JUHÁSZ IMRE

# OpenGL

mobiDIÁK könyvtár

Juhász Imre

# OpenGL

mobiDIÁK könyvtár

SOROZATSZERKESZTŐ

Fazekas István

**JUHÁSZ IMRE**
egyetemi docens
Miskolci Egyetem

# OpenGL

Egyetemi jegyzet
Első kiadás

**mobiDIÁK könyvtár**
Debreceni Egyetem

Lektor

Bancsik Zsolt, Lajos Sándor
Miskolci Egyetem



mobiDIÁK könyvtár Debreceni Egyetem Informatikai Intézet 4010 Debrecen, Pf. 12 Hungary http://mobidiak.inf.unideb.hu/

 A mű ,,A mobiDIÁK önszervező mobil portál" (IKTA, OMFB-00373/2003)) és a ,,GNU Iterátor, a legújabb generációs portál szoftver" (ITEM, 50/2003) projektek keretében készült.

# Tartalomjegyzék

# Előszó

Ez a leírás a [3] és [6] könyvek, valamint a rendszer használata során szerzett tapasztalatok alapján készült. Az utóbbi tevékenység során született az OpenGL görbe- és felületrajzoló szolgáltatásainak kiegészítésére a GLUJ függvénykönyvtár. Mivel ez saját fejlesztésű, a függvénykönyvtárt és a kapcsolódó header fájlt mellékeltük.

Tapasztalataink szerint egy grafikus rendszer megismeréséhez először a következő kérdésekre célszerű a válaszokat megkeresni:

1. Milyen alapelemekből építhetjük fel a képeket?
2. Hogyan lehet megadni ezeket a képelemeket?
3. Milyen transzformációkon mennek át a képelemek a megadásuktól a megjelenésükig?
4. Az alapelemek megjelenésének milyen tulajdonságai adhatók meg?
5. Milyen további szolgáltatásai vannak a rendszernek?

Ezeket a szempontokat és sorrendet ajánljuk az olvasónak is annak érdekében, hogy a rendszerről átfogó képet kapjanak. Az OpenGL nagyon sok "további szolgáltatást" nyújt, de ezek megértéséhez és használatához az 1.-4. kérdésekre adott válaszok elkerülhetetlenek.

Vannak leírások, amelyek arra vállalkoznak, hogy a minden előismeret nélküli reménybeli felhasználókat is bevezetik az OpenGL rejtelmeibe. Mi erre a feladatra nem vállalkozunk. Ennek a leírásnak a megértéséhez - egyébként az OpenGL használatához is - alapvető számítástechnikai és matematikai (főleg geometriai) ismeretekre van szükség. Ezt a leírást azok forgathatják legnagyobb haszonnal, akik az OpenGL-el kapcsolatos előadásokat hallgatnak, illetve akik már ismernek más grafikus rendszert (rendszereket).

A megismerést és megértést hivatottak szolgálni a mellékelt demonstrációs programok. Ezek a programok a GLUT ([11]), GLUI ([7]) és a GLUJ (15. fejezet) függvénykönyvtárakat használják. A futtatásukhoz szükséges `glut32.dll` fájlt is mellékeltük.

# 1. fejezet

# Bevezetés

A Silicon Graphics (SG) kifejlesztette a Graphics Library (GL) rendszert, ami az általuk gyártott munkaállomások grafikus lehetőségeinek minél hatékonyabb kihasználására kialakított koncepció. A GL több programozási nyelvből elérhető függvénykönyvtárakon keresztül. Az SG munkaállomások közismert előnye a gyors és igényes grafika, amit hardver oldalról a grafikus kártyába épített egy vagy több geometriai társprocesszor (SG terminológiával geometry engine) támogat. Ez a koncepció olyan sikeresnek bizonyult, hogy vezető hardver- és szoftvergyártó cégek – többek között a DEC, IBM, Intel, Microsoft és Silicon Graphics – összefogásával létrehoztak egy munkacsoportot, amely ez alapján specifikálta az OpenGL-t.

Az OpenGL teljesen hardverfüggetlen, fő célja az igényes, térbeli objektumok ábrázolására is alkalmas (un. 3D-s), interaktív, színes, raszteres grafikát igénylő alkalmazások létrehozásának támogatása. Az OpenGL-t úgy tervezték, hogy hálózati környezetben is hatékonyan működjön, még akkor is, ha az alkalmazást futtató (kliens) számítógép típusa eltér az eredményt megjelenítő (szerver) számítógép típusától. A legtöbb munkaállomás kategóriájú számítógép támogatja az OpenGL-t, de a PC-s világban használt WIN'95, WIN'98, Windows NT, Windows 2000 és Windows XP operációs rendszerek alatt az IBM PC (és vele kompatibilis) gépeken is futtathatunk ilyen alkalmazásokat. Természetesen a UNIX operációs rendszernek a PC-ken használatos különböző változatai (pl. Linux) is támogatják az OpenGL alkalmazások futtatását. Az egyes grafikus kártyák eltérő képességűek a beépített geometriai társprocesszorok (grafikai gyorsítók) típusától és számától függően. Az OpenGL alkalmazások portabilitása érdekében ezzel nem kell törődnie a programozónak, ugyanis az alkalmazás eredményét megjelenítő szerver gép megfelelő programja vizsgálja meg, hogy az adott funkciót támogatja-e a grafikus kártya hardver szinten, és ennek megfelelően hajtja végre azt. Ezt a felhasználó csak a sebesség alapján érzékeli, ugyanis a hardver által nem támogatott funkciókat szoftverrel kell megvalósítani, ami természetesen időigényesebb.

Az OpenGL platformtól és alkalmazástól független rendszer, a felhasználás széles skáláját támogatja, a pixelműveleteket igénylő képfeldolgozástól a bonyolultabb felületek igényes megjelenítését (láthatóság, megvilágítás, textúra) is megkövetelő CAD alkalmazásokon át a gyors képváltásokra épülő valós idejű animációig. A rendszer alkotói a tökéletes hardverfüggetlenségre törekedtek, ezért a több mint 100 parancs között nincs olyan, amely a felhasználói grafikus adatbevitelt, vagy az ablakkezelést támogatná. (Ez az anyag nem tartalmazza az összes OpenGL parancs leírását.) Ezen funkciók meg-

valósítására az OpenGL implementációkhoz kapcsolódó kiegészítő függvénykönyvtárak biztosítanak lehetőséget. Számos ilyen könyvtár van, köztük olyan is, amely lehetővé teszi az alkalmazások forrásnyelvi szintű portabilitását munkaállomások és PC-k között. Ez egy rendkívül fontos eredménynek tekinthető a sokszínű, hihetetlenül gyorsan változó hardver-szoftver környezetet figyelembe véve. Az OpenGL is folyamatos fejlesztés alatt áll, jelen leírás az 1.2 verziót ismerteti.

Nagyon fontos azt is tudnunk, hogy mit ne várjunk az OpenGL-től. Mindenekelőtt: az OpenGL nem geometriai modellezőrendszer, tehát nincsenek benne összetett geometriai objektumok megadására, manipulálására alkalmas parancsok. Az OpenGL nem interaktív rajzoló- vagy grafikus rendszer. Az OpenGL azonban rendkívül alkalmas a geometriai modellezőrendszerek, CAD rendszerek, grafikus rendszerek, de még az interaktív játékprogramok grafikai igényeinek kielégítésére is. Ennek köszönhetően széles körben elterjedt. Az OpenGL-t használó alkalmazások több géptípuson, így minden munkaállomás kategóriájú gépen és a nálunk legnépszerűbb PC-ken is futtathatók, még a programok forrásnyelvi szintű portabilitása is megoldható. Valószínűleg az óriási játékprogram-piacnak köszönhetően egyre több grafikus kártya hardverből támogatja az OpenGL-t, ami jelentősen megnöveli a sebességet.

**Az OpenGL szintaxisa**

Leírásunkban az OpenGL C nyelvi felhasználói felületét használjuk. Az egyes parancsokat a megfelelő függvény hívásával aktivizálhatjuk. A függvények neve mindig **gl**-el kezdődik, amit a végrehajtandó funkció angol megnevezése követ. A megnevezésben az egyes szavak nagybetűvel kezdődnek, pl. **glPolylineMode**().

**A leírás során alkalmazott szintaxis**

1.1. táblázat. A paraméterek lehetséges típusai

| rövidítés | adattípus | a megfelelő C nyelvi típus | az OpenGL-ben definiált típus |
|---|---|---|---|
| b | 8 bites egész | signed char | GLbyte |
| s | 16 bites egész | short | GLshort |
| i | 32 bites egész | long | GLint, GLsizei |
| f | 32 bites lebegőpontos | float | GLfloat, GLcampf |
| d | 64 bites lebegőpontos | double | GLdouble, GLclampd |
| ub | 8 bites egész | unsigned char | GLubyte, GLboolean |
| us | 16 bites egész | unsigned short | GLushort |
| ui | 32 bites egész | unsigned long | GLuint, GLenum,GLbitfield |

A kapcsos zárójelek {} között felsorolt lehetőségek közül pontosan egyet kötelező választani, a szögletes zárójelek [ ] között felsoroltakból pedig legfeljebb egyet lehet. Ilyen helyzettel a függvények nevének végén találkozunk, ahol is a funkciót leíró szavak után szám és betű szerepelhet, melyek arra utalnak, hogy hány darab, milyen típusú vagy milyen módon adható meg a függvény paramétere, ugyanis egy-egy adategyüttest többféleképpen is megadhatunk. Például

```
glVertex {234}{sifd}[v]();
```

ami azt jelenti, hogy a csúcspontot megadhatjuk 2,3 vagy 4 koordinátájával, ezek

típusa lehet short, integer, float vagy double; továbbá felsorolhatjuk a koordinátákat egyenként, vagy átadhatjuk a koordinátákat tartalmazó vektor címét is. Tehát a

```
glVertex3f(1.,2.,3.);
```

és a

```
float p[] = {1.,2.,3.};
glVertex3fv(p);
```

programrészletek ugyanazt eredményezik.

Ha a leírás során pusztán a függvény nevére akarunk hivatkozni, és a paraméterek megadásának módja a tárgyalt probléma megoldása szempontjából lényegtelen, akkor a * karakterrel helyettesítjük a megadási módot leíró karaktereket. A **glVertex***(); például az összes lehetséges megadási módot jelöli. A paraméterek lehetséges típusainak listáját az 1.1. táblázat tartalmazza.

Az OpenGL-ben definiált konstansok neve GL-el kezdődik és a névben szereplő szavak nagybetűvel írandók. Ha a megnevezés több szóból áll, akkor a szavakat az aláhúzás _ karakterrel választjuk el, pl. GL_TRIANGLE_STRIP.

**Állapotváltozók**

A rendszerben számos globális paraméter, úgynevezett állapot- vagy globális változó van, melyek kurrens értéke szerint hajtja végre a rendszer a parancsokat. Ilyen pl. az objektumok színe, ami azt eredményezi, hogy nem kell minden egyes rajzolási parancs előtt a rajzolás színét is beállítanunk, hanem a rendszer az állapotváltozóban tárolt értéket használja. Ez csökkenti a felhasználói program méretét, és növeli a program végrehajtási sebességét.

Több állapotváltozó valamilyen módra, lehetőségre utal. Ezeket a **glEnable**() paranccsal lehet engedélyezni és a **glDisable**() paranccsal le lehet tiltani. Minden állapotváltozónak van alapértelmezése, továbbá kurrens értékük mindig lekérdezhető a lekérdezendő adat típusától függően a **glGet***() függvények valamelyikével. A **glGet***() függvény részletes leírása és az állapotváltozók teljes listája a 17. fejezetben található.

A kurrens állapotváltozók a **glPushAttrib**() paranccsal tárolhatók az attribútumveremben (stackben), majd a **glPopAttrib**() paranccsal újra betölthetők onnan. Az attribútumverem használata sokkal hatékonyabb, mint egy saját tárolási rendszer kialakítása.

Az OpenGL nagyon hatékony, de csak egyszerű rajzolási és azt segítő parancsokat tartalmaz, ezért összetett alakzatok ábrázolására, speciális grafikai problémák megoldására saját függvényeket kell írnunk. Mindenképpen szükségünk van olyan függvénykönyvtárra, amely az ablakkezelést, és a grafikus adatbevitelt lehetővé teszi. Ilyen célú parancsok ugyanis nincsenek az OpenGL-ben, mivel ezek megvalósítása nagymértékben hardverfüggő.

Minden OpenGL implementáció részét képezi a GLU (OpenGL Utility) könyvtár, mely a térnek síkra való leképezésében, valamint görbék és felületek ábrázolásában nyújt segítséget. A másik, mindenütt megtalálható könyvtár a GLX (az OpenGL kiterjesztése az X Window rendszerre), mely az ablakkezelést és a grafikus adatbevitelt teszi lehetővé. Annak ellenére, hogy a GLX minden OpenGL implementációnak része, a GLX helyett mi a hasonló célú GLUT (OpenGL Utility Toolkit) függvénykönyvtárt preferáljuk, mivel ez egy szabadon használható szoftver ([11]), melynek még a forráskódja is rendelkezésre áll. A GLUT használata esetén programunk forrásnyelvi szinten portábilis lesz a különböző platformok, pl. a UNIX és WINDOWS operációs rendszerek között.

Ugyanez érhető el a Qt rendszerrel is ([10]), ennek egyetlen hátránya, hogy nem szabadon használható.

# 2. fejezet

# Rajzolás az OpenGL-ben

Az OpenGL-ben három rajzolási alapművelet van:

- ablak törlése;
- geometriai objektum rajzolása;
- raszteres objektum rajzolása.

Ebben a fejezetben csak az ablak törlésével foglalkozunk, a geometriai alapelemek rajzolásával a 3. fejezetben, a raszteres objektumok rajzolásával pedig a 9. fejezetben.

Kétféle rajzolási mód van: a közvetlen (azonnali) rajzolási mód (immediate mode), ez az alapértelmezés; és lehetőség van a rajzolási parancsoknak display-listán való tárolására a későbbi, hatékony felhasználás érdekében.

## 2.1. Ablak törlése

Az ablak törlése az ablakot képviselő téglalapnak a háttérszínnel való kitöltését jelenti, szövegszerkesztő esetén pl. a papír színét jelentő fehér, vagy a szemet kevésbé irritáló halványszürke a szokásos háttérszín.

Az ablak törléséhez a **glClearColor**() paranccsal állíthatjuk be a háttér színét, magát a törlést pedig a **glClear**(GL_COLOR_BUFFER_BIT) paranccsal hajthatjuk végre. A háttérszínt egy állapotváltozóban tárolja a rendszer, vagyis nem kell minden törlés előtt megadnunk.

void **glClearColor** (GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*);

A törlési szín kurrens értékét állítja be RGBA színmegadási mód esetén. A színmegadási mód kiválasztása hardverfüggő, ezért erre nincs OpenGL parancs. Erre a GLX vagy GLUT könyvtárban találunk függvényt. A paramétereket a rendszer a $[0., 1.]$ intervallumra levágja. A törlési szín alapértelmezése $(0., 0., 0., 0.)$, ami teljesen átlátszó, fekete háttérszínt eredményez.

void **glClearIndex** (GLfloat *c*);

A törlési színindex kurrens értékét állítja be színindex-mód esetén.

Az OpenGL-ben a **glClear**() paranccsal lehet a puffereket törölni.

void **glClear** (GLbitfield *mask*);

Törli a megadott puffert (puffereket) a megfelelő állapotváltozó kurrens értéke szerint. Több puffer egyidejű törlése esetén a pufferek azonosítóját a bitenkénti VAGY művelettel (C nyelvben a | operátorral) kell összekapcsolni. A törölhető pufferek teljes listája a 2.1. táblázatban található.

2.1. táblázat. Az OpenGL pufferei

| puffer | azonosítója az OpenGL-ben |
|---|---|
| színpuffer | GL_COLOR_BUFFER_BIT |
| mélységpuffer | GL_DEPTH_BUFFER_BIT |
| gyűjtőpuffer | GL_ACCUM_BUFFER_BIT |
| stencilpuffer | GL_STENCIL_BUFFER_BIT |

A törlés időigényes feladat, mivel az ablak minden pixelére (ezek száma a milliót is meghaladhatja) el kell végezni a megfelelő műveletet. Számos grafikus hardver esetén hatékonyabb a pufferek egyidejű törlése, mint az egyenkénti törlés, és biztos, hogy az egyidejű törlés egyetlen hardver esetén sem lassítja a folyamatot.

## 2.2. A rajzolás befejezésének kikényszerítése

A rajzolási parancs kiadásától az objektum megjelenéséig sok minden történik (transzformációk, vágás, színezés, árnyalás, textúraleképezés) ezt megjelenítési láncnak (graphics pipeline) nevezzük. Ezeket a műveleteket általában más-más, az adott célra kifejlesztett speciális hardver hajtja végre, és ilyenkor a CPU csak elküldi a parancsot és nem vár a következő parancs kiadásáig arra, hogy az előző végigmenjen a műveleti soron. Hálózati környezetben, ha a kliens és a szerver különböző gépeken van, akkor az adatátvitel csomagokban történik. Az adatátvitel hatékonysága érdekében a rendszer csak akkor küldi el a csomagot, ha az – a számára fenntartott puffer – már megtelt. Adott esetben hosszú ideig várhatunk arra, hogy megteljen a csomag. A csomagok elküldését a **glFlush**() paranccsal kényszeríthetjük ki.

void **glFlush** (void);

Kikényszeríti a korábban kiadott OpenGL parancsok végrehajtásának megkezdését, és garantálja, hogy a végrehajtás véges időn belül befejeződik.

Az animációkor használt puffer-cserélő parancs (swap buffer) automatikusan kiadja a **glFlush**() parancsot.

Van olyan eset, amikor nem elég pusztán kikényszeríteni a grafikus parancsok végrehajtásának megkezdését, hanem arról is meg kell győződni, hogy a grafikus hardver ezeket az utasításokat végre is hajtotta, vagyis a végrehajtás nyugtázását várjuk (pl.

biztosak akarunk lenni abban, hogy a kép megjelent a képernyőn, mielőtt felhasználói interakciót engedélyeznénk). Ennek megoldására szolgál a **glFinish**() parancs.

void **glFinish** (void);

Kikényszeríti a korábban kiadott OpenGL parancsok végrehajtását, és nem adja vissza a vezérlést a korábban kiadott parancsok teljes végrehajtásának befejezéséig.

Ezen parancs túl gyakori használata a grafikus teljesítmény (rajzolási sebesség) jelentős csökkenését eredményezheti.

# 3. fejezet

# Geometriai alapelemek rajzolása

Ebben a fejezetben azt írjuk le, hogy az OpenGL-ben egy-egy összetett geometriai alakzatot milyen egyszerű építőkövekből - geometriai alapelemekből - állíthatunk elő. Szó lesz még ezen alapelemek megjelenését meghatározó tulajdonságok megadásáról is.

## 3.1. Geometriai alapelemek megadása

A geometriai alapelemeket csúcspontok segítségével lehet leírni. A csúcspontokat $2, 3$ vagy $4$ koordinátával lehet megadni, azaz megadhatjuk síkbeli $(x, y)$, térbeli $(x, y, z)$ derékszögű Descartes–féle koordinátákkal, vagy térbeli $(x, y, z, w)$ homogén koordinátákkal. Az OpenGL a háromdimenziós projektív tér homogén koordinátáival végez minden belső számítást, ezért a megadott koordinátákat minden esetben térbeli projektív koordinátákká egészíti ki az alábbiak szerint: $(x, y) \rightarrow (x, y, 0., 1.)$ ; $(x, y, z) \rightarrow (x, y, z, 1.)$.

Ne feledjük, hogy amennyiben $(x, y, z, w)$ homogén koordinátákkal adunk meg egy pontot, $w \neq 0$ esetén $(x/w, y/w, z/w)$ alakban állíthatjuk elő a pont Descartes–féle koordinátáit! Különösen ügyeljünk erre a NURBS görbék és felületek megadásánál (lásd a 14. fejezetet)!

Csúcspontokat a **glVertex***() paranccsal adhatunk meg.

void **glVertex**{234}{sifd} (TYPE *coords*);

void **glVertex**{234}{sifd}**v** (const TYPE **coords*);

Geometriai objektumok leírásához szükséges csúcspont megadására szolgál. Választástól függően megadhatjuk $2, 3$ vagy $4$ koordinátával, melyek típusa short, integer, float vagy double lehet; továbbá felsorolhatjuk a koordinátákat egyenként, vagy átadhatjuk a koordinátákat tartalmazó vektor címét (a második forma). Ha a $z$ koordinátát nem adjuk meg, akkor annak értéke automatikusan 0. lesz, a meg nem adott $w$ értéke pedig 1. Ezt a parancsot csak akkor hajtja végre a rendszer, ha a **glBegin**() és **glEnd**() parancspár között adtuk ki. A TYPE jelölés azt jelenti, hogy a kiválasztott típusú (esetünkben s, i, f vagy d) változókat, konstansokat vagy címet kell megadni.

A vektor használata általában hatékonyabb, ugyanis gyorsítja a paraméterek átadását.

A másik fontos definiáló adat a normálvektor, vagy röviden normális. A normálvektorokra kitöltött poligonok igényes szemléltetése során az árnyalási, megvilágítási számításoknál van szükség. Egy felület valamely pontjában vett normálisán azt

a vektort értjük, mely az adott pontban merőleges a felületre, azaz a felület érintősíkjára. Az $\mathbf{s}(u, v)$ paraméteres formában adott felület $(u_0, v_0)$ pontjában vett normálisa a

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

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

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

A felületek legelterjedtebb ábrázolási módja, hogy a felületet poligonokkal, többnyire háromszögekkel közelítjük, és a közelítő poligonokat jelenítjük meg valamilyen megvilágítási rendszer figyelembe vételével. Annak érdekében, hogy a képen közelítő poliéder élei ne látszanak, a csúcsponthoz a pontban találkozó lapok normálisainak az átlagát rendeljük normálisként. A normálisok irányításának konzisztensnek kell lenni, általában a zárt felületből kifelé mutatnak. Az OpenGL-ben csak irányítható felületek kezelhetők. Ezeknek a felületeknek két oldala különböztethető meg. A normálisok irányítását ezeknek megfelelően kell érteni, azaz a normálisok a felületnek ugyanazon oldalán legyenek. Nyílt felület esetén a kifelé és befelé jelzőknek nincs sok értelme, csak az egységes tárgyalásmód érdekében használjuk.

A rendszer minden újonnan létrehozott csúcsponthoz a kurrens (az utoljára megadott) normálist rendeli hozzá. Az OpenGL-ben minden csúcsponthoz legfeljebb egy normálist rendelhetünk, és több csúcsponthoz is hozzárendelhetjük ugyanazt a normálist. A kurrens normálist a **glNormal***() paranccsal adhatjuk meg.

void **glNormal3**{bsifd} (TYPE *nx*, TYPE *ny*, TYPE *nz*);

void **glNormal3**{bsifd}**v** (const TYPE **v*);

A kurrens normálvektort állítja be a paraméterekkel megadott értékre. A b, s, i változat esetén a rendszer a paramétereket lineárisan leképezi a $[-1., 1.]$ intervallumra.

A normálisoknak csak az irányára és irányítására van szükség, a hosszuk közömbös. A megvilágítási számítások során azonban a rendszer azt feltételezi, hogy a normálvektorok hossza egységnyi (ezek segítségével számítja a szögeket), ezért célszerű egységnyi hosszúságú (ún. normalizált) normálvektorokat megadni. Mint látni fogjuk, (lásd az 5. fejezetet) a geometriai alapelemek, vagyis az őket meghatározó csúcspontok és normálisok különböző modelltranszformációkon mennek át. Amennyiben a transzformációk valamelyike nem egybevágóság (pl. skálázás), a normálvektorok hossza megváltozhat. Ezért abban az esetben, ha nem egységnyi normálvektorokat adunk meg, vagy nem egybevágósági modelltranszformációnak vetjük alá alakzatunkat, engedélyeznünk kell, hogy az OpenGL automatikusan normalizálja a normálisokat. Ezt a **glEnable**(GL_NORMALIZE) paranccsal tehetjük meg. Ha egységnyi hosszúságú normálisokat adunk meg, és a modellt csak hasonlósági transzformációnak ($x, y$ és $z$ irányban azonos mértékű skálázásnak) vetjük alá, akkor a **glEnable**(GL_RESCALE_NORMAL) paranccsal célszerű engedélyenünk a normálisok újraskálázását, vagyis azt, hogy a rendszer a nézőpont-modell transzformációból kiszámított konstanssal megszorozza a normálisok transzformáltját, hogy azok újra egységnyi hosszúak legyenek. A rendszer alapértelmezésként nem

engedélyezi sem az automatikus normalizálást sem az újraskálázást. Elképzelhető, hogy némely OpenGL implementációban az automatikus normalizálás csökkenti a hatékonyságot, az újraskálázás viszont általában hatékonyabb, mint az általános normalizálás.

## 3.2. Geometriai alapelemek megjelenítése

Az OpenGL-ben a csúcspontok segítségével pont, szakasz és poligon geometriai alapelemek rajzolhatók. A téglalap, mint gyakran előforduló alakzat, rajzolására külön függvényt hoztak létre, mellyel az $(x, y)$ koordinátasíkban átlóinak végpontjaival adott, koordinátatengelyekkel párhuzamos oldalú téglalap rajzolható. Ez az egyetlen kivétel, amikor geometriai alapelemet nem csúcspontjaival (nem a **glVertex***() paranccsal) adhatunk meg.

void **glRect**{sifd} (TYPE *x1*, TYPE *y1*, TYPE *x2*, TYPE *y2*);

void **glRect**{sifd}**v** (TYPE **v1*, TYPE **v2*);

Az $(x, y)$ koordinátasíkon az $x$ és $y$ tengelyekkel párhuzamos oldalú téglalapot rajzol, mely egyik átlójának végpontjai (*x1*,*y1*) és (*x2*,*y2*), illetve vektoros megadás esetén a *v1* és *v2* vektorok tartalmazzák a végpontok koordinátáit.

A pont és az egyenes szakasz alapelemek a geometriában megszokottakat jelentik, azzal a különbséggel, hogy a megrajzolás során természetesen mindkettőnek van kiterjedése, illetve vastagsága, amit beállíthatunk. A számítógépi grafikában a görbéket többnyire egyenes szakaszokkal közelítjük, azaz a görbébe írt töröttvonallal.

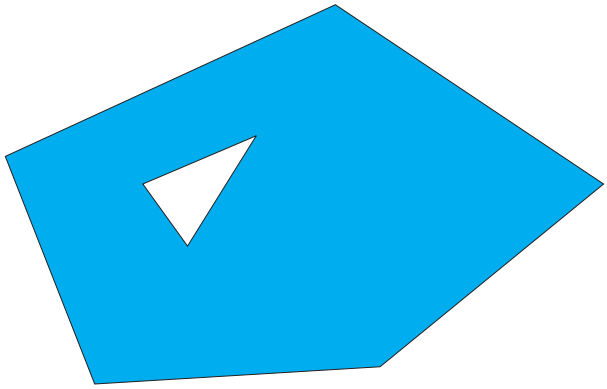
Az OpenGL poligonon zárt töröttvonal által határolt területet ért, amit többnyire a terület kifestésével jelenít meg, bár más lehetőségek is vannak. Az OpenGL csak konvex síkbeli poligonokat tud helyesen megjeleníteni, vagyis konkáv, nem egyszeresen összefüggő, vagy nem egyszerű poligonok megrajzolásával gondok lehetnek még akkor is, ha azok síkbeliek.

Egy síktartomány egyszeresen összefüggő, ha a benne fekvő bármely zárt síkgörbe egy pontra húzható össze úgy, hogy közben mindig a tartományban marad. Szemléletesen azt mondhatjuk, hogy az egyszeresen összefüggő síktartományban nem lehetnek lyukak. A 3.1. ábrán nem egyszeresen összefüggő alakzatot láthatunk.

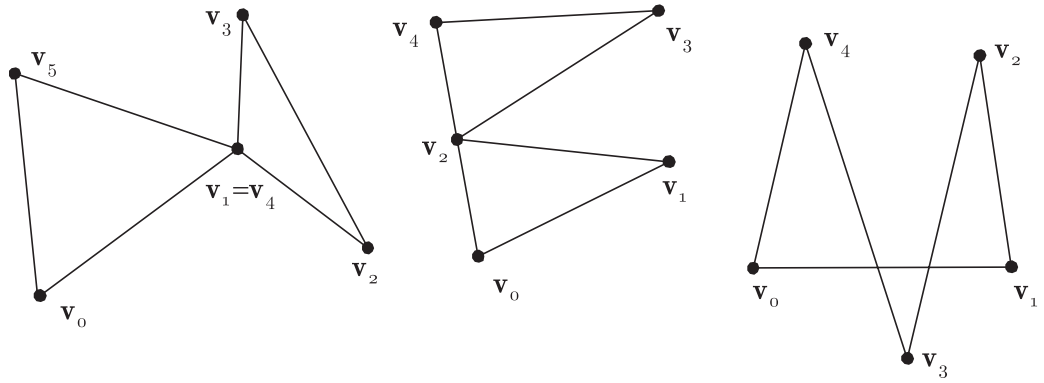
Egy sokszöget egyszerűnek nevezünk, ha valamennyi csúcsa különböző, egyik csúcs sem belső pontja a sokszög valamely oldalának, és az oldalak nem metszik egymást belső pontban. A 3.2. ábrán példákat láthatunk nem egyszerű sokszögekre.

Ezekre a megszorításokra célszerű odafigyelni, ugyanis bármilyen poligont megadhatunk csúcspontjaik felsorolásával, a rendszer nem végez érvényességi vizsgálatokat, ezért csak a nem várt eredmény figyelmeztet bennünket arra, hogy a megadott poligont nem tudja kezelni a rendszer. Ilyen nem várt eredmény lehet egy konkáv poligon esetén a poligon konvex burkának megjelenése.

Ponthalmaz konvex burkán a halmazt tartalmazó konvex ponthalmazok metszetét értjük. Síkbeli véges ponthalmaz konvex burka konvex poligon által határolt síktartomány. Ezt illusztrálja a 3.3. ábra.



3.1. ábra. Poligonok által határolt nem egyszeresen összefüggő síktartomány



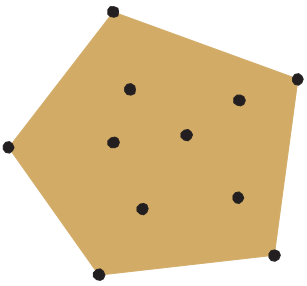

3.2. ábra. Nem egyszerű sokszögek

Nagyon fontos megszorítás, hogy csak síkbeli (nem csak koordinátasíkbeli) poligont tud a rendszer helyesen megjeleníteni. Már egy nem síkbeli egyszerű négyszög esetén is előfordulhat, hogy a különböző transzformációk következtében a négyszög képe nem egyszerű négyszög lesz, pl. a szemközti oldalak metszik egymást. A 3.4. ábrán ilyen csokornyakkendő szerű alakzatot látunk éleivel ábrázolva, illetve kitöltve. Az utóbbin jól érzékelhető a probléma. Ez síkbeli konvex, egyszeresen összefüggő poligonnal nem fordulhat elő. Ezért abban az esetben, ha görbült felületeket közelítünk poligonokkal, célszerű háromszögeket alkalmaznunk.

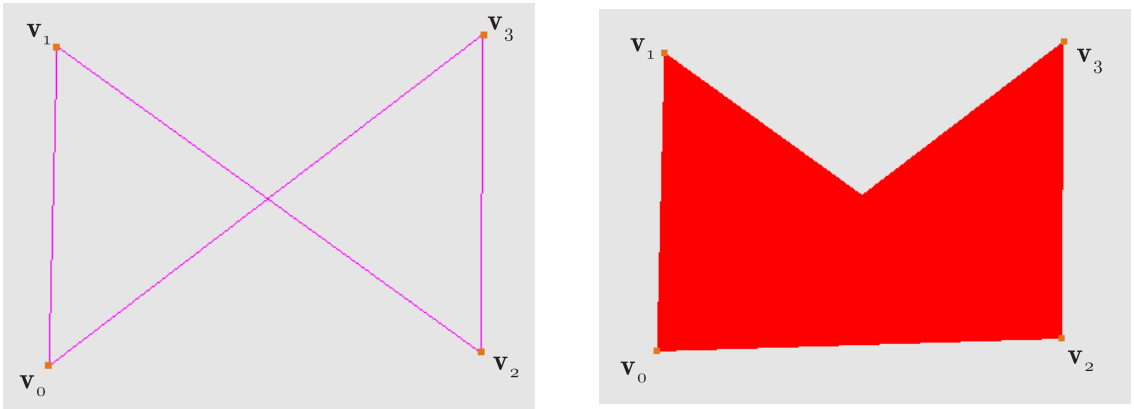
Az OpenGL-ben a geometriai alapelemeket a **glBegin**() és **glEnd**() parancsok között felsorolt csúcspontokkal hozhatunk létre. A **glBegin**() paraméterével azt adjuk meg, hogy a felsorolt csúcspontokat milyen geometriai alapelemként kell értelmezni.

void **glBegin** (GLenum *mode*);

A *mode* típusú geometriai alapelemet leíró csúcspontok listájának kezdetét jelöli. A *mode* paraméter lehetséges értékeit a 3.1. táblázat tartalmazza.



3.3. ábra. Konvex burok




3.4. ábra. Nem komplanáris csúcspontú négyszög vetületei; a bal oldali ábrán az éleivel, a jobb oldalin mindkét oldalát kifestve ábrázolva

void **glEnd** ();

Geometriai alapelemeket leíró csúcspontok listájának végét jelöli.

Az alábbiakban a **glBegin**() paraméterének részletes leírása következik. A leírás során azt feltételezzük, hogy $n$ csúcspont adott, és ezekre a $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$ jelöléssel hivatkozunk. A geometriai alapelemeket szemlélteti a 3.5. ábra.

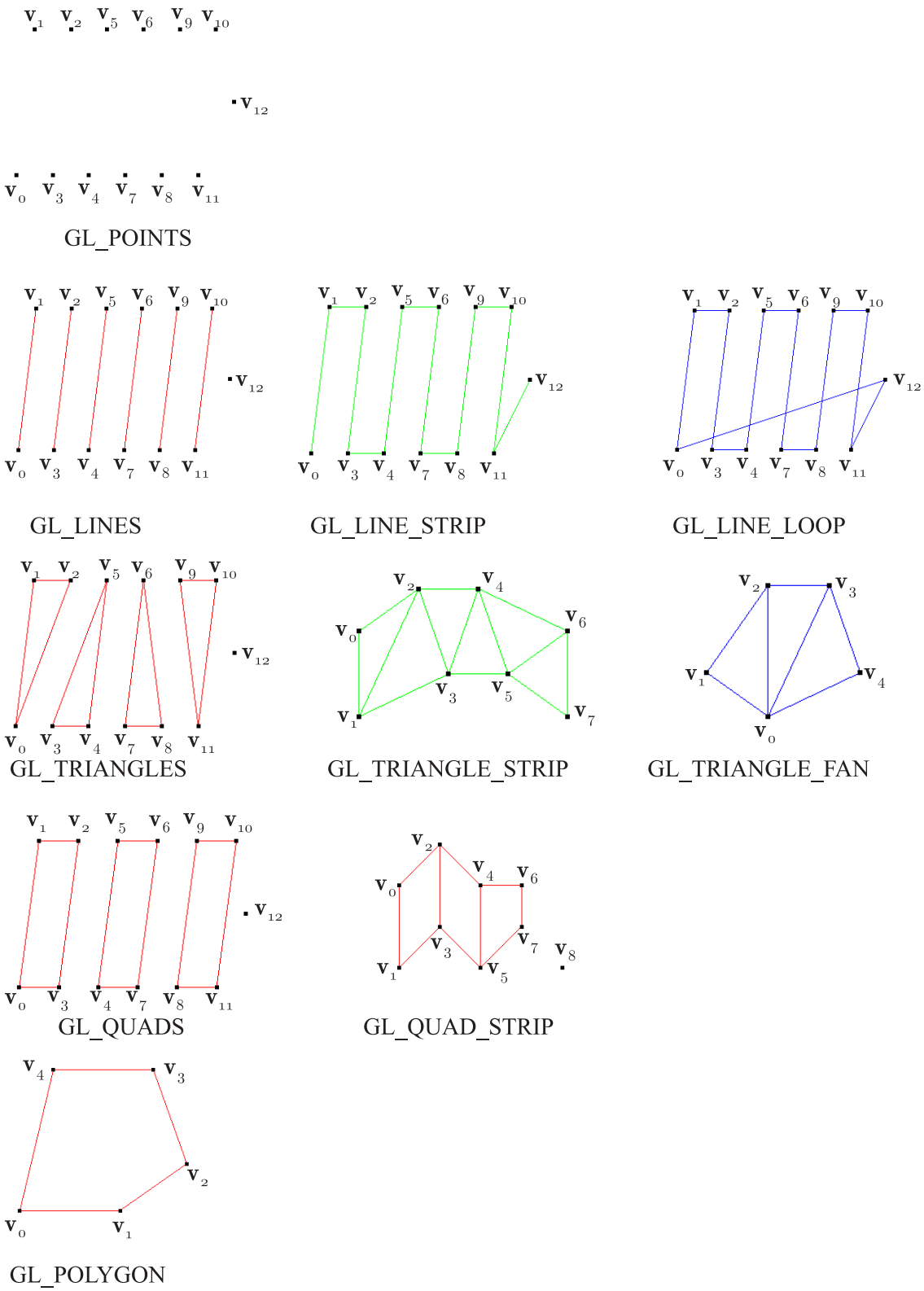
GL_POINTS A csúcspontokba pontot rajzol.

GL_LINES A $\mathbf{v}_0, \mathbf{v}_1; \mathbf{v}_2, \mathbf{v}_3; \ldots$ csúcspontokat, vagyis az egymást követő páros és páratlan indexű csúcspontokat, szakasszal köti össze. Ha $n$ páratlan, akkor az utolsó pontot figyelmen kívül hagyja.

GL_LINE_STRIP A csúcspontokat megadásuk sorrendjében egyenes szakaszokkal köti össze a rendszer, azaz egy $n-1$ oldalú töröttvonalat rajzol. $n < 2$ esetén nem történik semmi.

GL_LINE_LOOP A csúcspontokat megadásuk sorrendjében egyenes szakaszokkal köti össze a rendszer, majd a végpontot összeköti a kezdőponttal, vagyis a $\mathbf{v}_{n-1}, \mathbf{v}_0$ szakaszt is megrajzolja. Ez utóbbi az egyetlen eltérés a GL_LINE_STRIP opció hatásától. Tehát egy $n$ oldalú zárt töröttvonalat rajzol, $n < 2$ esetén nem történik semmi.

GL_POLYGON A csúcspontok által meghatározott poligont rajzolja meg. $n < 3$ esetén nem rajzol semmit. Ha a poligon nem egyszerű vagy nem konvex, az eredmény




3.5. ábra. Az OpenGL geometriai képelemei

3.1. táblázat. Az OpenGL geometriai alapelemei

| érték | jelentése |
|---|---|
| GL_POINTS | egyedi pontok |
| GL_LINES | az egymást követő csúcspontpárokat egyedi szakaszként értelmezi a rendszer |
| GL_POLYGON | egyszerű, konvex poligon határa |
| GL_TRIANGLES | az egymást követő csúcsponthármasokat háromszögként értelmezi a rendszer |
| GL_QUADS | az egymást követő csúcspontnégyeseket négyszögként értelmezi a rendszer |
| GL_LINE_STRIP | töröttvonal |
| GL_LINE_LOOP | zárt töröttvonal, azaz az utolsó csúcspontot az elsővel összeköti |
| GL_TRIANGLE_STRIP | egymáshoz kapcsolódó háromszögekből álló szalag |
| GL_TRIANGLE_FAN | legyezőszerűen egymáshoz kapcsolódó háromszögek |
| GL_QUAD_STRIP | egymáshoz kapcsolódó négyszögekből álló szalag |

meghatározatlan.

GL_TRIANGLES Egymáshoz nem kapcsolódó háromszögeket rajzol a $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2; \mathbf{v}_3, \mathbf{v}_4, \mathbf{v}_5; \ldots$ csúcspontok alapján. Ha $n$ nem három többszöröse, akkor a fennmaradó egy, vagy két csúcspontot figyelmen kívül hagyja a rendszer.

GL_TRIANGLE_STRIP Egymáshoz kapcsolódó háromszögeket (háromszögekből álló szalagot) rajzol a rendszer a $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2; \mathbf{v}_2, \mathbf{v}_1, \mathbf{v}_3; \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4; \ldots$ (lásd a 3.5. ábrát) csúcspontok felhasználásával. A csúcspontok sorrendje biztosítja, hogy a háromszögek irányítása megegyező legyen, vagyis az így létrehozott szalag egy felületdarab korrekt leírását adja. Ha $n < 3$ nem rajzol semmit.

GL_QUADS Egymáshoz nem kapcsolódó négyszögeket rajzol a rendszer a $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3; \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6, \mathbf{v}_7; \ldots$ csúcspontok alapján. Ha $n$ nem többszöröse négynek, akkor a fennmaradó csúcspontokat figyelmen kívül hagyja a rendszer.

GL_QUAD_STRIP Egymáshoz kapcsolódó négyszögeket (négyszögekből álló szalagot) rajzol a rendszer a $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_2; \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_5, \mathbf{v}_4; \ldots$ (lásd a 3.5. ábrát) csúcspontok felhasználásával. A csúcspontok sorrendje biztosítja, hogy a négyszögek irányítása megegyező legyen, vagyis az így létrehozott szalag egy felületdarab korrekt leírását adja. Ha $n < 4$ nem rajzol semmit, ha pedig $n$ páratlan, akkor az utolsó pontot figyelmen kívül hagyja.

GL_TRIANGLE_FAN A GL_TRIANGLE_STRIP opcióhoz hasonló, az egyetlen eltérés a csúcspontok figyelembe vételének sorrendjében van, most a $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2; \mathbf{v}_0, \mathbf{v}_2, \mathbf{v}_3; \ldots$ (lásd a 3.5. ábrát) a sorrend.

A **glBegin**() és **glEnd**() zárójelpár között a csúcspontok megadásán kívül más OpenGL parancsok is kiadhatók, mint pl. szín, normális vagy textúra megadása. A kiadható parancsok teljes listáját a 3.2. táblázat tartalmazza.

Ezeken kívül bármely más OpenGL parancs kiadása hibát eredményez. Ez a meg-

3.2. táblázat. A **glBegin**() és **glEnd**() között kiadható OpenGL parancsok

| parancs | hatása |
|---|---|
| **glVertex***() | csúcspont-koordináták megadása |
| **glColor***() | a kurrens szín megadása |
| **glIndex***() | a kurrens színindex megadása |
| **glNormal***() | normálvektor-koordináták megadása |
| **glEvalCoord***() | koordináták létrehozása |
| **glCallList**(), **glCallLists**() | display-lista(ák) végrehajtása |
| **glTexCoord***() | textúra-koordináták megadása |
| **glEdgeFlag***() | a határoló élek megjelölése |
| **glMaterial***() | anyagtulajdonság megadása |

szorítás csak az OpenGL parancsokra vonatkozik, nem köti meg a kezünket a programozási nyelv vezérlési szerkezeteinek, utasításainak tekintetében, vagyis tetszőleges vezérlési szerkezetek (ciklusok, feltételes kifejezések) vagy függvényhívások szerepelhetnek, feltéve, hogy a hívott függvények is csak megengedett OpenGL parancsokat tartalmaznak.

A **glVertex***() parancsnak csak akkor van hatása, ha **glBegin**(), **glEnd**() zárójelpár között jelenik meg. Ez vonatkozik a display-listán tárolt **glVertex***() parancsokra is, tehát akkor lehet bármilyen hatásuk, ha a listát egy **glBegin**(), **glEnd**() zárójelpár között hajtjuk végre.

A **glVertex***() parancs meghívásával létrehozott csúcsponthoz a rendszer hozzárendeli a csúcsponthoz kapcsolt attribútumok kurrens értékét, így a kurrens színt, normálist és textúrakoordinátát.

## 3.3. A geometriai alapelemek megjelenését befolyásoló tényezők

Ha másként nem rendelkezünk, akkor a pont képe egyetlen pixel lesz a képernyőn, a szakasz egy folytonos, egy pixel szélességű pixelsor, a poligon pedig kitöltött (kifestett) lesz. A geometriai alapelemek megjelenésének módja természetesen befolyásolható a felhasználó által.

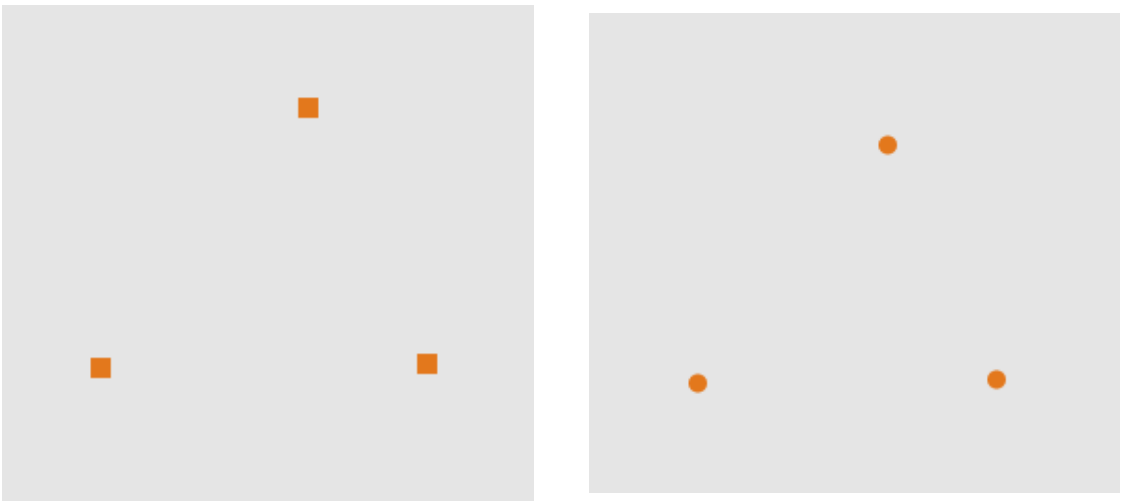
### 3.3.1. Pont

void **glPointSize** (GLfloat *size*);

Ezzel a paranccsal a pontot reprezentáló pixeltömb méretét lehet megadni a *size* paraméterrel. A megadott méretnek 0.-nál nagyobbnak kell lennie, az alapértelmezés 1.

Ha az antialiasing (kisimítás) nem engedélyezett – ami az alapértelmezés –, a rendszer a megadott float értéket egészre kerekíti és ennek megfelelő (*size* x *size*) méretű

pixeltömbbel ábrázolja a pontot. Ha az antialiasing engedélyezett, a rendszer nem kerekíti a megadott értéket, és egy *size* átmérőjű kör alakú pixelcsoporttal szemlélteti a pontot. Ezt szemlélteti a 3.6. ábra.



3.6. ábra. Pontok szemléltetése; a bal oldali képen a simítás nem engedélyezett, a jobb oldali engedélyezett

A GL_POINT_SIZE_RANGE paraméterrel kiadott **glGetFloatv**() paranccsal lekérdezhetjük, hogy az OpenGL általunk használt implementációjában mekkora lehet a pont maximális mérete. A pont színe a pont helyét meghatározó **glVertex***() parancs kiadásakor érvényben lévő szín lesz.

### 3.3.2. Szakasz

Szakasz esetén lehetőségünk van a szakasz színének, vonalvastagságának és vonaltípusának előírására. A szakasz színét két tényező befolyásolja: az érvényben lévő árnyalási modell és a szakasz végpontjainak megadásakor kurrens szín. Ezt bővebben a szín megadásával foglalkozó 4. fejezet tartalmazza.

#### A vonalvastagság megadása

void **glLineWidth** (GLfloat *width*);

A vonal vastagságának pixelekben mért értékét állítja be, $width \geq 0.$, alapértelmezés 1. A tényleges vonalvastagság függ még attól is, hogy az antialiasing engedélyezett vagy sem. Ha az antialiasing nem engedélyezett, akkor a rendszer a *width* értéket egészre kerekíti, vagyis a vastagság $1, 2, \ldots$ pixel lehet. Ilyen esetben a képernyőn a vonalvastagságot azonban nem a szakaszra merőlegesen mérjük, hanem az $x$, illetve $y$ tengelyek irányában, attól függően, hogy az egyenes iránytangense (az $x$ tengellyel bezárt szögének tangense) 1-nél nagyobb vagy kisebb. Ha az alakzatok határának kisimítása (antialiasing) engedélyezett, a rendszer nem kerekíti a vonalvastagságot, hanem kitöltött téglalapként

jeleníti meg a szakaszt. A GL_LINE_WIDTH_RANGE paraméterrel kiadott **glGetFloatv**() paranccsal lekérdezhetjük az OpenGL implementációnkban használható maximális vonalvastagságot.

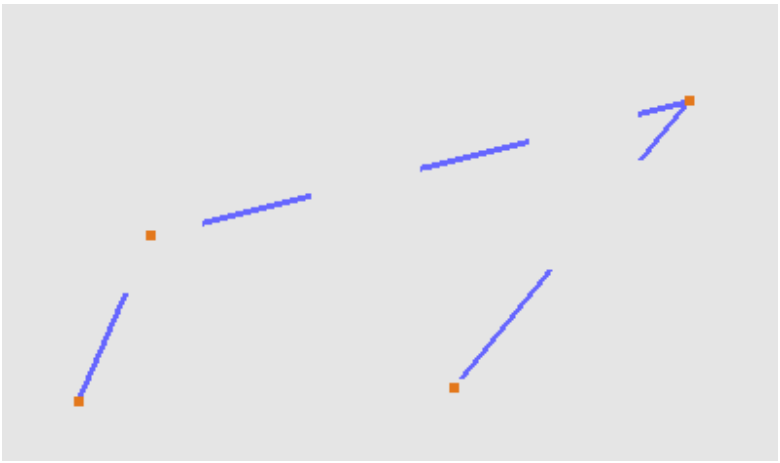
Ne feledjük, hogy az általunk látott szakasz tényleges vastagsága a képernyő pixelének fizikai méretétől is függ! Még nagyobb lehet az eltérés, ha ugyanazt a vonalvastagságot (pl. 1 pixel) használjuk grafikus display-n és nagyfelbontású tintasugaras plotteren vagy lézernyomtatón. Az utóbbiakon az 1 pixelnek megfelelő 1 dot szélességű szakasz alig lesz látható.

### A vonaltípus megadása

A vonaltípus változtatásához előbb engedélyeznünk kell a vonaltípus változtatást (a vonal mintázását) a **glEnable**(GL_LINE_STIPPLE) paranccsal, majd a **glLineStipple***() paranccsal meg kell adni a vonaltípus mintáját, amit a rendszer egy állapotváltozóban tárol. A vonaltípus változtatását a **glDisable**(GL_LINE_STIPPLE) paranccsal tilthatjuk le, ennek kiadása után a vonaltípus az alapértelmezés, azaz folytonos lesz.

void **glLineStipple** (GLint *factor*,GLushort *pattern*);

A vonaltípus mintájának megadására szolgál. A *pattern* paraméterrel azt a mintát írhatjuk le 16 biten, amit a rendszer ismétel a szakaszok rajzolásakor. A mintával pixelsort írunk le, 1 a rajzolnak, 0 a nem rajzolnak felel meg. A *factor* paraméterrel a minta nagyítását (az intervallumok nyújtását) írhatjuk elő. A factor értékét a rendszer az $[1, 255]$ intervallumra levágja.



3.7. ábra. A vonalminta ismétlése

Ha egymáshoz kapcsolódó szakaszokat, azaz töröttvonalat (GL_LINE_STRIP, GL_LINE_LOOP) rajzolunk, akkor a csúcspontoknál a már megkezdett mintaelem folytatódik, míg egyedi szakaszok rajzolásánál (GL_LINES) minden csúcspontnál az első mintaelemmel kezdődik a rajzolás (lásd a 3.7. ábrát).

### 3.3.3. Poligon

A poligon alapelem megjelenhet egy kitöltött síkidomként, a határát reprezentáló zárt töröttvonalként vagy a csúcspontjait jelölő pontsorozatként is. A rendszer minden poligonnak megkülönböztet elülső és hátsó oldalát. A különböző oldalak különböző módon jeleníthetők meg. Alaphelyzetben a rendszer mindkét oldalt azonos módon jeleníti meg. A fenti paramétereket a **glPolygonMode**() paranccsal állíthatjuk be.

void **glPolygonMode** (GLenum *face*, GLenum *mode*);

A poligonok elülső és hátsó oldalának megjelenítési módját állítja be. A *face* paraméter értéke GL_FRONT_AND_BACK, GL_FRONT vagy GL_BACK lehet. A *mode* értéke pedig GL_POINT, GL_LINE vagy GL_FILL, mely azt jelzi, hogy a poligonnak csak a csúcspontjait kell megrajzolni, a határát kell megrajzolni vagy a belsejét ki kell tölteni. Ha másként nem rendelkezünk, akkor mindkét oldalt kitöltve jeleníti meg a rendszer.

Az elülső és hátsó oldal értelmezésével részletesebben kell foglalkoznunk. Minden irányítható felület (a gyakorlatban használt felületek túlnyomó többsége – pl. gömb, kúp, henger, tórusz – ilyen, de pl. a Klein–féle palack vagy a Möbius szalag nem) közelíthető konzisztens irányítású poligonokkal (poligonhálóval). Tehát a közelítő poligonok mindegyike vagy pozitív (az óramutató járásával ellentétes) vagy negatív (az óramutató járásával egyező) körüljárási irányú a felületnek ugyanarról az oldaláról nézve. A **glFrontFace**() parancs segítségével beállíthatjuk, hogy a pozitív vagy negatív irányítást kell elülső – felénk néző – oldalnak tekinteni.

void **glFrontFace** (GLenum *mode*);

A *mode* paraméter GL_CCW értéke esetén a poligon akkor tekintendő felénk nézőnek, ha vetületének körüljárási iránya pozitív, GL_CW esetén pedig, ha negatív. Az alapértelmezés GL_CCW. Azt, hogy a képsíkon (képernyőn) egy poligonnak az elülső vagy hátsó oldala látszik a fenti beállítás és a poligon képének körüljárási iránya határozza meg. A poligon vetületének körüljárási irányát a rendszer a poligon előjeles területével határozza meg. A képsíkra illeszkedő $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ csúcspontú poligon előjeles területe

$$T = \frac{1}{2} \sum_{i=0}^{n} \mathbf{p}_i \wedge \mathbf{p}_{i+1},$$

ahol $\mathbf{p}_i = (x_i, y_i)$, $\mathbf{p}_i \wedge \mathbf{p}_{i+1} = x_i y_{i+1} - x_{i+1} y_i$ (a $\mathbf{p}_i$ és $\mathbf{p}_{i+1}$vektorok vektoriális szorzatának harmadik komponense) és $\mathbf{p}_{n+1} = \mathbf{p}_0$, vagyis az összeg tagjai az origó és a poligon oldalai által meghatározott háromszögek előjeles területei. Ezek alapján a 3.3. táblázat szerint dönthetjük el, hogy melyik oldalát látjuk a poligonnak.

Ennek a problémának van sokkal szemléletesebb, térbeli megközelítése is. Mint a normálisok megadásánál már szóltunk róla, a felületeket leíró poligonok csúcspontjaiban a normálisokat konzisztens módon kell megadni: a normálisok a felület ugyanazon oldalán legyenek. A poligonok és normálisok irányítását össze kell hangolni. Pozitív irányítású poligonok normálisa kifelé mutat, a negatívé befelé. Ezeket a poligonokat a tér egy pontjából nézzük, és vizsgáljuk, hogy a poligonok mely oldalát látjuk (feltételezzük, hogy az egyes poligonok síkbeliek). Egy poligon tetszőleges pontjából a nézőpontba mutató vektor és

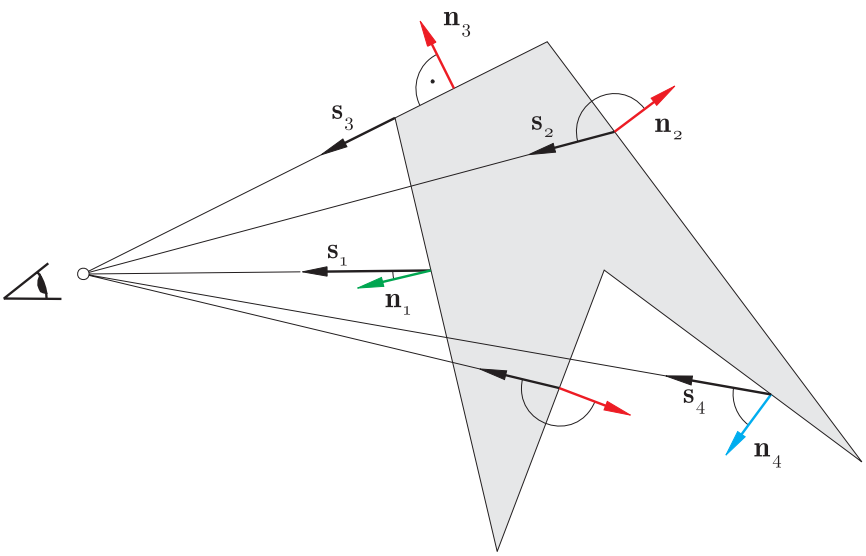
3.3. táblázat. A poligonok látható oldalának meghatározása

| | $T > 0$ | $T < 0$ |
|---|---|---|
| GL_CCW | elülső | hátsó |
| GL_CW | hátsó | elülső |

a poligon normálisa által bezárt $\alpha$ szög alapján eldönthető, hogy a poligon mely oldalát látjuk. Ha $\alpha < \pi/2$, akkor azt az oldalát amerre a normális mutat – ekkor azt mondjuk, hogy a poligon felénk néz –, $\alpha = \pi/2$ esetén a poligon élben látszik (a képe szakasz lesz), ha pedig $\alpha > \pi/2$, akkor a normális által megjelölttel ellentétes oldalát látjuk, és azt mondjuk, hogy a poligon hátsó oldala látszik. Nem kell természetesen magát a szöget meghatározni, ugyanis **n**-el jelölve a normálvektort, **s**-el a nézőpontba mutató vektort

$$\cos\alpha = \frac{\mathbf{n}\cdot\mathbf{s}}{|\mathbf{n}|\,|\mathbf{s}|},$$

tehát csak ennek az előjelét kell vizsgálni. A 3.8. ábrán egy hasábon szemléltetjük ezt a vizsgálatot. Az ábra szerinti $\mathbf{n}_1 \cdot \mathbf{s}_1 > 0$, vagyis a lap felénk néző; $\mathbf{n}_2 \cdot \mathbf{s}_2 < 0$, vagyis a lap hátsó oldalát látjuk; $\mathbf{n}_3 \cdot \mathbf{s}_3 = 0$, vagyis a lap élben látszik.




3.8. ábra. Felénk néző és hátsó poligonok értelmezése

### 3.3.4. Poligonok elhagyása

Konzisztens irányítású poligonokkal leírt zárt felületek (pl. egy gömböt közelítő poliéder) esetén a nem felénk néző poligonokat nem láthatjuk. Ezért a feldolgozás gyorsítása érdekében az ilyen poligonokat célszerű elhagyni még a képsíkra vetítés előtt, mivel ezek képe nem lesz része a végső képnek, hiszen más poligonok eltakarják. Ennek érdekében

engedélyezni kell a poligonok elhagyását a **glEnable**(GL_CULL_FACE) paranccsal, és meg kell mondanunk, hogy milyen helyzetű poligonokat akarunk elhagyni. Ez utóbbit a **glCullFace**() paranccsal tehetjük meg.

void **glCullFace** (GLenum *mode*);

Segítségével beállíthatjuk, hogy milyen helyzetű poligonok hagyhatók el még a képernyő-koordinátákra való transzformálás előtt, feltételezve, hogy a poligonok elhagyása engedélyezett. A *mode* paraméter GL_BACK értéke esetén azokat, amelyeknek a hátsó oldalát látjuk; GL_FRONT esetén a felénk néző oldalúakat; GL_FRONT_AND_BACK esetén pedig minden poligont.

Nyílt felület esetén a poligonok elhagyása helytelen eredményre vezethet, miután ilyenkor a felület mindkét oldalát láthatjuk – némely poligonnak az elülső oldalát, másoknak a hátsó oldalát –, ezért a hátsó oldalukat mutató poligonok elhagyása esetén a felület képén lyukak keletkezhetnek. Tehát nyílt felületek esetén ne használjuk ezt a funkciót!

Zárt felület esetén is csak akkor eredményez korrekt, láthatóság szerinti képet, ha egyetlen konvex poliédert ábrázolunk. Több, egymást legalább részben takaró poliéder, vagy akár csak egyetlen konkáv poliéder esetén az algoritmus alkalmazása után is maradhatnak nem látható, felénk néző lapok. Ilyen a 3.8. ábrán az $\mathbf{n}_4$ normálisú lap. Zárt felület esetén azonban célszerű ezt a funkciót aktivizálni, ugyanis ez a vizsgálat gyors, ezért jó előszűrője a lapoknak láthatósági vizsgálatokhoz, ami az OpenGL esetén a z-puffer algoritmus.

### 3.3.5. Kitöltés mintával

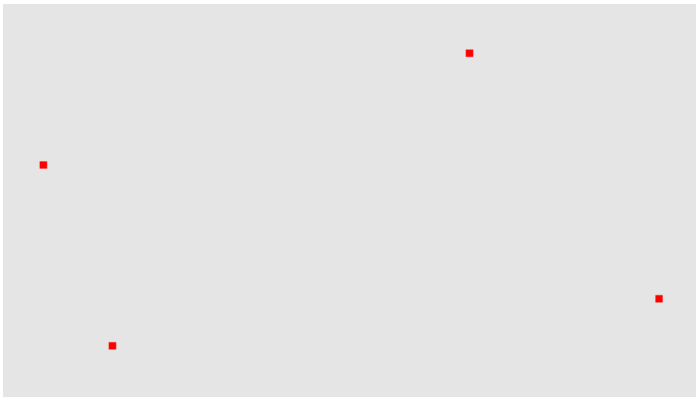
Ha másképp nem rendelkezünk, a rendszer a poligonok belsejét az érvényben lévő árnyalási modellnek (lásd a 4.2. szakaszt) megfelelően kiszínezi. Lehetőség van azonban arra is, hogy a poligonokat az ablak koordináta-rendszeréhez kapcsolt $32 \times 32$-es bittömbbel megadott mintával töltsük ki. (Ez nem tévesztendő össze a textúrával, ezt a mintát ugyanis a vetületre teszi a rendszer, nem pedig a térbeli modellre!) Ehhez engedélyezni kell a poligon mintával való kitöltését a **glEnable**(GL_POLIGON_STIPPLE) paranccsal és meg kell adni a mintát.

void **glPolygonStipple** (const GLubyte **mask*);

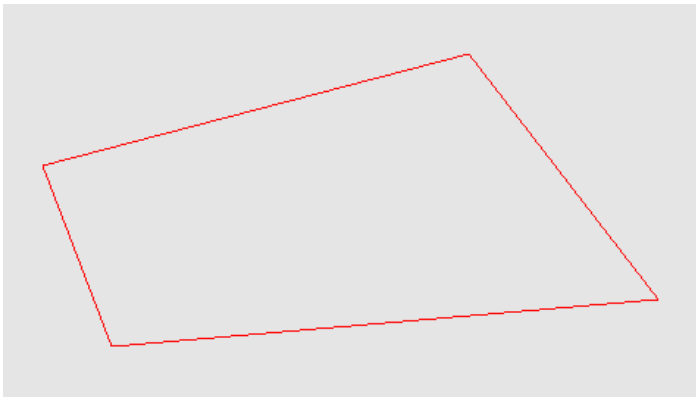
Poligonkitöltési mintát lehet vele megadni. A *mask* paraméter a kitöltési mintát pixelenként leíró $32 \times 32$-es bittömb címe. A tömbben az 1 a rajzolást, a 0 a nem rajzolást jelenti. Ezen parancs végrehajtására a **glPixelStore***(GL_UNPACK*) paranccsal beállított mód is hatással van. A poligonok különböző megjelenési módjait szemlélteti a 3.9. ábra.

### 3.3.6. Poligon határoló éleinek megjelölése

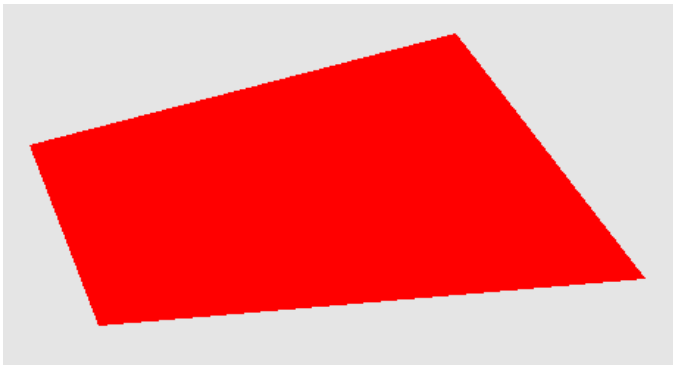
Az OpenGL csak konvex poligonokat tud helyesen megrajzolni, a gyakorlatban viszont nem számít kuriózumnak a konkáv poligon. Ezeket úgy tudjuk a siker reményében ábrázolni, hogy új oldalak hozzáadásával konvex poligonokra, többnyire háromszögekre,



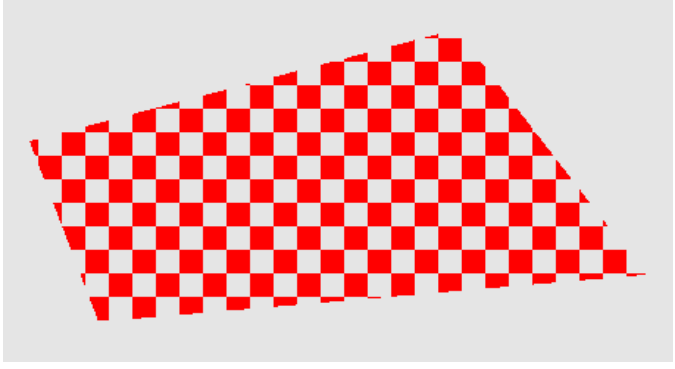
GL_POINT



GL_LINE



GL_FILL



GL_FILL és a kitöltés engedélyezett

3.9. ábra. Poligon különböző megjelenítése

bontjuk. Ha az így felbontott poligont GL_LINE módban ábrázoljuk, a felbontáshoz bevezetett új élek is megjelennek, holott mi csak a poligon határát szeretnénk látni. Ezen probléma kiküszöbölése érdekében az OpenGL a poligonok csúcspontjaihoz egy bitben tárolja, hogy a csúcspontot követő él határoló él vagy sem. Ezt a jelzőt a **glBegin**() és **glEnd**() parancsok között, a csúcspontokat leíró **glVertex***() parancs kiadása előtt lehet beállítani a **glEdgeFlag**() paranccsal.

void **glEdgeFlag** (GLboolean *flag*);

void **glEdgeFlagv** (const GLboolean **flag*);

Segítségével beállíthatjuk, hogy a következő csúcspont egy határoló él kezdőpontja, vagy sem. A *flag* paraméter GL_TRUE értékével azt jelezzük, hogy határoló él lesz, a GL_FALSE értékkel pedig azt, hogy nem. Az alapértelmezés GL_TRUE. Ezt a beállítást nem kell minden csúcspont létrehozása előtt elvégeznünk, mert a rendszer mindig a jelző kurrens értéket rendeli az újonnan létrehozott csúcsponthoz.

Ennek a parancsnak csak egyedi poligon, háromszög és négyszög (GL_POLYGON, GL_TRIANGLES, GL_QUADS) rajzolása esetén van hatása, háromszög- és négyszögsor (GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_TRIANGLE_FAN) esetén nincs.

# 4. fejezet

# Szín, árnyalás

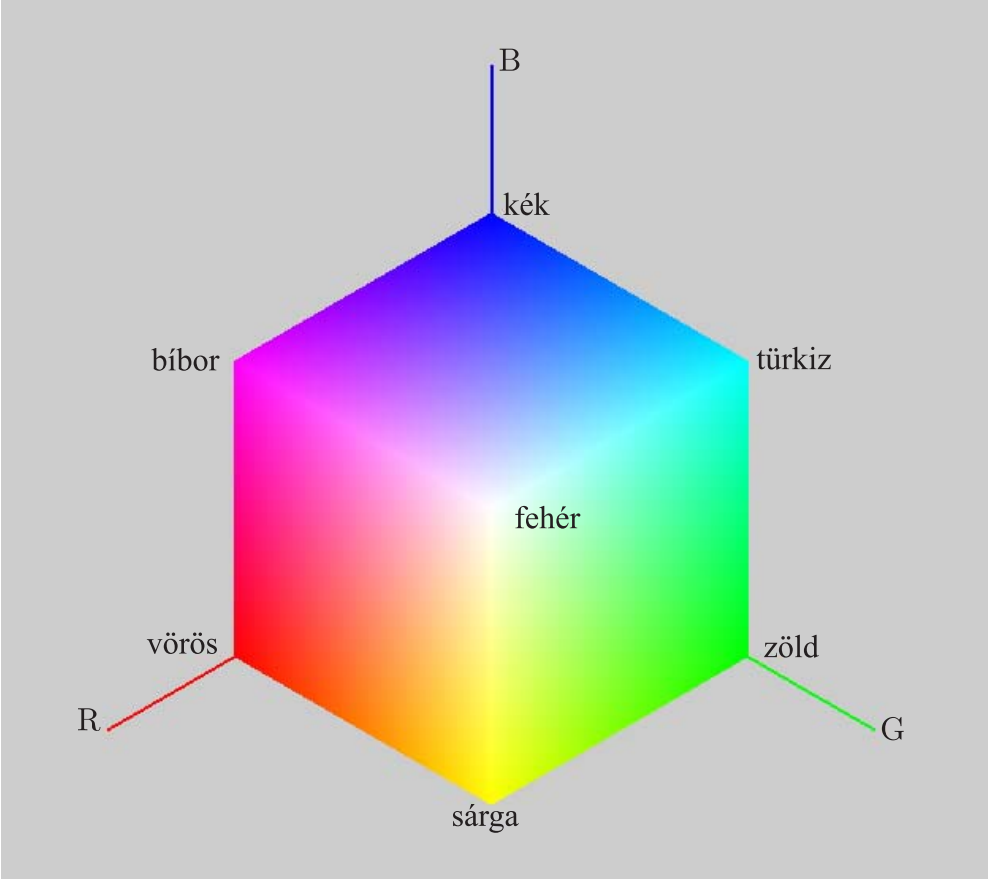
A színeket legcélszerűbb egy háromdimenziós tér pontjaiként felfogni. A tér egy bázisának (koordináta-rendszerének) rögzítése után minden színt egy számhármas segítségével lehet azonosítani. A számítógépi grafikában két színmegadási (színkeverési) mód, kétféle bázis terjedt el. Az egyik a színes grafikus display-nél használt additív (hozzáadó) színkeverés, a másik a színes nyomtatóknál hasznát szubtraktív (kivonó).

Additív színkeverés esetén azt írjuk elő, hogy mennyi vörös (Red), zöld (Green) és kék (Blue) színkomponenst adjunk a fekete színhez, a fekete képernyőhöz. A színteret az egységkockával szemléltetjük, mivel a színkomponensek értéke általában a $[0., 1.]$ intervallumon változtatható a különböző grafikus rendszerekben, beleértve az OpenGL-t is. Ezt a színelőállítási módot RGB színrendszernek szokás nevezni. A $(0., 0., 0.)$ számhármas a fekete, az $(1., 1., 1.)$ a fehér színnek, a legösszetettebb színnek, felel meg (lásd a 4.1. ábrát). A színes raszteres grafikus display-k működése ezzel összhangban van, hiszen a képernyő minden egyes pixele különböző intenzitású vörös, zöld és kék színt tud kibocsátani. Az RGB színmegadás az emberi szem működésével is összhangban van. Az emberi szem retinájában a világosságlátást lehetővé tevő érzékelő sejtek, az un. pálcikák, továbbá a színlátást lehetővé tevő érzékelő sejtek, az un. csapok találhatók. A csapok három fajtája különböztethető meg aszerint, hogy milyen hullámhosszú fény ingerli őket leginkább. Ez a három hullámhossz éppen a vörös, zöld és kék színé, vagyis leegyszerűsítve azt mondhatjuk, hogy az emberi szemben RGB érzékelők vannak, és a különböző arányban érzékelt színkomponensek összegzése útján áll elő az általunk látott szín.

Szubtraktív színkeverés esetén azt írjuk elő, hogy mennyi türkiz (Cyan), bíbor (Magenta) és sárga (Yellow) színt kell kivonnunk a legösszetettebb színből, a fehérből (a papír színe) a kívánt szín előállítása érdekében (lásd a 4.2. ábrát). Ezt röviden CMY színrendszernek nevezzük.

A grafikai hardver által használt színkeverésen kívül más módok is lehetségesek, más bázis is felvehető a színtérben. Ilyen pl. a képzőművészek által használt színárnyalat, fényesség és telítettség (HLS) színmeghatározás. Erről, és a színnel kapcsolatos egyéb problémákról a [4], [1], [2] könyvekben részletesebben olvashatunk.

A számítógépek grafikus alaprendszereiben a színek megadása általában kétféleképpen történhet. Az egyik színmegadási mód esetén közvetlenül megadjuk a kívánt szín RGB komponenseit, a másik esetén csupán egy indexszel hivatkozunk egy színtáblázat (színpaletta) valamely elemére. Természetesen a paletta színei a felhasználó által megváltoztathatók, vagyis ugyanahhoz a színindexhez más színt rendelhetünk hozzá.

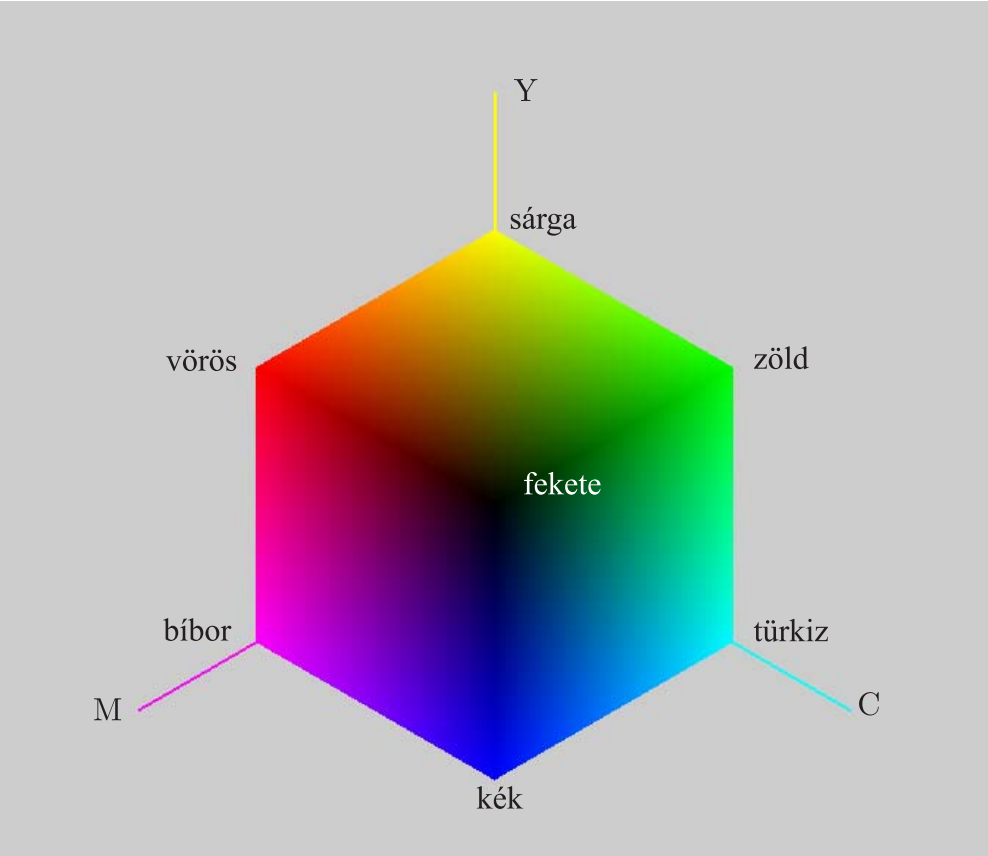



4.1. ábra. A színkocka az RGB koordináta-rendszerben

Az OpenGL-ben az RGB komponensek mellett egy negyedik, un. alfa (A) komponenst is megadhatunk, ezért itt RGBA módról beszélünk. Az alfa komponens természetesen nem a szín definiálásához kell, ahhoz pontosan három komponens szükséges, vele az átlátszóságot adhatjuk meg, vagyis azt kontrollálhatjuk, hogy az adott színnel kifestendő pixelen csak az új szín látsszon, vagy a pixel régi és új színéből együttesen álljon elő a pixel végső színe. Alfa 1. értéke esetén a szín nem átlátszó, 0. esetén teljesen átlátszó, a közbülső értékek esetén pedig a pixel régi és új színét keveri a rendszer az előírt arányban.

A színes grafikus hardver egyrészt felbontásban – a képernyő pixeleinek számában –, másrészt a pixeleken megjeleníthető színek számában különbözik. E kettőt a monitor és a grafikus kártya minősége, és azok összhangja határozza meg. Jó minőségű, valószerű képek előállításához nem elegendő a felbontást növelni, ugyanis viszonylag alacsony felbontás mellett nagyszámú szín esetén szinte tökéletes képet hozhatunk létre. Gondoljunk pl. a TV készülékekre, melyek felbontása elég kicsi (a PAL rendszerű készülékek 625 soron jelenítik meg a képet) a mai monitorok felbontásához képest, de a megjeleníthető színek száma – ezt szokás színmélységnek is nevezni – gyakorlatilag korlátlan.

A számítógépekben, pontosabban azok grafikus kártyáján, a képernyő minden pixeléhez ugyanannyi memória tartozik, melyben a pixelek színét tárolja a gép. A pixelek színének tárolására használt memóriát színpuffernek nevezzük. A puffer méretét azzal jellemezzük, hogy pixelenként hány bit áll rendelkezésre a színek tárolására. 8 bit/pixel




4.2. ábra. A színkocka a CMY koordináta-rendszerben

esetén a megjeleníthető színek száma $2^8 = 256$. A manapság használatos grafikus kártyák sokkal több színt tudnak megjeleníteni, a 24 bit/pixel kezd standarddá válni, már a 16 bit/pixel is kihalóban van. A legjobbnak mondható kártyák esetén 32 bit/pixel (az RGBA összetevők mindegyikére 8, vagyis 16.77 millió szín plusz az átlátszóság, az ilyen grafikus kártyákat szokás true colornak is nevezni) és akkora a memória, hogy ilyen színpufferből két darabot tud használni (erre pl. animáció esetén van szükségünk), tehát ezt felfoghatjuk 64 bit/pixelként is.

A pixelenkénti biteket a rendszerek általában egyenletesen osztják el a színkomponensek között, bár vannak kivételek. Azt, hogy az általunk használt rendszerben hány bit áll rendelkezésünkre az egyes színkomponensek, illetve az alfa érték tárolására, továbbá, hogy az egyes színindexekhez milyen RGB értékek tartoznak a **glGetIntegerv**() parancs GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS, illetve GL_INDEX_BITS paraméterrel való meghívásával kapjuk meg.

Az RGBA és színindex módok közötti választásra nincs OpenGL parancs, mivel ez hardverfüggő paraméter. Erre szolgáló függvényt az OpenGL-hez kapcsolt, ablakozást és felhasználói interakciókat lekezelő függvénykönyvtárakban találhatunk, pl. a minden OpenGL implementáció részét képező GLX-ben, vagy az általunk preferált GLUT-ben.

Színindex mód esetén előfordulhat, hogy saját színpalettát akarunk megadni. Természetesen erre sincs OpenGL parancs, erre is a fent említett kiegészítő könyvtárakban találhatunk függvényeket. Színpaletta definiálása esetén ügyelnünk kell arra, hogy az em-

ber által egyenletesen változónak érzékelt intenzitásváltozás nem úgy érhető el, hogy az RGB értékeket lineárisan változtatjuk. Például a csak kék komponensű színt a $[0., 1.]$ intervallumon tekintve, a 0.1-ről a 0.11-re váltást sokkal jelentősebbnek érzékeljük, mint a 0.5-ről a 0.51-re váltást. A 0.55 értékkel sokkal jobb hatást érünk el. Hasonlók a tapasztalatok a fényerő érzékelése kapcsán is, 50 W és 100 W között – mármint ilyen teljesítményű villanyégők között – nagyobb ugrást észlelünk, mint 100 W és 150 W között. A lineáris skála helyett sokkal jobb a logaritmikus skála, bár ez a probléma erősen hardverfüggő. Monitorok esetén ennek a problémának a megoldását gamma-korrekciónak nevezik (további információk [4]-ban találhatók).

A felhasználói programban döntenünk kell a színmegadás módjáról, és ez a továbbiakban nem változtatható meg, tehát egy alkalmazásban vagy végig RGBA módban, vagy színindex módban kell dolgoznunk. A rajzolás színének kiválasztása nem jelenti feltétlenül azt, hogy az adott objektum a kiválasztott színnel fog megjelenni a képernyőn, ugyanis sok más hatást is figyelembe vesz a rendszer, pl. megvilágítást, árnyalást vagy textúrát. A pixelek végső színe egy hosszú műveletsor eredményeként áll elő.

A rendszer a színt a csúcspontokban számolja ki. A csúcsponthoz rendeli a kurrens rajzolási színt. Ha a megvilágítás engedélyezett, akkor a fényforrás(ok) színe, és az árnyalási modelltől függően (lásd a 4.2. szakaszt) a transzformált csúcspontbeli normális, továbbá az anyagtulajdonságok (lásd a 6.3. szakaszt) is befolyásolják a végső színt. Ilyen esetben a rajzolási szín hatástalan. A rendszer a megjelenítő transzformációs lánc végén az objektumot a képmezőkoordináta-rendszerbe transzformálja, majd raszterekké konvertálja, vagyis meghatározza, hogy mely pixelek esnek az alakzat képére. Ezekhez a raszterekhez hozzárendeli az alakzat megfelelő részének színét, mélységét ( $z$ koordináta) és textúra koordinátáit. A raszter és a hozzárendelt értékek együttesét fragmentumnak nevezzük az OpenGL-ben. Miután a fragmentumokat létrehozta a rendszer, ezekre alkalmazza a köd, textúra, simítási effektusokat, ha azok használata engedélyezett. Végül az így módosított fragmentumra és a képernyő pixeleire végrehajtja az előírt alfa színkeverést, ditheringet és a bitenkénti logikai műveleteket – ha ezek engedélyezettek, illetve előírtak –, és az így kapott színnel felülírja a megfelelő pixel színét.

Ha a bit/pixel szám alacsony, pl. csak 4 vagy 8, akkor több színt úgy érzékeltethetünk, hogy négyzetes pixeltömbök pixeleit különböző elrendezésben festjük ki a rendelkezésre álló színekkel. Ezáltal új színhatású pixeltömbök jönnek létre. Természetesen csak akkor jó a hatás, ha a pixelek elég kicsik, illetve elég messziről nézzük a képet. Ezt a technikát ditheringnek nevezzük. Ezzel gyakorlatilag a felbontás csökkentése révén növeljük a színek számát. Használata a nyomtatóknál széles körben elterjedt, de szükség lehet rá gyengébb grafikus kártyák (pl. 24 bit/pixel) esetén is, ha animálunk, azaz mozgó képet hozunk létre. Ilyenkor ugyanis két azonos méretű színpuferre van szükség, tehát az eredeti 24 bit/pixelből csak 12 bit/pixel lesz, ami már nem alkalmas un. true color képek előállítására.

Színindex mód esetén a rendszer csak a színpalettába mutató indexet tárolja pixelenként, nem pedig magát a színt. A paletta a színek RGB komponenseit tartalmazza. A paletta színeinek a száma 2 hatványa, általában $2^8 = 256$-tól $2^{12} = 4036$-ig szokott terjedni. RGBA módban a pixelek színe teljesen független egymástól, vagyis, ha egy pixel színét megváltoztatjuk, annak nincs hatása a többi pixel színére. Színindex módban viszont a paletta valamely színének megváltoztatása esetén a rá mutató indexű pixelek

mindegyikének megváltozik a színe. Ezt használjuk ki az un. színanimáció során.

Felmerül a kérdés, hogy mikor használjunk RGBA és mikor színindex módot. Ezt alapvetően a rendelkezésre álló hardver, és a megoldandó feladat határozza meg. Legtöbb rendszerben RGBA módban több szín jeleníthető meg a képernyőn, mint színindex módban. Számos speciális effektus, mint pl. az árnyalás, megvilágítás, textúra-leképezés, atmoszférikus hatások, antialiasing és átlátszóság RGBA módban sokkal eredményesebben valósítható meg.

A színindex módnak is vannak azonban előnyei. Célszerű színindex módot választanunk,

- ha egy olyan programot kell átírnunk OpenGL-be, amely már ezt a módot használta;
- ha kevesebb színt akarunk használni, mint a színpuffer által kínált maximum;
- ha a bit/pixel érték alacsony, azaz kevés szín jeleníthető meg, ugyanis RGBA módban az árnyalás durvább lehet;
- ha olyan speciális trükköket akarunk megvalósítani, mint a színanimáció, vagy a rétegekbe rajzolás;
- ha pixelenkénti logikai műveleteket akarunk a színeken végrehajtani.

## 4.1. Szín megadása

RGBA módban a kurrens színt a **glColor***() paranccsal adhatjuk meg.

void **glColor3**{bsifdubusui} (TYPE *r*,TYPE *g*,TYPE *b*);

void **glColor4**{bsifdubusui} (TYPE *r*,TYPE *g*,TYPE *b*,TYPE *a*);

void **glColor**{34}{bsifdubusui}**v** (const TYPE **v*);

A kurrens színt az RGB(A) komponensekkel definiálja. A **v**-vel végződő nevű parancsok esetén a paramétereket tartalmazó vektor címét adja át. Ha alfa értékét nem adjuk meg, a rendszer automatikusan 1.-nek veszi. Lebegőpontos paraméterek – f és d opciók – esetén a rendszer a megadott értéket a $[0., 1.]$ intervallumra levágja, az előjel nélküli egész paramétereket – ub, us, ui opciók– lineárisan leképezi a $[0., 1.]$ intervallumra (lásd a 4.1. táblázatot), az előjeles egészeket pedig ugyancsak lineárisan képezi le a $[-1., 1.]$ intervallumra. A rendszer csak a színinterpolálás és a színpufferbe való beírás előtt végzi el a levágást.

Színindex módban a **glIndex***() paranccsal választhatjuk ki a kurrens színindexet.

void **glIndex**{sifd} (TYPE *c*);

void **glIndex**{sifd}v (TYPE *c*);

A kurrens színindexet a választott opciónak megfelelő módon megadott értékre állítja be.

4.1. táblázat. A színkomponensek leképezése

| jel | adattípus | megadható értékek intervalluma | amire leképezi |
|---|---|---|---|
| b | 8 bites egész | $[-128, 127]$ | $[-1., 1.]$ |
| s | 16 bites egész | $[-32768, 32767]$ | $[-1., 1.]$ |
| i | 32 bites egész | $[-2147483648, 2147483647]$ | $[-1., 1.]$ |
| ub | előjel nélküli 8 bites egész | $[0, 255]$ | $[0., 1.]$ |
| us | előjel nélküli 16 bites egész | $[0, 65535]$ | $[0., 1.]$ |
| ui | előjel nélküli 32 bites egész | $[0, 4294967295]$ | $[0., 1.]$ |

## 4.2. Árnyalási modell megadása

A szakasz és poligon alapelemeket megrajzolhatjuk egyetlen színnel, vagy sok különbözővel. Az első esetet konstans árnyalásnak (flat shading), a másodikat pedig folytonos vagy sima árnyalásnak (smooth shading) nevezzük. A **glShadeModel**() paranccsal választhatunk a két árnyalási technika között.

void **glShadeModel** (GLenum *mode*);

Az árnyalási modell kiválasztására szolgál. A *mode* paraméter értéke GL_SMOOTH vagy GL_FLAT lehet, alapértelmezés a GL_SMOOTH.

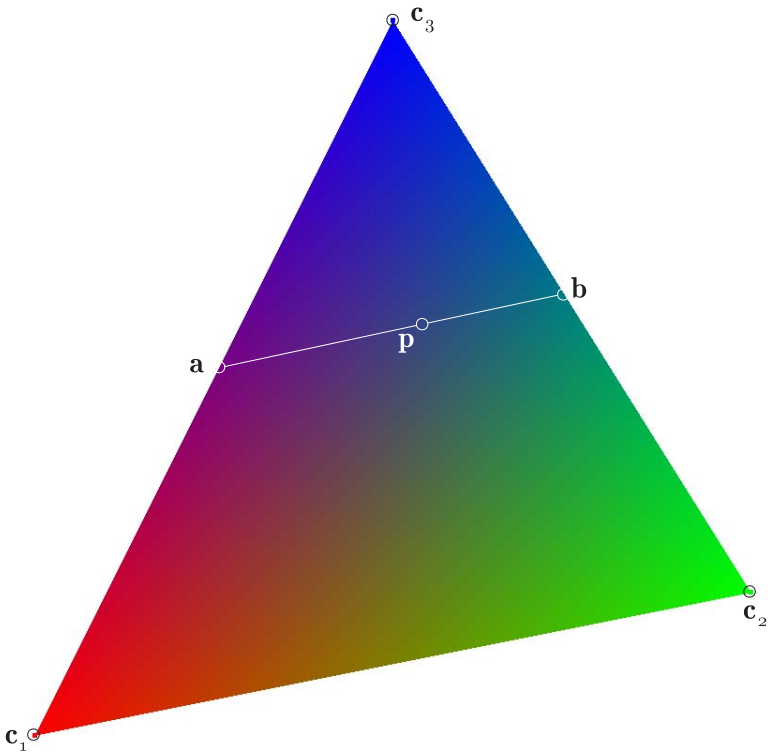
Konstans árnyalás esetén a rendszer egyetlen csúcspont színével színezi ki az egész objektumot. Töröttvonal esetén ez a végpont színe, poligon esetén pedig a 4.2. táblázat szerinti. A táblázat azt mutatja, hogy az $i$-edik poligont mely csúcspont színével rajzoljuk meg, feltételezve, hogy a csúcspontok és poligonok számozása 1-el kezdődik. Ezeket a szabályokat következetesen betartja a rendszer, de a legbiztosabb, ha konstans árnyalásnál csak egy színt adunk meg az alapelem létrehozásakor.

4.2. táblázat. A poligon színének meghatározása

| poligon fajta | az $i$-edik poligon melyik csúcspont szerint lesz kiszínezve |
|---|---|
| GL_POLYGON | 1 |
| GL_TRIANGLE_STRIP | $i + 2$ |
| GL_TRIANGLE_FAN | $i + 2$ |
| GL_TRIANGLES | $3i$ |
| GL_QUAD_STRIP | $2i + 2$ |
| GL_QUADS | $4i$ |

Folytonos árnyalás esetén a csúcspontok színét – pontosabban RGBA módban a színt komponensenként, színindex módban az indexeket – lineárisan interpolálja a rendszer a szakaszok belső pontjai színének kiszámításához. Poligon határának megrajzolásakor is ez az eljárás, a poligon belsejének kiszínezésekor pedig kettős lineáris interpolációt hajt végre.

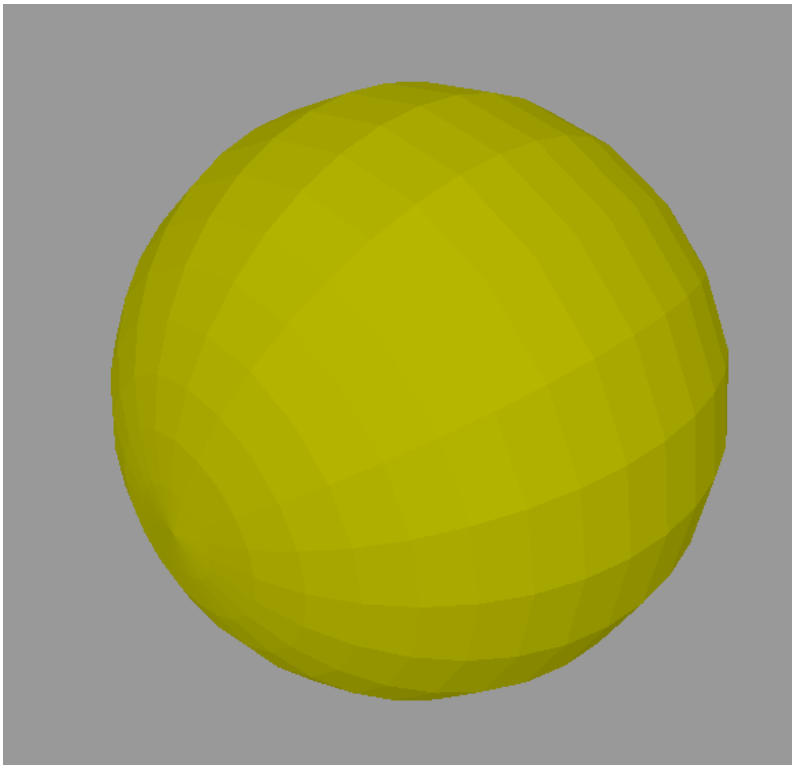
A 4.3. ábra szerinti **a** pontot a $\mathbf{c}_1$ és $\mathbf{c}_3$ csúcspontok, a **b** pontot a $\mathbf{c}_2$ és $\mathbf{c}_3$ csúcspontok alapján, majd a **p** belső pont színét az **a** és **b** pontok színének lineáris interpolációjával határozza meg. Ezt az árnyalást Gouroud–féle árnyalásnak is nevezzük.

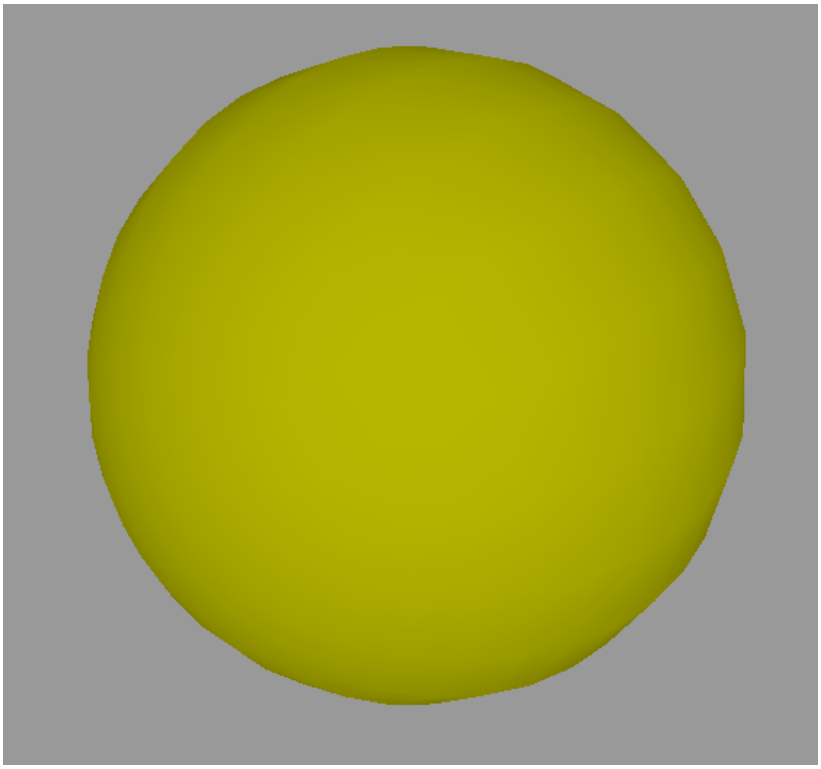



4.3. ábra. Gouroud-féle árnyalás

Folytonos árnyalás alkalmazásával el tudjuk tüntetni a képről a görbült felületeket közelítő poligonháló éleit. Ezt illusztrálja a 4.4. ábra, ahol egy gömböt láthatunk konstans (felső kép) és folytonos (alsó kép) árnyalással.

Színindex mód esetén meglepetések érhetnek bennünket, ha a szomszédos színindexekhez tartozó színek nagyon eltérőek. Ezért ha színindex módban folytonosan akarunk árnyalni, akkor előbb célszerű a megfelelő színpalettát létrehoznunk.





4.4. ábra. Gömb konstans és folytonos árnyalással

# 5. fejezet

# Koordináta-rendszerek és transzformációk

A geometriai alapelemek több transzformáción mennek keresztül mire végül megjelennek a képernyőn (ha egyáltalán). Nagymértékű analógia mutatható ki a műtermi fényképkészítés, beleértve a papírra vitelt is, és az OpenGL-el való képalkotás között. Ez az analógia a következő fő lépésekre épül (zárójelben az OpenGL megfelelő transzformációja szerepel):

- A fényképezőgép elhelyezése, ráirányítása a lefényképezendő térrészre (nézőpontba transzformálás).
- A lefényképezendő objektumok elhelyezése a kívánt pozícióba (modelltranszformáció).
- A megfelelő lencse kiválasztása, zoom beállítása (vetítési transzformáció).
- Az elkészült negatívról megfelelő méretű papírkép készítése (képmező-transzformáció).

Ahhoz, hogy az OpenGL-el képet hozzunk létre, a fenti transzformációkat kell megadnunk. Az alábbiakban ezen transzformációk megadásáról, az egyes transzformációk hatásáról, működésük leírásáról, a transzformációkat leíró mátrixokról és azok tárolásáról lesz szó.

Az OpenGL koordináta-rendszerei derékszögű Descartes–féle koordináta-rendszerek, de a belső számításokat homogén koordinátákban hajtja végre a rendszer, és a koordináta-rendszerek közti transzformációkat $4 \times 4$-es valós elemű mátrixszal írja le és tárolja. Az egyes transzformációk megadására OpenGL parancsok állnak rendelkezésünkre, de arra is lehetőség van, hogy a felhasználó magát a transzformációt leíró mátrixot adja meg közvetlenül. Az utóbbi esetben különös figyelmet kell fordítani arra, hogy a megadott mátrixok illeszkedjenek az OpenGL transzformációs láncába.

A transzformációkat $\mathbf{p}\prime = \mathbf{M}\mathbf{p}$ alakban írhatjuk fel, ahol $\mathbf{p}$ a transzformálandó pont, $\mathbf{p}\prime$ a transzformált pont homogén koordinátáit, $\mathbf{M}$ pedig a transzformációt leíró $4 \times 4$-es mátrixot jelöli. Ha a $\mathbf{p}$ pontra előbb az $\mathbf{M}_1$, majd az $\mathbf{M}_2$ mátrixú transzformációt alkalmazzuk, akkor $\mathbf{p}\prime = \mathbf{M}_1\mathbf{p}$, $\mathbf{p}\prime\prime = \mathbf{M}_2\mathbf{p} = \mathbf{M}_2\left(\mathbf{M}_1\mathbf{p}\right) = \left(\mathbf{M}_2\mathbf{M}_1\right)\mathbf{p}$ miatt az eredő

transzformáció az $\mathbf{M} = \mathbf{M}_2\mathbf{M}_1$ mátrixszal írható le. A sorrend nem közömbös, mivel a mátrixok szorzása nem kommutatív (nem felcserélhető) művelet!

Az OpenGL tárolja a kurrens transzformációs mátrixot és ezt megszorozza a megfelelő OpenGL paranccsal létrehozott új transzformáció mátrixával, de nem balról, hanem jobbról. Ezért az OpenGL számára sajnos éppen fordított sorrendben kell megadni a transzformációkat! Tehát az előző példa végeredményéhez előbb az $\mathbf{M}_2$, majd az $\mathbf{M}_1$ transzformációt kell megadni. Ez nem túl szerencsés, de nem tudunk rajta változtatni, alkalmazkodnunk kell hozzá.

## 5.1. Nézőpont, nézési irány kiválasztása és modelltranszformációk

Az ábrázolandó alakzatot elmozgathatjuk a koordináta-rendszerhez képest, vagy akár alakjukat és méretarányaikat is megváltoztathatjuk. Ezeket a transzformációkat az OpenGL terminológiája szerint modelltranszformációnak nevezzük.

A nézőpont és a nézési irány kiválasztásával azt határozzuk meg, hogy az alakzatot honnan és milyen irányból nézzük, vagyis azt, hogy az alakzat mely részét látjuk. Alapértelmezés szerint – ha nem adunk meg külön nézőpontot és nézési irányt – az origóból nézünk a negatív $z$ tengely irányába. A nézőpont, nézési irány és további feltételek segítségével egy új koordináta-rendszert, az un. nézőpontkoordináta-rendszert hozzuk létre. Ennek alapértelmezése tehát az objektumkoordináta-rendszer.

Nyilvánvaló, hogy a nézőpont és nézési irány beállítása elérhető a modell megfelelő transzformációjával is. Egy tárgyat körüljárva ugyanazt látjuk, mintha egy helyben maradva a tárgyat forgatnánk körbe. Míg a valóságban a tárgyak nem mindig forgathatók körbe – pl. méretük miatt –, számítógépi modell esetén ez nem jelent gondot. E két transzformáció szoros kapcsolata és helyettesíthetősége miatt az OpenGL egyetlen közös mátrixban tárolja a nézőpont- és modelltranszformációt. Ez a közös mátrix tehát a két komponenstranszformáció mátrixainak szorzata. Miután az OpenGL mindig jobbról szorozza meg a kurrens mátrixot az újjal, előbb a nézőpontba transzformálást kell megadni, majd a modelltraszformáció(ka)t, ha azt akarjuk, hogy előbb a modelltranszformációkat hajtsa végre a rendszer.

A nézőpont- és modelltranszformációk megadása előtt a **glMatrixMode**(GL_MODELVIEW) parancsot ki kell adni, amivel jelezzük, hogy ezt a mátrixot fogjuk módosítani (lásd az 5.4. pontot). Az alábbiakban a nézőpont- és modelltranszformációk megadását segítő OpenGL parancsokat ismertetjük.

### 5.1.1. Modelltranszformációk megadását segítő parancsok

Az OpenGL számára a felhasználó közvetlenül megadhatja a transzformáció $4 \times 4$-es mátrixát és a **glLoadMatrix**() parancsot használva ezzel felülírhatja, a **glMultMatrix**() parancs segítségével megszorozhatja a kurrens transzformációs mátrixot. Az OpenGL transzformációk megadását segítő parancsai előbb kiszámítják a transzformáció $4 \times 4$-es mátrixát, majd ezzel megszorozzák a kurrens mátrixot.

**Eltolás**

void **glTranslate**{fd} (TYPE $x$, TYPE $y$,TYPE $z$);

Előállítja az $[x, y, z, 1.]^T$ vektorral való eltolás mátrixát, és megszorozza vele a kurrens mátrixot.

**Elforgatás**

void **glRotate**{fd} (TYPE *angle*, TYPE $x$,TYPE $y$, TYPE $z$);

Előállítja a tér pontjait az origón áthaladó, $[x, y, z]^T$ irányvektorú egyenes körül *angle* szöggel elforgató mátrixot, és megszorozza vele a kurrens mátrixot. A szög előjeles, és fokban kell megadni.

**Skálázás és tükrözés**

void **glScale**{fd} (TYPE $\lambda$, TYPE $\mu$, TYPE $\nu$);

Előállítja annak a transzformációnak a mátrixát, amely az $x$ tengely mentén $\lambda$, az $y$ tengely mentén $\mu$, a $z$ tengely mentén pedig $\nu$ mértékű kicsinyítést vagy nagyítást, valamint az előjeltől függően esetleg koordinátasík(ok)ra vonatkozó tükrözést hajt végre, és megszorozza vele a kurrens mátrixot. Ez tehát egy speciális affin transzformáció, mely az alakzatok méretét, méretarányait (pl. szélesség / magasság) és irányítását is meg tudja változtatni. Az 1.-nél nagyobb abszolút értékű tényező az adott tengely menti nagyítást, az 1.-nél kisebb abszolút értékű kicsinyítést, a negatív tényező pedig az előzőek mellett az adott tengelyre merőleges koordinátasíkra vonatkozó tükrözést is eredményez.

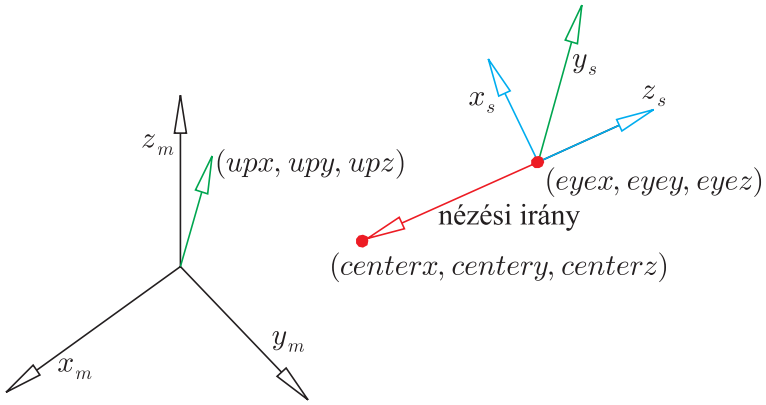
### 5.1.2. Nézőpont és nézési irány beállítása

Alapértelmezés szerint a nézőpont a modellkoordináta-rendszer origója, a nézési irány pedig a negatív $z$ tengely. Egy új nézőpont és nézési irány beállításával egyenértékű, ha az objektumot mozgatjuk el ellentétes irányban. Ezt az előzőekben ismertetett **glTranslate***() és **glRotate***() parancsokkal tehetjük meg. Ez azonban nem mindig kényelmes, a felhasználó gondolkodásához nem feltétlenül illeszkedő eljárás. Ha az új nézőpontot és nézési irányt, ezáltal új nézőpontkoordináta-rendszert explicite akarunk megadni, akkor a GLU függvénykönyvtár **gluLookAt**() függvényét használjuk!

void **gluLookAt** (GLdouble *eyex*, GLdouble *eyey*, GLdouble *eyez*,GLdouble *centerx*, GLdouble *centery*, GLdouble *centerz*,GLdouble *upx*, GLdouble *upy*, GLdouble *upz*);

Egy új nézőpontot és nézési irányt megadó mátrixot hoz létre, és ezzel megszorozza a kurrens mátrixot. Az (*eyex*, *eyey*, *eyez*) számhármas az új nézőpontot – az új origót –,

a (*centerx*, *centery*, *centerz*) számhármas az új nézési irányt – az új $z$ tengely irányát –, az (*upx*, *upy*, *upz*) számhármas pedig a felfelé irányt – az új $y$ tengely irányát – írja elő (lásd az 5.1. ábrát).




5.1. ábra. Az új nézőpont és nézési irány megadása a modellkoordináta-rendszerben a **gluLookAt**( ) függvénnyel

Ezzel a függvénnyel tehát azt adjuk meg, hogy honnan (*eyex*, *eyey*, *eyez*) hová (*centerx*, *centery*, *centerz*) nézünk, és a képen mely térbeli irány (*upx*, *upy*, *upz*) képe legyen függőleges.

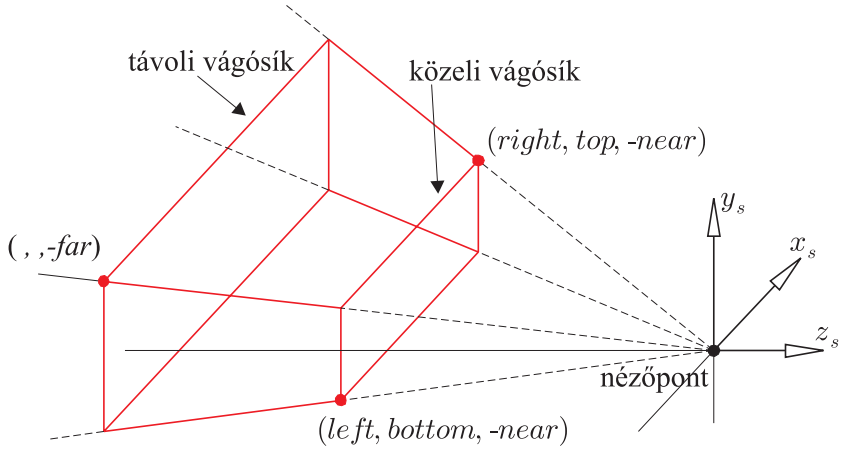
## 5.2. Vetítési transzformációk

Az OpenGL 3D-s grafikus rendszer, vagyis térbeli alakzatok ábrázolására is alkalmas. Ezért szükség van olyan transzformációkra, melyek a teret síkra – a képernyő is síkbeli tartománynak tekintendő – képezik le. A rendszer erre két lehetőséget kínál: a centrális és a merőleges vetítést. A vetítés módjának kiválasztásához a nézőpontkoordináta-rendszerben meg kell adnunk azt a térrészt, amit ábrázolni szeretnénk. Ez centrális vetítés esetén csonka gúla, merőleges vetítésnél téglatest. A képen csak azok az alakzatok, illetve az alakzatok azon részei jelennek meg, melyek az előző testeken belül vannak, ami azt jelenti, hogy az ebből kilógó részeket a rendszer eltávolítja, vagyis a vetítő poliéder lapjaival levágja. Ez a vágás a vetítési transzformáció után, az un. vágó koordinátákban történik. A vetítési transzformáció nem síkbeli képet eredményez – ezt várhatnánk el –, hanem a vetítendő térrészt (csonka gúlát, téglatestet) egy kockára képezi le.

A vetítés módjának kiválasztását és az ábrázolandó térrész kijelölését egyetlen paranccsal hajthatjuk végre. Centrális vetítésnél a **glFrustum**() vagy **gluPerspective**(), merőleges vetítésnél pedig a **glOrtho**() vagy **gluOrtho2D**() parancsokkal. Centrális (perspektív) vetítést akkor alkalmazunk, ha valószerű képet szeretnénk létrehozni (az emberi látás is így működik), merőleges vetítést pedig méret- vagy méretarány-helyes képek létrehozásakor. Ezen parancsok előtt ne felejtsük el kiadni a **glMatrixMode**(GL_PROJECTION); és **glLoadIdentity**(); parancsokat! Az elsővel megadjuk, hogy a vetítési mátrixot akarjuk módosítani (azt tesszük kurrenssé), a másodikkal betöltjük az egységmátrixot. A rendszer ugyanis az újonnan létrehozott vetítési mátrixokkal megszorozza a kurrens mátrixot.

void **glFrustum** (GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*);

Az 5.2. ábra szerinti csonka gúlával meghatározott centrális vetítés mátrixát állítja elő, és megszorozza vele a kurrens mátrixot. A *near* és *far* értékek a közeli és távoli vágósíknak a nézőponttól (a nézőpont koordináta-rendszerének origójától) mért távolságát jelentik, mindig pozitívak. A csonka gúla fedőlapján az egyik átló végpontjának koordinátái $(left, bottom, -near)$ és $(right, top, -near)$.




5.2. ábra. A **glFrustum**( ) függvény paramétereinek jelentése

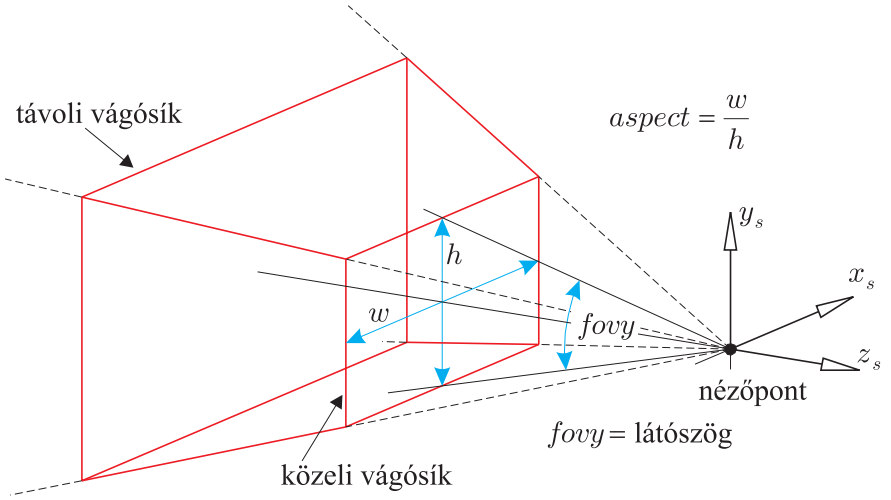
Az így megadott csonka gúlának nem kell feltétlenül egyenes gúlából származnia – a $z_s$ tengely és a fedőlap metszéspontja nem feltétlenül esik egybe a fedőlap átlóinak metszéspontjával –, vagy másként megközelítve: az $(z_s, s_s)$ és $(y_s, z_s)$ koordinátasíkok nem feltétlenül szimmetriasíkjai a csonka gúlának.

A **gluPerspective**() függvénnyel szimmetrikus csonka gúlát (ábrázolandó térrészt) adhatunk meg a látószög (a felső és alsó vágósíkok szöge), a kép szélességének és magasságának arányával, valamint a közeli és távoli vágósíkoknak a nézőponttól mért távolságával.

void **gluPerspective** (GLdouble *fovy*, GLdouble *aspect*, GLdouble *near*, GLdouble *far*);

Az 5.3. ábra szerinti szimmetrikus csonka gúlával meghatározott centrális vetítés mátrixát állítja elő, és megszorozza vele a kurrens mátrixot. Az $fovy \in [0., 180.]$ látószög az $(y, z)$ síkban mérendő. Az optimális látószöget a szemünknek a képernyőtől való távolságából és annak az ablaknak a méretéből kapjuk meg, melyen megjelenítjük a képet. Pontosabban, azt a szöget kell meghatározni, amely alatt az ablakot látjuk. A *near* és *far* értékek a közeli és távoli vágósíknak a nézőponttól (a nézőpont koordináta-rendszerének origójától) mért távolságát jelentik, mindig pozitívak.

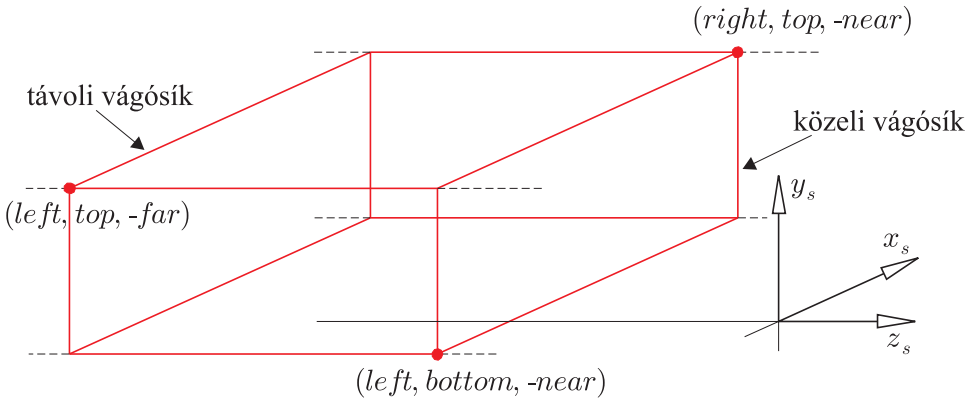
Merőleges vetítés megadására is két lehetőségünk van. Ilyen esetben természetesen a nézőpont helye gyakorlatilag közömbös, csak a nézési irány számít.




5.3. ábra. A **gluPerspective**( ) függvény paramétereinek jelentése

void **glOrtho** (GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*, GLdouble *near*, GLdouble *far*);

Az 5.4. ábra szerinti téglatesttel meghatározott merőleges vetítés mátrixát állítja elő, és megszorozza vele a kurrens mátrixot. A *near* és *far* paraméterek negatívok is lehetnek.




5.4. ábra. A **glOrtho**( ) függvény paramétereinek jelentése

Az $(x, y)$ koordinátasíkra illeszkedő alakzatok ábrázolását könnyíti meg a következő függvény.

void **gluOrtho2D** (GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*);

Ennek hatása csak abban különbözik a **glOrtho**() parancsétól, hogy nem kell a közeli és távoli vágósíkokat megadni, a rendszer automatikusan a $z \in [-1., 1.]$ koordinátájú pontokat veszi csak figyelembe, vagyis ezek lesznek a közeli és távoli vágósíkok.

A fenti parancsokkal megadott poliéderek oldallapjaival a rendszer levágja az alakzatokat, tehát csak a poliéderen (csonka gúlán, illetve téglatesten) belüli részt képezi le. Ezt a vágást a vetítési transzformáció után hajtja végre a rendszer, vagyis a csonka gúla vagy téglatest lapjai helyett a nekik megfelelő kocka lapjaival. Ezen vágósíkok mellett legalább további hat vágósík adható meg bármely OpenGL implementációban, melyek segítségével tovább szűkíthetjük az ábrázolandó térrészt. Ez a lehetőség felhasználható pl. csonkolt alakzatok ábrázolására is. A GL_MAX_CLIP_PLANES paraméterrel kiadott **glGetIntegerv**() parancs megadja, hogy az általunk használt implementációban hány vágósík adható meg.

void **glClipPlane** (GLenum *plane*, const GLdouble **equation*);

Ezzel a paranccsal egy vágósíkot adhatunk meg. A *plane* paraméter értéke GL_CLIP_PLANE$i$ lehet ($i = 0, \ldots,$ *vágósíkok száma*$-1$), ami az új vágósík azonosítója. Az **equation* paraméter a sík $Ax + By + Cz + D = 0$ implicit alakjának együtthatóit tartalmazó vektor címe. A rendszer az új vágósíkot a kurrens nézőpont-modell transzformációnak veti alá. A vágást a nézőpontkoordináta-rendszerben hajtja végre, úgy, hogy a transzformált sík pozitív oldalán lévő pontokat hagyja meg, vagyis azokat az $(x_s, y_s, z_s, w_s)$ koordinátájú pontokat, amelyekre $A_s x_s + B_s y_s + C_s z_s + D_s w_s \geq 0$, ahol $(A_s, B_s, C_s, D_s)$ a nézőpontba transzformált vágósík együtthatóit jelöli.

Az $i$-edik vágósíkra történő vágást a **glEnable**(GL_CLIP_PLANE$i$); paranccsal engedélyezhetjük, és a **glDisable**(GL_CLIP_PLANE$i$); paranccsal tilthatjuk le.

## 5.3. Képmező-transzformáció

A vetítési transzformáció után a csonka gúlából és a téglatestből egyaránt kocka lesz. Ha a homogén koordinátákról áttérünk Descartes–féle koordinátákra – $(x, y, z, w) \rightarrow (x/w, y/w, z/w)$, ha $w \neq 0$ – a kocka pontjaira $x_n \in [-1., 1.]$, $y_n \in [-1., 1.]$, $z_n \in [-1., 1.]$ teljesül. Ezeket normalizált koordinátáknak nevezzük az OpenGL-ben.

A képmező a képernyő ablakának az a téglalap alakú pixeltömbje, amire rajzolunk. A képmező oldalai az ablak oldalaival párhuzamosak. A képmező-transzformáció az előbb definiált kockát képezi le arra a téglatestre, melynek egyik lapja a képmező. A képmezőt a **glViewport**() paranccsal adhatjuk meg.

void **glViewport** (GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height)*;

A képernyő ablakában azt az ablak oldalaival párhuzamos téglalapot adja meg, melyben a végső kép megjelenik. Az $(x, y)$ koordinátapárral a képmező bal alsó sarkát, a *width* paraméterrel a szélességét, a *height* paraméterrel pedig a magasságát kell megadnunk pixelekben. A paraméterek alapértelmezése *(0, 0, winWidth, winHeight)*, ahol *winWidth* és *winHeigh* az ablak szélességét és magasságát jelöli.

A normalizált koordináta-rendszerbeli kockát egy skálázással arra a téglatestre képezi le a rendszer, melynek egy lapja a képmező, a rá merőleges él hossza pedig 1. Az így kapott koordinátákat ablakkoordinátáknak nevezzük. Az ablakkoordináták is térbeliek, a rendszer ebben végzi el a láthatósági vizsgálatokat. Az ablakkoordináták $z_a$ komponense

alapértelmezés szerint a $[0., 1.]$ intervallumon változik. Ezt az intervallumot a **glDepthRange**() paranccsal szűkíthetjük.

void **glDepthRange** (GLclamped *near*, GLclamped *far*);

Az ablakkoordináták $z_a$ komponensének minimális és maximális értékét írhatjuk elő. A megadott értékeket a rendszer a $[0., 1.]$ intervallumra levágja, ami egyben az alapértelmezés is.

A képmező-transzformáció mátrixát nem lehet explicite megadni, és nincs számára verem fenntartva.

## 5.4. A transzformációs mátrixok kezelése

A transzformációs mátrixok létrehozása, manipulálása előtt a **glMatrixMode**() paranccsal meg kell adnunk, hogy a műveletek melyik transzformációra vonatkozzanak. Az így kiválasztott mátrixot kurrensnek nevezzük.

void **glMatrixMode** (GLenum *mode*);

Segítségével beállíthatjuk, hogy melyik transzformációt akarjuk módosítani. A *mode* paraméter lehetséges értékei: GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE, alapértelmezése GL_MODELVIEW. A kurrens mátrix alapértéke mindhárom módban az egységmátrix.

void **glLoadIdentity** (void);

A kurrens mátrixot az egységmátrixszal felülírja. Ez a parancs azért fontos, mert a transzformációs mátrixokat előállító OpenGL parancsok mindig megszorozzák a kurrens mátrixot, tehát új transzformáció megadásánál célszerű előbb az egységmátrixot betölteni.

A felhasználónak lehetősége van saját mátrixok megadására a nézőpontba transzformáláshoz, a modelltranszformációhoz és a vetítési transzformációhoz, vagyis nem kell feltétlenül az ilyen célú OpenGL parancsokat használni. A felhasználó által létrehozott $4 \times 4$-es mátrixokat egy 16 elemű vektorban kell átadni, tehát a mátrixot vektorrá kell konvertálni, mégpedig a C nyelv konvenciójával ellentétben oszlopfolytonosan.

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

A konvertált mátrixokat a **glLoadMatrix**() és **glMultMatrix**() parancsokkal tudjuk beilleszteni az OpenGL transzformációs láncába. A $4 \times 4$-es valós elemű mátrixokkal tetszőleges térbeli projektív transzformációk leírhatók, nemcsak az OpenGL parancsok által felkínált legfeljebb affin modelltranszformációk.

void **glLoadMatrix**{fd} (const TYPE **m*);

Az $m$ vektorban tárolt mátrixszal felülírja a kurrens mátrixot.

void **glMultMatrix**{fd} (const TYPE *$m$);

Az $m$ vektorban tárolt mátrixszal jobbról megszorozza a kurrens mátrixot.

Az OpenGL két vermet (stack) tart karban a transzformációs mátrixokkal kapcsolatos manipulációk megkönnyítésére. Egyet az egyesített nézőpont-modell transzformációk, egy másikat a vetítési transzformációk számára. A nézőpont-modell transzformációk verme legalább 32 db $4 \times 4$-es mátrix tárolására alkalmas, az implementációbeli pontos számot a **glGetIntegerv**() parancs GL_MAX_MODELVIEW_STACK_DEPTH paraméterrel való meghívásával kapjuk meg. A vetítési mátrixok verme legalább 2 db $4 \times 4$-es mátrix tárolására alkalmas, pontos méretét a GL_MAX_PROJECTION_STACK_DEPTH paraméterrel kiadott **glGetIntegerv**() paranccsal kapjuk meg.

A **glMatrixMode**(GL_MODELVIEW) parancs a nézőpont-modell transzformációs verem legfelső elemét teszi kurrenssé, a **glMatrixMode**(GL_PROJECTION) pedig a vetítési mátrixok vermének legfelső elemét.

Minden mátrixműveletet a kurrens mátrixszal hajt végre a rendszer, tehát a **glLoadIdentity**() és **glLoadMatrix**() ezt írja felül; a **glMultMatrix**() és a transzformációkat beállító OpenGL parancsok (pl. **glRotate***(), **glFrustum**()) ezt szorozza meg (jobbról!).

A vermek használatát két parancs teszi lehetővé: a **glPushMatrix**() és a **glPopMatrix**(). Ezek mindig a **glMatrixMode**() paranccsal beállított vermet manipulálják.

void **glPushMatrix** (void);

A **glMatrixMode**() paranccsal beállított kurrens verem minden elemét egy szinttel lentebb tolja. A legfelső (a kurrens) mátrix a második mátrix másolata lesz, vagyis a legfelső két mátrix elemei megegyeznek. A megengedettnél több mátrix esetén a **glMatrixMode**() parancs kiadása hibát eredményez.

void **glPopMatrix** (void);

A **glMatrixMode**() paranccsal beállított kurrens verem legfelső elemét eldobja, és minden további elemet egy szinttel fentebb tol. A legfelső (a kurrens) mátrix a korábban a második szinten lévő mátrix lesz. Ha a verem csak egy mátrixot tartalmaz, akkor a **glPopMatrix**() parancs kiadása hibát eredményez.

# 6. fejezet

# Megvilágítás

A megvilágítás modellezésének fontos szerepe van a valószerű képek előállításában, segítségével jobban érzékeltethető a mélység és a térbeliség. A számítógépi grafikában használt megvilágítási számítások általában nem valamely fizikai folyamat leírásai, hanem tapasztalati úton előállított képletek, melyekről az idők során bebizonyosodott, hogy elég valószerű látványt biztosítanak.

Az OpenGL csak a csúcspontokban számítja ki a színt, a csúcspontok által definiált alakzatok (pl. szakasz, poligon) többi pontjában az árnyalási modelltől függően (lásd a 4.2. szakaszt) vagy a csúcspontokban kiszámított értékek lineáris interpolációjával határozza meg, vagy valamely csúcspont színét használja. Egy csúcspont színe nem más, mint a pontból a nézőpontba (a szembe) eljutó fény színe, amit a csúcspontot tartalmazó objektum anyagának tulajdonságai, az ábrázolt térrészben uralkodó fényviszonyok, továbbá az ábrázolt alakzatok optikai kölcsönhatásai határoznak meg. Az OpenGL viszonylag egyszerű modellt alkalmaz, de még így is elég sok lehetőségünk van igényes, valószerű képek létrehozására.

Az ábrázolandó térrészben uralkodó fényviszonyok leírására több fényösszetevőt vesz figyelembe a rendszer.

- Az ábrázolandó objektumok által kibocsátott fény, melyre nincsenek hatással a fényforrások.
- A környezeti fény (ambient light) az ábrázolandó térrészben mindenütt jelen lévő, állandó intenzitású fény, melynek forrása, iránya nem ismert (gondoljunk olyan nappali fényre, amikor a nap a felhők mögött van). A térrészben jelen lévő környezeti fény két részből tevődik össze. Egyrészt a térben a fényforrásoktól függetlenül (azokra vissza nem vezethető) jelen lévő környezeti fényből – ezt globális környezeti fénynek is nevezik –, másrészt a fényforrásokból származó (pl. többszörös tükröződések útján keletkező) környezeti fényből, amit a fényforrások környezeti fénykomponensének nevezünk.
- A szórt fénynek (diffuse light) van iránya, mindig valamelyik fényforrásból jön. Fő jellemzője, hogy az objektumokkal ütközve minden irányba azonos módon és mértékben verődik vissza, tehát teljesen mindegy, hogy milyen irányból nézzük az objektumokat, a hatás csak a fényforrástól, az anyagtulajdonságoktól és a csúcspontbeli normálistól függ.

- A tükrözött fénynek (specular light) is van iránya és forrása, és hatása nemcsak az anyagtulajdonságoktól és a csúcspontbeli normálistól, hanem a nézőponttól is függ. Gondoljunk egy sima felületű fémgömbre, amit erős, koncentrált fénnyel világítunk meg. Megfelelő szögből nézve egy fényes (többnyire fehér) foltot látunk, melynek mérete és helye a nézőponttól függ, fejünket mozgatva a folt is mozog, mígnem eltűnik.

Az objektumok anyagának optikai tulajdonságainál megadhatjuk az objektum által kibocsátott fényt, továbbá azt, hogy az anyag milyen mértékben veri vissza a környezeti, a szórt és a tükrözött fényt. Az utóbbi három, az anyagra jellemző konstansok, az un. visszaverődési együtthatók. A megvilágításnak és az objektumok optikai tulajdonságainak a térbeliség érzékeltetésében játszott szerepét illusztrálja a 6.1. ábra.

a) b) c)

6.1. ábra. Gömb képei különböző fényösszetevők figyelembe vételével: *a*) megvilágítás nélkül; *b*) környezeti fénnyel és szórt visszaverődéssel; *c*) környezeti fénnyel és tükröző visszaverődéssel

Nincs lehetőség a fénytörés, az alakzatok optikai kölcsönhatásának (árnyékolás, objektumok tükröződése egymáson) és felületi érdesség modellezésére.

Az OpenGL-ben a különböző típusú fényeket, a fényvisszaverő képességet jellemző konstansokat RGBA komponenseikkel kell megadni, vagyis vektorként kezelendők (lásd a 4. fejezetet). Ezen színvektorokon a szokásos összeadás és skalárral való szorzás műveletén kívül egy speciális szorzást is értelmezünk, amit *-gal jelölünk. A * művelet az $\mathbf{I} = (I_R, I_G, I_B, I_A)$ és $\mathbf{J} = (J_R, J_G, J_B, J_A)$ vektorokhoz az $\mathbf{I} * \mathbf{J} = (I_R \cdot J_R, I_G \cdot J_G, I_B \cdot J_B, I_A \cdot J_A)$ vektort rendeli.

Az OpenGL-ben a megvilágítás befolyásolására három lehetőség van:

- fényforrások (fények) megadása;
- az anyagok optikai tulajdonságainak előírása;
- megvilágítási modell definiálása.

A továbbiakban azt feltételezzük, hogy a szín megadási módja RGBA, a színindex módra a fejezet végén térünk ki röviden.

## 6.1. Megvilágítási modell

Az OpenGL megvilágítási modelljének három összetevője van:

- globális környezeti fény,
- a nézőpont helye (végesben vagy végtelen távolban van),
- a poligonok felénk néző és hátsó oldalai egyformán, vagy különbözőképpen kezelendők.

A megvilágítási modellt a **glLightModel***() paranccsal adhatjuk meg.

void **glLightModel**{if} (GLenum *pname*, TYPE *param*);

*pname* értéke GL_LIGHT_MODEL_LOCAL_VIEWER és GL_LIGHT_MODEL_TWO_SIDE lehet.

void **glLightModel**{if}v (GLenum *pname*, const TYPE **param*);

*pname* értéke GL_LIGHT_MODEL_AMBIENT, GL_LIGHT_MODEL_LOCAL_VIEWER és GL_LIGHT_MODEL_TWO_SIDE lehet.

A GL_LIGHT_MODEL_AMBIENT paraméterrel a globális (fényforrásoktól független) környezeti fényt adhatjuk meg. Alapértelmezése $(0.2, 0.2, 0.2, 1.)$, ami azt biztosítja, hogy lássunk valamit még akkor is, ha nem adunk meg egyetlen fényforrást sem.

A GL_LIGHT_MODEL_LOCAL_VIEWER paraméter a tükröződő fény hatására keletkező fényes folt(ok) számítását befolyásolja. Ezen folt kiszámításához a csúcspontbeli normális, a csúcspontból a fényforrásba, valamint a nézőpontba mutató vektorok szükségesek. Ha a nézőpont végtelen távoli, vagyis a teret párhuzamosan vetítjük a képsíkra, akkor a csúcspontokból a nézőpontba mutató irányok megegyeznek, ami leegyszerűsíti a számításokat. A végesben lévő (valódi) nézőpont használatával valószerűbb eredményt kapunk, de a számítási igény megnő, mivel minden egyes csúcspontra ki kell számítani a nézőpontba mutató irányt. A nézőpont helyének ilyen értelmű "megváltoztatása" nem jelenti a nézőpont helyének tényleges megváltoztatását, ez pusztán a megvilágítási számításokra van hatással.

A GL_LIGHT_MODEL_LOCAL_VIEWER paraméter alapértelmezése GL_FALSE, mely esetben a rendszer a nézőpontkoordináta-rendszer $z$ tengelyének $(0., 0., 1., 0.)$ pontját használja nézőpontként a tükröződő visszaverődés számításánál. A paraméter GL_TRUE értéke esetén a nézőpont a nézőpontkoordináta-rendszer origója lesz.

Mint azt a 3.3.3. pontban láttuk, az OpenGL a poligonok oldalait megkülönbözteti, így beszélhetünk felénk néző és hátsó poligonokról (amelyek hátoldalát látjuk). Nyílt felületek (ezeket közelítő poligonháló) esetén a nem felénk néző poligonokat – az előző terminológia szerint a hátsó lapokat – is láthatjuk, de az ezekkel kapcsolatos megvilágítási számítások nem a valóságot tükröző eredményre vezetnének. Ezért lehetőség van arra, hogy a poligonokat kétoldalúnak tekintve, mindkét oldalt a megvilágításnak megfelelően ábrázoljunk. Ezt úgy érhetjük el, hogy a GL_LIGHT_MODEL_TWO_SIDE paraméternek a GL_TRUE értéket adjuk. Ekkor a rendszer a hátsó lapok normálisait megfordítja ($-1.$-el megszorozza) és így végzi el a számításokat, ami már helyes eredményre vezet. Ez

a lehetőség a GL_LIGHT_MODEL_TWO_SIDE paraméternek adott GL_FALSE értékkel szüntethető meg, ami egyben az alapértelmezés is.

Ahhoz, hogy az OpenGL megvilágítási számításokat végezzen, engedélyezni kell a megvilágítást a **glEnable**(GL_LIGHTING); paranccsal. A megvilágítást a **glDisable**(GL_LIGHTING); paranccsal tilthatjuk le. Alapértelmezés szerint a megvilágítás nem engedélyezett, ilyen esetben a csúcspont színét a megadásakor kurrens rajzolási szín határozza meg.

## 6.2. Fényforrás megadása

Az OpenGL-ben legalább nyolc fényforrást adhatunk meg. Egy-egy fényforrásnak sok jellemzője van, mint pl. színe, helye, iránya, és egyenként lehet őket ki- és bekapcsolni.

void **glLight**{if} (GLenum *light*, GLenum *pname*, TYPE *param*);

void **glLight**{if}**v** (GLenum *light*, GLenum *pname*, const TYPE **param*);

A *light* azonosítójú fényforrást hozza létre. Minden OpenGL implementációban legalább 8 fényforrást lehet megadni, a pontos felső korlátot a GL_MAX_LIGHTS paraméterrel meghívott **glGetIntegerv**() paranccsal kaphatjuk meg. A *light* paraméter értéke GL_LIGHT$i$ $0 \leq i <$ GL_MAX_LIGHTS lehet. A függvény meghívásával a *light* fényforrásnak a *pname* paraméter értékét lehet megadni. A függvény első alakjával (a skalár változattal) csak az egyetlen skalárral megadható paraméterek állíthatók be. A a 6.1. táblázat *pname* lehetséges értékeit és azok alapértelmezését tartalmazza. Ha az alapértelmezés a GL_LIGHT0 és a GL_LIGHT$i$, $(i > 0)$ fényforrásokra különbözik, akkor azokat külön feltüntetjük.

Az $i$-edik fényforrást a **glEnable**(GL_LIGHT$i$); paranccsal lehet bekapcsolni, a **glDisable**(GL_LIGHT$i$); paranccsal pedig ki lehet kapcsolni.

### 6.2.1. A fény színe

A fényforrás által kibocsátott fény három összetevőből áll: a környezeti fény (ambient light, GL_AMBIENT), a szórt fény (diffuse light, GL_DIFFUSE) és a tükrözött fény (specular light, GL_SPECULAR). Mindhárom színét az RGBA komponenseivel kell megadni. A környezeti fényösszetevő azt adja meg, hogy az adott fényforrás milyen mértékben járul hozzá a térrész környezeti fényéhez. A fényforrás szórt fényének a színe fejezi ki leginkább azt, amit mi a fény színének látunk. A tükrözött fényösszetevő befolyásolja az objektumok képén megjelenő fényes folt színét. A sima felületű tárgyakon (pl. üveg) tapasztalhatunk ilyen jelenséget, és a fényes folt leggyakrabban fehér színűnek látszik. A valószerű hatás érdekében a GL_DIFFUSE és GL_SPECULAR paramétereknek célszerű ugyanazt az értéket adni.

### 6.2.2. A fényforrások helye

A GL_POSITION paraméterrel a fényforrás helyét adhatjuk meg. Ez lehet végesben lévő, vagy végtelen távoli ($w = 0$) pont is. A végtelen távolban lévő fényforrásból kibocsátott

sugarak párhuzamosak – ilyennek tekinthetők a valóságban a napsugarak –, és a fényforrás pozíciója ezek irányával adható meg. (Vigyázzunk, itt nem valódi homogén koordinátákkal kell megadnunk a fényforrás helyét, ugyanis a $-1$-gyel való szorzás más eredményt ad!) A végesben lévő fényforrások – sok rendszerben ezeket pontszerű fényforrásnak is nevezik – azonos intenzitású fényt bocsátanak ki minden irányba. A rendszer a fényforrás helyét (akár végesben, akár végtelen távolban van) a kurrens nézőpont-modell transzformációval a nézőpontkoordináta-rendszerbe transzformálja, mivel a megvilágítással kapcsolatos számításokat itt végzi el.

Végesben lévő fényforrás esetén megadható, hogy a fényforrástól távolodva milyen mértékben csökken a fény erőssége, azaz hogyan tompul a fény (attenuation). A fény tompulását a

$$T = \min\left\{\frac{1}{c_0 + c_1 d + c_2 d^2}, 1\right\} \tag{6.1}$$

kifejezéssel írja le a rendszer, ahol $d$ a csúcspontnak a fényforrástól mért távolsága, $c_0$ a távolságtól független (GL_CONSTANT_ATTENUATION), $c_1$ a távolsággal arányos (GL_LINEAR_ATTENUATION), $c_2$ pedig a távolság négyzetével arányos (GL_QUADRATIC_ATTENUATION ) együtthatója a fény tompulásának. Alapértelmezés szerint $c_0 = 1.$, $c_1 = c_2 = 0.$, azaz $T = 1.$, vagyis a fény nem tompul. Ezt az értéket használja a rendszer végtelen távoli fényforrás esetén is. A fénytompulás a fényforrásból kibocsátott fény mindhárom összetevőjére (környezeti, szórt, tükrözött) hat, de a globális környezeti fényre és az ábrázolt objektumok által kibocsátott fényre nem.

### 6.2.3. Reflektor

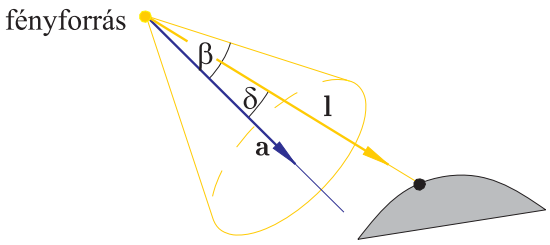
Végesben lévő fényforrásból reflektort is létrehozhatunk. Ennek érdekében a kibocsátott fénysugarakat forgáskúp alakú ernyővel lehatároljuk. Reflektor létrehozásához meg kell adnunk a fénykúp tengelyének irányát (GL_SPOT_DIRECTION) és fél nyílásszögét (GL_SPOT_CUTOFF), továbbá azt, hogy a fény intenzitása hogyan csökken a kúp tengelyétől a palást felé haladva. Reflektor fényforrásból csak akkor jut el fény a csúcspontba, ha a csúcspontot és a fényforrást összekötő egyenes és a fénykúp tengelyének $\delta$ szöge kisebb, mint a kúp $\beta$ fél nyílásszöge (lásd a 6.2. ábrát), vagyis ha $\delta < \beta$, azaz $\cos\delta > \cos\beta$, mivel $\beta \in [0., 90.]$. A számítások során a

$$\cos\delta = \frac{\mathbf{a}\cdot\mathbf{l}}{|\mathbf{a}|\,|\mathbf{l}|}$$

kifejezést célszerű használni, ahol $\mathbf{a}$ és $\mathbf{l}$ értelmezése a 6.2. ábra szerinti. Az $\mathbf{I}_l$ fényt kibocsátó reflektornak $\delta$ függvényében bekövetkező intenzitáscsökkenését az $\mathbf{I}_l \cos^r\delta$ kifejezéssel írjuk le, ahol $r$ (GL_SPOT_EXPONENT) nemnegatív szám. Tehát $r = 0.$ esetén a fény intenzitása nem változik a kúpon belül.

A GL_SPOT_CUTOFF paraméterrel a fénykúp $\beta$ fél nyílásszögét adhatjuk meg fokban. Az érték vagy 180., vagy egy $[0., 90.]$ intervallumbeli szám. Alapértelmezése 180., ami azt jelenti, hogy a fényforrás nem reflektor, mivel minden irányba bocsát ki fényt.

A GL_SPOT_DIRECTION paraméterrel a fénykúp $\mathbf{a}$ forgástengelyének irányát adhatjuk meg. Alapértelmezése $(0., 0., -1.)$. A rendszer ezt az irányt is transzformálja a kurrens nézőpont-modell transzformációval.




6.2. ábra. Reflektor fényforrás

A GL_SPOT_EXPONENT paraméterrel azt szabályozzuk, hogy a fény mennyire koncentrált, vagyis milyen mértékben csökken az erőssége a fénykúp tengelyétől a palást felé haladva. Minél nagyobb ez az érték, annál koncentráltabb a fény. Az alapértelmezés 0., ami azt jelenti, hogy a fény erőssége nem csökken a palást felé haladva.

Az $i$-edik fényforrás reflektor hatását kifejező skalár

$$R_i = \begin{cases} 1, & \text{ha GL\_SPOT\_CUTOFF=180°, vagyis nem reflektor;} \\ 0, & \text{ha } \delta > \beta, \text{ vagyis a csúcspont a fénykúpon kívül van;} \\ \left( \max \left\{ \dfrac{\mathbf{a} \cdot \mathbf{l}}{|\mathbf{a}|\,|\mathbf{l}|}, 0 \right\} \right)^r & \text{egyébként;} \end{cases} \tag{6.2}$$

ahol **a** és **l** a 6.2. ábra szerinti vektorok, $r$ pedig az $i$-edik fényforrás GL_SPOT_EXPONENT paramétere.

## 6.3. Anyagtulajdonságok

Az OpenGL az ábrázolandó objektumok anyagának a következő optikai tulajdonságait veszi figyelembe:

- az objektum által kibocsátott fényt;
- az anyag milyen mértékben veri vissza a környezeti, a szórt és a tükrözött fényt;
- a ragyogást.

Ezeket a tulajdonságokat nagyrészt a **glMaterial***() paranccsal adhatjuk meg. Az anyagnak a különböző fényeket visszaverő képességét kifejező konstansokat RGBA komponenseikkel adhatjuk meg, vagyis pontosan úgy, mint magát a fényt.

Fontos tudni, hogy amennyiben a megvilágítás engedélyezett, az alakzatok színére a rajzolási színnek nincs hatása! Ilyen esetben az alakzatok színét a fényviszonyok és az alakzatok optikai anyagtulajdonságai határozzák meg.

void **glMaterial**{if} (GLenum *face*, GLenum *pname*, TYPE *param*);

void **glMaterial**{if}**v** (GLenum *face*, GLenum *pname*, const TYPE **param*);

A megvilágítási számításoknál használt anyagtulajdonságokat adhatjuk meg segítségével. A *face* paraméter értéke GL_FRONT, GL_BACK és

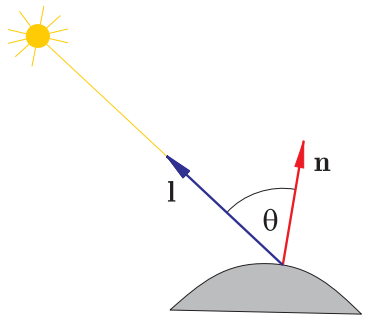
GL_FRONT_AND_BACK lehet, alapértelmezése GL_FRONT_AND_BACK. Ezzel azt állítjuk be, hogy a *pname* paraméterhez rendelt értékek a poligonok felénk néző oldalára, hátsó oldalukra, vagy mindkettőre vonatkozzanak. Ez a paraméter lehetővé teszi, hogy a poligonok különböző oldalai különböző optikai tulajdonságúak legyenek. *pname* lehetséges értékeit és a hozzájuk rendelhető értékek alapértelmezését a 6.2. táblázat tartalmazza. A függvény első (skalár) formája csak a GL_SHININESS paraméter megadásakor használható.

A GL_AMBIENT paraméterrel azt adjuk meg, hogy az anyag milyen mértékben veri vissza a környezeti fényt. A visszavert környezeti fény $\mathbf{I}_a * \mathbf{k}_a$, ahol $\mathbf{I}_a$ az ábrázolt térrészben jelenlévő összes környezeti fény, $\mathbf{k}_a$ pedig az anyag környezeti fény visszaverődési együtthatója.

A GL_DIFFUSE paraméterrel azt szabályozzuk, hogy a felület milyen mértékben veri vissza a csúcspontba eljutó szórt fényt. A visszaverés mértéke ezen felületi konstanson kívül a csúcspontbeli normálisnak és a csúcspontból a fényforrásba mutató iránynak a szögétől is függ. Minél kisebb ez a szög, annál nagyobb a visszaverődés. A pontos összefüggés

$$\mathbf{D} = \max\left\{\frac{\mathbf{l}\cdot\mathbf{n}}{|\mathbf{l}|\,|\mathbf{n}|}, 0\right\}\mathbf{I}_d * \mathbf{k}_d \tag{6.3}$$

ahol $\mathbf{l}$ a csúcspontból a fényforrásba (GL_POSITION) mutató vektor, $\mathbf{n}$ a csúcspontbeli normális ($\cos\alpha = \mathbf{l}\cdot\mathbf{n}/\left(|\mathbf{l}|\,|\mathbf{n}|\right)$), $\mathbf{I}_d$ a csúcspontba eljutó szórt fény, $\mathbf{k}_d$ pedig az anyag szórt visszaverődési együtthatója. $\mathbf{l}$, $\mathbf{n}$ és $\alpha$ értelmezése a 6.3. ábra szerinti.




6.3. ábra. Szórt visszaverődés

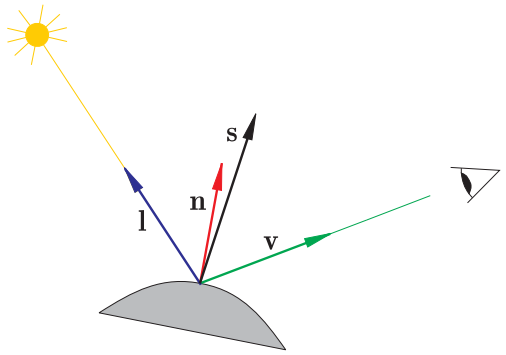
Minden fénykomponensnél és visszaverődési együtthatónál megadunk alfa (A) értéket is, a rendszer azonban csak az anyagtulajdonság GL_DIFFUSE paraméterénél megadott alfa értéket rendeli a csúcsponthoz, az összes többit figyelmen kívül hagyja. A GL_DIFFUSE paraméternél megadott fényvisszaverési konstans játssza a legnagyobb szerepet a felület színének kialakításában.

Míg a szórt visszaverődés mértéke függ a fényforrás és a felület kölcsönös helyzetétől, a visszavert környezeti fény ettől független.

A valóságban az anyagok környezeti és szórt visszaverődési együtthatója általában megegyezik, ezért az OpenGL lehetőséget biztosít ezek egyidejű megadására. Erre szolgál a GL_AMBIENT_AND_DIFFUSE paraméter.

A környezeti és szórt fények visszaverődésével ellentétben a szembe eljutó tükrözött visszaverődés függ attól is, hogy honnan nézzük az objektumokat. Ezen kívül szerepet

játszik még a csúcspontbeli normális, a fényforrás pozíciója, az anyag tükrözött visszaverődési együtthatója és az anyag ragyogási tényezője.




6.4. ábra. Tükrözött visszaverődés

A 6.4. ábra jelöléseit használva, a szembe eljutó tükrözött fény

$$\mathbf{S} = \begin{cases} 0, & \text{ha } \mathbf{l} \cdot \mathbf{n} \leq \mathbf{0} \\ \max\left\{ \left( \dfrac{\mathbf{s} \cdot \mathbf{n}}{|\mathbf{s}|\,|\mathbf{n}|} \right)^{s}, 0 \right\} \mathbf{I}_s * \mathbf{k}_s, & \text{egyébként} \end{cases} \tag{6.4}$$

ahol $\mathbf{s} = \mathbf{l}/|\mathbf{l}| + \mathbf{v}/|\mathbf{v}|$, $\mathbf{I}_s$ a csúcspontba eljutó tükrözött fény, $\mathbf{k}_s$ az anyag tükröző visszaverődési együtthatója (GL_SPECULAR), $s$ pedig az anyag ragyogási kitevője (GL_SHININESS).

Az anyag tükrözött visszaverődési együtthatóját a GL_SPECULAR paraméterrel, a ragyogási kitevőt pedig a GL_SHININESS-el adhatjuk meg. Az utóbbi a $[0., 128.]$ intervallumon változhat. Minél nagyobb ez a kitevő, a fényes terület annál kisebb és ragyogóbb (jobban fókuszált a fény).

A GL_EMISSION paraméterrel az objektum által kibocsátott fényt adhatjuk meg. Ezt a lehetőséget általában lámpák, vagy más fényforrások modellezéséhez használjuk. Ilyenkor célszerű még az adott objektumban egy fényforrást is létrehoznunk a világítótest hatás elérése érdekében.

## 6.4. Az anyagtulajdonságok változtatása

Egy-egy anyagtulajdonság mindaddig érvényben van, míg meg nem változtatjuk. A változtatás tulajdonságonként történik. Ezt megtehetjük az előzőekben ismertetett **glMaterial***() paranccsal, de arra is lehetőség van, hogy valamely anyagtulajdonságot a rajzolási színhez kapcsoljunk, amivel azt érjük el, hogy az adott anyagtulajdonság a rajzolási színnek (**glColor***()) megfelelően fog változni. Ezzel a programunk hatékonyságát növelhetjük.

void **glColorMaterial** (GLenum *face*, GLenum *mode*);

A poligonok *face* paraméterrel megadott oldalának a *mode* paraméterrel kijelölt tulajdonságát a kurrens rajzolási színhez kapcsolja. A kurrens rajzolási szín

megváltoztatása (**glColor***()) a *mode* anyagtulajdonság automatikus változtatását eredményezi. A *face* értéke GL_FRONT, GL_BACK és GL_FRONT_AND_BACK lehet, alapértelmezés GL_FRONT_AND_BACK. A *mode* paraméter a GL_AMBIENT, GL_DIFFUSE, GL_AMBIENT_AND_DIFFUSE, GL_SPECULAR és GL_EMISSION értékeket veheti fel, alapértelmezés a GL_AMBIENT_AND_DIFFUSE.

A **glColorMaterial**() parancs kiadása után engedélyeznünk kell a hozzákapcsolást a **glEnable**(GL_COLOR_MATERIAL) paranccsal, és ha már nem akarjuk használni az összekapcsolást, akkor a **glDisable**(GL_COLOR_MATERIAL) paranccsal le kell tiltanunk. A **glColorMaterial**() parancsot akkor használjuk, ha egyetlen anyagtulajdonságot kell gyakran – sok csúcspontban – változtatnunk, egyébként a **glMaterial***() parancs használata előnyösebb.

## 6.5. A csúcspontok színének meghatározása megvilágítás esetén

$n$ fényforrást feltételezve a csúcspontokba eljutó $\mathbf{V}$ fény

$$\mathbf{V} = \mathbf{I}_e + \mathbf{I}_g * \mathbf{k}_a + \sum_{i=0}^{n-1} T_i R_i \left(\mathbf{A}_i + \mathbf{D}_i + \mathbf{S}_i\right) \tag{6.5}$$

alakban írható fel, ahol

$\mathbf{I}_e$ az objektum által kibocsátott fény;

$\mathbf{I}_g$ a globális környezeti fény;

$T_i$ az $i$-edik fényforrás tompulása a (6.1) összefüggés szerint;

$R_i$ az $i$-edik fényforrás reflektor együtthatója a (6.2) összefüggés szerint;

$\mathbf{A}_i = \mathbf{I}_{ai} * \mathbf{k}_a$ az $i$-edik fényforrás környezeti fénykomponensének és az anyag környezeti fény visszaverődési együtthatójának a szorzata;

$\mathbf{D}_i$ az $i$-edik fényforrásból a szembe eljutó szórt fény a (6.3) összefüggés szerint;

$\mathbf{S}_i$ az $i$-edik fényforrásból a szembe eljutó tükrözött fény a (6.4) összefüggés szerint.

Ezen számítások után a rendszer a kapott RGBA értékeket a $[0., 1.]$ intervallumra levágja.

Ahhoz tehát, hogy az objektumok megvilágítás szerint legyenek ábrázolva definiálni kell egy megvilágítási modellt (**glLightModel**()); engedélyezni kell a megvilágítást (**glEnable**(GL_LIGHTING)); fényforrást kell létrehozni (**glLight***()) és azt be kell kapcsolni (**glEnable**(GL_LIGHT$i$)); anyagtulajdonságokat kell megadni (**glMaterial***()).

## 6.6. Megvilágítás színindex módban

Színindex módban az RGBA komponensekkel megadott megvilágítási paramétereknek vagy nincs hatásuk, vagy speciálisan értelmezettek. Ha csak lehet, kerüljük el a megvilágítás használatát színindex módban. Az RGBA formában megadott fényforrással kapcsolatos paraméterek közül csak a GL_DIFFUSE és GL_SPECULAR értékeket használja a rendszer. Az $i$-edik fényforrás szórt, illetve tükrözött fénykomponensét színindex módban a

$$d_{ci} = 0.3R(d_l) + 0.59G(d_l) + 0.11B(d_l)$$
$$s_{ci} = 0.3R(s_l) + 0.59G(s_l) + 0.11B(s_l)$$

összefüggésekkel számítja ki a rendszer, ahol $R(n), G(n), B(n)$ a színpaletta $n$-edik színének RGB komponenseit jelöli. A $0.3, 0.59, 0.11$ együtthatók az emberi szem színérzékelő képességének felelnek meg, vagyis a szem a zöld színre a legérzékenyebb és a kékre a legkevésbé. A

| void **glMaterial**{if}**v** (GLenum *face*, GL_COLOR_INDEXES, const TYPE **param*); |
|---|

paranccsal az anyag színének környezeti, szórt és tükrözött összetevőjét, pontosabban azok színindexeit adhatjuk meg. Alapértelmezés szerint a környezeti komponens színindexe 0., a szórt és a tükrözött komponensé 1. A **glColorMaterial**() parancsnak nincs hatása színindex módban.

Mint az várható, a csúcspontból a szembe eljutó fény színének kiszámítása is másképp történik, mint RGBA módban. A (6.5) kifejezésben a kibocsátott és a globális környezeti fény ($\mathbf{I}_e$ és $\mathbf{I}_g$ ) $\mathbf{0}$, a fénytompulást és a fényszóró hatást $(T_i, R_i)$ ugyanúgy számítja ki a rendszer, mint RGBA módban. A fényforrásoknak nincs környezeti fény komponense ($\mathbf{A}_i = \mathbf{0}$); a $\mathbf{D}_i$ kiszámítására használt (6.3) kifejezésben $\mathbf{I}_d * \mathbf{k}_d$-t helyettesítsük $d_{ci}$-vel, az $\mathbf{S}_i$ számításához használt (6.4) kifejezésben $\mathbf{I}_s * \mathbf{k}_s$-t $s_{ci}$-vel. Az így módosított szórt és tükrözött összetevők összegét jelöljük $d$-vel illetve $s$-el.

$$s_0 = \min\{s, 1\}$$
$$c_0 = a_m + d\,(1 - s_0)\,(d_m - a_m) + s_0\,(s_m - a_m)$$

ahol $a_m, d_m, s_m$ az anyagnak a GL_COLOR_INDEXES paraméterrel megadott környezeti, szórt és tükrözött fény színének indexe. A csúcspont végső színe

$$c = \min\{c_0, s_m\}$$

Az így kapott értéket a rendszer egész számmá konvertálja, majd bitenkénti és (AND) műveletbe hozza a $2^n - 1$ értékkel, ahol $n$ a színindex-pufferben az egy szín számára fenntartott bitek száma.

6.1. táblázat. A fényforrást meghatározó paraméterek

| *pname* | alapértelmezése | jelentése |
|---|---|---|
| GL_AMBIENT | $(0., 0., 0., 1.)$ | a fény környezeti összetevőjének RGBA komponensei |
| GL_DIFFUSE | GL_LIGHT0 esetén $(1., 1., 1., 1.)$ egyébként $(0., 0., 0., 1.)$ | a fény szórt összetevőjének RGBA komponensei |
| GL_SPECULAR | GL_LIGHT0 esetén $(1., 1., 1., 1.)$ egyébként $(0., 0., 0., 1.)$ | a fény tükröző összetevőjének RGBA komponensei |
| GL_POSITION | $(0., 0., 1., 0.)$ | a fényforrás helye $(x, y, z, w)$ |
| GL_SPOT_DIRECTION | $(0., 0., -1.)$ | a reflektor $(x, y, z)$ iránya |
| GL_SPOT_EXPONENT | 0. | a reflektor fényerejének csökkenése |
| GL_SPOT_CUTOFF | 180. | a reflektor forgáskúpjának fél nyílásszöge |
| GL_CONSTANT_ATTENUATION | 1. | a fény tompulásának konstans tagja |
| GL_LINEAR_ATTENUATION | 0. | a fénytompulás lineáris tagjának együtthatója |
| GL_QUADRATIC_ATTENUATION | 0. | a fénytompulás másodfokú tagjának együtthatója |

6.2. táblázat. Az anyagtulajdonság paraméterei

| *pname* | alapértelmezés | jelentés |
|---|---|---|
| GL_AMBIENT | $(0.2, 0.2, 0.2, 1.)$ | környezeti fény visszaverődési együttható |
| GL_DIFFUSE | $(0.8, 0.8, 0.8, 1.)$ | szórt fény visszaverődési együttható |
| GL_AMBIENT_AND_DIFFUSE | $(0.8, 0.8, 0.8, 1.)$ | környezeti és szórt fény visszaverődési együttható |
| GL_SPECULAR | $(0., 0., 0., 1.)$ | tükröző fény visszaverődési együttható |
| GL_SHININESS | 0. | ragyogás |
| GL_EMISSION | $(0., 0., 0., 1.)$ | kibocsátott fény |
| GL_COLOR_INDEXES | $(0, 1, 1)$ | környezeti, szórt és tükröző visszaverődés színindexe |

# 7. fejezet

# Display-lista

A display-lista OpenGL parancsok csoportja, melyet későbbi végrehajtás céljából tárolunk. A display-lista meghívásakor a benne tárolt parancsokat abban a sorrendben hajtja végre a rendszer, ahogy korábban a listára kerültek. Néhány – a későbbiekben felsorolt – kivételtől eltekintve, az OpenGL parancsok akár listában tárolhatók, akár azonnal végrehajthatók. Ez a két lehetőség tetszőlegesen kombinálható egy programon belül.

A display-lista lehetőséget elsősorban a hálózati környezetben futtatott programok optimális működése érdekében hozták létre. Az optimalizálás módja és mértéke implementációfüggő, az azonban garantált, hogy a display-listák használata sohasem csökkentheti a teljesítményt, még akkor sem, ha egyetlen gépen dolgozunk – a kliens és a szerver gép megegyezik –, ugyanis a rendszer, ha csak lehetséges, a grafikus hardver igényeinek megfelelően tárolja a lista parancsait. A display-lista parancs cache, nem pedig dinamikus adatbázis, ennek következtében tartalma utólag nem módosítható és a felhasználó nem fér hozzá a listán tárolt adatokhoz.

Arra nincsen garancia, hogy minden OpenGL implementáció optimalizálja a display-lista működését, de az biztos, hogy egyetlen implementációban sem kevésbé hatékony a display-listán tárolt parancsok végrehajtása, mint ugyanazok közvetlen módban való végrehajtása. Azt azonban figyelembe kell vennünk, hogy a display-listák használata némi adminisztrációval jár, ezért túl rövid listák használata esetén a lista végrehajtásából származó előny elhanyagolhatóvá válhat az adminisztráció okozta plusz munkához képest.

Az alábbiakban az alkalmazások néhány optimalizálási lehetőségét soroljuk fel, vagyis azt, hogy milyen esetekben célszerű display-listákat használni.

- Mátrixműveletek (lásd az 5. fejezet). Mivel a mátrixok kiszámítása időigényes lehet, továbbá általában a mátrixok inverzére is szükség van.
- Raszteres bittérképek és képek (lásd a 9. fejezet). Az a forma, ahogyan megadjuk a raszteres adatokat, általában nem egyezik meg a grafikus hardver számára szükséges formátummal. A display-lista lefordítása során az OpenGL valószínűleg a hardver számára optimális formátumban tárolja, ezért a lista végrehajtása sokkal hatékonyabb lehet, mint a parancsok közvetlen kiadása.
- Fényforrások, anyagtulajdonságok, megvilágítási modell (lásd a 6. fejezetet).
- Textúrák (lásd a 13. fejezetet).

- Poligonok kitöltése mintával.
- Köd, atmoszférikus hatások (lásd a 8.3. szakaszt).
- Opcionális vágósíkok használata (lásd az 5.2. szakaszt).

## 7.1. Display-lista létrehozása, végrehajtása

Display-listát a **glNewList**() és **glEndList**() parancsok között kiadott OpenGL parancsokkal lehet létrehozni. Egyszerre csak egy lista hozható létre. A **glEndList**() parancs kiadása OpenGL hibát eredményez, ha nem előzi meg lezáratlan **glNewList**() parancs.

void **glNewList** (GLenum *list*, GLenum *mode*);

Display-lista kezdetét jelöli meg. Az ezt követő OpenGL parancsok – a későbbiekben felsorolandó kivételektől eltekintve – a listára kerülnek mindaddig, amíg a **glEndList**() parancsot ki nem adjuk. A **glNewList**() és **glEndList**() zárójelpár között kiadott listán nem tárolható parancsokat közvetlenül végrehajtja a rendszer a lista létrehozása során. A *list* paraméter pozitív egész szám lehet, mely a lista globális azonosítója lesz. Ha már létezik ilyen azonosítójú lista, akkor azt a rendszer bármiféle figyelmeztetés nélkül felülírja. A *mode* paraméter értéke GL_COMPILE vagy GL_COMPILE_AND_EXECUTE lehet. GL_COMPILE esetén a parancsok nem kerülnek végrehajtásra miközben a listára kerülnek, tehát csak a display-listának megfelelő formátumra konvertálja a rendszer a parancsokat, és tárolja őket. GL_COMPILE_AND_EXECUTE esetén a konvertálás és tárolás mellet a rendszer végre is hajtja a parancsokat úgy, mintha közvetlen végrehajtási módban adtuk volna ki azokat.

void **glEndList** (void);

A display-lista végét jelöli meg, kiadását egy lezáratlan **glNewList**() parancsnak kell megelőznie.

## 7.2. A display-lista tartalma

A listán a kifejezések, paraméterek értéke kerül tárolásra, nem pedig a paramétereket tartalmazó vektor címe, vagy a pl. transzformációs mátrixok esetén maga a mátrix, és nem a kiszámításához szükséges adatok. A lefordított display-listát a szerver gépen (amin a kép megjelenik) tárolja a rendszer, ezért azok a parancsok, amelyek eredménye a kliens (a parancsot kiadó gép) állapotától függ, nem tárolhatók display-listán. A lista végrehajtásakor ugyanis a szervernek nem állnak rendelkezésére a szükséges adatok. A listán nem tárolható parancsok általában vagy valamilyen értéket adnak vissza (**glGet***(), **glIs***()), vagy a klienssel kapcsolatos információtól függnek (pl. **glFlush**(), **glFinish**()), vagy a kliens által kezelt állapotváltozó értékét változtatják meg. A következő parancsok nem tárolhatók display-listán:

| | | |
|---|---|---|
| **glDeleteLists**() | **glGenLists**() | **glPixelStore**() |
| **glFeedbackBuffer**() | **glGet***() | **glReadPixels**() |
| **glFinish**() | **glIsEnable**() | **glRenderMode**() |
| **glFlush**() | **glIsList**() | **glSelectBuffer**() |

A display-listák tetszőlegesen sokszor végrehajthatók, továbbá display-listák végrehajtása és közvetlen hatású parancsok kiadása tetszés szerint kombinálható.

void **glCallList** (GLuint *list*);

A *list* azonosítójú display-listát hajtja végre. A lista parancsai a listára kerülés sorrendjében hajtódnak végre. Ha nem létezik *list* azonosítójú display-lista, akkor nem történik semmi.

A display-listák tartalmazhatnak olyan parancsokat, melyek megváltoztatják a rendszer globális változóinak értékeit (pl. attribútumok, transzformációs mátrixok). Ezek a változtatások természetesen a lista végrehajtása után is érvényben maradnak, ha nem teszünk semmilyen óvintézkedést. Ha azt akarjuk, hogy az ilyen jellegű változások csak a display-lista végrehajtásának idejére korlátozódjanak, akkor az attribútumokat, illetve a transzformációs mátrixokat a megfelelő verem használatával a lista végrehajtása előtt mentsük el (**glPushAttrib**(), **glPushMatrix**()), majd a lista végrehajtás után állítsuk vissza (**glPopAttrib**(), **glPopMatrix**()).

A display-listákat programunkban bárhol használhatjuk, mivel indexük (azonosítójuk) globális és egyedi. Nincs lehetőség azonban a listák tartalmának adatfájlba mentésére, így természetesen beolvasására sem. A display-lista használata tehát a program futásának idejére korlátozott.

## 7.3. Hierarchikus diplay-listák

A display-listák tartalmazhatnak display-lista hívásokat, amivel listák hierarchikus rendszere hozható létre. Ez hasznos lehet pl. ismétlődő részeket tartalmazó objektumok ábrázolásakor. Az ilyen jellegű hívási láncnak van korlátja, ez bármely implementációban legalább 64, a pontos számot a GL_MAX_LIST_NESTING paraméterrel kiadott **glGetIntegerv**() paranccsal kaphatjuk meg. Nem szükséges, hogy egy lista a meghívásakor már létezzen. Egy nem létező lista meghívásának semmilyen következménye sincs. A hierarchikus display-listák segítségével közvetve módosítható egy lista tartalma, miután a meghívott listát bármikor felülírhatjuk, azaz ugyanazzal az azonosítóval új tartalmú listát hozhatunk létre, ami által a hívó lista tartalma (hatása) is megváltozik. Ez a módszer semmiképp sem biztosít optimális memória-felhasználást és maximális hatékonyságot, de adott esetben elfogadható, hasznos megoldás lehet.

## 7.4. Display-listák és indexek kezelése

A display-listák indexe tetszőleges pozitív egész szám lehet. A véletlen felülírás elkerülése érdekében a **glGetLists**() paranccsal kérhetünk a rendszertől egymást követő, még nem használt azonosítókat, vagy a **glIsList**() paranccsal lekérdezhetjük, hogy egy adott azonosító használatban van-e.

GLuint **glGenLists** (GLsizei *range*);

*range* darab egymást követő, használaton kívüli display-lista indexet allokál. A visszaadott érték nulla, ha nem tudja a rendszer a megfelelő számú indexet allokálni vagy *range = 0*; egyébként pedig a lefoglalt indextömb első eleme. A lefoglalt indexekhez a rendszer egy-egy üres display-listát hoz létre.

GLboolean **glIsList** (GLuint *list*);

Ez a parancs GL_TRUE értéket ad vissza, ha *list* indexű display-lista már létezik (az azonosító már foglalt), egyébként pedig a GL_FALSE értéket.

A **glDeleteLists**() paranccsal egymást követő indexű display-listákat törölhetünk, ami által indexüket felszabadítjuk.

void **glDeleteLists** (GLuint *list*, GLsizei *range*);

A *list* indextől kezdve *range* darab egymást követő indexű display-listát töröl. Nem létező listák törlésének nincs semmilyen következménye.

## 7.5. Több lista végrehajtása

Az OpenGL-ben hatékony mechanizmus van több display-lista egymás utáni végrehajtására. Ehhez a display-lista indexeket egy tömbbe kell tenni és a **glCallLists**() parancsot kell kiadni. Kézenfekvő ezen lehetőség használata, ha az indexeknek valamilyen jelentésük van, mint pl. karakterek esetén az ASCII kódoknak. Több karakterkészlet használata esetén mindegyik display-listának egy külön bázisindexet kell létrehozni. Ezt a célt szolgálja a **glListBase**() parancs.

void **glListBase** (GLuint *base*);

Azt az eltolást (offsetet) hozza létre, amit a rendszer a **glCallLists**() paranccsal végrehajtott display-lista indexeihez hozzáad a tényleges index kiszámításához. A display-lista bázisának kezdeti értéke 0. A bázisértéknek nincs hatása a **glCallList**() és **glNewList**() parancsok eredményére.

void **glCallLists** (GLsizei *n*, GLenum *type*, const GLvoid **lists*);

*n* darab display-listát hajt végre. A végrehajtandó listák indexeit úgy számítja ki, hogy a *list* címen kezdődő előjeles egész értékekhez hozzáadja – a **glListBase**() paranccsal létrehozott – kurrens bázisértéket.

A *type* paraméterrel a *list* címen kezdődő indexek adattípusát (méretét) kell megadni. Ez a típus általában az alábbi értékek valamelyike: GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT. A GL_2_BYTES, GL_3_BYTES, GL_4_BYTES értékek is adhatók a típusnak. Ebben az esetben két, három vagy négy egymást követő byte-ot tol el és ad össze a rendszer a display-lista offset kiszámításához, az alábbi algoritmus szerint:

```
/* b= 2, 3, 4; a byte-ok 0, 1, 2, 3-al vannak számozva a tömbben */
offset = 0;
for(i = 0; i < b; i++)
    {
    offset = offset << 8;
    offset += byte[i];
    }
index = offset + listbase;
```

# 8. fejezet

# Speciális optikai hatások

Az itt tárgyalandó három technikával a képek minősége javítható, valószerűségük növelhető.

## 8.1. Átlátszóság

Az RGBA színkomponensek közül az alfa ($A \in [0., 1.]$) érték az átlátszóság modellezésére használható. Ezzel ugyanis azt írhatjuk elő, hogy az új fragmentum színe milyen mértékben vegyüljön a pixel jelenlegi színével. Ha $A = 1.$, akkor az új szín teljesen elfedi (felülírja) a régit, tehát az új szín tökéletesen fed; $A = 0.$ esetén pedig egyáltalán nem fed, azaz a pixel régi színe minden változtatás nélkül megmarad. Minden közbülső érték azt eredményezi, hogy a pixel új színe az új fragmentum színének és a pixel régi színének valamilyen kombinációja lesz. Az alfa értéket tehát a legegyszerűbb a szín fedési képességének felfogni.

Ha a színvegyítés engedélyezett, akkor a művelet a raszterizálás és fragmentálás után zajlik le, tehát közvetlenül a pixelbe való írás előtt.

Az alfa értékek az un. alfa tesztre is használhatók, vagyis arra, hogy az alfa értékétől függően egy új fragmentumot elfogadjon vagy figyelmen kívül hagyjon a megjelenítő rendszer, lásd a 11.2. pontot.

Miután színindex módban nem adunk meg alfa értékeket, az átlátszóságot csak RGBA módban tudjuk modellezni.

Az OpenGL-ben az új fragmentumot forrásnak (source), a neki megfelelő pixelt – amire leképezi a rendszer – pedig célnak nevezzük. A forrás és a cél színének vegyítése érdekében engedélyeznünk kell ezt a lehetőséget a **glEnable**(GL_BLEND) parancs kiadásával, és meg kell adnunk, hogy a rendszer a forrás és cél színösszetevőinek kombináló tényezőit milyen módon számítsa ki. Az együtthatókat is RGBA komponenseikkel kell megadni csakúgy, mint a fényvisszaverő képességeket leíró konstansokat. A rendszer a vegyített szín komponenseit az

$$\mathbf{s} * \mathbf{S} + \mathbf{d} * \mathbf{D}$$

összefüggéssel számítja ki, ahol $\mathbf{S} = (S_R, S_G, S_B, S_A)$ az új fragmentum (a forrás) színe, $\mathbf{s} = (s_R, s_G, s_B, s_A)$ annak kombináló tényezője; $\mathbf{D} = (D_R, D_G, D_B, D_A)$ a pixel (a cél) jelenlegi színe, $\mathbf{d} = (d_R, d_G, d_B, d_A)$ pedig annak kombináló tényezője. A * karakter a 6. fejezetben bevezetett vektorműveletet jelöli.

A forrás és cél kombináló tényezőjének kiszámítási módját a **glBlendFunc**() paranccsal adhatjuk meg.

void **glBlendFunc** (GLenum *sfactor*, GLenum *dfactor*);

Az *sfactor* paraméterrel az új fragmentum (a forrás) együtthatójának, a *dfactor* paraméterrel pedig az új fragmentumnak megfelelő pixel színkombináláshoz használt együttható kiszámítási módját írhatjuk elő. Az *sfactor* alapértelmezése GL_ONE, a *dfactor*é pedig GL_ZERO. Az *sfactor* és *dfactor* paraméterek lehetséges értékeit a a 8.1. táblázat tartalmazza.

8.1. táblázat. Az átlátszósághoz haználható szimbolikus konstansok

| konstans | *sf* | *df* | a kiszámított együttható |
|---|---|---|---|
| GL_ZERO | + | + | $(0, 0, 0, 0)$ |
| GL_ONE | + | + | $(1, 1, 1, 1)$ |
| GL_DST_COLOR | + | − | $(D_R, D_G, D_B, D_A)$ |
| GL_SRC_COLOR | − | + | $(S_R, S_G, S_B, S_A)$ |
| GL_ONE_MINUS_DST_COLOR | + | − | $(1 - D_R, 1 - D_G, 1 - D_B, 1 - D_A)$ |
| GL_ONE_MINUS_SRC_COLOR | − | + | $(1 - S_R, 1 - S_G, 1 - S_B, 1 - S_A)$ |
| GL_SRC_ALPHA | + | + | $(S_A, S_A, S_A, S_A)$ |
| GL_ONE_MINUS_SRC_ALPHA | + | + | $(1 - S_A, 1 - S_A, 1 - S_A, 1 - S_A)$ |
| GL_DST_ALPHA | + | + | $(D_A, D_A, D_A, D_A)$ |
| GL_ONE_MINUS_DST_ALPHA | + | + | $(1 - D_A, 1 - D_A, 1 - D_A, 1 - D_A)$ |
| GL_SRC_ALPHA_SATURATE | + | − | $(m, m, m, 1)$; $m = \min(S_A, 1 - D_A)$ |

A 8.1. táblázatban szereplő forrás és cél kombináló tényezők nem minden párosításának van értelme, a felhasználói programok többnyire csak néhány párosítást használnak. Az alábbiakban felsorolunk néhány jellemző párosítást. Felhívjuk a figyelmet arra, hogy ezek közül néhány csak a forrás alfa értékét használja, vagyis abban az esetben is alkalmazható, ha a kép pixeleihez az alfa értéket nem tárolja a grafikus hardver. Vegyük figyelembe azt is, hogy ugyanaz a hatás többféleképpen is elérhető.

- Két kép egyenlő mértékű vegyítését (félig az egyik, félig a másik) elérhetjük pl. úgy, hogy az *sfactor* paramétert GL_ONE-ra állítjuk, megrajzoljuk az első képet; ezután mind az *sfactor*, mind a *dfactor* paramétereket GL_SRC_ALPHA-ra állítjuk és $A = 0.5$ alfával megrajzoljuk a második képet. Ha az első és második képet $0.75 : 0.25$ arányban akarjuk vegyíteni, akkor rajzoljuk meg az első képet úgy mint az előbb, majd a második képet *sfactor* = GL_SRC_ALPHA, *dfactor* = GL_ONE_MINUS_SRC_ALPHA paraméterekkel $A = 0.25$ érték mellett. Ez a típusú vegyítés a leggyakrabban használt.

- Három különböző kép egyenlő arányú vegyítéséhez *dfactor* = GL_ONE, *sfactor* = GL_SRC_ALPHA paraméterek mellett $A = 0.33333$ értékkel rajzoljuk meg mindhárom képet.

- Ecsettel való festés során az ecsetvonásokkal fokozatosan tudjuk elfedni a háttér színét. Ennek modellezése érdekében az *sfactor* = GL_SRC_ALPHA, *dfactor* = GL_ONE_MINUS_SRC_ALPHA paraméterekkel az ecsetvonás fedésének megfelelő (pl. 10%-os) alfa értékkel rajzoljuk meg az ecset képét. A valószerűség fokozása érdekében az ecset különböző részein változtathatjuk a fedést, az ecset közepén nagyobb, a szélein kisebb. Hasonló technikával tudunk radírt modellezni, ekkor a rajzolás színének a háttér színével kell megegyeznie.

- Az a színkombinálási számítás, amikor az *sfactor* paraméternek a GL_DST_COLOR vagy GL_ONE_MINUS_DST_COLOR értéket adjuk, a *dfactor* paraméternek pedig a GL_SRC_COLOR vagy GL_ONE_MINUS_SRC_COLOR értéket, lehetővé teszi a színkomponensenkénti szabályozást. Ezt a hatást egyszerű szűrő használatával is elérhetjük, ugyanis pl. a vörös komponenst 0.6-al, a zöldet 0.8-al a kéket 0.4-el megszorozva ugyanazt a hatást érjük el, mintha a vörös fényt 40 %-al, a zöldet 20 %-al a kéket 60 %-al csökkentő szűrőt alkalmaznánk.

- Tegyük fel, hogy egy átlátszatlan háttér előtt három különböző mértékben átlátszó, egymást részben eltakaró objektumot akarunk megrajzolni. Feltételezzük továbbá, hogy a legtávolabbi objektum a mögötte lévő szín 80 %-át, a középső 40 %-át, a legközelebbi pedig 90 %-át engedi át. Ennek modellezése érdekében a hátteret rajzoljuk meg az *sfactor* és *dfactor* paraméterek alapértelmezése szerint, majd az *sfactor* = GL_SRC_ALPHA, *dfactor* = GL_ONE_MINUS_SRC_ALPHA paraméterek mellett rajzoljuk meg a legtávolabbi objektumot $A = 0.2$, a középsőt $A = 0.6$ és a legközelebbit $A = 0.1$ alfa értékkel.

- Nem négyszög alakú raszterképek hatása érhető el azáltal, hogy a kép különböző fragmentumaihoz különböző alfa értéket rendelünk, pl. 0-át minden nem látható és 1-et minden láthatóhoz. Ezt az eljárást billboardingnak is szokták nevezni. Egy térbeli fa képének illúzióját kelthetjük azáltal, hogy pl. két egymásra merőleges síkbeli négyszögre, az előző módon megadott fakorona mintázatú textúrát (egy fóliát, melyre lombkorona mintát raktunk) teszünk. Ez sokkal gyorsabb, mintha térbeli poligonnal közelítenénk a lombkoronát és azt textúráznánk.

Az átlátszóság (fedés) modellezési módja alapján látható, hogy más lesz az eredmény ha egymás képét átfedő poligonokat a nézőponthoz képesti távolságuk szerint hátulról előre, vagy elölről hátulra haladva jelenítünk meg. Néhány poligon esetén a megjelenítés sorrendjét könnyen meghatározhatjuk, azonban sok poligonból álló, vagy helyüket változtató objektumok, vagy a nézőpontot változtató megjelenítések esetén a helyes sorrend megállapítása gyakorlatilag kivitelezhetetlen. Ezt a problémát az OpenGL mélységpufferének (z-puffer) speciális használatával oldhatjuk meg. Ha a láthatóság szerinti ábrázolás engedélyezett (**glEnable**(GL_DEPTH_TEST)), a rendszer felülírja a mélységpufferban a pixel mélységét, ha az új fragmentum közelebb van a nézőponthoz, mint a pixelen tárolt. Ez mindaddig helyes eredményt ad, míg nincsenek átlátszó objektumok. Átlátszó objektum esetén ugyanis, ha a rendszer a nézőponthoz közelebbi fragmentum mélységével felülírja a z-puffer megfelelő elemét, akkor egy később megjelenítendő objektum, mely az átlátszó objektum mögött van nem fog látszani, mivel a nézőponttól távolabb van, mint a korábban megjelenített átlátszó objektum. Ezt a problémát úgy

tudjuk megoldani, hogy az átlátszó objektum rajzolása előtt a mélységpuffert a **glDepthMask**(GL_FALSE) paranccsal csak olvashatóvá tesszük, ami által az átlátszó objektum megrajzolása során a távolságokat össze tudja hasonlítani a rendszer, de nem tudja felülírni a puffert. A következő, nem átlátszó objektum rajzolása előtt újra írhatóvá kell tenni a mélységpuffert a **glDepthMask**(GL_TRUE) paranccsal. (Átlátszó objektumok rajzolása előtt ne felejtsük el letiltani a hátsó lapok eltávolítását!)

## 8.2. Simítás (antialiasing)

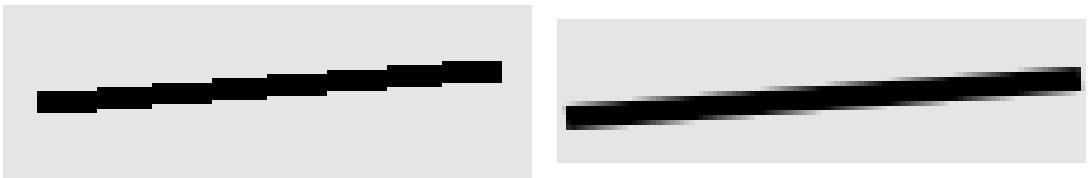
Raszteres grafikus display-n szakaszt rajzolva azt tapasztaljuk, hogy a nem függőleges vagy vízszintes szakaszok képének határa töredezett, lépcsős lesz. Kiváltképp így van ez a majdnem vízszintes és majdnem függőleges szakaszok esetén. Ezt a jelenséget angolul aliasingnak, a jelenség csökkentésére, a kép határának kisimítására irányuló technikákat antialiasingnak nevezik.

Az OpenGL-ben használt simítási technika a pixelek fedettségének számításán alapul. A szakaszok képe minden esetben téglalap lesz. A rendszer kiszámítja ezen téglalapot metsző pixelekre, hogy hányad részük esik a téglalap képére. Ezt a számot nevezzük fedettségi értéknek.

RGBA színmegadási mód esetén a rendszer a pixeleknek megfelelő fragmentumok alfa értékét megszorozza a megfelelő fedettségi értékkel. Az így kapott alfa értéket felhasználva a pixel színével vegyíthetjük a fragmentum színét a hatás elérése érdekében.

Színindex módban a rendszer a fragmentum fedettségének megfelelően állítja be a színindex 4 legkisebb helyiértékű bitjét (0000 ha nincs fedés, 1111 ha teljes a fedettség). Az, hogy ez a beállítás hogyan és milyen mértékben befolyásolja a képet, a felhasználó által használt színpalettán múlik.

A kép határának kisimítása tehát azt jelenti, hogy az alaphelyzethez képest több pixelt fest ki a rendszer, de ezeket már nem a szakasz színével, hanem a fedettségtől függően a szakasz és a háttér színének kombinációjával. Vagyis a szakasz és a háttér között fokozatos átmenet lesz, nem pedig ugrásszerű. A 8.1. ábrán egy szakasz simítás nélküli (bal oldali kép) és simított (jobb oldali kép) megjelenítését láthatjuk. Mindkét esetben jelentősen felnagyítottuk a képernyőről levett képet.



8.1. ábra. Szakasz képe kisimítás nélkül (bal oldali kép) és kisimítással (jobb oldali kép)

A fedettségi érték számítása implementációfüggő, de mindenképpen időigényes. A **glHint**() paranccsal befolyásolhatjuk a sebességet, illetve a kép minőségét, azonban ennek figyelembe vétele is implementációfüggő.

void **glHint** (GLenum *target*, GLenum *hint*);

Az OpenGL néhány tulajdonságát befolyásolhatjuk segítségével. A *target* paraméterrel azt adjuk meg, hogy mire vonatkozzon a minőségi előírás. A lehetséges értékek listáját a 8.2. táblázat tartalmazza. A *hint* paraméter lehetséges értékei: GL_FASTEST, GL_NICEST és GL_DONT_CARE. A GL_FASTEST érték a leghatékonyabb, a GL_NICEST a legjobb minőséget adó módszer használatára utasítja a rendszert, a GL_DONT_CARE értékkel pedig azt jelezzük, hogy nincs preferenciánk. Ezen utasítások figyelembe vétele teljes mértékben implementációfüggő, lehet, hogy teljesen figyelmen kívül hagyja a rendszer.

8.2. táblázat. A target paraméter értékei

| *target* | jelentése |
|---|---|
| GL_POINT_SMOOTH_HINT<br>GL_LINE_SMOOTH_HINT<br>GL_POLYGON_SMOOTH_HINT | A pont, szakasz, poligon képhatárok simításának minőségét adja meg. |
| GL_FOG_HINT | Segítségével beállíthatjuk, hogy a köd effektus számításait pixelenként (GL_NICEST), vagy csúcspontonként (GL_FASTEST) számítsa ki a rendszer. |
| GL_PERSPECTIVE_CORRECTION_HINT | A szín és textúra-koordináták interpolációjának minőségét adja meg. |

A GL_PERSPECTIVE_CORRECTION_HINT paraméter esetén azt adhatjuk meg, hogy a színeket és a textúrakoordinátákat a rendszer a kevesebb számítással járó lineáris interpolációval számítsa ki, vagy a több számítást igénylő, de a valóságnak jobban megfelelő perspektív hatást is figyelembe vevő módon számítsa ki a csúcspontbeli értékekből kiindulva. A rendszerek általában a lineáris interpolációt használják a színek esetén, miután ez vizuálisan elfogadható eredményt biztosít, jóllehet matematikailag nem pontos. Textúra esetén azonban a perspektív hatást is figyelembe vevő interpolációt kell használni a vizuálisan elfogadható eredmény érdekében.

### 8.2.1. Pont és szakasz simítása

Pont simítását a **glEnable**(GL_POINT_SMOOTH), a szakaszét a **glEnable**(GL_LINE_SMOOTH) paranccsal engedélyezni kell. A **glHint**() paranccsal a minőségre vonatkozó óhajunkat adhatjuk meg.

RGBA színmegadási mód esetén engedélyeznünk kell a színvegyítést (**glEnable**(GL_BLEND)). A kombinálási együtthatókat általában vagy *sfactor* = GL_SRC_ALPHA, *dfactor* = GL_ONE_MINUS_SRC_ALPHA, vagy *sfactor* = GL_SRC_ALPHA, *dfactor* = GL_ONE értékre állítjuk. (Az utóbbi megoldás a szakaszok metszéspontját jobban kiemeli.) Az ezek után rajzolt szakaszokat, illetve pontokat

kisimítva jeleníti meg a rendszer. Ezen lehetőség használatához tehát a színvegyítés (átlátszóság, fedettség) ismerete és megértése szükséges.

Színindex mód esetén minden a színpalettán múlik, vagyis azon, hogy sikerül-e 16 olyan színt találnunk, melyek a háttér és a rajzolási szín között folytonos átmenetet biztosítanak. Bonyolítja a helyzetet, ha több alakzat átfedi egymást, hiszen ekkor elvileg az ezek közötti folytonos átmenetet kellene biztosítani. Tehát mindenképpen meggondolandó színindex módban a kisimítást használni.

Átfedő alakzatok esetén RGBA módban sokkal jobb eredményt kaphatunk, mivel a rendszer az összes egymásra kerülő színt tudja kombinálni feltéve, hogy a mélységpuffert nem használjuk, pontosabban csak olvashatóvá tesszük a simítandó pontok, szakaszok rajzolásának idejére úgy, mint a 8.1. szakaszban láttuk.

### 8.2.2. Poligonok simítása

Ha a poligonok megjelenítési módja pont vagy szakasz (a **glPolygonMode**() parancsot GL_POINT vagy GL_LINE paraméterrel adtuk ki), akkor a poligon simított megjelenítése az előzőek szerint történik. A továbbiakban kitöltött poligonok (GL_FILL) simításáról lesz szó.

Kitöltött poligonok simítása abban az esetben a legkritikusabb, ha különböző színű, közös éllel rendelkező poligonokat rajzolunk. A legjobb eredményt a gyűjtő puffer használatával érhetjük el, amivel az egész kép kisimítható (lásd a 10.4. szakaszt), de ez nagyon számításigényes, így lassabb. A következőkben leírt eljárással is megoldható a probléma, ez viszont elég körülményes, nagy körültekintést igényel. Elvileg mind RGBA, mind színindex módban kisimíthatjuk a kitöltött poligonok határát, azonban több, egymást átfedő poligon kisimított rajzolása nem célszerű színindex módban.

Poligonok kisimított rajzolását a **glEnable**(GL_POLYGON_SMOOTH) paranccsal engedélyezni kell. Ennek hatására a poligonok határoló éleihez tartozó fragmentumok alfa értékeit megszorozza a rendszer a fedettségi indexükkel úgy, mint szakaszok simításánál. A megjelenítés minőségét a *target* = GL_POLYGON_SMOOTH_HINT paraméterrel kiadott **glHint**() paranccsal befolyásolhatjuk. Ezután csak olvashatóvá kell tenni a z-puffert (**glDepthMask**(GL_FALSE)), majd az alfa szerinti színvegyítést engedélyezni kell *sfactor* = GL_SRC_ALPHA_SATURATE, *dfactor* = GL_ONE együtthatókkal. Végül a poligonokat mélység szerint rendezni kell és hátulról előre haladva meg kell rajzolni.

## 8.3. Köd (atmoszférikus hatások)

A számítógéppel készült képekkel szemben gyakori és jogos kritika, hogy általában fémesen csillogó, éles kontúrú az objektumok képe függetlenül attól, hogy milyen közel vannak a nézőponthoz. Az előző szakaszban ismertetett képhatár-simítás mellett az OpenGL-ben további lehetőség a valószerűség fokozására a köd modellezése, aminek hatására az objektumok a nézőponttól távolodva fokozatosan elhomályosodnak, eltűnnek, beleolvadnak a köd színébe. A ködděl pára, homály, füst és szennyezett levegő is modellezhető.

A köd effektust a rendszer a transzformációk, megvilágítás és textúra-leképezés után adja hozzá a képhez. Komplex szimulációs programok esetén a köd alkalmazásával növelhetjük a sebességet, mert a sűrű ködděl borított objektumok megrajzolását elhagyhatjuk, mivel úgysem lesznek láthatók.

A ködhatást engedélyeznünk kell a **glEnable**(GL_FOG) paranccsal, majd a köd színét és sűrűségét kell megadnunk a **glFog***() paranccsal. A GL_FOG_HINT paraméterrel kiadott **glHint**() paranccsal a minőséget, illetve a sebességet befolyásolhatjuk (lásd a 8.2. szakaszt). A köd színét a rendszer a bejövő (forrás) fragmentum színével vegyíti, felhasználva a köd kombináló tényezőjét. Ezt az $f$ kombináló tényezőt az a 8.3. táblázat szerint adhatjuk meg.

8.3. táblázat. A köd kombináló tényezőjének megadása

| $f$ kiszámítása | *param* értéke a **glFog***()-ban |
|---|---|
| $f = e^{-density \cdot z}$ | GL_EXP |
| $f = e^{-(density \cdot z)^2}$ | GL_EXP2 |
| $f = \frac{end - z}{end - start}$ | GL_LINEAR |

ahol $z$ a fragmentum középpontjának a nézőponttól mért távolsága a nézőpontkoordináta-rendszerben. A *density*, *end*, *start* értékeket a **glFog***() paranccsal adhatjuk meg. Az így kiszámított $f$ együtthatót a rendszer a $[0., 1.]$ intervallumra levágja.

void **glFog**{if} (GLenum *pname*, TYPE *param*);

void **glFog**{if}**v** (GLenum *pname*, const TYPE **param*);

A köd effektus számításához szükséges értékeket adhatjuk meg vele. *pname* értéke GL_FOG_MODE, GL_FOG_DENSITY, GL_FOG_START, GL_FOG_END, GL_FOG_COLOR vagy GL_FOG_INDEX lehet.

GL_FOG_MODE esetén az $f$ együttható kiszámítási módját adhatjuk meg, ekkor a *param* értéke GL_EXP, GL_EXP2 vagy GL_LINEAR lehet. Alapértelmezés a GL_EXP.

Ha *pname* értéke GL_FOG_DENSITY, GL_FOG_START, GL_FOG_END, akkor a *param*-al a megfelelő *density*, *start*, illetve *end* értéket adjuk meg az $f$ számításához. Alapértelmezés szerint $density = 1$, $start = 0$ és $end = 1$.

RGBA módban a köd színét *pname* = GL_FOG_COLOR érték mellett a szín RGBA komponenseivel adhatjuk meg. (Ekkor csak a vektoros hívási forma megengedett.) Színindex módban pedig a *pname* = GL_FOG_INDEX érték mellett a *param*-mal átadott érték a köd színindexe.

RGBA módban a köd miatti színvegyítést a

$$\mathbf{C} = f\mathbf{C}_i + (1 - f)\mathbf{C}_k$$

összefüggés szerint számítja ki a rendszer, ahol $\mathbf{C}_i$ az új (forrás) fragmentum, $\mathbf{C}_k$ pedig a köd színe. Színindex módban pedig az

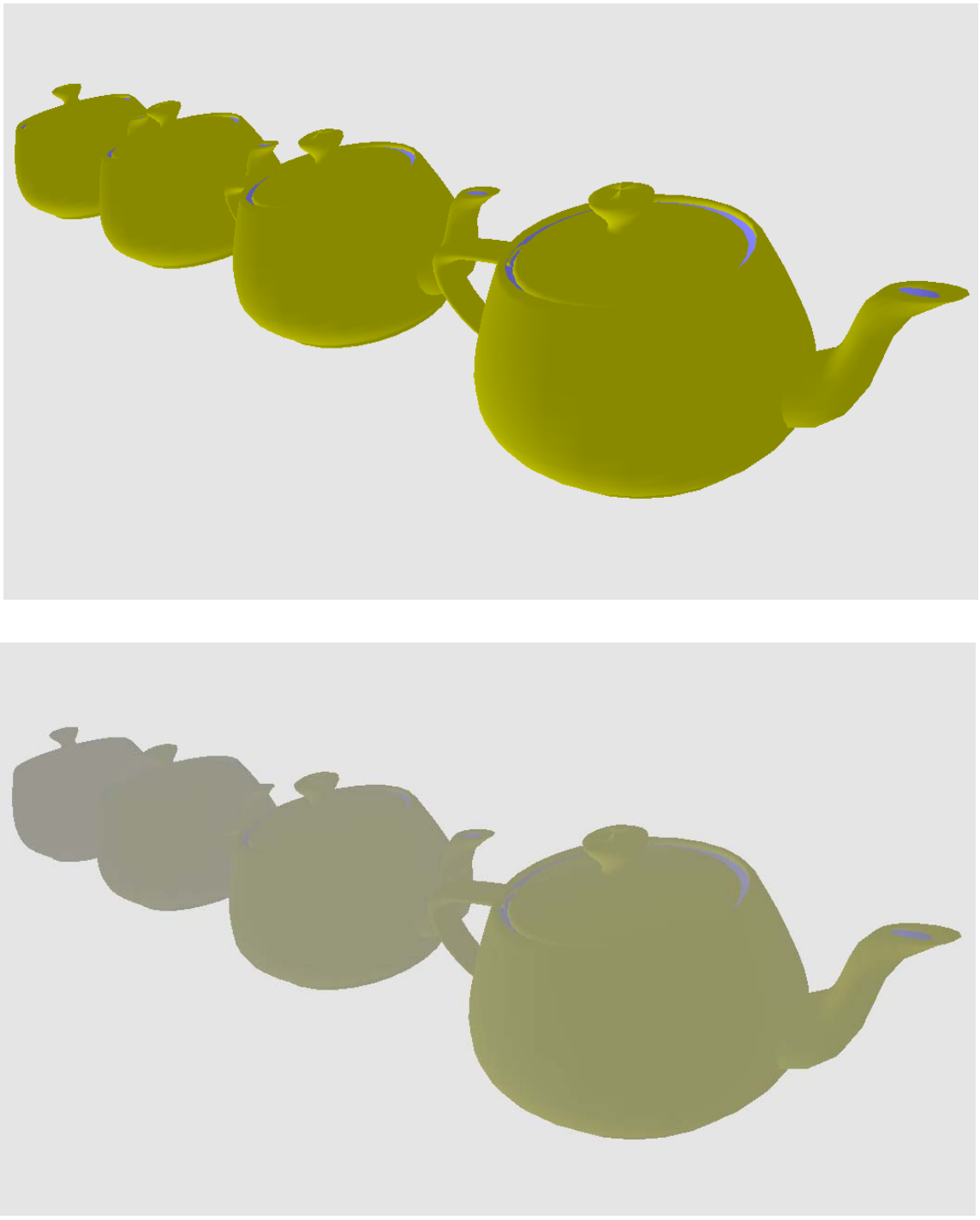
$$\mathbf{I} = \mathbf{I}_i + (1 - f)\mathbf{I}_k$$

kifejezés szerint, ahol $\mathbf{I}_i$ a forrásfragmentum, $\mathbf{I}_k$ pedig a köd színindexe. Színindex módban ügyelnünk kell a megfelelő színskála előállítására is. A skála első színének a köd nélküli objektum színével kell megegyeznie, az utolsónak pedig a ködével. Ez azt jelenti, hogy több

különböző színű objektum esetén több, az előzőeknek megfelelő színskálát kell létrehozni, ami meglehetősen nehézkessé, kissé reménytelenné teszi a ködhatás modellezését színindex módban.

A 8.2. ábra felső képén teáskannák[1] megvilágított, árnyalt képét láthatjuk. Az alsó képen pedig ugyanaz a beállítás ködhatással látható. A köd paramétereinek beállításai: GL_EXP, $density = 0.25$, $start = 1$ és $end = 5$, a köd színe pedig szürke (0.6,0.6,0.6).

[1] A számítógépi grafikában a teáskannának kitüntetett szerepe van, gyakran használják grafikai hatások demonstrálásánál. Az un. Utah teapot története az 1960-as évekre nyúlik vissza, amikor a University of Utah-on egy teáskannát digitalizáltak (ez a kanna azóta egy bostoni múzeumban van) és 306 kontrollponttal leírt 32db, egymáshoz simán kapcsolódó Bézier-felülettel modellezték.



8.2. ábra. Teáskannák köd nélkül (felső kép) és ködde l (alsó kép)

# 9. fejezet

# Raszteres objektumok rajzolása

Kétféle raszteres objektum rajzolható: a bittérkép és a kép. Mindkettő pixelek egy téglalap alakú tömbje, a különbség csak az, hogy a bittérkép minden pixelhez egyetlen bitben tárol információt, a kép pedig több bitben, de minden pixelhez ugyanannyiban (pl. RGBA értékeket). Különbség még, hogy a bittérképet egy maszkként teszi rá a rendszer a képre (pl. karakterek írása), a képpel pedig felülírja vagy kombinálja a tárolt adatokat (pl. az RGBA értékeket).

## 9.1. Bittérképek és karakterek

A bittérkép a 0 és 1 értékek tömbjének tekinthető, mely a képmező egy téglalap alakú területének rajzolási maszkjaként használható. Ahol a térképben 1 szerepel, a neki megfelelő pixelt a rendszer átírja a rajzolási színnel, vagy alfa szerint kombinálja a rajzolási színt a pixel színével attól függően, hogy milyen pixelenkénti műveletek vannak előírva. A bittérkép 0 elemeinek megfelelő pixelek változatlanok maradnak. Bittérképek használatának tipikus példája a karakterek írása.

Karakterek rajzolásához csak a legalacsonyabb szintű szolgáltatásokat biztosítja az OpenGL. Ezek a pozicionálás és bittérképek rajzolása. Karakterkészlet létrehozását és karaktersorozat rajzolását segítik a display-listáknál megismert (lásd a 7. fejezetet) parancsok. Karaktersorozat rajzolását támogató minden további függvényt a felhasználónak kell megírnia. A GLUT függvénykönyvtár is tartalmaz karakterek írását segítő függvényeket. A karaktereket leíró bittérképek bitekben mért szélességének és hosszúságának 8 többszörösének kell lenni, ami persze nem jelenti azt, hogy maguknak a karaktereknek is.

## 9.2. Kurrens raszterpozíció

A rendszer a bittérképeket és képeket mindig a kurrens raszterpozíciótól kezdődően rajzolja meg úgy, hogy a kurrens raszterpozíció lesz a bittérkép (kép) bal alsó sarka (ha másként nem rendelkezünk). A kurrens raszterpozíciót a rendszer egy globális változóban tárolja.

void gl**RasterPos**{234}{sifd} (TYPE *x*, TYPE *y*, TYPE *z*, TYPE *w*);

void **glRasterPos**{234}{sifd}**v** (conts TYPE **pos*);

A kurrens raszterpozíciót állítja be. Az *x, y, z, w* paraméterek az új pozíció koordinátái. Ha két koordinátával adjuk meg, akkor a rendszer automatikusan a $z = 0$, $w = 1$ értékekkel egészíti ki, három koordináta esetén pedig a $w = 1$ értékkel. Az így megadott raszterpozíciót pontosan úgy transzformálja a rendszer az ablakkoordináta-rendszerbe, mint a **glVertex***() paranccsal megadott csúcspontokat. Ha a transzformációk után a megadott pozíció a képmezőn kívülre esik, akkor a kurrens raszterpozíció státusza "érvénytelen" lesz.

A GL_CURRENT_RASTER_POSITION paraméterrel kiadott **glGetFloatv**() paranccsal lekérdezhetjük a kurrens raszterpozíció koordinátáit, ugyanezzel a paraméterrel kiadott **glGetBoolean**() paranccsal pedig megtudhatjuk, hogy a kurrens raszterpozíció érvényes-e.

## 9.3. Bittérkép rajzolása

void **glBitmap** (GLsizei *width*, GLsizei *hight*, GLfloat *xo*, GLfloat *yo*, GLfloat *xi*, GLfloat *yi*, const GLubyte **bitmap*);

A *bitmap* címen kezdődő bittérképet rajzolja meg a rendszer úgy, hogy a bittérkép origóját a kurrens raszterpozícióra helyezi. Az $(x_o, y_o)$ koordinátapárral gyakorlatilag a bittérkép bal alsó sarkának (origójának) eltolását adhatjuk meg pixelekben. Az eltolás komponensei lehetnek negatívok is. Ha a kurrens raszterpozíció érvénytelen, akkor nem rajzol semmit és a kurrens raszterpozíció továbbra is érvénytelen marad. A bittérkép raszterizálása után az $(x_c, y_c)$ kurrens raszterpozíciót az $(x_c + x_i, y_c + y_i)$ értékre állítja a rendszer. A *width* és *hight* paraméterek a bittérkép pixelekben mért szélességét és hosszúságát jelölik. Ezeknek nem kell feltétlenül 8 többszörösének lennie, bár a bittérkép által lefoglalt terület mindkét mérete mindig 8 többszöröse.

Az álló latin betűk esetén $y_i = 0$ és $x_i > 0$, de pl. héber karaktereknél – a héberben jobbról balra írnak – $x_i < 0$. Lehetőség van felülről lefelé, vagy alulról felfelé való írásra is. Az $(x_i, y_i)$ értékeket úgy szoktuk beállítani, hogy a karaktert követő betűközt is tartalmazza.

A **glBitmap**() paranccsal csak a képmező oldalaival párhuzamosan lehet írni, azaz nem lehet a szöveget elforgatni.

## 9.4. Képek

Az OpenGL-ben a kép nagymértékben hasonlít a bittérképhez. Alapvető különbség azonban, hogy a kép pixelenként több bitnyi adatot tartalmaz, pl. az RGBA számnégyest. A kép eredetét illetően lehet pl. egy lapleolvasóval raszterizált fénykép, a grafikus hardver videomemóriájából beolvasott kép vagy valamely program által kiszámolt és a memóriában

tárolt kép is. A képek amellett, hogy megjeleníthetők a képernyőn pl. textúraként is használhatók.

Kép hallatán általában a színpufferbe írt képre asszociálunk, azonban téglalap alakú pixeltömbök adatait – melyek nemcsak színek lehetnek – írhatjuk más pufferbe is, vagy olvashatjuk más pufferből is, pl. mélységpuffer, stencilpuffer, gyűjtőpuffer (lásd a 10. fejezetet).

### 9.4.1. Pixeladatok olvasása, írása és másolása

A képek manipulálására az OpenGL három alapvető parancsot tartalmaz:

- **glReadPixels**(), pixeltömb adatainak beolvasása a képpufferből (framebuffer) a processzor memóriájába;
- **glDrawPixels**(), pixeltömb adatainak beírása a memóriából a képpufferba;
- **glCopyPixels**(), pixeltömb másolása a képpufferen belül. Ez csak eredményét tekintve ugyanaz, mintha beolvasnánk a képpufferből, majd beírnánk annak egy másik részére, ugyanis a másolási parancs használata esetén az adat nem kerül a processzor memóriájába.

Ezen parancsok rendkívül egyszerű feladatokat hajtanak végre. A végrehajtás azonban nem mindig egyszerű, mivel nagyon sokféle adat van a képpufferben, és ezeket sokféleképpen lehet tárolni, így a fenti műveletek során általában többféle adatkonverzióra van szükség.

void **glReadPixels** (GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, const GLvoid **pixels*);

A képpuffer azon pixeltömbjének az adatait olvassa be, melynek bal alsó sarka $(x, y)$, szélessége *width*, magassága *height*. Az adatokat a *pixels* címen kezdődő memóriaterületre teszi. A *format* paraméterrel a pixelekről beolvasandó adatelemeket adjuk meg (lásd a 9.1. táblázatot), a *type* paraméterrel pedig az adattárolás típusát (lásd a 9.2. táblázatot).

void **glDrawPixels** (GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, const GLvoid **pixels*);

A *pixels* címen tárolt *width* szélességű, *height* magasságú pixeltömböt rajzolja meg úgy, hogy a pixeltömb bal alsó sarka a kurrens raszterpozícióba kerül. Ha a kurrens raszterpozíció érvénytelen, akkor nem rajzol semmit, és a kurrens raszterpozíció továbbra is érvénytelen marad. A *format* és *type* paraméterek értelmezése a **glReadPixels**() parancsnál írtakkal megegyezik.

Ha a pixeladat elemei folytonos adatot reprezentálnak, mint pl. az RGB értékek, akkor a rendszer az adattípusnak megfelelő intervallumra képezi le az adatokat, tehát adatvesztés is előállhat. Ha az adatelem egy index, pl. szín- vagy stencilindex, és a típus nem GL_FLOAT, akkor a rendszer az értéket az adott típus bitjeivel maszkolja.

9.1. táblázat. A pixelekről beolvasható adatelemek

| *format* | a beolvasott adat |
|---|---|
| GL_COLOR_INDEX | színindex |
| GL_RGB | vörös, zöld és kék színkomponensek |
| GL_RGBA | vörös, zöld, kék és alfa komponensek |
| GL_RED | vörös színkomponens |
| GL_GREEN | zöld színkomponens |
| GL_BLUE | kék színkomponens |
| GL_ALPHA | alfa komponens |
| GL_LUMINANCE | világosság |
| GL_LUMINANCE_ALPHA | világosság és alfa komponens |
| GL_STENCIL_INDEX | stencilindex |
| GL_DEPTH_COMPONENT | mélység |

9.2. táblázat. Az adattárolás típusa

| *type* | az adattípus leírása |
|---|---|
| GL_UNSIGNED_BYTE | 8 bites előjel nélküli egész |
| GL_BYTE | 8 bites előjeles egész |
| GL_BITMAP | 1 bit, 8 bites előjel nélküli egészekben |
| GL_UNSIGNED_SHORT | 16 bites előjel nélküli egész |
| GL_SHORT | 16 bites előjeles egész |
| GL_UNSIGNED_INT | 32 bites előjel nélküli egész |
| GL_INT | 32 bites egész |
| GL_FLOAT | egyszeres pontosságú lebegőpontos |

void **glCopyPixels** (GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*, GLenum *type*);

Egy pixeltömb adatait másolja ugyanazon a pufferen belül a kurrens raszterpozícióba. A másolandó tömb bal alsó sarkának koordinátái $(x, y)$, szélessége *width*, magassága *height*. A *type* paraméter értéke GL_COLOR, GL_STENCIL és GL_DEPTH lehet. A másolás során a **glPixelTransfer***(), **glPixelMap***() és **glPixelZoom**() parancsokkal beállított manipulációk hatása érvényesül. *type* = GL_COLOR esetén a színmegadási módtól függően vagy az RGBA komponensek, vagy a színindexek kerülnek másolásra; *type* = GL_DEPTH esetén a mélységértékek, *type* = GL_STENCIL esetén pedig a stencilindexek.

### 9.4.2. Képek kicsinyítése, nagyítása

void **glPixelZoom** (GLfloat *xfactor*, GLfloat *yfactor*);

Segítségével a pixeltömbök rajzolása és másolása során végrehajtandó skálázási faktorokat írhatjuk elő. *xfactor* és *yfactor* alapértelmezése 1. Raszterizáláskor a kép minden pixelét egy-egy $xfactor \times yfactor$ méretű téglalapként kezeli a rendszer, és a képernyő minden olyan pixele számára létrehoz egy fragmentumot, mely középpontja ebbe a téglalapba, vagy ennek a téglalapnak az alsó vagy bal oldalára esik. Negatív skálázási értékek a megfelelő tengelyre vonatkozó tükrözéseket is eredményeznek.

## 9.5. Pixelek tárolása, transzformálása, leképezése

A memóriában tárolt kép pixelenként$1 - 4$ adatelemet tartalmazhat. Pl. az adat állhat pusztán a színindexből, a világosságból (luminance, ami általában az RGB értékek átlaga) vagy magukból az RGBA értékekből. A pixeladatok formátuma határozza meg a tárolt adatok számát és sorrendjét.

Néhány adatelem egész szám (pl. a szín- vagy stencilindex), mások lebegőpontos értékek (pl. az RGBA komponensek vagy a mélység), melyek általában a $[0., 1.]$ intervallumon változnak. A lebegőpontos számokat a megfelelő pufferben a grafikus hardvertől függően általában kisebb helyen – színkomponens esetén 8 biten – tárolja a rendszer, mint amennyi szükséges lenne (32 bit). Ezért pazarlás lenne pl. a színpufferből beolvasott színkomponenseket 32 biten tárolni, annál is inkább, mivel egy kép pixeleinek száma könnyen meghaladhatja az 1 milliót is.

Az adatok többféleképpen tárolhatók a memóriában, 8 – 32 bit, egész vagy lebegőpontos szám. Az OpenGL explicite definiálja a különböző formátumok közötti konverziókat.

## 9.6. A pixeladatok tárolásának szabályozása

A tárolási mód a **glPixelStore**() paranccsal szabályozható.

void **glPixelStore**{if} (GLenum *pname*, TYPE *param*);

A **glDrawPixels***(), **glReadPixels***(), **glBitmap**(), **glPolygonStipple**(), **glTexImage1D**(), **glTexImage2D**() és **glGetTexImage**() parancsok működését befolyásoló pixeladat tárolási módot állítja be. A a 9.3. táblázat a pname paraméter lehetséges értékeit, azok adattípusát, kezdeti értékét és a megadható értékek intervallumát tartalmazza. GL_UNPACK* paraméter esetén azt szabályozza, hogy a **glDrawPixels***(), **glBitmap**(), **glPolygonStipple**(), **glTexImage1D**(), **glTexImage2D**() parancsok hogyan csomagolják ki az adatokat a memóriából; GL_PACK* esetén pedig azt, hogy a **glReadPixels***() és **glGetTexImage**() parancsok hogyan tömörítik (csomagolják össze) az adatokat a memóriába.

9.3. táblázat. A pixeladatok tárolásának paraméterei

| *pname* | adattípus | kezdeti érték | felvehető értékek |
|---|---|---|---|
| GL_UNPACK_SWAP_BYTES<br>GL_PACK_SWAP_BYTES | GLboolean | GL_FALSE | GL_TRUE<br>GL_FALSE |
| GL_UNPACK_LSB_FIRST<br>GL_PACK_LSB_FIRST | GLboolean | GL_FALSE | GL_TRUE<br>GL_FALSE |
| GL_UNPACK_ROW_LENGTH<br>GL_PACK_ROW_LENGTH | GLint | 0 | nemnegatív egész |
| GL_UNPACK_SKIP_ROWS<br>GL_PACK_SKIP_ROWS | GLint | 0 | nemnegatív egész |
| GL_UNPACK_SKIP_PIXELS<br>GL_PACK_SKIP_PIXELS | GLint | 0 | nemnegatív egész |
| GL_UNPACK_ALIGNMENT<br>GL_PACK_ALIGNMENT | GLint | 4 | 2 hatványai<br>$(1, 2, 4, \ldots)$ |

## 9.7. Műveletek pixelek mozgatása során

A pixeleken műveleteket hajthatunk végre, miközben a képpufferből (vagy a képpufferba) mozgatjuk. Ezen műveletek egyrészt a **glPixelTransfer***(), másrészt a **glPixelMap***() paranccsal írhatók le.

void **glPixelTransfer**{if} (GLenum *pname*, TYPE *param*);

A **glDrawPixels***(), **glReadPixels***(), **glCopyPixels***(), **glTexImage1D**(), **glTexImage2D**() és **glGetTexImage**() parancsok működését befolyásoló pixelátviteli módot állítja be. A a 9.3. táblázat a *pname* paraméter lehetséges értékei, azok adattípusát, kezdeti értékét és a megadható értékek intervallumát tartalmazza.

## 9.8. Transzformálás táblázat segítségével

A képernyő memóriájába való beírás előtt a színkomponenseket, a szín- és stencilindexeket egy táblázat segítségével is módosíthatjuk.

void **glPixelMap**{uiusf}**v** (GLenum *map*, GLint *mapsize*, const TYPE **values*);

Betölti a *values* címen kezdődő, *mapsize* méretű *map* típusú táblázatot. A a 9.5. táblázat tartalmazza a *map* paraméter lehetséges értékeit. A méret alapértelmezése 1, az értékeké pedig 0. A méretnek mindig 2 hatványának kell lennie.

9.4. táblázat. A **glPixelTransfer**() parancs paraméterei

| *pname* | adattípus | kezdeti érték | felvehető érték |
|---|---|---|---|
| GL_MAP_COLOR<br>GL_MAP_STENCIL | GLboolean | GL_FALSE | GL_TRUE<br>GL_FALSE |
| GL_INDEX_SHIFT | GLint | 0 | tetszőleges egész |
| GL_INDEX_OFFSET | GLint | 0 | tetszőleges egész |
| GL_RED_SCALE | GLfloat | 1. | tetszőleges valós |
| GL_GREEN_SCALE | GLfloat | 1. | tetszőleges valós |
| GL_BLUE_SCALE | GLfloat | 1. | tetszőleges valós |
| GL_ALPHA_SCALE | GLfloat | 1. | tetszőleges valós |
| GL_DEPTH_SCALE | GLfloat | 1. | tetszőleges valós |
| GL_RED_BIAS | GLfloat | 0. | tetszőleges valós |
| GL_GREEN_BIAS | GLfloat | 0. | tetszőleges valós |
| GL_BLUE_BIAS | GLfloat | 0. | tetszőleges valós |
| GL_ALPHA_BIAS | GLfloat | 0. | tetszőleges valós |
| GL_DEPTH_BIAS | GLfloat | 0. | tetszőleges valós |

9.5. táblázat. A **glPixelMap**() paranccsal betölthető táblázatok típusa

| *map* | mire vonatkozik | milyen értéket tartalmaz |
|---|---|---|
| GL_PIXEL_MAP_I_TO_I | színindex | színindex |
| GL_PIXEL_MAP_S_TO_S | stencilindex | stencilindex |
| GL_PIXEL_MAP_I_TO_R | színindex | R |
| GL_PIXEL_MAP_I_TO_G | színindex | G |
| GL_PIXEL_MAP_I_TO_B | színindex | B |
| GL_PIXEL_MAP_I_TO_A | színindex | A |
| GL_PIXEL_MAP_R_TO_R | R | R |
| GL_PIXEL_MAP_G_TO_G | G | G |
| GL_PIXEL_MAP_B_TO_B | B | B |
| GL_PIXEL_MAP_A_TO_A | A | A |

# 10. fejezet

# Pufferek

Raszteres grafikus display-n a képet pixelekből (képpontokból), apró téglalap alakú foltokból rakjuk össze. Ezen téglalapok oldalai az ablakkoordináta-rendszer tengelyeivel párhuzamosak és a bal alsó sarok koordinátáival azonosítjuk őket. Egy-egy kép előállításához és megjelenítéséhez a pixelekhez tárolni kell színüket, a láthatósági vizsgálatokhoz a mélységüket (az ablakkoordináta-rendszerbeli z koordinátát). Egyéb hatások elérése érdekében további adatok pixelenkénti tárolása is szükséges, pl. stencilindex.

Azokat a tárterületeket, melyekben minden pixelhez ugyanannyi adatot tárolunk puffernek nevezzük. Az OpenGL-ben több puffer van, így pl. a színpuffer, mélységpuffer. Egy puffernek lehetnek alpufferei, mint pl. RGBA színmegadási mód esetén a színpuffernek R, G, B, A vagy R, G, B alpufferei. Egy képhez tartozó összes puffert együttesen képpuffernek nevezzük.

Az OpenGL a következő puffereket használja:

- színpufferek: bal első, bal hátsó, jobb első, jobb hátsó és további kiegészítő színpufferek;
- mélységpuffer;
- stencilpuffer;
- gyűjtőpuffer.

A pufferek száma és a bit/pixel mérete implementációfüggő, azonban biztos, hogy minden implementációban van legalább egy színpuffer RGBA színmegadási módhoz és ehhez kapcsolódóan stencil-, mélység- és gyűjtőpuffer; továbbá egy színpuffer színindex módhoz stencil- és mélységpufferrel. A **glGetIntegerv**() paranccsal lekérdezhetjük, hogy implementációnk az egyes pufferekben pixelenként hány biten tárolja az információt. A lekérdezhető pufferek azonosítóját a a 10.1. táblázat tartalmazza.

## 10.1. Színpufferek

A színpuffer az, amibe rajzolunk. Ezek vagy színindexeket vagy RGB színkomponenseket és alfa értékeket tartalmaznak. Azok az OpenGL implementációk, melyek támogatják a

10.1. táblázat. A pufferek azonosítói

| azonosító | jelentése |
|---|---|
| GL_RED_BITS<br>GL_GREEN_BITS<br>GL_BLUE_BITS<br>GL_ALPHA_BITS | a színpufferben az R, G, B, A komponensek tárolására használt bitek száma pixelenként |
| GL_DEPTH_BITS | a mélységpuffer pixelenkénti mérete |
| GL_STENCIL_BITS | a stencilpuffer pixelenkénti mérete |
| GL_ACCUM_RED_BITS<br>GL_ACCUM_GREEN_BITS<br>GL_ACCUM_BLUE_BITS<br>GL_ACCUM_ALPHA_BITS | a gyűjtőpufferben az R, G, B, A komponensek tárolására használt bitek száma pixelenként |

sztereoszkópikus (bicentrális) ábrázolást, bal és jobb oldali színpuffereket is tartalmaznak. Ha egy implementáció nem támogatja a sztereoszkópikus ábrázolást, akkor csak bal oldali színpuffere van. Ha az animációt támogatja az implementáció, akkor van első és hátsó színpuffere is, egyébként csak első. Minden OpenGL implementációban lenni kell egy bal első színpuffernek.

Az implementációk legfeljebb 4 további, közvetlenül meg nem jeleníthető színpuffert tartalmazhatnak. Az OpenGL nem rendelkezik ezek használatáról, vagyis tetszésünk szerint hasznosíthatjuk őket, pl. ismétlődően használt képek tárolására, így ugyanis nem kell mindig újrarajzolni a képet, elég pusztán átmásolni egyik pufferből a másikba.

A **glGetBoolean**(GL_STEREO) paranccsal lekérdezhetjük, hogy implementációnk támogatja-e a sztereoszkópikus ábrázolást (van-e jobb oldali színpuffere is); a **glGetBoolean**(GL_DOUBLE_BUFFER) paranccsal pedig azt, hogy az animációt támogatja-e (van-e hátsó színpuffere). A GL_AUX_BUFFERS paraméterrel kiadott **glGetIntegerv**() paranccsal az opcionális színpufferek számát kaphatjuk meg.

## 10.2. Mélységpuffer

Az OpenGL a mélységpuffer (z-puffer) algoritmust használja a láthatóság megállapításához, ezért minden pixelhez mélységet (az ablakkoordináta-rendszerbeli $z$ értéket) is tárol. Ha másként nem rendelkezünk, a rendszer felülírja a hozzá tartozó pixelt az új fragmentummal, amennyiben annak az ablakkoordináta-rendszerbeli $z$ értéke kisebb, mivel ez a $z$ érték (a mélység) a nézőponttól mért távolságot fejezi ki.

## 10.3. Stencilpuffer

A stencilpuffert arra használjuk, hogy a rajzolást a képernyő bizonyos részeire korlátozzuk. Ahhoz hasonlít, mint amikor egy kartonlapba feliratot vágunk, és a lapot egy felületre helyezve lefestjük, aminek során a felületre csak a felirat kerül. A fénymásolók elterjedése előtt ezt a technikát használták kis példányszámú sokszorosításra. Más hasonlattal

élve, a stencilpuffer használatával tetszőleges alakú ablakon át nézhetjük a világot. Látni fogjuk, hogy e mellett még más "trükkökhöz", speciális hatásokhoz is jól használható a stencilpuffer.

## 10.4. Gyűjtőpuffer

A gyűjtőpuffer RGBA színkomponenseket tartalmaz ugyanúgy, mint a színpuffer, ha a színmegadási mód RGBA. Színindex módban a gyűjtőpuffer tartalma definiálatlan. A gyűjtőpuffert általában arra használjuk, hogy több képet összegezve állítsunk elő egy végső képet. Ezzel a módszerrel vihető végbe pl. a teljes kép kisimítása (antialiasing). A gyűjtőpufferbe nem tudunk közvetlenül rajzolni, az akkumulálás mindig téglalap alakú pixeltömbökre vonatkozik és az adatforgalom közte és a színpuffer között van.

A gyűjtőpuffer használatával hasonló hatás érhető el, mint amikor a fényképész ugyanarra a filmkockára többször exponál. Ekkor ugyanis a lefényképezett térben esetlegesen mozgó alakzat több példányban, elmosódottan fog megjelenni. Ezen kívül más hatások is elérhetők, így pl. a teljes kép kisimítása, a mélységélesség szimulálása is.

Az OpenGL parancsokkal nem lehet közvetlenül csak a gyűjtőpufferbe írni. A színpufferek valamelyikébe kell rajzolni és közben a gyűjtőpufferben is tárolni kell a képet, majd az összegyűjtött – a gyűjtőpufferben megkomponált – kép visszamásolható a színpufferbe, hogy látható legyen. A kerekítési hibák csökkentése érdekében a gyűjtőpuffer bit/pixel értéke általában nagyobb, mint a színpuffereké.

void **glAccum** (GLenum *op*, GLfloat *value*);

A gyűjtőpuffer működését szabályozza. Az *op* paraméterrel a műveletet választhatjuk ki, a *value* paraméterrel pedig a művelethez használt értéket. Az *op* lehetséges értékei és hatása:

- GL_ACCUM: a **glReadBuffer**() paranccsal olvasásra kiválasztott puffer pixeleinek R, G, B, A értékeit kiolvassa, megszorozza őket a value értékkel és hozzáadja a gyűjtőpufferhez.
- GL_LOAD: a **glReadBuffer**() paranccsal olvasásra kiválasztott puffer pixeleinek R, G, B, A értékeit kiolvassa, megszorozza őket a value értékkel és felülírja velük a gyűjtőpuffer megfelelő elemeit.
- GL_RETURN: a gyűjtőpufferből kiolvasott értékeket megszorozza a value értékkel, majd a kapott eredményt az írható színpufferekbe beírja.
- GL_ADD: a gyűjtőpufferbeli értékekhez hozzáadja a value értéket, majd az eredményt visszaírja a gyűjtőpufferbe.
- GL_MULT: a gyűjtőpufferbeli értéket megszorozza a value értékkel, az eredményt a $[-1., 1.]$ intervallumra levágja, majd visszaírja a gyűjtőpufferbe.

### 10.4.1. Teljes kép kisimítása

A teljes kép kisimításához előbb töröljük a gyűjtőpuffert és engedélyezzük az első színpuffer írását és olvasását. Ez után rajzoljuk meg $n$-szer a képet picit különböző pozícióból (a vetítési leképezést picit módosítsuk), mintha csak a fényképezőgépet tartó kéz remegne. A kamera remegtetésének olyan kicsinek kell lenni, hogy a képmezőn mérve az elmozdulás 1 pixelnél kisebb legyen. Amikor ugyanarról az objektumról több képet kell ilyen módon létrehozni, fontos kérdés, hogy miként válasszuk meg az elmozdulásokat. Egyáltalán nem biztos, hogy az a jó, ha a pixelt egyenletesen felosztjuk mindkét irányban és a rácspontokba toljuk el. A helyes felosztás módja csak tapasztalati úton állapítható meg.
A gyűjtőpufferbe minden rajzolásnál a

```
glAccum(GL_ACCUM,1./n);
```

beállítás mellett kerüljön a kép, végül a

```
glAccum(GL_RETURN,1.);
```

parancsot adjuk ki, amivel az összegyűjtött kép látható lesz. Az elmozdulás, a kamera remegése egy pixelnél kisebb legyen! Az eljárás egy kicsit gyorsabb, ha nem töröljük a gyűjtőpuffert, hanem az első kép megrajzolása előtt a

```
glAccum(GL_LOAD,1./n);
```

parancsot adjuk ki. Ha nem akarjuk, hogy az $n$ darab közbülső fázis is látható legyen, akkor abba a színpufferbe rajzoljunk, amelyik nem látható – ez dupla színpuffer használatakor lehetséges – és csak a

```
glAccum(GL_RETURN,1.);
```

parancs kiadása előtt váltsunk a látható színpufferre.

Másik lehetőség egy interaktív környezet létrehozása, amikor a felhasználó minden kép hozzáadása után dönthet arról, hogy tovább javítsa-e a képet. Ekkor a rajzolást végző ciklusban minden kirajzolás után a

```
glAccum(GL_RETURN,n/i);
```

parancsot kell kiadni, ahol i a ciklusváltozó.

### 10.4.2. Bemozdulásos életlenség (motion blur)

Feltételezzük, hogy az ábrázolt térrészben vannak rögzített és mozgó objektumok, és a mozgó alakzatok különböző helyzeteit ugyanazon a képen akarjuk ábrázolni úgy, hogy az időben visszafelé haladva a képek egyre elmosódottabbak legyenek. A megoldás a kép kisimításához hasonlít. A különbség az, hogy most nem a kamerát kell mozgatni, hanem az alakzatot, és a

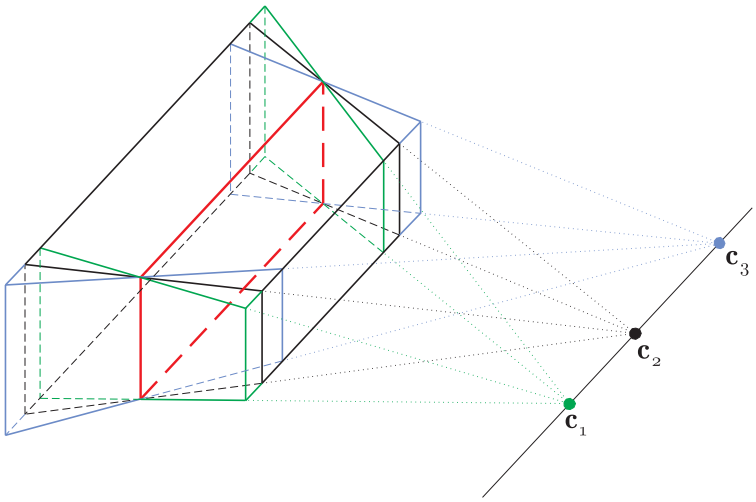
```
glAccum(GL_MULT,decayf); glAccum(GL_ACCUM,1.-decayf);
```

parancsokatt kell kiadni, ahol $decayf \in [0.9, 1.]$, ha ez a tényező kicsi a mozgás gyorsabbnak látszik. Végül a képet – a háttér, a mozgó alakzat pillanatnyi pozíciója és az előző helyzetek által leírt elmosódó csóva – a `glAccum(GL_RETURN,1.)` paranccsal a látható színpufferbe írhatjuk.

### 10.4.3. Mélységélesség

Egy fényképezőgéppel készített képen vannak objektumok, melyek élesen látszanak, más részek kissé (vagy nagyon) homályosak. Normál körülmények között az OpenGL-el

készített képek minden része ugyanolyan éles. A gyűjtőpufferrel azonban elérhető az, hogy egy adott síktól távolodva egyre homályosabb, elmosódottabb legyen a kép. Ez nem a fényképezőgép működésének a szimulálása, de a végeredmény eléggé hasonlít a fényképezőgéppel készített képhez.




10.1. ábra. A vetítési középpontok elhelyezése a mélységélesség modellezéséhez

Ennek érdekében többször kell megrajzolni a képet a perspektív leképezés olyan változtatása mellett, hogy az ábrázolandó térrészt megadó csonka gúláknak legyen egy közös téglalapja, mely a csonka gúla alaplapjával párhuzamos síkban van (lásd a 10.1. ábrát). Ennek a módosításnak természetesen nagyon csekélynek kell lennie. A képet a szokásos módon átlagolni kell a gyűjtőpufferben.

## 10.5. Pufferek törlése

A pufferek törlése nagyon időigényes feladat, egyszerűbb rajzok esetén a törlés hosszabb ideig tarthat, mint maga a rajzolás. Ha nemcsak a színpuffert, hanem pl. a mélységpuffert is törölni kell, ez arányosan növeli a törléshez szükséges időt. A törlésre fordított idő csökkentése érdekében legtöbb grafikus hardver egyidejűleg több puffert is tud törölni, amivel jelentős megtakarítás érhető el.

A törléshez előbb be kell állítani a törlési értéket, azt az értéket, amivel felül akarjuk írni a puffer elemeit, majd végre kell hajtani magát a törlést.

void **glClearColor** (GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*);

A törlési szín megadása RGBA színmegadási mód esetén. A megadott értékeket a rendszer a $[0., 1.]$ intervallumra levágja. Alapértelmezés: $0., 0., 0., 0.$.

void **glClearIndex** (GLfloat *index*);

A törlési szín megadása színindex mód esetén. Alapértelmezés: 0.

void **glClearDepth** (GLclampd *depth*);

A törlési mélység megadása. A megadott értékeket a rendszer a $[0., 1.]$ intervallumra levágja. Alapértelmezés: 1..

void **glClearStencil** (GLint *s*);

A stencilpuffer törlési értékének megadása. Alapértelmezés: 0.

void **glClearAccum** (GLclampf *red*, GLclampf *green*, GLclampf *blue*, GLclampf *alpha*);

A gyűjtőpuffer törlési színének megadása. A megadott értékeket a rendszer a $[0., 1.]$ intervallumra levágja. Alapértelmezés: $0., 0., 0., 0.$.

void **glClear** (GLbitfield *mask*);

A *mask* paraméterrel megadott puffereket törli. *mask* értéke a GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_STENCIL_BUFFER_BIT, GL_ACCUM_BUFFER_BIT konstansok valamelyike, vagy ezeknek a logikai vagy (OR) művelettel összekapcsolt tetszőleges kombinációja lehet. GL_COLOR_BUFFER_BIT esetén mindig a megfelelő színpuffer törlődik, azaz vagy a színindex-puffer, vagy az RGBA puffer; továbbá minden írható puffert töröl a rendszer.

## 10.6. Az írandó és olvasandó pufferek kiválasztása

A rajzolás eredménye az általunk kiválasztott színpufferbe, vagy színpufferekbe kerülhet, tehát a bal első, jobb első, bal hátsó, jobb hátsó vagy a kiegészítő színpufferekbe, illetve ezek tetszőleges kombinációjába, azaz egyszerre több színpufferbe is rajzolhatunk. Animáció esetén pl. a közös hátteret egyszerre megrajzolhatjuk az első és hátsó színpufferekbe, majd a mozgó alakzat különböző helyzeteit felváltva az első, illetve a hátsó pufferbe, gondoskodva a két puffer megfelelő cseréjéről.

A **glDrawBuffer**() paranccsal állíthatjuk be a kurrens puffert.

void **glDrawBuffer** (GLenum *mode*);

Az írandó vagy törlendő színpuffer kiválasztása. A mode paraméter az alábbi értékeket veheti fel:

GL_FRONT, GL_BACK, GL_RIGHT, GL_LEFT, GL_FRONT_RIGHT, GL_FRONT_LEFT, GL_BACK_RIGHT, GL_BACK_LEFT, GL_FRONT_AND_BACK, GL_AUX*i*, GL_NONE

A GL_LEFT és GL_RIGHT az első és a hátsó pufferekre is vonatkozik, hasonlóképpen a GL_FRONT és GL_BACK a bal és a jobb oldalira is. A GL_AUXi pedig az *i*-edik kiegészítő színpuffert azonosítja.

Kijelölhetünk olyan puffert is amelyik nem létezik mindaddig, míg a kijelölt pufferek között van létező. Ha a kiválasztott pufferek egyike sem létezik, hibára jutunk.

Egy képsík használata esetén (single-buffered mode) az alapértelmezés GL_FRONT, két képsík (double-buffered mode) esetén pedig GL_BACK. (A két mód közötti választás hardverfüggő, tehát a GLX vagy GLUT könyvtár megfelelő függvényével lehet megtenni.)

void **glReadBuffer** (GLenum *mode*);

Annak a színpuffernek a kiválasztása, amelyből a **glReadPixels**(), **glCopyPixels**(), **glCopyTexImage***(), **glCopyTexSubImage***() és **glCopyConvolutionFilter***() függvényekkel pixeladatokat olvashatunk be. A korábbi **glReadBuffer**() hívásának hatását érvényteleníti. Csak létező pufferekből való olvasás engedélyezhető. Egy képsík használata esetén (single-buffered mode) az alapértelmezés GL_FRONT, két képsík (double-buffered mode) esetén pedig GL_BACK. (A két mód közötti választás hardverfüggő, tehát a GLX vagy GLUT könyvtár megfelelő függvényével lehet megtenni.)

## 10.7. Pufferek maszkolása

Mielőtt az OpenGL beírna egy adatot a kiválasztott pufferbe (szín, mélység, stencil) maszkolja az adatokat, vagyis bitenkénti logikai és (AND) műveletet hajt végre a maszk megfelelő elemén és a beírandó adaton. A maszkolási műveleteket az alábbi parancsokkal adhatjuk meg:

void **glIndexMask** (GLuint *mask*);

Csak színindex színmegadási módban van hatása, segítségével a színindex maszkolható. Ahol a maszkban 1 szerepel, a színindex megfelelő bitjét beírja a rendszer a színpufferbe, ahol 0 szerepel azt nem. Alapértelmezés szerint a színindex maszkjának minden bitje 1.

void **glColorMask** (GLboolean *red*, GLboolean *green*, GLboolean *blue*, GLboolean *alpha*);

Csak RGBA színmegadási mód esetén van hatása, segítségével azt szabályozhatjuk, hogy a rendszer az R, G, B, A értékek közül melyiket írja be a színpufferbe. GL_TRUE esetén beírja, GL_FALSE esetén nem írja be. Alapértelmezés szerint mind a négy komponenst beírja.

void **glDepthMask** (GLboolean *flag*);

Ha a *flag* értéke GL_TRUE, a mélységpuffer írható, GL_FALSE esetén csak olvasható. Alapértelmezés: GL_TRUE.

void **glStencilMask** (GLuint *mask*);

Ahol a *mask*ban 1 szerepel, a színindex megfelelő bitjét beírja a rendszer a színpufferbe, ahol 0 szerepel azt nem. Alapértelmezés szerint a *mask* minden bitje 1.

Színindex módban számos trükk elérhető maszkolással.

A mélységpuffer maszkolásának alkalmazására már az átlátszó alakzatok ábrázolásánál láthattunk példát (lásd a 8.1. szakaszt). További alkalmazás lehet pl. ha ugyanabban a környezetben mozog egy alak, akkor a hátteret csak egyszer rajzoljuk meg, majd a mélységpuffert csak olvashatóvá tesszük, és az előtérben mozgó alakot így rajzoljuk meg képkockánként. Ez mindaddig jól működik, míg az előtérben lévő alak ábrázolásához nem szükséges a mélységpuffer írhatósága.

A stencilpuffer maszkolásával több 1 bit/pixel méretű stencilsík használható. Ezek segítségével hajtható végre pl. a zárt felületekből a vágósíkok által lemetszett részek befedése. A **glStencilMask**() paranccsal megadott maszk azt szabályozza, hogy melyik bitsík írható. Az itt megadott maszk nem azonos a **glStencilFunc**() parancs harmadik paramétereként megadandó maszkkal, azzal ugyanis azt szabályozzuk, hogy a stencil függvény mely bitsíkokat vegye figyelembe.

# 11. fejezet

# A fragmentumokon végrehajtott vizsgálatok és műveletek

Amikor geometriai alakzatokat, szöveget vagy raszteres képet rajzolunk az OpenGL segítségével, a megjelenítendő alakzatokon sok műveletet hajt végre a rendszer: modell-nézőpont transzformáció, megvilágítási számítások, vetítési transzformáció, képmező-transzformáció, fragmentálás. A fragmentumok is sok vizsgálaton mennek át míg végül valamilyen formában esetleg megjelennek a képernyőn. A rendszer pl. figyelmen kívül hagyja azokat a fragmentumokat, amelyek a képmező adott területén kívül esnek, vagy távolabb vannak a nézőponttól, mint a megfelelő pixel aktuális mélysége. A vizsgálatok után megmaradó fragmentumok színét a megfelelő pixelek színével kombinálhatja a rendszer az alfa komponensek szerint.

Ebben a fejezetben a fragmentumokon végrehajtható vizsgálatokat és műveleteket foglaljuk össze. Ezek teljes listája a végrehajtás sorrendjében a következő:

1. kivágási vizsgálat,
2. alfa-vizsgálat,
3. stencilvizsgálat,
4. mélységvizsgálat,
5. színkombinálás alfa szerint,
6. dithering,
7. logikai műveletek.

Ha egy fragmentum a fentiek közül valamelyik vizsgálaton fennakad, akkor a többi vizsgálatot nem hajtja végre rajta a rendszer.

## 11.1. Kivágási vizsgálat

A **glScissor**() paranccsal a rajzolást leszűkíthetjük az ablaknak, egy az ablak oldalaival párhuzamos oldalú téglalap alakú területére. Egy fragmentum akkor jut túl a kivágási vizsgálaton, ha erre a téglalapra esik a képe.

void **glScissor** (GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

A kivágási téglalap méreteit adja meg az ablakkoordináta-rendszerben. A téglalap bal alsó sarkának koordinátái $(x, y)$, szélessége *width*, magassága *height*. Alapértelmezés szerint a kivágási téglalap megegyezik az ablakkal.

A kivágás a stencilvizsgálat speciális esetének tekinthető. Létét az indokolja, hogy ezt a kivágást könnyű a hardverben implementálni, ezért nagyon gyors lehet, ellentétben a tetszőleges alakú kivágási területet megengedő stencilvizsgálattal, melyet szoftverből valósítanak meg.

A kivágási vizsgálatot a **glEnable**(GL_SCISSOR_TEST) paranccsal lehet engedélyezni, a **glDisable**(GL_SCISSOR_TEST) paranccsal letiltani, a **glIsEnabled**(GL_SCISSOR_TEST) paranccsal pedig lekérdezhető, hogy a kivágás engedélyezett-e. A GL_SCISSOR_BOX paraméterrel kiadott **glGetIntegerv**() paranccsal pedig a kivágási téglalap adatait kapjuk meg.

## 11.2. Alfa-vizsgálat

RGBA színmegadási mód esetén lehetőség van fragmentumok elhagyására alfa komponensük alapján. Ha ez a vizsgálat engedélyezett, a rendszer az új fragmentum alfa értékét összehasonlítja egy referenciaértékkel, és az összehasonlítás eredményétől függően eldobja a fragmentumot, vagy továbbengedi a megjelenítési műveletsoron. A referenciaértéket és az összehasonlító függvényt a **glAlphaFunc**() paranccsal adhatjuk meg.

void **glAlphaFunc** (GLenum *func*, GLclampf *ref*);

A referenciaértéket és az összehasonlító függvényt állítja be. A megadott *ref* érétket levágja a $[0., 1.]$ intervallumra. A 11.1. táblázat tartalmazza a *func* paraméter lehetséges értékeit, *alfa* az új fragmentum alfa komponensét jelöli. Az alfa szerinti vizsgálatot a **glEnable**(GL_ALPHA_TEST) paranccsal lehet engedélyezni, a **glDisable**(GL_ALPHA_TEST) paranccsal letiltani, a **glIsEnabled**(GL_ALPHA_TEST) paranccsal pedig lekérdezhető, hogy engedélyezett-e. A **glGetIntegerv**() parancsot a GL_ALPHA_TEST_FUNC paraméterrel kiadva a kurrens összehasonlító függvényt, a GL_ALPHA_TEST_REF paraméterrel kiadva pedig a referenciaértéket kérdezhetjük le. Alapértelmezés szerint *func* = GL_AWAYS, *ref* = *0* és nem engedélyezett.

Az alfa-vizsgálattal átlátszósági algoritmust implementálhatunk. Ehhez az alakzatot kétszer kell megjeleníteni. Az első megjelenítéskor csak azokat a fragmentumokat engedjük át, melyek alfa értéke 1, a második során csak azokat, melyek alfája nem 1. A mélységségvizsgálatot mindkét esetben engedélyezzük, de a második rajzolás előtt tegyük a mélységpuffert csak olvashatóvá.

Az alfa-vizsgálat segítségével billboarding hatás is megvalósítható. A textúra azon részeinek alfa értékét állítsuk 0-ra, amelyiket átlátszóvá akarjuk tenni, a többit 1-re, és állítsuk az alfa-vizsgálat referenciaértékét 0.5-re (vagy más 0 és 1 közé eső értékre), továbbá összehasonlító függvényként válasszuk a GL_GREATER-t.

11.1. táblázat. Az alfa-vizsgálathoz használható összehasonlító függvények

| *func* | hatása |
|---|---|
| GL_NEVER | soha nem engedi tovább a fragmentumot |
| GL_ALWAYS | mindig továbbengedi a fragmentumot |
| GL_LESS | továbbengedi ha $alfa < ref$ |
| GL_LEQUAL | továbbengedi ha $alfa \leq ref$ |
| GL_EQUAL | továbbengedi ha $alfa = ref$ |
| GL_GEQUAL | továbbengedi ha $alfa \geq ref$ |
| GL_GREATER | továbbengedi ha $alfa > ref$ |
| GL_NOTEQUAL | továbbengedi ha $alfa \neq ref$ |

## 11.3. Stencilvizsgálat

A stencilvizsgálatokat csak akkor hajtja végre a rendszer, ha van stencilpuffer, egyébként a fragmentumok ezen a ponton mindig túljutnak. A vizsgálat abból áll, hogy a rendszer összehasonlítja a fragmentumnak megfelelő pixelhez a stencilpufferben tárolt értéket egy referenciaértékkel, és az összehasonlítás eredményétől függően módosítja a pufferben tárolt értéket. A referenciaértéket és az összehasonlító függvényt a **glStencilFunc**(), a módosítást pedig a **glStencilOp**() paranccsal adhatjuk meg.

void **glStencilFunc** (GLenum *func*, GLint *ref*, GLuint *mask*);

A stencilvizsgálatnál használt összehasonlító függvényt (*func*), referenciaértéket (*ref*) és maszkot (*mask*) állítja be. A rendszer a referenciaértéket a *func* függvény szerint összehasonlítja a stencilpufferben tárolt értékkel, de annak csak azokat a bitjeit veszi figyelembe, amelyhez tartozó bitnek a *mask*-ban 1 az értéke. Ha a stencilpuffernek $s$ darab bitsíkja van, akkor a rendszer az összehasonlítás előtt a stencilpufferbeli értéket és a *mask* $s$ darab kisebb helyiértékű bitjét logikai és (AND) műveletbe hozza és ennek eredményén végzi el az összehasonlítást. A *func* paraméter lehetséges értékeit a 11.2. táblázat tartalmazza, a táblázatban *val* a fenti bitenkénti AND művelet eredményét jelöli.

Alapértelmezés szerint *func* = GL_ALWAYS, *ref = 0* és mask minden bitje *1*.

void **glStencilOp** (GLenum *fail*, GLenum *zfail*, GLenum *zpass*);

Segítségével azt adhatjuk meg, hogy a rendszer hogyan módosítsa a stencilpuffert amikor a fragmentum továbbmegy vagy fennakad a vizsgálaton. A *fail*, *zfail* és *zpass* paraméterek értékei a 11.3. táblázat szerintiek lehetnek.

Inkrementálás és dekrementálás után a rendszer a kapott értéket a $[0, 2^s - 1]$ intervallumra levágja, ahol $s$ a stencilpuffer bit/pixel mérete. A *fail* paraméterrel megadott függvényt akkor használja a rendszer, ha a fragmentum nem jut túl a stencilvizsgálaton; a *zfail*-el megadottat akkor, ha a stencilvizsgálaton túljut, de a mélységvizsgálaton nem; a *zpass*-al megadottat pedig akkor, ha a stencilvizsgálaton túljut és nincs mélységvizsgálat, vagy van de azon is túljutott. Alapértelmezés szerint *fail = zfail = zpass* = GL_KEEP.

11.2. táblázat. A stencilvizsgálathoz használható összehasonlító függvények

| *func* | hatása |
|---|---|
| GL_NEVER | soha nem engedi tovább a fragmentumot |
| GL_ALWAYS | mindig továbbengedi a fragmentumot |
| GL_LESS | továbbengedi ha $ref < val$ |
| GL_LEQUAL | továbbengedi ha $ref \leq val$ |
| GL_EQUAL | továbbengedi ha $ref = val$ |
| GL_GEQUAL | továbbengedi ha $ref \geq val$ |
| GL_GREATER | továbbengedi ha $ref > val$ |
| GL_NOTEQUAL | továbbengedi ha $ref \neq val$ |

11.3. táblázat. A **glStencilOp**() függvény paramétereinek lehetséges értékei

| érték | hatása |
|---|---|
| GL_KEEP | megtartja a kurrens értéket |
| GL_ZERO | 0-val felülírja a kurrens értéket |
| GL_REPLACE | a referenciaértékkel írja felül |
| GL_INCR | eggyel növeli a kurrens értéket |
| GL_DECR | eggyel csökkenti a kurrens értéket |
| GL_INVERT | bitenként invertálja a kurrens értéket |

A stencilvizsgálatot a **glEnable**(GL_STENCIL_TEST) paranccsal engedélyezhetjük, a **glDisable**(GL_STENCIL_TEST) paranccsal letilthatjuk, a **glIsEnabled**(GL_STENCIL_TEST) paranccsal pedig lekérdezhetjük, hogy engedélyezett-e. A stencilvizsgálatokkal kapcsolatos beállításokat a 11.4. táblázatbeli paraméterekkel kiadott **glGetIntegerv**() paranccsal kérdezhetjük le.

A stencilvizsgálat leggyakoribb alkalmazása, hogy a képernyő tetszőleges alakú területét lemaszkoljuk, azaz nem engedjük, hogy oda rajzoljon a rendszer. Ehhez előbb töltsük fel a stencilpuffert 0-val és rajzoljuk meg a kívánt alakot a stencilpufferbe 1 értékkel. Közvetlenül nem lehet a stencilpufferbe rajzolni, de a következő eljárással közvetett módon ugyanazt az eredményt kapjuk:

- a színpufferbe *zpass* = GL_REPLACE beállítás mellett rajzoljunk;
- a színmaszkot állítsuk 0-ra (vagy GL_ZERO), hogy a színpufferben ne történjen változás;
- a mélységpuffert tegyük csak olvashatóvá, ha nem akarjuk, hogy tartalma változzon.

A stencilterület megadása után állítsuk a referenciaértékeket 1-re, és az összehasonlító függvényt olyanra, hogy a fragmentum akkor jusson túl, ha a stencilsík értéke megegyezik a referenciaértékkel. A rajzolás során ne változtassuk a stencilsíkok tartalmát!

11.4. táblázat. A stencil-vizsgálatok beállításainak lekérdezéséhez használható paraméterek

| paraméter | a visszaadott érték |
|---|---|
| GL_STENCIL_FUNC | stencil függvény |
| GL_STENCIL_REF | stencil referenciaérték |
| GL_STENCIL_VALUE_MASK | stencil maszk |
| GL_STENCIL_FAIL | *fail* függvény |
| GL_STENCIL_PASS_DEPTH_FAIL | *zfail* függvény |
| GL_STENCIL_PASS_DEPTH_PASS | *zpass* függvény |

Ha egy zárt objektumot valamely vágósík elmetsz, akkor beleláthatunk a testbe. Amennyiben nem akarjuk a belsejét látni ezt konvex objektum esetén elérhetjük úgy, hogy egy konstans színű felülettel befedjük. Ennek érdekében

- töröljük a stencilpuffert 0-val;
- engedélyezzük a stencilvizsgálatot, és állítsuk az összehasonlító függvényt olyanra, hogy minden fragmentum túljusson;
- invertáljuk a stencilsíkok megfelelő értékét minden átengedett fragmentumnál;
- rajzoljuk meg a zárt konvex alakzatot.

A rajzolás után a képmező azon pixeleinek stencilértéke, melyeket nem kell lefedni 0 lesz, a lefedendőknek pedig 0-tól különböző. Állítsuk be az összehasonlító függvényt úgy, hogy csak akkor engedje át a fragmentumot, ha a stencilérték nem 0, és rajzoljunk a lefedés színével egy akkora poligont, amely az egész képmezőt lefedi.

Kitöltés mintával (stippling). Ha az alakzatok képét pontmintával akarjuk megrajzolni, kitölteni, írjuk be a mintát a stencilpufferbe (a mintával töröljük), majd a megfelelő feltételek mellett rajzoljuk meg az alakzatot.

## 11.4. Mélységvizsgálat

Az OpenGL a mélységpufferben minden pixelhez tárolja a mélységet, ami a pixelen ábrázolt objektumoknak a nézőponttól való távolságát fejezi ki. Egész pontosan: ez a mélység a képmezőkoordináta-rendszerbeli $z$ érték. A mélységpuffert leggyakrabban a láthatóság szerinti ábrázoláshoz használjuk. Ha az új fragmentum $z$ értéke kisebb mint a megfelelő pixelé, akkor mind a színpuffert, mind a mélységpuffert felülírja a rendszer az új fragmentum megfelelő értékeivel. Így végül az alakzatok láthatóság szerinti képét kapjuk. A mélységpuffer használatát a **glEnable**(GL_DEPTH_TEST) paranccsal lehet engedélyezni, a **glDisable**(GL_DEPTH_TEST) paranccsal lehet letiltani, a **glIsEnabled**(GL_DEPTH_TEST) paranccsal pedig lekérdezhető, hogy a mélységpuffer használata engedélyezett-e. Alapértelmezés szerint a mélységpuffer használata nem engedélyezett.

Ne feledjük, hogy a kép minden újrarajzolása előtt törölni kell a mélységpuffert, vagyis fel kell tölteni azzal a távolsággal, amelynél távolabbi fragmentumot nem veszünk figyelembe (lásd a 10.5. szakaszt).

void **glDepthFunc** (GLenum *func*);

A mélységek összehasonlítására használt függvényt adhatjuk meg vele. Ezen a vizsgálaton akkor jut túl egy fragmentum, ha a $z$ koordinátája és a megfelelő pixel mélysége a *func* függvénnyel előírt kapcsolatban van. A *func* paraméter lehetséges értékeit a 11.5. táblázat mutatja, $fz$ a fragmentum mélységét, $bz$ a pufferben tárolt mélységet jelöli. Alapértelmezés: *func* = GL_LESS.

11.5. táblázat. A mélységek összehasonlításához használható függvények

| *func* | hatása |
|---|---|
| GL_NEVER | soha nem engedi tovább a fragmentumot |
| GL_ALWAYS | mindig továbbengedi a fragmentumot |
| GL_LESS | továbbengedi ha $fz < bz$ |
| GL_LEQUAL | továbbengedi ha $fz \leq bz$ |
| GL_EQUAL | továbbengedi ha $fz = bz$ |
| GL_GEQUAL | továbbengedi ha $fz \geq bz$ |
| GL_GREATER | továbbengedi ha $fz > bz$ |
| GL_NOTEQUAL | továbbengedi ha $fz \neq bz$ |

A **glGetIntegerv**() parancsot a GL_DEPTH_FUNC paraméterrel kiadva a kurrens mélységösszehasonlítás függvényét kapjuk meg.

## 11.5. Színkombinálás, dithering, logikai műveletek

Ha egy új fragmentum az előző vizsgálatokon – kivágás, alfa, stencil, mélység – túljut, akkor annak színét a megfelelő pixel színével többféleképpen kapcsolatba hozhatjuk. Legegyszerűbb esetben, ami egyben az alapértelmezés is, felülírjuk a fragmentum színével a pixel színét. Átlátszó objektumok rajzolásakor, vagy a képhatár kisimításakor vegyíthetjük a két színt. Ha az ábrázolható színek számát a felbontás csökkentése révén növelni akarjuk, akkor a dithering hatást alkalmazhatjuk. Végül színindex módban tetszőleges bitenkénti logikai műveletet hajthatunk végre a színeken.

### 11.5.1. Színkombinálás

A színkombinálás az új fragmentum és a neki megfelelő pixelhez tárolt R, G, B és alfa komponensek vegyítését jelenti. A vegyítés többféleképpen végrehajtható, de minden esetben a fragmentum és a pixelhez tárolt (ha tárolja a rendszer) alfa értékek alapján történik.

### 11.5.2. Dithering

A grafikus hardverek egy része lehetőséget biztosít arra, hogy a megjeleníthető színek számát a felbontás rovására növeljük. Új szín úgy állítható elő, hogy az eredetileg használható színekkel kiszínezett pixelmintákat jelenítünk meg, melyek új szín hatását eredményezik. Vagyis a megjelenítés során a rajzolás pixele a képernyő pixeleinek egy négyzetes tömbje lesz, azaz csökken a felbontás, de ezek a pixeltömbök új, az eredeti pixeleken el nem érhető színűek lesznek. Ilyen technikát használnak a fekete-fehér újságok a fényképek megjelenítésénél, amikor a szürke különböző árnyalatait közelítik az előbb vázolt technikával.

A dithering megvalósítása teljesen hardverfüggő, az OpenGL-ből csak engedélyezhetjük, illetve letilthatjuk a **glEnable**(GL_DITHER), illetve **glDisable**(GL_DITHER) parancsokkal. Alapértelmezés szerint a dithering nem engedélyezett.

Ez a technika mind színindex, mind RGBA módban használható. RGBA módban ez az utolsó művelet a színpufferbe való beírás előtt, színindex módban még bitenkénti logikai műveletek végrehajtására is lehetőség van.

### 11.5.3. Logikai műveletek

Színindex módban az indexek tekinthetők egész számként és bitsorozatként is. Egész számnak célszerű tekinteni a színindexeket pl. árnyalásnál és ditheringnél. Ha azonban különböző rétegekre készült rajzok kompozíciójaként állítunk elő egy képet, vagyis különböző írási maszkot használunk, hogy a rajzolást a különböző bitsíkokra korlátozzuk, célszerű a színindexeket bitsorozatnak tekinteni.

A logikai műveleteket az új fragmentum és a megfelelő pixel színindexének bitsorozatán hajthatjuk végre. Ezek a műveletek különösen hasznosak és gyorsan végrehajthatók a bittömbök gyors mozgatására specializált (bitBlt – bit block transfer) hardverű grafikus berendezések esetén. Az OpenGL-ben a **glLogicOp**() parancs segítségével választhatjuk ki a végrehajtandó logikai műveletet.

void **glLogicOp** (GLenum *opcode*);

Színindex módban az új fragmentum $fi$ színindexén és a neki megfelelő pixel $pi$ színindexén végrehajtandó bitenkénti logikai művelet kiválasztására szolgál. Az *opcode* paraméter lehetséges értékeit és azok jelentését a 11.6. táblázat mutatja. Alapértelmezés szerint *opcode* = GL_COPY.

A logikai műveletek végrehajtását a **glEnable**(GL_LOGIC_OP) paranccsal engedélyezhetjük, a **glDisable**(GL_LOGIC_OP) paranccsal letilthatjuk és a **glIsEnabled**(GL_LOGIC_OP) paranccsal lekérdezhetjük, hogy engedélyezett-e. Alapértelmezés szerint nem engedélyezett. A GL_LOGIC_OP_MODE paraméterrel kiadott **glGetIntegerv**() paranccsal a kurrens logikai művelet kérdezhető le.

11.6. táblázat. A **glLogicOp**() függvénnyel előírható bitenkénti logikai műveletek

| *opcode* | hatása |
|---|---|
| GL_CLEAR | 0 |
| GL_COPY | $fi$ |
| GL_NOOP | $pi$ |
| GL_SET | 1 |
| GL_COPY_INVERTED | $\neg fi$ |
| GL_INVERT | $\neg pi$ |
| GL_AND_REVERSE | $fi \wedge \neg pi$ |
| GL_OR_REVERSE | $fi \vee \neg pi$ |
| GL_AND | $fi \wedge pi$ |
| GL_OR | $fi \vee pi$ |
| GL_NAND | $\neg(fi \wedge pi)$ |
| GL_NOR | $\neg(fi \vee pi)$ |
| GL_XOR | $fi\, XOR\, pi$ |
| GL_EQUIV | $\neg(fi\, XOR\, pi)$ |
| GL_AND_INVERTED | $\neg fi \wedge pi$ |
| GL_OR_INVERTED | $\neg fi \vee pi$ |

# 12. fejezet

# Kiválasztás, visszacsatolás

Interaktív grafikus programok írásakor gyakran van szükség arra, hogy a képernyőn látható objektumok közül válasszunk a grafikus kurzor segítségével. Ezt a grafikus inputot angolul pick inputnak, magyarul rámutató azonosításnak nevezzük. Ezt a funkciót támogatja az OpenGL kiválasztási mechanizmusa.

Ugyancsak fontos, hogy az OpenGL felhasználásával megírt alkalmazásaink futási eredményét más rendszer vagy eszköz bemenő adatként használhassa (pl. CAD rendszer, vagy plotter). Erre szolgál a visszacsatolási mechanizmus.

Mindkét funkcióhoz szükségünk van az OpenGL működési módjának beállítására.

GLint **glRenderMode** (GLenum *mode*);

A megjelenítés, kiválasztás és visszacsatolási üzemmódok közötti választást teszi lehetővé a mode paraméternek adott GL_RENDER, GL_SELECT és GL_FEEDBACK értéknek megfelelően. A rendszer a megadott üzemmódban marad a következő, más paraméterrel kiadott **glRenderMode**() parancs kiadásáig. A kiválasztási üzemmód beállítása előtt a **glSelectBuffer**(), a visszacsatolási üzemmód előtt a **glFeedbackBuffer**() parancsot kell kiadni. A **glRenderMode**() parancs által visszaadott értéknek akkor van jelentése, ha a kurrens működési mód (amiről átváltunk) vagy GL_SELECT, vagy GL_FEEDBACK. Kiválasztási üzemmódból való kilépés esetén a visszaadott érték a találatok száma, visszacsatolási üzemmódnál pedig a visszacsatolási pufferbe tett adatok száma. A negatív visszaadott érték azt jelzi, hogy a megfelelő puffer (kiválasztási, vagy visszacsatolási) túlcsordult.

A GL_RENDER_MODE paraméterrel kiadott **glGetIntegerv**() paranccsal lekérdezhetjük a kurrens üzemmódot.

Kiválasztási és visszacsatolási üzemmódban a rajzolási utasításoknak nincs látható eredménye, ilyenkor csak a megfelelő adatokat tárolja a rendszer a korábban létrehozott pufferbe.

## 12.1. Kiválasztás

A rámutató azonosítás megvalósítása érdekében:

- létre kell hozni az un. kiválasztási puffert, amiben a választás eredményét kapjuk vissza;

- át kell térni kiválasztási üzemmódra;
- olyan speciális vetítési mátrixot kell megadni, amelyik csak az adott pozíció (praktikusan a grafikus kurzor pozíciója) kis környezetét képezi le;
- a képet újra kell rajzolni úgy, hogy a választható képelemeket azonosítóval kell ellátni;
- vissza kell térni rajzolási üzemmódra, amivel a kiválasztási pufferben megkapjuk azon képelemek azonosítóját, melyeket legalább részben tartalmazott a speciális vetítési mátrixszal megadott térrész.

void **glSelectBuffer** (GLsizei *size*, GLuint **buffer*);

A kiválasztott képelemek adatainak visszaadására szolgáló puffer megadása. A *buffer* paraméter a *size* méretű előjel nélküli egészekből álló puffer címe. Ezt a parancsot mindig ki kell adnunk, mielőtt kiválasztási módra áttérnénk.

A választhatóság érdekében a képelemeket azonosítóval kell ellátni. A rendszer egy veremszerkezetet tart karban ezen azonosítók számára. Ezt névveremnek nevezzük. Az újonnan létrehozott képelem azonosítója a névverem legfelső szintjén tárolt azonosító lesz.

void **glInitNames** (void);

A névvermet törli.

void **glPushName** (GLuint *name*);

A *name* azonosítót a névverem legfelső szintjére teszi. A névverem mélysége implementációfüggő, de legalább 64. A GL_MAX_NAME_STACK_DEPTH paraméterrel kiadott **glGetIntegerv**() paranccsal kaphatjuk meg a pontos értéket. A megengedettnél több szint használatára tett kísérlet a GL_STACK_OVERFLOW hibát eredményezi.

void **glPopName** (void);

A névverem legfelső szintjén lévő azonosítót eldobja. Üres veremre kiadott **glPopName**() parancs a GL_STACK_UNDERFLOW hibát eredményezi.

void **glLoadName** (GLuint *name*);

A névverem legfelső szintjén lévő értéket a name értékkel felülírja. Ha a névverem üres, mint pl. a **glInitNames**() parancs kiadása után, akkor a GL_INVALID_OPERATION hiba keletkezik. Ennek elkerülése érdekében, üres verem esetén egy tetszőleges értéket tegyünk a verembe a **glPushName**() paranccsal, mielőtt a **glLoadName**() parancsot kiadnánk.

void **gluPickMatrix** (GLdouble *x*, GLdouble *y,* GLdouble *width*, GLdouble *height*, GLint *viewport[4]*);

Azt a vetítési mátrixot hozza létre, amely a képmező $(x, y)$ középpontú, *width* szélességű és *height* magasságú területére korlátozza a rajzolást, és ezzel a mátrixszal megszorozza a kurrens mátrixverem legfelső elemét. A képmező-koordinátákban megadott $(x, y)$ általában a kurzor pozíciója, a szélesség és magasság által kijelölt tartomány pedig a rámutató azonosító eszköz érzékenységeként fogható fel. A *viewport[]* paraméter a kurrens képmező határait tartalmazza, amit a

```
glGetIntegerv(GL_VIEWPORT,viewport);
```

paranccsal kaphatunk meg.

## 12.2. Visszacsatolás

A visszacsatolás megvalósítása nagyon hasonló a kiválasztáséhoz. A folyamat az alábbi lépésekből áll:

- létre kell hozni az un. visszacsatolási puffert, amiben a képelemek adatait majd megkapjuk;
- át kell térni visszacsatolási üzemmódra;
- újra kell rajzolni a képet;
- vissza kell térni rajzolási üzemmódra, amivel megkapjuk a pufferbe írt adatok számát;
- a pufferben visszakapott adatokat kiolvashatjuk, feldolgozhatjuk.

void **glFeedbackBuffer** (GLsizei *size*, GLenum *type*, GLfloat **buffer*);

A visszacsatolási üzemmód adatainak tárolására hoz létre egy puffert. *size* a pufferben tárolható adatok száma, *buffer* a tárterület címe. A *type* paraméter azt specifikálja, hogy a rendszer milyen adatokat adjon vissza a csúcspontokról. A lehetséges értékeket a 12.1. táblázat tartalmazza. A táblázatban $k$ értéke 1 színindexmód esetén, 4 RGBA módnál. A **glFeedbackBuffer**() parancsot ki kell adni mielőtt visszacsatolási üzemmódba lépnénk.

void **glPassThrough** (GLfloat *token*);

Ha visszacsatolási módban hívjuk meg, akkor egy markert tesz a visszaadott adatok közé, más működési módban meghívása hatástalan. A marker a GL_PASS_THROUGH_TOKEN konstansból és a *token* lebegőpontos értékből áll. A **glBegin**() és **glEnd**() közötti kiadása a GL_INVALID_OPERATION hibát eredményezi.

12.1. táblázat. A visszaadandó adatok típusa

| *type* | koordináták | szín | textúra | összes |
|---|---|---|---|---|
| GL_2D | $k$ | – | – | 2 |
| GL_3D | $x, y, z$ | – | – | 3 |
| GL_3D_COLOR | $x, y, z$ | $k$ | – | $3 + k$ |
| GL_3D_COLOR_TEXTURE | $x, y, z$ | $k$ | 4 | $7 + k$ |
| GL_4D_COLOR_TEXTURE | $x, y, z$ | $k$ | 4 | $8 + k$ |

12.2. táblázat. A visszaadott kódok

| alapelem típusa | kód | adat |
|---|---|---|
| pont | GL_POINT_TOKEN | csúcspont |
| szakasz | GL_LINE_TOKEN<br>GL_LINE_RESET_TOKEN | két csúcspont |
| poligon | GL_POLYGON_TOKEN | csúcspontok |
| bittérkép | GL_BITMAP_TOKEN | csúcspont |
| pixeltömb | GL_DRAW_PIXEL_TOKEN<br>GL_COPY_PIXEL_TOKEN | csúcspont |
| megjegyzés | GL_PASS_THROUGH_TOKEN | lebegőpontos szám |

# 13. fejezet

# Textúrák

A valószerű képek létrehozásának egy fontos eszköze a textúraleképezés. A valóságban a tárgyak nem egyszínűek, hanem a felületükön minták vannak. Ezen minták geometriai leírása és geometriai objektumként való kezelése (amit az eddig tárgyalt eszközök lehetővé tennének) azonban rendkívül időigényes és nagy tárolókapacitást lekötő feladat lenne. Mindamellett nem is lenne elég valószerű, hiszen a minták általában éles kontúrúak, túl szabályosak lennének. Ezzel szemben, ha egy valódi objektum felületének képét (mintázatát) visszük fel a számítógéppel előállított objektumra, sokkal életszerűbb lesz a kép. A térbeliség érzékeltetésének fokozására is alkalmas a textúra, ugyanis a textúrát a modelltérben rendeljük hozzá a geometriai alakzathoz, így a felületi mintán is érvényesül a centrális vetítésből eredő, un. perspektív torzítás, pl. egy téglafal távolabb lévő téglái kisebbek lesznek a képen, és a téglák szemközti élei nem lesznek párhuzamosak.

A textúrákat többnyire kétdimenziósnak (síkbelinek) gondoljuk, azonban a textúrák lehetnek egy- vagy háromdimenziósak is. A textúrákat leképezhetjük úgy, hogy azok lefedjék a poligonokat (poligonhálókat), de úgy is, hogy a textúra az objektum szintvonalait, vagy más jellemzőit szemléltesse. Az erősen ragyogó felületű objektumok úgy is textúrázhatók, hogy azt a hatást keltsék, mintha a környezet tükröződne az objektumon. A textúra geometriailag egy téglalap alakú terület, mely sorokba és oszlopokba rendezett textúraelemekből, röviden texelekből (texture element) épül fel. A textúra tehát adatok $1, 2$, vagy 3 dimenziós tömbjének tekinthető. Az egyes texelekhez tárolt adat képviselhet színt, fényerősséget vagy szín és alfa értéket, azaz $1, 2, 3$ vagy 4 adat tartozhat minden egyes texelhez.

A téglalap alakú textúrákat azonban tetszőleges alakú poligonokra, poligonhálókra kell ráhelyezni. A ráhelyezés mikéntjét a modelltérben kell megadni, így a textúrákra is hatnak a modell- és vetítési transzformációk. Ennek érdekében az objektumok létrehozásakor a csúcspontok geometriai koordinátái mellett a textúrakoordinátákat is meg kell adni. Egy kétdimenziós textúra koordinátái a $[0., 1.] \times [0., 1.]$ egységnégyzeten belül változnak. Amikor a csúcspontokhoz hozzárendeljük a textúra pontjait ezen a területen kívül eső koordinátapárt is megadhatunk, de elő kell írnunk, hogy az egységnégyzeten kívüli koordinátákat hogy értelmezze a rendszer, pl. ismételje a textúrát (tegye egymás mellé).

A textúra hatásának érvényesítése a képmezőkoordináta-rendszerében az ábrázolandó objektum fragmentálása után történik. Így előfordulhat, hogy a transzformációkon átesett textúra több eleme látszik egy fragmentumon, vagy ellenkezőleg, több fragmentumon látszik egyetlen texel. Ennek a problémáknak többféle megoldását kínálja a

rendszer, a felhasználók által kiválasztható un. szűrési műveleteken keresztül. Ezek a műveletek rendkívül számításigényesek, ezért a fejlettebb grafikus munkahelyek hardverből támogatják a textúraleképezést. Ha több textúrát felváltva használunk, akkor célszerű textúraobjektumokat létrehozni, melyek egy-egy textúrát (esetleg több felbontásban) tartalmaznak. Néhány OpenGL implementációban textúraobjektumok munkacsoportját lehet létrehozni. A csoporthoz tartozó textúrák használata hatékonyabb, mint a csoporton kívülieké. Ezeket a nagy hatékonyságú textúraobjektumokat rezidensnek nevezik, és ezek használatát általában hardveres vagy szoftveres gyorsítók segítik.

A felhasználó által előírható, hogy a textúra hogyan hasson a megjelenítendő objektum színére. Megadhatjuk, hogy a megfelelő texel(ek) színe legyen a fragmentum színe (egyszerűen felülírja a fragmentum színét, mintha egy matricát ragasztanánk rá), előírhatjuk, hogy a textúrával módosítsa (pontosabban skálázza) a fragmentum színét, ami a megvilágítás és textúrázás hatását kombinálja, végül a fragmentum színének és egy konstans színnek a textúraelemen alapuló keverését is előírhatjuk.
Textúrázott poligonháló megjelenítéséhez még a legegyszerűbb esetben is az alábbi lépéseket kell megtenni:

- engedélyezni kell a textúraleképezést;
- létre kell hozni egy textúraobjektumot és hozzá kell rendelni egy textúrát;
- meg kell adni, hogy a textúrát hogyan alkalmazza a rendszer a fragmentumokra (szűrő megadása);
- engedélyezni kell a textúraleképezést;
- meg kell rajzolni a textúrázandó objektumokat úgy, hogy a csúcspontok geometriai koordinátái mellett megadjuk a textúrakoordinátáit is.

## 13.1. A textúraleképezés engedélyezése

Az objektumok textúrázott megjelenítéséhez a megrajzolásuk előtt engedélyezni kell a textúraleképezést. Az engedélyezés, illetve letiltás a szokásos **glEnable**(), illetve **glDisable**() parancsoknak a GL_TEXTURE_1D, GL_TEXTURE_2D vagy GL_TEXTURE_3D szimbolikus konstansokkal való kiadásával lehetséges. Ezekkel rendre az 1, 2, illetve 3 dimenziós textúraleképezést engedélyezhetjük vagy tilthatjuk le. Ha egyszerre több textúraleképezés is engedélyezett, a rendszer mindig a legnagyobb dimenziójút veszi figyelembe. Az ilyen helyzeteket azonban lehetőség szerint kerüljük el!

## 13.2. Textúra megadása

Előbb a természetesnek tűnő kétdimenziós textúra megadásával, majd az első hallásra kevésbé kézenfekvő egy- és háromdimenziós textúrákéval foglalkozunk.

void **glTexImage2D** (GLenum *target*, GLint *level*, GLint *internalFormat*, GLsizei *width*, GLsizei *height*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Kétdimenziós textúrát hoz létre. A *target* paraméter értéke GL_TEXTURE_2D vagy GL_PROXY_TEXTURE_2D lehet. A *level* paramétert akkor használjuk, ha a textúrát több felbontásban is tárolni akarjuk, egyébként a *level* paraméternek adjuk a 0 értéket. (A több felbontásban tárolt textúrák használatát a 13.7. szakaszban tárgyaljuk.)

Az *internalFormat* (belső formátum) paraméterrel azt írjuk elő, hogy egy kép texeleinek a leírásához az R, G, B, A színkomponensek melyikét, esetleg fényerősséget vagy színerősséget használja a rendszer. Értéke 1, 2, 3, 4 vagy az alábbi előre definiált 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_RGB, GL_R3_G3_B2,
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,
GL_RGBA16.

Ezen konstansok jelentését a textúrafüggvények leírásánál ismertetjük.

Az OpenGL 1.0 verzióval való kompatibilitás érdekében az 1, 2, 3, illetve 4 értékek rendre a GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, illetve GL_RGBA szimbolikus konstansoknak felelnek meg. A szimbolikus konstansokkal a texelekhez tárolt komponenseket és azok pontosságára vonatkozó kívánságunkat adhatjuk meg, pl. a GL_R3_G3_B2 konstanssal azt, hogy a piros és zöld színt 3-3 biten, a kéket pedig 2 biten tárolja a rendszer. Ez nem jelenti azt, hogy a rendszer biztosan így is hajtja végre a parancsot. Csak az garantált, hogy az OpenGL az implementáció által támogatott formátumok közül az ehhez legközelebb álló szerint tárolja a texelekhez kapcsolt adatokat.

A *width* és *height* paraméterek a textúra szélességét, illetve magasságát adják meg. A *border* a textúra határának szélességét adja meg, értéke vagy 0, vagy 1 lehet. A *width*, illetve *height* paraméterek értékének $2^w+2b$, illetve $2^h+2b$ alakúnak kell lenni, ahol $0 \leq h, w$ egész számok és $b$ a *border* értéke. A textúrák méretének maximuma implementációfüggő, de legalább $64 \times 64$, illetve határral $66 \times 66$ -nak kell lennie.

A *format* és *type* paraméterek a textúraadatok formátumát és típusát írják le. Jelentésük megegyezik a **glDrawPixels**() parancs megfelelő paramétereinek jelentésével. Valójában a textúraadatok ugyanolyan formátumúak, mint a **glDrawPixels**() parancs által használt adatoké, ezért a **glPixelStorage***() és **glPixelTransfer***() parancsok által beállított formátumokat használja mindkettő. A *format* paraméter értéke GL_COLOR_INDEX, GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE vagy GL_LUMINANCE_ALPHA lehet, vagyis a GL_STENCIL_INDEX és GL_DEPTH_COMPONENT kivételével azokat, amelyeket a **glDrawPixels**() is használ.

A *type* paraméter pedig a GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_BITMAP vagy a tömörített pixeladatok valamelyikét.

A *texels* a létrehozandó textúra adatainak tárolására szolgáló tömb címe. A belső tárolási formátumnak (*internalFormat*) hatása van a textúraműveletek végrehajtási sebességére is. Az azonban, hogy melyik hatékonyabb, teljes mértékben implementációfüggő.

A textúrák sorai és oszlopai számának (a határt levonva) 2 hatványának kell lenni. Ha egy olyan képből szeretnénk textúrát létrehozni, amely mérete ennek nem tesz eleget, a GLU függvénykönyvtár **gluScaleImage**() függvényét célszerű használni.

int **gluScaleImage** (GLenum *format*, GLint *widthin*, GLint *heightin*, GLenum *typein*, const void **datain*, GLint *widthout*, GLint *heightout*, GLenum *typeout*, void **dataout*);

A *datain* címen tárolt képadatokat a megfelelő pixeltárolási módot használva kicsomagolja, a kívánt méretűvé skálázza, majd a *dataout* címen kezdődő memóriaterületre letárolja. A *format*, *typein* (a bejövő adatok típusa) és a *typeout* (a kimenő adatok típusa) a **glDrawPixels**() által támogatott bármely formátum, illetve típus lehet. A bejövő $widthin \times heightin$ méretű képet lineáris transzformációval és box szűréssel transzformálja $widthout \times heightout$ méretűvé, majd ezt a *dataout* címen kezdődő területre kiírja a kurrens GL_PACK* pixeltárolási módnak megfelelően. Az eredmény tárolására alkalmas memória lefoglalását a függvény meghívása előtt el kell végezni.

A visszaadott érték 0, ha a függvény sikeresen végrehajtódott, egyébként egy GLU hibaüzenet.

(Az OpenGL 1.2-ben bevezetett tömörített pixeltárolási formátumokat a GLU 1.3 támogatja.)

A képpuffer tartalmából is létrehozható textúra. Erre szolgál a **glCopyTexImage2D**(), mely a képpufferből kiolvas egy téglalap alakú területet, majd ennek pixeleiből létrehozza a textúra texeleit.

void **glCopyTexImage2D** (GLenum *target*, GLint *level*, GLint *internalFormat*, GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*, GLint *border*)

A képpuffer adataiból egy kétdimenziós textúrát hoz létre. A pixeleket a kurrens (GL_READ_BUFFER) pufferből kiolvassa és úgy dolgozza fel őket, mint a **glCopyPixels**() parancs esetén azzal a különbséggel, hogy a pixeleket nem a képpufferbe, hanem a textúra számára lefoglalt memóriába másolja. A **glPixelTransfer***() beállításait és a többi pixelátviteli műveletet használja.
A *target* paraméternek a GL_TEXTURE_2D értéket kell adni. A *level*, *internalFormat*, *border*, *width* és *height* paraméterek jelentése ugyanaz, mint a **glTexImage2D**() esetén. *x, y* a kimásolandó pixeltömb bal alsó sarkának a képpufferbeli koordinátái.

## 13.3. Textúrahelyettesítő

A textúrát használó OpenGL programok memóriaigénye tekintélyes, és implementációtól függően különböző megszorítások vannak a textúraformátumokra. Ezért egy speciális textúrahelyettesítő (proxy) használható annak eldöntésére, hogy vajon az adott OpenGL implementáció tudna-e kezelni egy adott textúrát a kívánt felbontás(ok)ban.
Ha a **glGetIntegerv**() függvényt meghívjuk a GL_MAX_TEXTURE_SIZE paraméterrel, akkor megkapjuk, hogy az adott implementációban legfeljebb mekkora méretű (szélesség, magasság a határok nélkül) 2D-s textúrák használhatók. A 3D-s textúrák esetén ugyanezt a GL_MAX_3D_TEXTURE_SIZE paraméter használatával érjük el. A fenti érték meghatározásakor azonban a rendszer nem veszi (nem veheti) figyelembe a textúrák tárolásának formátumát és azt, hogy a textúrát esetleg több felbontásban is tárolni kell (mipmapping).
Egy speciális textúrahelyettesítő funkció segít abban, hogy pontosan lekérdezhető legyen az OpenGL-től, hogy egy textúra adott formátum szerint tárolható-e a használt implementációban. Ehhez a textúrák létrehozását segítő **glTexImage2D**() függvényt kell meghívni úgy, hogy a *target* paraméternek GL_PROXY_TEXTURE_2D értéket adjuk, a *level*, *internalFormat*, *width*, *height*, *border*, *format* és *type* paramétereknek ugyanazt az értéket adjuk, mintha ténylegesen létre akarnánk hozni az adott textúrát, a *texels* paraméterben pedig a NULL címet adjuk. Ezután a **glGetTexLevelParameter**() paranccsal lekérdezhetjük, hogy rendelkezésre áll-e elegendő erőforrás a textúra létrehozására. Ha nem áll rendelkezésre, akkor a lekérdezett textúra jellemző értékeként 0-át ad vissza a függvény. Ezzel a függvénnyel ténylegesen létrehozott textúrák paraméterei is lekérdezhetők, nemcsak az információszerzés érdekében létrehozott textúrahelyettesítőké.

void **glGetTexLevelParameter**{if}**v** (GLenum *target*, GLint *level*, GLenum *pname*, TYPE **params*);

A *level* paraméterrel megadott részletezési szintnek megfelelő textúraparamétereket adja vissza a *params* változóban.

A *target* paraméter lehetséges értékei: GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_PROXY_TEXTURE_1D, GL_PROXY_TEXTURE_2D, GL_PROXY_TEXTURE_3D.

A *pname* paraméter lehetséges értékei:

GL_TEXTURE_WIDTH, GL_TEXTURE_HEIGHT, GL_TEXTURE_DEPTH,
GL_TEXTURE_BORDER, GL_TEXTURE_INTERNAL_FORMAT,
GL_TEXTURE_RED_SIZE, GL_TEXTURE_GREEN_SIZE,
GL_TEXTURE_BLUE_SIZE, GL_TEXTURE_ALPHA_SIZE,
GL_TEXTURE_LUMINANCE_SIZE, GL_TEXTURE_INTENSITY_SIZE.

Az OpenGL 1.0 verzióval való kompatibilitás érdekében a *pname* paraméternek a GL_TEXTURE_COMPONENTS érték is adható, a magasabb verziójú OpenGL implementációknál azonban a GL_TEXTURE_INTERNAL_FORMAT a javasolt szimbolikus konstans.
Példa a textúrahelyettesítő használatára

```
GLint height;
   glTexImage2D(GL_PROXY_TEXTURE_2D, 0, GL_RGBA8, 64, 64, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
   glGetTexLevelPrarameteriv(GL_PROXY_TEXTURE_2D, 0, GL_TEXTURE_HEIGHT,
&height);
```

A textúrahelyettesítő mechanizmusnak jelentős hiányossága, hogy segítségével csak azt tudjuk meg, hogy a rendszer képes lenne-e betölteni az adott textúrát. Nem veszi figyelembe, hogy pillanatnyilag a textúra erőforrások milyen mértékben foglaltak, vagyis azt nem tudjuk meg, hogy pillanatnyilag van-e elegendő kapacitás az adott textúra használatára.

## 13.4. Textúrák módosítása

Egy új textúra létrehozása több számítási kapacitást köt le, mint egy meglévő módosítása. Ezért az OpenGL-ben lehetőség van a textúrák tartalmának részbeni vagy teljes átírására. Ez különösen hasznos lehet olyan esetekben, amikor a textúrák alapját valós időben létrehozott videoképek alkotják. A textúra módosítása mellet szól az is, hogy míg a textúra szélességének és magasságának 2 hatványának kell lenni, addig egy textúra tetszőleges méretű téglalap alakú területe átírható, és a videoképek méretei általában nem 2 hatványai.

void **glTexSubImage2D** (GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLsizei *width*, GLsizei *height*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

A kurrens kétdimenziós textúra egy téglalap alakú területét átírja a *texels* címen kezdődő textúraadatokkal. A *target* paraméternek a GL_TEXTURE_2D értéket kell adni. A *level*, *format* és *type* paraméterek értelmezése a **glTexImage2D**() parancsnál használtakkal megegyezik. *xoffset*, *yoffset* az átírandó téglalap alakú terület bal alsó sarkának koordinátái a textúra bal alsó sarkához képest, *width* a téglalap szélessége, *height* a magassága. Az így elhelyezett téglalap nem lóghat ki a textúrát leíró téglalapból. Ha a szélesség vagy magasság értéke 0, akkor a parancsnak nem lesz hatása a kurrens textúrára, de nem kapunk hibaüzenetet.

A képrészre a **glPixelStore***(), **glPixelTransfer***() által beállított módok, továbbá az egyéb pixelátviteli műveletek is hatással vannak.

A textúrák létrehozásához hasonlóan a textúrák módosításához is használhatjuk a képpuffert.

void **glCopyTexSubImage2D** (GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLint x, GLint y, GLsizei *width*, GLsizei *height*);

A képpufferből kimásolt képpel helyettesíti a kurrens kétdimenziós textúra megadott téglalap alakú területét. A kurrens GL_READ_BUFFER pufferből kiolvassa a megadott pixeleket, és pontosan úgy dolgozza fel, mint a **glCopyPixels**() parancs esetén azzal a különbséggel, hogy nem a képpufferbe másolja hanem a textúramemóriába. A **glPixelTransfer***() és a többi pixelátviteli művelet beállításait használja.

A *target* és *level* paraméterek értelmezése megegyezik a **glTexSubImage2D**() parancs megfelelő paramétereinek értelmezésével. A textúra átírandó területét az *xoffset*, *yoffset*, *width*, *height* paraméterek határozzák meg ugyanúgy, mint **glTexSubImage2D**() parancs esetén. Erre a területre a képpuffernek az a téglalapja kerül, mely bal alsó sarkának koordinátái *x, y* szélessége *width*, magassága *height*.

## 13.5. Egydimenziós textúrák

Előfordulhat, hogy a mintázat csak egy dimenzióban változik. Ilyen esetben elegendő (takarékosabb) egydimenziós textúrát használni. Az egydimenziós textúra olyan speciális kétdimenziós textúrának tekinthető, melynek alul és felül nincs határa és a magassága 1 ($height = 1$). Az eddig megismert, kétdimenziós textúrákat létrehozó, módosító parancsok mindegyikének van egydimenziós megfelelője.

void **glTexImage1D** (GLenum *target*, GLint *level*, GLint *internalFormat*, GLsizei *width*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Egydimenziós textúrát hoz létre. A *target* paraméternek a GL_TEXTURE_1D vagy GL_TEXTURE_PROXY_1D értéket kell adni. A többi paraméter jelentése megegyezik a **glTexImage2D**() parancs megfelelő paraméterének jelentésével.

void **glTexSubImage1D** (GLenum *target*, GLint *level*, GLint *xoffset*, GLsizei *width*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Egydimenziós textúratömböt hoz létre, mellyel felülírja a kurrens egydimenziós textúra megadott szakaszát. A *target* paraméternek a GL_TEXTURE_1D értéket kell adni. A többi paraméter jelentése megegyezik a **glTexSubImage2D**() parancs megfelelő paraméterének jelentésével.

void **glCopyTexImage1D** (GLenum *target*, GLint *level*, GLint *internalFormat*, GLint *x*, GLint *y*, GLsizei *width*, GLint *border*)

Egydimenziós textúrát hoz létre a képpuffer felhasználásával. A *target* paraméternek a GL_TEXTURE_1D értéket kell adni. A többi paraméter jelentése megegyezik a **glCopyTexImage2D**() parancs megfelelő paraméterének jelentésével.

void **glCopyTexSubImage1D** (GLenum *target*, GLint *level*, GLint *xoffset*, GLint *x*, GLint *y*, GLsizei *width*);

A kurrens egydimenziós textúra kijelölt részét felülírja a képpufferből vett adatokkal. A *target* paraméternek a GL_TEXTURE_1D értéket kell adni. A többi paraméter jelentése megegyezik a **glCopyTexSubImage2D**() parancs megfelelő paraméterének jelentésével.

## 13.6. Háromdimenziós textúrák

A háromdimenziós textúrák egymás mögé helyezett kétdimenziós textúrák (textúrarétegek) összességének tekinthetők. Az ilyen textúrák adatai tehát háromdimenziós tömbben tárolhatók. A háromdimenziós textúrák leggyakoribb alkalmazási területe az orvostudomány és a geológia, ahol a textúra rétegei CT (computer tomograph) vagy MRI (magnetic resonance image) képeket, illetve kőzetréteget képviselnek. A háromdimenziós textúrák az OpenGL 1.2 verzióban váltak a rendszer magjának részévé, korábban csak elterjedt kiegészítések voltak.
Minden kétdimenziós textúrát létrehozó és manipuláló parancsnak megvan a háromdimenziós megfelelője.

void **glTexImage3D** (GLenum *target*, GLint *level*, GLint *internalFormat*, GLsizei *width*, GLsizei *height*, GLsizei *depth*, GLint *border*, GLenum *format*, GLenum *type*, const GLvoid **texels*);

Háromdimenziós textúrát hoz létre. A *target* paraméternek a GL_TEXTURE_3D vagy GL_TEXTURE_PROXY_3D értéket kell adni. A *level*, *internalFormat*, *width*, *height*, *border*, *format*, *type* és *texels* paraméterekre ugyanaz vonatkozik, mint a **glTexImage2D**() parancsnál. A *width*, illetve *height* paraméterek rendre a rétegek szélességét, illetve magasságát jelölik, a *depth* paraméter a rétegek számát, a textúra mélységét. A *depth* paraméterre ugyanazok a megszorítások érvényesek, mint a *width* és *height* paraméterekre.

void **glTexSubImage3D** (GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLint *zoffset*, GLsizei *width*, GLsizei *height*, GLsizei *depth*, GLint *format*, GLenum *type*, const GLvoid **texels*);

A kurrens háromdimenziós textúra kijelölt részét írja felül. A *target* paraméternek a GL_TEXTURE_3D értéket kell adni. A *level* és *type* paraméterek értelmezése a **glTexImage3D**() parancsnál használt ugyanilyen nevű paraméterekével megegyezik. A *width*, *height*, *depth* paraméterek rendre a módosítandó téglatest szélességét, magasságát és mélységét jelölik, az *xoffset*, *yoffset*, *zoffset* hármas pedig ezen téglatest bal alsó sarkának a textúra bal alsó csúcspontjához viszonyított koordinátáit. A módosítandó textúrarész nem lóghat ki a textúrából. Az új textúraadatok a *texels* címen kezdődnek.

void **glCopyTexSubImage3D** (GLenum *target*, GLint *level*, GLint *xoffset*, GLint *yoffset*, GLint *zoffset*, GLint *x*, GLint *y*, GLsizei *width*, GLsizei *height*);

A kurrens háromdimenziós textúra megadott rétegének előírt téglalap alakú területét felülírja a képpufferből kivett adatokkal. A kurrens GL_READ_BUFFER pufferből kiolvasott pixeleket ugyanúgy dolgozza fel, mint a **glCopyPixels**() parancs, az egyetlen különbség az, hogy a pixeleket nem a képpufferbe, hanem a textúramemóriába másolja.

A *target* paraméternek a GL_TEXTURE_3D értéket kell adni. A *level* paraméter jelentése megegyezik az előző parancsoknál leírtakkal. Hatása: a megfelelő képpuffer azon téglalapjának pixeleit, melynek szélessége *width*, magassága *height*, bal alsó sarkának képmezőbeli koordinátái *x, y* bemásolja a kurrens háromdimenziós textúra *zoffset*

rétegének azon *width* szélességű, *height* magasságú téglalapjára, mely bal alsó sarkának a rétegen belüli koordinátái *xoffset*, *yoffset*.

## 13.7. A textúrák részletességének szintje (mipmapping)

A textúrákat az objektumkoordináta-rendszerben kapcsoljuk hozzá az ábrázolandó alakzatokhoz, melyek aztán számos transzformáción mennek keresztül. Előfordulhat, hogy egy-egy poligon képe csak egyetlen pixel lesz. Ennek megfelelően a poligonra feszített textúra is egyetlen ponttá zsugorodik, amit a rendszernek az eredeti méretű, pl. $128 \times 128$ -as textúrából kell kiszámítania. Ilyen esetekben gyakran előfordul, hogy a környezetétől nagyon elütő lesz az eredmény, pl. csillogó-villogó pixelek jönnek elő. Ezen probléma kiküszöbölése érdekében a textúrából egyre csökkenő felbontású családot hoznak létre, egy textúra gúlát (ha a csökkenő felbontású textúrákat - téglalapokat - egymás fölé helyezzük) és az ábrázolt poligon csökkenő méretének megfelelő tagját használja a rendszer. Ezt az eljárást nevezzük mipmapping-nek, ahol a mip a multum in parvo (sok dolog kis helyen) latin kifejezés rövidítése. Ennek használatához a textúráról minden lehetséges kicsinyítést el kell készíteni, vagyis pl. egy $64 \times 16$ -os textúráról: $32 \times 8$, $16 \times 4$, $8 \times 2$, $4 \times 1$, $2 \times 1$, $1 \times 1$. Az egyes kicsinyített textúrák mindig az eggyel nagyobb felbontásból készülnek, így az ugrásszerű változás kiküszöbölhető. Ezeket a kicsinyített textúrákat is a **glTexImage2D**() függvénnyel lehet megadni, az alapfelbontás (a legnagyobb) szintje (*level*) legyen 0, és a felbontás csökkenésével a *level* értéke nő.

### 13.7.1. Automatikus létrehozás

Ha a 0-s szintet (a legnagyobb felbontást) már létrehoztuk, akkor az alábbi GLU függvényekkel automatikusan létrehozhatjuk az összes többit.

int **gluBuild1DMipmaps** (GLenum *target*, GLint *internalFormat*, GLint *width*, GLenum *format*, GLenum *type*, void **texels*);

int **gluBuild2DMipmaps** (GLenum *target*, GLint *internalFormat*, GLint *width*, GLint *height*, GLenum *format*, GLenum *type*, void **texels*);

int **gluBuild3DMipmaps** (GLenum *target*, GLint *internalFormat*, GLint *width*, GLint *height*, GLint *depth*, GLenum *format*, GLenum *type*, void **texels*);

Létrehozza a textúra összes szükséges alacsonyabb felbontású változatát és a **glTexImage*D**() felhasználásával betölti őket. A *target*, *internalFormat*, *width*, *height*, *format*, *type* és *texels* paraméterekre ugyanaz vonatkozik, mint a megfelelő **glTexImage*D**() függvénynél. A függvények a 0 értéket adják vissza, ha minden felbontást sikeresen létrehoztak, egyébként pedig egy GLU hibaüzenetet.

Arra is lehetőség van, hogy ne az összes lehetséges, hanem csak a megadott szintű felbontásokat hozzuk automatikusan létre.

int **gluBuild1DMipmapLevels** (GLenum *target*, GLint *internalFormat*, GLint *width*, GLenum *format*, GLenum *type*, GLint *level*, GLint *base*, GLint *max*, void **texels*);

int **gluBuild2DMipmapLevels** (GLenum *target*, GLint *internalFormat*, GLint *width*, GLint *height*, GLenum *format*, GLenum *type*, GLint *level*, GLint *base*, GLint *max*, void **texels*);

int **gluBuild3DMipmapLevels** (GLenum *target*, GLint *internalFormat*, GLint *width*, GLint *height*, GLint *depth*, GLenum *format*, GLenum *type*, GLint *level*, GLint *base*, GLint *max*, void **texels*);

A *level* szintű *texels* textúrából a *base* és *max* közötti texeleket hozza létre, és a **glTexImage*D**() függvénnyel betölti azokat. A többi paraméter értelmezése ugyanaz, mint a megfelelő **glTexImage*D**() függvénynél. A függvények a 0 értéket adják vissza, ha minden felbontást sikeresen létrehoztak, egyébként pedig egy GLU hibaüzenetet.

## 13.8. Szűrők

A textúrák téglalap alakúak, melyeket tetszőleges alakú poligonokra helyezünk rá a modelltérben, és a poligonnal együtt transzformálja a rendszer a képernyő koordinátarendszerébe. Ezen transzformációk eredményeként ritkán fordul elő, hogy egy pixelen pontosan egy texel látszik, sokkal gyakoribb, hogy egy pixelen több texel látszik vagy megfordítva, egy texel több pixelen. Az utóbbi két esetben nem egyértelmű, hogy mely texeleket használja a rendszer, és hogy a texeleket hogyan átlagolja vagy interpolálja. Erre a célra az OpenGL több, un. szűrőt biztosít, melyekből a felhasználó választhat. A választás az egymásnak ellentmondó minőség és sebesség közötti kompromisszum megkötését jelenti. Külön lehet szűrőt választani a texelek kicsinyítéséhez és nagyításához. Sok esetben nem kicsinyítésről vagy nagyításról van szó, hiszen pl. előfordulhat, hogy az egyik tengelyirányba kicsinyíteni, a másikba nagyítani kellene, azaz skálázásnak (affin transzformációnak) kellene alávetni az adott texeleket. Ilyen esetekben az OpenGL dönti el, hogy kicsinyít-e vagy nagyít, miután skálázni nem tudja a texeleket. Legjobb azonban az ilyen helyzetek elkerülése.

A texelek kicsinyítésére, illetve nagyítására használt szűrőket a **glTexParameter***() paranccsal választhatjuk ki. Ez egy általánosabb célú függvény, további alkalmazása a 13.11.2. pontban található.

void **glTexParameter**{if} (GLenum *target*, GLenum *pname*, TYPE *param*);

void **glTexParameter**{if}**v** (GLenum *target*, GLenum *pname*, TYPE **param*);

A *target* paraméternek a GL_TEXTURE_1D, GL_TEXTURE_2D vagy GL_TEXTURE_3D értéket adjuk attól függően, hogy hány dimenziós textúráról van szó. A *pname* paraméternek a GL_TEXTURE_MAG_FILTER a nagyításhoz használandó szűrő beállításához és a GL_TEXTURE_MIN_FILTER értéket a kicsinyítéshez. *param* lehetséges értékeit a 13.1. táblázat tartalmazza.

Ha a *param* = GL_NEAREST, akkor mind nagyítás, mind kicsinyítés esetén a pixel középpontjához legközelebbi texelt használja a rendszer. Ez gyakran a környezetétől

13.1. táblázat. A textúrák kicsinyítéséhez és nagyításához megadható szűrők

| *target* | *param* |
|---|---|
| GL_TEXTURE_MAG_FILTER | GL_NEAREST, GL_LINEAR |
| GL_TEXTURE_MIN_FILTER | GL_NEAREST, GL_LINEAR<br>GL_NEAREST_MIPMAP_NEAREST<br>GL_NEAREST_MIPMAP_LINEAR<br>GL_LINEAR_MIPMAP_NEAREST<br>GL_LINEAR_MIPMAP_LINEAR |

nagyon eltérő pixelt eredményez.

*param* = GL_LINEAR esetén a textúra dimenziójának megfelelően a pixel középpontjához legközelebbi 2, $2 \times 2$ vagy $2 \times 2 \times 2$ pixelt lineárisan interpolálja a rendszer mind kicsinyítésnél, mind nagyításnál. Előfordulhat, hogy az így kiválasztott texelek túlnyúlnak a textúra határán. Ilyen esetekben az eredmény két tényezőtől függ: egyrészt attól, hogy a textúrának van-e határa, másrészt a textúra ,,ismétlésének" beállításától (GL_REPEAT, GL_CLAMP, GL_CLAMP_TO_EDGE lásd a 13.11.2. pontot). Nagyítás esetén mindig az alapfelbontású textúrát használja a rendszer. Kicsinyítés esetén választhatunk, hogy az alapfelbontásút (GL_NEAREST, GL_LINEAR), vagy a legmegfelelőbb egy vagy két felbontást használja a rendszer (a többi opció).

Ha *param* = GL_NEAREST_MIPMAP_NEAREST, akkor a kicsinyítés mértékének megfelelő, legközelebb álló felbontásból veszi a rendszer a pixel középpontjához legközelebbi texelt, *param* = GL_LINEAR_MIPMAP_NEAREST esetén pedig a legközelebbi felbontásban lineáris interpolációval számítja ki.

Ha *param* = GL_NEAREST_MIPMAP_LINEAR, akkor a két legközelebbi felbontásban veszi a pixel középpontjához legközelebbi texeleket és ezeket lineárisan interpolálja, míg *param* = GL_LINEAR_MIPMAP_LINEAR esetén a két legközelebbi felbontásban külön-külön lineárisan interpolálja a pixel középpontjához legközelebbi pixeleket, majd az eredményeket újra lineárisan interpolálja. A lineáris interpoláció mindig lassabb, de jobb eredményt adó megoldás.

Ezen szűrők némelyikének más elnevezése is van. A GL_NEAREST opciót szokás pont-mintavételezésnek is nevezni, a GL_LINEAR-t bilineáris mintavételezésnek, a GL_LINEAR_MIPMAP_LINEAR-t pedig trilineáris mintavételezésnek.

Ha a mipmapre támaszkodó szűrőt akarunk alkalmazni, de nem hoztuk létre az összes lehetséges felbontását a textúrának (a GL_TEXTURE_BASE_LEVEL-től a GL_TEXTURE_MAX_LEVEL-ig), akkor az OpenGL egyszerűen nem fog textúrázni, de nem küld hibaüzenetet.

## 13.9. Textúraobjektumok

A textúraobjektumok biztosítják a textúraadatok tárolását és a tárolt adatok gyors elérését. Használatuk nagymértékben felgyorsítja a textúrák használatát, mivel egy már létrehozott textúra aktivizálása mindig gyorsabb, mint egy új létrehozása. A textúraobjektumokat az OpenGL 1.1 verziójában vezették be. Az implementációk egy

része lehetővé teszi, hogy textúraobjektumokból un. munkacsoportot hozzunk létre, mely tagjai a leghatékonyabban működnek. Ha ez a lehetőség rendelkezésünkre áll, célszerű a leggyakrabban használt textúraobjektumainkat a munkacsoportba tenni. A textúraobjektomok használatához a következő lépéseket kell tenni:

- Textúraobjektum neveket kell létrehozni.
- Létre kell hozni a textúraobjektomokat, azaz a névhez hozzá kell rendelni a textúraadatokat, és be kell állítani a tulajdonságait.
- Ha az implementációnk támogatja a munkacsoportot, meg kell nézni, hogy van-e elegendő hely az összes textúraobjektum tárolására. Ha nincs, akkor az egyes textúrákhoz prioritást kell rendelni, hogy a leggyakrabban használt textúraobjektumok kerüljenek a munkacsoportba.
- A megfelelő textúraobjektumot tegyük aktívvá, hogy a megjelenítés számára elérhetők legyenek.

### 13.9.1. A textúraobjektumok elnevezése

A textúrákhoz pozitív egész nevek (azonosítók) rendelhetők. A véletlen névütközések elkerülése érdekében célszerű a **glGenTextures**() függvényt használni.

void **glGenTextures** (GLsizei *n*, GLuint **textureNames*);

*n* darab, jelenleg használaton kívüli textúraobjektum nevet ad vissza a *textureNames* tömbben. A visszaadott nevek nem feltétlenül egymást követő pozitív egészek. A 0 a rendszer által lefoglalt textúraazonosító, ezt a **glGenTextures**() függvény soha sem adja vissza.

GLboolean **glIsTexture** (GLuint *textureName*);

A GL_TRUE értéket adja vissza, ha a *textureName* egy létező textúraobjektum azonosítója, és GL_FALSE értéket, ha nincs ilyen nevű textúraobjektum vagy *textureName* = 0.

### 13.9.2. Textúraobjektumok létrehozása, használata

void **glBindTexture** (GLenum *target*, GLuint *textureName*);

A *target* paraméter értéke GL_TEXTURE_1D, GL_TEXTURE_2D vagy GL_TEXTURE_3D lehet. Amikor először hívjuk meg egy pozitív *textureName* azonosítóval, akkor létrehoz egy textúraobjektumot és ezt hozzákapcsolja a névhez. Az így létrehozott textúraobjektum a textúraadatokra és textúratulajdonságokra az alapértelmezett beállításokat tartalmazza, ezeket később a **glTexImage***(), **glTexSubImage***(), **glCopyTexImage***(), **glCopyTexSubImage***(), **glTexParameter***() és

**glPrioritizeTextures**() függvények meghívásával tölthetjük fel a használandó textúra jellemzőivel.

Ha nem először hívjuk meg a *textureName* azonosítóval, akkor a *textureName* azonosítójú textúraobjektumot teszi kurrenssé (a textúraleképezés számára elérhetővé) a *target*-nek megfelelő dimenziójú textúraobjektumok közül.

Ha *textureName* = 0 értékkel hívjuk meg, akkor az OpenGL kikapcsolja a textúraobjektumok használatát, és a névnélküli alapértelmezett textúrát teszi kurrenssé.

### 13.9.3. Textúraobjektumok törlése

A nem kurrens textúraobjektumok is foglalják a helyet a textúramemóriában. Ahhoz, hogy helyet szabadítsunk fel, törölni kell az objektumot.

void **glDeleteTextures** (GLsizei *n*, const GLuint **textureNames*);

A *textureNames* tömbben tárolt azonosítójú textúraobjektumokat törli, azaz felszabadítja az adatok, paraméterek számára lefoglalt helyet a memóriában és az azonosítók használaton kívüli státuszt nyernek. Nem létező textúraobjektumok törlése nem eredményez hibát, egyszerűen figyelmen kívül hagyja a rendszer, a 0 azonosítójú objektum törlése a default textúrát teszi kurrenssé, mint a *textureName* = 0 paraméterrel meghívott **glBindTexture**() függvény.

### 13.9.4. Rezidens textúrák munkacsoportja

Néhány OpenGL implementáció lehetővé teszi, hogy nagy hatékonysággal működő textúraobjektumokból álló munkacsoportot hozzunk létre. A munkacsoporthoz tartozó objektumokat rezidensnek nevezzük. Ezeknél általában speciális hardver támogatja a textúraműveleteket, és egy korlátozott kapacitású cache áll rendelkezésre a textúrák tárolásához. Ilyen speciális cache létezése esetén mindenképpen célszerű textúraobjektumokat használni.
Ha a használandó textúrákhoz szükséges hely meghaladja a cache kapacitását, akkor nem lehet minden textúránk rezidens. Ha meg akarjuk tudni, hogy egy textúraobjektumunk rezidens-e, tegyük az objektumot kurrenssé és a **glGetTexParameter*v**() parancsot a GL_TEXTURE_RESIDENT paraméterrel kiadva lekérdezhetjük a státuszát. Ha több objektum státuszára vagyunk kíváncsiak, akkor használjuk a **glAreTexturesResident**() függvényt.

GLboolean **glAreTexturesResident** (GLsizei *n*, const GLuint **textureNames*, GLboolean **residences*);

A *textureNames* tömbben megadott azonosítójú *n* darab textúraobjektum státuszát kérdezhetjük le. Az objektumok státuszát a *residences* tömbben adja vissza. Ha a megadott *n* darab objektum mindegyike rezidens a függvény a GL_TRUE értéket adja vissza és a *residences* tömb elemeit nem változtatja meg. Ha az adott objektumok közül legalább egy nem rezidens, akkor a függvény a GL_FALSE értéket adja vissza és a *residences* tömbnek a nem rezidens objektumokhoz tartozó elemeit a GL_FALSE értékre állítja.

Ez a függvény a textúraobjektumok pillanatnyi állapotát adja vissza, ami általában nagyon gyorsan változik. Ha az implementáció nem támogatja a textúraobjektumok munkacsoportját, akkor a rendszer minden objektumot rezidensnek tekint és a **glAreTexturesResident**() függvény mindig a GL_TRUE értéket adja vissza.

### 13.9.5. Textúraobjektumok prioritása

Ha azt akarjuk, hogy programunk a lehető legnagyobb hatékonysággal működjön mindenkor, pl. játékprogramok esetén, akkor csak rezidens textúraobjektumot használjunk textúrázásra (ha a rendszer támogatja a textúraobjektumok munkacsoportját). Ha nincs elegendő textúramemória az összes textúra befogadására, akkor csökkenthetjük a textúrák méretét, a felbontások (mipmap) számát, vagy a **glTexSubImage***() függvénnyel a textúrák tartalmát folytonosan változtatva ugyanazt a textúramemóriát használjuk. (Ez utóbbi mindig hatékonyabb, mint textúrák törlése, majd új létrehozása.)

Az olyan alkalmazásoknál, ahol a textúrákat menetközben hozzák létre, elkerülhetetlen a nem rezidens textúraobjektumok használata is. Ilyen esetben a gyakrabban használt textúraobjektumokhoz célszerű magasabb prioritást rendelni, mint azokhoz amelyeket valószínűleg ritkábban használunk.

void **glPrioritizeTextures** (GLsizei *n*, const GLuint **textureNames*, const GLclampf **priorities*);

A *textureNames* tömbben megadott azonosítójú *n* darab textúraobjektumhoz a *priorities* tömbben adott prioritásokat rendeli. A prioritásokat a rendszer a $[0., 1.]$ intervallumra levágja. A 0 érték a legalacsonyabb prioritást rendeli, az ilyen objektumnak a legkisebb az esélye arra, hogy rezidens maradjon az erőforrások csökkenése esetén, az 1 prioritásúé a legnagyobb. A prioritás hozzákapcsolásához nem szükséges, hogy az objektum létezzen, de a prioritásnak csak létező objektumokra van hatása. A **glTexParameter***() is használható a kurrens objektum prioritásának beállítására, a default textúra prioritása csak ezzel állítható be.

## 13.10. Textúrafüggvények

A textúrák nemcsak arra használhatók, hogy a textúra színeit ráfessük az ábrázolt alakzatra, hanem használhatók az ábrázolt alakzat színének módosítására, vagy kombinálhatók az alakzat eredeti színével. A **glTexEnv***() függvénnyel írhatjuk elő, hogy a rendszer hogyan használja a textúrát.

void **glTexEnv**{if} (GLenum *target*, GLenum *pname*, TYPE *param*);

void **glTexEnv**{if}**v** (GLenum *target*, GLenum *pname*, TYPE **param*);

A *target* paraméternek a GL_TEXTURE_ENV értéket kell adni.

Ha *pname* = GL_TEXTURE_ENV_MODE, akkor a *param* értéke GL_DECAL, GL_REPLACE, GL_MODULATE vagy GL_BLEND lehet, mely értékek azt írják elő, hogy a rendszer a textúrát hogyan kombinálja a feldolgozandó fragmentum színével.

Ha *pname* = GL_TEXTURE_ENV_COLOR, akkor *param* az R, G, B, A komponenseket tartalmazó tömb címe. Ezeket az értékeket csak akkor használja a rendszer, ha a GL_BLEND textúrafüggvény is definiált.

A textúrafüggvény és a textúra belső formátumának (ezt a **glTexImage***() függvénnyel adjuk meg) kombinációja határozza meg, hogy a textúra egyes komponenseit hogyan használja a rendszer. A textúrafüggvény csak a textúra kiválasztott komponenseit és az ábrázolandó alakzat textúrázás nélküli színét veszi figyelembe.

A 13.2. táblázat a GL_REPLACE és GL_MODULATE , a 13.3. táblázat pedig a GL_DECAL és GL_BLEND hatását mutatja. Az $f$ alsó indexű elemek a fragmentumra, a $t$ a textúrára, a $c$ pedig a GL_TEXTURE_ENV_COLOR segítségével megadott színre vonatkoznak. Az index nélküli mennyiségek az eredményül kapott színt jelentik, $E$ pedig a csupa 1 komponensű vektort.

13.2. táblázat. A GL_REPLACE és GL_MODULATE függvényei

| belső formátum | GL_REPLACE | GL_MODULATE |
|---|---|---|
| GL_ALPHA | $C = C_f$<br>$A = A_t$ | $C = C_f$<br>$A = A_f A_t$ |
| GL_LUMINANCE | $C = L_t$<br>$A = A_f$ | $C = C_f * L_t$<br>$A = A_f$ |
| GL_LUMINANCE_ALPHA | $C = L_t$<br>$A = A_t$ | $C = C_f * L_t$<br>$A = A_f A_t$ |
| GL_INTENSITY | $C = I_t$<br>$A = A_t$ | $C = C_f * I_t$<br>$A = A_f I_t$ |
| GL_RGB | $C = C_t$<br>$A = A_f$ | $C = C_f * C_t$<br>$A = A_f$ |
| GL_RGBA | $C = C_t$<br>$A = A_t$ | $C = C_f * C_t$<br>$A = A_f A_t$ |

GL_REPLACE esetén a fragmentum színét felülírja a textúra színével. A GL_DECAL opció hatása nagyon hasonlít az előzőre, a különbség az, hogy az utóbbi csak RGB és RGBA belső formátum esetén működik és az alfa komponenst másként dolgozza fel. RGBA belső formátum esetén a textúra alfa komponensének megfelelően keveri a fragmentum színét a textúra színével, és a fragmentum alfa komponensét változatlanul hagyja.

Alapértelmezés szerint a textúraműveleteket a rendszer a megvilágítási számítások után hajtja végre. A tükröződő fényes foltok hatását csökkenti, ha azt textúrával keverjük. Lehetőség van arra, hogy a tükrözött fénykomponens hatását a textúrázás után vegye figyelembe a rendszer. Ez az eljárás sokkal hangsúlyosabbá teszi a textúrázott felületek megvilágítását.

## 13.11. Textúrakoordináták megadása

Textúrázott felületek megadásakor a csúcspontok geometriai koordinátái mellett meg kell adni a csúcspontok textúrakoordinátáit is. A textúrakoordináták azt adják meg, hogy

13.3. táblázat. A GL_DECAL és GL_BLEND függvényei

| belső formátum | GL_DECAL | GL_BLEND |
|---|---|---|
| GL_ALPHA | definiálatlan | $C = C_f$<br>$A = A_f A_t$ |
| GL_LUMINANCE | definiálatlan | $C = C_f * (E - L_t) + C_c * L_t$<br>$A = A_f$ |
| GL_LUMINANCE_ALPHA | definiálatlan | $C = C_f * (E - L_t) + C_c * L_t$<br>$A = A_f A_t$ |
| GL_INTENSITY | definiálatlan | $C = C_f * (E - I_t) + C_c * I_t$<br>$A = A_f (1 - I_t) + A_c I_t$ |
| GL_RGB | $C = C_t$<br>$A = A_f$ | $C = C_f * (E - C_t) + C_c * C_t$<br>$A = A_f$ |
| GL_RGBA | $C = C_f (1 - A_t) + C_t A_t$<br>$A = A_f$ | $C = C_f * (E - C_t) + C_c * C_t$<br>$A = A_f A_t$ |

mely texel tartozik a csúcsponthoz, és a rendszer a csúcspontokhoz tartozó texeleket interpolálja.
A textúrának lehet $1, 2, 3$ vagy $4$ koordinátája. Ezeket $s, t, r$ és $q$ –val jelöljük, megkülönböztetésül a geometriai koordinátákhoz használt *x, y, z* és *w*, valamint a felületek paraméterezéséhez használt *u, v* jelölésektől. A $q$ koordináta értéke általában 1, és ez homogén textúrakoordináták létrehozását teszi lehetővé. A textúrakoordinátákat is a **glBegin**(), **glEnd**() függvénypár között kell megadni, mint a csúcspontokat. Ezek a koordináták általában a $[0., 1.]$ intervallumon változnak, de megadhatunk ezen kívül eső értékeket is, melyek értelmezését előírhatjuk (lásd a 13.11.2. pontot).

void **glTexCoord**{1234}{sifd} (TYPE *coords*);

void **glTexCoord**{1234}{sifd}**v** (TYPE **coords*);

A kurrens $(s, t, r, q)$ textúrakoordinátákat állítja be. Az ezt követően létrehozott csúcspon(ok)hoz a rendszer ezt a textúrakoordinátát rendeli. Ha a $t$ vagy $r$ koordinátát nem adjuk, meg akkor azokat a rendszer 0-nak tekinti, a hiányzó $q$-t pedig 1-nek. A megadott textúrakoordinátákat a rendszer megszorozza a $4 \times 4$ -es textúramátrixszal, mielőtt feldolgozná őket.

### 13.11.1. A megfelelő textúrakoordináták kiszámolása

Az ábrázolandó poligonok, poligonhálók csúcspontjaihoz hozzárendeljük a textúrakoordinátákat. Ez a hozzárendelés történhet úgy, hogy a poligonokra (síkdarabokra) rátesszük a szintén síkdarabot reprezentáló textúrát. Síkbateríthető felületek esetén (kúp, henger általános érintőfelület) ezt torzításmentesen megtehetjük, minden más esetben azonban torzítás lép fel. Például gömb esetén a textúraleképezés történhet a gömb

$$
\begin{aligned}
x(\phi,\varphi) &= r\cos\varphi\cos\phi,\\
y(\phi,\varphi) &= r\sin\varphi\cos\phi,\\
z(\phi,\varphi) &= r\sin\phi,\\
&\varphi\in[0.,2\pi]\,,\phi\in[0.,2\pi]
\end{aligned}
$$

paraméteres alakjából kiindulva úgy, hogy az értelmezési tartomány $[0.,2\pi]\times[0.,2\pi]$ négyzetét képezzük le a textúra téglalapjára. ez a megoldás jelentős torzítást eredményez a gömb északi és déli pólusa (a $(0.,0.,r)$ és $(0.,0.,-r)$ ) felé haladva. Szabad formájú, pl. NURBS felület esetén a felület $u$, $v$ paraméterei használhatók a textúraleképezéshez, azonban a poligonhálóval közelített görbült felületek esztétikus textúrázásának jelentős művészi összetevői is vannak.

### 13.11.2. Textúrák ismétlése

A $[0.,1.]$ intervallumon kívül eső textúrakoordinátákat is megadhatunk, és rendelkezhetünk arról, hogy a rendszer a textúrákat ismételje vagy levágja a koordinátákat. A textúra ismétlése azt jelenti, hogy a textúra téglalapját, mit egy csempét, a rendszer egymás mellé teszi. A megfelelő hatás elérése érdekében olyan textúrát célszerű megadni, mely jól illeszthető. A textúrakoordináták levágása azt jelenti, hogy minden 1-nél nagyobb textúrakoordinátát 1-nek, minden 0-nál kisebbet 0-nak tekint a rendszer. Ezekben az esetekben külön gondot kell fordítani a textúra határára, ha van. A GL_NEAREST szűrő esetén a rendszer a legközelebbi texelt használja, és a határt mindig figyelmen kívül hagyja.

void **glTexParameter**{if} (GLenum *target*, GLenum *pname*, TYPE *param*);

void **glTexParameter**{if}**v** (GLenum *target*, GLenum *pname*, TYPE **param*);

Azokat a paramétereket állítja be, melyek azt befolyásolják, hogy a rendszer hogyan kezeli a texeleket a textúraobjektumokba való tárolás, illetve a fragmentumokra való alkalmazás során. A *target* paramétert a textúra dimenziójának megfelelően GL_TEXTURE_1D, GL_TEXTURE_2D vagy GL_TEXTURE_3D értékre kell állítani. A *pname* paraméter lehetséges értékeit a 13.4. táblázat tartalmazza, a hozzájuk kapcsolható értékekkel együtt.

### 13.11.3. Textúrakoordináták automatikus létrehozása

A textúrázással felületek szintvonalainak ábrázolása, valamint olyan hatás is elérhető, mintha a felületben tükröződne a környezete. Ezen hatások elérése érdekében az OpenGL automatikusan létrehozza a textúrakoordinátákat, vagyis nem kell azokat csúcspontonként megadni.

13.4. táblázat. Textúrák ismétlése

| *pname* | *pname* |
|---|---|
| GL_TEXTURE_WRAP_S | GL_CLAMP, GL_CLAMP_TO_EDGE, GL_REPEAT |
| GL_TEXTURE_WRAP_T | GL_CLAMP, GL_CLAMP_TO_EDGE, GL_REPEAT |
| GL_TEXTURE_WRAP_R | GL_CLAMP, GL_CLAMP_TO_EDGE, GL_REPEAT |
| GL_TEXTURE_MAG_FILTER | GL_NEAREST, GL_LINEAR |
| GL_TEXTURE_MIN_FILTER | GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR |
| GL_TEXTURE_BORDER_COLOR | tetszőleges számnégyes $[0., 1.]$ -ből |
| GL_TEXTURE_PRIORITY | egy $[0., 1.]$ intervallumbeli érték |
| GL_TEXTURE_MIN_LOD | tetszőleges lebegőpontos szám |
| GL_TEXTURE_MAX_LOD | tetszőleges lebegőpontos szám |
| GL_TEXTURE_BASE_LEVEL | tetszőleges nemnegatív egész |
| GL_TEXTURE_MAX_LEVEL | tetszőleges nemnegatív egész |

void **glTexGen**{ifd} (GLenum *coord*, GLenum *pname*, TYPE *param*);

void **glTexGen**{ifd}**v** (GLenum *coord*, GLenum *pname*, TYPE **param*);

Textúrakoordináták automatikus létrehozásához lehet vele függvényeket megadni.

A *coord* paraméter értékei GL_S, GL_T, GL_R vagy GL_Q lehet attól függően, hogy az s, t, r vagy q koordináta létrehozásáról van szó.

A *pname* értéke GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE vagy GL_EYE_PLANE lehet. *pname* = GL_TEXTURE_GEN_MODE esetén *param* értéke GL_OBJECT_LINEAR, GL_EYE_LINEAR vagy GL_SPHERE_MAP lehet, melyekkel a textúrát leképező függvényt adhatjuk meg. *pname* másik két lehetséges értékénél a leképező függvényhez adhatunk meg paramétereket.

A referenciasíkot akkor célszerű objektumkoordinátákban megadni (GL_OBJECT_LINEAR), ha a textúrázott kép rögzített marad egy mozgó objektumhoz képest. Mozgó objektumok szintvonalainak ábrázolásához célszerű a nézőpontkoordináta-rendszerben (GL_EYE_LINEAR) megadni a referenciasíkot. A GL_SPHERE_MAP leképezést elsősorban a környezet tükröződésének szimulációjakor használjuk.

## 13.12. Textúramátrix-verem

A rendszer a textúrakoordinátákat a $4 \times 4$ -es textúramátrixszal megszorozza, mielőtt bármilyen más feldolgozásnak alávetné. Alapértelmezés szerint a textúramátrix az egységmátrix, azaz nincs semmilyen hatása. A textúrázott objektum újrarajzolása előtt a textúramátrixot megváltoztatva speciális hatások érhetők el, pl. a textúra elmozdítható, nyújtható, zsugorítható, forgatható. Miután a mátrix $4 \times 4$ -es, tetszőleges projektív transzformáció írható le vele. A rendszer egy veremszerkezetet tart karban a textúramátrixok tárolására, mely verem legalább 2 szintű. A textúramátrix megadása, a verem manipulálása előtt a **glMatrixMode**(GL_TEXTURE) parancsot kell kiadni, és ez után a mátrixműveletek és a veremműveletek a textúramátrixra hatnak.

# 14. fejezet

# Görbék és felületek rajzolása

A geometriai alapelemekkel csak pont, töröttvonal és poligon rajzolható, azonban természetesen igény van görbék és görbült felületek megjelenítésére is. A grafikus rendszerekben a görbéket közelítő (a görbén valamilyen sűrűséggel felvett pontokat összekötő) töröttvonallal, a felületet pedig közelítő (a felületen valamilyen sűrűséggel kiszámított pontokkal meghatározott) poligonokkal (poligonhálóval) szokás megjeleníteni. Ha a közelítő töröttvonal , illetve poligon oldalai elég rövidek, a képen nem láthatók az oldalak (lapok) törései, következésképpen sima görbe, illetve felület benyomását keltik vizuálisan. A görbék, illetve felületek ilyen típusú megjelenítését paraméteres formában leírt alakzatokon célszerű alkalmazni.

Az OpenGL a Bézier-görbék és felületek megjelenítését támogatja közvetlenül, azonban a Bézier-alakzatokkal sok más, jól ismert görbe és felület is egzaktul leírható, így pl. az Hermite-ív, Ferguson-spline, B-spline.

## 14.1. Bézier-görbe megjelenítése

Az $n$-edfokú Bézier-görbe

$$\mathbf{b}(u) = \sum_{i=0}^{n} B_i^n(u)\,\mathbf{b}_i, u \in [u_1, u_2], \tag{14.1}$$

$$B_i^n(u) = \left( \begin{array}{c} n \\ i \end{array} \right) u^i (1-u)^{n-i} = \frac{n!}{i!\,(n-i)!} u^i (1-u)^{n-i} \tag{14.2}$$

alakban írható fel. A $\mathbf{b}_i$ pontokat kontroll, vagy Bézier pontoknak nevezzük, $B_i^n(u)$ pedig az $i$-edik $n$-edfokú Bernstein-polinomot jelöli. Általában $u_1 = 0$, $u_2 = 1$. A (14.1) kifejezés olyan görbét ír le, melynek a kezdőpontja $\mathbf{b}_0$, végpontja $\mathbf{b}_n$ és a többi megadott ponton általában nem megy át (kivéve, ha a pontok egy egyenesre illeszkednek).

Az OpenGL-ben a következőképpen tudunk Bézier-görbét megjeleníteni:

- Definiálunk egy úgynevezett egydimenziós kiértékelést a **glMap1***() paranccsal, azaz megadjuk a Bézier-görbe rendjét, paramétertartományát, kontrollpontjait, valamint azt, hogy a kontrollpontok mit reprezentálnak, pl. a modelltérbeli pontot vagy színt.

- Engedélyezzük a megfelelő objektum kiértékelését **glEnable**().

- A megadott leképezést kiértékeljük a kívánt paraméterértékeknél, azaz kiszámítjuk a helyettesítési értéket **glEvalCoord1***().

void **glMap1**{fd} (GLenum *target*, TYPE *u1*, TYPE *u2*, GLint *stride*, GLint *order*, const TYPE **points*);

Bézier-görbét leíró, úgynevezett egydimenziós kiértékelőt definiál. A *target* paraméterrel kell megadnunk, hogy a pontok mit reprezentálnak, amiből az is következik, hogy a *point* tömbben hány adatot kell átadni. A lehetőségek teljes listáját a 14.1. táblázat tartalmazza. A pontok reprezentálhatnak, csúcspontokat, RGBA vagy színindex adatokat, normálvektorokat, vagy textúrakoordinátákat. A számítást végző parancs kiadása előtt ugyanezzel a paraméterrel kell engedélyezni (**glEnable**())a kiértékelést.

*u1* és *u2* a Bézier-görbe paramétertartományának határait jelöli, a *stride* paraméterrel pedig azt adjuk meg, hogy két egymást követő pont között hány float vagy double típusú változó van. Az *order* paraméter a görbe rendje (fokszám + 1), aminek meg kell egyeznie a pontok számával. A *points* paraméter az első pont első koordinátájának a címe.

Több kiértékelőt is definiálhatunk, de adattípusonként csak egyet engedélyezzünk, tehát csak egyfajta csúcspont-, szín- vagy textúra-kiértékelést engedélyezzünk. Ha ugyanabból a típusból több is engedélyezett, a rendszer a legtöbb komponenssel megadhatót veszi figyelembe.

14.1. táblázat. Az **glMap1***() parancs *target* paraméterének értékei és jelentésük

| *target* | jelentése |
|---|---|
| GL_MAP1_VERTEX_3 | csúcspontok $x, y, z$ koordinátái |
| GL_MAP1_VERTEX_4 | csúcspontok $x, y, z, w$ koordinátái |
| GL_MAP1_INDEX | színindex |
| GL_MAP1_COLOR_4 | R, G, B, A színkomponensek |
| GL_MAP1_NORMAL | normális koordinátái |
| GL_MAP1_TEXTURE_COORD_1 | $s$ textúrakoordináták |
| GL_MAP1_TEXTURE_COORD_2 | $s, t$ textúrakoordináták |
| GL_MAP1_TEXTURE_COORD_3 | $s, t, r$ textúrakoordináták |
| GL_MAP1_TEXTURE_COORD_4 | $s, t, r, q$ textúrakoordináták |

void **glEvalCoord1**{fd} (TYPE $u$);

void **glEvalCoord1**{fd}**v** (const TYPE *$u$);

Az engedélyezett egydimenziós leképezések $u$ helyen vett helyettesítési értékét számítja ki. A **glBegin**() és **glEnd**() pár között aktivizálva a megfelelő OpenGL parancs(ok) kiadásának hatását is elérjük, tehát ha GL_MAP1_VERTEX_3 vagy GL_MAP1_VERTEX_4 engedélyezett, akkor egy csúcspontot hoz létre (mintha a

**glVertex***() parancsot adtuk volna ki); ha GL_MAP1_INDEX engedélyezett, akkor a **glIndex***() parancsot szimulálja; ha GL_MAP1_COLOR_4, akkor a **glColor***(); ha GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, akkor a **glTexCoord***() parancsot, ha pedig a GL_MAP1_NORMAL, akkor a **glNormal***() parancsot.

Az így létrehozott csúcspontokhoz a kiszámított színt, színindexet, normálist és textúrakoordinátát rendeli a rendszer, amennyiben azok engedélyezettek, a nem engedélyezettekhez pedig a rendszer által tárolt kurrens értéket. Az így kiszámított értékekkel azonban nem írja felül a kurrens értékeket.

A **glEvalCoord1***() paranccsal egyetlen görbepontot számíthatunk ki. A rendszer lehetővé teszi azt is, hogy az $[u_1, u_2]$ értelmezési tartományt $n$ egyenlő részre osztva a paramétertartományban rácspontokat hozzunk létre, majd egyetlen függvényhívással kiszámítsuk a rácspontokhoz tartozó görbepontokat.

void **glMapGrid1**{fd} (GLint *n*, TYPE *u1*, TYPE *u2*);

Az $[u1, u2]$ intervallumot $n$ egyenlő részre osztva egy $n+1$ darab rácspontot hoz létre az értelmezési tartományban.

void **glEvalMesh1** (GLenum *mode*, GLint *p1*, GLint *p2*);

Minden engedélyezett kiértékelővel kiszámolja a kurrens rácspontok *p1*-től *p2*-ig terjedő intervallumához tartozó értékeket ( $0 \leq p1 \leq p2 \leq n$ , ahol $n$ a **glMapGrid1***() paranccsal megadott osztások száma). A *mode* paraméter a GL_POINT vagy GL_LINE értékeket veheti fel, melynek megfelelően a görbén kiszámított pontokat pont alapelemmel jelöli meg, vagy töröttvonallal köti össze. A

```
glMapGrid1f(n,u1,u2);
glEvalMesh1(mode,p1,p2);
```

parancsok hatása, a kerekítési hibából adódó eltéréstől eltekintve, megegyezik a következő programrészletével:

```
glBegin(mode);
   for(i = p1; i <= p2; i++)
     glEvalCoord1f(u1 + i* (u2 - u1) / n);
glEnd();
```

Görbék megrajzolása nem egyszerű feladat, ugyanis igaz, hogy ha az oldalak nagyon rövidek (pl. 1 pixel hosszúak), akkor biztosan jó vizuális hatást érünk el. Az esetek többségében azonban az összes oldal hosszának csökkentése pazarló (túl sok szakaszt tartalmazó) közelítést eredményez. Olyan görbék esetén, melyek "majdnem" egyenes szakaszt és "nagyon görbült" részeket is tartalmaznak, az oldalhosszakat addig kell csökkenteni, míg a nagyon görbül rész is megfelelő lesz a képen, de ez már biztosan túl sok részre bontja a majdnem egyenes részt. A feladat megoldásánál tehát figyelembe kell venni a görbe görbületének változását is, vagyis az a gyakran használt durva megoldás, miszerint: "osszuk fel a paramétertartományt $n$ egyenlő részre, és az ezekhez tartozó görbepontokat kössük össze egyenes szakaszokkal, így $n$ növelésével előbb-utóbb esztétikus képet kapunk", csak konstans görbületű görbék esetén eredményezhet optimális megoldást. Ilyen görbe a síkon, mint tudjuk, csak az egyenes és a kör. A görbület változásához alkalmazkodó felosztási algoritmust nem könnyű találni tetszőleges görbéhez, de a Bézier-görbék

esetén a de Casteljau-algoritmus biztosítja ezt. A fenti szempontokat mindenképpen célszerű figyelembe venni, ha saját felbontási algoritmust akarunk kitalálni görbék rajzolására. A felületek poligonhálóval való közelítése még összetettebb feladat. Ezért azt tanácsoljuk, hogy ha csak lehet, használjuk a GLU, vagy a GLUJ függvénykönyvtárak NURBS objektumok megjelenítésére felkínált lehetőségeit.

## 14.2. Bézier-felület megjelenítése

A Bézier-felület paraméteres alakja

$$\mathbf{s}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)\, B_j^m(v)\, \mathbf{b}_{ij}, u \in [u_1,u_2], v \in [v_1,v_2],$$

ahol $B_i^n(u)$ és $B_j^m(v)$ értelmezése (14.2) szerinti. A felület átmegy a $\mathbf{b}_{00}, \mathbf{b}_{0m}, \mathbf{b}_{n0}, \mathbf{b}_{nm}$ kontrollpontokon, a többi ponton azonban általában nem, kivéve, ha a kontrollpontok egy síkra illeszkednek.

A Bézier felület megjelenítésekor követendő eljárás analóg a Bézier-görbe megjelenítéséhez követendővel, beleértve a parancsok nevét is.

void **glMap2**{fd} (GLenum *target*, TYPE *u1*, TYPE *u2*, GLint *ustride*, GLint *uorder*, TYPE *v1*, TYPE *v2*, GLint *vstride*, GLint *vorder*, const TYPE **points*);

Bézier-felületet leíró, úgynevezett kétdimenziós kiértékelőt definiál. A *target* paraméter a 14.2. táblázat szerinti értékeket veheti fel, vagyis a görbéhez hasonlóan azt írja elő, hogy a pontok hol adottak. A számítást végző parancs kiadása előtt ugyanezzel a paraméterrel kell engedélyezni (**glEnable**())a kiértékelést. A felület értelmezési tartománya az $(u,v)$ paramétersík $[u1,u2] \times [v1,v2]$ téglalapja. *uorder* a felület $u$, vorder pedig a $v$ irányú rendjét (fokszám + 1) jelöli. Az *ustride* és *vstride* paraméterek két egymást követő pontnak a koordináták típusában (float, double) mért távolságát adja meg. A *points* paraméter az első pont első koordinátájának a címe.

Például, ha a kontrollpontok tömbje
`GLdouble ctrlpoints[50][30][3];`
és az első kontrollpont első koordinátája a `ctrlpoints[10][10][0]`, akkor $ustride = 30{\cdot}3$, $vstride = 3$ és $points = \&(ctrlpoints[10][10][0])$.

void **glEvalCoord2**{fd} (TYPE *u*, TYPE *v*);

void **glEvalCoord2**{fd}**v** (const TYPE **u*, const TYPE **v*);

Az engedélyezett kétdimenziós leképezéseknek a paramétersík $(u,v)$ koordinátájú pontjában vett értékét számítja ki. Ha az engedélyezett leképezés csúcspontra vonatkozik - a **glMap2***() parancs *target* paramétere GL_MAP2_VERTEX_3 vagy GL_MAP2_VERTEX_4 - a rendszer automatikusan létrehozza a normálisokat is, és a csúcsponthoz kapcsolja, ha a normálisok automatikus létrehozását a **glEnable**(GL_AUTO_NORMAL) parancs kiadásával korábban engedélyeztük. Ha nem engedélyeztük a normálisok automatikus létrehozását, akkor a megfelelő kétdimenziós

14.2. táblázat. Az **glMap2***() parancs *target* paraméterének értékei

| *target* | jelentése |
|---|---|
| GL_MAP2_VERTEX_3 | csúcspontok $x, y, z$ koordinátái |
| GL_MAP2_VERTEX_4 | csúcspontok $x, y, z, w$ koordinátái |
| GL_MAP2_INDEX | színindex |
| GL_MAP2_COLOR_4 | R, G, B, A színkomponensek |
| GL_MAP2_NORMAL | normális koordinátái |
| GL_MAP2_TEXTURE_COORD_1 | $s$ textúrakoordináták |
| GL_MAP2_TEXTURE_COORD_2 | $s, t$ textúrakoordináták |
| GL_MAP2_TEXTURE_COORD_3 | $s, t, r$ textúrakoordináták |
| GL_MAP2_TEXTURE_COORD_4 | $s, t, r, q$ textúrakoordináták |

normális-leképezés definiálásával és engedélyezésével hozhatjuk létre a normálisokat. Ha az előzők egyikét sem engedélyeztük, akkor a rendszer a kurrens normálist rendeli a csúcsponthoz.

void **glMapGrid2**{fd} (GLint *nu*, TYPE *u1*, TYPE *u2*, GLint *nv*, TYPE *v1*, TYPE *v2*);

Az $[u1, u2]$ paramétertartományt $nu$ , a $[v1, v2]$ paramétertartományt $nv$ egyenlő részre osztva egy $(nu + 1) \times (nv + 1)$ pontból álló rácsot hoz létre az értelmezési tartományban.

void **glEvalMesh2** (GLenum *mode*, GLint *p1*, GLint *p2*, GLint *q1*, GLint *q2*);

A **glMapGrid2***() paranccsal létrehozott rácspontokban minden engedélyezett kiértékelővel kiszámítja a felület pontjait az $(nu + 1) \times (nv + 1)$ rácspontokban, és a *mode* paraméter GL_POINT értéke esetén ponthálóval, GL_LINE esetén poligonhálóval, GL_FILL esetén pedig kitöltött poligonhálóval jeleníti meg. A

```
glMapGrid2f(nu,u1,u2,nv,v1,v2);
glEvalMesh1(mode,p1,p2,q1,q2);
```

parancsok hatása gyakorlatilag a következő programrészletek valamelyikével egyezik meg, eltérés csak a kerekítési hibából adódhat.

```
glBegin(GL_POINTS);/* ha mode == GL_POINT */
  for(i = p1; i <= p2; i++)
    for(j == q1; j <= q2; j++)
      glEvalCoord2f(u1 + i *(u2 - u1) / nu, v1 + j *(v2 - v1) / nv);
glEnd();
for(i = p; i <= p2; i++)/* ha mode == GL_LINE */
  {
  glBegin(GL_LINES);
    for(j == q1; j <= q2; j++)
      glEvalCoord2f(u1 + i *(u2 - u1) / nu, v1 + j *(v2 - v1) / nv);
  glEnd();
```

```
    }
  for(i = p1; i <= p2; i++)/* ha mode == GL_FILL */
    {
    glBegin(GL_QUAD_STRIP);
      for(j == q1; j <= q2; j++)
        {
        glEvalCoord2f(u1 + i * (u2 - u1) / nu, v1 + j * (v2 - v1) / nv);
        glEvalCoord2f(u1 + (i + 1) * (u2 - u1) / nu, v1 + j * (v2 - v1) /
nv);
        }
    glEnd();
    }
```

## 14.3. Racionális B-spline (NURBS) görbék és felületek megjelenítése

Napjaink számítógépes rendszereinek legelterjedtebb görbe- és felületleírási módszere a racionális B-spline. Szinonimaként használatos a NURBS (Non-Uniform Rational B-Spline) elnevezés is. Nagy népszerűségüket az is indokolja, hogy sokféle alak írható le velük egzaktul, így pl. a Bézier-görbe vagy a hagyományosan használt kúpszeletek is. Jelen anyagban csak a vonatkozó OpenGL függvények megértéséhez elengedhetetlenül szükséges definíciókat adjuk meg.

Az OpenGL-ben az alábbi eljárást követve tudunk NURBS görbét vagy felületet megjeleníteni:

- Létrehozunk egy új NURBS objektumstruktúrát a **gluNewNurbsRenderer**() paranccsal. Az itt visszakapott címmel tudunk hivatkozni az objektumra a tulajdonságok beállításakor és a megjelenítéskor.
- A **gluNurbsProperty**() paranccsal beállíthatjuk az objektum megjelenését befolyásoló paramétereket, továbbá ezzel engedélyezhetjük a közelítő töröttvonal, illetve poligonháló adatainak visszanyerését.
- A **gluNurbsCallback**() paranccsal megadhatjuk azt a függvényt, amit a rendszer meghív, ha a NURBS objektum megjelenítése során hiba fordul elő, valamint megadhatjuk azt a függvényt, amivel a közelítő töröttvonal, illetve poligonháló adatait visszakapjuk.
- A görbe-, illetve felületmegadást és rajzolást a **gluBeginCurve**(), illetve **gluBeginSurface**() parancsokkal kezdjük.
- A görbék, illetve felületek megadására a **gluNurbsCurve**(), illetve **gluNurbsSurface**() parancsok szolgálnak. Ezeket legalább egyszer ki kell adni a közelítő töröttvonal, illetve poligonháló létrehozása érdekében, de meghívhatók a felületi normális és a textúrakoordináták létrehozásához is.

- A **gluEndCurve**(), illetve **gluEndSurface**() parancsokkal zárjuk a NURBS objektum megjelenítését.

GLUnurbsObj ***gluNewNurbsRenderer** (void);

Egy új NURBS objektumstruktúrát hoz létre és ad vissza. Az objektumra a tulajdonságok beállításánál és a megrajzolásnál ezzel az azonosítóval kell hivatkozni. Ha nincs elég memória az objektum allokálásához, akkor a visszaadott cím NULL.

void **gluDeleteNurbsRenderer** (GLUnurbsObj **nobj*);

Törli az *nobj* címen tárolt NURBS objektumot, felszabadítja a lefoglalt memóriát.

void **gluNurbsProperty** (GLUnurbsObj **nobj*, GLenum *property*, GLfloat *value*);

Az *nobj* azonosítójú NURBS objektum megjelenésének tulajdonságai állíthatók be segítségével. A *property* paraméter a tulajdonság azonosítója, *value* pedig az értéke. A *property* paraméter lehetséges értékei és jelentésük:

- GLU_DISPLAY_MODE esetén a megjelenítés módja írható elő, ekkor a *value* paraméter értéke GLU_FILL, GLU_OUTLINE_POLYGON vagy GLU_OUTLINE_PATCH lehet. GLU_FILL esetén kitöltött poligonokkal, GLU_OUTLINE_POLYGON esetén a közelítő poligonok oldalaival, GLU_OUTLINE_PATCH esetén pedig a felületfolt határoló görbéivel (beleértve a trimmelő görbéket is) ábrázolja a NURBS felületet a rendszer. Alapértelmezés: GLU_FILL.

- GLU_NURBS_MODE esetén azt írhatjuk elő, hogy a közelítő töröttvonal, illetve poligonhálót meg kell jeleníteni, vagy a visszahívási mechanizmust kell aktivizálni, hogy a közelítő töröttvonal, illetve poligonháló adatai elérhetők legyenek. Az első esetben a *value* paramétert GLU_NURBS_RENDERER értékre kell állítani, ami egyben az alapértelmezés is, a második esetben pedig GLU_NURBS_TESSELLATOR-ra.

- GLU_CULLING esetén a GL_TRUE érték megadásával a megjelenítési folyamat felgyorsítható, ugyanis ekkor a rendszer nem végzi el a töröttvonallal, illetve poligonokkal való közelítést, ha az objektum az ábrázolandó térrészt leíró csonka gúlán (vagy hasábon) kívül esik. Ha ez a paraméter GL_FALSE - ami egyben az alapértelmezés is -, akkor ilyen esetben is elvégzi.

- GLU_SAMPLING_METHOD esetén a mintavételezési módszert adhatjuk meg, másként nézve azt, hogy a közelítés pontosságát milyen módon akarjuk előírni. Ha *value* értéke:

  - GLU_PARAMETRIC_ERROR, a közelítő töröttvonalnak, illetve poligonoknak a görbétől, illetve a felülettől pixelekben mért távolsága nem lehet nagyobb a **gluNurbsProperty**() *type* = GLU_SAMPLING_TOLERANCE paraméterrel való meghívásánál megadott *value* értéknél.

  - GLU_PATH_LENGTH, a közelítő töröttvonal, illetve poligonok oldalainak pixelekben mért hossza nem lehet nagyobb a **gluNurbsProperty**() *type* = GLU_SAMPLING_TOLERANCE paraméterrel való meghívásánál megadott *value* értéknél.
  - GLU_OBJECT_PATH_LENGTH hatása csaknem teljesen megegyezik a GLU_PATH_LENGTH-nél leírtakkal, egyetlen eltérés, hogy a távolságot nem pixelben, hanem az objektum terének egységében írjuk elő.
  - GLU_OBJECT_PARAMETRIC_ERROR hatása majdnem megegyezik a GLU_PARAMETRIC_ERROR-nál leírtakkal, egyetlen eltérés, hogy a távolságot nem pixelben, hanem az objektum terének egységében írjuk elő.
  - GLU_DOMAIN_DISTANCE, akkor azt adjuk meg, hogy a közelítő töröttvonal, illetve poligonháló csúcspontjait a paramétertartományon mérve milyen sűrűn számítsa ki a rendszer. Ezt a sűrűséget $u$ irányban a **gluNurbsProperty**() *type* = GLU_U_STEP meghívásával, $v$ irányban a *type* = GLU_V_STEP meghívásával írhatjuk elő. Ezeknél a hívásoknál a *value* paraméterrel azt adhatjuk meg, hogy egységnyi paramétertartományon hány osztáspont legyen.

- GLU_SAMPLING_TOLERANCE esetén a közelítés pontosságát írhatjuk elő. Ha a mintavételezési módszer:

  - GLU_PATH_LENGTH, akkor a *value* paraméterrel a közelítő töröttvonal, illetve poligonok oldalainak pixelekben mért maximális hosszát írjuk elő. Alapértelmezés: 50.
  - GLU_OBJECT_PATH_LENGTH, akkor a *value* paraméterrel a közelítő töröttvonal, illetve poligonok oldalainak az objektumkoordináta-rendszerben mért maximális hosszát írjuk elő. Alapértelmezés: 50.

- GLU_PARAMETRIC_TOLERANCE esetén a közelítés pontosságát adhatjuk meg. Ha a mintavételezési módszer:

  - GLU_PARAMETRIC_ERROR, a közelítő töröttvonalnak, illetve poligonoknak a görbétől, illetve a felülettől mért eltérésének pixelekben mért maximumát írhatjuk elő a *value* paraméterrel. Alapértelmezés: 0.5.
  - GLU_OBJECT_PARAMETRIC_ERROR, a közelítő töröttvonalnak, illetve poligonoknak a görbétől, illetve a felülettől mért eltérésének maximumát írhatjuk elő az objektumkoordináta-rendszerben a *value* paraméterrel. Alapértelmezés: 0.5.

- GLU_U_STEP esetén azt adhatjuk meg, hogy az $u$ irányú paraméter 1 egységére hány osztáspont jusson a görbén, illetve a felületen, feltéve, hogy a mintavételezési módszer GLU_DOMAIN_DISTANCE. Alapértelmezés: 100.

- GLU_V_STEP esetén azt adhatjuk meg, hogy az $v$ irányú paraméter 1 egységére hány osztáspont jusson a görbén, illetve a felületen, feltéve, hogy a mintavételezési módszer GLU_DOMAIN_DISTANCE. Alapértelmezés: 100.

- GLU_AUTO_LOAD_MATRIX esetén a GL_TRUE érték, ami az alapértelmezés is, megadásával azt jelezzük, hogy az OpenGL szerverről kell letölteni a nézőpont-modell, a vetítési és a képmező-transzformáció mátrixát a megjelenítéshez. Ha ennek a paraméternek a GL_FALSE értéket adjuk, akkor a felhasználói programnak kell szolgáltatnia ezeket a mátrixokat a **gluSamplingMatricies**() paranccsal.

### 14.3.1. NURBS görbék rajzolása

Mindenekelőtt definiáljuk a NURBS görbéket és az ezekhez szükséges normalizált B-spline alapfüggvényt. Az

$$\begin{aligned} N_j^1(u) &= \begin{cases} 1, & \text{ha } u_j \le u < u_{j+1} \\ 0, & \text{egyébként} \end{cases} \\ N_j^k(u) &= \frac{u-u_j}{u_{j+k-1}-u_j} N_j^{k-1}(u) + \frac{u_{j+k}-u}{u_{j+k}-u_{j+1}} N_{j+1}^{k-1}(u) \end{aligned} \tag{14.3}$$

rekurzióval definiált függvényt $k-1$–edfokú ( $k$–adrendű) normalizált B–spline alapfüggvénynek nevezzük, az $u_j \le u_{j+1} \in \mathbb{R}$ számokat pedig csomóértékeknek. Megállapodás szerint $0/0 \doteq 0$.

Az

$$\mathbf{r}(u) = \sum_{l=0}^{n} \frac{N_l^k(u)\, w_l \mathbf{d}_l}{\sum_{j=0}^{n} N_j^k(u)\, w_j}, u \in [u_{k-1}, u_{n+1}] \tag{14.4}$$

kifejezéssel adott görbét $k-1$–edfokú ($k$–adrendű) NURBS (racionális B–spline) görbének nevezzük, a $\mathbf{d}_l$ pontokat kontrollpontoknak vagy de Boor–pontoknak, a $w_l \ge 0$ skalárokat pedig súlyoknak nevezzük. $N_l^k(u)$ az $l$–edik $k-1$–edfokú normalizált B–spline alapfüggvényt jelöli, melyek értelmezéséhez az $u_0, u_1, \ldots, u_{n+k}$ csomóértékek szükségesek.

Az így definiált görbe általában egyik kontrollponton sem megy át, azonban egybeeső szomszédos csomóértékek megadásával ez elérhető. Ha az első $k$ darab csomóérték megegyezik ($u_0 = u_1 = \cdots = u_{k-1}$), akkor a görbe átmegy a $\mathbf{d}_0$ kontrollponton, hasonlóképpen, ha az utolsó $k$ darab csomóérték megegyezik ($u_{n+1} = u_{n+2} = \cdots = u_{n+k}$), akkor a görbe átmegy a $\mathbf{d}_n$ kontrollponton. A közbülső csomóértékek esetében legfeljebb $k-1$ egymást követő eshet egybe.

void **gluBeginCurve** (GLUnurbsObj **nobj*);

void **gluEndCurve** (GLUnurbsObj **nobj*);

A **gluBeginCurve**() parancs az *nobj* azonosítójú NURBS görbe megadásának kezdetét jelzi, a **gluEndCurve**() parancs pedig a végét. A kettő között a **gluNurbsCurve**() parancs egy vagy több meghívásával lehet a görbét leírni. A **gluNurbsCurve**() parancsot pontosan egyszer kell a GL_MAP1_VERTEX_3 vagy GL_MAP1_VERTEX_4 paraméterrel meghívni.

void **gluNurbsCurve** (GLUnurbsObj **nobj*, GLint *uknot_count*, GLfloat **uknot*, GLint *u_stride*, GLfloat **ctrlarray*, GLint *uorder*, GLenum *type*);

Az *nobj* azonosítójú NURBS görbét rajzolja meg. *uorder* a görbe rendje (a (14.4)

kifejezésbeli $k$), *u_knot_count* a csomóértékek száma (a kontrollpontok száma + a rend, a (14.4) jelölésével $n + k + 1$), *uknot* az első csomóérték címe, *ctrlarray* az első kontrollpont első koordinátájának címe, *ustride* két egymást követő kontrollpontnak GLfloatban mért távolsága, *type* értéke GL_MAP1_VERTEX_3 nem racionális B-spline esetén, GL_MAP1_VERTEX_4 racionális B-spline esetén.

Racionális görbéknél a kontrollpontokat homogén koordinátákban kell megadni, tehát a (14.4) kifejezés $w_i$ súlyait és $\mathbf{d}_i$ kontrollpontjait $\mathbf{p}_i = \begin{bmatrix} w_i\mathbf{d}_i \\ w_i \end{bmatrix}$ alakban kell átadni, azaz a kontrollpontokat meg kell szorozni a súllyal és a 4. koordináta maga a súly lesz.

### 14.3.2. NURBS felületek megjelenítése

Az

$$\mathbf{s}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} \frac{N_i^k(u)\,N_j^l(v)}{\sum_{p=0}^{n}\sum_{q=0}^{m} N_p^k(u)\,N_q^l(v)\,w_{ij}} w_{ij}\mathbf{d}_{ij}\,, u \in [u_{k-1}, u_{n+1}]\,, v \in [v_{l-1}, v_{m+1}] \quad (14.5)$$

kifejezéssel adott felületet NURBS (racionális B–spline) felületnek nevezzük, a $\mathbf{d}_{ij}$ pontokat kontrollpontoknak vagy de Boor–pontoknak, a $w_{ij} \geq 0$ skalárokat pedig súlyoknak nevezzük. $N_i^k(u)$ az $i$–edik $k-1$–edfokú normalizált B–spline alapfüggvényt jelöli (lásd a (14.3) kifejezést), melyek értelmezéséhez az $u_0, u_1, \ldots, u_{n+k}$ csomóértékek szükségesek, $N_j^l(v)$-hez pedig a $v_0, v_1, \ldots, v_{m+l}$ csomóértékek.

Az így definiált felület általában egyik kontrollponton sem megy át, ám egybeeső szomszédos csomóértékek megadásával ez elérhető. A görbékkel analóg módon, ha $u_0 = u_1 = \cdots = u_{k-1}$, $u_{n+1} = u_{n+2} = \cdots = u_{n+k}$, $v_0 = v_1 = \cdots = v_{l-1}$ és $v_{m+1} = v_{m+2} = \cdots = v_{m+l}$ teljesül, akkor a felület illeszkedik a $\mathbf{d}_{00}, \mathbf{d}_{0m}, \mathbf{d}_{n0}, \mathbf{d}_{nm}$ kontrollpontokra.

void **gluBeginSurface** (GLUnurbsObj **nobj*);

void **gluEndSurface** (GLUnurbsObj **nobj*);

A **gluBeginSurface**() parancs az *nobj* azonosítójú NURBS felület megadásának kezdetét jelzi, a **gluEndSurface**() parancs pedig a végét. A kettő között a **gluNurbsSurface**() parancs egy vagy több meghívásával lehet a görbét leírni. A **gluNurbsSurface**() parancsot pontosan egyszer kell a GL_MAP2_VERTEX_3 vagy GL_MAP2_VERTEX_4 paraméterrel meghívni.

A felület trimmelése a **gluBeginTrim**(), **gluEndTrim**() parancsok között kiadott **gluPwlCurve**() és **gluNurbsCurve**() parancsokkal érhető el.

void **gluNurbsSurface** (GLUnurbsObj **nobj*, GLint *uknot_count*, GLfloat **uknot*, GLint *vknot_count*, GLfloat **vknot*, GLint *u_stride*, GLint *v_stride*, GLfloat **ctrlarray*, GLint *uorder*, GLint *vorder*, GLenum *type*);

Az *nobj* azonosítójú NURBS felületet rajzolja meg. *uorder* a felület $u$ irányú rendje (fokszám + 1), *vorder* pedig a $v$ irányú. *uknot_count* az $u$ paraméter csomóértékeinek

száma (*uknot_count* = *uorder* + az $u$ irányú kontrollpontok száma), *vknot_count* pedig a $v$ paraméteré. *ctrlarray* a kontrollháló első pontjának címe. A *type* paraméter lehetséges értékei: GL_MAP2_VERTEX_3 nem racionális B-spline felület esetén, GL_MAP2_VERTEX_4 racionális B-spline felületnél, GL_MAP2_TEXTURE_COORD_* textúrakoordináták, GL_MAP2_NORMAL normálisok létrehozásához.

Az *u_stride*, illetve *v_stride* paraméterekkel azt kell megadnunk, hogy $u$, illetve $v$ irányban a szomszédos kontrollpontok adatainak távolsága hány GLfloat változó.

### 14.3.3. Trimmelt felületek

Előfordulhat, hogy a NURBS felületfoltnak csak valamely darabját (darabjait) akarjuk megjeleníteni. Ezt úgy tehetjük meg, hogy az értelmezési tartományt, mely a (14.5) kifejezés jelöléseit használva eredetileg az $[u_{k-1}, u_{n+1}] \times [v_{l-1}, v_{m+1}]$ téglalap, töröttvonalak és NURBS görbék felhasználásával leszűkítjük. Az ilyen felületfoltokat trimmelt NURBS foltnak nevezzük. A trimmelés határát a paramétersík egységnégyzetében ($[0,1] \times [0,1]$) töröttvonalakból és NURBS görbékből álló zárt görbékkel adhatjuk meg. Több ilyen zárt határt is megadhatunk, melyek egymásba ágyazhatók (a trimmelt területben is lehet lyuk), de nem metszhetik egymást. A határoló görbék irányítottak, és úgy kell őket megadni, hogy a rendszer a görbétől balra lévő pontokat tekinti az értelmezési tartomány pontjainak.

void **gluBeginTrim** (GLUnurbsObj **nobj*);

void **gluEndTrim** (GLUnurbsObj **nobj*);

Az *nobj* azonosítójú NURBS felülethez, annak definiálása során a **gluBeginTrim**() és **gluEndTrim**() zárójelpár között adhatunk meg zárt trimmelő görbéket. A trimmelő görbe egy irányított zárt hurok, mely a NURBS felület határát írja le. A trimmelő görbéket (több is megadható) a felület definiálásakor, vagyis a `gluBeginSurface(nobj);` és `gluEndSurface(nobj);` között kell megadni. Egy-egy trimmelő görbe több NURBS görbéből (**gluNurbsCurve**())és töröttvonalból (**gluPwlCurve**()) is állhat, azonban ügyelnünk kell arra, hogy egy zárt - az egyik végpontja essen egybe a következő kezdőpontjával, valamint az utolsó végpontja az első kezdőpontjával - , önmagát nem metsző hurkot alkossanak.

A rendszer által megjelenített felület mindig az irányított trimmelő görbétől balra van (a paramétersíkon nézve), az elsődleges értelmezési tartomány határának irányítása tehát óramutató járásával ellentétes. Ha ebbe a felületbe egy lyukat akarunk vágni, akkor a lyuk határát az óramutató járásával megegyező irányítású görbével kell leírni.

Egy felülethez több trimmelő görbe is megadható, de ezek nem metszhetik egymást, és ügyelnünk kell az irányítások helyes megválasztására.

void **gluPwlCurve** (GLUnurbsObj **nobj*, GLint *count*, GLfloat **array*, GLint *stride*, GLenum *type*);

Az *nobj* azonosítójú NURBS felülethez egy trimmelő töröttvonalat ad meg. A trimmelő töröttvonal csúcsainak száma *count*, és a csúcspontok koordinátái az *array* címen kezdődnek. A *type* paraméter leggyakrabban GLU_MAP1_TRIM_2, ami azt jelenti, hogy

a paramétersíkra illeszkedő csúcspontokat az $(u, v)$ koordinátákkal adjuk meg, de lehet GLU_MAP1_TRIM_3 is, mely esetben az $(u, v, w)$ homogén koordinátákkal. A *stride* paraméterrel az egymást követő csúcspontoknak GLfloatokban mért távolságát kell megadni.

### 14.3.4. Hibakezelés

A GLU függvénykönyvtár a NURBS objektumokkal kapcsolatban 37 különböző hibalehetőséget figyel. Ha regisztráljuk hibafüggvényünket, akkor értesülhetünk az általunk elkövetett hibákról. Ezt a regisztrációt a **gluNurbsCallback**() paranccsal végezhetjük el.

void **gluNurbsCallback** (GLUnurbsObj **nobj*, GLenum *which*, void (**fn*)(GLenum *errorCode*));

*which* a visszahívás típusa, hibafigyelés esetén értéke GLU_ERROR (ez a függvény más célra is használható, a *which* paraméter lehetséges értékeinek teljes listáját a 14.3. táblázat tartalmazza). Amikor az *nobj* NURBS objektummal kapcsolatos függvények végrehajtása során a rendszer hibát észlel, meghívja az *fn* függvényt. Az *errorCode* a GLU_NURBS_ERROR$i$ ($i = 1, 2, \ldots, 37$) értékek valamelyike lehet, mely jelentését a **gluErrorString**() függvénnyel kérdezhetjük le.

```
gluNurbsCallback(nurbs, GLU_ERROR, nurbshiba); // a hibafüggvény
regisztrálása
void CALLBACK nurbshiba(GLenum errorCode); //a hibafüggvény
   {
   const GLubyte *hiba;
       hiba = gluErrorString(errorCode);
       fprintf(stderr,''NURBS hiba:  %s\n'',hiba);
       exit(0);
   }
```

### 14.3.5. A NURBS objektumokat közelítő adatok visszanyerése

A rendszer a NURBS objektumokat töröttvonalakkal, illetve poligonokkal közelíti, és ezeket jeleníti meg a beállított tulajdonságoknak megfelelően. A GLU függvénykönyvtár 1.3 verziója lehetővé teszi, hogy a közelítő adatokat ne jelenítse meg a rendszer, hanem azokat visszaadja a felhasználói programnak további feldolgozásra. A következő lépések szükségesek ennek elérése érdekében:

- Hívjuk meg a **gluNurbsProperty**() függvényt a *property* = GLU_NURBS_MODE, *value* = GLU_NURBS_TESSELLATOR paraméterekkel.
- A **gluNurbsCallback**() függvény meghívásaival regisztráljuk a rendszer által meghívandó függvényeket (lásd a 14.3. táblázatot).

void **gluNurbsCallback** (GLUnurbsObj **nobj*, GLenum *which*, void (**fn*)());

*nobj* a NURBS objektum azonosítója, *which* a regisztrálandó függvényt leíró, a rendszer által definiált konstans a 14.3. táblázat szerint. Ha a **gluNurbsProperty**() függvényt a *property* = GLU_NURBS_MODE, *value* = GLU_NURBS_TESSELLATOR paraméterekkel hívtuk meg előzőleg, akkor a GLU_ERROR-on kívül 12 további visszahívandó függvényt regisztráhatunk (lásd a 14.3. táblázatot).

A regisztrált függvényt bármikor kicserélhetjük egy másikra a **gluNurbsCallback**() újabb meghívásával, illetve törölhetjük a regisztrációt, ha a függvény neveként a NULL pointert adjuk meg. Az adatokhoz az általunk regisztrált függvényeken keresztül juthatunk hozzá.

A visszahívandó függvények közül hat lehetővé teszi, hogy a felhasználó adatokat adjon át neki.

void **gluNurbsCallbackData** (GLUnurbsObj **nobj* void **userData*);

*nobj* a NURBS objektum azonosítója, *userData* az átadandó adat címe.

14.3. táblázat. A **gluNurbsCallback**() parancs paramétereinek értékei

| a *which* paraméter értéke | prototípus |
|---|---|
| GLU_NURBS_BEGIN | void **begin**(GLenum *type*); |
| GLU_NURBS_BEGIN_DATA | void **beginDat**(GLenum *type*, void **uData*); |
| GLU_NURBS_TEXTURE_COORD | void **texCoord**(GLfloat **tCoord*); |
| GLU_NURBS_TEXTURE_COORD_DATA | void **texCoordDat**(GLfloat **tCoord*, void **uData*); |
| GLU_NURBS_COLOR | void **color**(GLfloat **color*); |
| GLU_NURBS_COLOR_DATA | void **colorDat**(GLfloat **color*, void **uData*); |
| GLU_NURBS_NORMAL | void **normal**(GLfloat **norm*); |
| GLU_NURBS_NORMAL_DATA | void **normalDat**(GLfloat **norm*, void **uData*); |
| GLU_NURBS_VERTEX | void **vertex**(GLfloat **vertex*); |
| GLU_NURBS_VERTEX_DATA | void **vertexDat**(GLfloat **vertex*, void **uData*); |
| GLU_NURBS_END | void **end**(void); |
| GLU_NURBS_END_DATA | void **endDat**(void **uData*); |
| GLU_ERROR | void **error**(GLenum *errorCode*); |

## 14.4. Gömb, kúp és körgyűrű rajzolása

A GLU függvénykönyvtár lehetőséget biztosít gömb, forgási csonkakúp, körgyűrű és körgyűrűcikk megjelenítésére. A könyvtár készítői ezeket az alakzatokat összefoglalóan

másodrendű felületnek nevezik, ami helytelen, mivel a körgyűrű nem másodrendű felület, a forgáskúp csak speciális esete a másodrendű kúpnak (ráadásul a rendszer ezt hengernek nevezi) és a másodrendű felületek túlnyomó többsége hiányzik (ellipszoid, egy- és kétköpenyű hiperboloid, elliptikus és hiperbolikus paraboloid, általános másodrendű kúp és henger).

A NURBS felületekhez hasonlóan ezeket az alakzatokat is poligonhálóval közelíti a rendszer és a közelítő poligonhálót ábrázolja. Az objektumok ábrázolásának a menete is nagymértékben hasonlít a NURBS görbék és felületek megjelenítéséhez.

- Létre kell hoznunk egy objektumstruktúrát a **gluNewQuadric**() paranccsal.
- Beállíthatjuk az objektum megjelenését befolyásoló attribútumokat, így:
  - A **gluQuadricOrientation**() paranccsal az irányítást adhatjuk meg, vagyis azt, hogy mi tekintendő külső, illetve belső résznek.
  - A **gluQuadricDrawStyle**() paranccsal azt írhatjuk elő, hogy a közelítő poligonhálót a csúcspontjaival, éleivel vagy kitöltött poligonjaival ábrázolja a rendszer.
  - A **gluQuadricNormal**() paranccsal kiválaszthatjuk, hogy a normálisokat minden csúcspontban kiszámolja a rendszer, vagy laponként csak egyet, vagy egyáltalán ne számoljon normálist.
  - A **gluQuadricTexture**() paranccsal írhatjuk elő a textúrakoordináták létrehozását.
- Hibafüggvényt regisztrálhatunk a **gluQuadricCallback**() paranccsal, vagyis azt a függvényt, amit a rendszer meghív, ha hibát észlel az objektum létrehozása vagy ábrázolása során.
- Meg kell hívni az objektumot megjelenítő parancsot (**gluSphere**(), **gluCylinder**(), **gluDisk**(), **gluPartialDisk**()).
- Törölhetjük az objektumot a **gluDeleteQuadric**() paranccsal, amennyiben a továbbiakban már nincs rá szükségünk.

GLUquadricObj* **gluNewQuadric** (void);

Egy új GLUquadricObj típusú struktúraváltozót hoz létre, és ennek a címét adja vissza. Sikertelen hívás esetén a visszaadott cím NULL.

void **gluDeleteQuadric** (GLUquadricObj **qobj*);

Törli a *qobj* címen tárolt GLUquadricObj típusú változót, azaz felszabadítja a létrehozásakor lefoglalt memóriát.

void **gluQuadricCallback** (GLUquadricObj **qobj*, GLenum *which*, void(**fn*)());

Az itt megadott *fn* függvényt hívja meg a rendszer, ha a *qobj* megjelenítése során hibát észlel. A *which* változó értéke csak GLU_ERROR lehet. Ha a meghívandó függvény

címeként a NULL címet adjuk át, akkor a rendszer ezen objektummal kapcsolatos hiba esetén nem fog semmit meghívni. A felhasználói program az *fn* függvény paraméterében kapja vissza a hiba kódját, melynek szöveges leírását ezen kóddal meghívott **gluErrorString**() függvénnyel kapjuk meg (lásd még a 14.3.4. pontot).

void **gluQuadricDrawStyle** (GLUquadricObj **qobj*, GLenum *drawStyle*);

A *qobj* objektum ábrázolásának módját állítja be. *drawStyle* lehetséges értékei:

- GLU_POINT, az objektumot a közelítő poligonháló csúcspontjaival ábrázolja.
- GLU_LINE, az objektumot a közelítő poligonháló éleivel ábrázolja.
- GLU_SILHOUETTE, az objektumot a közelítő poligonháló éleivel ábrázolja, de a komplanáris, közös oldallal bíró poligonoknak a közös oldalait nem rajzolja meg (ennek az opciónak pl. a kúpok és hengerek alkotóinak megrajzolásakor van jelentősége).
- GLU_FILL, az objektumot a közelítő poligonháló kitöltött poligonjaival ábrázolja. A poligonokat a normálisukhoz képest pozitív (az óramutató járásával ellentétes) irányításúnak tekinti.

void **gluQuadricOrientation** (GLUquadricObj **qobj*, GLenum *orientation*);

A *qobj* objektum normálisainak irányítását határozza meg. Az orientation paraméter lehetséges értékei GLU_OUTSIDE, illetve GLU_INSIDE, melyek kifelé, illetve befelé mutató normálisokat eredményeznek. Ezek értelmezése gömb és forgáskúp esetén értelemszerű, körgyűrű esetén a kifelé irány a lap $z$ tengelyének pozitív fele. Alapértelmezés: GLU_OUTSIDE.

void **gluQuadricNormals** (GLUquadricObj **qobj*, GLenum *normals*);

A *qobj* objektum normálisainak kiszámítási módját határozza meg. A *normals* paraméter lehetséges értékei:

- GLU_NONE, a rendszer nem hoz létre normálisokat. Ez csak akkor ajánlott, ha a megvilágítás nem engedélyezett.
- GLU_FLAT, laponként csak egy normálist hoz létre a rendszer, ami konstans árnyaláshoz (lásd a 4. fejezetet) ajánlott.
- GLU_SMOOTH, minden csúcsponthoz külön normálist hoz létre, ami folytonos árnyalás (lásd a 4. fejezetet) esetén ajánlott.

Alapértelmezés: GLU_NONE.

void **gluQuadricTexture** (GLUquadricObj **qobj*, GLboolean *textureCoords*);

A *qobj* objektumhoz a *textureCoords* paraméter értékének megfelelően textúrakoordinátákat is létrehoz (GL_TRUE), vagy nem hoz létre (GLU_FALSE). Alapértelmezés: GLU_FALSE. A textúraleképezés módja az objektum típusától függ.

void **gluSphere** (GLUquadricObj **qobj*, GLdouble *radius*, GLint *slices*, GLint *stacks*);

Megjeleníti a *qobj* azonosítójú, origó középpontú, *radius* sugarú gömböt. A közelítéshez a rendszer a $z$ tengelyre merőlegesen *slices* darab síkkal metszi a gömböt (szélességi körök), továbbá *stacks* darab $z$ tengelyre illeszkedő síkkal (hosszúsági körök). A textúra leképezése a következő: a textúra $t$ koordinátáját lineárisan képezi le a hosszúsági körökre az alábbiak szerint $t = 0 \rightarrow z = -radius$, $t = 1 \rightarrow z = radius$; az $s$ koordinátákat pedig a szélességi körökre a következő megfeleltetés szerint $s = 0 \rightarrow (R, 0, z)$, $s = 0.25 \rightarrow (0, R, z)$, $s = 0.5 \rightarrow (-R, 0, z)$, $s = 0.75 \rightarrow (0, -R, z)$, $s = 1 \rightarrow (R, 0, z)$, ahol $R$ a $z$ magasságban lévő szélességi kör sugara.

void **gluCylinder** (GLUquadricObj **qobj*, GLdouble *baseRadius*, GLdouble *topRadius*, GLdouble *height*, GLint *slices*, GLint *stacks*);

Azt a *qobj* azonosítójú $z$ tengelyű forgási csonkakúpot ábrázolja, melynek alapköre a $z = 0$, fedőköre a $z = height$ magasságban van. A közelítéshez a rendszer *slices* darab $z$ tengelyre merőleges kört és *stacks* darab alkotót vesz fel. Az alapkör sugara *baseRadius*, a fedőkör sugara pedig *topRadius*. $baseRadius = 0$ esetén forgáskúp felületet kapunk, $topRadius = baseRadius$ esetén pedig forgáshengert. A rendszer csak a csonkakúp palástját ábrázolja, az alap és fedőkört nem. A textúra leképezése a következő: a textúra $t$ koordinátáját lineárisan képezi le az alkotókra úgy, hogy a megfeleltetés $t = 0 \rightarrow z = 0$, $t = 1 \rightarrow z = height$ legyen. Az $s$ koordináták leképezése megegyezik a gömbnél leírtakkal.

void **gluDisk** (GLUquadricObj **qobj*, GLdouble *innerRadius*, GLdouble *outerRadius*, GLint *slices*, GLint *rings*);

A *qobj* azonosítójú körgyűrűt ábrázolja a $z = 0$ síkon. A körgyűrű külső sugara *outerRadius*, belső sugara pedig *innerRadius*. $innerRadius = 0$ esetén teljes körlap jön létre. A közelítéshez a rendszer *slices* darab körcikkre és *rings* darab koncentrikus körgyűrűre osztja az alakzatot. A normálisok irányításakor a $z$ tengely pozitív felét tekinti kifelé iránynak, ami a **gluQuadricOrientation**() paranccsal megváltoztatható. A textúra leképezését lineárisan végzi el a következő megfeleltetés szerint: $(s = 1, t = 0.5) \rightarrow (R, 0, 0)$, $(s = 0.5, t = 1) \rightarrow (0, R, 0)$, $(s = 0, t = 0.5) \rightarrow (-R, 0, 0)$ és $(s = 0.5, t = 0) \rightarrow (0 - R, 0)$, ahol $R = outerRadius$.

void **gluPartialDisk** (GLUquadricObj **qobj*, GLdouble *innerRadius*, GLdouble *outerRadius*, GLint *slices*, GLint *rings*, GLdouble *startAngle*, GLdouble *sweepAngle*);

A *qobj* azonosítójú körgyűrűcikket ábrázolja. A körgyűrűcikk a $z = 0$ síkon jön létre, külső sugara *outerRadius*, belső sugara *innerRadius*, a cikk kezdőszöge az $y$ tengely

pozitív felétől mért *startAngle*, középponti szöge *sweepAngle*. A szögeket fokban kell megadni. $innerRadius = 0$ esetén körcikket kapunk. A normálisok irányítása és a textúra leképezése megegyezik a körgyűrűnél leírtakkal.

# 15. fejezet

# A gluj függvénykönyvtár

A GLUJ függvénykönyvtár az OpenGL görbe- és felületmegjelenítő képességének kiegészítésére készült. Használatához inkludálni kell a `gluj.h` fájlt, valamint a programhoz kell szerkeszteni a `gluj32.lib` könyvtárat. A mellékelt változat a Visual C/C++ 6.0 rendszerhez készítettük.

## 15.1. Görbéket és felületeket megjelenítő függvények

Használatuk gyakorlatilag megegyezik a GLU függvénykönyvtár NURBS görbét, illetve felületet megjelenítő függvényeinek használatával, tehát a **gluNewNurbsRenderer**() függvénnyel létre kell hozni egy NURBS objektum struktúrát, mely után ennek a tulajdonságai a **gluNurbsProperty**() függvénnyel beállíthatók (a tulajdonságok közül a közelítés pontosságát - GLU_SAMPLING_TOLERANCE - veszik figyelembe a GLUJ függvények), majd görbék esetén a **gluBeginCurve**(), **gluEndCurve**(), felületeknél pedig a **gluBeginSurface**(), **gluNurbsSurface**() függvények között meghívhatjuk a megfelelő görbe-, illetve felületrajzoló **gluj** függvényt.

### 15.1.1. Görbék rajzolása

void **glujBezierCurve** (GLUnurbsObj **nobj*, GLint *cp_count*, GLint *stride*, GLfloat **ctrlarray*, GLenum *type*);

Az *nobj* azonosítójú Bézier-görbét rajzolja meg.

*cp_count* a kontrollpontok száma,

*stride* az egymást követő kontrollpontok GLfloatban mért távolsága,

**ctrlarray* az első kontrollpont címe,

*type* paraméter lehetséges értékei: GL_MAP1_VERTEX_3 nem racionális Bézier-görbe esetén (a pontok három, vagyis Descartes-féle koordinátával adottak), GL_MAP1_VERTEX_4 racionális Bézier-görbe esetén (a pontok négy, azaz homogén koordinátával adottak).

void **glujHermiteSpline** (GLUnurbsObj **hs*, GLint *cp_count,* GLint *cp_stride*, GLfloat **cp*, GLenum *type*, GLint *tg_stride,* GLfloat **tg*);

A *hs* azonosítójú Hermite-spline görbét rajzolja meg.

*cp_stride* az egymást követő pontok GLfloatban mért távolsága,

**cp* az első pont címe,

*type* paraméter lehetséges értékei: GL_MAP1_VERTEX_3 ha a pontok három, vagyis Descartes-féle koordinátával adottak, GL_MAP1_VERTEX_4 ha a pontok négy, azaz homogén koordinátával adottak,

*tg_stride* az egymást követő érintővektorok GLfloatban mért távolsága,

**tg* az első érintővektor címe.

void **glujCircle** (GLUnurbsObj **nobj*, GLint *stride*, GLfloat **points*, GLfloat *radius*, GLenum *type*);

Az *nobj* azonosítójú kört rajzolja meg.

*stride* az egymást követő pontok GLfloatban mért távolsága,

**points* a kör középpontját és síkjának normálisát tartalmazó vektor címe,

*radius* a kör sugara,

*type* paraméter lehetséges értékei: GL_MAP1_VERTEX_3 ha a pontok három, vagyis Descartes-féle koordinátával adottak, GL_MAP1_VERTEX_4 ha a pontok négy, azaz homogén koordinátával adottak.

void **glujTrimmedNurbsCurve** (GLUnurbsObj **nobj*, GLint *knot_count*, GLfloat **knot*, GLint *stride*, GLfloat **ctrlarray*, GLint *order*, GLenum *type*, GLfloat *u_min*, GLfloat *u_max*);

Az $[u\_min, u\_max]$ és $[knot[order-1], knot[knot\_count - order]]$ intervallumok metszetét veszi és az *nobj* azonosítójú NURBS görbének ezen intervallum fölötti darabját jeleníti meg.

*knot_count* a csomóértékek száma (*knot_count* = *order* + a kontrollpontok száma),

**knot* az első csomóérték címe,

*stride* a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

**ctrlarray* a kontrollpoligon első pontjának a címe,

*order* a görbe rendje,

*type* paraméter lehetséges értékei: GL_MAP1_VERTEX_3 nem racionális B-spline görbe esetén (a pontok három, vagyis Descartes-féle koordinátával adottak), GL_MAP1_VERTEX_4 racionális B-spline görbe esetén (a pontok négy, azaz homogén koordinátával adottak).

void **glujParamCurve** (void (**fn*)(float, float *), GLenum *mode*, GLint *ufixed_count*, GLfloat **u_fixed*, GLfloat *u_min*, GLfloat *u_max*);

Az *fn* függvénnyel leírt paraméteres görbének az $[u\_\min, u\_\max]$ intervallum fölötti részét ábrázolja töröttvonallal vagy pontjaival.

*fn* a görbe egy pontját kiszámító függvény. A függvény prototípusa:

void **fn**(float *u*, float *p[3]*);

mely a görbe $u$ paraméterű pontját kiszámítja és a $p$ vektorban visszaadja.

*mode* a megjelenítés módját írja elő, lehetséges értékei:

GLUJ_LINES a görbét *ufixed_count* $\geq 0$ számú rögzített $u$ értékhez tartozó görbepontot összekötő töröttvonallal jeleníti meg. $*u\_fixed$ a rögzített $u$ értékeket tartalmazó vektor címe. Ha *ufixed_count* $> 1$ és $*u\_fixed = NULL$, akkor a paramétertartományban egymástól egyenlő távolságra lévő *ufixed_count* darab $u$ értéket rögzít a függvény, és az ezekhez tartozó pontokat összekötő töröttvonalat ábrázolja.

GLUJ_POINT_MESH a görbét az adott $u$ értékekhez tartozó pontokkal ábrázolja. A *ufixed_count,* $*u\_fixed$ paraméterekre a GLUJ_LINES -nál leírtak érvényesek.

Például az $x(u) = 1/\cosh(u), y(u) = u - \tanh(u)$ koordinátafüggvényekkel adott traktrixot a

```
void traktrix(float u,float p[3])
{
    p[0] = 1./cosh(u);
    p[1] = u - tanh(u);
    p[2] = 0.;
}
```

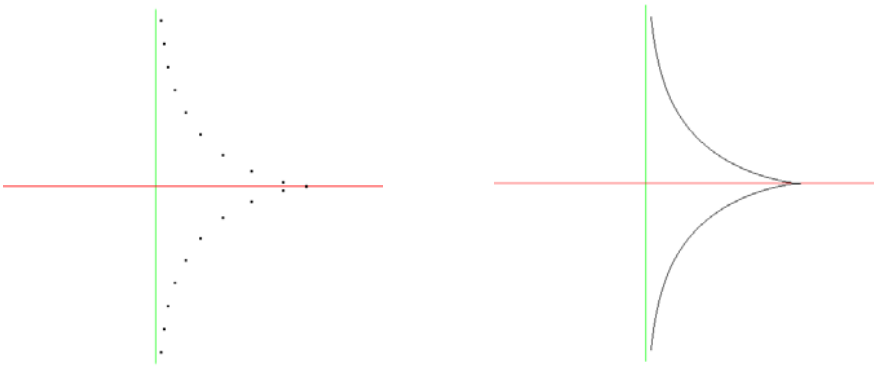
függvénnyel írjuk le. A

```
glujParamCurve(traktrix,GLUJ_POINT_MESH,20,NULL,-4.0,4.);
```

függvényhívás eredménye a 15.1./a ábra, a

```
glujParamCurve(traktrix,GLUJ_LINES,100,NULL,-4.0,4.);
```

hívásé pedig a 15.1./b ábra.



a b

15.1. ábra. A **glujParamCurve**( ) függvénnyel ábrázolt görbék

### 15.1.2. Felületek szemléltetése

void **glujNurbsSurface** (GLUnurbsObj **nobj*, GLint *uknot_count*, GLfloat **uknot*, GLint *vknot_count*, GLfloat **vknot*, GLint *u_stride*, GLint *v_stride*, GLfloat **ctrlarray*, GLint *uorder*, GLint *vorder*, GLenum *type*, GLenum *mode*, GLint *ufixed_count,* GLfloat **u_fixed*, GLint *vfixed_count*, GLfloat **v_fixed*, GLfloat *u_min*, GLfloat *u_max*, GLfloat *v_min*, GLfloat *v_max*, GLfloat *fl,* int *ud*, int *vd*);

Az *nobj* azonosítójú NURBS felület kontrollpontjait, kontrollpoligonját, vagy a $([u\_\min, u\_\max] \cap [uknt[uorder-1], uknt[uknot\_count - uorder]]) \times ([v\_\min, v\_\max] \cap [vknt[vorder-1], vknt[vknot\_count - vorder]])$ tartomány fölötti paramétervonalait, pontjait, vagy normálisait jeleníti meg.

*uknot_count* az $u$ paraméter csomóértékeinek száma ($uknot\_count = uorder +$ az $u$ irányú kontrollpontok száma),

**uknot* az első $u$ irányú csomóérték címe,

*vknot_count* az $v$ paraméter csomóértékeinek száma ($vknot\_count = vorder +$ az $v$ irányú kontrollpontok száma),

**vknot* az első $v$ irányú csomóérték címe,

*u_stride* $u$ irányban a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

*v_stride* $v$ irányban a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

**ctrlarray* a kontrollháló első pontjának a címe,

*uorder* a felület $u$ irányú rendje,

*vorder* a felület $v$ irányú rendje,

*type* paraméter lehetséges értékei: GL_MAP2_VERTEX_3 nem racionális B-spline felület esetén (a pontok három, vagyis Descartes-féle koordinátával adottak), GL_MAP2_VERTEX_4 racionális B-spline felület esetén (a pontok négy, azaz homogén koordinátával adottak),

*mode* a megjelenítés módját írja elő, lehetséges értékei:

GLUJ_CONTROL_POLYGON_MESH a kontrollhálót rajzolja meg,

GLUJ_CONTROL_POINT_MESH a kontrollpontokat jeleníti meg,

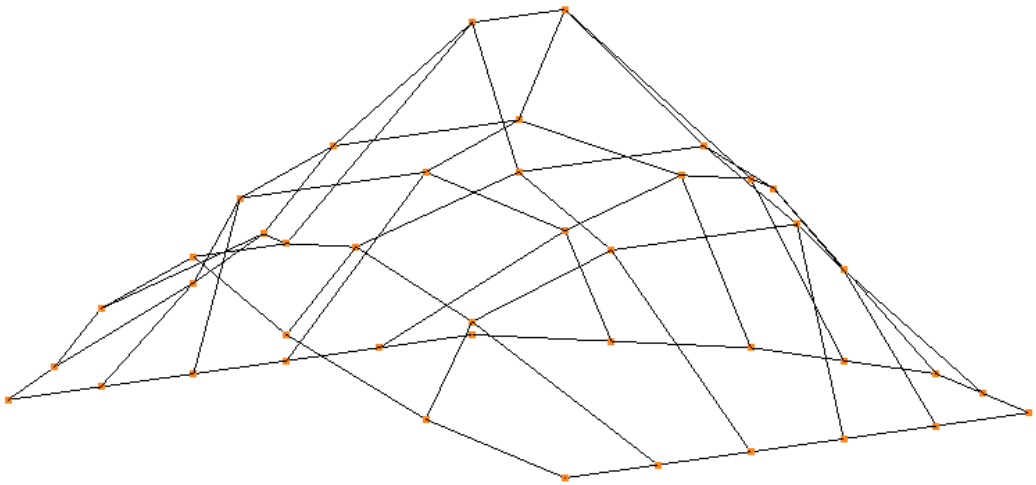
GLUJ_ISO_LINES a felületet paramétervonalaival ábrázolja, mégpedig az $ufixed\_count \geq 0$ számú rögzített $u$ értékhez tartozó $v$ irányú, és a $vfixed\_count \geq 0$ számú rögzített $v$ értékhez tartozó $u$ irányú paramétervonallal. **u_fixed* a rögzített $u$ értékeket tartalmazó vektor címe, **v_fixed* a rögzített $v$ értékeket tartalmazó vektoré. Ha $ufixed\_count > 1$ és $*u\_fixed = NULL$, akkor a paramétertartományban egymástól egyenlő távolságra lévő *ufixed_count* darab $u$ értéket rögzít a függvény és az ezekhez tartozó $v$ irányú paramétervonalakat jeleníti meg. A *vfixed_count* és **v_fixed* paraméterek használata ezzel analóg.

GLUJ_ISO_LINES_WITH_OFFSET a felületet paramétervonalaival ekvidisztáns (attól adott távolságra lévő) görbékkel ábrázolja. Az *ufixed_count, vfixed_count, *u_fixed és *v_fixed* paraméterek jelentése megegyezik a GLUJ_ISO_LINES módnál leírtakkal. $u$ irányú paramétervonalak esetén az értelmezési tartományt *ud* egyenlő részre osztja, az ezekhez tartozó felületi pontokat a pontbeli normális mentén *fl* mértékben el-

tolja és ezeket a pontokat köti össze egyenes szakaszokkal. A $v$ irányú paramétervonalak létrehozása ezzel analóg.

GLUJ_POINT_MESH a felületet a megadott $u$ és $v$ értékekhez tartozó pontokkal ábrázolja. Az *ufixed_count, *u_fixed, vfixed_count, *v_fixed* paraméterekre a GLUJ_ISO_LINES -nál leírtak érvényesek.

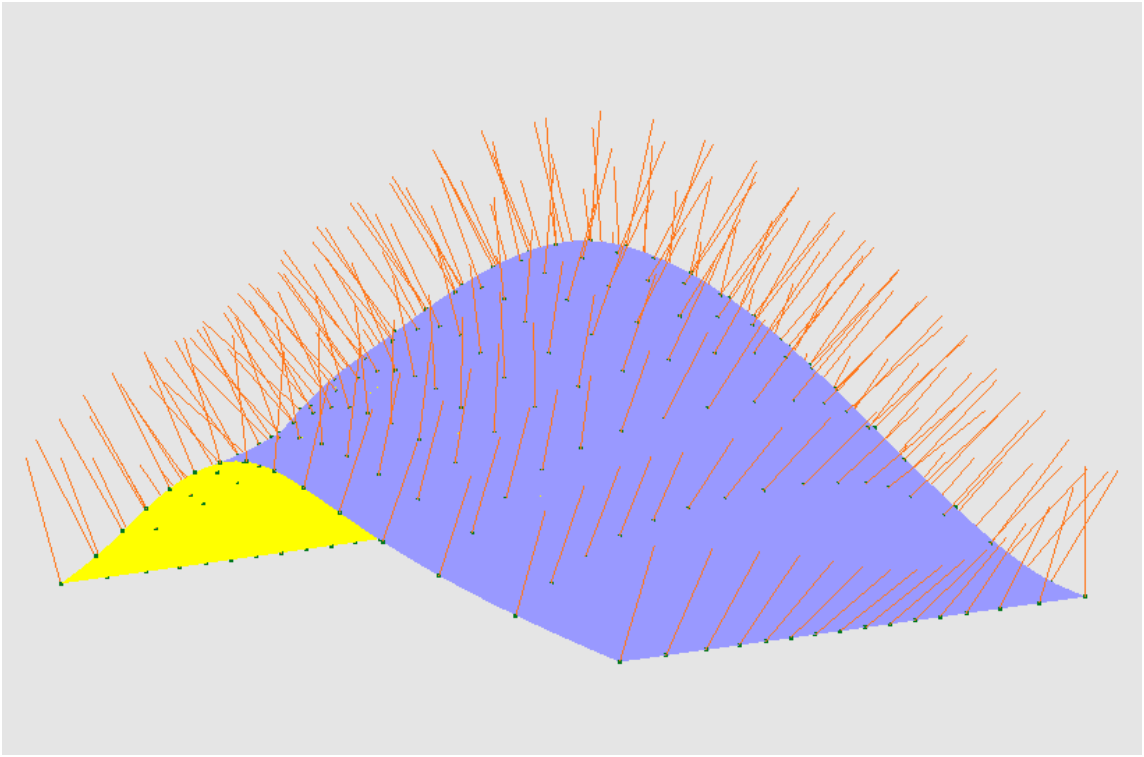
GLUJ_NORMAL_MESH a felületnek a megadott $u$ és $v$ értékekhez tartozó normálisait ábrázolja. Az *ufixed_count, *u_fixed, vfixed_count, *v_fixed* paraméterekre a GLUJ_ISO_LINES -nál leírtak érvényesek. A normálvektorok hosszát és irányítását az *fl* paraméter befolyásolja: $fl = 0$ esetén a normálvektorok hossza és irányítása a kiszámított lesz, egyébként a normalizált normálvektor *fl*-szerese.



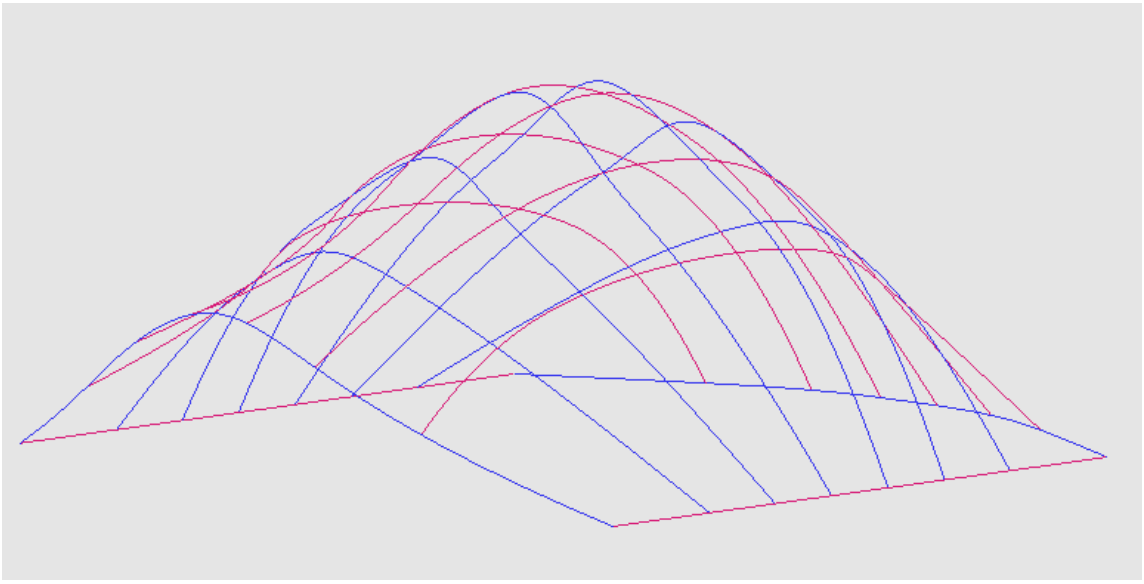
15.2. ábra. A felület kontrollhálója és kontrollpontjai

A 15.2. - 15.6. ábrák ugyanannak a felületnek a **glujNurbsSurface**() függvénnyel elérhető néhány szemléltetési módját demonstrálják. A 15.2. ábrán a felület kontrollhálóját és a kontrollpontjait láthatjuk. Ehhez a **glujNurbsSurface**() függvényt kétszer kell meghívni: egyszer a GLUJ_CONTROL_POLYGON_MESH, egyszer a GLUJ_CONTROL_POINT_MESH paraméterrel. A 15.3. ábrán a kitöltött poligonokkal ábrázolt megvilágított felületet láthatjuk (ezt a **gluNurbsSurface**() függvénnyel rajzoltuk), valamint felületi pontokat és ezekben az egységnyi hosszúságú normálvektorokat. Az utóbbiakat a **glujNurbsSurface**() függvénynek a GLUJ_POINT_MESH, illetve GLUJ_NORMAL_MESH paraméterekkel való meghívásával hoztuk létre. A 15.4. ábra paramétervonalaival szemlélteti a felületet (GLUJ_ISO_LINES opció). A 15.5. és a 15.6. ábra azt demonstrálja, hogy miért van szükség a paramétervonallal ekvidisztáns görbékre (GLUJ_ISO_LINES_WITH_OFFSET opció). A 15.5. ábrán a felület árnyalt képét látjuk, amire folytonos vonallal rárajzoltuk a paramétervonalakat. A képen a paramétervonalak nem folytonosak, szinte véletlenszerűen hol a felület, hol a paramétervonal látszik a megfelelő pixeleken. Ezért csaláshoz folyamodunk, a paramétervonalak helyett a velük ekvi-

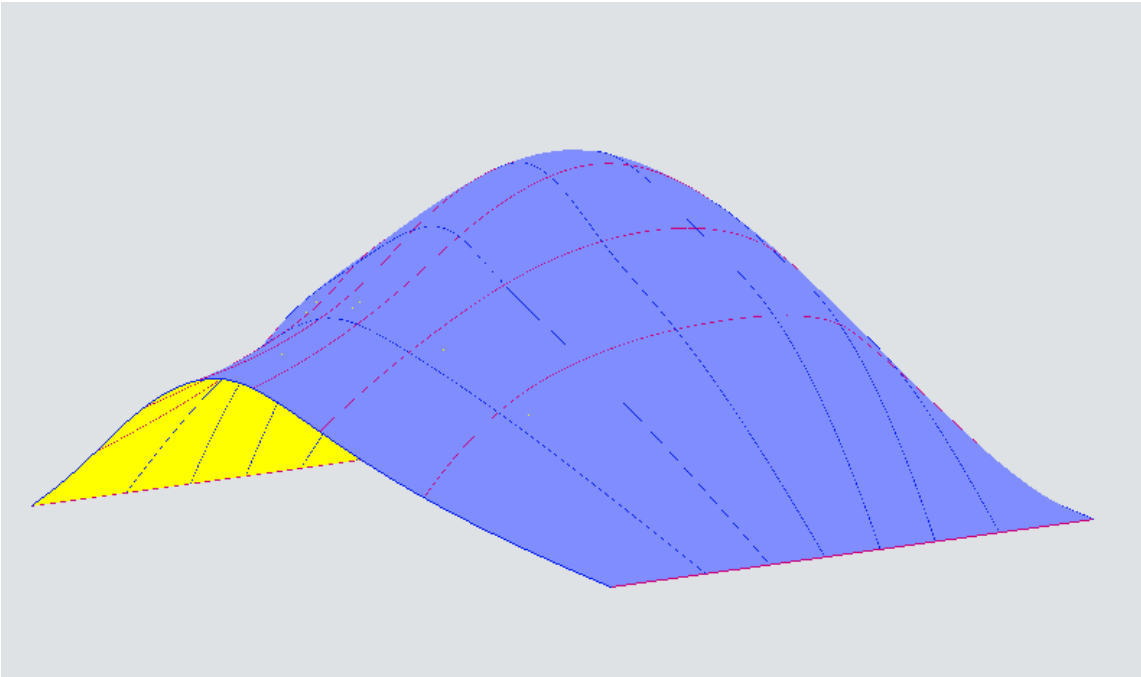
disztáns görbéket rajzoljuk meg, lásd a 15.6. ábrát. Már nagyon csekély eltolás - ábránkon ez 0.016 egység - is elegendő a kívánt hatás elérése érdekében.



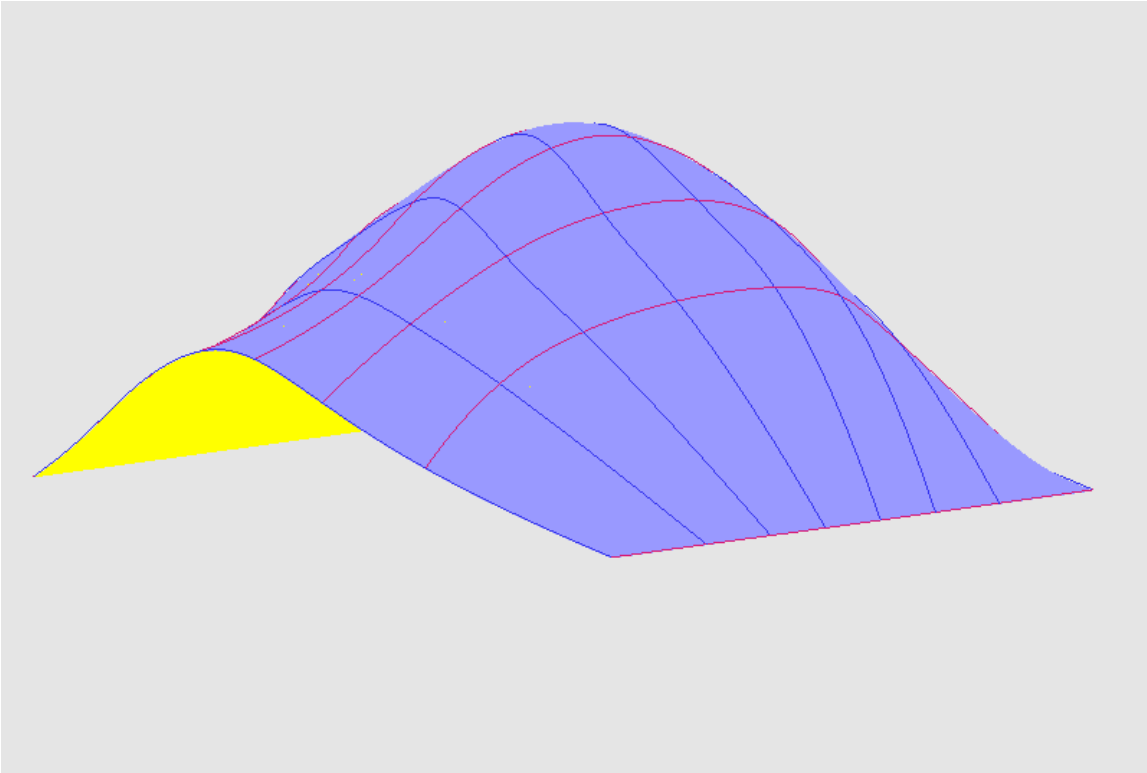
15.3. ábra. A megvilágított felület, néhány felületi pont és azokban a felületi normális



15.4. ábra. A felület néhány paramétervonala



15.5. ábra. A felület paramétervonalakkal



15.6. ábra. A felület és a paramétervonalaival ekvidisztáns görbék

void **glujBezierSurface** (GLUnurbsObj **nobj*, GLint *ucp_count*, GLint *vcp_count*, GLint *u_stride*, GLint *v_stride*, GLfloat **ctrlarray*, GLenum *type*, GLenum *mode*, GLint *ufixed_count,* GLfloat **u_fixed*, GLint *vfixed_count*, GLfloat **v_fixed*, GLfloat *u_min*, GLfloat *u_max*, GLfloat *v_min*, GLfloat *v_max*, GLfloat *fl,* int *ud*, int *vd*);

Az *nobj* azonosítójú Bézier-felület kontrollpontjait, kontrollpoligonját, vagy a felületet közelítő kitöltött poligonokat, azok határát, a felület határoló görbéit jeleníti meg, vagy a $([u\_\min, u\_\max] \cap [uknt[uorder-1], uknt[uknot\_count - uorder]]) \times ([v\_\min, v\_\max] \cap [vknt[vorder-1], vknt[vknot\_count - vorder]])$ tartomány fölötti paramétervonalait, pontjait, vagy normálisait ábrázolja.

*ucp_count* az $u$ irányú kontrollpontok száma,

*vcp_count* a $v$ irányú kontrollpontok száma,

*u_stride* $u$ irányban a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

*v_stride* $v$ irányban a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

**ctrlarray* a kontrollháló első pontjának a címe,

*type* paraméter lehetséges értékei: GL_MAP2_VERTEX_3 nem racionális B-spline felület esetén (a pontok három, vagyis Descartes-féle koordinátával adottak), GL_MAP2_VERTEX_4 racionális B-spline felület esetén (a pontok négy, azaz homogén koordinátával adottak),

*mode* a megjelenítés módját írja elő, lehetséges értékei:

GLUJ_CONTROL_POLYGON_MESH a kontrollhálót rajzolja meg,

GLUJ_CONTROL_POINT_MESH a kontrollpontokat jeleníti meg,

GLUJ_ISO_LINES a felületet paramétervonalaival ábrázolja, mégpedig az *ufixed_count* $\geq 0$ számú rögzített $u$ értékhez tartozó $v$ irányú, és a *vfixed_count* $\geq 0$ számú rögzített $v$ értékhez tartozó $u$ irányú paramétervonallal. **u_fixed* a rögzített $u$ értékeket tartalmazó vektor címe, **v_fixed* a rögzített $v$ értékeket tartalmazó vektoré. Ha *ufixed_count* $> 1$ és $*u\_fixed = NULL$, akkor a paramétertartományban egymástól egyenlő távolságra lévő *ufixed_count* darab $u$ értéket rögzít a függvény és az ezekhez tartozó $v$ irányú paramétervonalakat jeleníti meg. A *vfixed_count* és **v_fixed* paraméterek használata ezzel analóg.

GLUJ_ISO_LINES_WITH_OFFSET a felületet paramétervonalaival ábrázolja, mégpedig úgy, hogy a paramétervonalat a felületre merőlegesen eltolja. Az *ufixed_count, vfixed_count, *u_fixed és *v_fixed* paraméterek jelentése megegyezik a GLUJ_ISO_LINES módnál leírtakkal. $u$ irányú paramétervonalak esetén az értelmezési tartományt *ud* egyenlő részre osztja, az ezekhez tartozó felületi pontokat a pontbeli normális mentén *fl* mértékben eltolja és ezeket a pontokat köti össze egyenes szakaszokkal. A $v$ irányú paramétervonalak létrehozása ezzel analóg.

GLUJ_POINT_MESH a felületet a megadott $u$ és $v$ értékekhez tartozó pontokkal ábrázolja. Az *ufixed_count,* **u_fixed*, *vfixed_count*, **v_fixed* paraméterekre a GLUJ_ISO_LINES -nál leírtak érvényesek.

GLUJ_NORMAL_MESH a felületnek a megadott $u$ és $v$ értékekhez tartozó normálisait ábrázolja. Az *ufixed_count,* **u_fixed*, *vfixed_count*, **v_fixed* paraméterekre a GLUJ_ISO_LINES -nál leírtak érvényesek. A normálvektorok hosszát és irányítását az *fl* paraméter befolyásolja: $fl = 0$ esetén a normálvektorok hossza és irányítása a kiszámított

lesz, egyébként a normalizált normálvektor *fl*-szerese,

GLU_FILL a felületet közelítő kitöltött poligonokat jeleníti meg (mint a NURBS felületek esetén),

GLU_OUTLINE_POLYGON a felületet közelítő poligonok éleit jeleníti meg (mint a NURBS felületek esetén),

GLU_OUTLINE_PATCH a felület határoló görbéit jeleníti meg (mint a NURBS felületek esetén).

void **glujParamSurface** (void (**fn*)(float, float, float *), GLenum *mode*, GLint *ufixed_count*, GLfloat **u_fixed*, GLint *vfixed_count*, GLfloat **v_fixed*, GLfloat *u_min*, GLfloat *u_max*, GLfloat *v_min*, GLfloat *v_max*, int *ud*, int *vd*);

Az *fn* függvénnyel leírt paraméteres felületnek az $[u\_\min, u\_\max] \times [v\_\min, v\_\max]$ tartomány fölötti részét ábrázolja paramétervonalaival, pontjaival, háromszöghálóval, vagy kitöltött háromszögekkel.

*fn* a felület egy pontját kiszámító függvény. A függvény prototípusa:

void **fn**(float *u*, float *v*, float *p[3]*);

mely a felület $(u, v)$ paraméterű pontját kiszámítja és a $p$ vektorban visszaadja.

*mode* a megjelenítés módját írja elő, lehetséges értékei:

GLUJ_ISO_LINES a felületet paramétervonalaival ábrázolja, mégpedig az *ufixed_count* $\geq 0$ számú rögzített $u$ értékhez tartozó $v$ irányú, és a *vfixed_count* $\geq 0$, számú rögzített $v$ értékhez tartozó $u$ irányú paramétervonallal. **u_fixed* a rögzített $u$ értékeket tartalmazó vektor címe, **v_fixed* a rögzített $v$ értékeket tartalmazó vektoré. Ha *ufixed_count* $> 1$ és $*u\_fixed = NULL$, akkor a paramétertartományban egymástól egyenlő távolságra lévő *ufixed_count* darab $u$ értéket rögzít a függvény és az ezekhez tartozó $v$ irányú paramétervonalakat jeleníti meg. A megjelenítéshez a paramétervonal értelmezési tartományát *vd-1* egyenlő részre osztja, és az ezekhez tartozó görbepontokat egyenes szakaszokkal köti össze. A *vfixed_count*, **v_fixed* és *ud* paraméter használata ezzel analóg.

GLUJ_POINT_MESH a felületet a megadott $u$ és $v$ értékekhez tartozó pontokkal ábrázolja. A *ufixed_count,* **u_fixed*, *ud*, *vfixed_count*, **v_fixed, vd* paraméterekre a GLUJ_ISO_LINES -nál leírtak érvényesek.

GLUJ_TESS_POLYGON a felületet közelítő háromszöghálóval ábrázolja, melyet úgy állít elő, hogy a paramétertartományt $u$ irányban *ud*, $v$ irányban *vd* egyenlő részre osztja.

GLUJ_FILL_POLYGON a felületet közelítő kitöltött háromszöghálóval ábrázolja. A háromszögháló előállítása a GLUJ_TESS_POLYGON opciónál leírt módon történik.

void **glujCone** (void (**fn*)(float, float *), GLenum *mode*, GLint *ufixed_count*, GLfloat **u_fixed*, GLfloat **apex*, GLint *vfixed_count*, GLfloat **v_fixed*, GLfloat *u_min*, GLfloat *u_max*, GLfloat *len1*, GLfloat *len2*, int *ud*);

Az *fn* függvénnyel leírt vezérgörbéjű kúpot ábrázolja paramétervonalaival, pontjaival, háromszöghálóval, vagy kitöltött háromszögekkel.

*fn* az alapgörbe pontjait kiszámító függvény. A függvény prototípusa:

void **fn**(float *u*, float *p[3]*);

mely az alapgörbe $u$ paraméterű pontját kiszámítja és a $p$ vektorban visszaadja.

*mode* a megjelenítés módját írja elő, lehetséges értékei:

GLUJ_ISO_LINES a felületet paramétervonalaival ábrázolja, mégpedig az *ufixed_count* $\geq 0$ számú rögzített $u$ értékhez tartozó alkotóval, és a *vfixed_count* $\geq 0$, számú rögzített $v$ értékhez tartozó $u$ irányú paramétervonallal. **u_fixed* a rögzített $u$ értékeket tartalmazó vektor címe, **v_fixed* a rögzített $v$ értékeket tartalmazó vektoré. Ha *ufixed_count* $> 1$ és $*u\_fixed = NULL$, akkor a paramétertartományban egymástól egyenlő távolságra lévő *ufixed_count* darab $u$ értéket rögzít a függvény és az ezekhez tartozó alkotókat jeleníti meg. A *vfixed_count*, **v_fixed* és *ud* paraméter használata ezzel analóg.

GLUJ_POINT_MESH a felületet a megadott $u$ és $v$ értékekhez tartozó pontokkal ábrázolja. A *ufixed_count,* **u_fixed*, *ud*, *vfixed_count*, **v_fixed, vd* paraméterekre a GLUJ_ISO_LINES -nál leírtak érvényesek.

GLUJ_TESS_POLYGON a felületet közelítő háromszöghálóval ábrázolja, melyet úgy állít elő, hogy a paramétertartományt $u$ irányban *ud* egyenlő részre osztja és ezeket köti össze a csúcsponttal.

GLUJ_FILL_POLYGON a felületet közelítő kitöltött háromszöghálóval ábrázolja. A háromszögháló előállítása a GLUJ_TESS_POLYGON opciónál leírt módon történik.

*apex* a kúp csúcspontja

Az alkotók hossza a következő lesz: az alapgörbét tartalmazó kúpfélen a csúcsponttól az alapgörbéig terjedő irányított szakasz *len1*-szerese, a másikon pedig ugyanezen szakasz *len2*-szerese.

## 15.2. Pontok, érintők, normálvektorok

Az itt ismertetendő függvények nem rajzolnak, csak görbékre és felületekre illeszkedő pontokat, görbék érintőjét, valamint felületek normálvektorát számítják ki. Ezek a függvények tehát inkább a geometriai modellezéshez tartoznak, tapasztalatunk szerint azonban sok esetben szükség van rájuk az igényesebb szemléltetés során is.

int **glujPointOnNurbsCurve** (GLint *uknot_count*, GLfloat **uknot*, GLint *u_stride*, GLfloat **ctrlarray*, GLint *uorder*, GLfloat *u*, GLfloat **pt*, int *ncoord*);

NURBS görbe adott paraméterértékhez tartozó pontjának koordinátáit számítja ki.

*uknot_count* a csomóértékek száma (*uknot_count* = *uorder* + a kontrollpontok száma),

**uknot* az első csomóérték címe,

*u_stride* a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

**ctrlarray* a kontrollpoligon első pontjának a címe,

*uorder* a görbe rendje,

*u* a kiszámítandó ponthoz tartozó paraméterérték,

**pt* ebben a vektorban adja vissza a kiszámított pont koordinátáit,

*ncoord* a kontrollpontok koordinátáinak száma: 3, ha a pontok Descartes-féle koordinátával adottak; 4, ha homogén koordinátával.

A visszaadott érték negatív, ha a pont kiszámítása meghiúsult, egyébként 0.

int **glujPointOnBezierCurve** (GLint *cp_count*, GLint *u_stride*, GLfloat **ctrlarray*, GLfloat *u*, GLfloat **pt*, int *ncoord*);

Bézier-görbe adott paraméterértékhez tartozó pontjának koordinátáit számítja ki.
*cp_count* a kontrollpontok száma,
*u_stride* a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,
**ctrlarray* a kontrollpoligon első pontjának a címe,
*u* a kiszámítandó ponthoz tartozó paraméterérték,
**pt* ebben a vektorban adja vissza a kiszámított pont koordinátáit,
*ncoord* a kontrollpontok koordinátáinak száma: 3, ha a pontok Descartes-féle koordinátával adottak; 4, ha homogén koordinátával.
A visszaadott érték negatív, ha a pont kiszámítása meghiúsult, egyébként 0.

int **glujDerBsplineCurve** (GLint *uknot_count*, GLfloat **uknot*, GLint *u_stride*, GLfloat **ctrlarray*, GLint *uorder*, GLfloat *u*,GLfloat **p*, int *ncoord*);

B-spline görbe deriváltját (érintővektorát) számítja ki az adott pontban.
*uknot_count* a csomóértékek száma (*uknot_count* = *uorder* + a kontrollpontok száma),
**uknot* az első csomóérték címe,
*u_stride* a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,
**ctrlarray* a kontrollpoligon első pontjának a címe,
*uorder* a görbe rendje,
*u* az a paraméterérték, ahol a deriváltat ki kell számítani,
**p* ebben a vektorban adja vissza a kiszámított derivált koordinátáit,
*ncoord* a kontrollpontok koordinátáinak száma: 2 vagy 3.
A visszaadott érték negatív, ha a pont kiszámítása meghiúsult, egyébként 0.

int **glujDerNurbsCurve** (GLint *uknot_count*, GLfloat **uknot*, GLint *u_stride*, GLfloat **ctrlarray*, GLint *uorder*, GLfloat *u*,GLfloat **p*, int *ncoord*);

NURBS görbe deriváltját (érintővektorát) számítja ki az adott pontban.
*uknot_count* a csomóértékek száma (*uknot_count* = *uorder* + a kontrollpontok száma),
**uknot* az első csomóérték címe,
*u_stride* a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,
**ctrlarray* a kontrollpoligon első pontjának a címe,
*uorder* a görbe rendje,
*u* az a paraméterérték, ahol a deriváltat ki kell számítani,
**p* ebben a vektorban adja vissza a kiszámított derivált koordinátáit,
*ncoord* a kontrollpontok koordinátáinak száma: 3, ha a pontok Descartes-féle koordinátákkal adottak (vagyis B-spline görbéről van szó); 4, ha homogén koordinátákkal (vagyis a görbe racionális).
A visszaadott érték negatív, ha a pont kiszámítása meghiúsult, egyébként 0.

int **glujPointOnNurbsSurface** (GLint *uknot_count*, GLfloat **uknot*, GLint *vknot_count*, GLfloat **vknot*, GLint *u_stride*, GLint *v_stride*, GLfloat **ctrlarray*, GLint *uorder*, GLint *vorder*, GLfloat *u*, GLfloat *v*, GLfloat **pt*, int *ncoord*);

NURBS felület (*u, v*) paraméterű pontjának koordinátáit számítja ki.

*uknot_count* a felület *u* irányú csomóértékeinek száma (*uknot_count* = *uorder* + az *u* irányú kontrollpontok száma),

**uknot* az első *u* irányú csomóérték címe,

*vknot_count* az *v* paraméter csomóértékeinek száma (*vknot_count* = *vorder* + az *v* irányú kontrollpontok száma),

**vknot* az első *v* irányú csomóérték címe,

*u_stride u* irányban a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

*v_stride v* irányban a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

**ctrlarray* a kontrollháló első pontjának a címe,

*uorder* a felület *u* irányú rendje,

*vorder* a felület *v* irányú rendje,

*u, v* a kiszámítandó pont paraméterei,

**pt* ebben a vektorban adja vissza a kiszámított pont koordinátáit,

*ncoord* a kontrollpontok koordinátáinak száma: 3, ha a pontok Descartes-féle koordinátákkal adottak; 4, ha homogén koordinátákkal.

A visszaadott érték negatív, ha a pont kiszámítása meghiúsult, egyébként 0.

int **glujIsolineOnNurbsSurface** (GLint *uknot_count*, GLfloat **uknot*, GLint *vknot_count*, GLfloat **vknot*, GLint *u_stride*, GLint *v_stride*, GLfloat **ctrlarray*, GLint *uorder*, GLint *vorder*, int *ncoord*, int *dir*, float *val*, GLfloat **pv*);

NURBS felület *u* vagy *v* irányú paramétervonalának kontrollpontjait számítja ki.

*uknot_count* a felület *u* irányú csomóértékeinek száma (*uknot_count* = *uorder* + az *u* irányú kontrollpontok száma),

**uknot* az első *u* irányú csomóérték címe,

*vknot_count* az *v* paraméter csomóértékeinek száma (*vknot_count* = *vorder* + az *v* irányú kontrollpontok száma),

**vknot* az első *v* irányú csomóérték címe,

*u_stride u* irányban a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

*v_stride v* irányban a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

**ctrlarray* a kontrollháló első pontjának a címe,

*uorder* a felület *u* irányú rendje,

*vorder* a felület *v* irányú rendje,

*ncoord* a kontrollpontok koordinátáinak száma: 3, ha a pontok Descartes-féle koordinátákkal adottak; 4, ha homogén koordinátákkal.

*dir* a kiszámítandó paramétervonal irányát jelző szám. Ha értéke 1, akkor $v$ irányú paramétervonal kontrollpontjait számítja ki (azaz egy $u$ érték rögzített), ha értéke 2, akkor $u$ irányú paramétervonal kontrollpontjait számítja ki (azaz egy $v$ érték rögzített).

*val* a kiszámítandó paramétervonalhoz tartozó rögzített $u$ vagy $v$ paraméterérték, a *dir* paraméternek megfelelően.

**pv* ebben a vektorban adja vissza a kiszámított kontrollpontok koordinátáit.

A visszaadott érték negatív, ha a kontrollpontok kiszámítása meghiúsult, egyébként 0.

int **glujNormalOfNurbsSurface** (GLint *uknot_count*, GLfloat **uknot*, GLint *vknot_count*, GLfloat **vknot*, GLint *u_stride*, GLint *v_stride*, GLfloat **ctrlarray*, GLint *uorder*, GLint *vorder*, GLfloat *u*, GLfloat *v*, GLfloat **norm*, int *ncoord*);

NURBS felület (*u, v*) paraméterű pontjában a felület normálisának koordinátáit számítja ki.

*uknot_count* a felület $u$ irányú csomóértékeinek száma (*uknot_count* = *uorder* + az $u$ irányú kontrollpontok száma),

**uknot* az első $u$ irányú csomóérték címe,

*vknot_count* az $v$ paraméter csomóértékeinek száma (*vknot_count* = *vorder* + az $v$ irányú kontrollpontok száma),

**vknot* az első $v$ irányú csomóérték címe,

*u_stride* $u$ irányban a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

*v_stride* $v$ irányban a szomszédos kontrollpontok adatainak távolsága GLfloat-okban mérve,

**ctrlarray* a kontrollháló első pontjának a címe,

*uorder* a felület $u$ irányú rendje,

*vorder* a felület $v$ irányú rendje,

*u, v* a kiszámítandó pont paraméterei,

**norm* ebben a vektorban adja vissza a kiszámított normális koordinátáit,

*ncoord* a kontrollpontok koordinátáinak száma: 3, ha a pontok Descartes-féle koordinátákkal adottak; 4, ha homogén koordinátákkal.

A visszaadott érték negatív, ha a pont kiszámítása meghiúsult, egyébként 0.

# 16. fejezet

# Képességek engedélyezése, letiltása és lekérdezése

Az OpenGL-nek számos olyan képessége van, amelyet engedélyezhetünk (aktivizálhatunk), letilthatunk, illetve lekérdezhetjük az állapotát.

void **glEnable** (GLenum *cap*);

void **glDisable** (GLenum *cap*);

*cap* az engedélyezett, illetve letiltott képességet azonosító szimbolikus konstans.

Induláskor a GL_DITHER engedélyezett, az összes többi nem engedélyezett. A rendszer egy-egy állapotváltozóban tárolja ezen képességek kurrens beállításait, amiket a **glIsEnabled**() vagy **glGet***() függvényekkel lekérdezhetünk.

A *cap* paraméter lehetséges értékei és jelentésük:

- GL_ALPHA_TEST Alfa-vizsgálat,lásd a 11.2. szakaszt.
- GL_AUTO_NORMAL Normálisok automatikus létrehozása, ha a csúcspontok létrehozásához a GL_MAP2_VERTEX_3 vagy GL_MAP2_VERTEX_4 opciót használjuk, lásd a 14.2.szakaszt.
- GL_BLEND A fragmentum és a pixel színének alfa szerinti vegyítése (átlátszóság modellezése), lásd a 8.1. szakaszt.
- GL_CLIP_PLANE$i$ Az $i$-edik vágósík használata, lásd az 5.2. szakaszt.
- GL_COLOR_LOGIC_OP A fragmentum és a pixel színén logikai művelet végzése, lásd a 11.5.3. pontot.
- GL_COLOR_MATERIAL Az anyagtulajdonság hozzákapcsolása a rajzolási színhez, lásd a 6.4. szakaszt.
- GL_COLOR_TABLE A pixelek színének cseréje táblázat alapján.
- GL_CONVOLUTION_1D A pixelek egydimenziós konvolúciós szűrése.
- GL_CONVOLUTION_2D A pixelek kétdimenziós konvolúciós szűrése.

- GL_CULL_FACE Hátsó lapok elhagyása, lásd a 3.3.4. pontot.
- GL_DEPTH_TEST Láthatósági vizsgálat, lásd a 11.4. szakaszt.
- GL_DITHER Dithering, lásd a 11.5. szakaszt.
- GL_FOG Köd modellezése, lásd a 8.3. szakaszt.
- GL_HISTOGRAM Egy képen a színek eloszlásáról statisztika készítése.
- GL_INDEX_LOGIC_OP A fragmentum és a pixel színindexén logikai művelet végzése, lásd a 11.5.3. pontot.
- GL_LIGHT$i$ Az $i$-edik fényforrás használata, lásd a 6.2. szakaszt.
- GL_LIGHTING A megvilágítás használata, lásd a 6.5. szakaszt.
- GL_LINE_SMOOTH Szakaszok határának simítása, lásd a 8.2.1. pontot.
- GL_LINE_STIPPLE Vonaltípus használata, lásd a 3.3.2. pontot.
- GL_MAP1_COLOR_4 Az egydimenziós kiértékelő RGBA értékeket számoljon, lásd a 14.1. szakaszt.
- GL_MAP1_INDEX Az egydimenziós kiértékelő színindexeket számoljon, lásd a 14.1. szakaszt.
- GL_MAP1_NORMAL Az egydimenziós kiértékelő normálisokat számoljon, lásd a 14.1. szakaszt.
- GL_MAP1_TEXTURE_COORD_1 Az egydimenziós kiértékelő a textúra $s$ értékeit számolja, lásd a 14.1. szakaszt.
- GL_MAP1_TEXTURE_COORD_2 Az egydimenziós kiértékelő a textúra $s$ és $t$ értékeit számolja, lásd a 14.1. szakaszt.
- GL_MAP1_TEXTURE_COORD_3 Az egydimenziós kiértékelő a textúra $s$, $t$ és $r$ értékeit számolja, lásd a 14.1. szakaszt.
- GL_MAP1_TEXTURE_COORD_4 Az egydimenziós kiértékelő a textúra $s$, $t$, $r$ és $q$ értékeit számolja, lásd a 14.1. szakaszt.
- GL_MAP1_VERTEX_3 Az egydimenziós kiértékelő a csúcspont $x$, $y$, és $z$ értékeit számolja, lásd a 14.1. szakaszt.
- GL_MAP1_VERTEX_4 Az egydimenziós kiértékelő a csúcspont $x$, $y$, $z$ és $w$ értékeit számolja, lásd a 14.1. szakaszt.
- GL_MAP2_COLOR_4 A kétdimenziós kiértékelő RGBA értékeket számoljon, lásd a 14.2. szakaszt.
- GL_MAP2_INDEX A kétdimenziós kiértékelő színindexeket számoljon, lásd a 14.2. szakaszt.

- GL_MAP2_NORMAL A kétdimenziós kiértékelő normálisokat számoljon, lásd a 14.2. szakaszt.
- GL_MAP2_TEXTURE_COORD_1 A kétdimenziós kiértékelő a textúra $s$ értékeit számolja, lásd a 14.2. szakaszt.
- GL_MAP2_TEXTURE_COORD_2 A kétdimenziós kiértékelő a textúra $s$ és $t$ értékeit számolja, lásd a 14.2. szakaszt.
- GL_MAP2_TEXTURE_COORD_3 A kétdimenziós kiértékelő a textúra $s$, $t$ és $r$ értékeit számolja, lásd a 14.2. szakaszt.
- GL_MAP2_TEXTURE_COORD_4 A kétdimenziós kiértékelő a textúra $s$, $t$, $r$ és $q$ értékeit számolja, lásd a 14.2. szakaszt.
- GL_MAP2_VERTEX_3 A kétdimenziós kiértékelő a csúcspont $x$, $y$, és $z$ értékeit számolja, lásd a 14.2. szakaszt.
- GL_MAP2_VERTEX_4 A kétdimenziós kiértékelő a csúcspont $x$, $y$, $z$ és $w$ értékeit számolja, lásd a 14.2. szakaszt.
- GL_MINMAX Pixeltömbök színkomponensei minimumának, maximumának számítása.
- GL_NORMALIZE A normálisok automatikus normalizálása, lásd a 3.1. szakaszt.
- GL_POINT_SMOOTH Pont határának simítása, lásd a 3.3.1. pontot.
- GL_POLYGON_OFFSET_FILL Poligonok kitöltött megjelenítésekor, a fragmentumokhoz egy offset érték hozzáadása.
- GL_POLYGON_OFFSET_LINE Poligonok határának megjelenítésekor, a fragmentumokhoz egy offset érték hozzáadása.
- GL_POLYGON_OFFSET_POINT Poligonok csúcspontjainak megjelenítésekor, a fragmentumokhoz egy offset érték hozzáadása.
- GL_POLYGON_SMOOTH Poligon határának simítása, lásd a 3.3.3. pontot és a 8.2. szakaszt.
- GL_POLYGON_STIPPLE Poligon kitöltése mintával, lásd a 3.3.5. pontot.
- GL_POST_COLOR_MATRIX_COLOR_TABLE A színmátrixszal való transzformálás után táblázat szerinti színcsre.
- GL_POST_CONVOLUTION_COLOR_TABLE A konvolúciós szűrés után táblázat szerinti színcsre.
- GL_RESCALE_NORMAL A transzformációk után a **glNormal***() függvénnyel létrehozott normálisokat normalizálja, lásd a 3.1. szakaszt.
- GL_SEPARABLE_2D Két egydimenziós konvolúciós szűrőre szétválasztható kétdimenziós konvolúciós szűrő.

- GL_SCISSOR_TEST Kivágási vizsgálat, lásd a 11.1. szakaszt.
- GL_STENCIL_TEST Stencilvizsgálat, lásd a 11.3. szakaszt.
- GL_TEXTURE_1D Egydimenziós textúrázás, lásd a 13.1. szakaszt.
- GL_TEXTURE_2D Kétdimenziós textúrázás, lásd a 13.1. szakaszt.
- GL_TEXTURE_3D Háromdimenziós textúrázás, lásd a 13.1. szakaszt.
- GL_TEXTURE_GEN_Q A textúra $q$ koordinátáját a **glTexGen***() függvény szerint hozza létre, lásd a 13.11.3. pontot.
- GL_TEXTURE_GEN_R A textúra $r$ koordinátáját a **glTexGen***() függvény szerint hozza létre, lásd a 13.11.3. pontot.
- GL_TEXTURE_GEN_S A textúra $s$ koordinátáját a **glTexGen***() függvény szerint hozza létre, lásd a 13.11.3. pontot.
- GL_TEXTURE_GEN_T A textúra $t$ koordinátáját a **glTexGen***() függvény szerint hozza létre, lásd a 13.11.3. pontot.
- A következő függvénnyel azt tudhatjuk meg, hogy egy képesség engedélyezett-e.

GLboolean **glIsEnabled** (GLenum *cap*);

A függvény a GL_TRUE értéket adja vissza, ha a *cap* szimbolikus konstanssal azonosított képesség engedélyezett, egyébként a visszaadott érték GL_FALSE. A képességek alaphelyzetben nem engedélyezettek, kivéve a ditheringet. A *cap* paraméter lehetséges értékei:

GL_ALPHA_TEST, GL_AUTO_NORMAL, GL_BLEND, GL_CLIP_PLANE*i*, GL_COLOR_MATERIAL, GL_CULL_FACE, GL_DEPTH_TEST, GL_DITHER, GL_FOG, GL_LIGHT*i*, GL_LIGHTING, GL_LINE_SMOOTH, GL_LINE_STIPPLE, GL_LOGIC_OP, GL_MAP1_COLOR_4, GL_MAP1_INDEX, GL_MAP1_NORMAL, GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2, GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4, GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4, GL_MAP2_COLOR_4, GL_MAP2_INDEX, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1, GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3, GL_MAP2_TEXTURE_COORD_4, GL_MAP2_VERTEX_3, GL_MAP2_VERTEX_4, GL_NORMALIZE, GL_POINT_SMOOTH, GL_POLYGON_SMOOTH, GL_POLYGON_STIPPLE, GL_SCISSOR_TEST, GL_STENCIL_TEST, GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_GEN_Q, GL_TEXTURE_GEN_R, GL_TEXTURE_GEN_S, GL_TEXTURE_GEN_T.

# 17. fejezet

# Állapotváltozók értékének lekérdezése

Az itt ismertetett függvényekkel a globális változók, állapotváltozók kurrens értékeit kérdezhetjük le.

A **glGet***() függvénnyel az OpenGL állapotváltozóinak, globális paramétereinek az értékét kérdezhetjük le. *pname* a lekérdezendő paramétert azonosító szimbolikus konstans. A lekérdezett értéket a *params* címen kapjuk meg. Ha nem a változónak megfelelő függvénnyel kérdezzük le az értéket, a rendszer típuskonverziót hajt végre.

void **glGetBooleanv** (GLenum *pname*, GLboolean **params*);

void **glGetDoublev** (GLenum *pname*, GLdouble **params*);

void **glGetFloatv** (GLenum *pname*, GLfloat **params*);

void **glGetIntegerv** (GLenum *pname*, GLint **params*);

A *pname* paraméter a lekérdezendő változót azonosító szimbolikus konstans, a *params* címen pedig a lekérdezett értéket kapjuk vissza. A lekérdezendő érték típusának megfelelő függvényt kell használni. Ha a lekérdezendő adat és a meghívott függvény típusa különbözik, akkor a rendszer konverziót hajt végre.

Az alábbiakban felsoroljuk *pname* helyére írható szimbolikus konstansokat, valamint a hatásukra a *params* címen visszaadott értéket.

- GL_ACCUM_ALPHA_BITS A gyűjtőpufferben az alfa komponensek számára fenntartott bitsíkok száma, lásd a 10.4. szakaszt.
- GL_ACCUM_BLUE_BITS A gyűjtőpufferben a kék komponens számára fenntartott bitsíkok száma, lásd a 10.4. szakaszt.
- GL_ACCUM_CLEAR_VALUE A gyűjtőpuffer törlési színének RGBA komponensei, lásd a 10.4. szakaszt.
- GL_ACCUM_GREEN_BITS A gyűjtőpufferben a zöld komponens számára fenntartott bitek száma, lásd a 10.4. szakaszt.

- GL_ACCUM_RED_BITS A gyűjtőpufferben a vörös komponens számára fenntartott bitek száma, lásd a 10.4. szakaszt.
- GL_ACTIVE_TEXTURE_ARB Az aktív többszörös textúraegység.
- GL_ALIASED_POINT_SIZE_RANGE A kisimított pont alapelemek méretének minimuma és maximuma, lásd a 3.3.1. pontot.
- GL_ALIASED_LINE_WIDTH_RANGE A kisimított szakasz alapelemek vonalvastagságának minimuma és maximuma, lásd a 3.3.2. pontot.
- GL_ALPHA_BIAS A pixelmozgatások során az alfa komponensre alkalmazott eltolás.
- GL_ALPHA_BITS A színpufferek hány biten tárolják az alfa komponenst, lásd a 10.1. szakaszt.
- GL_ALPHA_SCALE A pixelmozgatások során az alfa komponensre alkalmazott skálázás.
- GL_ALPHA_TEST Az alfa-vizsgálat engedélyezett-e, lásd a 11.2. szakaszt.
- GL_ALPHA_TEST_FUNC Az alfa-vizsgálathoz használt függvény szimbolikus neve, lásd a 11.2. szakaszt.
- GL_ALPHA_TEST_REF Az alfa-vizsgálathoz használt referenciaérték, lásd a 11.2. szakaszt.
- GL_ATTRIB_STACK_DEPTH Az attribútumverem használatban lévő szintjeinek száma, lásd az 1. fejezetet.
- GL_AUTO_NORMAL A kétdimenziós kiértékelő automatikusan létrehozza-e a normálisokat, lásd a 14.2. szakaszt.
- GL_AUX_BUFFERS Az opcionális színpufferek száma, lásd a 10.1. szakaszt.
- GL_BLEND Az alfa szerinti színvegyítés engedélyezett-e, lásd a 8.1. szakaszt.
- GL_BLEND_COLOR Az alfa szerinti színvegyítéshez használt együtthatók RGBA komponensei.
- GL_BLEND_DST Alfa szerinti színvegyítéskor a cél kombináló tényezőjéhez használt függvény azonosítója, lásd a 8.1. szakaszt.
- GL_BLEND_EQUATION A forrás és a cél színeinek kombinálási módját azonosító szimbolikus konstans.
- GL_BLEND_SRC Alfa szerinti színvegyítéskor a forrás kombináló tényezőjéhez használt függvény azonosítója, lásd a 8.1. szakaszt.
- GL_BLUE_BIAS A pixelmozgatások során a kék színkomponensre alkalmazott eltolás.

- GL_BLUE_BITS A színpufferek hány biten tárolják a kék színkomponenst, lásd a 10.1. szakaszt.
- GL_BLUE_SCALE A pixelmozgatások során a kék színkomponensre alkalmazott skálázás.
- GL_CLIENT_ACTIVE_TEXTURE_ARB A kliens aktív többszörös textúraegysége.
- GL_CLIENT_ATTRIBE_STACK_DEPTH A kliens attribútumvermében a használatban lévő szintjeinek száma.
- GL_CLIP_PLANE*i* Az *i*-edik opcionális vágósík engedélyezett-e, lásd az 5.2. szakaszt.
- GL_COLOR_ARRAY A színtömb engedélyezett-e.
- GL_COLOR_ARRAY_SIZE A színtömb hány komponensben tárolja a színt.
- GL_COLOR_ARRAY_STRIDE A színtömbben az egymást követő színek távolsága.
- GL_COLOR_ARRAY_TYPE A színtömb milyen típusú változókban tárolja a színkomponenseket.
- GL_COLOR_CLEAR_VALUE A színpufferek törlési színének RGBA komponensei, lásd a 2.1. szakaszt.
- GL_COLOR_LOGIC_OP A fragmentumok színén a logikai műveletek engedélyezettek-e, lásd a 11.5.3. pontot.
- GL_COLOR_MATERIAL Az anyagtulajdonságnak a rajzolási színhez kapcsolása engedélyezett-e, lásd a 6.4. szakaszt.
- GL_COLOR_MATERIAL_FACE A poligonok melyik oldalának anyagtulajdonsága van hozzákapcsolva a rajzolási színhez, lásd a 6.4. szakaszt.
- GL_COLOR_MATERIAL_PARAMETER Melyik anyagtulajdonság van hozzákapcsolva a rajzolási színhez, lásd a 6.4. szakaszt.
- GL_COLOR_MATRIX A színmátrix-verem legfelső szintjén tárolt mátrix 16 eleme.
- GL_COLOR_MATRIX_STACK_DEPTH A színmátrix-verem szintjeinek maximális száma.
- GL_COLOR_TABLE A táblázat szerinti színcsere engedélyezett-e.
- GL_COLOR_WRITEMASK Az R, G, B, A szerinti színmaszkolás engedélyezett-e, lásd a 10.7. szakaszt.
- GL_CONVOLUTION_1D A pixelek egydimenziós konvolúciós szűrése engedélyezett-e.
- GL_CONVOLUTION_2D A pixelek kétdimenziós konvolúciós szűrése engedélyezett-e.

- GL_CULL_FACE A hátsó lapok elhagyása engedélyezett-e, lásd a 3.3.4. pontot.
- GL_CULL_FACE_MODE Felénk melyik oldalát mutató poligonokat kell elhagyni, lásd a 3.3.4. pontot.
- GL_CURRENT_COLOR A kurrens rajzolási szín RGBA komponensei, lásd a 4.1. szakaszt.
- GL_CURRENT_INDEX A kurrens színindex, lásd a 4.1. szakaszt.
- GL_CURRENT_NORMAL A kurrens normális $x$, $y$, $z$ komponensei, lásd a 3.1. szakaszt.
- GL_CURRENT_RASTER_COLOR A kurrens raszterpozíció színének RGBA komponensei, lásd a 9.2. szakaszt.
- GL_CURRENT_RASTER_DISTANCE A kurrens raszterpozíciónak a nézőponttól mért távolsága.
- GL_CURRENT_RASTER_INDEX A kurrens raszterpozíció színindexe, lásd a 9.2. szakaszt.
- GL_CURRENT_RASTER_POSITION A kurrens raszterpozíció $x$, $y$, $z$ és $w$ komponensei: $x$, $y$, $z$ ablakkoordináta-rendszerben, $w$ vágó koordinátákban, lásd a 9.2. szakaszt.
- GL_CURRENT_RASTER_POSITION_VALID A kurrens raszterpozíció érvényes-e, lásd a 9.2. szakaszt.
- GL_CURRENT_RASTER_TEXTURE_COORDS A kurrens raszterpozíció textúrájának $s$, $r$, $t$ és $q$ koordinátái.
- GL_CURRENT_TEXTURE_COORDS A kurrens $s$, $r$, $t$ és $q$ textúrakoordináták, lásd a 13.11. szakaszt.
- GL_DEPTH_BIAS A pixelmozgatások során alkalmazott eltolás.
- GL_DEPTH_BITS Hány bitben tárolja a rendszer a pixelek mélységét, lásd a 11.4. szakaszt.
- GL_DEPTH_CLEAR_VALUE A mélységpuffer törlési értéke, lásd a 10.5. szakaszt.
- GL_DEPTH_FUNC A mélységek összehasonlítására használt függvény szimbolikus azonosítója, lásd a 11.4. szakaszt.
- GL_DEPTH_RANGE Az ablakkoordináták intervalluma, lásd az 5.3. szakaszt.
- GL_DEPTH_SCALE A pixelmozgatások során a mélységre alkalmazott skálázás.
- GL_DEPTH_TEST A mélységvizsgálat engedélyezett-e, lásd a 11.4. szakaszt.
- GL_DEPTH_WRITEMASK A mélységpuffer írható-e, lásd a 10.7. szakaszt.

- GL_DITHER A dithering engedélyezett-e, lásd a 11.5.2. pontot.
- GL_DOUBLEBUFFER A dupla képsík használata (pl. animációhoz) engedélyezett-e, lásd a 10.1 szakaszt.
- GL_DRAW_BUFFER Az írható színpuffer szimbolikus azonosítója, lásd a 10.6 szakaszt.
- GL_EDGE_FLAG A határoló él jelzőjének értéke, lásd a 3.3.6. pontot.
- GL_EDGE_FLAG_ARRAY A határoló élek tömbben való tárolása engedélyezett-e.
- GL_EDGE_FLAG_ARRAY_STRIDE A határoló élek tömbjében az egymást követő értékek távolsága.
- GL_FEEDBACK_BUFFER_SIZE A visszacsatolási puffer mérete, lásd a 12.2 szakaszt.
- GL_FEEDBACK_BUFFER_TYPE A visszacsatolási puffer típusa, lásd a 12.2 szakaszt.
- GL_FOG A köd effektus engedélyezett-e, lásd a 8.3 szakaszt.
- GL_FOG_COLOR A köd színének RGBA komponensei, lásd a 8.3 szakaszt.
- GL_FOG_DENSITY A köd sűrűsége, lásd a 8.3 szakaszt.
- GL_FOG_END A lineáris köd-interpolációhoz az *end* paraméter, lásd a 8.3 szakaszt.
- GL_FOG_HINT A köd megvalósításának pontosságát azonosító szimbolikus konstans, lásd a 8.2 szakaszt.
- GL_FOG_INDEX A köd színindexe, lásd a 8.3 szakaszt.
- GL_FOG_MODE A köd kiszámításának módja, lásd a 8.3 szakaszt.
- GL_FOG_START A lineáris köd-interpolációhoz a *start* paraméter, lásd a 8.3 szakaszt.
- GL_FRONT_FACE Milyen irányítású poligonokat tekint a rendszer felénk nézőnek, lásd a 3.3.3. pontot.
- GL_GREEN_BIAS A pixelmozgatások során a zöld színkomponensre alkalmazott eltolás.
- GL_GREEN_BITS A színpufferek hány biten tárolják a zöld színkomponenst, lásd a 10.1. szakaszt.
- GL_GREEN_SCALE A pixelmozgatások során a zöld színkomponensre alkalmazott skálázás.
- GL_HISTOGRAM A színek eloszlásáról statisztika készítése engedélyezett-e.

- GL_INDEX_ARRAY A színindex tömb használata engedélyezett-e.
- GL_INDEX_ARRAY_STRIDE A színindex tömb egymást követő elemeinek távolsága.
- GL_INDEX_ARRAY_TYPE A színindex tömb elemeinek típusa.
- GL_INDEX_BITS A színpufferek hány biten tárolják a színindexet, lásd a 10.1. szakaszt.
- GL_INDEX_CLEAR_VALUE A színpuffer törlésére használt színindexe, lásd a 2.1 szakaszt.
- GL_INDEX_LOGIC_OP A színindexeken a logikai műveletek engedélyezettek-e, lásd a 11.5.3. pontot.
- GL_INDEX_MODE A rendszer színindex módban működik-e, lásd a 4 fejezetet.
- GL_INDEX_OFFSET A pixelmozgatások során a szín- és stencilindexhez hozzáadandó érték, lásd a 9.7. szakaszt.
- GL_INDEX_SHIFT A pixelmozgatások során a szín- és stencilindexek eltolása, lásd a 9.7. szakaszt.
- GL_INDEX_WRITEMASK A színindex puffer mely bitjei írhatók, lásd a 10.7. szakaszt.
- GL_LIGHT$i$ Az $i$-edik fényforrás engedélyezett-e, lásd a 6.2. szakaszt.
- GL_LIGHTING A megvilágítás engedélyezett-e, lásd a 6.1. szakaszt.
- GL_LIGHT_MODEL_AMBIENT A globális környezeti fény RGBA komponensei, lásd a 6.1. szakaszt.
- GL_LIGHT_MODEL_COLOR_CONTROL A tükrözött visszaverődés számításait a rendszer elkülöníti-e a normál megvilágítási számításoktól.
- GL_LIGHT_MODEL_LOCAL_VIEWER A tükrözött visszaverődési számításoknál a tényleges nézőpontot, vagy végtelen távoli nézőpontot használ a rendszer, lásd a 6.1. szakaszt.
- GL_LIGHT_MODEL_TWO_SIDE A poligonok különböző oldalainak különbözőek-e az anyagtulajdonságai, lásd a 6.1. szakaszt.
- GL_LINE_SMOOTH A szakasz alapelemek határának simítása engedélyezett-e, lásd a 8.2.1. pontot.
- GL_LINE_SMOOTH_HINT A szakasz alapelem határának simítása milyen minőségű, lásd a 8.2. szakaszt.
- GL_LINE_STIPPLE A vonaltípus használata engedélyezett-e, lásd a 3.3.2. pontot.
- GL_LINE_STIPPLE_PATTERN A vonalmintát leíró 16 bit, lásd a 3.3.2. pontot.

- GL_LINE_STIPPLE_REPEAT A vonalminta nagyítási tényezője, lásd a 3.3.2. pontot.
- GL_LINE_WIDTH A kurrens vonalvastagság, lásd a 3.3.2. pontot.
- GL_LINE_WIDTH_GRANULARITY A rendszer által támogatott vonalvastagságok közti különbség simított határú megjelenítésnél, lásd a 3.3.2. pontot.
- GL_LINE_WIDTH_RANGE Simított határú szakaszok vonalvastagságának minimuma és maximuma, lásd a 3.3.2. pontot.
- GL_LIST_BASE A **glCallLists**() végrehajtásakor használt offset, lásd a 7.5. szakaszt.
- GL_LIST_INDEX A feltöltés alatt álló display-lista azonosítója, lásd a 7.1. szakaszt.
- GL_LIST_MODE A feltöltés alatt álló display-lista létrehozásának módja, lásd a 7.1. szakaszt.
- GL_LOGIC_OP_MODE A színeken végrehajtandó logikai művelet kódja, lásd a 11.5.3. pontot.
- GL_MAP1_COLOR_4 Az egydimenziós kiértékelő színeket hoz-e létre, lásd a 14.1. szakaszt.
- GL_MAP1_GRID_DOMAIN A **glMapGrid1***() függvény értelmezési tartományának határai, lásd a 14.1. szakaszt.
- GL_MAP1_GRID_SEGMENTS A **glMapGrid1***() függvénnyel létrehozandó rácspontok száma, lásd a 14.1. szakaszt.
- GL_MAP1_INDEX Az egydimenziós kiértékelő színindexeket hoz-e létre, lásd a 14.1. szakaszt.
- GL_MAP1_NORMAL Az egydimenziós kiértékelő normálisokat hoz-e létre, lásd a 14.1. szakaszt.
- GL_MAP1_TEXTURE_COORD_1 Az egydimenziós kiértékelő $r$ textúrakoordinátákat hoz-e létre, lásd a 14.1. szakaszt.
- GL_MAP1_TEXTURE_COORD_2 Az egydimenziós kiértékelő $r$, $s$ textúrakoordinátákat hoz-e létre, lásd a 14.1. szakaszt.
- GL_MAP1_TEXTURE_COORD_3 Az egydimenziós kiértékelő $r$, $s$, $t$ textúrakoordinátákat hoz-e létre, lásd a 14.1. szakaszt.
- GL_MAP1_TEXTURE_COORD_4 Az egydimenziós kiértékelő $r$, $s$, $t$, $q$ textúrakoordinátákat hoz-e létre, lásd a 14.1. szakaszt.
- GL_MAP1_VERTEX_3 Az egydimenziós kiértékelő a csúcspontok $x$, $y$, $z$ koordinátáit hoz-e létre, lásd a 14.1. szakaszt.

- GL_MAP1_VERTEX_4 Az egydimenziós kiértékelő a csúcspontok $x$, $y$, $z$, $w$ koordinátáit hoz-e létre, lásd a 14.1. szakaszt.
- GL_MAP2_COLOR_4 A kétdimenziós kiértékelő színeket hoz-e létre, lásd a 14.2. szakaszt.
- GL_MAP2_GRID_DOMAIN A **glMapGrid2***() függvény $u$, illetve $v$ paraméterei értelmezési tartományának határai, lásd a 14.2. szakaszt.
- GL_MAP2_GRID_SEGMENTS A **glMapGrid2***() függvénnyel létrehozandó rácspontok száma $u$, illetve $v$ irányban, lásd a 14.2. szakaszt.
- GL_MAP2_INDEX A kétdimenziós kiértékelő színindexeket hoz-e létre, lásd a 14.2. szakaszt.
- GL_MAP2_NORMAL A kétdimenziós kiértékelő normálisokat hoz-e létre, lásd a 14.2. szakaszt.
- GL_MAP2_TEXTURE_COORD_1 A kétdimenziós kiértékelő $r$ textúrakoordinátákat hoz-e létre, lásd a 14.2. szakaszt.
- GL_MAP2_TEXTURE_COORD_2 A kétdimenziós kiértékelő $r$, $s$ textúrakoordinátákat hoz-e létre, lásd a 14.2. szakaszt.
- GL_MAP2_TEXTURE_COORD_3 A kétdimenziós kiértékelő $r$, $s$, $t$ textúrakoordinátákat hoz-e létre, lásd a 14.2. szakaszt.
- GL_MAP2_TEXTURE_COORD_4 A kétdimenziós kiértékelő $r$, $s$, $t$, $q$ textúrakoordinátákat hoz-e létre, lásd a 14.2. szakaszt.
- GL_MAP2_VERTEX_3 A kétdimenziós kiértékelő a csúcspontok $x$, $y$, $z$ koordinátáit hoz-e létre, lásd a 14.2. szakaszt.
- GL_MAP2_VERTEX_4 A kétdimenziós kiértékelő a csúcspontok $x$, $y$, $z$, $w$ koordinátáit hoz-e létre, lásd a 14.2. szakaszt.
- GL_MAP_COLOR A pixelmozgatások során a színek és színindexek táblázat alapján cserélendők-e, lásd a 9.7. szakaszt.
- GL_MAP_STENCIL A pixelmozgatások során a stencilindexek táblázat alapján cserélendők-e, lásd a 9.7. szakaszt.
- GL_MATRIX_MODE A kurrens mátrixverem, lásd az 5.4. szakaszt.
- GL_MAX_3D_TEXTURE_SIZE Az OpenGL implementáció által kezelhető 3D-s textúrák méretének durva becslése, lásd a 13.6. szakaszt.
- GL_MAX_ATTRIB_STACK_DEPTH A kliens attribútumvermében a szintek maximális száma.
- GL_MAX_CLIENT_ATTRIB_STACK_DEPTH Az attribútumverem szintjeinek maximális száma.

- GL_MAX_CLIP_PLANES Az opcionális vágósíkok maximális száma, lásd az 5.2. szakaszt.
- GL_MAX_COLOR_MATRIX_STACK_DEPTH A színmátrixok vermében a szintek maximális száma.
- GL_MAX_ELEMENTS_INDICES A tömbben tárolható csúcspontok indexei számának javasolt maximuma.
- GL_MAX_ELEMENTS_VERTICES A tömbben tárolható csúcspontok számának javasolt maximuma.
- GL_MAX_EVAL_ORDER A **glMap1***() és **glMap2***() kiértékelők által kezelt Bézier-görbék, és felületek rendjének maximuma, lásd a 14.1. és a 14.2.szakaszokat.
- GL_MAX_LIGHTS A fényforrások számának maximuma, lásd a 6.2. szakaszt.
- GL_MAX_LIST_NESTING A display-listák egymásba ágyazásának maximális szintje, lásd a 7.3. szakaszt.
- GL_MAX_MODELVIEW_STACK_DEPTH A nézőpont-modell transzformációk mátrixai számára fenntartott veremben a szintek számának maximuma.
- GL_MAX_NAME_STACK_DEPTH A kiválasztási névverem szintjeinek maximuma, lásd a 12.1. szakaszt.
- GL_MAX_PIXEL_MAP_TABLE A táblázat szerinti színcseréhez használható táblázatok számának maximuma, lásd a 9.8. szakaszt.
- GL_MAX_PROJECTION_STACK_DEPTH A vetítési transzformációk mátrixai számára fenntartott veremben a szintek számának maximuma.
- GL_MAX_TEXTURE_SIZE Az OpenGL implementáció által kezelhető textúrák méretének durva becslése, lásd a 13.2. szakaszt.
- GL_MAX_TEXTURE_STACK_DEPTH A textúrák mátrixai számára fenntartott veremben a szintek számának maximuma.
- GL_MAX_TEXTURE_UNITS_ARB A támogatott textúraegységek száma.
- GL_MAX_VIEWPORT_DIMS A képmező méreteinek maximuma, lásd az 5.3. szakaszt.
- GL_MINMAX A pixelekhez tárolt értékek minimumának és maximumának számítása engedélyezett-e.
- GL_MODELVIEW_MATRIX A kurrens nézőpont-modell transzformációs mátrix 16 eleme.
- GL_MODELVIEW_STACK_DEPTH A nézőpont-modell transzformációk mátrixai számára fenntartott veremben a szintek pillanatnyi száma.

- GL_NAME_STACK_DEPTH A kiválasztási névverem szintjeinek pillanatnyi száma, lásd a 12.1. szakaszt.
- GL_NORMAL_ARRAY A normálisok tömbökben való tárolása engedélyezett-e.
- GL_NORMAL_ARRAY_STRIDE A normálisok tömbjében az egymást követő adatok távolsága.
- GL_NORMAL_ARRAY_TYPE A normálisok tömbjének típusa.
- GL_NORMALIZE A normálisok automatikus normalizálása engedélyezett-e, lásd a 3.1. szakaszt.
- GL_PACK_ALIGNMENT A pixelek adatainak a memóriába írása során a byte-ok elrendezése, lásd a 9.6. szakaszt.
- GL_PACK_IMAGE_HIGHT A pixelek adatainak a memóriába írása során a kép magassága, lásd a 9.6. szakaszt.
- GL_PACK_LSB_FIRST Az 1 bit/pixel típusú adatoknak a memóriába írása során a memória byte-jainak legkisebb helyiértékű bitjébe kezdi-e az írást, lásd a 9.6. szakaszt.
- GL_PACK_ROW_LENGTH A pixelek adatainak a memóriában használt sorhossza, lásd a 9.6. szakaszt.
- GL_PACK_SKIP_IMAGES Az első pixelek adatainak a memóriába írása előtt kihagyandó képpixelek száma, lásd a 9.6. szakaszt.
- GL_PACK_SKIP_PIXELS Az első pixelek adatainak a memóriába írása előtt kihagyandó pixelek száma, lásd a 9.6. szakaszt.
- GL_PACK_SKIP_ROWS Az első pixelek adatainak a memóriába írása előtt kihagyandó pixelsorok száma, lásd a 9.6. szakaszt.
- GL_PACK_SWAP_BYTES Az pixelek adatainak a memóriába írása előtt a 2 vagy 4 byte-on tárolt adatok felcserélendők-e, lásd a 9.6. szakaszt.
- GL_PERSPECTIVE_CORRECTION_HINT A színeknek és textúráknak a perspektív torzítás miatt szükséges intepolációjához a pontosság, lásd a 8.2. szakaszt.
- GL_PIXEL_MAP_A_TO_A_SIZE Az pixelek adatainak a memóriába írásakor az alfából alfába típusú eltoláshoz használt táblázat mérete, lásd a 9.6. szakaszt.
- GL_PIXEL_MAP_B_TO_B_SIZE Az pixelek adatainak a memóriába írásakor a kékből kékbe típusú eltoláshoz használt táblázat mérete, lásd a 9.6. szakaszt.
- GL_PIXEL_MAP_G_TO_G_SIZE Az pixelek adatainak a memóriába írásakor a zöldből zöldbe típusú eltoláshoz használt táblázat mérete, lásd a 9.6. szakaszt.
- GL_PIXEL_MAP_I_TO_A_SIZE Az pixelek adatainak a memóriába írásakor az alfából indexbe típusú eltoláshoz használt táblázat mérete, lásd a 9.6. szakaszt.

- GL_PIXEL_MAP_I_TO_B_SIZE Az pixelek adatainak a memóriába írásakor az indexből kékbe típusú eltoláshoz használt táblázat mérete, lásd a 9.6. szakaszt.
- GL_PIXEL_MAP_I_TO_G_SIZE Az pixelek adatainak a memóriába írásakor az indexből zöldbe típusú eltoláshoz használt táblázat mérete, lásd a 9.6. szakaszt.
- GL_PIXEL_MAP_I_TO_I_SIZE Az pixelek adatainak a memóriába írásakor az indexből indexbe típusú eltoláshoz használt táblázat mérete, lásd a 9.6. szakaszt.
- GL_PIXEL_MAP_I_TO_R_SIZE Az pixelek adatainak a memóriába írásakor az indexből vörös típusú eltoláshoz használt táblázat mérete, lásd a 9.6. szakaszt.
- GL_PIXEL_MAP_R_TO_R_SIZE Az pixelek adatainak a memóriába írásakor a vörösből vörösbe típusú eltoláshoz használt táblázat mérete, lásd a 9.6. szakaszt.
- GL_PIXEL_MAP_S_TO_S_SIZE Az pixelek adatainak a memóriába írásakor a stencilből stencilbe típusú eltoláshoz használt táblázat mérete, lásd a 9.6. szakaszt.
- GL_POINT_SIZE A pont alapelem mérete, lásd a 3.3.1. pontot.
- GL_POINT_SIZE_GRANULARITY A rendszer által támogatott pontméretek közti különbség simított határú megjelenítésnél, lásd a 8.2.1. pontot.
- GL_POINT_SIZE_RANGE Simított határú pontok méretének minimuma és maximuma, lásd a 8.2.1. pontot.
- GL_POINT_SMOOTH A pont alapelemek határának simítása engedélyezett-e, lásd a 8.2.1. pontot.
- GL_POINT_SMOOTH_HINT A pont alapelem határának simítása milyen minőségű, lásd a 8.2. szakaszt.
- GL_POLYGON_MODE A poligonok két oldalának megjelenítési módja (oldalanként), lásd a 3.3.3. pontot.
- GL_POLYGON_OFFSET_FACTOR A poligon offsetjének skálázásához használt tényező.
- GL_POLYGON_OFFSET_UNITS A poligon raszterizálásakor a fragmentumhoz adandó érték.
- GL_POLYGON_OFFSET_FILL Kitöltött poligonok offsettel való megjelenítése engedélyezett-e.
- GL_POLYGON_OFFSET_LINE Határukkal reprezentált poligonok offsettel való megjelenítése engedélyezett-e.
- GL_POLYGON_OFFSET_POINT Csúcspontjaival reprezentált poligonok offsettel való megjelenítése engedélyezett-e.
- GL_POLYGON_SMOOTH A poligonok határának simítása engedélyezett-e, lásd a 8.2.2. pontot.

- GL_POLYGON_SMOOTH_HINT A poligon alapelem határának simítása milyen minőségű, lásd a 8.2. szakaszt.
- GL_POLYGON_STIPPLE A poligonok mintával való kitöltése engedélyezett-e, lásd a 3.3.5. pontot.
- GL_POST_COLOR_MATRIX_COLOR_TABLE A színmátrixszal való transzformálás után táblázat szerinti színcsre engedélyezett-e.
- GL_POST_COLOR_MATRIX_RED_BIAS A színmátrixszal való transzformálás után a fragmentumokra alkalmazandó, vörös szerinti eltolás, lásd a 9.7. szakaszt.
- GL_POST_COLOR_MATRIX_GREEN_BIAS A színmátrixszal való transzformálás után a fragmentumokra alkalmazandó, zöld szerinti eltolás, lásd a 9.7. szakaszt.
- GL_POST_COLOR_MATRIX_BLUE_BIAS A színmátrixszal való transzformálás után a fragmentumokra alkalmazandó, kék szerinti eltolás, lásd a 9.7. szakaszt.
- GL_POST_COLOR_MATRIX_ALPHA_BIAS A színmátrixszal való transzformálás után a fragmentumokra alkalmazandó, alfa szerinti eltolás, lásd a 9.7. szakaszt.
- GL_POST_COLOR_MATRIX_RED_SCALE A színmátrixszal való transzformálás után a fragmentumokra alkalmazandó, vörös szerinti skálázás tényezője, lásd a 9.7. szakaszt.
- GL_POST_COLOR_MATRIX_GREEN_SCALE A színmátrixszal való transzformálás után a fragmentumokra alkalmazandó, zöld szerinti skálázás tényezője, lásd a 9.7. szakaszt.
- GL_POST_COLOR_MATRIX_BLUE_SCALE A színmátrixszal való transzformálás után a fragmentumokra alkalmazandó, kék szerinti skálázás tényezője, lásd a 9.7. szakaszt.
- GL_POST_COLOR_MATRIX_ALPHA_SCALE A színmátrixszal való transzformálás után a fragmentumokra alkalmazandó, alfa szerinti skálázás tényezője, lásd a 9.7. szakaszt.
- GL_POST_CONVOLUTION_COLOR_TABLE A pixelek konvolúciós szűrése után táblázat szerinti színcsre engedélyezett-e.
- GL_POST_CONVOLUTION_RED_BIAS A pixelek konvolúciós szűrése után a fragmentumokra alkalmazandó, vörös szerinti eltolás, lásd a 9.7. szakaszt.
- GL_POST_CONVOLUTION_GREEN_BIAS A pixelek konvolúciós szűrése után a fragmentumokra alkalmazandó, zöld szerinti eltolás, lásd a 9.7. szakaszt.
- GL_POST_CONVOLUTION_BLUE_BIAS A pixelek konvolúciós szűrése után a fragmentumokra alkalmazandó, kék szerinti eltolás, lásd a 9.7. szakaszt.
- GL_POST_CONVOLUTION_ALPHA_BIAS A pixelek konvolúciós szűrése után a fragmentumokra alkalmazandó, alfa szerinti eltolás, lásd a 9.7. szakaszt.

- GL_POST_CONVOLUTION_RED_SCALE A pixelek konvolúciós szűrése után a fragmentumokra alkalmazandó, vörös szerinti skálázás tényezője, lásd a 9.7. szakaszt.
- GL_POST_CONVOLUTION_GREEN_SCALE A pixelek konvolúciós szűrése után a fragmentumokra alkalmazandó, zöld szerinti skálázás tényezője, lásd a 9.7. szakaszt.
- GL_POST_CONVOLUTION_BLUE_SCALE A pixelek konvolúciós szűrése után a fragmentumokra alkalmazandó, kék szerinti skálázás tényezője, lásd a 9.7. szakaszt.
- GL_POST_CONVOLUTION_ALPHA_SCALE A pixelek konvolúciós szűrése után a fragmentumokra alkalmazandó, alfa szerinti skálázás tényezője, lásd a 9.7. szakaszt.
- GL_PROJECTION_MATRIX A kurrens vetítési mátrix 16 eleme, lásd az 5.2. szakaszt.
- GL_PROJECTION_STACK_DEPTH A vetítési transzformációk mátrixai számára fenntartott veremben a szintek pillanatnyi száma.
- GL_READ_BUFFER Az olvasható színpuffert azonosító szimbolikus konstans.
- GL_RED_BIAS A pixelmozgatások során a vörös színkomponensre alkalmazott eltolás.
- GL_RED_BITS A színpufferek hány biten tárolják a vörös színkomponenst, lásd a 10.1. szakaszt.
- GL_RED_SCALE A pixelmozgatások során a vörös színkomponensre alkalmazott skálázás.
- GL_RENDER_MODE A megjelenítés módját azonosító szimbolikus konstans, lásd a 12. fejezetet.
- GL_RESCALE_NORMAL A transzformációk után a normálisok normalizálása engedélyezett-e.
- GL_RGBA_MODE Az OpenGL RGBA módban működik-e.
- GL_SCISSOR_BOX A kivágási vizsgálatokhoz használt ablak bal alsó sarkának koordinátái, szélessége, magassága, lásd a 11.1. szakaszt.
- GL_SCISSOR_TEST A kivágási vizsgálat engedélyezett-e, lásd a 11.1. szakaszt.
- GL_SELECTION_BUFFER_SIZE A kiválasztási puffer mérete, lásd a 12.1. szakaszt.
- GL_SEPARABLE_2D A szétválasztható kétdimenziós konvolúciós szűrő engedélyezett-e.
- GL_SHADE_MODEL A kurrens árnyalási modell szimbolikus konstansa, lásd a 4.2. szakaszt.

- GL_SMOOTH_LINE_WIDTH_RANGE Simított határú szakaszok vonalvastagságának minimuma és maximuma, lásd a 3.3.2. pontot.
- GL_SMOOTH_LINE_WIDTH_GRANUALITY A rendszer által támogatott vonalvastagságok közti különbség simított határú megjelenítésnél, lásd a 3.3.2. pontot.
- GL_SMOOTH_POINT_SIZE_RANGE Simított határú pontok méretének minimuma és maximuma, lásd a 8.2.1. pontot.
- GL_SMOOTH_POINT_SIZE_GRANUALITY A rendszer által támogatott pontméretek közti különbség simított határú megjelenítésnél, lásd a 8.2.1. pontot.
- GL_STENCIL_BITS A stencilpuffer bitsíkjainak száma, lásd a 11.3. szakaszt.
- GL_STENCIL_CLEAR_VALUE A stencilpuffer törlési értéke, lásd a 10.5. szakaszt.
- GL_STENCIL_FAIL Azt a műveletet azonosító szimbolikus konstans, amit a stencilvizsgálaton fennakadó fragmentumokra alkalmaz a rendszer, lásd a 11.3. szakaszt.
- GL_STENCIL_FUNC A stencilvizsgálathoz használt függvényt azonosító szimbolikus konstans, lásd a 11.3. szakaszt.
- GL_STENCIL_PASS_DEPTH_FAIL Azt a műveletet azonosító szimbolikus konstans, amit a stencilvizsgálaton átmenő, de a mélységvizsgálaton fennakadó fragmentumokra alkalmaz a rendszer, lásd a 11.3. szakaszt.
- GL_STENCIL_PASS_DEPTH_PASS Azt a műveletet azonosító szimbolikus konstans, amit a stencilvizsgálaton és a mélységvizsgálaton is átmenő fragmentumokra alkalmaz a rendszer, lásd a 11.3. szakaszt.
- GL_STENCIL_REF A stencilvizsgálathoz használt referenciaérték, lásd a 11.3. szakaszt.
- GL_STENCIL_TEST A stencilvizsgálat engedélyezett-e, lásd a 11.3. szakaszt.
- GL_STENCIL_VALUE_MASK A stencilvizsgálathoz használt maszk, lásd a 11.3. szakaszt.
- GL_STENCIL_WRITEMASK A stencilpuffer maszkolásához használt érték, lásd a 10.7. szakaszt.
- GL_STEREO A sztereoszkópikus (bicentrális) leképezést támogatja-e az implementáció.
- GL_SUBPIXEL_BITS Az alpixelek (amiket a rendszer az ablakkoordinátarendszerben a raszterizált alakzatok pozícionálásakor használ) felbontásához használt bitek becsült száma.
- GL_TEXTURE_1D Az egydimenziós textúraleképezés engedélyezett-e, lásd a 13.5. szakaszt.

- GL_TEXTURE_BINDING_1D A kurrens egydimenziós textúraobjektum azonosítója, lásd a 13.9.2. pontot.
- GL_TEXTURE_2D A kétdimenziós textúraleképezés engedélyezett-e, lásd a 13.2. szakaszt.
- GL_TEXTURE_BINDING_2D A kurrens kétdimenziós textúraobjektum azonosítója, lásd a 13.9.2. pontot.
- GL_TEXTURE_3D A háromdimenziós textúraleképezés engedélyezett-e, lásd a 13.6. szakaszt.
- GL_TEXTURE_BINDING_3D A kurrens háromdimenziós textúraobjektum azonosítója, lásd a 13.9.2. pontot.
- GL_TEXTURE_COORD_ARRAY A textúrakoordináták tömbjének használata engedélyezett-e.
- GL_TEXTURE_COORD_ARRAY_SIZE A textúrakoordináták tömbjének egy-egy elemében hány koordináta tárolható.
- GL_TEXTURE_COORD_ARRAY_STRIDE A textúrakoordináták tömbjében az egymást követő elemek távolsága.
- GL_TEXTURE_COORD_ARRAY_TYPE A textúrakoordináták tömbjében az elemek típusa.
- GL_TEXTURE_GEN_Q A textúrák $q$ koordinátájának automatikus létrehozása engedélyezett-e, lásd a 13.11.3. pontot.
- GL_TEXTURE_GEN_R A textúrák $r$ koordinátájának automatikus létrehozása engedélyezett-e, lásd a 13.11.3. pontot.
- GL_TEXTURE_GEN_S A textúrák $s$ koordinátájának automatikus létrehozása engedélyezett-e, lásd a 13.11.3. pontot.
- GL_TEXTURE_GEN_T A textúrák $t$ koordinátájának automatikus létrehozása engedélyezett-e, lásd a 13.11.3. pontot.
- GL_TEXTURE_MATRIX A kurrens textúramátrix 16 eleme.
- GL_TEXTURE_STACK_DEPTH A textúramátrix szintjeinek pillanatnyi száma.
- GL_UNPACK_ALIGNMENT Pixeleknek a memóriából való olvasásakor a byte-ok elrendezése, lásd a 9.6. szakaszt.
- GL_UNPACK_IMAGE_HIGHT Pixeleknek a memóriából való olvasásakor a kép magassága, lásd a 9.6. szakaszt.
- GL_UNPACK_LSB_FIRST Az 1 bit/pixel típusú adatoknak a memóriából való olvasása során a memória byte-jainak legkisebb helyiértékű bitjébe kezdi-e az írást, lásd a 9.6. szakaszt.

- GL_UNPACK_ROW_LENGTH Pixeleknek a memóriából való olvasásakor használt sorhossz, lásd a 9.6. szakaszt.
- GL_UNPACK_SKIP_IMAGES Az első pixelek adatainak a memóriából való olvasása előtt kihagyandó pixelek száma, lásd a 9.6. szakaszt.
- GL_UNPACK_SKIP_PIXELS Az első pixelek adatainak a memóriából való olvasása előtt kihagyandó pixelek száma, lásd a 9.6. szakaszt.
- GL_UNPACK_SKIP_ROWS Az első pixelek adatainak a memóriából való olvasása előtt kihagyandó pixelsorok száma, lásd a 9.6. szakaszt.
- GL_UNPACK_SWAP_BYTES Pixeleknek a memóriából való olvasásakor a 2 vagy 4 byte-on tárolt adatok felcserélendők-e, lásd a 9.6. szakaszt.
- GL_VERTEX_ARRAY A csúcspontok tömbjének használata engedélyezett-e.
- GL_VERTEX_ARRAY_SIZE A csúcspontokat hány koordinátával tároljuk a tömbben.
- GL_VERTEX_ARRAY_STRIDE A csúcspontok tömbjében az egymást követő elemek távolsága.
- GL_VERTEX_ARRAY_TYPE A csúcspontok tömbjének típusa.
- GL_VIEWPORT A képmező bal alsó sarkának koordinátái, szélessége és magassága, lásd az 5.3. szakaszt.
- GL_ZOOM_X Pixelmozgatáskor az $x$ irányú skálázási tényező, lásd a 9.4.2. pontot.
- GL_ZOOM_Y Pixelmozgatáskor az $y$ irányú skálázási tényező, lásd a 9.4.2. pontot.

void **glGetClipPlane** (GLenum *plane*, GLdouble **equation*);

A *plane* paraméterrel azonosított opcionális vágósík nézőpontkoordináta-rendszerbeli implicit alakjának együtthatóit adja vissza az *equation* címen. A *plane* paraméter értéke GL_CLIP_PLANE$i$ lehet, ($i = 0, \ldots,$ *vágósíkok száma*$-1$) (lásd az 5.2. szakaszt).

GLenum **glGetError** (void);

Visszaadja a kurrens hibakódot és törli a hibajelzőt (GL_NO_ERROR). A rendszer minden általa figyelt hibához egy numerikus kódot és egy szimbolikus konstanst rendel. Ha a rendszer hibát észlel, a hibajelzőhöz hozzárendeli a megfelelő értéket és mindaddig nem rögzít újabb hibát, míg a hibajelzőt ki nem olvassuk a **glGetError**() függvénnyel. A rendszer a hibát okozó függvény hívását figyelmen kívül hagyja, de nincs egyéb mellékhatása a hibának.

A következő hibajelzők definiáltak:

- GL_NO_ERROR A rendszer nem jegyzett fel hibát.

- GL_INVALID_ENUM Rossz szimbolikus konstanssal hívtuk meg valamelyik függvényt.
- GL_INVALID_VALUE Valamely függvény hívásánál olyan numerikus értéket adtunk meg, amelyik kívül esik az értelmezési tartományon.
- GL_INVALID_OPERATION A végrehajtandó művelet nem megengedett a rendszer pillanatnyi állapotában.
- GL_STACK_OVERFLOW A meghívott függvény verem-túlcsordulást okozna.
- GL_STACK_UNDERFLOW A meghívott függvény verem-alulcsordulást okozna.
- GL_OUT_OF_MEMORY Nincs elég memória a függvény végrehajtásához. A rendszer további működése nem meghatározható.
- GL_TABLE_TOO_LARGE A megadott táblázat nagyobb, mint az implementációban megengedett maximum.

void **glGetLightfv** (GLenum *light*, GLenum *pname*, GLfloat **params*);

void **glGetLightiv** (GLenum *light*, GLenum *pname*, GLint **params*);

A *light* azonosítójú fényforrás paramétereit adja vissza valós számként (az első hívási forma), vagy egész számként (a második hívási forma). A *light* paraméter értéke GL_LIGHT$i$ $0 \leq i <$ GL_MAX_LIGHTS lehet. A kívánt értéket a *params* címen kapjuk vissza.

Ha egész számként kérdezünk le színkomponenseket, akkor úgy konvertálja a komponenst, hogy a $[-1., 1.]$ intervallumot lineárisan leképezi a rendszer által ábrázolható legkisebb és legnagyobb egész által határolt intervallumra.

A *pname* értékei, és a hatásukra visszakapott értékek az alábbiak:

- GL_AMBIENT A fényforrás környezeti fényösszetevőjének RGBA komponenseit adja vissza.
- GL_DIFFUSE A fényforrás szórt fényösszetevőjének RGBA komponenseit adja vissza.
- GL_SPECULAR A fényforrás tükrözött fényösszetevőjének RGBA komponenseit adja vissza.
- GL_POSITION A fényforrás helyének $x, y, z, w$ koordinátái a nézőpontkoordinátarendszerben.
- GL_SPOT_DIRECTION A reflektorszerű fényforrás tengelyének iránya ( $x, y, z$).
- GL_SPOT_EXPONENT A reflektor fényerejének csökkenése.
- GL_SPOT_CUTOFF A reflektor kúpjának fél nyílásszöge.

- GL_CONSTANT_ATTENUATION A fény tompulásának konstans tagja.
- GL_LINEAR_ATTENUATION A fénytompulás lineáris tagjának együtthatója.
- GL_QUADRATIC_ATTENUATION A fénytompulás másodfokú tagjának együtthatója.

void **glGetMapdv** (GLenum *target*, GLenum *query*, GLdouble **values*);

void **glGetMapfv** (GLenum *target*, GLenum *query*, GLfloat **values*);

void **glGetMapiv** (GLenum *target*, GLenum *query*, GLint **values*);

A **glMap1***() és **glMap2***() függvényekkel létrehozott kiértékelők (Bézier-görbe, illetve felület megadások) paramétereit adja vissza a *values* címen, a hívott függvénynek megfelelően double, float vagy integer értékként.

A *target* paraméter lehetséges értékei:

GL_MAP1_COLOR_4, GL_MAP1_INDEX, GL_MAP1_NORMAL,
GL_MAP1_TEXTURE_COORD_1, GL_MAP1_TEXTURE_COORD_2,
GL_MAP1_TEXTURE_COORD_3, GL_MAP1_TEXTURE_COORD_4,
GL_MAP1_VERTEX_3, GL_MAP1_VERTEX_4, GL_MAP2_COLOR_4 ,
GL_MAP2_INDEX, GL_MAP2_NORMAL, GL_MAP2_TEXTURE_COORD_1,
GL_MAP2_TEXTURE_COORD_2, GL_MAP2_TEXTURE_COORD_3,
GL_MAP2_TEXTURE_COORD_4, GL_MAP2_VERTEX_3, GL_MAP2_VERTEX_4.

A *query* paraméterrel specifikáljuk a lekérdezendő adatokat. Értéke:

- GL_COEFF A kontrollpontokat adja vissza homogén koordinátákban $(x, y, z, w)$. Kétdimenziós kiértékelő (felület) esetén oszlopfolytonosan adja vissza a kontrollpontok tömbjét.
- GL_ORDER A görbe rendjét, illetve felület esetén az $u$ és $v$ irányú rendeket adja vissza.
- GL_DOMAIN A paraméter(ek) értelmezési tartományát (tartományait) adja vissza.

void **glGetMaterialfv** (GLenum *face*, GLenum *pname*, GLfloat **params*);

void **glGetMaterialiv** (GLenum *face*, GLenum *pname*, GLint **params*);

A *params* címen visszaadja a poligonok *face* oldalának a *pname* paraméterrel azonosított anyagtulajdonságait. Ha egész számként kérdezünk le színkomponenseket, akkor úgy konvertálja a komponenst, hogy a $[-1., 1.]$ intervallumot lineárisan leképezi a rendszer által ábrázolható legkisebb és legnagyobb egész által határolt intervallumra.

*face* értéke GL_FRONT vagy GL_BACK lehet.

*pname* értékei:

- GL_AMBIENT Az anyag környezeti fény visszaverődési együtthatójának RGBA komponenseit adja vissza.
- GL_DIFFUSE Az anyag szórt visszaverődési együtthatójának RGBA komponenseit adja vissza.
- GL_SPECULAR Az anyag tükrözött visszaverődési együtthatójának RGBA komponenseit adja vissza.
- GL_EMISSION Az anyag által kibocsátott fény RGBA komponenseit adja vissza.
- GL_SHININESS Az anyag ragyogási együtthatóját adja vissza.
- GL_COLOR_INDEXES Az anyag környezeti, szórt és tükrözött visszaverődési együtthatójának színindexeit adja vissza.

void **glGetPixelMapufv** (GLenum *map*, GLfloat **values*);

void **glGetPixelMapuiv** (GLenum *map*, GLuint **values*);

void **glGetPixelMapusv** (GLenum *map*, GLushort **values*);

A színkomponensek *map* táblázat szerinti módosításának beállításait adja vissza a *values* címen.

*map* lehetséges értékei:

GL_PIXEL_MAP_I_TO_I, GL_PIXEL_MAP_S_TO_S, GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, GL_PIXEL_MAP_I_TO_A, GL_PIXEL_MAP_R_TO_R, GL_PIXEL_MAP_G_TO_G, GL_PIXEL_MAP_B_TO_B, and GL_PIXEL_MAP_A_TO_A.

void **glGetPolygonStipple** (GLubyte **mask*);

A poligonok kitöltésére használt $32 \times 32$ mintát adja vissza a *mask* címen.

const GLubyte ***glGetString** (GLenum *name*);

Az implementációra vonatkozó szöveges információ címét adja vissza.
*name* lehetséges értékei:

- GL_VENDOR Az implementációért felelős cég.
- GL_RENDERER A megjelenítéshez használt hardver platform konfigurációjának neve.
- GL_VERSION Az implementált OpenGL verziószáma.
- GL_EXTENSIONS A támogatott OpenGL kiegészítések listája.

void **glGetTexEnvfv** (GLenum *target*, GLenum *pname*, GLfloat **params*);

void **glGetTexEnviv** (GLenum *target*, GLenum *pname*, GLint **params*);

A *pname* paraméterben specifikált textúrafüggvény paramétereit adja vissza a *params* címen.

A *target* paraméternek a GL_TEXTURE_ENV értéket kell adni. *pname* lehetséges értékei:

- GL_TEXTURE_ENV_MODE A kombinálás módját azonosító szimbolikus konstans.
- GL_TEXTURE_ENV_COLOR A textúrafüggvény színének RGBA komponensei.

void **glGetTexGendv** (GLenum *coord*, GLenum *pname*, GLdouble **params*);

void **glGetTexGenfv** (GLenum *coord*, GLenum *pname*, GLfloat **params*);

void **glGetTexGeniv** (GLenum *coord*, GLenum *pname*, GLint **params*);

A **glTexGen***() paranccsal megadott textúrakoordinátát létrehozó függvénynek a *pname* paraméterrel megadott jellemzőit adja vissza a *params* címen.

*coord* a textúra koordinátáját azonosítja, értéke GL_S, GL_T, GL_R, vagy GL_Q lehet. *pname* lehetséges értékei:

GL_TEXTURE_GEN_MODE A textúrakoordináták létrehozására használt függvény azonosítója.

GL_OBJECT_PLANE A referenciasík objektumkoordináta-rendszerbeli implicit alakjának együtthatói.

GL_EYE_PLANE A referenciasík nézőpontkoordináta-rendszerbeli implicit alakjának együtthatói.

void **glGetTexImage** (GLenum *target*, GLint *level*, GLenum *format*, GLenum *type*, GLvoid **pixels*);

A *pixels* címen a *target* paramétertől függően egy-, két- vagy háromdimenziós textúrát ad vissza. A *level* $\geq 0$ paraméterrel a lekérdezendő textúra részletességének szintjét kell megadni. *target* lehetséges értékei: GL_TEXTURE_1D, GL_TEXTURE_2D és GL_TEXTURE_3D. A *format* paraméterrel azt kell megadni, hogy a pixelek adatait milyen formában akarjuk visszakapni. Lehetséges értékei: GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR, GL_GBRA, GL_LUMINANCE, GL_LUMINANCE_ALPHA. A *type* paraméterrel azt adjuk meg, hogy a pixelek adatait milyen típusú adatként tároljuk, lehetséges értékei:

GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT,
GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2,
GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5,
GL_UNSIGNED_SHORT_5_6_5_REV, GL_UNSIGNED_SHORT_4_4_4_4,
GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1,

GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8,
GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2,
GL_UNSIGNED_INT_2_10_10_10_REV.

void **glGetTexParameterfv** (GLenum *target*, GLenum *pname*, GLfloat **params*);

void **glGetTexParameteriv** (GLenum *target*, GLenum *pname*, GLint **params*);

A *params* címen a *target* textúra *pname* paramétereit adja vissza. *target* lehetséges értékei: GL_TEXTURE_1D, GL_TEXTURE_2D és GL_TEXTURE_3D. *pname* lehetséges értékei:

GL_TEXTURE_MAG_FILTER , GL_TEXTURE_MIN_FILTER,
GL_TEXTURE_MIN_LOD, GL_TEXTURE_MAX_LOD,
GL_TEXTURE_BASE_LEVEL, GL_TEXTURE_MAX_LEVEL,
GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_WRAP_R,
GL_TEXTURE_BORDER_COLOR,GL_TEXTURE_PRIORITY,
GL_TEXTURE_RESIDENT.

# Tárgymutató

# Irodalomjegyzék


[1] Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F., Computer Graphics: principles and practice, second edition, Addison-Wesley, Reading, MA, 1990.

[2] Salomon, D., Computer graphics and geometric modeling, Springer, New York, 1999.

[3] Shreiner, D. (edt), OpenGL reference manual, The official reference document to OpenGL, Version 1.2, third edition, Addison-Wesley, Reading, MA, 1999.

[4] Szirmay-Kalos, L., Számítógépes grafika, ComputerBooks, Budapest, 1999.

[5] Szirmay-Kalos, L., Antal Gy., Csonka F., Háromdimenziós grafika, animáció és játékfejlesztés, ComputerBooks, Budapest, 2003.

[6] Woo, M., Neider, J., Davis, T., Shreiner, D., OpenGL programming guide, The official guide to learning OpenGL, Version 1.2, third edition, Addison-Wesley, Reading, MA, 1999.

[7] http://www.cs.unc.edu/~rademach/glui/

[8] http://www.opengl.org/

[9] http://www.sgi.com/software/opengl/

[10] http://www.trolltech.com/

[11] http://www.xmission.com/~nate/opengl.html