SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM

MAROSVÁSÁRHELYI KAR

jegyzet

KOVÁCS D. LEHEL ISTVÁN

SZÁMÍTÓGÉPES GRAFIKA

2021

Scientia Kiadó

KOVÁCS D. LEHEL ISTVÁN

SZÁMÍTÓGÉPES GRAFIKA

SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MŰSZAKI ÉS HUMÁNTUDOMÁNYOK KAR, MAROSVÁSÁRHELY
MATEMATIKA–INFORMATIKA TANSZÉK

KOVÁCS D. LEHEL ISTVÁN

SZÁMÍTÓGÉPES GRAFIKA

Scientia Kiadó
Kolozsvár · 2021

A kiadvány megjelenését a Sapientia Alapítvány támogatta.

Lektor:
Makó Zoltán (Csíkszereda)

Felelős kiadó:
Sorbán Angella

Kiadói koordinátor:
Szabó Beáta

Sorozatborító:
Miklósi Dénes

Első magyar nyelvű kiadás: 2009
Második változatlan kiadás: 2021

ISBN: 978-606-975-054-4

TARTALOM

CONTENTS

CUPRINS

ELŐSZÓ

Az ember mindig is ábrázolni akarta gondolatait, hogy másoknak megmutathassa, szemléltesse, másokkal megossza ezeket. Valószínű, hogy e célból születtek meg mintegy 30 000 éve az első barlangrajzok is.

A számítógépes grafika segítségével a számítógépből tudunk olyan eszközt varázsolni, amely vázlatos gondolatainkról képet tud alkotni, és ezáltal elképzeléseinket képre, képsorozatra, rajzfilmre, filmre tudja vinni. A számokkal leírt digitális modellt a számítógép „lefényképezi", a generatív számítógépes grafika segítségével fotorealisztikus képet állít elő, amelyet megszemlélhetünk, amely alapján igényeinknek megfelelően testreszabhatjuk, megváltoztathatjuk, finomíthatjuk a modellt, majd arról újabb képet készíthetünk.

E könyv bevezető a generatív számítógépes grafika varázslatos világába, ahol a kacsalábon forgó palota kockákból, téglatestekből, gúlákból, kúpokból van felépítve, az RGB(0, 124, 195) színű égen fraktál-felhők úsznak, és a legkisebb királyfi textúra-füves tájon, Perlin-zajjal generált sziklák alatt, Barnsley-páfrányok között megy megkeresni az inverz kinematikával mozgatott hétfejű sárkányt.

A könyv megírásakor a Sapientia – EMTE marosvásárhelyi karán oktatott *Számítógépes grafika* tanrendjét vettem figyelembe, de az egyetemi hallgatókon kívül jól használhatják a középiskolás tanulók is, vagy mindazok, akik az ismereteiket szeretnék elmélyíteni e terén, vagy fel kívánnak készülni az olyan speciális informatika-versenyekre (pl. KovInfo, render.hu stb.), amelyeken a számítógépes grafika a fő téma.

Jelen könyv nem tartalmazza a görbék és felületek matematikai alapjait, generáló algoritmusait, valamint a teljes virtuális valóság bemutatását, ezeket egy külön kötetben fogom ismertetni.

dr. Kovács Lehel István
klehel@ms.sapientia.ro

Marosvásárhely, 2009. március 31.

1. FEJEZET

BEVEZETÉS A SZÁMÍTÓGÉPES GRAFIKÁBA

1.1. A számítógépes grafika célja

A számítógépes grafika, animáció, képfeldolgozás fejlődése az elmúlt 30–40 évben rendkívül felgyorsult. A generatív számítógépes grafika és animáció mára a számítástechnika egyik külön tudományágává fejlődött. A felhasználási területek is igen elszaporodtak.

Mindez a következő tényezőknek köszönhető [12]:

- A grafikus felületek (GUI) használata világszabvánnyá vált. A programozók és az egyes szoftvergyártó cégek egyaránt arra törekednek, hogy programjaik minél szebbek, látványosabbak legyenek.
- A technológia lehetővé teszi a fotorealisztikus, valósághű 3D megjelenítést, és ennek az interaktív szerkesztését.
- A hardver, főleg a videokártyák rohamosan fejlődtek.
- A fotorealisztikus képábrázolást lehetővé tevő algoritmusok (pl. sugárkövetés) hatékonysága rohamosan javul.
- Az animáció, speciális effektusok használata a filmiparban kinőtte magát.
- A multimédia és a generatív számítógépes grafika között a határ elmosódott.
- A grafikus programcsomagok előállítói (pl. Corel, Autodesk, Adobe stb.) hatalmas, és kiéleződött a piaci verseny.
- A nyomdatechnika hatalmasat fejlődött.
- A fontosabb tervezőprogramok (CAD) már képesek az osztott csapatmunkára.
- Az orvostudományban egyre nagyobb az igény a 3D képfeldolgozásra.
- Az élet majdnem minden területét betöltik a grafikus szimulációk, szimulátorok.
- A televízióadások, szórakoztató média egyre intenzívebben használja a speciális grafikai effektusokat.

A gyors fejlődés fő okát a képi információk kifejezőerejében kell keresni. A diagramok, az ábrák, a képek sokkal átláthatóbbak, hatékonyabban hordozzák az információt, mint a szöveges leírás.

Ezt már az ókorban is tudták, sőt maga a *grafika* szó is az ógörög γράφω (grápho), γραπικόζ (graphikós) szóból származik, amely a *vésni*, *véset* szavakat jelenti, az ókorban leginkább így állították elő az ábrákat.

A grafika ma a rajzművészet összefoglaló fogalmát jelenti. A *grafika* a képzőművészet azon ága, amelyhez a sokszorosítási eljárással készült, de eredetinek tekinthető alkotások tartoznak, illetve azok az egyszeri alkotásokról (pl. festmény) sokszorosító eljárással készült reprodukciók, amelyek nem tekinthetők egyedi alkotásnak. Gyakran idesorolnak olyan képzőművészeti eljárásokat is, amelyek nem nyomatok, de szintén papír alapot használnak, mint például a ceruza-, toll- és krétarajzok, akvarellek, esetleg pasztellképek; vagy nyomtatási eljárással készülnek ugyan, de csak egy példányban, mint a monotípia. A felület kitöltése többnyire vonalak segítségével történik, szemben a festészettel, ahol inkább foltokkal.

A számítógépek kezdetben nem voltak képesek grafikus ábrázolásra, szöveggel fejeztek ki mindent. Később jöttek létre az első vonalas ábrázolások, majd a formák, végül a háromdimenziós ábrázolás.

Mára már a számítógépes grafikának is relatív önálló ágai különültek el, ilyenek:

- *Generatív számítógépes grafika (interactive computer graphics)*: a képi információ tartalmára vonatkozó adatok és algoritmusok alapján modelleket állít fel, képeket jelenít meg (*renderel*). Idetartozik a speciális effektusok előállítása vagy az animáció is, amely a generált grafikát az időtől teszi függővé. Általában két- (2D) vagy háromdimenziós (3D) grafikus objektumok számítógépes generálását, tárolását, felhasználását és megjelenítését fedi a fogalom. A cél a *fotorealisztikus*, valós ábrázolásmód, vagyis az, hogy a számítógépes grafikával generált képeket gyakorlatilag nem lehet megkülönböztetni a fényképtől vagy videofelvételektől. Rendszerprogramozói, programozói és kevésbé felhasználói szintű műveletek összessége.
- *Számítógéppel segített grafika (computer aided graphics – CAG)*: a számítógép bevonása ábrázolásmódok, számítások, folyamatok megkönnyítésére, pl. függvényábrázolás, nyomdai grafikai munkálatok, sokszorosítás, diagramkészítés, illusztrátorok stb. Felhasználói és programozói szintű műveletek összessége.
- *Képfeldolgozás (image processing)*: mindazon számítógépes eljárások és módszerek összessége, amelyekkel a számítógépen tárolt képek minőségét valamilyen szempont szerint javítani lehet. Itt nem generált képekkel dolgozunk, hanem inputként megkapott képekkel, pl. digitális fényképezőgép, szkenner vagy más digitalizáló eszközzel előállított raszteres képekkel. Felhasználói és kevésbé programozói szintű műveletek összessége.
- *Képelemzés, alakfelismerés (picture analysis, form recognition)*: a raszteres képeken lévő grafikus objektumok azonosítását végzi el. Felhasználói és programozói szintű műveletek összessége.
- *Számítógéppel segített tervezés és gyártás (computer aided design and manufacturing – CAD/CAM)*: olyan számítógépen alapuló eszközök

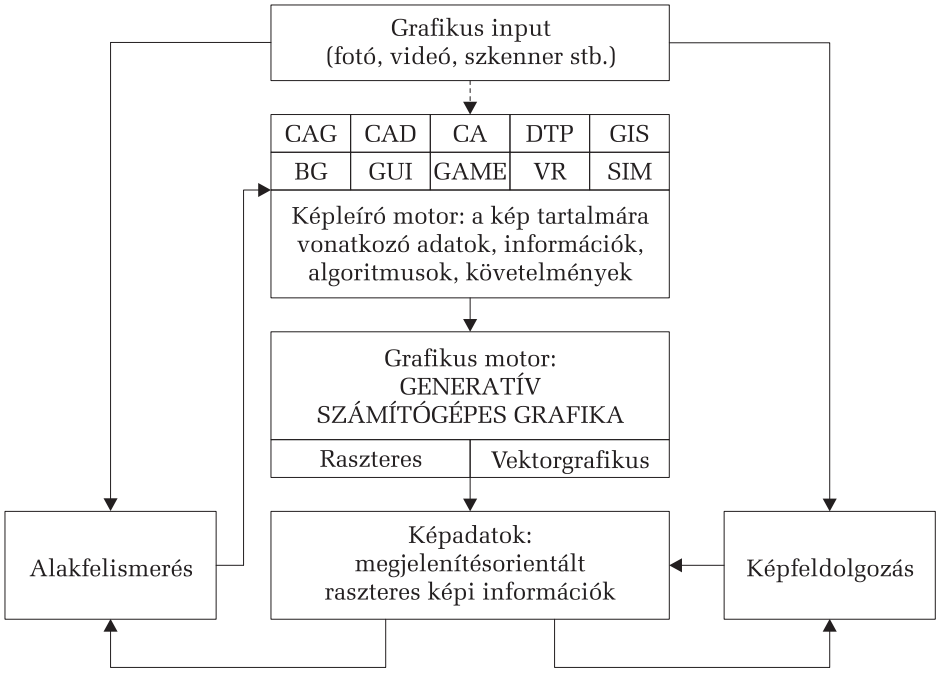
összessége, amely a mérnököket és más tervezési szakembereket tervezési tevékenységükben segíti. A jelenleg használatos CAD programok a 2D (síkbeli) vektorgrafika alkalmazásán rajzoló rendszerektől a 3D (térbeli) parametrikus felület- és szilárdtest-modellező rendszerekig a megoldások széles skáláját kínálják. Felhasználói és kevésbé programozói szintű műveletek összessége.

- *Térképészeti információs rendszerek (geographical information system – GIS)*: a térképek számítógépes feldolgozását lehetővé tevő rendszerek. Felhasználói és kevésbé programozói szintű műveletek összessége.
- *Grafikus bemutatók (bussines graphics)*: az üzleti életben, tudományban, közigazgatásban stb. bemutatott grafikus alapú prezentációk elkészítése a vizuális információ átadásának céljából. Multimédiás oktatóprogramok, reklámok, honlapok készítése. Felhasználói szintű műveletek összessége.
- *Folyamatok felügyelésére szakosodott grafikus rendszerek*: különböző szenzorok által szolgáltatott mérési adatok grafikus feldolgozása és ezek alapján bizonyos folyamatok vezérlése, felügyelése. Idetartoznak az ipari folyamatok vezérlései, de például egy ház fűtőrendszerének a felügyelete is. Rendszerprogramozói, programozói és felhasználói szintű műveletek összessége.
- *Számítógépes szimulációk*: repülőgép és űrhajó-szimulátorok, időjárás-előrejelzés készítése számítógépes szimulációval, egyszerű folyamatok szimulálása, valósághű jelenetek valósidejű megjelenítése. Rendszerprogramozói, programozói és felhasználói szintű műveletek összessége.
- *Számítógépes játékok*: olyan játékok, amelyekkel a játékos egy felhasználói felületen keresztül lép kölcsönhatásba, és arról egy kijelző eszközön keresztül kap visszajelzéseket. A visszajelzések történhetnek látványban, hangban és fizikailag is, különböző, folyamatosan fejlődő technikai eszközök segítségével. Két főcsoportja ismeretes: a személyi számítógépekre írt játékok és a videojáték-konzolokra írt játékok. Rendszerprogramozói, programozói és felhasználói szintű műveletek összessége.
- *Felhasználói grafikus felületek (graphical user interface – GUI)*: operációs rendszerek, számítógépes alkalmazások grafikus felületeinek megtervezése, és így a felhasználóval egy magasabb szintű interakció megvalósítása. Rendszerprogramozói, programozói és felhasználói szintű műveletek összessége.
- *Szöveg- és kiadványszerkesztés (desk top publishing – DTP)*: számítógéppel segített nyomdai kiadványszerkesztés, speciális képek, betűtípusok, emblémák, logók, reklámfigurák elkészítése. Felhasználói és kevésbé programozói szintű műveletek összessége.
- *Virtuális valóság (virtual reality – VR):* olyan technológiák összessége, mely különleges eszközök révén a felhasználó szoros interakcióba kerül

a grafikus világgal, mintegy részévé válik. Rendszerprogramozói, programozói és felhasználói szintű műveletek összessége.

Nyilvánvaló, hogy a felsoroltak nagy többsége beleillik a generatív számítógépes grafika tágabban vett fogalmába, sőt mindegyiknek a magvát, az alapját a képgenerálás (képszintézis) képezi, mindazonáltal önálló szakterületté nőtték ki magukat, saját módszertannal, eszközökkel rendelkeznek.

Ha a fentieket egy diagramba kívánnánk összefoglalni, az 1.1. ábrán látható viszonyrendszert kapnánk ([12] alapján).



1.1. ábra. *A számítógépes grafika szakágazatai*

*

Jelen könyvben a *generatív számítógépes grafiká*t tárgyaljuk (ezt általában egyszerűen csak *számítógépes grafiká*nak nevezzük), és ennek is a programozói szintű bemutatására törekszünk, algoritmusokat, lehetőségeket, eszközöket ismertetünk.

*

Amint az 1.1. ábrán is megfigyelhetjük, a generatív számítógépes grafika grafikus motra kétfajta feldolgozásra (eredmény-előállításra) képes: *raszteres*re és *vektorgrafikus*ra.

A *vektorgrafikus ábrázolásmód* esetében a grafikai modell egyes elemei (objektumai) matematikailag egyértelműen leírható alakzatok, vonalak, görbék stb. A kis helyigényen kívül előnyük, hogy felépítésüknél fogva tetszőlegesen átméretezhetők anélkül, hogy minőségük romlana, így a vektorgrafikus képek nyomtatásánál csak a nyomtató felbontása szab határt. Az objektumokat önállóan tároljuk, ezek egyedileg is visszakereshetők, módosíthatók stb., a köztük lévő strukturális kapcsolatok a számítógép által feldolgozhatók. A vektorgrafikus rendszerekben az objektumokat lebegőpontos világ-koordinátarendszerben ábrázoljuk. Egy pontot a hozzá vezető helyzetvektorral lehet azonosítani. Az objektumokat *drótvázas (wireframe)*, *árnyalt (solid)* vagy *fotorealisztikus (photorealistic)* módon jeleníthetjük meg.

Drótvázas módban a testeket csak az éleikkel ábrázoljuk. Az ábrán nincsenek takart vonalak, minden él teljes egészében megjelenik. Ez a legegyszerűbb és leggyorsabb megjelenítési mód, viszont a legkevésbé valósághű.

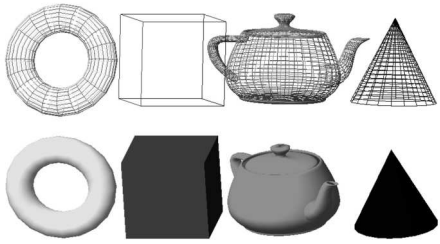
Árnyalt megjelenítés esetében a testek felületét is ábrázoljuk, a határoló felületek kitöltött képét rajzoljuk ki. Az árnyalással ábrázolhatjuk a testek anyagainak jellemzőit, a fényhatásokat, a takarásokat. A képen az eltakart részek nem fognak megjelenni. A vektorgrafikus objektumok árnyalt megjelenítését *renderelés*nek (*rendering*) nevezzük.

Fotorealisztikus megjelenítésen azt értjük, hogy a vektorgrafikus modelltérbeli jelenetről olyan minőségű képet állítunk elő, amely teljesen valószerű, a valós világról készített fényképtől nem lehet megkülönböztetni.

1.2. ábra. *POV-Ray-jel renderelt fotorealisztikus kép (forrás: [37])*

Fotorealisztikus képek előállításának követelményei ([12] alapján):

- *Térhatás (depth cueing)*: A 3D-s modelltér jelenete a 2D-s raszteres képen is térhatású legyen. Érvényesüljön a perspektivikus ábrázolási mód. Reálisan ábrázoljuk a tárgyak látható és nem látható éleit, felületeit. Érvényesüljön a mélység-élesség. A messzeségbe tűnő objektumok legyenek elmosódottabbak, kevésbé kidolgozottak. Használjuk a *mip-maping* technikát.
- *Felületek megvilágítása, tükröződés, árnyékok*: modellezzük és használjuk fel a természetben is lezajló jelenségeket. A képeken a fényhatások feleljenek meg a természet és a fizika törvényeinek. A természethűség érdekében használjunk természetes (természetutánzó) textúrákat. Érdes, göröngyös térhatású felületeket tudunk elkészíteni a *bump-maping* technikával, amikor a felületre merőlegesen véletlenszerűen módosítjuk a tárgy felszínét: kiemelünk, lesüllyesztünk. A testek egymásra vetett árnyékait meg kell jeleníteni.
- *Átlátszóság, áttetszőség, köd, füst modellezése*: figyelembe kell venni a fénytörést, a fény intenzitásának csökkenését. Használjuk az *alpha-blending* technikát.



1.3. ábra. *Testek drótvázas és árnyalt ábrázolása*

A *raszteres ábrázolásmód* esetében a kép *pixel*ekből (*picture element* – a legkisebb ábrázolható egység), vagyis képpontokból áll. A képi információ csak képként kereshető vissza. Csak az egyes képpontok színét tároljuk, így tetszőleges árnyalatot adhatunk vissza. Ennek előnye a nagyjából korlátlan színhasználat, amelynek segítségével a fényképek tökéletesen megjeleníthetők. Hátrányuk viszont a nagy helyigény és a méretváltoztatáskor fellépő minőségromlás.

A képen található objektumok számítógéppel csak speciális alakfelismerő algoritmusok segítségével azonosíthatók be.

Generatív számítógépes grafikában a képszintézis utolsó fázisában a 3D modellről 2D-s raszteres grafikát állítunk elő, ez jeleníthető meg a képernyőn vagy nyomtatásban.

1.4. ábra. *Digitális fénykép – raszteres grafika*

1.2. A számítógépes grafika története

1.2.1. Korai előzmények

A mintegy 30 000 éves barlangrajzokkal kezdődően [62] az emberiség történetét átszövi a művészet, az ábrázolás, a grafika, a festészet, szobrászat, architektúra, dizájn. Megszámlálhatatlan próbálkozás történt a valós, háromdimenziós világ síkban történő ábrázolására, megjelenítésére. Ezen próbálkozások között voltak matematikailag pontatlanok, de pontosak is, voltak olyanok, amelyek kielégítették a mai értelemben vett *képiesség* fogalmát, és voltak, amelyek kevésbé. A művészettörténetre támaszkodva elmondhatjuk, hogy az ókori görögök már minden bizonnyal ismerték és alkalmazták a *perspektivikus* képalkotás fogalmát – sajnos festmények nem, de leírások maradtak fenn – (számunkra kiemelkedően fontos Eukleidész (kb. Kr. e. 300–Kr. e. 250), akinek geometriai meglátásai a grafika alapjait képezik), de a kérdéskört matematikai pontossággal csak a reneszánszban kezdték el vizsgálni.

A *perspektíva* szabályainak kikísérletezésére szánta életét Giotto di Bondone (1267–1337), aki a következő módszert fejlesztette ki: A szemlélő feltételezett szemmagasságába húzott egy, a kép alsó szélével párhuzamos egyenest, majd az efölé eső, távolodó vonalakat lefelé, az egyenes alá esőket felfelé térítette el. Ügyelt a távolabbi alakok méretére, valamint a megfelelő színek használatára is.

Eljárása nem volt matematikailag alátámasztva, ám próbálkozásai nagyban hozzájárultak a későbbi reneszánsz mesterek tudományos alapú ábrázolásának fejlődéséhez.

Filippo Brunelleschi (1377–1446) kiterjedt geometriai ismeretekkel rendelkező művészként szükségesnek érezte, hogy pontos munkamódszert dolgozzon ki, amit esetleg társai is hasznosítani tudnak. Eljárása azon alapult, hogy a majdani kompozíció látószögének megfelelően kijelölt egy pontot a vásznon, ahová az összes, a kép síkjára merőleges vonal összefut. Az ábrázolt tárgyak és alakok az így megválasztott *enyészpont*tól mért távolságuk alapján lesznek kisebbek vagy nagyobbak – megközelítőleg úgy, ahogy a valóságban látjuk őket.

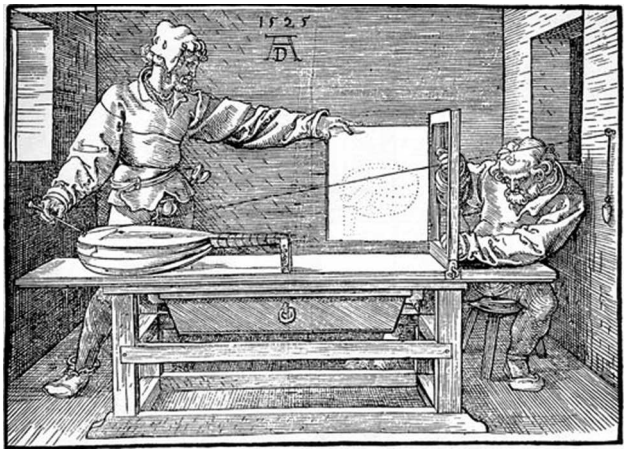
Leone Battista Alberti (1404–1472) vette észre először, hogy kört úgy érdemes perspektivikusan ábrázolni, hogy azt először egy négyzethálós lapra rajzoljuk, majd a négyzethálót „elferdítve" megkeressük az eredeti körrel való metszéspontoknak megfelelő (transzformált) pontokat, s így *ellipszis*t kapunk. Ő volt az, aki a festményre is pontos matematikai definíciót kívánt adni: „Egy képzeletbeli, rögzített középpontú gúla metszete bizonyos távolságból, a fény meghatározott helyzete mellett, vonalak és színek által, művészi módon, adott felületen ábrázolva."

1450 körül Johann Gutenberg (kb. 1400–1468) feltalálta a könyvnyomtatást. Habár 1041-ben már Kínában alkalmaztak gépi eljárást szövegek papírra való nyomtatására, a könyvnyomtatást, ahogy azt ma ismerjük, Gutenberg vezette be.

Leonardo da Vinci (1452–1519) maga is folytatott geometriai tanulmányokat. Ezek során rájött, hogy az egy enyészponton alapuló perspektíva különböző méretűnek tünteti fel a szemlélőtől azonos, ám az enyészponttól eltérő távolságban levő alakokat. A hiba kiküszöbölésére megalkotta a *természetes perspektívá*t, amelyben a rövidülés a nézőtől való távolság arányában történik. Megkülönböztetésül a vonalperspektívát *mesterséges perspektívá*nak nevezte el [1].

A perspektíva szabályainak tanulmányozásában kiemelkedően tevékenykedett Ajtósi Dürer (1471–1528), aki fizikai eszközt szerkesztett a centrális projekció tanulmányozására. Az 1.1. ábrán bemutatott eszközzel a művész egy lantot próbál lerajzolni. Jobbra a falon van a *centrumpont*, jelen esetben egy csiga. A tárgy egy pontjából fonal vezet a csigán át, amelyet súly feszít ki. Ekkor a keretben lévő függőleges és vízszintes vonalzókat a fonalhoz tolja a jobb oldali ember. A fonalat leengedik, a lapot, amelyet most a bal oldali ember tart, ráhajtják a keretre és megjelölik rajta az előbbi fonál „döféspontját". Ha az eljárást kellő számú tárgypontra megismételték, akkor megjelent a papíron a hangszer képe.

Az 1500-as éveket követő nagyszámú feltalálásra, újításra való tekintettel, a teljesség igénye nélkül soroljuk fel azokat a kiemelkedő személyiségeket, akiknek munkássága jelentős előzményt nyújtott a számítógépes grafika ma is használt elemeinek megjelenéséhez [87].



1.5. ábra. *Ajtósi Dürer eszköze*

René Descartes (1596–1650) vezette be az analitikus mértant, és a róla elnevezett sajátos koordináta-rendszert.

Gottfried Wilhelm Leibniz (1646–1716) és Issac Newton (1642–1727) a dinamikus rendszerek elméletét alapozták meg.

Az 1800-as évek elején egymástól függetlenül több kutató is megoldotta a *camera obscura* által rajzolt kép rögzítésének technikai problémáját. Így jelent meg a fényképészet és ennek különböző válfajai.

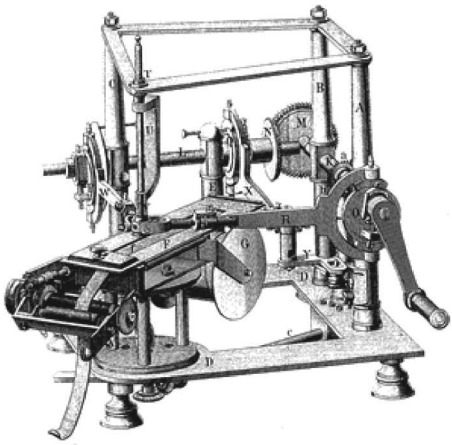
1843-ban Alexander Bain (1811–1877) megalkotta a fax elődjét.

1884-ben Paul Nipkow (1860–1940) feltalálta a képfelbontás elvét a róla elnevezett pásztázótárcsával. Megalkotta a szkenner ősét is.

Jedlik Ányos (1800–1895) 1872-ben mutatta be Herkulesfürdőn a *Vibrograph*ot. Lissajous 1855-től számos értekezésében tárgyalta, hogyan lehet különböző rezgések eredőjét meghatározni. A Lissajous-idomok (görbék) mechanikus eszközzel való megrajzoltatására szinte minden kísérletező megalkotta saját szerkezetét. A Vibrographon az összetett mozgás képét tű rajzolta kormozott üvegre. Ezeket Jedlik vékony lakkréteggel vonta be, így eredeti ábrái megmaradtak. A gép igen pontos mechanikus konstrukció volt, Jedlik találmányával jóval megelőzte korát, a mai elektronikus rajzológépekkel sem lehet sokkal pontosabb görbéket rajzolni. A Lissajous-féle görbék lerajzolásához felhasználta, hogy a körmozgás vetülete rezgőmozgás. Két kerék mozgása adta a két merőleges rezgést, vezette a rajzolót.

1888-ban Thomas Alva Edison (1847–1931) és William Dickson (1860–1935) megalkották a *kinetoszkóp*ot, amely egymás utáni képekből mozgóképsorozatot állított elő egy hengeren.

James Joseph Sylvester (1814–1897) alkotta meg a mátrixokat. A számítógépes grafikában használatos transzformációk mátrixokkal írhatók le.



1.6. ábra. *Jedlik Ányos eszköze. Két rezgésszerű és egy haladó mozgásnak eredőjét lerajzoló gépezet (forrás: Pannohalmi Könyvtár és levéltár)*

1895. december 28-án Louis Jean Lumière (1864–1948) és fivére, Auguste (1862–1954) bemutatták saját filmjeikből álló előadásukat a párizsi Grand Caféban. Így született meg a film és a mozi, az alkotó művészetek között az első, amelyik a teret és az időt egyszerre, közvetlenül használja föl, időben és térben egyszerre működik.

1897-ben Karl Ferdinand Braun (1850–1918) kifejlesztette a katódsugárcsövet (CRT – *Cathode Ray Tube*).

1907-ben a Lumière fivérek bemutatták az *autokróm eljárást*. 1909-ben a londoni Palace varietében levetítették az első színes hatású filmet. 1917-ben a tényleges színes film bemutatkozására az Amerikai Egyesült Államokban a *technicolor* eljárás adott lehetőséget. 1936-ban a *szubtraktív színkeverés* fejlettebb eljárást alkalmazták, ami az *Agfacolor* néven vált ismertté a színes filmek körében.

A XX. század elején analóg számítógépeket kezdtek építeni olyan problémák megoldására, amelyeket másképp nem tudtak megoldani, 1911-ben megjelennek a *totalizátorok*. Ezeket a fix programozású, számkijelzős (előre megrajzolt „grafikus kijelző") elektromechanikus gépeket leginkább a kutya- és lóversenyek fogadási esélyeinek kiszámítására használták.

1923-ban alakult meg a *Disney Brothers Cartoon Studio* (Walt Disney), amely ma is a rajzfilmgyártás élvonalában jár, eddig 67 rajzfilmet, több ezer rövidfilmet (rajzfilm), valamint 10 filmbe animációs jeleneteket készített. Az 1986-ban alakult és számítógépes grafikával előállított rajzfilmekre szakosodott Pixar stúdióval 14 közös rajzfilmet készített.

1.7. ábra. *Totalizátor*

1924-ben találta fel Tihanyi Kálmán (1897–1947) a teljesen elektronikus, töltéstároló típusú televíziós rendszert, 1926-ban kelt a magyar szabadalmi bejelentése.

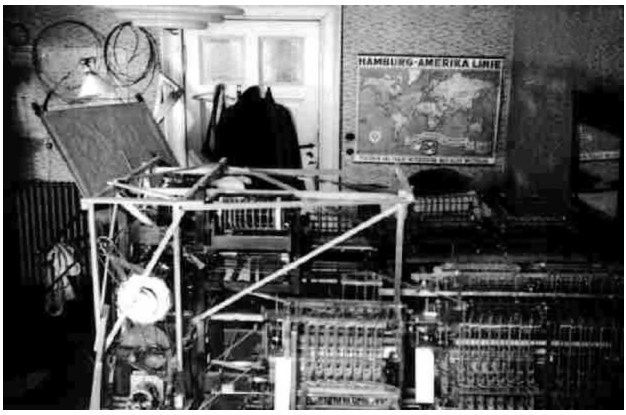
1927-ben került sor London–Glasgow között az első, nagy távolságra vezetéken továbbított televíziós adásra John Logie Baird (1888–1946) skót feltaláló jóvoltából.

1936 és 1938 között Konrad Zuse (1910–1995) *Z*1 néven olyan szabadon programozható számítógépet épített, amely a kettes számrendszert használta, lebegőpontos számokkal dolgozott, az adatbevitelre billentyűzet szolgált, az adatkivitel pedig egy fénymátrix segítségével történt.

1938-ban találta fel Chester Carlson (1906–1968) a száraznyomtatás technikáját.

A második világháború ideje alatt, *Neumann János* (1903–1957) magyar származású matematikus elgondolása alapján kezdte el *John Presper Mauchly* (1919–1995) és *John William Eckert* (1907–1980) az ENIAC (*Electronic Numerical Integrator And Computer*) tervezését katonai célokra. Ezek a számítógépek többnyire papíron, lyukkártyán, lyukszalagokon jelenítették meg a számítások eredményét, vagy egyszerű égőket (pl. fénymátrix) használtak.

Isaac Jacob Schoenberg (1903–1990) 1946-ban vezette be a *spline-görbé*ket, olyan görbéket, amelyek szakaszosan parametrikus polinomokkal leírhatók. A spline-okat azért használják előszeretettel a számítógépes grafika területén, mert egyszerű és interaktív szerkesztést tesznek lehetővé, pontosságuk, stabilitásuk



1.8. ábra. *Zuse gépe – a Z*1

és könnyű illeszthetőségük révén igen komplex formákat lehet velük jól közelíteni.

1947-ben Gábor Dénes (1900–1979) feltalálta a *hologram*ot és a *holográfiá*t. Ezzel a képek rögzítésének egy olyan módját fedezte fel, ami több információ visszaadását tette lehetővé, mint bármelyik addig ismert eljárás. A holográfia a fény hullámtermészetén alapuló olyan képrögzítő eljárás, amellyel a tárgy struktúrájáról tökéletes térhatású, vagyis 3D kép hozható létre. Találmányáért Gábor Dénes 1971-ben fizikai Nobel-díjat kapott.

1.2.2. A számítógépes grafika hőskora

A számítógépes grafika első fontos momentuma a katonai jellegű *Whirlwind Project* 1945-ös elindulása volt. Az MIT-nél helyet kapó projekt fő célja egy repülés-szimulátor elkészítése volt SAGE számítógépes rendszeren. A SAGE kijelzője egy vektorgrafikus kijelző volt, és itt használták először a fényceruzát. A projekt keretében a Whirlwind Computer kifejlesztette az első valósidejű grafikus megjelenítőt, 1949-ben megjelent a képernyő (CRT-elvű). Valószínűleg az első radarok és oszcilloszkópok mintájára az első képernyő is még kerek volt.

1953-ban a Remington-Rand megalkotta a Univac számítógéphez az első gyorsnyomtatót.

1956-ban Ray Dolby (1933–), Charles Ginsburg (1920–1992) és Alexander M. Poniatoff (1892–1980) az Ampexnél megalkották az első videofelvevő kamerát, később Dolby találja fel a róla elnevezett, mai napig használatos hangrendszert.

1959-ben az MIT-en megalkotják a TX-2 számítógépet, mely grafikus konzollal volt ellátva.

1.9. ábra. *Az első képernyő és fényceruza*

1959-ben dr. Julesz Béla (1928–2003) megalkotta az első véletlenpont sztereogramot (RDS – *Random Dot Stereogram*).

1960-ban megjelenik a DEC PDP-1 számítógép. John Whitney (1917–1995) megalapította a *Motion Graphics, Inc.* animációs céget. A számítógépes animáció atyjaként tartjuk számon.

1961-ben jelent meg az első számítógépes játék. A *Spacewar!*-t Steve Russell (1937–) programozta le az MIT-nél egy PDP-1-es gépen.

1963-ban Ivan E. Sutherland (1938–) kifejlesztette a *Sketchpad* rajzoló rendszert, az első valósidejű grafikus rendszert: vektorgrafikus ábrákat lehetett megrajzolni egy fényceruza segítségével. Találmányáért 1988-ban Turing-díjat kapott. A TX-2-es gépre megírt rajzolóprogram legördülő menüket, hierarchikus modellezőrendszert és megkötés-elvű rajzolóalgoritmusokat tartalmazott. Ekkor született meg a számítógépes grafika. Edward Norton Lorenz (1917–2008) meteorológus egy egyszerű időjárási modell felállításával próbálkozott. Amikor a rendszer viselkedését fázistérben ábrázolta, egy igen furcsa attraktor képe bontakozott ki a szemei előtt: megszületett a *Lorenz-attraktor*.

1964-ben alkalmazta a General Motors DAC-1 rendszere az első grafikus konzolt: grafikus parancsokat lehetett bevinni, ezeket értelmezte a rendszer. Ekkor született meg az IBM és a GM közös projektjeként az első CAD (*Computer Aided Design*) tervezőrendszer is. Ugyanekkor jelent meg a RAND grafikus digitalizáló konzolja is, a *Grafacon*, valamint az IBM 2250, az első kereskedelemben forgalmazott grafikus számítógép.

1965-ben jelent meg az első egér: fából és műanyagból készítette Douglas Engelbart (1925–).

1.10. ábra. *Az első egér*

1965-ben vezette be Roberts G. Lawrence (1937–) a homogén koordináták fogalmát [82]. Ekkor publikálta Jack Elton Bresenham (1937–) a híres vonalrajzoló algoritmusát is [10].

1966-ban alkotta meg Ralph H. Baer (1922–) az *Odyssey* játékkonzolt, az első széles körben eladott számítógépes grafika terméket. Erre írta meg híres játékát, a *Pong*ot.

1966–1967-ben alkotta meg Wally Feurzeig (1927–) és Seymour Papert (1928–) a cambridge-i BBN kutatóintézetben a LOGO programozási nyelvet.

1967-ben üzemeltették be a NASA-nál az első színes, valósidejű repülésszimulátort.

1968-ban alakult meg a utahi egyetemen az első számítógépes grafika tanszék, vezetője David C. Evans (1924–1998) volt. Aristid Lindenmayer (1925–1989) magyar származású elméleti biológus és botanikus alkotta meg a róla Lindenmayer-rendszernek, röviden L-Systemnek nevezett formális fraktál leírási módszert.

1969-ben megalakult a Computer Image Corporation és a SIGGRAPH. Alan Kay (1940–) a Xeroxnál megalkotta az első grafikus felhasználói felületet (GUI – *Graphical User Interface*).

1.2.3. A számítógépes grafika elterjedése és fejlődése

A grafikát is támogató számítógépek, operációs rendszerek, programozási nyelvek (pl. *BASIC*, 1964; *LOGO*, 1966; *Pascal*, 1970) széles körű elterjedésével, az 1970-es évektől kezdődően a számítógépes grafika széles körű felhasználásnak örvendett, szinte havi gyakorisággal történtek grafikát befolyásoló események. Próbáljuk meg áttekinteni a legkiemelkedőbbeket.

1970-ben jelent meg a Sonic Pen 3D beviteli eszköz. Gary Scott Watkins a utahi egyetemen megvédett doktori dolgozatában a látható felületek meghatározására valósidejű algoritmust mutat be. Pierre Étienne Bézier (1910–1999) megalkotta a *Bézier-görbé*ket.

1971-ben az Addison-Wesley Educational Publishers Inc. kiadónál, 301 oldalon megjelent az első számítógépes grafikával foglalkozó könyv: David M. Prince: *Interactive Graphics for Computer Aided Design*. Az első filmbeli 2D képalkotás is ekkor jelent meg *Az Androméda-törzs* (The Andromeda Strain) c. filmben (Michael Crichton). Szintén ekkor jelent meg a Henri Gouraud (1944–) féle *shading* algoritmus.

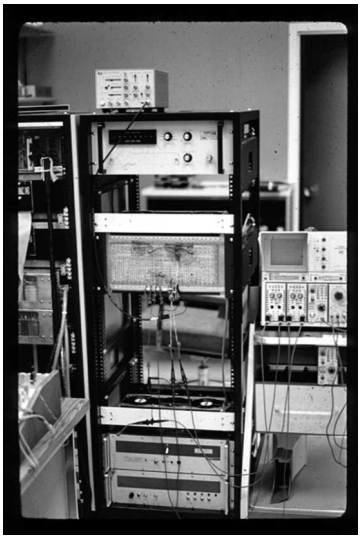
1971-ben alkotta meg Gary Starkweather a Xeroxnál az első lézernyomtatót.

1972–1973-ban a Xerox Palo Alto Research Centernél (PARC) Richard Shoup megtervezte a *SuperPaint* első digitális rajzolórendszert, amely 16,7 millió színt, illetve animációkat, videókat is tudott kezelni.

1972-ben Nolan Bushnell (1943–) megalapította az Atari céget.

Rich Franklin Riesenfeld 1973-ban bevezette a *b-spline görbé*ket [81]. Ekkor jelent meg 640 oldalon a McGraw-Hill Inc. kiadó gondozásában az első átfogó, számítógépes grafikával foglalkozó monográfia: William Newman és Robert L. Sproull: *Principles of Interactive Computer Graphics*. 2D-s CGI-t (*Computer-Generated Imagery*) is először 1973-ban használtak a *Feltámad a vadnyugat* (Westworld) c. filmben (Michael Crichton).

1973-ban a Sharp (Japán) kifejlesztette az LCD (*Liquid Crystal Display*) monitort, azonban az elterjedéséhez 20 év kellett.



1.11. ábra. *A* SuperPaint

1974-ben jelent meg az Edwin Catmull (1945–) által kifejlesztett *z-buffer algoritmus* [15]. A Philips cég elkészítette az első videotelefont. Az első teljesen számítógépes animációval készült 2D film a 11 perces kanadai *The Hunger* (1974) volt. Simonyi Károly (1948–) a Xerox Palo Altoi kutatóközpontjában megalkotta a *Bravo* szövegszerkesztőt, az első WYSIWYG (*What You See Is What You Get*) rendszert, amelyet magyarul ALAKHŰ-nek mondhatnánk (*Azt Látod, Amit Kapsz, HŰen*).

1975-ben jelent meg Benoît B. Mandelbrotnak (1924–) az első *fraktál*lal kapcsolatos cikke, Bui-Toung Phong pedig a megvilágítás számítógépes modelljeiről publikálta a *Phong-shading algoritmus*t [77]. Martin Newell a utahi egyetemen megrajzolta a CGI teáskannát (Utah teapot), a számítógépes grafika „kabala-figuráját", logóját. Bill Gates (1955–) megalapította a Microsoftot.

1976-ban alapította meg Steve Jobs (1955–) és Steve Wozniak (1950–) az Apple-t. Háromdimenziós kép először a *Futureworld*ben (1976) volt látható, ahol egy számítógép által generált kezet és arcot alkotott Edwin Catmull és Fred Parke (utahi egyetem). Joel Orr szerkesztésében megjelent az első számítógépes grafikával foglalkozó folyóirat *Computer Graphics Newsletter* néven (1978-tól *Computer Graphics World* a neve). Megalkották az első tintasugaras nyomtatót, de ez csak 1988-tól kezdett elterjedni.

1977-ben kezdődött el a személyi számítógépek korszaka. A Matsushita bevezeti a VHS formátumot (*Video Home System*). Az első film, amelyben 3D számítógépes animációt használtak, a *Csillagok háborúja* (1977) volt, ahol a Halálcsillag tervrajzai követelték a beavatkozást. Frank Crow megalkotta az élsimító *antialiasing algoritmus*t [18]. Az Oscar-díjaknál külön kategóriát képezett a vizuális effektusok díjazása. Megjelent az Atari Video Computer System (VCS) játékkonzol (Atari 2600).

1978-ban James F. Blinn bevezette a *Bump mapping technikát*.

1980-ban alakult meg az EUROGRAPHICS (*The European Association for Computer Graphics*), és Genfben megtartották első konferenciájukat. Turner Whitted megalkotta a sugárkövető (*Ray-Tracing*) algoritmust.

1981-ben a Penguin Software (most Polarware) bevezette a *Complete Graphics System*et. A Sony Corporation megalkotta a *Mavicá*t, az első digitális fényképezőgépet.

1982-ben James H. Clark (1944–) megalapította a Silicon Graphics Inc. céget, John Warnock (1940–) pedig az Adobe-ot. Létrejött az AutoDesk, és piacra dobták az első *AutoCAD*-ot. Tom Brigham megalkotta a *morphing*ot. Az első CGI karakter az 1982-ben bemutatott *Tron* c. filmbeli *Bit* volt (egy poliéder). Az animációs szoftvert Bill Kovács (1949–2006) készítette.

1983-ban alkotta meg Steve Dompier a *Micro Illustrator*t. Az AutoDesk a piacra dobta az első PC-kre szánt CAD programot. Williams Lance bevezette a textúrázás *mip-mapping* technikáját [101]. A Sony és a Philips megjelentette az első CD-lejátszót.

1.12. ábra. *Bit, az első CGI-karakter*

1984-ben a Robert Able & Associates bemutatta az első számítógéppel generált 30 perces Super Bowl reklámot. Eladták az első Macintosh számítógépeket. A Cornell Egyetemen megszületik a *radiosity*.

Az első ember alakú CGI karakter 1985-ban jelent meg a *Sherlock Holmes és a félelem piramisa* (Young Sherlock Holmes) c. filmben (John Lasseter). A karakter egy festett üvegablakból összeállt lovag formájában jelent meg a vásznon. Ken Perlin bevezette a róla elnevezett zajfüggvényeket [76]. Michael Cowpland (1943–) megalapította a Corel céget.

1986-ban megalakult a Pixar stúdió. Az MIT Athena-projektje keretén belül létrejött az *X-Window rendszer*.

1987-ben szabványosították a GIF és JPEG képformátumokat. Megjelent az Adobe Illustrator. Az IBM megalkotja a VGA (Video Graphic Array) kártyát és megjelenik az IBM 8514. Az Apple létrehozta a *TrueType* fontokat.

A Disney és a Pixar 1988-ban megalkotja a CAPS rendszert (*Computer Animation Paint System*).

1989-ben jelent meg az Adobe Photoshop. A Pixar elkezdi megírni a máig is használt *RenderMan* animációs szoftverét. *A mélység titka* (The Abyss) elnyerte a legjobb vizuális effektusokért járó Oscar-díjat, a vízlény fotorealisztikus CGI karakter volt. Megjelent az első Corel Draw verzió.

1990-ben a DOS grafikus felületeként megjelent a Windows 3.1, az AutoDesk megjelentette a *3D Studio*t. John Wiley & Sons elkezdi kiadni a *The Journal of Visualization and Computer Animation*t.

A CGI 1991-ben a James Cameron rendezte *Terminátor 2*-ben kapott központi szerepet, ahol a T-1000-es terminátor folyékony fém-mivoltával és alakváltó effektusaival káprázta el a közönséget. A *Terminátor 2* szintén meghozta az ILM-nek az Oscar-díjat a különleges hatásokért. Ekkor jelentek meg az SGI Indigo gépek is.

1992-ben jelentette meg az Apple a *QuickTime*-ot. Az SGI megjelentette az *OpenGL* első verzióját. Az OpenGL platform- és operációs rendszer független

grafikus API. Jelenlegi verziója az 1.5-ös. A projekt annyira sikeresnek bizonyult, hogy a Microsoft is beállt az OpenGL fejlesztésébe. A függvénykönyvtár pár száz alacsony szintű rutinból áll, amelyek által nagyon jól ki lehet használni a hardvereket – több hardverkészítő is már beépítette ezeket a rutinokat hardver szinten. Az OpenGL nem tartalmaz komplex formákat, alakzatokat stb., csak a legegyszerűbb elemeket: pontot (vertexet), vonalat, poligonokat. A programozó kell ezekből felépítse a saját komplex formáit. Az OpenGL alacsony szintű függvényeket magas szintű utility könyvtárak támogatják (pl. GLU, GLUT), ezeknek a feladata az ablakozó rendszer kezelése, a magasabb szintű objektumok (kocka, gömb, kúp, henger, görbék, felületek stb.) kialakítása és megjelenítése. Az OpenGL funkciói: színtér definiálása; nézőpont specifikálása; megvilágítási modellek alkalmazása; a megvilágított színtérről árnyalt modell készítése; árnyalások és textúrák alkalmazása; antialiasing (élsimítás); motion blur (mozgó objektumok körvonalainak elmosása); atmoszféra-effektusok kezelése (pl.: köd); animáció. A Hewlett-Packard (HP) megalkotta a népszerű LaserJet4-et, az első 600×600 dpi felbontású lézernyomtatót.

1993-ban jelent meg az Adobe Acrobat, Windows NT, Doom. Az 1993-as *Jurassic Park* dinóinak életszerű megjelenése, mely hibátlanul ötvözte a CGI-t és a live-actiont, hozta meg a filmipar forradalmát. E pont jelentette Hollywood áttérését a stop-motion animációról és a hagyományos optikai effektusokról a digitális technikákra.

1994-ben Mark Pesce (1962–) megteremti a *virtuális valóság* fogalmát és megalkotja a VRML-t. A CGI-t hasznosították a *Forrest Gump* különleges hatásainak megalkotására. A leginkább megjegyzendő trükk a filmben Gary Sinise színész lábainak digitális módon történő eltávolítása volt, vagy a napalmtámadás, a gyorsan mozgó pingponglabdák és a madártoll a nyitójelenetben.

1995-ben az első teljes egészében számítógép alkotta mozifilm, a Pixar cég és a Walt Disney produkciója, a *Toy Story* zajos sikereket ért el. CGI a filmekben általában 1.4-6 megapixellel renderelt. A *Toy Story*t például 1536×922 (1.42MP)-vel renderelték. Egy képkocka renderelése jellemzően 2–3 óra körüli időt vesz igénybe, a legbonyolultabb jeleneteknél ennek tízszerese is előfordulhat. Ez nem sokat változott az utóbbi évtizedben, mert a képminőség azonos szinten halad előre a hardverfejlődéssel, mivel gyorsabb gépekkel egyre összetettebb megvalósítás válik lehetővé. A GPU feldolgozási erejének exponenciális növekedése, illetve a CPU erejének, tárolási kapacitásának és memóriasebességének és -méretének jelentős emelkedése rendkívül kiszélesítette a CGI lehetőségeit. Megalakult a DreamWorks SKG (Steven Spielberg, Jeffrey Katzenberg és David Geffen). Ekkor jelent meg az MP3 szabvány és a Sony Playstation. A Microsoft megjelentette a *DirectX* első verzióját. Arra volt tervezve, hogy a különböző típusú kártyákat, drivereket egységesítse, illetve hogy direkt hozzáférést biztosítson a hardverhez. Az OpenGL-lel ellentétben a DirectX nemcsak grafikát tud kezelni, hanem más multimédiás lehetőségei is vannak, például a hangkártya programozása vagy a hálózatkezelés.

1996-ban megjelent a Windows 95 grafikus felülettel rendelkező operációs rendszer, valamint az SGI O2-es gépei.

1997-ben jelent meg a Flash 1.0-ás verziója, a DVD technológia, és az IBM Deep Blue gépe először vert meg profi sakkozót.

1998-ban jelent meg a *Maya*, vált szabvánnyá az XML, az MPEG-4, és a *Titanic* megdöntött majdnem minden filmes rekordot.

1999-ben jelent meg a *Csillagok háborúja* első része, amely 66 digitális karaktert használt.

2000-ben jelent meg a Playstation 2, a Microsoft X-Box, a Mc OS-X, valamint a *Maya* Macintosh gépekre.

2001-ben jelent meg a Windows XP. A Square Pictures megalkotta a *Final Fantasy – A harc szelleme* című CGI-filmet, amely magas szinten részletezett és fényképminőségű grafikát vonultatott fel. *Gollam* karaktere *A Gyűrűk Ura*-trilógiából teljes egészében CGI-vel készült, motion capture segítségével.

2003-ban jelent meg az Apple Power Mac G5.

2008 júniusában az AMD bejelentette az 1 teraflops teljesítményű *ATi Radeon HD 4870* videokártyát. Jellemzői: 512 MB GDDR5 memória; 1,2 teraflops teljesítmény; 750 MHz GPU; PCI Express 2.0 interface; 160 W.

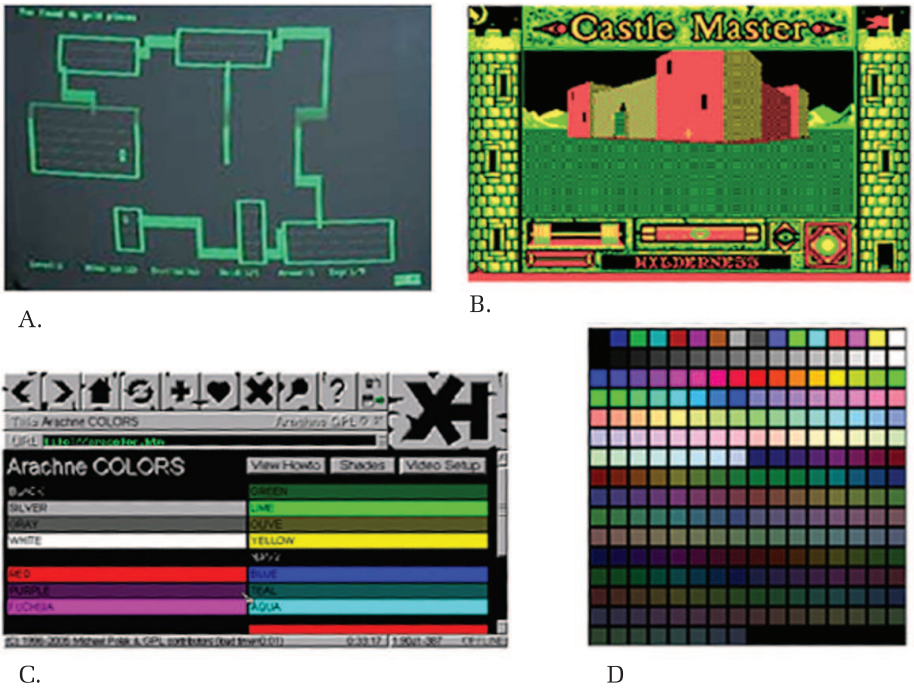
Ha végigtekintünk a számítógépes grafika történetén, következtetésként elmondhatjuk, hogy a grafika fejlődését eleinte a konzol játékgépek, a személyi számítógépes játékok és a filmipar igényelték. 1980 körül a PC-k nagy elterjedésnek kezdtek örvendeni, megjelent a beépített raszter grafika (IBM, APPLE), bit-térképek (bitmap, pixel alapú), desktop-felületek, ablakkezelő rendszerek.

A 80-as évek eleje: a felbontás 320×200 pixel, a használható színek száma 4, amelyet 16 alapszínből lehet kiválasztani. Megjelent a CGA videokártya. A videomemória nagysága kb. 64 KB volt.

A 80-as évek közepére-végére megjelentek az EGA videokártyák max. 256 KB memóriával. Felbontásuk 640×480 pixel 64 szín használatával. Emellett teret hódítottak a Hercules kártyák a hozzájuk tartozó monokróm monitorokkal, ugyanis a színes monitorok abban az időben nagyon drágák voltak. A Hercules kártyák nagyobb (758×512) felbontást nyújtottak, de csak fekete-fehér (vagy zöld, narancssárga monokróm) grafika mellett. Megjelentek a különféle emulációk az egyes működési módok között.

A 90-es évek elején jelentek meg a VGA kártyák 256 KB memóriától egészen 4 MB kivitelig. A 640×480-as működési módot minimum teljesítették, azonban a több memóriával rendelkező darabok akár egészen a 2048×1536-os felbontást is tudták kezelni. Itt jelent meg először a 65 536 színű (16 bites) üzemmód, majd később a 16,7 millió színű (24 bites) ábrázolás. Látható, hogy a felbontás és a pixelenként tárolt egyre több színinformáció egyre nagyobb memóriát igényel.

A 90-es évek végére megjelentek a 3D gyorsítást végző modellek. Napjainkban memóriájuk 4 MB-tól 512 MB-ig terjed. Kezdetben csak célfeladatokat gyorsítottak, azonban manapság külön programozható a videokártyák GPU-ja *shader* programok segítségével.



1.13. ábra. *A. Grafika AT&T PC6300 üzemmódban, B. Grafika CGA üzemmódban, C. Grafika EGA üzemmódban, D. Grafika VGA üzemmódban*

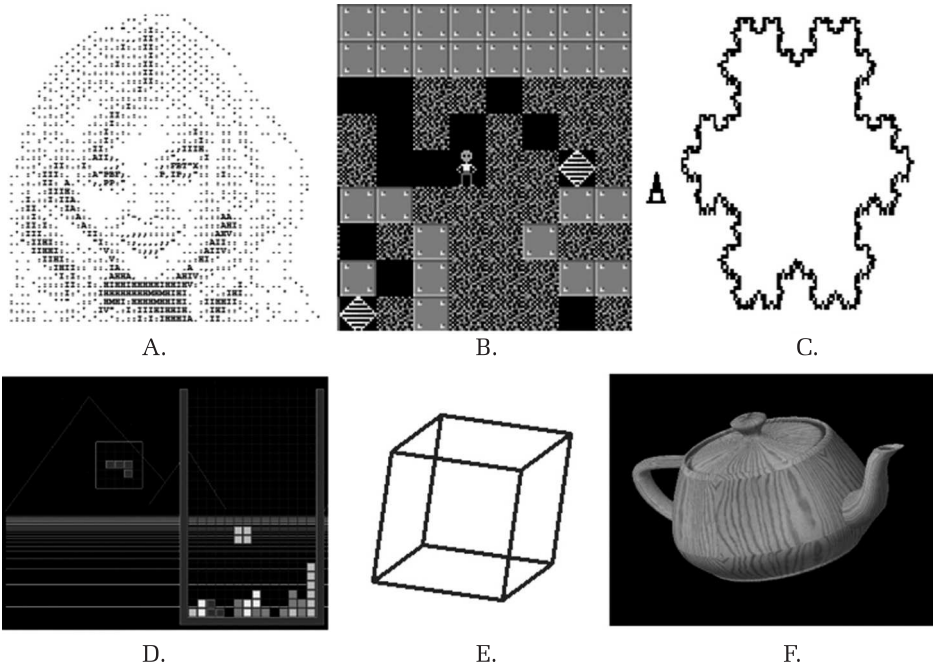
Meg kell jegyeznünk azt, hogy habár a személyi számítógépek hatalmasat fejlődtek számítógépes grafika tekintetében is (manapság valósidejű animáció, filmvágás, házimozirendszerek is jól működnek PC-ken), komolyabb (pl. orvosi, tervezési) feladatokhoz a mai napig célszámítógépeket használnak.

Ha a grafikus rendszerek fejlődését próbáljuk nyomon követni (mind pixelgrafika, mind vektorgrafika területén) – például programozás, grafikus könyvtárak használatának szemszögéből, akkor a következő nagy rendszereket sorolhatjuk fel:

- Rajzolás szöveges karakterek segítségével, vagy karakterek átdefiniálása szöveges üzemmódban (1.14. ábra A és B);
- Teknőc (Turtle) grafika (1.14. ábra C);
- Geometrikus BGI grafika (1.14. ábra D);
- Windowsos grafika (GDI) (1.14. ábra E);
- OpenGL (1.14. ábra F);
- DirectX.

A legegyszerűbb grafika a személyi számítógépek karakteres (szöveges) üzemmódját használta ki. Átdefiniálta a memóriában lévő karaktertömböt, és

oda bármilyen grafikus ábrát be tudott tenni (pl. egy téglás fal képe), ezután egy egyszerű kiíratással nem a karakter képe (pl. 'A') jelent meg, hanem az átdefiniált, megrajzolt ábra.



1.14. ábra. *A. Karakterekből kirakott ábra DOS szöveges üzemmódban, B. Karakterek átdefiniálása DOS szöveges üzemmódban, C. Fraktál (Koch-pehely) képe* LOGO *teknőc grafikával DOS grafikus üzemmódban, D. BGI grafika DOS grafikus üzemmódban, E. Kocka képe Windows alatti GDI grafikával, F. A Utah Teapot textúrás képe OpenGL-ben*

A *LOGO* nyelvből jól ismert *teknőc grafika* már grafikus üzemmódot használt. Parancsai előre, hátra, jobbra, balra való mozgatást, valamint forgatásokat tudtak elérni. A koordináták a képernyő középpontjához relatívak. A felhasználható grafikus üzemmódok: 320×200, 640×200 (fekete-fehér, 16 szín), a függvénygyűjtemény mintegy 25 rutint tartalmaz.

A *DOS-geometrikus BGI grafika* közel 80 rutint tartalmazó grafikus gyűjtemény, mely a bitműveletektől egészen a magas szintű funkciókig mindenféle rutint tartalmaz. A grafikus üzemmódot egy vagy több grafikus meghajtó (pl. .BGI állományok *Borland Graphic Interface*) segítségével tudja kezelni a rendszer. Amilyen meghajtóprogramunk van, olyan felbontást és színhasználatot

lehet elérni. A rendszer parancsai köröket, téglalapokat, ellipsziseket, vonalakat meg hasonló geometrikus primitíveket tudnak kirajzolni. A koordináták a képernyő bal felső sarkához relatívak.

A *GDI* (Graphic Device Interface) grafika szintén saját – de jóval fejlettebb – meghajtóprogramokon keresztül tud vektor- vagy pixelgrafikus ábrákat megjeleníteni. A többszáz függvényt tartalmazó könyvtár A GDI eszközvezérlő programokon keresztül kezeli a grafikus perifériákat és ezáltal lehetővé teszi, hogy a rajzgépet, a nyomtatót, a képernyőt egységesen használjuk. A GDI programozásakor bármilyen hard eszközt, meghajtót figyelmen kívül hagyhatunk. A színek használata is úgy van megoldva, hogy nem kell foglalkoznunk a konkrét fizikai keveréssel és kialakítással. A *TrueType* fontok használata biztosítja azt, hogy a megtervezett szöveg nyomtatásban is ugyanolyan lesz, mint ahogy azt a képernyőn láttuk. A GDI nagy előnye az is, hogy saját koordináta-rendszerrel dolgozhatunk, virtuális távolságokkal írhatjuk meg, a konkrét hardvertől függetlenül, az alkalmazásunkat. Azonban a GDI továbbra is kétdimenziós, egészkoordinátájú grafikus rendszer maradt. A GDI nem támogatja az animációt. A GDI filozófiának az alapja az, hogy először meghatározunk egy eszközleírót, amely a fizikai eszközzel való kapcsolatot rögzíti. Ez tulajdonképpen egy rajzeszközhalmaz és egy sor adat kapcsolata. Az adatokkal megadhatjuk a rajzolás módját. Ezután ezt az eszközleírót használva specifikálhatjuk azt az eszközt, amelyen rajzolni szeretnénk. Például ha egy szöveget szeretnénk megjelentetni a képernyőn, akkor először rögzítjük az eszközkapcsolat révén a karakterkészletet, a színt, a karakterek nagyságát, típusát, azután pedig specifikáljuk a kiírás helyét (x és y koordinátáit), illetve a kiírandó szöveget. A rendszernek van alapértelmezett saját eszköze (rajzvászon, toll, ecset, font, bittérkép stb.). Ha mást szeretnénk használni, akkor létrehozunk magunknak egyet, elvesszük a rendszertől az övét (megőrizzük), átadjuk a miénket, hogy azzal dolgozzon a rendszer, a végén pedig ismét cserélünk.

1.2.4. Mit várunk el a jövőtől?

Jósolni nehéz, és a számítástechnika története azt mutatja, hogy a rohamos, gyors fejlődés bárhová vezethet. Számítógépes grafika területén több irányvonal mentén is el tudjuk képzelni a közeljövőt, amely bekövetkezhet hónapokon, de éveken belül is.

A cél nyilvánvalóan a valósidejű, széles körű felhasználásnak örvendő 3D grafika és képalkotás. A felhasználóknak szükségük van arra, hogy egyszerű parancsok segítségével, interaktívan, gyorsan és nagyon egyszerűen szintetizálni, vizualizálni tudjanak gondolatokat, elképzeléseket, a számítógépes grafika nonverbális kommunikációkat közvetítsen. Az ember–gép kapcsolat perifiriáit tovább kell fejleszteni oly módon, hogy a virtuális valóság az ember összes érzékszervére képes legyen hatni.

A világháló gyors elterjedése motiválja a virtuális valóságmodellek fejlődését. Egy hatalmas osztott virtuális hallható és látható világot kell megteremteni, amelynek a web az egyik alapja. Ez nemcsak a játék kedvéért, hanem a problémamegoldás és szimulációk elvégzése érdekében is szükséges.

A számítógépes grafika lehetőségei az orvostudományok számára is fontosak, a különböző letapogató és diagnosztizáló rendszerektől kezdve el egészen például a romlott látást megsegítő kamerákig, amelyek direkt a retinára vetítenek képet.

A 3D televízió, képernyő és mozi megjelenése elterjedése várható fejlemény. Emellett szükség van olyan rendszerekre, amelyek 3D modelleket tudnak interaktívan elkészíteni, így számítógépes segítséggel 3D bemutatókat majdnem mindenki alkothat.

Az animáció tökéletesítése a filmipar nagy kihívása. Robotok virtuális manipulálása, az emberi karakterek valósidejű hű ábrázolása mind megoldandó feladatok.

Algoritmusok terén a valósidejű globális fényhatás-számítások, valósidejű *radiosity* és sugárkövető algoritmusok, különböző effektusok leprogramozása a közeljövő kihívásai.

Animáció, szimuláció, szintetizálás, valósághű ábrázolás és megjelenítés, egyszerű és mindenki számára elérhető 3D grafika a közeljövő kulcsszavai [59].

1.3. A látás

A körülöttünk zajló világról öt érzékszervünk által szerzünk tudomást, azonban az információk legnagyobb részét, több mint 90%-át, a látás során a szemünktől kapjuk.

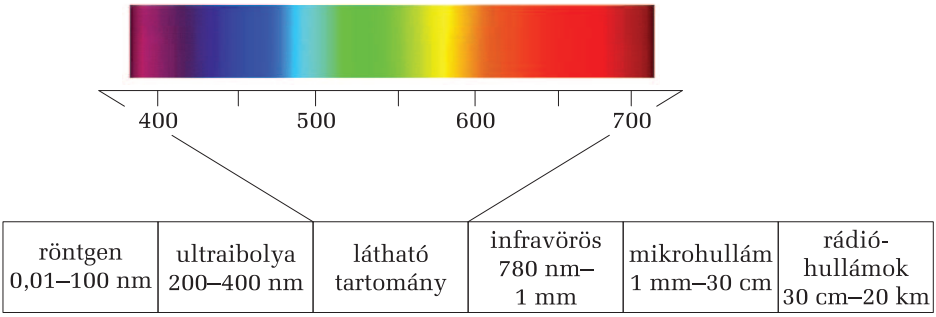
A *látás* a vizuális információk feldolgozása, amelynek fő célja a tárgyak azonosítása és azok közvetlenül nem észlelhető tulajdonságainak felismerése, illetve a cselekvés vezérlése.

A vizuális információk a *fény* segítségével terjednek, érik el az emberi *szem*et, a látás receptorát.

Az ember számára a fény az elektromágneses sugárzásnak az a része, amelyet a szem érzékelni képes és amelynek a hatására az agyban képérzet alakul ki. Ez a rész a hullámhossztartomány kb. 380 nm–780 nm közötti intervalluma.

Látószervünk a szem. Az emberi szemben kb. 126 millió fényérzékelő receptor található, melyek felfogják az elektromágneses sugárzást. A szemben található kb. 1 millió idegszál a keletkezett ingerületet az agyba továbbítja.

A szem *optikai rendszer*e (pl. szemlencse) a beeső fény alapján egy képet vetít a *retiná*ra, ahol a fény különböző kémiai és elektromos reakciókat indít be. A kémiai reakciókért felelős anyagot *fotopigmens*nek nevezzük. A retinában kb. 6 millió *csap* és kb. 120 millió *pálcika* található. Miután a kémiai reakció



1.15. ábra. *Elektromágneses hullámok, a fény hullámhossztartománya*

beindult, a pálcikák és a csapok „üzennek” az agynak, hogy „ehhez a sejthez fény érkezett”. A pálcikák a fény erősségét vagy világosságát érzékelik, a csapok pedig a színlátásban játszanak fontos szerepet.

A színeket az S, L, M *típusú* színérzékelő csapok különböző erősségű ingerlése alapján látjuk, éspedig:

– a kék-sárga árnyalatokat az S-csap, L-csap+M-csap segítségével,
– a piros-zöld árnyalatokat az L-csap és M-csap segítségével.

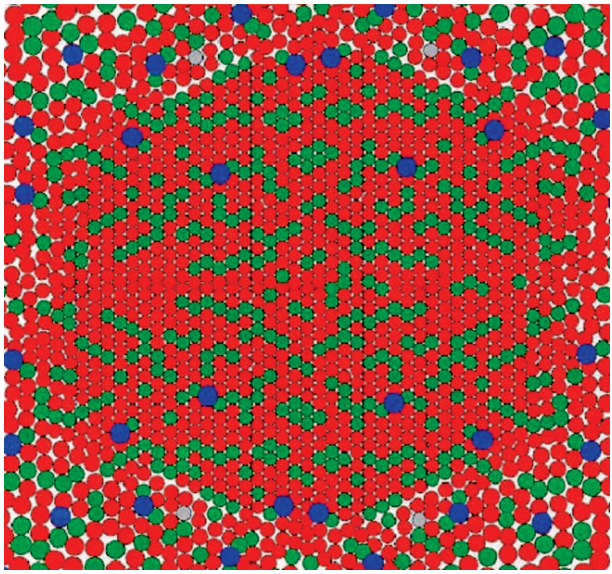
Az L, M, és S csapok eloszlása (1.16. ábra) a 40:20:1 arányt követi, az S érzékenységi tartománya kb. 400–500 nm, az M-é 450–630 nm, az L-é pedig 500–700 nm.

Az emberi szem kb. 200 színárnyalat eltérését képes megkülönböztetni, ez függ a hullámhossztól, legnagyobb érzékenység az 555 nm körül (zöld szín közelében) mutatható ki, és ez jelentősen csökken, ahogyan a látható színtartomány szélei felé haladunk.

Amennyiben a teljes spektrumban egyenletes energiával sugároz egy fényforrás, akkor a háromfajta csap ingerületi állapota azonos lesz. Ezt a fényt nevezzük *akromatikus fény*nek és a színérzete *fehér* lesz. Azt mondhatjuk tehát, hogy a fehér az összes szín jelenlétét jelenti, a *fekete* pedig az összes szín hiányát.

A *világosság*nak vagy *fényerősség*nek is nagy szerepe van. A szemünkbe érkező fényenergia mennyisége meghatározza, hogy mennyire megfelelően érzékeljük a színeket. Az emberi szem nem érzékeli a 10^{-6} lumen alatti fényt, a 10^{4} lumen fölötti pedig elvakít. Világosság terén a szemünk mintegy 50 fokozatot tud megkülönböztetni. Sötétben (ha nagyon kicsi a fényerősség) csak fekete-fehéren látunk, nem érzékeljük a színeket.

A színlátást a *színtelítettség* is befolyásolja. A színtelítettség a szín fehérrel való felhígítottságának, fátyolosságának mértéke. A monokromatikus színek nem tartalmaznak fehér összetevőt, így ezek 100%-os telítettségűek. Például



1.16. ábra. *A csapok eloszlása a retinán*

1.17. ábra. *Fényerősség*

1.18. ábra. *Vörös–rózsaszín átmenet színtelítettséggel*

ha a vörös szín telítettségét csökkentjük (keverjük fehérrel), ez fokozatosan átmegy rózsaszínbe. Szemünk egy színen belül kb. 20 telítettségi fokozatot tud megkülönböztetni.

Összefoglalva, az ember színlátásában a következő tényezők játszanak szerepet [12]:

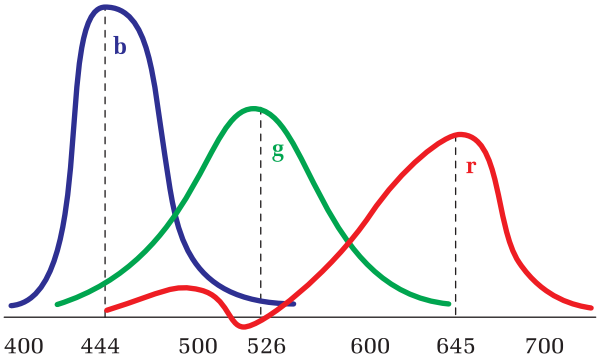
- szín (színárnyalat vagy színezet – *hue*), a szemünkbe jutó fény hullámhosszától függ;
- fényerősség (*brightness*), a szemünkbe érkező fényenergia mennyisége;
- színtelítettség (*saturation*), az érzékelt fényben megtalálható fehér fény százalékos összetevője.

1.3.1. A színlátás matematikai modellje

Az emberi szem a beérkező fényt három különböző, ám kissé átlapolódó tartományban összegzi, ezért az agyban kialakuló színérzet három skalárral is megadható. Három komponensből nem tudjuk kikeverni a természetben előforduló összes színt, viszont a kikevert szín az agyban ugyanazt az érzetet kelti, mintha a szem egy természetes színt látott volna meg. Így gyakorlatilag három összetevő segítségével az összes szín érzetét elő tudjuk állítani az agyban, vagyis például a monitoron nem szükséges a számított spektrumot visszaadni, csupán egy olyant kell találni, amely az agyban ugyanazt a színérzetet kelti.

Ezt a folyamatot nevezzük *színleképzés*nek (*tone mapping*) vagy *színillesztés*nek (*color matching*).

Az előbbiek alapján a lehetséges színészletek elképzelhetők egy háromdimenziós térben [94], vagyis ki tudunk jelölni egy koordináta-rendszert úgy, hogy kiválasztunk három távoli hullámhosszt, majd megadjuk, hogy három ilyen hullámhosszú fénynyaláb milyen keverékével kelthető az adott érzet.



1.19. ábra. *A csapok érzékenységei RGB alapon*

A komponensek intenzitásait *tristimulus koordináták*nak nevezzük.

A látható fénytartományt figyelembe véve a legkézenfekvőbb, ha a *vörös (red – r)*, *zöld (green – g)* és a *kék (blue – b)* színek hullámhosszait választjuk ki, ezek kellően távol esnek egymástól:

$$\lambda_r = 645\ nm,\ \lambda_g = 526\ nm,\ \lambda_b = 444\ nm.$$

Egy tetszőleges λ hullámhosszú fénynyaláb keltette színérzetet így meg tudunk határozni az $r(\lambda)$, $g(\lambda)$, $b(\lambda)$ *színillesztő függvény*ekkel, vagyis megadjuk, hogy az *RGB* összetevőkből hogyan keverhető ki a fény.

Ha az érzékelt fénynyalábban több hullámhossz is keveredik (a fény nem *monokromatikus*), az R, G, B tristimulus koordinátákat összegként állítjuk elő.

Ha a fényenergia spektrális eloszlása $\Phi(\lambda)$, akkor a megfelelő koordináták:

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

Mivel két függvénynek is lehet ugyanaz az integrálja, két eltérő spektrumhoz is tartozhat ugyanaz a színérzet. Ezeket a spektrumokat *metamer*eknek nevezzük.

1.3.2. A színkeverés alapjai

Az ember ősidők óta törekszik arra, hogy utánozza a természet színeit vagy olyan árnyalatokat állítson elő, amelyek a természetben nem fordulnak elő.

A színkeverés szabályait a *Grassmann-törvények* (1853) írják le:

- Bármely szín kikeveréséhez három független alapszín szükséges és elegendő.
- A színkeverés folytonos. A színérzet a világossággal nem változik.
- A keverékszín színösszetevői csak az alapszínek színösszetevőitől függenek (a spektrális összetétel nem elsődleges fontosságú).

A színkeverési kísérletek eredményeit szabványosított színdiagramok foglalják össze. Néhány színt (például a barnát, khakit stb.) még ezek a diagramok sem tartalmaznak. Ennek az az oka, hogy ezek a színek önmagukban nem léteznek. A barna például egy olyan sárgásvörös keverékszín, amelyet csak bizonyos háttér előtt érzünk barnának.

Létezik négy szín, amelyik kiemelkedik a többi közül: a *vörös*, a *kék*, a *sárga* és a *zöld*. Az első három az ún. *elemi elsődleges szín*ek (a vörös *magenta* árnyalata, a kék *cián* árnyalata, valamint a sárga a szubsztraktív színkeverés alapszínei – CMY).

*Generatív alapszínek*nek nevezzük a *vörös*, a *kék* és a *zöld* színeket (az additív színkeverés alapszínei – RGB), amelyekkel fizikai úton a színek széles sorozatát lehet létrehozni. A számítógépes grafikában, képfeldolgozásban a generatív alapszíneket használjuk. Ezek köré tudjuk csoportosítani az összes többi színt, és ezek azok a színek, amelyeket nem látunk a spektrumban körülöttük elhelyezkedő színek keverékének.

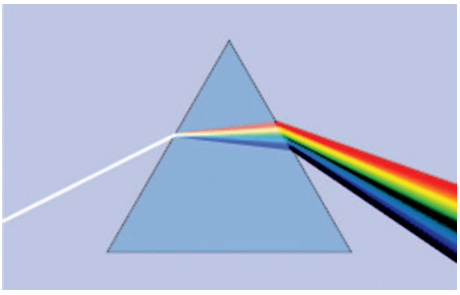
Két szín egymás *komplementere (komplementer szín*ek) vagy *kiegészítő*je, ha keverékük akromatikus színérzetet (rendszerint szürkét) hoz létre. Fizikai értelemben két szín egymás kiegészítője, ha keverékük fehér színt ad vissza.

A *másodlagos színe*ket az elemi elsődleges színek keverésével kapjuk: *zöld*, *narancs* és *lila*.

A *harmadlagos szín*ek az elemi elsődleges és a másodlagos színek keverésével jönnek létre, ilyen szín pl. a barna. Ezeknek fontos szerepük van, amikor a kiegészítő színeket osztjuk meg egy kompozícióban.

1.3.2.1. Additív színkeverés

Az optikai prizma a fehér fényt spektrális színekre bontja fel. Ha ezeket a színeket megfelelő módon összegezzük, újra előállíthatjuk a fehér fényt.



1.20. ábra. *Az optikai prizma*

A színelmélet szerint a szem három különböző típusú színreceptorának gerjesztésével gyakorlatilag bármely szín érzékelhető.

A színkeverés elméletével már Newton is foglalkozott. Maxwell és Helmholtz állapította meg (1860), hogy megfelelően megválasztott 3 szín adott arányú összegzésével bármilyen mintaszín (színérzet) kikeverhető:

$$\text{Szín} = a \cdot R + b \cdot G + c \cdot B,$$

ahol R a vörös, G a zöld és B a kék színeket jelöli, illetve az a, b, c együtthatók ezek arányát.

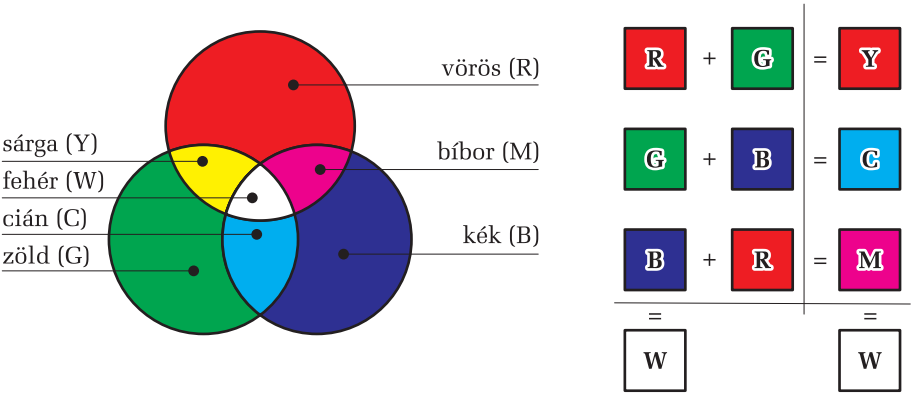
Az additív színkeverés során tehát különböző hullámhosszúságú fények együtt, egymással összeadódva érik el szemünket. Fontos az, hogy itt csak egy pszichofizikai jelenségről van szó, az összeadás csak a szemünkben jön létre.

Ez háromféleképpen történhet:

- színes fénynyalábok összeadásával,
- színes tárcsa forgatásával,
- raszterpontok segítségével.

Az összeadó színkeverés során, ha színes fénynyalábokat használunk, a két szín összeadásából létrejövő harmadik mindig világosabb lesz, mint a kiinduló színek bármelyike. A színes tárcsa forgatásával vagy raszterpontok összeadódásával létrejövő szín világosságértéke azonban az eredeti színek átlaga lesz. A három különböző szín összeadásával keletkezett új szín lehet tetszőleges, vagy lehet akár a fehér is.

Ilyen elven működik a monitor és a színes televízió, amely a vörös, zöld, kék (*RGB*) színrendszert használja.



1.21. ábra. *RGB – az additív színkeverés*

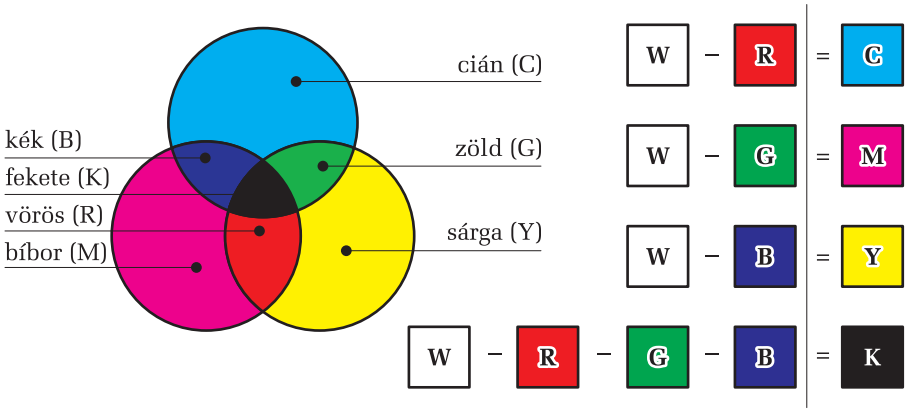
1.3.2.2. Szubsztraktív színkeverés

A festészeti, fotográfiai ismeretekkel rendelkezők bizonyára kételkednek a színek előállításának előbb ismertetett módjában. A festészetben ugyanis pl. a *sárga* alapszínnek számít, és a zöld és a vörös keveréke sohasem sárga, hanem sötétbarna vagy fekete lesz. Ez esetben azonban nem additív, hanem úgynevezett *szubsztraktív* vagy *kivonó színkeverés*ről van szó, amelyre másfajta szabályok érvényesek. A sárga festék pl. elnyeli a kék fényt és visszaveri a zöldet és a pirosat, ezért látjuk sárgának. A sárga színszűrő is hasonlóan működik: elnyeli a kék színt, átengedi a zöldet és a pirosat. A videokamerák technológiájában ezt a tulajdonságot kitűnően fel is használják.

A szubsztraktív vagy kivonó színkeverés fizikai jelenség és többféleképpen valósulhat meg: a fényforrás elé rakott színes szűrőkkel, vagy festékanyagok (*pigmensek* – a természetben előforduló festékanyagok) keverésével. A kivonás a legtöbbször magától is megtörténik, amikor a fényforrás fénye a tárgyakról visszaverődik, vagy rajtuk áthalad. A tárgyak ugyanis a rájuk eső fényt, illetve annak bizonyos összetevőit részben vagy egészben visszaverik, vagy elnyelik, vagy átengedik – anyaguktól függően. Emiatt látjuk színesnek a világot.

A festékek és a színes szűrők a teljes spektrumot tartalmazó fehér fény egy részét elnyelik (kivonják), másik részét áteresztik (szűrők) vagy visszaverik (festékanyagok). A visszavert sugárzás spektrális eloszlása adja meg a létrejövő színérzetet.

Egy felület vagy anyag színe nem más, mint annak a fénynek a hullámhossza, amelyet a felület vagy anyag visszaver. Ha minden fényt elnyel, akkor fekete, ha minden fényt visszaver, akkor fehér lesz.



1.22. ábra. *CMYK – a szubsztraktív színkeverés*

A szűrők kombinálásával vagy a pigmensek keverésével hozhatunk létre új árnyalatokat. Mindkét esetben a kialakuló új szín az alapszíneknél sötétebb lesz. Ha a két festéket összekeverjük, akkor a keverék a kékes és a sárgás színű sugárzásokat is elnyeli, így az általa visszavert sugárzás egy zöldes szín képzetét kelti az ember érzékelőrendszerében.

Szubsztraktív színkeverést alkalmaz a nyomdatechnika (és a színes nyomtatók vagy a festők is) a keverékszínek előállítására.

Az alapszínek a *ciánkék (C – cyan)*, a *bíbor (M – magenta)* és a *sárga (Y – yellow)*. Ez a CMY színmodell.

Mivel a nyomdatechnikában és a gyakorlati élet egyéb területein tiszta fekete szín a festékanyagok tulajdonságai miatt nem állítható elő ilyen módszerrel (csak egy erősen sötét barnát kaphatunk), az alapszíneket kiegészítik a *feketével (K – black)*. Ez a színmodell a CMYK nevet viseli. Mivel a fekete kezdőbetűje *(B – black)* foglalt az RGB modell kék (blue) kezdőbetűje miatt, a szó utolsó betűjét használták fel, ez megegyezik a „*key*” rövidítésével, így nevezték a régi nyomdákban a feketét.

1.3.2.3. A színrebontás

*Színrebontás*nak (*color separation*) nevezzük azt a folyamatot, amikor a színeket alapszínekre bontjuk, vagyis meghatározzuk, hogy minden egyes színben mennyi R, G, B vagy C, M, Y, K komponensmennyiség van.

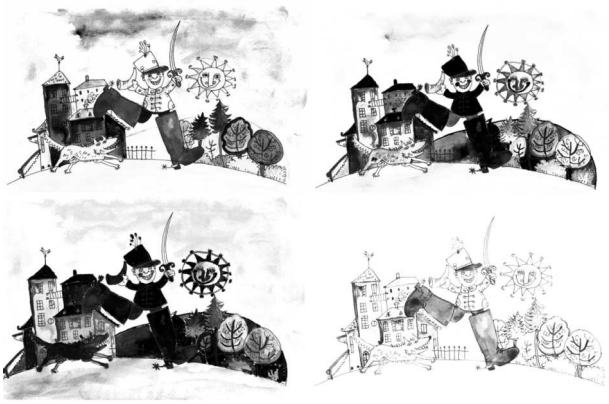
1.23. ábra. *CMYK-módú színes kép (Tomos Tünde rajza)*

A színes képek nyomtatásakor a szubsztraktív színkeverés elvét alkalmazzuk. A nyomdatechnikában nem teljesen a színelméleti alapszíneket veszik alapul, hanem amelyek a legpontosabb gyakorlati színeket eredményezik. Ezek a cián (C), a bíbor (M), a sárga (Y) és egy előre meghatározott szín. Az előre meghatározott szín az esetek túlnyomó többségében a fekete (K). Ritka kivétel az, amikor a kép nem tartalmaz fekete összetevőt, ám egy szín olyan nagy felületet képez, hogy egyszerűbb azt nem a három színből kikeverni, hanem eleve az adott színt használni.

A színrebontás elve a következő: Egy adott képet négy részre bontanak. Ez valójában négy új képet jelent. Az egyes képek úgy jönnek létre, hogy mindegyiket meg lehet feleltetni a CMYK színek egyikének. Miután ezeket a nyomdában egymásra nyomják, újra megkapjuk (most már papíron) az eredeti színes képet. A négy színrebontott kép, mivel csak mennyiségeket jelenítenek meg (a világosság mértékében), elegendő, ha szürke árnyalatban készül el.

1.3.3. Színmodellek, színterek és színmódok

Amikor a végtelen sok színváltozattal akarunk dolgozni, számítógép segítségével kívánjuk kezelni a színeket, először is megfelelő áttekintésre, rendszerezésre van szükségünk. A színeket bizonyos rendszer szerint csoportosítanunk



1.24. ábra. *CMYK-módú színes kép színrebontva a C (bal felső), M (jobb felső), Y (bal alsó), K (jobb alsó) komponensek szerint*

kell ahhoz, hogy tájékozódni tudjunk közöttük. Fel kell állítani egy egzakt, kiszámítható rendszert, amelyben minden színnek saját helye van. Azért van erre szükség, hogy az egyes színeket meg tudjuk határozni, meg tudjuk nevezni, ha mással közölni akarjuk (lehetőleg színminta nélkül); a közöltek alapján előállítani (reprodukálni) lehessen a színeket; végül hogy szabályokat állíthassunk fel a különböző színek együttes alkalmazására, megállapíthassuk, melyek azok a színek, amelyek egymás mellett alkalmazva kellemes hatásúak, harmonikusak és melyek azok, amelyeket egymás szomszédságában nem használhatunk, mert kellemetlen látványt nyújtanak, diszharmonikusak.

A *színmodell* a digitális képeken látható és felhasználható színeket írja le. Mindegyik színmodell (pl. RGB, CMYK vagy HSB) más és más (általában számokon alapuló) módszert alkalmaz a színek leírására.

A *színtér* a színmodell egy változata, amely speciális színárnyalatokkal, színtartománnyal rendelkezik. Például az RGB színmodellen belül több színtér is található: Adobe RGB, sRGB, ProPhoto RGB stb.

Minden eszköznek (pl. képernyőnek vagy nyomtatónak) megvan a maga színtere, és csak annak színtartományában képes a színeket visszaadni. Ha egy kép egyik eszközről a másikra kerül, megváltozhatnak a színei, mert minden eszköz a saját színterének megfelelően értelmezi az RGB vagy a CMYK modell értékeit. Ilyen esetekben színkezelést célszerű alkalmazni annak biztosítására, hogy a legtöbb szín azonos vagy legalábbis hasonló maradjon, így következetesnek tűnjön.

A színeket a jellemző tulajdonságaik alapján csoportosítják. A csoportosítás eredménye a *színrendszer*, amely a felületszínek megjelölésére, besorolására és lehetőleg tökéletes áttekintésére szolgál.

A színeket többféleképpen rendszerezik:

– *Színsor*okat, *színskálá*kat alakítanak ki.
– *Színminta-asztal*t készítenek. Ebben meghatározott rendszer alapján elhelyezett felületi színek találhatók gyűjteményesen (pl. a nyomdaiparban használatos színasztal).
– *Színtest*et, szabványos színekkel kitöltött összetett testeket határoznak meg. A két legismertebb színtest a Munsell-féle színtest, illetve a CIE (*Commission internationale de l'éclairage – Nemzetközi Világítástechnikai Bizottság*) színtestmodell.

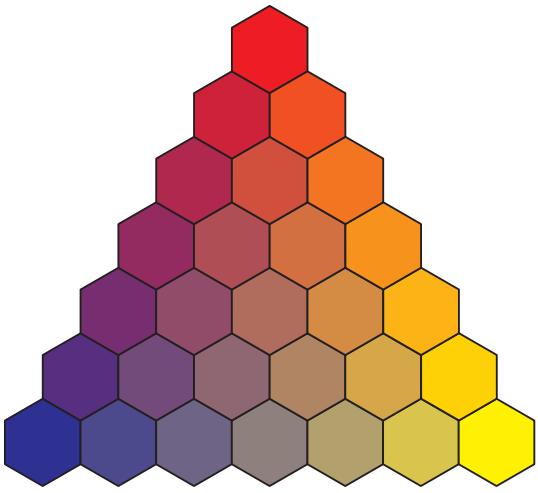
Az alábbiakban tárgyalt színrendszereken kívül természetesen még sok más színrendszer is kialakult, majdnem minden eszköz (pl. színestelevízió-típusok) használhat saját szabványt, mégis az alábbiak – a teljesség igénye nélkül – nagy általánosságukban összefoglalják a legelterjedtebb színrendszereket.

1.3.3.1. Korai színrendszerek

Sigfrid Aronus Forsius (1611) volt az egyik első szerző, aki a színmintákat háromdimenziós modellben ábrázolta, gömbbe rendezett formában. A pólusokon helyezte el a fehéret és a feketét. A sárgát és kéket, illetve a vöröset és a zöldet az egyenlítő átellenes pontjain ábrázolta.

Newton (1672) a színkörében három ún. *elsődleges (primer) szín*t különböztet meg: *vöröset*, *sárgát*, *kéket*; ezek keverékéből származtatja a narancs, a bíbor és a zöld színeket mint *másodlagos szín*eket, majd ismét ezek keveréke adja a barna, szürke és olajzöld, ún. *harmadlagos szín*eket.

Tobias Mayer (1758) háromszögbe rendezte a színeket.



1.25. ábra. *Tobias Mayer színháromszöge*

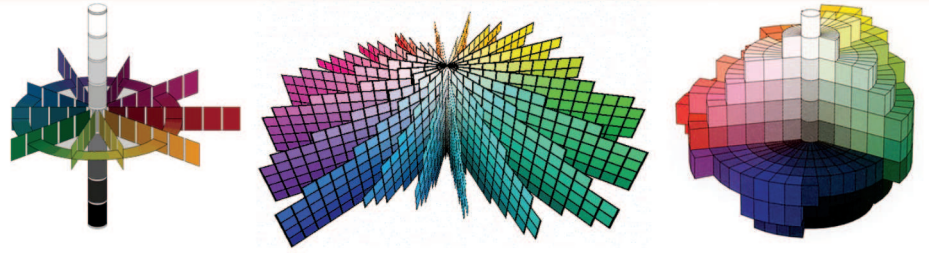
Harris (1766) és Goethe (1810) a síkban ábrázolt színköröket, azonban több neves fizikus, nyomdász és művész térábrázolást készített: a kettős kúpot Ostwald, a hengeres testet Munsell (amire a mai amerikai nyomdai színszabványok épülnek), a színgömböt Runge (1810), Schrödinger a színkúpot alkotta meg. Mindnyájan felismerték, hogy a színek és a színkeverés tudományos vizsgálatához nem elég a színek beosztására a sík, ki kell lépni a térbe.

Schopenhauer (1816) a három elsődleges és a három másodlagos színt a színkörében olyan nagyságú felületen szerepelteti, ahogy azok fényerő szerint egymást kiegyenlítik: az ún. *meleg*, igen feltűnő színek kisebb felületei itt egyenértékűek a nagyobb felületen szereplő, ún. *hideg*, kevésbé élénk színekkel, így ha valamely színcsoportosítással harmonikus hatást akarunk elérni, akkor a szemben fekvőket alkalmazhatjuk, de a színek területének is a színkörön megadott arányban kell állnia.

1.3.3.2. A Munsell-színrendszer

Az egyik legszemléletesebb színrendszer kidolgozása Albert Henry Munsell (1859–1918), amerikai festőművész nevéhez fűződik. Munsell 1915-ben dolgozta ki színmodelljét (*Atlas of the Munsell color system*), amelyben a *színérzet (H – Hue)*, *világosság (V – Value)* és *telítettség (C – Chroma)* paraméterek szerint, háromdimenziós rendszerben helyezi el az összes létező színt. A rendszer függőleges tengelyén, legbelül találhatók az akromatikus színek, azaz szürkék, mégpedig fentről lefelé sötétedve (V). A tengelytől kifelé a szín egyre telítettebbé válik (C). A színezetet a vízszintesen elhelyezkedő körön való elhelyezkedés adja meg (H). A Munsell-féle színrendszerben az egymás mellett szereplő színek látszólagos eltérése nagyjából azonos.

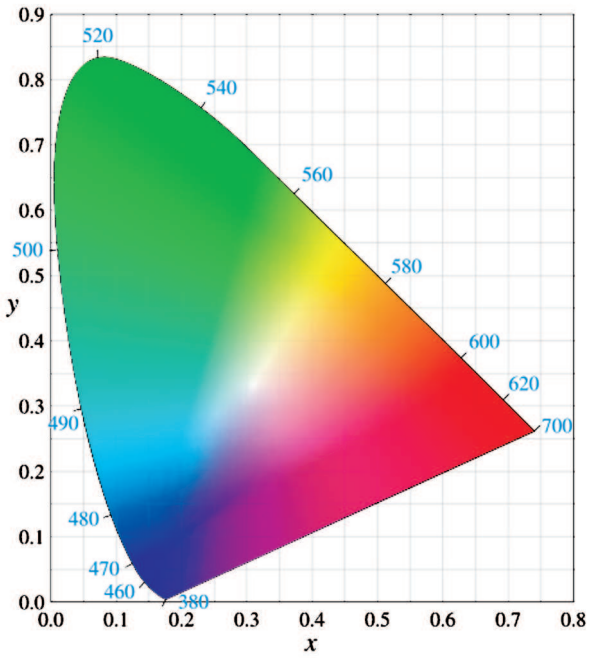
Ilyenformán a Munsell-féle színfán az azonos oszlopban elhelyezkedő színek egyformán telítettek és egyforma színezetűek, de világosságuk eltérő; a vízszintes síkban a tengelytől kifelé a színek azonos világosságúak, de telítettségük nő, míg ugyanebben a sorban a tengely másik oldalán ennek a színnek a komplementerét találjuk.



1.26. ábra. *Munsell-féle színfák John Kopplin nyomán*

1.3.3.3. Az XYZ színrendszer

Ha az RGB-rendszert színingermérő rendszerként használjuk, azt figyelhetjük meg, hogy például az 520 nm hullámhosszú vörös fényt a látóközpont úgy értelmezi, mintha a környezet lenne vörösebb 0,093 értékkel a megfigyelt felületnél, így ezen a hullámhosszon a relatív ingerküszöb –0,093 lenne.



1.27. ábra. *A CIE-1931 színdiagram*

A negatív értékek megjelenése problémákat okozhatnak a színrendszerben, így a CIE 1931-ben bevezette az *XYZ-színrendszer*t, amely kiküszöböli a negatív értékeket. Az XYZ-színrendszer a látható színek pusztán matematikai leírása, hisz az $X(\lambda)$, $Y(\lambda)$, $Z(\lambda)$ színillesztő függvények hullámhosszhoz nem köthetők. Az egyes színeket egy háromdimenziós térvektor határozza meg, amelyet a három spektrális színösszetevő képvisel. A színek térbeli ábrázolása igen körülményes, így kétszeres transzformációval a térgörbéből előállítottak egy gyakorlatban is jól használható színdiagramot, amelyen az x és y koordináták alapján minden szín azonosítható.

1.3.3.4. A YUV, YIQ színrendszerek

Az 1940-es években intenzív kutatások folytak a színes televíziók elkészítése érdekében, emiatt többfajta színrendszert is megalkottak. Az európai TV-sugárzásban a *YUV-színrendszer* használatos, míg az észak- és közép-amerikai, valamint a japán televíziózásban a *YIQ-színrendszer*.

Az Y a *fényességet (luma)*, az U a *kék színkülönbséget (krominancia, chroma)*, a V pedig a *piros színkülönbséget* jelöli. Az IQ komponens szintén krominanciainformációkat hordoz. Az UV (IQ) komponensnek nincs fényességtartalma, csak az adott jel színéről hordoz információt. Ezek a színrendszerek tehát egy fényességinformációt és kért krominanciakomponenst hoznak össze, előnye pedig az, hogy az emberi szem kevésbé érzékeny az UV (IQ) összetevőre, mint az Y összetevőre (így itt kevesebb információt kell átvinni).

Kezdetben külön kódrendszer alakult ki az analóg (*YUV*, *Y'UV*) és a digitális (*YCbCr*, *YPbPr*) kódolás megvalósítására, azonban napjainkban ezek teljesen összemosódnak, sőt a *YUV-rendszer* a PAL, SECAM, NTSC videoszabványok alapja lett, és ilyen elven működik az MPEG és JPEG kódolás is.

1.28. ábra. *A Pantone-skála*

1.3.3.5. A Pantone-skála

A *Pantone-színskála* (1963) egy, a nyomdaiparban általánosan használt szabvány, amely a nyomdai színek gyűjteményét jelenti. Gyakorlatilag a kikevert CMYK-színek szabványos ábrázolását jelenti.

1.3.3.6. A CIELAB színinger tér

A lineáris függvénytranszformációk nem hoztak létre elegendően egyenletes színteret. Az RGB- és CMYK-rendszerek eléggé eszközfüggők. A CIE 1976-ban döntött egy kellően komplex, a szemhez legjobban illeszkedő, eszközfüggetlen színrendszer kidolgozásáról. A *CIELAB-rendszer* már köbgyökös kifejezéseket tartalmaz:

- $L*$ a világossági tényező,
- $a*$ a vörös-zöld színezetre jellemző,
- $b*$ pedig a kékes-sárga színezetre jellemző tényező.

A CIELAB színinger-térben nem értelmezhető a spektrális színek vonala. Úgy kell elképzelnünk, mint két egybevágó, talpával összeillesztett kúpot a térben, az alsó csúcsa a fekete pont ($L*$ = 0), a legszélesebb része az $a*$, vagy $b*$ értékkel jellemezhető telített színeket tartalmazza, majd felfelé újra keskenyedik, és csúcsa a fehér pont ($L*$ = 100), az előjelek:

- $+a*$ piros, $-a*$ zöld,
- $+b*$ sárga, $-b*$ kék.

Az $a*$ pozitív a 430–477 nm közötti kékeknél (zafírkék) és az 578 nm-nél nagyobb hullámhosszú színeknél (a kadmiumsárgától a vörösig), valamint valamennyi bíborszínnél. Az $a*$ negatív a 477–578 nm közötti kék, zöld és sárga színeknél. A $b*$ pozitív az 505 nm (türkizzöld) fölötti színeknél, és a bíborszínek tartományában is az erika-ibolyáig. A $b*$ negatív a bíborszínek említett pontjától a bíbor tartományban, majd a 435–505 nm tartományba eső kékekre és zöldekre.

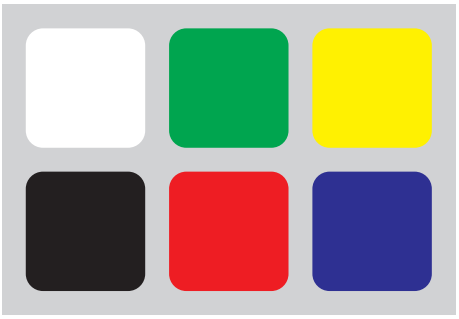
1.3.3.7. Az NCS

Az NCS *(Natural Color System)* a Skandináv Színintézet színmodellje (1979), amely hat színellentét-páron alapszik (fekete–fehér, zöld–vörös, sárga–kék). Minden más szín kevert szín. E 6 színt igen gyakran használják pl. játékok festésére, de a Microsoft Windows logó vagy az olimpiai jelvény is ezekből tevődik össze.

A színeket három érték határozza meg: az *erősség*, a *telítettség* és az alapszínek *százalékos összetétele*. Például a kék alapon sárga keresztet ábrázoló svéd zászlóban megtalálható kék a szabvány szerint NCS 4055-R95B, vagyis 40%-ban sötét, 55%-os telítettségű, 5% alapvörös (R) + 95% alapkék (B); a sárga pedig: NCS 0580-Y10R, vagyis 5%-ban sötét, 80%-os telítettségű, 90% alapsárga + 10% alapvörös.

1.3.3.8. A Nemcsics-féle Coloroid színatlasz

A Nemcsics Antal (sz. 1927) -féle Coloroid színatlasz (1980) az ismert három koordináta, a színezet, telítettség, illetve a világosság alapján osztályozza



1.29. ábra. *Az NCS színellentétek: fehér–fekete, zöld–vörös, sárga–kék*

a színeket. Nem ingerküszöbökre, hanem harmóniaküszöbökre épül. Más színrendszerekkel szemben, a Coloroid nem az emberi szem érzékenységén, hanem az ember ítéletalkotó képességén alapszik [68].

1.3.3.9. HSB, HLS, HSV színrendszerek

Az RGB, CMY és CMYK színterek felépítését alapvetően technikai szempontok határozták meg, ezért olyan színtereket is kialakítottak, amelyek jobban alkalmazkodnak az emberi érzékeléshez, látáshoz.

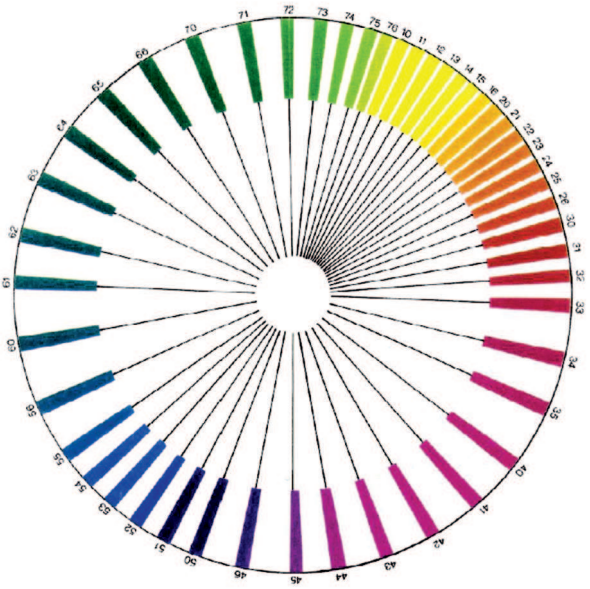
A HSB (*Hue – színárnyalat*, *Saturation – telítettség*, *Brightness – fényesség*) színtér egy henger, ahol a kör 360°-ából egy konkrét szögértékkel jellemezhetjük az RGB színek közötti átmeneteknek megfelelő színárnyalatokat. A kör középpontjától mért távolsággal fejezhetjük ki a telítettséget, és a henger alsó alapkörétől mért távolság adja meg a fényerősséget. Egy szín leírását így egy fokérték és két értékhármassal tehetjük meg.

A HLS vagy HSL a *Lightness* vagy *Luminance – világosság*, a HSV pedig a *Value – színérték* tényezőkkel ábrázolja az RGB színek közötti átmeneteket.

1.3.4. Átalakítások színrendszerek között

A számítógép digitális elven működik, minden adat, információ – így a színek is – számokkal vannak leírva. A képernyő működéséből adódóan az additív színkeverést alkalmazza, vagyis minden szín leírható a vörös, a zöld és a kék árnyalataival.

A számítógépen használatos *RGB színmodell* egy olyan háromdimenziós színtér, melyben minden színt egy számhármassal *(R, G, B)* adhatunk meg. A gyakorlatban használt értékek a (0, 0, 0)-tól a (255, 255, 255)-ig terjedhetnek, vagyis minden színkomponenst egy 8-bites szám ábrázol. (0, 0, 0) a fekete kódja, (255, 0, 0) a vörösé, (0, 255, 0) a zöldé, (0, 0, 255) a kéké, (255, 255, 255) pedig a fehérnek felel meg.



1.30. ábra. *A Nemcsics-féle Coloroid színingerei: 10–16 sárgák, 20–26 narancsok, 30–35 vörösek, 40–46 bíborok, 50–56 kékek, 60–66 hideg zöldek, 70–76 meleg zöldek*

Ebben a rendszerben $2^{8+8+8} = 2^{24} = 16\ 777\ 216$ színt ábrázolhatunk, ez az úgynevezett *TrueColor* tartomány.

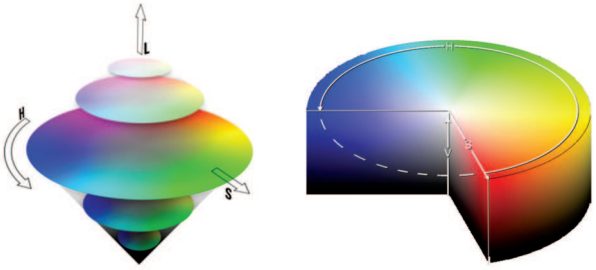
Láttuk azonban, hogy a gyakorlatban más színrendszerek is elterjedtek. Annak érdekében, hogy a számítógépen ábrázolt színeket más színrendszerekben is le tudjuk írni, *színskála-transzformáció*kat kell alkalmaznunk.

1.3.4.1. Színes kép ábrázolása szürkeárnyalatokkal

A *szürkeárnyalatos (Grayscale)* képek a fekete-fehér fotókhoz hasonlatosak: a színek szürkeárnyalatokkal vannak ábrázolva, így folyamatos, lágy tónusátmenetet érzékelünk.

A szürkeárnyalat képzése úgy történik, hogy az adott színt felbontjuk a három színösszetevőre (vörös, zöld, kék), majd az így kapott három szám átlagát visszaírjuk az adott szín minden egyes színösszetevőjére.

$$\begin{cases} R := \frac{R+G+B}{3} \\ G := \frac{R+G+B}{3} \\ B := \frac{R+G+B}{3} \end{cases}$$



1.31. ábra. *A HLS és HSV színtestek*

vagy vektoros alakban ezt így írjuk:

$$\left[\begin{array}{ccc} R & G & B \end{array}\right] := \left[\begin{array}{ccc} \frac{R+G+B}{3} & \frac{R+G+B}{3} & \frac{R+G+B}{3} \end{array}\right],$$

ahol := az értékadást jelenti.

1.3.4.2. Átalakítás az RGB és a CMY, CMYK színrendszerek között

A CMYK-színrendszerben a színkomponenseket általában egy 0–100 közötti érték írja le (százalékos előfordulást jelzünk), így elvileg 104 060 401 árnyalata lehet egy képpontnak.

A CMY alapszínei az RGB alapszíneinek komplementerei, így könnyen elvégezhető a transzformáció a két modell között:

$$\begin{cases} C := 255 - R \\ M := 255 - G \\ Y := 255 - B \end{cases}, \text{ vagy } \begin{cases} R := 255 - C \\ G := 255 - M \\ B := 255 - Y \end{cases}$$

Vektoros alakban ezt így írhatjuk:

$$\left[\begin{array}{ccc} C & M & Y \end{array}\right] := \left[\begin{array}{ccc} 255 & 255 & 255 \end{array}\right] - \left[\begin{array}{ccc} R & G & B \end{array}\right]$$
$$\left[\begin{array}{ccc} R & G & B \end{array}\right] := \left[\begin{array}{ccc} 255 & 255 & 255 \end{array}\right] - \left[\begin{array}{ccc} C & M & Y \end{array}\right]$$

Az RGB és a CMY színterek közötti átszámítás kölcsönösen egyértelmű. Az RGB és a CMYK színterek színei között viszont nem lehetséges kölcsönösen egyértelmű megfeleltetés, mert az RGB értékeket számértékként, a CMYK értékeket viszont százalékként használjuk. Emiatt vannak olyan RGB színek, amelyek a CMYK alapszínek keverésével nem nyomtathatók ki (leginkább a kék szín környékén).

A színeket közelíteni kell. Azt az eljárást, amellyel a monitoron megjeleníthető színeket közelítjük a nyomtatott színekhez, *kalibrálás*nak nevezzük.

A legelterjedtebb megvalósítás az úgynevezett γ-*kalibrálás* vagy γ-*korrekció*, amelynek lényege az, hogy a képernyőn 50%-os intenzitással kigyújtott képpontok színe egyezzen meg az 50%-os fedettségű nyomdai raszter színével. Ezt a kalibrálást professzionális szoftverek végzik el (pl. Adobe Photoshop).

1.3.4.3. Átalakítás az RGB és a YUV színrendszerek között

Sok színrendszer nem egész, hanem valós számokkal dolgozik, így nem az RGB komponensek 0–255 közötti értékeit használja, az úgynevezett *normalizált RGB* értékeket, amelyek a valós $[0, 1]$ intervallumból vannak.

A normalizálást egyszerűen elvégezhetjük a következőképpen (jelölje r, g, b a normalizált értékeket:

$$\begin{cases} r := \frac{R}{R+G+B} \\ g := \frac{G}{R+G+B} \\ b := \frac{B}{R+G+B} \end{cases},$$

vagy fordítva:

$$\begin{cases} R := 255 \cdot r \\ G := 255 \cdot g \\ B := 255 \cdot b \end{cases}.$$

A normalizálás után az átalakítási képlet:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} := \begin{bmatrix} 0{,}299 & 0{,}587 & 0{,}114 \\ -0{,}14713 & -0{,}28886 & 0{,}436 \\ 0{,}615 & -0{,}51499 & -0{,}10001 \end{bmatrix} \cdot \begin{bmatrix} r \\ g \\ b \end{bmatrix},$$

vagy fordítva:

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} := \begin{bmatrix} 1 & 0 & 1{,}13983 \\ 1 & -0{,}39465 & -0{,}58060 \\ 1 & 2{,}03211 & 0 \end{bmatrix} \cdot \begin{bmatrix} Y \\ U \\ V \end{bmatrix}.$$

1.3.4.4. Átalakítás az RGB és az XYZ színrendszerek között

Itt is a normalizált RGB értékekkel dolgozunk:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} := \begin{bmatrix} 2{,}7688 & 1{,}7517 & 1{,}1301 \\ 1 & 4{,}5906 & 0{,}0600 \\ 0 & 0{,}0565 & 5{,}5941 \end{bmatrix} \cdot \begin{bmatrix} r \\ g \\ b \end{bmatrix}$$

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} := \begin{bmatrix} 1{,}967 & -0{,}548 & -0{,}297 \\ -0{,}955 & 1{,}938 & -0{,}027 \\ 0{,}064 & -0{,}130 & 0{,}982 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Mivel az RGB színtér erősen eszközfüggő, az XYZ pedig pusztán matematikai, ezért minden eszközre meg kell adni a megfelelő transzformációs képletet. A fenti képlet a CRT monitorok *fehér pontjá*nak függvényében volt kiszámítva. A fehér pont a monitor fehér fényének *színhőmérséklet*ét jelenti, azaz azt a hőfokot, amelyre egy ideális fekete testet hevítve, a sugárzó monitor fehér fényével azonos színt bocsát ki [94].

1.3.4.5. Átalakítás az RGB és a HLS, HSV színrendszerek között

Ezek az átalakítások már bonyolultabb műveleteket (algoritmust) igényelnek, itt is a normalizált RGB értékekkel dolgozunk.

Legyen $MAX := \max(r, g, b)$és $MIN := \min(r, g, b)$, ekkor:

$$H := \begin{cases} 0, & \text{ha } MAX = MIN \\ 60^\circ \cdot \frac{g-b}{MAX-MIN} + 360^\circ \pmod{360^\circ}, & \text{ha } MAX = r \\ 60^\circ \cdot \frac{b-r}{MAX-MIN} + 120^\circ, & \text{ha } MAX = g \\ 60^\circ \cdot \frac{r-g}{MAX-MIN} + 240^\circ, & \text{ha } MAX = b \end{cases}$$

$$L := \frac{MAX + MIN}{2}$$

$$S_{HLS} := \begin{cases} 0, & \text{ha } MAX = MIN \\ \frac{MAX-MIN}{MAX+MIN}, & \text{ha } L \leqslant \frac{1}{2} \\ \frac{MAX-MIN}{2-(MAX+MIN)}, & \text{ha } L > \frac{1}{2} \end{cases}$$

$$S_{HSV} := \begin{cases} 0, & \text{ha } MAX = 0 \\ \frac{MAX-MIN}{MAX} & \text{különben} \end{cases}$$

$$V := MAX.$$

A fordított átalakítási algoritmusok:
HLS esetében legyen:

$$q := \begin{cases} L \cdot (1 + S_{HLS}), & \text{ha } L < \frac{1}{2} \\ L + S_{HLS} - (L \cdot S_{HLS}), & \text{ha } L \geqslant \frac{1}{2} \end{cases}$$

$$p := 2 \cdot L - q$$

$h_k := \frac{H}{360}$, a H érték normalizálva

$$
\begin{aligned}
t_R &:= h_k + \frac{1}{3} \\
t_G &:= h_k \\
t_B &:= h_k - \frac{1}{3} \\
&\text{ha } t_C < 0, \text{ akkor } t_C := t_C + 1, \text{ minden } C \in \{r, g, b\} \\
&\text{ha } t_C > 1, \text{ akkor } t_C := t_C - 1, \text{ minden } C \in \{r, g, b\}.
\end{aligned}
$$

Ezután kiszámoljuk a normalizált *(r, g, b)* értékeket a következőképpen:

$$
\text{Szín\_C} := \begin{cases} p + 6 \cdot (q - p) \cdot t_C, & \text{ha } t_C < \frac{1}{6} \\ q, & \text{ha } \frac{1}{6} \leqslant t_C < \frac{1}{2} \\ p + 6 \cdot (q - p) \cdot (\frac{2}{3} - t_C), & \text{ha } \frac{1}{2} \leqslant t_C < \frac{2}{3} \\ p, & \text{különben} \end{cases}, \quad \text{minden } C \in \{r,g,b\}
$$

HSV esetében legyen:

$$
\begin{aligned}
h_i &:= \left\lfloor \frac{h}{60} \right\rfloor \pmod 6 \\
f &:= \frac{h}{60} - \left\lfloor \frac{h}{60} \right\rfloor \\
p &:= v \cdot (1 - S_{HSV}) \\
q &:= v \cdot (1 - f \cdot S_{HSV}) \\
t &:= v \cdot (1 - (1 - f) \cdot S_{HSV}).
\end{aligned}
$$

Ezután kiszámoljuk a normalizált *(r, g, b)* értékeket a következőképpen:

$$
(r, g, b) := \begin{cases} (v, t, p), & \text{ha } h_i = 0 \\ (q, v, p), & \text{ha } h_i = 1 \\ (p, v, t), & \text{ha } h_i = 2 \\ (p, q, v), & \text{ha } h_i = 3 \\ (t, p, v), & \text{ha } h_i = 4 \\ (v, p, q), & \text{ha } h_i = 5 \end{cases}
$$

1.3.5. A színhasználat esztétikája, szimbolikája

„A szín élet. A színek nélkül halott lenne a világ” – írta Itten [48].

A színek mindig is meghatározó szerepet játszottak életünkben, ezért nehezen lehet létünket nélkülük elképzelni. A bennünket körülvevő természet színeinek pompáját szépnek és tökéletesnek látjuk. A színes ruhákban, illetve környezetben jobban érezzük magunkat, mint a szürkében vagy az egyszínűben. Társadalmunkban az emberek ruháik színeivel hovatartozásukat, rangjukat, sőt foglalkozásukat is kifejezhetik. Persze a színek ennél sokkal többre képesek, befolyásolhatják érzelmeinket, hangulatunkat.

A fénytannal és színtannal nem kisebb személyiségek végeztek kutatásokat és értek el máig érvényes eredményeket, mint Huygens, Young, Newton, Goethe, Schopenhauer, Helmholtz, Haering, Munsell, Ostwald, Maxwell, Grassmann és Einstein és mások.

A fizikusok között nem véletlen Goethe német költő neve, aki a színtan pszichikai megközelítésével készített tanulmányával, míg Newton (angol fizikus) a fizikai alapkísérletekkel szerzett örök érdemeket. Ezen tanulmányok tették lehetővé az alaptézisek lefektetését és azok gyakorlati alkalmazását. Mindkét mű, bár más-más módszerrel közelítette meg a színtan leírását, máig elévülhetetlen érdemeket szerzett szerzőiknek.

A fényhatások érzelmet kiváltó képességét Goethe fogalmazta meg [31]. Erre annyira büszke volt, hogy *Színtan* című munkáját költészete fölé helyezte. Goethe színtanának legtöbb megállapítása a mai napig nem veszített érvényességéből.

Tény az, hogy a szemünkön keresztül a különböző látható hullámhosszúságú rezgések által kiváltott agyingereknek a színvilágban és az érzetvilágban történő összekapcsolódását szubjektív módon érzékeljük.

Színek kiválasztásánál figyeljünk a színek különböző tulajdonságaira (kontrasztok, hideg-meleg színek; tónusok, árnyalatok). A felületet színes kontrasztra (kiegészítő színek, hideg-meleg, világos-sötét) vagy monokróm tónusokra építhetjük.

A felület akkor kellemes a szemnek, ha a színek minden tulajdonságát alkalmazzuk (például ha kiválasztjuk a kék-narancs kiegészítő színeket, az egyiket sötétebbre állítjuk, mint a másikat, és jobban kihangsúlyozzuk az egyik meleg vagy hideg tulajdonságát).

Több szín használata esetén ügyeljünk az összhangra, és csak egy dominánsszín legyen. Ha az ábrázolt információk között értékbeli különbség van – egyik fontosabb, másik nem –, ennek szemléltetésére a tónusos ábrázoláshoz folyamodhatunk. A legfontosabb információt a fénnyel legtelítettebb színnel, az utána következőket halványabb tónusú színnel ábrázoljuk.

A felületek megszerkesztésénél ajánlott a pasztellszínek használata, kerüljük az erőteljes színeket, ezeket esetleg csak a legfontosabb információk kiemelésére használjuk, de ne ezekből építsük fel a teljes felületet.

A világos színek vonzzák a tekintetet, a meleg világos színek vonzása még nagyobb (pl. cinóbervörös), a harsány citromsárgától egy idő után fájni kezd a szem, nyugtalanná válik, a kékben és a zöldben keres megnyugvást magának.

Egy tónusban ne szerepeljen azonos mennyiségben a három alapszín.

A *szín tónusértéké*ről vagy *valőr*jéről akkor beszélünk, ha meg akarjuk határozni világosságának vagy sötétségének fokát.

A színek szorosan kötődnek az érzelmekhez és a lelkiállapotokhoz is.

A sárga, a narancs és a vörös az öröm és a bőség eszméjét képviseli. A vörös nyugtalan, mozgékony, a világos vörös energikus szín. A piros a szeretet, a bátorság és a buzgóság; a zöld a remény, termékenység, ifjúság; a lila a bánat, méltóság; a fekete bűntudat; a kék a végtelenség, igazságosság, tudás; a sárga a becsületesség, alázat; a narancs a végesség, állandóság, kitartás; a fehér a hit és a tisztaság szimbóluma. A sárga, narancs és piros színeknek étvágygerjesztő hatásuk van, ezért az ilyen színű ételeket előszeretettel kínálják a gyorséttermek.

Színasszociációk a nyugati kultúrákban:

- piros: állj, veszély, forró, tűz, közeli;
- sárga: óvatosság, lassan, ellenőrzés;
- zöld: mehet, rendben, érthető, biztonság, növényzet;
- kék: hideg, víz, nyugalom, ég, távoli, biztonság;
- meleg színek: tevékenység, közelség;
- hideg színek: állapot, távolság;
- szürke, fehér: semlegesség.

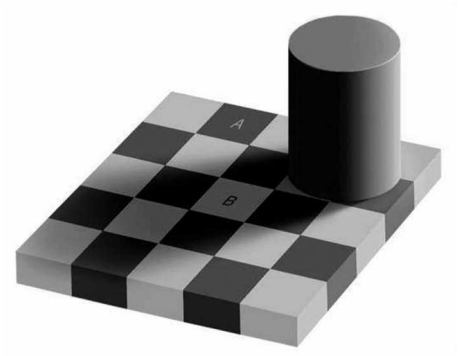
Kínában a sárga szín kizárólag a császárt illette meg, az Ég fiát. A sárga a legmélyebb bölcsesség, a legteljesebb megvilágosodás jelképe volt. A gyászoló kínaiak fehérbe öltöztek, jeléül annak, hogy elkísérik a megboldogultat a tisztaság és fény országába.

A harmónia egyensúly, az erők szimmetriája. A színek akkor harmonikusak, ha keverékükből fehér jön létre (Rumford 1707). A középszürke tehát a látóérzékünk által megkívánt egyensúlyi állapotnak felel meg.

Két vagy több szín akkor harmonikus, ha keverékükből semleges szürke jön létre. A másképpen csoportosított színek, amelyeknek keverékéből nem keletkezik szürke, minden esetben expresszív vagy diszharmonikus jellegűek. Feldúlják, felzaklatják az embert, mert egyoldalú hangsúllyal használják fel valamennyi színt. A szem és az agy megkívánja a középszürkét, ha ez nincs jelen, nyugtalanná válik. Ha egy fekete alapon fekvő fehér négyzetet nézünk, majd elfordítjuk róla a tekintetünket, utóképként fekete négyzet jelenik meg szemünkben, mert az egyensúlyi állapot megkíséreli visszaállítani önmagát. Ha szürke alapon semleges szürke négyzetet figyelünk meg, nem jelenik meg eltérő utókép.

A színvalóság és színhatás csupán harmonikus hangzatokban azonosak egymással, minden más esetben a szín valósága szimultán módon új hatást hoz létre. Így ha egy szürke lapot fekete lapra helyezünk, és egy ugyanolyan szürke lapot fehér lapra teszünk, ez utóbbit sötétebbnek fogjuk látni, mint azt a szürke lapot, amely fekete lapon fekszik.

A színek a formákkal is összhangban kell hogy álljanak. Az egyik forma fokozza, a másik csökkenti ugyanannak a színnek a jelentőségét. A hegyes forma



1.32. ábra. *Edward H. Adelson tanulmánya: az A-val és B-vel jelölt szürke egy és ugyanaz az árnyalat!*

kiemeli a harsány színek sajátosságait (pl. háromszög – sárga), a telt színek kerek formák esetén hatásosabbak (pl. sötétkék – kör), a négyzet a pirosat vonzza.

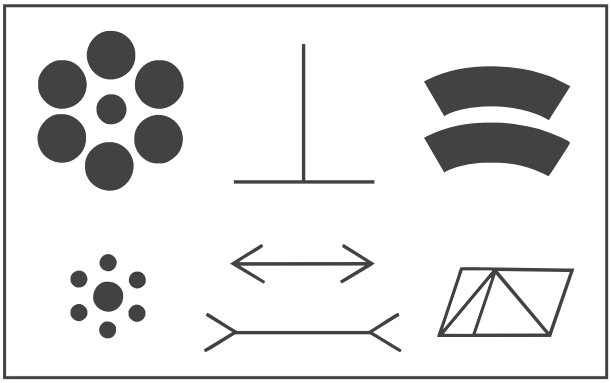
A fehér négyzet fekete alapon nagyobbnak hat, mint egy vele azonos nagyságú fekete négyzet fehér alapon. A fehér kisugárzik, túlsugárzik önnön határain, a fekete szín összehúzza a formát.

1.33. ábra. *Fehér és fekete négyzetek*

A színeknek fontos szerep jut a figyelemfelkeltésben is. A legfigyelemfelkeltőbb színkombináció a fekete a sárgán, ezután: fekete a fehéren, sárga a feketén, fehér a feketén, sötétkék a fehéren, fehér a sötétkéken.

A színek térhatása különböző tényezőktől függ, magukban a színekben is rejlenek a mélység felé ható erők, amelyek egy képen világos–sötét vagy hideg–meleg értékként, minőségként vagy mennyiségként jelennek meg. Fekete alapon a világos színek a maguk világossági fokozatának mértékében lépnek elő, fehér alapon ezek a hatások megfordulnak, a világosabb színek megmaradnak a fehér alap síkjában, a sötétebbek pedig fokozatosan előretolódnak.

Egyenlő világosság esetén a hideg és a meleg színek úgy viselkednek, hogy a meleg színek az előtérbe, a hideg színek pedig a mélység felé törekednek. Ha fény–árnyék kontraszt lép fel, akkor a mélységi erők vagy összeadódnak, vagy kioltják egymást, vagy pedig az ellentétükbe fordulnak át. A minőségi



1.34. ábra. *Példa a látás kontextusfüggőségére. Két ugyanakkora kör közül kisebbnek látjuk azt, amelyik nagyobb körök környezetében van, mint azt, amely kisebb körök környezetében van. A nyílhegyeknek vagy a párhuzamos, merőleges irányoknak megtévesztő hatásuk van*

kontrasztban a világító szín az előtérbe lép a vele egyenlő világosságú, tompább színnel szemben.

Az egyes színeknek a következő pszichikai és optikai hatásuk van [2]:

- *Sárga*: Ösztönző, vidám, kommunikatív. Tágítja a teret, ha világos árnyalatú, előtérben tolakodó, ha erős színárnyalatot választunk.
- *Sárgászöld*: Barátságos, vidám, természetközeli. Világos árnyalatai tágítják a teret, sötét árnyalatai pedig szűkítik.
- *Zöld*: Kiegyensúlyozott, megnyugtató, kikapcsolódást segítő. Semleges.
- *Kékeszöld*: Stabilizáló, megnyugtató, hűvös. Szűkíti a teret.
- *Kék*: Hideg, friss, elegáns, távolságtartó. Tágítja a teret.
- *Indigókék*: Megnyugtató, komoly, távolságtartó. Kicsinyíti a teret, mélységet ad a térnek.
- *Kékeslila*: Komoly, ünnepélyes. Szűkíti a teret.
- *Püspöklila*: Extravagáns, kétélű, titokzatos, finom, rózsaszínes árnyalatok esetén kislányosan nőies, élesen túlzó. Sötét árnyalatai szűkítik a teret.
- *Bíborvörös*: Vöröses, püspöklilához hasonló. Viszonylag semleges.
- *Bordó*: Dinamikus, cselekvésre serkentő, agresszív. Lehet nyomasztó, és szorongó érzést is kiválthat.
- *Narancsvörös* és sárgásvörös: Izgató, agresszív, cselekvésre ösztönző. Szűkíti a teret.
- *Fehér*: Világos, tiszta, könnyed. Tágítja a teret.
- *Fekete*: Tárgyilagos, komoly, súlyos. Erősen szűkíti a teret.
- *Szürke*: Passzív, semleges, kiegyensúlyozott. Semleges.
- *Meleg színek általában*:

  - a világos árnyalatok: Vidám, könnyed, serkentő. Tágas, eleven terek.
  - a sötét árnyalatok (barna, rozsda, okker): Megnyugtató, hangulatos, kiegyensúlyozott. Szűkíti a teret, körbezár.
- *Hideg színek általában*:
  - a világos árnyalatok: Passzív, tiszta, friss, világos. Háttérbe vonuló, erősen tértágító.
  - a sötét árnyalatok: Tárgyilagos, előkelő, komoly. Erősen korlátozó, mélységérzetet kelt.

A látványban mutatkozó színek felfogása is eltér az egyes embereknél. Van, aki a színeket egyenként fogja fel, minden színt külön érzékel, és van, aki egyszerre fogja fel a színfoltok sokaságát, az összhangot ragadja meg.

1.3.6. A sztereó látás

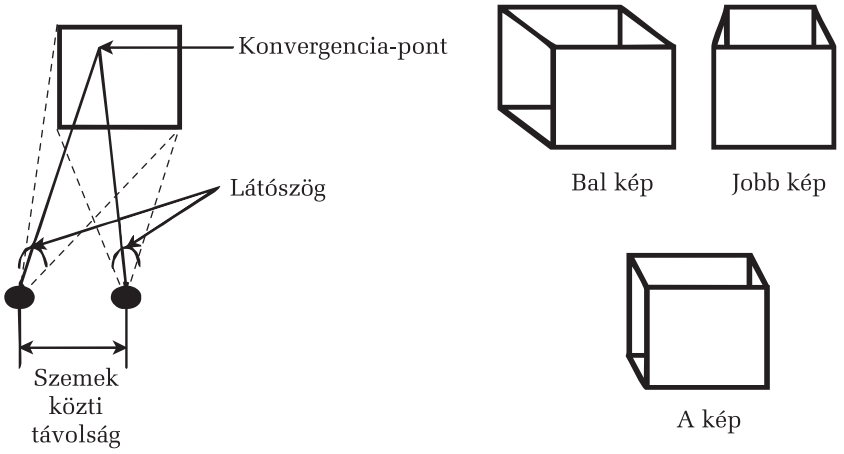
A minket körülvevő anyagi, valós világ háromdimenziós, a tér három koordináta mentén *(x, y, z)* szerveződik. Beszélhetünk hosszúságról, szélességről, magasságról és jobbra-balra, előre-hátra, fel-le mozoghatunk. Az ábrázolási lehetőségeink nagy többsége (papír, könyv, TV, monitor stb.) azonban csak kétdimenziósak, két koordinátánk van (x, y), csak hosszúságról és magasságról beszélhetünk, jobbra-balra, csak fel- vagy le mozoghatunk.

Az ember – mint vizuális lény – mindig is arra törekedett, hogy a lehető legpontosabban, legtöbb információval ábrázolja a háromdimenziós valós világot a kétdimenziós adattárolókon. Ebből a célból fejlesztették ki a különböző fényképezési technikákat, vetítéseket, ábrázolási módokat. Mindezek által az ábrázolási mód így is szűkös marad: egy szobor fényképét nem tudjuk például bejárni, nem tudjuk megnézni, hogy „mi van hátul". Napjaink grafikus szoftverei hűen ábrázolják a valóságot, már forgatni tudnak, körbejárhatóvá teszik az objektumokat, de ezekhez az ábrázolási módokhoz rengeteg információt kell tárolni. Összegezve elmondhatjuk, hogy lehetőségeink így is szűkösek.

A mélységlátás a szem alapvető funkciói közé tartozik. Nemrég mutatták ki, hogy az emberi agykéreg mintegy ötven százaléka szerepet játszik a vizuális érzékelésben, vagyis legalább két pályarendszer és számos egymástól elkülönült, független funkciójú terület bonyolult együttműködése teszi lehetővé a háromdimenziós látást. A vizuális inger értelmezésében jelentős szerepet játszik a tapasztalat is. A retinára vetülő kép valódi, kicsinyített és fordított állású, ám egyenes állásúnak érzékeljük, mert a tapasztalataink ezt diktálják. Hasonlóan – mivel a retina és a rávetülő kép egyaránt kétdimenziós – a térlátásunk a kétdimenziós vetületek elemzésével és értékelésével valósul meg. A tárgyak mélységdimenziójának felismerése, vagyis a *térbeli (sztereó) látás* a két szemmel való nézés eredménye. A két szemtengely eltérése, a két szem helyzete enyhén különböző képeket hoz létre a két retinán és ennek következtében az agyban

is. Az emberi agy az, amely elemzi, értékeli és összegezi a két képet. A sztereó látás a kb. 0,25–50 m távolsághatárok között fekvő tárgyakról ad közvetlen távolságérzetet.

Tehát térlátásunk azon alapszik, hogy két szemünk más-más képet lát, és ezeket az agy térinformációkká alakítja át.



1.35. ábra. *A sztereó, vagyis a térbeli látás*

1.3.7. Sztereogramok

Térlátásunkat szimulálja a *sztereogram*. A sztereogramok egy újfajta grafikai irányzat eredményei, amelyek lényege, hogy egy papírlapra nyomtatott kép is okozhat valódi térhatást, ha azt megfelelően nézzük: a kép mögé fókuszálunk, vagy keresztezzük a szemeinket, „elbambulunk". Ekkor mindkét szemünk a papírlap más-más részére fókuszálódik, és más-más képet lát, vagyis létrejöhet a kívánt térhatás.

A sztereogramok fogalmával szorosan összefügg Julesz Béla (1928–2003) magyar neuropszichológus neve. Az 1960-as években Julesz Béla által kifejlesztett véletlen-pont sztereogramok *(Random Dot Stereogram)* forradalmasították a mélységészlelés kutatási területét, és kutatók generációinak szolgáltak inspirációul.

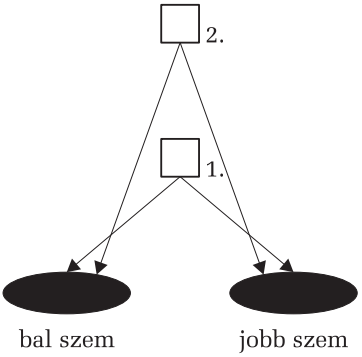
Ha a sztereogramokat osztályozni próbáljuk, a következő három kategóriát különíthetjük el:

- *véletlen-pont sztereogramok* (SIRDS – Single Image Random Dot Stereograms),
- *véletlen-szöveg sztereogramok* (SIRTS – Single Image Random Text Stereograms),
- *egyképes sztereogramok* (SIS – Single Image Stereograms).

1.36. ábra. *dr. Julesz Béla*

A *véletlen-pont sztereogramok* az eredeti, Julesz Béla által bevezetett sztereogramok. Működésük lényege, hogy a közelebbi tárgyak mindig távolabb vetülnek a két szem retinájára, mint a távolabbiak. Így ha egy adott mintázatot a jobb és bal szemnek szánt képen közelebb hozunk egymáshoz, azt a mintázatot egyre távolabbinak fogjuk látni.

Véletlen-pont sztereogramokat generáló algoritmust a 6.3. (*Az első OpenGL példaprogram Visual C++-ban*) fejezetben láthatunk.



1.37. ábra. *Tárgyak vetülése a retinákra*

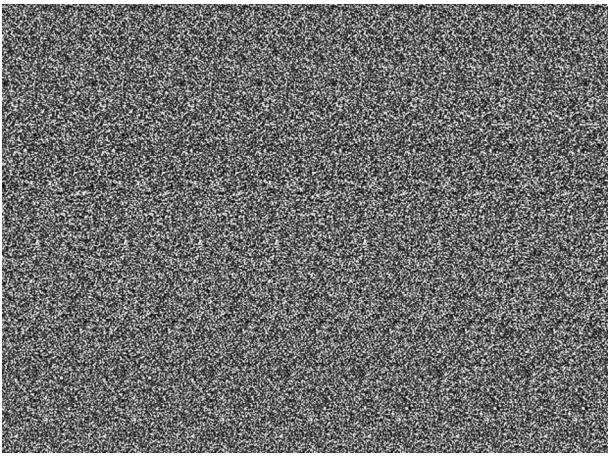
A sztereogramok elkészítéséhez elengedhetetlenül szükséges a számítógép. A képet jobb és bal oldali nézőpontból vizsgáljuk.

Képzeljük el, hogy egy tárgyat úgy nézünk egy üveglapon vagy papíron keresztül, hogy ahol a tárgy egy pontjából kiinduló és a bal, illetve jobb szembe érkező fénysugár áthatol ezen a lapon, oda egy pontot rajzolunk. Így a tárgy

minden egyes pontjának a lapon két pont felel meg, egy a jobb, a másik a bal szem számára. Ha megoldjuk, hogy ezeket a pontokat a két szem külön érzékelje, ezekből az agyunk egy térbeli képet rak össze.

A kép készítésekor először az alakzatot véges sok pontra kell bontani, majd soronként végighaladva rajta, az előbb ismertetett leképezéssel minden pontról el kell készíteni a képpontokat. Az egyes sorokat általában más színnel jelenítjük meg, az élek mentén pedig szintén más színűek lesznek a megfelelő pontok. Így tehát olyan ponthalmazokat kapunk, amelyet a látósugarak rajzoltak volna ki a lapra. Ha most egyesíteni akarjuk a képet, ellazult, „elbambult" szemmel csak annyit kell elérnünk, hogy a megfelelő pontok külön-külön a két szembe jussanak. Nem mindenki látja a Julesz-féle sztereogramokat. Az emberek 10–15%-a egyáltalán nem látja, másoknak pedig néhány percbe is telhet az első alkalommal, hogy összeálljon a kép.

A számítógéppel az is megoldható, hogy egy tartományon belül más nézőpontból is elvégezzék ezt a leképezést, így az észlelt kép a fejünk mozgatásakor ugyanúgy változik, mint amikor a valódi tárgyat is egy kissé más szögből nézzük, tehát a térbeliség illúziója tökéletes.



1.38. ábra. *Véletlen-pont sztereogram*

A *véletlen-szöveg sztereogramok* hasonlóak a véletlen-pont sztereogramokhoz, csak itt a pixelek (képpontok) szerepét a karakterek veszik át, számítógéppel generálva tehát szöveges üzemmódban is láthatók, nemcsak grafikus üzemmódban.

Az *egyképes sztereogramok* kissé bonyolultabbak, mint az előbbiek. Itt két képre van szükség: egy előtérre és egy háttérre. Az előtérkép akármilyen lehet, egy egyszerű fénykép, festmény vagy grafika. A háttérkép valamilyen módon olyan információkat tartalmaz, hogy az azon lévő test egy-egy pontja milyen

1.39. ábra. *Véletlen-szöveg sztereogram*

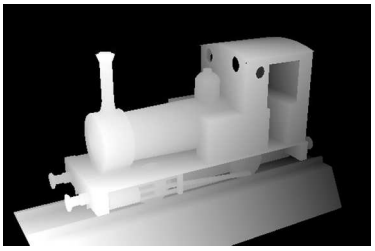
messze van a szemlélőtől. Egy ilyen módszer a *z-bufferelt* kép, amelynél a képpont színe hordozza a térinformációt, azaz a mélységre vonatkozó adatokat. Ez a kép egy szürke árnyalatú kép, amelyen az egyes szürke árnyalatok a test térbeli távolságát ábrázolják. A z-bufferelt képet előállíthatjuk háromdimenziós tervezőprogramokkal, sugárkövetőkkel.

A számítógépes program, figyelembe véve a szemek közötti távolságot, a térbeli látás tulajdonságait, valamint a háttérképet, torzítja és egymás mellé másolja az előtérképet – mintegy beledolgozza a háttérképet a sokszorozott előtérbe. Az ismétlés és a torzítás adja végül ki a sztereogramot, amelyre nézve látni fogjuk a háromdimenziós háttérképet.

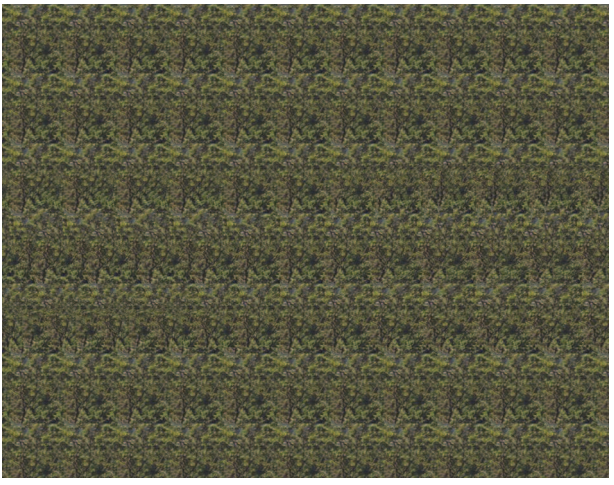
Hogyan nézzük a sztereogramokat? Sztereogramok nézésére három módszer ismeretes:

- Ellazulva, meredten kell nézni a képet 40–50 cm távolságból néhány percig úgy, hogy ne egy pontra koncentráljunk, hanem csak „bambuljunk”.
- Hajoljunk teljesen közel a képhez, majd lassan távolodjunk el tőle 40–50 cm-re, miközben a szemünk ugyanúgy néz, mint mikor közel volt a képhez.
- 40–50 cm távolságból a kép felé nézve ne a látható képre nézzünk, hanem a kép mögé 40–50 cm távolságra.

1.40. ábra. *Előtérkép – egy fa*



1.41. ábra. *Háttérkép*



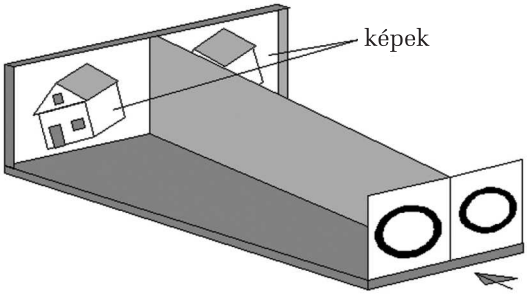
1.42. ábra. *Egyképes sztereogram*

1.3.8. Sztereofotók

A *sztereofényképe*ket speciális, kétobjektíves fényképezőgéppel készítik. A sztereofotózás az 1850-es évektől kezdve, röviddel a fotográfia felfedezése után indult el népszerűsége útján. Az 1920-as évektől kezdve sztereofilmek is készültek, melyek közül néhányat videokazettán is kiadtak, sőt, napjainkban sztereorészleteket tartalmazó DVD-k is napvilágot láttak.

1.43. ábra. *Sztereó fényképezőgép*

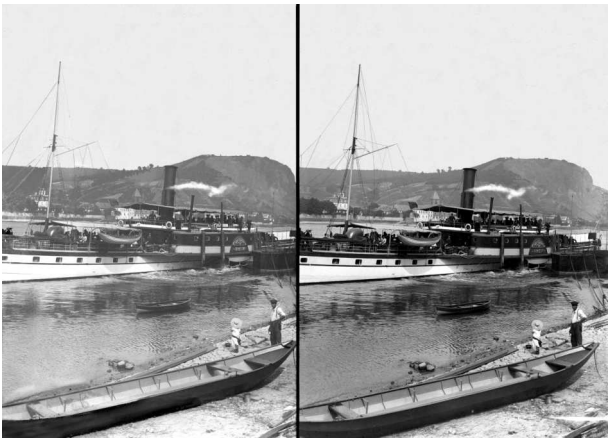
A sztereofényképeket legegyszerűbben az ún. *sztereonéző* vagy *sztereoszkóp* segítségével szemlélhetjük. A sztereoszkóp az emberi szempár távolságának megfelelően elhelyezett, két egyforma, párhuzamos tengelyű gyűjtőlencsét tartalmaz. Ezeken át mindegyik szem a neki megfelelő képet látja felnagyítva.



1.44. ábra. *A sztereoszkóp vázlatos szerkezete*

Idetartoznak a különféle, virtuális valóságot megjelenítő eszközök is, például a *Shutter-technológia*, amely úgy működik, hogy a felhasználó egy két LCD kijelzőből álló szemüveget kap, melynek kijelzői felváltva eltakarják a szemeit,

a monitoron pedig, ezzel szinkronban, mindig az éppen el nem takart szemnek megfelelő kép látható.



1.45. ábra. *Remageni sztereofotó, készítette Baptist Schneider (1867–1946)*

Sztereofotók, térhatású ábrák megjelenítésére a legközismertebb, legelterjedtebb módszer az *anaglif-technika (anaglyph)*. Az anaglif-módszer lényege az, hogy a bal és a jobb szem helyzetének megfelelően felvett két képet kiegészítő színekkel (pl. az egyik kép vörös, a másik cián árnyalatú) másolja egymásra. Ha egy ilyen képet vörös-cián szemüveggel nézünk (vörös lencse a bal szemen, cián a jobbon), a kiegészítő színek hatásmechanizmusának köszönhetően térhatást érünk el, a képet az agy térben képes érzékelni.

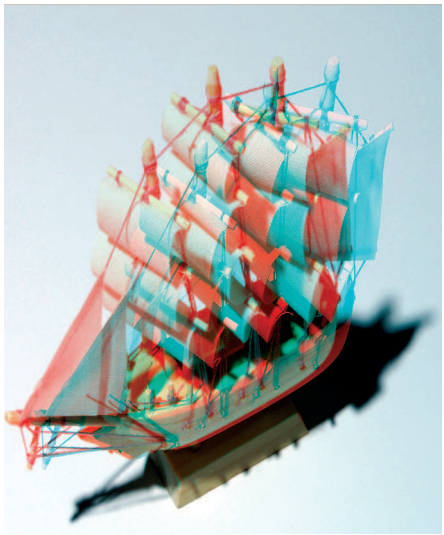
Az elkülönítés úgy történik (vörös-cián színpár) esetében, hogy a bal képet vörös, a jobb képet pedig cián szűrőn át képezzük le ugyanarra a felületre. Majd a szabad szemmel kissé zavarosnak látszó képet vörös-cián szemüvegen át nézve érzékeljük a sztereó hatást.

Diavetítésnél két, szinkronban működő, különálló – esetleg ikervetítőgépet alkalmaznak, közös vetítővászonnal, az objektívek elé tett színszűrőkkel.

A papírképek nyomdai színeltolással készülnek, de napjainkban a számítástechnika ezt az eljárást is hétköznapivá, bárki számára hozzáférhetővé tette. A digitalizált sztereoképpárokból szinte minden komolyabb képszerkesztő programmal tudunk anaglif képeket előállítani egyszerűen úgy, hogy az egyik kép vörös, a másik zöld- és kék-csatornájának tartalmát egyesítjük egy képben. A hagyományos 2D-s képeinkből is készíthetünk térhatású képet az egyes képrészletek másolásával, mozgatásával. Azonban már erre is léteznek komoly szoftverek.

Az anaglif-eljárás nem sokkal fiatalabb a fényképezésnél. 1853-ban W. Rollman néhány – piros és kék – vonalból álló ábrával, piros-kék szemüveg

segítségével igazolta a hatást. 1858-ban Joseph D'Almeida vetített először közönség előtt anaglif képeket. Ő piros-zöld színszűrőket és szemüveget használt. 1891-ben Louis Ducos du Hauron anaglif papírképeket készített, és ő szabadalmaztatta az eljárást. Az eljárás ezzel egy időben a filmtechnikában is teret nyert. 1889-ben elkészült William Friese-Greene anaglif mozifilmje, amit 1893-ban mutatott be a nagyközönségnek. 1897-ben Claude-Agricol-Louis Grivolas kettős felvevő- és vetítőgépet használt erre a célra [58]. A kuriózumszámba menő anaglif, 3D mozik napjainkban is nagy népszerűségnek örvendenek.



1.46. ábra. *Anaglif fénykép (forrás: [46])*

1.4. Fénytan, megvilágítás és árnyékolás

A fény homogén és izotróp közegben egyenes vonalban terjed. Mérések szerint a fény légüres térben terjed a legnagyobb sebességgel, c = 299 792 458 m/s (*fénysebesség*).

*Fényforrás*nak nevezünk minden olyan entitást (természetest és mesterségest egyaránt), amely látható fény előállítására szolgál.

A fényforrásokat akkor látjuk, ha a róluk kiinduló fény a szemünkbe érkezik. A nem világító testeket akkor látjuk, ha valamely fényforrás megvilágítja azokat, és a róluk visszaverődő fény a szemünkbe jut. Ezeket a testeket *másodlagos fényforrás*oknak nevezzük.

Az egyenes vonalban haladó keskeny fényt *fénysugár*nak nevezzük. Több, együttes fénysugár alkotja a *fénynyaláb*ot.

Ha egy fénysugár egy objektumra (tárgy, test stb.) esik, akkor a fényt alkotó elektromágneses sugárzás hullámhosszának függvényében az objektum átengedi vagy nem engedi át a fénysugarat, általában az objektumok a rájuk eső fény egy részét elnyelik, más részét átengedik, illetve visszaverik.

A fény visszaverődése (*reflexió*) a tárgy felületétől függ.

Egy felületről visszavert fény jellemző tulajdonságai függnek a beeső fény intenzitásától, a fényforrás mértani alakjától és helyzetétől, valamint a felület anyagának a tulajdonságaitól.

Egy felületről visszavert fény két komponensből tevődik össze:

- egy *diffúz (szórt, terjedő)* komponensből és
- egy *spekuláris (tükrözött)* komponensből.

1.4.1. A diffúz visszaverődés

Egy felület által visszavert fény minden irányban terjed, és az intenzitás nem függ a megfigyelő helyzetétől. Lambert törvénye megadja egy pontszerű fényforrástól származó, tökéletesen diffúz felület által visszavert fény intenzitását. Ennek alapján egy tökéletesen diffúz felület által visszavert fény intenzitása egy P pontban egyenesen arányos a beeső fény irányítása és a felületre a P pontban állított *normális*sal *(normálvektor)* bezárt szög koszinuszával [24].

$$I_d = I_i \cdot k_d \cdot \cos i,\ 0 \leqslant i \leqslant \pi/\pi 22,$$

ahol:

I_i = a beeső fény intenzitása,
k_d = a beeső fény diffúziós együtthatója $0 \leqslant k_d \leqslant 1$,
i = a normálvektorral bezárt szög.

Egy felület valamely pontjában vett *normális*án azt az egységnyi hosszúságú vektort értjük, amely az adott pontban merőleges a felületre, vagyis a felület érintősíkjára. Az $s(u, v)$paraméteres formában adott felület (u_0, v_0) pontjában vett normálisa a

$$\frac{\partial}{\partial u} s\,(u_0, v_0) \times \frac{\partial}{\partial v} s\,(u_0, v_0)$$

vektor, az $F(x, y, z) = 0$ implicit formában adotté pedig a

$$\left(\frac{\partial}{\partial x} F, \frac{\partial}{\partial y} F, \frac{\partial}{\partial z} F \right).$$

Minden normális három komponensből áll (x, y, z) és egységnyi hosszúságú, ezért

$$\sqrt{x^2 + y^2 + z^2} = 1.$$

Egy sík felület esetén a merőleges irány a felület összes pontjára ugyanaz, de egy nem egyenletes felület esetén a normális a felület minden pontján más és más lehet.

Ha i nagyobb mint $\pi/2$, akkor a felület nem kap fényt a fényforrástól, más szóval a fényforrás a felület mögött van.

A diffúziós együttható függ a felület anyagának tulajdonságaitól és a beeső fény hullámhosszától. Ezt az együtthatót általában konstansnak szokták tekinteni egy felület minden pontjában.

Az objektumok nemcsak a fényforrásoktól kapnak fényt, hanem a környező objektumok által visszavert vagy átengedett fény is eljut hozzájuk. A lokális megvilágítási modellekben a más objektumok által visszavert vagy átengedett fényt *ambiens (környezeti)* fénynek nevezzük, és úgy ábrázoljuk, mint egy egyenletesen eloszlott fényforrást a térben.

A megvilágítási modell a következőképpen alakul:

$$I_d = I_a \cdot k_a + I_i \cdot k_d \cdot \cos i,\ 0 \leqslant i \leqslant \pi/\pi 22,$$

ahol:

I_a = az ambiens fény intenzitása,
k_a = az ambiens fény diffúziós együtthatója, amely függ a felület anyagától.

Amikor a fényforrás pontszerű és nagyon távol van az objektumoktól, a fényforrást *egyenes fényforrás*nak nevezzük.

A fény intenzitása fordítottan arányos a fényforrás és objektum közötti távolság négyzetével. Tehát a fényforrástól távolabb eső objektumok gyengébben lesznek megvilágítva. Ezt figyelembe véve, a modell megváltozik:

$$I_d = I_a \cdot k_a + f_{att} \cdot I_i \cdot k_d \cdot \cos i,\ 0 \leqslant i \leqslant \pi/\pi 22,$$

ahol $f_{att} = 1/d^2$egy tompító függvény, d a távolság a fényforrás és objektum között.

Ha a fényforrás nagyon közel van, az intenzitás túl nagy lesz, ezért a f_{att}-ot ez esetben másképp írjuk fel:

$$f_{att} = \min\left(\frac{1}{c_1 + c_2 \cdot d + c_3 \cdot d_2}, 1\right),$$

ahol $c1$, c_2, c_3 a fényforráshoz rendelt konstansok, c_1-et úgy választjuk meg, hogy a nevező ne legyen túl kicsi, ha a távolság kicsi. Ahhoz, hogy a tompítás megtörténjen, 1-gyel határoltuk el a függvényt.

Mivel általában a fény nem monokromatikus és a felület, amire esik, úgyszintén színes is lehet, a fenti képletet átírhatjuk a fénynek minden komponensére. Ha például a használt fénymodell az RGB modell, akkor a piros komponensre a képlet a következőképpen néz ki:

$$I_{dR} = I_{aR} \cdot k_a + f_{att} \cdot I_{iR} \cdot k_d \cdot \cos i,\ 0 \leqslant i \leqslant \pi/\pi 22,$$

és hasonlóan felírhatjuk az I_{dG}-t és I_{dB}-t.

Általánosan:

$$I_{d\lambda} = I_{a\lambda} \cdot k_a + f_{att} \cdot I_{i\lambda} \cdot k_d \cdot \cos i,\ 0 \leqslant i \leqslant \pi/2.$$

Ez bármilyen hullámhosszú fényre és bármilyen megvilágítási modellre igaz.

1.4.2. A spekuláris visszaverődés

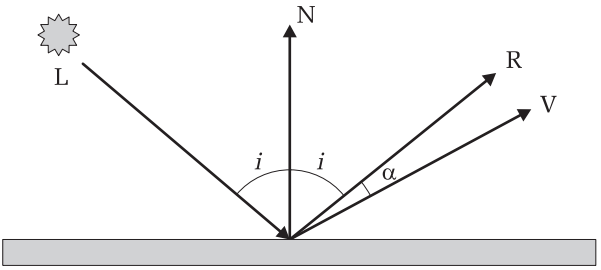
Ha a fénysugarak egy nagyon fényes és egyenletes, sima felületre esnek, akkor *tükrös visszaverődés*ről beszélhetünk.

Egy tökéletes visszaverő anyag (pl. egy tükör) a fényt csak egy irányba veri vissza.

A tükörnek azt a pontját, ahol a beeső fénysugár eléri a tükröt és visszavert fénysugárrá változik, *beesési pont*nak nevezzük. A beesési pontban a tükörre állított merőleges a *beesési merőleges*. A beeső fénysugár és a beesési merőleges által bezárt szög a *beesési szög*, a visszavert fénysugár és a beesési merőleges által bezárt szög a *visszaverődési szög*.

A fényvisszaverődés törvényei:

- A visszaverődési szög mindig ugyanakkora, mint a beesési szög.
- A beeső sugár, a beesési merőleges és a visszavert sugár egy síkban vannak.
- Azok a fénysugarak, amelyek merőlegesen esnek a felületre, önmagukban verődnek vissza.
- Ha a beeső fénysugarak párhuzamosak, akkor a visszavert fénysugarak is párhuzamosak.



1.47. ábra. *A fényvisszaverődés*

Mivel R és L szimmetrikus az N normálishoz viszonyítva, a visszavert fényt csak akkor veszi észre a megfigyelő, ha épp a megfelelő irányításon nézi.

A nem tökéletesen visszaverő felületeknél a megfigyelőhöz jutott fénymennyiség függ a spekulárisan visszavert fény eloszlásától. A sima felületeknél az eloszlás egyenletes, a durvább felületeknél viszont szétszóródik. Általában a visszavert fénynek ugyanolyan jellemzői vannak, mint a beeső fénynek.

A nem tökéletesen visszaverő felületeknél a hirtelen intenzitáscsökkenést, amikor a beesési szög nő, $\cos^n \alpha$-el lehet megközelíteni, ahol n a spekuláris visszaverési hatványa a felület anyagának.

Így a spekuláris fény intenzitása:

$$I_s = I_i \cdot w\,(i,\,\lambda) \cdot \cos^n \alpha,$$

ahol:

I_i = a beeső fény intenzitása,
$w\,(i,\,\lambda)$ = a visszaverődési függvény,
i = a normálvektorral bezárt szög,
λ = a beeső fény hullámhossza.

Az n-et az anyag típusától függően kell megválasztani. A nagy n értékek a fémekre és más fényes anyagokra jellemzők, a kis n értékek pedig a nemfémes anyagokra, mint például a papír.

Mivel a visszaverési függvény eléggé komplex, ezért a gyakorlatban egy konstanssal lehet helyettesíteni, amit *spekuláris visszaverődési konstans*nak nevezünk.

Így a felületek megvilágítási modellje a következőképpen alakul:

$$I_\lambda = I_{a\lambda} \cdot k_a + f_{att} \cdot I_{i\lambda} \cdot (k_d \cdot \cos i + k_s \cdot \cos^n \alpha)\,.$$

Felhasználva, hogy:

$$\cos i = \frac{L \cdot N}{|L|\,|N|} = L_u \cdot N_u$$
$$\cos \alpha = \frac{R \cdot V}{|R|\,|V|} = R_u \cdot V_u,$$

a képletet így írhatjuk át:

$$I_\lambda = I_{a\lambda} \cdot k_a + f_{att} \cdot I_{i\lambda} \cdot (k_d \cdot (L_u \cdot N_u) + k_s \cdot (R_u \cdot V_u)^n)\,.$$

Általában nem csak egy fényforrás világítja meg az objektumokat, s mindegyik hozzájárul a visszavert fény intenzitásához. Feltételezve, hogy az objektumot m fényforrás világítja meg, a képlet így alakul:

$$I_\lambda = I_{a\lambda} \cdot k_a + \sum_{j=1}^{m} f_{att_j} \cdot I_{i\lambda_j} \cdot \left(k_d \cdot \left(L_{u_j} \cdot N_u\right) + k_s \cdot \left(R_{u_j} \cdot V_u\right)^n\right).$$

1.4.3. A fénytörés, áttetszőség és átlátszóság

Két közeg határfelületére érve a beeső fény egy része visszaverődik, a többi megtörik és a másik közegben halad tovább. Ha a két közeg átlátszó anyagból van, akkor a fénysugár az egyik átlátszó anyagból egy másik átlátszó anyagba hatol, csak egy kis része verődik vissza, és a nagyobbik része a fénysugárnak megváltoztatott iránnyal halad tovább.

Ezt a jelenséget *fénytörés*nek (*refrakció*) nevezzük.

A fénytörés törvényei:

- A beeső fénysugár, a megtört fénysugár és a beesési merőleges egy síkban van.
- A beesési szög szinusza egyenesen arányos a törési szög szinuszával (Snellius-örvény, 1620): $\sin i = n_{2,1} \cdot \sin \beta$. A megtört fénysugár és a beesési merőleges által bezárt szöget *törési szög*nek nevezzük. Az $n_{2,1}$ arányossági tényező a második közegnek az első közegre vonatkozó *relatív törésmutató*ja, amelynek értéke a két közegben mért fénysebességeknek a hányadosa: $n_{2,1} = c_1/c_2$.
- A beesési szög és a törési szög szinuszának hányadosa ugyanarra a két közegre állandó, ez a relatív törésmutató.
- Ha az első közeg légüres tér, akkor a második közegre vonatkoztatott törésmutatót *abszolút törésmutató*nak nevezzük.
- Ha a fénysugár merőleges a felületre, akkor a fény irányváltoztatás nélkül halad tovább.
- Ha a fénysugár párhuzamos oldalú (*planparalell*) lemezen haladva keresztül kettős törést szenved, a fény iránya nem változik, csak eltolódik az eredeti iránytól.
- Ha a fény prizmán keresztül halad át, akkor is kétszeres törést szenved, de a fény iránya megváltozik.

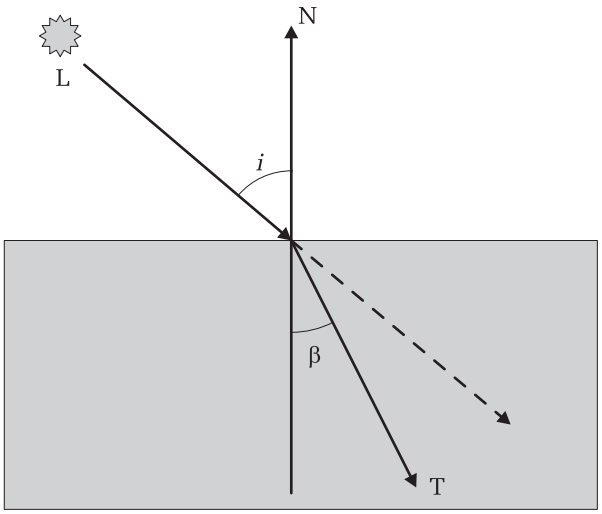
Az egyes objektumok *átlátszó*k vagy *áttetsző*k lehetnek. A fény terjedése az átlátszó objektumokon keresztül spekuláris, míg az áttetszőkön keresztül diffúz.

Az 1.49. ábrán a 3-as és 4-es objektum átlátszatlan, az 1-es és 2-es pedig átlátszó, ugyanazzal a törésmutatóval. Ha nem vesszük figyelembe a fénytörést, az a fénysugár a 3-as objektummal találkozna. A valóságban a törés miatt a 4-es objektumot metszi, amely emiatt megvilágított objektum lesz. Hasonlóan, a fénytörés figyelembevétele nélkül, a b fénysugár a 4-es objektumot metszené a 3-as helyett [21].

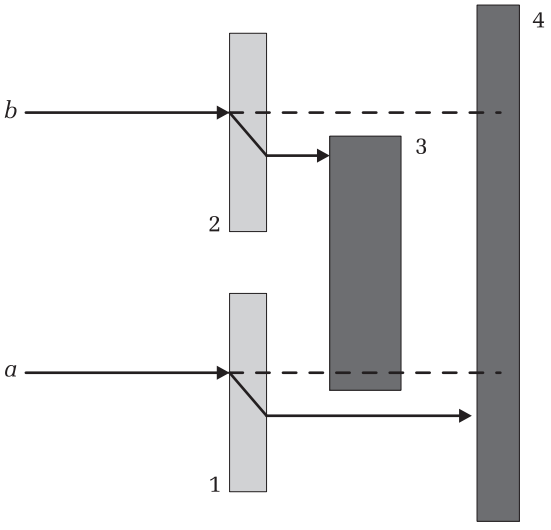
A fénytörés torzítja is az objektumokat a perspektivikus vetítéshez hasonlóan. A valósághűség érdekében számolni kell ezzel is.

Ha egy látható felület átlátszó, a színét a látható felület színének és a rögtön utána található felület színének az összetevéséből kapjuk meg, a következő interpolálási képletet használva [24]:

$$I_\lambda = (1 - k_{t_1}) \cdot I_{\lambda_1} + k_{t_1} \cdot I_{\lambda_2}, \quad 0 \leqslant k_{t_1} \leqslant 1,$$



1.48. ábra. *A fénytörés*



1.49. ábra. *A fénytörés szerepe*

ahol k_{t_1} a látható felület átlátszhatóságát méri az adott pontban. Ha $k_{t_1} = 0$, akkor a felület átlátszatlan, ezért a pont színe a felület színe lesz. Ha $k_{t_1} = 1$, a felület tökéletesen átlátszó, és a színe nem járul hozzá a pont színéhez. Ha $k_{t_1} = 1$ és a hátul levő felület szintén átlátszó, a számításokat rekurzívan folytatjuk, mindaddig, amíg egy átlátszatlan felületet kapunk vagy a háttérhez értünk.

Az átlátszóság eme megközelítése nem ad jó eredményt a görbe felületeknél, azért, mert a felület körvonalához közeledve, az anyag vastagsága megváltoztatja az átlátszóságot. Ebben az esetben a következő egyszerű nemlineáris megközelítést használjuk:

$$k_t = k_{t_{\min}} + (k_{t_{\max}} - k_{t_{\min}}) \left[1 - (1 - N_z)^m\right],$$

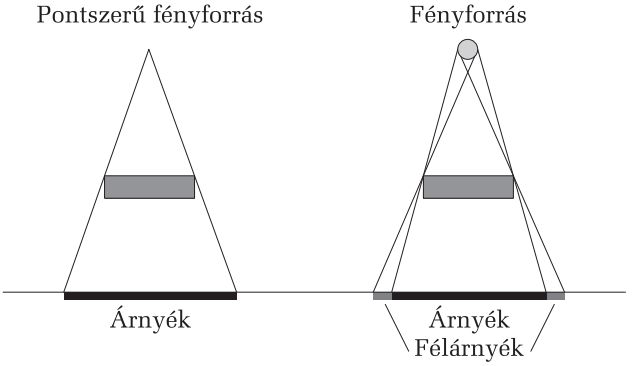
ahol $k_{t_{\min}}$ és $k_{t_{\max}}$ az objektum minimális, illetve maximális fénytörését jellemzi, N_z a pontba húzott normálvektor z komponense és m egy hatvány, amely az átlátszhatóságot jellemzi (a használt értékek általában 2 és 3).

Ez a képlet meghatározza a felület áttetszési együtthatóját.

1.4.4. Árnyékolás

Ha a megfigyelő egy fényforrás által megvilágított színtér objektumait nézi, a fényforrás pozíciójától különböző pozícióból, az objektumok által létrehozott árnyékokat is megfigyelheti.

Egy árnyék két részből áll: a *valódi árnyék*ból és a *félárnyék*ból. A valódi árnyék sűrű, fekete és jól elkülöníthető határa van. A félárnyék körülveszi a valódi árnyékot. A félárnyékban levő objektumok egy kis fényt kapnak a fényforrástól. A pontszerű fényforrások csak valódi árnyékot hoznak létre.



1.50. ábra. *Árnyék és félárnyék*

Az árnyékok meghatározása hasonló feladat, mint az objektumok láthatóságának a meghatározása (lásd *A sugárkövetési algoritmus* című fejezetet). Ezért

egy árnyékolt kép létrehozása a látható felületek kétszeri meghatározását jelenti: egyszer a fényforrások pozíciójából, majd a megfigyelő pozíciójából figyelve a színteret.

Két típusú árnyék létezik: *sajátos* és *nem sajátos árnyék*ok [66]. A sajátos árnyékot az objektum hozza létre úgy, hogy a fény nem jut el az egyik oldalához. A nem sajátos árnyék egy másik objektum által létrehozott árnyék.

A nem sajátos árnyékot meg lehet határozni úgy, hogy fényforrás pozíciójából levetítjük azokat az oldalakat, amelyek nincsenek sajátos árnyékokkal árnyékolva. Az így kapott sokszögek megadják a nem sajátos árnyékot.

Egy jobb módszer az, ha az objektum körvonalát vetítjük le a fényforrás pozíciójából. Egy felület pontja, amely látható a megfigyelő pozíciójából, de a fényforrásából nem, az árnyékolási intenzitással vagy más objektumoktól származó intenzitással lesz megjelenítve.

Egy felület P pontjában a fény intenzitásának a kiszámítása a következőképpen alakul:

$$I_\lambda = I_{a\lambda} \cdot k_a + \sum_{j=1}^{m} S_j \cdot f_{att_j} \cdot I_{i\lambda_j} \cdot \left(k_d \cdot \left(L_{u_j} \cdot N_u\right) + k_s \cdot \left(R_{u_j} \cdot V_u\right)^n\right),$$

ahol $S_j = 0$, ha a fény a j fényforrásból nem ér el a P pontba, és $S_j = 1$, ha a fény a j fényforrásból elér a P pontba.

1.5. A modellezés

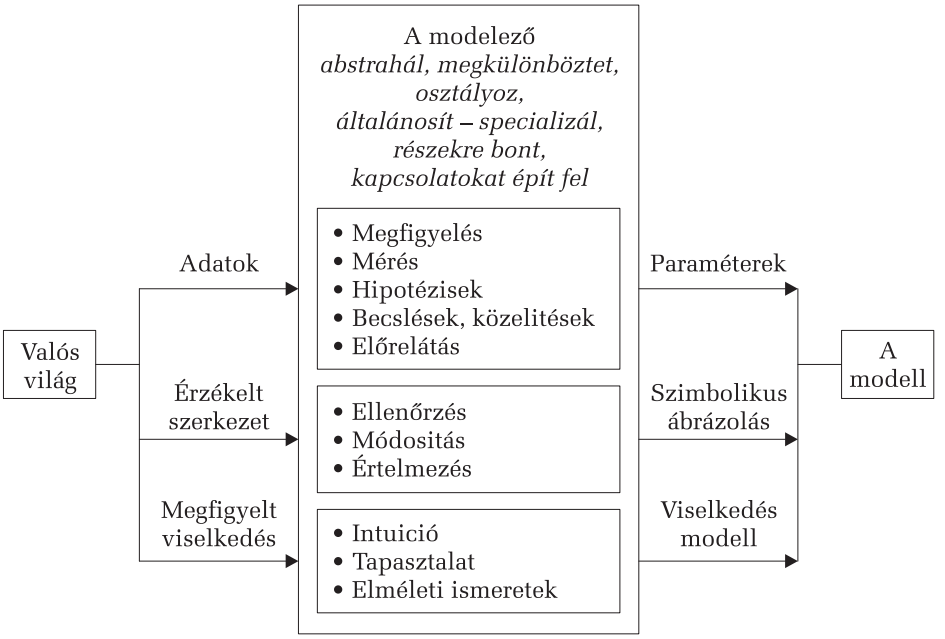
A generatív számítógépes grafikában és a képfeldolgozás során nem a valódi objektumokat (valóságbeli tárgyakat), hanem azok egy *modell*jét dolgozzuk fel.

A *modellezés* során a valós tárgyakból entitásokat absztrahálunk. Az ember a körülötte lévő tárgyakat, valós entitásokat észreveszi, leegyszerűsíti, megkülönbözteti és rendszerezi. A végső cél a bonyolult rendszer megismerése, működésének megértése, a felhasznált eszköz pedig a modellezés. A modellezés során az ember tulajdonképpen egy alapvető, elemi gondolatmenetet (algoritmust) használ, amelynek segítségével absztrahál, megkülönböztet, osztályoz, általánosít – specializál, részekre bont és kapcsolatokat épít fel.

Az *absztrakció* az a szemléletmód, amely segítségével egy végtelenül bonyolult rendszert leegyszerűsítünk úgy, hogy csak a lényegre, a cél elérése érdekében feltétlenül szükséges részekre koncentrálunk. Az absztrahálás tehát azt jelenti, hogy elvonatkoztatunk a számunkra pillanatnyilag nem fontos, közömbös információktól, és kiemeljük az elengedhetetlen fontosságú részleteket.

A megkülönböztetés és az osztályozás szinte automatikus folyamat. Az entitásokat a számunkra lényeges tulajdonságaik, viselkedési módjuk alapján megkülönböztetjük és kategóriákba, osztályokba soroljuk őket, oly módon,

hogy a hasonló tulajdonságokkal rendelkező entitások egy osztályba, a különböző vagy eltérő tulajdonságokkal rendelkező entitások pedig külön osztályokba kerülnek. Az osztályozás folyamata tulajdonképpen az *általánosítás* és a *specializálás* műveleteinek segítségével valósul meg. Az entitások között állandóan hasonlóságokat vagy különbségeket keresünk, hogy ezáltal bővebb vagy szűkebb kategóriákba, osztályba soroljuk őket. Minden entitás valamilyen osztály *példánya*, rendelkezik osztályának sajátosságaival, átveszi annak tulajdonságait.



1.51. ábra. *A modellezés folyamata*

A generatív számítógépes grafikában a feldolgozott grafikus objektumokat (testek, felületek, alakzatok, görbék stb.) *modell-*, *objektum-* vagy *színte*rekben (*scene*) írjuk le matematikai eljárások segítségével. A modellterek általában két- vagy háromdimenziós koordináta-rendszerek (2D, 3D).

A pixelgrafika modelltere kétdimenziós egész koordináta-rendszer.

A vektorgrafika modelltere két- vagy háromdimenziós valós euklidészi tér – lebegőpontos koordinátaértékekkel.

1.5.1. 3D modellezők

A 3D modellezők olyan alkalmazások, amelyek a valóságból vett objektumokat dolgoznak fel. Az objektumok úgy viselkednek vagy úgy néznek ki,

hasonló struktúrájuk, esetleg néhány fizikai tulajdonsággal rendelkeznek (tömeg, térfogat), mint a valóságban. A 3D modellezők moduláris felépítésűek. Egyik modul kezeli az objektumok térbeli modellezését, egy másik a fizikai tulajdonságokat, egy harmadik a vetítést, megvilágítást (perspektíva, fényforrások, atmoszféra, a fény terjedésének törvényei), és esetleg létezik egy animációs modul is, amely az objektumok vagy fényforrások mozgását valósítja meg. A modellezők legújabb generációi tartalmaznak egy interaktivitás-modult is, amely segítségével a felhasználó dinamikusan közbeléphet és interaktívan módosíthatja az objektumok tulajdonságait.

A [20] alapján a következő modellező szoftvereket tudjuk megkülönböztetni:

- *Felületmodellező*: egy „vázra” rányújtunk egy „bőrt”. A váz sokszögekből vagy görbékből állhat. Az eredmény egy struktúra, amelynek térfogata van, de nincsenek fizikai tulajdonságai: tömeg, sűrűség stb.
- *Szilárdtest-modellező*: több modellezési lehetőséget nyújt, viszont a számítási idő észrevehetően megnő. A szilárdtest-modellezőt nehezebb kezelni, mint a felületmodellezőt, viszont le tudják kezelni a sűrűséget, tömeget, súrlódási tényezőket, gravitációs erőket, ütközéseket stb., így a 3D objektumok viselkedését közelebb hozzák a valósághoz.
- *Sokszög-modellező*: konvex sokszögek összességéből állítja össze a 3D objektumot. A hátránya, hogy a görbe felületeket mindig sík felületekkel közelíti meg. Előnye, hogy egyszerűbb számításokat kell elvégezni, és az algoritmusok is sokkal egyszerűbbek, gyorsabbak.
- *Spline-görbéken alapuló modellező*: a 3D objektumokat (felületek vagy szilárd testek) matematikailag könnyen leírható görbékkel közelítik meg (spline görbék). Nagyobb rugalmasságot és pontosságot nyújtanak, mint a sokszög-modellezők. A legtöbb modellező program egyenletes splineokon alapszik, egy modernebb verzió a NURBS (*Non-Uniform Rational B-Splines*) modellező. A NURBS esetében a pontok nem egyenletesen vannak elosztva a görbén, így sokkal jobban meg tudják közelíteni a modellezett objektumokat. A pontosság azonban sok matematikai műveletet igényel, és a renderelési algoritmusuk kivitelezése még nehézkes és lassú.

1.5.2. 3D testek modellezésének módszerei

Minden 3D modellező szoftver rendelkezik egy pár alaptulajdonsággal, ezeket foglaljuk össze a következőkben [20]:

- *Primitívek használata*: minden modellező alkalmazás ismer néhány 3D primitív objektumot: gömböket, kúpokat, hengereket, kockákat és néha komplexebb objektumot is, mint a tórusz, dodekaéder stb. Ezekből a primitívekből komplexebb objektumokat lehet előállítani.

- *Extrudálás (extrusion)*: a 3D modellezés egyik alaptechnikája, amely abból áll, hogy egy 2D objektumot (kör, téglalap, sokszög) eltolunk egy görbe vagy egyenes mentén a harmadik dimenzióba, így létrehozunk egy 3D objektumot (pl. egy kör extrudálásával egy egyenes mentén egy hengert kapunk). Az extrudálás parametrizálásával (a 2D objektum méretezése, forgatása) a végső objektum formáját befolyásolhatjuk.
- *Forgástest kialakítása (lathing)*: egy görbe tengely körüli forgatásával generálunk 3D testeket.
- *„Bőrözés" (skinning)*: egy több görbéből álló vázra kihúzunk egy felületet (hasonló egy hajó építéséhez, ahol előbb a metszeti síkokat építik meg, majd ezek haránt irányú felületekkel lesznek összekötve). A szerkezet minden *bordája* egy ponthalmaz, amelyen keresztülmegy az adott felület.
- *Vertex-szerkesztés*: megadja azt a lehetőséget, hogy a 3D objektum bármely pontjának a pozícióját lehessen változtatni. A vertex-szerkesztés kétfajta: *statikus,* amikor a vertexek egymástól függetlenül mozognak, és *dinamikus*, amikor egy vertex elmozdítása maga után vonja a szomszédos vertexek kisebb mértékbeli elmozdulását (így egy simább felületet kapunk).
- *Boole-műveletek*: a legtöbb számítást igénylő műveletek, két vagy több 3D objektum egyesítéséből, különbségéből, metszetéből komplexebb 3D formát állítunk elő.

1.5.3. Képek generálása

A modellről generált képek előállítását, szintézisét *renderelés*nek (*rendering*) nevezzük. Matematikai számítások alapján meghatározzuk a fénysugarak útját a különböző fényforrásoktól, ezek viselkedését a különböző objektumokkal való találkozáskor (az optikai tulajdonságuk függvényében), és végezetül mindezek hatását a megfigyelőre. Ugyanakkor az atmoszférikus effektusok (köd, füst, felhő, szórt háttérfény stb.), valamint a fényforrások alapján az objektumok által vetett árnyékokat is ki kell számítani.

A képek rendelerésére több algoritmus ismeretes (Fiat, Phong, Gouraud, Metál, globális illumináció, radiosity), de ami a teljesítményt illeti, a *Ray-Tracing (sugárkövetés)* algoritmus a legjobb (a legpontosabb és legtermészetesebb képeket nyújtja), annak ellenére, hogy talán a leglassúbb.

A renderelés négy tényező alapján valósul meg:

- 3D modellezés,
- a felület tulajdonságainak meghatározása (anyag, tükröződés, áttetszőség, fényesség stb.),
- a fényforrások és a kamera helyzete,

– animáció esetén az elmozdulási görbék és a különböző objektumok eseményeinek (átmeneteinek) a meghatározása (pl. felület színének a változtatása).

A következő egyszerű példaprogram a POV-Ray 3D modellező leírónyelvben egy gömböt és egy tóruszt definiál. A program elején beállítjuk a kamera (megfigyelő) helyét és irányát, a háttérszínt, és egy fényforrást is definiálunk.

A leírónyelv szintaxisa igen egyszerű, parancsokkal és paraméterekkel állíthatjuk be a színtér objektumait.

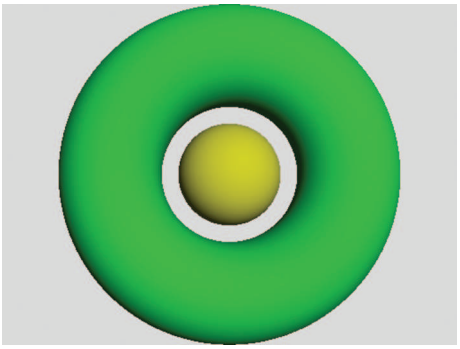
A POV-Ray rendszer a leírtak (program) alapján képet generál.

```
#include "colors.inc"

camera {
    location <0, 0.1, -25>
    look_at 0
    angle 30
}

background {color Gray75}
light_source {<300, 300, -1000> White}

torus {
    3.5, 1.5
    rotate -90*x
    pigment {Green}
}

sphere {
    <0, 0, 0>, 1.5
    texture {
      pigment {color Yellow}
    }
}
```

A képszintézis grafikus *csővezeték (graphics pipeline)* által valósul meg. Megjelenítés céljából a grafikus primitíveken végzendő elemi műveleteket (transzformációk, vetítés, vágás stb.) a rendszer egymás után, sorozatban (csővezetékben) végzi el.

Általánosan azt mondhatjuk, hogy a feldolgozás az *alkalmazás–parancs–geometria–raszterizálás–textúrázás–fragmentálás–megjelenítés* útvonalon történik.

Az *alkalmazás* szint tartalmazza a szimuláció leírását, az eseménykezelést, az adatstruktúrákat, algoritmusokat, esetleges adatbázisokat, primitívek generálását és más eszközöket.



1.52. ábra. *Testek egyszerű modellje POV-Ray-ben*

A *parancs* szint a végrehajtható, értelmezhető grafikus parancsokat tartalmazza.

A *geometria* szint a modell mértani leírását, a mértani műveleteket tartalmazza.

A *raszterizálás* által kapjuk meg a képernyőn ábrázolható pixeleket, a vektorgrafika átalakul pixelgrafikává.

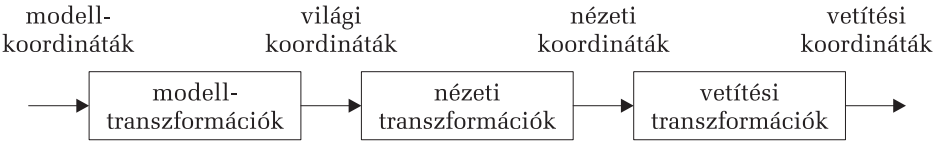
Ez tökéletesedik ki a *textúrázás*sal. A megrajzolt felületekre képeket húzhatunk rá.

A *fragmentálás*sal véglegesen eldől minden pixel színe. Itt alkalmazzuk az atmoszférikus effektusokat (pl. köd), itt határozzuk meg az átlátszóságot stb.

A folyamatot a *megjelenítés* zárja, az előállított kép megjelenik a képernyőn.

A parancs és a geometria szintet összevonva *objektum* vagy *vertex* szintnek nevezzük, a textúrázás és a fragmentálás szintet pedig *kép* vagy *fragmentum* szintnek.

Általában minden szintnek más koordináta-rendszere van, más tulajdonságok és műveletek érvényesek rá, és másképp támogatja a hardver. Ha a koordináta-rendszereket és a transzformációkat szeretnénk csővezetékbe helyezni, akkor az 1.53. ábrán látható rendszert kapjuk.



1.53. ábra. *Koordináta-rendszerek és transzformációk*

Szükség szerint a szinteket alszintekre bonthatjuk, ha egy-egy műveletet ki szeretnénk emelni, például a geometria 3D-ben *modell-transzformációk–triviális elvetés–illumináció–nézeti transzformációk–vágás–vetítés* alszintekre bontható.

A modellező szoftverek a hardver grafikus csővezetékére építve saját logikai csővezetékeket állíthatnak fel a saját funkcionalitásuk megvalósítása érdekében, így ezek esetenként eltérhetnek egymástól, csak nagyvonalakban követik az elvi csővezeték modellt.

2. FEJEZET

A SZÁMÍTÓGÉPES GRAFIKA ALAPJAI

2.1. A grafikus hardver és szoftver

2.1.1. A grafikus hardver

A modern elektronikus számítógépek működési elvét Neumann János fogalmazta meg 1946-ban. Az elvek a következők:

1. A számítógép legyen soros működésű: a gép az egyes utasításokat egymás után, egyenként hajtsa végre.

2. A számítógép a kettes számrendszert használja, és legyen teljesen elektronikus: a kettes számrendszert és a rajta értelmezett aritmetikai, illetve logikai műveleteket könnyű megvalósítani kétállapotú áramkörökkel (pl.: 1 – magasabb feszültség, 0 – alacsonyabb feszültség, 1 – be van kapcsolva, 0 – nincs bekapcsolva).

3. A számítógépnek legyen belső memóriája: a belső memóriában tárolhatók az adatok és az egyes számítások részeredményei, így a gép bizonyos műveletsorokat automatikusan el tud végezni.

4. A tárolt program elve: a programot alkotó utasítások kifejezhetők számokkal (gépi kód), azaz adatként kezelhetők, és ezek is a belső memóriában tárolhatók, mint bármelyik más adat. Ezáltal a számítógép önállóan képes működni, hiszen az adatokat és az utasításokat egyaránt a memóriából olvassa ki.

5. A számítógép legyen univerzális: a számítógép különféle feladatainak elvégzéséhez nem kell speciális berendezéseket készíteni.

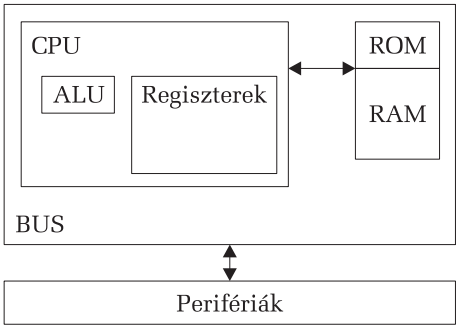
A Neumann-féle számítógép vázlatos felépítése:

- A központi vezérlő egység (CPU – *Central Processor Unit*) feladatai:
  - A számítógép működésének irányítása, vezérlése.
  - Adatforgalom irányítása.
  - Utasítások értelmezése és végrehajtása.
  - Operandusok címének kiszámítása.
- Regiszterek:
  - Gyors elérésű, közvetlenül címezhető, rendszerfelhasználók által osztottan hozzáférhető tárolók.
  - Számuk gépfüggő.
  - Használatuk a gyors elérés miatt csökkenti a program futási idejét.
- Az aritmetikai, logikai egység (ALU – *Arythmetical Local Unit*) feladatai:

  - Adott adatokkal végrehajtja az aritmetikai és logikai műveleteket (+, -, *, /, AND, OR, NOT, stb.).
  - Saját regisztereik (akkumulátoraik) lehetnek.
- A csak olvasható memória (ROM – *Read Only Memory*) tulajdonságai:
  - A gép futásához szükséges alapprogramokat tartalmazza.
  - Kikapcsoláskor is megőrzi tartalmát.
  - Lehet fix, cserélhető, újraprogramozható és törölhető.
  - Tartalmazhatja az operációs rendszert (kis rendszerek).
- A memória (RAM – *Random Access Memory*):
  - Írható, olvasható, véletlen hozzáférésű tár.
- Az adatbusz (BUS):
  - Segítségével valósul meg a kommunikáció a számítógép különböző alkotóelemei között.
- Perifériák:
  - Ki/Beviteli eszközök (képernyő, billentyűzet, egér stb.)
  - Háttértárolók (merevlemez, mágneses lemez, CD-ROM, DVD stb.)

Az IBM kompatibilis személyi számítógépek grafikus hardverei a *perifériák* kategóriába tartoznak. Bemeneti eszközök a *billentyűzet*, az *egér*, a *spaceball*, a *digitalizáló tábla*, a *szkenner*, *digitális fényképezőgép*, *kamera*, *botkormány* stb. Kimeneti eszközök a *képernyő (monitor)*, a különféle *nyomtató*k, *rajzgép*ek.

Mindezen eszközök speciális *meghajtók*kal, *illesztőprogramok*kal *(driver)* vezérelhetők, ezek valósítják meg az adatátvitelt és a magas szintű programozásukat is.



2.1. ábra. *A Neumann-féle számítógép vázlatos felépítése*

A generatív számítógépes grafika szempontjából számunkra a *képernyő (display, monitor),* valamint a *grafikus kártya* a fontos.

Három típusú képernyő létezik:

- CRT,
- LCD / TFT,
- PDP.

A *CRT (Cathode Ray Tube)* a hagyományos katódsugárcsöves képernyő. Az első működőképes televíziót 1926. január 26-án mutatták be Londonban. Az első színes adást 1928. július 3-án továbbították nagy távolságra. A technika feltalálója Karl Ferdinand Braun (1850–1918) volt, aki 1897-ben már meg tudott így egy képpontot jeleníteni. (Ezért régi neve a Braun-cső.) A töltéscsatolt elvű CRT tévé és kamera feltalálója (1928-ban) Tihanyi Kálmán (1897–1947). A CRT monitorban egy katódsugárcső található, amelynek az egyik végén elektronágyú, a másik végén foszforral bevont képernyő található. Az elektronágyú elektronnyalábot lő ki, ezt a mágneses mező irányítja. Az elektronnyaláb a foszforborításba ütközik és felvillan, majd elhalványodik. Ha elég gyorsan követik egymást az elektronnyalábok, akkor az a pont nem halványodik el. Tehát az elektronágyúk írnak a képernyőre a számítógép utasításának megfelelően, balról jobbra, egy másodperc alatt többször is frissítve a képpontokat. Azt, hogy másodpercenként hányszor frissíti a képpontokat, képfrissítési frekvenciának nevezzük. Ezt Hertzben adjuk meg. A mai monitorok 60–130 hertzesek. A monitor az additív színkeverés elve alapján működik, a három alapszínhez (R, G, B) tartozik egy-egy elektronágyú.

Az *LCD (Liquid Crystal Display)* folyadékkristályos képernyő. A folyadékkristályos kijelzők őse a kvarcórák kijelzője. Folyadékkristállyal már 1911 óta kísérleteznek, működő LCD monitor az 1960-as években készült először. Az LCD monitor két belső felületén mikronméretű árkokkal ellátott átlátszó lap közé folyadékkristályos anyagot helyeznek, amely nyugalmi állapotában igazodik a belső felület által meghatározott irányhoz, így csavart állapotot vesz fel. A kijelző első és hátsó oldalára egy-egy polárszűrőt helyeznek, amelyek a fény minden irányú rezgését csak egy meghatározott síkban engedik tovább. A csavart elhelyezkedésű folyadékkristály különleges tulajdonsága, hogy a ráeső fény rezgési síkját elforgatja. Ha hátul megvilágítják a panelt, akkor a hátsó polarizátoron átjutó fényt a folyadékkristály elforgatja (innen ered a Twisted Nematic, TN megnevezés), így a fény az első szűrőn átjut, és világos képpontot kapunk. Ha kristályokra feszültséget kapcsolunk, nem forgatják el a fényt, az eredmény pedig fekete képpont. A polárszűrő elé már csak egy színszűrőt kell helyezni. Előfordulhat a gyártás tökéletlensége miatt, hogy a képernyőn halott vagy „beragadt” képpontokat találunk. A TFT (*Thin Film Transistor*) vékonyfilm tranzisztor. Az LCD technológián alapuló TFT minden egyes képpontja egy saját tranzisztorból áll, mely aktív állapotban elő tud állítani egy világító pontot. Az ilyen kijelzőket gyakran aktív-mátrixos LCD-nek is szokás nevezni.

A PDP (Plasma Display Panel) plazmakijelzők első, monokróm típusát 1964-ben a Plató Computer System készítette el, Gábor Dénes plazmával kapcsolatos kutatásai nyomán. Az első plazmatelevíziót a Pioneer mutatta be 1997-ben. Jelenleg is folyik a gyártók versenye a minél nagyobb képátlóért: már a 100”-et is bőven meghaladják a legnagyobb kijelzők. A PDP működése az LCD-nél is egyszerűbb. A cél az, hogy a három alapszínnek megfelelő képpont fényerejét szabályozni lehessen. A PDP-nél a képpontok a CRT-hez hasonlóan látható

2.2. ábra. *Képernyőtípusok: CRT, LCD / TFT, PDP*

fényt sugároznak ki, ha megfelelő hullámhosszú energia éri őket. Ebben az esetben a neon és xenon gázok keverékének nagy UV-sugárzással kísért ionizációs kisülése készteti a képpont anyagát színes fény sugárzására, pont úgy, mint a neoncsövekben. Mivel minden egyes képpont egymástól függetlenül, akár folyamatos üzemben vezérelhető, a monitor villódzástól mentes, akár 10 000:1 kontrasztarányú, tökéletes színekkel rendelkező képet is adhat, bármely szögből nézve. [40]

A monitoron ábrázolt kép legkisebb egysége a *képpont* vagy *pixel (picture element)*. Minél több pontból áll egy kép, annál élesebb, szebb a megjelenítés. Ezt a tulajdonságot nevezzük *felbontás*nak. A képernyő felbontását a pixel sorok és oszlopok száma adja meg. Manapság használatos felbontások: 800×600, 1024×768, professzionális rendszereknél 1280×1024, 1600×1200, vagy még nagyobb is. Természetesen egy bizonyos határon túl már nem érzékelhető a különbség. A felbontást általában *pont per hüvelyk*ben (*dots per inch – dpi*) mérik. Ez mutatja meg, hogy egy hüvelykben (2,54 cm) hány képpont található. Mivel a képernyő felbontása alapértelmezés szerint 72 dpi, általában ezzel az értékkel dolgozunk.

2.3. ábra. *ATi Radeon™ HD 4870 videokártya – 512 MB GDDR5 memória; 1,2 teraflops teljesítmény; 750 MHz GPU; PCI Express 2.0 interface; 160 W*

A monitorokat a *videokártyá*k vezérlik. A processzor (CPU) elküldi a videokártyának a megjelenítési utasításokat, adatokat, a videokártya pedig a monitor

számára is értelmezhető jellé alakítja azokat. Az olyan műveleteknek, mint elsimítás, árnyékolás, komoly számítási igényei vannak, ezért a grafikus kártyáknak több feldolgozó egységük, külön grafikus processzoruk (*GPU – Graphics Processing Unit*), illetve jelentős memóriájuk van (64 MB–1 GB, GDDR 2/3/4/5). A videokártya *AGP (Accelarated Graphics Port)* vagy *PCI-Express* porton keresztül csatlakozik az alaplaphoz. A monitorhoz a jelt pedig vagy analóg módon *(D-SUB, D-subminiature)*, vagy digitális módon *(DVI – Digital Visual Interface),* vagy a nagyfelbontású tartalmak miatt kifejlesztett *HDMI (High-Definition Multimedia Interface)* módon küldheti.

2.4. ábra. *A grafikus hardver vázlatos felépítése*

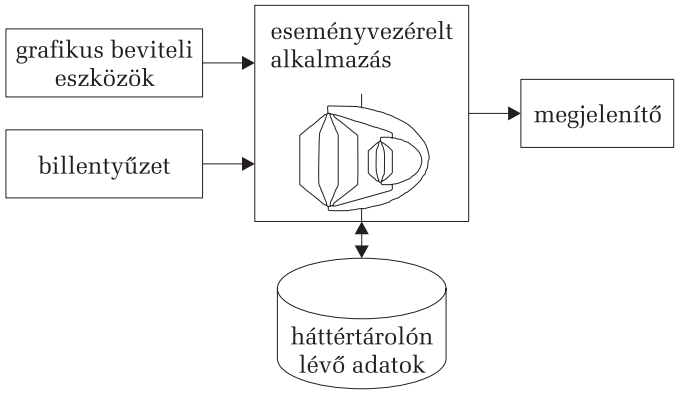
A *Graphics Processing Unit (GPU)* a grafikus vezérlő központi egysége, amely az összetett grafikus műveletek elvégzéséért felelős. A GPU feladata a grafikák létrehozásával és megjelenítésével közvetlenül kapcsolatban hozható magas szintű feladatok átvétele a CPU-tól, hogy annak számítási kapacitása más műveletek elvégzésére legyen felhasználható. A modern GPU-k 2D és 3D műveletek elvégzésére egyaránt alkalmasak, alapműveletei közé tartoznak például a négyzetes mátrixok szorzása (koordináta-transzformáció számítás).

Manapság 2 vezető gyártó van a piacon, az Nvidia és az ATi céget felvásárló AMD.

A modern grafikus hardverek a *grafikus csővezeték (graphics pipeline)* elve alapján működnek. A GPU végrehajt egy grafikus utasítást, de e mellett egy másik egység transzformál, vág, árnyal, textúrát tömörít stb., majd az eredmény megjelenik a különböző bufferekben. Amikor egy pont (vertex) megjelenik a csővezeték bemeneténél, lehet, hogy a transzformációs hardver még az előző elküldött elemen dolgozik.

2.1.2. A grafikus szoftver

A grafikus szoftverek interaktív alkalmazások, a felhasználó a grafikus beviteli eszközök segítségével avatkoznak be a szoftver működésébe, adatokat, utasításokat közölnek, eseményeket váltanak ki. A beviteli eszközöket az operációs rendszer illeszti az alkalmazáshoz. Az eredmények és az eseményekre való reakciók hatása a képernyőn jelenik meg.



2.5. ábra. *A grafikus szoftver vázlatos felépítése*

2.1.2.1. Állományformátumok

A beviteli eszközökön kívül grafikus adatok lehetnek különböző háttértárolókon (*merevlemez*, *optikai lemezek*, *hajlékony lemezek* stb.) is, ezek többnyire képeket vagy koordináta-, vertexinformációkat tárolnak.

A képek minősége a felbontáson kívül nagymértékben függ a felhasznált színek számától is. De minél több színt tartalmaz egy kép, annál több információra van szükség a tárolásához, ami néha gondot jelenthet. Így van ez a felbontás esetén is. Választanunk kell tehát a jó minőség és a kis helyfoglalás között, ezért ajánlatos egyfajta középutat keresnünk.

Négy alapvető színmód létezik:

- fekete–fehér (*monochrome*),
- szürkeárnyalatos (*grayscale*),
- palettás (*indexed color*),
- valódi színezetű (*TrueColor*).

A *fekete-fehér* képek minden pontja két értéket vehet fel: 1-est (fekete) vagy 0-t (fehér). Így bármely pixel tárolására elegendő 1 bit. A *szürkeárnyalatos* képek, amint a nevük is sugallja, a szürke szín 8 biten tárolt 256 különböző árnyalatának az ábrázolására képesek, ami gyakorlatilag a fekete-fehér fényképnek felel meg. A *palettás (színindex módú)* képek egy 256 elemű táblázatot (palettát) tartalmaznak, amelyben a különböző színek számkódjai szerepelnek, így minden pixel esetében csak azt kell tárolni, hogy az ő színe a paletta hányadik elemének felel meg (8 bit). Ez a színmód nagyon elterjedt (például az interneten), mert segítségével színes képeket viszonylag kis tárkapacitással is elmenthetünk. A *valódi színezetű* képek 24 biten tárolják az egyes képpontok színét, ezáltal 16,7 millió különböző árnyalatot ábrázolhatnak, ami már tökéletes színátmenetet jelent az emberi szem számára.

Megfigyelhettük, hogy míg a fekete-fehér képek ábrázolására elegendő volt képpontonként 1 bit, addig a valódi színezetű képek 24-szer több helyet igényelnek. Ezért ajánlatos azt a legkisebb típust választani, amelyik még éppen megfelel.

Szerencsére van más megoldásunk is. A gyakorlatban számos olyan matematikai algoritmus létezik, amellyel jelentősen csökkenthetjük grafikus állományaink méretét. Ezeket az eljárásokat nevezzük *tömörítés*nek. A tömörítés mértéke függ az állomány tartalmától is: minél részletgazdagabb a kép, annál nehezebb a tömörítés.

Két tömörítési fajta ismert: a *veszteség nélküli* és a *veszteséges*.

Veszteség nélküli tömörítés esetén az állomány mérete lecsökken, de az eredeti kép bármikor tökéletesen visszanyerhető. Ez az eljárás 10% és 80% közötti tömörítésre képes. A veszteséges tömörítés kihasználja az emberi szem tökéletlenségét, és azonosnak tekinti az egyes közeli vagy alig különböző színeket, így hatékonysága elérheti a 95%-ot. Az ilyen eljárásoknál megadhatjuk a veszteség mértékét, azaz választhatunk a legjobb minőség (leggyengébb tömörítés) és a leggyengébb minőség (legjobb tömörítés) között.

Az egyes cégek igényeiknek megfelelően saját képformátumokat dolgoztak ki. Az ilyen állományok általában tartalmaznak egy fejlécet (a formátum, szín, méret, paletta stb. tárolására) és magát a képet pixeles, vektoros vagy metaállomány formájában. A metaállományban egy időben tárolhatók pixeles és vektoros grafikák is. A legelterjedtebb formátumok a következők [4]:

BMP (*Windows Bitmap* és *RLE*) – A BMP formátumot a Microsoft fejlesztette ki. A Windows belső pixeles képformátuma, amelyet szinte minden Windows alatt futó program képes értelmezni. Az összes színmódot támogatja, sőt a 4 és 8 bites képek esetében RLE tömörítésre is lehetőségünk van. Nyomdai használatra nem alkalmas, mivel a CMYK-színmodellt nem ismeri, csak a vonalast, szürkeárnyalatost, palettást és RGB-t.

CompuServe GIF (*Graphic Interchange Format*) – A CompuServe által kifejlesztett GIF kifejezetten az internet számára készült 8 bites formátum, azaz legfeljebb 256 szín megjelenítésére képes. Palettás kép, ezért támogatja a vonalas és a szürkeárnyalatos színmódokat is. Veszteségmentes tömörítési algoritmusának (LZW) köszönhetően alkalmas hálózati felhasználásra. Előnye, hogy egy kiválasztott szín segítségével a kép egyes részei átlátszóvá tehetők (így képünk látszólag nem csak téglalap alakú lehet). Alkalmas *váltottsoros megjelenítés*re, valamint animációk tárolására is. A váltottsoros (*interlaced*) kirajzolásnál előbb a kép minden nyolcadik sora jelenik meg, majd ezt „finomítja” folyamatosan a megjelenítő. Ez a módszer sokkal gyorsabb, mert a felhasználó már a betöltéskor dönthet, hogy végigvárja-e vagy továbblép. A GIF-et kis helyigénye és hasznos szolgáltatásai tették népszerűvé. GIF animációkat állóképek összetűzésével készíthetünk, amelyek megjelenítésére ma már a legtöbb böngésző képes – hátrányuk, hogy sok helyet igényelnek.

JPEG (*Joint Photographic Experts Group*) – A JPEG (JPG) napjainkban az egyik legelterjedtebb formátum, főleg fényképek tárolására használják. A szürkeárnyalatostól a TrueColor-ig minden modellt támogat. A JPEG tömörítés veszteséges, de sokkal hatékonyabb, mint a GIF képek sűrítése. Beállíthatjuk a tömörítés mértéket, ami fordítottan arányos a minőséggel. Az alkalmazott színmodell szürkeárnyalatos, RGB vagy CMYK lehet, ezért a nyomdákban is használható. Népszerűségét az is igazolja, hogy kezdi kiszorítani a világhálón eddig egyeduralkodó GIF állományokat. Bitek száma pixelenként: 8 vagy 24.

Adobe Photoshop – Az Adobe Photoshop képfeldolgozó program saját állományformátuma (PSD), amely egyesíti az előbbiek összes tulajdonságait. A PSD állományokban lehetőség van több réteg tárolására, illetve a beállítások mentésére, így a későbbi módosítások során munkánkat ott folytathatjuk, ahol abbahagytuk. A formátum a rétegeken kívül egy összetett képet is tartalmaz, amelyet a – főleg más programokkal való – gyors megtekintésnél használ. Hátránya, hogy nem alkalmaz semmilyen tömörítést, így mérete a több réteg miatt lényegesen megnőhet. Ismeri a fekete-fehér, szürke árnyalatos, palettás, duplex, RGB, CIELAB, CMYK, 16 bit/csatorna színmodelleket; 1, 4, 8 és 24 bites színmódokat tud.

Acrobat PDF (*Portable Document Format*) – A PDF az Adobe cég terméke, amelyet elsősorban az Acrobat Reader program használ. Népszerűségét annak köszönheti, hogy egyszerre képes kezelni a pixeles és a vektoros képeket is (tehát metaállomány). Többféle tömörítési algoritmust használ (LZW, JPEG, ZIP, CCITT, RLE), mindig az adatok típusának megfelelő módszer szerint. Másképp fogalmazva: különbözőképpen tömöríti a képeket, a szövegeket és egyéb információkat, így egyrészt hatékonyabbá teszi a tömörítést, másrészt pedig szétválasztja az egyes objektumokat. Ezért szkennelt oldalak szövegeihez akár hivatkozást (linket) is rendelhetünk. Nyomdai munkálatokra kitűnően alkalmas, és népszerű az elektronikus sajtóban is. Ismeri a fekete-fehér, szürke árnyalatos, palettás, RGB, CIELAB, CMYK színmodelleket; 1, 4, 8 és 24 bites színmódokat tud.

PNG (Portable Network Graphics) – Képek tárolására, veszteségmentes tömörítésére alkalmas állományformátum. A PNG egy viszonylag fiatal állományformátum, a GIF utódaként emlegetik. Elsősorban a számítógépes hálózatokban lévő képek átvitelére szolgál. Tömörítésre egy *deflation* nevű algoritmust (az *LZ77* egy módosított változatát) használ. A PNG számos előnnyel rendelkezik a GIF-hez képest: alfa-csatornákat használ (RGBA színmodell), amelyek a fokozatosan átlátszó képeket teszik lehetővé; γ-korrekciót használ, amely a képek fényességét (elméletben) függetleníteni tudja a megjelenítéstől (tehát a színek ugyanúgy néznek ki nyomtatásban és eltérő képességű kijelzőkön); egyik újdonsága a képek fokozatos megjelenítésének módja (*Adam-7*), amely lehetővé teszi, hogy lassú átvitel vagy nagy méretű kép esetén már a letöltés elején látni lehessen elnagyoltan (kis felbontásban) a kép tartalmát, ez a letöltés előrehaladtával fokozatosan nyeri el részletgazdagságát. A GIF-hez képest viszont hátránya,

hogy nem támogatja a több képet tartalmazó állományokat, s így az animációt sem. 1, 4, 8, 24, 32 és 48 bites színmódokat támogat.

2.1.2.2. A BMP állományformátum

A BMP állományok három – vagy paletta használatának esetén négy – elkülöníthető részből állnak:

- *Állományfejléc*: az állományra vonatkozó alapvető adatokat tárolja.
- *Információs fejléc*: az eltárolt kép jellemzőit írja le (felbontás, színmélység stb.)
- *Paletta (ha van)*: az eltárolt kép által használt színek RGB kódjait sorolja fel.
- *Bittérkép*: a kép tényleges tárolási helye, ahol képpontról képpontra jegyzik fel azok színeit, vagy paletta esetében a paletta indexét.

Az állományfejléc 14 byte hosszon tárolja a következő adatokat:

- 0-tól 2 byte-on a szignatúra. A képkezelő alkalmazások ezen két byte alapján azonosítják be a BMP formátumot. Az első byte-on a B (66), a másodikon az M (77) ASCII-kódját helyezik el.
- 2-től 4 byte-on az állomány mérete byte-okban.
- 6-tól 4 byte szabad terület. Az egyes képszerkesztő, illetve generáló szoftverek saját bejegyzésüket helyezhetik el ide.
- 10-től 4 byte-on a bittérkép kezdőcíme, amely megadja, hogy hányadik byte-tól kezdődik a bittérkép leírása az állományon belül (az első byte sorszáma a 0-s). Ha nincs paletta, ez mindig 54.

Az információs fejléc 40 byte hosszon tárolja a következő adatokat:

- 14-től 4 byte: információs fejléc mérete, ez mindig 40.
- 18-tól 4 byte-on a kép szélessége (pixelben).
- 22-től 4 byte-on a kép magassága (pixelben).
- 26-tól 2 byte-on a használt színsíkok, ez mindig 1.
- 28-tól 2 byte-on a színmélység, amely megadja, hogy a bittérképben hány bit vonatkozik egyetlen képpont színére. Jellemző értékei: 1 (1 bites színindexek, két szín); 4 (4 bites színindexek, legfeljebb 16 szín); 8 (8 bites színindexek, legfeljebb 256 szín); 24 (24 bites RGB színkódok, TrueColor); 32 (32 bites mód).
- 30-tól 4 byte-on a bittérképen alkalmazott tömörítés típusa: 0 – nincs tömörítés, 1 – 8 bites szakaszhossz-tömörítés (RLE8), 2 – 4 bites szakaszhossz-tömörítés (RLE4), 3 – a 16 és 32 bites módban a bitmező (*bitfield*), 4 – a bittérkép JPEG adatot tartalmaz, 5 – a bittérkép PNG adatot tartalmaz.
- 34-től 4 byte-on 0, ha nincs tömörítés, különben a bittérkép mérete (byte-ban), amely nem tévesztendő össze az állomány méretével.
- 38-tól 4 byte-on a kép vízszintes felbontása (pixel/méterben); nyomtatásnál praktikus érték.

- 42-től 4 byte-on a kép függőleges felbontása (pixel/méterben).
- 46-tól 4 byte-on a palettában definiált színek száma (= 0, ha nincs paletta, vagy ha a paletta színeinek száma egyenlő a maximális színek számával a színindex módban).
- 50-től 4 byte-on megadja, hogy a paletta színei közül hányat használ fel a bittérkép. Szinte mindig megegyezik az előbbi értékkel (0 az értéke, ha nincs paletta).

A palettát színindex módban használja (a színmélység legfeljebb 8 bit/pixel). Ekkor a definiált színek számaszor ismétlődik a következő 4 byte:

- 1 byte: kék intenzitás (B), 0–255 közötti érték,
- 1 byte: zöld intenzitás (G), 0–255 közötti érték,
- 1 byte: piros intenzitás (R), 0–255 közötti érték,
- 1 byte: szabad terület (általában 0, de az egyes képszerkesztő, generáló szoftverek saját bejegyzést helyezhetnek itt el).

A bittérkép a kép képpontjait sorfolytonosan tárolja, amely alapvetően kétféleképpen történhet:

- Ha a színmélység 8 vagy ennél kisebb, akkor színindexeket sorol fel, amelyek a paletta színeire mutatnak (a paletta első színe kapja a 0-s indexet). Minden színindex egy képpontot ír le.
- Ha a színmélység 24 bites vagy ennél nagyobb, akkor RGB színkódokat sorol fel, és ilyenkor nincs az állományban paletta. Minden (3×8 bites) RGB színkód egy-egy képpontot ír le.

A bittérkép jellemzői:

- Az RGB számhármasokat (mind a bittérképben, mind a palettában) B, G, R sorrendben tárolja.
- A kép soronként lentről felfelé haladva tárolódik (vagyis a kép legalsó sora kerül a bittérkép legfelső sorába).
- A bittérképen belül a képnek megfelelő sorokat szükség szerint 0-s bitekkel egészítik ki úgy, hogy minden sorhoz néggyel osztható számú byte tartozzék.

Bittérkép (**.BMP** állomány) beolvasására és felhasználására példát láthatunk a 4.2. (*DirectX*) fejezetben.

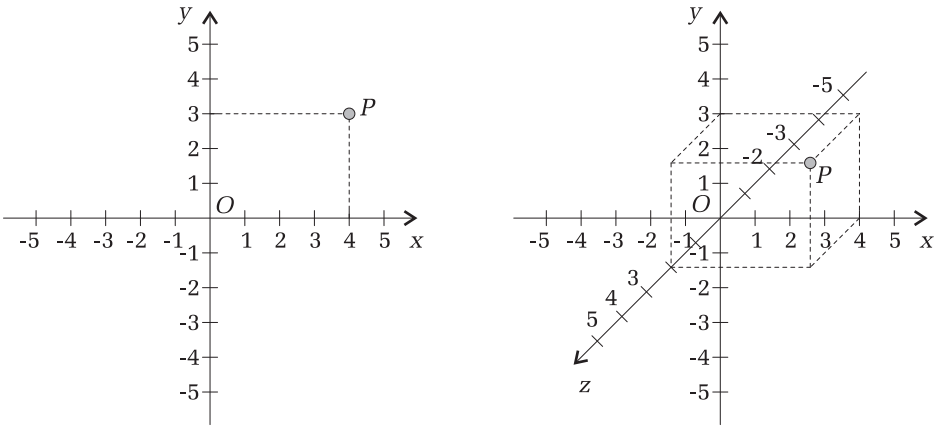
2.2. Koordináta-rendszerek, transzformációk

2.2.1. Descartes-féle koordináták

Egy tetszőleges pont helyzete a térben vagy a síkban *koordináták* (egymástól független méretek) segítségével adható meg. A koordináta-rendszer egy sík vagy egy tér, amelyben egy kezdőpontot és tengelyeket jelölünk ki, és ezektől mérhetők a koordináták.

A Descartes-féle koordináta-rendszer egymásra merőleges tengelyekből álló koordináta-rendszer. A síkbeli derékszögű koordináta-rendszer felállításához rögzítjük a sík egy 0 pontját (latinul *origo*, „kiindulási pont"), és ezen át két egymásra merőleges egyenest (tengelyt) fektetünk. Valamely választott egység segítségével mindkét tengelyen beosztást készítünk az origóból kiindulva. (Az egyik irányban a pozitív, a másik irányban a negatív értékeket tüntetjük fel. Tehát a két tengely egy-egy olyan számegyenes, amelyek a 0 pontjukban metszik egymást és merőlegesek egymásra.) Az így elkészített derékszögű koordináta-rendszerben kölcsönösen egyértelmű hozzárendelést tudunk létrehozni a sík pontjai és a két tengelyen mért adatokból álló számpárok között, a következő módon. Ha a P pont merőleges vetülete az x-tengelyen P_x, az y-tengelyen pedig P_y, akkor a ponthoz rendelt számpár (P_x,P_y). A hozzárendelés kölcsönösen egyértelmű: minden ponthoz tartozik egy és csakis egy számpár, és minden számpárhoz tartozik egy és csakis egy pont.

A térbeli derékszögű koordináta-rendszer három egymásra merőleges tengelyből áll, és a segítségével (a síkbeli derékszögű koordináta-rendszer elve alapján) a tér pontjai és számhármasok között létesítünk kölcsönösen egyértelmű hozzárendelést.

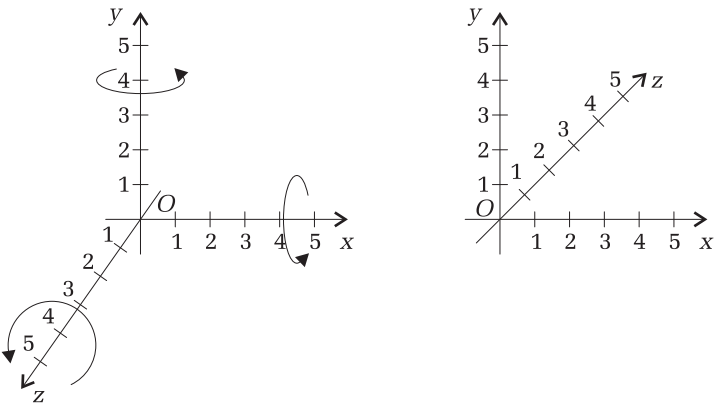


2.6. ábra. *Descartes-féle koordináta-rendszerek a síkban és a térben*

3D-s Descartes-féle koordináta-rendszer esetén beszélhetünk *jobbsodrású (jobbos)* vagy *balsodrású (balos)* koordináta-rendszerekről (2.7. ábra). A számítógépes grafikában a jobbsodrású koordináta-rendszereket használjuk.

2.2.2. Polárkoordináták

A *polárkoordináta-rendszer* egy olyan kétdimenziós koordináta-rendszer, mely a sík minden pontját egy szög és egy távolság függvényében határozza



2.7. ábra. *Jobbsodrású és balsodrású koordináta-rendszerek*

meg. Tulajdonképpen itt a sík egy paraméterezéséről beszélhetünk. Konkrétan a hozzárendelés, mely a sík derékszögű koordináta-rendszerben megadott (x, y) koordinátájú pontjait ellátja polárkoordinátákkal, a következő kapcsolatban van a derékszögű koordinátákkal:

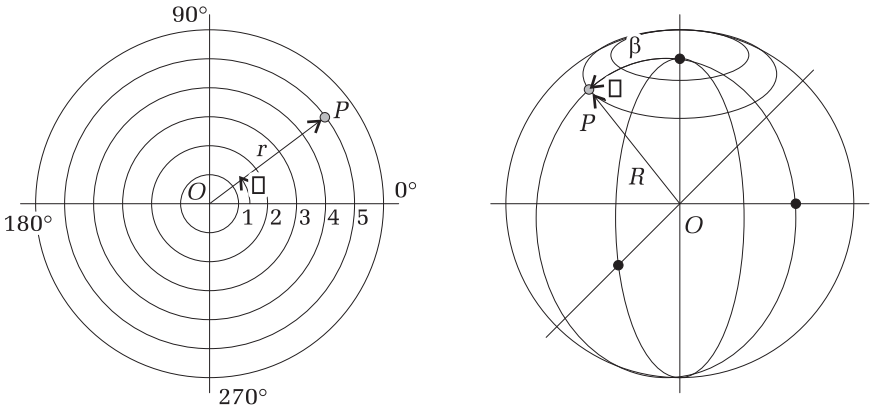
$$\begin{cases} x = r \cdot \cos\varphi \\ y = r \cdot \sin\varphi, \end{cases}$$

ahol r a sík $P(x, y)$ pontjának origótól mért távolsága (pozitív szám), ϕ pedig az x tengely és az *OP* szakasz irányított szögtávolsága (ez radiánban egy 0 és 2π közötti érték, fokban 0° és 360° közötti). A koordinátavonalakat ebben a rendszerben egyfelől azon pontok alkotják, melyek mentén a ϕ koordináta állandó, vagyis az origóból induló félegyenesek, másrészt azok, amelyek mentén r állandó, vagyis az origó középpontú körök.

A polárkoordináta-rendszert olyankor célszerű használni az elterjedtebb Descartes-féle derékszögű koordináta-rendszerrel szemben, ha a pontok helyének megadása egyszerűbb távolságokkal és szögekkel, mint két egymásra merőleges szakasz hosszával.

Visszaalakításkor vigyázni kell arra – mint minden esetben, amikor trigonometrikus értékekből következtetünk vissza szögértékre –, hogy helyes szöget adjon vissza a számítás. Ehhez a következőket kell szem előtt tartani.

- $r = 0$ esetén ϕ a polárkoordináta-rendszerben határozatlan, azaz bármely valós érték alkalmas lenne az origó szögének jellemzésére, hiszen ez az érték egyáltalán nem jellemzője az origónak.
- $r \neq 0$ esetén ahhoz, hogy a ϕ polárkoordinátára egyetlen értéket kapjunk, 2π hosszúságú intervallumra kell korlátozódnunk. A szokásos tartományok $[0, 2\pi)$ vagy $(-\pi, \pi]$.



2.8. ábra. *Polárkoordináták a síkban és a térben*

$$\begin{cases} r = \sqrt{x^2 + y^2} \\ \varphi = \tan^{-1}\left(\frac{y}{x}\right). \end{cases}$$

A háromdimenziós térben egy gömbfelület pontjait a felületi koordináta-rendszerekben adhatjuk meg. A koordináták mérőszáma a kör alakú koordinátavonalak ívének központi szöge. Az R sugarú gömb centrumát derékszögű koordinátarendszer origójába helyezve adjuk meg a felületi pontok térbeli helyzetét.

A felületi koordináta-rendszerek lehetnek *polárkoordináták*, vagy a földmérésben, földrajzban, csillagászattanban használatos *geodetikus* és *ekvatoriális koordináták*.

Polárkoordináták esetében:

$$\begin{cases} x = R \cdot \sin\beta \cdot \cos\varphi \\ y = R \cdot \sin\beta \cdot \sin\varphi \\ z = R \cdot \cos\beta. \end{cases}$$

2.2.3. Homogén koordináták

A homogén koordináták az n dimenziós tér egy pontjának helyzetét n+1 koordináta segítségével írják le, oly módon, hogy egy tetszőleges nullától eltérő értékkel az eredeti n dimenziós térben értelmezett koordinátákat megszorozzuk, és ezt a konstanst tekintjük az n+1-dik koordinátának.

Az n dimenziós tér egy pontja $(x_1, x_2, x_3, ..., x_n)$ homogén koordinátákkal kifejezve: $(xh_1, xh_2, xh_3, ..., xh_n, w)$. Az eredeti n dimenziós és a homogén koordináták közötti kapcsolatot az $xh_i = x_i \cdot w$ összefüggés fejezi ki, így egy

n dimenziós térben értelmezett pontnak végtelen számú homogén koordinátás megfelelője létezik.

Homogén koordinátákat használni célszerű, mert:

- Használatukkal az ideális térelemek (pont, egyenes, sík stb.) is könnyen megadhatók.
- A geometriai transzformációkat mátrix műveletek segítségével hajthatjuk végre.
- Több egymás után végrehajtandó transzformáció eredőjét egy transzformációs mátrixba foglalhatjuk össze.
- Használatuk és az alkalmazott módszerek könnyen általánosíthatók az n dimenziós térre.
- Végtelenben levő pontokat véges koordinátákkal fejezhetünk ki, pl. melyik 2D-s pont homogén koordinátás felírása a következő: (2, 7, 0)?
- Könnyebben meg tudjuk oldani segítségükkel a vágási feladatokat.

2.2.4. Objektumok viszonya egymáshoz

2.2.4.1. Pont és egyenes viszonya

A 2D-s térben egy egyenes általános alakja:

$$ax + by + c = 0.$$

(Az $y = mx + b$ nem az egyenes általános alakja, mert ezzel a kifejezéssel nem tudjuk leírni az y tengellyel párhuzamos egyeneseket!)

Egy 2D-s pont koordinátái akkor elégíti ki az egyenes egyenletét, ha pont az egyenesre esik.

A pont homogén koordinátáit használva az egyenes egyenletének bal oldala az egyenes együtthatóiból alkotott vektor (e) és a pont homogén koordinátáiból alkotott vektor (p) skaláris szorzata.

Az $\langle e\, p \rangle$ skaláris szorzat eredményének előjele megadja, hogy a pont az egyenes melyik oldalára esik, az értéke pedig arányos a pont és az egyenes távolságával.

2.2.4.2. Két ponton átmenő egyenes

Legyen p_1 és p_2 két nem egybeeső 2D-s pont homogén koordinátás vektora. A két ponton áthaladó egyenes egyenletének együtthatói megegyeznek a p_1 és p_2 vektorok vektoriális szorzatának koordinátáival ($e = p_1 \times p_2$).

2.2.4.3. Két egyenes metszéspontja

Vegyük észre, hogy egy egyenes és egy pont homogén koordinátás alakja egy háromelemű vektorral írható le. Ezek alapján a két egyenes metszéspontjának kiszámítására is alkalmazható a két ponton átmenő egyenes együtthatóinak kiszámításához felírt összefüggés.

Pont és egyenes távolsága

Egy pont homogén koordinátás vektora és egy e egyenes együttható vektorának a skaláris szorzata a pont és az egyenes távolságával arányos.

A skaláris szorzat normálásával a távolságot is megkaphatjuk.

A $p(xh, yh, w)$ pont és az $e(a, b, c)$ egyenes távolsága (t):

$$t = \langle p\ e \rangle \frac{1}{w\sqrt{a^2 + b^2}}.$$

2.2.5. 3D transzformációk

3D-ben az (x, y, z, w) homogén koordináták az $(x/w, y/w, z/w)$ háromdimenziós koordináták megfelelői. Homogén koordináták segítségével a lineáris és a perspektív transzformációk leírhatók egy 4×4-es mátrix segítségével. A homogén koordinátás megadással az összes transzformáció összevonható egy transzformációba – összeszorozva a mátrixokat.

Az (x,y,z) koordinátákkal rendelkező térbeli pontot homogén koordináták segítségével a következő oszlopvektorral ábrázoljuk:

$$\begin{bmatrix} xh \\ yh \\ zh \\ w \end{bmatrix},$$

ahol w egy tetszőleges valós konstans, $xh = x/w$, $yh = y/w$, $zh = z/w$. Azok a pontok, amelyeknél $w = 0$, a végtelenben vannak.

Az általánosított transzformációs mátrix 3D homogén koordinátákra a következőképpen néz ki:

$$T = \begin{bmatrix} a & b & c & l \\ d & e & f & m \\ g & h & i & n \\ p & q & r & s \end{bmatrix}.$$

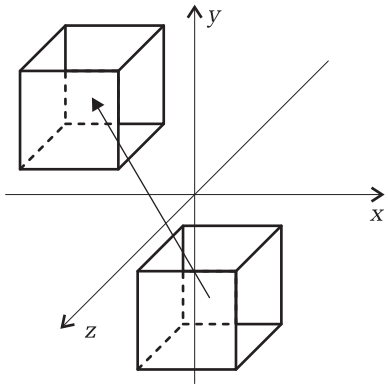
A mátrixot négy részre lehet felosztani, a következőképpen:

$$\begin{bmatrix} & & 3 \\ 3 \times 3 & & \times \\ & & 1 \\ 1 \times 3 & & 1 \times 1 \end{bmatrix},$$

– a 3×3-as mátrix magába foglalja a lokális átméretezést, torzítást, tükrözést és forgatást,
– az 1×3-as mátrix a perspektivikus vetítést jelképezi,
– a 3×1-es mátrix az eltolást jelképezi,
– az 1×1-es mátrix pedig a globális átméretezést.

2.2.5.1. Az eltolás

A mértanban az *eltolás (translation)* az egybevágósági transzformációk közé tartozik. Ha a sík vagy a tér minden pontjának képe ugyanabban az irányban, ugyanakkora távolságban fekszik, akkor a transzformáció eltolás. Ha adva van egy v vektor, akkor a vele való eltolásban minden P pont P' képére teljesül, hogy a PP' vektor egyenlő v-vel. Az identitás is felfogható eltolásnak; ekkor az eltolásvektor a nullvektor.



2.9. ábra. *Eltolás*

Az eltolás transzformációs mátrixa:

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Legyen P egy pont a térből az (x, y, z) koordinátákkal. Ha P-t eltoljuk az Ox tengelyen t_x-el, Oy-on t_y-al, illetve Oz-n t_z-vel, akkor P a P' pontba kerül, az (x', y', z') koordinátákkal, ahol:

$$\begin{cases} x' = x + t_x \\ y' = y + t_y \\ z' = z + t_z \end{cases}$$

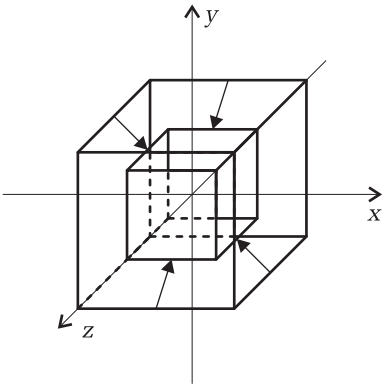
vagy mátrixos alakban:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = T \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

2.2.5.2. Átméretezés, skálázás

Az átméretezés (scaling) egy objektum nagyítását vagy kicsinyítését, torzítását jelenti. A skálázás két típusú lehet:

- lokális és
- globális.



2.10. ábra. *Átméretezés*

A lokális méretezés mátrixa a következő:

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Vegyünk egy P pontot a térből, az (x, y, z) kordinátákkal, ez a lokális méretezés következtében a P' pontba kerül, az (x', y', z') koordinátákkal, ahol:

$$\begin{cases} x' = x \cdot s_x \\ y' = y \cdot s_y \\ z' = z \cdot s_z \end{cases}$$

vagy mátrixos alakban:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = S \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Az átméretezési, skálázási tényezők mind pozitív számok. Ha a tényező 0 és 1 között van, akkor az átméretezett pont helyzetvektora kisebb lesz (közelebb kerül az origóhoz) – ekkor kicsinyítésről beszélünk –, ha a méretezési tényező nagyobb mint 1, akkor a helyzetvektor növelve lesz – ekkor nagyításról beszélünk.

A globális méretezés mátrixa a következő:

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & s \end{bmatrix}.$$

Vegyünk egy P pontot a térből, az (x, y, z) kordinátákkal, ez a globális méretezés következtében a P' pontba kerül, az (x', y', z') koordinátákkal, ahol:

$$\begin{cases} x' = \frac{x}{s} \\ y' = \frac{y}{s} \\ z' = \frac{z}{s} \end{cases}$$

vagy mátrixos alakban:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = S \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Ha $s < 1$, akkor a helyzetvektor nő, ha $s > 1$, a helyzetvektor csökken.

A globális átméretezést lokális átméretezéssel is meg lehet oldani, ha a következő mátrixot használjuk:

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

2.2.5.3. Forgatás egy koordinátatengely körül

A mértanban a *forgatás (rotation)* az egybevágósági transzformációk közé tartozik. A síkban pont körüli, a térben tengelyes forgatások léteznek. A síkban a forgatás az a transzformáció, amelyre teljesül, hogy az O középpont körüli

forgatás során bármely P pont esetére a *POP'* szög a sík minden pontjára ugyanakkora. A térben forgatás az a transzformáció, ami egy adott egyenesen kívüli P pontot egy olyan P' pontba viszi, amely a P-n átmenő, az egyenesre merőleges síkban ugyanakkora távolságra fekszik, mint a P pont, és a *POP'* irányított szög ugyanakkora minden ilyen P pontra.

A síkban kitüntetett szerepet játszik a 180 fokos forgatás, amelyet *középpontos tükrözés*nek nevezünk.

Az identitás is felfogható forgatásnak.

A síkbeli tengelyes tükrözések a térben kiterjeszthetők forgatássá, amelyet szintén tengelyes tükrözésnek nevezünk, és részben hasonló szerepet tölt be, mint a pontra tükrözés a síkban.

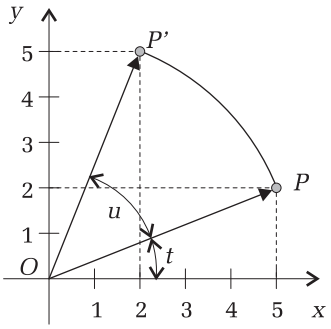
Az egyszerűbb tárgyalás kedvéért először bemutatjuk az origó körüli kétdimenziós forgatást.

Ezt a transzformációt egy szög határozza meg; ha ez a szög pozitív, akkor a forgatás trigonometrikus irányban lesz, ha negatív, akkor az óramutató mozgási irányába történik.

Legyen $P(x, y)$ egy pont a síkból és u egy szög. A $P'(x', y')$ pont koordinátáinak a meghatározása egyszerűbb, ha P és P' koordinátáit parametrikusan adjuk meg (2.11. ábra):

$$\begin{cases} x = r \cdot \cos t \\ y = r \cdot \sin t \end{cases} \qquad \begin{cases} x' = r \cdot \cos(t+u) \\ y' = r \cdot \sin(t+u) \end{cases},$$

ahol r az OP helyzetvektor hossza és t az általa a vízszintessel bezárt szög.



2.11. ábra. *Forgatás*

Ha kifejezzük a $\cos(t+u)$ és $\sin(t+u)$ képleteket:

$$\begin{cases} x' = r \cdot (\cos t \cdot \cos u - \sin t \cdot \sin u) \\ y' = r \cdot (\cos t \cdot \sin u + \sin t \cdot \cos u) \end{cases},$$

de $r \cdot \cos t = x$, $r \cdot \sin t = y$, vagyis:

$$\begin{cases} x' = x \cdot \cos u - y \cdot \sin u \\ y' = x \cdot \sin u + y \cdot \cos u \end{cases}$$

vagy mátrixos alakban:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos u & -\sin u \\ \sin u & \cos u \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}.$$

3D-ben az *Ox* tengely körüli forgatáskor a helyzetvektor x koordinátái nem változnak. A forgatások az *Ox* tengelyre merőleges síkokra történnek. Hasonlóképpen az *Oy* vagy *Oz* tengelyek körüli forgatáskor a helyzetvektor y, illetve z koordinátái nem változnak, a forgatás az *Oy*, illetve *Oz* tengelyekre merőleges síkokban történik.

A helyzetvektor transzformációja mindegyik ilyen síkban egy kétdimenziós síkban levő forgatás.

Kiindulva az origó körüli síkbeli forgatási mátrixból és figyelembe véve, hogy az *Ox* tengely körüli forgatáskor az x koordináta nem változik, az α szöggel történő forgatási mátrix:

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

Hasonlóképpen az *Oy* tengely körüli forgatási mátrix egy β szöggel a következő lesz:

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

A forgatási mátrix egy γ szöggel az *Oz* tengely körül:

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

A forgatási mátrix bal felső sarkában levő 3×3-as mátrix oszlopai és sorai merőleges vektorok (minden két oszlop vagy minden két sor szorzata a 0 vektort eredményezi).

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = R \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Az *Ox* tengely körüli forgatást *billentő (pitch)*, az *Oy* körülit *forduló (jaw)*, az Oz körülit pedig *csavaró (roll) forgatás*nak nevezzük.

2.2.5.4. Tükrözés a koordináta-rendszer egyik síkjához viszonyítva

Egy 3D objektum áthelyezése egy másik pontba nem csak forgatásokkal történhet. Szükségesek a *tükrözés*i *(reflection, mirror)* transzformációk is. A 3D tükrözés egy síkhoz viszonyítva történik.

Az *xy* síkkal szembeni tükrözés esetében csak a z koordinátának változtatjuk meg az előjelét, az x, illetve y koordináták nem változnak, így a tükrözési mátrix az *xy* sík esetében:

$$M_{xy} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Az *yz* sík esetében:

$$M_{yz} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Az *xz* sík esetében:

$$M_{xz} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

2.2.5.5. Torzítás, ferdítés

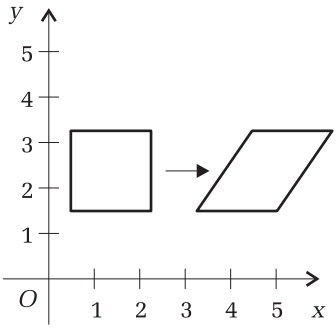
A *torzítás (skew, shear, transvection)* lineáris leképzés, lerögzíti a pontokat az egyik tengely szerint, a másik tengely szerint viszont eltolja őket a tengelyhez mért távolságukkal arányosan.

A torzítás mátrixai:

$$H_{xy} = \begin{bmatrix} 1 & 0 & h_x & 0 \\ 0 & 1 & h_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$H_{xz} = \begin{bmatrix} 1 & h_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & h_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$H_{yz} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ h_y & 1 & 0 & 0 \\ h_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



2.12. ábra. *Torzítás*

A fent bemutatott transzformációk térbeli pontokra érvényesek. Ha például egy szakaszt akarunk transzformálni, akkor a két végpontján hajtjuk végre a transzformációkat, és így megkapjuk az új szakaszt. Hasonlóképpen, egy három pont által meghatározott sík esetében a három ponton hajtjuk végre a transzformációkat.

2.2.5.6. Transzformációk konkatenálása

Egy pontra több elemi transzformációt (pl. hármat) alkalmazva a következő összefüggés adódik:

$$p'^T = \left(\left(\left(p^T \cdot M_1\right) \cdot M_2\right) \cdot M_3\right).$$

Mivel a mátrix szorzás asszociatív (csoportosítható), az összefüggést felírhatjuk a következő alakban:

$$p'^T = p^T \cdot \left(\left(M_1 \cdot M_2\right) \cdot M_3\right).$$

A transzformációs mátrixok szorzatát előre kiszámíthatjuk, és egy eredő M transzformációs mátrixot írhatunk fel.

Egy 2D vagy 3D objektumon végrehajtott transzformációk sorozatát egy transzformációba össze tudjuk foglalni. Az összetett 3×3-as vagy 4×4-es mátrixot úgy kapjuk, hogy összeszorozzuk az elemi transzformációknak megfelelő mátrixokat.

Figyelembe kell viszont venni, hogy a mátrixszorzás nem kommutatív művelet, így a transzformációs mátrixok sorrendjét nem szabad felcserélni:

$$M_1 \cdot M_2 \neq M_2 \cdot M_1.$$

2.2.5.7. Transzformációk ellentettje

Egy M transzformáció ellentettjét a transzformációs mátrix inverzével fejezhetjük ki. Az eddig felírt elemi transzformációk mindegyike invertálható.

Az egyes elemi transzformációk ellentett transzformációja és az eredeti transzformációs mátrix inverze azonos. Például a 15 fokos forgatás transzformációs mátrixának az inverze a mínusz 15 fokos forgatás transzformációs mátrixával azonos.

2.2.6. 2D transzformációk

Az elemi 2D-s geometriai transzformációk 3×3-as mátrixok segítségével írhatók fel. Egy pont transzformáció utáni koordinátáit a pont homogén koordinátás vektora és a transzformációs mátrix szorzata adja.

A transzformációs mátrix 4 részre bomlik:

$$\begin{bmatrix} & 2 \\ 2\times 2 & \times \\ & 1 \\ 1\times 2 & 1\times 1 \end{bmatrix},$$

- a 2×2-es mátrix magába foglalja a lokális átméretezést, torzítást, tükrözést és forgatást,
- az 1×2-es mátrix a perspektivikus vetítést jelképezi,
- a 2×1-es mátrix az eltolást jelképezi,
- az 1×1-es mátrix pedig a globális átméretezést.

2.2.6.1. A transzformációs mátrixok

Eltolás:

$$T = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Lokális átméretezés:

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Globális átméretezés:

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & s \end{bmatrix}$$

Tükrözés: a skálázás segítségével az x, illetve az y tengely szerinti tükrözést is végrehajthatjuk. Például ha $s_x = -1$, és $s_y = 1$ az x tengely menti tükrözés.

Origó körüli forgatás:

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

A pozitív forgási irány az óramutató irányával ellentétes.

Torzítás, ferdítés:

$$H = \begin{bmatrix} 1 & h_x & 0 \\ h_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

2.3. Vetítés

A vetítés a számítógépes grafika egyik legfontosabb transzformációja, hisz segítségével tudjuk megvalósítani a 3D valós világban lévő tárgyak szemléletes ábrázolását a 2D képernyőn vagy papíron.

Ha a valós tárgyakat úgy ábrázolnánk a képernyőn, hogy egyszerűen elhagynánk a mélységet jelző z koordinátát, egy nagyon szegényes, nem szemléletes képet kapnánk, hisz a gömb nem kör, a kocka nem négyzet stb. Valamilyen úton-módon szemléltetni kell a z koordinátát is, ennek függvényében kell kiszámítani a másik kettőt.

Vetítésre két módszer terjedt el. A *képies* ábrázolásmód az emberi látáshoz (és fényképezéshez) nagymértékben illeszkedik. Az ilyen ábra igen szemléletes, de torzításai jelentősek. A *mérnöki* ábrázolásmód a tárgy tényleges méreteiből, arányaiból lehetőleg sokat megtartó módszer. Az ilyen módon készült ábrák szükségképpen kevésbé szemléletesek. A perspektivikus torzulásokhoz szokott emberi szem sokszor éppen ezeket az ábrákat látja „torzaknak”.

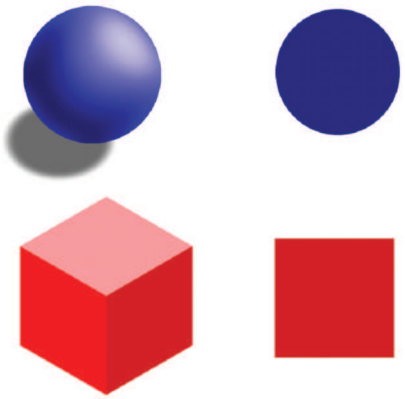
De mit is nevezünk vetítésnek? *Vetítés*ek *(projection)* azok a dimenzióveszteséggel járó ponttranszformációk, melyeknél bármelyik képpont és a neki megfelelő összes tárgypont egy egyenesen helyezkedik el.

A fény egyenes vonalú terjedése folytán az optikában létrejövő leképezési folyamatok nagy része ilyen transzformációval egyenértékű – ez teszi magától értetődővé a vetítés szó használatát.

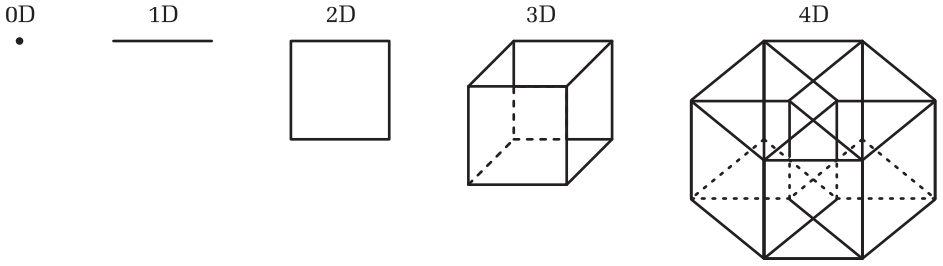
Az összetartozó tárgy- és képpontokon áthaladó egyenest *vetítősugár*nak nevezzük.

- *Képfelület*: az a felület, amire vetítünk.
- *Tárgypont*: pont a valódi tárgyon.
- *Képpont*: pont a képen (vetületen).
- *Vetület*: a tárgynak a képfelületen létrejött képe.

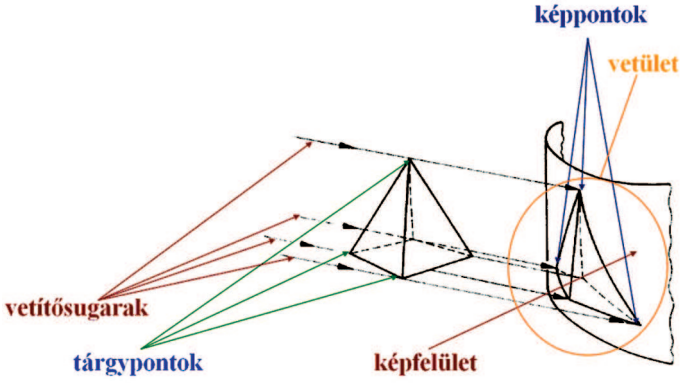
Mivel a képernyő, a fénykép, a nyomtatópapír sima, sík felületek, a számítógépes grafikában elsődlegesen a sík képfelületek érdekelnek – ezek a *képsík*ok.



2.13. ábra. *Ha csak egyszerűen elhagyjuk a z koordinátát*



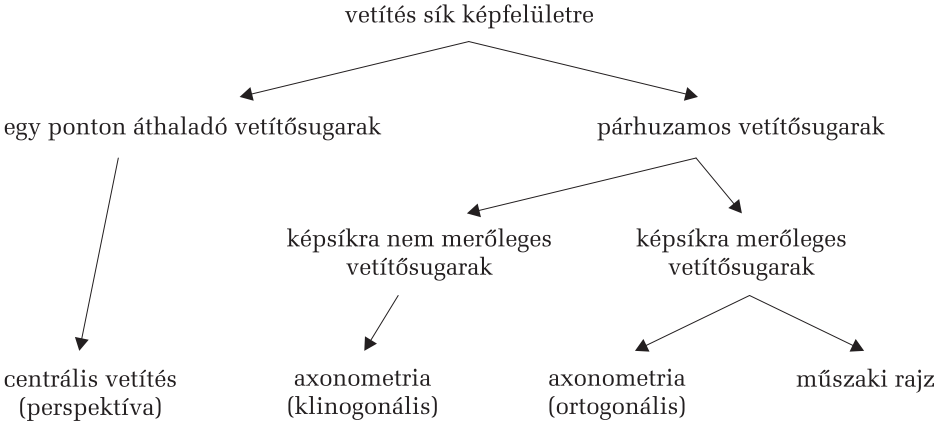
2.14. ábra. *Dimenzióveszteség*



2.15. ábra. *A vetítés fogalmai*

Az a szabály, amely szerint vetítősugarainkat kiválasztjuk, alapvetően befolyásolja a kialakuló kép jellegét [93].

Ennek megfelelően sorolhatjuk csoportokba a vetítés fajtáit.



2.16. ábra. *A vetítés fajtái*

2.3.1. A centrális vetítés (perspektíva)

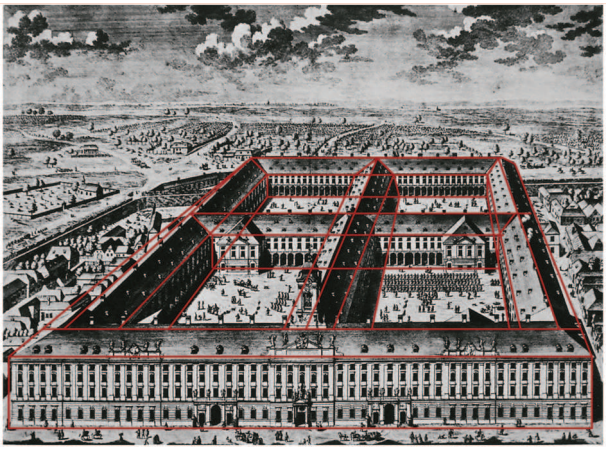
- A vetítősugarak mindegyike áthalad egy vetítési középponton, a *centrumpont*on.
- A létrejövő kép igen közel áll az emberi szem, a fényképezőgép által alkototthoz.
- A perspektivikus hatás elsősorban a tárgy és a centrumpont távolságától függ.
- Ha ez a távolság minden határon túlnő, a középpontos vetítés párhuzamos vetítésbe megy át (a centrumpont a végtelenben lesz).

A *perspektíva* szó a latin *per* (teljesen, alaposan) és *specto* (lát, megnéz) szavak összetételeként született. A geometriához tartozó *vonalperspektíván* túl meg kell említeni a művészethez, esztétikához tartozó *színperspektívát* és *légi perspektívát.*

A vonalperspektívát feltehetőleg már az ókori görögök is ismerték (azonban festmények nem, csak leírások maradtak fenn). A kérdést tudományos alapossággal először a reneszánszban kezdték vizsgálni. A XIV. században Giotto di Bondone (1267–1337) egy életen át kísérletezett a perspektívával. Ajtósi Dürer (1471–1528) fizikai eszközt szerkesztett a centrális projekció tanulmányozására.

A vonalperspektíva szabályai:

- Két hasonló tárgy közül a közelebbit nagyobbnak látjuk.



2.17. ábra. *Vonalperspektíva. A pesti Invalidus-palota, Salomon Kleiner (1700–1761) rézmetszete*

- Két egyforma tárgy közül távolabbinak látjuk azt, amelyik a képen magasabban áll.
- Az összetartó vonalak távolodó párhuzamosaknak látszanak.
- Ha két azonos tárgy egyike részben takarja a másikat, akkor a takaró tárgy közelebbinek látszik.
- Ha apróbb, egyforma tárgyak tömeget alkotnak, akkor a távolabbiak kisebbnek és egymáshoz közelebb állónak látszanak.
- Egymás mögötti tárgyak méretcsökkenése távlati hatást vált ki.

A légi perspektívát Leonardo da Vinci alkotta meg. Szabálya, hogy a légkörnek köszönhetően a távoli tárgyak kéknek látszanak, mégpedig minél közelebb esnek a horizonthoz, annál kékebbek. Festői „szabály", hogy ami ötször távolabb van a valóságban, a vásznon ötszörte kékebb kell hogy legyen.

A színperspektíva is Leonardo da Vinci alkotása. Szabálya, hogy az előtérben levő tárgyak világosabbak és melegebb színűek, a háttérben lévő tárgyak sötétebbek és hidegebb színűek, valamint az, hogy a kiegészítő színek használata perspektivikus hatást kelt.

A perspektivikus ábrázolásmód nem mentes a hibáktól, ezek pedig optikai csalódásokat okozhatnak. William Hogarth (1697–1764), valamint Maurits Cornelis Escher (1898–1972) nagy előszeretettel alkalmazták a hamis perspektívát, s így a perspektivikus ábrázolás veszélyeire hívták fel a figyelmet.

A hamis perspektíva szabályai:

- Egy szürke tárgy fekete környezetben világosabbnak tűnik, mint fehérben.
- Egyforma hosszúságú, egymásra merőleges vonalak közül a függőlegesek hosszabbnak tűnnek, mint a vízszintes.

- A fehéren izzó objektum nagyobbnak látszik, mint a valóságban.
- Tiszta időben távolabbi tárgyak közelebbieknek látszanak, párás levegőben ez fordítva történik.
- Egy objektum piros fénnyel fehér lapra vetett árnyéka zöldes színű (kiegészítő színek).
- Felületesen nézve bizonyos ábrákat térbelinek látunk annak ellenére, hogy ilyen térbeli ábrák nem is léteznek.
- Ha egy ábra sok olyan elemet tartalmaz, amelyek a perspektíva érzékeltetésére szolgálnak, akkor azt akkor is perspektivikusnak látjuk, ha nem az.

2.18. ábra. *William Hogarth (1697–1764) műve, amely a perspektivikus ábrázolás veszélyeire hívja fel a figyelmet*

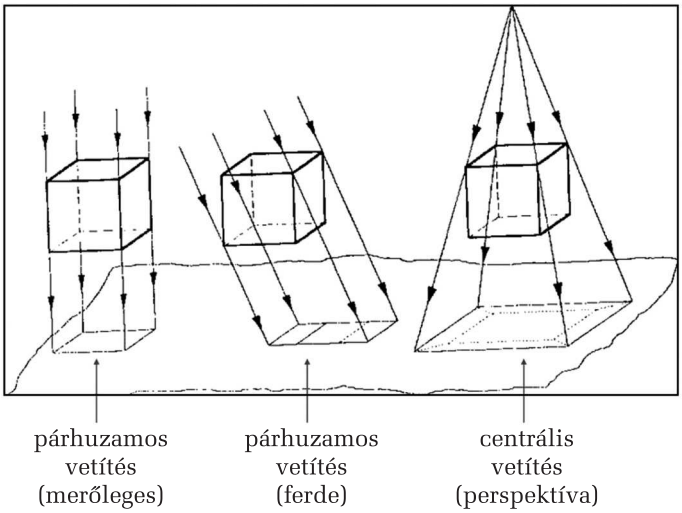
2.3.2. Párhuzamos vetítések

*Párhuzamos vetítés*ről beszélünk, ha a vetítősugarak egymással párhuzamosak. Ha ezen kívül a vetítősugarak még merőlegesek is a képsíkra, a *merőleges (ortogonális) vetítés*, egyébként a *ferde (klinogonális) vetítés* elnevezést használjuk.

Habár a vetítések pontosan leírhatók geometriai transzformációkkal, erre csak a XIX. században jöttek rá (Pohlke tétele, 1860), addig a mérnökök az ún. *axonometriákat*, vagyis a tengelyekre való felméréseket használták.

Az *axonometria* térbeli tárgyak szemléletes síkbeli ábrázolására szolgáló módszer. Az axonometria a latin *axis* (tengely) és *metrum* (mérték) szavakból ered. Az axonometriákat úgy lehet elérni, hogy a valós tárgyat megmérjük, a

méreteket pedig felmérjük egy 2D képzeletbeli koordináta-rendszer tengelyeire, vagy felhasználjuk a Monge-féle ábrázoló geometriát, amely a műszaki rajz alapja, tulajdonképpen egy kétképsíkos projekció (a tárgyat különböző szemszögekből több képsíkra vetítjük le, így teljes képet kapunk róla – a segítségével teljes mértékben a 2D rajz alapján rekonstruálni tudjuk a 3D tárgyat).



2.19. ábra. *Vetítések*

Minden axonometrikus ábrázoláshoz meg kell adni egy térbeli derékszögű koordináta-rendszer x, y és z tengelye képének irányát és az egyes irányokhoz tartozó q_x, q_y és q_z rövidüléseket. *Rövidülés* az a szorzószám, amellyel az eredeti térbeli koordinátát megszorozva az axonometrikus vetület megfelelő távolsága lesz. A gyakorlati axonometriák mind nem elfajulók, azaz egyik rövidülésük sem egyenlő nullával, és a koordinátatengelyek különböző irányúak.

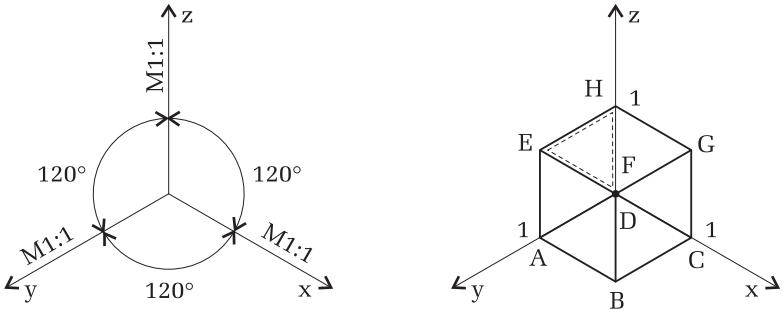
Párhuzamos egyenesek az axonometriában is párhuzamosak maradnak. A kör képe ellipszis, a kör köré rajzolt négyzet az axonometriában parallelogramma lesz, oldalai az ellipszis érintői.

Gyakorlati axonometriák:

- Izometrikus (egyméretű) axonometria
- Dimetrikus (kétméretű) axonometria
- Kavalier (frontális) axonometria
  - Madártávlat (katonai axonometria)
  - Békatávlat

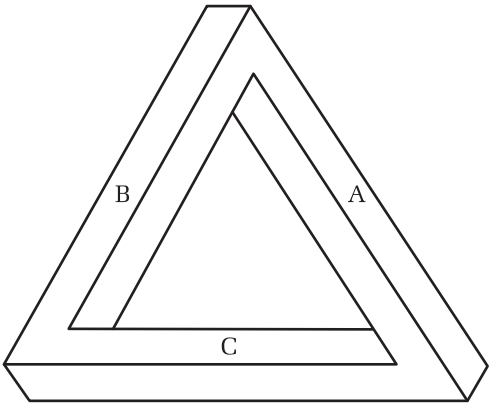
2.3.2.1. Izometrikus axonometria

A koordinátatengelyek egymással 120–120°-os szöget zárnak be. A rövidülések egyenlők: $q_x = q_y = q_z = 1$. Ezt az axonometriát igen egyszerű szerkeszteni, de nem nagyon képies. Kiterjedten használják térbeli csővezetékek rajzainak készítésénél. Ezekhez néha előnyomott 120°-os hálót tartalmazó rajzlapot használnak, melyen szabadkézi vázlatok is könnyen készíthetők.



2.20. ábra. *Izometrikus axonometria*

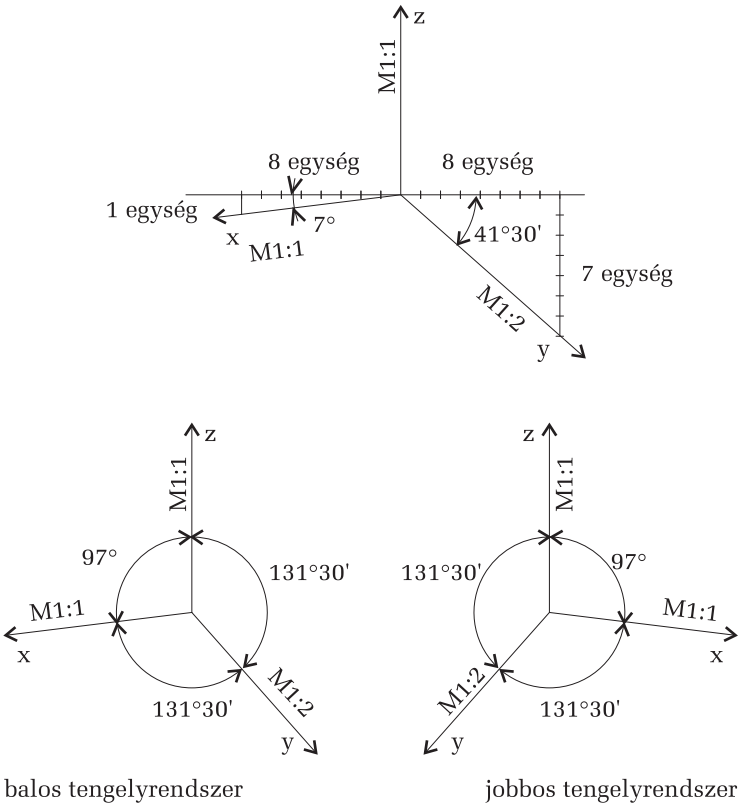
A 2.20. ábrán megfigyelhetjük, hogy kocka esetében a D pont egybeesik az F ponttal, emiatt a DEH alakzat a *lehetetlen háromszög* (2.21. ábra) tulajdonságaival rendelkezik, és ez képezi az alapját a hasonló jellegű optikai csalódásoknak.



2.21. ábra. *A lehetetlen háromszög*

2.3.2.2. *Dimetrikus axonometria*

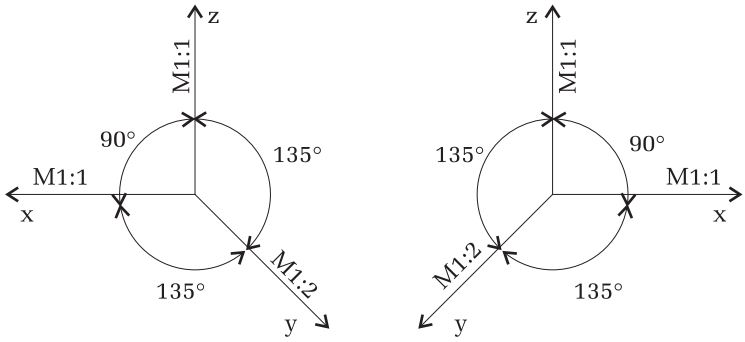
A z tengelyt megtartjuk függőlegesnek, a vízszintes tengelyirányokat pedig 1:8 és 7:8 arányú lejtéssel rajzoljuk meg. Így az x tengelyt 97°-ra, az y tengelyt pedig 131°30'-re rajzoljuk a függőleges z tengelytől. A rövidülések: $q_x = q_z = 1$, $q_y = 0{,}5$. Ez az ábrázolás elégíti ki leginkább a képiesség követelményét.



2.22. ábra. *Dimetrikus axonometria*

2.3.2.3. *Kavalier-axonometria*

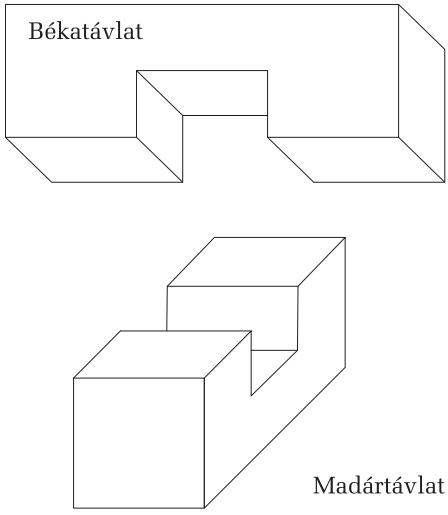
A z tengely függőleges helyzetű. Az x tengely a z tengelyre merőleges, és mindkét tengelyre a méreteket valódi nagyságban rajzoljuk. Az y tengelyt a vízszinteshez képest 135°-os lejtéssel rajzoljuk, és a méreteket 1:2 arányú rövidüléssel mérjük fel.



2.23. ábra. *Kavalier-axonometria*

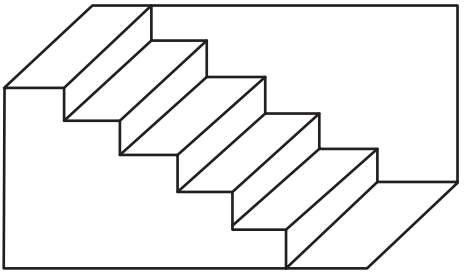
Neve onnan ered, hogy ezt a módszert használták a katonai objektumok lerajzolásánál. Kavalier-axonometria megvalósítására példát láthatunk a 6.1. (*Vetítés és forgatás*) fejezetben.

Kétféleképpen valósulhat meg: ha felülnézetből rajzoljuk le a tárgyat, akkor *madártávlat*ról, ha alulnézetből, akkor *békatávlat*ról beszélhetünk. A kétféle ábrázolásmódot keverni is lehet, sőt igen hasznos például gerendakötések ábrázolásakor.



2.24. ábra. *Békatávlat és madártávlat*

A Schröder-lépcső (1858) esetében nem lehet eldönteni, hogy a Kavalier-axonometriát béka- vagy madártávlat szerint alkalmaztuk-e, ez az alapja a hasonló jellegű optikai csalódásokat tartalmazó képeknek.



2.25. ábra. *A Schröder-lépcső: Békatávlat és madártávlat?*

2.3.3. A vetítések matematikai leírása

2.3.3.1. Az ortografikus vetítés

Az ortografikus vetítés a legegyszerűbb párhuzamos vetítés, amelyet leginkább a műszaki rajzban alkalmaznak. Ez a vetítés pontosan adja vissza az objektum a koordináta-rendszer egyik síkjával párhuzamos oldalainak a méreteit és a formáját. Az ortografikus vetítések a koordináta-rendszernek az $x = 0$, $y = 0$, illetve $z = 0$ síkokra történő vetítések. A vetítési mátrix a $z = 0$ síkra:

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

Ha egy helyzetvektorra alkalmazzuk ezt a vetítést, a vektor z koordinátája 0 lesz. A vetítési mátrix az $x = 0$ síkra:

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

Ha egy helyzetvektorra alkalmazzuk ezt a vetítést, a vektor x koordinátája 0 lesz. A vetítési mátrix az $y = 0$ síkra:

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

Ha egy helyzetvektorra alkalmazzuk ezt a vetítést, a vektor y koordinátája 0 lesz.

Amint a műszaki rajzból ismeretes, egy ortografikus vetítés nem elegendő az objektum rekonstruálásához, ezért több ortografikus vetítésre van szükség. Hat ortografikus vetítés létezik, éspedig:

- *elölnézet* – a $z = 0$ síkban, a centrumpont a végtelenben, a $+z$ tengelyen;
- *hátulnézet* – a $z = 0$ síkban, a centrumpont a végtelenben, a $-z$ tengelyen;
- *jobb oldali nézet*: – az $x = 0$ síkban, a centrumpont a végtelenben, a $+x$ tengelyen;
- *bal oldali nézet* – az $x = 0$ síkban, a centrumpont a végtelenben, a $-x$ tengelyen;
- *felülnézet* – az $y = 0$ síkban, a centrumpont a végtelenben, a $+y$ tengelyen;
- *alulnézet* – az $y = 0$ síkban, a centrumpont a végtelenben, a $-x$ tengelyen.

Egy objektum rekonstruálásához általában három nézetet használunk: elölnézetet, felülnézetet és jobb oldali nézetet.

A hat vetületet egyetlen vetületre lehet lecsökkenteni az $x = 0$ síkban, a centrumponttal a $+x$ tengelyen a végtelenben, ha előzőleg az objektumon néhány transzformációt hajtunk végre. Pl. a bal oldali nézetet úgy kaphatjuk meg, ha az objektumot elforgatjuk +90°-kal a z tengely körül, majd levetítjük az $x = 0$ síkra.

2.3.3.2. A perspektivikus vetítés

Perspektivikus vetítés esetén megfigyelhetünk egy *kicsinyítési effektus*t. Egy objektum vetületének a nagysága fordítottan arányos az objektum és a centrumpont közötti távolsággal. Minél távolabb van egy objektum a centrumponttól, annál kisebb lesz a síkban a vetülete.

A számítógépes grafikában gyakran használjuk a perspektivikus vetítést a kicsinyítési effektus miatt, amely egy mélységi érzetet kelt a síkbeli képeken. Ez a vetítés nem őrzi meg az objektum méreteit.

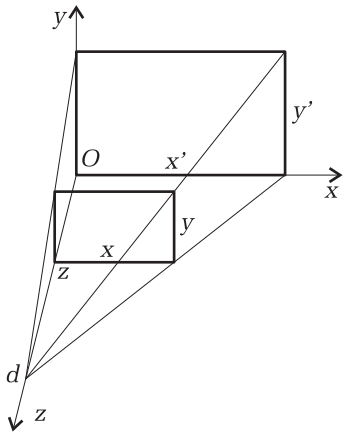
Egy *perspektivikus transzformáció*t egy vagy több centrumpont és egy vetítési sík határoz meg. A transzformáció egy 3D teret egy 3D térre képez le. Ha egy síkra akarunk levetíteni, akkor a perspektivikus vetítés egy perspektivikus transzformáció és egy ortografikus vetítésből fog összetevődni.

A centrumpontok és a vetítési sík helyzetétől függően a perspektivikus vetítések háromfélék lehetnek:

- 1 centrumponttal,
- 2 centrumponttal,
- 3 centrumponttal.

Az egy centrumponttal rendelkező perspektivikus vetítés a standard perspektíva.

A vetítési sík a $z = 0$ és a centrumpont a z tengelyen van, a $(0, 0, d)$ koordinátákban. Legyen $P(x, y, z)$ egy pont, amely a $P'(x', y', z')$ pontba vetítődik (2.26. ábra).



2.26. ábra. *Perspektivikus vetítés 1 centrumponttal*

A háromszögek hasonlóságából következik:

$$\frac{x'}{d} = \frac{x}{d-z}, \text{ és } \frac{y'}{d} = \frac{y}{d-z},$$

tehát:

$$x' = \frac{d \cdot x}{d-z} = \frac{x}{1-z/d}, \text{ és } y' = \frac{d \cdot y}{d-z} = \frac{y}{1-z/zdd}.$$

A z' pedig 0.

Jelöljük $-1/d$-t r-rel, ekkor a transzformációs mátrix így fog kinézni:

$$P_{pers_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & r & 1 \end{bmatrix}.$$

Ha kihangsúlyozzuk a perspektivikus transzformációt és az ortografikus vetítést, akkor:

$$P_{pers_1} = P_{ort} \cdot T_{pers_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & r & 1 \end{bmatrix}.$$

A perspektivikus transzformáció nincs hatással azokra a pontokra, amelyek a vetítési síkban vannak, tehát ha a vetítési sík metsz egy objektumot, akkor azt a metszetet a valós formában és méretben kell ábrázolni, az objektumnak minden más része eltorzul.

Ha a centrumpont az x tengelyen van, $(d, 0, 0)$ koordinátákkal, és a vetítési sík az $x = 0$, a perspektivikus transzformáció mátrixa így alakul:

$$T_{pers_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p & 0 & 0 & 1 \end{bmatrix},$$

ahol $p = -1/d$.

Ha a centrumpont az y tengelyen van, $(0, d, 0)$ koordinátákkal, és a vetítési sík az $y = 0$, a perspektivikus transzformáció mátrixa így alakul:

$$T_{pers_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & q & 0 & 1 \end{bmatrix},$$

ahol $q = -1/d$.

Ha egy tetszőleges $C\,(x_C,y_C,z_C)$ centrumpontból vetítünk a z = 0 síkra, a képletek a következőképpen alakulnak:

Legyen $P\,(x,y,z)$ egy tetszőleges pont a térből, $P'\,(x',y',z')$ pedig a C centrumpont szerinti vetülete a z = 0 síkra.

A perspektíva szabályainak megfelelően *CP* és *CP*' párhuzamos, z' = 0. Ekkor:

$$CP||CP' \quad \Rightarrow \quad \frac{x'-x_C}{x-x_C} = \frac{y'-y_C}{y-y_C} = \frac{z'-z_C}{z-z_C} \quad \Rightarrow \quad \begin{cases} x' = \frac{x\cdot z_C - z\cdot x_C}{z_C - z} \\ y' = \frac{y\cdot z_C - z\cdot y_C}{z_C - z} \\ z' = 0 \end{cases}$$

Két centrumpontos perspektíva esetében, ha a vetítési sík z x tengellyel párhuzamos, akkor a transzformációs mátrix a következőképpen néz ki:

$$T_{pers_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & q & r & 1 \end{bmatrix}.$$

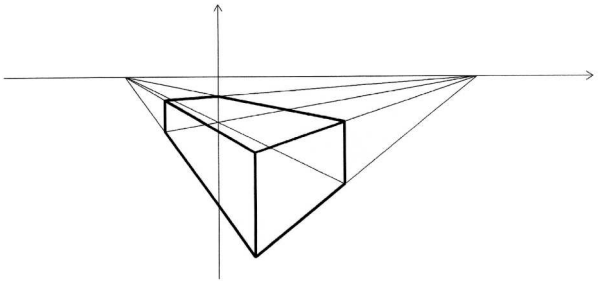
A két centrumpontos perspektivikus vetítést meg lehet kapni az egy centrumpontos vetítésből, ha előzőleg az objektumot elforgatjuk a koordináta-rendszer egyik tengelye körül. A forgatási tengely merőleges arra a tengelyre, amelyen a centrumpont található.

Jelen esetben a két centrumpontos perspektivikus vetítés mátrixa egy x tengely körüli forgatásmátrix és egy egy centrumpontos perspektivikus vetítési

mátrix szorzatából áll:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1/d & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \sin\theta/d & -\sin\theta/d & 1 \end{bmatrix}$$



2.27. ábra. *Perspektivikus vetítés 2 centrumponttal*

Három centrumpontos perspektivikus vetítés esetében a transzformációs mátrix a következőképpen néz ki:

$$T_{pers_3} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p & q & r & 1 \end{bmatrix}.$$

A vetítési sík tetszőleges lehet, nem feltétlenül párhuzamos a koordináta-rendszer egyik síkjával.

Legyen P egy tetszőleges pont a térből, a három centrumpontos perspektivikus vetítéssel a P pont koordinátái a következők lesznek:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p & q & r & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ px+qy+rz+1 \end{bmatrix} = \begin{bmatrix} \frac{x}{px+qy+rz+1} \\ \frac{y}{px+qy+rz+1} \\ \frac{z}{px+qy+rz+1} \\ 1 \end{bmatrix}.$$

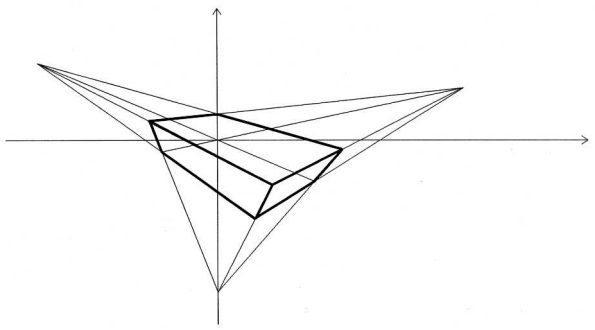
A három centrumpontos perspektivikus vetítést két vagy több, a koordináta-rendszer tengelyei körüli forgatásokból és egy egycentrumpontos perspektivikus

vetítésből kapjuk meg. Pl. az y tengely körüli, majd az x tengely körüli forgatások után, és egy egycentrumpontos vetítés a $z = 0$ síkban a $(0, 0, d)$ centrumponttal a következő transzformációs mátrixot adja [21]:

$$T_{pers_3} = P_{pers_1} \cdot R_x \cdot R_y =$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1/d & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ \sin\beta\sin\alpha & \cos\alpha & \cos\beta\sin\alpha & 0 \\ \sin\beta\cos\alpha & -\sin\alpha & -\cos\beta\cos\alpha & 0 \\ -\sin\beta\cos\alpha/d & \sin\alpha/d & -\cos\beta\cos\alpha/d & 1 \end{bmatrix}$$



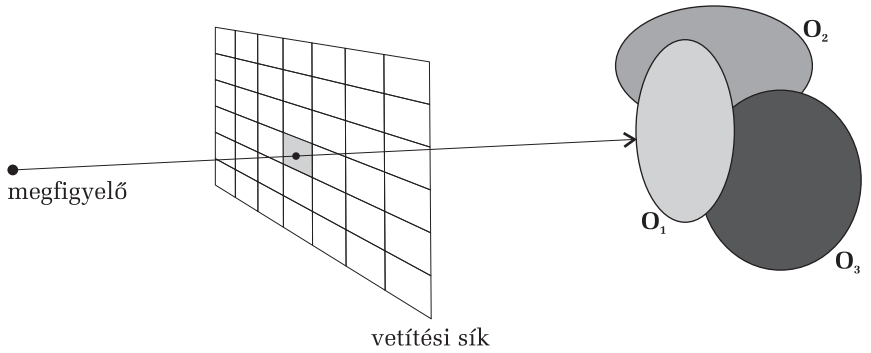
2.28. ábra. *Perspektivikus vetítés 3 centrumponttal*

2.4. A sugárkövetési algoritmus

A sugárkövetési algoritmus a megvilágítási és árnyékolási effektusok által valósághű képet szintetizál a modellről.

Az algoritmus minden pont színének a meghatározásakor (kiszámításakor) figyelembe veszi a lokális és globális megvilágítást. A lokális megvilágítás egy vagy több fényforrástól származik. A globális megvilágítás a háttérfénytől, a felületek tükröződéséből vagy a fénytörésből származik. A sugárkövetési algoritmus a megfigyelő pozíciójától függően generálja a képet, képzeletbeli fénysugarak irányát követve. Ezek a megfigyelőtől indulnak a színtér (objektumok és fényforrások) felé. Az algoritmus felosztja a vetítési síkot egy olyan téglalap-hálóval, amely megfelel a képernyő felbontásának (annyi hálószem lesz a képernyőn,

amekkora a képernyő felbontása). Mindegyik pixelre veszünk egy fénysugarat, amely a megfigyelőtől indul a pixelnek megfelelő téglalap közepébe. Erre a sugárra megvizsgáljuk, hogy metszi-e vagy sem a színtér objektumait (2.29. ábra [24]).



2.29. ábra. *A sugárkövetés elvi vázlata*

A sugárkövetési algoritmus már eleve kiküszöböli a nem látható felületeket, nem számítja ki az objektumok közötti metszeteket, csak egy egyenes és egy objektum közötti metszetet kell meghatározzon, így a színteret egy – a metszetekből származó – ponthalmazzal közelíti.

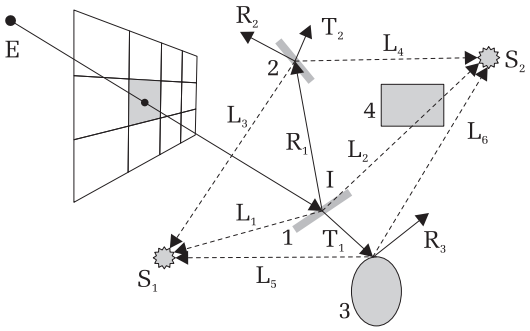
A sugárkövetési algoritmus alapművelete egy egyenes és egy bármilyen típusú objektum metszetének a meghatározása.

A következő algoritmus egy egyszerű sugárkövetést ír le, amely csak a látható felületeket rajzolja ki [24]:

1. Határozd meg a vetítési síkon a kép méretét!
2. A kép minden vonalára végezd el:
3. A vonal minden pixelére végezd el:
4. Határozd meg az R sugarat a megfigyelőtől a pixelig!
5. A színtér minden O objektumára végezd el:
6. Ha létezik, határozd meg R és O metszetét!
7. Határozd meg a megfigyelőhöz legközelebb metszett O_1 objektumot!
8. Rajzold ki a pixelt O_1 színével!

Ez az egyszerű algoritmus az első metszett objektumnál már megáll, és a pixel színe az objektum színe lesz. A metszési pont színének a meghatározásához egy lokális megvilágítási modellt kell használni.

Egy bonyolultabb sugárkövetési algoritmus folytatja a sugár útját. A sugarat, amely a megfigyelőtől indul, *elsődleges sugár*nak nevezzük, azokat pedig, amelyek a metszési pontokból indulnak, *megvilágítási sugarak*nak, *tükrözött és megtört sugarak*nak vagy másképpen *másodlagos sugarak*nak. A megvilágítási sugarak a metszéspontokból indulnak a fényforrások irányába. A tükrözött és megtört sugárra rekurzívan hívjuk a sugárkövetést.



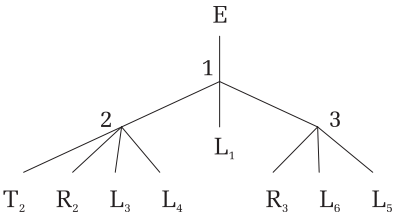
2.30. ábra. *Sugárkövetés visszaverődésekkel és fénytörésekkel*

Az E elsődleges sugár metszi az 1 objektumot az I pontban. Feltételezzük, hogy az 1-es objektum spekulárisan visszaveri a fényt és ugyanakkor áttetsző is. A visszavert sugarat R_l-el jelöljük, a törési sugarat pedig T_1-el. Az R_l és T_1 sugarak a globális megvilágítást határozzák meg az I pontban. A lokális megvilágítás meghatározásához vegyük az L_l és L_2 sugarakat, amelyek a két fényforrás, S_l és S_2 felé tartanak. A fényforrásokat pontszerűnek vesszük. Egy felület egy pontja akkor van megvilágítva egy fényforrás által, ha a fényforrás irányába húzott sugár nem metsz más objektumot. Ha egy pont nincs megvilágítva egy fényforrás által, akkor az a pont az illető fényforrás árnyékzónájában van, és az ambiens fény világítja meg. Mivel az L_l sugár megszakítás nélkül ér el az S_l fényforráshoz, az 1-es objektum direktbe kap fényt az S_1 fényforrástól. Az L_2 sugár metszi a 4-es objektumot, következik, hogy az 1-es objektum nincs megvilágítva az S_2 fényforrás által, ezért az L_2 sugarat nem vesszük számításba. Folytatva a T_1 sugár útját, ez metszi a 2-es objektumot, amely átlátszó. Az R_l sugár és a 2-es objektum metszéspontjából két megvilágítási sugár indul, az L_3 és L_4, a visszaverődési sugár R_2 és a törési sugár T_2. A T_2 sugár elhagyja a színteret és a környező teret metszi, amelynek van egy konstans alapszíne. A T_1 sugár metszi a 3-as objektumot, amely nem átlátszó. Tehát a T_1 sugár és a 3-as objektum metszéséből csak az L_5, L_6 és az R_3 visszaverődési sugár indul ki.

Rekurzívan folytatjuk a visszaverődési és törési sugarak útját addig, amíg egy sugár elhagyja a színteret, vagy a hozzájárulása a pixel színéhez túl kicsi.

Az I metszéspontnak a színéhez az L_l, R_l és T_1 sugarak járulnak hozzá, az 1-es objektum anyagának a fizikai tulajdonságai alapján. Így például ha az 1-es objektumot 30%-os átlátszónak és a 2-es objektumot 50%-osan tükrözőnek definiáljuk, az R_2 sugár 15%-kal járul hozzá az I pont színéhez.

A képernyő egy pixelére vonatkozó sugarakat egy fa típusú adatszerkezettel lehet ábrázolni, amelynek a kiértékelése a levelektől a gyökér felé történik. Egy csomópont intenzitását az alatta levő csomópontok intenzitása határozza meg [21].



2.31. ábra. *A sugarak tárolására szolgáló fa*

Egy sugár intenzitását (I) a Whitted-képlet adja meg [100]:

$$I = I_a + k_d + \sum_{j=1}^{ls} \left(\overline{N} \cdot \overline{L_j}\right) + k_s S + k_t T,$$

ahol:

I_a = az ambiens (környezeti) fény-komponens,
k_d = diffúz (szórt) fény-konstans,
ls = a fényforrások száma,
N = a felület normálvektora (normálisa),
L_j = a j-edik irányba mutató vektor,
k_s = spekuláris (tükrözött) fény-koefficiens,
S = egy R irányból érkező fény-komponens (visszaverődés),
k_t = áthatolt fény-koefficiens (megtört fény),
T = egy P irányból érkező fény-komponens (áthatolás, törés).

A következő algoritmus egy egyszerű rekurzív sugárkövetést ír le [24]. Az *RT-trace* eljárás meghatározza a sugár legközelebbi metszetét egy ponttal és meghívja az *RT-Shade* eljárást a pont árnyékolására. Először az *RT-Shade* kiszámítja az illető pontban az ambiens színt. A következő lépésben egy árnyékolási sugarat indít minden fényforrás irányába, hogy számítsa ki az illető fényforrás hozzájárulását a pont megvilágításához. Egy átlátszatlan objektum megállítja a fényt, egy áttetsző objektum pedig módosítja a fény hozzájárulását. Ha nem vagyunk túl mélyen a sugárfában, akkor rekurzívan hívjuk az *RT-trace* eljárást a tükrözési sugarakra a tükröződő felületek esetében, illetve törési sugarakra az áttetsző felületek esetében.

Az algoritmust fel lehetne gyorsítani úgy, hogy egy kép esetében megőrizzük a sugárfákat. Ez megengedné a felületek fizikai tulajdonságainak a módosítását, mivel a metszéspontokat nem kell újra kiszámítani, viszont a megfigyelő pozíciójának a megváltoztatása már maga után vonja az egész fastruktúrák újraszámolását.

```
1 Válaszd ki a vetítési központot és a vetítési síkot!
2 Minden sorra a képből végezd el:
```

```
3     Minden pixelre a sorból végezd el:
4        Határozd meg a sugarat a vetítési központtól
5     a pixelen keresztül!
6        Pixel := RT_Trace(ray, 1).
7
8  {Metszi a sugarat az objektumokkal, és kiszámítja
9  az árnyékolást a legközelebbi metszésnél.}
10 {Mélység az aktuális mélység a sugárfában.}
11
12 function RT_Trace(sugár: RT_Ray;
13             mélység: integer): RT_Color;
14 BEGIN
15    Határozd meg a sugár legközelebbi metszését
16 egy objektummal!
17    Ha objektum metszve, akkor:
18       Határozd meg a normálist a metszéspontban!
19       RT_Trace := RT_Shade(legközelebbi metszett
20       objektum, sugár, metszéspont, normális,
21       mélység).
22 Különben:
23    RT_Trace := BACKGROUND_VALUE. {a háttérszín}
24 END
25
26 {Kiszámítja a megvilágítást egy pontban az objektumon,
27 követi az árnyékolási, tükröződési és törési sugarakat.)
28
29 function RT_Shade(
30                objektum: RT_Object; {A metszett objektum}
31                sugár: RT_Ray;       {A beeső sugár}
32                pont: RT_Point;      {A metszéspont}
33                normális: RT_Normal; {A normális}
34                mélység: integer;    {A mélység}):
35       RT_Color;
36 var
37    szín: RT_Color; {A sugár színe}
38    rRay, tRay, sRay: RT_Ray; {Tükrözött, megtört és
39    árnyékolási sugarak}
40    rColor, tColor: RT_Color; {Tükrözött és megtört
41    sugarak színei}
42 BEGIN
43    szín := ambiens tényező.
44    Minden fényforrásra végezd el:
```

```
45          sRay := a sugár a pontból a fényforráshoz.
46          Ha a normális és a fényforráshoz néző irány
47          közötti skaláris szorzat pozitív, akkor:
48             Számítsd ki mennyi fényt állítanak meg az
49             átlátszó és nem átlátszó objektumok, és
50             ezt szorozd be a diffúz tényezővel, azután
51             add hozzá a színhez!
52       Ha a mélység < MAX_DEPTH, akkor:
53       {visszatér ha a mélység túl nagy}
54          Ha az objektum tükröző, akkor:
55             rRay := a tükröződési sugár a pontban.
56             rColor := RT_Trace(rRay, mélység+1).
57             szorozd be rColor-t a tükröződési
58             koefficienssel, és add hozzá a színhez!
59          Ha az objektum áttetsző, akkor:
60             tRay := a törési sugár a pontban.
61             tColor := RT_Trace(tRay, mélység+1).
62             Szorozd be tColor-t a törési koefficienssel,
63             és add hozzá a színhez!
64       RT_Shade := szín. {visszatéríti a sugár színét}
65    END
```

A bemutatott sugárkövetési algoritmus egy elsődleges sugarat vesz minden képernyő-pontra (pixelre). A generált kép valósághűségét úgy lehet növelni, ha minden pixelre több elsődleges sugarat használunk. A pixel színét pedig a használt elsődleges sugarak színeinek az átlaga adná meg. Egy pixel színe hozzájárulna a szomszédos pixelek színének a meghatározásához, így *élsimítás (anti-aliasing)* valósul meg.

Az *aliasing* effektus a nagyon kicsi objektumok esetén jelenik meg, amelyeket nem metszenek az elsődleges sugarak. Ezt a *befoglaló test*ek (*bounding volumes*, pl. *befoglaló gömb, téglatest*) használatával tudjuk elkerülni. A befoglaló gömbök körülfogják az objektumot, és eléggé nagyok ahhoz, hogy legalább egy elsődleges sugár metssze őket. Egy ilyen befoglaló test mérete fordítottan arányos a megfigyelő–objektum távolsággal. Ha a sugár metszi a befoglaló testet, de az objektumot nem, akkor folytatjuk a terület felosztását mindaddig, amíg legalább egy sugár nem metszi az objektumot.

2.4.1. Metszéspontok meghatározása

A sugárkövetési algoritmus alapművelete egy sugár és egy objektum közötti metszet kiszámítása. Így egy elsődleges, tükrözött vagy megtört sugárra ki kell számítani a sugár origójához legközelebb álló metszéspontot. A befoglaló testek

használata esetén tesztelni kell, hogy a sugár metszi-e vagy sem a befoglaló testet, és ha igen, csak akkor teszteljük az objektumra.

Egy sugár és egy objektum közötti metszetének a kiszámítására az egyenes parametrikus egyenletét használjuk:

$$\begin{cases} x = x_0 + t\,(x_1 - x_0) \\ y = y_0 + t\,(y_1 - y_0) \\ z = z_0 + t\,(z_1 - z_0) \end{cases},$$

ahol:

(x_0, y_0, z_0) = a vetítési központ (a megfigyelő pozíciója),
(x_1, y_1, z_1) = a pixelnek a képernyőn megfelelő téglalap középpontja.

Ha $t > 1$, akkor az (x, y, z) pont a képernyő túloldalán van, a megfigyelővel ellentett oldalon. A sugarat külön minden objektummal kell metszeni, és amelyik objektumra a legkisebb t-t kapjuk (0-hoz legközelebbi pozitív), az lesz a megfigyelő pozíciójából látható objektum.

2.4.2. A metszéspontok kiszámításának optimalizálása

A legegyszerűbb esetben, amelyben a sugárkövető algoritmus csak az objektumok láthatóságát határozza meg, minden pixelre metszetszámításokat végez egy sugár és a 3D objektumok között. Ily módon, pl. egy 1024×1024 felbontású képernyőn, melyen 10 objektum van megjelenítve, 10 485 760 metszetszámolás szükséges. Ez hosszú időt igényel még egy performens gép esetén is. Elvégzett mérések alapján megállapították [24], hogy a metszetszámolások 75–95% időt vesznek igénybe az egész algoritmus végrehajtási idejéből. Éppen ezért szükséges ezen számolások optimalizálása és ezeknek csökkentése.

Azokban az egyenletekben, amelyekből megkapjuk egy sugár metszéspontját egy objektummal, sok tényező állandó egy sajátos sugárra vagy akár az egész képre nézve. Ezek a tényezők előre kiszámíthatók és felhasználhatók más olyan esetben, amikor a sugár egy ugyanolyan típusú objektumot metsz.

Módszereket fejlesztettek ki egy sugár és egy objektum metszésének gyors meghatározására. Például ha a sugár a z tengely mentén halad, a metszéspontok kiszámítása egyszerűsödik, de ez a feltétel egy transzformációt von maga után mind a sugár, mind azon objektumok esetén, amelyeket metszhet a sugár. A transzformáció által meg kell határozni a legközelebbi objektumot, a z-n történő mélységrendezéssel (*depth-test*). A metszéspont meghatározása után az inverz transzformációt kell alkalmazni az algoritmus folytatásához.

Szabálytalan objektumok esetén a metszési tesztek sok időt emésztenek fel. Ebben az esetben optimalizálni kell. Az egyik optimalizálási lehetőség, ha az illető objektumot beírjuk egy szabályos testbe: gömbbe, ellipszoidba, hengerbe, derékszögű parallelipipedonba stb. A metszetszámítások csak akkor hajtódnak végre, ha a sugár metszi a befoglaló testeket.

A sugárkövető algoritmus gyorsítása befoglaló testek segítségével függ a testek formájától. Például a gömböt gyakran használják befoglaló testként, mivel nagyon egyszerűen lehet kiszámítani a metszéspontját egy sugárral, de egyik hátránya, hogy nem fogja be jól a nyújtott alakú objektumokat, a metszés kiszámításának az egyszerűsége nem szabad az egyedüli kritérium legyen a befoglaló testek meghatározásánál.

Egy objektum metszései kiszámításának költségét a következő összefüggés adja meg:

$$T = b \cdot B + i \cdot I,$$

ahol: T – az összköltség; b – a tesztek száma egy sugár és a befoglaló test között; B – a sugár és a befoglaló test közötti metszési teszt költsége; i – a metszési tesztek száma egy sugár és az objektum között ($i \leqslant b$); I – pedig a sugár és objektum közötti metszési tesztek költsége.

A két tényező függ egymástól. A befoglaló test egyszerűsítése maga után vonja a B csökkenését, de ugyanakkor I növekedését.

Egy egyszerű befoglaló test egy konvex poliéder, amely négy pár párhuzamos síkból áll. Ezek körbefogják az objektumot. A párhuzamos síkok 0°, 45°, 90°, illetve 135°-ra vannak megdőltve a vízszinteshez képest [30].

Egy sugár metszése a befoglaló poliéderrel síkokkal történő metszést von maga után. A metszéseket külön minden párhuzamos sík-párra teszteljük.

Legyen t_1 a sugár és a megfigyelőhöz közelebb álló síkkal levő metszet paramétere, és t_2 a távolabb álló síknak megfelelő paraméter. Feltételezésünk alapján $t_1 < t_2$.

Egy sugár metszetét a befoglaló poliéderrel a t_1 minimum értékek maximuma és a t_2 maximum értékek minimuma adja meg.

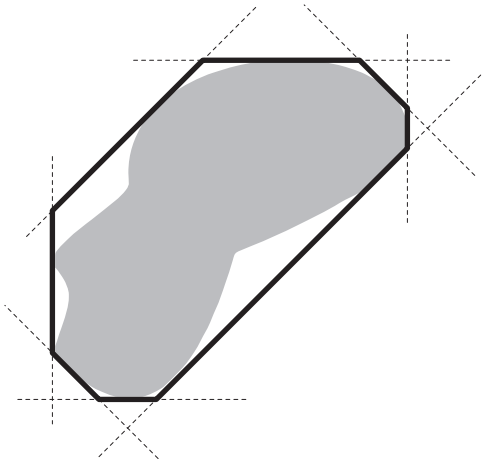
Ha a végső t_1 érték nagyobb lesz, mint a végső t_2 érték, akkor a sugár nem metszi a befoglaló poliédert, és így nem metszi az objektumot sem. Ellentett esetben a sugár metszi a befoglaló testet, és ekkor meg kell vizsgálni, hogy a sugár metszi-e az objektumot vagy sem (a befoglaló test metszése szükséges, de nem elégséges feltétel).

2.5. Árnyalás

Raszteres (pixeles) megjelenítőkön a látható felületek színének és fényerősségének helyes megválasztásával elősegíthetjük a tárgyak alakjának és tömörségének érzékeltetését. Ezt nevezzük *árnyalás*nak.

A háromdimenziós színtér raszterképének valósághűsége az árnyalást előidéző fizikai jelenségek sikeres szimulációjától függ. Árnyalási modellt használunk a felület megjelenítésekor a fényerősség és a szín kiszámításához.

Generatív számítógépes grafikában a következő árnyalási modellek terjedtek el:



2.32. ábra. *Szabálytalan test befoglaló poliéderének meghatározása 0°, 45°, 90°, illetve 135°-os síkokkal (metszet-kép)*

- drótvázas modell,
- árnyalási poliéder használata,
- Gouraud-módszer,
- Phong-módszer,
- pontok független árnyalása.

Drótvázas modell (wireframe) esetén a geometriai modell három- és négyszögekből áll, csak az élvonalak látszanak.

Árnyalási poliéder használata (flat) esetén a megjelenítés a lapok független árnyalásával történik.

A *Gouraud-módszer* folytonos árnyalást állít elő. Mindegyik lap csúcspontjaiban meghatározza a normálisokat, majd ezekből a csúcsok színét. A lapon belüli árnyalást a csúcsponti értékekből interpolálja. A Gouraud-módszer akkor jó, ha a lapon belül a szín valóban közelítőleg lineárisan változik. Ez igaz a diffúz visszaverődésű objektumokra, de elfogadhatatlan tükrös, illetve spekuláris visszaverődésű felületekre. A lineáris interpoláció ilyen esetben egyszerűen kihagyhatja vagy szétkenheti a fényforrás tükröződő foltját.

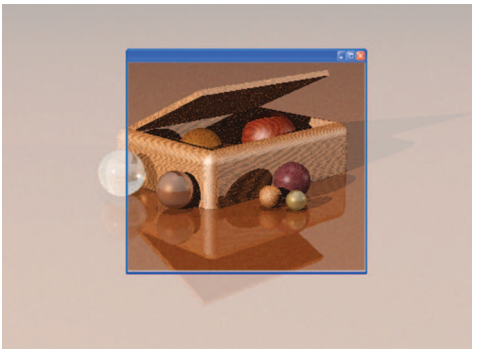
A *Phong-módszer* is folytonos árnyalást állít elő, alapelve, hogy nem a színeket, hanem a normálvektorokat interpolálja, és ebből számítja ki minden pixel színét. Több számítást igényel, de valósághűbb eredményt ad. A Phong-módszer a színtérben nemlineáris interpolációnak felel meg, így nagyobb poligonokra is megbirkózik a tükrös felületek gyorsan változó radianciájával.

*Pontok független árnyalása*kor minden pontban egyenként meghatározzuk a normálvektort és ebből a pixel színét. A legpontosabb, de a leglassúbb számítási modell.

2.33. ábra. *Gömb képe drótvázas, poliéderes, Gouraud-, Phong- és pontonkénti árnyalással*

2.6. A vágás

A generatív számítógépes grafika segítségével előállított grafikus modell a teljes színteret magába foglalja, a számítógép képernyőjén azonban lehet, hogy ennek csak egy része fog látszani, egy része pedig kilóg a képből.



2.34. ábra. *POV-Ray-ben előállított színtér bevágása az ablakba*

A modellből mindig csak annyit szabad mutatni, amennyi látszik. Sem a takart, sem a képből kilógó részek nem szabad felkerüljenek a generált képre.

A következőkben [61] alapján összefoglaljuk a vágás módszereit:

- a modellt vágjuk le a megjelenítés előtt, azaz számítsuk ki a metszéspontokat, és az új végpontokkal rajzoljunk;
- *ollózás*: pásztázzuk a teljes modellt, de csak a látható képpontokat jelenítjük meg (ellenőrizzük minden (x, y)-ra);
- a teljes modell legenerálása egy munkaterületre, majd innen másoljuk át a megfelelő látható részt.

A pontok vágása egyszerű, egy $P(x, y)$ pont akkor látható, ha $x_{\min} \leqslant x \leqslant x_{\max}$, és $y_{\min} \leqslant y \leqslant y_{\max}$, ahol $(x_{\min}, y_{\min})$, $(x_{\max}, y_{\max})$ az ablakot (képernyőt) körbefogó téglalap bal felső és jobb alsó sarkának a koordinátái.

Vonalak, szakaszok esetén elegendő a végpontokat vizsgálni. Ha mindkét végpont belül van, akkor a teljes vonal belül van, nem kell vágni. Ha csak

egy végpont van belül, akkor ki kell számítani a metszéspontot a téglalappal és vágni kell. Ha mindkét végpont kívül van, akkor lehet, hogy nincs közös része a vágási téglalappal, további vizsgálat szükséges. A vágási téglalap minden élére megvizsgáljuk, hogy van-e az élnek közös része az egyenessel (egyenesek metszéspontjának a meghatározása, élen belül van-e a metszéspont?).

Vonalak vágására Cohen és Sutherland adott optimális vágási algoritmust bitkódok segítségével [88].

A Cohen–Sutherland-féle szakaszvágáshoz előzetes vizsgálatokra és kódolásra van szükség.

Legyen a szakasz két végpontja $(x_1,\ y_1)$ és $(x_2,\ y_2)$. Mindkét végpont a következő táblázat szerint kódot kap ($kód_1$, $kód_2$), annak megfelelően, hogy melyik tartományban van:

1001	1000	1010
0001	0000	0010
0101	0100	0110

- Ha a végpontok belül vannak, akkor nincs mit vágni, mindkét végpont kódja: $kód_1 = kód_2 = 0000$ (*triviális elfogadás*).
- Ha mindkét végpont a vágási téglalapon kívül van, ugyanazon az oldalon, vagyis ha $x_1,\ x_2 < x_{min}$ (***1), vagy ha $x_1,\ x_2 > x_{max}$ (**1*), vagy ha $y_1,\ y_2 < y_{min}$ (*1**), vagy ha $y_1,\ y_2 > y_{max}$ (1***), akkor a szakasz nem látható, bittenként $kód_1$ ÉS $kód_2 = 1$ (*triviális elvetés*).
- Különben a szakasz metszi a vágási téglalap valamelyik oldalát, tehát egy részét ki kell rajzolni (el kell fogadni), egy részét pedig nem (el kell vetni). Ekkor vegyünk egy külső végpontot (legalább az egyik az, ha mindkettő az, válasszuk az elsőt felülről lefelé és jobbról balra haladva), számítsuk ki a metszéspontot, a két részre bomlott szakasz egyik fele pedig az előbbi pont alapján triviálisan elvethető.

A Cohen–Sutherland-féle szakaszvágás gyakorlati megvalósítását lásd a 6.4. (*Az OpenGL lehetőségei, alapvető rajzolási algoritmusok*) fejezetben.

Körök, ellipszisek vágására fogjuk körbe az alakzatot egy kerettel (írjuk be a kört egy négyzetbe, az ellipszist egy téglalapba). Ha a keret belül van, akkor a kör vagy ellipszis is belül van, nem kell vágni. Ha a keret kívül van, akkor az alakzat is kívül van, nincs mit vágni. Különben: körnegyedekre (nyolcadokra) megismételjük a kör és él metszéspont-meghatározó eljárást, majd pásztázunk.

Sokszögek vágásánál sok esetet kell megvizsgálni. A Sutherland–Hodgman-algoritmus szerint sorba vesszük az éleket és egyenként vágunk.

2.7. A fraktálok világa

A *fraktál*ok *önhasonló*, végtelenül komplex matematikai alakzatok, amelyek változatos formáiban legalább egy felismerhető (tehát matematikai eszközökkel leírható) ismétlődés tapasztalható. Az elnevezést 1975-ben Benoît Mandelbrot adta, a latin *fractus* (vagyis törött; törés) szó alapján, ami az ilyen alakzatok tört számú dimenziójára utal. „A természet geometriájának fraktál arculata van" – vallotta Mandelbrot.

Az önhasonlóság azt jelenti, hogy egy kisebb rész felnagyítva ugyanolyan struktúrát mutat, mint egy nagyobb rész. Ilyen például a természetben a villám mintázata, a levél erezete, a felhők formája, a hópelyhek alakja, a hegyek csipkézete, a fa ágai, a hullámok fodrozódása és még sok más. „Hogy fölfrissülj a nagy Egészben, lásd meg az Egészet minden kicsi részben" – írta Goethe.

A fraktál szóval rendszerint az önhasonló alakzatok közül azokra utalunk, amelyeket egy matematikai formulával le lehet írni vagy meg lehet alkotni.

A generatív számítógépes grafikában fraktálok segítségével tudunk leírni olyan objektumokat (pl. felhők, hegyek, növények stb.), amelyek egyszerű geometriai formáknak nem felelnek meg.

Matematikailag a fraktál egy olyan halmaz, amelynek a *fraktál dimenzió*ja nagyobb a topológiai dimenziójánál (törtdimenziós).

Az euklideszi geometriában egy alakzat térbeli kiterjedtségét egy pozitív egész szám, az ún. *dimenzió* jellemzi. A pontnak nincs kiterjedése, tehát a dimenziója 0. Az egyenes egydimenziós, mivel egyetlen irány szerinti kiterjedése mérhető. Egy síkidomnak a síkbeli kiterjedése két egymásra merőleges irány mentén mérhető (kétdimenziós). A bennünket körülvevő világ háromdimenziós, a matematikában pedig nagyobb dimenziók is léteznek.

Egy H halmaz *topológiai dimenziója* k, ha minden pontjának van olyan tetszőlegesen kis környezete, aminek a határa H-ban egy k–1 dimenziós halmaz és k a legkisebb ilyen tulajdonságú nemnegatív egész. A topológiai dimenzió mindig egész szám.

Szigorúan önhasonló halmazok összetettségének jó mérőszáma az ún. *hasonlósági dimenzió*. Ha $f_1,...,f_n$ hasonlósági transzformációk, amelyeknek hasonlósági tényezői $r_1,...,r_n$ számok és K az $(f_1,...,f)_n$ függvényrendszer invariáns halmaza, akkor azt a pozitív s számot, amelyre teljesül az $r_1^s + \cdots + r_n^s = 1$ egyenlőség, a K halmaz hasonlósági dimenziójának nevezzük.

Azt is mondhatjuk, a dimenzió azt jelenti, hogy milyen hatvány szerint aránylik a méret a nagyításhoz.

Tegyük fel, hogy a H halmaz N darab hasonló részből áll, amelyek s-szeres ($s < 1$) nagyításai H-nak. Ekkor:

$$D(H) = \frac{\log N}{\log s}.$$

2.35. ábra. *Fotorealisztikus táj – fraktálok segítségével*

Induljunk ki egy szakaszból. Ha az eredeti szakaszt az N-ed részére kicsinyítjük (skálázzuk), akkor ebből az új szakaszból pontosan N (vagyis N^1) darabra van szükség, ha le akarjuk fedni velük az eredeti szakaszt. Ha egy négyzetet kicsinyítünk az N-ed részére, akkor pontosan N^2 darab, kocka esetén N^3 darab kicsinyített másra van szükségünk. Könnyen észrevehetjük, hogy a kitevőben lévő szám az objektum euklideszi (vagy topológiai) dimenziójával egyezik meg. Ha a kitevő értéke valós szám, tetszőleges önhasonló alakzat dimenziója kiszámítható az alábbi módon:

$$D = \lim_{\varepsilon \to 0} \frac{\log N(\varepsilon)}{\log \frac{1}{\varepsilon}},$$

ahol $N(\varepsilon)$ darab méretű alakzatra (az eredeti objektum skálázott változataira) van szükség a teljes, eredeti objektum letakarásához. Az így bevezetett dimenziófogalom a *Haussdorf-dimenzió*.

Fraktálokra jellemző az ún. *box-dimenzió* (vagy *doboz-dimenzió*) is.

A box-dimenzió meghatározásához egy négyzetrácsot (magasabb dimenzióban kockarácsot stb.) kell a vizsgált alakzatra helyeznünk.

Ezután meghatározzuk azon cellák minimális számát, amelyek segítségével az alakzatunk lefedhető. Ha ezzel megvagyunk, finomítsuk a rácsot, használjunk például fele akkora cellaméretet, mint kezdetben. A lefedéshez szükséges cellák száma így nyilvánvalóan megnő, számunkra most az az érdekes, hogy mennyivel. Egy egyenes szakasz esetében a fele akkora cellákból kétszer annyira, míg síkidomok esetén négyszer annyira lenne szükség.

Jelöljük N-nel egy adott alakzat lefedéséhez szükséges cellák számát és jelölje r az alkalmazott cellaméretet. Ekkor a következő összefüggés érvényes:

$$N = r^{-D_B},$$

ahol a kitevőben szereplő D_B-t az alakzat *box-dimenzió*jának nevezzük.

Rendezve az egyenletet:

$$D_B = \frac{\log N}{\log \frac{1}{r}}.$$

Előnye, hogy nem szükséges egzakt önhasonlóság a használatához, így akár ún. *önaffin alakzat*ok dimenziójának mérésére is felhasználható.

2.7.1. Lineáris fraktálok

A *Cantor-halmaz* megszámlálhatatlanul végtelenül sok pontból álló halmaz, amelynek a teljes hossza 0 (Cantor-por, 1877).

2.36. ábra. *A Cantor-por*

Hány dimenziós a Cantor-halmaz? Nem 1 dimenziós, mert nincs hossza, de nem is 0 dimenziós, mert a pontok kontinuumot alkotnak. A Cantor-halmaz dimenziója: D = ln(2)/ln(3) ≅ 0,6309.

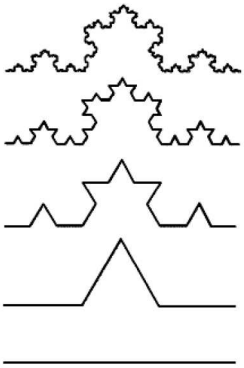
Iteratívan így állíthatjuk elő a Cantor-halmazt: *vegyünk egy n hosszúságú szakaszt (1. lépés), rajzoljuk ki (2. lépés), osszuk fel három egyenlő részre (3. lépés), tartsuk meg az első részt és az utolsót, a középsőt pedig vessük el (4. lépés), egy adott iterálási értékig folytassuk a 2. lépéstől az első és az utolsó megmaradt szakaszra.*

A *Koch-görbé*t Helge von Koch svéd matematikus 1904-ben példaként hozza fel olyan görbére, amelynek semelyik pontjába nem húzható érintő. Dimenziója kb. 1,261. A gyakorlati megvalósítását lásd a 6.6. (*Fraktálok*) fejezetben.

A *Sierpinski-szőnyeg*et 1915-ben alkotta meg Wacław Sierpiński lengyel matematikus. Az alakzatban harmadakkora részekből nyolcat hagyunk meg, így a szőnyeg dimenziója: ln(8)/ln(3) = 1,8928. Háromszög alakú változata a *Sierpinski-háromszög*, 3D változata a *Menger-szivacs*.

Iteratívan így tudjuk előállítani a Sierpinski-háromszöget:

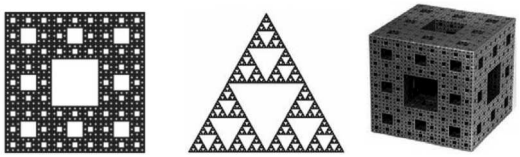
- Vegyünk egy tetszőleges méretű szabályos háromszöget. Rajzoljuk be a középvonalait.
- Ezek a szakaszok 4 kisebb (egybevágó) háromszögre osztják fel az eredetit. Ezek közül:



2.37. ábra. *A Koch-görbe iteratív generálása*

- távolítsuk el a középsőt. A maradék hárommal ismételjük meg a fentieket.

– Egy elég nagyszámú iterációs lépés után az eredmény a Sierpinski-háromszög lesz.



2.38. ábra. *A Sierpinski-szőnyeg, háromszög, valamint a Menger-szivacs*

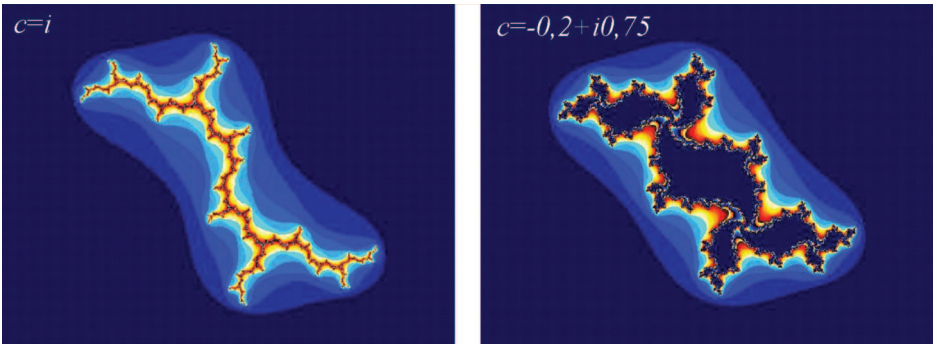
A fent említett fraktálok az ún. *lineáris fraktál*ok. A fraktálok úgy generálhatók, hogy az önhasonlóságot jellemző mintázatot ismételjük egyre kisebb méretarányokban (azaz nagyobb felbontásban). Ha az egyes felbontások között az átmenetet affin transzformációkkal képezzük, akkor lineáris fraktálokat kapunk.

2.7.2. Komplex fraktálok

Ha az egyes iterációs lépésekben nem lineáris leképzést alkalmazunk, akkor nem lineáris fraktálokat kapunk eredményül.

A *Julia-halmaz*ok Azon z komplex számok halmaza, amelyekre a $z_0 = z$, $z_{i+1} = z_i^2 + c$ iteráció eredménye nem a végtelenbe konvergál (c itt egy komplex paraméter, azaz minden c-hez tartozik egy Julia-halmaz).

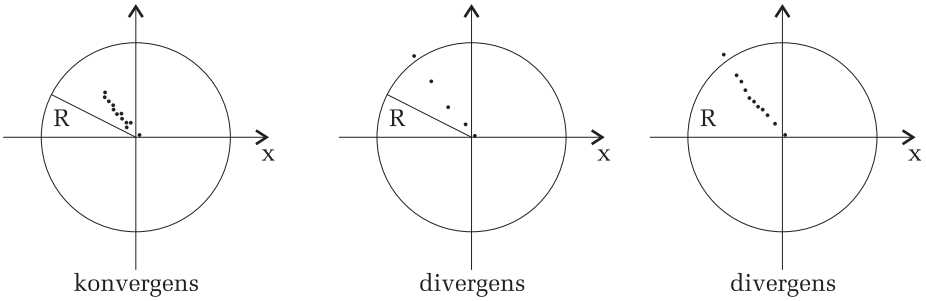
Gaston Julia (1893–1978) francia matematikusról kapták a nevüket, aki 1918-ban megjelent munkájában ismertette őket.



2.39. ábra. *Julia-halmaz a $c = i$, valamint a $c = 0{,}2 + i0{,}75$ konstansokra*

A Mandelbrot-halmaz azon c komplex számok halmaza, amelyekre a $z_0 = 0$, $z_{i+1} = z_i^2 + c$ iteráció eredménye nem a végtelenbe konvergál. ($|c| \leqslant 2$)

A fekete-fehér Mandelbrot-halmazt könnyű kirajzolni, hisz a fenti definíció értelmében szinte egyértelműen adódik az algoritmus (ábrázoljuk a c komplex számot az x, y koordináták segítségével).



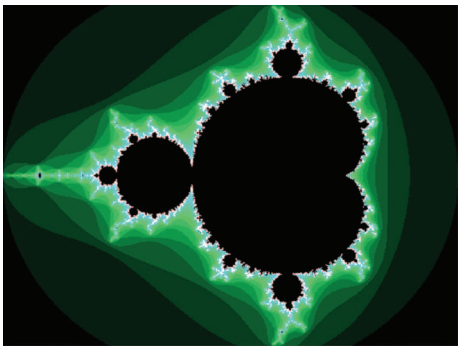
2.40. ábra. *Konvergencia és divergencia*

A színes Mandelbrot-halmaz megvalósításához értelmezzük először a *divergencia* fogalmát.

Definiáljunk egy kört: $x^2 + y^2 = R$, iteráljunk az (x_0, y_0) pontból kiindulva K-szor, ha egy pont a körön kívülre kerül, akkor divergens, különben konvergens.

A divergencia fogalmának ismeretében definiáljuk a *szökési idő* fogalmát: azon L lépésszám, amikor egy pont a körön kívülre kerül. Minél kisebb az L, a pont annál gyorsabban kerül a végtelenbe.

Minden ponthoz rendeljünk hozzá egy színt az L szökési idő függvényében, és megvan a színes Mandelbrot-halmazunk.



2.41. ábra. *Mandelbrot-halmaz*

2.7.3. L-System fraktálok

1968-ban Aristid Lindenmayer, magyar származású elméleti biológus és botanikus alkotta meg a róla Lindenmayer-rendszernek, röviden L-Systemnek nevezett formális leírási módszert. Különböző növények növekedését vizsgálva rájött, hogy közülük igen sok leírható néhány egyszerű formális szabállyal. Ehhez csak egy megfelelő generatív grammatikát kell rögzíteni.

Egy generatív grammatika a $G = < N, T, S, P >$ rendezett négyes, ahol:

- N: ábécé, változók halmaza (nemterminálisok),
- T: konstansok (terminálisok) halmaza,
- S: kezdőszimbólum,
- P: levezetési szabályok halmaza.

2.42. ábra. *A Barnsley-páfrány*

Például a Koch-görbét a következőképpen lehet leírni:

- változók: S,

– konstansok: +, –,
– kezdőszimbólum: S,
– szabályok: $S \to S + S – S – S + S$.

Az S rajzolást, a + jel kilencven fokkal balra, a – jel pedig kilencven fokkal jobbra történő fordulást jelent.

A Barnsley-páfrányt így írhatjuk le:

– változók: S, F,
– konstansok: +, –, [,],
– kezdőszimbólum: S,
– szabályok: S $\to$ F–[[S]+S]+F[+FS] –S, F $\to$ FF,
– fordulási szög: 25°.

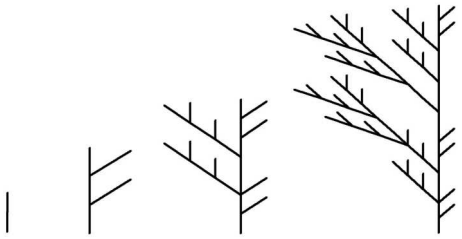
Az F rajzolást, a – jel adott szöggel balra, a + jel jobbra fordulást jelent. Az S-hez semmilyen rajzolási művelet nem kapcsolódik, a szerepe a különleges forma kialakításában van. A [menti az aktuális pozíció- és szög-értékeket, amelyek a] jellel visszatölthetők.

Speciális L-System fraktálok a *graftál*ok. A *graftál*ok egyszerű szabályokból iteratív eljárással létrehozott alakzatok, amelyek növényeket modelleznek.

– Legyen egy négy jelből álló nyelv: 0, 1, [,].
– A [-t mindig követi egy], a] előtt mindig áll egy [.
– A [ ] páros között egy vagy több jel is állhat.
– A 0 és 1 jelentése: lépj előre egy egységnyit.
– A [jelentése: jegyezd meg az aktuális pozíciót és irányt, majd fordulj el meghatározott szöggel.
– A] jelentése: menj vissza és fordulj a legutóbb megjegyzett pozícióba és irányba.

A graftál iterálása (pl.):

– Cseréljünk ki minden 0-át 1[0]1[0]0-ra.
– Cseréljünk ki minden 1-et 11-re.



2.43. ábra. *Példa graftálra*

2.7.4. IFS fraktálok

Az IFS az *Iterated Function System* (iterált függvényrendszer) kifejezés rövidítése. Egy IFS nem más, mint kontraktív, $R^2 \rightarrow R^2$ alakú transzformációk kollekciója, mely szintén egy leképezés. Az ilyen típusú leképezéseknek mindig van egy egyedi fixpontja, digitális képekre alkalmazva ez a fixpont általában egy fraktálkép. IFS-sel előállítható a fent említett fraktálok nagy része, pl: Cantor-halmaz, Sierpinski-háromszög és -szőnyeg, Koch-görbe.

1988 Barnsley kidolgozott egy módszert, amely digitális képek jó hatásfokú tömörítését tette lehetővé IFS-ek felhasználásával [6].

A Barnsley-páfrányt úgy állíthatjuk elő IFS-ként, hogy kiindulunk az origóból ($x_0 = 0$, $y_0 = 0$), kirajzoljuk a pontot, majd véletlenszerűen alkalmazunk egy transzformációt a következő négyből (pl. 300 000-szer), a kapott új pontokat kirajzoljuk:

1. $\begin{cases} x_{n+1} = 0 \\ y_{n+1} = 0{,}16 \cdot y_n \end{cases}$, ezt a transzformációt 1%-os valószínűséggel alkalmazzuk.
2. $\begin{cases} x_{n+1} = 0{,}2 \cdot x_n - 0{,}26 \cdot y_n \\ y_{n+1} = 0{,}23 \cdot x_n + 0{,}22 \cdot y_n + 1{,}6 \end{cases}$, 7%-os valószínűséggel.
3. $\begin{cases} x_{n+1} = -0{,}15 \cdot x_n + 0{,}28 \cdot y_n \\ y_{n+1} = 0{,}26 \cdot x_n + 0{,}24 \cdot y_n + 0{,}44 \end{cases}$, 7%-os valószínűséggel.
4. $\begin{cases} x_{n+1} = 0{,}85 \cdot x_n + 0{,}04 \cdot y_n \\ y_{n+1} = -0{,}04 \cdot x_n + 0{,}85 \cdot y_n + 1{,}6 \end{cases}$, 85%-os valószínűséggel.

A gyakorlati megvalósítását lásd a 6.6. (*Fraktálok*) fejezetben.

2.7.5. A „káosz-játék” fraktálok

A „káosz-játék” fraktálok előállításának egy „játékosabb módja”, például a Sierpinski-háromszög, előállítható a következőképpen:

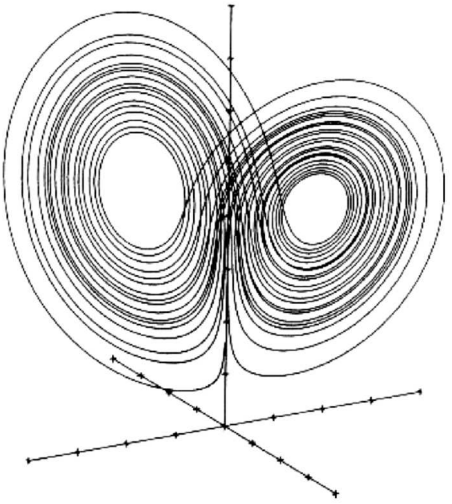
- Vegyünk fel három pontot a síkban, úgy, hogy azok egy egyenlő szárú háromszög csúcsait határozzák meg.
- Címkézzük fel ezeket a pontokat az 1, 2, 3 számokkal. Ezeket bázisoknak fogjuk hívni.
- Ezután kezdetét veszi a játék. Vegyünk fel egy tetszőlegesen kiválasztott pontot a három bázispont által meghatározott háromszögön belül. Ezt játékpontnak fogjuk nevezni. Majd újabb és újabb pontokat veszünk fel, a következő szabály szerint: Sorsoljunk ki véletlenszerűen egy 1 és 3 közötti számot.
- Tegyük fel, hogy a kisorsolt szám az x volt. Ekkor kössük össze képzeletben a játékpontunkat az x címkéjű bázisponttal, és vegyük fel új játékpontként az így kapott szakasz felezőpontját.

– Ha elegendően sok pontot vettünk fel, akkor tisztán felismerhető lesz a Sierpinski-háromszög jellegzetes alakja.

2.7.6. Különös attraktorok

Edward Lorenz amerikai meteorológus 1963-ban egy egyszerű időjárási modell felállításával próbálkozott. Az alábbi egyenletrendszert vizsgálta:

$$\begin{cases} \dot{x} = \sigma\,(y - x) \\ \dot{y} = r|x - y - xz \\ \dot{z} = -bz + xy \end{cases}$$



2.44. ábra. *A Lorenz-attraktor*

Észrevette, hogy r = 28, σ = 10, b = 8/3 paraméterek mellett kis kezdeti feltételekbeli különbség esetén is igen eltérő időfejlődés tapasztalható. Amikor a rendszer viselkedését fázistérben ábrázolta, egy igen furcsa attraktor képe bontakozott ki a szemei előtt. Ez a róla Lorenz-attraktornak elnevezett különös ábra azóta a káosz egyik jelképévé vált.

2.7.7. Véletlen fraktálok

Láttuk, hogy már az IFS-fraktáloknál nagy szerepe van a véletlennek, a valószínűségnek: a megadott transzformációkat csak egy bizonyos valószínűséggel alkalmazzuk.

A valóságmodellezéskor is nagy szerephez jutnak a véletlen fraktálok, hisz a természet alkotta valós objektumok nem teljesen szabályosak.

A véletlen fraktálok vagy véletlen halmazokból veszik fel értékeiket, vagy egy generált véletlen-számmal perturbáljuk a fraktál értékét, vagy valamilyen más szinten kötődnek a véletlenhez, pl. a Brown-féle mozgás pályájának a fraktál jellegű tulajdonságait használjuk fel.

A valóság modellezésében felületeket, felhőzetet, atmoszférikus effektusokat stb. nagyon jól elő tudunk állítani *Perlin-zaj* [76] alkalmazásával.

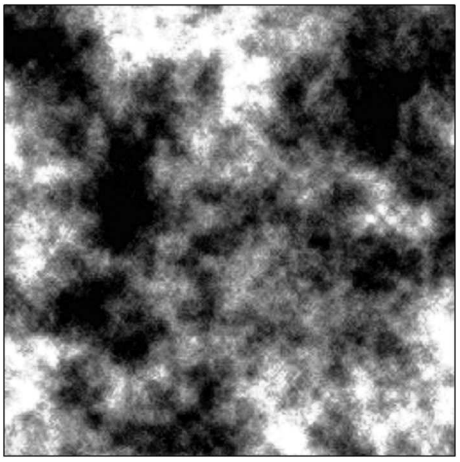
Perlin zajfüggvénye R^n-en értelmezett ($f : R^n \rightarrow [-1,\ 1]$), az egész számokban csomópontokat képző rácshoz igazított pszeudo-véletlen spline függvény, amely a véletlenszerűség hatását kelti, de ugyanakkor rendelkezik azzal a tulajdonsággal, hogy azonos bemeneti értékekre azonos függvényértéket térít vissza. A gyakrabban használt n értékei 1 – animáció esetén, 2 – egyszerű textúrák, 3 – bonyolultabb 3D textúrák, 4 – animált 3D textúrák (pl. mozgó felhők).

A következőképpen generálhatunk Perlin-zajt: adott egy bemeneti pont. Minden környező rács-csomópontra választunk egy pszeudo-véletlen értéket egy előre generált halmazból. Interpolálunk az így megkapott csomópontokhoz rendelt értékek között, valamilyen S görbét használva (pl. $3t^2 - 2t^3$).

Ha a Perlin-zajfüggvényt kifejezésben használjuk, különböző procedurális mintákat és textúrákat hozhatunk létre.

Ha ezeket a kifejezéseket fraktál-összegben használjuk, minden iterációban új adatot vihetünk be, amely valamilyen módon befolyásolja a teljes képet. Például domborzatgenerálás esetén, az iteráció során a fraktál dimenzióját akarjuk befolyásolni, azaz minden iterációban az amplitúdót osztani fogjuk egy bizonyos értékkel.

A gyakorlati megvalósítását lásd a 6.13. (*Effektusok*) fejezetben.



2.45. ábra. *Felhőzet Perlin-zajjal*

2.8. Animáció

Az *animáció* olyan filmkészítési technika, amely élettelen tárgyak (többnyire bábok) vagy rajzok, ábrák stb. „kockázásával" olyan illúziót kelt a nézőben, mintha az egymástól kis mértékben eltérő képkockák sorozatából összeálló történésben a szereplők megelevenednének vagy élnének. Ahhoz, hogy a szemünk folytonosnak lássa a mozgást, minimum 16 képkockára van szükség másodpercenként.

A XX. század végére az animáció számos válfaja elkülönült: rajzfilm, árnyfilm, kollázsfilm, bábfilm, gyurmafilm, homok- vagy szénporfilm, festményfilm, fotóanimáció, tárgymozgatás, számítógépes animáció, Flash animáció, trükkfilm, pixilláció, stop motion.

A *CGI* (*Computer-Generated Imagery*) a film és egyéb vizuális médiumok készítése során alkalmazott számítógépes grafika legelterjedtebbé vált elnevezése mind 2D-ben, mind 3D-ben.

A videojátékok leggyakrabban valósidejű számítógépes grafikát használnak, de esetenként tartalmaznak előrenderelt „vágott jeleneteket", illetve intrófilmeket, amelyek tipikusan CGI-alkalmazások: FMV-k (Full Motion Video).

Filmkészítés során filmtrükköt igénylő jelenetekhez azért használják egyre gyakrabban a digitális technika, s így a CGI nyújtotta lehetőségeket, mert:

- magasabb minőséget eredményez,
- jóval tudatosabban irányítható,
- olyan képek kidolgozására is alkalmas, melyeket semmilyen más technológiával nem lehet létrehozni,
- lehetővé teszi egy művésznek, hogy akár színész, költséges díszletek vagy kellékek nélkül dolgozzon.

Az első filmbeli 2D képalkotás 1971-ben jelent meg *Az Andromeda-törzs* (The Andromeda Strain) című filmben (Michael Crichton).

2D-s CGI-t először 1973-ban használtak a *Feltámad a vadnyugat* (Westworld) című filmben (Michael Crichton).

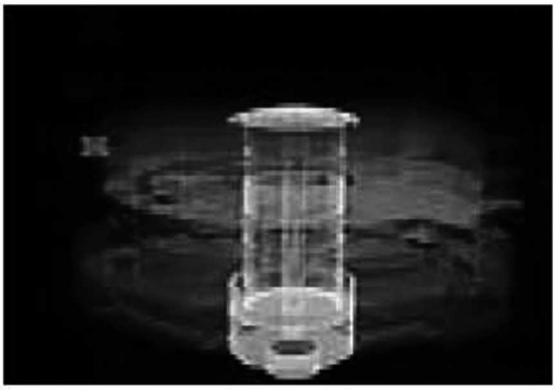
Az első 2D-s, teljesen számítógépes animációval készült film a 11 perces kanadai *The Hunger* (1974) volt [38].

3 dimenziós kép először a *Futureworld*ben (1976) volt látható, ahol egy számítógép által generált kezet és arcot alkotott Edwin Catmull és Fred Parke a számítógépes grafika fellegvárában, a utahi egyetemen.

Az első film, amelyben 3D számítógépes animációt használtak, a *Csillagok háborúja* (1977) volt, ahol a Halálcsillag tervrajzai követelték a beavatkozást.

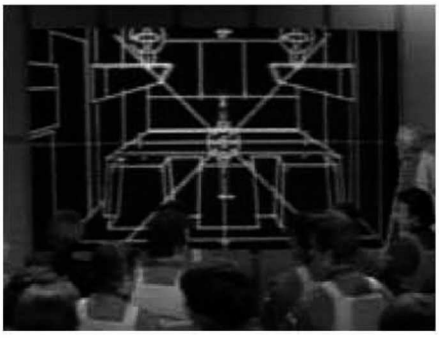
Az első CGI karakter az 1982-ben bemutatott *Tron* c. filmbeli *Bit* volt (egy poliéder).

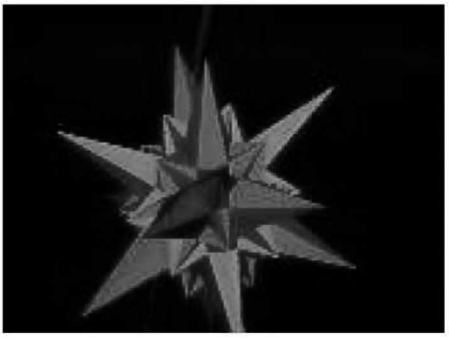
CGI-t használtak az *Utolsó csillagharcos* (The Last Starfighter) c. filmben is (1984). Azonban e két film pénzügyi kudarcnak bizonyult, így rövid időre száműzték a CGI-t, s helyette olyan képeket alkalmaztak, amelyek csak azt a látszatot



2.46. ábra. *Az Andromeda-törzs; Feltámad a vadnyugat*

2.47. ábra. *The Hunger; Futureworld*





2.48. ábra. *Csillagok háborúja; Bit*

keltették, mintha számítógép alkotta volna őket (a technikai körülmények még nem voltak megfelelők).

1986-ban megalakult a Pixar stúdió, amely saját renderelő szoftver gyártásába kezdett (RenderMan), és amely kimagaslón teljesített a rajzfilmek területén.

2.49. ábra. *Utolsó csillagharcos; Pixar*

Az első ember alakú CGI karakter 1985-ban jelent meg a *Sherlock Holmes és a félelem piramisa* (Young Sherlock Holmes) c. filmben (John Lasseter). A karakter egy festett üvegablakból összeállt lovag formájában jelent meg a vásznon.

2.50. ábra. *Young Sherlock Holmes; The Abyss*

A fotorealisztikus CGI egészen 1989-ig nem jelent meg a filmiparban, amikor is *A mélység titka* (The Abyss) elnyerte a legjobb vizuális effektusokért járó Oscar-díjat, a vízlény tökéletes CGI karakter volt.

Ezt követően a CGI a *Terminátor 2*-ben (1991) kapott központi szerepet, ahol a T-1000-es terminátor folyékony fém-mivoltával és alakváltó effektusaival kápráztatta el a közönséget, melyek szerves részét alkották a film akciójeleneteinek. A *Terminátor 2* szintén meghozta az ILM-nek az Oscar-díjat a különleges hatásokért. A két utóbbi filmet James Cameron rendezte.

Az 1993-as *Jurassic Park* dinóinak életszerű megjelenése, mely hibátlanul ötvözte a CGI-t és a live-actiont, hozta meg a filmipar forradalmát. E pont jelentette Hollywood áttérését a stop-motion animációról és a hagyományos optikai effektusokról a digitális technikákra.

2.51. ábra. *Terminátor 2; Jurassic Park*

1994-ben a CGI-t hasznosították a *Forrest Gump* különleges hatásainak megalkotására. A leginkább megjegyzendő trükk a filmben Gary Sinise színész lábainak digitális módon történő eltávolítása volt, vagy a napalmtámadás, a gyorsan mozgó pingponglabdák és a madártoll a nyitójelenetben.

2.52. ábra. *Forrest Gump; Szépség és a szörnyeteg*

A 2D CGI használata folyamatosan nőtt a hagyományos animációs filmek készítésénél, ahol a kézzel rajzolt filmkockákat egészítették ki vele. Széles skálán mozgott az alkalmazása, a két képkocka közti átmenet elősegítésére használt animációktól egészen a szemkápráztató 3D-s effektusokig, amilyen a *Szépség és a szörnyeteg* báltermi jelenete.

1995-ben az első teljes egészében számítógép alkotta mozifilm, a Pixar cég és a Walt Disney produkciója, a *Toy Story* zajos sikereket ért el.

2.53. ábra. *Toy story*

A CGI a filmekben általában 1,4–6 megapixellel renderelt. A *Toy Story*t például 1536×922 (1,42 MP)-vel renderelték. Egy képkocka renderelése jellemzően 2–3 óra körüli időt vesz igénybe, a legbonyolultabb jeleneteknél ennek tízszerese is előfordulhat. Ez nem sokat változott az utóbbi évtizedben, mert a képminőség azonos szinten halad előre a hardverfejlődéssel, mivel gyorsabb gépekkel egyre összetettebb megvalósítás válik lehetővé. A GPU (a videokártya processzora) feldolgozási erejének exponenciális növekedése, illetve a CPU erejének, tárolási kapacitásának és memória sebességének és méretének jelentős emelkedése rendkívül kiszélesítette a CGI lehetőségeit.

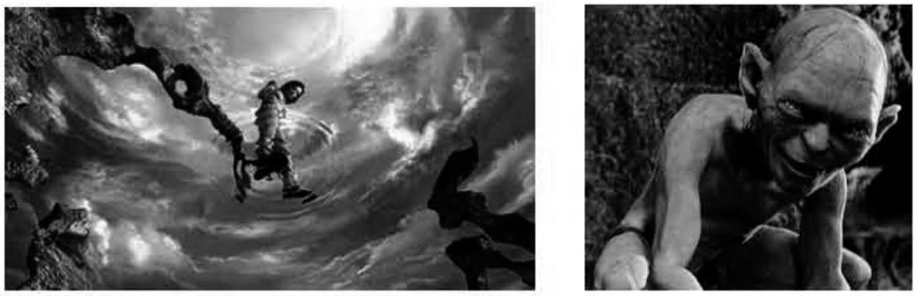
1995 után további animációs cégek jöttek létre: Blue Sky Studios (Fox), a DNA Productions (Paramount és WB), az Onation Studios (Paramount Pictures), a Sony Pictures Animation (Columbia Pictures), a Vanguard Animation (Walt Disney Pictures, LGF és 20th Century Fox), a Big Idea Productions (UP és FHE Pictures) és a Pacific Data Images (DreamWorks), illetve már létezők (Walt Disney Pictures) tértek át a hagyományos animációról a CGI-re.

1995 és 2005 között a széles körben vetített filmek effektusköltségeinek átlaga 5 millió dollárról 40 millió dollárra ugrott.

2005-ben a filmek több mint felénél alkalmaztak jelentősebb effektusokat.

2001-ben a Square Pictures megalkotta a *Final Fantasy – A harc szelleme* című CGI-filmet, amely magas szinten részletezett és fotorealisztikus grafikát vonultatott fel.

Gollam karaktere *A Gyűrűk Ura*-trilógiából teljes egészében CGI-vel készült, motion capture segítségével.



2.54. ábra. *Final Fantasy; Gollam*

A számítógépes játékok és 3D-s videokártyák fejlesztői igyekeznek elérni ugyanazt a vizuális minőséget személyi számítógépeken valós időben (real-time), melyet a CGI-filmeknél és animációnál használnak.

A real-time renderelési minőség villámgyorsaságú előrehaladásával a művészek elkezdték alkalmazni a videojátékok szoftverének magját alkotó *game engine*-t a nem interaktív filmek renderelésére. Ezen művészeti formát *machinimá*nak nevezik.

Az animációs film nem csupán állóképek sorozata. Nagy szerepe van a ritmusnak, plánozásnak, világításnak, vágásnak stb. Karaktereink gondos mozgatásával gondolatokat, érzelmeket vagy történeteket mesélhetünk el. Az animáció nem más, mint a mozgás művészete.

Éppen ezért az animáció pár alapelvre épül [97]:

Nyúlás és összenyomódás: Gyorsulás és lassulás esetén a kőkemény testeken kívül minden tárgy és élőlény megváltoztatja alakját. Ennek alapvető oka a szerkezetek rugalmassága. Ezt a jelenséget kicsit felnagyítva jól tudjuk érzékeltetni a sebességet, gyorsulást és a testek merevségét. Alapszabály, hogy a megnyúló vagy összenyomódó tárgyak térfogata nem változik.

Időzítés: A mozgások időzítésével, egy-egy mozgás sebességével nagyon sok mindent el tudunk mondani: érzelmeket, hangulatokat vagy éppen fizikai súlyt, méretet. Egy erős óriás vagy egy szomorú ember mozgását teljesen más ritmusúra hangoljuk, mint egy apró vidám törpéét.

Előkészítés: Nagyon fontos, hogy a néző ne maradjon le semmiről. Ha túl gyorsan történik valamilyen cselekmény – kellő előkészítés nélkül –, az egész mozgás hatás nélkül marad. Egy mozgás előkészítésének lényege, hogy a néző figyelmét valamilyen mozdulattal odairányítsuk. Gondoljunk arra, hogy a valóságban is minden pofon előtt lendületet vesz a kéz! Hasonlóképpen egy animált autó vagy figura is mindig „nekiveselkedik" a gyors mozgásnak.

Levezetés: A mozgások soha nem állnak hirtelen „csak úgy" le. Az autó a fékezésnél megbillen, a labdát dobó kéz továbblendül stb. A mozgások levezetése gyakran a következő mozdulat előkészítése.

Beállítások: Mindig cél, hogy a néző észrevegye azt, amit üzenni szeretnénk. Ha túl sok dolog elvonja a figyelmet, vagy éppen nem oda figyel a közönség, mint ahova mi szeretnénk, a film mindenképpen veszít az erejéből. Úgy kell a fényeket, a kompozíciót és a mozgásokat megszerkeszteni, hogy mindig a középpontban legyen az, amit mutatni szeretnénk.

Eltúlzás: A hagyományos kézi animációban szinte kivétel nélkül minden esetben minden el van túlozva. A szomorú figura nagyon szomorú, a gyors autó nagyon gyors és a gonosz nagyon gonosz. Ez a megközelítés sokkal érthetőbbé teszi az animációt, és az ábrázolás esetleges hiányosságait bőven kompenzálhatja. Gondoljunk a *Final Fantasy* című 3D animációs film csúfos bukására. Mivel a film alkotói mindent „valószerűre" próbáltak csinálni – és nem éltek a rajzfilmes túlzásokkal –, a „szereplők" jelleme, érzelmi világa meglehetősen erőtlen lett.

Ha az animációs technikákat próbálnánk meg összefoglalni, a következő válfajokat különíthetjük el [97]:

- kulcs-animáció,
- programvezérelt animáció,
- összetett animáció,
- motion capture.

A leggyakrabban használt technika a „*keyframe*" (kulcs) animáció. Ekkor a mozgást kulcspozíciók megadásával határozzuk meg. Ezen pozíciók között a program számítja ki, vagyis interpolálja az animációs görbét. A felhasználónak mindig lehetősége van az interpoláció paraméterezésére, így lehetőségünk van a mozgás ritmusát, dinamikáját és puhaságát befolyásolni.

A programvezérelt animáció már bonyolultabb technika. Bizonyos mozgásokat, színváltozásokat, esetleg alakváltozásokat célszerű automatikusan vezérelni. Ezt általában az általunk használt program script-nyelvével tehetjük meg. Hosszabb-rövidebb programokat írhatunk, melyek képesek objektumok között komplex összefüggéseket létesíteni. Nem okoz például komoly gondot egy kézzel animált autó kerekeinek automatikus forgatása, csupán koordináta-geometriai ismereteinket kell felfrissíteni.

2.8.1. Az összetett animáció

A megmozgatni (animálni) kívánt modellhez a topológiája alapján egy csont/ízület-rendszert rendelnek – az úgynevezett „rigging" eljárás során –, a virtuális marionett különböző irányítókat kap, azt az animátor így manipulálni tudja.

A *Toy Story* „Woody" nevű szereplőjéhez például 700 specializált animációs irányítót használtak.

A számítógépes karakteranimáció leginkább egy merev darabokból álló bábu animálására hasonlít.

Csontokat és őket összekötő ízületeket hozunk létre, amelyek a valódi csontvázhoz hasonlóan mozgatják a felületeket.

A „digitális világban” két lehetőség van a csontozatok, azaz a figurák mozgatására:

– „forward” (direkt) kinematika,
– inverz kinematika.

2.8.2. „Forward” kinematika

Lehetőségünk van a figurák „hagyományos” mozgatására.

Először a karok, lábak felső csuklóit állítjuk be, és egyesével haladunk a csontrendszer utolsó csuklói, az ujjak felé.

A gyakorlott, hagyományos technikákon nevelkedett animátorok szinte kizárólag ezt a megközelítést alkalmazzák, hiszen így a mozdulatok legapróbb részleteit is kezükben tarthatják.

2.8.3. Inverz kinematika

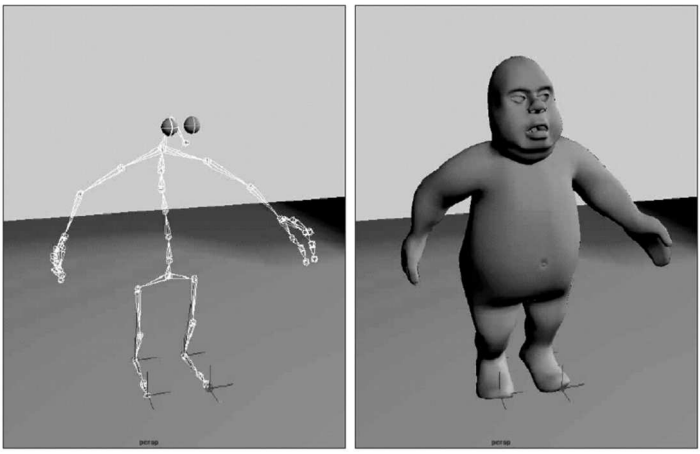
Ez a technika sokkal kényelmesebb és hatékonyabb, mint az előbb említett.

Az animátornak nincsen más feladata, mint a csontrendszer utolsó csontját mozgatni, a többiről a számítógép gondoskodik.

A gyakorlatban ez az jelenti, hogy elég a figurának a kézfejét mozgatni, a könyök és a váll mozgását a program kikövetkezteti.

Komplex mozgásokat szinte csak ezzel a technikával lehet elkészíteni.

Egy sétáló figura animálásához elegendő a láb- és a kézfejeket mozgatni.



2.55. ábra. *Csont/ízületrendszer*

2.8.4. A „skinning”

A csontváz és a figura teste közötti összeköttetést nekünk kell meghatároznunk.

Mivel a valóságban a bőr, azaz a figura felszíne nem közvetlenül a csont mozgását követi (gondoljunk a bonyolult ín- és izomrendszerre), nekünk kell a csontokhoz a felszín egyes darabjait hozzárendelni. Ez a hozzárendelés az ún. *skinning*.

A munkának ez a fázisa gyakran nehezebb, mint maga az animálás. A rossz skinning eredménye az, hogy a hajlatoknál „gyűrődések” jönnek létre, azaz pl. a felkar mozgásánál a mellkas egy darabja is elmozdul.

A programok különböző technikákat ajánlanak fel, de a legáltalánosabb megközelítés egyértelműen a súlyok festése.

Ez azt jelenti, hogy az egyes csontokat kiválasztva egy ecsettel adjuk (festjük) meg azt, hogy a felületre mennyire erősen hasson a csont elmozdulása.

2.8.5. Más technikák

A csontokkal nyilván nem tudunk minden szükséges mozgást létrehozni. Az izmok feszülését, az arc grimaszait más technikákkal kell megoldanunk.

Ez a deformáció a *morphing* vagy *blending*.

Ennek lényege, hogy a modellező eszközökkel különböző formákat alakítunk ki ugyanabból a testből, és ezeket „úsztatjuk” egymásba.

A grimaszok vagy beszéd elkészítése során az összes karakterisztikus szájtartást (betűk formálását) megmodellezzük, majd a hangsáv alapján ezeket „aktivizáljuk”.

2.8.6. A „motion capture”

A motion capture technológia lényege az, hogy a színészek testére fényvisszaverő pontokat (vagy szenzorokat) helyeznek, melyeket több kamera lekövet. A programok ezen pontok alapján milliméterpontosan rekonstruálják a valódi mozgást.

A *King-Kong* 2005-ös remake-jénél Andy Serkis színész segítette a szakembereket a gorilla mozgásának hajszálpontos, precíz lokalizálására a testére helyezett speciális szenzorokkal, amelyek arckifejezéseit is rögzítették, hogy életszerű mozgást kölcsönözzenek a teremtménynek.

2.8.7. Animációs sablonok

A számítógépes programkódok újrahasznosításának elve már rég megjelent az animációkban is. Számos animátor, ha már tökéletesen elkészített egy jelenetet, leírta a karakter mozgását, megszerkesztette a hátteret, felhasználta

2.56. ábra. *Motion caption szenzor*

ezeket egy későbbi jelenet megtervezésekor is. Így könnyen és egyszerűen lehetett hasonló jeleneteket kivitelezni. A számítógépes animálást könnyen meg lehet valósítani megfelelő paraméterezéssel, kód-újrafelhasználással vagy a már elkészített részletek, objektumok többszöri felhasználásával [43].

2.8.8. Animációs szoftverek

A felhasználói szoftverekhez hasonlóan a grafikus, animációra is képes szoftverek is igen szép számban jelentek meg a piacon az idők során. A teljesség igénye nélkül kiragadunk egy pár kiemelkedőbb megoldást.

1985-től kezdődően fejlesztette ki a Pixar a saját számítógépes 3D grafika és animációs szoftverét, a *RenderMan*t. A szoftver nemcsak rajzfilmek gyártását teszi lehetővé, hanem bármilyen vizuális effektus elkészítését filmekben is. Az utóbbi 15 évben a vizuális effektusokért járó Oscar-díjra benevezett 50 film közül 47-et készítettek *RenderMan*nel.

Otthoni számítógépekre – az akkori DOS-os rendszerekre – készült el 1988-ban a *Cartooners*, 1989-ben pedig az *Autodesk Animator*. Mindkettőben jeleneteket, háttereket, karaktereket lehetett definiálni és mozgásokat rendelni a figurákhoz.

Az Autodesk fejlesztette ki a *3D Studio Max*ot is. A *3D Studio Max* (melyet néha *3DS* Maxnek hívnak) egy 3 dimenziós modellező és animációs program, talán a legelterjedtebbek egyike.

Jól használható szoftver a *3D Animation Lab* is.

Kétségtelen viszont, hogy az egyik legjobb a *Maya*. A *Maya* egy felsőkategóriás 3D-s grafikai szoftvercsomag az Aliastól (jelenleg az Autodesk Media & Entertainment tulajdonában van), amelyet főként a filmes és televíziós iparban használnak, valamint számítógépes és videojátékok készítésénél. Az Autodesk 2005 októberében tett szert az *Alias PowerAnimator*ból kifejlődött programra, megvásárolva az Alias Systems Corporationt. Két fő változatban kapható, az

2.57. ábra. *Animációs sablonok A dzsungel könyve (1967) és a Micimackó (1977) rajzfilmekben*

egyik a *Maya Complete* (amely a kisebb csomag), a másik pedig a *Maya Unlimited*. A *Maya Personal Learning Edition (PLE)* egy otthoni használatra szánt tanulóverzió, amely ingyenesen elérhető (cserében a *Maya PLE*-vel renderelt képekben egy vízjel van).

A Maya elérhető Windows, Linux, IRIX és Mac OS X operációs rendszerek alá.

2.8.9. Állományformátumok

Az animációkat többféleképpen rögzíthetjük. Leggyakoribb állományformátumok a FLI, FLC, MPEG, WMV, MOV és AVI.

A legegyszerűbb állományformátum az animált GIF. Az állóképek (GIF87) tárolása mellett a GIF alkalmas képek animálására (GIF89a) is. Weblapokon sokszor találkozhatunk ilyennel. Lényege, hogy megadott időpontokban váltakoznak a különböző képek, így egy animációt hozva létre. Hátránya, hogy nincs hangja.

Általában mindegyik formátum valamilyen módon tömörített, hiszen hosszabb filmek tárolása ily módon a leggazdaságosabb.

2.58. ábra. *Animációs sablonok a Sword in the stone (1963) és A dzsungel könyve (1967) rajzfilmekben*

A FLI és FLC oly módon tömörít, hogy csak azokat a képrészeket tárolja, amelyek különböznek az őket megelőző képkocka ugyanazon helyén lévő adattól.

Az MPEG mágikusan úgy hangzik, mint a JPEG. Nem is csalódhatunk, mert hasonló, némi minőségvesztéssel járó tömörítést valósítanak meg. A minőségromlás mértéke szabályozható a végtermék fájlhosszának a rovására. A gyorsan változó képkockákból álló animáción viszont fel sem tűnik a minőségvesztő tömörítő algoritmus „keze nyoma".

A MOV állományformátumot az Apple alkalmazza.

Az AVI (*Audio Video Interleave* – audio-video-összefésülés) egy olyan állományformátum, amelyet mind a hang-, mind pedig a videoadatok egy meghatározott csomagban való tárolására és ezen adatok lejátszására hoztak létre. A Microsoft 1992 novemberében mutatta be ezt a formátumot a Windows technológia videó részeként.

3. FEJEZET

GRAFIKA DOS ALATT

A DOS operációs rendszer a személyi számítógépek szöveges üzemmódú parancssoros operációs rendszere.

Grafikus alkalmazásokat is lehetett DOS alatt készíteni, ha a számítógépben videokártya ezt megengedte. Az 1980-as évek végén, az 1990-es évek elején, közepén a grafikus üzemmódot egyértelműen meghatározta a videokártya memóriájának a nagysága, a videokártyák nem rendelkeztek külön GPU-val, minden műveletet a CPU végzett el.

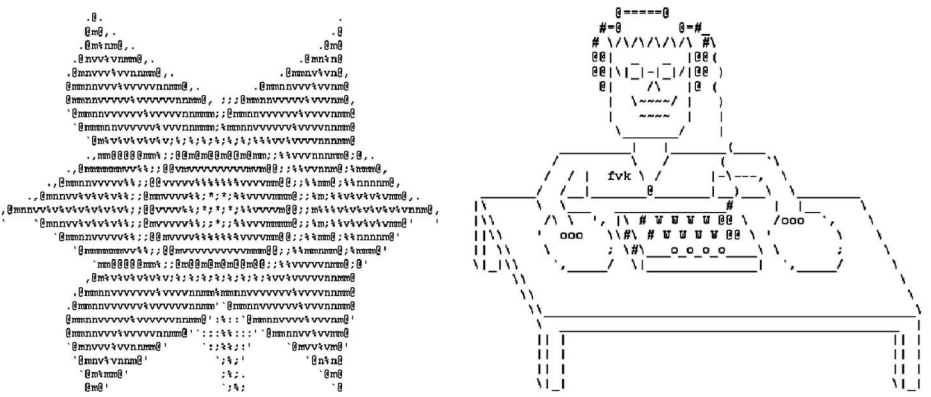
A számítógépünkhöz többféle grafikus kártyát csatolhatunk. Ennek megfelelően a grafikus üzemmódban a felbontás és színeinek száma (ezek a grafikus kártya legjellemzőbb adatai) eltérőek voltak.

A legismertebb grafikus kártyák:

- CGA (Color Graphics Adapter) 320×200 képpont, 16 szín; 640×200 képpont, 2 szín (monochrome);
- MCGA (Multi Color Graphics Adapter) 320×200 képpont, 256 szín;
- TANDY 320×200 képpont, 16 szín;
- HERCULES (vagy röviden HERC) 720×348 képpont, monochrome (2 szín);
- EGA (Enached Graphics Adapter) 640×350 képpont, 16/64 szín;
- VGA (Video Graphics Array) 640×480 képpont, 16/64/256 szín; 800×600 képpont, 256 szín (SVGA 512 KB memória); 1024×768 képpont, 256 szín (SVGA 1 MB memória);
- XGA (eXtended Graphics Array) felbontása és színek száma azonos a VGA-val, de sebessége DOS-ban 90%-kal, Windows-ban 50%-kal nagyobb.

A számítógépek hőskorában csak szöveges üzemmód létezett, a programok az eredményeket a szöveges képernyőn vagy a nyomtatón jelenítették meg, a felhasználók vizuális igényeinek megfelelően azonban a programozók itt is megtalálták a módját annak, hogy grafikus ábrákat állítsanak elő. A legegyszerűbb grafikák (de felépítés, generálás szempontjából talán a legbonyolultabbak) a szöveges karakterekből kirakott képek voltak. Ekkor egy egyszerű soronkénti kiírással szöveges üzemmódban karaktereket jelenítettünk meg a képernyőn (vagy a nyomtatón), amelyek távolról figyelve képpé álltak össze.

Manapság ez művészeti irányzattá fejlődött, speciális generáló szoftvereket is írtak, vagy gyűjteményes kiállításokat is szerveznek (pl. http://www.ascii-art.de/, http://chris.com/ascii/).



3.1. ábra. *Virág karakterekből (készítette Susie Oviatt), valamint 3D hatású karakterekből kirakott kép*

Fejlettebb technika volt a számítógép memóriájában lévő karaktertábla átdefiniálása, számos DOS alatti számítógépes játék készült így. A karakterek is pontokból vannak definiálva, minden karaktert egy „pontmátrix” ír le, a mátrix sorait byte típusú számokká konvertálva. Nem kellett mást tenni, mint átrajzolni a karaktereket, például az ’A’ karakter képe helyett egy olyan számsorozatot beírni, amely egy téglafal pontmátrix-képének felel meg, így valahányszor kiírtuk az ’A’ betűt a képernyőre, nem az ’A’ karakter képe (*glyph*) jelent meg, hanem a téglafal.

3.2. ábra. *Glyph-ek átdefiniálása DOS alatt*

Egy másik közkedvelt DOS-os grafikus megoldás a BOB-ok programozása volt. A *BOB*-ok (*Blitter OBject*) olyan 256 színű, téglalap alakú grafikai objektumok, melyek tetszőlegesen mozgathatóak, eltüntethetők és megjeleníthetőek.

Az elnevezés az Amiga gépekről származik. BOB-nak tekinthető például az egér ikonja grafikus képernyőn, ami általában nyíl formájú. Vagy például BOB egy játékban egy futó ember. Egy BOB bizonyos részein áttetsző (transzparens) lehet, vagyis ott az látszik, ami mögötte van. A BOB memóriában lévő grafikus adatait *shape*-nek nevezzük. Egy W szélességű, H magasságú shape helyfoglalása $W \times H$ byte.

A BOB-okat általában objektumorientáltan szokás programozni magas szintű nyelvekben, de a gyorsaság miatt igen elterjedt az assemblyben való programozás is.

Assemblyben a grafikus üzemmódot a 10h megszakításon (*video and scree services*) keresztül lehet elérni.

A videomód beállítása a 00h funkcióval történik.

Például az MCGA (320×200, 256 színű) üzemmódot, amely az egyetlen hagyományos 256 színű üzemmód, így kell bekapcsolni:

```
1 mov ax, 0013h  {00h funkció, 13h üzemmód}
2 int 10h        {meghívjuk a megszakítást}
```

Visszatérni szöveges üzemmódba:

```
1 mov ax, 0003h  {00h funkció, 03h szöveges üzemmód}
2 int 10h        {meghívjuk a megszakítást}
```

A képernyő memóriabeli kezdőcíme az A000:0000, egy byte egy pixelnek felel meg, a pixelek sorfolytonosan vannak tárolva, balról jobbra, fentről lefele. Az *Ox* koordinátatengely tehát balról jobbra nő, az *Oy* koordinátatengely pedig fentről lefele. A képernyő helyigénye 320×200, azaz 64000 byte.

Az MCGA képszerkezetének megfelelően, egy (x, y) koordinátájú pixel ofszettcímét a videomemóriában a CÍM = 320*$y + x$ képlettel határozhatjuk meg.

Egy C színű, (x, y) koordinátájú pixelt tehát így rajzolhatunk ki:

```
1 mov es, A000h
2 mov ax, 320
3 mul y             {A pixel sorának kezdőcíme 320*y}
4 add ax, x         {A pixel címe: 320*y+x}
5 mov di, ax        {ES:[DI] a felgyújtandó pixel címe}
6 mov al, c         {a pixel színe}
7 mov es:[di], al   {a pixel kigyújtása}
```

Egy (x, y) koordinátájú pixel színének a lekérdezése:

```
1 mov es, A000h
2 mov ax, 320       {A kép szélessége 320 pixel}
3 mul y             {A pixel sorának kezdőcíme: 320*y}
4 add ax, x         {A pixel címe: 320*y+x}
```

```
5 mov di, ax        {ES:[DI] a leolvasandó pixel címe}
6 mov al, es:[di]   {a szín az AL-ben}
```

Közkedvelt, széles körben elterjedt, grafikai megjelenítésre képes magas szintű programozási nyelv a *LOGO*.

A programozási nyelvet és a hozzá kapcsolódó pedagógiai elveket Seymour Papert amerikai matematikus dolgozta ki az 1960-as években. *LOGO* az interpreter (értelmező) nyelvek közé tartozik, azaz közvetlenül lehet utasítást adni és végrehajtatni. Grafikai része alapján az automata elvű nyelvek közé, szövegkezelő része alapján pedig a funkcionális nyelvek családjába sorolható.

LOGO-ban nincs hagyományos értelemben vett változó, mert a változók száma, típusa, elnevezése rögzített, illetve utasítások paraméterei lehetnek. A paraméterek érték szerinti paraméterek, híváskor kapnak értéket. Eljárások, ciklusok paraméterfüggőek lehetnek. A paraméterfüggő ciklus csak egy primitív, ciklusváltozó nélküli, adott lépésszámú lehet. Minden más feladatra rekurzív eljárást célszerű írni.

A *LOGO* grafikus rendszere a *teknőc-grafika*. Eredeti állapotában a képernyő közepén lévő teknőc jellemzője a helye és iránya. Megtanítható tetszés szerint alakzatok rajzolására, mozgó ábrák készítésére, a toll és a rajzlap színének változtatására, matematikai műveletekre, véletlen jelenségek bemutatására, szövegírásra, zenélésre, animációra. A teknőc által a képernyőn megtett út a grafikus ábra. A nyelv utasításai a teknőc vezérlésére szolgálnak.

Példa: egy 100 egység oldalú négyzet kirajzolása:

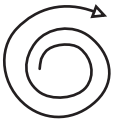
```
1 forward 100
2 right 90
3 forward 100
4 right 90
5 forward 100
6 right 90
7 forward 100
```

A 3.3. ábrán látható spirál kirajzolása rekurzív módon:

```
1 to spiral :size
2    if  :size > 30 [stop]
3    forward :size right 15
4    spiral :size *1.02
5 end
```

Meghívás például:

```
1 spiral 10
```



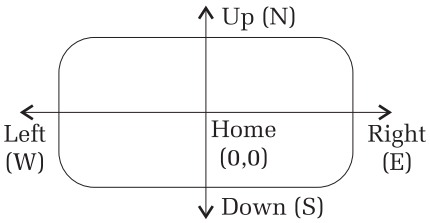
3.3. ábra. *Spirál* LOGO*-ban*

DOS alatt számos – ma is használatos – magas szintű programozási nyelv biztosított grafikus lehetőségeket.

A következőkben a *Turbo (Borland) Pascal* grafikus lehetőségeit tekintjük át, ezek – mivel a fordítóprogramot és a környezetet ugyanaz a cég írta (Borland) – ugyanúgy működnek DOS alatti *Borland C++*-ban is – természetesen az eljárásokat, utasításokat a $C++$ szintaxisának megfelelően kell írni.

3.1. Graph3 – A Borland teknőc-grafikája

A Graph3 a *Turbo Pascal* 3.0-val való kompatibilitást biztosítja és a *TURTLE* grafikus rendszert implementálja. A *TURTLE* grafikus rendszer a *LOGO* programozási nyelvből jól ismert „teknősbéka" által megtett út szerint rajzol. Parancsai előre, hátra, jobbra, balra való mozgatást, valamint forgatásokat tartalmaznak. A koordináták a képernyő középpontjához relatívak.



3.4. ábra. *A Graph3 koordináta-rendszere*

3.1.1. Függvények, eljárások

3.1.1.1. Grafikus mód inicializáló

`procedure` **`GraphColorMode;`**

Beállítja a 320×200-as színes grafikus üzemmódot és elvégzi a grafikus egység működéséhez szükséges memóriafoglalásokat.

`procedure` **`GraphMode;`**

Beállítja a 320×200-as fehér-fekete grafikus üzemmódot.

`procedure` **`HiRes`**`;`
640×200-as nagyobb felosztású grafikus üzemmódra tér át.

3.1.1.2. Rajzoló

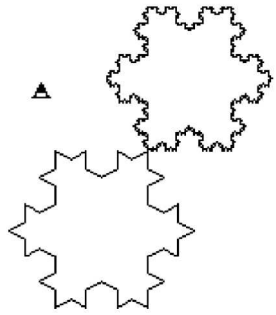
`procedure` **`Arc`**`(X, Y, Angle, Radius, Color: integer);`
Egy `Radius` sugarú, `Angle` szögű, `X`, `Y` középpontú, `Color` színű körívet rajzol.
`procedure` **`Circle`**`(X, Y, Radius, Color: integer);`
Egy `Radius` sugarú, `X`, `Y` középpontú, `Color` színű kört rajzol.
`procedure` **`Draw`**`(X1, Y1, X2, Y2, Color: integer);`
Az (`X1, Y1`) koordinátájú pontot összeköti az (`X2, Y2`) koordinátájú ponttal egy `Color` színű szakasszal.

3.1.1.3. A TURTLE grafikus rendszer

`procedure` **`Back`**`(Dist: Integer);`
Visszalépteti a teknősbékát `Dist` távolságnyira.
`procedure` **`ClearScreen`**`;`
Letörli az aktív ablakot és a teknősbékát kezdeti (`Home`) állásba viszi.
`procedure` **`Forwd`**`(Dist: Integer);`
Előre lépteti a teknősbékát `Dist` lépéssel.
`function` **`Heading`**`: Integer;`
Megadja az aktuális békairányt.
`procedure` **`HideTurtle`**`;`
Elrejti a teknőst.
`procedure` **`Home`**`;`
Kezdeti pozícióba (`Home`) helyezi a békát.
`procedure` **`NoWrap`**`;`
Letiltja a kilépéses rajzolást. Ha az aktív ablakon átlépett, akkor a másik felén alul jelenik meg.
`procedure` **`PenDown`**`;`
„Leengedi a tollat”, vagyis a béka bármely mozdulata rajzolással jár.
`procedure` **`PenUp`**`;`
„Felemeli a tollat”, vagyis léptetéskor nem rajzol.
`procedure` **`SetHeading`**`(Angle: Integer);`
A megadott `Angle` irányba állítja a békát.
`procedure` **`SetPenColor`**`(Color: Integer);`
A toll színét állítja be.
`procedure` **`SetPosition`**`(X, Y: Integer);`
A teknőst a megadott `X`, `Y` helyre mozdítja, rajzolás nélkül.
`procedure` **`ShowTurtle`**`;`
Láthatóvá teszi a teknőst.

`procedure` **`TurnLeft`**`(Angle: Integer);`
`Angle` szögben balra fordítja a teknőst.
`procedure` **`TurnRight`**`(Angle: Integer);`
`Angle` szögben jobbra fordítja a teknőst.
`procedure` **`TurtleDelay`**`(Delay: integer);`
A teknős lépései között `Delay`-nyi idő telik el.
`procedure` **`TurtleWindow`**`(X, Y, W, H: Integer);`
Egy grafikus ablakot definiál az (`X`, `Y`) pontból `H` magasságra és `W` szélességre.
`function` **`TurtleThere`**`: Boolean;`
Teszteli, hogy a béka látható-e az aktív ablakban.
`procedure` **`Wrap`**`;`
A béka kiléphet az aktív ablak kereteiből, a rajzolás folytatódik.
`function` **`Xcor`**`: Integer;`
A béka aktuális `X` koordinátáját adja meg.
`function` **`Ycor`**`: Integer;`
A béka aktuális `Y` koordinátáját adja meg.

Példaprogram. Rajzoljuk ki teknőc-grafikával a Koch-pelyhet!



3.5. ábra. *A Koch-pehely*

```
1  program Turtle;
2  uses Graph3;
3
4  procedure Lep(l, n: integer); forward;
5
6  procedure Pehely(l, n: integer);
7  var i: integer;
8  begin
9    for i := 1 to 3 do
10     begin
11       Lep(l, n);
```

```
12        TurnRight(120);
13      end;
14  end;
15
16  procedure Lep;
17  begin
18    if n = 0 then Forwd(l)
19     else
20       begin
21            Lep(l div 3, n-1);
22            TurnLeft(60);
23            Lep(l div 3, n-1);
24            TurnRight(120);
25            Lep(l div 3, n-1);
26            TurnLeft(60);
27            Lep(l div 3, n-1);
28       end;
29  end;
30
31  var i: integer;
32  begin
33    HiRes;
34    Pehely(-90, 2);
35    Pehely(160, 4);
36    SetPosition(-60, 30);
37    ShowTurtle;
38    readln;
39  end.
```

3.1.1.4. Színhasználat

`procedure` **`ColorTable`**`(C1, C2, C3, C4: integer);`
Egy színtranszlációs táblázatot hoz létre.

`procedure` **`GraphBackground(`**`Color: integer);`
A háttérszínt állítja be `Color` színűre.

`procedure` **`Palette`**`(N: integer);`
Aktívvá teszi az `N` által azonosított palettát.

`procedure` **`HiResColor`**`(Color: integer);`
`Color` színűre állítja a rajzolást 640×200-as grafikus üzemmódban.

3.1.1.5. Festés, kitöltés

`procedure` **`FillScreen`**`(Color: integer);`
`Color` színnel tölti ki az aktív ablakot.

`procedure` **`FillShape`**`(X, Y, FillCol, BorderCol: integer);`
Bármely satírozás területét az adott `FillCol` színnel tölti ki. A kitöltés az (`X`, `Y`) koordinátájú pontból indul és a `BorderCol` színnel határolt területet fogja be. Hasonlóan működik a `Graph.FloodFill` eljáráshoz.

`procedure` **`FillPattern`**`(X1, Y1, X2, Y2, Color: integer);`
Az `X1`, `Y1`, `X2`, `Y2` pontok által meghatározott négyszöget festi ki `Color` színnel.

`procedure` **`Pattern`**`(var P);`
Egy 8×8-as méretű kitöltőminta mátrixot definiál.

3.1.1.6. Általános

`procedure` **`GraphWindow`**`(X1, Y1, X2, Y2: integer);`
Az `X1`, `Y1`, `X2`, `Y2` koordinátájú pontok által meghatározott grafikus ablakot definiálja.

`procedure` **`Plot`**`(X, Y, Color: Integer);`
Az `X, Y` koordinátájú pontba kitesz egy `Color` színű pixelt.

`procedure` **`GetPic`**`(var Buffer; X1, Y1, X2, Y2: integer);`
Az `X1`, `Y1`, `X2`, `Y2` koordinátájú pontok által meghatározott területet (téglalapot) a `Buffer` változóba tárolja. Ezt később a `PutPic` eljárással lehet visszaállítani.

`procedure` **`PutPic`**`(var Buffer; X, Y: integer);`
A `Buffer` változóban tárolt grafikus területet kiteszi az `X`, `Y` ponttól kezdődően.

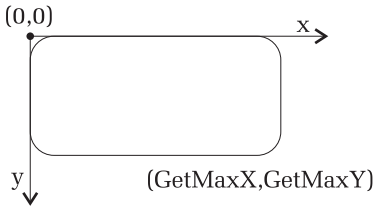
`function` **`GetDotColor`**`(X, Y: Integer): integer;`
Az `X`, `Y` koordinátájú pont színével tér vissza.

3.1.2. Konstansok

Név	Érték	Jelentés
North	0	Északi fordulatszög
East	90	Keleti fordulatszög
South	180	Déli fordulatszög
West	270	Nyugati fordulatszög

3.2. Graph – A Borland grafikus rendszere

A *Graph* unit közel 80 rutint tartalmazó grafikus gyűjtemény, amely a bitműveletektől a magas szintű funkciókig mindenféle rutint tartalmaz.



3.6. ábra. *A BGI grafika koordináta-rendszere*

Hogy egy Graph unitot használó programot futtathassunk, szükségünk van egy vagy több grafikus meghajtóra (**.BGI** állományok *Borland Graphic Interface*) az **.EXE** programon kívül. Ha a programunk fontokat is használ, akkor szükségünk van még font (**.CHR**) állományokra is. Ezeket az állományokat a telepítőprogram a megfelelő (rendszerint ...\bp\bgi) alkönyvtárba helyezi el. A **.BGI** állományokat be lehet fordítani az **.EXE** állományba. Erre a célra a **BINOBJ** nevű programot kell felhasználni. Ennek a segítségével a .**BGI** állományt **.OBJ** állománnyá alakíthatjuk át, majd ezt a ***{$L név}*** direktívával az **.EXE** állományba fordíthatjuk.

Példaként álljon itt az **EGAVGA.BGI** grafikus meghajtó befordítása. Először a **DOS** promptnál elindítjuk a **BINOBJ** programot a kívánt paraméterekkel: a meghajtó-állomány neve, az **OBJ** állomány neve és a **public**-ként deklarált főeljárás neve:

```
c:\>BINOBJ EGAVGA.BGI VGADRV.OBJ VGADriver
```

Majd megírjuk a megfelelő *Pascal* programot:

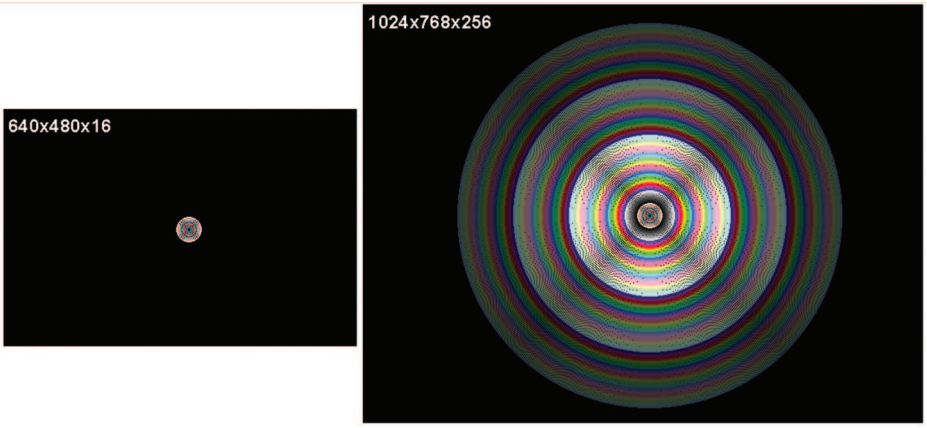
```
1  program VGAMode;
2
3  uses Graph;
4
5  procedure VGADriver; external;
6  {$L VGADRV.OBJ}
7
8  procedure InitVGA(Mode: integer);
9  var gd: integer;
10 begin
11    gd := RegisterBGIDriver(@VGADriver);
12    if gd < 0 then
13     begin
14       writeln(GraphErrorMsg(GraphResult));
```

```
15        Halt(1);
16      end;
17    gd := VGA;
18    InitGraph(gd, Mode, '');
19    gd := GraphResult;
20    if gd <> GrOK then
21     begin
22       writeln(GraphErrorMsg(gd));
23       Halt(1);
24      end;
25  end;
26
27  begin
28    InitVGA(VGAHi);
29    Line(0, 0, GetMaxX, GetMaxY);
30    readln;
31    CloseGraph;
32  end.
```

Az alábbi *Pascal* program hagyományosan (a **.BGI** grafikus meghajtó belefordítása nélkül) inicializálja a grafikus üzemmódot, és különböző színű koncentrikus köröket rajzol ki:

```
1   program EGAVGA;
2
3   uses Graph;
4
5   var
6     GraphMode, GraphDriver, GrErr: integer;
7     i: byte;
8   begin
9     GraphDriver := Detect;
10    InitGraph(GraphDriver, GraphMode, '');
11    GrErr := GraphResult;
12    if GrErr <> grOK then
13     begin
14       writeln('Graphics error: ',
15                GraphErrorMsg(GrErr));
16       readln;
17       Halt(1);
18    end;
19    for i := 0 to GetMaxColor do
20      begin
```

```
21        SetColor(i);
22        Circle(GetMaxX div 2, GetMaxY div 2, i+1);
23      end;
24    readln;
25    CloseGraph;
26  end.
```



3.7. ábra. *640×480-as, 16 színű EGAVGA, illetve 1024×768-as felbontású, 256 színű BGI grafikus üzemmódok*

Mivel az **SVGA256.BGI** nem szabványos grafikus meghajtó (nem a Borland írta, hanem letölthető pl. a *http://pascal.sources.ru/graph/svga256t.htm* honlapról), ezt így kell inicializálni és használni pl. 256 színű koncentrikus körök kirajzolására:

```
1   program SVGA256;
2
3   uses Graph;
4
5   {$F+}
6   function DetectSVGA256: integer;
7   begin
8     { 0: 320x200x256
9       1: 640x400x256
10      2: 640x480x256
11      3: 800x600x256
12      4: 1024x768x256 }
13    DetectSVGA256 := 4;
14  end;
```

```
15 {$F-}
16
17 var
18   GraphMode, GraphDriver, GrErr: integer;
19   i: byte;
20 begin
21   GraphDriver := InstallUserDriver('SVGA256',
22                                    @DetectSVGA256);
23   GraphDriver := Detect;
24   InitGraph(GraphDriver, GraphMode, '');
25   GrErr := GraphResult;
26   if GrErr <> grOK then
27    begin
28      writeln('Graphics error: ',
29              GraphErrorMsg(GrErr));
30      readln;
31      Halt(1);
32   end;
33   for i := 0 to GetMaxColor do
34     begin
35       SetColor(i);
36       Circle(GetMaxX div 2, GetMaxY div 2, i+1);
37     end;
38   readln;
39   CloseGraph;
40 end.
```

3.2.1. Függvények, eljárások

3.2.1.1. Grafikus módot inicializáló eljárások

`procedure DetectGraph(var GraphDriver, GraphMode: integer);`
Megvizsgálja a hardvert, és megállapítja a grafikus kiépítettségét. A `GraphDriver`-ben kapjuk meg a grafikus meghajtó számát, a `GraphMode`-ban pedig a használható legnagyobb felbontású üzemmód számát.

`procedure InitGraph(var GraphDriver, GraphMode: integer;`
`PathToDriver: string);`
Inicializálja a grafikus rendszert és átállítja a hardvert (képernyőt) grafikus módra. A `PathToDrive` a **.BGI** állomány elérési útvonalát jelenti.

`function GetDriverName: string;`
Megadja az aktuális grafikus képernyőmeghajtó nevét (pl. EGAVGA).

`function` **`GetGraphMode`**`: integer;`
Visszaadja az aktuális grafikusmód kódját.

`function` **`GetModeName`**`(ModeNumber: Integer): string;`
A függvény értéke a bemeneti paraméterhez mint kódhoz tartozó grafikus mód teljes neve.

`function` **`GetMaxMode`**`: integer;`
Az aktuálisan betöltött grafikus meghajtó legnagyobb felbontású üzemmódjának számát adja vissza.

`procedure` **`GetModeRange`**`(GraphDriver: integer; var LoMode, HiMode: integer);`
A megadott grafikus meghajtó kódjához (`GraphDriver`) tartozó grafikus üzemmódok közül a legkisebb és legnagyobb értékűt adja vissza.

`procedure` **`GraphDefaults`**`;`
A grafikus kurzort a bal felső sarokba teszi és alaphelyzetbe állítja a grafikus rendszert.

`function` **`GetMaxX`**`: integer;`
A grafikus képernyő utolsó oszlopának számát adja vissza.

`function` **`GetMaxY`**`: integer;`
Megadja a grafikus képernyő utolsó sorának számát.

`function` **`GetX`**`: integer;`
Az aktuális képpont (CP – *Current Position*) vízszintes koordinátáját adja vissza.

`function` **`GetY`**`: integer;`
Az aktuális képpont (CP) függőleges koordinátáját adja vissza.

`procedure` **`SetGraphBufSize`**`(BufSize: word);`
Megváltoztatja az alapértelmezett grafikus buffer méretét. A belső buffer mérete `BufSize`-nak definiálódik a heap-en az `InitGraph` eljárás hívásakor. Alapértelmezés: 4 KB.

`procedure` **`CloseGraph`**`;`
Lezárja a grafikusmódot, visszatér a szöveges képernyőmódhoz.

`procedure` **`RestoreCrtMode`**`;`
Visszaállítja a grafikus rendszer installálása előtt használt képernyőmódot.

`procedure` **`SetGraphMode`**`(Mode: integer);`
Grafikus módba állítja a rendszert és letörli a képernyőt.

`function` **`InstallUserDriver`**`(Name: string; AutoDetectPtr: pointer): integer;`
A *Pascal* grafikus rendszerében előre nem telepített képernyőtípushoz új grafikus meghajtót telepít. `Name`: az új képernyőmeghajtó (**.BGI**) állomány neve, `AutoDetectPtr`: ha **`nil`**, automatikusan vizsgálja a hardvert; ha saját függvényt írunk rá, a címét kell itt megadnunk (pl. `@Vizsgal`)

`function` **`InstallUserFont`**`(FontFileName: string): integer;`
Új fontot telepít. Új fontnak nevezzük azt a fontot, mely még nincs beépítve a BGI rendszerbe. Ezt egy **.CHR** állomány tárolja.

`function` **`RegisterBGIDriver`**`(driver: pointer): integer;`
E függvény segítségével a grafikus rendszerben nem szereplő képernyőtípusokon is dolgozhatunk. A függvény a grafikus rendszer részévé tesz egy **.BGI** driver állományt, amely egy grafikus képernyőt kezel. A paraméterben a heap-ben a meghajtónak lefoglalt terület kezdőcímét kell megadni. Az `InitGraph` használata előtt mindig alkalmazni kell, ha nincs előredefiniált grafikus meghajtó a képernyőhöz. A függvény visszatérési értéke a meghajtó száma lesz.

`function` **`RegisterBGIFont`**`(Font: pointer): integer;`
Olyan betűtípus használatakor alkalmazzuk, amely nem része a grafikus rendszernek. A telepítendő új fontot először töltsük a memóriába (`Font` kezdőcímtől), majd ezzel a paraméterrel hívjuk meg a függvényt.

3.2.1.2. Grafikus hibakezelés

`function` **`GraphResult`**`: integer;`
A függvény értéke a legutóbbi grafikus művelet hibakódja.

`function` **`GraphErrorMsg`**`(ErrorCode: integer): string;`
A függvény értéke a paraméterben megadott kódú grafikus hiba szövegét adja.

3.2.1.3. Ablak- és lapkezelő eljárások

`procedure` **`SetVisualPage`**`(Page: word);`
Beállítja a látható képernyőt (amennyiben több van). Ez nem feltétlenül lesz aktív képernyőlap. Aktívvá a `SetActivePage` eljárással tehető egy képernyőlap.

`procedure` **`SetActivePage`**`(Page: word);`
Az aktív grafikus képernyőlapot állítja be. Ez nem feltétlenül lesz látható a képernyőn. Egy képernyőlap a `SetVisualPage` eljárással tehető láthatóvá.

`procedure` **`SetViewPort`**`(x1, y1, x2, y2: integer; Clip: boolean);`
Beállít egy aktuális képernyőablakot a grafikus képernyőn. Az (`x1`, `y1`) és (`x2`, `y2`) definiálják az ablak bal felső és jobb alsó sarkait. A `Clip` a vágás állapotát adja meg. Ha `true`, a kiírás az ablak szélén túl nem folytatódik.

`procedure` **`GetViewSettings`**`(var ViewPort: ViewPortType);`
Visszatérési rekordja a grafikus képernyőn definiált aktuális ablak koordinátáit és vágási paramétereit tartalmazza.

`procedure` **`ClearDevice`**`;`
Letörli az aktív grafikus képernyőt, és alapállapotba (`0`, `0`) helyezi a grafikus kurzort. A képernyő háttérszínű lesz.

`procedure` **`ClearViewPort`**`;`
Letörli az aktuális grafikus ablakot, a grafikus kurzort pedig a (`0`, `0`) helyre teszi. Az ablak háttérszínű lesz.

`procedure` **`GetAspectRatio`**`(var Xasp, Yasp: word);`
A képernyő vízszintes és függőleges képméretarányát, azaz a képszélességet, képfelbontást adja vissza. A képméretarány (`Xasp : Yasp`).
`procedure` **`SetAspectRatio`**`(Xasp, Yasp: word);`
Beállítja az aktuális képméretarányt megadó módosító tényezőt.

3.2.1.4. Grafikus kurzor mozgatás

`procedure` **`MoveTo`**`(X, Y: integer);`
Az aktuális pozíciót az (`X`, `Y`) koordinátájú pontra teszi.
`procedure` **`MoveRel`**`(Dx, Dy: integer);`
Az aktuális pozíciót eredeti helyzetéből relatívan mozgatja a grafikus képernyőn. Ha az aktuális pozíció az (X, Y), az ($(X$ + `Dx`), (Y +`Dy`)) koordinátájú helyre mozgatja.

3.2.1.5. Pontok

`procedure` **`PutPixel`**`(X, Y: integer; Pixel: word);`
Az (`X`, `Y`) koordinátájú képpontot `Pixel` színűre festi.
`function` **`GetPixel`**`(X, Y: integer);`
Az (`X`, `Y`) koordinátájú pont színét adja vissza.

3.2.1.6. Vonalak

`procedure` **`Line`**`(x1, y1, x2, y2: integer);`
Az (`x1`, `y1`) pontból szakaszt húz (`x2`, `y2`)-be.
`procedure` **`LineTo`**`(X, Y: integer);`
Az aktuális pozíciótól (CP) (`X`, `Y`)-ba szakaszt húz.
`procedure` **`LineRel`**`(Dx, Dy: integer);`
Az aktuális CP-től kezdve relatívan rajzol, majd a CP-t az új pozícióra állítja. A vonal a CP-ből (x_0, y_0) megy az (x_1, y_1) pontig, ahol $x_1 = x_0$ + `Dx`, $y_1 = y_0$ + `Dy`.
`procedure` **`SetLineStyle`**`(LineStyle: word; Pattern: word; Thickness: word);`
Beállítja az aktuális vonalvastagságot és színt.
`procedure` **`GetLineSettings`**`(var LineInfo: LineSettingsType);`
Információt ad az aktuális vonalmintáról, stílusról és vonalvastagságról, ahogy azt a `SetLineStyle` definiálta.
`procedure` **`SetWriteMode`**`(WriteMode: integer);`
Az egyenesek rajzolásának módját állítja be. (`XORPUT`, `ANDPUT`, `ORPUT` stb.)

3.2.1.7. Körök, körívek és más görbék

`procedure` **`Circle`**`(X, Y: integer; Radius: word);`
A **SetColor**-ral beállított aktuális színnel egy (X, Y) középpontú, Radius sugarú kört rajzol.

`procedure` **`Arc`**`(X, Y: integer; StAngle, EndAngle, Radius: word);`
Körívet rajzol (nem szükségszerűen kört). A körív az StAngle (kezdő szög)-től az EndAngle-ig tart, Radius sugárral és (X, Y)-t használva középpontul.

`procedure` **`Ellipse`**`(X, Y: integer; StAngle, EndAngle: word; XRadius, YRadius: word);`
Megrajzolja egy ellipszis körvonalát. A körvonalat az StAngle szögtől EndAngle-ig rajzolja XRadius nagytengelyű és YRadius kistengelyű sugárral az (X, Y) középpontból.

`procedure` **`GetArcCoords`**`(var ArcCoords: ArcCoordsType);`
A legutóbbi **Arc** utasítással megrajzolt kör vagy ellipszis középpontját, és az ív kezdő- és végpontját adja vissza.

`procedure` **`PieSlice`**`(X, Y: integer; StAngle, EndAngle, Radius: word);`
Megrajzol és kitölt egy körcikket. Az (X, Y) a középpont. A szelet StAngle szögtől EndAngle-ig tart, Radius sugárral.

`procedure` **`Sector`**`(X, Y: integer; StAngle, EndAngle, XRadius, YRadius: word);`
Megrajzol és kitölt egy ellipsziscikkelyt. A változók jelentését lásd az Ellipse ill. a PieSlice eljárásnál.

`procedure` **`FillEllipse`**`(X, Y: integer; XRadius, YRadius: word);`
Megrajzol egy kitöltött ellipszist. (X, Y) a középpont; XRadius és YRadius a függőleges és vízszintes sugarak.

3.2.1.8. Sokszögek, satírozások

`procedure` **`Rectangle`**`(x1, y1, x2, y2: integer);`
Megrajzolja egy téglalap körvonalát az aktuális színnel és vonalstílussal. A téglalap bal felső és jobb alsó sarkát az (x1, y1) és (x2, y2) koordináták adják meg.

`procedure` **`Bar`**`(x1, y1, x2, y2: integer);`
Téglalapot rajzol az aktuális kitöltési stílussal és színnel. Az (x1, y1) a téglalap bal felső sarkának, az (x2, y2) pedig a jobb alsó sarkának koordinátáit tartalmazza.

`procedure` **`Bar3D`**`(x1, y1, x2, y2: integer; Depth: word; Top: boolean);`
3 dimenziós téglatestet rajzol az aktuális kitöltési stílussal és színnel. A téglatest első lapja olyan, mintha a Bar eljárással az x1, y1, x2, y2 paraméterekkel rajzoltuk volna meg. A Depth paraméterbe a téglatest mélységét kell írni. A Top azt határozza meg, hogy meg kell-e rajzolni a test felső lapját vagy sem.

`procedure` **`DrawPoly`**`(NumPoints: word; var PolyPoints);`
Megrajzolja egy sokszög körvonalát az aktuális színnel és vonalstílussal. A `NumPoints` a csúcsok számát határozza meg, amelyek koordinátái a `PolyPoints`-ban vannak tárolva. Egy koordináta két word-ból áll, egy `X` és egy `Y` értékből.

`procedure` **`FillPoly`**`(NumPoints: word; var PolyPoints);`
Megrajzol egy kitöltött sokszöget. A `NumPoints` a sokszög csúcsainak számát adja meg, a csúcsok koordinátáit pedig a `PolyPoints` paraméterbe kell tenni. Egy csúcs koordinátája két word-ból áll, az egyik az `X`, a másik az `Y`.

`procedure` **`FloodFill`**`(X, Y: integer; Border: word);`
Egy körülhatárolt területet a megadott színnel és mintával tölt ki. Az (`X`, `Y`) koordinátájú pontnak benne kell lennie a területben. A `Border` a színezendő területet határoló szín kódja.

`procedure` **`SetFillStyle`**`(Pattern: word; Color: word);`
Beállítja a kitöltési mintát és színt. A `Pattern` a minta számát, a `Color` pedig a minta színét tartalmazza.

`procedure` **`GetFillSettings`**`(var FillInfo: FillSettingsType);`
Az aktuális színt és töltési mintát adja vissza, amit a `SetFillStyle` vagy `SetFillPattern` állított be legutoljára.

`procedure` **`SetFillPattern`**`(Pattern: FillPatternType; Color: word);`
Egy felhasználó által definiált kitöltési mintát állít be. A `Color` a minta színét adja meg.

`procedure` **`GetFillPattern`**`(var FillPattern: FillPatternType);`
Az aktuálisan kiválasztott tónus és minta értékét adja vissza, amelyeket a `SetFillStyle` vagy a `SetFillPattern` parancsokkal állítottunk be.

3.2.1.9. Képmentés és visszaállítás

`function` **`ImageSize`**`(x1, y1, x2, y2: integer): word;`
A megadott ablak (vagy terület) byte-okban mért memóriabeli helyfoglalását adja vissza. A mérendő téglalap bal felső és jobb alsó sarkát határozzák meg a paraméterek.

`procedure` **`GetImage`**`(x1, y1, x2, y2: integer; var BitMap);`
A kijelölt terület bittérképét bufferbe menti. Az (`x1`, `y1`) és (`x2`, `y2`) adják meg a másolandó terület bal felső és jobb alsó sarkát. A bittérképet a `BitMap` változóba menti.

`procedure` **`PutImage`**`(X, Y: integer; var BitMap; BitBlt: word);`
Egy terület tartalmát visszatölti a képernyőre. A visszatöltés a `BitMap` típus nélküli változóból történik egy olyan téglalapba, amelynek bal felső sarka (`X`, `Y`). A `BitBlt` változóval a visszatöltés módját definiálhatjuk (`XorPut`, `NormalPut`, `AndPut`, `OrPut`, stb.).

3.2.1.10. Szövegkezelés

`procedure` **`SetTextStyle`**`(Font, Direction: word; CharSize: word);`
Grafikus módban beállítja a szöveges kiírás paramétereit (stílusát). A `Font` a betűtípus száma, a `Direction` a kiírás iránya, a `CharSize` pedig a kiírás mérete.

`procedure` **`SetUserCharSize`**`(MultX, DivX, MultY, DivY: Word);`
Megváltoztatja a grafikus betűk szélességét és magasságát. Az aktuális betűtípus szélessége: `MultX / DivX`-szeresre, a magassága pedig `MultY / DivY`-szeresre növekszik.

`procedure` **`SetTextJustify`**`(Horiz, Vert: word);`
Ez az eljárás a következő szövegkiírások pozicionálásának fajtáját állítja be. A `Horiz` a vízszintes, a `Vert` a függőleges igazítás kódját tartalmazzák. Ezek a kódok az igazító konstansok is lehetnek.

`procedure` **`GetTextSettings`**`(var TextInfo: TextSettingsType);`
A `SetTextStyle` és `SetTextJustify` eljárásokkal beállított szövegstílusról adja meg a következő információkat: szövegfont (betűtípus), irány, méret, pozicionálás.

`function` **`TextHeight`**`(TextString: string): word;`
A függvény értéke a paraméterben megadott szöveg képpontokban mért magassága.

`function` **`TextWidth`**`(TextString: string): word;`
A függvény értéke a paraméterben megadott szöveg képpontokban mért szélessége.

`procedure` **`OutText`**`(TextString: string);`
Az aktuális pozícióba szöveget ír ki a grafikus képernyőn.

`procedure` **`OutTextXY`**`(X, Y: integer; TextString: string);`
Kiír egy stringet a grafikus képernyőre az (`X`, `Y`) ponttól kezdve.

3.2.1.11. Színhasználat

`procedure` **`GetDefaultPalette`**`(var Palette: PaletteType);`
Az adott képernyőtípusnak megfelelő alapértelmezett palettát leíró rekordot adja vissza. Ezt a rekordot az `InitGraph` állítja be.

`procedure` **`SetPalette`**`(ColorNum: word; Color: shortint);`
Az aktuális paletta `ColorNum` színét `Color` színűre cseréli.

`procedure` **`SetAllPalette`**`(var Palette);`
Lecseréli az aktív paletta összes színét a megadott paletta színeire. Csak grafikus módban, akkor is csak a következő grafikus meghajtóknál alkalmazható: EGA, EGA64, VGA. Az IBM8514 és VGA256 színes módjában nem használható.

`procedure` **`GetPalette`**`(var Palette: PaletteType);`
Az aktuális palettát és méretét adja vissza.

`function` **`GetPaletteSize`**`: integer;`
A grafikus üzemmód palettájának méretét, azaz a színek számát adja vissza.

`function` **`GetMaxColor`**`: word;`
A **SetColor** eljárással beállítható legmagasabb értékű szín kódját adja meg.

`procedure` **`SetBkColor`**`(Color: word);`
Beállítja az aktuális háttérszínt. Csak az aktuális paletta színeiből választhatunk.

`procedure` **`SetColor`**`(Color: word);`
Beállítja az aktuális rajzoló színt. Csak az aktuális paletta színeiből választhatunk.

`function` **`GetBkColor`**`: word;`
Visszaadja az aktuális háttérszínt.

`function` **`GetColor`**`: word;`
Visszaadja az aktuális színt, melyet a legutóbbi `SetColor`-ral definiáltunk.

3.2.2. Típusok, konstansok, változók

3.1. táblázat. *Típusok*

Név	Deklaráció	Jelentés
ArcCoordsType	`ArcCoordsType = record X, Y, XStart, YStart, XEnd, YEnd: integer; end;`	Rekord a görbék számára
FillPatternType	`FillPatternType = array[1..8] of byte`	Kitöltőminta meghatározására szolgáló vektor
FillSettingsType	`FillSettingsType = record Pattern: word; Color: word; end;`	Kitöltés beállítására szolgál
LineSettingsType	`LineSettingsType = record LineStyle: word; Pattern: word; Thickness: word; end;`	Egyenesek rajzolására szolgáló rekord

PaletteType	`PaletteType = record`
	`Size: byte;`
	`Colors: array[0..Max Colors] of Shortint;`
	`end;`
		A Paletta beállításait tárolja

PointType	`PointType = record`
	`X, Y: integer;`
	`end;`
		Egy pont koordinátái

TextSettingsType	`TextSettingsType = record`
	`Font: word;`
	`Direction: word;`
	`CharSize: word;`
	`Horiz: word;`
	`Vert: word;`
	`end;`
		Egy szöveg kiíratására vonatkozó adatokat tárolja

ViewPortType	`ViewPortType = record`
	`x1, y1, x2, y2: integer;`
	`Clip: boolean;`
	`end;`
		Egy grafikus ablakra vonatkozó adatokat tartalmazza

3.2. táblázat. *Változók*

Név	Típus	Jelentés
GraphGetMemPtr	pointer	A grafikus memóriaterületre mutat
GraphFreeMemPtr	pointer	A szabad grafikus memóriaterület címét tartalmazza

3.3. táblázat. *Konstansok*

Név	Érték	Jelentés
TopOn	true	A Bar3D számára, a felső vonal berajzolása
TopOff	false	A Bar3D számára, a felső vonal elhagyása
		Kitevési konstansok
NormalPut	0	Normális megjelenítés
CopyPut	0	Mozgatás
XOrPut	1	Kizáró VAGY
OrPut	2	VAGY

3.3. táblázat. *Konstansok (folytatás)*

Név	Érték	Jelentés
AndPut	3	ÉS
NotPut	4	Negáció
		Vágási konstansok
ClipOn	true	Az ablak fed
ClipOff	false	Az ablak nem fed
		Sötét háttér- és előtérszínek
Black	0	Fekete
Blue	1	Kék
Green	2	Zöld
Cyan	3	Cián
Red	4	Piros
Magenta	5	Tüdő
Brow	6	Barna
LightGray	7	Világosszürke
		Előtérszínek
DarkGray	8	Sötétszürke
LightBlue	9	Világoskék
LightGreen	10	Világoszöld
LightCyan	11	Világoscián
LightRed	12	Világospiros
LightMagenta	13	Világostüdő
Yellow	14	Sárga
White	15	Fehér
Blink	128	Villogó előtér
		Sötét háttér- és előtérszínek
EGABlack	0	Fekete
EGABlue	1	Kék
EGAGreen	2	Zöld
EGACyan	3	Cián
EGARed	4	Piros
EGAMagenta	5	Tüdő
EGABrow	20	Barna
EGALightGray	7	Világosszürke
		Előtérszínek
EGADarkGray	56	Sötétszürke
EGALightBlue	57	Világoskék
EGALightGreen	58	Világoszöld

3.3. táblázat. *Konstansok (folytatás)*

Név	Érték	Jelentés
EGALightCyan	59	Világoscián
EGALightRed	60	Világospiros
EGALightMagenta	61	Világostüdő
EGAYellow	62	Sárga
EGAWhite	63	Fehér
		Kitöltőminta konstansok
EmptyFill	0	A háttér színe
SolidFill	1	Az előtér színe
LineFill	2	Vonal —— minta
LtSlashFill	3	Ferde ritka / / / / minta
SlashFill	4	Ferde sűrű ////// minta
BkSlashFill	5	Ferde sűrű \\\\\\ minta
LtBkSlahFill	6	Ferde ritka \ \ \ \ minta
HatchFill	7	Kockás minta ###
XHatchFill	8	Dőlt kockás × × ×
InterleaveFill	9	Egyenletes tónus
WideDotFill	10	Egyenletes gyengébb tónus
CloseDotFill	11	Egyenletes közepes tónus
UserFill	12	Felhasználó által definiált minta
		Grafikus meghajtó konstansok
CurrentDrive	−128	Az aktuális meghajtó
Detect	0	Automatikus detektálás
CGA	1	CGA driver
MCGA	2	Monochrom CGA
EGA	3	EGA
EGA64	4	EGA64
EGAMono	5	Monochrom EGA
IBM8514	6	IBM8514
HercMono	7	Monochrom Hercules
ATT400	8	ATT400
VGA	9	VGA driver
PC3270	10	PC3270
		Grafikus üzemmódok
CGAC0	0	CGA 320×200 színes
CGAC1	1	CGA 320×200 színes
CGAC2	2	CGA 320×200 színes
CGAC3	3	CGA 320×200 színes
CGAHi	4	CGA 640×200 színes

3.3. táblázat. *Konstansok (folytatás)*

Név	Érték	Jelentés
MCGAC0	0	CGA 320×200 Mono
MCGAC1	1	CGA 320×200 Mono
MCGAC2	2	CGA 320×200 Mono
MCGAC3	3	CGA 320×200 Mono
MCGAMed	4	CGA 640×200 Mono
MCGAHi	5	CGA 640×480 Mono
EGALo	0	EGA 640×200 színes
EGAHi	1	EGA 640×350 színes
EGAMonoLo	2	EGA 640×20 Mono
EGAMonoHi	3	640×350 Mono
EGA64Lo	0	EGA64 640×200 színes
EGA64Hi	1	EGA64 640×350 színes
ATT400C0	0	ATT400 320×200 színes
ATT400C1	1	ATT400 320×200 színes
ATT400C2	2	ATT400 320×200 színes
ATT400C3	3	ATT400 320×200 színes
ATT400Med	4	ATT400 640×200 színes
ATT400Hi	5	ATT400 640×400 színes
HercMonoHi	0	Hercules 720×348 Mono
IBM8514Lo	0	IBM8514 640×480 színes
IBM8514Hi	1	IBM8514 1024×768 színes
PC3270Hi	0	PC3270 720×350 színes
VGALo	0	VGA 640×200 színes
VGAMed	1	VGA 640×350 színes
VGAHi	2	VGA 640×480 színes
		Szövegirányítási konstansok
LeftText	0	Balra
CenterText	1	Középre
RightText	2	Jobbra
BottomText	0	Le
TopText	2	Fel
		Vonalrajzolási konstansok
SolidLn	0	Folytonos
DottedLn	1	Pontozott
CenterLn	2	Pontozott – szaggatott
DashedLn	3	Szaggatott
UserBitLn	4	Felhasználó által definiált
NormWidth	1	Normális vastagság

3.3. táblázat. *Konstansok (folytatás)*

Név	Érték	Jelentés
ThickWidth	3	Vastagított
		Fontok
DefaultFont	0	8×8-as alapfont
TriplexFont	1	Vonalas font
SmallFont	2	Vonalas font
SansSerifFont	3	Vonalas font
GothicFont	4	Vonalas font
		Irányítás
HorizDir	0	Balról jobbra
VertDir	1	Alulról felfele
UserCharSize	0	Felhasználó által definiált karakterméret
		Grafikus hibák
grOK	0	Nincs hiba
grNoInitGraph	–1	A grafikus rendszer nincs telepítve
grNotDetected	–2	Megvizsgálatlan hardware
grFileNotFound	–3	A **.BGI** állomány nem létezik
grInvalidDriver	–4	Helytelen driver
grNoLoadMem	–5	Kevés a memória
grNoScanMem	–6	Memória vége direkt betöltésnél
grNoFloodMem	–7	Soros töltéskor kevés a memória
grFontNotFound	–8	A **.CHR** állomány hiányzik
grNoFontMem	–9	Kevés a memória fontbetöltéshez
grInvalidMode	–10	Helytelen grafikus mód
grError	–11	Grafikus hiba
grIOError	–12	Grafikus I/O hiba
grInvalidFont	–13	Helytelen font állomány
grInvalidontNum	–14	Helytelen a betűtípus száma

4. FEJEZET

GRAFIKA WINDOWS ALATT

4.1. A GDI grafika

A Windows grafikus felülettel rendelkező multitaszking, többfelhasználós operációs rendszer. Szerkezetét tekintve három fontos függvénykönyvtárra épül: Kernel32.dll főleg a memória menedzselési funkciókat tartalmazza, az operációs rendszer magvát képezi; User32.dll a felhasználói felületet kezelését biztosítja; Gdi32.dll a rajzolási rutinokat és az ezekkel kapcsolatos funkciókat tartalmazza.

A Windows operációs rendszer grafikus alrendszerének magját a *GDI (Graphics Device Interface)*, azaz a grafikus eszközcsatoló adja. A GDI valójában nem más, mint egy absztrakt, az alkalmazások és a megjelenítő eszközök (képernyő, nyomtató stb.) meghajtóprogramjai közötti kapcsolatot biztosító illesztőfelület. Feladata az alkalmazások által az eszközfüggetlen rutinkészlet felhasználásával kezdeményezett rajzolási műveletek eszközfüggő hívásokká történő átalakítása, azaz a grafikus kimenet generálása a mindenkori megjelenítő/leképező eszközön [41].

A Windows grafikus alrendszere a GDI (*Graphics Device Interface*). A GDI eszközvezérlő programokon keresztül kezeli a grafikus perifériákat, és ezáltal lehetővé teszi, hogy a rajzgépet, a nyomtatót, a képernyőt egységesen használjuk. A GDI programozásakor bármilyen hard eszközt, meghajtót figyelmen kívül hagyhatunk. A színek használata is úgy van megoldva, hogy nem kell foglalkoznunk a konkrét fizikai keveréssel és kialakítással. Ezáltal a pixel-adatokat is eszközfüggetlenül használhatjuk. Hasonlóképpen van megoldva a karakterek, fontok eszközfüggetlen megjelenítése is. A *TrueType* fontok használata biztosítja azt, hogy a megtervezett szöveg nyomtatásban is ugyanolyan lesz, mint ahogy azt a képernyőn láttuk. A GDI nagy előnye az is, hogy saját koordinátarendszerrel dolgozhatunk, virtuális távolságokkal írhatjuk meg, a konkrét hardvertől függetlenül, az alkalmazásunkat. Mindezen előnyök mellett azonban a GDI továbbra is kétdimenziós, egészkoordinátájú grafikus rendszer maradt. A GDI nem támogatja az animációt.

A GDI filozófiának az alapja az, hogy először meghatározunk egy *eszközleírót* (*eszközkörnyezet*, *device context*, *DC*), amely a fizikai eszközzel való kapcsolatot rögzíti. Ez tulajdonképpen egy rajzeszközhalmaz és egy sor adat kapcsolata. Az adatokkal megadhatjuk a rajzolás módját. Ezután ezt az eszközleírót használva specifikálhatjuk azt az eszközt, amelyen rajzolni szeretnénk. Például ha egy szöveget szeretnénk megjelentetni a képernyőn, akkor először rögzítjük

az eszközkapcsolat révén a karakterkészletet, a színt, a karakterek nagyságát, típusát, azután pedig specifikáljuk a kiírás helyét (x és y koordinátáit), illetve a kiírandó szöveget. Mielőtt egy alkalmazás rajzolni szeretne egy adott eszközre, egy eszközkörnyezetet kell létrehoznia, amin majd a későbbiekben a rajzolási műveleteket elvégzi. Az eszközkörnyezet valójában egy, a GDI által kezelt belső struktúra, ami különböző információkat tárol az eszköz és a rajzolás mindenkor aktuális állapotáról. Az eszközkörnyezet ezek mellett felhasználható az eszköz fizikai és logikai jellemzőinek megállapításához és az eszközzel történő direkt kommunikációhoz is.

A következő C++-program jól szemlélteti ezt a filozófiát.

```
1  void CBMPView::OnDraw(CDC* pDC)
2  {
3     CBMPDoc* pDoc = GetDocument();
4     ASSERT_VALID(pDoc);
5     CDC MemDC;
6     CPen Pen, *POldPen;
7     RECT ClientRect;
8     GetClientRect(&ClientRect)
9     MemDC.CreateCompatibleDC(NULL);
10    MemDC.SelectObject(&a);
11    int w = BM.bmWidth;
12    int h = BM.bmHeight;
13    pDC->BitBlt(10, 10, w, h, &MemDC, 0, 0, SRCCOPY);
14    Pen.CreatePen(PS_SOLID, 3, RGB(128, 128, 128));
15    POldPen=pDC->SelectObject(&Pen);
16    pDC->MoveTo(14, 11+BM.bmHeight);
17    pDC->LineTo(11+w, 11+h);
18    pDC->LineTo(11+w, 14);
19    pDC->SelectObject(POldPen);
20    Pen.DeleteObject();
21 }
```

A Windows GDI funkciók és objektumok széles skáláját bocsátja az alkalmazások rendelkezésére, amelyek segítségével azok különböző grafikus elemeket: egyeneseket, görbéket, sokszögeket, zárt alakzatokat, szöveget és bittérképeket jeleníthetnek meg. A megjelenítés során az alkalmazások különféle torzításokat: eltolást, skálázást, forgatást, komplex leképezéseket használhatnak, illetve kitöltést és mintázást alkalmazhatnak a képezett alakzatokon. A rajzolást tetszőleges területre korlátozhatják (vágás), és meghatározhatják azt is, hogy a rajzolófunkciók milyen módon módosítsák a már meglévő képet. A rendelkezésre álló eszköztárat bővebben *A Borland Delphi* grafikája című fejezetben tárgyaljuk, $C++$ szintaxisnak megfeleltetve ugyanúgy kell ezeket használni *Visual C++*-ban is.

A rajzolás számára lényeges, hogy az ablakban megjelenítendő grafika kódját egy speciális eseménykezelőben az *OnPaint* (*Visual C++*-ban *OnDraw*) kell elhelyezni, ugyanis ez automatikusan lefut, amikor az ablakot frissíti a rendszer (például előbukkan egy takarásból, kicsinyítettük, nagyítottuk, elmozdítottuk).

Két fogalmat meg kell még említenünk: a *téglalap (rectangle)* és a *régió (region)* fogalmát.

Windows alatt minden kontrollt, beleértve az ablakot is, egy téglalappal írhatunk le, pontosabban két koordináta-párost kell megadnunk: a téglalap bal felső és a jobb alsó sarkát. Ezekre a *Top*, *Left*, *Bottom*, *Right* adatokkal hivatkozhatunk. A téglalapok mellett fontos Windows felületi egységek a régiók, tetszőleges alakú, de mindenképpen zárt alakzatok, amelyek közvetlenül nem kerülnek megjelenítésre, de amelyek igen fontos funkciót töltenek be: a rajzoló műveletek hatókörét az adott alakzaton belülre korlátozzák. Felhasználásukkal nyílik lehetősége az alkalmazásoknak a téglalaptól eltérő kifestett alakzatok létrehozására, ill. egy adott rajzoló művelet az előre meghatározott határokon túlnyúló (vagy éppen hogy azon belülre eső) részeinek megjelenítésének megakadályozására. A régiók ellipszis, sokszög és téglalap (kerekített, ill. szögletes sarkú), valamint ezek tetszőleges számú és sorrendű kombinációjából létrehozható alakokat vehetnek fel. A régiók kombinálásához logikai *és*, *vagy*, *kizáró vagy* és *különbség* műveletek alkalmazhatók, amelyeknek köszönhetően gyakorlatilag bármilyen szabad alakzat kialakítható.

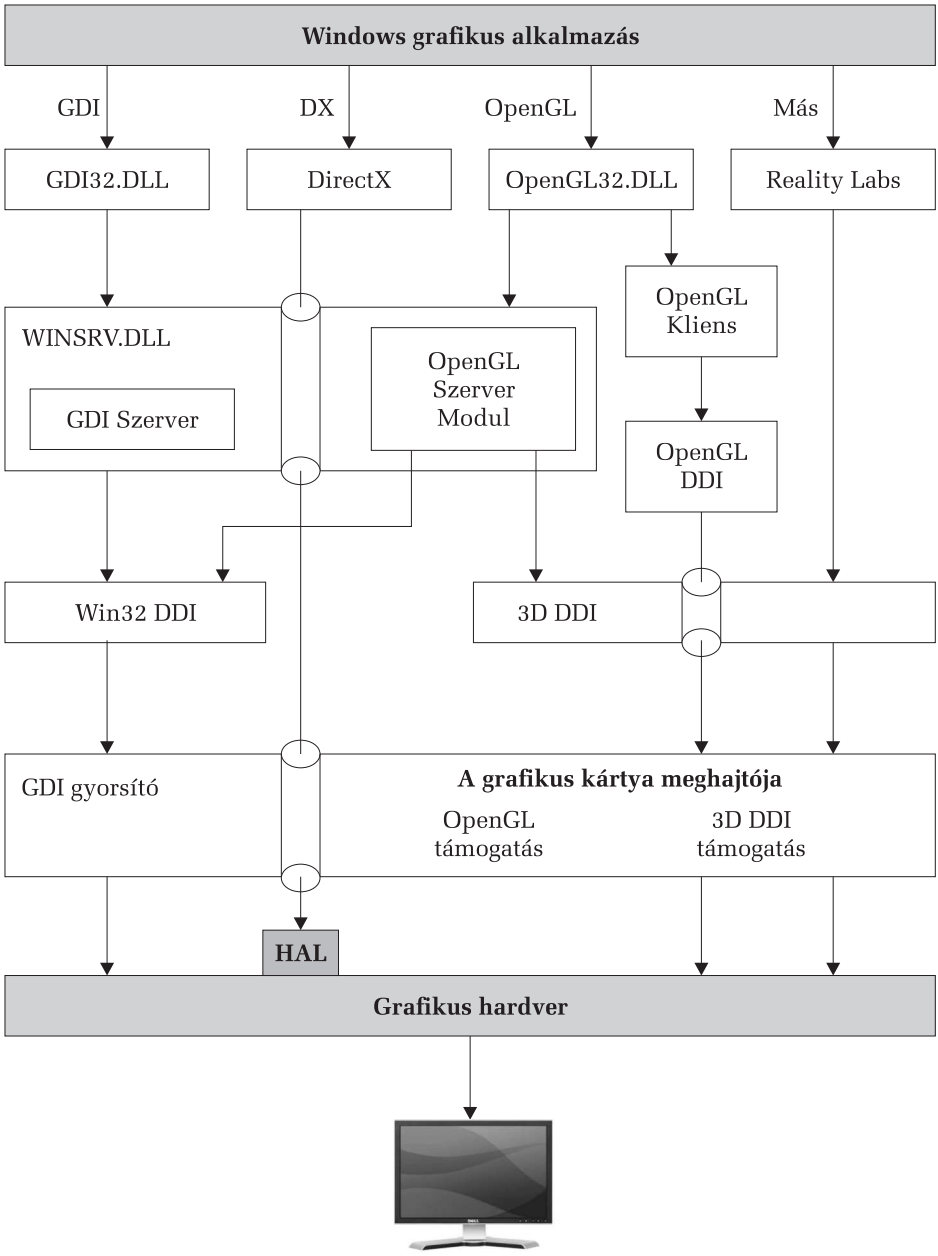
Régiókkal számos műveletet lehet elvégezni, tesztelni lehet, hogy két régió megegyezik-e, a régiók invertálhatók, eltolhatók, forgathatók, valamint megállapítható, hogy tartalmaznak-e egy adott koordinátájú pontot. Megfeleltetés létezik a régiók és a téglalapok között is, lekérdezhetők a régió minden pontját magába foglaló legkisebb téglalap sarokpontjai [42].

Ha rá akarjuk venni a Windowst, hogy fesse újra – soron kívül – az ablakot, a következő eljárásokat kell meghívnunk: `Invalidate`: érvénytelenné teszi az ablak területét és értesíti a Windowst, hogy fesse újra az ablakot; `update`, `refresh`: azonnal újrafesti az ablakot, vagy `repaint`, ami nem más, mint egy `invalidate` és egy `update` hívás.

A 4.1. ábra a Windows grafikus lehetőségeit foglalja össze. A DDI a *Device Dependent Interface (eszközfüggő interfész)*, a HAL a *Hard Array Logic (hardverszintű tömb-logika)* rövidítése.

4.2. DirectX

A DirectX (DX) a Microsoft által fejlesztett, csak Windows alatt használható rutingyűjtemény. A GDI filozófiával ellentétben a DirectX célja a hardver közvetlen elérése, és így a vezérlési folyamatok felgyorsítása. Arra volt kitalálva,



4.1. ábra. *A Windows grafikus rendszere*

hogy a különböző típusú kártyákat, drivereket egységesítse, hardverfüggetlenül lehessen alkalmazást írni.

A DirectX alapjait abban kell keresni, hogy a valós módban futó DOS lehetővé tette a hardver közvetlen manipulálását, a védett módú Windows kernel azonban már nem, többek között azért, mert egy általánosított felületet (GDI) biztosított. A programozóknak – főként a játékprogramozóknak – szükségük volt viszont arra, hogy továbbra is közvetlenül hozzáférhessenek a hardverhez, de nem akarták feladni a védett mód előnyeit. Így született meg először a Windows Games SDK, majd a DirectX.

A DirectX-et tipikusan multimédiás alkalmazások használják: játékok, médialejátszók stb. Az OpenGL-lel ellentétben szakterülete nemcsak a 2D és 3D grafika, hanem a hálózatkezelés, hangkártyakezelés, beviteli eszközök kezelése stb.

A DirectX több ezer API függvényt definiál, amelyeket az alkotók kisebb modulokra bontottak aszerint, hogy mi a feladatuk:

- *DirectDraw* – 2D grafika;
- *Direct3D* – 3D grafika;
- *DirectInput* – bemeneti eszközök: egér, billentyűzet, joystick stb.;
- *DirectPlay* – halózat, a 8-as verziótól kezdődően;
- *DirectSound* – hangkártyák, hanglejátszás és felvétel;
- *DirectSound3D* – surround hangzás;
- *DirectMusic* – zenelejátszás, egy játékban például a háttérzenét a DirectMusic szolgáltatja;
- *DirectShow* – multimédiás anyagok megjelenítését végzi, a legtöbb lejátszó program ezt használja;
- *DirectSetup* – a DirectX API összetevőinek telepítéséhez szükséges.

Könyvünknek nem célja aprólékosan ismertetni a DirectX-et, csupán egy példa segítségével ([35] alapján) illusztráljuk a működési elvét.

A DirectX programozásához első lépésként a Microsoft honlapjáról (például: *http://msdn.microsoft.com/en-us/directx/aa937788.aspx*) le kell tölteni, majd telepíteni kell a *DirectX SDK*-t, ezután megfelelően konfigurálni kell a programozási környezetet (pl. *Visual Studio* – be kell állítani a DirectX *Include* és *Lib* könyvtárak elérési útvonalait).

Ha ezzel megvagyunk, létrehozhatjuk a projektet (*Windows 32 Application*-ként), és elkezdhetjük megírni a programot.

Először az ablak létrehozását és *CALLBACK* függvényét írjuk meg. Az ablakhoz kapcsolunk egy osztályszerkezetű DirectDraw objektumot, így elérésünk lesz a DirectX parancsaihoz, adataihoz. Be kell állítanunk az ablak felbontását, színmélységét és azt, hogy teljes képernyős legyen-e vagy sem.

Következő lépésben létrehozzuk a *primary surface*-t, a képernyőn látható felületet (olyan memóriaterület a videokártya memóriájában, amely képeket, vizuális információt tartalmaz). Amit ebbe beleírunk, az azonnal megjelenik a képernyőn.

Beírjuk a használt include-állományokat, DirectX szempontjából a legfontosabb a `ddraw.h`.

```
1  #include "stdafx.h"
2  #include <windows.h>
3  #include <ddraw.h>
4  #include <stdio.h>
```

A DirectDraw objektum egy `LPDIRECTDRAW7` típusú változó lesz.

Deklaráljuk a *primary,* valamint a *back surface*-t (a háttérfelület – így valósul meg a *double buffering*: a háttérbe rajzolunk, majd amikor kész van, akkor tesszük át az előtérbe, így a megjelenítés villogásmentes lesz).

Külön felületet deklarálunk egy bittérkép számára, ezt fogjuk beolvasni és megjeleníteni a képernyőn (`lpDDSmiley`).

Ha olyan ábrát akarunk megjeleníteni, amely kilóg a képernyőből, a DirectX ebből semmit nem fog megjeleníteni. A vágást nekünk kell megoldani egy clipper segítségével (`pClipper`).

```
5  LPDIRECTDRAW7 lpDD;
6  LPDIRECTDRAWSURFACE7 lpDDPrimary;
7  LPDIRECTDRAWSURFACE7 lpDDBack;
8  LPDIRECTDRAWSURFACE7 lpDDSmiley;
9  LPDIRECTDRAWCLIPPER pClipper;
```

Ezután deklaráljuk a használt eljárásokat:

```
10 LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM,
11    LPARAM);
12 bool InitDirectX(HWND, int, int, int);
13 void ClearSurface(LPDIRECTDRAWSURFACE7, DWORD);
14 LPDIRECTDRAWSURFACE7 LoadBitmap(char*, DWORD);
15 void Render();
```

Majd megírjuk a `main` eljárást:

```
16 int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
17    hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
18 {
19   HWND hwnd;          // Az ablak leírója
20   MSG messages;       // Az alkalmazás üzenetei
21   WNDCLASSEX wincl;   // Az ablakosztály leírója
22   wincl.hInstance = hThisInstance;
23   wincl.lpszClassName = "DirectX";
24   wincl.lpfnWndProc = WindowProcedure;
25   wincl.style = CS_DBLCLKS;
```

```
26    wincl.cbSize = sizeof(WNDCLASSEX);
27    wincl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
28    wincl.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
29    wincl.hCursor = LoadCursor(NULL, IDC_ARROW);
30    wincl.lpszMenuName = NULL;
31    wincl.cbClsExtra = 0;
32    wincl.cbWndExtra = 0;
33    wincl.hbrBackground =
34    (HBRUSH)GetStockObject(LTGRAY_BRUSH);
35    // Regisztráljuk az ablakosztályt
36    if(!RegisterClassEx(&wincl)) return 0;
37    // Létrehozzuk az ablakot
38    hwnd = CreateWindowEx(0, "DirectX", "DirectX",
39    WS_POPUP, CW_USEDEFAULT, CW_USEDEFAULT, 550,
40    375, HWND_DESKTOP, NULL, hThisInstance, NULL);
41    // Inicializáljuk a DirectX-et
42    if(!InitDirectX(hwnd, 800, 600, 32))
43    {
44      MessageBox(hwnd, "Hiba a DirectDraw inicializá
45      lásánál!", "Hiba!", 0);
46      return 0;
47    }
48    // Megjelenítjük az ablakot
49    ShowWindow(hwnd, nCmdShow);
50    // Belépünk az eseménykezelő ciklusba
51    while(true)
52    {
53      Render();
54      if(PeekMessage(&messages, NULL, 0,
55      0, PM_REMOVE))
56      {
57        TranslateMessage(&messages);
58        DispatchMessage(&messages);
59      }
60    }
61    return messages.wParam;
62  }
```

Megírjuk az ablak *Callback* függvényét, amely az eseményeket kezeli le:

```
63  LRESULT CALLBACK WindowProcedure(HWND hwnd,
64  UINT message, WPARAM wParam, LPARAM lParam)
65  {
```

```
66    switch (message)  // Feldolgozzuk az üzeneteket
67    {
68      case WM_KEYDOWN:
69           switch(wParam)
70           {
71              case VK_ESCAPE:
72                exit(0);
73                break;
74           }
75           break;
76      case WM_DESTROY:
77           PostQuitMessage(0);
78           break;
79      default:
80           return DefWindowProc(hwnd, message,
81                  wParam, lParam);
82    }
83    return 0;
84  }
```

A következő függvény inicializálja a DirectX-et (2D grafika):

```
85  bool InitDirectX(HWND hwnd, int width,
86  int height, int bpp)
87  {
88    HRESULT hr;  // A visszatérési értékek
89    hr = DirectDrawCreateEx(NULL, (VOID**)&lpDD,
90    IID_IDirectDraw7, NULL);
91    if(hr!=DD_OK) return false;
92    // Beállítjuk a teljesképernyős módot
93    hr = lpDD->SetCooperativeLevel(hwnd,
94    DDSCL_EXCLUSIVE|DDSCL_FULLSCREEN );
95    if(hr!=DD_OK) return false;
96    // Beállítjuk a képernyő felbontását és színmélységét
97    hr=lpDD->SetDisplayMode(width, height, bpp, 0, 0);
98    if(hr!=DD_OK) return false;
99    // Deklarálunk egy surface leírót
100   DDSURFACEDESC2 ddsd;
101   ZeroMemory(&ddsd, sizeof(ddsd));
102   ddsd.dwSize = sizeof(ddsd);
103   ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
104   ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
105   DDSCAPS_FLIP | DDSCAPS_COMPLEX ;
```

```
106   ddsd.dwBackBufferCount = 1;
107   // Létrehozzuk a primary surface-t
108   hr = lpDD->CreateSurface(&ddsd, &lpDDPrimary,
109   NULL);
110   if(hr!=DD_OK) return false;
111   // Létrehozzuk a back surface-t
112   DDSCAPS2 ddscaps;
113   ZeroMemory(&ddscaps, sizeof(ddscaps));
114   ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
115   // Hozzácsatoljuk a primary surface-hez
116   hr = lpDDPrimary->GetAttachedSurface(
117   &ddscaps, &lpDDBack);
118   if(hr!=DD_OK) return false;
119   // Létrehozzuk a clippert és hozzárendeljük az ablakhoz
120   hr = lpDD->CreateClipper(0, &pClipper, NULL);
121   pClipper->SetHWnd(0, hwnd);
122   hr = lpDDBack->SetClipper(pClipper);
123   // Beolvassuk a Bitmap-et
124   lpDDSmiley=LoadBitmap("smiley.bmp",0x00000000);
125   if(lpDDSmiley==NULL)
126   {
127     MessageBox(0, "Smiley.bmp - betöltési hiba!",
128     "Hiba!", 0);
129     return false;
130   }
131   return true;
132 }
```

A következő függvény letöröl egy felületet, pontosabban feltölti azt egy megadott színnel.

```
133 void ClearSurface(LPDIRECTDRAWSURFACE7 lpDDSurf, DWORD szin)
134 {
135   DDBLTFX ddbltfx;
136   ZeroMemory(&ddbltfx, sizeof(ddbltfx));
137   ddbltfx.dwSize = sizeof(ddbltfx);
138   ddbltfx.dwFillColor = szin;
139   lpDDSurf->Blt(NULL, NULL, NULL,
140   DDBLT_COLORFILL, &ddbltfx );
141 }
```

Fontos itt a `Blt` függvény használata, amelynek segítségével átmásolunk egy surface-ból valamekkora területet egy másik surface-be (*blitting*). A függvény egy

téglalapot másol át egy másik téglalapba, ha a két téglalap mérete nem egyezik meg, akkor megnyújtja vagy szűkíti a képet (*stretch*).

A következő függvény segítségével olvasunk be egy Bitmap (BMP) képet (lásd a **A BMP állományformátum** című alfejezetet).

A képet úgy olvassuk be, hogy definiálunk egy transzparens színt, így minden, ami ilyen színű a képen, átlátszó lesz.

```
142 LPDIRECTDRAWSURFACE7 LoadBitmap(char *path, DWORD colorkey)
143 {
144   HRESULT hRet;
145   LPDIRECTDRAWSURFACE7 lpDDPic;
146   DWORD* lPSurface=NULL;
147   int lPitch=0;
148   FILE *fp;
149   DWORD szine=0, size=0;
150   unsigned long width,height;
151   short type, lineend;
152   unsigned char *sor;
153
154   fp=fopen(path, "rb");
155   if(fp==NULL) return NULL;
156   fread(&type, sizeof(short), 1, fp);
157   if(type!=0x4D42) // nem BM
158   {
159     fclose(fp);
160     return NULL;
161   }
162   fread(&size, sizeof(DWORD), 1, fp);
163   fseek(fp, 18, SEEK_SET);
164   fread(&width, sizeof(long), 1, fp);
165   fread(&height, sizeof(long), 1,fp);
166   lineend = (size-54-(width*height*3))/height;
167   DDSURFACEDESC2 ddsd;
168   ZeroMemory(&ddsd, sizeof(ddsd));
169   ddsd.dwSize = sizeof(ddsd);
170   ddsd.dwFlags = DDSD_CAPS | DDSD_PIXELFORMAT |
171   DDSD_WIDTH | DDSD_HEIGHT;
172   ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
173   DDSCAPS_VIDEOMEMORY;
174   ddsd.dwWidth = width;
175   ddsd.dwHeight = height;
176   ddsd.ddpfPixelFormat.dwSize =
177   sizeof(ddsd.ddpfPixelFormat);
```

```
178   ddsd.ddpfPixelFormat.dwFlags = DDPF_RGB;
179   ddsd.ddpfPixelFormat.dwRGBBitCount = 32;
180   ddsd.ddpfPixelFormat.dwRBitMask = 0x00ff0000;
181   ddsd.ddpfPixelFormat.dwGBitMask = 0x0000ff00;
182   ddsd.ddpfPixelFormat.dwBBitMask = 0x000000ff;
183   ddsd.ddpfPixelFormat.dwRGBAlphaBitMask =
184   0x00000000;
185   hRet = lpDD->CreateSurface(&ddsd, &lpDDPic,
186   NULL);
187   if (hRet!=DD_OK) return NULL;
188   // Az átlátszó szín a felületen
189   DDCOLORKEY ddck;
190   ddck.dwColorSpaceLowValue = colorkey;
191   ddck.dwColorSpaceHighValue = colorkey;
192   lpDDPic->SetColorKey(DDCKEY_SRCBLT, &ddck);
193   ZeroMemory(&ddsd, sizeof(ddsd));
194   ddsd.dwSize=sizeof(ddsd);
195   if((lpDDPic->Lock(NULL, &ddsd, DDLOCK_WAIT,
196   NULL))!= DD_OK) MessageBox(0, "Felület lock
197   hiba!", "Hiba!", MB_OK);
198   lPitch = (int)ddsd.lPitch;
199   lPSurface = (DWORD*)ddsd.lpSurface;
200   fseek(fp, 54, SEEK_SET);
201   for(int k=height-1; k>=0; --k)
202   {
203     sor=(unsigned char*)calloc(3*width,
204     sizeof(unsigned char));
205     fread(sor, width*3*sizeof(char), 1, fp);
206     for(int i=0; i<width; ++i)
207     {
208       szine = (*(sor+i*3+2)<<16) |
209       (*(sor+i*3+1)<<8) | (*(sor+i*3));
210       lPSurface[i+k*(lPitch>>2)]=szine;
211     }
212     free(sor);
213     if(!feof(fp)) fseek(fp, lineend, SEEK_CUR);
214   }
215   fclose(fp);
216   lpDDPic->Unlock(NULL);
217   return lpDDPic;
218 }
```

4.2. ábra. *A megjelenítendő bitmap*

```
219 void Render()
220 {
221   HRESULT hRet;
222   RECT r;
223   ClearSurface(lpDDBack, 0x000000ff);
224   r.top=275;
225   r.left=180;
226   r.right=r.left+50;
227   r.bottom=r.top+50;
228   lpDDBack->Blt(&r, lpDDSmiley, 0, DDBLT_KEYSRC,
229   0);
230   hRet = lpDDPrimary->Flip(NULL, 0);
231 }
```

Itt végezzük el a rajzolási feladatokat (jelen pillanatban itt jelentetjük meg a bitmap-et). Lényeges a `hRet = lpDDPrimary->Flip(NULL, 0);` sor, ez flippingeli a primary és a back surface-ket.

4.3. A Borland Delphi grafikája

A *Delphi* grafikája teljesen ráépül a Windows grafikus alrendszerére, a GDI-re.

A *Delphi* rendszer az összes grafikus objektumot és megjelenítő rutint a *Graphics* unitban tárolja. Az eszközkapcsolatot és magát a rajzolás alapegységét is megvalósító objektumot a `TCanvas` osztály képezi. Minden speciális megjelenítő objektum (Form, Printer, Image) tartalmaz egy `TCanvas` típusú `Canvas` nevet viselő tulajdonságot. A konkrét eszközkapcsolat-meghatározás és -rajzolás ezen `Canvas` objektum segítségével történik, amely nem más, mint az eszközkapcsolat objektumorientált megfogalmazása.

A *Graphics* unit használja a hagyományos API (*Application Programming Interface*) függvényeket és eljárásokat is. A `Canvas` `Handle` tulajdonsága tulajdonképpen az eszközkapcsolat `HDC` típusú leírásával egyezik meg. A tulajdonság

segítségével tehát bármikor áttérhetünk a hagyományos API rutinok használatára is.

A Canvas objektumot egy festőkészletként képzelhetjük el. A Canvas tulajdonságok a rajzolási attribútumokat, a rajzeszközök és a rajzvászon jellegzetességeit állítják, a metódusok pedig a konkrét rajzoláshoz szükséges rutinokat biztosítják. A Canvas objektum alapvető tulajdonságai alapvető információkat szolgálnak a toll (vonalas ábrák rajzolása), az ecset (kitöltőminták), a fontok (szövegek megjelenítése) és a bittérképek attribútumairól, jellegzetességeiről.

4.3.1. Tollak

A vonalas ábrák készítésének alapvető eszköze a toll. A tollakat a TPen osztály és az objektumok Pen tulajdonságai valósítják meg. A tollak jellemzői a szín (Color), vonalvastagság (Width), vonaltípus (Style) és a rajzolási mód (Mode).

A *Delphi* rendszer a színeket a TColor = -(COLOR_ENDCOLORS + 1)..$2FFFFFF; típussal kezeli le. A színdefinícióban a piros, zöld és kék értékeket az *rr*, *gg* és *bb* számok jellemzik ($00bbggrr). Saját szín keverésére is van lehetőség a **function**RGB(R: byte; G: byte; B: byte): longint; függvény segítségével. A *Graphics* unit a leggyakrabban használt színeket konstansként deklarálja (clBlack = TColor($000000);, clRed = TColor ($0000FF); stb.).

A húzott vonal vastagságát a Width tulajdonság által lehet megadni. A mértékegység itt a pixel.

A húzott vonal típusát a Style tulajdonsággal lehet beállítani. Ez a tulajdonság TPenStyle = (psSolid, psDadh, psDot, psDashDot, psDashDotDot, psClear, psInsideFrame); típusú.

A Mode tulajdonság segítségével a rajzolási módot állíthatjuk be. A rajzolási mód azt jelenti, hogy bizonyos logikai műveleteket használva, a háttér színe és a toll színe fogja meghatározni a vonal színét. A megfelelő logikai műveleteket a TPenMode = (pmBlack, pmWhite, pmNop, pmNot, pmCopy, pmNotCopy, pmMergePenNot, pmMaskPenNot, pmMergeNotPen, pmMaskNotPen, pmMerge,pmNotMerge, pmMask, pmNotMask, pmXor, pmNotXor); típus definiálja.

Ebben a szellemben, a TPen osztály a következő deklarációkat foglalja magában:

```
1  TPen = class(TGraphicsObject)
2    private
3      FMode: TPenMode;
4      procedure GetData(var PenData: TPenData);
5      procedure SetData(const PenData: TPenData);
6    protected
7      function GetColor: TColor;
```

```
8     procedure SetColor(Value: TColor);
9     function GetHandle: HPen;
10    procedure SetHandle(Value: HPen);
11    procedure SetMode(Value: TPenMode);
12    function GetStyle: TPenStyle;
13    procedure SetStyle(Value: TPenStyle);
14    function GetWidth: Integer;
15    procedure SetWidth(Value: Integer);
16  public
17    constructor Create;
18    destructor Destroy; override;
19    procedure Assign(Source: TPersistent); override;
20    property Handle: HPen read GetHandle write
21    SetHandle;
22  published
23    property Color: TColor read GetColor write SetColor
24    default clBlack;
25    property Mode: TPenMode read FMode write SetMode
26    default pmCopy;
27    property Style: TPenStyle read GetStyle write
28    SetStyle default psSolid;
29    property Width: Integer read GetWidth write SetWidth
30    default 1;
31  end;
```

4.3.2. Ecsetek

Ábrák kifestéséhez ecseteket használunk. A `Canvas` objektum hasonlóan kezeli a tollakat és az ecseteket. Minden festő metódus az aktuális ecsetet használja. Az ecset objektumorientált koncepciója a `TBrush` osztály által valósul meg. A `Brush` változók jellemzői a szín és a kifestés módja. A kifestés módja a tulajdonképpeni kitöltőmintát adja meg. Ez a következő típusdeklarációnak felel meg: `TBrushStyle = (bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross);`. Ha beállítjuk a `Bitmap` tulajdonságát, akkor az így megadott bittérképet használja festőmintaként. A `TBrush` osztály tehát a következő:

```
1  TBrush = class(TGraphicsObject)
2    private
3      procedure GetData(var BrushData: TBrushData);
4      procedure SetData(const BrushData: TBrushData);
5    protected
6      function GetBitmap: TBitmap;
```

```
7     procedure SetBitmap(Value: TBitmap);
8     function GetColor: TColor;
9     procedure SetColor(Value: TColor);
10    function GetHandle: HBrush;
11    procedure SetHandle(Value: HBrush);
12    function GetStyle: TBrushStyle;
13    procedure SetStyle(Value: TBrushStyle);
14  public
15    constructor Create;
16    destructor Destroy; override;
17    procedure Assign(Source: TPersistent); override;
18    property Bitmap: TBitmap read GetBitmap write
19    SetBitmap;
20    property Handle: HBrush read GetHandle write
21    SetHandle;
22  published
23    property Color: TColor read GetColor write SetColor
24    default clWhite;
25    property Style: TBrushStyle read GetStyle write
26    SetStyle default bsSolid;
27  end;
```

4.3.3. Fontok

A karakterek eszközfüggetlen megjelenítését a Windows a *TrueType* fontok segítségével érte el. A *TrueType* fontok tulajdonképpen pontok és speciális algoritmusok halmaza, amelyek eszköztől és felbontástól függetlenül képesek karaktereket megjeleníteni.

A `Canvas` tulajdonsága a `Font` is, amely egy `TFont` típusú objektum és a karakterek beállításait szolgálja. A `TFont` tulajdonságai a font mérete (`Size: integer`), a karakterek színe (`Color: TColor`), a karakter által lefoglalt cella magassága (`Height: integer`), a font neve (`Name: TfontName`), valamint a karakter stílusa (`Style: TFontStyles`). A dőlt, félkövér, aláhúzott vagy áthúzott betűket a következő típus segítségével lehet definiálni: `TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut); TfontStyles = setofTFontStyle;`

A `TFontName` típust a következő deklaráció határozza meg: `TfontName = string(LF_FACESIZE - 1);`

Természetesen amikor karaktereket akarunk megjelentetni, akkor beállíthatjuk a `TFont` objektum ezen tulajdonságait, de elegánsabb megoldás az, hogy egy `TFontDialog` típusú dialógusdoboz segítségével állítjuk be a karakterek jellemzőit.

4.3.4. Bittérképek

A bittérképek speciális memóriaterületeket jelölnek, amelyeknek bitjei egy-egy kép megjelenését definiálják. Fekete-fehér képernyőn nagyon egyszerű ez a megjelenítés, ha az illető bit 0, akkor a képpont fekete, ha pedig 1, akkor a képpont fehér. Színes képernyők esetén nem elegendő egyetlen bit a képpont tárolásához, ekkor vagy több szomszédos bit segítségével kódoljuk a képpontot, vagy a bittérképet több színsíkra tagoljuk, és ezek együttesen határozzák meg a képpontot.

A bittérképet a `TBitmap` típus valósítja meg, amely számos információt tartalmaz a bittérkép méretéről (`Height`, `Width`), típusáról (`Monochrome`), arról, hogy tartalmaz-e értékes információt (`Empty`), valamint metódusai segítségével kimenthetjük, beolvashatjuk (`SaveToFile`, `LoadFromFile`, `LoadFromStream`, `SaveToStream`) vagy a vágóasztal segítségével átadhatjuk a tárolt információt (`LoadFromClipboardFormat`, `SaveToClipboardFormat`).

Maga a `TBitmap` is tartalmaz egy `Canvas` tulajdonságot, amely segítségével rajzolhatunk, írhatunk a bittérképre.

4.3.5. A Canvas

Ezen ismeretek birtokában rátérhetünk a `TCanvas` objektum ismertetésére. Mint már említettük, a `Canvas` nem más, mint az eszközkapcsolat-leíró objektumorientált megfogalmazása. A `Canvas` tulajdonságok a rajzolás jellemzőit állítják be, a `Canvas` metódusok pedig megvalósítják a rajzolást. A `TCanvas` típus a következő:

```
1  TCanvas = class(TPersistent)
2    private
3      FHandle: HDC;
4      State: TCanvasState;
5      FFont: TFont;
6      FPen: TPen;
7      FBrush: TBrush;
8      FPenPos: TPoint;
9      FCopyMode: TCopyMode;
10     FOnChange: TNotifyEvent;
11     FOnChanging: TNotifyEvent;
12     FLock: TRTLCriticalSection;
13     FLockCount: Integer;
14     procedure CreateBrush;
15     procedure CreateFont;
16     procedure CreatePen;
17     procedure BrushChanged(ABrush: TObject);
```

```
18      procedure DeselectHandles;
19      function GetClipRect: TRect;
20      function GetHandle: HDC;
21      function GetPenPos: TPoint;
22      function GetPixel(X, Y: Integer): TColor;
23      procedure FontChanged(AFont: TObject);
24      procedure PenChanged(APen: TObject);
25      procedure SetBrush(Value: TBrush);
26      procedure SetFont(Value: TFont);
27      procedure SetHandle(Value: HDC);
28      procedure SetPen(Value: TPen);
29      procedure SetPenPos(Value: TPoint);
30      procedure SetPixel(X, Y: Integer; Value: TColor);
31    protected
32      procedure Changed; virtual;
33      procedure Changing; virtual;
34      procedure CreateHandle; virtual;
35      procedure RequiredState(ReqState: TCanvasState);
36    public
37      constructor Create;
38      destructor Destroy; override;
39      procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4:
40      Integer);
41      procedure BrushCopy(const Dest: TRect; Bitmap:
42      TBitmap; const Source: TRect; Color: TColor);
43      procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4:
44      Integer);
45      procedure CopyRect(const Dest: TRect; Canvas:
46      TCanvas; const Source: TRect);
47      procedure Draw(X, Y: Integer; Graphic: TGraphic);
48      procedure DrawFocusRect(const Rect: TRect);
49      procedure Ellipse(X1, Y1, X2, Y2: Integer);
50      procedure FillRect(const Rect: TRect);
51      procedure FloodFill(X, Y: Integer; Color: TColor;
52      FillStyle: TFillStyle);
53      procedure FrameRect(const Rect: TRect);
54      procedure LineTo(X, Y: Integer);
55      procedure Lock;
56      procedure MoveTo(X, Y: Integer);
57      procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4:
58      Integer);
59      procedure Polygon(const Points: array of TPoint);
```

```
60      procedure Polyline(const Points: array of TPoint);
61      procedure Rectangle(X1, Y1, X2, Y2: Integer);
62      procedure Refresh;
63      procedure RoundRect(X1, Y1, X2, Y2, X3, Y3:
64      Integer);
65      procedure StretchDraw(const Rect: TRect; Graphic:
66      TGraphic);
67      function TextExtent(const Text: string): TSize;
68      function TextHeight(const Text: string): Integer;
69      procedure TextOut(X, Y: Integer; const Text:
70      string);
71      procedure TextRect(Rect: TRect; X, Y: Integer;
72      const Text: string);
73      function TextWidth(const Text: string): Integer;
74      function TryLock: Boolean;
75      procedure Unlock;
76      property ClipRect: TRect read GetClipRect;
77      property Handle: HDC read GetHandle write
78      SetHandle;
79      property LockCount: Integer read FLockCount;
80      property PenPos: TPoint read GetPenPos write
81      SetPenPos;
82      property Pixels[X, Y: Integer]: TColor read
83      GetPixel write SetPixel;
84      property OnChange: TNotifyEvent read FOnChange
85      write FOnChange;
86      property OnChanging: TNotifyEvent read FOnChanging
87      write FOnChanging;
88    published
89      property Brush: TBrush read FBrush write SetBrush;
90      property CopyMode: TCopyMode read FCopyMode write
91      FCopyMode default cmSrcCopy;
92      property Font: TFont read FFont write SetFont;
93      property Pen: TPen read FPen write SetPen;
94    end;
```

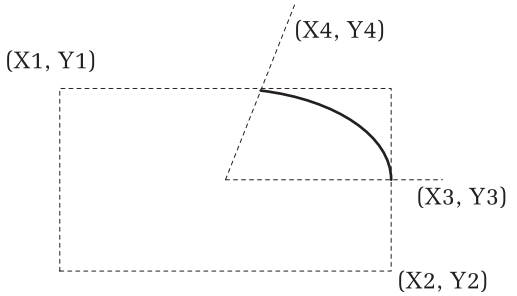
A Canvas rajzolási módszerei hasonlítanak a *Borland Pascal* BGI grafikájához, egy pár fontosabb eltéréssel. A pixelgrafika itt a `Pixels[X, Y: Integer]: TColor;` tulajdonság segítségével valósul meg. Az `X` és az `Y` indexek a képernyő megfelelő pontjának a koordinátáit jelentik, a tömbelem pedig a pont színét. Teljes kifestett ellipszist rajzolhatunk az `Ellipse(X1, Y1, X2, Y2: Integer);` metódus segítségével. A megadott paraméterek azt a téglalapot definiálják,

amely tartalmazza az ellipszist. Az ellipszis középpontja a téglalap középpontja lesz, illetve tengelyei is megegyeznek a téglalap tengelyeivel.

Az ellipszisívek, ellipsziscikkek és ellipszisszeletek rajzolása egy kissé szokatlan. Ezek a következő metódusok segítségével történnek:

```
procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);
procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);
procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);
```

A metódusoknak meg kell adni az ellipszist befogadó téglalapot (`X1`, `Y1`, `X2`, `Y2`), egy kezdőpontot (`X3`, `Y3`), valamint egy végpontot (`X4`, `Y4`). A kezdő- és a végpont egy szögtartományt definiál. Ez az ellipszisív, -cikk vagy -szelet ebben a szögtartományban lesz meghúzva, az aktuális tollal és rajzolási móddal, az óramutató járásával ellentétes irányban.



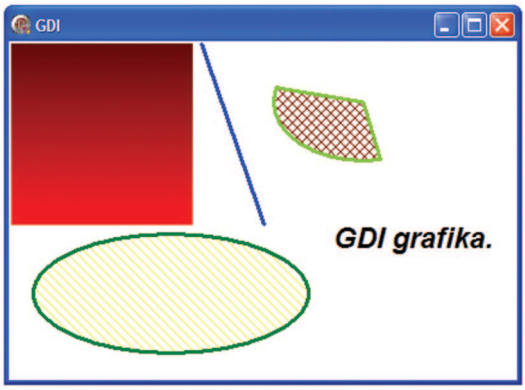
4.3. ábra. *Ellipszisívek rajzolása*

Lekerekített sarkú téglalapot rajzolhatunk a `RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer);` metódus segítségével. Az `X3`, `Y3` az ellipszis nagy-, illetve kistengelye.

A rajzvászonra a `TextOut(X, Y: Integer; const Text: String);` illetve a `TextRect(Rect: TRect; const Text: String);` metódus segítségével írhatunk. A `TextOut` az (`X`, `Y`) ponttól kezdve kiírja a `Text` szöveget, a `TextRect` pedig a `Text` szöveget csak a `Rect` téglalap által meghatározott részben jeleníti meg. Azt, hogy mekkora helyet foglal le a kiírt szöveg, a `TextExtent(const Text: string): TSize;` függvény segítségével tudhatjuk meg. Ha csak a szöveg hosszára vagy magasságára vagyunk kíváncsiak, akkor a `TextHeight(const Text: string): Integer;` vagy a `TextWidth(const Text: string): Integer;` függvényeket használjuk.

Ha valamilyen grafikus ábrát vagy bittérképet kívánuk megjeleníteni a rajzvásznon, akkor a `Draw(X, Y: Integer; Graphic: TGraphic);` vagy a `StretchDraw(const Rect: TRect; Graphic: TGraphic);` metódust használjuk. A `StretchDraw` metódus nagyítva vagy kicsinyítve jelenteti meg az ábrát úgy, hogy ez teljesen töltse ki a `Rect` téglalapot.

A következő példaprogram a `Canvas` rajzolási lehetőségeit mutatja be. Az űrlap (form) rajzvásznára rajzolunk, de a leírt kódrész ugyanígy használható bármilyen komponens esetén, amely rendelkezik `Canvas`szal (pl. `TImage`, `TPaintBox`, `TPanel` stb.).



4.4. ábra. *GDI lehetőségek* Delphi*ben*

- Indítsuk el a *Delphi* környezetet, megjelenik az üres űrlap (form).
- Az *Object Inspector*ban adjunk nevet a formnak: *Name* = *frmMain*, és adjuk meg az ablak címét: *Caption* = *GDI*.
- Állítsuk be a form színét fehérre: *Color* = *clWhite*.
- Kattintsunk duplán az *Object Inspector Events* (Események) fülecskéjén az *OnPaint* eseménykezelőre, és máris írhatjuk a grafikus utasításokat (a grafikus kódot mindig ebbe az eseménykezelőbe kell elhelyezni, így a grafika nem tűnik el, ha az ablakot frissíti a rendszer).
- A unit forráskódja a következő:

```
1  unit uMain;
2
3  interface
4
5  uses
6    Windows, Messages, SysUtils, Variants, Classes,
7    Graphics, Controls, Forms, Dialogs;
8
9  type
10   TfrmMain = class(TForm)
11     procedure FormPaint(Sender: TObject);
12   end;
13
14 var
```

```
15    frmMain: TfrmMain;
16
17  implementation
18
19  {$R *.dfm}
20
21  procedure TfrmMain.FormPaint(Sender: TObject);
22  var
23    x, y: integer;
24  begin
25    with Canvas do
26     begin
27       for x := 2 to 152 do
28        for y := 2 to 152 do
29          Pixels[x, y] := RGB(100+y, 0, 0);
30       Pen.Color := clBlue;
31       Pen.Width := 3;
32       MoveTo(160, 2);
33       LineTo(212, 152);
34       Pen.Color := RGB(128, 234, 45);
35       Brush.Color := clMaroon;
36       Brush.Style := bsDiagCross;
37       Pie(220, 2, 370, 100, 1, 1, 400, 400);
38       Pen.Color := clGreen;
39       Brush.Color := clYellow;
40       Brush.Style := bsFDiagonal;
41       Ellipse(20, 160, 250, 260);
42       Font.Name := 'Arial';
43       Font.Size := 18;
44       Font.Style := [fsBold, fsItalic];
45       TextOut(270, 150, 'GDI grafika.');
46     end;
47  end;
48
49  end.
```

4.3.6. Nyomtatás

*Delphi*ben a grafikus nyomtatás a *Printers* unit használatával valósul meg. Ez a unit deklarál egy `TPrinter` típusú `Printer` objektumot, amelynek tulajdonságai között szerepel a `Canvas` is. Ha erre a `Canvas`ra rajzolunk vagy írunk, akkor az megjelenik a nyomtatón. Az aktuális papírméretről információkat nyerhetünk

a Printer objektum PageHeight, illetve PageWidth tulajdonságai segítségével. A nyomtatást a BeginDoc metódussal kezdeményezhetjük és az EndDoc metódussal fejezzük be. Bármikor áttérhetünk új oldalra a NewPage metódus meghívásával.

```
1  with Printer do
2    begin
3      BeginDoc;
4      Canvas. TextOut (20, 20, 'Az első lap.');
5      Canvas.MoveTo(50, 50);
6      Canvas.LineTo(200, 200);
7      Canvas.Rectangle(40, 40, 250, 220);
8      NewPage;
9      Canvas.TextOut(20, 20, 'A második lap.');
10     EndDoc;
11   end;
```

5. FEJEZET

AZ OPENGL

Az *OpenGL* platform- és operációs rendszer független grafikus API. A Silicon Graphics, Inc. (SGI) kifejlesztette rendszer jelenlegi verziója 2.1. A projekt annyira sikeresnek bizonyult, hogy a Microsoft is beállt az OpenGL fejlesztésébe. A függvénykönyvtár pár száz alacsony szintű rutinból áll, amelyek által nagyon jól ki lehet használni a hardvereket – több hardverkészítő is beépítette már ezeket a rutinokat hardver szinten. Az OpenGL nem tartalmaz komplex formákat, alakzatokat stb., csak a legegyszerűbb elemeket: pontot (vertexet), vonalat, poligonokat. A programozó kell ezekből felépítse a saját komplex formáit. Ellentétben a DirectX-szel, az OpenGL nem tartalmaz hang-, hálózati vagy olyan komponenseket, melyek nincsenek direkt kapcsolatban a grafikával.

Az OpenGL funkciói:

- színtér definiálása háromdimenziós primitívekkel,
- nézőpont specifikálása,
- megvilágítási modellek alkalmazása,
- a megvilágított színtérről árnyalt modell készítése,
- árnyalások és textúrák alkalmazása,
- antialiasing (élsimítás),
- motion blur (mozgó objektumok körvonalainak elmosása),
- atmoszféraeffektusok kezelése (pl. köd),
- animáció,
- árnyaló nyelv.

Az OpenGL kliens–szerver architektúrára épül, a kliens a program, amely utasításokat küld a szervernek, a szerver a grafikus kártya, amely parancsokat fogad, feldolgozza azokat és megjeleníti a képet. A parancsok átadása szabványos protokoll szerint működik, így a kliens és a szerver egy-egy külön gép is lehet.

Az OpenGL 1.0-es verziója 1992-ben jelent meg az SGI fejlesztésében (IRIS GL). Funkcionalitása az alap geometriai primitívek kirajzolására, flat és Gouraud-féle árnyalásra, parametrizálható textúrákra, bufferes ábrázolásra terjedt.

1996-ban jelent meg az 1.1-es verzió, amely vertex-tömböket, magasabb szintű textúrázási műveleteket tartalmazott.

Az 1.2-es verzió 1998-ban jelent meg, bővült az OpenGL mechanizmusa, megjelentek a 3D textúrák.

Az OpenGL 1.3 2001-ben már tömörített és multitextúrázást is tudott.

2002-ben az OpenGL 1.4-es automatikus MipMap-technológiát, ködkoordinátákat vezetett be.

Rá egy évre jelent meg a vertex-buffereket tartalmazó 1.5-ös.

2004-ben már az OpenGL 2.0 látott napvilágot, a legfontosabb újítás a videokártya GPU programozási lehetősége volt.

2006-ban jelent meg az utolsó verzió, a 2.1-es, itt már egységes mátrixműveletek (nemcsak négyzetes mátrixokra) és új árnyaló nyelvlehetőségek bővítették a funkcionalitást.

1992-ben alakult meg az ARB (Architecture Review Board), az OpenGL felügyeleti testülete, amelynek a Compaq, az IBM, az Intel, a Microsoft és a Silicon Graphics voltak a tagjai. A testület 1994-ben az Evans & Sutherlanddel, valamint a 3D Labsszal bővült, 1995-ben a Hewlett Packarddal, 1996-ban a Sun Microsystemsszel, 1998-ban az nVidia-val, 1999-ben az ATi-val, 2001-ben az Apple-lel.

2003-ban a Microsoft kilépett.

2006. július 31-én az ARB átadta az OpenGL felügyeleti jogát a Khronosnak. A Khronos ipari konzorciumot 2000-ben alapította a 3D Labs, ATi, Discreet, Evans & Sutherland, Intel, nVidia, SGI, Sun Microsystems. A tagság nyitott, jelenleg kb. 120 tagja van, a fent említetteken kívül a legfontosabbak a Nokia, Ericsson, Creative, AMD, DELL, Samsung, Texas Instruments, Sony, ARM stb.

A Khronos felügyelete alatt számos szabvány jelent meg: OpenKODE™ , OpenGL® ES, OpenMAX™, OpenVG™, OpenSL ES™, OpenML™, COLLADA™, glFX, OpenGL® SC, a játékprogramozástól a vektorgrafikus gyorsítókon keresztül a 2 és 3D-s API-kig.

Az OpenGL alacsony szintű függvényeket magas szintű utility-könyvtárak támogatják (pl. GLU, GLUJ, GLUT), ezeknek a feladata az ablakozó rendszer kezelése, a magasabb szintű objektumok (kocka, gömb, kúp, henger, görbék, felületek stb.) kialakítása és megjelenítése.

A következőkben [45], [49], [60], valamint [86] alapján összefoglaljuk az OpenGL lehetőségeit. Az OpenGL parancsait itt *C* szintaxis szerint adjuk meg, de ezek értelemszerűen átírhatók bármilyen nyelvre.

5.1. Az OpenGL alapfogalmai

5.1.1. Az OpenGL adattípusai

Az OpenGL a jobb hordozhatóság (platformfüggetlenség) érdekében saját adattípusokkal rendelkezik, amelyeket a következő felsorolás foglal össze:

`GLbyte` belső ábrázolása: 8 bites egész, C megfelelője: `signed char`, Delphi megfelelője: `shortint`, Szuffix: `b`

`GLshort` belső ábrázolása: 16 bites egész, C megfelelője: `short`, Delphi megfelelője: `smallint`, Szuffix: `s`

`GLint`, `GLsize` belső ábrázolása: 32 bites egész, C megfelelője: `long`, Delphi megfelelője: `longint`, Szuffix: `l`

`GLfloat`, `GLclampf` belső ábrázolása: 32 bites lebegőpontos, C megfelelője: `float`, Delphi megfelelője: `single`, Szuffix: `f`

`GLdouble`, `GLclampd` belső ábrázolása: 64 bites lebegőpontos, C megfelelője: `double`, Delphi megfelelője: `double`, Szuffix: `d`

`GLubyte`, `GLboolean` belső ábrázolása: 8 bites előjel nélküli egész, C megfelelője: `unsigned char`, Delphi megfelelője: `byte`, Szuffix: `ub`

`GLushort` belső ábrázolása: 16 bites előjel nélküli egész, C megfelelője: `unsigned short`, Delphi megfelelője: `word`, Szuffix: `us`

`GLuint`, `GLenum`, `GLbitfield` belső ábrázolása: 32 bites előjel nélküli egész, C megfelelője: `unsigned long`, Delphi megfelelője: `longword`, Szuffix: `ui`

5.1.2. Az OpenGL parancsok szintaxisa

Egy OpenGL parancs eljárás vagy függvény lehet. Minden OpenGL parancs a `gl` prefixszel kezdődik. Egy parancsnak általában több változata is lehet, amelyek az argumentumok átadásában különböznek, így egy OpenGL parancs egy névből áll, amelyet maximum 4 karakter követ. Az első karakter az argumentumok számát jelöli. A második karakter vagy karakterpár az argumentumok típusát jelzi: 8 bites egész, 16 bites egész, 32 bites egész, egyszeres pontosságú lebegőpontos, vagy duplapontosságú lebegőpontos szám. Az utolsó karakter, ha van, akkor ez `v`, és azt jelzi, hogy az argumentum egy vektorra mutató pointer.

A fentiek alapján egy OpenGL parancs általános alakja:

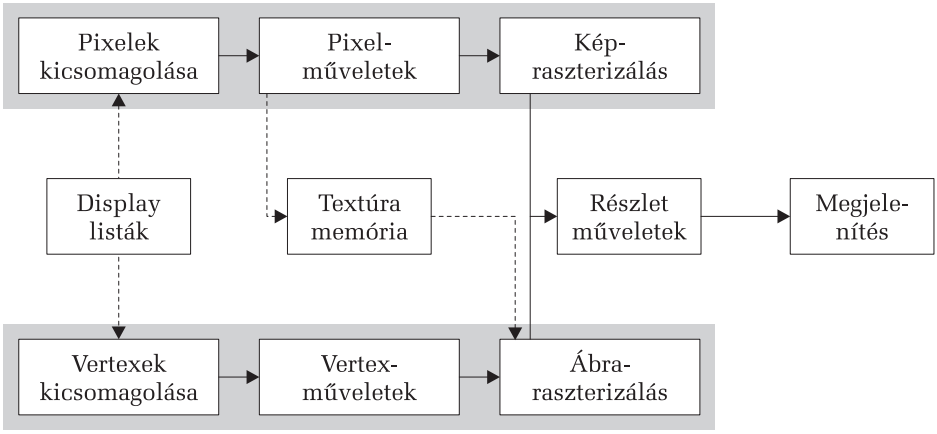
```
VisszatérésiÉrtékTípusa Név{# 1 2 3 4}{# b s i f d ub us
ui}{# v}(# T arg1, ..., T argn);
```

A `#` az üres karaktert jelenti (semmi). Ha a parancs nevének utolsó karaktere `v`, akkor csak az arg_1 van jelen, és az egy n darab, adott típusú értéket tartalmazó vektorra mutató pointer.

5.1.3. Az OpenGL primitívei

Az OpenGL *geometriai primitíveket* rajzol, ezek a számítógépes grafika alapelemei. Az OpenGL geometriai primitívei a *pontok*, *szakaszok* és *sokszögek*. A geometriai primitíveken kívül az OpenGL *raszterprimitíveket* is kezelni tudja. Raszterprimitívek a *pixelnégyszögek* és a *bittérképek*.

A két típusú primitívet az OpenGL két külön rendszerben tárolja – más-más adatstruktúrák segítségével, más típusú műveleteket tud velük végezni, más koordináta-rendszerben ábrázolja őket. A geometriai, illetve a raszterprimitíves rendszereket a következő ábra foglalja össze:



5.1. ábra. *Raszteres és vektorgrafikus műveletek, az OpenGL grafikus csővezetéke*

A geometriai primitíveket *vertexek* (csúcspontok) definiálják. Egy vertex definiálhat egy pontot, egy szakasz végpontját vagy egy sokszög csúcspontját – minden OpenGL geometriai primitívet meg tudunk határozni a vertexeivel, minden geometriai primitívet vertexek rendezett sorozataként tudunk specifikálni.

Adatábrázolásukat tekintve a vertexek struktúrák, melyek tartalmazzák az illető csúcspont térbeli koordinátáit, színét és egyéb adatait.

5.1.4. Az OpenGL koordináta-rendszerei

Az OpenGL különböző koordináta-rendszereket használ a geometriai objektumok megjelenítésekor, a raszteres objektumok megjelenítésekor, valamint a számítások elvégzése alatt.

A megjelenítéskor az OpenGL a 3 dimenziós Descartes-féle koordináta-rendszert használja, tehát a bázis olyan vektorokból áll, melyek mindegyike merőleges a többire. A koordinátákat a megszokott x, y, z hármassal jelöljük. Az OpenGL jobbsodrású koordináta-rendszert használ, vagyis a (0, 0, 0) pontban van az origó, az x, y tengely pozitív része az origótól jobbra, illetve fölfelé található, a z tengely pozitív része a képernyőből kifelé mutat.

Raszteres objektumok megjelenítésekor az OpenGL az ablak-koordináta-rendszert használja, vagyis a rendszer 2 dimenziós, az origó az ablak bal felső sarkában van, az x tengely jobbra nő, az y tengely pedig lefele nő.

A számítások elvégzésekor az OpenGL a *homogén koordinátákat* használja.

Az OpenGL minden vertexet olyan 3 dimenziós vertexként tárol, melynek 4 koordinátája van: (x, y, z, w). Amikor a vertexeket meg kívánjuk adni,

a következő lehetőségeink vannak: `glVertex2d`, `glVertex2f`, `glVertex2i`, `glVertex2s`, `glVertex3d`, `glVertex3f`, `glVertex3i`, `glVertex3s`, `glVertex4d`, `glVertex4f`, `glVertex4i`, `glVertex4s`, `glVertex2dv`, `glVertex2fv`, `glVertex2iv`, `glVertex2sv`, `glVertex3dv`, `glVertex3fv`, `glVertex3iv`, `glVertex3sv`, `glVertex4dv`, `glVertex4fv`, `glVertex4iv`, valamint `glVertex4sv`.

```
void glVertex{2 3 4}{s i f d}{# v}(T coords);
```

A `glVertex` a parancs neve; a `2`, `3`, `4` azt jelenti, hogy 2 dimenziós, 3 dimenziós vagy homogén koordinátákkal ábrázolt vertexet kívánunk-e megadni (ha a z vagy a w hiányzik, akkor a z-t implicit `0`-nak, a w-t implicit `1`-nek veszi); a `d`, `f`, `i`, `s` azt jelenti, hogy a paramétereket *double, float, integer* vagy *short* formátumban adjuk meg; a `v` pedig azt, hogy a paramétereket nem értékenként külön, hanem egy vektorban adjuk meg.

A

```
void glTexCoord{1 2 3 4}{s i f d}{# v}(T coords);
```

a textúra koordinátákat állítja be, név szerint az s , t , r és q koordinátákat. A `glTexCoord1` parancs az s koordinátát állítja be a meghívott értékre, a t és r koordinátákat `0`-ra, a q-t pedig `1`-re állítja. A `glTexCoord4` mind a négy koordinátát beállítja.

Az érvényes normálist a

```
void glNormal3{b s i f d}{# v}(T coords);
```

paranccsal specifikálhatjuk. Az érvényes normálisok a fényszámításokban, az árnyalásban, illetve a látható felszín meghatározásában vesznek részt. Mivel minden normális egy 3 dimenziós vektor, ezért a `glNormal` parancsnak csak egy változata van.

5.1.5. Az OpenGL színmódjai

Az OpenGL kétféle színmódot használ: az *RGBA* szín módot, illetve a *szín index* módot.

Az RGBA színmódban minden színt négy komponens definiál, a *vörös* (Red), *zöld* (Green), *kék* (Blue), illetve az *alfa* (Alpha) komponens. Az Alpha komponens az átlátszóságot határozza meg: minél nagyobb ez a komponens, annál transzparensebb (átlátszóbb) a szín.

Az OpenGL-ben, RGBA színmódban egy vertex színét a `glColor3b`, `glColor3d`, `glColor3f`, `glColor3i`, `glColor3s`, `glColor3ub`, `glColor3ui`, `glColor3us`, `glColor4b`, `glColor4d`, `glColor4f`, `glColor4i`, `glColor4s`, `glColor4ub`, `glColor4ui`, `glColor4us`, `glColor3bv`, `glColor3dv`, `glColor3fv`, `glColor3iv`, `glColor3sv`, `glColor3ubv`, `glColor3uiv`, `glColor3usv`, `glColor4bv`, `glColor4dv`, `glColor4fv`, `glColor4iv`, `glColor4sv`, `glColor4ubv`, `glColor4uiv`, `glColor4usv` parancsokkal lehet megadni,

```
void glColor{3 4}{b s i f d ub us ui}{# v}(T components);
```

ahol `glColor` a parancs neve; 3 vagy 4 azt jelenti, hogy RGB vagy RGBA – 3 vagy 4 értéket sorolunk fel, vagy tömbben adjuk át ezeket (`v`); a `b`,`d`,`f`,`i`,`s`,`ub`,`ui`,`us` pedig a paraméterek típusát jelentik. Ha *double* vagy *float* a típus, akkor egy színkomponens intenzitását a 0.0–1.0 skálán kell megadni, ha *byte*, akkor 0–255 között, ha *integer*, akkor 0 és *MaxInt* között.

Szín index módban minden színt egy lebegőpontos érték ír le, és minden ilyen lebegőpontos értékhez hozzá van rendelve három 8 bites érték a memóriában, rendre a három szín intenzitása.

Index módban a `glIndexd`, `glIndexf`, `glIndexi`, `glIndexs`, `glIndexub`, `glIndexdv`, `glIndexfv`, `glIndexiv`, `glIndexsv`, `glIndexubv` parancsokkal adhatjuk meg, hogy egy vertex milyen színt vegyen fel az adott palettából (meg kell adni a palettaelem indexét).

```
void glIndex{s i f d ub}{# v}(T index);
```

A kép részletességét, valósághűségét befolyásolja a képernyő felbontásának finomsága, valamint a megjeleníthető színek száma. A *színmélység* azt jelenti, hogy a pixelek színét hány biten ábrázoljuk. 8-bites színmélység esetén 256 különböző szín megjelenítésére van lehetőségünk. 24-bites színmélység esetén egy pixel színét 24 bittel írjuk le, mégpedig úgy, hogy mindhárom színkomponens intenzitását 8 biten ábrázoljuk. Egyes videokártyák rendelkeznek továbbá 32 bites, vagy *true color* színmóddal. A 32 bites színmódban nem tudunk több színt kikeverni, mint a 24 bites színmódban, de teljesítmény szempontjából itt gyorsabb a memória-hozzáférés (viszont van 8 elvesztegetett bit).

5.1.6. Az OpenGL transzformációi

A koordinátákat az OpenGL transzformálja, mielőtt azokat felhasználná egy kép megalkotásában. A vertex-transzformációkat (forgatás, eltolás, skálázás, nyírás) 4×4-es mátrixként reprezentáljuk. Ha v egy homogén vertexet reprezentál, M pedig egy 4×4-es transzformációmátrix, akkor $M*v$ a v vertex képe az M transzformáció után.

A vertex-koordinátákat (amelyeket megadjuk a `glVertex` paranccsal és amelyek a valódi tárgy koordinátái) *tárgykoordináták*nak nevezzük.

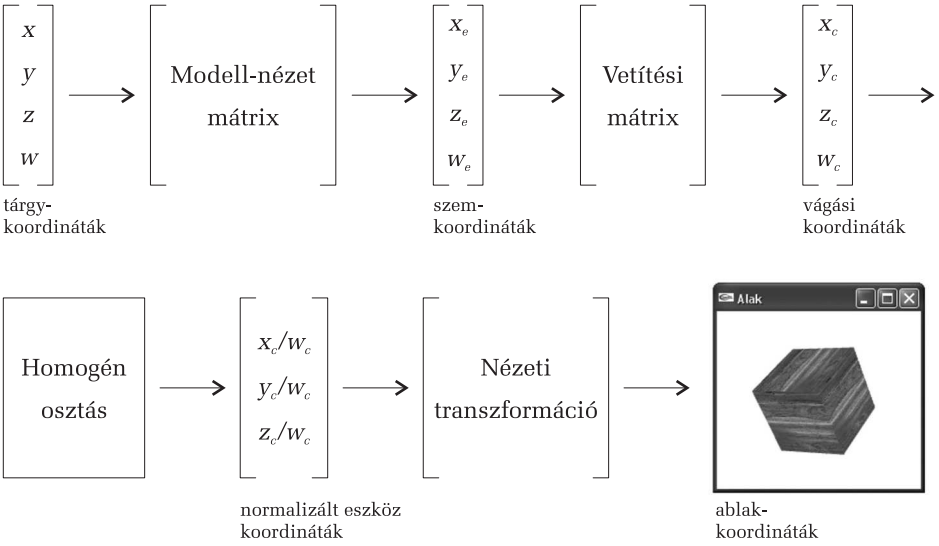
A tárgykoordinátákat a *modell-nézet (ModelView) mátrix* transzformálja *szemkoordinátákká*. A szemkoordinátákból a *vetítési mátrix (Projection)* által lesznek a *vágási koordináták* (clip). Ez a transzformáció egy *látóteret* (viewing volume) definiál (amely párhuzamos vetítés esetén egy téglalap, perspektivikus vetítés esetén pedig egy csonkagúla), úgy, hogy az ezen kívül eső objektumokból vágott objektumok lesznek, így azok a végső képen nem fognak látszani. Ezután a *homogén osztás (perspective division)* következik, és a clip koordináták

normalizált eszköz koordinátákká (normalized device coordinates) transzformálódnak. Az utolsó lépés egy *nézeti transzformáció* (viewport), és létrejönnek az *ablak koordináták*.

Az OpenGL-ben mind a vetítési, mind a modell-nézet mátrix módosítható, illetve beállítható. Az érvényes mátrix módot a

```
void glMatrixMode(GLenum mode);
```

eljárással lehet beállítani, ahol a `mode` a `GL_TEXTURE`, `GL_MODELVIEW`, `GL_COLOR`, illetve `GL_PROJECTION` szimbolikus konstansok valamelyike lehet. A vetítési mátrix a szem koordináta-rendszer és a vetítési (vágási) koordináta-rendszer közötti transzformációt írja le. Ha az érvényes mátrix mód a `GL_MODELVIEW`, akkor a mátrix műveletek a modell-nézet mátrixra vannak hatással. Ezen mátrix segítségével tudunk geometriai transzformációkat (forgatás, eltolás, nagyítás) végezni az objektumokon. Ha `GL_PROJECTION`, akkor a vetítési mátrixra vannak hatással.



5.2. ábra. *Az OpenGL megjelenítési transzformációi*

A két alapvető eljárás, amellyel az aktuális mátrixot inicializálhatjuk, a

```
void glLoadMatrix{f d}(T m[16]);,
```

és a

```
void glMultMatrix{f d}(T m[16]);
```

Mindkét parancs paramétere egy 16 elemű vektor, amely egy 4×4-es mátrixot ábrázol úgy, hogy 16 lebegőpontos számot tárol oszlopfolytonosan.

A `glLoadMatrix` az érvényes mátrixot helyettesíti a megadott mátrixszal. A `glMultMatrix` beszorozza az érvényes mátrixot balról a megadott mátrixszal. A

```
void glLoadIdentity();
```

parancs meghívása után az érvényes mátrix az egységmátrix lesz.

A lineáris transzformációk az érvényes `GL_MODELVIEW` mátrixot módosítják. Ezek a következő eljárásokkal specifikálhatók:

```
void glRotate{f d}(T a , T x , T y , T z);
void glTranslate{f d}(T x , T y , T z);
void glScale{f d}(T x , T y , T z);
```

A `glRotate` eljárásban az `a` adja meg, hogy hány fokkal forgatunk. A kiszámolt mátrix egy óramutató járásával ellentétes forgatást specifikál az (`x`, `y`, `z`)-vel megadott tengely körül.

A `glTranslate` paraméterei az eltolás vektort adják meg: (`x`, `y`, `z`).

A `glScale` eljárás általános skálázást végez az `x`, `y`, `z` tengely mentén.

A *viewport* az az ablakon belüli rész, ahová rajzolunk. A viewport-transzformációval adjuk meg a létrejövő kép méretét. Ez általában maga az ablak, de elképzelhető az is, hogy a viewport az ablaknak csupán egy része. Az aktuális viewportot a

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei
height);
```

eljárással specifikálhatjuk. Az eljárás egy téglalapot definiál az OpenGL ablakba, ahová rajzolni fogunk. Az (`x`, `y`) paraméterek specifikálják a viewport bal alsó sarkát, `width` és `height` pedig a viewport téglalap méretét. Alapértelmezésben a paraméterek a (`0`, `0`, `winWidth`, `winHeight`) értékeket veszik fel, ahol `winWidth` és `winHeight` az ablak méretei.

Az OpenGL-ben párhuzamos (ortografikus) és perspektivikus vetítést is specifikálhatunk, a vetítéseket úgy adjuk meg, hogy megadjuk a látótér nagyságát és alakját. Ami a látótéren belülre esik, az fog a képernyőn látszani.

A párhuzamos vetítés látóterét a

```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,
GLdouble top, GLdouble nearVal, GLdouble farVal);
```

eljárással specifikálhatjuk; `left` és `right` adják meg a bal oldali és jobb oldali függőleges vágósíkok koordinátáit. `bottom` és `top` az alsó és felső vízszintes vágósíkok koordinátái. `nearVal` és `farVal` adják meg a közeli és távoli vágósíkok távolságát a szemtől.

A (`left`, `bottom`, –`nearVal`) és a (`right`, `top`, –`nearVal`) specifikálják a közeli vágósík pontjait, amelyek ráfeszülnek az ablak bal alsó és jobb felső sarkaira (a szem a (`0`, `0`, `0`) pontban van); `farVal` adja meg a távoli vágósík távolságát a szemtől. A közeli vágósík ablakba eső részének bal alsó sarka: (`left`, `bottom`,

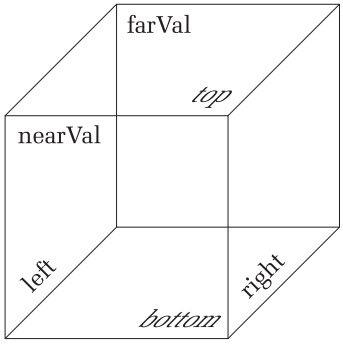
–nearVal), a jobb felső sarka: (right, top, –nearVal). A távoli vágósík ablakba eső részének bal alsó sarka: (left, bottom, farVal), jobb felső sarka: (right, top, farVal).

Ha kétdimenziós objektumokat akarunk rajzolni, akkor a

```
void gluOrtho2D(GLdouble left, GLdouble right,
GLdouble bottom, GLdouble top);
```

eljárással is specifikálhatjuk a vetítési mátrixot. Ekkor a vágási téglalap egyszerűen a (left, bottom, right, top) koordinátákkal megadott téglalap.

A glOrtho (gluOrtho2D) megszorozza az aktuális vetítési mátrixot a specifikált mátrixszal, és ez lesz az új vetítési mátrix.



5.3. ábra. *Párhuzamos vetítés OpenGL-ben*

A perspektivikus vetítés látóterét a

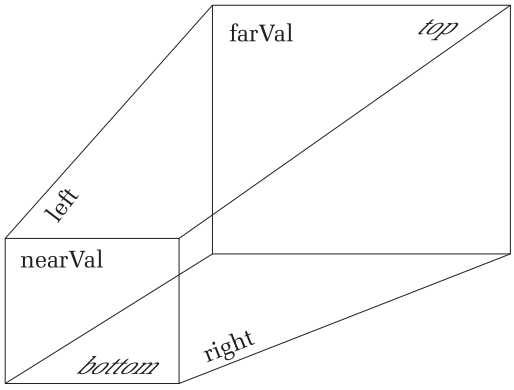
```
void glFrustum(GLdouble left, GLdouble right,
GLdouble bottom, GLdouble top, GLdouble nearVal,
GLdouble farVal);
```

eljárással specifikálhatjuk. A left és right adják meg a bal oldali és jobb oldali függőleges vágósíkok koordinátáit. A bottom és top az alsó és felső vízszintes vágósíkok koordinátái. A nearVal és farVal adják meg a közeli és távoli vágósíkok távolságát a szemtől.

A (left, bottom, –nearVal) és (right, top, –nearVal) specifikálják a közeli vágósík pontjait, amelyek ráfeszülnek az ablak bal alsó és jobb felső sarkaira. A farVal adja meg a távoli vágósík távolságát a szemtől.

Lehetőség van a perspektivikus vetítési mátrix egy szemléletesebb specifikációjára is, a

```
void gluPerspective(GLdouble fovy, GLdouble aspect,
GLdouble nearVal, GLdouble farVal);
```



5.4. ábra. *Perspektivikus vetítés OpenGL-ben*

eljárás segítségével, amely szimmetrikus látóteret specifikál. A `fovy` adja meg a látótér szögét az $x--z$ sík irányában, az `aspect` a vágási téglalap szélességének és magasságának arányát, A `nearVal` és `farVal` pedig a vágósíkok távolságát.

A `glFrustum` (`gluPerspective`) megszorozza az aktuális vetítési mátrixot a specifikált mátrixszal, és ez lesz az új vetítési mátrix. Ennél a vetítésnél a látótér egy csonkagúla.

Minden mátrixmód számára az OpenGL fenntart egy mátrixvermet. Az érvényes mátrix minden módban a verem tetején lévő mátrix. A műveletek:

```
void glPushMatrix();
void glPopMatrix();
```

A `glPushMatrix` és `glPopMatrix` az érvényes mátrixmódnak megfelelő mátrixokkal dolgoznak. Kezdetben minden veremben egy mátrix van, mégpedig az egységmátrix. A kezdő mátrixmód a `GL_MODELVIEW`. A mátrixverem-műveletekkel lehetővé válik az egyes mátrixok elmentése, visszaállítása. A mátrixverem-műveleteket többnyire akkor használjuk, ha összetett objektumok esetében az egyes részeket külön-külön szeretnénk transzformálni.

A primitíveket kirajzolás előtt be kell vágni a látótérbe, amelyet hat sík határoz meg. Az OpenGL-ben lehetőség van ezen kívül más vágósíkok specifikálására is a

```
void glClipPlane(GLenum plane, const GLdouble *equation);
```

eljárással. A `plane` jelzi, hogy melyik vágósíkot állítjuk be. Értéke a `GL_CLIP_PLANE`i valamelyike lehet, ahol i egy `0` és `GL_MAX_CLIP_PLANES-1` közötti érték. Az `equation` egy négy lebegőpontos értéket tartalmazó vektor címe. Ezen értékek egy sík-egyenlet együtthatói objektum koordinátákban: p_1, p_2, p_3, p_4 (ebben a sorrendben).

A `glClipPlane` parancs egy félteret specifikál egy négyegyüttható sík-egyenletet felhasználva. A `glClipPlane` meghívása után a sík-egyenlet transzformálódik a modelview mátrix inverzével, és az eredményként létrejövő egyenlet fog tárolódni. A modellview mátrix további változtatásai nincsenek hatással az eltárolt sík-egyenlet együtthatóira. Ha egy vertex koordinátáit ezen sík-egyenletbe behelyettesítve pozitív vagy nulla értéket kapunk, akkor a vertex belül van, egyébként kívül.

Az így definiált vetítési síkok engedélyezhetők, illetve letilthatók a generikus `glEnable`, illetve `glDisable` utasítással. Mindkét utasítás paramétere a `GL_CLIP_PLANE`i, jelezve azt, hogy melyik síkot akarjuk engedélyezni vagy letiltani.

5.1.7. Az OpenGL mint állapotautomata

Az OpenGL-t állapotautomataként is fel lehet fogni, mivel rendelkezik egy ún. *state*-tel (állapot). Ezen state tartalmazza azokat az érvényes adatokat, amelyek szükségesek a specifikált objektumok leképezéséhez. Minden egyes parancsnak azonnali hatása van az aktuális rajzolási állapotra (*rendering state*).

A grafikus primitívek sok jellemzővel (attribútummal) rendelkezhetnek, így ezek átadása paraméterként körülményes, éppen ezért ezeket összevonjuk egy belső táblázatban, és a hatásuk addig érvényben marad, amíg meg nem változtatjuk. A táblázat tárolja például, hogy a világítás, azon belül mely fényforrások, az élsimítás, az árnyalás, textúrázás stb. engedélyezve van, vagy le van tiltva.

Ezeket az információkat általában egyetlen bit tárolja, ha a bit 1, akkor engedélyezett, ha 0, akkor nem.

Az OpenGL-ben minden felhasznált paraméter rendelkezik egy *iniciális* vagy *alapértelmezett* (*default*) értékkel, pl: az alapértelmezett RGBA szín az (1.0, 1.0, 1.0, 1.0); az alapértelmezett transzformáció és vetítési mátrix pedig az egységmátrix.

Az állapotok értékei `void glGet*` alakú parancsokkal kérdezhetők le. A numerikus értékek lekérdezése:

```
void glGetTv(GLenum pname, GLtype *params);
```

ahol a `T` lehet `Boolean`, `Double`, `Float`, vagy `Integer`.

Logikai értékeket le lehet kérdezni a

```
GLboolean glIsEnabled(GLenum cap);
```

paranccsal, vagy be lehet őket állítani a

```
void glEnable(GLenum cap);
void glDisable(GLenum cap);
```

parancsokkal. Az első 1-re, a második 0-ra állítja a megfelelő bitet.

Az aktuális állapotok elmenthetők az attribútumverembe, illetve visszatölthetők onnan:

```
void glPushAttrib(GLbitfield mask);
```

ahol a `mask` a `GL_ALL_ATTRIB_BITS`, `GL_ENABLE_BIT`, `GL_FOG_BIT`, `GL_LINE_BIT`, `GL_LIGHTING_BIT`, `GL_POINT_BIT`, `GL_POLYGON_BIT`, `GL_TEXTURE_BIT` szimbolikus konstansok értékeit veheti fel. A veremből az attribútumokat a

```
void glPopAttrib(void);
```

paranccsal vehetjük ki.

5.2. Rajzolás OpenGL-ben

5.2.1. Rajzolási műveletek

OpenGL-ben kétféleképpen rajzolhatunk: vagy közvetlenül (azonnal), vagy a rajzolási parancsokat ún. *display-listá*ban (*megjelenítési lista*) tároljuk, és később dolgozzuk fel őket.

Az első rajzolási művelet az ablak törlése, amely nem más, mint az ablakot képviselő téglalap háttérszínnel való kitöltése.

A háttérszín – törlési szín – RGBA értékeit a

```
void glClearColor(GLclampf red, GLclampf green,
GLclampf blue, GLclampf alpha)
```

parancs segítségével állíthatjuk be. A paraméterek a [0.0, 1.0] valós intervallumban ábrázolt RGBA értékek. Az alapértelmezett törlőszín a (0, 0, 0, 0).

Ha színindex módban vagyunk, az aktuális törlőszínt a

```
void glClearIndex(GLfloat c)
```

paranccsal állíthatjuk be.

A bufferek tartalmát a

```
void glClear(GLbitfield mask);
```

paranccsal törölhetjük. A `mask` argumentum egy bitenkénti vagy kombinációja a `GL_COLOR_BUFFER_BIT` (színbuffer – színek kezelése), `GL_DEPTH_BUFFER_BIT` (mélységbuffer – a Z-buffer adatai, mélységteszt), `GL_STENCIL_BUFFER_BIT` (stencilbuffer) és `GL_ACCUM_BUFFER_BIT` (gyűjtőbuffer) szimbolikus konstansoknak.

Azokat a tárterületeket, amelyekben minden pixelhez ugyanannyi adatot tárolunk, *buffer*nek nevezzük.

A *színbuffer* az, amiben rajzolunk. Animáció esetében létezik egy első és egy hátsó színbuffer, sztereoszkópikus ábrázolás esetén létezik egy bal és egy jobb színbuffer is.

Az OpenGL a *mélységbuffer (z-buffer)* algoritmust használja a láthatóság megállapításához, ezért minden pizelhez z értéket is eltárol.

A *stencilbuffer*t arra használjuk, hogy a rajzolást a képernyő bizonyos részeire korlátozzuk.

A *gyűjtőbuffer*t arra használjuk, hogy több képet összegezve állítsunk elő egy végső képet. Így valósítható meg a teljes kép kisimítása (*antialiasing*), a *motion blur* (mozgó objektumok körvonalának elmosása), a mélységélesség.

A bufferek törlési értékei beállíthatók a

```
void glClearDepth(GLclampd depth)
void glClearStencil(GLint s)
void glClearAccum(GLfloat red, GLfloat green,
GLfloat blue, GLfloat alpha)
```

parancsokkal.

Például a

```
1 glClearColor(1.0, 1.0, 1.0, 1.0);
2 glClear(GL{\_}COLOR{\_}BUFFER{\_}BIT);
```

kódrészlet beállítja a törlőszínt fehérre, majd törli vele a színbuffert.

A rajzolás eredménye az általunk kiválasztott színbufferbe vagy bufferekbe kerülnek.

Az aktuális buffert a

```
void glDrawBuffer(GLenum mode);
```

paranccsal állíthatjuk be, ahol a `mode` a következő értékeket veheti fel: `GL_NONE`, `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT`, `GL_BACK_RIGHT`, `GL_FRONT`, `GL_BACK`, `GL_LEFT`, `GL_RIGHT`, `GL_FRONT_AND_BACK`, és `GL_AUX`i, ahol i egy `0` és `GL_AUX_BUFFERS-1` közötti érték.

Pixeladatokat olvashatunk be a

```
void glReadBuffer(GLenum mode);
```

paranccsal kiválasztott bufferből.

A buffereket maszkolhatjuk logikai *és (AND)* művelettel, és így számos trükköt tudunk megvalósítani a következő parancsokkal:

```
void glIndexMask(GLuint mask);
void glColorMask(GLboolean red, GLboolean green,
GLboolean blue, GLboolean alpha);
void glDepthMask(GLboolean flag);
void glStencilMask(GLuint mask);
```

Ha kiadjuk a rajzolási parancsot, az objektum megjelenéséig az OpenGL végrehajtja a transzformációkat, vág, színez, árnyal, textúrát képez le stb., vagyis végigjárja a teljes *megjelenítési lánc*ot (*graphic pipeline*). Ezeket a műveleteket általában más-más hardverelemek hajtják végre. A CPU nem várja meg, hogy ezek végigmenjenek a teljes láncon, hanem folyamatosan adja ki a parancsokat.

Ha azt szeretnénk, hogy a CPU csak akkor adja ki a következő parancsot, ha az előző grafikus parancs már befejeződött, használjuk a

```
void glFinish();
```

parancsot. Ez kikényszeríti a korábban kiadott OpenGL parancsok végrehajtását, és nem adja vissza a vezérlést az előző parancsok teljes végrehajtásának befejezéséig. Használatának hátránya, hogy lelassul a lánc, s így a végrehajtás.

A

```
void glFlush();
```

parancs kikényszeríti a korábban kiadott OpenGL parancsok végrehajtásának megkezdését, és garantálja, hogy ez véges időn belül befejeződik.

A raszterizálás módját a

```
Glint glRenderMode(GLenum mode);
```

parancs segítségével állíthatjuk be. A `mode` értékei `GL_RENDER`, `GL_SELECT`, vagy `GL_FEEDBACK` lehetnek. Az alapértelmezett és a normál mód a `GL_RENDER`. Ekkor a primitíveket raszterizálja a rendszer, a pixelek a frame-bufferbe és onnan a képernyőre kerülnek. `GL_SELECT` vagy `GL_FEEDBACK` esetén semmiféle raszterizálási művelet nem kerül sorra, hanem a primitívek nevei egy általunk megadott bufferbe (`GL_SELECT`) vagy a primitívek adatai a *feedback-bufferbe* kerülnek (`GL_FEEDBACK`). Ekkor a függvény visszatéríti a bufferbe írt bejegyzések számát. `GL_SELECT` mód esetén a

```
void glSelectBuffer(GLsizei size, GLuint *buffer);
```

parancs segítségével lehet az eredménybuffert kiválasztani, `size` a buffer mérete, `buffer` pedig a címe lesz. A primitíveket, raszterizálási műveleteket egyedi névvel tudjuk ellátni. A

```
void glInitNames();
```

parancs segítségével inicializálhatjuk a névvermet, a

```
void glLoadName(GLuint name);
```

segítségével be tudunk tölteni egy nevet a névverem legfelsőbb eleme helyére, a

```
void glPushName(GLuint name);
```

paranccsal a verembe menthetünk egy nevet, a

```
void glPopName();
```

segítségével pedig kivehetjük a legfelsőbb nevet.

GL_FEEDBACK mód esetén a

```
void glFeedbackBuffer(GLsizei size, GLenum type,
GLfloat *buffer);
```

paranccsal állíthatjuk be a visszajelzés módját, ahol size a buffer mérete (ennyi adatot írhatunk bele), type a bufferelemek típusa (minden vertexről ez az információ kerül a bufferbe), ez a következő szimbolikus konstansok valamelyike lehet: GL_2D, GL_3D, GL_3D_COLOR, GL_3D_COLOR_TEXTURE, GL_4D_COLOR_TEXTURE; a buffer pedig a tömb, amibe az adatokat írtuk.

5.2.2. Geometriai objektumok rajzolása

A geometriai objektumokat a vertexek segítségével lehet leírni.

A legtöbb geometriai objektumot glBegin / glEnd párok között specifikáljuk. A specifikációba beletartozik a vertex, textúra és szín koordináták megadása. A parancsok:

```
void glBegin(GLenum mode);
void glEnd();
```

A glBegin parancsnak a következő argumentumai lehetnek: POINTS, LINE_STRIP, LINE_LOOP, LINES, POLYGON, TRIANGLE_STRIP, TRIANGLE_FAN, TRIANGLES, QUAD_STRIP, QUADS, attól függően, hogy milyen geometriai objektumot specifikálunk.

A POINTS (pontok) segítségével független pontokat adhatunk meg. A LINE_STRIP (szakasz sorozat) egy vagy több összekötött szakaszt specifikál a végpontok sorozatának megadásával. Az első vertex specifikálja az első szakasz kezdőpontját, a második vertex az első szakasz végpontját és a második szakasz kezdőpontját stb. A LINE_LOOP (szakasz hurkok) ugyanaz, mint a LINE_STRIP, de az utolsóként specifikált vertexet összeköti az elsőként specifikált vertexszel. A LINES független szakaszokat specifikál. Az elsőként specifikált két vertex határozza meg az első szakaszt, a második két vertex a második szakaszt stb. Ebben az esetben, ha páratlan számú vertexet specifikálunk a glBegin / glEnd pár között, akkor az utolsóként specifikált vertexet az OpenGL nem veszi figyelembe.

A POLYGON sokszöget specifikál. Polygonokat úgy specifikálhatunk, ha specifikáljuk a határvonalát szakaszok sorozataként, ugyanúgy, mint a LINE_LOOP-nál. Az OpenGL csak konvex sokszögek helyes kirajzolását garantálja (konkáv sokszögeket pl. úgy tudunk rajzolni, hogy felbontjuk konvex sokszögekre).

A TRIANGLE_STRIP háromszögek sorozata közös oldalakkal. Ebben az esetben az első három vertex specifikálja az első háromszöget, minden további

vertex egy további háromszöget specifikál úgy, hogy a másik két vertex az előző háromszögből származik. A TRIANGLE_FAN (háromszög legyező): az összes háromszögnek van egy közös csúcsa.

Ezeket az OpenGL úgy valósítja meg, hogy mindig eltárol két vertexet, egy A-t és egy B-t, valamint egy bit mutatót, amely jelzi, hogy melyik eltárolt vertex helyettesítődik az új vertexszel. A TRIANGLE_STRIP hívás után a mutató az A vertexre mutat, minden további vertex átkapcsolja a mutatót, így az első vertex A vertexként, a második vertex B vertexként, a harmadik A vertexként stb. lesz tárolva. Minden vertex, a harmadiktól kezdve, egy háromszöget specifikál az A és B vertexszel.

TRIANGLE_FAN esetében a két vertex közül az A vertex mindig az elsőként specifikált vertex, az összes többi pedig helyettesíti a B vertexet.

A TRIANGLES független háromszögeket definiál.

A QUAD_STRIP (négyszög sorozat) párhuzamos oldalakkal rendelkező téglalapokat hoz létre.

A QUADS független négyszögeket specifikál.

A pontok mérete a

```
void glPointSize(GLfloat size);
```

paranccsal állítható be. Az alapértelmezett érték 1.0 (pontosan egy pixelből áll). Ha a méret 2.0, akkor minden pontot egy 2×2-es négyzet ábrázol. A lebegőpontos számok azt jelentik, hogy a pont átlagos átmérője ennyi lesz.

Lehetőség van pont élsimítás (antialiasing) használatára is, amely letiltható, illetve engedélyezhető, ha az általános glEnable, illetve glDisable parancsot a POINT_SMOOTH szimbolikus konstanssal hívjuk meg. Alapértelmezés szerint a pont élsimítás le van tiltva, ekkor a pixelek négyzet alakú régiója rajzolódik ki. Ha az élsimítás engedélyezett, akkor a pixelek kör alakúak.

A raszterizált szakasz szélességét a

```
void glLineWidth(GLfloat width);
```

paranccsal állíthatjuk be. Az alapértelmezett szélesség 1.0.

A szakaszok élsimítását a glEnable, glDisable parancsokkal lehet szabályozni, ha azokat a GL_LINE_SMOOTH argumentummal hívjuk meg. Ha az élsimítás engedélyezett, akkor valós szélességek is megadhatók, és ekkor a szakasz szélén kirajzolt pixelek intenzitása kisebb lesz, mint a szakasz közepén lévő pixeleké.

A vonal stílust a

```
void glLineStipple(GLint factor, GLushort pattern);
```

paranccsal állíthatjuk be. A pattern 16 bit hosszú bináris sor. Az 1-esek azt jelentik, hogy rajzolni kell a pixelt, a 0-ás pedig azt, hogy nem. A pattern megnyújtható a factor használatával, amely minden bináris részsorozatot megsokszoroz. Például ha a pattern tartalmaz három 1-est egymás után, és a factor 2, akkor a három 1-es helyén hat 1-es lesz. A szakasz stílust engedélyezni, illetve

letiltani lehet a `glEnable`, `glDisable` parancsokkal, ha azokat a `GL_LINE_STIPPLE` szimbolikus konstanssal hívjuk meg.

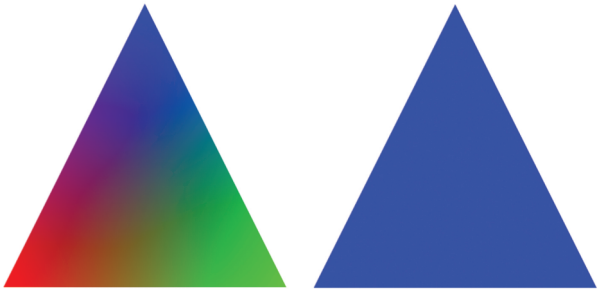
Mivel egy szakaszt két vertex határoz meg, ezért a szakaszhoz tartozó pixelek színe is ezen két vertex színéből származik. Mivel a két vertex színe különböző lehet, a szakasz színe az árnyalási modelltől függ. *Smooth* árnyalási modellben a szakasz egyes pontjainak színe a két különböző színű vertex között átmenetet képez (színinterpoláció). *Flat* árnyalási modellben a szakasz egyszínű lesz, mégpedig olyan színű, amilyen az utolsóként specifikált vertex színe.

Az árnyalási modellt a

```
void glShadeModel(GLenum mode);
```

paranccsal állíthatjuk be. A `mode` a `GL_FLAT`, illetve `GL_SMOOTH` valamelyike lehet.

A színeket nem a primitívekhez, hanem a vertexekhez rendeljük hozzá, így a sokszögek színe is valamiképpen a vertexeinek színéből származik. Az árnyalási modell aszerint dolgozik, hogy a sokszögek egyszínűek (ebben az esetben az adott sokszög színe megegyezik az utolsóként specifikált vertexének színével, kivételt képeznek ezalól a `GL_POLYGON`-nal létrehozott primitívek: itt az elsőként specifikált vertex színe lesz a primitív színe), vagy a belső pontok színe a vertexek színéből számítódik ki interpolációval (*Smooth* árnyalási modellben).



5.5. ábra. Smooth *és* Flat *árnyalási modellek OpenGL-ben*

A látható felszínek meghatározásának alapelve egyszerű: ha egy *pixelt* kirajzolunk, akkor hozzárendelünk egy z értéket (*z-buffer*), amely a pixel megfigyelőtől való távolságát jelzi. Ezután, ha egy új pixelt akarunk rajzolni ugyanarra a helyre, akkor az új pixel z értéke összehasonlítódik az eredeti pixel z értékével. Ha az új pixel z értéke nagyobb, akkor közelebb van a megfigyelőhöz, ezért az eredeti pixelt felülírjuk az új pixellel, egyébként marad az eredeti pixel.

A mélységibeli összehasonlítást engedélyezhetjük, illetve letilthatjuk a `glEnable`, `glDisable` parancsok `GL_DEPTH_TEST` szimbolikus konstanssal való meghívásával.

Egy sokszögnek két oldala van – az elülső és a hátulsó oldal –, és ezért különbözőképpen jelenhet meg a képernyőn, attól függően, hogy melyik oldalát

látjuk. Alapértelmezésben mindkét oldal ugyanúgy rajzolódik ki. Ezen tulajdonságon a

```
void glPolygonMode(GLenum face, GLenum mode);
```

paranccsal lehet változtatni, amely kontrollálja a polygon elülső és hátulsó oldalának rajzolási módját. A `face` paraméter a `GL_FRONT_AND_BACK`, `GL_FRONT`, illetve `GL_BACK`; a `mode` paraméter pedig a `GL_POINT`, `GL_LINE`, illetve `GL_FILL` szimbolikus konstansok valamelyike lehet, aszerint, hogy csak a poligon pontjai, határvonala legyen kirajzolva, vagy ki legyen töltve. Alapértelmezésben a poligon mindkét oldala kitöltve rajzolódik ki. Az elülső oldal alapértelmezésben az, amelynek vertexei az óramutató járásával ellentétes irányban voltak specifikálva.

Ha ellenkezőjére akarjuk változtatni az elülső és hátulsó oldalak meghatározását, akkor ezt a

```
void glFrontFace(GLenum mode);
```

paranccsal tehetjük meg. A `mode` a `GL_CW` és `GL_CCW` szimbolikus konstansok valamelyike, ahol `GL_CW` azt jelenti, hogy az elülső oldal az az oldal lesz, amelynek vertexeit az óramutató járásával megegyező irányban specifikáltunk, `GL_CCW` pedig az ellenkezője.

Ha egy objektumot specifikálunk, akkor előfordulhatnak olyan felszínek, melyek soha nem fognak látszani. Például egy kockát határoló négyzetek belső oldala soha nem látszik. Alapértelmezés szerint az OpenGL azonban minden oldalt kirajzol, tehát a határoló négyzetek belső oldalát is. Ha elkerülnénk a belső oldalak kirajzolását, sok időt spórolnánk meg a kép kirajzolásakor.

A sokszögek elülső vagy hátulsó oldalának figyelmen kívül hagyását *culling*nak (*választás*) nevezzük.

A

```
void glCullFace(GLenum mode);
```

paranccsal specifikálhatjuk, hogy a sokszögek elülső vagy hátulsó oldalát figyelmen kívül hagyjuk a rajzolásnál. A parancs a sokszög meghatározott oldalán letiltja a világítási, árnyalási és színszámítási műveleteket. A `mode` a `GL_FRONT` vagy a `GL_BACK` szimbolikus konstans valamelyike lehet.

A cullingot engedélyezhetjük, illetve letilthatjuk a `glEnable`, `glDisable` paranccsal, ha azt a `GL_CULL_FACE` paraméterrel hívjuk meg.

Alapértelmezés szerint a sokszögek teljesen kitöltöttek. A kitöltési mintát, amelyet egy 32×32-es bináris mátrix reprezentál, a

```
void glPolygonStipple(const GLubyte* mask);
```

paranccsal lehet beállítani.

A sokszög minta engedélyezhető, illetve letiltható a `glEnable`, illetve `glDisable` paranccsal, ha azt a `GL_POLYGON_STIPPLE` szimbolikus konstanssal hívjuk meg.

Sokszögek kisimított rajzolását a `glEnable(GL_POLYGON_SMOOTH)` paranccsal engedélyezhetjük.

A

```
void glRect{s i f d}{# v}(T x1, T y1, T x2, T y2);
```

paranccsal egy (x1, y1) és (x2, y2) pontok által meghatározott téglalapot rajzolhatunk.

5.2.3. Raszteres objektumok rajzolása

OpenGL-ben kétféle raszteres objektum rajzolható: *bittérkép* és *kép*. OpenGL-ben a *bittérkép* pixelenként egyetlen bitben tárol információt (van vagy nincs képpont). és a rendszer maszkként kezeli ezt, a *kép* pedig pixelenként tárolja pl. az RGBA értékeket, és nem maszkként kezeli a rendszer.

Az OpenGL a bittérképeket és képeket mindig az aktuális *raszterpozíció*tól kezdődően rajzolja meg úgy, a kurrens raszterpozíció lesz a bittérkép vagy kép bal alsó sarka.

A kurrens raszterpozíció (x_c, y_c) a

```
void glRasterPos{2 3 4}{s i f d}{# v}(T x, Ty, Tz, Tw);
```

paranccsal adható meg.

Értékét a

```
glGetFloatv(GL_CURRENT_RASTER_POSITION)
```

függvénnyel kérdezhetjük le.

Bittérképek rajzolására a

```
void glBitmap(GLsizei width, GLsizei hight, GLfloat xo,
GLfloat yo, GLfloat xi, GLfloat yi, const GLubyte *bitmap);
```

parancsot használjuk. A `*bitmap` a bittérkép címe, a `width` és a `hight` a bittérkép pixelekben mért szélessége és magassága. Az (x0, y0) párral a bittérkép bal alsó sarkának az eltolását adhatjuk meg, a raszterizálás után a rendszer a kurrens raszterpozíciót (xi, yi)-vel tolja el.

A képek kirajzolása a

```
void glDrawPixels(GLsizei width, GLsizei height,
GLenum format, GLenum type, const GLvoid* pixels);
```

parancs segítségével történik, ahol `format` határozza meg, hogy hogyan kell értelmezni az egyes pixeleket, `type` a pixelek méretét és tárolási módját írja le, `pixels` pedig a kép tömbjére mutató pointer.

Az OpenGL az alábbi formátumokat támogatja:
- RGB képek (RGB hármassal megadva);
- intenzitás képek (szürkeárnyalatos);
- mélység képek (mélységi buffer);
- stencil képek (stencil buffer).

Pixelek kiolvasására szakosodott a

```
void glReadPixels(GLint x, GLint y, GLsizei width,
GLsizei height, GLenum format, GLenum type,
const GLvoid* pixels);
```

amely a képbuffer (x, y) pontjáról olvas ki pixeleket. A pixelek automatikusan konvertálódnak a képbuffer fomátumáról a megadott formátumra és típusra.

A

```
void glCopyPixels(GLint x, GLint y, GLsizei width,
GLsizei height, type GLenum format);
```

paranccsal lehet a képbuffer egy részét átmásolni a képbuffer egy másik területére. A paraméterek a forráspixelek helyét írják le, az eredmény az aktuális raszterpozíció által meghatározott helyre kerül.

Képeket kicsinyíteni, nagyítani a

```
void glPixelsZoom(GLfloat xfactor, GLfloat yfactor);
```

segítségével lehet.

5.3. Megvilágítás és árnyalás

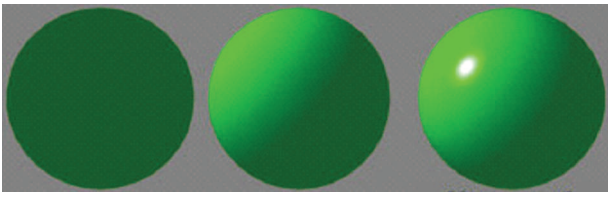
Megvilágítási modellekkel írjuk le a színtér objektumai és a fényforrások kapcsolatát. Az OpenGL csak lokális megvilágítási modellekkel foglalkozik, ami azt jelenti, hogy az objektumok színe, világossága csak az objektumoktól, a fényforrásoktól és a nézőponttól függenek, más objektumoktól nem (nincs fénytörés, tükrözés, árnyékolás, a felület érdességének modellezése).

Ezek a modellek a következők:
- szórt háttérvilágítás (*ambient light*),
- diffúz fényvisszaverődés (*diffuse light*),
- fényvisszaverődés fényes és csillogó felületekről (*specular light*).

A szórt háttérvilágítás modelljében az objektumok egyenletesen, minden irányból kapnak fényt. Hatása a nappali fényviszonyoknak felel meg erősen felhős égbolt esetén. A számítógépes grafikában azért van rá szükség, hogy a felhasználó az ábrázolt jelenet összes objektumának a megvilágítását szabályozhassa. Ebben a modellben nincs fényforrás, az objektumok „saját” fényt bocsátanak ki.

A diffúz fényvisszaverődés a matt felületek jellemzője. Ekkor a megvilágított felület minden irányban ugyanannyi fényt ver vissza.

A sima felületekre általában az a jellemző, hogy rajtuk fényes foltokat is látunk, melyek helye nézőpontunkkal együtt változik. Ezek a felületek bizonyos irányokban visszatükrözik a fényforrásokat. Ekkor a matt felületekre jellemző diffúz és a tökéletesen (ideálisan) tükröző felületekre jellemző visszaverődés közti átmeneti esetet kell modelleznünk.



5.6. ábra. *Zöld gömb csak környezeti, környezeti és diffúz, valamint környezeti, diffúz és tükrözött fényben*

5.3.1. Fényforrások

A megvilágítási modellek úgy valósulnak meg, hogy minden fényforrás három világítási komponensből tevődik össze: *ambiens*, *diffúz* és *spekuláris*. Akárcsak a színeket, a világítási komponenseket is az RGBA értékeivel definiálhatjuk úgy, hogy megadjuk a vörös, zöld és kék intenzitását.

A specifikálható fényforrások számának maximuma implementációfüggő, de legalább nyolc.

Ezeket a fényforrásokat bárhová elhelyezhetjük, például egész közel az objektumokhoz, vagy végtelen messzire. Az előbbi esetben *pozicionális*, az utóbbi esetben pedig *direkcionális fényforrás*ról beszélünk (a negyedik homogén koordináta 0.0). Ezen kívül beállíthatjuk, hogy a fényforrás szűk, fókuszált vagy széles fénysugarat bocsásson ki.

A fényforrások szín, pozíció és irány tulajdonságait a `glLight` paranccsal állíthatjuk be. A parancsnak három paramétere van: az első kijelöli, hogy melyik fényforrás paramétereit szeretnénk beállítani, a második a beállítandó tulajdonságot határozza meg, a harmadik pedig a tulajdonságnak az értékét.

A

```
void glLight{i f}{# v}(GLenum light, GLenum pname, T param);
```

parancs létrehozza a `light`-tal jelölt fényforrást, amely a `GL_LIGHT0`, `GL_LIGHT1`, ..., `GL_LIGHT7` szimbolikus konstansok valamelyike lehet. A `pname` jelöli ki a beállítandó fényforrás-jellemzőt, a `param` pedig az érték, amelyekre a `pname` beállítódik.

5.1. táblázat. *A pname által jelölt paraméterek alapértelmezett értékei*

Paraméternév	*Alapértelmezett érték*	*Jelentés*
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	A fény ambient RGBA intenzitása
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	A fény diffúz RGBA intenzitása
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	A fény spekuláris RGBA intenzitása
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	A fény (x, y, z, w) pozíciója
GL_SPOT_DIRECTION	(0.0, 0.0, −1.0)	A fény (x, y, z) iránya
GL_SPOT_EXPONENT	0.0	Reflektorfény exponens
GL_SPOT_CUTOFF	180.0	Reflektorfény sugárzásának kúpszöge
GL_CONSTANT_ATTENUATION	1.0	Konstans enyhítő faktor
GL_LINEAR_ATTENUATION	0.0	Lineáris enyhítő faktor
GL_QUADRATIC_ATTENUATION	0.0	Négyzetes enyhítő faktor

A GL_DIFFUSE és GL_SPECULAR alapértelmezett értékei csak a GL_LIGHT0-val jelzett fényforrásra érvényesek. A többi fényforrásnál az alapértelmezett érték (0.0, 0.0, 0.0, 1.0) mind a GL_DIFFUSE-ra, mind a GL_SPECULAR-ra.

A fényforrást engedélyezni kell a `glEnable(GL_LIGHT`*i*`)` paranccsal.

Az OpenGL fényforrásoknak három színkomponensekkel megadható paramétere van. A GL_AMBIENT paraméter adja meg a fényben szereplő ambient komponens RGBA intenzitását. A GL_DIFFUSE paraméterrel a diffúz komponens intenzitását specifikálhatjuk, ez jelenti tulajdonképpen a fény színét. A GL_SPECULAR paraméterrel a specular komponens intenzitását adhatjuk meg, ami gyakorlatilag az objektumokon látható *fényes folt (specular highlight)* színét adja meg.

Az OpenGL-ben kétféle fényforrást specifikálhatunk: *pozicionális* és *direkcionális fényforrás*t. A pozicionális fényforrásoknak meghatározott pozíciója van a modelltérben, amely a modell-nézet mátrixszal transzformálódik (a vetítési mátrix nincs hatással a fényforrások pozíciójára), és szemkoordinátákban tárolódik el; ekkor a pozícióvektor w koordinátája 1.0. Direkcionális fényforrások esetén csak a fényforrás irányát adjuk meg, a fényforrás pozícióvektora ekkor

is transzformálódik a modell-nézet mátrixszal; ebben az esetben a pozícióvektor w koordinátája 0.0.

A valós világban a fényforrás távolságával a fény intenzitása csökken. Az OpenGL ezt az intenzitáscsökkenést egy gyengítő faktor bevezetésével valósítja meg, amelyet a megvilágítási egyenletekben használ fel. A gyengítő faktor:

$$fatt = \frac{1}{ek + el \cdot \|VP\| + en \cdot \|VP\|^2}, \text{ ha } w \text{ nem nulla,}$$

ahol, *ek* a konstans gyengítő faktor (`GL_CONSTANT_ATTENUATION`), *el* a lineáris gyengítő faktor (`GL_LINEAR_ATTENUATION`), *en* pedig a négyzetes gyengítő faktor (`GL_QUADRATIC_ATTENUATION`), $||VP||$ a vertex és a fényforrás távolsága (a V vertex színét szeretnénk meghatározni, ha P az egyedüli fényforrás).

Direkcionális fényforrásoknál (w = 0.0), ekkor *fatt* = 1.0.

Az ambiens, diffúz és spekuláris komponensek mindegyikét gyengíti a megadott faktor. Az emissziós (az objektumok saját színe) és a globális ambiens értékekre nincs hatással a gyengülés.

Alapértelmezésben egy létrehozott fényforrás minden irányban sugároz fényt. Lehetőségünk van reflektorszerű pozicionális fényforrások specifikálására is. Ekkor a kibocsátott fény kúp alakot vesz fel. Ahhoz, hogy egy ilyen fényforrást létrehozzunk, meg kell adnunk ennek a kúpnak a szögét a `GL_SPOT_CUTOFF` paraméter beállításával.

Alapértelmezésben ez a kúpszög 180.0^o, vagyis a fényforrás minden irányban sugároz fényt. A kúpszögön kívül meg kell határozni a reflektorfény irányát is (`GL_SPOT_DIRECTION`).

A fénykúp intenzitásának eloszlását a reflektorfény exponensének (`GL_SPOT_EXPONENT`) beállításával specifikálhatjuk, amely alapértelmezésben 0.0. Az exponens segítségével megadhatjuk, hogy a reflektorfény a középvonalhoz közel koncentráltabb legyen, attól távolabb pedig egyre jobban enyhüljön az intenzitása. Az exponens növelésével egyre fókuszáltabb reflektorfényt kapunk.

5.3.2. A megvilágítási modell

A megvilágítási modell paramétereit a következő paranccsal adhatjuk meg:

```
void glLightModel{i f}{# v}(GLenum pname, T param);
```

A beállítandó tulajdonságot a `pname` jelöli ki, a `param` pedig az érték. A `pname` értékei:

`GL_LIGHT_MODEL_LOCAL_VIEWER`: a `param` paraméter egy egész vagy lebegőpontos szám, amely azt adja meg, hogyan számítódjon ki a spekuláris fényvisszaverődés szöge. Alapértelmezett értéke 0.0.

`GL_LIGHT_MODEL_TWO_SIDE`: a param paraméter egy egész vagy lebegőpontos szám, amely megadja, hogy egy- vagy kétoldalas világítási számításokat kell

végezni a sokszögeknél. Nincs hatással a pontok, szakaszok és bitmapek megvilágítására. Ha `params` 0 (vagy 0.0), akkor egyoldalas világítás állítódik be, és csak az elülső oldal paramétereit használja fel az OpenGL a megvilágítási egyenleteknél. Máskülönben kétoldalas megvilágítás specifikálódik. Ebben az esetben a hátulsó sokszögek vertexei a hátulsó anyag paraméterei szerint világítódnak meg, és a normálisaik is módosulnak, mielőtt a világítási egyenlet kiértékelődik. Alapértelmezett értéke 0.0.

Az eljárás vektoros verziója segítségével állíthatjuk be az ambiens modellt:

`GL_LIGHT_MODEL_AMBIENT`: a `params` paraméter egy vektor, amely négy egész vagy lebegőpontos értéket tárol. Ezek az értékek specifikálják a tér szórt háttérvilágításának RGBA intenzitását (globális fény). Az alapértelmezett érték: (0.2, 0.2, 0.2, 1.0).

A megvilágítást engedélyezni kell: `glEnable(GL_LIGHTING)`.

5.3.3. Anyagok tulajdonságai

OpenGL-ben nemcsak a fényforrások tulajdonságait, hanem az objektumok anyagjellemzőit is beállíthatjuk. Egy objektum színe azt határozza meg, hogy a rá érkező fény mely komponensét milyen arányban nyeli el, illetve veri vissza. Ha fényforrásokat is alkalmazunk, akkor ahelyett, hogy azt mondanánk, hogy egy sokszög zöld, azt mondjuk, hogy a sokszög anyaga olyan, mely túlnyomórészt a zöld fényt veri vissza, vagyis specifikálnunk kell az anyag visszaverődési tulajdonságait az ambiens, diffúz és spekuláris fényforrások számára. Az anyagok egy másik tulajdonsága az emissziós érték, amely az anyagok saját fényét jelentik. Az anyag színkomponensei meghatározzák a visszavert fény hányadát, vagyis azt, hogy az egyes komponensekből mennyi verődik vissza.

Az anyagjellemzőket a

```
void glMaterial{i f}{# v}(GLenum face, GLenum pname, T
param);
```

paranccsal állíthatjuk be, ahol `face` a `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK` szimbolikus konstansok valamelyike lehet, attól függően, hogy az objektum elülső, hátulsó vagy mindkét oldalának anyag paramétereit specifikáljuk, a `pname` a specifikálandó paraméter neve, a `param` pedig az értéke.

A diffúz tükröződésnek van a legnagyobb szerepe abban, hogy egy objektumot milyen színűnek érzékelünk. Az érzékelt szín a bejövő fény diffúz komponensének arányától és az objektum és a fényforrás szögétől függ.

Az ambiens tükröződésnek ott van szerepe, ahol az objektumot nem éri közvetlen fény. Az ambiens tükröződésre sincs hatással a nézőpont helyzete. Mivel általában az objektumok diffúz és ambiens tükröződése megegyezik, a kettőt egyszerre specifikáljuk.

5.2. táblázat.

Paraméternév	*Alapértelmezett érték*	*Jelentés*
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Az ambiens RGBA tükröződés
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	A diffúz RGBA tükröződés
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	A spekuláris tükröződés
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	Az emissziós fény intenzitása
GL_SHININESS	0	A spekuláris exponens
GL_AMBIENT_AND_DIFFUSE	(0.8, 0.8, 0,8, 1.0)	Az ambiens és diffúz szín együtt
GL_COLOR_INDEXES	(0, 1, 1)	Ambiens, diffúz és spekuláris indexek

Az objektumok spekuláris tükröződése fényes foltokat eredményez. Függ a nézőponttól is: a tükröződés bizonyos pontokban élesebben jelentkezik. A spekuláris tükröződési hatást a GL_SPECULAR paraméterrel, a foltok (*specular highlight*) méretét és fényességét pedig a GL_SHININESS paraméterrel specifikálhatjuk (magasabb érték kisebb és fényesebb, jobban fókuszált foltot eredményez).

A GL_EMISSION paraméterrel specifikálhatjuk egy objektum saját fényét.

Az anyag paramétereit úgy is specifikálhatjuk, hogy azok kövessék az objektumok azon a színét, amelyet a glColor parancsban megadtunk (*színkövetés – color tracking*). Ezt a

```
void glColorMaterial(GLenum face, GLenum mode);
```

paranccsal tehetjük meg, ahol a face a GL_FRONT, GL_BACK, GL_FRONT_AND_BACK szimbolikus konstansok valamelyike lehet, attól függően, hogy az objektum elülső, hátulsó vagy mindkét oldala a glColor-ban megadott színt kövesse. Alapértelmezett értéke a GL_FRONT_AND_BACK. A mode a GL_EMISSION, GL_AMBIENT, GL_SPECULAR, GL_DIFFUSE, GL_AMBIENT_AND_DIFFUSE konstansok egyike, jelezve azt, hogy melyik anyagjellemzőt határozza meg az érvényes szín. Alapértelmezett érték a GL_AMBIENT_AND_DIFFUSE.

A parancs kiadása után engedélyeznünk kell a színkövetést: glEnable(GL_COLOR_MATERIAL).

Összefoglalva: ahhoz, hogy az objektumok a megvilágítási modell szerint legyenek ábrázolva,

– definiálni kell egy megvilágítási modellt (glLightModel()),

– engedélyezni kell a megvilágítást (`glEnable(GL_LIGHTING)`),
– létre kell hozni egy vagy több fényforrást (`glLight()`),
– be kell kapcsolni a fényforrásokat (`glEnable(GL_LIGHT`*i*`)`),
– anyagtulajdonságot kell megadni (`glMaterial()`).

5.4. Display-listák

A *display-lista* (vagy *megjelenítési lista*) OpenGL parancsok csoportja, amelyet a későbbi végrehajtás céljából tárolunk. Ezt a lehetőséget elsősorban a hálózatban futtatott programok optimális működése érdekében hozták létre (az OpenGL kliens–szerver architektúra alapján működik). A rendszer a grafikus hardver igényeinek megfelelően tárolja a lista parancsait. A parancsok a listában cache-gyorsító szintjén jelennek meg, nem dinamikus adatszerkezet szintjén, így ezek utólag már nem módosíthatók, és hozzá sem férhetünk már a tárolt adatokhoz.

Egy display-listát a `glNewList()`, `glEndList()` parancsok közé írt OpenGL parancsok jelentik. Egyszerre csak egy lista hozható létre.

```
void glNewList(GLuint list, GLenum mode);
void glEndList();
```

A `list` paraméter egy pozitív egész, a lista globális azonosítója. Ha már létezett egy ilyen azonosítójú lista, a rendszer felülírja ezt. A `mode` értéke `GL_COMPILE` vagy `GL_COMPILE_AND_EXECUTE` lehet. Az első esetben a parancsok a listára kerülnek, és a rendszer a megfelelő formátumra konvertálva tárolja őket, de nem futtatja. A második esetben a tárolás után azonnal végre is hajtja a megadott parancsokat.

A display-listákon (mivel ezek a szervergépen tárolódnak) nem szerepelhetnek kliens-függő parancsok, vagyis olyanok, amelyek a kliens konfigurációját adják vissza, a klienstől függnek, vagy olyanok sem, amelyek magukon a listákon operálnak.

A display-listák tartalmazhatnak display-lista hívásokat is, így hierarchiába szervezhetők. Az sem szükséges, hogy a lista meghíváskor már létezzen, egy nem létező lista meghívásának semmiféle következménye nincs.

Egy definiált listát akárhányszor végre tudunk később hajtani, valamint a listák és a parancsok tetszőlegesen kombinálhatók.

A `list` azonosítójú listát azonnal végrehajtja a

```
void glCallList(GLuint list);
```

parancs.

A

```
GLuint glGenList(GLsizei range);
```

parancs `range` darab egymást követő, használaton kívüli display-lista indexet és üres listákat generál, és visszatéríti a lefoglalt tömb első elemét. A

```
GLboolean glIsList(GLuint list);
```

parancs `GL_TRUE` értéket szolgáltat vissza, ha már létezik `list` indexű display-lista.

Egymást követő list indexű display-listákat törölhetünk a

```
void glDeleteLists(GLuint list, GLsizei range);
```

paranccsal, a `list` indextől `range` darabot. Nem létező listák törlésének nincs semmiféle következménye.

Több listát is végrehajthatunk egymás után, ha a display-lista indexeket egy tömbbe tesszük. A

```
void glCallLists(GLsizei n, GLenum type, const GLvoid*
lists);
```

parancs `n` darab listát hajt végre. A listák indexeit úgy számítja ki az OpenGL, hogy a `lists` címen kezdődő értékekhez hozzáadja a

```
void glListBase(GLuint base);
```

paranccsal létrehozott aktuális bázisértékeket.

A `type` paraméterrel az indexek méretét kell megadni.

5.5. Effektusok

5.5.1. Átlátszóság

Az RGBA színkomponensek közül az A (*alpha – alfa*) az átlátszóság modellezésére használható, azt írja elő, hogy az új szín milyen mértékben vegyüljön a régi színnel. Ha az értéke maximális, akkor az új szín tökéletesen fed, ha minimális, akkor a régi szín marad meg. Közbelső értékekre a pixel színe a régi és az új szín valamilyen kombinációja lesz.

Ha színvegyítés engedélyezett, akkor a művelet közvetlenül a pixelbe való írás előtt hajtódik végre. A színvegyítés engedélyezésere meg kell hívnunk a `glEnable(GL_BLEND)` parancsot, és meg kell adnunk, hogy a rendszer a régi és új színösszetevőinek kombináló tényezőit milyen módon számítsa ki. Ezt a

```
void glBlendFunc(GLenum sfactor, GLenum dfactor);
```

parancs segítségével tehetjük meg.

Az `sfactor` értéke határozza meg a forrás RGBA, a `dfactor` pedig a cél RGBA komponenseinek kombinálásához szükséges együtthatókat.

Értékeik a következők lehetnek: GL_ZERO, GL_ONE, GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_DST_COLOR, GL_ONE_MINUS_DST_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA, GL_CONSTANT_COLOR, GL_ONE_MINUS_CONSTANT_COLOR, GL_CONSTANT_ALPHA, GL_ONE_MINUS_CONSTANT_ALPHA, vagy GL_SRC_ALPHA_SATURATE.

Az sfactor alapértelmezett értéke GL_ONE, a dfactor-é pedig GL_ZERO. Értelemszerűen a SRC-s tagok a *source*-ra, a DST-s tagok a *destination*-ra vonatkoznak.

A parancs segítségével egyaránt szabályozhatjuk a háttér és az előtér színét. Szorzófaktorokat definiálunk színkomponensenként a *megjelenítendő (forrás – source)* színkomponensekre (S_r, S_g, S_b, S_a) és a *már pixelen lévő (cél – destination)* színkomponensekre (D_r, D_g, D_b, D_a). Ha a kép színe (R_S, G_S, B_S, A_S), és a háttér színe (R_D, G_D, B_D, A_D), akkor a végső RGBA színkomponensek a következőképpen számítódnak ki:

$$RGBA = (R_S \cdot S_r + R_D \cdot D_r, G_S \cdot S_g + G_D \cdot D_g, \\ B_S \cdot S_b + B_D \cdot D_b, A_S \cdot S_a + A_D \cdot D_a).$$

A felsorolt konstansok jelentése:

A

```
void glAlphaFunc(GLenum func, GLclampf ref);
```

parancs előírhatja, hogy a fenti összetevést az alfa értékek függvényében hogyan használjuk. Az első paraméter az összetevés módját (GL_NEVER – soha, GL_LESS – kisebb, GL_EQUAL – egyenlő, GL_LEQUAL – kisebb vagy egyenlő, GL_GREATER – nagyobb, GL_NOTEQUAL – nem egyenlő, GL_GEQUAL – nagyobb vagy egyenlő, vagy GL_ALWAYS – mindig), a második a küszöb alfa értéket szabályozza. Alapértelmezett a GL_ALWAYS.

Átlátszó objektumok esetén problémák adódhatnak a mélység-teszttel, hisz az átlátszó objektum mögötti objektumnak látszódnia kell, a mélység-teszt pedig már felülírta a z-koordinátát az átlátszó objektum koordinátájával. A problémát úgy tudjuk megoldani, hogy az átlátszó objektum rajzolása előtt a mélységbuffert a glDepthMask(GL_FALSE) parancs segítségével csak olvashatóvá tesszük, így a z-koordináták nem íródnak felül, a távolságokat pedig össze tudja hasonlítani a rendszer. A következő, nem átlátszó objektum rajzolása előtt pedig ismét írhatóvá tesszük a mélységbuffert.

5.5.2. Köd

OpenGL-ben lehetőség van köd modellezésére is, amelynek hatására az objektumok a nézőponttól távolodva fokozatosan elhomályosodnak, majd eltűnnek. Köd segítségével modellezhetők az olyan atmoszférikus hatások, mint a pára, homály, füst, szennyezett levegő.

5.3. táblázat.

GL_ZERO	(0, 0, 0, 0)
GL_ONE	(1, 1, 1, 1)
GL_SRC_COLOR	(R_S, G_S, B_S, A_S)
GL_ONE_MINUS_SRC_COLOR	(1, 1, 1, 1) – (R_S, G_S, B_S, A_S)
GL_DST_COLOR	(R_D, G_D, B_D, A_D)
GL_ONE_MINUS_DST_COLOR	(1, 1, 1, 1) – (R_D, G_D, B_D, A_D)
GL_SRC_ALPHA	(A_S, A_S, A_S, A_S)
GL_ONE_MINUS_SRC_ALPHA	(1, 1, 1, 1) – (A_S, A_S, A_S, A_S)
GL_DST_ALPHA	(A_D, A_D, A_D, A_D)
GL_ONE_MINUS_DST_ALPHA	(1, 1, 1, 1) – (A_D, A_D, A_D, A_D)
GL_CONSTANT_COLOR	Konstans szín
GL_ONE_MINUS_CONSTANT_COLOR	1 – konstans szín
GL_CONSTANT_ALPHA	Konstans alfa
GL_ONE_MINUS_CONSTANT_ALPHA	1 – konstans alfa
GL_SRC_ALPHA_SATURATE	(f, f, f, f), ahol $f = \min(A_S, 1 - A_D)$

A ködöt a rendszer a transzformációk, megvilágítás és textúra-képzés után adja hozzá a képhez.

A köd effektust a `glEnable(GL_FOG)` segítségével engedélyezhetjük. A köd színét az OpenGL a forrás színével vegyíti, a köd kombináló tényező (f) felhasználásával.

Az f tényező meghatározására három lehetőségünk van. A GL_EXP azt jelenti, hogy $f = e^{-density \cdot z}$, a GL_EXP2 szerint $f = e^{-(density \cdot z)^2}$, valamint a GL_LINEAR szerint $f = \frac{end - z}{end - start}$. A *density*, *start*, *end* értékeket a `glFog()` paranccsal adhatjuk meg, alapértelmezés szerint *density* = 1, *start* = 0, *end* = 1. A parancs alakja:

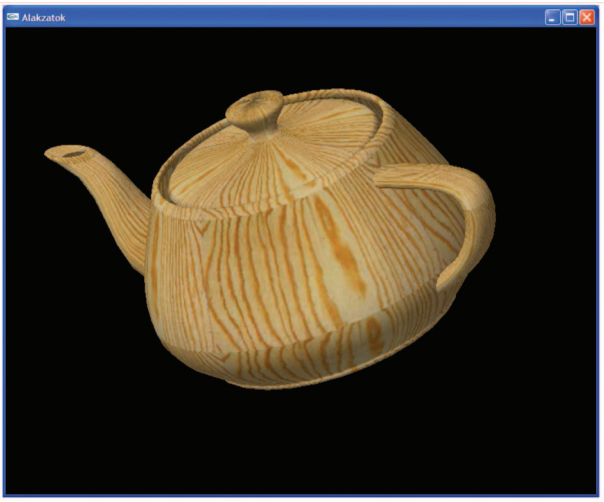
```
void glFog{i f}{# v}(GLenum pname, T param);
```

A pname értéke GL_FOG_MODE, GL_FOG_DENSITY, GL_FOG_START, GL_FOG_END, GL_FOG_COLOR,GL_FOG_INDEX, vagy GL_FOG_COORD_SRC lehet. A GL_FOG_MODE-dal adhatjuk meg az f együttható kiszámítási módját, a következő hárommal a *density*, *start*, *end* értékeket, a GL_FOG_COLOR, valamint a GL_FOG_INDEX segítségével adhatjuk meg a köd színét, a GL_FOG_COORD_SRC pedig a ködbeli távolság megállapítására szolgál.

5.6. Textúrák

A fotorealisztikus képek létrehozásának fontos eszköze a textúrázás. A valós tárgyak felülete mintázott, a mintákat pedig textúrák segítségével tudjuk reprodukálni.

A textúrákat képek segítségével tudjuk az OpenGL-nek átadni, általában bitmap (BMP) vagy JPG képeket szoktunk használni, de bármilyen formátumú kép felhasználható, sőt lehetőségünk van mozgóképek (pl. AVI) átadására is, és ebből textúrát készíteni (például egy szobabelső modellezésekor a tévéképernyőre textúraként feltehetünk egy filmet).



5.7. ábra. *A utahi teáskanna textúrás képe*

A textúra – amint a kép is – geometriailag egy téglalap alakú terület, amelyet tetszőleges alakú poligonokra, poligonhálókra rá tud húzni a rendszer. A ráhelyezést a modelltérben adjuk meg, így a textúrákra is hatnak a transzformációk (pl. perspektíva, forgatás stb.). A ráhelyezést úgy tudjuk megadni, hogy a vertex-koordináták mellett megadjuk a textúra koordinátákat is.

Textúrázáshoz a következő lépéseket kell megtenni:

- engedélyezni kell a textúraleképzést;
- létre kell hozni egy textúraobjektumot és hozzá kell rendelni a textúrát;
- szűrővel meg kell adni, hogy a textúrát hogyan alkalmazza a rendszer;
- meg kell rajzolni a textúrázandó objektumot és meg kell adni a textúrakoordinátákat.

A textúraleképzés engedélyezését a `glEnable()` paranccsal tehetjük meg, ha a `GL_TEXTURE_1D`, `GL_TEXTURE_2D` vagy `GL_TEXTURE_3D` szimbolikus konstansokkal hívjuk meg. Általában 2D textúrákat használunk, az 1D textúrák a

vonalstílusok, a 3D textúrák pedig egymás mögé helyezett 2D textúrák, amelyek segítségével mélységet tudunk érzékeltetni. Ezeket használja a CT (*Computed Tomography*), MRI (*Magnetic Resonance Imaging*), vagy 3D textúrák segítségével jelenítjük meg a kőzetrétegeket, bányákat, barlangokat stb.

Itt a 2D textúrákat mutatjuk be, a parancsokat értelemszerűen kell használni 1D és 3D textúrák esetében (2D helyett 1D vagy 3D írandó).

A textúrát egy pixeles kép alapján készíthetjük el, szükségünk van tehát egy olyan rutinra, amely beolvas pl. egy BMP állományt és feldolgozza azt (kiolvassa és a rendelkezésünkre bocsátja a pixeladatokat).

Kétdimenziós textúrát hoz létre a

```
void glTexImage2D(GLenum target, GLint level,
GLint internalformat, GLsizei width, GLsizei height,
GLint border, GLenum format, GLenum type, const GLvoid
*pixels);
```

parancs. A `target` paraméter értéke `GL_TEXTURE_2D` vagy `GL_PROXY_TEXTURE_2D` lehet. A `level` paramétert akkor kell használni, ha a textúrát több felbontásban is szeretnénk tárolni, különben értéke `0`. Az `internalformat` a textúrában használatos színkomponenseket határozza meg. Értéke `1`, `2`, `3`, `4` vagy a következő szimbolikus konstansok valamelyike lehet: `GL_ALPHA`, `GL_ALPHA4`, `GL_ALPHA8`, `GL_ALPHA12`, `GL_ALPHA16`, `GL_LUMINANCE`, `GL_LUMINANCE4`, `GL_LUMINANCE8`, `GL_LUMINANCE12`, `GL_LUMINANCE16`, `GL_LUMINANCE_ALPHA`, `GL_LUMINANCE4_ALPHA4`, `GL_LUMINANCE6_ALPHA2`, `GL_LUMINANCE8_ALPHA8`, `GL_LUMINANCE12_ALPHA4`, `GL_LUMINANCE12_ALPHA12`, `GL_LUMINANCE16_ALPHA16`, `GL_INTENSITY`, `GL_INTENSITY4`, `GL_INTENSITY8`, `GL_INTENSITY12`, `GL_INTENSITY16`, `GL_R3_G3_B2`, `GL_RGB`, `GL_RGB4`, `GL_RGB5`, `GL_RGB8`, `GL_RGB10`, `GL_RGB12`, `GL_RGB16`, `GL_RGBA`, `GL_RGBA2`, `GL_RGBA4`, `GL_RGB5_A1`, `GL_RGBA8`, `GL_RGB10_A2`, `GL_RGBA12`, vagy `GL_RGBA16`. A `width` és a `height` a textúra szélességét és magasságát jelentik, a `border` a textúra határának szélességét adja meg, értéke `0` vagy `1` lehet. A `width` és a `height` értékek 2^w+2·`border`, 2^h+2·`border` alakúak kell hogy legyenek, ahol w és h természetes számok. A `format` a pixeladatok formátuma (`GL_COLOR_INDEX`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE`, vagy `GL_LUMINANCE_ALPHA`), a `type` a pixeladatok típusa (`GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, vagy `GL_FLOAT`), a `pixels` pedig a képadatokra mutató pointer.

A képbuffer tartalmából is létrehozható textúra a

```
void glCopyTexImage2D(GLenum target, GLint level,
GLenum internalformat, GLint x, GLint y, GLsizei width,
GLsizei height, GLint border);
```

parancs segítségével. A paraméterek ugyanazok, mint az előbb bemutatottak, az `x` és az `y` a kimásolandó pixeltömb bal alsó sarkának a koordinátái.

A textúrákhoz pozitív egész neveket rendelhetünk. A

```
void glGenTextures(GLsizei n, GLuint *textures);
```

paranccsal `n` darab használaton kívüli nevet generálhatunk, ezeket a `textures` tömbben tárolja el az OpenGL.

Egy tetszőleges nevet lekérdezhetünk a

```
GLboolean glIsTexture(GLuint name);
```

paranccsal. A parancs `GL_TRUE`-t térít vissza, ha a `name` egy létező textúranév, különben `GL_FALSE`-t.

A textúraneveket hozzá kell rendelni a textúrákhoz. Ezt tehetjük meg a

```
void glBindTexture(GLenum target, GLuint name);
```

parancs segítségével. A `target` értéke `GL_TEXTURE_1D`, `GL_TEXTURE_2D` vagy `GL_TEXTURE_3D` lehet, a `name` pedig a textúra neve. Amikor először hívjuk meg a parancsot, akkor leköti a nevet egy alapértelmezett textúrával, majd a textúra létrehozása után feltölti a lefoglalt részt a valós adatokkal. Ha nem először hívjuk meg a parancsot, akkor aktuálissá teszi a `name`-mel hivatkozott textúrát az összes többi közül. Ha `0`-val hívjuk meg a parancsot, az alapértelmezett textúra lesz az aktuális.

Textúraneveket a

```
void glDeleteTextures(GLsizei n, const GLuint* names);
```

paranccsal törölhetünk.

OpenGL-ben (a GLU szintjén) lehetőség van *mip-map-technika* megvalósítására is. Az

```
int gluBuild1DMipmaps(GLenum target, GLint components,
GLint width, GLenum format, GLenum type, void *data);
```

illetve

```
int gluBuild2DMipmaps(GLenum target, GLint components,
GLint width, GLint height, GLenum format, GLenum type,
void *data);
```

parancsok segítségével 1D, illetve 2D *mip-maps* textúraképek generálhatók.

A *mip-map-technika*nak a lényege, hogy amikor a betöltött bitmap képből a textúrát generálja a rendszer, az OpenGL több változatot is elkészít belőle (a `gluScaleImage` segítségével), különféle részletességi szinttel, és ezek közül fog válogatni a távolság függvényében.

Képzeljünk el mondjuk egy footballpályát, amely meglehetősen nagy. Az egyik sarkában álló játékos nem fogja látni külön-külön a pálya másik végében az összes fűszálat. Felesleges tehát a nagy felbontású, részletes textúrát használni ott, hisz csak nagyjából látszanak a képek és csak a renderelést lassítják a

méretük miatt. Viszont a hozzá közel lévő részeken élesnek kell lennie a képnek. A mip-map-technika a távolság arányában többféle részletességi szintű textúrát alkalmaz.

A `target` paraméter az első parancs esetében `GL_TEXTURE_1D`, a másodikéban pedig `GL_TEXTURE_2D`. A `components` a színkomponensek számát jelenti (1, 2, 3 vagy 4), a `width`, illetve a `height` a kép szélessége és magassága, a `format` a pixeladatok formátuma (`GL_COLOR_INDEX`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE`, vagy `GL_LUMINANCE_ALPHA`), a `type` a pixeladatok típusa (`GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, vagy `GL_FLOAT`), a `data` pedig a képadatokra mutató pointer.

A téglalap alakú textúrákat az OpenGL ráfeszíti a nem téglalap alakú objektumokra, és ezeket együtt is transzformálja a színtér objektumaival, ezért gyakran megtörténik, hogy egy képpixelnek nem csak egy textúrapixel fog megfelelni. Ezekben az esetekben *filterezni* kell a textúrát. A *textúrafilterezés* lényege, hogy megadhatjuk, kicsinyítésnél (távol van) és nagyításnál (közel került) hogyan viselkedjenek a textúrák, mennyivel kell nagyítani vagy kicsinyíteni a textúrapixeleket, hogy ráférjenek az objektum képének a pixeleire. Ugyancsak ezen paraméterek segítségével adhatjuk meg, hogy a textúra ismétlődjön a felületen (a felületet kitöltjük a textúraképpel úgy, hogy egymás mellé másoljuk többször ugyanazt a képet), vagy csak egyszerűen feszüljön rá a felületre. A

```
void glTexParameter{i f}{# v}(GLenum target,
GLenum pname, T param);
```

parancs segítségével a textúrafilterezéshez szükséges szűrőket adhatjuk meg. A `target` értéke `GL_TEXTURE_1D`, `GL_TEXTURE_2D` vagy `GL_TEXTURE_3D` lehet, a `pname` értékei a `GL_TEXTURE_MIN_FILTER`, `GL_TEXTURE_MAG_FILTER`, `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T` lehetnek.

Az első a kicsinyítéshez, a második a nagyításhoz, a harmadik és negyedik pedig a textúra s és t koordinátája szerinti ismétléshez szükséges. Kicsinyítés esetén összesen hatféle textúrafilterezésre van lehetőségünk (`param`), ezek közül a `GL_NEAREST` a legrosszabb minőségű, de a leggyorsabb, míg a `GL_LINEAR` a legjobb minőségű. A mipmap-technika igyekszik valahol egyensúlyt találni a kettő között. Ennek négy változata van, `GL_xx_MIPMAP_xx` alakban, ahol `xx` vagy `LINEAR` vagy `NEAREST`.

Nagyítás esetén a `GL_NEAREST` és a `GL_LINEAR` közül választhatunk.

A másik kettő esetében pedig a `GL_CLAMP` vagy a `GL_REPEAT` a lehetőségek.

Textúrák használata esetén azt is el tudjuk érni, hogy az objektum színét keverjük a textúra színével. Erre a

```
void glTexEnv{i f}{# v}(GLenum target, GLenum pname,
GLfloat param);
```

parancsot használjuk. A `target` értéke `GL_TEXTURE_ENV` lehet, a `pname` értéke `GL_TEXTURE_ENV_MODE` vagy `GL_TEXTURE_ENV_COLOR` lehet.

Ha az érték `GL_TEXTURE_ENV_MODE`, a `param` értékei `GL_MODULATE`, `GL_DECAL`, `GL_BLEND` vagy `GL_REPLACE` lehetnek, ezek írják elő, hogy a rendszer a textúrát hogyan kombinálja a feldolgozandó pixel színével.

Ha a `pname` értéke `GL_TEXTURE_ENV_COLOR`, akkor a `param` az RGBA komponenseket tartalmazó tömb címe lesz, de ezeket csak akkor fogja használni a rendszer, ha értelmezett a `GL_BLEND` is.

Ha egy objektumot textúrázni akarunk, a vertexek mellett meg kell adnunk a csúcspontok textúrakoordinátáit is, amelyek azt mondják meg, hogy az adott vertexhez melyik textúrapixel tartozik. A textúrának 1, 2, 3 vagy 4 koordinátája lehet, ezeket az s, t, r, q betűkkel jelöljük. A q koordináta értéke általában 1 (homogén koordináta), a többi alapértelmezett értéke 0. Az aktuális textúrakoordinátákat a

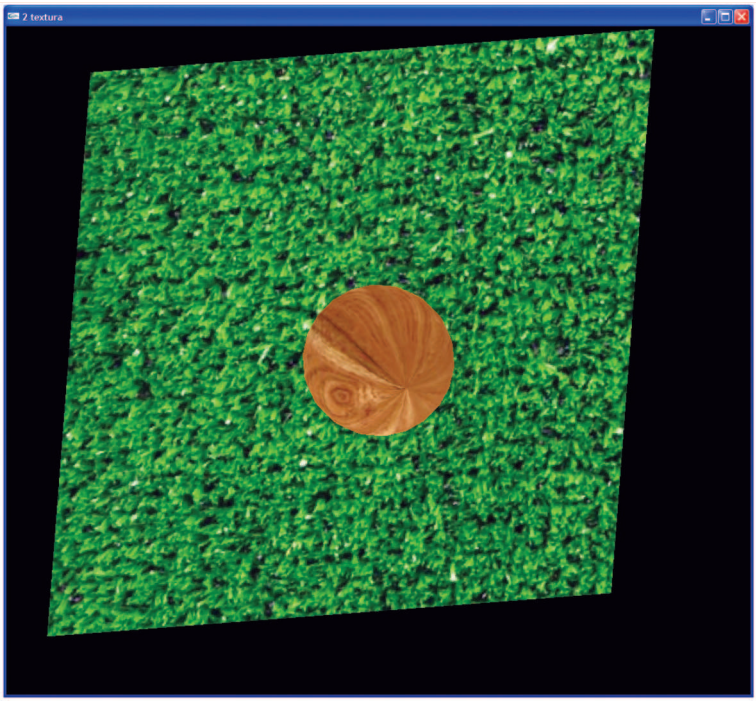
```
void glTexCoord{1 2 3 4}{s i f d}{# v}(T coords);
```

parancs segítségével adhatjuk meg.

A megadott textúrakoordinátákat a rendszer megszorozza a textúramátrixszal.

A következő példaprogram bemutatja, hogyan tudunk betölteni és használni két textúrát ([13] alapján).

```
1  #include <stdlib.h>
2  #include <GL/glut.h>
3  #include "RgbImage.h" // BMP állomány beolvasása
4
5  static GLuint textureName[2]; // a textúrák
6
7  // textúra beolvasása és inicializálása
8  void LoadTextureFromFile(char* filename)
9  {
10   glClearColor (0.0, 0.0, 0.0, 0.0);
11   glShadeModel(GL_FLAT);
12   glEnable(GL_DEPTH_TEST);
13   RgbImage image(filename);
14   glTexParameteri(GL_TEXTURE_2D,
15                   GL_TEXTURE_WRAP_S,
16                   GL_REPEAT);
17   glTexParameteri(GL_TEXTURE_2D,
18                   GL_TEXTURE_WRAP_T,
19                   GL_REPEAT);
20   glTexParameteri(GL_TEXTURE_2D,
21                   GL_TEXTURE_MAG_FILTER,
```



5.8. ábra. *Két textúra használata*

```
22                    GL_NEAREST);
23    glTexParameteri(GL_TEXTURE_2D,
24                    GL_TEXTURE_MIN_FILTER,
25                    GL_NEAREST);
26    gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB,
27                      image.GetNumCols(),
28                      image.GetNumRows(),
29                      GL_RGB, GL_UNSIGNED_BYTE,
30                      image.ImageData());
31  }
32
33  // a textúrák inicializálása
34  void InitTexture(char* filenames[])
35  {
36    glGenTextures(2, textureName);
37    for(int i=0; i<2; ++i)
38    {
39      glBindTexture(GL_TEXTURE_2D, textureName[i]);
```

```
40      LoadTextureFromFile(filenames[i]);
41    }
42  }
43
44  void display()
45  {
46    glClear(GL_COLOR_BUFFER_BIT |
47            GL_DEPTH_BUFFER_BIT);
48    glEnable(GL_TEXTURE_2D);
49    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
50              GL_MODULATE);
51    glRotatef(31, 1, 1, 0);
52    // a textúrák használata
53    glBindTexture(GL_TEXTURE_2D, textureName[0]);
54    glBegin(GL_QUADS);
55      glTexCoord2f(0.0, 0.0);
56      glVertex3f(-2.0, -2.0, -0.5);
57      glTexCoord2f(0.0, 1.0);
58      glVertex3f(-2.0, 2.0, -0.5);
59      glTexCoord2f(1.0, 1.0);
60      glVertex3f(2.0, 2.0, -0.5);
61      glTexCoord2f(1.0, 0.0);
62      glVertex3f(2.0, -2.0, -0.5);
63    glEnd();
64    glBindTexture(GL_TEXTURE_2D, textureName[1]);
65    GLUquadricObj* sphere;
66    sphere = gluNewQuadric();
67    gluQuadricOrientation(sphere, GLU_OUTSIDE);
68    gluQuadricNormals(sphere, GLU_SMOOTH);
69    gluQuadricTexture(sphere, GL_TRUE);
70    gluSphere(sphere, 0.5, 20, 20);
71    gluDeleteQuadric(sphere);
72    glFlush();
73    glDisable(GL_TEXTURE_2D);
74  }
75
76  void ResizeWindow(int w, int h)
77  {
78    float viewWidth = 2.2;
79    float viewHeight = 2.2;
80    glViewport(0, 0, w, h);
81    h = (h==0)?1:h;
```

```
82    w = (w==0)?1:w;
83    glMatrixMode(GL_PROJECTION);
84    glLoadIdentity();
85    if(h < w) viewWidth *= (float)w/(float)h;
86    else viewHeight *= (float)h/(float)w;
87    glOrtho(-viewWidth, viewWidth, -viewHeight,
88              viewHeight, -2.0, 2.0);
89    glMatrixMode(GL_MODELVIEW);
90    glLoadIdentity();
91  }
92
93  void keyboard (unsigned char key, int x, int y)
94  {
95    switch (key)
96    {
97      case 27:
98        exit(0);
99        break;
100     default:
101       break;
102   }
103 }
104
105 char* filenameArray[2] =
106 {
107   "fu.bmp",
108   "fa.bmp",
109 };
110
111 int main(int argc, char** argv)
112 {
113   glutInit(&argc, argv);
114   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB |
115                       GLUT_DEPTH);
116   glutInitWindowSize(500, 400);
117   glutInitWindowPosition(100, 100);
118   glutCreateWindow("2 textura");
119   InitTexture(filenameArray);
120   glutDisplayFunc(display);
121   glutReshapeFunc(ResizeWindow);
122   glutKeyboardFunc(keyboard);
123   glutMainLoop();
```

```
124   return 0;
125 }
```

5.7. A GLU

A *GLU* (*OpenGL Utility Library*) magasabb szintű függvények gyűjteménye, amelynek segítségével könnyebben programozhatjuk az OpenGL lehetőségeit.

A függvényeket a következőképpen csoportosíthatjuk:

- görbékkel és felületekkel kapcsolatos függvények: ezekkel, akárcsak a gluj függvényeivel, egy külön kötetben foglalkozunk;
- hibaüzenet függvény;
- általános transzformációs függvények;
- kvadratikus objektumokat kezelő függvények;
- textúra függvények: lásd az előbbi **Textúrák** című alfejezetet.

5.7.1. Hibaüzenet függvény

A

```
const GLubyte* gluErrorString(GLenum errorCode);
```

függvény segítségével a GLU egy hibaüzenetet állít elő a megadott OpenGL vagy GLU hibakód alapján (`errorCode`).

5.7.2. Általános transzformációs függvények

Az általános transzformációs függvények az OpenGL mátrixaival operálnak, segítségükkel egyszerűbben lehet pl. vetítést specifikálni.

A

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
GLdouble centerx, GLdouble centery, GLdouble centerz,
GLdouble upx, GLdouble upy, GLdouble upz);
```

parancs segítségével a nézőpontot (a szem, kamera helyét) tudjuk megadni. Egy nézeti transzformációt hajt végre. (`eyex`, `eyey`, `eyez`) a szem pozíciója, (`centerx`, `centery`, `centerz`) a referenciapont helyzete, (`upx`, `upy`, `upz`) pedig az irányt adja meg. A parancs kiadása után a rendszer megszorozza az aktuális mátrixot a beállított értékek alapján létrehozott mátrixszal.

A

```
void gluOrtho2D(GLdouble left, GLdouble right,
GLdouble bottom, GLdouble top);
```

parancs segítségével egy egyszerű 2D-s vetítést tudunk megadni.

Perspektivikus vetítést a

```
void gluPerspective(GLdouble fovy, GLdouble aspect,
GLdouble zNear, GLdouble zFar);
```

paranccsal állíthatunk be. A látótér egy szimmetrikus csonkagúla lesz, a `fovy` az y irányú látószög fokban megadva, az `aspect` az x irányú hosszúság/magasság arány, a `zNear` a néző és a közeli vágósík, a `zFar` pedig a néző és a távoli vágósík közötti távolság.

Egy képet tetszőlegesen átméretezhetünk az általános

```
int gluScaleImage(GLenum format, GLint widthin, GLint
heightin,
GLenum typein, const void *datain,
GLint widthout, GLint heightout, GLenum typeout,
void *dataout);
```

parancs használatával. A `format` a pixelformátum, használható értékek: `GL_COLOR_INDEX`, `GL_STENCIL_INDEX`, `GL_DEPTH_COMPONENT`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE` vagy `GL_LUMINANCE_ALPHA`. A `widthin`, `heightin`, `widthout`, `heightout` a bemeneti, illetve az eredmény kép hossza, magassága, a `typein`, `typeout` pedig a bemeneti, illetve eredmény kép típusa. Használható típusok: `GL_UNSIGNED_BYTE`, `GL_BYTE`, `GL_BITMAP`, `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`, vagy `GL_FLOAT`. A `datain`, illetve a `dataout` a bemeneti és az eredmény kép adatokra mutató pointerek.

A

```
void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width,
GLdouble height, GLint viewport[4]);
```

parancs egy kis régiót specifikál az aktuális *Viewport*on belül, vagyis egy olyan vetítési mátrixot hoz létre, amely a rajzolást leszűkíti a (`x`, `y`) középpontú, `width` hosszúságú, `height` magasságú régióra az adott `viewport`on belül. Az adott régiót felhasználhatjuk azon objektumok beazonosítására, amelyek közel vannak a kurzorhoz. A `gluPickMatrix` segítségével jelöljünk ki egy régiót a kurzor körül, majd `glRenderMode` paranccsal állítsuk be a kijelölésmódot és hívjuk meg a rajzolást. A régióban lévő objektumok adatait visszakapjuk a bufferből. A parancs által létrehozott mátrix meg lesz szorozva az aktuális vetítési mátrixszal.

Ha átalakításokat szeretnénk eszközölni az ablak és a színtér objektumainak koordinátái között, az alábbi két parancsot használhatjuk:

```
int gluProject(GLdouble objx, GLdouble objy, GLdouble objz,
const GLdouble modelMatrix[16], const GLdouble
projMatrix[16],
```

```
const GLint viewport[4], GLdouble *winx, GLdouble *winy,
GLdouble *winz);
int gluUnProject(GLdouble winx, GLdouble winy, GLdouble winz,

const GLdouble modelMatrix[16], const GLdouble
projMatrix[16],
const GLint viewport[4], GLdouble *objx, GLdouble *objy,
GLdouble *objz);
```

Az első az objektumkoordinátákat alakítja ablakkoordinátákká, a második pedig ennek a fordított művelete, az ablakkoordinátákat alakítja objektumkoordinátákká. Az ablak- és objektumkoordináták mellett meg kell adni a vetítési és a modell-nézet mátrixot, valamint a viewportot is.

5.7.3. Kvadratikus objektumokat kezelő függvények

GLU-t használva lehetőségünk van kvadratikus objektumok rajzolására (henger, gömb, korong, korongszelet) és textúrázására, a következő parancsok segítségével:

```
GLUquadricObj* gluNewQuadric();
```

létrehoz egy új kvadratikus objektumot (konstruktor) és visszatérít egy, az objektumra mutató pointert.

```
void gluDeleteQuadric(GLUquadricObj *state);
```

megszünteti a `state` mutatóval referált kvadratikus objektumot (destruktor).

```
void gluQuadricCallback(GLUquadricObj *qobj, GLenum which,
void (*fn));
```

egy callback függvényt hozzárendel a kvadratikus objektumhoz.

```
void gluQuadricDrawStyle( GLUquadricObj *quadObject,
GLenum drawStyle);
```

a kvadratikus objektum rajzolási módját állítja be. A `drawStyle` paraméter értéke `GLU_FILL` (sokszögekkel megrajzolt objektum), `GLU_LINE` (vonalas objektum), `GLU_SILHOUETTE` (csak a látható élvonalakat rajzolja meg), vagy `GLU_POINT` (pontok halmaza) lehet.

```
void gluQuadricNormals(GLUquadricObj *quadObject,
GLenum normals);
```

a kvadratikus objektumok normálisait állítja be. A `normals` a `GLU_NONE`, `GLU_FLAT`, vagy `GLU_SMOOTH` értéket veheti fel.

```
void gluQuadricOrientation(GLUquadricObj *quadObject,
```

```
GLenum orientation);
```

a normálisok kifele vagy befele mutató irányát állítja be a GLU_OUTSIDE vagy GLU_INSIDE konstansokkal.

```
void gluQuadricTexture(GLUquadricObj *quadObject,
GLboolean textureCoords);
```

Megmondja, hogy a rendszer generáljon-e (GL_TRUE) vagy sem (GL_FALSE) textúrakoordinátákat.

Az effektív kvadratikus testek a következők:

```
void gluCylinder(GLUquadricObj *qobj, GLdouble baseRadius,
GLdouble topRadius, GLdouble height,
GLint slices, GLint stacks);
```

ahol:

- qobj a gluNewQuadric által létrehozott kvadratikus objektum
- baseRadius a $z = 0$ koordinátában a henger alap-sugara
- topRadius a $z =$ height koordinátában a henger (vagy csonkakúp, kúp) sugara
- height a test magassága
- slices a z tengely körüli felosztások száma
- stacks a z tengely mentén a felosztások száma

```
void gluDisk(GLUquadricObj *qobj, GLdouble innerRadius,
GLdouble outerRadius, GLint slices, GLint loops);
```

ahol:

- qobj a gluNewQuadric által létrehozott kvadratikus objektum
- innerRadius a korong belső sugara
- outerRadius a korong külső sugara
- slices a z tengely körüli felosztások száma
- loops a koncentrikus körök száma

```
void gluPartialDisk(GLUquadricObj *qobj, GLdouble
innerRadius,
GLdouble outerRadius, GLint slices, GLint loops,
GLdouble startAngle, GLdouble sweepAngle);
```

ahol:

- qobj a gluNewQuadric által létrehozott kvadratikus objektum
- innerRadius a korong belső sugara
- outerRadius a korong külső sugara
- slices a z tengely körüli felosztások száma
- loops a koncentrikus körök száma
- startAngle a korongszelet kezdő szöge fokban mérve

– `sweepAngle` a korongszelet szöge fokban mérve

```
void gluSphere(GLUquadricObj *qobj, GLdouble radius,
GLint slices, GLint stacks);
```

– `qobj` a `gluNewQuadric` által létrehozott kvadratikus objektum
– `radius` a gömb sugara
– `slices` a z tengely körüli felosztások száma
– `stacks` a z tengely mentén a felosztások száma

A következő programrészlet a kvadratikus objektumok használatát mutatja be:

```
1 GLUquadricObj* sphere;
2 sphere = gluNewQuadric();
3 gluQuadricOrientation(sphere, GLU_OUTSIDE);
4 gluQuadricNormals(sphere, GLU_SMOOTH);
5 gluQuadricTexture(sphere, GL_FALSE);
6 gluSphere(sphere, 3, 20, 20);
7 gluDeleteQuadric(sphere);
```

5.8. A GLUT

A platformfüggetlen OpenGL alapból nem tartalmazza az ablakozó rendszert, hisz minden operációs rendszer, minden architektúra másképp oldja meg ezt.

A *GLUT* (*OpenGL Utility Toolkit*) az OpenGL kibővítése, amely már tartalmazza az OpenGL ablakok létrehozásához szükséges eljárásokat, így néhány sor megírásával létre tudunk hozni egy OpenGL renderelésre alkalmas ablakot (OpenGL-felület).

A GLUT saját eseménykezelő-rendszerrel is rendelkezik, és olyan rutinokat is tartalmaz, amelyekkel karaktereket és magasabb szintű geometriai objektumokat, mint például gömböket, kúpokat, ikozaédereket tudunk megjeleníteni.

A GLUT eseménykezelő rendszere hasonlít a Windows eseménykezelő rendszeréhez. Bizonyos eseményekhez (pl. egy billentyű vagy egy egérgomb lenyomása) *callback* rutinokat rendelhetünk. Ezután egy *main loop*ba (fő esemény-hurok) lépünk, majd ha egy esemény történik a hurokban, akkor az ezen eseményhez rendelt *callback* rutin végrehajtódik. (Windows terminológiában *callback rutin*nak nevezzük azokat az eljárásokat, függvényeket, amelyek paraméterként átadhatók más eljárásoknak, függvényeknek – pl. eseményfigyelőknek –, és ezek meg tudják hívni, végre tudják hajtani a paraméterként kapott rutint.)

A GLUT ablak- és képernyő-koordináták pixelekben vannak kifejezve. A képernyő vagy ablak bal felső koordinátája $(0, 0)$. Az x koordináta jobbra haladva nő, az y koordináta pedig lefelé; ez nem egyezik meg az OpenGL 3D koordináta-rendszerével, de megegyezik a legelterjedtebb ablakozó rendszerek koordináta-rendszerével (az OpenGL valós 3D Descartes-féle koordináta-rendszere helyett itt megkapjuk a pixel alapú 2D ablakkoordinátákat).

Ha használni óhajtjuk a GLUT nyújtotta lehetőségeket, inkludolni kell a *glut.h* könyvtárat, pl.: `#include<GL\glut.h>`. Telepítve kell még, hogy legyen a *glut32.lib* (az exportbekötő könyvtárállomány), illetve a *glut32.dll* (a futásidejű dinamikus csatolású könyvtár).

A GLUT-függvények nevei a **glut** előtaggal kezdődnek.

5.8.1. GLUT ablakkezelés

Az OpenGL (GLUT) ablak létrehozásához meg kell adnunk annak tulajdonságait. Ehhez a következő eljárásokat használhatjuk:

`void glutInit (int argc , char **argv );`

A **glutInit** eljárást minden más GLUT eljárás előtt kell meghívni, mert ez inicializálja a GLUT könyvtárat. Az eljárást nem kötelező használni, és csak akkor használhatjuk, ha egyszerű szöveges alkalmazásként hoztuk létre az OpenGL-alkalmazásunkat (*File / New.. / Projects / Win32 Console Application*), ekkor a **glutInit** eljárás paraméterei megegyeznek a main függvény paramétereivel, és át tudják venni a parancssor argumentumait.

`void glutInitDisplayMode (unsigned int mode);`

`void glutInitDisplayString (char *string);`

A **glutInitDisplayMode** a képernyőmódot specifikálja (egyszeresen vagy kétszeresen pufferelt ablak, RGBA vagy színindex mód stb.). Például a `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)` egy egyszeresen pufferelt, RGB módban lévő ablakot specifikál. A `glutInitDisplayMode` eljárásnak meglehetősen sok lehetséges paramétere van (`GLUT_DOUBLE`, `GLUT_INDEX`, `GLUT_STEREO`, ...), de egyszerűbb programok írásához nekünk ezek közül csak néhányra lesz szükségünk. A kétszeresen pufferelt ablak (`GLUT_DOUBLE`) a jó minőségű animációnál szükséges. A beállításokat stringként is megadhatjuk, ha a második változatot használjuk.

`void glutInitWindowSize(int width, int height)`

Az ablak méreteit adhatjuk meg pixelekben; `width`: szélesség, `height`: magasság. Például a `glutInitWindowSize(640, 480)` eljárás egy 640×480 pixel méretű ablakot specifikál.

`void glutInitWindowPosition(int x, int y);`

Az ablak bal felső sarkának x és y pozíciója. Például a `glutInitWindowPosition(50, 50)` eljáráshívás hatására az ablak bal felső koordinátái az (50, 50) pontba kerülnek.

`int` **`glutCreateWindow`**`(char *name);`
Létrehoz és megnyit egy ablakot az előző eljárásokkal megadott tulajdonságokkal. Ha az ablakozó rendszer lehetővé teszi, akkor a name megjelenik az ablak fejlécén. A visszatérési érték egy egész, amely az ablak azonosítója. Ezt az értéket használhatjuk fel az ablak kontrollálására. Például a `glutCreateWindow("próba")` egy *próba* névvel ellátott ablakot hoz létre.

`void` **`glutPostRedisplay`**`(void);`
Az érvényes ablak frissítését eredményezi. A `glutPostRedisplay` eljárásra többnyire az animációkészítésnél lesz szükségünk, ugyanis ezzel az eljárással tudjuk az ablakot periodikusan frissíteni.

`int` **`glutLayerGet`**`(GLenum info);`
Az aktuális ablakhoz tartozó rétegek (*layer*) és felső burkolat (*overlay*) információit adja meg. Az `info` paraméter lehetséges értékei: `GLUT_OVERLAY_POSSIBLE`, `GLUT_LAYER_IN_USE`, `GLUT_HAS_OVERLAY`, `GLUT_TRANSPARENT_INDEX`, `GLUT_NORMAL_DAMAGED`, `GLUT_OVERLAY_DAMAGED`.

`void` **`glutEstablisOverlay`**`();`
Az aktuális ablakhoz hozzárendel egy felső burkolót (*overlay*).

`void` **`glutUseLayer`**`(GLenum layer);`
A réteget (*layer*) vált. A `layer` paraméter értékei: `GLUT_NORMAL` vagy `GLUT_OVERLAY`.

`void` **`glutShowOverlay`**`();`
`void` **`glutHideOverlay`**`();`
Az aktuális ablak-overlayt teszi láthatóvá vagy rejti el.

`void` **`glutRemoveOverlay`**`();`
Az aktuális ablak-overlayt semmisíti meg.

`void` **`glutPostOverlayRedisplay`**`(void);`
`void` **`glutPostOverlayWindowRedisplay`**`(int win);`
Az aktuális vagy a megadott ablak-overlay frissítését eredményezi.

`void` **`glutSwapBuffers`**`();`
Ha az aktuális ablak kétszeresen pufferelt (pl. animációk esetén – `GLUT_DOUBLE`), megcseréli egymással a puffereket.

`void` **`glutSetCursor`**`(int cursor);`
A GLUT lehetőséget biztosít a megjelenő egérkurzor beállítására is. A kurzor kinézetét (nyíl, kereszt, homokóra, kéz stb.) szimbolikus konstansokkal adhatjuk meg: `GLUT_CURSOR_INHERIT`, `GLUT_CURSOR_NONE`, `GLUT_CURSOR_RIGHT_ARROW`, `GLUT_CURSOR_INFO`, `GLUT_CURSOR_CYCLE` stb.

`void` **`glutFullScreen`**`(void);`
Teljes képernyőssé teszi az aktuális ablakot.

`int` **`glutCreateSubWindow`**`(int win, int x, int y, int width, int height);`
Létrehoz egy, a megadott win ablakhoz kötődő alablakot, az x, y koordinátákkal, `width` szélességgel, `height` magassággal.

`void` **`glutSetWindow`**`(int win);`
Aktuálissá (fókuszálttá) teszi a win azonosítóval rendelkező ablakot.

`int` **`glutGetWindow`**`(void);`
Visszatéríti az aktuális ablak azonosítóját (egész számú kódját).
`void` **`glutDestroyWindow`**`(int win);`
Megsemmisíti a megadott azonosítóval rendelkező ablakot.
`void` **`glutPositionWindow`**`(int x, int y);`
Megváltoztatja az aktuális ablak pozícióját a képernyőn.
`void` **`glutReshapeWindow`**`(int width, int height);`
Megváltoztatja az aktuális ablak méretét.
`void` **`glutShowWindow`**`(void);`
Megjeleníti az aktuális ablakot.
`void` **`glutHideWindow`**`(void);`
Eltünteti (láthatatlanná teszi) az aktuális ablakot.
`void` **`glutIconifyWindow`**`(void);`
Ikon-állapotba hozza az aktuális ablakot.
`void` **`glutSetWindowTitle`**`(char *name);`
Beállítja az aktuális ablak címzónájának szövegét.
`void` **`glutSetIconTitle`**`(char *name);`
Beállítja az ikon címzónájának szövegét.
`void` **`glutPopWindow`**`(void);`
Az ablak-veremből kiveszi az aktuális ablakot.
`void` **`glutPushWindow`**`(void);`
Az ablak-verembe menti az aktuális ablakot.
`void` **`glutWarpPointer`**`(int x, int y);`
A megadott koordinátájú pontra helyezi a kurzormutatót.

5.8.2. A GLUT és a színek, videofelbontások, játékmódok

`int` **`glutVideoResizeGet`**`(GLenum param);`
Információt szolgáltat az aktuális videofelbontásról.
`int` **`glutEnterGameMode`**`();`
`void` **`glutLeaveGameMode`**`();`
Belép, vagy elhagyja a GLUT játék üzemmódját.
`int` **`glutGameModeGet`**`(GLenum info);`
Információt szolgáltat az aktuális játék üzemmódról.
`void` **`glutGameModeString`**`(const char *string);`
A játék üzemmód konfigurációját állítja be a megadott string alapján.
`GLfloat` **`glutGetColor`**`(int cell, int component);`
`void` **`glutSetColor`**`(int cell, GLfloat red, GLfloat green, GLfloat blue);`
Az aktuális ablak palettájának színindexét kérdezi le vagy állítja be a megadott RGB értékek alapján.
`void` **`glutCopyColormap`**`(int win);`
A megadott ablak palettáját lemásolja az aktuális ablakra.

5.8.3. GLUT eseménykezelés

Miután létrehoztuk a GLUT ablakot, de még nem léptünk be a fő eseményhurokba (nem hívtuk meg a `glutMainLoop();` függvényt), kijelölhetjük, regisztrálhatjuk az eseményvezényléshez szükséges *callback* függvényeket. Ezt a következő GLUT rutinokkal tehetjük meg:

`void` **`glutWindowStatusFunc`**`(void (*`*`func`*`)(int state));`
: Az ablak-státusz *callback*et állítja be. A state paraméter lehetséges értékei: `GLUT_HIDDEN`, `GLUT_FULLY_RETAINED`, `GLUT_PARTIALLY_RETAINED`, vagy `GLUT_FULLY_COVERED`.

`void` **`glutDisplayFunc`**`(void(*func)(void));`
: Azt a függvényt specifikálja, amelyet akkor kell meghívni, ha az ablak tartalmát újra akarjuk rajzoltatni. Ide írjuk be a rajzoló kódot, az OpenGL programunkat.

`void` **`glutOverlayDisplayFunc`**`(void(*func)(void));`
: Az *overlay callback*-et definiálja.

`void` **`glutReshapeFunc`**`(void(*func)(int width, int height));`
: Azt a függvényt specifikálja, amelyet akkor kell meghívni, ha az ablak mérete vagy pozíciója megváltozik. A `func` argumentum egy függvényre mutat, amelynek két paramétere van, az ablak új szélessége és magassága. Ha a `glutReshapeFunc` függvényt nem hívjuk meg vagy `NULL` az argumentuma, akkor egy alapértelmezett függvény hívódik meg, amely meghívja a `glViewport(0, 0, width, height)` függvényt.

`void` **`glutVisibilityFunc`**`(void (*func)(int state));`
: Azt a függvényt specifikálja, amely akkor hívódik meg, ha láthatóvá vagy nem láthatóvá válik az ablak. A `state` parameter értékei: `GLUT_VISIBLE` vagy lehetnek.

`void` **`glutKeyboardFunc`**`(void(*func)(unsigned char key, int x, int y);`
: Azt a függvényt specifikálja, melyet egy billentyű lenyomásakor kell meghívni. key egy ASCII karakter. Az `x` és `y` paraméterek az egér pozícióját jelzik a billentyű lenyomásakor (ablak relatív koordinátákban). Például megírhatjuk a következő függvényt:

```
1  void keyboard(unsigned char key, int x, int y)
2  {                        //billentyűkezelés
3    switch(key)
4    {
5      case 27:             // ha escape-et nyomtunk
6        exit(0);           // lépjen ki a programból
7        break;
8    }
9  }
```

majd ezt a függvényt átadjuk paraméterként a `glutKeyboardFunc` eljárásnak a következőképpen: `glutKeyboardFunc(keyboard);`

Ekkor a programunkból az *Esc* billentyű lenyomásakor léphetünk ki.

`void` **`glutKeyboardUpFunc`**`(void(*func)(unsigned char key, int x, int y);`

Azt a callback függvényt specifikálja, amely akkor hívódik meg, ha felengedtünk egy lenyomott billentyűt.

`void` **`glutSpecialFunc`**`(void (*func)(int key, int x, int y));`

A billentyűzeten olyan billentyűk is vannak, amelyek nem egy, hanem két karakterkódot generálnak. Ilyenek például a nyíl billentyűk vagy az F1–F12 funkcióbillentyűk. Ezek kezelését nem tudja megoldani a `glutKeyboardFunc`, hanem önálló kezelőfüggvénnyel rendelkeznek, amelyet a `glutSpecialFunc` segítségével lehet regisztrálni. A billentyűk kódjait itt már egész konstansként kell megadni, nem karakterként: `GLUT_KEY_F1`, `GLUT_KEY_LEFT`, `GLUT_KEY_PAGE_UP`, `GLUT_KEY_HOME` stb.

`void` **`glutSpecialUpFunc`**`(void (*func)(int key, int x, int y));`

Azt a *callback* függvényt specifikálja, amely akkor hívódik meg, ha felengedtünk egy lenyomott speciális billentyűt.

`int` **`glutSetKeyRepeat`**`(int repeatMode);`

`void` **`glutIgnoreKeyRepeat`**`(int ignore);`

A billentyűismétlést állítja be vagy hagyja figyelmen kívül. A `repeatMode` lehetséges értékei: `GLUT_KEY_REPEAT_OFF`, `GLUT_KEY_REPEAT_ON`, `GLUT_KEY_REPEAT_DEFAULT`.

`int` **`glutGetModifiers`**`();`

Billentyűzet *callback*-hívásokkor a *Shift*, *Ctrl*, *Alt* billentyűk státuszát téríti vissza.

`void` **`glutMouseFunc`**`(void(*func)(int button, int state, int x, int y);`

Azt a függvényt specifikálja, amely egy egérgomb lenyomásakor, illetve elengedésekor hívódik meg. A button *callback* paraméter a `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, illetve a `GLUT_RIGHT_BUTTON` egyike. A state *callback* paraméter a `GLUT_UP` és a `GLUT_DOWN` szimbolikus konstansok egyike. Az `x` és `y` *callback* paraméterek az egér pozícióját jelzik az egér esemény megtörténtekor (ablak relatív koordinátákban).

`void` **`glutMotionFunc`**`(void (*func)(int x, int y));`

`void` **`glutPassiveMotionFunc`**`(void (*func)(int x, int y));`

A gomblenyomással, illetve gomblenyomás nélküli egérmozgatás eseményfüggvényeit állíthatjuk be. Az `x` és `y` paraméterek az egérkurzor koordinátái az ablak koordináta-rendszerében.

`void` **`glutEntryFunc`**`(void (*func)(int state));`

Az a függvényt állíthatjuk be, amely akkor hívódik meg, ha az egér elhagyta vagy belépett az ablak területére (az egér az ablak fölött van-e vagy sem). A state paraméter értékei: `GLUT_LEFT`, illetve `GLUT_ENTERED` lehetnek.

`void` **`glutJoystickFunc`**`(void (*func)(unsigned int buttonMask, int x, int y, int z), int pollInterval);`
Beállítja a botkormány (*joystick*) *callback* függvényét. A `buttonMask` paraméter értékei `GLUT_JOYSTICK_BUTTON_A`, `GLUT_JOYSTICK_BUTTON_B`, `GLUT_JOYSTICK_BUTTON_C` vagy `GLUT_JOYSTICK_BUTTON_D` lehetnek, `x`, `y` a koordináták, `pollInterval` pedig a botkormány milliszekundumokban megadott érzékenységi intervalluma.

`void` **`glutForceJoystickFunc`**`();`
Meghívja a botkormány *callback*et.

`void` **`glutTabletButtonFunc`**`(void (*func)(int button, int state, int x, int y));`

`void` **`glutTabletMotionFunc`**`(void (*func)(int x, int y));`
A rajzolótábla (*Tablet*) eseményeit lekezelő függvényeket adhatjuk meg: mozgatás, illetve gombnyomás esetén.

`void` **`glutSpaceballMotionFunc`**`(void (*func)(int x, int y, int z));`

`void` **`glutSpaceballRotateFunc`**`(void (*func)(int x, int y, int z));`

`void` **`glutSpaceballButtonFunc`**`(void (*func)(int button, int state));`
A fenti függvények segítségével a *SpaceBall* (az egérhez hasonló beviteli eszköz, egy talapzatra helyezett gömb, amit kézzel forgatni lehet, és így egyből be lehet vinni az `x`, `y`, `z` 3D koordinátákat) eseményeit lekezelő függvényeket adhatjuk meg: mozgatás, forgatás, gombnyomás esetén. A paraméterek mindhárom esetben az x, y és z valós 3D koordináták.

5.9. ábra. *Rajzolótábla (Tablet) és SpaceBall*

`void` **`glutButtonBoxFunc`**`(void (*func)(int button, int state));`

`void` **`glutDialsFunc`**`(void (*func)(int dial, int value));`
Az opcionálisan felszerelhető numerikus billentyűzet eseménykezelőit regisztrálja.

`int` **`glutDeviceGet`**`(GLenum info);`
A rendszerre telepített eszközökről szolgáltat információt (egér, billentyűzet, rajzolótábla, spaceball, joystick stb.). Az info paraméter lehetséges értékei:

`GLUT_HAS_KEYBOARD`, `GLUT_HAS_MOUSE`, `GLUT_HAS_SPACEBALL`, `GLUT_HAS_DIAL_AND_BUTTON_BOX`, `GLUT_HAS_TABLET`,

GLUT_NUM_MOUSE_BUTTONS, GLUT_NUM_SPACEBALL_BUTTONS,
GLUT_NUM_DIALS, GLUT_NUM_BUTTON_BOX_BUTTONS,
GLUT_NUM_TABLET_BUTTONS, GLUT_DEVICE_IGNORE_KEY_REPEAT,
GLUT_DEVICE_KEY_REPEAT, GLUT_KEY_REPEAT_OFF,
GLUT_KEY_REPEAT_ON, GLUT_KEY_REPEAT_DEFAULT,
GLUT_JOYSTICK_POLL_RATE, GLUT_HAS_JOYSTICK,
GLUT_JOYSTICK_BUTTONS, GLUT_JOYSTICK_AXES.

`void` **`glutIdleFunc`**`(void (*func)(void));`
Itt beállíthatjuk, melyik legyen az a függvény, amely meghívódik az *Idle* (üresjárati) *callback* esetén. Amikor a program nem csinál semmit, ez a függvény hajtódik végre. Alapértelmezés szerint az értéke `NULL`.

`void` **`glutTimerFunc`**`(unsigned int msecs, void (*func)(int value), value);`
Beállíthatunk egy adott időközönként meghívott függvényt (timert). Ez főleg akkor hasznos, amikor adott sebességű animációkat akarunk előállítani. A **`glutIdleFunc`** *callback*es megoldás nagyban függ a processzor órajelétől, gyorsabb számítógépen így az animáció (játék) is gyorsabbá válik, ami nem biztos, hogy jó. A paraméterek közül a `(*func)` az átadott függvény, melynek egyetlen egész paramétere lehet, ez pedig a `value`. Egyszerre több ilyen időzített hívás is beállítható, viszont egyiket sem lehet visszavonni. Helyette – érdekes megoldás! – a `value` paraméter alapján figyelmen kívül lehet hagyni szükség esetén az időzített hívást.

`void` **`glutMainLoop`**`();`
Belép a fő eseményhurokba.

5.8.4. GLUT menük

A GLUT függvényei segítségével egy OpenGL ablakhoz *popup* (legördülő) menüt rendelhetünk. A használható függvények a következők:

`int` **`glutCreateMenu`**`(void (*func)(int value));`
Menü létrehozása: az adott azonosítóval rendelkező menüponthoz hozzárendeljük az adott függvényt.

`void` **`glutDestroyMenu`**`(int menu);`
Megsemmisíti a menu azonosítóval rendelkező menüt.

`void` **`glutSetMenu`**`(int menu);`
`int` **`glutGetMenu`**`(void);`
Beállítja (vagy visszatéríti) az aktuális menüt.

`void` **`glutAddMenuEntry`**`(char *name, int value);`
A megadott nevű és azonosítójú menüpontot beszúrja a menübe.

`void` **`glutAddSubMenu`**`(char *name, int menu);`
A megadott nevű és azonosítójú almenüt beszúrja a menübe.

`void` **`glutAttachMenu`**`(int button);`

`void` **`glutDetachMenu`**`(int button);`
Hozzárendeli az aktuális menüt az egéreseményekhez (gombnyomáshoz), vagy lekapcsolja ezt. A button értéke `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, vagy `GLUT_RIGHT_BUTTON` lehet.

`void` **`glutChangeToMenuEntry`**`(int entry, char *name, int value);`
`void` **`glutChangeToSubMenu`**`(int entry, char *name, int menu);`
Kicseréli a megadott menüelemeket.

`void` **`glutRemoveMenuItem`**`(int entry);`
Eltávolítja az entry azonosítójú menüpontot.

`void` **`glutMenuStatusFunc`**`(void (*func)(int status, int x, int y));`
`void` **`glutMenuStateFunc`**`(void (*func)(int status));`
Beállítja a globális menüstátusz *callback*et. A status paraméter értékei: `GLUT_MENU_IN_USE` vagy `GLUT_MENU_NOT_IN_USE`.

5.8.5. GLUT karakterek

Karakterek megjelenítésére a GLUT kétféle betűtípusrendszert bocsát rendelkezésünkre: a rasztelgrafikus (*bitmap*) és a vektorgrafikus (*stroke*) betűtípusokat.

`void` **`glutBitmapCharacter`**`(void *font, int character);`
A megadott fonttal megjeleníti a megadott karaktert. A következő betűtípusok, méretek beállítására van lehetőségünk:

`GLUT_BITMAP_8_BY_13`, `GLUT_BITMAP_9_BY_15`,
`GLUT_BITMAP_TIMES_ROMAN_10`, `GLUT_BITMAP_TIMES_ROMAN_24`,
`GLUT_BITMAP_HELVETICA_10`, `GLUT_BITMAP_HELVETICA_12`,
`GLUT_BITMAP_HELVETICA_18`.

`int` **`glutBitmapWidth`**`(GLUTbitmapFont font, int character);`
Visszatéríti a megadott karakter méretét a megadott fontban.

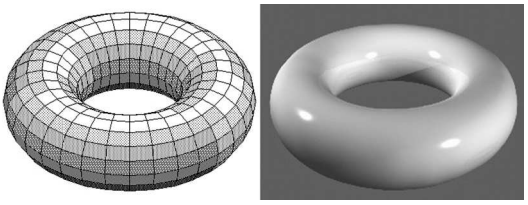
`void` **`glutStrokeCharacter`**`(void *font, int character);`
A megadott fonttal megjeleníti a megadott karaktert. A következő betűtípusok, méretek beállítására van lehetőségünk: `GLUT_STROKE_ROMAN`, `GLUT_STROKE_MONO_ROMAN`.

`int` **`glutStrokeWidth`**`(GLUTstrokeFont font, int character);`
Visszatéríti a megadott karakter méretét a megadott fontban.

5.8.6. GLUT testek

Az OpenGL alapból nem tartalmaz olyan eljárásokat, amelyek magas szintű geometriai objektumok rajzolását teszik lehetővé. Ezeket az eljárásokat a GLUT tartalmazza. GLUT-ban lehetőségünk van mind kitöltött (*solid*), mind drótvázas (*wire*) ábrázolású testek megadására.

`void` **`glutSolidCube`**`(GLdouble size);`
`void` **`glutWireCube`**`(GLdouble size);`
Origó középpontú size élhosszúságú kockát rajzol ki.
`void` **`glutSolidSphere`**`(GLdouble radius, GLint slices, GLint stacks);`
`void` **`glutWireSphere`**`(GLdouble radius, GLint slices, GLint stacks);`
Origó középpontú gömböt rajzol ki. A `radius` a gömb sugara, `slices` a z tengely körüli beosztások száma (mint a földrajzi hosszúság), `stacks` a z tengely menti beosztások száma (mint a földrajzi szélesség).
`void` **`glutSolidCone`**`(GLdouble base, GLdouble height, GLint slices, GLint stacks);`
`void` **`glutWireCone`**`(GLdouble base, GLdouble height, GLint slices, GLint stacks);`
Kúpot rajzol ki. A `base` a kúp alapjának sugara, `height` a kúp magassága, `slices` adja meg a z tengely körüli beosztások számát, `stacks` pedig a z tengely menti beosztások számát jelenti. A kúp alapja z =0-nál helyezkedik el, teteje pedig z =`height`-nél.
`void` **`glutSolidTorus`**`(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);`
`void` **`glutWireTorus`**`(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);`
Tóruszt jelentet meg. Az `innerRadius` a tórusz belső sugara, `outerRadius` a tórusz külső sugara, `nsides` adja meg a radiális részek oldalainak számát, `rings` pedig a tórusz radiális beosztásainak számát. A tórusz középpontja koordináta-rendszer középpontjában lesz.



5.10. ábra. *Drótvázas és kitöltött tórusz*

`void` **`glutSolidDodecahedron`**`(void);`
`void` **`glutWireDodecahedron`**`(void);`
`void` **`glutSolidOctahedron`**`(void);`
`void` **`glutWireOctahedron`**`(void);`
`void` **`glutSolidTetrahedron`**`(void);`
`void` **`glutWireTetrahedron`**`(void);`
`void` **`glutSolidIcosahedron`**`(void);`
`void` **`glutWireIcosahedron`**`(void);`
Kitöltött (*solid*) vagy drótvázas (*wire*) dodekaédert, oktaédert, tetraédert,

ikozaédert rajzol ki. A testek középpontja az origóban lesz. A méret beállításához skálázást, pozíciójának megváltoztatásához forgatást vagy eltolást kell alkalmaznunk.

`void` **`glutSolidTeapot`**`(GLdouble size);`

`void` **`glutWireTeapot`**`(GLdouble size);`

A GLUT ki tudja rajzolni a *Utah teapot* vagy *Newell teapot* néven elhíresült teáskannát, ahol size a teáskanna mérete. A teáskannát számítógépes grafika-referenciaként alkotta meg 1975-ben a utahi egyetemen Martin Newell. A számítógépes grafika hőskorából ránk maradt teáskanna mindmáig a számítógépes grafika „maszkotája”, logója maradt.

5.11. ábra. *A utahi teáskanna árnyalt képe*

5.8.7. Más GLUT lehetőségek

`void` **`glutReportErrors`**`();`

Kiírja a GLUT futásközbeni (*run-time*) hibákat. Nagyon hasznos lehetőség debugolás közben.

`int` **`glutExtensionSupported`**`(char *extension);`

Információt szolgáltat a GLUT lehetőségekről.

`int` **`glutGet`**`(GLenum state);`

A GLUT mint állapotautomata összes beállítását visszaszolgáltatja. Általános rutin, a state által azonosított beállítás értékét téríti vissza.

5.9. OpenGL Visual C++-ban

Ha OpenGL programot szeretnénk létrehozni *VisualC++*-ban, három lehetőségünk van: *Win32 alkalmazás*, *Win32 konzol alkalmazás* és *MFC platformon történő programozás*.

Ha az első kettőt választjuk, akkor a GLUT (OpenGL Utility Toolkit) feladata az ablakozó rendszer kezelése és a grafika megjelenítése. A harmadik esetben az ablakozó rendszert a *Visual C++* MFC osztályhierarchiája oldja meg, és a grafika egy Windowsos kontrollban jelenik meg.

5.9.1. Win32 alkalmazás

Ha Win32 alkalmazásként szeretnénk létrehozni az OpenGL programot, akkor:

- Elindítjuk a *Visual C++* 6.0-ást.
- *File / New... / Projects / Win 32 Application* utat járjuk be a menüből kiindulva.
- Beírjuk a project nevét: *Project name: Elso*.
- Beállítjuk a mentési útvonalat.
- OK gomb, majd:
- *A simple Win32 application*.
- Így a következő főprogram-modul jött létre:

```
1  // Elso.cpp : Defines the entry point
2  // for the application.
3
4  #include "stdafx.h"
5
6  int APIENTRY WinMain(HINSTANCE hInstance,
7                       HINSTANCE hPrevInstance,
8                       LPSTR     lpCmdLine,
9                       int       nCmdShow)
10 {
11   // TODO: Place code here.
12
13   return 0;
14 }
```

- Ha ezzel megvagyunk (a varázsló befejeződött), előjön a *Visual C++* programozói felülete, és elkészült a projektnek megfelelő könyvtárstruktúra is.
- Ha nincs OpenGL bekonfigurálva *Visual C++* alá, akkor ezt a következőképpen tehetjük meg:

- Például a *http://www.xmission.com/~nate/glut.html* honlapról töltsük le a *glut-3.7.6-bin.zip* állományt (vagy ha közben frissítették, akkor az újabb verziót).
- Kicsomagolás után öt állományt kapunk, amelyből három fontos számunkra: *glut.h*, *glut32.lib*, valamint *glut32.dll*.
- Ha nincs írásjogunk rendszerkönyvtárakhoz, akkor másoljuk be a *glut.h*-t és a *glut32.lib*-et a projekt könyvtárába, a *glut32.dll*-t pedig a projekt *Debug* könyvtárába.
- Ha van írásjogunk a rendszerkönyvtárakhoz, akkor véglegesen is feltelepíthetjük az OpenGL-t (így minden projekt tudja használni a fent említett állományokat): másoljuk a *glut32.dll*-t a *Windows / System32* könyvtárba, a *glut32.lib*-et a *Visual Studio* Library könyvtárába (pl. *c:\Program Files\Microsoft Visual Studio\VC98\Lib*), *glut.h* állománynak pedig hozzunk létre egy saját *GL* nevű könyvtárat a *Visual Studio* Include könyvtárában (pl. *c:\Program Files\Microsoft Visual Studio\VC98\Include\GL*).
- A *Visual C++* menüjéből kiindulva, a *Project / Settings* beállításoknál, a *Link* fülnél írjuk hozzá a már meglévő *Object/libary modules* sorhoz a következőket: glut32.lib glu32.lib opengl32.lib glaux.lib.
- A fenti főprogram-modul *include* sorába írjuk be az OpenGL headerállományát is: `#include<GL\glut.h>`, vagy `#include"glut.h"`, ha a *glut.h* a projekt könyvtárában van.

Ha bekonfiguráltuk és használható az OpenGL, akkor megírhatjuk az első programunkat, amely három pontot fog kirajzolni.

```
1  #include "stdafx.h"
2  #include <GL\glut.h>
3
4  // A programunk inicializáló eljárása
5  void init()
6  {
7    // az aktuális törlőszín a fekete
8    glClearColor(0.0, 0.0, 0.0, 0.0);
9    // az aktuális mátrix a vetítési mátrix
10   glMatrixMode(GL_PROJECTION);
11   // betölti az egységmátrixot
12   glLoadIdentity();
13   // párhuzamos vetítés, origó a képernyő közepén
14   gluOrtho2D(-300,300,-300,300);
15 }
16
17 // A programunk rajzoló eljárása
18 void display()
```

```
19 {
20   // letöröljük a képernyőt
21   glClear(GL_COLOR_BUFFER_BIT);
22   // 10-es nagyságú pontjaink legyenek
23   glPointSize(10);
24   // pontokat fogunk specifikálni
25   glBegin(GL_POINTS);
26     glColor3f(1.0, 0.0, 0.0); // piros szín
27     glVertex2i(0, 250);       // egy pont
28     glColor3f(0.0, 1.0, 0.0); // zöld szín
29     glVertex2i(-250, -150);   // még egy pont
30     glColor3f(0.0, 0.0, 1.0); // kék szín
31     glVertex2i(250, -150);    // még egy pont
32      glEnd();                 // több pont nem lesz
33     glFlush();                  // rajzolj!
34 }
35
36 // A billentyűzetkezelés
37 void keyboard(unsigned char key, int x, int y)
38 {
39   switch(key)
40   {
41     case 27:      // ha escape-et nyomtunk
42       exit(0);    // lépjen ki a programból
43       break;
44   }
45 }
46
47 // A főprogram
48 int APIENTRY WinMain(HINSTANCE hInstance,
49                      HINSTANCE hPrevInstance,
50                      LPSTR     lpCmdLine,
51                      int       nCmdShow)
52 {
53   // az ablak egyszeresen bufferelt, RGB módú
54   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
55   // az ablak 600x600-as
56   glutInitWindowSize(600, 600);
57   // az ablak bal felső sarkának koordinátája
58   glutInitWindowPosition(100, 100);
59   // neve: Első
60   glutCreateWindow("Első");
```

```
61    // inicializálás
62    init();
63    // a képernyő események kezelése (Callback)
64    glutDisplayFunc(display);
65    // billentyűzet események kezelése (Callback)
66    glutKeyboardFunc(keyboard);
67    // belépés az esemény hurokba...
68    glutMainLoop();
69    return 0;
70  }
```

5.9.2. Win32 konzol alkalmazás

Ha konzol alkalmazásként hozzuk létre az OpenGL programot, akkor a grafikus ablakunk mellett egy szöveges üzemmódú ablakunk is lesz, ahová adatokat tudunk kiírni egy egyszerű `printf` utasítással. A parancssor paramétereit is át tudjuk adni az OpenGL-nek.

Az alkalmazás létrehozása annyiban különbözik az előzőtől, hogy ebben az esetben a *Visual C++* elindítása után a *File / New... / Projects / Win 32 Console Application* utat járjuk be a menüből kiindulva.

Az alkalmazás főfüggvénye egyszerűen a következő lesz:

```
1  #include "stdafx.h"
2
3  int main(int argc, char* argv[])
4  {
5    return 0;
6  }
```

Az alkalmazás csak annyiban különbözik az előzőtől, hogy a `main` függvény első sorában meghívjuk a `glutInit(&argc, argv);` parancsot, amely inicializálja a glut lib-et a parancssor paramétereinek megfelelően.

5.9.3. MFC alkalmazás

Ha ezt a lehetőséget választjuk, a programunk nem lesz többé átvihető más platformra, csak Windows alatt fog futni, viszont kényelmesen, Windows kontrollok segítségével vezényelhetjük a rajzolást.

[25] alapján összefoglaljuk, hogyan valósíthatjuk meg az OpenGL MFC-be való integrálását.

Az alkalmazás létrehozásakor a *Visual C++* elindítása után a *File / New... / Projects / MFC AppWizard (exe)* utat járjuk be a menüből kiindulva, és a varázsló segítségével létrehozzuk a dialógus alapú *(dialog based)* MFC

alkalmazásoknál megszokott pl. `COglMFCDialogApp` és `CoglMFCDialogDlg` nevű osztályokat, valamint az ezeket tartalmazó header és cpp forrásállományokat.

Moduláris programozás szempontjából jó, ha az OpenGL-t kezelő programrészeket egy külön osztályba írjuk, ezért hozzunk létre egy osztályt a következő definíciókkal (*OpenGLControl.h* headerállomány):

```
#if !defined(AFX_OPENGLCONTROL_H__71C34264_3EB3_
41CF_AD97_0A6020768E03__INCLUDED_)
#define AFX_OPENGLCONTROL_H__71C34264_3EB3_
41CF_AD97_0A6020768E03__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// OpenGLControl.h : header file

#include <GL\glut.h>

// COpenGLControl window

class COpenGLControl : public CWnd
{
public:
  COpenGLControl();
public:
  UINT_PTR m_unpTimer;
  bool m_bIsMaximized;
private:
  CWnd    *hWnd;
  HDC     hdc;
  HGLRC   hrc;
  int     m_nPixelFormat;
  CRect   m_rect;
  CRect   m_oldWindow;
  CRect   m_originalRect;
  float   m_fPosX;
  float   m_fPosY;
  float   m_fZoom;
  float   m_fRotX;
  float   m_fRotY;
  float   m_fLastX;
  float   m_fLastY;
public:
```

```
38    void oglCreate(CRect rect, CWnd *parent);
39    void oglInitialize(void);
40    void oglDrawScene(void);
41    //{{AFX_VIRTUAL(COpenGLControl)
42    //}}AFX_VIRTUAL
43  public:
44    virtual ~COpenGLControl();
45  public:
46    afx_msg void OnDraw(CDC *pDC);
47    //{{AFX_MSG(COpenGLControl)
48    afx_msg void OnPaint();
49    afx_msg int OnCreate(LPCREATESTRUCT
50                         lpCreateStruct);
51    afx_msg void OnTimer(UINT nIDEvent);
52    afx_msg void OnSize(UINT nType, int cx, int cy);
53    afx_msg void OnMouseMove(UINT nFlags,
54                             CPoint point);
55    //}}AFX_MSG
56    DECLARE_MESSAGE_MAP()
57  };
58
59  #endif // !defined(AFX_OPENGLCONTROL_H__71C34264_
60  3EB3_41CF_AD97_0A6020768E03__INCLUDED_)
```

Az OpenGL beállításait és a rajzoló kódrészletet az *OpenGLControl.cpp* forrásállományba írjuk be. Mivel itt az inicializálást már nem a GLUT végzi el és ez erősen rendszerfüggő, egy pixelformátum-leíróval be kell állítanunk a használatos rendszerelemeket és meg kell feleltetnünk ezeket a Windows GDI rendszerének. Az ablak frissítését a legegyszerűbb, ha egy időzítő (*timer*) segítségével végezzük. Külön kell ügyelnünk arra, hogy az ablak átméretezése hogyan érinti az OpenGL felület átméretezését. Itt teremtjük meg az OpenGL felület és a Windows kontrollok kapcsolatát is, valamint itt végezzük el az eseményvezérlést (az eseménykezelést sem a GLUT végzi).

```
1   #include "stdafx.h"
2   #include "oglMFCDialog.h"
3   #include "oglMFCDialogDlg.h"
4   #include "OpenGLControl.h"
5
6   #ifdef _DEBUG
7   #define new DEBUG_NEW
8   #undef THIS_FILE
9   static char THIS_FILE[] = __FILE__;
10  #endif
```

```
11
12 COpenGLControl::COpenGLControl()
13 {
14   m_fPosX = 0.0f;
15   m_fPosY = 0.0f;
16   m_fZoom = 10.0f;
17   m_fRotX = 0.0f;
18   m_fRotY = 0.0f;
19   m_fLastX = 0.0f;
20   m_fLastY = 0.0f;
21   m_bIsMaximized = false;
22 }
23
24 COpenGLControl::~COpenGLControl()
25 {
26 }
27
28 void COpenGLControl::oglCreate(CRect rect,
29                                CWnd *parent)
30 {
31   CString className =
32     AfxRegisterWndClass(CS_HREDRAW |
33     CS_VREDRAW | CS_OWNDC, NULL,
34     HBRUSH)GetStockObject(BLACK_BRUSH), NULL);
35   CreateEx(0, className, "OpenGL", WS_CHILD |
36     WS_VISIBLE | WS_CLIPSIBLINGS |
37     WS_CLIPCHILDREN, rect, parent, 0);
38   m_oldWindow = rect;
39   m_originalRect = rect;
40   hWnd = parent;
41 }
42
43 BEGIN_MESSAGE_MAP(COpenGLControl, CWnd)
44   //{{AFX_MSG_MAP(COpenGLControl)
45   ON_WM_PAINT()
46   ON_WM_CREATE()
47   ON_WM_TIMER()
48   ON_WM_SIZE()
49   ON_WM_MOUSEMOVE()
50   //}}AFX_MSG_MAP
51 END_MESSAGE_MAP()
52
```

```
53 void COpenGLControl::OnPaint()
54 {
55   ValidateRect(NULL);
56 }
57
58 int COpenGLControl::OnCreate(LPCREATESTRUCT
59     lpCreateStruct)
60 {
61   if (CWnd::OnCreate(lpCreateStruct) == -1)
62     return -1;
63   oglInitialize();
64   return 0;
65 }
66
67 void COpenGLControl::oglInitialize(void)
68 {
69   static PIXELFORMATDESCRIPTOR pfd =
70   {
71     sizeof(PIXELFORMATDESCRIPTOR),
72     1,
73     PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
74     PFD_DOUBLEBUFFER,
75     PFD_TYPE_RGBA,
76     32,    // bit depth
77     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
78     16,    // z-buffer depth
79     0, 0, 0, 0, 0, 0, 0,
80   };
81   hdc = GetDC()->m_hDC;
82   m_nPixelFormat = ChoosePixelFormat(hdc, &pfd);
83   SetPixelFormat(hdc, m_nPixelFormat, &pfd);
84   hrc = wglCreateContext(hdc);
85   wglMakeCurrent(hdc, hrc);
86   glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
87   glClearDepth(1.0f);
88   glFrontFace(GL_CCW);
89   glCullFace(GL_BACK);
90   glEnable(GL_DEPTH_TEST);
91   glDepthFunc(GL_LEQUAL);
92   GLfloat ambientLight[] = {0.3f, 0.3f,
93                             0.3f, 1.0f};
94   GLfloat diffuseLight[] = {0.7f, 0.7f,
```

```
95                                0.7f, 1.0f};
96    glEnable(GL_LIGHTING);
97    glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
98    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
99    glEnable(GL_LIGHT0);
100   glEnable(GL_DEPTH_TEST);
101   glShadeModel(GL_SMOOTH);
102   glDisable(GL_CULL_FACE);
103   glEnable(GL_COLOR_MATERIAL);
104   glColorMaterial(GL_FRONT,
105                   GL_AMBIENT_AND_DIFFUSE);
106   glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
107   OnDraw(NULL);
108 }
109
110 void COpenGLControl::OnDraw(CDC *pDC)
111 {
112   glLoadIdentity();
113   glTranslatef(0.0f, 0.0f, -m_fZoom);
114   glTranslatef(m_fPosX, m_fPosY, 0.0f);
115   glRotatef(m_fRotX, 1.0f, 0.0f, 0.0f);
116   glRotatef(m_fRotY, 0.0f, 1.0f, 0.0f);
117 }
118
119 void COpenGLControl::OnTimer(UINT nIDEvent)
120 {
121   switch (nIDEvent)
122   {
123     case 1:
124     {
125       glClear(GL_COLOR_BUFFER_BIT |
126               GL_DEPTH_BUFFER_BIT);
127        oglDrawScene();
128        SwapBuffers(hdc);
129        break;
130     }
131     default:
132        break;
133   }
134   CWnd::OnTimer(nIDEvent);
135 }
136
```

```
137 void COpenGLControl::OnSize(UINT nType,
138                               int cx, int cy)
139 {
140   CWnd::OnSize(nType, cx, cy);
141   if (0 >= cx || 0 >= cy ||
142       nType == SIZE_MINIMIZED) return;
143   glViewport(0, 0, cx, cy);
144   glMatrixMode(GL_PROJECTION);
145   glLoadIdentity();
146   gluPerspective(35.0f, (float)cx / (float)cy,
147                  0.01f, 2000.0f);
148   glMatrixMode(GL_MODELVIEW);
149   switch (nType)
150   {
151     case SIZE_MAXIMIZED:
152     {
153       GetWindowRect(m_rect);
154       MoveWindow(100, 6, cx - 110, cy - 14);
155       GetWindowRect(m_rect);
156       m_oldWindow = m_rect;
157       break;
158     }
159     case SIZE_RESTORED:
160     {
161       if (m_bIsMaximized)
162       {
163          GetWindowRect(m_rect);
164          MoveWindow(m_oldWindow.left,
165                     m_oldWindow.top - 18,
166                     m_originalRect.Width() - 4,
167                     m_originalRect.Height() - 4);
168          GetWindowRect(m_rect);
169          m_oldWindow = m_rect;
170       }
171       break;
172    }
173 }
174   GLfloat lightPos[] = {-50.f, 50.0f,
175                         100.0f, 1.0f};
176   glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
177 }
178
```

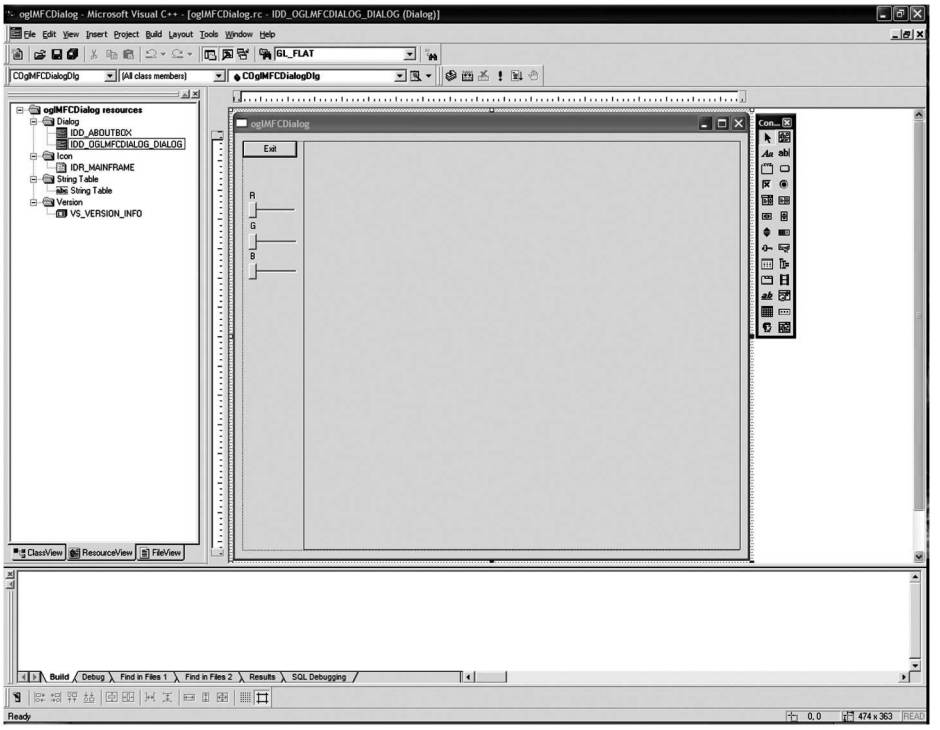
```
179 void COpenGLControl::oglDrawScene()
180 {
181   COglMFCDialogDlg* d =
182     (COglMFCDialogDlg*)theApp.GetMainWnd();
183   glColor3ub(d->m_SliderR.GetPos(),
184     d->m_SliderG.GetPos(),d->m_SliderB.GetPos());
185   d->m_StaticR.Format("R: %i",
186     d->m_SliderR.GetPos());
187   d->m_StaticG.Format("G: %i",
188     d->m_SliderG.GetPos());
189   d->m_StaticB.Format("B: %i",
190     d->m_SliderB.GetPos());
191   d->UpdateData(false);
192   glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
193   glutSolidCube(1);
194   glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
195   glLineWidth(3);
196   glBegin(GL_QUADS);
197     glColor3f(1.0f, 1.0f, 1.0f);
198     glVertex3f(1.0f, 1.0f, 1.0f);
199     glVertex3f(1.0f, 1.0f, -1.0f);
200     glVertex3f(-1.0f, 1.0f, -1.0f);
201     glVertex3f(-1.0f, 1.0f, 1.0f);
202     glVertex3f(-1.0f, -1.0f, -1.0f);
203     glVertex3f(1.0f, -1.0f, -1.0f);
204     glVertex3f(1.0f, -1.0f, 1.0f);
205     glVertex3f(-1.0f, -1.0f, 1.0f);
206     glVertex3f(1.0f, 1.0f, 1.0f);
207     glVertex3f(-1.0f, 1.0f, 1.0f);
208     glVertex3f(-1.0f, -1.0f, 1.0f);
209     glVertex3f(1.0f, -1.0f, 1.0f);
210     glVertex3f(-1.0f, -1.0f, -1.0f);
211     glVertex3f(-1.0f, 1.0f, -1.0f);
212     glVertex3f(1.0f, 1.0f, -1.0f);
213     glVertex3f(1.0f, -1.0f, -1.0f);
214     glVertex3f(-1.0f, -1.0f, -1.0f);
215     glVertex3f(-1.0f, -1.0f, 1.0f);
216     glVertex3f(-1.0f, 1.0f, 1.0f);
217     glVertex3f(-1.0f, 1.0f, -1.0f);
218     glVertex3f(1.0f, 1.0f, 1.0f);
219     glVertex3f(1.0f, -1.0f, 1.0f);
220     glVertex3f(1.0f, -1.0f, -1.0f);
```

```
221     glVertex3f(1.0f, 1.0f, -1.0f);
222   glEnd();
223 }
224
225 void COpenGLControl::OnMouseMove(UINT nFlags,
226                                  CPoint point)
227 {
228   int diffX = (int)(point.x - m_fLastX);
229   int diffY = (int)(point.y - m_fLastY);
230   m_fLastX  = (float)point.x;
231   m_fLastY  = (float)point.y;
232   if (nFlags & MK_LBUTTON)
233   {
234     m_fRotX += (float)0.5f * diffY;
235     if ((m_fRotX > 360.0f) ||
236         (m_fRotX < -360.0f))
237       m_fRotX = 0.0f;
238     m_fRotY += (float)0.5f * diffX;
239     if ((m_fRotY > 360.0f) ||
240         (m_fRotY < -360.0f))
241       m_fRotY = 0.0f;
242   }
243   else if (nFlags & MK_RBUTTON)
244     m_fZoom -= (float)0.1f * diffY;
245   else if (nFlags & MK_MBUTTON)
246   {
247     m_fPosX += (float)0.05f * diffX;
248     m_fPosY -= (float)0.05f * diffY;
249   }
250   OnDraw(NULL);
251   CWnd::OnMouseMove(nFlags, point);
252 }
```

A többi beállítást a varázsló által generált header és forrásállományokban végezzük el. Miután megterveztük a felületet (erőforrásként, resource view, dialog), a legfontosabb, hogy a `COglMFCDialogDlg` osztályba helyezzünk el egy `COpenGLControl m_oglWindow;` változót, így tudjuk megteremteni a kapcsolatot az OpenGL felület és az MFC dialógusablak között.

Ezután a `COglMFCDialogDlg::OnInitDialog()` függvénybe helyezzük el a következő kódrészt:

```
1   CRect rect;
2   GetDlgItem(IDC_OPENGL)->GetWindowRect(rect);
3   ScreenToClient(rect);
```



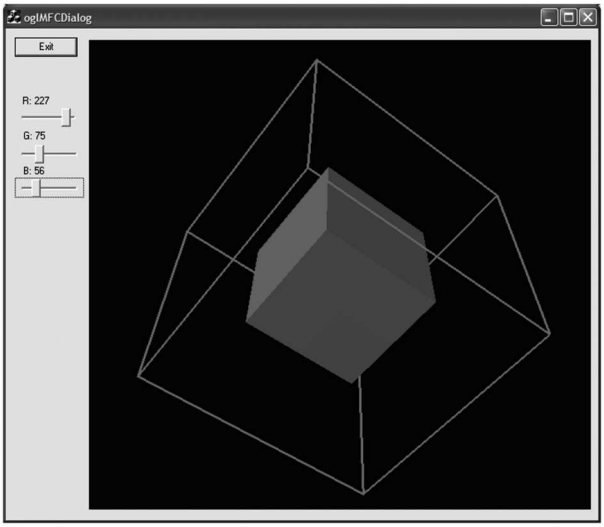
5.12. ábra. *A dialógusablak, az OpenGL felületet egy* Picture *testesíti meg*

```
4   m_oglWindow.oglCreate(rect, this);
5   m_oglWindow.m_unpTimer = m_oglWindow.SetTimer(1,
6       1, 0);
```

Hogy az ablak átméretezése tökéletesen működjön, a következő függvényt kell még megírnunk:

```
1  void COglMFCDialogDlg::OnSize(UINT nType,
2                                int cx, int cy)
3  {
4    CDialog::OnSize(nType, cx, cy);
5    switch(nType)
6    {
7      case SIZE_RESTORED:
8      {
9        if (m_oglWindow.m_bIsMaximized)
10       {
11         m_oglWindow.OnSize(nType, cx, cy);
```

```
12            m_oglWindow.m_bIsMaximized = false;
13          }
14          break;
15        }
16        case SIZE_MAXIMIZED:
17        {
18          m_oglWindow.OnSize(nType, cx, cy);
19          m_oglWindow.m_bIsMaximized = true;
20          break;
21        }
22      }
23    }
```



5.13. ábra. *Egyszerű MFC–OpenGL példa*

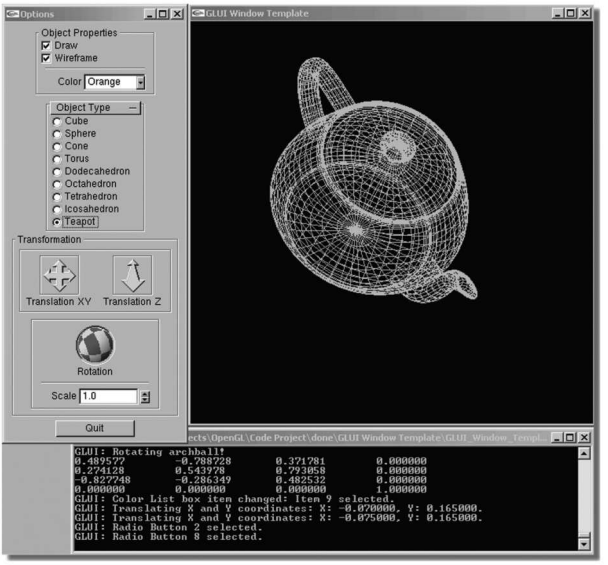
5.10. A GLUI

A GLUI egy Paul Rademacher által fejlesztett GLUT alapú C++-ban felhasználói felületet megvalósító függvénykönyvtár [79], amely letölthető a *http://www.cs.unc.edu/~rademach/glui/* honlapról.

A GLUI a GLUT teljes integrálása mellett grafikus felhasználói felületet (ablakokat és kontrollokat) biztosít az OpenGL alkalmazások számára. Segítségével

könnyen és egyszerűen fejleszthetünk olyan felületeket az OpenGL alkalmazásunk számára, amelyek gombokat (button), jelölődobozokat (checkbox), (radio button), szövegdobozokat (text box), listákat (listbox), címkéket (static text), panelleket (panel), csoportosító dobozokat (groupbox) és más grafikus kontrollokat tartalmaznak, amelyek *callback* függvények segítségével megvalósítják az eseményvezérlést. A kontrollok változókkal szinkronizálhatók, így az értékük valós időben változik, ha az OpenGL kód megváltoztatja őket. Hasonlóan a kontrollok kiválthatják az OpenGL kép frissítését.

A hagyományos (Windows alatt is jó ismert) kontrollokon kívül a GLUI rendszer egy pár saját kontrollt is bevezet, például a grafikus objektumok eltolására, skálázására, forgatására vonatkozó egyszerű, de annál szemléletesebb kontrollokat.



5.14. ábra. *Egyszerű GLUI példa [5]*

5.11. OpenGL Delphiben

Mivel a *Delphi* már eleve felkínálja a *form*ot, azt az űrlapot, amelyet megtervezve megkapjuk az alkalmazás ablakát, az OpenGL használata *Delphi*ből nagyon hasonlít az előbbi MFC példához, csak sokkal egyszerűbb. Több unitot is használhatunk, több OpenGL implementáció létezik *Delphi* alá.

Például ha az *OpenGL12* unitot használjuk (*http://delphi-jedi.org*), a rajzoló felületünk (a form unitja) így fog kinézni:

```
1  unit PontokMain;
2
3  interface
4
5  uses Windows, Messages, SysUtils, Variants,
6    Classes, Graphics, Controls, Forms, Dialogs,
7    ExtCtrls;
8
9  type
10   TForm1 = class(TForm)
11     procedure FormPaint(Sender: TObject);
12     procedure FormDestroy(Sender: TObject);
13     procedure FormCreate(Sender: TObject);
14   end;
15
16 var
17   Form1: TForm1;
18   dc: HDC; // az OpenGL inicializálásához
19   gc: HGLRC; // az OpenGL inicializálásához
20
21 implementation
22
23 uses OpenGL12;
24
25 {$R *.dfm}
26
27 procedure TForm1.FormCreate(Sender: TObject);
28 var h: HPALETTE;
29 begin
30   // a felület inicializálása
31   dc := GetDC(Handle);
32   gc := CreateRenderingContext(dc, [], 32, 0, 0,
33                                0, 0, h);
34   ActivateRenderingContext(dc, gc);
35   // az OpenGL programunk inicializálása
36   glViewport(0, 0, ClientWidth, ClientHeight);
37   glMatrixMode(GL_PROJECTION);
38   glLoadIdentity();
39   gluPerspective(60, ClientWidth / ClientHeight,
40                  0.1, 30.0);
```

```
41    glClearColor(0, 0, 0, 0);
42    glMatrixMode(GL_MODELVIEW);
43    glLoadIdentity;
44  end;
45
46  procedure TForm1.FormDestroy(Sender: TObject);
47  begin
48    DeactivateRenderingContext;
49    DestroyRenderingContext(gc);
50  end;
51
52  procedure TForm1.FormPaint(Sender: TObject);
53  begin
54    glClear(GL_COLOR_BUFFER_BIT);
55    glPointSize(10);
56    glPushMatrix();
57    glTranslatef(-3.0, 2.5, -5.0 );
58    glBegin(GL_POINTS);
59      glColor3f(1.0, 1.0, 0.0);
60      glVertex3f(0.0, 0.0, 0.0);
61      glColor3f(1.0, 0.0, 0.0);
62      glVertex3f(2.0, -2.0, 0.0);
63    glEnd();
64    glPopMatrix();
65    glFlush;
66  end;
67
68  end.
```

Ha a *Delphi*vel forgalmazott *OpenGL* unitot használjuk (`uses OpenGL;`), akkor az MFC-hez hasonlóan ki kell töltenünk egy pixelformátum-leírót. Ekkor az inicializáló függvényünk így néz ki:

```
1   function InitOpenGL(aDC: HDC; ColorBits: integer;
2     DoubleBuffer: boolean ): boolean;
3   // aDC a Device Context, ahova rajzolni fogunk
4   // ColorBits a rajz színskálája: 16/24/32 bit
5   // DoubleBuffer a dupla buffer használata
6   var
7     PixelFormat: TPixelFormatDescriptor;
8     cPixelFormat: integer;
9     gc: HGLRC;
10  begin
```

```
11  FillChar(PixelFormat, SizeOf(PixelFormat), 0);
12  with PixelFormat do
13    begin
14     nSize := Sizeof(TPixelFormatDescriptor);
15     if DoubleBuffer then
16       dwFlags := PFD_DOUBLEBUFFER or
17                  PFD_DRAW_TO_WINDOW or
18                  PFD_SUPPORT_OPENGL
19     else
20       dwFlags := PFD_DRAW_TO_WINDOW or
21                  PFD_SUPPORT_OPENGL;
22     iLayerType := PFD_MAIN_PLANE;
23     iPixelType := PFD_TYPE_RGBA;
24     nVersion := 1;
25     cColorBits := ColorBits;
26     CdepthBits := 16;
27    end;
28  cPixelFormat := ChoosePixelFormat(aDC,
29                  @PixelFormat);
30  Result := cPixelFormat <> 0;
31  if not Result then
32    begin
33      MessageBox(0,
34           pChar(SysErrorMessage(GetLastError)),
35           'Init OpenGL', mb_OK);
36      exit;
37    end;
38  Result := SetPixelFormat(aDC, cPixelFormat,
39            @PixelFormat);
40  if not Result then
41    begin
42      MessageBox(0,
43           pChar(SysErrorMessage(GetLastError)),
44           'Init OpenGL', mb_OK);
45      exit;
46    end;
47  gc := wglCreateContext(aDC);
48  Result := gc <> 0;
49  if not Result then
50    begin
51      MessageBox(0,
52           pChar(SysErrorMessage(GetLastError)),
```

```
53              'Init OpenGL', mb_OK);
54        exit;
55      end;
56    Result := wglMakeCurrent(aDC, gc);
57    if not Result then
58      begin
59        MessageBox(0,
60             pChar(SysErrorMessage(GetLastError)),
61             'Init OpenGL', mb_OK);
62        exit;
63      end;
64  end;
```

A fenti inicializáló függvény meghívása után az elkészített rajzaink meg fognak jelenni a DC által mutatott objektum területén. Windows alatt ügyelnünk kell arra, hogy a használt színmélység egyezzen meg a Windows által beállítottal (Start menü\Control Panel\Display – Settings – Color Quality), ellenkező esetben alkalmazásunk jóval lassúbb lesz, mert az operációs rendszer színkonverziót hajt végre.

5.12. Az OpenGL árnyaló nyelv

Az OpenGL árnyaló nyelve ([75], [84], [55]) a GLSL *(OpenGL Shading Language)*, amely segítségével *vertex-* és *pixel-* (fragment) *shader*ek által programozhatjuk a GPU-t. A vertex-shader lefut minden vertexre, a pixel-shader lefut minden egyes képernyőre kerülő pixelre.

A GLSL egyszerű C szintaxisra épülő nyelv, amely a következő adattípusokkal rendelkezik:

- `float`, `int`, `bool`, `void`: C-szerű típusok;
- `vec2`, `vec3`, `vec4`: 2, 3 és 4 lebegőpontos elemű vektorok;
- `ivec2`, `ivec3`, `ivec4`: 2, 3 és 4 egész elemű vektorok;
- `bvec2`, `bvec3`, `bvec4`: 2, 3 és 4 boolean elemű vektorok;
- `mat2`, `mat3`, `mat4`: 2×2, 3×3, 4×4-es lebegőpontos mátrixok;
- `mat2x2`, `mat2x3`, `mat2x4`, `mat3x2`, `mat3x3`, `mat3x4`, `mat4x2`, `mat4x3`, `mat4x4`: a nevüknek megfelelő méretű mátrixok
- `sampler1D`, `sampler2D`, `sampler3D`: 1D, 2D és 3D textúra;
- `samplerCube`: „Cube Map” textúra;
- `sampler1Dshadow`, `sampler2Dshadow`: 1D és 2D „depth-component” textúra.

Felhasználó által definiálható típusként léteznek a struktúrák (`struct`) és a tömbök (`[]`).

A programozást számos beépített változó segíti, a fontosabbak a következők:

- `gl_Vertex`: egy 4D vektor, a vertex helyzetvektora;
- `gl_Normal`: 3D vektor, a vertexhez tartozó normális;
- `gl_Color`: 4D vektor, a vertex RGBA színe;
- `gl_MultiTexCoordX`: 4D vektor, az X. textúra-elem koordinátái;
- `gl_ModelViewMatrix`: 4×4-es mátrix, a modell-nézet mátrix;
- `gl_ModelViewProjectionMatrix`: 4×4-es mátrix, a modell-nézet és vetítési mátrix;
- `gl_NormalMatrix`: 3×3-as mátrix, amelyet transzformációkhoz használ a rendszer;
- `gl_FrontColor`: 4D vektor, az előtér színe;
- `gl_BackColor`: 4D vektor a háttér színe;
- `gl_TexCoord[X]`: 4D vektor, az X-edik textúra koordinátái;
- `gl_Position`: 4D vektor, az utoljára feldolgozott vertex koordinátái vertex-shader esetén;
- `gl_FragColor`: 4D vektor, az utoljára írt pixel színe pixel-shader esetén;
- `gl_FragDepth`: valós érték, a mélységbufferbe utoljára beírt mélységérték pixel-shader esetén.

A nyelv más elemei (pl. operátorok, függvények stb.) megegyeznek a C-vel, azzal a különbséggel, hogy léteznek tömbműveletek, például összeadást (+), szorzást (*) stb. tömbökkel is végezhetünk.

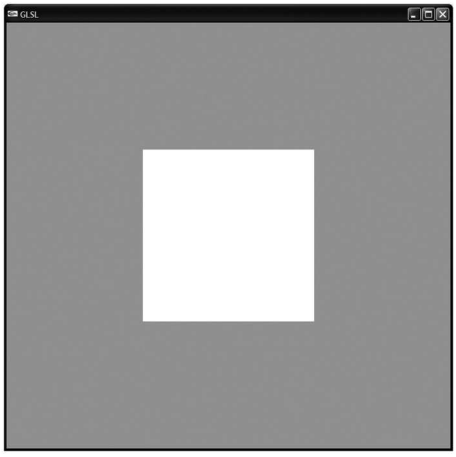
A számításokat számos függvény is segíti a szögkonverziós és trigonometriai függvényektől a különböző analitikus mértan képleteken át a vektor és mátrixműveletekig. Álljon itt egy pár példa:

- `genType` **`radians`**`(genType degrees)`: a fokban mért szöget radiánná alakítja: $\pi/180°$`degrees`;
- `genType` **`sin`**`(genType angle)`: szinusz függvény;
- `genType` **`pow`**`(genType x, genType y)`: hatványozás: x^y;
- `genType` **`sqrt`**`(genType x)`: négyzetgyök: $\sqrt{x}$;
- `genType` **`min`**`(genType x, genType y)`: minimum függvény: x, ha $x < y$, különben y;
- `genType` **`clamp`**`(genType x, genType minVal, genType maxVal)`: szorító függvény: `min(max(x, minVal), maxVal)`;
- `genType` **`mix`**`(genType x, genType y, genType a)`: az `x` és `y` lineáris keveréke: $x \cdot (1-a) + y \cdot a$;
- `float` **`length`**`(genType x)`: az `x` vektor hossza: $\sqrt{x[0]^2 + x[1]^2 + \ldots}$;
- `float` **`dot`**`(genType x, genType y)`: két vektor skaláris szorzata: $x[0] \cdot y[0] + x[1] \cdot y[1] + \ldots$;
- `vec3` **`cross`**`(vec3 x, vec3 y)`: két 3D vektor vektoriális szorzata: $\begin{bmatrix} x[1] \cdot y[2] - y[1] \cdot x[2] \\ x[2] \cdot y[0] - y[2] \cdot x[0] \\ x[0] \cdot y[1] - y[0] \cdot x[1] \end{bmatrix}$;

– `genType` **`normalize`**`(genType x)`: normalizálja a vektort, egy azonos irányú, de 1-es hosszúságú vektort térít vissza;

A következőkben [44] alapján egy egyszerű példát mutatunk be a GLSL alkalmazására.

A feladat: *rajzoljunk ki OpenGL-ben egy fehér négyzetet, majd vertex-shadert használva forgassuk el, valamint pixel-shadert használva színezzük sárgára!*



5.15. ábra. *Fehér négyzet kirajzolása OpenGL-ben*

Az OpenGL program a fehér négyzet kirajzolására egyszerű, a fennebb bemutatott sablont követi:

```
1  #include <iostream>
2  #include <math.h>
3  #include <stdio.h>
4  #include <GL/glut.h>
5
6  void init()
7  {
8    glClearColor(0.75, 0.75, 0.75, 0.0);
9  }
10
11 void render()
12 {
13   glClear(GL_COLOR_BUFFER_BIT |
14           GL_DEPTH_BUFFER_BIT);
15   glLoadIdentity();
16   gluLookAt(0.0, 0.0, 3.0, 0.0, 0.0,
```

```
17            0.0, 0.0, 1.0, 0.0);
18    glBegin(GL_QUADS); // a négyzet kirajzolása
19      glVertex3f(-0.5, -0.5, 0.0);
20      glVertex3f(0.5, -0.5, 0.0);
21      glVertex3f(0.5, 0.5, 0.0);
22      glVertex3f(-0.5, 0.5, 0.0);
23    glEnd();
24    glutSwapBuffers( );
25  }
26
27  void reshape(int w, int h)
28  {
29    glViewport(0, 0, w, h);
30    glMatrixMode(GL_PROJECTION);
31    glLoadIdentity();
32    if (h == 0) h = 1;
33    gluPerspective(45, (float)w/(float)h,
34                   0.1, 100.0);
35    glMatrixMode(GL_MODELVIEW);
36    glLoadIdentity();
37  }
38
39  void keyboard(unsigned char key, int x, int y)
40  {
41    switch (key)
42    {
43      case 27:
44        exit(0);
45        break;
46      default:
47        break;
48    }
49  }
50
51  int main(int argc, char** argv)
52  {
53    glutInit(&argc, argv);
54    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE |
55                        GLUT_DEPTH);
56    glutCreateWindow("GLSL");
57    init();
58    glutDisplayFunc(render);
```

```
59   glutReshapeFunc(reshape);
60   glutKeyboardFunc(keyboard);
61   glutMainLoop();
62   return 0;
63 }
```

Az elforgatás és a színezés megvalósítására GLSL kódrészleteket használunk, ezért az OpenGL-t fel kell készíteni az árnyaló nyelv felismerésére, a shader programok fordítására, futtatására. Ehhez a 2006-ban Ben Woodhouse által kifejlesztett *GLee*-t (*OpenGL Easy Extension Library*) használjuk, amely letölthető a *http://elf-stone.com/glee.php* honlapról.

Először is, a **#include**<GL/glut.h> elé írjuk be:

```
1 #include "GLee.h"
```

Ezután (az *include*-ok után) globális változókként deklaráljuk a shader-programok azonosítóit, valamint string konstansokként deklaráljuk a GLSL programokat:

```
1  GLuint program_object;  // a GLSL program azonosítója
2  GLuint vertex_shader;   // a vertex-shader azonosítója
3  GLuint fragment_shader; // a pixel-shader azonosítója
4
5  // a vertex-shader forráskódja, 45°-os szögben elforgat
6  // egy vertexet
7  static const char *vertex_source =
8  {
9   "void main()"
10  "{"
11  "  float PI = 3.14159265358979323846264;"
12  "  float angle = 45.0;"
13  "  float rad_angle = angle*PI/180.0;"
14  "  vec4 a = gl_Vertex;"
15  "  vec4 b = a;"
16  "  b.x = a.x*cos(rad_angle) - a.y*sin(rad_angle);"
17  "  b.y = a.y*cos(rad_angle) + a.x*sin(rad_angle);"
18  "gl_Position = gl_ModelViewProjectionMatrix*b;"
19  "}"
20 };
21
22 // a pixel-shader forráskódja, sárgára színezi a pixelt
23 static const char *fragment_source =
24 {
25  "void main(void)"
```

```
 "{"
 "  gl_FragColor = vec4(1.0, 1.0, 0.0, 1.0);"
 "}"
};
```

A shader-programokról szóló esetleges fordítási információkat a következő függvénnyel írhatjuk ki:

```
static void printProgramInfoLog(GLuint obj)
{
  GLint infologLength = 0, charsWritten = 0;
  glGetProgramiv(obj, GL_INFO_LOG_LENGTH,
                 &infologLength);
  if(infologLength > 2)
  {
    GLchar* infoLog = new GLchar[infologLength];
    glGetProgramInfoLog(obj, infologLength,
                        &charsWritten, infoLog);
    std::cerr << infoLog << std::endl;
    delete infoLog;
  }
}
```

Az `init` eljárást az aktuális törlőszín definiálása után a következőkkel bővítjük ki:

```
void init()
{
  glClearColor(0.75, 0.75, 0.75, 0.0);

  // létrehozzuk a program objektumot
  program_object  = glCreateProgram();
  // létrehozzuk a vertex-shadert
  vertex_shader = glCreateShader(GL_VERTEX_SHADER);
  // létrehozzuk a pixel-shadert
  fragment_shader = glCreateShader(
                    GL_FRAGMENT_SHADER);
  printProgramInfoLog(program_object);

  // hozzárendeljek a vertex-shader forráskódot
  glShaderSource(vertex_shader, 1,
                 &vertex_source, NULL);
  // hozzárendeljek a pixel-shader forráskódot
  glShaderSource(fragment_shader, 1,
```

```
19                    &fragment_source, NULL);
20    printProgramInfoLog(program_object);
21
22    // lefordítjuk a vertex-shadert és hozzárendeljük
23    //a program objektumhoz
24    glCompileShader(vertex_shader);
25    glAttachShader(program_object, vertex_shader);
26    printProgramInfoLog(program_object);
27
28    // lefordítjuk a pixel-shadert és hozzárendeljük
29    //a program objektumhoz
30    glCompileShader(fragment_shader);
31    glAttachShader(program_object, fragment_shader);
32    printProgramInfoLog(program_object);
33
34    // összeszerkesztjük a teljes GLSL programot
35    glLinkProgram(program_object);
36    printProgramInfoLog(program_object);
37
38    // minden hibát leellenőrizünk
39    GLint prog_link_success;
40    glGetObjectParameterivARB(program_object,
41    GL_OBJECT_LINK_STATUS_ARB, &prog_link_success);
42    if(!prog_link_success)
43    {
44      fprintf(stderr, "Hiba a szerkesztésnél\n");
45      exit(1);
46    }
47  }
```

Most már megvan a teljesen lefordított, összeszerkesztett, működő GLSL programunk, nem maradt más hátra, mint használjuk ezt.

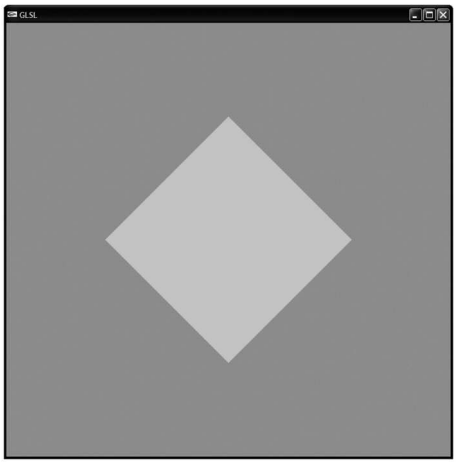
A `render` függvénybe, a négyzet effektív kirajzolása elé (`glBegin (GL_QUADS)`) írjuk be:

```
1   glUseProgram(program_object);
```

utána pedig (`glEnd()` után) szüntessük meg a GLSL program használatát:

```
1   glUseProgram(0);
```

A kód többi része változatlan marad.



5.16. ábra. *A négyzet elforgatása és kiszínezése GLSL-t használva*

6. FEJEZET

GYAKORLATOK ÉS PÉLDAPROGRAMOK

6.1. Vetítés és forgatás

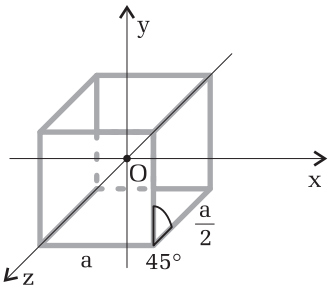
6.1.1. A gyakorlat célja

- A DOS / BGI grafika megismerése
- Az alapvető geometriai transzformációk megismerése, alkalmazása (eltolás, forgatás)
- A vetítés megértése (kavalier-axonometria)
- Munkaidő: 1 labor (programozás)

6.1.2. A feladat

Ábrázoljunk egy origó középpontú, a oldalhosszúságú kockát kavalier-axonometriával, majd a kockát forgassuk el az x, y, z tengelyek körül!

6.1.3. Útmutatás a megoldáshoz



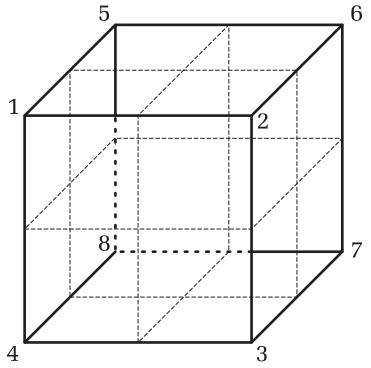
6.1. ábra. *A kavalier-axonometria*

A kavalier-axonometria szerint a z tengelyen 1/2-ed rövidüléssel kell felmérni a kocka oldalait, valamint a z tengely 45° (135°) szöget zár be a másik két tengellyel.

A vetítést több lépésben végezzük el.

1. Először megadjuk a kocka koordinátáit. A kockát 8 pont határozza meg, a nyolc pont mindegyike pedig x, y, z koordinátákkal rendelkezik. Ha a kocka

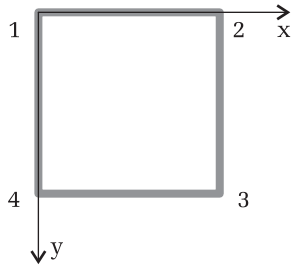
élhosszúsága a, a középpontja pedig az origóban van, a koordináták a következők (x, y, z): ($-a/2$, $a/2$, $a/2$), ($a/2$, $a/2$, $a/2$), ($a/2$, $-a/2$, $a/2$), ($-a/2$, $-a/2$, $a/2$), ($-a/2$, $a/2$, $-a/2$), ($a/2$, $a/2$, $-a/2$), ($a/2$, $-a/2$, $-a/2$), ($-a/2$, $-a/2$, $-a/2$).



6.2. ábra. *A kocka koordinátái*

2. A kavalier-koordináták *(képernyő koordináták)* kiszámítása: a valós z koordinátát elhagyjuk, az x, y-t átszámoljuk z függvényében:

- Vetített$_i$.z = 0, minden i = 1, . . . , 8
- Az elülső oldal változatlan: a középpont egyelőre az oldal közepe lesz:
- Vetített$_i$.x= Kocka$_i$.x, minden i = 1, . . . , 4
- Vetített$_i$.y= Kocka$_i$.y, minden i = 1, . . . , 4



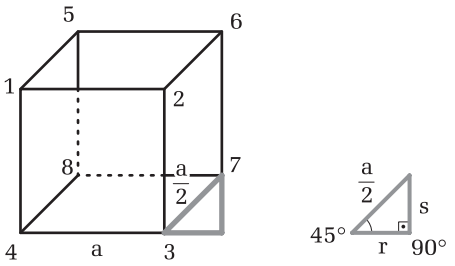
6.3. ábra. *Az elülső oldal*

- A hátulsó oldal koordinátáit el kell tolni az axonometria szabályainak megfelelően. Ki kell számítani az r-et és s-et.
- A szög melleti befogó egyenlő az átfogó és a szög koszinuszának a szorzatával:

$$r = a/2 \cdot \cos(45°)$$

- A szöggel szembeni befogó egyenlő az átfogó és a szög szinuszának a szorzatával:

$$s = a/2 \cdot \sin(45°)$$



6.4. ábra. *A hátulsó oldal*

– A hátulsó oldalra alkalmazzuk a vetítést:
  - Vetített$_i$.x= Kocka$_i$.$x + r$, minden i = 5, . . . , 8
  - Vetített$_i$.y= Kocka$_i$.$y - s$, minden i = 5, . . . , 8

3. Eltolás, hogy az origó a kocka testközéppontjába essen, és itt legyen az ablak középpontja is. A részfeladatot megnehezíti, hogy az ablak koordináta-rendszere eltér a valós koordináta-rendszertől. Míg a valóságban a koordináta-rendszer olyan, hogy az x balról jobbra nő, az y lentről felfele nő, a z pedig hátulról előre (kiáll a képernyőből), addig a képernyő koordináta-rendszerére az jellemző, hogy az x balról jobbra nő, az y fentről lefele nő, a z pedig nincs egyáltalán. Tehát transzformációt kell alkalmazni a két koordináta-rendszer között.

– Az ablak középpontjának a meghatározása:
  - KözX = AblakHossz/2
  - KözY = AblakMagasság/2
– Az eltolások meghatározása:
  - Az ABC derékszögű háromszögben $AB = a\sqrt{2}$, $AM = \frac{a\sqrt{2}}{2}$, $BC = \frac{a}{2}$. Felírhatjuk, hogy $\frac{AM}{AB} = \frac{MO}{BC}$, ahonnan $MO = \frac{a}{4}$.
  - A MNO háromszög hasonló az APQ háromszöghöz, ahonnan az $MN = \frac{r}{2}$, az $NO = \frac{s}{2}$.
– Eltoljuk a koordinátákat:
  - Vetített$_i$.x= KözX + Vetített$_i$.$x - r/2$, minden i = 1, . . . , 8
  - Vetített$_i$.y= KözY + Vetített$_i$.$y + s/2$, minden i = 1, . . . , 8

4. Kirajzolás. Egésszé kerekítjük a vetített koordinátákat, és vonalakból összetéve kirajzoljuk a kockát.

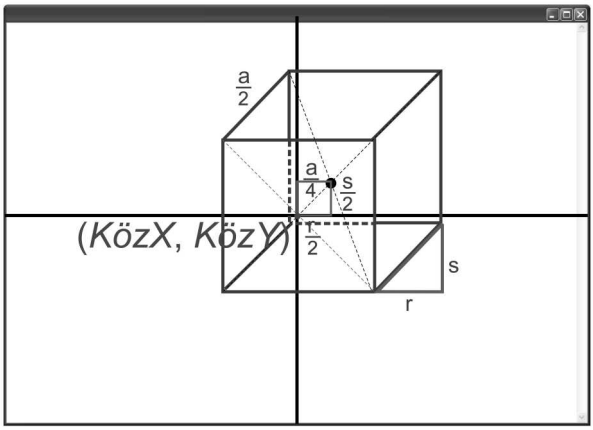
Egy pont origó körüli elforgatásához a tanult képleteket használjuk:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\vartheta & -\sin\vartheta \\ \sin\vartheta & \cos\vartheta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

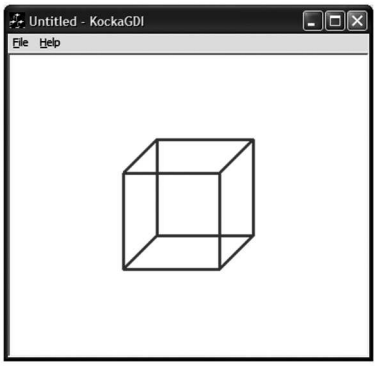
vagy:

$$x' = x \cdot \cos\vartheta - y \cdot \sin\vartheta$$

$$y' = x \cdot \sin\vartheta + y \cdot \cos\vartheta$$



6.5. ábra. *Az eltolások meghatározása*



6.6. ábra. *A kirajzolás*

A transzformációs mátrixok 3D-ben:

$$R_{x,\vartheta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\vartheta & -\sin\vartheta \\ 0 & \sin\vartheta & \cos\vartheta \end{bmatrix}$$

$$R_{y,\vartheta} = \begin{bmatrix} \cos\vartheta & 0 & \sin\vartheta \\ 0 & 1 & 0 \\ -\sin\vartheta & 0 & \cos\vartheta \end{bmatrix}$$

$$R_{z,\vartheta} = \begin{bmatrix} \cos\vartheta & -\sin\vartheta & 0 \\ \sin\vartheta & \cos\vartheta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

6.1.4. A megoldás

A feladatot *Borland Pascal*ban, *Free Pascal*ban, *Borland C++*-ban javasoljuk megoldani.

A fentiek alapján a teljes program *Borland Pascal*ban:

```
1  uses graph, crt;
2
3  type
4     TPont = record {A 3D pont.}
5        x, y, z: real;
6     end;
7
8  var
9     gd, gm, i: integer;
10    kocka, vetitett: array[1..8] of TPont;
11    r: real;
12    KozX, KozY: integer;
13    ch: char;
14
15 const
16    A = 100; {Az el hossza.}
17    C = 0.707106781; {Cos(45)}
18
19 procedure Init; {A koordinatak megadasa.}
20 begin
21    for i := 1 to 8 do
22     begin
23      kocka[i].x := A/2;
24      kocka[i].y := A/2;
```

```
25      if i < 5 then kocka[i].z := A/2
26               else kocka[i].z := -A/2;
27     end;
28     kocka[1].x := -A/2;
29     kocka[3].y := -A/2;
30     kocka[4].x := -A/2;
31     kocka[4].y := -A/2;
32     kocka[5].x := -A/2;
33     kocka[7].y := -A/2;
34     kocka[8].x := -A/2;
35     kocka[8].y := -A/2;
36 end;
37
38 procedure Vetit; {a vetites megvalositasa.}
39 begin
40     for i := 1 to 8 do
41       vetitett[i].z := 0;
42
43     for i := 1 to 4 do
44      begin
45         vetitett[i].x := kocka[i].x;
46         vetitett[i].y := kocka[i].y;
47      end;
48
49     r := A/2*C;
50     for i := 5 to 8 do
51      begin
52         vetitett[i].x := kocka[i].x + r;
53         vetitett[i].y := kocka[i].y - r;
54      end;
55
56     KozX := GetMaxX div 2;
57     KozY := GetMaxY div 2;
58
59     for i := 1 to 8 do
60      begin
61         vetitett[i].x := KozX +
62         vetitett[i].x - r/2;
63         vetitett[i].y := KozY +
64         vetitett[i].y + r/2;
65      end;
66 end;
```

```
67
68  procedure Rajzol; {Kirajzolja a kockat}
69  begin
70     ClearDevice;
71     MoveTo(Round(vetitett[1].x),
72     Round(vetitett[1].y));
73     LineTo(Round(vetitett[2].x),
74     Round(vetitett[2].y));
75     LineTo(Round(vetitett[3].x),
76     Round(vetitett[3].y));
77     LineTo(Round(vetitett[4].x),
78     Round(vetitett[4].y));
79     LineTo(Round(vetitett[1].x),
80     Round(vetitett[1].y));
81     MoveTo(Round(vetitett[5].x),
82     Round(vetitett[5].y));
83     LineTo(Round(vetitett[6].x),
84     Round(vetitett[6].y));
85     LineTo(Round(vetitett[7].x),
86     Round(vetitett[7].y));
87     LineTo(Round(vetitett[8].x),
88     Round(vetitett[8].y));
89     LineTo(Round(vetitett[5].x),
90     Round(vetitett[5].y));
91     MoveTo(Round(vetitett[1].x),
92     Round(vetitett[1].y));
93     LineTo(Round(vetitett[5].x),
94     Round(vetitett[5].y));
95     MoveTo(Round(vetitett[2].x),
96     Round(vetitett[2].y));
97     LineTo(Round(vetitett[6].x),
98     Round(vetitett[6].y));
99     MoveTo(Round(vetitett[3].x),
100    Round(vetitett[3].y));
101    LineTo(Round(vetitett[7].x),
102    Round(vetitett[7].y));
103    MoveTo(Round(vetitett[4].x),
104    Round(vetitett[4].y));
105    LineTo(Round(vetitett[8].x),
106    Round(vetitett[8].y));
107 end;
108
```

```
109 procedure ForgatZ;
110 var i: integer;
111     x, y: real;
112 begin
113     for i := 1 to 8 do
114        begin
115          x := kocka[i].x;
116          y := kocka[i].y;
117          kocka[i].x := x*COS(2)-y*SIN(2);
118          kocka[i].y := x*SIN(2)+y*COS(2);
119        end;
120 end;
121
122 procedure ForgatX;
123 var i: integer;
124     y, z: real;
125 begin
126     for i := 1 to 8 do
127        begin
128          y := kocka[i].y;
129          z := kocka[i].z;
130          kocka[i].y := y*COS(2)-z*SIN(2);
131          kocka[i].z := y*SIN(2)+z*COS(2);
132        end;
133 end;
134
135 procedure ForgatY;
136 var i: integer;
137     x, z: real;
138 begin
139     for i := 1 to 8 do
140        begin
141          x := kocka[i].x;
142          z := kocka[i].z;
143          kocka[i].x := x*COS(2)-z*SIN(2);
144          kocka[i].z := x*SIN(2)+z*COS(2);
145        end;
146 end;
147
148 begin {Foprogram.}
149    gd := Detect;
150    InitGraph(gd, gm, '');
```

```
151    Init;
152    Vetit;
153    Rajzol;
154    repeat
155      ch := ReadKey;
156      if (ch = 'z') then
157       begin
158          ForgatZ;
159          Vetit;
160          Rajzol;
161       end;
162      if (ch = 'x') then
163       begin
164          ForgatX;
165          Vetit;
166          Rajzol;
167       end;
168      if (ch = 'y') then
169       begin
170          ForgatY;
171          Vetit;
172          Rajzol;
173       end;
174    until (ch=#27);
175    CloseGraph;
176  end.
```

6.2. A MahJong Solitaire játék

6.2.1. A gyakorlat célja

- A Windows GDI grafika megismerése
- Az MFC grafikus rendszerének és eseménykezelésének megismerése, elsajátítása
- Bitmap (**.BMP**) állományok létrehozása, kezelése, beolvasása, felhasználása
- Munkaidő: 2 labor (BMP-k megtervezése, alkalmazás megtervezése, programozás)

6.2.2. A feladat

1. Írjunk egy MahJong Solitaire-t megvalósító grafikus játékot, amely véletlenszerűen kirak a teknősbéka alakzat szerint 144 általunk tervezett dominót (36 dominót tervezünk meg, és mindegyikből 4 példány lesz jelen. A 4 példány egymással bárhogy párosítható). Esetünkben a kiválasztott dominóra az egérrel rá kell kattintani, és ha az szabad, a számítógép egy sárga kerettel jelöli meg, mindaddig, amíg a lehetséges párjára nem kattintunk rá az egérrel. Ha valóban párt választottunk ki, azt a gép azonnal el is távolítja. Vegyük figyelembe a fenti szabályokat. A játék mindaddig folytatódik, ameddig a játékos el nem távolította a megadott idő alatt (legyen ez 15 perc) mind a 144 dominót (ekkor nyert), vagy le nem telik az idő, vagy a játékos feladja, mert nem tud többet lépni (ezekben az esetekben a játékos veszít)!

2. A játékosok regisztráljanak és a játék pontozza a teljesítményüket (például minden eltávolított pár legyen 2 pont, a pontok megduplázódnak, ha az időkorlát fele alatt sikerül megoldani a játékot), és tartsa nyilván egy állományban a pontokat!

3. A számítógép kínáljon fel segítséget (javasoljon elmozdítható párokat), ebben az esetben ne pontozza a játékos teljesítményét!

6.2.3. Útmutatás a megoldáshoz

A MahJong egy ősi kínai játék, amely különböző neveken – Majong, Ma Jong, Mah Jong, Mah Jongg, Ma Diao, Ma Cheuk, Mah Cheuck, Baak Ling vagy Pung Chow – ismert. Az eredeti játék egy négy fő által játszott, jellegében a römi kártyajátékra emlékeztető szerencsejáték volt, a játékhoz használt dominók szimbolikája azt mutatja, hogy azokat jósláshoz is használták.

A MahJong Solitaire Táblajáték egy pasziánszszerű párosító logikai játék, amely igen népszerű az egész világon.

Mit is jelent a játék neve? A MahJong az angol változata a sanghaji dialektusban kiejtett *ma-chong* szónak. Ennek a szónak *veréb* a jelentése. Mi lehet ennek az oka? Az utolsó dominók a többiek fogságában vannak, úgy, mint ahogyan a verébcsapat közrefogja társait.

Ez a párosítósdi már korunk számítógépes játéka, amely az ősi játék rekvizitjeit használja. Az első számítógépes solitaire mahjong játékot Brodie Lockard készítette 1981-ben a PLATO típusú számítógépre, amely Mah-Jongg néven lett közismert. Az eredeti táblaelrendezés neve *Teknősbéka* volt. Vagyis a játék negyedszázados múltra tekint vissza. (Azt nehéz elképzelni, hogy a számítógép nélküli „ősi Kínában” valaki valakinek lerakosgatott volna 144 db követ: „No, kezd el párosítgatni!”)

A MahJong Solitaire Táblajátékot egy játékos játssza, az előkészített MahJong dominókból felállított torony által képzett játéktéren (Teknősbéka).

A teknősbéka alakzatot egy 15×8×5-ös tömbbel lehet megvalósítani.

A játék folyamán a játékosnak mindaddig el kell távolítani kettő – a MahJong Solitaire Táblajáték szabályai szerint párt képező (mi határozzuk meg, hogy mi mivel képez párt, például a két dominó egyforma, ugyanaz a szám van rajta, ugyanolyan színű stb.) – dominót, amíg a lehetséges párok el nem fogynak. Csak olyan dominó képezhet párt, amelynek vagy a bal, vagy a jobb oldala szabad, és nem takarja másik dominó. Vagyis eltávolíthatjuk azon két dominót, amelyek szabadok és a játék szabályai szerint párt képeznek.

Ez természetesen nem is olyan könnyű, mint ahogyan elsőre hangzik. A játékos könnyen a játék feladására kényszerül, ha nem talál több párt, mert nem veszi azt észre, vagy a többi – nem pároztatható – dominó blokkolja a lehetséges párokat. A játék során óvatosan kell eljárni, mielőtt bármely párt eltávolítunk a játékmezőről. Nem csak a megtalált párt kell figyelembe venni, hanem az egész játékmezőt, ahol az adott dominónak más párjai is lehetnek. Ha nem vesszük ezeket is figyelembe, akkor a visszamaradó párt blokkolhatjuk. Különösen oda kell figyelni azokra a párokra, amelyek a dominók összekeverése során egymás mellé kerülnek.

Vagyis a MahJong Solitaire nem szerencsejáték, hanem stratégiai.

A játékos nyer, ha a rendelkezésre álló idő alatt mind a 144 dominót eltávolította.

A játékos vereséget szenved, ha nem tud több párt elmozdítani, vagy elfogy a rendelkezésre álló idő.

6.7. ábra. *A teknősbéka alakzat*

6.3. Az első OpenGL példaprogram Visual C++-ban

6.3.1. A gyakorlat célja

- Bevezetés az OpenGL-be
- Az OpenGL primitívek kipróbálása, elsajátítása
- Vertexek megadása, véletlen-pont sztereogram generátor
- Munkaidő: 1 labor (programozás)

6.3.2. A feladat

1. Írjunk egy Win32 *Visual C++* 6.0 alkalmazást, amely inicializálja az OpenGL környezetet, majd színeket és vertexeket definiálva példákat ad az OpenGL primitívek kirajzolására.

A primitívek a következők:

> `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP` és `GL_POLYGON`.

2. Valósítsunk meg egy véletlen-pont sztereogramot generáló alkalmazást!

6.3.3. Útmutatás a megoldáshoz

1. Ha OpenGL programot szeretnénk létrehozni *VisualC++*-ban, három lehetőségünk van: *Win32 alkalmazás*, *Win32 konzol alkalmazás* és MFC *platformon történő programozás*.

Ha az első kettőt választjuk, akkor a GLUT (OpenGL Utility Toolkit) feladata az ablakozó rendszer kezelése és a grafika megjelenítése. A harmadik esetben az ablakozó rendszert a *Visual C++* MFC osztályhierarchiája oldja meg és a grafika egy Windowsos kontrollban jelenik meg.

Jelen példaprogramunkban az első (Win32 alkalmazás) lehetőséget választjuk. Ehhez a következőket kell tenni:

- Elindítjuk a *Visual C++* 6.0-ást.
- *File / New... / Projects / Win 32 Application* utat járjuk be a menüből kiindulva.
- Beírjuk a project nevét: *Project name: Elso*.
- Beállítjuk a mentési útvonalat.
- OK gomb, majd:
- *A simple Win32 application*.
- Így a következő főprogram-modul jött létre:

```
1 // Elso.cpp : Defines the entry point
2 // for the application.
```

```
3
4  #include "stdafx.h"
5
6  int APIENTRY WinMain(HINSTANCE hInstance,
7                       HINSTANCE hPrevInstance,
8                       LPSTR     lpCmdLine,
9                       int       nCmdShow)
10 {
11   // TODO: Place code here.
12
13   return 0;
14 }
```

– Ha ezzel megvagyunk (a varázsló befejeződött), előjön a *Visual C++* programozói felülete, és elkészült a projektnek megfelelő könyvtárstruktúra is.
– Ha nincs OpenGL bekonfigurálva *Visual C++* alá, akkor ezt a következőképpen tehetjük meg:
  • Például a *http://www.xmission.com/~nate/glut.html* honlapról töltsük le a *glut-3.7.6-bin.zip* állományt (vagy, ha közben frissítették, akkor az újabb verziót).
  • Kicsomagolás után öt állományt kapunk, amelyből három fontos számunkra: *glut.h*, *glut32.lib*, valamint *glut32.dll*.
  • Ha nincs írásjogunk rendszerkönyvtárakhoz, akkor másoljuk be a *glut.h*-t és a *glut32.lib*-et a projekt könyvtárába, a *glut32.dll*-t pedig a projekt *Debug* könyvtárába.
  • Ha van írásjogunk a rendszerkönyvtárakhoz, akkor véglegesen is feltelepíthetjük az OpenGL-t (így minden projekt tudja használni a fent említett állományokat): másoljuk a *glut32.dll*-t a *Windows / System32* könyvtárba, a *glut32.lib*-et a *Visual Studio* Library könyvtárába (pl. *c:\Program Files\Microsoft Visual Studio\VC98\Lib*), *glut.h* állománynak pedig hozzunk létre egy saját *GL* nevű könyvtárat a *Visual Studio* Include könyvtárában (pl. *c:\Program Files\Microsoft Visual Studio\VC98\Include\GL*).
  • A *Visual C++* menüjéből kiindulva, a *Project / Settings* beállításoknál, a *Link* fülnél írjuk hozzá a már meglévő *Object/libary modules* sorhoz a következőket: glut32.lib glu32.lib opengl32.lib glaux.lib.
  • A fenti főprogram-modul *include* sorába írjuk be az OpenGL headerállományát is: `#include<GL\glut.h>`, vagy `#include"glut.h"`, ha a *glut.h* a projekt könyvtárában van.

Ha bekonfiguráltuk és használható az OpenGL, akkor megírhatjuk az első példaprogramunkat, amely az OpenGL geometriai primitíveit mutatja be.

A főprogramban a GLUT-re bízzuk az ablakozást:

`glutInitDisplayMode` (az ablak beállításai),
`glutInitWindowSize` (az ablak mérete),
`glutInitWindowPosition` (az ablak bal felső sarkának a koordinátái),
`glutCreateWindow` (az ablak létrehozása).

Szintén itt hívjuk meg az OpenGL-t inicializáló függvényt: `init`, majd az eseménykezelő callback-függvényeit állíthatjuk be. A `glutDisplayFunc`-kal beállított `display` függvény mindig meghívódik az ablak frissítésekor, tehát itt rajzoljunk, a `glutKeyboardFunc`-kal beállított `keyboard` függvény pedig a billentyűzet eseménykezelőjét regisztrálja. A főprogram végén belépünk a fő eseményhurokba: `glutMainLoop`.

Természetesen a főprogram előtt nekünk kell megírnunk az `init`, `display`, `keyboard` függvényeket.

A `display` függvényben történik az effektív rajzolás, itt specifikálhatjuk a vertexeket (csúcspontokat), színeket `glBegin()`, `glEnd()` közé zárva egy-egy primitívet (Begin-End objektum). A primitívek a következők:

`GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP` és `GL_POLYGON`.

A primitívek funkcióit és rajzolási módjukat a 6.8. ábra mutatja (figyeljünk a v csúcspontok – vertexek specifikálási sorrendjére).

A specifikálás után a `glFlush` paranccsal kényszeríthetjük ki a rajzolást.

2. Az elméletben leírt módszert alkalmazzuk.

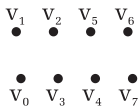
A két szem egymástól egy bizonyos távolságra helyezkedik el a fejen (E– legyen ez nagy általánosan most kb. 6,4 cm), egy árnyalatnyival különböző nézőpontból látják a háromdimenziós világot, így a tárgyakról is kissé különböző kép keletkezik a két szemben, tehát beszélhetünk egy bal és egy jobb oldali képről.

A sztereolátás két lépésből áll: azonosítani kell az összepárosított jegyeket a két szemben, valamint ki kell számítani a jegyek közötti retinális diszparitások nagyságát és irányát.

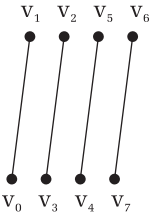
A véletlen-pont sztereogram, amelyet elsőként Julesz Béla (1971) fejlesztett ki, mindkét fele fekete és – itt most – kék pontokból áll. A két fél azonos, egy részlet kivételével. A sztereogram egyik felén a pontok egy központi részhalmazát oldalirányban több sorral elcsúsztattuk. Ez az elmozdítás a két fél között retinális diszparitást eredményez.

Ezek alapján a sztereogram előállításának algoritmusa a következő:

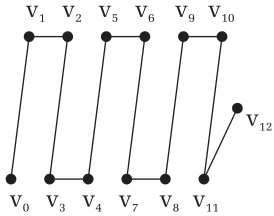
– Induljunk ki egy adott méretű mélységi maszkból ($maxX$, $maxY$). A mélységi maszkot most – az egyszerűség kedvéért – nem egy szürkeárnyalatos képből, hanem egy valós számokat tartalmazó tömbből fogjuk meghatározni (1 a legmagasabban lévő pont, 0 a legalacsonyabban lévő pont).



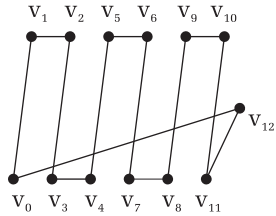
GL_POINTS
Pontok



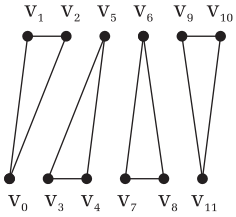
GL_LINES
Vonalak



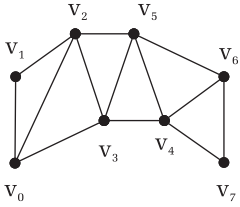
GL_LINE_STRIP
Vonalsáv



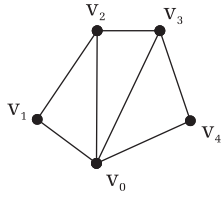
GL_LINE_LOOP
Vonalciklus



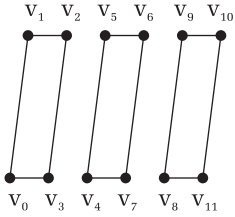
GL_TRIANGLE
Háromszögek



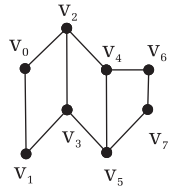
GL_TRIANGLE_STRIP
Háromszögsáv



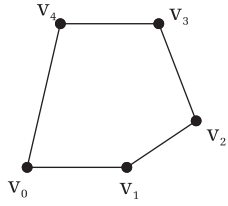
GL_TRIANGLE_FAN
Háromszög-legyező



GL_QUADS
Négyszögek



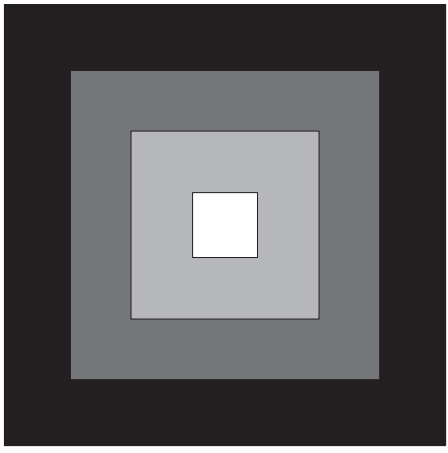
GL_QUAD_STRIP
Négyszögsáv



GL_POLYGON
Sokszög

6.8. ábra. *OpenGL primitívek*

– Minden vízszintes képsoron, balról jobbra haladva, meghatározzuk a mélységi maszk minden pontjának a párját (sztereó szétválasztás) és „összekötjük őket”, vagyis megőrzünk egy referenciát a pontra.
– Ismét minden vízszintes képsoron, balról jobbra haladva, hozzárendelünk egy véletlen színt minden össze nem kötött ponthoz, az összekötött pontok esetében pedig mindkét pontot ugyanolyan színűre festjük (az egyik pont átveszi a másik már meglévő színét).



0,00	0,00	0,00	0,00	0,00	0,00	0,00
0,00	0,33	0,33	0,33	0,33	0,33	0,00
0,00	0,33	0,66	0,66	0,66	0,33	0,00
0,00	0,33	0,66	0,99	0,66	0,33	0,00
0,00	0,33	0,66	0,66	0,66	0,33	0,00
0,00	0,33	0,33	0,33	0,33	0,33	0,00
0,00	0,00	0,00	0,00	0,00	0,00	0,00

6.9. ábra. *Egy háromemeletes piramis mélységi képe*

Valósítsunk meg színes véletlen-pont sztereogramokat és egyképes sztereogramokat megjelenítő alkalmazást is! Valósítsuk meg a szürkeárnyalatos képek (pl. **.BMP**) mélységi maszkként való felhasználását!

6.3.4. A megoldás

1. A program a következő:

```
1  // Elso.cpp : Defines the entry point for the application.
2  //
3
4  #include "stdafx.h"
5  #include <GL\glut.h>
6
7  void init(void)
8  {
9    glClearColor(1.0, 1.0, 1.0, 0.0);
10   // a törlőszín a fehér
11   glMatrixMode(GL_PROJECTION);
```

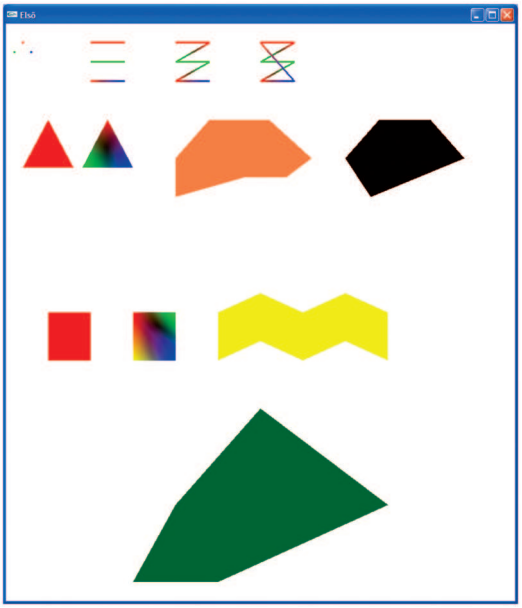
```
12    // az aktuális mátrix mód a vetítési mátrix
13    glLoadIdentity();
14    // betölti az egységmátrixot
15    gluOrtho2D(-300,300,-300,300);
16    // párhuzamos vetítés, origó a képernyő közepén
17  }
18
19  void display(void)
20  {
21    glClear(GL_COLOR_BUFFER_BIT);
22    // letöröljük a képernyőt
23    glPointSize(3);  // 3-as nagyságú pontjaink legyenek
24    glLineWidth(3);  // 3-as vastagságú egyeneseink legyenek
25    glBegin(GL_POINTS);      // pontokat fogunk specifikálni
26      glColor3f(1.0, 0.0, 0.0);    // piros szín
27      glVertex2i(-280, 280);
28      // egy pont a (-280, 280) koordinátába
29      glColor3f(0.0, 1.0, 0.0);    // zöld szín
30      glVertex2i(-290, 270);       // még egy pont
31      glColor3f(0.0, 0.0, 1.0);    // kék szín
32      glVertex2i(-270, 270);       // még egy pont
33    glEnd();                       // több pont nem lesz
34    glBegin(GL_LINES);
35    // vonalakat specifikálunk (hármat)
36      glColor3f(1.0, 0.0, 0.0);
37      // a két pont által meghatározott vonal egyszínű
38      glVertex2i(-200, 280);       // első végpont
39      glVertex2i(-160, 280);       // második végpont
40      glColor3f(0.0, 1.0, 0.0);    // zöld szín
41      glVertex2i(-200, 260);       // első végpont
42      glVertex2i(-160, 260);       // második végpont
43      glColor3f(1.0, 0.0, 0.0);
44      // a vonal színét interpolációval számoljuk ki
45      glVertex2i(-200, 240);       // első végpont
46      glColor3f(0.0, 0.0, 1.0);    // új szín
47      glVertex2i(-160, 240);       // második végpont
48    glEnd();
49    glBegin(GL_LINE_STRIP);
50      // vonalsávot specifikálunk (összekötött vonalak)
51      glColor3f(1.0, 0.0, 0.0);
52      glVertex2i(-100, 280);
53      glVertex2i(-60, 280);
```

```
54      glColor3f(0.0, 1.0, 0.0);
55      glVertex2i(-100, 260);
56      glVertex2i(-60, 260);
57      glColor3f(1.0, 0.0, 0.0);
58      glVertex2i(-100, 240);
59      glColor3f(0.0, 0.0, 1.0);
60      glVertex2i(-60, 240);
61    glEnd();
62    glBegin(GL_LINE_LOOP);
63      // vonalciklust specifikálunk (vissza az elsőhöz)
64      glColor3f(1.0, 0.0, 0.0);
65      glVertex2i(0, 280);
66      glVertex2i(40, 280);
67      glColor3f(0.0, 1.0, 0.0);
68      glVertex2i(0, 260);
69      glVertex2i(40, 260);
70      glColor3f(1.0, 0.0, 0.0);
71      glVertex2i(0, 240);
72      glColor3f(0.0, 0.0, 1.0);
73      glVertex2i(40, 240);
74    glEnd();
75    glBegin(GL_TRIANGLES);          // egyszínű háromszög
76      glColor3f(1.0, 0.0, 0.0);     // piros szín
77      glVertex2i(-250, 200);        // egy pont
78      glVertex2i(-280, 150);        // még egy pont
79      glVertex2i(-220, 150);        // még egy pont
80    glEnd();
81    glBegin(GL_TRIANGLES);          // színháromszög
82      glColor3f(1.0, 0.0, 0.0);     // piros szín
83      glVertex2i(-180, 200);        // egy pont
84      glColor3f(0.0, 1.0, 0.0);     // zöld szín
85      glVertex2i(-210, 150);        // még egy pont
86      glColor3f(0.0, 0.0, 1.0);     // kék szín
87      glVertex2i(-150, 150);        // még egy pont
88    glEnd();
89    glBegin(GL_TRIANGLE_STRIP);     // háromszögsáv
90      glColor3f(1.0, 0.5, 0.25);
91      glVertex2i(-100, 160);  // első háromszög: v0, v1, v2
92      glVertex2i(-100, 120);
93      glVertex2i(-60, 200);
94      glVertex2i(-20, 140);
95      // egy pont a következőhöz: v3
```

```
96      glVertex2i(10, 200);
97      // egy pont a következőhöz: v4
98      glVertex2i(30, 140);
99      // egy pont a következőhöz: v5
100     glVertex2i(60, 160);
101     // egy pont a következőhöz: v6
102   glEnd();
103   glBegin(GL_TRIANGLE_FAN);      // háromszög-legyező
104     glColor3f(0.5, 0.0, 0.0);
105     glVertex2i(130, 120);  // első háromszög: v0, v1, v2
106     glVertex2i(100, 160);
107     glVertex2i(140, 200);
108     glVertex2i(200, 200);
109     // egy pont a következőhöz: v3
110     glVertex2i(240, 160);
111     // egy pont a következőhöz: v4
112   glEnd();
113   glBegin(GL_QUADS);             // egyszínű négyszög
114     glColor3f(1.0, 0.0, 0.0);    // piros szín
115     glVertex2i(-250, 0);         // egy pont
116     glVertex2i(-200, 0);         // még egy pont
117     glVertex2i(-200, -50);       // még egy pont
118     glVertex2i(-250, -50);       // és az utolsó
119   glEnd();
120   glBegin(GL_QUADS);             // sokszínű négyszög
121     glColor3f(1.0, 0.0, 0.0);    // piros szín
122     glVertex2i(-150, 0);         // egy pont
123     glColor3f(0.0, 1.0, 0.0);    // zöld szín
124     glVertex2i(-100, 0);         // még egy pont
125     glColor3f(0.0, 0.0, 1.0);    // kék szín
126     glVertex2i(-100, -50);       // még egy pont
127     glColor3f(1.0, 1.0, 0.0);    // sárga szín
128     glVertex2i(-150, -50);       // és az utolsó
129   glEnd();
130   glBegin(GL_QUAD_STRIP);        // négyszögsáv
131     glColor3f(1.0, 1.0, 0.0);
132     glVertex2i(-50, 0);  // első négyszög: v0, v1, v2, v3
133     glVertex2i(-50, -50);
134     glVertex2i(0, 20);
135     glVertex2i(0, -30);
136     glVertex2i(50, 0);
137     // két pont a következőhöz: v4, v5
```

```
138     glVertex2i(50, -50);
139     glVertex2i(100, 20);
140     // két pont a következőhöz: v6, v7
141     glVertex2i(100, -30);
142     glVertex2i(150, 0);
143     // két pont a következőhöz: v8, v9
144     glVertex2i(150, -50);
145   glEnd();
146   glBegin(GL_POLYGON);           // sokszöget rajzol
147     glColor3f(0.0, 0.5, 0.0);
148     glVertex2i(-100, -200);
149     glVertex2i(-150, -280);
150     glVertex2i(-50, -280);
151     glVertex2i(50, -240);
152     glVertex2i(150, -200);
153     glVertex2i(0, -100);
154   glEnd();
155   glFlush();                     // rajzolj!
156 }
157
158 void keyboard(unsigned char key, int x, int y)
159 {                        //billentűkezelés
160   switch(key)
161   {
162     case 27:             // ha escape-et nyomtunk
163       exit(0);           // lépjen ki a programból
164       break;
165   }
166 }
167
168 // Főprogram
169 int APIENTRY WinMain(HINSTANCE hInstance,
170                      HINSTANCE hPrevInstance,
171                      LPSTR     lpCmdLine,
172                      int       nCmdShow)
173 {
174   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
175   // az ablak egyszeresen bufferelt, RGB módú
176   glutInitWindowSize(600, 600);
177   // az ablak 600x600-as
178   glutInitWindowPosition(100, 100);
179   // az ablak bal felső sarkának koordinátája
```

```
180   glutCreateWindow("Első");
181   // neve: Első
182   init();
183   // inicializálás
184   glutDisplayFunc(display);
185   // a képernyő események kezelése (Callback)
186   glutKeyboardFunc(keyboard);
187   // billentyűzet események kezelése (Callback)
188   glutMainLoop();
189   // belépés az esemény hurokba...
190   return 0;
191 }
```



6.10. ábra. *A program eredménye*

2. A sztereogramot rajzoló program a következő ([95] alapján):

```
1 #include "stdafx.h"
2 #include "glut.h"
3 #include <stdlib.h>
4 #include <memory.h>
5
6 // felkerekítés
```

```
7  #define round(X) (int)((X)+0.5)
8  // DPI beállítás
9  #define DPI 72
10 // a szemek közötti távolság 2.5"=6.4 cm
11 #define E round(2.5*DPI)
12 // mélységi látás-arány
13 #define mu (1/3.0)
14 // a sztereó szétválasztás a Z mélységnek megfelelően
15 #define separation(Z) round((1-mu*Z)*E/(2-mu*Z))
16 // a kép mérete
17 #define maxX 640
18 #define maxY 480
19
20 // a mélységi értékeket tároló tömb [0..1]
21 float Z[maxX][maxY];
22
23 // a rajzolóalgoritmus
24 void DrawStereogram()
25 {
26   // az aktuális pozíció
27   int x, y;
28   // egy sor pixeleinek színe
29   int pix[maxX];
30   // a jobboldali pixeleket ilyen színűre állítjuk
31   int same[maxX];
32   // a sztereo szétválasztás mértéke
33   int s;
34   // a bal és a jobb szemnek megfelelő X értékek
35   int left, right;
36   // soronként dolgozzuk fel a pixeleket
37   for(y=0;y<maxY;++y)
38   {
39     // kezdetben minden pixel magára mutat
40     for(x=0;x<maxX;++x) same[x]=x;
41     // a mélység meghatározása és a pontok bal-jobb szórása
42     for(x=0;x<maxX;x++)
43     {
44       s=separation(Z[x][y]);
45       left=x-(s/2);
46       right=left+s;
47       if(0<=left&&right<maxX)
48       {
```

```
49          for(int k=same[left];k!=left&&k!=right;
50              k=same[left])
51            if(k<right)
52              left=k;
53            else
54            {
55              left=right;
56              right=k;
57            }
58          same[left]=right;
59        }
60      }
61      // fekete vagy kék szín meghatározása
62      for(x=maxX-1;x>=0;--x)
63      {
64        if(same[x]==x) pix[x]=rand()&1;
65        else pix[x]=pix[same[x]];
66        glColor3f(0, 0, pix[x]);
67        glVertex2i(x, y);
68      }
69    }
70  }
71
72  void init()
73  {
74    glClearColor(1.0, 1.0, 1.0, 1.0);
75    glMatrixMode(GL_PROJECTION);
76    glLoadIdentity();
77    gluOrtho2D(0, maxX, 0, maxY);
78    // a mélységi Z tömb feltöltése
79    memset(Z, 0, sizeof(Z));
80    int x, y;
81    for(x=maxX/7;x<=6*maxX/7;++x)
82      for(y=maxY/7;y<=6*maxY/7;++y)
83        Z[x][y] = 0.33;
84    for(x=2*maxX/7;x<=5*maxX/7;++x)
85      for(y=2*maxY/7;y<=5*maxY/7;++y)
86        Z[x][y] = 0.66;
87    for(x=3*maxX/7;x<=4*maxX/7;++x)
88      for(y=3*maxY/7;y<=4*maxY/7;++y)
89        Z[x][y] = 0.99;
90  }
```

```
91
92  void display() {
93   glClear(GL_COLOR_BUFFER_BIT);
94   glBegin(GL_POINTS);
95       DrawStereogram();
96   glEnd();
97   glFlush();
98  }
99
100 void keyboard( unsigned char key, int x, int y) {
101  switch (key) {
102  case 27:
103   exit(0);
104   break;
105  }
106 }
107
108 int APIENTRY WinMain(HINSTANCE hInstance,
109                      HINSTANCE hPrevInstance,
110                      LPSTR     lpCmdLine,
111                      int       nCmdShow)
112 {
113   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
114   glutInitWindowSize(maxX, maxY);
115   glutInitWindowPosition(100, 100);
116   glutCreateWindow("Sztereogram");
117   init();
118   glutDisplayFunc(display);
119   glutKeyboardFunc(keyboard);
120   glutMainLoop();
121   return 0;
122 }
```

6.4. Az OpenGL lehetőségei, alapvető rajzolási algoritmusok

6.4.1. A gyakorlat célja

- Bevezetés az OpenGL-be
- Az OpenGL lehetőségeinek kipróbálása, elsajátítása

6.11. ábra. *Sztereogram: egy háromemeletes piramis*

- Élsimítás, színinterpoláció
- Alapvető rajzoló algoritmusok: vonal (Bresenham), kör, ellipszis (Da Silva), Cohen–Sutherland-féle vágási algoritmus
- Forgatás, skálázás, eltolás
- Timer, IdleFunc, egéresemények használata
- Dupla bufferelés kipróbálása
- Munkaidő: 1 labor (programozás)

6.4.2. A feladat

1. Rajzoljunk ki három különböző színű pontot, majd kössük össze háromszöggé. Próbáljuk ki a

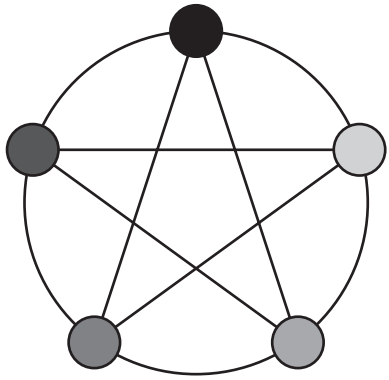
```
glPointSize, glLineWidth,
glLineStipple, glPolygonStipple,
glShadeModel (GL_FLAT, GL_SMOOTH), GL_LINE_SMOOTH,
GL_POINT_SMOOTH
```

eljárásokat, beállításokat.

2. Rajzoljuk meg a 6.12. ábrán látható alakzatot.

3. Próbáljuk ki a skálázás (nagyítás, kicsinyítés), forgatás, eltolás (`glScale`, `glRotate`, `glTranslate`) transzformációs eljárásokat.

4. A forgatást valósítsuk meg automatikusan (animáció). Az animációt valósítsuk meg a `glutIdleFunc` segítségével és timer (időzítő) segítségével is. Mi a különbség? Próbáljuk ki a dupla bufferelést!



6.12. ábra. *Körök, csillag*

5. Implementáljuk a Bresenham-algoritmust vonalrajzolásra, rajzoljunk kört, ellipszist és kitöltött kört!

6. Implementáljuk a Cohen–Sutherland-féle vágási algoritmust!

6.4.3. Útmutatás a megoldáshoz

A Bresenham-algoritmus pszeudokódban:

```
1  Eljárás Vonal(x1, y1, x2, y2, szín: egész);
2    változók:
3     du, dv, dd,
4     p1, p2,
5     u, v, uf, vf,
6     S1, S2: hosszú egész;
7
8    du := x2 - x1;
9    dv := y2 - y1;
10   ha abs(dv) <= abs(du) akkor
11       ha x1 <= x2 akkor
12          u := x1;
13          v := y1;
14          uf := x2;
15       különben
16          u := x2;
17          v := y2;
18          uf := x1;
19          du := -du;
20          dv := -dv;
```

```
21        (ha) vége
22        ha dv >= 0 akkor
23           dd := 2 * dv - du;
24           p1 := 2 * (dv - du);
25           p2 := 2 * dv;
26           S1 := 1; S2 := 0;
27        különben
28           dd := 2 * dv + du;
29           p1 := 2 * dv;
30           p2 := (dv + du);
31           S1 := 0;
32           S2 := -1;
33        (ha) vége
34        TegyélPontot(u, v, szín);
35        ameddig u < uf végezd el
36           u := u + 1;
37           ha dd >= 0 akkor
38             v := v + S1;
39             dd := dd + p1;
40           különben
41             v := v + S2;
42             dd := dd + p2;
43           (ha) vége
44           TegyélPontot(u, v, szín);
45        (ameddig) vége
46     különben
47        ha y1 <= y2 akkor
48           v := y1;
49           u := x1;
50           vf := y2;
51        különben
52           v := y2;
53           u := x2;
54           vf := y1;
55           du := -du;
56           dv := -dv;
57        (ha) vége
58        ha du > 0 akkor
59          dd := 2 * du - dv;
60          p1 := 2 * (du - dv);
61          p2 := 2 * du;
62          S1 := 1;
```

```
63          S2 := 0;
64        különben
65          dd := -2 * du - dv;
66          p1 := -2 * (du + dv);
67          p2 := -2 * du;
68          S1 := -1;
69          S2 := 0;
70        (ha) vége
71        TegyélPontot(u, v, szín);
72        ameddig v < vf végezd el
73          v := v + 1;
74          ha dd >= 0 akkor
75            u := u + S1;
76            dd := dd + p1;
77          különben
78            u := u + S2;
79            dd := dd + p2
80          (ha) vége
81          TegyélPontot(u, v, szín);
82        (ameddig) vége
83      (ha) vége
84  (Eljárás) vége;
```

Ellipszist rajzoló Da Silva algoritmus OpenGL-ben ([61] alapján):

```
1   #include "glut.h"
2
3   void init()
4   {
5     glClearColor(1.0, 1.0, 1.0, 1.0);
6     glMatrixMode(GL_PROJECTION);
7     glLoadIdentity();
8     gluOrtho2D(-400, 400, -400, 400);
9   }
10
11  void pont(double e, double f)
12  {
13    glPointSize(2.0);
14    glBegin(GL_POINTS);
15      glVertex2d(e, f);
16      glVertex2d(-e, f);
17      glVertex2d(-e, -f);
18      glVertex2d(e, -f);
```

```
19    glEnd();
20  }
21
22  void ellipse(double a, double b)
23  {
24    double x, y;
25    double d1, d2;
26    x=0.0;
27    y=b;
28    d1=b*b-a*a*b+a*a/4;
29    pont(x, y);
30    while((a*a*(y-1/2)) > (b*b*(x+1)))
31    {
32      if(d1<0)
33      {
34        d1=d1+b*b*(2*x+3);
35        ++x;
36      }
37      else
38      {
39        d1=d1+b*b*(2*x+3)+a*a*(-2*y+2);
40        ++x;
41        --y;
42      }
43    pont(x, y);
44  }
45  d2=b*b*(x*x+1/4+x)+a*a*(y*y-2*y+1)-a*a*b*b;
46  while(y>0)
47  {
48    if(d2<0)
49    {
50      d2=d2+b*b*(2*x+2)+a*a*(-2*y+3);
51      ++x;
52      --y;
53    }
54    else
55    {
56      d2=d2+a*a*(-2*y+3);
57      --y;
58    }
59    pont(x, y);
60   }
```

```
61 }
62
63 void display()
64 {
65   glClear(GL_COLOR_BUFFER_BIT);
66   glColor3d(0.0, 0.0, 0.0);
67   ellipse(150, 360);
68   glFlush();
69 }
70
71 void keyboard(unsigned char key, int x, int y)
72 {
73   switch (key)
74   {
75     case 27:
76       exit(0);
77       break;
78   }
79 }
80
81 int APIENTRY WinMain(HINSTANCE hInstance,
82                      HINSTANCE hPrevInstance,
83                      LPSTR     lpCmdLine,
84                      int       nCmdShow)
85 {
86   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
87   glutInitWindowSize(200, 200);
88   glutInitWindowPosition(100, 100);
89   glutCreateWindow("ellipszis");
90   init();
91   glutDisplayFunc(display);
92   glutKeyboardFunc(keyboard);
93   glutMainLoop();
94   return 0;
95 }
```

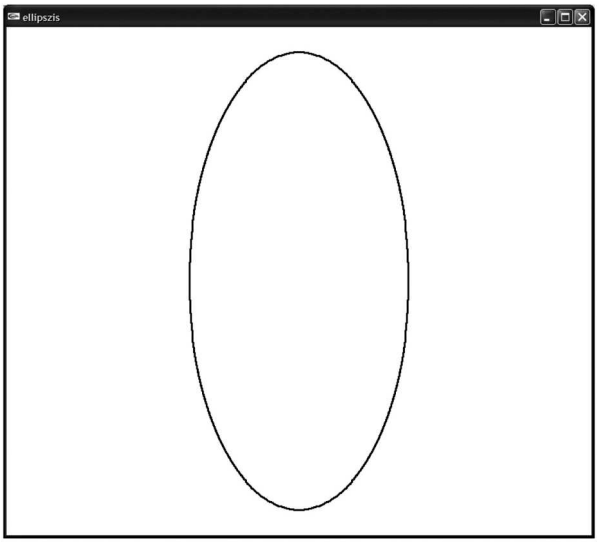
A Cohen–Sutherland-féle vágási algoritmus *Delphi*-ben ([96], [36] alapján):

```
1 unit uMain;
2
3 interface
4
5 uses
```

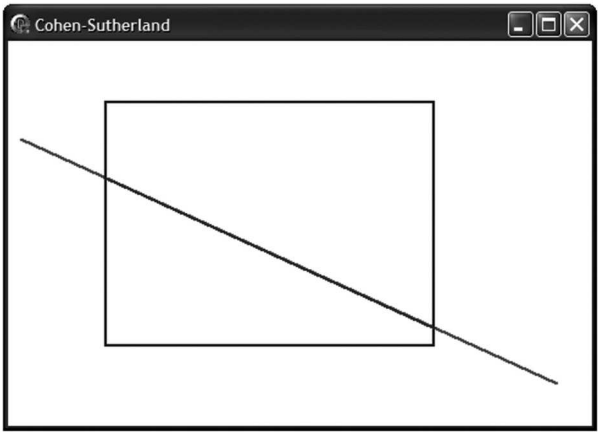
```
6   Windows, Messages, SysUtils, Variants, Classes,
7   Graphics, Controls, Forms, Dialogs;
8
9  type
10  TfrmMain = class(TForm)
11    procedure FormPaint(Sender: TObject);
12  end;
13
14  TEdge = (teLEFT, teRIGHT, teBOTTOM, teTOP);
15  TOutcode = set of TEdge;
16
17 var
18  frmMain: TfrmMain;
19
20 implementation
21
22 {$R *.dfm}
23
24 procedure TfrmMain.FormPaint(Sender: TObject);
25 var
26  x1, y1, x2, y2, xmin, xmax, ymin, ymax: real;
27
28    procedure CompOutCode(x, y: real;
29                          var code: TOutcode);
30    begin
31      code := [];
32      if y > ymax then include(code, teTOP)
33      else if y < ymin then
34        include(code, teBOTTOM);
35      if x > xmax then include(code, teRIGHT)
36      else if x < xmin then include(code, teLEFT);
37    end;
38
39 var
40  accept, done: boolean;
41  outcode1, outcode2, outcodeOut: TOutcode;
42  x, y: real;
43 begin
44   x1 := 10;
45   y1 := 80;
46   x2 := 450;
47   y2 := 280;
```

```
48    xmin := 80;
49    ymin := 50;
50    xmax := 350;
51    ymax := 250;
52    with Canvas do
53      begin
54         Pen.Width := 2;
55         Pen.Color := clRed;
56         MoveTo(Round(x1), Round(y1));
57         LineTo(Round(x2), Round(y2));
58         Pen.Color := clBlue;
59         Brush.Style := bsClear;
60         Rectangle(Round(xmin), Round(ymin),
61                   Round(xmax), Round(ymax));
62      end;
63    accept := false;
64    done := false;
65    CompOutCode(x1, y1, outcode1);
66    CompOutCode(x2, y2, outcode2);
67    repeat
68      if (outcode1=[]) and (outcode2=[]) then
69        begin
70          { trivialis elfogadas }
71          accept := true;
72          done := true;
73        end
74      else if (outcode1*outcode2) <> [] then
75        { trivialis elvetes }
76        done := true
77      else
78        begin
79          if outcode1 <> [] then
80            outcodeOut := outcode1
81          else outcodeOut := outcode2;
82          { metszespont meghatározása }
83          if teTOP in outcodeOut then
84            begin
85              x := x1 + (x2 - x1) *
86                   (ymax - y1) / (y2 - y1);
87              y := ymax;
88            end
89          else if teBOTTOM in outcodeOut then
```

```
90              begin
91                x := x1 + (x2 - x1) *
92                     (ymin - y1) / (y2 - y1);
93                y := ymin;
94              end;
95            if teRIGHT in outcodeOut then
96              begin
97                y := y1 + (y2 - y1) *
98                    (xmax - x1) / (x2 - x1);
99                x := xmax;
100             end
101           else if teLEFT in outcodeOut then
102             begin
103               y := y1 + (y2 - y1) *
104                    (xmin - x1) / (x2 - x1);
105               x := xmin;
106             end;
107           if (outcodeOut = outcode1) then
108             begin
109               x1 := x;
110               y1 := y;
111               CompOutCode(x1, y1, outcode1);
112             end
113           else
114             begin
115               x2 := x;
116               y2 := y;
117               CompOutCode(x2, y2, outcode2);
118             end;
119         end;
120     until done;
121     if accept then
122       with Canvas do
123         begin
124           Pen.Color := clGreen;
125           MoveTo(Round(x1), Round(y1));
126           LineTo(Round(x2), Round(y2));
127         end;
128 end;
129
130 end.
```



6.13. ábra. *Ellipszis rajzolása*



6.14. ábra. *Vágás*

6.5. Függvényábrázolás, görbék

6.5.1. A gyakorlat célja

- Függvények ábrázolása OpenGL-lel
- Koordináta-rendszer kirajzolása
- Írás, betűk, karakterek, betűtípusok használata
- Munkaidő: 1 labor (programozás)

6.5.2. A feladat

1. Rajzoljunk ki egy négyzethálós 2D Descartes-féle koordináta-rendszert (a tengelyeket húzzuk meg vastag vonallal, az origó legyen a képernyő középpontjában), majd ábrázoljunk rajta különféle $y = f(x)$ függvényeket (pl. $y = x^2$, $y = x^3$, $y = \sin(x)$, $y = \cos(x)$ stb.).

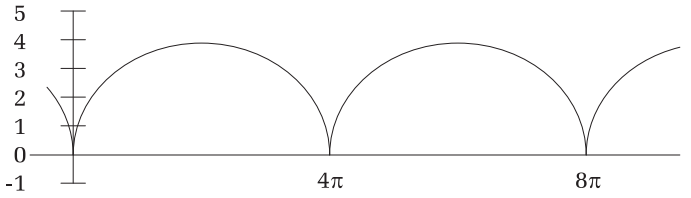
A koordinátatengelyeken tüntessük fel a négyzetháló (és a függvény) megfelelő értékeit is.

2. Rajzoljuk ki az alábbi paraméteres egyenletrendszerrel megadott görbét! Milyen görbét határoz meg az egyenletrendszer?

$$\begin{cases} x = \frac{a}{\cos t} \\ y = b \cdot tgt \end{cases}$$

3. Rajzoluk ki a *ciklois* görbét! A kör tetszőleges pontja cikloist ír le, ha a kör egy egyenesen csúszás nélkül gördül. Egyenletrendszere:

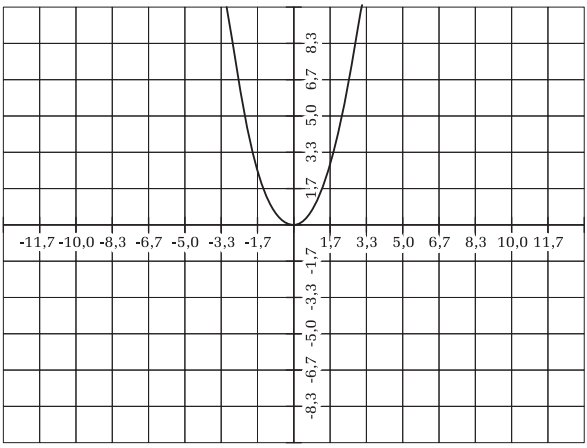
$$\begin{cases} x = a \cdot (t - \sin t) \\ y = a \cdot (1 - \cos t) \end{cases}$$



6.15. ábra. *A ciklois-görbe*

4. Rajzoljuk ki a fogaskerekek kialakításában lényeges *körevolvens* görbét! A körevolvens az origó-középpontú, a sugarú körön legördülő egyenes rögzített pontjának pályagörbéje. Egyenletrendszere:

$$\begin{cases} x = a \cdot (\cos t + t \cdot \sin t) \\ y = a \cdot (\sin t - t \cdot \cos t) \end{cases}$$



6.16. ábra. *Függvényábrázolás*

6.5.3. Útmutatás a megoldáshoz

A 6.16. ábrán látható rajzot kell elkészíteni. Az értékek kiírásához használjuk a

`glutBitmapCharacter`,
`glutStrokeCharacter`

eljárásokat.

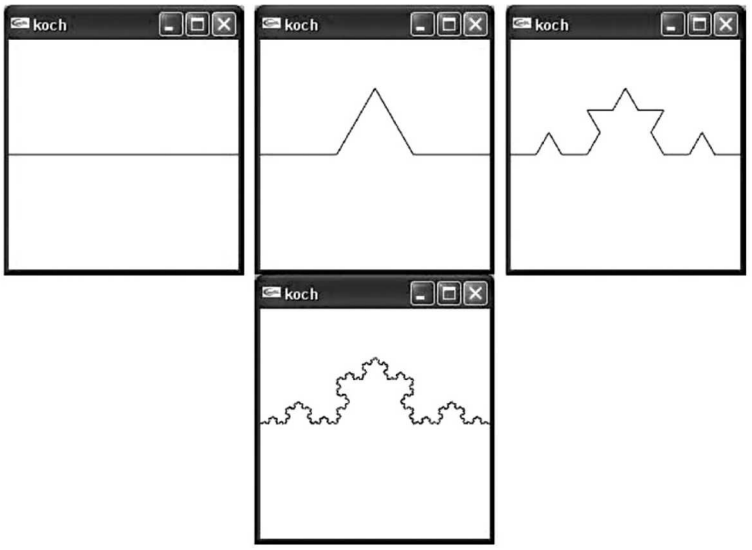
6.6. Fraktálok

6.6.1. A gyakorlat célja

- Bevezető a fraktál-geometriába
- Fraktálok ábrázolása OpenGL-ben, *Delphi*-ben
- Munkaidő: 1 labor (programozás)

6.6.2. A feladat

1. Az óra első részében a Koch-görbét kirajzoló algoritmust ismertetjük.



6.17. ábra. *A Koch-görbe különböző n értékekre (0, 1, 2, 10)*

```
1  //Rajzoljuk meg a Koch görbét!
2
3  #include "stdafx.h"
4  #include "glut.h"
5  #include <stdlib.h>
6  #include <math.h>
7
8  #define rads 0.017453293
9
10 typedef struct {
11    double x;
12    double y;
13 } point2d;
14
15 point2d akt = {0.0, 0.0};
16 point2d rel = {0.0, 0.0};
17
18
19 void koch(double dir, double len, int n)
```

```
20 {
21    if(n > 0) {
22        koch(dir, len/3.0, n-1);
23        dir += 60.0;
24        koch(dir, len/3.0, n-1);
25        dir -= 120.0;
26        koch(dir, len/3.0, n-1);
27        dir += 60.0;
28        koch(dir, len/3.0, n-1);
29    }
30    else {
31        rel.x = len*cos(rads*dir);
32        rel.y = len*sin(rads*dir);
33        glBegin(GL_LINES);
34        glVertex2d(akt.x, akt.y);
35        glVertex2d(akt.x + rel.x, akt.y + rel.y);
36        glEnd();
37        akt.x += rel.x;
38        akt.y += rel.y;
39    }
40 }
41
42 void init()
43 {
44    glClearColor(1.0, 1.0, 1.0, 1.0);
45    glMatrixMode(GL_PROJECTION);
46    glLoadIdentity();
47    gluOrtho2D(0.0, 200.0, 0.0, 200.0);
48    glMatrixMode(GL_MODELVIEW);
49    glLoadIdentity();
50
51 }
52
53 void keyboard(unsigned char key, int x, int y)
54 {
55    switch(key) {
56    case 27:
57        exit(0);
58        break;
59    }
60 }
61
```

```
62 void display()
63 {
64    glClear(GL_COLOR_BUFFER_BIT);
65    glColor3f(0.0f, 0.0f, 0.0f);
66    glTranslatef(0.0f, 100.0f, 0.0f);
67
68
69    koch(0.0, 200.0, 10);
70
71    glFlush();
72
73 }
74
75 int APIENTRY WinMain(HINSTANCE hInstance,
76                      HINSTANCE hPrevInstance,
77                      LPSTR     lpCmdLine,
78                      int       nCmdShow)
79 {
80    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
81    glutInitWindowSize(200, 200);
82    glutInitWindowPosition(20, 20);
83    glutCreateWindow("koch");
84    init();
85    glutDisplayFunc(display);
86    glutKeyboardFunc(keyboard);
87    glutMainLoop();
88    return 0;
89 }
```

2. Barnsley-páfrány

A Barnsley-páfrányt IFS-ként rajzoljuk ki [23] alapján, a következőképpen:

Kiindulunk az origóból (x= 0, y = 0), majd véletlenszerűen (az adott p valószínűségek alapján) 300 000-szer alkalmazzuk az

$$\begin{cases} x' = ax + by + e \\ y' = cx + dy + f \end{cases}$$

transzformációkat, és zöld színnel kirajzoljuk az új pontot.

A transzformációkban szereplő a, b, c, d, e, együtthatókat, valamint a p valószínűséget a következő táblázat foglalja össze:

A *Delphi*ben megvalósított alkalmazás kódja a következő:

```
1 unit uFern;
```

V_i	a	b	c	d	e	f	p
1	0.00	0.00	0.00	0.16	0	0.00	0.01
2	0.85	0.04	−0.04	0.85	0	1.60	0.85
3	0.20	−0.26	0.23	0.22	0	1.60	0.07
4	−0.15	0.28	0.26	0.24	0	0.44	0.07

```
2
3  interface
4
5  uses
6    Windows, Messages, SysUtils, Variants, Classes,
7   Graphics, Controls, Forms, Dialogs;
8
9  type
10   TfrmFern = class(TForm)
11     procedure FormPaint(Sender: TObject);
12   end;
13
14 type
15   TVec = array[1..4] of real;
16
17 const
18   a: TVec = (0, 0.85, 0.2, -0.15);
19   b: TVec = (0, 0.04, -0.26, 0.28);
20   c: TVec = (0, -0.04, 0.23, 0.26);
21   d: TVec = (0.16, 0.85, 0.22, 0.24);
22   e: TVec = (0, 0, 0, 0);
23   f: TVec = (0, 1.6, 1.6, 0.44);
24   p: TVec = (0.01, 0.85, 0.07, 0.07);
25
26 var
27   frmFern: TfrmFern;
28
29 implementation
30
31 {$R *.dfm}
32
33 procedure TfrmFern.FormPaint(Sender: TObject);
34 var
35   x, y, newx, newy, r: real;
36   k, xp, yp: integer;
```

```
37    n: longint;
38  begin
39    x := 0;
40    y := 0;
41    n := 0;
42    repeat
43      inc(n);
44      r := Random;
45      if (r > p[1]) and (r < (p[1] + p[2])) then
46        k := 2
47      else if (r > (p[1] + p[2])) and
48              (r < (p[1] + p[2] + p[3])) then
49        k := 3
50      else if (r > (p[1] + p[2] + p[3])) then
51        k := 4
52      else
53        k := 1;
54      newx := a[k] * x + b[k] * y + e[k];
55      newy := c[k] * x + d[k] * y + f[k];
56      x := newx;
57      y := newy;
58      xp := 300 + round(75 * x);
59      yp := 510 - round(50 * y);
60      Canvas.Pixels[xp, yp] := clGreen;
61    until n = 300000;
62  end;
63
64  end.
```

3. Bináris fa

a) Rajzoljunk meg egy bináris fát!

b) A fa ágainak dőlésszögét perturbáljuk véletlenszerűen!

c) A fa törzsét és ágait rajzoljuk meg barnával, a leveleit pedig zölddel!

d) A fa törzsét rajzoljuk meg vastag vonallal, majd az ágak csúcsai felé vékonyodjanak a vonalak. A leveleket húzzuk ki vastagabb vonallal!

4. Az előadáson tanult algoritmus alapján rajzoljuk meg a Mandelbrot-halmazt!

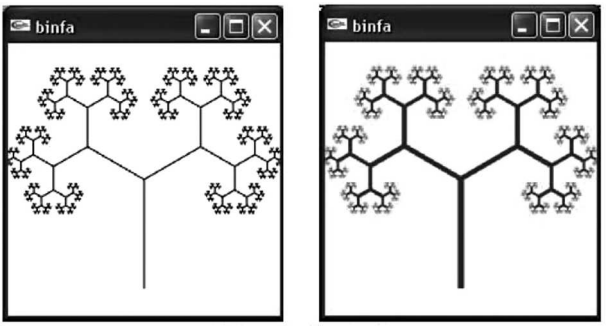
6.6.3. Útmutatás a megoldáshoz

3. A bináris fa törzse merőleges az OX tengelyre és h egységnyi hosszú. A törzsből két ág ágazik ki, mindegyik $h/2$ hosszú, és jobbra, illetve balra 120^o-os szöget zárnak be a törzzsel.

6.18. ábra. *Barnsley-páfrány*

Mindegyik ágból az előbb ismertetett módon újabb ágak hajtanak ki.

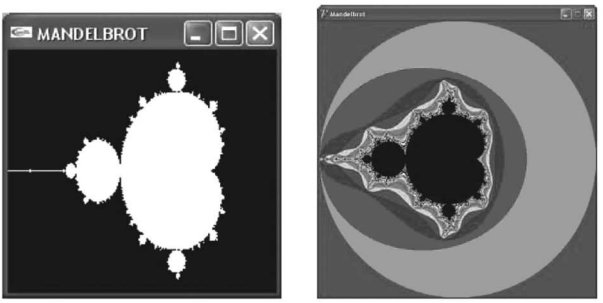
Az utolsó szinten lévő ágakat (melyekből nem hajtanak ki újabb ágak) leveleknek nevezzük.



6.19. ábra. *Bináris fák*

4. A Mandelbrot-halmaz azon c komplex számok halmaza, amelyekre a z_0 = 0, $z_{i+1} = z_i^2 + c$ iteráció eredménye nem a végtelenbe konvergál ($|c|$ <= 2).

Írjunk osztályt a komplex számok ábrázolására, a műveletek megvalósítására!



6.20. ábra. *A Mandelbrot-halmazok*

6.7. 3D grafika, kockák, testek

6.7.1. A gyakorlat célja

- Bevezetés az OpenGL 3D grafikájába
- A *glu* és a *glut* testeinek kipróbálása, elsajátítása
- Árnyalás, fények, színek, megvilágítás használata
- Munkaidő: 1 labor (programozás)

6.7.2. A feladat

1. Alkalmazzuk a

> `gluCylinder`, `gluDisk`, `gluPartialDisk`, `gluSphere`, `glutSolidSphere`, `glutWireSphere`, `glutSolidCube`, `glutWireCube`, `glutSolidCone`, `glutWireCone`, `glutSolidTorus`, `glutWireTorus`, `glutSolidDodecahedron`, `glutWireDodecahedron`, `glutSolidOctahedron`, `glutWireOctahedron`, `glutSolidTetrahedron`, `glutWireTetrahedron`, `glutSolidIcosahedron`, `glutWireIcosahedron`, `glutSolidTeapot`, `glutWireTeapot`

eljárásokat 3D testek kirajzolására!

2. Rajzoljunk ki egy origó középpontú, o oldalhosszúságú kockát, amelynek minden oldala különböző színű!

3. Definiáljunk megvilágítási modelleket, fényforrásokat!

6.7.3. Útmutatás a megoldáshoz

1. A *glu* könyvtár kvadratikus objektumokkal dolgozó eljárásait és a *glut* könyvtár 3D testeit drótvázasan és kitöltötten megjelenítő eljárásokat kell kipróbálni.

Az origó középpontú testeket el kell tolni a megfelelő koordinátákra, át kell őket méretezni. Minden testet külön kiszínezhetünk.

2. Minden oldallapot külön kell kirajzolni `glVertex` eljárásokkal és kiszínezni ezeket.

3. Az előadáson tanult eljárásokat használjuk.

6.21. ábra. *3D testek a glut könyvtárból*

6.8. A CAD alapjai

6.8.1. A gyakorlat célja

- Bevezetés a tervezőprogramok alapjaiba
- Relatív méretek, metrikus rendszerek
- Munkaidő: 1 labor (programozás)

6.8.2. A feladat

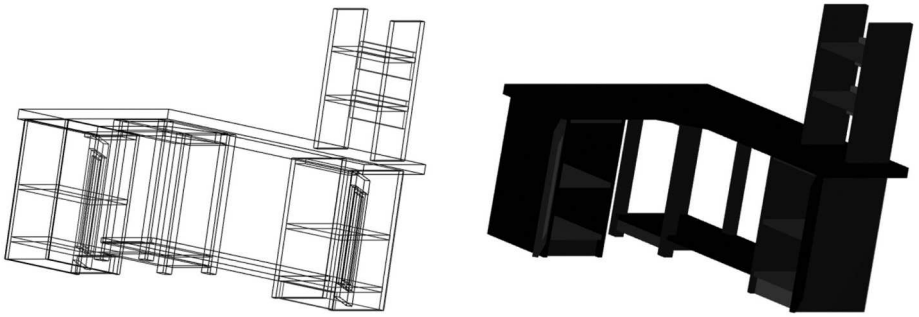
Egy szöveges állományban soronként $x \quad y\ x\ h\ sz\ m$ alakban téglatestek adatai vannak eltárolva. Az x, y, z a téglatest test-középpontjának a koordinátái, a h, sz, m pedig a téglatest hossza, szélessége és magassága. Az adatok milliméterben vannak megadva.

Olvassuk be a szöveges állomány tartalmát, majd jelenítsük meg 3D-ben a testeket a beolvasott adatok alapján. A rajzról készítsünk felülnézetet és oldalnézeteket!

Ábrázoljuk a testeket perspektivikusan és merőleges vetítéssel is!

Rajzoljuk meg egy ház alaprajzát (a falakat)!

6.8.3. Útmutatás a megoldáshoz



6.22. ábra. *Íróasztal drótvázas és árnyalt 3D modellje*

Próbáljunk meg objektumorientáltan gondolkodni! A beolvasott adatok alapján hozzunk létre objektumokat, majd jelenítsük meg ezeket.

6.9. Rubik-kocka

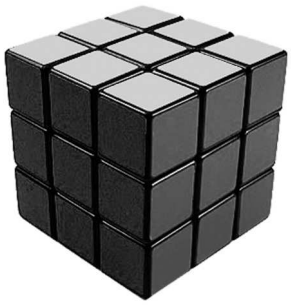
6.9.1. A labor célja

- A forgatás (egyéni és csoportos) alkalmazása
- `glPushMatrix`, `glPopMatrix` utasítások használata
- Objektumorientált programozás alkalmazása, adatstruktúrák használata
- Munkaidő: 2 labor (tervezés, programozás)

6.9.2. A feladat

Jelenítsünk meg egy 3×3×3-as Rubik-kockát. Szimuláljuk a kocka működését! Legyen lehetőség a kocka oldallapjainak az elforgatására, a kocka összekeverésére és kirakására!

6.9.3. Útmutatás a megoldáshoz



6.23. ábra. *A Rubik-kocka (bűvös kocka)*

A Rubik-kocka (másként bűvös kocka) mechanikus, egyéni logikai játék. A kocka oldalai különféle színűek és elforgathatók a lap középpontja körül. A forgatás során a szomszédos oldalak színe megváltozik. A rendszertelen forgatással az oldalak színösszeállítása összekeverhető.

Összesen 43 252 003 274 489 856 000-féle (kb. $4{,}3 \cdot 10^{19}$) eltérő állás hozható létre.

A játék célja, hogy egy előzetesen összekevert kockából forgatással visszaállítsuk az eredeti, rendezett színösszeállítást, vagyis minden oldalon azonos színű lapocskák legyenek.

A kockát Rubik Ernő (1944–) magyar feltaláló, tervező alkotta meg 1975-ben, és hamarosan világszinten népszerű játék lett.

A feladat megoldásához célszerű objektumorientáltan eljárni. Egy kis kocka legyen egy objektum, melyen minden oldallapot különböző színűre állíthatunk be. Az objektumokat tároljuk el egy $3 \times 3 \times 3$-as mátrixban.

Oldjuk meg a kis kockák csoportosítását oldallapok szerint (egy kis kocka több oldallaphoz is tartozhat), oldjuk meg az oldallapok forgatását, a színek kicserélését!

6.9.4. Általánosítás

Próbáljunk meg $2 \times 2 \times 2$, $4 \times 4 \times 4$, ..., $n \times n \times n$-es Rubik-kockákat is megvalósítani!

6.10. Tükrök és trükkök

6.10.1. A gyakorlat célja

- A tükrözés alkalmazása
- Egy kaleidoszkóp elkészítése
- A sugárkövetés alapjainak a megismerése
- Munkaidő: 2 labor (programozás)

6.10.2. A feladat

1. Rajzoljunk képernyőre n síktükröt, mindegyiket az (x_1, y_1), ..., (x_n, y_n) koordinátákra θ_1,...,θ_n dőlésszöggel az x tengely szerint, majd az első tükörre essen be egy α szögű fénysugár. Kövessük (rajzoljuk meg) a fénysugár útját a tükrök között!

2. Ha a fény az anyag felületére érkezik, valójában három dolog történik:

- a fény egy része visszaverődik,
- a fény egy része elnyelődik és az anyagban hővé alakul,
- a fény egy része áthatol a közegen.

Tökéletes tükörről az összes fény visszaverődik. A valóságban mindig vannak veszteségek, s a visszaverődött fény intenzitása csökken. A legnagyobb reflexiós tényezője az ezüstnek van (95%), ezt követi az alumínium (92%).

Minden egyes tükörre legyen jellemző egy reflexiós tényező (százalékban megadva). Módosítsuk az 1. feladatot úgy, hogy a fény intenzitását a fénysugár vastagságával jelöljük, és rajzoljuk meg a fénysugár útját úgy, hogy figyelembe vesszük minden tükör reflexiós tényezőjét (kezdetben a fénysugár vastag, majd a tükrökkel való „ütközések" során egyre vékonyabb lesz, minden tükör reflexiós tényezőjének függvényében).

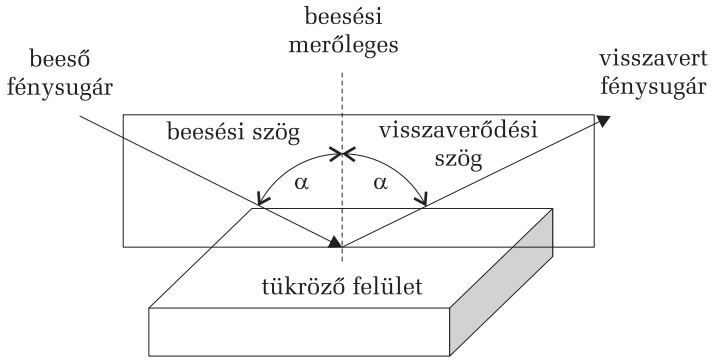
3. Írjunk kaleidoszkópot szimuláló grafikus alkalmazást!

6.10.3. Útmutatás a megoldáshoz

Tekintsünk egy fénysugarat, amely egyenes vonalban terjed mindaddig, amíg valamely test vagy eltérő közeg határfelületére nem érkezik. Ha a felület tükröző, a fénysugár visszaverődik. Az abba a pontba állított merőlegest, ahol a beeső fénysugár eléri a felületet, *beesési merőleges*nek hívják. A beeső és a visszavert fénysugárnak a beesési merőlegessel bezárt szögét *beesési,* illetve *visszaverődési szög*nek hívjuk (6.24. ábra).

A 6.24. ábrára tekintve könnyen megállapíthatók a fényvisszaverődés törvényei (Eukleidész, i. e. 300 körül):

- a beeső fénysugár, a beesési merőleges és a visszavert fénysugár egy síkban vannak
- a beesési szög megegyezik a visszaverődési szöggel



6.24. ábra. *A fény visszaverődése*

6.10.3.1. Középpontos tükrözés

Az egyenesen értelmezett középpontos tükrözés az egyenes egy P pontját abba a P^1 pontba viszi, ami ugyanolyan távol van a középponttól, mint P, hogy a középpont a PP^1 szakasz felezőpontja legyen.

Legtöbbször ennek síkbeli kiterjesztéséről beszélnek. Ekkor a sík egy P pontjának P^1 képe az a pont, ami a P pontot a középponttal összekötő egyenesen fekszik, hogy a középpont a PP^1 szakasz felezőpontja.

A középpontos tükrözésnek a következő tulajdonságai vannak a síkban:

- olyan forgatás, amelynek szöge 180 fok
- irányítástartó
- megadható középpontjával, vagy egy pont-pont képe párral
- van egy fixpontja: a középpontja
- invariáns egyenesei a középponton átmenő egyenesek
- felírható két tengelyes tükrözés szorzataként, melyek tengelyei merőlegesek egymásra, és a középpontban metszik egymást
- a transzformációszorzásban eltolással szorozva középpontos tükrözést ad
- két középpontos tükrözés szorzata eltolás

6.10.3.2. Tengelyes tükrözés

Tengelyes tükrözés az a transzformáció, ami szerint egy P pont képe az a P^1 pont, ami a P pontból a tükrözés tengelyére bocsátott merőlegesen fekszik, és távolsága megegyezik a P pont tengelytől mért távolságával.

Síkban a tengely a PP^1 szakasz felezőmerőlegese. Térben a szakasz felezőmerőleges síkjában helyezkedik el.

Tulajdonságok a síkban:

– irányításváltó
– megadható tengelyével vagy pont-pont képe párral
– fixegyenese a tengelye
– invariáns egyenesei a tengelyre merőleges egyenesek
– egy tengelyes tükrözés a tengelyes tükrözések közül csak önmagával vagy rá merőleges tengelyű tükrözéssel cserélhető fel
– önmagával vett szorzata identitás
– három, egy ponton átmenő tengelyű vagy párhuzamos tengelyű tükrözés szorzata tengelyes tükrözés
  • ebben a szorzatban a két szélső tényező felcserélhető egymással
– három, páronként egymást metsző tengelyű tükrözés szorzata csúsztatva tükrözés
– két tengelyes tükrözés szorzata
  • forgatás, ha a tengelyek metszik egymást
  • eltolás, ha párhuzamosak
– páros számú tengelyes tükrözés szorzata nem írható fel páratlan számú tengelyes tükrözés szorzataként
– a sík összes transzformációja előáll legfeljebb három tengelyes tükrözés szorzataként

Tulajdonságok a térben:

– irányítástartó
– 180 fokos forgatás a tengely körül
– megkapható két merőleges síkra tükrözéssel
– önmagával vett szorzata identitás
– megadható tengelyével
– fixegyenese a tengelye
– invariáns egyenesei a tengelyre merőleges egyenesek
– egy egyenes képe abban a pontban metszi a tengelyt, amiben az egyenes is metszi
– a tengellyel párhuzamos egyenesek képei is párhuzamosak a tengellyel, és tengelytől vett távolságuk megegyezik az eredeti egyenesek tengelytől mért távolságával
– három párhuzamos tengelyre vett tükrözés szorzata tengelyes tükrözés

Tengelyes szimmetria: egy alakzat tengelyesen szimmetrikus a síkban, ha van tengely, amire tükrözve önmagába megy át. Ilyen például az egyenlő szárú háromszög, a húrtrapéz, a deltoid, a rombusz, a téglalap, a kör.

6.10.3.3. Síkra tükrözés

Síkra tükrözés az a (téren értelmezett) transzformáció, ami szerint egy P pont képe az a P^1 pont, ami a P pontból a tükrözés síkjára bocsátott merőlegesen fekszik, és távolsága megegyezik a P pont tengelytől mért távolságával.

Ekkor a tükrözés síkja a PP^1 szakasz felező merőleges síkja.

Tulajdonságok:
- irányításváltó
- egyértelműen megadható a tükrözés síkjával, vagy egy pont-pont képe párral
- a tükrözés síkja fix
- a tükrözés síkjára merőleges egyenesek, síkok invariánsak
- a tükrözés síkjával párhuzamos egyenes, sík képe párhuzamos az eredeti egyenessel, síkkal
- a tükrözés síkját metsző egyenes, sík képe ugyanabban a pontban metszi a síkot, mint az eredeti egyenes, sík
  - ekkor a tükrözés síkja az eredeti és a képsíkok szögfelező síkja
- önmagával vett szorzata az identitás
- három közös egyenesen átmenő vagy párhuzamos síkra vett tükrözés szorzata síkra vett tükrözés, és ebben a szorzatban a két szélső tényező felcserélhető
- a tér bármely egybevágósága előáll legfeljebb négy síkra tükrözés szorzataként
- páros számú síkra tükrözés szorzata nem fejezhető ki páratlan számú síkra tükrözés szorzataként
- két síkra tükrözés csak akkor cserélhető fel, ha síkjaik merőlegesek egymásra vagy megegyeznek

6.10.3.4. Képletek

A síkban (az origón átmenő) egyenesre, a térben (origón átmenő) síkra tükrözés mátrixa -1 determinánsú ortogonális mátrix.

A koordinátasíkokra, koordinátatengelyekre, origóra tükrözés mátrixai a térben:

az YZ koordinátasíkra:

$$\mathrm{Ref}(YZ) = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

az X tengelyre:

$$\mathrm{Ref}(X) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

az origóra:

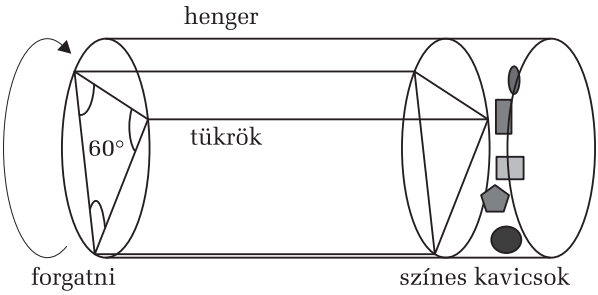
$$\mathrm{Ref}(O) = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

6.10.3.5. A kaleidoszkóp

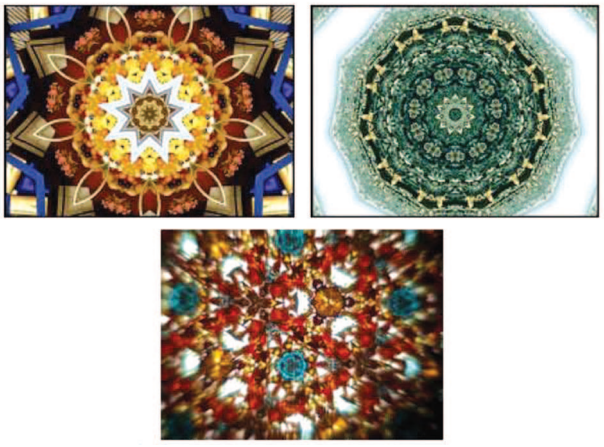
A görögök már ismerték, Sir David Brewster (1781–1868) 1816-ban találta fel ismét a *kaleidoszkóp*ot.

Egyszerűen úgy készíthetünk kaleidoszkópot, hogy három azonos méretű téglalap alakú tükörből egyenlő oldalú háromszög alapú hasábot képezünk, majd ezt egy hengerbe (csőbe) csúsztatjuk. A cső egyik végére feszítsünk átlátszó műanyag fóliát vagy zsírpapírt, majd a csövet kívülről borítsuk be fekete papírral! Szórjunk a cső lezárt végébe apró színes tárgyakat, ezüstpapírt, műanyagdarabkákat, gyöngyöt stb.!

A cső szabad végét szemünk elé emelve, szabályos háromszögekből álló szimmetrikus mintázatban gyönyörködhetünk. A minta a cső mozgatásával (forgatásával) változik (a gravitáció miatt átrendeződnek a színes tárgyak).



6.25. ábra. *A kaleidoszkóp vázlatos szerkezete*



6.26. ábra. *A kaleidoszkóp mintázatai*

6.11. Textúrák alkalmazása

6.11.1. A gyakorlat célja

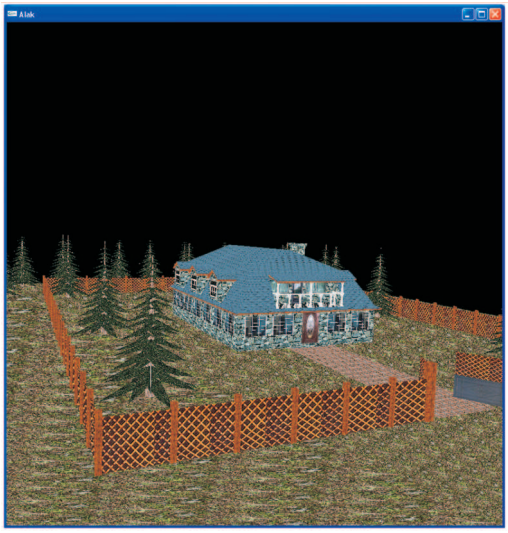
- A textúrák alkalmazása
- Kvadratikus objektumok használata
- **.BMP**-k szerkesztése, beolvasása, felhasználása
- Munkaidő: 2 labor (tervezés, programozás)

6.11.2. A feladat

Legalább 6 megtervezett **.BMP** textúrát alkalmazva, rajzoljunk meg egy általunk választott (tervezett) valós világbeli épületet, tájat, városrészt, parkot stb.

6.11.3. Útmutatás a megoldáshoz

Az előadáson tanult textúrázási technikákat kell alkalmazni.



6.27. ábra. *Textúrás táj*

6.12. Görbék és felületek

6.12.1. A gyakorlat célja

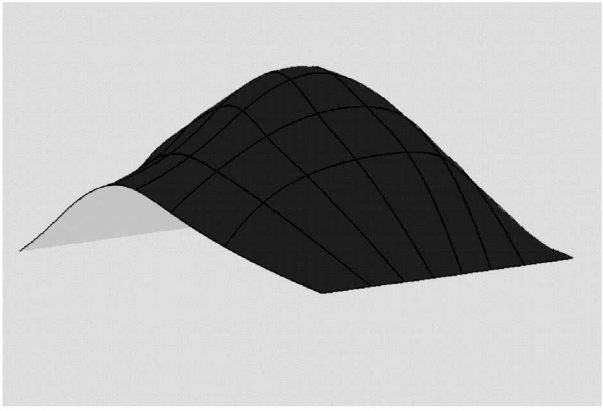
- Az OpenGL-lel megjeleníthető görbék (2D) és felületek (3D) megismerése
- Munkaidő: 1 labor (programozás)

6.12.2. A feladat

Rajzoljunk Bézier, B-spline, NURBS, Hermite, paraméteres görbéket és felületeket!

6.12.3. Útmutatás a megoldáshoz

Az előadáson tanult görbe- és felület-megjelenítő technikákat kell alkalmazni, próbájuk ki a GLU és GLUJ könyvtárak eljárásait!



6.28. ábra. *OpenGL felület*

6.13. Effektusok

6.13.1. A gyakorlat célja

- Grafikus effektusok megismerése, elsajátítása
- Munkaidő: 1 labor (programozás)

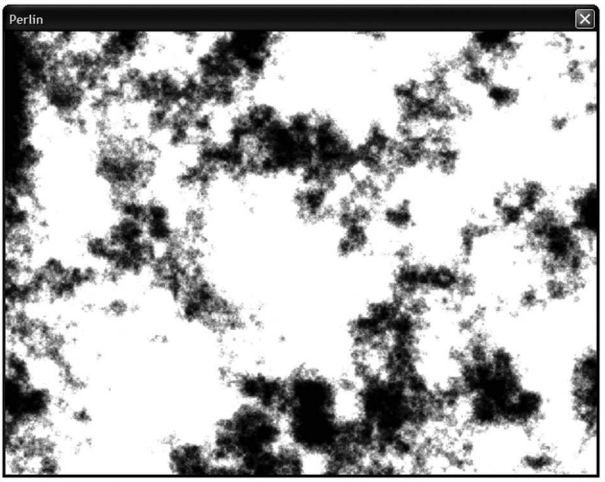
6.13.2. A feladat

Programozzuk le az előadáson tanult köd, felhő, hullám, tűz, alagút stb. effektusokat!

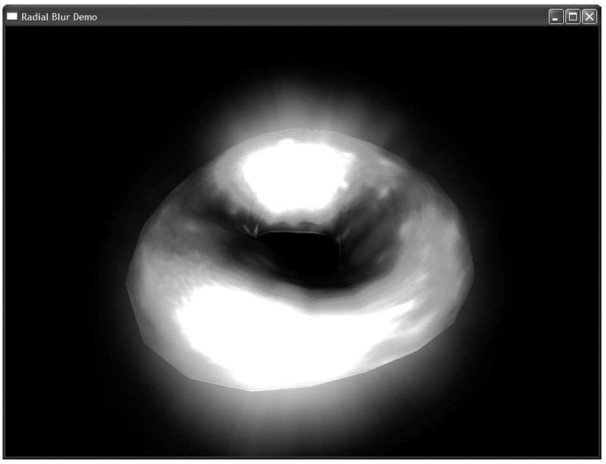
6.13.3. Útmutatás a megoldáshoz

Az előadáson tanult algoritmusokat kell alkalmazni.
Felhők generálására használjunk Perlin-zajt.

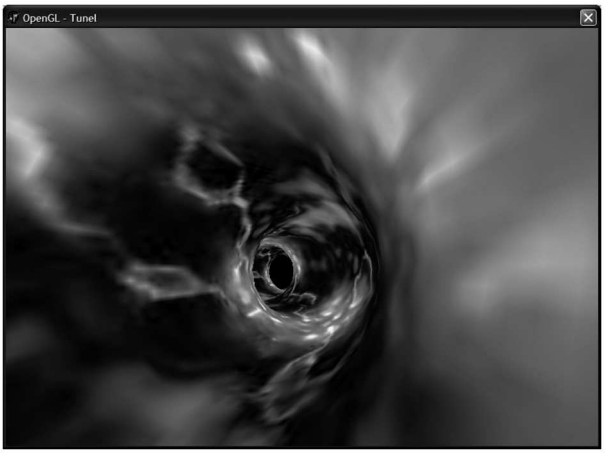
```
1  unit uPerlin;
2
3  interface
4
5  uses
6    Windows, Messages, SysUtils, Variants, Classes,
7    Graphics, Controls, Forms, Dialogs;
8
9  type
10   TfrmPerlin = class(TForm)
11     procedure FormPaint(Sender: TObject);
12   end;
13
14 var
15   frmPerlin: TfrmPerlin;
16   r1, r2, r3: integer;
17 implementation
18 {$R *.dfm}
19
20 function Noise(x, y: integer): extended;
21 var
22   n: integer;
23 begin
24   n := x+y*57;
25   n := (n shl 13) xor n;
26   Result := (1.0-((n*(n*n*r1+r2)+r3) and
27             $7fffffff)/1073741824.0);
28 end;
29
30 function Interpolate(x, y, a: extended):
31          extended;
32 var
33   val: extended;
```



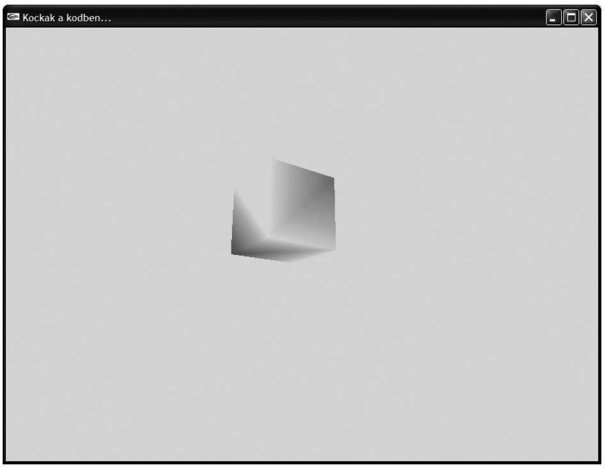
6.29. ábra. *Felhők Perlin-zajjal*



6.30. ábra. *Radial blur effektus*



6.31. ábra. *Alagút-effektus*



6.32. ábra. *Kocka a ködben*

```
34 begin
35   val := (1-cos(a*PI))*0.5;
36   Result := x*(1-val)+y*val;
37 end;
38
39 function Smooth(x, y: extended): extended;
40 var
41   n1, n2, n3, n4, i1, i2: extended;
42 begin
43   n1 := Noise(trunc(x), trunc(y));
44   n2 := Noise(trunc(x)+1, trunc(y));
45   n3 := Noise(trunc(x), trunc(y)+1);
46   n4 := Noise(trunc(x)+1, trunc(y)+1);
47   i1 := Interpolate(n1, n2, x-trunc(x));
48   i2 := Interpolate(n3, n4, x-trunc(x));
49   Result := Interpolate(i1, i2, y-trunc(y));
50 end;
51
52 function PerlinNoise2d(x, y: integer): extended;
53 var
54   frequency: extended;
55   persistence: extended;
56   octaves: integer;
57   amplitude: extended;
58   cloudCoverage: extended;
59   cloudDensity: extended;
60   lcv: integer;
61   total: extended;
62 begin
63   frequency := 0.015;
64   persistence := 0.60;
65   octaves := 20;
66   amplitude := 1.5;
67   cloudCoverage := 0.2;
68   cloudDensity := 1;
69   total := 0;
70   for lcv := 0 to octaves do
71     begin
72       total := total+Smooth(x*frequency,
73                y*frequency)*amplitude;
74       frequency := frequency*2;
75       amplitude := amplitude*persistence;
```

```
76      end;
77      total := (total+cloudCoverage)*cloudDensity;
78      if (total<0) then total := 0.0;
79      if (total>1) then total := 1.0;
80      Result := total;
81  end;
82
83  procedure TfrmPerlin.FormPaint(Sender: TObject);
84  var
85    i, j: integer;
86    v: byte;
87  begin
88    r1 := 1000+Random(10000);
89    r2 := 100000+Random(1000000);
90    r3 := 1000000000+Random(2000000000);
91    for i := 0 to frmPerlin.ClientWidth-1 do
92     for j := 0 to frmPerlin.ClientHeight-1 do
93      begin
94        v := trunc(PerlinNoise2D(i, j)*255);
95        frmPerlin.Canvas.Pixels[i, j] :=
96        RGB(255-v,255-v,255);
97      end;
98  end;
99
100 end.
```

6.14. Animáció

6.14.1. A gyakorlat célja

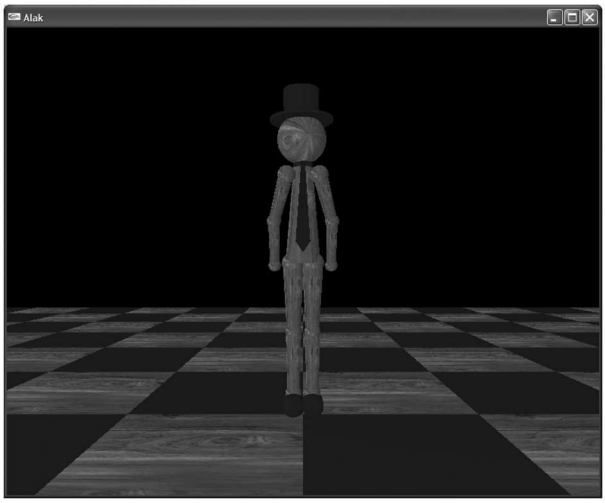
- Egy egyszerű karakter megtervezése és mozgatása
- `glPushMatrix`, `glPopMatrix` használata
- Objektumorientált programozás alkalmazása, adatstruktúrák használata
- Munkaidő: 2 labor (tervezés, programozás)

6.14.2. A feladat

Tervezzünk meg egy egyszerű karaktert és animáljuk (bármilyen mozgást végezhet: sétálhat, ugrálhat, táncolhat stb.)!

6.14.3. Útmutatás a megoldáshoz

Az előadáson tanult animációs technikákat kell alkalmazni.



6.33. ábra. *Egy egyszerű karakter*

SZAKIRODALOM

[1] ***
A perspektíva fejlődése 2.
http://mattort.fvt.hu/cikk.php?cikk=perspektiva2

[2] ***
A színek hatása a helyiségekre.
http://www.rs.hu/tanacsok/szinekhatasa.html

[3] AMMERAAL, Leendert
1992 *Programming Principles in Computer Graphics,* John Wiley & Sons Ltd.

[4] AVORNICULUI Mihály et alii
2004 *Bevezetés a gazdasági informatikába.* Erdélyi Tankönyvtanács, Kolozsvár

[5] BADEREDDIN, Ali
GLUI Window Template.
http://www.codeproject.com/KB/openGL/GLUI_Window_Template.aspx.

[6] BARNSLEY, Michael
1988 *Fractals Everywhere* Academic Press, Inc.

[7] BENKŐ Tiborné–BENKŐ László
1997 *Programozási feladatok és algoritmusok Turbo C és C++ nyelven.* ComputerBooks, Budapest

[8] BERKE József–HEGEDŰS Gy. Csaba–KELEMEN Dezső–SZABÓ József
1998 *Digitális képfeldolgozás.* Keszthelyi Akadémia Alapítvány, Pictron Kft.

[9] BERNOLÁK Kálmán
1981 *A fény.* Műszaki Könyvkiadó, Budapest

[10] BRESENHAM, J. E.
1965 *Algorithm for Computer Control of a Digital Plotter.* IBM Systems Journal 4(1), 25–30.

[11] BRUNT, Ineke–EISEMA, Joan
2004 *Gegrepen door het moment, Amsterdam in olieverf.* Stichting de Heeren Keyser, 5, Amsterdam

[12] BUDAI Attila
1999 *A számítógépes grafika.* LSI Oktatóközpont, Budapest

[13] BUSS, Samuel R.
2003 *3D Computer Graphics: A Mathematical Introduction with OpenGL.* Cambridge University Press

[14] CARLSON, Wayne E.
http://design.osu.edu/carlson/history/timeline.html

[15] CATMULL, E.
1974 *A Subdivision Algorithm for Computer Display of Curved Surfaces.* Ph.D. Thesis, Report UTEC-CSc-74-133, Computer Science Department, University of Utah, Salt Lake City, UT
[16] CHATZINIKOS, Fotis
2000 *A 3D Case Study Using OpenGL.* University of Hull
[17] CHIN, Norman et alii
1997 *The OpenGL Graphics System Utility Library.* Silicon Graphics
[18] CROW, Franklin C.
1977 *The aliasing problem in computer-generated shaded images.* In: Communications of the ACM, v.20 n.11, Nov.
[19] DIAMANDI, Ion
1994 *Cine ştie LOGO.* Editura Agni, Bucureşti
[20] DUŞA, Cristian
1995 *Modelare 3D.* PC Report 32, mai
[21] FAZAKAS Tibor
1997 *Háromdimenziós grafikus szerkesztés.* BBTE államvizsga-dolgozat (témavezető Robu Judit), Kolozsvár
[22] FIEGGEN, Ian W.
History of Computer Graphics. http://www. fieggen.com/ian/g_history.htm
[23] FIREBAUGH, Morris
Objects of Fractional Dimension. William C. Brown Publishers, Inc., http://www.firebaugh.com/FDA/Courses/CS.320/Week.11/Ch11a.www/Ch11a.html.
[24] FOLEY, J.–VAN DAM, A,–FEINER, S.–HUGHES, J.
1992 *Computer Graphics – Principles and Prectice.* Addison Wesley
[25] FOWLE, Brett
Setting Up OpenGL in an MFC Control. http://www.codeguru.com/cpp/g-m/opengl/openfaq/article.php/c10975/#more
[26] FÜZI János
1997 *3D grafika és animáció PC-n.* ComputerBooks, Budapest
[27] FÜZI János
1997 *Interaktív grafika.* ComputerBooks, Budapest
[28] GARCÍA, Oscar Xavier Chavarro
http://sophia.javeriana.edu.co/~ochavarr/computer_graphics_history/historia/
[29] GILLE, Bertrand
1970 *Alberti, Leone Battista. Dictionary of Scientific Bibliography.* New York, Charles Scribner's Sons
[30] GLASSNER, Andrew S.
1990 *Graphics Gems.* Cambridge Academic Press

[31] GOETHE, Johann Wolfgang
1983 *Színtan – Művészet és elmélet.* Corvina Kiadó, Budapest
[32] HORVÁTH Imre–JUHÁSZ Imre
1996 *Számítógéppel segített gépészeti tervezés.* Műszaki Kiadó, Budapest
[33] HOSCHEK, J.–LASSER, D.
1993 *Fundamentals of Computer Aided Geometric Design.* A K Peters Ltd., Wellesley, Massachusetts
[34] ***
2008 http://baldmonkeys.blogspot.com//09/disney-reuse.html
[35] ***
http://directxprog.uw.hu/
[36] ***
http://en.wikipedia.org/wiki/Cohen-Sutherland
[37] ***
http://en.wikipedia.org/wiki/POV-Ray
[38] ***
http://hu.wikipedia.org/wiki/CGI_(film)
[39] ***
http://hu.wikipedia.org/wiki/Maya_(program)
[40] ***
http://hu.wikipedia.org/wiki/Monitor
[41] ***
http://prog.hu/cikkek/493/A+GDI.html
[42] ***
http://prog.hu/cikkek/494/Fontok+bitterkepek+regiok.html
[43] ***
http://www.hemmy.net/2006/04/26/disney-animation-reuse/
[44] ***
http://www.lcg.ufrj.br/Cursos/GPUProg/GLSLfirst: *Starting with GLSL and OpenGL.*
[45] ***
http://www.opengl.org/
[46] ***
http://www.shughes.org/phantograms/Download_images.htm
[47] ***
https://renderman.pixar.com/
[48] ITTEN, Johannes
1978 *A színek művészete.* Corvina Kiadó, Budapest
[49] JUHÁSZ Imre
2003 *OpenGL,* jegyzet. Miskolci Egyetem
[50] JUHÁSZ Imre
1995 *Számítógépi geometria és grafika.* Miskolci Egyetemi Kiadó, Miskolc

[51] KANDINSZKIJ, Vaszilij
1987 A *szellemiség a művészetben.* Corvina Kiadó, Budapest
[52] KARDOS Lajos
1984 *Tárgy és árnyék. Tanulmányok a színlátás pszichológiai kutatása tárgyköréből.* Akadémiai kiadó, Budapest
[53] KÁTAI Zoltán
2008 *C-nyelv és programozás.* Debreceni Egyetem, Debrecen
[54] KÉPES János
1987 *Mikroszámítógépes grafika. Grafikai algoritmusok.* Műszaki Könyvkiadó, Budapest
[55] KESSENICH, John–BALDWIN, Dave–ROST, Randi
2006 *The OpenGL® Shading Language,* 3Dlabs, Inc. Ltd.
[56] KILGARD, Mark J.
1996 *The OpenGL Utility Toolkit (GLUT) Programming Interface.* Silicon Graphics
[57] KIRÁLY Sándor
1994 *Általános színtan és látáselmélet.* Nemzeti Tankönyvkiadó, Budapest
[58] KOVÁCS György
2003 Az anaglif eljárás. *Digitális Fotó Magazin,* Budapest, március.
[59] KOVÁCS Lehel István
2008 A számítógépes grafika története. *Műszaki Szemle – Technical Review* no. 44. p. 17–25, EMT, Kolozsvár
[60] KUBA Attila
Az OpenGL grafikai rendszer. http://www.inf.u-szeged.hu/oktatas/jegyzetek/KubaAttila/opengl/starthu.xml
[61] KUBA Attila
Számítógépes grafika. http://www.inf.u-szeged.hu/oktatas/jegyzetek/KubaAttila/grafika_html/szgrafika/.
[62] LAMING, Annette
1969 *Őskori barlangművészet*; *Lascaux.* Budapest, Gondolat Kiadó
[63] LÁSZLÓ József
1994 *A VGA-kártya programozása Pascal és Assembly nyelven.* ComputerBooks, Budapest
[64] LÁSZLÓ Judit Beáta
2004 *Számítógépes képfeldolgozás.* BBTE államvizsga-dolgozat (témavezető: Kovács Lehel István), Kolozsvár
[65] LŐRINC Pál–PETRICH Géza
1989 *Ábrázoló geometria.* Tankönyvkiadó, Budapest
[66] MOLDOVEANU, Fl. et alii
1996 *Grafică pe calculator.* Editura TEORA, Bucureşti
[67] NAGY Sándor–PERJÉS László
1996 *A számítógépes grafika.* LSI, Budapest

[68] NEMCSICS Antal
1982 A COLOROID-színrendszer esztétikailag egyenletes pszichometriai skáláinak kísérleti meghatározása. *Magyar Pszichológiai Szemle,* 38. 40–60.
[69] NEMCSICS Antal
1990 *Színdinamika. Színes környezet tervezése.* Akadémiai kiadó, Budapest
[70] NEWMAN, William M.–SPROULL, Robert L.
1985 *Interaktív számítógépes grafika.* Műszaki Könyvkiadó, Budapest
[71] NEWMAN, William M.–SPROULL, Robert L.
1973 *Principles of Interactive Computer Graphics.* McGraw-Hill Inc.
[72] NULAND, Sherwin B.
2001 *Leonardo Da Vinci.* Phoenix Press
[73] Nyisztor Károly
2005 *Grafika és Játékprogramozás DirectX-szel.* Szak Kiadó Kft., Budapest
[74] OSTERBY, Ole–ZLATEV, Zahari
1983 *Direct methods for sparse matrices.* Berlin
[75] PÁNYIK Árpád
2007 *A shader nyelvek lehetőségei.* Debrecen
[76] PERLIN, Ken
1985 An Image Synthesizer. *Computer Graphics* (SIGGRAPH 85 Proceedings) 19(3) July
[77] PHONG Bui-Tuong
1975 Illumination for Computer Generated Pictures. *Communications of the ACM,* 18(6) June
[78] PIRKÓ József
1988 *Perspektivikus grafika IBM PC-n Turbo Pascalban.* LSI Oktatóközpont, Budapest
[79] RADEMACHER, Paul
1999 *GLUI – A GLUT-Based User Interface Library*
[80] REVITZKY János
1985 *A számítógépes grafika felületkitöltő algoritmusai.* SZTAKI, Budapest
[81] RIESENFELD, R. F.
1973 *Applicatoins of B-Spline Approximation to Geometric Problems of Computer Aided Design.* PhD Dissertation, Syracuse University
[82] ROBERTS, Lawrence G.
1965 *Homogenous Matrix Representation and Manipulation of N-Dimensional Constructs.* MS-1505, MIT Lincoln Laboratory, Lexington, Mass.

[83] ROST, Randi J.
2006 *OpenGL Shading Language* (Second Edition). Addison-Wesley Professional
[84] ROST, Randi J.
2005 *OpenGL Shading Language.* Addison-Wesley Professional
[85] SAUPE, J.–YUNKER, P.
1991 Fractals for the classroom. *Strategic activities,* Vol. 1., Springer-Verlag
[86] SEGAL, Mark–AKELEY, Kurt
1998 *The OpenGL Graphics System: A specification (Version 1.2).* Silicon Graphics
[87] SHOAFF, William D.
A Short History of Computer Graphics.
http://www.cs.fit.edu/∼wds/classes/graphics/History/history/history.html#SECTION00020000000000000000
[88] SHOAFF, William D.
Cohen-Sutherland Algorithm Source Code.
http://www.cs.fit.edu/∼wds/classes/cse5255/thesis/lineClip/ code.html
[89] SHREINER, Dave–WOO, Mason–NEIDER, Jackie–DAVIS, Tom
2005 *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2* (5th Edition). Addison-Wesley Professional
[90] SUTHERLAND, I. E.
1963 *Sketchpad: A man-machine graphical communication system.* Summer Joint Computer Conference
[91] SZABÓ József
2005 *Lineáris leképezések.* Debreceni Egyetem, Debrecen
[92] SZABÓ L. I.
1997 *Ismerkedés a fraktálok matematikájával.* Polygon, Szeged
[93] SZÉKELY Vladimír–POPPE András
1992 *A számítógépes grafika alapjai IBM PC-n.* ComputerBooks, Budapest
[94] SZIRMAY-KALOS László–ANTAL György–CSONKA Ferenc
2006 *Háromdimenziós grafika, animáció és játékfejlesztés.* Computerbooks, Budapest
[95] Thimbleby, Harold W.–INGLIS, Stuart–WITTEN, Ian H.
1994 Displaying 3D Images: Algorithms for Single Image Random Dot Stereograms. *IEEE Computer,* 27 (10), pp. 38–48.
[96] VARDY, Andrew
Point and line clipping.
http://www.cs.mun.ca/∼av/courses/cg/notes/raster_clip.pdf
[97] VASS Gergely
2002 Számítógépes grafika IV. rész: animáció. *Videó Praktika Magazin,* Digitális Videó, 2001–

[98] VÉGVÁRI Lajos
1961 *Giotto, 1266–1337.* A Képzőművészeti Alap Kiadóvállalata, Budapest
[99] VICSEK M.–VICSEK T.
1993 Fraktálok a fizikában. *Fizikai Szemle* XLIII. évf. 2. szám
[100] WHITTED, Turner
1980 An improved illumination model for shaded display. *Communications of the ACM,* V. 23 nr. 6, p. 343–349, June
[101] WILLIAMS, Lance
1983 Pyramidal Parametrics. *Computer Graphics* (SIGGRAPH 83 Proceedings) 17(3), July
[102] WRIGHT, Richard S.–SWEET, Michael
1998 *The OpenGL SuperBible.* Waite Group Press
[103] YOUNG, Michael J.
1998 *Visual C++ 6 mesteri szinten.* Kiskapu Kiadó, Budapest

FOGALOMTÁR

γ-kalibrálás [gamma-calibration, calibrare-gamma]: színkorrekció színterek közötti átalakításkor

ablak [window, fereastră]: téglalap alakú terület Windows alatt, egy alkalmazás képernyője

additív színkeverés [additive color mixing, sinteza aditivă a culorilor]: monokromatikus fénysugarak összekeverése

AGP *Accelarated Graphics Port*: videokártya csatlakoztató szabvány

akromatikus fény [achromatic light, lumină acromatică]: a teljes spektrumban egyenletes energiával sugárzó fényforrás. Színérzete a fehér

alakfelismerés [form recognition, recunoaşterea formelor]: a raszteres képeken lévő grafikus objektumok azonosítása

alpha-blending: technika az átlátszóság megvalósítására

ambiens világítás [ambient light, lumină ambientă]: háttérvilágítás, környezeti fény

anaglif eljárás [anaglyph, anaglif]: a bal és a jobb szem helyzetének megfelelően felvett két képet kiegészítő színekkel (pl. az egyik kép vörös, a másik cián árnyalatú): másoljuk egymásra

animáció [animation, animaţie]: olyan filmkészítési technika, amely élettelen tárgyak (többnyire bábok), vagy rajzok, ábrák stb. „kockázásával" olyan illúziót kelt a nézőben, mintha az egymástól kis mértékben eltérő képkockák sorozatából összeálló történésben a szereplők megelevenednének vagy élnének

anyag [material, materie]: számítógépes grafikában egy test felületének a jellemzői

árnyaló nyelv [shading language, limbaj shading]: a GPU programozási nyelve

árnyalt [solid, solidă]: a testek felületének ábrázolása, a határoló felületek kitöltött képét rajzoljuk ki

átméretezés [scaling, scalare]: egy objektum nagyítását vagy kicsinyítését, torzítását jelenti

AVI *Audio Video Interleave*: audio-video-összefésüléses állományformátum

axonometria [axonometry, axonometrie]: a koordináta-rendszer tengelyeire való felmérés

befoglaló test [bounding volumes, corp circumscris]: olyan szabályos 2D alakzat vagy 3D test, amely magában foglal egy másik, kevésbé szabályos testet

BGI *Borland Graphic Interface*: a Borland grafikus meghajtói

BMP *Bitmap* (bittérkép): a Microsoft által kifejlesztett képformátum

BOB *Blitter OBject* – olyan 256 színű, téglalap alakú grafikai objektumok, melyek tetszőlegesen mozgathatóak, eltüntethetőek és megjeleníthetőek

Bresenham-algoritmus [Bresenham's line algorithm, algoritmul lui Bresenham]: vonalrajzoló algoritmus

bump-maping: technika a göröngyös térhatású felületek elkészítésére

centrumpont [vanishing point, punct de fugă]: perspektivikus vetítés esetén az a pont, ahová a képsíkra merőlegesen futó párhuzamos vonalak összetartanak, összefutnak

CGI *Computer-Generated Imagery*: számítógép által generált képek, animációs karakterek

CMYK *Cian, Magenta, Yellow, blacK*: a színes nyomtatásban használt színmodell

CRT *Cathode Ray Tube*: katódsugárcső

csúcspont [vertex, vârf]: a vektorgrafika alapegysége, egy pontra vonatkozó információk összessége (koordináták, szín stb.)

diffúz fény [diffuse light, lumină difuză]: szórt fény

dimetrikus axonometria [dimetric axonometry, axonometrie dimetrică]: kétméretű →*axonometria*. A z tengelyt megtartjuk függőlegesnek, a vízszintes tengelyirányokat pedig 1:8 és 7:8 arányú lejtéssel rajzoljuk meg. A rövidülések: $q_x = q_z = 1$, $q_y = 0{,}5$

DirectX: a Microsoft által fejlesztett, csak Windows alatt használható rutingyűjtemény. A hardver direkt elérését valósítja meg

direkt kinematika [direct – forward kinematics, chinematică directă]: olyan animációs technika, amely segítségével egy csont/ízület-rendszert a központtól a végtagok felé mozgatunk

doboz-dimenzió [box dimension, dimensiunea de acoperire]: fraktálok dimenziója, amelyet úgy határozunk meg, hogy egységnyi méretű négyzetekkel, kockákkal lefedjük az alakzatot, majd megszámoljuk ezeket

DPI *Dots Per Inch*: a →*felbontás* mértékegysége

drótvázas [wireframe, cadru de sârmă]: olyan vektorgrafikus ábrázolási mód, amelyben a testeket csak az éleikkel ábrázoljuk

D-SUB: videokártya–képernyő közötti átviteli szabvány (analóg)

DVI *Digital Visual Interface*: videokártya–képernyő közötti átviteli szabvány (digitális)

ecset [brush, pensulă]: foltok festésére alkalmas eszköz

elemi elsődleges színek [primary colors, culori primare]: CMY színmodell esetén az a három szín (cián, magenta, sárga), amelyből az összes többi kikeverhető

élsimítás [anti-aliasing, antialiasing]: a nagy kontrasztú széleket, ferde vonalakat puhává, simává teszi

eltolás [translation, translație]: egy alakzat minden pontját egy adott irányba, adott távolsággal mozdítunk el
enyészpont: lásd →*centrumpont*

felbontás [resolution, rezoluție]: a képernyő által megjeleníthető pixelek száma: egy képernyősorban található képpontok számának és a képernyősorok számának szorzata
felhasználói grafikus felületek [graphical user interface, interfețe grafice cu utilizatorul]: operációs rendszerek, alkalmazások grafikus felülete, eseményorientált, grafikus kontrollok, a felhasználóval való magasabb szintű interakció
fény [light, lumină]: az elektromágneses sugárzásnak az a része, amelyet a szem érzékelni képes és amelynek a hatására az agyban képérzet alakul ki
fényerősség: lásd →*világosság*
fényforrás [light source, sursă de lumină]: minden olyan entitás (természetes és mesterséges egyaránt), amely látható fény előállítására szolgál
fénysugár [light ray, rază de lumină]: egyenes vonalban haladó keskeny fény
fénytörés [refraction, refracție]: egy új közeg határához érkező fény egy része behatol az új közegbe, és eközben megváltozik a terjedésének iránya és sebessége
fényvisszaverődés [reflection, reflexie]: sima átlátszatlan felületről a fény visszaverődik
ferde vetítés [clinogonal projection, proiecție oblică]: az egymással párhuzamos vetítősugarak nem merőlegesek a képsíkra (klinogonális)
ferdítés: lásd →*torzítás*
flipping: memóriacímek cseréje
FMV *Full Motion Video*: mozgókép
font: betűtípus
forgatás [rotation, rotare]: egy alakzat vagy test minden pontjának elmozdítása egy adott pont körül, egy adott szöggel, egy adott irányban
fotopigmens [photopigment, fotopigment]: a kémiai reakciókért felelős festékanyag a szemben
fotorealisztikus [photorealistic, fotorealistic]: a vektorgrafikus modelltérbeli jelenetről olyan minőségű képet állítunk elő, amely teljesen valószerű, a valós világról készített fényképtől nem lehet megkülönböztetni
fraktál [fractal, fractal]: önhasonló, végtelenül komplex, törtdimenziós matematikai alakzatok, amelyek változatos formáiban legalább egy felismerhető (tehát matematikai eszközökkel leírható) ismétlődés tapasztalható

GDI *Graphic Device Interface*: a Windows grafikus rendszere
generatív alapszínek [additive primary colors, culori primare aditive]: RGB színmodell esetén az a három szín (vörös, zöld, kék), amelyből az összes többi szín érzete kikeverhető

generatív számítógépes grafika [interactive computer graphics, grafică interactivă pe calculator]: a képi információ tartalmára vonatkozó adatok és algoritmusok alapján modellek felállítása, képek megjelenítése

GIF *Graphic Interchange Format*: a CompuServe által kifejlesztett képformátum

GLU *OpenGL Utility Library*: magas szintű OpenGL függvénykönyvtár

GLUT *OpenGL Utility Toolkit*: magas szintű OpenGL függvénykönyvtár

glyph: a →*karakter*ek grafikus képe

GPU *Graphics Processing Unit*: a videokártya processzora

grafika[1] γράφω **(gráfho),** γραπικός **(graphikós)** vésni , véset

grafika[2] [graphics, grafică]: a képzőművészet azon ága, amelyhez a sokszorosítási eljárással készült, de eredetinek tekinthető alkotások tartoznak, illetve azok az egyszeri alkotásokról (pl. festmény) sokszorosító eljárással készült reprodukciók, amelyek nem tekinthetők egyedi alkotásnak. A felület kitöltése többnyire vonalak segítségével történik, szemben a festészettel, ahol foltokkal

grafikus bemutatók [bussines graphics, prezentări grafice]: az üzleti életben, tudományban, közigazgatásban stb. bemutatott grafikus alapú prezentációk elkészítése a vizuális információ átadásának céljából

grafikus csővezeték [graphics pipeline, pipeline grafic]: grafikus primitíveken végzett elemi műveletek sorozata (transzformációk, vetítés, vágás stb.) a színtér objektumairól készített pixeles kép előállítása céljából. A műveleteket a grafikus hardver csővezetékben végzi

graftál [graftal, graftal]: egyszerű szabályokból iteratív eljárással létrehozott alakzatok, amelyek növényeket modelleznek

harmadlagos szín [tertiary colors, culori terțiale]: az →*elemi elsődleges* és a →*másodlagos színek* keverésével jönnek létre, ilyen szín pl. a barna

HDMI *High-Definition Multimedia Interface*: videokártya–képernyő közötti átviteli szabvány (digitális)

holográfia [holography, holografie]: a fény hullámtermészetén alapuló olyan képrögzítő eljárás, amellyel a tárgy struktúrájáról tökéletes térhatású kép hozható létre

homogén koordináták [homogeneous coordinates, coordonate omogene]: az n dimenziós tér egy pontjának helyzetét n+1 koordináta segítségével írják le, oly módon, hogy egy tetszőleges nullától eltérő értékkel az eredeti n dimenziós térben értelmezett koordinátákat megszorozzuk, és ezt a konstansot tekintjük az n+1-dik koordinátának

IFS *Iterated Function System*: iterált függvényrendszer

inverz kinematika [inverse kinematics, chinematică inversă]: olyan animációs technika, amely segítségével egy csont/ízület-rendszernek a végpontjait mozgatjuk, a többi pont elmozdulását pedig a számítógép határozza meg

izometrikus axonometria [isometric axonometry, axonometrie izometrică]: egyméretű →*axonometria*. A koordinátatengelyek egymással 120–120˚-os szöget zárnak be. A rövidülések egyenlők: $q_x = q_y = q_z = 1$

JPEG *Joint Photographic Experts Group*: képformátum

karakter[1] [character, caracter]: az ASCII-táblázat egy eleme (lehet betű, számjegy, írásjelek, speciális karakterek stb.)

karakter[2] [character, caracter]: animáció esetén a megtervezett figura, bábu stb., amelyet animálunk

kavalier axonometria [cavalier axonometry, axonometrie cavalieră]: frontális →*axonometria*. A z tengely függőleges helyzetű. Az x tengely a z tengelyre merőleges, és mindkét tengelyre a méreteket valódi nagyságban rajzoljuk. Az y tengelyt a vízszinteshez képest 135˚-os lejtéssel rajzoljuk, és a méreteket 1:2 arányú rövidüléssel mérjük fel

képelemzés [picture analysis, analizarea imaginilor]: lásd →*alakfelismerés*

képernyő [monitor, monitor]: a számítógép grafikus megjelenítő perifériája

képfeldolgozás [image processing, procesarea imaginilor]: mindazon számítógépes eljárások és módszerek összessége, amelyekkel a számítógépen tárolt képek minőségét valamilyen szempont szerint javítani lehet

képpont: lásd →*pixel*

kiegészítő színek [complementary colors, culori complementare]: két szín, amelyeknek keveréke akromatikus színérzetet (rendszerint szürkét) hoz létre

koordináta [coordinate, coordonată]: független méretek, amelyek megadják egy tetszőleges pont helyzetét a térben vagy a síkban

koordináta-rendszer [coordinate system, sistem de coordonate]: egy sík vagy egy tér, amelyben egy kezdőpontot és tengelyeket jelölünk ki, és ezektől mérhetők a →*koordináták*

kulcs animáció [keyframe animation, animație bazată pe cadre cheie]: a mozgást kulcspozíciók megadásával határozzuk meg

látás [eyesight, vedere]: a vizuális információk feldolgozása, amelynek fő célja a tárgyak azonosítása, és azok közvetlenül nem észlelhető tulajdonságainak felismerése, illetve a cselekvés vezérlése

LCD *Liquid Crystal Display*: folyadékkristály

machinima: a videojátékok szoftverének magját alkotó game engine használata a nem interaktív filmek renderelésére

másodlagos színek [secondary colors, culori secundare]: az →*elemi elsődleges színek* keverésével kapjuk: *zöld*, *narancs* és *lila*

megvilágítási modell [light model, model de iluminare]: az optikai és felületi kölcsönhatások modellezése

merőleges vetítés (*orthogonal proiection, proiecție ortogonală*): az egymással párhuzamos vetítősugarak merőlegesek a képsíkra (ortogonális)
mip-maping: olyan technika, amely a távolság arányában többféle részletességi szintű textúrát alkalmaz
modell [model, model]: a valóság leírása vektorgrafikus és raszteres objektumok segítségével
modelltér [scene, scenă]: a matematikailag modellezett 3D objektumok helyszíne
morphing: az alakváltás animációs technikája, egy képet fokozatosan átvisz egy másikba
motion blur: mozgó objektumok körvonalainak elmosása
motion capture: a színészek testére fényvisszaverő pontokat (vagy szenzorokat) helyeznek, melyeket több kamera lekövet. A programok ezen pontok alapján milliméterpontosan rekonstruálják a valódi mozgást

normális: lásd →*normálvektor*
normálvektor [surface normal, normală la suprafață]: az az egységnyi hosszúságú vektor, amely az adott pontban merőleges a felületre (a felület érintősíkjára)
NURBS *Non-Uniform Rational B-Splines*: nem egyenletes elosztású racionális B-spline görbe

objektumtér: lásd →*modelltér*
OpenGL: platform- és operációs rendszer független grafikus API

paletta [palette, paletă]: színek összessége
párhuzamos vetítés [parallel projection, proiecție paralelă]: a vetítősugarak egymással párhuzamosak
PCI-Express: videokártya csatlakoztató szabvány
PDF *Portable Document Format*: az Adobe által kifejlesztett platformfüggetlen dokumentumformátum
PDP *Plasma Display Panel*: plazmakijelző
Perlin-zaj [Perlin noise, zgomot Perlin]: R^n-en értelmezett $(f : R^n \to [-1,\ 1])$, az egész számokban csomópontokat képző rácshoz igazított pszeudovéletlen spline függvény, amely a véletlenszerűség hatását kelti, de ugyanakkor rendelkezik azzal a tulajdonsággal, hogy azonos bemeneti értékekre azonos függvényértéket térít vissza
perspektíva [perspective, perspectivă]: tárgyak térbeliségi érzetét keltő ábrázolásmód sík felületen
pigmens [pigment, pigment]: a természetben előforduló festékanyag
pixel [pixel – picture element, pixel]: a monitoron ábrázolt kép legkisebb egysége

pixelgrafika [raster graphics, grafică raster]: olyan digitális kép, ábra, melyen minden egyes képpontot (→*pixel*t): önállóan definiálunk
PNG *Portable Network Graphics*: képformátum
primary surface: a képernyőn látható felület (olyan memóriaterület a videokártya memóriájában, amely képeket, vizuális információt tartalmaz). Amit ebbe beleírunk, az azonnal megjelenik a képernyőn

radiosity: egyfajta →*renderelés*i eljárás
rajzvászon [canvas, pânză]: az a grafikus objektum, amelyre →*toll*al vagy →*ecset*tel rajzolni tudunk
raszter: lásd →*pixel*
rasztergrafika: lásd →*pixelgrafika*
reflexió: lásd →*fényvisszaverődés*
refrakció: lásd →*fénytörés*
régió [region, regiune]: tetszőleges alakú, de mindenképpen zárt alakzatok, amelyek közvetlenül nem kerülnek megjelenítésre, de a rajzoló műveletek hatókörét az adott alakzaton belülre korlátozzák
renderelés [rendering, renderare]: a vektorgrafikus objektumok árnyalt megjelenítése. Képalkotás
RGB *Red, Green, Blue*: a képernyő színmodellje
RGBA: az →*RGB* színmodell kiegészítve az A (alfa): komponenssel, amely az átlátszóságot, áttetszőséget jelöli
rigging: egy →*karakter* csont/ízületrendszerének az elkészítése
rövidülés [foreshortening factor, factor de prescurtare]: az a szorzószám, amellyel az eredeti térbeli koordinátát megszorozva az axonometrikus vetület megfelelő távolsága lesz

shader: GPU program
skálázás: lásd →*átméretezés*
skinning: bőr ráhúzása egy →*karakter* csont/ízületrendszerére
spekuláris [specular light, lumină specular]: tükrözött fény
spline-görbe [spline curve, curbă spline]: szakaszosan parametrikus polinomokkal leírható görbe
stop-motion vagy frame-by-frame: olyan animációs technika, amely segítségével egy apránként elmozgatott tárgyat összefilmeznek, s így folytonos mozgás áll elő
sugárkövetés [ray-tracing, urmărirea razei de lumină]: a számítógépes képalkotás (→*renderelés*) egy olyan módszere, amely a fény útját, annak fizikai tulajdonságait figyelembe véve szintetizálja a képet

számítógépes játékok [computer games, jocuri pe calculator]: olyan játékok, amellyel a játékos egy felhasználói felületen keresztül lép kölcsönhatásba és arról egy kijelző eszközön keresztül kap visszajelzéseket

számítógéppel segített grafika [computer aided graphics, grafică asistată pe calculator]: a számítógép bevonása ábrázolásmódok, számítások, folyamatok megkönnyítésére, pl. függvényábrázolás, nyomdai grafikai munkálatok, sokszorosítás, diagramkészítés, illusztrátorok stb.

számítógéppel segített tervezés és gyártás [computer aided design and manufacturing, proiectare şi fabricare asistată de calculator]: olyan, számítógépen alapuló eszközök összessége, amely a mérnököket és más tervezési szakembereket tervezési tevékenységükben segíti

színárnyalat [hue, nuanță]: szín, a szemünkbe jutó fény hullámhosszának függvénye

színillesztés [color matching, potrivire de culoare]: a színérzet előállítása színkeveréssel, az árnyalatok kódolása valamilyen színmodellben, annak érdekében, hogy a színek azonosítása, kikeresése hatékonyan valósuljon meg

színleképzés [tone mapping, ajustarea tonalității culorii]: lásd →*színillesztés*

színmodell [color model, model de culoare]: a digitális képeken látható és felhasználható színeket írja le

színrebontás [color separation, separarea culorilor]: az a folyamat, amikor a színeket alapszínekre bontjuk, vagyis meghatározzuk, hogy minden egyes színben mennyi R, G, B vagy C, M, Y, K komponensmennyiség van

színtelítettség [saturation, saturație]: a szín fehérrel való felhígítottságának, fátyolosságának mértéke

színtér[1] [color space, spațiu de culoare]: a színmodell egy változata, amely speciális színárnyalatokkal, színtartománnyal rendelkezik

színtér[2]**:** lásd →*modelltér*

szöveg- és kiadványszerkesztés [desk top publishing, tehnoredactare computerizată]: számítógéppel segített nyomdai kiadványszerkesztés, speciális képek, betűtípusok, emblémák, logók, reklámfigurák elkészítése

sztereó fotó [stereo photography, fotografie stereo]: speciális kétobjektíves fényképezőgéppel készített fénykép, amely 3D hatást kelt

sztereogram [stereogram, stereogramă]: számítógéppel előállított speciális 3D hatást keltő kép

szubsztraktív színkeverés [subtractive color mixing, sinteza substractivă a culorilor]: festékek keverése

szürkeárnyalatos kép [grayscaled image, imagine în tonuri de gri]: a színárnyalatokat szürke tónusokkal ábrázoljuk, így a fekete-fehér fényképekhez hasonló kép jön létre

téglalap [rectangle, dreptunghi]: Windows alatt egy kontroll felülete, egy bal felső, jobb alsó sarokpárossal azonosított terület

teknőc-grafika [turtle graphics, grafică LOGO]: a *LOGO* nyelv grafikai rendszere

térhatás [depth cueing, efect de spațiu]: a 2D-ben ábrázolt kép olyan hatást kelt, mintha 3D-s valós tájat szemlélnénk

térképészeti információs rendszerek [geographical information system, sisteme informatice geografice]: a térképek számítógépes feldolgozását lehetővé tevő rendszerek

textúra [texture, textură]: a valósághűség érdekében a →*modelltér* objektumaira ráfeszített kép

TFT *Thin Film Transistor*: vékonyfilm tranzisztor

toll [pen, peniță]: vonalas ábrák előállításának eszköze

torzítás: lineáris leképzés, lerögzíti a pontokat az egyik tengely szerint, a másik tengely szerint viszont eltolja őket a tengelyhez mért távolságukkal arányosan (skew, shear, transvection, înclinare)

TrueColor: az →*RGB* színmodellben ábrázolt 16 777 216 színárnyalat

tükrözés [reflection – mirroring, oglindire]: egy alakzat vagy test összes pontjának az ellentétes térrészben történő ábrázolása

vágás [clip, tăiere]: egy kép azon részeinek elhagyása, amelyek nem férnek be a →*viewport*ba

váltottsoros megjelenítés [interlaced, afişare întrețesută]: olyan ábrázolási mód, amely szerint a teljes kép nem egyszerre, hanem meghatározott részekben kerül megjelenítésre, például először a páros, majd a páratlan sorszámú pixelsorok kirajzolásával történik meg az ábrázolás

vektorgrafika [vector graphics, grafică vectorială]: az az eljárás, melynek során geometriai primitíveket (rajzelemeket), mint például pontokat, egyeneseket, görbéket, sokszögeket használunk képek leírására

vetítés [projection, proiectare]: azok a dimenzióveszteséggel járó ponttranszformációk, melyeknél bármelyik képpont és a neki megfelelő összes tárgypont egy egyenesen helyezkedik el

VHS *Video Home System*: videoszabvány

videokártya [graphics card – video card, placă video]: a számítógép hardver része, feladata, hogy a számítógép által küldött képi információkat feldolgozza, és egy megjelenítő egység számára (→*képernyő*) értelmezhető jelekké alakítsa

viewport: az ablakon belüli rész, ahová rajzolunk

világosság [brightness, strălucire]: a szemünkbe érkező fényenergia mennyisége

virtuális valóság [virtual reality, realitate virtuală]: olyan technológiák összessége, amely különleges eszközök révén a felhasználó szoros interakcióba kerül a grafikus világgal, mintegy részévé válik

WYSIWYG *What You See Is What You Get*: ALAKHŰ (*Azt Látod, Amit Kapsz, HŰen*)

z-buffer: a látható felületek meghatározásának algoritmusa. Minden pixelhez hozzárendelünk egy z értéket, amely megmondja, hogy milyen mélyen helyezkedik el, ezáltal kiszámíthatók a takarások

ÁBRÁK JEGYZÉKE

ABSTRACT

Man has always wanted to illustrate his thoughts in order to share them. It is most likely that the first cave paintings were born with this purpose 30,000 years ago.

With the help of computer graphics, can create an image of our a computer schematic thoughts and thus it can change our ideas to pictures, series of pictures, cartoons, films. The computer "takes a photo" of the digital model described with digits, creates a photorealistic picture with the help of generative computer graphics, that we can look at and change according to our demands or we can refine the model that will be photographed again.

The first part of the book formulates the aim of computer graphics, it presents its history and then it creates models according to the procession of real life (human sight, optics, lighting, and shading).

The second part presents the bases of computer graphics. Following the presentation of graphic hardware and software, the mathematical background and the description of fundamental algorithms are given: coordinate systems, transformations, methods of projection, ray tracing, shading and clipping algorithms precede the world of fractals and computer animation.

The third part presents the possibilities of DOS graphic from the diagrams of characters in text mode, from BOB programming through *LOGO's* turtle-graphic and until the Borland graphic system.

The fourth part discusses the possibilities of Windows considering GDI graphics (especially through the achievements of Borland Delphi) and DirectX.

The fifth part presents Open GL platform independent graphic system programming in *Visual C++* and *Borland Delphi*, then it shows the possibilities of GLU, GLUT, GLUI.

The sixth part contains 14 exercises introducing us into the world of computer graphics.

The book ends with 103 bibliography items and a Hungarian–English–Romanian thesaurus.

The book is based on the schedule of *Computer graphics* taught at Sapientia University – Faculty of Târgu-Mureş, but it can be used by pupils as well or anybody wishing to deepen their knowledge in this field, to prepare themselves for special informatics contests (for example: KovInfo or render.hu) which have as their main topic computer graphics.

Through the topics presented above, this book is an introduction into the magic world of computer graphics, where the palace is built of cubes, pyramids, and cones, there are fractal clouds on the RGB(0, 124, 195) colour sky, and the youngest prince sets out to find the dragon moved by inverse kinematics, under

rocks generated with Perlin noise, on a grass-textured field among Barnsley-ferns.

REZUMAT

Omul a dorit întotdeauna să-şi vizualizeze gândurile, putând astfel să le împartă cu cei din jur. De aceea, probabil, au luat fiinţă acum 30 000 de ani primele picturi rupestre. În ziua de azi, cu ajutorul graficii pe calculator putem obţine din acesta o maşină care poate sintetiza imagini din schiţele gândurilor noastre şi, în acest fel, imaginaţia noastră capătă viaţă pe monitor, hârtie, film etc.

Calculatorul „fotografiază" modelul digital, descris prin şiruri de numere, iar grafica generativă sintetizează o imagine fotorealistică, pe care o admirăm şi, pe baza căreia, putem personaliza, adapta, finisa modelul din care putem obţine iarăşi o imagine sintetizată.

Cartea este structurată în şase părţi după cum urmează:

- În prima parte este definit scopul graficii pe calculator, este prezentată istoria acesteia, după care este realizat un model, având la bază procedee din lumea reală (vederea, optica, iluminarea, umbrirea).
- În partea a doua sunt descrise bazele graficii pe calculator şi anume: prezentarea hardului şi a softului grafic, bazele matematicii şi algoritmii fundamentali: sisteme de coordonate, transformări, proiectări, algoritmul raytracing, algoritmi de umbrire, tăiere, precedate de lumea fractalilor şi a animaţiei pe calculator.
- Partea a treia prezintă posibilităţile graficii sub MS DOS, de la imaginile efectuate din caractere şi programare BOB, prin sistemul „Turtle" al limbajului *LOGO*, până la posibilităţile de nivel superior ale graficii Borland.
- Partea a patra expune posibilităţile graficii sub Windows: GDI (prin *Borland Delphi*) şi DirectX.
- Partea a cincea prezintă sistemul independent de platformă OpenGL. Posibilităţile GLU, GLUT, GLUI sunt programate în *Visual C++* şi *Borland Delphi*.
- Partea a şasea ne introduce, prin 14 exemple şi exerciţii, în lumea graficii pe calculator.

Cartea se încheie cu enumerarea a 103 de referinţe bibliografice şi cu un dicţionar explicativ maghiar–englez–român.

Cartea respectă programa disciplinei *Grafică pe calculator,* care se predă la Facultatea de Ştiinţe Tehnice şi Umaniste din Târgu Mureş, din cadrul Universităţii Sapientia, adresându-se studenţilor, elevilor de liceu, cititorilor pasionaţi de aprofundarea temei şi celor care doresc să participe la concursuri speciale de grafică pe calculator (ex. KovInfo sau renderer.hu).

Prin tematica prezentată, cartea ne introduce în lumea miraculoasă a graficii pe calculator, unde castelul fermecat este construit din cuburi, cilindri, sfere şi conuri, iar dincolo de norii fractali străluceşte cerul de culoarea RGB(0, 124, 195), şi, în faţa peisajului texturat, prinţul cel mic caută dragonul animat, cu ajutorul chinematicii inverse, printre stânci generate cu zgomot-Perlin şi ferigi-Barnsley.

A SZERZŐRŐL

Kovács D. Lehel István 1975-ben született Brassóban. Elemi iskolai tanulmányait Négyfaluban, a középiskolát Brassóban végezte. 1997-ben szerzett egyetemi oklevelet a Babeş–Bolyai Tudományegyetem informatika szakán. 1998-ban magiszteri oklevelet szerzett, ezt követően doktori tanulmányokat folytatott Budapesten és Kolozsváron. Doktori disszertációját 2006-ban védte meg Kolozsváron.

1997-től a kolozsvári Babeş–Bolyai Tudományegyetem, 2007-től a marosvásárhelyi Sapientia – Erdélyi Magyar Tudományegyetem oktatója.

Főbb kutatási területei: fordítóprogramok, értelmezők, programozási nyelvek, számítógépes rendszerek tervezése, számítógépes grafika.

2009-ig 54 szakmai konferencián vett részt, 6 szakkönyv szerzője, 3 kötet szerkesztője, 22 elismert folyóiratban közölt szakcikk, 81 tudománynépszerűsítő cikk szerzője, 3 szaklap szerkesztője, 11 kutatási program résztvevője, vezetője.

A SAPIENTIA –
ERDÉLYI MAGYAR TUDOMÁNYEGYETEM JEGYZETEI

Bege Antal
: Számelméleti feladatgyűjtemény. Marosvásárhely, Műszaki és Humán Tudományok Kar, Matematika–Informatika Tanszék. 2002.

Bege Antal
: Számelmélet. Bevezetés a számelméletbe. Marosvásárhely, Műszaki és Humán Tudományok Kar, Matematika–Informatika Tanszék. 2002.

Vofkori László
: Gazdasági földrajz. Csíkszereda, Csíkszeredai Kar, Gazdaságtan Tanszék. 2002.

Tőkés Béla–Dónáth-Nagy Gabriella
: Kémiai előadások és laboratóriumi gyakorlatok. Marosvásárhely, Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2002.

Irimiaş, George
: Noţiuni de fonetică şi fonologie. Csíkszereda, Csíkszeredai Kar, Humán Tudományok Tanszék. 2002.

Szilágyi József
: Mezőgazdasági termékek áruismerete. Csíkszereda, Csíkszeredai Kar, Gazdaságtan Tanszék. 2002.

Nagy Imola Katalin
: A Practical Course in English. Marosvásárhely, Műszaki és Humán Tudományok Kar, Humán Tudományok Tanszék. 2002.

Balázs Lajos
: Folclor. Noţiuni generale de folclor şi poetică populară. Csíkszereda, Csíkszeredai Kar, Humán Tudományok Tanszék. 2003.

Popa-Müller Izolda
Műszaki rajz. Marosvásárhely, Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2004.

Fodorpataki László–Szigyártó Lídia–Bartha Csaba
Növénytani ismeretek. Kolozsvár, Természettudományi és Művészeti Kar, Környezettudományi Tanszék. 2004.

Marcuş, Andrei–Szántó Csaba–Tóth László
Logika és halmazelmélet. Marosvásárhely, Műszaki és Humán Tudományok Kar, Matematika–Informatika Tanszék. 2004.

Kakucs András
Műszaki hőtan. Marosvásárhely, Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2004.

Biró Béla
Drámaelmélet. Csíkszereda, Gazdasági és Humántudományi Kar, Humántudományi Tanszék. 2004.

Biró Béla
Narratológia. Csíkszereda, Gazdasági és Humántudományi Kar, Humántudományi Tanszék. 2004.

Márkos Zoltán
Anyagtechnológia. Marosvásárhely. Műszaki és Humán Tudományok Kar, Gépészmérnöki Tanszék. 2004.

Grecu, Victor
Istoria limbii române. Csíkszereda, Gazdasági és Humántudományi Kar, Humántudományi Tanszék. 2004.

Varga Ibolya
Adatbázis-kezelő rendszerek elméleti alapjai. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2004.

Csapó János
Biokémia. Csíkszereda, Műszaki és Társadalomtudományi Kar, Műszaki és Természettudományi Tanszék. 2004.

Csapó János–Csapóné Kiss Zsuzsanna
Élelmiszer-kémia. Csíkszereda, Műszaki és Társadalomtudományi Kar, Műszaki és Természettudományi Tanszék. 2004.

Kátai Zoltán
Programozás C nyelven. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2004.

Weszely Tibor
Analitikus geometria és differenciálgeometria. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2005.

Györfi Jenő
A matematikai analízis elemei. Csíkszereda, Gazdaság- és Humántudományok Kar, Matematika–Informatika Tanszék. 2005.

Finta Béla–Kiss Elemér–Bartha Zsolt
Algebrai struktúrák – feladatgyűjtemény. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2006.

Antal Margit
Fejlett programozási technikák. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2006.

Csapó János–Salamon Rozália
Tejipari technológia és minőségellenőrzés. Csíkszereda, Műszaki és Társadalomtudományok Kar, Élelmiszertudományi Tanszék. 2006.

Oláh-Gál Róbert
Az informatika alapjai közgazdász- és mérnökhallgatóknak. Csíkszereda, Gazdaság- és Humántudományok Kar, Matematika–Informatika Tanszék. 2006.

Józon Mónika
: Általános jogelméleti és polgári jogi ismeretek. Csíkszereda, Gazdaság- és Humántudományok Kar, Üzleti Tudományok Tanszék. 2007.

Kátai Zoltán
: Algoritmusok felülnézetből. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2007.

Csapó János–Csapóné Kiss Zsuzsanna–Albert Csilla
: Élelmiszer-fehérjék minősítése. Csíkszereda, Műszaki és Társadalomtudományi Kar, Élelmiszertudományi Tanszék. 2007.

Ágoston Katalin–Domokos József–Márton Lőrinc
: Érzékelők és jelátalakítók. Laboratóriumi útmutató. Marosvásárhely, Műszaki és Humántudományok Kar, Villamosmérnöki Tanszék. 2007.

Szász Róbert
: Komplex függvénytan. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2007.

Kakucs András
: A végeselem-módszer alapjai. Marosvásárhely, Műszaki és Humántudományok Kar, Gépészmérnöki Tanszék. 2007.

Antal Margit
: Objektumorientált programozás. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2007.

Majdik Kornélia–Tonk Szende-Ágnes
: Biokémiai alkalmazások. Kémiai laboratóriumi jegyzet. Kolozsvár, Természettudományi és Művészeti Kar, Környezettudományi Tanszék. 2007.

Györfi Jenő–András Szilárd
: Valószínűségszámítás és lineáris programozás. A játékelmélet alapjai. Csíkszereda, Gazdaság- és Humántudományok Kar, Matematika és Informatika Tanszék. 2007.

Kátai Zoltán

Gráfelméleti algoritmusok. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–Informatika Tanszék. 2008.

Dimény Gábor

Minőségirányítási rendszerek. Marosvásárhely, Műszaki és Humántudományok Kar, Villamosmérnöki Tanszék. 2008.

Zsigmond Andrea

Minőségi és mennyiségi analitikai kémia laborkönyv. Kolozsvár, Természettudományi és Művészeti Kar, Természettudományi Tanszék. 2008.

Csapó János–Albert Csilla–Csapóné Kiss Zsuzsanna

Élelmiszer-analitika. Válogatott fejezetek. Csíkszereda, Műszaki és Társadalomtudományi Kar, Élelmiszertudományi Tanszék. 2008.

Márton Gyöngyvér

Kriptográfiai alapismeretek. Marosvásárhely, Műszaki és Humántudományok Kar, Matematika–informatika Tanszék. 2008.

Nagy Imola Katalin

A guidebook to Language Exams. English for Human Sciences. Marosvásárhely, Műszaki és Humántudományok Kar, Humántudományok tanszék. 2008.

Gagyi József

Örökség és közkapcsolatok (PR). Marosvásárhely, Műszaki és Humántudományok Kar, Humántudományok tanszék. 2008.

Fodor László

Szociálpedagógia. Marosvásárhely, Műszaki és Humántudományok Kar, Humántudományok tanszék. 2008.

Fodorpataki László–Szigyártó Lídia–Bartha Csaba

Növénytani ismeretek, Kolozsvár, Természettudományi és Művészeti Kar, Környezettudományi Tanszék. 2009.

A PARTIUMI KERESZTÉNY EGYETEM JEGYZETEI

Kovács Adalbert
: Alkalmazott matematika a közgazdaságtanban. Lineáris algebra. Nagyvárad, Alkalmazott Tudományok Kar, Közgazdaságtan Tanszék. 2002.

Horváth Gizella
: A vitatechnika alapjai. Nagyvárad, Bölcsészettudományi Kar, Filozófia Tanszék. 2002.

Angi István
: Zeneesztétikai előadások. I. Nagyvárad, Alkalmazott Tudományok Kar, Zenepedagógiai Tanszék. 2003.

Péter György–Kinter Tünde–Pajzos Csaba
: Makroökonómia. Feladatok. Nagyvárad, Alkalmazott Tudományok és Művészetek Kar, Közgazdaságtan Tanszék. 2003.

Angi István
: Zeneesztétikai előadások. II. Nagyvárad, Alkalmazott Tudományok Kar, Zenepedagógiai Tanszék. 2005.

Tonk Márton
: Bevezetés a középkori filozófia történetébe. Nagyvárad, Bölcsészettudományi Kar, Filozófia Tanszék. 2005.

Scientia Kiadó
400112 Kolozsvár (Cluj-Napoca)
Mátyás király (Matei Corvin) u. 4. sz.
Tel./fax: +40-364-401454
E-mail: scientia@kpi.sapientia.ro
www.scientiakiado.ro

Korrektúra:
Szenkovics Enikő

Műszaki szerkesztés:
Lineart Kft.

Tipográfia:
Könczey Elemér

ISBN 978-606-975-054-4