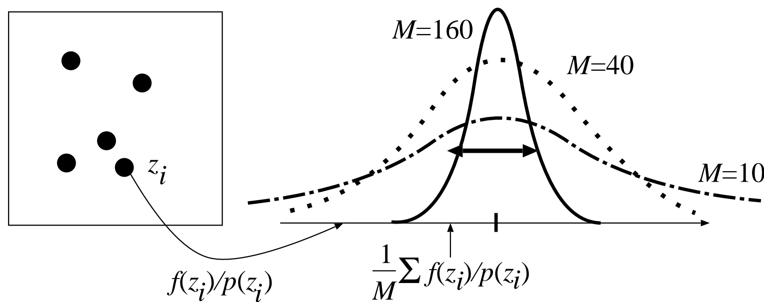
$$\int_V f(\mathbf{z})\, d\mathbf{z} = \int_V \frac{f(\mathbf{z})}{p(\mathbf{z})} \cdot p(\mathbf{z})\, d\mathbf{z}.$$

Ebben a formában az $f(\mathbf{z})/p(\mathbf{z})$ függvényt egy valószínűség-sűrűségfüggvénnyel súlyozva integráljuk. Vegyük észre, hogy ez éppen az $f(\mathbf{z})/p(\mathbf{z})$ *várható érték*ének képlete [104], ha a $\mathbf{z}$ változó sűrűségfüggvénye $p(\mathbf{z})$:

$$\int_V f(\mathbf{z})\, d\mathbf{z} = \int_V \frac{f(\mathbf{z})}{p(\mathbf{z})} \cdot p(\mathbf{z})\, d\mathbf{z} = E\left[\frac{f(\mathbf{z})}{p(\mathbf{z})}\right].$$

A várható értéket pedig jól becsülhetjük a véletlen minták átlagával, hiszen a *nagy számok törvénye* szerint a becslés 1 valószínűséggel a tényleges várható értékhez tart. Formálisan:

$$\int_V f(\mathbf{z})\, d\mathbf{z} = E\left[\frac{f(\mathbf{z})}{p(\mathbf{z})}\right] \approx \frac{1}{M} \cdot \sum_{i=1}^{M} \frac{f(\mathbf{z}_i)}{p(\mathbf{z}_i)}. \tag{8.25}$$



8.16. ábra. *Az átlag sűrűségfüggvénye a mintaszám függvényében*

Mivel a $\mathbf{z}_i$ minták véletlenszerűek, a fenti integrálbecslés is véletlen, azaz valószínűségi változó, amely a valódi integrálérték körül fluktuál. A fluktuáció mértékét a valószínűségi változó *szórás*a fejezi ki. Ahogy a mintaszámot növeljük, a fluktuáció egyre kisebb, így egyre jobban elhihetjük, hogy a véletlen eredmény közel van az integrálhoz. Vizsgáljuk meg, hogy milyen gyors ez a folyamat! Jelöljük a $p(\mathbf{z})$ sűrűségfüggvényű $\mathbf{z}$ valószínűségi változó $f(\mathbf{z})/p(\mathbf{z})$ transzformáltjának szórását σ-val!

Ha a mintákat egymástól függetlenül választjuk ki, akkor az M minta átlagának szórása $\sigma/\sqrt{M}$, tehát az átlagolásnak köszönhetően a szórás és így a fluktuáció is egyre kisebb lesz. A szórás és a klasszikus hiba fogalmát a *centrális határeloszlás tétel* segítségével kapcsolhatjuk össze, amely kimondja, hogy független valószínűségi változók átlaga előbb-utóbb Gauss-féle normális eloszlású lesz, az eredeti változók eloszlásától függetlenül (8.16. ábra). A Gauss-eloszlás harangszerű sűrűségfüggvénye alapján megállapíthatjuk, hogy annak valószínűsége, hogy a valószínűségi változó az átlagtól a szórás háromszorosánál kisebb mértékben tér el (azaz a haranggörbe alatti terület azon része, ahol a középtől legfeljebb a szórás háromszorosával távolodunk el) körülbelül 0.997. Ezek szerint 99.7% valószínűséggel mondhatjuk, hogy M kísérlet elvégzése után az integrálbecslés hibája $3\sigma/\sqrt{M}$-nél kisebb lesz. Vegyük észre, hogy a hibában sehol sem tűnik fel az integrálási tartomány dimenziója, tehát ez akkor is így lesz, ha az egy, kettő, nyolc, vagy éppenséggel 200 dimenziós!

A véletlen mintapontokkal dolgozó eljárást *Monte-Carlo módszer*nek nevezzük, amelynek ezek szerint nagy előnye, hogy a szükséges mintapontok száma nem függ a tartomány dimenziójától [111]. A dimenziófüggetlenség magyarázata az, hogy a véletlen pontok magasabb dimenzióban egyenletesebben sűrűek, mint a szabályos rácson kijelöltek. A szabályos rácsot ugyanis egy dimenziós felosztások sorozatával állítjuk elő, azaz egy pont elhelyezésénél csak egyetlen dimenzió egyenletes sűrű lefedését tartjuk szem előtt, emiatt magasabb dimenziókban a szabályos rács oszlopai és sorai között előbb-utóbb nagy űrök tátonganak. A véletlen pont véletlen koordinátái azonban egyszerre az összes koordinátatengely mentén megpróbálnak egyenletes sűrűséget felvenni. Emlékezzünk vissza arra, hogy a téglányszabály miként vezeti vissza a többdimenziós integrálást egydimenziós integrálok sorozatává! Az első koordináta x_j mintájának rögzítése után még m mintát vesz a második koordinátából, minden első és második koordinátához még újabb m mintát a harmadik koordinátából stb. Ennek következtében az első koordinátaminták a tartományukban csak nagyon gyéren bukkanhatnak fel. A Monte-Carlo integrál ezzel szemben egy első koordinátamintához egyetlen második, harmadik stb. koordinátát párosít, így egy első koordináta csak egyetlen sokdimenziós mintapont kialakításában vesz részt. Így aztán az első (és bármelyik) koordináta mentén a minták sűrűn ellepik az integrálási tartományt.

A Monte-Carlo módszer, mint a matematika megannyi más eredménye, Neumann János nevéhez kötődik.

8.6.1. Kvázi Monte-Carlo módszerek

A Monte-Carlo módszer a vakszerencsére bízza az egyenletesen sűrű mintaponthalmaz előállítását. A homo sapiens képességeibe vetett hitünk azt mondatja velünk, hogy lennie kell ennél jobb, determinisztikus stratégiának is, amely a véletlennél egyenletesebb, úgynevezett *alacsony diszkrepanciájú sorozat*okat eredményez. Valami olyan-

nak, amit akkor követünk, ha pontokat rajzolunk egy papírlapra úgy, hogy azok mindig nagyjából egyenletes sűrűséggel népesítsék be a rendelkezésre álló területet. Az elsőt nagyjából a lap közepére tesszük, a másodikat a közép és a bal alsó sarok közé, a harmadikat a jobb felső sarok környezetében stb. Egy tetszőleges dimenzióban működő módszer megismerését az egydimenziós $(0,1)$ tartomány felszabdalásával kezdjük [96, 102, 73, 111]. Egy jónak ígérkező stratégia az első pontot a szakasz felezőpontjába teszi. Ez a pont két szakaszra bontja a tartományt. A második és a harmadik pont ezen szakaszok felezőpontjai, ami most már négy új szakaszt hoz létre. Az előző szint szakaszainak a felezgetését pedig tetszőleges szintig folytathatjuk.

Az i-edik pont koordinátáit a következő, egyszerű algoritmussal számíthatjuk ki:

1. Felírjuk i-t kettes számrendszerben.

2. Tükrözzük a számot a végén levő kettedes pontra (például 100-ból 0.001 lesz).

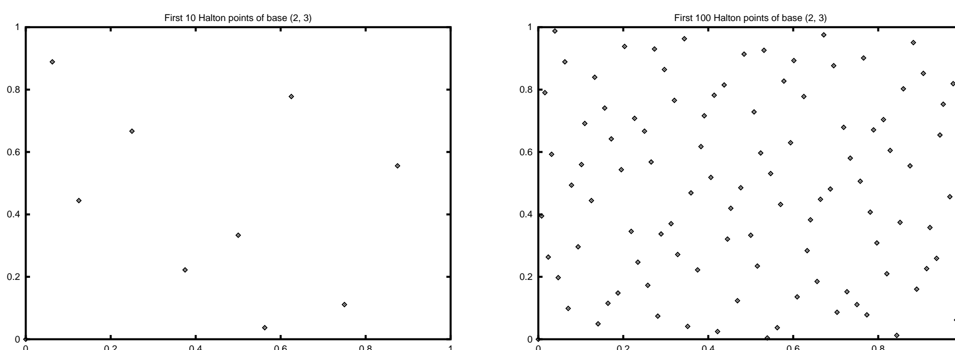
3. A kapott bináris törtszámot tekintjük a sorozat adott elemének.

i	i bináris formája	a bináris pontra vett tükörkép	H_i
1	1	0.1	0.5
2	10	0.01	0.25
3	11	0.11	0.75
4	100	0.001	0.125
5	101	0.101	0.625
6	110	0.011	0.375
7	111	0.111	0.875

8.1. táblázat. *Az első néhány kettes bázisú H_i Halton (Van der Corput) pont*

A sorozat egyenletességét a következőképpen láthatjuk be. A bináris formában minden L hosszú kombináció megjelenik, mielőtt az ennél hosszabb kombinációk feltűnnek. Ezért a tükörképben az első L jegyben minden kombinációt megkapunk, mielőtt egy olyan szám bukkanna fel, amelyik az első L jegyben megegyezne egy már szereplővel. Tehát az algoritmus csak akkor rak egy már létező pont 2^{-L} nagyságú környezetébe egy új pontot, ha már az összes 2^{-L} hosszú intervallumban van pont. Mivel ez minden L-re teljesül, sohasem fordulhat elő, hogy az intervallum egy részében a pontok sűrűsödnek, mialatt egy másik részében még nagyobb űrök találhatók.

A fenti konstrukció akkor is érvényben marad, ha nem kettes, hanem hármas, négyes stb. számrendszereket használunk, így végtelen különböző sorozatot állíthatunk elő. A kettes bázisú sorozatot *Van der Corput-sorozat*nak, a tetszőleges bázisút pedig *Halton-sorozat*nak nevezzük.



8.17. ábra. *Kétdimenzióban egyenletes Halton-sorozat első 10 és 100 pontja*

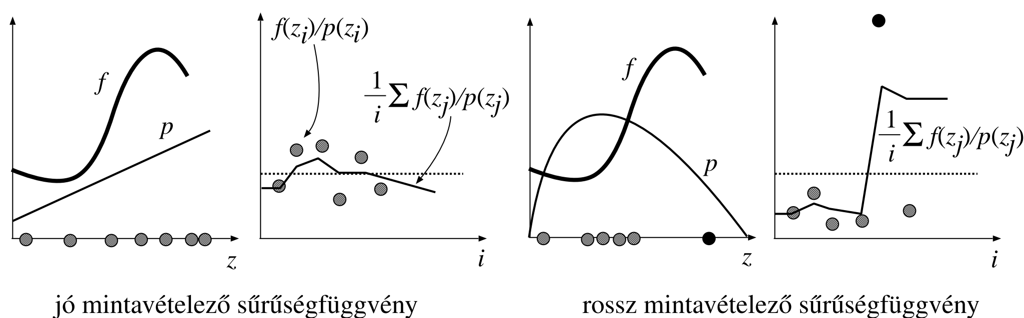
Most lépjünk a második dimenzióba! A pontoknak a kétdimenziós négyzetben két koordinátája van, amihez két Halton-sorozatot alkalmazhatunk. Nyilván a két Halton-sorozat nem lehet megegyező, azaz nem alapulhat ugyanazon a számrendszeren, hiszen a mintapontjaink ekkor csak a főátlóra kerülhetnének, ami aligha fedi le egyenletes sűrűséggel a négyzetet. Használjunk tehát két eltérő számrendszert a két koordinátához, amelyeket úgy kell megválasztani, hogy a sorozat most a kétdimenzióban is egyenletes legyen! Az egydimenziós, kettes bázisú Halton-sorozat esetén beláttuk, hogy egy 2^{-L} hosszú intervallumhoz csak minden 2^L-edik lépésben térünk vissza. Általában, ha a számrendszer alapja b, a módszer minden b^L-edik lépésben teszünk egy újabb pontot egy b^{-L} hosszú intervallumba. Ha kétdimenzióban az egyik koordinátához b_1-es számrendszert, a másikhoz pedig b_2-es számrendszert használunk, akkor egy b_1^{-L} széles oszlophoz minden b_1^L-edik lépésben, egy b_2^{-L} magas sorhoz pedig minden b_2^L-edik lépésben helyezünk el egy újabb pontot. A sorok és oszlopok metszésénél található cellákhoz a sor- és oszlopperiódus legkisebb közös többszörösének megfelelő periódussal találunk vissza. Az egyenletesség azt kívánja meg, hogy a cellaperiódus a cellák számával, azaz a sor- és oszlopperiódus szorzatával megegyező legyen, ami akkor következik be, ha a két szám legkisebb közös többszöröse a szorzatuk, azaz, ha b_1 és b_2 relatív prímek. Ezek a feltételek tetszőleges dimenzióban is igazak, tehát egy sokdimenziós integrálhoz a mintapontok koordinátáit olyan alapú sorozatokból kell venni, amelyek páronként relatív prímek. Kézenfekvő prímszámokat választani alapnak, hiszen azok mindenkivel relatív prímek. A 8.17. ábrán a vízszintes tengely mentén kettes, a függőleges mentén hármas számrendszert használtunk.

A következő osztály egy tetszőleges bázisú Halton-pontot állít elő, illetve a `Next` függvénye a sorozat következő elemét adja vissza egy gyors, inkrementális módszer alkalmazásával:

```
//=============================================================
class Halton {
//=============================================================
   float value, inv_base;                // érték és a bázis reciproka
public:
   Number(long i, int base) {            // base alapú sorozat i. elemére lép
      float f = inv_base = 1.0/base;
      value = 0.0;
      while ( i > 0 ) {
          value += f * (double)(i % base);
          i /= base;
          f *= inv_base;
      }
   }
   void Next() {                         // a sorozat közetkező elemére lép
      float r = 1.0 - value - 0.0000001;
      if (inv_base < r) value += inv_base;
      else {
         float h = inv_base, hh;
         do { hh = h; h *= inv_base; } while (h >= r);
         value += hh + h - 1.0;
      }
   }
   float Get() { return value; }         // az aktuális elem
};
```

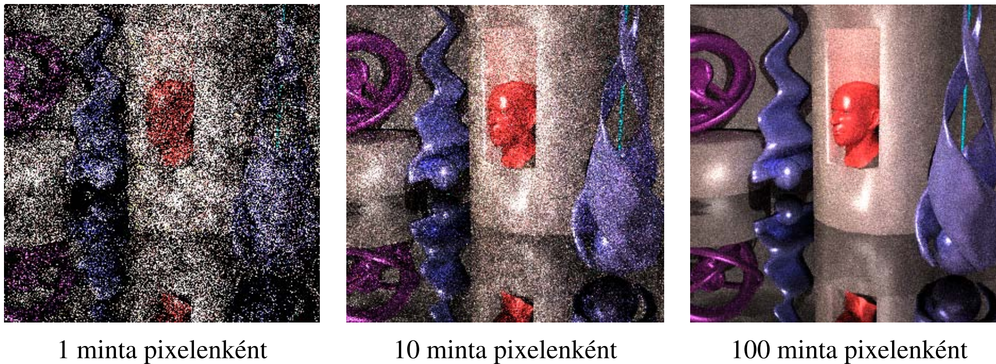
8.6.2. A fontosság szerinti mintavételezés

A Monte-Carlo integrálás $3\sigma/\sqrt{M}$ hibáját részben az $f(\mathbf{z})/p(\mathbf{z})$ valószínűségi változó σ szórása határozza meg. Az ebből adódó hibát úgy csökkenthetjük, hogy a minták $p(\mathbf{z})$ sűrűségét a lehetőségek szerint az integrandussal arányosan választjuk meg, azaz, ahol az integrandus nagy, oda sok mintapontot koncentrálunk. Ennek a szóráscsökkentő eljárásnak a neve *fontosság szerinti mintavételezés* (*importance sampling*).



8.18. ábra. *Fontosság szerinti mintavételezés*

A 8.18. ábra egy jó és egy rossz valószínűség-sűrűség alkalmazását mutatja. A bal oldali (jó) esetben a p valószínűség-sűrűség ott nagy, ahol az f integrandus nagy, így az $(1/M) \cdot \sum_{i=1}^{M} f(\mathbf{z}_i)/p(\mathbf{z}_i)$ integrálközelítő összegben szereplő f/p hányadosok nagyjából hasonló értékűek, és így az átlagukat kifejező integrálközelítő összegtől nem esnek messze. Ahogy egy új értéket adunk az integrálközelítő összeghez, az új érték alig változtatja meg ezt az átlagot, tehát az integrálközelítő összeg végig az átlag közelében marad, csak kis mértékben fluktuál körülötte. A jobb oldali (rossz) esetben van egy tartomány, ahol az f integrandus nagy, de a p valószínűség-sűrűség kicsi, azaz erre a tartományra csak igen ritkán tévedünk. Ha viszont nagy ritkán ide vet a szerencse, akkor a leolvasott nagy f integrandust egy kicsiny p értékkel osztjuk, amely óriási többletet jelent az integrálközelítő összegben.



1 minta pixelenként | 10 minta pixelenként | 100 minta pixelenként

8.19. ábra. *A Monte-Carlo módszerek jellegzetes pont zaja*

Az integrálközelítő összeg tehát sokáig a valódi átlag alatt mozog, amíg nem tévedünk a fontos tartományba. Ekkor viszont egy óriási értéket kap az összeg, ezért jelentősen az átlag fölé lendül és csak lassan tér vissza az átlaghoz. Az integrálközelítő összeg tehát erősen fluktuál az átlag körül. A képszintézisben minden pixelhez egy-egy integrált számítunk ki, amelyek rossz mintavételezés esetén sokáig a valódi értéknél kisebbek (azaz a színek sötétebbek) lesznek. Egyszer aztán egy pixel szerencséjére kap egy óriási többletet, így színe nagyon világossá válik, mialatt a kevésbé szerencsés szomszédai még mindig sötétek. A pixelünk szupernóvaként világlik fel a képernyőn, ami a Monte-Carlo eljárások jellegzetes *pont zaj*át (*dot-noise*) okozza (8.19. ábra).

8.7. Az árnyalási egyenlet megoldása véletlen gyűjtősétákkal

A matematikai bevezető után térjünk vissza a globális illuminációs feladat megoldásához! A 8.24. egyenlet szerint egy pixelbe írandó érték kiszámításához a

$$P = \int_{A_p} L(h(\vec{p}, -\vec{\omega}_{\vec{p}}), \vec{\omega}_{\vec{p}}) \cdot \frac{1}{A_p}\, dp$$

integrált kell kiértékelnünk. Mivel az L sugársűrűség maga is többdimenziós integrál, megállapíthatjuk, hogy egy sokdimenziós integrállal állunk szemben, amit Monte-Carlo (vagy kvázi Monte-Carlo) eljárással célszerű kiszámítani. A Monte-Carlo eljáráshoz véletlen $\vec{p}$ pontokat állítunk elő a pixel felületén, majd a véletlen ponton keresztül, az $-\vec{\omega}_{\vec{p}}$ irányba látható $\vec{x}_1 = h(\vec{p}, -\vec{\omega}_{\vec{p}})$ felületi pont sugársűrűségét a pont mintavételezési valószínűségével osztjuk. Ilyen hányadosok átlaga az integrált a mintaszámmal növekvő pontossággal becsli. Mivel nincs semmiféle indokunk arra, hogy a pixel különböző részeit eltérő gyakran mintavételezzük, a pixel pontjait egyenletes valószínűség-sűrűség szerint állítjuk elő. A pixel területe A_p, így az egyenletes eloszlás sűrűségfüggvénye $1/A_p$. Ha M mintát használunk, az integrálbecslés alakja:

$$P = \int_{A_p} L(\vec{x}_1, \vec{\omega}_{\vec{p}}) \cdot \frac{1}{A_p}\, dp \approx \frac{1}{M} \cdot \sum_{i=1}^{M} \frac{L(\vec{x}_1^{(i)}, \vec{\omega}_{\vec{p}}^{(i)})}{A_p} \cdot \frac{1}{1/A_p} = \frac{1}{M} \cdot \sum L(\vec{x}_1^{(i)}, \vec{\omega}_{\vec{p}}^{(i)}).$$

Az összegben feltűnik a látható $\vec{x}_1$ pont szemirányú sugársűrűsége, amelyet az

$$L(\vec{x}_1, \vec{\omega}_{\vec{p}}) = L^e(\vec{x}_1, \vec{\omega}_{\vec{p}}) + \int_{\Omega} L(h(\vec{x}_1, -\vec{\omega}'_1), \vec{\omega}'_1) \cdot f_r(\vec{\omega}'_1, \vec{x}_1, \vec{\omega}_{\vec{p}}) \cdot \cos\theta'_1\, d\omega'_1 \qquad (8.26)$$

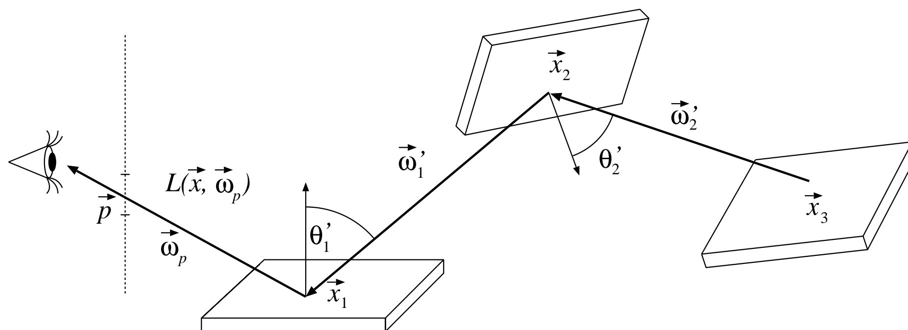
árnyalási egyenlet megoldásával kaphatunk meg (a pontot és a bejövő irányt egy 1-es indexszel láttuk el, hogy megkülönböztessük a későbbiekben előbukkanó újabb pontoktól és irányoktól).

Az egyenlet jobb oldalon szereplő integrált egy újabb Monte-Carlo eljárással becsüljük, azaz olyan véletlen mintákat használunk, ahol az $\vec{\omega}'_1$ integrálási változót egy alkalmas, $\vec{x}_1$-től is függő $p_{\vec{x}_1}(\vec{\omega}'_1)$ valószínűség-sűrűséggel mintavételezzük és a

$$\frac{L(h(\vec{x}_1, -\vec{\omega}'_1), \vec{\omega}'_1) \cdot f_r(\vec{\omega}'_1, \vec{x}_1, \vec{\omega}_{\vec{p}}) \cdot \cos\theta'_1}{p_{\vec{x}_1}(\vec{\omega}'_1)}$$

hányadost tekintjük az integrál egy véletlen becslésének. Ha több ilyen véletlen becslő átlagát képezzük, az átlag a valódi értékhez konvergál.

A véletlen becslő számításához azonosítanunk kell az $\vec{x}_2 = h(\vec{x}_1, -\vec{\omega}'_1)$ pontot, amely az $\vec{x}_1$ pontból a $-\vec{\omega}'_1$ irányban látszik és meg kell határoznunk ebben a pontban az $\vec{\omega}'_1$



8.20. ábra. *Gyűjtőséta*

irányú sugársűrűséget. Ezt a sugársűrűség értéket az árnyalási egyenletnek erre a pontra történő ismételt alkalmazásával kaphatjuk meg:

$$L(\vec{x}_2, \vec{\omega}_1') = L^e(\vec{x}_2, \vec{\omega}_1') + \int_{\Omega} L(h(\vec{x}_2, -\vec{\omega}_2'), \vec{\omega}_2') \cdot f_r(\vec{\omega}_2', \vec{x}_2, \vec{\omega}_1') \cdot \cos\theta_2' \, d\omega_2'.$$

Ez bizony egy újabb integrál, de nem esünk kétségbe, hanem ezt is a Monte-Carlo eljárással támadjuk meg, azaz egy $p_{\vec{x}_2}(\vec{\omega}_2')$ sűrűség szerint véletlen $\vec{\omega}_2'$ irányt állítunk elő, és az integrált a

$$\frac{L(h(\vec{x}_2, -\vec{\omega}_2'), \vec{\omega}_2') \cdot f_r(\vec{\omega}_2', \vec{x}_2, \vec{\omega}_1') \cdot \cos\theta_2'}{p_{\vec{x}_2}(\vec{\omega}_2')}$$

hányadossal becsüljük. Ezt a becslést a látható $\vec{x}_1$ pont sugársűrűségének 8.26. képletébe behelyettesítve:

$$\begin{aligned} L(\vec{x}_1, \vec{\omega}_{\vec{p}}) \approx\ & L^e(\vec{x}_1, \vec{\omega}_{\vec{p}}) + \\ + & \frac{f_r(\vec{\omega}_1', \vec{x}_1, \vec{\omega}_{\vec{p}}) \cdot \cos\theta_1'}{p_{\vec{x}_1}(\vec{\omega}_1')} \cdot L^e(\vec{x}_2, \vec{\omega}_1') + \\ + & \frac{f_r(\vec{\omega}_1', \vec{x}_1, \vec{\omega}_{\vec{p}}) \cdot \cos\theta_1'}{p_{\vec{x}_1}(\vec{\omega}_1')} \cdot \frac{f_r(\vec{\omega}_2', \vec{x}_2, \vec{\omega}_1') \cdot \cos\theta_2'}{p_{\vec{x}_2}(\vec{\omega}_2')} \cdot L(h(\vec{x}_2, -\vec{\omega}_2'), \vec{\omega}_2'). \end{aligned}$$

Sajnos még ez a véletlen becslő is tartalmazza az ismeretlen sugársűrűség-függvényt, de most már csak a kétszeres visszaverődést leíró tagban. Az ismeretlen függvényt innen úgy küszöbölhetjük ki, mint ahogy azt az egyszeres visszaverődésnél tettük, azaz az árnyalási egyenletet újból felírjuk, és az árnyalási egyenlet integrálját egyetlen véletlen becsléssel közelítjük. Mivel a visszaverődések csökkentik az energiát, az egyre hátrébb kerülő sugársűrűség-függvény egyre kisebb mértékben befolyásolja a becsült értéket.

Ha a rekurzív helyettesítést végtelen sokszor végezzük el, akkor az ismeretlen függvényt tökéletesen kiküszöbölhetjük a becslésből, amely ekkor egy végtelen sor lesz:

$$\begin{aligned} L(\vec{x}_1,\vec{\omega}_{\vec{p}}) &\approx L^e(\vec{x}_1,\vec{\omega}_{\vec{p}})+ \\ &+ \frac{f_r(\vec{\omega}'_1,\vec{x}_1,\vec{\omega}_{\vec{p}})\cdot\cos\theta'_1}{p_{\vec{x}_1}(\vec{\omega}'_1)}\cdot L^e(\vec{x}_2,\vec{\omega}'_1)+ \\ &+ \frac{f_r(\vec{\omega}'_1,\vec{x}_1,\vec{\omega}_{\vec{p}})\cdot\cos\theta'_1}{p_{\vec{x}_1}(\vec{\omega}'_1)}\cdot\frac{f_r(\vec{\omega}'_2,\vec{x}_2,\vec{\omega}'_1)\cdot\cos\theta'_2}{p_{\vec{x}_2}(\vec{\omega}'_2)}\cdot L^e(\vec{x}_3,\vec{\omega}'_2)+\ldots. \end{aligned}$$

Az eljárás tehát a szemből indul az adott pixelen keresztül. A látható $\vec{x}_1$ pontban véletlen $\vec{\omega}'_1$ irányban megkeresi a látható $\vec{x}_2$ pontot, ahonnan egy $\vec{\omega}'_2$ véletlen irányba lép tovább, majd ezt a műveletet ismételgeti. A meglátogatott pontok emisszióját, a korábbi pontok BRDF tényezőivel és az irányok és normálvektorok közötti szögek koszinuszával szorozzuk, valamint a részút valószínűség-sűrűségével osztjuk. A véletlen csatangolás miatt nevezzük ezt a módszert *véletlen bolyongás*nak (*random walk*). Akkor fejezhetjük be a bolyongást, ha egy irányban nem látunk semmit, illetve akkor is, ha az eltalált felület képtelen a fény visszaverésére. Egyéb esetekben sem kell a módszerünket végtelen bolyongásra kárhoztatni, hiszen a sokszori visszaverődéseknek már elhanyagolható a hatásuk, ezért mondhatjuk azt, hogy például 10 visszaverődés után már nem követjük a fény útját. A folyamat ilyen önkényes leállítása csak egy kis hibát okoz.

Egy véletlen fényút hozama a Monte-Carlo integrálás egyetlen mintája lesz. Ilyen mintákból sokat kell előállítanunk, hogy az átlagos hozam a valódi sugársűrűséget kicsiny hibával becsülje.

8.8. Az árnyalási egyenlet megoldása véletlen lövősétákkal

A globális illuminációs feladat megoldásához nem csak szemből induló, véletlen fényutakat használhatunk, hanem a fényutak a fényforrásokban is eredhetnek, és a teret a valódi fénnyel megegyező irányban járják be. Az ilyen fényutakat *lövősétáknak* nevezzük. A fényforrások közvetlen (azaz visszaverődéseket, töréseket nem tartalmazó) hatását egyetlen pixelre a 8.19. egyenlet szerint a

$$P_0 = \int_{\Omega}\int_{S} L^e(\vec{y},\vec{\omega})\cos\theta\cdot W^e(\vec{y},\vec{\omega})\;dy\,d\omega$$

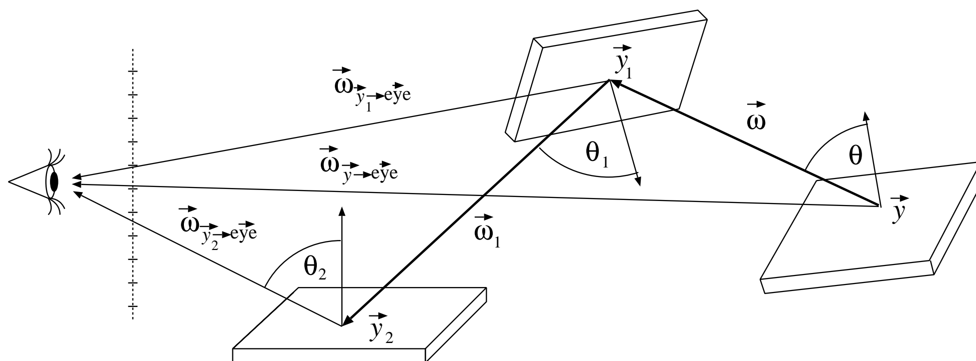
integrállal fejezhetjük ki. A mért érték 0 indexe arra utal, hogy csak a közvetlen hatást (azaz a 0-adik visszaverődést) vettük figyelembe. A $W^e(\vec{y},\vec{\omega})$ érzékenységfüggvény csak azon pontokra és irányokra különböző zérustól, amely pontok az adott pixelben látszanak, és amely irányok a pontból a szem felé mutatnak. Kicsiny (pontszerű) pupilla

esetén, minden ponthoz csak egyetlen irány tartozik, így a fenti integrálból a 8.22. egyenlet szerint az irány szerinti integrálás kiküszöbölhető:

$$P_0 = \int_{S_p} L(\vec{y}, \vec{\omega}_{\vec{y}\to\vec{\text{eye}}}) \cdot \frac{\cos\theta_{\vec{y}\to\vec{\text{eye}}}}{\Omega_p \cdot |\vec{y} - \vec{\text{eye}}|^2} \, dy,$$

ahol S_p a pixelben lálható felületi pontok halmaza. Az integrált alkalmassá tehetjük az összes pixel közvetlen hozzájárulásának számítására, ha az integrálási tartománynak a fényforrások S_e felületét tekintjük és az integrandust minden j pixelre megszorozzuk egy $v_j(\vec{y})$ láthatósági függvénnyel, amely 1 értékű, ha az $\vec{y}$ pont látszik a j pixelben és egyébként zérus:

$$P_0[j] = \int_{S_e} L^e(\vec{y}, \vec{\omega}_{\vec{y}\to\vec{\text{eye}}}) \cdot \frac{\cos\theta_{\vec{y}\to\vec{\text{eye}}}}{\Omega_p \cdot |\vec{y} - \vec{\text{eye}}|^2} \cdot v_j(\vec{y}) \, dy.$$



8.21. ábra. *Lövőséta*

Ezt az integrált Monte-Carlo eljárással becsüljük, azaz egy $p^e(\vec{y})$ sűrűségfüggvény szerint M pontot veszünk fel a fényforrás felületén, és az egyes pixelek integráljait a mintapontok hatásainak átlagával közelítjük. Az átlagban szereplő egyetlen tag, amely az $\vec{y}$ pontban található minta hatását írja le:

$$P_0[j] \approx \frac{L^e(\vec{y}, \vec{\omega}_{\vec{y}\to\vec{\text{eye}}})}{p^e(\vec{y})} \cdot \frac{\cos\theta_{\vec{y}\to\vec{\text{eye}}}}{\Omega_p \cdot |\vec{y} - \vec{\text{eye}}|^2} \cdot v_j(\vec{y}).$$

Most térjünk rá az egyszeres visszaverődések hatásának elemzésére! A közvetlen hozzájárulás integráljában az $L^e(\vec{y}, \vec{\omega}) \cos\theta dy \, d\omega = d\Phi(\vec{y}, \vec{\omega})$ a fényforrást az $\vec{y}$ pontban, $\vec{\omega}$ irányban elhagyó fénynyaláb teljesítménye. Ahhoz, hogy az egyszeres és többszörös visszaverődések hatását is kiszámíthassuk, az egyes fénynyalábok egyszeres visszaverődését és többszörös visszaverődéseit kell számba venni, és a hatásukat összegezni,

azaz integrálni. Ehhez az integrálhoz, lévén, hogy sokdimenziós, Monte-Carlo eljárást fogunk alkalmazni. A fénynyalábok közül válasszunk ki egyet véletlenszerűen $p^e(\vec{y}, \vec{\omega})dyd\omega$ valószínűséggel! Ezzel a lépéssel a közvetlen hozzájárulás számításához kijelölt $\vec{y}$ pont mellé még egy $\vec{\omega}$ irányt is mintavételezünk. Az irány és a felületi normális közötti szöget θ-val jelöljük. A Monte-Carlo módszer szabályai szerint a nyaláb teljesítményét elosztjuk a kiválasztás valószínűségével:

$$d\Phi = \frac{L^e(\vec{y}, \vec{\omega}) \cos\theta}{p^e(\vec{y}, \vec{\omega})}.$$

A fénynyaláb az $\vec{y}_1 = h(\vec{y}, \vec{\omega})$ pontot találja el, ahol az $f_r \cos\theta_1$ tényezővel súlyozva verődhet vissza. Az egyszeres visszaverődés egy véletlen becslése $\vec{\omega}_1$ irányban:

$$\frac{L^e(\vec{y}, \vec{\omega}) \cos\theta}{p^e(\vec{y}, \vec{\omega})} \cdot f_r(\vec{\omega}, \vec{y}_1, \vec{\omega}_1) \cdot \cos\theta_1,$$

ahol θ_1 az $\vec{\omega}_1$ irány és az $\vec{y}_1$ pontbeli felületi normális közötti szög. Ez a visszaverődés akkor lehet hatással az adott pixelre, ha az $\vec{y}_1$ pont a pixelen keresztül látszik, és az $\vec{\omega}_1$ irány az $\vec{y}_1$ pontból a szem felé mutat, azaz $\vec{\omega}_1 = \vec{\omega}_{\vec{y}_1 \to \vec{\mathrm{eye}}}$ és $\theta_1 = \theta_{\vec{y}_1 \to \vec{\mathrm{eye}}}$. A hozzájárulás nagyságát a direkt megvilágításhoz hasonlóan kaphatjuk meg:

$$P_1[j] \approx \frac{L^e(\vec{y}, \vec{\omega}) \cos\theta}{p^e(\vec{y}, \vec{\omega})} \cdot f_r(\vec{\omega}, \vec{y}_1, \vec{\omega}_{\vec{y}_1 \to \vec{\mathrm{eye}}}) \cdot \cos\theta_{\vec{y}_1 \to \vec{\mathrm{eye}}} \cdot \frac{v_j(\vec{y}_1)}{\Omega_p \cdot |\vec{y}_1 - \vec{\mathrm{eye}}|^2}.$$

A kétszeres visszaverődés hozzájárulásának számításához a fényt újból vissza kell vernünk, ezért az egyszeres visszaverődés $\vec{y}_1$ pontjában egy újabb $\vec{\omega}_1$ irányt válasszunk $p_{\vec{y}_1}(\vec{\omega}_1)$ valószínűség-sűrűség szerint és a fénynyaláb ilyen irányú visszaverődését kilőjük. A fénynyaláb által eltalált $\vec{y}_2$ felületi pontból újból számítjuk a képhozzájárulást. Tehát a kétszeres visszaverődés véletlen becslése:

$$P_2[j] \approx \frac{L^e(\vec{y}, \vec{\omega}) \cos\theta}{p^e(\vec{y}, \vec{\omega})} \cdot \frac{f_r(\vec{\omega}, \vec{y}_1, \vec{\omega}_1) \cdot \cos\theta_1}{p_{\vec{y}_1}(\vec{\omega}_1)} \cdot f_r(\vec{\omega}_1, \vec{y}_2, \vec{\omega}_{\vec{y}_2 \to \vec{\mathrm{eye}}}) \cdot \cos\theta_{\vec{y}_2 \to \vec{\mathrm{eye}}} \cdot \frac{v_j(\vec{y}_2)}{\Omega_p \cdot |\vec{y}_2 - \vec{\mathrm{eye}}|^2}.$$

A teljes megoldáshoz ezt a műveletet sokszor (elvileg végtelen sokszor) meg kell ismételni. A Monte-Carlo módszer a j-edik pixel eredményét a véletlen becslők átlagával állítja elő:

$$P[j] \approx \frac{1}{M} \cdot \sum_{i=1}^{M} \left(P_0^{(i)}[j] + P_1^{(i)}[j] + P_2^{(i)}[j] + \ldots \right).$$

8.9. Fontosság szerinti mintavételezés a véletlen bolyongásnál

A véletlen irányokat célszerű olyan valószínűség-eloszlásból mintavételezni, ami arányos az integrandussal, azaz a bejövő sugársűrűség és a visszaverődési-sűrűségfüggvény

szorzatával. Ez az árnyalási egyenlet

$$\int_{\Omega} L(h(\vec{x},-\vec{\omega}'),\vec{\omega}') \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \, d\omega'$$

integráljára azt jelenti, hogy az irányokat mintavételező $p(\vec{\omega}')$ sűrűségfüggvénynek az $f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \cdot L(h(\vec{x},-\vec{\omega}'),\vec{\omega}')$ szorzattal kell arányosnak lennie. A mintavételezés során általában nem közvetlenül az irányt állítjuk elő, hanem az irányt meghatározó polárszögeket, azaz $p(\vec{\omega})$ helyett a polárszögekre vonatkoztatott $p(\alpha,\beta)$ sűrűségfüggvényt keressük. Jelen pillanatban azért jelöljük a hosszúsági körök menti polárszöget β-val, a szélességi körök menti polárszögeket pedig α-val, és nem pedig a megszokott ϕ,θ jelöléseket alkalmazzuk, mert a továbbiakban előfordulhat, hogy a mintavételezés nem a felületi normálvektor irányába, hanem máshová képzeli el az iránygömb északi pólusát. Írjuk át az árnyalási egyenlet integrálját az új polárszögek mentén végrehajtott integrálra a differenciális térszögre vonatkozó $d\omega' = \sin\beta d\alpha d\beta$ összefüggés (8.1. egyenlet) felhasználásával:

$$\int_{\alpha=0}^{2\pi} \int_{\beta=0}^{\pi} L(h(\vec{x},-\alpha,-\beta),\alpha,\beta) \cdot f_r(\alpha,\beta,\vec{x},\vec{\omega}) \cdot \cos\theta' \cdot \sin\beta \, d\alpha d\beta.$$

Azt a $p(\alpha,\beta)$ sűrűségfüggvényt keressük, amelyik arányos az alábbi függvénnyel:

$$L(h(\vec{x},-\alpha,-\beta),\alpha,\beta) \cdot f_r(\alpha,\beta,\vec{x},\vec{\omega}) \cdot \cos\theta' \cdot \sin\beta.$$

A bejövő sugársűrűséget nem ismerjük (éppen azért számolunk, hogy ezt meghatározzuk), ezért közelítésekkel kell élnünk. A *BRDF mintavételezés* a fontosság szerinti mintavételt csak a visszaverődési-valószínűségsűrűség szerint, azaz koszinuszos taggal súlyozott BRDF fontos irányai szerint végzi el. A másik, *fényforrás mintavételezés* nevű eljárás pedig arra a felismerésre épít, hogy a bejövő sugársűrűség az adott irányban látható pont saját emissziójának és visszaverődésének az összege, ezért érdemes azokat az irányokat előnyben részesíteni, amelyekben a fényforrások találhatók.

8.9.1. BRDF mintavételezés

A BRDF alapú fontosság szerinti mintavételezés azt jelenti, hogy a választott irány p valószínűség-sűrűségfüggvénye arányos a BRDF és az *orientációs szög* (azaz a felületi normális és az adott irány közötti szög) koszinuszának szorzatával, vagyis $p(\alpha,\beta)$ arányos a $f_r(\alpha,\beta,\vec{x},\vec{\omega}) \cdot \cos\theta' \cdot \sin\beta$ függvénnyel. Az arányosság skálatényezőjét abból a feltételből kapjuk meg, hogy a p valószínűség-sűrűséget jelent (a Monte-Carlo módszerek

valószínűség-sűrűségét), ezért az integrálja egységnyi, a skálatényező tehát a célfüggvény integrálja:

$$\int_{\alpha=0}^{2\pi}\int_{\beta=0}^{\pi} f_r(\alpha,\beta,\vec{x},\phi,\theta)\cdot\cos\theta'\cdot\sin\beta\, d\alpha d\beta = \int_{\Omega_H} f_r(\vec{\omega}',\vec{x},\vec{\omega})\cdot\cos\theta',\, d\omega' = a(\vec{x},\vec{\omega}),$$

ahol $a(\vec{x},\vec{\omega})$ a felület $\vec{x}$ pontjának az *albedó*ja (8.12. egyenlet).

BRDF mintavételezés diffúz anyagokra

A nem átlátszó diffúz anyagok konstans BRDF-fel rendelkeznek a külső Ω_H féltérben, illetve zérus a visszaverődésük a belső irányokra. Tegyük fel, hogy az α,β gömbi koordinátákat egy olyan koordinátarendszerhez viszonyítjuk, amelynek északi pólusa éppen a normálvektor irányába mutat. Ekkor $\beta=\theta'$ a normálvektor és a választott irány közötti szög, így a felület „felett” a $\beta\le\pi/2$ irányok, a felület alatt, azaz a test belsejében pedig a $\beta>\pi/2$ irányok vannak. Az arányossági tényezőt, azaz a diffúz felület albedóját, a jobb oldal integrálásával kapjuk:

$$\int_{\alpha=0}^{2\pi}\int_{\beta=0}^{\pi/2} f_r\cdot\cos\beta\sin\beta\, d\beta d\alpha = f_r\cdot\pi.$$

A megfelelő valószínűség-sűrűségfüggvény:

$$p(\alpha,\beta)=\frac{f_r\cdot\cos\beta\sin\beta}{f_r\cdot\pi}=\frac{\cos\beta\sin\beta}{\pi}.$$

Tételezzük fel, hogy az α,β koordináta-minták előállításához használt valószínűségi változók függetlenek! Ekkor a valószínűség-sűrűség szorzat formájában írható fel:

$$p(\alpha,\beta)=\frac{1}{2\pi}\cdot[2\cos\beta\sin\beta],$$

ahol $1/(2\pi)$ az α, és $2\cos\beta\sin\beta=\sin2\beta$ pedig a β sűrűségfüggvénye.

A megfelelő szögek valószínűség-eloszlásfüggvényeit a sűrűségfüggvények integráljaként kapjuk:

$$P(\alpha)=\int_0^{\alpha}\frac{1}{2\pi}\,d\alpha=\frac{\alpha}{2\pi},\qquad P(\beta)=\int_0^{\beta}\sin2\beta\,d\beta=\sin^2\beta.$$

Egy tetszőleges $P(\xi)$ eloszlásfüggvényű ξ valószínűségi változó mintái egy egyenletes eloszlású *r valószínűségi változó* mintáinak a $\xi=P^{-1}(r)$ összefüggéssel leírt transzformációjával állíthatók elő. Következésképpen a keresett α és β valószínűségi változók a

$[0,1]$ intervallumon egyenletes eloszlású, u,v változók transzformációjával kereshetők meg:

$$\alpha = 2\pi \cdot u, \quad \beta = \theta' = \arcsin\sqrt{v}.$$

Egységintervallumba eső, egyenletes eloszlású véletlen mintákat a C-könyvtárban található `rand()` függvénnyel állíthatunk elő a következőképpen:

```
(float)rand()/RAND_MAX
```

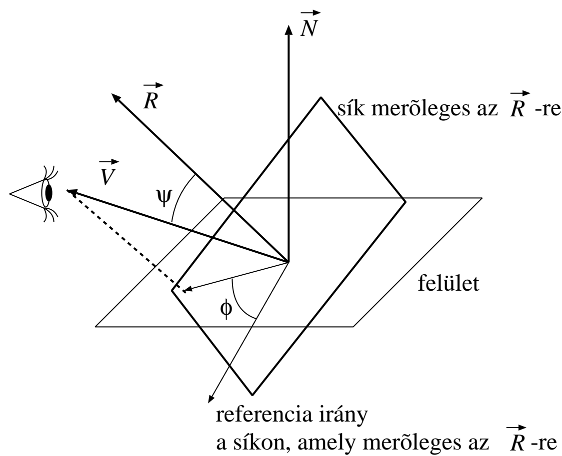
Kvázi Monte-Carlo eljárásoknál pedig ezek az értékek egy-egy alacsony diszkrepanciájú sorozat elemei.

BRDF mintavételezés Phong-modellel leírt spekuláris anyagokra

A spekuláris anyagok például a *Phong BRDF modell* reciprok változatával (8.16. egyenlet) jellemezhetők, amelynek alakja

$$f_r = k_s \cdot \cos^n \psi,$$

ahol ψ a nézeti iránynak, valamint a bejövő iránynak a felület normálisára vett tüköriránya közötti szög.



8.22. ábra. *Parametrizálás az albedó számolásához*

Az iránygömb megfelelő paraméterezéséhez az északi pólus az $\vec{R}$ tükörirány szerint választandó (8.22. ábra). Jelöljük $\beta = \psi$ szöggel az $\vec{\omega}_r$ iránytól való eltérést, és α-val pedig ennek az iránynak az $\vec{R}$-ra merőleges síkra vett vetülete és ezen sík szabadon választott vektora közötti szöget!

A BRDF mintavételezés olyan sűrűséget követel meg, amely a

$$k_s \cdot \cos^n \beta \cdot \cos\theta' \cdot \sin\beta$$

szorzattal arányos. Sajnos a $\cos\theta'$ tényező miatt nem tudjuk ezt a függvényt szimbolikusan integrálni, ezért olyan sűrűségfüggvényt fogunk alkalmazni, amely csak a $k_s \cdot \cos^n\beta\sin\beta$ kifejezéssel arányos. A valószínűség-sűrűséget a kifejezés normalizálásával kapjuk meg:

$$p(\alpha,\beta) = \frac{k_s \cdot \cos^n\beta\sin\beta}{\int\limits_{\alpha=0}^{2\pi}\int\limits_{\beta=0}^{\pi/2} k_s \cdot \cos^n\beta\sin\beta \, d\beta d\alpha} = \frac{n+1}{2\pi}\cos^n\beta\sin\beta.$$

Tételezzük ismét fel, hogy a koordináta-minták előállítására használt valószínűségi változók függetlenek, azaz a sűrűségfüggvényt szorzat alakban írhatjuk fel:

$$p(\alpha,\beta) = \frac{1}{2\pi} \cdot [(n+1)\cos^n\beta\sin\beta], \tag{8.27}$$

ahol $1/(2\pi)$ az α, és $(n+1)\cos^n\beta\sin\beta$ pedig a β valószínűség-sűrűségfüggvénye.

A megfelelő valószínűség-eloszlásfüggvények a következők:

$$P(\alpha) = \frac{\alpha}{2\pi}, \qquad P(\beta) = \int\limits_0^\beta (n+1)\cos^n\beta\sin\beta \, d\beta = 1 - \cos^{n+1}\beta.$$

A keresett α és β valószínűségi változók a $[0,1]$ intervallumon egyenletes eloszlású, u, v változók transzformációjával kereshetők meg:

$$\alpha = 2\pi \cdot u, \qquad \beta = \psi = \arccos(1-v)^{1/(n+1)}.$$

8.9.2. A fényforrás mintavételezése

A véletlen bolyongás BRDF mintavételezésén kívül még érdemes az egyes lépésekben a fényforrásokat is külön-külön mintavételezni [110]. Mivel ebben az esetben a mintákat az iránygömb helyett a fényforrásból választjuk, a $d\omega' = dy \cdot \cos\theta_{\vec{y}}/|\vec{x}-\vec{y}|^2$ összefüggés felhasználásával — ahol $\cos\theta_{\vec{y}}$ a fényforrás felületi normálisának és az $\vec{y} \to \vec{x}$ iránynak a szöge — a fényátadás operátort, mint a felületeken futó integrált írjuk fel:

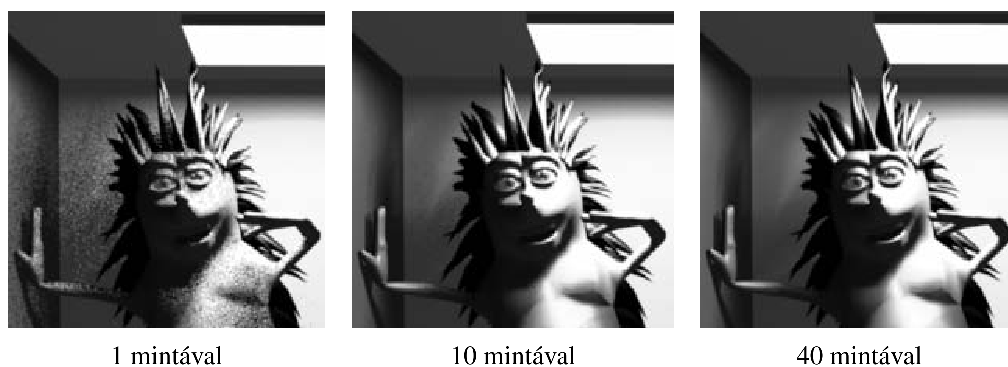
$$(\mathcal{T}_{f_r} L^e)(\vec{x},\vec{\omega}) = \int\limits_\Omega L^e(h(\vec{x},-\vec{\omega}'),\vec{\omega}') \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \, d\omega' =$$

$$\int\limits_{S_e} L^e(\vec{y},\vec{\omega}_{\vec{y}\to\vec{x}}) \cdot f_r(\vec{\omega}_{\vec{y}\to\vec{x}},\vec{x},\vec{\omega}) \cdot \frac{\cos\theta' \cdot \cos\theta_{\vec{y}}}{|\vec{x}-\vec{y}|^2} \cdot v(\vec{y},\vec{x}) \, dy, \tag{8.28}$$

ahol S_e a fényforrás területe, és $v(\vec{y},\vec{x}) = 1$, ha az $\vec{x}$ és $\vec{y}$ pontok láthatók egymásból, egyébként $v(\vec{y},\vec{x}) = 0$. A Monte-Carlo becslés kiszámításához N darab $\vec{y}_1,\ldots,\vec{y}_N$ pontot választunk egyenletes eloszlás szerint ($1/S_e$ sűrűséggel) a fényforráson, és a következő képletet használjuk:

$$(\mathcal{T}_{f_r}L^e)(\vec{x},\vec{\omega}) \approx \frac{S_e}{N} \cdot \sum_{i=1}^{N} L^e(\vec{y}_i,\vec{\omega}_{\vec{y}_i\to\vec{x}}) \cdot v(\vec{y}_i,\vec{x}) \cdot f_r(\vec{\omega}_{\vec{y}_i\to\vec{x}},\vec{x},\vec{\omega}) \cdot \frac{\cos\theta'_i \cdot \cos\theta_{\vec{y}_i}}{|\vec{x}-\vec{y}_i|^2},$$

ahol θ'_i az $\vec{x}$ pontban a felületi normális és a $\vec{x} \to \vec{y}_i$ irány közötti szög. Ha a térben csak egyetlen homogén fényforrás helyezkedik el, amely aránylag kicsi és messze van a vizsgált ponttól, akkor az integrandus megközelítőleg konstans a fényforrás felületén, ezért a szórás kicsi.



1 mintával 10 mintával 40 mintával

8.23. ábra. *Fényforrás mintavételezés*

Mivel a fényforrás mintavételezése a mintákat csak a direkt fényforráson állítja elő, ezért teljesen elhanyagolja az indirekt megvilágítást. Következésképpen a fényforrás mintavételezés önmagában nem használható a globális illuminációs algoritmusokban, csak mint kiegészítő eljárás, például a BRDF mintavételezéshez.

A BRDF és a fényforrás mintavételezést úgy kombinálhatjuk, hogy a bolyongás irányait BRDF mintavételezéssel állítjuk elő, de minden meglátogatott pontból árnyéksugarakat is küldünk a fényforrások felé. Az árnyéksugarak által szállított sugársűrűség visszaverődését kiszámítjuk és ezt tekintjük a felület saját emissziójának. Ez a módszer akkor nagyon hatékony, ha a térben pontszerű fényforrások vannak. Pontszerű fényforrások esetén az illumináció pontosan meghatározható.

8.9.3. Orosz rulett

Az árnyalási egyenlet megoldását mint egy végtelen sort írtuk fel, és módszert adtunk arra, hogy az egymást követő integrálokat hogyan lehet véletlen fényutakkal becsülni. Arról azonban mélyen hallgattunk, hogy ezt végtelen sok integrálra semmiképpen sem tudjuk megtenni, hiszen ahhoz végtelen sok időre lenne szükségünk. Abban legalább biztosak lehetünk, hogy ha az anyagok albedója egynél kisebb, akkor az egyre nagyobb számú visszaverődéseket leíró tagok egy geometriai sor szerinti sebességgel egyre kisebbek lesznek, így a sor konvergens. Az egyre kisebb tagok számításától általában eltekintünk, azaz a sort csonkoljuk.

Például mondhatjuk azt, hogy csak adott számú visszaverődésig vagyunk hajlandók szimulálni a fény útját. Ez az elhanyagolás nyilván torzítja a becslésünket. Ha a számítás célja „csak” egy jó kép előállítása, akkor nem kell tökéletes pontosságra törekednünk, így ez is elfogadható. Az *Arnold* nevű program erőteljesen él is ezzel a lehetőséggel (8.9., 8.10. és 8.8. ábra). Ha viszont fizikai pontosságra törekszünk, akkor nem hanyagolhatjuk el a magasabb rendű visszaverődéseket. Szerencsére egy ügyes trükkel kiküszöbölhetjük ezt a hibát, anélkül, hogy a végtelen sok integrált ténylegesen kiszámítanánk. A jól ismert pisztolyos „játék” alapján *orosz rulett*nek nevezett módszer a csonkítás determinisztikus hibáját egy zérus várható értékű zajjal cseréli fel. Mivel a Monte-Carlo eljárás úgyis véletlen becslők átlagaként állítja elő a végeredményt, határértékben ez a zaj is eltűnik.

A bolyongás i-edik lépése után az orosz rulett véletlenszerűen dönt, hogy folytassa-e a bolyongást vagy sem. Dobjunk fel egy kétforintost, amely s_i valószínűséggel esik arra az oldalra, hogy folytassuk a bolyongást és $1-s_i$ valószínűséggel arra, hogy fejezzük be! Ha nem folytatjuk a bolyongást, akkor az $n > i$ visszaverődésekből származó energia zérus. Ha viszont folytatjuk a bolyongást, akkor a véletlenszerűen elhanyagolt energia kompenzálása érdekében megszorozzuk a számított L^{in} bejövő sugársűrűség értéket $1/s_i$-vel. Az orosz rulett becslése tehát:

$$\tilde{L}^{in} = \begin{cases} L^{in}/s_i, & \text{ha folytatjuk a bolyongást,} \\ 0, & \text{egyébként.} \end{cases}$$

Várható értékben a becslés helyes lesz:

$$E[\tilde{L}^{in}] = s_i \cdot \frac{L^{in}}{s_i} + (1-s_i) \cdot 0 = L^{in}. \tag{8.29}$$

Az orosz rulett növeli a becslés szórását. A s_i valószínűséget általában úgy választjuk meg, hogy az *albedó*val (8.12. egyenlet) azonos legyen, ugyanis ekkor minden sugár nagyjából hasonló sugársűrűséget továbbít.

8.9.4. BRDF mintavételezés összetett anyagmodellekre

A gyakorlati anyagmodellek több BRDF-modell összegéből állnak (leggyakrabban diffúz + spekuláris). Az idáig ismertetett eljárások az egyes, elemi BRDF-modellek szerint képesek mintavételezni. Az összetett anyagmodelleket az orosz rulett általánosításával kezelhetjük. Bontsuk a visszaverődési-sűrűségfüggvényt az elemi sűrűségfüggvények összegére:

$$w = w_1 + w_2 + \ldots + w_n.$$

Az egy lépésben visszavert sugársűrűség ekkor szintén felbontható:

$$L = \int_{\Omega} w \cdot L^{\text{in}}\, d\omega = \int_{\Omega} w_1 \cdot L^{\text{in}}\, d\omega + \ldots + \int_{\Omega} w_n \cdot L^{\text{in}}\, d\omega.$$

Ez egy összeg, amit Monte-Carlo eljárással becsülhetünk. Az összeg i-edik tagját p_i valószínűséggel választjuk ki, és a Monte-Carlo becslés szabályai szerint a tagot p_i-vel osztjuk. A maradék $p_0 = 1 - p_1 - \ldots - p_n$ valószínűséggel pedig fejezzük be a bolyongást, és tételezzük fel, hogy a hozzájárulás zérus! Az így kapott L^{ref} véletlen becslő várható értéke a tényleges visszavert sugársűrűséggel egyezik meg:

$$E[L^{\text{ref}}] = p_1 \cdot E\left[\frac{w_1 \cdot L^{\text{in}}}{p_1}\right] + \ldots + p_n \cdot E\left[\frac{w_n \cdot L^{\text{in}}}{p_n}\right] + (1 - p_1 - \ldots - p_n) \cdot 0 =$$

$$E\left[(w_1^* + \ldots + w_n^*) L^{\text{in}}\right] = L. \tag{8.30}$$

A véletlen becslő szórása akkor lesz kicsiny, ha a fontosság szerinti mintavételezés szerint az egyes tagokat a nagyságukkal arányosan választjuk ki. Mivel a bejövő sugársűrűségről nem tudunk semmit, feltételezzük, hogy állandó. Ebben az esetben az egyes tagok az állandó sugársűrűség és az albedó szorzatával egyenlőek. Ennek megfelelően az elemi anyagmodellek albedójával arányosan kell a modellek közül választani.

A BRDF mintavételezést az alábbi programrészletben foglaljuk össze. Az eljárás a bejövő irány (`in`) és a felületi normális (`normal`) alapján egy véletlen kimeneti irányt (`out`) állít elő, és a minta valószínűség-sűrűségét (`prob`) is megadja:

```
BRDFSampling(in, normal, out) {
    prob = SelectBRDFModel(normal, in);
    if (prob == 0) return 0;
    prob *= Reflection(in, normal, out);
    if (prob == 0) return 0;
    return prob;
}
```

A `SelectBRDFModel()` véletlenszerűen választ az elemi BRDF-modellek közül az elemi albedók valószínűségeivel. A függvény zérus visszatérési értékkel jelzi, hogy a

bolyongást be kell fejezni az orosz rulettnek megfelelően. A `Reflection` előállítja az új „out” kimeneti irányt, a választott elemi BRDF fontos irányainak a hangsúlyozásával.

8.9.5. Fontosság szerinti mintavételezés színes terekben

Az eddigiekben feltételeztük, hogy a globális illuminációs feladatot egyetlen hullámhosszon oldjuk meg, azaz a BRDF-ek, az albedók és az emissziót tartalmazó súlyok valós változók, ezért a sűrűség arányos lehet velük. Jóllehet, ha színes képekre van szükségünk, akkor az árnyalási egyenletet néhány (legalább 3) különböző hullámhosszon kell megoldanunk. Ha a különböző hullámhosszokat teljesen függetlenül kezeljük, akkor a javasolt fontosság szerinti mintavételezés változatlanul használható. Ekkor a geometriai számításokat feleslegesen megismételnénk a különböző hullámhosszokra, ezért ez a módszer nem javasolható.

Jobban járunk, ha olyan sugarakat használunk, amelyek egyszerre minden hullámhosszon szállítják a fényt. Ebben az esetben az emisszió, az albedó és a BRDF vektor formájában írható fel, ezért az arányosság közvetlen értelmezése nem használható. Nyilván a fontosság szerinti mintavételezésnek ekkor azokat a tartományokat kell kiemelnie, ahol ezen vektormennyiségek elemei jelentősek. Egy $\mathcal{I}$ *fontosság függvény*re van szükségünk, amely nagy, ha a vektor elemei nagyok, és kicsi, ha az elemek kicsik. A fontosság függvény a spektrum függvénye, például reprezentatív hullámhosszok értékeinek az összege, vagy akár súlyozott összege. A súlyozáshoz felhasználhatjuk, hogy az emberi szem érzékenysége változik a hullámhossz függvényében. A spektrumnak a szem érzékenységi függvényével súlyozott átlaga a *luminancia*. Számítsuk ki tehát az emisszió, BRDF illetve az albedó luminanciáját, amelyek már skaláris mennyiségek, így a fontosság szerinti mintavételezésben közvetlenül használhatók!

8.10. Véletlen bolyongási algoritmusok

A véletlen bolyongási algoritmusok véletlen fényutakat állítanak elő. A fényutak a szemből illetve a fényforrásokból egyaránt indulhatnak. A szemből induló bolyongást gyűjtősétának, a fényforrásból indulót pedig lövősétának nevezzük.

A *gyűjtőséta* a meglátogatott pontok emisszióját gyűjti össze a séta kiindulási pontjában lévő pixel számára. A gyűjtőséták általános algoritmusa a következő:

```
for (minden p pixelre) {
    color = 0;
    for (i = 1; i ≤ M; i++) {                    // M a fényútminták száma
        ray = a szemből a p pixelen keresztülhaladó véletlen sugár;
        samplecolor = Trace(ray);
        color += samplecolor/M;
    }
```

```
        SetPixel(p, color);
}
```

A különböző algoritmusok a `Trace()` függvényt különbözőképpen implementálják. Ez a függvény a sugár által elkezdett fényútnak a szembe bevitt sugársűrűségét határozza meg.

A *lövőséta* a fényforrásból kilépő fénynyaláb energiáját szórja szét a térben. A lövőséták általános algoritmusa a következő:

```
Kép törlése;
for (i = 1; i ≤ M; i++) {                                    // M a fényútminták száma
        ray = egy fényforrás pont és irány mintavételezés p^e sűrűséggel;
        power = L^e · cos θ / p^e / M;
        Shoot(ray, power);
}
```

A különböző lövőséta algoritmusok a `Shoot()` függvényt különbözőképpen valósítják meg. Ez a függvény a teljes út által a szembe bevitt energiát határozza meg, valamint azt a pixelt, amelyen keresztül az út a szembe érkezik.

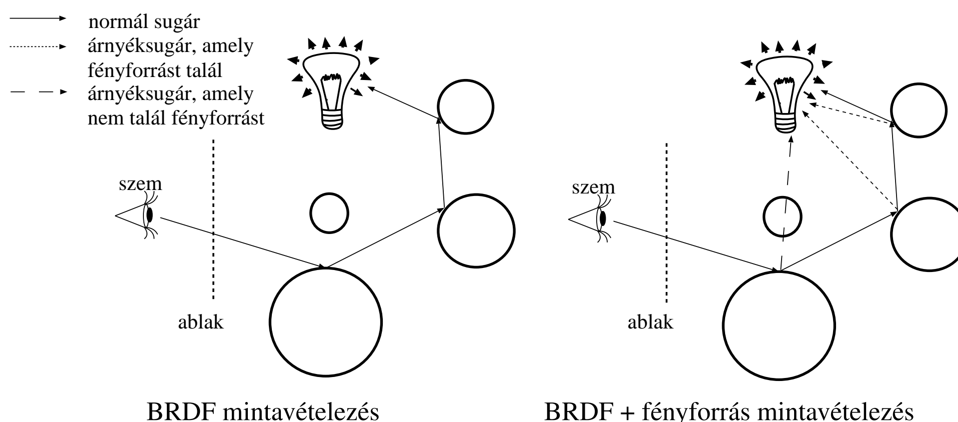
A fontosság szerinti mintavételezés értelmében a visszaverődéseknél az újabb irányokat olyan valószínűség-sűrűségek szerint érdemes mintavételezni, amelyek arányosak a visszaverődési-sűrűségfüggvénnyel. Másrészt az orosz rulett alkalmazásával a helyi albedónak megfelelő valószínűséggel érdemes leállítani a bolyongást.

Mivel annak a valószínűsége nagyon kicsiny, hogy egy BRDF mintavételezést alkalmazó gyűjtőséta a véletlen bolyongás során rátalál egy kis fényforrásra, ilyen esetekben a fényutak döntő részének a hozzájárulása zérus. Ezt elkerülendő, a BRDF mintavételezést érdemes fényforrás mintavételezéssel kombinálni, azaz minden meglátogatott pontban a fényforrások felé egy vagy több determinisztikusan vagy véletlenszerűen kiválasztott *árnyéksugarat* (*shadow ray*) indítunk és az onnan érkező megvilágítás visszaverődését továbbítjuk a szem felé. Hasonlóképpen — mivel a szem pontszerű — zérus annak a valószínűsége, hogy a fényforrásból indított BRDF mintavételezést használó lövőséta valaha eltalálja a szemet, ezért az ilyen fényutak hasznavehetetlenek. Ezen úgy segíthetünk, hogy a lövőséta meglátogatott pontjait összekötjük a szemmel, *láthatósági sugarak* segítségével.

A következőkben először egy gyűjtősétát alkalmazó véletlen bolyongási algoritmussal ismerkedünk meg, majd egy lövősétát végrehajtó módszert tárgyalunk. Végül olyan eljárásokat is bemutatunk, amelyek egyszerre alkalmaznak lövő- és gyűjtősétát.

8.10.1. Inverz fényútkövetés

A Kajiya által javasolt *inverz fényútkövetés* (*path tracing*) [68] véletlen gyűjtősétával dolgozik. A szempozícióból indulunk, akár a sugárkövetésnél, de most minden egyes metszéspontnál véletlenszerűen választjuk ki a továbbhaladási irányt, mégpedig olyan valószínűség-sűrűségfüggvény szerint, ami arányos a BRDF és a kilépő szög koszinuszának a szorzatával (BRDF mintavételezés). Minden lépés után az orosz rulett szabályai szerint, az albedónak megfelelő valószínűséggel folytatjuk a bolyongást.



8.24. ábra. *Inverz fényútkövetés*

Ideális esetben a fontosság szerinti mintavétel és az orosz rulett súlya együttesen kioltja a BRDF-eket és a koszinuszos tagokat. Így a bolyongás végén leolvasott emissziót semmilyen tényezővel sem kell szorozni, csupán az átlagolást kell elvégezni.

Az inverz fénykövetés a legegyszerűbben implementálható globális illuminációs eljárás [2], amelynek optimalizált változatait használják az *Arnold*[3] és a *Radiance* [5] programokban is. Az inverz fénykövetés `Trace()` függvényének pszeudo-kódja:

```
Trace(ray) {
    (object, x⃗) = FirstIntersect(ray);
    if (nincs metszéspont) return L_sky;
    color = L^e(x⃗, -ray.direction) + DirectLightsource(x⃗, -ray.direction);
    newray.start = x⃗;                                              // az új sugár
    prob = BRDFSampling(-ray.direction, normal, newray.direction);
    if (prob == 0) return color;                                   // orosz rulett
    color += Trace(newray) * w(newray.direction, normal, -ray.direction) / prob;
```

[2]az implementációs részletek megtalálhatók a [118, 116] könyvekben

[3]http://www.3dluvr.com/marcosss/

```
        return color;
    }
```

Ebben az algoritmusban az L_{sky} a háttér megvilágítás intenzitása (pl. égbolt), a `FirstIntersect()` függvény a sugár által elsőként metszett testet és a metszéspontot adja vissza. A `DirectLightsource()` függvény a fényforrások fényének egyszeri visszaverődését becsüli meg, és kiszámítja ennek hatását az $\vec{x}$ pontban adott irányban. Például, ha a tér l darab pontforrást tartalmaz az $\vec{y}_1,\ldots,\vec{y}_l$ helyeken és $\Phi_1,\ldots,\Phi_l$ teljesítményekkel, akkor ezek visszaverődése az $\vec{x}$ pontban:

$$L^{\text{ref}}(\vec{x},\vec{\omega}) = \sum_{i=1}^{l} \frac{\Phi_l}{4\pi|\vec{y}_i-\vec{x}|^2} \cdot v(\vec{y}_i,\vec{x}) \cdot f_r(\vec{\omega}_{\vec{y}_i\to\vec{x}},\vec{x},\vec{\omega}) \cdot \cos\theta_i', \tag{8.31}$$

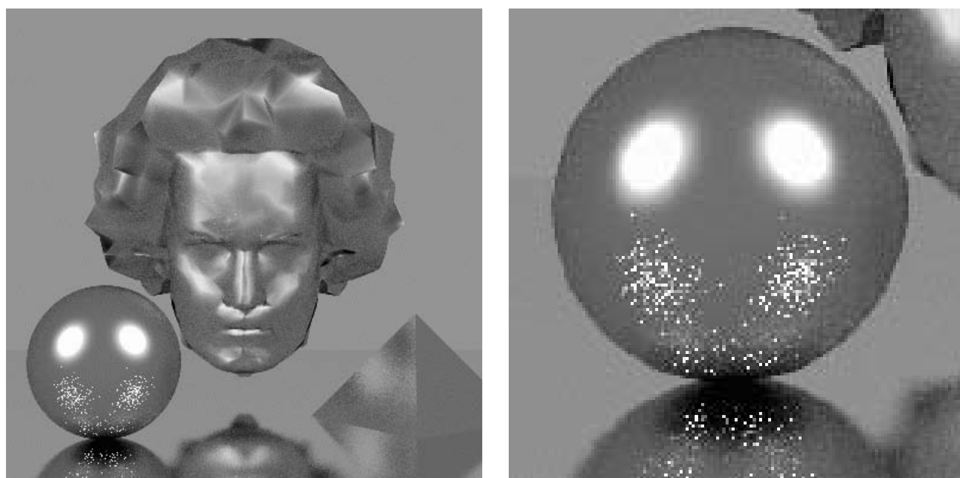
ahol θ_i' az $\vec{\omega}_{\vec{y}_i\to\vec{x}}$ és a felület normálisa közti szög, és az *árnyék sugarak*kal számított $v(\vec{y}_i,\vec{x})$ a két pont kölcsönös láthatóságát jelzi. A képletben a $4\pi|\vec{y}_i-\vec{x}|^2$ nevező az $\vec{y}_i$ középpontú $|\vec{y}_i-\vec{x}|$ sugarú gömb felszíne, amelyen szétoszlik a fényforrás energiája.

A felületi fényforrások $L^e(\vec{y},\vec{\omega})$ emissziójának kezelése céljából Monte-Carlo integrált használhatunk, amely N egyenletesen elosztott $\vec{y}_i$ mintát választ az S_e felszínű fényforrás felületén, és a következő becslést alkalmazza:

$$L^{\text{ref}}(\vec{x},\vec{\omega}) \approx \frac{S_e}{N} \cdot \sum_{i=1}^{N} L^e(\vec{y}_i,\vec{\omega}_{\vec{y}_i\to\vec{x}}) \cdot v(\vec{y}_i,\vec{x}) \cdot f_r(\vec{\omega}_{\vec{y}_i\to\vec{x}},\vec{x},\vec{\omega}) \cdot \frac{\cos\theta_i' \cdot \cos\theta_{\vec{y}_i}}{|\vec{x}-\vec{y}_i|^2}. \tag{8.32}$$

A programban a `BRDFSampling()` vagy új irányt talál, vagy nullával tér vissza, ha az orosz rulett miatt a bolyongást be kell fejezni. Az algoritmus, a sugárkövetéshez hasonlóan rekurzívan hívja saját magát a magasabb rendű visszaverődések számítása miatt. Amennyiben a véletlen minták helyett kvázi Monte-Carlo eljárást alkalmazunk, az újabb rekurziós szintekhez az alacsony diszkrepanciájú sorozat újabb koordinátáit használjuk fel.

Vegyük észre, hogy ez az algoritmus a fényút utolsó lépésén kívül BRDF mintavételezést használ, míg az utolsó lépés a fényforrás mintavételezéséből adódik! Ha az utolsó lépésben meglátogatott felület közel van az ideális tükörhöz vagy az ideális törő anyaghoz, akkor csak nagyon kevés irányra nem elhanyagolható a visszaverődés. A fontos visszaverődési, illetve törési irányokat a BRDF mintavételezés kiválasztaná ugyan, de az utolsó lépésben kénytelenek vagyunk ehelyett fényforrás mintavételezést használni, amely nagyon rossz fontosság szerinti mintavételezést eredményezhet. Mivel a fényforráshoz közeli ideális felületek okozzák a *kausztikus* optikai jelenségeket (mint például a lencse összegyűjti a fényt egy diffúz felületen), ezért az inverz fénykövetés — mint más gyűjtőséták — rosszak a kausztikus effektusok megjelenítésében (ilyen jelenségek közé tartozik, amikor egy lencse, vagy tükör fókuszálja a fényforrás fényét). Figyeljük meg, hogy



8.25. ábra. *Inverz fénykövetéssel számított kép és a kinagyított kausztikus folt*

a 8.25. ábrán, a rézgömb alján a kausztikus foltot a módszer csak nagy szórással tudta kiszámolni, pedig 800 fényútmintát (!) vettünk pixelenként!

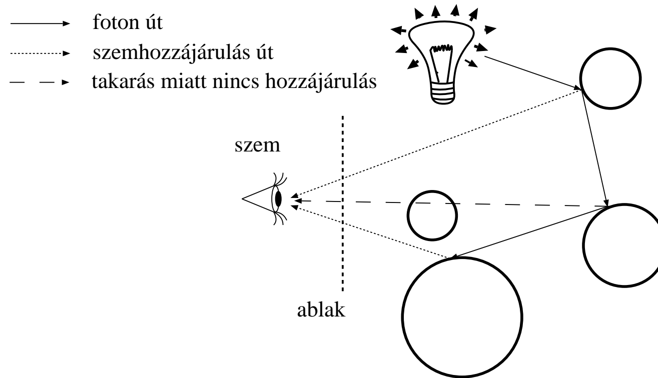
8.10.2. Fénykövetés

A *fénykövetés* (*light tracing*) [36] lövősétákat alkalmaz. Az egyes séták kezdőpontját és irányát véletlenszerűen választjuk ki a fényforrások pontjaiból és a sugárzási irányaiból. A fénysugár indítása után véletlenül verődik ide-oda a térben. Az irányokat a BRDF és a koszinuszos tag szorzatával arányos valószínűség-sűrűségfüggvényből mintavételezzük, a bolyongást minden lépés után az orosz rulett felhasználásával, az albedóval megegyező valószínűséggel folytatjuk. Minden visszaverődési pontot összekötünk a szempozícióval, és ellenőrizzük, hogy lehet-e ennek hatása valamely pixelre. Ha lehet, a pixel színéhez hozzáadjuk a visszaverődés hatását.

A fénykövetés `Shoot()` függvénye:

```
Shoot(ray, power) {
    (object, x⃗) = FirstIntersect(ray);
    if (nincs metszés) return;
    if (x⃗ látható a p pixelen keresztül)
        color[p] += power ·w(ray.direction, x⃗, eye direction ) / (Ω_p · |x⃗ − p⃗ |^2);
    newray.start = x⃗;                                          // az új sugár
    prob = BRDFSampling(-ray.direction, normal, newray.direction);
    if (prob == 0) return color;                              // orosz rulett
    newpower = power * w(-ray.direction, normal, newray.direction) / prob;
```

```
        Shoot(newray, newpower);
    }
```

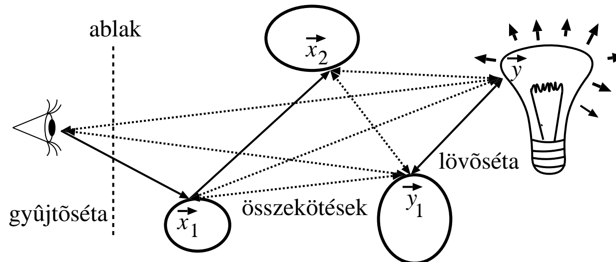


8.26. ábra. *Fénykövetés*

Ez az algoritmus az utolsó lépés kivételével szintén BRDF mintavételezést használ. Az utolsó sugár a meglátogatott pontot a szemmel köti össze, amelynek iránya messze eshet a BRDF által előnyben részesített irányoktól. Ez csökkenti a fontosság szerinti mintavételezés hatékonyságát, ha a látható felület nagyon spekuláris. A látható tükrök vagy törő felületek (üveg) tehát nehézségeket jelentenek.

8.10.3. Kétirányú fényútkövetés

A *kétirányú fényútkövetés* (*bi-directional path tracing*) [82, 131] az *inverz fényútkövetés* és a *fénykövetés* kombinációja. Ez a módszer egyszerre indít egy gyűjtősétát és egy lövősétát, majd a két séta végpontjait összeköti (8.27. ábra).



8.27. ábra. *Kétirányú fényutak az összes lehetséges összekötő sugárral*

A lövőséta a fényforrás teljesítményét juttatja el egy véletlen ponthoz. Tegyük fel, hogy a lövőséta egy $\vec{y}$ pontjába $\vec{\omega}_{in}$ irányból $d\Phi(\vec{y}, \vec{\omega}_{in})$ teljesítmény érkezik! A

gyűjtőséta a szemből indul és a meglátogatott pontok emisszióját, illetve fényforrás mintavételezésnél a fényforrásnak az adott pontra eső megvilágítását szállítja a szembe. Tekintsük a lövőséta egy $\vec{y}$ pontját kis fényforrásnak, ami a gyűjtőséta egy $\vec{x}$ pontjában fényvisszaverődést okoz! Jelöljük az $\vec{x}$ pontba érkezés irányát $\vec{\omega}_{out}$-tal! A fényvisszaverődés számításához összekötjük a gyűjtőséta pontját a „virtuális" fényforrással. A virtuális fényforrás teljesítményéből az $\vec{\omega}_{\vec{y}\to\vec{x}}$ összekötési irányban $d\omega_{\vec{x}}$ térszögbe az alábbi teljesítmény jut el:

$$d\Phi \cdot f_r(\vec{\omega}_{in}, \vec{y}, \vec{\omega}_{\vec{y}\to\vec{x}}) \cdot \cos\theta_{\vec{y}} \cdot d\omega_{\vec{x}},$$

hiszen egy belépő fotonra annak valószínűsége, hogy éppen a $d\omega_{\vec{x}}$ térszögben folytatja az útját $f_r(\vec{\omega}_{in}, \vec{y}, \vec{\omega}_{\vec{y}\to\vec{x}}) \cdot \cos\theta_{\vec{y}} \cdot d\omega_{\vec{x}}$.

A $d\omega_{\vec{x}}$ térszög és a dx terület közötti $d\omega_{\vec{x}} = dx \cdot \cos\theta_{\vec{x}}/|\vec{x}-\vec{y}|^2$ összefüggés szerint a dx felületelemre sugárzott teljesítmény:

$$d\Phi \cdot f_r(\vec{\omega}_{in}, \vec{y}, \vec{\omega}_{\vec{y}\to\vec{x}}) \cdot \cos\theta_{\vec{y}} \cdot \frac{dx\cos\theta_{\vec{x}}}{|\vec{x}-\vec{y}|^2}.$$

Ennek a teljesítménynek viszont az $\vec{\omega}_{out}$, gyűjtőséta utolsó iránya felé a

$$f_r(\vec{\omega}_{\vec{y}\to\vec{x}}, \vec{x}, \vec{\omega}_{out}) \cdot \cos\theta_{out} \cdot d\omega_{out}$$

része verődik vissza. Az $\vec{\omega}_{out}$ irányból látható sugársűrűség a teljesítmény osztva a térszöggel ($d\omega_{out}$) és a látható területtel ($dx \cdot \cos\theta_{out}$).

Összefoglalva, az átalakításhoz a lövőséta $\vec{y}$ pontjára ω_{in} irányból érkező $d\Phi$ teljesítmény a gyűjtőséta $\vec{x}$ pontjából az $\vec{\omega}_{out}$ irányba a következő sugársűrűséget eredményezi:

$$L(\vec{x}, \vec{\omega}_{out}) = d\Phi(\vec{y}, \vec{\omega}_{in}) \cdot f_r(\vec{\omega}_{in}, \vec{y}, \vec{\omega}_{\vec{y}\to\vec{x}}) \cdot \frac{\cos\theta_{\vec{y}} \cdot \cos\theta_{\vec{x}}}{|\vec{x}-\vec{y}|^2} \cdot f_r(\vec{\omega}_{\vec{y}\to\vec{x}}, \vec{x}, \vec{\omega}_{out}).$$

Ezek az összefüggések akkor igazak, ha az összekötött $\vec{x}, \vec{y}$ pontokat nem takarja el egymástól valamilyen tárgy, ellenkező esetben a hozzájárulás zérus. A láthatóságot sugárkövetéssel dönthetjük el.

A képlet alapján akkor fordulhat elő, hogy egy nagy hozzájárulást kis valószínűséggel mintavételezünk, ha a gyűjtő és a lövőséta végpontjain erősen spekuláris (erre az irányra nagy BRDF-fel rendelkező) felületet találtunk, vagy az összekötés hossza kicsiny. A fontosság szerinti mintavételezés elvei szerint ezek a minták rosszak, ezért jó lenne megszabadulni tőlük, lehetőleg úgy, hogy a várható érték továbbra is helyes legyen. Másrészt vegyük észre, hogy a kétirányú fénykövetésben egy n hosszú fényút többféle módon is előállhat! Például keletkezhet, mint egy $n-1$ hosszú lövőséta, amit a szemhez kötöttünk, mint egy $n-2$ hosszú lövőséta, amelyet egy 1 hosszú gyűjtősétával kötöttünk össze stb., sőt akár mint egy $n-1$ hosszú gyűjtőséta, amit a fényforrás mintavételezés árnyéksugara kapcsol a fényforráshoz. Ha ezt a jelenséget nem

vesszük figyelembe, akkor könnyen előfordulhat, hogy egyetlen fényutat többszörösen is beépítünk a képbe, ami viszont elrontja a végeredményt. Mind a két problémára a következő alfejezet ad megoldást.

Többszörös fontosság szerinti mintavételezés

Mint láttuk, a kétirányú fénykövetés egyetlen fényutat több különféle módon is előállíthat. Más oldalról nézve, a kétirányú fénykövetést úgy tekinthetjük, mint több mintavételi eljárás kombinációját. Természetesen ha ezek eredményét egyszerűen összeadnánk, akkor rossz eredményt kapnánk, amely a valódi érték többszöröse lenne. Akkor juthatunk helyes eredményhez, ha az egyes módszerek eredményét súlyozva adjuk össze, ahol a súlyok összege mindig egységnyi. Elvileg állandó súlyokat is használhatnánk, amelyek megegyeznek a használt módszerek számának reciprokával, de ennél sokkal jobb eredményt kapunk, ha a súlyokat a mintától függően állítjuk be, aszerint, hogy egy adott módszer milyen jó egy minta előállításához. A formális vizsgálathoz tegyük fel, hogy az $\int f(\mathbf{z})\, d\mathbf{z}$ integrál számításához n különböző mintavételi eljárást használhatunk, amelyek egy véletlen $\mathbf{z}$ mintát rendre $p_1(\mathbf{z}),...,p_n(\mathbf{z})$ valószínűség-sűrűséggel állítanak elő! Ezek a módszerek az

$$\frac{f(\mathbf{z})}{p_1(\mathbf{z})},\ \frac{f(\mathbf{z})}{p_2(\mathbf{z})},\ \ldots,\ \frac{f(\mathbf{z})}{p_n(\mathbf{z})}$$

becslőkkel közelítenék az integrált. A kombinált becslő az egyes módszerek becslőinek súlyozott összege:

$$\int f(\mathbf{z})\, d\mathbf{z} \approx \sum_{i=1}^{n} w_i(\mathbf{z}) \cdot \frac{f(\mathbf{z})}{p_i(\mathbf{z})}.$$

A kombinált becslő akkor lesz *torzítatlan* — azaz várható értékben akkor adja vissza a pontos értéket — ha minden $\mathbf{z}$ lehetséges mintára fennáll a $\sum_i w_i(\mathbf{z}) = 1$ egyenlőség, azaz a súlyok összege egységnyi. Ezen feltétel betartása mellett még meglehetősen szabadon megválaszthatjuk a súlyozást. Nyilván olyan súlyozást érdemes alkalmazni, ami minimalizálja a kombinált becslő szórását. Sajnos a minimalizálás pontos megvalósítása nem lehetséges, de kellően jó eredményeket érhetünk el heurisztikus meggondolásokkal is. Emlékezzünk vissza, hogy egy jó fontosság szerinti mintavételezési séma mindent elkövet annak érdekében, hogy nehogy egy nagy f integrandusú pontot kis valószínűség-sűrűséggel mintavételezzen! Eszerint, ha egy pontot két különböző módszer mintavételez, az egyik p_1, a másik pedig p_2 valószínűség-sűrűséggel, és p_1 nagyobb p_2-nél, akkor az első módszer jobb ezen pont mintavételezésére.

A következőkben heurisztikus sémákat ismertetünk, amelyek erre a felismerésre építenek. Az *egyensúly heurisztika* (*balance heuristic*) [130] a súlyokat a mintavételezési sűrűséggel arányosan állítja be, azaz

$$w_i(\mathbf{z}) = \frac{p_i(\mathbf{z})}{\sum_{k=1}^{n} p_k(\mathbf{z})}. \tag{8.33}$$

A súlyokat a kombinált becslő képletébe behelyettesítve megállapíthatjuk, hogy ez annak felel meg, mintha az átlagos sűrűséggel mintavételeznénk:

$$\hat{p}(\mathbf{z}) = \frac{1}{n} \cdot \sum_{i=1}^{n} p_i(\mathbf{z}). \tag{8.34}$$

A másik, a *maximum heurisztika* nevű séma sokkal szélsőségesebben választ a lehetséges stratégiák közül. A különböző módszerek mintái közül csak azét használja, amelyik ezt a mintát a legnagyobb sűrűséggel állítja elő:

$$w_i(\mathbf{z}) = \begin{cases} 1, & \text{ha } p_i(\mathbf{z}) \text{ a legnagyobb a } p_1(\mathbf{z}), \ldots, p_n(\mathbf{z}) \text{ közül,} \\ 0, & \text{egyébként.} \end{cases} \tag{8.35}$$

Mind az egyensúly, mind pedig a maximum heurisztika teljesíti a torzítatlanság $\sum_i w_i(\mathbf{z}) = 1$ feltételét.

A kétirányú fénykövetés mintáinak súlyozása

A kétirányú fénykövetés olyan fényútmintákat állít elő, amelyek k hosszú gyűjtő és $n-k$ hosszú lövő séta kombinációi, ahol $k = 0, \ldots, n$. Egy kombinált séta valószínűségsűrűsége a gyűjtő séta $p_g(k)$ sűrűségének és a lövőséta $p_s(n-k)$ sűrűségének szorzata, azaz

$$p_k = p_g(k) \cdot p_s(n-k).$$

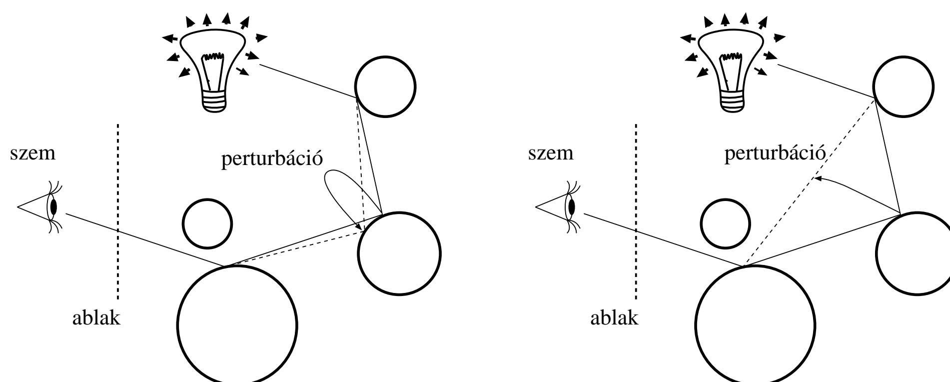
Ugyanezt az utat $(n+1)$-féleképpen állíthatjuk elő, ha a gyűjtőséta hosszával végigfutunk a megengedett $k = 0, \ldots, n$ tartományon. A különböző változatok előállítási valószínűsége persze más és más. Ezeket a valószínűségeket a 8.33. vagy a 8.35. egyenletbe helyettesítve a heurisztika súlyait meghatározhatjuk.

8.10.4. Metropolis-fénykövetés

A véletlen bolyongási algoritmusok a fényutakat függetlenül állítják elő. Ez egy sokprocesszoros rendszerben előnyös is lehet, hiszen akár minden utat külön processzorhoz rendelhetünk. Más esetekben azonban a független bolyongás nem kellően hatékony, ugyanis a fényút felépítésekor a nehezen megszerzett tudást minden út után elfelejti. Képzeljük el, hogy a fény csak egy kis lyukon juthat át az egyik szobából a másikba! A véletlen tapogatózásnak nagyon sok próbálkozásába kerül, hogy végre ráakadjon erre a kis lyukra. Ebbe ugyan bele kell törődni, azt azonban már nagyon nehezen fogadjuk el, hogy a lyukon átmenő fényút hatásának számítása után az egészet elfelejtjük, és a lyukat megint vakon keressük. Az ilyen nehéz megvilágítási problémák hatékony módszere a *Metropolis-fénykövetés* [132], amely a fényutakat nem függetlenül, hanem

az előző fényút kismértékű perturbációjával állítja elő. A bolyongási algoritmusokban alkalmazott, fontosság szerinti mintavételezés sohasem pontos, hiszen egy olyan mennyiségtől függ, amit éppen most számolunk ki. A különböző véletlen bolyongási algoritmusok úgy is tekinthetők, mint az optimális fontosság szerinti mintavételezés különböző heurisztikus közelítései. A többi eljárással szemben a Metropolis-módszer tökéletes fontosság szerinti mintavételezést ígér, igaz egy adaptív, így elvileg végtelen sokáig tartó eljárás keretében. Az eljárás alapja az ötvenes évek elejéről származó *Metropolis-mintavételezés* [90], amelynek magyar vonatkozása is van. A módszert közreadó, egyébiránt a hidrogénatom numerikus elemzéséről szóló cikk szerzői között, a névadó Metropolis úron kívül, megtaláljuk Edward Tellert, azaz a magyar Teller Edét is (az ötvenes évek, a hidrogénatom és Teller Ede a hidrogénbombát juttatják eszünkbe, de most az eljárást kizárólag békés célokra használjuk fel).

Tegyük fel, hogy a tartományt úgy szeretnénk mintavételezni, hogy egy pontjának a választási valószínűsége arányos legyen egy tetszőleges $\mathcal{I}$ fontosságfüggvénnyel! A globális illuminációs feladat megoldása során a fényutak terét kívánjuk mintavételezni olyan valószínűség-sűrűséggel, amely arányos a fényút által szállított teljesítménnyel. Mivel a teljesítményt több hullámhosszon párhuzamosan számítjuk, a fontosságfüggvényt a különböző hullámhosszok teljesítményeinek valamely súlyozott összegeként definiáljuk. Ha a súlyozás megfelel az emberi szem érzékenységének, akkor a *luminancia* fogalmához jutunk.



8.28. ábra. *Fényutak előállítása mutációval*

A minták előállítását egy alkalmas Markov-folyamatra bízzuk. A *Markov-folyamat* olyan *sztochasztikus folyamat*, azaz véletlen függvény, ahol a folyamat jövőbeli viselkedése csak az aktuális állapottól függ, és a múlttól független. Az üzleti életben például a holnapi vagyonunk — a szerencsénken kívül — csak a mai vagyonunktól függ, és független attól, hogy a mai vagyonunkat a múltban milyen lépésekben „guberáltuk” össze. A Markov-folyamat az állapotain fut végig, az üzleti életből vett példában az állapot egy

számmal, a pillanatnyi vagyonnal jellemezhető. A globális illuminációs feladat megoldása során az állapotok a fényforrást a szemmel összekötő fényutakat tartalmazzák, egy állapot tehát egy fényútmintát jelent. A véletlen átmeneteket pedig úgy konstruáljuk meg, hogy az egyes fényutakat éppen a fontosságukkal arányos valószínűséggel látogassa meg a folyamat.

A mintaelőállításhoz használható Markov-folyamathoz egy majdnem tetszőleges $T(\mathbf{z}_i \to \mathbf{z}_t)$ *kísérleti átmeneti függvény*t (*tentative transition function*) választunk, amely a $\mathbf{z}_i$ aktuális állapothoz egy $\mathbf{z}_{i+1}$ véletlenszerű kísérleti állapotot keres. A kísérleti állapotot vagy elfogadjuk és ekkor a kísérleti állapot lesz a következő állapot, vagy pedig elutasítjuk, és az aktuális állapotot tartjuk meg következő állapotnak is. Az $a(\mathbf{z}_i \to \mathbf{z}_t)$ elfogadási valószínűség megválasztása a módszer kulcsa. Olyan elfogadási valószínűséget keresünk, amely biztosítja, hogy az állandósult (*stacioner*) helyzetben a módszer éppen a fontosságfüggvénnyel arányos valószínűséggel látogatja meg az állapotokat.

Legyen az állandósult helyzetben a $\mathbf{z}$ előállításának valószínűsége $p(\mathbf{z})$! Annak valószínűsége, hogy egy $\mathbf{z}$ állapot után a következő állapot éppen $\mathbf{y}$ lesz, egyenlő a $\mathbf{z}$-ben tartózkodás, a kísérleti átmenet és az elfogadás valószínűségeinek a szorzatával, lévén, hogy ezek független események:

$$p(\mathbf{z}) \cdot T(\mathbf{z} \to \mathbf{y}) \cdot a(\mathbf{z} \to \mathbf{y}).$$

A rendszer akkor lehet állandósult helyzetben, ha a $p(\mathbf{z})$ és $p(\mathbf{y})$ valószínűségek nem változnak, tehát minden lépésben egy állapotból való kilépés valószínűsége egyenlő a belépés valószínűségével. A be- és kilépés valószínűségeinek egyenlősége még mindig túl nagy szabadságot ad nekünk az elfogadási valószínűség felírására, ezért egy további önkényes megkötést teszünk. Nem csak az állapotonkénti bemenő és kimenő valószínűségek egyenlőségét kívánjuk meg, hanem ehelyett azt a sokkal erősebb követelményt támasztjuk, hogy bármely két $\mathbf{z}$, $\mathbf{y}$ állapot között az átlépési valószínűség a két irányban megegyező legyen (*detailed balance*):

$$p(\mathbf{z}) \cdot T(\mathbf{z} \to \mathbf{y}) \cdot a(\mathbf{z} \to \mathbf{y}) = p(\mathbf{y}) \cdot T(\mathbf{y} \to \mathbf{z}) \cdot a(\mathbf{y} \to \mathbf{z}).$$

Az ehhez szükséges elfogadási valószínűségek aránya a következő:

$$\frac{a(\mathbf{y} \to \mathbf{z})}{a(\mathbf{z} \to \mathbf{y})} = \frac{p(\mathbf{z}) \cdot T(\mathbf{z} \to \mathbf{y})}{p(\mathbf{y}) \cdot T(\mathbf{y} \to \mathbf{z})}.$$

Azt szeretnénk, ha az állandósult helyzetben az állapotok meglátogatásának $p(\mathbf{z})$ valószínűsége az $\mathcal{I}(\mathbf{z})$ fontossággal arányos legyen, ezért a $p(\mathbf{z})$, $p(\mathbf{y})$ valószínűségek arányát a fontosságértékek arányából fejezzük ki:

$$\frac{p(\mathbf{z})}{p(\mathbf{y})} = \frac{\mathcal{I}(\mathbf{z})}{\mathcal{I}(\mathbf{y})}.$$

Ezen egyenletet kielégítő elfogadási valószínűségek közül azt érdemes választani, amelyben az átmeneti valószínűségek a lehető legnagyobbak, hiszen ekkor jutunk el a leghamarabb az állandósult helyzetig. Mivel a valószínűségek maximuma 1, a két irányra adódó elfogadási valószínűségek közül a nagyobbat egyre állítjuk be, a másikat pedig az előírt aránynak megfelelően. A végeredményt a következő elfogadási valószínűségben foglalhatjuk össze:

$$a(\mathbf{y} \to \mathbf{z}) = \min\left\{ \frac{\mathcal{I}(\mathbf{z}) \cdot T(\mathbf{z} \to \mathbf{y})}{\mathcal{I}(\mathbf{y}) \cdot T(\mathbf{y} \to \mathbf{z})},\ 1 \right\}.$$

Értelmezzük a kapott összefüggést arra az esetre, amikor a perturbációk szimmetrikusak, azaz $T(\mathbf{y} \to \mathbf{z}) = T(\mathbf{z} \to \mathbf{y})$! Ebben az esetben az elfogadási valószínűség a fontosságváltozás arányával egyezik meg:

$$a(\mathbf{y} \to \mathbf{z}) = \min\left\{ \frac{\mathcal{I}(\mathbf{z})}{\mathcal{I}(\mathbf{y})},\ 1 \right\}.$$

A nagyobb fontosságú pontba mindig átlépünk, az alacsonyabb fontosságúba pedig csak a fontosságcsökkenés valószínűségével. Ebből is érezhető, hogy a folyamat a magasabb fontosságú pontokat gyakrabban fogja meglátogatni.

Összefoglalva a Metropolis-mintavételezés a következő folyamattal állítja elő a véletlen mintákat:

```
for (i = 1; i ≤ M; i++) {                                    // M a fényútminták száma
    A z_i állapotból egy z_t kísérleti állapot választása a T(z_i → z_t) alapján;
    a(z_i → z_t) = I(z_t)·T(z_t → z_i) / (I(z_i)·T(z_i → z_t));
                                                  // Elfogadás a(z_i → z_t) valószínűséggel
    Egyenletes eloszlású r véletlen szám előállítása a [0,1] intervallumban;
    if (r < a(z_i → z_t))  z_{i+1} = z_t;
    else                   z_{i+1} = z_i;
}
```

A Metropolis-algoritmus állandósult állapotában éppen $\mathcal{I}(\mathbf{z})$ fontossággal arányos valószínűség-sűrűséggel látogatja meg az állapotokat, azaz fennáll a $\mathcal{I}(\mathbf{z}) = b \cdot p(\mathbf{z})$ egyenlőség. Az b arányossági tényezőt abból a feltételből kapjuk, hogy a p valószínűség-sűrűség, tehát integrálja 1:

$$b = \int_{\mathcal{P}} \mathcal{I}(\mathbf{z})\, d\mathbf{z},$$

ahol $\mathcal{P}$ az összes fényút tartománya. Mivel a valószínűség-sűrűséget nem, csak a fontosságot ismerjük közvetlenül, a Monte-Carlo integrálbecslést olyan formára kell hozni, amelyben a valószínűség-sűrűség helyett a fontosság szerepel:

$$\int_{\mathcal{P}} f(\mathbf{z})\, d\mathbf{z} = \int_{\mathcal{P}} \frac{f(\mathbf{z})}{\mathcal{I}(\mathbf{z})} \cdot \mathcal{I}(\mathbf{z})\, d\mathbf{z} = b \cdot \int_{\mathcal{P}} \frac{f(\mathbf{z})}{\mathcal{I}(\mathbf{z})} \cdot p(\mathbf{z})\, d\mathbf{z} = b \cdot E\left[\frac{f(\mathbf{z})}{\mathcal{I}(\mathbf{z})}\right] \approx \frac{b}{M} \cdot \sum_{i=1}^{M} \frac{f(\mathbf{z}_i)}{\mathcal{I}(\mathbf{z}_i)}.$$

A globális illuminációs feladat megoldásakor a $\mathbf{z}$ minták fényutakat képviselnek, amelyek a fényforrást a szemmel visszaverődéseken és töréseken keresztül kötik össze. Az $f(\mathbf{z})$ a fényút képhozzájárulását jelenti.

A globális illuminációs feladat megoldása a képernyő minden egyes pixelén egy integrál kiszámítását igényli, amelyben a j-edik pixel $W_j^e(\mathbf{z})$ érzékenységfüggvényének és a pixelben látható felület $L(\mathbf{z})$ sugársűrűségének a szorzata szerepel:

$$P[j] = \int_{\mathcal{P}} W_j^e(\mathbf{z}) \cdot L(\mathbf{z}) \, d\mathbf{z}.$$

Ebben az alakban a $W_j^e(\mathbf{z})$ egy pixelre azon fényutakat választja ki, amelyek a szemben végződnek és az adott pixelen mennek át, az $L(\mathbf{z})$ pedig általánosan egy fényút végén mérhető sugársűrűség. Az $L(\mathbf{z})$ sugársűrűség független a pixeltől, így ez a tag minden integrálban közös. Az összes pixel értékét egyetlen perturbációs folyamattal számíthatjuk ki, ha az $\mathcal{I}$ fontosságfüggvényt csak az $L(\mathbf{z})$ sugársűrűséggel próbáljuk meg arányossá tenni. Ekkor az előző egyenlet a j-edik pixelre a következő alakú:

$$P[j] = \int_{\mathcal{P}} W_j^e(\mathbf{z}) \cdot \frac{L(\mathbf{z})}{\mathcal{I}(\mathbf{z})} \cdot \mathcal{I}(\mathbf{z}) \, d\mathbf{z} \approx \frac{b}{M} \cdot \sum_{i=1}^{M} W_j^e(\mathbf{z}_i) \cdot \frac{L(\mathbf{z}_i)}{\mathcal{I}(\mathbf{z}_i)}.$$

A következő kritikus pont a zérus fontosságú részek kezelése. A globális illuminációs feladatban gyakran előfordul, hogy a fényutak terében egyes régiók jelentős hozzájárulású utakat tartalmaznak, de a régiók közötti fényutak hozzájárulása zérus. Gondoljunk csak két szobára, az egyikben van a kamera, a másikban a fényforrás, és a két szoba között kis lyukak vannak! A Metropolis-eljárás minden olyan kísérleti fényutat visszautasít, amelyik nem bújik át valamelyik lyukon. Ha a mutációs méret nem elegendően nagy ahhoz, hogy egyetlen lyukon átmenő fényútból egy másik lyukon átmenő utat csináljon, a Markov-folyamat képtelen lesz a teljes fontos fényútteret feltérképezni. A Metropolis fényterjedés eredeti algoritmusa ezt úgy oldja meg, hogy véletlenszerűen az aktuális fényúttól teljesen független kísérleti mintákat is előállít.

Végül meg kell jegyeznünk, hogy a visszautasított minták ugyancsak hasznos információt hordoznak a megvilágítási viszonyokat illetően, így ezek eldobása pazarló. Vegyük figyelembe, hogy a kísérleti mintát a valószínűséggel fogadjuk el, míg az eredeti mintát $1-a$ valószínűséggel tartjuk meg! Cseréljük fel ezt a véletlen változót a várható értékével (ez egy jól bevált szóráscsökkentési eljárás), és mind az eredeti, mind pedig a kísérleti mintát építsük be az integrálformulába úgy, hogy a kísérleti minta hozamát a-val, az eredetiét $(1-a)$-val súlyozzuk.

Összefoglalva a módosított Metropolis-algoritmus pszeudo-kódja az alábbi:

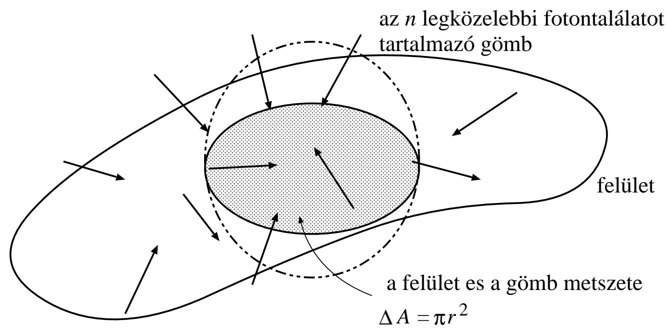
A $b = \int \mathcal{I} \, d\mathbf{z}$ becslése;
A $\mathbf{z}_1$ első minta előállítása;

```
for (i = 1; i ≤ M; i++) {                                  // M a fényútminták száma
    A z_i-ből egy z_t kísérleti minta előállítása a T(z_i → z_t)-vel;
    a(z_i → z_t) = min{ I(z_t)·T(z_t→z_i) / (I(z_i)·T(z_i→z_t)), 1 };
    Azon j pixel meghatározása, amelyhez a z_i hozzájárul;
    Φ_j += b/M · W_j(z_i) · L(z_i)/I(z_i) · (1 − a(z_i → z_t));
    Azon k pixel meghatározása, amelyhez a z_t hozzájárul;
    Φ_k += b/M · W_k(z_t) · L(z_t)/I(z_t) · a(z_i → z_t);
                                             // Elfogadás a(z_i → z_t) valószínűséggel
    Egyenletes eloszlású r véletlen szám előállítása a [0,1] intervallumban;
    if (r < a(z_i → z_t))  z_{i+1} = z_t;
    else                   z_{i+1} = z_i;
}
```

A Metropolis-módszer egy hatékony, mégis egyszerű implementációjához jutunk, ha egy tetszőleges, véletlen bolyongási algoritmust (inverz fénykövetés, kétirányú fénykövetés stb.) úgy módosítunk, hogy az egység intervallumba eső véletlen számokat nem a véletlenszám generátor újbóli hívásával, hanem az előző értékek kismértékű perturbációjával számítjuk ki [70].

8.10.5. Foton térkép

A kétirányú fényútkövetés egy gyűjtősétát egyetlen lövősétával köt össze. Milyen jó lenne, ha először a lövősétákat számíthatnánk ki, és a gyűjtősétákat pedig nem csupán egyetlen egy, hanem egyszerre az összes lövősétával összekapcsolhatnánk! Kívánságunkat a foton térképek [62, 63, 64] alkalmazásával teljesíthetjük [4]. A *foton térkép* (*photon-map*) olyan adatstruktúra, amely a sok lövőséta hatását tömören tárolja.



8.29. ábra. *Foton térkép*

[4]Ezzel az algoritmussal dolgozik a Mental Ray program (http://www.mentalimages.com)

A foton térkép a foton találatok gyűjteménye. Egy találatot a foton által a különböző hullámhosszokon szállított energiával (ez nem igazi fizikai foton, amely csak egyetlen hullámhosszon vinne energiát), a találat helyével, a foton érkezési irányával és a felületi normálissal együtt tárolunk (a felületi normálist ugyan a találat helyéből mindig kiszámíthatnánk, de annak tárolásával egy kis többletmemória árán számítási időt takaríthatunk meg). A foton találatokat a hatékony előkeresés érdekében *kd-fa* adatstruktúrába szervezzük (6.4. fejezet). A gyűjtőséták alatt az árnyalási egyenlet következő közelítésével dolgozunk:

$$L(\vec{x},\vec{\omega}) = \int_{\Omega} L(h(\vec{x},-\vec{\omega}'),\vec{\omega}') \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \, d\omega' =$$

$$\int_{\Omega} \frac{d\Phi(\vec{\omega}')}{dA\cos\theta' d\omega'} \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \, d\omega' \approx \sum_{i=1}^{n} \frac{\Delta\Phi(\vec{\omega}_i')}{\Delta A} \cdot f_r(\vec{\omega}_i',\vec{x},\vec{\omega}), \qquad (8.36)$$

ahol $\Delta\Phi(\vec{\omega}_i')$ a ΔA felületre az $\vec{\omega}_i'$ irányból érkező foton energiája. A $\Delta\Phi$ és a ΔA mennyiségeket az $\vec{x}$ pont környezetében található foton találatok tulajdonságaiból közelítjük a következő eljárással (8.29. ábra). Az $\vec{x}$ pont köré egy gömböt teszünk, amelyet addig pumpálunk, amíg az éppen n foton találatot tartalmaz (az n az algoritmus globális paramétere, általában 20 és 200 között van). Ha ekkor a gömb sugara r, akkor a gömb által a felületből kimetszett felületelem területe $\Delta A = \pi r^2$.

Az algoritmus három jól elkülöníthető fázisra bontható:

- Fotonok lövöldözése (*photon shooting*).
- A foton térkép felépítése és a kd-fa kiegyenlítése.
- Képszintézis (*final gathering*).

Az eredeti fotontérkép módszerben a fotontérképek jelentős memóriát emészthetnek fel, a számítási idő döntő részét pedig az n legközelebbi foton megkeresésével töltjük. A memóriaigény csökkenthető, ha a fotontalálatokat tömörítetten tároljuk. A pozíció három float változóban megadható (12 bájt). A foton három hullámhosszon (R,G,B) képviselt teljesítménye leírható 4 bájton, ha bevetjük a Ward [135] által javasolt „*valós pixel*” módszert. Egy valós pixel a lebegőpontos $[R,G,B]$ hármast, három mantisszában és egyetlen közös exponensben tárolja. Az exponenst eltolt bináris formában ábrázoljuk, azaz 128-t adunk hozzá az értékéhez. Végül a három valós $[R,G,B]$ számot egy $[rm,gm,bm,ex]$ bájt négyessel adjuk meg. A konverziókhoz a C könyvtár `frexp` és `ldexp` rutinjait használhatjuk. Az `frexp` egy valós szám egész exponensét és a $[0.5,1]$ tartományba eső mantisszáját adja vissza. Az `ldexp` az inverz műveletet hajtja végre.

Az $[R,G,B]$-ről $[rm,gm,bm,ex]$-re konvertáló eljárás:

```
m = max{R,G,B};
if (m < 10^-32) {
    rm = gm = bm = ex = 0;
} else {
    v = frexp(m,e) · 256.0/m;                    // e a visszaadott exponens
    rm = R · v;   gm = G · v;   bm = B · v;
    ex = e + 128;
}
```

Az inverz művelet, amely egy $[rm, gm, bm, ex]$ négyest $[R, G, B]$-re alakít:

```
if (ex == 0) {
    R = G = B = 0;
} else {
    v = ldexp(1/256, ex − 128);
    R = (rm + 0.5) · v;   G = (gm + 0.5) · v;   B = (bm + 0.5) · v;
}
```

A beérkezési irány és a felületi normális szintén egy-egy bájton leírható, ha a gömböt 256 diszkrét tartományra osztjuk fel. Kihasználhatjuk továbbá, hogy csak a kausztikus jelenségekért felelős fotonoknak kell nagyon sűrűn lefedniük a felületet, ezért érdemes a kausztikus fotonokat külön fotontérképben tárolni.

8.30. ábra. *Az eredeti és a javított foton térkép módszer*

A legközelebbi n foton felhasználása sajátos szűrési mintákat eredményez (8.30. ábra). Ezen úgy segíthetünk, hogy csak ott használjuk a foton térképek eredményét, ahol a képernyőre tett hatás kicsiny. Ezt a hatást a fényúton található BRDF-ek és koszinuszos tényezők szorzata fejezi ki. Ahol a hatás nagy, ott a szokásos Monte-Carlo becslést alkalmazzuk, azaz árnyéksugarakat küldünk a fényforrások felé és BRDF

mintavételezéssel a többi felület felé is, majd az onnan visszaérkező sugársűrűségértékekből becsüljük a visszavert sugársűrűséget. Az újabb felületeken ismét eldöntjük, hogy a foton térkép becslését alkalmazzuk-e, vagy folytatjuk a Monte-Carlo eljárást. Ha a sugár hossza kicsiny (például egy sarokban vagyunk), akkor érdemes mindenképpen folytatni a Monte-Carlo becslést, de viszonylag kevés, 10–20 sugárral.

Spekuláris visszaverődést nem tartalmazó, azaz csak diffúz és ideális tükör, illetve törő kombinációjával előállított anyagok esetén a legközelebbi fotonok megkeresését jelentősen lehet gyorsítani [30]. Az előfeldolgozási fázisban a diffúz visszaverődést minden egyes fotontalálat helyén egyszer becsüljük a környéken lévő fotonok alapján. A képszintézis során pedig csak a legközelebbi fotontalálatot keressük meg, és az itt tárolt diffúz sugársűrűséget használjuk fel.

8.11. A globális illuminációs feladat iterációs megoldása

A véletlen bolyongás mélységi kereséssel építi fel a fényutakat, azaz egy fényút hosszát addig növeli, amíg az össze nem köti a fényforrást a szemmel, vagy amíg a továbbépítése nem reménytelen. Amennyiben szélességi keresést alkalmazunk, tehát egy lépésben az összes fényutat egy lépéssel megtoldjuk, akkor másfajta algoritmushoz, az iterációhoz jutunk. Matematikai szempontból az *iteráció* alapja az a felismerés, hogy az árnyalási egyenlet megoldása a következő iterációs séma *fixpont*ja[5]:

$$L^{(m)} = L^e + \mathcal{T}_{f_r} L^{(m-1)}. \tag{8.37}$$

Ha az iteráció konvergens, akkor bármely kezdeti függvényből a megoldáshoz konvergál. A konvergenciát az biztosítja, hogy energiamegmaradást nem sértő, azaz egynél kisebb albedójú anyagmodellek esetén minden visszaverődés után az energia csökken.

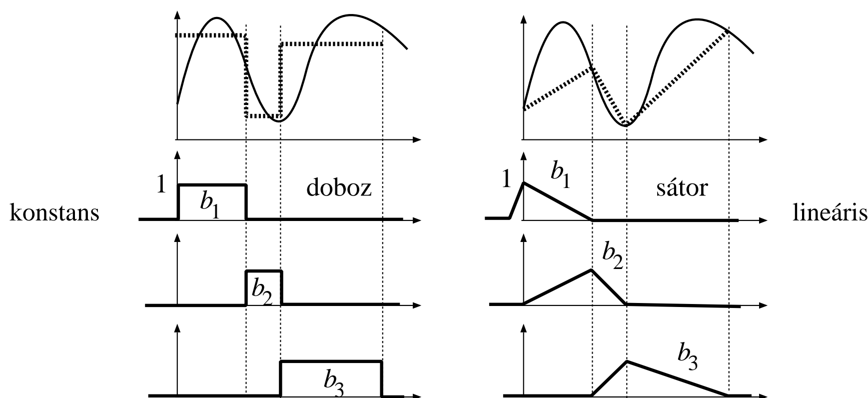
8.11.1. Végeselem-módszer

Ahhoz, hogy a sugársűrűség-függvényt az iterációs formulába be tudjuk helyettesíteni, annak ideiglenes változatát tárolni kell. Ez korántsem egyszerű, hiszen a sugársűrűségfüggvény végtelen sok pontra és irányra ad intenzitás értéket. A folytonos paraméterű függvények véges adattal történő közelítő megadására a *végeselem-módszer*t (*finite-element method*) használhatjuk. Ennek lényege, hogy a függvényt függvénysorral közelítjük, azaz a következő alakban keressük:

$$L(\mathbf{z}) \approx \sum_{j=1}^{n} \mathbf{L}_j \cdot b_j(\mathbf{z}), \tag{8.38}$$

[5] fixpont alatt azt az L-t értjük, amelyet behelyettesítve az iterációs képletbe önmagát kapjuk vissza

ahol a $b_j(\mathbf{z})$-k előre definiált bázis függvények, az $\mathbf{L}_j$-k pedig skalár tényezők (8.31. ábra). Ha a bázisfüggvényeket úgy kapjuk, hogy a tartományt résztartományokra bontjuk, és a j-edik bázisfüggvényt a j-edik tartományban egynek tekintjük, az összes többiben pedig zérusnak, akkor a végeselem közelítés megfelel a függvény tartományonkénti konstans közelítésének. Ha viszont a j-edik bázisfüggvény a j-edik tartományhatáron 1 értékű, és lineárisan csökken a szomszédos tartományhatárokig, akkor a végeselem módszer a függvényt tartományonként lineárisan közelíti.



8.31. ábra. *Függvények leírása véges számú adattal: a végeselem-módszer*

Leggyakrabban a konstans bázisfüggvényeket alkalmazzuk, azaz a közelítendő függvény értelmezési tartományát véges számú $\mathcal{Z}_1, \ldots, \mathcal{Z}_n$ tartományra bontjuk, és az i-edik bázisfüggvényt 1 értékűnek tekintjük az i-edik tartományban, minden más tartományban pedig 0-nak. A megoldandó $L(\mathbf{z}) = L^e(\mathbf{z}) + \mathcal{T}_{f_r} L(\mathbf{z})$ integrálegyenletbe a függvénysoros közelítést behelyettesítve a következő egyenletet kapjuk:

$$\sum_{j=1}^{n} \mathbf{L}_j \cdot b_j(\mathbf{z}) \approx \sum_{j=1}^{n} \mathbf{L}_j^e \cdot b_j(\mathbf{z}) + \mathcal{T}_{f_r} \sum_{j=1}^{n} \mathbf{L}_j \cdot b_j(\mathbf{z}).$$

Szorozzuk meg az egyenlet minkét oldalát $b_i(\mathbf{z})$-vel és integráljuk a teljes tartományban! Mivel $b_i(\mathbf{z})$ csak a $\mathcal{Z}_i$-ben zérustól különböző, ahol az összes többi bázisfüggvény zérus, a bázisfüggvénnyel történő szorzás majd integrálás a következő egyenletre vezet:

$$\mathbf{L}_i \cdot \int_{\mathcal{Z}_i} 1 \; d\mathbf{z} = \mathbf{L}_i^e \cdot \int_{\mathcal{Z}_i} 1 \; d\mathbf{z} + \sum_{j=1}^{n} \cdot \int_{\mathcal{Z}_i} \mathcal{T}_{f_r} b_j(\mathbf{z}) d\mathbf{z} \cdot \mathbf{L}_j.$$

Osszuk el mindkét oldalt az i-edik tartomány méretével, azaz $\mathcal{Z}_i = \int_{\mathcal{Z}_i} 1 \, d\mathbf{z}$-vel:

$$\mathbf{L}_i = \mathbf{L}_i^e + \sum_{j=1}^{n} \frac{\int_{\mathcal{Z}_i} \mathcal{T}_{f_r} b_j(\mathbf{z}) d\mathbf{z}}{\mathcal{Z}_i} \cdot \mathbf{L}_j.$$

A végeselem-módszer alkalmazása a függvénysor együtthatóira egy lineáris egyenletrendszert eredményezett:

$$\mathbf{L} = \mathbf{L}^e + \mathbf{R} \cdot \mathbf{L}, \qquad \text{ahol } \mathbf{R}_{ij} = \frac{1}{\mathcal{Z}_i} \cdot \int_{\mathcal{Z}_i} \mathcal{T}_{f_r} b_j(\mathbf{z}) d\mathbf{z}. \tag{8.39}$$

Ez a lineáris egyenletrendszer iterációval, vagy *Gauss-elimináció* alkalmazásával már megoldható. A Gauss-elimináció a gyakorlatban nem ajánlott, mert a számítási szükséglete az ismeretlenek számának köbével arányos.

Az iteráció számítási ideje azonban csupán az ismeretlenek számának négyzetével arányos, ráadásul numerikusan stabil és konvergens, ha az $\mathbf{R}$ mátrix valamely normája (például az egyes sorok elemeinek abszolút értékeiből képzett összegek maximumát tekinthetjük a mátrix normájának) 1-nél kisebb. A mi esetünkben ez valóban fennáll, köszönhetően annak, hogy az energiamegmaradást betartó anyagmodellek az $\mathbf{R}$ mátrix normáját 1 alá szorítják. Így a megoldást a következő iterációs eljárással kapjuk meg:

$$\mathbf{L}^{(m)} = \mathbf{L}^e + \mathbf{R} \cdot \mathbf{L}^{(m-1)}. \tag{8.40}$$

Figyeljük meg, hogy szemben a véletlen bolyongással, amely egymástól független mintákból közelíti a megoldást, az iteráció mindig az előző lépés eredményét finomítja! Az iteráció tehát képes kihasználni a korábbi lépések „ismereteit”, így elvileg lényegesen kevesebb lépésben konvergál. Amíg a Monte-Carlo családba tartozó véletlen bolyongás hibája m lépés után $\mathcal{O}(m^{-0.5})$ nagyságrendű, addig az iteráció geometriai sor szerint konvergál, tehát néhány (gyakorlatban 6–10) iteráció után már nem változik az eredmény. A menyasszony azonban mégsem olyan gyönyörű, mint amilyennek az első pillanatban látszik. A gyakorlatban általában nagyon nagy méretű egyenletrendszer adódik, hiszen az ismeretlenek száma megegyezik a végeselemek számával. Mivel a sugársűrűség-függvény általános esetben négydimenziós (két dimenzió a felületi pontot, újabb két dimenzió pedig az irányt azonosítja), ráadásul gyorsan változó, a bázisfüggvények száma könnyen milliós (milliárdos) nagyságrendű lehet. Ilyen méretű mátrixokkal pedig nem öröm iterálni, hiába kell csupán néhány iterációs lépést megtenni. Kedvezőbb helyzetben vagyunk, ha a felületek és a fényforrások csak diffúz jellegűek, ugyanis ekkor a sugársűrűség-függvény csak a felületi ponttól függ, a nézeti iránytól viszont független. Ekkor elegendő a felületeket kis foltokra (háromszögekre) bontani, és minden felületelemhez egyetlen sugársűrűség-értéket rendelni. Az ismeretlenek száma a

felületelemek száma lesz, ami még mindig több tíz- vagy százezer lehet. Ezt a speciális esetet *radiozitás*nak (*radiosity*) nevezzük [118].

Az általános, nem diffúz esetben a teljes mátrix felírásáról nem is álmodhatunk, így olyan iterációs sémára van szükségünk, amely a mátrixnak mindig csak egy kis részét használja. Egy lehetséges megoldást ismertetünk, amely a feladatot randomizálja, ezért kapta a *sztochasztikus iteráció* (*stochastic iteration*) [117] nevet. Térjünk vissza az eredeti iterációs sémához, és helyettesítsük a fényátadás operátort egy véletlen operátorral, amely csak átlagban (várható értékben) adja vissza az eredeti operátor hatását:

$$L(m) = L^e + \mathcal{T}^*_{f_r} L(m-1), \qquad E[\mathcal{T}^*_{f_r} L] = \mathcal{T}_{f_r} L. \tag{8.41}$$

Egy véletlen iterációs séma nem konvergál, hanem az iterált értékek a kívánt eredmény körül fluktuálnak. Ezért az egyes iterációs lépések után az $L(m)$ aktuális sugársűrűségből egy $\mathcal{M}L(m)$ képbecslőt számítunk, majd ezen képbecslők átlagaként állítjuk elő a végső képet:

$$P(m) = \frac{1}{m} \cdot \sum_{n=1}^{m} \mathcal{M}L(n) = \frac{1}{m} \cdot \mathcal{M}L(m) + \left(1 - \frac{1}{m}\right) \cdot P(m-1).$$

Az elmélet általános elveivel készen is volnánk, csupán olyan véletlen operátorokat kell találnunk, amelyek átlagban visszaadják a valódi fényátadás operátor hatását. Nagyon sok ilyen operátor létezik, amelyek közül azokat érdemes felhasználni, amelyek

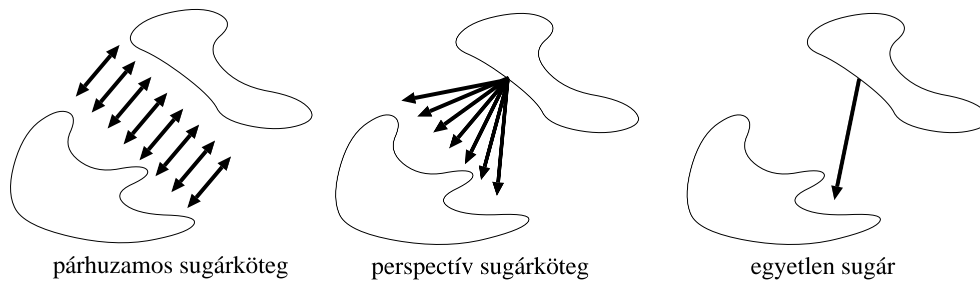
- legalább részben építenek a korábbi iteráció eredményére, így gyorsabban konvergálnak, mint a független mintákkal dolgozó véletlen bolyongás,
- nem függenek a teljes sugársűrűség-függvénytől, tehát azt nem kell egészben tárolnunk, ezért a klasszikus iteráció óriási tárolási igényétől megmenekülünk,
- az aktuális hardveren hatékonyan számíthatók.

A következőkben három különböző eljárást mutatunk be (8.32. ábra) röviden. Ezek a módszerek a véletlen bolyongás gyakran órás nagyságrendű számítási ideje helyett néhány másodperc alatt kiszámítják a képet.

8.11.2. Párhuzamos sugárköteg módszer

A párhuzamos sugárköteg módszer egy véletlen irányt választ és az összes felületi pont sugársűrűségét ebben a véletlen irányban adja tovább [117].

A sugársűrűség átviteléhez azokat a pontpárokat kell azonosítani, amelyek az adott irányban látják egymást. Helyezzünk egy ablakot az irányra merőlegesen, és bontsuk fel az ablakot kis pixelekre! Az egyes pixelekben látható felületek azonosításához az inkrementális láthatósági algoritmusok, sőt akár a grafikus kártyák z-buffere is felhasználható.

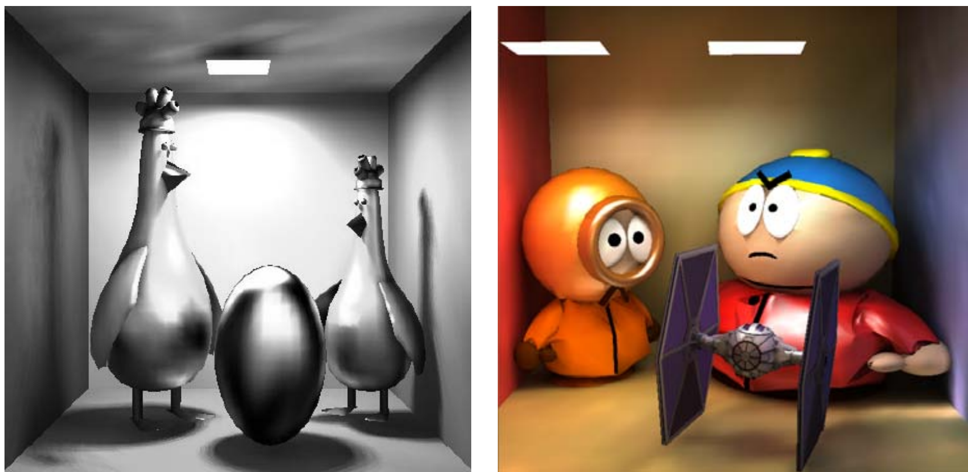


8.32. ábra. *Három elemi véletlen operátor*

8.33. ábra. *Interaktív navigáció párhuzamos sugárkötegekkel [121]*

8.11.3. Perspektív sugárköteg módszer

A perspektív sugárköteg módszer egyetlen pontot választ, és ezen pont sugársűrűségét az összes olyan pontba átviszi, amely innen látható [11]. A fontosság szerinti mintavételezés elveinek megfelelően érdemes a pontot a kis környezete által kisugárzott teljes teljesítménnyel arányos valószínűséggel kiválasztani. A véletlen operátor kiértékeléséhez egy pontból látható többi felületi pontot kell azonosítani, amit a grafikus hardver segítségével tehetünk meg. Tegyünk félkocka elrendezésben 5 ablakot a lövőpont és a lövőpontot tartalmazó felület fölé és fényképezzük le a teret az ablakokon keresztül a lövőpontot tekintve a szemnek! A fényképezés során az i-edik felületelem színét állítsuk éppen i-re, ugyanis ekkor a keletkezett képek színeiből könnyen eldönthetjük, hogy a lövőpontból mely más felületek és mekkora térszög alatt látszanak.

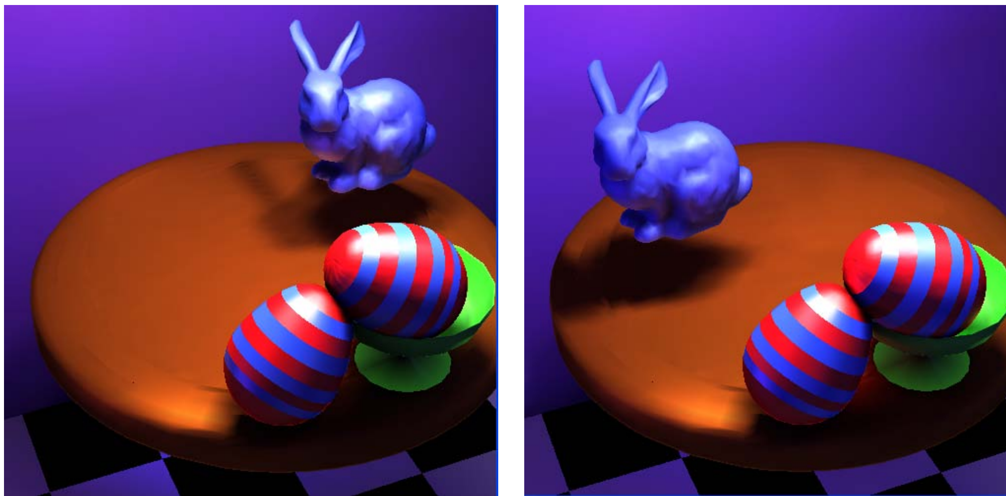


8.34. ábra. *Perspektív sugárkötegekkel 28 másodperc alatt (P4/2GHz) számított képek*

8.11.4. Sugárlövés módszer

A sugárlövés módszer egyetlen véletlen sugarat használ a sugársűrűség átvitelére [120].
A sugár kezdőpontját és irányát a fontosság szerinti mintavételezés szerint a sugársűrűséggel arányosan célszerű mintavételezni. Egy ilyen eljárással készítettük a 8.35. képet. A sugarakat a sugárkövetés jól ismert algoritmusaival követhetjük.

8.35. ábra. *Sugáriterációval 50 másodperc alatt számított kép*

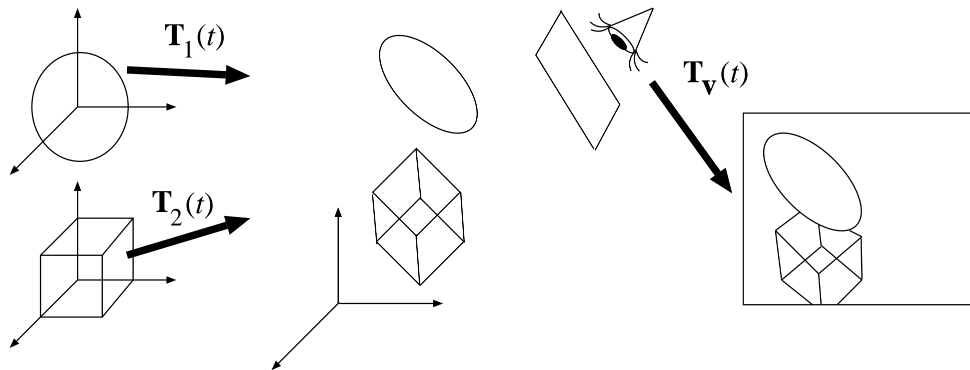


8.36. ábra. *Sztochasztikus iterációval készült 1 kép/sec sebességű globális illumináció*

9. fejezet

Animáció

Az *animáció* a virtuális világ és a kamera tulajdonságait az időben változtatja, amit a szemlélőnek követnie kell, aki így már nem egyetlen képet, hanem egy időben változó képsorozatot érzékel. Elméletileg a virtuális világ és a kamera bármilyen paramétere módosulhat, legyen az pozíció, orientáció, méret, szín, normálvektor, BRDF, alak stb., de ebben a könyvben főleg csak a mozgás (modellezési transzformáció) és a kamera (nézeti transzformáció) animációjával foglalkozunk.



9.1. ábra. *Az animáció a modellezési és a nézeti transzformációk időbeli változása*

A mozgás megjelenítéséhez az animáció nem csupán egyetlen képet állít elő, hanem egy teljes képsorozatot, ahol minden egyes kép egyetlen időpillanatnak felel meg. A felhasználóban a mozgás illúzióját kelthetjük, ha a képsorozat képeit gyorsan egymás után jelenítjük meg. Az egyes objektumok geometriáját a lokális modellezési-koordinátarendszereikben adjuk meg, így az objektum pozíciója illetve orientációja a világ-koordinátarendszerbe átvivő modellezési transzformáció változtatásával vezérelhető (a 9.1. ábrán a $\mathbf{T}_1$ és $\mathbf{T}_2$).

A kamera pozíciójával, orientációjával, látószögeivel és vágósíkjaival definiáljuk a nézeti transzformációt, amely az objektumot a világ-koordinátarendszerből a képernyő-koordinátarendszerbe viszi át (a 9.1. ábrán a $\mathbf{T}_V$).

A transzformációk 4×4-es mátrixokkal írhatók le. Legyen az o objektum időfüggő modellezési transzformációja $\mathbf{T}_{M,o}(t)$, az időfüggő nézeti transzformáció pedig $\mathbf{T}_V(t)$. Az animációhoz elegendően sűrű időpillanatokban kiszámítjuk a mátrixokat, transzformáljuk a testeket, majd végrehajtjuk a képszintézis műveletet. Az időt a számítógép órájáról is leolvashatjuk. A beépített órát használó animációs program vázlata:

```
Óra inicializálás(t_start);
for (t = t_start; t < t_end; t = Óra lekérdezés) {
    for (minden egyes o objektumra) T_M,o = T_M,o(t);
    T_V = T_V(t);
    Képszintézis;
}
```

A felhasználó akkor érzékeli a képsorozatot folyamatos mozgásként, ha másodpercenként legalább 15 képet vetítünk neki (a mozifilmekben 24 képet láthatunk másodpercenként). Ha a számítógép képes ilyen sebességgel elvégezni a képszintézis lépéseit, akkor *valós idejű animáció*ról beszélünk. Ha viszont a számítógépünk nem ilyen gyors, akkor az animáció két fázisban készülhet. Az elsőben kiszámítjuk és a háttértárra mentjük a képeket, majd a másodikban a háttértárról beolvasva a mozgáshoz szükséges sebességgel visszajátsszuk őket. Az eljárás neve *nem valós idejű animáció*, amelynek általános programja:

```
for (t = t_start; t < t_end; t += Δt) {                      // képrögzítés
    for (minden egyes o objektumra) T_M,o = T_M,o(t);
    T_V = T_V(t);
    Képszintézis és a kép eltárolása;
}

Óra inicializálás(t_start)                                    // animáció: visszajátszás
for (t = t_start; t < t_end; t += Δt) {
    Következő kép betöltése és kirajzolása;
    while (t + Δt > Óra lekérdezés) Várj;
}
```

Amennyiben az első fázisban a képeket valamely szabványos formátumban (*MPEG*, *AVI*) mentjük el, akkor a második fázist már bármely mozgóképlejátszó program elvégzi helyettünk.

9.1. Folyamatos mozgatás különböző platformokon

Az ablakozott felhasználói felületek (Ms-Windows, X-Window) *eseményvezérelt paradigma* szerint működnek (2.4. fejezet). A felhasználói felület *eseménykezelő ciklus*a periodikusan ellenőrzi, hogy történt-e olyan, az adott ablakhoz tartozó esemény, amelyre az ablakhoz tartozó alkalmazásnak reagálnia kell. Ha történt ilyen esemény (például a felhasználó lenyomta, vagy elengedte a klaviatúra valamely billentyűjét, megmozdította az egeret, vagy az ablakunk tartalmát más ablak tönkretette, így most újra kell rajzolni), akkor az ablakozó környezet meghívja az alkalmazói program ezen eseménytípushoz rendelt kiszolgáló függvényét. A kiszolgáló függvény lefutása után a közönséges interaktív programok újabb felhasználói eseményekre várnak.

Az animációnak azonban folyamatosan futnia kell, akkor is, ha a felhasználó nem is nyúl hozzá a klaviatúrához. El kell tehát érnünk, hogy a program akkor is megkapja a vezérlést, ha történetesen nincs felhasználói beavatkozás, így olyan esemény sem, ami miatt az ablakozó rendszer meghívná az alkalmazásunk valamely függvényét.

A folyamatos mozgatást megoldó első ötletünk az lehet, hogy a program indulása után nem térünk vissza az ablakozó környezethez, hanem rögtön a szimulációs hurok végtelen ciklusának a végrehajtásához fogunk. Ez az ötlet azonban rossz, ugyanis nem ad lehetőséget arra, hogy az ablakozó rendszer figyelemmel kövesse a felhasználói beavatkozásokat, így programunk nem fog reagálni a billentyű lenyomásokra, sőt még leállítani sem hagyja magát. Olyan ciklusra van tehát szükség, amely mind az ablakozó rendszer eseményfigyelő hurkából, mind pedig a program szimulációs hurkából végrehajt egy-egy részt. A folyamatos működtetést úgy érhetjük el, hogy az animáció egy lépését az ablakozó program eseményfigyelő ciklusában hajtjuk végre. A GLUT rendszerben nem nyúlhatunk bele közvetlenül az eseményfigyelő ciklusba, de szerencsére a GLUT maga biztosít kiugrási lehetőséget. Az ablakokhoz ugyanis egy *üresjárati eseményt* (*idle callback*) is rendelhetünk, amelyhez tartozó eseménykezelőt a GLUT akkor is meghívja, ha éppen nincs más feldolgozandó esemény. A folyamatos működtetést tehát itt kell elvégezni. Legyen az üresjárati eseménykezelő az `IdleFunc()` függvény, amelynek GLUT eseménykezelőkénti regisztrációja a következőképpen történik:

```
glutIdleFunc(IdleFunc);
```

Az `IdleFunc()` törzsében kiderítjük az eltelt időt, és végrehajtjuk az animáció egyetlen lépését. Az animáció készítésekor illetve lejátszásakor szükségünk van az aktuális időre. A GLUT környezetben az eltelt, ezredmásodpercben mért idő egy állapotváltozó, amit a `glutGet(GLUT_ELAPSED_TIME)` hívással kérdezhetünk le. A megismert időkezelő felhasználásával az üresjárati függvény az eltelt keretidőnek megfelelő szimulációs lépést hajtja végre:

```
long time;
//-----------------------------------------------------------------
void IdleFunc(void) { // Üresjárati esemény
//-----------------------------------------------------------------
    long newTime = glutGet(GLUT_ELAPSED_TIME); // indulás óta eltelt idő
    Application::gApp->Do_a_Step((newTime - time)/1000.0);  // egy lépés
    time = newTime;
}
```

Az alkalmazás `Do_a_Step()` függvényének az eltelt időt másodpercekben adjuk át. Ebben a függvényben sorra kell vennünk a mozgó objektumokat és a kamerát, és meg kell hívnunk az objektumok `AnimateIt()` függvényét, amely az objektumot az új helyzetbe mozgatja, végül az új helyzetnek megfelelően újrarajzoljuk a képet.

Ms-Windows környezetben a főciklus a kezünkben van, így abban is elhelyezhetjük az animáció egyetlen lépésének végrehajtatását. Az időlekérdezéshez a szabványos C könyvtárra támaszkodhatunk, vagy akár az Ms-Windows nagyobb felbontású órájára is ránézhetünk. Az alábbi megoldás a C könyvtár `clock()` függvényét használja:

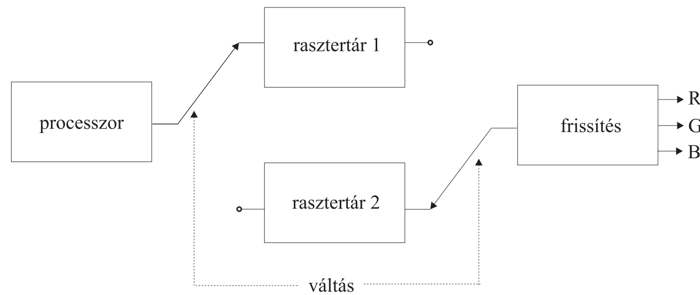
```
while (msg.message != WM_QUIT) {   // a fő üzenethurok
   if (PeekMessage(&msg, NULL, 0U, 0U, PM_REMOVE)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   } else {
      long newTime = clock();      // óra lekérdezés
      Application::gApp->Do_a_Step((newTime - time)/1000.0); // egy lépés
      time = newTime;
   }
}
```

A főciklusban a `PeekMessage()` függvényt használjuk, mert ez a `GetMessage()` függvénnyel ellentétben nem vár arra, hogy egy üzenet érkezzen, hanem akkor is visszatér, ha éppen semmi sincs az üzenetsorban. Az üzenetsor üres voltát a `PeekMessage()` visszatérési értékéből ismerhetjük fel. Ha ez az érték igaz, akkor egy üzenetet kaptunk, amit a szokásos módon a `TranslateMessage()` eljáráson átvezetünk, majd a `DispatchMessage()` segítségével az alkalmazás üzenetkezelő eljárásához juttatunk. Amikor a `PeekMessage()` visszatérési értéke hamis, az üzenetsor üres, tehát átadhatjuk magunkat az animáció örömeinek. Lekérdezzük a rendszeridőt, majd a tárolt korábbi idő alapján kiszámítjuk az utolsó animációs fázis óta eltelt keretidőt. Ezek után végrehajtatjuk az animáció egyetlen lépését.

Ezen a megoldáson kívül használhatnánk az Ms-Windows `SetTimer()` függvényét is, amellyel rábírhatjuk az Ms-Windows-t arra, hogy időnként `WM_TIMER` üzenetet küldjön a programnak. Két egymást követő üzenet között legalább a megadott idő eltelik, de az Ms-Windows nem garantálja, hogy ilyen időnként valóban kapunk is tőle üzenetet. Az ismertetett, üzenethurokban működő módszert nemcsak egyszerűbben használhatjuk, de az minden időt ki is használ a rajzolásra, és csak a rendszer terhelése miatt lassulhat le.

9.2. Dupla bufferelés

Az animáció alatt a képeket egymás után állítjuk elő és kihasználjuk, hogy a gyorsan levetített állóképsorozatot a szem mozgásként érzékeli.



9.2. ábra. *Dupla buffer rendszerek*

A különböző képszintézis eljárások a képet általában fokozatosan építik fel, amely alatt rövid időre olyan részletek is feltűnhetnek, amelyek egyáltalán nem látszhatnának. Ez észrevehető villogáshoz vezet. A probléma megoldásához két rasztertár szükséges. Egy adott pillanatban az egyiket megjelenítjük, a másikba pedig rajzolunk. Amikor a kép elkészült, az elektronsugár képvisszafutási ideje alatt a két rasztertár szerepet cserél.

Az OpenGL platformok maguk is adnak támogatást a *dupla bufferelés*hez. Tekintsük először a GLUT környezetet! Egyrészt az ablak megnyitásához kérnünk kell, hogy a GLUT két rasztertárat tartson fenn (vagy ossza az egyetlen rasztertárat kétfelé). Ennek érdekében a `glutInitDisplayMode()` függvénynek egy `GLUT_DOUBLE` kapcsolót is át kell adnunk, tehát a szokásos inicializálási lépés a következőképpen alakul:

```
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
```

Másrészt minden képszintézis fázis végén a rajzolási eredményeket fogadó és a monitort kiszolgáló rasztertár szerepet cserél, amelyhez a `glutSwapBuffers()` függvényt kell meghívnunk. A képszintézis lépés tehát a következő:

```
glClear(GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT);
// rajzol ...
glutSwapBuffers();
```

Az Ms-Windows WGL környezetében a globális `SwapBuffers()` függvénnyel válthatunk rasztertárat. A függvénynek szüksége van az OpenGL kontextusra, amelyet a `wglGetCurrentDC()` hívással kérdezhetünk le.

```
SwapBuffers(wglGetCurrentDC());
```

9.3. Valószerű mozgás feltételei

Az animáció célja *valószerű mozgás* létrehozása. A mozgás akkor valószerű, ha kielégíti a természet törvényeit, ugyanis mindennapjaink során ilyen mozgásokkal találkozunk.

A *dinamika alaptörvénye* (*Newton 2. törvénye*) szerint egy szabadon mozgó test sebessége állandó, ha pedig erő hat rá, akkor a *sebesség* megváltozása, a *gyorsulás* arányos az erővel és fordítottan arányos a test tömegével. Jelöljük a test tömegét m-mel az időben változó helyét (pozícióját) $r(t)$-vel, sebességét $v(t)$-vel, gyorsulását pedig $a(t)$-vel, és legyen a testre ható, akár időben változó erő $F(t)$! A sebesség a pozíció időegység szerinti változása, azaz deriváltja. A gyorsulás pedig a sebesség deriváltja, így a pozíció második deriváltja:

$$v(t) = \frac{dr(t)}{dt}, \quad a(t) = \frac{dv(t)}{dt} = \frac{d^2r(t)}{dt^2} = \frac{F(t)}{m}.$$

Vegyünk egy egydimenziós mozgást és tegyük fel például, hogy egy m tömegpont a $t = 0$ időpillanattól kezdve szabadon esik! A tömegpontra ható nehézségi erő $F = mg$, ahol $g \approx 10m/s^2$ a nehézségi gyorsulás, tehát a test gyorsulása $a = F/m = g$. A sebesség a gyorsulás integrálja, tehát, ha kezdetben a sebesség zérus volt, akkor a t időpillanatban $v(t) = \int_0^t a\, dt = gt$. Az út pedig a sebesség integrálja: $r(t) = \int_0^t v(t)\, dt = gt^2/2$.

Ezek a mennyiségek csak az egydimenziós mozgás esetén skalárok, általános esetben viszont vektorok, és a fenti összefüggések minden koordinátára külön-külön igazak. Vektor jelölésekkel a haladó mozgás törvényeit a következőképpen írhatjuk fel:

$$\vec{v}(t) = \frac{d\vec{r}(t)}{dt}, \quad \vec{a}(t) = \frac{d\vec{v}(t)}{dt} = \frac{d^2\vec{r}(t)}{dt^2} = \vec{a}(t) = \frac{\vec{F}(t)}{m}.$$

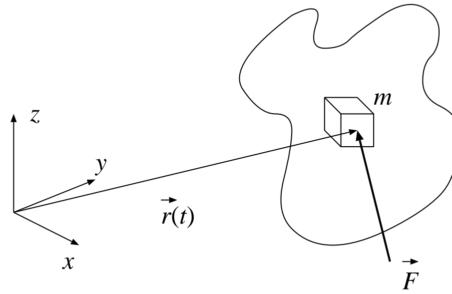
Ha a test nem pontszerű, hanem kiterjedt, akkor a haladó mozgáson kívül forgó mozgást is végezhet. Ebben az esetben a test minden pontja más és más pályát jár be, amelyet úgy tekinthetünk, hogy a test egésze haladó mozgást végez, miközben egy — akár időben változó — tengely körül forog.

A 3D grafikában egy pont világ-koordinátarendszerbeli $\vec{r}(t)$ helyét a pont $\vec{r}_L$ lokális koordinátáiból a modellezési transzformáció fejezi ki. A modellezési transzformáció akkor írható fel egyetlen mátrixszal, ha Descartes-koordinátákról homogén koordinátákra térünk át, azaz formálisan a szokásos három koordináta mellé egy negyedik, 1 értékű koordinátát is felveszünk:

$$[\vec{r}(t), 1] = [\vec{r}_L, 1] \cdot \mathbf{T_M}(t). \tag{9.1}$$

Ha a test nem deformálódik, a lokális koordinátái állandók, így a mozgásért kizárólag a modellezési transzformáció felelős.

Tegyük fel, hogy a pont kis környezetében a test tömege m, és erre a részre $\vec{F}$ erő hat (9.3. ábra)! Az erő származhat kívülről, vagy ugyanazon test többi részéből egyaránt.



9.3. ábra. *Egy m tömegű pont dinamikája*

A Newton-törvény miatt ezen pont pályája kielégíti a következő egyenletet:

$$\frac{d^2}{dt^2}[\vec{r}(t),1] = [\vec{r}_L,1] \cdot \frac{d^2\mathbf{T_M}(t)}{dt^2} = \left[\frac{\vec{F}}{m},0\right]. \tag{9.2}$$

Az erők valamilyen rugalmas mechanizmuson keresztül hatnak, így a gyorsulás nem változhat ugrásszerűen, következésképpen a mozgásvektor C^2 *folytonos* (3.3.6. fejezet). Gyakran azonban kevesebbel is beérjük. Ha a deformációtól eltekintünk, azaz *merev test*ekkel dolgozunk, akkor az erő és a gyorsulás is ugrásszerűen változhat, tehát előfordulhat, hogy a mozgásvektor bizonyos időpillanatokban csak C^1 *folytonos*. Sőt, két test ütközésekor élhetünk azzal a feltételezéssel, hogy az ütközés végtelenül kis idő alatt ment végbe, tehát akár a sebesség is megváltozhat ugrásszerűen, ami csak C^0 *folytonos* mozgásvektorhoz vezet (ez annak a feltételezésnek felelne meg, hogy ütközéskor végtelenül kicsiny idő alatt végtelenül nagy erők ébrednének). Összefoglalva, igaz ugyan, hogy a klasszikus fizika törvényei szerint a mozgásgörbék C^2 *folytonos*ak, de egyes időpillanatokban a derivált olyan nagy lehet, hogy ezekben az időpontokban jó közelítéssel alacsonyabb folytonossági osztályú függvény is használható.

Az animáció központi feladata olyan $\mathbf{T_M}(t)$ és $\mathbf{T_V}(t)$ mátrixok előállítása, amelyek egyrészt a felhasználó által elképzelt mozgást adják vissza, másrészt kielégítik a valószerű mozgás követelményeit.

A Newton-törvényeken kívül fennállnak az úgynevezett *megmaradási törvények*, amelyek szerint egy zárt mechanikai rendszer energiája, impulzusa és impulzusmomentuma állandó (ha a súrlódástól eltekintünk). Ezekre a fogalmakra a későbbiekben visszatérünk, most elég annyi, hogy ezen törvények miatt pattan vissza a biliárdgolyó az asztal oldalfaláról úgy, hogy a beesési szög megegyezik a visszaverődési szöggel, és az oldalfallal párhuzamos sebességkomponens változatlan marad, mialatt az oldalfalra merőleges komponens előjelet vált. Lám-lám egy régi ismerős. Az ideális tükörről a fény is éppen így verődik vissza.

9.4. ábra. *Karakteranimációt alkalmazó multimédiás oktatórendszer [13]*

Az élőlények (*karakter*ek) viselkedését a fizikai törvényeken felül még *fiziológiai törvények* is befolyásolják. Az ilyen rendszereket függetlenül mozgatható csontok alkotják, amelyeket *ízület*ek (más néven *csukló*k) kapcsolnak össze. A mozgás során az ízületek a csontokat összetartják, azok semmiképpen sem távolodhatnak el egymástól (ez ugyanis nagyon fájdalmas lenne). Az élőlények csontjainak hossza, legalábbis az animáció ideje alatt, általában nem változik (egy robotnál azonban akár ez is megtörténhet). A karakterek csontjai tehát egymáshoz képest csak az adott rendszer tulajdonságai szerint, korlátozottan mozoghatnak. Egy rendszerben a függetlenül megváltoztatható paraméterek számát *szabadságfok*nak nevezzük. A korlátok ellenére egy karakter szabadságfoka igen nagy, ezért nagyon változatosan mozoghat. Nagyon sokféleképpen rakosgathatjuk a lábainkat egymás elé, így elvileg nagyon különbözően járhatnánk. Mégis az emberek döntő többsége, hacsak valamilyen betegség ebben meg nem akadályozza, hasonlóan jár. Ez azt jelenti, hogy még a fiziológiai korlátokon felül is léteznek törvények, amelyek testünket irányítják. A legfontosabb ilyen törvény a *lustaság törvénye*, amely szerint az élőlények egy mozgásfázist rendszerint úgy hajtanak végre, hogy közben minimális energiát használnak fel. Ez nem egy természeti törvény miatt van így, hanem hosszas tanulás eredménye, ugyanis így lehetünk adott körülmények között a leggyorsabbak, legkitartóbbak, azaz így lehet a legnagyobb esélyünk a túlélésre.

9.4. Pozíció–orientáció mátrixok interpolációja

A mozgás a transzformációs mátrixok elemeinek időbeli változtatását jelenti. Az animátor az elemek időfüggvényeit a geometriai tervezésnél megismert interpolációs eljárásokkal adhatja meg. Mint azt a 3.2.7. fejezetben láttuk, tetszőleges pozíció, illetve ori-

entáció megadható a következő mátrixszal:

$$\begin{bmatrix} & 0 \\ \mathbf{A}_{3\times3} & 0 \\ & 0 \\ \vec{q} & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ q_x & q_y & q_z & 1 \end{bmatrix}.$$

A $\vec{q}$ vektor a pozíciót, az $\mathbf{A}_{3\times3}$ pedig az orientációt határozza meg. A $\vec{q}$ vektor elemeit egymástól függetlenül vezérelhetjük, az $a_{11},\ldots,a_{33}$ elemeket viszont nem, hiszen azok összefüggőek. A függés abból is látszik, hogy az orientáció szabadságfoka 3, a mátrixelemek száma pedig 9. Egy érvényes orientációs mátrix nem módosíthatja az objektum alakját, amelynek elégséges feltétele, hogy a mátrix sorvektorai egymásra merőleges egységvektorok legyenek.

Az interpoláció során a pozícióvektor elemeit függetlenül interpolálhatjuk, az orientációmátrix elemeit azonban nem, hiszen a független változtatás nem érvényes orientációkat is eredményezhetne (azaz a test eltorzulna). A megoldást a független orientációs paraméterek terében végrehajtott interpoláció jelenti. Például használhatjuk az orientáció jellemzésére a *csavaró–billentő–forduló* szögeket, amelyek egy orientációhoz úgy visznek el, hogy először a z tengely körül α szöggel, majd a megváltozott y tengely körül β szöggel, végül a két korábbi forgatás utáni x tengely körül γ-szöggel forgatnak. Összefoglalva a mozgás függetlenül vezérelhető paraméterei:

$$\mathbf{p}(t) = [x(t), y(t), z(t), \alpha(t), \beta(t), \gamma(t)]. \tag{9.3}$$

A képszintézis során a modellezési transzformációra van szükségünk, amit az interpolált *paraméter vektor*ból számíthatunk ki:

$$\mathbf{A} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & \sin\gamma \\ 0 & -\sin\gamma & \cos\gamma \end{bmatrix},$$

$$\vec{q} = [x, y, z].$$

A *valószerű mozgás* biztosításához az $\mathbf{A}$ mátrix és a $\vec{q}$ vektor elemeinek folytonos görbéknek kell lenniük. Ezt a $\mathbf{p}(t)$ paraméter vektor elemeinek C^2, C^1, C^0 folytonos interpolációjával vagy approximációjával teljesíthetjük.

A *kamera animáció* kissé bonyolultabb, mint az objektumok mozgatása, mert a kamerához több paraméter tartozik, mint a pozíció és az orientáció. Emlékezzünk vissza, hogy a kamerát általában a következő folytonos paraméterekkel jellemezzük:

1. $\vec{eye}$: a szempozíció,
2. $\vec{lookat}$: a pont amely felé a szem néz,

3. $\vec{up}$: az ablak függőleges iránya,

4. fov: a függőleges látószög,

5. $aspect$: az ablak magasságának és szélességének az aránya,

6. f_p, b_p: az első és hátsó vágósíkok.

Ezen paraméterek egymástól függetlenül vezérelhetők, így a kamera paramétervektora:

$$\mathbf{p_{cam}}(t) = [\vec{eye}, \vec{lookat}, \vec{up}, aspect, fov, f_p, b_p]. \tag{9.4}$$

Egy t időpillanatra a paramétervektor aktuális értékéből számíthatjuk a $\mathbf{T_V}$ nézeti transzformációs mátrixot (lásd a 7. fejezetet).

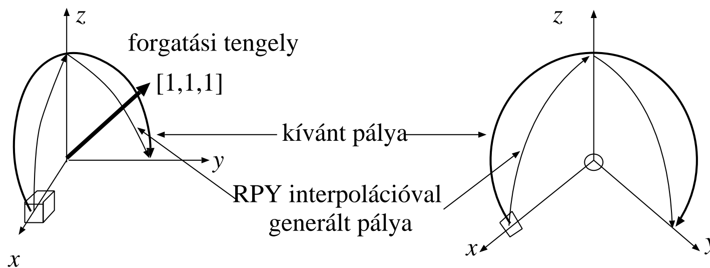
9.5. Az orientáció jellemzése kvaternióval

Az előző fejezetben megállapítottuk, hogy az animáció során a szabadon vezérelhető paramétereket kell tetszőleges időpillanatra kiszámítani. A paraméterekből pedig a transzformációs mátrixok előállíthatók, amelyek ténylegesen „mozgatják” a tárgyakat. A kétlépéses módszer biztosítja, hogy a transzformációk tetszőleges időpillanatra is érvényesek legyenek, azaz csak eltolják és forgassák a merev testeket, de semmiképpen se tegyék tönkre azok alakját.

A mozgás-paramétertérben végrehajtott számítás azonban olyan újabb problémákat vet fel, amelyek mellett nem mehetünk el. Tegyük fel például, hogy egy objektumot két kulcspozícióban felvett pozíció és orientáció között egyenletes sebességgel kell átvinni. A mozgásparaméterek terében a két paraméterkészlet között a legrövidebb úton, egy egyenes mentén mehetünk át. Sajnos, a paramétertérben talált szakasz csak a pozíciókra felel meg a valódi, „természetes” legrövidebb útnak, az orientációkra nem.

A probléma az orientációs paraméterek megválasztásából ered. A *csavaró–billentő–forduló* szögek (*RPY* szögek) ugyanis három mesterséges koordinátatengely körül forgatnak, amelyek csak a mi képzeletünkben léteznek. A valódi testek egyetlen, akár időben változó tengely körül forognak. Mivel a transzformációs mátrix a csavaró–billentő–forduló szögektől nemlineáris módon függ, a képzeletbeli koordinátatengelyek láthatóvá válnak a mozgásban. Például, ha egy forgástengely mentén egyenletes sebességgel egy α szögig szeretnénk forgatni, a csavaró–billentő–forduló szögek a mozgás egy közbenső fázisában nem feltétlenül felelnek meg az elvárt arányos résznek. Ez egyenetlen, nem természetes mozgást eredményez. Annak érdekében, hogy még mélyebben átérezzük ennek szörnyűségét, tegyük fel, hogy egy az [1,0,0] pontban elhelyezkedő tárgyat az [1,1,1] vektor körül 240 fokkal szeretnénk az óramutató járásával megegyező irányban elfordítani! A mozgást három kulcspozícióval adjuk meg, amikor

a tárgyat 0 fokkal, 120 fokkal és végül 240 fokkal forgattuk el (9.5. ábra). A 120 fokos elforgatás az x tengelyt a z tengelybe viszi át, a 240 fokos elforgatás pedig az x tengelyt az y tengelybe. Ha a csavaró–billentő–forduló szögeket használjuk, ezeket a transzformációkat nyilván 90 fokos elforgatásokkal kapjuk meg, mégpedig először az y tengely körül, majd az x tengely körül. Tehát a csavaró–billentő–forduló szögek alkalmazása arra kényszeríti a testünket, hogy először az y tengely körül, majd az x tengely körül 90–90 fokot forduljon, ahelyett, hogy egyenletesen az [1,1,1] tengely körül forogna.



9.5. ábra. *A csavaró–billentő–forduló szögek interpolációjából eredő problémák*

A csavaró–billentő–forduló szögek független interpolációja tehát gyakran nem megfelelő. A kellemetlen hatások persze a kulcspontok számának növelésével csökkenthetők, de ez viszont az animátorok rosszallását váltaná ki. Más megoldás után kell néznünk, és az orientációváltozást úgy kell kezelnünk, mintha egyetlen tengely körüli forgatásról lenne szó. A tengely szükség szerint változhat az időben. A forgatási tengely sajnos nem állítható elő sem a csavaró–billentő–forduló szögekből sem pedig a transzformációs mátrixból, hiszen azok csak egy pillanatnyi állapotot jelentenek, a tengely pedig egy időpillanat környékén zajló folyamathoz kapcsolódik. Térjünk vissza az alapokhoz, és adjuk meg az orientációt egy alapvetően különböző módon! Az ehhez szükséges eszközt *kvaternió*nak (*quaternion*) nevezik.

Akár egy mátrix, egy $\mathbf{q}$ kvaternió is arra szolgál, hogy egy $\vec{u}$ vektort egy másik $\vec{v}$ vektorba vigyen át:

$$\vec{u} \stackrel{\mathbf{q}}{\Longrightarrow} \vec{v}. \tag{9.5}$$

A mátrixok ezt a műveletet elég nagy redundanciával végzik el, azaz végtelen sok olyan mátrix található, amely egy vektort egy másikba transzformál át. Háromdimenziós esetben a mátrixok 9 eleműek, holott 4 skalár bőven elegendő volna ezen leképzés megfogalmazására. Ez a 4 elem a vektor hosszát megváltoztató skálázás, a forgatás síkját leíró két adat (két adott tengellyel bezárt szög, vagy a normálvektor iránya), és a forgatás szöge.

Egy $\mathbf{q}$ kvaternió tehát éppen 4 elemet tartalmaz. Az elemeket célszerű egy skalárra

és egy 3D vektorra bontani:

$$\mathbf{q} = [s,x,y,z] = [s,\vec{w}].$$

A kvaterniók tehát négyesek, ami indokolja a nevüket (a zenerajongóknak a kvartett juthat eszébe). A kvaterniókra a vektorokhoz hasonlóan definiálhatunk *összeadás*, *számmal szorzás*, *skaláris szorzás*, *abszolút érték* műveleteket:

$$\mathbf{q}_1 + \mathbf{q}_2 = [s_1,\vec{w}_1] + [s_2,\vec{w}_2] = [s_1 + s_2, \vec{w}_1 + \vec{w}_2],$$

$$\lambda\mathbf{q} = \lambda[s,\vec{w}] = [\lambda s, \lambda\vec{w}],$$

$$\langle \mathbf{q}_1, \mathbf{q}_2 \rangle = \langle [s_1,x_1,y_1,z_1],[s_2,x_2,y_2,z_2] \rangle = s_1 s_2 + x_1 x_2 + y_1 y_2 + z_1 z_2,$$

$$|\mathbf{q}| = |[s,x,y,z]| = \sqrt{\langle \mathbf{q},\mathbf{q} \rangle} = \sqrt{s^2 + x^2 + y^2 + z^2}.$$

A kvaterniókat úgy is elképzelhetjük, mint a komplex számok négydimenziós általánosítását, ahol s a valós rész, x,y,z pedig a három képzetes vagy más néven imaginárius rész. A három imaginárius egységet (a normál komplex számnál csak egy ilyen volna, a $\sqrt{-1}$) jelöljük **i**, **j** és **k**-val, így az általánosított komplex szám:

$$\mathbf{q} = s + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}.$$

Amikor Sir Hamilton feltalálta a kvaterniókat, éppen a komplex számok általánosítása hajtotta. Először azzal próbálkozott, hogy a valós rész mellé két imaginárius tengelyt vegyen fel, úgy, hogy a szokásos műveletek, mint az összeadás és szorzás a szokásos tulajdonságokkal definiálhatók legyenek. Legnagyobb igyekezete ellenére kudarcot vallott, mert nem tudta az általánosítást egy valós és két imaginárius részre elvégezni. Egy hideg téli estén egy hídon szomorúan sétálgatva azonban rájött, hogy a feladat egy valós és három imaginárius részre viszont már megoldható. Gondolatát fel is írta a híd pillérére, a világ pedig gazdagabb lett a kvaterniók fogalmával.

Az imaginárius tengelyek közötti összefüggést a következőképpen fogalmazta meg:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1, \;\; \mathbf{ji} = -\mathbf{k}, \;\; \mathbf{kj} = -\mathbf{i}, \;\; \mathbf{ik} = -\mathbf{j}.$$

Ezzel, a komplex számokhoz hasonlóan a kvaterniók szorzása is definiálható.

A történelmi távlatokból visszatérve, mi egy másik, ekvivalens megközelítést ismertetünk, és a *szorzás*t a szokásos 3D vektorműveletek segítségével adjuk meg:

$$\mathbf{q}_1 \cdot \mathbf{q}_2 = [s_1,\vec{w}_1] \cdot [s_2,\vec{w}_2] = [s_1 s_2 - \vec{w}_1 \cdot \vec{w}_2, \; s_1\vec{w}_2 + s_2\vec{w}_1 + \vec{w}_1 \times \vec{w}_2]. \tag{9.6}$$

A kvaternió szorzás és összeadás disztributív ($\mathbf{a}\cdot(\mathbf{b}+\mathbf{c}) = \mathbf{a}\cdot\mathbf{b} + \mathbf{a}\cdot\mathbf{c}$), az összeadás kommutatív ($\mathbf{a}+\mathbf{b} = \mathbf{b}+\mathbf{a}$) és asszociatív ($(\mathbf{a}+\mathbf{b})+\mathbf{c} = \mathbf{a}+(\mathbf{b}+\mathbf{c})$), a szorzás pedig

asszociatív, de nem kommutatív (aki nem hiszi, helyettesítsen be a műveletek definíciójába, és máris igazolhatja ezeket a kijelentéseket). A szorzás egysége az $[1,\vec{0}]$ *egységkvaternió*, azaz ha egy kvaterniót ezzel az egységgel szorzunk, akkor az eredeti kvaterniót kapjuk eredményül. Az egység segítségével az *inverz* is bevezethető, mégpedig úgy, hogy egy $\mathbf{q}$ kvaternió inverze az a $\mathbf{q}^{-1}$ kvaternió, amivel összeszorozva az egységkvaterniót kapjuk:

$$\mathbf{q}\cdot\mathbf{q}^{-1} = \mathbf{q}^{-1}\cdot\mathbf{q} = [1,\vec{0}] \quad\Longrightarrow\quad \mathbf{q}^{-1} = \frac{[s,-\vec{w}]}{|\mathbf{q}|^2}.$$

Ezt be is bizonyítjuk, tehát megmutatjuk, hogy egy kvaternió és az így definiált inverz szorzata valóban az egységkvaternió:

$$[s,\vec{w}]\cdot\frac{[s,-\vec{w}]}{|\mathbf{q}|^2} = \frac{[s^2+|\vec{w}|^2,\vec{0}]}{s^2+|\vec{w}|^2} = [1,\vec{0}].$$

Az is könnyen bebizonyítható, hogy a mátrixokhoz hasonlóan egy szorzat inverzében a résztvevő tényezők sorrendje megfordul:

$$(\mathbf{q}_2\cdot\mathbf{q}_1)^{-1} = \mathbf{q}_1^{-1}\cdot\mathbf{q}_2^{-1}.$$

Végre visszatérhetünk az eredeti célunkhoz, a 3D forgatások megvalósításához. Egy 3D forgatás ugyanis kvaternószorzásokkal megvalósítható. Ahhoz persze, hogy egy vektort kvaternióval szorozhassunk a háromelemű 3D vektorból is kvaterniót, azaz négyest kell csinálnunk. Egészítsük ki tehát a vektor három elemét egy negyedik $s = 0$ elemmel! Az $\vec{u}$ vektort egy $\vec{v}$ vektorba úgy forgatjuk, hogy a $[0,\vec{u}]$ kvaterniót egy $\mathbf{q}$ kvaternióval balról, majd a $\mathbf{q}$ kvaternió inverzével jobbról szorozzuk:

$$\vec{u}\overset{\mathbf{q}}{\Longrightarrow}\vec{v}: \qquad [0,\vec{v}] = \mathbf{q}\cdot[0,\vec{u}]\cdot\mathbf{q}^{-1} = \frac{[0,\ s^2\vec{u}+2s(\vec{w}\times\vec{u})+(\vec{w}\cdot\vec{u})\vec{w}+\vec{w}\times(\vec{w}\times\vec{u})]}{|\mathbf{q}|^2}. \tag{9.7}$$

Mindenekelőtt vegyük észre, hogy a $\mathbf{q} = [s,\vec{w}]$ kvaternió skálázása egyáltalán nem módosítja a művelet eredményét! Ugyanis, ha az $[s,\vec{w}]$ kvaterniót egy skalárral szorozzuk, a $\mathbf{q}^{-1} = [s,-\vec{w}]/|\mathbf{q}|^2$ inverze éppen ennek a reciprokával lesz hosszabb, tehát a két kvaternió a skálázás hatását kioltja. A továbbiakban ezért az általánosság korlátozása nélkül feltételezhetjük, hogy a forgatáshoz használt kvaterniók mind *egység hosszú kvaternió*k:

$$|\mathbf{q}|^2 = s^2+|\vec{w}|^2 = 1.$$

Az egység hosszú kvaterniókra a 9.7. egyenletet a következő formában is felírhatjuk:

$$[0,\vec{v}] = \mathbf{q}\cdot[0,\vec{u}]\cdot\mathbf{q}^{-1} = [0,\vec{u}+2s(\vec{w}\times\vec{u})+2\vec{w}\times(\vec{w}\times\vec{u})], \tag{9.8}$$

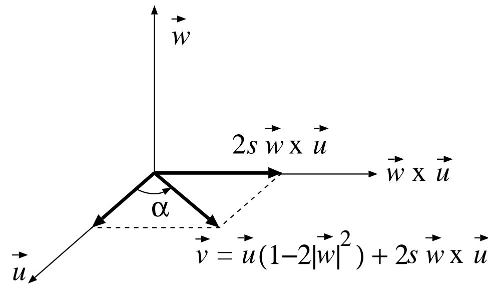
mivel

$$s^2\vec{u} = \vec{u} - |\vec{w}|^2\vec{u} \quad \text{és} \quad (\vec{w}\cdot\vec{u})\vec{w} - |\vec{w}|^2\vec{u} = \vec{w}\times(\vec{w}\times\vec{u}).$$

Most bizonyítsuk be, hogy a fenti kvaterniószorzások valóban a 3D forgatást írják le! A bizonyítást két speciális esetre végezzük el, amelyekből már következik az állítás bármely további esetre is. Először feltesszük, hogy az $\vec{u}$ és $\vec{w}$ vektorok egymásra merőlegesek. A második esetben a két vektort párhuzamosnak fogjuk tekinteni.

Ha az $\vec{u}$ vektor a kvaternióban szereplő $\vec{w}$ vektorra merőleges, akkor az egység hosszú kvaterniókra érvényes 9.8. egyenletből a kifejtési tétel felhasználásával a következőt kapjuk:

$$\mathbf{q}\cdot[0,\vec{u}]\cdot\mathbf{q}^{-1} = [0,\ \vec{u}(1-2|\vec{w}|^2) + 2s(\vec{w}\times\vec{u})] = [0,\vec{v}].$$



9.6. ábra. *Forgatás, amikor az $\vec{u}$ merőleges a kvaternió $\vec{w}$ vektor részére*

Ezek szerint a $\vec{v}$ eredményvektor az egymásra merőleges $\vec{u}$ és $\vec{w}\times\vec{u}$ vektorok lineáris kombinációja (9.6. ábra), tehát az eredményvektor az $\vec{u}$ és $\vec{w}\times\vec{u}$ vektorok által kifeszített síkban van. Mivel a feltételezésünk szerint az $\vec{u}$ vektor merőleges a $\vec{w}$ vektorra, a $\vec{w}\times\vec{u}$ vektor pedig a vektoriális szorzat tulajdonságai miatt merőleges a $\vec{w}$ vektorra, a két vektor lineáris kombinációi, így a $\vec{v}$ is, szükségképpen merőlegesek a $\vec{w}$ vektorra.

Most nézzük a $\vec{v}$ vektor hosszát:

$$|\vec{v}| = |\vec{u}|\sqrt{(1-2|\vec{w}|^2)^2 + (2s|\vec{w}|)^2} = |\vec{u}|\sqrt{(1+4|\vec{w}|^2(s^2+|\vec{w}|^2-1)} = |\vec{u}|.$$

Azt kaptuk tehát, hogy a $\vec{v}$ eredményvektor a $\vec{w}$-re merőleges és az $\vec{u}$ vektort is tartalmazó síkban van, hossza pedig megegyezik az $\vec{u}$ hosszával. Az ilyen műveletet pedig $\vec{w}$ körüli forgatásnak nevezzük.

A forgatás szögét abból az összefüggésből kapjuk meg, hogy két vektor skaláris szorzata egyenlő a vektorok abszolút értékeinek és a bezárt szögük koszinuszának a szorzatával. Ebből viszont az $\vec{u}$ és $\vec{v}$ szöge (α) kifejezhető:

$$\cos\alpha = \frac{\vec{u}\cdot\vec{v}}{|\vec{u}|\cdot|\vec{v}|} = \frac{(\vec{u}\cdot\vec{u})(1-2|\vec{w}|^2) + 2s\vec{u}\cdot(\vec{w}\times\vec{u})}{|\vec{u}|^2} = 1-2|\vec{w}|^2. \tag{9.9}$$

Ezzel az állítást arra az esetre, amikor az $\vec{u}$ vektor a $\vec{w}$ vektorra merőleges, bebizonyítottuk.

Ha az $\vec{u}$ vektor párhuzamos a $\vec{w}$ vektorral, akkor az egység hosszú kvaterniókra felírt 9.8. egyenlet a következő alakot ölti:

$$[0, \vec{v}] = \mathbf{q} \cdot [0, \vec{u}] \cdot \mathbf{q}^{-1} = [0, \vec{u}],$$

azaz a művelet nem változtatja meg a vektort. Ez is tökéletesen rendben van, hiszen a forgatás a forgatási tengellyel párhuzamos vektorokat változatlanul hagyja, tehát a második speciális esetet is bebizonyítottuk.

Az általános vektorokat viszont mindig felbonthatjuk egy $\vec{w}$ vektorral párhuzamos, valamint egy arra merőleges komponensre. A kvaternió szorzás és összeadás disztributivitása miatt a két komponensre külön-külön végezhetjük el a forgatást megvalósító műveletet. Mint beláttuk, a párhuzamos tag nem változik, a merőleges pedig α szöggel elfordul, ahol a szög koszinuszát a $\cos\alpha = 1 - 2|\vec{w}|^2$ egyenletből kapjuk meg. Ez azt jelenti, hogy az eredmény valóban az eredeti $\vec{u}$ vektornak a kvaternió vektorkomponense körüli α szögű elforgatottja.

Megállapítottuk tehát, hogy egy kvaternióval balról, majd az inverzével jobbról végrehajtott szorzás egy vektort elforgat. Fordítsuk meg gondolatmenetünket, és keressük meg azt a kvaterniót, amely egy adott $\vec{d}$ forgástengely körüli α szöggel történő elforgatást jelent! Beláttuk, hogy a kvaternió a vektor része körül forgat, tehát azon egység hosszú kvaternió, amely egy $\vec{d}$ egységvektor körül forgat, a következő alakú:

$$\mathbf{q} = [s, r \cdot \vec{d}], \qquad s^2 + r^2 = 1. \tag{9.10}$$

Az s és r paramétereket abból a feltételből határozhatjuk meg, hogy a kvaterniónak éppen α szöggel kell forgatnia. A 9.9. és 9.10. egyenletek felhasználásával azt kapjuk, hogy:

$$\cos\alpha = 1 - 2r^2, \qquad s = \sqrt{1 - r^2}.$$

Ezekből az egyenletekből némi trigonometriai ügyeskedés után kifejezhetjük az ismeretlen paramétereket. Összefoglalva, a $\vec{d}$ egységvektor körül α szöggel forgató kvaternió:

$$\mathbf{q} = [\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2} \cdot \vec{d}]. \tag{9.11}$$

A transzformációs mátrixokhoz hasonlóan, a kvaterniók is konkatenálhatók, azaz több egymás utáni forgatás egyetlen kvaternióval írható le:

$$\mathbf{q}_2 \cdot (\mathbf{q}_1 \cdot [0, \vec{u}] \cdot \mathbf{q}_1^{-1}) \cdot \mathbf{q}_2^{-1} = (\mathbf{q}_2 \cdot \mathbf{q}_1) \cdot [0, \vec{u}] \cdot (\mathbf{q}_2 \cdot \mathbf{q}_1)^{-1}.$$

A kvaterniókkal tehát kifejezhetjük az orientációt, és a kvaternióműveletekkel követhetjük annak változásait. A test megjelenítése során a megfelelő orientáció beállításához

a kvaternióból, azaz a forgatási tengelyből és az elfordulás szögéből, elő kell állítani a transzformációs mátrixot, pontosabban a 4×4-es transzformációs mátrixnak az elforgatásért felelős bal felső 3×3-as minormátrixát (a negyedik sorban az eltolás van, amihez a kvaternióknak semmi köze sincs, a negyedik oszlopba pedig csak a perspektív transzformáció tesz $[0,0,0,1]$-től eltérő számokat). A transzformációs mátrix előállításához azt kell megvizsgálni, hogy a koordinátarendszer egységvektoraival mi történik, ha a kvaternió segítségével elforgatjuk őket. Alkalmazva a $\mathbf{q} = [s,x,y,z]$ kvaterniót az [1,0,0], [0,1,0] és [0,0,1] bázisvektorokra a 9.8. egyenlet szerint, a 3×3-as transzformációs mátrix első, második és harmadik sorát kaphatjuk meg:

$$\mathbf{A}_{3\times 3} = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2sz & 2xz-2sy \\ 2xy-2sz & 1-2x^2-2z^2 & 2yz+2sx \\ 2xz+2sy & 2yz-2sx & 1-2x^2-2y^2 \end{bmatrix}. \tag{9.12}$$

Az OpenGL `glRotatef()` függvényével a mátrixot egyszerűbben is felépíthetjük. Ez a függvény ugyanis egy tetszőleges tengely körül forgat. A kvaternió vektor részéből megkaphatjuk a forgatási tengelyt, a skalár részéből a forgatási szög felének a koszinuszát, a vektor részének az abszolút értékéből pedig a forgatási szög szinuszát. A C könyvtár `atan2()` függvénye ebből már kiszámítja a szöget radiánban, amit még gondosan fokokra kell váltani, hiszen a `glRotatef()` így követeli meg.

A kvaterniót forgatási mátrixszá alakító eljárás megfordítható, azaz egy forgatási mátrixhoz előállíthatjuk a neki megfelelő kvaterniót. Az $[s,x,y,z]$ kvaternió elemeket a 9.12. egyenletből kifejezve azt kapjuk, hogy:

$$s = \frac{1}{2}\sqrt{a_{11}+a_{22}+a_{33}+1}, \quad x = \frac{a_{23}-a_{32}}{4s}, \quad y = \frac{a_{31}-a_{13}}{4s}, \quad z = \frac{a_{12}-a_{21}}{4s}.$$

Az orientáció csavaró–billentő–forduló (α,β,γ) szögeit szintén kvaternióra válthatjuk, csupán az elemi elfordulásokat kell egymással kombinálni:

$$\mathbf{q}(\alpha,\beta,\gamma) = [\cos\frac{\alpha}{2},(0,0,\sin\frac{\alpha}{2})]\cdot[\cos\frac{\beta}{2},(0,\sin\frac{\beta}{2},0)]\cdot[\cos\frac{\gamma}{2},(\sin\frac{\gamma}{2},0,0)].$$

A következőkben egy *kvaternió osztály*t mutatunk be:

```
//===============================================================
class Quaternion {         // kvaternió
//===============================================================
   float s;    // a ,,valós rész'' = cos(alpha/2)
   Vector d;   // imaginárius rész, a forgatás tengelye
public:
   Quaternion(float m[3][3]) { // mátrixból kvaternió
      s = sqrt(m[0][0] + m[1][1] + m[2][2] + 1) / 2;
      d = Vector(m[1][2]-m[2][1], m[2][0]-m[0][2], m[0][1]-m[1][0]) / (4*s);
   }
   Quaternion operator+(Quaternion& q) { // kvaternió összeadás
```

```
      return Quaternion(s + q.s, d + q.d);
   }
   Quaternion operator*(float f) {        // számmal szorzás
      return Quaternion(s * f, d * f);
   }
   float operator*(Quaternion& q) {       // skaláris szorzás
      return (s * q.s + d * q.d);
   }
   void Normalize() {                     // egység hosszúvá változtatás
      float length = sqrt(s * s + d.x * d.x + d.y * d.y + d.z * d.z);
      (*this) = (*this) * (1/length);
   }
   Quaternion operator%(Quaternion& q) { // kvaternió szorzás
      return Quaternion(s * q.s - d * q.d, q.d * s + d * q.s + d % q.d);
   }

   void GetMatrix(float m[3][3]) {        // kvaternióból mátrix
      m[0][0] = 1 - 2 * d.y * d.y - 2 * d.z * d.z;
      m[0][1] = 2 * d.x * d.y + 2 * s * d.z;
      m[0][2] = 2 * d.x * d.z - 2 * s * d.y;
      m[1][0] = 2 * d.x * d.y - 2 * s * d.z;
      m[1][1] = 1 - 2 * d.x * d.x - 2 * d.z * d.z;
      m[1][2] = 2 * d.y * d.z + 2 * s * d.x;
      m[2][0] = 2 * d.x * d.z + 2 * s * d.y;
      m[2][1] = 2 * d.y * d.z - 2 * s * d.x;
      m[2][2] = 1 - 2 * d.x * d.x - 2 * d.y * d.y;
    }

    float GetRotationAngle() {          // forgatási szög
        float cosa2 = s, sina2 = d.Length();
        float angRad = atan2(sina2, cosa2) * 2;
        return angRad * 180 / M_PI;
    }
    Vector& GetAxis() { return d; }     // forgatási tengely
};
```

9.5.1. Interpoláció kvaterniókkal

Vizsgáljuk meg, hogy miként alkalmazhatók a kvaterniók két orientáció közötti átmenet kialakítására! A kezdeti és a célorientációt a $\mathbf{q}_1$ és a $\mathbf{q}_2$ egység hosszú kvaterniókkal adjuk meg. Először az egyszerűség kedvéért feltételezzük, hogy a két kvaternió ugyanazon $\vec{d}$ egységvektor körül forgat el, azaz:

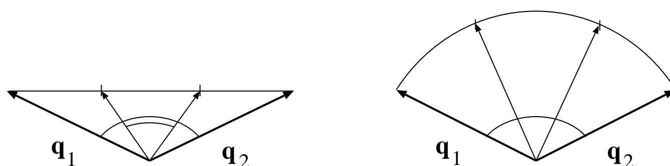
$$\mathbf{q}_1 = [\cos\frac{\alpha_1}{2}, \sin\frac{\alpha_1}{2}\cdot\vec{d}], \qquad \mathbf{q}_2 = [\cos\frac{\alpha_2}{2}, \sin\frac{\alpha_2}{2}\cdot\vec{d}]. \tag{9.13}$$

Ha kiszámítjuk a $\mathbf{q}_1$ és a $\mathbf{q}_2$ kvaternió skaláris szorzatát,

$$\langle \mathbf{q}_1, \mathbf{q}_2 \rangle = \cos\frac{\alpha_1}{2}\cdot\cos\frac{\alpha_2}{2} + \sin\frac{\alpha_1}{2}\cdot\sin\frac{\alpha_2}{2} = \cos\frac{\alpha_2-\alpha_1}{2},$$

akkor arra az érdekes következtetésre juthatunk, hogy a kvaterniók által képviselt orientációk közötti szög éppen kétszerese a két kvaternió által a 4D térben közrezárt szögnek.

A kvaternió, mint négyes tekinthető egy 4D vektornak, ahol a bezárt szög koszinusza két egység hosszú kvaternió skaláris szorzata.

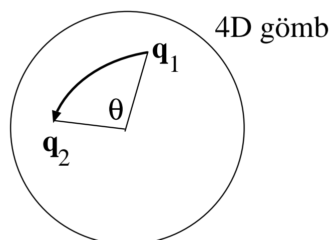


9.7. ábra. *Lineáris és gömbi interpoláció összehasonlítása*

A kitűzött célunk az, hogy egy objektumot egyenletes mozgással egy orientációból egy másikba vigyünk át. Ha a két kvaternió között lineáris interpolációt alkalmaznánk, akkor az egyes interpolált kvaterniók szöge, és így az elemi elfordulási szögek sem lennének állandók (9.7. ábra). Az objektumunk forgatása tehát nem lenne állandó sebességű, hanem a 9.7. ábra szerint gyorsulással indulna és lassulással érne véget (figyeljük meg, hogy a 9.7. ábra bal oldalán a szakasz kezdetén és végén egy egységnyi szakaszdarabnak kisebb szög felel meg, mint a szakasz közepén). A lineáris interpoláció helyett tehát mást kell kitalálnunk, ami az interpolált kvaterniók közötti szöget állandó értéken tartja (9.7. ábra jobb oldala). A *gömbi interpoláció* nyilván teljesíti ezt a feltételt, amikor az interpolált kvaterniókat a 4D gömb $\mathbf{q}_1$ és $\mathbf{q}_2$ pontjai közötti ívről választjuk. Ha $\mathbf{q}_1$ és $\mathbf{q}_2$ egység hosszú kvaterniók, az interpolált kvaternió is egység hosszú kvaternió lesz. A gömbfelületen végrehajtott interpoláció (*spherical linear interpolation* vagy *SLERP*) a következő összefüggéssel írható le [123]:

$$\mathbf{q}(t) = \frac{\sin(1-t)\theta}{\sin\theta} \cdot \mathbf{q}_1 + \frac{\sin t\theta}{\sin\theta} \cdot \mathbf{q}_2, \tag{9.14}$$

ahol $\cos\theta = \langle \mathbf{q}_1, \mathbf{q}_2 \rangle$, a t paraméter pedig a $[0, 1]$ intervallumon fut végig (9.8. ábra).



9.8. ábra. *Kvaternió interpoláció a 4D egységgömbön*

9.6. A mozgásgörbék megadási lehetőségei

Az animáció felvétele az egyes objektumok és a kamera paramétergörbéinek a megadását jelenti. A feladat megoldása egyrészt a szabadformájú görbéknél megismert módszerekkel lehetséges, másrészt speciális, a mozgás tulajdonságait is kihasználó eljárások is bevethetők. A legfontosabb lehetőségek az alábbiak:

- *Spline animáció*: az egyes paraméter–idő függvényeket 2D görbékkel adjuk meg, amelyeket a folytonossági és a lokális vezérelhetőségi igények miatt általában spline-ok segítségével definiálunk.

- *Képletanimáció* (*script animation*): a paraméter–idő függvényeket közvetlenül az algebrai alakjukkal adjuk meg. Például, az x tengely mentén, az origóhoz egy rugóval hozzáerősített, és a $t = 0$-ban az origóból induló, rezgő tömegpont x koordinátája az $x(t) = A \cdot \sin(\omega t)$ képlettel írható le. Itt az A a rezgés amplitúdója, azaz maximális kiterjedése, $\omega = 2\pi f$ pedig a rezgés körfrekvenciája, azaz az f frekvencia 2π-szerese. Egy másik példa lehet a $t = 0$-ban az origóból (v_x, v_y) sebességgel kilőtt lövedék pályája, amelyet az

$$x(t) = v_x \cdot t, \quad y(t) = v_y \cdot t - g \cdot \frac{t^2}{2}$$

  képletekkel adhatunk meg, amelyben g a nehézségi gyorsulás.

- *Kulcskeret animáció* (*keyframe animation*): A felhasználó a mozgás során bejárt pozíciókat és orientációkat csak néhány „kulcspontban” definiálja, amelyből a program a többi időpillanat mozgásparamétereit interpolációs vagy approximációs technikákkal határozza meg. Az approximációs vagy interpolációs eljárások során általában spline-okat használunk, így ez a megközelítés a spline animációval rokon. A kulcskeret animáció népszerűségét szemléletes tartalmának köszönheti. A mindennapi életünk során is így írjuk le a mozgásokat: „Móricka belépett az ajtón, majd elment a szoba sarkába, megfordult és elsápadt.” Ez kulcskeret „nyelven” úgy fogalmazható meg, hogy Móricka t_0 időpillanatban az ajtóban volt piros arccal, t_1-ben a sarokban még mindig piros arccal a sarok felé nézett, t_2-ben a sarokban háttal továbbra is piros arccal állt, t_3-ben pedig a sarokban háttal állt, de már falfehér arccal. A kijelölt időpillanatok közötti történések (a sarokba megy, lassan megfordul, fokozatosan elsápad) ezekből levezethetők, azaz interpolálhatók.

- *Pálya animáció* (*path animation*): A mozgást most egyetlen 3D görbével adjuk meg, és elvárjuk, hogy a kiválasztott test a görbe mentén haladjon végig. Ez első hallásra kevésnek tűnik, ugyanis a pozíció és az orientáció hat szabadságfokából

látszólag csak hármat, a pozíciót adtuk meg. Gondoljunk azonban egy madárra, repülőre, vagy akár egy autóra, amelyek ilyen pályagörbéken futnak végig! A madár a csőrét, a repülő és az autó az orrát „követi” a mozgás során. Tehát a test sebességvektora a test „orra” felé mutat. A sebességvektort a pályagörbe differenciálásával kapjuk meg. Ezzel rögtön rendelkeztünk az orientáció három szabadságfoka közül kettő felett. Nem kötöttük azonban meg, hogy a haladási irány, mint tengely körül hogyan forduljon el a test. Az autók és polgári repülők a mozgásuk során igyekeznek állandó „függőleges” irányt tartani, tehát mondhatjuk azt, hogy ez az irány a mozgás során legyen állandó (műrepülők és harci repülők különleges megközelítést, és különleges gyomrot is igényelnek). Egy másik lehetőség arra a felismerésre épít, hogy azt érezzük függőleges iránynak, amellyel ellentétesen erők hatnak ránk (ezért dőlünk be a kanyarban). A Newton-törvény miatt azonban a gyorsulás iránya megegyezik az erők irányával. A gyorsulás pedig a 3D görbe második deriváltjaként állítható elő.

- *Fizikai animáció*: A testekre időben változó erők hathatnak, amelyek hatására a testek elmozdulnak és elfordulnak. Ennek következtében akár újabb erők is ébredhetnek, vagy az erők megváltoznak. Tegyük fel, hogy egy űrhajót vezetünk, amelyet vonz egy bolygó, mégpedig az űrhajó és a bolygó tömegével egyenesen, a távolság négyzetével pedig fordítottan arányosan (ezt Newton gravitációs törvényének hívják, ami egy másik törvény, mint amiről korábban szó volt)! Az űrhajónk elmozdul a bolygó irányába, így a vonzóerő is nő, azaz egyre fokozódó gyorsulással közeledünk a bolygó felé. Ha nem teszünk semmit, becsapódunk a felszínbe, amellyel szerencsés esetben rugalmasan, szerencsétlenebb esetben rugalmatlanul ütközünk. Az ütközés utáni állapotot az energiamegmaradás, illetve az impulzusmegmaradás törvényei szerint határozhatjuk meg. Ha az űrhajónkon hajtóművek is vannak, azokat bekapcsolva újabb erők ébrednek, így a bolygótól eltávolodhatunk, vagy bolygó körüli pályára állhatunk.

  E kis történet alapján általánosságban is elfogadhatjuk, hogy a fizikai rendszereket a következő körforgás irányítja: a pillanatnyi erők meghatározzák a gyorsulást, amely pedig módosítja a sebességet és a pozíciót, minek következtében maguk az erők is változnak. Nem kell mást tennünk tehát, mint az erőket leírni, majd a mozgástörvényeket szimulálva kiszámítani, hogy a valós rendszerek hogyan mozognának ilyen körülmények között.

- *Mozgáskövető animáció* (*motion capture animation*): Az eddig ismertetett eljárásokkal létrehozott animációk valószerűsége a fizikai animációnál a fizikai modell és a szimuláció pontosságától, a többi esetben pedig az animátor ügyességétől függ. Pontos fizikai modell felépítésére csak egyszerű rendszerek esetén van esély, és komoly kihívás lenne például egy emberi lény több száz csontját, ízületét,

izmát precízen leírni. Az animátorok ügyessége lélegzetelállító, akik nem csak valószerű, azaz reális, hanem akár szürrealisztikus jelenetek elkészítésére is képesek. Mindenki fel tudja idézni Frédit a Flintstone családból, akinek mozgása nyilván semmilyen fizikai törvényt sem elégít ki, mégis hihető, és még élvezhetőbb is, mint ha igazi emberhez hasonlatosan totyogna. Mégis, ha általunk jól ismert mozgásokat látunk, és azt akarják elhitetni velünk, hogy a mozgás valódi, akkor hirtelen nagyon kritikussá válunk. Egy valódinak látszó, tehát nem karikatúra jellegű embert például nagyon nehéz lenne a fenti technikákkal animálni. Nem véletlen az, hogy számítógépes animációval már sok éve készülnek filmek, abban azonban játékok, illetve tárgyak (Luxo, Toy Story), és karikatúra stílusban ábrázolt állatok vagy emberek voltak a főszereplők (Egy bogár élete, Z a hangya, Shrek), a „hús-vér” virtuális emberek általában csak a távolban és rövid időre tűnhettek fel (Titanic, Pearl Harbor, Hídember). Mit tehetünk, ha már nem tudunk a természettel versenyre kelni? Lopunk tőle, ami a mozgáskövető animáció alapötlete. Egy valódi szereplőt, aki lehet ember, állat, tárgy stb. rábírunk arra, hogy végezze el a kívánt mozgást, amit kamerákkal rögzítünk. Az elkészült filmekből kinyerjük a számunkra fontos mozgásadatokat, majd a modellünket ezekkel az adatokkal vezéreljük. Az első „hús-vér” embernek látszó, virtuális főszereplőkkel készült film (Final Fantasy) mozgásainak 90%-át ezzel az eljárással vették fel.

A következő fejezetekben ezen eljárások részleteivel ismerkedünk meg.

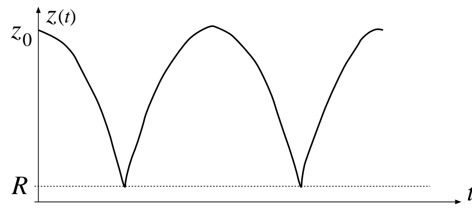
9.7. Képlet animáció

A képlet animációt akkor célszerű alkalmazni, ha a mozgás viszonylag egyszerű, és a mozgásváltozók időfüggvénye zárt alakban kifejezhető. Ebben a fejezetben egy pattogó labda példájával mutatjuk be ezt az eljárást.

Tegyük fel, hogy az R sugarú labda a $t = 0$ pillanatban az $[x_0, y_0, z_0]$ pontból $[v_x, v_y, 0]$ sebességgel indul! A földet a $z = 0$ sík képviseli, a labda a föld felett van ($z_0 > R$, $z(t) \geq R$). A nehézségi erő miatt a labda g gyorsulással gyorsuló sebességgel közeledik a föld felé, mialatt az x és y irányú sebessége változatlan. Első közelítésben a labda $[x(t), y(t), z(t)]$ pályája:

$$x(t) = x_0 + v_x \cdot t, \quad y(t) = y_0 + v_y \cdot t, \quad z(t) = z_0 - g \cdot \frac{t^2}{2}.$$

Ebben a képletben a $z(t)$ csak addig helyes, amíg a labda alja nem találkozik a földdel, ekkor ugyanis rugalmasan visszapattan, majd elérve a z_0 kezdeti magasságot újból a föld felé veszi az útját. Az ütközés abban a T időpontban következik be, amikor a magasság éppen a labda sugara, azaz $z(T) = z_0 - g \cdot T^2/2 = R$, amiből $T = \sqrt{2(z_0 - R)/g}$.



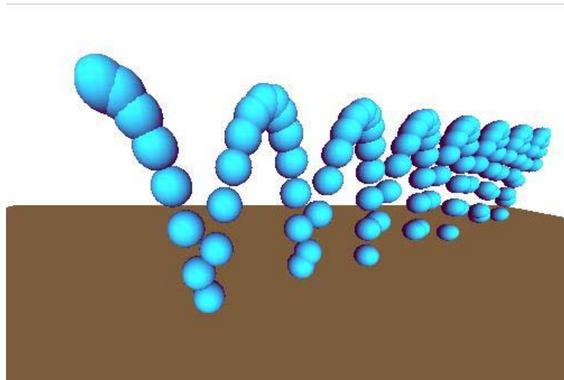
9.9. ábra. *A labda magasságának időfüggvénye ($z(t)$)*

Ránézve a 9.9. ábrára megállapíthatjuk, hogy a mozgás $2T$ szerint periodikus, tehát egy tetszőleges t időpillanatban a $z(t)$ számítását visszavezethetjük a $[-T,T]$ időintervallumra, ahol a $z(t) = z_0 - g \cdot t^2/2$ összefüggéssel dolgozhatunk. Vezessünk be egy $\tau(t)$ időtranszformációs függvényt, amely tetszőleges t időponthoz hozzárendeli azt a $\tau(t)$ időpontot, amelyre a $(t - \tau(t))$ a $2T$ periódus egész számú többszöröse, és $\tau(t)$ a $-T$ és T között van! A $\tau(t)$ függvény felhasználásával a labda pályája:

$$x(t) = x_0 + v_x \cdot t, \quad y(t) = y_0 + v_y \cdot t, \quad z(t) = z_0 - g \cdot \frac{\tau^2(t)}{2}.$$

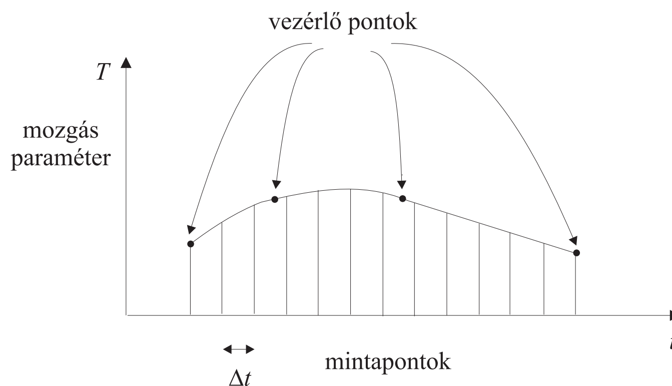
A labdát pattogtató OpenGL program:

```
const float g = 10;
//===============================================================
class Ball {
//===============================================================
   float x0, y0, z0;         // kezdeti pozíció
   float x, y, z;            // aktuális pozíció
   float vx, vy;             // kezdeti sebesség
   float R;                  // a gömb sugara
   float T;                  // pattogás fél periódusideje
   GLUquadricObj * sphere;   // a gömb
   Color kd, ks, ka;         // BRDF paraméterek
public:
   Ball(float x00, float y00, float z00, float vx0, float vy0, float R0)
      : kd(0.0, 1.0, 1.0), ks(1, 0.5, 1.0), ka(0.2, 0.0, 0.2) {
      x0 = x00; y0 = y00; z0 = z00; vx = vx0; vy = vy0;
      R = R0;                            // labda sugara
      T = sqrt(2.0 * (z0 - R) / g);      // fél peridósidő számítás
      sphere = gluNewQuadric();          // a labda gömbje
   }
   void AnimateIt(float t) {   // a labda animálása
      x = x0 + vx * t;
      y = y0 + vy * t;
      while(t > T) t -= 2 * T; // a tau függvény
      z = z0 - g * t * t / 2;
   }
   void DrawIt();              // a labda felrajzolása
};

//===============================================================
class BallWindow : public Application {
//===============================================================
   Ball  * ball;               // pattogó labda
   float   time;               // abszolút idő
public:
   BallWindow() : Application("Bouncing Ball", 400, 400)  { time = 0; }
   void Init();  // transzformációk, fényforrások inicializálása
   void Render() {
      glClearColor(0, 0, 0, 0);  // képernyő törlés
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      ball->DrawIt();            // labda felrajzolás
      SwapBuffers();             // buffercsere
   }
   void Do_a_Step(float dt) {    // egyetlen keret
      time += dt;                // abszolút idő számítása
      ball->AnimateIt(time);     // a labda mozgatása
      Render();                  // a labda felrajzolása
   }
};

void Application::CreateApplication() { new BallWindow(); }
```



9.10. ábra. *Képletanimációval mozgatott labda pályája*

9.8. Kulcskeret animáció



9.11. ábra. *Mozgástervezés interpolációval*

A kulcskeret animációban a *mozgástervezés* a kulcspontok felvételével kezdődik. A felhasználó megad egy $t_1, t_2, \ldots, t_n$ időpontsorozatot és elhelyezi a mozgatandó objektumot vagy a kamerát ezen időpontokban. A t_i időpillanatban beállított elrendezés az egyes objektumok paramétereire egy $\mathbf{p_o}(t_i)$ vezérlőpontot, más néven kulcspontot határoz meg. Ezen kulcspontokat felhasználva a program az objektum paramétereire egy-egy folytonos görbét illeszt.

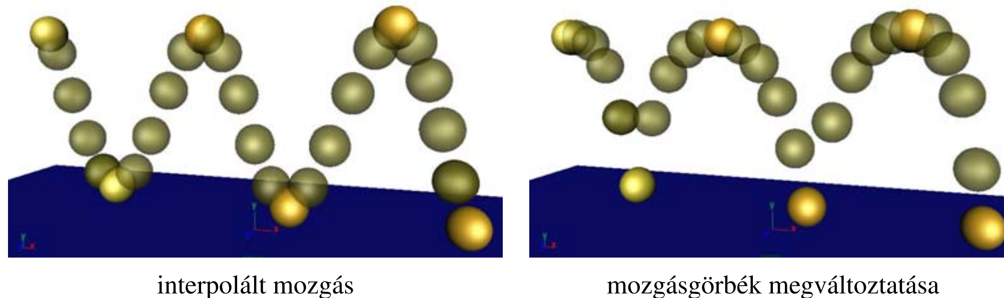
Az animációs fázisban a program az aktuális idő szerint mintavételezi a paraméterfüggvényeket, majd a paraméterekből kiszámítja a transzformációs mátrixokat, végül a transzformációs mátrixok felhasználásával előállítja a képet.

Összefoglalva a mozgástervezés és az animáció főbb lépései:

for (minden egyes o objektumra) {
    Kulcspontok idejének definiálása: $t_1^o, \ldots, t_n^o$; *// mozgástervezés*
    for (minden egyes k kulcspontra) {
        o objektum elrendezése: $\mathbf{p_o}(t_k^o) = [x(t_k^o), y(t_k^o), z(t_k^o), \alpha(t_k^o), \beta(t_k^o), \gamma(t_k^o)]_o$;
    }
    Interpolálj egy C^2 függvényt: $\mathbf{p_o}(t) = [x(t), y(t), z(t), \alpha(t), \beta(t), \gamma(t)]_o$;
}
Kamera kulcspontok idejének definiálása: $t_1^{cam}, \ldots, t_n^{cam}$; *// kamera pályatervezés*
for (minden egyes k kulcspontra) {
    Kamera beállítás: $\mathbf{p_{cam}}(t_k^{cam})$;
}
Interpolálj egy C^2 függvényt a kameraparaméterekhez: $\mathbf{p_{cam}}(t)$;

Óra inicializálás(t_{start}); *// animáció*
for ($t = t_{\text{start}}$; $t < t_{\text{end}}$; t = Óra leolvasás) {
    for (minden egyes o objektumra) {
        mintavételezés t-ben: $\mathbf{p_o} = [x(t), y(t), z(t), \alpha(t), \beta(t), \gamma(t)]_o$;
        $\mathbf{T_{M,o}} = \mathbf{T_{M,o}}(\mathbf{p_o})$;
    }
    A kamerához mintavételezés t-ben: $\mathbf{p_{cam}} = \mathbf{p_{cam}}(t)$;
    $\mathbf{T_V} = \mathbf{T_V}(\mathbf{p_{cam}})$;
    Képszintézis;
}

A kulcskeret animáció önmagában általában nem ad kielégítő eredményt, ugyanis az interpolációs eljárás a kulcskereteken kívül nem vesz figyelembe semmilyen további szempontot. Tegyük fel például, hogy egy pattogó labdát szeretnénk megjeleníteni (9.12. ábra).



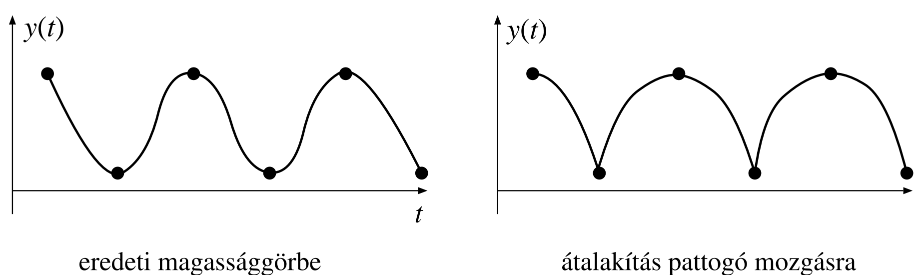
interpolált mozgás   mozgásgörbék megváltoztatása

9.12. ábra. *A labda animációjának négy kulcskerete és az interpolált mozgás*

A kulcspozíciók a következők:

1. a labda magasan van,
2. a labda a földön van, kissé távolabb,
3. a labda ismét magasan van, még távolabb,
4. a labda megint a földön van, egészen távol stb.

Az interpolációs eljárás hasonló görbéket illeszt a pattogás maximumaira, mint a minimumaira. Ez azonban nem ad pattogó hatást, hanem a labda látszólag csúszkál a felületen. Ebben az esetben a kulcskeret animáció által javasolt spline-okat kézzel alakíthatjuk át. A pattogó hatás érdekében a görbéknek parabolaívekhez hasonlatosnak kell lenniük, a föld közelében pedig az érintőknek hirtelen kell megváltozniuk (9.13. ábra).



9.13. ábra. *A pattogó labda kulcskeretei és az interpolált $y(t)$ mozgásgörbe*

9.8.1. Animációs spline-ok

A következőkben olyan interpolációs eljárásokkal foglalkozunk, amelyeket előszeretettel használnak a mozgásgörbék előállítására. Igazából ezek nem is feltétlenül *spline*-ok, azaz nem mindenhol C^2 folytonos összetett görbék. A C^2 folytonosság csak a görbeszegmensek belsejére teljesül, a szegmensek találkozási pontjaiban legfeljebb csak C^1 folytonosság áll fenn. A találkozási pontokban, például az ütközés hirtelen bekövetkező változásainak megfelelően, akár C^0-s illeszkedés is beállítható.

Az interpolációs feladat tehát a következő. Adottak az ismeretlen $\mathbf{f}(t)$ mozgásváltozó időfüggvény értékei a $t_1, t_2, \ldots, t_n$ helyeken: $\mathbf{f}_1 = \mathbf{f}(t_1)$, $\mathbf{f}_2 = \mathbf{f}(t_2)$, ..., $\mathbf{f}_n = \mathbf{f}(t_n)$. A t_i és t_{i+1} időpontok között a függvényt a C^2 folytonosság érdekében harmadrendű polinom formában keressük, azaz:

$$\mathbf{f}(t) = \mathbf{a}_i \cdot (t - t_i)^3 + \mathbf{b}_i \cdot (t - t_i)^2 + \mathbf{c}_i \cdot (t - t_i) + \mathbf{d}_i, \qquad \text{ha } t_i \le t < t_{i+1}.$$

Az ismeretlen tényezők az $\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i, \mathbf{d}_i$ polinomegyütthatók. Ezeket az együtthatókat részint az interpolációs feltételből kaphatjuk meg:

$$\begin{aligned} \mathbf{f}_i &= \mathbf{f}(t_i) = \mathbf{d}_i, \\ \mathbf{f}_{i+1} &= \mathbf{f}(t_{i+1}) = \mathbf{a}_i \cdot (t_{i+1} - t_i)^3 + \mathbf{b}_i \cdot (t_{i+1} - t_i)^2 + \mathbf{c}_i \cdot (t_{i+1} - t_i) + \mathbf{d}_i, \end{aligned}$$

de ez még csak két egyenlet, amely a 4 ismeretlen egyértelmű meghatározásához kevés. Tegyük fel továbbá, hogy valahogyan a sebességvektorra, azaz mozgásfüggvény deriváltjaira is szert teszünk a kulcspontokban. Ha a sebességek a $t_1, t_2, \ldots, t_n$ időpontokban $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ értékűek, akkor ezek alapján a még hiányzó egyenleteket is felírhatjuk:

$$\begin{aligned} \mathbf{v}_i &= \mathbf{f}'(t_i) = \mathbf{c}_i, \\ \mathbf{v}_{i+1} &= \mathbf{f}'(t_{i+1}) = 3 \cdot \mathbf{a}_i \cdot (t_{i+1} - t_i)^2 + 2 \cdot \mathbf{b}_i \cdot (t_{i+1} - t_i) + \mathbf{c}_i. \end{aligned}$$

Ebből a négy egyenletből már az ismeretlen polinomegyütthatók kiszámíthatók:

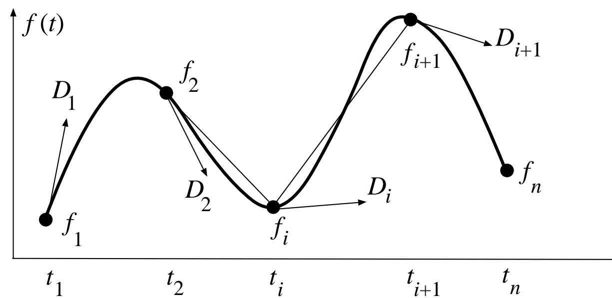
$$\begin{aligned} \mathbf{a}_i &= \frac{\mathbf{v}_{i+1} + \mathbf{v}_i}{(t_{i+1} - t_i)^2} - \frac{2(\mathbf{f}_{i+1} - \mathbf{f}_i)}{(t_{i+1} - t_i)^3}, \\ \mathbf{b}_i &= \frac{3(\mathbf{f}_{i+1} - \mathbf{f}_i)}{(t_{i+1} - t_i)^2} - \frac{\mathbf{v}_{i+1} + 2\mathbf{v}_i}{(t_{i+1} - t_i)}, \\ \mathbf{c}_i &= \mathbf{v}_i, \\ \mathbf{d}_i &= \mathbf{f}_i. \end{aligned} \tag{9.15}$$

Mivel két görbeszegmens közösen birtokol egy kulcspontot és az ott érvényes deriváltat (például az $\mathbf{f}_i$-t és $\mathbf{v}_i$-t az $(i-1)$-edik görbeszegmens végén és az i-edik görbeszegmens elején), a két görbeszegmens C^1 folytonosan illeszkedik egymáshoz.

A kérdés most már csak az, hogy honnan vegyük a deriváltak értékét a kulcspontokban. Az első ötlet a *harmadrendű spline* (*cubic spline*) fogalmához vezet. Válasszuk meg úgy a deriváltakat, hogy a szegmensek érintkezési pontjaiban C^2 folytonosság is teljesüljön! Ennek menete a következő: A szegmensek polinomegyütthatóit az $\mathbf{f}_i$, $\mathbf{v}_i$ paraméterek segítségével fejezzük ki (megoldjuk a 9.15. egyenletrendszert az $\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i, \mathbf{d}_i$ ismeretlenekre). A szegmensek kezdő- és végpontban érvényes második deriváltját a polinomegyütthatókból, végső soron az $\mathbf{f}_i$, $\mathbf{v}_i$ paraméterekből számítjuk ki, majd felírjuk azt az egyenletet, hogy az i-edik görbe második deriváltja a t_{i+1} helyen egyezzen meg az $(i+1)$-edik görbe második deriváltjával ugyanezen a t_{i+1} helyen minden $i = 1,2,3,\ldots,n-2$-re. Ez egy $n-2$ ismeretlenes egyenletrendszer az ismeretlen $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ deriváltakra. Mivel az ismeretlenek száma meghaladja az egyenletekét, a megoldás nem egyértelműen meghatározott. Teljesen meghatározottá tehetjük a megoldást, ha a $\mathbf{v}_1$ kezdeti deriváltat és a $\mathbf{v}_n$ végderiváltat önkényesen felvesszük, például abból a feltételből, hogy a test nyugalomból indul (a sebesség, azaz a derivált zérus), és a mozgás után a test nyugalmi állapotba jut. A harmadrendű spline kialakításához tehát egy nagyméretű,

lineáris egyenletrendszert kell megoldani, amit általában szeretnénk elkerülni. Még ennél is kellemetlenebb azonban, hogy ezzel elveszítjük a görbe lokális vezérelhetőségét. Bármelyik kulcspozíciót változtatjuk is meg, ez a lineáris egyenletrendszeren keresztül az összes deriváltat befolyásolja, így a görbe teljes tartományán érezteti a hatását. Ez egy nagyon súlyos érv, ami arra indít bennünket, hogy más megoldás után nézzünk, még akár azon az áron is, hogy a szegmensek illeszkedési pontjaiban be kell érnünk C^1 folytonossággal. Ezt már akármilyen $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ választás biztosítja, keressünk tehát egy olyat, ami vélhetőleg szép, sima görbét eredményez.

Catmull – Rom spline



9.14. ábra. *Catmull – Rom spline*

Ha a $(t_{i-1}, \mathbf{f}_{i-1})$ és a $(t_i, \mathbf{f}_i)$ pontok között egyenletes sebességgel mozognánk, a sebesség

$$\frac{\mathbf{f}_i - \mathbf{f}_{i-1}}{t_i - t_{i-1}}$$

értékűre adódna. Hasonlóan, a $(t_i, \mathbf{f}_i)$-től a $(t_{i+1}, \mathbf{f}_{i+1})$-be átlagosan

$$\frac{\mathbf{f}_{i+1} - \mathbf{f}_i}{t_{i+1} - t_i}$$

sebességgel jutnánk át. A két görbe találkozásánál válasszuk a sebességet a két intervallum átlagos sebességeinek középértékének, azaz

$$\mathbf{v}_i = \frac{1}{2} \cdot \left(\frac{\mathbf{f}_i - \mathbf{f}_{i-1}}{t_i - t_{i-1}} + \frac{\mathbf{f}_{i+1} - \mathbf{f}_i}{t_{i+1} - t_i} \right).$$

Mint a harmadrendű spline-nál, a kezdeti és a végsebességet itt is önkényesen vehetjük fel. Ezzel kész is volnánk, hiszen az összes derivált értékét ismerjük. Az így előállított görbéket *Catmull – Rom spline*-nak nevezzük. A Catmull – Rom spline harmadfokú polinomokból összerakott összetett görbe, amelyben a szegmensek C^1 folytonosan

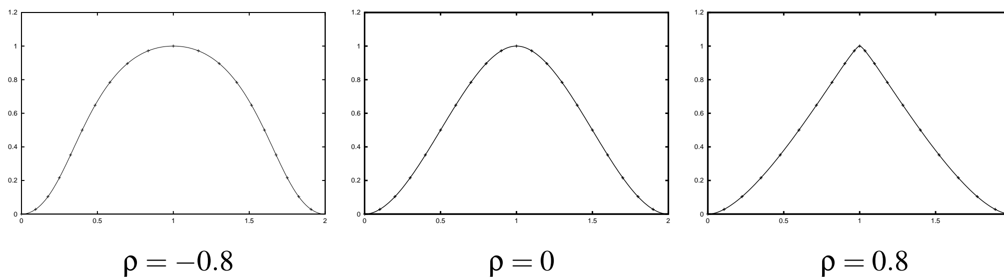
illeszkednek. Ha az illeszkedési pontban eltérő deriváltakat is megengedünk, akkor a folytonossági szint akár C^0-ig csökkenthető. A Catmull – Rom spline lokálisan vezérelhető, hiszen egyetlen kulcspont megváltoztatása közvetlenül a két, itt találkozó szegmensre hat, valamint a deriváltak értékére ebben és a szomszédos pontokban. A deriváltakon keresztül így indirekt módon még a két következő szegmens is megváltozhat, a többi azonban semmiképpen sem.

Kochanek – Bartels spline

A Catmull – Rom spline a deriváltak értékét egy heurisztikus szabály szerint az előző és a következő szegmens sebességeinek átlagaként állította elő. A mozgást szabadabban vezérelhetjük, ha a heurisztikus szabályt lazábban fogalmazzuk meg. Az alábbi műveletsor eredménye *Kochanek – Bartels spline* [74] néven ismeretes. Vezessünk be először egy ρ *feszültség* (*tension*) paramétert, amely a kulcspontban az átlagos sebességet arányosan csökkenti, illetve növeli. A feszültség a $[-1,1]$ tartományban változhat. A Catmull – Rom spline rögzített $1/2$-es szorzója helyett egy $(1-\rho)/2$ tényezőt fogunk használni:

$$\mathbf{v}_i = \frac{1-\rho}{2} \cdot \left(\frac{\mathbf{f}_i - \mathbf{f}_{i-1}}{t_i - t_{i-1}} + \frac{\mathbf{f}_{i+1} - \mathbf{f}_i}{t_{i+1} - t_i} \right).$$

Ha a ρ feszültség értékben nagy (egyhez közeli), akkor az illeszkedési pontban a sebesség értéke kicsi, és megfordítva, ha ρ kicsi (-1-hez közeli), akkor a sebesség értéke nagy. A nagy sebességet nem könnyű megváltoztatni, ezért a kulcspontot megelőzően és azt követően is a mozgás jellege hasonló lesz, mint a kulcspontban (a 9.15. ábrán a pontok az azonos idők alatt bejárt görbetartományokat választják el).

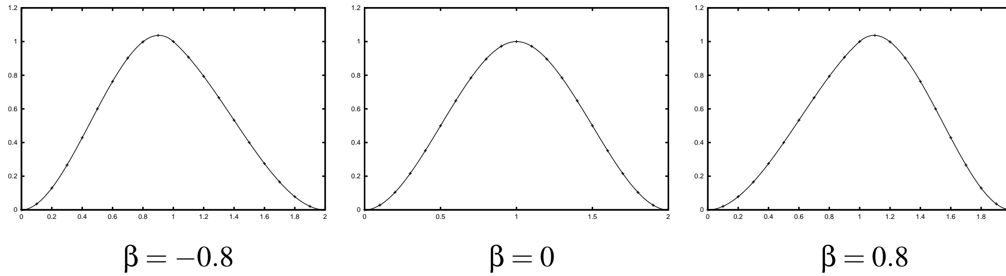


9.15. ábra. *Kochanek – Bartels spline különböző feszültség értékekre*

A második ötlet a mozgásgörbe általánosítására az, hogy nem kell egyenlő arányban támaszkodni a két illeszkedő görbeszegmens átlagos meredekségére, azaz a testvéries $(0.5, 0.5)$ súlyozás helyett részrehajló $((1+\beta)/2, (1-\beta)/2)$ súlyozást is alkalmazhatunk, ahol a β *torzítás* -1 és 1 között van:

$$\mathbf{v}_i = \frac{1+\beta}{2} \cdot \frac{\mathbf{f}_i - \mathbf{f}_{i-1}}{t_i - t_{i-1}} + \frac{1-\beta}{2} \cdot \frac{\mathbf{f}_{i+1} - \mathbf{f}_i}{t_{i+1} - t_i}.$$

Ha az első szegmenst vesszük nagyobb súllyal figyelembe, a test látszólag túllendül a kulcsponton, ha viszont a másodikat, akkor a test lendületet vesz a második szakaszhoz (9.16. ábra).



9.16. ábra. *Kochanek – Bartels spline különböző torzítás értékekre*

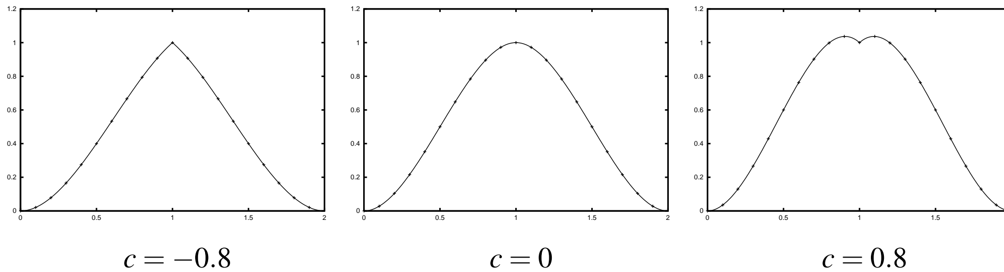
Végül megengedhetjük, hogy a görbe a kulcspont két oldalán eltérő deriválttal rendelkezzen, azaz csupán C^0 folytonos legyen. A két deriváltat most is a két szegmens átlagos deriváltjából számítjuk, de a bal oldaliban az első szegmens meredeksége, a jobb oldaliban pedig a második szegmens átlagos meredeksége vesz részt nagyobb súllyal:

$$\begin{aligned}
\mathbf{v}_i^{bal} &= \frac{1-c}{2} \cdot \frac{\mathbf{f}_i - \mathbf{f}_{i-1}}{t_i - t_{i-1}} + \frac{1+c}{2} \cdot \frac{\mathbf{f}_{i+1} - \mathbf{f}_i}{t_{i+1} - t_i}, \\
\mathbf{v}_i^{jobb} &= \frac{1+c}{2} \cdot \frac{\mathbf{f}_i - \mathbf{f}_{i-1}}{t_i - t_{i-1}} + \frac{1-c}{2} \cdot \frac{\mathbf{f}_{i+1} - \mathbf{f}_i}{t_{i+1} - t_i}.
\end{aligned}$$

A c paramétert a *folytonosság* mértékének nevezzük. A $c = 0$ esetben a C^1 folytonos Catmull – Rom spline-t kapjuk vissza. Ha c zérustól eltérő, a görbe deriváltja nem lesz folytonos, hanem a kulcspontban egy c-vel arányos nagyságú ugrást tartalmaz (9.17. ábra). A feszültséget, torzítást és folytonosságot össze is vonhatjuk, így a Kochanek-görbe deriváltjainak legáltalánosabb alakja:

$$\mathbf{v}_i^{bal} = \frac{(1-\rho)(1-c)(1+\beta)}{2} \cdot \frac{\mathbf{f}_i - \mathbf{f}_{i-1}}{t_i - t_{i-1}} + \frac{(1-\rho)(1+c)(1-\beta)}{2} \cdot \frac{\mathbf{f}_{i+1} - \mathbf{f}_i}{t_{i+1} - t_i},$$

$$\mathbf{v}_i^{jobb} = \frac{(1-\rho)(1+c)(1+\beta)}{2} \cdot \frac{\mathbf{f}_i - \mathbf{f}_{i-1}}{t_i - t_{i-1}} + \frac{(1-\rho)(1-c)(1-\beta)}{2} \cdot \frac{\mathbf{f}_{i+1} - \mathbf{f}_i}{t_{i+1} - t_i}.$$



$c = -0.8$ $c = 0$ $c = 0.8$

9.17. ábra. *Kochanek – Bartels spline különböző folytonossági paraméterekre*

A Kochanek – Bartels spline harmadfokú szegmensekből épül fel, amelyet az alábbi C++ osztály valósít meg. A programban felhasználtunk egy `NDVector` osztályt, amely a `Vector` osztályhoz hasonlatos, de nem csak három, hanem tetszőleges számú koordinátát tartalmazhat.

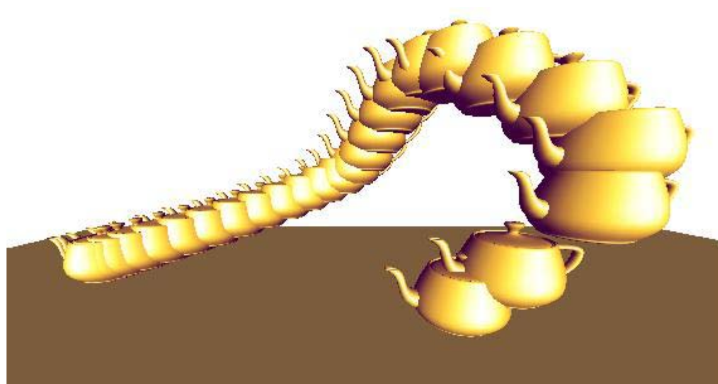
```
//=================================================================
class Segment { // a spline egy szegmense
//=================================================================
   NDVector a, b, c, d;   // polinom
   float    tstart, Dt;   // kezdeti idő és hossz
public:
   void Init(NDVector& f0, NDVector& v0, float t0,
             NDVector& f1, NDVector& v1, float t1) {
      tstart = t0;        // polinomegyütthatók számítása
      Dt = t1 - t0;
      a = (v1 + v0) / Dt / Dt - (f1 - f0) * 2 / Dt / Dt / Dt;
      b = (f1 - f0) * 3 / Dt / Dt - (v1 + v0*2) / Dt;
      c = v0;
      d = f0;
   }
   NDVector Value(float t) {  // polinom kiértékelése
      float T = t - tstart;
      return (a * T * T * T + b * T * T + c * T + d);
   }
};
```

A kulcspontokban a függvényértéket és az időpontot tároljuk, valamint az interpolációt szabályozó paramétereket:

```
//=================================================================
struct Key { // egy kulcs
//=================================================================
   NDVector f;              // függvény érték
   float t;                 // időpont
   float tens, bias, cont;  // feszültség, torzítás, folytonosság
   Key() { tens = bias = cont = 0; }
};
```

Végül a spline a kulcspontok alapján kiszámolja az egyes szegmenseket és az aktuális idő alapján a kiválasztott szegmens szerint interpolál:

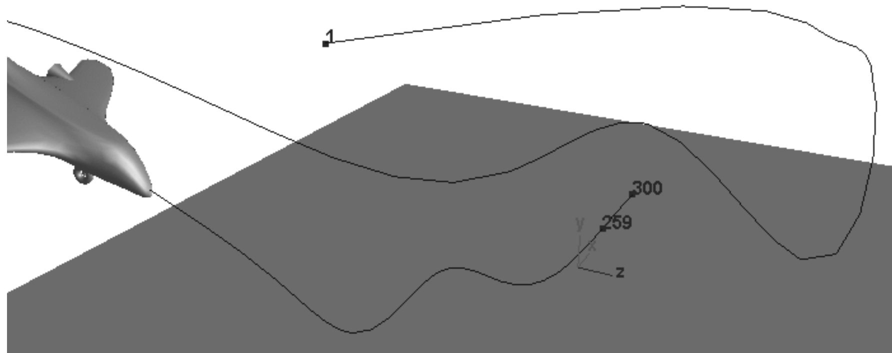
```
//===============================================================
class KochanekSpline {
//===============================================================
   int       nkeys;    // a kulcspozíciók száma
   Key     * keys;     // a kulcsok tömbje
   Segment * segments; // a szegmensek tömbje
public:
   KochanekSpline(Key * keys0, int nkeys0, NDVector& vstart, NDVector& vend) {
      keys = keys0; nkeys = nkeys0;
      segments = new Segment[nkeys - 1];
      for(int i = 0; i < nkeys - 1; i++) {
         NDVector v0 = vstart, v1 = vend;
         if (i == 0) v0 = vstart;
         else v0 = (keys[i].f - keys[i-1].f)/(keys[i].t - keys[i-1].t) *
                   (1-keys[i].tens)*(1+keys[i].cont)*(1+keys[i].bias)/2  +
                   (keys[i+1].f - keys[i].f)/(keys[i+1].t - keys[i].t) *
                   (1-keys[i].tens)*(1-keys[i].cont)*(1-keys[i].bias)/2;
         if (i == (nkeys - 2)) v1 = vend;
         else v1 = (keys[i+1].f - keys[i].f)/(keys[i+1].t - keys[i].t) *
                  (1-keys[i+1].tens)*(1-keys[i+1].cont)*(1+keys[i+1].bias)/2 +
                  (keys[i+2].f - keys[i+1].f)/(keys[i+2].t - keys[i+1].t) *
                  (1-keys[i+1].tens)*(1+keys[i+1].cont)*(1-keys[i+1].bias)/2;
         segments[i].Init(keys[i].f,v0,keys[i].t, keys[i+1].f,v1,keys[i+1].t);
      }
   }

   NDVector Value(float t) {   // az interpolált érték
      if (t < keys[0].t) return keys[0].f;
      for(int i = 0; i < nkeys - 1; i++)
         if (keys[i].t <= t && t < keys[i+1].t) return segments[i].Value(t);
      return keys[nkeys-1].f;
   }
};
```



9.18. ábra. *Kochanek – Bartels spline mentén mozgatott teáskanna*

9.9. Pálya animáció

A mozgás leírásának gyakran a legtermészetesebb módja a mozgás pályájának a megadása. Egy repülő vagy madár esetében a pálya a háromdimenziós térben szép ívek mentén kanyaroghat, egy biliárdgolyó viszont az asztalon egyenes szakaszokból álló pályát követ, egy vonat pedig a felszínre illesztett íves pályát jár be.



9.19. ábra. *Egy repülő mozgásának megadása pálya animációval*

A *pálya animáció* (*path animation*) során tehát először egy háromdimenziós görbét definiálunk, majd a test egy kitüntetett pontját a megadott pályagörbén adott időzítési viszonyok közepette vezetjük végig. Általános íves és egyenes szakaszokkal leírható görbékkel már a geometriai modellezéssel foglalkozó fejezetben találkoztunk, így a pálya megadása nem tűnik különösebben nehéz feladatnak. A görbemodellezés egy paraméteres $\vec{p}(u) = [x(u), y(u), z(u)]$ függvényt eredményez, amely az $u = [u_{start}, u_{end}]$ intervallum bejárása során végigfut a görbe pontjain.

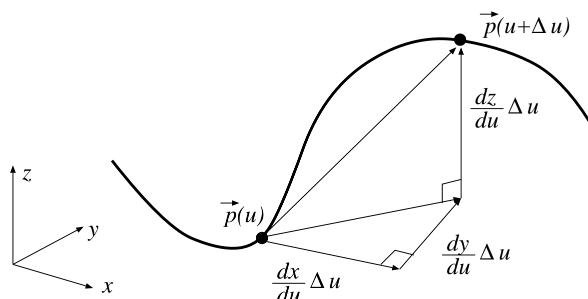
Az időzítési viszonyokról a t időparaméter és az u görbeparaméter összekapcsolásával rendelkezhetünk. Kézenfekvő lenne az $u = t$ megfeleltetés, vagy valamilyen lineáris $u = at + b$ függvény, ez azonban gyakran nem ad kielégítő eredményt. Képzeljük el, hogy egyenletes B-spline görbét használunk, amelynek kezdetén a vezérlőpontok meglehetős sűrűséggel, a végén pedig egymástól távol helyezkednek el! Ez azt jelenti, hogy a görbe kezdetén az egységnyi idő alatt bejárt görbeszegmensek kicsinyek, a görbe végén pedig nagyok. A mozgás lassan indul, majd a későbbiekben felgyorsul. Ezen segíthetnénk ugyan azzal, ha a vezérlőpontokat nagyjából egyenlő távolságra vennénk fel, de ez ellentmondana a geometriai definíció elvárásainak, miszerint, ahol a görbe bonyolult, kanyargós, ott sok vezérlőpontot, ahol pedig egyszerű, ott kevés vezérlőpontot kell használnunk. Nem érdemes a geometria és az animáció eltérő követelményeit összekeverni, mert akkor egyikét sem tudjuk maradéktalanul kielégíteni. Ehelyett a

geometriának megfelelő paraméterezést célszerű alkalmazni, az animációhoz viszont az időzítési viszonyokat jól tükröző $u(t)$ függvényt kell használni.

Írjuk elő a megtett s utat az idő függvényében:

$$s = f(t).$$

Például a pálya egyenletes sebességű bejárása az $s = vt$ út–idő függvénynek felelne meg. A görbe pontjait azonban az u paraméterből számíthatjuk ki, ezért kapcsolatot kell teremteni a megtett út és a görbeparaméter között.



9.20. ábra. *A pályán bejárt úthossz számítása*

Tegyük fel, hogy a görbeparaméter kicsiny Δu értékkel megnő! Ennek hatására a görbén lévő $\vec{p}(u)$ pont az x, y, z irányokban $dx/du \cdot \Delta u$, $dy/du \cdot \Delta u$ és $dz/du \cdot \Delta u$ távolsággal mozdul el (9.20. ábra). A háromdimenziós Pitagorasz-tétel értelmében ez éppen

$$\Delta s = \sqrt{\left(\frac{dx}{du}\right)^2 + \left(\frac{dy}{du}\right)^2 + \left(\frac{dz}{du}\right)^2} \cdot \Delta u$$

távolságnak felel meg. A görbeparaméter teljes megváltozását ilyen kicsiny megváltozások összegeként, azaz integráljaként írhatjuk fel:

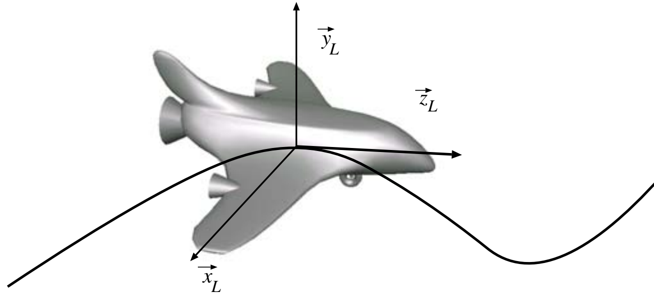
$$s(u) = \int_{u_{start}}^{u} \sqrt{\left(\frac{dx}{du}\right)^2 + \left(\frac{dy}{du}\right)^2 + \left(\frac{dz}{du}\right)^2}\, du. \tag{9.16}$$

A megtett s útnak az idő és a görbeparaméter szerinti kifejezését összekapcsolva, előállíthatjuk az u görbeparamétert a t idő függvényében:

$$s(u) = f(t) \quad \Longrightarrow \quad u = s^{-1}(f(t)).$$

A 9.16. egyenlet szerint az s monoton növekvő és nemlineáris, tehát mindig invertálható, de a számítás közelítő módszereket igényel. A következőkben egy rendkívül

egyszerű iterációs eljárást ismertetünk, amely kihasználja, hogy az időkeretek elején ismerjük az összetartozó $u_{start} - t_{start}$ párt (ez éppen az előző időkeret végén érvényes értékpár), valamint azt is, hogy az időkeret hossza általában nem túlságosan nagy. Haladjunk az időkeret belsejében Δu lépésekben és alkalmazzuk az út–paraméter integrál közelítő összeggel történő becslését! Amint az $s(u)$ nagyobb lesz mint az előírt $f(t_{end})$, leállítjuk az iterációt, és az aktuális u értéket tekintjük a megoldásnak. Az utolsó és utolsó előtti lépések között az út–paraméter függvényt lineárisnak tekintve még tovább javíthatjuk a megoldás pontosságát.



9.21. ábra. *Az orientáció vezérlése a pálya animáció során*

Az ismertetett pálya animációs eljárás a mozgó objektumunk kitüntetett pontjának pályájáról rendelkezik. Ehhez képest azonban a test el is fordulhat, amiről még semmit sem mondtunk. Az orientáció megállapításának egyik lehetősége arra a felismerésre épít, hogy egyes objektumok (madarak, repülők stb.) úgy repülnek, hogy a csőrüket, orrukat stb. követik. A követés pontosabban azt jelenti, hogy a pillanatnyi sebességvektor mindig a csőr, orr irányába mutat. Rendeljünk egy modellezési-koordinátarendszert a tárgyunkhoz, amelynek egységvektorai a világ-koordinátarendszerben az $\vec{x}_L, \vec{y}_L, \vec{z}_L$ egymásra merőleges vektoroknak felelnek meg! Tegyük fel, hogy a követendő irány (csőr, orr) a $\vec{z}_L$ vektor. A $\vec{z}_L$-t a $\vec{p}(t)$ pályagörbe sebességvektorának a normalizálásával kapjuk:

$$\vec{z}_L = \left(\frac{d\vec{p}(t)}{dt}\right)^0 = \left(\frac{d\vec{p}(u)}{du}\right)^0.$$

A 0 kitevő a normalizálásra utal, az idő szerinti differenciálást pedig azért cserélhettük fel a paraméter szerinti differenciálással, mert a paraméter–idő függvény skalár, tehát csak járulékos skálázást jelent, amelyet a normalizálás úgyis kiegyenlít.

A $\vec{z}_L$ ismerete még mindig nem jelenti az orientáció teljes ismeretét, hiszen a test a repülési irány körül még foroghat. Az egyik lehetséges megoldás, ha önkényesen jelöljük ki a repülés függőleges irányát. Például mondhatjuk azt, hogy az $\vec{y}_L$ függőleges irány mindig egy előre definiált $\vec{Y}$ vektornak a $\vec{z}_L$ vektorra merőleges komponense. Az

orientációs irányokat ekkor a következőképpen számíthatjuk ki:

$$\vec{z}_L = \left(\frac{d\vec{p}(u)}{du}\right)^0, \quad \vec{x}_L = \left(\vec{Y} \times \vec{z}_L\right)^0, \quad \vec{y}_L = \vec{z}_L \times \vec{x}_L.$$

A másik lehetőség arra a felismerésre épít, hogy azt az irányt érezzük függőlegesnek, amely felé a többi erőt kiegyenlítő kényszererők nyomnak bennünket (ezért dőlünk be a kanyarban). A dinamika alaptörvénye szerint az eredő erő a pályafüggvény második deriváltjával arányos, tehát az idáig önkényesen felvett $\vec{Y}$ irányt a következőképpen érdemes megválasztani:

$$\vec{Y} = \frac{d^2\vec{p}(t)}{dt^2}.$$

Ezt az eljárást a kitalálójáról *Frenet-keret*nek nevezzük. A Frenet-keret módszer nehézségekbe ütközik, ha a pályafüggvény második deriváltja zérus, hiszen ekkor a függőleges irány nem definiált. Ilyen helyzetekben az utolsó nem zérus második deriváltat kell a függőleges iránynak tekinteni.

9.10. Fizikai animáció

A fizikai animáció a dinamika törvényei alapján szimulálja a testek mozgását. A törvények a *Newton-féle axiómák*on alapulnak, amelyek a következőket mondják ki:

1. Van olyan koordinátarendszer, amelyben egy külső erőktől nem befolyásolt pontszerű test vagy nyugalomban van, vagy pedig egyenes vonalú egyenletes mozgást végez (*tehetetlenség törvénye*).

2. A pontszerű testre ható $\vec{F}$ erő és az általa létesített $\vec{a}$ gyorsulás között az $\vec{F} = m \cdot \vec{a}$ összefüggés teremt kapcsolatot, ahol az m a test tömege (*dinamika alaptörvénye*).

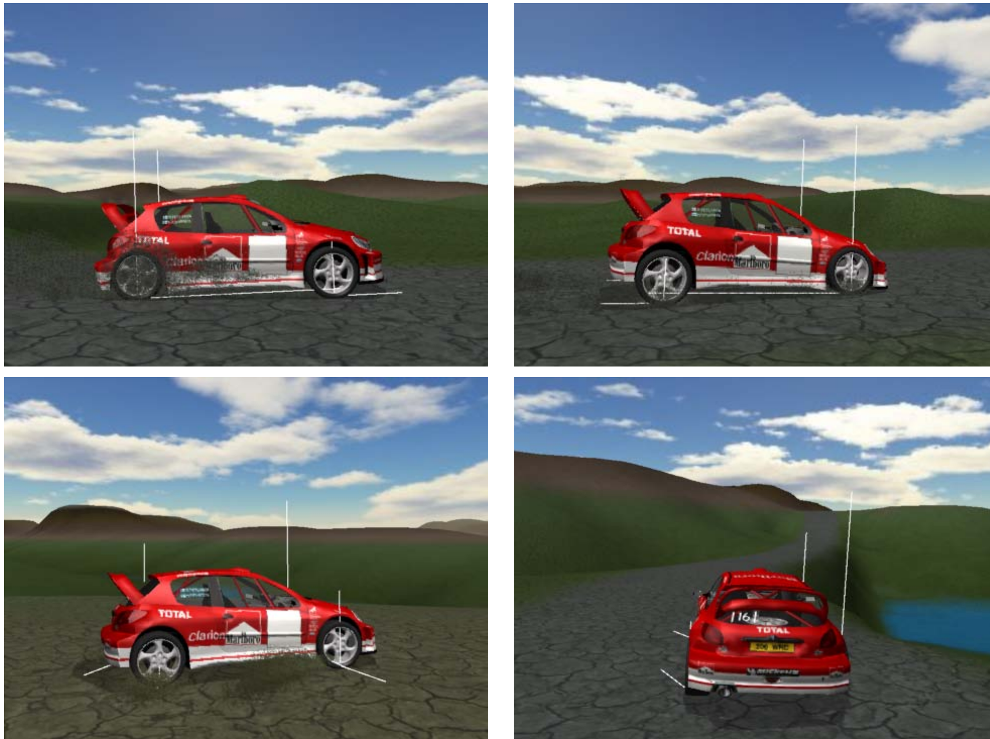
3. Ha egy A test $\vec{F}$ erővel hat egy B testre, akkor a B test éppen $-\vec{F}$ erővel hat az A testre (*hatás–ellenhatás törvénye*).

4. Egy pontszerű testre ható erők hatása megegyezik az erők vektoriális összegének a hatásával.

Mivel a gyorsulás a sebesség deriváltja, és a tömeg a mozgás során általában nem változik (ez alól a rakéták kivételek), a második Newton-törvény a következőképpen is felírható:

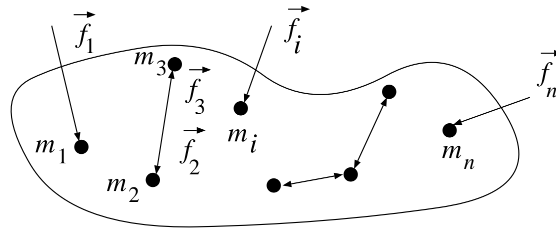
$$\vec{F} = m \cdot \vec{a} = \frac{d(m\vec{v})}{dt} = \frac{d\vec{I}}{dt}.$$

A képletben megjelenő $\vec{I} = m\vec{v}$ mennyiséget *lendület*nek vagy *impulzus*nak (*linear momentum*) nevezzük. Az első axióma értelmében, ha a testre nem hat erő, a sebessége, így az impulzusa állandó. Az impulzus szemléletes jelentése az „összegyűjtött erő”, hiszen $I = \int F\, dt$.



9.22. ábra. *Gépkocsi fizikai szimulációja (az erőket szakaszok jelzik)[35]*

9.10.1. Kiterjedt testek haladó mozgása és forgása



9.23. ábra. *Kiterjedt testek, mint tömegpontok gyűjteményei*

A Newton-törvények csak pontszerű testekre vonatkoznak (a hatás–ellenhatás törvényét kivéve, amely tetszőleges testre alkalmazható). A gyakorlatban előforduló testek nem ilyenek, hanem kiterjedtek. A *kiterjedt test*eket tekinthetjük apró m_i tömegű tömegpontok összességének (9.23. ábra). Az egyes tömegpontokra fennállnak a már ismert Newton-törvények, miszerint az impulzusvektorok deriváltja arányos az erővel. Legyen az i-edik tömegpontra ható erő $\vec{f}_i$. A dinamika alaptörvénye szerint:

$$m_i \cdot \frac{d^2\vec{r}_i}{dt^2} = \frac{d\vec{I}_i}{dt} = \vec{f}_i.$$

Az elemi tömegpontok mozgásegyenleteinek összegzésével azt írhatjuk, hogy

$$\sum_i \frac{d\vec{I}_i}{dt} = \frac{d\sum_i \vec{I}_i}{dt} = \frac{d\vec{I}}{dt} = \sum_i \vec{f}_i = \vec{F},$$

ahol $\vec{I} = \sum_i \vec{I}_i$ a test teljes impulzusa, az $\vec{F} = \sum \vec{f}_i$ pedig az *eredő erő*. Vegyük észre, hogy míg az $\vec{f}_i$ elemi erők a külső és belső erőket egyaránt tartalmazzák, az eredő erőben már csak a külső erők szerepelnek! Ha a test belsejében az egyik részecske $\vec{f}$ belső erővel hat egy másikra, akkor a hatás–ellenhatás törvénye miatt a másik éppen $-\vec{f}$ erővel hat vissza, így a belső erők az összegzés során „kiejtik” egymást. Ha tehát a pontrendszerre nem hat külső erő (a rendszer zárt), akkor az impulzusának deriváltja zérus, azaz az összes impulzusa állandó. Ez az *impulzus megmaradás törvénye*.

Jelöljük a teljes $\sum m_i$ tömeget m-mel, a test *tömegközéppont*ját pedig $\vec{c}$-vel:

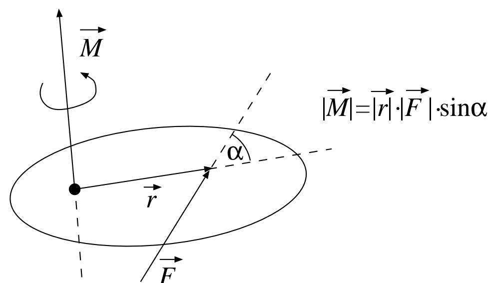
$$\vec{c} = \frac{\sum m_i \cdot \vec{r}_i}{m}.$$

Az impulzusok behelyettesítésével egy újabb fontos összefüggéshez jutunk:

$$\frac{d\sum_i \vec{I}_i}{dt} = \frac{d\sum_i m_i\vec{v}_i}{dt} = \frac{d^2\sum_i m_i\vec{r}_i}{dt^2} = m \cdot \frac{d^2}{dt^2}\frac{\sum m_i \cdot \vec{r}_i}{m} = m \cdot \frac{d^2\vec{c}}{dt^2} = \vec{F}.$$

A tömegközéppont a test kitüntetett pontja, hiszen az előző összefüggés szerint a kiterjedt test tömegét ide koncentrálhatjuk, és az így kapott egyetlen tömegpont mozgása megegyezik az eredeti test tömegközéppontjának mozgásával. A tömegközéppont homogén erőtérben megegyezik a *súlypont*tal. A súlypont az elnevezését onnan kapta, hogy egy testet a súlypontjában felfüggesztve a test nyugalomban marad. A kedves Olvasó a tömegközépponttal, illetve a súlyponttal nem először találkozik ebben a könyvben. Ezt a fogalmat használtuk a paraméteres görbék súlyfüggvényeinek és a homogén koordinátáknak a bevezetésénél is (3.1.5. fejezet).

Ha az összes erő eredője zérus, a test tömegközéppontja nyugalomban van, vagy egyenletes sebességgel mozog. Ha az erők nem egyetlen pontban érik a pontrendszert, ez nem feltétlenül jelenti azt, hogy maga a test is nyugalomban van, hiszen eközben a súlypont körül foroghat. Az erők forgató hatását *forgatónyomaték*nak nevezzük. A forgatónyomaték arányos az erővel és az *erőkar*ral, azaz a forgástengely és az erő hatásvonalának a távolságával. Ezt bárki saját kezével megtapasztalhatja, ha összeveti egy kis és egy nagy kormány forgatásához szükséges erőt, vagy amikor sörnyitót, illetve feszítővasat (nagy erőkart) használ egy üveg vagy egy ajtó felnyitásához. Ha az erő iránya nem merőleges az erőkarra, akkor csak az erő merőleges komponense járul hozzá a forgatónyomatékhoz (egy ajtó kinyitásakor az erőt megpróbáljuk merőlegesen tartani, hiszen ha az ajtót a zsanérok felé tuszkolnánk, azzal nem sokra mennénk). A forgatónyomaték tehát ebben az esetben az erő nagyságának, az erőkar hosszának és az erő és az erőkar közötti szög szinuszának a szorzata (9.24. ábra).



9.24. ábra. *A forgatónyomaték*

Ezt a vektoriális szorzás tulajdonságai szerint is kifejezhetjük, és az $\vec{M}$ forgatónyomatékot a következőképpen definiálhatjuk:

$$\vec{M} = \vec{r} \times \vec{F},$$

ahol az $\vec{r}$ a forgatás középpontjából az erő támadási pontjába mutató vektor, az $\vec{F}$ pedig a testet támadó erő. A forgatónyomaték is vektormennyiség, amely merőleges a kiváltott forgás síkjára.

A forgatás szempontjából tehát az erők nem egyenrangúak, hanem annál nagyobb hatást gyakorolnak, minél távolabb hatnak a forgástengelytől. A lendület (mint az összegyűjtött erő) analógiájára érdemes összegyűjtött forgatóhatásról is beszélni. Az összegyűjtött forgatóhatást a

$$\vec{J} = \vec{r} \times \vec{I} = \vec{r} \times (m\vec{v})$$

összefüggéssel definiáljuk, és *perdület*nek vagy *impulzusmomentum*nak (*angular momentum*) nevezzük. A forgatónyomaték és az impulzusmomentum nem abszolút mennyiségek, hanem függnek attól, hogy mely pontot tekintjük a forgatás középpontjának. Ezen pont megválasztása általában önkényesen történik, mégpedig úgy, hogy a számítások egyszerűek legyenek. Ha a test valamely pontja rögzített, akkor a forgatási középpontot célszerű ide elképzelni. Ha a test szabadon mozog, akkor — amint azt a következő fejezetben megmutatjuk — a tömegközéppont a legmegfelelőbb választás.

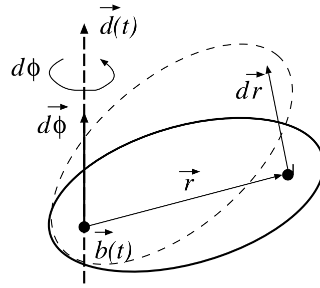
Az erők külső erőtérből, illetve a testek egymásra hatásából származhatnak. A legismertebb *erőtér* a *nehézségi erőtér*, amely „lefelé” mutat, és minden pontban állandó. Persze ez csak közelítés, ami azért elfogadható mert Földünk lényegesen nagyobb tömegű, mint a vizsgált tárgyaink, és a mozgásterünk közelében a földfelszín síknak tekinthető. Az általános esetet a *gravitáció* Newton-féle törvénye írja le, amely két test kölcsönhatásának az eredménye. A közöttük fellépő erő arányos a testek tömegével és fordítva arányos a távolságuk négyzetével. A testek további kölcsönhatásai közül a számítógépes grafikában különösen nagy jelentősége van az *ütközés*nek, amikor két test között, az egymásba hatolásukat megakadályozandó rövid időre igen nagy erők ébrednek. A szimuláció során tehát fel kell ismernünk azokat a helyzeteket, amikor ütközés történne, és a fizikai szabályok szerint ki kell számítanunk az ütközés utáni állapotot.

A következőkben először merev testek dinamikájával foglalkozunk. A merev testek olyan pontrendszerek, amelyek nem deformálódnak. A fejezet további részeit az ütközési helyzetek felismerésének és az ütközési eseményekre adott válasznak szenteljük.

9.10.2. Merev testek mozgásegyenletei

Ha a test nem pontszerű, hanem kiterjedt, akkor a *haladó mozgás*on kívül *forgó mozgás*t is végezhet. Ebben az esetben a test minden pontja más és más pályát jár be, amelyet úgy lehet leírni, hogy a test egésze haladó mozgást végez, miközben a test egy — akár időben változó — tengely körül forog.

A haladó és forgó mozgás szétválasztásának érdekében tekintsük a test egy — egyelőre tetszőleges — $\vec{b}(t)$ referencia pontját, és nevezzük el ezen pont mozgását a test *haladó mozgás*ának! Ehhez a ponthoz képest a test többi pontjának pillanatnyi helyét egy-egy $\vec{r}(t)$ helyvektorral adjuk meg. Az $\vec{r}(t)$-t *futópont*nak is nevezzük, hiszen a test bármely pontját képviselheti, azaz „végigfuthat” a test pontjain. Ha a test nem deformálódik, akkor ezen helyvektorok hossza állandó. Az ilyen testeket *merev test*eknek (*rigid body*) nevezzük. Az $\vec{r}(t)$ tehát csak úgy változhat, hogy a $\vec{b}(t)$-hez képest a forgás



9.25. ábra. *Az elfordulás jellemzése*

síkjában elfordul. Az elfordulás tengelye átmegy a referencia ponton. Az elfordulás pillanatnyi tengelyének irányát jelöljük $\vec{d}(t)$-vel, amely merőleges a forgás síkjára!

Amennyiben a test a forgás tengelye körül $d\phi$ szöggel fordul el, a test egy $\vec{r}$ pontja $d\vec{r}$-rel kerül odébb. A $d\vec{r}$ változás a forgás síkjában van és merőleges az $\vec{r}$ vektorra (9.25. ábra). A változás nagysága (vektor abszolút értéke) arányos $d\phi$-vel, azaz az elfordulás szögével, és az $\vec{r}$-nek a forgás síkjába eső vetületének hosszával, azaz az $\vec{r}$ és a forgás tengelyének távolságával. Az elfordulást egyértelműen megadhatjuk az elfordulási szöggel és a forgás tengelyével. Érdemes ezt a két dolgot összekapcsolni, és magát az elfordulást olyan vektornak tekinteni, amelynek iránya az elfordulás tengelye, nagysága pedig az elfordulás szöge. Egy tengellyel párhuzamos vektort kétféleképpen is irányíthatunk. Az egyértelműség érdekében mondjuk azt, hogy ha az elfordulásvektor felénk néz, akkor az az óramutatóval ellentétes irányú forgatásnak felel meg. Összefoglalva, a referenciaponthoz képest egy tetszőleges pont elmozdulása merőleges az elfordulás vektorra (forgástengelyre), hossza arányos az elfordulási szöggel és a futó és referencia pont távolságának a forgás síkjába eső vetületével. Ezt a vektoriális szorzat jelöléseivel így fejezhetjük ki:

$$d\vec{r} = d\vec{\phi} \times \vec{r}.$$

Az elfordulási szög egységnyi idő alatti változását (deriváltját) *szögsebesség*nek nevezzük és $\vec{\omega}$-val jelöljük:

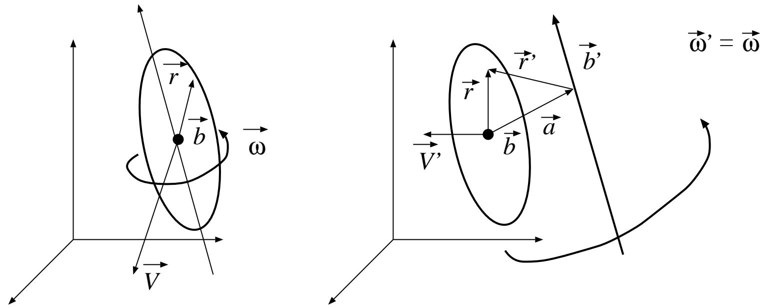
$$\frac{d\vec{\phi}}{dt} = \vec{\omega}.$$

A szögsebesség felhasználásával a futópontnak a referenciaponthoz viszonyított sebessége:

$$\frac{d\vec{r}(t)}{dt} = \vec{\omega} \times \vec{r}.$$

Ha a $\vec{b}$ referenciapont sebessége $\vec{V}$, a futópontunknak koordinátarendszerben mért sebessége:

$$\vec{v} = \vec{V} + \vec{\omega} \times \vec{r}. \tag{9.17}$$



9.26. ábra. *A szögsebesség független a forgástengely helyétől*

Fontos megjegyeznünk, hogy a szögsebesség a test mozgására jellemző és független a referenciapont megválasztásától (9.26. ábra). Ezen állítás belátásához vegyünk fel egy, a $\vec{b}$ referencia ponttól különböző $\vec{b}' = \vec{b} + \vec{a}$ referenciapontot, és jelöljük ezen pont koordinátarendszerbeli sebességét $\vec{V}'$-vel, egy tetszőleges $\vec{r}'$ pont szögsebességét pedig $\vec{\omega}'$-vel. A pontnak az új referenciaponthoz viszonyított helyét az eredeti referenciaponthoz mért helyéből is kifejezhetjük: $\vec{r}' = \vec{r} - \vec{a}$. A koordinátarendszerben a futópont sebessége nyilván független a referenciapont megválasztásától, így a 9.17. egyenletet a két esetre felírva az alábbi egyenlőséghez jutunk:

$$\vec{v} = \vec{V} + \vec{\omega} \times \vec{r} = \vec{V}' + \vec{\omega}' \times \vec{r}'.$$

Az $\vec{r}$ vektorra az $\vec{r} = \vec{r}' + \vec{a}$ helyettesítéssel:

$$\vec{V} + \vec{\omega} \times \vec{a} + \vec{\omega} \times \vec{r}' = \vec{V}' + \vec{\omega}' \times \vec{r}'.$$

Ez az egyenlőség minden $\vec{r}'$ helyvektorra fennáll, ami csak akkor lehetséges, ha az $\vec{r}'$-től függő és független részek külön-külön is egyenlőek, amelyből azt kapjuk, hogy

$$\vec{V}' = \vec{V} + \vec{\omega} \times \vec{a}, \quad \vec{\omega}' = \vec{\omega}.$$

A második egyenlet éppen a bizonyítandó állítást tartalmazza, a szögsebesség tehát valóban független a referenciapont megválasztásától.

Eddig a $\vec{b}$ referenciapontot — amely alapján a test mozgását haladó és forgó mozgásra bontjuk — teljesen szabadon vettük fel, így a levezetett összefüggések a referenciapont bármilyen választása mellett is igazak maradnak. A továbbiakban azonban érdemes a forgás referenciapontját úgy megválasztani, hogy a képleteink a lehető legegyszerűbbek maradjanak. Már megállapítottuk, hogy a test tömegközéppontja úgy mozog, mintha a test teljes tömege ebben a pontban lenne összesűrítve. A tömegközéppont eme tulajdonsága miatt érdemes a referenciapontot a tömegközéppontba (súlypontba) tenni, ez ugyanis jelentősen egyszerűsíti a további mozgásegyenleteket. A továbbiakban tehát feltételezzük, hogy a referenciapont a test tömegközéppontja, így a haladó

mozgás a tömegközéppont mozgása, valamint azt is, hogy ezt a pontot választjuk a koordinátarendszerünk origójának.

A tömegközéppont körüli forgás leírásához tekintsük az egyes tömegpontok perdületeinek az összegét, amelyet a test perdületének nevezünk:

$$\vec{J} = \sum J_i = \sum m_i \vec{r}_i \times \vec{v}_i.$$

A sebesség helyére a 9.17. egyenlet alapján helyettesítsük be a súlypont haladó mozgásának $\vec{V}$ sebességét és a súlypont körüli forgás $\vec{\omega}$ szögsebességét:

$$\vec{J} = \sum m_i \vec{r}_i \times (\vec{V} + \vec{\omega} \times \vec{r}_i) = \sum m_i \vec{r}_i \times \vec{V} + \sum m_i \vec{r}_i \times \vec{\omega} \times \vec{r}_i.$$

A jobb oldalon álló első tag a következő alakban írható fel:

$$\sum m_i \vec{r}_i \times \vec{V} = m \cdot \frac{\sum m_i r_i}{m} \times \vec{V}.$$

Vegyük észre, hogy a vektoriális szorzat első tényezője éppen a súlypont, amit gondosan az origóba tettünk, így ez a tag zérus! Az impulzusmomentum tehát:

$$\vec{J} = \sum m_i \vec{r}_i \times \vec{\omega} \times \vec{r}_i.$$

A vektoriális szorzat *antiszimmetrikus* ($\vec{a} \times \vec{b} = -\vec{b} \times \vec{a}$), így az impulzusmomentumot a következő alakban is kifejezhetjük:

$$\vec{J} = \sum m_i (\vec{r}_i \times \vec{\omega}) \times \vec{r}_i = \sum m_i (-\vec{r}_i) \times (\vec{r}_i \times \vec{\omega}). \tag{9.18}$$

Tekintsük a második vektoriális szorzatot, amelynek kifejtésekor az $\vec{r}_i$ koordinátáit x_i, y_i, z_i-vel az $\vec{\omega}$ elemeit pedig $\omega_x, \omega_y, \omega_z$-vel jelöljük! A művelet egy mátrixszorzással is felírható[1]:

$$\vec{r}_i \times \vec{\omega} = [y_i\omega_z - z_i\omega_y,\ z_i\omega_x - x_i\omega_z,\ x_i\omega_y - y_i\omega_x] =$$
$$\begin{bmatrix} 0 & -z_i & y_i \\ z_i & 0 & -x_i \\ -y_i & x_i & 0 \end{bmatrix} \cdot \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}.$$

A képletben szereplő mátrix az $\vec{r}_i$-vel képzett vektoriális szorzásért felelős. Jelöljük ezt a mátrixot a következő módon:

$$[\vec{r}_i \times] = \begin{bmatrix} 0 & -z_i & y_i \\ z_i & 0 & -x_i \\ -y_i & x_i & 0 \end{bmatrix}. \tag{9.19}$$

[1] A korábbi fejezetekben a vektorokat sorvektornak tekintettük, most viszont — a fizika hagyományait követve — oszlopvektorokkal dolgozunk. Az oszlopvektorokat pedig egy mátrixszal balról kell szorozni.

A 9.18. egyenletbeli impulzusmomentumban egy második vektoriális szorzás is felbukkan, de most az előző szorzatot $-\vec{r}_i$-vel kell szorozni. Vegyük észre, hogy ha a 9.19. egyenlet mátrixában az $[x_i, y_i, z_i]$ értékeket -1-gyel megszorozzuk, akkor egy ugyanolyan mátrixhoz jutunk, mintha a mátrixot a főátlójára tükröztük, azaz transzponáltuk volna (az ilyen mátrixokat *antiszimmetrikus*nak nevezzük)! Ezt a felismerést felhasználva a 9.18. egyenletben az impulzusmomentumot mátrixszorzásokkal is felírhatjuk:

$$\vec{J} = \sum m_i [\vec{r}_i \times]^T \cdot [\vec{r}_i \times] \cdot \vec{\omega} = \Theta \cdot \vec{\omega}, \qquad (9.20)$$

ahol

$$\Theta = \sum m_i [\vec{r}_i \times]^T \cdot [\vec{r}_i \times]$$

a 3×3-as *tehetetlenségi mátrix*. A mátrix elemeit a mátrixszorzás szabályai szerint számíthatjuk ki:

$$\Theta = \begin{bmatrix} \sum m_i(y_i^2 + z_i^2) & -\sum m_i x_i y_i & -\sum m_i x_i z_i \\ -\sum m_i y_i x_i & \sum m_i(x_i^2 + z_i^2) & -\sum m_i y_i z_i \\ -\sum m_i z_i x_i & -\sum m_i z_i y_i & \sum m_i(x_i^2 + y_i^2) \end{bmatrix}. \qquad (9.21)$$

Ha a test anyaga folytonos, akkor az összegek helyett a test anyagának $\rho(x,y,z)$ sűrűségét kell integrálnunk:

$$\Theta = \begin{bmatrix} \int_V \rho \cdot (y^2 + z^2)\, dxdydz & -\int_V \rho \cdot xy\, dxdydz & -\int_V \rho \cdot xz\, dxdydz \\ -\int_V \rho \cdot yx\, dxdydz & \int_V \rho \cdot (x^2 + z^2)\, dxdydz & -\int_V \rho \cdot yz\, dxdydz \\ -\int_V \rho \cdot zx\, dxdydz & -\int_V \rho \cdot zy\, dxdydz & \int_V \rho \cdot (x^2 + y^2)\, dxdydz \end{bmatrix}. \qquad (9.22)$$

Az impulzus és a sebesség között egyszerű arányosság áll fenn, ahol az arányossági tényező a test tömege. A 9.20. egyenlet szerint az impulzusmomentumot viszont egy mátrix kapcsolja a szögsebességhez. Tehát amíg a sebességvektor és az impulzusvektor mindig párhuzamos, az impulzusmomentum-vektorra és a szögsebesség-vektorra ez nem feltétlenül áll fenn.

Azokat a speciális forgástengelyeket, amelyekre az impulzusmomentum-vektor és a szögsebesség-vektor ugyanabba az irányba mutat, *fő tehetetlenségi irány*oknak nevezzük. Ezekben a speciális esetekben az impulzusmomentumot és a szögsebességet egyetlen skalár arányossági tényező kapcsolja össze, amit *tehetetlenségi együttható*nak nevezünk. A 9.1. táblázatban néhány fontos test tehetetlenségi együtthatóját adtuk meg a figyelembe vett, a test súlypontján átmenő forgástengellyel együtt.

A dinamika alaptörvényéből ($\vec{F} = m \cdot \vec{a}$) közvetlenül következett, hogy az impulzus deriváltja a testre ható erővel egyezik meg. Az alábbiakban bebizonyítjuk, hogy az impulzusmomentum (perdület) deriváltja pedig a forgatónyomatékot adja. Írjuk fel tehát az impulzusmomentum deriváltját és alkalmazzuk a szorzat deriválási szabályát a vektoriális szorzatra:

$$\frac{d\vec{J}}{dt} = \frac{d(\sum \vec{r}_i \times \vec{I}_i)}{dt} = \sum \frac{d\vec{r}_i}{dt} \times \vec{I}_i + \sum \vec{r}_i \times \frac{d\vec{I}_i}{dt}.$$

test	tengely	Θ
henger (sugár R, magasság h)	szimmetriatengely	$mR^2/2$
henger (sugár R, magasság h)	szimmetriatengelyre merőleges	$mR^2/4 + mh^2/12$
téglatest (élhosszúság a, b, c)	a c éllel párhuzamos tengely	$m(a^2 + b^2)/12$
gömb (sugár R)	bármelyik súlypont tengely	$2mR^2/5$
ellipszoid (tengelyek $2a$, $2b$, $2c$)	a c tengely	$m(a^2 + b^2)/5$
kúp (sugár R, magaság h)	szimmetriatengely	$3mR^2/10$

9.1. táblázat. *Néhány m tömegű homogén test tehetetlenségi együtthatója [25]*

Az első tagban a $d\vec{r}_i/dt$ derivált a pont sebessége, az impulzus pedig ezzel a sebességvektorral arányos. A vektoriális szorzás két párhuzamos vektorhoz zérust rendel, így az első tag eltűnik. A második tagban a második Newton-törvény szerint az impulzus deriváltja a pontra ható $\vec{f}_i$ erő. Összefoglalva, az impulzusmomentum deriváltja:

$$\frac{d\vec{J}}{dt} = \sum \vec{r}_i \times \vec{f}_i = \vec{M}$$

éppen a teljes forgatónyomaték. Az erőhöz hasonlóan a hatás–ellenhatás miatt az $\vec{M}$ forgatónyomatékban csak a külső erők okozta nyomaték szerepel. A belső erők nyomatékai kölcsönösen kioltják egymást. Vegyük észre, hogy ez azt is jelenti, hogy zárt rendszerben, ahol nincs külső forgatónyomaték, az impulzusmomentum állandó! Ez az *impulzusmomentum megmaradás törvénye* (*perdületmegmaradás törvénye*).

9.10.3. A tehetetlenségi mátrix tulajdonságai

A dinamikai szimulációhoz szükségünk van a test tehetetlenségi mátrixára, amit a 9.21. és a 9.22. egyenletek alapján számíthatunk ki. A test tömegének megadását általában az animátortól várjuk, a tehetetlenségi mátrixot azonban már célszerű programmal számítani. Erre annál is inkább szükség van, mert a tehetetlenségi mátrix nem csupán a testtől, hanem a forgási tengelytől, vagy más szemszögből a test orientációjától is függ.

A 9.1. táblázatban néhány fontosabb alakzat tehetetlenségi nyomatékát adtuk meg. A táblázat alapján a test tehetetlenségi mátrixát is felírhatjuk olyan esetekben, amikor a

koordinátarendszer tengelyei a fő tehetetlenségi irányok. Például egy origó középpontú, a tengelyekkel párhuzamos a, b, c oldalú téglatest (doboz) tehetetlenségi mátrixa:

$$\Theta_{doboz}(a,b,c) = \begin{bmatrix} m(b^2+c^2)/12 & 0 & 0 \\ 0 & m(a^2+c^2)/12 & 0 \\ 0 & 0 & m(a^2+b^2)/12 \end{bmatrix}.$$

Amennyiben a forgástengely továbbra is párhuzamos valamely fő tehetetlenségi iránnyal, de nem megy át a súlyponton, a *Steiner-tétel*t használhatjuk a tehetetlenségi együttható meghatározására:

$$\Theta_s = \Theta_0 + m \cdot s^2,$$

ahol s a forgástengely és a test tömegközéppontjának távolsága, Θ_s az s távolságra tolt test tehetetlenségi együtthatója, Θ_0 a súlyponton átmenő forgástengelyre mért tehetetlenségi együttható, m pedig a test tömege. Ha egyszerűbb testekből rakunk össze bonyolultabb testeket, felhasználhatjuk a tehetetlenségi mátrix additív tulajdonságát, azaz azt, hogy a test tehetetlenségi mátrixa megegyezik a részek tehetetlenségi mátrixaiból képzett összeggel. A Steiner-tétel és az additivitás közvetlenül bizonyítható a 9.21. egyenletből. Ha ezekkel a trükkökkel sem érünk célba, akkor közvetlenül a 9.21. egyenletet kell alkalmazni a tehetetlenségi mátrix kiszámítására.

A tehetetlenségi mátrix a test orientációjától is függ. Szerencsére nem kell minden orientációváltozáskor teljesen nulláról kezdeni a tehetetlenségi mátrix kiszámítását, ugyanis ha azt valamilyen orientációra ismerjük, akkor ebből tetszőleges más orientációra is átszámítható. Vegyük szemügyre a tehetetlenségi mátrix definícióját:

$$\Theta = \sum m_i [\vec{r}_i \times]^T \cdot [\vec{r}_i \times].$$

Most tételezzük fel, hogy a testre egy forgatási transzformációt alkalmazunk, azaz az $\vec{r}_i$ pont a következő $\vec{r}_i'$ pontba megy át:

$$\vec{r}_i' = \vec{r}_i \cdot \mathbf{A}.$$

Ebben az esetben az új tehetetlenségi mátrix a következőképpen számítható (a részletes bizonyítás megtalálható a [16]-ben):

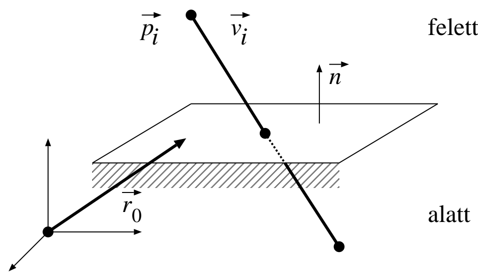
$$\Theta' = \sum m_i \cdot [\vec{r}_i' \times]^T \cdot [\vec{r}_i' \times] = \mathbf{A}^T \cdot \sum m_i \cdot [\vec{r}_i \times]^T \cdot [\vec{r}_i \times] \cdot \mathbf{A} = \mathbf{A}^T \cdot \Theta \cdot \mathbf{A}.$$

Ez azt jelenti, hogy a tehetetlenségi mátrix orientációváltozásból eredő módosulását követhetjük, ha az orientációs mátrixokkal balról és jobbról szorozzuk azt. Célszerű a fő tehetetlenségi irányokra ismert mátrixból kiindulni, ugyanis ekkor a mátrix diagonális, azaz csak a főátlójában vannak nem zérus elemek.

9.10.4. Ütközésdetektálás

Az ütközésdetektálás azokat az időpillanatokat és helyzeteket azonosítja, amikor két objektum ütközik egymással. Az animációval követett rendszer folyamatosan mozog, amelyre csak diszkrét időpontokban tekintünk. Az ütközések bármikor bekövetkezhetnek, elvileg két diszkrét időpont között is, ezért akkor járunk el helyesen, ha az ütközés számításánál nem csak az objektumoknak a diszkrét időpontokban felvett állapotát vizsgáljuk, hanem figyelemmel kísérjük a következő diszkrét időpontig tartó mozgásukat is. Ezt a megközelítést *folytonos ütközésdetektálás*nak nevezzük. Amennyiben az objektumok lassan mozognak (legalábbis a méretükhöz képest), elegendő ha csak a diszkrét időpontokban ellenőrizzük, hogy nem hatoltak-e egymásba. Ez az egyszerűbb, közelítő módszer a *diszkrét ütközésdetektálás*.

Pont–féltér ütközésdetektálás



9.27. ábra. *Pont–féltér ütközésdetektálás*

Tekintsük először azt az esetet, amikor az egyik test egy mozgó pont, a másik pedig egy $\vec{n}$ normálvektorú, $\vec{r}_0$ helyvektorú mozdulatlan síkkal határolt féltér! Feltételezzük, hogy a normálvektor a féltérből kifelé mutat. Jelöljük az i-edik időkeret kezdetét t_i-vel a végét t_{i+1}-gyel! A pont helye az időkeret kezdetén $\vec{p}_i$, a sebessége pedig az időkeretben végig $\vec{v}_i$. A pontszerű objektum pályája:

$$\vec{p}(t) = \vec{p}_i + \vec{v}_i \cdot t.$$

Az $\vec{r}$ pontok síktól mért

$$d(\vec{r}) = \vec{n} \cdot (\vec{r} - \vec{r}_0)$$

előjeles távolsága a tér pontjait három részhalmazra bontja. Azok az $\vec{r}$ pontok, amelyekre ez a kifejezés pozitív, a sík „felett” vannak, azok, amelyekre az érték zérus, a síkra esnek, míg azon pontok, amelyekre a kifejezés negatív, a sík „alatt” helyezkednek el. Ennek alapján mindaddig nincs ütközés, amíg a $d(\vec{p})$ előjeles távolság pozitív.

Ha *diszkrét ütközésdetektálási megközelítést* alkalmazunk, akkor minden vizsgált időpillanatban ellenőrizzük a $d(\vec{p}_i)$ előjelét, és ha ez történetesen nem pozitív, akkor ütközésért kiáltunk.

A *folytonos ütközésdetektálási eljárás* a következőképpen dolgozik. Ha a keretidőre semmilyen korlátozás sem lenne, akkor az ütközés akkor következik be, amikor a

$$d(\vec{p}(t)) = \vec{n} \cdot (\vec{p}_i + \vec{v}_i \cdot t - \vec{r}_0) = 0,$$

egyenlőség teljesül, amiből az ütközés ideje:

$$t^* = \frac{\vec{n} \cdot (\vec{r}_0 - \vec{p}_i)}{\vec{n} \cdot \vec{v}_i}.$$

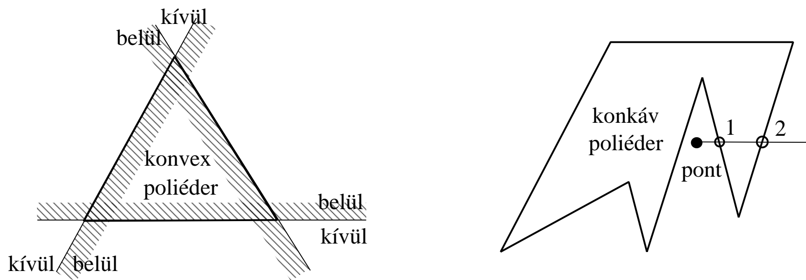
Előfordulhat, hogy ez az érték negatív, vagy nagyobb a keretidőnél, ami nyilván nem jelent ütközést. Ha viszont $0 \leq t^* \leq t_{i+1} - t_i$, akkor ismét ütközést jelenthetünk, sőt most a pontos idejével is tisztában vagyunk.

A megoldás során feltételeztük, hogy a második objektum áll (a diszkrét eljárás független attól, hogy mozognak-e az objektumok, úgyis csak kijelölt időpillanatokban néz rájuk). Az általánosítás a folytonos módszerre kézenfekvő abban az esetben, amikor a második objektum csak haladó mozgást végez. Ekkor ugyanis csak egy olyan, alkalmas koordinátarendszert kell találni, ahol a második objektum áll, tehát a relatív sebességekkel kell dolgoznunk. Ha az objektum csak haladó mozgást végez, akkor a pontszerű objektum sebességvektorából le kell vonnunk a másik objektum sebességvektorát. Forgásnál már nehezebb helyzetben volnánk, ezért hacsak az objektumok nem gömb alakúak, a forgó alakzatok ütközésvizsgálatát diszkrét megközelítéssel kezeljük.

Pont–poliéder ütközésvizsgálat

Most térjünk rá a diszkrét ütközésvizsgálat azon általánosabb esetére, amikor a második objektum egy konvex, illetve egy konkáv poliéder! Egy konvex poliéder előállítható a lapjaira illeszkedő síkok által határolt félterek metszeteként (9.28. ábra bal oldala). Minden lap síkja tehát a teret két részre bontja, egy „jó” oldalra, amelyikben maga a poliéder található, és egy „rossz” oldalra. Vessük össze a pontot a poliéder lapjaival, pontosabban azok síkjaival! Ha a pontunk minden sík tekintetében a jó oldalon van, a pont a poliéderen belül van, tehát ütközés következett be, ha viszont valamely sík esetén a rossz oldalon van, a pont nem lehet a poliéder belsejében.

Konkáv poliéderekre egy kicsit többet kell számolnunk (9.28. ábra jobb oldala). Indítsunk egy félegyenest a vizsgált pontból a végtelen felé, és próbáljuk elmetszeni a poliéder lapjait (a metszéspontok számításához a sugárkövetéshez kidolgozott, a 6. fejezetben megismert eljárások használhatók)! Ha páratlan számú metszéspontot számolunk össze, akkor a poliéder belsejében, egyébként pedig azon kívül van a pontunk.

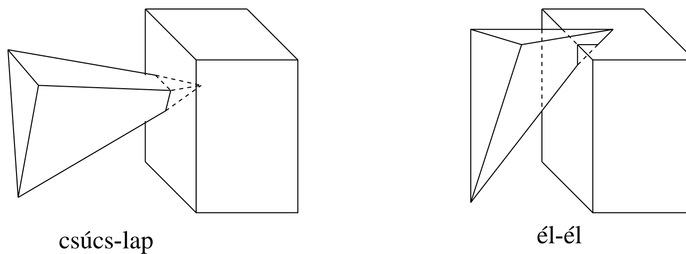


9.28. ábra. *Pont–poliéder ütközésvizsgálat*

A numerikus pontatlanságok miatt a lapok találkozásánál gondot jelenthet annak eldöntése, hogy félegyenesünk itt hány lapot is metszett egyszerre. Ha ilyen helyzetbe kerülünk, akkor a legegyszerűbb egy olyan új félegyenest választani, amely elkerüli a lapok találkozását.

A folytonos ütközésvizsgálat elvégzéséhez a pontszerű objektum pályáját sugárnak tekintve, a sugárkövetés algoritmusainak felhasználásával meghatározzuk a lapok és a sugár metszéspontjait. A metszéspontok sugárparaméterei közül megkeressük a legkisebbet. Amennyiben ez az érték létezik, nem negatív, és kisebb a keretidőnél, úgy ütközés következik be.

Poliéder–poliéder ütközésvizsgálat



9.29. ábra. *Poliéder–poliéder ütközésvizsgálat*

Két általános poliéder ütközhet egymással úgy, hogy az egyikük egy csúcsa a másik belsejébe hatol (9.29. ábra bal oldala). Ez az eset a korábbi módszerekkel megoldható. Először az első poliéder összes csúcsára ellenőrizzük, hogy behatol-e a második poliéderbe, majd a két poliéder szerepét felcserélve vizsgáljuk, hogy a második csúcsai ütköznek-e az első lapjaival.

A csúccsal történő ütközésen kívül előfordulhat, hogy két poliéder élei a másikba hatolnak anélkül, hogy a másik csúcsai belülre kerülnének (9.29. ábra jobb oldala). Az él–él metszés eldöntéséhez az egyik poliéder összes élét össze kell vetni a második poliéder összes lapjával. Egy él és lap tekintetében először ellenőrizzük, hogy az él két végpontja a lap síkjának két ellentétes oldalán van-e. Ha igen, akkor kiszámítjuk az él és a lap síkjának a metszéspontját, végül pedig eldöntjük, hogy a metszéspont a lapon belül van-e. Konvex lapoknál ellenőrizhetjük, hogy a pontból a lap éleinek a látószögét összegezve 360 fokot kapunk-e, vagy megvizsgálhatjuk, hogy minden élre a pont ugyanazon az oldalon van-e, mint a lap többi csúcspontja (3.4.1. fejezet). Konkáv lapoknál a pont–konkáv poliéder ütközésvizsgálatának kétdimenziós változatát használhatjuk. A pontból egy félegyenest indítunk a lap síkján a végtelen felé, és megszámoljuk az élekkel képzett metszéspontokat. Ha az éleket páratlan sokszor metsszük, akkor a pont belül van, azaz ütközés következett be.

Vegyük észre, hogy az él–él metszés magában foglalja a csúcs behatolás esetét is, tehát annak vizsgálata szükségtelennek látszik! Azonban a csúcs behatolását kevesebb számítással is felismerhetjük, így érdemes először ezt vizsgálni.

Az ütközésszámítás gyorsítása

A poliéderek ütközésvizsgálata során az egyik poliéder összes élét a másik poliéder összes lapjával össze kell vetni, amely bonyolultabb színterekben meglehetősen sokáig tarthat. Szerencsére a módszer a befoglaló térfogatok elvének alkalmazásával jelentősen gyorsítható (6.4. fejezet). Keressünk minden objektumhoz egy olyan egyszerű alakzatot, amely tartalmazza azt. Különösen népszerűek a 9.30. ábrán is látható *befoglaló gömb*ök (*bounding sphere*), vagy a koordinátatengelyekkel párhuzamos élű befoglaló téglatestek (*axis aligned bounding box (AABB)*).

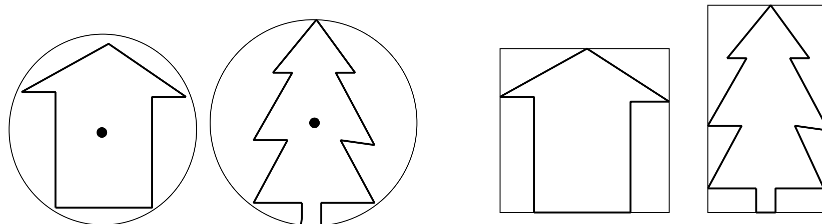
A befoglaló gömb előállításának algoritmusa a középpont megkeresésével kezdődik. Ez lehet a test tömegközéppontja, vagy a minimális és maximális x, y, z koordináták felezőpontja. Ezután sorra vesszük a csúcspontokat, és kiszámítjuk a csúcspontok és a középpont maximális távolságát. A gömb sugara ez a távolság lesz.

Az AABB előállítása még egyszerűbb, a csúcspontok maximális és minimális x, y, z koordinátái alkotják a téglatest két szemközti csúcsának koordinátáit.

Az ütközésvizsgálatot először a befoglaló alakzatokra végezzük el. Két gömb akkor metszi egymást, ha a középpontjaik távolsága a sugaruk összegénél kisebb. Az $[x^1_{\min}, y^1_{\min}, z^1_{\min}, x^1_{\max}, y^1_{\max}, z^1_{\max}]$ és $[x^2_{\min}, y^2_{\min}, z^2_{\min}, x^2_{\max}, y^2_{\max}, z^2_{\max}]$ AABB pedig akkor hatol egymásba, ha valamennyi alábbi egyenlőtlenség fennáll:

$$x^1_{\min} \le x^2_{\max}, \;\; x^1_{\max} \ge x^2_{\min}, \;\; y^1_{\min} \le y^2_{\max}, \;\; y^1_{\max} \ge y^2_{\min}, \;\; z^1_{\min} \le z^2_{\max}, \;\; z^1_{\max} \ge z^2_{\min}.$$

Ha a befoglaló alakzatok nem találkoznak, akkor nyilván a befoglalt objektumok sem ütközhetnek. Amennyiben a befoglaló alakzatok egymásba hatolnak, akkor folytatni



9.30. ábra. *Befoglaló gömbök és AABB-k*

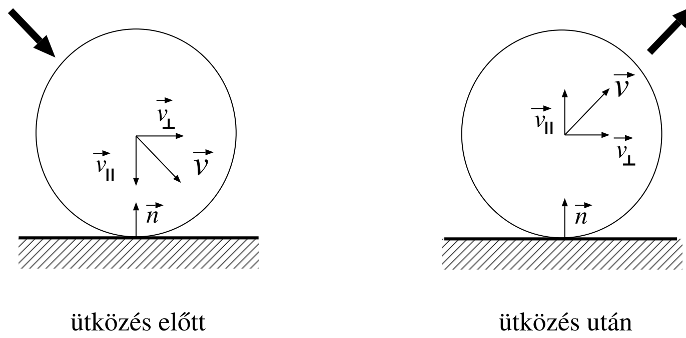
kell a vizsgálatot. Az egyik objektumot összevetjük a másik befoglaló alakzatával, és ha itt is ütközés mutatkozik, akkor magával az objektummal. Remélhetőleg ezen utóbbi eset nagyon ritkán fordul elő, és az ütközésvizsgálatok döntő részét a befoglaló alakzatokkal gyorsan el lehet intézni.

9.10.5. Ütközésválasz

Az ütközések rövid idő alatt zajlanak le, és igen nagy erőket ébresztenek. Az ütközések következményeinek számításánál két megközelítést alkalmazhatunk. Az első megközelítés az ütközést folyamatában vizsgálja, és figyelemmel kíséri, hogy a két test érintkezése következtében milyen erők és mekkora ideig lépnek fel. Annak érdekében, hogy ne kelljen a merev test modellel szakítani, ezek az eljárások feltételezik, hogy a két közeli test közé rugók kerülnek, amelyek a merev testek helyett deformálódnak, és végül visszalökik az ütköző testeket. Bár az elképzelés elegánsnak tűnik, a gyakorlatban mégsem szívesen használják. Nagyon nehéz ugyanis a rugómodell állandóit megfelelően megválasztani. Ha a rugók túl erősek, a testek még azelőtt visszapattannak, hogy elérték volna a másik tárgyat. Ha viszont túl gyengék, a testek egymásba hatolnak.

A második megközelítés nem él ilyen absztrakt rugóképpel, sőt nem is vizsgálja az ütközés folyamatát részleteiben. Az eljárás, amelyet *impulzus alapú ütközésválasz*nak nevezünk, arra koncentrál, hogy mi lesz a testek mozgásállapota az ütközés után. Mivel az ütközés nagyon gyorsan (a merev testek feltételezés esetében végtelen gyorsan) zajlik le, ez éppen elegendő, és az ütközés alatti folyamatok részletei nem érdekesek. Kérdés persze, hogy el lehet-e dönteni, hogy hogyan viselkednek a testek az ütközés után, anélkül, hogy vizsgálnánk az ütközés folyamatát. A válasz szerencsére igen, mégpedig a mechanika megmaradási tételei alapján.

Mielőtt az általános esetre rátérnénk, nézzünk meg egy egyszerű, ámde igen fontos példát, amikor egy gömb síklapokkal határolt térrészben pattog! A labda minden pontjának sebessége $\vec{v}$. A síklapok rögzítettek, tehát nem mozdulhatnak el. Amikor a labda egy pontja ütközik valamelyik síklappal, az ütközés gyorsan, Δt idő alatt következik be, mialatt átlagosan $\vec{F}$ erő ébred a két test között. A labdánk ezalatt $\Delta\vec{I} = \vec{F}\Delta t$ impulzust



9.31. ábra. *A pattanó labda*

kap. Ha a súrlódástól eltekintünk, az $\vec{F}$ erő csak az érintkező felületekre merőlegesen ébredhet, tehát az $\vec{F}$ erő iránya a síklapnak és a gömbnek az ütközési pontbeli közös normálvektora. Ezt az $\vec{n}$ vektort *ütközési normális*nak nevezzük, és általában feltételezzük, hogy hossza egységnyi. Bontsuk fel a $\vec{v}$ sebességvektort az ütközési normálissal, azaz az erővel párhuzamos ($\vec{v}_{||}$) és merőleges ($\vec{v}_\perp$) komponensekre:

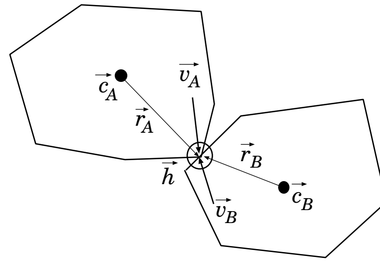
$$\vec{v}_{||} = \vec{n}(\vec{n}\cdot\vec{v}), \quad \vec{v}_\perp = \vec{v} - \vec{v}_{||}.$$

Ezek a formulák csak akkor érvényesek, ha az $\vec{n}$ normálvektor egységvektor.

Az erőre merőlegesen nincs gyorsulás, tehát a sebességvektor merőleges komponense változatlan. A párhuzamos komponens viszont visszájára fordul, hiszen a közeledő mozgásból az ütközés után távolodó mozgás lesz. A kérdés tehát csak az, hogy milyen nagy az új párhuzamos sebességkomponens. Ha az ütközés tökéletesen rugalmas, azaz a mechanikai energia megmarad, az ütközés után az új sebességvektor hossza a régivel megegyező, ami csak akkor lehetséges, ha az új párhuzamos sebességvektor éppen ellentettje a réginek. Ez azt jelenti, hogy a teljes új sebességvektor a réginek a normálvektorra vett tükörképe. A biliárdgolyók jó közelítéssel így pattannak vissza az asztal oldalfaláról. A természetben az ütközés nem teljesen rugalmas, azaz egy kis mechanikai energia hő formájában távozik a rendszerből. Ezt úgy vehetjük figyelembe, hogy az új párhuzamos sebességvektor hosszát egy $b \leq 1$ *rugalmassági tényező*vel (*bounciness*) megszorozzuk. Az ütközés utáni állapot:

$$\vec{v}'_{||} = -b\cdot\vec{v}_{||}, \quad \vec{v}'_\perp = \vec{v}_\perp, \quad \vec{v}' = \vec{v}'_{||} + \vec{v}'_\perp.$$

A labdás példa tapasztalatainak birtokában áttérünk az általános eset vizsgálatára, amikor a két test geometriája tetszőleges, és egyik test sem rögzített (a rögzített testet is rögzítetlennek tekinthetjük, ha tömegét és tehetetlenségi együtthatóit végtelenre állítjuk). A két ütköző test jellemzőit a 9.32. ábrán láthatjuk. A két test tömegközéppontja,



9.32. ábra. *Két test ütközése*

haladó mozgásának sebessége és szögsebessége, valamint a testek tömege legyen rendre $\vec{c}_A, \vec{V}_A, \vec{\omega}_A$, m_A és $\vec{c}_B, \vec{V}_B, \vec{\omega}_B$, m_B. Az ütközés a $\vec{h}$ pontban következik be. A testek tömegközéppontjaihoz képest tehát az ütközési pont helye:

$$\vec{r}_A = \vec{h} - \vec{c}_A, \quad \vec{r}_B = \vec{h} - \vec{c}_B.$$

Az ütközést megelőző pillanatban a két test $\vec{h}$-ban tartózkodó pontjai éppen

$$\vec{v}_A = \vec{V}_A + \vec{\omega}_A \times \vec{r}_A, \quad \vec{v}_B = \vec{V}_B + \vec{\omega}_B \times \vec{r}_B$$

sebességgel haladnak. Az ütközés gyorsan, Δt idő alatt következik be, mialatt átlagosan $\vec{F}$ erő ébred a két test között (a hatás–ellenhatás törvénye miatt, ha az A testre $\vec{F}$ átlagos erő hat, akkor a B testre $-\vec{F}$). Ha a súrlódástól eltekintünk, az $\vec{F}$ erő csak az érintkező felületekre merőlegesen, azaz az $\vec{n}$ *ütközési normális*sal párhuzamosan léphet fel, tehát $\vec{F} = F \cdot \vec{n}$. Pontosabban ez azt jelenti, hogy ha az egyik test egy csúcsa a másik test egy lapjával ütközik, akkor a második test lapjának normálvektora határozza meg az erő irányát. Ha a két test élei ütköznek, akkor az erő iránya mindkét élre merőleges, azaz az élek vektoriális szorzataként állítható elő. Annak a valószínűsége pedig zérus, hogy két párhuzamos él vagy két csúcspont ütközzön. Ha mégis ez az eset következne be, akkor a találkozó lapok normálvektorainak átlagával dolgozhatunk.

Az ütközés Δt ideje alatt tehát átlagosan $F \cdot \vec{n}$ illetve $-F \cdot \vec{n}$ erő hatott a testekre, és ez átlagosan $\vec{r}_A \times (F \cdot \vec{n})$ illetve $\vec{r}_B \times (-F \cdot \vec{n})$ forgatónyomatékot jelentett, ha a súlypontot tekintjük forgatási centrumnak. A két test impulzusának és impulzusmomentumának a változása az ütközés következtében:

$$\Delta\vec{I}_A = F\Delta t \cdot \vec{n}, \quad \Delta\vec{I}_B = -F\Delta t \cdot \vec{n},$$
$$\Delta\vec{J}_A = F\Delta t \cdot \vec{r}_A \times \vec{n}, \quad \Delta\vec{J}_B = -F\Delta t \cdot \vec{r}_B \times \vec{n}. \tag{9.23}$$

Az ütközés utáni haladási sebesség és a szögsebesség változását az impulzus és impulzusmomentum változásából határozhatjuk meg:

$$\Delta\vec{V}_A = \frac{\Delta\vec{I}_A}{m_A}, \quad \Delta\vec{V}_B = \frac{\Delta\vec{I}_B}{m_B}, \quad \Delta\vec{\omega}_A = \Theta_A^{-1}(t)\Delta\vec{J}_A, \quad \Delta\vec{\omega}_B = \Theta_B^{-1}(t)\Delta\vec{J}_B.$$

Ebből viszont a két összetalálkozott pont ütközés utáni sebessége:

$$\vec{v}'_A = \vec{V}'_A + \vec{\omega}'_A \times \vec{r}_A = \vec{V}_A + \frac{\Delta\vec{I}_A}{m_A} + (\vec{\omega}_A + \Theta_A^{-1}(t)\Delta\vec{J}_A) \times \vec{r}_A,$$
$$\vec{v}'_B = \vec{V}'_B + \vec{\omega}'_B \times \vec{r}_B = \vec{V}_B + \frac{\Delta\vec{I}_B}{m_B} + (\vec{\omega}_B + \Theta_B^{-1}(t)\Delta\vec{J}_B) \times \vec{r}_B. \quad (9.24)$$

Most is ugyanúgy járunk el, mint az egyszerű pattogó gömbnél. Mivel az erő az ütközési normális irányába hatott, a sebességek ütközési normálisra merőleges komponensei változatlanok. Az ütközési normálissal párhuzamos komponensekre elmondhatjuk, hogy a két test az ütközés előtti pillanatban $v = \vec{n} \cdot (\vec{v}_A - \vec{v}_B)$ sebességgel tartott egymás felé, az ütközés után a testek ezen pontjai távolodni fognak, mégpedig $-b \cdot v$ normálirányú sebességgel, ahol a b az ütközés rugalmassága. Az ütközés utáni és előtti, normálirányú sebességkülönbség tehát:

$$\vec{n} \cdot (\vec{v}'_A - \vec{v}'_B) - \vec{n} \cdot (\vec{v}_A - \vec{v}_B) = -(1+b)v.$$

A 9.24. egyenletből az ütközés utáni állapotra vonatkozó sebességeket és szögsebességeket behelyettesítve, majd a 9.23. egyenletből az impulzus és impulzusmomentum változásokat az $F\Delta t$ segítségével felírva, az ütközés $F\Delta t$ impulzusát kifejezhetjük:

$$F\Delta t = \frac{-(1+b)v}{\frac{1}{m_A} + \vec{n} \cdot (\Theta_A^{-1}(t)\vec{r}_A \times \vec{n}) \times \vec{r}_A + \frac{1}{m_B} + \vec{n} \cdot (\Theta_B^{-1}(t)\vec{r}_B \times \vec{n}) \times \vec{r}_B}.$$

Az ütközési impulzusból viszont a 9.23. egyenlet alapján előállíthatjuk a testek új impulzusát és impulzusmomentumát. Összefoglalva, a $\vec{h}$ pontban fellépő, $\vec{n}$ normálisú ütközés utáni mozgásállapot meghatározásának algoritmusa:

```
r_A = h − c_A;   r_B = h − c_B;
v_A = V_A + ω_A × r_A;   v_B = V_B + ω_B × r_B;
v = n · (v_A − v_B);
if (|v| > ε) {
    FΔt = −(1+b)v/(1/m_A + n · (Θ_A^-1 r_A × n) × r_A + 1/m_B + n · (Θ_B^-1 r_B × n) × r_B);
    I_A += FΔt · n;   I_B −= FΔt · n;
    J_A += r_A × (FΔt · n);   J_B −= r_B × (FΔt · n);
} else „nyugalmi érintkezés”;
```

A fenti programban a relatív sebességet egy kicsiny ε értékkel vetettük össze, és ha a sebesség ezt abszolút értékben nem haladta meg, akkor úgy tekintettük, hogy a két test *nyugalmi érintkezés*ben (*resting contact*) van.

Előfordulhat, hogy egyszerre több ponton is találkozik két test. Ekkor a fenti lépéseket minden pontra végre kell hajtani, majd a teljes ütközésellenőrzési eljárást mindaddig folytatni, amíg nem találunk újabb ütközési pontokat.

A *súrlódás*i erők az ütközési normálisra merőlegesen (a felület síkjában) és a sebességgel ellentétes irányban ébrednek. A súrlódási erő arányos a két testet összenyomó erővel és a súrlódási együtthatóval. A fent ismertetett eljárással az ütközés során fellépő átlagos összenyomó erő és az ütközés idejének szorzata számítható. Ezt az ütközési impulzust a súrlódási tényezővel szorozva a test ütközésre merőleges irányú impulzuscsökkenését kaphatjuk meg. A számításnál figyelni kell arra, hogy a súrlódás következtében az testek lassulhatnak, esetleg megállhatnak, de semmiképpen sem indulhatnak el az ellenkező irányba. Így az ütközésre merőleges irányú impulzus csökkenés ezen komponens előjelét nem változtathatja meg.

A testek előbb-utóbb megállnak, ami nem jelenti azt, hogy ekkor a testek között ne ébredne erő. Az asztalon nyugvó tárgyakat a nehézségi erő továbbra is lefelé húzza, de az alátámasztás kényszerereje éppen ugyanekkora erővel tolja vissza, így az eredő erők végsősoron kioltják egymást. A *kényszererő*k számítása a mechanika egy különlegesen nehéz területe [16, 100], így a számítógépes animációban ettől gyakran eltekintenek. Ez úgy lehetséges, hogy a látszólag nyugalomban lévő testeknél sok apró ütközést számítunk ki. Ha nem járunk el körültekintően, a nyugalomban lévő test indokolatlanul beremeghet, sőt táncra is perdülhet. A másik hátránya ennek a megközelítésnek, hogy nem tudunk különbséget tenni a csúszási és a tapadási súrlódási együtthatók között (a tapadási együttható mindig nagyobb) pedig ennek az autóvezetésben, és így az autószimulátorokban nagy jelentősége van [35, 76].

9.10.6. A merev testek mozgásegyenleteinek megoldása

Az előző fejezetekben a fizikai ismereteinket csiszolgattuk annak érdekében, hogy a testek pályáját a testre ható erőtér és a mozgást akadályozó és megváltoztató ütközések alapján ki tudjuk számolni. A tényleges számolási eljárással ebben a fejezetben ismerkedünk meg, amely a megismert mozgásegyenleteket az idő előrehaladtával lépésenként oldja meg. A részletek tisztázása előtt azt kell végiggondolnunk, hogy milyen ismeretek szükségesek ahhoz, hogy egy test helyzete ne csak az aktuális, hanem a jövőbeli időpillanatokban is kiszámítható legyen. Ezen ismeretek összessége a test *állapotvektor*a.

A mozgás állapotvektorának a tömegközéppont pozícióját ($\vec{c}(t)$), a pillanatnyi orientációs mátrixot ($\mathbf{A}(t)$), a test impulzusát ($\vec{I}(t)$) és impulzusmomentumát ($\vec{J}(t)$) választjuk (a tömegközéppont pozíciója és az orientációs mátrix meghatározza a test pillanatnyi helyzetét, az impulzus és impulzusmomentum pedig ahhoz kell, hogy a test jövőbeli viselkedésére is következtetni tudjunk). Feltételezzük, hogy a test teljes m tömege és Θ_{test} lokális modellezési-koordinátarendszerbeli tehetetlenségi mátrixa állandó. A lokális modellezési-koordinátarendszer középpontját a test tömegközéppontjába tesszük. Ezekkel a feltételezésekkel a test tömegközéppontjának $\vec{V}(t)$ sebessége, pillanatnyi tehetet-

lenségi mátrixa és $\vec{\omega}$ szögsebessége az állapotinformációkból meghatározható [2]:

$$\vec{V}(t) = \frac{\vec{I}(t)}{m}, \qquad \Theta(t) = \mathbf{A}^T(t) \cdot \Theta_{test} \cdot \mathbf{A}(t), \qquad \vec{\omega}(t) = \Theta^{-1}(t) \cdot \vec{J}(t).$$

A test tetszőleges, a lokális modellezési-koordinátarendszerben $\vec{r}_L$-vel definiált pontjának pillanatnyi $\vec{r}$ helyét és $\vec{v}$ sebességét a következőképpen számíthatjuk ki:

$$\vec{r}^T(t) = \vec{r}_L^T \mathbf{A} + \vec{c}^T(t), \qquad \vec{v}(t) = \vec{V}(t) + \vec{\omega} \times \vec{r}.$$

Valóban elegendő tudni a test pontjainak a pillanatnyi helyét és sebességét, azaz a helyvektort és annak első deriváltját ahhoz, hogy a test jövőbeli mozgását egyértelműen meg tudjuk határozni? A Newton-törvények szerint a test impulzusát és impulzusmomentumát erők változtathatják meg. A tapasztalatok szerint az erők csak a test pontjainak helyzetétől (gravitáció, ütközés, kényszererők, rugóerők stb.) és sebességétől (közegellenállás) függhetnek, a magasabb deriváltaktól nem. A válasz a költői kérdésre tehát az, hogy valóban elegendő a pozíció és a sebesség ismerete, így az $S(t) = [\vec{c}(t), \mathbf{A}(t), \vec{I}(t), \vec{J}(t)]$ vektor megfelelő állapotvektornak tekinthető.

Tekintsük most az állapotvektor deriváltját! A tömegközéppont helyének deriváltja a test haladó mozgásának a sebessége, amit az impulzus és a tömeg hányadosaként kaphatunk meg:

$$\frac{d\vec{c}(t)}{dt} = \vec{V}(t) = \frac{\vec{I}(t)}{m}.$$

Az orientációs mátrix deriváltjának meghatározásához felhasználjuk, hogy sorai a modellezési-koordinátarendszer bázisvektorainak transzformált (elforgatott) változatai, azaz

$$\mathbf{A} = \begin{bmatrix} \vec{x}_L^T \\ \vec{y}_L^T \\ \vec{z}_L^T \end{bmatrix}.$$

Ezen vektorok deriváltjai sebességek. Amennyiben a test tömegközéppontja az origóban van, és haladó mozgás nincsen, egy tetszőleges $\vec{r}$ pont sebessége:

$$\vec{v} = \vec{\omega} \times \vec{r}.$$

Ezt a modellezési-koordinátarendszer három bázisvektorára felírva, és felhasználva a vektoriális szorzás mátrixát (9.19. egyenlet):

$$\frac{d\mathbf{A}}{dt} = \begin{bmatrix} (\vec{\omega} \times \vec{x}_L)^T \\ (\vec{\omega} \times \vec{y}_L)^T \\ (\vec{\omega} \times \vec{z}_L)^T \end{bmatrix} = \mathbf{A} \cdot [\vec{\omega} \times]^T.$$

[2]Mivel ebben a fejezetben általában oszlopvektorokkal dolgozunk, „T” (transzponált) felsőindexszel külön jelöljük a sorvektorokat és a mátrix transzponálását.

Az impulzus deriváltja a testre ható eredő erő, amelyet feltételezésünk szerint az S állapot egyértelműen meghatároz:

$$\frac{d\vec{I}(t)}{dt} = \vec{F}(S).$$

Végül az impulzusmomentum deriváltja a forgatónyomaték:

$$\frac{d\vec{J}(t)}{dt} = \vec{M}(S).$$

Tehát, ha egy t pillanatban ismerjük az $S(t)$ mozgásállapotot, akkor annak a deriváltját is előállíthatjuk. A derivált ismeretében pedig következtethetünk a mozgásállapot későbbi $S(t+\Delta t)$ értékeire. Ezt a lépést ismételgetve (iterálva), a mozgásállapot idősora kiszámítható. Formálisan egy

$$\frac{dS(t)}{dt} = \mathcal{F}(t, S(t))$$

differenciálegyenletet oldunk meg, ahol $\mathcal{F}$ a fenti egyenletek szerint az S állapotból előállítja a sebességet, az orientációs mátrix deriváltját, az erőt és a forgatónyomatékot. Az iteráció általános sémája:

Kezdeti S állapot beállítása;
`for` $(t = t_{start};\, t \le t_{end};\, t \mathrel{+}= \Delta t)$ {
  $S \mathrel{+}= \mathcal{F}(t,S)\cdot\Delta t$;
}

A differenciálhányadosokat tehát egyszerű differenciahányadosokkal helyettesíthetjük és az időben Δt-vel előre lépkedve oldjuk meg a differenciálegyenletet. Ez az *Euler-módszer*. Összefoglalva, a dinamikai szimuláció általános algoritmusa a következő:

Kezdeti $[\vec{c}, \mathbf{A}, \vec{I}, \vec{J}]$ állapot beállítása;
`for` $(t = t_{start};\, t \le t_{end};\, t \mathrel{+}= \Delta t)$ {
  $[\vec{c}, \mathbf{A}]$ alapján rajzolás;
  Ütközésdetektálás, $\vec{F}\cdot\Delta t$ és $\vec{M}\cdot\Delta t$ számítása;
  $\vec{\omega} = \Theta^{-1}(t)\cdot\vec{J}$;
  $\vec{c} \mathrel{+}= \vec{I}(t)/m\cdot\Delta t$;
  $\mathbf{A} \mathrel{+}= \mathbf{A}\cdot[\vec{\omega}\times]^T\cdot\Delta t$;
  $\vec{I} \mathrel{+}= \vec{F}\cdot\Delta t$;
  $\vec{J} \mathrel{+}= \vec{M}\cdot\Delta t$;
}

Az algoritmus futtatásakor gondot jelent, hogy a numerikus pontatlanságok miatt az **A** mátrix egyre kevésbé elégíti ki azt a feltételt, hogy sorai egymásra merőleges

egységvektorok, ami végső soron az objektum torzulásához vezet. A torzulást elkerülhetjük, ha minden lépés után a sorvektorok merőlegességéről és normalizáltságáról külön gondoskodunk, de a számítás mindenképpen pontatlan lesz.

Egyszerűbb, és egyszersmind pontosabb megoldáshoz jutunk, ha az orientációt nem egy 3×3 elemű mátrixszal, hanem egy 4 elemű kvaternióval írjuk le. Ebben az esetben a mozgásállapotot a $[\vec{c}, \mathbf{q}, \vec{I}, \vec{J}]$ négyes jellemzi, ahol $\mathbf{q} = [\cos\frac{\alpha}{2}, \sin\frac{\alpha}{2} \cdot \vec{d}]$ a $\vec{d}$ egységvektor körüli α szögű forgatást leíró egység hosszú kvaternió. A kvaternió deriváltja [16]:

$$\frac{d\mathbf{q}}{dt} = \frac{1}{2}[0, \vec{\omega}] \cdot \mathbf{q},$$

vagyis a szögsebességből képzett $[0, \vec{\omega}]$ kvaternió és az eredeti kvaternió szorzatának a fele. Természetesen ekkor is előfordulhat, hogy a számítási pontatlanság miatt a kvaternió új értéke nem lesz egységnyi hosszú, ezért a normalizálást itt is el kell végezni.

A dinamikai szimuláció kvaterniókat alkalmazó, javított változata tehát:

Kezdeti $[\vec{c}, \mathbf{q}, \vec{I}, \vec{J}]$ állapot beállítása;
`for` $(t = t_{start};\ t \leq t_{end};\ t \mathrel{+}= \Delta t)$ {
    A $\mathbf{q}$ kvaternióból az $\mathbf{A}$ orientációs mátrix előállítása;
    $[\vec{c}, \mathbf{A}]$ alapján rajzolás;
    Ütközésdetektálás, $\vec{F} \cdot \Delta t$ és $\vec{M} \cdot \Delta t$ számítása;
    $\vec{\omega} = \Theta^{-1}(t) \cdot \vec{J}$;
    $\vec{c} \mathrel{+}= \vec{I}(t)/m \cdot \Delta t$;
    $\mathbf{q} \mathrel{+}= \frac{\Delta t}{2} \cdot [0, \vec{\omega}] \cdot \mathbf{q}$;
    $\mathbf{q} \mathrel{/}= |\mathbf{q}|$;   *// normalizálás*
    $\vec{I} \mathrel{+}= \vec{F} \cdot \Delta t$;
    $\vec{J} \mathrel{+}= \vec{M} \cdot \Delta t$;
}

Az előre lépegető Euler-módszer, bár a legegyszerűbb, egyben a legpontatlanabb is. A hibák az iteráció során összegződnek, és elviselhetetlen mértékűvé akkumulálódhatnak (a szimuláció instabillá válhat), ezért a gyakorlatban ritkán alkalmazzák ezt az eljárást.

Most egy jól bevált eljárást mutatunk be, a negyedrendű *Runge – Kutta módszer*t. Térjünk vissza a differenciálegyenlet $dS(t)/dt = \mathcal{F}(t, S(t))$ formájához! Ez az eljárás a következő képletsorral állítja elő $S(t)$-ből az $S(t+\Delta t)$-t:

$$\begin{aligned}
k_1 &= \Delta t \cdot \mathcal{F}(t, S(t)), \\
k_2 &= \Delta t \cdot \mathcal{F}(t + \Delta t/2, S(t) + k_1/2), \\
k_3 &= \Delta t \cdot \mathcal{F}(t + \Delta t/2, S(t) + k_2/2), \\
k_4 &= \Delta t \cdot \mathcal{F}(t + \Delta t, S(t) + k_3), \\
S(t + \Delta t) &\approx S(t) + k_1/6 + k_2/3 + k_3/3 + k_4/6. \qquad (9.25)
\end{aligned}$$

A *negyedrendű* jelző a becslés pontosságára utal, ugyanis a hiba $o((\Delta t)^5)$ nagyságrendű.

9.11. A hierarchikus mozgás

A valódi objektumok gyakran összetettek, azaz több kapcsolt részből, úgynevezett *szegmensből* állnak. Egy naprendszer például, mint egész, a galaxisban mozog, de belsejében a bolygók a Nap körül, a holdak pedig a bolygók körül keringenek. A *hierarchikus rendszer* mozgását több szinten érdemes megfogni: hogyan mozog a naprendszer, azon belül a Naphoz képest hogyan mozognak a bolygók, a bolygókhoz képest hogyan mozognak a holdak.

A mozgást végső soron az időfüggő modellezési transzformációk írják le. Tekintsünk két objektumot, A-t és B-t! Az A például legyen a Föld, a B pedig a Hold! Ha a B objektum mozgását az A objektumhoz képest kívánjuk megadni, akkor a B objektumot először a két test közötti $\mathbf{T}_{BA}$ relatív modellezési transzformációval az A objektum modellezési-koordinátarendszerébe helyezzük. Mivel az A objektum a saját modellezési-koordinátarendszerében rögzített, a $\mathbf{T}_{BA}$ transzformáció a B-nek az A-hoz viszonyított mozgását adja meg.

A B objektum egy $\vec{r}_B$ pontja az A szegmens koordinátarendszerében az alábbi pontba kerül:

$$[\vec{r}_{BA}, 1] = [\vec{r}_B, 1] \cdot \mathbf{T}_{BA} \qquad \Longrightarrow \qquad \vec{r}_{BA} = \vec{r}_B \cdot \mathbf{A}_{BA} + \vec{p}_{BA},$$

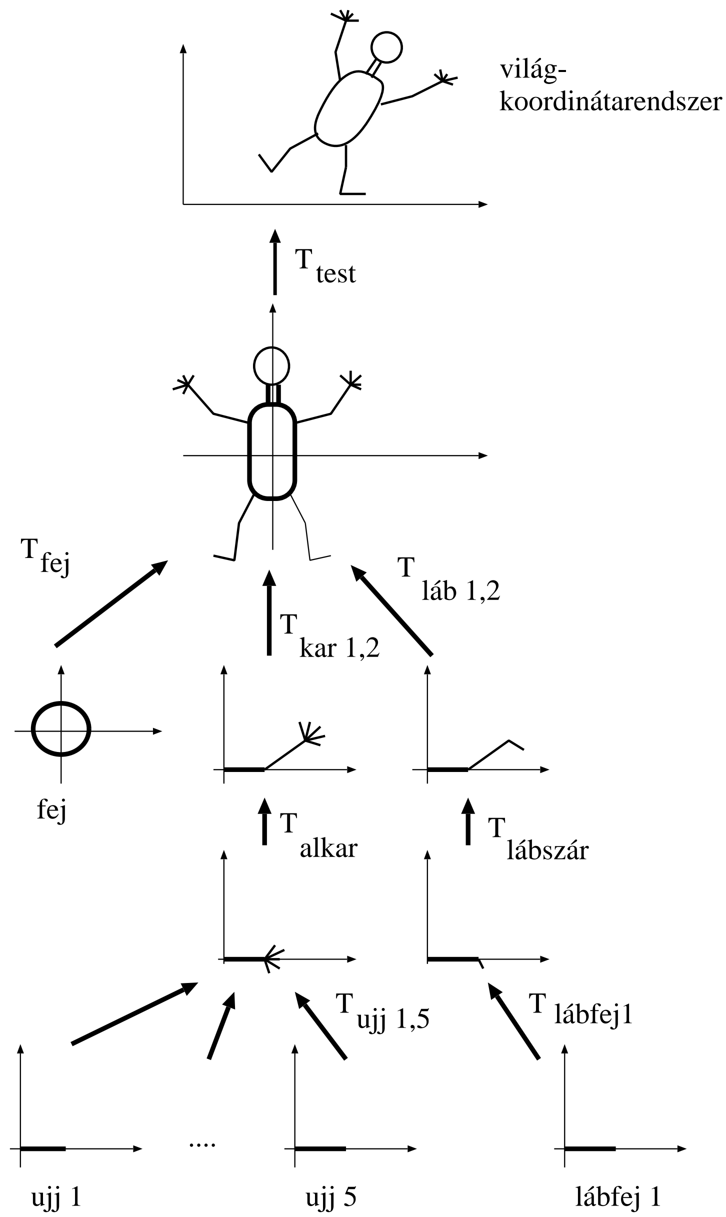
ahol az $\mathbf{A}_{BA}$ a transzformáció orientációs mátrixa, és $\vec{p}_{BA}$ pedig az eltolás vektora. Az animáció során a $\mathbf{T}_{BA}$ relatív transzformációs mátrix időben változó. Ha csak az $\mathbf{A}_{BA}$ orientációs mátrix változik az időben, az A és B szegmens távolsága állandó, azaz a két testet egy forgatást engedélyező ízülettel kapcsoltuk össze, mégpedig úgy, hogy a B szegmens origóját az A szegmens lokális modellezési-koordinátarendszerében az $\vec{p}_{BA}$ ponthoz csatoltuk. Hasonlóan, ha az orientációs mátrix állandó az időben, akkor egy transzlációs ízületet hozhatunk létre, amely az A és B objektum egymáshoz képesti elmozdulását megengedi, de a orientációjukat állandó értéken tartja.

A $\mathbf{T}_{BA}$ transzformáció a B objektumot az A modellezési-koordinátarendszerébe helyezi át. Amikor az A objektum világ-koordinátarendszerbeli helyére vagyunk kíváncsiak, egy újabb transzformációt, az A szegmens $\mathbf{T}_A$ modellezési transzformációját alkalmazzuk:

$$[\vec{r}_{w,A}, 1] = [\vec{r}_A, 1] \cdot \mathbf{T}_A, \quad [\vec{r}_{w,B}, 1] = [\vec{r}_B, 1] \cdot \mathbf{T}_{BA} \cdot \mathbf{T}_A. \qquad (9.26)$$

Amennyiben az A objektumot mozgatjuk, a B a $\mathbf{T}_{BA}$ által meghatározott relatív eltolással és orientációval követi az A mozgását, hiszen az A modellezési transzformációját a B szegmensre is alkalmazzuk.

Az A objektumot általában *szülő*nek (*parent*), a B objektumot pedig *gyermek*nek (*child*) nevezzük. Egy gyermek szintén lehet újabb gyermekek szülője. Egy emberi testben például a felkar a törzs gyermeke, és egyúttal az alkar szülője (9.33. ábra). Az alkarnak szintén van egy gyermeke, a kéz, ami pedig a szülője az öt ujjnak. A szülő–gyermek

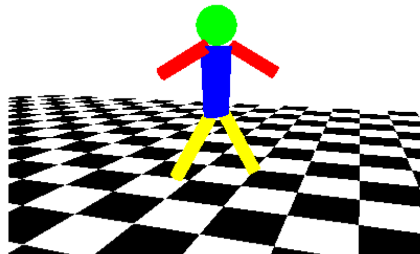


9.33. ábra. *Az emberi test hierarchikus felépítése*

kapcsolatok a szegmensek hierachikus rendszerét alakítják ki. Ez a hierarchikus rendszer határozza meg az összetett rendszer által végrehajtható mozgásfajtákat. Az összetett objektum mozgását az egyes szülő–gyermek párok relatív mozgásaival — azaz a közöttük lévő ízületek mozgásaival — adhatjuk meg. Ezen transzformációk leírásához a szokásos eljárások használhatók. Egy animációs kulcspont kialakításakor először a hierarchia tetején álló testet állítjuk be, amely maga után húzza a többi részt is. Ezután elrendezzük a felkarokat (amelyek áthelyezik az alkarokat, kezet stb.) a lábakat, majd hasonlóan folytatjuk az eljárást a hierarchia alsóbb szintjein is. Figyeljük meg, hogy a hierarchiában felülről lefelé haladunk! Ezért kapta ez az eljárás az *előremenő kinematika* (*forward kinematics*) nevet. Az interpoláció alatt a transzformációkat függetlenül interpoláljuk, figyelembe véve az egyes ízületek által megengedhető paraméterváltozásokat (a csak forgást engedő ízületnél az eltolási részhez nem nyúlunk). Egy szegmens teljes modellezési transzformációját az összes ős transzformációinak szorzataként kapjuk meg. Így az egyes ízületek kényszereit minden pillanatban ki tudjuk elégíteni.

9.11.1. Program: a primitív ember

A hierarchikus mozgást egy primitív ember animációjával szemléltetjük (9.34. ábra). A primitív emberünk mindössze egy testből, egy fejből, két kézből és két lábból áll, a könyök, térd, csukló stb. ízületeket merevnek tekintjük. A test részeit a nyak, a két váll és a két csípőízület fogja össze. Mindegyik ízület csak forgó mozgást engedélyez. A primitív emberünk geometriája is egyszerű, a feje ellipszoid, a teste csonkakúp, a végtagjai pedig henger alakúak.



9.34. ábra. *A primitív ember*

Ha az emberünk a modellezési-koordinátarendszerben az $y = 0$ síkon áll, és a pozitív x tengely irányába néz, akkor a lábak és a vállak a z tengely körül, a fej az y tengely körül fordulhat el. Az ember *mozgásparaméter*ei, azaz az animációnál módosítható változók, az x tengely mentén megtett út (`distance`), a test középpontjának a talaj feletti magassága (`up`), a vállízületek forgásszögei (`leftarm_angle` és `rightarm_angle`) és a csípőízületek forgásszögei (`leftleg_angle`, illetve `rightleg_angle`).

```
//=================================================================
class PrimitiveMan {
//=================================================================
   float forward, up;       // előre haladás és középpont magassága
   float leftarm_angle, rightarm_angle, leftleg_angle, rightleg_angle;
   float dleftarm_angle, drightarm_angle, dleftleg_angle, drightleg_angle;
   float leg;               // a láb hossza
   GLUquadricObj * quad;    // másodrendű felület azonosító
public:
   PrimitiveMan() {
      leftarm_angle = 30;  rightarm_angle = 150;
      leftleg_angle = 60;  rightleg_angle = 120;
      dleftarm_angle = 60; drightarm_angle = -60;
      dleftleg_angle = 30; drightleg_angle = -30;
      quad = gluNewQuadric();  // másodrendű felület azonosító
      forward = 0; up = leg = 5;
   }
   float Forward() { return forward; }

   void Animate(float dt) {    // a paraméterek időfüggésének számítása
      float oldleg_angle = rightleg_angle;  // kezek és lábak himbálása
      leftarm_angle  += dleftarm_angle * dt;
      rightarm_angle += drightarm_angle * dt;
      leftleg_angle  += dleftleg_angle * dt;
      rightleg_angle += drightleg_angle * dt;

      if (leftarm_angle > 150) { dleftarm_angle = -60; drightarm_angle = 60; }
      if (rightarm_angle > 150){ dleftarm_angle = 60; drightarm_angle = -60; }
      if (leftleg_angle > 120) { dleftleg_angle = -30; drightleg_angle = 30; }
      if (rightleg_angle > 120){ dleftleg_angle = 30; drightleg_angle = -30; }
      forward += 0.5 * dt;
   }
   void DrawHead(float xPos, float yPos, float zPos) { // fej rajzolás
      glPushMatrix();                 // transzformáció mentése
      glColor3f(0.0, 1.0, 0.0);       // zöld színű
      glTranslatef(xPos, yPos, zPos); // fej a nyakra
      gluSphere(quad, 1.5, 16, 32);   // 1.5 sugarú 16x32-re tesszellált gömb
      glPopMatrix();                  // transzformáció visszaállítása
   }
   void DrawTorso(float xPos, float yPos, float zPos) { // test rajzolás
      glPushMatrix();                 // transzformáció mentése
      glColor3f(0.0, 0.0, 1.0);       // kék
      glTranslatef(xPos, yPos, zPos); // test az ember közepére
      glScalef(2.0, 1.0, 1.0);        // testméret
      glRotatef(90.0, 1.0, 0.0, 0.0); // fekvő helyett álló henger
      gluCylinder(quad, 1.0, 0.8, 5.0, 32, 4); // csonkakúp
      glPopMatrix();                  // transzformáció visszaállítása
   }
```

```
   void DrawArm(float xPos, float yPos, float zPos, float angle) {
      glPushMatrix();                 // transzformáció mentése
      glColor3f(1.0, 0.0, 0.0);       // vörös
      glTranslatef(xPos, yPos, zPos); // kéz a vállhoz
      glRotatef(angle, 1.0, 0.0, 0.0);// lóbálás
      gluCylinder(quad, 0.5, 0.5, 4.0, 32, 4); // állandó sugarú henger
      glPopMatrix();                  // transzformáció visszaállítása
   }

   void DrawLeg(float xPos, float yPos, float zPos, float angle) {
      glPushMatrix();                 // transzformáció mentése
      glColor3f(1.0, 1.0, 0.0);       // sárga
      glTranslatef(xPos, yPos, zPos); // láb a csípőhöz
      glRotatef(angle, 1.0, 0.0, 0.0);// lóbálás
      gluCylinder(quad, 0.5, 0.5, leg, 32, 4); // állandó sugarú henger
      glPopMatrix();                  // transzformáció visszaállítása
   }

   void Draw() {     // az ember felrajzolása a paraméterek alapján
      glTranslatef(0, up, forward); // az egész ember eltolása
      DrawHead(0.0, 11.5, 0.0);                // fejrajzolás
      DrawTorso(0.0, 10.0, 0.0);               // testrajzolás
      DrawArm(-2.0, 10.0, 0.0, leftarm_angle); // bal kar
      DrawArm(2.0, 10.0, 0.5, rightarm_angle); // jobb kar
      DrawLeg(-1.5, 5.0, 0.0, leftleg_angle);  // bal láb
      DrawLeg(1.5, 5.0, 0.0, rightleg_angle);  // jobb láb
   }
};
```

A `Draw()` függvény az ember rajzolását a testrészek rajzolására vezeti vissza úgy, hogy elhelyezi a testrészeket. Az emberhez képest a fej (*head*) és a test (*torso*) helye és orientációja állandó. A kezek (*arm*) és a lábak (*leg*) elhelyezésében azonban részt vesznek az időben változó szögek is. Az egyes testrészeket rajzoló eljárások elmentik az aktuális transzformációs mátrixot, a transzformációs mátrixhoz hozzáfűzik a szülőhöz viszonyított saját transzformációjukat, elvégzik a tényleges rajzolást, végül visszaállítják a transzformációs mátrix elmentett változatát. A transzformációs mátrix mentésére és visszaállítására azért van szükség, mert egy szegmensre a saját és a szülők transzformációi hathatnak, de az ugyanolyan szinten álló, korábban feldolgozott szegmensek transzformációi semmiképpen sem. Nyilván a kezeknek és a lábaknak az emberrel együtt kell haladniuk, de azt nem szeretnénk, ha az egyik kéz forgatása, a másik kezet, vagy valamelyik lábat is elfordítaná. Ez minden hierarchikus rendszernél így van, ezért jó megjegyezni, hogy a szegmensek rajzolását `glPushMatrix()` és `glPopMatrix()` függvényhívásokkal kell körülvenni.

Az időfüggő paraméterek, azaz az előrehaladásért felelős `forward` és a lábak és a kezek himbálását végző `leftleg_angle`, `rightleg_angle`, `leftarm_angle`, `rightarm_angle` az `Animate()` függvényben kapnak értéket. A lábak és a kezek állandó szögsebességgel lengedeznek, amikor pedig elérték a maximális kitérést, akkor a szögsebesség irányt vált.

A test pozícióját meghatározó `forward` és `up` változók kiszámítása külön figyelmet érdemel. Ha a fenti megoldásban az előrehaladás `forward` értékét egyenletes sebességgel növelnénk, az `up` magasságot pedig állandó értéken tartanánk, akkor a primitív emberünk úgy mozogna, mintha kötélen húznák, mialatt kezeit és a lábait lóbálja. Azt az egyetlen esetet kivéve, amikor a lábak zártak, az ember talpa nem érintkezne a talajjal. Mi hús-vér emberek nem így járunk, hanem úgy, hogy az egyik lábunk mindig a földhöz ér, és a földhöz képest nem mozdul el. A valószerűbb szimulációhoz tehát a láb végének a pályáját írjuk elő és ebből számítjuk vissza a hierarchia felsőbb szintjeihez tartozó mozgásparamétereket. Mivel az ilyen irányú okoskodás éppen ellentétes az előremenő kinematika által követett iránnyal, ezt a megközelítést *inverz kinematikának* (*inverse kinematics*) nevezzük. A függőleges irány, a láb pillanatnyi helyzete és a talaj egy derékszögű háromszöget képez, amelynek magassága éppen a test magassága (`up`), a másik befogója a test középpontjának és a lábnak a vízszintes távolsága, az átfogója pedig a láb hossza. A háromszög szöge a láb kitérési szögéből adódik, így ebből és a láb hosszából a test magassága kiszámítható. Az előrehaladást kifejező `forward` változó értékét pedig úgy kapjuk meg, hogy megnézzük, hogy az utolsó kerethez képest a földhöz tapadó láb és a test vízszintes távolsága mennyit változott. Mivel a láb áll, a testet pontosan ezzel a változással kell előremozgatni.

Az egyszerű inverz kinematikát alkalmazó animációs függvény:

```
//-----------------------------------------------------------------
void PrimitiveMan::Animate(float dt) {
//-----------------------------------------------------------------
   float oldleg_angle = rightleg_angle;     // kezek és lábak himbálása
   leftarm_angle  += dleftarm_angle * dt;
   rightarm_angle += drightarm_angle * dt;
   leftleg_angle  += dleftleg_angle * dt;
   rightleg_angle += drightleg_angle * dt;

                                            // irányváltás
   if (leftarm_angle > 150) { dleftarm_angle = -60; drightarm_angle = 60; }
   if (rightarm_angle > 150){ dleftarm_angle = 60; drightarm_angle = -60; }
   if (leftleg_angle > 120) { dleftleg_angle = -30; drightleg_angle = 30; }
   if (rightleg_angle > 120){ dleftleg_angle = 30; drightleg_angle = -30; }

                                            // "inverz kinematika"
   forward += leg * fabs(sin((rightleg_angle - 90) * M_PI/180) -
                         sin((oldleg_angle - 90) * M_PI/180));
   up = leg * (cos((rightleg_angle - 90) * M_PI/180) - 1);
}
```

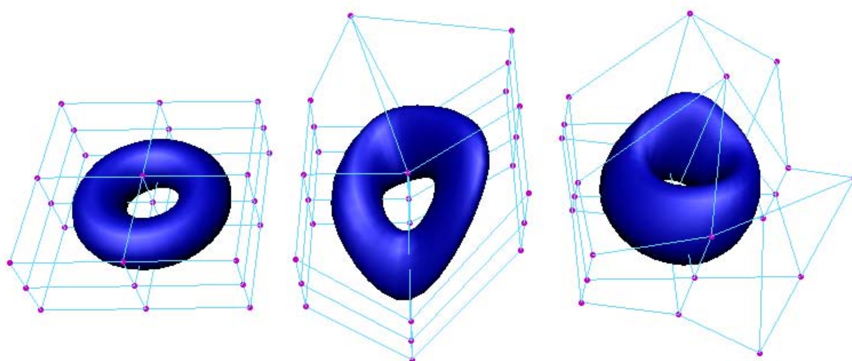
Az ember alá még egy sakktáblát is rajzolunk, a kameratranszformációt pedig úgy alakítjuk ki, hogy a kamera adott távolsággal kövesse az embert, miközben forog körülötte:

```
//===============================================================
class PManWindow : public Application {
//===============================================================
    PrimitiveMan  pman;       // a primitív ember
    float         cam_angle;  // kamera ebből a szögből néz
public:
    PManWindow() : Application("Primitive Man", 400, 400)  { }
    void Init() {
        glEnable(GL_DEPTH_TEST);    // z-bufferes takarás engedélyezése
        glViewport(0, 0, windowWidth, windowHeight); // képernyő transzformáció
        glMatrixMode(GL_PROJECTION);       // projekciós mátrix
        glLoadIdentity();                  // alaphelyzet
        gluPerspective(54,                 // látószög,
                       (float)windowWidth/windowHeight, // oldalarány
                       0.1,                // első vágósík
                       100.0);             // hátsó vágósík
        cam_angle = 0;
    }
    void DrawPlane() {         // alaplap felrajzolása, az ember ezen sétál
        glBegin(GL_QUADS);     // az alaplap 40x40-es sakktáblamező (négyszög)
        for(int x = -20; x < 20; x++) {
            for(int z = -20; z < 20; z++) {
                float col = (x ^ z) & 1;  // sakktábla színezés
                glColor3f(col, col, col);
                glVertex3f(x * 5,     0.0, z * 5);
                glVertex3f((x+1) * 5, 0.0, z * 5);
                glVertex3f((x+1) * 5, 0.0, (z+1) * 5);
                glVertex3f(x * 5,     0.0, (z+1) * 5);
            }
        }
        glEnd();
    }
    void Render() {
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();                // kamerabeállítás
        glTranslatef(0, -10, -30);       // távolság
        glRotatef(cam_angle, 0, 1, 0);   // kameraforgatás az y tengely körül
        glTranslatef(0, 0, -pman.Forward());   // a kamera követi az embert
        glClearColor(0, 0, 0, 0);        // törlési szín fekete
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // képernyő törlés
        DrawPlane();                     // a sík rajzolás, amin az ember mozog
        pman.Draw();                     // ember rajzolás
        SwapBuffers();                   // buffercsere
    }
    void Do_a_Step(float dt) {
        cam_angle += 10 * dt;            // a kamera folyamatosan forog
        pman.Animate(dt);                // embert mozgatjuk
        Render();                        // rajzolás
    }
};

void Application::CreateApplication() { new PManWindow(); }
```

9.12. Deformációk

Az animációnál nem csak a testek helyzete, de alakjuk is változhat (gondoljunk két összeütköző autóra, vagy akár saját testünkre, amely a mozgás során folyamatosan gyűri a bőrünket). Az alakváltozások kis részét (*összenyomás–széthúzás*, *nyírás*) leírhatjuk lineáris transzformációkkal, így ezek a forgatáshoz és az eltoláshoz hasonlóan kezelhetők. A nemlineáris transzformációk (3.2.11. fejezet) bonyolultabb alakváltozásokat (*hegyesítés*, *csavarás*, *hajlítás*) is előidézhetnek. Ezen transzformációk paramétereit az időben változtatva pedig az alakváltozás animálható is. További lehetőség a poligonháló csúcspontjainak és a paraméteres felületek csomópontjainak egyenkénti mozgatása, amely a felület alakját módosítja.

Amennyiben ezen módszerek egyike sem megfelelő számunkra, mert egyikkel sem tudjuk az elképzelt formaváltozást pontosan megvalósítani, akkor egy erre a célra kidolgozott eljáráshoz kell folyamodnunk, amelynek neve *szabadformájú deformáció* (*free-form deformation*, *FFD*). Az elnevezés a szabadformájú felületek tervezésénél megismert módszerekre utal. Ugyanis itt is vezérlő pontokkal dolgozunk, majd egy interpolációs eljárásra bízzuk azt, hogy a vezérlőpontok alapján a többi helyen elvégezze a szükséges módosításokat. A deformációs eszközünk egy háromdimenziós rács (*lattice*), amelynek csomópontjai kapják meg a deformációs vezérlőpontok szerepét (9.35. ábra).



9.35. ábra. *A deformációs rács és a módosításának hatása a felületre*

A görbéknél és felületeknél megszokott módon, a vezérlőpontokat paraméteres függvényekkel szorozva a háromdimenziós tér pontjait is leírhatjuk:

$$\vec{r}(u,v,w) = \sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} B_i(u)\cdot B_j(v)\cdot B_k(w)\cdot r_{i,j,k}. \qquad (9.27)$$

A görbéknél a vezérlőpontok egy sorozatot alkottak, amelyeket egyváltozós súlyfüggvényekkel szoroztunk. A felületeknél a vezérlőpontokat egy kétdimenziós rácsba szer-

veztük és két, különböző változóval paraméterezett súlyfüggvények szorzataival súlyoztuk. Most pedig a vezérlőpontok egy háromdimenziós rácsot alkotnak, amelyeket három, különböző változóval paraméterezett súlyfüggvények szorzataival súlyozunk. Ez érthető is, ugyanis most a térnek nem csupán egy egydimenziós vagy kétdimenziós részhalmazát szeretnénk előállítani, hanem a teljes teret. A súlyfüggvények mindhárom esetben ugyanazok: most is alkalmazhatjuk a lineáris és a sátorszerű súlyfüggvényeket, vagy akár a Bézier-módszer Bernstein-polinomjait is.

A deformációhoz tekintsünk egy tetszőleges $\vec{p}$ pontot, és fejezzük ki azt a deformációs rács paramétereivel, azaz oldjuk meg a $\vec{p} = \vec{r}(u,v,w)$ egyenletet, és számítsuk ki az u_p, v_p, w_p koordinátákat! A deformációt a deformációs vezérlőpontok elmozdításával irányítjuk. A $\vec{p}$ pontunk új helyét a 9.27. egyenlet alkalmazásával kaphatjuk meg, ha abba behelyettesítjük a régi u_p, v_p, w_p koordinátákat és a megváltozott vezérlőpontokat. Ez az eljárás működik poligonmodellekre, ha a csúcspontokat transzformáljuk, de alkalmazhatjuk parametrizált felületekre is. A művelet linearitása miatt, ha a felület vezérlőpontjait deformáljuk, akkor ugyanahhoz az eredményhez jutunk, mintha a deformátorral magát a felületet torzítanánk.

9.13. Karakteranimáció

A hierarchikus rendszerek egy fontos osztályát a *karakter*ek (emberek, állatok, egyéb lények) alkotják. A karakterek voltaképpen a merev részekből álló csontozatukat mozgatják, amelyhez tapadó hús, bőr a csontok pillanatnyi helyzetének megfelelően deformálódik. A karakterekben az egyes hierarchikus szintek szülő–gyermek kapcsolataiban a relatív transzformáció korlátozott. Például az emberek és állatok ízületei a gyermekszegmenseknek csak az orientációját engedik megváltoztatni, azt is csak korlátozott mértékben. A váll- és csuklóízületben az orientációt szabadabban változtathatjuk, mint a könyök- és a térdízületben, az utóbbiak ugyanis csak rögzített tengely körüli elforgatást tesznek lehetővé. Minden emberi ízületre igaz, hogy csak a relatív orientáció változhat, a pozíció nem, hiszen az ízületek összetartják és nem engedik eltávolodni az itt csatlakozó csontokat (*rotációs csukló*). Robotok, szállítószalagok stb. tartalmazhatnak *transzlációs csukló*kat is, amelyek a relatív elmozdulást módosítják.

A függetlenül állítható paraméterek összességét *állapot*nak (*state*) nevezzük és **S**-sel jelöljük. Az állapotnak, valamint az animált hierarchikus rendszer felépítésének és geometriájának ismeretében a rendszer minden pontjának helye meghatározható.

A mozgatásnak rendszerint valamilyen célja van (például, kinyújtom a kezem, hogy elérjem a sörösüveget), azaz a rendszer valamely pontját (kéz) szeretnénk valamilyen előírt pontig (sörösüveg) és orientációig (a kézbe jól belesimuljon a sörösüveg nyaka) vezérelni. A rendszer kitüntetett pontját, amelynek pozíciójáról és orientációjáról rendelkezni szeretnénk (a sörösüveg példában a kéz), *végberendezés*nek (*end effector*)

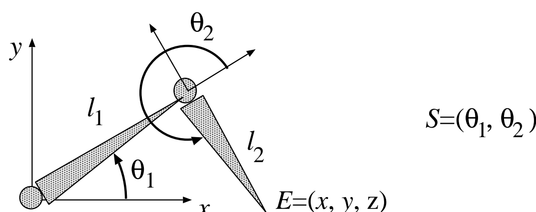
nevezzük. A végberendezés helyzetét **E**-vel jelöljük, amely a pillanatnyi pozícióval és orientációval adható meg:

$$\mathbf{E} = [x, y, z, \alpha, \beta, \gamma].$$

Gyakran csak a végberendezés pozíciójáról rendelkezünk, az orientáció nem érdekes. Ebben az esetben a végberendezés helyzete három skalárral leírható: $\mathbf{E} = [x, y, z]$. Az állapot a rendszer minden pontjának helyzetét meghatározza, így a végberendezését is. Léteznie kell tehát az **E** végberendezés helyzetét az **S** állapotból kifejező

$$\mathbf{E} = \mathbf{F}(\mathbf{S})$$

*strukturális függvény*nek, amely csak a rendszer felépítésétől és a geometriájától függ.



9.36. ábra. *Kétszabadságfokú, rotációs csuklókat tartalmazó rendszer*

Vegyünk egy egyszerű példát, a két rotációs csuklót tartalmazó rendszert (9.36. ábra)! A csuklók a z tengely körüli elfordulást engedik meg. A rendszer állapota tehát $\mathbf{S} = [\theta_1, \theta_2]$, ahol θ_1 az első csuklóbeli, a θ_2 pedig a másik csuklóbeli elfordulási szög. Tekintsük a rendszer végberendezésének a második csukló végpontját és tételezzük fel, hogy az első csukló kezdőpontja az origóban rögzített! A 9.36. ábra és geometriai megfontolások alapján a végberendezés pozíciója:

$$\begin{aligned} x(\theta_1, \theta_2) &= l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2), \\ y(\theta_1, \theta_2) &= l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2), \\ z(\theta_1, \theta_2) &= 0, \end{aligned}$$

ahol l_1 az első, l_2 pedig a második csukló hossza. Figyeljük meg, hogy a θ_1, θ_2 állapotváltozókból az x, y, z végberendezéshelyzetet kifejező strukturális egyenlet nemlineáris!

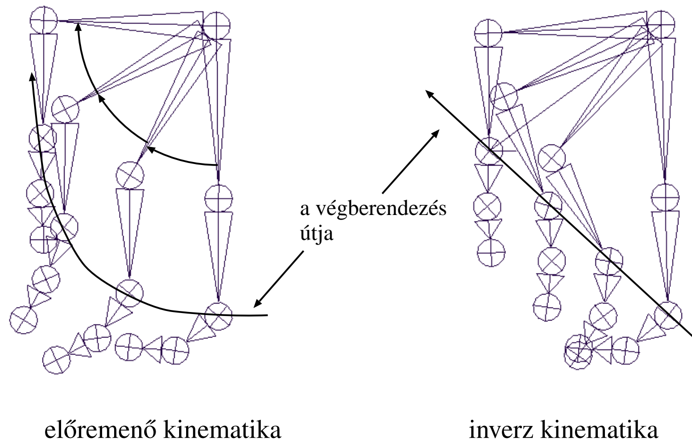
A teljes mozgás létrehozásához az egyes szintek transzformációs mátrixait kell minden időpillanatra kiszámítani, azaz például a kulcskeretekből interpolálni. Elegendő csak az állapotváltozók interpolációjáról szót ejteni, ugyanis a transzformációs mátrixok közvetlen kapcsolatban állnak az egyes állapotváltozókkal: egy rotációs csukló a pillanatnyi forgástengely körül az állapotváltozóban megadott szöggel forgat el, a transzlációs csukló pedig egy eltolást jelent.

Az interpolációt két különböző térben is elvégezhetjük, amely alapján előremenő és inverz kinematikai megközelítésről beszélhetünk.

9.13.1. Előremenő kinematika

Az előremenő kinematika közvetlenül az állapotváltozók terében dolgozik, a kulcspontokat itt veszi fel, és ezeket interpolálja:

for (minden $k = 1 \dots K$ keretre) { *// mozgástervezés*
    A karakter beállítása a k-adik keretben;
    Az állapot elmentése az $\mathbf{S}_1[k], \mathbf{S}_2[k], \dots, \mathbf{S}_n[k]$ változókban;
}
for (minden $i = 1 \dots n$ állapotváltozóra) {
    Görbeillesztés az $\mathbf{S}_i(t)$ változóra az $\mathbf{S}_i[1], \mathbf{S}_i[2], \dots, \mathbf{S}_i[K]$ kulcsértékekből;
}
for ($t = t_{\text{start}}$; $t \leq t_{\text{end}}$; t += Δt) { *// animáció*
    Az $\mathbf{S}_1(t), \mathbf{S}_2(t), \dots, \mathbf{S}_n(t)$ alapján a transzformációk beállítása;
    Rajzolás;
}



9.37. ábra. *Az előremenő és az inverz kinematika összehasonlítása*

9.13.2. Inverz kinematika

Az előremenő kinematika nem használható, ha a strukturális összefüggés erősen nemlineáris. Hiába interpolálunk szépen egyenletesen az állapottérben, a végberendezés vadul kalimpálhat a kulcspontok között. Gondoljunk arra, hogy egy tárgyat egy kicsiny lyukba illesztünk, vagy a lábunkat a talaj felett előrenyújtjuk! Hiába igazítjuk el a szereplőket a mozgás elején és végén úgy, hogy ne hatoljanak egymásba, a közbenső

interpolált kereteknél előfordulhat, hogy a tárgyunk átlép a lyuk határain, vagy a lábunk belenyomul a földbe.

Az ilyen nehéz eseteknél jelent megoldást az *inverz kinematika*, amely nem az állapotot, hanem a kritikus végberendezés helyzetét interpolálja, majd az állapotot a végberendezés interpolált helyzetéből számítja vissza. Az inverz kinematika másik neve, a *cél-orientált animáció* jól kifejezi, hogy most a végberendezésre koncentrálunk, és annak helyzetéből következtetünk az állapotváltozókra. A 9.37. ábrán az előremenő és az inverz kinematikát hasonlítottuk össze egy „láb” két kulcskeret közötti animációjával. A végberendezés a bokánál van, amit a jobb oldali ábrán a csuklólánc kezdetét és a végberendezést összekötő vonal is jelez. Figyeljük meg, hogy a boka pályája inverz kinematikánál egyenes, előremenő kinematikánál viszont nem. Az elfordulási szögek azonban az előremenő kinematikánál változnak egyenletesen, inverz kinematikánál viszont nem.

Az inverz kinematikánál tehát a végberendezés pályáját tervezzük meg. Az $\mathbf{S}$ állapot a strukturális összefüggés inverzével állítható elő a végberendezés helyzetéből:

$$\mathbf{S} = \mathbf{F}^{-1}(\mathbf{E}).$$

Az invertálás azonban több problémát is felvet. Egyrészt az $\mathbf{F}$ nemlineáris, tehát az inverz függvény kiszámítása nem triviális, másrészt nem egy-egy értelmű, azaz több állapothoz tartozhat ugyanaz a végberendezés helyzet. Ragadjuk meg a kezünkben lévő könyvet, és úgy, hogy a könyv ne mozduljon, mozgassuk könyökünket, vállunkat, vagy akár az egész testünket! Ugye megy? A végberendezés helyzet állandó, amit nagyon sok különféle ízületi beállítással elérhetünk. Az inverzió nemlinearitásával és többértelműségével egy iterációs eljárás segítségével birkózhatunk meg, amely a lehetséges megoldások közül egyet állít elő. Az iteráció alapötlete az, hogy, ha egy t pillanatban ismerjük az összetartozó végberendezés helyzet–állapot párt, akkor ebből következtethetünk a $t+\Delta t$ időpontban érvényes párra. Ráadásul, ha Δt kicsiny, akkor a nemlineáris függvényeket az érintőjükkel közelíthetjük (*linearizálás*), azaz kicsiben mégiscsak lineáris egyenletekkel dolgozhatunk.

Jelöljük a t időpillanatban érvényes állapotot $\mathbf{S}(t)$-vel, a hozzá tartozó végberendezés helyzetet pedig $\mathbf{E}(t)$-vel. Kis Δt idővel később ezek a jellemzők némileg megváltoznak:

$$\mathbf{S}(t+\Delta t) = \mathbf{S}(t) + \Delta\mathbf{S}, \quad \mathbf{E}(t+\Delta t) = \mathbf{E}(t) + \Delta\mathbf{E}.$$

A megváltozott helyzet ugyanannak a rendszernek egy másik állapota, ezért itt is érvényes a strukturális összefüggés:

$$\mathbf{E}(t+\Delta t) = \mathbf{F}(\mathbf{S}(t+\Delta t)) \implies \mathbf{E}(t) + \Delta\mathbf{E} = \mathbf{F}(\mathbf{S}(t) + \Delta\mathbf{S}).$$

Alkalmazzuk a strukturális függvény Taylor-soros közelítését (linearizálunk):

$$\mathbf{E}(t) + \Delta\mathbf{E} = \mathbf{F}(\mathbf{S}(t) + \Delta\mathbf{S}) \approx \mathbf{F}(\mathbf{S}(t)) + \frac{\partial\mathbf{F}}{\partial\mathbf{S}_1}\Delta\mathbf{S}_1 + \frac{\partial\mathbf{F}}{\partial\mathbf{S}_2}\Delta\mathbf{S}_2 + \ldots + \frac{\partial\mathbf{F}}{\partial\mathbf{S}_n}\Delta\mathbf{S}_n.$$

Mivel t-ben az $\mathbf{E}(t)$ végberendezés helyzet és az $\mathbf{S}(t)$ állapot összetartozott, fennáll a $\mathbf{E}(t) = \mathbf{F}(\mathbf{S}(t))$ strukturális összefüggés, tehát az egyenlet bal oldaláról levehetjük az $\mathbf{E}(t)$-t, mialatt a jobb oldaláról eltávolítjuk a vele megegyező $\mathbf{F}(\mathbf{S}(t))$-t:

$$\Delta\mathbf{E} \approx \frac{\partial\mathbf{F}}{\partial\mathbf{S}_1}\Delta\mathbf{S}_1 + \frac{\partial\mathbf{F}}{\partial\mathbf{S}_2}\Delta\mathbf{S}_2 + \ldots + \frac{\partial\mathbf{F}}{\partial\mathbf{S}_n}\Delta\mathbf{S}_n. \tag{9.28}$$

Ez egy vektoregyenlet, amely a végberendezés m darab koordinátájára külön-külön is érvényes. A 9.28. egyenletet mátrixos alakban a következőképpen is felírhatjuk:

$$\begin{bmatrix} \Delta\mathbf{E}_1 \\ \vdots \\ \Delta\mathbf{E}_m \end{bmatrix} = \begin{bmatrix} \frac{\partial\mathbf{F}_1}{\partial\mathbf{S}_1} & \cdots & \frac{\partial\mathbf{F}_1}{\partial\mathbf{S}_n} \\ \vdots & & \vdots \\ \frac{\partial\mathbf{F}_m}{\partial\mathbf{S}_1} & \cdots & \frac{\partial\mathbf{F}_m}{\partial\mathbf{S}_n} \end{bmatrix} \cdot \begin{bmatrix} \Delta\mathbf{S}_1 \\ \vdots \\ \Delta\mathbf{S}_n \end{bmatrix}. \tag{9.29}$$

Az egyenletrendszer mátrixát *Jacobi-mátrix*nak nevezzük. Az egyenletrendszer egyenleteinek száma a végberendezés szabadságfokaival egyezik meg, az ismeretlenek száma pedig az állapotváltozók számával. Példaképpen, a 9.36. ábrán látható, kétszabadságfokú síkbeli rendszer Jacobi-mátrixa az alábbi:

$$\mathbf{J} = \begin{bmatrix} -l_1\sin\theta_1 - l_2\sin(\theta_1+\theta_2) & -l_2\sin(\theta_1+\theta_2) \\ l_1\cos\theta_1 + l_2\cos(\theta_1+\theta_2) & l_2\cos(\theta_1+\theta_2) \\ 0 & 0 \end{bmatrix}.$$

A gyakorlatban a végberendezés szabadságfokainak száma 3, ha csak a pozíció érdekes, vagy 6, ha a pozíciót és az orientációt is előírjuk. Ezzel szemben az ismeretlen állapotváltozók száma ennél jóval nagyobb lehet, azaz az egyenletben sokkal több ismeretlent találunk, mint ahány egyenletünk van. A kétszabadságfokú rendszernek ugyan csak két állapotváltozója van, de ez egy különlegesen egyszerű példa. Az emberi test szabadságfoka viszont több száz. Így a 9.29. egyenletnek nagyon sok megoldása lehet, amint azt már korábban is megállapítottuk. A megoldások közül egyet kell kiválasztanunk, lehetőleg olyat, amely valamilyen szempontból a legjobb tulajdonságokkal rendelkezik. A feladatot általánosabban is megfogalmazhatjuk. Tekintsünk egy olyan

$$\Delta\mathbf{E}[m] = \mathbf{J}[m\times n]\cdot\Delta\mathbf{S}[n]$$

egyenletrendszert, ahol az egyenletek száma, azaz $\Delta\mathbf{E}[m]$ dimenziója m, az ismeretlenek száma, azaz a $\Delta\mathbf{S}[n]$ dimenziója n, és a $\mathbf{J}[m\times n]$ mátrix mérete $m\times n$! Az egyenletek száma kisebb az ismeretlenek számánál, azaz $m < n$. Egy megoldás előállításához szorozzuk meg az egyenlet mindkét oldalát a $\mathbf{J}$ transzponáltjával, azaz a $\mathbf{J}^T[n\times m]$ mátrixszal, amely $n\times m$ méretű:

$$\mathbf{J}^T[n\times m]\cdot\Delta\mathbf{E}[m] = \mathbf{J}^T[n\times m]\cdot\mathbf{J}[m\times n]\cdot\Delta\mathbf{S}[n].$$

Vegyük észre, hogy a $\mathbf{J}^T[n \times m] \cdot \mathbf{J}[m \times n]$ $n \times n$ méretű négyzetes mátrix, így már reményünk van arra, hogy az inverzét kiszámítsuk (ez az inverz akkor létezik, ha a mátrix *nem szinguláris*, azaz a *determinánsa* nem zérus). Az egyenletet a mátrix inverzével szorozva az ismeretlen $\Delta\mathbf{S}[n]$ vektor kifejezhető:

$$\Delta\mathbf{S}[n] = (\mathbf{J}^T[n \times m] \cdot \mathbf{J}[m \times n])^{-1} \cdot \mathbf{J}^T[n \times m] \cdot \Delta\mathbf{E}[m].$$

A megoldásban a $\Delta\mathbf{E}[m]$ vektort szorzó mátrixot a $\mathbf{J}$ *pszeudóinverz*ének nevezzük és általában a következőképpen jelöljük:

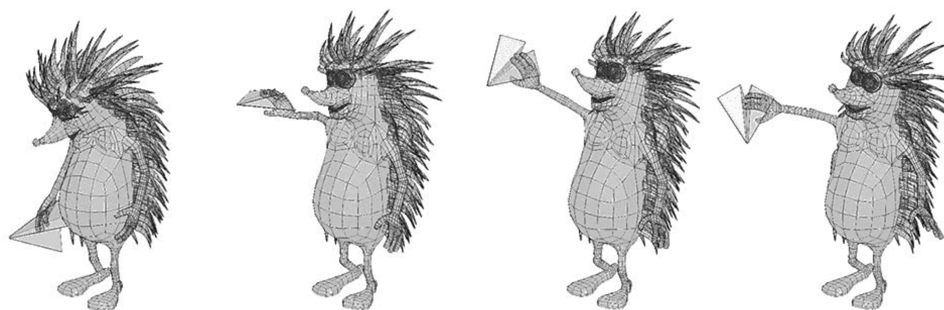
$$\mathbf{J}^+ = (\mathbf{J}^T[n \times m] \cdot \mathbf{J}[m \times n])^{-1} \cdot \mathbf{J}^T[n \times m].$$

A pszeudóinverz tehát nem négyzetes mátrixok „invertálásához”, azaz alulhatározott egyenletek egy lehetséges megoldásának előállításához használható. A pszeudóinverzzel kapott állapotváltozások minimálisak, tehát a lehetséges mozgások közül azt kapjuk meg így, amelyekben az ízületekben a csontok relatív sebessége minimális.

Az iteratív megoldásban kihasználhatjuk, hogy a t_0 időpontbeli kezdeti állapotban a végberendezés helyzete és a rendszer állapot ismert. A végberendezést az előírt pályán kis lépésekben mozgatjuk, és minden lépésben a Jacobi-mátrix pszeudóinverzének felhasználásával kiszámítjuk az ehhez szükséges állapotváltozást. Az állapotváltozók ismeretében a rendszert fel tudjuk rajzolni. Mivel a Jacobi-mátrix maga is függ az állapottól, ezért minden lépésben újra ki kell számolni. Összefoglalva, az inverz kinematika a következőképpen mozgat egy karaktert:

```
E = E(t_start); S = S(t_start);
for (t = t_start; t ≤ t_end; t += Δt) {
        S alapján a transzformációs mátrixok előállítása;
        Képszintézis;
        J(S) Jacobi-mátrix számítása;
        J+ = J pszeudo-inverze;
        E(t + Δt) számítása kulcskeret interpolációval;
        ΔE = E(t + Δt) − E(t);
        ΔS = J+ · ΔE;
        S += ΔS;
}
```

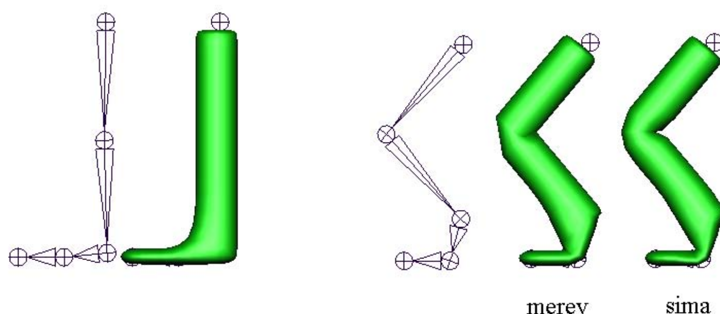
Az animáció kezdetén, az első kulcskeretben az összetartozó pár nyilván ismert, hiszen a kulcskeretben egyaránt leolvashatjuk az állapotváltozókat és a végberendezés helyzetét. Ebből indulunk és Δt lépésekkel haladva az időben a teljes animációt kiszámítjuk.



9.38. ábra. *Karakteranimáció inverz kinematikával és bőrözéssel (Tüske Imre)*

9.13.3. Bőrözés

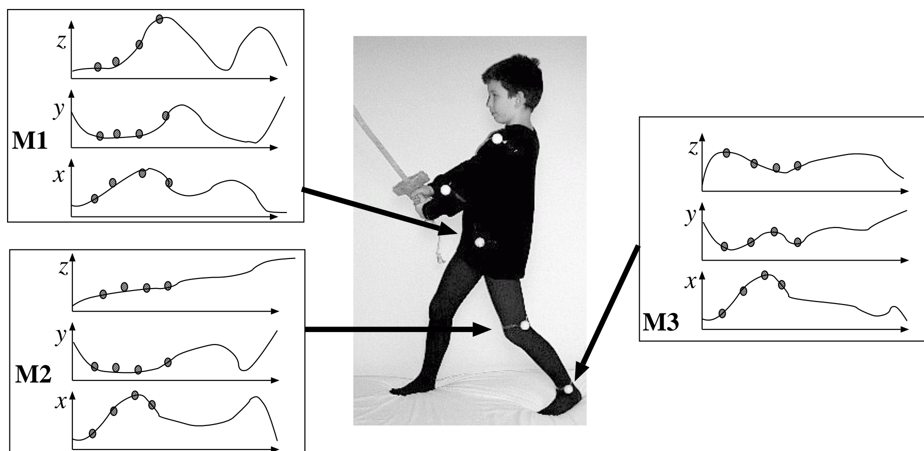
A *bőr* egy felület, amely a test felszínét mutatja, és amelyet a képszintézis megjelenít. A bőrt a csontvázzal kapcsoljuk össze, azaz a csontváz mozgatása a bőrt is deformálja, mégpedig úgy, hogy egy bőrfelület közelében lévő ízületek mozgása határozza meg a bőrfelület alakjának változásait. Ezt a felület vezérlőpontjainak az ízületek helyével történő összekapcsolásával valósíthatjuk meg. Az egyes ízületek elmozdulása az ízület és a vezérlőpont távolságával csökkenő mértékben a vezérlőpontot is elmozdítja, a vezérlőpont mozgása pedig deformálja a felületet. Amennyiben egy vezérlőpontra csak egyetlen, célszerűen a legközelebbi ízület hat, *merev bőrözés*ről (*rigid binding*) beszélünk. Ha egy vezérlőpont elmozdulása több ízület elmozdulásának súlyozott átlaga, akkor *sima bőrözés*t (*smooth binding*) kapunk. Ebben az esetben az ízületek súlya az animáció beállítható paramétere.



9.39. ábra. *Bőrözés*

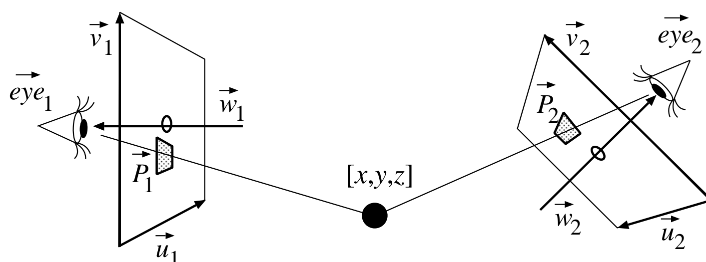
9.14. Mozgáskövető animáció

A *mozgáskövető animáció* mozgástervezés helyett méréssel határozza meg a mozgásgörbéket. A mérési eljárásban élő „karaktereket” mozgatunk (9.40. ábra), amelyeken jól azonosítható pontokat jelölünk ki. Az azonosítás érdekében a karakter testére, a környezettől jelentősen eltérő színű „markereket” szerelünk fel. A mérés e kitüntetett pontok pályáját határozza meg.



9.40. ábra. *A mozgásgörbék létrehozása a markerek pályagörbéiből*

A mérés eszköze a kamera, amely kétdimenziós képeket készít. Háromdimenziós térbeli adatokat úgy állíthatunk elő kétdimenziós képekből, hogy több kameraállásból készítünk képet, és a képeken megkeressük, hogy hol látjuk ugyanazt a háromdimenziós pontot. Az egyszerűség kedvéért tételezzük fel, hogy két, különböző kameraállásból felvett képünk van! Két kép ugyanis már elegendő a háromdimenziós rekonstrukcióhoz, a többi kép csak a mérési pontatlanságokat csökkentené. A két képet használó eljárást *sztereolátás*nak nevezzük.



9.41. ábra. *Sztereolátás*

Tegyük fel, hogy a két kamerának megfelelő szempozíció $\vec{eye}_1$, illetve $\vec{eye}_2$, a kamerák vízszintes irányait kijelölő egységvektorok $\vec{u}_1$, illetve $\vec{u}_2$, a függőleges irányokat kijelölő egységvektorok $\vec{v}_1$, illetve $\vec{v}_2$, a kamera nézeti irányai $\vec{w}_1$, illetve $\vec{w}_2$, a fókusztávolságok, azaz a szemek és az ablakok távolságai f_1, illetve f_2! Az ablak, azaz a film mérete pedig mindkét esetben legyen $s \times h$, a képek felbontása pedig $X_{\max} \times Y_{\max}$!

Az ismertetett kameraparaméterek alapján, a képen egy X, Y pixel a háromdimenziós tér következő $\vec{P}$ pontjában van (9.41. ábra):

$$\vec{P}(X,Y) = \vec{eye} - f \cdot \vec{w} + \frac{s \cdot (X - X_{\max}/2)}{X_{\max}} \cdot \vec{u} + \frac{h \cdot (Y - Y_{\max}/2)}{Y_{\max}} \cdot \vec{v}.$$

Vegyük észre, hogy egy adott pixelbe vetülő pontok nem változnak, ha a kép méretét és a fókusztávolságot arányosan változtatjuk meg! Ezért érdemes a fókusztávolságot úgy rögzíteni, hogy az ablakon a pixel mérete éppen egységnyi legyen. Ekkor fennállnak az $s = X_{\max}$ és a $h = Y_{\max}$ összefüggések, így a pixel koordinátákat és a világ koordinátákat összerendelő összefüggés az alábbi egyszerűbb alakot veszi fel:

$$\vec{P}(X,Y) = \vec{eye} - f \cdot \vec{w} + (X - X_{\max}/2) \cdot \vec{u} + (Y - Y_{\max}/2) \cdot \vec{v}. \tag{9.30}$$

Egyetlen kamerát tehát a $[\vec{eye}, \vec{u}, \vec{v}, f, X_{\max}, Y_{\max}]$ paraméterekkel jellemzünk. Ezek a paraméterek nem függetlenek egymástól, hiszen például az $\vec{u}, \vec{v}, \vec{w}$ egymásra merőleges egységvektorok. A három bázisvektort három skalárral is azonosíthatjuk, ha megadjuk, hogy az eredeti bázisvektorokat milyen csavaró–billentő–forduló szögekkel kell elforgatni ahhoz, hogy azok éppen a kameraorientációt meghatározó vektorokkal essenek egybe.

Ne zavarjon meg bennünket, hogy a valódi kamerában a film a lencse mögött van, a virtuális kamerában viszont a filmet képviselő ablak a lencsét képviselő szem és a tárgy között helyezkedik el! Ha a valódi kamera filmjét gondolatban tükrözzük a lencsére, akkor éppen a virtuális kamera elrendezését kapjuk.

Egy kamerában azok az $\vec{r} = [x, y, z]$ pontok vetülhetnek az X, Y pixel helyét jelentő $\vec{P}(X,Y)$ pontra, amelyek a szemből induló és a $\vec{P}$ ponton keresztülmenő egyenesen vannak, azaz kielégítik ezen egyenes következő egyenletét:

$$[x, y, z] = \vec{eye} + (\vec{P} - \vec{eye}) \cdot t. \tag{9.31}$$

Amennyiben egy $[x, y, z]$ pont képe az első kamera $[X_1, Y_1]$ pixelére, a másik kamerában pedig az $[X_2, Y_2]$ pixelre esik, akkor a pont a következő két vetítőegyesen található:

$$\begin{aligned}
[x,y,z] &= \vec{eye}_1 + (\vec{P}_1(X_1,Y_1) - \vec{eye}_1) \cdot t_1, \\
[x,y,z] &= \vec{eye}_2 + (\vec{P}_2(X_2,Y_2) - \vec{eye}_2) \cdot t_2.
\end{aligned}$$

Ezen egyenletekből az $[x, y, z]$ koordinátákat kifejezve a $\vec{P}_1$ és $\vec{P}_2$ pontokon keresztül látható pontot a háromdimenziós térben visszaállítottuk. Ha a kamerák paramétereit

ismerjük, akkor ebben az egyenletrendszerben öt ismeretlen (x,y,z,t_1,t_2) található. Az egyenletek száma viszont hat (két egyenlet x,y,z-re), azaz az egyenletrendszer túlhatározott. Ebben semmi meglepő sincsen, hiszen abból a feltételből, hogy a két vetítőegyenes metszi egymást, következik, hogy a két szempozíció, a $\vec{P}_1$ és $\vec{P}_2$ pixelpontok és az $[x,y,z]$ pont egy síkon, az úgynevezett *epipoláris sík*on (9.41. ábra) vannak, azaz a hat egyenlet nem független egymástól. A mérési pontatlanságok miatt azonban előfordulhat, hogy az egyenletekben ez a függés megsérül, és így az egyenletrendszernek nem lesz megoldása. Szemléletesen ez annak az esetnek felel meg, amikor a két vetítőegyenes kitérő. Ebben az esetben a két kitérő egyenes legközelebbi pontjainak a felezőjét tekinthetjük közelítő megoldásnak. A másik lehetőség az, hogy az epipoláris síkot az egyik vetítőegyenes és a két szempozíciót összekötő egyenes síkjának tekintjük, amelyre vetítjük a másik vetítőegyenes $\vec{P}$ pontját. Ezzel a metszést egy síkbeli feladatra vezettük vissza. A síkban viszont két egyenes biztosan metszi egymást, hacsak nem párhuzamosak. Erre a témakörre épül az *epipoláris geometria* elmélete, amelynek részleteivel itt nem foglalkozunk [129].

Idáig feltételeztük, hogy a kameraparaméterek rendelkezésre állnak, de nem mondtunk semmit arról, hogy honnan kaphatjuk meg ezeket. A paraméterek közvetlen mérése elég nehézkesnek tűnik (fókusztávolság, film pontos mérete stb.) ezért közvetett módszert alkalmazunk. Ennek az eljárásnak nem csak a mozgáskövetésnél van jelentősége, hanem akkor is, ha a virtuális világunkat a valós világ képeivel szeretnénk kombinálni. A módszer részleteivel a következő fejezetben foglalkozunk.

9.15. Valós és virtuális világok keverése

A fényképezésre és a számítógépes grafikára idáig mint két független műfajra tekintettünk, az előbbi a valós világ tárgyait, az utóbbi pedig a virtuális világ elemeit fényképezi le. Mindkét módszernek megvannak a maga előnyei és korlátai. A valóságban könnyen elérhető tárgyak (tájak, emberek, egyszerű díszletek) virtuális megvalósítása felesleges, hiszen „csak” exponálnunk kell a kezünkben lévő kamerával. Nem létező épületeket, városokat viszont könnyebben építhetünk fel virtuálisan, mint valóságosan, a virtuális tűzvész és robbanás is sokkal kevésbé veszélyes. Űrbéli és fantasztikus jeleneteknél pedig a számítógépes grafika helyett szóba sem jöhet a valós terek fényképezése.

A két módszer előnyeit jó lenne egyszerre élvezni, ami akkor lehetséges, ha a valóságban rögzíthető elemeket (szereplők mozgása, szereplők használati tárgyai) hagyományos filmtechnikával vesszük fel, a valóságban nehezen létrehozható részeket (járművek, épületek, fantasztikus tájak, nagylétszámú hadseregek, veszélyes jelenetek stb.) pedig a számítógépes grafika eszközeivel hozzuk létre. Végül a két képet (filmet) kombináljuk (*compositioning*) egy maszk segítségével, amely kijelöli, hogy mely képpontokban szeretnénk a valós és melyekben a virtuális világot látni. A maszk elkészí-

9.42. ábra. *Illesztetlen (bal) és illesztett (jobb) kamerával készített képek*

tésének legismertebb módszere a kék háttér (*blue-box*) eljárás. A valós világ szereplőit kék háttér előtt vesszük filmre, gondosan ügyelve arra, hogy ruházatuk véletlenül se tartalmazzon kék színű darabokat. A kompozitálás a kék színű pixeleket cseréli ki a virtuális világból érkező kép pixeleivel.

A valós és a virtuális világból származó képelemek nem ütnek el egymástól, ha hasonló körülmények között fényképeztük le őket. A körülményeken egyrészt a kamera beállítást, másrészt a fényviszonyokat értjük. Amíg a számítógépes képszintézisben a kamera és fényforrásparaméterek, valamint a tér optikai tulajdonságai a kezünkben vannak, a valós világban ezek a tulajdonságok nem, vagy csak nagyon pontatlanul mérhetők közvetlenül. Közvetett eljárásokra kell támaszkodnunk, amely a született kép alapján következtet a beállításokra.

Tekintsük először a kameraparaméterek meghatározását! Az előző fejezetben már láttuk, hogy a kamerát az adott modellben a szempozíció, az eredeti bázisvektorokat a kameraorientációba forgató csavaró–billentő–forduló szögek, a fókusztávolság, és a kép felbontása határozza meg. Külön hangsúlyozzuk, hogy ez egy adott kameramodell mellett érvényes, amely nem veszi figyelembe például a lencse torzítását [129].

A felbontást a kép digitalizálásánál magunk állíthatjuk be, ezért ezt ismerjük. A következő, egymástól független paramétereket kell megmérni: az $\vec{eye}$ szempozíció, az eredeti bázisvektorokat a kameraorientációba forgató csavaró–billentő–forduló szögek, és az f fókusztávolság.

A *kamerakalibráció*nak nevezett közvetett mérési eljárás ismert $[x,y,z]$ pontoknak (*marker*eknek) a képen lévő helye alapján számítja ki a kamerát definiáló paramétereket. A markerek lehetnek kitűzött, azonosítható színű tárgyak, minták, vagy akár a környezeti elemek jól felismerhető pontjai is (a 9.42. és 9.43. ábrákon a követ négyzetrácsának sarkait használtuk markerként). A markerek helyét a térben egy önkényesen felvett

referenciaponthoz képest megmérjük (a képen szereplő egyik marker például a felszínen a bejárattól, az ajtóra merőlegesen 5 méterre, az ajtóval párhuzamosan pedig 3 méterre található). A markereket a képen megkeresve megkaphatjuk a hozzájuk tartozó pixelkoordinátákat. Egy markernek ismerjük az $\vec{r} = [x, y, z]$ háromdimenziós koordinátáit és P vetületének az $[X, Y]$ pixelkoordinátáit is, amelyeket behelyettesíthetünk a 9.31. egyenletbe:

$$\vec{r}[x, y, z] = \vec{eye} + (\vec{P}[X, Y] - \vec{eye}) \cdot t.$$

Mindkét oldalból kivonva az $\vec{eye}$ kamerapozíciót és felhasználva a 9.30. egyenletet, a következő alakot kapjuk:

$$\vec{r}[x, y, z] - \vec{eye} = (-f \cdot \vec{w} + (X - X_{\max}/2) \cdot \vec{u} + (Y - Y_{\max}/2) \cdot \vec{v}) \cdot t.$$

Szorozzuk meg az egyenletet skalárisan rendre az $\vec{u}$, a $\vec{v}$ és a $\vec{w}$ egységvektorokkal! Mivel az $\vec{u}, \vec{v}, \vec{w}$ egymásra merőleges egységvektorok, skaláris szorzataik zérus eredményt adnak, így ezek a vegyes tagok eltűnnek az egyenletekből. Végül a t paraméter kiküszöbölése után a következő alakot kapjuk:

$$\frac{\vec{u} \cdot (\vec{r} - \vec{eye})}{\vec{w} \cdot (\vec{r} - \vec{eye})} = -\frac{(X - X_{\max}/2)}{f}, \qquad \frac{\vec{v} \cdot (\vec{r} - \vec{eye})}{\vec{w} \cdot (\vec{r} - \vec{eye})} = -\frac{(Y - Y_{\max}/2)}{f}.$$

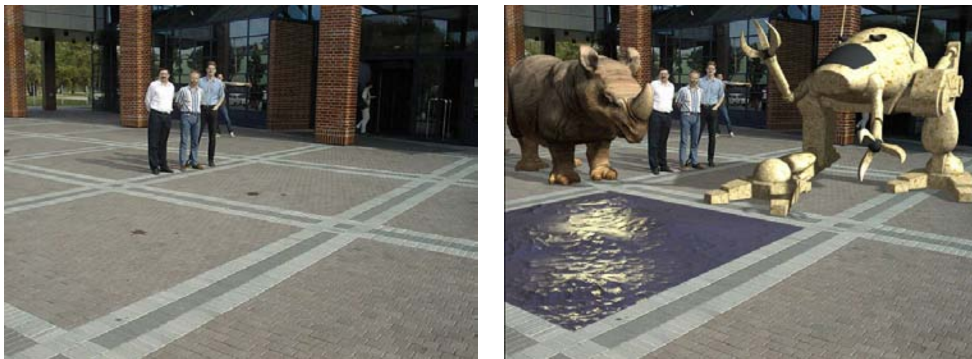
Ugyanezeket az egyenleteket a vektorok koordinátáival is felírhatjuk:

$$\frac{u_x \cdot (x - eye_x) + u_y \cdot (y - eye_y) + u_z \cdot (z - eye_z)}{w_x \cdot (x - eye_x) + w_y \cdot (y - eye_y) + w_z \cdot (z - eye_z)} = -\frac{(X - X_{\max}/2)}{f},$$

$$\frac{v_x \cdot (x - eye_x) + v_y \cdot (y - eye_y) + v_z \cdot (z - eye_z)}{w_x \cdot (x - eye_x) + w_y \cdot (y - eye_y) + w_z \cdot (z - eye_z)} = -\frac{(Y - Y_{\max}/2)}{f}.$$

A kameraorientációt meghatározó $\vec{u} = [u_x, u_y, u_z]$, $\vec{v} = [v_x, v_y, v_z]$ és $\vec{w} = [w_x, w_y, w_z]$ vektorok az ugyancsak ismeretlen csavaró–billentő–forduló szögekből kifejezhetők, így ennek az egyenletnek az ismeretlenei éppen a mérendő kameraparaméterek (az $\vec{eye}$ szempozíció, az eredeti bázisvektorokat a kameraorientációba forgató csavaró–billentő–forduló szögek, és az f fókusztávolság). Mivel a szempozíció mindhárom koordinátájára szükségünk van, összesen 7 ismeretlen mennyiséget kell meghatároznunk. Egyetlen marker segítségével két egyenletet állítottunk fel, így legalább 4 markerre van szükségünk, hogy a megoldás egyértelmű legyen. Használhatunk négynél több markert is, amellyel a mérési hibákat tovább csökkenthetjük.

A fényviszonyok illesztésének érdekében a valós világ fényforrásait kell megismernünk. Ehhez egy tükröző gömböt (egy nagyobbacska karácsonyfadíszt) teszünk a tér közepére, és fényképeket készítünk róla. A gömb fényes pontjaiból egyszerű tükrözéssel a fényforrások iránya meghatározható. A gömb átlagos fényességéből pedig az ambiens fényt kapjuk meg.



9.43. ábra. *Virtuális és valós világok kameráinak és fényviszonyainak illesztése [20]*

10. fejezet

Számítógépes játékok

Egy játék adott környezetben, adott szabályok szerint, több játékos (ellenfél) részvételével folyik. A játékosok hatnak egymásra (látják, hallják, megérintik egymást, lőnek egymásra stb.), és cselekedeteiket az őket ért hatások befolyásolják.

A számítógépes játék is ilyen, amely azonban egy *virtuális világ*ban zajlik. A virtuális világ környezetét és az ellenfeleket egy számítógépes program hozza létre. A számítógépes játékban tehát egy program objektumai versengenek egymással. Az objektumok közül egy magát a játékost testesíti meg. A virtuális valóság rendszerekben és a játékokban a felhasználót a virtuális világban képviselő objektumot *avatár*nak nevezik. A játékos az avatárt a beviteli eszközökön (billentyűzet, egér) keresztül irányítja, mialatt az avatár szemével lát és a fülével hall. A játékos tehát beépül egy programobjektum működésébe, és ezáltal vesz részt a játékban. Az ellenfelek kizárólag szoftver „érzékszervekkel” rendelkeznek, üzenetekkel információkat szereznek a többi objektum állapotáról (látnak), majd ennek megfelelően „erőt” fejtenek ki, és végül a külső és belső erők szerint mozognak. Miként minden játékot, a számítógépes játékot is szabályok tartják keretek között. Ilyen szabályok lehetnek a fizika törvényei, vagy az is, hogy egy fal mögül nem látjuk az ellenfelünket, és a falakon nem hatolhatunk keresztül. A természet törvényeihez azonban nem kell körömszakadtáig ragaszkodni, azokon tetszés szerint könnyíthetünk is. A játék akkor izgalmas, ha a játékosnak van esélye győzni, de a győzelemért keményen meg kell dolgoznia. Ehhez a játékost megtestesítő szoftverobjektum képességeit (például a kifejthető erőt, a kilőhető golyók számát és pusztító erejét stb.), valamint az ellenfelek képességeit és „intelligenciáját” arányosan kell beállítani. Az ellenfelek programobjektumok, így helyettük a program „gondolkodik” *mesterséges intelligencia* algoritmusok segítségével.

Összefoglalva, egy játék készítéséhez a következő feladatokat kell megoldani:

- *Képszintézis*: az ellenfeleket és a környezetet az avatár helyéről le kell fényképezni és az eredményt meg kell jeleníteni. Ez a számítógépes grafika területe, amellyel ebben a könyvben már meglehetősen sokat foglalkoztunk. A korai játékok

a megjelenítés során 2D rajzolást használtak, majd a kétdimenziós képelemeket olyan ügyesen cserélgették, hogy azok háromdimenziós mozgó tárgyaknak nézzenek ki (*2.5D játékok*). Ebben a könyvben csak a modern, valódi *3D játékok*kal foglalkozunk, amikor a játék háromdimenziós térben zajlik, amit 3D képszintézissel jelenítünk meg.

- *Hangszintézis*: a történéseket az avatár a „fülével” hallja, amit a hangszórókon keresztül a valódi játékosnak tovább kell adni. A megoldást a számítógépes hangelőállítás (vagy akár beszédszintézis) adja, amely önmagában is óriási témakör, ezért ezzel a területtel ebben a könyvben nem foglalkozunk [53, 14].

- *Bemeneti illesztés*: a játékost a billentyűzeten és az egéren keresztül hozzá kell kapcsolni az őt a virtuális világban képviselő avatárhoz.

- *Szimuláció*: a játékost a bemeneti parancsok szerint, az ellenfeleket és a környezetet pedig a saját szabályaiknak megfelelően folyamatosan működtetni kell. A virtuális világ szereplőit hatások érik, amelyek megváltoztatják mozgásállapotukat és meghatározzák a cselekedeteiket. A mozgásállapotok követését *mozgás szimulációnak* vagy *fizikai szimulációnak* nevezzük, a cselekedetek pedig a gondolkozás, azaz a mesterséges intelligencia algoritmusok eredményei. A játékprogram sorra veszi a virtuális világ elemeit, kiszámítja az egyes résztvevőket érő hatásokat és az azokra adott válaszokat. A válaszok a mozgásállapotot módosítják, amit ugyancsak követni kell. A játékos bekapcsolása érdekében az avatár megváltozott helyzete alapján a kamera beállításait megváltoztatjuk, majd a virtuális világot és az ellenfeleket az új helyükön fényképezzük le. Ezt a műveletsort aztán ciklikusan ismételgetjük. A ciklust *játékhurok*nak (*game loop*) vagy *szimulációs hurok*nak nevezik. A játékhurok egy ciklusának végrehajtásához idő kell, ezért a játékprogram csak diszkrét időpillanatokban nézhet rá a szereplőkre, és végezheti el a szimulációs lépéseket.

  A szimulációs ciklus végrehajtási ideje függ a szereplők számától, kezelésük bonyolultságától, a számítógépünk teljesítményétől és a többi program okozta terheléstől is. Nem volna szerencsés, ha ettől függően a játék hol lassabban, hol pedig gyorsabban zajlana, ezért a szimulációs ciklusban le kell kérdezni a számítógép óráját, és a szimulációs műveleteket az eddig a pillanatig eltelt valós időnek megfelelően kell végrehajtani. Egy autószimulátor esetében például helytelen volna minden szimulációs ciklusban az autót 1 méterrel arrébb helyezni, hiszen egy lassú gépen az autónk éppen csak vánszorogna, egy gyors gépen viszont kiviharzana a képernyőről. Ehelyett az előző ciklus óta eltelt tényleges idő és az autó pillanatnyi sebessége alapján kell az új helyzetet kiszámolni. Lassabb gépeken esetleg a mozgás a század eleji filmekhez hasonlatosan kevésbé lesz folyamatos, de legalább a szereplők ugyanolyan sebességgel mozognak. Két egymás utáni

ciklus kezdete közötti időt *keretidő*nek nevezzük. A keretidő a játék során ingadozhat a játék és más programok terhelésének megfelelően.

10.1. A felhasználói beavatkozások kezelése

A játék indulásakor a program a képernyőn egy ablakot nyit meg, amelybe az avatár által látható világ képét teszi, a felhasználói beavatkozásokat pedig az avatárhoz vezeti. A játékok (és általában a *virtuális valóság* rendszerek) megvalósításakor az általános animációs feladatokon túl a folyamatos felhasználói kapcsolatra is figyelnünk kell. Nézzük meg, hogy ez mit jelent az eseményvezérelt felhasználói felületek szempontjából! Egy eseményvezérelt rendszerben a felhasználói események hatására az ablakozó program meghívja az alkalmazás eseményvezérlő függvényét. Ez a séma két problémát is felvet.

Ha a felhasználó folyamatosan nyomva tart egy billentyűt, az ablakozó program a lenyomás pillanatában küld egy megfelelő eseményt a programnak, majd vár egy kicsit, és ha a felhasználó még mindig nem engedte el a billentyűt, akkor mindaddig ontja a billentyűzet eseményeket, amíg a felhasználó el nem engedi azt. Ezzel nem is volna semmi baj, hiszen a folyamatosan nyomva tartott billentyűvel a felhasználó a gyors billentyű ütögetést szeretné kiváltani. A problémát az ablakozó rendszer kezdeti várakozása okozza. Ez ugyanis a játék működésében pillanatnyi fennakadást okozhat, ami pedig a közelharcban végzetes lehet. Az ablakozó rendszer azért működik így, mert eredetileg irodai alkalmazásokhoz, és nem játékhoz tervezték.

A második nehézség pedig abból származik, hogy a játékhurkot végrehajtó üresjárati eseménykezelő a többi eseménykezelőtől függetlenül fut, tehát ha eseményvezérelt sémát használnánk, a felhasználói eseményre más függvényben kell reagálnunk, nem pedig akkor, amikor a játékhurokban éppen az avatár feldolgozásánál tartunk.

Mindkét problémát megoldhatjuk, ha a bemeneti eszközöket nem eseményvezérelt eljárással, hanem *lekérdezéses módszer*rel (*polling*) kezeljük. Ahelyett, hogy arra várnánk, hogy az ablakozó rendszer értesítsen minket a billentyűzet esemény bekövetkezéséről, az avatár feldolgozása során ellenőrizzük, hogy a klaviatúra mely billentyűi vannak lenyomott illetve elengedett állapotban. Egyes eseményvezérelt könyvtárak (Ms-Windows) maguk is segítik a lekérdezéses eljárást. Más esetekben viszont csak a billentyű lenyomásáról, illetve elengedéséről kapunk értesítést (*Java/AWT*), amelyet nyomon követve magunk készíthetünk lekérdezéses interfészt. Ekkor az eseményvezérelt interfészre egy újabb réteget építünk, amely a bejövő események alapján nyilvántartja az egyes billentyűk állapotát, így azok már tetszőleges pillanatban lekérdezhetők. Vegyünk fel egy tömböt, amelyben az elemek az egyes billentyűk állapotait mutatják (lenyomott/elengedett)! A billentyű lenyomás és elengedés események hatására az állapotokat billegtetjük, a program pedig az aktuális állapotokat kérdezheti le.

A lekérdezéses billentyűzet kezelés megvalósítása érdekében az idáig kialakított alkalmazás osztályunkat (2.5.3. fejezet) egy kicsit módosítani kell:

```
//===============================================================
class Application {
//===============================================================
public:
   static  Application*    gApp;       // az alkalmazás objektum címe
   static  ApplicationType applicationType; // GLUT vagy Ms-Windows
   static  void CreateApplication();   // az alkalmazás belépési pontja

   char    windowTitle[64];            // ablak címe
   long    windowWidth, windowHeight;  // ablak mérete
   bool    keys[256];                  // billentyűk állapota
   bool    mousePressed;               // egérbillentyű lenyomva?
   long    mousePosX, mousePosY;       // egér pozíciója
   float   mouseSensitivity;           // egérmozgáshoz rendelt érzékenység
   float   time;                       // eltelt idő

   Application(char* windowTitle, long width, long height);
   virtual void Init() {};             // az ablak első megjelenése után hívjuk
   virtual void Render() {};           // színtér kirajzolása végett hívják
   virtual void Do_a_Step(float dt) {};    // animáció egy lépése
   virtual void KeyPressed(KeyCode key) {} // üzenet a billentyű lenyomásáról
   virtual void MouseMotion(int x, int y); // üzenet az egér mozgatásáról

   bool GetKeyStatus(int key);         // egy billentyű lenyomott-e?
   void SwapBuffers();                 // buffercsere
};
```

A 2.5.3. fejezetben megismert és a 9.1. fejezetben továbbfejlesztett alkalmazás osztályt kiegészítettük a billentyűzet és az egér állapotát tartalmazó változókkal, mint például az egér billentyű lenyomását jelző `mousePressed` változóval és az egérkurzor pillanatnyi helyét mutató `mousePosX` és `mousePosY` változókkal. A `mouseSensitivity` egy beállítható konstans, amellyel az egérmozgásnak az alkalmazásra tett hatását hangolhatjuk. A lekérdezéses módszernél csak ezekre a változókra támaszkodunk, illetve felhasználhatjuk a `GetKeyStatus()` függvényt, amely egy billentyűre eldönti, hogy az lenyomott állapotban van-e. Az eseményvezérelt üzemmódban az esemény reakciókat a `KeyPressed()` és a `MouseMotion()` virtuális függvényeket átdefiniáló eljárásokban fogalmazhatjuk meg.

Az állapotváltozókat globális függvények töltik fel, amelyek szükség esetén meghívják az eseményeket kezelő virtuális függvényeket is. Ezek kialakítása függ az ablakozó rendszer típusától, ezért a programszintű megoldásokat külön tárgyaljuk GLUT és Ms-Windows környezetekre.

10.1.1. A billentyűzet és az egér kezelése GLUT környezetben

A GLUT rendszer eseményvezérelt, amelyben az egyes eseményekhez globális kezelőfüggvényeket rendelhetünk. Most a billentyűzet és az egérkezelés miatt ezen eszközök-

kel kapcsolatos eseményekre is reagálnunk kell, ezért a korábbi „ablak érvénytelen”, „nincs esemény”, „ablak átméretezés” események mellé még a „billentyű lenyomása”, „billentyű elengedése”, „speciális billentyű lenyomása”, „speciális billentyű elengedése”, „egér mozgatás” és „egérgomb lenyomás vagy elengedés” eseményekhez is saját kezelőket regisztrálunk:

```
//-----------------------------------------------------------------
void main(int argc, char * argv[]) {
//-----------------------------------------------------------------
   Application::applicationType = GlutApplication; // GLUT alkalmazás
   glutInit(&argc, argv);               // GLUT inicializálás
   Application::CreateApplication();    // az alkalmazás létrehozása

   glutInitWindowPosition(-1, -1);      // alapértelmezett ablak hely
   glutInitWindowSize(Application::gApp->windowWidth, // ablak méret
                      Application::gApp->windowHeight);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutCreateWindow(Application::gApp->windowTitle);
                                        // eseménykezelők regisztrálása
   glutDisplayFunc(RenderFunc);         // ablak érvénytelen
   glutIdleFunc(IdleFunc);              // nincs esemény
   glutReshapeFunc(WindowReshapedFunc); // ablak átméretezés esetén
   glutKeyboardFunc(KeyboardFunc);      // billentyű lenyomás
   glutKeyboardUpFunc(KeyboardUpFunc);  // billentyű elengedés
   glutSpecialFunc(SpecialKeysFunc);    // speciális billentyű lenyomás
   glutSpecialUpFunc(SpecialKeysUpFunc);// speciális billentyűzet elengedés
   glutMouseFunc(MouseFunc);            // egérgomb lenyomás vagy elengedés
   glutMotionFunc(MouseMotionFunc);     // egér mozgatás

   Application::gApp->Init();           // az alkalmazás inicializálása
   glutMainLoop();                      // GLUT üzenethurok
}
```

Az új eseménykezelők egyrészt a billentyűzet és az egér állapotát jelző változóknak adnak értéket, másrészt az alkalmazás virtuális eseménykezelő függvényeit is meghívják. A `KeyboardFunc()` függvény az ASCII karakterek billentyűit kezeli, és bemeneti paramétereiben megkapja a leütött ASCII billentyű kódját és a grafikus kurzor pillanatnyi helyének koordinátáit is. Az esemény hatására a billentyű lenyomott állapotát a `keys` tömb egy elemében tároljuk, amelyet a lekérdezéses megoldásnál használhatunk fel. Másrészt meghívjuk az alkalmazás `KeyPressed()` függvényét, hogy, ha szükséges, az eseményvezérelt paradigma szerint reagáljon az adott billentyű lenyomására:

```
//-----------------------------------------------------------------
void KeyboardFunc(unsigned char key, int x, int y) { // ASCII karakterek
//-----------------------------------------------------------------
   Application::gApp->keys[key] = true;     // ez a billentyű lenyomva
   Application::gApp->KeyPressed(key);      // esemény
}
```

A `SpecialKeysFunc()` függvénnyel a nem ASCII billentyűk, köztük a játékokban különösen fontos iránybillentyűk lenyomására reagálhatunk:

```
//-----------------------------------------------------------------
void SpecialKeysFunc(int key, int x, int y) { // iránybillentyűt lenyom
//-----------------------------------------------------------------
   KeyCode platformIndependentKeyCode = UnknownKey;
   switch (key) {
   case GLUT_KEY_LEFT:  platformIndependentKeyCode = KeyLeft;  break;
   case GLUT_KEY_RIGHT: platformIndependentKeyCode = KeyRight; break;
   case GLUT_KEY_UP:    platformIndependentKeyCode = KeyUp;    break;
   case GLUT_KEY_DOWN:  platformIndependentKeyCode = KeyDown;  break;
   }
   Application::gApp->keys[platformIndependentKeyCode] = true;
   Application::gApp->KeyPressed(platformIndependentKeyCode);  // esemény
}
```

A billentyűk állapotát jelző `keys` tömbből akkor kell törölni a „lenyomott” állapotot, ha az adott billentyűre „elengedés” esemény érkezett:

```
//-------------------------------------------------------------------
void KeyboardUpFunc(unsigned char key, int x, int y) {
//-------------------------------------------------------------------
   Application::gApp->keys[key] = false;     // ezt a billentyűt elengedték
}
```

A nem ASCII billentyűk elengedését kezelő függvény megvalósítása hasonló. A lenyomáshoz és elengedéshez rendelt eseménykezelők a `keys` tömbben tárolt állapotokat billetgetik. Egy billentyű aktuális állapotát a `GetKeyStatus()` függvénnyel kérdezhetjük le:

```
//-----------------------------------------------------------------
bool Application::GetKeyStatus(int key)  { // billentyű állapot lekérdezése
//-----------------------------------------------------------------
   return keys[key];
}
```

A `MouseMotionFunc()` és a `MouseFunc()` szerepe az egér helyének és gombállapotának a követése. Itt mind a lenyomásról, mind pedig az elengedésről értesülünk, így a lekérdezéses kezelés minden gond nélkül megoldható. A `MouseMotionFunc()` az egér mozgatásáról jelez vissza, ha közben valamelyik gombját lenyomva tartjuk:

```
//-----------------------------------------------------------------
void MouseMotionFunc(int x, int y) { // egeret az egérgombot lenyomva mozgatjuk
//-----------------------------------------------------------------
   Application::gApp->mousePosX = x;       // megjegyezzük a kurzor helyét
   Application::gApp->mousePosY = y;
   Application::gApp->MouseMotion(x, y);   // esemény
}
```

A GLUT a `MouseFunc()` függvényt akkor hívja meg, ha az egér valamely gombjának állapota (lenyomott/elengedett) megváltozik:

```
//-----------------------------------------------------------------
void MouseFunc(int button, int state, int x, int y) {
//-----------------------------------------------------------------
  if ( button == GLUT_LEFT_BUTTON ) {    // melyik gomb
     if ( state == GLUT_DOWN ) {        // lenyomtuk vagy elengedtük?
        Application::gApp->mousePressed = TRUE;      // lenyomott
        Application::gApp->MousePressed(x, y);       // esemény
     } else {
        Application::gApp->mousePressed = FALSE;     // elengedett
        Application::gApp->MouseReleased(x, y);      // esemény
     }
  }
}
```

10.1.2. A billentyűzet és az egér kezelése Ms-Windows környezetben

Az Ms-Windows környezetben a billentyűzetet lekérdezéssel is kezelhetjük. Egy tetszőleges `vkeyCode` virtuális klaviatúrakódú billentyű állapotát a `GetKeyState()` Ms-Windows függvénnyel kaphatjuk meg. Ez egy 16 bites kódot ad vissza, amelynek legnagyobb helyiértékű bitje akkor 1 értékű, ha a billentyű le van nyomva, egyébként zérus. A következő programrészlet az iránybillentyűk állapotát vizsgálja:

```
//-----------------------------------------------------------------
bool Application::GetKeyStatus(int platformIndependentKeyCode)  {
//-----------------------------------------------------------------
   short vkeyCode;
   switch (platformIndependentKeyCode) {
   case KeyLeft:  vkeyCode = VK_LEFT;  break;
   case KeyRight: vkeyCode = VK_RIGHT; break;
   case KeyUp:    vkeyCode = VK_UP;    break;
   case KeyDown:  vkeyCode = VK_DOWN;  break;
   }
   return (GetKeyState(vkeyCode) & 0x8000 != 0); // legfelső bit kiválasztása
}
```

Az egéreseményeket az Ms-Windows üzenetkezelő függvényében dolgozhatjuk fel. Itt a lekérdezéses üzemmódhoz eltároljuk a kurzor pillanatnyi helyét, az eseményvezérelt üzemmódhoz pedig meghívjuk az alkalmazás eseménykezelőit. A kurzor helyének koordinátáit az `lParam` változó alsó és felső részében találjuk, amelyeket a `LOWORD` és a `HIWORD` makrókkal vehetünk ki. Az üzenetkezelő egy lehetséges megvalósítása:

```
//-----------------------------------------------------------------
LRESULT WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
//-----------------------------------------------------------------
   switch (message) {
   case WM_LBUTTONDOWN:    // bal egérgomb lenyomása
      Application::gApp->mousePressed = TRUE;                      // állapot
      Application::gApp->MousePressed(LOWORD(lParam), HIWORD(lParam));// esemény
      break;
   case WM_LBUTTONUP:      // bal egérgomb elengedése
      Application::gApp->mousePressed = FALSE;                     // állapot
      Application::gApp->MouseReleased(LOWORD(lParam), HIWORD(lParam)); // esem.
      break;
   case WM_MOUSEMOVE:      // egér mozgatása
      Application::gApp->mousePosX = LOWORD(lParam);               // állapot
      Application::gApp->mousePosY = HIWORD(lParam);
      Application::gApp->MouseMotion(LOWORD(lParam), HIWORD(lParam)); // esemény
      break;
      ...
   }
}
```

10.2. A játékmotor

A számítógépes játék a virtuális világ objektumainak harca. Az objektumokat a program tárolja és működteti. Például egy űrharc játékban az űr, bolygók, űrhajók, fotonrakéták stb. vesznek részt. A játékok szereplőinek viselkedése, megjelenése, műveletei között a különbségek ellenére számos hasonlóság ismerhető fel. A játék *objektum-orientált programozás*a [85] során érdemes a hasonló működésű szereplőket azonosítani, és azokat az öröklés segítségével kiemelni, hiszen így elkerülhetjük, hogy ugyanazon feladatokat a programozás során többször meg kelljen oldani.

Az olyan általános osztályok összességét, amelyek vélhetőleg sok különféle játékban felhasználhatók, együttesen játék könyvtárnak, vagy *játékmotor*nak (*game engine*) nevezzük. Ebben a fejezetben egy általános játékmotort építünk fel, amely jól használható alapot ad a későbbi fejezetek játékaihoz. Azoknak, akik bonyolultabb játékmotorral szeretnének dolgozni, a *CrystalSpace*[1] vagy a *Fly3D*[2] szabad felhasználású programokat, illetve a http://www.gametutorials.com/ olvasgatását ajánljuk.

10.2.1. A Camera osztály

A játékosnak a virtuális világbeli helyzetét fejezi ki a kamera, amit a `Camera` osztállyal implementálunk. A kamera állapotát a virtuális függvényekben számított `Eye` szempozíció, `Head` nézeti irány és `Up` függőleges irány írják le. A játékban a játékos az

[1] http://crystal.sourceforge.net
[2] http://www.fly3d.com.br

avatár szemével lát, azaz ezeket az állapotváltozókat az avatár tulajdonságaiból számítjuk ki.

Az OpenGL számára a kamerát a nézeti transzformáció határozza meg, amelyet a `SetCameraTransform` függvényben állítunk be.

```
//=================================================================
class Camera { // a játékos ezen a kamerán keresztül lát
//=================================================================
public:
   virtual Vector  Head() { return Vector(0, 1, 0); } // nézeti irány
   virtual Vector  Up()   { return Vector(0, 0, 1); } // függőleges irány
   virtual Vector& Eye() = 0                          // szempozíció

   void SetCameraTransform(Application * app) { // kamera transzformáció
      glViewport(0, 0, app->windowWidth, app->windowHeight);  // képernyő
      glMatrixMode(GL_PROJECTION);          // projektív transzformáció
      glLoadIdentity();
      gluPerspective(90,                    // látószög,
                     (float)app->windowWidth/(float)app->windowHeight,
                     0.01,                  // első vágósík
                     10000.0                // hátsó vágósík
                    );

      glMatrixMode(GL_MODELVIEW);           // modell-nézeti transzformáció
      glLoadIdentity();                     // a kamera helyzete szerint
      gluLookAt(Eye().x, Eye().y, Eye().z,
                Eye().x + Head().x, Eye().y + Head().y, Eye().z + Head().z,
                Up().x, Up().y, Up().z);
   }
        // befoglaló gömb nem látható, biztosan mögöttem van az objektum?
   bool InViewFrustrum(Vector& p, float radius = 0) {
      return ((p - Eye()) * Head() > -radius);
   }
};
```

Az `InViewFrustrum` *nézeti gúla vágás*hoz (*view culling*) alkalmazható. Bár az OpenGL gondoskodik a láthatósági gúlán kívül eső objektumok vágásáról, nem érdemes ezzel terhelni, ha magunk is könnyen felismerhetjük, hogy egy tárgynak biztosan nincs látható része. A vizsgálathoz a tárgy befoglaló gömbjének sugarát (`radius`) használjuk. Ha a tárgy középpontja a szem mögött helyezkedik el, és távolsága nagyobb, mint a befoglaló gömb sugara, akkor biztosan semelyik része sem látható. Az egyes objektumok `Draw` metódusa megkapja a kamera azonosítóját, így az `InViewFrustrum` függvény segítségével könnyen eldöntheti, hogy van-e minimális esélye annak, hogy a képernyőn látható lesz. Ha nincs, akkor nem kell terhelnie az OpenGL-t a vágási feladatokkal.

10.2.2. A GameObject osztály

A játékobjektumokat érintő közös műveleteket a játékhurok tanulmányozásával azonosíthatjuk. A játékhurok a következő feladatokat látja el:

1. Meghatározza az előző ciklus kezdete óta eltelt időt.

2. Lekérdezi a beavatkozó szervek (billentyűzet, egér stb.) pillanatnyi állapotát, és ennek megfelelően vezérli az avatárobjektumot.

3. Sorra veszi a világ egyes objektumait és rábírja őket, hogy saját és társaik állapota alapján éljenek a vezérlési lehetőségeikkel. Például egy űrhajó esetében a pillanatnyi helyzet és a bolygók állása meghatározza az űrhajóra ható gravitációs erőket, az ellenfelek pozíciójának ismeretében pedig az űrhajó bekapcsolhatja vagy leállíthatja a hajtóműveit. Ezen vezérlési lépés végén az egyes objektumokra ható eredő erő ismertté válik. Ha a játékobjektumok egymásra hatása azzal a szomorú következménnyel jár, hogy egy játékobjektum elpusztul (például egy lövedék eltalálja), akkor őt a további játékból ki kell vonni.

4. Eltünteti a halottakat, és felszabadítja a memóriából foglalt helyüket.

5. Ismét sorra veszi az objektumokat és a keretidő, a pillanatnyi pozíció, a sebesség, valamint az objektumra ható eredő erő okozta gyorsulás alapján kiszámítja a pozíció és a sebesség új értékeit.

6. Az avatár pillanatnyi helyzete szerint beállítja a kamerát, és lefényképezi a virtuális világot, majd az eredményt megjeleníti a képernyőn.

Egyetlen játékobjektumon tehát a következő műveleteket kell végrehajtani:

- *Vezérlés* (*ControlIt()*), amely a saját és a többi játékobjektum állapota alapján, valamint a szimulációs időszelet ismeretében él a rendelkezésre álló vezérlési lehetőségekkel (például hajtómű szabályozás).

- *Párbeszéd* (*InteractIt()*), amely az adott objektumot egyetlen másik objektummal veti össze. A vezérlés során a többi játékobjektum állapotának felderítéséhez minden egyes objektumnak minden más objektummal párbeszédet kell kezdeményeznie.

- *Ütközésdetektálás* (*CollideIt()*): A párbeszéd egy speciális fajtája, amelyben felderítjük, hogy két objektum pályája nem keresztezi-e egymást.

- *Típuslekérdezés* (*GetType()*): A játékobjektumok közötti párbeszéd nyilván függ attól, hogy egy szereplő éppen kivel áll szemben (egy jól nevelt cowboy az ajtónál a hölgyeket előzékenyen előreengedi, az ellenséges urakra viszont csípőből tüzel), ezért a párbeszéd kezdetekor az objektum megkérdezi a partnere típusát.

- *Felszabadítás* (*KillIt()*): Az interakció eredményeként az egyes objektumok elpusztulhatnak, amit az `alive` állapotváltozóban jelzünk. Az interakció végén a nem élő szereplők által foglalt memóriaterületet felszabadítjuk.

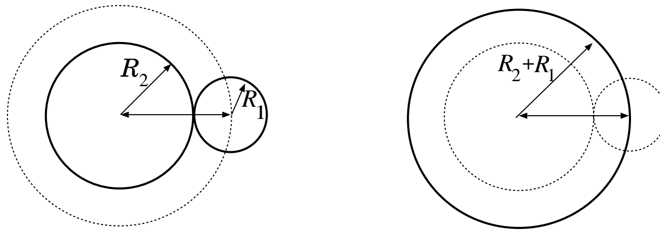
- *Animáció* (*AnimateIt()*), amely az objektumra ható erők és az eltelt idő alapján módosítja az objektum mozgásállapotát.
- *Rajzolás* (*DrawIt()*), amely az objektumot felrajzolja.

Az általános játékobjektumot tehát a következő alaposztály definiálja:

```
//===============================================================
class GameObject { // a játék egy szereplője
//===============================================================
public:
   Vector position, velocity, acceleration;   // pozíció, sebesség, gyorsulás
   bool   alive;                              // életben van-e
   float  bounding_radius;                    // befoglaló gömb sugara

   GameObject(Vector pos0):position(pos0) { alive = TRUE; bounding_radius = 0; }
   virtual void KillIt() { alive = FALSE; }   // megsemmisítés
   virtual int GetType() = 0;                 // típus lekérdezés

   virtual void InteractIt(GameObject * obj) { } // párbeszéd
   virtual bool CollideIt(GameObject * obj, float& hit_time, Vector& hit_point);
   virtual void ControlIt(float dt) { }       // vezérlés a dt hosszú keretben
   virtual void AnimateIt(float dt) {         // mozgásváltozók integrálása
      position += velocity * dt;
      velocity += acceleration * dt;
   }
   virtual void DrawIt(Camera * camera) { }   // rajzolás
};
```

Ezen a szinten még csak azt tudtuk meghatározni, hogy milyen üzenetekre kell reagálniuk az egyes szereplőknek, a reakció mibenlétét csak nagyon kezdetlegesen fogalmaztuk meg. Például alapértelmezés szerint egy objektum nem kezdeményez párbeszédet, nem gondolkodik, állandó gyorsulással mozog, és a képernyőn nem látszik. A műveletek virtuális függvények, amelyeket a tényleges szereplők szükség esetén átértelmeznek. Ha egy adott objektumra a fenti műveletek közül valamelyik lényegtelen, akkor ezekkel a kezdetleges függvényekkel is beérjük. Például az űr nem változik, tehát vezérlése, animálása során nem történik semmi.

Az objektumok egy része kitölti és kizárólagosan lefoglalja a háromdimenziós tér egy részét, amely azért érdekes, mert az ilyen objektumok ütközhetnek egymással, illetve az ilyen objektumot el lehet találni egy másik kiterjedt, vagy pontszerű objektummal (ez ugye egy lövöldözős játék sava-borsa). Az *ütközésfelismerő eljárások* precíz megvalósítását a 9.10.4. fejezetben tárgyaltuk. Az ottani eredményeket használhatnánk most is, de gyakran lényegesen egyszerűbb, közelítő megoldások is kielégítő játékélményt adnak. Közelítsük például a kiterjedt objektumok által elfoglalt térfogatot gömbbel, így a bonyolult geometriák helyett csak a gömbökre végezzük el az ütközés detektálását! A gömb óriási előnye, hogy a tengelye körüli forgatás nem változtatja meg, tehát csak az objektum haladó mozgásával kell foglalkozni.



10.1. ábra. *Ütközésdetektálás két gömb között*

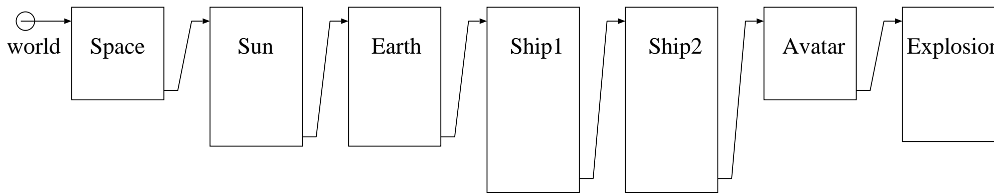
Két gömb alakú test akkor ütközik egymással, ha a középpontjaik távolsága kisebb a gömbök R_1 és R_2 sugarainak összegénél, azaz $R_1 + R_2$-nél (10.1. ábra). Ez viszont akkor következik be, ha az egyik gömb középpontja a másik középpontjába helyezett $R_1 + R_2$ sugarú gömbön átmegy. Ezzel a vizsgálatot visszavezethetjük pont és gömb ütközésvizsgálatára, amit a gömb–sugár metszéspontszámítással (6.3.2. fejezet) oldhatunk meg (a sugár kezdőpontja az első gömb középpontja, az irányvektora pedig a másik gömbhöz viszonyított relatív sebességvektora). Az ütközések felismerését a `CollideIt()` tagfüggvényre bízzuk, amelynek definíciója a következő:

```
//-----------------------------------------------------------------
bool GameObject::CollideIt(GameObject* obj, float& hit_time, Vector& hit_point){
//-----------------------------------------------------------------
   Sphere comb_sphere(obj->position, bounding_radius + obj->bounding_radius);
   Ray ray(position, velocity - obj->velocity); // pályát leíró sugár
   if (comb_sphere.Intersect(ray, hit_time)) {  // ha a sugár metszi a gömböt
      Vector hit_pos = position + velocity * hit_time; // első objektum itt van
      Vector obj_hit_pos = obj->position + obj->velocity * hit_time;
      float a = bounding_radius / (bounding_radius + obj->bounding_radius);
      hit_point = hit_pos * (1-a) + obj_hit_pos * a;   // az ütközés helye
      return TRUE;                                     // ütközés
   } else return FALSE;                                // nincs ütközés
}
```

A függvény bemeneti paramétere az `obj` célobjektum. Ha létezik metszéspont, akkor az ütközés idejét a `hit_time` változó, a találati pontot pedig a `hit_point` változó tárolja. Ebből számítjuk ki, hogy az ütközés pillanatában hol van a két ütköző objektum. Az ütközés pontos helyét a két befoglaló gömb középpontja között, a gömbök sugárarányának megfelelő helyen találjuk.

10.2.3. A Member osztály

A játékban különböző típusú objektumok vesznek részt, amelyeket közös adatszerkezetben (ún. *heterogén kollekció*ban [85]) kell elhelyezni. A játékobjektumok többsége dinamikusan születik, és hal meg, tehát az adatszerkezetnek gondoskodnia kell az új szereplők befogadásáról és az elpusztult tagok helyének felszabadításáról.



10.2. ábra. *Az űrhajós játék objektumait tároló adatszerkezet*

Az egyik legegyszerűbb adatszerkezet, amely megfelel ezen követelményeknek, a *láncolt lista* (10.2. ábra). Megemlítjük, hogy összetettebb játékokban az objektumok alá- és fölérendeltségi viszonyait is ki kell fejezni (például egy puska melyik szereplő fegyvere), amit hierarchikus adatszerkezetekkel, fákkal valósíthatunk meg. Egyelőre azonban a láncolt lista is tökéletesen elegendő, a hierarchikus szerkezetekhez a 10.4. fejezetben térünk vissza.

A láncolt listánk első elemének címét a világot jelentő `world` mutatóban tároljuk. Minden listaelemben szerepel a következő elemre mutató `next` mutató, így a `world` mutatótól indulva az összes elemet meglátogathatjuk és műveleteket végezhetünk rajtuk. Elvileg előfordulhat, hogy a láncolt lista első eleme is elhalálozik, ezért a `world` mutatót át kell helyezni, ami némi adminisztrációt igényel. Ettől megszabadulhatunk, ha a lista elejére egy elpusztíthatatlan szereplőt (például az űrt) teszünk.

Rendeljük a listakezelés képességét és a láncolómutatót egy `Member` osztályhoz, amit a `GameObject` osztályból származtatunk. A `Member` segítségével tehát olyan objektumok definiálhatók, amelyek a szereplők társadalmában is elfoglalhatják méltó helyüket. A játék során az egyes objektumokat egyenként kell vezérelni, animálni és rajzolni, azokkal egyenként kell párbeszédet és ütközésvizsgálatot kezdeményezni. Ezt a `Member` osztály rekurzív vezérlő, animáló, rajzoló, párbeszéd és ütközés ellenőrző függvényeivel valósíthatjuk meg, amelyek meghívják az adott tagra vonatkozó megfelelő függvényt, majd rekurzív módon a következő listaelemre, azaz a lista farokrészére hajtják végre ugyanezt a műveletet. A `Member` osztály definíciója az alábbi:

```
//===============================================================
class Member : public GameObject {
//===============================================================
protected:
   static Member * root;                // a lista első eleme
   Member        * next;                // láncoló mutató
public:
   Member(Vector pos0):GameObject(pos0) { next = NULL; if (!root) root = this; }

   void Join(Member * obj) {            // új elem hozzávétele a listához
      if (next) next->Join(obj);        // ha van farok, új elem a farokhoz
      else next = obj;                  // különben az utolsó helyre
   }
```

```
   void Interact(Member * obj) {          // párbeszéd
      if (obj != this) InteractIt(obj);   // nem beszélünk magunkban
      if (obj->next) Interact(obj->next); // párbeszéd a farokkal
   }

   GameObject * Collide(Member * obj, float& mhit_time,  // ütközésfigyelés
                     Vector& mhit_point, GameObject * source);

   void Control(float dt) {               // vezérlés a dt keretben
      ControlIt(dt);                      // ezt vezéreljük
      if (next) next->Control(dt);        // vezérlés a farokra
   }

   void Animate(float dt) {               // animáció a dt keretben
      AnimateIt(dt);                      // ezt animáljuk
      if (next) next->Animate(dt);        // a farok animációja
   }

   virtual void BeforeDraw() {            // rajzolás prológus
      glPushMatrix();                     // transzformáció mentése
   }

   void Draw(Camera * camera) {           // rajzolás
      BeforeDraw();                       // állapot mentése
      DrawIt(camera);                     // ezt rajzoljuk
      AfterDraw();                        // állapot visszaállítás
      if (next) next->Draw(camera);       // farok rajzolása
   }

   virtual void AfterDraw() {             // rajzolás epilógus
      glPopMatrix();                      // transzformáció visszaállítás
   }

   void BuryDead(Member * exclude);       // halottak helyének felszabadítása
};
```

A `Member` lista láncoló mutatója a `next` változó. Ezen kívül a `root` statikus tag a lista első elemének a címét tárolja, amit egy elem akkor használ, ha a többi szereplővel kapcsolatba kíván lépni. A `Join` tagfüggvény a lista végére egy új elemet tesz. A vezérlő `Control()`, az animáló `Animate()` és a rajzoló `Draw()` függvények a lista minden egyes elemére meghívják az objektumonkénti műveletet elvégző függvényeket (`ControlIt()`, `AnimateIt()` és `DrawIt()`). A rajzolófüggvényt kiegészítettük egy felüldefiniálható `BeforeDraw()` prológus és egy `AfterDraw()` epilógus rutinnal, amelyek az alapértelmezésük szerint elmentik majd visszaállítják az aktuális transzformációs mátrixot. Erre azért van szükség, mert a listában egymás után elhelyezett objektumok egymástól függetlenül mozognak, így az egyik transzformációi nem hathatnak a többi elemre. A prológus és epilógus rutinokat akkor kell megváltoztatni, ha más állapotjellegű tulajdonságokat is menteni kívánunk. A párbeszédért felelős `Interact()` és az ütközést ellenőrző `Collide()` az adott objektumot a lista összes többi elemével összeveti.

Az ütközésdetektálás megvalósítása:

```
//-----------------------------------------------------------------
GameObject * Member::Collide(Member * obj, float& mhit_time,
                             Vector& mhit_point, GameObject * source) {
//-----------------------------------------------------------------
   GameObject * hit_obj = NULL;    // először a farok elemeivel ütköztetünk
   if (obj->next) hit_obj = Collide(obj->next, mhit_time, mhit_point, source);
   float  hit_time;                // ezen objektummal az ütközés ideje
   Vector hit_point;               // ezen objektummal az ütközés helye
   if (obj != this && obj != source && // magunkkal és a forrással nem ütközünk
       CollideIt(obj, hit_time, hit_point) &&  // van ütközés ?
       hit_time < mhit_time) {     // korábbi-e mint az előző ütközések ?
      mhit_time = hit_time;
      mhit_point = hit_point;
      hit_obj = obj;
   }
   return hit_obj;
}
```

A `Collide()` összehasonlítja az adott objektumot az `obj` célobjektummal, és ha ezzel az objektummal korábban történik ütközés, mint az idáig feldolgozottakkal, akkor az új ütközési időt az `mhit_time` változóban, az ütközés helyét pedig az `mhit_point` változóban adja vissza. A `source` változóban egy objektumot adhatunk meg, amelyet ki szeretnénk zárni az ütközésvizsgálatból. A kivételezés szükségességét egy példával illusztráljuk. Amikor egy puskagolyó megszületik és áldásosnak nem nevezhető tevékenységét megkezdi, akkor a puskához, azaz a forrásobjektumához nagyon közel van. A számítási pontatlanságok miatt ezért azt találhatjuk, hogy a golyó rögtön a születésekor eltalálja a forrásobjektumát. Az ember pedig nem azért lövöldöz, hogy saját magát terítse le, nemde? Ilyen esetekben érdemes a puskát — a golyó *forrásobjektum*át — kivenni a lehetséges céltárgyak közül.

Az utolsó tagfüggvény az elpusztult objektumok helyét szabadítja fel:

```
//-----------------------------------------------------------------
void Member::BuryDead(Member* exclude) { // halottak helyének felszabadítása
//-----------------------------------------------------------------
   for(Member * m = root; m->next != NULL; m = m->next) {
      if (m->next->IsAlive() == FALSE && m->next != exclude) {
         Member * dead = m->next;
         m->next = m->next->next;
         delete dead;
         if (m->next == NULL) break;
      }
   }
}
```

A temetési szertartást elvégző `BuryDead` függvény bemeneti paraméterével kijelölhetünk egy olyan szereplőt, akit sohasem temetünk el. Ez a szereplő a játékost

megtestesítő avatár lesz. Erre a megkülönböztetésre azért van szükség, mert a felhasználó a virtuális haláláról értesítést kap, amit még „holtában” is nézhet. Sőt, a játékos akár fel is támaszthatja saját virtuális alteregóját.

A `Member` osztályok szolgáltatásaira minden játékobjektum igényt tart. A továbbiakban figyelembe vesszük a játékobjektumok közötti eltéréseket is, az öröklési hierarchiát tehát több szálra bontjuk szét.

10.2.4. Az Avatar osztály

A `Member` típus egy fajtája a játékost megszemélyesítő `Avatar`. Az `Avatar` példányának a felhasználói kapcsolattartás miatt, a többi játékobjektumhoz képest még két feladata van: a beavatkozó szervek állapota alapján vezérli a működését, illetve a saját helyzete szerint beállítja a rajzolás kameráját.

```
//===============================================================
class Avatar : public Member, public Camera {
//===============================================================
public:
   Avatar(Vector pos0) : Member(pos0) {}
   virtual void ProcessInput(Application * input) = 0; // felhasználói vezérlés
   Vector& Eye() { return position; }           // avatár pozíció = szempozíció
   Vector Steering(Application * input);        // kormányzás
};
```

A `ProcessInput()` beavatkozó szervek állapotát kérdezi le, és ennek megfelelően működteti az avatárt. A beavatkozó szervek hatása a konkrét játéktól függ, ezért egyelőre csak a függvény interfészét írjuk le, tényleges tartalmat csak később, az egyes játékoknál kap. A játékok különbözősége ellenére az avatárt gyakran hasonlóan kormányozzuk, az egér vagy iránybillentyűk segítségével. Ezért az általános avatár osztály egy `Steering()` kormányzó függvényt bocsát rendelkezésre, hogy azt ne kelljen minden játékban külön elkészíteni. A kormányzás az iránybillentyűk vagy az egérkurzor helyzete szerint egy haladási irányváltozást számít ki.

```
//---------------------------------------------------------------
Vector Avatar::Steering(Application * input) {   // kormányzás
//---------------------------------------------------------------
      // a klaviatúra iránybillentyűivel kormányozunk
   if (input->GetKeyStatus(KeyUp))     return Up() * (-1.0);
   if (input->GetKeyStatus(KeyDown))   return Up();
   if (input->GetKeyStatus(KeyLeft))   return Up() % Head();
   if (input->GetKeyStatus(KeyRight))  return Head() % Up();
      // ha az egérgombot lenyomtuk, akkor az egérrel kormányozunk
   if (input->mousePressed) {
      float width = input->windowWidth, height = input->windowHeight;
      float dx = (2 * input->mousePosX - width) / width;
      float dy = (2 * input->mousePosY - height) / height;
      return ((Head() % Up()) * dx + Up() * dy) * input->mouseSensitivity;
   }
```

```
   return Vector(0, 0, 0); // nem nyúltunk a kormányhoz
}
```

10.2.5. A TexturedObject osztály

Az objektumokat fel is kell rajzolni a képernyőre, amit az objektum geometriája és megjelenítési attribútumai alapján tehetünk meg. Játékokban gyakori, hogy nem bajlódunk fényforrásokkal és illuminációs képletekkel, minden szereplőt a saját színével rajzolunk. Annak érdekében, hogy a látvány mégis izgalmas legyen, a felületekre textúrákat húzunk. A textúrát egy képfájlból tölthetjük be. A különböző típusú képfájlok (TARGA, BMP, PCX stb.) kezelését a `Texture` osztállyal végeztethetjük el:

```
//===============================================================
class Texture {
//===============================================================
protected:
   unsigned int texture_id;   // aktuális OpenGL textúra azonosító
public:
   Texture(char * filename, bool transparent = FALSE);
   unsigned int Id() { return texture_id; }
};
```

Az osztály konstruktora a megadott nevű fájlból betölti a textúra képét, amit átlátszatlan textúráknál RGB textúraként, átlátszó képeknél viszont RGBA textúraként használ fel.

Ebben a megoldásban minden textúrát külön tárolunk, és azokat külön adjuk át az OpenGL-nek is. Figyelembe véve, hogy gyakran nagyon sok objektumhoz tartozik ugyanaz a textúra (például akkor, ha egyetlen típusnak sok példánya van), ez a megoldás feleslegesen sok helyet pazarol a textúrák raktározására. Érdemes ezért az osztályt úgy továbbfejleszteni, hogy figyelje, hogy a kért textúrát betöltötte-e már, és ha igen, akkor az újbóli létrehozás helyett csak a korábbi változatra hivatkozik. Ezt a megoldást nevezik *raktár*nak:

```
//===============================================================
class Texture {  // textúra raktárakkal
//===============================================================
    static char         names[MAXTEXTURES][64];// tárolt fájl nevek
    static unsigned int ids[MAXTEXTURES];      // OpenGL textúra azonosítók
    static int          nid;                   // lefoglalt textúrák száma
    int texture_id;                            // aktuális textúra azonosító
public:
    Texture(char * filename, bool transparent = FALSE);
    unsigned int Id() { return texture_id; }
};
```

Az osztály interfésze változatlan, ezért a program többi része nem is veszi észre, hogy helytakarékosan kezeljük a textúrákat. A raktározás magja a konstruktorban van:

```
//-----------------------------------------------------------------
Texture::Texture(char * filename, bool transparent) {
//-----------------------------------------------------------------
    glEnable(GL_TEXTURE_2D); // textúrázás engedélyezése az OpenGL-ben
    if ( nid == 0 ) {        // ha még nem vettünk fel textúrát
        glGenTextures(MAXTEXTURES, ids);    // OpenGL textúrák kérése
    }
    for(int i = 0; i < nid; i++) {          // Ha ezt a textúrát már felvettük
        if (strcmp(filename, &names[i][0]) == 0) {
            texture_id = ids[i];            //    akkor csak az id kell
            return;
        }
    }
    texture_id = ids[nid];                  // Ha új fálj
    strcpy(&names[nid++][0], filename);     //    akor megjegyezzük a nevet

    glBindTexture(GL_TEXTURE_2D, texture_id);
    int width, height;
    ImageFile image( filename, width, height ); // képfájl betöltése
    if ( transparent ) {
        gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, width, height,
                          GL_RGBA, GL_UNSIGNED_BYTE, image.LoadWithAlpha());
    } else {
        gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, width, height,
               GL_RGB, GL_UNSIGNED_BYTE, image.Load());
    }
    glDisable(GL_TEXTURE_2D); // textúrázás tiltása
}
```

A képet az `ImageFile` osztály segítségével tölthetjük be egy adott nevű fájlból. A CD-mellékletben megtalálható implementációt *BMP*, *TGA* és *PCX* formátumokra készítettük fel, de ezek részleteivel itt nem foglalkozunk. A `Texture` osztály segítségével már textúrázott, kiterjedt játékobjektumokat hozhatunk létre. Az ilyen objektumok típusát a `TexturedObject` osztály definiálja:

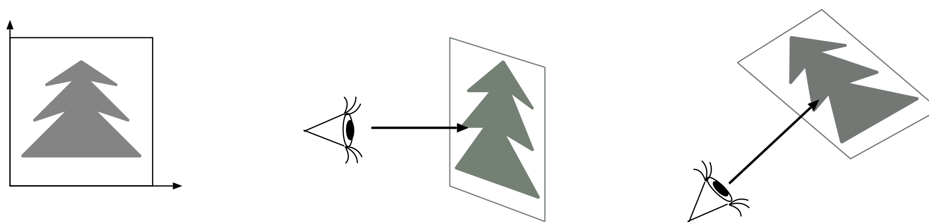
```
//=================================================================
class TexturedObject : public Member {
//=================================================================
   Texture texture;
public:
   TexturedObject(Vector& pos0, char * texture_file, bool transparent = FALSE)
      : Member(pos0), texture(texture_file, transparent) { }

   void BeforeDraw() {
      Member::BeforeDraw();           // transzformációs mátrix mentése
      glEnable(GL_TEXTURE_2D);        // textúrázás engedélyezése
      glBindTexture(GL_TEXTURE_2D, texture.Id());  // textúra választás
      glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
   }
   void AfterDraw() {
      glDisable(GL_TEXTURE_2D);       // textúrázás tiltása
      Member::AfterDraw();            // transzformációs mátrix visszaállítása
   }
};
```

A `TexturedObject` osztályban újraértelmeztük a rajzolás prológus és epilógus rutinjait. A prológusban a transzformációs mátrixok mentésén kívül a textúrázást is engedélyeztük, és bekapcsoltuk az objektumhoz rendelt textúrát. Alapértelmezésben a textúrákban tárolt színek a rajzolási színek (`GL_REPLACE` mód), amit persze a rajzolás során felülbírálhatunk.

10.2.6. Plakátok: a Billboard osztály

Habár ez a könyv a háromdimenziós grafikával foglalkozik, nem mondhat le teljesen a kétdimenziós fényképek alkalmazásáról sem. A valós idejű animációban a képszintézisre jutó idő nagyon kevés, így összetett tárgyak megjelenítésekor mindenképpen korlátokba ütközünk. Mivel egy fénykép semmivel sem lesz bonyolultabb attól, ha összetettebb tárgy képét jelenítjük meg rajta, ezért kézenfekvő, hogy a nagyon bonyolult tárgyakat a fényképükkel helyettesítsük. A következőkben egy ilyen elven működő eljárást ismertetünk, amit a részecskerendszereknél is felhasználunk.

A természet gyakran hoz létre bonyolult geometriájú tárgyakat, amelyeket csak nagyon sok háromszöggel írhatnánk le. Gondoljunk csak egy fára, felhőre, sziklára, robbanásra, tűzre stb! Ilyen esetekben a valós idejű képszintézis sebességi követelményei csalásra kényszerítenek bennünket. A trükk arra építhet, hogy ezek a jelenségek közelítőleg szimmetrikusak, azaz a fontos nézeti irányokból szemlélve őket, hasonló képet mutatnak. A fontos irányokat mindig az alkalmazás határozza meg. Például egy gyalogost mozgató játékban elegendő a hengerszimmetria, hiszen nem repülhetünk a tárgyak fölé és nem áshatjuk be magunkat alájuk. A különböző irányokból hasonló kinézetű tárgyakat pedig helyettesíthetjük a képükkel, így a bonyolult geometria helyett egyetlen téglalapot kell megjeleníteni. A képet tároló téglalap nem lehet rögzített, hiszen ekkor észrevehetően elvékonyodik, ha laposabb szögben tekintünk rá, sőt, ha a szem a téglalap síkjában van, a kép el is tűnhet. Egyik lehetséges megoldásként használhatunk két egymásra merőlegesen elhelyezett képet, vagy pedig a téglalapot mindig úgy forgatjuk, hogy az merőleges legyen a nézeti irányra. Részleteiben csak a dinamikusan forgatott képekkel foglalkozunk.



10.3. ábra. *A plakátok felépítése és forgatása a nézeti iránynak megfelelően*

A *plakát* (*billboard*) egy olyan kétdimenziós kép, amelyet mindig a kamera felé fordítunk a háromdimenziós térben, és ezért egy valódi háromdimenziós test látszatát kelti (10.3. ábra). A plakátot megvalósító osztályt az átlátszó textúrát tároló `TexturedObject` osztályból származtathatjuk:

```
//===============================================================
class BillBoard : public TexturedObject {
//===============================================================
   float size;  // a textúrázott 2D négyzet mérete
public:
   BillBoard(Vector pos0, float size0, char * texture_filename)
      : TexturedObject(pos0, texture_filename, TRUE) { size = size0; }
   void  DrawIt(Camera * camera);
};
```

A plakát forgatásához nézzük végig, hogy a négyszögünk milyen transzformációkon megy keresztül:

1. Modellezési transzformáció, amely szembe forgatja a kamerával a képet, és eltolja az objektum helyére.
2. Nézeti transzformáció, amely úgy tolja el a pontokat, hogy a szem az origóba kerüljön, majd elforgatja úgy a teret, hogy a kamera nézeti iránya a $-z$ tengelyre essen.

Azt szeretnénk, hogy a transzformáció végrehajtása során a szem–objektum távolság az előírt legyen, de a plakátnégyszög a nézeti irányra merőleges legyen, azaz normálvektora a $-z$ irányba mutasson. Ezt két úton is elérhetjük, vagy úgy, hogy a nézeti transzformációt ennek megfelelően állítjuk be, vagy pedig úgy, hogy a nézeti transzformációhoz nem nyúlunk, hanem a téglalapot az aktuális transzformáció alapján „előforgatjuk”.

Nézzük először az első megoldást és tegyük fel, hogy a plakátot egy origó középpontú, `size` méretű, a z tengelyre merőlegesen álló négyzetre ragasztjuk rá! Ekkor a `MODELVIEW` transzformációnak nem szabad forgatnia, csupán a tárgy és a kamera közötti vektorral kell eltolnia a tárgyat. A transzformáció forgatási része a 4×4-es transzformációs mátrix bal felső 3×3-as minormátrixa, amelyet úgy olthatunk ki anélkül, hogy az eltolási részt tönkretennénk, hogy a mátrix inverzével szorozzuk azt. Hála annak, hogy a forgatási mátrix sorai egymásra merőleges egységvektorok, a forgatási mátrixot úgy invertálhatjuk, hogy a főátlójára tükrözzük (transzponáljuk).

```
//-----------------------------------------------------------------
void BillBoard::DrawIt(Camera * camera) {
//-----------------------------------------------------------------
   float viewmatx[4][4]; // MODELVIEW mátrix lekérdezés
   glGetFloatv(GL_MODELVIEW_MATRIX, &viewmatx[0][0]);
   float tmp;            // bal felső minormátrix invertálása (=transzponálása)
   tmp = viewmatx[0][1]; viewmatx[0][1] = viewmatx[1][0]; viewmatx[1][0] = tmp;
   tmp = viewmatx[0][2]; viewmatx[0][2] = viewmatx[2][0]; viewmatx[2][0] = tmp;
   tmp = viewmatx[1][2]; viewmatx[1][2] = viewmatx[2][1]; viewmatx[2][1] = tmp;
   viewmatx[3][0] = position.x;      // eltolás a plakát helyére
   viewmatx[3][1] = position.y;
   viewmatx[3][2] = position.z;
   glMultMatrixf(&viewmatx[0][0]);   // forgatás kioltása + relatív hely

   ,,átlátszósági beállítások engedélyezése''
   glBegin(GL_QUADS);                // plakát négyszögének felrajzolása
   glTexCoord2f(0, 0); glVertex2f(-size, -size);
   glTexCoord2f(1, 0); glVertex2f(size, -size);
   glTexCoord2f(1, 1); glVertex2f(size, size);
   glTexCoord2f(0, 1); glVertex2f(-size, size);
   glEnd();
   ,,átlátszósági beállítások tiltása''
}
```

A másik lehetőség az, hogy a `MODELVIEW` transzformációt nem bántjuk, hanem a plakátnégyzetet előtranszformáljuk a kameratranszformáció forgatási részének inverzével. A plakát síkjába eső $[1,0,0]$ és $[0,1,0]$ egységvektorokat az inverz forgatási mátrixszal előtranszformálva a `right` és `up` vektorokhoz jutunk, amelyekből a plakát sarokpontjai már előállíthatók:

```
//-----------------------------------------------------------------
void BillBoard::DrawIt(Camera * camera) {
//-----------------------------------------------------------------
   float viewmatx[4][4];  // MODELVIEW mátrix lekérdezés
   glGetFloatv(GL_MODELVIEW_MATRIX, &viewmatx[0][0]);
   Vector right(viewmatx[0][0], viewmatx[1][0], viewmatx[2][0]);// (1,0,0)-ből
   Vector up(viewmatx[0][1], viewmatx[1][1], viewmatx[2][1]);   // (0,1,0)-ből

   ,,átlátszósági beállítások engedélyezése''
   glBegin(GL_QUADS);
   glTexCoord2f(0, 0); glVertex3fv((position - (right+up) * size).GetArray());
   glTexCoord2f(1, 0); glVertex3fv((position + (right-up) * size).GetArray());
   glTexCoord2f(1, 1); glVertex3fv((position + (right+up) * size).GetArray());
   glTexCoord2f(0, 1); glVertex3fv((position + (up-right) * size).GetArray());
   glEnd();
   ,,átlátszósági beállítások tiltása''
}
```

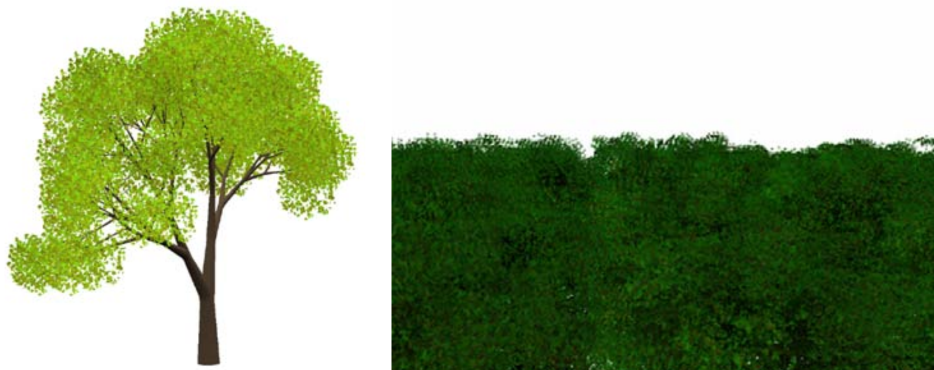
Ebben a megoldásban a plakát sarokpontjait tömbökként adtuk át, és az x,y,z koordinátákat tartalmazó tömb kezdőcímét a `Vector` osztály `GetArray()` tagfüggvényével kérdeztük le.

A plakátok képeinek elkészítésénél tekintetbe kell vennünk, hogy a természeti jelenségek (tűz, fák stb.) nem téglalap alakúak, hanem a határuk szabálytalan. Ez nem jelent nehézséget akkor, ha a képen átlátszó színeket is használhatunk, és a megjelenítendő jelenségen kívüli pontokat a képben átlátszó színnel töltjük fel. Az átlátszó színek kezelésének rejtelmeit a 7. fejezetben tárgyaltuk.

A plakátokban általában csak teljesen átlátszatlan ($A = 1$) és teljesen átlátszó ($A = 0$) színek találhatók, közbenső eset nincs. Azt szeretnénk, hogy ahol a szín átlátszó, ott a rasztertár tartalmat a plakát ne változtassa meg, ahol pedig átlátszatlan, ott írja felül függetlenül az eredeti értékétől. A 7.1. táblázat lehetőségeit áttanulmányozva több megoldás is adódik, például használhatjuk a következő beállítást[3]:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Ha úgy gondoljuk, hogy ezzel készen is volnánk, nagyon tévedünk. A pixelek ugyanis nem csak a rasztertáron keresztül hatnak egymásra, hanem a z-bufferen keresztül is. Amikor a plakátot raszterizáljuk a pixeleihez tartozó z-értékeket az OpenGL összehasonlítja a z-bufferben tárolt értékekkel. Ha az újabb z-érték kisebb, akkor egyrészt felülírja a z-buffert, másrészt pedig a tárolt R, G, B komponenseket a fent ismertetett súlyozás szerint megváltoztatja. Ha tehát egy teljesen átlátszó ($A = 0$) pixelt dolgozunk fel, az ugyan nem változtatja meg a színbuffert, de a z-buffer tartalmát elronthatja. Egy később beírt, az átlátszó rész mögötti pixel feldolgozását a z-buffer meg fogja tagadni, tehát az átlátszóság mégsem érvényesül teljes mértékben.



10.4. ábra. *Fa és erdő pixel-árnyalóval javított plakátokkal [115]*

[3] ez a beállítás a rajzolási színt a saját alfa értékével, a rasztertár tartalmát pedig a rajzolási szín alfa értékének a komplemensével szorozza, és a két részeredményt összeadja

Szerencsére két megoldásunk is van erre a problémára. Az első rendkívül egyszerű, de csak abban az esetben használható, ha a képek csak teljesen átlátszó és teljesen átlátszatlan pixeleket tartalmaznak. Az OpenGL ugyanis megengedi, hogy az egyes pixelek rajzolását az alfa érték alapján még a z-bufferbe írás előtt visszautasítsuk. Az eljárást *alfa-teszt*nek nevezik és az engedélyezése után azt kell megadni, hogy milyen feltételt teljesítő alfájú pixelek mehetnek át ezen a teszten. A következő utasítások a nem zérus alfájú pixeleket engedik át, a zérus alfájú pixeleket pedig kiszűrik:

```
glEnable(GL_ALPHA_TEST);    // alfa teszt engedélyezése
glAlphaFunc(GL_GREATER, 0); // csak a 0-nál nagyobb alfájú pixeleket írjuk
```

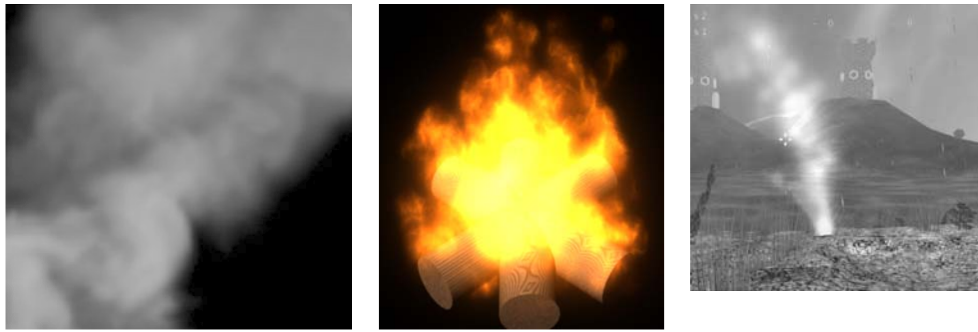
A bonyolultabb megoldás féligáteresztő felületekkel is megbirkózik. Az átlátszóságnál az okozza a gondot, hogy a színek súlyozott átlagát mindig az átlátszó felületre és a mögötte lévő tárgy képére kell képezni, azaz a felületek raszterizálási sorrendje nem marad tetszőleges. A felületeket a szemtől való távolságuk alapján rendezni kell, majd a legtávolabbi felülettől kezdve a közelebbiek felé haladva kell elvégezni a raszterizációt. Ekkor a z-buffert akár ki is kapcsolhatjuk, hiszen a takarási feladatot a rendezéssel oldjuk meg. Bonyolult terekben a rendezés nehéz lehet, és a z-buffer azon előnyéről sem szívesen mondunk le, hogy a háromszögeket tetszőleges sorrendben képes feldolgozni. A z-buffer módszert csak részben csempészhetjük vissza, mégpedig a nem átlátszó felületek rajzolásához. Az átlátszókat továbbra is a sorrendezés után kell felrajzolni. Egy ilyen módszerrel a következő fejezetben fogunk megismerkedni.

A plakátokon nem csak állóképeket, hanem mozgást is megjeleníthetünk (*sprite*), ha egy plakáthoz nem egyetlen képet, hanem egy teljes képsorozatot rendelünk. Az egyes képkockákhoz idősávok tartoznak, és minden egyes felrajzolásakor az utolsó rajzolás óta eltelt idő alapján dönthetjük el, hogy melyik következő képkockát használjuk textúraként. Ezzel a módszerrel animált lángot, robbanást, füstöt stb. hozhatunk létre.

10.2.7. Részecskerendszerek: a ParticleSystem osztály

A plakátokat úgy animáltuk, hogy a hozzájuk rendelt képeket az időben változtattuk. További lehetőség, ha a plakát négyszög helyét és méretét is változtatjuk az időben, ami egy új fogalomhoz, a *részecskerendszer*ekhez (*particle system*) vezet el.

A részecskerendszerek alapötlete az a felismerés, hogy sok, a természetben előforduló dinamikus jelenség (hóesés, tűz, robbanás, vízesés, füst) a rendszerben kavargó kicsiny elemeknek, úgynevezett részecskéknek köszönheti a változó alakját, tehát ezen jelenségek szimulációjánál is a részecskéket kell követni. A részecskékre hasonló törvények vonatkoznak, viszont eltérő tulajdonságaik lehetnek és önálló életet élnek. Például egy füst részecskére ható felhajtó erő felfelé mutat, de tömege, sebessége, színe, pillanatnyi helye stb. már részecskéről-részecskére különböző, és időben változó is lehet. Ahhoz, hogy ezekből a füstrészecskékből hihető, gomolygó füst alakuljon ki,



10.5. ábra. *Részecskerendszerek és alkalmazásuk egy játékban [15]*

és ne legyen a néző számára rögtön nyilvánvaló, hogy különálló elemeket rajzolunk, nagyon sok, akár több ezer részecskével kell dolgoznunk. A részecskék nagy száma indokolja azt az egyszerűsítő feltételezést, hogy a részecskék nem hatnak egymás viselkedésére, minden részecske a saját tulajdonságai és a teljes rendszerre vonatkozó törvények alapján önállóan mozog[4]. A részecskék állapotát általában a következőkkel jellemezzük:

- Hely (`position`): a részecske helye a háromdimenziós térben.
- Sebesség (`velocity`): a részecske sebessége, amely az eltelt idővel szorozva a helyváltozást fejezi ki.
- Gyorsulás (`acceleration`): a részecske gyorsulása, amely az eltelt idővel szorozva a sebességváltozást adja meg.
- Tömeg (`weight`): a részecske „tömege” kifejezi a rendszerben működő erőtér (a füstöt a felhajtóerő felfelé, a vízesést a nehézségi erő lefelé húzza) és az adott részecske gyorsulása közötti arányt (a dinamika alaptörvénye szerint a gyorsulás az erő és a tömeg hányadosa, amit most elvontabban is értelmezhetünk úgy, hogy a tömeg azt mutatja meg, hogy a részecske milyen mértékben áll ellen az erőtér hatásainak).
- Tömegváltozás (`dweight`): a részecske „tömegének” változási sebessége. A füstrészecskék például folyamatosan hűlnek, ezért a felhajtóerő csökken. Ezt úgy modellezhetjük, hogy minden részecskére ugyanaz az erő hat, de a részecskék öregedésével a tömegük nő, így egyre kevésbé képesek az erő hatására gyorsulni.

[4] n elem között $n(n-1)/2$ kapcsolat lehetséges, ami nagyon nagy lenne, ha már n is nagy

- Hátralévő élettartam (`time_to_live`): a részecske haláláig hátralévő idő. Amikor a részecske meghal, akkor eltűnik. Ezzel a módszerrel szimulálhatjuk azt a jelenséget, hogy a füst, tűz stb. részecskék egy idő után elenyésznek. A tűz kialszik, ha nem szítjuk folyamatosan új részecskékkel.
- Szín (`color`): a részecske színe és átlátszósága az (R,G,B,A) csatornákon.
- Színváltozás (`dcolor`): a részecske színének változási sebessége. Egy lángrészecske például a születésekor, a láng magjában még fehér, de ahogy öregszik és a lángnyelv széle felé halad, egyre vörösebb és átlátszóbb lesz.
- Méret (`size`): a részecske rajzolásakor használt szám, például a plakát mérete, amely a kameratávolsággal együtt meghatározza, hogy a részecskét hány pixelen jelenítsük meg.
- Méretváltozás (`dsize`): a méret változási sebessége. Egy robbanás például egyetlen pontból indul, ahonnan kicsiny részecskék törnek ki, amelyek a későbbiekben egyre nagyobbak lesznek.

Egy részecske megjeleníthető *pont*, *vonal*, vagy a plakátokhoz hasonlóan, *textúrázott téglalap* rajzolásával. A textúrázott téglalapnak van a legnagyobb jelentősége, hiszen ez még közelről nézve is kellemes hatást nyújt. A textúra szinte mindig tartalmaz átlátszó képelemeket is, ugyanis a részecskék a legritkább esetben téglalapok, ráadásul maguk is átlátszóak (a lángon keresztül láthatjuk a többi lángot, vagy akár a hátteret is).

A következőkben egy részecske osztályt mutatunk be. A teljes részecskerendszer sok részecskeobjektumot tartalmazhat, amelyek dinamikusan születnek meg, majd tűnnek el a haláluk után. A dinamikus adatszerkezetet egy láncolt listával implementálhatjuk, amelyet a részecske `next` adattagja kapcsol össze. Az `AnimateIt` tagfüggvény az eltelt idő alapján a részecskeállapotot frissíti, az `Animate` pedig ezt a műveletet rekurzív módon a teljes listára végrehajtatja. Az `Animate` a részecskehullák tetemei által lefoglalt helyet is felszabadítja. A `DrawIt` a plakátoknál megismert módon egy textúrázott négyszöget rajzol fel úgy, hogy a négyszög normálvektora mindig a szem felé nézzen. A `Draw` eljárás pedig rekurzívan a lista összes elemét felrajzoltatja.

```
//===============================================================
class Particle {  // egyetlen részecskét leíró osztály
//===============================================================
public:
   Vector  position, velocity, acceleration; // pozíció, sebesség, gyorsulás
   float   weight, dweight;        // tömeg és változása: gyorsulás=erő/tömeg
   float   time_to_live;           // hátralevő élettartam
   Color   color, dcolor;          // szín és változása (derivált)
   float   size,  dsize;           // rajzoláskor használt méret és változása
   Particle * next;     // láncoló mutató
```

```
   Particle * Animate(float dt, Vector& force) { // láncolt lista animálása
      Particle * new_next = next;
      if (next) new_next = next->Animate(dt, force);
      if (time_to_live > 0) {
         AnimateIt(dt, force);              // ezen részecske animálása
         next = new_next; return this;      // ,,next" = a kövekező élő
      } else {                              // ha meghalt, töröljük
         delete this; return new_next;      // a kövekező élő
      }
   }

   void AnimateIt(float dt, Vector& force) { // egyetlen részecske animálása
      time_to_live -= dt;
      acceleration  = force / weight;
      velocity     += acceleration * dt;
      position     += velocity * dt;
      weight       += dweight * dt;
      size         += dsize * dt;
      color        += dcolor * dt;
   }

   void Draw(Vector& right, Vector& up) {     // lista rekurzív rajzolása
      DrawIt(right, up);                      // ezen részecske rajzolása
      if (next) next->Draw(right, up);        // többi részecske rajzolása
   }
   void DrawIt(Vector& right, Vector& up) {   // egyetlen részecske rajzolása
      if (time_to_live < 0) return;
      glColor4f( color.r, color.g, color.b, color.a ); // szorzószín
      glTexCoord2f(0, 0); glVertex3fv((position-(right+up)*size).GetArray());
      glTexCoord2f(1, 0); glVertex3fv((position+(right-up)*size).GetArray());
      glTexCoord2f(1, 1); glVertex3fv((position+(right+up)*size).GetArray());
      glTexCoord2f(0, 1); glVertex3fv((position+(up-right)*size).GetArray());
   }
};
```

A részecskék gyűjteményét, a *részecskerendszer*t a következő osztállyal valósíthatjuk meg:

```
//=================================================================
class ParticleSystem : public TexturedObject {
//=================================================================
protected:
   Particle *  particles; // a részecskék láncolt listája
   float       age;       // a rendszer kora
   Vector      force;     // erő (gravitáció, szél stb.)
public:
   ParticleSystem(char * texture_filename) : Texture(texture_filename) {
      particles = NULL; age = 0.0;
   }
   ~ParticleSystem() { if (particles) delete particles; }
   virtual void Emit(int n) {}
   void  AnimateIt(float dt) {
      age += dt;
      if (particles) particles = particles->Animate(dt, force);
   }
   void  DrawIt(Camera * camera) {
```

```
      float viewmatx[4][4];     // MODELVIEW mátrix lekérdezés
      glGetFloatv(GL_MODELVIEW_MATRIX, &viewmatx[0][0]);
      Vector right(viewmatx[0][0], viewmatx[1][0], viewmatx[2][0]);// (1,0,0)
      Vector up(viewmatx[0][1], viewmatx[1][1], viewmatx[2][1]);   // (0,1,0)
                                // szín = részecskeszín * textúraszín
      glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
      glDepthMask(GL_FALSE);    // z-buffer átírásának tiltása
      glEnable(GL_BLEND);       // átlátszóság engedélyezés
      glBlendFunc(GL_SRC_ALPHA, GL_ONE); // átlátszósággal súlyozva összegzünk
      glBegin(GL_QUADS);
      if (particles) particles->Draw(right, up); // részecskék egyenként
      glEnd();
      glDisable(GL_BLEND);
      glDepthMask(GL_TRUE);     // z-buffer ismét írható
   }
};
```

A részecskerendszer tartalmazza a részecskék láncolt listáját (`particles`), a rendszer korát (`age`) és a részecskékre ható erőteret (`force`). Az `Emit` függvény bocsátja útjára a láncolt listára fűzött részecskéket. A részecskék típusa már a modellezett jelenségtől függ, így az általános részecskerendszerben ez csupán egy üres törzsű virtuális függvény. A rendszer animálása a láncolt lista elemeinek animálását, esetlegesen az erőtér megváltoztatását jelenti (például a szél okozta erőtér változó). A részecskerendszer animálása (`AnimateIt`) a láncolt listára fűzött részecskék egyenkénti, független mozgatását jelenti.

A `DrawIt` függvény az átlátszó plakátok rajzolásához szükséges előkészületeket végzi el, majd egyenként felrajzolja a részecskék textúrázott négyszögeit, végül visszaállítja az átlátszóság miatt elvégzett változtatásokat. A `DrawIt` függvény először a plakátoknak a nézeti irányra merőleges oldalvektorait számítja ki, majd olyan textúrarajzolást állít be, amelyben a rajzolási szín a textúrák színének és a `glColor()` függvénnyel beállított színnek a szorzata lesz (`GL_MODULATE`). Ezzel egyetlen textúrával különböző színű részecskéket is megjeleníthetünk. A következő lépésben felkészülünk az átlátszó elemek megjelenítésére. Ehhez a számított és a már rasztertárban tárolt színek alfacsatorna szerinti átlagolását kell engedélyezni.

Az összemosó (*blending*) függvény kiválasztásánál két szempontot célszerű érvényesíteni (7.1. táblázat). Egyrészt szeretnénk a részecskéket a láncolt listában felvett sorrendjükben felrajzolni, azaz el kívánjuk kerülni a távolság szerinti rendezést, amire pedig az átlátszóságnál általában szükség van. Amennyiben olyan összemosó függvényt választunk, amely a forrás és cél változókban szimmetrikus, akkor az összemosás operandusai felcserélhetők, így a rajzolást tetszőleges sorrendben elvégezhetjük. A másik szempont az, hogy a kép hűen adja vissza a szimulált természeti jelenséget. Például tűz, robbanás stb. esetén a fényjelenségek hozzáadódnak a háttér képéhez. Mindkét szempontot kielégíthetjük a `glBlendFunc(GL_SRC_ALPHA, GL_ONE)` beállítással, amely a forrást, azaz a részecskét a saját átlátszóságával súlyozza és a súlyozott összeget a rasztertárban képzi. Ezzel látszólag megúsztuk a részecskék rendezését, de

nem szabad elfeledkezni a z-bufferről sem. A *z-buffer* ugyanis a feldolgozási sorrend és a szemtől való távolság szerint megtagadhatja az egyes részecskék felrajzolását. Ha kikapcsoljuk a z-buffer működését, akkor az összes részecskét felrajzoljuk (ami helyes), még azokat is, amelyek valamely nem átlátszó és nem a részecskerendszerhez tartozó tárgy mögött tűnnének fel (ami viszont helytelen). Ezt a problémát úgy oldhatjuk meg, hogy a z-buffer ellenőrzési és engedélyezési funkcióját továbbra is használjuk, viszont letiltjuk a z-buffer átírását. Első menetben a normális, nem átlátszó tárgyakat rajzoljuk fel, a szokásos z-buffer üzemmódban. A menet végén a nem átlátszó tárgyak képe a rasztertárban, a távolságuk pedig a z-bufferben lesz. A második menetben az átlátszó tárgyak, például a részecske rendszerek következnek, amely előtt egy `glDepthMask(GL_FALSE)` hívással a z-buffer átírását letiltjuk, de a rajzolás megtagadására továbbra is igényt tartunk, amikor tárolt z-érték az új z-értéknél kisebb. Amikor a második menetben az OpenGL egy részecskét rajzol, a z-bufferrel eldönti, hogy valamely nem átlátszó tárgy takarja-e, és ha igen, akkor a rajzolást megtagadja. Ha a részecskét egy átlátszatlan tárgy nem takarja, akkor a részecske képét a rasztertárba visszük, de a z-buffer tartalmat nem változtatjuk meg, így nem fordulhat elő, hogy egy részecskét a másik takarása miatt ne rajzolnánk fel.

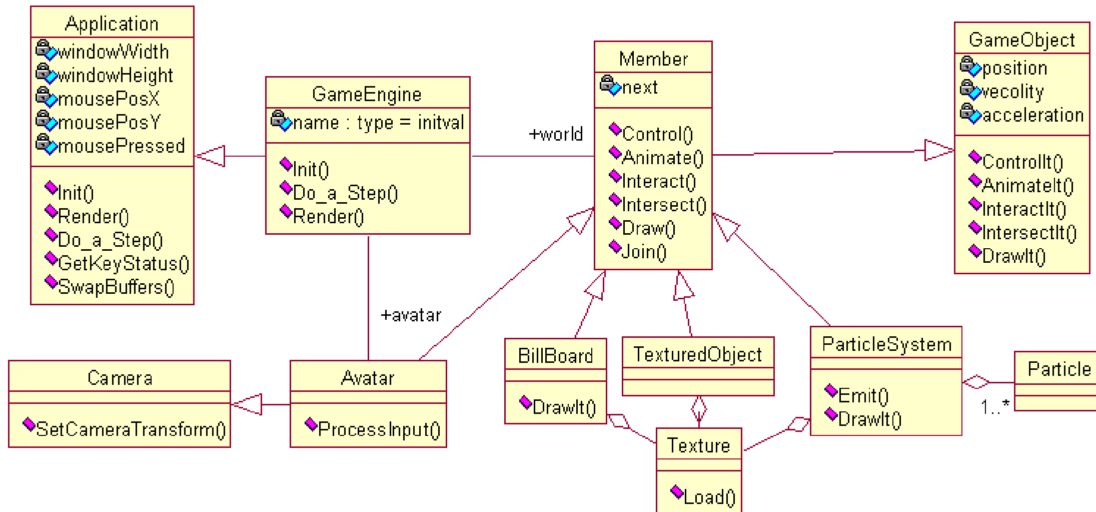
Idáig egy általános részecske osztályt és részecskerendszer osztályt ismertettünk. Az egyes jelenségekhez ezekből az osztályokból kell alosztályokat származtatni, amelyek értelmet adnak a részecskerendszer `Emit()` tagfüggvényének és feltöltik az egyes részecskék paramétereit.

10.2.8. A játékmotor osztály

A játékok szereplőit a *játékmotor* (`GameEngine`) működteti. A játékmotor a program fő vezérlő és megjelenítő osztálya:

```
//===============================================================
class GameEngine : public Application {  // játékmotor
//===============================================================
protected:
   Member * world;    // a szereplőket tartalmazó láncolt lista kezdete
   Avatar * avatar;   // a játékost megszemélyesítő objektum
public:
   GameEngine(char * caption, int width, int height)
       : Application(caption, width, height) {}
   void Init() {
      glEnable(GL_DEPTH_TEST);           // z-buffer bekapcsolása
   }

   void Do_a_Step(float dt) {            // játékhurok egyetlen ciklusa
      avatar->ProcessInput(this);        // avatárt a felhasználó vezérli
      world->Animate(dt);                // világ objektumainak animálása
      world->Control(dt);                // világ objektumainak vezérlése
      world->BuryDead(avatar);           // halottak helyének felszabadítása
      Render();                          // képszintézis
```

```
    }

    void Render() {                          // képszintézis
       glClearColor(0, 0, 0, 0);             // képernyő törlés
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
       avatar->SetCameraTransform(this);  // avatár szemével látunk
       world->Draw(avatar);                  // világ objektumainak rajzolása
       SwapBuffers();                        // buffercsere
    }
};
```



10.6. ábra. *A játékobjektumok általános keretrendszerének osztálydiagramja*

A `GameEngine` osztály felüldefiniálja az alkalmazás `Render()` és `Do_a_Step()` tagfüggvényeit. A `Render()` az ablak tartalmát újrarajzolja, a `Do_a_Step()` tagfüggvényt pedig az üresjárati esemény aktiválja. A játékmotor szíve a játékhurok, amelynek egyetlen ciklusát a `Do_a_Step()` függvénybe tettük. A függvény folyamatos hívogatásáért az ablakozó rendszer felelős. A létrehozott hierarchiát bemutató osztálydiagramot a 10.6. ábrán láthatjuk. A következő fejezetekben egy egyszerű játékot építünk fel, majd javítunk egy kicsit a játékmotorunkon, és egy újabb játékot valósítunk meg.

10.3. Az űrharc játék

A következőkben a játékprogramozás alapelveit egy űrhajós játék elkészítésével szemléltetjük. A játék a mi naprendszerünkben zajlik, ahol a bolygók a Nap körül keringenek, a távolban pedig az állócsillagok képe sejlik fel. Ebben a környezetben repülünk

űrhajónkkal, néhány hasonló felépítésű ellenséges űrhajó társaságában. Az űrhajók mozgását a bolygók gravitációs ereje és a hajtóműveik tolóereje határozza meg. Az űrhajók az orruk irányába fotonrakétákat indíthatnak, amelyek az eltalált űrhajókat robbanás kíséretében semmisítik meg. Az űrhajók a bolygókkal ütközve ugyancsak felrobbannak. A játékos célja, hogy a saját űrhajójával elkerülje a megsemmisülést, miközben az ellenfeleket leteríti. Az ellenfelek célja nem kevésbé nemes, ők az avatárt szeretnék lepuffantani.

Az űrharc játék objektumai tehát az űr, az égitestek, illetve bolygók (Nap, Merkur, Vénusz, Föld, Mars, Jupiter, Szaturnusz, Uránusz, Neptunusz, Plútó), az ellenséges űrhajók, a játékos űrhajója, fotonrakéták és robbanások. A játékobjektumok különböző típusokhoz sorolhatók, az űrt a `Space`, a játékost a `Self`, a fotonrakétákat a `PhotonRocket`, a bolygókat a `Planet`, az űrhajókat pedig a `Ship` osztály segítségével hozzuk létre. Az osztályokat a játékmotor osztályaiból származtatjuk, mindig abból, amely az új osztály viselkedéséből a lehető legtöbbet tartalmazza. Szükség esetén a származtatott osztályokban az örökölt rajzoló-, animáló- és vezérlőfüggvényeket felüldefiniáljuk.

10.7. ábra. *Az űrharc játék egy pillanatfelvétele*

Az osztályok mindegyike megvalósítja a `GetType()` függvényt, amely az osztály típusával tér vissza, így a játékobjektumok ez alapján ismerhetik fel egymást. A lehetséges változatokat egy felsorolástípussal adjuk meg:

```
enum GameObjectType {   // az űrharc játék objektumtípusai
    SPACE,              // űr
    PLANET,             // bolygó
    SHIP,               // űrhajó
    AVATAR,             // avatár
```

```
    EXPLOSION,          // robbanás
    PHOTON_ROCKET       // fotonlövedék
};
```

A következőkben ezeket az osztályokat egyenként vizsgáljuk.

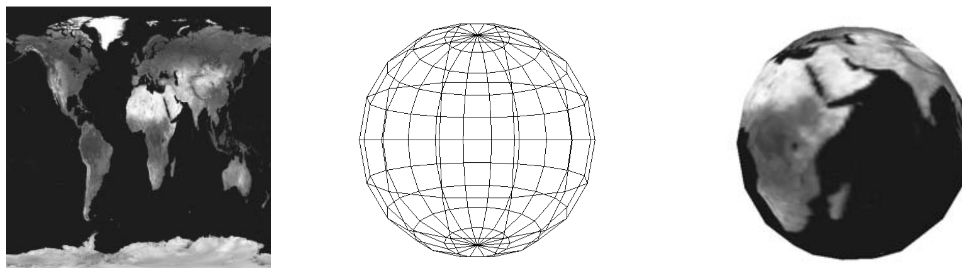
10.3.1. A bolygók

Egy bolygó geometriai értelemben egy gömb, amelyet az űrfelvételekből nyert textúrákkal díszíthetünk. A Naprendszer négy óriásbolygójánál gyűrűket is megfigyelhetünk, amelyet egy kivágott, átlátszó körlemezzel jelenítünk meg.

A játékmotor objektumtípusai között keresgélve a `TexturedObject` tűnik megfelelőnek, amiből a bolygók `Planet` nevű osztálya származtatható:

```
//=================================================================
class Planet : public TexturedObject { // a bolygó egy textúrázott objektum
//=================================================================
   GLUquadricObj *sphere, *disk;// kvadratikus felületek
   float  mass;                 // tömeg
   float  rot_angle, rot_speed; // forgási szög és sebesség
   float  rev_angle, rev_speed; // keringési szög és sebesség
   float  dist;                 // bolygó-nap távolság
   float  axis_angle;           // tengelyferdülés
   float  r1, r2;               // gyűrű külső-belső sugarak
public:
   Planet(Vector& pos0, float R, char * filename, float axis_angle0 = 0,
          float r10 = 0, float r20 = 0) : TexturedObject(pos0, filename) {
      bounding_radius = R;      // a bolygó sugara = a befoglaló gömb sugara
      mass = pow(R, 3);         // a tömeg arányos a térfogattal
      axis_angle = axis_angle0; // a tengely dőlésszög
      r1 = r10; r2 = r20;       // gyűrű
      dist = pos0.Length();     // bolygó-nap távolság
      rot_angle = rev_angle = 0;// forgási és keringési szögek
                 // keringési sebesség az égi mechanika törvényei szerint
      rev_speed = (dist > EPSILON) ? 100 / pow(dist, 1.5) : 0;
      rot_speed = 10;           // forgási sebesség

      sphere = gluNewQuadric(); // GLU textúrázott gömb létrehozása
      gluQuadricTexture(sphere, GL_TRUE); // textúrázott gömb
      if (r1 > 0 && r2 > 0)
         disk = gluNewQuadric();// GLU átlátszó gyűrű létrehozása
   }
   int GetType() { return PLANET; }
   float Radius() { return bounding_radius; }  // sugár
   float Mass() { return mass; }               // tömeg
   void AnimateIt(float dt);
   void DrawIt(Camera * camera);
};
```

A textúrakezelést tehát az alaposztálytól örököljük, most csak a geometriával kell megbirkóznunk. Az OpenGL a gömb háromszög vagy négyszög hálóval történő közelítését fogadja el, amelyet a tesszelláció elvégzése után kaphatunk meg.



10.8. ábra. *A Föld textúrája, geometriája és képe*

Szerencsére az OpenGL kiegészítő könyvtára (*glu*) tartalmaz olyan függvényeket, amelyek *másodrendű felületek*re (*quadrics*), így a *gömb*re és kivágott *körlemez*re is átvállalják a tesszelláció és textúra koordináták számításának feladatait. Egy textúrázott másodrendű felület létrehozásához a következő két sorra van szükség:

```
sphere = gluNewQuadric();           // kvadratikus felület azonosító
gluQuadricTexture(sphere, GL_TRUE); // a felületet textúrázni kell
```

A `gluNewQuadric()` egy táblázatot hoz létre, és annak azonosítójával tér vissza. Minden további műveletben ezzel az azonosítóval hivatkozhatunk a másodrendű felületre. Például, ha a másodrendű felület paramétereit a gömbnek megfelelően kívánjuk beállítani, amit szeretnénk rögtön tesszellálni és az OpenGL segítségével felrajzoltatni, akkor a `gluSphere(quadric,radius,hor,vert)` függvényt alkalmazhatjuk. Ennek a függvénynek az azonosítón kívül a gömb sugarát, és a függőleges illetve a vízszintes tesszelláció finomságát (a hosszúsági és a szélességi körök számát) kell átadni. Az esetleges gyűrűt egy körlemez `gluDisk(disk,r1,r2,hor,vert)` paranccsal jeleníthetjük meg. Az azonosítót követő paraméterek a lemez belső és külső sugara, valamint a tesszellációs pontok száma a kör és a sugár mentén.

Egy bolygó nem „gondolkozik”, hanem az égi mechanika törvényeit szolgai módon követi. A fizikusoknak Einstein általános relativitáselmélete juthatna eszébe, mi azonban beérjük a kopernikuszi világképpel és a Newton-féle gravitációs törvénnyel is, ugyanis ezek jóval egyszerűbbek, a valóságot pedig a játékhoz elegendően pontosan írják le.

A bolygók tengelyük körül forognak, miközben a Nap körül keringenek. A tengely körüli forgás sebessége állandó. A Nap körüli keringésből származik a bolygó mozgása, azaz pozíciójának változása. A bolygót két különböző eljárással is mozgathatjuk, fizikai animációval (9.10. fejezet) és képletanimációval (9.7. fejezet).

A *fizikai animáció* Newton gravitációs és mozgástörvényeit használja. *Newton gravitációs törvénye* kimondja, hogy az $\vec{r}_1$ és $\vec{r}_2$ pontokban lévő m_1 és m_2 tömegű testek

között vonzóerő ébred, amelynek nagysága egyenesen arányos a testek tömegével és fordítottan arányos a távolságuk négyzetével:

$$|\vec{F}| = f \cdot \frac{m_1 \cdot m_2}{|\vec{r}_2 - \vec{r}_1|^2},$$

ahol $f = 6.67 \cdot 10^{-11}[\frac{m^3}{kg \cdot s^2}]$ a Newton-féle tömegvonzási együttható. A vonzóerő a másik égitest felé mutat, tehát az első égitestre ható erő vektoriális formában:

$$\vec{F}_1 = f \cdot \frac{m_1 \cdot m_2}{|\vec{r}_2 - \vec{r}_1|^3} \cdot (\vec{r}_2 - \vec{r}_1).$$

Ezen összefüggés alapján egy bolygóra az összes többi bolygó eredő vonzóereje meghatározható. A bolygó gyorsulása az eredő erő és a tömeg hányadosa, amiből az utolsó játékciklus óta bekövetkezett sebességváltozás és pozícióváltozás becsülhető.

A *képletanimáció* alkalmazásához azzal a feltételezéssel élünk, hogy a bolygók a Nap körül körpályán keringenek, állandó szögsebességgel (ez a sebesség a Föld esetében 360 fok/év lenne, a játékban viszont érdemes nagyobb sebességeket használni). A keretidő ismeretében a keringési szög új értéke a régi értékből számítható. A keringési szög és a Nap–bolygó távolság alapján pedig a pozíció előállítható.

A fizikai animációt és a képletanimációt összehasonlítva megállapíthatjuk, hogy a fizikai animáció lényegesen általánosabb, és nem csak a Nap vonzását, hanem a bolygók egymásra hatását is figyelembe veszi. A képletanimáció viszont egyszerű és robusztus. A fizikai animáció a pozíció- és sebességváltozást csupán becsüli, mégpedig annál pontatlanabbul, minél nagyobb a keretidő. A számítási hibák halmozódása miatt a bolygónk lassan le is térhet a körpályáról, sőt ki is sodródhat a Naprendszerből. A képletanimáció azonban sohasem vezet ilyen dezertáló bolygókhoz.

A képletanimáció alkalmazásakor az `AnimateIt()` függvény kiszámítja a mozgás két állapotváltozójának, a forgási és a keringési szögnek a pillanatnyi értékét, majd a Nap–bolygó távolság alapján meghatározza a bolygó pillanatnyi helyét:

```
//-----------------------------------------------------------------
void Planet::AnimateIt(float dt) {
//-----------------------------------------------------------------
   rot_angle += rot_speed * dt;   // saját tengely körüli forgási szög
   if (rot_angle > 360) rot_angle -= 360;
   rev_angle += rev_speed * dt;   // keringési szög
   if (rev_angle > 360) rev_angle -= 360;
   position.x = dist * cos(rev_angle * M_PI/180); // az origó körül kering
   position.y = dist * sin(rev_angle * M_PI/180);
   position.z = 0;
}
```

A `DrawIt()` függvény az állapotváltozók alapján elhelyezi a bolygót, a döntött tengely körül elforgatja, és fel is rajzolja azt:

```
//-----------------------------------------------------------------
void Planet::DrawIt(Camera * camera) {
//-----------------------------------------------------------------
   glTranslatef(position.x, position.y, position.z);  // középpont helye
   glRotatef(axis_angle, 1, 0, 0);              // a dőlt forgástengely
   glRotatef(rot_angle, 0, 0, 1);               // saját tengely körüli forgatás
   gluSphere(sphere, bounding_radius, 16, 10);// 16x10 részre tesszellált gömb
   if (r1 > 0 && r2 > 0) {                      // van gyűrű ?
      glEnable(GL_BLEND);                       // a gyűrű átlátszó
      glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
      glColor4f(1, 0.8, 0.6, 0.5);              // a gyűrű színe
      gluDisk(disk, r1, r2, 32, 2);      // r1, r2 sugarú körlemez 32x2 db-ból
      glDisable(GL_BLEND);
   }
}
```

Befejezésképpen megjegyezzük, hogy a Kepler törvények értelmében a bolygók nem is kör, hanem ellipszis pályán keringenek, de ezzel egy játékban már igazán nem érdemes vesződni.

10.3.2. Az űr

Az űr az állócsillagok háttérben felsejlő fényeit mutatja. Az űr az időben változatlan, a többi játékobjektumra nem hat, tehát csak a rajzolási műveletének van tényleges tartalma. Az állócsillagok képét egy 2D rajzolóprogram segítségével készíthetjük el, amit textúraként jeleníthetünk meg. A textúrához egy nagy kockát választunk, amelynek oldalfalaira rátapétázzuk a csillagok megrajzolt képét (a kocka helyett használhatnánk hengert vagy gömböt is). Ezzel rögtön korlátoztuk is a játék terét a kocka belsejére, és tulajdonképpen csaltunk, hiszen a valóságban az állócsillagok nem egy óriási kocka belső lapjain helyezkednek el. A csalás nyilvánvalóvá válik, ha a kocka lapjaihoz közel kerülünk, vagy esetleg kirepülünk a kocka belsejéből. Amíg viszont a nagy méretű kocka közepén mozgunk, ez a közelítés maradéktalanul elfogadható.

```
//=================================================================
 class Space : public TexturedObject {  // az űr egy textúrázott játékobjektum
//=================================================================
 public:
     Space(char * filename) : TexturedObject(Vector(0, 0, 0), filename) { }
     int GetType() { return SPACE; }
     void DrawIt(Camera * camera);
};
```

Az űr rajzolása tehát egy `SPACE_SIZE` oldalú kocka hat textúrázott lapjának a lefényképezését jelenti:

```
//-----------------------------------------------------------------
void Space::DrawIt(Camera * camera) {
//-----------------------------------------------------------------
   glBegin(GL_QUADS);
```

```
        // -z sík
      glTexCoord2i(0, 0); glVertex3i(-SPACE_SIZE, -SPACE_SIZE, -SPACE_SIZE);
      glTexCoord2i(0, 1); glVertex3i(-SPACE_SIZE,  SPACE_SIZE, -SPACE_SIZE);
      glTexCoord2i(1, 1); glVertex3i( SPACE_SIZE,  SPACE_SIZE, -SPACE_SIZE);
      glTexCoord2i(1, 0); glVertex3i( SPACE_SIZE, -SPACE_SIZE, -SPACE_SIZE);
        // z sík
      glTexCoord2i(0, 0); glVertex3i(-SPACE_SIZE, -SPACE_SIZE, SPACE_SIZE);
      glTexCoord2i(0, 1); glVertex3i( SPACE_SIZE, -SPACE_SIZE, SPACE_SIZE);
      glTexCoord2i(1, 1); glVertex3i( SPACE_SIZE,  SPACE_SIZE, SPACE_SIZE);
      glTexCoord2i(1, 0); glVertex3i(-SPACE_SIZE,  SPACE_SIZE, SPACE_SIZE);
        // x sík
      glTexCoord2i(0, 0); glVertex3i( SPACE_SIZE, -SPACE_SIZE, -SPACE_SIZE);
      glTexCoord2i(0, 1); glVertex3i( SPACE_SIZE, -SPACE_SIZE,  SPACE_SIZE);
      glTexCoord2i(1, 1); glVertex3i( SPACE_SIZE,  SPACE_SIZE,  SPACE_SIZE);
      glTexCoord2i(1, 0); glVertex3i( SPACE_SIZE,  SPACE_SIZE, -SPACE_SIZE);
        // -x sík
      glTexCoord2i(0, 0); glVertex3i(-SPACE_SIZE, -SPACE_SIZE,  SPACE_SIZE);
      glTexCoord2i(0, 1); glVertex3i(-SPACE_SIZE, -SPACE_SIZE, -SPACE_SIZE);
      glTexCoord2i(1, 1); glVertex3i(-SPACE_SIZE,  SPACE_SIZE, -SPACE_SIZE);
      glTexCoord2i(1, 0); glVertex3i(-SPACE_SIZE,  SPACE_SIZE,  SPACE_SIZE);


        // y sík
      glTexCoord2i(0, 0); glVertex3i(-SPACE_SIZE,  SPACE_SIZE, -SPACE_SIZE);
      glTexCoord2i(0, 1); glVertex3i( SPACE_SIZE,  SPACE_SIZE, -SPACE_SIZE);
      glTexCoord2i(1, 1); glVertex3i( SPACE_SIZE,  SPACE_SIZE,  SPACE_SIZE);
      glTexCoord2i(1, 0); glVertex3i(-SPACE_SIZE,  SPACE_SIZE,  SPACE_SIZE);
        // -y sík
      glTexCoord2i(0, 0); glVertex3i(-SPACE_SIZE, -SPACE_SIZE, -SPACE_SIZE);
      glTexCoord2i(0, 1); glVertex3i(-SPACE_SIZE, -SPACE_SIZE,  SPACE_SIZE);
      glTexCoord2i(1, 1); glVertex3i( SPACE_SIZE, -SPACE_SIZE,  SPACE_SIZE);
      glTexCoord2i(1, 0); glVertex3i( SPACE_SIZE, -SPACE_SIZE, -SPACE_SIZE);
   glEnd();
}
```

10.3.3. Az űrhajó

Az űrhajó összetett geometriai alakzat, amelyre egy festést és felségjeleket szimbolizáló textúrát feszítünk. Az űrhajó alakját négyszöghálóval adjuk meg, amit egy geometriai modellezőprogrammal (*Maya*) alakítottunk ki (3.26. ábra).

A négyszögháló csúcspontjait a `shipgeom` 3D vektorokat tartalmazó tömbbe, a csúcspontokhoz tartozó textúra koordinátákat pedig a `shiptext` tömbbe olvassuk be.

```
//=================================================================
class Ship : public TexturedObject {
//=================================================================
   float   mass;                        // űrhajó tömege
   Vector gravity_force, rocket_force; // gravitáció és a hajtómű tolóereje
   enum AI_State {                      // gondolkodási állapot
      ESCAPE_FROM_PLANET,               // kitér a bolygóval ütközés elől
      ESCAPE_FROM_AVATAR,               // menekül az avatár elől
      CHASE_AVATAR                      // az avatárt üldözi
   } ai_state;
   float last_shot;                     // utolsó lövés óta eltelt idő
```

10.9. ábra. *Az űrhajó textúrája, geometriája és képe [10]*

```
   float closest_planet_dist;          // a legközelebbi bolygó távolsága

   struct Vector3 { float x, y, z; } * shipgeom;     // négyszögcsúcsok
   struct Vector2 { float u, v;} * shiptext;         // textúrakoordináták
   int nvertex;                                      // csúcspontok száma
public:
   Ship(Vector& pos0) : TexturedObject(pos0, "ship_texture.bmp") {
      mass = 0.1;
      ai_state = CHASE_AVATAR;         // gondolkozási állapot
      last_shot = 0;                   // utolsó lövés óta eltelt idő

        ,,a shipgeom és shiptext tömbök feltöltése''
    }
    int GetType() { return SHIP; }

    void ControlIt(float dt);
    void InteractIt(GameObject * object);
    void DrawIt(Camera * camera);
};
```

Az aktuális pozíció ismeretében a hajót el kell tolni a virtuális térben, majd az OpenGL hívások segítségével meg kell jeleníteni. Először egy rossz, de legalábbis tökéletesnek nem nevezhető megoldást mutatunk be:

```
//-----------------------------------------------------------------
void Ship::DrawIt(Camera * camera) { // rossz: nincs orientáció változás
//-----------------------------------------------------------------
   glTranslatef(position.x, position.y, position.z); // eltolás a pozícióra
   glBegin(GL_QUADS);                                // négyszögek rajzolása
      for(int i = 0; i < nvertex; i++) {
         glTexCoord2f(shiptext[i].u, shiptext[i].v);
         glVertex3f(shipgeom[i].x, shipgeom[i].y, shipgeom[i].z);
      }
   glEnd();
}
```

Ez az eljárás nem ad kielégítő eredményt, hiszen csak az űrhajó pozícióját módosítja, az orientációját nem. Tapasztalataink szerint a különböző repülőalkalmatosságok az orrukat követik, tehát az orientációjuk mindig olyan, hogy az orruk a pillanatnyi

sebességvektor irányába mutat (9.9. fejezet). Az eltoláson kívül tehát az űrhajót forgatni is kell, mégpedig a hajó orrát a sebességvektor irányába. Tegyük fel, hogy a modellben az űrhajó orra a `modell_head` egységvektor irányába mutat, a sebességvektorral párhuzamos egységvektor pedig `world_head`! Az első vektorból a másikra leképező forgatás tengelye merőleges mindkét vektorra, tehát a két vektor vektoriális szorzataként állítható elő. A forgatási szög koszinusza pedig a két vektor skaláris szorzata, hiszen feltételezésünk szerint a vektorok egységnyi hosszúak. Az űrhajó javított rajzolási függvénye tehát:

```
//-----------------------------------------------------------------
void Ship::DrawIt(Camera * camera) {
//-----------------------------------------------------------------
   glTranslatef(position.x, position.y, position.z);  // eltolás
   Vector modell_head(0, 0, 1);                        // orientáció számítás
   Vector world_head = velocity.UnitVector();          // orr iránya
   Vector rotate_axis = world_head % modell_head;      // forgatási tengely
   float cos_rotate_angle = world_head * modell_head; // forgatási szög
   float rotate_angle = acos(cos_rotate_angle) * 180 / M_PI;
   glRotatef(-rotate_angle, rotate_axis.x, rotate_axis.y, rotate_axis.z);

   glBegin(GL_QUADS);                                  // négyszögek rajzolása
   for(int i = 0; i < nvertex; i++) {
      glTexCoord2f(shiptext[i].u, shiptext[i].v);
      glVertex3f(shipgeom[i].x, shipgeom[i].y, shipgeom[i].z);
   }
   glEnd( );
}
```

Az űrhajó mozgásállapotát a bolygók vonzóereje (`gravity_force`) és a hajtóművek tolóereje (`rocket_force`) módosítja. Maguk az erők is időben változók, minek következtében az űrhajó bonyolult pályát jár be, amit csak fizikai animációval követhetünk. A fizikai animációhoz az eredő erőt, majd abból a gyorsulást kell meghatározni. A gyorsulás ismeretében a `GameObject` osztálytól örökölt animációs függvényt használhatjuk:

```
//-----------------------------------------------------------------
void GameObject::AnimateIt(float dt) {
//-----------------------------------------------------------------
   velocity += acceleration * dt;
   position += velocity * dt;
}
```

Szemben az idáig tárgyalt objektumokkal, az űrhajó intelligens, azaz hajtóműveit a hosszú távú céljainak érdekében működteti. A hosszú távú célok és az érdekükben foganatosított műveletek az alábbiak lehetnek:

- Az űrhajó szeretné elkerülni, hogy a bolygókkal ütközzön, hiszen egy ilyen ütközés számára fatális kimenetelű. Amennyiben az űrhajó egy bolygó közelébe kerül

(például a bolygó középpontjából mért távolsága kisebb mint a bolygó sugarának háromszorosa), a hajtóműveit a bolygó középpontja felé irányítva távolodni próbál mindaddig, amíg a távolság legalább 4 nem lesz.

- Az avatár az űrhajókra vadászik, amit az űrhajó a lehetőségei szerint megnehezít. Mivel az avatár az orra irányába tud lőni, ha az űrhajó az avatár orrát látja, akkor az avatár irányára merőlegesen próbál meglépni, ugyanis ekkor a legnehezebb eltalálni.

- Az űrhajó el akarja pusztítani az avatárt, így ha éppen nem egy bolygó közeléből menekül, akkor az avatárt veszi üldözőbe, tehát a hajtóműveit az avatártól az adott űrhajó felé mutató irányba állítja. Az avatárt kellő távolságra megközelítve, ha az avatár nagyjából az űrhajó orrának irányában van, az űrhajó egy foton rakétát lő ki. A lövés után az újabb rakéta indításához időre van szüksége, ezért a következő rakétát csak két másodperccel később indíthatja el (bajban is lenne a játékos, ha az ellenfelek golyószóróként onthatnák a rakétáikat).

A fentiek alapján az ellenséges űrhajók intelligenciája egy háromállapotú automatával jellemezhető, melynek állapotai:

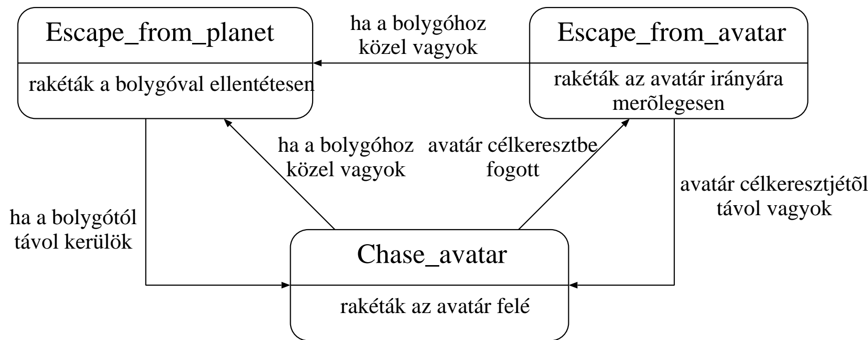
1. `ESCAPE_FROM_PLANET`: Menekülés a bolygó közeléből. Ilyenkor az űrhajó rakétát úgy vezéreli, hogy a tolóerő a bolygó közepétől feléje mutasson. Akkor kerül ebbe az állapotba, ha a távolság kisebb lesz, mint a bolygó sugarának háromszorosa, és mindaddig ezt a műveletet erőlteti, amíg nem távolodik el négy egység távolságra.

2. `ESCAPE_FROM_AVATAR`: Kitérés az avatár elől. Ez az állapot akkor következik be, ha az űrhajó éppen nem egy bolygótól menekül, és az avatár orrának (azaz sebességének) iránya és az avatárt az űrhajóval összekötő irány közötti szög kicsiny (mondjuk, ha a célzási szög koszinusza 0.9-nél nagyobb). Ekkor az űrhajó a tolóerőt az avatár és az űrhajó közötti irányra merőlegesen állítja be. Ebből az állapotból akkor lép ki, ha veszélyesen közel került egy bolygóhoz, vagy az avatár célzási szögének koszinusza 0.5 alá csökkent.

3. `CHASE_AVATAR`: Az avatár üldözése, amikor a tolóerő az űrhajótól az avatár felé mutat. Az űrhajó akkor lép ebbe az állapotba, ha a bolygóktól mért távolsága legalább az adott bolygó sugarának háromszorosa, és éppen nem az avatár elől tér ki.

Az állapotokat és az állapotátmeneteket a 10.10. ábra véges állapotú gépében foglalhatjuk össze. A lekerekített sarkú téglalapok az állapotok nevét és az itt végzett tevékenységet mutatják, az állapotátmeneteknél pedig az átmenet feltételét tüntettük fel.

Az űrhajó vézérlésének magja a `ControlIt()` függvényben van:



10.10. ábra. *Az űrhajó gondolkodásának véges állapotú gép modellje*

```
//-----------------------------------------------------------------
void Ship::ControlIt(float dt) {
//-----------------------------------------------------------------
   closest_planet_dist = 20;        // gyakorlatilag végtelenre állítjuk
   gravity_force = Vector(0, 0, 0); // az összegzéshez az erőt zérusra állítjuk
   Interact(root);                  // a világ objektumaival kapcsolatba lépünk
   acceleration = (gravity_force + rocket_force) / mass; // dinamika alaptv.
   last_shot += dt;                 // utolsó lövés óta eltelt idő

   if (ai_state == ESCAPE_FROM_PLANET && closest_planet_dist > 4)
      ai_state = CHASE_AVATAR;
}
```

Ez az eljárás az `Interact()` függvény segítségével kapcsolatba lép a játéktér többi szereplőjével, amelynek során a gravitációs erők összeadódnak (ezért inicializáljuk a gravitációs erőt zérusra), illetve az űrhajó beállítja a saját tolóerejét. Az eredő erő a gravitációs és a tolóerő összege, amelyből már a gyorsulás számítható. A `last_shot` változóban az utolsó lövés óta eltelt időt tartjuk nyilván, hogy a tüzelés frekvenciáját korlátozzuk.

Az űrhajó viselkedése tehát az egyes játékobjektumokkal folytatott párbeszéd eredménye. Az `Interact()` függvény végigszalad az összes játékobjektumon, és az űrhajó `InteractIt()` függvényének visszahívásával egyenként felkínálja a párbeszéd lehetőségét:

```
//-----------------------------------------------------------------
void Ship::InteractIt(GameObject * object) {
//-----------------------------------------------------------------
   if (object->GetType() == PLANET) {     // párbeszéd egy bolygóval
      Planet * planet = (Planet *)object;
      Vector dr = planet->position - position; // relatív helyzet
      float dist = dr.Length();                // távolság
      dr.Normalize();
                                               // Newton gravitációs törvény
      gravity_force += dr * fNewton * mass * planet->Mass() / pow(dist, 3);
```

```
      if (dist < planet->Radius()) {           // Ha bolygóval ütközik
         KillIt();                             // meghal
         root->Join(new Explosion(position));  // új robbanás
      }
      if (dist < closest_planet_dist) {        // Ha ez a legközelebbi bolygó
         closest_planet_dist = dist;
         if (dist < planet->Radius() * 3) ai_state = ESCAPE_FROM_PLANET;
         rocket_force = dr * (-ROCKET_POWER);  // menekül
      }
   }
```

```
   if (object->GetType() == AVATAR) {    // párbeszéd az avatárral
      Avatar * avatar = (Avatar *)object;
      Vector dr = avatar->position - position; // relatív helyzet
      float dist = dr.Length();                // távolság
      dr.Normalize();                          // az avatár iránya az űrhajóból
      Vector head = velocity.UnitVector();     // űrhajó haladási iránya
      Vector avatar_head = avatar->velocity.UnitVector(); // avatár iránya

      switch(ai_state) {                       // gondolkozik és cselekszik
      case ESCAPE_FROM_PLANET:                 // bolygótól távolodott
         break;
      case ESCAPE_FROM_AVATAR:                 // avatártól menekült
         if (-(dr * avatar_head) < 0.5) ai_state = CHASE_AVATAR;
         else rocket_force = (avatar_head % dr) * ROCKET_POWER;
         break;
      case CHASE_AVATAR:                       // avatárt üldözte
         if (-(dr * avatar_head) > 0.9) ai_state = ESCAPE_FROM_AVATAR;
         rocket_force = dr * ROCKET_POWER;
         break;
      }
                                               // avatár lőtávolban?
      if (last_shot > 2 && head * dr > 0.9 && dist < 8)  {
         root->Join(new PhotonRocket(position, velocity, this));    // tüzel
         last_shot = 0;
      }
   }
}
```

A függvény elején az űrhajó megkérdezi beszélgetőtársának típusát (`GetType()`), hiszen egészen más „hangnemet” használ egy bolygóval szemben, mint egy avatárral, a többiekkel pedig szóba sem áll.

A bolygók megnövelik az űrhajóra ható gravitációs erőt a bolygó tömegének és a bolygó–űrhajó távolságnak megfelelően. Ha a bolygó középpontjának és az űrhajónak a távolsága a bolygó sugaránál kisebb, akkor az űrhajó a bolygóval összeütközött, így az űrhajót megsemmisítjük, és helyére egy robbanást teszünk. Ha az űrhajó még él, akkor gondolkodik, azaz a bolygó távolságának megfelelően kijelöli az aktuális mesterséges intelligencia állapotot. Ha az űrhajó arra a következtetésre jut, hogy távolodnia kell a

bolygótól, akkor a rakétáit a bolygó irányába fordítja.

Az avatárral folytatott párbeszéd a relatív helyzet, az avatár és az űrhajó haladási irányának tisztázásával indul, majd az űrhajó elgondolkodik. Ha éppen egy bolygó mellől távolodik, akkor ez mindennél fontosabb számára, ezért az avatár helyzete nem érdekes. Ha viszont a bolygók elég távol vannak, akkor megvizsgálja, hogy az avatár célkeresztjének közelébe került-e, és ekkor megpróbál merőleges irányban eliszkolni. Ha az avatár éppen nem fenyegeti az űrhajót, akkor az űrhajó üldözőbe veszi az avatárt. Végül az űrhajó az állapotától függetlenül azt is megvizsgálja, hogy az avatár az orrának irányában van-e, mert ekkor egy fotonrakétát lő ki.

10.3.4. A fotonrakéta

A lövedék egy fotonrakéta, amely fényes tűzgolyóként vágtázik az űrben. Az ilyen geometriailag bonyolult, ámde gömbszimmetrikus elemekhez plakátokat használhatunk, ezért a lövedék osztályát (`PhotonRocket`) a plakátok `Billboard` osztályából származtatjuk.

```
//===============================================================
 class PhotonRocket : public BillBoard { // A lövedék egy plakát
//===============================================================
    float        age;      // a lövedék kora
    GameObject * source;   // a lövedéket kilövő játékobjektum
 public:
    PhotonRocket(Vector& pos0, Vector& shooter_velocity, GameObject* source0)
        : BillBoard(pos0, 0.3, "photon.bmp") {
        velocity = shooter_velocity + shooter_velocity.UnitVector() * 2;
        age = 0;
        source = source0;
    }
    int GetType() { return PHOTON_ROCKET; }
    void ControlIt(float dt);
};
```

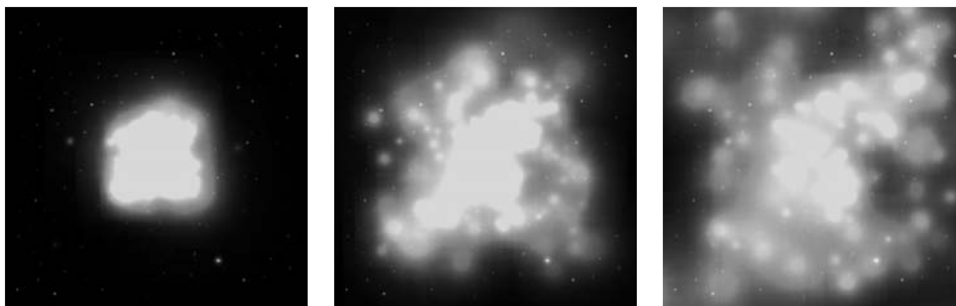
A lövedék meglehetősen kegyetlenül viselkedik. Ha a rendelkezésre álló keretidőben eltalál egy másik objektumot, akkor azt és önmagát is megsemmisíti. A találatot sugárkövetéssel ellenőrizzük, mert a lövedék gyors, így a diszkrét ütközésdetektálás könnyen hibázna. A lövedék a `source` változóban tárolja azon objektum (űrhajó vagy avatár) azonosítóját, amely kilőtte őt, így a forrásobjektumot kizárhatjuk az eltalálható céltárgyak közül. Erre azért van szükség, mert a lövés pillanatában a lövedék a forrásobjektumban van, tehát azt „eltalálja”, de mégsem semmisíti meg. Végül a lövedék akkor is megsemmisül, ha már 10 másodperce úton van anélkül, hogy célba ért volna.

```
//---------------------------------------------------------------
void PhotonRocket::ControlIt(float dt) {
//---------------------------------------------------------------
   float hit_time = dt;  // a keretidő, ameddig az ütközés találatnak számít
   Vector hit_point;     // találat helye
```

```
   GameObject * hit_object = Collide(root, hit_time, hit_point, source);

   if (hit_object) {                                 // Ha van találat
      KillIt();                                      // lövedék megsemmisül
      int hit_object_type = hit_object->GetType(); // Az eltalált tárgy típusa
      if (hit_object_type == PLANET ||              // Ha bolygót
          hit_object_type == SHIP)                   //    vagy űrhajót ér
         root->Join( new Explosion( hit_point ) ); //    akkor robban
      if (hit_object_type == AVATAR ||              // Ha avatárt
          hit_object_type == SHIP)                   //    vagy űrhajót ér
         hit_object->KillIt();                       //    akkor öl is
   } else {
      age += dt;                 // Ha 10 másodperc alatt sem talált el semmit
      if (age > 10) KillIt();    //    akkor a lövedék megsemmisül
   }
}
```

10.3.5. A robbanás

Ebben a fejezetben a *robbanás* jelenséggel mutatjuk be, hogy hogyan használhatjuk az általános részecske és részecskerendszer osztályokat. Az általános részecske osztály működését úgy finomíthatjuk, hogy a részecske állapotváltozóit a részecske születésekor kitöltjük. A természeti jelenségekben a részecskék véletlenszerű tulajdonságokat vesznek fel, az adott jelenség biztosította határok között (például egy tűzrészecske a függőleges iránytól legfeljebb 10 fokkal eltérve indulhat el, és színe két vörös árnyalat között változhat). A szimuláció során tehát ezeket a változókat a jelenségnek megfelelő valószínűségeloszlás mintáival kell kitölteni.



10.11. ábra. *A robbanás különböző fázisai*

Egy robbanásrészecskéhez például a következő osztályt használhatjuk:

```
//===============================================================
class ExplosionParticle : public Particle {
//===============================================================
public:
```

```
   ExplosionParticle(Vector& center) {
      position = center;            // minden részecske a középpontból indul
      time_to_live = Rand(2, 1);    // élettartam [1-3] sec között
      size = 0.001;                 // plakát méret kezdetben kicsi,
      dsize = Rand(0.5, 0.25) / time_to_live; // majd nő
                                    // a kezdeti sebesség gömbszimmetrikus
      velocity = Vector(Rand(0, 0.4), Rand(0, 0.4), Rand(0, 0.4));
                                    // a gyorsulás gömbszimmetrikus
      acceleration = Vector(Rand(0, 0.4), Rand(0, 0.4), Rand(0, 0.4));
            // szín kezdetben [1, 0.0--1, 0, 1] azaz átlátszatlan piros-sárga
      color = Color(1, Rand(0.5, 0.5), 0, 1);
            // szín egyre pirosabb és egyre átlátszóbb lesz
      dcolor = Color(0, -0.25, 0.0, -0.5) / time_to_live;
   }
};
```

A programban felhasználtuk a `Rand()` függvényt, amely a megadott átlag (`mean`) körül egy egyenletes eloszlású véletlenszámot állít elő a `variation` változóban megadott szélességű tartományban:

```
float  Rand(float mean, float variation) {
   return (mean + (2 * (float)rand()/RAND_MAX - 1) * variation);
}
```

A robbanás részecskerendszere (`Explosion`) a `ParticleSystem` általános részecskerendszer osztályból származik, megvalósítja, és meg is hívja annak a részecskéket kibocsátó `Emit()` tagfüggvényét:

```
//=================================================================
class Explosion : public ParticleSystem {
//=================================================================
public:
   Explosion(Vector pos0):ParticleSystem(pos0, "explosion.bmp") { Emit(200); }
   void Emit(int n) {                    // n részecskét kibocsát
      for(int i = 0; i < n; i++) {
         ExplosionParticle * particle = new ExplosionParticle(position);
         particle->next = particles;  // felvétel a láncolt listába
         particles = particle;
      }
   }
};
```

Értelmezzük a robbanás osztály működését és a robbanásrészecskék paramétereinek a hatását! A rendszer születésekor rögtön létrehoz 200 részecskét, amelyek aztán önálló életre kelnek. A robbanáskor a kibocsátás egyszeri, újabb részecskéket a rendszer nem bocsát ki. A részecskék a részecskerendszer helyén születnek meg. Ez a robbanás centruma. A sebesség és a gyorsulás gömbszimmetrikus, tehát a részecskefelhő is gömbszerűen fog fejlődni. A részecske kezdetben kicsiny, az idő előrehaladtával mérete fokozatosan nő, mialatt átlátszatlan sárgás színből egyre átlátszóbb pirosra vált. A színváltozás a kihűlés következménye.

A példából is látszik, hogy sok paramétert kell állítgatnunk, amíg a kívánt hatást elérjük. A folyamat során érdemes fizikai ismereteinkre és intuíciónkra támaszkodni (a robbanásnál például a mechanikai törvényekre és a feketetest sugárzás elméletére), de mégoly alapos előkészület után is a megfelelő jelenség létrehozása hosszadalmas próbálgatásos folyamat. A próbálgatásban sokat segíthetnek a *részecskerendszer-szerkesztő*k, amelyekben a paramétereket interaktívan állíthatjuk be, és a szerkesztő rögtön megmutatja a keletkező hatást.

10.3.6. Az avatár

A játékost a virtuális világban megszemélyesítő avatár típusát (`Self`) a játékmotor `Avatar` osztályából származtatjuk, hiszen ebben már benne van a számítógép felhasználójával való kapcsolattartás képessége:

```
//===============================================================
class Self : public Avatar {
//===============================================================
   float   mass;                         // a játékos űrhajójának tömege
   Vector  gravity_force, rocket_force; // gravitáció és rakétatolóerő
public:
   Self(Vector& pos0) : Avatar(pos0) {
      mass = 0.1;                // tömeg
      bounding_radius = 0.2;     // befoglaló gömb, hogy el lehessen találni
   }
   int GetType() { return AVATAR; }
   Vector Head() { return velocity.UnitVector(); }
   void ControlIt(float dt);
   void InteractIt(GameObject * object);
   void ProcessInput(Application * input);
};
```

A `Self` osztály vezérlő függvénye (`ControlIt()`) betűről-betűre megegyezik az űrhajó vezérlő függvényével, hiszen ez is egy űrhajó. A többi objektummal párbeszédet folytató `InteractIt()` azonban egyszerűbb, hiszen az avatárnak nem kell önállóan gondolkodnia, az ő eszét a játékos kölcsönzi:

```
//---------------------------------------------------------------
void Self::InteractIt(GameObject * object) {
//---------------------------------------------------------------
   if (object->GetType() == PLANET) {          // a bolygók vonzzák az űrhajót
      Planet * planet = (Planet *)object;
      Vector dr = planet->position - position;// relatív helyzet
      float dist = dr.Length();               // távolság
      gravity_force += dr * fNewton * mass * planet->Mass() / pow(dist, 3);
      if (dist < planet->Radius()) KillIt();  // a bolygóval ütközés fatális
   }
}
```

A beviteli eszközök állapotát a felüldefiniált `ProcessInput()` tagfüggvényben kérdezhetjük le. Ebben a játékban a játékos a SPACE billentyűvel újabb lövedéket

lőhet ki, a „q” billentyűvel gyorsíthat, az „a” billentyűvel lassíthat, az iránybillentyűkkel és az egérrel pedig a tolóerő irányát egy botkormányhoz hasonlatosan állíthatja. A kormányzást az `Avatar`-tól örökölt `Steering()` függvénnyel valósítjuk meg.

```
//-----------------------------------------------------------------
Avatar::ProcessInput(Application * input) {
//-----------------------------------------------------------------
   rocket_force = Steering(input) * ROCKET_POWER;  // kormányzás
   if (input->GetKeyStatus('q'))  rocket_force = Head() * ROCKET_POWER;
   if (input->GetKeyStatus('a'))  rocket_force = Head() * (-ROCKET_POWER);
   if (input->GetKeyStatus(' '))                   // SPACE-re lövünk
      root->Join(new PhotonRocket(position + velocity, velocity, this));
}
```

10.3.7. Az űrhajós játék főosztálya

A játékobjektumokat az általános játékmotorból származtatott `SpaceShootGame` osztály `Init()` függvénye hozza létre:

```
//=================================================================
 class SpaceShootGame : public GameEngine {
//=================================================================
public:
   SpaceShootGame(int window_width, int window_height)
       : GameEngine("Space Shoot Game", window_width, window_height) { }

   void Init()  {                         // játékmotor létrehozása
      GameEngine::Init();                 // z-buffer bekapcsolás

      world = new Space("stars.bmp");     // az űr létrehozása

                                          // az űrhajók létrehozása
      world->Join(new Ship(Vector(-4, 8, 2)));
      world->Join(new Ship(Vector( 3,10, 4)));
      world->Join(new Ship(Vector( 5, 6, 6)));
      world->Join(new Ship(Vector( 6, 9,-4)));
      world->Join(new Ship(Vector( 5, 5,-3)));
      world->Join(new Ship(Vector( 0,10,-2)));

                                         // a bolygók létrehozása
      world->Join(new Planet(Vector( 0,0,0), 3.0, "sun.bmp"));
      world->Join(new Planet(Vector( 6,0,0), 0.2, "mercury.bmp"));
      world->Join(new Planet(Vector( 7,0,0), 0.4, "venus.bmp"));
      world->Join(new Planet(Vector( 8,0,0), 0.4, "earth.bmp",  23));
      world->Join(new Planet(Vector(10,0,0), 0.3, "mars.bmp",   25));
      world->Join(new Planet(Vector(11,0,0), 0.7, "jupiter.bmp", 3, 0.8, 0.9));
      world->Join(new Planet(Vector(12,0,0), 0.6, "saturn.bmp", 26, 0.7, 1.2));
      world->Join(new Planet(Vector(13,0,0), 0.5, "uranus.bmp", 82, 0.6, 0.8));
      world->Join(new Planet(Vector(15,0,0), 0.5, "neptun.bmp", 29, 0.6, 0.8));
      world->Join(new Planet(Vector(17,0,0), 0.2, "pluto.bmp",  62));

                                         // az avatár létrehozása
      avatar = new Self(Vector(0,-9,1));
      world->Join(avatar);
```

```
   }
};
```

Végül az alkalmazás belépési pontján nem kell mást tennünk, mint a játék főobjektumát létrehozni:

```
void Application::CreateApplication() { new SpaceShootGame(500, 500); }
```

A főobjektum megszületésekor a játékmotor is létrejön, a játékhurok beindul, és a főobjektum létrehozza a játék szereplőit is. A többi már az avatárt vezérlő játékos ügyességén múlik.

10.4. Hierarchikus szereplők

Az ismertetett játékban minden játékobjektum egyenlő volt, nem vezettünk be alá- és fölérendeltségi viszonyokat. Az egyenlő elbírálás elvét a játékobjektumok láncolt lista adatstruktúrája is mutatja, hiszen ebben nincsenek különböző hierarchia szintek. Az életben és az összetettebb játékokban azonban a hierarchikus viszonyok kezelésére is fel kell készülni. Bolygórendszerünkben idáig minden bolygó a koordinátarendszer origója körül keringett, holott a valóságban a bolygók a Nap körül keringenek, a holdak pedig az anyabolygók körül. Ez már egy háromszintű hierarchia, amelynek a tetején a Nap áll, alatta gyermekei, a bolygók, amelyek pedig a saját holdjaikat uralják. Egy másik lehetséges példa a fegyveres ellenségeké, akik a fegyverüket magukkal hordozzák, amitől csak halálukkor válnak el. A hierarchikus mozgással a 9.11. fejezetben foglalkoztunk, és megállapítottuk, hogy ekkor egy objektumra nem csupán a saját mozgásának transzformációit, hanem a szülő, nagyszülő stb. transzformációit is alkalmazni kell (pontosabban ezen transzformációk egy részét). A holdak például követik az anyabolygójuk mozgását, de nem hat rájuk az anyabolygó tengelyferdülése és saját forgása. A hierarchikus viszonyok kifejezésére a `Member` osztályunkat módosítani kell, és az egyes láncolt lista elemekhez, gyermek láncolt listákat és a szülő azonosítóját is hozzá kell tenni.

```
//===============================================================
class Member : public GameObject {
//===============================================================
protected:
   static Member * root;              // a hierarchia gyökere
   Member        * next;              // láncoló mutató
   Member        * children;          // a gyermekek
   Member        * parent;            // a szülő
public:
   Member(Vector pos0) : GameObject(pos0) {
       next = children = parent = NULL; if (!root) root = this;
   }
```

```
void Join(Member * obj);          // új elem hozzávétele a listához

void AddChild(Member * obj) {     // új elem a gyermeklistához
   if (!children) children = obj; // ő az elsőszülött
   else children->Join( obj );    // különben az utolsó helyre
   obj->parent = this;
}

void Interact(Member * obj) {     // párbeszéd
   if (obj != this) {             // nem beszélünk magunkban
        InteractIt(obj);          // az obj objektummal
        if (obj->children) Interact(obj->children); // az ő gyermekeivel is
   }
   if (children) children->Interact(obj); // a mi gyermekeink is beszélnek
   if (obj->next)  Interact(obj->next); // párbeszéd a farokkal
}
GameObject * Collide(Member * obj, float& mhit_time,
                     Vector& mhit_point, GameObject * source) {
   GameObject * hit_obj = NULL;    // először a farok elemeivel ütköztetünk
   if (obj->next) hit_obj = Collide(obj->next, mhit_time, mhit_point, source);
   float hit_time;                 // ezen objektummal az ütközés ideje
   Vector hit_point;               // ezen objektummal az ütközés helye
   if (obj != this && obj != source) { // magunkkal, forrással nem ütközünk
      if (obj->children) {             // gyermekeink ütköznek
         GameObject *chit;
         chit = Collide(obj->children, mhit_time, mhit_point, source);
         if (chit) hit_obj = chit;
      }
      if (CollideIt(obj, hit_time, hit_point) &&  // magunk ütközünk
          hit_time < mhit_time ) {   // az ütközés korábbi-e mint az előzőek
         mhit_time = hit_time;
         mhit_point = hit_point;
         hit_obj = obj;
      }
   }
   return hit_obj;
}

void Control(float dt) {                 // vezérlés
   ControlIt(dt);                        // ezt vezéreljük
   if (children) children->Control(dt); // gyermekekre
   if (next) next->Control(dt);          // vezérlés a farokra
}

void Animate(float dt) {                 // animáció
   AnimateIt(dt);                        // ezt animáljuk
   if (children) children->Animate(dt); // gyermekekre
   if (next) next->Animate(dt);          // a farok animációja
}

virtual void BeforeDraw();               // rajzolás prológus
void Draw(Camera * camera) {             // rajzolás
   BeforeDraw();                         // állapot mentése
   DrawIt(camera);                       // ezt rajzoljuk
   if (children) children->Draw(camera);// gyermekekre
   AfterDraw();                          // állapot visszaállítás
   if (next) next->Draw(camera);         // farok rajzolása
```

```
   }
   virtual void AfterDraw();                // rajzolás epilógus

   virtual ~Member() {   // a gyermekeket is felszabadítjuk
      while (children) {
         Member * child = children;
         children = children->next;
         delete child;
      }
   }
};
```

Figyeljük meg, hogy a hierarchikus rendszert vezérlő, animáló, rajzoló stb. függvények nagyon hasonlóak az egyetlen láncolt listát kezelő megvalósításhoz, de most a rekurziót nem csak a láncolt lista farokrészére, hanem a gyermekek irányában is folytatni kell! A `Draw()` metódus megvalósításában a gyermekeket a transzformációt visszaállító `AfterDraw()` hívás előtt rajzoljuk fel, így lehetőséget adunk arra, hogy a gyermek örökölje szülőjének transzformációs mátrixát, amihez a saját transzformációját hozzácsatolhatja.

Példaként alakítsuk át a Naprendszer modellünket úgy, hogy a bolygók a Nap gyermekei, a Hold pedig a Föld gyermeke legyen:

```
//=================================================================
class Planet : public TexturedObject { // a bolygó egy textúrázott objektum
//=================================================================
    GLUquadricObj *sphere, *disk; // kvadratikus felületek
    float   mass;                 // tömeg
    float   rot_angle, rot_speed; // forgási szög és sebesség
    float   rev_angle, rev_speed; // keringési szög és sebesség
    float   dist;                 // bolygó-nap távolság
    float   axis_angle;           // tengelyferdülés
    float   r1, r2;               // gyűrű külső-belső sugarak
public:
    Planet(Vector& pos0, float R, char * filename,
           float axis_angle0 = 0, float r10 = 0, float r20 = 0);

    void AddChild(Planet * child) { // új elem hozzávétele a gyermeklistához
        Member::AddChild(child);
                     // keringési sebesség az égi mechanika törvényei szerint
        child->position -= position;
        child->dist = child ->position.Length(); // szülőbolygó távolság
        float parent_force = pow(bounding_radius, 3) * fNewton * planet_density;
        child -> rev_speed = sqrt(parent_force/pow(child->dist, 3));
        child->rev_angle = Rand(180, 180);        // véletlen kezdeti szög
    }

    int GetType() { return PLANET; }
    float Radius() { return bounding_radius; }
    float Mass() { return mass; }

    void DrawIt(Camera * camera) {
       glPushMatrix();
       ,,rajzolás mint a korábbi megoldásban''
```

```
      glPopMatrix();
    }
    void AnimateIt(float dt) {
      ,,animáció mint a korábbi megoldásban''
      if (parent) position += parent->position; // megy a szülővel
    }
};
```

Újdonságként megjelent az `AddChild()` tagfüggvény, amely egy gyermek bolygót rendel a szülőhöz, áttranszformálja a gyermekbolygót a szülőhöz rögzített koordinátarendszerbe és kiszámítja a gyermek keringési sebességét. A `DrawIt()` függvényben az egyetlen változás az, hogy a transzformációt elmenti és visszaállítja. Erre azért van szükség, hogy a bolygó saját transzformációi, a helyzetnek megfelelő eltoláson kívül ne hassanak a holdra (például a tengely körüli forgatás és a dőlésszög ne befolyásolja a hold pályáját). A bolygó helyvektorát az `AnimateIt()` függvényben adjuk hozzá a hold helyvektorához, így a hold követi az anyabolygót.

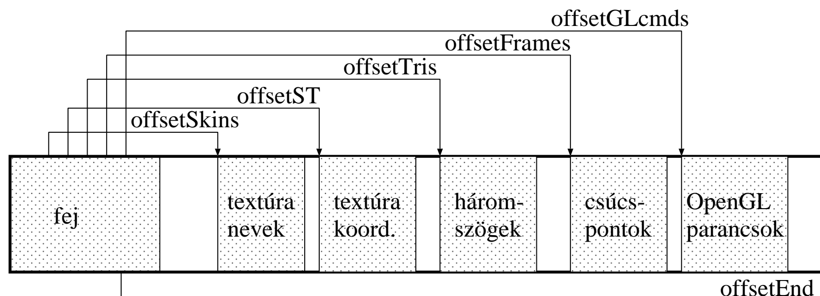
A hierarchikus naprendszert működtető javított játékunk főosztálya:

```
//=================================================================
class SpaceShootGame : public GameEngine {
//=================================================================
public:
   SpaceShootGame(int window_width, int window_height)
       : GameEngine("Space Shoot Game", window_width, window_height) { }

   void Init()  {          // játékmotor létrehozása
      GameEngine::Init();
      world = new Space("stars.bmp");    // az űr létrehozása
                                         // a bolygók létrehozása
      Planet * sun = new Planet(Vector(0,0,0), 3, "sun.bmp");
      world->Join( sun );
      sun->AddChild(new Planet(Vector(6,0,0), 0.2, "mercury.bmp"));
      sun->AddChild(new Planet(Vector(7,0,0), 0.4, "venus.bmp"));
      Planet * earth = new Planet(Vector(8,0,0), 0.4, "earth.bmp", 23);
      sun->AddChild( earth );
      earth->AddChild(new Planet(Vector(9,0,0), 0.1, "moon.bmp"));
      sun->AddChild(new Planet(Vector(10,0,0), 0.3, "mars.bmp", 25));
      sun->AddChild(new Planet(Vector(11,0,0), 0.7, "jupiter.bmp", 3, 0.8, 0.9));
      sun->AddChild(new Planet(Vector(12,0,0), 0.6, "saturn.bmp", 26, 0.7, 1.2));
      sun->AddChild(new Planet(Vector(13,0,0), 0.5, "uranus.bmp", 82, 0.6, 0.8));
      sun->AddChild(new Planet(Vector(15,0,0), 0.5, "neptun.bmp", 29, 0.6, 0.8));
      sun->AddChild(new Planet(Vector(17,0,0), 0.2, "pluto.bmp", 62));
                                         // az avatár létrehozása
      avatar = new Self(Vector(0, -9, 1));
      world->Join(avatar);
   }
};
```

10.5. Mozgó karakterek

A játékokban olyan szereplőkkel is találkozhatunk, akik geometriája a mozgás során változik, például emberszerű lényekkel, akik sétálnak, lehajolnak, elesnek, birodalmi lépegetőkkel, amelyek lábaik emelgetésével haladnak előre, vagy helikopterekkel, amelyek mozgatják a propellerüket stb. Az ilyen belső mozgással rendelkező szereplők leírásához az összes lehetséges mozgásfázist ismerni kell. Az OBJ fájlformátum sajnos nem tárolja a mozgásfázisokat, a *VRML*, a *3DS* (3D Studio formátuma), illetve a Quake játék *MD2* [53] vagy *MD3*[5] [15] formátuma viszont igen. A következőkben az MD2 formátumú karakterek beolvasását és animálását tárgyaljuk. Ilyen karaktereket például a *Milkshape 3D*[6] programmal hozhatunk létre.

Az MD2 egy bináris fájlformátum, amely a karakter geometriáját és mozgási lehetőségeit írja le. A fájl, mint a geometria vagy képleíró fájlok általában, fejrésszel kezdődik, amely összefoglalja a karakter jellemzőit, és megadja azt is, hogy a modell különböző részei, mint a textúrafáljok nevei, a textúra koordináták, a háromszögháló, a keretek definíciója, és a háromszögháló értelmezéséhez használható OpenGL parancsok a fájlban hol találhatóak (10.12. ábra).



10.12. ábra. *Egy MD2-es fájl szerkezete*

A fájl fejrészének szerkezete az Intel processzorok számábrázolását feltételezve a következő struktúrának felel meg:

```
//===============================================================
struct MD2FileHeader {
//===============================================================
   int ident;        // mágikus szám, amely karakteresen "IDP2"
   int version;      // verzió, jelenleg 8
   int skinwidth;    // textúra szélessége
   int skinheight;   // textúra magassága
   int framesize;    // egy keret mérete bájtban
```

[5] http://www.planetquake.com/polycount
[6] http://www.milkshape3d.com/

```
   int numSkins;     // textúrák száma
   int numXYZ;       // 3D pontok száma
   int numST;        // textúra pontok száma
   int numTris;      // háromszögek száma
   int numGLcmds;    // OpenGL parancsok száma
   int numFrames;    // keretek száma
   int offsetSkins;  // a textúranevek helye a fájlban (minden név 64 bájtos)
   int offsetST;     // a textúra koordináták helye a fájlban
   int offsetTris;   // a háromszögek helye a fájlban
   int offsetFrames; // az összes keret pontjainak helye a fájlban
   int offsetGLcmds; // az OpenGL parancsok helye a fájlban
   int offsetEnd;    // a fájl mérete (a fájl végének helye a fájlban)
};
```

A fájl testének első blokkja 64 bájtonként egy-egy textúranevet tartalmaz, amelyek *PCX* típusú képfájlokat jelölnek ki. Felmerülhet a kérdés bennünk, hogy mi értelme van egyetlen karakterhez egyszerre több textúrát is hozzárendelni. A játék ezeket a textúrákat tetszése szerint használhatja, például a karaktert az állapota alapján átöltöztetheti. A karakter teljes harci díszben indul a csatába, de közben páncélzatának egy részét elveszíti, ruhája és teste pedig egyre véresebb lesz. Másrészt, a járulékos képeket felhasználhatjuk multitextúrázáshoz, vagy akár bucka leképzéshez is.

A következő blokk a textúra koordinátákat tárolja, mint egészekből álló számpárok tömbjét, amelynek egyetlen elemét az alábbi struktúrával jellemezhetjük:

```
struct MD2TextureIndex { short s, t; };
```

Ezekből a számokból úgy kapunk az egységnégyzetbe eső u, v textúra koordinátákat, hogy elosztjuk őket a textúra szélességével illetve magasságával. A normalizált textúra koordinátákat a következő struktúrában tároljuk:

```
struct MD2TextureCoord { float u, v; };
```

A modell mozog, azaz az egyes pontok az időben máshova kerülhetnek. Az MD2 formátum a változást úgy adja meg, hogy egy-egy pont helyét annyiszor tartalmazza, ahány kulcskeretből áll az animáció. Az összes pont összes kulcskeretben érvényes koordinátáit egy tömbben találjuk, amelynek egyetlen eleme:

```
struct MD2FramePoint {
   unsigned char v[3];          // pont koordináták
   unsigned char normalIndex;   // normálvektor index (nem használt)
};
```

Egyetlen pont Descartes-koordinátái a `v` tömbben vannak, egyetlen koordinátát csak egyetlen bájton adhatunk meg. Ez egyrészt tömör, másrészt viszont így önmagában jelentősen korlátozná a modell nagyságát és mozgásának kiterjedését. Szerencsére a koordinátákat egy keretenként szabályozható skálázási és eltolási transzformációnak is

alávethetjük, amely a tömör geometriát a szükséges méretben kiterjesztheti. Egyetlen kerethez skálázási és eltolási transzformáció, a keret neve, valamint a pontok tömbje tartozik:

```
//===============================================================
struct MD2Frame {
//===============================================================
    float         scale[3];       // skálázás
    float         translate[3];   // eltolás
    char          name[16];       // a keret neve, például: "run04"
    MD2FramePoint fp[1];  // az egy bájtos koordinátákat tartalmazó tömb

    Vector Point(int i) {
        float x = scale[0] * fp[i].v[0] + translate[0];
        float y = scale[1] * fp[i].v[1] + translate[1];
        float z = scale[2] * fp[i].v[2] + translate[2];
        return Vector(x, y, z);
    }
};
```

Ebben az adatstuktúrában mindig a fájl fejrészében található `numXYZ` változóban megadott darabszámú `MD2FramePoint` típusú pont szerepel. Mivel ez nem konstans kifejezés, a C nyelv szintaktikai szabályai nem engedik meg, hogy az `fp` tömb méretét így vegyük fel. Ehelyett a deklarációban a tömb méretét 1-nek tekintettük, amit szükség esetén túlcímzünk. Ha a helyfoglalás korrekt, akkor ebből semmiféle probléma sem származhat. A struktúrát még kiegészítettük a `Point()` tagfüggvénnyel, amely kiszámítja `i`-edik pont helyét az adott keretben.

Az MD2-es formátum a felületeket háromszöghálóval közelíti, az egyes háromszögeket pedig a három csúcspont indexével, valamint a textúrapontok indexeivel adja meg:

```
//===============================================================
struct MD2Triangle {
//===============================================================
   unsigned short vertex_index[3]; // a csúcspontok indexei
   unsigned short st_index[3];     // a textúra koordináták indexei
};
```

A tárolt indexek a fájlban átadott, az `offsetFrames`-edik bájtról kezdődő csúcspont tömbre, illetve az `offsetST`-edik bájttól induló textúrapont tömbre vonatkoznak.

Az MD2 fájlban a háromszögeket az OpenGL számára optimalizált formátumban is megtaláljuk. A `offsetGLcmds` kezdőcímtől kezdve `numGLcmds` darab háromszöghálót találunk, amely `GL_TRIANGLE_STRIP` vagy `GL_TRIANGLE_FAN` típusú lehet (3.4.1. fejezet). Minden hálóleírás egy egész számmal (`int`) kezdődik, amelynek abszolút értéke a hálóban szereplő csúcsok számát, előjele pedig a háló típusát adja meg (a pozitív érték `GL_TRIANGLE_STRIP`-et, a negatív pedig `GL_TRIANGLE_FAN`-t jelent). A kezdő egészt követően az egész abszolút értékének megfelelő darabszámú csúcspontot találunk az alábbi formátumban:

```
//=============================================================
struct MD2GLCommandPoint {
//=============================================================
   float s, t;                  // textúra koordináták
   int   vertex_index;          // normálvektor index (nem használt)
};
```

A karakterhez tartozó háromszöghálót, és a csúcspontokhoz tartozó textúra koordinátákat tehát két különböző módon is kiolvashatjuk. A második módszer kétségkívül hatékonyabb, mert tálcán kínálja az adatokat az OpenGL számára. Egyrészt az illeszkedő háromszögek közös csúcspontjait nem kell annyiszor átadni, ahány háromszög illeszkedik egy pontra, másrészt pedig a textúra koordinátákat is már normalizálták az OpenGL igényeinek megfelelően.

Az MD2 formátumú karakterek általában textúrázottak, ezért az `MD2Object` típust a `TexturedObject` osztályból származtatjuk. Egy karakter tárolja az animációs kulcskereteinek a számát (`nframes`), a felület csúcspontjainak a számát (`nvertices`), ami a mozgás miatt a statikus karakter csúcspontszámának és a kulcskeretek számának a szorzata, valamint a felületet közelítő háromszögeknek (`ntriangles`) és a textúrapontoknak (`nST`) a számát. A modellhez három tömb tartozik. Egy tömb a háromszögeket, egy a normalizált textúrapontokat, végül pedig egy a csúcspontokat tartalmazza az összes keretben.

```
//=============================================================
class MD2Object : public TexturedObject {
//=============================================================
   int           nframes;    // keretek száma
   int           nvertices;  // csúcsok száma
   int           ntriangles; // háromszögek száma
   int           nST;        // textúra koordináták száma
   MD2Triangle  * triangles;  // háromszögek
   TextureCoord * text_coord; // textúra indexek
   Vector       * vertices;   // csúcspontok

protected:
   float model_time;          // modell animációs ideje
   int   frame1, frame2;      // a modell időt közrefogó két kulcskeret
   float inbetween;           // a két kulcskeret közötti pillanatnyi idő
   int   anim_start, anim_end;// animációs keretek kezdete és vége
   float anim_speed;          // animáció sebessége
   float head_angle;          // merre néz

   float ComputeBoundingRadius(); // a befoglaló gömb sugarának számítása

public:
   MD2Object(Vector& pos0, char * model_file, float scale);
   void AnimateIt(float dt);
   void DrawIt(Camera * camera);

   void SetAnimationState(int start, int end) { // animációs fázis
       anim_start = start;
```

```
      anim_end = end;
   }

   virtual void AnimationOver();  // mit tegyünk a fázis végén
};
```

A karakter összes lehetséges mozgása a kulcskeretekben van, amit több fázisra bonthatunk (a karakter álldogál, guggol, támad, fut stb.). A karakter animálása egy fázis egyszeri, vagy akár ciklikus (például a futás) lejátszását jelenti. Ezt a műveletet a fázis kezdetét (`anim_start`), végét (`anim_end`), lejátszásának sebességét (`anim_speed`) és az aktuális időponthoz tartozó modellidőt (`model_time`) tároló változók felhasználásával hajthatjuk végre. A modellidő és az animációs sebesség szorzata meghatározza, hogy pontosan melyik keretet kell bemutatni. A kulcskereteket az egész számokhoz kapcsoljuk, a kulcskeretek közötti keretek azonosítója viszont általában nem egész szám. A keretazonosítót közrefogó két egész szám, a `frame1` és `frame2` két kulcskeretet jelöl ki, amelyek között az aktuális keretet interpolálni kell. Az interpolációs paramétert a keretazonosító törtrésze fejezi ki, amelyet az `inbetween` változóban tárolunk. A `head_angle` a karakter orrának irányát jelöli ki a vízszintes síkon.

Az osztály konstruktora betölti a karaktert leíró geometriai és textúrafájlt, majd inicializálja a mozgásváltozókat. Figyeljük meg, hogy most a karakter betöltése előtt nem ismerjük a textúrafájl nevét, ezért az osztály konstruktorában az alaposztályt enélkül inicializáljuk! A textúra a tényleges értékét csak a modell betöltése után veheti fel. A betöltéshez a teljes fájlt a memóriába másoljuk, majd a fejrész alapján kimazsolázzuk a fájlban tárolt információkat, és feltöltjük a karakter háromszög, csúcspont és textúrapont tömbjeit, valamint beolvassuk a karakterhez tartozó textúrát:

```
//-----------------------------------------------------------------
MD2Object::MD2Object(Vector& pos0, char * model_file, float scale)
        : TexturedObject(pos0, NULL)  {
//-----------------------------------------------------------------
   FILE * file = fopen(model_file, "rb");     // fájl megnyitása
   fseek(file, 0, SEEK_END);                  // fájl hosszának kitalálása
   int file_length = ftell(file);
   fseek(file, 0, SEEK_SET);                  // vissza a fájl elejére

   char * buffer = new char [file_length+1];  // fájlt a bufferbe olvassuk
   fread(buffer, sizeof(char), file_length, file);
   MD2FileHeader * header  = (MD2FileHeader *)buffer;   // a fájl feje
   nvertices = header->numXYZ;                // a csúcspontok száma
   nframes = header->numFrames;               // a keretek száma
   nST = header->numST;                       // textúrapontok száma
   ntriangles = header->numTris;              // háromszögek száma

   vertices = new Vector [header->numXYZ * header->numFrames]; // csúcs tömb
   for(int iframe = 0; iframe < nframes; iframe++) {
      int act_frame_offset = header->offsetFrames + header->framesize * iframe;
      MD2Frame * pframe = (MD2Frame *)&buffer[act_frame_offset];
      for (int i = 0; i < nvertices; i++)
```

```
         vertices[nvertices * iframe + i] = pframe->Point(i) * scale;
   }
   bounding_radius = ComputeBoundingRadius(); // befoglaló gömb sugara

                              // a textúrapontok beolvasása
   MD2TextureIndex * st = (MD2TextureIndex *)&buffer[header->offsetST];
   text_coord = new TextureCoord[nST];
   for(int i = 0; i < nST; i++) {
       text_coord[i].u = (float)st[i].s / header->skinwidth;   // normalizálás
       text_coord[i].v = (float)st[i].t / header->skinheight;
   }

   triangles = new MD2Triangle[ntriangles]; // háromszögek beolvasása
   memcpy(triangles,&buffer[header->offsetTris],ntriangles*sizeof(MD2Triangle));
   texture = Texture(tfilename, FALSE);     // textúra létrehozása

   anim_start = 0; anim_end = nframes - 1;  // animációs állapot
   anim_speed = 1;
   model_time = head_angle = frame1 = frame2 = 0;
   delete buffer;
}
```

Az animáció az aktuális időhöz tartozó modellidő megváltoztatását jelenti, amelyből az animációs sebesség szerint kiszámíthatjuk a megjelenítendő keret azonosítóját, majd a keretazonosítót közrefogó két egész szám kiválasztásával az interpoláció két kulcskeretét (`frame1` és `frame2`), végül a keretazonosító törtrészével az interpolációs paramétert.

```
//-----------------------------------------------------------------
void MD2Object::AnimateIt(float dt) {
//-----------------------------------------------------------------
   model_time += dt;                       // globális idő
   float fframe = anim_start + model_time * anim_speed;
   frame1 = (int)fframe;                   // első kulcskeret
   inbetween = fframe - frame1;            // interpolációs súly
   frame2 = frame1 + 1;                    // második kulcskeret
   if (frame2 > anim_end) AnimationOver();  // fázis vége
   position += velocity * dt;              // mozgás
   if (velocity.Length() > 0.00001)        // erre néz
      head_angle = atan2(velocity.y, velocity.x) * 180/M_PI;
}
```

A művelet elvégzésekor arra is figyelnünk kell, hogy túlléptük-e az utolsó keretet. Ha az animációs szekvencia befejeződött, akkor újra elindíthatjuk a lejátszását, vagy másik mozgássor feldolgozásába kezdhetünk. A döntés a karakter típusától és az adott alkalmazástól függ, ezért egy virtuális `AnimationOver()` függvényt hívunk meg, amely beállítja az animáció folytatási paramétereit.

Az `AnimationOver()` alapértelmezésű változata ciklikus lejátszást valósít meg:

```
//------------------------------------------------------------------
void MD2Object::AnimationOver() { // kezdi elölről az animációt
```

```
//-----------------------------------------------------------------
   int length = anim_end - anim_start;
   while(frame1 > anim_end) frame1 -= length;
   while(frame2 > anim_end) {
      frame2 -= length;
      model_time -= length / anim_speed;
   }
}
```

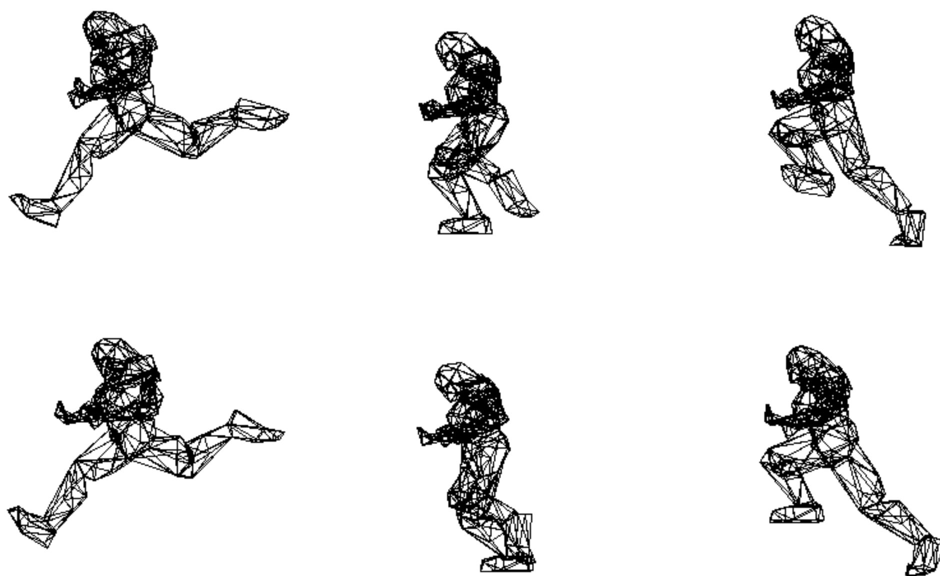
A `DrawIt` rajzoló függvény annak ellenőrzésével indul, hogy a karakter a látható tartományban van-e (nem látható karakterek rajzolásával felesleges vesződni), majd a modellidőhöz tartozó mozgásfázist a kulcskeretekből interpoláljuk. A modellidőt közrefogó két kulcskeret között általában elegendő a modellidőnek a kulcskeretektől vett távolsága alapján a két kulcskeret pontjait lineárisan súlyozva átlagolni (lineáris interpoláció):

```
//---------------------------------------------------------------
void MD2Object::DrawIt(Camera * camera) {
//---------------------------------------------------------------
   if (!camera->InViewFrustrum(position, bounding_radius)) return;
                                     // karakter globális transzformációi
   glTranslatef(position.x, position.y, position.z);
   glRotatef(head_angle, 0, 0, 1);      // erre néz

   glBegin(GL_TRIANGLES);               // kirajzolás háromszögenként
   for (int i = 0; i < ntriangles; i++) {
      for(int j = 0; j < 3; j++) {      // interpoláció + rajzolás
          glTexCoord2f(text_coord[triangles[i].st_index[j]].u,
                       text_coord[triangles[i].st_index[j]].v);

          Vector v1 = vertices[nvertices*frame1+triangles[i].vertex_index[j]];
          Vector v2 = vertices[nvertices*frame2+triangles[i].vertex_index[j]];
          Vector v = v1 + (v2 - v1) * inbetween;   // lineáris interpoláció

          glVertex3f(v.x, v.y, v.z);
      }
   }
   glEnd();
}
```

A mozgásfázisokat tetszőleges módon szétoszthatjuk a rendelkezésre álló kulcskeretekben, de célszerű játékprogramonként egy-egy egységes rendszert kialakítani. A 10.1. táblázatban a *Quake2* játékban alkalmazott kulcskeretrendszert mutatjuk be. Ha a saját játékunkban is ezt alkalmazzuk, akkor a Quake2 játék számára tervezett karaktereket is megfelelően mozgathatjuk.

10.6. Terepek

A játékok gyakran hegyes-völgyes vidéken folynak, amelyen épületek állnak és karakterek szaladgálnak. Ebben a fejezetben a *terep*ek (*terrain*) létrehozásával foglalko-



10.13. ábra. *A Morbo nevű ellenség futásának kulcskeretei (40-45)*

mozgásfázis	kulcskeretek
álldogál	0 – 39
fut	40 – 45
támad	46 – 53
szenved (három változat)	54 – 57, 58 – 61, 62 – 65
ugrik	66 – 71
legyint	72 – 83
tiszteleg (szalutál)	84 – 94
gúnyolódik	95 – 111
integet	112 – 122
mutat, céloz	123 – 134
helyben guggol	135 – 153
guggolva halad	154 – 159
guggolva támad	160 – 168
guggolva szenved	169 – 172
guggolásból indulva meghal	173 – 177
meghal (három változat)	178 – 183, 184 – 189, 190 – 197

10.1. táblázat. *A Quake2 játékban alkalmazott mozgásfázis rendszer*

zunk. A terep egy alkalmas *magasságmező* (3.4.5. fejezet). Az alapsík rácspontjait egy kép pixeleinek, az itt érvényes magasságértékeket pedig a képpont szürkeségi szintjének feleltethetjük meg, így a terep geometriai adatait egy szürkeárnyalatos képből olvashatjuk ki (10.14. ábra). A terep megjelenítéséhez még egy, vagy több textúra is szükséges. A terep magasságmezőjét és textúráját párhuzamosan hozhatjuk létre a terepmodellező programokban, vagy egy térkép digitalizálásából is kiindulhatunk.

A terep egy statikus, textúrázott játékobjektum:

```
//===============================================================
class TerrainObject : public TexturedObject {
//===============================================================
   int    width, length;            // a magasságmezőt leíró kép felbontása
   Byte * height_field;             // magasságmező kép
   float  wwidth, wlength, wheight; // az objektum méretei a virtuális világban
public:
   TerrainObject(Vector& pos0, char * height_file, char * texture_file,
                 float wwidth0, float wlength0, float wheight0)
      : TexturedObject(pos0, texture_file) {
      ImageFile himage(height_file, width, length); // magasságmező kép
      height_field = new Byte[width * length];
      for (int i = 0; i < width*length; i++) height_field[i] = himage.Gray(i);
      wwidth = wwidth0; wlength = wlength0; wheight = wheight0 / 256;
   }
   Vector GetPoint(int X, int Y) { // egy magasságképbeli rácshoz tartozó pont
        return Vector(X * wwidth / width - wwidth/2,
                      Y * wlength / length - wlength/2,
                      height_field[Y * width + X] * wheight);
   }
   float Height(float x, float y); // a magasság egy x, y világbeli pont felett
   void DrawIt(Camera * camera);
};
```

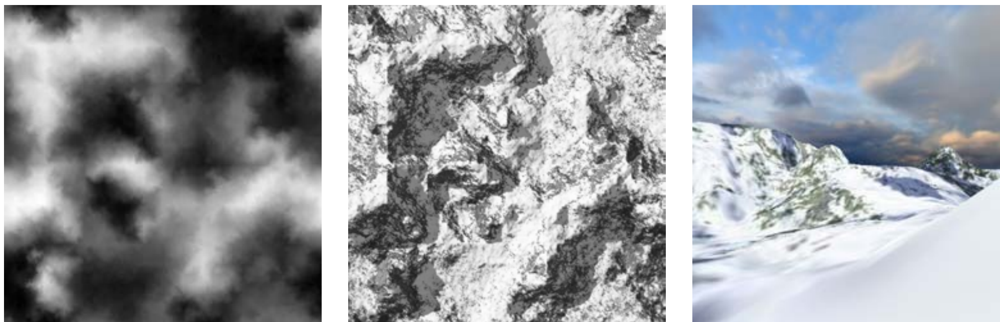
A `TerrainObject` konstruktora betölti a textúrafájlt és azt a képfájlt, amelynek szürkeségi szintjeit magasságként értelmezi. A `GetPoint()` tagfüggvény a magasságkép egy pixelét a háromdimenziós térbe transzformálja, figyelembe véve a terep háromdimenziós méreteit.

A terepen karakterek járnak, tárgyak épülhetnek rá, ami azt jelenti, hogy ezek alapját a terep magasságára kell állítani. A terep magasságát a világ egy (`x,y`) pontján a `Height()` függvény mondja meg:

```
//---------------------------------------------------------------
float TerrainObject::Height(float x, float y) { // x, y pont feletti magasság
//---------------------------------------------------------------
   x += wwidth/2; y += wlength/2;                // (0,0) a kép közepe
   x = x / wwidth * width; y = y / wlength * length;
   int X = (int)x, Y = (int)y;                   // rácspont
   if (X >= width-1 || X < 0 || Y >= length-1 || Y < 0) return 0.0;
   float h1 = height_field[Y * width + X] * wheight;
   float h2 = height_field[Y * width + X+1] * wheight;
   float h3 = height_field[(Y+1) * width + X] * wheight;
   float h4 = height_field[(Y+1) * width + X+1] * wheight;
```

```
   float xd = x - X, yd = y - Y;
   float hx1 = h1 + xd * (h2 - h1), hx2 = h3 + xd * (h4 - h3);
   return (hx1 + yd * (hx2 - hx1));              // bilineáris interpoláció
}
```

Ez a függvény először megkeresi azt a rácsot, amely tartalmazza a kapott (`x,y`) koordinátapárt, majd bilineáris interpolációval állítja elő a magasságértéket a négy legközelebbi rácspont magasságából. Megjegyezzük, hogy ez nem vezet pontosan ugyanarra az eredményre, mint amit a rajzolásnál használunk. A rajzolásnál ugyanis a rács négyszögét két háromszögre bontjuk, amelyek a magasságot belül lineárisan (és nem bilineárisan) interpolálják. Ha a háromszögek mérete nem túl nagy és a magasság nem változik rohamosan, akkor a közelítés tökéletesen elfogadható.



10.14. ábra. *Egy magasságmező magasságképe, textúrája és textúrázott megjelenítése*

A magasságmezőt úgy rajzolhatjuk fel, hogy az alaprács négyszögeit háromszögekre bontjuk, és a háromszögeket jelenítjük meg. Mivel egy rácssorban a két egymást követő háromszög éle közös, érdemes a sort háromszög szalagnak tekinteni.

```
//-----------------------------------------------------------------
void TerrainObject::DrawIt(Camera * camera) {
//-----------------------------------------------------------------
   for(int X = 0; X < width - 1; X++) {
      glBegin(GL_TRIANGLE_STRIP);         // háromszög szalag
      for(int Y = 0; Y < length; Y++) {
         glTexCoord2f((float)X/width, (float)Y/length);
         glVertex3fv(GetPoint(X, Y).GetArray());
         glTexCoord2f((float)(X+1)/width, (float)Y/length);
         glVertex3fv(GetPoint(X+1, Y).GetArray());
      }
      glEnd();
   }
}
```

A 10.14. ábrán egy magasságmező definícióját és a képét láthatjuk. Nagyméretű terepekből, különösen ha azt többször megismételjük, nagyon sok háromszög keletkezhet,

amely a rajzolást jelentős mértékben lelassíthatja. Egyrészt érdemes bevetni valamilyen láthatósági vágást, és csak azokat a háromszög szalagokat átadni az OpenGL-nek, amelyek a láthatósági gúlába esnek. A másik lehetőség a részletezettségi szintek alkalmazása, amely nagyobb háromszögekkel dolgozik a terep kis változású részein, valamint azokon, amelyek a szemtől távolabb helyezkednek el [56, 105].

10.7. A hegyivadász játék

A karakterek és terepek megismerése után egy újabb játék elkészítésébe vágjuk a fejszénket. Ez is lövöldözős játék lesz („a Barbie vásárolni megy" típusú játékoktól idegenkedünk), amelyben a hegyek között küzdünk meg egy szörnyhadsereggel.

10.15. ábra. *A hegyivadász játék egy pillanatfelvétele*

A szörnyek animált karakterek. Az avatár és a szörnyek a terepen mozoghatnak, és tüzérségi lövedékekkel lövöldözhetnek. A tüzérségi lövedékre a föld vonzóereje hat, a lövés lényegében egy ferde hajítás. A terepet egy magasságmezővel hozzuk létre, az égboltot pedig egy gömb belső felületére textúrázzuk fel. A hegyivadász játék objektumai a terep, az égbolt, a különböző fajtájú ellenségek, a tüzérségi lövedékek és a robbanások. A terepet a `Mountain`, az eget a `Sky`, a játékost a `Self`, a lövedékeket a `Bullet`, az ellenségeket pedig a típusuk szerint az `Ogro`, a `Morbo`, az `Alita` és a `Sod` osztályokkal definiáljuk. Az osztályok `GetType()` tagfüggvényei azonosítják a többiek számára az objektum típusát, amely a következő lehetőségek közül kerülhet ki:

```
enum GameObjectType {        // hegyivadász játék objektumtípusai
```

```
    SKY,                        // ég
    MOUNTAIN,                   // terep
    ENEMY,                      // ellenségek
    WEAPON,                     // fegyver
    AVATAR,                     // mi magunk
    EXPLOSION,                  // robbanás
    BULLET                      // lövedék
};
```

A következőkben ezeket az osztályokat egyenként vizsgáljuk. A robbanást leíró `Explosion` osztály az űrhajós játékban ismertetettel megegyezik (10.3.5. fejezet), így itt nem tárgyaljuk.

10.7.1. Az ég

Az eget az űrhöz hasonlóan valósítjuk meg, de most nem egy kockát teszünk a terep köré, hanem egy gömböt, és arra textúrázzuk az égről készített és gondosan előtorzított panorámaképet.

10.16. ábra. *Az ég textúrája és a félgömbre vetített képe*

10.7.2. A hegyvidék

A hegyvidéket a `TerrainObject` osztályból származtatjuk, az általános terep szolgáltatásaihoz nem is kell semmit sem hozzátenni. A terep magasságmezejét és textúráját a konstruktorban adjuk át a terepnek.

```
//===============================================================
class Mountain : public TerrainObject {
//===============================================================
public:
   Terrain() : TerrainObject(Vector(0, 0, 0),
```

```
                          "height.pcx", "terraintext.bmp", 10, 10, 1.5) { }
   int GetType() { return TERRAIN; }
};
```

10.7.3. Az ellenségek

10.17. ábra. *Az ellenségek: Ogro, Alita, Sod és Morbo*

Az ellenség karaktereket a Quake2 játékból kölcsönöztük[7]. A karakterek definícióját MD2 formátumú fájlokban vesszük át. Az MD2 formátumú karaktereket az `MD2Object` osztály szolgáltatásaival olvashatjuk be és animálhatjuk, ezért az ellenségek típusát ebből az osztályból származtatjuk.

```
//=================================================================
class Enemy : public MD2Object {
//=================================================================
protected:
   enum AI_State {                // gondolkodási állapot
      DONT_CARE, ATTACK_AVATAR, ESCAPE_FROM_AVATAR, CHASE_AVATAR, DYING
   } ai_state, new_ai_state;
   float speed;                   // haladási sebesség
   float scale;                   // az ellenség mérete
   float attack_dist, chase_dist; // támadási, üldözési  távolság
   float fear_dist, fear_cangle;  // menekülési távolság és szög
public:
   Enemy(Vector& pos0, char * md2_file, float scale0, float speed0)
      : MD2Object(pos0, md2_file, scale0) {
      ai_state = DONT_CARE;       // egyelőre semmi dolga
      scale = scale0; speed = 0.03 * speed0; anim_speed = speed0;
      attack_dist = chase_dist = fear_dist = fear_cangle = 0;
```

[7] http://www.quake.com

```
   }
   void AnimationOver() {          // az adott animációs ciklus vége
      if (ai_state == DYING) Member::KillIt(); // haldoklani csak egyszer
      MD2Object::AnimationOver();
      if (ai_state == ATTACK_AVATAR) Shoot();  // minden fázisban lő
   }

   void Shoot() {  // ha van fegyverünk, akkor tűz!
      if (children) { // van fegyverünk?
                     // a fegyver lövéskor a (30,10,10) ponton van
         Vector weapon(30, 10, 10);
         weapon = weapon.Rotate(Vector(0,0,1), head_angle) * scale;
         Vector dir(cos(head_angle * M_PI/180), sin(head_angle * M_PI/180),
                    Rand(0.2,0.1));    // véletlen függőleges irányzás
         root->Join(new Bullet(position + weapon, dir, this));
      }
   }

   void KillIt() { new_ai_state = DYING; } // megölték, haldoklani kezd
   void ControlIt(float dt);               // interakció, gondolkodás
   void InteractIt(GameObject * object);   // interakció egyetlen objektummal
};
```

Az ellenség konstruktora alaphelyzetbe állítja a mesterséges intelligencia állapotot, a karakter méretét (`scale`) és sebességet (`speed`), illetve a gondolkodást befolyásoló távolságokat és szögeket. Ezek közül az `attack_dist` azt a távolságot jelenti, amelyen belül az ellenség megáll, hogy tüzeljen, a `chase_dist` azt a minimális távolságot, ahonnan az ellenség észreveszi az avatárt, a `fear_cangle` az avatár nézeti iránya és az ellenség iránya közötti szög koszinuszának azt az értékét, amikor már az ellenség úgy gondolja, hogy az avatár feléje néz, így célszerű menekülésre fogni a dolgot. Hasonlóan a `fear_dist` azt a távolságot adja meg, amelyen belül az ellenség azt hiszi, hogy az avatár megláthatja őt.

Az ellenség felüldefiniálja az animációs fázis végét jelző, és alapértelmezésben ciklikus végrehajtást előíró `AnimationOver()` függvényt. Ennek célja kettős. Egyrészt, a haldoklás végén meg is kell halni, ezt a fázist nem lehet ciklikusan végrehajtani. A többi esetben azonban az alapértelmezés szerint ciklikusan ismételgetjük az animációs fázist. Másrészt a támadási fázis elején az ellenség egy golyót is kilő, amennyiben rendelkezik megfelelő fegyverrel. A lövést a `Shoot()` függvényben találjuk. A fegyver az ellenség gyermekobjektuma, amelynek közelítő helyéről engedjük el a lövedéket. Figyeljük meg, hogy ha egy ellenségnek nincs „gyermeke”, akkor nincs fegyvere, tehát a lövés műveletet kihagyjuk!

Az általános `KillIt()` tagfüggvényt ugyancsak felül kell bírálni, mert ha egy ellenséget eltalálunk, az nem szívódik fel rögtön, hanem először egy megfelelően megrendezett haldoklási fázisba kezd.

A `ControlIt()` tagfüggvény a párbeszéd kezdeményezéséért és a gondolkodásért felelős. A párbeszéd (`Interact()`) eredményeként kialakul egy új mesterséges intelligencia állapot (`new_ai_state`), amelybe átlépve az ellenség cselekedeteit és mozgás-

fázisát az új helyzethez igazítjuk. Ismét gondot kell fordítani arra, hogy az esetleges gyermekek, azaz a fegyver, kövesse az ellenség animációját, hiszen csak ekkor marad az ellenségünk kezében.

```
//-----------------------------------------------------------------
void Enemy::ControlIt(GameObject * object) { // interakció egy objektummal
//-----------------------------------------------------------------
   if (new_ai_state != DYING) Interact(root);  // párbeszéd a többi objektummal
   if (new_ai_state != ai_state) {
      switch (new_ai_state) {  // gondolkodási állapotból animációs ciklus
      case DONT_CARE:          // ha nem foglalkozik semmivel
         SetAnimationState(STAND_START, STAND_END); break;   // akkor lődörög
      case ATTACK_AVATAR:      // ha az avatárt támadja
         Shoot();                                            // akkor lő és
         SetAnimationState(ATTACK_START, ATTACK_END); break; //   támad
      case ESCAPE_FROM_AVATAR: // ha az avatár elől menekül
         SetAnimationState(RUNNING_START, RUNNING_END); break; // akkor fut
      case CHASE_AVATAR:       // ha az avatárt üldözi
         SetAnimationState(RUNNING_START, RUNNING_END); break; // akkor fut
      case DYING:              // ha éppen haldoklik
         speed = 0;                        // akkor innen már nem mozdul el
         float r = (float)rand()/RAND_MAX; // véletlenszerű haldoklási változat
         if (r < 0.3)      SetAnimationState(DYING1_START, DYING1_END);
         else if (r < 0.6) SetAnimationState(DYING2_START, DYING2_END);
         else              SetAnimationState(DYING3_START, DYING3_END);
      }

      // a fegyvert pontosan ugyanazokban a kulcskeretekben kell animálni
      if (children)
         ((MD2Object*)children)->SetAnimationState(anim_start, anim_end);
      ai_state = new_ai_state;
   }
}
```

Az ellenség helye és a mesterséges intelligencia állapota a többi játékobjektummal folytatott párbeszéd eredménye. A tereppel történő párbeszéd során az ellenség kideríti, hogy ott, ahol ő áll, milyen magas a terep, majd a talpát erre a szintre állítja. Ezért fog mindig a terep tetején járni. Az avatárral folytatott csevegés a relatív helyzet, azaz a távolság (`dist`) és azon szög tisztázásával kezdődik, amely kifejezi, hogy az ellenség mennyire áll éppen az avatár előtt, azaz az avatár milyen messze van a jó tüzelési helyzettől. Ha az avatár a tüzelési helyzethez közel van, és távolsága `fear_dist`-nél kisebb, akkor az ellenség menekül, és igyekszik minél távolabb kerülni (`ESCAPE_FROM_AVATAR` állapot). Ha viszont az avatár nincs tüzelési helyzetben, és elég közel van ahhoz, hogy az ellenség észrevegye (`chase_dist`), akkor az ellenség az avatárra ront (`CHASE_AVATAR` állapot). Ha a távolság még ennél is kisebb, akkor támadásba lendül, és fegyveréből lövéseket ad le (`ATTACK_AVATAR` állapot). Ha egyetlen feltétel sem áll fenn, akkor lődörög, és vár a megfelelő pillanatra (`DONT_CARE` állapot). Végül, ha az ellenség elkapja az avatárt, azaz ütközik vele, akkor puszta kézzel is megöli.

```
//-----------------------------------------------------------------
void Enemy::InteractIt(float dt) { // interakció, gondolkodás
//-----------------------------------------------------------------
   if (object->GetType( ) == MOUNTAIN) { // a terep megemeli a karaktert
      Mountain * terrain = (Mountain *)object;
      float terrain_height = terrain->Height(position.x, position.y);
      position.z = terrain_height + bounding_radius;
   }
   if (object->GetType( ) == AVATAR) {
      if (ai_state == DYING) return;     // ha már haldoklik, akkor nem érdekes
      Avatar * avatar = (Avatar *)object;
      Vector dr = avatar->position - position;// relatív helyzet
      float dist = dr.Length();              // távolság
      dr.Normalize();                        // ellenség->avatár irány
      Vector avatar_head = avatar->Head();   // avatár haladási iránya

      if (dr * avatar_head < -fear_cangle) { // ha az avatár célkeresztjében van
         if ( dist < fear_dist ) {           //    akkor pucolás
            velocity = dr * (-speed);        //    az avatártól elől
            new_ai_state = ESCAPE_FROM_AVATAR;
         }
      } else {                               // ha az avatár nem néz oda
         if (dist < attack_dist) {           // ha közel van az avatárhoz
            new_ai_state = ATTACK_AVATAR;    //    felkészül a támadásra és
            velocity = dr * speed / 10;      //    lassan megy az avatár felé
         } else if (dist < chase_dist) { // kicsi távolság és az avatár rá néz
            new_ai_state = CHASE_AVATAR;     //    akkor üldözőbe veszi
            velocity = dr * speed;           //    azaz az avatár felé indul
         } else {
            new_ai_state = DONT_CARE;        // semmi dolga
            velocity = Vector(0, 0, 0);      // egy helyben lődörög
         }
      }
      if (dist < bounding_radius + avatar->bounding_radius) // ha elkapta
         avatar->KillIt();                   // akkor agyoncsapja az avatárt
   }
}
```

Az `Alita` és a `Morbo` típusú ellenségek osztályát az általános `Enemy` osztályból származtatjuk úgy, hogy a viselkedésüket leíró paraméterek egy részét véletlenszerűen választjuk meg. A véletlen választásnak köszönhetően lesz az egyes ellenségeknek önálló, és a játékos számára kevésbé kiszámítható viselkedése. A véletlen számok előállításához a 10.3.5. fejezet `Rand()` függvényét használjuk, amelyet úgy paraméterezünk, hogy `Morbo` általában bátrabb, de kevésbé gyors és rosszabb látású legyen, mint `Alita`.

```
//=================================================================
class Alita : public Enemy {
//=================================================================
public:
   Alita(Vector& pos0) : Enemy(pos0, "alita.md2", 0.003, 2) {
      chase_dist = Rand(4, 1);
      fear_dist = Rand(3, 0.5); fear_cangle = Rand(0.8, 0.1);
   }
};
```

```
//===============================================================
class Morbo : public Enemy {
//===============================================================
public:
   Morbo(Vector& pos0) : Enemy(pos0, "morbo.md2", 0.003, 1.2) {
      chase_dist = Rand(3, 0.5);
      fear_dist = Rand(2, 0.5); fear_cangle = Rand(0.6, 0.1);
   }
};
```

Figyeljük meg, hogy mindkét esetben változatlanul hagytuk az `attack_dist` változó zérus kezdeti értékét! Ezek az ellenségek fegyverrel nem támadnak, csak az avatárt akarják megfogni és puszta kézzel szeretnének végezni vele.

A fegyveres ellenségekhez egy fegyver osztályt is fel kell vennünk, amit ugyancsak az `MD2Object` típusból származtatunk:

```
//===============================================================
class Weapon : public MD2Object {
//===============================================================
public:
   Weapon(char * md2_file, float scale, float speed0)
      : MD2Object(Vector(0,0,0), md2_file, scale) { anim_speed = speed0; }
   int GetType() { return WEAPON; }
};
```

Kétféle fegyveres ellenségünk van, `Ogro` és `Sod`. A fegyver az ellenség gyermekobjektuma, hiszen követi a mozgását, sőt a karakter animációját is (végig kézben marad). A két fegyveres közül `Sod` ügyesebb, `Ogro` meglehetősen lomha és ügyetlen.

```
//===============================================================
class Ogro : public Enemy {
//===============================================================
public:
   Ogro(Vector& pos0) : Enemy(pos0, "ogro.md2", 0.003, 1.1) {
      chase_dist = Rand(3, 1);  attack_dist = Rand(1, 0.5);
      fear_dist = Rand(2, 0.5); fear_cangle = Rand(0.7,0.2);
      AddChild(new Weapon("weapon.md2", 0.003, 1.1));
   }
};
//===============================================================
class Sod : public Enemy {
//===============================================================
public:
   Sod(Vector& pos0) : Enemy(pos0, "sodf8.md2", 0.003, 2) {
      chase_dist = Rand(4, 1);  attack_dist = Rand(1, 0.5);
      fear_dist = Rand(3, 0.5); fear_cangle = Rand(0.6, 0.2);
      AddChild(new Weapon("sodweapon.md2", 0.003, 2));
   }
};
```

10.7.4. A lövedék

A lövedék (`Bullet`) egy textúrázott gömb, amelyet fizikai animációval a ferde hajításnak megfelelően mozgatunk.

```
//===============================================================
class Bullet : public TexturedObject {
//===============================================================
   GameObject    * source;    // ki lőtte ki?
   GLUquadricObj * quadric;   // gömb (golyó)
 public:
    Bullet(Vector& pos0, Vector& v, GameObject * source0)
       : TexturedObject(pos0, "bullet.bmp") {
       velocity = v.UnitVector();
       acceleration = Vector(0, 0, -g);   // nehézségi gyorsulás
       source = source0;
       bounding_radius = 0.01;
       quadric = gluNewQuadric();
       gluQuadricTexture(quadric, GL_TRUE);
   }

   int GetType() { return BULLET; }

   void DrawIt(Camera * camera) {
        glTranslatef(position.x, position.y, position.z);
        gluSphere(quadric, bounding_radius, 8, 6);
   }

   bool CollideIt(GameObject * obj, float& hit_time, Vector& hit_point);
   void ControlIt(float dt);    // vezérlés
};
```

Az ütközésdetektálási eljárást újraértelmeztük, és a lövedék–terep párhoz egy külön algoritmust rendeltünk. A terepre ugyanis az alapértelmezésként használt befoglaló gömb módszer nyilván nem adna elfogadható eredményt.

```
//---------------------------------------------------------------
bool Bullet::CollideIt(GameObject * obj, float& hit_time, Vector& hit_point) {
//---------------------------------------------------------------
   if (obj->GetType() == MOUNTAIN) {      // lövedék-terep
      Mountain * terrain = (Mountain *)obj;
      float height = terrain->Height(position.x, position.y);
      if (height > position.z) {          // lövedék a terep alatt van?
         hit_time = 0;                    // akkor ütközik
         hit_point = Vector(position.x, position.y, height);
         return TRUE;
      }
      return FALSE;                       // a terep felett nem ütközik
   } else     // a többi objektumra az befoglaló gömbös alapértelmezés
      return GameObject::CollideIt(obj, hit_time, hit_point);
}
```

A vezérlés az ütközésfelismeréssel foglalkozik. Ha a lövedék eltalál valamit, akkor robbanás keletkezik. A találat az ellenségeket és az avatárt megöli, a terepet viszont nyilván nem:

```
//-----------------------------------------------------------------
void Bullet::ControlIt(float dt) {
//-----------------------------------------------------------------
   float hit_time = dt;                    // ütközés felismerés
   Vector hit_point;
   GameObject * hit_object = Collide(root, hit_time, hit_point, source);
   if (hit_object) {                       // ha volt ütközés
      KillIt();                            //    akkor a lövedék megsemmisül
      root->Join(new Explosion(hit_point));//    és robbanás keletkezik
      int hit_object_type = hit_object->GetType();
                                           // ha avatárt vagy ellenséget talál
      if (hit_object_type == AVATAR || hit_object_type == ENEMY)
         hit_object->KillIt();             //    akkor megöli
   }
}
```

10.7.5. Az avatár

A játék avatárobjektuma a terepen bóklászhat és lövöldözhet. A többi objektum közül csak a tereppel létesít közvetlen kapcsolatot, és a helyzetét a terep magasságának megfelelően állítja be.

```
//=================================================================
class Self : public Avatar {
//=================================================================
    Vector head;     // erre néz
    float  speed;    // haladási sebesség
 public:
    Self(Vector& pos0) : Avatar(pos0), head(1, 0, 0) {
       speed = 0; bounding_radius = 0.05;
    }
    int GetType() { return AVATAR; }
    Vector Head() { return head; }               // nézeti irány a kamerához
    void ControlIt(float dt) { Interact(root); } // párbeszéd

    void InteractIt(GameObject * object) {
       if (object->GetType() == MOUNTAIN) {      // a hegy megemeli
          Mountain * terrain = (Mountain *)object;
          float height = terrain->Height(position.x, position.y);
          position.z = height + bounding_radius; // magasságállítás
       }
    }
    void ProcessInput(Application * input) {
       speed = 0;
       if (input->GetKeyStatus('q')) speed =  0.1;  // előre lép
       if (input->GetKeyStatus('a')) speed = -0.1;  // hátra lép
       if (input->GetKeyStatus(' '))                // lő
          root->Join(new Bullet(position, head, this));
       Vector dir = Steering(input);                // kormányzás
       head += Vector(dir.x, dir.y, -dir.z) * 0.1;
       head.Normalize();                            // nézeti irány
       velocity = head * speed;                     // sebesség
    }
};
```

A `ControlIt()` és `InteractIt()` metódusok egyetlen feladata, hogy a terep aktuális magasságára emeljék az avatárt. A `ProcessInput()` az avatárt az űrhajóhoz hasonlatosan, de szigorúan a földön kormányozza. Az űrhajónál alkalmazott megoldáshoz képest a `Steering()` függvénytől kapott irány z-koordinátájának megváltoztattuk az előjelét. Erre azért van szükség, mert a repülőszerkezetek botkormánya a fel–le irányokat éppen ellentétesen értelmezi (a `Camera` osztályban az y-irányt tekintettük a nézeti iránynak a z-irányt pedig a függőleges iránynak).

10.7.6. A hegyivadász játék főosztálya

A hegyivadász játék objektumait az általános játékmotorból (`GameEngine`) származtatott `GroundShootGame` osztály `Init()` függvénye hozza létre, így az alkalmazásnak csak egy ilyen típusú objektumot kell felvennie:

```
//===============================================================
class GroundShootGame : public GameEngine {
//===============================================================
public:
   GroundShootGame(int window_width, int window_height)
    : GameEngine("Ground Shoot Game", window_width, window_height) { }

   void Init() {
      GameEngine::Init();
      world = new Mountain();                    // terep
      world->Join(new Sky());                    // égbolt
      world->Join(new  Ogro(Vector( 0, 0, 0)));  // Ogro
      world->Join(new  Ogro(Vector(-2,-2, 0)));  // még egy Ogro
      world->Join(new Morbo(Vector( 2, 2, 0)));  // Morbo
      world->Join(new Alita(Vector(-2, 2, 0)));  // Alita
      world->Join(new   Sod(Vector( 2,-2, 0)));  // Sod
                                                 // avatár
      avatar = new Self(Vector(0, -1, 0));
      world->Join(avatar);
   }
};

void Application::CreateApplication() { new GroundShootGame(500, 500); }
```

10.8. A teljesítmény növelése

A játékok valós idejű grafikus rendszerek, azaz a kellő játékélményhez másodpercenként legalább 20-szor kell lefényképezni a virtuális világot. Ezt a feltételt még a nagy teljesítményű grafikus kártyák mellett sem egyszerű kielégíteni, különösen, ha a virtuális világban sok és bonyolult objektumot szerepeltetünk. Bonyolult világokban a megjelenítés mellett az ütközésdetektálás számítása is kritikus. Ebben a fejezetben áttekintjük azokat az eljárásokat, amelyekkel a rendszer teljesítménye jelentősen növelhető.

10.8.1. Megjelenítési listák

A megjelenítés szűk keresztmetszete gyakran a CPU és a grafikus kártya közötti kommunikáció, így az itt átadott adatmennyiséget minimalizálni kell. Például a különálló háromszögek helyett hálókkal dolgozhatunk, ugyanis ezek a közösen birtokolt csúcspontokat csak egyszer adják meg.

Egy másik lehetőség, hogy a geometriát csak egyszer töltjük le az OpenGL-nek, ahol *megjelenítési listák*ban tároljuk (*display list*), majd az egyes rajzolási ciklusokban csak a transzformációs mátrixokat állítjuk át. Ez a lehetőség merev testek megjelenítésére alkalmazható. Példaképpen a terep (10.6. fejezet) `TerrainObject` osztályát alakítjuk át. A terep pontjait most már a konstruktorban átadjuk az OpenGL-nek, és megkérjük, hogy tárolja el azt egy megjelenítési listájába. A rajzolás során már csak erre a listára hivatkozunk:

```
//===============================================================
class TerrainObject : public TexturedObject {
//===============================================================
   int    width, length;            // a magasságmezőt leíró kép felbontása
   Byte * height_field;             // magasságmező kép
   float  wwidth, wlength, wheight; // az objektum méretei a virtuális világban
   int    display_list;             // megjelenítési lista azonosító
public:
   TerrainObject( Vector& pos0, char * height_file, char * texture_file,
                  float wwidth0, float wlength0, float wheight0)
      : TexturedObject(pos0, texture_file) {
      ImageFile height_image( height_file, width, length); // magasságmező fájl
      height_field = new Byte[ width * length ];
      for (int i = 0; i < width * length; i++)
          height_field[i] = height_image.Red(i);
      wwidth = wwidth0; wlength = wlength0; wheight = wheight0 / 256;

      display_list = glGenLists(1);        // megjelenítési lista létrehozása
      glNewList(display_list, GL_COMPILE); // átadjuk, most ne jelenítse meg
      for(int X = 0; X < width - 1; X++) {
         glBegin(GL_TRIANGLE_STRIP);       // háromszög szalag
         for(int Y = 0; Y < length; Y++) {
            glTexCoord2f((float)X/width, (float)Y/length);
            glVertex3fv(GetPoint(X, Y).GetArray());
            glTexCoord2f((float)(X+1)/width, (float)Y/length);
            glVertex3fv(GetPoint(X+1, Y).GetArray());
         }
         glEnd();
      }
      glEndList();                         // megjelenítési lista lezárása
   }
   void DrawIt(Camera* camera) { glCallList(display_list); } // lista rajzolás

   ~TerrainObject() {
       delete height_field;
       glDeleteLists(display_list, 1);
   }
};
```

A megjelenítési listákhoz először egy vagy több azonosítót kell kérnünk az OpenGL-től a `glGenLists()` függvénnyel. A függvény bemeneti paramétere a kért listák száma, visszatérési értéke pedig az első lista azonosítója (a többiek a rákövetkező egész számok). A lista építését a `glNewList()` függvénnyel indítjuk. Az OpenGL addig gyűjti a parancsokat, amíg a listát egy `glEndList()` hívással le nem zárjuk. A `glNewList()` paraméterei a lista azonosítója, valamint egy kapcsoló amellyel vezérelhetjük, hogy az OpenGL a listát csak megjegyezze (`GL_COMPILE`), vagy rögtön fel is rajzolja (`GL_COMPILE_AND_EXECUTE`). A tárolt listákban szereplő parancsokat a `glCallList()` függvénnyel hajthatjuk végre. Ha már nincs szükségünk a listákra, a `glDeleteLists(id, range)` hívással szabadíthatjuk fel a helyüket. A lefoglaláshoz hasonlóan ez a függvény egyszerre több listát is felszabadíthat, az `id` az első lista azonosítója, a `range` pedig a listák darabszáma.

Méréseink szerint a megjelenítési listás változat három–négyszer gyorsabban rajzolja fel a terepet, mint a 10.6. fejezetben szereplő megoldás.

10.8.2. Részletezettségi szintek

A játékobjektumokat néha közelről, máskor nagyon távolról látjuk. A közelről látható objektumokat sok háromszöggel kell leírni, hiszen ekkor mindenféle közelítés könnyen észrevehető. A távoli objektumokat, amelyek csak néhány pixelen tűnnek fel, nem érdemes ilyen részletesen felrajzolni, hiszen ez csak elvesztegetett idő lenne. A megoldást több részletezettségi szinten rendelkezésre álló modellek adják. Egy részletes modellből az egyszerűsített változatokat a 3.4.4. fejezet *progresszív háló* algoritmusával állíthatjuk elő. A megjelenítéskor először megvizsgáljuk az avatár és az objektum távolságát, majd a távolság szerint a megfelelő részletezettségű modellt vesszük elő, és ezt rajzoljuk fel. Zavaró lehet a részletezettségi szintek közötti hirtelen váltás, amit kiküszöbölhetünk, ha mindig két egymást követő részletezettségű modellel dolgozunk, és a megjelenített modellt a kettőből lineárisan interpoláljuk.

10.8.3. Láthatatlan részek eldobása

A képernyőn nem látszhatnak azok a tárgyak, amelyek kívül esnek a nézeti gúlán, vagy más tárgyak eltakarják előlünk. Ezeket a vizsgálatokat az OpenGL is elvégzi, de ha mi magunk ezt egyszerűbben el tudjuk dönteni, akkor nem érdemes ezzel is az OpenGL-t terhelni. Akkor tudjuk egyszerűbben eldönteni, hogy egy objektum biztosan nem látható, ha nem háromszögenként, hanem nagyobb egységekre (például objektumok befoglaló gömbjére) végezzük el ezt a vizsgálatot, vagy az előfeldolgozás során megfelelő térparticionáló adatstruktúrát építünk fel.

Azon tárgyak eldobását, amelyek biztosan kívül esnek a nézeti gúlán, *nézeti gúla vágás*nak (*view culling*) nevezzük. A `Camera` osztályának `InViewFrustrum()` függ-

vénye egy csavar ebben a gépezetben. Az általános megoldás az, hogy a befoglaló téglatestet vagy gömböt összevetjük a nézeti gúlával, és ha nincs közös részük, akkor a tárgyból biztosan nem látszik semmi. A vizsgálatot elvégezhetjük világ-, kamera-, vagy akár a képernyő-koordinátarendszerben is.

Azokat a tárgyakat sem érdemes az OpenGL-lel felrajzoltatni, amelyeket biztosan takar egy másik objektum. A biztosan takart tárgyak gyors eldobását *láthatósági vágás*nak nevezzük. Egy szobában állva általában csak a szobában lévő tárgyakat látjuk, a többi szoba berendezését nem, így a világot szobánként csoportosítva, a megjelenítendő tárgyak köre jelentősen csökkenthető. Az ajtók és ablakok sajnos megnehezítik az életünket, hiszen ezeken keresztül mégiscsak átlátunk az egyik helyiségből a másikba. Az egyik lehetséges megoldás, ha egy előfeldolgozási lépésben tisztázzuk, hogy egy szobából mely másik szobák tartalma látható. Ezt az információt hívják *potenciálisan látható halmaz*nak (*potential visible set* vagy *PVS*). A játék alatt pedig egy szoba mellett a potenciálisan látható halmazának tartalmát is rajzoljuk.

A másik lehetséges megoldás durván egyszerűsít, és egy *portálkép*pel, azaz a szomszédos szoba tartalmának egyszer elkészített képével váltja ki az ajtón keresztül látható látványt.

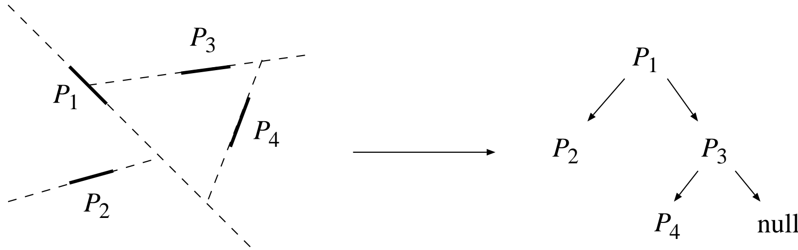
10.8.4. Térparticionáló adatstruktúrák

A nézeti gúla és a láthatósági vágás hatékonyabb megvalósítása a virtuális világ ismeretét tételezi fel, azaz az előfeldolgozás során megfelelő információkat kell szereznünk arról, hogy egy adott pontból mely objektumok láthatók. Ilyen információt kifejező adatstuktúrákkal már a 6.4. fejezetben foglalkoztunk, amikor a sugárkövetés gyorsítását tárgyaltuk. Ott ezeket *térparticionáló adatstruktúrák*nak neveztük el. A térparticionáló adatstruktúrák elsősorban statikus tárgyak kezelésére alkalmasak, ugyanis ha a bennük foglalt objektumok mozognak, akkor az adatszerkezetet újra fel kell építeni.

A térparticionáló adatstruktúrák hatékonyan használhatók a nézeti gúla vágásban, hiszen ha egy cella nincs a nézeti gúlában, akkor gyermek cellái, és a gyermek cellák objektumai sem lehetnek benne. Felhasználhatjuk őket a diszkrét ütközésdetektálásban is, ugyanis egy testet csak azokkal a tárgyakkal kell összevetni, amelyekkel azonos cellában van. Végül a térparticionáló adatstruktúrák alkalmasak a folytonos ütközésdetektáláshoz is, hiszen ez lényegében egy sugárkövetési feladat megoldását jelenti. A legközelebbi céltárgy megtalálásához azon cellákon kell végigmenni, amelyeket az ütköző tárgy pályája metsz, mégpedig a kiindulási ponttól mért távolság szerinti sorrendben. Ebben a fejezetben nem ismételjük meg a sugárkövetésnél megismert adatstruktúrákat, hanem egy olyan struktúrát mutatunk be, amelynek a sugárkövetésben kisebb a jelentősége, ám játékoknál hasznos eszköznek bizonyul. A módszer ugyanis egyszerre alkalmas a nézeti gúla vágásra, takarási algoritmusként is megállja a helyét, sőt az ütközésfelismeréshez is alkalmazható.

BSP-fa

A *BSP-fa* egy bináris térparticionáló fa (*Binary Space Partitioning tree*), amely minden szinten egy alkalmas síkkal a reprezentált térrészt két térrészre bontja. A BSP-fa egy közeli rokona a 6.4. fejezetben megismert kd-fa, amely koordinátatengelyekkel párhuzamos elválasztósíkokat használ. Jelen fejezetünk BSP-fája azonban a háromszögek síkját választja elválasztó síkként.



10.18. ábra. *BSP-fa*

A fa csomópontjaiban sokszögeket találunk, amelyek síkja választja szét a két gyermek által definiált térrészt. A fa levelei vagy üresek, vagy egyetlen sokszöget tartalmaznak.

A BSP-fát felépítő `BSPTree()` algoritmus egy S sokszöglistát kap. Az algoritmusban egy csomópontot N-nel, a csomóponthoz tartozó sokszöget $S(N)$-nel, a sokszög síkját, azaz a vágósíkot $P(N)$-nel, a csomópont két gyermekét pedig $L(N)$-nel illetve $R(N)$-nel jelöljük. Egy $\vec{r}$ pontot az $\vec{n}\cdot(\vec{r}-\vec{r}_0)$ skalárszorzat előjele alapján sorolunk az $\vec{n}$ normálvektorú és $\vec{r}_0$ helyvektorú sík pozitív és negatív tartományába.

```
BSPTree(S) {
    Egy új N csomópont létrehozása;   S(N) = S;
    if (S zérus vagy egy darab sokszöget tartalmaz) {
        P(N) = null; L(N) = null; R(N) = null;
    } else {
        P(N) = egy sokszög választása az S listából, S-ből P(N)-t töröljük;
        S+ = az S-beli sokszögekből a P(N) nem negatív félterébe lógók;
        S− = az S-beli sokszögekből a P(N) negatív félterébe lógók;
        L(N) = BSPTree(S+);
        R(N) = BSPTree(S−);
    }
    return N;
}
```

A hatékonyság érdekében a BSP-fát úgy érdemes felépíteni, hogy mélysége minimális legyen. A BSP-fa mélysége függ a sokszög kiválasztási stratégiájától, de ez a

függés nagyon bonyolult, ezért heurisztikus szabályokat kell alkalmazni [123], [44], [65].

Ha felépítettük a BSP-fát, azt több feladatban is felhasználhatjuk. Egyrészt a BSP-fa megoldja a takarási feladatot, ha olyan sorrendben járjuk be, hogy minden csomópontnál azon gyermek irányába lépünk később, amelyben a szempozíció is található. Ennek különösen az átlátszó tárgyaknál van jelentősége, hiszen ekkor a rendezésre szükség van, egyéb esetekben azonban a z-buffer is használható a takarás kezelésére.

A BSP-fa a nézeti gúla vágáshoz is hasznos segédeszköz. A fát a gyökértől kezdve járjuk be, és a vágósíkot minden csomópontra összevetjük a nézeti gúlával. Ha a vágósík nem metszi a nézeti gúlát, akkor a két gyermektartomány közül az egyik egyáltalán nem vehet részt a kép kialakításában, tehát ezen gyermektartománynak megfelelő ágban nem is kell folytatni a fa bejárását.

Végül a BSP-fa hatékonyan vethető be diszkrét és folytonos ütközésfelismerési eljárásokban is. Tekintsük először a diszkrét esetet! Az ütköző tárggyal belépünk a fa gyökerénél, és megvizsgáljuk, hogy az elválasztó sík metszi-e a tárgyat (vagy annak befoglaló térfogatát). Ha nem, akkor csak arra a térrészre kell rekurzívan alkalmazni ugyanezt az eljárást, amely a tárgyat tartalmazza. Ha metszi az elválasztó sík a tárgyat, akkor megvizsgáljuk, hogy ütközik-e a csomópont sokszögével, majd mindkét gyermekre folytatjuk a rekurziót. A leveleket elérve, ha a levél nem üres, akkor az ott található sokszöget összevetjük az ütköző tárggyal.

A *folytonos ütközésdetektálás* egyetlen pontszerű tárgyra lényegében a sugárkövetési feladat megoldását jelenti. Ismét a fa gyökerénél lépünk be a sugár egyenletével (kezdőpont és sebességvektor) és a sugárparaméter, azaz az idő minimális és maximális értékével (ütközésvizsgálatnál a minimum zérus, a maximum pedig az animációs keretidő). Kiszámítjuk a sugár és az elválasztó sík metszéspontját, és az itt érvényes sugárparamétert összevetjük a maximális és minimális idővel. Ha a metszés sugárparamétere nagyobb, mint a maximális idő, akkor csak abban a térrészben lehet a metszéspont, amelyben a sugár kiindulópontja van, tehát csak ezen gyermek irányába kell továbblépni. Ha a metszés sugárparamétere kisebb, mint a minimális idő, akkor csak a túloldali térrészt kell vizsgálni. Ha a minimális és maximális idő közrefogja a metszés sugárparaméterét, akkor meg kell vizsgálni, hogy az elválasztó sokszöget találtuk-e el, majd folytatni kell a vizsgálatot mindkét gyermek irányában. Mielőtt továbblépnénk, a maximális és minimális idő intervallumát a metszés paraméterével kettévágjuk, és a továbbiakban a két térrészben a részintervallumokkal dolgozunk.

11. fejezet

DirectX

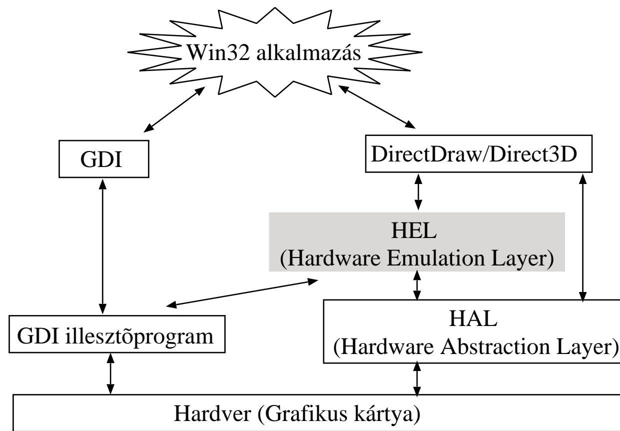
Könyvünk a háromdimenziós grafika megjelenítését OpenGL alapokra támaszkodva tárgyalja. Napjaink számítógépes játékainak többsége azonban egy másik API-t, a DirectX-et használja a színtér megjelenítésére. Könyvünkben ezért röviden ismertetjük a Microsoft eme üdvöskéjét is.

Az 1995-ben született *DirectX* 1.0 gyermekbetegségeit kinőve napjainkra már jól kiforrott, objektum-orientált API-vá érett. Csak Windows platformon használható, a 8.1-es verziótól és a Windows XP-től kezdve az operációs rendszer szerves része. Ebben a fejezetben és a példaprogramjainkban a jelenleg legfrissebb, 9.0-ás verziót használjuk. Számos programozási nyelv (C++, C#, Visual Basic, Delphi, SmallTalk) támogatja a DirectX API programozását.

A DirectX API objektumai az ActiveX technológiát megalapozó *COM* (*Component Object Model*) [107] objektumok. Funkcionalitásukat egy vagy több interfészen keresztül érhetjük el. Implementációjuk leggyakrabban egy C++-os virtuális metódusokkal rendelkező osztály. Az osztálynak egyrészt meg kell felelni egy bináris szabványnak, másrészt néhány kötelező virtuális metódussal kell rendelkeznie. A csak virtuális függvényeket tartalmazó osztálytípusokat interfészeknek hívjuk (például IDirect3D9). Az objektumos szerkezetből kifolyólag a DirectX-et C-ben nem lehet programozni.

A DirectX valójában különböző feladatokra kihegyezett csomagok gyűjteménye. (DirectDraw, Direct3D, DirectMusic, DirectSound, DirectPlay, DirectInput, DirectSetup). A számunkra fontos 2D megjelenítéssel a DirectDraw, 3D megjelenítéssel pedig a Direct3D foglalkozik. Mielőtt a Direct3D programozását elkezdenénk, a 11.1. ábrán érdemes tanulmányoznunk egy grafikus gyorsító kártyát használó Windows alkalmazás működésének szerkezetét.

A Windows *GDI* (*Graphics Device Interface*) [144] egy olyan absztrakciós szintet definiáló rendszerkomponens, amelynek segítségével az alkalmazások a képernyőre tudnak rajzolni. A GDI-t sajnos nem grafikus és multimédia célokra, hanem üzleti alkalmazásokhoz (szövegszerkesztő, táblázatkezelő) fejlesztették ki. A GDI filozófia



11.1. ábra. *A GDI és a DirectX integrációja*

szerint a keletkező kép a rendszermemóriába, és nem a videokártyába kerül, továbbá a GDI felület nincs felkészítve bővítésre, a videokártyák által támogatott új funkciók befogadására. Játékokhoz és multimédiás alkalmazásokhoz (videózás) a lassúsága miatt sem alkalmas.

A megváltóként beharangozott DirectDraw már a videokártya szolgáltatásaira épít. A DirectDraw utasítások mindenféle feldolgozás nélkül, közvetlenül a kártya utasításait használják. Ezért a rajzolás nagyságrendekkel gyorsabb, mint a GDI esetén.

A *HAL* a hardver absztrakciós réteg (*Hardware Abstraction Layer*) neve, amelyet a videokártya gyártók a csatolóprogramban (*driver*) valósítanak meg. A kompatibilitás érdekében minden kártyagyártó kötelezően ugyanazt a HAL felületet mutatja. A HAL része egy olyan metódus, amellyel lekérdezhető, hogy melyek a kártya által hardveresen gyorsított rajzolási funkciók. Ha egy funkció hiányzik (például a mátrixtranszformációkat a kártya képtelen végrehajtani), akkor azt a DirectX *HEL* emulációs rétege (*Hardware Emulation Layer*) fogja elvégezni. Természetesen ez lelassítja a programot, de a sebességcsökkenéstől eltekintve az alkalmazás ebből semmit sem vesz észre.

A DirectX-hez tartozik egy szabványos segédeszköz csomag, a *D3DX*, amely a programozást megkönnyítő hasznos rutinokat tartalmaz (például Mesh objektum létrehozása, transzformációs mátrix beállítása, textúrázás). A függvények névkonvenciója a D3DX API esetén a D3DX előtag (például `D3DXMatrixInverse()`). A Direct3D API esetén nincs névkonvenció, viszont függvényekről egyáltalán nem is beszélhetünk. Csak a COM objektumok metódusait lehet hívni. Egyetlen kivétel ez alól a `Direct3DCreate9()` függvény.

A DirectX számára a geometria alapelemeket csúcsokkal kell megadni, amelyeket

egy tömbbe szervezünk. Ezt a struktúrát nevezzük *csúcsbuffer*nek (*vertex buffer*). A grafikus alkalmazásunk feladata általában az, hogy az inicializálás során feltöltsön egy csúcsbuffert. A megjelenítés (`Render()`) alkalmával a csúcsbuffer címének megadásával a DirectX automatikusan kirajzolja a csúcsok által definiált geometriát.

11.1. Program: HelloDirectX alkalmazás

A 2.5.1. fejezethez hasonlóan el fogunk készíteni egy minimális grafikus alkalmazást, amely egy kocka kirajzolásához a DirectX alrendszert használja. Ha visszatekintünk az OpenGL fejezetre és összehasonlítjuk az ott szereplő OpenGL hívásokat a DirectX hívásokkal, akkor sok hasonlóságot fogunk találni.

A Direct3D programozásához a korábban készített HelloWindows alkalmazást fogjuk továbbfejleszteni. Első lépésben fel kell venni a `d3d9.h` és `d3dx9.h` fejléc (*header*) fájlokat a kódba. A hozzájuk tartozó `d3d9.lib` és `d3dx9.lib` könyvtár fájlokat pedig hozzá kell szerkeszteni a programhoz. A DirectX 9.0-ás verziójának inicializálása a következőképpen történik:

```
#include <d3d9.h>
#include <d3dx9.h>

//-----------------------------------------------------------------
void Application::Init(void) {
//-----------------------------------------------------------------
  // 1. Windows inicializálás
  MyRegisterClass(hInstance);
  if (!MyCreateInstance(hInstance, nCmdShow)) return;

  // 2. lekérdezzük a D3D interfészt
  g_pD3D = Direct3DCreate9(D3D_SDK_VERSION);
  if (g_pD3D == NULL) return;

  // 3. lekérdezzük a képernyőmódot, segítségével feltöltjük d3dpp-t
  D3DDISPLAYMODE d3ddm;
  if (FAILED(g_pD3D->GetAdapterDisplayMode(D3DADAPTER_DEFAULT,&d3ddm)))
      return;
  D3DPRESENT_PARAMETERS d3dpp;              // paraméterek feltöltése
  ZeroMemory(&d3dpp, sizeof(d3dpp));        // struktúra törlése
  d3dpp.Windowed    = TRUE;                 // nem teljes képernyő
  d3dpp.SwapEffect  = D3DSWAPEFFECT_FLIP;   // első-hátsó bufferváltás
  d3dpp.BackBufferFormat = d3ddm.Format;    // színbuffer, RGB vagy RGBA
  d3dpp.EnableAutoDepthStencil= TRUE;       // z-buffer bekapcsol
  d3dpp.AutoDepthStencilFormat= D3DFMT_D16; // z-buffer szómérete

  // 4. elkészítjük a D3D eszközt
  if (FAILED(g_pD3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, g_hWnd,
              D3DCREATE_HARDWARE_VERTEXPROCESSING, d3dpp, &g_pd3dDevice)))
      return;
}
```

OpenGL esetén az `Init()` metódus feladata egy OpenGL kontextus definiálása volt. Most egy Direct3D eszköz elkészítése a cél. Ehhez egyrészt az operációs rendszertől a `Direct3DCreate9()` függvény segítségével kell egy mutatót igényelni a Direct3D interfészhez. Ez az interfész a hardvert, azaz a grafikus kártyát jelképezi. Az interfész `CreateDevice()` metódusával egy Direct3D eszközt hozhatunk létre. Az elkészítéshez szükség van még egy `D3DPRESENT_PARAMETERS` struktúrára, amelyet gondosan fel kell paraméterezni. A `Windowed` mező azt mondja meg, hogy az alkalmazás teljes képernyős módban vagy egy ablakban fut majd. Ha a `SwapEffect` mezőnek `D3DSWAPEFFECT_FLIP` értéket adunk, akkor az első és hátsó színbuffer szerepét képkockánként váltogatni kell. Az `EnableAutoDepthStencil` kapcsolja be a *z-buffer* és a *stencil buffer* használatát, amelyben egy pixelhez a `D3DFMT_D16` vagy `D3DFMT_D32` konstansokkal lehet 16 vagy 32 bites adatot rendelni.

A `CreateDevice()` metódus negyedik paramétere azt jelöli ki, hogy szoftveres (`D3DCREATE_SOFTWARE_VERTEXPROCESSING`) vagy ha lehetőség van rá hardveres (`D3DCREATE_HARDWARE_VERTEXPROCESSING`) rajzolást használunk. Általában azt mondhatjuk, hogy a hardveres megjelenítés hatékonyabb. Egy gyors számítógépen a rendszermemóriában végzett szoftveres képszintézis azonban néha gyorsabb lehet. Ennek oka, hogy könyvünk írásakor a processzorok 3 GHz körüli működési frekvenciájához képest a grafikus kártyák majdnem egy nagyságrenddel kisebb, kb. 500 MHz körüli órajel frekvencián üzemelnek.

A `BackBufferFormat` mezőben állítjuk be az OpenGL kapcsán megismert *PixelFormat* információkat. A 32 bites színkódolást megvalósító `D3DFMT_A8R8G8B8`, a 16 bites `D3DFMT_A1R5G5B5`, illetve indexelt színmód esetén a `D3DFMT_P8` a leggyakrabban használt értékek. Az éppen aktuális értéket a `GetAdapterDisplayMode()` metódussal kapott struktúra `Format` mezője adja meg.

Az alkalmazás leállításakor az `Exit()` metódusban felszabadítjuk a csúcs buffert (*vertex buffer*), a Direct3D eszközt, és a Direct3D interfészt.

```
//-----------------------------------------------------------------
void Application::Exit(void) {
//-----------------------------------------------------------------
    if (g_pd3dDevice != NULL)   g_pd3dDevice->Release();
    if (g_pD3D != NULL)         g_pD3D->Release();
}
```

A színteret a `Render()` metódus rajzolja ki. Az OpenGL példához hasonlóan egy egységkockát teszünk ki a képernyőre.

```
//-----------------------------------------------------------------
void Application::Render(void) {
//-----------------------------------------------------------------
    // 1. DirectX állapot-attributumok beállítása
    g_pd3dDevice->Clear(0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
                        D3DCOLOR_XRGB(0,0,255), 1.0, 0);
```

```
// megvilágítás számításának bekapcsolása
g_pd3dDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
// z-buffer bekapcsolása
g_pd3dDevice->SetRenderState(D3DRS_ZENABLE, TRUE);
// színezési stílus
g_pd3dDevice->SetRenderState(D3DRS_FILLMODE, D3DFILL_SOLID);
//hátsólap eldobás
g_pd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
 // kikapcsolja a programozott árnyalást
g_pd3dDevice->SetVertexShader(NULL);
// alapértelmezett csúcspont árnyaló
g_pd3dDevice->SetFVF(MYVERTEX::FVF);
```

A képernyőt a `Clear()` metódus törli, amelynek egyrészt előírjuk, hogy a színbuffert (`D3DCLEAR_TARGET`) és a *z-buffert* (`D3DCLEAR_ZBUFFER`) is törölje, valamint megadjuk a háttér színt (`D3DCOLOR_XRGB`(0,0,255)), illetve a mélységértéket (1.0).

A megjelenítési attribútumokat a `SetRenderState()` metódus állítja be. A megvilágítás a `D3DRS_LIGHTING`, a z-buffer használata pedig a `D3DRS_ZENABLE` paraméterrel állítható. A triviális hátsólap eldobás működéséért a `D3DRS_CULLMODE` attribútum felelős. A `D3DCULL_CW` az óramutató járásával megegyező (*Clock-Wise*) csúcspont sorrendű háromszögeket fogja eldobni. Az alapértelmezett `D3DCULL_CCW` pedig azokat, amelyeknek sorrendje ezzel ellentétes (*Counter Clock-Wise*). Vigyázat! Ez azért van így, mert a DirectX — ellentétben az OpenGL-lel — balkezes koordinátarendszert használ. A `D3DCULL_NONE` kikapcsolja a hátsólap eldobást. A háromszög kirajzolása esetén a `D3DRS_FILLMODE` állapot mondja meg, hogy azt hogyan kell megjeleníteni. `D3DFILL_POINT` esetén csak a csúcspontokat, `D3DFILL_WIREFRAME` esetén csak az éleket rajzolja ki, amellyel így drótváz ábra készíthető. A példánkban használt `D3DFILL_SOLID` azt jelenti, hogy rajzoláskor teljesen kitöltjük a háromszöget.

A megvilágítási viszonyok leírása a következőképpen történik:

```
// 2. a fényviszonyok beállítása. Először a globális ambiens szín
g_pd3dDevice->SetRenderState(D3DRS_AMBIENT, D3DCOLOR_XRGB(55, 55, 55));

D3DLIGHT9 light;
light.Type      = D3DLIGHT_DIRECTIONAL;             // irány-fényforrás
light.Diffuse   = D3DXCOLOR(0.6, 0.6, 0.6, 1.0);
light.Ambient   = D3DXCOLOR(0.2, 0.2, 0.2, 1.0);
light.Range     = sqrtf(FLT_MAX);
light.Direction = D3DXVECTOR3(-5.0, -5.0, -5.0);    // irányvektor
g_pd3dDevice->SetLight(0, &light);
g_pd3dDevice->LightEnable(0, TRUE);
```

A `D3DLIGHT9` osztállyal lehetőség nyílik irány- (`D3DLIGHT_DIRECTIONAL`), és pontszerű (`D3DLIGHT_POINT`) fényforrás, illetve szpotlámpa (`D3DLIGHT_SPOT`) definiálására. A `Range` mezőnek csak a pontszerű fényforrásnál és a szpotlámpáknál van szerepe. A mező egy olyan távolságot definiál, amelynél távolabb a fényforrásnak már

nincs hatása. A fény irányát a `Direction` mező adja, amelyet nem kötelező normalizálni. A Direct3D interfész `SetLight()` metódusával 0-ás indexűként vesszük fel a fényforrást, amelyet a `LightEnable()` metódussal kapcsolgatni.

Ezek után a kamera beállítása következik:

```
    // 3. kamera beállítás: projekciós mátrix
    RECT rect;
    GetClientRect(g_hWnd, &rect);  // átméretezés miatt dinamikus
    float width     = rect.right - rect.left;
    float height    = rect.bottom - rect.top;
    float aspect    = (height == 0) ? width : width / height;

    D3DXMATRIXA16 matProj;
    D3DXMatrixPerspectiveFovLH(&matProj,(45.0/180)*M_PI,aspect,1,100);
    g_pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj);

    // 4. kamera beállítás: modellnézeti mátrix (balkezes koordináta-rendszer)
    D3DXMATRIXA16 matView;
    D3DXMatrixLookAtLH(&matView,
                       &D3DXVECTOR3(2.0, 3.0, 4.0),  // szem pozíció
                       &D3DXVECTOR3(0.0, 0.0, 0.0),  // nézett pont
                       &D3DXVECTOR3(0.0, 1.0, 0.0)); // felfelé irány
    g_pd3dDevice->SetTransform(D3DTS_VIEW, &matView);
```

A kamerát is ebben a függvényben kell beállítani. A `SetTransform()` metódussal `D3DTS_VIEW` esetén a nézeti mátrixot, `D3DTS_PROJECTION` esetén a projekciós mátrixot, `D3DTS_WORLD` esetén pedig a modell mátrixot módosítjuk.

A továbblépés előtt definiálni kell, hogy egy csúcshoz milyen attribútumok tartoznak. Természetesen hozzátartozik egy X, egy Y és egy Z attribútum (`D3DFVF_XYZ`), de hozzátartozhat még normálvektor (`D3DFVF_NORMAL`), szín (`D3DFVF_DIFFUSE`, illetve `D3DFVF_SPECULAR`), nyolc különböző textúrakoordináta (`D3DFVF_TEX0`) stb. Ezeket a jellemzőket egy úgynevezett *FVF* (*Flexible Vertex Format*) típus írja le. A mi esetünkben, mivel a megvilágítás miatt normálvektorra mindenképp szükség van, az FVF a következő formát ölti:

```
struct MYVERTEX {                // saját csúcspont típus
    D3DXVECTOR3 position;        // a pozíció
    D3DXVECTOR3 normal;          // a felület normálisa az adott pontban

    static const DWORD FVF;      // Flexible Vertex Format a SetFVF()-hez
};
const DWORD MYVERTEX::FVF = D3DFVF_XYZ | D3DFVF_NORMAL;
```

Egyszerű példánkban — redundáns módon — a kocka minden oldalát két háromszöggel, azaz hat csúccsal adjuk meg. A kocka rajzolása tehát a következőképpen néz ki:

```
    // 5. színtér felépítése
    const D3DCOLORVALUE RedSurface   = {1, 0, 0, 1};
    const D3DCOLORVALUE GreenSurface = {0, 1, 0, 1};
    const D3DCOLORVALUE BlueSurface  = {0, 0, 1, 1};

    g_pd3dDevice->BeginScene();
    D3DXMATRIXA16 matWorld;
    D3DXMatrixIdentity(&matWorld);
    g_pd3dDevice->SetTransform(D3DTS_WORLD, &matWorld );

    MYVERTEX vd[6];     // 2 háromszög = 6 csúcs
    D3DMATERIAL9 mtrl; ZeroMemory(&mtrl, sizeof(D3DMATERIAL9));
    D3DXVECTOR3 v[8] = {    // a csúcspontok
        D3DXVECTOR3( 0.5, 0.5, 0.5), D3DXVECTOR3(-0.5, 0.5, 0.5),
        D3DXVECTOR3(-0.5,-0.5, 0.5), D3DXVECTOR3( 0.5,-0.5, 0.5),
        D3DXVECTOR3( 0.5, 0.5,-0.5), D3DXVECTOR3(-0.5, 0.5,-0.5),
        D3DXVECTOR3(-0.5,-0.5,-0.5), D3DXVECTOR3( 0.5,-0.5,-0.5)};

    mtrl.Ambient = mtrl.Diffuse = RedSurface;
    g_pd3dDevice->SetMaterial(&mtrl);
    D3DXVECTOR3 frontFace[6] = {v[0],v[1],v[2],v[0],v[2],v[3]}; // előlap
    for (int i = 0; i < 6; i++) vd[i].normal = D3DXVECTOR3(0.0, 0.0, 1.0);
    for (int i = 0; i < 6; i++) vd[i].position  = frontFace[i];
    g_pd3dDevice->DrawPrimitiveUP(D3DPT_TRIANGLELIST,2,vd,sizeof(MYVERTEX));

    D3DXVECTOR3 backFace[6] = {v[4],v[5],v[6],v[4],v[6],v[7]}; // hátlap
    for (int i = 0; i < 6; i++) vd[i].normal = D3DXVECTOR3(0.0, 0.0, -1.0);
    for (int i = 0; i < 6; i++) vd[i].position = backFace[i];
    g_pd3dDevice->DrawPrimitiveUP(D3DPT_TRIANGLELIST,2,vd,sizeof(MYVERTEX));

    mtrl.Ambient = mtrl.Diffuse = GreenSurface;
    g_pd3dDevice->SetMaterial(&mtrl);
    D3DXVECTOR3 topFace[6] = {v[0],v[4],v[5],v[0],v[5],v[1]}; // tetőlap
    for (int i = 0; i < 6; i++) vd[i].normal = D3DXVECTOR3(0.0, 1.0, 0.0);
    for (int i = 0; i < 6; i++) vd[i].position = topFace[i];
    g_pd3dDevice->DrawPrimitiveUP(D3DPT_TRIANGLELIST,2,vd,sizeof(MYVERTEX));

    D3DXVECTOR3 bottomFace[6] = {v[3],v[7],v[6],v[3],v[6],v[2]}; // padlólap
    for (int i = 0; i < 6; i++) vd[i].normal = D3DXVECTOR3(0.0, -1.0, 0.0);
    for (int i = 0; i < 6; i++) vd[i].position = bottomFace[i];
    g_pd3dDevice->DrawPrimitiveUP(D3DPT_TRIANGLELIST,2,vd,sizeof(MYVERTEX));

    mtrl.Ambient = mtrl.Diffuse = BlueSurface;
    g_pd3dDevice->SetMaterial(&mtrl);
    D3DXVECTOR3 leftFace[6] = {v[0],v[3],v[7],v[0],v[7],v[4]}; // bal oldal
    for (int i = 0; i < 6; i++) vd[i].normal = D3DXVECTOR3(1.0, 0.0, 0.0);
    for (int i = 0; i < 6; i++) vd[i].position = leftFace[i];
    g_pd3dDevice->DrawPrimitiveUP(D3DPT_TRIANGLELIST,2,vd,sizeof(MYVERTEX));

    D3DXVECTOR3 rightFace[6] = {v[1],v[2],v[6],v[1],v[6],v[5]}; // jobb oldal
    for (int i = 0; i < 6; i++) vd[i].normal = D3DXVECTOR3(-1.0, 0.0, 0.0);
    for (int i = 0; i < 6; i++) vd[i].position  = leftFace[i];
    g_pd3dDevice->DrawPrimitiveUP(D3DPT_TRIANGLELIST,2,vd,sizeof(MYVERTEX));

    g_pd3dDevice->EndScene();
    g_pd3dDevice->Present(NULL, NULL, NULL, NULL);  // buffer csere
} // Render() függvény vége
```

A megjelenítés nagyon hasonlít az OpenGL működéséhez. A rajzolási rutinok előtt kötelezően szerepel a `BeginScene()`, utána pedig az `EndScene()`. A háromszögeket anyagonként csoportosítva rajzoljuk ki.

A `DrawPrimitiveUP()` metódus első paramétere mondja meg, hogy a csúcspontokat hogyan kell kezelni. `D3DPT_TRIANGLELIST` esetén minden egymást követő 3 csúcs tekinthető egy háromszögnek. Megadható még a `D3DPT_TRIANGLESTRIP` vagy `D3DPT_TRIANGLEFAN` is. A második paraméter a rajzolandó primitívek számát adja meg, a harmadik pedig a csúcspontokat tartalmazó bufferre mutat. A negyedik paraméter közli a DirectX-szel, hogy 1 csúcsponthoz hány bájt adat tartozik.

A kirajzolás végén a `Present()` mutatja meg a képet, azaz dupla bufferelésnél megcseréli az első és a hátsó színbuffer tartalmát.

Egy API programozása esetén könnyen előfordulhat, hogy hibásan működik a program. Ilyenkor válik fontossá a hiba okának felderítése. A DirectX utasítások hibakódokkal térnek vissza, amelyek a `D3DXGetErrorString()` függvény segítségével alakíthatók szöveges üzenetté.

11.2. Program: VRML színtér megjelenítése

Az 5.3.3. fejezetben az OpenVRML csomag segítségével egy VRML fájlt olvastunk be, majd elkészítettük a memóriában a színtérnek megfelelő háromdimenziós adatszerkezetet. Ebben a fejezetben megvalósítjuk a színteret megjelenítő programot is. Közben a DirectX újabb részleteit is megismerjük.

Az adatok átadásához egy csúcsbuffert (*vertexbuffer*) készítünk. A gyorsabb rajzolás érdekében a háromszögeket anyaguk alapján most is csoportokba szervezzük. Először is megszámoljuk (`nPatchesPerMaterial`), hogy egy anyaghoz hány háromszög tartozik. Erre a csúcsbuffer (`g_pVB`) felhasználásakor, a rajzolásnál lesz szükség.

```
IDirect3DVertexBuffer9* g_pVB               = NULL;
int*                    nPatchesPerMaterial = NULL;
```

Az `Init()` metódusban állítjuk be a DirectX megjelenítési állapotait és a megvilágítási viszonyokat, valamint feltöltjük a csúcsbuffert.

```
//-----------------------------------------------------------------
void VRMLViewerDX::Init(void) {
//-----------------------------------------------------------------
    ...
    // megszámoljuk hogy egy anyaghoz hány háromszög tartozik
    nPatchesPerMaterial = new int [gMaterials.size()];
    for (k = 0; k < gMaterials.size(); k++)
        nPatchesPerMaterial[k] = 0;
    for (k = 0; k < gPatches.size(); k++)
        nPatchesPerMaterial[gPatches[k].matIndex]++;
```

Ezek után már nincs akadálya, hogy a grafikus kártya memóriájában elkészítsük a csúcsbuffert. Az `Init()` befejező része a következőképpen alakul:

```
    // vertex buffer létrehozása
    if (FAILED(g_pd3dDevice->CreateVertexBuffer(
            3*gPatches.size()*sizeof(MYVERTEX), // méret bájtban
            D3DUSAGE_WRITEONLY,              // csak írásra
            D3DFVF_CUSTOMVERTEX,             // MYVERTEX::FVF megadásához
            D3DPOOL_MANAGED,                 // melyik memóriában tárolja
            &g_pVB, NULL)))                  // vertex buffer
        return;

    MYVERTEX* pVertices;
    if (FAILED(g_pVB->Lock( 0, 0, (void**)&pVertices, 0))) return;

    D3DXMATRIXA16 matLeftHanded; // DirectX balkezes koordináta-rsz.
    D3DXMatrixScaling(&matLeftHanded, 1.0, 1.0, -1.0);
    int indVertices = 0;
    for (k = 0; k < gMaterials.size(); k++) {    // anyagonként csoportosítva
        for (int i = 0; i < gPatches.size(); i++) { // háromszögekre
            short matInd = gPatches[i].matIndex;
            if (matInd != k)  continue;           // ha az anyag nem megfelelő

            Patch& p = gPatches[i];
            D3DXVec3TransformCoord(&pVertices[indVertices].position,
                &D3DXVECTOR3(p.a->x,p.a->y,p.a->z), &matLeftHanded);
            D3DXVec3TransformCoord(&pVertices[indVertices].normal,
                &D3DXVECTOR3(p.normal.x,p.normal.y,p.normal.z), &matLeftHanded);

            D3DXVec3TransformCoord(&pVertices[indVertices+2].position,
                &D3DXVECTOR3(p.b->x,p.b->y,p.b->z), &matLeftHanded);
            D3DXVec3TransformCoord(&pVertices[indVertices+2].normal,
                &D3DXVECTOR3(p.normal.x,p.normal.y,p.normal.z), &matLeftHanded);

            D3DXVec3TransformCoord(&pVertices[indVertices+1].position,
                &D3DXVECTOR3(p.c->x,p.c->y,p.c->z), &matLeftHanded);
            D3DXVec3TransformCoord(&pVertices[indVertices+1].normal,
                &D3DXVECTOR3(p.normal.x,p.normal.y,p.normal.z), &matLeftHanded);

            indVertices += 3;
        }
    }
    g_pVB->Unlock();

    g_pd3dDevice->SetStreamSource(0, g_pVB, 0, sizeof(MYVERTEX));
}
```

A Direct3D interfész `CreateVertexBuffer()` metódusával lehet egy csúcsbuffert elkészíteni. A lefoglalt memória méretét bájtban kell megadni.

A `D3DUSAGE_WRITEONLY` paraméterrel azt állítjuk be, hogy a csúcsbuffert csak írni szeretnénk, olvasni nem fogunk belőle. A gyorsabb futás érdekében adjuk ezt meg, hiszen a videokártya memóriájának olvasása nem olcsó mulatság, mert az egész buffert a buszon keresztül be kellene másolni a központi memóriába. A buffer elkészítésekor megadhatjuk, hogy azt a rendszer memóriájában (`D3DPOOL_SYSTEMMEM`) vagy a

grafikus kártya memóriájában (`D3DPOOL_MANAGED`) szeretnénk-e tárolni. Írás előtt a memóriaterületet a `Lock()` függvénnyel le kell zárni, majd az `Unlock()` metódussal fel kell szabadítani.

Mivel a VRML-től eltérően a Direct3D balkezes koordinátarendszert használ, a színtér háromszögei némi előfeldolgozást igényelnek. Egyrészt a Z koordinátájukat invertálni kell, amelyet a `D3DXVec3TransformCoord()` függvény végez el. Azonban ettől a körüljárási irány megfordul, amit úgy javítunk, hogy a háromszög csúcsait nem a 0., 1., 2. sorrendben, hanem a 0., 2., 1. sorrendben adjuk meg.

A buffer feltöltése után a `SetStreamSource()` metódus mondja meg a Direct3D-nek, hogy a továbbiakban a rajzoláshoz ezt a csúcsbuffert használja.

A színteret a `Render()` metódus rajzolja ki:

```
//-----------------------------------------------------------------
void VRMLViewerDX::Render(void) {
//-----------------------------------------------------------------
    ...
    // a színtér kirajzolása
    D3DMATERIAL9 mtrl;
    ZeroMemory(&mtrl, sizeof(D3DMATERIAL9));
    int startVertexIndex = 0;
    for (k = 0; k < gMaterials.size(); k++) {
        mtrl.Ambient.r = gMaterials[k].ambientColor.r;
        mtrl.Ambient.g = gMaterials[k].ambientColor.g;
        mtrl.Ambient.b = gMaterials[k].ambientColor.b;
        mtrl.Diffuse.r = gMaterials[k].diffuseColor.r;
        mtrl.Diffuse.g = gMaterials[k].diffuseColor.g;
        mtrl.Diffuse.b = gMaterials[k].diffuseColor.b;
        mtrl.Diffuse.a = mtrl.Ambient.a = 1.0;
        g_pd3dDevice->SetMaterial(&mtrl);
        g_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST,
                    startVertexIndex, nPatchesPerMaterial[k]);
        startVertexIndex += 3 * nPatchesPerMaterial[k];
    }
}
```

A háromszögeket most is anyagonként csoportosítva rajzoljuk ki. Minden anyagnál ugyanazt a csúcsbuffert használjuk. A `DrawPrimitive()` metódus első paramétere mondja meg, hogy a csúcspontokat hogyan kell kezelni. Mivel ez `D3DPT_TRIANGLE-LIST`, ezért minden egymást követő 3 csúcs tekinthető egy háromszögnek. A második paraméter az aktuális csúcsbufferbeli kezdő indexet, a harmadik paraméter pedig a rajzolandó primitívek számát jelenti.

11.3. OpenGL kontra DirectX

Az OpenGL és a DirectX versengését látva nagy valószínűséggel a következő ki nem mondott kérdés fogalmazódik meg a kedves Olvasó fejében: Melyik API jobb? Melyiket használjam? Ez az egyszerű kérdés azonban már számos vita elindítója, betört orr és monoklis szem okozója volt. Mivel testi épségünket mi is féltjük, nem fogunk győztest hirdetni.

Szerintünk mindkét API egyformán jól használható. Természetesen hátrányok is vannak mindkét oldalon. Tagadhatatlan például, hogy mivel az OpenGL-nek még nem létezik 2.0-ás verziója, csak a gyártóspecifikus OpenGL kiterjesztésekkel lehet a pixel- és csúcspont-árnyalókat (*shader*) (7.19.1. fejezet) programozni. Ez azonban még nem billenti a DirectX oldalára a mérleg nyelvét. Súlyosabb vádak, gyakran félreértések is beépültek már a köztudatba, amelyek felett már nem tudunk szemet hunyni:

- Nem valós az az állítás, hogy azért készül a játékok 90%-a DirectX-szel, mert a DirectX gyorsabb, mint az OpenGL. A legtöbb esetben ennek éppen az ellenkezője az igaz. Azonban győztest sebességben sem tudunk hirdetni, mert az egyes API-k sebessége elsősorban a gyártók által írt illesztőprogramok minőségén múlik, amely verzióról-verzióra különbözhet.
- Nem igaz az az állítás sem, amelyet a hívők az OpenGL próféta John Carmacknak tulajdonítanak, miszerint a DirectX annyira bonyolult, hogy a legjobb a szemétbe dobni. A DirectX 5-ös verziója esetén ez még talán igaz lehetett. A 8-as verziótól kezdve azonban a DirectX egyre inkább olyan, mint az OpenGL. Így téves lenne azt gondolni, hogy bonyolultabb.

Akkor mégis miért írja Carmack a DOOM III-at OpenGL-ben és nem DirectX-ben? Talán azért, mert platformfüggetlen (PC, Mac, Linux), vagy azért, mert a DOOM színtér adatszerkezeteihez jobban illeszkedik. Talán azért, mert már megszokta és jobban ért hozzá. Az esetek jelentős részében ez a döntő tényező.

A DirectX fejezet végén még pár olyan dologra szeretnénk felhívni a kedves Olvasó figyelmét, amelyre most nem tértünk ki.

- A DirectX-ben — az OpenGL-lel ellentétben — indexelt primitívek rajzolására is van lehetőségünk. A valós alkalmazásokban — mivel akár felére is csökkenthetik a transzformálandó csúcspontok számát — igen fontos szerepük van.
- Meg kell, hogy említsük még a Direct3D `Mesh` osztályát, amellyel különböző anyagokkal rendelkező háromszögek is egy egységbe vonhatók. Hab a tortán, hogy egy Mesh objektumot segédfüggvények segítségével fájlba írni és állományból olvasni is nagyon egyszerűen lehet. Az osztály különböző tesszellációs és optimalizációs függvényekkel is rendelkezik.

A DirectX mélyebb megismeréséhez az angol nyelvű [58] könyvet ajánljuk. Hasznosnak bizonyulhat még a fejlesztői környezethez adott dokumentációk és mintaprogramok tanulmányozása is.

Irodalomjegyzék

[1] *ART: Advanced Rendering Toolkit*. http://www.artoolkit.org/.

[2] *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2 (3rd Edition)*. 1999.

[3] *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2 (3rd Edition)*. 1999.

[4] *OpenVRML*. 2002. http://www.openvrml.org.

[5] *Radiance*. 2002. http://radsite.lbl.gov/radiance/HOME.html.

[6] *Siggraph OpenGL Course*. 2002. http://www.siggraph.org.

[7] *SourceForge*. 2002. http://sourceforge.net.

[8] *The Mesa 3D Graphics Library*. 2002. http://www.mesa3d.org/.

[9] Gy. Ábrahám. *Optika*. Panem-McGraw-Hill, Budapest, 1997.

[10] Alias|Wavefront. *Learning Maya 4.0*. 2001.

[11] Gy. Antal, L. Szirmay-Kalos, F. Csonka, Cs. Kelemen. Multiple strategy stochastic iteration for architectural walkthroughs. *Computers & Graphics*, 27:285–292, 2003. http://www.iit.bme.hu/˜szirmay/puba.html

[12] M. Ashikhmin, P. Shirley, S. Marschner, J. Stam. State of the Art in Modeling and Measuring of Surface Reflection. 2001. Siggraph Course Notes.

[13] B. Aszódi, Sz. Czuczor. Around the PC with Micro Professzor — Interactive teaching tool with rich multimedia content. *Első Magyar Számítógépes Grafika és Geometria Konferencia*, pp. 17–23, 2002. http://www.iit.bme.hu/˜szirmay/katt/Czuczor.pdf.

[14] B. Aszódi, Sz. Czuczor. Calculating 3D sound-field using 3D image synthesis and image processing. *CESCG 2002, Central European Seminar on Computer Graphics*, 2002. http://www.cg.tuwien.ac.at/studentwork/CESCG/CESCG-2002/BAszodiSCzuzcor/index.html.

[15] Z. Balogh, G. Jakab. Terep, karakterek, és effektusok számítógépes játékokban. *BME IIT, TDK dolgozat*, 2002. http: //www.iit.bme.hu/˜szirmay/icetdk.doc.

[16] D. Baraff. Rigid body simulation. *SIGGRAPH 2001 Course Notes*, 2001.

[17] B. G. Baumgart. Winged-edge polyhedron representation. Technical Report STAN-CS-320, Computer Science Department, Stanford University, Palo Alto, CA, 1972.

[18] P. Beckmann, A. Spizzichino. *The Scattering of Electromagnetic Waves from Rough Surfaces*. MacMillan, 1963.

[19] P. Benkő. *Reconstructing Conventional Engineering Objects from Measured Data*. PhD thesis, Geometric Modelling Studies, GML 2001/2, SZTAKI, Budapest, 2001.

[20] G. Blaskó. Vision based camera matching using markers. *CESCG 2000, Central European Seminar on Computer Graphics*, 2000. http://www.cg.tuwien.ac.at/studentwork/CESCG/CESCG-2000/GBlasko/index.html.

[21] J. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):135–256, 1982.

[22] J. F. Blinn. Simulation of wrinkled surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, pp. 286–292, 1978.

[23] J. F. Blinn. Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, 8(1):82–86, 1988.

[24] P. Bodrogi, J. Schanda. Testing the calibration model of colour CRT monitors. *Displays*, 16(3):123–133, 1995.

[25] Á. Budó. *Kísérleti fizika I-II-III*. Tankönyvkiadó, 1970.

[26] A. Budai. *A számítógépes grafika*. LSI, 1999.

[27] J. Carmack. John Carmack on Shadow Volumes. 2000. http://developer.nvidia.com/docs/IO/2585/ATT/CarmackOnShadowVolumes.txt.

[28] E. Catmull, J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10:350–355, 1978.

[29] D. Chetverikov, D. Stepanov. Robust euclidean alignment of 3D point sets. *Első Magyar Számítógépes Grafika és Geometria Konferencia*, pp. 70–75, 2002. http://www.iit.bme.hu/~szirmay/katt/Chetverikov.pdf.

[30] P. Christensen. Faster photon map global illumination. *Journal of Graphics Tools*, 4(3):1–10, 2000.

[31] R. Cook, K. Torrance. A reflectance model for computer graphics. *Computer Graphics*, 15(3), 1981.

[32] F. C. Crow. Shadow algorithm for computer graphics. *Computer Graphics (SIGGRAPH '77 Proceedings)*, pp. 242–248, 1977.

[33] B. Csébfalvi. *Interactive Volume-Rendering Techniques for Medical Data Visualization*. PhD thesis, Technische Universität Wien, Institut für Computergraphik und Algorithmen, 2001. http://www.cg.tuwien.ac.at/research/theses/.

[34] I. Deák. *Random Number Generators and Simulation*. Akadémia Kiadó, Budapest, 1989.

[35] Sz. Deák. Dynamic simulation in a driving simulator game. *CESCG 2003, Central European Seminar on Computer Graphics*, 2003. http://www.cg.tuwien.ac.at/studentwork/CESCG/CESCG-2003/SDeak/index.html.

[36] Ph. Dutre, E. Lafortune, Y. D. Willems. Monte Carlo light tracing with direct computation of pixel intensities. *Compugraphics '93*, pp. 128–137, Alvor, 1993.

[37] N. Dyn, J. Gregory, D. Levin. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics*, 9:160–169, 1990.

[38] G. Enderle. *Computer graphics programming : GKS, the graphics standard*. Springer-Verlag, 1998.

[39] W. Engel. *Introduction to Shader Programming*. 2002. http://www.gamedev.net/columns/ hardcore/dxshader1/default.asp.

[40] Nyékiné Gaizler Judit et al. *Java 2 Útikalauz Programozóknak*. ELTE TTK Hallgatói Alapítvány, 2001.

[41] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, New York, 1988.

[42] E. Fausett, A. Pasko, V. Adzhiev. Space-time and higher dimensional modeling for animation. *Computer Animation 2000*, pp. 140–145, 2000.

[43] J. D. Foley, A. van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, Mass., 1982.

[44] H. Fuchs, Z. M. Kedem, B. F. Naylor. On visible surface generation by a priory tree structures. *Computer Graphics (SIGGRAPH '80 Proceedings)*, pp. 124–133, 1980.

[45] A. Fujimoto, T. Takayuki, I. Kansei. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[46] T. Gaskins. *PHIGS Programming Manual*. O'Reilly & Associates, 1998.

[47] A. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers, Inc., San Francisco, 1995.

[48] A. S. Glassner. *An Introduction to Ray Tracing*. Academic Press, London, 1989.

[49] P. J. Green and R. Sibson. Computing Dirichlet tessellations in the plane. *Computer Journal*, 21(2):168–173, 1977.

[50] L. Guibas, R. Stolfi. Primitives for the manipulations of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[51] Gy. Hajós. *Bevezetés a geometriába*. Tankönyvkiadó, 1972.

[52] V. Havran. *Heuristic Ray Shooting Algorithms*. Czech Technical University, PhD thesis, 2001. http://www.mpi-sb.mpg.de/˜havran/DISSVH/phdthesis.html

[53] K. Hawkins, D. Astle, A. LaMothe. *OpenGL Game Programming*. PrimaTech, 2002.

[54] P. Heckbert. *BRDF viewer*. 1997. http://www-2.cs.cmu.edu/˜ph/src/illum/.

[55] T. Herman, A. Kuba. *Discrete Tomography: Foundations, Algorithms and Applications*. Birkhauser, Boston, 1999.

[56] H. Hoppe. *Smooth view-dependent level-of-detail control and its application to terrain rendering*. Technical Report http://research.microsoft.com/˜hoppe/svdlod.pdf, Microsoft.

[57] H. Hoppe. Progressive meshes. *SIGGRAPH '96*, pp. 99–108, 1996.

[58] Parberry Ian. *Introduction to Computer Game Programming With DirectX 8.0*. 2001.

[59] D. S. Immel, M. F. Cohen, D. P. Greenberg. A radiosity method for non-diffuse environments. *Computer Graphics (SIGGRAPH '86 Proceedings)*, pp. 133–142, 1986.

[60] Bach Iván. *Formális nyelvek*. Typotex Elektronikus Kiadó, 2002.

[61] H. Jensen, S. Marschner, M. Levoy, P. Hanrahan. A practical model for subsurface light transport. *Computer Graphics (SIGGRAPH 2001 Proceedings)*, 2001.

[62] H. W. Jensen. Global illumination using photon maps. *Rendering Techniques '96*, pp. 21–30, 1996.

[63] H. W. Jensen, N. J. Christensen. Photon maps in bidirectional Monte Carlo ray tracing of complex objects. *Computers and Graphics*, 19(2):215–224, 1995.

[64] H. W. Jensen, P. H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. *Computers and Graphics (SIGGRAPH '98 Proceedings)*, pp. 311–320, 1998.

[65] K. I. Joy, C. W. Grant, N. L. Max, L. Hatfield (editors). *Computer Graphics: Image Synthesis*. IEEE Computer Society Press, Los Alamitos, CA., 1988.

[66] I. Juhász, M. Hoffmann. Knot modification of B-spline curves. *Első Magyar Számítógépes Grafika és Geometria Konferencia*, pp. 38–43, 2002. http://www.iit.bme.hu/˜szirmay/katt/Juhasz.pdf.

[67] J. T. Kajiya. Anisotropic reflection models. *Computer Graphics (SIGGRAPH '85 Proceedings)*, pp. 15–21, 1985.

[68] J. T. Kajiya. The rendering equation. *Computer Graphics (SIGGRAPH '86 Proceedings)*, pp. 143–150, 1986.

[69] Cs. Kelemen, L. Szirmay-Kalos. A microfacet based coupled specular-matte BRDF model with importance sampling. *Eurographics 2001, Short papers, Manchester*, 2001. http://www.iit.bme.hu/˜szirmay/puba.html

[70] Cs. Kelemen, L. Szirmay-Kalos, G. Antal, F. Csonka. Simple and robust mutation strategy for Metropolis light transport. *Eurographics '02*, 2002. http://www.iit.bme.hu/˜szirmay/puba.html

[71] M. Kilgard. *Improving Shadows and Reflections via the Stencil Buffer*. 2000. http://developer.nvidia.com/docs/IO/1348/ATT/stencil.pdf.

[72] M. Kilgard. *OpenGL Programming for the X Window System*. Addison-Wesley Pub Co, 1996.

[73] D. E. Knuth. *The art of computer programming. Volume 2 (Seminumerical algorithms)*. Addison-Wesley, Reading, Mass. USA, 1981.

[74] D. Kochanek. Interpolating splines with local tension, continuity and bias control. *Computer Graphics (SIGGRAPH '84 Proceedings)*, pp. 33–41, 1984.

[75] A. Kónya. *Fizikai kézikönyv műszakiaknak*. Műszaki Könyvkiadó, Budapest, 1985.

[76] Z. Konyha. Aspects of developing a driving simulation game. *CESCG 2000, Central European Seminar on Computer Graphics*, 2000. http://www.cg.tuwien.ac.at/studentwork/CESCG/CESCG-2000/ZKonyha/index.html.

[77] G. Kós. *Computer Aided Geometric Algorithms for Reverse Engineering*. PhD thesis, Geometric Modelling Studies, GML 2001/2, SZTAKI, Budapest, 2001.

[78] G. Krammer. Notes on the mathematics of the PHIGS output pipeline. *Computer Graphics Forum*, 8(8):219–226, 1989.

[79] G. Krammer. *Bevezetés a számítógépi grafikába - jegyzet*. 1999. http://valerie.inf.elte.hu/~krammer/eltettk/grafika/jegyzet/index.html.

[80] J. Kundert and P. L. Gibbs. *Mastering Maya 3*. Sybex Inc., 2003.

[81] E. Lafortune. *Reflectance Data*. 1997. http://www.graphics.cornell.edu/online/ measurements/reflectance/index.html.

[82] E. Lafortune, Y. D. Willems. Bi-directional path-tracing. *Compugraphics '93*, pp. 145–153, Alvor, 1993.

[83] E. Lafortune, Y. D. Willems. Using the modified Phong reflectance model for physically based rendering. Technical Report RP-CW-197, Department of Computing Science, K.U. Leuven, 1994.

[84] B. Lantos. *Robotok Irányítása*. Akadémiai Kiadó, Budapest, Hungary, 1991.

[85] Z. László, K. Kondorosi, L. Szirmay-Kalos. *Objektum-orientált szoftverfejlesztés*. ComputerBooks, Budapest, 1995.

[86] R. Lewis. Making shaders more physically plausible. *Rendering Techniques '93*, pp. 47–62, 1993.

[87] D. Lischinski. Incremental Delaunay triangulation. Paul Heckbert, editor, *Graphics Gems IV*, pp. 47–59. Academic Press, Boston, 1994.

[88] G. Márton. *Sugárkövető algoritmusok átlagos bonyolultságának vizsgálata*. Kandidátusi disszertáció, Magyar Tudományos Akadémia, Budapest, 1995.

[89] K. Matkovic, L. Neumann, W. Purgathofer. *A survey of tone mapping techniques*. Technical report, TU Vienna, 1999. TR-186-2-97-12. http://www.cg.tuwien.ac.at/research/publications/

[90] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1091, 1953.

[91] Microsoft. *DirectX SDK*. 2002. http://www.microsoft.com/windows/directx/default.asp.

[92] G. S. Milller, C. R. Hoffman. Illumination and reflection maps: Simulated objects in simulated and real environment. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 1984.

[93] M. Minnaert. The reciprocity principle in lunar photometry. *Astrophysical Journal*, 93:403–410, 1941.

[94] A. Nemcsics. *Színdinamika, színes környezet mérése*. BME, Budapest, 1990.

[95] L. Neumann, A. Neumann, L. Szirmay-Kalos. Compact metallic reflectance models. *Computer Graphics Forum (Eurographics'99)*, 18(3):161–172, 1999. http://www.iit.bme.hu/˜szirmay/puba.html

[96] H. Niederreiter. *Random number generation and quasi-Monte Carlo methods*. SIAM, Pennsilvania, 1992.

[97] T. Nishita, E. Nakamae. Method of displaying optical effects within water using accumulation buffer. *SIGGRAPH'94*, 1994.

[98] P. Omedas, F. Berrizbeitia, G. Szijártó, B. Kiss, B. Takács. Model-based Facial Animation for Mobile Communication. *Ibero-American Symposium on Comp. Graphics*. 2002, Portugal. http://www.digitalelite.net/Pages/Papers/SIACG2002.pdf

[99] M. Oren. *Computer Graphic Rendering of Material Surfaces*. 1999. http://math.nist.gov/˜FHunt/appearance/rendered.html.

[100] R. Parent. *Computer Animation*. Morgan Kauffmann, 2002.

[101] B. T. Phong. Illumination for computer generated images. *Communications of the ACM*, 18:311–317, 1975.

[102] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling. *Numerical Recipes in C (Second Edition)*. Cambridge University Press, Cambridge, USA, 1992. http://www.library.cornell.edu/nr/nr_index.cgi

[103] G. Renner, A. Ekárt. Genetic algorithms in CAD. *Computer-Aided Design*, 2002.

[104] A. Rényi. *Valószínűségszámítás*. Tankönyvkiadó, Budapest, Hungary, 1981.

[105] *ROAM*. http://www.cs.umu.se/˜tdv94aog/ROAM.pdf.

[106] D. F. Rogers, J. A. Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill, New York, 1989.

[107] D. Rogerson. *Inside COM. Microsoft's Component Object Model*. Microsoft Press, 1997.

[108] P. Rózsa. *Lineáris algebra és alkalmazásai*. Műszaki Könyvkiadó, Budapest, 1976.

[109] Sz. Rusinkiewicz. *bv - a BRDF browser*. 2001. http://graphics.stanford.edu/˜smr/brdf/bv/.

[110] P. Shirley, C. Wang, K. Zimmerman. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(1):1–36, 1996.

[111] I. Sobol. *A Monte-Carlo módszerek alapjai*. Műszaki Kiadó, 1981.

[112] I.E. Sutherland, G.W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[113] L. Szécsi. An effective kd-tree implementation. Jeff Lander, editor, *Graphics Programming Methods*. Charles River Media, 2003.

[114] V. Székely, A. Poppe. *Számítógépes grafika alapjai IBM PC-n*. ComputerBooks, Budapest, 1992.

[115] G. Szijártó, J. Koloszár. Hardware accelerated rendering of foliage for real-time applications. *Spring Conference of Computer Graphics '03*, 2003.

[116] L. Szirmay-Kalos. *Monte-Carlo Methods in Global Illumination*. Institute of Computer Graphics, Vienna University of Technology, Vienna, 1999. http://www.iit.bme.hu/˜szirmay/script.pdf.

[117] L. Szirmay-Kalos. Stochastic iteration for non-diffuse global illumination. *Computer Graphics Forum (Eurographics'99)*, 18(3):233–244, 1999. http://www.iit.bme.hu/˜szirmay/puba.html

[118] L. Szirmay-Kalos. *Számítógépes grafika*. ComputerBooks, Budapest, 1999.

[119] L. Szirmay-Kalos. *Photorealistic Image Synthesis with Ray-Bundles*. Akadémiai doktori disszertáció, Magyar Tudományos Akadémia, Budapest, 2000. http//www.iit.bme.hu/˜szirmay/ThesisSzKL.htm.

[120] L. Szirmay-Kalos, B. Benedek. Stochastic iteration for non-diffuse global illumination. Jeff Lander, editor, *Graphics Programming Methods*. Charles River Media, 2003.

[121] L. Szirmay-Kalos, F. Csonka, Gy. Antal. Global illumination as a combination of continuous random walk and finite-element based iteration. *Computer Graphics Forum (Eurographics'2001)*, 20(3):288–298, 2001.

[122] L. Szirmay-Kalos, V. Havran, B. Benedek, L. Szécsi. On the efficiency of ray-shooting acceleration schemes. *Proc. Spring Conference on Computer Graphics (SCCG '2002)*, pp. 97–106. Comenius University Press, 2002. http://www.iit.bme.hu/˜szirmay/puba.html

[123] L. Szirmay-Kalos (editor). *Theory of Three Dimensional Computer Graphics*. Akadémia Kiadó, Budapest, 1995. http://www.iit.bme.hu/˜szirmay/puba.html

[124] T. Szirányi, Z. Tóth. Optimization of paintbrush rendering of images by dynamic MCMC methods. *Lecture Notes on Computer Science, Springer Verlag*, 2134:201–215, 2001.

[125] T. Várady, R. Martin. Reverse engineering. G. Farin, J. Hoschek, and M. S. Kim, editors, *Handbook of Computer Aided Geometric Design*, pp. 651–681. Springer, 2002.

[126] T. Várady, R. R. Martin, J. Cox. Reverse engineering of geometric models - An introduction. *Computer-Aided Design*, 29(4):255–269, 1997.

[127] M. Varga. *Játékprogramok készítése*. ComputerBooks, Budapest, 1998.

[128] G. Vass. Diffuse and specular interreflections with classical, deterministic ray tracing. *CESCG 2001, Central European Seminar on Computer Graphics*, 2001. http://www.cg.tuwien.ac.at/studentwork/CESCG/CESCG-2002/GVass/index.html.

[129] G. Vass. *Camera Matching in Computer Graphics*. BME, IIT, 2003. Diplomaterv. http://www.vassg.hu/pubs_en.htm

[130] E. Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, http://graphics.stanford.edu/papers/veach_thesis, 1997.

[131] E. Veach, L. Guibas. Bidirectional estimators for light transport. *Computer Graphics (SIGGRAPH '95 Proceedings)*, pp. 419–428, 1995.

[132] E. Veach, L. Guibas. Metropolis light transport. *Computer Graphics (SIGGRAPH '97 Proceedings)*, pp. 65–76, 1997.

[133] J. Vida, R. Martin, and T. Várady. A survey of blending methods that use parametric surfaces. *Computer-Aided Design*, 26(5):341–365, 1994.

[134] VRML2.0. *The Virtual Reality Modeling Language Specification*. 1997. http://www.web3d.org/technicalinfo/specifications/vrml97/index.htm.

[135] G. Ward. Real pixels. James Arvo, editor, *Graphics Gems II*, pp. 80–83. Academic Press, Boston, 1991.

[136] G. Ward. Measuring and modeling anisotropic reflection. *Computer Graphics*, 26(2):265–272, 1992.

[137] A. Watt. *3D Computer Graphics*. Addision-Wesley, 1999.

[138] A. Watt. *3D Games: Real-Time Rendering and Software Technology*. Addision-Wesley, 2001.

[139] L. Williams. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, pp. 270–274, 1978.

[140] L. Williams. Pyramidal parametric. *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pp. 1–11, 1983.

[141] G. Wyvill, C. McPheeters, B. Wyvill. Data structure for soft objects. *The Visual Computer*, 4(2):227–234, 1986.

[142] H. Y. Kwoon. *The Theory of Stencil Shadow Volumes*. 2002. http://www.gamedev.net/reference/articles/article1873.asp.

[143] M. J. Young. *Visual C++ 6 Mesteri Szinten*. Kiskapu Kiadó, 1999.

[144] F. Yuan. *Windows Graphics Programming: Win32 GDI and DirectDraw*. Prentice Hall PTR, 2000.