

SZÁMÍTÓGÉPES GRAFIKA

ÍRTA:
DR. SZIRMAY-KALOS LÁSZLÓ

Borító: Tikos Dóra és Tüske Imre
CD melléklet: Dornbach Péter
Programok: dr. Szirmay-Kalos László és Fóris Tibor
Ábrák: Szertaridisz Elefteria

Lektor: dr. Tamás Péter

1999

Tartalomjegyzék

| | |
|--|-----------|
| 1. A számítógépes grafika céljai és feladatai | 15 |
| 1.1. A modellezés feladatai | 16 |
| 1.2. A képszintézis | 18 |
| 1.3. A képszintézis lépései | 22 |
| 1.3.1. Objektum-primitív dekompozíció | 23 |
| 1.3.2. Világ-kép transzformáció | 23 |
| 1.3.3. Vágás | 23 |
| 1.3.4. Takarási feladat | 23 |
| 1.3.5. Árnyalás | 24 |
| 1.3.6. Színleképzés a megjelenítőeszköze | 24 |
| 1.4. A számítógépes grafika jelfeldolgozási megközelítése | 24 |
| 1.5. Rasztergrafikus rendszerek felépítése | 24 |
| 1.6. A képek tárolása és utófeldolgozása | 26 |
| 1.7. Program: TARGA formátumú képállományok kezelése | 27 |
| 2. Grafikus szoftver alrendszerek felépítése | 29 |
| 2.1. Programvezérelt és eseményvezérelt interakció | 30 |
| 2.2. A grafikus hardver illesztése | 31 |
| 2.3. Program: egy egyszerű grafikus könyvtár | 32 |
| 2.3.1. A logikai és a fizikai szintek összekapcsolása | 35 |
| 2.3.2. A könyvtár megvalósítása DOS operációs rendszer alatt | 36 |
| 2.3.3. A könyvtár megvalósítása Ms-Windows környezetben | 38 |
| 2.3.4. Programtervezés eseményvezérelt környezetekben | 41 |
| 3. A geometriai modellezés | 47 |
| 3.1. Pontok | 47 |
| 3.2. Görbék | 47 |
| 3.3. Szabadformájú görbék | 48 |
| 3.3.1. Lagrange-interpoláció | 49 |

| | | |
|-----------|--|-----------|
| 3.3.2. | Bézier-approximáció | 50 |
| 3.3.3. | Összetett görbék | 51 |
| 3.4. | Területek | 57 |
| 3.5. | Felületek | 57 |
| 3.5.1. | Kvadratikus felületek | 58 |
| 3.5.2. | Parametrikus felületek | 58 |
| 3.6. | Testek | 59 |
| 3.6.1. | Felületmodellezés | 59 |
| 3.6.2. | Konstruktív tömörtest geometria alapú modellezés | 61 |
| 3.7. | Program: paraméteres görbék | 62 |
| 4. | Színelméleti alapok | 65 |
| 4.1. | A színek definiálása | 66 |
| 4.1.1. | RGB színrendszer | 67 |
| 4.1.2. | CMY színrendszer | 67 |
| 4.1.3. | HLS színrendszer | 67 |
| 4.2. | Színkezelés a 2D és a 3D grafikában | 68 |
| 4.3. | Program: színkezelés | 69 |
| 4.3.1. | Színillesztő függvények | 69 |
| 4.3.2. | Spektrumok kezelése | 70 |
| 4.3.3. | Színérzetek | 71 |
| 5. | Geometriai transzformációk | 73 |
| 5.1. | Elemi affin transzformációk | 73 |
| 5.1.1. | Eltolás | 73 |
| 5.1.2. | Skálázás a koordinátatengely mentén | 73 |
| 5.1.3. | Forgatás | 74 |
| 5.1.4. | Nyírás | 75 |
| 5.2. | Transzformációk homogén koordinátás megadása | 75 |
| 5.3. | Transzformációk projektív geometriai megközelítése | 76 |
| 5.4. | Program: geometriai transzformációs osztály | 79 |
| 6. | Virtuális világmodellek tárolása | 83 |
| 6.1. | Hierarchikus adatszerkezet | 83 |
| 6.2. | A geometria és topológia szétválasztása | 83 |
| 6.3. | CSG-fa | 85 |
| 6.4. | Megjelenítő állományok | 86 |
| 6.5. | Szabványos világmodellek | 86 |
| 6.6. | Program: hierarchikus 3D adatszerkezet | 87 |

| | |
|--|------------|
| 7. A 2D képszintézis | 91 |
| 7.1. Vektorizáció | 92 |
| 7.2. Modellezési transzformáció | 93 |
| 7.3. Ablak-nézet transzformáció | 94 |
| 7.4. A modellezési és az ablak-nézet transzformációk összefűzése | 94 |
| 7.5. 2D vágás | 95 |
| 7.6. 2D raszterizáció | 100 |
| 7.6.1. Szakaszok rajzolása | 100 |
| 7.6.2. Területelárasztás | 105 |
| 7.6.3. Területkitöltés | 105 |
| 7.7. Pixel műveletek | 110 |
| 7.7.1. Dither alkalmazása | 110 |
| 7.8. Interaktív 2D grafikus rendszerek | 111 |
| 7.9. Program: 2D grafikus rendszer | 114 |
| 8. Az árnyalás optikai alapmodellje | 123 |
| 8.1. A fényerősség alapvető mértékei | 123 |
| 8.2. A kamerák jellemzése | 125 |
| 8.3. A fény-felület kölcsönhatás: az árnyalási egyenlet | 127 |
| 8.4. Az árnyalási egyenlet adjungáltja: a potenciál egyenlet | 129 |
| 8.5. Az árnyalási illetve a potenciál egyenlet megoldása | 130 |
| 8.6. BRDF modellek | 131 |
| 8.6.1. Klasszikus BRDF modellek | 132 |
| 8.6.2. Lambert-törvény | 132 |
| 8.6.3. Ideális visszaverődés | 133 |
| 8.6.4. Ideális törés | 134 |
| 8.6.5. Phong illuminációs modell és változatai | 134 |
| 8.7. Fényelnyelő anyagok | 137 |
| 8.8. Program: BRDF modellek | 139 |
| 9. A 3D inkrementális képszintézis | 143 |
| 9.1. Felületek tesszellációja | 146 |
| 9.2. Modellezési transzformáció | 146 |
| 9.3. Kamera definíció | 146 |
| 9.4. A nézeti transzformáció | 148 |
| 9.4.1. Világ-koordinátarendszer — ablak-koordinátarendszer transz- formáció | 149 |
| 9.4.2. Ablak-képtér transzformáció párhuzamos vetítés esetén | 149 |
| 9.4.3. Ablak-képtér transzformáció perspektív vetítés esetén | 151 |
| 9.5. Nézeti csővezeték | 154 |

| | | |
|------------|---|------------|
| 9.5.1. | Vágás homogén koordinátákban | 155 |
| 9.6. | Takarási feladat megoldása | 156 |
| 9.6.1. | Triviális hátsólap eldobás | 156 |
| 9.6.2. | Z-buffer algoritmus | 157 |
| 9.6.3. | Területfelosztó módszerek | 159 |
| 9.6.4. | Festő algoritmus | 161 |
| 9.7. | Lokális illuminációs algoritmusok | 162 |
| 9.7.1. | Saját színnel történő árnyalás | 164 |
| 9.7.2. | Konstans árnyalás | 164 |
| 9.7.3. | Gouraud-árnyalás | 164 |
| 9.7.4. | Phong-árnyalás | 165 |
| 9.8. | Program: 3D grafikus rendszer inkrementális képszintézissel | 165 |
| 10. | A sugárkövetés | 183 |
| 10.1. | Az illuminációs modell egyszerűsítése | 183 |
| 10.2. | A tükör és törési irányok kiszámítása | 185 |
| 10.3. | A rekurzív sugárkövető program | 186 |
| 10.4. | Metszéspontszámítás egyszerű felületekre | 187 |
| 10.4.1. | Háromszögek metszése | 188 |
| 10.4.2. | Implicit felületek metszése | 189 |
| 10.4.3. | Paraméteres felületek metszése | 189 |
| 10.4.4. | Transzformált objektumok metszése | 190 |
| 10.4.5. | CSG modellek metszése | 190 |
| 10.5. | A metszéspontszámítás gyorsítási lehetőségei | 192 |
| 10.5.1. | Befoglaló keretek | 192 |
| 10.5.2. | Az objektumtér szabályos felosztása | 192 |
| 10.5.3. | Az objektumtér adaptív felosztása | 193 |
| 10.6. | Foton követés | 195 |
| 10.7. | Program: rekurzív sugárkövetés | 195 |
| 11. | Globális illuminációs algoritmusok | 203 |
| 11.1. | Integrálegyenletek megoldása | 203 |
| 11.1.1. | Inverzió | 204 |
| 11.1.2. | Véges-elem módszer | 204 |
| 11.1.3. | Expanzió | 205 |
| 11.1.4. | Monte-Carlo integrálás | 207 |
| 11.1.5. | Iteráció | 212 |
| 11.2. | Diffúz eset: radiosity | 212 |
| 11.2.1. | Forma faktor számítás | 214 |
| 11.2.2. | A lineáris egyenletrendszer megoldása | 217 |

| | |
|--|------------|
| 11.2.3. Progresszív finomítás | 218 |
| 11.3. Véletlen bolyongáson alapuló algoritmusok | 219 |
| 11.3.1. Inverz fényútkövetés | 219 |
| 11.4. Program: inverz fényútkövetés | 223 |
| 12. Raszteres képek csipkézettségének a csökkentése | 229 |
| 12.1. Előszűrés | 230 |
| 12.1.1. A szakaszok csipkézettségének csökkentése | 231 |
| 12.2. Utószűrés | 234 |
| 12.3. Program: sugárkövetés kiegészítése csipkézettség csökkentéssel | 236 |
| 13. Textúra leképezés | 237 |
| 13.1. Paraméterezés | 238 |
| 13.1.1. Explicit egyenlettel definiált felületek paraméterezése | 238 |
| 13.1.2. Háromszögek paraméterezése | 239 |
| 13.2. Textúra leképezés a sugárkövetésben | 239 |
| 13.3. Textúra leképezés az inkrementális képszintézisben | 240 |
| 13.4. A textúrák szűrése | 241 |
| 13.5. Bucka leképezés | 243 |
| 13.6. Visszaverődés leképezés | 245 |
| 13.7. Program: sugárkövetés kiegészítése textúra leképezéssel | 245 |
| 14. Térfogat modellek és térfogatvizualizáció | 247 |
| 14.1. Direkt térfogatvizualizációs módszerek | 247 |
| 14.1.1. Térfogat vetítés | 249 |
| 14.1.2. Térfogati sugárkövetés | 249 |
| 14.2. A voxel szín és az opacitás származtatása | 250 |
| 14.3. Indirekt térfogatvizualizációs módszerek | 251 |
| 14.3.1. Masírozó kockák algoritmus | 251 |
| 14.3.2. Fourier-tér módszerek | 252 |
| 14.4. Program: masírozó kockák algoritmus | 254 |
| 15. Fraktálok | 257 |
| 15.1. A Hausdorff-dimenzió | 257 |
| 15.1.1. Fraktális dimenzió nem önhasznó objektumokra | 260 |
| 15.2. Brown-mozgás alkalmazása | 261 |
| 15.3. Kaotikus dinamikus rendszerek | 263 |
| 15.4. Kaotikus dinamikus rendszerek a síkon | 265 |
| 15.4.1. Julia-halmazok | 266 |
| 15.4.2. A Mandelbrot-halmaz | 270 |

| | |
|--|------------|
| 15.5. Iterált függvényrendszerek | 272 |
| 15.5.1. Iterált függvényrendszerek attraktorának előállítása | 273 |
| 15.5.2. IFS modellezés | 276 |
| 15.5.3. Fraktális képtömörítés | 278 |
| 15.6. Program: IFS rajzoló | 279 |
| 16. Számítógépes animáció | 281 |
| 16.1. Pozíció-orientáció mátrixok interpolációja | 284 |
| 16.2. A kameraparaméterek interpolációja | 285 |
| 16.3. Mozgás tervezés | 285 |
| 16.4. Dupla pufferelés | 288 |
| 17. Térfogat modellek és térfogatvizualizáció | 289 |
| 17.1. Direkt térfogatvizualizációs módszerek | 289 |
| 17.1.1. Térfogat vetítés | 291 |
| 17.1.2. Térfogati sugárkövetés | 291 |
| 17.2. A voxel szín és az opacitás származtatása | 292 |
| 17.3. Indirekt térfogatvizualizációs módszerek | 293 |
| 17.3.1. Masírozó kockák algoritmus | 293 |
| 17.3.2. Fourier-tér módszerek | 294 |
| 17.4. Program: masírozó kockák algoritmus | 296 |
| 18. CD melléklet | 315 |
| 18.1. Demonstrációs programok a könyv fejezeteihez | 315 |
| 18.1.1. A programokat felépítő közös fájlok | 316 |
| 18.1.2. Grafikus keretrendszer Ms-Windows és DOS/BGI környezetre | 320 |
| 18.1.3. Példa alkalmazások | 320 |
| Irodalomjegyzék | 327 |

Előszó

Életünk folytonos szemlélődéssel telik, figyeljük környezetünket és feldolgozzuk a minket ért hatásokat. A hatások gondolatokat és érzelmeket keltenek bennünk, amelyeket szeretnénk kifejezni, maradandóvá tenni és másokkal megosztani. A képek formájában befogadott és továbbadott információk mindig is fontosak voltak az emberek számára. Képek, rajzok segítségével a bonyolult gondolatokat is egyszerűen és közérthetően kifejezhetjük, az ilyen formában kapott információkat gyorsan befogadjuk, megértjük és könnyen megjegyezzük. A kis gyermekektől kezdve, a festőkön át, a tervező mérnökökig mindenki szívesen “rajzolgat”, hogy elképzeléseit mások számára is elérhetővé tegye. A számítógép ebben a folyamatban hatékony társ lehet, mert képes arra, hogy a fejünkben körvonalazódó vázlatokat átvegye és azokból meggyőző képeket készítsen. A számítógép munkája során alkalmazhatja a fizika törvényeit, Dali vagy Picasso stílusát, az építészek vagy a gépészek által követett rajzolási szabályokat, vagy akár teljesen újszerű látásmódot is követhet. Így a kapott eredmény lehet olyan, mintha csak fényképezőgéppel vagy ecsettel készítettük volna, olyan, mintha egy tervezőiroda műszaki rajzolóinak a szorgalmát és ügyességét dicsérné, de bepillantást engedhet olyan világokba is, amelyekből még sohasem értek minket képi hatások, ezért többségünk számára mindig is felfoghatatlanok voltak.

A *számítógépes grafika* célja az, hogy a számítógépből olyan eszközt varázsoljon, amely vázlatos gondolatainkról képeket alkot. Egy ilyen eszköz sokrétű ismereteket foglal magában. A gondolatainkban szereplő alakzatok megadásához a geometriához kell értenünk, a fény hatásának modellezéséhez az optika törvényeit alkalmazzuk. A számítógép monitorán megjelenő képet az emberi szem érzékeli és az agy dolgozza fel, ezért a számítási folyamatoknak figyelembe kell venniük az emberi szem és agy lehetőségeit és korlátait is. Mivel a “fényképezést” számítógépes programmal kell megoldani, a szoftvertechnológia, algoritmusok és adatszerkezetek ismeretétől sem tekinthetünk el. Ráadásul a képek megjelenítéséhez és előállításához a szűkre szabott idő miatt hardver támogatás is szükséges, ezért a legjobb, ha már most elkezdjük felfrissíteni a hardver ismereteinket.

A számítógépes grafika nehéz, mert nagyon sokféle tudást kell megszerezni ahhoz, hogy igazán sajátunknak érezzük. Ugyanakkor a számítógépes grafika nagyon

szép is, mert kincseket lelhet benne az integrálegyenletekkel foglalkozó matematikus, az optikában vagy a Maxwell-egyenletekben elmélyedő fizikus, a látás rejtelmait kutató orvos, az adatstruktúrákkal és az algoritmusokkal bővészkedő programozó, és az egész alkalmazó képzőművész vagy tervezőmérnök. Az interaktív grafikus programok, képek, filmek, számítógépes játékok formájában megjelenő eredmény pedig mindannyiunk gyönyörűségére szolgál. Ezen könyv elsősorban szoftvertervezők és programozók számára készült, szerkezete a Budapesti Műszaki Egyetem informatikus és villamosmérnöki szakjain előadott számítógépes grafika tárgy tematikáját követi.

A könyv célja, hogy megtanítsa az olvasót arra, hogy hogyan kell grafikus rendszereket fejleszteni. Az előismeretek is ennek megfelelőek, a könyv nagy része ugyan csupán középiskolai matematikai és fizikai ismereteket használ, azonban néhány rész épít a természettudományi és műszaki egyetemeken oktatót matematikára is.

Habár a könyv elsősorban szoftverfejlesztőknek szól, a magam részéről reménykedem abban, hogy a szoftverfejlesztőkön kívül a grafikus rendszerek felhasználóihoz és a számítógépes játékokat megszállottként űzőkhöz is eljut, és ezáltal jobban megértik és megbecsülik kedvenc alkalmazói rendszerüket, és talán kedvet kapnak ahhoz is, hogy a felhasználók roppant széles táborából a fejlesztők sokkal szűkebb táborába kalandozzanak el.

A könyv algoritmusainak a nyelve

A grafikai algoritmusokat két szinten tárgyaljuk. Először egy matematikai jelöléseket használó pszeudokód segítségével fogalmazzuk meg az algoritmusokat. Az utasításokat külön sorba írjuk vagy vesszővel választjuk el egymástól. Ez a pszeudokód a feltételt a ciklus elején vizsgáló ciklusokat `for` és `endfor` illetve `while` és `endwhile` utasítások közé, a feltételt a ciklus végén vizsgáló ciklusokat `do` és `while` illetve `repeat` és `until` utasítások közé, végül a feltétel nélküli ciklusokat `loop` és `endloop` utasítások közé helyezi el. A függvényekből, szubrutinokból a `return` utasítás segítségével térhetünk vissza, a függvények bemeneti és kimeneti paramétereit általában a \rightarrow jel választja el. Ahol ezt külön szeretnénk hangsúlyozni, ott az egész változókat nagy betűvel jelöljük. A szokásos matematikai jelölések mellett a pszeudokód még alkalmazza a legközelebbi egészt megkereső `round` és az egészsrészt előállító `trunc` függvényeket, valamint a C++ nyelvből kölcsönzött, a változót eggyel inkrementáló `++` operátort és a x változót y -nal növelő $x += y$ műveletet. A `Pixel(x, y, color)` művelet az x, y koordinátájú képpontot $color$ színre állítja.

A pszeudokódon túl a legfontosabb algoritmusok C++ nyelvű implementációját is megadjuk. Ezen példák megértéséhez alapvető programozási ismeretek szükségesek. A programok elkészítése során arra törekedtünk, hogy azok szépek és könnyen érthetők legyenek.

A CD melléklet

A könyvhöz CD melléklet is tartozik, amelyet Dornbach Péter állított össze részben a saját maga, Fóris Tibor és jómagam által írt programokból, részben szabadon terjeszthető, forrásnyelven rendelkezésre álló programokból. A CD tartalmazza a könyvben tárgyalt példaprogramokat, valamint forráskóddal együtt feltett kész grafikai alkalmazásokat, mint a *StingRay* Monte-Carlo sugárkövető program, az *Eagles* valósidejű helikopter szimulátor, a népszerű *DOOM* játékprogram, a *PovRay* sugárkövető program különböző platformokra a kiegészítő programokkal együtt, *OpenGL* könyvtárak, dokumentációk és példák, *DirectX* futtató környezet, és egy *JPEG* konverter program. A fentiekén kívül *MGF*, *3DS* és *PovRay* formátumú geometriai adatbázisokat, ezen adatbázisok értelmező programjait és képeket tettünk a CD-re. A CD lehetőségeit *html* böngészőkkel, azaz például a *Internet Explorer* program segítségével, tárhatjuk fel.

Hogyan készült a könyv?

A szerkesztési munkákat \LaTeX szövegszerkesztővel végeztük, a magyar ékezetekkel és az elválasztással Verhás Péter által készített HION program, a magyar megjegyzésekkel ellátott programlistákkal pedig Kocsis Tamás HUTA programja birkózott meg. A könyvben szereplő képek egy részét a CD-n található mintaprogramok segítségével számítottuk ki és *TARGA* illetve *JPEG* formátumban mentettük el. A mások által készített képeket általában *GIF* vagy *JPEG* formátumban kaptuk meg. A képeket az *xv* program alakította egységesen *EPS* formátumra. A rajzokat Szertaridisz Elefteria részben *tgif* programmal, részben *CorelDraw* programmal készítette, és a további műveletekhez ugyancsak *EPS* formátumban állította elő. A grafikonokat a *gnuplot* program rajzolta meg a mérési eredményekből. A lefordított \LaTeX fájlt és az *EPS* formátumú grafikus elemeket a *dvips* programmal *Postscript* alakra hoztuk amit egy saját programmal tükröztünk. Ezzel a tükrőírásos fekete-fehér oldalakat már nyomtathattuk is, a színes oldalakat, azonban még cián, magenta, sárga és fekete árnyalatokra bontottuk és a 4 színre külön készítettünk nyomdai sablonokat.

Hogyan készült a borító?


A borítót Tikos Dóra és Tüske Imre tervezte. A borítón szereplő 3 dimenziós objektumtér képét sugárkövető algoritmus számította ki. A keletkezett képre a *Photoshop* programmal kerültek rá a feliratok és a logók, majd ugyanezen program bontotta fel a színes képet cián, magenta, sárga és fekete árnyalatokra, amelyeket egyenként levilágítva kaptuk meg a nyomdai maszkokat.

Köszönetnyilvánítás

A könyv a Budapesti Műszaki Egyetem Irányítástechnika és Informatika Tanszékén készült, az itt rendelkezésre álló számítástechnikai infrastruktúra felhasználásával, és a kollegáim támogatásával. Személyes köszönetemet szeretném kifejezni dr. Márton Gábornak, dr. Horváth Tamásnak, Fóris Tibornak, Csébfalvi Balázsnak, Dornbach Péternek, Szalavári Zsoltnak, Pap Gábornak, Fábián Jánosnak, Szirmay-Kalos Barnabásnak és a Bécsi Műszaki Egyetem Számítógépes Grafika Intézete oktatóinak, hogy az általuk készített képeket, programrészleteket felhasználhattam a könyvben, dr. Tamás Péternek, Megyeri Zsuzsának és Keszthelyi Máriának a könyv különböző változatainak az átnézéséért. A kutatási munkát a KHVM Konvergencia stratégia az informatikában elnevezésű pályázat és a T029135 számú OTKA pályázat támogatta.

Egy könyv elkerülhetetlen sorsa az, hogy olvasói kisebb-nagyobb hibákat találnak benne. Hálás lennék azoknak, akik kritikai észrevételeiket eljuttatják hozzám (email: szirmay@iit.bme.hu). A megjegyzéseknek — legyenek azok bármilyen elmarasztalóak is — nemcsak azért örülnék, mert hozzájárulnának ahhoz, hogy egy esetleges későbbi kiadásban kevesebb hiba maradjon, hanem azért is, mert ezek az észrevételek talán azt is mutatnák, hogy vannak, akik a könyvet elmélyülten és gyakorta forgatják.

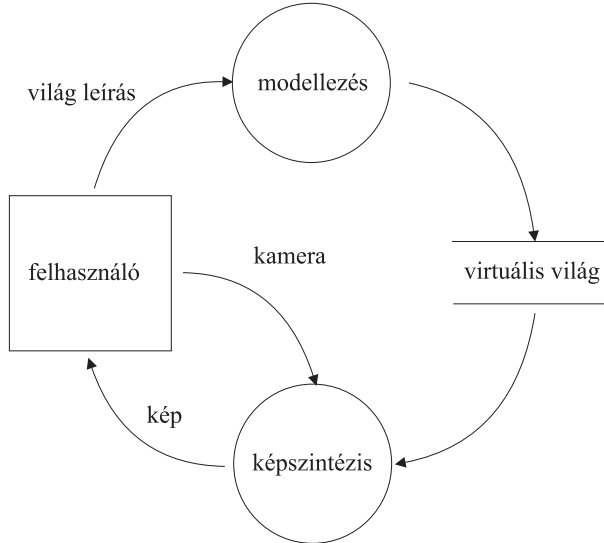
Budapest, 1999.



1. fejezet

A számítógépes grafika céljai és feladatai

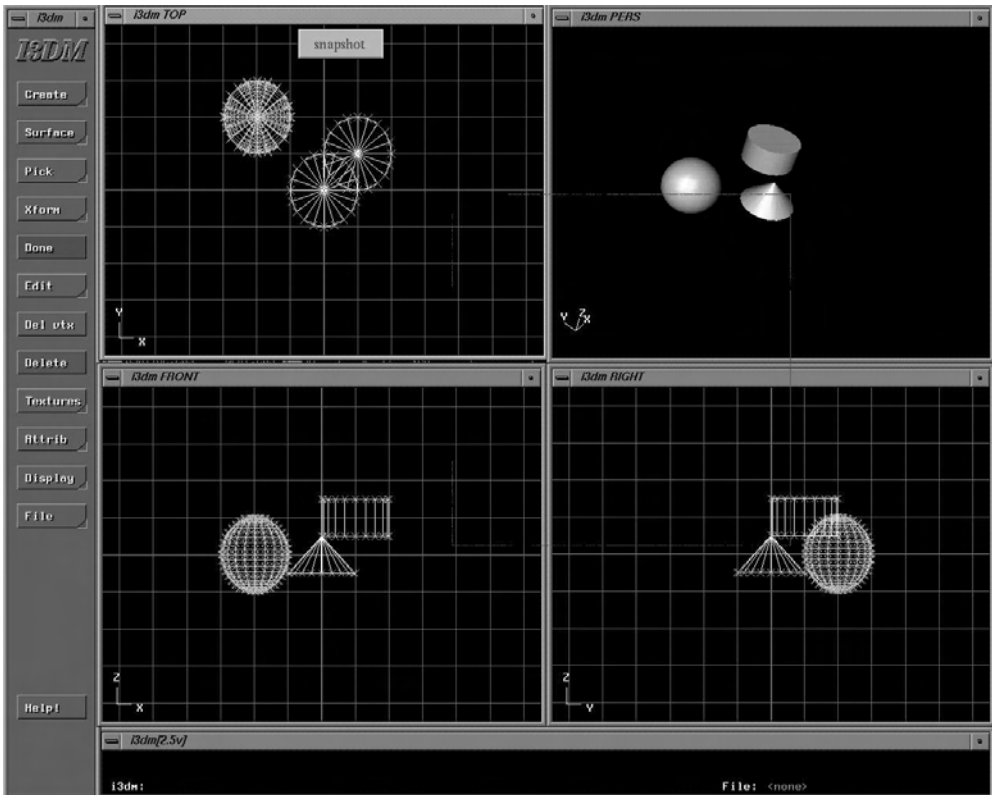
A számítógépes grafika feladata az, hogy a felhasználó számára egy *virtuális világról* fényképeket készítsen és azt a számítógép képernyőjén megjelenítse. A világ leírását nevezzük *modellezésnek*. Ezt követi a *képszintézis*, amely a memóriában tárolt világról fényképet készít.



1.1. ábra. A számítógépes grafika adatfolyam-modellje

1.1. A modellezés feladatai

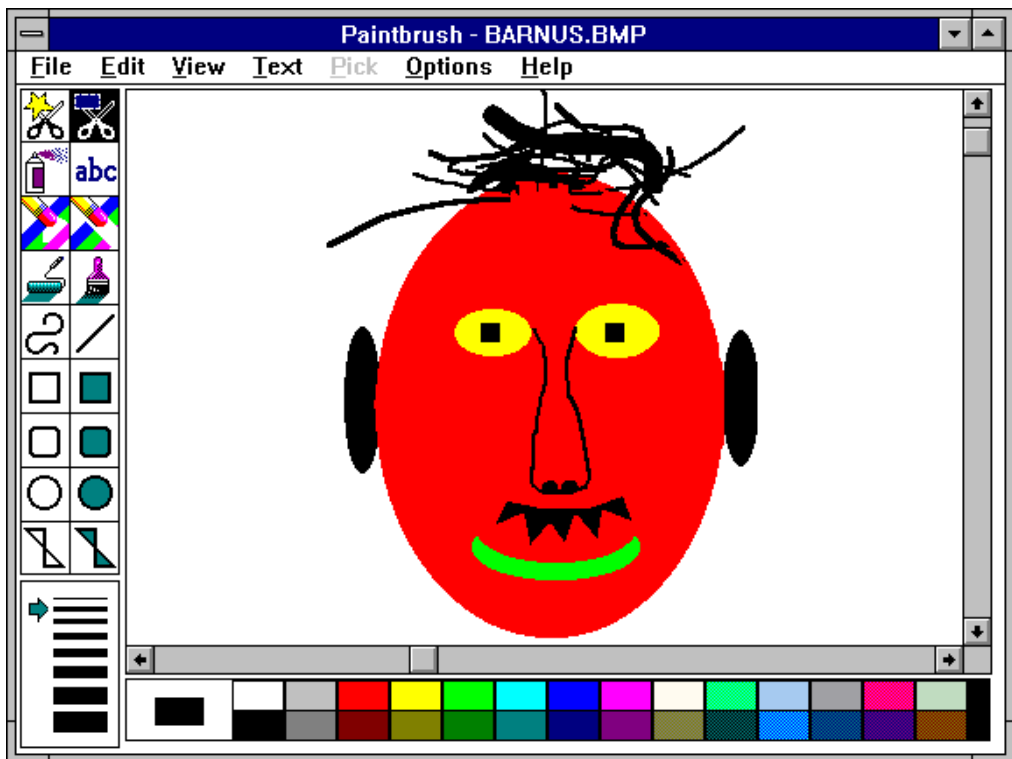
A *modellezés* során virtuális világot írunk le a modellezőprogram utasításai és rögzített elemkészlete segítségével. A modellező lehet a felhasználó, vagy akár az alkalmazói program. A felhasználók a modellt általában *interaktív* módon építik fel. Az interaktív világépítés során elemi parancssorozattal definiáljuk a modellt, és minden egyes parancs után a képszintézis képet készít a modell aktuális állapotáról. Ily módon folyamatos visszajelzést kapunk a készülő virtuális világról.



1.2. ábra. Egy tipikus modellezőprogram felhasználói felülete

A modellezés terméke a *virtuális világ*, amelyet a felhasználó módosíthat, a képszintézis programmal megjeleníthet, vagy akár más programokkal analizálhat. A módosíthatóság és más programokkal történő elemezhetőség érdekében a virtuális világ modell belső reprezentációja nem kötődhet nagyon szorosan a képszintézishez, hanem természetes fogalmakat és dimenziókat kell használnia. A felhasználói interfészen megjelenő fogalmak (objektum, primitív, stb.) alapján építkező virtuális világokban könnyen meg-

oldható, hogy a felhasználó egy modellelemet kiválasszon, és azt interaktív eszközökkel módosítsa, esetleg le is törölje. A természetes dimenziókat használó tér referencia rendszerét gyakran *világ-koordinátarendszer*nek nevezzük. Az interaktív rendszerekben szükségképpen megjelenik az utolsó művelet hatását megszüntető *visszavonás* (*Undo*) művelet. Ezt többféleképpen is megvalósíthatjuk. A legegyszerűbb esetben tároljuk a világmodell korábbi állapotait és szükség esetén azokhoz térünk vissza. Jobb megoldásnak tűnik, ha a felhasználó által kiadott parancsokat tároljuk, és vagy az utolsó műveletet inverzét hajtjuk végre, vagy a teljes modellt letöröljük és a műveletsort a korábbi műveletig újra lejátszunk.



1.3. ábra. Egy egyszerű rajzolóprogram (*Paintbrush*)

Egyszerűbb *rajzolóprogramok* (például a *Paintbrush*) magát a képet tekintik a virtuális világ modelljének. Ilyen rendszerekben a korábban bevitt elemek nem választhatók ki, nem módosíthatók és nem távolíthatók el, csupán újabb elemekkel elfedhetők. Az ilyen rendszerek használata tehát egyrészt nehézkes, másrészt az elkészült "modell" a megjelenítésen kívül másra nem használható.

1.2. A képszintézis

A *képszintézis* (*rendering* vagy *image synthesis*) célja a virtuális világ lefényképezése. A fényképezés során többféle "látásmódot" követhetünk. A két legkézenfekvőbb módszer a rajzolás és a természet folyamatainak a szimulálása. Az első esetben a keletkező képek műszaki rajzszerűek lesznek, a második esetben pedig a keletkező képek annyira fognak hasonlítani a valódi fényképekre, amennyire a szimuláció során követtük a fizikai törvényeket (1.4., 17.19. ábra). Habár ezen könyv döntő részben a 2 dimenziós műszaki rajz és a 3 dimenziós fényhatások fizikai törvényeit alkalmazó fényképezési eljárásokkal foglalkozik, a számítógépes grafika lehetőségei nem merülnek ki ezekben. Fényképezhetünk absztrakt világokat is, és ezáltal láthatóvá tehetjük a gazdasági folyamatokat, bonyolult áramlási rendszereket (1.5. ábra), ipari irányítási rendszerek pillanatnyi állapotát (1.7. ábra), egy számítógépes tomográf vagy más fizikai mérőeszköz által összegyűjtött adathalmazt (1.6. ábra), egy többdimenziós függvényt vagy akár egy síkbeli illetve térbeli gráfot (1.8. ábra).

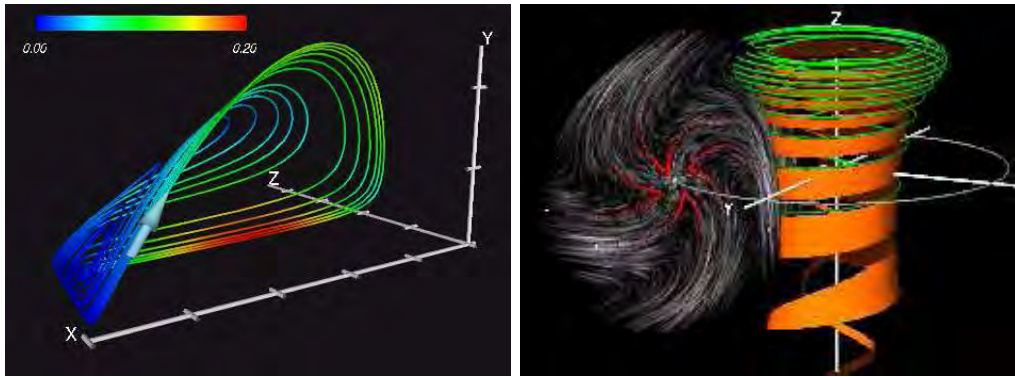


1.4. ábra. 3D világok fényképei: Bal: a fény fizikailag pontos szimulációja [SK98b]; Jobb: Egyszerűsített árnyalási modellel dolgozó valósidejű helikopter szimulátor [Dor97]

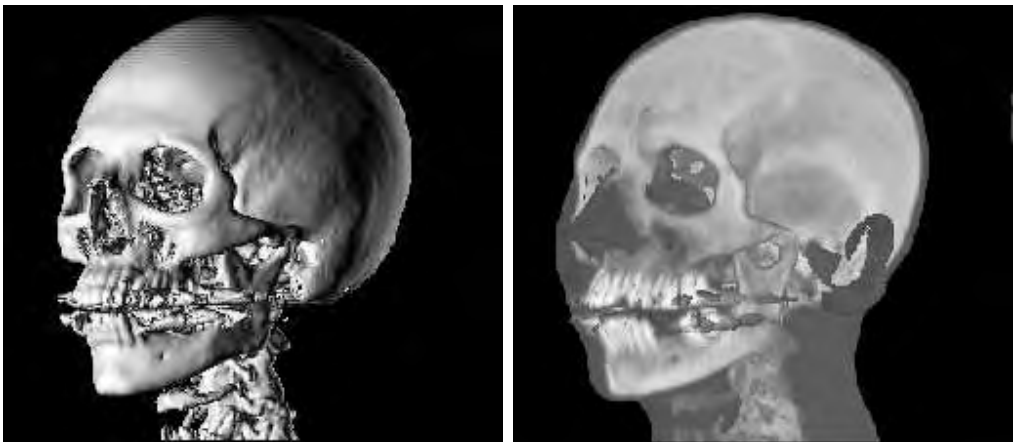
Az igazán valóság-hű képek előállítását *fotorealisztikus képszintézis*nek nevezzük. *Valóság-hűség*en azt értjük, hogy a számítógép monitorán kibocsátott hullámok megközelítőleg hasonló illúziót keltenek, mintha a valós világot szemlélnénk.

Ebben a folyamatban három alapvető szereplő vesz részt:

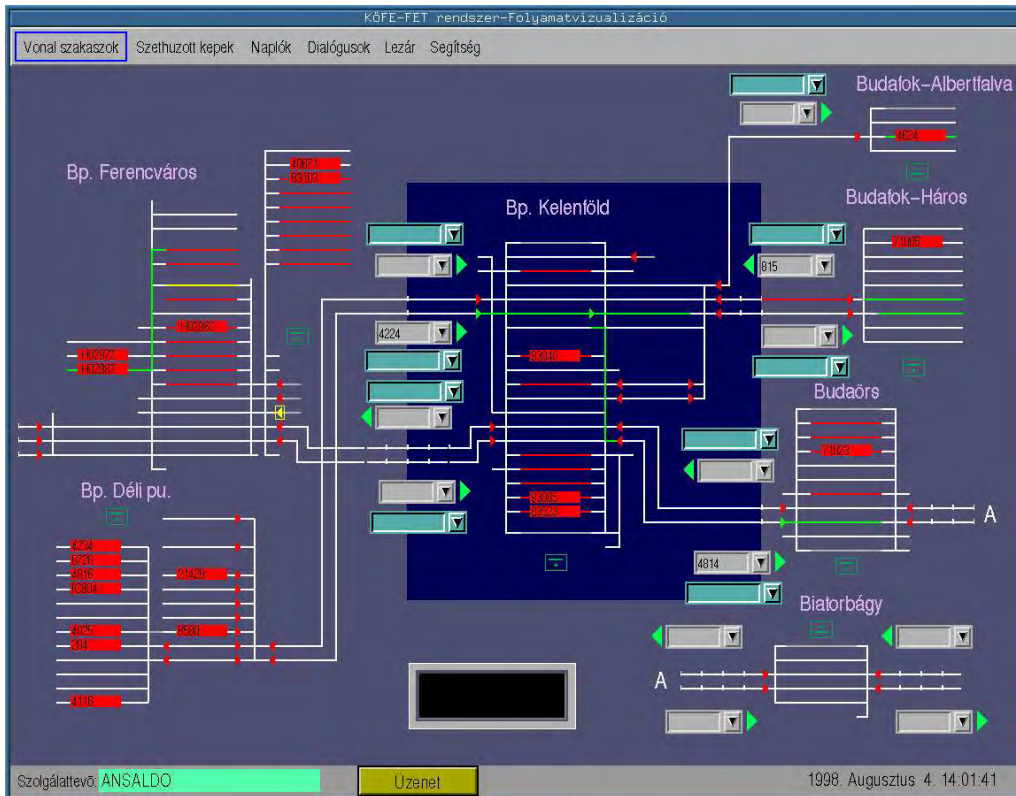
- Az első a számítógépes program, amely a világleírás alapján szimulálja a virtuális térben bekövetkező fényhatásokat és vezérli a grafikus megjelenítőt.



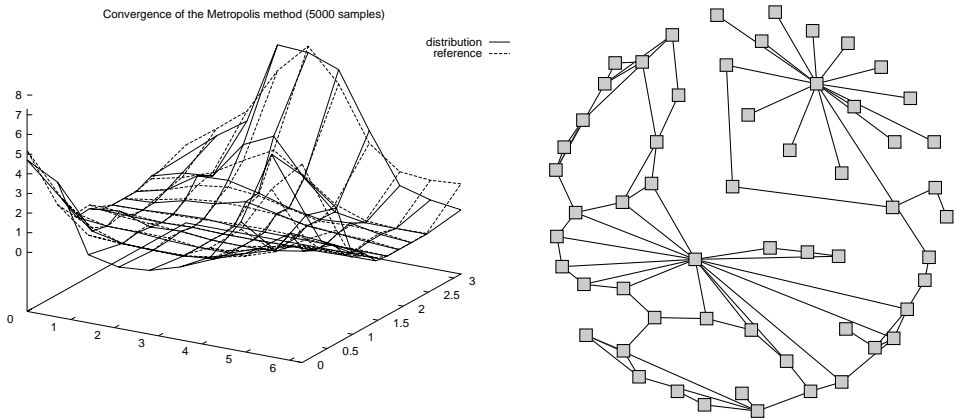
1.5. ábra. Absztrakt világok fényképei: gazdasági és áramlástan modellek vizualizációja (Bécsi Műszaki Egyetem, Számítógépes Grafika Intézet) [Lőf98]



1.6. ábra. Absztrakt világok fényképei: számítógépes tomográf mérési eredményeinek vizualizációja [BSKG97]



1.7. ábra. Bp-Hegyeshalom vasút folyamat-vizualizációja [SKMFF96]



1.8. ábra. Absztrakt világok fényképei: függvények és gráfok megjelenítése [SK94]

- A második a grafikus megjelenítő, amely a képernyő kijelölt pontjait a vezérlés által meghatározott spektrumú fény kibocsátására gerjeszti.
- Végül az utolsó szereplő az emberi szem, amely a fényhullámokat felfogja és az agyban színérzetet hoz létre.

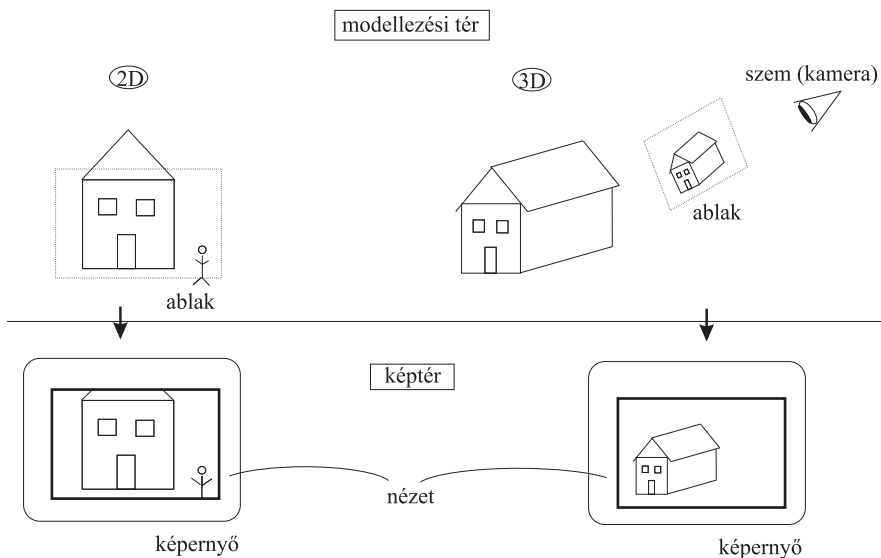
Képszintézis rendszerek létrehozása során tehát ismernünk kell a fény-anyag kölcsönhatás alapvető fizikai modelljét, a megjelenítő rendszerek képességeit és az emberi szem tulajdonságait egyaránt.

Ezek alapján nyilvánvaló, hogy a számítógépes programnak olyan pontosan kell szimulálnia a valós fényhatásokat, amilyenek a követésére a kijelző egyáltalán képes, és amilyenek a megkülönböztetésére a szem alkalmas. Ez utóbbi miatt a számítógépes grafika nem csupán a fizikai illetve matematikai modellek alkalmazása, hanem olyan tudomány, amely az emberi szem korlátait is képes saját előnyére fordítani. Erre annál is inkább szüksége van, mert a bonyolult fizikai modellek megoldásához roppant kevés idő áll rendelkezésre. Gondoljunk csak a mozgókép sorozatot valós időben előállító repülőgép szimulátorra, vagy csupán egy kommersz játékprogramra. Ahhoz, hogy a mozgás folytonosnak tűnjön, másodpercenként legalább 15 képet kell előállítani. Mivel egy közepesnél jobb képernyő képe viszont kb. egy millió képpontból áll, egyetlen képpont színének meghatározásához átlagosan kb. $1\text{sec}/15/10^6 \approx 60\text{ nsec}$ (!) idő áll rendelkezésre, ami viszont az operatív tár egyetlen memória ciklusidejénél is kisebb. Gondolkozzunk el ezen egy kicsit! Egy képpont színének meghatározásához ki kell nyomoznunk, hogy abban melyik objektum látszik, majd a fényhatásokat szimulálva

számítanunk kell a róla visszaverődő szint. Ehhez a processzornak számtalan utasítást kell végrehajtania, amelyek mindegyike több ciklusból áll, majd beírnia az eredményt a tárbá. Hogyan is lehetséges a számításokat és a beírást kevesebb idő alatt elvégezni, mint egyetlen ciklusidő, vagy akár az eredmény beírásának az ideje? Ezen könyv elkészítésekor többek között erre a kérdésre próbáltunk választ adni.

A képszintézis algoritmusai függenek attól, hogy a virtuális modell 2 vagy 3 dimenziós (mérnöki vizualizációban ennél magasabb dimenziók is előfordulhatnak), ezért a képszintézis algoritmusokat alapvetően eszerint osztályozzuk. Tehát a két dimenziós, vagy röviden *2D* modellek megjelenítése esetén *2D* képszintézisről vagy *2D* számítógépes grafikáról, míg a három dimenziós, azaz *3D* modellek megjelenítése esetén *3D* grafikáról fogunk beszélni.

1.3. A képszintézis lépései



1.9. ábra. 2D és 3D képszintézis összehasonlítása

A *2D képszintézis* egy téglalap alakú *2D ablakot* (*window*) helyez a síkban ábrázolt virtuális világra, lefényképezi azon objektumokat, objektumrészteket, amelyek az ablak belsejébe esnek, majd a képet a képernyő ugyancsak téglalap alakú *nézetében* (*viewport*) megjeleníti. A *3D képszintézis* ezzel szemben egy általános helyzetű téglalapot tesz be a világ-koordináta-rendszerbe, *szemet* vagy *kamerát* helyez el az ablak

mögött és a világnak a szemből az ablakon keresztül látható részéről készít képet, amit végül ugyancsak a képernyő nézetében jelenít meg.

1.3.1. Objektum-primitív dekompozíció

A virtuális világot általában az adott alkalmazási területen természetes fogalmakkal definiáljuk és tároljuk. Például egy térinformatikai rendszer alapvető objektumai az épület, út, hálózat, település, stb. Egy általános képszintézis program viszont nyilván nem készülhet fel egyszerre az összes alkalmazási terület fogalmainak megértésére, hanem a saját alkalmazásfüggetlen objektumaival dolgozik. A képszintézis program által kezelt alapobjektumok általában a *geometriai primitívek*, mint például a poligon, gömb, fényforrás, stb. Ezért a képszintézis első lépése a virtuális világmodellnek, a képszintézis program számára történő lefordítása.

1.3.2. Világ-kép transzformáció

A világ-koordinátarendszerben rendelkezésre álló primitívek alapján a képernyőn — azaz egy másik koordináta rendszerben — kell képet készíteni. A koordinátarendszer váltásához geometriai transzformációk szükségesek. 3D grafikában ez a transzformáció vetítést is tartalmaz, hiszen a modell 3 dimenziós, míg a kép mindig 2.

1.3.3. Vágás

A képszintézis a modell azon részét fényképezi, amely a 2D ablakon belül, vagy a 3D ablak és a szem által definiált végtelen piramison belül helyezkedik el. Az ezeken kívül eső objektumokat, objektumrészleteket valamikor ki kell válogatni és el kell hagyni. Ezt a folyamatot nevezzük *vágásnak* (*clipping*).

1.3.4. Takarási feladat

A transzformációk több objektumot is vetíthetnek ugyanarra a képpontra. Ilyenkor el kell döntenünk, hogy melyiket jelenítsük meg, azaz melyik takarja a többi objektumot az adott pontban. Ezt a lépést *takarásnak*, vagy *takart felület/él elhagyásnak* (*hidden surface/line elimination*) nevezzük. 2D grafikában nincs olyan geometriai információ, amely alapján ezt a döntést meghozhatnánk, ezért a *takarást* az objektumok egy különleges tulajdonsága, az ún. *prioritása* alapján határozhatjuk meg. 3D grafikában nyilván a szempozícióhoz legközelebbi objektumot kell választanunk.

1.3.5. Árnyalás

Ha sikerült eldönteni, hogy egy képpontban melyik objektum látszik, akkor a képpontot ennek megfelelően kell kiszínezni. 2D grafikában a színezés az objektum saját színével történik. 3D grafikában viszont a látható szín a térben fennálló fényviszonyok bonyolult függvénye. Összehasonlítva a 2D és 3D grafika képszintézis lépéseit megállapíthatjuk, hogy a 3D grafikát a takarás és az *árnyalás (shading)* teszi nehezebbé a 2D grafikánál.

1.3.6. Színleképzés a megjelenítőeszközre

Az árnyalás a kép egyes pontjain keresztül a szembe jutó fény intenzitását határozza meg a hullámhossz függvényében. A hullámhosszfüggő intenzitás-eloszlást *spektrum*nak nevezzük. Ezen spektrum által keltett színérzetet kell a megjelenítőeszköz lehetőségeinek a figyelembevételével a lehető legpontosabban visszaadni, azaz a spektrumot le kell képezni az eszköz által megjeleníthető hullámhosszokra és intenzitásokra (*tone-mapping*).

1.4. A számítógépes grafika jelfeldolgozási megközelítése

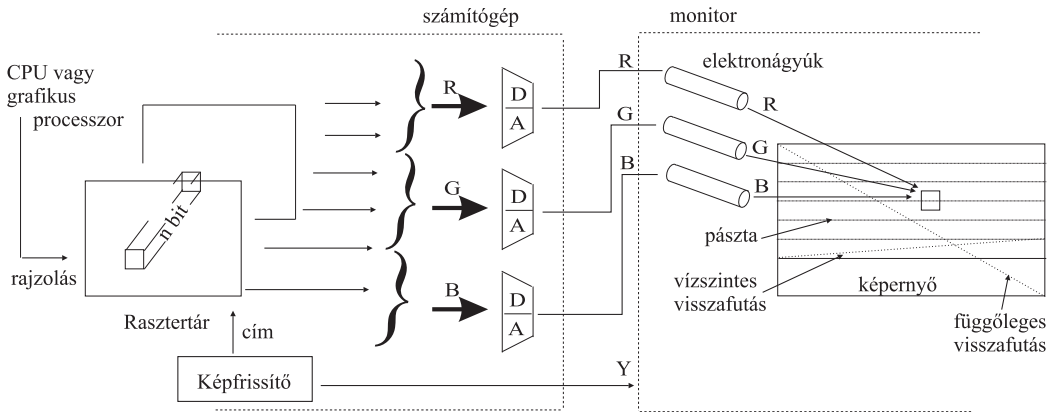
A grafikus program a képet a számítógép memóriájában állítja elő, amely alapján a megjelenítő elektronika vezérli a monitort. A kép memóriában lévő reprezentációját *digitális kép*nek nevezzük. A digitális kép a folytonos, 2 dimenziós képet véges számú elemből építi fel. Amennyiben az alapvető építőelem a szakasz, *vektorgrafikáról* beszélünk. Ha az építőelem téglalap alakú kicsiny terület, ún. *pixel*, akkor a rendszerünk *rasztergrafikus*. Jelen jegyzet csak a rasztergrafikus megjelenítőekkel foglalkozik. A *pixel* szó a *picture* és *element*, kép és elem angol szavak kompozíciója.

A digitális képben tárolt szakaszhalmazt, vagy pixelhalmazt a monitoron kell megjelenítenünk. A jelenleg használatos *katódsugár csöves monitor*ban a megjelenítő egyes pontjait vörös, kék és zöld színű fény emittálására bírhatjuk 3 elektronsugár segítségével. A három különböző hullámhosszon kibocsátott fotonok arányának változtatásával a szemben különböző színérzet keletkezik.

1.5. Rasztergrafikus rendszerek felépítése

A *rasztergrafikus rendszereknél* a kép szabályos négyzetrácsba szervezett pixelekből állítható össze. Az egyes pixelek színét meghatározó számot speciális memóriába, a *rasztertárba* kell beírni. Az elektronsugarak állandó pályát járnak be és egymás alatti vízszintes vonalakat húzva végigpásztázzák a képernyőt. A megjelenítő elektronika a pásztázás alatt rasztertár tartalom alapján modulálja az elektronsugarak intenzitását, ily módon kialakítva a képet.

Az 1.10. ábra egyszerű rasztergrafikus rendszert mutat be. A *grafikus processzor* a rasztertárat illeszti a számítógép rendszerbuszához és elvégzi az alacsony szintű rajzoló műveleteket. A legegyszerűbb rendszerekben a grafikus processzor el is maradhat, ilyenkor a számítógép központi processzora hajtja végre a rajzoló műveleteket is.



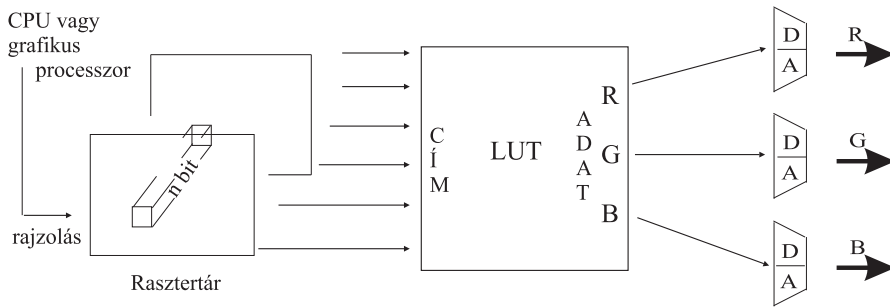
1.10. ábra. Rasztergrafikus rendszerek felépítése (valós szín mód)

A rasztertár olyan nagy kapacitású, speciális szervezésű memória, amely minden egyes pixel színét egy memóriaszóban tárolja. A szó szélessége (n) a legegyszerűbb rendszerekben általában 4, személyi számítógépekben 8, grafikus munkaállomásokban 12, 24 sőt 36, vagy 48. A pixel színének a szavakban tárolt biteken történő kódolására két módszer terjedt el.

1. **Valós szín mód** esetén a szót három részre osztjuk, ahol az egyes részek a vörös, zöld és kék színkomponensek színintenzitását jelentik.
2. **Indexelt szín mód**, vagy más néven *pseudo szín mód* esetén az egyes szavakat dekódoló memória, ún. *lookup tábla (LUT)* vagy *paletta* címeiként értelmezzük, és a vörös, zöld és kék komponensek tényleges intenzitásait ebben a dekódoló memóriában helyezzük el (1.11. ábra).

Ha a rasztertárban egy pixelhez n bit tartozik, akkor valós szín módban a megjeleníthető színek száma 2^n . Indexelt szín módban az egyszerre megjeleníthető színek száma ugyancsak 2^n , de, hogy melyek ezek a színek, az már a paletta tartalmától függ. Ha a palettában egy színkomponenst m biten ábrázolunk, akkor a lehetséges színek száma 2^{3m} .

Látnunk kell, hogy az indexelt szín mód egy pótmegoldás, hogy a pixelenként kevés bitet tartalmazó grafikus rendszerekben a rövidebb szabott takarónkat meghosszabbítsuk.



1.11. ábra. Indexelt szín módot használó rasztergrafikus rendszer felépítése

A monitor képének stabilizálásához a rasztartár tartalmát rendszeresen (legalább másodpercenként 50-100-szor) ki kell olvasni, és a képernyőre a képet újra fel kell rajzolni. A pixelek egymás utáni kiolvasását a *képernyőfrissítő egység* vezérli, amely szinkronizációs jeleket biztosít a monitor számára annak érdekében, hogy az elektron-sugár a pixelsor végén fusson vissza a képernyő bal szélső oldalára.

A monitor számára a digitális színinformációt analóg jellé kell átalakítani, amelyet három D/A átalakító végez el.

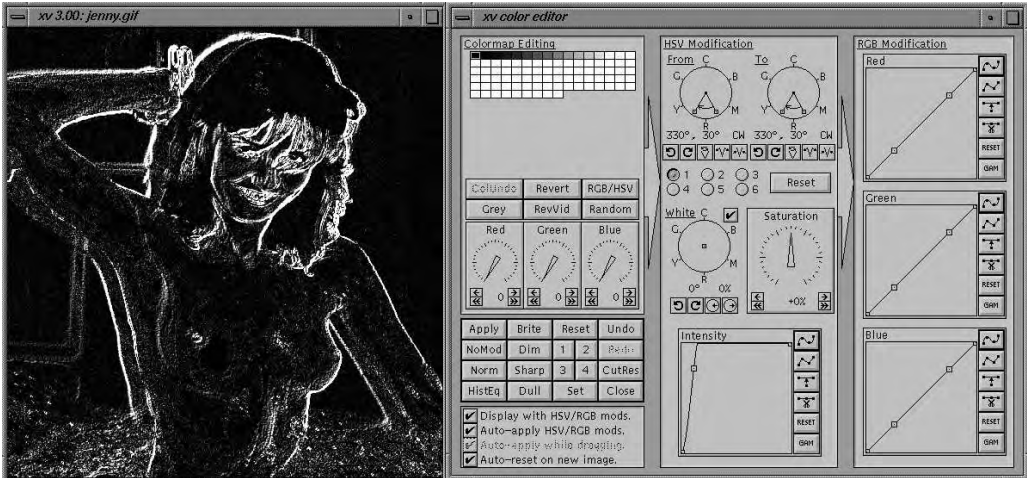
A pixel sorok és oszlopok száma definiálja a grafikus rendszer *felbontását*. Egy olcsóbb rendszerben a tipikus felbontás 640×480 , 1024×768 , a professzionális grafika pedig 1280×1024 , vagy 1600×1200 felbontással jellemezhető. Professzionális rendszerekben tehát a pixelek száma 1 millió felett van. Figyelembe véve, hogy egy másodperc alatt ezt az 1 millió pixelt legalább 50-szer kell kiolvasni, az egy pixel kiolvasására, dekódolására és digitál-analóg átalakítására kevesebb mint 20 nsec áll rendelkezésre.

A rasztartár mérete — színkomponensenként 8 bitet feltételezve $1280 \times 1024 \times 24$ bits ≈ 3 Mbyte — nem engedi meg, hogy ilyen nagysebességű memória áramkörökből építkezzünk. Szerencsére a rasztartárhozzáférés koherens jellege (mindig soronként, egy soron belül pedig egymás utáni pixelenként vesszük elő az adatokat a képernyőfrissítéshez) lehetővé teszi, hogy a pixeleket párhuzamosan olvassuk ki a rasztartárból. A kiolvasott pixelek egy shift-regiszterbe kerülnek, amelynek végén az egymás utáni pixelek már 20 nsec-ként kicsöpögtethetők.

1.6. A képek tárolása és utófeldolgozása

Azon túl, hogy a képszintézis által kiszámított kép általában a megjelenítőeszközezre kerül, gyakran felmerül az igény, hogy a képet egy állományba elmentsük és egy másik rendszerbe átvigyük. A képeket akkor vihetjük át másik rendszerbe, ha az állomány formátuma szabványos és a másik rendszer számára érthető. Számos különböző formátum

létezik, amelyek tárolhatják a kép méretét és a képpontok színértékeit tömörítetlen formában (*BMP*, *TARGA*, stb.) vagy különböző veszteség nélküli vagy veszteséges tömörítési módszerrel sűrítve (*GIF*, *TIFF*, *JPEG*, *PCX*, stb.). Egyes formátumokban (*MPEG*) nem csupán önálló képeket, hanem képsorozatot, animációkat is tárolhatunk.



1.12. ábra. Egy képfeldolgozó program kezelői felülete (xv)

A képfeldolgozó programok a grafikus rendszereknek egy különleges típusát jelentik. Ezen rendszerek bemeneti adatként egy digitális képet kapnak, amelyből általában transzformált képeket, ritkábban a képek alapján valamilyen geometriai információt állítanak elő [SP92].

1.7. Program: TARGA formátumú képállományok kezelése

Ebben a fejezetben TARGA formátumú képek kiírásához és beolvasásához adunk meg osztályokat. A *TGAOutputFile* osztály segítségével a képet egy *TARGA* formátumú képállományba menthetjük, a *TGAInputFile* felhasználásával pedig a képet az állományból betölthetjük. Egy TARGA formátumú állomány egy 18 bájtos fejrészrel kezdődik, ami tartalmazza a kép szélességét (*width*), magasságát (*height*), és az egyes pixelekhez tartozó bitek számát (jelen megoldásban egy pixelt 1 bájtos vörös (*r*), 1 bájtos zöld (*g*) és 1 bájtos kék (*b*) értékkel írunk le). A két implementációs osztály a konstruktorban kezeli a fájl fejrészét, a *Pixel* tagfüggvény pedig az egyes pixeleket kiírja, illetve beolvassa.

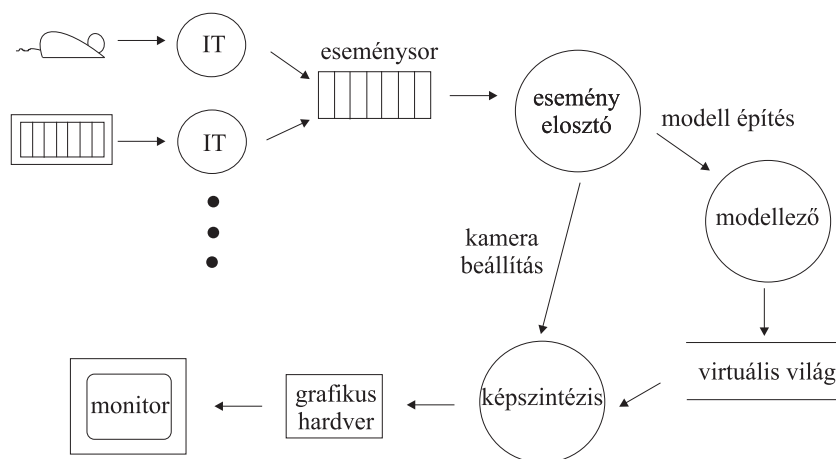
```
//=====
class TGAOutputFile {
//=====
    FILE * file;
public :
    TGAOutputFile( char * outputfilename, int width, int height ) {
        file = fopen(outputfilename, "wb");
        fputc(0,file); fputc(0,file); fputc(2,file);
        for(int i = 3;i < 12; i++) fputc(0,file);
        fputc(width & 0xff, file);
        fputc(width / 256, file);
        fputc(height & 0xff, file);
        fputc(height / 256, file);
        fputc(24,file); fputc(32,file);
    }
    void Pixel( double r, double g, double b ) {
        if (b > 1.0) b = 1.0; fputc(b * 255, file);
        if (g > 1.0) g = 1.0; fputc(g * 255, file);
        if (r > 1.0) r = 1.0; fputc(r * 255, file);
    }
    ~TGAOutputFile( ) { fclose(file); }
};

//=====
class TGAInputFile {
//=====
    FILE * file;
public :
    TGAInputFile( char * inputfilename, int& width, int& height ) {
        file = fopen(inputfilename, "rb");
        for(int i = 0;i < 12; i++) fgetc(file);
        width = fgetc(file) + fgetc(file) * 256L;
        height = fgetc(file) + fgetc(file) * 256L;
        fgetc(file); fgetc(file);
    }
    void Pixel( double& r, double& g, double& b ) {
        b = fgetc(file) / 255.0;
        g = fgetc(file) / 255.0;
        r = fgetc(file) / 255.0;
    }
    ~TGAInputFile( ) { fclose(file); }
};
```

2. fejezet

Grafikus szoftver alrendszerek felépítése

A 2.1. ábra egy tipikus interaktív grafikus program struktúráját mutatja be.

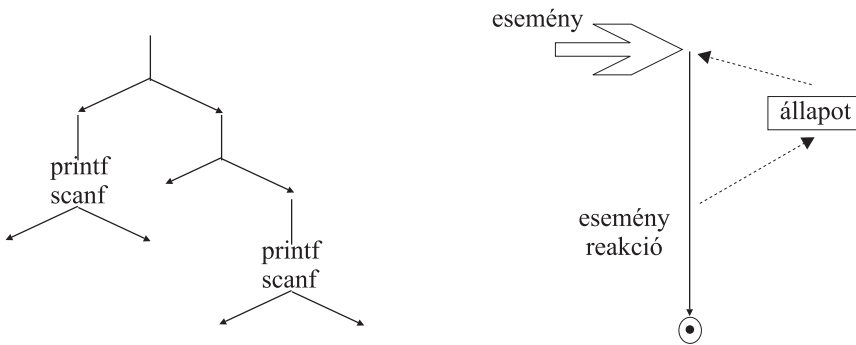


2.1. ábra. A grafikus szoftver felépítése

A felhasználó a *grafikus beviteli eszközök* segítségével avatkozhat be a program működésébe. A grafikus beviteli eszközök 2D abszolút vagy relatív pozíció adatot, 3D abszolút vagy relatív pozíció adatot vagy karaktersorozatokat szolgáltathatnak. A beviteli eszközöket megszakítási rutinok illesztik a programhoz. A megszakítási rutinok a beviteli eszközök eseményeit egy vagy több *esemény sorba* pakolják. A grafikus program ezt az esemény sort figyeli, és ha abban megjelenik valami, akkor reagál rá.

2.1. Programvezérelt és eseményvezérelt interakció

A felhasználói beavatkozások kezelésére alapvetően két programozási technikát használhatunk. A hagyományos ún. *programvezérelt interakció*ban a program tölti be az irányító szerepet, a felhasználó pedig válaszol a feltett kérdésekre. Amikor a számítások során a programnak új bemeneti adatra van szüksége, erről értesítést küld a felhasználónak, majd addig várakozik, amíg az választ nem ad a kérdésre. A jól ismert `printf-scanf` C függvénypár ennek tipikus megvalósítása. Ebben az esetben a begépett karakterek értelmezéséhez szükséges állapotinformációt (például a 123 valakinek a kora vagy egy banki átutalás összege) az határozza meg, hogy pontosan hol tartunk a program végrehajtásában. A programvezérelt interakció alapvető hiányossága, hogy egyszerre egyetlen beviteli eszköz kezelésére képes. Ha ugyanis a program felteszi a kérdését a felhasználónak, akkor addig nem lép tovább, amíg a `scanf` függvény vissza nem tér a kérdésre adott válasszal, így ezalatt rá sem nézhet a többi beviteli eszközre. A másik fő probléma, hogy a felhasználói kommunikáció és a program feldolgozó része nem válik el élesen egymástól. Emiatt a felhasználói kommunikációban nem használhatunk előre definiált magas szintű könyvtári szolgáltatásokat. Következésképpen csak korlátozott minőségű felhasználói kommunikáció valósítható meg elfogadható ráfordítás árán.



2.2. ábra. Programvezérelt és eseményvezérelt programok szerkezete

Az *eseményvezérelt interakció*ban a felhasználó irányít, a program passzívan reagál a felhasználói beavatkozásokra. A program nem vár egyetlen eszközre sem, hanem periodikusan teszteli, hogy valamelyik eszközön történt-e esemény, így tetszőleges számú beviteli eszköz is kezelhető. Minden pillanatban a felhasználó választhat, hogy melyik beviteli eszközt használja. Ebben az esetben az esemény értelmezését nem végezhetjük el aszerint, hogy éppen hol tartunk a programban, hiszen az eseményt mindig ugyanott, az eseménysor tesztelésénél kezdjük feldolgozni. Az események értelmezéséhez szük-

séges állapotinformációt explicit módon, változóknak kell tárolni. Vegyük észre, hogy az eszközök tesztelése és az eseménysor kezelése, sőt bizonyos alapvető eseményekre elvárt reakció (például az egér mozgásakor a kurzort is mozgatni kell) független az alkalmazástól, ezért ezt egyszer kell megvalósítani és egy könyvtárban elérhetővé tenni. Az alkalmazásfüggő rész az egyes eseményekre reagáló rutinok gyűjteménye (szokásos elnevezések az esemény kezelő vagy *trigger*). Ez egyrészt előnyös, mert az alkalmazói program fejlesztőjét megkímélhetjük az interakció alapvető programjainak a megírásától. Másrészt viszont felborul az a jól megszokott világképünk, hogy a belépési ponttól kezdve a program egy jól meghatározott, átlátható szálon fut végig. A programfejlesztő szempontjából az eseményvezérelt rendszerek különálló triggerek látszólag független gyűjteményei, ahol még azt sem mindig mondhatjuk meg, hogy ezeket milyen sorrendben hajtja végre a program. Az eseményvezérelt rendszerek programozása tehát nehezebb, de a nagyobb odafigyelés feltétlenül megtérül. Ezt bizonyítja az a tény is, hogy a modern, interaktív szoftvereket előállító eszközök mind az eseményvezérelt filozófiát követik.

A eseményekre reagáló program egyrészt modellezési feladatokat végezhet, azaz megváltoztathatja a virtuális világot reprezentáló adatstruktúrát (vagy adatbázist), másrészt módosíthatja a kamera paramétereit. Mindkét esetben a képet újra kell számítani a képszintézis alrendszer segítségével. A program az újrászámított képet a grafikus megjelenítő eszközön mutatja meg a felhasználónak.

2.2. A grafikus hardver illesztése

A program a grafikus hardver szolgáltatásait a *grafikus könyvtárak* segítségével érheti el. A grafikus könyvtárak általában hierarchikus rétegeket képeznek, és többé-kevésbé szabványosított interfésszel rendelkeznek. A grafikus könyvtárak kialakításakor igyekeznek követni a *logikai ki-bevitel* és a *rajzolási állapot* elveit.

A logikai ki-bevitel azt jelenti, hogy a műveletek paraméterei nem függenek a hardver jellemzőitől, így az erre a felületre épülő program hordozható lesz. A koordinátákat például a megjelenítőeszköz felbontásától függetlenül célszerű megadni, a szint pedig elvonatkoztatva az egy képponthez tartozó rasztertárbeli bitek számától.

A rajzolási állapot használatához az a felismerés vezet, hogy már az olyan egyszerűbb grafikus primitívek rajzolása is, mint a szakasz, igen sok jellemzőtől, ún. *attribútumtól* függhet, például a szakasz színétől, vastagságától, mintázatától, a szaggatási közők színétől és átlátszóságától, 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 érvényben ma-

rad, amíg meg nem változtatjuk azokat. Az attribútumok kezelése a rajzolóparancsoktól elkülönített attribútumállító függvényekkel lehetséges.

Számos grafikus könyvtár ismeretes, amelyek kapcsolódhatnak az operációs rendszerhez (*Ms-Windows GDI*, *X-Window*, a hardverhez (*OpenGL*, *Starbase*, *TEK-STI*), a programozási nyelvhez (*ObjectWindows*, *MFC*), de léteznek szabványos, gyártófüggetlen interfészek is (*GKS*, *CGI*, *PHIGS*, stb.). Az elérhető szolgáltatások köre a szakaszrajzolástól egészen a textúrákkal kiegészített 3D felületek megjelenítéséig terjedhet.

2.3. Program: egy egyszerű grafikus könyvtár

A *grafikus könyvtárak* általában a bemeneti események feldolgozását az esemény elosztóig bezárólag végzik el, a grafikus megjelenítő vezérlését pedig az alacsony szintű primitívek megjelenítésétől kezdve vállalják fel. A következőkben egy egyszerű grafikus könyvtárat mutatunk be, amely a beviteli eszközöket eseményvezérelten kezeli, a grafikus kimenetet pedig mind fizikai, mind pedig logikai módon illeszti. A logikai szint a fizikai szintre épül, elfedve annak hardverfüggő sajátosságait.

A könyvtárunktól mindössze a pont és a szakasz rajzolását várjuk el. A pont és a szakasz attribútumai a pont illetve a szakasz színe és a *raszter operáció*, amely azon logikai műveletet határozza meg, amelyet a rasztertár eredeti tartalmára és a szakasz színére végre kell hajtani, hogy a rasztertár új értékét előállítsuk. Tekintsük továbbá a kezdőpont koordinátáit is a szakasz attribútumának, így a szakaszrajzolás függvényben csak a végpont szerepel. A könyvtár rutinjai a fizikai szinten a következők:

```
typedef int   PCoord;
typedef long  PColor;

void Pixel(PCoord X, PCoord Y, PColor color);
void PLine(PCoord X, PCoord Y);
void PMove(PCoord X, PCoord Y);
void PSetColor( PColor color );
```

Ezekben a rutinokban az *X, Y* koordináták a pixelkoordinátákban értendők. A fizikai címek használatához szükséges lehet a megjelenítőeszköz felbontásának lekérdezése is:

```
void GetResolution( PCoord& px, PCoord& py );
```

A raszteroperációk közül a rasztertárnak az új színnel történő átírását (*SET*) és az új és a régi szín bitenkénti modulo 2 összegét (ún. kizáró vagy, illetve *XOR*) engedjük meg.

```
typedef enum {SET, XOR} ROP;

void RasterOp( ROP rop );
```

A logikai ki-bevitel elveinek megfelelően a rutinok paraméterezése nem függhet az aktuális eszköz fizikai jellemzőitől, mint például a felbontástól, vagy az egy pixelhez tartozó bitek számától. A koordináták logikai megadásának egyik lehetséges módja, ha a paraméterek a teljes ablakméret arányában adják a kívánt pozíciót, így a koordináták egy $[0...1]$ tartományban lévő értéket vehetnek fel. A logikai koordinátákat `Coord` típusal definiáljuk. Az eszközkoordináták maximális értékének lekérdezéséhez létrehozuk a `GetDevice(Coord& x, Coord& y)` függvényt. A kimenetet valóban logikai szinten kezelő könyvtáraknál tehát ez az `x` és `y` változóba 1 értéket ír. Fizikai szintű eszközkezelés esetén a változóba a vízszintes és a függőleges felbontás kerül. A szint logikai módon a rendszerben elérhető maximum értékre vetített relatív `R`, `G`, `B` értékekkel adhatjuk meg. Összefoglalva a könyvtár logikai szintű szolgáltatásai:

```
typedef double Coord;
typedef struct { double R, G, B; } Color;

void GetDevice(Coord& x, Coord& y);
void Pixel(Coord x, Coord y, Color color);
void DrawLine(Coord x, Coord y);
void Move(Coord x, Coord y);
void SetColor(Color color);
void Clear();
```

Az eseményvezérelt filozófiának megfelelően a bemeneti eseményekről a könyvtár értesíti az alkalmazást, szemben a programvezérelt megoldással, amikor az alkalmazás rákérdez, hogy történt-e bemeneti esemény. Az egyszerű könyvtárunk egér és billentyűzet eseményeket kezel. A könyvtár egy billentyű lenyomásakor a `KeyboardEvent` üzenetet küld az alkalmazásnak, a bal egérgomb lenyomásakor egy `MouseLeftBtnDown` üzenetet, az egérgomb elengedésekor egy `MouseLeftBtnUp` üzenetet, végül az egér mozgásakor `MouseMove` üzeneteket. A `KeyboardEvent` üzenet paramétere a lenyomott billentyű ASCII kódja, az egérüzenetek paramétere pedig a kurzor aktuális pozíciója. Előfordulhat, hogy valamilyen ok miatt — például a felhasználó egy másik ablakot húzott el ezen ablak előtt — az ablak tartalma érvénytelenné válik, és ezért újra kell rajzolni. Erről a könyvtár a `ReDraw` üzenettel értesítheti az alkalmazást.

Összefoglalva a következő rutinokat az alkalmazásban kell implementálni, és ezeket a könyvtár hívja a megfelelő események bekövetkeztekor:

```
void KeyboardEvent( int keyASCII );
void MouseLeftBtnDown( Coord x, Coord y );
void MouseLeftBtnUp( Coord x, Coord y );
void MouseRightBtnDown( Coord x, Coord y );
void MouseRightBtnUp( Coord x, Coord y );
void MouseMove( Coord x, Coord y );
void ReDraw( );
```

Objektum-orientált környezetekben a grafikus kimenethez kapcsolódó műveleteket általában egyetlen ablakosztályban `Window` foglaljuk össze. A konkrét alkalmazás ebből örökléssel hozza létre az elvárt működésnek megfelelő ablakot. Mivel az ablakok “egyénsége” abból adódik, hogy a bemeneti eseményekre másképpen reagálnak, az öröklés során a eseménykezelő virtuális függvényeket át kell definiálni. Az új függvények természetesen használhatják a könyvtárban megírt rajzoló parancsokat.

```
//=====
class Window {
//=====
    void GetDevice( Coord& x, Coord& y );
    void Pixel(Coord x, Coord y, Color c);
    void SetColor( Color c );
    void Move(Coord x, Coord y);
    void DrawLine(Coord x, Coord y);
    void RasterOp( ROP r );
    void Clear( );

    virtual void KeyboardEvent( int keyASCII ) {}
    virtual void MouseLeftBtnDown( Coord x, Coord y ) {}
    virtual void MouseLeftBtnUp( Coord x, Coord y ) {}
    virtual void MouseRightBtnDown( Coord x, Coord y ) {}
    virtual void MouseRightBtnUp( Coord x, Coord y ) {}
    virtual void MouseMove( Coord x, Coord y ) {}
    virtual void ReDraw( ) {}
public:
    Window( ) { pwindow = this; }
    void Execute( );
};

extern Window * pwindow = NULL;
```

Végül a program indulásakor a könyvtár az alkalmazás inicializálásához meghív egy `AppStart` függvényt, ami az alkalmazás belépési pontjának tekinthető. Az `AppStart` létrehozza az alkalmazói ablak egy példányát, és a könyvtár `Execute` függvényével beindítja az üzenetsor ciklikus lekérdezését:

```
class MyWindow : public Window { ... };

void AppStart( ) {
    MyWindow win;
    win.Execute( );
}
```


2.3.1. A logikai és a fizikai szintek összekapcsolása

A logikai szinten elérhető szolgáltatások a fizikai szint szolgáltatásaira épülnek. Ehhez a bemeneti láncon a fizikai eszközkordinátákat logikai eszközkordinátákra kell alakítani, a kimeneti láncon pedig éppen fordítva, a logikai koordinátákat vissza kell alakítani eszközfüggő értékekre.

A transzformációk elvégzéséhez feltételezzük, hogy az eszköz fizikai felbontásának megfelelően már kitöltöttük a `device` téglalap változót, amivel az átalakítás már könnyen elvégezhető:

```
typedef struct { int left, top, right, bottom; } RECT;
RECT device;

void Physical2LogicalCoord( PCoord X, PCoord Y, Coord& x, Coord& y ) {
    x = (Coord)(X - device.left) / (device.right - device.left);
    y = (Coord)(Y - device.bottom) / (device.top - device.bottom);
}

void Logical2PhysicalCoord( Coord x, Coord y, PCoord& X, PCoord& Y ) {
    X = x * (device.right - device.left) + device.left;
    Y = y * (device.top - device.bottom) + device.bottom;
}
```

A színek átalakításához a paletta azon bejegyzéseit kell azonosítani, amelyek a leginkább hasonlítanak a megjelenítendő színre. Amennyiben a paletta a BLACK, BLUE, GREEN, RED, YELLOW, MAGENTA, CYAN és WHITE sorokban rendre a fekete, kék, zöld, piros, sárga, magenta, cián és fehér színeket tartalmazza, akkor a konverzió a következőképpen végezhető el:

```
int Logical2PhysicalColor( Color c ) {
    if (col.R <= 0.5 && col.G <= 0.5 && col.B <= 0.5) return BLACK;
    if (col.R < 0.5 && col.G < 0.5 && col.B >= 0.5) return BLUE;
    if (col.R < 0.5 && col.G >= 0.5 && col.B < 0.5) return GREEN;
    if (col.R >= 0.5 && col.G < 0.5 && col.B < 0.5) return RED;
    if (col.R >= 0.5 && col.G >= 0.5 && col.B < 0.5) return YELLOW;
    if (col.R >= 0.5 && col.G < 0.5 && col.B >= 0.5) return MAGENTA;
    if (col.R < 0.5 && col.G >= 0.5 && col.B >= 0.5) return CYAN;
    if (col.R >= 0.5 && col.G >= 0.5 && col.B >= 0.5) return WHITE;
}
```

Ezek felhasználásával a kimenetet logikai szinten kezelő szolgáltatásokat a fizikai szintű szolgáltatásokra vezethetjük vissza:

```

void Window :: Move( Coord x, Coord y ) {
    PCoord X, Y; Logical2PhysicalCoord( x, y, X, Y ); PMove( X, Y );
}

void Window :: DrawLine( Coord x, Coord y ) {
    PCoord X, Y; Logical2PhysicalCoord( x, y, X, Y ); PLine( X, Y );
}

void Window :: Pixel( Coord x, Coord y, Color col ) {
    PCoord X, Y; Logical2PhysicalCoord(x, y, X, Y); Pixel(X, Y, col);
}

void Window :: SetColor( Color col ) {
    PSetColor( Logical2PhysicalColor( col ) );
}

```

2.3.2. A könyvtár megvalósítása DOS operációs rendszer alatt

A billentyűzet és az egér illesztését elvégezhetjük a hardver portok fizikai kezelésével is, de jelentős fáradságot takarítunk meg, ha kihasználjuk azt, hogy a DOS/BIOS operációs rendszer, illetve a C könyvtár már számos dolgot megvalósít a billentyűzet és az egér kezeléséből. A billentyűzetet kezelő operációs rendszer szolgáltatásokat legkönnyebben C könyvtári rutinokon keresztül érhetjük el. A kbhit rutin ellenőrzi, hogy történt-e klaviatúra esemény, a getch rutin pedig visszaadja a leütött billentyű kódját.

Az egérkezelő DOS hívásokhoz sajnos nem tartoznak C könyvtári függvények, ezért a gépi kódú részleteket magunknak kell a magas szintű nyelvhez illeszteni. Az egérkezelő funkciók a 33h DOS híváshoz kapcsolódnak. Például az egér pillanatnyi, a képernyő pxelegységeiben mért pozícióját és a gombok státuszát a következő rutinnal kaphatjuk meg:

```

#define IRET                0xCF
static REGS regs;
#define REG( r ) regs.x.##r
#define MOUSE_IT   int86(0x33, &regs, &regs)

//-----
void getmouse( int * px, int * py, int * pstat ) {
//-----
    REG(ax) = 3;                // funkció = státusz lekérdezés
    MOUSE_IT;
    *pstat = REG(bx);          // gomb státusz: 0. bit bal, 1. bit jobb
    *px    = REG(cx);          // X pozíció pixel koordinátákban
    *py    = REG(dx);          // Y pozíció pixel koordinátákban
}

```

A rutin az egér aktuális pozícióját a pX és pY paraméterek által megcímzett változóba, a bal gomb státuszát pedig pstat című változóba teszi.

A fizikai eszközkoordinátákat logikai eszközkoordinátákká kell alakítani. A következő programrészlet BGI grafikus szolgáltatásokkal a fizikai felbontásnak megfelelően kitölti a `device` téglalap változót:

```
device.left    = 0; device.right = getmaxx( );
device.bottom = 0; device.top   = getmaxy( );
```

A beviteli eszközöket pediorikusan tesztelő főciklus ugyancsak a könyvtárba kerül:

```
//-----
void Window :: Execute( ) {
//-----
    PCoord X, Y, X_old, Y_old;
    int stat, leftstat_old = 0, rightstat_old = 0;

    for( ; ; ) {                                // főciklus
        if ( kbhit( ) ) {                        // billentyűzet tesztelése
            int c = getch();                     // billentyűzet lekérdezése
            KeyboardEvent( c );
        }

        getmouse(&X, &Y, &stat);                // egér státusz lekérdezése
        int leftstat = stat & 1;                 // bal gomb
        int rightstat = stat & 2;                // jobb gomb
        Coord x, y;
        Physical2LogicalCoord( X, Y, x, y );    // átalakítás logikai koordinátákká

                                                // ha a koordináta változott ...
        if ( X != X_old || Y != Y_old ) pwindow -> MouseMove( x, y );

        if ( leftstat != leftstat_old ) { // ha a bal gomb státusza változott ...
            if ( leftstat > 0 ) pwindow -> MouseLeftBtnDown( x, y );
            else pwindow -> MouseLeftBtnUp( x, y );
        }
        if ( rightstat != rightstat_old ) { // ha a bal gomb státusza változott ...
            if ( rightstat > 0 ) pwindow -> MouseRightBtnDown( x, y );
            else pwindow -> MouseRightBtnUp( x, y );
        }

                                                // felkészülünk a következő ciklusra
        X_old = X; Y_old = Y;
        leftstat_old = leftstat, rightstat_old = rightstat;
    }
}
```

A grafikus üzemmód be- és kikapcsolását, valamint a fizikai szintű grafikus kimeneti rutinokat például a *BGI grafikus könyvtár* [SP92] szolgáltatásaira építve valósíthatjuk meg.

```
#include <graphics.h>

void InitGraph( ) {
    int    GraphDriver = DETECT;
    int    GraphMode;
    initgraph( &GraphDriver, &GraphMode, "." );
    device.left    = 0; device.right = getmaxx( );
    device.bottom = 0; device.top = getmaxy( );
}

void CloseGraph( ) { closegraph(); }
void Pixel( PCoord X, PCoord Y, PColor color ) { putpixel(X, Y, col); }
void PSetColor( PColor col ) { setcolor( col ); }
void PMove( PCoord X, PCoord Y ) { moveto(X, Y); }
void PLine( PCoord X, PCoord Y ) { lineto(X, Y); }
```

2.3.3. A könyvtár megvalósítása Ms-Windows környezetben

Az Ms-Windows már maga is egy eseményvezérelt grafikus könyvtárat tartalmaz, ezért a könyvtárunk megvalósítása során csak az üzenet- és paraméterkonverzióval kell megbirkóznunk.

Az Ms-Windows minden olyan helyzetben, amikor a programunktól valamilyen reakciót vár el, egy üzenetet továbbít az alkalmazás ablakkezelő függvényének. Az ablakkezelő függvény egy lehetséges kialakítása az alábbi:

```
#include <windows.h>

static HDC    hdc;                // attribútum tábla azonosító
char          szClassName[] = "grafika"; // ablak osztály neve

//-----
long FAR PASCAL WndProc(HWND hwnd, WORD wmsg, WORD wPar, LONG lPar) {
//-----
    PAINTSTRUCT    ps;

    switch ( wmsg ) {
        case WM_PAINT:                // Ablak tartalma érvénytelen
            GetClientRect( hwnd, &device ); // ablakméret lekérdezése
            hdc = BeginPaint(hwnd, &ps);
            if (pwindow) pwindow -> ReDraw( );
            EndPaint( hwnd, &ps );
            break;

        case WM_LBUTTONDOWN:         // Bal egérgomb lenyomás
        case WM_LBUTTONUP:           // Bal egérgomb elengedés
        case WM_RBUTTONDOWN:         // Jobb egérgomb lenyomás
        case WM_RBUTTONUP:           // Jobb egérgomb elengedés
```

```

case WM_MOUSEMOVE:           // Egér mozgatás
    hdc = GetDC( hwnd );      // attribútum tábla lekérés
    PCoord X = LOWORD(lPar);   // az esemény fizikai koordinátái
    PCoord Y = HIWORD(lPar);
    Coord x, y;
    Physical2LogicalCoord( X, Y, x, y );    // átalakítás logikai koordinátává
    switch ( wmsg ) {          // az alkalmazás eseménykezelőjének átadjuk
    case WM_LBUTTONDOWN:      pwindow -> MouseLeftBtnDown(x, y); break;
    case WM_LBUTTONUP:        pwindow -> MouseLeftBtnUp(x, y); break;
    case WM_RBUTTONDOWN:      pwindow -> MouseRightBtnDown(x, y); break;
    case WM_RBUTTONUP:        pwindow -> MouseRightBtnUp(x, y); break;
    case WM_MOUSEMOVE:        pwindow -> MouseMove(x, y);      break;
    }
    ReleaseDC( hwnd, hdc );    // attribútum tábla felszabadítás
    break;

case WM_CHAR:                 // klaviatúra esemény
    hdc = GetDC( hwnd );
    pwindow -> KeyboardEvent( wPar );      // átadjuk az alkalmazásnak
    ReleaseDC( hwnd, hdc );
    break;

case WM_COMMAND:              // Menü elem kiválasztás esemény
    hdc = GetDC( hwnd );
    pwindow -> MenuCommand( wPar );        // átadjuk az alkalmazásnak
    ReleaseDC( hwnd, hdc );
    break;

default:                       // Minden más eseményre az alapértelmezésű reakció
    return DefWindowProc( hwnd, wmsg, wPar, lPar );
}
return 0;
}

```

Ez az ablakkezelő függvény az ablak érvényesítése miatti újrarajzolás (WM_PAINT) esemény hatására az alkalmazói ablak `Redraw` függvényét aktivizálja. Ebben az üzenetben az ablakkezelő függvény felméri az aktuális ablak méreteit, és az eredményt a `device` változóba írja. Ezt a méretet használja a program a fizikai-logikai eszközkordináta transzformációk során.

Az ablakkezelő függvény az egérrel kapcsolatos eseményeknél (WM_LBUTTONDOWN, WM_LBUTTONUP, WM_RBUTTONDOWN, WM_RBUTTONUP, WM_MOUSEMOVE) először a fizikai eszközkordinátákat logikai eszközkordinátákká alakítja, majd az egér bal illetve jobb gombjának megnyomása esetén meghívja az alkalmazói ablak az adott eseménynek megfelelő rutinját. Végül a WM_CHAR klaviatúra eseményt ugyancsak az alkalmazói ablak tagfüggvényéhez továbbítja.

A grafikus rajzoló parancsok:

```
void Pixel( PCoord X, PCoord Y, PColor color )
    SetPixel( hdc, X, Y, color );
}
void PMove( PCoord X, PCoord Y ) { MoveTo( hdc, X, Y ); }
void PLine( PCoord X, PCoord Y ) { LineTo( hdc, X, Y ); }

void PRasterOp( ROP r ) {
    switch ( r ) {
        case SET: SetROP2( hdc, R2_COPYPEN ); break;
        case XOR: SetROP2( hdc, R2_XORPEN ); break;
    }
}
```

Az inicializáló rész az üzenetciklussal:

```
//-----
void InitWindowClass( HANDLE hInstance, HANDLE hPrevInstance ) {
//-----
    WNDCLASS wndclass;
    strcpy( szClassName, "grafika" );
    if ( !hPrevInstance ) {
        wndclass.style           = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfnWndProc     = WndProc;           // ablakkezelő függvény
        wndclass.hInstance      = hInstance;        // program azonosító
        wndclass.hIcon           = LoadIcon( hInstance, IDI_APPLICATION );
        wndclass.hCursor         = LoadCursor( NULL, IDC_ARROW );
        wndclass.hbrBackground   = GetStockObject( WHITE_BRUSH );
        wndclass.lpszMenuName    = "windowsmenu";    // menünév az erőforrás fájlban
        wndclass.lpszClassName   = szClassName;     // osztálynév
        wndclass.cbClsExtra      = 0;
        wndclass.cbWndExtra      = 0;
        if ( !RegisterClass( &wndclass ) ) exit( -1 );
    }
}

//-----
void InitWindow( HANDLE hInstance, int nCmdShow ) {
//-----
    HWND hwnd = CreateWindow( szClassName,           // osztálynév
                              "grafika",           // megfogó csík
                              WS_OVERLAPPEDWINDOW, // az ablak stílusa
                              CW_USEDEFAULT,       // kezdeti x pozíció
                              CW_USEDEFAULT,       // kezdeti y pozíció
                              CW_USEDEFAULT,       // kezdeti x méret
                              CW_USEDEFAULT,       // kezdeti y méret
                              NULL,                // szülő ablak azonosító
                              NULL,                // menü azonosító, ha nem az osztályé
                              hInstance,           // program azonosító
```

```

                NULL );           // paraméterlista vége
    if ( ! hwnd ) exit( -1 );
    ShowWindow(hwnd, nCmdShow );   // ablak megjelenítése
    UpdateWindow( hwnd );         // érvénytelenítés
}

//-----
// egy Windows program itt indul
int PASCAL WinMain( HANDLE hInstance,           // program azonosító
                   HANDLE hPrevInstance,       // ugyanezen osztály már futó példánya
                   LPSTR lpszCmdLine,          // parancssor argumentumok
                   int nCmdShow ) {           // ablak megjelenése
//-----
    InitWindowClass(hInstance, hPrevInstance); // ablak osztály inicializálás
    InitWindow( hInstance, nCmdShow );         // ablak példány inicializálás
    AppStart( );                               // alkalmazás indítás
    return 0;
}

//-----
void Window :: Execute( ) {                   // eseménykezelés
//-----
    MSG msg;
    while( GetMessage( &msg, NULL, 0, 0 ) ) { // esemény hurok
        TranslateMessage( &msg );           // klaviatúra esemény konverzió
        DispatchMessage( &msg );           // esemény átadása az alkalmazásnak
    }
}

```

Az `InitWindowClass` és `InitWindow` függvények részletes megvalósítása bármilyen Windows programozással kapcsolatos irodalomban megtalálható.

2.3.4. Programtervezés eseményvezérelt környezetekben

Az interaktív rendszerek modellezésének, tervezésének és implementációjának alapelveit egy egyszerű feladat megoldásával mutatjuk be. A példa különböző színű szakaszokat helyez el a képernyőre “*gumivonal*” technikával. A felhasználó a gumivonal módszerrel egy szakaszt úgy definiálhat, hogy a szakasz egyik végpontján lenyomja az egér gombját, majd lenyomott gombbal elkezd mozgatni az egeret. A kijelölt kezdőpont és az aktuális pozíció közé a program egy ideiglenes szakaszt húz, ami követi az egér mozgását. A szakasz a gumi formából akkor merevedik meg, ha elengedjük az egérgombot. Ebben a pillanatban a program a végleges színnel felrajzolja a szakaszt a képernyőre, majd felkészül a következő szakasz fogadására.

Egy program tervezése során a programunk adatait, funkcionalitását és dinamikáját kell kialakítanunk [LKSK95]. A funkcionalitáson azon feldolgozási lépéseket (ügyne-

vezett transzformációkat) értjük, amelyek a bemeneti adatokból előállítják a kimeneti adatokat. A dinamika a feldolgozási lépések időbeliségét határozza meg. Interaktív rendszerekben különösen nagy jelentősége van a rendszer dinamikájának, hiszen a felhasználók minden pillanatban nagyon sokféle igényt támaszthatnak a rendszerrel szemben, ezért a feldolgozási lépéseket nagyon sok különféle sorrendben kell végrehajtani.

A grafikus rendszerekben az adat-, a funkcionális- és a dinamikus-modell hármását még a program megjelenési modellje egészíti ki, amely rendelkezik arról, hogy a program milyen látványt produkáljon a felhasználó számára a képernyőn. A megjelenési modellt az adott grafikus felület építőelem készletéből (*widget*) alakíthatjuk ki.

Az interaktív rendszerek specifikálása általában a probléma szereplőinek azonosításával, azaz az adatmodell kialakításával kezdődik. A megoldandó feladatunkban ilyen szereplők a *felhasználó* (User) (ő minden interaktív rendszerben jelen van) és a rajzolás alatt álló *szakasz* (Line).

Az eseményvezérelt rendszereket a felhasználói igények működtetik, lényegében mindent a felhasználói beavatkozások, események indítanak el. Ezért a következő lépés a felhasználó és a program közötti lehetséges párbeszéd típusok megfogalmazása. Ez a párbeszéd olyan, mintha a felhasználó a fejében futtatná le a “programot”, amely során a programból rutinokat hívogat. A példában három *dialogus* szerepel, nevezetesen “egy szakasz felvétele”, a “rajzolási szín beállítása” és a “képernyő törlése”. Egy szakasz felvétele során a felhasználói események sorrendje és a program reakciói az egyes eseményekre a következők:

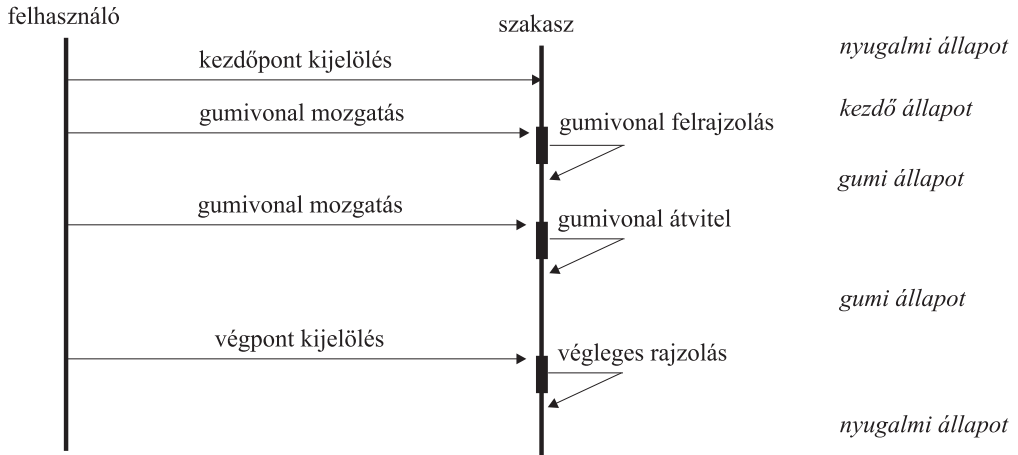
1. Kezdőpont kijelölés
2. Gumivonal mozgatás: gumivonal felrajzolás
3. Gumivonal mozgatás: gumivonal áthelyezés
4. ...
5. Gumivonal mozgatás: gumivonal áthelyezés
6. Végpont kijelölés: végleges felrajzolás a rajzolási színnel

Ebben a *forgatókönyvben* az egyes bejegyzésekben a felhasználói eseményeket és a program reakcióit kettősponttal választottuk el.

A szín beállítása és képernyő törlés triviális dinamikával rendelkezik, hiszen a felhasználó kiadja az ilyen értelmű parancsot, a program pedig tudomásul veszi azt.

Mivel a párbeszéd a felhasználói felület szintjén definiálják a rendszert, a párbeszédet ki kell terjeszteni a rendszer belső objektumaira is, azaz meg kell mondani, hogy egy a felhasználói felületen megjelent igény kielégítésében a program objektumai

miként vesznek részt. A program objektumainak együttműködését *kommunikációs diagramokon* írjuk le, amelyek azt mutatják be, hogy egy felhasználói esemény melyik objektumhoz jut el, és az milyen más objektumoknak üzen a feladat elvégzése során.



2.3. ábra. A gumivonal rajzolás kommunikációs diagramja

Az objektumok közötti párbeszédék feltérképezése után az egyes objektumok belső működésének modellezése következik. Miként azt a kommunikációs diagram bemutatja, egy objektum az őt ért üzenet hatására más objektumoknak üzenhet. Előfordulhat, hogy a küldött üzenet nem csak az utolsó kapott üzenet függvénye, hanem a korábbi üzenetek is befolyásolják a viselkedést. Az objektum tehát emlékeket hordozhat a korábbi üzenetekről. Az emlékeket az objektum *állapotában* tároljuk. Első megközelítésben a kommunikációs diagramon minden két kapott üzenet közötti időt az objektum egy új állapotának tekintjük. Az állapotváltozásokat és az állapotokhoz kapcsolódó tevékenységeket pedig az objektum állapotáblájában foglaljuk össze. Az *állapottábla* oszlopai az üzeneteket, a sorai pedig a lehetséges állapotokat jelképezik, az egyes táblázatelemekbe pedig az adott állapotban elvégzendő tevékenységet és a következő állapotot írjuk be.

Figyeljük meg, hogy az állapotábla kitöltését soronként végezzük, hiszen a kommunikációs diagramról az olvasható le, hogy az objektum egy adott állapotból milyen következő állapotba jut a különböző üzenetek hatására. A szakasz kommunikációs diagramjának vizsgálata során három állapotot különböztethetünk meg: “nyugalmi” (IDLE) állapotot, amikor nem rajzolunk, a kezdőpont kijelölését szimbolizáló “kezdő” (START) állapotot, és a vonalat gumiként húzogató “gumi” (RUBBER) állapotot. Az állapotok tevékenységének leírása során a Set start és Set end a szakasz kezdő és végpontjának beállítását, a DrawRubber függvény a szakasz XOR módban történő felrajzolá-

| üzenet állapot | kezdőpont kijelölés | gumivonal mozgatás | végpont kijelölés |
|-------------------|------------------------|---------------------------------------|---------------------------------|
| nyugalmi | Set start | | |
| | kezdő | | |
| kezdő | | Set end DrawRubber | nyugalmi |
| | | gumi | |
| gumi | | DeleteRubber Set end DrawRubber | DeleteRubber Set end Draw |
| | | gumi | nyugalmi |

2.4. ábra. A szakasz állapotátlája

sát, a DeleteRubber függvény egy másodszeri XOR típusú felrajzolással a szakasz letörlését, a Draw pedig a SET típusú felrajzolását jelenti.

Az állapotátlák felvétele után a rendszer specifikálása befejeződött, a következő fázis a rendszer tervezése. A tervezés során figyelembe kell venni a konkrét implementációs környezet lehetőségeit, és az absztrakt eseményeket és funkciókat a rendelkezésre álló elemekre kell leképezni.

A kialakított könyvtárunk a következő fizikai eseményeket kezeli: karakter bevitel, egérgomb lenyomás, egér mozgatás, egérgomb elengedés. Kézenfekvő a logikai események és a fizikai események következő összerendelése: az egérgomb lenyomása a szakasz kezdőpontját jelöli ki, az egér mozgatás a gumivonal mozgatás eseményt jelenti, az egérgomb elengedése a végpont kijelölésének felel meg, az R billentyű piros, a G billentyű zöld, a B pedig kék rajzolási színt állít be, a C billentyű lenyomásának hatására pedig töröljük a képernyőt.

A logikai eseményeknek a fizikai eseményekkel történő felváltása után az állapotátlák közvetlenül implementációs programokká alakíthatók. Mivel az eseményvezérelt programoknak az egyes fizikai eseményekre kell reagálniuk, az állapotátlákat a fizikai eseményeknek (üzeneteknek) megfelelő oszloponként dolgozzuk fel, és az üzenetek kezelését a lehetséges állapotok szerint külön ágakban végezzük el. Az egyes ágakban a következő állapotot is beállítjuk, hogy a jövőbeli események feldolgozása során emlékezhessünk ezen esemény bekövetkeztére. Vegyük észre, hogy a soronként felírt állapotátlák oszloponkénti kiolvasása azt jelenti, hogy a felhasználó szemszögéből a program szemszögére térünk át.

Az állapotokban végzett tevékenységeket programutasításokra és a rendelkezésre álló szolgáltatásokra kell visszavezetni. Ezek a szolgáltatások a Window osztály nem virtuális tagfüggvényei. Jelen feladatban a Set start és Set end a saját adattagok értékének megváltoztatását, a Draw normál rajzolást, a DrawRubber és DeleteRubber funkciók pedig XOR módú rajzolást jelentenek. Ezek alapján a Line osztály definíciója a következő:

```
//=====
class Line {
//=====
    enum { IDLE, START, RUBBER } state;
    Coord start_x, start_y, end_x, end_y;
    Color c;

    void DrawRubber( ) {
        pwindow -> RasterOp(XOR);
        Draw( );
        pwindow -> RasterOp(SET);
    }

    void DeleteRubber( ) { DrawRubber( ); }

    void Draw( ) {
        pwindow -> SetColor( c );
        pwindow -> Move( start_x, start_y );
        pwindow -> DrawLine( end_x, end_y );
    }
public:
    Line( ) { state = IDLE; }

    void SetColor( Color c0 ) { c = c0; }

    void SetStart( Coord x, Coord y ) {
        start_x = x; start_y = y; state = START;
    }

    void SetEnd( Coord x, Coord y ) {
        switch (state) {
            case START: state = IDLE; break;
            case RUBBER: DeleteRubber( );
                        end_x = x; end_y = y;
                        Draw( );
                        state = IDLE; break;
        }
    }
}
```

```

void MoveRubber( Coord x, Coord y ) {
    switch (state) {
    case IDLE:    break;
    case START:  end_x = x; end_y = y;
                  DrawRubber( );
                  state = RUBBER; break;
    case RUBBER: DeleteRubber( );
                  end_x = x; end_y = y;
                  DrawRubber( ); break;
    }
}
};

```

Az alkalmazás elkészítéséhez az általános Window osztályt az elvárt viselkedésnek megfelelően specializálni kell, majd a belépési ponton egy ilyen típusú objektumot kell létrehozni. A specializáció a szükséges mezők felvételét és azon felhasználói eseményeket kezelő rutinok átírását jelenti, amelyekre az alapértelmű reakció nem megfelelő:

```

//=====
class RubberWindow : public Window {
//=====
    Line line;

    void MouseLeftBtnDown( Coord x, Coord y ) { line.SetStart(x, y); }
    void MouseLeftBtnUp(  Coord x, Coord y ) { line.SetEnd(x, y); }
    void MouseMove( Coord x, Coord y ) { line.MoveRubber(x, y); }
    void KeyboardEvent( int keyASCII ) {
        switch ( keyASCII ) {
        case 'R': line.SetColor( Color(1, 0, 0) ); break;
        case 'G': line.SetColor( Color(0, 1, 0) ); break;
        case 'B': line.SetColor( Color(0, 0, 1) ); break;
        case 'C': Clear( ); break;
        case 'Q': Quit( );
        }
    }
public:
    RubberWindow( ) { line.SetColor( Color(1, 0, 0) ); }
};

```

Végül az alkalmazás belépési pontján létrehozzuk az ablakobjektumot és beindítjuk az üzenetciklus működését:

```

void AppStart( ) {
    RubberWindow win;
    win.Execute( );
}

```

3. fejezet

A geometriai modellezés

A virtuális világ definiálását *modellezésnek* nevezzük. A modellezés során megadjuk a világban szereplő objektumok geometriáját, megjelenítési attribútumait és egyéb alkalmazásfüggő paramétereit (például egy ellenállás nagyságát, egy alkatrész anyagát, stb.). A következőkben a 2D és 3D geometria definiálásával foglalkozunk. A 2D és 3D grafikának azon objektumok lehetnek a részei, amelyek nem lógnak ki a 2 illetve a 3 dimenzióból. Ezek a 0 dimenziós pontok, az 1 dimenziós görbék, a 2 dimenziós síkbeli területek és térbeli felületek, valamint a 3 dimenziós testek. A *mérnöki megjelenítésben* (*scientific visualisation*) magasabb dimenziójú adatok is előfordulhatnak, ezeket a megjelenítés előtt vetíteni kell a 2 vagy 3 dimenziós térbe. A dimenzió fogalmának megfelelő általánosítása esetén beszélhetünk nem egész dimenziós objektumokról, ún. *fraktálokról* is. A fraktálokat külön fejezetben tárgyaljuk.

3.1. Pontok

Egy *pont* egy alkalmasan választott koordinátarendszerben a koordináták megadásával definiálható.

3.2. Görbék

Görbén folytonos vonalat értünk. Matematikai szempontból a görbe pontok halmaza. Ez a halmaz skalár egyenletekkel vagy vektor egyenlettel definiálható, amelyeket a görbe pontjai elégítenek ki. Egy 2D görbét megadó egyenletet felírhatunk *explicit* módon:

$$x = x(t), \quad y = y(t), \quad t \in [0, 1], \quad (3.1)$$

vagy *implicit* formában is:

$$f(x, y) = 0. \quad (3.2)$$

Például egy (x_0, y_0) középpontú, R sugarú kör explicit egyenlete:

$$x = x_0 + R \cdot \cos 2\pi t, \quad y = y_0 + R \cdot \sin 2\pi t, \quad t \in [0, 1], \quad (3.3)$$

illetve implicit egyenlete:

$$(x - x_0)^2 + (y - y_0)^2 - R^2 = 0. \quad (3.4)$$

Az explicit forma akkor előnyös, ha pontokat kell generálnunk a görbén. Ekkor a $[0, 1]$ intervallumon kijelölünk megfelelő számú t_i paraméterpontot, és behelyettesítjük az egyenletbe. Az implicit forma viszont különösen alkalmas arra, hogy eldöntsük, hogy egy adott pont illeszkedik-e a görbére. Ehhez az adott pontot be kell helyettesítenünk az egyenletbe és ellenőrizni, hogy 0-t kapunk-e eredményül. A számítógépes grafikában elsősorban az első funkcióra van szükség, ezért általában explicit egyenleteket használunk.

A 3D görbéket explicit formában adhatjuk meg:

$$x = x(t), \quad y = y(t), \quad z = z(t), \quad t \in [0, 1]. \quad (3.5)$$

Például egy (x_1, y_1, z_1) -től (x_2, y_2, z_2) -ig tartó 3D szakasz egyenlete:

$$x = x_1 \cdot t + x_2 \cdot (1 - t), \quad y = y_1 \cdot t + y_2 \cdot (1 - t), \quad z = z_1 \cdot t + z_2 \cdot (1 - t), \quad t \in [0, 1]. \quad (3.6)$$

Az explicit vagy implicit egyenletek implementálásával a grafikai programunkat felkészíthetjük *klasszikus görbeszegmensek* (kör, szakasz, ellipszis, stb.) kezelésére. A modellezés ekkor az egyenletek ismeretlen paramétereinek (például egy kör középpontja és sugara) megadását jelenti.

3.3. Szabadformájú görbék

A modellezési igények általában nem elégíthetők ki csupán klasszikus görbeszegmensekkel. Felvetődhet ugyan, hogy bármely görbe kellő pontossággal közelíthető például sok kis szakasszal, de ez nem mindenütt differenciálható görbéket eredményez, ami például mechanikai alkatrészeknél megengedhetetlen. Ezért egy olyan függvényosztályra van szükség, amelyben a görbe alakja a differenciálhatóság garantálásával tetszőlegesen kialakítható.

Egy kézenfekvő függvényosztály a polinomok osztálya,

$$x(t) = \sum_{i=0}^n a_i \cdot t^i, \quad y(t) = \sum_{i=0}^n b_i \cdot t^i, \quad t \in [0, 1], \quad (3.7)$$

amelyben egy görbét az a_i, b_i polinomegyütthatók megadásával specifikálhatunk. Vektoros alakban:

$$\vec{r}(t) = \sum_{i=0}^n [a_i, b_i] \cdot t^i, \quad t \in [0, 1]. \quad (3.8)$$

Sajnos a polinomegyütthatóknak nincs szemléletes tartalma, ezért a modellezés során használatuk kényelmetlen. Az együtthatók közvetlen megadása helyett azt az eljárást követhetjük, hogy a felhasználótól csak ún. *vezérlőpontokat* (*control point*) kérünk, amelyek meghatározzák a görbe alakját. Egy pont könnyen kijelölhető interaktív módszerekkel, például az egér segítségével. Majd a modellezőprogramra bízunk, hogy a megadott vezérlőpont sorozatra egy görbét illesszen, azaz kiszámolja a megfelelő polinomegyütthatókat.

Alapvetően két illesztési stratégia létezik. Amennyiben megköveteljük, hogy a görbe átmenjen a vezérlőpontokon, az eljárást *interpolációnak* nevezzük. Az *approximációs* módszerek ezzel szemben nem garantálják, hogy a számított görbe telibe találja a vezérlőpontokat, csak annyit, hogy nagyjából követi az általuk kijelölt irányvonalat. Az engedményért cserébe számos jó tulajdonságot várhatunk a görbétől.

3.3.1. Lagrange-interpoláció

Tegyük fel, hogy a megadott vezérlőpont sorozat $\vec{r}_1, \vec{r}_2, \dots, \vec{r}_n$. Keressük azt a minimális fokszámú $L(t)$ polinomot, amely t_1 -nél \vec{r}_1 , t_2 -nél \vec{r}_2 , stb. t_n -nél \vec{r}_n értéket vesz fel. Figyelembe véve a feltételek számát, a megfelelő polinom $n - 1$ -d fokú, az ismeretlen $[a_i, b_i]$ együtthatókat pedig megkaphatjuk, ha minden j ($j = 1, 2, \dots, n$) vezérlőpontra felírjuk a

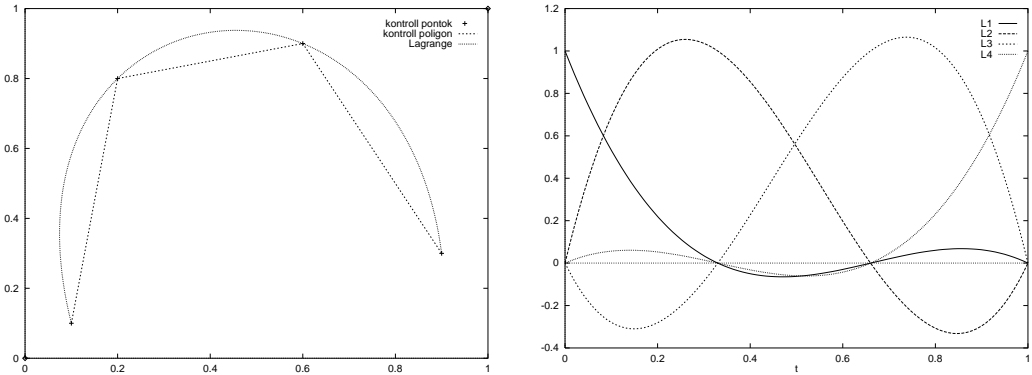
$$\vec{r}(t_j) = [x(t_j), y(t_j)] = \sum_{i=0}^{n-1} [a_i, b_i] \cdot t_j^i = \vec{r}_j$$

egyenletet és megoldjuk $[a_i, b_i]$ -re a keletkező egyenletrendszerrel. Ennek az egyenletrendszernek mindig van megoldása, hiszen az egyenletrendszerből keletkező Vandermonde-determináns nem lehet zérus.

Van azonban egy egyszerűbb eljárás is, ugyanis kapásból fel tudjuk írni a megoldást:

$$\vec{r}(t) = \sum_{i=1}^n L_i(t) \cdot \vec{r}_i, \quad \text{ahol } L_i(t) = \frac{\prod_{j \neq i} (t - t_j)}{\prod_{j \neq i} (t_i - t_j)}. \quad (3.9)$$

Az $L_i(t)$ lényegében az i . pont súlyát határozza meg a paraméter függvényében, ezért *súlyfüggvénynek* (*blending function*) nevezzük. A 3.1. ábra súlyfüggvényei $t_1 = 0$, $t_2 = 0.33$, $t_3 = 0.67$ és $t_4 = 1$ paraméterekkel készültek. Figyeljük meg, hogy a $t = t_i$ értékre egyetlen súlyfüggvény vesz fel 1 értéket, az összes többi pedig zérus,



3.1. ábra. Lagrange-interpoláció és súlyfüggvényei

így valóban $\vec{r}(t_i) = \vec{r}_i$. Sajnos bizonyos tartományokban a súlyfüggvények negatív értékűek, itt a vonatkozó vezérlőpont taszítja a görbét. Ebből származik az a tény, hogy a Lagrange-interpoláció hajlamos az oszcillációra, azaz olyan kanyarulatok létrehozására, amely nem következne a vezérlőpont-sorozatból. Másik fő nehézség az, hogy a görbe kényelmetlenül alakítható, hiszen a súlyfüggvények a teljes tartományon zérustól különböznek, azaz egy vezérlőpont a görbe minden részére hat. Gondoljunk arra, hogy a görbe már majdnem mindenütt jó, csak egy kicsit kellene alakítani rajta, de egy vezérlőpont módosítása a görbét mindenhol megváltoztatja, tehát ott is ahol már nagy nehezen elegyengettük.

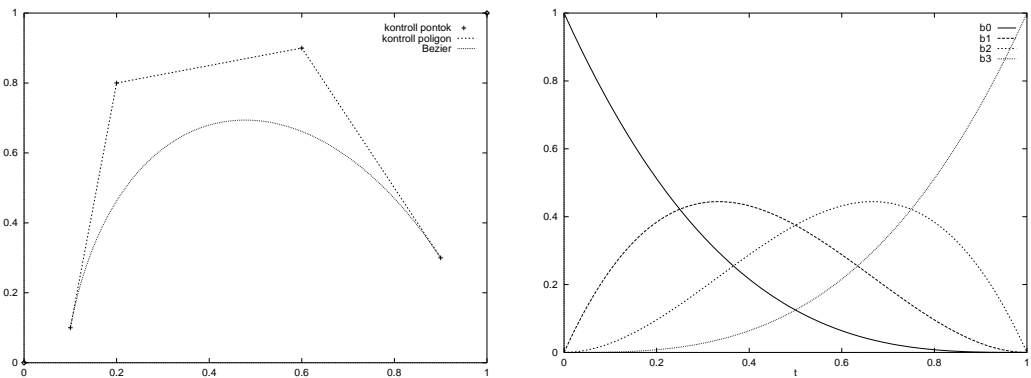
3.3.2. Bézier-approximáció

Ha feladjuk azt a megkötést, hogy a görbének át kell mennie a vezérlőpontokon, akkor a vezérlőpontokból nem következő hullámosság eltüntethető. Azok a görbék simák és hullámoktól mentesek, amelyek nem lépnek ki a vezérlőpontok konvex burkából (egy ponthalmaz *konvex burka* (*convex hull*) az a minimális konvex halmaz, ami a ponthalmazt tartalmazza). Konvex burokkal például az ajándékok becsomagolása során találkozhatunk, hiszen a szépen kifeszített csomagolópapír éppen a tárgyak konvex burkára simul rá). A konvex burokban maradás feltétele az, hogy a súlyfüggvények ne legyenek negatívak és összegük mindenhol 1 legyen. Ekkor egy adott paraméterre a görbe pontját egy olyan mechanikai rendszer súlypontjaként is elképzelhetjük, amelyben az egyes referenciapontokba a súlyfüggvények pillanatnyi értékével megegyező súlyt helyezünk el. Nyilvánvaló, hogy pozitív súlyoknál a súlypont nem kerülhet a rendszer konvex burkán kívülre.

Egy fontos súlyfüggvénykészlethez juthatunk a $(t + (1 - t))^m$ binomiális tétel szerinti kifejtésével. A kifejtés egyes tagjait *Bernstein-polinomoknak* nevezünk:

$$(t + (1 - t))^m = \sum_{i=0}^m \binom{m}{i} t^i \cdot (1 - t)^{m-i}. \quad (3.10)$$

$$B_i^{(m)}(t) = \binom{m}{i} t^i \cdot (1 - t)^{m-i}. \quad (3.11)$$



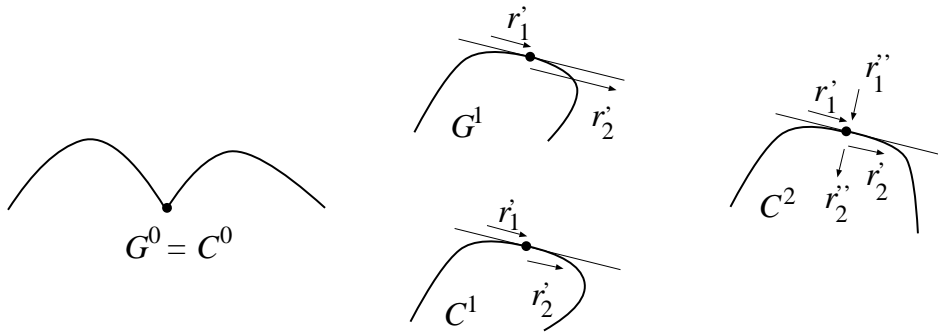
3.2. ábra. Bézier-approximáció és súlyfüggvényei

A definícióból rögtön adódik, hogy $\sum_{i=0}^m B_i^{(m)}(t) = 1$, és ha $t \in [0, 1]$, akkor $B_i^{(m)}(t) \geq 0$, tehát a görbe kielégíti a konvex burok tulajdonságát.

Mivel $B_0^{(m)}(0) = 1$ és $B_m^{(m)}(1) = 1$, a görbe átmegy az első és utolsó vezérlőpon-
ton, de általában nem megy át a többi vezérlőpon-
ton. Mint az könnyen bebizonyítható,
a görbe kezdete és vége érinti a vezérlőpontok által alkotott sokszöget (3.2. ábra).

3.3.3. Összetett görbék

A bonyolult görbéket nagyon sok vezérlőponttal definiálhatjuk. A görbeillesztés során alapvetően két eljárást követhetünk. Vagy egyetlen magas fokszámú polinomot illesztünk a görbére, vagy több alacsony fokszámút. A magas fokszámú polinomok hajlamosak a hullámosságra, ezért nem szeretjük őket. Ezért vonzóbb lehetőséget nyújt a több alacsony fokszámú görbeszegmens alkalmazása. A több görbeszegmensből építkező görbéket *összetett görbéknek* (*composite curve*) nevezzük.



3.3. ábra. Két görbe illeszkedésének folytonossági osztályai

Az összetett görbék alkalmazása során meg kell birkóznunk a szegmensek folytonos illesztésének a problémájával. A kérdéskör tárgyalását néhány definícióval kezdjük.

Két görbét geometriai értelemben 0-d rendűen folytonos (G^0 folytonos) illeszkedésűnek mondunk, ha a keletkező görbe megrajzolható anélkül, hogy a ceruzánkat fel kellene emelnünk. Más megközelítésből, két görbe parametrikus értelemben 0-d rendűen folytonos (C^0 folytonos) illeszkedésű, ha a keletkező függvény folytonos, azaz $r_1(t_{\text{end}}) = r_2(t_{\text{start}})$. Nyilván a G^0 és a C^0 ugyanazt a tulajdonságot írja le két eltérő szemszögből. Mind a G geometriai, mind pedig a C parametrikus folytonosság tovább fokozható. Beszélhetünk például G^1 folytonos illeszkedésről, ha a görbék érintői is párhuzamosak. A parametrikus illeszkedés tetszőleges fokszámra általánosítható. Két görbét akkor nevezünk C^n folytonos illeszkedésűnek, ha az egyik görbe deriváltjai a végponton megegyeznek a másik görbe deriváltjaival a kezdőponttól az n . deriváltig bezárólag.

Ezek után az alapvető kérdés az, hogy milyen szintű folytonosságot értelmes megkövetelnünk. Vegyünk két példát! Legyen egy meghajlított rúd alakja az $y(x)$ függvény. 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, megfelelő mérnöki lendülettel jelentsük ki, hogy a mérnöki alkalmazásokban gyakran C^2 folytonos görbékre van szükségünk. A C^2 folytonos összetett görbék neve *spline*.

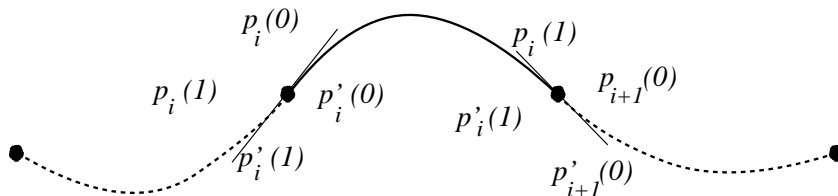
Harmadfokú spline

A C^2 folytonos illesztés követelménye az egyes görbeszegmensek illesztési pontjaira eltérő második deriváltat írhat elő. A legegyszerűbb polinom, amelynél a második derivált nem állandó, harmadfokú. A következőkben ilyen szegmensekkel foglalkozunk. Egy harmadfokú szegmens általános alakja

$$\vec{p}(t) = \vec{a}_3 t^3 + \vec{a}_2 t^2 + \vec{a}_1 t + \vec{a}_0. \quad (3.12)$$

Az $\vec{a}_3, \vec{a}_2, \vec{a}_1, \vec{a}_0$ polinomegyütthatók helyett használhatunk más reprezentációt is, például a szegmenst jellemző függvények és a deriváltjaik értékét a kezdő és végpontban. Ezek és a polinomegyütthatók között egy-egy értelmű kapcsolat van:

$$\begin{aligned} \vec{p}(0) &= \vec{a}_0, \\ \vec{p}(1) &= \vec{a}_3 + \vec{a}_2 + \vec{a}_1 + \vec{a}_0, \\ \vec{p}'(0) &= \vec{a}_1, \\ \vec{p}'(1) &= 3\vec{a}_3 + 2\vec{a}_2 + \vec{a}_1. \end{aligned} \quad (3.13)$$



3.4. ábra. Harmadfokú spline

Tegyük fel, hogy a megadott $\vec{r}_1, \vec{r}_2, \dots, \vec{r}_n$ vezérlőpont sorozatra úgy illesztjük a szegmenseket, hogy az első szegmensünk az \vec{r}_1 -től az \vec{r}_2 -ig tartson, a második az \vec{r}_2 -től az \vec{r}_3 -ig, stb. Ehhez az i . szegmens paramétereit úgy kell megválasztani, hogy

$$\vec{p}_i(0) = \vec{r}_i, \quad \vec{p}_i(1) = \vec{r}_{i+1}. \quad (3.14)$$

Ezzel a szegmensek 4 reprezentánsából kettőt kötöttünk meg. Vegyük még ehhez hozzá további feltételként, hogy a görbe legyen C^1 folytonos, azaz az egymást követő szegmensek megfelelő reprezentánsai legyenek azonosak:

$$\vec{p}'_i(1) = \vec{p}'_{i+1}(0). \quad (3.15)$$

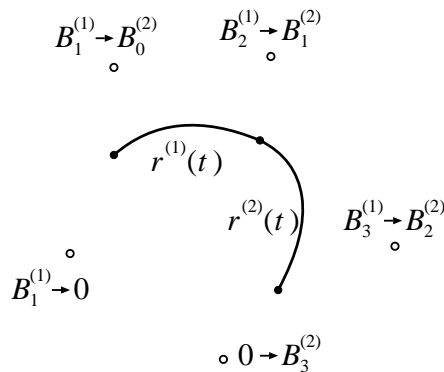
A deriváltak tényleges értékét pedig úgy határozzuk meg, hogy a határon még a C^2 folytonosság is teljesüljön:

$$\vec{p}''_i(1) = \vec{p}''_{i+1}(0). \quad (3.16)$$

Ez egy lineáris egyenletrendszer jelent az ismeretlen $\vec{p}_i'(0), \vec{p}_i'(1)$ paraméterekre, amelyet megoldva minden görbeszegmens reprezentációja meghatározható. Mivel az ismeretlenek száma éppen kétfővel több mint az egyenletek száma, az egyenletrendszernek végtelen sok megoldása van, azaz végtelen sok különböző interpolációs görbe létezik. Ha a görbe kezdő és végpontján a deriváltak értékét (a sebességet) megadjuk, akkor a feladat megoldása egyértelművé válik.

B-spline

A harmadfokú spline-nál egy ügyes reprezentációt használtunk, amellyel a C^1 folytonosságot már azáltal is sikerült biztosítani, hogy a 4 reprezentánsból 2-t a két egymás utáni szegmens közösen birtokolt. Ezek után már csak a C^2 folytonosság garantálása igényelt izzadságceppeket.



3.5. ábra. A B-spline szegmensek illeszkedése

Felvetődhet a kérdés, hogy nincs-e olyan reprezentáció, amelyben a 4 reprezentánsból háromnak a közös birtoklása automatikusan garantálja a C^2 folytonosságot is. Keressük a görbeszegmenst a szokásos alakban, ahol a 4 vezérlőpont súlyozásával kapjuk meg a görbét:

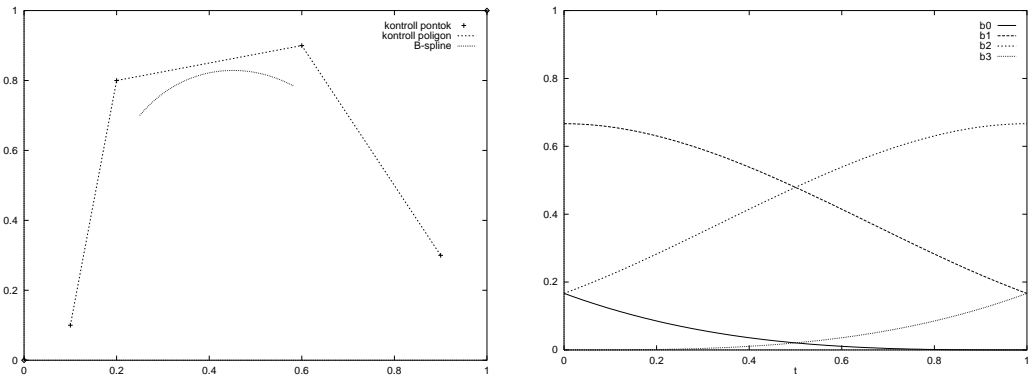
$$\vec{r}(t) = B_0(t) \cdot \vec{r}_0 + B_1(t) \cdot \vec{r}_1 + B_2(t) \cdot \vec{r}_2 + B_3(t) \cdot \vec{r}_3. \quad (3.17)$$

Ekkor a görbe természetes reprezentánsai a vezérlőpontok, tehát célunk olyan súlyfüggvények keresése, amelyek biztosítják, hogy ha két egymást követő szegmens közösen használ a 4 vezérlőpontból hármát, akkor a két szegmens C^2 folytonosan illeszkedik egymáshoz.

Egy elégséges feltételrendszerhez jutunk, ha a súlyfüggvények úgy kapcsolódnak egymáshoz, mint a cirkuszi elefántok, és ráadásul C^2 folytonosan. Pontosabban a B_0

az 1 értéknél C^2 folytonosan a 0-hoz tart, B_1 az 1 értéknél C^2 folytonosan folytatható a B_0 0-nál induló alakjával, hasonlóan a B_2 a B_1 -ével, a B_3 a B_2 -ével, végül a B_3 az 0-ból C^2 folytonosan indul. Ez összesen 15 vektorfeltételt jelent, a szegmensek 4 vektoregyütthatói összesen pedig 16 vektorismeretlent tartalmaznak. Ha még hozzávesszük feltételként, hogy a súlyfüggvények összege mindig 1-t adjon, akkor a feladat teljesen határozottá válik, amelyet megoldva a következő súlyfüggvényekhez jutunk:

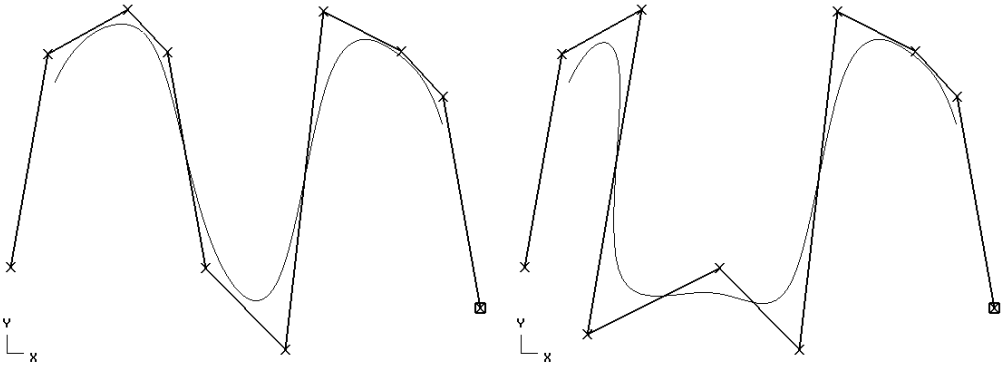
$$\begin{aligned} B_0(t) &= \frac{(1-t)^3}{6}, \\ B_1(t) &= \frac{1+3(1-t)+3t(1-t)^2}{6}, \\ B_2(t) &= \frac{1+3t+3(1-t)t^2}{6}, \\ B_3(t) &= \frac{t^3}{6}. \end{aligned} \quad (3.18)$$



3.6. ábra. B-spline approximáció és bázisfüggvényei

Vegyük észre, hogy a súlyfüggvények nem negatívak, tehát a görbeszegmens mindig a 4 vezérlőpont konvex burkán belül van. Másrészt, nincs olyan paraméterérték, ahol a súlyfüggvények egyetlen súlyfüggvény kivételével 0 értéket vennének fel, így a görbe általában nem megy át a vezérlőpontokon (approximációs tulajdonságú).

Ezen súlyfüggvénykészlettel definiált szegmensekből összerakott görbe neve *B-spline*. A B-spline görbének van egy igen hasznos tulajdonsága, amellyel a korábban definiált görbék nem rendelkeznek. Egy vezérlőpont csak az összetett görbe 4 legközelebbi szegmensének alakjára hat, a távolabbi részekre nem, tehát egy vezérlőpont megváltoz-



3.7. ábra. B-spline lokális vezérelhetősége: a 4. vezérlőpont áthelyezése csak a görbe első részét módosítja

tatása a görbe egy kicsiny részét módosítja. Az ilyen típusú görbéket *lokálisan vezérelhető* görbéknek nevezzük. A lokális vezérelhetőséget az tudja értékelni, aki interaktív módszerekkel már próbált bonyolult görbét rajzolni.

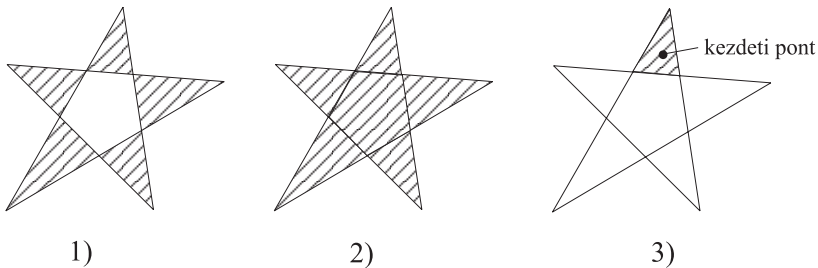
Bár a B-spline görbe alapvetően approximációs jellegű, interpolációs feladatokra is használható. Ha például olyan görbét szeretnénk, amely átmegy a $\vec{p}_1, \vec{p}_2, \dots, \vec{p}_n$ pontokon, akkor az interpolációs B-spline görbe $\vec{r}_0, \vec{r}_2, \dots, \vec{r}_{n+1}$ vezérlőpontjait például azal a feltételrendszerrel határozhatjuk meg, hogy az első szegmens kezdőpontja legyen \vec{p}_1 , végpontja pedig \vec{p}_2 , a második kezdete \vec{p}_2 , végpontja \vec{p}_3 , stb., a $n + 1$. kezdőpontja \vec{p}_n , végpontja pedig \vec{p}_{n+1} . Ez egy lineáris egyenletrendszert eredményez az ismeretlen vezérlőpont seregére [Wat89].

A spline-ok általánosításaival újabb, még rugalmasabban alakítható görbecsaládokhoz jutunk. Például mind a harmadfokú spline-t, mind a B-spline-t kiterjeszthetjük oly módon, hogy az egymást követő szegmensek különböző méretű paramétertartományt fedjenek le (idáig feltettük, hogy minden szegmensnek a paramétertartományban egy egységnyi intervallum felel meg). A B-spline ezen változatát *nem-uniform B-spline*-nak (*non-uniform B-spline* vagy NUBS) nevezzük. Egy másik fajta általánosítás a súlyfüggvényekre nem csupán polinomokat enged meg, hanem két polinom hányadosát is. A B-spline-ből ezen a módon *racionális B-spline*-t (*rational B-spline* vagy RBS) hozhatunk létre. A két kiterjesztés egyszerre is alkalmazható, amivel a *nem-uniform racionális B-spline*-hoz (*non-uniform rational B-spline* vagy NURBS) juthatunk el.

3.4. Területek

A *területek* síkbeli alakzatok, amelyeknek határa és belseje van. A határ lényegében egy görbe, amelyet a korábbi fejezet módszereivel írhatunk le. A belső tartomány fogalma többféleképpen is értelmezhető:

1. A belső pontok azok, amelyeket ha a végtelen távolból közelítenénk meg, a határgörbét páratlan számúszor lépünk át.
2. A belső pontok azok, amelyeket nem lehet a végtelen távolból anélkül elérni, hogy ne metszenék a határgörbét.
3. Egy adott kezdeti ponthoz képest belső pontok azok, amelyeket a kezdeti pontból elérhetünk anélkül, hogy a határon átlépünk.



3.8. ábra. A terület belsejének három értelmezése

3.5. Felületek

A 3D *felületek*, a 2D görbékhez hasonlóan definiálhatók *explicit egyenletekkel*:

$$x = x(u, v), \quad y = y(u, v), \quad z = z(u, v), \quad u, v \in [0, 1], \quad (3.19)$$

vagy *implicit egyenlettel*:

$$f(x, y, z) = 0. \quad (3.20)$$

Például egy (x_0, y_0, z_0) középpontú, R sugarú *gömb* explicit egyenletei:

$$x = x_0 + R \cdot \cos 2\pi u \cdot \sin \pi v, \quad y = y_0 + R \cdot \sin 2\pi u \cdot \sin \pi v, \quad z = z_0 + R \cdot \cos \pi v, \\ u, v \in [0, 1], \quad (3.21)$$

illetve implicit egyenlete

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - R^2 = 0. \quad (3.22)$$

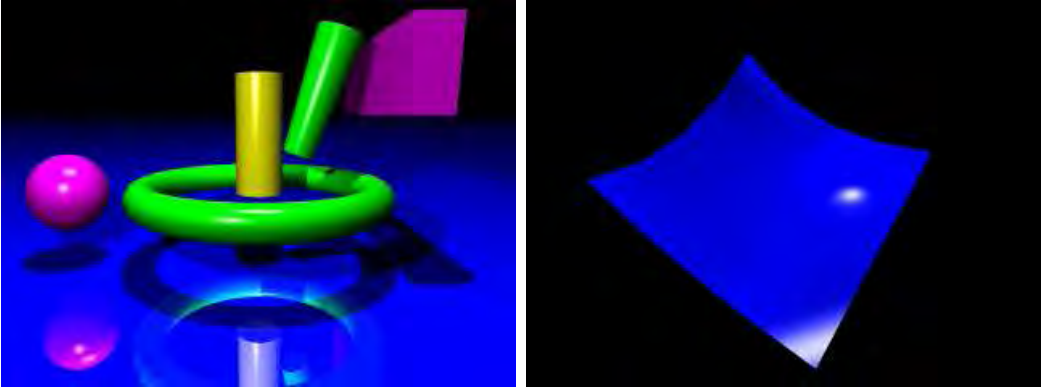
3.5.1. Kvadratikus felületek

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

$$[x, y, z, 1] \cdot \mathbf{Q} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0, \quad (3.23)$$

ahol \mathbf{Q} egy 4×4 -es konstans együtthatómátrix.

A kvadratikus felületek speciális típusai a gömb, hengerpalást, kúp, paraboloid, hiperboloid, stb. (3.9. ábra).



3.9. ábra. Kvadratikus felületek (bal) és Bézier-felület (jobb)

3.5.2. Parametrikus felületek

A parametrikus felületek kétváltozós polinomok:

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

A polinomokat a görbékhez hasonlóan általában nem közvetlenül a polinomegyütthatókkal, hanem a vezérlőpontokból súlyfüggvényekkel állítjuk elő:

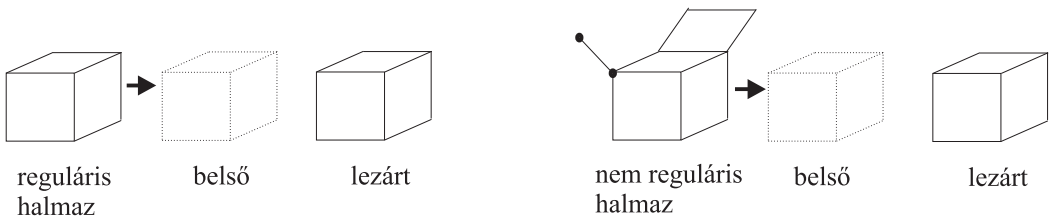
$$\vec{r}(u, v) = \sum_{i=0}^n \sum_{j=0}^m \vec{r}_{ij} \cdot B_{ij}(u, v). \quad (3.24)$$

A $B_{ij}(u, v)$ súlyfüggvény egy kézenfekvő definíciójához jutunk, ha két, a görbéknel megismert súlyfüggvény szorzatát képezzük. Például a *Bézier-felület* (3.9. ábra) súlyfüggvénye:

$$B_{ij}(u, v) = \binom{n}{i} \cdot u^i \cdot (1 - u)^{n-i} \cdot \binom{m}{j} \cdot v^j \cdot (1 - v)^{m-j}. \quad (3.25)$$

3.6. Testek

*Test*nek a 3D tér egy korlátos részhalmazát nevezzük. Ebben a halmazban *belső pontok* azok, amelyeknek van olyan bármilyen kicsiny nem zérus méretű környezete, amelyben minden pont a halmazhoz tartozik. A halmaz nem belső pontjait *határpontok*knak nevezzük. Elvárjuk, hogy a határpontok valóban 3D tartományokat fogjanak közre, azaz, hogy a határpontok bármely környezetében legyenek belső pontok is. Ez a feltétel lényegében azt akadályozza meg, hogy a testnek alacsonyabb dimenzióssá fajuló részei legyenek. Más szemszögből, ha a test pontjaiból eltávolítjuk a határpontokat, majd a keletkező halmazt lezárjuk, azaz hozzávesszük azon pontokat, amelyek feltétlenül szükségesek ahhoz, hogy a halmaz minden pontját belső ponttá tegyék, akkor éppen az eredeti halmazt kapjuk vissza. Az olyan halmazokat, amelyek a belső pontjaik *lezártjai*, *reguláris halmazok*knak nevezzük.



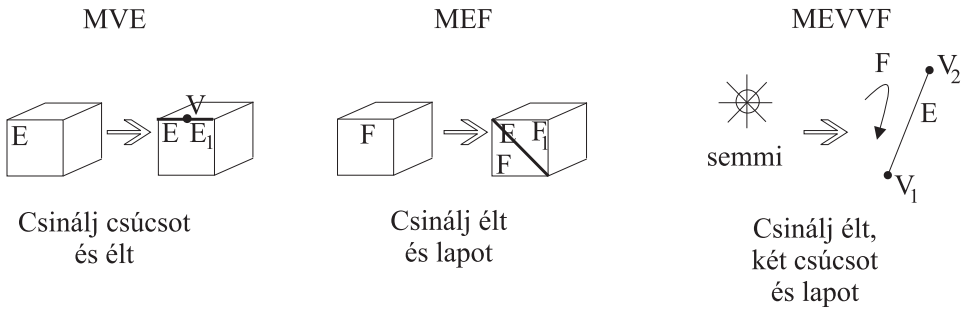
3.10. ábra. Egy reguláris halmaz (bal szélső kép) és egy nem reguláris halmaz (jobb szélső kép) belső pontjai és lezártja

A következőkben az általános testek létrehozásának két legfontosabb módszerével ismerkedünk meg.

3.6.1. Felületmodellezés

Egy általános test létrehozása visszavezethető a *határfelületek* definíciójára. Ha a határfelületeket egymástól függetlenül adjuk meg, akkor nem lehetünk biztosak abban, hogy azok hézagmentesen illeszkednek egymáshoz és egy érvényes 3D testet fognak

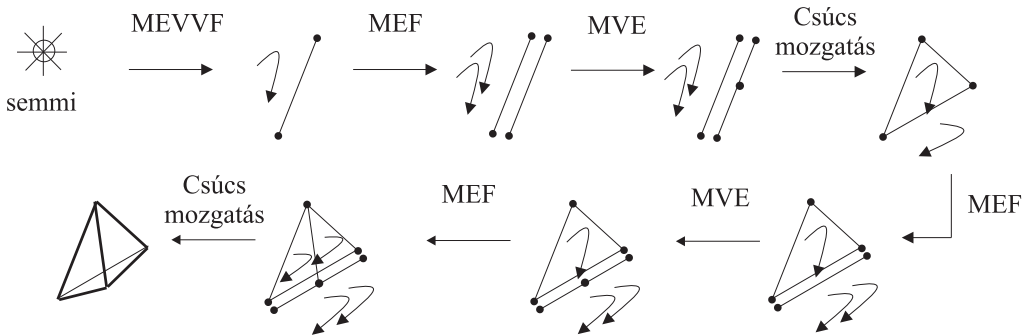
közre. Ezért a testmodellezésnek olyan műveletekkel kell építkeznie, amelyek a test *topológiai helyességét* garantálják. Az ilyen felületi modellezőket nevezzük *határfelület reprezentációs (boundary representation) vagy B-rep* rendszereknek.



3.11. ábra. Euler-műveletek

Az egyszerűség kedvéért foglalkozunk csak lyukakat nem tartalmazó *poliéderek* létrehozásával. Egy ilyen poliéder érvényességének szükséges feltétele, hogy ha l lapot, c csúcst és e élt tartalmaz, akkor fennáll az *Euler-tétel*:

$$l + c = e + 2. \quad (3.26)$$



3.12. ábra. Tetraéder felépítése Euler-műveletekkel

Azokat az elemi műveleteket, amelyek során ezen egyenlet érvényes marad, *Euler-műveleteknek* nevezzük. Az Euler-műveleteknek egy egyszerűsített halmazát láthatjuk a 3.11. ábrán (valóságban ennél több művelet szükséges, hiszen ezekkel csak olyan test hozható létre, amely nem tartalmaz lyukakat).

Ha tehát az Euler-műveletek segítségével építkezünk, a testünk minden pillanatban

kielégíti az Euler-egyenletet. A 3.12. ábrán egy tetraéder Euler-műveletekkel történő felépítése látható.

3.6.2. Konstruktív tömörtest geometria alapú modellezés

A *konstruktív tömörtest geometria* (*Constructive Solid Geometry* vagy *CSG*) az összetett testeket *primitív testekből* halmazműveletek (egyesítés, metszet, negáció) alkalmazásával építi fel (3.13. ábra). Annak érdekében, hogy a keletkező test mindig kielégítse a

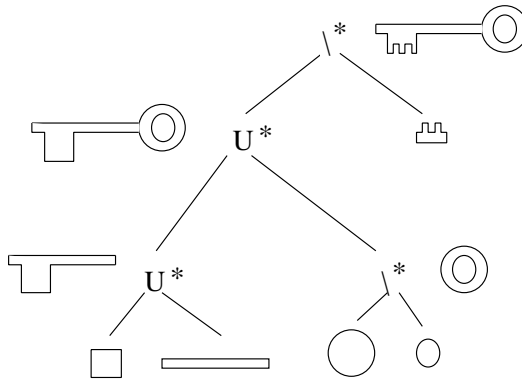


3.13. ábra. A három alapvető halmazművelet egy nagy gömbre és 6 kis gömbre: egyesítés, különbség és metszet

testekkel szemben támasztott követelményeinket — azaz ne tartalmazzon alacsonyabb dimenziójú elfajult részeket — nem a közöséges halmazműveletekkel, hanem azok regularizált változataival dolgozik. Egy *regularizált halmazműveletet* úgy végzünk el, hogy először kivesszük a halmazokból a határpontokat, elvégezzük a belső pontokra a normál halmazműveletet, majd a keletkező halmazt lezárjuk, azaz hozzávesszük annak határpontjait.

Bonyolult objektumok nem állíthatók elő a primitív testekből valamely reguláris halmazművelet egyszeri alkalmazásával, hanem egy teljes műveletsorozatot kell végrehajtani. Mivel az egyes műveletek két operandusa lehet primitív test vagy akár primitív testekből korábban összerakott összetett test, a teljes konstrukció egy bináris fával szemléltethető, amelynek csúcán a végleges objektum, levelein a primitív objektumok, közbenső csúcspontjain pedig a műveletsor részeredményei láthatók (3.14. ábra).

A CSG módszer érdekes kiterjesztéséhez jutunk, ha a szigorú faszerkezet helyett megengedünk ciklusokat is a gráfban (17.5. ábra).



3.14. ábra. Összetett objektum felépítése halmazműveletekkel

3.7. Program: paraméteres görbék

Ebben a fejezetben paraméteres görbeprimitíveket megvalósító osztályokat mutatunk be. Az osztályok létrehozásánál felhasználjuk a dinamikusan nyújtózó tömböt megvalósító generikus `Array` osztályt. Az ilyen osztállyal definiált tömbök kezdő méretét a konstruktorban adhatjuk meg. Ha az index operátor index határ túllépést észlel, a tömb dinamikusan megnöveli a lefoglalt területet. A `Size` tagfüggvény megmondja, hogy mi volt az eddigi legnagyobb hivatkozott index.

```
//=====
template < class Type > class Array {
//=====
    int    alloc_size, size;
    Type * array;
public:
    Array( int s = 0 );
    void operator=( Array& a );
    Type& operator[] ( int idx );
    int    Size();
};
```

Minden modell alapja a “pont” illetve a “vektor”, amit a `Point2D` osztály definiál:

```
//=====
class Point2D {
//=====
    Coord x, y;
public:
    Point2D( double x0 = 0, double y0 = 0 ) { x = x0; y = y0; }
```

```

Point2D operator-( ) { return Point2D( -x, -y ); }
Point2D operator+( Point2D p ) { return Point2D(x+p.x, y+p.y); }
Point2D operator*( double s ) { return Point2D(x*s, y*s); }
double Length( ) { return sqrt( x*x + y*y ); }
double& X() { return x; }
double& Y() { return y; }
};

typedef Point2D Vector2D;

```

Egy primitívet, legyen az szakasz, sokszög, görbe, stb. pontokkal adunk meg, így az általános primitív pontokat tárol. A konkrét típusokat az általános típusból örökléssel definiálhatjuk.

```

//=====
class Primitive2D {
//=====
    Array<Point2D> points;
    Color          color;
public:
    Primitive2D( Color& c, int n = 0 ) : color(c), points(n) { }
    Point2D& Point( int i ) { return points[i]; }
    int      PointNum( ) { return points.Size(); }
};

```

A görbe a vezérlőpontjaiból interpolációval (vagy approximációval) állítja elő egy tetszőleges paraméterértékre a görbe egy adott pontját.

```

//=====
class Curve2D : public Primitive2D {
//=====
public:
    Curve2D( Color& c ) : Primitive2D( c ) { }
    virtual Point2D Interpolate( double tt ) = 0;
};

```

A különböző görbetípusok különböző polinomokat használnak ahhoz, hogy az interpoláció során a vezérlőpontokból a görbe pontját kiszámítsák. Ezért a görbetípusokat az általános görbe fogalomból és egy polinomból építhetjük fel.

```

//=====
class LagrangePolinom {
//=====
    Array<double> knot_pars;
public:
    int Degree( ) { return knot_pars.Size(); }
    double& t( int i ) { return knot_pars[i]; }
};

```

```

double L( int i, double tt ) {
    double Li = 1.0;
    for(int j = 0; j < Degree(); j++) {
        if (i != j) Li *= (tt - t(j)) / (t(i) - t(j));
    }
    return Li;
}

void DistributeKnotPars( int n ) {
    for (int i = 0; i <= n; i++) t(i) = (double)i / n;
}
};

//=====
class LagrangeCurve2D : public Curve2D, public LagrangePolinom {
//=====
public:
    LagrangeCurve2D( Color c ) : Curve2D( c ), LagrangePolinom( ) { }
    Point2D Interpolate( double tt ) {
        Point2D rr(0, 0);
        for(int i = 0; i < Degree(); i++) rr += Point(i) * L(i, tt);
        return rr;
    }
};

//=====
class BezierPolinom {
//=====
public:
    double B( int i, double tt, int m ) {
        double Bi = 1.0;
        for(int j = 1; j <= i; j++) Bi *= tt * (m-j)/j;
        for(      ; j < m; j++) Bi *= (1-tt);
        return Bi;
    }
};

//=====
class BezierCurve2D : public Curve2D, public BezierPolinom {
//=====
public:
    BezierCurve2D( Color c ) : Curve2D( c ), BezierPolinom( ) { }
    Point2D Interpolate( double tt ) {
        double Bi = 1.0;
        Point2D rr(0, 0);
        for(int i = 0; i < PointNum(); i++)
            rr += Point(i) * B(i, tt, PointNum());
        return rr;
    }
};

```

4. fejezet

Színelméleti alapok

A *fény* elektromágneses hullám, amelynek spektrális tulajdonságai a szemben *színérzetet* keltenek. A *szem* igen rossz spektrométer, amely a három különböző típusú érzékelőjével (ún. csappal vagy fotopigmenttel) a beérkező fényenergiát három, kissé átlapolódó tartományban összegzi. Következésképpen bármely színérzet három skalárral, ún. *tristimulus értékekkel* megadható. A lehetséges színérzetek alkotják a színteret, amely az elmondottak szerint egy három dimenziós térként képzelhető el. A térben kijelölhető egy lehetséges koordináarendszer oly módon, hogy kiválasztunk három elég távoli hullámhosszt, majd megadjuk, hogy három ilyen hullámhosszú monokromatikus fénynyaláb milyen keverékével kelthető az adott érzet. A komponensek intenzitásait *tristimulus koordinátáknak* nevezzük.

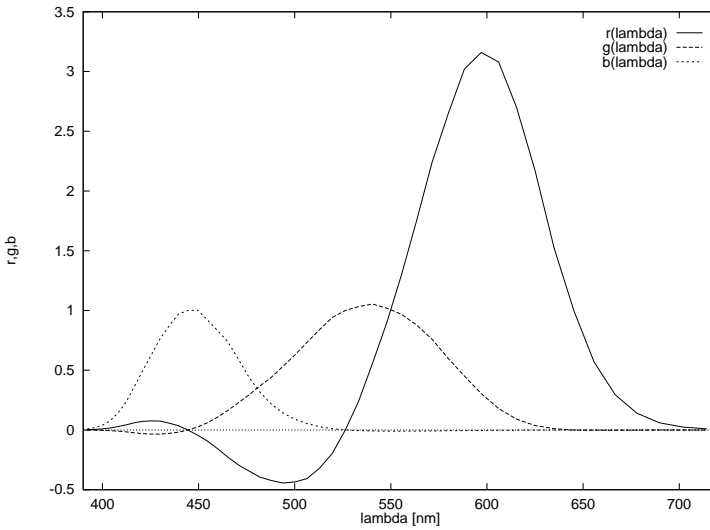
Az alábbi egy megfelelő készlet, amely az önmagukban vörös (*red*), zöld (*green*) és kék (*blue*) színérzetet okozó hullámhosszokból áll:

$$\lambda_{\text{red}} = 700 \text{ nm}, \quad \lambda_{\text{green}} = 561 \text{ nm}, \quad \lambda_{\text{blue}} = 436 \text{ nm}. \quad (4.1)$$

Egy λ hullámhosszú monokromatikus fénynyaláb keltette ekvivalens színérzetet ezek után az ($r(\lambda)$, $g(\lambda)$ és $b(\lambda)$) *színillesztő függvényekkel* adjuk meg, amelyeket fiziológiai mérésekkel határozhatunk meg (4.1. ábra). Ha a λ hullámhosszal végigmegyünk a látható tartományon, akkor a szivárvány színeit, azaz egy prizma által előállított színképet jeleníthetjük meg (17.10. ábra), hiszen a prizma is a keverékszint monokromatikus komponensekre bontja.

Amennyiben az érzékelt fénynyaláb nem monokromatikus, az r, g, b tristimulus koordinátákat az alkotó hullámhosszak által keltett színérzetek összegeként állíthatjuk elő. Például, ha a fényenergia spektrális eloszlása $\Phi(\lambda)$, akkor a megfelelő koordináták:

$$r = \int_{\lambda} \Phi(\lambda) \cdot r(\lambda) d\lambda, \quad g = \int_{\lambda} \Phi(\lambda) \cdot g(\lambda) d\lambda, \quad b = \int_{\lambda} \Phi(\lambda) \cdot b(\lambda) d\lambda. \quad (4.2)$$



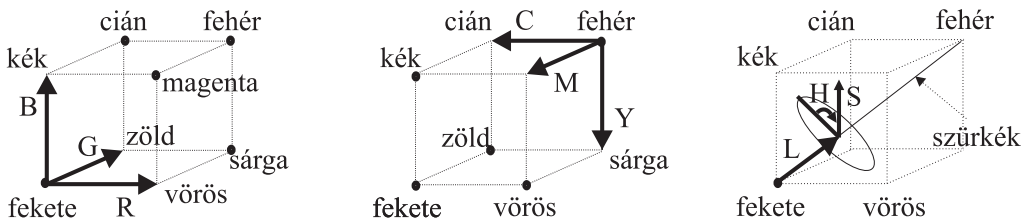
4.1. ábra. Az $r(\lambda)$, $g(\lambda)$ és $b(\lambda)$ színillesztő függvények

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 keverékével készítenek színes képet, lesznek olyan színek, amelyeket sohasem reprodukálhatunk a számítógépünk képernyőjén.

Az emberi szem-agy páros nem tud különbséget tenni két eltérő spektrum között, amennyiben azokra ugyanaz az r, g, b érték adódik. A hasonló színérzetet keltő fénynyalábokat *metamerek*nek nevezzük. A számítógép képernyőjén, a tv-képernyőhöz hasonlóan, vörös, zöld és kék színű fénynyalábokat emittáló foszforrétegek gerjesztésére van lehetőségünk. A célunk tehát az, hogy kiszámítsuk, hogy milyen intenzíven kell gerjeszteni ezeket a rétegeket ahhoz, hogy az elvárt színérzettel megegyező színérzetet keltsenek.

4.1. A színek definiálása

A színérzet három skalárral jellemezhető, így a színérzetek terét mint egy 3 dimenziós teret képzelhetjük el (4.2. és 17.11. ábrák). A tér pontjainak azonosításához egy bázist (koordinátarendszert) kell megadnunk. A bázis számtalan különböző módon létrehozható, így a színek is többféleképpen definiálhatók.



4.2. ábra. RGB, CMY és a HLS színrendszerek

4.1.1. RGB színrendszer

Az *RGB színrendszer* vagy *additív színrendszer* a bázis felállítása során a monitorban végbemenő folyamatot követi. Azt határozzuk meg, hogy a *vörös (Red)*, *zöld (Green)* és *kék (Blue)* érzetet keltő, monokromatikus nyalábokból milyen keverék hozza létre az elvárt színérzetet.

4.1.2. CMY színrendszer

A *CMY színrendszer* vagy *szubtraktív színrendszer* a nyomtatás szimulációja. Ebben a rendszerben azt mondjuk meg, hogy mennyi *cián (Cyan)*, *magenta (Magenta)* és *sárga (Yellow)* színt kell kivonni a fehérből (mennyi ilyen festéket kell rákenni a fehér lapra), hogy a keverék a kívánt színérzetet keltse.

4.1.3. HLS színrendszer

Az RGB és CMY színrendszerek fizikai folyamatokhoz kapcsolódnak, így nem kellően szemléletesek. Természetesebb, ha egy színt a *színárnyalattal*, a *fényességgel* és a *telítettség*gel adunk meg. Ez olyan bázis felállítást jelent, ahol a színekocka egy pontjához úgy jutunk el, hogy elsétálunk a főátlón valameddig, azaz megmondjuk, hogy a szín mennyi szürkét tartalmaz, azután elfordulunk valamerre a főátlóval merőleges irányban, azaz definiáljuk a színárnyalatot, végül eltávolodunk a főátlótól, azaz relatíve csökkentjük a szürke arányát és ezáltal növeljük a szín telítettségét. A főátlón megtett relatív távolságot fényességnek (L), a színárnyalatot meghatározó szög árnyalatnak (H), a főátlótól a kocka széléhez viszonyított relatív távolságot (S) pedig telítettségnek nevezzük (17.11. ábra). A 17.11. ábrán a főátlóra merőleges hatszögeket körökbe foglaltuk. Ezekben a körökben a szürke a kör középpontján van. Két, a középpontra nézve szimmetrikus elhelyezkedésű színt komplementer színeknek nevezünk, mert keverékük mindig fehér (pontosabban szürke). Megjegyezzük, hogy az emberi szem számára egy több színből álló színvilág akkor harmonikus, ha az abban előforduló színek keveréke fehér. Tehát ha egy hölgy csinosnak szeretne látszani, akkor azt tanácsolhatjuk neki,

hogy a szoknyáját és a blúzát célszerűen komplementer színekből, tehát valamely kör két szimmetrikusan elhelyezkedő pontjáról válassza. Ha a cipője is eltérő színű, akkor pedig három, 120 fokos szögben látszó pontot válasszon ki.

4.2. Színkezelés a 2D és a 3D grafikában

A 2D grafikában az objektumok saját színét jelenítjük meg, így miután meghatároztuk, hogy egy pixelben melyik objektum látszik, az adott pixelt az objektum színével kell kiszínezni.

3D grafikában azonban egy objektumról a kamerába jutó fény spektruma a térben lévő anyagok optikai tulajdonságainak és a fényforrásoknak a függvénye. Jelöljük a fényforrások által kibocsátott spektrumfüggvényt $\Phi_l(\lambda)$ -val (ez a hullámhosszon kívül a kibocsátási ponttól és az iránytól is függhet). Egy P pixelen keresztül a kamerába jutó spektrum a fényforrások spektrumának lineáris funkcionálja:

$$\Phi_P(\lambda) = \mathcal{L}(\Phi_l(\lambda)).$$

A funkcionált a felületi geometria, optikai tulajdonságok és a kamera állása határozza meg. A pixel r, g, b értékeit a színillesztő függvényekkel súlyozott integrálokkal határozhatjuk meg. Az integrálokat numerikus módszerekkel becsüljük. Például a vörös komponens:

$$r_P = \int_{\lambda} \Phi_P(\lambda) \cdot r(\lambda) d\lambda = \int_{\lambda} \mathcal{L}(\Phi_l(\lambda)) \cdot r(\lambda) d\lambda \approx \sum_{i=1}^n \mathcal{L}(\Phi_l(\lambda_i)) \cdot r(\lambda_i) \cdot \Delta\lambda. \quad (4.3)$$

Ezt azt jelenti, hogy a fényforrások intenzitását és a felületek visszaverődését n különböző hullámhosszon kell megadni (n szokásos értékei 3, 8, 16). A reprezentatív hullámhosszokon a pixelen keresztüljutó teljesítményt egyenként számítjuk ki, majd a 4.3 képlet alkalmazásával meghatározzuk a megjelenítéshez szükséges r, g, b értékeket.

A számítást gyakran csak a vörös, zöld és kék hullámhosszokra végezzük el. Ekkor a fényforrások spektruma helyett dolgozhatunk azok r, g, b értékeivel.

A megjelenítőeszközre történő színleképzés során azt is figyelembe kell vennünk, hogy a monitorok és nyomtatók által előállítható színdinamika (intenzitásváltozás) messze elmarad az emberi szem által érzékelhetőtől, ezért a számított értékeket még mindenképpen skálázni kell. A skálázás lehet lineáris vagy logaritmusos. A skálázás járulékos feladata lehet a *gamma-korrekción* is, azaz a monitor nem linearitásának a kiküszöbölése. Igényes alkalmazásokban a leképzés még figyelembe veszi a monitor környezetében érvényes fényviszonyokat is, hiszen a szem ezekhez adaptálódott, mielőtt a képernyőre néztünk volna.

Általában azt mondhatjuk, hogy ha a fizikai modellel számított színt közvetlenül íránk be a rasztertárba, a keltett színérzet a megjelenítő eszköz és az érzékelés nem-linearitásai miatt torz lenne. Ezért a beírás előtt a színt a megjelenítő eszköz és érzékelés nem-linearitásainak az inverzével elő kell torzítani. Ezt a torzítást *színleképező operátornak* (*tone-mapping operator*) nevezzük.

4.3. Program: színkezelés

Ebben a fejezetben a színek kezeléséhez szükséges osztályokat és függvényeket adjuk meg.

4.3.1. Színillesztő függvények

Egy monokromatikus fénysugár által keltett színérzetet a *színillesztő függvények* segítségével határozzuk meg. A színillesztő függvényeket (4.1. ábra) diszkrét pontokban a `matchfunc` táblázatban tároljuk, a függvényértéket egy tetszőleges hullámhosszra a tárolt értékekből a `ColorMatch` függvény interpolálja.

```
#define LAMBDALOW      393
#define LAMBDAHIGH    689

SpectrumVal matchfunc[NMATCHVALUE] = {
    {392, 0.0022,-0.0006, 0.0090}, {408, 0.0290,-0.0095, 0.1440},
    {425, 0.0760,-0.0340, 0.6300}, {444, 0.0000, 0.0000, 1.0000},
    {465,-0.2250, 0.1630, 0.7400}, {487,-0.4230, 0.4410, 0.2160},
    {512,-0.3220, 0.8370, 0.0278}, {540, 0.5610, 1.0540,-0.0082},
    {571, 2.2400, 0.7590,-0.0078}, {606, 3.0800, 0.1790,-0.0026},
    {645, 1.0000, 0.0000, 0.0000}, {689, 0.0601,-0.0005, 0.0000}
};

void ColorMatch( double lambda, double& r, double& g, double& b ) {
    for( int l = 1; l < sizeof matchfunc; l++ ) {
        if (lambda < matchfunc[l].lambda) {
            double la2 = lambda - matchfunc[l-1].lambda;
            double la1 = matchfunc[l].lambda - lambda;
            double la = la1 + la2;
            r = (la1 * matchfunc[l-1].r + la2 * matchfunc[l].r)/la;
            g = (la1 * matchfunc[l-1].g + la2 * matchfunc[l].g)/la;
            b = (la1 * matchfunc[l-1].b + la2 * matchfunc[l].b)/la;
            break;
        }
    }
}
```

4.3.2. Spektrumok kezelése

A spektrum egy hullámhosszfüggvény, amelynek értékét NLAMBDA diszkrét pontban tartjuk nyilván. A spektrumfüggvényt megvalósító osztályt sablonként definiáljuk, amelyet az NLAMBDA változóval paraméterezhetünk. A diszkrét pontokban a hullámhosszértéket a `lambdas[NLAMBDA]` tömb, az intenzitásértékeket pedig a `I[NLAMBDA]` tömb tárolja. A közbenső értékeket pedig ezekből a pontokból interpoláljuk.

A `ConvertToRGB` tagfüggvény a spektrumból numerikus integrálással számítja ki az ekvivalens vörös, zöld és kék komponenseket. A spektrumfüggvényen aritmetikai műveletek definiálhatók, mint két függvény összeadása, szorzása és skalárral történő nyújtása. A `Luminance` tagfüggvény egyetlen skalárral jellemzi a spektrumfüggvény összes energiáját.

```
#define foreach( l ) for(int l = 0; l < NLAMBDA; l++ )

//=====
template < int NLAMBDA > class Spectrum {
//=====
    static double lambdas[NLAMBDA];
protected:
    double I[NLAMBDA];
public:
    Spectrum( double gray = 0 ) { foreach(l) I[l] = gray; }
    double& Int( int l ) { return I[l]; }
    void ConvertToRGB( double& R, double& G, double& B ) {
        R = 0; G = 0; B = 0;
        double prev_lambda = 392;
        foreach(l) {
            double r, g, b, dl;
            ColorMatch( lambdas[l], r, g, b );
            dl = (lambdas[l] - prev_lambda) / (LAMBDAHIGH-LAMBDALOW);
            R += I[l] * r * dl; G += I[l] * g * dl; B += I[l] * b * dl;
            prev_lambda = lambdas[l];
        }
        if (R < 0) R = 0; if (G < 0) G = 0; if (B < 0) B = 0;
    }
    Spectrum operator*( double s ) {
        Spectrum<NLAMBDA> res; foreach(l) res.I[l] = I[l] * s;
        return res;
    }
    Spectrum operator*( Spectrum& m ) {
        Spectrum<NLAMBDA> res; foreach(l) res.I[l] = I[l] * m.I[l];
        return res;
    }
    void operator*=( Spectrum& m ) { (*this) = (*this) * m; }
```

```

Spectrum operator/( double d ) {
    Spectrum<NLAMBDA> res; foreach(l) res.I[l] = I[l] / d;
    return res;
}
void operator/=( double d ) { (*this) = (*this) / d; }
Spectrum operator+( Spectrum& d ) {
    Spectrum<NLAMBDA> res; foreach(l) res.I[l] = I[l] + d.I[l];
    return res;
}
void operator+=( Spectrum& d ) { (*this) = (*this) + d; }
double Red() {
    double R, G, B; ConvertToRGB( R, G, B ); return R;
}
double Green() {
    double R, G, B; ConvertToRGB( R, G, B ); return G;
}
double Blue() {
    double R, G, B; ConvertToRGB( R, G, B ); return B;
}
double Luminance( ) {
    double sum = 0; foreach(l) sum += I[l]/NLAMBDA; return sum;
}
int operator!=( double c ) { return ( c != Luminance( ) ); }
};

```

4.3.3. Színérzetek

A színérzet (Color) lényegében három elég távoli hullámhosszt tartalmazó spektrummal jellemezhető.

```

#define R I[0]
#define G I[1]
#define B I[2]

double Spectrum<3> :: lambdas[] = {436, 561, 700};

//=====
class Color : public Spectrum<3> {
//=====
    double HexaCone( double s1, double s2, double hue );
public:
    Color(double R0, double G0, double B0) { R = R0; G = G0; B = B0; }
    Color(double gray = 0) : Spectrum<3> ( gray ) { }
    void ConvertToRGB( double& R0, double& G0, double& B0 ) {
        R0 = R; G0 = G; B0 = B;
    }
}

```

```

void CMYToRGB( double C, double M, double Y );
void HLSToRGB( double H, double L, double S );
double Red() { return R; }
double Green() { return G; }
double Blue() { return B; }
};

```

CMY átalakítása RGB modellre

Az RGB és CMY modellek ugyanazon kocka két átlellenes sarokpontján felállított, a kocka élével megegyező bázisvektorokkal rendelkező koordináta-rendszerek.

```

//-----
void Color :: CMYToRGB( double C, double M, double Y ) {
//-----
    R = 1.0 - C; G = 1.0 - M; B = 1.0 - Y;
}

```

HLS átalakítása RGB modellre

A HLS színrendszer a csúcsára állított színekocka pontjait, az átlón megtett távolság (L), az átlótól való eltávolodás (S) és az eltávolodási irány (H) hármásával azonosítja.

```

//-----
double Color :: HexaCone( double s1, double s2, double hue ) {
//-----
    while (hue > 360)    hue -= 360;
    while (hue < 0)     hue += 360;
    if (hue < 60)      return (s1 + (s2 - s1) * hue/60);
    if (hue < 180)     return (s2);
    if (hue < 240)     return (s1 + (s2 - s1) * (240-hue)/60);
    return (s1);
}

//-----
void Color :: HLSToRGB( double H, double L, double S ) {
//-----
    double s2 = (L <= 0.5) ? L * (1 + S) : L * (1 - S) + S;
    double s1 = 2 * L - s2;
    if (S == 0) { R = G = B = L; }
    else {
        R = HexaCone(s1, s2, H - 120);
        G = HexaCone(s1, s2, H);
        B = HexaCone(s1, s2, H + 120);
    }
}

```

5. fejezet

Geometriai transzformációk

A számítógépes grafikában a kép geometriáját az objektumok geometriája alapján határozzuk meg. A geometria megváltoztatását *geometriai transzformációnak* nevezzük.

Mivel a számítógépekben mindent számokkal jellemezünk, a geometriai leírást is számok megadására vezetjük vissza. Az euklideszi geometria algebrai alapját a *Descartes-koordinátarendszer* adja, amelyben egy pontot a tengelyekre vetített távolságokkal jellemezünk. Síkban ez egy (x, y) számpárt, térben pedig egy (x, y, z) számhármast jelent. A transzformáció pedig ezeken a vektorokon értelmezett matematikai művelet.

5.1. Elemi affin transzformációk

A párhuzamos egyeneseket párhuzamos egyenesekbe átvivő transzformációt *affin transzformációnak* nevezzük. A *lineáris transzformációk* mind ilyenek.

5.1.1. Eltolás

Az *eltolás* egy konstans \vec{p} vektort ad hozzá a transzformálandó ponthoz:

$$\vec{r}' = \vec{r} + \vec{p}. \quad (5.1)$$

5.1.2. Skálázás a koordinátatengely mentén

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

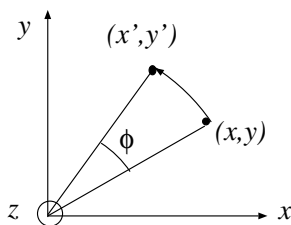
$$x' = S_x \cdot x, \quad y' = S_y \cdot y, \quad z' = S_z \cdot z. \quad (5.2)$$

Ezt a transzformációt mátrixszorzással is leírhatjuk:

$$\vec{r}' = \vec{r} \cdot \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix}. \quad (5.3)$$

5.1.3. Forgatás

A z tengely körüli ϕ szöggel történő forgatás az x és y koordinátákat módosítja, a z koordinátát változatlanul hagyja.



5.1. ábra. Forgatás a z tengely körül

Az elforgatott pont x és y koordinátái a következőképpen fejezhetők ki:

$$x' = x \cdot \cos \phi - y \cdot \sin \phi, \quad y' = x \cdot \sin \phi + y \cdot \cos \phi. \quad (5.4)$$

Mátrix műveletekkel:

$$\vec{r}'(z, \phi) = \vec{r} \cdot \begin{bmatrix} \cos \phi & \sin \phi & 0 \\ -\sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (5.5)$$

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

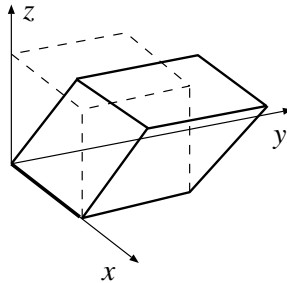
$$\vec{r}'(x, \phi) = \vec{r} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & \sin \phi \\ 0 & -\sin \phi & \cos \phi \end{bmatrix}, \quad \vec{r}'(y, \phi) = \vec{r} \cdot \begin{bmatrix} \cos \phi & 0 & -\sin \phi \\ 0 & 1 & 0 \\ \sin \phi & 0 & \cos \phi \end{bmatrix}. \quad (5.6)$$

A három tengely körüli egymás utáni forgatással bármely orientáció előállítható:

$$\vec{r}'(\alpha, \beta, \gamma) = \vec{r} \cdot \begin{bmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \beta & 0 & -\sin \beta \\ 0 & 1 & 0 \\ \sin \beta & 0 & \cos \beta \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \gamma & \sin \gamma \\ 0 & -\sin \gamma & \cos \gamma \end{bmatrix}. \quad (5.7)$$

5.1.4. Nyírás

Tegyük fel, hogy az xy lapjánál rögzített téglatestet a lappal párhuzamos erővel paralelepipedoná deformáljuk (5.2. ábra). A torzítást leíró transzformáció a z koordinátát változtatlanul hagyja, mialatt az x és y koordinátákat a z koordinátával arányosan módosítja. A torzító transzformációt *nyírásnak* (*shearing*) nevezzük.



5.2. ábra. Nyírás

A nyírás ugyancsak megadható egy mátrix művelettel:

$$\vec{r}' = \vec{r} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}. \quad (5.8)$$

5.2. Transzformációk homogén koordinátás megadása

Az idáig megismert transzformációk, az eltolást kivéve, mátrixszorzással is elvégezhetőek. Ennek különösen azért örülünk, mert ha egymás után több ilyen transzformációt kell végrehajtani, akkor a transzformációs mátrixok szorzatával (más néven *konkatenáljával*) való szorzás egyszerre egy egész transzformáció sorozaton átvezeti a pontot. Sajnos az eltolás ezt a szép képet eltorzítja. Más oldalról megközelítve a kérdést a lineáris transzformációknál az eredményül kapott koordináták az eredeti koordináták lineáris függvényei, tehát általános esetben:

$$\vec{r}' = \vec{r} \cdot \mathbf{A} + \vec{p}, \quad (5.9)$$

ahol \mathbf{A} egy mátrix, amely az elmondottak szerint jelenthet forgatást, nyírást, 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 mátrixba, mert egyszerűen ott már nincs erre hely. Ha azonban a mátrixot 4×4 -esre egészítjük ki, akkor már az eltolást is mátrixszorzással kezelhetjük. Ehhez persze a vektorunkat is ki kell bővíteni egy újabb koordinátával. Tekintsük ezen újabb koordinátát 1 értékűnek, hiszen ekkor

$$[\vec{r}', 1] = [\vec{r}, 1] \cdot \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 \\ A_{21} & A_{22} & A_{23} & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ p_x & p_y & p_z & 1 \end{bmatrix} = [r \cdot \mathbf{A} + \vec{p}, 1]. \quad (5.10)$$

A Descartes-koordináták egy újabb 1 értékű koordinátával történő kiegészítését a pont *homogén koordinátás alakjának* nevezzük.

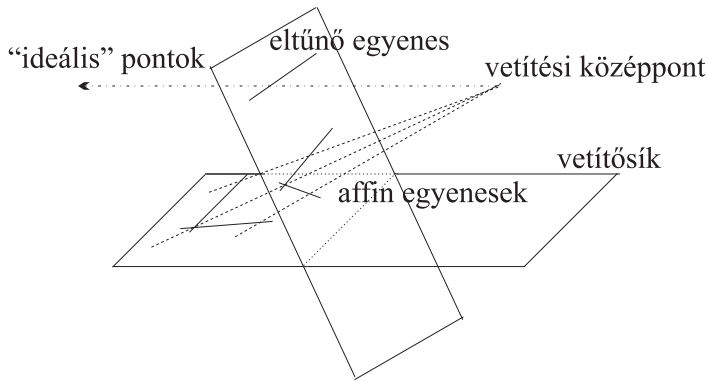
5.3. Transzformációk projektív geometriai megközelítése

Idáig pusztán annak érdekében, hogy az eltolást is beleerőltessük a mátrixművelettel megadható transzformációk körébe, a Descartes-koordinátákat kiegészítettük egy misztikus, konstans 1 értékű koordinátával, és az egészet elneveztük homogén koordinátás alaknak. Természetesen ennek a kis trükknek sokkal mélyebb matematikai tartalma van, amit ebben a fejezetben szeretnénk megvilágítani.

Mivel a számítógépes grafika 2 dimenziós képeket állít elő 3 dimenziós objektumokról, a képszintézisben mindenképpen megjelenik a *vetítés*, mint dimenziócsökkentő művelet. A természetes képalkotásnak leginkább megfelelő *középponti vetítés*, vagy más néven *centrális vetítés* kezelésénél azonban nehézségeket jelent, hogy az *euklideszi térben* nem minden pont vetíthető centrálisan. Ezért a transzformációinkat érdemes egy másfajta geometriára, az ún. *projektív geometriára* alapozni [Her91] [Cox74].

A projektív geometriát a középpontos vetítést tanulmányozva közelíthetjük meg (5.3. ábra). A képsíkkal párhuzamos vetítősugarakkal jellemezhető pontoknak nem felel meg képpont az euklideszi geometriában. Ezek a pontok a “végtelenbe” vetülnek, amely nem része az euklideszi térnek. Az euklideszi tér tehát lyukas. A projektív geometria ezeket a lyukakat tömi be oly módon, hogy kiegészíti az euklideszi teret újabb pontokkal. Az új pontok, ún. *ideális pontok*, az euklideszi térben nem vetíthető pontok vetületei lesznek. Az ideális pontok párhuzamos egyenesek illetve síkok “metszéspontjaként” is elképzelhetőek.

A számítástechnikában mindent számokkal jellemezünk, így a projektív tér pontjait is számokkal szeretnénk definiálni. Mivel a szokásos *Descartes-koordinátarendszer* és az euklideszi tér pontjai között kölcsönösen egyértelmű kapcsolat áll fenn, a Descartes-koordinátarendszer nyilván alkalmatlan az új pontok befogadására. A új algebrai alapot



5.3. ábra. Közepontos vetítés

jelentő homogén koordinátákat például egy mechanikai analógia segítségével vezetjük be.

Adjuk meg a pontjainkat, mint egy mechanikai rendszer súlypontját, amelyben egy p_1 referencia pontban X_h súlyt helyezünk el, egy p_2 referencia pontban Y_h súlyt, egy p_3 pontban Z_h súlyt és végül egy p_4 pontban w súlyt. A súlyok nem feltétlenül pozitívak, így a súlypont valóban bárhová kerülhet, ha a referencia pontok nem esnek egy síkba. Ha a referencia pontok az euklideszi tér pontjai és a $h = X_h + Y_h + Z_h + w$ összsúly nem zérus, akkor a súlypont nyilván az euklideszi térben marad.

Definíciószerűen nevezzük a (X_h, Y_h, Z_h, h) négyest ($h = X_h + Y_h + Z_h + w$) a súlypont homogén koordinátáinak. Az elnevezés onnan származik, hogy ha az összes súlyt ugyanazzal a skalárral szorozzuk, a súlypont nem változik, tehát minden, nem zérus λ -ra ekvivalensek a $(\lambda X_h, \lambda Y_h, \lambda Z_h, \lambda h)$ pontok.

Mivel a projektív tér része az euklideszi tér (az euklideszi tér pontjait az ideális pontokból megkülönböztetendő *affin pontok*nak nevezzük), és az euklideszi tér pontjait Descartes-koordinátákkal is megadhatjuk, ezekre a pontokra léteznie kell valamilyen összefüggésnek a Descartes-koordináták és a homogén koordináták között. Egy ilyen összefüggés felállításához a két koordináta rendszer 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 teljes súly nem zérus) az $\mathbf{i}, \mathbf{j}, \mathbf{k}$ Descartes-koordinátarendszerben

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

$$\frac{X_h}{h} \cdot \mathbf{i} + \frac{Y_h}{h} \cdot \mathbf{j} + \frac{Z_h}{h} \cdot \mathbf{k}. \quad (5.11)$$

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

$$x = \frac{X_h}{h}, \quad y = \frac{Y_h}{h}, \quad z = \frac{Z_h}{h}. \quad (5.12)$$

A negyedik koordinátával történő osztást *homogén osztásnak* nevezzük.

Homogén koordinátákkal síkokat is jellemezhetünk. Tekintsünk egy Descartes-koordinátarendszerben megadott síkot, amelynek egyenlete:

$$a \cdot x + b \cdot y + c \cdot z + d = 0. \quad (5.13)$$

Alkalmazzuk a homogén és Descartes-koordináták között fennálló összefüggést:

$$a \cdot \frac{X_h}{h} + b \cdot \frac{Y_h}{h} + c \cdot \frac{Z_h}{h} + d = 0 \implies a \cdot X_h + b \cdot Y_h + c \cdot Z_h + d \cdot h = 0. \quad (5.14)$$

Vegyük észre, hogy az ezen egyenletet kielégítő pontok köre nem változik, ha az egyenlet együtthatóit ugyanazzal a skalárral szorozzuk. Egy $[a, b, c, d]$ homogén koordináta négyes tehát nem csak pontokat, hanem síkokat is definiálhat. Ennek messzemenő következményei vannak a projektív geometriában. Az összes olyan tétel, amely pontokra érvényes, igaz lesz síkokra is. Az elvet általánosan *dualitásnak* nevezik.

Most terjesszük ki a vizsgálatunkat az ideális pontokra is. Ezek úgy képzelhetők el, mint a párhuzamos síkok közös pontjai, ezért a továbbiakban a síkok metszését tanulmányozzuk. Legyen a két párhuzamos sík homogén koordinátás alakja $[a, b, c, d]$ és $[a, b, c, d']$ ($d \neq d'$). A síkok egyenlete:

$$\begin{aligned} a \cdot X_h + b \cdot Y_h + c \cdot Z_h + d \cdot h &= 0, \\ a \cdot X_h + b \cdot Y_h + c \cdot Z_h + d' \cdot h &= 0. \end{aligned} \quad (5.15)$$

Formálisan mindkét egyenletet a következő pontok elégítik ki:

$$a \cdot X_h + b \cdot Y_h + c \cdot Z_h = 0 \quad \text{és} \quad h = 0. \quad (5.16)$$

Az euklideszi geometriában párhuzamos síkoknak nincs közös pontja, így ezek a közös pontok nem lehetnek az euklideszi térben, következésképpen a projektív tér ideális pontjai. Az ideális pontok homogén koordinátaiban tehát $h = 0$. Ezek az ideális pontok a "végtelent" reprezentálják, de oly módon, hogy különbséget tesznek különböző végtelenek között aszerint, hogy azok milyen (a, b, c) irányban találhatók.

A geometriai transzformációk a pontok koordinátáira értelmezett függvények. A számítógépes grafikában általában lineáris függvényeket használunk. Az euklideszi térben a lineáris transzformáció általános alakja a következő:

$$[x', y', z'] = [x, y, z] \cdot \mathbf{A}_{3 \times 3} + [p_x, p_y, p_z]. \quad (5.17)$$

Homogén lineáris transzformációk definiálásánál figyelembe kell vennünk, hogy a homogén koordináták invariánsak a skalárral történő szorzásra, ezért nem engedhetünk meg additív konstans a transzformációban:

$$[X'_h, Y'_h, Z'_h, h'] = [X_h, Y_h, Z_h, h] \cdot \mathbf{T}_{4 \times 4}. \quad (5.18)$$

Az euklideszi tér lineáris transzformációi a homogén lineáris transzformációk részhalmazát képezik, hiszen affin pontokra:

$$[x', y', z', 1] = [x, y, z, 1] \cdot \begin{bmatrix} & & & 0 \\ \mathbf{A}_{3 \times 3} & & & 0 \\ & & & 0 \\ \vec{p} & & & 1 \end{bmatrix}. \quad (5.19)$$

A transzformációs mátrixban a forgatást, nyírást, skálázást, stb. leíró 3×3 -as \mathbf{A} mátrix a \mathbf{T} mátrix bal felső minormátrixa, a \vec{p} eltolásvektor pedig az utolsó sorba kerül. A mátrix 4. oszlopa az euklideszi tér lineáris transzformációiban konstans $[0,0,0,1]$. Ha a 4. oszlopot megváltoztatjuk, akkor olyan homogén lineáris transzformációkhoz juthatunk, amelyek az euklideszi tér nem lineáris transzformációi.

Amennyiben egymás után több transzformációt kell végrehajtanunk, a pontokat definiáló vektorokat a transzformációs mátrixokkal kell végigszorozni. Ugyanezt a hatást érhetjük el, ha először előállítjuk a mátrixok szorzatát, és az eredő mátrixszal szorzunk. Végző soron bármilyen összetett transzformáció végrehajtható egyetlen 4×4 mátrix szorzással. A mátrixszorzás után, ha a 4. koordináta 1-től különböző, még a koordinátákat végig kell osztanunk a 4. koordinátával.

A homogén koordináták jól alkalmazhatók kvadratikus alakok (3.5.1. fejezet) és a racionális spline-ok, például a *NURBS* tárgyalásánál (3.3.3. fejezet). Egy Descartes-koordinátarendszerben a vezérlőpontokra racionális törtfüggvényekkel illesztett görbe ugyanis elképzelhető úgy is, mintha a homogén koordinátás térben normál polinomokkal illesztenénk.

5.4. Program: geometriai transzformációs osztály

A geometriai transzformációk pontokat és vektorokat változtatnak meg, ezért először a pontokat képviselő `Point3D` osztályt írjuk le, majd a `Vector3D` típust ennek hasonmásaként definiáljuk. A pontokon és a vektorokon a szokásos műveletek a negáció, összeadás, kivonás, skalárral történő szorzás és osztás, két vektor skaláris és vektoriális szorzatának képzése, a vektor hosszának meghatározása, a normalizálás és az egyes koordináták lekérdezése.

```
//=====
class Point3D {
//=====
    Coord x, y, z;
public:
    Point3D(Coord x0 = 0, Coord y0 = 0, Coord z0 = 0) {
        x = x0; y = y0; z = z0;
    }
    Point3D operator-( ) { return Point3D( -x, -y, -z ); }
    Point3D operator+( Point3D& p ) {
        return Point3D( x + p.x, y + p.y, z + p.z );
    }
    Point3D operator-( Point3D& p ) {
        return Point3D( x - p.x, y - p.y, z - p.z );
    }
    Point3D operator%( Point3D& v ) {
        return Point3D(y*v.z - z*v.y, z*v.x - x*v.z, x*v.y - y*v.x);
    }
    void operator+=( Point3D& p ) { x += p.x; y += p.y; z += p.z; }
    void operator/=( double d ) { x /= d; y /= d; z /= d; }
    Point3D operator*( double s ) { return Point3D( x*s, y*s, z*s ); }
    Point3D operator/( double s ) { return Point3D( x/s, y/s, z/s ); }
    double operator*( Point3D& p ) { return (x*p.x + y*p.y + z*p.z); }
    double Length( ) { return sqrt( x * x + y * y + z * z ); }
    void Normalize( ) { *this = (*this) * (1/Length()); }

    double& X() { return x; }
    double& Y() { return y; }
    double& Z() { return z; }
};

typedef Point3D Vector3D;
```

Egy homogén koordinátás pont a Descartes-koordinátahármasnak egy negyedik koordinátával történő kiegészítéseként adható meg.

```
//=====
class HomPoint3D : public Point3D {
//=====
    double H;
public:
    HomPoint3D(double X0=0, double Y0=0, double Z0=0, double h0=1 )
        : Point3D( X0, Y0, Z0 ) { H = h0; }
    HomPoint3D( Point3D& p ) : Point3D( p ) { H = 1.0; }
    double& h( ) { return H; }
    Point3D HomDiv( ) { return Point3D( X()/H, Y()/H, Z()/H ); }
    int IsIdeal( ) { return (H == 0); }
};
```

Végül egy homogén lineáris transzformáció egy 4×4 -es mátrixszal írható le. A mátrixelemeket a különböző jellegű transzformációk különböző módon inicializálják. A transzformációkat egymással konkatenálhatjuk, és alkalmazhatjuk azokat homogén koordinátás pontokra.

```
enum TransType {TRANSLATION, SCALE, ZROT, YSHEAR,
                ZSHEAR, AFFIN, PROJECTIVE};
HomPoint3D nv;

//=====
class Transform3D {
//=====
    double m[4][4];
public:
    Transform3D() { SetIdentity(); }
    Transform3D(TransType t, HomPoint3D v1, HomPoint3D v2 = nv,
                HomPoint3D v3 = nv, HomPoint3D v4 = nv) {

        switch( t ) {

            case TRANSLATION:
                SetIdentity( );
                m[3][0] = v1.X(); m[3][1] = v1.Y(); m[3][2] = v1.Z();
                break;

            case AFFIN:
                m[0][0] = v1.X(); m[0][1] = v1.Y(); m[0][2] = v1.Z();
                m[0][3] = 0;
                m[1][0] = v2.X(); m[1][1] = v2.Y(); m[1][2] = v2.Z();
                m[1][3] = 0;
                m[2][0] = v3.X(); m[2][1] = v3.Y(); m[2][2] = v3.Z();
                m[2][3] = 0;
                m[3][0] = v4.X(); m[3][1] = v4.Y(); m[3][2] = v4.Z();
                m[3][3] = 1;
                break;

            case PROJECTIVE:
                m[0][0] = v1.X(); m[0][1] = v1.Y(); m[0][2] = v1.Z();
                m[0][3] = v1.h();
                m[1][0] = v2.X(); m[1][1] = v2.Y(); m[1][2] = v2.Z();
                m[1][3] = v2.h();
                m[2][0] = v3.X(); m[2][1] = v3.Y(); m[2][2] = v3.Z();
                m[2][3] = v3.h();
                m[3][0] = v4.X(); m[3][1] = v4.Y(); m[3][2] = v4.Z();
                m[3][3] = v4.h();

        }

    }
}
```

```

Transform3D( TransType t, double s1, double s2=0, double s3=0 ) {
    SetIdentity( );
    switch( t ) {
    case SCALE:
        m[0][0] = s1; m[1][1] = s2; m[2][2] = s3;
        break;
    case ZROT:
        m[1][1] = cos(s1);    m[1][2] = sin(s1);
        m[2][1] = -sin(s1);   m[2][2] = cos(s1);
        break;
    case ZSHEAR:
        m[2][0] = s1;    m[2][1] = s2;
        break;
    }
}

void SetIdentity( ) {
    for(int i = 0; i < 4; i++)
        for(int j = 0; j < 4; j++) m[i][j] = (i == j);
}

Transform3D operator*( Transform3D& tr ) {
    Transform3D res;
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            res.m[i][j] = 0;
            for(int k = 0; k < 4; k++)
                res.m[i][j] += m[i][k] * tr.m[k][j];
        }
    }
    return res;
}

void operator*=( Transform3D& tr ) { *this = (*this) * tr; }

HomPoint3D Transform( HomPoint3D& p ) {
    HomPoint3D res;
    res.X() = p.X() * m[0][0] + p.Y() * m[1][0] +
              p.Z() * m[2][0] + p.h() * m[3][0];
    res.Y() = p.X() * m[0][1] + p.Y() * m[1][1] +
              p.Z() * m[2][1] + p.h() * m[3][1];
    res.Z() = p.X() * m[0][2] + p.Y() * m[1][2] +
              p.Z() * m[2][2] + p.h() * m[3][2];
    res.h() = p.X() * m[0][3] + p.Y() * m[1][3] +
              p.Z() * m[2][3] + p.h() * m[3][3];
    return res;
}
};

```


6. fejezet

Virtuális világmodellek tárolása

A modellezés során a számítógépbe bevitt információt a program adatszerkezetekben tárolja. Az adatszerkezetek többféleképpen kialakíthatóak. Az egyes struktúrák különböző mértékben illeszkednek az adott modellezési folyamathoz, illetve a megjelenítéshez.

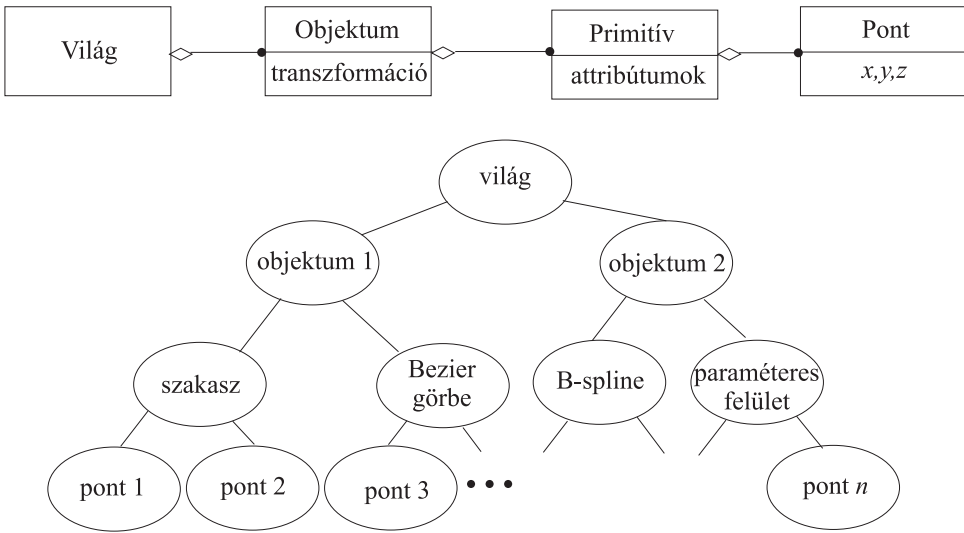
6.1. Hierarchikus adatszerkezet

A virtuális világ szerkezete hierarchikus. A világ objektumokat tartalmaz, az objektumok primitív objektumokat, a primitív objektumok geometriáját pedig leggyakrabban pontok, ritkábban paraméterek határozzák meg (például egy paraméteres görbe reprezentálható a vezérlőpontjaival vagy a polinomegyütthatóival). A hierarchikus felépítésnek megfelelő objektum-modell az 6.1. ábrán látható.

Egy *objektum* szokásos *attribútumai*: az objektum neve, a *modellezési transzformációja*, a 2D képszintézisben használt *prioritása*, a képszintézis gyorsítását szolgáló *befoglaló doboz*, stb. A *primitíveknek* többféle típusa lehetséges, úgy mint szakasz, görbe, felület, poligonháló, test, stb. A primitívek attribútumai a primitív típusától függnnek.

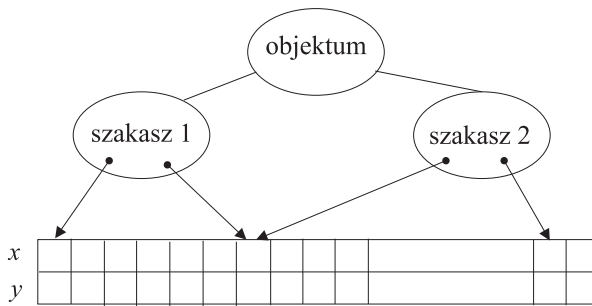
6.2. A geometria és topológia szétválasztása

Az 6.1. 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 objektu-



6.1. ábra. A világleírás osztály és objektum diagramja

mokból és egy közös tömbben foglaljuk össze. Az objektumokban csupán mutatókat helyezünk el a saját pontok azonosítására (6.2. ábra).



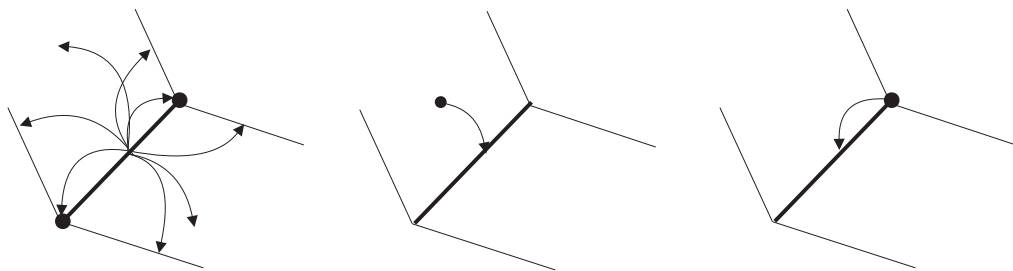
6.2. ábra. A világleírás kiemelt geometriai információval

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, hogy egy objektum mely primitívekből áll és a primitíveknek melyek a definíciós pontjai.

A hierarchikus modellel szemben a következő kifogásunk az lehet, hogy az adatstruktúrából nem olvasható ki közvetlenül a teljes *topológiai* 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 objektumoknak a gyűjteménye, vagy elfajult korcsok is az objektumaink közé keveredtek. A beteg objektumok kiszűrése nem a képszintézis miatt fontos, hanem azért, mert a modell alapján esetleg szeretnénk térfogatot számítani, vagy egy NC szerszámgéppel legyártatni a tervezett objektumot.

A teljes topológiai információ az illeszkedéseket kifejező mutatók beépítésével reprezentálható. Egy ilyen modell a 3D felületi modellek tárolására kifejlesztett *szárnyas-él adatstruktúra* [Bau72] (6.3. ábra), amelyben minden illeszkedési relációt mutatók fejeznek ki. Az adatszerkezet központi eleme az él, amelyben mutatókkal hivatkozunk a két végpontra, az él jobb és bal oldalán lévő lapokra, és a két lapon a megelőző és a következő élekre. A végpontok ugyancsak hivatkoznak az egyik illeszkedő élre, amiből a mutatókon keresztül már minden illeszkedő él előállítható. Hasonlóképpen a lapok is hivatkoznak egyik élükre, amelyből az összes határgörbe származtatható.



6.3. ábra. Szárnyas él adatstruktúra

A szárnyas él adatstruktúrát általában a topológiai helyességet hangsúlyozó *B-rep modellezők* használják.

6.3. CSG-fa

A hierarchikus modell más irányú javításához juthatunk, ha nem korlátozzuk a hierarchia mélységét, és egy szinten nem csupán az alacsony szintű objektumok egyesítését, hanem bármilyen halmazműveletet megengedünk. Mivel a halmazműveletek általában binárisak, a keletkező modell egy bináris fa, amelynek levelei primitív testeket képviselnek, a többi csomópontja pedig a gyermekobjektumokon végrehajtott halmazműveletet (3.14. ábra). Ezen modell különösen jól illeszkedik a konstruktív tömörtest geometriához, ezért a bináris fa szokásos elnevezése a *CSG-fa*.

6.4. Megjelenítő állományok

Ismét másfajta adatstruktúrához juthatunk, ha a lehető leggyorsabb megjelenítés érdekében alakítjuk át a szerkezetet. A megjelenítés gyorsítását szem előtt tartva, a primitív objektumok és pontok elérésénél nem alkalmazunk mutatókat, és kihasználjuk, hogy az egymást követő primitívek attribútumai igen gyakran megegyezőek, ezért nem érdemes az attribútumokat minden primitívénél tárolni, majd a megjelenítés során beállítani. Ehelyett az éppen érvényes attribútumok értékét külön táblázatban tároljuk, és ezek addig érvényben maradnak, amíg egy attribútumállító primitív meg nem változtatja valamelyiket.

A mutatók kiküszöbölése az adatstruktúra összetolását jelenti. Az összetolás során problémát jelent, hogy a különböző típusú primitívek mérete általában eltérő. A megoldást a változó hosszú utasításkészletű processzorok programtárolási módszere adja. Helyezzük el a modellt egy bájtokat tartalmazó tömbben, ahol minden primitív egy vagy több bájtot foglal el. Minden primitív első bájta a primitív típusát azonosítja, amelyből az is kiderül, hogy még hány bájt tartozik a primitívhez. Egyrészt ezen tény miatt, másrészt pedig amiatt, mert az attribútum állításokat elválasztottuk a primitívektől, a primitíveket nem lehet tetszőleges sorrendben feldolgozni, kizárólag a leírás sorrendjében.

Ezek után a lineáris adatstruktúrát, más néven *megjelenítő állományt* úgy is elképzelhetjük, mint egy gépi kódú programot, amelynek utasításai a vonalhúzás, vonalszín beállítás, poligon rajzolás, stb. A Tektronix grafikus munkaállomásokban például a grafikus processzor utasításkészlete éppen ilyen, ezért a megjelenítés úgy történik, hogy a grafikus processzor memóriájába beírjuk a megjelenítő állományt, majd a grafikus processzor végrehajtja a "programot".

6.5. Szabványos világmodellek

Az állományokban tárolt virtuális világ szerkezetére számos több, szélesebb körben elfogadott, szabványos megoldás ismeretes. Ezek egy része valóban termékfüggetlen és szabványnak tekinthető (*IGES*, *NFS*, *MGF*, stb.). Másik részük viszont csak elterjedt modellező vagy képszintézis programok leíró nyelvei (*POVRAY*, *3D-Studio*, *AutoCad*, *Open-Inventor*, stb.). Ha magunk írunk grafikus rendszert, akkor azt is célszerű felkészíteni valamely elterjedt formátum megértésére, mert ebben az esetben könnyen átvehetjük a mások által sok fáradság árán létrehozott modelleket. Elegendő egy gyakori formátum értelmezését beépíteni a programba, hiszen léteznek és elérhetőek olyan konverziós programok, amelyek a szabványos formákat egymásba átalakítják.

6.6. Program: hierarchikus 3D adatszerkezet

Az adatszerkezetet megvalósító osztályokat a hierarchia szerint alulról felfelé építjük fel. A hierarchia alján a primitívek vannak, amelyek az általános `Primitive3D` osztályból származtathatók. Egy primitív geometriáját pontokkal írjuk le, megjelenítéséhez pedig legalább a színe szükséges. A pontokat a dinamikusan nyújtózkodó tömböt megvalósító generikus `Array` osztályból paraméterezett `Array<Point3D>` típusú `points` tömbben tároljuk, a színt pedig az R, G, B komponenseket tartalmazó `Color` típusú változó jellemzi. Az `Array` osztály deklarációját a 3.7. fejezetben találhatjuk meg.

```
//=====
class Primitive3D {
//=====
    Array<Point3D> points;
    Color          color;
public:
    Primitive3D( Color c = 0, int n = 0 ) : points(n), color( c ) { }
    Color Col() { return color; }
    Point3D& Point( int i ) { return points[i]; }
    int      PointNum( ) { return points.Size(); }
};
```

A `PolyLine3D` az általános primitívben tárolt pontokat egyenes szakaszokkal köti össze. A poligon megadásához egyéb adatra nincs szükség, így ebben az osztályban nem definiálunk új adattagokat és tagfüggvényeket.

```
//=====
class PolyLine3D : public Primitive3D {
//=====
public:
    PolyLine3D( Color& c ) : Primitive3D( c ) { }
};
```

A `Curve3D` osztály egy általános görbetípust képvisel. Az alaposztálytól örökölt pontok most a görbe vezérlő pontjai. A görbe pontjait a vezérlő pontokból a görbe típusának megfelelő interpolációs vagy approximációs eljárással állíthatjuk elő, amelyet az `Interpolate` tagfüggvény valósít meg. A különböző görbetípusok ezt a tagfüggvényt másképpen implementálják, amelyhez más polinomfüggvényt használnak fel.

```
//=====
class Curve3D : public Primitive3D {
//=====
public:
    Curve3D( Color& c ) : Primitive3D( c ) { }
    virtual Point3D Interpolate( double tt ) = 0;
};
```

A LagrangeCurve3D az interpolációhoz a LagrangePolinom osztályban definiált $L(i, tt)$ Lagrange interpolációs függvényt alkalmazza.

```
//=====
class LagrangeCurve3D : public Curve3D, public LagrangePolinom {
//=====
public:
    LagrangeCurve3D( Color c ) : Curve3D( c ), LagrangePolinom( ) { }
    Point3D Interpolate( double tt ) {
        Point3D rr(0, 0, 0);
        for(int i = 0; i < Degree(); i++) rr += Point(i) * L(i, tt);
        return rr;
    }
};
```

A BezierCurve3D az approximációhoz a BezierPolinom osztályban definiált $B(i, tt, m)$ *Bernstein-polinomokat* használja.

```
//=====
class BezierCurve3D : public Curve3D, public BezierPolinom {
//=====
public:
    BezierCurve3D( Color c ) : Curve3D( c ), BezierPolinom( ) { }
    Point3D Interpolate( double tt ) {
        double Bi = 1.0;
        Point3D rr(0, 0, 0);
        for(int i = 0; i < PointNum(); i++)
            rr += Point(i) * B(i, tt, PointNum());
        return rr;
    }
};
```

A PolyFace3D osztály háromszögeket tárol, amelyek tetszőleges felületeket közelíthetnek. Most az alapsztálytól örökölt pontok a háromszögek csúcspontjai.

```
//=====
class PolyFace3D : public Primitive3D {
//=====
public:
    PolyFace3D( Color& c ) : Primitive3D( c ) { }
};
```

Egy 3D objektum (Object3D) tetszőleges számú és típusú primitívet tartalmazhat, ezért itt a primitívek címeit tároljuk. Másrészt az objektumhoz egy modellezési transzformáció tartozik, amely elhelyezi az objektumot a világ-koordináta-rendszerben.

Az osztály lehetőséget ad arra, hogy egy új primitívet az objektumhoz vegyünk (AddPrimitive), egy adott primitívet (Primitive), illetve a primitívek számát (PrimitiveNum) lekérdezzük, és hogy a transzformációhoz hozzáférjünk (Transform).

```
//=====
class Object3D {
//=====
    Array<Primitive3D *> prs;
    Transform3D          tr;
public:
    void AddPrimitive( Primitive3D * p ) { prs[ prs.Size() ] = p; }
    Primitive3D * Primitive( int i ) { return prs[i]; }
    Transform3D& Transform( ) { return tr; }
    int PrimitiveNum() { return prs.Size(); }
};
```

A virtuális világ (VirtualWorld) objektumokból áll, amelyhez új objektumokat lehet hozzávenni (AddObject), le lehet kérdezni az objektumok számát (ObjectNum) és az egyes objektumokat (Object).

```
//=====
class VirtualWorld {
//=====
    Array<Object3D *> objs;
public:
    VirtualWorld( ) : objs( 0 ) { }
    Object3D * Object( int o ) { return objs[o]; }
    void AddObject( Object3D * o ) { objs[ objs.Size() ] = o; }
    int ObjectNum() { return objs.Size(); }
};
```


7. fejezet

A 2D képszintézis

A *2D képszintézis* célja az objektumok képernyőre transzformálása és a nézetbe eső részek megjelenítése. A megjelenítés a rasztertár megfelelő pixeleinek kiszínezésével történik. Az objektumok geometriai definíciója a lokális modellezési koordináta-rendszerben áll rendelkezésre, így a képszintézis innen indul. Először az objektumokat egy közös világ-koordináta-rendszerbe transzformáljuk. A 2D világ-koordináta-rendszerben megadjuk a 2D kamerát jelképező ablakot. Az ablak belsejébe eső elemeket a képernyő-koordináta-rendszer nézetébe vetítjük, és itt megkeressük a geometriai elemeket közelítő pixeleket.

A képszintézis során a következő feladatokat végezzük el:

1. *Vektorizáció*: A virtuális világban tárolt szabadformájú elemeket (például körívek, interpolációs vagy spline görbék) pontok, szakaszok és poligonok halmazával közelítjük.
2. *Transzformáció*: a lokális koordináta-rendszerben adott geometriai elemekre a világ-koordináta-rendszerben, majd a képernyő koordináta-rendszerben van szükségünk. A koordináta-rendszer-váltás homogén lineáris geometriai transzformációt igényel, amelyet mátrixszorzással fogunk realizálni.
3. *Vágás*: A képernyőn csak az jeleníthető meg, ami a nézet téglalapján belül van, így azon geometriai elemeket, illetve a geometriai elemek azon részeit, amelyek a nézet, illetve az ablak téglalapján kívülre kerülnek, el kell távolítani.
4. *Páztakonverzió*, vagy más néven *raszterizáció*: A képernyő-koordináta-rendszerbe transzformált és a nézet belsejébe eső geometriai elemek képét pixelek átszínezésével közelítjük.

7.1. ábra. A 2D képszintézis lépései

7.1. Vektorizáció

A vektorizáció a szabad formájú elemeket pontokkal, szakaszokkal és poligonokkal közelíti. Erre a lépésre azért van szükség, mert a transzformációs és a vágási lépéseket a pontokra, szakaszokra és poligonokra viszonylag könnyen végre tudjuk hajtani, ráadásul ezek a lépések nem változtatják meg az elem típusát. Gondoljunk arra, hogy egy szakasz a homogén lineáris transzformáció és a vágás után is szakasz lesz. A szakasz transzformációja a végpontok transzformációjával elvégezhető, a vágáshoz pedig lineáris egyenleteket kell megoldani. Ezzel szemben ha például egy kört transzformálnánk, abból könnyen születhet a körnél sokkal bonyolultabb képződmény.

A vektorizáció során mind az önálló görbéket, mind a területek határait egyenes szakaszok sorozatával közelítjük, így a görbékből szakaszok, a területekből poligonok keletkeznek.

Tegyük fel, hogy a görbe $\vec{r} = \vec{r}(t), t \in [0, 1]$ explicit egyenletével adott. Egy megfelelő szakasz sorozat előállítható, ha a $[0, 1]$ tartományon kijelöljük a $t_0 = 0 < t_1 < \dots < t_{n-1} < t_n = 1$ pontokat, és a görbe ilyen paraméterértékeknek megfelelő pontjait szakaszokkal kötjük össze. A közelítő szakaszok végpontjai a következők:

$$[\vec{r}(t_0), \vec{r}(t_1)], [\vec{r}(t_1), \vec{r}(t_2)], \dots, [\vec{r}(t_{n-1}), \vec{r}(t_n)].$$

Az n és t_0, t_1, \dots, t_n paraméterek megválasztása jelentős mértékben meghatározza a közelítés pontosságát. Kiválasztásuk heurisztikus módszerrel történhet. Például lehet a kiválasztás célja az, hogy a keletkező szakaszok képernyő koordinátarendszerbeli képe ne legyen hosszabb néhány pixelnél, hiszen ekkor a raszterizáció hibája mellett az egyenes szakaszokkal történő közelítés hibája elhanyagolható.

7.2. Modellezési transzformáció

A *modellezési transzformáció* a koordinátákat a lokális modellezési koordinátarendszerből a világ-koordinátarendszerbe viszi át. A modellezési transzformáció a *koordinátarendszer váltó transzformációk* egy esete.

7.2. ábra. 2D modellezési transzformáció

Tegyük fel, hogy a geometria az \vec{a}, \vec{b} lokális modellező koordinátarendszerben ismert, de nekünk az x, y világ-koordinátarendszerbeli koordinátákra van szükségünk. Tegyük fel továbbá, hogy a lokális modellező koordinátarendszer \vec{a} és \vec{b} egységvektori, valamint \vec{o} origója a világ-koordinátarendszerben adott:

$$\vec{a} = [a_x, a_y], \quad \vec{b} = [b_x, b_y], \quad \vec{o} = [o_x, o_y]. \quad (7.1)$$

Rendeljük össze egy \vec{p} pont x, y világ-koordinátarendszerbeli koordinátáit, az α, β lokális modellezési koordinátarendszerbeli koordinátáival:

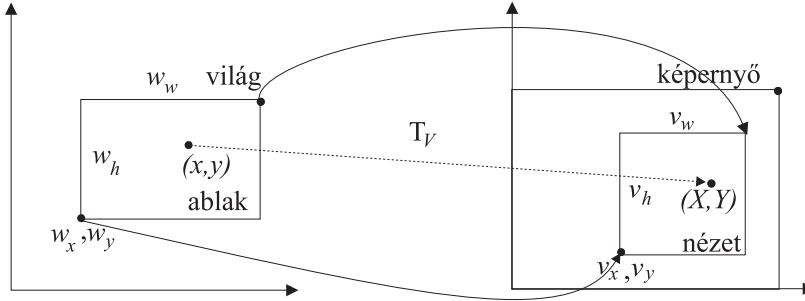
$$\vec{p} = \alpha \cdot \vec{a} + \beta \cdot \vec{b} + \vec{o} = [x, y]. \quad (7.2)$$

Ezen egyenlet homogén koordinátás alakban ugyancsak felírható:

$$[x, y, 1] = [\alpha, \beta, 1] \cdot \begin{bmatrix} a_x & a_y & 0 \\ b_x & b_y & 0 \\ o_x & o_y & 1 \end{bmatrix} = [\alpha, \beta, 1] \cdot \mathbf{T}_M. \quad (7.3)$$

7.3. Ablak-nézet transzformáció

Az ablak-nézet transzformáció a 2D sík pontjait eltolja, és a két koordinátatengely mentén nagyítja oly módon, hogy az ablak téglalapjának a képe a nézet téglalapja legyen (7.3. ábra). Legyen az ablak bal alsó sarka a (w_x, w_y) pont, szélessége w_w , magassága w_h . Hasonlóképpen a nézet bal alsó sarka (v_x, v_y) , szélessége v_w , magassága v_h .



7.3. ábra. Ablak-nézet transzformáció

Miként behelyettesítéssel könnyen meggyőződhetünk róla, az ablak sarkait a nézet sarkaiba átvivő transzformáció:

$$X = \frac{x - w_x}{w_w} \cdot v_w + v_x, \quad Y = \frac{y - w_y}{w_h} \cdot v_h + v_y. \quad (7.4)$$

Mátrix alakban:

$$\mathbf{T}_V = \begin{bmatrix} v_w/w_w & 0 & 0 \\ 0 & v_h/w_h & 0 \\ v_x - v_w \cdot w_x/w_w & v_y - v_h \cdot w_y/w_h & 1 \end{bmatrix}. \quad (7.5)$$

7.4. A modellezési és az ablak-nézet transzformációk összefűzése

A képszintézis egy pont lokális modellezési koordinátarendszerbeli \vec{r}_l koordinátái alapján először meghatározza a világ-koordinátarendszerbeli \vec{r}_w koordinátákat, majd kiszámítja a képernyő-koordinátarendszerbeli \vec{r}_s koordinátákat.

$$\vec{r}_w = \vec{r}_l \cdot \mathbf{T}_M, \quad \vec{r}_s = \vec{r}_w \cdot \mathbf{T}_V. \quad (7.6)$$

Két mátrixszorzás helyett ugyanez egyetlen mátrixszorzással is megoldható, ha a szorzásban a modellezési és az ablak-nézet transzformációk konkatenáltját, az ún. *összetett*

transzformációt (\mathbf{T}_C) használjuk:

$$\mathbf{T}_C = \mathbf{T}_M \cdot \mathbf{T}_V, \quad \vec{r}_s = \vec{r}_i \cdot \mathbf{T}_C. \quad (7.7)$$

7.5. 2D vágás

A 2D vágással a geometriai elemek azon részét távolítjuk el, amelyek a vágási téglalapon kívülre esnek. A vágási téglalap lehet az ablak téglalapja, amennyiben a világkoordináta-rendszerben végezzük ezt a műveletet, vagy a nézet téglalapja, ha a vágásra a képernyő-koordináta-rendszerben kerül sor. Ha a virtuális világ közvetlenül világkoordinátákban adott, akkor célszerű az ablakra vágni, hiszen ekkor az eldobott elemek transzformációját megtakaríthatjuk. Ha viszont a virtuális világ egyes objektumai külön modellezési koordináta-rendszerben definiáltak, akkor a nézetre vágás jár kevesebb művelettel. Az ablakra vágás ilyenkor egy modellezési transzformációt igényel minden elemre, majd a vágáson átjutott elemekre egy ablak-nézet transzformációt. Ezzel szemben a nézetre vágáshoz minden pontra egyetlen összetett transzformáció szükséges.

A vágás folyamata természetesen független attól, hogy ablakról vagy nézetről van szó, hiszen mindkét esetben a vágási tartomány egy koordinátatengelyekkel párhuzamos oldalú téglalap. Jelöljük ezen téglalap minimális és maximális koordinátáit $x_{\min}, y_{\min}, x_{\max}, y_{\max}$ -szal.

Emlékezzünk vissza, hogy a vektorizáció jóvoltából csak pontok, szakaszok és poligonok vágásával kell foglalkoznunk. A pontok vágása könnyen elintézhető. Egy x, y pont a téglalapon belül van, ha az alábbi egyenlőtlenségek teljesülnek:

$$x \geq x_{\min}, \quad x \leq x_{\max}, \quad y \geq y_{\min}, \quad y \leq y_{\max}. \quad (7.8)$$

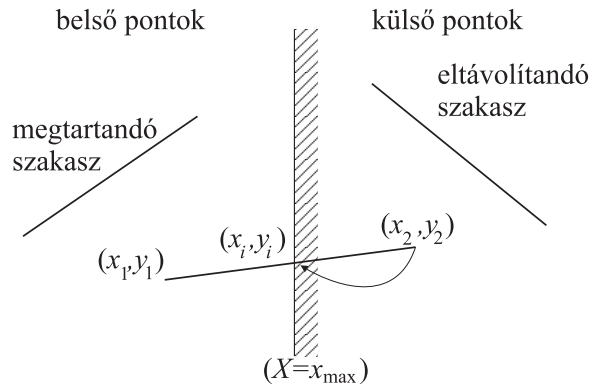
Vegyük észre, hogy az egyes egyenlőtlenségek azt mutatják, hogy az adott pont a téglalapot határoló 4 egyenesnek a téglalap felőli vagy azon túli oldalán van! Ez lényegében azt jelenti, hogy a pont objektum téglalapra vágását az objektum 4 félsíkra való vágására vezettük vissza. Mivel a téglalap a négy félsík metszete, egy pont akkor és csak akkor belső pontja a téglalagnak, ha mind a négy félsík tekintetében belső pont. Ezt érdemes megjegyezni, mert a szakasz- és a poligonvágás is ugyanezre az elvre épül.

Szakaszok vágása

Vezessük vissza az $[(x_1, y_1), (x_2, y_2)]$ végpontokkal definiált szakasz téglalapra vágását félsíkokra történő vágásra. Például tekintsük az $x \leq x_{\max}$ félsíkot (a többi teljesen analóg módon kezelhető).

Három esetet kell megkülönböztetni:

1. Ha a szakasz mindkét végpontja belső pont, akkor a teljes szakasz belső pontokból áll, így megtartjuk.



7.4. ábra. A szakasz és félsík határának lehetséges elhelyezkedése és metszéspontja

2. Ha a szakasz mindkét végpontja külső pont, akkor a szakasz minden pontja külső pont, így a vágás a teljes szakaszt eltávolítja.
3. Ha a szakasz egyik végpontja külső pont, a másik végpontja belső pont, akkor ki kell számítani a szakasz és a félsík határának metszéspontját, és a külső végpontot fel kell cserélni a metszésponttal. Figyelembe véve, hogy a szakasz pontjai kielégítik a

$$x(t) = x_1 + (x_2 - x_1) \cdot t, \quad y(t) = y_1 + (y_2 - y_1) \cdot t$$

egyenletet, a félsík határa pedig kielégíti az $x = x_{\max}$ egyenletet, a metszéspont t_i paraméterét és (x_i, y_i) koordinátáit a következőképpen határozhatjuk meg:

$$x_{\max} = x(t_i) = x_1 + (x_2 - x_1) \cdot t_i \quad \implies \quad t_i = \frac{x_{\max} - x_1}{x_2 - x_1} \quad \implies$$

$$(x_i, y_i) = (x_{\max}, y_1 + (y_2 - y_1) \cdot \frac{x_{\max} - x_1}{x_2 - x_1}). \quad (7.9)$$

A szakasz téglalpra vágásához ezt az algoritmust mind a négy félsíkra végre kell hajtani. Ez az algoritmus ugyan jól működik, de nem elég gyors. A gyorsításhoz észre kell vennünk, hogy a vágás három felvázolt esetéből az első kettő gyorsan megoldható, de a harmadik lényegesen nagyobb számítási időt igényel. Ha a négy félsíkra vágást egymástól függetlenül, szekvenciálisan hajtjuk végre, akkor előfordulhat, hogy az egyik félsíkra végigkínlódjuk a harmadik esetet, majd rájövünk, hogy a következő félsík tekintetében a szakasz teljes egészében külső, így eldobandó. Érdemes lenne az egyszerű logikai műveleteket előrehozni, és a metszéspontszámítással csak végső esetben foglalkozni. Meglepő módon már ezt az egyszerűnek látszó feladatot is nagyon sokféleképpen

lehet megoldani, így sok különböző vágóalgorithmus ismeretes [CB78, LB84, Duv90]. Mi csupán a legelterjedtebbel fogunk megismerkedni, amelyet *Cohen-Sutherland vágási algoritmus*nak neveznek.

| | | | |
|------|------|-----------------|-------------------|
| 0110 | 0100 | 1100 | |
| 0010 | 0000 | 1000 | triviális eldobás |
| 0000 | 0001 | 1001 | |

triviális elfogadás

7.5. ábra. A sík pontjainak kódolása

Rendeljünk minden egyes félsíkhoz egy bitet, amely egy pontra 1 értékű, ha a pont a félsíkon belül van, és 0, ha kívül. A szakasz végpontjait ily módon egy-egy 4 bites kóddal jellemezhetjük (7.5. ábra).

Ha mindkét végpont kódja 0000, a végpontok a félsíkokon belül vannak, tehát a vágási téglalapon is belül vannak. Ekkor a szakasz összes pontja a vágási téglalapon belül van. Ezt az esetet nevezzük *triviális elfogadás*nak. Ha viszont valamelyik bitben mindkét végpont kódja 1, akkor mindkét végpont a bitnek megfelelő félsíkon kívül van, így a teljes szakasz eldobandó. Ez a *triviális eldobás* esete. Ezen két esetet anélkül sikerült elintézni, hogy egyetlen metszéspontot is számítani kellett volna. Ha egyik eset sem áll fenn, akkor van legalább egy olyan bit, amelyben az egyik végpont 0, míg a másik 1 értékű. Ez azt jelenti, hogy van egy olyan félsík, amelyhez képest az egyik végpont külső pont, míg a másik belső. Az előző algoritmussal erre a félsíkra kell a metszéspontot meghatározni és a külső pontot a metszésponttal felváltani. Az új szakaszra a vizsgálatok újra kezdhetők.

Összefoglalásképpen bemutatjuk a Cohen-Sutherland vágás programját. A program bemeneti paraméterként kapja a szakasz két végpontját. Az algoritmus a visszatérési értékében jelzi, hogy a vágás eredményeképpen maradt-e valami a szakaszból, és ha igen, akkor a bemeneti paraméterek módosításával végzi el a vágást.

```

BOOL LineClipping( $P_1, P_2$ )
   $C_1 = \text{Code}(P_1), C_2 = \text{Code}(P_2)$ 
  loop
    if ( $C_1 = 0$  AND  $C_2 = 0$ ) then return TRUE           //elfogad
    if ( $C_1 \& C_2 \neq 0$ ) then return FALSE           //eldob
     $f = \text{Bit index ahol } C_1 \ll C_2$ 
     $P^* = A(P_1, P_2)$  szakasz és a  $f$ . felsík határ metszéspontja
     $C^* = \text{Code}(P^*)$ 
    if  $C_1[f] = 1$  then  $P_1 = P^*, C_1 = C^*$ ;
    else  $P_2 = P^*, C_2 = C^*$ 
  endloop
end

```

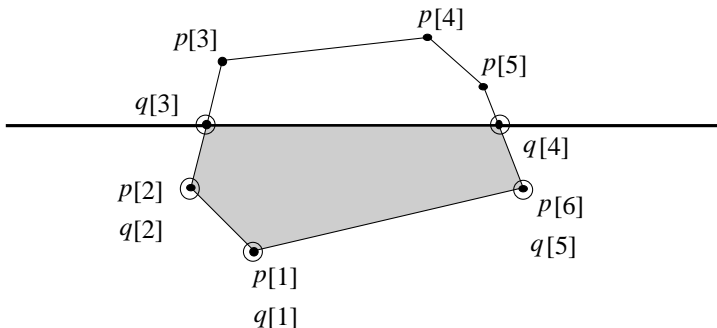
Először a C_1 és C_2 kódokat számítjuk ki a végpontok x, y koordinátáira az

$$x < x_{\min}, y < y_{\min}, x > x_{\max}, y > y_{\max}$$

relációk ellenőrzésével. A hurokban először a triviális elfogadás és eldobás lehetőségét ellenőrizzük, majd ha egyikkel sem élhetünk, kiválasztjuk azt a felsíkot, amelynek megfelelő kódbit a két végpontra eltérő. A szakasz és a felsík határoló egyenese közötti metszéspontot a 7.9 egyenlet megfelelő változatával határozhatjuk meg. A rossz oldalon, azaz az 1. kódbittel rendelkező végpontot a metszéspontra cseréljük, majd a következő ciklusban ismét a triviális eldobás illetve elfogadás esetével próbálkozunk.

Poligonok vágása

A poligonok téglalpra vágását is 4 egymás után végrehajtott felsíkra vágással realizáljuk.



7.6. ábra. Poligonvágás

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 (7.6. ábra).

Tegyük fel, hogy az eredeti poligonunk pontjai a $p[0], \dots, p[n-1]$ tömbben érkeznek, a vágott poligon csúcsait pedig a $q[0], \dots, q[m-1]$ tömbbe kell elhelyezni. A vágott poligon csúcsait az m változóban számoljuk. Az implementáció során egy kis apró kellemetlenséget okoz az, hogy általában az i . csúcst követő csúcs az $i+1$., kivéve az utolsó, az $n-1$. csúcs esetében, hiszen az ezt követő a 0. Ezt a kellemetlenséget elháríthatjuk, ha a p tömböt kiegészítjük még egy $(p[n] = p[0])$ elemmel, amely még egyszer tárolja a 0. elemet. Ezek után a vágóalgoritmus, amely a *Sutherland-Hodgeman-poligonvágás* [SH74] nevet viseli:

```

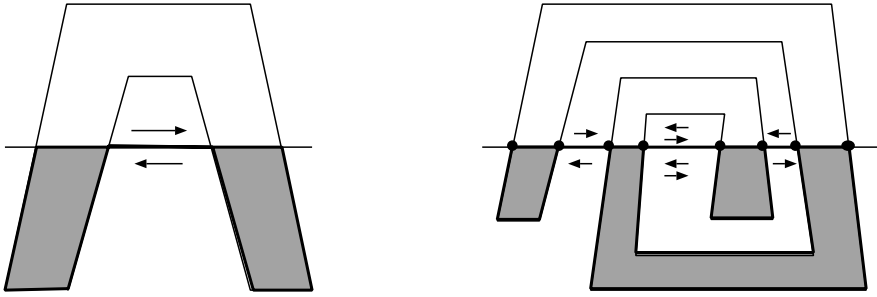
PolygonClipping( $p[n] \rightarrow q[m]$ )
   $m = 0$ 
  for  $i = 0$  to  $n - 1$  do
    if  $p[i]$  belső pont then
       $q[m++] = p[i]$ 
      if  $p[i + 1]$  külső pont then
         $q[m++] = (p[i], p[i + 1])$  szakasz és a félsík határ metszéspontja
      endif
    else
      if  $p[i + 1]$  belső pont then
         $q[m++] = (p[i], p[i + 1])$  szakasz és a félsík határ metszéspontja
      endif
    endif
  endfor
end

```

Az algoritmusban a “belső pont” illetve a “külső pont” vizsgálatokhoz a félsík határoló egyenesének koordinátáját és a csúcspont koordinátáját hasonlítjuk össze. Például az ablak jobb oldalánál ha $x > x_{\max}$ akkor a pont külső, egyébként belső.

Alkalmazzuk gondolatban a vágóalgoritmust olyan konkáv poligonra, amelynek a vágás következtében több részre kellene esnie. (7.7. ábra). Az egyetlen tömböt produkáló algoritmus képtelen a széteső részek elkülönítésére, és azokra a helyekre, ahol valójában nem keletkezhet él, páros számú élt hoz létre.

A különálló részeket összekapcsoló páros számú él nem okoz problémát a kitöltésnél, ha olyan kitöltő algoritmust használunk, amely a belső pontokat a következő elv



7.7. ábra. Konkáv poligonok vágása

alapján határozza meg: a kérdéses pontból egy félegyenest indítunk a végtelen felé és megvizsgáljuk, hogy az hányszor metszi a poligon határát. Páratlan számú metszéspont esetén a pontot belső pontnak tekintjük, egyébként a pont külső pont.

7.6. 2D raszterizáció

A *raszterizáció* során azokat a pixeleket azonosítjuk, amelyek átszínezésével a képernyő-koordinátarendszerbe transzformált geometriai alakzat formáját közelíthetjük.

A raszterizáció alapobjektuma a pixel, a geometriai primitívekkel dolgozó korábbi lépésekkel ellentétben. Előfordulhat, hogy egy geometriai objektum rajzolásához nagyon sok pixel kell (szakaszoknál 1000, területeknél akár egy millió), ezért a raszterizáció elvárt sebessége több nagyságrenddel meghaladja az idáig tárgyalt műveletekét.

A vektorizációnak köszönhetően, akár csak a vágásnál, most is csak pontok, szakaszok és poligonok raszterizációjával kell foglalkoznunk.

A pont most is a könnyű eset. Nyilván azt a pixelt színezzük át, amelynek középpontja a ponthoz a legközelebb van. Ha a pont koordinátái (X, Y) , akkor a legközelebbi pixel a rasztertár $(\text{round}(X), \text{round}(Y))$ eleme.

7.6.1. Szakaszok rajzolása

Jelöljük a szakasz végpontjait $(x_1, y_1), (x_2, y_2)$ -vel. Ezen képernyő-koordináták a transzformációk, a vágás és az egész értékre kerekítés után állnak elő. Tegyük fel továbbá, hogy midőn az első végpontból a második felé haladunk, mindkét koordináta nő, és a gyorsabban változó irány az x , azaz

$$\Delta x = x_2 - x_1 \geq \Delta y = y_2 - y_1 \geq 0.$$

Ebben az esetben a szakasz enyhén emelkedő. A többi eset a végpontok és az x, y koordináták megfelelő felcserélésével analóg módon kezelhető.

A szakaszrajzoló algoritmusokkal szemben alapvető elvárás, hogy az átszínezett képpontok között ne legyenek lyukak, és a keletkezett kép ne legyen vastagabb a feltétlenül szükségesnél. Ez az enyhén emelkedő szakaszok esetén azt jelenti, hogy minden pixel oszlopban pontosan egy pixelt kell átszínezni, nyilván azt, amelynek középpontja a szakaszhoz a legközelebb van. Az egyenes egyenlete:

$$y = m \cdot x + b, \quad \text{ahol } m = \frac{y_2 - y_1}{x_2 - x_1}, \quad \text{és } b = y_1 - x_1 \cdot \frac{y_2 - y_1}{x_2 - x_1}, \quad (7.10)$$

alapján, az x koordinátájú oszlopban a legközelebbi pixel függőleges koordinátája:

$$Y = \text{round}(m \cdot x + y_1).$$

Ezzel el is jutottunk az első szakaszrajzoló algoritmusunkhoz:

```

DrawLine( $x_1, y_1, x_2, y_2$ )
   $m = (y_2 - y_1) / (x_2 - x_1)$ 
   $b = y_1 - x_1 \cdot (y_2 - y_1) / (x_2 - x_1)$ 
  for  $X = x_1$  to  $x_2$  do
     $y = m \cdot x + b$ 
     $Y = \text{round}(y)$ 
    Pixel( $X, Y, color$ )
  endfor
end

```

Ennek az algoritmusnak egyetlen szépséghibája a lassúsága, ami abból adódik, hogy minden pixel előállításához lebegőpontos szorzást, összeadást és lebegőpontos-egész átalakítást végez.

Asszimmetrikus DDA szakaszrajzoló algoritmus

A gyorsítás alapja a számítógépes grafika alapvető módszere, amelyet *inkrementális elv*nek nevezünk. Ez azon a felismerésen alapul, hogy általában könnyebben meghatározhatjuk az $y(X + 1)$ értéket az $y(X)$ felhasználásával, mint közvetlenül az X -ből. Egy szakasz esetén:

$$y(X + 1) = m \cdot (X + 1) + b = m \cdot X + b + m = y(X) + m,$$

ehhez egyetlen lebegőpontos összeadás szükséges (m törtszám). Az elv lényegében a Taylor-soros közelítésből származik, ezért az elv gyakorlati alkalmazását *DDA (Digitális Differenciális Analizátor)* algoritmusnak nevezik.

A DDA elvű szakaszrajzoló algoritmus:

```

DDADrawLine( $x_1, y_1, x_2, y_2$ )
   $m = (y_2 - y_1) / (x_2 - x_1)$ 
   $y = y_1$ 
  for  $X = x_1$  to  $x_2$  do
     $Y = \text{round}(y)$ 
    Pixel( $X, Y, \text{color}$ )
     $y = y + m$ 
  endfor
end

```

További gyorsítás érhető el *fixpontos számábrázolás*, segítségével. Ez azt jelenti, hogy a törtszám 2^T -szeresét tároljuk egy egész változóban, ahol T a törtbitek alkalmasan megválasztott száma. A törtbitek számát úgy kell megválasztani, hogy a leghosszabb ciklusban se halmozódhasson fel akkora hiba, hogy elrontsa a pixelkoordinátákat. Ha a leghosszabb szakasz hossza L , akkor az ehhez szükséges bitek száma $\log_2 L$.

A fixpontos, nem egész számok összeadásához az egész változókat kell összeadni, amely egyetlen gépi utasítást igényel, sőt néhány elemmel áramköri szinten realizálható. A kerekítés operáció pedig helyettesíthető egészrész képzéssel, ha előtte a számhoz 0.5-öt hozzáadunk. Az egészrész képzés a törtbitek levágását jelenti, így a teljes szakaszrajzolás hardverben könnyen implementálható. Hardver implementáción olyan szinkron digitális hálózatot értünk, amely minden egyes órajelre egy újabb pixel címét állítja elő a kimenetén (7.8. ábra).

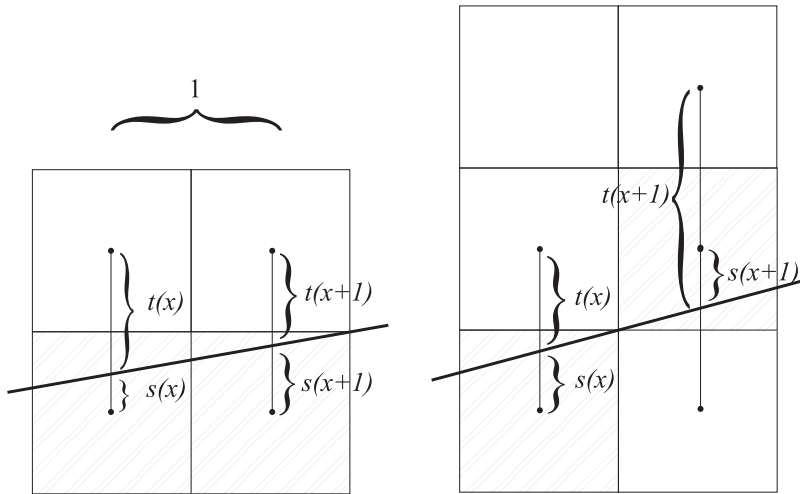
7.8. ábra. DDA szakaszrajzoló hardware

A DDA algoritmussal még mindig nem lehetünk teljes mértékben elégedettek. Egyrészt a szoftver implementáció során a fixpontos ábrázolás és egészrész képzés eltolási

(shift) műveleteket igényel. Másrészt, igaz szakaszonként csupán egyszer, az m meredekség kiszámításához osztani kell.

Mindkét problémával sikeresen birkózik meg a *Bresenham-algoritmus* [Bre65], amelyet a következő fejezet mutat be.

Bresenham szakaszrajzoló algoritmus



7.9. ábra. A Bresenham-algoritmus által használt jelölések

Jelöljük a szakasz és a legközelebbi pixel középpont függőleges, előjeles távolságát s -sel, a szakasz és a legközelebbi pixel feletti pixel függőleges távolságát t -vel (7.9. ábra). Ahogy a következő oszlopra lépünk, az s és t értékei változnak. Nyilván az eredetileg legközelebbi pixel sora és az egyvel feletti sor közül addig választjuk az alsó sort, amíg annak a távolsága tényleg kisebb mint a felette lévő pixel középpont és a szakasz távolsága, azaz ha $s < t$. Bevezetve az $e = s - t$ hibaváltozót, addig nem kell megváltoztatnunk az átfestendő pixel sorát, amíg $e < 0$. Az s, t, e változók számításához az inkrementális elvet használhatjuk ($\Delta x = x_2 - x_1, \Delta y = y_2 - y_1$):

$$s(X + 1) = s(X) + \frac{\Delta y}{\Delta x}, \quad t(X + 1) = t(X) - \frac{\Delta y}{\Delta x} \quad \Rightarrow$$

$$e(X + 1) = e(X) + 2 \frac{\Delta y}{\Delta x}.$$

Ezek az összefüggések akkor igazak, ha az $X + 1$. oszlopban ugyanazon sorokban lévő pixeleket tekintjük, mint a megelőzőben. Előfordulhat azonban, hogy az új oszlopban

már a felső pixel kerül közelebb a szakaszhoz (az e hibaváltozó pozitívvá válik), így az s, t, e mennyiségeket ezen pixel, és az ezen pixel feletti pixelre kell meghatározni. Erre az esetre a következő képletek vonatkoznak:

$$s(X+1) = s(X) + \frac{\Delta y}{\Delta x} - 1, \quad t(X+1) = t(X) - \frac{\Delta y}{\Delta x} + 1, \implies \\ e(X+1) = e(X) + 2\left(\frac{\Delta y}{\Delta x} - 1\right).$$

Figyeljük meg, hogy az s előjeles távolságot jelent, azaz az s negatív, ha a szakasz az alsó pixelközéppont alatt található. Feltételezhetjük, hogy az algoritmus indulásakor egy pixel középpontban vagyunk, tehát:

$$s(x_1) = 0, \quad t(x_1) = 1 \implies e(x_1) = s(x_1) - t(x_1) = -1.$$

Ezekkel a képletekkel önmagukban még nem nyertünk semmit. Az e hibaváltozó léptetéséhez nem egész összeadás szükséges, a növekmény meghatározása pedig osztást igényel. Vegyük észre azonban, hogy nekünk csak a hibaváltozó előjelére van szükségünk, hiszen akkor kell az Y változót eggyel léptetni, ha a hibaváltozó pozitívvá válik! Használjuk a hibaváltozó helyett, az $E = e \cdot \Delta x$ *döntési változót*! Enyhén emelkedő szakaszok esetén ($\Delta x > 0$) ez pontosan akkor vált előjelet, amikor a hibaváltozó. A döntési változóra érvényes képleteket a hibaváltozóra vonatkozó képletek Δx -szel történő szorzásával kapjuk meg:

$$E(X+1) = \begin{cases} E(X) + 2\Delta y, & \text{ha } Y\text{-t nem kell léptetni,} \\ E(X) + 2(\Delta y - \Delta x), & \text{ha } Y\text{-t léptetni kell.} \end{cases}$$

A döntési változó kezdeti értéke pedig $E = e(x_1) \cdot \Delta x = -\Delta x$.

A döntési változó egész kezdeti értékről indul és minden lépésben egész számmal változik, tehát az algoritmus egyáltalán nem használ törteket. Ráadásul a növekmények előállításához csupán egész összeadás (illetve kivonás), és 2-vel való szorzás szükséges.

A teljes *Bresenham-algoritmus*:

```
BresenhamLine( $x_1, y_1, x_2, y_2$ )
   $\Delta x = x_2 - x_1, \Delta y = y_2 - y_1$ 
   $dE^+ = 2(\Delta y - \Delta x), dE^- = 2\Delta y$ 
   $E = -\Delta x$ 
   $Y = y_1$ 
  for  $X = x_1$  to  $x_2$  do
    if  $E \leq 0$  then  $E += dE^-$ 
    else  $E += dE^+, Y++$ 
    Pixel( $X, Y, color$ )
  endfor
end
```

7.6.2. Területelárasztás

A területelárasztó algoritmusok (*flood-fill*) a rasztertár aktuális tartalma alapján működnek. Azon pixeleket színezik át, amelyek egy adott kezdeti pontból (*magból* vagy *forrásból*) a valamilyen módon definiált határ átlépése nélkül elérhetőek. Szükségünk van tehát egy feltételre, amely alapján egy pixelről megmondhatjuk, hogy az hozzátartozik-e az átszínezendő területhez, vagy határpont (például az átszínezendő pixelek színe “piros”, a határpontoké pedig nem).

A kitöltő algoritmusnak ezek után két feladata van. Egyrészt ellenőrzi, hogy az aktuális pixel belső pont-e, és ha igen, akkor kitölti. Másrészt, ha a pixel belső pixel, akkor a szomszéd pixelek szintén potenciális belső pixelek, így azokat ugyanúgy ellenőrizni kell és szükség estén ki kell tölteni. A szomszéd pixelek fogalmát kétféleképpen is lehet értelmezni. Egyrészt definiálhatunk két pixel szomszédként, ha közös élük van, másrészt mondhatjuk azt is, hogy két pixel akkor szomszédos, ha közös csúcsuk van. Az első értelmezés szerint egy pixelnek 4 szomszédja van, ezért az ezt alkalmazó algoritmust *4-szeresen összetett területkitöltő algoritmus*nak nevezzük. A második értelmezés szerint egy pixelnek 8 szomszédja van, így a realizáció *8-szorosan összetett területkitöltő eljárás*.

Egy 4-szeresen összetett területkitöltő algoritmus rendkívül egyszerű rekurzív szubrutinnal megvalósítható. A szubrutin a mag koordinátáit kapja meg bemeneti paraméterként.

```

FloodFill( $x, y$ )
  if pixel[ $x$ ][ $y$ ] belső pont then
    Pixel( $x, y, color$ )
    Flood( $x, y - 1$ )
    Flood( $x, y + 1$ )
    Flood( $x - 1, y$ )
    Flood( $x + 1, y$ )
  endif
end

```

A “belső pont” feltétel annak ellenőrzését jelenti, hogy a pixel hozzátartozik-e a kitöltendő területhez vagy sem. A 8-szorosan összetett területkitöltő szubrutin igen hasonló, csak a rekurziót még az

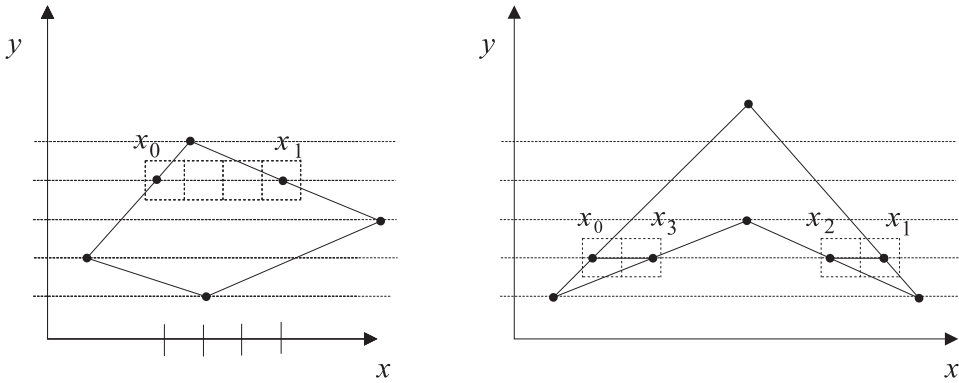
$$(x - 1, y - 1), (x + 1, y - 1), (x - 1, y + 1), (x + 1, y + 1)$$

pixelekre is folytatni kell.

7.6.3. Területkitöltés

A területkitöltő algoritmusok a határoló szakaszok geometriai definícióját kapják meg. A geometriai definíció szokásos formája a csúcsok egy m elemű tömbje (ez a tömb

általában a poligonvágó algoritmus kimenete).



7.10. ábra. Területkitöltés

A kitöltést célszerűen vízszintes pásztánként végezzük. Egyetlen pásztára az át-színezendő pixelek a következőképpen határozhatók meg. Kiszámítjuk a poligon éleinek metszéspontjait a vízszintes pásztával. A metszéspontokat az x koordináta alapján nagyság szerint rendezzük, majd átszínezzük a nulladik és az első pont közötti, a második és a harmadik pont közötti, általában a $2i$. és $2i + 1$. pont közötti pixeleket (7.10. ábra). Ez a módszer azokat a pontokat színezi ki, amelyeket ha végtelen távolból közelítünk meg, akkor páratlan számúszor kell átlépnünk a poligon határán.

Az első poligonkitöltő algoritmusunk tehát:

```

FillPolygonSlow( $q[m]$ )
  for  $Y = 0$  to  $Y_{\max}$  do
    scanline =  $Y$ 
     $k = 0$ 
    for  $e = 0$  to  $m - 1$  do
      if scanline a  $q[e]$  és  $q[e + 1]$  csúcsok között van then
         $x[k++] = (q[e], q[e + 1])$  szakasz és a scanline metszéspontja
      endif
    endfor
     $x[k]$  tömb rendezése
    for  $i = 0$  to  $k/2 - 1$  do
      for  $X = x[2i]$  to  $x[2i + 1]$  do Pixel( $X, Y, \text{Color}(X, Y)$ )
    endfor
  endfor
end

```

Az algoritmus a kitöltést az Y koordinátával definiált vízszintes pásztákra (scanline) végzi el. Egyetlen pásztára végigveszi az összes élt. Az e . él végpontjai a poligon

e . és $e + 1$. csúcsa. Ha a két végpont közrefogja a pásztát, akkor létezik metszéspont az él és a pászta között. A metszéspontok az x tömbben gyűlnek, a metszéspontok számát a k változó tárolja. Mivel a metszéspontok nem feltétlenül rendezettek az x koordináta szerint, a *Sort* függvény elvégzi a rendezést, majd minden második, és az azt követő metszéspont közé eső pixeleket kiszínezzük.

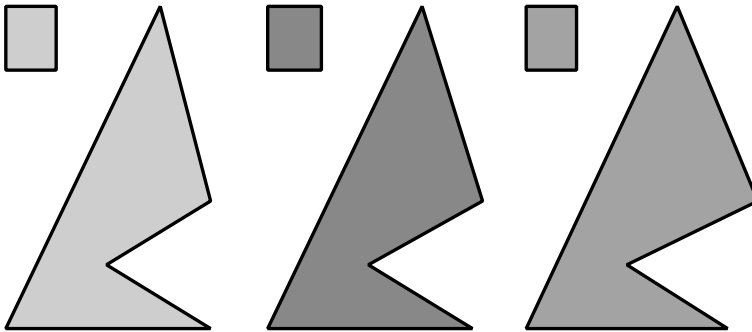
Az egyes pixelek színét meghatározó *Color* függvény lehet konstans, amennyiben egy adott színnel szeretnénk kitölteni. Mintával való kitöltés esetén azonban a szín függ a pixel koordinátáktól.

Egyszerű, 2D mintázatokat hozhatunk létre mintacsempék segítségével. Egy *mintacsempe* a színek 2 dimenziós tömbje, amely mindig ugyanabban a méretben és a koordináta-rendszer tengelyeivel párhuzamosan kerül a képernyőre. A csempézés folyamatát úgy képzelhetjük el, hogy egy burkoló adott referenciaponttól kezdve kicsempézi az egész képernyőt, és azon részeket, amelyek a poligonon kívülre esnének, kalapáccsal leveri. Ha a mintacsempe definíciója a $\text{pattern}[s_x][s_y]$ tömbben van, és a referenciapont koordinátái az (o_x, o_y) , akkor a *Color* függvény implementációja:

```

Color( $X, Y$ )
     $x = (X - o_x) \bmod s_x$ 
     $y = (Y - o_y) \bmod s_y$ 
    return pattern[ $x$ ][ $y$ ]
end

```



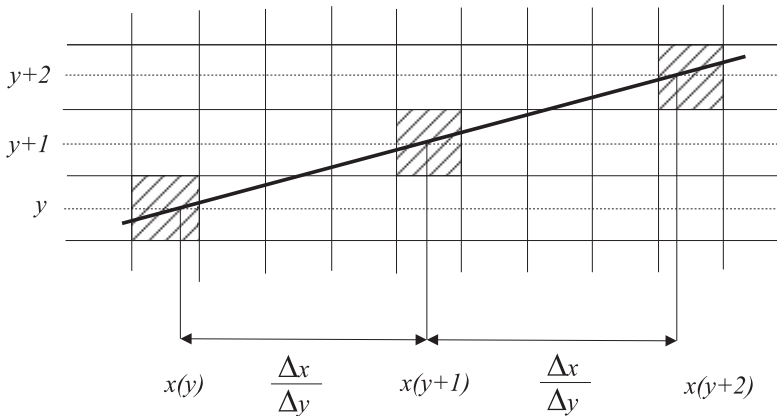
7.11. ábra. Kitöltés mintacsempékkel

Az első poligonkitöltő algoritmusunk túl lassú, ezért javításra szorul. A gyenge pontok és kiküszöbölésük módjai az alábbiak:

1. Az élék és a pászta között csak akkor keletkezhet metszéspont, ha a pászta y koordinátája az él minimális és maximális y koordinátája között van, ezért csak ezekre érdemes a metszéspontot kiszámítani. Az ilyen éléket *aktív él*eknek nevezzük. Az

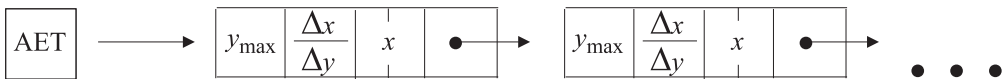
implementációhoz létre kell hoznunk az ún. *aktív él listát*, amely mindig csak az aktív éleket tartalmazza.

2. Két szakasz közötti metszéspontszámítás lebegőpontos szorzást, osztást és összeadást tartalmaz, ezért időigényes. Az *inkrementális elv* felhasználásával azonban a metszéspont meghatározható a megelőző pászta metszéspontjából egyetlen fixpontos, nem-egész összeadással (7.12. ábra).



7.12. ábra. A pászta és az élek közötti metszéspont inkrementális számítása

Az inkrementális elv használatakor figyelembe kell vennünk, hogy az x koordináta növekménye az egymást követő y egész értékekre $\Delta x / \Delta y$, ami nem egész szám, tehát az x érték tárolására fixpontos tört ábrázolást kell használnunk. Egy aktív él reprezentációja tehát tartalmazza a fixpontos ábrázolású $\Delta x / \Delta y$ növekményt, az ugyancsak fixpontos ábrázolású x metszéspontot, valamint a szakasz maximális függőleges koordinátáját (y_{\max}). Erre azért van szükségünk, hogy el tudjuk dönteni, hogy az Y pászták növelése során mikor fejezi be az él aktív pályafutását, azaz mikor kell eltávolítani az aktív él listából.



7.13. ábra. Aktív él lista szerkezete

A programunk működése tehát a következőképpen foglalható össze. A Y pásztákat egymás után generáljuk. Minden pásztára megnézzük, hogy mely élek válnak pont

ekkor aktívvá, azaz mely élek minimális y koordinátája egyezik meg a pászta koordinátájával. Ezeket az éleket betesszük az aktív él listába. Egyúttal az aktív él listát átvizsgáljuk, hogy vannak-e ott nyugdíjba vonuló élek is, amelyek maximális y koordinátája megegyezik a pászta koordinátájával. A nyugdíjba vonuló éleket kivesszük a listából (vegyük észre, hogy ebben a megoldásban az él alsó végpontját az él részének tekintjük, a felső pontját viszont nem). A kitöltés előtt gondoskodunk arról, hogy az aktív él listában az élek az x koordináta szerint rendezettek legyenek, majd minden második él közötti pixeleket átszínezzük. A kitöltés után az aktív él lista tagjaiban a metszéspontokat felkészítjük a következő pásztára, azaz minden él x tagjához hozzáadjuk az él $\Delta x/\Delta y$ növekményét. Majd kezdjük az egészet előlről a következő pásztára.

```

FillPolygonFast( $q[m]$ )
  for  $Y = 0$  to  $Y_{\max}$  do
    for  $e = 0$  to  $m - 1$  do
      edge = ( $q[e], q[e + 1]$ )
      if  $y_{\min}(\text{edge}) = Y$  then Put_AET( edge )
    endfor
    for minden edge élre az AET-ben do
      if  $y_{\max}(\text{edge}) \geq Y$  then Delete_AET( edge )
    endfor
    Resort_AET
    for minden második  $l$  élre az AET-ben do
      for  $X = \text{round}(x[l])$  to  $\text{round}(x[l + 1])$  do
        Pixel(  $X, Y, \text{Color}(X, Y)$  )
      endfor
    endfor
    for minden  $l$  élre az AET-ben do  $x[l] += \Delta x/\Delta y$ 
  endfor
end

```

A gyors algoritmus is vízszintes pásztánként dolgozik, egy pászta feldolgozását az aktívvá váló élek ($y_{\min}(\text{edge}) = Y$) aktív listába fűzésével kezdi. Az aktív él listát három művelet kezeli. A Put_AET(edge) művelet az él adatai alapján előállítja az aktív él lista egy elemének az adatait ($y_{\max}, \Delta x/\Delta y, x$), és a keletkező rekordot beteszi a listába. A Delete_AET művelet egy listaelemet töröl a listából. Erre akkor kerül sor, ha egy él éppen befejezi az aktív létet ($y_{\max}(\text{edge}) \geq Y$). A Resort_AET az x mező alapján átrendezi a listát. A rendezés után az algoritmus minden második él és a következő él közötti pixeleket kiszínezi, és végül az inkrementális képletek alkalmazásával az aktív él lista elemeit a következő pásztára lépteti.

Még ezen algoritmuson is tudunk gyorsítani. Vegyük észre ugyanis, hogy egyrészt felesleges a kitöltést a képernyő minden vízszintes pásztájára megkísérelni, elegendő lenne csak a csúcspontok minimális és maximális y koordinátái közötti intervallumot tekinteni. Ahhoz, hogy eldöntsük, hogy egy adott pásztánál melyik él válik aktívvá, ebben

a megoldásban mindig az összes élt ellenőriznünk kell. Érdeemes tehát a pászták indítása előtt egy listák tömbje kiegészítő adatstruktúrát felépíteni, amelyben az y . többelem azon éleket tartalmazza, amelyek éppen az y koordinátánál válnak aktívvá, majd a kitöltés során egy pásztánál csak a megfelelő többelemhez tartozó listát átmásolni az aktív élt listába.

7.7. Pixel műveletek

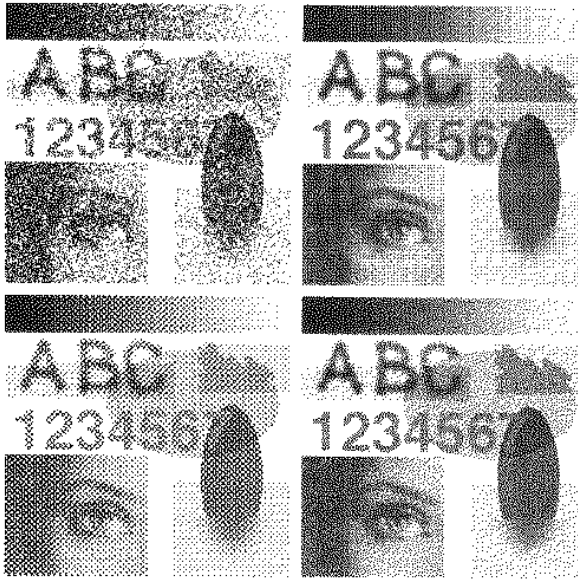
A raszterizációs fázis kimenete egy pixelsorozat, amelynek végső célja a rasztertár. A pixel műveletek az egyes pixelek színét még a beírás előtti utolsó pillanatban módosíthatják.

A módosítást elvégezhetjük a rasztertár adott helyén lévő pixel értékével úgy, hogy a rasztertárból kiolvasott és a raszterizáció eredményeként kapott színeket valamilyen aritmetikai vagy logikai műveletnek vetjük alá, és ennek az eredményét írjuk be a rasztertárba. Például súlyozott átlag képzésével az objektum átlátszóvá tehető. “Kizáró vagy” (\oplus) művelettel pedig ideiglenes rajzokat vihetünk be a rasztertárba, amelyeket aztán az alakzat újabb “kizáró vagy” típusú rajzolásával el is tüntethetünk onnan, hiszen fennáll az $A \oplus B \oplus B = A$ azonosság (ez a grafikus kurzor legegyszerűbb megvalósítása).

7.7.1. Dither alkalmazása

A rasztertár korlátozott kapacitása miatt az egy pixelhez tárolható bitek száma nem lehet túlságosan nagy (olcsóbb rendszerekben mindössze 4 vagy 8 bit). Az előállított színeket tehát újra kell kvantálni, hogy beférjenek a rasztertárba. Az újrakvantálás hatása különösen akkor zavaró, ha a megjelenített szín nem állandó, hanem lassan változó. Ekkor ugyanis a kevés rendelkezésre álló szín miatt a képernyőn a változás ugrásszerűen következik be, a folytonosan változó szín helyett pedig állandó színű csíkokat látunk (17.12. ábra).

A megoldást a *dither* jelenti, amely a végső kvantálás előtt egy 0 várható értékű nagyfrekvenciás zajt kever a megjelenítendő színekhez, majd a zajos jelet kvantálja. A zaj megmozgatja a kvantáló előtt a jelet. Az állandó színt a zaj és a kvantálás együttes hatása az egyik pixelben lefelé csonkítja, a szomszédjában pedig esetleg felfelé kerekíti, mégpedig azzal a valószínűséggel, amennyire közel vagyunk a kvantálási szintekhez. Messziről ránézve ezekre a zajos képekre, a szem átlagolja a nagyfrekvenciás zajt, és az átlag értéket, azaz a tényleges színt érzékeli. Különböző dither típusokat láthatunk a 17.13. és 7.14. ábrákon.



7.14. ábra. Véletlen és szabályos ditherek fekete-fehér képen

Egy periodikus ditherfüggvény elemei például egy mátrixban is megadhatók:

$$D^{(4)} = \frac{1}{16} \cdot \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix} \quad (7.11)$$

7.8. Interaktív 2D grafikus rendszerek

Az interaktív rendszerekben a felhasználó és a grafikus rendszer egy “folyamatos szabályozási körré” kapcsolódik össze (7.15. ábra). A felhasználó minden elemi parancsának hatását rögtön láthatja.

Eddig a képszintézis lépéseit tárgyaltuk, azaz azon műveleteket, amelyek a virtuális világ lefényképezésével előállítják képet. Ezt a műveletsorozatot *kimeneti csővezetéknek (output pipeline)* nevezzük. A *képszintézis (rendering)* során a kimeneti csővezetékét úgy működtetjük, hogy sorban elővesszük a virtuális világ objektumait, egy objektumra pedig annak primitívjeit, vektorizáljuk őket és a keletkező pontokat, szakaszokat és poligonokat (sőt karaktereket, bit-térképeket, stb.) végigvezetjük a kimeneti csővezetéken. Mivel a később rajzolt objektum elfedheti a korábban rajzoltakat, az ob-

7.15. ábra. 2D grafikus rendszerek felépítése

jektumok és primitívjeik *prioritását* a virtuális világban felvett sorrend határozza meg. A *takarási* sorrendet az objektumok és a primitívek sorrendjének megváltoztatásával módosíthatjuk.

Az interaktív grafikus rendszerekben a felhasználó a grafikus *beviteli eszközzel* (tablet, egér, stb.), az eszköz által mozgatott kurzor segítségével a képernyőn jelöl ki pontokat. A kurzor mozgatásához a beviteli eszköz saját *eszköz-koordinátarendszeréből* az adatokat át kell transzformálni a képernyő-koordinátarendszerbe. Például egy tablet az aktuális pozíciót általában 12 bites x és y koordinátákkal adja fel, amely alapján a kurzort az 1280×1024 -as felbontású képernyő megfelelő helyére kell vinni.

Az ily módon kijelölt pontokat a virtuális világ a lokális modellezési koordinátarendszerekben tárolja, így az inverz összetett transzformációval ide kell konvertálni a koordinátákat. Ezt a transzformációs láncot *bemeneti csővezetéknek* (*input pipeline*) nevezzük.

A felhasználó több különböző céllal vihet be pontokat. Egyrészt a pontot beépítheti a virtuális világ valamely objektumába, vagy egy már létrehozott objektumot kiválaszthat azzal, hogy rámutat a kurzor segítségével (*Pick*), végül egy kiválasztott pontot módosíthat az új pont segítségével.

A kiválasztási művelet

A *kiválasztási művelet* végrehajtása során végig kell nézni a virtuális világ összes elemét és az egyes elemekre meg kell vizsgálni, hogy a bevitt pont — legalábbis közelítőleg — rajta van-e az adott elemen. Előfordulhat, hogy a képszintézis több elemet is ugyanarra képpontra vetít, ekkor a képernyőn a legnagyobb prioritású, azaz a világban a leghátrébb lévő elem látható. A kiválasztásnak ilyen esetekben nyilván ezt a legnagyobb prioritású elemet kell azonosítani. Ezért a kiválasztás során a virtuális világot a képszintézis által használt prioritással ellentétes sorrendben dolgozzuk fel, és az első olyan elemet jelöljük ki, amelyen rajta van a bevitt pont.

Annak eldöntése, hogy egy pont rajta van-e egy általános elemen, nem tűnik egyszerű feladatnak. Mégis könnyen megbirkózhatunk vele, ha felismerjük, hogy a képszintézis éppen ezen feladat inverze, ugyanis a képszintézis megkeresi az összes olyan pixelt, amely az elemhez hozzátartozik. Ezért egy lehetséges megoldás az, ha beindítjuk a rajzolást — esetleg úgy, hogy a rasztertár tartalmának átírását letiltjuk — és figyeljük, hogy a kiválasztási pontnak megfelelő pixelt kellene-e rajzolni vagy sem. Ha ez bekövetkezik, a pont rajta van az elemen.

Az általános tapasztalat szerint remegő kézzel elég nehéz egy 1 pixel széles szakaszt eltalálni, ezért a kiválasztást tűréssel végezzük. Például a kiválasztási pont körül felvesszünk egy kis (például 10×10 pixeles) *tűrésű négyzetet* (*pick-window*), és azt figyeljük, hogy ezen négyzet belső pixeleit megváltoztatjuk-e.

A kiválasztási műveletet felgyorsíthatjuk azzal, hogy a raszterizáció lépését kihagy-

jük és a vágási téglalapnak a kiválasztási pont körüli tőrési négyzetet tekintjük. Ha a vágási művelet azt mondja, hogy az elemet rajzolni kellene, akkor biztosan megváltoztatnánk olyan pixeleket, amelyek a tőrési négyzet belsejében vannak.

7.9. Program: 2D grafikus rendszer

Az ismertető világmodellben törtvonalakat `PolyLine2D` és görbéket `Curve2D` tárolhatunk.

A Lagrange- és a Bézier-görbék megadásánál felhasználtuk a 6.6. fejezetben definiált osztályokat. Az ottani tagfüggvényeket kiegészítettük a képszintézishez szükséges vektorizációs művelettel, amely egy `Primitive2D` típusú objektumot egy csak szakaszból álló és a képszintézis műveletsorán végigvezethető `RenderPrimitive2D` típusú objektummá alakítja. A vektorizáció pontokat jelöl ki a görbén, amelyhez az `Interpolate` tagfüggvényt használjuk.

```
//=====
class Primitive2D {
//=====
    Array<Point2D> points;
    Color          color;
public:
    Primitive2D( Color& c, int n = 0 ) : color(c), points(n) { }
    Point2D& Point( int i ) { return points[i]; }
    int      PointNum( ) { return points.Size(); }
    virtual RenderPrimitive2D * Vectorize( ) { return NULL; }
};

//=====
class PolyLine2D : public Primitive2D {
//=====
public:
    PolyLine2D( Color& c ) : Primitive2D( c ) { }
    RenderPrimitive2D * Vectorize( ) {
        LineList2D * p = new LineList2D(Col(), 2 * PointNum() - 2);
        for( int i = 0; i < PointNum(); i++ ) {
            if ( i < PointNum() - 1 ) p -> Point(2*i)      = Point(i);
            if ( i > 0 )                p -> Point(2*i - 1) = Point(i);
        }
        return p;
    }
};
```



```
//=====
class Curve2D : public Primitive2D {
//=====
public:
    Curve2D( Color& c ) : Primitive2D( c ) { }
    virtual Point2D Interpolate( double tt ) = 0;
    RenderPrimitive2D * Vectorize( ) {
        LineList2D * p = new LineList2D(Col(), 2 * NVEC);
        for( int i = 0; i <= NVEC; i++ ) {
            Point2D pi = Interpolate( (double)i/NVEC );
            if ( i < NVEC) p -> Point( 2 * i ) = pi;
            if ( i > 0)    p -> Point( 2 * i - 1 ) = pi;
        }
        return p;
    }
};

//=====
class LagrangeCurve2D : public Curve2D, public LagrangePolinom {
//=====
public:
    LagrangeCurve2D( Color c ) : Curve2D( c ), LagrangePolinom( ) { }
    Point2D Interpolate( double tt ) {
        Point2D rr(0, 0);
        for(int i = 0; i < Degree(); i++) rr += Point(i) * L(i, tt);
        return rr;
    }
};

//=====
class BezierCurve2D : public Curve2D, public BezierPolinom {
//=====
public:
    BezierCurve2D( Color c ) : Curve2D( c ), BezierPolinom( ) { }
    Point2D Interpolate( double tt ) {
        double Bi = 1.0;
        Point2D rr(0, 0);
        for(int i = 0; i < PointNum(); i++)
            rr += Point(i) * B(i, tt, PointNum());
        return rr;
    }
};
```

A modellhierarchia magasabb szintjein a primitívekből objektumokat (Object2D) képezhetünk, amelyeket azután betehetjük a virtuális világba (VirtualWorld). Minden objektumhoz önálló modellezési transzformáció tartozik.

```
//=====
class Object2D {
//=====
    Array<Primitive2D *> prs;
    Transform2D          tr;
public:
    Object2D( ) { }
    void AddPrimitive( Primitive2D * p ) { prs[ prs.Size() ] = p; }
    Primitive2D * Primitive( int i ) { return prs[i]; }
    Transform2D& Transform( ) { return tr; }
    int PrimitiveNum() { return prs.Size(); }
};

//=====
class VirtualWorld {
//=====
    Array<Object2D *> objs;
public:
    VirtualWorld( ) : objs( 0 ) { }
    Object2D * Object( int o ) { return objs[o]; }
    void AddObject( Object2D * o ) { objs[ objs.Size() ] = o; }
    int ObjectNum() { return objs.Size(); }
};
```

A képszintézis előkészítéséhez definiáljuk a téglalap `RectAngle` osztályt:

```
//=====
class RectAngle {
//=====
    Coord left, right, bottom, top;
public:
    RectAngle(Coord l=0, Coord b=0, Coord r=1, Coord t=1)
    : left(l), right(r), bottom(b), top(t) { }
    double Left( ) { return left; }
    double Right( ) { return right; }
    double Bottom( ) { return bottom; }
    double Top( ) { return top; }
    double HSize( ) { return (right - left); }
    double VSize( ) { return (top - bottom); }
    double HCenter( ) { return ((right + left)/2); }
    double VCenter( ) { return ((top + bottom)/2); }
    Point2D Origin( ) { return Point2D(left, bottom); }
    int Code( Point2D& p ) {
        int c0 = left > p.X(),    c1 = right < p.X(),
        int c2 = bottom > p.Y(), c3 = top < p.Y();
        return (c0 | (c1 << 1) | (c2 << 2) | (c3 << 3));
    }
};
```

A vektorizáció során `RenderPrimitive2D` objektumok keletkeznek, amelyekre már értelmezhetők a képszintézishez szükséges transzformáció (`Transform`), vágás (`Clip`) és raszterizáció (`Draw`) műveletek. A transzformáció általánosan elvégezhető a definíciós pontok transzformálásával, a vágás és rajzolás azonban már attól függ, hogy milyen konkrét primitívtípust dolgozunk fel.

```
//=====
class RenderPrimitive2D : public Primitive2D {
//=====
public:
    RenderPrimitive2D( Color& c, int n = 0 ) : Primitive2D( c, n ) { }
    void Transform( Transform2D tr ) {
        for(int i = 0; i < PointNum(); i++) {
            HomPoint2D r = tr.Transform( (HomPoint2D)Point(i) );
            Point(i) = r.HomDiv();
        }
    }
    virtual BOOL Clip( RectAngle& cliprect ) = 0;
    virtual void Draw( Window * scr ) = 0;
};
```

A szakasz (`Line2D`) és a törtvonal (`LineList2D`) a `RenderPrimitive2D` konkrét változatai, amelyekben a vágás és rajzolás műveletek értelmet kapnak.

```
//=====
class Line2D : public RenderPrimitive2D {
//=====
public:
    Line2D( Point2D v1, Point2D v2, Color c )
        : RenderPrimitive2D( c, 2 ) { Point(0) = v1; Point(1) = v2; }
    BOOL    Clip( RectAngle& cliprect );
    void    Draw( Window * scr );
};

//=====
class LineList2D : public RenderPrimitive2D {
//=====
public:
    LineList2D( Color c, int n = 0 ) : RenderPrimitive2D( c, n ) { }
    BOOL    Clip( RectAngle& cliprect );
    void    Draw( Window * scr );
};
```

A szakasz vágásához a tárgyalt *Cohen-Sutherland vágási algoritmust* használhatjuk:

```
//-----
BOOL Line2D :: Clip( RectAngle& cliprect ) { // Cohen-Sutherland
//-----
    Point2D& p1 = Point(0);
    Point2D& p2 = Point(1);

    int c1 = cliprect.Code( p1 );
    int c2 = cliprect.Code( p2 );

    for( ; ; ) {
        if (c1 == 0 && c2 == 0) return TRUE;
        if ( (c1 & c2) != 0 ) return FALSE;

        int f;
        if ( (c1 & 1) != (c2 & 1) ) f = 1;
        else if ( (c1 & 2) != (c2 & 2) ) f = 2;
        else if ( (c1 & 4) != (c2 & 4) ) f = 4;
        else f = 8;

        double dy = p2.Y() - p1.Y(), dx = p2.X() - p1.X();

        double xi, yi;
        Point2D pi;

        switch ( f ) {
            case 1:
                yi = p1.Y() + dy * (cliprect.Left() - p1.X()) / dx;
                pi = Point2D( cliprect.Left(), yi );
                break;
            case 2:
                yi = p1.Y() + dy * (cliprect.Right() - p1.X()) / dx;
                pi = Point2D( cliprect.Right(), yi );
                break;
            case 4:
                xi = p1.X() + dx * (cliprect.Bottom() - p1.Y()) / dy;
                pi = Point2D( xi, cliprect.Bottom() );
                break;
            case 8:
                xi = p1.X() + dx * (cliprect.Top() - p1.Y()) / dy;
                pi = Point2D( xi, cliprect.Top() );
        }

        if (c1 & f) { p1 = pi; c1 = cliprect.Code( pi ); }
        else { p2 = pi; c2 = cliprect.Code( pi ); }
    }
}
```

A rajzolást a fizikai szinten a PLine függvény végzi el, amit a *Bresenham-algoritmus*nak megfelelően implementálhatunk:

```
extern Pixel( int x, int y, PColor color );
static int x1 = 0, y1 = 0;

#define XSTEPINIT int dep = 2*(dy - dx), dem = 2*dy, e = -dx, y = y1;
#define YSTEPINIT int dep = 2*(dx - dy), dem = 2*dx, e = -dy, x = x1;

#define XSTEP_YINCR {                                     \
    if ( e <= 0 ) { e += dem; } else { e += dep; y++; } \
    Pixel( x, y, color );                               \
}
#define XSTEP_YDEC {                                     \
    if ( e <= 0 ) { e += dem; } else { e += dep; y--; } \
    Pixel( x, y, color );                               \
}

#define YSTEP_XINCR {                                     \
    if ( e <= 0 ) { e += dem; } else { e += dep; x++; } \
    Pixel( x, y, color );                               \
}

#define YSTEP_XDEC {                                     \
    if ( e <= 0 ) { e += dem; } else { e += dep; x--; } \
    Pixel( x, y, color );                               \
}

//-----
void PLine( PCoord x2, PCoord y2 ) { // Bresenham
//-----
    int dx = x2 - x1, dy = y2 - y1;
    if (dx >= 0 && dy >= 0) {
        if (dx >= dy) {
            XSTEPINIT
            for( int x = x1; x <= x2; x++ ) XSTEP_YINCR
        } else {
            YSTEPINIT
            for( int y = y1; y <= y2; y++ ) YSTEP_XINCR
        }
    } else if (dx < 0 && dy >= 0) {
        dx = -dx;
        if (dx >= dy) {
            XSTEPINIT
            for( int x = x1; x >= x2; x-- ) XSTEP_YINCR
        } else {
            YSTEPINIT
            for( int y = y1; y <= y2; y++ ) YSTEP_XDEC
        }
    }
}
```

```

} else if (dx >= 0 && dy < 0) {
    dy = -dy;
    if (dx >= dy) {
        XSTEPINIT
        for( int x = x1; x <= x2; x++ ) XSTEP_YDEC
    } else {
        YSTEPINIT
        for( int y = y1; y >= y2; y-- ) YSTEP_XINCR
    }
} else {
    dx = -dx; dy = -dy;
    if (dx >= dy) {
        XSTEPINIT
        for( int x = x1; x >= x2; x-- ) XSTEP_YDEC
    } else {
        YSTEPINIT
        for( int y = y1; y >= y2; y-- ) YSTEP_XDEC
    }
}
}
}

```

A 2D képszintézis kamerája két téglalpból áll. Az egyik téglalap a virtuális világ megjelenítendő tartományát jelöli ki (*window*), a másik pedig a képernyő azon tartományát, ahová a képet el kell helyezni (*viewport*). A vágási tartomány (*cliprect*) általában megegyezik a nézettel (*viewport*), de az alábbi modell megenged a nézettől eltérő vágási téglalapot is. A *CalcTransf* tagfüggvény az ablak és a nézet paramétereiből meghatározza a nézeti transzformációs mátrixot (*transf*).

```

//=====
class Camera2D {
//=====
    RectAngle window, cliprect, viewport;
    Transform2D transf;
    void CalcTransf( ) {
        Transform2D wintrans(TRANSLATION, - window.Origin() );
        Transform2D winviep(SCALE, viewport.HSize()/window.HSize(),
                             viewport.VSize()/window.VSize() );
        Transform2D viewtrans(TRANSLATION, viewport.Origin() );
        transf = wintrans * winviep * viewtrans;
    }
public:
    Camera2D( )
    : window(0, 0, 1, 1), viewport(0, 0, 1, 1), cliprect(0, 0, 1, 1) {
        CalcTransf();
    }
    void SetWindow( RectAngle w ) { window = w; CalcTransf(); }
    void SetViewport( RectAngle v ) { viewport = v; CalcTransf(); }
}

```

```

void SetClipWindow( RectAngle v ) { cliprect = v; }
RectAngle Window( ) { return window; }
RectAngle Viewport( ) { return viewport; }
RectAngle ClipWindow( ) { return cliprect; }
Transform2D ViewTransform( ) { return transf; }
};

```

A képszintézishez szükséges összes információt a színtérben (Scene) foglaljuk össze. A színtér a virtuális világmodellen (world) kívül még tartalmazza a kameraparamétereket (camera), a megjelenítőobjektum azonosítóját (scr), és az interaktív felépítés során aktuális objektum (actobj) és primitív (actprim) sorszámát. A színtér Render művelete előállítja a virtuális világ képét a megadott kameraállásból, a Pick művelet pedig megkeresi, hogy egy pontban melyik objektum látszik a képernyőn.

```

//=====
class Scene {
//=====
    Window *      scr;
    VirtualWorld  world;
    Camera2D      camera;
    int           actobj, actprim;

    Point2D InputPipeline( Coord x, Coord y );
public:
    Scene( Window * ps ) { scr = ps; }
    void Render( );
    void Pick( Coord x, Coord y );
};

```

Az InputPipeline a bemeneti csővezetéken vezeti végig a felhasználó által megadott pontot és nézeti transzformáció, valamint az aktuális objektum modellezési transzformációja alapján előállítja a pont lokális modellezési koordináta-rendszerbeli képét.

```

//-----
Point2D Scene :: InputPipeline( Coord x, Coord y ) {
//-----
    Object2D * obj = world.Object( actobj );
    Transform2D Tm = obj -> Transform();
    Transform2D Tv = camera.ViewTransform();
    Transform2D Tci = Tm * Tv;
    Tci.InvertAffine( );
    return Tci.Transform( HomPoint2D(x, y, 1) );
}

```

A Render tagfüggvény elvégzi a képszintézist, amely a következő lépésekből áll: képernyő törlése, az egyes objektumoknak megfelelő modellezési illetve összetett transzformációk számítása, az objektumok primitívjeinek vektorizálása és a vektorizált primitívek transzformálása, vágása és végül raszterizálása.

```
//-----
void Scene :: Render( ) {
//-----
    scr -> Clear( );
    for(int o = 0; o < world.ObjectNum(); o++) {
        Object2D * obj = world.Object( o );
        Transform2D Tv = camera.ViewTransform();
        Transform2D Tm = obj -> Transform();
        Transform2D Tc = Tm * Tv;

        for(int p = 0; p < obj -> PrimitiveNum( ); p++) {
            RenderPrimitive2D * rp = obj->Primitive(p) -> Vectorize();
            rp -> Transform( Tc );
            if ( rp -> Clip( camera.ClipWindow() ) ) rp -> Draw( scr );
            delete rp;
        }
    }
}
}
```

A `Pick` tagfüggvény megkeresi, hogy a felhasználó melyik objektumra mutatott rá, és visszaadja annak sorszámát, vagy `-1`-t, ha a megadott pont környezetében nincs objektum. Az objektumok vizsgálatát a prioritásnak megfelelően a rajzolással ellentétes sorrendben végezzük el. Egyetlen objektum ellenőrzése a kijelölt pont környezetét jelentő ablakra (`pickwindow`) történő vágási algoritmussal történik.

```
//-----
int Scene :: Pick( Coord x, Coord y ) {
//-----
    RectAngle pickwindow( x - PICKRECT_X, y - PICKRECT_Y,
                          x + PICKRECT_X, y + PICKRECT_Y );

    for(int actobj = world.ObjectNum() - 1; actobj >=0; actobj--) {
        Object2D * obj = world.Object( actobj );
        Transform2D Tm = obj -> Transform();
        Transform2D Tv = camera.ViewTransform();
        Transform2D Tc = Tm * Tv;

        for(int p = obj -> PrimitiveNum( ) -1; p >= 0; p--) {
            RenderPrimitive2D * rp = obj -> Primitive(p) -> Vectorize();
            rp -> Transform( Tc );
            if ( rp -> Clip(pickwindow) ) { delete rp; return actobj; }
            delete rp;
        }
    }
    return -1;
}
}
```


8. fejezet

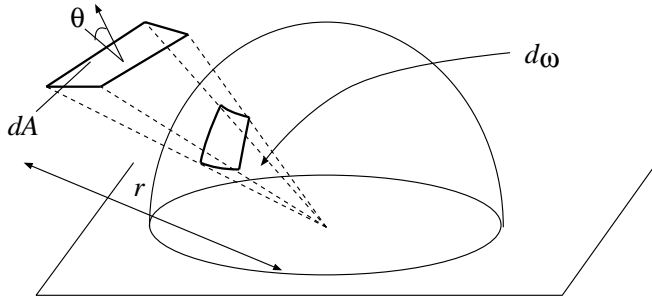
Az árnyalás optikai alapmodellje

A 3D képszintézis célja, hogy a fényforrások, a felületek geometriája, a felületek optikai jellemzői és a kamerák tulajdonságai alapján meghatározza, hogy az egyes kamerák milyen "színt", azaz milyen spektrumú fényt érzékelnek. A kérdéskör bevezetése során meg kell ismerkednünk a fényerősség alapvető mértékeivel, a kamerák és a fényvisszaverődés fizikai modelljeivel. Ez a fejezet összefoglalja azon optikai törvényeket, amelyeket a 3D számítógépes grafika használ. Az intuitív magyarázatok mellett a teljesség kedvéért ismertetjük a levezetéseket is, ám bár ezek tényleges megértése nélkül is alkalmazhatjuk a kiadódó képleteket a grafikus algoritmusainkban.

8.1. A fényerősség alapvető mértékei

Ebben a fejezetben a fényátadás alapvető mérőszámait és számítási eljárásait tekintjük át. A vizsgálatunkat λ hullámhosszú monokromatikus fényre végezzük el, mivel a teljes spektrumban történő analízis több ilyen elemzésre vezethető vissza. A bemutatandó anyagjellemzők nyilván függhetnek a megadott hullámhossztól.

Egy felület különböző irányokban sugározhat, ezért szükséges a térbeli irányok formalizálása. Emlékezzünk arra, hogy a síkban az irányokat szögekkel jellemezhetjük. Egy szög egy egységkör egy ívével adható meg, értéke pedig ezen ív hossza. A szög azon irányokat foglalja magában, amelyek a szög csúcsából az ív valamely pontjába mutatnak. A normál szög fogalmának általánosításával jutunk el az illuminációs gömb és a térszög fogalmához. A térbeli irányokat a 2D egységkör mintájára ún. *illuminációs gömb* segítségével definiálhatjuk egyértelműen. Ez az egység sugarú gömb azon térszögeket tartalmazza, ahová a középpontban lévő forrás sugározhat. A *térszög* (*solid angle*) az egységgömb felületének egy része, amelyet ezen felület méretével számszerűsítünk. Egy térszög azon irányokat tartalmazza, amelyek a gömb középpontjából a felületrész valamely pontjába mutatnak. A térszög mértékegysége a szteradián [*sr*].



8.1. ábra. A térszög definíciója

Egy dA felületelem egy \vec{p} pontból

$$d\omega = \frac{dA \cdot \cos \theta}{r^2} \quad (8.1)$$

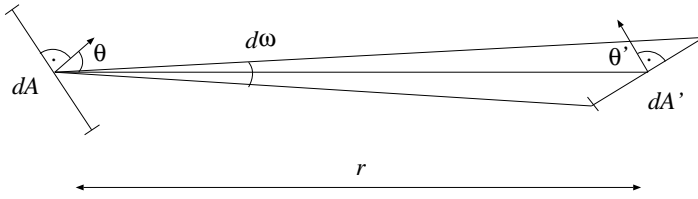
térszög alatt látszik, ahol r a \vec{p} pont és dA felületelem távolsága, θ pedig a dA felületi normálisa és a \vec{p} iránya közötti szög (8.1. ábra).

Az átadott fény erősségét több különböző mértékkel jellemezhetjük. A *fluxus* (Φ) egységnyi idő alatt, adott hullámhossz tartományban, egy hipotetikus határfelületen átadott energia. A fluxus mértékegysége a watt [W]. 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 a radianciát használjuk. A *radiancia*, vagy *intenzitás* (L), egy dA felületelemet $d\omega$ térszögben elhagyó $d\Phi$ infinitezimális fluxus osztva a kilépési irányból látható területtel ($dA \cdot \cos \theta$) és a térszöggel:

$$L = \frac{d\Phi}{dA \cdot d\omega \cdot \cos \theta}. \quad (8.2)$$

A radiancia mértékegysége: [$W \cdot m^{-2} \cdot sr^{-1}$]. Figyeljük meg, hogy a radiancia valóban csak az adott irányú sugárzás erősségét minősíti. Ha kétszer akkora térszögben mérjük, a fluxus ugyan kétszer akkora lesz, de a térszöggel történt osztás után változatlan eredményhez jutunk. Hasonlóan, ha a sugárzó kétszer akkora területét vizsgáljuk, akkor a fluxus megint közel kétszer akkora lesz, viszont a területtel osztva megint csak függetleníthetjük a radianciát a sugárzó területétől.

Miután megismerkedtünk az alapvető mennyiségekkel, nézzük meg, hogy miként határozhatók meg egy olyan elrendezésben, ahol egy dA felületelem kibocsátott fényteltjesítménye egy másik dA' felületelemre jut (8.2. ábra). Ha a felületelemek látják



8.2. ábra. Két infinitezimális felületelem között átadott fluxus

egymást, és a dA intenzitása a dA' irányába L , akkor 8.2 egyenlet szerint az átadott fluxus:

$$d\Phi = L \cdot dA \cdot d\omega \cdot \cos \theta. \quad (8.3)$$

Az 8.1 definíció felhasználásával a térszöget kifejezhetjük a dA' látható területével. Ezzel egy alapvető egyenlethez jutunk, amely a *fotometria alaptörvénye*:

$$d\Phi = L \cdot \frac{dA \cdot \cos \theta \cdot dA' \cdot \cos \theta'}{r^2}. \quad (8.4)$$

Ezen egyenlet szerint az átadott fluxus egyenesen arányos a forrás radianciájával, forrás és az antenna látható területével és fordítottn arányos a távolságukkal.

Vegyük észre, hogy 8.1 definíció alkalmazásával az átadott teljesítmény a következő alakban is felírható:

$$d\Phi = L \cdot dA' \cdot \frac{dA \cdot \cos \theta}{r^2} \cdot \cos \theta' = L \cdot dA' \cdot d\omega' \cdot \cos \theta', \quad (8.5)$$

amely szerint ugyanolyan képlet vonatkozik a sugárzó felületelemre (8.2 egyenlet), mint a sugárzást felfogó antennára.

8.2. A kamerák jellemzése

Egy *kamera* elemi kamerák, vagy mérőeszközök gyűjteményeként fogható fel, ahol minden elemi kamera egyetlen skalár mennyiséget mér. Egy elemi kamera általában egy pixelen átjutó fényt detektál, de mérheti a felületelemet adott térszögben elhagyó fénytjeljesítményt is.

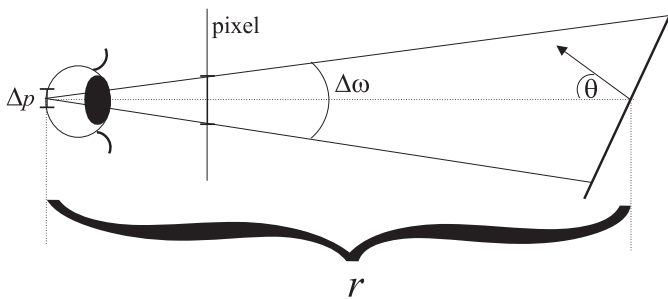
Rendeljünk minden elemi kamerához egy $W^e(\vec{y}, \omega)$ *érzékenység függvényt*, amely megmutatja, hogy az \vec{y} pontból az ω irányba kibocsátott egységnyi energiájú foton mekkora hatást kelt a műszerünkben. Ha az elemi kamera a pixelen átjutó teljesítményt méri, akkor nyilván az érzékenység függvény 1 értékű azon pontokra és irányokra, amely pontokat 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. Felhasználva a fluxus és a radiancia közötti összefüggést, egy adott hullámhosszon detektált fényteljesítmény

$$\int_S \int_{\Omega} d\Phi(\vec{y}, \omega) \cdot W^e(\vec{y}, \omega) = \int_S \int_{\Omega} L(\vec{y}, \omega) \cdot \cos \theta \cdot W^e(\vec{y}, \omega) d\omega d\vec{y} = \mathcal{M}L, \quad (8.6)$$

ahol \mathcal{M} a radianciamérő operátor. A képletet a következőképpen értelmezhetjük. Ahhoz, hogy például egy pixelen keresztül a szembe jutó teljesítményt meghatározzuk, számba kell venni a szemből a pixelen keresztül látható felületi pontok szemirányú radianciáját ($L(\vec{y}, \omega)$). A szemből látható pontokat és az innen a szembe mutató irányokat az érzékenység függvény jelöli ki ($W^e(\vec{y}, \omega)$). A $\cos \theta$ azért van a képletben, hogy képviselje azt a hatást, hogy lapos szögben a felületre nézve nagy területeket is igen kicsinek láthatunk.

Összefoglalva, az egyes pixelek színének a meghatározásához a felületi radiancia ismerete szükséges.



8.3. ábra. Egy egyszerű kameramodell

Egy konkrét kameramodell létrehozásához tekintsük a pixelen keresztül a szemretina Δp területére egységnyi idő alatt eső energiát (8.3. ábra). Jelöljük a pixelen keresztülmenő irányokat összefogó térszöget $\Delta\omega$ -val. A pixelen keresztül látható terület nagysága $r^2 \cdot \Delta\omega / \cos \theta$, ahol r a látható terület távolsága, θ pedig a látható terület normálvektorának és a szem irányának a szöge. Ezen felület pontjaiból azon irányokba sugárzott fény jut a retinára, amelyek a $\Delta p / r^2$ nagyságú térszögön belül vannak. Amennyiben a retinára a pixelen keresztül érkező fényt mérjük, a $W^e(\vec{y}, \omega)$ érzékenység függvény 1 a pixelen keresztül látható felületi pontokra és azon irányokra, amelyek ezen pontokból a retina felé mutatnak.

Így a mért fényteljesítmény:

$$\Phi_p = \int_S \int_{\Omega} L(\vec{y}, \omega) \cos \theta \cdot W^e(\vec{y}, \omega) d\omega d\vec{y} \approx$$

$$L(\vec{y}, \omega) \cdot \cos \theta \cdot \frac{r^2 \cdot \Delta\omega}{\cos \theta} \cdot \frac{\Delta p}{r^2} = L(\vec{y}, \omega) \cdot \Delta p \cdot \Delta\omega, \quad (8.7)$$

azaz egy arányossági tényezőn kívül csak a látható pont radianciájától függ. A látható terület távolsága kiesett a képletből, ami megfelel annak a tapasztalatnak, hogy egy objektumra (például a falra) ránézve ugyanolyan fényesnek érezzük akkor is, ha közelebb megyünk hozzá, vagy ha eltávolodunk tőle.

8.3. A fény-felület kölcsönhatás: az árnyalási egyenlet

A fény-felület kölcsönhatás során egy fénysugár által megvilágított felület a beérkező fényteljesítmény egy részét különböző irányokban visszaveri, míg másik részét elnyeli.

Az optikailag tökéletesen sima felületekre a visszaverődést a *visszaverődési törvény*, a fénytörést a *Snellius-Descartes-törvény* írja le. A felületi egyenletlenségek miatt azonban a valódi felületek bármely irányba visszaverhetik, illetve törhetik a fényt. Ezeket a hatásokat a valószínűségszámítás eszközeivel modellezhetjük.

Tegyük fel, hogy az ω' irányból egy foton érkezik a felület \vec{x} pontjába. A foton ω irányú továbbhaladását a következő feltételes valószínűség-sűrűségfüggvénnyel jellemezzük:

$$r(\omega', \vec{x}, \omega) \cdot d\omega = \Pr\{\text{a foton az } \omega \text{ irány körüli } d\omega \text{ térszögben megy} \mid \omega' \text{ irányból jön}\}. \quad (8.8)$$

Ez a valószínűség-sűrűségfüggvény az anyag optikai tulajdonságait írja le. 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.

Most térjünk rá annak vizsgálatára, hogy a felület egy adott irányból milyen fényesnek látszik. Egy ω irány körüli $d\omega$ térszögbe visszavert vagy tört fluxust megkaphatjuk, ha tekintjük az összes lehetséges ω' bejövő irányt, és az ezekből érkező fluxusok hatását összegezzük:

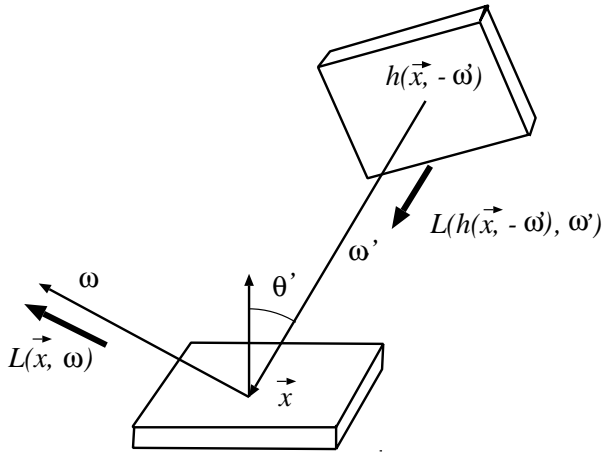
$$\int_{\Omega} (r(\omega', \vec{x}, \omega) d\omega) \cdot \Phi^{\text{in}}(\vec{x}, \omega', d\omega') \quad (8.9)$$

Az összegzés során a bejövő energiát (fotonokat) aszerint súlyozzuk, hogy mi a valószínűsége, hogy a bejövő foton éppen a nézeti irányba verődik vissza.

Amennyiben a felület maga is fényforrás, a

$$\Phi^e(\vec{x}, \omega) = L^e(\vec{x}, \omega) \cdot dA \cdot \cos \theta \cdot d\omega \quad (8.10)$$

kisugárzott fény mennyiség ugyanancsak hozzájárul a kimeneti fluxushoz.



8.4. ábra. Az árnyalási egyenlet geometriája

A lehetséges hatásokat összegezve:

$$\Phi^{\text{out}}(\vec{x}, \omega) = \Phi^e(\vec{x}, \omega) + \int_{\Omega} (r(\omega', \vec{x}, \omega) d\omega) \cdot \Phi^{\text{in}}(\vec{x}, \omega', d\omega'). \quad (8.11)$$

A fluxus és a radiancia közötti 8.2 összefüggést felhasználva:

$$\Phi^{\text{in}}(\vec{x}, \omega', d\omega') = L^{\text{in}}(\vec{x}, \omega') \cdot dA \cdot \cos \theta' \cdot d\omega', \quad \Phi^{\text{out}}(\vec{x}, \omega, d\omega) = L(\vec{x}, \omega) \cdot dA \cdot \cos \theta \cdot d\omega. \quad (8.12)$$

Behelyettesítve ezeket a 8.11 egyenletbe, és mindkét oldalt elosztva $dA \cdot d\omega \cdot \cos \theta$ -val:

$$L(\vec{x}, \omega) = L^e(\vec{x}, \omega) + \int_{\Omega} L^{\text{in}}(\vec{x}, \omega') \cdot \cos \theta' \cdot \frac{r(\omega', \vec{x}, \omega)}{\cos \theta} d\omega'. \quad (8.13)$$

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)*:

$$f_r(\omega', \vec{x}, \omega) = \frac{r(\omega', \vec{x}, \omega)}{\cos \theta}. \quad (8.14)$$

A BRDF mértékegysége 1 per szteradián [sr^{-1}].

Visszatérve a 8.13 egyenlethez, az $L^{\text{in}}(\vec{x}, \omega')$ bejövő radiancia egyenlő az \vec{x} pontból a $-\omega'$ irányba látható \vec{y} pont ω' irányú radianciájával. Vezessük be a

$$\vec{y} = h(\vec{x}, \omega').$$

*láthatóság függvény*nt, amely megmondja, hogy egy pontból egy adott irányba milyen másik felületi pont látszik. Ezzel végre eljutottunk a fényátadás alapvető integrálegyenletéhez, az *árnyalási egyenlethez* (*rendering equation*) [Kaj85]:

$$L(\vec{x}, \omega) = L^e(\vec{x}, \omega) + \int_{\Omega} L(h(\vec{x}, -\omega'), \omega') \cdot f_r(\omega', \vec{x}, \omega) \cdot \cos \theta' d\omega'. \quad (8.15)$$

Az árnyalási egyenlet, bár bonyolultnak látszik, valójában rendkívül egyszerűen értelmezhető. Egy felületi pont adott irányú radianciája ($L(\vec{x}, \omega)$) megegyezik a felületi pont ilyen irányú saját emissziójának ($L^e(\vec{x}, \omega)$) és a különböző irányból ide jutó radiancia ($L(h(\vec{x}, -\omega'), \omega')$) az adott irányba történő visszaverődésének az összegével. A visszaverődést a $f_r(\omega', \vec{x}, \omega) \cdot \cos \theta' d\omega'$ tag jellemzi, amely lényegében annak az 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 $(L^e, f_r(\omega', \vec{x}, \omega))$ paraméterek a különböző hullámhosszokon eltérőek lehetnek.

Bevezetve a fény-felület kölcsönhatást leíró \mathcal{T} integráloperátort

$$(\mathcal{T}L)(\vec{x}, \omega) = \int_{\Omega} L(h(\vec{x}, -\omega'), \omega') \cdot f_r(\omega', \vec{x}, \omega) \cdot \cos \theta' d\omega', \quad (8.16)$$

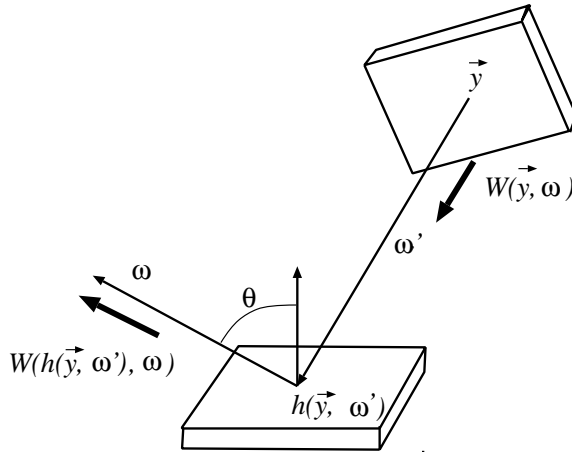
felállíthatjuk az árnyalási egyenlet rövid alakját:

$$L = L^e + \mathcal{T}L. \quad (8.17)$$

8.4. Az árnyalási egyenlet adjungáltja: a potenciál egyenlet

A radiancia nem az egyetlen mérték, amelyet a számítógépes grafika használ. A fényátadás vizsgálatával más mértékekre nyilván más egyenleteket állíthatunk fel. Ebben a fejezetben, a részletes levezetést mellőzve, a potenciált meghatározó egyenletet ismer-tetjük.

Az idáig tárgyalt radiancia a felületek sugárzási intenzitását fejezi ki, amely részint a saját emisszióból, részint a többi felület sugárzásának visszaverődéséből áll össze. Ha a fényátadási jelenséget egy sugárzó és egy detektor kölcsönhatásaként képzeljük el, akkor a radiancia a jelenséget a sugárzó szempontjából írja le. A *potenciál* a radianciához hasonlóan alapvető mérték, amely azonban ugyanazt a jelenséget a detektor szemszögéből tárgyalja. A potenciál definíciója a következő: tekintsünk egy elemi kamerát, és tegyük fel, hogy az \vec{y} felületi pont az ω' irányba egy egységnyi energiájú fotonnyalábot bocsát ki. A fotonok egy része a visszaverődések után az elemi kamerába jut. A kamerába jutó energiát az \vec{y} pont ω' irányú potenciáljának nevezzük.



8.5. ábra. A potenciál egyenlet által használt jelölések

A fény-felület kölcsönhatás a radiancia átadás mellett a potenciál átadásával is leírható. Ha a visszaverődésektől eltekintünk, akkor nyilván $W(\vec{y}, \omega') = W^e(\vec{y}, \omega')$.

Ha a visszaverődéseket is követjük, a potenciálra az árnyalási egyenlet adjungált egyenlete, a *potenciál egyenlet* [PM95] írható fel.

$$W = W^e + \mathcal{T}'W. \quad (8.18)$$

\mathcal{T}' a potenciálátadást leíró integráloperátor

$$(\mathcal{T}'W)(\vec{y}, \omega') = \int_{\Omega} W(h(\vec{y}, \omega'), \omega) \cdot f_r(\omega', h(\vec{y}, \omega'), \omega) \cdot \cos \theta \, d\omega, \quad (8.19)$$

ahol θ a felületi normális és az ω kimeneti irány közötti szög. Ez az egyenlet azt mondja, hogy egy felületi pontból adott irányba kibocsátott foton műszerre gyakorolt hatását felírhatjuk a közvetlen hatás és a visszaverődések utáni indirekt hatás összegeként.

Ezen integrálegyenlet megoldásával ugyancsak meghatározhatjuk az elemi kamerába jutó fényteljesítményt:

$$\int_S \int_{\Omega} L^e(\vec{y}, \omega) \cos \theta \cdot W(\vec{y}, \omega) \, d\omega \, d\vec{y}. \quad (8.20)$$

8.5. Az árnyalási illetve a potenciál egyenlet megoldása

Matematikai szempontból az árnyalási (és a potenciál) egyenlet egy másodfajú *Fredholm-féle integrálegyenlet*, amelyben az ismeretlen radianciafüggvényt kell meghatározni. Ez a radianciafüggvény egyrészt önállóan, másrészt az integrálon belül jelenik

meg. Azt is mondhatjuk, hogy az egyenletben az integrál és az azon kívüli részek között csatolás van, mert mindkettő függ az ismeretlen radianciától. Intuitíven megközelítve ezt a kérdést, 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 radianciáját.

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 technikákat három nagy kategóriába sorolhatjuk, amelyek a gyorsaság-valósághűség ellentmondó követelménypárt különböző kompromisszummal elégítik ki (17.15. ábra).

A *lokális illuminációs algoritmusok* az árnyalási egyenlet drasztikus egyszerűsítésével kiküszöbölnék mindenféle csatolást, azaz egy felület fényességének meghatározásához nem veszik figyelembe a többi felület fényességét. Megvilágítás csak a képen közvetlenül nem látható absztrakt fényforrásokból érkezik. A csatolás megszüntetésével az árnyalási egyenletben az integrálból eltűnik az ismeretlen függvény, így az integrálegyenlet megoldása helyett csupán egy egyszerű integrált kell kiértékelnünk.

A *sugárkövetés illuminációs algoritmus* a csatolást csak véges számú ideális visszaverődésre és törésre követi.

A *globális illuminációs algoritmusok* az integrálegyenletet a csatolással együtt próbálják megoldani, vállalva az ezzel járó munkamennyiséget is.

8.6. BRDF modellek

Valósághű képek előállításánál olyan BRDF modelleket kell használnunk, amelyek nem sértik az alapvető fizikai törvényeket, mint például a BRDF-k szimmetriáját kimondó *Helmholtz-törvényt*, vagy az *energiamegmaradás* törvényét.

A Helmholtz-féle szimmetria, vagy *reciprocitás* [Min41] szerint a fénysugár megfordítható, azaz a BRDF-ben a bejövő és kimenő irányok felcserélhetőek:

$$f_r(\omega, \vec{x}, \omega') = f_r(\omega', \vec{x}, \omega). \quad (8.21)$$

Ez a tulajdonság az, amely miatt a valószínűség-sűrűségfüggvényekkel szemben a BRDF-eket részesítjük előnyben az optikai anyagmodellek megadásánál. A szimmetria miatt

$$f_r(\omega', \vec{x}, \omega) \cdot \cos \theta' = f_r(\omega, \vec{x}, \omega') \cdot \cos \theta = \frac{r(\omega, \vec{x}, \omega')}{\cos \theta'} \cdot \cos \theta' = r(\omega, \vec{x}, \omega'). \quad (8.22)$$

Az energiamegmaradás elve értelmében, egy önállóan nem sugárzó felületem nem adhat ki több fotont (nagyobb fluxust), mint amit maga kapott, vagy másképpen, a tesztoleges irányú visszaverődés teljes valószínűsége nyilván nem lehet egynél nagyobb.

A tetszőleges irányú visszaverődést *albedo*nak nevezzük. Az albedo definíciója:

$$a(\vec{x}, \omega') = \int_{\Omega} f_r(\omega', \vec{x}, \omega) \cdot \cos \theta \, d\omega \leq 1. \quad (8.23)$$

Az energiamegmaradás elve értelmében az árnyalási egyenlet integráloperátora *kontrakció*, azaz a visszavert radianciafüggvény normája az eredeti radianciafüggvény normájánál kisebb. Ennek az a következménye, hogy az operátor egymás utáni alkalmazása során a visszavert radiancia zérushoz tart. Miként a megoldási módszerek ismertetésénél látni fogjuk, a kontrakció garantálja, hogy az iterációs megoldások konvergálnak.

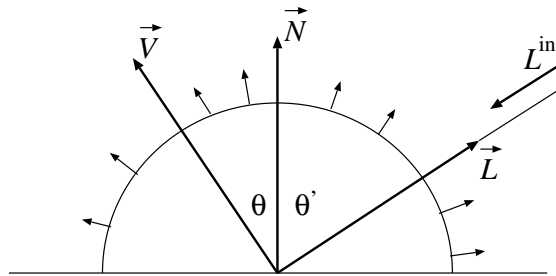
A reciprocitást és az energiamegmaradás elvét nem sértő BRDF-eket *fizikailag plauzibilis*nek nevezzük [Lew93].

8.6.1. Klasszikus BRDF modellek

A BRDF modellek bemutatása során a következő jelöléseket használjuk: \vec{N} a felületemre 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.

8.6.2. Lambert-törvény

Optikailag nagyon durva, ún. *diffúz* anyagok esetén a visszavert radiancia független a nézeti iránytól. Fehérre meszelt falra, homokra, matt felületre nézve ugyanazt a hatást érzékeljük ha merőlegesen nézünk rá, mintha élesebb szögben vizsgálódnánk.



8.6. ábra. Diffúz visszaverődés

A Helmholtz-féle reciprocitás értelmében a BRDF ekkor a bejövő iránytól sem függhet, azaz a BRDF konstans:

$$f_r(\vec{L}, \vec{V}) = k_d. \quad (8.24)$$

Az energiamegmaradás miatt az albedo diffúz visszaverődés esetén sem lehet 1-nél nagyobb, így a k_d diffúz visszaverődési együtthatóra a következő korlát állítható fel:

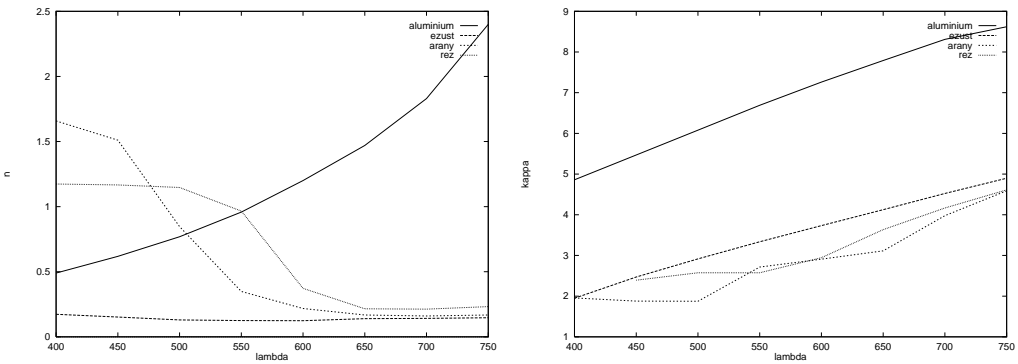
$$a(\vec{L}) = \int_{\Omega} k_d \cdot \cos \theta \, d\omega = k_d \cdot \pi \implies k_d \leq \frac{1}{\pi}. \quad (8.25)$$

8.6.3. Ideális visszaverődés

Az *ideális tükör* teljesíti a *geometriai optika* által kimondott *visszaverődési törvényt* miszerint a beesési irány (\vec{L}), felületi normális (\vec{N}) és a kilépési irány (\vec{V}) egy síkban van, és a θ' beesési szög megegyezik a θ visszaverődési szöggel ($\theta' = \theta$). Az ideális tükör tehát csak az \vec{R} visszaverődési irányba ver vissza, egyéb irányokba nem. A BRDF tehát *Dirac-delta* függvénnyel adható meg (a Dirac-delta a 0 értéknél végtelen, minden más értéknél zérus, de integrálja 1):

$$f_r(\vec{L}, \vec{V}) = k_r \cdot \frac{\delta(\vec{R} - \vec{V})}{\cos \theta'}, \quad \text{az energiamegmaradáshoz } k_r \leq 1. \quad (8.26)$$

Még a tökéletes tükrök is elnyelik a beérkező fény egy részét. A visszavert és beeső energia hányadát az anyag *Fresnel-együtthatója* fejezi ki, ami pedig az anyag *törésmutatójából* számítható ki. A törésmutató dielektrikumoknál skalár, az elektromosságot vezető fémeknél azonban komplex szám. Jelöljük a törésmutató valós részét n -nel, a vezetőképességet kifejező képzetes részét pedig κ -val.



8.7. ábra. Arany, réz és ezüst törésmutatója a hullámhossz függvényében

Legyen a beérkező fénysugár és a felületi normális által bezárt szög θ' , a törési irány és a normális közötti szög pedig θ . A *Fresnel-egyenletek* a visszavert és a be-

érkező fénynyalábok energiahányadát fejezik ki külön arra az esetre, amikor a fény polarizációja párhuzamos, és arra, amikor a polarizáció merőleges a felülettel:

$$F_{\perp}(\lambda, \theta') = \left| \frac{\cos \theta - (n + \kappa j) \cdot \cos \theta'}{\cos \theta + (n + \kappa j) \cdot \cos \theta'} \right|^2, \quad F_{\parallel}(\lambda, \theta') = \left| \frac{\cos \theta' - (n + \kappa j) \cdot \cos \theta}{\cos \theta' + (n + \kappa j) \cdot \cos \theta} \right|^2, \quad (8.27)$$

ahol $j = \sqrt{-1}$. Ezen egyenleteket az elektromágneses hullámok terjedését leíró Maxwell-egyenletekből származtathatjuk. Nem polarizált fény esetében a párhuzamos (\vec{E}_{\parallel}) és merőleges (\vec{E}_{\perp}) mezőknek ugyanaz az amplitúdója, így a visszaverődési együttható:

$$k_r = F(\lambda, \theta') = \frac{|F_{\parallel}^{1/2} \cdot \vec{E}_{\parallel} + F_{\perp}^{1/2} \cdot \vec{E}_{\perp}|^2}{|\vec{E}_{\parallel} + \vec{E}_{\perp}|^2} = \frac{F_{\parallel} + F_{\perp}}{2}. \quad (8.28)$$

8.6.4. Ideális törés

Az ideális törés során a fény útja követi a *Snellius-Descartes-törvényt*, miszerint a beesési irány (\vec{L}), felületi normális (\vec{N}) és a törési irány (\vec{V}) egy síkban van, és

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

ahol n az anyag relatív törésmutatója. Így a BRDF az ideális visszaverődéshez hasonlóan ugyancsak Dirac-delta jellegű függvény

$$f_r(\vec{L}, \vec{V}) = k_t \cdot \frac{\delta(\vec{T} - \vec{V})}{\cos \theta'}, \quad (8.29)$$

ahol \vec{T} a törési irány.

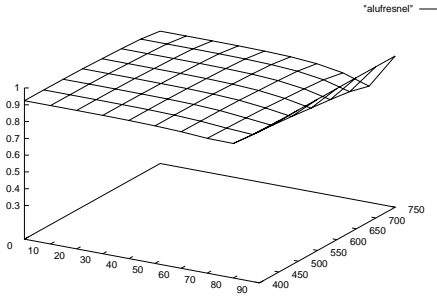
8.6.5. Phong illuminációs modell és változatai

Az inkoherens visszaverődést általában két tényezőre bontjuk. Diffúz visszaverődésre, amelyet a Lambert-törvénnyel írunk le, és spekuláris visszaverődésre, amelyre külön modellt állítunk fel.

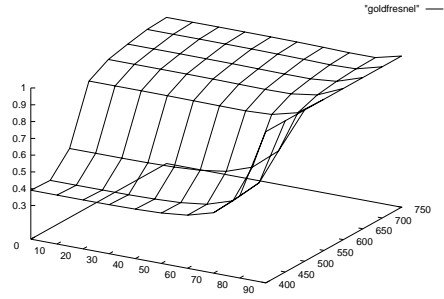
A Phong BRDF a spekuláris visszaverődés egyszerű empirikus modellje [Pho75]. A spekuláris felületek a beérkező fény jelentős részét az elméleti visszaverődési irány környezetébe verik vissza. Ezt a jelenséget modellezhetjük bármely olyan függvénnyel, amely a visszaverődési irányban nagy értékű, és attól távolodva rohamosan csökken.

Phong a következő függvényt javasolta erre a célja:

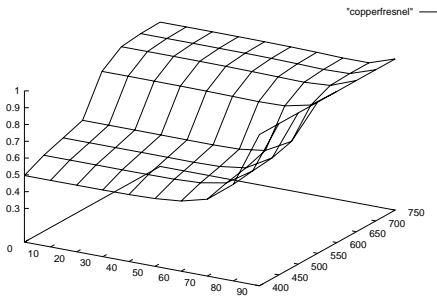
$$f_{r,\text{Phong}}(\vec{L}, \vec{V}) = k_s \cdot \frac{(\vec{R} \cdot \vec{V})^n}{(\vec{N} \cdot \vec{L})} \quad (8.30)$$



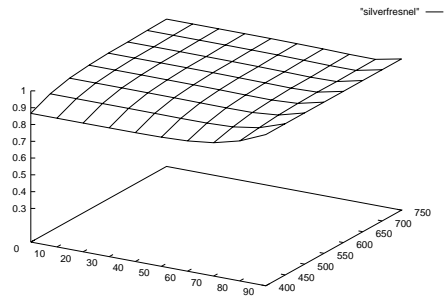
Alumínium



Arany

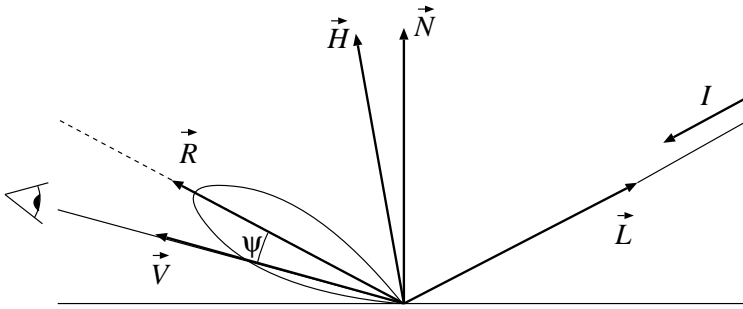


Réz



Ezüst

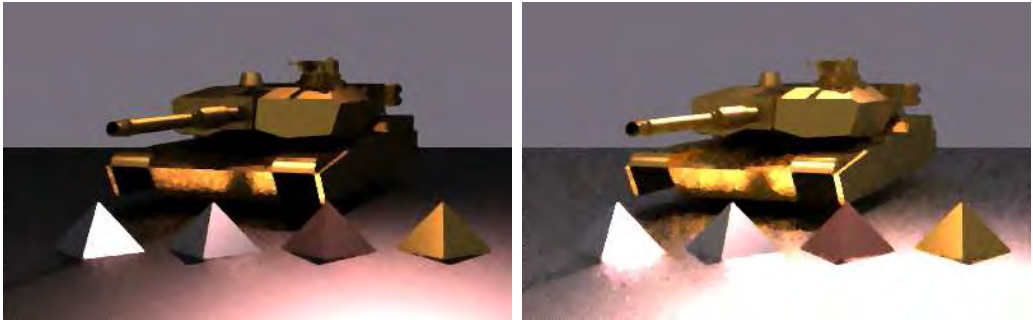
8.8. ábra. Alumínium, arany, réz és ezüst Fresnel-együtthatója a hullámhossz és a beesési szög függvényében



8.9. ábra. Szekuláris visszaverődés

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. A k_s faktort dielektrikumoknál tekinthetjük hullámhossz és beesési szög függetlennek (egy műanyagon, a fehér fény által létrehozott tükrös visszaverődés fehér).



8.10. ábra. A reciprok Phong (bal) és a max Phong (jobb) modellek összehasonlítása ($n = 20$). A reciprok Phong modell nagy látószögekre elfeketedik.

Az eredeti Phong-modell fizikailag nem plauzibilis, mert nem szimmetrikus. Ezért a fotorealisztikus képszintézisben ehelyett a következő változatokat használják [ICG86]:

$$f_{r,\text{reciprocalPhong}}(\vec{L}, \vec{V}) = k_s \cdot (\vec{R} \cdot \vec{V})^n \quad (8.31)$$

Az ilyen modell által visszavert radiancia nagy beesési szögekre zérushoz tart, ami nem felel meg a gyakorlati tapasztalatainknak. Ezt a hiányosságot küszöböli ki a következő változat [NNSK98]:

$$f_{r,\text{maxPhong}}(\vec{L}, \vec{V}) = k_s \cdot \frac{(\vec{R} \cdot \vec{V})^n}{\max((\vec{N} \cdot \vec{V}), (\vec{N} \cdot \vec{L}))} \quad (8.32)$$

Az energiamegmaradáshoz a következő feltételt kell garantálni [LW94]:

$$k_s \leq \frac{n+2}{2\pi}. \quad (8.33)$$

Ha a k_s paramétert a Fresnel-együttható alapján határozzuk meg, akkor gondot jelent az, hogy milyen beesési szögre tekintjük 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. Így a beesés szögének koszinuszát a $(\vec{H} \cdot \vec{L})$ skalárszorzatból számolhatjuk ki.

8.7. Fényelnyelő anyagok

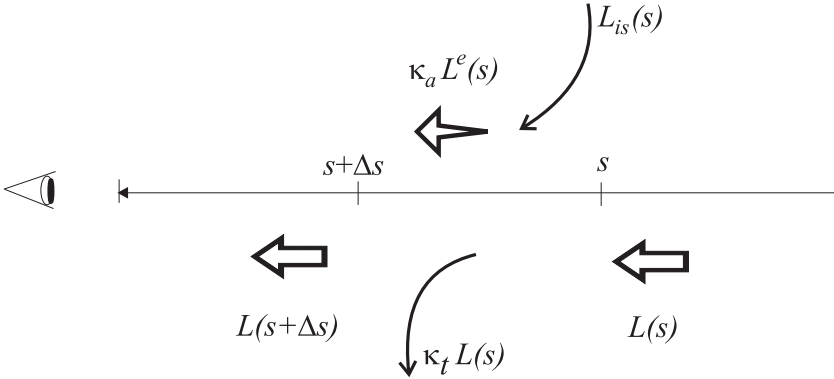
Az árnyalási, illetve a potenciál egyenlet származtatása során feltételeztük, hogy a felületek között a fényintenzitás nem csökken, azaz a térben nincsenek fényelnyelő és szóró anyagok (*participating media*). Ha *felhőket*, *tüzet*, *füstöt*, *ködöt*, stb. szeretnénk megjeleníteni, akkor a korábbi feltételezésekkel alkotott modellek elégtelennek bizonyulnak, tehát általánosítani kell őket.

Tekintsünk egy fényelnyelő, fényszóró, sőt akár fényemittáló (*tűz*) 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 $-\kappa_t \cdot L$ mértékű változás (*outscattering*).
- A fényintenzitás az anyag saját emissziójával növekedhet: $\kappa_a \cdot L^e$.
- Az eredetileg más irányú fotonok a molekulákba ütközve éppen a sugár irányában folytatják az útjukat (*inscattering*). Ha az ω' irányból az elemi ds szakasz környezetébe $L_i(\omega')$ radiancia érkezik, az ω sugárirányban történő visszaverődés valószínűség-sűrűségfüggvénye pedig $f(\omega', \omega)$, akkor ez a hatás az intenzitást

$$L_{\text{is}}(s) = \int_{\Omega} L_i(\omega') \cdot f(\omega', \omega) d\omega'$$

mennyiséggel növeli.



8.11. ábra. A sugár intenzitásának változása

Összefoglalva a sugár radianciájára a következő egyenlet érvényes:

$$\frac{dL(s, \omega)}{ds} = -\kappa_t(s) \cdot L(s, \omega) + \kappa_a(s) \cdot L^e(s, \omega) + L_{is}(s, \omega) =$$

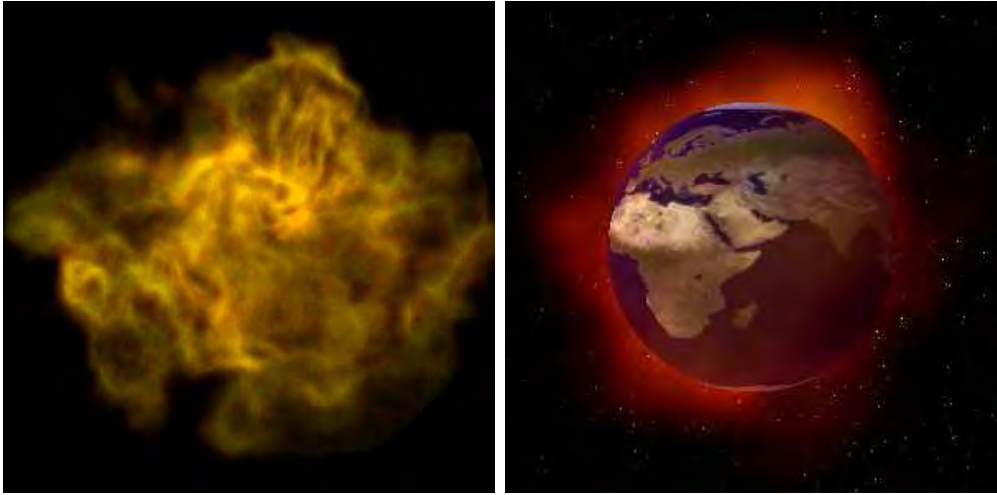
$$-\kappa_t(s) \cdot L(s, \omega) + \kappa_a(s) \cdot L^e(s, \omega) + \int_{\Omega} L_i(s, \omega') \cdot f(\omega', \omega) d\omega'. \quad (8.34)$$

Ebben az egyenletben az ismeretlen radiancia 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 radianciától és valaki megszúgná nekünk az L_i 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 egyszerűsített integrálegyenletet a *szuperpozíció elv* segítségével oldhatjuk meg. Először tegyük fel, hogy az intenzitást növelő $L^e(s, \omega)$ és $L_{is}(s, \omega)$ tényezők csak egy τ pontban különböznek zérustól. Ezek hatását egy s pontban úgy kapjuk, hogy figyelembe vesszük a folyamatos csökkenést:

$$\delta_{\tau} L(s, \omega) = e^{-\int_s^{\tau} \kappa_t(p) dp} \cdot (\kappa_a(\tau) \cdot L^e(\tau, \omega) + L_{is}(\tau, \omega)). \quad (8.35)$$

Amennyiben nem csak a τ pontban növekedhet az intenzitás, úgy az elemi hatásokat



8.12. ábra. A “nagy bumm” és a föld környezeti katasztrófájának szimulációja [Sza95]

összegezni kell, tehát:

$$L(s, \omega) = \int_s^T \delta_\tau L(s, \omega) d\tau = \int_s^T e^{-\int_s^\tau \kappa_t(p) dp} \cdot (\kappa_a(\tau) \cdot L^e(\tau, \omega) + L_{is}(\tau, \omega)) d\tau, \quad (8.36)$$

ahol T a maximális sugárparaméter.

8.8. Program: BRDF modellek

Az anyagok optikai tulajdonságai a BRDF függvénnyel jellemezhetők. A BRDF függvény visszatérési értéke egy spektrum, hiszen a belépési és kilépési irányon kívül még függhet a hullámhossztól is.

```
typedef Spectrum<NLAMBDA> SColor;

//=====
class Material {
//=====
public:
    virtual SColor BRDF(Vector3D& L, Vector3D& N, Vector3D& V);
    virtual double AverageAlbedo( Vector3D& N, Vector3D& V );
};
```

Az Emitter fényt bocsát ki magából:

```
//=====
class Emitter : public Material {
//=====
    SColor LE;
public:
    SColor& Le() { return LE; }
};
```

A diffúz anyagok a fényt a különböző irányokba egyenletesen verik vissza. A DiffuseMaterial osztály BRDF-je tehát konstans.

```
//=====
class DiffuseMaterial : virtual public Material {
//=====
    SColor Kd;
public:
    SColor& kd() { return Kd; }
    SColor BRDF(Vector3D& L, Vector3D& N, Vector3D& V) { return Kd; }
    double AverageAlbedo( Vector3D& N, Vector3D& V ) {
        return Kd.Luminance() * M_PI;
    }
};
```

A SpecularMaterial a Phong-modell szerinti BRDF-t valósítja meg. A K_s visszaverődési tényező a fizikai plauzibilitáshoz még megengedett maximumra vetített érték. Ezen kívül az osztály még a shine simasági paramétert tartalmazza.

```
//=====
class SpecularMaterial : virtual public Material {
//=====
    SColor Ks;
    double shine;
public:
    SpecularMaterial( ) : Ks(0) { shine = 10; }
    SColor& ks() { return Ks; }
    double& Shine( ) { return shine; }
    SColor BRDF(Vector3D& L, Vector3D& N, Vector3D& V) {
        double cos_in = L * N;
        if (cos_in > 0 && ks() != 0) {
            Vector3D R = N * (2.0 * cos_in) - L;
            double cos_refl_out = R * V;
            if (cos_refl_out > EPSILON) {
                SColor ref = ks() * (shine + 2) / M_PI / 2.0;
                return (ref * pow(cos_refl_out, shine));
            }
        }
    }
};
```

```

    }
  }
  return SColor(0);
}
double AverageAlbedo( Vector3D& N, Vector3D& V ) {
    return (ks().Luminance() * 2 * M_PI/(Shine() + 2));
}
};

```

A Phong-modell visszaverődési tényezője műanyagoknál általában hullámhossz független konstans (egy fehér fényel megvilágított műanyag a tükrözési folt a műanyag színétől függetlenül fehér), fémeknél azonban erősen függ a hullámhossztól és a beesési szögtől. Ezt a függést úgy modellezhetjük, ha a visszaverődési tényezőt megszorozzuk az anyag *Fresnel-együtthatójával*. A Fresnel-függvényt az anyag komplex törésmutatójából számíthatjuk. A következő programrészlet különböző hullámhosszokra megadja az alumínium és az arany komplex törésmutatóját, valamint a Fresnel-függvény számítási algoritmusát.

```

#define NREFIDX 8

struct RefIdx {double lambda, n, k; }
    alu[NREFIDX] = {
        {400, 0.4900, 4.8600},
        {450, 0.6180, 5.4700},
        {500, 0.7690, 6.0800},
        {550, 0.9580, 6.6900},
        {600, 1.2000, 7.2600},
        {650, 1.4700, 7.7900},
        {700, 1.8300, 8.3100},
        {750, 2.4000, 8.6200}
    },

    gold[NREFIDX] = {
        {400, 1.6580, 1.9560},
        {450, 1.5102, 1.8788},
        {500, 0.8469, 1.8753},
        {550, 0.3485, 2.7144},
        {600, 0.2177, 2.9097},
        {650, 0.1676, 3.1138},
        {700, 0.1605, 3.9784},
        {750, 0.1680, 4.5886},
    };

```

```

//=====
class FresnelFunction {
//=====
    RefIdx refidx[NREFIDX];
public:
    FresnelFunction( RefIdx r[] ) {
        for(int l = 0; l < NREFIDX; l++) refidx[l] = r[l];
    }
    double Fresnel( double lambda, double theta ) {
        double n, k;
        for( int l = 1; l < NREFIDX; l++ ) {
            if (lambda < refidx[l].lambda) {
                double la2 = lambda - refidx[l-1].lambda;
                double la1 = refidx[l].lambda - lambda;
                double la = la1 + la2;
                n = (la1 * refidx[l-1].n + la2 * refidx[l].n)/la;
                k = (la1 * refidx[l-1].k + la2 * refidx[l].k)/la;
                break;
            }
        }
        double t1 = n*n - k*k - sin(theta) * sin(theta);
        double t2 = sqrt(t1*t1 + 4.0*n*n*k*k);
        double a2 = 0.5 * (t2 + t1), a = sqrt(a2);
        double t3 = a2 + 0.5 * (t2 - t1);
        double t4 = 2.0 * a * cos(theta);
        double fsd = (t3 + t4 + cos(theta) *cos(theta));
        double Fs = (fsd > EPSILON) ?
            (t3 - t4 + cos(theta) *cos(theta))/fsd : 0;
        double t5 = 2.0 * a * sin(theta) * tan(theta);
        double t6 = t3 + sin(theta)*sin(theta) * tan(theta)*tan(theta);
        double Fp = (t6 + t5 > EPSILON) ?
            Fs * (t6 - t5)/(t6 + t5) : 0;
        return ((Fp + Fs)/2.0);
    }
};

```

9. fejezet

A 3D inkrementális képszintézis

A 3D képszintézis során a 3D lokális koordináta-rendszerekben, vagy közvetlenül a világ-koordináta-rendszerben definiált modellről egy, a világ-koordináta-rendszerben elhelyezett kamerával fényképet készítünk, és azt a képernyő nézetében megjelenítjük. Az alapfeladatok — transzformáció, vágás, takarás és árnyalás — végrehajtása során két eljárást követhetünk. Az alapfeladatokat vagy pixelenként egymástól függetlenül hajtjuk végre, vagy pedig a feladatok egy részének elvégzése során elvonatkoztatunk a pixelektől, és az objektumtér nagyobb részeit egységesen kezeljük. Az első módszert sugárkövetésnek, a másodikat inkrementális képszintézisnek nevezzük. Mindkét eljárásnak megvannak a maga előnyei és hátrányai.

A *sugárkövetés* (*ray-tracing*) teljesen önállóan kezel egy pixelt, és azt a pixelközépponton keresztül látható objektumnak megfelelően színezi ki. Egyetlen pixel számításához tehát az objektumtér egyetlen pontját kell ismernünk. Ezen pont azonosításához félegyenest indítunk a szemből az adott pixel középpontján keresztül az objektumtérbe, meghatározzuk a félegyenes metszéspontjait az objektumtér objektumaival, majd a metszéspontok közül kiválasztjuk a szemhez legközelebbit. Vegyük észre, hogy a metszéspont előállításával a transzformáció, vágás és takarás feladatait is egy füst alatt elintéztük! Az egyetlen még megoldandó probléma az árnyalás, amely a látható pont radianciáját az optikai modell egyszerűsített változatával számítja ki.

Az *inkrementális képszintézis* az alapfeladatok egy részét a pixel felbontástól teljesen függetlenül, az objektumtér ábrázolási pontosságának megfelelően a feladathoz optimálisan illeszkedő koordináta-rendszerben végzi el, a feladatok másik részénél pedig 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 nulláról. Annak érdekében, hogy az objektumainkat transzformálni és vágni tudjunk, olyan reprezentációra van szükségünk, amelyre ezen műveletek könnyen elvégezhetőek, és nem vezetnek ki a reprezentáció-

ból. A 2D képszintézishez hasonlóan — ahol ezt a feladatot a vektorizációval oldottuk meg — az inkrementális képszintézis első lépéseként a felületeket háromszöghálóval közelítjük. Ezt a lépést *tesszellációnak* nevezzük.

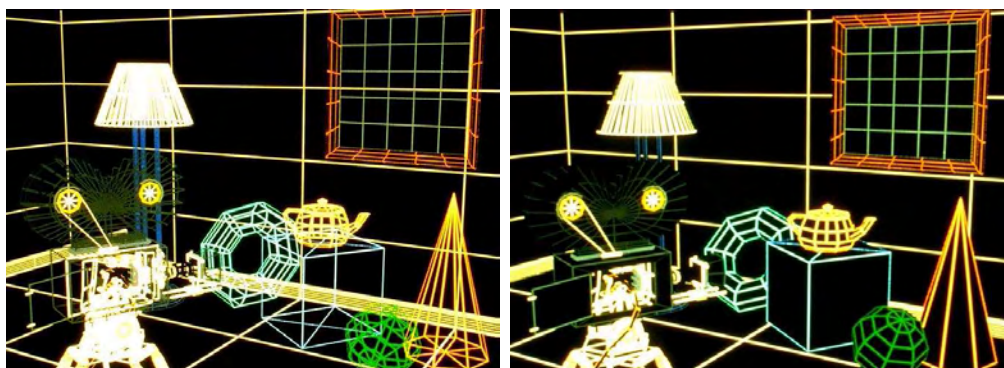
A sugárkövetést összehasonlítva az inkrementális képszintézissel, a sugárkövetés javára elmondható, hogy nem igényel tesszellációt, explicit transzformációkat és vágást, így lényegesen könnyebben implementálható. Hátránya viszont, hogy mivel a pixeleket függetlenül kezeli, nem használja újra az egyszer nagy nehezen megszerzett takarási vagy árnyalási információkat, így kétségtelenül lassabb, mint az inkrementális algoritmus.

9.1. ábra. Az inkrementális 3D képszintézis lépései

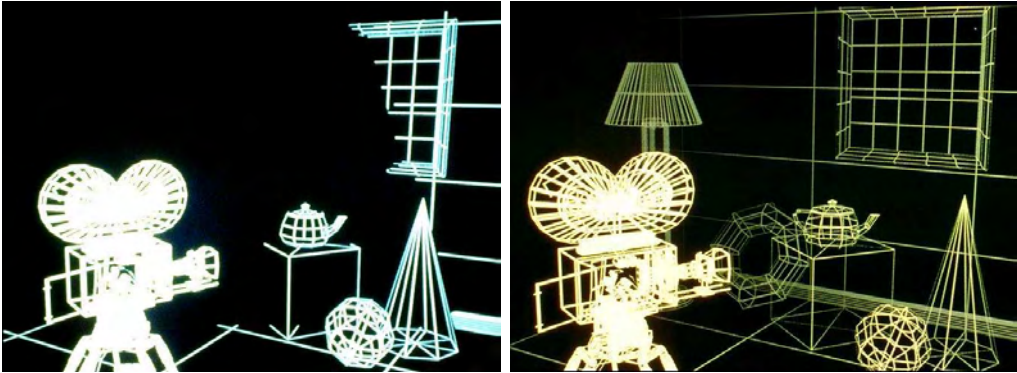
Ebben a fejezetben az inkrementális képszintézissel foglalkozunk, a sugárkövetésre a következő fejezetben térünk vissza. Az inkrementális képszintézis során a következő feladatokat végezzük el:

1. *Tesszelláció*: A virtuális világban tárolt szabadformájú elemeket (pl. felületek, testek) 3D szakaszok és poligonok halmazával közelítjük. Amennyiben a képszintézis során végig kitöltött poligonokkal dolgozunk, *tömörtest megjelenítésről* (*solid rendering*) fogunk beszélni, ha viszont a testeket, felületeket a poligonhálós közelítésük éleinek felrajzolásával jelenítjük meg, *huzalváz megjelenítéshez* (*wireframe rendering*) jutunk.
2. *Transzformáció*: A lokális koordináta-rendszerben adott geometriai elemekre a világ-koordináta-rendszerben, majd a képernyő-koordináta-rendszerben van szükségünk. A koordináta-rendszer váltás homogén lineáris geometriai transzformáció igényel, amit mátrixszorzással valósítunk meg.

3. *Vágás:* Ebben a lépésben eltávolítjuk azon geometriai elemeket, illetve a geometriai elemek azon részeit, amelyek nem vetülhetnek az ablak téglalapjára.
4. *Takarási feladat:* A transzformációk több objektumot is vetíthetnek ugyanarra a képpontra. Ilyenkor el kell döntenünk, hogy melyik van a legközelebb a szempozícióhoz, azaz ebben a pontban melyik takarja a többi objektumot. Huzalváz megjelenítés esetén ezt a lépést ki is hagyhatjuk (9.2. ábra).
5. *Árnyalás:* Ha sikerült eldönteni, hogy egy képpontban melyik objektum látszik, akkor a képpontot ennek megfelelően kell kiszínezni. Legegyszerűbb esetben a 2D grafika mintájára az objektumok saját színét használjuk. Ezt gyakran alkalmazuk huzalváz megjelenítéskor (9.2. ábra), sőt használhatjuk tömörtest megjelenítéskor is, de ez nagyon csúnya képeket eredményez (17.16. ábra). Szép képekhez akkor juthatunk, ha a színt a térben fennálló fényviszonyokból a fizikai folyamatok egyszerűsített szimulációjával számítjuk (17.16. ábra). Még huzalváz megjelenítés esetén is érdemes valamilyen rendkívül egyszerű színezést alkalmazni. Például a távolabbi éldarabokat levághatjuk, vagy az élek azon pontjait, amelyek a szemhez közelebb vannak, nagyobb intenzitással rajzolhatjuk, mint a hátrébb lévőket. Ezen eljárásokat *mélységi vágásnak* (*depth clipping*) illetve *mélységi intenzitás modulációnak* (*depth cuing*) nevezzük (9.3. ábra).



9.2. ábra. Huzalváz képszintézis takarás nélkül (bal) és takarással (jobb) (Pixar)



9.3. ábra. Távlabbi élék vágása (bal) és mélységi intenzitásmoduláció (jobb) (Pixar)

9.1. Felületek tesszellációja

A *tesszelláció* a 2D vektorizáció általánosításaként képzelhető el. Egy $\vec{r}(u, v)$ felületet oly módon közelíthetünk háromszögekkel, hogy az $u \in [0, 1], v \in [0, 1]$ paramétertartományban kijelölünk $n \times m$ pontot és az

$$[\vec{r}(u_i, v_j), \vec{r}(u_{i+1}, v_j), \vec{r}(u_{i+1}, v_{j+1})] \quad \text{és a} \quad [\vec{r}(u_i, v_j), \vec{r}(u_{i+1}, v_{j+1}), \vec{r}(u_i, v_{j+1})]$$

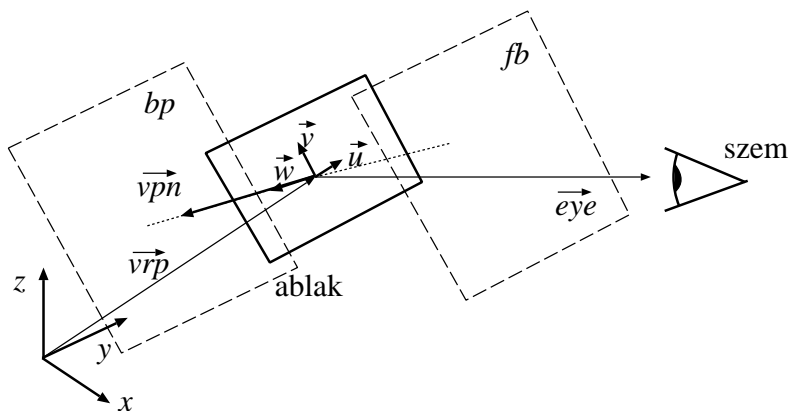
háromszögeket minden $i = 1 \dots n-1$ és $j = 1 \dots m-1$ indexre hozzáadjuk a keletkező háromszöglistához.

9.2. Modellezési transzformáció

Ha az objektumok a saját lokális modellezési koordinátarendszereikben állnak rendelkezésre, akkor a képszintézis során a közös világ-koordinátarendszerbe kell átvinni őket. Ez egy \mathbf{T}_M *modellezési transzformációt* igényel, amely a 2D modellezési transzformáció közvetlen 3D kiterjesztése.

9.3. Kamera definíció

A 3D grafikában a szempozícióból egy téglalap alakú ablakon keresztül látható képet szeretnénk előállítani, így a pixeleknek az ablakon kis téglalapok felelnek meg. Az ablak a világ-koordinátarendszerben általános helyzetű és orientációjú lehet, az ablak mögött a szempozíció is bárhol elhelyezhető.



9.4. ábra. A kamera geometriai paramétereinek a definíciója

A kamerát a következő paraméterekkel definiálhatjuk:

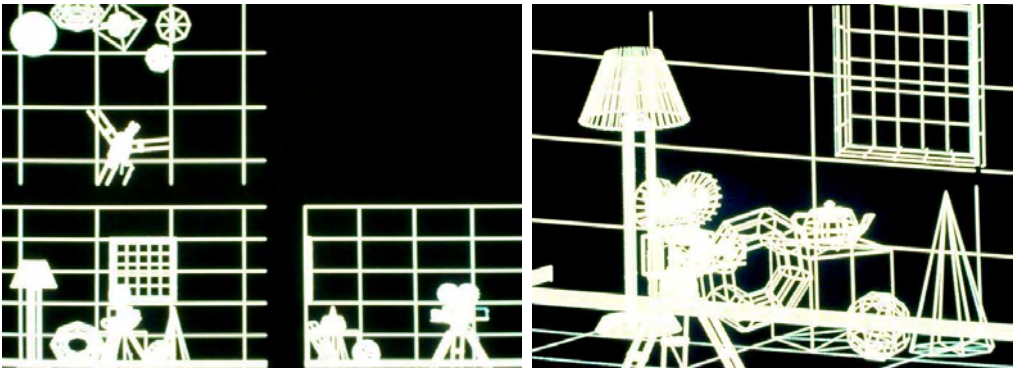
- Az ablak középpontját egy világ-koordináta-rendszerbeli ponttal, ún. *nézeti referencia ponttal* adjuk meg, amelyet általában a $v\vec{r}p$ vektorral jelölünk. Az ablak orientációját egy u, v, w koordináta-rendszerrel jellemezzük, amelynek origója a nézeti referencia pont, az \vec{u} és \vec{v} vektorok pedig az ablak vízszintes és függőleges irányát határozzák meg. A \vec{w} vektor az ablak síkjára merőleges egységvektor. Az $\vec{u}, \vec{v}, \vec{w}$ egységvektorok definíciójához a felhasználó általában megadja az ablak síkjának normálvektorát ($v\vec{p}n$) és a függőleges irányt jelölő $v\vec{u}p$ vektort, amelyekből az egységvektorok a következőképpen számíthatók ki:

$$\vec{w} = \frac{v\vec{p}n}{|v\vec{p}n|}, \quad \vec{u} = \frac{\vec{w} \times v\vec{u}p}{|\vec{w} \times v\vec{u}p|}, \quad \vec{v} = \vec{u} \times \vec{w}. \quad (9.1)$$

Szemben a jobbsodrású x, y, z világ-koordináta-rendszerrel, az u, v, w koordináta-rendszer balsodrású, hiszen ez felel meg annak a természetes képnek, hogy az \vec{u} jobbra mutat, a \vec{v} felfelé és a \vec{w} pedig arra, amerre nézünk.

- Az ablak vízszintes és függőleges méreteit két számmal, a w_w és w_h paraméterekkel adjuk meg. A *zoom* kameraművelet az ablak méretének szabályozásával érhető el.
- A kép a virtuális világnak az ablak síkjára vett vetülete. A képszintézis során általában kétféle *vetítési típus* használatos: *párhuzamos vetítés*, amikor a vetítősugarak párhuzamosak, és *perspektív vetítés*, amikor a vetítősugarak a szempozícióban találkoznak.

- Perspektív vetítés esetén a *szempozíció* határozza meg a vetítési középpontot (*ēye*). Párhuzamos vetítéskor a kamera csupán egy irányt definiál, amellyel a vetítősugarak párhuzamosak.
- Nyilván az objektumtérnek csak azon részei láthatók a képen, amelyek a szem előtt helyezkednek el, tehát a szem mögötti objektumokat a képszintézis során vágással el kell távolítani. A vágási tartományt az első és hátsó vágósík bevezetésével tovább korlátozhatjuk. Csak azon objektumok vesznek részt a képszintézisben, amelyek az ablakkal párhuzamos két vágósík között helyezkednek el. A vágósíkok az ablak síkjával párhuzamosak így az u, v, w koordinátarendszerben egy-egy számmal megadhatók (fp az első vágósíkra, bp a hátsó vágósíkra). Perspektív vetítés esetén ez a paraméter a szemtől való távolságot, párhuzamos vetítés esetén pedig a nézeti referencia ponttól való távolságot jelenti.



9.5. ábra. A koordinátatengelyekkel párhuzamos irányokkal dolgozó párhuzamos vetítés (bal) és általános helyzetű perspektív vetítés (jobb) (Pixar)

9.4. A nézeti transzformáció

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, ekkor azonban egy pont vetítése egy általános helyzetű egyenes és az ablak metszéspontjának a kiszámítását igényelné, a takarás pedig az általános pozíciójú szemtől való távolsággal dolgozna. Sokkal jobban járunk, ha ezen műveletek előtt átranszformáljuk a teljes objektumteret egy olyan

koordinátarendszerbe, ahol a vetítés és takarás triviálissá válik. Ezt a rendszert *3D képernyő-koordinátarendszernek* 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. Célszerűen a Z koordináta normalizált, azaz az első vágósík transzformált koordinátája 0, a hátsó vágósík pedig 1.

A transzformációt egy koordinátarendszeren átvezető transzformáció sorozattal definiáljuk, bár végül az eredő transzformációt egyetlen mátrixszorzással valósítjuk meg. A transzformációsorozat, és így az eredő mátrix is függ attól, hogy a vetítés párhuzamos avagy perspektív.

9.4.1. Világ-koordinátarendszer — ablak-koordinátarendszer transzformáció

Először a pontokat az ablakhoz rögzített u, v, w koordinátarendszerbe visszük át. Ez egy koordinátarendszerváltó transzformáció, így a 7.2. fejezet eredményei hasznosíthatók. A keresett transzformáció:

$$[\alpha, \beta, \gamma, 1] = [x, y, z, 1] \cdot \mathbf{T}_{uvw}^{-1}, \quad (9.2)$$

ahol a \mathbf{T}_{uvw} mátrix sorai az $\vec{u}, \vec{v}, \vec{w}$ egységvektorok és az origót meghatározó $v\vec{r}_p$ nézeti referencia pont világ-koordinátarendszerbeli koordinátái:

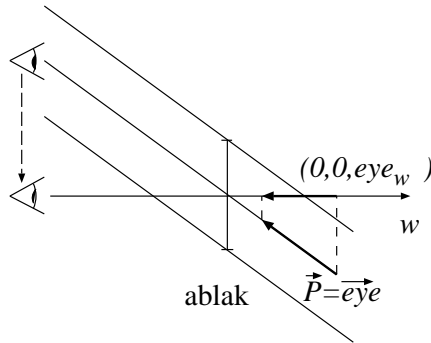
$$\mathbf{T}_{uvw} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ vrp_x & vrp_y & vrp_z & 1 \end{bmatrix}. \quad (9.3)$$

9.4.2. Ablak-képtér transzformáció párhuzamos vetítés esetén

Nyírás

Speciális esetben előfordulhat, hogy nem merőlegesen nézünk rá az ablakra, azaz a szempozíció eye_u illetve eye_v koordinátái zérustól különbözők (9.6. ábra). A nézeti irányt kiegyenesíthetjük az objektumtér torzításával. A torzítás egy *nyírás transzformációt* igényel. Az eredeti w koordinátát megtartó nyírás általános formája:

$$\mathbf{T}_{\text{shear}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s_u & s_v & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (9.4)$$



9.6. ábra. Nyírás, amely a nézeti irányt az ablakkal merőlegessé teszi

A transzformációnak az $[eye_u, eye_v, eye_w, 1]$ projektort a $[0, 0, eye_w, 1]$ vektorba kell átvinnie. Ezt a feltételt a következő s_u és s_v értékek elégítik ki:

$$s_u = -\frac{eye_u}{eye_w}, \quad s_v = -\frac{eye_v}{eye_w}. \quad (9.5)$$

Képernyő transzformáció

Párhuzamos vetítés esetén a nyírás után a világ képre vetíthető tartománya téglatest alakú, amelynek oldalai párhuzamosak a koordinátatengelyekkel, két átellenes sarokpontja pedig $(-w_w/2, -w_h/2, fp)$, $(w_w/2, w_h/2, bp)$. A képernyő-koordinátarendszerben a világ ugyancsak ilyen állású téglatest, de a sarokpontjai $(V_x - V_{sx}/2, V_y - V_{sy}/2, 0)$, $(V_x + V_{sx}/2, V_y + V_{sy}/2, 1)$.

9.7. ábra. Nézeti transzformáció párhuzamos vetítés esetén

Az első téglatestet a másikba átvivő transzformáció:

$$\mathbf{T}_{\text{viewport}} = \begin{bmatrix} V_{sx}/w_w & 0 & 0 & 0 \\ 0 & V_{sy}/w_h & 0 & 0 \\ 0 & 0 & 1/(bp - fp) & 0 \\ V_x & V_y & -fp/(bp - fp) & 1 \end{bmatrix}. \quad (9.6)$$

Összefoglalva, párhuzamos vetítés esetén a teljes nézeti transzformáció a következő elemi transzformációk kompozíciója:

$$\begin{aligned} \mathbf{T}_V &= \mathbf{T}_{\text{uvw}}^{-1} \cdot \mathbf{T}_{\text{shear}} \cdot \mathbf{T}_{\text{viewport}}, \\ [X, Y, Z, 1] &= [x, y, z, 1] \cdot \mathbf{T}_V. \end{aligned} \quad (9.7)$$

A \mathbf{T}_V mátrixot *nézeti transzformációs mátrix*nak nevezzük.

9.4.3. Ablak-képtér transzformáció perspektív vetítés esetén

Mozgatás a szembe

Első lépésként a koordináta-rendszer középpontját az ablak középpontjából a szempozícióba helyezzük át. Ez egy $-\vec{e}_j e$ vektorral jellemzett eltolás:

$$\mathbf{T}_{\text{eye}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -eye_u & -eye_v & -eye_w & 1 \end{bmatrix}. \quad (9.8)$$

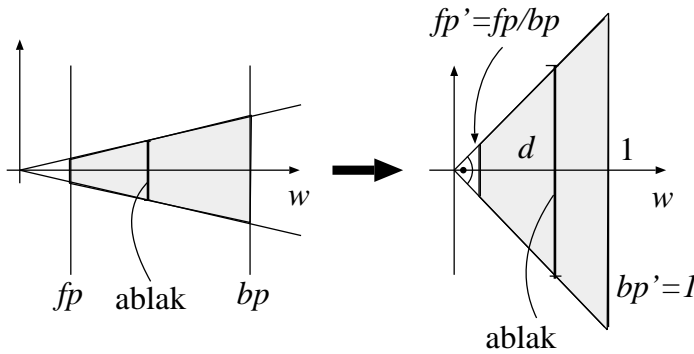
Nyírási transzformáció

Hasonlóan a párhuzamos vetítéshez, ha eye_u illetve eye_v nem zérus, az ablak középpontján átmenő vetítősugar nem merőleges az ablakra. A szükséges korrekciót az objektumoknak nyírási transzformációval történő torzításával végezzük el:

$$\mathbf{T}_{\text{shear}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -eye_u/eye_w & -eye_v/eye_w & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (9.9)$$

Normalizáló transzformáció

A nyírás után a képszintézisben részt vevő pontok tartománya egy szimmetrikus csonka gúla (9.8. ábra). A további műveletekhez normalizáljuk ezt a gúlát oly módon, hogy a



9.8. ábra. Normalizáló transzformáció

csúcsában a nyílásszög 90 fok legyen, a hátsó vágósík pedig az 1 értékre kerüljön. A normalizálás egy egyszerű skálázás a koordinátatengelyek mentén:

$$\mathbf{T}_{\text{norm}} = \begin{bmatrix} -2 \cdot eye_w / (w_w \cdot bp) & 0 & 0 & 0 \\ 0 & -2 \cdot eye_w / (w_h \cdot bp) & 0 & 0 \\ 0 & 0 & 1/bp & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (9.10)$$

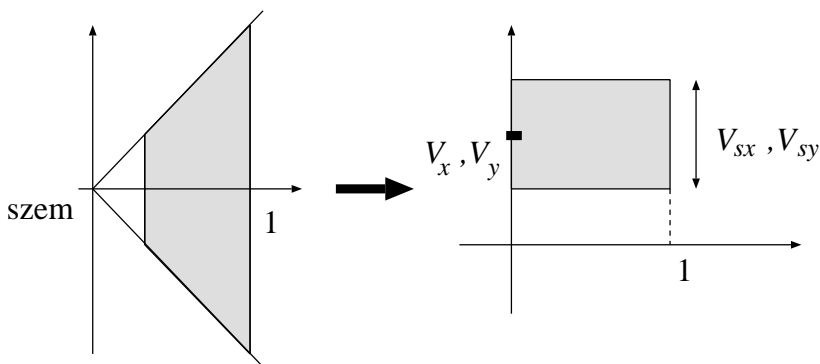
Perspektív transzformáció

Utolsó lépésként a csonka gúlát a képernyő-koordinátarendszer téglatestére kell leképezni.

Egy ilyen transzformáció a gúla csúcsát, azaz a szempozíciót, a végtelenbe viszi át, így nem lehet az euklideszi tér lineáris transzformációja. Szerencsére a projektív tér lineáris transzformációi között találunk megfelelőt:

$$\mathbf{T}_{\text{persp}} = \begin{bmatrix} V_{sx}/2 & 0 & 0 & 0 \\ 0 & V_{sy}/2 & 0 & 0 \\ V_x & V_y & bp/(bp - fp) & 1 \\ 0 & 0 & -fp/(bp - fp) & 0 \end{bmatrix}. \quad (9.11)$$

Miként behelyettesítéssel meggyőződhetünk róla, ez a transzformáció az eredetileg a szempozícióban található projektorokból párhuzamosakat csinál, hiszen a $[0, 0, 0, 1]$ szempozíciót valóban a $[0, 0, -fp/(bp - fp), 0]$ ideális pontba viszi át. A perspektív transzformáció az euklideszi tér nem lineáris transzformációja, ezért a transzformációs mátrix utolsó oszlopa nem a szokásos $[0, 0, 0, 1]$ vektor. Következésképpen a keletkező



9.9. ábra. Transzformáció a normalizált nézetből a képernyő-koordinátarendszerbe

homogén koordinátanégyes negyedik koordinátája nem lesz 1 értékű. Ezért, ha Descartes-koordinátákban szeretnénk megkapni a transzformáció eredményét, a 4. homogén koordinátával végig kell osztani a többi koordinátát. Így a teljes perspektív transzformáció:

$$\begin{aligned}
 [X_h, Y_h, Z_h, h] &= [X_c, Y_c, Z_c, 1] \cdot \mathbf{T}_{\text{persp}}, \\
 [X, Y, Z, 1] &= \left[\frac{X_h}{h}, \frac{Y_h}{h}, \frac{Z_h}{h}, 1 \right].
 \end{aligned}
 \tag{9.12}$$

A homogén lineáris transzformációk lineáris halmaz tartók, tehát egyenest egyenesbe, síkot síkba visznek át. Véges objektumok esetén, mint a szakasz vagy a poligon azonban, problémák merülhetnek fel, ha a $\mathbf{T}_{\text{persp}}$ mátrixszal való szorzás az objektum valamely pontját ideális pontba, azaz a $h = 0$ hipersíkra transzformálja. Mivel a szem a vetítés középpontja, és az ablak a vetítés síkja, ez akkor fordul elő, ha az objektumunk metszi a szempozíción átmenő, és az ablakkal párhuzamos síkot. Egy szem előtt kezdődő és a szem mögött végződő szakaszból olyan szakasz lesz, amely egy végtelen távoli pontot is tartalmaz. Ez még nem lenne baj, hiszen a projektív geometriában ez megengedett, csak a Descartes-koordinátarendszerbe való visszatérés során tudomásul kell venni, hogy a szakaszunk képe 2 félegyenes lesz. A perspektív transzformációt tehát nem lehet minden járulékos meg gondolás nélkül, csak a szakasz két végpontjára (vagy a poligon csúcsaira) végrehajtani, aztán azt mondani, hogy a keletkező szakasz a transzformált végpontok között van (vagy a keletkező poligont a transzformált csúcsok definiálják). Előfordulhat, hogy a transzformáció eredményét a szakasz egyenesének azon pontjai alkotják, amelyek éppenhogy nem a transzformált pontok között, hanem azok átellenes oldalain találhatók. Ez az *átfordulási probléma*, amit a vágási tartomány megfelelő beállításával és a vágás megfelelő időben történő elvégzésével

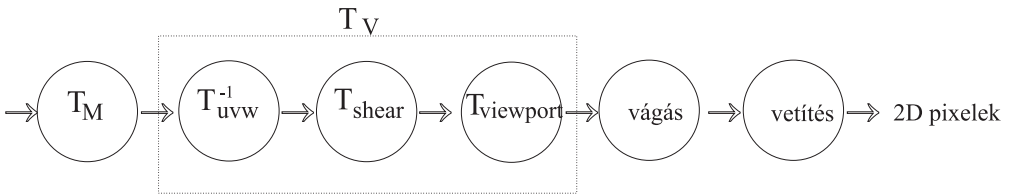
oldhatunk meg. Ha megköveteljük, hogy az első vágósík mindig a szem előtt legyen, és a vágást a perspektív transzformáció homogén osztásának elvégzése előtt végrehajtjuk, akkor a homogén osztás pillanatában már semelyik objektum sem tartalmazhat ideális pontot.

Összefoglalva, a perspektív vetítéskor érvényes nézeti transzformáció:

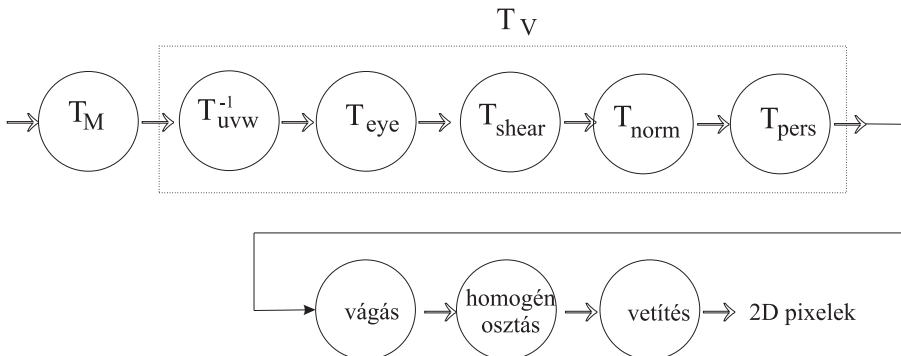
$$\begin{aligned} \mathbf{T}_V &= \mathbf{T}_{uvw}^{-1} \cdot \mathbf{T}_{eye} \cdot \mathbf{T}_{shear} \cdot \mathbf{T}_{norm} \cdot \mathbf{T}_{persp}, \\ [X_h, Y_h, Z_h, h] &= [x, y, z, 1] \cdot \mathbf{T}_V, \\ [X, Y, Z, 1] &= \left[\frac{X_h}{h}, \frac{Y_h}{h}, \frac{Z_h}{h}, 1 \right]. \end{aligned} \quad (9.13)$$

9.5. Nézeti csővezeték

A lokális modellezési koordinátarendszertől a képernyőig tartó transzformáció sorozatot *nézeti csővezetéknek* (viewing pipeline) nevezzük, amelyen az objektumokat definiáló pontok “végigfolynak”.



9.10. ábra. Nézeti csővezeték párhuzamos vetítés esetére



9.11. ábra. Nézeti csővezeték perspektív vetítés esetére

A vágási feladatot elvileg több koordináta-rendszerben is végre lehet hajtani, ezek közül a 9.10. ábra csupán egyetlen lehetőséget mutat be. A csővezeték végén a primitívek a képernyő-koordináta-rendszerben "csöpögnek ki", ahol optimális körülmények között oldhatjuk meg a takarási feladatot, és végezhetjük el a vetítést. Ebben a koordináta-rendszerben ugyanis két pont akkor takarja egymást, ha az X és Y koordinátáik megegyeznek, és az látszik, amelynek Z koordinátája kisebb. A vetítés pedig a Z koordináta nullázását jelenti.

Sajnos a nézeti transzformációk nem szögtartók, így az árnyalási számításokat nem végezhetjük a képernyő-koordináta-rendszerben (emlékezzünk vissza az árnyalási egyenletben szereplő $\cos \theta$ -ra és a BRDF-ben szereplő szögekre, amelyeket nem torzíthatunk el). Az árnyaláshoz szükséges szögeket tehát a világ-koordináta-rendszerben kell számolni.

A vágás célja az összes olyan objektumrészlet eltávolítása, amely nem vetülhet az ablakra, vagy amely nem az első és hátsó vágósíkok között van. Párhuzamos vetítés esetén a vágást elvileg bármely koordináta-rendszerben elvégezhetjük, de ez a legkevesebb művelettel a képernyő-koordináta-rendszerben jár. Itt a vágási tartomány egy koordinátatengelyekkel párhuzamos téglalapot, amelyre a 2D vágási algoritmusok (*Cohen-Sutherland vágási algoritmus*, *Sutherland-Hodgeman-poligonvágás*) közvetlen 3D kiterjesztéseivel vághatunk, így ezzel nem is fárasztjuk a tisztelt olvasót. Perspektív vetítés esetén, az *átfordulási probléma* 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 pontjaink a 4D térben találhatóak. A következő szakaszban azt mutatjuk meg, hogy ebben a 4D térben mi a megfelelő vágási tartomány és a 2D vágási algoritmusok milyen apróbb módosításokat igényelnek.

9.5.1. Vágás homogén koordinátákban

A homogén koordináták vágási határokat a képernyő-koordináta-rendszerben megfogalmazott feltételek visszatranszformálásával kaphatjuk meg. A homogén osztás után a vágási határok a következők:

$$X_{\min} = V_x - V_{sx}/2, \quad X_{\max} = V_x + V_{sx}/2, \quad Y_{\min} = V_y - V_{sy}/2, \quad Y_{\max} = V_y + V_{sy}/2.$$

Az ún. belső pontok tehát a következő egyenlőtlenségeket elégítik ki:

$$X_{\min} \leq X_h/h \leq X_{\max}, \quad Y_{\min} \leq Y_h/h \leq Y_{\max}, \quad 0 \leq Z_h/h \leq 1 \quad (9.14)$$

A másik oldalról, a szem előtti tartományok a normalizált nézeti koordináta-rendszerben pozitív Z_c koordinátákkal rendelkeznek, és a perspektív transzformációs mátrixszal való szorzás után a 4. homogén koordináta $h = Z_c$ lesz. Tehát további követelményként megfogalmazzuk a $h > 0$ feltételt. Ekkor viszont beszorozhatjuk a 9.14.

egyenlőtlenségeket h -val, így eljutunk a 4D vágási tartomány definíciójához:

$$X_{\min} \cdot h \leq X_h \leq X_{\max} \cdot h, \quad Y_{\min} \cdot h \leq Y_h \leq Y_{\max} \cdot h, \quad 0 \leq Z_h \leq h. \quad (9.15)$$

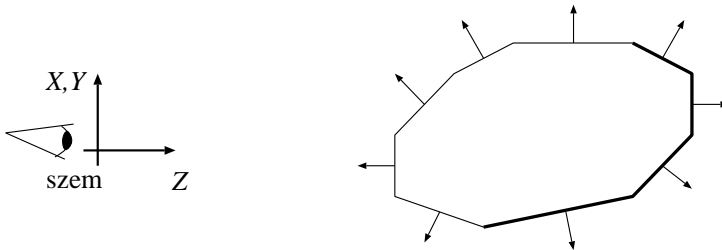
9.6. Takarási feladat megoldása

A *takarási feladatot* a képernyő-koordinátarendszerben oldjuk meg. Pontokra és szakaszokra a probléma triviális, így itt csak a poligonok takarását vizsgáljuk. Gyakran feltételezzük, hogy a poligon háromszög, ami nem jelent különösebb korlátozást, hiszen minden poligon háromszögekre bontható. Feltételezzük továbbá azt is, hogy kívülről nézve a testre a poligonok csúcsainak sorrendje az óramutatóval megegyező bejárású. Ekkor az

$$\vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1) \quad (9.16)$$

formulával minden poligonra kiszámítható egy olyan normálvektor, amely a testből kifelé mutat.

9.6.1. Triviális hátsólap eldobás



9.12. ábra. Normál vektorok és hátsó lapok

A *triviális hátsólap eldobás* azon a felismerésen alapszik, hogy ha a képernyő-koordinátarendszerben egy lap normál vektorának pozitív Z koordinátája van, akkor ez a lap a test hátsó, nem látható oldalán foglal helyet, így eldobandó. 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.

9.6.2. Z-buffer algoritmus

A *z-buffer algoritmus* a takarási feladatot az egyes pixelekre oldja meg, oly módon, hogy minden pixelre megkeresi azt a poligont, amelynek a pixelen keresztül látható pontjának a z koordinátája minimális. A keresés támogatására minden pixelhez, a feldolgozott pillanatának megfelelően tároljuk az abban látható felületi pontok közül a legközelebbi z koordinátáját. Ezt a z értékeket tartalmazó tömböt nevezzük *z-buffernek* vagy *mélység-puffernek*.

A poligonokat egyenként dolgozzuk fel, és meghatározzuk az összes olyan pixelt, amely a poligon vetületén belül van. Ehhez egy 2D poligonkitöltő algoritmust kell végrehajtani. Amint egy pixelhez érünk, kiszámítjuk a felületi pont z koordinátáját és összehasonlítjuk a z -bufferben lévő értékkel. Ha az ott található érték kisebb, akkor a már feldolgozott poligonok között van olyan, amelyik az aktuális poligont ebben a pontban takarja, így az aktuális poligon ezen pontját nem kell megrajzolni. Ha viszont a z -bufferbeli érték nagyobb, akkor az idáig feldolgozott poligonokat az aktuális poligon takarja ebben a pontban, ezért ennek a színét kell beírni az aktuális pixelbe és egyúttal a z értékét a z -bufferbe.

A z -buffer módszer algoritmus a tehát:

```

raszter memória = háttér szín
z-buffer = ∞;
for minden o objektumra do
  for o objektum vetületének minden p pixelére do
    if o objektum p-ben látható pontjának Z koordinátája < zbuffer[p] then
      p színe = o színe ebben a pontban
      zbuffer[p] = o objektum p pixelben látható pontjának Z koordinátája
    endif
  endfor
endfor

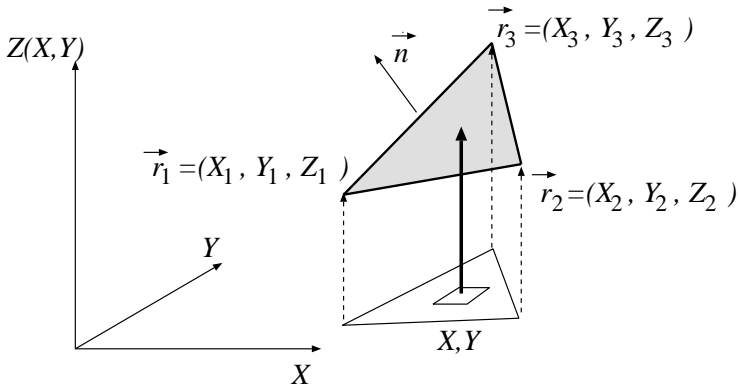
```

A z -buffer ∞ -nel történő inicializálása ténylegesen a lehetséges legnagyobb Z érték használatát jelenti. Az algoritmus részleteinek a bemutatása során feltesszük, hogy az objektumok háromszögek, és aktuálisan a

$$\vec{r}_1 = [X_1, Y_1, Z_1], \quad \vec{r}_2 = [X_2, Y_2, Z_2], \quad \vec{r}_3 = [X_3, Y_3, Z_3]$$

csúcspontokkal definiált háromszöget dolgozzuk fel. A raszterizációs algoritmusnak elő kell állítania a háromszög vetületébe eső X, Y pixel címeket a Z koordinátákkal együtt (9.13. ábra). Az X, Y pixel címből a megfelelő Z koordinátát a háromszög síkjának az egyenletéből származtathatjuk, azaz a Z koordináta az X, Y koordináták valamely lineáris függvénye. A háromszög síkjának az egyenlete:

$$\vec{n} \cdot [X, Y, Z] = C, \quad \text{ahol } \vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1), \quad C = \vec{n} \cdot \vec{r}_1. \quad (9.17)$$



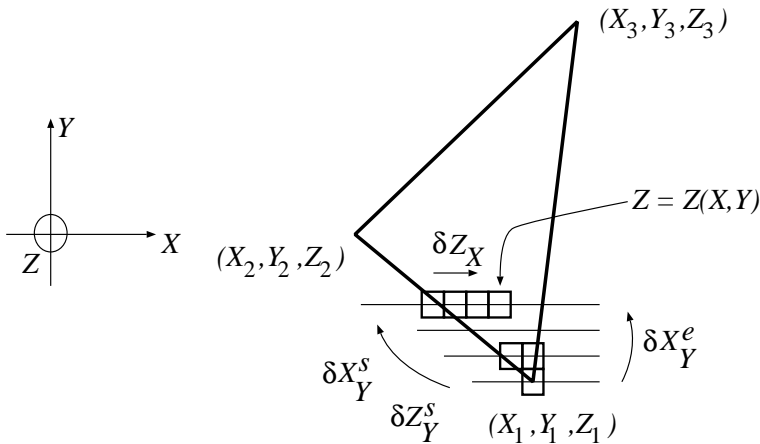
9.13. ábra. Egy háromszög a képernyő-koordinátarendszerben

Ebből a $Z(X, Y)$ függvény:

$$Z(X, Y) = \frac{C - n_X \cdot X - n_Y \cdot Y}{n_Z} \tag{9.18}$$

Az inkrementális elv felhasználásával ezen képlet jelentősen egyszerűsíthető:

$$Z(X + 1, Y) = Z(X, Y) - \frac{n_X}{n_Z} = Z(X, Y) + \delta Z_X \tag{9.19}$$



9.14. ábra. Inkrementális Z érték számítás

Mivel a δZ_X paraméter állandó az egész háromszögre, csak egyszer kell kiszámítani. Egyetlen pásztán belül, a Z koordináta kiszámítása tehát egyetlen összeadást igényel. A határvonalakat a poligonkitöltésnél megismert módon ugyancsak előállíthatjuk egyenesgenerátorok segítségével, sőt a határvonal mentén a pászták kezdeti Z koordinátája is egyetlen összeadással kiszámítható a megelőző pászta kezdeti Z koordinátájából (9.14. ábra). A teljes inkrementális algoritmus, amely a háromszög alsó illetve felső felét tölti ki:

```

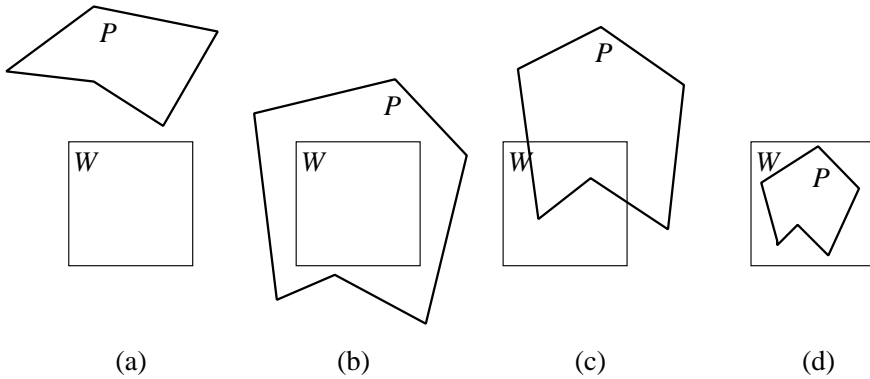
 $X_{\text{start}} = X_1 + 0.5, X_{\text{end}} = X_1 + 0.5, Z_{\text{start}} = Z_1 + 0.5$ 
for  $Y = Y_1$  to  $Y_2$  do
   $Z = Z_{\text{start}}$ 
  for  $X = \text{Trunc}(X_{\text{start}})$  to  $\text{Trunc}(X_{\text{end}})$  do
     $z = \text{Trunc}(Z)$ 
    if  $z < \text{Zbuffer}[X, Y]$  then
       $\text{Pixel}(X, Y, \text{color})$ 
       $\text{Zbuffer}[X, Y] = z$ 
    endif
     $Z += \delta Z_X$ 
  endfor
   $X_{\text{start}} += \delta X_Y^s, X_{\text{end}} += \delta X_Y^e, Z_{\text{start}} += \delta Z_Y^s$ 
endfor

```

Ha a számokat fixpontosan ábrázoljuk, ez az algoritmus egyszerű MSI elemek felhasználásával hardverben is realizálható. A mai korszerű munkaállomások és PC-s 3D grafikus kártyák ilyen egységekkel rendelkeznek.

9.6.3. Területfelosztó módszerek

A különböző felületelemek a képen összefüggő pixeltartományon keresztül látszanak. Ezen koherenciatulajdonság miatt célszerűnek látszik a takarási feladatot pixelnél nagyobb egységekre megoldani. A poligonok és az ablak lehetséges viszonyait a 9.15. ábrán láthatjuk. Ha szerencsénk van, és az objektumtérben csak különálló és körülvevő poligonok vannak, akkor a teljes ablakban vagy egyetlen poligon látszik, vagy egyetlen egy sem. Így a takarási feladatot elegendő egyetlen pixelre megoldani, a láthatóság a többi pixelben is hasonló. Egyetlen pixelre például *sugárkövetéssel* dönthetjük el, hogy abban melyik objektum látszik. Ez a módszer egy félegyeneset (ún. sugarat) indít a szempozícióból a pixel középpontján keresztül az objektumtérbe, meghatározza a félegyenes és a poligonok metszéspontjait, végül azonosítja azt a poligont, amelynek metszéspontja a szemhez a legközelebb van. Az egy pixel elemzésével leküzdhető szerencsés eset akkor áll fenn, amikor egyetlen él sem vetül az ablakra. Ekkor a poligon élek vetületére alkalmazott szakaszvágó algoritmus (7.5. fejezet) úgy találja, hogy a szakasz teljes egészében eldobandó.



9.15. ábra. Poligon-ablak relációk: különálló (a), körülvevő (b), metsző (c), tartalmazott (d)

Ha viszont nem vagyunk ebben a szerencsés helyzetben, akkor az ablakot négy egy-bevágó ablakra bontjuk fel és újra megvizsgáljuk, hogy szerencsénk van-e vagy sem. Az eljárás, amelyet *Warnock-algoritmus*nak neveznek, rekurzíven ismételgeti ezt a lépést, amíg vagy sikerül visszavezetni a takarási feladatot a szerencsés esetre, vagy az ablak mérete a pixel méretére zsugorodik. A pixel méretű ablaknál az újabb felosztások már értelmetlenné válnak, így erre a pixelre már a szokásos módon (például sugárkövetéssel) kell megoldanunk a takarási feladatot.

A módszer algoritmusának leírása során X_1, Y_1 -gyel jelöljük az ablak bal alsó és X_2, Y_2 -vel a jobb felső koordinátáit:

```

Warnock( $X_1, Y_1, X_2, Y_2$ )
  if  $X_1 \neq X_2$  or  $Y_1 \neq Y_2$  then
    if legalább egy él esik az ablakba then
       $X_m = (X_1 + X_2)/2$ 
       $Y_m = (Y_1 + Y_2)/2$ 
      Warnock( $X_1, Y_1, X_m, Y_m$ )
      Warnock( $X_1, Y_m, X_m, Y_2$ )
      Warnock( $X_m, Y_1, X_2, Y_m$ )
      Warnock( $X_m, Y_m, X_2, Y_2$ )
    return ;
  endif
endif
// a  $X_1, Y_1, X_2, Y_2$  téglalap homogén
polygon = a  $(X_1 + X_2)/2, (Y_1 + Y_2)/2$  pixelhez legközelebbi poligon
if nincs poligon then  $X_1, Y_1, X_2, Y_2$  téglalap kitöltése háttér színnel
else
   $X_1, Y_1, X_2, Y_2$  téglalap kitöltése a polygon színével
end
  
```

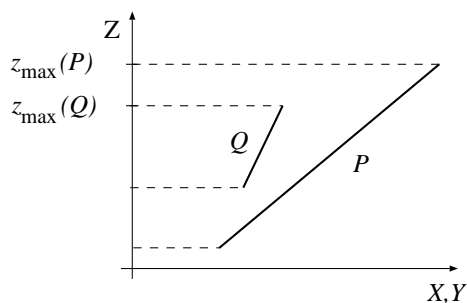
9.6.4. Festő algoritmus

A festés során a későbbi ecsetvonások elfedik a korábbiakat. Ezen egyszerű elv kiaknázásához rendezzük a poligonokat oly módon, hogy egy P poligon csak akkor állhat a sorrendben egy Q poligon után, ha *nem takarja* azt. Majd a kapott sorrendben visszafelé haladva rajzoljuk a poligonokat egymás után a rasztertárba! Ha egynél több poligon vetül egy pixelre, a pixel színe az utoljára rajzolt poligon színével egyezik meg. Mivel a rendezés miatt éppen ez takarja a többit, ezzel a *festő algoritmussal* a takarási feladatot megoldottuk [NNS72].

A poligonok megfelelő rendezése több problémát is felvet, ezért vizsgáljuk meg ezt a kérdést kicsit részletesebben! Azt mondjuk, hogy egy “ P poligon *nem takarja* a Q poligont”, ha P -nek semelyik pontja sem takarja Q valamely pontját. Ezen reláció teljesítéséhez a következő feltételek valamelyikét kell kielégíteni:

1. $z_{\min}(P) > z_{\max}(Q)$ (a P poligon minden pontja hátrébb van a Q poligon bármely pontjánál);
2. a P poligon vetületét befoglaló téglalapnak és a Q poligon vetületét befoglaló téglalapnak nincs közös része;
3. P valamennyi csúcsa (így minden pontja) messzebb van a szemtől, mint a Q síkja;
4. Q valamennyi csúcsa (így minden pontja) közelebb van a szemhez, mint a P síkja;
5. a P és Q vetületeinek nincs közös része.

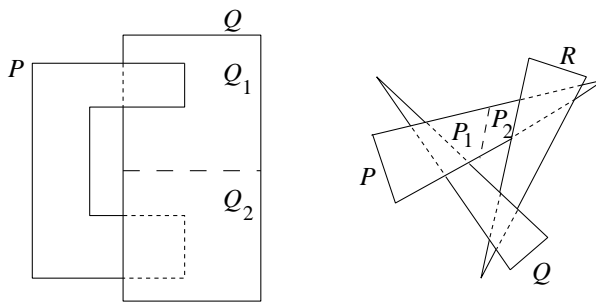
A felsorolt feltételek ellenőrzésének számításigénye a sorrendnek megfelelően nő, ezért az ellenőrzéseket a fenti sorrendben végezzük el.



9.16. ábra. Egy példa, amikor $z_{\max}(P) > z_{\max}(Q)$, de P takarja Q -t

Első lépésként rendezzük a poligonokat a maximális z koordinátájuk szerint úgy, hogy a közeli poligonok a lista elején, a távoli poligonok pedig a lista végén foglaljanak helyet. Ez önmagában még nem elég, hiszen előfordulhat, hogy az így kapott listában valahol borul a “ P poligon *nem takarja* a Q poligont” reláció (9.16. ábra).

Ezért minden egyes poligont össze kell vetni valamennyi, a listában előtte álló poligonnal, és ellenőrizni kell a megadott feltételeket. Ha azok valamelyike minden előbb álló poligonra teljesül, akkor az adott poligon helye megfelelő. Ha viszont a poligonunk takarja valamelyik előbb álló poligont, akkor a takart poligont az aktuális poligon mögé kell vinni a listában, és a mozgatott poligonra visszalépve újra kell kezdeni a feltételek ellenőrzését.



9.17. ábra. Ciklikus takarás

Előfordulhat, hogy ez az algoritmus mókuserékbe kerül, amiből képtelen szabadulni. Például ha két poligon egymást takarja (9.17. ábra bal oldala), az ismert algoritmus ezen két poligon végtelenített cserélgetésébe torkollik. Még gonoszabb esetet mutat be ugyanezen ábra jobb oldala, amikor kettőnél több poligon ciklikus takarásának lehetünk tanúi.

Ezeket a ciklikus átlapolásokat a poligonok megfelelő vágásával oldhatjuk fel, és ezáltal átsegíthetjük az algoritmusunkat a kritikus pontokon. A ciklikus átlapolások felismeréséhez a mozgatáskor a poligonokat megjelöljük. Ha még egyszer mozgatni kell őket, akkor valószínűsíthető, hogy ennek oka a ciklikus átlapolás. Ekkor az újból mozgatott poligont a másik poligon síkja mentén két részre vágjuk.

9.7. Lokális illuminációs algoritmusok

A takarási algoritmusok minden pixelre meghatározzák az ott látható poligont. A hátralévő feladat az adott pixelben látható felületi pont színének kiszámítása. Az árnyalási egyenlet szerint ehhez az adott pontból látható többi felületet kell számba venni, és azok

fényességéből (radianciájából) következtetni az adott pont fényességére. Ez persze a róka fogta csuka esete, mert a többi pont fényessége viszont az adott pont fényességétől függ, azaz a különböző pontok fényességei között kölcsönös csatolás van.

A lokális illuminációs algoritmusok az árnyalási egyenlet drasztikus egyszerűsítésével kiküszöbölnék mindenféle csatolást, azaz egy felület radianciájának meghatározásához nem veszik figyelembe a többi felület fényességét. Megvilágítás csak a képen közvetlenül nem látható absztrakt fényforrásokból érkezik. A csatolás megszüntetésével az árnyalási egyenletben az integrálból eltűnik az ismeretlen függvény, így az integrálegyenlet megoldása helyett csupán egy egyszerű integrált kell kiértékelnünk.

A feladat tovább egyszerűsíthető, ha a lokális megvilágítás számításánál nem csupán a többi tárgy fényességét nem vesszük figyelembe, hanem azok geometriai elhelyezkedését sem, és az absztrakt fényforrásokat minden pontból láthatóként kezeljük. Ezzel a lépéssel lényegében az *árnyékok* számítását takarítjuk meg [Cro77].

A fizikai modell durva egyszerűsítése természetesen a valóságghűség csorbulásához vezet. Cserébe viszont lényegesen könnyebben kaphatunk képeket, sőt megfelelő hardvertámogatás birtokában, akár valós időben futó animációkat is készíthetünk.

Az elmondott egyszerűsítések következtében a 8.15 árnyalási egyenlet a következő *illuminációs képlettel* helyettesíthető:

$$L(x, \omega) = L^e(x, \omega) + k_a \cdot L^a + \sum_l f_r(\omega'_l, \vec{x}, \omega) \cdot \cos \theta'_l \cdot L^{\text{in}}(\vec{x}, \omega'_l), \quad (9.20)$$

ahol $L^{\text{in}}(\vec{x}, \omega'_l)$ az l . absztrakt fényforrásból az \vec{x} pontba az ω'_l irányból érkező radiancia.

A $k_a \cdot L^a$ egy új tényező, amit *ambiens tagnak* nevezünk. Az ambiens tag feladata az, hogy az egyéb elhanyagolásokat kompenzálja. Az ambiens megvilágítást a tér minden pontjában és irányában állandónak tekintjük. A felületi pont az ambiens fény k_a -szorosát veri vissza.

Mivel az illuminációs képletben szögek is találhatóak, és a képernyő-koordinátarendszerbe vezető transzformációk között nem szögtartók is vannak, az illuminációs képletet a világ-koordinátarendszerben kell kiértékelni.

Miként a takarási feladatnál is láttuk, 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 paramétereit, 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 technikákat ismertetünk.

9.7.1. Saját színnel történő árnyalás

A *saját színnel történő árnyalás* a 2D képszintézis árnyalási módszerének direkt alkalmazása. Előnye, hogy nem igényel semmiféle illuminációs számítást, viszont a keletkezett képeknek sincs igazán 3D hatásuk (17.16. ábra).

9.7.2. Konstans árnyalás

A *konstans árnyalás* a poligonokra csak egyszer számítja ki az absztrakt fényforrások hatását. Amennyiben valamelyik pixelben a poligon látszik, akkor mindig ezt a konstans színt írja a rasztertárba. Az eredmény általában elég lesújtó, mert a képről ordít, hogy a felületeket sík poligonokkal közelítettük (17.16. ábra).

9.7.3. Gouraud-árnyalás

A *Gouraud-árnyalás* a háromszögek csúcspontjaiban értékeli ki a fényforrásokból idejutó 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, amit a poligonháló visz magával a transzformációk során. Ezután a Gouraud-árnyalás a háromszög belső pontjainak színét a csúcspontok színéből lineárisan interpolálja. Vegyük észre, hogy ez pontosan ugyanaz az algoritmus, ahogyan a z mélység koordinátát a háromszög három csúcspontjából lineáris interpolációval határozzuk meg, így az ott említett inkrementális módszer itt is használható!

A Gouraud-árnyalás programja, amely egy háromszög alsó felét színezi ki:

```

 $X_{\text{start}} = X_1 + 0.5, \quad X_{\text{end}} = X_1 + 0.5$ 
 $R_{\text{start}} = R_1 + 0.5, \quad G_{\text{start}} = G_1 + 0.5, \quad B_{\text{start}} = B_1 + 0.5$ 
for  $Y = Y_1$  to  $Y_2$  do
     $R = R_{\text{start}}, \quad G = G_{\text{start}}, \quad B = B_{\text{start}}$ 
    for  $X = \text{Trunc}(X_{\text{start}})$  to  $\text{Trunc}(X_{\text{end}})$  do
         $\text{Pixel}(X, Y, \text{Trunc}(R), \text{Trunc}(G), \text{Trunc}(B))$ 
         $R += \delta R_X, \quad G += \delta G_X, \quad B += \delta B_X$ 
    endfor
     $X_{\text{start}} += \delta X_Y^s, \quad X_{\text{end}} += \delta X_Y^e$ 
     $R_{\text{start}} += \delta R_Y^s, \quad G_{\text{start}} += \delta G_Y^s, \quad B_{\text{start}} += \delta B_Y^s$ 
endfor

```

A Gouraud-árnyalás akkor jó, ha a háromszögon belül a szín valóban közelítőleg lineárisan változik. Ez nagyjából igaz diffúz visszaverődésű objektumokra, de elfogadhatatlan tükrös illetve spekuláris visszaverődésű felületekre. A lineáris interpoláció ilyen esetben egyszerűen kihagyhatja vagy szétkenheti a fényforrás tükröződő foltját (17.16. ábra).

9.7.4. Phong-árnyalás

A *Phong-árnyalás* az illuminációs képletben felhasznált normálvektort interpolálja a háromszög csúcspontjaiban érvényes normálvektorokból. Az illuminációs képletet pedig minden pixelre külön határozza meg.

A Phong-árnyalás a színtérben nemlineáris interpolációnak felel meg, így nagyobb poligonokra is megbirkózik a tükrös felületek gyorsan változó radianciájával (17.16. ábra). A Phong-árnyalás programja:

```

 $X_{start} = X_1 + 0.5, X_{end} = X_1 + 0.5, \vec{N}_{start} = \vec{N}_1$ 
for  $Y = Y_1$  to  $Y_2$  do
   $\vec{N} = \vec{N}_{start}$ 
  for  $X = \text{Trunc}(X_{start})$  to  $\text{Trunc}(X_{end})$  do
     $(R, G, B) = \text{ShadingModel}(\vec{N})$ 
     $\text{Pixel}(X, Y, \text{Trunc}(R), \text{Trunc}(G), \text{Trunc}(B))$ 
     $\vec{N} += \delta\vec{N}_X$ 
  endfor
   $X_{start} += \delta X_Y^s, X_{end} += \delta X_Y^e, \vec{N}_{start} += \delta\vec{N}_Y^s$ 
endfor

```

A Phong-árnyalás a Gouraud-árnyalás olyan határeseteként is elképzelhető, amikor a tesszelláció finomításával poligonok vetített területe a pixelek méretével összevethető.

9.8. Program: 3D grafikus rendszer inkrementális képszintézissel

A 3D világmodellben primitívek (`Primitive3D`) szerepelnek, amelyeket kiegészítünk tesszellációs függvényekkel (`Tessellate`). A képszintézis során a modell primitívjeit tesszelláljuk, azaz az általános görbéket és felületeket szakaszokkal és háromszögekkel közelítjük. A tesszelláció eredménye egy `RenderPrimitive3D` típusú objektum.

```

//=====
class Primitive3D : public Emitter {
//=====
    Array<Point3D> points;
public:
    Primitive3D( Color c = 0, int n = 0 ) : points(n), Emitter( c ) { }
    Point3D& Point( int i ) { return points[i]; }
    int PointNum( ) { return points.Size(); }
    virtual RenderPrimitive3D * Tessellate( ) { return NULL; }
};

```



```

for(int i = 0, t = 0; i < NFI; i++) {
    for( int j = 0; j < NTETA; j++ ) {
        double dfi = 2.0*M_PI/NFI, dte = M_PI/NTETA;

        double x1 = sin(i*dfi) * sin(j*dte);
        double y1 = -cos(i*dfi) * sin(j*dte);
        double z1 = cos(j*dteta);

        double x2 = sin(i*dfi) * sin((j+1)*dteta);
        double y2 = -cos(i*dfi) * sin((j+1)*dteta);
        double z2 = cos((j+1)*dteta);

        int ii = (i+1 == NFI) ? 0 : i+1;
        double x3 = sin(ii*dfi) * sin(j*dteta);
        double y3 = -cos(ii*dfi) * sin(j*dteta);

        double x4 = sin(ii*dfi) * sin((j+1)*dteta);
        double y4 = -cos(ii*dfi) * sin((j+1)*dteta);

        Vector3D n1(x1, y1, z1), n2(x2, y2, z2),
                n3(x3, y3, z1), n4(x4, y4, z2);
        Point3D p1 = Point(0) + n1 * R,
                p2 = Point(0) + n2 * R,
                p3 = Point(0) + n3 * R,
                p4 = Point(0) + n4 * R;

        if (j == 0)
            p -> AddTriangle( t++, p2, p4, p1, n2, n4, n1 );
        else if (j == NTETA-1)
            p -> AddTriangle( t++, p1, p2, p3, n1, n2, n3 );
        else {
            p -> AddTriangle( t++, p1, p2, p3, n1, n2, n3 );
            p -> AddTriangle( t++, p2, p4, p3, n2, n4, n3 );
        }
    }
}
return p;
};

```

A `DiffuseSpecularMaterial` osztály olyan anyagokat képes modellezni, amelyek részben diffúz, részben spekuláris jelleg szerint verik vissza a rájuk eső fényt.

```
//=====
class DiffuseSpecularMaterial : public DiffuseMaterial,
                               public SpecularMaterial {
//=====
public:
    DiffuseSpecularMaterial( SColor kd0, SColor ks0, double shine0)
        : DiffuseMaterial(kd0), SpecularMaterial(ks0, shine0) { }
    SColor BRDF( Vector3D& L, Vector3D& N, Vector3D& V ) {
        return ( DiffuseMaterial :: BRDF(L, N, V) +
                SpecularMaterial :: BRDF(L, N, V));
    }
};
```

A tesszelláció eredményeként kapott `RenderPrimitive3D` típusú objektumokat már transzformálhatjuk (`Transform`), vágthatjuk és raszterizálhatjuk (`Draw`), sőt a fényforrások és a szempozíció ismeretében a megvilágításukat is meghatározhatjuk. A vágást két lépésben végezzük el. A `DepthClip` tagfüggvény az első és hátsó vágósíkra vág, és mivel ezt a lépést a homogén osztás előtt kell végrehajtani, homogén koordinátákkal dolgozik. A `Clip` tagfüggvény pedig a homogén osztás után a nézet téglalapjára vág. A primitív árnyalása a világkoordinátarendszerben történik, ezért a normálvektorokat a `TransformNormals` tagfüggvénnyel kell transzformálni. Az illuminációs képleteket az `Illuminate` tagfüggvényben számítjuk.

```
//=====
class RenderPrimitive3D : public Emitter {
//=====
protected:
    Array<HomPoint3D> tpoints;
public:
    RenderPrimitive3D( Color c = 0, int n = 0 )
        : tpoints(n), Emitter( c ) { }
    HomPoint3D& Point( int i ) { return tpoints[i]; }
    int PointNum( ) { return tpoints.Size(); }
    void Transform( Transform3D tr ) {
        for(int i = 0; i < PointNum(); i++)
            Point(i) = tr.Transform( Point(i) );
    }
    void HomDivPoints( ) {
        for(int i = 0; i < PointNum(); i++)
            Point(i) = Point(i).HomDiv();
    }
    virtual void TransformNormals( Transform3D tr ) { }
    virtual BOOL DepthClip( ) = 0;
    virtual BOOL Clip( RectAngle& cliprect ) = 0;
    virtual void Illuminate( Lights& l, Color& La, Point3D& eye ) { };
    virtual void Draw( Window * scr ) = 0;
};
```

A Line3D a legegyszerűbb olyan primitív, amelyre a képszintézis ténylegesen elvégezhető.

```
//=====
class Line3D : public RenderPrimitive3D {
//=====
public:
    Line3D( HomPoint3D& v1, HomPoint3D& v2, Color c )
        : RenderPrimitive3D( c, 2 ) { Point(0) = v1; Point(1) = v2; }
    BOOL    DepthClip( );
    BOOL    Clip( RectAngle& viewport );
    void    Draw( Window * scr );
};
```

A 3D szakaszok vágására a *Cohen-Sutherland vágási algoritmust* használhatjuk.

```
//-----
BOOL Line3D :: DepthClip( ) { // Cohen-Sutherland
//-----
    HomPoint3D& p1 = Point(0);
    HomPoint3D& p2 = Point(1);

    for( ; ; ) {
        int c1 = (p1.Z() < 0) | ((p1.Z() > p1.h()) << 1);
        int c2 = (p2.Z() < 0) | ((p2.Z() > p2.h()) << 1);

        if (c1 == 0 && c2 == 0) return TRUE;
        if ( (c1 & c2) != 0 ) return FALSE;

        int f = ( (c1 & 1) != (c2 & 1) ) ? 1 : 2;

        HomPoint3D pi;
        double ti;

        switch ( f ) {
        case 1:
            ti = (0 - p1.Z()) / (p2.Z() - p1.Z());
            pi = p1 + (p2 - p1) * ti;
            break;
        case 2:
            ti = (p1.h()-p1.Z()) / (p2.Z()-p1.Z()-p2.h()+p1.h());
            pi = p1 + (p2 - p1) * ti;
        }
        if (c1 & f) { p1 = pi; }
        else      { p2 = pi; }
    }
}
```

```
//-----
BOOL Line3D :: Clip( RectAngle& cliprect ) { // Cohen-Sutherland
//-----
    HomPoint3D& p1 = Point(0);
    HomPoint3D& p2 = Point(1);

    for( ; ; ) {
        int c1 = cliprect.Code( Point2D(p1.X(), p1.Y()) );
        int c2 = cliprect.Code( Point2D(p2.X(), p2.Y()) );

        if (c1 == 0 && c2 == 0) return TRUE;
        if ( (c1 & c2) != 0 ) return FALSE;

        int f;
        if ( (c1 & 1) != (c2 & 1) ) f = 1;
        else if ( (c1 & 2) != (c2 & 2) ) f = 2;
        else if ( (c1 & 4) != (c2 & 4) ) f = 4;
        else f = 8;

        double dx = p2.X() - p1.X();
        double dy = p2.Y() - p1.Y();

        HomPoint3D pi;
        switch ( f ) {
            case 1: pi = p1 + (p2 - p1) * (cliprect.Left() - p1.X()) / dx;
                    break;
            case 2: pi = p1 + (p2 - p1) * (cliprect.Right() - p1.X()) / dx;
                    break;
            case 4: pi = p1 + (p2 - p1) * (cliprect.Bottom() - p1.Y()) / dy;
                    break;
            case 8: pi = p1 + (p2 - p1) * (cliprect.Top() - p1.Y()) / dy;
                    }
            if (c1 & f) { p1 = pi; }
            else { p2 = pi; }
        }
    }
}
```

Egy 3D szakasz raszterizálása a 2D vetületének felrajzolását jelenti. A rajzolási színek az objektum saját színét választjuk.

```
//-----
void Line3D :: Draw( Window * scr ) {
//-----
    scr -> SetColor( Le() );
    scr -> Move(Point(0).X(), Point(0).Y() );
    scr -> DrawLine(Point(1).X(), Point(1).Y() );
    return;
}
```


A háromszöglista felületek tesszellációjából keletkezik. Az örökölt `points` tömb most a háromszögek csúcspontjait tartalmazza. A csúcspontokban érvényes normálvektorokat a `normals` tömbben tároljuk. A két `AddTriangle` tagfüggvény abban tér el egymástól, hogy az első a három csúcspontban a háromszög tényleges normálvektorát használja, a második pedig a kapott normálvektorokat. Az illuminációs képletet az `Illuminate` függvényben számítjuk a `DiffuseSpecularMaterial`-tól örökölt módszerrel és az ambiens visszaverődési tényező (k_a) felhasználásával. A számított színek a `colors` tömbbe kerülnek.

```
//=====
class TriangleList3D : public RenderPrimitive3D,
                      public DiffuseSpecularMaterial {
//=====
    Array< Vector3D > normals;
    Array< Color >     colors;
    SColor             ka;
    Point3D& N1(int i) { return normals[3 * i]; }
    Point3D& N2(int i) { return normals[3 * i + 1]; }
    Point3D& N3(int i) { return normals[3 * i + 2]; }
public:
    TriangleList3D( Color e, Color ka0, Color kd0,
                  Color ks0, double shine, int n = 0 )
: RenderPrimitive3D(e, 3 * n), ka(ka0),
  DiffuseSpecularMaterial(kd0, ks0, shine),
  normals(3 * n), colors(3 * n) { }
    void AddTriangle( int i, Point3D p1, Point3D p2, Point3D p3 ) {
        P1(i) = p1; P2(i) = p2; P3(i) = p3;
        Vector3D normal = ((p2 - p1) % (p3 - p1));
        normal.Normalize(); N1(i) = N2(i) = N3(i) = normal;
    }
    void AddTriangle( int i, Point3D p1, Point3D p2, Point3D p3,
                    Vector3D n1, Vector3D n2, Vector3D n3) {
        P1(i) = p1; P2(i) = p2; P3(i) = p3;
        N1(i) = n1; N2(i) = n2; N3(i) = n3;
    }
    int TriangleNum( ) { return PointNum() / 3; }
    HomPoint3D& P1(int i) { return Point(3 * i); }
    HomPoint3D& P2(int i) { return Point(3 * i + 1); }
    HomPoint3D& P3(int i) { return Point(3 * i + 2); }
    void TransformNormals( Transform3D tr ) {
        for(int i = 0; i < normals.Size(); i++)
            normals[i] = tr.Transform( normals[i] ).HomDiv();
    }
    BOOL DepthClip( );
    BOOL Clip( RectAngle& cliprect );
    void Illuminate(Lights& lights, Color& La, Point3D& eye);
    void Draw( Window * scr );
};
```

A háromszögeket a *Sutherland-Hodgeman-poligonvágással* vághatjuk. A mélységi vágást homogénkoordinátákban, a nézeti vágást pedig Descartes-koordinátákban kell végrehajtani. Itt csak a mélységi vágást adjuk közre, de a CD mellékletben a nézeti vágás is megtalálható.

```
//-----
BOOL TriangleList3D :: DepthClip( ) { // Sutherland-Hodgeman
//-----
    Array<HomPoint3D> clippoints( PointNum() );
    Array<Color> clipcolors( PointNum() );
    const double Zmin = 0, Zmax = 1;

    for( int t = 0, clipped = 0; t < PointNum() / 3 ; t++ ) {
        HomPoint3D q1[8], q2[8];
        Color c1[8], c2[8];
        for( int j = 0; j < 3; j++ ) {
            c1[j] = colors[3*t + j]; q1[j] = Point(3*t + j);
        }

        int n = 3, m = 0, i;
        for(i = 0; i < n; i++) {
            int i1 = (i+1) % n;
            double z1 = q1[i].Z(), z2 = q1[i1].Z();
            double h1 = q1[i].h(), h2 = q1[i1].h(), ti;

            if (z1 >= Zmin * h1) {
                c2[m] = c1[i]; q2[m++] = q1[i];
                if (z2 < Zmin * h2) {
                    ti = (z1 - Zmin*h1) / (Zmin*(h2 - h1) - (z2 - z1));
                    c2[m] = c1[i] + (c1[i1] - c1[i]) * ti;
                    q2[m++] = q1[i] + (q1[i1] - q1[i]) * ti;
                }
            } else {
                if (z2 >= Zmin * h2) {
                    ti = (z1 - Zmin*h1) / (Zmin*(h2 - h1) - (z2 - z1));
                    c2[m] = c1[i] + (c1[i1] - c1[i]) * ti;
                    q2[m++] = q1[i] + (q1[i1] - q1[i]) * ti;
                }
            }
        }
    }

    n = m; m = 0;
    for(i = 0; i < n; i++) {
        int i1 = (i+1) % n;
        double z1 = q2[i].Z(), z2 = q2[i1].Z();
        double h1 = q2[i].h(), h2 = q2[i1].h(), ti;

        if (z1 <= Zmax * h1) {
```

```

        c1[m] = c2[i]; q1[m++] = q2[i];
        if (z2 > Zmax * h2) {
            ti = (z1 - Zmax*h1) / (Zmax*(h2 - h1) - (z2 - z1));
            c1[m] = c2[i] + (c2[i1] - c2[i]) * ti;
            q1[m++] = q2[i] + (q2[i1] - q2[i]) * ti;
        }
    } else {
        if (z2 <= Zmax * h2) {
            ti = (z1 - Zmax*h1) / (Zmax*(h2 - h1) - (z2 - z1));
            c1[m] = c2[i] + (c2[i1] - c2[i]) * ti;
            q1[m++] = q2[i] + (q2[i1] - q2[i]) * ti;
        }
    }
}
for(i = 1; i < m-1; i++) {
    clipcolors[clipped]=c1[0];    clippoints[clipped++]=q1[0];
    clipcolors[clipped]=c1[i];    clippoints[clipped++]=q1[i];
    clipcolors[clipped]=c1[i+1]; clippoints[clipped++]=q1[i+1];
}
}

if (clipped == 0) return FALSE;
else { tpoints = clippoints; colors = clipcolors; return TRUE; }
}

```

A háromszöglista árnyalásához a háromszögek csúcspontjaiban értékeljük ki az illuminációs képletet.

```

//-----
void TriangleList3D :: Illuminate( Lights& lights,
                                  Color& La,
                                  Point3D& eye) {
//-----
    for( int i = 0; i < PointNum(); i++ ) {
        Point3D x = Point(i);
        colors[i] = Le() + ka * La;
        for(int l = 0; l < lights.Size(); l++) {
            Vector3D L = lights[l] -> LightDir( x );
            Vector3D N = normals[i];
            double cost = N * L;
            if (cost > 0) {
                Vector3D V = eye - x; V.Normalize();
                colors[i] += BRDF(L,N,V) * lights[l]->Le(x,-L) * cost;
            }
        }
    }
}
}

```

A rajzoláshoz a csúcspontok Descartes-koordinátáit és a kiszámított színeket az ablak 3D háromszögrajzoló függvényének (`DrawTriangle`) adjuk át.

```
//-----
void TriangleList3D :: Draw( Window * scr ) {
//-----
    for( int t = 0; t < TriangleNum(); t++ ) {
        Coord x[3], y[3], z[3];
        double r[3], g[3], b[3];
        for( int i = 0; i < 3; i++ ) {
            x[i] = Point(3*t + i).X();
            y[i] = Point(3*t + i).Y();
            z[i] = Point(3*t + i).Z();
            r[i] = colors[3*t + i].Red();
            g[i] = colors[3*t + i].Green();
            b[i] = colors[3*t + i].Blue();
        }
        scr -> DrawTriangle( x, y, z, r, g, b );
    }
}
```

Az ablak 3D háromszögrajzoló függvénye elvégzi a koordináták és a színek logikai-fizikai konverzióját és továbbadja a háromszöget a fizikai szintű `PFacet` függvénynek, amely *z-buffer*-t használ a takarási feladat megoldásához és *Gouraud-árnyalás*-t a háromszög belső pontjaiban a szín előállításához.

```
//-----
void Window :: DrawTriangle( Coord x[3], Coord y[3], Coord z[3],
                             double r[3], double g[3], double b[3] ) {
//-----
    PVertex p[3];
    for(int i = 0; i < 3; i++) {
        if (r[i] > 1.0) r[i] = 1.0;
        if (g[i] > 1.0) g[i] = 1.0;
        if (b[i] > 1.0) b[i] = 1.0;
        Logical2PhysicalCoord( x[i], y[i], p[i].x, p[i].y );
        p[i].z = z[i]*MAXZ;
        p[i].R = r[i]*255; p[i].G = g[i]*255; p[i].B = b[i]*255;
    }
    PFacet( p );      // z-buffer / Gouraud
}
```

A ZBuffer osztály a z-buffer kezelését valósítja meg.

```
typedef unsigned char ZCoord;

//=====
class ZBuffer {
//=====
    ZCoord ** z_buffer;
    int XMAX, YMAX;
public:
    void Initialize( PCoord xmax, PCoord ymax ) {
        XMAX = xmax; YMAX = ymax;
        z_buffer = new ZCoord*[YMAX+1];
        for(int y=0; y<=YMAX; y++) z_buffer[y] = new ZCoord[XMAX+1];
        Clear();
    }
    ZCoord Get(int x, int y) { return z_buffer[y][x]; }
    void Set(int x, int y, ZCoord z) {
        if (x>=0 && x<=XMAX && y>=0 && y<=YMAX) z_buffer[y][x] = z;
    }
    void Clear( ) {
        for(int y = 0; y <= YMAX; y++)
            for(int x = 0; x <= XMAX; x++) z_buffer[y][x] = MAXZ;
    }
};

ZBuffer zbuffer;
```

A PFacet függvény egy 3D háromszöget jelenít meg *z-buffer* takarási algoritmus-sal és *Gouraud-árnyalás* alkalmazásával. A sebességi igények miatt (és a hardver implementáció illusztrálása céljából) a rutin csak egész műveleteket használ. Ehhez a nem egész mennyiségeket fixpontosan ábrázolja, ahol a felhasznált egész változó alsó NFRACHT bitje a törtrészt, a többi bit pedig az egészrészt jelképezi. A FIXP makro egy számot normál ábrázolásból fixpontos ábrázolásúra, a TRUNC pedig fixpontosról normál ábrázolásúra alakít át.

```
#define NFRACHT 12
#define TRUNC(x) ((x) >> NFRACHT)
#define FIXP(x) ((long)(x) << NFRACHT)
#define HALF ((long)1 << (NFRACHT-1))

//-----
void PFacet( PVertex p[3] ) {
//-----
    PVertex p0, p1, p2;
    if (p[0].y<=p[1].y && p[1].y<=p[2].y) { p0=p[0]; p1=p[1]; p2=p[2]; }
```

```

else
if (p[1].y<=p[2].y && p[2].y<=p[0].y) { p0=p[1]; p1=p[2]; p2=p[0]; }
else
if (p[2].y<=p[0].y && p[0].y<=p[1].y) { p0=p[2]; p1=p[0]; p2=p[1]; }
else
if (p[0].y<=p[2].y && p[2].y<=p[1].y) { p0=p[0]; p1=p[2]; p2=p[1]; }
else
if (p[1].y<=p[0].y && p[0].y<=p[2].y) { p0=p[1]; p1=p[0]; p2=p[2]; }
else
if (p[2].y<=p[1].y && p[1].y<=p[0].y) { p0=p[2]; p1=p[1]; p2=p[0]; }

long nz = (long)(p1.x - p0.x) * (p2.y - p0.y) -
           (long)(p2.x - p0.x) * (p1.y - p0.y);
if (nz == 0) return;

int Dy10 = p1.y - p0.y, Dy20 = p2.y - p0.y, Dy21 = p2.y - p1.y;
long
Dx10=FIXP(p1.x-p0.x), Dx20=FIXP(p2.x-p0.x), Dx21=FIXP(p2.x-p1.x),
Dz10=FIXP(p1.z-p0.z), Dz20=FIXP(p2.z-p0.z), Dz21=FIXP(p2.z-p1.z),
DR10=FIXP(p1.R-p0.R), DR20=FIXP(p2.R-p0.R), DR21=FIXP(p2.R-p1.R),
DG10=FIXP(p1.G-p0.G), DG20=FIXP(p2.G-p0.G), DG21=FIXP(p2.G-p1.G),
DB10=FIXP(p1.B-p0.B), DB20=FIXP(p2.B-p0.B), DB21=FIXP(p2.B-p1.B);
long dx02, dx01, dx12, dz02, dz01, dz12;
long dR02, dR01, dR12, dG02, dG01, dG12, dB02, dB01, dB12;

if (Dy20 != 0) {
    dx02 = Dx20/Dy20; dz02 = Dz20/Dy20;
    dR02 = DR20/Dy20; dG02 = DG20/Dy20; dB02 = DB20/Dy20;
}
if (Dy10 != 0) {
    dx01 = Dx10/Dy10; dz01 = Dz10/Dy10;
    dR01 = DR10/Dy10; dG01 = DG10/Dy10; dB01 = DB10/Dy10;
}
if (Dy21 != 0) {
    dx12 = Dx21/Dy21; dz12 = Dz21/Dy21;
    dR12 = DR21/Dy21; dG12 = DG21/Dy21; dB12 = DB21/Dy21;
}
int left = nz < 0;

long nzx = (long)(p1.y - p0.y) * (p2.z - p0.z) -
           (long)(p2.y - p0.y) * (p1.z - p0.z);
long nRx = (long)(p1.y - p0.y) * (p2.R - p0.R) -
           (long)(p2.y - p0.y) * (p1.R - p0.R);
long nGx = (long)(p1.y - p0.y) * (p2.G - p0.G) -
           (long)(p2.y - p0.y) * (p1.G - p0.G);
long nBx = (long)(p1.y - p0.y) * (p2.B - p0.B) -
           (long)(p2.y - p0.y) * (p1.B - p0.B);

long dz_x= -FIXP(nzx)/nz;

```

```

long dR_x= -FIXP(nRx)/nz, dG_x= -FIXP(nGx)/nz, dB_x= -FIXP(nBx)/nz;
long dxS, dxE, dzS, dRs, dGs, dBs, z, R, G, B;

if( Dy10 > 0 ) {
    if ( left ) { dxS = dx01; dxE = dx02; dzS = dz01;
                 dRs = dR01; dGs = dG01; dBs = dB01; }
    else        { dxS = dx02; dxE = dx01; dzS = dz02;
                 dRs = dR02; dGs = dG02; dBs = dB02; }
    long xs=FIXP(p0.x)+HALF, xe=FIXP(p0.x)+HALF, zs=FIXP(p0.z)+HALF;
    long Rs=FIXP(p0.R)+HALF, Gs=FIXP(p0.G)+HALF, Bs=FIXP(p0.B)+HALF;

    for(int y = p0.y; y <= p1.y; y++) {
        z = zs, R = Rs, G = Gs, B = Bs;
        for(int x = TRUNC(xs); x <= TRUNC(xe); x++) {
            int Z = TRUNC(z);
            if ( Z <= zbuffer.Get(x, y) ) {
                zbuffer.Set(x, y, Z);
                Pixel(x, y, TRUNC(R), TRUNC(G), TRUNC(B));
            }
            z += dz_x; R += dR_x; G += dG_x; B += dB_x;
        }
        xs += dxS; xe += dxE; zs += dzS;
        Rs += dRs; Gs += dGs; Bs += dBs;
    }
}

if (Dy21 > 0){
    if (left) { dxS = -dx12; dxE = -dx02; dzS = -dz12;
                 dRs = -dR12; dGs = -dG12; dBs = -dB12; }
    else      { dxS = -dx02; dxE = -dx12; dzS = -dz02;
                 dRs = -dR02; dGs = -dG02; dBs = -dB02; }
    long xs=FIXP(p2.x)+HALF, xe=FIXP(p2.x)+HALF, zs=FIXP(p2.z)+HALF;
    long Rs=FIXP(p2.R)+HALF, Gs=FIXP(p2.G)+HALF, Bs=FIXP(p2.B)+HALF;

    for(int y = p2.y; y >= p1.y; y--) {
        z = zs, R = Rs, G = Gs, B = Bs;
        for(int x = TRUNC(xs); x <= TRUNC(xe); x++) {
            int Z = TRUNC(z);
            if ( Z <= zbuffer.Get(x, y) ) {
                zbuffer.Set(x, y, Z);
                Pixel(x, y, TRUNC(R), TRUNC(G), TRUNC(B));
            }
            z += dz_x; R += dR_x; G += dG_x; B += dB_x;
        }
        xs += dxS; xe += dxE; zs += dzS;
        Rs += dRs; Gs += dGs; Bs += dBs;
    }
}
}

```

A kameraosztály tartalmazza a nézeti transzformáció számításához szükséges adatokat.

```
enum ProjType { PARALLEL, PERSPECTIVE };

//=====
class Camera3D {
//=====
    Point3D vrp;
    Vector3D vpn, vup, eye, world_eye;
    Coord fp, bp;
    RectAngle window, viewport;
    Transform3D transf;
    ProjType projtype;

    void CalcTransf( );
public:
    Camera3D( )
    : vrp(0,0,0), vpn(0,0,-1), vup(0,1,0), eye(0,0,-1),
      window(-1,-1,1,1), viewport(0, 0, 1, 1) {
        fp = 0.5, bp = 1.5, projtype = PERSPECTIVE;
        CalcTransf( );
    }
    Vector3D GetWorldEye( ) { return world_eye; }
    RectAngle Viewport( ) { return viewport; }
    Transform3D ViewTransform( ) { return transf; }
};
```

A CalcTransf a kameraparaméterek alapján meghatározza a nézeti transzformációs mátrixot.

```
//-----
void Camera3D :: CalcTransf( ) {
//-----
    Vector3D w = vpn; w.Normalize( );
    Vector3D u = w % vup; u.Normalize( );
    Vector3D v = u % w;
    Transform3D Tuvw( AFFIN, u, v, w, vrp );

    world_eye = Tuvw.Transform( (HomPoint3D)eye );
    Tuvw.InvertAffine( );

    Transform3D Tshear(ZSHEAR, -eye.X()/eye.Z(), -eye.Y()/eye.Z());
```



```

switch( projtype ) {
case PARALLEL:
    Transform3D Tviewport(SCALE,
                          viewport.HSize()/window.HSize(),
                          viewport.VSize()/window.HSize(),
                          1/(bp-fp));
    Tviewport *= Transform3D(TRANSLATION,
                            Vector3D(viewport.HCenter(),
                                       viewport.VCenter(),
                                       -fp/(bp-fp)) );

    transf = Tuvw * Tshear * Tviewport;
    break;
case PERSPECTIVE:
    Transform3D Teye(TRANSLATION, -eye);
    Transform3D Tnorm(SCALE,
                      -2 * eye.Z() / (window.HSize() * bp),
                      -2 * eye.Z() / (window.VSize() * bp),
                      1/bp);
    Transform3D Tpers(PROJECTIVE,
                      HomPoint3D(viewport.HSize()/2, 0, 0, 0),
                      HomPoint3D(0, viewport.VSize()/2, 0, 0),
                      HomPoint3D(viewport.HCenter(),
                                  viewport.VCenter(),
                                  bp/(bp-fp), 1),
                      HomPoint3D(0, 0, -fp/(bp-fp), 0));
    transf = Tuvw * Teye * Tshear * Tnorm * Tpers;
}
}

```

A megvilágításért az ambiens fényen kívül az absztrakt fényforrások felelősek. Egy absztrakt fényforrás lehet *irányfény* típusú, amikor a fény egy irányból jön, és intenzitása független a fényforrás távolságától (a nap lényegében ilyen), vagy *pontfény* típusú, amikor a fény egy pontból jön, és erőssége a távolság négyzetével csökken. Az irányfényeket a `DirectionalLight` osztállyal, a pontfényeket pedig a `PositionalLight` osztállyal definiálhatjuk, amelyeket a közös `Light` osztályból származtattunk.

```

//=====
class Light {
//=====
    SColor intensity;
public:
    Light( SColor& inten ) : intensity(inten) { }
    virtual Vector3D LightDir(Point3D& x) { return Vector3D(0,0,0); }
    virtual SColor Le(Point3D& x, Vector3D& dir) { return intensity; }
};

```

```
//=====
class DirectionalLight : public Light {
//=====
    Vector3D dir;
public:
    DirectionalLight( Vector3D& p, SColor& inten ) : Light(inten) {
        p.Normalize(); dir = p;
    }
    Vector3D LightDir(Point3D& x) { return dir; }
};

//=====
class PositionalLight : public Light {
//=====
    Point3D pos;
public:
    PositionalLight( Point3D& p, SColor& inten )
    : Light(inten), pos(p) { }
    Point3D& Pos( ) { return pos; }
    Vector3D LightDir(Point3D& x) {
        Vector3D v = (pos - x);
        v.Normalize();
        return v;
    }
    SColor Le(Point3D& x, Vector3D& dir) {
        return (Light::Le(pos, dir) / ((x - pos) * (x - pos)));
    }
};

typedef Array<Light *> Lights;
```

A 2D grafikus rendszerhez hasonlóan a képszintézishez szükséges összes információt a színtérben (Scene) foglaljuk össze. A színtér tartalmazza a megjelenítőobjektum azonosítóját (scr), a virtuális világmodellt (world), a kameraparamétereket (camera), a fényforrásokat (lightsources), és az ambiens megvilágítás intenzitását (La). A Define tagfüggvény felépíti a virtuális modellt és definiálja a kamerát, valamint a fényforrásokat, a Render tagfüggvény pedig elvégzi a képszintézist.

```
//=====
class Scene {
//=====
    Window *      scr;
    VirtualWorld  world;
    Camera3D      camera;
    Lights        lightsources;
    Color         La;
```


A Render tagfüggvény számítja ki a képet. Először törli a képernyőt és inicializálja a z-buffer memóriát, majd sorra veszi a virtuális világ objektumait. Az egyes objektumok feldolgozása során a függvény az objektum primitívjeit tesszellálja, és a tesszellált primitívet a világ-koordinátarendszerbe transzformálja. Itt kiszámítjuk a primitív megvilágítását, majd végigvezetjük a nézeti csővezeték transzformációs és vágási lépésein, végül felrajzoljuk a képernyőre. A helyes takarási viszonyok kialakításáért a z-buffer felelős.

```
//-----
void Scene :: Render( ) {
//-----
    scr -> Clear( );
    zbuffer.Clear( );

    for(int o = 0; o < world.ObjectNum(); o++) {
        Object3D * obj = world.Object( o );
        Transform3D Tv = camera.ViewTransform();
        Transform3D Tm = obj -> Transform();
        for(int p = 0; p < obj -> PrimitiveNum( ); p++) {
            RenderPrimitive3D * rp = obj->Primitive(p)->Tessellate();
            rp -> Transform( Tm );
            rp -> TransformNormals( Tm );
            rp -> Illuminate(lightsources, La, camera.GetWorldEye());
            rp -> Transform( Tv );
            if ( rp -> DepthClip( ) ) {
                rp -> HomDivPoints( );
                if ( rp -> Clip(camera.Viewport()) ) rp -> Draw(scr);
            }
            delete rp;
        }
    }
}
```

10. fejezet

A sugárkövetés

A *sugárkövetés* a képernyő pixeleire egymástól függetlenül oldja meg a takarási és árnyalási feladatokat.

10.1. Az illuminációs modell egyszerűsítése

A *rekurzív sugárkövetés* a lokális illuminációs algoritmusokhoz hasonlóan, de kevésbé durván egyszerűsíti az árnyalási egyenletet. A lehetséges visszaverődésekből és törésekből elkülöníti a geometriai optikának megfelelő ideális (ún. *koherens*) eseteket, és csak ezekre hajlandó a többszörös visszaverődések és törések követésére. A többi, ún. *inkoherens komponensre* viszont lokális illuminációs módszerekhez hasonlóan elhanyagolja a csatolásokat és csak az absztrakt fényforrások direkt megvilágítását veszi figyelembe.

Az elmondott elhanyagolások következtében az *illuminációs képlet* a következő alakra egyszerűsödik:

$$L(\vec{x}, \omega) = L^e(\vec{x}, \omega) + k_a \cdot L^a + \sum_l f_{ri}(\omega'_l, \vec{x}, \omega) \cdot \cos \theta'_l \cdot L^{\text{in}}(\vec{x}, \omega'_l) + k_r \cdot L^{\text{in}}(\vec{x}, \omega_r) + k_t \cdot L^{\text{in}}(\vec{x}, \omega_t), \quad (10.1)$$

ahol ω_r az ω tükriránya, ω_t a fénytörésnek megfelelő irány, $f_{ri}(\omega'_l, \vec{x}, \omega)$ a diffúz és a spekuláris visszaverődést jellemző BRDF, $L^{\text{in}}(\vec{x}, \omega'_l)$ pedig az l . absztrakt fényforrásból az \vec{x} pontba az ω'_l irányból érkező radiancia.

Egy pixel színének számításához mindenképp a pixelben látható felületi pontot kell megkeresnünk. Ehhez a szempozícióból a pixel középpontján keresztül egy félegyenest, ún. sugarat indítunk. A sugár legközelebbi metszéspontja az illuminációs képletben szereplő \vec{x} pont lesz, a félegyenes irányvektora pedig a $-\omega$ iránynak felel

meg. Ezekkel a paraméterekkel kiértékeljük az illuminációs képletet, és a pixelt ennek megfelelően kiszínezzük.

Az illuminációs képlet kiértékeléséhez a következőket kell elvégezni:

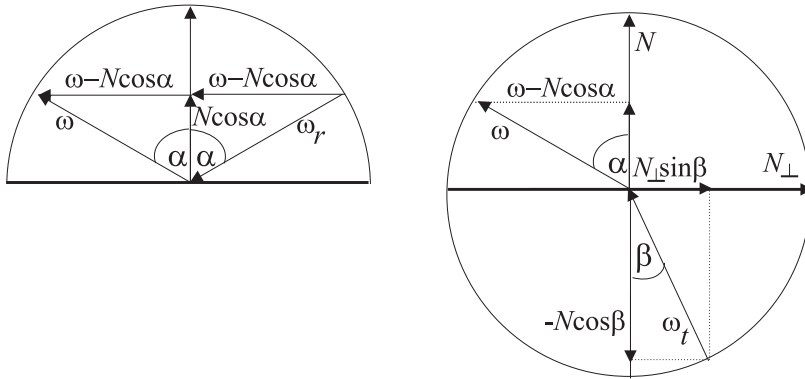
- Az \vec{x} felületi pont és ω nézeti irány ismeretében a saját sugárzás és az ambiens visszaverődés minden további nélkül kiértékelhető.
- A tükörirányból érkező fény visszaveréséhez kiszámítjuk a tükörirányt és meghatározzuk az innen érkező radianciát, amit a látható színben k_r súllyal veszünk figyelembe. Vegyük észre, a tükörirányból érkező radiancia számítása pontosan ugyanarra a feladatra vezet, amit a pixel színének a számításához végzünk el, csupán a vizsgált irányt most nem a szem és a pixel középpont, hanem 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 radianciát a k_t tényezővel megszorozzuk.
- Az inkoherens visszaverődések kiszámításához minden egyes fényforrásról eldöntjük, hogy az látszik-e az adott pontból vagy sem. Most ugyanis nem tételizzük fel automatikusan, hogy a fényforrások a felületi pontból láthatóak, így a képen *árnyékok* is megjelenhetnek. Ha tehát az l . fényforrás teljesítménye Φ_l , pozíciója pedig \vec{y}_l , akkor a beérkező radiancia:

$$L^{\text{in}}(\vec{x}, \omega'_l) = v(\vec{x}, \vec{y}_l) \cdot \frac{\Phi_l}{|\vec{x} - \vec{y}_l|^2}, \quad (10.2)$$

ahol a $v(\vec{x}, \vec{y})$ láthatósági indikátor változó 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 objektumok vannak, a v 0 és 1 közötti értéket is felvehet. A v tényező kiszámításához egy sugarat indítunk az \vec{x} pontból a fényforrás felé és ellenőrizzük, hogy ez az *árnyék sugár* metsz-e objektumot, mielőtt elérné a fényforrást, majd a metszett objektumok k_t átlátszósági tényezőit összeszorozva meghatározzuk v értékét. 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 radianciáját minden reprezentatív hullámhosszon tovább kell adnunk.

10.2. A tükör és törési irányok kiszámítása



10.1. ábra. A tükörirány és a törési irány kiszámítása

A tükörirányt a következőképpen számíthatjuk ki (10.1. ábra):

$$\omega_r = (\omega - \vec{N} \cdot \cos \alpha) - \vec{N} \cdot \cos \alpha = \omega - 2 \cos \alpha \cdot \vec{N}. \quad (10.3)$$

ahol α a beesési szög, melynek koszinusza a $\cos \alpha = (\vec{N} \cdot \omega)$ skalárszorzattal állítható elő.

A törési irány meghatározása egy kicsit bonyolultabb. Ha a törés szöge β , akkor a törés irányába mutató egységvektor:

$$-\omega_t = -\cos \beta \cdot \vec{N} + \sin \beta \cdot \vec{N}_\perp. \quad (10.4)$$

ahol \vec{N}_\perp a normálvektorra merőleges, a normálvektor és a beesési vektor síkjába eső egységvektor:

$$\vec{N}_\perp = \frac{\cos \alpha \cdot \vec{N} - \omega}{|\cos \alpha \cdot \vec{N} - \omega|} = \frac{\cos \alpha \cdot \vec{N} - \omega}{\sin \alpha} \quad (10.5)$$

Ezt behelyettesítve és felhasználva a Snellius-Descartes-törvényt, miszerint

$$\frac{\sin \alpha}{\sin \beta} = n$$

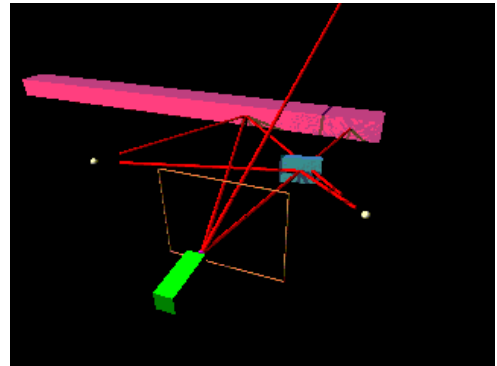
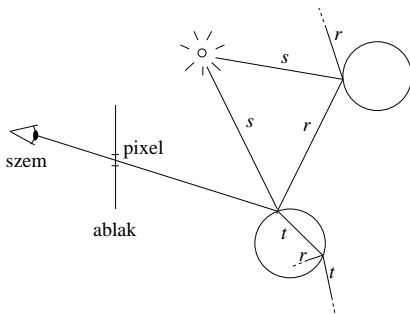
(n a relatív törésmutató), a következő összefüggéshez jutunk:

$$\omega_t = \cos \beta \cdot \vec{N} - \frac{\sin \beta}{\sin \alpha} (\cos \alpha \cdot \vec{N} - \omega) = \frac{\omega}{n} - \vec{N} \left(\frac{\cos \alpha}{n} - \cos \beta \right) =$$

$$\frac{\omega}{n} - \vec{N} \left(\frac{\cos \alpha}{n} - \sqrt{1 - \sin^2 \beta} \right) = \frac{\omega}{n} - \vec{N} \left(\frac{\cos \alpha}{n} - \sqrt{1 - \frac{(1 - \cos^2 \alpha)}{n}} \right). \quad (10.6)$$

A képletben szereplő n relatív törésmutató értéke attól függ, hogy most éppen belépünk-e az anyagba, vagy kilépünk belőle (a két esetben ezek az értékek egymásnak reciprokai). Az aktuális helyzetet a sugár iránya és a felületi normális által bezárt szög, illetve a skaláris szorzat előjele alapján ismerhetjük fel. 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 ritkébb anyagba.

10.3. A rekurzív sugárkövető program



10.2. ábra. Rekurzív sugárkövetés

A sugárkövető programunk az egyes pixelek színét egymás után és egymástól függetlenül számítja ki:

```
for minden  $p$  pixelre do
     $r$  = szemből a pixel középpontjába mutató sugár
    color of  $p$  = Trace( $r$ , 0)
endfor
```

A **Trace**(r , d) szubrutin az r sugár irányából érkező radianciát határozza meg rekurzív módon. A d változó a visszaverődések, illetve törések számát tartalmazza, annak érdekében, hogy a rekurzió mélységét korlátozzuk:


```

Color Trace( $r, d$ )
  if  $d > d_{\max}$  then return  $L^a$  // rekurzió korlátozása
  ( $q, \vec{x}$ ) = Intersect( $r$ ) //  $q$ : objektum,  $\vec{x}$ : felületi pont
  if nincs metszéspont then return  $L^a$ 
   $\omega = r$  irányvektora
   $c = L_q^e(\vec{x}, \omega) + k_a \cdot L^a$ 
  for  $l$ . fényforrásra do
     $r_s = \vec{x}$ -ből induló,  $\vec{y}_l$  felé mutató sugár // árnyék sugár
    ( $q_s, \vec{x}_s$ ) = Intersect( $r_s$ )
    if nincs metszéspont OR  $|\vec{x}_s - \vec{x}| > |\vec{y}_l - \vec{x}|$  then // fényforrás nem takart
       $c += f_{r_i}(\omega'_l, \vec{x}, \omega) \cdot \cos \theta'_l \cdot \Phi_l / |\vec{x} - \vec{y}_l|^2$ 
    endif
  endfor
  if  $k_r(\vec{x}) > 0$  then
     $r_r =$  az  $r$  tükkörirányába mutató sugár
     $c += k_r(\vec{x}) \cdot \mathbf{Trace}(r_r, d + 1)$ 
  endif
  if  $k_t(\vec{x}) > 0$  then
     $r_t =$  az  $r$  törési irányába mutató sugár
     $c += k_t(\vec{x}) \cdot \mathbf{Trace}(r_t, d + 1)$ 
  endif
  return  $c$ 
end

```

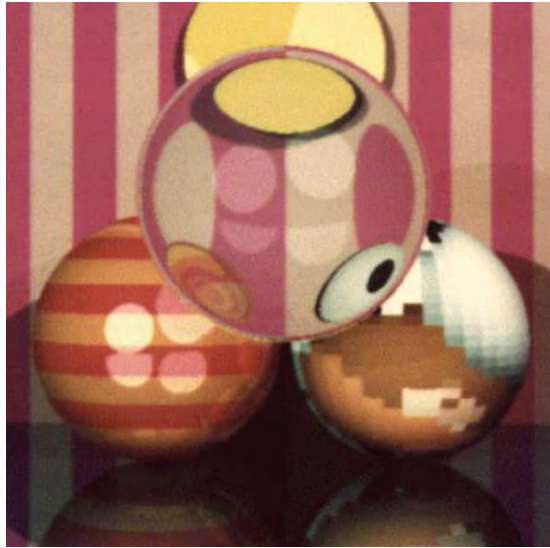
A szubrutin kezdetén a rekurzió mélységének korlátozására egyrészt azért van szükség, hogy a tükköraszobában fellépő végtelen rekurziót elkerüljük, másrészt pedig azért, hogy az elhanyagolható sokadik visszaverődések kiszámítására ne pazaroljuk a drága időnket.

10.4. Metszéspontszámítás egyszerű felületekre

Az **Intersect**(r) függvény az r sugár és a legközelebbi felület metszéspontját keresi meg. A gyakorlati tapasztalatok szerint a sugárkövető programunk a futás során az idő 75–96%-t az **Intersect**(r) rutinban tölti, ezért ennek hatékony implementációja a gyors sugárkövetés kulcsa. A sugarat általában a következő egyenlettel adjuk meg:

$$\vec{r}(t) = \vec{s} + t \cdot \vec{d}, \quad (t \in [0, \infty)). \quad (10.7)$$

ahol \vec{s} a kezdőpont, $\vec{d} = -\omega$ a sugár iránya, a t *sugárparaméter* pedig kifejezi a kezdőponttól való távolságot. A következőkben áttekintjük, hogy a különböző primitív típusok hogyan metszhetők el az ily módon megadott sugárral.



10.3. ábra. Sugárkövetéssel előállított tipikus kép: középső gömb koherensen törő, a többi gömb és az alaplap koherensen visszaverő; a fényforrások láthatóságszámítása miatt éles árnyékok keletkeznek

10.4.1. Háromszögek metszése

A háromszögek metszése két lépésben történik. Először előállítjuk a sugár és a háromszög síkjának a 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 $(\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})$, egy helyvektora pedig \vec{a} , tehát a sík p pontjai kielégítik a következő egyenletet:

$$((\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})) \cdot (\vec{p} - \vec{a}) = 0. \quad (10.8)$$

A sugár és a sík közös pontját megkaphatjuk, ha a sugár egyenletét behelyettesítjük a sík egyenletébe, 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 van. A sík metszése után azt kell ellenőriznünk, hogy a kapott \vec{p} pont vajon a háromszögon kívül vagy belül helyezkedik-e el. A \vec{p} metszéspont akkor van a háromszögon belül, ha a háromszög mind a három oldalegyeneséhez viszonyítva a háromszöget tartalmazó félsíkban van:

$$\begin{aligned} ((\vec{b} - \vec{a}) \times (\vec{p} - \vec{a})) \cdot ((\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})) &\geq 0, \\ ((\vec{c} - \vec{b}) \times (\vec{p} - \vec{b})) \cdot ((\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})) &\geq 0, \\ ((\vec{a} - \vec{c}) \times (\vec{p} - \vec{c})) \cdot ((\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})) &\geq 0. \end{aligned} \quad (10.9)$$

10.4.2. Implicit felületek metszése

A síkmetszéshez hasonlóan egy gömbre úgy kereshetjük a metszéspontot, ha a sugár egyenletét behelyettesítjük a gömb egyenletébe:

$$|(\vec{s} + t \cdot \vec{d}) - \vec{c}|^2 = R^2 \quad (10.10)$$

majd megoldjuk t -re az ebből adódó

$$(\vec{d})^2 \cdot t^2 + 2 \cdot \vec{d} \cdot (\vec{s} - \vec{c}) \cdot t + (\vec{s} - \vec{c})^2 - R^2 = 0 \quad (10.11)$$

egyenletet. 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, amit a megoldóképlet alkalmazásával könnyen megoldhatunk.

Általánosan egy $F(x, y, z) = 0$, implicit egyenlettel definiált felületek 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 [SKe95].

10.4.3. 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

$$\vec{r}(u, v) = \vec{s} + t \cdot \vec{d} \quad (10.12)$$

háromváltozós nem lineáris egyenletrendszer, 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 követik, hogy a felületeket poligonhálóval közelítik (emlékezzünk vissza, hogy ez az ún. *tesszellációs* folyamat különösen egyszerű paraméteres felületekre), majd ezen poligonhálót próbálják elmetszeni. Ha sikerül metszéspontot találni, az eredményt úgy lehet pontosítani, hogy a metszéspont környezetének megfelelő paramétertartományban egy finomabb tesszellációt végzünk, és a metszéspontszámítást újra elvégezzük.

10.4.4. Transzformált objektumok metszése

A sugárkövetés egyedülálló tulajdonsága, hogy nem igényel tesszellációt, azaz az objektumokat nem kell poligonhálójával közelíteni, mégis implicit módon elvégzi a nézeti transzformációs, vágási, vetítési és takarási feladatokat. Ha az objektumokat közvetlenül a világ-koordinátarendszerben írjuk le, ezek elegendőek is a teljes képszintézishez. Ha viszont az objektumok különálló modellezési koordinátarendszerben találhatók, a modellezési transzformációt valahogyan meg kell valósítani. Ez ismét csak elvezet minket ahhoz a problémához, hogy hogyan is kell transzformálni például egy gömböt. Szerencsére ezt a kérdést megkerülhetjük, ha nem az objektumot, hanem a sugarat transzformáljuk, hiszen a sugár és egy \mathbf{T} transzformációval torzított objektum metszéspontját kiszámíthatjuk úgy is, hogy meghatározzuk a \mathbf{T} inverzével transzformált sugár és az objektum metszetét, majd a \mathbf{T} alkalmazásával az eredeti sugárra képezzük a pontokat.

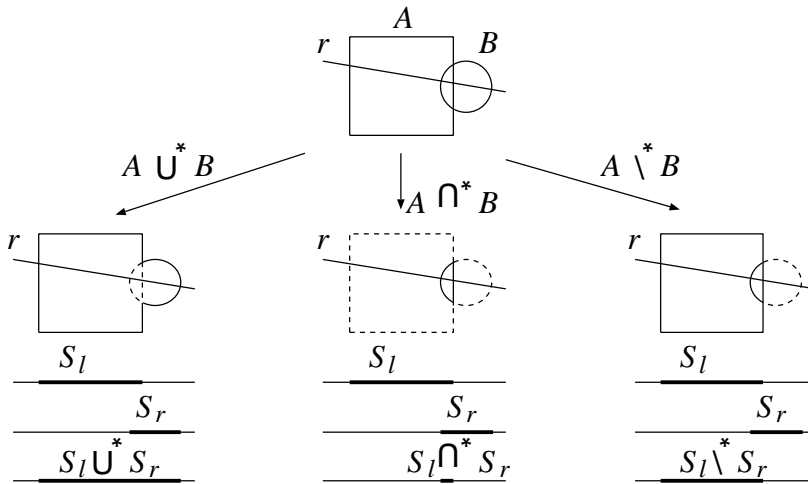
10.4.5. CSG modellek metszése

A *konstruktív tömörtest geometria (CSG)* a modelleket egyszerű primitívekből (kocka, henger, kúp, gömb, stb.) halmazműveletek (\cup^* , \cap^* , \setminus^*) segítségével állítja elő. Egy objektumot általában egy bináris fa adatstruktúra ír le, amelyben a levelek a primitívet azonosítják, a belső csomópontok pedig a két gyermekén 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 az objektum felülete közötti metszéspontok $t_1 \leq t_2, \dots \leq t_{2k}$ sugárparamétereknél találhatók. Ekkor a sugár a $(\vec{s} + t_1 \cdot \vec{d}, \vec{s} + t_2 \cdot \vec{d}), \dots (\vec{s} + t_{2k-1} \cdot \vec{d}, \vec{s} + t_{2k} \cdot \vec{d})$, pontpárok közötti szakaszokon (ún. belső szakaszok (*ray-span*)) 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 az előző fejezetben javasolt megoldással kezelhetjük.

Most tegyük fel, hogy a sugárral nem csupán egy primitív objektumot, hanem egy CSG fával leírt struktúrát kell el metszeni! A fa csúcán egy halmazművelet található, ami 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 há-

romnál 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.



10.4. ábra. Belső szakaszok és a kombinálásuk

Az ismertett 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.

CSGIntersect(ray, node)

```

if node nem levél then
    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);
endif
end

```

10.5. A metszéspontszámítás gyorsítási lehetőségei

A sugárkövetést megvalósító algoritmus minden egyes sugarat minden objektummal összevet és eldönti, hogy van-e közöttük metszéspont. A módszer jelentősen gyorsítható lenne, ha az objektumok egy részére kapásból meg tudnánk mondani, hogy egy adott sugár biztosan nem metszheti őket (mert például 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 biztosan ezen metszéspont mögött helyezkednek el. Ahhoz, hogy ilyen döntéseket hozzassunk, ismernünk kell az objektumteret. A megismeréshez egy előfeldolgozási fázis szükséges, amelyben egy adatstruktúrát építünk fel. A sugárkövetés végrehajtásakor pedig a kívánt információkat ebből az adatstruktúrából olvassuk ki.

10.5.1. Befoglaló keretek

A legegyszerűbb gyorsítási módszer a *befoglaló keretek* (*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 elmetezni. 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 vágási algoritmus* 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 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.

10.5.2. Az objektumtér szabályos felosztása

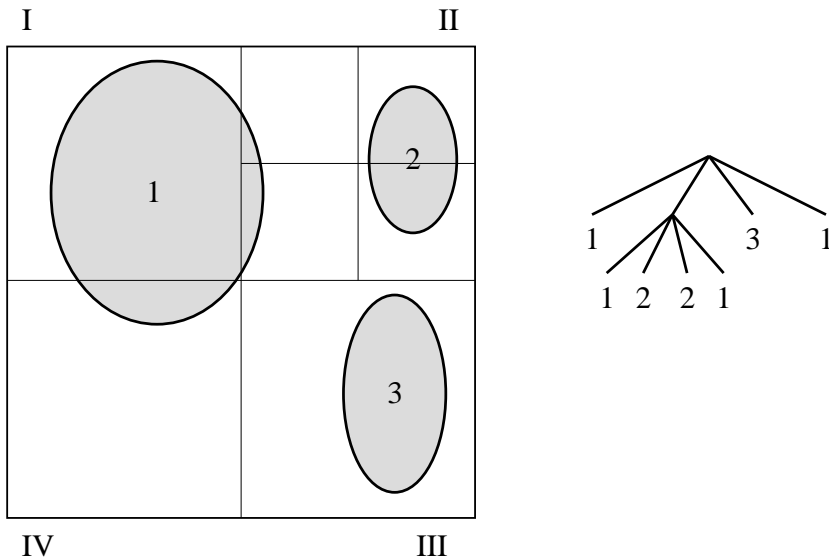
Tegyünk az objektumtérre egy szabályos 3D rácsot és az előfeldolgozás során minden cellára határozzuk meg a cellában lévő, vagy a cellába lógó objektumokat. A sugár-

követés 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 metszéspontunk mögött van.

Az objektumtér szabályos felosztásának előnye, hogy a meglátogatandó cellák könnyen előállíthatók a DDA algoritmus három dimenziós általánosításának segítségével [FTK86], 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.

10.5.3. Az objektumtér adaptív felosztása

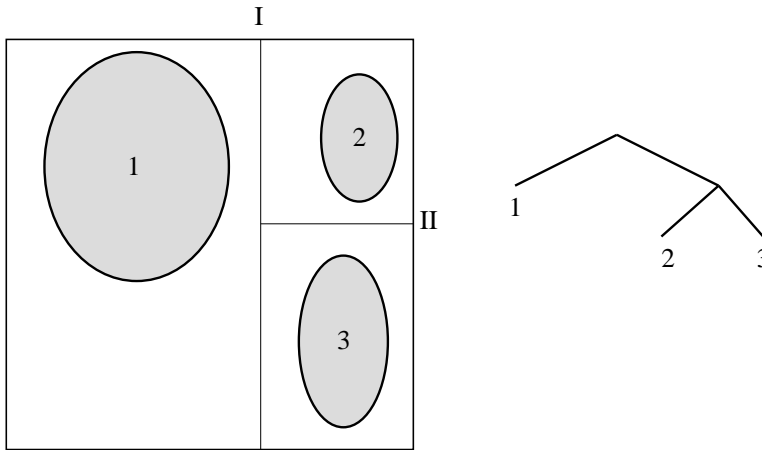
Az objektumtér adaptív felosztása rekurzív megközelítéssel lehetséges. Foglalkozunk kezdetben az objektumainkat egy koordinátatengelyekkel párhuzamos oldalú dobozba. Vizsgáljuk meg, hogy a dobozunk homogénnek tekinthető-e, azaz a benne legfeljebb 1 (általánosabban legfeljebb adott számú) objektum van-e. Ha igen, a felosztással elkészültünk. Ha nem, a dobozt a felezősíkjai mentén 8 egybevágó részre bontjuk és a keletkező részdobozokra ugyanezt az eljárást folytatjuk.



10.5. ábra. A síkot felosztó négyes fa. Ennek a 3D változata az oktális fa

Az eljárás eredménye egy *oktális fa* (10.5. ábra). A fa levelei azon elemi cellák, amelyekhez a belógó objektumokat nyilvántartjuk.

A felosztás adaptivitását fokozhatjuk, ha egy lépésben nem mind a három felezősík mentén vágunk, hanem egy olyan (általában ugyancsak a koordináta-rendszer valamely tengelyére merőleges) síkkal, amely az objektumteret a lehető legigazságosabban felezi meg. Ez a módszer egy bináris fához vezet, amelynek neve *bináris particionáló fa*, vagy *BSP-fa*.



10.6. ábra. Bináris particionáló fa

Az adaptív felosztás kétségkívül kevesebb memóriát igényel, mint a tér szabályos felosztása. Azonban egy új problémát vet fel, amivel foglalkoznunk kell. A szabályos felosztás rácsán szakaszrajzoló algoritmusok segítségével kényelmesen sétálhatunk, azaz könnyen eldönthetjük, hogy egy cella után melyik lesz a következő, ami 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. 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 kapunk. Azt, hogy ez melyik cellához tartozik, az adatstruktúra bejárásával dönthetjük el. Kézbe fogván a pontunkat a fa csúcsán belépünk az adatstruktúrába. A pont koordinátáit a felosztási feltétellel (oktális fánál az aktuális doboz középpontjával, bináris particionáló fánál pedig a sík koordinátájával) összehasonlítva eldönthetjük, hogy melyik úton kell folytatni az adatszerkezet bejárását. Előbb-utóbb eljutunk egy levélig, azaz azonosítjuk a pontot tartalmazó cellát.

10.6. Foton követés

A *foton követés* (*photon-tracing*) a sugárkövetés inverze, amikor a fényutakat nem a szemből, hanem a fényforrásoktól kezdjük építeni. Egy fényút a fényforrás egy pontján kezdődik. Itt kiválasztunk egy irányt, és elindítunk egy sugarat ebben az irányban. Ha az eltalált felület diffúz visszaverődést tartalmaz, annak intenzitását a sugár által szállított radiancia és a megtalált pont optikai jellemzői alapján számítjuk ki. Ezt követően eldöntjük, hogy ez a pont hat-e közvetlenül valamely pixelre, azaz látható-e valamely pixelben, és ha igen, annak színéhez hozzáadjuk a pont hatását. A pont láthatóságának eldöntéséhez egy sugarat indítunk a pontból a szem felé, és ellenőrizzük, hogy ez a sugár metsz-e valamilyen objektumot, mielőtt elérné a szemet. Ha a megtalált pont koherensen törő vagy visszaverő felülethez tartozik, akkor rekurzíven gyermekugarakat indítunk a törő és visszaverő irányokba, és az egész eljárást megismételjük.

10.7. Program: rekurzív sugárkövetés

Egy sugár (Ray) kezdőponttal (*start*) és irányvektorral (*dir*) jellemezhető.

```
//=====
class Ray {
//=====
    Vector3D start, dir;
public:
    Ray( Vector3D start0, Vector3D dir0 ) {
        start = start0; dir = dir0; dir.Normalize();
    }
    Vector3D Dir() { return dir; }
    Vector3D Start() { return start; }
};
```

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. Ehelyett az ideális tükröket jellemző anyagoknál előállíthatjuk azt az irányt, amerre a fény folytatja az útját (*ReflectionDir*).

```
//=====
class IdealReflector {
//=====
    SColor Kr;
public:
    SColor& kr( ) { return Kr; }
    void ReflectionDir(Vector3D& L, Vector3D& N, Vector3D& V) {
        L = N * (N * V) * 2 - V;
    }
};
```

Az ideális fénytörő anyag szintén csak egyetlen irányba adja tovább a fényt, amelyet a `RefractionDir` függvényvel számíthatunk ki az anyag törésmutatójából (N). A függvény bemeneti paraméterei között szerepel az `out` változó is, amely jelzi, 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.

```
//=====
class IdealRefractor {
//=====
    SColor Kt;
    double N;
public:
    IdealRefractor( ) : Kt(0) { N = 1; }
    SColor& kt( ) { return Kt; }
    double& n( ) { return N; }
    BOOL RefractionDir(Vector3D& L, Vector3D& N, Vector3D& V,
        BOOL out) {
        double cn = ( out ) ? n( ) : 1.0/n( );
        double cosa = N * V;
        double disc = 1 - (1 - cosa * cosa) / cn / cn;
        if (disc < 0) return FALSE;
        L = N * (cosa / cn - sqrt(disc)) - V / cn;
        return TRUE;
    }
};
```

Általános esetben egy anyag a beérkező fényt részben diffúz és spekuláris jelleg szerint, vagy akár ideális tükör vagy fénytörő anyagként veri vissza:

```
//=====
class GeneralMaterial : public DiffuseMaterial,
                        public SpecularMaterial,
                        public IdealReflector,
                        public IdealRefractor {
//=====
public:
    SColor BRDF(Vector3D& L, Vector3D& N, Vector3D& V) {
        return (DiffuseMaterial :: BRDF(L, N, V) +
            SpecularMaterial :: BRDF(L, N, V));
    }
};
```

Egy általános objektum primitívekből áll.

```
//=====
class Object3D {
//=====
    Array<Primitive3D *> prs;
    Transform3D          tr;
public:
    Object3D( ) { }
    void AddPrimitive( Primitive3D * p ) { prs[ prs.Size() ] = p; }
    Primitive3D * Primitive( int i ) { return prs[i]; }
    Transform3D& Transform( ) { return tr; }
    int PrimitiveNum() { return prs.Size(); }
};
```

Egy primitív felületének optikai tulajdonságaival és a geometriájával jellemezhető. Az optikai tulajdonságok az általános anyagjellemzőket és az ambiens visszaverődési tényezőt (K_a) foglalják magukban. A geometriai tulajdonságok két eljárással kérdezhetők le: egy adott sugár metszi-e a primitívet, és ha igen, milyen sugárparaméter-nél (Intersect); illetve egy adott felületi pontban hogyan áll a felület normálvektora (Normal).

```
Point3D dummy;
```

```
//=====
class Primitive3D : public Emitter {
//=====
protected:
    SColor          Ka;
    GeneralMaterial mat;
public:
    Primitive3D( ) : Ka( 0.1 ) { }
    SColor& ka( Point3D x = dummy ) { return Ka; }
    SColor& kd( Point3D x = dummy ) { return mat.kd(); }
    SColor& ks( Point3D x = dummy ) { return mat.ks(); }
    double& Shine( Point3D x = dummy ) { return mat.Shine(); }
    SColor& Le( Point3D x, Vector3D dir ) { return Emitter :: Le(); }
    SColor& kr( Point3D x = dummy ) { return mat.kr(); }
    SColor& kt( Point3D x = dummy ) { return mat.kt(); }
    double& n( Point3D x = dummy ) { return mat.n(); }
    BOOL ReflectionDir( Vector3D& L, Vector3D& N, Vector3D& V,
                       Point3D& x ) {
        return mat.ReflectionDir(L, N, V);
    }
    BOOL RefractionDir( Vector3D& L, Vector3D& N, Vector3D& V,
                       Point3D& x, BOOL out ) {
        return mat.RefractionDir(L, N, V, out );
    }
}
```

```

    SColor BRDF(Vector3D& L, Vector3D& N, Vector3D& V, Point3D& x) {
        return mat.BRDF(L, N, V);
    }
    virtual double Intersect( Ray& r ) = 0;
    virtual Vector3D Normal(Point3D& x) = 0;
};

```

A geometriai információ és műveletek tényleges megadásához ismernünk kell az objektum típusát és alakját. Például egy gömböt definiáló osztály a következőképpen adható meg.

```

//=====
class Sphere : public virtual Primitive3D {
//=====
    Point3D center;
    double radius;
public:
    Sphere(Point3D& cent, double rad)
    : Primitive3D() { center = cent; radius = rad; }
    double Intersect( Ray& r );
    Vector3D Normal(Point3D& x) { return ((x - center)/radius); }
};

//-----
double Sphere :: Intersect( Ray& r ) {
//-----
    Vector3D dist = r.Start() - center;
    double b = (dist * r.Dir()) * 2.0;
    double a = (r.Dir() * r.Dir());
    double c = (dist * dist) - radius * radius;

    double discr = b * b - 4.0 * a * c;
    if ( discr < 0 ) return -1;
    double sqrt_discr = sqrt( discr );
    double t1 = (-b + sqrt_discr)/2.0/a;
    double t2 = (-b - sqrt_discr)/2.0/a;

    if (t1 < EPSILON) t1 = -EPSILON;
    if (t2 < EPSILON) t2 = -EPSILON;
    if (t1 < 0 && t2 < 0) return -1;

    double t;
    if ( t1 < 0 && t2 >= 0 )      t = t2;
    else if ( t2 < 0 && t1 >= 0 ) t = t1;
    else if (t1 < t2)           t = t1;
    else                        t = t2;
    return t;
}

```

Az inkrementális képszintézisben használt kamerát egyetlen olyan tagfüggvénnyel kell kiegészíteni, amely egy képernyőn lévő ponthoz megkeresi azt a sugarat, amely a szemből indul, és éppen ezen a ponton megy keresztül.

```
//=====
class RayCamera : public Camera3D {
//=====
public:
    Ray GetRay( Coord X, Coord Y ) {
        Vector3D w = vpn; w.Normalize( );
        Vector3D u = w % vup; u.Normalize( );
        Vector3D v = u % w;
        Transform3D Tuvw( AFFIN, u, v, w, vrp );
        Vector3D world_eye = Tuvw.Transform( (HomPoint3D)eye );
        double x, y;
        x = window.HSize()/viewport.HSize() * (X - viewport.HCenter());
        y = window.VSize()/viewport.VSize() * (Y - viewport.VCenter());
        Vector3D dir = u * x + v * y - world_eye;
        return Ray( world_eye + vrp, dir );
    }
};
```

A virtuális világ különböző típusú objektumok gyűjteménye. Az egyes objektumok primitívekből állnak, amelyeket egymás után a `NextPrimitive` tagfüggvény szolgáltat.

```
//=====
class VirtualWorld {
//=====
    Array<Object3D *>  objs;
    int actobj, actprim;
public:
    VirtualWorld( ) { actobj = actprim = 0; }
    Object3D * Object( int o ) { return objs[o]; }
    void AddObject( Object3D * o ) { objs[ objs.Size() ] = o; }
    int ObjectNum() { return objs.Size(); }
    Primitive3D * NextPrimitive( );
};
```

A színtér ismét a virtuális világból, a kamerából és a fényforrásokból áll.

```
//=====
class Scene {
//=====
    VirtualWorld          world;
    RayCamera             camera;
    Array< PositionalLight * > lightsources;
    SColor                La;

    Primitive3D * Intersect( Ray& r, Point3D& x );
    SColor               IntersectShadow( Ray r, double maxt );
    SColor               DirectLightsource(Primitive3D * q, Vector3D& V,
                                           Vector3D& N, Point3D& x);
    SColor               Trace(Ray r, int depth);
    void                 WritePixel( PCoord X, PCoord Y, SColor col );
public:
    Scene( ) { Define( ); }
    void Define( );
    void Render( );
};
```

Az `Intersect` tagfüggvény megkeresi azon primitívet, amelyet a sugár a kezdő-pontjához legközelebb metsz, és visszatérési értéként megadja a primitív címét, az `x` változóban pedig a metszéspont koordinátáit.

```
//-----
Primitive3D * Scene :: Intersect( Ray& r, Point3D& x ) {
//-----
    double    t = -1;
    Primitive3D * po = NULL, * p;
    while( (p = world.NextPrimitive( )) != NULL ) {
        double tnew = p -> Intersect( r );
        if ( tnew > 0 && (tnew < t || t < 0) ) {
            t = tnew;
            po = p;
        }
    }
    if ( t > 0 ) x = r.Start() + r.Dir() * t;
    return po;
}
```

Az `IntersectShadow` az árnyéksugarakat követi a sugár kezdőpontjától a `maxt` maximális sugárparaméterig, és kiszámítja az ezalatt eltalált primitívek eredő átlátszóságát.

```
//-----
SColor Scene :: IntersectShadow( Ray r, double maxt ) {
//-----
    SColor att = SColor(1);
    Primitive3D * p;
    while( ( p = world.NextPrimitive( ) ) != NULL ) {
        double t = p -> Intersect( r );
        if ( t > EPSILON && t < maxt ) {
            Point3D x = r.Start() + r.Dir() * t;
            att *= p -> kt( x );
        }
    }
    return att;
}
```

A `DirectLightsource` tagfüggvény a saját emisszió és az absztrakt fényforrások közvetlen hatását határozza meg.

```
//-----
SColor Scene :: DirectLightsource(Primitive3D * q, Vector3D& V,
                                   Vector3D& N, Point3D& x) {
//-----
    SColor c = q->Le(x, V) + q->ka(x) * La;

    for(int l = 0; l < lightsources.Size(); l++) {
        Vector3D L = lightsources[l] -> Pos() - x;
        double lightdist = L.Length();
        L /= lightdist;
        SColor atten = IntersectShadow(Ray(x, L), lightdist);
        if (atten != 0) {
            double cost = N * L;
            if (cost > 0)
                c += atten * q->BRDF(L, N, V, x) *
                    lightsources[l] -> Le(x, -L) * cost;
        }
    }
    return c;
}
```

A program legfontosabb része a sugarat rekurzívan követő Trace függvény.

```
//-----
SColor Scene :: Trace(Ray r, int d) {
//-----
    if (d > maxdepth) return La;
    Point3D x;
    Primitive3D * q = Intersect(r, x);
    if (q == NULL) return La;

    Vector3D normal = q -> Normal(x);
    BOOL out = TRUE;
    if ( normal * (-r.Dir()) < 0 ) { normal = -normal; out = FALSE; }

    SColor c = DirectLightsource(q, -r.Dir(), normal, x);

    if ( q->kr(x) != 0 ) {
        Vector3D reflectdir;
        q -> ReflectionDir(reflectdir, normal, -r.Dir(), x);
        c += q->kr(x) * Trace( Ray(x, reflectdir), d+1);
    }
    if ( q->kt(x) != 0 ) {
        Vector3D refractdir;
        if (q -> RefractionDir(refractdir, normal, -r.Dir(), x, out)) {
            c += q->kt(x) * Trace( Ray(x, refractdir), d+1);
        }
    }
    return c;
}
}
```

A képszintézis végrehajtása során minden egyes pixelközépponton keresztül egy sugarat indítunk az objektumtérbe, majd a sugárkövetés által számított színnek megfelelően kiszínezzük a pixelt.

```
//-----
void Scene :: Render( ) {
//-----
    for(int y = 0; y < YMAX; y++) {
        for(int x = 0; x < XMAX; x++) {
            Ray r = camera.GetRay(x, y);
            SColor col = Trace( r, 0 );
            WritePixel( x, y, col );
        }
    }
}
```


11. fejezet

Globális illuminációs algoritmusok

A *globális illuminációs algoritmusok* az árnyalási egyenletet (vagy a potenciál egyenletet) a benne lévő csatolás elhanyagolása nélkül oldják meg, ily módon képesek a többszörös fényvisszaverődések pontos kezelésére (emlékezzünk vissza, hogy a *lokális illuminációs algoritmusok* a többszörös visszaverődéseket egyáltalán nem vették figyelembe, a *rekurzív sugárkövetés* pedig csak a koherens komponensekre követte a fény útját). Formálisan a globális illuminációs algoritmusok az

$$L = L^e + \mathcal{T}L$$

integrálegyenlet (8.15) megoldása után kiszámolják a mérőeszközbe — tipikusan egy pixelbe — jutó fényteljesítményt a

$$P = \mathcal{M}L$$

operátor (8.6 egyenlet) alkalmazásával (vagy megoldják a potenciál egyenletet, és ez alapján határozzák meg a mérőeszközbe jutó teljesítményt).

A következőkben először az integrálegyenletek numerikus megoldásának általános elveivel foglalkozunk, majd konkrét globális illuminációs algoritmusokat ismertetünk.

11.1. Integrálegyenletek megoldása

A globális illuminációs algoritmusoknak egy integrálegyenletet kell numerikusan megoldaniuk, amelyet alapvetően három különböző módon tehetünk meg: inverzió, expanszió vagy iteráció felhasználásával.

11.1.1. Inverzió

Az *inverzió* az ismeretlen függvénytől függő tagokat az egyenlet egyik oldalán csoportosítja, majd formálisan egy inverziós műveletet hajt végre:

$$L = L^e + \mathcal{T}L \implies (1 - \mathcal{T})L = L^e \implies L = (1 - \mathcal{T})^{-1}L^e. \quad (11.1)$$

A mért teljesítmény tehát:

$$\mathcal{M}L = \mathcal{M}(1 - \mathcal{T})^{-1}L^e. \quad (11.2)$$

Sajnos a \mathcal{T} operátor egy integrál operátor, így nem invertálható közvetlenül (fájdalom, de az integrál jellel nem lehet osztani). Így a következő fejezetben ismertetendő véges-elem módszer segítségével az integráloperátort egy mátrixszal közelítjük, majd a keletkező lineáris egyenletrendszert a szokásos módszerekkel oldhatjuk meg.

11.1.2. Véges-elem módszer

A folytonos paraméterű függvények véges adattal történő közelítő megadására a *véges-elem módszert* (*finite-element method*) használhatjuk. Ennek lényege, hogy a függvényt függvénysorral közelítjük, azaz a következő alakban keressük:

$$L(p) \approx \sum_{j=1}^n L_j \cdot b_j(p), \quad (11.3)$$

ahol $b_j(p)$ előre definiált bázis függvények, L_j pedig skalár tényezők. Két függvény szorzatának a teljes térre vett integrálját a két függvény *skalárszorzatának* hívjuk, és a következőképpen jelöljük:

$$\langle f, g \rangle = \int f(p) \cdot g(p) dp.$$

Az egyszerűség kedvéért feltételezzük, hogy a különböző bázisfüggvények szorzatának a teljes térre vett integrálja zérus, tehát:

$$\langle b_i, b_j \rangle = \int b_i(p) \cdot b_j(p) dp = 0 \quad \text{ha } i \neq j.$$

A zérus skalárszorzat szemléletesen úgy is értelmezhető, hogy a bázisfüggvények egymásra “merőlegesek”, tehát ortogonális rendszert alkotnak.

Az egyik leggyakrabban használt bázisfüggvény készlet a közelítendő függvény értelmezési tartományát véges számú tartományra bontja, és az i . bázisfüggvényt 1 értékűnek tekinti az i . tartományban, minden más tartományban 0-nak. A megoldandó

$L(p) = L^e(p) + \mathcal{T}L(p)$ integrálegyenletbe a függvénysoros közelítést behelyettesítve a következő egyenletet kapjuk:

$$\sum_{j=1}^n \mathbf{L}_j \cdot b_j(p) = \sum_{j=1}^n \mathbf{L}_j^e \cdot b_j(p) + \mathcal{T} \sum_{j=1}^n \mathbf{L}_j \cdot b_j(p). \quad (11.4)$$

Szorozzuk meg skalárisan az egyenletet egyenként az összes bázisfüggvénnyel. Az ortogonalitási feltétel miatt az i . bázisfüggvénnyel való szorzás a következő egyenletre vezet:

$$\mathbf{L}_i = \mathbf{L}_i^e + \sum_{j=1}^n \frac{\langle \mathcal{T}b_j, b_i \rangle}{\langle b_i, b_i \rangle} \cdot \mathbf{L}_j. \quad (11.5)$$

A véges-elem módszer alkalmazása tehát a függvénysor együtthatóira egy lineáris egyenletrendszert eredményezett:

$$\mathbf{L} = \mathbf{L}^e + \mathbf{R} \cdot \mathbf{L}, \quad \text{ahol } \mathbf{R}_{ij} = \frac{\langle \mathcal{T}b_j, b_i \rangle}{\langle b_j, b_i \rangle}. \quad (11.6)$$

Erre a lineáris egyenletrendszerre az inverzió — például Gauss-elimináció alkalmazásával — már ténylegesen elvégezhető:

$$\mathbf{L} = \mathbf{L}^e + \mathbf{R} \cdot \mathbf{L} \implies (\mathbf{1} - \mathbf{R}) \cdot \mathbf{L} = \mathbf{L}^e \implies \mathbf{L} = (\mathbf{1} - \mathbf{R})^{-1} \mathbf{L}^e. \quad (11.7)$$

A függvénysor együtthatóiból viszont a függvényérték a $\sum_{j=1}^n \mathbf{L}_j \cdot b_j(p)$ képlet alkalmazásával tetszőleges p pontra megkapható.

11.1.3. Expanzió

Az *expanzió* az integrálegyenletben lévő csatolást rekurzív behelyettesítéssel oldja fel, amelynek eredményeképpen a megoldást egy végtelen *Neumann-sor* alakjában állítjuk elő. Helyettesítsük be a jobb oldali L függvénybe az $L^e + \mathcal{T}L$ alakú teljes jobb oldalt, ami az egyenlet szerint nyilván egyenlő L -lel:

$$L = L^e + \mathcal{T}L = L^e + \mathcal{T}(L^e + \mathcal{T}L) = L^e + \mathcal{T}L^e + \mathcal{T}^2L. \quad (11.8)$$

Ugyanezt ismételjük meg n -szer:

$$L = \sum_{i=0}^n \mathcal{T}^i L^e + \mathcal{T}^{n+1}L. \quad (11.9)$$

Mivel minden egyes visszaverődés csökkenti a teljes energiát, a \mathcal{T} operátor *kontrakció*, ezért $\lim_{n \rightarrow \infty} \mathcal{T}^{n+1}L = 0$, tehát:

$$L = \sum_{i=0}^{\infty} \mathcal{T}^i L^e. \quad (11.10)$$

A mért fénytelsítmény pedig:

$$\mathcal{M}L = \sum_{i=0}^{\infty} \mathcal{M}T^i L^e. \quad (11.11)$$

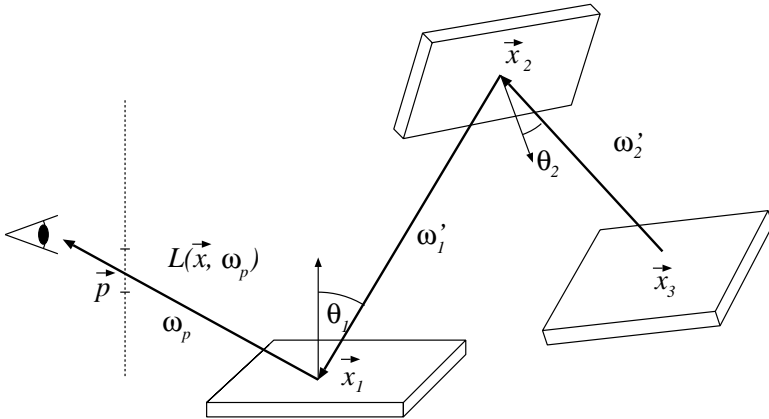
A végtelen sor egyes tagjainak intuitív jelentése van: $T^0 L^e = L^e$ a fényforrások direkt hozzájárulása a mért fénytelsítményhez, $T^1 L^e$ az egyszeres visszaverődésekből származik, $T^2 L^e$ a kétszeres visszaverődésekből, stb.

Vizsgáljuk meg a sor i . tagját ($T^i L^e$), amely egy i dimenziós integrál. Például $i = 2$ esetben

$$(T^2 L^e)(\vec{x}_1, \omega) = \int_{\Omega_1} \int_{\Omega_2} f_r(\omega'_1, \vec{x}_1, \omega) \cdot \cos \theta'_1 \cdot f_r(\omega'_2, \vec{x}_2, \omega'_1) \cdot \cos \theta'_2 \cdot L^e(\vec{x}_3, -\omega'_2) d\omega_2 d\omega_1, \quad (11.12)$$

ahol:

$$\begin{aligned} \vec{x}_2 &= h(\vec{x}, -\omega'_1), \\ \vec{x}_3 &= h(h(\vec{x}, -\omega'_1), -\omega'_2). \end{aligned} \quad (11.13)$$



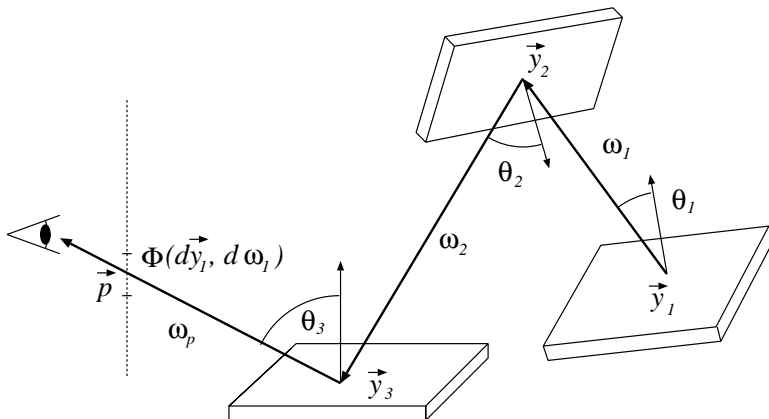
11.1. ábra. Az $T^2 L^e$ integrandusa egy két lépéses gyűjtőséta

Általában az integrandus értéke az $(\omega'_1, \omega'_2, \dots, \omega'_i)$ pontban a következőképpen határozható meg. A szemből a pixel középponton keresztül egy sugarat küldünk a térbe a látható pont meghatározására. Majd innen rekurzív módon folytatjuk a sugárkövetést ω'_1 irányba, a megtalált felületi pontból ω'_2 irányba, egészen az i . visszaverődésig. A visszaverődési lánc végén leolvassuk a felületi pont emisszióját, majd megszorozzuk az

egyes visszaverődések koszinuszos taggal súlyozott BRDF értékeivel. Az ilyen láncok elnevezése *gyűjtőséta* (*gathering walk*).

Vegyük észre, hogy egyetlen n hosszú visszaverődési láncot felhasználhatunk az 1-szeres, 2-szeres, \dots , n -szeres visszaverődési tagok becslésére, ha az emissziót nem csupán az utolsó pontban, hanem minden meglátogatott pontban leolvassuk.

A *potenciál egyenletet* szintén megoldhatjuk az *expanszió* segítségével. Ekkor egy $(\vec{y}, \omega'_1, \omega'_2, \dots, \omega'_i)$ pontban az integrandus értékét úgy kaphatjuk meg, hogy elindulunk egy fényforrásbeli \vec{y} pontból az ω_1 irányba, majd a megtalált pontból rekurzív az ω_2 , stb. ω_i irányba, végül az utolsó pontot összekötjük a szempozícióval. Ha az utolsó pont és a szem között nincs takaró objektum, akkor ez a fénypálya azon pixel színéhez járul hozzá, amelyet az utolsó pontot és a szemet összekötő szakasz metsz. Az úton szállított energia pedig a kezdeti pont emissziója szorozva a BRDF-ekkel, és az egyes visszaverődéseknél érvényes nézőirány és a normálvektor szögének koszinuszával. Ezen utak neve *lövőséta* (*shooting walk*).

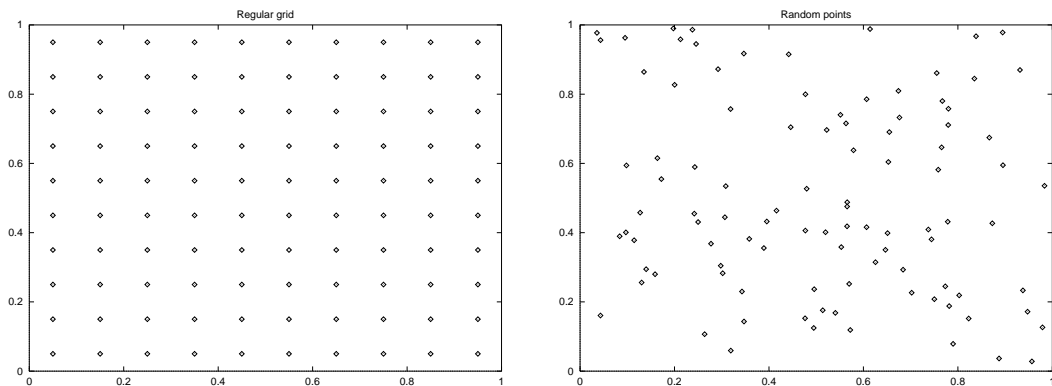


11.2. ábra. Az T^2W^e integrandusa egy két lépéses lövőséta

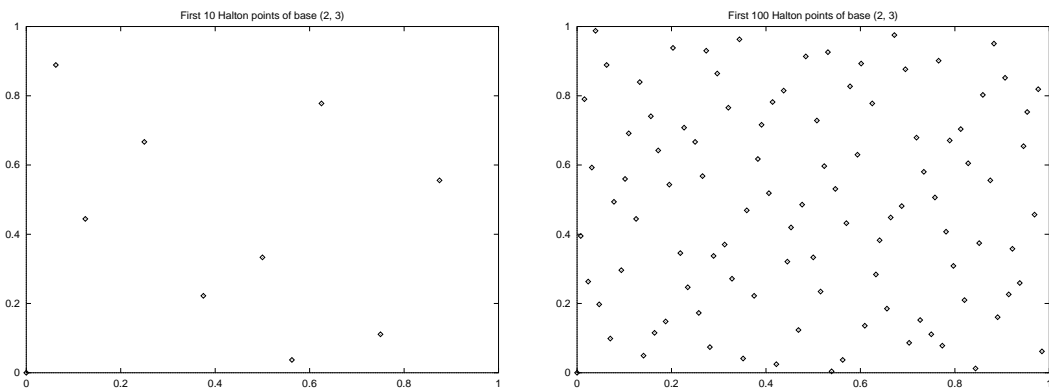
Megjegyezzük, hogy a koszinuszos tényezőben a gyűjtőséták a fényirány és a normálvektor szögét, a lövőséták pedig a nézeti irány és a normálvektor szögét használják. Másrészt a gyűjtősétákban a fényforrás szöge, a lövősétában pedig az utoljára meglátogatott felület és a szem iránya által bezárt szög koszinusza kiesik, így ezekkel nem kell szorozni.

11.1.4. Monte-Carlo integrálás

Az árnyalási egyenlet megoldása során igen magas dimenziós integrálokat kell kiértékelnünk, amelyhez numerikus integrálformulákat használhatunk. Egy *integrálformula*



11.3. ábra. 100 mintapont szabályos rácson (bal) és véletlenszerűen (jobb)



11.4. ábra. Az első 10 és 100 mintapont a Halton alacsony diszkrpanciájú sorozatból

általános alakja:

$$\int_V f(\mathbf{z}) \, d\mathbf{z} \approx \frac{1}{M} \cdot \sum_{i=1}^M f(\mathbf{z}_i) \cdot w(\mathbf{z}_i), \quad (11.14)$$

ahol w az integrálformulának megfelelő súlyfüggvény. Az alapvető kérdés az, hogy hol kell felvenni a \mathbf{z}_i mintapontokat ahhoz, hogy a minták számának növelésével gyorsan a tényleges megoldáshoz konvergáljunk.

A legismertebb lehetőség a mintapontok *szabályos rácson* történő elhelyezése, ami konstans $1/V$ súlyfüggvénnyel az integrál *téglányszabály* alkalmazásával történő kiértékeléséhez, nem konstans súlyfüggvényekkel pedig a *trapéz szabály*hoz illetve a *Simpson-szabály*hoz vezet. Sajnos ugyanolyan pontosság eléréséhez ezek a formulák egy 1 dimenziós integrálhoz M darab, egy 2 dimenziós integrálhoz M^2 darab, általában egy D dimenziós integrálhoz már M^D darab mintapontot igényelnek, azaz a számítási komplexitás az integrálási tartomány dimenziójának exponenciális függvénye. A jelenség magyarázata az, hogy magas dimenziókban a szabályos rács sorai és oszlopai között nagy űrök tátognak, ezért a mintapontok nem töltik ki elegendően sűrűn az integrálási tartományt (11.3. ábra). Az M^D darab mintapontigény elfogadhatatlan a magas dimenziójú integráloknál, ezért más stratégia után kell néznünk.

A mintapontokat megválaszthatjuk véletlenszerűen is. A következőképpen láthatjuk be, hogy asszimptotikusan ez is korrekt módon becsüli az integrál értékét. Szorozzuk be és egyszersmind osszuk is el az integrandust egy $p(\mathbf{z})$ valószínűség sűrűség függvénnyel (az osztás kioltja a szorzás hatását, tehát ez nyilván semmiféle változást nem okoz)! Majd ismerjük fel, hogy az így kapott integrál a várható érték képlete! A várható értéket pedig jól becsülhetjük a minták átlagával és a nagy számok törvénye szerint a becslés a tényleges várható értékhez tart. Formálisan:

$$\int_V f(\mathbf{z}) \, d\mathbf{z} = \int_V \frac{f(\mathbf{z})}{p(\mathbf{z})} \cdot p(\mathbf{z}) \, d\mathbf{z} = E \left[\frac{f(\mathbf{z})}{p(\mathbf{z})} \right] \approx \frac{1}{M} \cdot \sum_{i=1}^M \frac{f(\mathbf{z}_i)}{p(\mathbf{z}_i)}. \quad (11.15)$$

A becslés hibáját most a szórás fejezi ki. Legyen a $p(\mathbf{z})$ sűrűségfüggvényű \mathbf{z} valószínűségi változó $f(\mathbf{z})/p(\mathbf{z})$ transzformáltjának szórása σ . Ha a mintákat egymástól függetlenül választjuk ki, akkor az M minta átlagának szórása σ/\sqrt{M} , az integrálási tartomány dimenziójától függetlenül. A szórás és a klasszikus hiba fogalmát ugyancsak a nagy számok törvényei segítségével kapcsolhatjuk össze. Ezek szerint 0.997 valószínűséggel mondhatjuk, hogy M kísérlet elvégzése után az integrálbecslés hibája $3\sigma/\sqrt{M}$ -nél kisebb lesz.

A σ tényezőt úgy csökkenthetjük, hogy a $p(\mathbf{z})$ -t 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 mintavétel* (*importance sampling*).



11.5. ábra. A fontosság szerinti mintavétel hatása — A felső kép csak a koszinuszos taggal arányos sűrűségű mintákkal, az alsó pedig a BRDF és a koszinuszos taggal arányos valószínűségi sűrűség felhasználásával készült

A véletlen mintapontokkal dolgozó eljárást *Monte-Carlo módszernek* nevezzük, amelynek nagy előnye, hogy a komplexitása nem függ a tartomány dimenziójától [Sob91]. A véletlen pontsorozatok magasabb dimenzióban egyenletesebbek, mint a szabályos rácsok. Megjegyezzük, hogy léteznek olyan determinisztikus pontsorozatok, amelyek még a véletlen pontsorozatoknál is egyenletesebben töltik ki a rendelkezésre álló teret. A nagyon egyenletes eloszlás miatt *alacsony diszkrepanciájú sorozatoknak* nevezik őket (11.4. ábra) [Nie92, PFTV92, Knu81, Sob91].

Végtelen dimenziós integrálok kiértékelése

Az árnyalási egyenlet megoldásánál a kiértékelendő integrálok alakja $n = 1, 2, \dots, \infty$ értékre a következő:

$$(\mathcal{T}^n L^e)(\vec{x}_1, \omega) = \int_{\Omega_1} \dots \int_{\Omega_n} r_1 \cdot \dots \cdot r_n \cdot L^e d\omega_n \dots d\omega_1, \quad (11.16)$$

ahol r_i az i . visszaverődés BRDF-je és koszinusz tényezője. Elvileg végtelen sok ilyen integrált kellene kiértékelni, amelyek dimenziója szintén végtelenhez tart. Ezt nyilván nem tudjuk elvégezni, ezért valahogy határt kell szabni a számításoknak. Például mondhatjuk azt, hogy csak legfeljebb n_{\max} visszaverődésig vagyunk hajlandók szimulálni a fény útját, így csak n_{\max} darab integrált kell számítanunk, ahol az utolsó dimenziója $2n_{\max}$ (egy irányt két skalár jellemez). Ez az elhanyagolás nyilván torzítja a becslésünket.

Szerencsére egy ügyes trükkel kiküszöbölhetjük ezt a hibát. A jól ismert pisztolyos "játék" analógiája miatt *orosz rulettnek* nevezett módszer lényege a következő. Miután a bolyongás i . lépését megtettük, véletlenszerűen eldöntjük, hogy folytassuk-e a bolyongást vagy sem. Dobjunk fel egy kétforintost, amely p_i valószínűséggel esik arra az oldalra, hogy folytassuk a bolyongást és $1 - p_i$ valószínűséggel arra, hogy fejezzük be. Ha nem folytatjuk a bolyongást, az $n > i$ visszaverődésekből származó energia zérus. Ha viszont folytatjuk a bolyongást, akkor a véletlenszerűen elhanyagolt energia kompenzálása érdekében megszorozzuk a számított L^{in} értéket $1/p_i$ -vel. Várható értékben a becslés helyes lesz:

$$E[\tilde{L}^{in}] = p_i \cdot \frac{L^{in}}{p_i} + (1 - p_i) \cdot 0 = L^{in}. \quad (11.17)$$

A p_i valószínűséget általában úgy választjuk meg, hogy az az r_i súly integráljával, azaz az *albedo*val (8.23 egyenlet) azonos legyen.

11.1.5. Iteráció

Az *iteráció* alapja az a felismerés, hogy az árnyalási egyenlet megoldása a következő iterációs séma *fixpontja*:

$$L^{(m)} = L^e + \mathcal{T}L^{(m-1)}. \quad (11.18)$$

Tehát ha az iteráció konvergens, akkor bármely kezdeti függvényből a megoldáshoz konvergál. Az integráloperátor kontrakciós jellege miatt az iteráció konvergens, és a megoldás mindig létezik és egyértelmű. A mért fénytjeljesítményt ekkor határértékként állítjuk elő:

$$\mathcal{M}L = \lim_{m \rightarrow \infty} \mathcal{M}L^{(m)}. \quad (11.19)$$

A radianciafüggvény iteráció alatti tárolására véges-elem megközelítést alkalmazhatunk. Legyen most is:

$$L^{(m)} \approx \sum_{j=1}^n \mathbf{L}_j^{(m)} \cdot b_j(p). \quad (11.20)$$

Behelyettesítve a 11.18. egyenletbe, majd az egyenletet skalárisan szorozva a bázis-függvényekkel:

$$\mathbf{L}^{(m)} = \mathbf{L}^e + \mathbf{R} \cdot \mathbf{L}^{(m-1)}. \quad (11.21)$$

Ezzel lényegében a véges-elemes megközelítésből adódó lineáris egyenletrendszer iterációs megoldásához jutunk.

11.2. Diffúz eset: radiosity

Tekintsük azt az egyszerűsített esetet, amikor minden felület csak diffúz módon sugároz, és a fényforrások is csak diffúz emisszióra képesek. Ekkor a radiancia irányfüggetlen. A véges-elem megközelítéshez osszuk fel a felületeket kis elemi poligonokra! Az i . poligon területét és pontjainak halmazát jelöljük ΔA_i -vel. A bázis-függvények diffúz esetben szintén csak a pozíciótól függnének. Tehát a véges-elem közelítés formális alakja:

$$L(\vec{x}) \approx \sum_{j=1}^n \mathbf{L}_j \cdot b_j(\vec{x}) \quad (11.22)$$

ahol egy alkalmasan választott bázis:

$$b_i(\vec{x}) = \begin{cases} 1 & \text{ha } \vec{x} \in \Delta A_i, \\ 0 & \text{egyébként.} \end{cases} \quad (11.23)$$

A véges-elem módszerből kapott lineáris egyenletrendszer:

$$\mathbf{L} = \mathbf{L}^e + \mathbf{R} \cdot \mathbf{L} \quad \implies \quad (\mathbf{1} - \mathbf{R}) \cdot \mathbf{L} = \mathbf{L}^e. \quad (11.24)$$

ahol a 11.6 egyenlet szerint:

$$\mathbf{R}_{ij} = \frac{\langle T b_j(\vec{x}), b_i(\vec{x}) \rangle}{\langle b_i, b_i \rangle} = \frac{1}{\Delta A_i} \cdot \int_S \int_{\Omega} b_j(h(\vec{x})) \cdot f_r(\vec{x}) \cdot \cos \theta'_{\vec{x}} \cdot d\omega' \cdot b_i(\vec{x}) \, d\vec{x}. \quad (11.25)$$

A térszög definíciója szerint

$$b_j(h(\vec{x})) \cdot d\omega' = v(\vec{x}, \vec{y}) \cdot \frac{\cos \theta_{\vec{y}} \cdot d\vec{y}}{r^2},$$

ahol $r = |\vec{x} - \vec{y}|$ az \vec{x} és \vec{y} pontok távolsága, $v(\vec{x}, \vec{y})$ pedig a *láthatósági indikátor függvény*, amely 1 értéket vesz fel, ha a két pont látja egymást, illetve 0 értéket, ha a két pontot valami eltakarja egymás elől. Behelyettesítve a 11.25. egyenletbe:

$$\mathbf{R}_{ij} = \frac{1}{\Delta A_i} \cdot \int_S \int_S v(\vec{x}, \vec{y}) \cdot b_j(\vec{y}) \cdot b_i(\vec{x}) \cdot f_r(\vec{x}) \cdot \frac{\cos \theta'_{\vec{x}} \cdot \cos \theta_{\vec{y}}}{r^2} \, d\vec{y} \, d\vec{x}. \quad (11.26)$$

Vezessük be a diffúz reflektanciát a $\varrho = f_r \cdot \pi$ egyenlettel! A *diffúz reflektancia* a felület *albedoja*, amely azt fejezi ki, hogy a felület a beérkező energia hányad részét veri vissza a féltérbe.

Mivel az i . bázisfüggvény az i . felületeleмен 1, azon kívül pedig zérus, a 11.25. egyenlet a következő alakra hozható:

$$\mathbf{R}_{ij} = \frac{\langle T b_j(\vec{x}), b_i(\vec{x}) \rangle}{\langle b_i, b_i \rangle} = \frac{f_i}{\Delta A_i} \cdot \int_{\Delta A_i} \int_{\Delta A_j} v(\vec{x}, \vec{y}) \cdot \frac{\cos \theta'_{\vec{x}} \cdot \cos \theta_{\vec{y}}}{r^2} \, d\vec{y} \, d\vec{x} = \varrho_i \cdot F_{ij}. \quad (11.27)$$

Az \mathbf{R}_{ij} mátrixelem tehát két tényező, a BRDF és egy geometriától függő tag szorzata. A geometriai tényezőt *forma faktornak* nevezzük. A forma faktor a j . felületelemről kibocsátott energiának azt a hányadát adja meg, amely éppen az i . felületelemre jut:

$$F_{ij} = \frac{1}{\Delta A_i} \cdot \int_{\Delta A_i} \int_{\Delta A_j} v(\vec{x}, \vec{y}) \cdot \frac{\cos \theta'_{\vec{x}} \cdot \cos \theta_{\vec{y}}}{\pi \cdot r^2} \, d\vec{y} \, d\vec{x}. \quad (11.28)$$

Az F_{ii} formafaktor jelentése a felület által önmagára sugárzott energiahányad. Mivel az elemi felületek sík poligonok, az F_{ii} zérus.

A formafaktorok bevezetésével a radianciára vonatkozó 11.24 egyenletrendszer a következő alakú lesz:

$$\begin{bmatrix} 1 - \varrho_1 F_{11} & -\varrho_1 F_{12} & \dots & -\varrho_1 F_{1N} \\ -\varrho_2 F_{21} & 1 - \varrho_2 F_{22} & \dots & -\varrho_2 F_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ -\varrho_N F_{N1} & -\varrho_N F_{N2} & \dots & 1 - \varrho_N F_{NN} \end{bmatrix} \begin{bmatrix} L_1 \\ L_2 \\ \vdots \\ L_N \end{bmatrix} = \begin{bmatrix} L_1^e \\ L_2^e \\ \vdots \\ L_N^e \end{bmatrix}. \quad (11.29)$$

A lineáris egyenlet megoldásával a radianciát egyetlen hullámhosszon határozhatjuk meg. Színes képek előállításához legalább három különböző hullámhosszon kell a fényátadás elemzését elvégezni.

Összefoglalva a radiosity módszer alapvető lépései:

1. F_{ij} forma faktor számítás.
2. A fényforrások (L_i^e) emissziójának leolvasása a $\lambda_1, \lambda_2 \dots \lambda_n$ reprezentatív hullámhosszokon (legegyszerűbb esetben a vörös, zöld és kék szín hullámhosszain).
3. A lineáris egyenletrendszer megoldása Gauss-eliminációval vagy iterációval minden egyes reprezentatív hullámhosszra, amelynek eredményeként kapjuk a $L_i^{\lambda_1}, L_i^{\lambda_2} \dots L_i^{\lambda_n}$ értékeket.
4. A kép generálása a kameraparaméterek figyelembevételével elvileg bármely takarási algoritmus felhasználásával.

A módszer direkt végrehajtása szögletes képeket eredményez azon feltételezés miatt, hogy a felületelemek radianciája állandó. Ez a kellemetlen vizuális hatás eltüntethető *Gouraud-árnyalás* segítségével. A Gouraud-árnyalás alkalmazása esetén először az egy csúcspontban illeszkedő felületek radianciáinak az átlagát hozzárendeljük a csúcsponthoz. Majd minden felületelem belsejében a radianciát a csúcspontjainak a radianciáiból lineáris interpolációval határozzuk meg.

11.2.1. Forma faktor számítás

A forma faktorok meghatározásának több módszere is geometriai megfontolásokon alapul. Ezek az eljárások a forma faktor integrált valamely közvetítő felületen értékelik ki. A közvetítő felület lehet félgömb [GCT86], félkocka [CG85], tetraéder [BKP91], sík, stb. A geometriai módszerek a forma faktor kettős integráljából a külső integrált egyszerű, egyponos téglányszabállyal közelítik:

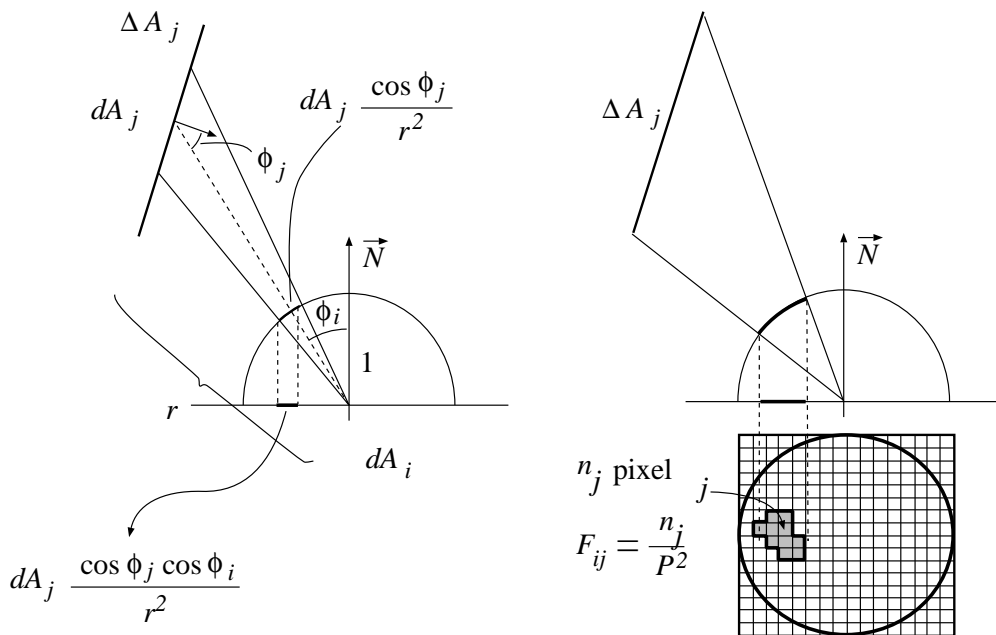
$$F_{ij} = \frac{1}{\Delta A_i} \cdot \int_{\Delta A_i} \int_{\Delta A_j} v(\vec{x}, \vec{y}) \cdot \frac{\cos \theta'_x \cdot \cos \theta'_y}{\pi \cdot r^2} d\vec{y} d\vec{x} \approx \int_{\Delta A_j} v(\vec{x}_i, \vec{y}) \cdot \frac{\cos \theta'_{\vec{x}_i} \cdot \cos \theta_j}{\pi \cdot r^2} d\vec{y}, \quad (11.30)$$

ahol \vec{x}_i az i . poligon középpontja.

Nusselt [SH81] észrevette, hogy ez a képlet azon alakzat területének π -ed részét határozza meg, amelyet a ΔA_j -nak az \vec{x}_i -ből látható pontjainak az \vec{x}_i fölé emelt félgömbre vetítésével, majd onnan a félgömb alapkörére történő továbbvetítésével kapunk. Tehát a formafaktorok számításához ezt a területet kell meghatározni. A félgömb központi szerepének elismeréseként az algoritmust *félgömb algoritmusnak* (*hemisphere*) nevezzük.



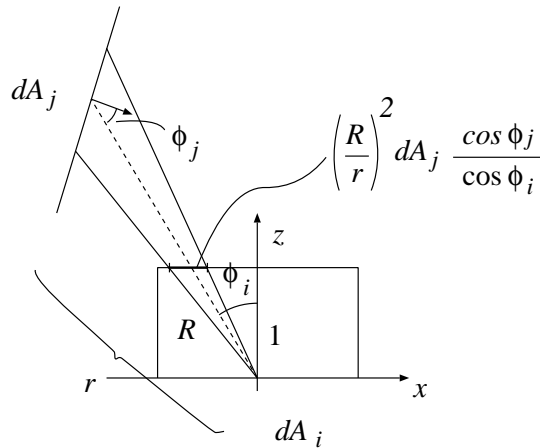
11.6. ábra. Radiosity módszerrel készült kép



11.7. ábra. A félgömb algoritmus geometriai háttere

Az i . poligonra az összes F_{ij} forma faktor meghatározható egyetlen olyan láthatósági számítással, ahol az ablak a félgömb felülete, a szempozíció pedig az i . poligon középpontja. Bontsuk fel az alapkör területét P^2 darab azonos területű kis "pixelre", és minden pixel félgömbre vetített pontján keresztül keressük meg a látható felületet. Ha a j . felület n_j darab pixelnek megfelelő irányban látszik, akkor az F_{ij} formafaktor becslése n_j/P^2 .

A bonyolult ablak miatt a félgömb algoritmus általában sugárkövetést használ, ami nem kellően gyors. Az ablak formája szerencsére egyszerűsíthető, és közvetítő felületként *félkocka* (*hemicube*) alakzatot [CG85] vagy akár *derékszögű tetraéder* [BKP91] alkalmazhatunk. A *félkocka algoritmusban* a láthatóságot a félkocka 5 oldalapjára, a *derékszögű tetraéder algoritmusban* a három oldalra kell meghatározni. Mivel ezek egyszerű téglalapok (a tetraéder esetén kiterjeszhető egyszerű téglalappá), az inkrementális képszintézisben megismert gyors láthatósági algoritmusok az eredeti formájukban használhatók.



11.8. ábra. Forma faktor számítás félkockával

A közvetítő geometria megváltoztatását kompenzálnunk kell a területszámítás során. Ezért ezekben az algoritmusokban nem csupán a pixelek számát összegezzük, hanem a pixelek helyének megfelelően súlyfüggvényeket használunk. A súlyfüggvények az egységnyi magasságú és 2 egység szélességű és mélységű félkocka z , y és x tengelyekre merőleges lapjain (11.8. ábra):

$$w_z = \frac{1}{\pi(x^2 + y^2 + 1)^2}, \quad w_y = \frac{z}{\pi(x^2 + z^2 + 1)^2}, \quad w_x = \frac{z}{\pi(z^2 + y^2 + 1)^2}. \quad (11.31)$$

A félkocka módszer z -bufferes takarási algoritmussal kombinált változata a következő-

képpen néz ki:

```

for i = 1 to N do for j = 1 to N do  $F_{ij} = 0$ 
for i = 1 to N do
szem =  $\Delta A_i$  középpontja
for k = 1 to 5 do // félkocka oldalaira
ablak = a félkocka kth lapja
for x = 0 to P - 1 do for y = 0 to P - 1 do  $pixel[x, y] = 0$ 
Z-BUFFER ALGORITMUS (a j. felület színe legyen j)
for x = 0 to P - 1 do for y = 0 to P - 1 do
if ( $pixel[x, y] > 0$ ) then  $F_{i,pixel[x,y]} += w_k(x - P/2, y - P/2)/P^2$ 
endfor
endfor
endfor
endfor

```

11.2.2. A lineáris egyenletrendszer megoldása

A radiosity módszer alkalmazása során adódó lineáris egyenletrendszert elvileg több különböző módszerrel is megoldhatjuk. Az egyik legkézenfekvőbb módszer, a *Gauss-elimináció* numerikusan instabil, és időigénye az ismeretlenek számának köbével arányos (a poligonszám gyakran a $10^4..10^6$ tartományba esik).

Az Gauss-elimináció helyett az *iterációt* használhatjuk:

$$\mathbf{L}^{(m+1)} = \mathbf{R} \cdot \mathbf{L}^{(m)} + \mathbf{L}^e. \quad (11.32)$$

Az iteráció konvergenciáját az biztosítja, hogy az \mathbf{R} mátrix ∞ -normája a maximális diffúz reflektanciát nem haladhatja meg, ami pedig fizikailag plauzibilis modelleknél 1-nél kisebb.

Ha a felületek saját emissziójával indulunk az iteráció egyes lépései mindig eggyel több visszaverődést építenek be a megoldásba:

$$\begin{aligned}
\mathbf{L}^{(0)} &= \mathbf{L}^e, \\
\mathbf{L}^{(1)} &= \mathbf{R} \cdot \mathbf{L}^{(0)} + \mathbf{L}^e = \mathbf{R} \cdot \mathbf{L}^e + \mathbf{L}^e, \\
\mathbf{L}^{(2)} &= \mathbf{R} \cdot \mathbf{L}^{(1)} + \mathbf{L}^e = \mathbf{R}^2 \cdot \mathbf{L}^e + \mathbf{R} \cdot \mathbf{L}^e + \mathbf{L}^e, \\
&\vdots \\
\mathbf{L}^{(m)} &= \mathbf{R} \cdot \mathbf{L}^{(m-1)} + \mathbf{L}^e = \mathbf{R}^m \cdot \mathbf{L}^e + \mathbf{R}^{m-1} \cdot \mathbf{L}^e + \dots + \mathbf{L}^e.
\end{aligned} \quad (11.33)$$

A konvergencia sebessége növelhető, ha a normál (ún. *Jacobi-iteráció*) helyett *Gauss-Seidel-iterációt* használunk vagy a *szukcesszív túlrelaxálás* módszerét alkalmazzuk [R76].

11.2.3. Progresszív finomítás

A *progresszív finomítás* [CCWG88] speciális iterációs eljárás, ami a numerikus matematikában *Southwell-iteráció* néven ismeretes.

A normál iteráció minden egyes lépése az összes poligon radianciáját továbbadja. Ez a teljes formafaktor mátrix előállítását igényli (elrettentésképpen, ha a poligonok száma 10^5 , a formafaktorok száma 10^{10}), ráadásul csak az időt pazaroljuk olyan poligonok esetében, amelyeknek a radianciája zérus, vagy legalábbis elhanyagolható. Ha viszont csak egy vagy néhány poligon radianciáját adnánk tovább, akkor a többi poligon radianciája elveszne az iteráció során, ami a végső képből energiahiányhoz vezetne. A megoldást olyan iterációs séma kidolgozása jelenti, ahol nyilvántartjuk, hogy melyik felületről, mennyi radianciát adtunk tovább, és mennyi vár még továbbadásra (így energia nem vész el az iteráció alatt) és egy iterációs lépésben csak annak a poligonnak a még át nem adott radianciáját szórjuk a térbe a többi felület felé, amelyeknek a még szét nem szórt energiája maximális. Mivel minden iterációs lépésben csak egyetlen poligon radianciáját löjük szét a térben, a formafaktor mátrixnak mindig csak egyetlen oszlopára van szükségünk, így a normál iteráció gigantikus memóriagigántól megszabadulhatunk.

Az j . poligonnak az U_j még szét nem szórt radianciájának a szétszórása az i . poligon még szét nem szórt radianciáját $\rho_j \cdot F_{ij} \cdot U_j$ értékkel növeli, azaz egyetlen poligon energiájának a szórásához a forma faktor mátrix egy oszlopát kell ismernünk. Egy félkocka lépéssel a forma faktor mátrix egy sorát tudjuk meghatározni, ezt a tudást azonban közvetlenül hasznosíthatjuk a mátrix oszlopának előállításakor is, hiszen:

$$F_{ji} \cdot \Delta A_j = F_{ij} \cdot \Delta A_i \implies F_{ij} = F_{ji} \cdot \frac{\Delta A_j}{\Delta A_i} \quad (i = 1, \dots, N) \quad (11.34)$$

Ezek alapján az iteratív algoritmus:

```

for j = 1 to N do Lj = Lje, Uj = Lje
do
  j = a maximális Uj · Aj értékkel rendelkező felület indexe
  Fj1, Fj2, ..., FjN számítása egy félkockával
  for i = 1 to N do
    ΔLi = ρi · Uj · Fji · ΔAj / ΔAi
    Ui += ΔLi
    Li += ΔLi
  endfor
  Uj = 0
  error = max{U1, U2, ..., UN}
while error > ε

```

Belátható, hogy ez az algoritmus is mindig konvergens, ha a diffúz reflektanciák egynél kisebbek [SKe95].

11.3. Véletlen bolyongáson alapuló algoritmusok

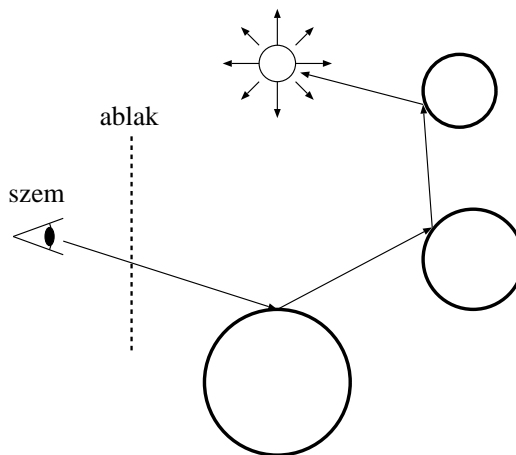
Az expansziós eljárás magas dimenziójú integrálok kiértékelésére vezet vissza az árnyalási (vagy potenciál) egyenlet megoldását. A magas dimenziójú integráloknál a klasszikus integrálformulák exponenciális komplexitása miatt Monte-Carlo eljárásokat kell alkalmaznunk. Ez azt jelenti, hogy az integrálszabály mintapontjait véletlenszerűen választjuk ki. Mivel az árnyalási egyenlet változó irányok, ez olyan sugárkövető eljárásra vezet, ahol a következő irányt véletlenszerűen választjuk ki. Intuitíve az eljárás olyan, mintha véletlenszerűen bolyonganánk a térben.

A Monte-Carlo integrálás elveinek ismertetésénél láttuk, hogy a véletlen irányokat célszerű olyan valószínűség eloszlásból generálni, ami arányos az integrandussal, azaz a radiancia, a BRDF valamint a felületi normális és az irány koszinuszának szorzatával. Sajnos a bejövő radianciát nem ismerjük (éppen azért számolunk, hogy ezt meghatározzuk), ezért a fontosság szerinti mintavételt általában csak a koszinuszos taggal súlyozott BRDF fontos irányai szerint végezzük el.

A véletlen bolyongáson alapuló algoritmusokat aszerint osztályozhatjuk, hogy azok az árnyalási egyenletet megoldó gyűjtősetákat tesznek, vagy pedig a potenciál egyenletet megoldó lövősetákat követnek.

11.3.1. Inverz fényűtkövetés

A Kajiya által javasolt *inverz fényűtkövetés* (*path tracing*) [Kaj86] véletlen gyűjtősetákkal dolgozik. A szempozícióból indulunk, akár a sugárkövetésnél, de most minden



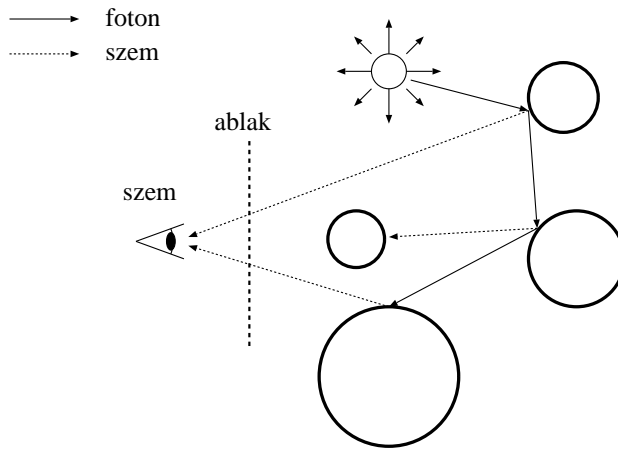
11.9. ábra. Inverz fényűtkövetés

egyes metszéspontnál véletlenszerűen választjuk ki a továbbhaladási irányt, mégpedig olyan valószínűség-sűrűségfüggvény szerint, ami arányos a BRDF és a kilépő szög koszinuszának a szorzatával. Minden lépés után az orosz rulett szabályai szerint, az albedonak megfelelő valószínűséggel folytatjuk a bolyongást.

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.

Fénykövetés

A *fénykövetés* (*light tracing*) [DLW93] lövősétákat alkalmaz. Az egyes séták kezdőpontját és irányát véletlenszerűen választjuk ki a fényforrások pontjaiból és a sugárzási irányjaiból. A fénysugár az 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 albedoval megegyező valószínűséggel folytatjuk.

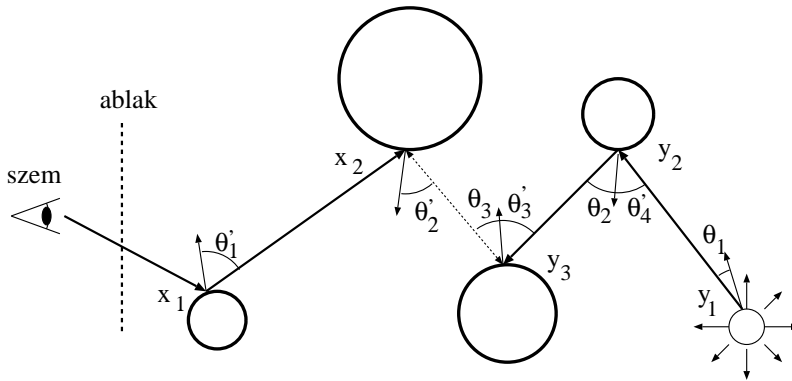


11.10. ábra. Fénykövetés

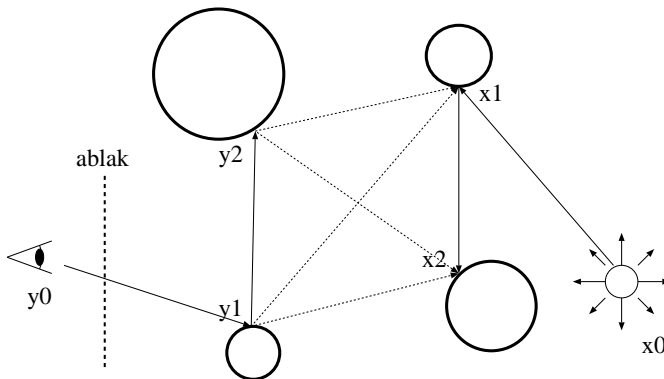
Minden visszaverődési pontot összekötünk a szempozícióval, és ellenőrizzük, hogy lehet-e ennek hatása valamely pixelre. Ha lehet, a pixel színéhez hozzáadjuk a visszaverődés hatását.

Kétirányú fényűtkövetés

A kétirányú fényűtkövetés (*bi-directional path tracing*) [LW93, VG95] az inverz fényűtkövetés és a fényűtkövetés kombinációja. Ez a módszer egyszerre indít egy gyűjtőjét és egy lövőjét majd a két séta végpontjait összeköti.



11.11. ábra. Kétirányú fényutak egyetlen összekötő sugárral



11.12. ábra. Kétirányú fényutak az összes lehetséges összekötő sugárral

Ha az összekötő sugár útjába más objektumok kerülnek, a fényút által szállított energia zérus, egyébként pedig a lövő séta végén érvényes teljesítményt kell radianciává alakítani, majd a gyűjtőjével a szembe szállítani. Az átalakításhoz a lövő séta teljesítményét a

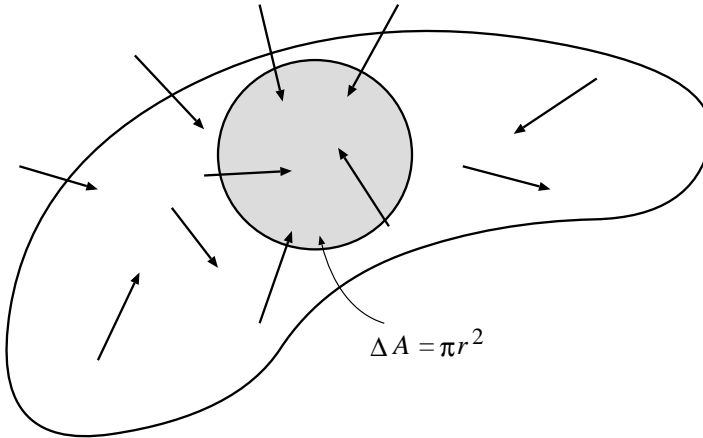
$$\frac{\cos \theta' \cdot \cos \theta}{r^2}$$

tényezővel kell szorozni, ahol az r az összekötő sugár hossza, θ' illetve θ pedig a gyűjtőséta és a lövőséta utolsó felületelemein érvényes normálvektorok és az összekötő sugár által bezárt szögek.

Az algoritmust tovább javíthatjuk, ha nem csak a két séta végpontjait, hanem az összes pontját összekötjük (11.12. ábra).

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 megpróbálnánk összekötni. Kívánságunkat a foton térképek [JC95, Jen96, JC98] alkalmazásával teljesíthetjük. A *foton térkép* (*photon-map*) olyan adatstruktúra, amely a sok lövőséta hatását tömören tárolja.



11.13. ábra. Foton térkép

A foton térkép a foton találatok gyűjteménye. Egy találatot a foton által a különböző hullámhosszokon szállított energiával (ez nem fizikai foton, ami csak egy hullámhosszon visz energiát), a találat helyével, a foton érkezési irányával és a felületi normálissal együtt tárolunk. A foton találatokat a hatékony előkeresés érdekében *kd-fa* adatstruktúrába szervezzük. A gyűjtőséták alatt az árnyalási egyenlet következő közelítésével dolgozunk:

$$L(\vec{x}, \omega') = \int_{\Omega} L(h(\vec{x}, -\omega'), \omega') \cdot f_r(\omega', \vec{x}, \omega) \cdot \cos \theta' d\omega' =$$

$$\int_{\Omega} \frac{d\Phi(\omega')}{dA \cos \theta' d\omega'} \cdot f_r(\omega', \vec{x}, \omega) \cdot \cos \theta' d\omega' \approx \sum_{i=1}^n \frac{\Delta\Phi(\omega'_i)}{\Delta A} \cdot f_r(\omega'_i, \vec{x}, \omega), \quad (11.35)$$

ahol $\Delta\Phi(\omega'_i)$ a ΔA felületre a ω'_i irányból érkező foton energiája. A $\Delta\Phi$ és a ΔA mennyiségeket a \vec{x} pont környezetében található foton találatok tulajdonságaiból approximáljuk a következő eljárással: Az \vec{x} köré egy gömböt teszünk, amelyet addig pumpálunk, amíg az éppen n foton találatot tartalmaz (az n az algoritmus globális paramétere). Ha ekkor a gömb sugara r , akkor a felületelem területe $\Delta A = \pi r^2$.

11.4. Program: inverz fényűtkövetés

A következőkben a sugárkövető programunkat úgy módosítjuk, hogy a metszéspontokban véletlenszerűen generálja a következő irányt. A fontosság szerinti mintavétel alkalmazásához a folytatási irány valószínűségi sűrűsége a BRDF és a koszinuszos tényező szorzatával arányos, ezért először a BRDF modelleket egészítjük ki egy-egy Reflection tagfüggvénnyel, amely előállítja a megfelelő L véletlen irányt és visszaadja ennek a valószínűségét is. A UNIFORM(i) makro Monte-Carlo algoritmusoknál a $[0, 1]$ intervallumba eső véletlen számokat állít elő. Alacsony diszkrepanciájú sorozatoknál azonban a UNIFORM(i) a i . független sorozat következő elemét adja vissza.

```
//=====
class DiffuseMaterial : virtual public Material {
//=====
    SColor Kd;
public:
    DiffuseMaterial( SColor kd0 ) : Kd(kd0) { }
    SColor& kd() { return Kd; }
    SColor BRDF( Vector3D& L, Vector3D& N, Vector3D& V ) { return Kd; }

    double Reflection( Vector3D& L, Vector3D& N, Vector3D& V, int d ) {
        double u = UNIFORM(3*d + 3), v = UNIFORM(3*d + 4);

        double theta = asin(sqrt(u)), phi = M_PI * 2.0 * v;
        Vector3D O = N % Vector3D(0, 0, 1);
        if (O.Length() < EPSILON) O = N % Vector3D(0, 1, 0);
        Vector3D P = N % O;
        L = N * cos(theta) + O * sin(theta) * cos(phi) +
            P * sin(theta) * sin(phi);
        double prob = cos(theta) / M_PI;
        return prob;
    }
    double AverageAlbedo( Vector3D& N, Vector3D& V ) {
        return Kd.Luminance() * M_PI;
    }
};
```

```
//=====
class SpecularMaterial : virtual public Material {
//=====
    SColor Ks;
    double shine;
public:
    SpecularMaterial( ) : Ks(0) { shine = 10; }
    SColor& ks() { return Ks; }
    double& Shine( ) { return shine; }
    SColor BRDF(Vector3D& L, Vector3D& N, Vector3D& V);

    double Reflection(Vector3D& L, Vector3D& N, Vector3D& V, double d) {
        double u = UNIFORM(3*d + 3), v = UNIFORM(3*d + 4);
        double cos_ang_V_R = pow(u, 1.0/(shine+1) );
        double sin_ang_V_R = sqrt( 1.0 - cos_ang_V_R * cos_ang_V_R );

        Vector3D O = V % Vector3D(0, 0, 1);
        if (O.Length() < EPSILON) O = V % Vector3D(0, 1, 0);
        Vector3D P = O % V;
        Vector3D R = O * sin_ang_V_R * cos(2.0 * M_PI * v) +
                    P * sin_ang_V_R * sin(2.0 * M_PI * v) +
                    V * cos_ang_V_R;
        L = N * (N * R) * 2.0 - R;
        double cos_ang_N_L = N * L;
        if (cos_ang_N_L < 0) return 0;
        double prob = (shine+1)/2/M_PI * pow(cos_ang_V_R, shine);
        return prob;
    }
    double AverageAlbedo( Vector3D& N, Vector3D& V ) {
        return ks().Luminance();
    }
};
```

Az egyszerre többféle visszaverődési tulajdonságot mutató anyagokat úgy kezelhetjük, hogy a visszaverődés során véletlenszerűen választunk egy visszaverődési típust és a következő sugárirányt ezen típusnak megfelelően határozzuk meg. A fontosság szerinti mintavételezés elve szerint a modellek között az átlag albedojuk arányában kell választani. A `GeneralMaterial` osztály `SelectReflectionModel` tagfüggvénye az átlagalbedo szerint véletlenszerűen választja ki a következő `Reflection` függvény által felhasznált visszaverődés típust (a választás eredménye a `selected` változóba kerül). Az ily módon generált irányból érkező fényt természetesen a kiválasztott típusnak megfelelő BRDF-el kell szorozni. Mivel annak valószínűsége zérus, hogy az ideális visszaverődés vagy törés éppen egy pont- vagy iránytípusú absztrakt fényforrást talál telibe, a direkt megvilágítás számításához csak a diffúz és spekuláris visszaverődés összegét kell figyelembe venni. Ehhez a `DeselectReflectionModel` függvény alaphelyzetbe állítja a `selected` változót.

```

//=====
class GeneralMaterial : public DiffuseMaterial,public SpecularMaterial,
                       public IdealReflector, public IdealRefractor {
//=====
    enum {NO, DIFFUSE, SPECULAR, REFLECTOR, REFRACTOR, ALL} selected;
public:
    GeneralMaterial( ) { selected = ALL; }
    double SelectReflectionModel(Vector3D& L, Vector3D& N,
                                Vector3D& V, int d) {
        double akd = DiffuseMaterial :: AverageAlbedo(N, V);
        double aks = SpecularMaterial :: AverageAlbedo(N, V);
        double akr = kr().Luminance();
        double akt = kt().Luminance();
        double r = UNIFORM(3*d + 2);
        if ((r -= akd) < 0) { selected = DIFFUSE; return akd; }
        if ((r -= aks) < 0) { selected = SPECULAR; return aks; }
        if ((r -= akr) < 0) { selected = REFLECTOR; return akr; }
        if ((r -= akt) < 0) { selected = REFRACTOR; return akt; }
        selected = NO; return 0.0; // orosz rulett
    }
    double Reflection(Vector3D& L, Vector3D& N, Vector3D& V,
                      BOOL out, int d) {
        switch (selected) {
        case DIFFUSE: return DiffuseMaterial :: Reflection(L, N, V, d);
        case SPECULAR: return SpecularMaterial :: Reflection(L, N, V, d);
        case REFLECTOR: return (double)ReflectionDir(L, N, V);
        case REFRACTOR: return (double)RefractionDir(L, N, V, out);
        default: return 0;
        }
    }
    void DeselectReflectionModel( ) { selected = ALL; }
    SColor BRDF(Vector3D& L, Vector3D& N, Vector3D& V) {
        double cost;
        switch (selected) {
        case DIFFUSE: return DiffuseMaterial :: BRDF(L, N, V);
        case SPECULAR: return SpecularMaterial :: BRDF(L, N, V);
        case REFLECTOR: cost = N * L;
            if (cost > EPSILON) return (kr( ) / cost);
            else return SColor(0);
        case REFRACTOR: cost = -(N * L);
            if (cost > EPSILON) return (kt( ) / cost);
            else return SColor(0);
        case ALL: return (DiffuseMaterial :: BRDF(L, N, V) +
                          SpecularMaterial :: BRDF(L, N, V));
        default: return SColor(0);
        }
    }
};

```

Az *inverz fényűtkövetés* módszerét a `PathTrace` tagfüggvényben implementáltuk, amely a fényút irányait rekurzív módon véletlenszerűen határozza meg. A sugár színének meghatározásához először megkeressük a sugár kezdőpontjához legközelebbi felületi pontot és ebben a pontban kiszámítjuk a saját emissziót, valamint az ambiens fény és a direkt fényforrások hatását. Ezután az átlagalbedoknak megfelelően visszaverődési modellt választunk, majd a választott modellnek megfelelően véletlen fényirányt állítunk elő. A fényirányból érkező fényt a `PathTrace` rekurzív hívásával számítjuk ki, amit a választott modell szerinti BRDF-fel szorzunk és a fontosság szerinti mintavételezés képletének megfelelően a kiválasztási valószínűséggel osztunk.

```
//-----
SColor Scene :: PathTrace(Ray r, int d) {
//-----
    if (d > MAXDEPTH) return La;
    Point3D x;
    Primitive3D * q = Intersect(r, x);
    if (q == NULL) return La;

    Vector3D normal = q -> Normal(x);
    BOOL out = TRUE;
    if ( normal * (-r.Dir()) < 0) { normal = -normal; out = FALSE; }

    SColor c = q -> Le(x, -r.Dir()) + q->ka(x) * La;
    q -> DeselectReflectionModel( );
    c += DirectLightsource(q, -r.Dir(), normal, x);

    double prob = q->SelectReflectionModel(normal, -r.Dir(), x );
    if (prob < EPSILON) return c; // orosz rulett

    Vector3D newdir;
    prob *= q->Reflection( newdir, normal, -r.Dir(), x, out );
    if (prob < EPSILON) return c;

    double cost = newdir * normal;
    if (cost < 0) cost = -cost;
    if (cost > EPSILON) {
        SColor w = q->BRDF(newdir, normal, -r.Dir(), x) * cost / prob;
        if (w.Luminance() > EPSILON)
            c += PathTrace( Ray(x, newdir), d+1) * w;
    }
    return c;
}
}
```


Az inverz fényűtkövetés az egyes pixelekhez tartozó fényutak hatásának átlagolásával határozza meg a pixelek színét. Most az első irány nem a pixel középpontján megy keresztül, hanem a pixel területén egyenletes valószínűségeloszlás szerint véletlenszerűen választjuk.

```
//-----  
void Scene :: MonteCarloRender( TGAOutputFile& output ) {  
//-----  
    for(int y = 0; y < YMAX; y++) {  
        for(int x = 0; x < XMAX; x++) {  
            SColor col(0);  
            for( int i = 0; i < NSAMPLE; i++ ) {  
                double dx = UNIFORM(0) - 0.5;  
                double dy = UNIFORM(1) - 0.5;  
                Ray r = camera.GetRay(x + dx, y + dy);  
                col += PathTrace( r, 0 );  
            }  
            col /= NSAMPLE;  
            WritePixel( x, y, col );  
        }  
    }  
}
```


12. fejezet

Raszteres képek csipkézetttségének a csökkentése

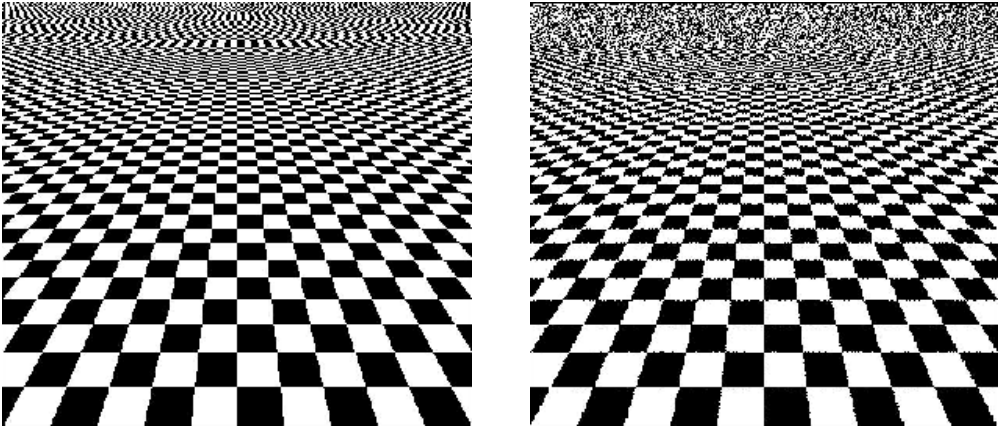
A képszintézis *raszterizációs* lépése a folytonos geometriát diszkrét pontokban mintavételezi, majd a képet kis téglalapokból rakja össze. A *mintavételezési eljárás* minősége a felbontástól függ, de tökéletes sohasem lehet, ugyanis a geometriában hirtelen változások is előfordulhatnak, aminek következményeként a kép Fourier-transzformáltjában tetszőlegesen magas frekvenciák is megjelennek. Mint ismeretes, a mintavételezés következményeként az eredeti jel Fourier-transzformáltja periodikusan megismétlődik, így az eredetileg magas frekvenciájú komponensek “álsruhában” alacsony frekvencián is feltűnhetnek. Ezt nevezzük *alias* jelenségnek. A mintavételezett jel analóg rekonstrukciója a kép konstans színű téglalapokkal történő előállításával történik, ami ugyancsak messze van az optimálistól. A mintavételezési és visszaállítási hibák együttesen a raszteres képek *csipkézetttségéhez* vezetnek.

A hibák csökkentésére a következő lehetőségek állnak rendelkezésre:

1. A rasztertár és a képernyő felbontásának, azaz a mintavételi frekvenciának a növelése. Ennek a megközelítésnek erős technológiai korlátai vannak, ráadásul az emberi szem igen érzékeny a csipkék periodikus mintázataira, így még nagy felbontásnál is kiszúrja ezeket.
2. Az *alias* jelenséget kiküszöbölhetjük, ha a geometriát a mintavételezés előtt szűrjük, annak érdekében, hogy a Fourier-transzformáltja sávkorlátozott legyen. Ez az eljárás a raszterizációt valamilyen aluláteresztő szűrővel kombinálja. A nagyfrekvenciás viselkedést persze ez a módszer is torzítja, de legalább megakadályozza, hogy a mintavételi frekvencia felénél nagyobb frekvenciájú komponensek az alacsonyfrekvenciás tartományban is feltűnjenek. Ezt a megközelítést *előszűrésnek* nevezzük.
3. A képet a rasztertár felbontásánál nagyobb felbontással számítjuk, majd a beírás

pillanatában a pixelhez tartozó szubpixelek színtartalmait átlagoljuk. A módszer neve *tűlmintavételezés* illetve *utószűrés*.

4. A mintavételi pontokat nem egy szabályos rácsból, hanem véletlenszerűen választjuk ki. Ezzel a mintavételi hibák periodikus jellegzetességeit véletlen zajjal cseréljük ki, amely a szem számára sokkal kevésbé zavaró [Coo86, SKe95, SK95]. Ezt *sztochasztikus mintavételezésnek* nevezzük.



12.1. ábra. Sakktábla szabályos ráccsal történő mintavételezéssel (bal) és sztochasztikus mintavételezéssel (jobb)

12.1. Előszűrés

Tegyük fel, hogy az előállítandó képet a megjelenítés során az x tengely mentén Δx periódussal, az y tengely mentén pedig Δy periódussal mintavételezzük (a képernyő-kordinátarendszerben $\Delta x = 1, \Delta y = 1$). A *mintavételi törvény* szerint a *mintavételezési frekvencia* felénél nagyobb frekvenciák kiszűrése szükséges az alias hatás megszüntetéséhez. A szűrés a frekvenciatartomány helyett a képsíkon is elvégezhető, ha a szűrő súlyfüggvényével konvolváljuk a jelet. Legyen az eredeti jel $I(x, y)$, a szűrő súlyfüggvénye pedig $f(x, y)$. A szűrt jel a következő konvolúciós integrállal állítható elő:

$$I_f(x, y) = I(x, y) * f(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(t, \tau) \cdot f(x - t, y - \tau) dt d\tau. \quad (12.1)$$

Az ideális aluláteresztő szűrő súlyfüggvénye a sinc függvény:

$$f(x, y) = \frac{\sin(x \cdot \pi / \Delta x)}{x \cdot \pi / \Delta x} \cdot \frac{\sin(y \cdot \pi / \Delta y)}{y \cdot \pi / \Delta y} = \text{sinc}\left(\frac{x \cdot \pi}{\Delta x}\right) \cdot \text{sinc}\left(\frac{y \cdot \pi}{\Delta y}\right) \quad (12.2)$$

Sajnos az ezt a függvényt alkalmazó konvolúciós integrál meglehetősen bonyolult, ráadásul negatív — azaz a képernyőn nem ábrázolható — színeket is eredményezhet. Ezért a gyakorlatban közelítő aluláteresztő szűrőket alkalmazunk.

A két legfontosabb szűrőtípus a

1. *doboz szűrő*:

$$f(x, y) = \begin{cases} 1 & \text{ha } |x| < \Delta x/2 \text{ és } |y| < \Delta y/2, \\ 0 & \text{egyébként.} \end{cases} \quad (12.3)$$

2. *kúp szűrő*:

$$f(x, y) = \begin{cases} (1 - r) \cdot 3/\pi & \text{ha } r < 1, \\ 0 & \text{egyébként,} \end{cases} \quad (12.4)$$

$$\text{ahol } r(x, y) = \sqrt{(x/\Delta x)^2 + (y/\Delta y)^2}.$$

A doboz szűrő alkalmazása után az X, Y pixel középpontban a jel értéke:

$$I_{\text{box}}(X, Y) = \int_{X-0.5}^{X+0.5} \int_{Y-0.5}^{Y+0.5} I(t, \tau) dt d\tau. \quad (12.5)$$

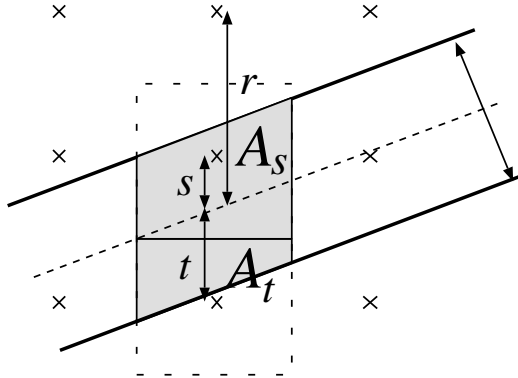
Ha a pixelt metsző primitívek (például szakaszok, poligonok, stb.) közül a p . színe I_p , a metszet területe pedig A_p , akkor a pixel szűrt színe:

$$I_{\text{box}}(X, Y) = \sum_{p=1}^P I_p \cdot A_p. \quad (12.6)$$

12.1.1. A szakaszok csipkézettségének csökkentése

Építsünk be doboz szűrőt a szakaszrajzoló algoritmusba! A 12.6. egyenlet értelmében a pixelek intenzitásait úgy határozhatjuk meg, hogy egy 1 pixel szélességű szakaszt — ún. vonalzó — a rasterhálóra teszünk, és a pixelek színében a szakasz színét a pixelekkel alkotott közös résznek megfelelően súlyozzuk.

Csak az enyhén emelkedő szakaszokkal foglalkozunk. A 12.2. ábra alapján kijelenthetjük, hogy egyetlen oszlopban maximum három pixel metszheti a vonalzónkat. Jelöljük a három pixel és a szakasz közötti függőleges távolságokat r -rel, s -sel t -vel, és tegyük fel, hogy $s < t \leq r$! Egyszerű geometriai megfontolások alapján fennállnak a $s, t < 1, s + t = 1$ és $r \geq 1$ relációk.



12.2. ábra. Szakasz doboz szűrése

A A_s, A_t és A_r területek nem csupán az r, s és t távolságoktól, hanem a szakasz meredekségétől is függnek, azonban ezt a következő közelítésekkel kiküszöbölhetjük:

$$A_s \approx (1 - s), \quad A_t \approx (1 - t), \quad A_r \approx 0. \quad (12.7)$$

Az $y = m \cdot x + b$ szakaszra az s és t paramétereket a következőképpen számíthatjuk ki:

$$s = m \cdot x + b - \text{Round}(m \cdot x + b) = \text{Error}(x) \quad \text{és} \quad t = 1 - s \quad (12.8)$$

ahol az $\text{Error}(x)$ a raszteres közelítés hibája.

A doboz szűrő alkalmazásával a két legközelebbi pixel színe:

$$I_s = I \cdot (1 - \text{Error}(x)), \quad I_t = I \cdot \text{Error}(x), \quad (I \text{ az } R, G \text{ és } B \text{ komponenseket jelenti}). \quad (12.9)$$

Az inkrementális elv ezen képletek kiértékelését is egyszerűsíti. Az inkrementális képletek arra az esetre, amikor az y koordinátát nem léptetjük:

$$I_s(x + 1) = I_s(x) - I \cdot m, \quad I_t(x + 1) = I_t(x) + I \cdot m. \quad (12.10)$$

Az inkrementális képletek arra az esetre, amikor az y koordinátát léptetjük:

$$I_s(x + 1) = I_s(x) - I \cdot m + I, \quad I_t(x + 1) = I_t(x) + I \cdot m - I. \quad (12.11)$$

Végül a Bresenham-algoritmus csipkézettség csökkentővel kiegészített változata:

AntiAliasedBresenhamLine(x_1, y_1, x_2, y_2, I)

$$\Delta x = x_2 - x_1, \quad \Delta y = y_2 - y_1$$

$$E = -\Delta x$$

$$dE^+ = 2(\Delta y - \Delta x), \quad dE^- = 2\Delta y$$

$$dI^- = \Delta y / \Delta x \cdot I, \quad dI^+ = I - dI^-$$

$$I_s = I + dI^-, \quad I_t = -dI^-$$

$$y = y_1$$

for $x = x_1$ to x_2 do

 if $E \leq 0$ then $E += dE^-, I_s -= dI^-, I_t += dI^-$

 else $E += dE^+, I_s += dI^+, I_t -= dI^+, y++$

 Add Frame Buffer(x, y, I_s)

 Add Frame Buffer($x, y + 1, I_t$)

endfor

Az “Add Frame Buffer(x, y, I)” rutin a rasztertár tartalmával átlagolja a szakasz színét:

$$\text{color}_{\text{old}} = \text{frame_buffer}[x, y]$$

$$\text{frame_buffer}[x, y] = \text{color}_{\text{line}} \cdot I + \text{color}_{\text{old}} \cdot (1 - I)$$

Ezeket a sorokat az R, G, B komponensekre külön kell végrehajtani.

12.3. ábra. Normál, doboz szűrővel [SKM94] és kúp szűrővel [GSS81] szűrt szakaszok

12.2. Utószűrés

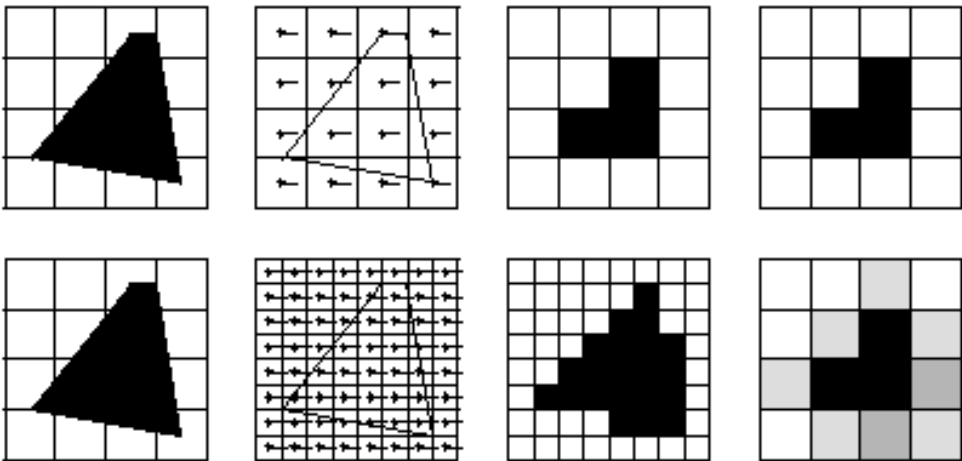
Az *utószűrés* a rasztertár felbontásánál nagyobb felbontással számítja ki a képet ($\Delta x = 1/N, \Delta y = 1/N$), majd valamilyen digitális szűrőalgoritmussal határozza meg a tényleges pixelek színét.

$$I_{sf}(X, Y) = \sum_i \sum_j I(i \cdot \Delta x, j \cdot \Delta y) \cdot f(X - i \cdot \Delta x, Y - j \cdot \Delta y) \quad (12.12)$$

Az egyik legegyszerűbb szűrő a doboz szűrő digitális változata:

$$I_{\text{box}}(X, Y) = \frac{1}{(N+1)^2} \sum_{i=-N/2}^{N/2} \sum_{j=-N/2}^{N/2} I(X - i \cdot \Delta x, Y - j \cdot \Delta y) \quad (12.13)$$

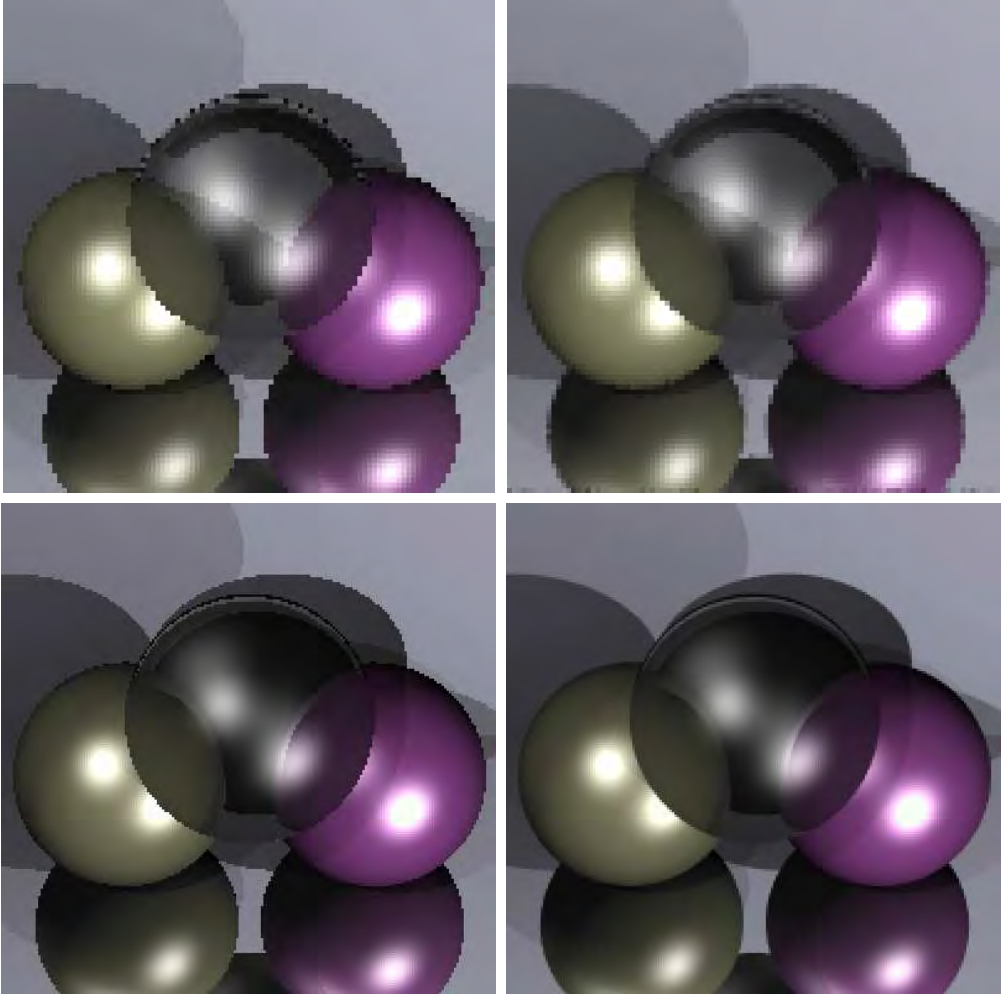
Ezen képlet azt mondja, hogy a pixel színét a pixelhez tartozó *szubpixelek* színértékeinek az átlagaként kaphatjuk meg.



12.4. ábra. Polygon rajzolás utószűrés nélkül (felső sor) és utószűréssel (alsó sor)

Az utószűrés jól használható 2D területek és 3D felületek raszterizációja során. A képszintézis algoritmust változtatás nélkül végrehajtjuk a nagyobb felbontásra, majd a rasztertárba írás pillanatában megvalósítjuk az utószűrés műveletét.

Amennyiben a szubpixeleket a pixelen belül nem szabályos rácson, hanem véletlenszerűen helyezük el, a *sztochasztikus mintavételezés* és az utószűrés házasságához jutunk. A módszert gyakran használják a *sugárkövetéssel* együtt, hiszen az nem igényli a mintavételi pontok szabályos rácsta szervezését.



12.5. ábra. 100×100 felbontással (felső sor) és 200×200 felbontással (alsó sor) készült képek. A bal oldalon lévő képeknél nem használtunk csipkézettség csökkentést, a jobb oldali kép viszont sztochasztikus mintavételezés és utószűrés kombinációját használó csipkézettség csökkentő eljárással készült. A pixelek színét 10 mintából doboz szűrővel számítottuk.

12.3. Program: sugárkövetés kiegészítése csipkézetttség csökkentéssel

A sugárkövetés könnyen kiegészíthető a sztochasztikus mintavételezés és az utószűrés kombinációját alkalmazó csipkézetttségcsökkentő algoritmussal. Ekkor egyetlen pixel színét nem egyetlen sugár követésével számítjuk, hanem több olyan sugárból kapott szín átlagából, amelyeknek a dőfési pontja a pixel területén egyenletes eloszlású. Az alábbi programban az `UNIFORM(i)` jelenthet egy $[0, 1]$ tartományban egyenletes eloszlású valószínűségi változót, vagy az i . független alacsony diszkrepanciájú sorozatot is.

```
#define NSAMPLE 10

//-----
void Scene :: AntiAliasRender( ) {
//-----
    for(int y = 0; y < camera.Viewport().Top(); y++) {
        for(int x = 0; x < camera.Viewport().Right(); x++) {
            SColor col(0);
            for( int i = 0; i < nsample; i++ ) {
                double dx = UNIFORM(0) - 0.5;
                double dy = UNIFORM(1) - 0.5;
                Ray r = camera.GetRay(x + dx, y + dy);
                col += Trace( r, 0 );
            }
            col /= nsample;
            WritePixel( x, y, col );
        }
    }
}
```

13. fejezet

Textúra leképzés

Az árnyalási egyenletben szereplő BRDF nem szükségképpen állandó a felületen, hanem pontról pontra változhat. Változó BRDF-ek segítségével finom részleteket, ún. textúrát tudunk megjeleníteni anélkül, hogy a felületek geometriáját túlságosan elbonyolítanánk.

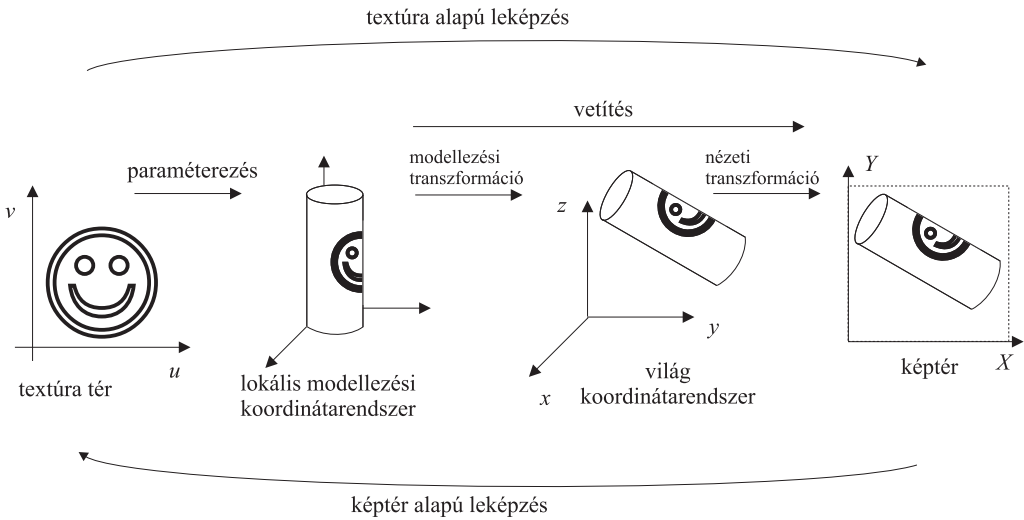
A változó optikai paramétereket általában egy, a geometriától független koordináta-rendszerben, a *textúratér*ben tároljuk. Ebben a térben a textúra megadható függvényként, vagy akár tömbben tárolt adathalmazként is. Az utóbbi esetben egyetlen tömbelem neve textúra elem, vagy röviden *texel*. A textúratér pontjait és a lokális modellezési koordináta-rendszerben definiált objektumok felületi pontjait egy transzformációval kell összerendelni. Ezt a transzformációt *paraméterezésnek* nevezzük.

A modellezési transzformáció a lokális modellezési koordináta-rendszert a világ-koordináta-rendszerbe viszi át, ahol elvégezzük az árnyalási számításokat. A sugárkövetés a takarási feladatot is itt oldja meg. Az inkrementális képszintézis módszerek azonban a világ-koordináta-rendszerből továbblépnek a képernyő-koordináta-rendszerbe a vetítés és a takarási probléma egyszerűsítésének érdekében. A világ-koordináta-rendszerből a pixelekhez vezető transzformációt *vetítésnek* nevezzük (13.1. ábra).

A pixelek és a textúratérbeli pontok közötti kapcsolat bejárására két lehetőségünk van:

1. a *textúra alapú leképzés* a textúra térben lévő ponthoz keresi meg a hozzátartozó 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 a textúra térben egyenletesen kijelölt pontok képei a képernyőn is egyenletesen helyezkednek el. Így előfordulhat, hogy nem minden érintett pixelt színezzük ki, vagy éppenséggel egy pixel színét feleslegesen sokszor számoljuk ki. A képtér alapú leképzés jól illeszkedik az inkrementális képtér algoritmusok működéséhez,



13.1. ábra. Textúra leképzés

viszont használatához elő kell állítani a paraméterezési és a vetítési transzformációk inverzét, ami korántsem könnyű feladat.

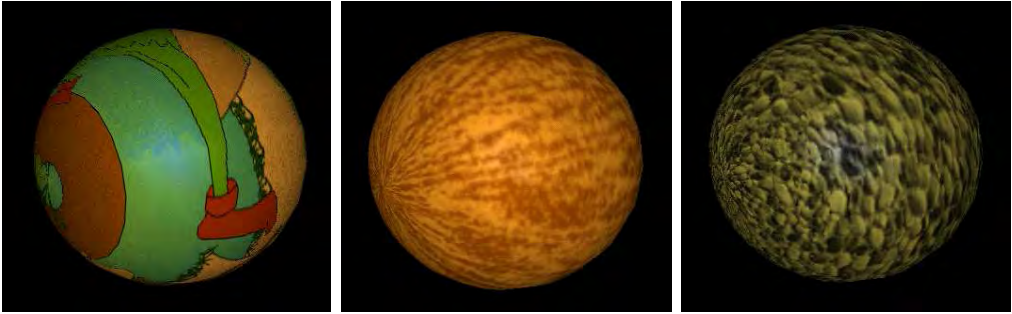
A textúrák lehetnek 1, 2, sőt 3 [Pea85] dimenziósak. Mi csak a 2 dimenziós esettel foglalkozunk, amely úgy is elképzelhető, hogy a textúrát “rátapétázzuk” a felületekre.

13.1. Paraméterezés

A paraméterezés a 2D textúratér egységnyezetét az objektum felületére vetíti. A következőkben a legfontosabb primitívtípusok paraméterezési lehetőségeivel ismerkedünk meg.

13.1.1. Explicit egyenlettel definiált felületek paraméterezése

Mivel az explicit egyenlettel definiált felületeket úgyis két, a $[0, 1]$ tartományba eső paraméter segítségével definiáljuk, ezen paraméterek közvetlenül alkalmazhatók textúra koordinátáknak is. A képtér alapú leképzésnél használt inverz transzformáció, azaz amikor egy $[x, y, z] = r(u, v)$ pontból kell a megfelelő u, v koordinátákat előállítani azonban már nem ilyen egyszerű, mert nemlineáris egyenletek megoldását igényli.



13.2. ábra. Gömb különböző 2D textúrákkal

13.1.2. Háromszögek paraméterezése

A paraméterezés egy textúra térbeli, $v_1(u, v)$, $v_2(u, v)$, $v_3(u, v)$ csúcsú 2D háromszöget képez le egy $\vec{V}_1(x, y, z)$, $\vec{V}_2(x, y, z)$, $\vec{V}_3(x, y, z)$ csúcsú 3D háromszögre. A transzformációtól elvárjuk, hogy a háromszöget valóban háromszögbe vigye át. A legegyszerűbb ilyen transzformáció a lineáris leképzés:

$$[x, y, z] = [u, v, 1] \cdot \begin{bmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{bmatrix} = [u, v, 1] \cdot \mathbf{P}. \quad (13.1)$$

Az ismeretlen mátrixelemeket azokból a feltételekből határozhatjuk meg, hogy a $v_1(u, v)$ pontot a $\vec{V}_1(x, y, z)$ pontra kell transzformálni, a $v_2(u, v)$ pontot a $\vec{V}_2(x, y, z)$ -re, a $v_3(u, v)$ -t pedig a $\vec{V}_3(x, y, z)$ -ra.

Képtér alapú leképzéskor az inverz transzformáció szükséges:

$$[u, v, 1] = [x, y, z] \cdot \mathbf{P}^{-1}. \quad (13.2)$$

13.2. Textúra leképzés a sugárkövetésben

A sugárkövetés a világ-koordinátarendszerben határozza meg a látható pontokat, amelyekre számítani kell a visszavert radianciát. A világ-koordinátarendszerbeli pontból az inverz modellezési transzformációval először a lokális modellezési koordinátarendszerbe megyünk vissza, majd innen az inverz paraméterezéssel a textúra térbe, ahol a BRDF paraméterek megtalálhatóak.

Explicit egyenlettel definiált felületek esetén a sugár-felület metszéspont számítás során, mintegy melléktermékként kiadódik a metszéspont u, v paramétere is, ami automatikus inverz paraméterezést jelent. Egyéb felületekre az inverz paraméterezést külön el kell végezni.

13.3. Textúra leképzés az inkrementális képszintézisben

Az inkrementális képszintézis algoritmusok az egyes pixelekben látható felületi pontokat a képernyő-koordinátarendszerben keresik meg. Ebben a koordinátarendszerben a látható felületi pontot az (X, Y) pixel cím egyértelműen azonosítja, a Z koordináta csak a takarási feladat megoldásához szükséges, a textúra leképzés során nem.

A paraméterezés tárgyalása során egy homogén lineáris transzformációval teremtünk kapcsolatot a textúra tér és a lokális modellezési koordinátarendszer között. A lokális modellezési koordinátarendszert a világ-koordinátarendszerrel majd a képernyő-koordinátarendszerrel ugyancsak homogén lineáris transzformációk kapcsolják össze, amit modellezési illetve nézeti transzformációknak nevezünk. Tehát a textúra teret a képernyő-koordinátarendszerbe átvivő transzformáció szintén homogén lineáris transzformáció, ami a paraméterezés, a modellezési transzformáció és a nézeti transzformáció kompozíciója.

A transzformáció általános alakja:

$$[X \cdot q, Y \cdot q, q] = [u, v, 1] \cdot \mathbf{C}_{3 \times 3}. \quad (13.3)$$

Az egyes koordinátákra (c_{ij} a $\mathbf{C}_{3 \times 3}$ mátrix i, j . eleme):

$$X(u, v) = \frac{c_{11} \cdot u + c_{21} \cdot v + c_{31}}{c_{13} \cdot u + c_{23} \cdot v + c_{33}}, \quad Y(u, v) = \frac{c_{12} \cdot u + c_{22} \cdot v + c_{32}}{c_{13} \cdot u + c_{23} \cdot v + c_{33}}. \quad (13.4)$$

Az inverz transzformáció pedig (C_{ij} a $\mathbf{C}_{3 \times 3}^{-1}$ mátrix i, j . eleme)

$$[u \cdot w, v \cdot w, w] = [X, Y, 1] \cdot \mathbf{C}_{3 \times 3}^{-1}. \quad (13.5)$$

$$u(X, Y) = \frac{C_{11} \cdot X + C_{21} \cdot Y + C_{31}}{C_{13} \cdot X + C_{23} \cdot Y + C_{33}}, \quad v(X, Y) = \frac{C_{12} \cdot X + C_{22} \cdot Y + C_{32}}{C_{13} \cdot X + C_{23} \cdot Y + C_{33}}. \quad (13.6)$$

A képtér algoritmusok során alkalmazott inverz textúra leképzést az inkrementális koncepció alkalmazásával tehetjük még hatékonyabbá. Legyen az $u(X)$ tényezőzt meghatározó hányados számlálójá $uw(X)$, a nevezője pedig $w(X)$. Az $u(X + 1)$ -t az

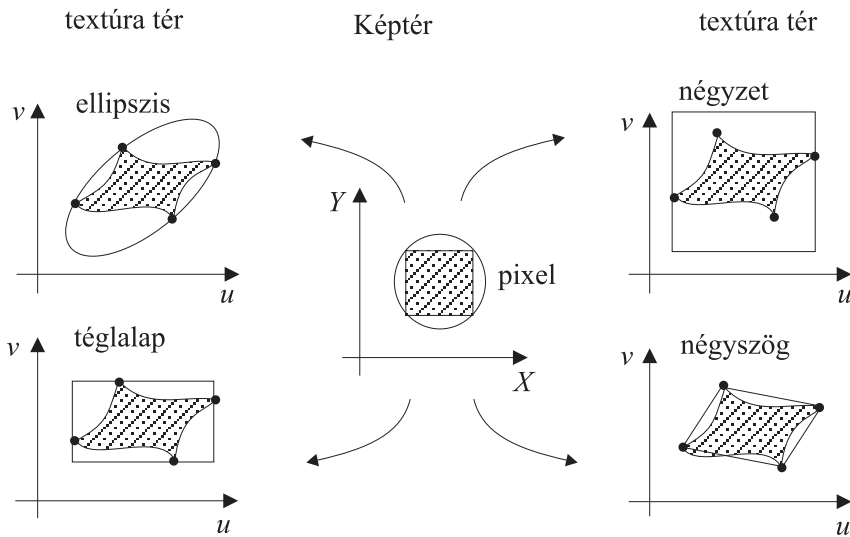
$u(X)$ -ből két összeadással és egyetlen osztással számíthatjuk a következő képlet alkalmazásával:

$$uw(X+1) = uw(X) + C_{11}, \quad w(X+1) = w(X) + C_{13}, \quad u(X+1) = \frac{uw(X+1)}{w(X+1)}. \quad (13.7)$$

Hasonló összefüggések érvényesek a v koordinátára is.

13.4. A textúrák szűrése

A textúra tér és a képernyő-koordináta-rendszer közötti leképezés a textúra tér egyes részeit nagyíthatja, más részeit pedig összenyomhatja. Ez azt jelenti, hogy a képernyőtérben egyenletes sűrűséggel kiválasztott pixel középpontok igen egyenlőtlenül mintavételezhetik a textúrát, ami végső soron mintavételezési problémákat okozhat. Ezért a textúra leképezésnél a mintavételezési problémák elkerülését célzó szűrésnek különleges jelentősége van.



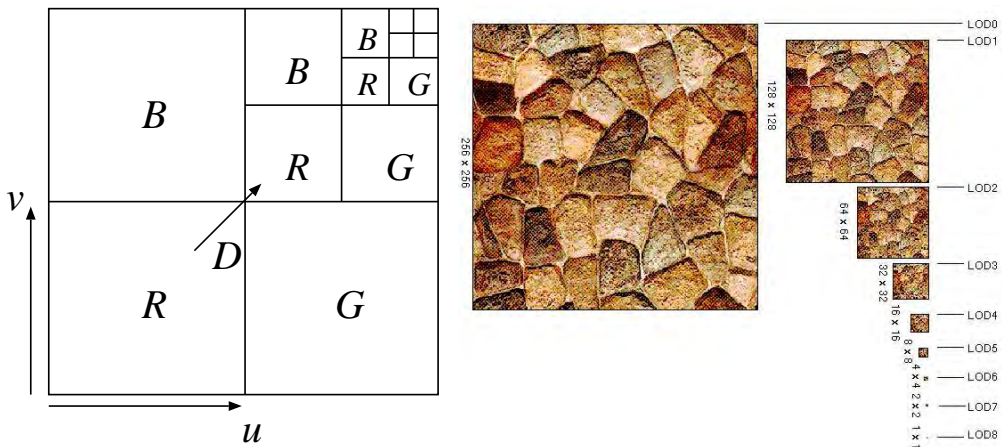
13.3. ábra. A pixel képének approximációja

A textúra szűrés nehézsége abból fakad, hogy a textúra tér és a képtér közötti leképezés nem lineáris. Például, ha doboz szűrést szeretnénk alkalmazni, a pixel textúra térbeli képében kell a texeleket átlagolni, ami szabálytalan, általános görbék által határolt terület. A szokásos eljárások ezt az általános területet egyszerű területekkel, például ellipszissel, négyszöggel, téglalappal vagy négyzettel közelítik (13.3. ábra).

Négyzettel történő közelítés esetén egyetlen pixel színét ezek után 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. A számítási időt integráló táblázatok, ún. *piramisok* használatával csökkenthetjük.

A képpiramis

A piramisok a textúrát (általános esetben egy képet) több felbontáson tárolják. Két egymást követő tábla felbontásának aránya általában 2. Az egymást követő képeket egy kép piramisként is elképzelhetjük, amelyben a legnagyobb felbontású kép a piramis alján, a legdurvább felbontású pedig a piramis tetején foglal helyet.



13.4. ábra. A textúratár mip-map szervezése

A textúráképeket általában *mip-map adatstruktúrába* szervezik [Wil83] (13.4. ábra), amelyben a $M \times M$ eredeti felbontású MM mip-map tömbben a D szintű textúra u, v koordinátájú pontját a következőképpen kereshetjük meg:

$$\begin{aligned}
 R(u, v, D) &= MM[(1 - 2^{-D}) \cdot M + u \cdot 2^{-D}, (1 - 2^{-D}) \cdot M + v \cdot 2^{-D}], \\
 G(u, v, D) &= MM[(1 - 2^{-(D+1)}) \cdot M + u \cdot 2^{-D}, (1 - 2^{-D}) \cdot M + v \cdot 2^{-D}], \\
 B(u, v, D) &= MM[(1 - 2^{-D}) \cdot M + u \cdot 2^{-D}, (1 - 2^{-(D+1)}) \cdot M + v \cdot 2^{-D}].
 \end{aligned}
 \tag{13.8}$$

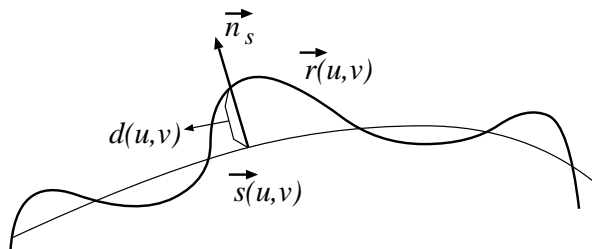
A képpiramis egy speciális képviselője egy rendkívül hasznos elvnek, amit *részletezettségi szintek* elvének (*level of detail (LOD)*) nevezünk. Ez az elv azt mondja,

hogy a különböző modelleket érdemes több különböző pontosságon is nyilvántartani, és valamilyen automatikus mechanizmussal mindig azt a minimális bonyolultságú változatot kiválasztani, ami az adott kameraállásnak még éppen megfelel. A képpiramis a textúrákra és a képekre alkalmazza ezt az elvet. Lényegében hasonló megközelítést alkalmaznak a szimulációs és animációs programok is, amelyek a tárgyakat különböző geometriai pontosságú modellekkel írják le, és a kamera távolsága alapján automatikusan választják az éppen megfelelőt.

13.5. Bucka leképzés

A felületi normálvektor alapvető szerepet játszik 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, de a hatás tekintetében a geometriai modellektől nem marad el lényegesen. A módszer, amit *bucka leképzésnek* (*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 [Bli78]. 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 normálvektorjának irányába egy kis $d(u, v)$ eltolással módosítjuk (13.5. ábra).



13.5. ábra. A buckák leírása

Mivel az $\vec{s}(u, v)$ felület \vec{n}_s normálvektorát a felület (\vec{s}_u, \vec{s}_v) parciális deriváltjainak

vektoriális szorzataiként is kifejezhetjük, a következő kifejezéshez jutunk:

$$\vec{r}(u, v) = \vec{s}(u, v) + d(u, v) \cdot [\vec{s}_u(u, v) \times \vec{s}_v(u, v)]^0 = \vec{s}(u, v) + d(u, v) \cdot \vec{n}_s^0 \quad (13.9)$$

(0 hatvány az egységvektort jelöli). Az $\vec{r}(u, v)$ felület parciális deriváltjai:

$$\vec{r}_u = \vec{s}_u + d_u \cdot \vec{n}_s^0 + d \cdot \frac{\partial \vec{n}_s^0}{\partial u}, \quad \vec{r}_v = \vec{s}_v + d_v \cdot \vec{n}_s^0 + d \cdot \frac{\partial \vec{n}_s^0}{\partial v}. \quad (13.10)$$

Az utolsó tagok elhanyagolhatók, hiszen mind a $d(u, v)$ eltolás, mind pedig a sima felület normálvektorának változása kicsiny:

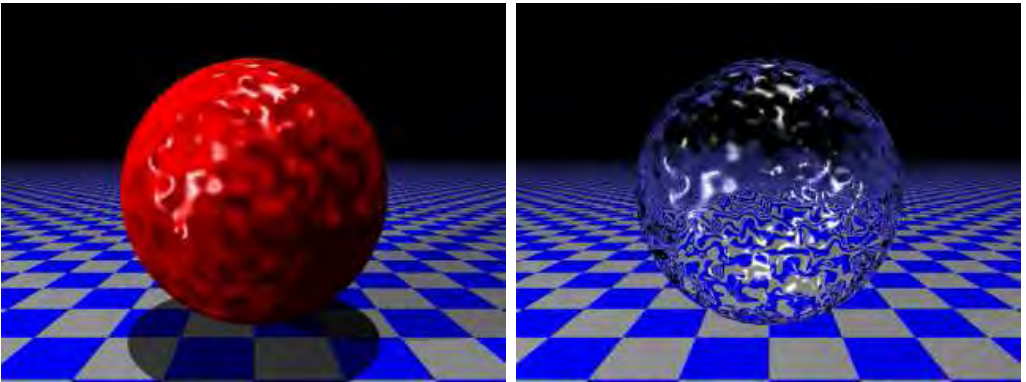
$$\vec{r}_u \approx \vec{s}_u + d_u \cdot \vec{n}_s^0, \quad \vec{r}_v \approx \vec{s}_v + d_v \cdot \vec{n}_s^0. \quad (13.11)$$

A buckás felület normálvektora ezek után:

$$\vec{n}_r = \vec{r}_u \times \vec{r}_v = \vec{s}_u \times \vec{s}_v + d_u \cdot \vec{n}_s^0 \times \vec{s}_v + d_v \cdot \vec{s}_u \times \vec{n}_s^0 + d_u d_v \cdot \vec{n}_s^0 \times \vec{n}_s^0. \quad (13.12)$$

Az utolsó tag nyilván zérus a vektoriális szorzat tulajdonságai miatt. Ezen kívül használhatjuk a következő helyettesítéseket:

$$\vec{n}_s = \vec{s}_u \times \vec{s}_v, \quad \vec{s}_u \times \vec{n}_s^0 = -\vec{n}_s^0 \times \vec{s}_u,$$



13.6. ábra. Buckás gömb textúrázott alaplapon

Végül a buckás felület normálvektora a következőképpen közelíthető:

$$\vec{n}_r = \vec{n}_s + d_u \cdot \vec{n}_s^0 \times \vec{s}_v - d_v \cdot \vec{n}_s^0 \times \vec{s}_u. \quad (13.13)$$

A $d(u, v)$ eltolásfüggényt a textúrákhoz hasonló táblázatban tároljuk, amely neve *bucka tábla*. A buckás felület normálvektora az eltolásfüggvény deriváltjait tartalmazza, amit véges differenciákkal közelíthetünk. Legyen a B bucka tábla egy $N \times N$ méretű tömb. A közelítő deriváltak:

$$\begin{aligned}
 U &= \text{Trunc}(u * N), \quad V = \text{Trunc}(v * N) \\
 \text{if } U < 1 &\text{ then } U = 1 \\
 \text{if } U > N - 2 &\text{ then } U = N - 2 \\
 \text{if } V < 1 &\text{ then } V = 1 \\
 \text{if } V > N - 2 &\text{ then } V = N - 2 \\
 d_u(u, v) &= (B[U + 1, V] - B[U - 1, V]) \cdot N/2 \\
 d_v(u, v) &= (B[U, V + 1] - B[U, V - 1]) \cdot N/2
 \end{aligned}$$

13.6. Visszaverődés leképzés

A textúraleképzés egy szellemes alkalmazása az ideális tükrök szimulációja az inkrementális képszintézis keretein belül, amelyet *visszaverődés leképzésnek* (*reflection-mapping*) nevezünk [MH84]. Ennek lényege az, hogy külön képszintézis lépéssel meghatározzuk, hogy mi látszik a tükörirányban, majd a képet textúraként rátapétázzuk a tükröző objektumra (17.17. ábra).

13.7. Program: sugárkövetés kiegészítése textúra leképzéssel

Egy textúra (`Texture`) texelek tömbjeként adható meg, amely az egységnégyzetbeli ponthoz előkeresi a megfelelő texelt.

```

//=====
class Texture {
//=====
    int UMAX, VMAX;
    Array< SColor > texture;
public:
    Texture( int UMAX0, int VMAX0 )
        : texture( UMAX0 * VMAX0 ) { UMAX = UMAX0; VMAX = VMAX0; }
    void SetTexel(int U, int V, SColor& c) { texture[V*UMAX + U] = c; }
    SColor Texel( double u, double v ) {
        int U = u*UMAX; if (U >= UMAX) U = UMAX - 1; if (U < 0) U = 0;
        int V = v*VMAX; if (V >= VMAX) V = VMAX - 1; if (V < 0) V = 0;
        return texture[ V * UMAX + U ];
    }
};

```

Egy textúrázott primitív (`TexturedPrimitive3D`) a normál primitív és a textúra képességeit hordozza magában. A textúrázott primitívben a normál primitív BRDF-jét átdefiniáljuk, hiszen most az a felület mentén változhat. Jelen esetben a diffúz visszaverődési tényezőt vesszük a textúra tömbből. A felületi pontok és a textúráter koordináták összerendelését a `Parameterize` tagfüggvény végzi el.

```
//=====
class TexturedPrimitive3D : virtual public Primitive3D,
                           public Texture {
//=====
public:
    TexturedPrimitive3D( int UMAX, int VMAX )
        : Primitive3D(), Texture(UMAX, VMAX) { }

    SColor BRDF(Vector3D& L, Vector3D& N, Vector3D& V, Point3D& x);
    virtual Point2D Parameterize(Point3D& x) = 0;
};

//-----
SColor TexturedPrimitive3D :: BRDF(Vector3D& L, Vector3D& N,
                                   Vector3D& V, Point3D& x) {
//-----
    Point2D tc = Parameterize( x );
    mat.kd() = Texel( tc.X(), tc.Y() );
    return mat.BRDF(L, N, V);
}
```

A tényleges textúrázott primitívtípusokat az általános típus specializálásával készíthetjük el, amelyben a (`Parameterize`) tényleges értelmet nyer. Például egy textúrázott gömb (`TexturedSphere`) definíciója:

```
//=====
class TexturedSphere : public Sphere, public TexturedPrimitive3D {
//=====
public:
    TexturedSphere(Point3D& cent, double rad, int UMAX, int VMAX)
        : Sphere(cent, rad), TexturedPrimitive3D( UMAX, VMAX ) { }
    double Intersect( Ray& r ) { return Sphere :: Intersect( r ); }
    Vector3D Normal(Point3D& x) { return Sphere :: Normal( x ); }
    Point2D Parameterize(Point3D& x) {
        Vector3D dir = x - Center();
        double u = (Point2D(dir.X(), dir.Y()).Length() > EPSILON ?
                    ((atan2( dir.Y(), dir.X() ) + M_PI) / 2 / M_PI) : 0;
        double v = acos( dir.Z() / Radius() ) / M_PI;
        return Point2D( u, v );
    }
};
```

14. fejezet

Térfogat modellek és térfogatvizualizáció

Egy *térfogat modell* (*volumetric model*) úgy képzelhető el, hogy a 3D tér egyes pontjaiban sűrűségértékeket adunk meg. A feladat tehát egy $v(x, y, z)$ függvény reprezentálása. A gyakorlatban általában térfogatmodellekre vezetnek a mérnöki számítások (pl. egy elektromágneses térben a potenciáeloszlás). 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.

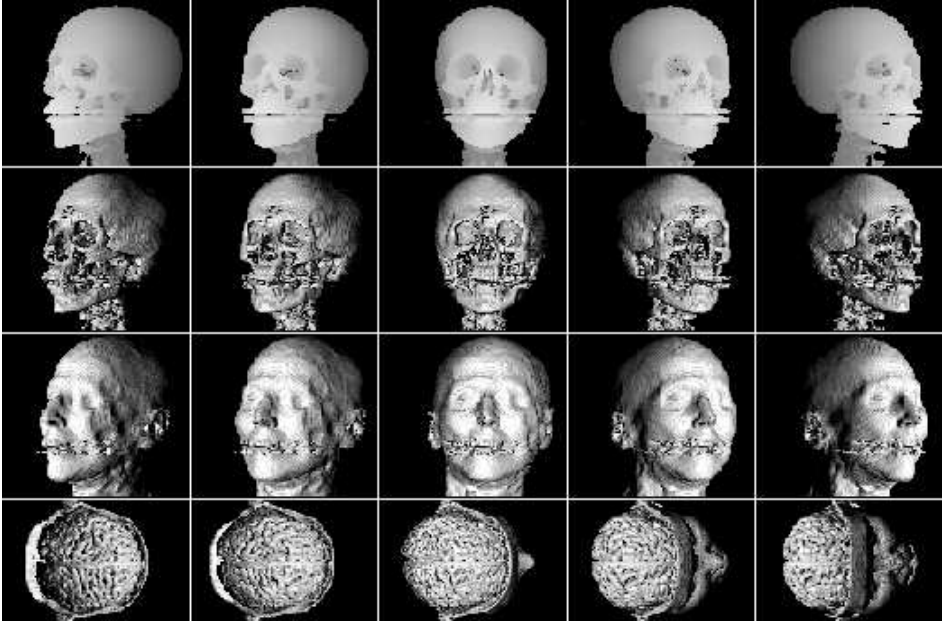
A térfogati modellt általában szabályos ráccsal mintavételezzük, és az értékeket egy 3D mátrixban tároljuk. Úgy is tekinthetjük, hogy egy mintavételi érték a térfogat egy kicsiny kockájában érvényes függvényértéket képviseli. Ezen elemi kockákat térfogatelemnek, vagy *voxel*nek nevezzük.

Nyilván az elemi kockák direkt felrajzolásának nem sok értelme lenne, hiszen ekkor csak a térfogat külső oldalain lévő értékeket láthatjuk (egy páciens fejéről azért csinálunk CT-t, hogy belsejét vizsgálhassuk, nem pedig azért, hogy az arcában gyönyörködjünk). A teljes térfogat áttekintéséhez bele kell látnunk a térfogatba, tehát a térfogati modellt célszerű úgy kezelni, mintha az részben átlátszó anyagból állna. A térfogatot alkotó “ködöt” két alapvetően különböző módon jeleníthetjük meg. A *direkt módszerek* közvetlenül a térfogati modellt fényképezik le, az *indirekt módszerek* pedig először átalakítják egy másik modellre, amelyet azután a szokásos módszerekkel jeleníthetünk meg.

14.1. Direkt térfogatvizualizációs módszerek

A direkt módszerek a ködöt a képsíkra vetítik és számba veszik az egyes pixeleknek megfelelő vetítősugarak által metszett voxeleket. A pixelek színét a megfelelő voxelek színéből és átlátszóságából határozzák meg.

Elevenítsük fel a radianciát a fényelnyelő anyagokban leíró 8.34. egyenletet! Feltehetjük, hogy az anyag nem bocsát ki fényt magából. Ekkor egy sugár mentén a radiancia



14.1. ábra. CT és MRI mérések vizualizációja

a fényelnyelés miatt csökken, a sugár irányába ható visszaverődés miatt viszont nő:

$$\frac{dL(s, \omega)}{ds} = -\kappa_t(s) \cdot L(s, \omega) + L_{\text{is}}(s). \quad (14.1)$$

Amennyiben $L_{\text{is}}(s)$ ismert, az egyenlet megoldása:

$$L(s, \omega) = \int_s^T e^{-\int_s^\tau \kappa_t(p) dp} \cdot L_{\text{is}}(\tau, \omega) d\tau, \quad (14.2)$$

ahol T a maximális sugárparaméter.

Az integrálok kiértékeléséhez a $[0, T]$ sugárparaméter tartományt kis intervallumokra osztjuk és az integrált téglányszabállyal becsüljük. Ez megfelel a folytonos differenciálegyenletet véges differenciaegyenlettel történő közelítésének:

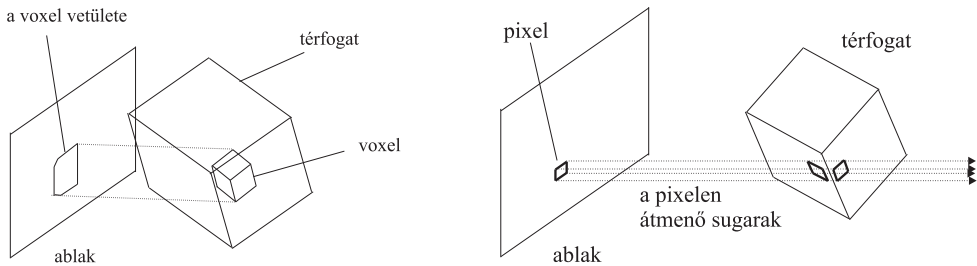
$$\frac{\Delta L(s, \omega)}{\Delta s} = -\kappa_t(s) \cdot L(s, \omega) + L_{\text{is}}(s) \quad \implies$$

$$L(s + \Delta s, \omega) = L(s, \omega) - \kappa_t(s) \cdot \Delta s \cdot L(s, \omega) + L_{\text{is}}(s) \cdot \Delta s. \quad (14.3)$$

Jelöljük a $L_{\text{is}}(s) \cdot \Delta s$ szórt radianciát $C(s)$ -sel, amit ezek után a voxel saját színének tekintünk. Az $\alpha(s) = \kappa_t \cdot \Delta s$ érték — amelyet *opacitásnak* nevezünk — a két minta közötti fényelnyelést jellemzi. Ezekkel az új jelölésekkel:

$$L(s + \Delta s, \omega) = (1 - \alpha(s)) \cdot L(s, \omega) + C(s). \quad (14.4)$$

Ezt az egyenletet a térfogat vizualizálásakor a pixelekhez tartozó sugarakra kell megoldani. A sugarak definiálása során kiindulhatunk a térfogattól, vagy a képernyő pixeleiből egyaránt. Az első módszer neve *térfogat vetítés* [DCH88] a másodiké pedig *térfogati sugárkövetés* [Lev88, Lev90]. A térfogat vetítés az inkrementális képszintézishez, azon belül pedig a festő algoritmushoz hasonlít, a térfogati sugárkövetés pedig a normál sugárkövetés adaptációja.



14.2. ábra. *Térfogat vetítés (bal) és térfogati sugárkövetés (jobb)*

14.1.1. Térfogat vetítés

A térfogat vetítés a térfogattól indul, és a térfogati adathalmazt az ablak síkjával párhuzamos, Δs távolságra lévő síkok mentén mintavételezi, majd az egyes síkokat az ablakra vetíti. A feldolgozást a szemtől távolabbi síkokkal kezdjük és onnan közeledünk a szempozíció felé. A feldolgozás adott pillanatában érvényes $L(s, \omega)$ akkumulálódó radianciát a pixeleken tároljuk. Így egy sík vetítése során a 17.4. egyenletet az egyes pixeleken tárolt $L(s)$ érték és a $C(s)$ vetített érték felhasználásával számítjuk ki. Ha az összes síkon végigmentünk, a megjelenítendő színeket a pixeleken akkumulálódott radiancia határozza meg.

14.1.2. Térfogati sugárkövetés

A térfogati sugárkövetés a pixelektől indul. A pixel középpontokon keresztül egy-egy sugarat indítunk a térfogatba, és a sugár mentén haladva oldjuk meg a 17.4 egyenletet.

Amennyiben a fényvel megegyező irányban haladunk, a 17.4 egyenletet az eredeti formájában értékeljük ki. Sajnos ez a módszer gyakran felesleges számításokat igényel, hiszen a számításokat a legtávolabbi voxeleknél kezdjük, amelyek az esetek döntő részében úgyszem látszanak a képen. Ezért célszerűbb, ha a fényvel szemben haladunk a sugárintegrál kiértékelésekor. Természetesen ekkor a 17.4 egyenletet ki kell facsarni.

Tegyük fel, hogy a szemből már az s paraméterig jutottunk, és idáig úgy találtuk, hogy a sugár mentén az s paraméter és a szem között $L^*(s, \omega)$ radiancia akkumulálódott, és az integrált opacitás, amivel a s utánról érkező radianciát kell szorozni pedig $\alpha^*(s)$. Ha a sugárparamétert Δs -sel léptetjük, akkor a következő inkrementális összefüggések érvényesek:

$$L^*(s - \Delta s, \omega) = L^*(s, \omega) + (1 - \alpha^*(s)) \cdot C(s), \quad (14.5)$$

$$1 - \alpha^*(s - \Delta s) = (1 - \alpha(s)) \cdot (1 - \alpha^*(s)). \quad (14.6)$$

Most a pixelek színét az $L^*(0)$ érték határozza meg. Ha az összefüggések inkrementális kiértékelése során úgy találjuk, hogy az $1 - \alpha^*(s - \Delta s)$ mennyiség egy küszöb érték alá került, befejezhetjük a sugár követését, mert a hátrébb levő voxelek hatása elhanyagolható.

14.2. A voxel szín és az opacitás származtatása

A térfigatvizualizációs algoritmusok a voxelek saját színével és opacitásértékeivel dolgoznak, amelyeket a $v(x, y, z)$ mért voxelértékekből származtathatunk. A gyakorlatban többféle módszer terjedt el, amelyek különböző megjelenítésekhez vezetnek.

Az egyik leggyakrabban használt eljárás a felületi modellek árnyalását adaptálja a térfigatra (17.8. ábra). Az árnyalási paraméterek származtatáshoz osszuk fel a mért $v(x, y, z)$ sűrűségfüggvény értékészletét adott számú intervallumra (például, a CT és az MRI képeknél a levegő, lágyszövet és a csont sűrűségértékeit érdemes elkülöníteni). Az egyes intervallumokhoz diffúz és spekuláris visszaverődési tényezőt, valamint átátlátszóságot adunk meg. Absztrakt fényforrások jelenlétét feltételezve, például a Phong illuminációs képlet segítségével meghatározzuk a voxelhez rendelhető színt. Az illuminációs képletek által igényelt normálvektort a $v(x, y, z)$ gradienséből kaphatjuk meg.

Ha a 3D voxelek mérete a, b és c , akkor a gradiens vektor egy x, y, z pontban:

$$\text{grad } v = \begin{pmatrix} \frac{v(x + a, y, z) - v(x - a, y, z)}{2a} \\ \frac{v(x, y + b, z) - v(x, y - b, z)}{2b} \\ \frac{v(x, y, z + c) - v(x, y, z - c)}{2c} \end{pmatrix}. \quad (14.7)$$

Teljesen homogén tartományokban a gradiens zérus, tehát a normálvektor definiálatlan, ami a képen zavarokat okozhat. Ezeket eltüntethetjük, ha a voxelek opacitását a megadott átlátszóság és a gradiens abszolút értékének a szorzataként számítjuk, hiszen ekkor a homogén tartományok teljesen átlátszóak lesznek.

Egy másik módszer azt feltételezi, hogy a megvilágítás a térfogat túlsó oldaláról jön. Ha az opacitást a $v(x, y, z)$ -vel arányosan választjuk, a pixelek színe arányos lesz az integrált opacitással, azaz a $v(x, y, z)$ sugármenti integráljával. Mivel a röntgensugarak is hasonlóan nyelődnek el, a kép hatásában a *röntgen képek*re emlékeztet.

Végül egy gyorsan kiértékelhető, és ugyanakkor az orvosi diagnosztikában rendkívül hasznos megjelenítési eljáráshoz jutunk, ha minden sugár mentén az $v(x, y, z)$ maximumával arányosan színezzük ki a pixeleket (1.6. ábra jobb oldala).

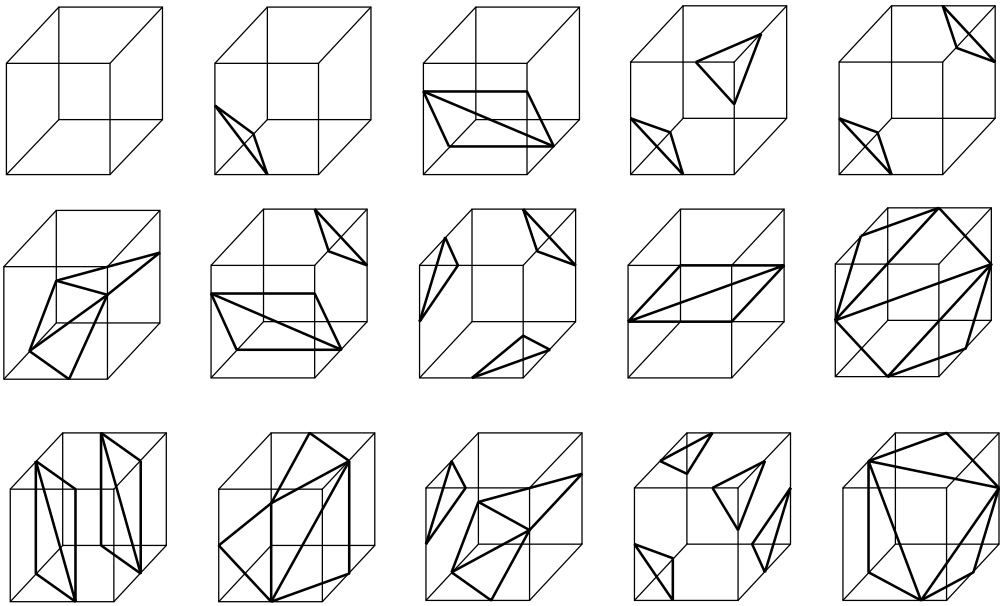
14.3. Indirekt térfogatvizualizációs módszerek

Az indirekt módszerek a térfogati modellt egy másfajta modellre alakítják át, majd azt fényképezik le.

14.3.1. Masírozó kockák algoritmus

A legkézenfekvőbb közbenső reprezentáció a felületi modell, hiszen a felületi modellek megjelenítése a számítógépes grafika leginkább kimunkált területe. Egy térfogati modellből elvileg úgy nyerhetünk felületeket, hogy azonosítjuk a 3D térfogat *szintfelületeit*, azaz azon 2D ponthalmazokat, ahol a $v(x, y, z)$ megegyezik a megadott szintértékkel. Ez korántsem olyan könnyű, mint ahogyan első pillanatban látszik, hiszen mi a $v(x, y, z)$ függvényt csak diszkrét értékekben ismerjük, a közbenső pontokat a tárolt adatok interpolációjával kell előállítani.

Egy ilyen, az interpolációt és a szintfelületet megkeresését párhuzamosan elvégző módszer a *masírozó kockák algoritmus* (*marching cubes algorithm*). Az algoritmus első lépésben a szintfelület értékének és a voxelek értékének összehasonlításával minden voxelre eldönti, hogy az belső voxel-e avagy külső voxel. Ha két szomszédos voxel eltérő típusú, akkor közöttük határnak kell lennie. A határ pontos helye a voxelek élein az érték alapján végzett lineáris interpolációval határozható meg. Végül az éleken kijelölt pontokra háromszögeket illesztünk, amelyekből összeáll a szintfelület. A háromszög illesztéshez figyelembe kell venni, hogy egy zárt alakzat az egy pontra illeszkedő 8 voxel összesen 256-féleképpen metszheti, amiből végül 14 ekvivalens eset különíthető el (17.3. ábra). A metszéspontokból a háromszögekhez úgy juthatunk el, hogy először azonosítjuk a megfelelő esetet, majd eszerint definiáljuk a háromszögeket.



14.3. ábra. Egy zárt alakzat az egy pontra illeszkedő 8 voxel összesen 14 féleképpen metszheti

14.3.2. Fourier-tér módszerek

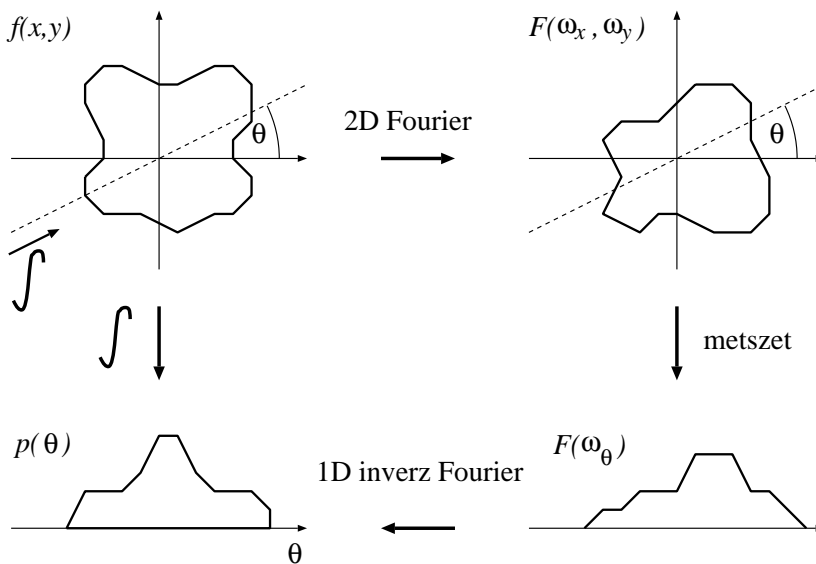
A Fourier-transzformációnak több hasznos tulajdonsága van: egyrészt a gyors Fourier-transzformációs módszerrel hatékonyan elvégezhető akár magasabb dimenziókban is, másrészt a transzformált értéke a 0 frekvencián megegyezik a függvény integráljával. Mivel a röntgenszerű megjelenítéshez a $v(x, y, z)$ függvényt a különböző sugarak mentén kell integrálni, a megjelenítéshez az adathalmaz Fourier-transzformáltja vonzóbbnak tűnik mint maga a mért adat.

A $V(\omega_x, \omega_y, \omega_z)$ Fourier-transzformált és a $v(x, y, z)$ eredeti adat között a következő összefüggés áll fenn:

$$V(\omega_x, \omega_y, \omega_z) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} v(x, y, z) \cdot e^{-2\pi j(x\omega_x + y\omega_y + z\omega_z)} dx dy dz, \quad (14.8)$$

ahol $j = \sqrt{-1}$.

Vegyük észre, hogy a $V(\omega_x, \omega_y, 0)$ *metszet (slice)* éppen a z irányú sugarakkal számított kép 2D Fourier-transzformáltja. Hasonlóan a $V(\omega_x, 0, \omega_z)$ az y irányú sugarakkal, a $V(0, \omega_y, \omega_z)$ pedig az x irányú sugarakkal vett integrálok Fourier-transzformáltja. Ráadásul — a Fourier-transzformáció sajátosságai miatt — általános orientáció-



14.4. ábra. Fourier-tér módszer

jú metszetek képzésével tetszőleges irányból látott kép Fourier-transzformáltja számítható. Ha az ablak orientációját az oldalakkal párhuzamos $W_u = (\omega_{ux}, \omega_{uy}, \omega_{uz})$ és $W_v = (\omega_{vx}, \omega_{vy}, \omega_{vz})$ vektorokkal definiáljuk, akkor a kép Fourier-transzformáltja az

$$P(\omega_u, \omega_v) = V(\omega_{ux}\omega_u + \omega_{vx}\omega_v, \omega_{uy}\omega_u + \omega_{vy}\omega_v, \omega_{uz}\omega_u + \omega_{vz}\omega_v) \quad (14.9)$$

összefüggéssel határozható meg. Ebből pedig egy u, v koordinátájú pixelnek megfelelő integrál értéke inverz Fourier-transzformációval számítható:

$$p(u, v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} P(\omega_u, \omega_v) \cdot e^{2\pi j(u\omega_u + v\omega_v)} d\omega_u d\omega_v. \quad (14.10)$$

Az eljárás a viszonylag számításigényes 3D Fourier-transzformáció egyszeri végrehajtása után, tetszőleges irányú megjelenítéshez már csak 2D inverz Fourier-transzformációkat alkalmaz, ezért interaktív körbejáráshoz vagy animációhoz javasolható.

14.4. Program: masírozó kockák algoritmus

A masírozó kockák algoritmus a voxel tömb által reprezentált függvény szintfelületét azonosítja, és azokra egy háromszögsorozatot illeszt. A szintfelület sokféleképpen metszhet egy voxel, amit aszerint osztályozhatunk, hogy melyik csúcspont van a keresett felületi érték felett illetve alatt. A voxel 8 csúcának osztályozása során kapott érték egy konfigurációt jelent. Ezek a konfigurációk egy 256 soros táblázatba gyűjthetők (`polytab`), amelynek soraiban a -1 -gyel lezárt sorozat számhármasai azonosítják az adott eset poligonizációjához szükséges háromszögeket [Pap98]. A táblázatból kiolvashatjuk, hogy melyik éleket kell metszenünk a poligonok csúcsainak meghatározásához. A csúcsokat lineáris interpolációval számoljuk ki. Az alábbiakban ezen táblázatból csupán egy részletet mutatunk be, a teljes tábla a CD-n megtalálható.

```
int polytab[256][32]=
{
{-1, }, // 0:00000000, f:0
{1, 0, 4, 0, 3, 0, -1, }, // 1:00000001, f:1
{1, 0, 1, 2, 1, 5, -1, }, // 2:00000010, f:1
{1, 5, 4, 0, 1, 2, 1, 2, 4, 0, 3, 0, -1, }, // 3:00000011, f:2
...
{1, 0, 1, 5, 1, 2, -1, }, // 253:11111101, f:1
{1, 0, 3, 0, 4, 0, -1, }, // 254:11111110, f:1
{-1, }, // 255:11111111, f:0
};
```

A következő tömbökben azt tartjuk nyilván, hogy a kocka egyes csúcsai a bal-alsó-hátsó sarokhoz képest milyen relatív x , y és z koordinátákkal rendelkeznek.

```
int x_relpos_tab[8]={0, 1, 1, 0, 0, 1, 1, 0}; // x-ben
int y_relpos_tab[8]={0, 0, 0, 0, 1, 1, 1, 1}; // y-ben
int z_relpos_tab[8]={0, 0, -1, -1, 0, 0, -1, -1}; // z-ben
```

A `Volume` osztály egy `size` felbontású térfogatmodellt testesít meg. Az osztály konstruktora fájlból beolvassa voxelértékeket, az `Interpolate` tagfüggvénye a voxel-kockák élein interpolálja a hely- és irányvektorokat. A `MarchingCube` pedig az ismert algoritmussal egyetlen voxelre megvizsgálja, hogy a szintfelület metszi-e azt, és előállítja a metsző szintfelület háromszöghálós közelítését. A teljes térfogat modell megjelenítéséhez a `MarchingCube` tagfüggvényt minden egyes voxelre meg kell hívni. A kapott háromszögeket a szokásos 3D csővezetéken végigvezetve jeleníthetjük meg.

```

//=====
class Volume {
//=====
    int size;
    BYTE *** grid;
public:
    Volume( char * filename );
    int GridSize( ) { return size; }
    BYTE V(int x, int y, int z) {
        if (x < 0 || y < 0 || z < 0 ||
            x >= size || y >= size || z >= size) return 0;
        return grid[x][y][z];
    }

    void Interpolate( Point3D& point1, Point3D& point2,
                    Point3D& norm1, Point3D& norm2,
                    double value1, double value2, double isolevel,
                    Point3D& point, Point3D& norm ) {
        double m = (isolevel - value1) / (value2 - value1);
        point = point1 + (point2 - point1) * m;
        norm = norm1 + (norm2 - norm1) * m;
    }
    TriangleList3D * MarchingCube(int x, int y, int z, double isolevel);
};

//-----
Volume :: Volume( char * filename ) {
//-----
    size = 0;
    FILE * file = fopen(filename, "rb");
    if (fscanf(file,"%d", &size) != 1) return;

    grid = new BYTE**[size];
    for(int x = 0; x < size; x++) {
        grid[x] = new BYTE*[size];
        for(int y = 0; y < size; y++) {
            grid[x][y] = new BYTE[size];
            for(int z = 0; z < size; z++)
                grid[x][y][z] = fgetc(file);
        }
    }
}

```

```

//-----
TriangleList3D * Volume :: MarchingCube( int x, int y, int z,
                                         double isolevel ) {
//-----
    BYTE cubeindex = 0;
    if (V(x,y,z) < isolevel) cubeindex|=1;
    if (V(x+1,y,z) < isolevel) cubeindex|=2;
    if (V(x+1,y,z-1) < isolevel) cubeindex|=4;
    if (V(x,y,z-1) < isolevel) cubeindex|=8;
    if (V(x,y+1,z) < isolevel) cubeindex|=16;
    if (V(x+1,y+1,z) < isolevel) cubeindex|=32;
    if (V(x+1,y+1,z-1) < isolevel) cubeindex|=64;
    if (V(x,y+1,z-1) < isolevel) cubeindex|=128;
    if ( cubeindex == 0 || cubeindex == 255 ) return NULL;

    TriangleList3D * tlist =
        new TriangleList3D(0 /* emisszio */, 0.1 /* ka */,
                          0.4 /* kd */, 0.2 /* ks */, 10 /* shine */);

    for(int j = 0, t = 0; polytable[cubeindex][j] != -1; j += 6) {
        Point3D p1, p2, point[3], n1, n2, norm[3];
        for( int j1 = 0; j1 < 6; j1 += 2 ) {
            int x1 = x + x_relpos_tab[ polytable[cubeindex][j+j1] ];
            int y1 = y + y_relpos_tab[ polytable[cubeindex][j+j1] ];
            int z1 = z + z_relpos_tab[ polytable[cubeindex][j+j1] ];
            Point3D point1( x1, y1, z1 );
            Point3D norm1( V(x1-1,y1,z1) - V(x1+1,y1,z1),
                          V(x1,y1-1,z1) - V(x1,y1+1,z1),
                          V(x1,y1,z1-1) - V(x1,y1,z1+1) );
            double value1 = V(x1,y1,z1);

            int x2 = x + x_relpos_tab[ polytable[cubeindex][j+j1+1] ];
            int y2 = y + y_relpos_tab[ polytable[cubeindex][j+j1+1] ];
            int z2 = z + z_relpos_tab[ polytable[cubeindex][j+j1+1] ];
            Point3D point2( x2, y2, z2 );
            Point3D norm2( V(x2-1,y2,z2) - V(x2+1,y2,z2),
                          V(x2,y2-1,z2) - V(x2,y2+1,z2),
                          V(x2,y2,z2-1) - V(x2,y2,z2+1) );
            double value2 = V(x2,y2,z2);

            Interpolate( point1, point2, norm1, norm2, value1, value2,
                          isolevel, point[j1/2], norm[j1/2] );
            norm[j1/2].Normalize( );
        }
        tlist -> AddTriangle( t++, point[0], point[1], point[2],
                              norm[0], norm[1], norm[2] );
    }
    return tlist;
}

```

15. fejezet

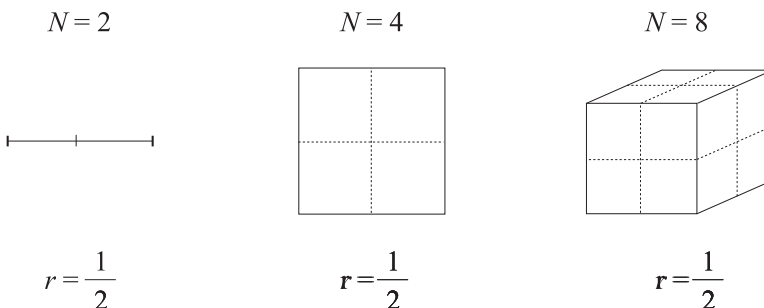
Fraktálok

15.1. A Hausdorff-dimenzió

Fraktálon tört-dimenziós alakzatot értünk. Ez első hallásra meglepő, hiszen a klasszikus definíció alapján a *dimenziószám* csak természetes szám lehet. A klasszikus *topológiai dimenzió* definíciója így hangzik:

1. A pont 0 dimenziós.
2. Egy alakzat n dimenziós, ha két részre lehet vágni egy $n - 1$ dimenziós alakzattal, de kevesebb dimenzióssal nem.

Ahhoz tehát, hogy nem egész dimenziókról beszélhessünk, magának a dimenzióknak kell új értelmezést adni. Nyilván olyat, ami a megszokott objektumainkra visszaadja a klasszikus dimenziószámot, viszont lehetőség van olyan beteg objektumok létrehozására, amelyekre nem egész dimenzió adódik. Ezt az általánosított dimenziót nevezzük *Hausdorff-dimenzió*nak, amely az *önhasonlóság* fogalmán alapul.



15.1. ábra. Klasszikus önhasonló objektumok dimenziója

Vegyünk például egy 1 dimenziós szakaszt és $r = 1/n$ -szeres hasonlósági transzformációval kicsinyítsük le. A kapott kis szakaszból $N = n$ darab összeragasztásával megkaphatjuk az eredeti szakaszunkat. Hasonló kísérletet elvégezhetünk egy 2 dimenziós téglalappal is. Az $r = 1/n$ hasonlósági transzformációval kapott kis téglalapból $N = n^2$ szükséges az eredeti téglalap lefedéséhez. A térben egy 3 dimenziós téglalapról $r = 1/n$ arányú kicsinyített változatából $N = n^3$ darabot kell felhasználnunk az eredeti téglalapról lefedéséhez.

Úgy tűnik, hogy D dimenzióban az r kicsinyítési arány és a kicsinyített változattal az eredeti lefedéséhez szükséges darabszám között az alábbi összefüggés állítható fel:

$$N = \frac{1}{r^D}. \quad (15.1)$$

Ha ezt elfogadjuk, sőt a definíció rangjára emeljük, akkor ezen összefüggés alapján definiálhatjuk egy pontthalmaz Hausdorff-dimenzióját:

$$D = \frac{\log N}{\log 1/r}. \quad (15.2)$$

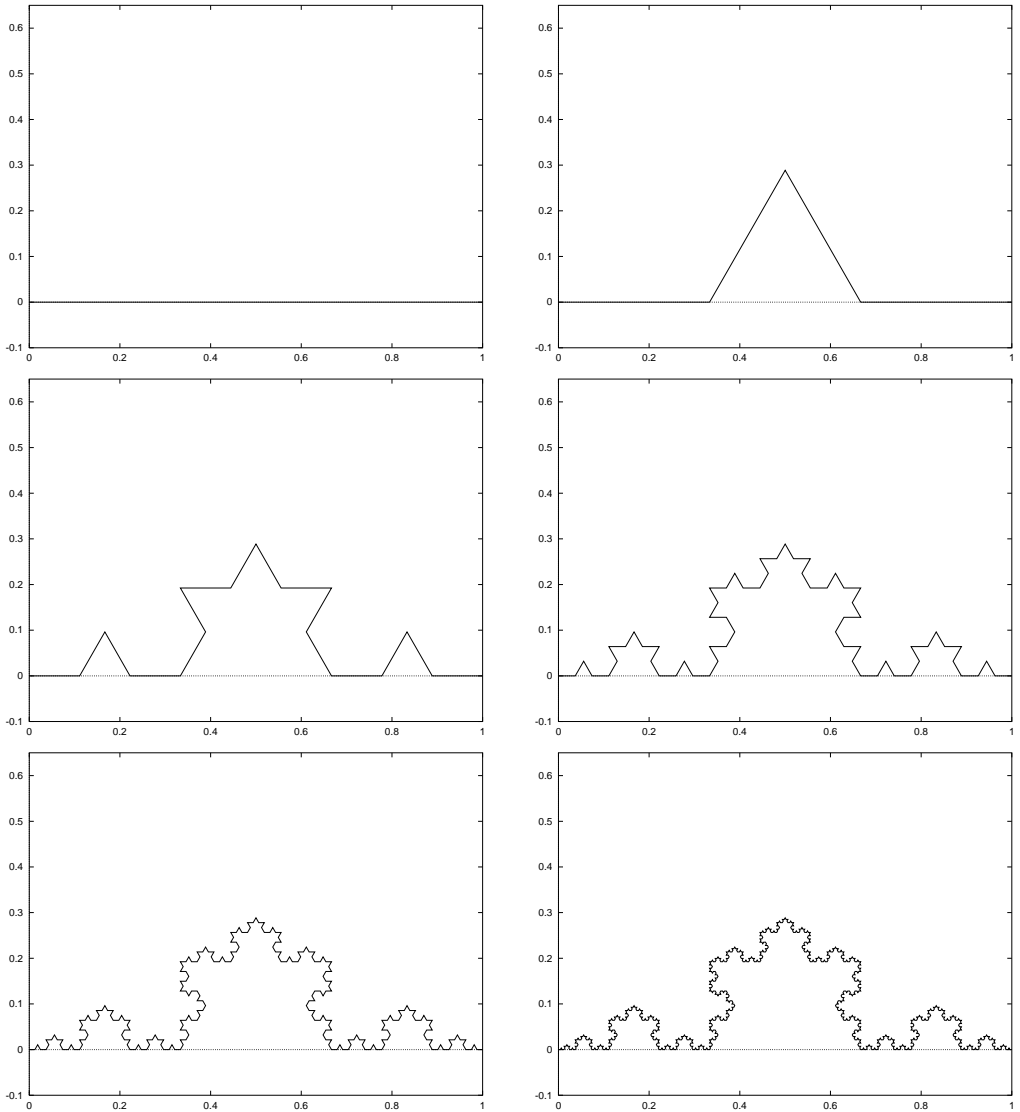
Az első fraktálunk világra hozásához már nincs más feladatunk, mint egy olyan pontthalmazt találni, amelyre a fenti mérőszám nem egész. Egy megfelelő pontthalmaz a Koch-görbe, amely rekurzív definícióval adható meg.

Induljunk ki egy szakaszból, harmadoljuk el, majd a középső harmad felé emeljük egy szabályos háromszöget és a szakaszdarabot cseréljük le a háromszög másik két oldalával (15.2. ábra). Ezzel a művelettel a szakaszt egy négy szakaszból álló törtvonallal váltottuk fel. Most ismételjük meg a műveletet mind a 4 szakaszra, majd rekurzív módon végtelen sokszor minden keletkező szakaszra. A határértékként kapott alakzat neve *Koch-görbe*.

A görbe önhasonló, mert 4 darab, az eredetivel hasonló alakzattal áll, amelyből egy az eredetiből $1/3$ szoros nyújtással állítható elő. Tehát a Koch-görbére $N = 4$ és $r = 1/3$. Így a Koch-görbe dimenziója

$$D_{\text{Koch}} = \frac{\log 4}{\log 3} \approx 1.26$$

ami nem egész, tehát a Koch-görbe fraktál. A fraktális tulajdonság intuitíve is érzékelhető abból, hogy a görbe hossza végtelen (minden iteráció során a hossz $4/3$ -szorosára nő), azaz már kilóg az 1 dimenzióból, de területe — lévén, hogy mégiscsak egy véges tartományban tekergőző görbéről van szó — zérus, azaz még nem kétdimenziós. A dimenzió tehát valahol az 1 és 2 között kell lennie. A fraktálokra általában is jellemző, hogy a szokásos mértékek, mint a hossz, terület, térfogat, rájuk nem értelmezhetők.



15.2. ábra. A Koch-görbe alakulása az iteráció során

15.1.1. Fraktális dimenzió nem önhasonló objektumokra

A Koch-görbe kapcsán megfigyeltük, hogy az iteráció során egyre kisebb kanyarokat véve, a hossza végtelenhez divergál annak ellenére, hogy csak egy véges tartományban tekergőzik. Ez azt is jelenti, hogy a görbére nézve nem látszik rajta, hogy végtelen hosszú, csak akkor tudatosul bennünk, ha egyre inkább kinagyítva a görbét a hullámossága csak nem akar csökkenni. Ezen tulajdonság alapján megpróbálhatjuk véges vonalzóval megmérni a görbénk hosszát.

Pontosabban tegyük fel, hogy egy l hosszú vonalzót akarunk ráilleszteni a görbére, és számláljuk, hogy ez hányszor sikerül. Ha $l = r = 1/3$, akkor $n = N = 4$ -szer tudjuk rátenni a Koch-görbére. Ha viszont $l = r^2 = 1/3^2$, akkor $n = N^2 = 4^2$ -szer sikerül. Általában, ha a vonalzónk hossza $l = r^m$, akkor $n = N^m$ -szer illeszthetjük a görbére, tehát a mért hossz $h(l) = l \cdot n = l \cdot N^m$. Felhasználva a fraktális dimenzió definíciójaként szolgáló 15.2 egyenletet, a mért hossz:

$$h(l) = l \cdot N^m = l \cdot \left(\frac{1}{r^D}\right)^m = l \cdot \left(\frac{1}{r^m}\right)^D = l \cdot \left(\frac{1}{l}\right)^D = \frac{1}{l^{D-1}}. \quad (15.3)$$

A szokásos értelemben vett hosszt az $l \rightarrow 0$ határesetként kapjuk, amikor erre a görbére $h(l) \rightarrow \infty$, így számunkra nincs különösebb információtartalma. Viszont a divergencia sebessége a képlet szerint a fraktális dimenziótól függ, így ez alapján is definiálhatjuk a fraktális dimenziót. Ráadásul a véges vonalzóval történő hosszmérést nem önhasonló objektumokra is elvégezhetjük, így egy általános dimenziófogalomhoz juthatunk.

Legyen tehát a dimenzió értelmezése a következő: végezzünk véges, egyre kisebb vonalzókkal hosszméréseket. A mért hosszt ábrázoljuk a vonalzó hosszának függvényeként logaritmikus skálán. Mivel

$$\log h(l) = -(D - 1) \cdot \log l \quad (15.4)$$

a mérési pontok egy olyan egyenes mentén helyezkednek el, amelynek meredeksége $-(D - 1)$. Az emelkedési szögéből tehát a görbe fraktális dimenziója meghatározható. Ez a dimenziófogalom önhasonló objektumokra visszaadja a 15.2 egyenlet definícióját, de annál általánosabb, hiszen nem önhasonló objektumokra is alkalmazható.

Tudományunkat felhasználhatjuk például Nagy-Britannia partjának vagy az EKG görbék fraktális dimenziójának meghatározására. Az EKG vagy EEG görbék fraktális dimenziójának orvosi diagnosztikai értéke van. Egy másik gyakorlati alkalmazás lehet a katonai légi és űrfelvételek automatikus feldolgozása. Mivel az euklideszi geometrián nevelkedett mérnökeink egész dimenziókban gondolkodnak, az ember alkotta objektumok határvonalai tipikusan egész dimenziósak. Nem így a természet, amely sokkal inkább a fraktális dimenziókat kedveli. Ezért ha egy légi felvételen azonosítjuk a jó közelítéssel egész dimenziós határral rendelkező objektumokat, akkor van némi esélyünk arra, hogy egy erdő mélyére rejtett rakétasilóra bukkanunk.

A véges vonalzóanalógia mintájára bevezethetjük a *dobozdimenzió* fogalmát is. Tegyük a görbére egy rácsot, melyben egy elemi téglalap a görbét befoglaló téglalap λ -szoros kicsinyítése. Ha λ -val 0-hoz tartunk, azaz a rácsot fokozatosan finomítjuk, a vonalzóval történő mérésnél használt gondolatmenettel bizonyíthatjuk, hogy egy fraktálnál a görbe által metszett elemi négyzetek száma arányos $1/\lambda^D$ -vel. Ezt az információt szintén felhasználhatjuk a dimenzió kiszámítására.

15.2. Brown-mozgás alkalmazása

A fraktálok számítógépes grafikai felhasználásához szükségünk van olyan matematikai gépekre, amelyek tömegtermékként ontják az kívánt fraktálokat. Egy ilyen gép a *Brown-mozgás* matematikai modellje, a *Wiener-féle sztochasztikus folyamat*. Ez olyan véletlen valós $X(t)$ függvény, amely 1 valószínűséggel a 0-ból indul, folytonos, és az $X(t+s) - X(t)$ növekmények 0 várható értékű normális eloszlást követnek, és két nem átlapolódó intervallumban egymástól függetlenek.

A független növekményűségből következik, hogy a növekmények szórásnégyzete arányos az intervallum hosszával. A bizonyításhoz írjuk fel a szórás képletét

$$\sigma^2(s) = D^2[X(t+s) - X(t)] = E[(X(t+s) - X(t))^2] - E^2[(X(t+s) - X(t))]. \quad (15.5)$$

Mivel a növekmény várható értéke zérus, a második tag eltűnik. Vegyünk fel egy tetszőleges $t + \tau$ pontot a $t + s$ és a t között:

$$\begin{aligned} E[(X(t+s) - X(t))^2] &= E[(X(t+s) - X(t+\tau) + X(t+\tau) - X(t))^2] = \\ &= E[(X(t+s) - X(t+\tau))^2] + E[(X(t+\tau) - X(t))^2] + \\ &+ 2 \cdot E[(X(t+s) - X(t+\tau)) \cdot (X(t+\tau) - X(t))] \end{aligned} \quad (15.6)$$

A független növekményűség felhasználásával, majd kihasználva, hogy a növekmények várható értéke zérus, a

$$\begin{aligned} E[(X(t+s) - X(t+\tau)) \cdot (X(t+\tau) - X(t))] &= \\ E[(X(t+s) - X(t+\tau))] \cdot E[(X(t+\tau) - X(t))] &= 0. \end{aligned} \quad (15.7)$$

tag eltűnik, így:

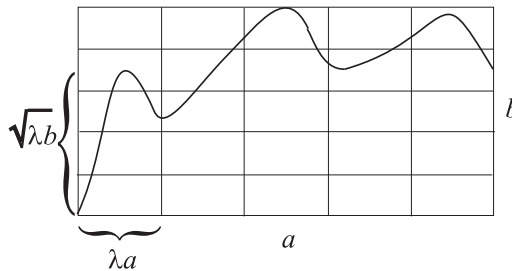
$$E[(X(t+s) - X(t))^2] = E[(X(t+s) - X(t+\tau))^2] + E[(X(t+\tau) - X(t))^2]. \quad (15.8)$$

Tehát a szórásnégyzetre a következő függvényegyenlet érvényes:

$$\sigma^2(s) = \sigma^2(s - \tau) + \sigma^2(\tau). \quad (15.9)$$

Mint arról behelyettesítéssel könnyen meggyőződhetünk, ezen függvényegyenlet megoldása a lineáris függvény, azaz $\sigma^2(s) = c \cdot s$, ahol c állandó.

Ez a görbe csak statisztikai értelemben önhasználó. Hasonlítsuk össze az eredeti $X(t)$ függvényt és a paraméter tartomány transzformálásával kapott $X(\lambda \cdot t)$ függvényt. A növekmény mindkettőben zérus várható értékű, normális eloszlású valószínűségi változó, de az eredeti szórásnégyzete t -vel arányos, a transzformálté pedig $\lambda \cdot t$ -vel. Ha tehát a paraméterében λ -val zsugorított függvényt értékben $\sqrt{\lambda}$ -val összenyomjuk, az eredeti $X(t)$ -vel statisztikailag megegyező folyamathoz jutunk.



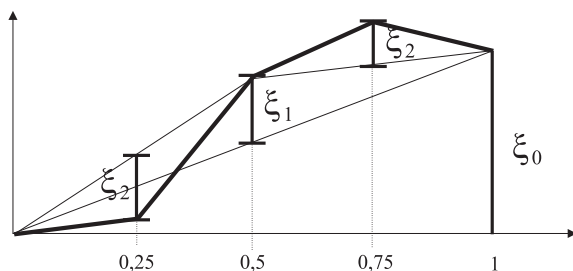
15.3. ábra. A Brown-mozgás dobozdimenziójának meghatározása

Ezt a tulajdonságot felhasználhatjuk a görbe *dobozdimenziójának* kiszámítására. Tekintsük a görbe egy szakaszát és foglaljuk egy téglalapba (15.3. ábra). Bontsuk fel a téglalapot $N \times N$ darab, az eredeti téglalapról λ hasonlósági transzformációval előállítható kis téglalapra és számláljuk meg, hány kis téglalapon megy át a görbénk. Egy oszlop a téglalap paramétertartománybeli $\lambda = 1/N$ zsugorításával állítható elő. A statisztikai önhasznóság miatt, a görbe várható magassága ekkor a téglalap magasságának $\sqrt{\lambda}$ -szorosa, tehát egy oszlopban a görbe átlagosan $N \cdot \sqrt{\lambda}$ dobozt metsz. Ez minden oszlopra így van, azaz a metszett dobozok száma:

$$n(\lambda) = N \cdot N \cdot \sqrt{\lambda} = \frac{1}{\lambda^{1.5}} \quad (15.10)$$

A Brown-mozgás dobozdimenziója tehát 1.5.

A Brown-mozgás egy realizációját közelítőleg például a véletlen középpont elhelyező algoritmussal állíthatjuk elő (a tényleges függvény, miként a Koch-görbénél, ezen folyamat határértékeként kapható meg). A görbét a $[0, 1]$ intervallumban közelítjük. A 0 pontban, a görbe értéke 0, az egy pontban pedig egy normális eloszlású valószínűségi változó, amelynek a szórásnégyzete σ^2 . Generáljunk tehát ilyen eloszlású ξ_0 véletlen számot és a kapott értéket rendeljük az 1 ponthoz. Most folytassuk a görbe pontjának meghatározását az intervallum felezőpontjában. Mind az intervallum kezdőpontjához képest, mind pedig az intervallum végpontjához képest a felezőpont normális eloszlású



15.4. ábra. Brown-mozgás trajektóriájának létrehozása véletlen középpont elhelyezéssel

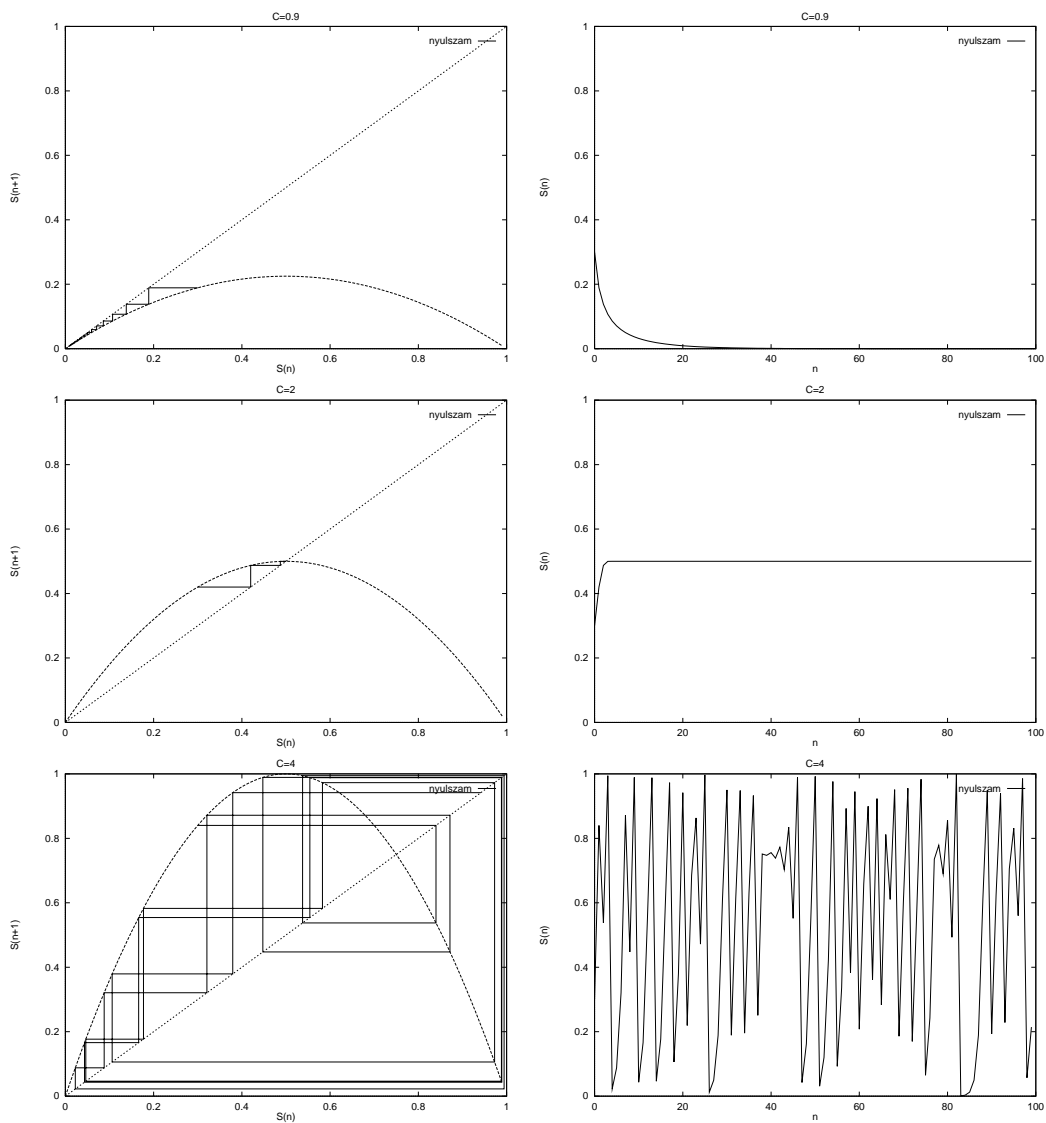
$\sigma^2/2$ szórással. Ha tehát a két végpont átlagát tekintjük, amelynek mindkét végponthoz képest $\sigma^2/2$ a szórása, az átlaghoz képest már csak egy $\sigma^2/4$ szórást véletlen változóval kell perturbálni a felezőpontban felvett értéket. Legyen a perturbáció a véletlen ξ_1 . A kiadódó két intervallumra ugyanez az eljárás folytatható, az n . lépésben a ξ_n perturbáció szórása $\sigma^2/2^{n+1}$.

A módszer könnyen kiterjeszthető felületekre. Egy egységnégyzetből indulunk ki és a négy sarokpontban véletlenszerűen kijelölünk egy értéket. A négyzet oldalfelező pontjaiban és a középpontjában lévő pontok magasságait a sarokpontok átlagának perturbációjaként kapjuk meg. Az új pontok segítségével a négyzetet 4 kis négyzetre bontjuk, és ugyanezt az eljárást rekurzívan folytatjuk, a perturbációk szórását minden rekurziós szinten felezve.

15.3. Kaotikus dinamikus rendszerek

Kaotikus dinamikus rendszeren olyan determinisztikus rendszert értünk, amely látszólag véletlenszerűen működik. Ebben semmi meglepő sincs, környezetünk tele van ilyen rendszerekkel. Például a csapból kifolyó vízre ható nehézségi erő egy nagyon egyszerű mechanikai rendszert képez, amelyet megoldva folyamatosan vékonyodó vízsugarat kellene kapnunk. Ez egy darabig így is van, de jobban kinyitva a csapot turbulens jelenségek lépnek fel, amelyek a viselkedést véletlenszerűvé teszik.

A jelenség lényegét egy klasszikus példán mutatjuk be, amely az egész témakör vizsgálatát elindította. A példa egy szigeten zárt közösségben élő nyulak számának évenkénti változását vizsgálja. Nyilván amíg kevés nyúl van, a táplálék nem okoz gondot, de a kevés papanyúl és mamanyúl következménye a kevés gyermeknyúl, így a nyúlpopuláció nő, de csak az aktuális nyúlszámmal arányosan. Ha viszont a nyulak száma nagy, a táplálékszűke miatt sokan éhenpusztulnak, mégpedig a nyulak számával arányosan. Legyen a sziget maximális teherbírására vetített normalizált nyúlszám az n . évben S_n . Mind a nyulak szaporulatát, mind a táplálék szűkösségét figyelembe véve, a



15.5. ábra. A nyúl populáció változása különböző C erősítési tényezőkre

nyúlzsám évenkénti változására a következő modell állítható fel:

$$S_{n+1} = C \cdot S_n \cdot (1 - S_n).$$

A modell szerint az S_{n+1} maximális ha $S_n = 0.5$, innen 0-ig a kevés papa és mama miatt, innen 1-ig pedig a répa és fű szűke miatt az S_{n+1} csökken.

Vizsgáljuk meg a nyúlzsám alakulását az egymást követő években különböző C értékekre (15.5. ábra). Ha C kicsi (jelen modellben 1-nél kisebb), a nyulak száma 0-hoz konvergál (15.5. ábra). Közepes C értéknél a nyúlzsám egy pozitív értékhez tart, azaz a nyulak száma állandósul a szigeten. Ez a két eset nem is annyira érdekes. Viszont ha C nagy, akkor a nyúlzsám kaotikusan ugrándozik, és a görbére nézve az az érzésünk támad, hogy az teljesen véletlenszerű. Pedig a görbét egy egyszerű, determinisztikus modell iterálásával állítottuk elő. A véletlenszerűségnek még van egy másik fontos ismérve is. Ha az iterációt két akármilyen közel lévő, de különböző pontból indítjuk, a görbék egy idő után teljesen különbözőkké válnak. Az ilyen jellegű mozgást nevezzük *kaotikus mozgásnak*.

Ezt a jelenséget matematikushoz illő módon is megfogalmazhatjuk. A kaotikus mozgásnál a mozgás autokorrelációs függvénye zérushoz tart, azaz egy idő után a korábbi pontok, így a kezdeti pont is teljesen jelentéktelenné válik. Ekkor viszont a mozgás teljesítménysűrűség spektruma — ami az autokorrelációs függvény Fourier-transzformáltjaként állítható elő — nem lesz sávkorlátozott. A véletlenszerűnek látszó viselkedést éppen az okozza, hogy a mozgásban akármilyen magas frekvenciák is előfordulhatnak.

Megjegyezzük, hogy a kaotikus, tehát véletlennak látszó, de azért mégiscsak determinisztikus folyamatokat aknázzák ki a *véletlenszám generátorok* is.

15.4. Kaotikus dinamikus rendszerek a síkon

A számítógépes grafikában a *kaotikus dinamikus rendszerek* azért fontosak, mert segítségével szép képeket állíthatunk elő. Mivel a kép két dimenziós, ezért olyan rendszereket fogunk vizsgálni, ahol az aktuális állapot a sík egy pontjának felel meg. Algebrailag a sík egy pontját egy x, y valós számpárral, vagy akár egyetlen z komplex számmal is jellemezhetjük. A rendszer mozgása során egy pontsorozatot — ún. trajektóriát — jár be, amelyet megjelenítve képeket kaphatunk.

Tekintsük példaként a

$$z_{n+1} = F(z_n) = z_n^2$$

rendszer, amely az aktuális állapotot jellemző komplex számot minden iterációban négyzetre emeli. Polárkoordinátákban ($z_n = r_n \cdot e^{j\phi_n}$) az iterációs formula:

$$r_{n+1} = r_n^2, \quad \phi_{n+1} = \phi_n \cdot 2. \quad (15.11)$$

Nyilván ha $r_0 > 1$, akkor az iteráció során a szám abszolút értéke divergens lesz, tehát az iteráció a végtelenbe visz bennünket. Ha viszont $r_0 < 1$, akkor az abszolút érték és ezáltal a szám is a zérushoz konvergál. A 0 és a ∞ az iteráció *fixpontjai*, mert ezeket behelyettesítve az iterációs képletbe a következő pontként önmagukat kapjuk vissza.

A konvergens és divergens esetek közötti határeset az $r_0 = 1$ esete. Ekkor a szám abszolút értéke végig 1 marad, fázisszöge pedig a $\phi_0 \cdot 2^n$ értékeket veszi fel, azaz az iteráció az egységkörön kalandozik (mégpedig kaotikus módon). Az egységkör bármely pontjából is indulunk, az iteráció nem térít le bennünket az egységkörrel. A kör bármely z pontjára az $y = F(z)$ pont is a körön van, és megfordítva, bármely y körön lévő ponthoz találunk a körön olyan z pontot, amelyre $y = F(z)$. Ezt röviden úgy is megfogalmazhatjuk, hogy az egységkör pontjainak H halmaza kielégíti a

$$H = F(H) \quad (15.12)$$

halmazegyenletet, azaz a H halmaz az iteráció fix "pontja". Definíciószerűen egy F függvényre a $H = F(H)$ egyenletet kielégítő halmazt a függvény *attraktorának* nevezük.

Egy fixpontot, vagy akár a teljes attraktort *stabilnak* mondunk, ha egy kicsit elmozdulva onnan az iterációt folytatva visszatalálunk a fixpontba illetve az attraktorba. A fixpont illetve az attraktor *labilis*, ha az elmozdítás után az iteráció egyre jobban távolodik a fixponttól illetve az attraktortól. A labilis fixpont egy hegycsúcsnak, a labilis attraktor egy hegygerincnek, a stabil fixpont egy mélyedésnek, a stabil attraktor pedig egy völgynek felel meg. Mivel az attraktor két különböző fixpont vonzáskörzetének a határa, az attraktor akkor stabil, ha a két fixpont labilis, és megfordítva az attraktor akkor labilis, ha a két fixpont stabil. A $z_{n+1} = z_n^2$ iterációban a 0 és a ∞ stabil fixpontok, az egységkör pedig labilis attraktor.

15.4.1. Julia-halmazok

Számunkra az attraktorok azért érdekesek, mert általában igen összetett alakúak és fraktális tulajdonságúak. A z^2 függvényre ez még nem igaz, de csak egy kicsit kell módosítani rajta, hogy valóban bonyolult attraktorokhoz jussunk. Tekintsük az

$$F(z) = z^2 + c, \quad \text{ahol } c \text{ tetszőleges komplex szám} \quad (15.13)$$

függvényt, amelynek attraktorait (15.7. ábra) *Julia-halmazoknak* nevezzük (a férfi olvasókat ki kell ábrándítanom, a Julia nem egy szép hölgy keresztnéve, hanem egy közel egy évszázada élt francia matematikus vezetékneve).

Egy függvény attraktorát alapvetően két algoritmussal jeleníthetjük meg, attól függően, hogy az attraktor stabil vagy labilis.

Labilis attraktorok megjelenítése

Ha az attraktor labilis, akkor bármely pontból kiindulva előbb-utóbb valamelyik fixpontot közelítjük meg, még akkor is, ha valamilyen csoda folytán a kezdeti pont az attraktoron van, hiszen a számítási pontatlanságok miatt úgyis el fogunk onnan távolodni. A sík minden pontjából tehát egy iterációt kell indítanunk, és megvizsgálunk, hogy az hova konvergál (divergál), ami alapján az egyes pontok vonzáskörzetekhez rendelhetők. Az egyik vonzáskörzetet más színűre színezve mint a másikat, egy kitöltött Julia-halmazt kapunk, amelynek határa a tényleges attraktor. Legyen például a divergens tartomány fehér, a konvergens tartomány fekete.

A sík összes pontjának tesztelése nyilván lehetetlen, de szerencsére kihasználhatjuk, hogy a képernyőnk úgyis véges számú pixelből áll, ezért elég csak a pixelközéppontoknak megfelelő komplex számokra elvégezni a vizsgálatot.

Tegyük fel, hogy az $X_{\max} \times Y_{\max}$ felbontású képernyőt úgy helyezzük rá a komplex számsíkra, hogy a bal alsó sarok a $x_l + j \cdot y_b$ komplex számra, a jobb felső pedig a $x_r + j \cdot y_t$ komplex számra kerüljön. Tehát a nézet a $[(0, 0), (X_{\max}, Y_{\max})]$ téglalap, az ablak pedig az $[(x_l, y_b), (x_r, y_t)]$. A nézetből az ablakra vetítő transzformáció:

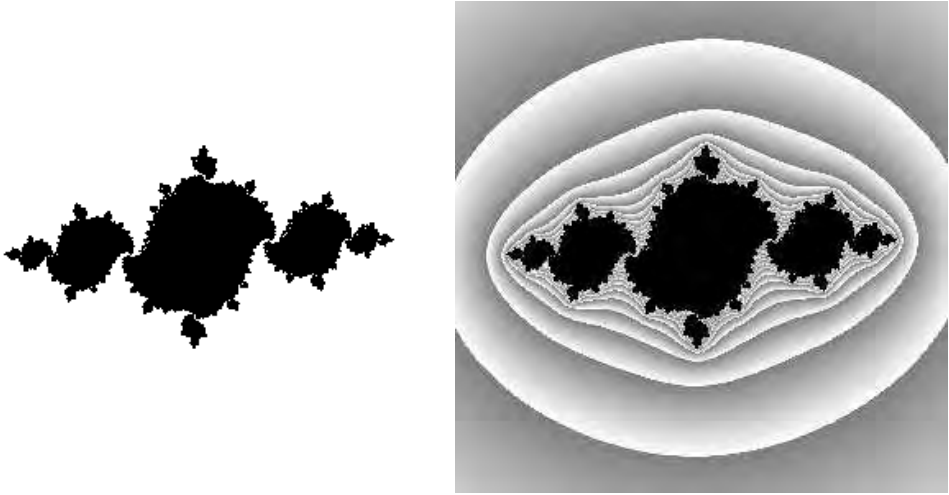
```
ViewportWindow( X, Y → x, y)
    x = x_l + (x_r - x_l) · X/X_max
    y = y_b + (y_t - y_b) · Y/Y_max
end
```

A rajzoló program, amely a Julia-halmaznak az ablakon belüli részét rajzolja:

```
JuliaDraw( c )
  for Y = 0 to Y_max do
    for X = 0 to X_max do
      ViewportWindow(X, Y → x, y)
      z = x + j · y
      for i = 0 to n do z = z2 + c
      if |z| > "infinity" then Pixel(X, Y, fehér)
      else Pixel(X, Y, fekete)
      endfor
    endfor
  end
```

Az algoritmusban a divergenciát csak közelítőleg vizsgálhatjuk. Az iterációs formulát például $n = 10^6$ -szor alkalmazzuk, majd ellenőrizzük, hogy a kapott komplex szám abszolút értéke például "infinity"= 10^4 -t meghaladja-e vagy sem.

Úgynevezett *színes Julia-halmazok* kapunk, ha a divergencia sebessége alapján egy színinformációt rendelünk a kiindulási állapotot meghatározó pixelhez, azaz a színt a $|z_n|$ alapján számítjuk.



15.6. ábra. Kitöltött és színes Julia-halmazok a $-2 \leq \Re z \leq 2, -2 \leq \Im z \leq 2$ tartományban ($c = -0.9 + 0.12j$)

Stabil attraktorok megjelenítése

A stabil attraktorok előállításakor könnyű dolgunk van. Bárhonnan is indulunk el, előbb-utóbb megközelítjük az attraktort, majd ettől kezdve az attraktor pontjait látogatjuk meg. Tehát csak iterálni kell, és az első néhány pont eldobása után kiszínezni a meglátogatott pontoknak megfelelő pixeleket.

Ez a módszer az $F(z) = z^2 + c$ iteráció során nem működik, hiszen az attraktor labilis. Ha viszont kihasználjuk azt a felismerést, hogy ha egy H halmaz az F -nek attraktora, azaz $F(H) = H$, akkor az inverz függvénynek is attraktora, hiszen ekkor $F^{-1}(H) = H$. Ráadásul ami eredetileg stabil volt, az labilissá válik, és megfordítva, ami eredetileg labilis volt abból stabil lesz. Mivel ekkor a függvény inverzét iteráljuk, az eljárás neve *inverz iterációs módszer*.

A $F(z) = z^2 + c$ inverze nem függvény, csupán leképzés, mert nem egyértelmű:

$$F^{-1}(z) = \pm\sqrt{z - c}. \quad (15.14)$$

Ez azt jelenti, hogy ha egyetlen pontból indítjuk az iterációt, az első lépés után 2, a második után 4, az n . után 2^n pontot kell kezelnünk, amely keserves memóriagon-dotokat okozhat. Felmerül a kérdés, hogy nincs-e olyan stratégia, amely a $+\sqrt{z - c}$ és $-\sqrt{z - c}$ közül mindig csak egyet választ ki, mégis az iteráció elegendően sűrűn bejárja a teljes attraktort. A probléma megvilágításának érdekében tegyük fel, hogy $c = 0$,

amikor az attraktor az egységkör, a polárkoordinátás iterációs formulák pedig:

$$r_{n+1} = \sqrt{r_n} = r_0^{1/2^n}, \quad \phi_{n+1} = \begin{cases} \phi_n/2, & \text{ha } a + \sqrt{z - c} \text{ függvényt alkalmazzuk,} \\ \phi_n/2 + \pi, & \text{ha } a - \sqrt{z - c} \text{ függvényt alkalmazzuk.} \end{cases} \quad (15.15)$$

Az abszolút érték bármely kezdeti értékről rohamléptekkel tart 1-hez, így gyorsan megközelítjük az attraktort. A fázisszög vizsgálatára vezessünk be egy indikátor változót. Legyen $\delta_n = 1$, ha a $-\sqrt{z - c}$ formulát alkalmazzuk az n . iteráció során, különben pedig 0. Ezzel az indikátorváltozóval a fázisszög az n . iteráció után:

$$\phi_{n+1} = \frac{\phi_0}{2^n} + \pi \cdot \left(\delta_n + \frac{\delta_{n-1}}{2} + \frac{\delta_{n-2}}{4} + \dots + \frac{\delta_0}{2^n} \right). \quad (15.16)$$

Vegyük észre, hogy a kezdeti pont ϕ_0 hatása rohamosan eltűnik a képletből, tehát a mozgás kaotikus. Mivel a δ számok 0 és 1 értékeket vehetnek fel, a zárójelben megadott összeget úgy is elképzelhetjük, mint egy 2-s számrendszerben felírt kettedes törtszámot:

$$\phi_{n+1} = \frac{\phi_0}{2^n} + \pi \cdot (\delta_n \cdot \delta_{n-1} \delta_{n-2} \dots \delta_0) \quad (15.17)$$

A törtszám hossza minden iterációban nő, de a tört végén levő biteknek már nincs különösebb jelentőségük. Ezért amikor azt vizsgáljuk, hogy az iteráció során valóban megfelelő sűrűn lefedjük-e az attraktort, elegendő csak a tört első N jegyét tekinteni:

$$\phi_{n+1} \approx \pi \cdot (\delta_n \cdot \delta_{n-1} \delta_{n-2} \dots \delta_{n-N}). \quad (15.18)$$

Az attraktor megfelelő sűrű lefedéséhez az szükséges, hogy a $[0..2\pi]$ tartományban előállított fázisszögek között ne legyen nagy lyuk, azaz, hogy a $(\delta_n \cdot \delta_{n-1} \delta_{n-2} \dots \delta_{n-N})$ bináris számban mindenféle lehetséges kombináció előforduljon. Ezt legegyszerűbben úgy biztosíthatjuk, ha a δ számokat egymástól függetlenül és véletlenszerűen választjuk ki.

Általában is igaz, hogy a többértékű leképezések iterációja során elegendő minden lépésben csak az egyik alkotó függvényt alkalmazni, ha azt véletlenszerűen választjuk ki.

Érdekes megfigyelnünk, hogy az alkotó függvények kiválasztási valószínűségének (amennyiben az nem zérus) semmiféle hatása sincs az attraktor alakjára. Csupán azt határozza meg, hogy az iteráció során az attraktor mely tartományait milyen valószínűséggel látogatjuk meg.

A programban ismét feltételezzük, hogy a nézet a $[(0, 0), (X_{\max}, Y_{\max})]$ téglalap, az ablak pedig az $[(x_l, y_b), (x_r, y_t)]$ téglalap. Most alapvetően a komplex számsíkon mozgunk és onnan vetítünk a képernyőre, tehát az ablakból a nézetbe átvivő transzformációra van szükségünk:

```

WindowViewport(  $x, y \rightarrow X, Y$  )
     $X = X_{\max} \cdot (x - x_l) / (x_r - x_l)$ 
     $Y = Y_{\max} \cdot (y - y_b) / (y_t - y_b)$ 
end

```

E vetítés által kijelölt pixel akkor van a képernyő belsejében, ha az eredeti pont az ablakon belül van. Ennek ellenőrzésére használhatjuk a következő pontvágó rutint:

```

BOOL ClipWindow(  $x, y$  )
    if  $(x_l \leq x \leq x_r \text{ AND } y_b \leq y \leq y_t)$  then return TRUE
    else return FALSE
end

```

A Julia-halmaz előállítására inverz iterációs módszerrel:

```

JuliaInverseIteration(  $c$  )
    Kezdeti  $z$  érték választása
    for  $i = 0$  to  $n$  do
         $x = \Re z$ 
         $y = \Im z$ 
        if ClipWindow( $x, y$ )
            WindowViewport( $x, y \rightarrow X, Y$ )
            Pixel( $X, Y$ , fekete)
        endif
         $z = \sqrt{z - c}$ 
        if  $\text{rand}() > 0.5$  then  $z = -z$ 
    endfor
end

```

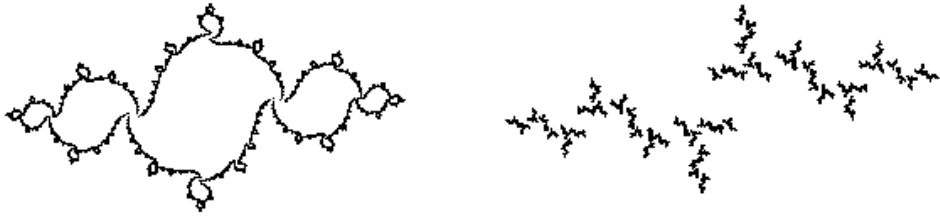
//z valós része
//z imaginárius része

A kezdeti z érték egyetlen jelentősége az, hogy ha túl távol van az attraktortól, akkor az iterációból sok kezdeti elemet kell figyelmen kívül hagyni. A legszerencsésebb, ha rögtön az attraktorból indulunk. A $\sqrt{z - c}$ illetve a $-\sqrt{z - c}$ fixpontjai biztosan részei az attraktornak, ezért célszerű valamelyiket kijelölni kezdeti értékként:

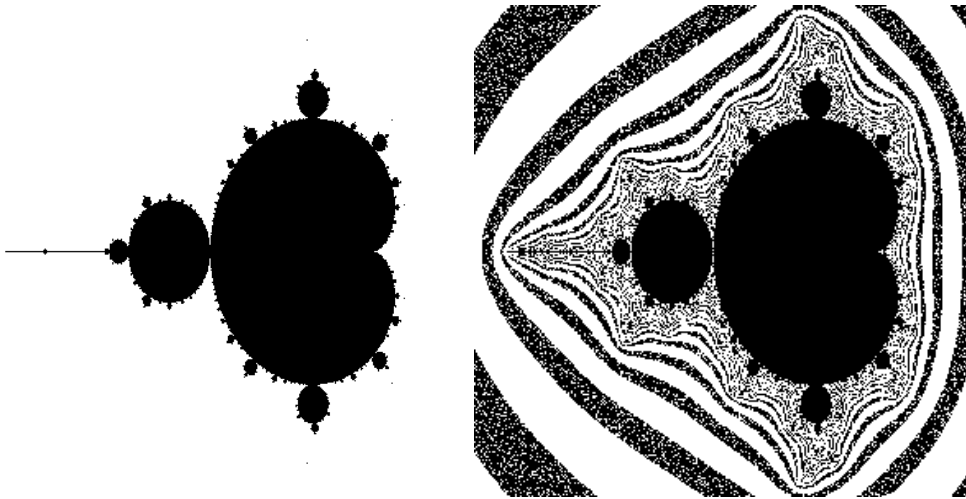
$$z = \pm\sqrt{z - c} \implies z^2 - z + c = 0 \implies z = \frac{1}{2} + \sqrt{\frac{1}{4} - c}. \quad (15.19)$$

15.4.2. A Mandelbrot-halmaz

Különböző komplex c számokkal előállított Julia-halmazokat (15.7. ábra) nézegetve megállapíthatjuk, hogy azok vagy összefüggő halmazok, vagy különálló, egymással nem érintkező pontok gyűjteményei (ún. *Cantor-féle halmaz*). Egy Mandelbrot nevű matematikus arra keresett választ, hogy c -nek milyen tulajdonságúnak kell lennie ahhoz, hogy a Julia-halmaz összefüggő legyen, és hogy nevét megörökítse, el is nevezte az ilyen tulajdonságú komplex számok halmazát *Mandelbrot-halmaznak*.



15.7. ábra. A Julia-halmazok a $-2 \leq \Re z \leq 2, -2 \leq \Im z \leq 2$ tartományban: bal:
 $c = -0.9 + 0.12j$; jobb: $c = -1.2 + 0.4j$



15.8. ábra. A normál és a színes Mandelbrot-halmaz

Miként az viszonylag könnyen belátható [PSe88], a Julia-halmaz akkor lesz összefüggő, ha a c komplex szám része vagy a Julia-halmaz attraktorának, vagy a konvergencia tartományának. Ennek megfelelően a c komplex számról úgy dönthető el, hogy része-e a Mandelbrot-halmaznak, hogy megvizsgáljuk, hogy a $z_{n+1} = z_n^2 + c$ iteráció divergens-e a $z_0 = c$ értékre (vegyük észre, hogy $z_0 = 0$ kezdeti értékkel is dolgozhatunk, hiszen az első lépés ekkor éppen a c -be visz).

A Mandelbrot-halmaz rajzoló program tehát:

```

MandelbrotDraw()
  for  $Y = 0$  to  $Y_{\max}$  do
    for  $X = 0$  to  $X_{\max}$  do
      ViewportWindow( $X, Y \rightarrow x, y$ )
       $c = x + j \cdot y$ 
       $z = 0$ 
      for  $i = 0$  to  $n$  do  $z = z^2 + c$ 
        if  $|z| > \text{"infinity"}$  then Pixel( $X, Y$ , fehér)
        else Pixel( $X, Y$ , fekete)
      endfor
    endfor
  endfor
end

```

A Julia-halmazokhoz hasonlóan *színes Mandelbrot-halmazok*at is előállíthatunk, ha nem csupán a divergencia meglétét ellenőrizzük, hanem a divergencia sebessége alapján (a $|z_n|$ felhasználásával) színezzük ki a kezdeti pontot.

15.5. Iterált függvényrendszerek

A Julia-halmazok, bár nagyon szépek, nem kellően változatosak. A változatosság hiánya abból adódik, hogy a Julia-halmazt definiáló egyetlen komplex szám túlságosan kicsiny szabadságfokot biztosít számunkra az attraktor kialakítására. Ezért érdemes más leképezésekkel dolgozni, amelyekben a szabad paraméterek száma az igényeknek megfelelően változtatható.

A legegyszerűbb függvény, ami eszünkbe juthat, a lineáris, amely a sík x, y pontját a következőképpen képezi le:

$$[x', y'] = W(x, y) = [x, y] \cdot \mathbf{A} + \vec{p} \quad (15.20)$$

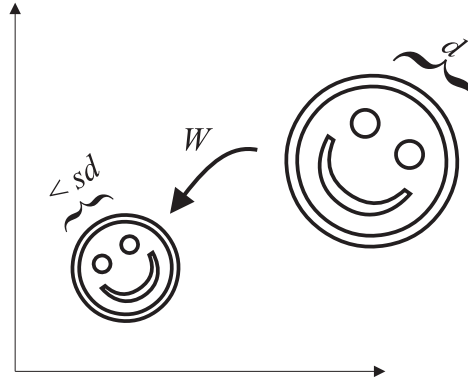
Ennek az iterációnak két fixpontja van. Az egyik a

$$[x, y] = [x, y] \cdot \mathbf{A} + \vec{p} \quad (15.21)$$

egyenlet véges megoldása, a másik a végtelen. A lineáris függvénnyel végzett iteráció konvergens, ha az \mathbf{A} mátrixszal való szorzás *kontrakció*, azaz

$$\|\vec{u} \cdot \mathbf{A}\| \leq s \cdot \|\vec{u}\| \quad (15.22)$$

ahol $s < 1$. A kontrakció szemléletes jelentése az, hogy ha egy halmazra végrehajtjuk a leképezést, akkor a halmaz pontjai közötti eredeti távolságok legalább $s < 1$ -szeresükre zsugorodnak (15.9. ábra). A leképezés akkor kontrakció, ha az \mathbf{A} mátrix valamely normája 1-nél kisebb.



15.9. ábra. Kontraktív leképezés

Az iteráció mindenképpen egyetlen pontra fog rázsugorodni (ha a végtelent is egy pontnak tekintjük), tehát a $H = F(H)$ halmazegyenletet kielégítő halmaz, azaz az attraktor, egyetlen pontból áll. Az egyetlen pontból álló képek pedig nem különösebben izgalmasak. A lineáris függvényeket mégsem kell teljesen elvetnünk. Több lineáris függvény alkalmazásával ugyanis többértékű lineáris leképezéseket építhetünk fel, amelyeknek már sokkal szebb attraktora van. Legyen tehát a leképezésünk:

$$F(x, y) = W_1(x, y) \vee W_2(x, y) \vee \dots \vee W_n(x, y). \quad (15.23)$$

Az F attraktorát iterációs módszerrel fogjuk előállítani, amelyre akkor van esélyünk, ha az attraktor stabil. Az attraktor stabilitásának feltétele az, hogy a végtelen egyik lineáris függvénynek se legyen stabil fixpontja, azaz minden alkotó lineáris függvény kontrakció legyen.

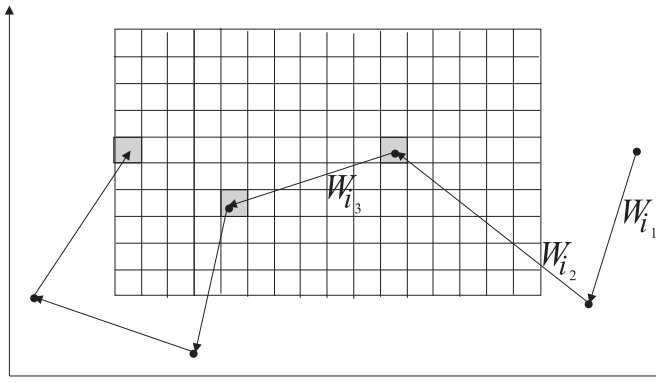
15.5.1. Iterált függvényrendszerek attraktorának előállítása

Ha mindegyik alkotó lineáris leképezés kontrakció, akkor az attraktor stabil, tehát bármely pontból indulunk is, az iteráció az attraktorhoz fog konvergálni, és ha már az attraktoron vagyunk, akkor ott is maradunk. Miként azt a többértékű leképezések attraktorának vizsgálatánál megállapítottuk, elegendő a többértékű leképezésből annak egyetlen

függvényét alkalmazni minden egyes iterációs lépésben, ha a választást véletlenszerűen tesszük meg. Rendeljük a lineáris függvényekhez egy valószínűséget oly módon, hogy az i . függvényt minden lépésben, egymástól függetlenül p_i valószínűséggel választjuk. Amíg a p_i valószínűségek nem nullák, a tényleges megválasztásuk semmiféle hatással sincs az attraktorra, csak az attraktor tartományainak a meglátogatási valószínűségébe szól bele. Elképzelhetjük úgy is az iterációs folyamatot, mint egy részeg sofőr által vezetett, homokot szállító teherautó bolyongását. A következő pontot az aktuális pontból egy véletlenül választott függvénnyel állítjuk elő, majd a pontra érkeve ledobunk egy lapát homokot. Elegendően sokat bolyongva a homokunkat az attraktorra terítjük szét. A lineáris függvények meghatározzák, hogy hol lesz homok, a valószínűségek pedig azt, hogy hol lesz nagyobb és hol kisebb kupac. A homok mennyisége alapján az attraktor pontjaihoz színinformációt rendelhetünk.

A $W_1(x, y), W_2(x, y), \dots, W_n(x, y)$ lineáris függvények halmazát az alkalmazásuk p_1, p_2, \dots, p_n valószínűségével együtt *iterált függvényrendszernek* (*Iterated Function System*) vagy *IFS*-nek nevezzük.

Miként a Julia-halmaz iterációs elvű rajzolásánál láttuk, célszerű már a kezdeti pontot is az attraktoron kiválasztani, mert ekkor nem kell eldobni az iteráció első néhány pontját sem. Egy alkalmas attraktorbeli pont lehet bármely lineáris függvény véges fixpontja, amit a 15.21 egyenlet megoldásával kaphatunk.



15.10. ábra. IFS rajzolás véletlen bolyongással

Az IFS rajzoló programunk ezek után:


```

IFSDraw()
  for  $X = 0$  to  $X_{\max}$  do for  $Y = 0$  to  $Y_{\max}$  do  $m[X][Y] = 0$ 
   $[x, y]$  kezdeti értéke az  $[x, y] = [x, y] \cdot \mathbf{A}_1 + \vec{p}_1$  megoldása
  for  $i = 0$  to  $n$  do
    if ClipWindow( $x, y$ )
      WindowViewport( $x, y \rightarrow X, Y$ )
       $m[X][Y]++$ 
    endif
     $k$  választása véletlenszerűen  $p_k$  valószínűséggel
     $[x, y] = [x, y] \cdot \mathbf{A}_k + \vec{p}_k$ 
  endfor
  for  $X = 0$  to  $X_{\max}$  do for  $Y = 0$  to  $Y_{\max}$  do
    Pixel( $X, Y, color(m[X][Y])$ )
  endfor
end

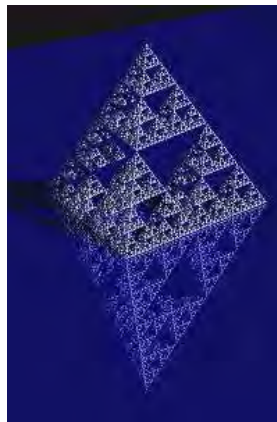
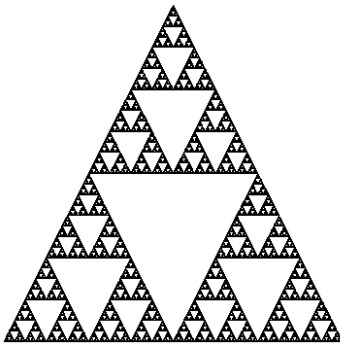
```

A k index p_k valószínűséggel történő kiválasztása a következőképpen lehetséges:

```

 $k = 1, s = p_1$ 
 $r = rand()$ 
while  $s < r$  do  $s += p_k, k++$ 

```



15.11. ábra. IFS által definiált egyszerű rajzok: 2D és 3D Sierpienski-halmazok (a 2D Sierpienski-halmaz IFS-e 3 lineáris transzformációt tartalmaz), és egy páfrány (az IFS 4 lineáris transzformációt tartalmaz)

15.5.2. IFS modellezés

Az előző fejezetben azt tárgyaltuk, hogy egy IFS által definiált attraktort hogyan lehet megjeleníteni. Most a másik irányt fogjuk követni, amikor adott egy T halmaz és keresünk azt az IFS-t, tehát azon $W_1(x, y), W_2(x, y), \dots, W_n(x, y)$ lineáris függvényeket, amelyek éppen ezt állítják elő. Az attraktort definiáló egyenlet:

$$T = W_1(T) \cup W_2(T) \cup \dots \cup W_n(T). \quad (15.24)$$

Emlékezzünk vissza, hogy az attraktor stabilitásának az a feltétele, hogy a $W_i(T)$ a T halmaz kicsinyítése legyen, tehát ezen egyenlet szerint az IFS definiálásához a halmazunkat saját kicsinyített képeivel kell lefedni.

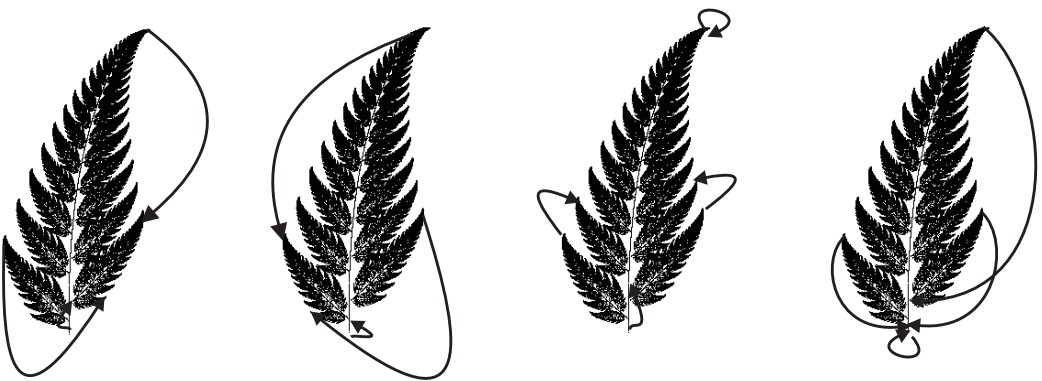
A modellezés általános programja tehát

```

IFSModel( )
   $i = 0$ 
  repeat
     $i++$ 
    Keress  $W_i$ -t úgy, hogy  $W_i(T) \subseteq T$ 
  until  $T = W_1(T) \cup W_2(T) \cup \dots \cup W_i(T)$ 
end

```

A halmaz részét lefedő W_i transzformációkat létrehozhatjuk manuális vagy automatikus eljárásokkal egyaránt. A manuális eljárás, amelynek neve a *referencia pontok módszere*, az eredeti halmazból 3 pontra mondja meg, hogy az hova kerüljön. Ezzel 6 skaláregyenletet állítunk fel, amelyeket megoldva a W_i leképezés 6 paramétere kiszámítható.



15.12. ábra. IFS modellezés a referencia pontok módszerével

Az 15.12. ábra a páfrányt definiáló 4 lineáris transzformáció előállítását mutatja be. Az első transzformáció a páfrányt a jobb alsó levelébe viszi át:

$$W_1(x, y) = [x, y] \cdot \begin{bmatrix} -0.15 & 0.26 \\ 0.28 & 0.24 \end{bmatrix} + [0, 100].$$

A második a páfrányt a bal alsó levelére képezi le:

$$W_2(x, y) = [x, y] \cdot \begin{bmatrix} 0.2 & 0.23 \\ -0.26 & 0.22 \end{bmatrix} + [0, 44].$$

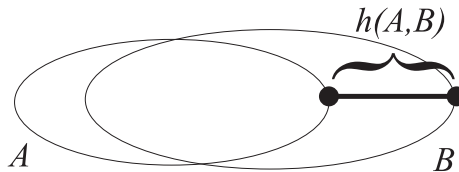
A harmadik egy csöppet kicsinyít, és a páfrányt arra a részére vetíti, amelyből a két alsó levél és a közöttük lévő függőleges szárdarab hiányzik:

$$W_3(x, y) = [x, y] \cdot \begin{bmatrix} 0.85 & -0.04 \\ 0.04 & 0.85 \end{bmatrix} + [0, 160].$$

Végül az utolsó erősen kicsinyít függőleges irányban és brutálisan összenyomja a páfrányt vízszintes irányban, hogy előállítsa a szárnak az alsó két levél közötti részét:

$$W_4(x, y) = [x, y] \cdot \begin{bmatrix} 0 & 0 \\ 0 & 0.16 \end{bmatrix} + [0, 0].$$

A gyakorlati esetekben az eredeti halmazunkat általában csak közelítőleg tudjuk lefedni a halmaz kicsinyített képeivel, így a kapott IFS attraktora az eredeti halmaztól eltér. A hiba nagysága egyrészt a lefedés hibájától, másrészt a lineáris függvények kontrakciójától függ.



15.13. ábra. Hausdorff-távolság

Két ponthalmaz közötti eltérés számszerűsítésére a *Hausdorff-távolság*ot használhatjuk, amely azon utak maximumát jelenti, amelyet az egyik halmaz pontjaiból elindulva kell megtenni, hogy a másik halmaz legközelebbi pontjába eljussunk. A Hausdorff-távolság teljesíti a matematikában használt távolságfogalom kritériumait: nem negatív, egy halmaz önmagától vett távolsága zérus, és fennáll a háromszög-egyenlőség.

A közelítő lefedés miatt az IFS attraktora és az eredeti T halmaz nem fog megegyezni. A távolságukat a *kollázs tétel* fogalmazza meg. A kollázs tétel kimondja, hogy ha a T halmaz és a $W(T) = \cup W_i(T)$ lefedésének távolsága ε -on belül van, azaz

$$h(T, W(T)) < \varepsilon,$$

akkor a $W(T)$ leképezés \mathcal{A} attraktora és a T halmaz közötti távolságra fennáll a következő egyenlőtlenség:

$$h(T, \mathcal{A}) < \frac{\varepsilon}{1-s}, \quad (15.25)$$

ahol s a $W_i(T)$ függvények kontrakciói közül a maximális.

A tétel bizonyítása során egyrészt a háromszög-egyenlőtlenséget használjuk ki, másrészt pedig a leképezés kontraktív tulajdonságát. A feltétel szerint $h(T, W(T)) < \varepsilon$. Ha T -t és $W(T)$ -t is leképezzük a W leképezéssel, akkor a kontrakció miatt minden távolság s -szeresére zsugorodik, tehát:

$$h(W(T), W^2(T)) < s \cdot \varepsilon.$$

Teljesen analóg módon $h(W^i(T), W^{i+1}(T)) < s^i \cdot \varepsilon$. Tekintsük a $h(T, W^i(T))$ távolságot. A háromszög-egyenlőtlenség felhasználásával

$$\begin{aligned} h(T, W^i(T)) &< h(T, W(T)) + h(W(T), W^2(T)) + \dots + h(W^{i-1}(T), W^i(T)) < \\ &\varepsilon + s\varepsilon + \dots + s^{i-1}\varepsilon. \end{aligned} \quad (15.26)$$

Mivel az attraktor definíciója szerint $\mathcal{A} = \lim_{i \rightarrow \infty} W^i(T)$, a T halmaz és az \mathcal{A} attraktor távolsága:

$$h(T, \mathcal{A}) = \lim_{i \rightarrow \infty} h(T, W^i(T)) < \varepsilon + s\varepsilon + s^2\varepsilon + \dots = \frac{\varepsilon}{1-s}. \quad (15.27)$$

Ezzel a kollázs tételt bebizonyítottuk.

15.5.3. Fraktális képtömörítés

Az IFS rendszerek egyik legfontosabb alkalmazási területe a *képtömörítés* (*image compression*). Láttuk ugyanis, hogy egyszerű, néhány paraméterrel megadható IFS-k milyen bonyolult attraktorokat eredményezhetnek. Mivel az IFS a képet saját transzformáltjával fedi le, ez a módszer akkor hatékony, ha a képben sok önhasonló részlet ismerhető fel. Természetes objektumoknál ez a feltétel teljesül. A *fraktális képtömörítéshez* az idáig megismert IFS fogalom némi általánosításra szorul, egyrészt azért, hogy árnyalatos, illetve színes képeket is kezelni tudjon, másrészt azért, hogy a hatékonyság növelésének érdekében az önhasonlóságot csak ott erőltesse, ahol az tényleg fennáll.

A sík egyes pontjaihoz tehát egy g szürkességi szintet (általános esetben R, G, B értékeket) rendelünk, a lineáris transzformációt pedig még két paraméterrel egészítjük ki. Az egyik paraméter skálázza a pont szürkességi szintjét, a másik pedig kijelöli a kép azon részalmazát, amelyre a transzformáció végrehajtandó. Ezeket a rendszereket *particionált IFS*-nek vagy *PIFS*-nek nevezzük [Fis97].

15.6. Program: IFS rajzoló

Az affin leképezések inicializálásához, végrehajtásához és a fixpont megkereséséhez mindenekelőtt a 2D transzformációs osztályt egészítjük ki.

```
//=====
class Transform2D {
//=====
    double m[3][3];
public:
    ....
    Transform2D( double m00, double m01, double m10, double m11,
                double m20, double m21 ) {
        m[0][0] = m00; m[0][1] = m01; m[0][2] = 0;
        m[1][0] = m10; m[1][1] = m11; m[1][2] = 0;
        m[2][0] = m20; m[2][1] = m21; m[2][2] = 1;
    }
    Point2D AffineTransform( Point2D& p ) {
        return Point2D( p.X() * m[0][0] + p.Y() * m[1][0] + m[2][0],
                       p.X() * m[0][1] + p.Y() * m[1][1] + m[2][1]);
    }
    Point2D FindFixPoint( ) {
        double det = (m[0][0]-1.0)*(m[1][1]-1.0)-m[0][1]*m[1][0];
        return Point2D( (m[0][1]*m[2][1]-m[2][0]*(m[1][1]-1))/det,
                       (m[1][1]*m[2][0]-(m[0][0]-1)*m[2][1])/det);
    }
};
```

A virtuális világmodell jelen esetben egy IFS, azaz affin leképezések és az alkalmazási valószínűségük gyűjteménye.

```
//=====
class VirtualWorld {
//=====
    Array<Transform2D> w;
    Array<double> p;
public:
    Transform2D& AffineMap( int i ) { return w[i]; }
```

```

double MapProbability( int i ) { return p[i]; }
void AddMap( Transform2D& w0, double p0 ) {
    w[ w.Size() ] = w0; p[ p.Size() ] = p0;
}
int MapNum() { return w.Size(); }
};

```

A szintér továbbra is a virtuális világot, a kamerát és a megjelenítőeszközt foglalja magában. A képszintézis véletlen bolyongással történik. Az első leképzés fixpontjából indulunk, majd az alkalmazási valószínűségek szerint választunk egy leképzést, amivel transzformáljuk az aktuális pontot. Az így előállított pontsorozat azon elemeinek képét, amelyek az ablak belsejében vannak, a képernyőn kiszínezzük.

```

//=====
class Scene {
//=====
    Window *        scr;
    VirtualWorld    world;
    Camera2D        camera;
public:
    Scene( Window * ps ) { scr = ps; }
    void Render( );
};

//-----
void Scene :: Render( ) {
//-----
    unsigned long niteration = 100000L;

    scr -> Clear( );
    Point2D p = world.AffineMap(0).FindFixPoint( );
    for( long i = 0; i < niteration; i++ ) {
        if ( camera.Window( ).Code( p ) == 0 ) {
            Point2D ip = camera.ViewTransform().AffineTransform(p);
            scr -> Pixel( ip.X(), ip.Y(), Color(1) );
        }
        double r = RND, sum = world.MapProbability(0);
        int k = 1;
        while( sum <= r ) sum += world.MapProbability( k++ );
        p = world.AffineMap( k-1 ).AffineTransform( p );
    }
}

```

16. fejezet

Számítógépes animáció

Az *animáció* a virtuális világ és a képszintézis időbeli változásainak a követését jelenti. Elméletileg a virtuális világ és a kamera bármilyen paramétere definiálható időfüggvényként, legyen az pozíció, orientáció, méret, szempozíció, szín, normálvektor, BRDF, alak, stb., de ebben a könyvben csak a mozgás és a kamera animációval foglalkozunk.

A mozgás megjelenítéséhez az animáció nem csupán egyetlen képet generál, 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. Figyelembe véve, hogy az objektumaink geometriáját az objektumok lokális modellezési koordinátarendszereiben adjuk meg, az objektum pozíciója illetve orientációja a modellezési transzformáció változtatásával vezérelhető. Ez a modellezési transzformáció a világ-koordinátarendszerbe helyezi át az objektumot, ahol a kamera relatív pozícióját és orientációját is meghatározzuk. A kamera paraméterek viszont a nézeti transzformációra vannak hatással, ami az objektumot a világ-koordinátarendszereből a képernyő-koordinátarendszerbe viszi át. A transzformációk 4×4 -es mátrixokkal reprezentálhatók. Legyen az o objektum időfüggő modellezési transzformációja $\mathbf{T}_{M,o}(t)$, az időfüggő nézeti transzformáció pedig $\mathbf{T}_V(t)$. A beépített órát használó animációs program vázlata:

```
Óra inicializálás(  $t_{\text{start}}$  )  
do  
     $t = \text{Óra lekérdezés}$   
    for minden egyes  $o$  objektumra do  $\mathbf{T}_{M,o} = \mathbf{T}_{M,o}(t)$   
     $\mathbf{T}_V = \mathbf{T}_V(t)$   
    Képszintézis  
while  $t < t_{\text{end}}$ 
```

A felhasználó akkor érzékeli a képsorozatot folyamatos mozgásként, ha másodpercenként legalább 15 képet vetítünk neki. Ha a számítógép képes ilyen sebességgel el-

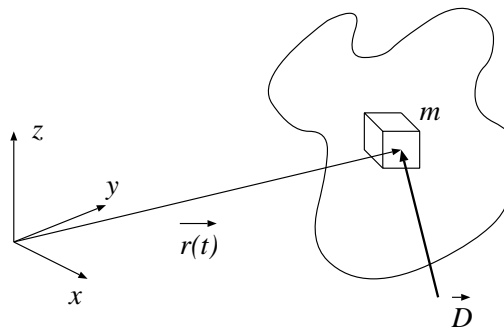
végezni a képszintézis lépéseit, *valós idejű animációról* beszélünk. Ha viszont a számítógépünk nem ilyen gyors, akkor az animáció két fázisban készülhet. Az elsőben kiszámítjuk és a háttértárra mentjük a képeket, majd a második fázisban a háttértárról beolvasva a mozgáshoz szükséges sebességgel visszajátsszuk őket. Amennyiben az első fázisban a képeket analóg jelként video szalagra írjuk, az animáció számítógép nélkül egy videomagnóval is lejátszható. A nem valós idejű animáció általános programja:

```

t = t_start
do
    for minden egyes o objektumra do  $\mathbf{T}_{M,o} = \mathbf{T}_{M,o}(t)$ 
         $\mathbf{T}_V = \mathbf{T}_V(t)$ 
        Képszintézis
        Képtárolás
        t +=  $\Delta t$ 
while t < t_end
// képrögzítés

Óra inicializálás( t_start )
do
    t = Óra lekérdezés
    Következő kép betöltése
    t +=  $\Delta t$ 
    while (t > Óra lekérdezés) Várj
while t < t_end
// animáció: visszajátszás

```



16.1. ábra. Egy m tömegű pont dinamikája

Az animáció célja *valószerű mozgás* létrehozása. A mozgás akkor valószerű, ha kielégíti a fizikai törvényeket, ugyanis mindennapjaink során ilyen mozgásokkal találkozunk (a természet általában jól tudja a fizikát, és be is tartja a törvényeit). A mozgás

alapvető törvénye a *Newton-törvény*, amely szerint a testre ható erő arányos a mozgásvektor második deriváltjával.

A mozgásvektort a pont \vec{r}_L lokális koordinátáiból a modellezési transzformáció fejezi ki

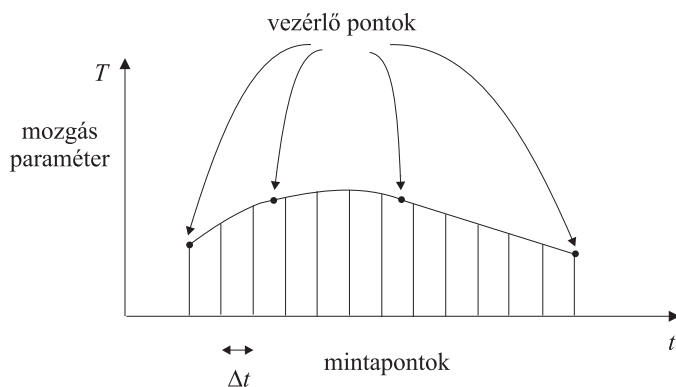
$$\vec{r}(t) = \vec{r}_L \cdot \mathbf{T}_M(t), \quad (16.1)$$

így \vec{D} erőt elszenvedő, m tömegű test pályája:

$$\frac{\vec{D}}{m} = \frac{d^2\vec{r}(t)}{dt^2} = \vec{r}_L \cdot \frac{d^2\mathbf{T}_M(t)}{dt^2}. \quad (16.2)$$

Mivel az erők valamilyen rugalmas mechanizmuson keresztül hatnak, nem változhatnak ugrásszerűen, következésképpen a mozgásvektor C^2 folytonos (3.3.3. fejezet).

Az animáció központi feladata olyan $\mathbf{T}_M(t)$ és $\mathbf{T}_V(t)$ mátrixok definiálása, amely egyrészt a felhasználó által elképzelt mozgást adja vissza, másrészt kielégíti a valószerű mozgás követelményeit. A feladat megoldása a szabadformájú görbénél megismert módszerekkel lehetséges. A felhasználó a mozgás során bejárt pozíciókat és orientációkat csak néhány vezérlőpontban definiálja, amiből a program a többi pillanat mozgásparamétereit interpolációs vagy approximációs technikákkal határozza meg.



16.2. ábra. Mozgástervezés interpolációval

16.1. Pozíció-orientáció mátrixok interpolációja

Mint azt a 5.2. fejezetben láttuk, tetszőleges pozíció, illetve orientáció megadható a következő mátrixszal:

$$\begin{bmatrix} 0 \\ \mathbf{A}_{3 \times 3} & 0 \\ 0 \\ \vec{q} & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ q_x & q_y & q_z & 1 \end{bmatrix}. \quad (16.3)$$

A \vec{q} vektor a pozíciót, az $\mathbf{A}_{3 \times 3}$ pedig az orientációt határozza meg. A \vec{q} vektor elemeit egymástól függetlenül vezérelhetjük, az a_{11}, \dots, a_{33} elemeket viszont nem, hiszen azok összefüggnek egymással. 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 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. A megoldást a független orientációs paraméterek terében végrehajtott interpoláció jelenti. Például használhatjuk az orientáció jellemzésére a *roll/pitch/yaw* szögeket, amelyek egy orientációhoz úgy visznek el, hogy először a z tengely körül α szöggel, majd y tengely körül β szöggel, végül az x tengely körül γ -szöggel forgatnak. Összefoglalva a mozgás függetlenül vezérelhető paraméterei:

$$\mathbf{p}(t) = [x(t), y(t), z(t), \alpha(t), \beta(t), \gamma(t)]. \quad (16.4)$$

A képszintézis során a modellezési transzformációra van szükségünk, amit az interpolált *paraméter vektorból* számíthatunk ki:

$$\mathbf{A} = \begin{bmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \beta & 0 & -\sin \beta \\ 0 & 1 & 0 \\ \sin \beta & 0 & \cos \beta \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \gamma & \sin \gamma \\ 0 & -\sin \gamma & \cos \gamma \end{bmatrix}, \quad (16.5)$$

$$\vec{q} = [x, y, z]. \quad (16.6)$$

A *valószerű mozgás* biztosításához az \mathbf{A} mátrix és a \vec{q} vektor elemeinek C^2 folytonos görbéknek kell lenniük. Ezt a paraméter vektor elemeinek C^2 folytonos interpolációjával vagy approximációjával teljesíthetjük. Kézenfekvő lehetőség például a *B-spline* görbék alkalmazása.

16.2. A kameraparaméterek interpolációja

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. $v\vec{r}p$: a nézeti referencia pont, amely az ablak középpontja,
2. $v\vec{p}n$: ablak normálvektora,
3. $v\vec{u}p$: az ablak függőleges iránya,
4. w_h, w_w : az ablak vízszintes és függőleges méretei,
5. $e\vec{y}e$: a szempozíció,
6. fp, bp : az első és hátsó vágósíkok.

Ezen paraméterek egymástól függetlenül vezérelhetők, így a kamera paraméter vektora:

$$\mathbf{p}_{\text{cam}}(t) = [v\vec{r}p, v\vec{p}n, v\vec{u}p, w_h, w_w, e\vec{y}e, fp, bp]. \quad (16.7)$$

Egy t időpillanatra a paraméter vektor aktuális értékéből számíthatjuk a \mathbf{T}_V nézeti transzformációs mátrixot.

16.3. Mozgás tervezés

A *mozgás tervezés* a vezérlőpontok felvételével kezdődik. A felhasználó felvesz egy $t_1, t_2 \dots t_n$ időpont sorozatot és elhelyezi az objektumokat és a kamerát ezen időpontokban. Az elhelyezés történhet a transzformációs mátrix interaktív vezérlésével, vagy közvetlenül a paraméter vektor megadásával. Az első esetben a paraméter vektort a program számítja ki a transzformációs mátrixból. A t_i időpillanatban beállított elrendezés az egyes objektumok paramétereire egy $\mathbf{p}_o(t_i)$ vezérlőpontot határoz meg. Ezen vezérlőpontokat felhasználva a program minden objektum minden paraméterére egy C^2 folytonos görbét illeszt (például B-spline-t).

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:

```

Vezérlőpontok definiálása:  $t_1, \dots, t_n$  // tervezés
for minden egyes  $k$  vezérlőpontra do
  for minden egyes  $o$  objektumra do
     $o$  objektum elrendezése:  $\mathbf{p}_o(t_k) = [x(t_k), y(t_k), z(t_k), \alpha(t_k), \beta(t_k), \gamma(t_k)]_o$ 

```

```

    endfor
    Kamera beállítás:  $\mathbf{p}_{\text{cam}}(t_k)$ 
endfor
for minden egyes  $o$  objektumra do
    Interpolálj egy  $C^2$  függvényt:  $\mathbf{p}_o(t) = [x(t), y(t), z(t), \alpha(t), \beta(t), \gamma(t)]_o$ 
endfor
Interpolálj egy  $C^2$  függvényt a kameraparaméterekhez:  $\mathbf{p}_{\text{cam}}(t)$ 

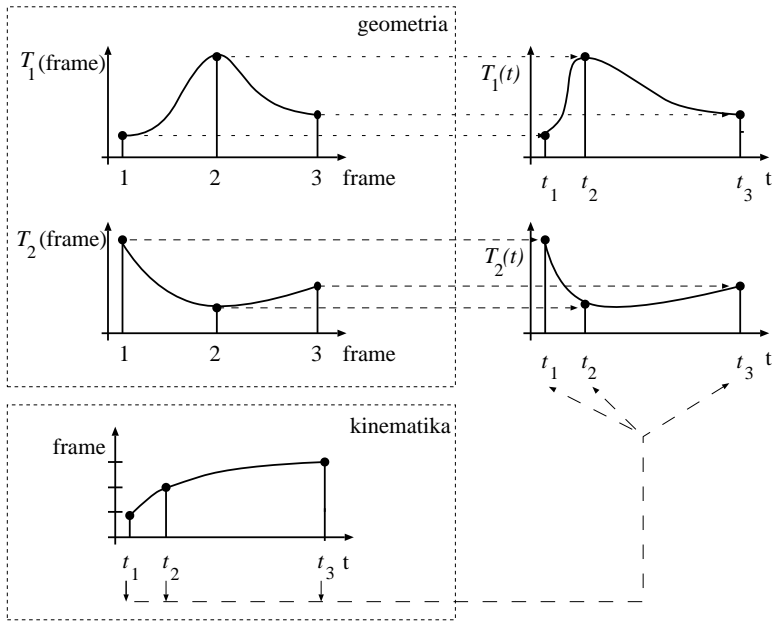
Óra inicializálás( $t_{\text{start}}$ ) // animáció
do
     $t = \text{Óra leolvasás}$ 
    for minden egyes  $o$  objektumra do
        mintavételezés  $t$ -ben:  $\mathbf{p}_o = [x(t), y(t), z(t), \alpha(t), \beta(t), \gamma(t)]_o$ 
         $\mathbf{T}_{M,o} = \mathbf{T}_{M,o}(\mathbf{p}_o)$ 
    endfor
    A kamerához mintavételezés  $t$ -ben:  $\mathbf{p}_{\text{cam}} = \mathbf{p}_{\text{cam}}(t)$ 
     $\mathbf{T}_V = \mathbf{T}_V(\mathbf{p}_{\text{cam}})$ 
    Képszintézis
while  $t < t_{\text{end}}$ 

```

Ennek a megközelítésnek több hátránya is van. Tegyük fel például, hogy az animáció megtekintése után úgy találjuk, hogy a film egy része túl lassú, ezért fel kívánjuk gyorsítani ezt a részt. Az egyetlen dolog amit tehetünk, hogy újból kezdjük a tervezést, ami nyilván meglehetősen keserves. A probléma abból származik, hogy a mozgástervezési eljárás nem választja szét az időzítési információkat a pályák geometriai megfogalmazásától.

A megoldást a színészi és rendezői álmokat dédelgetők jól ismerik. Egy színházi előadás színpadra állítása ugyanis nagyon hasonló az animáció tervezéséhez. Ha az előadást a fenti algoritmusnak megfelelően rendeznénk, az azt jelentené, hogy minden színésznek tudnia kellene a színpadra lépésének pontos idejét. Nem nehéz elképzelni, hogy ha az előadás késik, akkor milyen zűrzavart okozna, hogy minden egyes színész menetrendjét megfelelően átprogramozzuk. Szerencsére a színházak nem így működnek, hanem ehelyett a színészek azt tudják, hogy nekik melyik színben kell megjelenniük. Az előadás alatt csak a színházi ügyelő követi az előadást, és közli a színészekkel, hogy melyik szín következik. Tehát az időzítést (színházi ügyelő) és a mozgást (színészek mely színben jelennek meg) sikeresen szétválasztottuk.

Használjuk ugyanezt az eljárást! Az animációs szekvenciát *keretekre* (*frame*) osztjuk, és a mozgást a keretek függvényében specifikáljuk. Az interpoláció vezérlőpontjaihoz, az ún. *kulcskeretekhez* (*keyframe*) célszerűen az első, második, harmadik, stb. számot rendeljük. Majd a geometriától függetlenül megmondjuk, hogy a kereteknek mely időpillanatokban kell bekövetkezniük. Az időzítés megadása során a keretekhez



16.3. ábra. Animáció keretek felhasználásával

időfüggvényt rendelünk. Az animációs fázisban először az aktuális időhöz a keret számát adjuk meg, majd a keret ismeretében a mozgás paramétereit, ezekből pedig a transzformációs mátrixokat.

// Geometriai tervezés

```

for minden egyes  $k_f$  kulcskeretre do
  for minden egyes  $o$  objektumra do
     $o$  objektum elrendezése:  $\mathbf{p}_o(k_f) = [x(k_f), y(k_f), z(k_f), \alpha(k_f), \beta(k_f), \gamma(k_f)]_o$ 
  endfor
  Kamera beállítás:  $\mathbf{p}_{\text{cam}}(k_f)$ ;
endfor
for minden egyes  $o$  objektumra do
  Interpolálj egy  $C^2$  függvényt:  $\mathbf{p}_o(f) = [x(f), y(f), z(f), \alpha(f), \beta(f), \gamma(f)]_o$ 
endfor
Interpolálj egy  $C^2$  függvényt a kameraparaméterekhez:  $\mathbf{p}_{\text{cam}}(f)$ 

```

// Kinematikai tervezés

```

for minden egyes  $k_f$  kulcskeretre do
  Add meg azt a  $t_{k_f}$ -t, amelyre  $\mathcal{F}(t_{k_f}) = k_f$ 
  Interpolálj egy  $C^2$  függvényt:  $\mathcal{F}(t)$ 
endfor

```

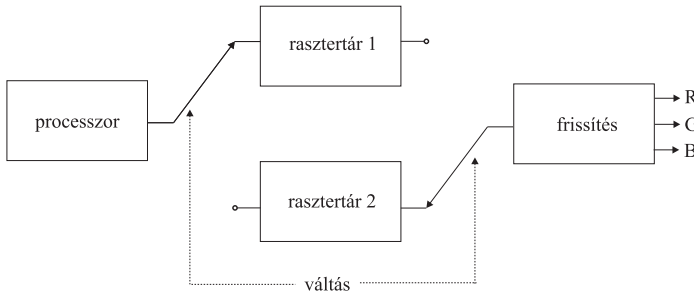
```

// Animáció
Óra inicializálás( $t_{\text{start}}$ )
do
   $t = \text{Óra leolvasás}$ 
   $f = \mathcal{F}(t)$ ;
  Minden  $o$  objektumra
    mintavételezés  $f$ -ben:  $\mathbf{p}_o = [x(f), y(f), z(f), \alpha(f), \beta(f), \gamma(f)]_o$ 
     $\mathbf{T}_{M,o} = \mathbf{T}_{M,o}(\mathbf{p}_o)$ 
  endfor
  A kamerához mintavételezés  $f$ -ben:  $\mathbf{p}_{\text{cam}}(f)$ 
   $\mathbf{T}_V = \mathbf{T}_V(\mathbf{p}_{\text{cam}})$ 
  Képszintézis
while  $t < t_{\text{end}}$ 

```

16.4. Dupla puffrelés

Az animáció alatt a képeket gyorsan egymás után generáljuk és kihasználjuk, hogy a gyorsan levetített állóképsorozat a szem mozgásként érzékeli.



16.4. ábra. Dupla puffer rendszerek

Valós idejű megjelenítés esetén a használt takarási algoritmus függvényében a számított kép fokozatosan alakul ki a számítógép képernyőjén, amely alatt rövid időre olyan poligonok is feltűnhetnek, amelyek egyáltalán nem látszhatnak. Ez észrevehető villogáshoz vezet. A klasszikus mozgófilmek világában hasonló probléma kiküszöbölésére a vetítőt letakarjuk, mialatt az egyik képkockáról a másikra lépünk át. Ugyanezt az elvet itt is használhatjuk. Ehhez két rasztetár szükséges. Egy adott pillanatban az egyiket megjelenítjük, a másikba pedig rajzolunk. Amikor a kép elkészült, az elektronsugar képvisszafutási ideje alatt a két rasztetár szerepet cserél.

17. fejezet

Térfogat modellek és térfogatvizualizáció

Egy *térfogat modell* (*volumetric model*) úgy képzelhető el, hogy a 3D tér egyes pontjaiban sűrűségértékeket adunk meg. A feladat tehát egy $v(x, y, z)$ függvény reprezentálása. A gyakorlatban általában térfogatmodellekre vezetnek a mérnöki számítások (pl. egy elektromágneses térben a potenciáeloszlás). 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.

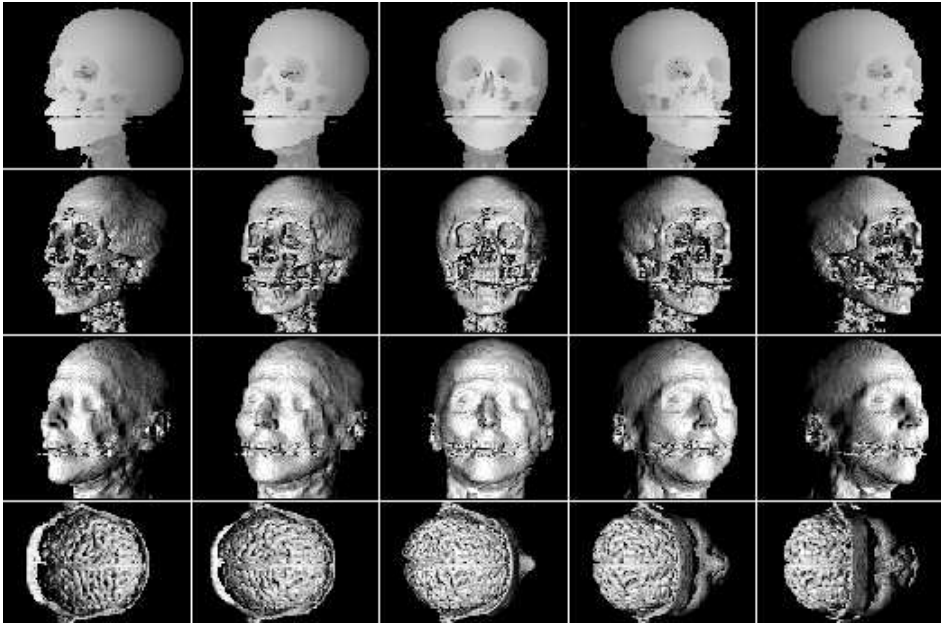
A térfogati modellt általában szabályos ráccsal mintavételezzük, és az értékeket egy 3D mátrixban tároljuk. Úgy is tekinthetjük, hogy egy mintavételi érték a térfogat egy kicsiny kockájában érvényes függvényértéket képviseli. Ezen elemi kockákat térfogat-elemnek, vagy *voxel*nek nevezzük.

Nyilván az elemi kockák direkt felrajzolásának nem sok értelme lenne, hiszen ekkor csak a térfogat külső oldalain lévő értékeket láthatjuk (egy páciens fejéről azért csinálunk CT-t, hogy belsejét vizsgálhassuk, nem pedig azért, hogy az arcában gyönyörködjünk). A teljes térfogat áttekintéséhez bele kell látnunk a térfogatba, tehát a térfogati modellt célszerű úgy kezelni, mintha az részben átlátszó anyagból állna. A térfogatot alkotó “ködöt” két alapvetően különböző módon jeleníthetjük meg. A *direkt módszerek* közvetlenül a térfogati modellt fényképezik le, az *indirekt módszerek* pedig először átalakítják egy másik modellre, amelyet azután a szokásos módszerekkel jeleníthetünk meg.

17.1. Direkt térfogatvizualizációs módszerek

A direkt módszerek a ködöt a képsíkra vetítik és számba veszik az egyes pixeleknek megfelelő vetítősugarak által metszett voxeleket. A pixelek színét a megfelelő voxelek színéből és átlátszóságából határozzák meg.

Elevenítsük fel a radianciát a fényelnyelő anyagokban leíró 8.34. egyenletet! Feltehetjük, hogy az anyag nem bocsát ki fényt magából. Ekkor egy sugár mentén a radiancia



17.1. ábra. CT és MRI mérések vizualizációja

a fényelnyelés miatt csökken, a sugár irányába ható visszaverődés miatt viszont nő:

$$\frac{dL(s, \omega)}{ds} = -\kappa_t(s) \cdot L(s, \omega) + L_{\text{is}}(s). \quad (17.1)$$

Amennyiben $L_{\text{is}}(s)$ ismert, az egyenlet megoldása:

$$L(s, \omega) = \int_s^T e^{-\int_s^\tau \kappa_t(p) dp} \cdot L_{\text{is}}(\tau, \omega) d\tau, \quad (17.2)$$

ahol T a maximális sugárparaméter.

Az integrálok kiértékeléséhez a $[0, T]$ sugárparaméter tartományt kis intervallumokra osztjuk és az integrált téglányszabállyal becsüljük. Ez megfelel a folytonos differenciálegyenletet véges differenciaegyenlettel történő közelítésének:

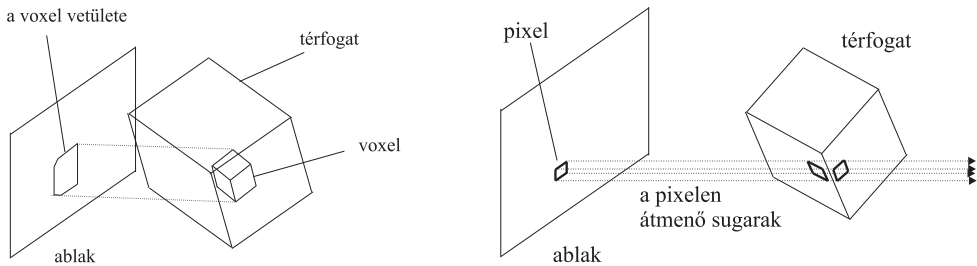
$$\frac{\Delta L(s, \omega)}{\Delta s} = -\kappa_t(s) \cdot L(s, \omega) + L_{\text{is}}(s) \quad \implies$$

$$L(s + \Delta s, \omega) = L(s, \omega) - \kappa_t(s) \cdot \Delta s \cdot L(s, \omega) + L_{\text{is}}(s) \cdot \Delta s. \quad (17.3)$$

Jelöljük a $L_{\text{is}}(s) \cdot \Delta s$ szórt radianciát $C(s)$ -sel, amit ezek után a voxel saját színének tekintünk. Az $\alpha(s) = \kappa_t \cdot \Delta s$ érték — amelyet *opacitásnak* nevezünk — a két minta közötti fényelnyelést jellemzi. Ezekkel az új jelölésekkel:

$$L(s + \Delta s, \omega) = (1 - \alpha(s)) \cdot L(s, \omega) + C(s). \quad (17.4)$$

Ezt az egyenletet a térfogat vizualizálásakor a pixelekhez tartozó sugarakra kell megoldani. A sugarak definiálása során kiindulhatunk a térfogattól, vagy a képernyő pixeleiből egyaránt. Az első módszer neve *térfogat vetítés* [DCH88] a másodiké pedig *térfogati sugárkövetés* [Lev88, Lev90]. A térfogat vetítés az inkrementális képszintézishez, azon belül pedig a festő algoritmushoz hasonlít, a térfogati sugárkövetés pedig a normál sugárkövetés adaptációja.



17.2. ábra. *Térfogat vetítés (bal) és térfogati sugárkövetés (jobb)*

17.1.1. Térfogat vetítés

A térfogat vetítés a térfogattól indul, és a térfogati adathalmazt az ablak síkjával párhuzamos, Δs távolságra lévő síkok mentén mintavételezi, majd az egyes síkokat az ablakra vetíti. A feldolgozást a szemtől távolabbi síkokkal kezdjük és onnan közeledünk a szempozíció felé. A feldolgozás adott pillanatában érvényes $L(s, \omega)$ akkumulálódó radianciát a pixeleken tároljuk. Így egy sík vetítése során a 17.4. egyenletet az egyes pixeleken tárolt $L(s)$ érték és a $C(s)$ vetített érték felhasználásával számítjuk ki. Ha az összes síkon végigmentünk, a megjelenítendő színeket a pixeleken akkumulálódott radiancia határozza meg.

17.1.2. Térfogati sugárkövetés

A térfogati sugárkövetés a pixelektől indul. A pixel középpontokon keresztül egy-egy sugarat indítunk a térfogatba, és a sugár mentén haladva oldjuk meg a 17.4 egyenletet.

Amennyiben a fényvel megegyező irányban haladunk, a 17.4 egyenletet az eredeti formájában értékeljük ki. Sajnos ez a módszer gyakran felesleges számításokat igényel, hiszen a számításokat a legtávolabbi voxeleknél kezdjük, amelyek az esetek döntő részében úgyszemint látszanak a képen. Ezért célszerűbb, ha a fényvel szemben haladunk a sugárintegrál kiértékelésekor. Természetesen ekkor a 17.4 egyenletet ki kell facsarni.

Tegyük fel, hogy a szemből már az s paraméterig jutottunk, és idáig úgy találtuk, hogy a sugár mentén az s paraméter és a szem között $L^*(s, \omega)$ radiancia akkumulálódott, és az integrált opacitás, amivel a s utánról érkező radianciát kell szorozni pedig $\alpha^*(s)$. Ha a sugárparamétert Δs -sel léptetjük, akkor a következő inkrementális összefüggések érvényesek:

$$L^*(s - \Delta s, \omega) = L^*(s, \omega) + (1 - \alpha^*(s)) \cdot C(s), \quad (17.5)$$

$$1 - \alpha^*(s - \Delta s) = (1 - \alpha(s)) \cdot (1 - \alpha^*(s)). \quad (17.6)$$

Most a pixelek színét az $L^*(0)$ érték határozza meg. Ha az összefüggések inkrementális kiértékelése során úgy találjuk, hogy az $1 - \alpha^*(s - \Delta s)$ mennyiség egy küszöb érték alá került, befejezhetjük a sugár követését, mert a hátrébb levő voxelek hatása elhanyagolható.

17.2. A voxel szín és az opacitás származtatása

A térfogatvizualizációs algoritmusok a voxelek saját színével és opacitásértékeivel dolgoznak, amelyeket a $v(x, y, z)$ mért voxelértékekből származtathatunk. A gyakorlatban többféle módszer terjedt el, amelyek különböző megjelenítésekhez vezetnek.

Az egyik leggyakrabban használt eljárás a felületi modellek árnyalását adaptálja a térfogatra (17.8. ábra). Az árnyalási paraméterek származtatáshoz osszuk fel a mért $v(x, y, z)$ sűrűségfüggvény értékészletét adott számú intervallumra (például, a CT és az MRI képeknél a levegő, lágyszövet és a csont sűrűségértékeit érdemes elkülöníteni). Az egyes intervallumokhoz diffúz és spekuláris visszaverődési tényezőt, valamint átátlátszóságot adunk meg. Absztrakt fényforrások jelenlétét feltételezve, például a Phong illuminációs képlet segítségével meghatározzuk a voxelhez rendelhető színt. Az illuminációs képletek által igényelt normálvektort a $v(x, y, z)$ gradienséből kaphatjuk meg.

Ha a 3D voxelek mérete a, b és c , akkor a gradiens vektor egy x, y, z pontban:

$$\text{grad } v = \begin{pmatrix} \frac{v(x + a, y, z) - v(x - a, y, z)}{2a} \\ \frac{v(x, y + b, z) - v(x, y - b, z)}{2b} \\ \frac{v(x, y, z + c) - v(x, y, z - c)}{2c} \end{pmatrix}. \quad (17.7)$$

Teljesen homogén tartományokban a gradiens zérus, tehát a normálvektor definiálatlan, ami a képen zavarokat okozhat. Ezeket eltüntethetjük, ha a voxelek opacitását a megadott átlátszóság és a gradiens abszolút értékének a szorzataként számítjuk, hiszen ekkor a homogén tartományok teljesen átlátszóak lesznek.

Egy másik módszer azt feltételezi, hogy a megvilágítás a térfogat túlsó oldaláról jön. Ha az opacitást a $v(x, y, z)$ -vel arányosan választjuk, a pixelek színe arányos lesz az integrált opacitással, azaz a $v(x, y, z)$ sugármenti integráljával. Mivel a röntgensugarak is hasonlóan nyelődnek el, a kép hatásában a *röntgen képek*re emlékeztet.

Végül egy gyorsan kiértékelhető, és ugyanakkor az orvosi diagnosztikában rendkívül hasznos megjelenítési eljáráshoz jutunk, ha minden sugár mentén az $v(x, y, z)$ maximumával arányosan színezzük ki a pixeleket (1.6. ábra jobb oldala).

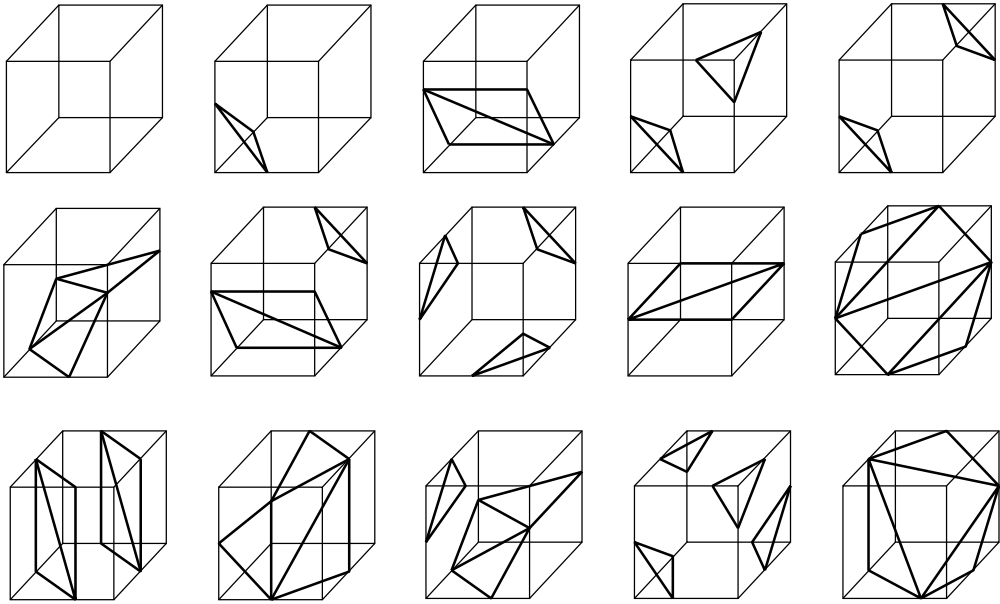
17.3. Indirekt térfogatvizualizációs módszerek

Az indirekt módszerek a térfogati modellt egy másfajta modellre alakítják át, majd azt fényképezik le.

17.3.1. Masírozó kockák algoritmus

A legkézenfekvőbb közbenső reprezentáció a felületi modell, hiszen a felületi modellek megjelenítése a számítógépes grafika leginkább kimunkált területe. Egy térfogati modellből elvileg úgy nyerhetünk felületeket, hogy azonosítjuk a 3D térfogat *szintfelületeit*, azaz azon 2D ponthalmazokat, ahol a $v(x, y, z)$ megegyezik a megadott szintértékkel. Ez korántsem olyan könnyű, mint ahogyan első pillanatban látszik, hiszen mi a $v(x, y, z)$ függvényt csak diszkrét értékekben ismerjük, a közbenső pontokat a tárolt adatok interpolációjával kell előállítani.

Egy ilyen, az interpolációt és a szintfelületet megkeresését párhuzamosan elvégző módszer a *masírozó kockák algoritmus* (*marching cubes algorithm*). Az algoritmus első lépésben a szintfelület értékének és a voxelek értékének összehasonlításával minden voxelre eldönti, hogy az belső voxel-e avagy külső voxel. Ha két szomszédos voxel eltérő típusú, akkor közöttük határnak kell lennie. A határ pontos helye a voxelek élein az érték alapján végzett lineáris interpolációval határozható meg. Végül az éleken kijelölt pontokra háromszögeket illesztünk, amelyekből összeáll a szintfelület. A háromszög illesztéshez figyelembe kell venni, hogy egy zárt alakzat az egy pontra illeszkedő 8 voxel összesen 256-féleképpen metszheti, amiből végül 14 ekvivalens eset különíthető el (17.3. ábra). A metszéspontokból a háromszögekhez úgy juthatunk el, hogy először azonosítjuk a megfelelő esetet, majd eszerint definiáljuk a háromszögeket.



17.3. ábra. Egy zárt alakzat az egy pontra illeszkedő 8 voxel összesen 14 féleképpen metszheti

17.3.2. Fourier-tér módszerek

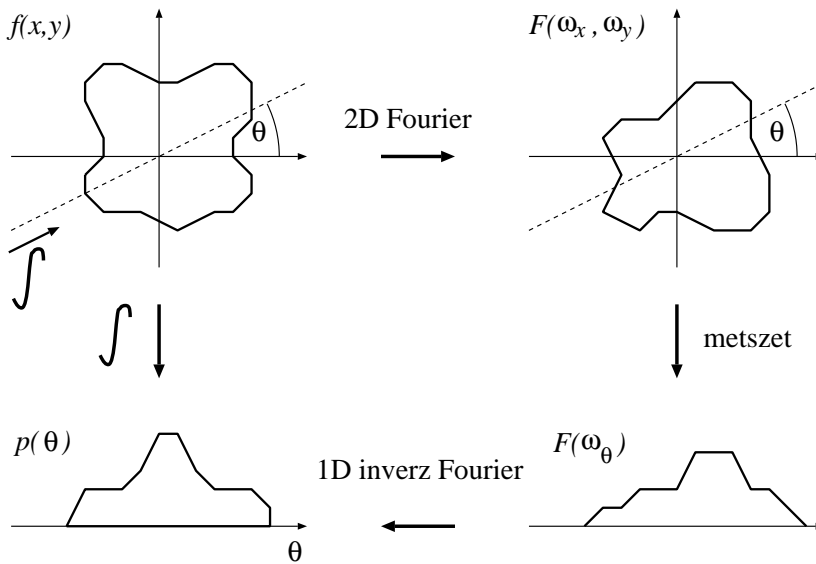
A Fourier-transzformációnak több hasznos tulajdonsága van: egyrészt a gyors Fourier-transzformációs módszerrel hatékonyan elvégezhető akár magasabb dimenziókban is, másrészt a transzformált értéke a 0 frekvencián megegyezik a függvény integráljával. Mivel a röntgenszerű megjelenítéshez a $v(x, y, z)$ függvényt a különböző sugarak mentén kell integrálni, a megjelenítéshez az adathalmaz Fourier-transzformáltja vonzóbbnak tűnik mint maga a mért adat.

A $V(\omega_x, \omega_y, \omega_z)$ Fourier-transzformált és a $v(x, y, z)$ eredeti adat között a következő összefüggés áll fenn:

$$V(\omega_x, \omega_y, \omega_z) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} v(x, y, z) \cdot e^{-2\pi j(x\omega_x + y\omega_y + z\omega_z)} dx dy dz, \quad (17.8)$$

ahol $j = \sqrt{-1}$.

Vegyük észre, hogy a $V(\omega_x, \omega_y, 0)$ metszet (slice) éppen a z irányú sugarakkal számított kép 2D Fourier-transzformáltja. Hasonlóan a $V(\omega_x, 0, \omega_z)$ az y irányú sugarakkal, a $V(0, \omega_y, \omega_z)$ pedig az x irányú sugarakkal vett integrálok Fourier-transzformáltja. Ráadásul — a Fourier-transzformáció sajátosságai miatt — általános orientáció-



17.4. ábra. Fourier-tér módszer

jú metszetek képzésével tetszőleges irányból látott kép Fourier-transzformáltja számítható. Ha az ablak orientációját az oldalakkal párhuzamos $W_u = (\omega_{ux}, \omega_{uy}, \omega_{uz})$ és $W_v = (\omega_{vx}, \omega_{vy}, \omega_{vz})$ vektorokkal definiáljuk, akkor a kép Fourier-transzformáltja az

$$P(\omega_u, \omega_v) = V(\omega_{ux}\omega_u + \omega_{vx}\omega_v, \omega_{uy}\omega_u + \omega_{vy}\omega_v, \omega_{uz}\omega_u + \omega_{vz}\omega_v) \quad (17.9)$$

összefüggéssel határozható meg. Ebből pedig egy u, v koordinátájú pixelnek megfelelő integrál értéke inverz Fourier-transzformációval számítható:

$$p(u, v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} P(\omega_u, \omega_v) \cdot e^{2\pi j(u\omega_u + v\omega_v)} d\omega_u d\omega_v. \quad (17.10)$$

Az eljárás a viszonylag számításigényes 3D Fourier-transzformáció egyszeri végrehajtása után, tetszőleges irányú megjelenítéshez már csak 2D inverz Fourier-transzformációkat alkalmaz, ezért interaktív körbejáráshoz vagy animációhoz javasolható.

17.4. Program: masírozó kockák algoritmus

A masírozó kockák algoritmus a voxel tömb által reprezentált függvény szintfelületét azonosítja, és azokra egy háromszögsorozatot illeszt. A szintfelület sokféleképpen metszhet egy voxel, amit aszerint osztályozhatunk, hogy melyik csúcspont van a keresett felületi érték felett illetve alatt. A voxel 8 csúcának osztályozása során kapott érték egy konfigurációt jelent. Ezek a konfigurációk egy 256 soros táblázatba gyűjthetők (`polytab`), amelynek soraiban a -1 -gyel lezárt sorozat számhármasai azonosítják az adott eset poligonizációjához szükséges háromszögeket [Pap98]. A táblázatból kiolvashatjuk, hogy melyik éleket kell metszenünk a poligonok csúcsainak meghatározásához. A csúcsokat lineáris interpolációval számoljuk ki. Az alábbiakban ezen táblázatból csupán egy részletet mutatunk be, a teljes tábla a CD-n megtalálható.

```
int polytab[256][32]=
{
{-1, }, // 0:00000000, f:0
{1, 0, 4, 0, 3, 0, -1, }, // 1:00000001, f:1
{1, 0, 1, 2, 1, 5, -1, }, // 2:00000010, f:1
{1, 5, 4, 0, 1, 2, 1, 2, 4, 0, 3, 0, -1, }, // 3:00000011, f:2
...
{1, 0, 1, 5, 1, 2, -1, }, // 253:11111101, f:1
{1, 0, 3, 0, 4, 0, -1, }, // 254:11111110, f:1
{-1, }, // 255:11111111, f:0
};
```

A következő tömbökben azt tartjuk nyilván, hogy a kocka egyes csúcsai a bal-alsó-hátsó sarokhoz képest milyen relatív x , y és z koordinátákkal rendelkeznek.

```
int x_relpos_tab[8]={0, 1, 1, 0, 0, 1, 1, 0}; // x-ben
int y_relpos_tab[8]={0, 0, 0, 0, 1, 1, 1, 1}; // y-ben
int z_relpos_tab[8]={0, 0, -1, -1, 0, 0, -1, -1}; // z-ben
```

A `Volume` osztály egy `size` felbontású térfogatmodellt testesít meg. Az osztály konstruktora fájlból beolvassa voxelértékeket, az `Interpolate` tagfüggvénye a voxel-kockák élein interpolálja a hely- és irányvektorokat. A `MarchingCube` pedig az ismert algoritmussal egyetlen voxelre megvizsgálja, hogy a szintfelület metszi-e azt, és előállítja a metsző szintfelület háromszöghálós közelítését. A teljes térfogat modell megjelenítéséhez a `MarchingCube` tagfüggvényt minden egyes voxelre meg kell hívni. A kapott háromszögeket a szokásos 3D csővezetéken végigvezetve jeleníthetjük meg.

```

//=====
class Volume {
//=====
    int size;
    BYTE *** grid;
public:
    Volume( char * filename );
    int GridSize( ) { return size; }
    BYTE V(int x, int y, int z) {
        if (x < 0 || y < 0 || z < 0 ||
            x >= size || y >= size || z >= size) return 0;
        return grid[x][y][z];
    }

    void Interpolate( Point3D& point1, Point3D& point2,
                    Point3D& norm1, Point3D& norm2,
                    double value1, double value2, double isolevel,
                    Point3D& point, Point3D& norm ) {
        double m = (isolevel - value1) / (value2 - value1);
        point = point1 + (point2 - point1) * m;
        norm = norm1 + (norm2 - norm1) * m;
    }
    TriangleList3D * MarchingCube(int x, int y, int z, double isolevel);
};

//-----
Volume :: Volume( char * filename ) {
//-----
    size = 0;
    FILE * file = fopen(filename, "rb");
    if (fscanf(file,"%d", &size) != 1) return;

    grid = new BYTE**[size];
    for(int x = 0; x < size; x++) {
        grid[x] = new BYTE*[size];
        for(int y = 0; y < size; y++) {
            grid[x][y] = new BYTE[size];
            for(int z = 0; z < size; z++)
                grid[x][y][z] = fgetc(file);
        }
    }
}

```

```

//-----
TriangleList3D * Volume :: MarchingCube( int x, int y, int z,
                                         double isolevel ) {
//-----
    BYTE cubeindex = 0;
    if (V(x,y,z) < isolevel) cubeindex|=1;
    if (V(x+1,y,z) < isolevel) cubeindex|=2;
    if (V(x+1,y,z-1) < isolevel) cubeindex|=4;
    if (V(x,y,z-1) < isolevel) cubeindex|=8;
    if (V(x,y+1,z) < isolevel) cubeindex|=16;
    if (V(x+1,y+1,z) < isolevel) cubeindex|=32;
    if (V(x+1,y+1,z-1) < isolevel) cubeindex|=64;
    if (V(x,y+1,z-1) < isolevel) cubeindex|=128;
    if ( cubeindex == 0 || cubeindex == 255 ) return NULL;

    TriangleList3D * tlist =
        new TriangleList3D(0 /* emisszio */, 0.1 /* ka */,
                          0.4 /* kd */, 0.2 /* ks */, 10 /* shine */);

    for(int j = 0, t = 0; polytable[cubeindex][j] != -1; j += 6) {
        Point3D p1, p2, point[3], n1, n2, norm[3];
        for( int j1 = 0; j1 < 6; j1 += 2 ) {
            int x1 = x + x_relpos_tab[ polytable[cubeindex][j+j1] ];
            int y1 = y + y_relpos_tab[ polytable[cubeindex][j+j1] ];
            int z1 = z + z_relpos_tab[ polytable[cubeindex][j+j1] ];
            Point3D point1( x1, y1, z1 );
            Point3D norm1( V(x1-1,y1,z1) - V(x1+1,y1,z1),
                          V(x1,y1-1,z1) - V(x1,y1+1,z1),
                          V(x1,y1,z1-1) - V(x1,y1,z1+1) );
            double value1 = V(x1,y1,z1);

            int x2 = x + x_relpos_tab[ polytable[cubeindex][j+j1+1] ];
            int y2 = y + y_relpos_tab[ polytable[cubeindex][j+j1+1] ];
            int z2 = z + z_relpos_tab[ polytable[cubeindex][j+j1+1] ];
            Point3D point2( x2, y2, z2 );
            Point3D norm2( V(x2-1,y2,z2) - V(x2+1,y2,z2),
                          V(x2,y2-1,z2) - V(x2,y2+1,z2),
                          V(x2,y2,z2-1) - V(x2,y2,z2+1) );
            double value2 = V(x2,y2,z2);

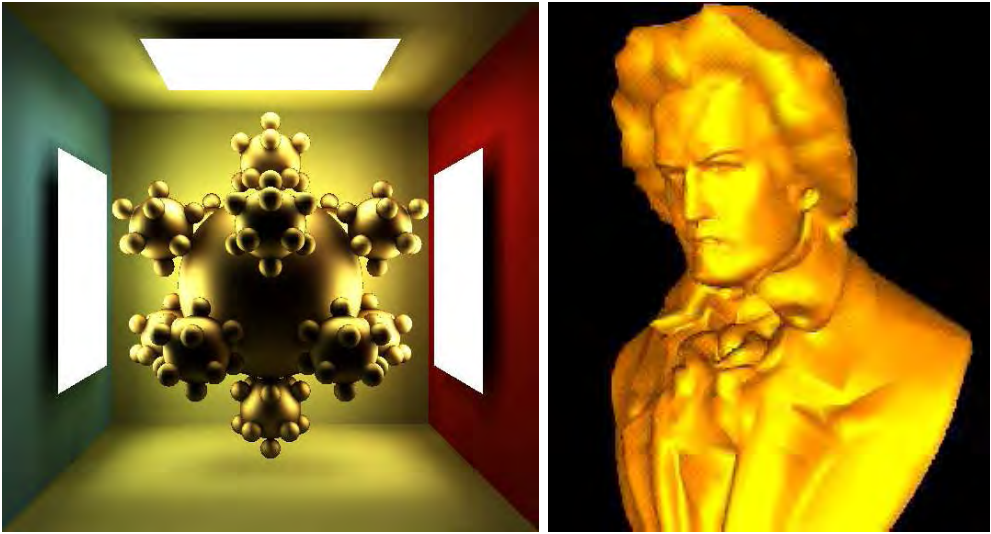
            Interpolate( point1, point2, norm1, norm2, value1, value2,
                          isolevel, point[j1/2], norm[j1/2] );
            norm[j1/2].Normalize( );
        }
        tlist -> AddTriangle( t++, point[0], point[1], point[2],
                              norm[0], norm[1], norm[2] );
    }
    return tlist;
}

```

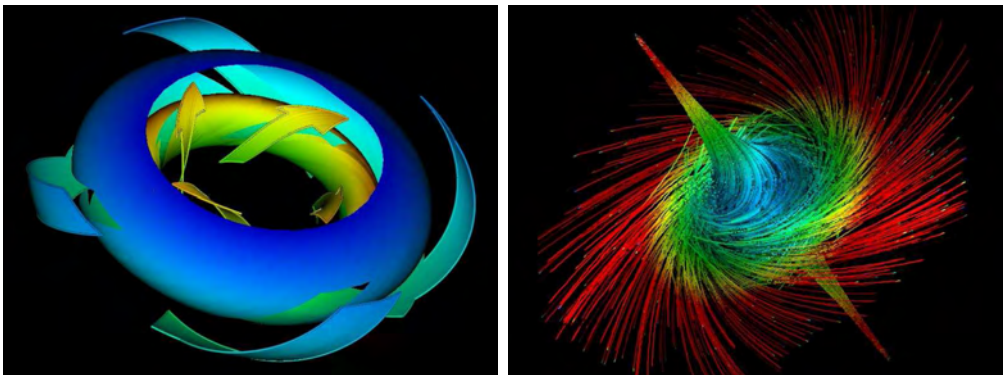

Színes képek



17.5. ábra. A CSG módszer ciklikus általánosításával készült modell. A hóembereket közönséges CSG modellek, a fenyőfát rekurzív CSG definiálja. A képet az ART program sugárkövetéssel készítette (Bécsi Műszaki Egyetem, Számítógépes Grafika Intézet).



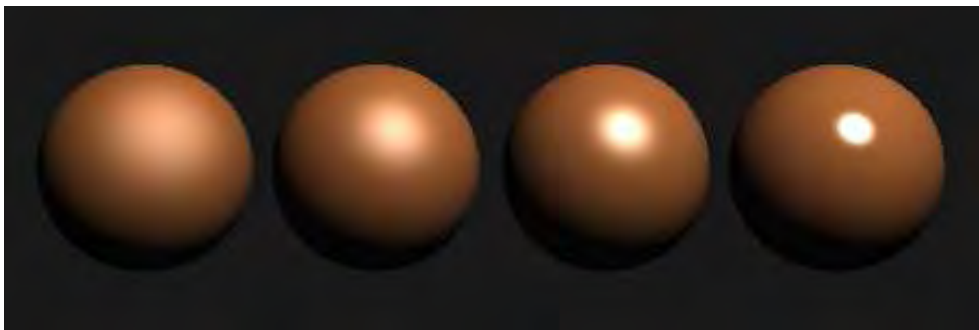
17.6. ábra. 3D világok fényképei: Bal: Globális illuminációs módszer, amely a fényvisszaverődéseket fizikailag pontosan szimulálja; Jobb: Lokális illuminációs algoritmus.



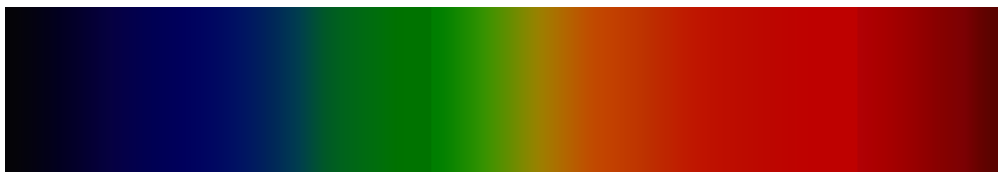
17.7. ábra. Dinamikus rendszerek viselkedésének megjelenítése [Löf98]



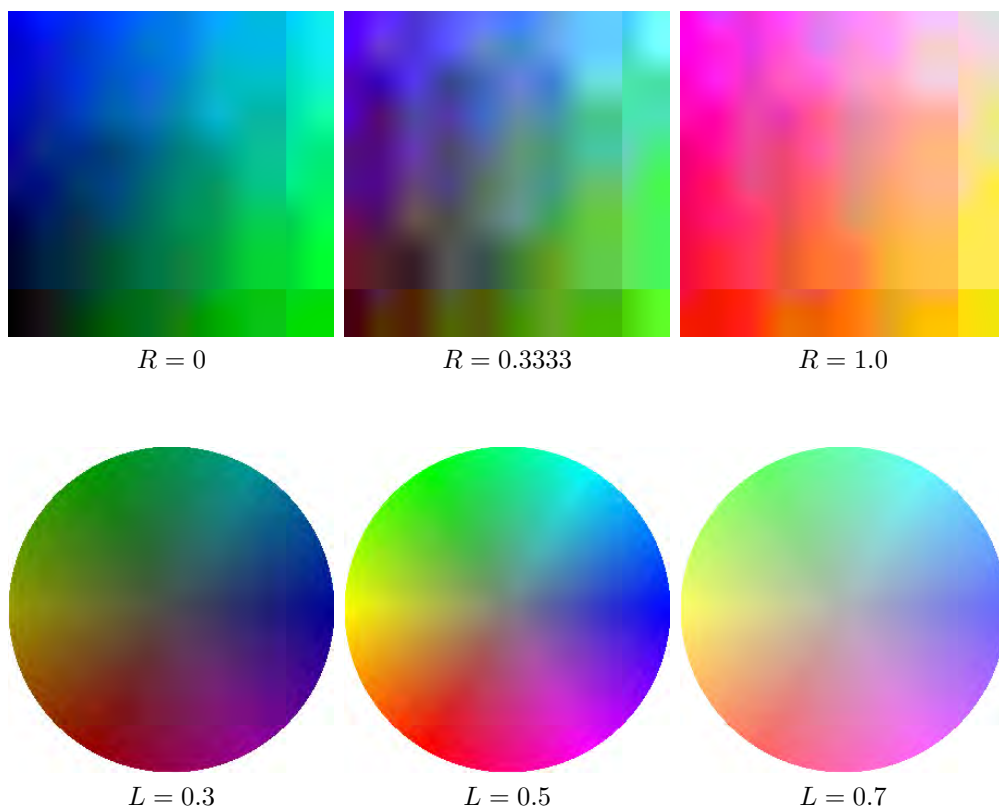
17.8. ábra. Számítógépes tomográf mérési eredményeinek vizualizációja [BSKG97].



17.9. ábra. Phong illuminációs modell $n = 2, 5, 10, 50$ értékekre.



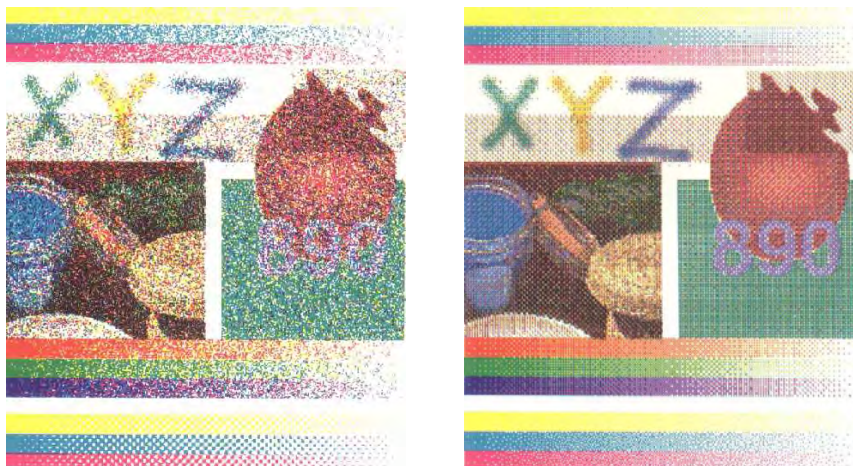
17.10. ábra. Monokromatikus fények keltette színérzetek a 390-690 nm tartományban. A szivárvány színei, amit prizmával is előállíthatunk.



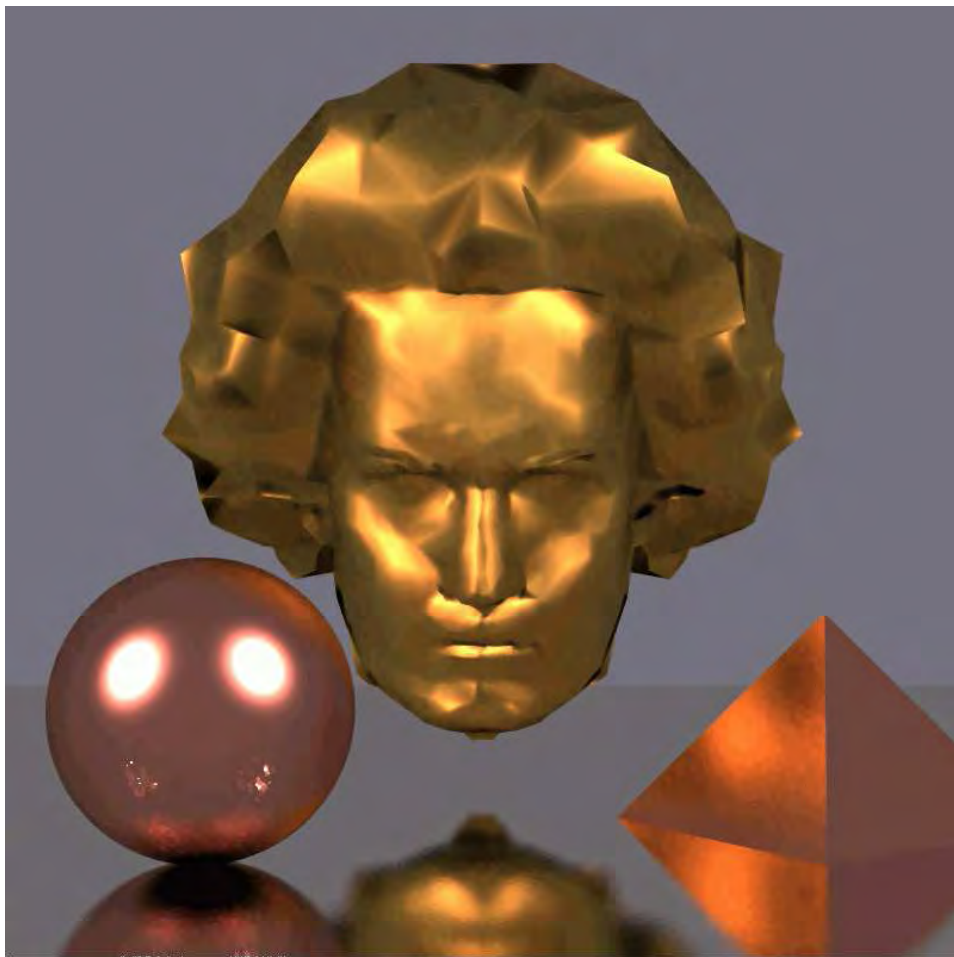
17.11. ábra. Felső sor: színek a színekockában: állandó R értékek mentén vett metszetek;
Alsó sor: színek a HLS színeképben: állandó L érték mellett vett metszetek.



17.12. ábra. Jenny 256, 16 és 8 színben. Kevés szín alkalmazása esetén az eredetileg folytonos színfüggvény a durva kvantálás miatt jól látható ugráshelyeken változik.



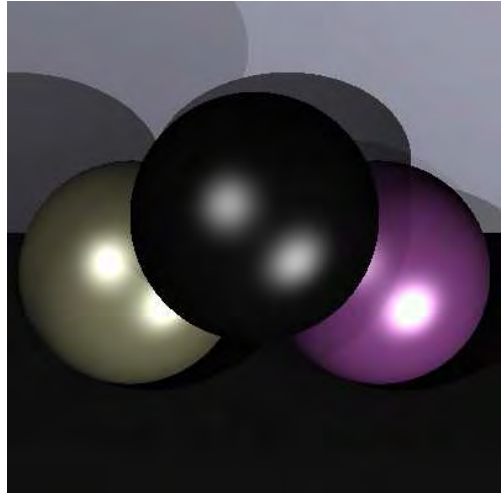
17.13. ábra. Véletlen és szabályos ditherek színes képen [PTG95]. A szem számára a véletlen ditherek kellemesebbek, mert nem ismeri fel a periodikus hatásokat.



17.14. ábra. Arany Beethoven, réz golyó és piramis egy ezüst tálcán. A fémes hatást a Fresnel-függvénynek a max Phong modell BRDF-jébe történő beépítésével értük el. A tálca ($n = 5000$) nem ideális tükrő. A kép Monte-Carlo sugárkövetéssel (inverz fényút követés) készült [NNSK98].



lokális illuminációs módszer



lokális illuminációs módszer árnyékszámítással

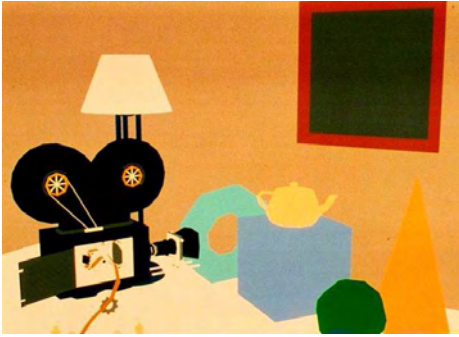


rekurzív sugárkövetés

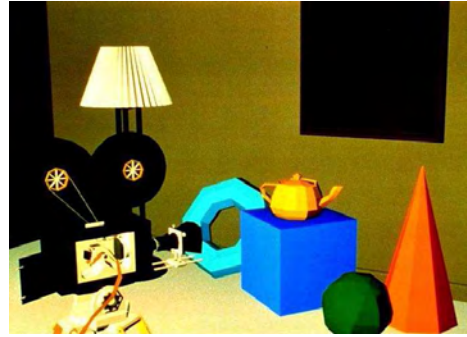


globális illuminációs módszer

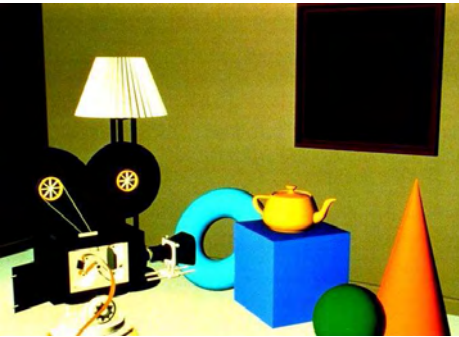
17.15. ábra. Lokális illuminációs módszer, sugárkövetés és globális illuminációs módszer összehasonlítása. A képek a CD-n levő példaprogrammal készültek. A futási idők rendre 90 sec, 95 sec, 135 sec, 9 óra



árnyalás saját színnel



konstans árnyalás



Gouraud-árnyalás diffúz felületekre



Gouraud-árnyalás spekuláris felületekre



Phong-árnyalás finoman tesszellált felületekre



Phong-árnyalás durván tesszellált felületekre

17.16. ábra. Képszintézis különböző árnyalási modellel (Pixar). Figyeljük meg, hogy a Gouraud-árnyalás spekuláris visszaverődésű felületekre csak igen finom tesszelláció mellett használható, míg a Phong-árnyalás mindig kielégítő eredményt ad.



Fresnel együtthatóval súlyozott BRDF



textúrák alkalmazása

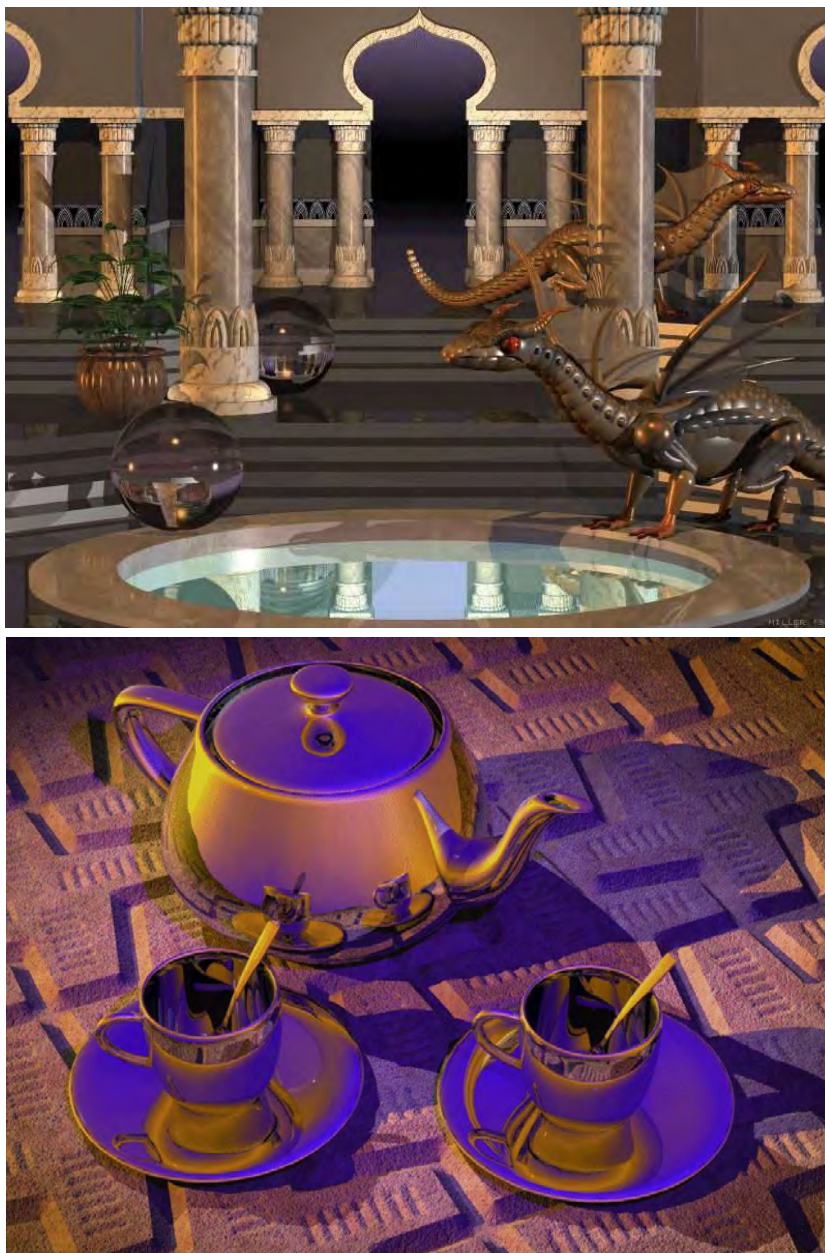


Bucka leképzés a tóruszon



visszaverődés leképzés a padlón

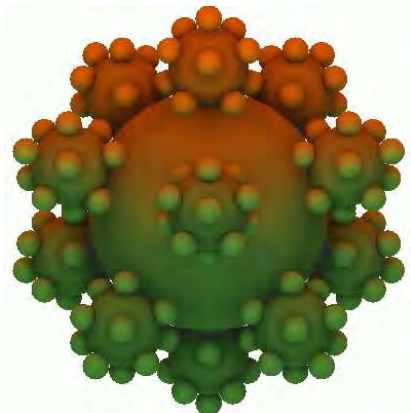
17.17. ábra. Bonyolult árnyalási modellek használata (Pixar).



17.18. ábra. Sugárkövetéssel készült képek (POVRAY program).



17.19. ábra. Fotorealisztikus képszintézis és a valóság összevetése. A felső kép az Aizu Egyetem előcsarnokáról készült fénykép, az alsó az előcsarnok modelljének Monte-Carlo sugárkövetéssel számított szintetikus képe (University of Aizu).



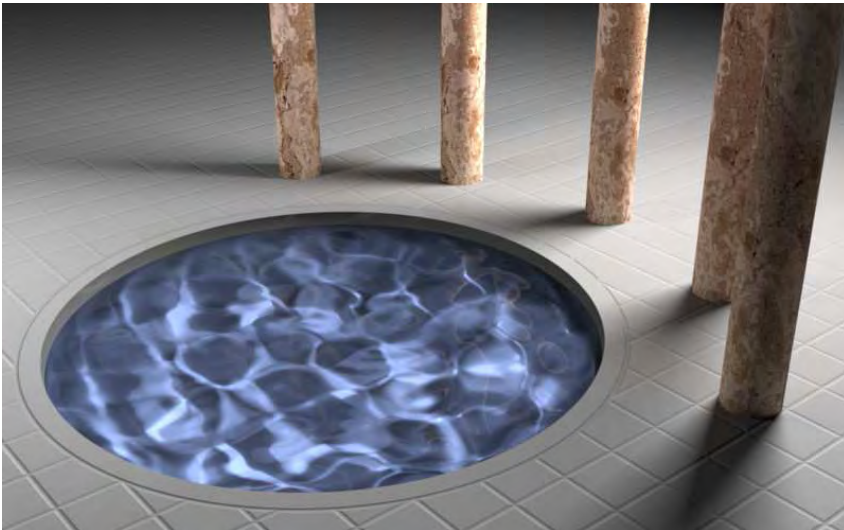
17.20. ábra. Radiosity módszerrel előállított képek.



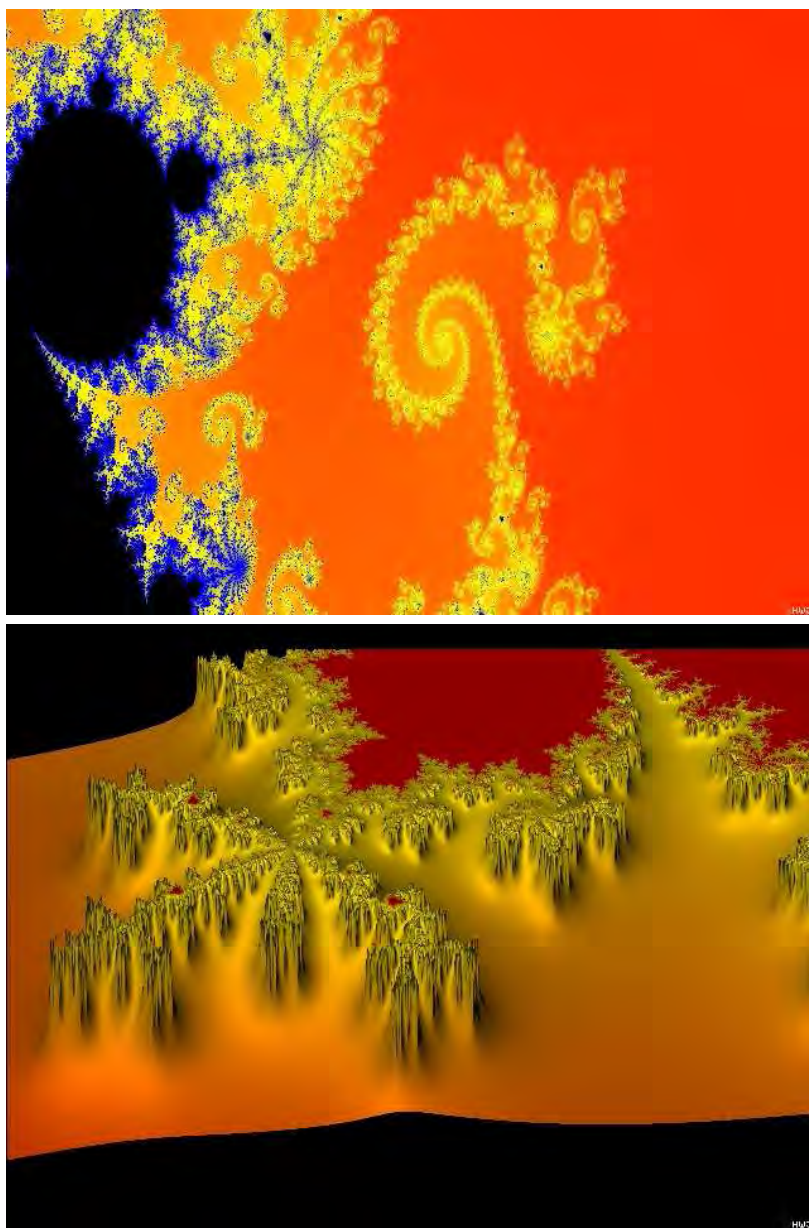
17.21. ábra. Nagyon bonyolult objektumtér radiosity módszerrel készült képe (University of Cornell).



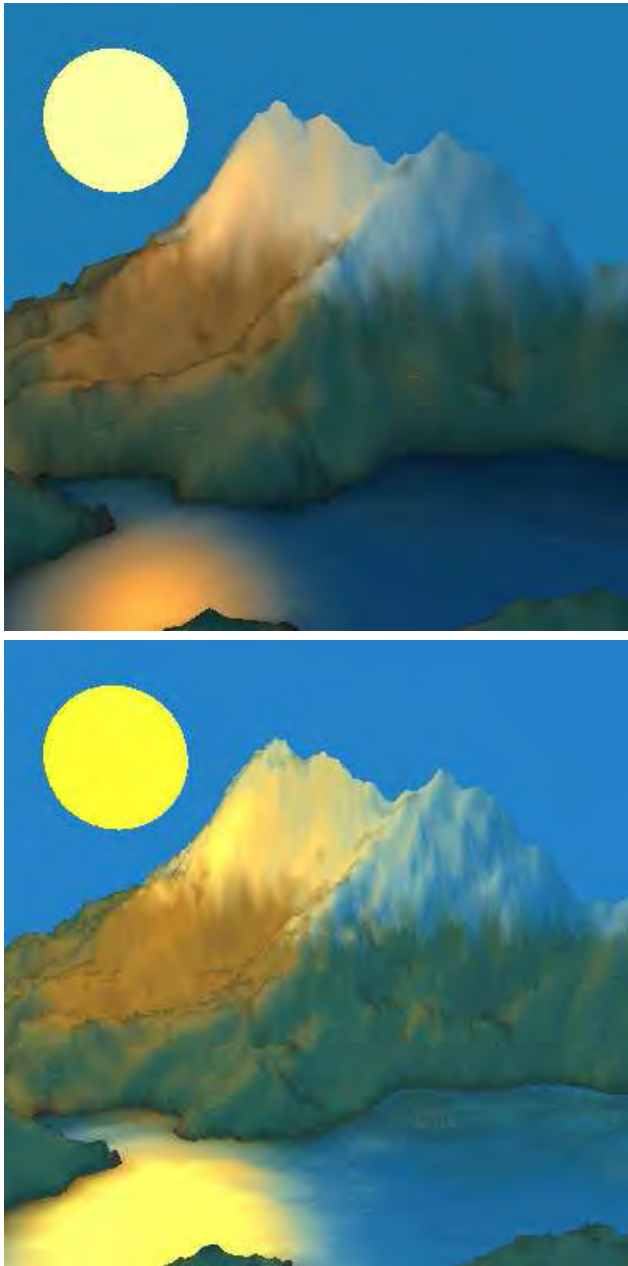
17.22. ábra. Foton térkép alkalmazása (*Mental Images*)



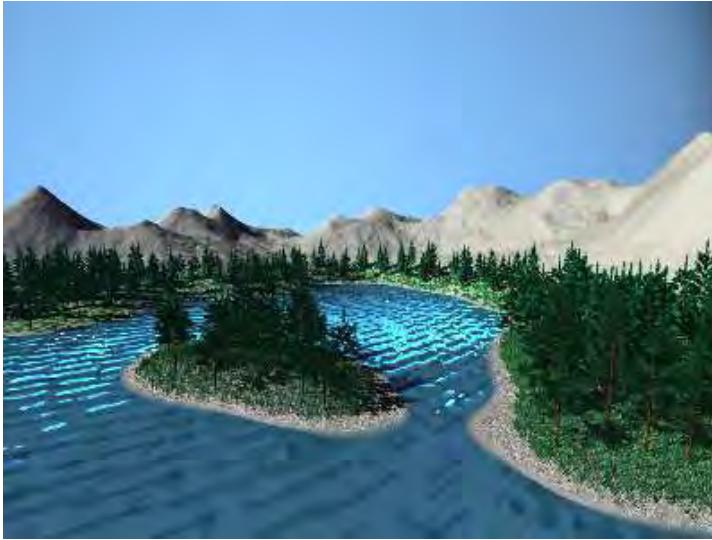
17.23. ábra. Kétirányú fénykövetés Metropolis fontosság szerinti mintavételezéssel (*Eric Veach [VG97]*).



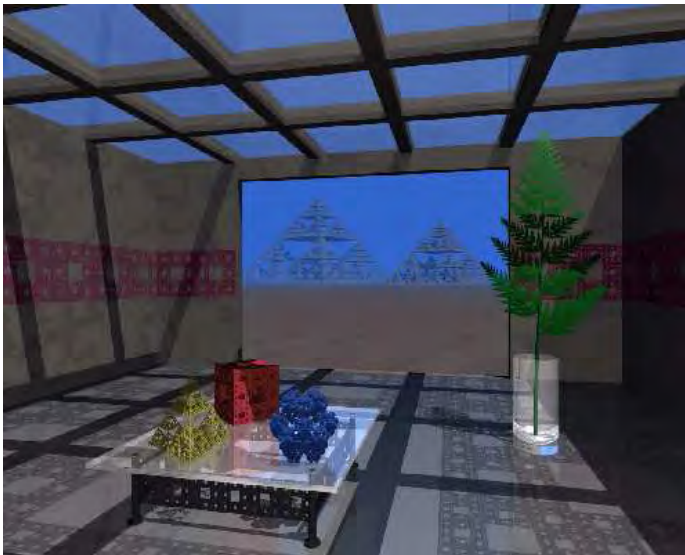
17.24. ábra. Mandelbrot-halmaz megjelenítése. A felső képen a divergencia sebessége csak a színt határozza meg, az alsó képen pedig mind a színt mind pedig a magasságértéket (W. Jensen).



17.25. ábra. Véletlen középpont-elhelyezéssel generált hegyek (a felső képen a rekurziós mélység 4, az alsó képen pedig 7). A megjelenítéshez sugárkötegekkel dolgozó fénykövetési algoritmust használtunk [SK98c].



17.26. ábra. IFS és CSG modellezés kombinációjával előállított modell (Bécsi Műszaki Egyetem, Számítógépes Grafika Intézet).



17.27. ábra. IFS modellek — “Hausdorff-szoba” (Bécsi Műszaki Egyetem, Számítógépes Grafika Intézet).

18. fejezet

CD melléklet

A könyvhöz tartozó CD melléklet *WEB* böngészők (*Netscape Communicator*, *Internet Explorer*, stb.) felhasználásával interaktívan kezelhető. Az összes funkció kihasználásához Windows95 vagy Windows98 operációs rendszer szükséges. A CD lehetőségeinek feltárását a gyökérben lévő *index.htm* fájl megnyitásával kezdhetjük. Amennyiben a böngészőprogramunk a munkakönyvtárat nem helyezi át automatikusan, a böngészőt, úgy célszerű indítani, hogy a “kezdet” tulajdonságot a CD gyökérkönyvtárára állítsuk. Ennek hiányában a példaprogramok esetleg nem találják meg az adatfájlokat.

A bejárható lapokon található horgok némelyike nem a CD-re mutat, hanem a további, részletesebb információk eredeti internet címére. Így ha gépünk közvetlenül kapcsolódik az internethez, a CD-ről indulva a számítógépes grafika állandóan bővülő és frissülő virtuális könyvtárhoz juthatunk el.

A CD-n a példaprogramok, teljes grafikus alkalmazások forrásnyelvű programjai (*Eagles* helikopter szimulátor, *StingRay* Monte-Carlo sugárkövető program, *DOOM* játékprogram, *JPEG* konverter, *OpenGL* könyvtárak és segédprogramok, *DirectX* futtatókörnyezet), grafikus tervezőprogramok leírónyelvének értelmezői (*MGF*), számítógépes grafikával készült képek, *MGF*, *3DS* és *PovRay* formátumú geometriai adatbázisok és teljes, angol nyelvű dokumentációk találhatóak (*OpenGL*, *MGF*). A képtárban híres képszintézis programok által készített képek, és a szerzők saját munkái láthatók. A képekhez tartozó horgokról a képek létrehozását tárgyaló honlapokhoz is eljuthatunk.

18.1. Demonstrációs programok a könyv fejezeteihez

A könyv egyes fejezetei végén C++ nyelvű programrészletek mutatják be, hogy az elmélet hogyan alkalmazható a gyakorlatban. A programok teljes változatban a CD-n is megtalálhatók. A programokat Borland C++ és Microsoft C++ fordítóprogrammal fordítottuk le. A fordítás során “nagy memóriamodellt” illetve WIN32 módot kell választani. A fordítóprogram kapcsolóinak és a *defines.h* fájl *WINDOWS* konstansának

a megfelelő beállításával Ms-Windows és DOS környezetben futtatható programokat hozhatunk létre. Mivel a demonstrációs célok miatt a programok minden grafikus funkciót maguk valósítanak meg, az Ms-Windows és a DOS/BGI könyvtár szolgáltatásai közül csak a pixel írást és olvasást, a képernyő törlést, valamint a billentyűzet- és egérkezelést használjuk. A különböző példaprogramok közös keretrendszert használnak és az algoritmus fájlok egy részét is közösen birtokolják. A következőkben áttekintjük a forrásfájlokat, a könyvtár és az egyes példák működését.

18.1.1. A programokat felépítő közös fájlok

- **defines.h: fordítási paraméter fájl**

Ebben a fájlban leírhatjuk, hogy a következő fordítás során DOS vagy Ms-Windows alkalmazást kívánunk-e a készíteni (WINDOWS), a könyvtár logikai vagy fizikai eszközkoordinátákkal dolgozzon-e (LOGCOORD), foglaljunk-e memóriát a z-buffernek (ZBUFFER), és hogy mekkora legyen a grafikus terület maximális mérete (XRES, YRES). Előfordulhat, hogy a lefordított program rögtön az indítás után *“Dinamikus memória elfogyott”* hibüzenettel leáll. Ezen úgy segíthetünk, hogy vagy teljes egészében kikapcsoljuk a z-buffer memóriát (a 2D programok és a sugárkövetés úgyszemint használják), vagy pedig csökkentjük a grafikus terület méretét.

- **types.h: általános típus fájl**

Ez a fájl a beállított környezet alapján elhelyezi a megfelelő `include` direktívákat (Ms-Windows esetén a `windows.h-t`, DOS esetén a `graphics.h-t`) és definiálja a környezetfüggetlen elérési típusait (`Coord`, `RGBColor`).

- **array.h: generikus dinamikus tömb**

Ebben a fájlban az `Array` generikus dinamikus tömböt definiáltuk.

- **menu.h: egyszerű menü**

Ebben a fájlban található meg a lehetséges felhasználói parancsokat bemutató `Menu` osztályt.

- **draw.h: a fizikai szintű kezelés deklarációs fájlja**

Itt a fizikai szintű elérési típusait (`PColor`, `PCoord`, `ROP`, `PVertex`) és függvényeinek a prototípusát adjuk meg.

- **draw.cpp: a fizikai szintű rajzolás fájlja**

A `PLine` rutinban a szakaszrajzoláshoz a *Bresenham-algoritmust* implementáltuk. A 3D háromszögek árnyalt megjelenítését a `PFacet` eljárás végzi el, amely

a takarási viszonyokat *z-buffer algoritmus*sal határozza meg, a szint pedig *Gou-raud-árnyalással* interpolálja.

- **objwin.h: az objektum-orientált, logikai szintű eszközközkezelő deklarációs fájlja**

Itt található a `Window` osztály, ami a grafikus kimenetet eszközfüggetlen módon illeszti és eseményvezérelt felhasználói kommunikációt valósít meg.

- **window.cpp: az objektum-orientált eszközközkezelő implementációs fájlja**

A `Window` osztály tagfüggvényein kívül itt írtuk le azokat a beviteli eszközöket illesztő rutinokat és fizikai szintű kiviteli eljárásokat (`Pixel`, `PReadPixel`, `PClear`) is, amelyek függenek a futási környezettől (jelen esetben `Ms-Windows` vagy `DOS/BGI`). Ez azt jelenti, hogy ha az olvasó más környezetekben is szeretné használni a megadott programokat, csak ezt a fájlt kell kiegészítenie.

- **tga.h: TARGA fájlkezelő osztályok definíciója**

A TARGA formátum kezelését két osztály támogatja. A `TGAOutputFile` egy TARGA fájlt készít el, a `TGAInputFile` pedig egy TARGA fájlt olvas be. Ezen osztályokat használjuk a képek mentéséhez és betöltéséhez.

- **color.h: színkezelés deklarációs fájlja**

Ebben a fájlban található a `Spectrum` generikus osztály és a `Color` szín osztály definíciója.

- **color.cpp: színtonverziós függvények**

Ez az implementációs fájl a `Color` osztály színtonverziós rutinjait és a színillesztő függvényeit definiálja.

- **2d.h: 2D geometria**

Itt írtuk le az euklideszi pont (`Point2D`), a projektív pont (`HomPoint2D`), a téglalap (`RectAngle`), és a geometriai transzformációk (`Transform2D`) osztályait.

- **polynom.h: paraméteres görbék polinomjai**

A Lagrange-interpolációhoz és Bézier-approximációhoz szükséges polinomok szerepelnek ebben a fájlban.

- **world2.h: a 2D virtuális világ objektumtípusai**

Ez a fájl írja le a 2D virtuális modellek szerkezetét.

- **world2.cpp: a 2D virtuális világ vektorizációja**

A fájlban található tagfüggvények a 2D virtuális világ objektumaihoz tartozó vektorizációs algoritmusokat implementálják.

- **camera2.h: 2D kamera**

Az ablakból és a nézetből álló 2D kameraosztályt (`Camera2D`) találhatjuk meg itt, amely a 2D nézeti transzformáció előállításáért felelős.

- **pipe2.h: 2D kimeneti csővezeték típusai**

A fájl a 2D kimeneti csővezetéken átvihető objektumtípusokat (pont, szakasz, téglalap, szakaszlista) adja meg, és leírja a transzformációjukat.

- **pipe2.cpp: 2D kimeneti csővezeték eljárásai**

Itt lelhetjük fel a 2D kimeneti csővezeték eljárásait, mint például a *Cohen-Sutherland szakasz vágás* algoritmusát.

- **3d.h: 3D geometria**

Ez a 3D euklideszi (`Point3D`) és projektív pont (`HomPoint3D`), és a 3D homogen lineáris geometriai transzformáció (`Transform3D`) definíciós fájlja.

- **world3.h: a 3D virtuális világ objektumtípusai**

A fájl a 3D virtuális világ objektumait definiálja. A virtuális világ transzformálható objektumokból áll, amelyek primitívekből épülnek fel. Egy primitív képviselhet egy gömböt (`Sphere`), háromszögekkel közelített felületet (`PolyFace3D`), törtvonalat (`PolyLine3D`) vagy különböző súlyfüggvényeket használó paraméteres görbét (`Curve3D`).

- **world3.cpp: a 3D virtuális világ vektorizációja és tesszellációja**

Itt adtuk meg a 3D virtuális világ objektumaihoz tartozó vektorizációs és tesszellációs algoritmusokat. A képszintézis első lépéseként az általános primitíveket pontokkal, szakaszokkal és háromszögekkel közelítjük. A szakaszokkal történő közelítést *vektorizációnak*, a háromszögekkel történő közelítést pedig *tesszellációnak* nevezzük. Görbét nyilván csak vektorizálni lehet, a felületeket viszont tetszés szerint vektorizálhatjuk vagy tesszellálhatjuk. Az első esetben *huzalváz megjelenítéshez*, a másodikban pedig *tömör megjelenítéshez* jutunk.

- **camera3.h: 3D kamera deklarációja**

A fájl a 3D virtuális világ leképzéséhez szükséges kamerát definiálja (`Camera3D`), amely a nézeti transzformáció előállításáért felelős.

- **camera3.cpp: 3D nézeti transzformáció előállítása**

A fájlban 3D nézeti transzformációt kiszámító `CalcTransf` tagfüggvényt implementáltuk mind párhuzamos, mind pedig perspektív vetítés esetére.

- **pipe3.h: 3D kimeneti csővezeték objektumtípusai**

A 3D kimeneti csővezetéken átvihető objektumokat a `RenderPrimitive3D` osztályból származtathatjuk. Az ilyen objektumok konkrét típusai a *pont* (`Marker3D`), a *szakasz* (`Line3D`), a *szakaszlista* (`LineList3D`) és a *háromszöglista* (`TriangleList3D`). Az objektumok pontjait (`Transform`) és normálvektorait (`TransformNormals`) transzformálhatjuk, és homogén osztással előállíthatjuk a transzformált pontok Descartes-koordinátáit (`HomDivPoints`). A vágási műveleteket két lépésben hajthatjuk végre. A `DepthClip` homogén koordinátákban az első és hátsó vágósíkra vág, a `Clip` pedig Descartes-koordinátákban nézet téglalapjára. Az `Illuminate` tagfüggvény a normálvektorok alapján az egyes pontokban látható radianciát számítja ki, a `Draw` pedig raszterizálja a primitíveket.

- **pipe3.cpp: a 3D csővezeték eljárásai**

A 2D kimeneti csővezeték eljárásait találhatjuk itt meg, mint például a *Cohen-Sutherland szakasz vágás* homogén koordinátákra és 3D térre általánosított algoritmusát, és a `brdf.h`-ban megfogalmazott BRDF modellekre építő illuminációs algoritmust.

- **brdf.h: BRDF modellek**

A felületek optikai tulajdonságait a BRDF modellekkel írjuk le. Egy felület lehet fénykibocsátó (`Emitter`) és a környezetéből ideérkező fényt is visszaverheti illetve törheti. A visszaverődés lehet diffúz (`DiffuseMaterial`), spekuláris (`SpecularMaterial`) vagy ideális (`IdealReflector`). A fénytörésnél csak az ideális esetet modellezzük (`IdealRefractor`). Ebből az inkrementális képszintézisben a `DiffuseSpecularMaterial` osztályt használjuk, a sugárkövetés pedig mindet. A `FresnelFunction` fémekre a visszaverődési tényezőt számítja ki. A `GeneralMaterial` a különböző visszaverődési és törési típusokat egyesíti. Egy BRDF modellnek alapvetően két funkciója van. Egyrészt a BRDF függvény megmondja, hogy adott irányú megvilágítás és nézőpont esetén milyen intenzitású fényt érzékel a megfigyelő. Másrészt, egy belépő irány birtokában a BRDF és a kilépő szög koszinusza szorzatának arányában egy véletlen kilépő irányt kell generálni. Ebből a két funkcióból az elsőre minden algoritmusnak szüksége van, a másodikat viszont csak a Monte-Carlo módszerek használják. A `brdf.h` fájlban az `SColor` típus megadásánál két lehetőség közül

választhatunk. Mivel a sugárkövető program mindenütt az `SCOLOR` típust használja, a `brdf.h`-ban szereplő `RGBCOL` konstans értékével szabályozhatjuk, hogy a spektrumot csak a vörös, zöld és kék színek hullámhosszain számítjuk vagy több hullámhosszon követjük a fény terjedését a térben.

- **brdf.cpp: fémek törésmutatói**

A fájlban a fémekhez szükséges törésmutató táblázatokat találhatjuk meg, amelyeket a Fresnel-együtthatók számításához használunk. Itt definiáljuk az alacsony diszkrepanciájú sorozatot jelképező objektum egy példányát is.

18.1.2. Grafikus keretrendszer Ms-Windows és DOS/BGI környezetre

A programok egy közös grafikus könyvtárat illetve keretet használnak, amely a fordításnak megfelelően Ms-Windows vagy DOS környezetben eszközfüggetlen grafikus megjelenítést és eseményvezérelt interakciót valósít meg. A könyvtár felépítését a 2.3. fejezetben tárgyaltuk.

A könyvtárhoz a következő fájlok tartoznak: `defines.h`, `types.h`, `array.h`, `menu.h`, `draw.cpp`, `objwin.h`, `window.cpp`. Egy alkalmazás úgy építhet a könyvtár szolgáltatásaira, hogy a `Window` osztályból egy alkalmazás specifikus osztályt származtat (ennek neve `MyWindow`), amelyben az ablak alkalmazásfüggő részeit leírja. Ezt követően a belépési ponton (`AppStart`) létrehozuk az ablakobjektumot és beindítjuk az üzenetciklust.

18.1.3. Példa alkalmazások

TARGA formátumú képek megjelenítése: `tgashow.exe`

A `tgashow` példaprogram TARGA típusú képfájlokat jelenít meg. A programot az 1.7. fejezetben tárgyaltuk. A képfájl nevét parancssor argumentumként kell megadni. A megjelenített képet *medián szűrő* és *doboz szűrő* algoritmusokkal módosíthatjuk, majd az eredményt eltárolhatjuk (18.1. ábra).

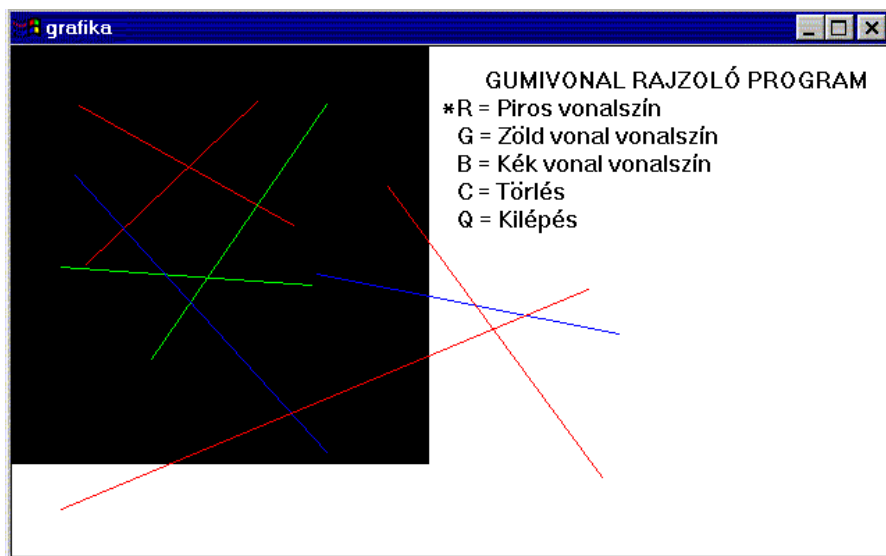
Az alkalmazás belépési pontját, az ablakobjektumát és a szűrőalgoritmusokat a `tgashow.cpp` fájl tartalmazza. A programhoz, az általános keretfájlokon kívül, a `tga.h` fájl tartozik, amelyben a TARGA fájlkezelő osztályok definíciója található.

Gumivonal rajzoló program: `gumi.exe`

Ez a program a 2.3.4. fejezetben tárgyalt gumivonal rajzoló implementációja, amely az eseményvezérelt programozás fogásait mutatja be (18.2. ábra). A teljes program a `gumi.cpp` fájlban olvasható, amelyet a keretrendszerhez kell hozzászerkeszteni.

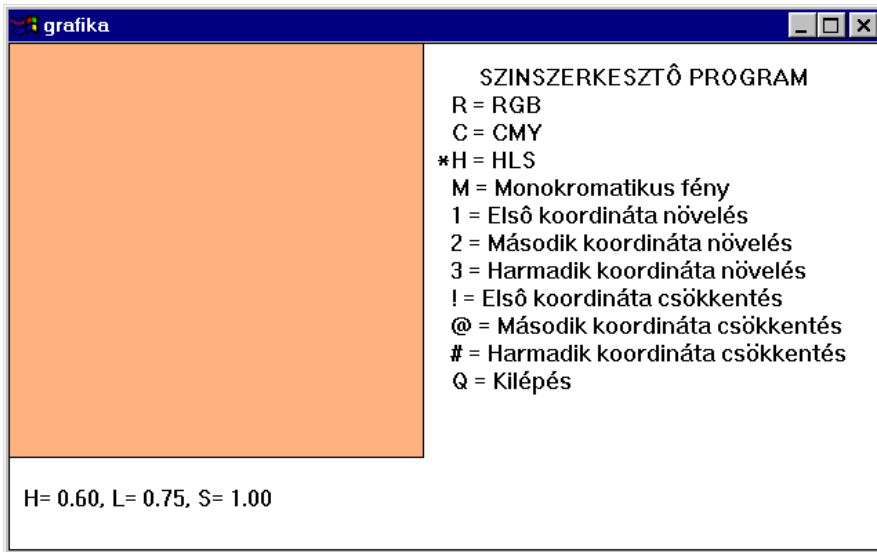


18.1. ábra. TARGA formátumú képek megjelenítése



18.2. ábra. Gumivonal rajzoló program

Színszerkesztő: color.exe



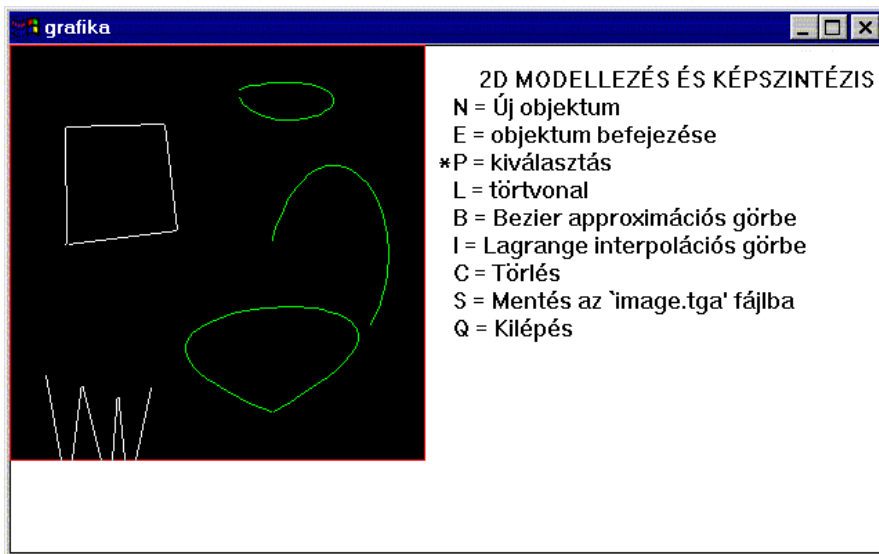
18.3. ábra. Színszerkesztő program

A színszerkesztő program a 4.3. fejezet színkezelő osztályait és algoritmusait használja fel. A program segítségével a grafikus terület háttérszínét változtathatjuk RGB, CMY, HLS színrendszerben, illetve egy monokromatikus spektrum hullámhosszának a módosításával (18.3. ábra). Az interaktív kezelői felületet a `colgen.cpp` fájl valósítja meg, amely épít a `color.h` színsztályaira és a `color.cpp`-ben implementált színkonverziós és a színillesztő algoritmusokra.

2D grafikus rendszer: 2d.exe

Ez az alkalmazás egy teljes 2D grafikus rendszert mutat be. A program lényeges elemeit a 7.9. fejezetben ismertettük. A program segítségével interaktív módon törtvonalakat, Bézier-görbéket és Lagrange-görbéket definiálhatunk. A korábban definiált görbéket az egér kurzor és a bal egérgomb segítségével kiválaszthatjuk. A kiválasztott görbét áthelyezhetjük, ha a kiválasztott görbe felett a bal egérgombot ismételten lenyomjuk, majd az egérgombot nyomva tartva az egeret mozgatjuk. A kiválasztott objektumokat XOR módban rajzoljuk, az áthelyezés hasonló a gumivonal technikához (18.4. ábra).

A program tartalmazza a görbék modellezéséhez szükséges adatszerkezeteket és algoritmusokat. A megjelenítéshez a teljes 2D grafikus csővezeték implementáltuk, amely vektorizálja a görbéket, alkalmazza a modellezési és nézeti transzformációkat,



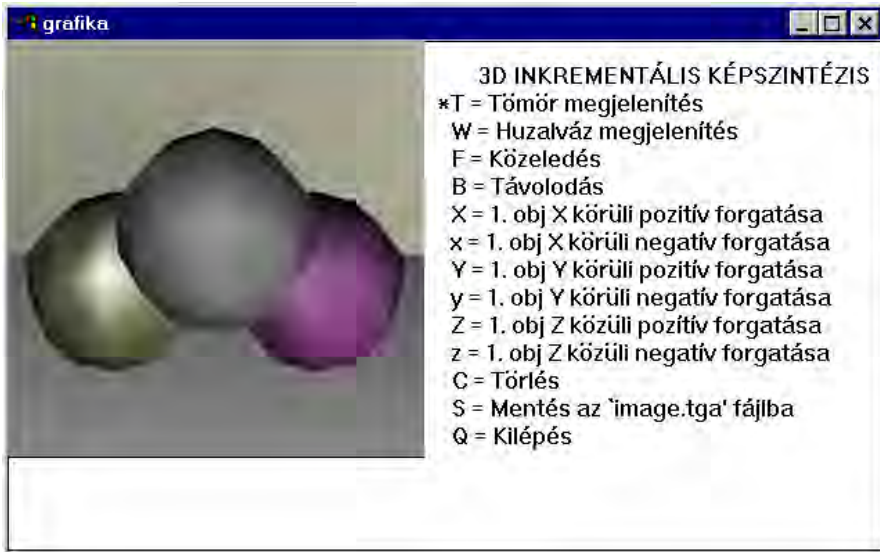
18.4. ábra. 2D grafikus rendszer

elvégzi a vágást, majd raszterizálja a kapott szakaszokat. A `2d.cpp` fájl a virtuális világot és a kamerát összefogó színteret (`Scene`), a színtér módosításához és megjelenítéséhez szükséges függvényeket, azaz a bemeneti és a kimeneti csővezeték algoritmusait, és a felhasználói interakció eljárásait adja meg. Az alkalmazás a virtuális világmodell felállításához a `world2.h` és `world2.cpp` fájlok programjait, a kamera kezeléshez a `camera2.h` osztályait, a kimeneti csővezeték megvalósításához pedig a `pipe2.h` és `pipe2.cpp` fájlok szolgáltatásait használja. A kimeneti csővezeték végén megjelenő szakaszokat a keret `draw.cpp` fájljában implementált a *Bresenham-algoritmus* raszterizálja. A pixel műveleteket a `window.cpp` fájlban valósítottuk meg.

3D grafikus rendszer inkrementális képszintézissel: 3d.exe

Ez az alkalmazás egy teljes 3D grafikus képszintézis rendszert mutat be, amely z-buffer takarási algoritmust és Gouraud-árnyalást alkalmaz (18.5. ábra).

A `3D.cpp` fájl a virtuális világot és a kamerát összefogó színteret (`Scene`), a kimeneti csővezeték működtető függvényt (`Render`), és a felhasználói interakció eljárásait adja meg. A virtuális világot a `sphere` fájlban építjük fel. Az alkalmazás a virtuális világmodell felállításához felhasználja a `world3.h` és a `world3.cpp` fájlok programjait, a kamera kezeléshez a `camera3.h` fájl és a `camera3.cpp` fájl osztályait, a kimeneti csővezeték megvalósításához a `pipe3.h` és a `pipe3.cpp` szolgáltatásait, az illuminá-



18.5. ábra. 3D grafikus rendszer inkrementális képszintézissel

ciós számításoknál pedig a `brdf.h`, a `brdf.cpp` és a `light.h` fájlokat.

A kimeneti csővezeték végén megjelenő szakaszokat a keret `draw.cpp` fájljában implementált *Bresenham-algoritmus* raszterizálja, a 3D háromszögeket (*facet*) pedig a z-buffer takarási módszert és a Gouraud árnyalási eljárást megvalósító `PFacet` függvény alakítja át pixelekké. Ezen műveletek kódolása során a nagysebességű, célszerűen gépi kódú implementáció, illetve a hardver támogatás lehetőségének a bemutatása érdekében csak egész műveleteket használtunk.

Sugárkövetés: `ray.exe`

Ez a példa több alapvető módszert demonstrál, a nem-rekurzív és a *rekurzív sugárkövetést*, a sugárkövetés kiegészítését a sztochasztikus mintavételezés és az utószűrés kombinálását alkalmazó *csipkézettség csökkentéssel*, és egy *inverz fényútkövetés* elvű Monte-Carlo globális illuminációs algoritmust. A felületek optikai tulajdonságainál *textúra leképzést* is beállíthatunk (18.6. ábra). A sugárkövető algoritmusok a `ray.cpp` fájlban találhatóak, amelyek a textúra leképzésnél, az illuminációs és mintavételezési lépéseknél építenek a `brdf.h` és a `brdf.cpp` fájlok szolgáltatásaira, a fényforrások kezelésénél a `light.h` fájl osztályaira, a kamera szimulációja során pedig a `camera3.h`-ban definiált osztályra. A virtuális világot a `tsphere` fájlban építjük fel. A véletlen és kvázi-véletlen számsorozatokot a `uniform.h` fájlban állítjuk elő.



18.6. ábra. Sugárkövetés

Térfogatvizualizáció masírozó kockák algoritmussal: march.exe

Ez az alkalmazás egy voxeltömböt tölt be a parancssor argumentumával megnevezett fájlból, majd az interaktívan változtatható érték felhasználásával a masírozó kockák algoritmussal szintfelületet generál, amit az inkrementális 3D képszintézis rendszer kimeneti csővezetékén jelenít meg. A térfogatmodellt leíró fájl bináris. A fájl egy egész számmal kezdődik, amely meghatározza a voxeltömb felbontását. A méret mezőt követő bájt sorozat pedig az egyes voxelek sűrűségértékeit adja meg. Ha például az első szóban 128-t találunk, a fájl még ezen kívül $128 \times 128 \times 128$ bájtot tartalmaz.

A masírozó kockák algoritmus és a kezelői felület a `march.cpp` fájlban található. A voxel és a szintfelület lehetséges metszeteit a `polytab.h` fájlban adtuk meg. Az alkalmazás ezen kívül felhasználja az általános keretrendszert és a 3D kamera fájlokat (`camera3.h`, `camera3.cpp`), a kimeneti csővezeték fájljait (`pipe3.h` és `pipe3.cpp`) és a BRDF fájlt (`brdf.h`).

IFS megjelenítő: ifs.exe

Ez az alkalmazás a parancssorban megadott nevű fájlból egy IFS-t olvas be és azt véletlen bolyongással megjeleníti. Az IFS megjelenítő és a kezelői felület az `ifs.cpp` fájlban található. Az alkalmazás ezen kívül felhasználja az általános keretrendszert, a 2D kamera fájlt (`camera2.h`) és a transzformációs fájlt (`2d.h`).



18.7. ábra. Tértfogatvizualizáció masírozó kockák algoritmussal



18.8. ábra. IFS megjelenítő

Irodalomjegyzék

- [ANW67] J. Ahlberg, E. Nilson, and J. Walsh. *The Theory of Splines and their Applications*. Academic Press, 1967.
- [Arv86] J. Arvo. Backward ray tracing. In *SIGGRAPH '86 Developments in Ray Tracing*, 1986.
- [Arv91] J. Arvo. Linear-time voxel walking for octrees. *Ray Tracing News*, 1(2), 1991. available under anonymous ftp from weedeater.math.yale.edu.
- [AW87] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics '87*, pages 3–10, 1987.
- [Bau72] B. G. Baumgart. Winged-edge polyhedron representation. Technical Report STAN-CS-320, Computer Science Department, Stanford University, Palo Alto, CA, 1972.
- [BBB87] R. Bartels, J. Beatty, and B. A. Barsky. *An Introduction on Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann, Los Altos, CA, 1987.
- [Bez72] P. Bezier. *Numerical Control: Mathematics and Applications*. Wiley, Chichester, 1972.
- [BG89] P. Burger and D. Gillies. *Interactive Computer Graphics: Functional, Procedural and Device-Level Methods*. Addison-Wesley, Wokingham, England, 1989.
- [BKP91] J. C. Beran-Koehn and M. J. Pavicic. A cubic tetrahedra adaption of the hemicube algorithm. In James Arvo, editor, *Graphics Gems II*, pages 299–302. Academic Press, Boston, 1991.
- [Bli77] J. F. Blinn. Models of light reflections for computer synthesized pictures. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 192–198, 1977.
- [Bli78] J. F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 286–292, 1978.
- [BN76] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.
- [Bre65] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

- [BS63] P. Beckmann and A. Spizzichino. *The Scattering of Electromagnetic Waves from Rough Surfaces*. MacMillan, 1963.
- [BSKG97] Csébfalvi B., L. Szirmay-Kalos, and Márton G. Fast opacity control in rendering of volumetric ct data. In *Winter School of Computer Graphics '97*, pages 79–87, Plzen, Czech Republic, 1997.
- [Cat78] E. Catmull. A hidden-surface algorithm with anti-aliasing. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 6–11, 1978.
- [CB78] M. Cyrus and J. Beck. Generalized two- and three-dimensional clipping. *Computers and Graphics*, 3(1):23–28, 1978.
- [CCWG88] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, pages 75–84, 1988.
- [CG85] M. Cohen and D. Greenberg. The hemi-cube, a radiosity solution for complex environments. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, pages 31–40, 1985.
- [Chi88] H. Chiyokura. *Solid Modelling with DESIGNBASE*. Addison Wesley, 1988.
- [CLSS97] P. H. Christensen, D. Lischinski, E. J. Stollnitz, and D. H. Salesin. Clustering for glossy global illumination. *ACM Transactions on Graphics*, 16(1):3–33, 1997.
- [Coo86] R. L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, 1986.
- [Cox74] H.S.M. Coxeter. *Projective Geometry*. University of Toronto Press, Toronto, 1974.
- [CPC84] R. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, pages 137–145, 1984.
- [Cro77] F. C. Crow. Shadow algorithm for computer graphics. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 242–248, 1977.
- [Cse97] B. Csebfalvi. A review of Monte-Carlo ray tracing algorithms. In *CESCG '97, Central European Seminar on Computer Graphics*, pages 87–103, 1997.
- [CT81] R. Cook and K. Torrance. A reflectance model for computer graphics. *Computer Graphics*, 15(3), 1981.
- [DCH88] A.D. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, 22(4), 1988.
- [Deá89] I. Deák. *Random Number Generators and Simulation*. Akadémia Kiadó, Budapest, 1989.
- [Dév93] F. Dévai. *Computational Geometry and Image Synthesis*. PhD thesis, Computer and Automation Institute, Hungarian Academy of Sciences, Budapest, Hungary, 1993.

- [DH92] John Denskin and Pat Hanrahan. Fast algorithms for volume ray tracing. *Workshop on Volume Visualization*, pages 91–98, 1992.
- [DLW93] P. Dutre, E. Lafortune, and Y. D. Willems. Monte Carlo light tracing with direct computation of pixel intensities. In *Compugraphics '93*, pages 128–137, Alvor, 1993.
- [Dor97] P. Dornbach. Eagles — pc based virtual reality engine for terrain modelling. In *Central European Seminar on Computer Graphics, CESC G '97*, pages 11–22, 1997.
- [Duv90] V. Duvanenko. Improved line segment clipping. *Dr. Dobb's Journal*, july, 1990.
- [Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, Berlin, 1987.
- [Far88] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, New York, 1988.
- [Fed95] M. Fedá. *Radiosity*. Institute of Computer Graphics, Vienna University of Technology, Vienna, 1995.
- [FFC82] A. Fournier, D. Fussell, and L. C. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.
- [Fis97] Y. Fisher. *Fractal Image Compression*. 1997.
- [FS75] R. Floyd and L. Steinberg. An adaptive algorithm for spatial gray scale. In *Society for Information Display 1975 Symposium Digest of Technical Papers*, page 36, 1975.
- [FTK86] A. Fujimoto, T. Takayuki, and I. Kansei. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.
- [FvD82] J. D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, Mass., 1982.
- [FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, Mass., 1990.
- [GCT86] D. P. Greenberg, M. F. Cohen, and K. E. Torrance. Radiosity: A method for computing global illumination. *The Visual Computer* 2, pages 291–297, 1986.
- [Gla89] A. S. Glassner. *An Introduction to Ray Tracing*. Academic Press, London, 1989.
- [Gla95] A. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers, Inc., San Francisco, 1995.
- [Gro95] E. Groeller. *Scientific Visualisation*. Technical University of Vienna, Institute of Computer Graphics, Vienna, 1995. in German.
- [GSS81] S. Gupta, R. Sproull, and I. Sutherland. Filtering edges for gray-scale displays. In *Computer Graphics (SIGGRAPH '81 Proceedings)*, pages 1–5, 1981.

- [Hal86] R. Hall. A characterization of illumination models and shading techniques. *The Visual Computer* 2, pages 268–277, 1986.
- [Hal89] R. Hall. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, 1989.
- [Hap63] B. Hapke. A theoretical photometric function for the lunar surface. *Journal of Geophysical Research*, 68(15), 1963.
- [Hec86] P. S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, 1986.
- [Hec97] P. Heckbert. Brdf viewer, <http://www.cs.cmu.edu/~afs/cs.cmu.edu/~user/ph/www/src/illum>. 1997.
- [Her91] Ivan Herman. *The Use of Projective Geometry in Computer Graphics*. Springer-Verlag, Berlin, 1991.
- [HS67] H. C. Hottel and A. F. Sarofin. *Radiative Transfer*. McGraw-Hill, New-York, 1967.
- [HSA91] P. Hanrahan, D. Salzman, and L. Aupperle. Rapid hierarchical radiosity algorithm. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 1991.
- [HTSG91] X. He, K. Torrance, F. Sillion, and D. Greenberg. A comprehensive physical model for light reflection. *Computer Graphics*, 25(4):175–186, 1991.
- [Hun87] R. W. Hunt. *The Reproduction of Colour*. Fountain Press, Tolworth, England, 1987.
- [ICG86] D. S. Immel, M. F. Cohen, and D. P. Greenberg. A radiosity method for non-diffuse environments. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, pages 133–142, 1986.
- [JC95] H. W. Jensen and N. J. Christensen. Photon maps in bidirectional Monte Carlo ray tracing of complex objects. *Computers and Graphics*, 19(2):215–224, 1995.
- [JC98] H. W. Jensen and P. H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. *Computers and Graphics (SIGGRAPH '98 Proceedings)*, pages 311–320, 1998.
- [Jen96] H. W. Jensen. Global illumination using photon maps. In *Rendering Techniques '96*, pages 21–30, 1996.
- [JGMHe88] K. I. Joy, C. W. Grant, N. L. Max, and Lansing Hatfield (editors). *Computer Graphics: Image Synthesis*. IEEE Computer Society Press, Los Alamitos, CA., 1988.
- [Kaj85] J. T. Kajiya. Anisotropic reflection models. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, pages 15–21, 1985.
- [Kaj86] J. T. Kajiya. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, pages 143–150, 1986.

- [Kel96] A. Keller. Quasi-Monte Carlo Radiosity. In X. Pueyo and P. Schröder, editors, *Rendering Techniques '96 (Proc. 7th Eurographics Workshop on Rendering)*, pages 101–110. Springer, 1996.
- [Ken94] E. Ken. Reflectance phenomenology and modeling tutorial. 1994. <http://www.erim.org>.
- [Knu81] D.E. Knuth. *The art of computer programming. Volume 2 (Seminumerical algorithms)*. Addison-Wesley, Reading, Mass. USA, 1981.
- [Kra89] G. Krammer. Notes on the mathematics of the phigs output pipeline. *Computer Graphics Forum*, 8(8):219–226, 1989.
- [LB84] Y.D. Lian and B.A. Barsky. A new concept and method for line clipping. *ACM TOG*, 3(1):1–22, 1984.
- [Lev88] M. Levoy. Display of surfaces from ct data. *IEEE Computer Graphics and Application*, 8:29–37, 1988.
- [Lev90] M. Levoy. Efficient ray tracing of volume data. *ATG*, 9(3):245–261, 1990.
- [Lew93] R. Lewis. Making shaders more physically plausible. In *Rendering Techniques '93*, pages 47–62, 1993.
- [LH91] David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics (SIGGRAPH '91 Proceedings)*, pages 285–288, 1991.
- [LKSK95] Z. László, K. Kondorosi, and L. Szirmay-Kalos. *Objektum-orientált szoftverfejlesztés*. ComputerBooks, Budapest, 1995.
- [Löf98] H. Löffelmann. *Visualizing Local Properties and Characteristic Structures of Dynamical Systems*. PhD thesis, Vienna University of Technology, <http://www.cg.tuwien.ac.at/~helwig/diss>, 1998.
- [LW93] E. Lafortune and Y. D. Willems. Bi-directional path-tracing. In *Compugraphics '93*, pages 145–153, Alvor, 1993.
- [LW94] E. Lafortune and 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.
- [Män88] M. Mäntylä. *Introduction to Solid Modeling*. Computer Science Press, Rockville, MD., 1988.
- [Már95a] G. Márton. Acceleration of ray tracing via voronoi-diagrams. In Alan W. Paeth, editor, *Graphics Gems V*, pages 268–284. Academic Press, Boston, 1995.
- [Már95b] G. Márton. *Stochastic Analysis of Ray Tracing Algorithms*. PhD thesis, Department of Process Control, Technical University of Budapest, Budapest, Hungary, 1995.

- [MH84] G. S. Miller and C. R. Hoffman. Illumination and reflection maps: Simulated objects in simulated and real environment. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, 1984.
- [Min41] M. Minnaert. The reciprocity principle in lunar photometry. *Astrophysical Journal*, 93:403–410, 1941.
- [NFML88] Derek R. Ney, Elliot K. Fishman, Donna Magid, and Marc Levoy. Computed tomography data: Principles and techniques. *IEEE Computer Graphics and Application*, 8, 1988.
- [Nie92] H. Niederreiter. *Random number generation and quasi-Monte Carlo methods*. SIAM, Pennsylvania, 1992.
- [NNL98] L. Neumann, A. Neumann, and Szirmay-Kalos L. Analysis and pumping up the albedo function. Technical Report TR-186-2-98-20, Institute of Computer Graphics, Vienna University of Technology, 1998. www.cg.tuwien.ac.at/.
- [NNS72] M. E. Newell, R. G. Newell, and T. L. Sancha. A new approach to the shaded picture problem. In *Proceedings of the ACM National Conference*, pages 443–450, 1972.
- [NNSK98] L. Neumann, A. Neumann, and L. Szirmay-Kalos. New simple reflectance models for metals and other specular materials. Technical Report TR-186-2-98-17, Institute of Computer Graphics, Vienna University of Technology, 1998. www.cg.tuwien.ac.at/.
- [NS79] W. M. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics, Second Edition*. McGraw-Hill Publishers, New York, 1979.
- [ON94] M. Oren and S. Nayar. Generalization of lambert's reflectance model. *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 239–246, 1994.
- [Pap98] G. Papp. *Skalármező poligonizációja, marching cubes, felületrekonstrukció*. <http://www.inf.bme.hu/~rod/onlab>, 1998.
- [Pea85] D. R. Peachey. Solid texturing of complex surfaces. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, pages 279–286, 1985.
- [PF90] P. Poulin and A. Fournier. A model for anisotropic reflection. *Computer Graphics*, 24(4):273–281, 1990.
- [PFTV92] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C (Second Edition)*. Cambridge University Press, Cambridge, USA, 1992.
- [Pho75] B. T. Phong. Illumination for computer generated images. *Communications of the ACM*, 18:311–317, 1975.
- [PM95] S. N. Pattanik and S. P. Mudur. Adjoint equations and random walks for illumination computation. *ACM Transactions on Graphics*, 14(1):77–102, 1995.

- [PP98] J. Prikryl and W. Purgathofer. Perceptually based radiosity. In *Eurographics '98, STAR — State of the Art Report*, 1998.
- [PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.
- [PSe88] F. Peitgen and D. Saupe (editors). *The Science of Fractal Images*. Springer-Verlag, New York, 1988.
- [PTG95] W. Purgathofer, R. Tobler, and M. Geiler. Improved threshold matrices for ordered dithering. In Alan W. Paeth, editor, *Graphics Gems V*, pages 297–301. Academic Press, Boston, 1995.
- [R76] P. Rózsa. *Lineáris algebra és alkalmazásai*. Műszaki Könyvkiadó, Budapest, 1976.
- [RA89] D. F. Rogers and J. A. Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill, New York, 1989.
- [Rog85] D. F. Rogers. *Procedural Elements for Computer Graphics*. McGraw Hill, New York, 1985.
- [SAWG91] F. X. Sillion, J. R. Arvo, S. H. Westin, and D. P. Greenberg. A global illumination solution for general reflectance distributions. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):187–198, 1991.
- [Sbe96] M. Sbert. *The Use of Global Directions to Compute Radiosity*. PhD thesis, Catalan Technical University, Barcelona, 1996.
- [SC94] F. Sillion and Puech C. *Radiosity and Global Illumination*. Morgan Kaufmann Publishers, Inc., San Francisco, 1994.
- [SDS95] F. Sillion, G. Drettakis, and C. Soler. Clustering algorithm for radiance calculation in general environments. In *Rendering Techniques '95*, pages 197–205, 1995.
- [SH74] I.E. Sutherland and G.W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.
- [SH81] R. Siegel and J. R. Howell. *Thermal Radiation Heat Transfer*. Hemisphere Publishing Corp., Washington, D.C., 1981.
- [SK94] L. Szirmay-Kalos. Dynamic layout algorithm to display general graphs. In Paul Heckbert, editor, *Graphics Gems IV*. Academic Press, Boston, 1994. <http://www.iit.bme.hu/~szirmay>.
- [SK95] L. Szirmay-Kalos. Stochastic sampling of two-dimensional images. In *COMPUTER GRAPHICS '95*, Alvor, 1995.
- [SK96] L. Szirmay-Kalos. Application of variational calculus in the radiosity method. *Periodica Polytechnica (Electrical Engineering)*, 40(2):123–138, 1996.
- [SK98a] L. Szirmay-Kalos. Global ray-bundle tracing. Technical Report TR-186-2-98-18, Institute of Computer Graphics, Vienna University of Technology, 1998. www.cg.tuwien.ac.at/.

- [SK98b] L. Szirmay-Kalos. Stochastic iteration for non-diffuse global illumination. Technical Report TR-186-2-98-21, Institute of Computer Graphics, Vienna University of Technology, 1998. www.cg.tuwien.ac.at/.
- [SK98c] L. Szirmay-Kalos. Stochastic methods in global illumination — state of the art report. Technical Report TR-186-2-98-23, Institute of Computer Graphics, Vienna University of Technology, 1998. www.cg.tuwien.ac.at/.
- [SKe95] L. Szirmay-Kalos (editor). *Theory of Three Dimensional Computer Graphics*. Akadémia Kiadó, Budapest, 1995. <http://www.iit.bme.hu/~szirmay>.
- [SKFNC97] L. Szirmay-Kalos, T. Fóris, L. Neumann, and B. Csébfalvi. An analysis to quasi-Monte Carlo integration applied to the transillumination radiosity method. *Computer Graphics Forum (Eurographics'97)*, 16(3):271–281, 1997.
- [SKM94] L. Szirmay-Kalos and G. Márton. On hardware implementation of scan-conversion algorithms. In *8th Symp. on Microcomputer Appl.*, Budapest, Hungary, 1994.
- [SKMFF96] L. Szirmay-Kalos, G. Márton, T. Fóris, and J. Fábíán. Application of object-oriented methods in process visualisation. In *Winter School of Computer Graphics '96*, pages 349–358, Plzen, Czech Republic, 12–16 February 1996. <http://www.iit.bme.hu/~szirmay>.
- [SKP98] L. Szirmay-Kalos and W. Purgathofer. Quasi-Monte Carlo solution of the rendering equation. Technical Report TR-186-2-98-24, Institute of Computer Graphics, Vienna University of Technology, 1998. www.cg.tuwien.ac.at/.
- [Sob91] I. Sobol. *Die Monte-Carlo Methode*. Deutscher Verlag der Wissenschaften, 1991.
- [SP92] V. Székely and A. Poppe. *Számítógépes grafika alapjai IBM PC-n*. Computer Books, Budapest, 1992.
- [SPS98] M. Stamminger, Slussalek P., and H-P. Seidel. Three point clustering for radiance computations. In *Rendering Techniques '98*, pages 211–222, 1998.
- [SSS74] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, 1974.
- [SWZ96] P. Shirley, C. Wang, and K. Zimmerman. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(1):1–36, 1996.
- [Sza95] Zs. Szalavári. Rendering natürlicher atmosphärischer lichteffecte. Technical report, <http://www.cg.tuwien.ac.at/research/rendering/halos>, 1995.
- [Tob98] R. F. Tobler. ART — Advanced Rendering Toolkit. Technical report, 1998. <http://www.cg.tuwien.ac.at/research/rendering/ART>.
- [TS66] K. Torrance and M. Sparrow. Off-specular peaks in the directional distribution of reflected thermal distribution. *Journal of Heat Transfer — Transactions of the ASME*, pages 223–230, May 1966.
- [Uli87] R. Ulichney. *Digital Halftoning*. Mit Press, Cambridge, MA, 1987.

- [Vea97] E. Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, http://graphics.stanford.edu/papers/veach_thesis, 1997.
- [VG95] E. Veach and L. Guibas. Bidirectional estimators for light transport. In *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 419–428, 1995.
- [VG97] E. Veach and L. Guibas. Metropolis light transport. *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 65–76, 1997.
- [War92] G. Ward. Measuring and modeling anisotropic reflection. *Computer Graphics*, 26(2):265–272, 1992.
- [War94] G. J. Ward. The RADIANCE lighting simulation and rendering system. *Computer Graphics*, 28(4):459–472, 1994.
- [Wat89] A. Watt. *Fundamentals of Three-dimensional Computer Graphics*. Addison-Wesley, 1989.
- [Wil83] L. Williams. Pyramidal parametric. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 1–11, 1983.
- [Wol90] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, Washington, DC., 1990.
- [WS82] G. Wyszecki and W. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. Wiley, New York, 1982.