Írta:
**NAGY ANTAL**

# FEJLETT GRAFIKAI ALGORITMUSOK

**Egyetemi tananyag**


**2011**

LEKTORÁLTA: Dr. Szécsi László, Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Kar Irányítástechnika és Informatika Tanszék



TÁMOGATÁS:

Készült a TÁMOP-4.1.2-08/1/A-2009-0008 számú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus képzésekhez” című projekt keretében.

ISBN 978-963-279-516-4

KÉSZÜLT: a Typotex Kiadó gondozásában

FELELŐS VEZETŐ: Votisky Zsuzsa

AZ ELEKTRONIKUS KIADÁST ELŐKÉSZÍTETTE: Csépány Gergely László

KULCSSZAVAK:

grafikus csővezeték, OpenGL függvénykönyvtár, geometriai transzformációk, modellezés, árnyalás, textúrázás, ütközés detektálás, térbeli adatstruktúrák, realisztikus színtér.

ÖSSZEFOGLALÁS:

A jegyzet a Szegedi Tudományegyetem Természettudományi és Informatikai Karán, a programozó informatikus mesterszakon folyó Fejlett Grafikai Algoritmusok című kurzus tematikája alapján készült. A jegyzet első fejezetében bevezetésként az OpenGL alapok mellett a grafikus csővezeték fázisait is ismertetjük. Emellett külön kitérünk a programozható grafikus csővezeték bemutatására is. A következő fejezetben egy háromdimenziós objektum felépítése példáján keresztül mutatjuk be az alapvető modellezési szabályokat és más primitívek használatát. A negyedik fejezetben geometriai transzformációkkal foglalkozunk, ahol néhány speciális transzformációt is bemutatunk. A következő két fejezetben az árnyalással és az ahhoz szorosan kapcsolódó textúrázással foglalkozunk. Az ütközésdetektálás fejezetben néhány alap algoritmust mutatunk be. A nyolcadik fejezetben egyrészt olyan technikákat ismertetünk, amelyek az objektumok hatékony megjelenítését biztosítják, valamint olyan algoritmusokat ismertetünk, amelyek az objektumok felépítésének a kialakításában használhatóak. Az utolsó fejezetben olyan algoritmusokkal foglalkozunk, amelyek segítségével színterünket tehetjük valósághűbbé.

# Tartalomjegyzék

# 1. fejezet

# Bevezetés

Ez a jegyzet a Szegedi Tudományegyetem Természettudományi és Informatikai Karán, a programozó informatikus mester szakán folyó Fejlett Grafikai Algoritmusok című alap kurzus tematikája alapján készült.

A Fejlett Grafikai Algoritmusok kurzushoz az [1], [2] és [4] könyveket ajánljuk a hallgatóknak felhasználható irodalomként. A kurzus tematikája főleg ezen könyvek fejezetei alapján alakult ki. Megjegyezzük, hogy a kurzusnak számos előzménye volt, speciálkollégiumok és reguláris kurzusok, amelyek szintén befolyásolták a tematikát.

A Fejlett Grafikai Algoritmusok kurzust először a 2008-2009-es tanév tavaszi félévben hirdették meg a Szegedi Tudomány Egyetemen. Feltételeztük, hogy az MSc-s hallgatók a BSc-n előzőleg a Számítógépes Grafika kurzust már elvégezték. A tapasztalat azt mutatta, hogy az alapképzésben nem, vagy csak részlegesen szerezték meg ezeket az ismereteket a hallgatók. Ennek következtében a jegyzet írása során megpróbáltuk ezeket a hiányosságokat is pótolni.

A jegyzet első fejezetében bevezetésként az OpenGL alapok mellet a grafikus csővezeték fázisait is ismertetjük. Emellett külön kitérünk a programozható grafikus csővezeték bemutatására is. A következő fejezetben egy háromdimenziós objektum felépítés példáján keresztül mutatjuk be az alapvető modellezési szabályokat és más primitívek használatát. A negyedik fejezetben geometriai transzformációkkal foglalkozunk, ahol néhány speciális transzformációt is bemutatunk. A következő két fejezetben az árnyalással és az ahhoz szorosan kapcsolódó textúrázással foglalkozunk. A ütközésdetektálás fejezetben néhány alap algoritmust mutatunk be. A nyolcadik fejezetben egyrészt olyan technikákat ismertetünk, amelyek az objektumok hatékony megjelenítését biztosítják, valamint olyan algoritmusokat ismertetünk, amelyek az objektumok felépítésének a kialakításában használhatóak. Az utolsó fejezetben olyan algoritmusokkal foglalkozunk, amelyek segítségével színterünket tehetjük valósághűbbé.

A jegyzet terjedelmi okokból nem tartalmaz programozható grafikus hardver programozásával kapcsolatos ismereteket, bár a jegyzetben található algoritmusok (pl. árnyalás, környezeti leképezés, realisztikus színtér kialakítása) jól szemléltethetőek ezekkel az eszközökkel (Cg, GLSL és HLSL).

A jegyzet elkészítését a TÁMOP-4.1.2-08/1/A-2009-0008 pályázati azonosítójú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus

képzésekhez” című pályázati projekt támogatta. Köszönetet szeretnék mondani az Informatikai Tanszékcsoport vezetőségének, hogy lehetőséget biztosított számomra a jegyzet elkészítéséhez. Köszönettel tartozom Dr. Szécsi Lászlónak, a jegyzet lektorának, aki megjegyzéseivel, kiegészítéseivel tette teljesebbé a jegyzet végső változatát. Hálával tartozom családomnak, feleségemnek Áginak és fiaimnak Kristófnak, Simonnak és Vencelnek, hogy a jegyzet írása közben szeretetükkel és bizalmukkal támogattak. Végezetül köszönetet szeretnék mondani azoknak a hallgatóknak, akik érdeklődésükkel és segítségükkel motiváltak a munkám során.

Dr. Nagy Antal
Szeged, 2011. június 17.

# 2. fejezet

# A grafikus csővezeték és az OpenGL függvénykönyvtár

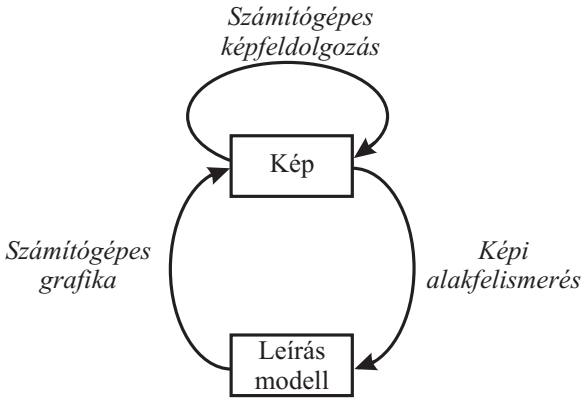
A digitális képek egyik első alkalmazása az 1920-as évek elején a Bartlane kábeles képátviteli rendszer volt, amikor is London és New York között egy tenger alatti kábelen küldtek át egy képet, melyet speciális nyomtató eszközzel kódoltak és állítottak helyre. A számítógéppel vezérelt képernyő csak 1950-ben jelent meg. Az első olyan számítógépek, amelyek képesek voltak modern interaktív grafikai tartalmakat megjeleníteni az 1960-as évek elején fejlesztették ki. A fejlődés lassú volt, mivel a hardver és a számítógépes erőforrások drágák voltak, valamint nehéz volt nagy programokat írni a megfelelő programozási és fejlesztési eszközök hiányában.

Az egyik jelentős mérföldkő az volt, amikor a személyi számítógépekhez megjelent az első videokártya. A videokártyák fejlődésével később megjelentek a pixelek megjelenítését támogató raszteres kijelzők. Kezdetekben minden számítást a számítógép központi egysége (CPU) végzett el, később ezt a feladatot a videokártyán elhelyezett grafikus feldolgozó egység (GPU) vette át. A hardver fejlődésével együtt láttak napvilágot az új szabványok, amelyek irányt mutattak egyrészt hardver, másrészt a grafikai szoftver fejlesztéseknek.

Jól látható, hogy a képek számítógépeken való megjelenítése már a kezdetek óta foglalkoztatja a szakembereket. A korai telegráfnyomtató után megjelenő vektor képernyő, majd az azt felváltó raszteres képmegjelenítéstől, mára a digitális képek feldolgozása, a képek alakfelismerése és a számítógépes grafika jelentős fejlődésen ment keresztül. Míg a számítógépes képfeldolgozással és a képi alakfelismeréssel foglalkozó algoritmusok bemenete egy digitális kép, addig a számítógépes grafikai alkalmazások egy matematikai leírás alapján állítanak elő egy képet (lásd 2.1. ábrát).

Egy 3 dimenziós (3D) színtér leírásához a színteret alkotó objektumokat primitívek segítségével építhetjük fel. Ezek a pontok, élek, illetve poligonok. Ezeknek a primitíveknek a szögpontjait vertexeknek nevezzük (lásd a 2.3. fejezetet). A leírás alapján adott sorrendben végrehajtott műveletek segítségével áll elő a számítógép raszteres képernyőjén a 3D-s színtér 2 dimenziós (2D) képe, ami lényegében egy 2D-s pixel tömb.

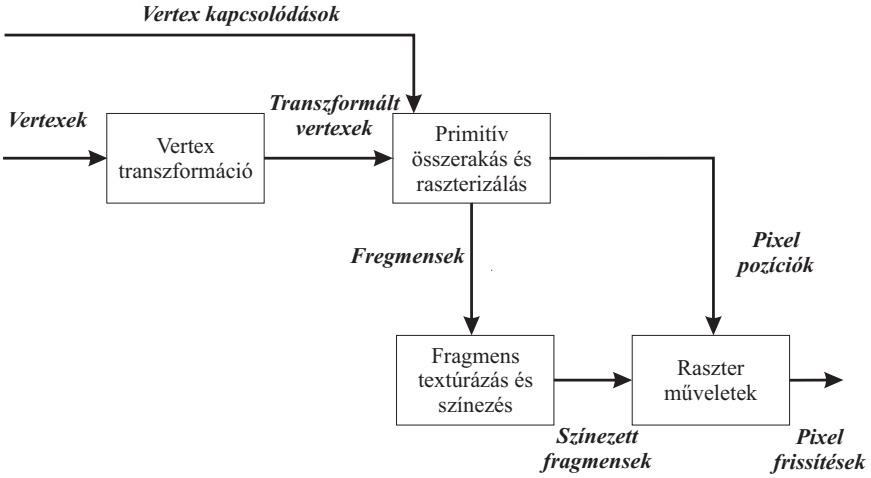
A műveletek adott sorrendjét grafikus csővezetéknek nevezzük. A csővezetékek között megkülönböztetünk rögzített műveleti sorrendű és programozható grafikus csővezetékeket. A következőkben ezeknek a grafikus csővezetékeknek a lépéseit fogjuk ismertetni röviden.




*2.1. ábra. A számítógépes grafika és a kapcsolódó tudományágak viszonya*

## 2.1. Rögzített műveleti sorrendű grafikus csővezeték

Egy 3D-s színtér primitíveinek kirajzolása, a kirajzolási paraméterek figyelembevételével, adott műveletek meghatározott sorrendjében történik. Mindegyik művelet az előző eredményét kapja meg bemeneti adatként és a feladat végrehajtása után továbbítja az eredményét az őt követő művelethez (lásd 2.2 ábrát).




*2.2. ábra. A grafikus csővezeték*

Egy 3D-s alkalmazás a geometriai primitívekhez tartozó vertexek kötegeit küldi a grafikai feldolgozó egységnek (GPU). Mindegyik vertexnek van pozíciója, de gyakran más attribútumok, nem geometriai információk is kapcsolódhatnak hozzájuk a megjelenítés módjától függően, mint például színinformáció, textúrakoordináták és normálvektorok. A színinformáció az objektum színét, a textúrakoordinátákkal megadott textúra adat az objektum mintázatát[1]

[1] Ebben az esetben ez szintén színinformációt jelent (lásd 6. fejezetet), amit általában egy 2D-s képben tárolunk.

határozza meg, a normálvektorok pedig az objektum árnyalásnál (lásd 5. fejezetet) játszanak fontos szerepet. A folyamat végén a képernyőn megjelenő pixelek egy adott méretű 2D-s tömbbe kerülnek, amit színpuffernek nevezünk.

### 2.1.1. Vertex transzformációk

Mindegyik vertexen matematikai műveletek sorozata hajtódik végre ebben a fázisban. Egyrészt meg kell határozni azt, hogy a primitívek szögpontjai hova fognak kerülni a képernyőn, amely alapján a raszterizáló egység a pixeleket fogja kiszínezni. A vertexek képernyőpozíciója mellett az adott vertexek színe és textúra-koordinátája is átadódik ebben a fázisban, amelyek a megvilágítás figyelembevételével szerepet játszanak a raszterizálás során kialakuló végső színértékek kiszámításában.

### 2.1.2. Primitív összerakás és raszterizálás

Az első lépésben a vertexek geometriai primitívekké állnak össze a vertexeket kísérő vertex kapcsolódási információk alapján. Ezek az információk azt határozzák meg, hogy a vertexek milyen geometriai primitíveket állítanak elő. Legegyszerűbb esetben háromszögek, vonalak vagy pontok sorozatát adják meg. Ezeket a primitíveket el kell vágni a nézeti csonka gúlának (a 3D-s színtér látható térfogata) valamint az alkalmazás által definiált vágósíkoknak megfelelően. A raszterizáló szintén eldobhat (hátsólap-eldobás/culling) poligonokat az elő- és hátlap információ miatt.

Azokat a poligonokat, amelyek túlélték a vágást és elő- illetve hátsólap-eldobást, raszterizálni kell. A raszterizálás egy olyan eljárás, amely meghatározza azon pixelek halmazát, amelyek a geometriai primitíveket lefedik. Poligonok, vonalak és pontok mindegyikét az adott típusú primitíveknek megfelelő szabályok szerint kell raszterizálni.

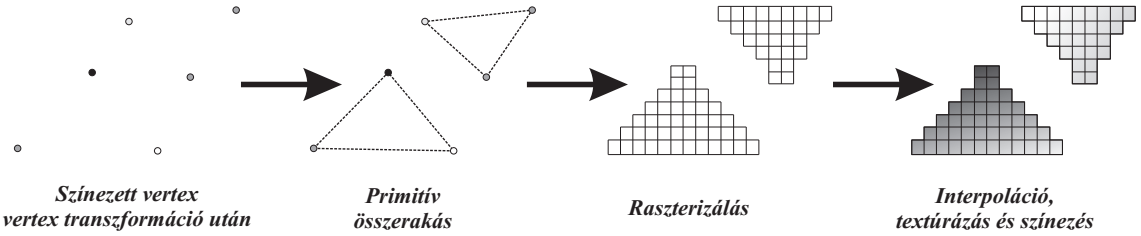
A raszterizálás eredményeként, a geometriai, szín, textúra adatok felhasználásával egy 2D-s színes képet kapunk. Ez a színes kép a primitíveket lefedő képpontok halmazaiból áll össze (lásd 2.3. ábrát). Mivel a primitívekhez tartozó pixelek nem biztos, hogy megjelennek a képernyőn (lásd a 2.1.4. fejezetet), ezért ezeket a potenciális pixeleket *fragmenseknek*[2] nevezzük azért, hogy megkülönböztessük őket az eredmény képen található végleges pixelektől. A raszterizálás eredményeként, a geometriai, szín, textúra adatok felhasználásával egy 2D-s színes képet kapunk.

### 2.1.3. Fragmens textúrázás és színezés

A primitívek raszterizálása után textúrázás és matematikai műveletek sorozata hajtódik végre mindegyik fragmens esetén, amelyek meghatározzák a végső szín értékét. A fragmensekhez a transzformált vertexekből származó interpolált szín információ mellett interpolált textúra-koordináták is kapcsolódnak. A textúra-koordináták segítségével nyerhetjük ki a textúrából a fragmeshez tartozó textúra elemet, melyet röviden *texelnek* nevezünk. Ezek után az adott texel és a fragmens színinformációinak a felhasználásával számíthatjuk ki a fragmens színét.

[2] Az eredeti kifejezés az angol fragment, ami töredéket jelent. Az elnevezése onnan ered, hogy a raszterizálás során a geometriai primitívek széttöredeznek pixel szintű fragmensekre, amelyek lefedik az adott primitívet.

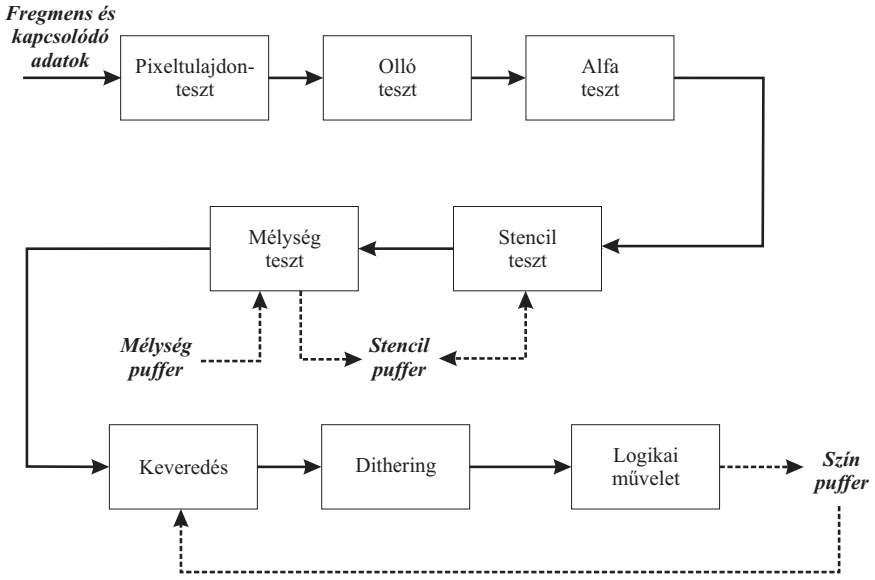
A 2.3 ábrán láthatóak a grafikus csővezeték eddig ismertetett, első három fázisának be- és kimeneti adatai két háromszög esetén. Jól látható, hogy alig néhány vertex adatból milyen sok fragmens jött létre.




*2.3. ábra. A grafikus csővezeték vizualizálása*

## 2.1.4. Rasztermüveletek

Az utolsó fragmensenkénti műveletként (lásd 2.4 ábra) a rasztermüveletek hajtódnak végre. Ezek a műveletek szintén szabványos részei a szabványos grafikai csővezetéknek.




*2.4. ábra. Standard OpenGL és Direct3D rasztermüveletek*

A raszter műveleteknél mindegyik fragmens esetén számos tesztet kell végrehajtani. Ezek a pixeltulajdon, olló, alfa, stencil és mélység tesztek. Az utóbbi három esetén a színpufferrel megegyező méretű alfa-, stencil- és mélységpuffert használunk a tesztek végrehajtására. A tesztek eredményétől függően alakul ki a fragmensek végső színe vagy mélység értéke, a pixel pozíciók és a pixelenkénti értékek, mint például a pixel mélység- és stencilértékei az adott pufferekben.

A következőkben röviden összefoglaljuk a rasztermüveletek fázisban végrehajtott teszteket.

- A pixeltulajdon-teszt meghatározza, hogy a képernyő adott pixelére az alkalmazás írhat-e. Amennyiben a pixel tulajdon teszt eredménye hamis, akkor ez azt jelenti, hogy például egy másik alkalmazásablak eltakarja a nézeti ablak egy részét. Ebben az esetben a fragmens nem rajzolódik ki.

- Az alkalmazás egy téglalapot definiálhat az ablak-nézetben, melyet olló téglalapnak nevezünk. Erre a téglalapra nézve korlátozhatjuk a kirajzolást. A téglalapon kívül eső fragmenseket eldobjuk.

- Ha a fragmensek túlélték az olló tesztet, akkor a fragmensek az alfa teszten mennek keresztül. A fragmens végső színének a kiszámításakor egy alkalmazás szintén meghatározhat alfa értéket, amit a vörös, zöld és kék komponensek mellett negyedik elemként adhatunk meg[3]. Ezt az értéket általában két különböző szín keveredés mértékének a meghatározására használjuk, amely lényegében a fragmenshez kapcsolódó átlátszóságot jelenti (lásd 5.4. fejezetet). Az alfa teszt összehasonlítja a fragmens végső alfa értékét egy, az adott alkalmazásban előre megadott értékkel. Attól függően, hogy az alkalmazás milyen relációt (kisebb, nagyobb, egyenlő) használ, az alfa teszt vagy igaz vagy hamis eredménnyel tér vissza. Utóbbi esetben a fragmens eldobódik[4].

- Stencil teszt során a fragmens pozíciójának megfelelő stencilpufferben lévő értéket és egy, az alkalmazás által megadott értéket hasonlít össze. A stencil teszt sikeres, ha az összehasonlítás eredménye igaz. Ellenkező esetben a fragmenst szintén eldobjuk. Az alkalmazásban meg lehet adni olyan műveleteket, amelyek akkor hajtódnak végre a stencil pufferen, amikor a stencil teszt sikeres vagy sikertelen. Továbbá ha a stencil teszt sikeres, akkor a következő pontban végrehajtott mélység teszt végeredményétől függően szintén meg lehet adni műveleteket, amelyek a stencil puffer értékeit befolyásolhatják.

- Az utolsó teszt a mélység teszt, ahol a fragmens mélység értékét hasonlítjuk össze a mélységpufferben tárolt értékével. Amennyiben a teszt sikeres, akkor a fragmens szín és mélység értékével frissítjük a színpuffert valamint a mélységpuffert, ami alapesetben azt jelenti, hogy a nézőponthoz közelebbi fragmens fog bekerülni a színpufferbe valamint a hozzátartozó mélység érték a mélységpufferbe.

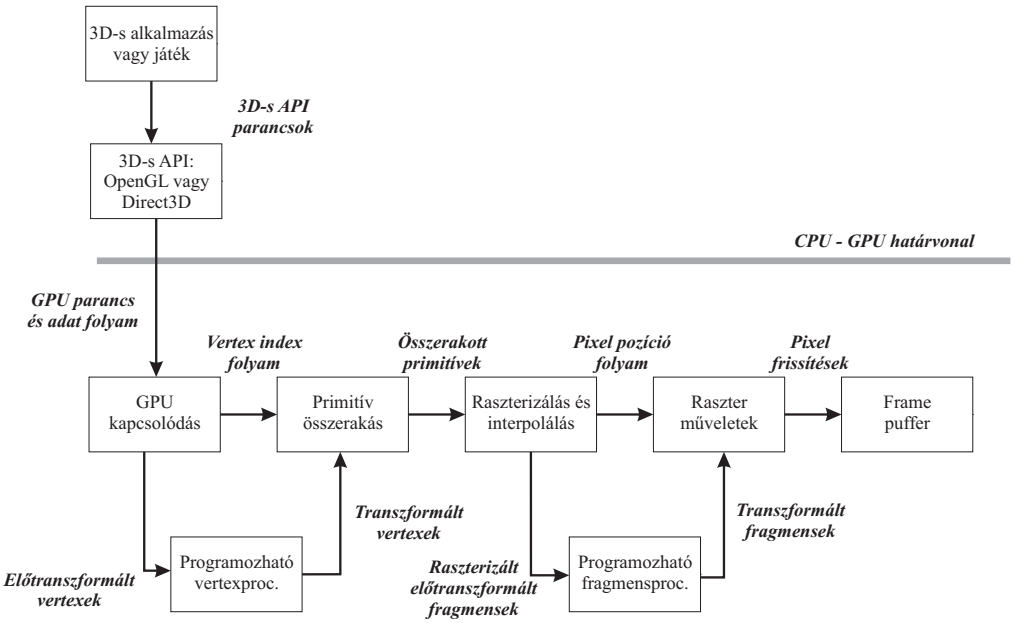
A tesztek után a keveredés művelet a végső fragmens és a neki megfelelő pixel színeket egyesíti. Végül a színpuffer író művelete kicseréli a pixel színét az előzőleg előállított kikevert színnel.

[3] Mivel az alfa értéket az RGB komponensek meghatározásakor a legtöbb esetben felhasználjuk, ezért az alfa értéket tekinthetjük egy negyedik színkomponensnek.

[4] Ez a teszt hasznos, amikor egy textúrának átlátszó pixelei vannak.

## 2.2. Programozható grafikus csővezeték

A grafikus hardver fejlődésével a GPU egyes részei programozható egységekkel bővültek, amely lehetővé teszik, hogy a felhasználók a grafikus csővezeték bizonyos fázisaiban programokat futtassanak. Ezzel a képességgel rugalmasabban lehet felhasználni a grafikus kártyákat. A 2.5 ábrán láthatóak a vertex és fragmensfeldolgozó egy programozható GPU csővezetékében. A 2.5 ábra több részletet mutat, mint a 2.2 ábra, de a legfontosabb az, hogy a vertex és fragmens feldolgozás egy-egy programozási egységgel bővült. A *programozható vertexprocesszor* az a hardveres egység, amely a vertexeken hajtja végre az előre megadott műveleteket, hasonlóan a *programozható fragmensprocesszor* pedig a fragmenseken végez műveleteket.



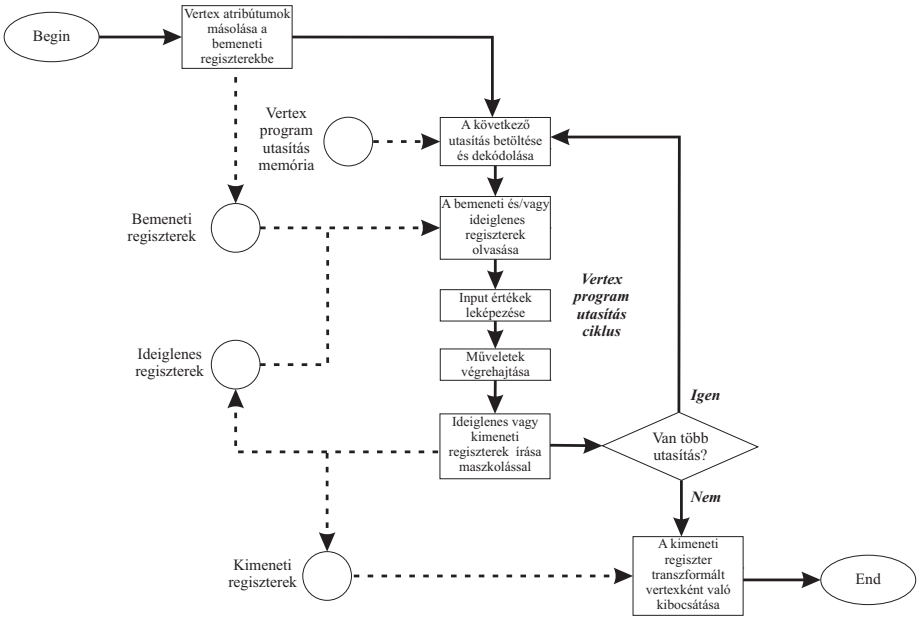

*2.5. ábra. A programozható grafikus csővezeték*

A következő két fejezetben, a teljesség igénye nélkül, bemutatjuk a programozható vertex és fragmens processzorok működési jellegzetességeit.

### 2.2.1. A programozható vertexprocesszor

A vertex feldolgozás az attribútumok (pl. pozíció, szín, textúra-koordináták stb.) vertex-processzorba való betöltésével kezdődik (lásd 2.6 ábra). A vertexek feldolgozása általában egy rövid *vertexprogram* (vertex-árnyaló) utasításainak a végrehajtásaival történik. Az utasítások különböző regiszterhalmazokat érnek el. A vertexattribútum-regiszterek csak olvashatóak, és alkalmazásspecifikus vertex információkat tartalmaznak (például pozíciót, normál- és színvektor értékeket). A feldolgozási folyamatban léteznek ideiglenes regiszterek, melyek olvashatóak és írhatóak is. Ezeket a regisztereket köztes eredmények kiszámítására lehet használni. A vertex program kimeneti regiszterekbe írja ki az eredményeket, és

ezek a regiszterek csak írhatóak. A vertex program befejeződésével a kimeneti regiszterek tartalmazzák az újonnan transzformált vertex adatokat. A primitív összerakás és raszterizálás után az interpolált értékek a fragmensprocesszor megfelelő regisztereibe íródnak.




*2.6. ábra. A programozható vertexprocesszor folyamatábrája*

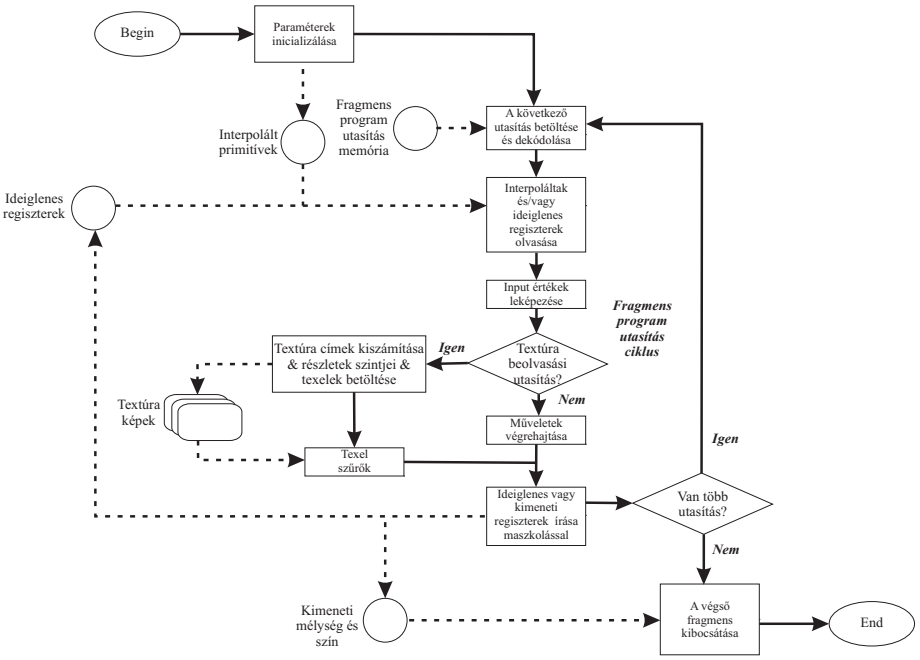
A legtöbb vertexfeldolgozás során a műveletek korlátozott palettáját használjuk. Szükség van lebegőpontos 2, 3 és 4 komponensű vektorokon végzett matematikai műveletekre, melyek magukba foglalják az összeadást, szorzást, szorzás-összeadást, skaláris szorzatot, minimum és maximum műveleteket. A hardveresen támogatott vektornegálás és a vektorok komponenseinek tetszőleges átrendezése az előbbi matematikai műveletek felhasználásával biztosítja a negálást, kivonást és a vektoriális szorzat műveleteket is. Kombinálva a reciprok és a reciprok négyzetgyök műveleteket a vektorszorzással és a skalárszorzattal, lehetővé teszi a vektor skalárral való osztás és a normalizálás műveletek elvégzését. Az exponenciális, logaritmikus és trigonometrikus közelítések a megvilágítási, köd és a geometriai számításokat könnyítik meg. A speciális műveletek a megvilágításhoz és csillapításhoz tartozó számítások elvégzését segítik.

További műveletek lehetővé teszik konstansok relatív címzését, valamint több modern vertexprocesszor is támogatja már a vezérlési szerkezeteket (elágazások, ciklusok).

### 2.2.2. A programozható fragmensprocesszor

A fragmensprocesszoroknak is hasonló műveletekre van szükségük, mint a vertexprocesszoroknak, de ezek a processzorok a textúraműveleteket is támogatják. Ezen műveletek segítségével a processzorok elérik a textúra képeket a textúra-koordinátákat felhasználva és utána visszaadják a textúra kép szűrt mintáját/pixelét.

A 2.7 ábrán jól látható, hogy hasonlóan a programozható vertexprocesszorhoz, az adatfolyam magába foglalja az utasítások sorozatának a végrehajtását a program befejeződéséig. A fragmensprocesszorban ismét találhatóak bemeneti regiszterek. A vertex attribútumokkal ellentétben, a fragmensprocesszor olvasható bemeneti regiszterei a fragmens primitív vertexenkénti paramétereiből származtatott, interpolált fragmensenkénti paramétereket tartalmaznak. Az írható/olvasható ideiglenes regiszterek közbenső értékeket tárolnak. A kiíró utasítások a csak írható regiszterekbe a fragmens szín és opcionálisan új mélység értékét írják ki. A fragmens program utasítások magukba foglalják a textúraolvasással kapcsolatos parancsokat is.



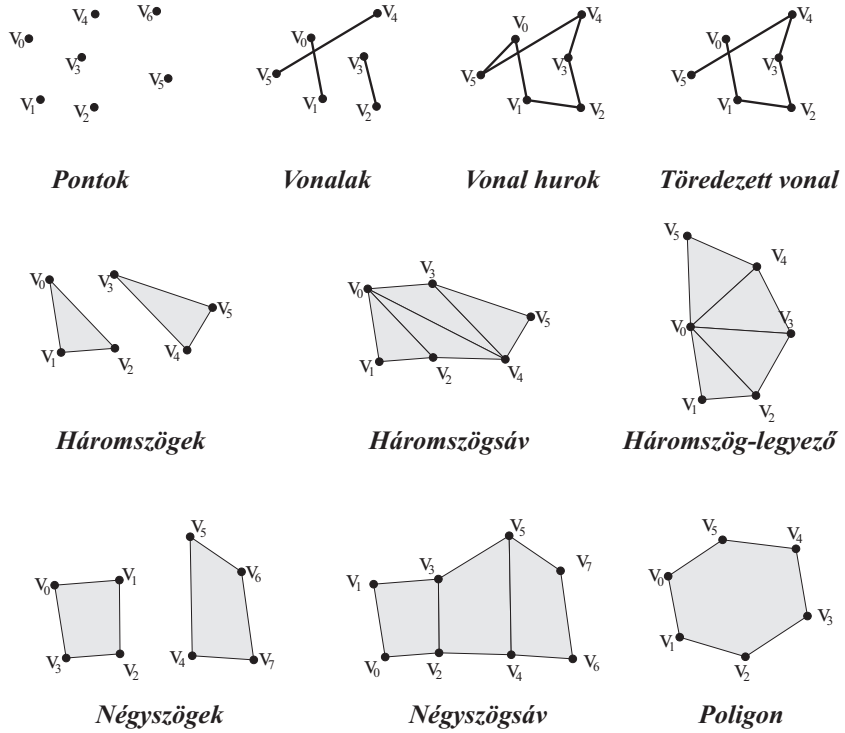

*2.7. ábra. A programozható fragmensprocesszor folyamatábrája*

## 2.3. Az OpenGL függvénykönyvtár

Az OpenGL lényegében egy hordozható, 3 dimenziós (3D) grafikus függvénykönyvtár, amely szoftveres felületet biztosít a számítógép grafikus hardveréhez. Több száz C függvényt és a hozzátartozó definíciókat tartalmaz. Így egy 3D-s színtér létrehozásához OpenGL függvény hívások sorozatát kell megadnunk. Ezek a parancsok egyrészt grafikus primitívek (lásd 2.8 ábra), mint például pontok, vonalak és poligonok kirajzolására szolgálnak.

A primitívek létrehozásához be kell vezetnünk a *vertex* fogalmát, amely segítségével az adott OpenGL primitív szögpontjait tudjuk megadni. Ezek a szögpontok a 2.8 ábrán látható $V_i$-vel jelölt 2- és 3D-s pozíciók, amelyek meghatározzák az adott primitív alakját és helyzetét az adott koordinátarendszerben.

Az OpenGL támogatja a megvilágítást, árnyalást, textúrázást, keveredést, átlátszóságot, animációt és sok más speciális hatást és képességet is. Mivel az OpenGL egy platform-




*2.8. ábra. OpenGL primitívek*

független függvénykönyvtár, ezért nem tartalmaz ablakkezelő, felhasználói interaktivitást és be- és kiviteli műveleteket végrehajtó függvényeket. Nincs OpenGL file formátum sem a modellek, sem pedig a virtuális környezet tárolására. Ezeket a programozónak kell létrehoznia, amennyiben magasabb szintű környezet kialakítására van szüksége.

Habár az OpenGL egy szabványos programozási függvénykönyvtár, ennek a könyvtárnak nagyon sok megvalósítása és verziója létezik. A legtöbb platformon az OpenGL-t, az OpenGL GLU segéd-függvénykönyvtárral (OpenGL Utility Library) együtt találhatjuk meg. Ez a segédkönyvtár olyan függvényeket tartalmaz, amelyek megszokott (néha azonban bonyolult) műveleteket hajtanak végre (például speciális mátrixműveletek vagy egyszerű típusú görbék vagy felületek támogatása).

Az OpenGL függvénykönyvtárat olyan emberek tervezték, akik nagyon sok tapasztalattal rendelkeztek a grafikus programozási és az alkalmazásprogramozási felületek, röviden API-k tervezésében. Néhány alapvető szabály alkalmazásával meghatározták a függvények és a változók elnevezési módját.

### 2.3.1. Adattípusok

Ahhoz, hogy egy OpenGL-es programot könnyedén tudjunk egyik platformról a másikra átvinni, szükség van arra, hogy az OpenGL saját adattípusokat definiáljon. Ezek az adattípusok normál C/C++ adattípusokra vannak leképezve. A különféle fordítók és környezetek által okozott problémák miatt célszerű ezeket az előredefiniált típusokat használni. Így nem

kell aggódni azon, hogy 32 bites vagy 64 bites rendszert használunk. A belső reprezentáció mindig ugyanaz lesz minden platformon. A következő táblázat (lásd 2.1) néhány ilyen adattípust ad meg a teljesség igénye nélkül.

| OpenGL adattípus | Belső reprezentáció | C adattípusként definiálva |
|---|---|---|
| `GLbyte` | 8 bites egész | `signed char` |
| `GLshort` | 16 bites egész | `short` |
| `GLint`, `GLsizei` | 32 bites egész | `long` |
| `GLfloat` | 32 bites lebegőpontos | `float` |
| `GLclampf` | pont | |
| `GLuint`, `GLenum`, `GLbitfield` | 32 bites előjel nélküli egész | `unsigned long` |

*2.1. táblázat. OpenGL adattípusok*

Mindegyik adattípus `GL`-lel kezdődik, ami az OpenGL-t jelöli. A legtöbb esetben a hozzákapcsolódó C adattípus (`byte`, `short`, `int`, `float` stb.) követi. Néhány adattípusnál az `u` jelöli az előjel nélküli típust. Vannak egészen beszédes nevek is, pl. `size`, ami egy érték hosszát vagy mélységét jelöli. A `clamp` megjelölés egy utalás arra, hogy az adott értéket a $[0.0, 1.0]$ intervallumba kell leképezni a későbbiek folyamán.

### 2.3.2. Függvényelnevezési szabályok

A legtöbb OpenGL függvény azt a konvenciót követi, ami megadja, hogy melyik függvénykönyvtárból való, és legtöbbször azt is meg lehet állapítani, hogy hány és milyen típusú argumentumot vár az adott függvény. Mindegyik függvénynek van egy alaptöve, amely megadja az OpenGL parancsot. Például a `glColor3f` alaptöve a `Color`. A `gl` prefix jelöli a gl könyvtárat és a `3f` suffix azt jelenti, hogy a függvény 3 lebegőpontos argumentumot vár. Az összes OpenGL függvény a következő formátumot követi:

<KÖNYVTÁR PREFIX><ALAP PARANCS><OPCIONÁLIS ARGUMENTUM SZÁM><OPCIONÁLIS ARGUMENTUM TÍPUS>

Előfordulhat, hogy abba a kísértésbe esünk, hogy olyan függvényeket használunk, melyeknek az argumentuma dupla pontosságú lebegőpontos típus, ahelyett hogy `float`-os típust választanánk bemenetnek. Ugyanakkor az OpenGL belül float-okat használ a `double` adattípus helyett[5]. Ráadásul a `double` dupla annyi helyet foglal, mint a `float`.

### 2.3.3. Platformfüggetlenség

Ahogy már korábban is említettük, az OpenGL nem tartalmaz olyan utasításokat, amelyek az operációs rendszerhez kapcsolódó feladatokat látnak el (pl. ablakkezelés, felhasználói interakciók kezelése stb.). Nem a grafikus kártyát kell megkérdezni arról, hogy a felhasználó leütötte-e az `Enter` billentyűt. Természetesen léteznek olyan platformfüggetlen absztrakciói

[5] Tulajdonképpen a grafikus hardver is float értékekkel dolgozik.

ennek a problémának, amelyeket nagyon jól lehet használni, de ezek a feladatok kívül esnek a grafikus renderelés témakörén.

**A GLUT használata**

A kezdetekben az OpenGL kiegészítő függvénykönyvtára az AUX volt. Ezt a könyvtárat váltotta ki a GLUT függvénykönyvtár a kereszt-platformos programozási példákhoz és szemléltetésre. A GLUT az *OpenGL utility toolkit* rövidítése (nem összetévesztendő a szabványos GLU - OpenGL segéd könyvtárral). Ez a függvénykönyvtár magába foglalja a pop-up menük használatát, más ablakok kezelését és még joystick támogatást is nyújt. A GLUT széles körben elérhető a legtöbb UNIX disztribúción (beleértve a Linux-ot is), natívan támogatja a Mac OS X, ahol az Apple tartja karban és fejleszti a könyvtárat. A Windows-os GLUT fejlesztését abbahagyták. Mivel a GLUT eredetileg nem rendelkezik nyílt forráskódú licenccel, egy új GLUT megvalósítás (*freeglut*) átvette annak a helyét.

A GLUT mindezek mellett kiküszöböli azt, hogy bármit is tudni kelljen az alap GUI (grafikus felhasználói felület) programozásáról adott platformon. A következő fejezetekben bemutatjuk azt, hogy hogyan lehet az adott platform specifikus GUI ismerete nélkül, a GLUT használatával egy OpenGL programot megvalósítani.

**Az első program**

Ahhoz, hogy jobban megértsük a GLUT könyvtárat, nézzünk meg egy egyszerű programot (lásd 2.1 kódrészlet), amely egyben az OpenGL használatba is bevezet minket. Ez a program nem sok mindent csinál. Létrehoz egy szabványos GUI ablakot az `Egyszeru` felirattal és tiszta kék kitöltési színnel.

```
1  #include <GL/glut.h>
2
3  // a színtér rajzolása
4  void RenderScene(void)
5  {
6  // Az aktuális törlő színnel való ablak törlés
7    glClear(GL_COLOR_BUFFER_BIT);
8
9  // Flush rajzoló parancs
10   glFlush();
11 }
12
13 // A renderelési állapotok beállítása
14 void SetupRC(void)
15 {
16 //A színpuffer törlőszínének a beállítása
17   glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
18 }
19
20 // A program belépési pontja
21 int main(int argc, char* argv[])
22 {
23   glutInit(&argc, argv);
```

```
24    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
25    glutCreateWindow("Egyszeru");
26    glutDisplayFunc(RenderScene);
27    SetupRC();
28    glutMainLoop();
29    return 0;
30 }
```

*2.1. kódrészlet. Egy egyszerű OpenGL program*

Ez az egyszerű program öt GLUT-os függvényt tartalmaz (`glut` prefix-szel) és három OpenGL függvényt (`gl` prefix-szel).

A 2.1 program egy file-t include-ol, amelyben az adott platformon betölti a további szükséges header-eket (pl. `GL/gl.h`-t, `GL/glu.h`-t vagy éppen a `Windows.h`-t az MS-Windows operációs rendszer esetén):

**A fejléc**

```
#include <GL/glut.h>
```

**A törzs**

Ugorjunk a C program `main` belépési pontjára:

```
int main(int argc, char* argv[])
{
  glutInit(&argc, argv);
```

A main függvény első parancsa a `glutInit`-et hívja, amely egyszerűen továbbítja a parancssori paramétereket és inicializálja a GLUT függvénykönyvtárat.

**Megjelenítési mód**

A következő lépésben meg kell mondanunk a GLUT könyvtárnak, hogy az ablak létrehozásakor milyen típusú megjelenítési módot használjon.

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
```

A flag-ek azt mutatják, hogy egy egyszeresen pufferelt (`GLUT_SINGLE`) ablakot használunk majd RGBA színmódban (`GLUT_RGBA`). Az egyszeres puffer azt jelenti, hogy minden rajzolási parancs (vagyis pontosabban, minden OpenGL csővezetékbe elküldött parancs) a megjelenített ablakban lesz végrehajtva. Egy alternatíva a duplán pufferelt ablak, ahol a rajzolási parancsok egy háttérben lévő pufferben történnek és aztán egy gyors csere művelet segítségével jelennek meg az ablakban[6]. Ez a módszer folytonos megjelenítést biztosít, ezért gyakran használják animációk készítése során. Igazából az összes többi példában duplán pufferelt ablakot fogunk használni. Az RGBA színmód azt jelenti, hogy a színek megadásához elkülönített piros, zöld, kék és alfa intenzitás komponenseket használunk. A másik, de manapság már igen elavult választási lehetőség az indexelt szín mód lenne, ahol színpaletta indexeket használunk a színek megadásakor.

[6] Az egyszeres puffer használata során a puffer törlési és az újrarajzolási parancsok egy pufferen hatódnak végre. Ennek az az eredménye, hogy a felhasználó az adott színteret az ablakban villódzva fogja látni.

**OpenGL ablak létrehozása**
A következő függvényhívással a GLUT könyvtár létrehoz egy ablakot a képernyőn, melynek a címsorán megjelenik az ”Egyszeru” felirat.

```
glutCreateWindow("Egyszeru");
```

**A megjelenítő callback függvény**
A következő GLUT-specifikus sor

```
glutDisplayFunc(RenderScene);
```

Ennek a függvénynek a meghívásával az előzőleg definiált `RenderScene` függvényt regisztrálja (2.1 kódrészlet 4-ik sora), mint megjelenítő callback függvényt. Ez azt jelenti, hogy amikor az ablakot újra kell rajzolni, akkor a GLUT mindig ezt a függvényt fogja meghívni. Ez például az ablak első megjelenítésekor vagy az ablak előtérbe helyezésekor történik meg. Ez a függvény tartalmazza lényegében az OpenGL-es renderelési függvény hívásainkat.

**A környezet beállítása és Rajt!**
A következő sor nem GLUT- és nem OpenGL specifikus függvény hívás.

```
SetupRC();
```

Ebben a függvényben (2.1 kódrészlet 14-ik sora) bármilyen OpenGL inicializálást végrehajthatunk a renderelés előtt. Az OpenGL kirajzolási paraméterek közül sokat elég egyszer beállítani, vagyis nincs szükség állandóan újra állítani minden egyes frame renderelése előtt.

Az utolsó GLUT-os függvényhívás a program végén található.

```
glutMainLoop();
```

Ez a függvény elindítja a GLUT keretrendszer eseménykezelőjét. A megjelenítési és más callback függvények definiálása után átadjuk a vezérlést a GLUT-nak. A `glutMainLoop` soha nem tér vissza a meghívása után a fő ablak bezárásáig, és csak egyetlen egyszer kell meghívni azt. Ez a függvény dolgozza fel az összes operációsrendszer-specifikus üzenetet, billentyűleütéseket stb. amíg a program be nem fejeződik.

**OpenGL grafikus függvényhívások**
A `SetupRC` függvény a következő egyszerű OpenGL függvény hívást tartalmazza:

```
glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
```

Ez a függvény beállítja az ablak törlésére használt színt. Tulajdonképpen a színpuffer inicializálásakor használt színt adjuk meg. A függvény prototípusa a következő:

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

A `GLclampf`, egy 0 és 1 közé leképzett `float`-os értéket jelent a legtöbb OpenGL megvalósításban. OpenGL-ben egy egyszerű szín a vörös, zöld és kék összetevők egy keverékeként van megadva. A lebegőpontos megadás miatt így végtelen sok potenciális színt keverhetünk ki ezeknek az értékeknek a segítségével. Természetesen az OpenGL veszi ezt a színértéket és belül átkonvertálja a legközelebbi lehetséges színre, amit az adott

videóhardver képes megjeleníteni. Például a vörös=0.0, zöld=0.0 és kék=0.0 esetén fekete színt, a vörös=1.0, zöld=1.0 és kék=1.0 beállítás esetén pedig fehér színt kapunk eredményül.

A `glClearColor` utolsó argumentuma az alfa komponens, amelyet keveredésre és speciális hatások elérésére (pl. átlátszóság) használunk. Az átlátszóság arra az objektum-tulajdonságra utal, hogy a fény áthalad rajta.

**A színpuffer törlése**

A RenderScene függvényben hajtódik végre a tényleges színpuffer törlése a

```
glClear(GL_COLOR_BUFFER_BIT);
```

utasítással, ami vagy csak bizonyos puffereket töröl vagy azok kombinációját. Több fajta puffert lehet használni az OpenGL-ben (pl. szín, mélység, stencil, összegző stb.), melyekről később még bővebben szót fogunk ejteni. A frame-puffer kifejezést a pufferek összességére fogjuk használni, hiszen ezeket lényegében együtt használjuk.

**Az OpenGL parancssor ürítése**

Az OpenGL parancsok és utasítások gyakran feltorlódnak, amíg azokat az OpenGL egyszerre fel nem dolgozza. A rövid 2.1 programban a `glFlush()` függvény meghívása egyszerűen azt mondja meg az OpenGL-nek, hogy nem kell további rajzoló utasításokra várnia, hanem folytassa az eddig beérkezetteknek a feldolgozását.

Az "`Egyszeru`" program nem a legérdekesebb OpenGL program, de bemutatja azt, hogy hogyan épül fel egy alap OpenGL program a GLUT segéd-függvénykönyvtár segítségével. A jegyzet ezen fejezetének nem célja az, hogy teljes részletességgel ismertesse az OpenGL és GLUT függvénykönyvtárak összes lehetőségét, bár a további fejezetekben igyekszünk az alaptechnikákat OpenGL példákon keresztül is bemutatni.

# 3. fejezet

# Geometriai transzformációk

A valóságban az igazi 3D-s látáshoz szükség van arra, hogy az adott objektumot mind a két szemmel nézzük. Mindegyik szem egy kétdimenziós képet érzékel, amelyek kissé eltérnek egymástól, mivel két különböző szögből néztük azokat. Ezután az agyunk összerakja ezt a két eltérő képet, amelyből egy 3D-s kép áll elő az agyunkban. A számítógép képernyője egy sík kép egy sík felületen. Így amit 3D-s számítógépes grafikának tartunk, az lényegében csupán az igazi 3D közelítése. Ezt a közelítést hasonlóan lehet elérni, amit a művészek a rajzokon látszólagos mélységgel évek óta használnak.

A geometriai transzformációk segítségével átformálhatjuk és animálhatjuk az objektumokat, fényforrásokat és kamerákat/nézőpontokat. A legtöbb grafika alkalmazásprogramozási felület (API) magába foglalja a mátrixműveleteket, amelyek lényegében a transzformációk matematikai megvalósításai.

A geometriai transzformációk gyakorlatilag az $\mathbf{x}' = \mathbf{A}\mathbf{x}$ mátrix-vektor szorzással[1] hajtható végre, ahol az $\mathbf{A}$ mátrix az adott transzformációs mátrix, $\mathbf{x}$ a transzformálandó oszlopvektor (például egy vertex pozíció) és az $\mathbf{x}'$ pedig az eredményt tartalmazó transzformált oszlopvektor (transzformált vertex pozíció).

A következőkben ismertetjük a transzformációs csővezetéket, valamint egyéb speciális transzformációs technikákat is bemutatunk.

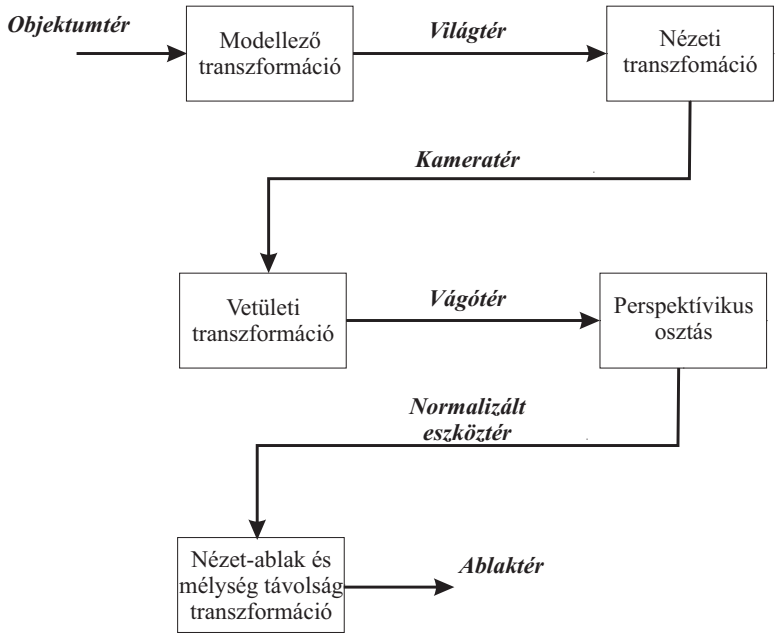
## 3.1. Transzformációs csővezeték

Ahogy azt az előzőekben láthattuk, a grafikus csővezeték célja az, hogy képeket hozzunk létre és megjelenítsük azokat a képernyőn. A grafikus csővezeték veszi az objektumokat, vagy színteret megjelenítő geometriai adatokat (rendszerint három dimenzióban) és kétdimenziós képet készít azokból. Az alkalmazások szolgáltatják a geometriai adatokat vertexek gyűjteményeként, amelyek poligonokat, vonalakat és pontokat alkotnak. Az eredmény általában egy megfigyelő vagy kamera szemszögéből látható képet ábrázol.

Ahogy a geometriai adat átfolyik a csővezetéken, a GPU vertex processzorai transzformálják az alkotó vertexeket egy vagy több koordináta-rendszerbe, amelyek bizonyos

[1] Egy mátrix és egy oszlopvektor szorzása esetén az adott vektor a mátrix jobb oldalán található. Így ha az $\mathbf{x}'$ transzformált vektort újból transzformálni akarjuk egy $\mathbf{B}$ transzformációs mátrixszal, akkor $\mathbf{x}'' = \mathbf{B}\mathbf{x}' = \mathbf{B}\mathbf{A}\mathbf{x}$ kaphatjuk meg a végleges eredményt.

célokat szolgálnak. A 3.1. ábra a transzformációk egy szokásos elrendezését ábrázolja. Az ábrán megjelöltük a transzformációk közötti koordináta-tereket, melyekbe a vertexek pozíciói kerülnek a transzformációk során.




*3.1. ábra. Koordinátarendszerek és transzformációk a vertex feldolgozás során*

### 3.1.1. Az objektumtér

Az alkalmazások egy koordinátarendszerben határozzák meg a vertex pozíciókat, melyet *objektumtérnek* vagy másképpen *modelltérnek* nevezünk. Amikor egy modellező elkészíti egy objektum 3D-s modelljét, akkor kiválaszt egy megfelelő orientációt, skálát és helyzetet, melyek segítségével elhelyezi a modellt alkotó vertexeket. Minden objektum saját objektumtérrel rendelkezik. Például egy henger esetén az objektumtér koordinátarendszer origója a henger lapjának a középpontjában lehet és a $z$ tengely lehet a henger szimmetria tengelye.

Mindegyik vertex pozícióját egy vektorként (pl. koordináta-hármasokkal) ábrázolhatjuk. A transzformációk segítségével az egyik térben lévő pozíciókat képezzük le egy másik térben lévő pozícióra. Mindegyik vertexhez hozzárendelhető egy objektumtérbeli felületi normálvektor is, ami az adott felületre merőleges egység hosszú vektor.

### 3.1.2. Homogén koordináták

Egy Descartes koordinátával megadott $(x, y, z)$ helyvektor egy speciális esete a négy-komponensű $(x, y, z, w)$ alaknak. Az ilyen típusú négy-komponensű pozíciót *homogén pozíciónak* nevezzük. Két $(x, y, z, w)$ és $(x', y', z', w')$ homogén koordináta-vektor abban az esetben egyezik meg, ha létezik egy olyan $h \neq 0$ érték, hogy $x' = hx$, $y' = hy$, $z' = hz$, $w' = hw$ egyenlőségek teljesülnek. A definícióból jól látszik, hogy egy pozícióhoz végtelen

sok homogén koordináta megadható. $w = 0$ homogén pozíciók esetében végtelenben lévő pontot értünk. Továbbá a $(0, 0, 0, 0)$ homogén koordináta nem megengedett. Amennyiben $w \neq 0$, akkor $(x, y, z, w)$ szokásos jelölése:

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right). \tag{3.1}$$

Egy Descartes $(x, y, z)$ koordinátához egy 1-est, negyedik komponensként hozzávéve adhatjuk meg a homogén alakot. Egy $(x, y, z, w)$ homogén pozíció homogén osztás (lásd 3.1. egyenletet) után ez $(x', y', z', 1)$ pozícióként fog megjelenni, továbbá ebből a homogén pozícióból az utolsó 1-es komponens elhagyásával kapjuk meg a homogén pozícióhoz tartozó Descartes koordinátát.

### 3.1.3. A világtér

Az objektumtéren az adott objektumok között térbeli viszonyok nincsenek definiálva. A *világtér* célja az, hogy valamilyen abszolút hivatkozását adjuk meg az összes objektumnak a színterünkön. Hogyan lehet általánosan a világteret tetszőlegesen megadni? Például dönthetünk úgy, hogy a szobában lévő objektumok a szoba közepéhez viszonyítva vannak elhelyezve, egy adott mértékegységben (pl. méter) és valamilyen irányítottsággal/orientációval (pl. az $y$-tengely pozitív része felfele mutat) megadva.

### 3.1.4. A modellező transzformáció

A modellező transzformáció segítségével tudjuk elhelyezni a világtéren az objektumtéren létrehozott modelleket. Például szükségünk lehet a 3D-s modell forgatására, eltolására és skálázására ahhoz, hogy a megfelelő pozícióban, méretben helyezzük el az általunk létrehozott világunkban. Például két szék objektum használhatja ugyanazt a 3D-s szék modellt, de különböző modellező transzformációk segítségével helyezzük el azokat egy szobában.

Az összes transzformációt egy $4 \times 4$-es mátrixszal ábrázolhatjuk matematikailag és a mátrix tulajdonságokat kihasználva több eltolást, forgatást, skálázást és vetítést kombinálhatunk össze mátrixok szorzásával egyetlen egy $4 \times 4$-es mátrixba. Amikor a mátrixokat ilyen módon fűzzük össze, akkor a kombinált mátrix szintén a megfelelő transzformációk kombinációit fejezi ki.

Amennyiben egy modellező transzformációt tartalmazó mátrixszal megszorzunk egy objektum téren lévő modell homogén vertex pozícióját (feltételezve, hogy a $w = 1$-gyel), akkor eredményképpen megkapjuk a világtérre transzformált pozícióját a modellnek.

### 3.1.5. A kameratér

A létrehozott színterünkre egy bizonyos nézőpontból (szem/kamera) tekintünk rá. A *kameratérként* ismert koordinátarendszerben a szem a koordinátarendszer origójában van. Követve a szabványos konvenciót, a képernyő felé nézünk, így a szem a $z$-tengely negatív része felé néz és a felfele mutató irány az $y$-tengely pozitív része felé mutat.

### 3.1.6. A nézeti transzformáció

Azt a transzformációt, ami a világtéren lévő pozíciókat a kameratérre viszi át *nézeti transzformációnak* nevezzük. Ezt a transzformációt is egy $4 \times 4$-es mátrixszal fejezhetjük ki. Egy tipikus nézeti transzformáció egy eltolás és egy elforgatás kombinációja, amely a világtéren lévő szem pozícióját a kameratér origójába viszi és ezután egy megfelelően végrehajtott kamera forgatást jelent. Ily módon a nézeti transzformáció meghatározza a kamera helyét és irányítottságát.

A nézeti transzformációs mátrix megadására a `gluLookAt` segédfüggvényt használhatjuk:
`void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,`
`GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx,`
`GLdouble upy, GLdouble upz),`
ahol a kamera koordinátáját, irányát és a felfele mutató irányát kell megadnunk. Ezt a függvényt kell először megadnunk, hogy az összes objektumra kifejtse a hatását a színtéren.

**Modell-nézeti mátrix**

A legtöbb megvilágítási és más árnyalási számítások esetén szükség van pozíció és felületi normál értékekre. Általában ezen számításokat hatékonyabban el lehet végezni a kameratérben vagy az objektumtérben. A világtér jól használható az alkalmazásokban a színtéren az objektumok általános térbeli viszonyainak a meghatározására, de nem különösebben hatékony a megvilágítási és más árnyalási számítások elvégzésére. Ezért általában a modellező és a nézeti transzformációkat egy modell-nézeti mátrixba vonjuk össze egy egyszerű mátrixszorzással.

OpenGL függvénykönyvtárban a Modell-nézeti (Modelview) transzformációs mátrixot `glMatrixMode(GL_MODELVIEW)` utasítással lehet kiválasztani/kijelölni. Ezután minden OpenGL mátrixutasítás a modelview mátrixra hat és a vertexek koordinátáit ezzel balról megszorozva kerülnek a kameratér megfelelő pozíciójába. A modell-nézeti mátrix inicializálását egy egységmátrix betöltésével tudjuk végrehajtani a `glLoadIdentity()` függvény meghívásával.

A modell-nézeti mátrix állandó inicializálása az objektumok elhelyezésekor nem mindig kívánatos. Gyakran előfordulhat, hogy az aktuális transzformációs állapotot el szeretnénk menteni, majd néhány objektum elhelyezése után visszaállítjuk azt. Ez a fajta megközelítés akkor a legkényelmesebb, amikor több objektum esetén a rájuk alkalmazandó transzformációs sorozatoknak a végrehajtási sorrendben utolsó elemei megegyeznek. Így a közös részekhez tartozó transzformációkat elég egyszer végrehajtani.

Az OpenGL függvénykönyvtár ennek az eljárásnak a megkönnyítésére egy *mátrixvermet* tart fenn a modell-nézeti mátrix tárolására. A mátrix verem hasonlóan működik a programozási vermekhez. Az aktuális mátrixot elmenthetjük/rátehetjük a verem tetejére a `glPushMatrix()` utasítással. A verem tetejéről az elmentett mátrixot a `glPopMatrix()` függvénnyel vehetjük le, amely egyben a globális modell-nézeti mátrixba be is tölti azt. A verem méretét a `glGet(GL_MAX_MODELVIEW_STACK_DEPTH)` függvény meghívásával kérdezhetjük le. Természetesen amennyiben túl sok elemet próbálunk a veremre helyezni, akkor verem túlcsordulás hibát, ha pedig egy üres veremből egy elemet próbálunk levenni, akkor verem alulcsordulás hibát kapunk.

**Eltolás**

Az egyik pozícióból egy másik pozícióba való mozgatást egy $\mathbf{T}$ mátrixszal írhatjuk le:

$$\mathbf{T}(\mathbf{t}) = \mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.2}$$

ahol a $t_x$, $t_y$ és $t_z$ értékek az adott tengelyek menti eltolások mértékét adják meg. $\mathbf{T}(\mathbf{t})$ alkalmazásával egy $\mathbf{p}$ pontot a $\mathbf{p}'$ pontba tolhatunk el az alábbiak szerint:

$$\mathbf{p}' = (p_x + t_x, p_y + t_y, p_z + t_z, 1). \tag{3.3}$$

Az inverz transzformációs mátrix $\mathbf{T}(\mathbf{t})^{-1} = \mathbf{T}(-\mathbf{t})$, ahol is a $\mathbf{t}$ vektort negáltuk.

A $\mathbf{T}(\mathbf{t})$ mátrixot OpenGL-ben a `glTranslatef(GLfloat x, GLfloat y, GLfloat z)` függvény meghívásával állíthatjuk elő, amelynek paraméterei rendre az iménti $t_x$, $t_y$ és $t_z$ értékek. A függvény meghívásával jobbról megszorozzuk az aktuális modell-nézeti mátrixot a létrehozott transzformációs mátrixszal. Az alábbi példa az $y$ tengely menti 5 egységnyi eltolásra mutat példát.

```
1 // y-tengely menti eltolás
2 glTranslatef(0.0f, 5.0f, 0.0f);
3
4 // Drótvázas kocka
5 glutWireCube(10.0f)
```

*3.1. kódrészlet. Eltolás 5 egységgel*

**Forgatás**

A forgatást bonyolultabb transzformációs sorozattal lehet leírni. A $\mathbf{R}_x(\phi)$, $\mathbf{R}_x(\phi)$ és $\mathbf{R}_z(\phi)$ forgatási mátrixokkal az adott objektumot az $x$, $y$ és $z$ tengelyek körül lehet elforgatni $\phi$ szöggel:

$$\mathbf{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.4}$$

$$\mathbf{R}_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.5}$$

$$\mathbf{R}_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.6}$$

Az $\mathbf{R}$ mátrixok bal-felső $3 \times 3$-as részmátrix fő diagonális elemeinek az összege állandó, függetlenül az adott tengelytől. Ez az összeg, amit a *mátrix nyomának*[2] nevezünk:

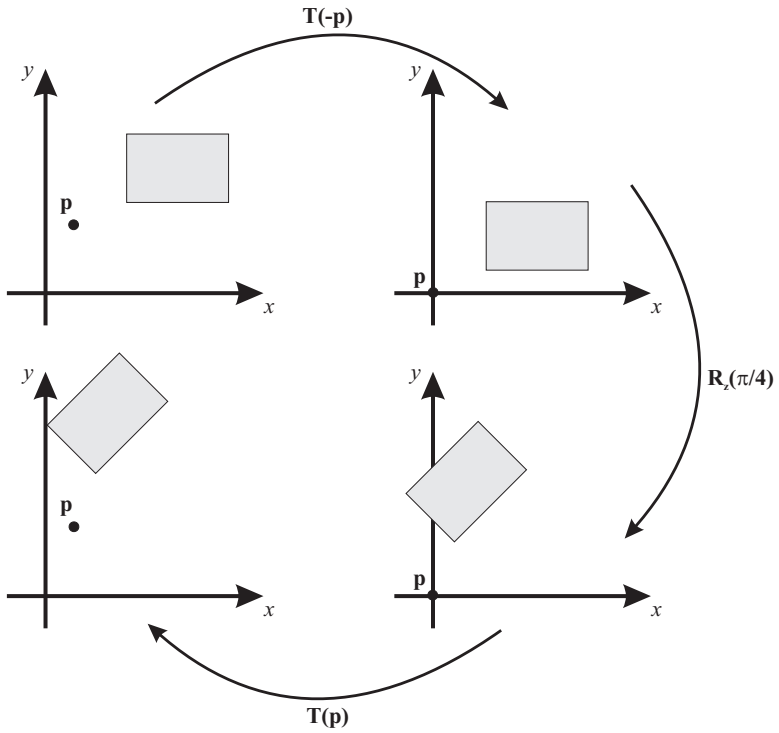
$$\mathrm{tr}(\mathbf{R}) = 1 + 2\cos\phi. \tag{3.7}$$

Mindegyik forgatási mátrix ortogonális, ami azt jelenti, hogy a forgatási mátrix inverze megegyezik a mátrix transzponáltjával, ami az elemek főátlóra való tükrözésével kapható meg ($\mathbf{R}^{-1} = \mathbf{R}^T$). Továbbá az is igaz, hogy az inverz forgatási mátrix előáll $\mathbf{R}_i^{-1}(\phi) = \mathbf{R}_i(-\phi)$ módon is, ahol az $i$ index az adott tengelyt jelöli[3]. Azt is könnyen bizonyíthatjuk, hogy a forgatási mátrix determinánsa mindig eggyel egyenlő.

A $z$ tengellyel párhuzamos, $p$ ponton átmenő tengely körüli forgatást az alábbi módon adhatunk meg:

$$\mathbf{X} = \mathbf{T}(\mathbf{p})\mathbf{R}_z(\phi)\mathbf{T}(-\mathbf{p}), \tag{3.8}$$

amit a 3.2. ábra szemléltet.




*3.2. ábra. Egy adott ponton átmenő, $z$ tengellyel párhuzamos, tengely körüli forgatás szemléltetése*

Egy objektum forgatásának definiálásához az OpenGL-ben a `glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)` függvényt használhatjuk, amely egy $x$, $y$, és $z$ vektor

[2] A tr az angol trace szóból ered.

[3] $\mathbf{R}$-rel a tetszőleges tengely körüli forgatást jelöljük.

körül adott szöggel[4] való elforgató mátrixszal szorozza meg a globális modell-nézeti mátrixot balról. A legegyszerűbb esetekben a forgatást csak a koordinátarendszer fő tengelyei körül adjuk meg.

```
// (1,1,1)-tengely körüli elforgatás
glRotatef(45.0f, 1.0f, 1.0f, 1.0f);

// Drótvázas kocka
glutWireCube(10.0f)
```

*3.2. kódrészlet. Elforgatás 45°-kal*

**Skálázás**

A skálázó mátrix $\mathbf{S}(\mathbf{s}) = \mathbf{S}(s_x, s_y, s_z)$ (ahol $s_x \neq 0, s_y \neq 0$ és $s_z \neq 0$) az $x$, $y$ és $z$ irányokban az $s_x$, $s_y$ és $s_z$ értékekkel skáláz (kicsinyít/nagyít) egy adott objektumot.

$$\mathbf{S}(\mathbf{s}) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.9}$$

Egy skálázást *uniformnak* nevezünk, ha $s_x = s_y = s_z$, ellenkező esetben *nem-uniform* skálázásról beszélünk. Bizonyos esetekben az *izotropikus* és *anizotropikus* kifejezéseket használják a uniform és a nem-uniform helyett. A skálázás inverze megadható a következő alakban: $\mathbf{S}^{-1}(\mathbf{s}) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$. Egy homogén koordináta-vektor $w$ komponensének a manipulációjával uniform skálázási mátrixot hozhatunk létre a következő módon:

$$\mathbf{S}(\mathbf{s}) = \begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/5 \end{pmatrix} \tag{3.10}$$

Amennyiben az s három komponenséből az egyik negatív értéket vesz fel, akkor egy *tükröző mátrixot* kapunk, amit *tükör mátrixnak* is nevezünk. A tükörkép mátrix használatával a háromszögek vertexeinek a körüljárási sorrendje megfordul, ami hatással van a megvilágításra és a hátsólap-eldobására. Ennek meghatározására elegendő kiszámítani a bal-felső $3 \times 3$ mátrix determinánsát. Ha ez az érték negatív, akkor a mátrix tükröző mátrix.

A következő példában egy adott irányba történő skálázást mutatunk be. Mivel a skálázás csak a 3 fő irányba hajtható végre, ezért egy összetett transzformációra lesz szükségünk. Tegyük fel, hogy a skálázást $\mathbf{f}^x$, $\mathbf{f}^y$ és $\mathbf{f}^z$ ortonormált vektorok mentén kell végrehajtani. Először hozzuk létre a következő mátrixot:

$$\mathbf{F} = \begin{pmatrix} \mathbf{f}^x & \mathbf{f}^y & \mathbf{f}^z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.11}$$

[4]Megjegyezzük, hogy a `glRotatef` függvénynél fokban kell megadni a szöget, míg például a C nyelvben a `sin()` vagy `cos()` függvény esetében radiánban kell megadni az adott szögeket.

Az alapötlet az, hogy a 3 tengellyel adott koordinátarendszert úgy transzformáljuk, hogy az eredmény egybeessen a szabványos tengelyekkel. Ezután skálázunk, majd visszatranszformáljuk a pozíciókat. Az első lépésben az $\mathbf{F}$ inverzével szorzunk, vagyis az $\mathbf{F}$ mátrix transzponáltjával. Ezután skálázunk, majd a visszatranszformálunk:

$$\mathbf{X} = \mathbf{F}\mathbf{S}(\mathbf{s})\mathbf{F}^T. \tag{3.12}$$

Az OpenGL függvénykönyvtárban a skálázó mátrixot `glScalef(GLfloat x, GLfloat y, GLfloat z)` függvény meghívásával „állíthatjuk” elő.

```
1 // nem uniform skálázás
2 glScalef(2.0f, 1.0f, 2.0f);
3
4 // Drótvázas kocka
5 glutWireCube(10.0f)
```

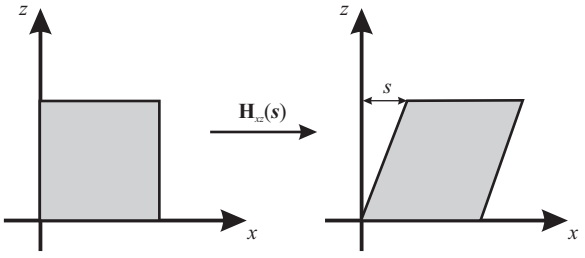
*3.3. kódrészlet. Skálázás*

**Nyírás**

A transzformációk egy másik osztálya a nyíró mátrixok halmaza. A hat alap nyírást a $\mathbf{H}_{xy}(s)$, $\mathbf{H}_{xz}(s)$, $\mathbf{H}_{yx}(s)$, $\mathbf{H}_{yz}(s)$, $\mathbf{H}_{zx}(s)$, $\mathbf{H}_{zy}(s)$ mátrixokkal jelöljük. Az első index azt a koordinátát jelöli, amelyet a nyíró mátrix megváltoztat. A második index pedig azt koordinátát jelöli, amely értékétől a változás mértéke függ. Így a $\mathbf{H}_{xz}(s)$ mátrix a következőképpen néz ki:

$$\mathbf{H}_{xz}(s) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.13}$$

Egy $\mathbf{P} = (p_x, p_y, p_z)^T$ pontot balról megszorozva ezzel a mátrixszal a $\mathbf{P}'$ pontba transzformálja, melynek a koordinátái $(p_x + sp_z, p_y, p_z)^T$. A 3.3. ábra szemlélteti az előbbi nyírásmátrix hatását.




*3.3. ábra. Egy egységnégyzet nyírása a $\mathbf{H}_{xz}(s)$ mátrixszal. Az $y$ és $z$ értékek nem változnak a transzformáció során.*

A $\mathbf{H}_{ij}(s)$ (ahol $i \neq j$) mátrix inverzét az ellentétes irányba való nyírással lehet előállítani $\mathbf{H}_{ij}^{-1}(s) = \mathbf{H}_{ij}(-s)$.

Bizonyos esetekben a nyírást egy kicsit más formában adják meg:

$$\mathbf{H}'_{xy}(s,t) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & t & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.14}$$

Itt a két indexszel azt jelöljük, hogy a két koordinátát nyírjuk a harmadik koordinátával. Az összefüggés a két jelölés között a következő: $\mathbf{H}'_{ij}(s,t) = \mathbf{H}_{ik}(s)\mathbf{H}_{jk}(t)$, ahol $k$ a harmadik koordinátát jelöli.

Mivel bármelyik nyíró mátrix esetén $|\mathbf{H}| = 1$ áll fenn, ezért a nyírás térfogatmegőrző transzformáció.

Az OpenGL függvénykönyvtár nem ad közvetlen megoldást a nyírás végrehajtására. A $4 \times 4$-es mátrixok nem kétdimenziós tömbbel vannak megvalósítva az OpenGL-ben. Amikor `GLfloat matrix[16]` mátrix elemeit oszloponként lefele haladva egyenként kell megadni. Ezt *oszlopfolytonos mátrixrendezésnek* nevezzük (lásd 3.15 mátrixot.).

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix} \tag{3.15}$$

Egyszerű modellezési transzformáció esetén, amikor egy modellt szeretnénk elhelyezni adott pozícióban és irányítottsággal[5], ez a 16 érték meghatározza azt, hogy a modellezési origó a világtér mely pontjára kerüljön a szem koordinátarendszerének megfelelően. Ezeknek a számoknak a megadása nem nehéz. A négy oszlop mindegyike egy négy-elemű vektor. Az egyszerűség kedvéért csak az első három elemét tekintjük ezeknek a vektoroknak. A negyedik oszlop vektor a transzformált koordinátarendszer origójának $x$, $y$ és $z$ értékeit tartalmazza (lásd 3.16. mátrixot). Az egységmátrix betöltése után a `glTranslate` függvény meghívásával az $x$, $y$ és $z$ értékeket a mátrix 12-ik, 13-ik és 14-ik pozíciójában helyezi el.

Az első három oszlop első három eleme (lásd 3.16. mátrixot) csak irány vektorok, amelyek az $x$-, $y$- és $z$-tengelyek irányítottságát adják meg a térben. A legtöbb esetben ez a három vektor merőleges egymásra és egység hosszúak[6] (hacsak skálázást vagy nyírást nem alkalmazunk).

$$\begin{bmatrix} X_x & Y_x & Z_x & T_x \\ X_y & Y_y & Z_y & T_y \\ X_z & Y_z & Z_z & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.16}$$

Ha van egy ilyen $4 \times 4$-es mátrix, amely egy eredeti koordináta-rendszer bázisvektorait és origóját tartalmazza az új koordináta-rendszerben kifejezve, akkor akkor az eredmény egy új vertex lesz, ami az új koordináta-rendszerben helyezkedik el.

[5] Ez egy merevtest-transzformációnak felel meg.

[6] Ortonormáltnak nevezzük a vektorokat, ha merőlegesek és egység hosszúak. Ha csak merőlegesek, akkor az ortogonális kifejezést használjuk.

Ha a saját 4 × 4-es transzformációs mátrixot összeállítottuk, akkor be tudjuk tölteni az adott globális modell-nézeti mátrixba a `glLoadMatrixf(GLfloat *m)`[7] OpenGL függvény meghívásával. A 3.4. példában egy egységmátrixot töltünk be a modell-nézeti mátrixba. Természetesen az `m` mátrix nem csak az előzőleg ismertetett struktúrájú lehet, hanem tetszőleges transzformációt is tartalmazhat, akár nyírást is [8].

```
// Egységmátrix betöltése
GLfloat m[] = { 1.0f, 0.0f, 0.0f, 0.0f,   // X oszlop
                0.0f, 1.0f, 0.0f, 0.0f,   // Y oszlop
                0.0f, 0.0f, 1.0f, 0.0f,   // Z oszlop
                0.0f, 0.0f, 0.0f, 1.0f }; // Eltolás

glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(m);
```

*3.4. kódrészlet. Egységmátrix betöltése*

Habár az OpenGL megvalósítások az oszlopfolytonos mátrix rendezést használják. A következő függvény a mátrix verembe való töltésekor hajtja végre a mátrixok transzponálását: `void glLoadTransposeMatrixf(Glfloat *m)`[9].

**Transzformációk összefűzése**

A mátrixok szorzásának nem-kommutatív tulajdonsága miatt a transzformációs mátrixok összefűzésének a sorrendje hatással van az eredményre. Vegyük a következő transzformációs mátrixokat: $\mathbf{S}(1, 0.5, 1)$ és az $\mathbf{R}_z(\pi/6)$ forgatási mátrixot. A két transzformációs mátrix két lehetséges szorzatát a 3.4. ábra szemlélteti.

Az egyik nyilvánvaló indoka a mátrixok összeszorzásának az, hogy például több ezer vertex esetén nem kell külön-külön elvégezni a szorzásokat a mátrixokkal, hanem elég csak az összetett mátrixszal beszorozni a vertexeket. Például egy skálázás, forgatás és eltolás mátrixnál az összetett mátrix $\mathbf{C} = \mathbf{TRS}$ alakú. Megjegyezzük, hogy először a skálázás hajtódik végre, majd a forgatás, végül az eltolás. Ebből a rendezésből adódik ($\mathbf{TRSp} = \mathbf{T}(\mathbf{R}(\mathbf{Sp}))$).

**Merevtest-transzformáció**

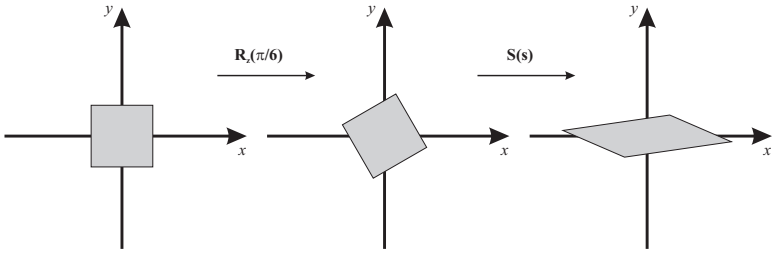
Amikor valaki egy térbeli objektumot megfog, mondjuk egy tollat felvesz az asztalról és egy másik pozícióba mozgatja azt, például az egyik zsebébe beteszi, akkor csak az objektum irányítottsága és a helyzete változik, az alakja nem. Ezeket a transzformációkat, amelyek csak eltolás és forgatás transzformációk összefűzésével állnak elő *merevtest-transzformációknak* nevezzük. A merevtest-transzformációk tulajdonsága, hogy a hosszakat és a szögeket megőrzik.

[7] A függvény másik változata a `glLoadMatrixd(GLdouble *m)`, amely `double` értékeket tartalmazó mátrixot tud betölteni. A legtöbb OpenGL megvalósítás `float`-ként tárolja és kezeli a csővezetékben lévő adatokat. Így a `double` értékek használata teljesítmény csökkenéssel járhat a dupla pontosságú és egyszeres pontosságú számok konvertálása miatt.

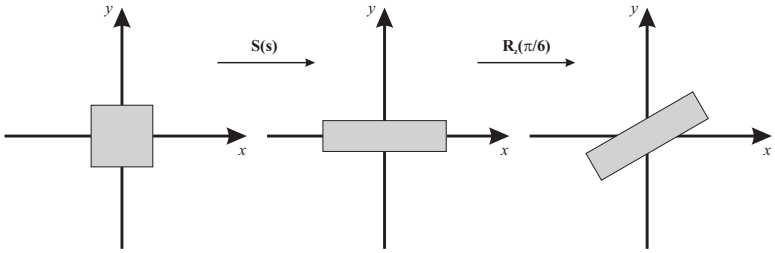
[8] Megjegyezzük, hogy ezzel a technikával akár egy projekciós mátrixot is létrehozhatunk és betölthetjük a globális projekciós mátrixba, melyről a 3.1.8. alfejezetben lesz szó bővebben.

[9] A dupla pontosságú számok transzponálására az OpenGL biztosítja a megfelelő függvényt: `void glLoadTransposeMatrixd(Gldouble *m)`.




*(a) A forgatás, majd a skálázás transzformáció összetett mátrix alakja* $\mathbf{S}(1, 0.5, 1)\mathbf{R}_z(\pi/6)$ *szorzattal áll elő.*




*(b) A skálázás, majd a forgatás transzformáció összetett mátrix alakja* $\mathbf{R}_z(\pi/6)\mathbf{S}(1, 0.5, 1)$ *szorzattal áll elő.*

*3.4. ábra. Transzformációk összefűzésének sorrendfüggősége. Tetszőleges* $\mathbf{N}$ *és* $\mathbf{M}$ *mátrixok esetén igaz a* $\mathbf{NM} \neq \mathbf{MN}$ *állítás.*

Bármely $\mathbf{X}$ merevtest-transzformáció felírható $\mathbf{T}(\mathbf{t})$ eltolás- és $\mathbf{R}$ elforgatásmátrix szorzataként. Ennek következtében $\mathbf{X}$ mátrix alakja a következőképpen néz ki:

$$\mathbf{X} = \mathbf{T}(\mathbf{t})\mathbf{R} = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.17}$$

Az $\mathbf{X}$ inverze kiszámítható az alábbi összefüggések alapján:

$$\mathbf{X}^{-1} = (\mathbf{T}(\mathbf{t})\mathbf{R})^{-1} = \mathbf{R}^{-1}\mathbf{T}(\mathbf{t})^{-1} = \mathbf{R}^T\mathbf{T}(-\mathbf{t}). \tag{3.18}$$

Ez alapján elegendő a jobb felső $3 \times 3$-as $\mathbf{R}$ mátrix transzponáltját képezni, a $\mathbf{T}$ eltolási mátrix értékeinek az előjelét megfordítani és az így kapott mátrixokat fordított sorrendben összeszorozni.

Az $\mathbf{X}$ mátrix inverze más módon is kiszámítható. Írjuk fel az $\mathbf{R}$ és $\mathbf{X}$ mátrixokat a következő alakban:

$$\overline{\mathbf{R}} = (\mathbf{r}_{,0} \;\; \mathbf{r}_{,2} \;\; \mathbf{r}_{,2}) = \begin{pmatrix} \mathbf{r}_{0,}^T \\ \mathbf{r}_{1,}^T \\ \mathbf{r}_{2,}^T \end{pmatrix}$$
$$\Rightarrow \tag{3.19}$$
$$\mathbf{X} = \begin{pmatrix} \overline{\mathbf{R}} & \mathbf{t} \\ 0^T & 1 \end{pmatrix}$$

Itt a 0 egy $3 \times 1$ csupa nullákkal feltöltött oszlopvektor. Néhány egyszerű átalakítás után, az $\mathbf{X}$ inverze felírható a következő módon:

$$\mathbf{X}^{-1} = \begin{pmatrix} \mathbf{r}_{0,} & \mathbf{r}_{1,} & \mathbf{r}_{2,} & -\overline{\mathbf{R}}^T\mathbf{t} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.20}$$

**Normálvektor transzformációja**

Egy pont transzformált képe egy pont lesz a transzformációs mátrixszal való szorzás után. Ennek következtében, az összetettebb geometriákat meghatározó pontok transzformálása után az adott geometriát meghatározó, transzformált pontokat kapunk eredményül. Mivel a lineáris transzformációk esetén pontok különbségének a képe a a képek különbsége $(\mathbf{A}(\mathbf{x}_1 - \mathbf{x}_2) = \mathbf{A}\mathbf{x}_1 - \mathbf{A}\mathbf{x}_2)$, ezért a geometria felületi pontjai között lévő vektorok is transzformálhatóak egy mátrixszal. Ugyanakkor, nem szögtartó transzformációkat tartalmazó mátrixok (skálázás, nyírás) esetén nem mindig használhatóak fel a felületi normálvektorok és a vertexek megvilágítási normálvektorok transzformációjára, mivel ebben esetben a transzformált normálvektorok nem feltétlenül lesznek merőlegesek a transzformált felületre. A normálvektorokat a geometriai transzformációs mátrix inverzének a transzponáltjával kell megszorozni, hogy a megfelelő eredményt kapjuk [3]. Tehát, ha a geometriai transzformáció mátrixát $\mathbf{M}$-mel jelöljük, akkor az $\mathbf{N} = (\mathbf{M}^{-1})^T$ mátrixszal kell a geometriához tartozó normálvektorokat transzformálni.

A gyakorlatban, ha tudjuk, hogy a mátrix ortogonális, például csak forgatásokat tartalmaz, akkor nem kell kiszámolni az inverzét, mivel a mátrix inverze maga a transzponált mátrix. Így a két egymás utáni transzponálás az eredeti mátrixot adja vissza. Továbbá az eltolás nincs hatással a vektor irányára, ezért tetszőleges számú eltolás alkalmazható anélkül, hogy a normálvektor megváltozna. Az eltolás és forgatás után a normálvektor egységre való normalizálást sem kell végrehajtani, hiszen ezek a transzformációk a hosszokat megőrzik. Így az eredeti mátrixot használhatjuk a normálvektorok transzformálására.

Ráadásul ha egy vagy több uniform skálázó mátrixot is felhasználtunk a transzformációs mátrix előállítása során, akkor szintén nincs szükség az inverz kiszámítására. Az ilyen fajta skálázások csak a vektor hosszát módosítják, nem pedig az irányát. Ebben az esetben a normálvektorokat egységnyi hosszúra kell normalizálni. Amennyiben ismerjük a skálázási faktort, akkor ezt felhasználhatjuk a normálvektorok normalizálására. Például ha ez a faktor 2.3, akkor a normálvektorokat 2.3-mal kell elosztani.

Abban az esetben, amikor kiderül, hogy a teljes inverzet ki kell számítani, elegendő a

mátrix bal felső $3 \times 3$-as mátrixának *adjungáltjának*[10] a transzponáltját meghatározni. Nincs szükség az osztásra, úgy ahogy az inverz kiszámításánál láttuk, hiszen tudjuk, hogy úgyis egységre kell normalizálni a normálvektorokat.

Megjegyezzük, hogy a normál transzformációk nem jelentenek problémát azoknál a rendszereknél, ahol a transzformáció után a felületi normálvektorokat a háromszögből határozzák meg (például a háromszög éleinek a keresztszorzatából).

**Inverzek kiszámítása**

A mátrixok inverzének a meghatározására sok esetben szükségünk van, például normálvektorok transzformációja esetén. A meglévő transzformációs információk alapján a következő három módszert lehet használni egy mátrix inverzének a kiszámításakor:

- Ha a mátrix egy transzformációt vagy egyszerű transzformációk sorozatát tartalmazza adott paraméterekkel, akkor a mátrixot egyszerűen lehet invertálni a „paraméterek invertálásával" és a mátrixok sorrendjének a megfordításával. Például, ha $\mathbf{M} = \mathbf{T}(\mathbf{t})\mathbf{R}(\phi)$, akkor $\mathbf{M}^{-1} = \mathbf{R}(-\phi)\mathbf{T}(-\mathbf{t})$.
- Ha tudjuk, hogy a mátrix ortogonális, akkor $\mathbf{M} = \mathbf{M}^T$, vagyis az adott mátrix transzponáltja az inverze. Bármely forgatások sorozata ortogonális.
- Amennyiben semmilyen pontos információnk nincs a transzformációkról, akkor az adjungált módszer, Cramer szabály, LU felbontás vagy Gauss elimináció használható a mátrix inverzének a kiszámítására. A Cramer szabály és az adjungált módszer használata általában ajánlottabb, mivel kevesebb elágazó műveletet tartalmaznak[11].

Az inverzszámítás célját az optimalizáláskor is figyelembe vehetjük. Például, ha irányvektort transzformálunk inverz mátrixszal, akkor a bal-felső $3 \times 3$-as almátrixot kell csak invertálni általában.

### 3.1.7. Vágótér

A kameratérben lévő primitíveket a következő lépésben a képsíkra kell leképezni. A vágótérben a koordináták azt adják meg, hogy egy pont a képernyőn hova kerül és mi annak a pontnak a mélység értéke. A homogén koordináták használata miatt ezek az értékek $w$ súllyal vannak módosítva.

A tengelyhez-igazított kocka bal-alsó koordinátája $(-1,-1,-1)$ és a bal-felső sarka (1,1,1). Ezt a kockát *kanonikus nézeti térfogatnak* és a koordinátákat normalizált eszköz koordinátáknak nevezzük ebben a térfogatban. A vágást sokkal hatékonyabban lehet végrehajtani a nézeti és vetületi transzformáció után, mivel a nem a képsíkra vetülő illetve a mélységi vágósíkokon kívül lévő pontok könnyen meghatározhatók. Ezek tulajdonképpen a vágókockán kívül esnek, amely határoló lapjait nevezzük vágósíkoknak.

[10] Egy $\mathbf{A}$ általános méretű mátrix $\mathbf{M}$ adjungált mátrixának az elemeit a következőképpen kell kiszámítani $\left[a_{ij}\right] = \left[(-1)^{(i+j)} d_{ij}^{\mathbf{M}}\right]$, ahol $d_{ij}^{\mathbf{M}}$ az $\mathbf{A}$ mátrix aldeterminánsa, amit az $\mathbf{A}$ mátrixból az $i$-ik sor és $j$-ik oszlop elhagyásával képzünk.

[11] A modern architektúrák esetében érdemes az „if" műveletet elkerülni.

A kanonikus nézeti térfogatba való transzformálás után, a geometria megjelenítendő vertexeit levágjuk a kockának megfelelően. A kockán belüli részt jelenítjük meg a megmaradó egységkocka képernyőre való leképezésével.

### 3.1.8. A vetületi transzformáció

A kameratér koordinátáit a vágótér koordinátáiba a *vetületi transzformáció* segítségével transzformáljuk át. Ezt a leképezést egy $4 \times 4$-es mátrixszal fejezhetjük ki. Ezt a $4 \times 4$-es mátrixot *vetületi mátrixnak* nevezzük. A nézeti térfogat pontos alakja a vetületi transzformáció típusától függ: egy perspektív transzformáció egy csonka gúlát határoz meg, míg egy ortogonális vetítés egy téglatestet visz át vágókockába. Csak azok a poligonok, vonalak és pontok lesznek potenciálisan láthatóak a raszterizálás során, amelyek az adott térfogaton belül helyezkednek el.

**Ortogonális vetítés**

Az ortogonális vetítésnél a párhuzamos vonalak párhuzamosak maradnak a transzformáció után. A $\mathbf{P}_O$ az $x$ és $y$ koordinátákat nem változtatja, míg a $z$ komponenst 0-ra állítja, vagyis a $z = 0$ síkra vetíti ortografikusan.

$$\mathbf{P}_O = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.21}$$

A $\mathbf{P}_O$ nem invertálható, mivel a determinánsa $|\mathbf{P}_O| = 0$. Más szavakkal, a transzformáció az egyik dimenziót elhagyja és nincs mód arra, hogy az elveszett dimenziót visszaszerezze. Az ilyen ortogonális vetítés használatával az a probléma a néző számára, hogy mind a pozitív és mind a negatív $z$ értékeket levetíti a vetítési síkra. Így a $z$ irányban nem lehet vágást végrehajtani és a takarásban lévő felület-primitívek kezelése is problémássá válik.

Az ortogonális vetítés végrehajtásához a leggyakoribb mátrix megadását egy hatossal $(l, r, b, t, n, f)$ tehetjük meg, ahol a bal, jobb, alsó, felső, közeli és távoli síkokat jelöljük meg rendre. Ez a mátrix lényegében skálázza és eltolja az ezekkel a síkokkal kialakított Tengelyhez Igazított Befoglaló Dobozt (AABB[12]) egy origó középpontú, tengelyhez-igazított vágókockába. A bal-alsó sarka az AABB-nek $(l, b, n)$ és a jobb-felső sarka pedig a $(r, t, f)$.

Az OpenGL-ben a $z$-tengely negatív része felé nézünk, mivel a $-z$ féltérben lévő modelleket szeretnénk megjeleníteni. Ezért $n > f$, mivel a $z$-tengely negatív irányába nézünk. Könnyebb dolgunk van akkor, ha a közeli értékek kisebbek, mint a távoliak, ezért az ortogonális mátrix előállításért felelős `glOrtho` függvény esetén a bemenő közeli értékét kisebb értékként kell megadni, mint a távolit. Azaz negált értékeket kell megadnunk.

Az ortogonális transzformációs mátrixot a következő képen adjuk meg:

[12] Az angol *Axis-Aligned Bounding Box* rövidítése.

$$\mathbf{P}_o = \mathbf{S}(\mathbf{s})\mathbf{T}(\mathbf{t}) = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.22}$$

Ez a mátrix invertálható: $\mathbf{P}_o^{-1} = \mathbf{T}(-\mathbf{t})\mathbf{S}((r-l)/2, (t-b)/2, (f-n)/2))$.

Az OpenGL parancs, amely az ortogonális vetítési transzformációs mátrixszal szorozza meg a globális projekciós mátrixot a `glOrtho`:

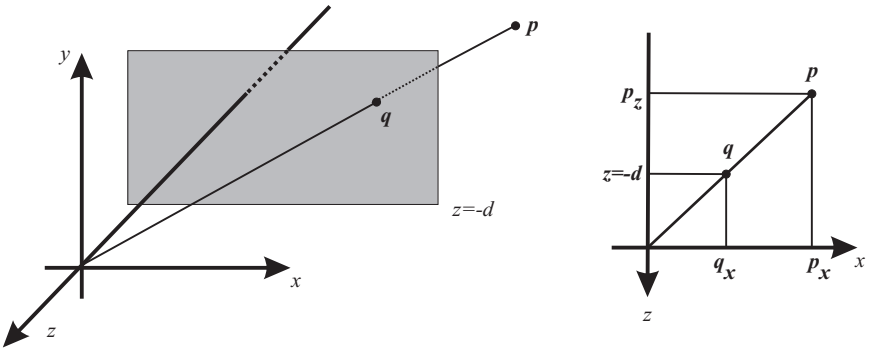
```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,
GLdouble top, GLdouble near, GLdouble far )
```

ahol a paraméterek a már jól ismert határoló síkok. A függvény hívása előtt a vetítési mátrixot ki kell jelölni a `glMatrixMode(GL_PROJECTION)` függvényhívással és a `glLoadIdentity()` függvénnyel inicializálni kell azt.

**Perspektív vetítés**

A párhuzamos vetítésnél sokkal érdekesebb a perspektív (középpontos) vetítési transzformáció, amelyet számítógépes grafikai alkalmazások többségében használnak. Ebben az esetben a párhuzamos vonalak általában a vetítés után nem lesznek párhuzamosak, inkább egy pontban találkoznak. Szélsőséges esetben, a szempozíción áthaladó metsző vonalakból párhuzamos vonalakat kapunk a perspektív vetítés elvégzése után. A perspektív vetítés sokkal közelebb áll ahhoz, ahogyan a világot érzékeljük[13].

Először vezessük le azt a perspektív vetítési mátrixot, amely a $z = -d, d > 0$ síkra képez le. Tegyük fel, hogy a nézőpont az origóban van és egy $\mathbf{p}$ pontot vetítünk, melynek a képe $\mathbf{q} = (q_x, q_y, -d)$ (lásd 3.5. ábrát).




*3.5. ábra. A perspektív vetítési transzformációs mátrix levezetéséhez használt jelölések*

[13] Például középpontos vetítéssel képződik a kép fényképezéskor a filmen.

A hasonló háromszögek alapján a $\mathbf{q}$ pont $x$ összetevőjére a következő összefüggés írható fel:

$$\frac{q_x}{p_x} = \frac{-d}{p_z} \Longrightarrow q_x = -d\frac{p_x}{p_z}. \tag{3.23}$$

A $\mathbf{q}$ többi komponensére hasonlóan kaphatók meg a $q_y = -d\frac{p_y}{p_z}$ és $q_z = -d$ összefüggések. Ezekből kapjuk a $\mathbf{P}_p$ perspektív vetítési mátrixot:

$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}. \tag{3.24}$$

A mátrixot ellenőrizve kapjuk, hogy

$$\begin{aligned} \mathbf{q} = \mathbf{P}_p\mathbf{p} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \\ &= \begin{pmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{pmatrix} \Rightarrow \begin{pmatrix} -dp_x/p_z \\ -dp_y/p_z \\ -d \\ 1 \end{pmatrix}. \end{aligned} \tag{3.25}$$

Az utolsó lépésben a $w$ komponenssel elosztjuk a vektort azért, hogy az utolsó pozícióban 1-et kapjunk. A geometriai értelmezése az iménti *homogenizálási eljárásnak* a $(p_x, p_y, p_z)$ pont egy 4D-s tér $w = 1$ hipersíkra való leképezése. Hasonlóan a ortogonális transzformációhoz, van egy perspektív transzformáció, amely ahelyett, hogy egy síkra vetítene (ami nem invertálható), a nézeti csonka gúlát a kanonikus nézeti térfogatba transzformálja, ahogy azt korábban láttuk. A csonka gúla nézet esetén feltesszük, hogy a $z = n$-ben kezdődik és a $z = f$-ben végződik $0 > n > f$ értékek esetén. A téglalap $z = n$ esetén a bal-alsó sarok $(l, b, n)$, $(r, t, n)$ esetén pedig a jobb felső sarkot kapjuk.

A $(l, r, b, t, n, f)$ (jobb, bal, alsó, felső, közeli, távoli) paraméterek meghatározzák a kamera nézeti csonka gúláját. A vízszintes látómezőt a csonka gúla bal és jobb síkjai között lévő szög határozza meg. Hasonló módon a függőleges látómezőt az alsó és felső síkok közötti szög adja meg. Minél nagyobb a látószög, annál többet „lát” a kamera. Aszimmetrikus csonka gúlát $r \neq -l$ vagy $t \neq -b$ értékekkel hozhatunk létre, melyeket sztereó látásnál használnak.

A látómező egy fontos tényezője a színtér észlelésének. Összehasonlítva a számítógép képernyőjével, a szemnek van egy fizikai látó mezeje. A kettő közötti összefüggést a következőképpen írhatjuk fel:

$$\phi = 2\arctan(w/(2d)), \tag{3.26}$$

ahol a látómező $\phi$, $w$ a színtér szélessége, amely merőleges a nézeti irányra és $d$ az objektumtól való távolság. Szélesebb látómezőt beállítva az objektumok torzítva fognak megjelenni a képen.

A perspektív transzformációs mátrix, amely a csonka gúlát az egység kockába transzformálja, a következőképpen adható meg:

$$\mathbf{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{3.27}$$

Ezt a transzformációt végrehajtva egy $\mathbf{q} = (q_x, q_y, q_z, q_w)^T$ pontra, a $w$ komponens értéke (a legtöbb esetben) nem nulla lesz és nem lesz egyenlő eggyel. A vetített $\mathbf{p}$ ponthoz osztanunk kell $q_w$-vel $\mathbf{p} = (q_x/q_w, q_y/q_w, q_z/q_w, 1)^T$. A $\mathbf{P}_p$ mátrix a $z = f$-et $+1$-re és a $z = n$-et $-1$-re képezi le. A perspektív transzformáció végrehajtása után vágással és homogenizálással kapjuk meg a normalizált eszköz koordinátákat.

OpenGL-ben a perspektív transzformációnál, először a $\mathbf{S}(1, 1, -1)$ mátrixszal kell megszorozni. Ez egyszerűen negálja a 3.27. egyenlet harmadik oszlopának értékeit. Ezután a tükrözés után a közeli és távoli értékek pozitív értékek lesznek ($0 < n' < f'$), bár ezek az értékek még mindig távolságot ábrázolnak a nézeti irány mentén.

Az OpenGL parancs, amely az perspektív vetítési transzformációs mátrixszal szorozza meg a globális projekciós mátrixot a `glFrustum`:

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

ahol a paraméterek a már jól ismert határoló síkok. A függvény hívása előtt a vetítési mátrixot ki kell jelölni a `glMatrixMode(GL_PROJECTION);` függvényhívással és a `glLoadIdentity();` függvénnyel inicializálni kell azt.

A `glFrustum` mellett érdemes megemlítenünk, hogy ennél a függvénynél van egy intuitívabb segédfüggvény, amely közelebb áll a mindennapi használathoz:

`void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar),`

ahol a `fovy` a függőleges nézeti mező szöge; az `aspect` a képméretarány. A másik két paraméter a közeli és távoli síkok pozícióját határozzák meg.

### 3.1.9. A normalizált eszköz koordináták

A vágási koordináták homogén formában vannak tárolva $(x, y, z, w)$, de nekünk $x$ és $y$ párokra van szükségünk a mélység értékekkel együtt.

Az $x$, $y$ és $z$ értékeket $w$-vel elosztva tudjuk elvégezni a *perspektív osztást*. Az eredmény koordinátákat *normalizált eszköz koordinátáknak* nevezzük. Mostantól mindegyik geometriai adat koordinátái egy kockán belül találhatóak, ahol a pozíciók OpenGL-ben a $(-1, -1, -1)$ és $(1, 1, 1, )$ értékek között lehetnek.

### 3.1.10. Ablak koordináták

Az utolsó lépésben mindegyik vertex normalizált eszköz koordinátáját átkonvertáljuk a végső koordinátarendszerbe, ahol $x$ és $y$ pixel pozíciókat használunk. Ezt a lépést *nézet-ablak transzformációnak* nevezzük.

A nézeti ablak meghatározza azt a teret az ablakon belül aktuális képernyő koordinátákban, amit az OpenGL használhat a rajzolásra, amelynek a megadására a `glViewport` OpenGL függvényt használjuk. A `glViewport` függvény a következőképpen van definiálva:

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height)
```

ahol az `x` és `y` paraméterek a nézeti ablak bal-alsó sarkát adják meg az ablakon belül és a `width` és `height` paraméterekkel pixelben adjuk meg a méretet.

A raszterizáló a vertexekből pontokat, vonalakat és poligonokat állít elő és fragmenseket generál, amelyek majd meghatározzák a végső eredmény képet. A *mélység-távolság transzformáció* egy másik transzformáció, amely a vertexek $z$ értékeit skálázza be a mélységpuffer értékeinek az intervallumába.

## 3.2. Speciális transzformációk

Ebben a fejezetben számos mátrix-transzformációt és műveletet mutatunk be, melyek a valósidejű grafikában nélkülözhetetlenek.

### 3.2.1. Euler transzformáció

Ez a transzformáció intuitív módja egy olyan mátrix létrehozásának, amely egy tetszőleges forgatás megadására alkalmas, így egy kamera irányítottságának a beállítására is. A transzformáció a nevét a svájci Leonard Euler (1707-1783) matematikusról kapta.

Alapesetben legtöbbször a kamera az origóban helyezkedik el, a $z$-tengely negatív irányába néz és a felfele mutató irány az $y$-tengely pozitív irányába mutat (lásd 3.6. ábrát). Az Euler transzformáció három mátrix szorzata (lásd 3.6. ábrát), melynek a szokásos jelölése a következő:

$$\mathbf{E}(h,p,r) = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h), \tag{3.28}$$

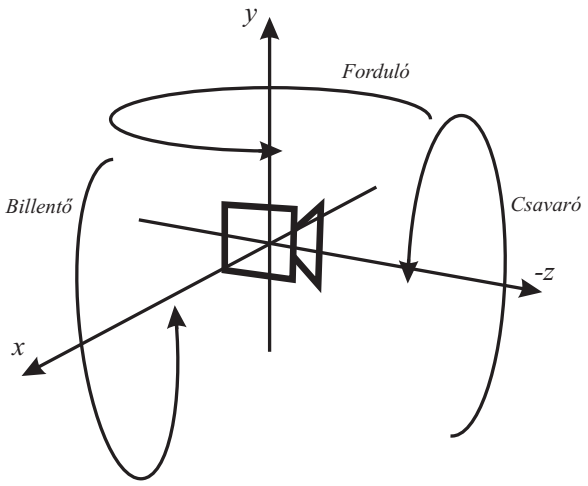
ahol a $h$ a forduló (head/yaw), $p$ a billentő (pitch) és az $r$ a csavaró (roll) szögeket jelölik.

Mivel az $\mathbf{E}$ forgatások összefűzésével áll elő, ezért az Euler transzformáció ortogonális. Emiatt az inverze könnyen számítható, mint $\mathbf{E}^{-1} = \mathbf{E}^T = (\mathbf{R}_z\mathbf{R}_x\mathbf{R}_y)^T = \mathbf{R}_y^T\mathbf{R}_x^T\mathbf{R}_z^T$ mátrixok szorzata. Természetesen az $\mathbf{E}$ mátrix transzponáltját egyszerűbben meg lehet határozni.

Az Euler szögek $h$, $p$ és $r$ adják meg, hogy milyen sorrendben és mennyivel kell elforgatni a nézőpontot a nekik megfelelő tengelyek mentén. Megjegyezzük, hogy ez a transzformáció nem csak a kamerát, de bármilyen objektumot vagy entitást el tud transzformálni.

Az Euler transzformációk használatakor a *gimbal lock*-nak nevezett jelenség fordulhat elő, amikor is a forgatások következtében egy szabadsági fokot elveszítünk. Például először is, ne forgassunk az $y$ tengely körül, vagyis $h = 0$. Ezután az $x$ tengely mentén mondjuk $\pi/2$-vel forgassunk $p = 90°$. Végül a $z$-tengely mentén szeretnénk forgatni, de az előző




*3.6. ábra. Euler transzformáció*

forgatás miatt a $z$-tengely körüli forgatás lényegében az $y$-tengely körüli forgatásnak fog megfelelni. A végeredmény az, hogy elveszítettünk egy szabadsági fokot - nem tudunk a $z$ világtengely körül forgatni. A következő 3.3. fejezetben tárgyalt kvaterniók esetén nem áll fenn ez a jelenség.

Egy másik mód annak az igazolására, hogy egy szabadsági fokot elveszítettünk, ha tekintjük az $\mathbf{E}(h,p,r)$ mátrixot $p=\pi/2$ esetén:

$$\mathbf{E}(h,\pi/2,r)=\begin{pmatrix}\cos r\cos h-\sin r\sin h & 0 & \cos r\sin h+\sin r cosh\\ \sin r\cos h+\cos r\sin h & 0 & \sin r\sin h-\cos r\cos h\\ 0 & 1 & 0\end{pmatrix}= \begin{pmatrix}\cos(r+h) & 0 & \sin(r+h)\\ \sin(r+h) & 0 & \cos(r+h)\\ 0 & 1 & 0\end{pmatrix}. \tag{3.29}$$

Mivel a mátrix csak egy $(r+h)$ szögtől függ, ezért ebből az következik, hogy egy szabadsági fokot elveszítettünk.

### 3.2.2. Paraméterek kinyerése az Euler transzformációból

Néhány esetben hasznos a $h$, $p$ és $r$ Euler paraméterek kinyerése az ortogonális mátrixból. Ezt az eljárást a 3.30. egyenlet adja.

$$\mathbf{F}=\begin{pmatrix}f_{00} & f_{01} & f_{02}\\ f_{10} & f_{11} & f_{12}\\ f_{20} & f_{21} & f_{22}\end{pmatrix}=\mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h)=\mathbf{E}(r,p,h). \tag{3.30}$$

A forgatási mátrixszorzását (3.30 egyenlet) elvégezve kapjuk, hogy :

$$\mathbf{F} = \begin{pmatrix} \cos r \cos h - \sin r \sin p \sin h & -\sin r \cos p & \cos r \sin h + \sin r \sin p \cos h \\ \sin r \cos h + \cos r \sin p \sin h & \cos r \cos p & \sin r \sin h - \cos r \sin p \cos h \\ -\cos p \sin h & \sin p & \cos p \cos h \end{pmatrix}. \tag{3.31}$$

Ebből az alakból a billentő $p$ paramétert a $\sin p = f_{21}$-ből kapjuk meg. Továbbá az $f_{01}$-et elosztva az $f_{11}$-gyel valamint az $f_{20}$-at elosztva az $f_{22}$-vel a következő összefüggéseket kapjuk a csavaró $r$ és forduló $h$ paraméterekre:

$$\begin{aligned} \frac{f_{01}}{f_{11}} &= \frac{-\sin r}{\cos r} = -\tan r \\ \frac{f_{20}}{f_{22}} &= \frac{-\sin h}{\cos h} = -\tan h \end{aligned} \tag{3.32}$$

Összefoglalva az Euler paramétereket az $\mathbf{F}$ mátrixból a következő összefüggések alapján határozhatjuk meg:

$$\begin{aligned} h &= \arctan(-\frac{f_{20}}{f_{22}}), \\ p &= \arcsin(f_{21}), \\ r &= \arctan(-\frac{f_{01}}{f_{11}}). \end{aligned} \tag{3.33}$$

Természetesen, le kell kezelnünk azt a speciális esetet, amikor $\cos p = 0$, mivel ekkor $f_{01} = f_{11} = 0$, amikor az $\arctan$ függvény nem értelmezett. Ebből az következik, hogy $\sin p = \pm 1$. Ezzel egyszerűsíthetjük $\mathbf{F}$-et a következőképpen:

$$\mathbf{F} = \begin{pmatrix} \cos(r \pm h)) & 0 & \sin(r \pm h) \\ \sin(r \pm h) & 0 & \cos(r \pm h) \\ 0 & \pm 1 & 0 \end{pmatrix}. \tag{3.34}$$

A többi paramétert megkapjuk $h = 0$-ra való beállításával, ekkor $\sin r / \cos r = \tan r = f_{10}/f_{00}$, amiből $r = \arctan(f_{10}/f_{00})$ következik.

Megjegyezzük, hogy az $\arcsin$ definíciójából következik, hogy $-\pi/2 \leq p \leq \pi/2$, ami azt jelenti, hogy ha az $\mathbf{F}$ olyan $p$ paraméterrel lett előállítva, ami kívül van ezen az intervallumon, akkor az eredeti paraméter nem nyerhető ki szögfüggvények periódusa miatt. Az, hogy a $h$, $p$ és $r$ paraméterek nem egyediek, azt jelenti, hogy ugyanazt a transzformációt elő lehet állítani több Euler paraméter halmazzal.

### 3.2.3. Mátrix felbontás

A feladat eddig az volt, hogy egy transzformációs mátrixot állítsunk elő, de előfordulhat az hogy különböző transzformációk paramétereit kell meghatároznunk egy transzformációk

összefűzésével előállított mátrixból. Ezt a művelet *mátrix felbontásnak* nevezzük. Nagyon sok oka lehet annak, hogy egy transzformációs halmazt szeretnénk kinyerni. A használata magába foglalja a következő eseteket:

- Csak a skálázási paramétereket nyerjük ki;
- Egy transzformáció megkeresése egy bizonyos rendszer részére. Például VRML állományok előállításakor a `Transform` csomópontban nem lehet egy tetszőleges $4 \times 4$ mátrixot használni;
- Meghatározni azt, hogy a modellen csak egy merevtest-transzformációt alkalmaztak-e vagy sem;
- Egy animáció kulcspozíciói közötti interpolálás végrehajtása, ahol az objektum számára csak a mátrix áll rendelkezésre;
- A nyírások eltávolítása a forgatási mátrixból.

Korábban bemutattunk két felbontást, ahol az egyik esetben az eltolás és forgatási mátrixot származtattunk egy merevtest-transzformáció számára és az Euler szögek meghatározása egy ortogonális mátrixból. Ahogy láttuk, triviális egy eltolási mátrix kinyerése a $4 \times 4$-es mátrix utolsó oszlopából. Szintén meghatározhatjuk a mátrix determinánsából azt, hogy tükrözést hajtottak-e végre vagy sem. A forgatás, skálázás és nyírás szétválasztása komolyabb erőfeszítést igényel.

### 3.2.4. Forgatás tetszőleges tengely mentén

Néha kényelmes, ha van egy olyan eljárásunk, amely egy objektumot valamilyen szöggel forgat el egy tetszőleges tengely körül. Tegyük fel hogy az $\mathbf{r}$ forgatási tengely normalizált, és a transzformáció $\alpha$ szöggel forgat $\mathbf{r}$ körül. Ennek a végrehajtásához találnunk kell két másik egység hosszú tetszőleges tengelyt, melyek egymásra és $\mathbf{r}$-rel ortogonálisak (merőlegesek), azaz ortonormáltak. Ezek egy bázist alkotnak ebben az esetben. Az alapötlet az, hogy a bázist változtassuk meg az alapról erre az új bázisra, majd forgassuk el az adott objektumot $\alpha$ szöggel mondjuk az $x$-tengely (megfelel az $\mathbf{r}$-nek ebben az esetben) körül. Ezután transzformáljunk vissza az alap bázisba.

Az első lépés az, hogy számítsuk ki a bázis ortonormált tengelyeit. Az első tengely az $\mathbf{r}$ azaz az, amelyik körül forgatni akarunk. Most a második $\mathbf{s}$-tengely megkeresésére koncentrálunk, tudva azt, hogy a harmadik $\mathbf{t}$-tengelyt az első és második tengely keresztszorzataként ($\mathbf{t} = \mathbf{r} \times \mathbf{s}$) kapjuk meg. Egy numerikusan stabil módja ennek az $\mathbf{r}$ abszolút értékben legkisebb komponensének 0-ra állítása, majd a két másik komponens megcserélése után negáljuk az elsőt ezek közül. Ezt a három vektort helyezzük el egy mátrix soraiban:

$$\mathbf{M} = \begin{pmatrix} \mathbf{r}^T \\ \mathbf{s}^T \\ \mathbf{t}^T \end{pmatrix}. \tag{3.35}$$

Ez a mátrix az $\mathbf{r}$ vektort az $x$-tengelybe, az $\mathbf{s}$ vektort az $y$-tengelybe és a $\mathbf{t}$ vektort a $z$-tengelybe transzformálja. Így a normalizált $\mathbf{r}$-tengely körüli $\alpha$ szöggel való végső transzformáció a következőképpen néz ki:

$$\mathbf{X} = \mathbf{M}^T \mathbf{R}_x(\alpha)\mathbf{M}. \tag{3.36}$$

Egy másik módszer (Goldman) a tetszőleges normalizált $\mathbf{r}$-tengely körüli forgatásra $\phi$ szöggel:

$$\mathbf{R} = \begin{pmatrix} \cos\phi + (1-\cos\phi)r_x^2 & (1-\cos\phi)r_x r_y - r_z \sin\phi & (1-\cos\phi)r_x r_z + r_y \sin\phi \\ (1-\cos\phi)r_x r_y + r_z \sin\phi & \cos\phi + (1-\cos\phi)r_y^2 & (1-\cos\phi)r_y r_z - r_x \sin\phi \\ (1-\cos\phi)r_x r_z - r_y \sin\phi & (1-\cos\phi)r_y r_z + r_x \sin\phi & \cos\phi + (1-\cos\phi)r_z^2 \end{pmatrix}. \tag{3.37}$$

## 3.3. Kvaterniók

A kvaterniók lenyűgöző tulajdonságukkal hatékony eszközt nyújtanak transzformációk készítésére és a kvarteniók az Euler szögek és mátrixok felett állnak, különösen akkor, amikor forgatásra és irányításra használjuk azokat. Nagyon tömören vannak ábrázolva és felhasználhatóak az irányítottságok interpolálására.

A kvanterniónak négy összetevője van, így vektorként ábrázoljuk őket, de megkülönböztetjük azoktól és $\hat{\mathbf{q}}$-val jelöljük őket.

### 3.3.1. Matematikai háttér

A $\hat{\mathbf{q}}$ kvaterniót a következő módokon definiálhatjuk:

$$\begin{aligned} \hat{\mathbf{q}} &= (\mathbf{q}_v, q_w) = iq_x + jq_y + kq_z + q_w = \mathbf{q}_v + q_w \\ \mathbf{q}_v &= iq_x + jq_y + kq_z = (q_x, q_y, q_z) \\ i^2 &= j^2 = k^2 = -1, jk = -kj = i, ki = -ik = j, ij = -ji = k \end{aligned} \tag{3.38}$$

A $q_w$ változó a $\hat{\mathbf{q}}$ kvaternió valós része. A képzetes része $\mathbf{q}_v$ és $i$, $j$ és $k$ a képzetes egységek.

A $q_v$ képzetes részen értelmezve van az összes vektor művelet, mint például a skálázás, skaláris szorzat és kereszt szorzat. A kvaternió definíciója alapján a két $\hat{\mathbf{q}}$ és $\hat{\mathbf{r}}$ szorzatát a következőképpen tudjuk levezetni. Megjegyezzük, hogy a képzetes egységek szorzása nem kommutatív.

$$\begin{aligned}\hat{\mathbf{q}}\hat{\mathbf{r}} &= (iq_x + jq_y + kq_z + q_w)(ir_x + jr_y + kr_z + r_w)\\ &= i(q_y r_z - q_z r_y + r_w q_x + q_w r_x)\\ &+ j(q_z r_x - q_x r_z + r_w q_y + q_w r_y)\\ &+ k(q_x r_y - q_y r_x + r_w q_z + q_w r_z)\\ &+ q_w r_w - q_x r_x - q_y r_y - q_z r_z =\\ &= (\mathbf{q}_v \times \mathbf{r}_v + r_w \mathbf{q}_v + q_w \mathbf{r}_v), q_w r_w - \mathbf{q}_v \cdot \mathbf{r}_v\end{aligned} \tag{3.39}$$

A kvaterniók definíciója mellett szükség van az összeadás, konjugált, norma és az egység megadásra is:

**Összedás:** $\hat{\mathbf{q}} + \hat{\mathbf{r}} = (\mathbf{q}_v + \mathbf{r}_v, q_w + r_w)$

**Konjugált:** $\hat{\mathbf{q}}^* = (\mathbf{q}_v, q_w)^* = (-\mathbf{q}, q_w)$

**Norma:** $n(\hat{\mathbf{q}}) = \sqrt{\hat{\mathbf{q}}\hat{\mathbf{q}}^*} = \sqrt{\hat{\mathbf{q}}^*\hat{\mathbf{q}}} = \sqrt{\mathbf{q}_v \cdot \mathbf{q}_v + q_w^2} = \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}$

**Egység:** $\hat{\mathbf{i}} = (0, 1)$

A multiplikatív inverz esetén igaz a $\hat{\mathbf{q}}^{-1}\hat{\mathbf{q}} = \hat{\mathbf{q}}\hat{\mathbf{q}}^{-1} = 1$ állítás. Az inverzet a norma definíciójából vezetjük le:

$$n(\hat{\mathbf{q}})^2 = \hat{\mathbf{q}}\hat{\mathbf{q}}^* \tag{3.40}$$

$$\Longleftrightarrow$$

$$\frac{\hat{\mathbf{q}}\hat{\mathbf{q}}^*}{n(\hat{\mathbf{q}})^2} = 1, \tag{3.41}$$

amiből megkapjuk a multiplikatív inverzet:

$$\hat{\mathbf{q}}^{-1} = \frac{1}{n(\hat{\mathbf{q}})^2}\hat{\mathbf{q}}^*. \tag{3.42}$$

Könnyen levezethető a definíció alapján, hogy a skalárral való szorzás kommutatív: $s\hat{\mathbf{q}} = \hat{\mathbf{q}}s = (s\mathbf{q}_v, sq_w)$.

**Konjugálási szabályok**

$$\begin{aligned}(\hat{\mathbf{q}}^*)^* &= \hat{\mathbf{q}}\\ (\hat{\mathbf{q}} + \hat{\mathbf{r}})^* &= \hat{\mathbf{q}}^* + \hat{\mathbf{r}}^*\\ (\hat{\mathbf{q}}\hat{\mathbf{r}})^* &= \hat{\mathbf{r}}^*\hat{\mathbf{q}}^*\end{aligned} \tag{3.43}$$

**Normálási szabályok**

$$\begin{aligned}n(\hat{\mathbf{q}}^*) &= n(\hat{\mathbf{q}})\\ n(\hat{\mathbf{q}}\hat{\mathbf{r}}) &= n(\hat{\mathbf{q}})n(\hat{\mathbf{r}})\end{aligned} \tag{3.44}$$

**Linearitás**

$$\begin{aligned}\hat{\mathbf{p}}(s\hat{\mathbf{q}} + t\hat{\mathbf{r}}) &= s\hat{\mathbf{p}}\hat{\mathbf{q}} + t\hat{\mathbf{p}}\hat{\mathbf{r}} \\ (s\hat{\mathbf{p}} + t\hat{\mathbf{q}})\hat{\mathbf{r}} &= s\hat{\mathbf{p}}\hat{\mathbf{r}} + t\hat{\mathbf{q}}\hat{\mathbf{r}}\end{aligned} \tag{3.45}$$

**Asszocivitás**

$$\hat{\mathbf{p}}(\hat{\mathbf{q}}\hat{\mathbf{r}}) = (\hat{\mathbf{p}}\hat{\mathbf{q}})\hat{\mathbf{r}} \tag{3.46}$$

Egy egységkvaternió $\hat{\mathbf{q}} = (\mathbf{q}_v, q_w)$ normája 1-gyel egyenlő ($n(\hat{\mathbf{q}}) = 1$). Ebből következik, hogy $\hat{\mathbf{q}}$ megadható a következő módon:

$$\hat{\mathbf{q}} = (\sin\phi\mathbf{u}_q, \cos\phi) = \sin\phi\ \mathbf{u}_q + \cos\phi, \tag{3.47}$$

ahol $\mathbf{u}_q$ egy három-dimenziós vektor, melynek hossza 1 ($\|\mathbf{u}_q\| = 1$), mivel

$$\begin{aligned}n(\hat{\mathbf{q}}) = n(\sin\phi\mathbf{u}_q, \cos\phi) &= \sqrt{\sin^2\phi(\mathbf{u}_q \cdot \mathbf{u}_q) + \cos^2\phi} \\ &= \sqrt{\sin^2\phi + \cos^2\phi} = 1,\end{aligned} \tag{3.48}$$

akkor és csak akkor, ha $\mathbf{u}_q \cdot \mathbf{u}_q = 1 = \|\mathbf{u}_q\|^2$. Ahogy azt a következő fejezetben látni fogjuk az egységkvaterniók alkalmasak arra, hogy hatékonyan hozzunk létre forgatásokat és iránybeállításokat. Mielőtt viszont erre rátérnénk, néhány műveletet még be kell vezetnünk az egységkvaterniókon.

A komplex számok esetén, egy két dimenziós egység vektor megadható $\cos\phi + i\sin\phi = e^{i\phi}$ formában. Ez alkalmazható a kvaterniók esetén is:

$$\hat{\mathbf{q}} = \sin\phi\ \mathbf{u}_q + \cos\phi = e^{\phi\mathbf{u}_q}. \tag{3.49}$$

A logaritmus és hatvány függvények következnek a 3.49. egyenletből:

**Logaritmus:**

$$\log\hat{\mathbf{q}} = \log(e^{\phi\mathbf{u}_q}) = \phi\mathbf{u}_q, \tag{3.50}$$

**Hatvány:**

$$\hat{\mathbf{q}}^t = (e^{\phi\mathbf{u}_q})^t = e^{\phi t\mathbf{u}_q} = \sin(\phi t)\ \mathbf{u}_q + \cos(\phi t). \tag{3.51}$$

### 3.3.2. Kvaternió-transzformáció

A következőkben az egység hosszú kvaterniókat tanulmányozzuk, melyeket *egységkvaterniónak* nevezünk ezentúl. A legfontosabb dolog az, hogy az egységkvaterniók egy három dimenziós forgatást fejezhetnek ki nagyon tömör és egyszerű alakban.

Először egy pont vagy egy vektor négy koordinátáját $\mathbf{p} = (p_x, p_y, p_z, p_w)^T$ helyezzük el egy $\hat{\mathbf{p}}$ kvaternió komponenseibe és tételezzük fel, hogy van egy egységkvaterniónk $\hat{\mathbf{q}} = (\sin\phi\ \mathbf{u}_q, \cos\phi)$. Ekkor:

$$\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^{-1} \tag{3.52}$$

$\hat{\mathbf{p}}$-t forgatja el (és így $\mathbf{p}$-t is) az $\mathbf{u}_q$-tengely körül $2\phi$ szöggel[14]. Ezt a forgatást bármely tengely körüli forgatás esetén használhatjuk.

$\hat{\mathbf{q}}$ bármely nem nulla szorzása szintén ugyanazt a transzformációt fejezi ki, ami azt jelenti, hogy $\hat{\mathbf{q}}$ és $\hat{\mathbf{q}}^{-1}$ ugyanazt a forgatást hajtja végre. Ez azt is jelenti, hogy egy mátrixból való kvaternió kinyerése akár $\hat{\mathbf{q}}$-val vagy $-\hat{\mathbf{q}}$-val is visszatérhet.

Adott $\hat{\mathbf{q}}$ és $\hat{\mathbf{r}}$ két egységkvaternió. A két kvaternióval $\hat{\mathbf{p}}$-n végrehajtott összetett transzformáció a következőképpen néz ki:

$$\hat{\mathbf{r}}(\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^*)\hat{\mathbf{r}}^* = (\hat{\mathbf{r}}\hat{\mathbf{q}})\hat{\mathbf{p}}(\hat{\mathbf{r}}\hat{\mathbf{q}})^* = \hat{\mathbf{c}}\hat{\mathbf{p}}\hat{\mathbf{c}}^*, \tag{3.53}$$

ahol $\hat{\mathbf{c}} = \hat{\mathbf{r}}\hat{\mathbf{q}}$ szintén egy egységkvaternió, amely a $\hat{\mathbf{q}}$ és $\hat{\mathbf{r}}$ kvaterniók összefűzésével kapott forgatási transzformáció.

**Mátrix átalakítás**

Mivel a mátrixszorzás egyes rendszereken hardveresen támogatott, ezért ezt a fajta szorzást hatékonyabban el lehet végezni, mint a 3.52. egyenletben megadott számítást. Így szükségünk van arra, hogy a kvaterniót oda-vissza át tudjuk alakítani mátrix formába. Egy $\hat{\mathbf{q}}$ kvaternió az $\mathbf{M}^q$ mátrixba a következőképpen alakítható át:

$$\mathbf{M}^q = \begin{pmatrix} 1 - s(q_y^2 + q_z^2) & s(q_x q_y - q_w q_z) & s(q_x q_z + q_w q_y) & 0 \\ s(q_x q_y + q_w q_z) & 1 - s(q_x^2 + q_z^2) & s(q_y q_z - q_w q_x) & 0 \\ s(q_x q_z - q_w q_y) & s(q_y q_z + q_w q_x) & 1 - s(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.54}$$

ahol $s = 2/n(\hat{\mathbf{q}})$. Egységkvaterniók esetén ez a következőképpen egyszerűsödik:

$$\mathbf{M}^q = \begin{pmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_w q_z) & 2(q_x q_z + q_w q_y) & 0 \\ 2(q_x q_y + q_w q_z) & 1 - s(q_x^2 + q_z^2) & 2(q_y q_z - q_w q_x) & 0 \\ 2(q_x q_z - q_w q_y) & 2(q_y q_z + q_w q_x) & 1 - 2(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.55}$$

Amennyiben a kvaterniót ilyen formában előállítottuk, akkor ezután nincs szükség trigonometrikus függvények használatára, így ez az átalakítás a gyakorlatban is hasznos.

A fordított irányú átalakítás egy kicsit bonyolultabb. A kulcsát ennek a műveletnek a következő mátrix elemekből előállított különbségek adják:

$$\begin{aligned} m_{21}^q - m_{12}^q &= 4q_w q_x, \\ m_{02}^q - m_{20}^q &= 4q_w q_y, \\ m_{10}^q - m_{01}^q &= 4q_w q_z. \end{aligned} \tag{3.56}$$

[14] Megjegyezzük, hogy mivel $\hat{\mathbf{q}}$ egységkvaternió ezért $\hat{\mathbf{q}}^{-1} = \hat{\mathbf{q}}^*$.

Ezekből az egyenletekből az következik, hogy ha $q_w$-t ismerjük, akkor a $\mathbf{v}_q$ és így $\hat{\mathbf{q}}$ kiszámítható. A mátrix nyoma:

$$\begin{aligned}\mathrm{tr}(\mathbf{M}^{\mathbf{q}}) &= 4 - 2s(q_x^2 + q_y^2 + q_z^2) = 4\left(1 - \frac{q_x^2 + q_y^2 + q_z^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2}\right) \\ &= \frac{4q_w^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2} = \frac{4q_w^2}{n(\hat{\mathbf{q}})}.\end{aligned} \tag{3.57}$$

Ebből következik, hogy :

$$\begin{aligned} q_w &= \tfrac{1}{2}\sqrt{\mathrm{tr}(\mathbf{M}^q)} \quad & q_x &= \tfrac{m^q_{21} - m^q_{12}}{4q_w} \\ q_y &= \tfrac{m^q_{02} - m^q_{20}}{4q_w} \quad & q_z &= \tfrac{m^q_{10} - m^q_{01}}{4q_w} \end{aligned} \tag{3.58}$$

Numerikusan stabil eljárás eléréséhez el kell kerülnünk a kis számokkal való osztásokat. Ezért legyen $t = q - w^2 - q_x^2 - q_y^2 - q_z^2$, amiből az következik, hogy :

$$\begin{aligned} m_{00} &= t + 2q_x^2, \\ m_{11} &= t + 2q_y^2, \\ m_{22} &= t + 2q_z^2, \end{aligned} \tag{3.59}$$

$$u = m_{00} + m_{11} + m_{22} = t + 2q_w^2, \tag{3.60}$$

amelyből $m_{00}, m_{11}, m_{22}$ és $u$ meghatározza, hogy a $q_x$, $q_y$, $q_z$ és $q_w$ közül melyik a legnagyobb. Amennyiben $q_w$ a legnagyobb, akkor a 3.58. egyenlet segítségével meghatározható a $\hat{\mathbf{q}}$ kvaternió többi komponense. Ellenkező esetben vegyük észre a következő egyenlőségeket:

$$\begin{aligned} 4q_x^2 &= +m_{00} - m_{11} - m_{22} + m_{33} \\ 4q_y^2 &= -m_{00} + m_{11} - m_{22} + m_{33} \\ 4q_z^2 &= -m_{00} - m_{11} + m_{22} + m_{33} \\ 4q_w^2 &= \mathrm{tr}(\mathbf{M}^q). \end{aligned} \tag{3.61}$$

A fenti megfelelő egyenletet alkalmazásával kiszámítjuk a $q_x$, $q_y$ és $q_z$ közül a legnagyobbat, majd a 3.56. egyenletek felhasználásával a maradék $\hat{\mathbf{q}}$ komponenseket is meg lehet határozni.

**Gömbi lineáris interpoláció**

A gömbi lineáris interpoláció egy olyan művelet, amelyet $\hat{\mathbf{q}}$ és $\hat{\mathbf{r}}$ egységkvaternió és egy $t \in [0, 1]$ paraméter esetén egy interpolált kvaterniót számít ki. Ez a művelet hasznos például az objektumok animálásánál, viszont nem annyira hasznos kamera irányítottságok interpolálásakor, mivel a kamera felfele mutató vektor megdőlhet az interpolálás alatt, ami zavaró lehet.

Az algebrai formája egy $\hat{\mathbf{s}}$ összetett kvaternió:

$$\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = (\hat{\mathbf{r}}\hat{\mathbf{q}}^{-1})^t\hat{\mathbf{q}}. \tag{3.62}$$

Mindazonáltal, a szoftveres megvalósításokra a következő forma, ahol a `slerp` a gömbi lineáris interpolációt jelöli, sokkal alkalmasabb:

$$\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \texttt{slerp}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \frac{\sin(\phi(q-t))}{\sin\phi}\hat{\mathbf{q}} + \frac{\sin\phi t}{\sin\phi}\hat{\mathbf{r}}. \tag{3.63}$$

$\phi$ kiszámításához, amelyre az előbbi egyenletben szükségünk lenne, felhasználjuk a $\cos\phi = q_x r_x + q_y r_y + q_z r_z + q_w r_w$ összefüggést. $t \in [0,1]$ -re a `slerp` függvény egyedi[15] interpolált kvaterniókat számít ki, amelyek együtt a legrövidebb ívet alkotják egy négydimenziós egységgömbön $\hat{\mathbf{q}}(t=0)$-tól $\hat{\mathbf{r}}(t=1)$-ig. Az ív a körön helyezkedik el, ami a $\hat{\mathbf{q}}$, $\hat{\mathbf{r}}$ és az origó által meghatározott sík és a négy dimenziós egységgömb metszeteként alakul ki.

A `slerp` függvény tökéletesen alkalmas két orientáció/irányítottság közötti interpolációra. Ezt az eljárást Euler transzformációval elvégezni nem olyan egyszerű, mivel több Euler szög interpolálásakor gimbal lock állhat elő (lásd 3.6. fejezetet).

Amikor kettőnél több orientáció, mondjuk $\hat{\mathbf{q}}_0, \hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_{n-1}$ áll rendelkezésünkre és $\hat{\mathbf{q}}_0$-ból $\hat{\mathbf{q}}_1$-be, majd $\hat{\mathbf{q}}_2$-be egészen $\hat{\mathbf{q}}_{n-1}$-be akarunk interpolálni, akkor a `slerp` függvényt egyszerű módon használhatjuk. Amennyiben a `slerp` függvényt alkalmazzuk minden szakaszon, akkor hirtelen irányváltás látható az orientáció interpolálásakor. Hasonló dolog történik akkor is, amikor lineárisan interpolálunk pontokat.

Jobb, ha úgy interpolálunk, hogy valamilyen spline-t használunk. Vezessük be $\hat{\mathbf{a}}_i$ és $\hat{\mathbf{b}}_{i+1}$ kvaterniókat $\hat{\mathbf{q}}_i$ és $\hat{\mathbf{q}}_{i+1}$ között. Az interpoláció során áthaladunk a kezdeti $\hat{\mathbf{q}}_\mathbf{i}, \mathbf{i} \in [0, \ldots, \mathbf{n}-1]$ orientációkon, de az $\hat{\mathbf{a}}_i$-ken nem. Ezeket arra használjuk, hogy jelezzük az érintőleges irányítottságokat az eredeti orientációknál. Ezt meglepő módon a következőképpen lehet kiszámítani:

$$\hat{\mathbf{a}}_i = \hat{\mathbf{b}}_i = \hat{\mathbf{q}}_i \exp\left[-\frac{\log(\hat{\mathbf{q}}_i^{-1}\hat{\mathbf{q}}_{i-1}) + \log(\hat{\mathbf{q}}_i^{-1}\hat{\mathbf{q}}_{i+1})}{4}\right]. \tag{3.64}$$

A $\hat{\mathbf{q}}_i$-t, $\hat{\mathbf{a}}_i$-t és $\hat{\mathbf{b}}_i$-t a kvaterniók gömbi interpolációjára használjuk sima köbös spline használatával a következő egyenlet szerint:

$$\begin{aligned}&\texttt{squad}(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, \hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t) = \\ &\texttt{slerp}(\texttt{slerp}(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, t), \texttt{slerp}(\hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t), 2t(1-t)).\end{aligned} \tag{3.65}$$

**Egy vektor forgatása egy másikba**

Gyakran szeretnénk olyan transzformációt találni, ami egy $\mathbf{s}$ irányvektort egy $\mathbf{t}$ irányvektorba forgat. Először normalizálnunk kell $\mathbf{s}$-t és $\mathbf{t}$-t. Ezután kiszámítjuk az $\mathbf{u}$ egység forgatási tengelyt az $\mathbf{u} = (\mathbf{s}\times\mathbf{t})/\|\mathbf{s}\times\mathbf{t}\|$ egyenlőség alapján. Legyen $e = \mathbf{a}\cdot\mathbf{t} = \cos(2\phi)$ és $\|\mathbf{s}\times\mathbf{t}\| = \sin(2\phi)$, ahol $2\phi$ az $\mathbf{s}$ és $\mathbf{t}$ közötti szög. A $\hat{\mathbf{q}}$ kvaternió, ami az adott forgatást hajtja

[15] Akkor és csak akkor, ha két kvaternió nem ellentétesek

végre $\mathbf{s}$-ből $\mathbf{t}$-be $\hat{\mathbf{q}} = (\mathbf{u}\sin\phi, \cos\phi)$. Valójában a $\hat{\mathbf{q}} = \left(\frac{\sin\phi}{\sin 2\phi}(\mathbf{s}\times\mathbf{t}), \cos\phi\right)$-t egyszerűsítve felhasználva a fél-szög összefüggéseket és a trigonometrikus azonosságokat kapjuk, hogy:

$$\hat{\mathbf{q}} = (\mathbf{q}_v, q_w) = \left(\frac{1}{\sqrt{2(1+e)}}(\mathbf{s}\times\mathbf{t}), \frac{\sqrt{2(1+e)}}{2}\right). \tag{3.66}$$

Ilyen módon előállítva a kvaterniót, elkerülhetjük a numerikus instabilitást amikor $\mathbf{s}$ és $\mathbf{t}$ szinte ugyanabba az irányba áll. Stabilitási probléma akkor is felléphet, ha $\mathbf{t}$ és $\mathbf{s}$ ellentétes irányba állnak, ekkor nullával való osztás fordul elő. Amikor ilyen esettel állunk szemben, akkor bármely $\mathbf{s}$-re merőleges forgatási tengely használható $\mathbf{t}$ forgatására.

Néha szükség van az $\mathbf{s}$-ből $\mathbf{r}$-be való forgatásmátrix ábrázolására. Néhány algebrai és trigonometriai egyszerűsítés után a 3.55. egyenletet a következő módon alakíthatjuk át:

$$\mathbf{R}(\mathbf{s},\mathbf{t}) = \begin{pmatrix} e+hv_x^2 & hv_xv_y - v_z & hv_xv_z + v_y & 0 \\ hv_xv_y + v_z & e+hv_y^2 & hv_yv_z - v_x & 0 \\ hv_xv_z & hv_yv_z + v_x & e+hv_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{3.67}$$

ahol

$$\begin{aligned} \mathbf{v} &= \mathbf{s}\times\mathbf{t}, \\ e &= \cos(2\phi) = \mathbf{s}\cdot\mathbf{t}, \\ h &= \frac{1-\cos 2\phi}{\sin^2(2\phi)} = \frac{1-e}{\mathbf{v}\cdot\mathbf{v}} = \frac{1}{1+e}. \end{aligned} \tag{3.68}$$

Ahogy látható, az összes négyzetgyök és trigonometrikus függvény eltűnt az egyszerűsítésnek köszönhetően és így hatékonyan lehet a mátrixot előállítani.

Megjegyezzük, hogy azt az esetet külön kell lekezelni, amikor $\mathbf{s}$ és $\mathbf{t}$ ellentétes irányú és párhuzamos vagy közel párhuzamos, mivel ebben az esetben $\|\mathbf{s}\times\mathbf{t}\| \approx 0$ és így egységmátrixot kaphatunk eredményül, ami azt jelenti, hogy nem változik semmi a 180°-os fordulattal ellentétben. Ha $2\phi \approx \pi$, akkor $\pi$ radiánnal tetszőleges tengely körül forgathatunk. Ez a tengely megkapható $\mathbf{s}$ és bármely olyan vektor vektoriális szorzataként, ami nem párhuzamos $\mathbf{s}$-sel.

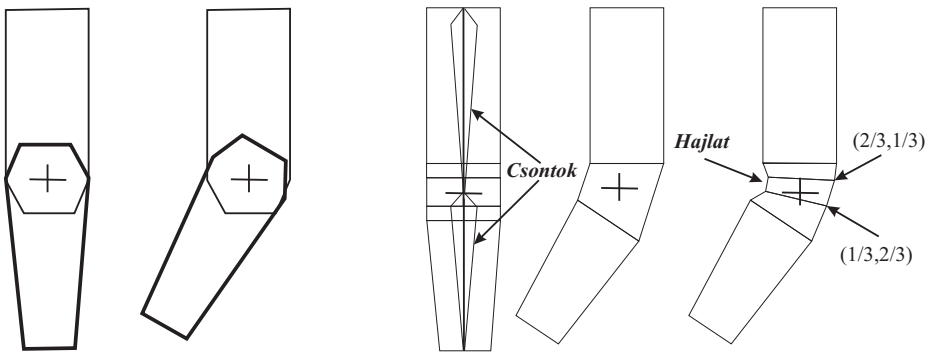
*Példa*

Tegyük fel, hogy a virtuális kamera pozíciója $(0,0,0)^T$ és az alap nézeti irány $\mathbf{v}$ a $z$ tengely negatív része felé néz, azaz $\mathbf{v} = (0,0,-1)^T$. A feladat az, hogy hozzunk létre egy transzformációt, ami a kamerát egy új $p$ pozícióba mozgatja egy új $\mathbf{w}$ irányítottsággal. Kezdjük egy kamera orientációval, ami egy forgatással megoldható a kezdeti irányból a cél nézeti irányba (legyen ez a $\mathbf{R}(\mathbf{v},\mathbf{w})$ forgatás). A pozicionálást egy egyszerű eltolással hajthatjuk végre $\mathbf{p}$ pontba. Így az összetett transzformáció a következőképpen néz ki: $\mathbf{X} = \mathbf{T}(\mathbf{p})\mathbf{R}(\mathbf{v},\mathbf{w})$. A gyakorlatban az első forgatás után még egy vektor-vektor forgatásra lesz szükség a nézeti irány körül nagy valószínűséggel, ami valamilyen kívánt irányba forgatja a nézet felfelé mutató irányát.

## 3.4. Vertex keveredés

Képzeljük el, hogy egy digitális karakter animálásakor az alkarját és felkarját mozgatjuk. Ezt a modellt animálhatjuk merevtest-transzformációval (lásd 3.7.(a) ábra). Azonban a két rész közötti kapcsolódása nem fog hasonlítani egy valódi könyökre. Ez azért van, mivel két elkülönülő objektumot használunk és ezért kapcsolódás a két elkülönülő objektum átfedő részeiből áll.

A *vertex keveredés* egy megoldás erre a problémára (lásd 3.7.(b) ábra). Ezt a technikát nyúzás, beburkolás és csontváz-altér deformáció néven is szokták emlegetni. Szokásos technika az, amikor csontokat definiálnak, melyeket bőr vesz körül és a bőr a csontok mozgásának megfelelően változik a számítógépes animáció során. A mi egyszerű példánkban az al- és a felkart elkülönülve animáljuk, de a két rész kapcsolódásánál egy rugalmas bőrrel kapcsoljuk össze a részeket. Így a rugalmas rész egy vertex halmazát a felkar, míg a másik vertex halmazt az alkar mátrixszal transzformáljuk. Azok a háromszögek, amelyek vertexeit különböző mátrixokkal transzformáltuk, a transzformáció következtében megnyúlnak illetve összemennek. Ezt az alapmódszert néha *varrásnak/tűzésnek* (stitching) nevezik.




*(a) Merevtest-transzformáció* *(b) Vertex keveredés*

*3.7. ábra. Egy kar animálása különböző módon. (a) Az al- és felkar merevtest-transzformációval animálva. A könyök nem tűnik realisztikusnak. (b) A vertex keveredés egy egyszerű objektumon. A középső ábra azt mutatja, amikor a két rész egy egyszerű bőrrel van összekapcsolva. A jobb oldali ábra azt mutatja, hogy mi történik akkor, amikor néhány vertexet keverünk különböző súllyal.* $(2/3, 1/3)$ *azt jelenti, hogy a vertexre* $2/3$ *súllyal a felkar és* $1/3$*-dal az alkar van hatással. A jobb oldali ábrán jól látható a* hajlat*, ami elkerülhető több csont használatával a modellben és gondosabban megválasztott súlyokkal.*

Egy lépéssel tovább haladva, megengedhetjük egy egyszerű vertex esetén azt, hogy több különböző mátrixszal transzformálunk, majd az eredményeket súlyozzuk és keverjük. Ezt akkor hajtjuk végre, amikor az animált objektumnak van egy csontváza, ahol mindegyik csonttranszformáció hatással van mindegyik vertexre egy felhasználó által megadott súly alapján.

Matematikailag ezt a következőképpen írhatjuk le:

$$\mathbf{u}(t) = \sum_{i=0}^{n-1} w_i \mathbf{B}_i(t) \mathbf{M}_i^{-1} \mathbf{p}, \text{ ahol } \sum_{i=0}^{n-1} w_i = 1, \, w_i \geq 0, \tag{3.69}$$

ahol $\mathbf{p}$-vel az eredeti vertexet jelöljük, az $\mathbf{u}(t)$ transzformált vertex pozíciója, amely függ $t$ időtől. A $\mathbf{p}$ vertex pozícióra $n$ csont van hatással, amely pozíciók a modelltérben vannak megadva. Az $\mathbf{M}_i$ mátrix az iniciális csont koordinátarendszerből a világ koordinátarendszerbe transzformál. Egy csont vezérlő kapcsolódását a saját koordinátarendszerének origójában találhatjuk tipikusan. $\mathbf{B}_i(t)$ az $i$-ik csont világ transzformációja, amely az időben változik. Végül a $w_i$ az $i$-ik csont súlya a $\mathbf{p}$ vertex esetén.

Az $\mathbf{M}_i$ mátrix néhány vertex keveredés tárgyalásában nem szerepel explicit módon, inkább $\mathbf{B}_i(t)$ részének tekintik. Az $\mathbf{M}_i$ mátrix inverze modelltérből a csont saját terébe, a $\mathbf{B}_i(t)$ csont aktuális transzformációjával vissza a modellérbe transzformál. A gyakorlatban a $\mathbf{B}_i(t)$ és az $\mathbf{M}_i^{-1}$ mátrixokat összefűzzük mindegyik csont és az animáció mindegyik képkockája esetén és az eredménymátrixot használjuk a vertex transzformáció végrehajtására.

# 4. fejezet

# Modellezés

A 2. fejezetben jól látható volt az, hogy az alakzatokat primitívek (2.8. ábra) segítségével építhetjük fel. Remélhetőleg a jegyzet olvasója azt is érzékelte, hogy a primitíveket felépítő vertexek és az azokhoz kapcsolódó attribútumok egy jól meghatározott folyamat során alakulnak át a képernyő egy ablakában megjelenő raszteres látvánnyá.

Rögzített műveleti sorrendű grafikus csővezeték esetén egy adott geometria kirajzolását nagyon sok dolog, változó értéke befolyásolhatja. A változók gyűjteményét a csővezeték állapotainak nevezzük. Az OpenGL egy állapotmodellt vagy állapotgépet alkalmaz az OpenGL állapotváltozók nyomon követésére. Amikor egy állapot értéke be van állítva, az addig nem változik meg, amíg egy másik függvény meg nem változtatja azt.

Sok állapotnak csak két (`on` vagy `off`) értéke lehet. Például a rejtett felület eltávolításakor (lásd 4.1.1. fejezetet) a mélységellenőrzés vagy be van kapcsolva vagy nincs bekapcsolva. Az ilyen típusú állapot változókat a `glEnable(GLenum Képesség)` függvénnyel kapcsolhatjuk be és a `glDisable(GLenum Képesség)` OpenGL függvény meghívásával kapcsolhatjuk ki. Az adott állapot lekérdezésre a `GLboolean glIsEnabled(GLenum Képesség)` függvényt használhatjuk.

Nem mindegyik állapotváltozó ilyen egyszerű. Sok OpenGL függvény különböző értékeket állít be. Ezeket az értékeket lekérdezni a következő függvényekkel lehet: `glGetBooleanv(GLenum pname, GLboolean *params)`, `glGetDoublev(GLenum pname, GLdouble *params)`, `glGetFloatv(GLenum pname, GLfloat *params)`, `glGetIntegerv(GLenum pname, GLint *params)`.

Ebben a fejezetben bemutatjuk, hogy hogyan hozhatunk létre bonyolultabb alakzatokat. Továbbá, különböző modellezési szabályokat és algoritmusokat is ismertetünk, amelyek egy része az egyértelmű megjelenítéshez, másik része pedig az adott alakzatok „optimális” renderelési sebességének eléréséhez szükséges.

## 4.1. Egy objektum felépítése

Egy objektum felépítése során az objektumot felépítő primitívek vertexeit külön-külön is definiálhatjuk. Ez a módszer egyszerű modellek felépítésekor elfogadható, de egy több ezer primitívből álló alakzatnál már igen sok türelmet igényel. A legjobb megoldás az, hogy ilyen

bonyolult alakzatokat és azok attribútumait egy jól meghatározott struktúrájú adatszerkezetben tároljuk és egyetlen függvényhívás segítségével gondoskodunk azok megjelenítéséről.

Minden bonyolult alakzat/felület létrehozásakor mindig figyelembe kell vennünk bizonyos szabályokat. Az első szabály az, hogy a poligonoknak síkbeli alakzatoknak kell lenniük, vagyis a poligon összes vertexének egy síkon kell feküdnie. Ez jó indok a háromszögek használatára, ugyanis matematikailag három pont egyértelműen meghatároz egy síkot, amennyiben ez a három pont nem egy egyenesen található. A második szabály az, hogy a poligon élei nem metszhetik egymást és a poligonnak konvexnek kell lennie.

A poliéderek az előbbi szabályoknak megfelelnek és bonyolult alakzatok jól közelíthetőek velük. Továbbá az adott alakzatot felépítő poligonok könnyen bonthatóak háromszögekre, amelyre később különböző technikákat is bemutatunk (lásd 8.5.1. fejezet). Azt gondolhatnánk, hogy ezek a szabályok korlátozzák a felhasználók lehetőségeit, de nem, hiszen elég sűrű háromszöghálóval tetszőleges felület jól közelíthető. A poligonok szabályoktól eltérő kezelése nagyon összetetté válhat és ezek az OpenGL-es megszorítások lehetővé teszik, hogy az adott poligonokat nagyon gyorsan rendereljük le.

A következőkben egy tengelyesen szimmetrikus alakzat egyik alkotójának (hengerpalást) a megvalósítását mutatjuk be OpenGL-függvények segítségével, ahol a hengerpalást az origóban helyezkedik el. A hengerpalástot `GL_QUAD_STRIP` (négyszögsáv) OpenGL primitív felhasználásával hozzuk létre (lásd 4.1. kódrészlet). Az objektum szögpontjainak kiszámításakor felhasználjuk azt a tényt, hogy ezek a vertexek lényegében egy-egy körlapon helyezkednek el és csak a $z$ koordinátáiban térnek el a felső és alsó peremei esetén. A körlap pontjainak a meghatározásához a kör egyenletének a polárkoordinátás alakját használjuk ($x = R * \sin(\alpha; y = R * \cos(\alpha)$, ahol $R$ az adott kör sugarát jelöli). Az OpenGL koordináta-rendszer az $x$ tengelyen balról jobbra, az $y$ tengelyen lentről felfele és a $z$ tengelyen pedig hátulról előrefelé növekednek az értékek. A koordináta-rendszer origójának (középpontjának) a helye sok mindentől függ, például a vetületi vagy a nézeti transzformációtól is (lásd 3. fejezetet).

```
1  // A színtér megrajzolását végző callback függvény
2  void RenderScene(void)
3  {
4    // változók a koordináták és szögek tárolására
5    GLfloat x,y,angle;
6    // Flag a szín váltakozására
7    int iPivot = 1;
8
9    // A szín és mélységpuffer törlése
10   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
11
12   // A mélység ellenőrzés bekapcsolása
13   glEnable(GL_DEPTH_TEST);
14
15   // A hátlap poligonként való megjelenítése
16   glPolygonMode(GL_BACK,GL_LINE);
17
18   // A mátrix állapot elmentése és a forgatás végrehajtása
```

```
19    glPushMatrix();
20    glRotatef(-85, 1.0f, 0.0f, 0.0f);
21
22    // ...
23
24    // Négyszögsáv kezdete
25    glBegin(GL_QUAD_STRIP);
26
27    // Egy kör mentén számítjuk ki a henger
28    // palástjának a vertex pontjait
29    for(angle = 0.0f; angle < (2.0f*GL_PI);
30    angle += (GL_PI/8.0f))
31    {
32      // A következő vertex x és y pozíciójának a kiszámítása
33      x = 50.0f*sin(angle);
34      y = 50.0f*cos(angle);
35
36      // A színek váltása zöld és piros között
37      if((iPivot %2) == 0)
38        glColor3f(0.0f, 1.0f, 0.0f);
39      else
40        glColor3f(1.0f, 0.0f, 0.0f);
41
42      // A színváltáshoz szükséges változó
43      // növelése a következő alkalomra
44      iPivot++;
45
46      // A négyszögsáv alsó és felső pontjainak
47      // a megadása; csak a z koordinátákban
48      // térnek el
49      glVertex3f(x, y, -100.0f/2.0);
50      glVertex3f(x, y, 100.0f/2.0);
51    }
52
53    // A hengerpalást rajzolásának a vége
54    glEnd();
55
56    // ...
57
58    // A transzformációs mátrix helyreállítása
59    glPopMatrix();
60
61    // Az elő- és háttérképek cseréje
62    glutSwapBuffers();
63  }
```

*4.1. kódrészlet. Egy hengerpalást modellezése*

Megjegyezzük, hogy a 3D-s alakzatoknál szükség van egy olyan vetítési transzformáció megadására, ami azt határozza meg hogy az alakzat csúcsai hova kerülnek a képernyő/nézeti ablak síkján. Részletes információt a jegyzet 3. fejezetében találhatunk.

A 4.1 (a) ábrán látható a hengerpalást megjelenítése egy ablakban a 4.1. listában lévő OpenGL függvények segítségével. Az ábrán a hengerpalástot alkotó négyszögsáv téglalapjai váltakozva piros és zöld színűek, illetve a hengerpalást belső oldala drótvázas módon van megjelenítve. A következő alfejezetekben a 4.1. kódrészlet specifikus lépéseit ismertetjük[1].

A `RenderScene` függvény elején néhány segédváltozót definiálunk pozíció, szög és a szín nyilvántartására.

```
void RenderScene(void)
{
  // változók a koordináták és szögek tárolására
  GLfloat x,y,angle;
  // Flag a szín váltakozására
  int iPivot = 1;
```

### 4.1.1. Rejtett felületek eltávolítása

Mivel a hengerpalást oldalainak kirajzolása egy adott sorrendben történik, így azok a lapok, amelyek amúgy takarásban vannak, felülírhatják azokat a pixeleket a frame pufferben, amelyek már korábban bekerültek és az adott nézőponthoz közelebb lévő pixelek (lásd 4.1.(b) ábrán a hengerpalást alsó pereme). Ezt a problémát egy egyszerű módszerrel korrigálhatunk, amit *mélységellenőrzésnek* hívnak.

A koncepció nagyon egyszerű: egy kirajzolt pixelhez a hozzá tartozó $z$ értéket is eltároljuk, ami a nézőponttól vett távolságot jelöli. Később, amikor egy másik pixel-t ugyanarra a képernyő/frame puffer pozícióra kellene kirajzolni, az új pixel $z$ értékét összehasonlítjuk azzal a $z$ értékkel, amit már korábban eltároltunk. Ha az új pixel $z$ értéke kisebb, mint a régi pixel $z$ értéke, akkor az azt jelenti, hogy az új pixel közelebb van a nézőponthoz, vagyis takarja a mögötte lévő objektumokat. Ebben az esetben a szín puffer tartalmát frissíteni lehet az új pixel értékkel, valamint az új $z$ értéket is el kell tárolnunk. Ellenkező esetben az történik, hogy mivel az új pixel távolabb van a régi pixeltől, ezért az be sem kerül szín pufferbe és a $z$ értékét sem kell eltárolni. A $z$ értékek tárolását egy szín pufferrel megegyező méretű pufferrel oldják meg.

A mélységpuffer inicializálását a `main` függvényben tehetjük meg a következő módon:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

A `RenderScene` függvény elején egyrészt ugyanúgy törölni kell a mélységpuffert, mint a szín puffer tartalmát a `glClear` függvény segítségével. Ez a törlő érték alap esetben a legtávolabbi lehetséges $z$ érték[2]. Továbbá engedélyezni kell a mélység ellenőrzést a

[1] A kódrészlet 24-ik és 59-ik sorába, a hengerpalásthoz nagyon hasonlóan, a fedőlapok is létrehozhatóak, amelyeket célszerű `GL_TRIANGLE_FAN` OpenGL primitívvel megvalósítani. A háromszög-legyező középpontját (lásd 2.8. ábra) a hengerpalást alsó illetve felső peremének középpontjába kell rakni. A többi vertex koordinátája megegyezik a hengerpalást vertex koordinátáival.

[2] Ez a mi esetünkben 1.


*(a) Mélységellenőrzéssel* *(b) Mélységellenőrzés nélkül*

*4.1. ábra. Hengerpalást megvalósítása*

`glEnable` függvénnyel, mivel csak ebben az esetben történik meg a mélység értékek összehasonlítása.

```
// A szín és mélységpuffer törlése
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// A mélységellenőrzés bekapcsolása
glEnable(GL_DEPTH_TEST);
```

Az előzőekben láttuk azt, hogy nagyon egyszerű módon megoldható a takarásban lévő objektumok eltávolítása az adott színtérről.

Bizonyos esetekben előfordulhat, hogy ideiglenesen fel akarjuk függeszteni a mélységpuffer írását. Ezt a `glDepthMask(GLboolean mask)` OpenGL függvénnyel tehetjük meg `GL_FALSE` bemenő `mask` paraméter értéke esetén. Ebben az esetben a mélység teszt ugyanúgy végrehajtódik a már korábban beírt értékeket felhasználva.

Alapesetben a mélységellenőrzés a kisebb relációt használja a nem látható objektumhoz tartozó pixelek eltávolítására. Az OpenGL lehetőséget biztosít a mélységpuffer összehasonlítás relációjának a megadására is a `glDepthFunc(GLenum func)` függvény segítségével. A függvény lehetséges bemenő összehasonlító relációit a 4.1. táblázatban láthatjuk.

A `GL_EQUAL` és `GL_NOTEQUAL` relációk használata esetén szükség van az alap $[0.0 - 1.0]$ mélység értékek tartományának a megváltoztatására. Ezt a `glDepthRange(GLclampd nearVal, GLclampd farVal)` OpenGL függvény segítségével tehetjük meg, melynek az első paramétere a közeli vágósík, a második paramétere pedig a távoli vágósík leképezését adja meg az ablak koordinátákra nézve (lásd 3. fejezetet). Röviden összefoglalva, a vágás és a homogén koordináták negyedik $w$ elemével való osztás után, a mélység értékek a $[-1.0, 1.0]$ tartományba képződnek le, a közeli és távoli vágósíkoknak megfelelően. A `glDepthRange` adja meg a lineáris leképezését ezeknek a normalizált mélység koordinátáknak az ablak mélységértékeire nézve.

| Reláció | Jelentése |
|---|---|
| GL_NEVER | Mindig hamis. |
| GL_LESS | Igaz, ha a bejövő mélység érték kisebb, mint az eltárolt érték. |
| GL_EQUAL | Igaz, ha a bejövő mélység érték megegyezik az eltárolt értékkel. |
| GL_LEQUAL | Igaz, ha a bejövő mélység érték kisebb vagy egyenlő, mint az eltárolt érték. |
| GL_GREATER | Igaz, ha a bejövő mélység érték nagyobb, mint az eltárolt érték. |
| GL_NOTEQUAL | Igaz, ha a bejövő mélység érték nem egyenlő az eltárolt értékkel. |
| GL_GEQUAL | Igaz, ha a bejövő mélység érték nagyobb vagy egyenlő, mint az eltárolt érték. |
| GL_ALWAYS | Mindig igaz. |

*4.1. táblázat. Összehasonlító relációk*

### 4.1.2. Poligon módok

A 4.1. példánkban a 4.1.(a) ábrán látható módon a hengerpalástunk hátlapja vonalasan/drótvázasan van megadva. Az első kérdés az, hogy hogyan különböztetjük meg egy adott poligon elő- illetve hátlapját, hiszen a példánkban lévő hengerpalást is poligonokból van felépítve.

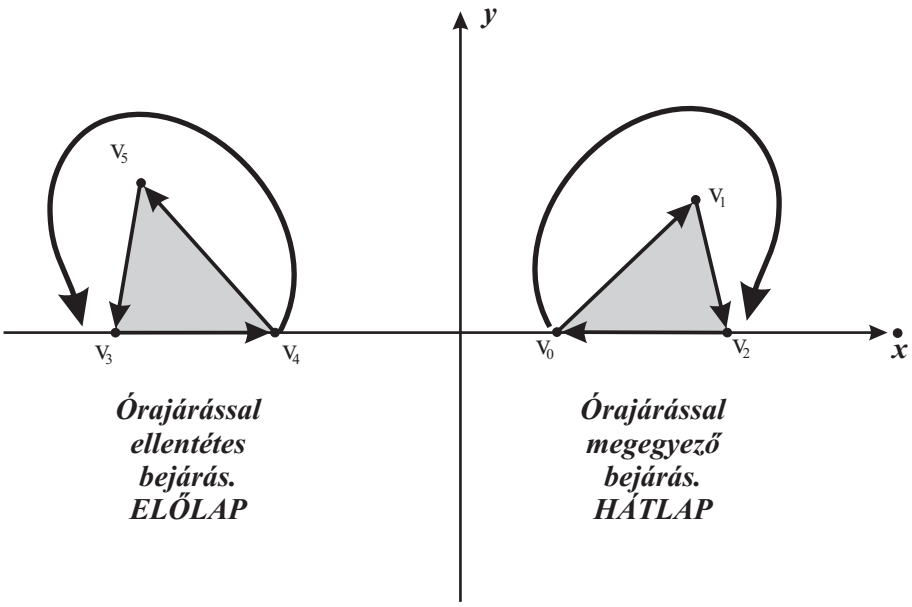
Nos, ezt az OpenGL-ben, az adott primitíveket alkotó vertex/szögpontok sorrendjének a megadása határozza meg, amit nevezzünk *körüljárási iránynak*[3]. Például háromszögek esetén (GL_TRIANGLES) alapesetben, ha veszünk egy jobbsodrású rendszert (nyújtsuk ki a jobb kezünk hüvelyk ujját, a többi ujjunkat pedig görbítsük be), akkor ha a háromszöget alkotó vertexeket a begörbített ujjaink által meghatározott irányban adjuk meg, akkor az adott háromszög előlapja a hüvelykujjunk irányába néz (lásd 4.2. ábrát). Természetesen minden OpenGL primitív esetén ismernünk kell az adott vertexek megadási sorrendjét ahhoz, hogy megfelelő módon tudjuk létrehozni az objektumainkat.

Amennyiben meg szeretnénk változtatni az alap körüljárási irányt, akkor ezt a körüljárási irányt a `glFrontFace(GLenum mode)` OpenGL függvénnyel tehetjük meg, ahol a `mode` paraméter értékei GL_CW és GL_CCW előredefiniált értékek lehetnek. Alapértelmezett érték a GL_CCW. Ezt a tulajdonságot még kihasználjuk a szabályoknak megfelelő poliéderek hátlapjának a figyelmen kívül hagyásakor is (lásd 4.1.4. fejezetet).

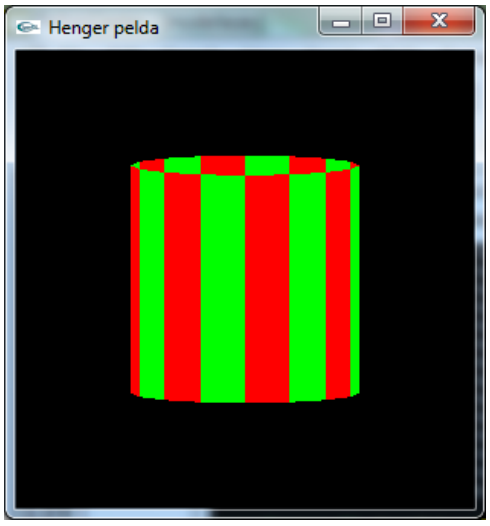
Térjünk vissza az alapproblémára, amikor is a hengerpalástot alkotó négyszögsávok hátlapja drótvázként jelenik meg! Alapesetben a síkbeli OpenGL primitívek kitöltött alakzatokként jelennek meg (4.3. ábra).

Amennyiben ettől eltérő megjelenést akarunk elérni, akkor a `glPolygonMode(GLenum face,GLenum mode)` függvényt kell használnunk, ahol a `face` paraméterrel adjuk meg az adott poligon oldalát. Lehetséges értékei GL_FRONT, GL_BACK és GL_FRONT_AND_BACK lehetnek, melyek az előlapot, hátlapot és elő- és hátlapot jelentik rendre. A második `mode`

[3] OpenGL-ben ezt windings-nek nevezik




*4.2. ábra. Két háromszög vertexeinek megadása két különböző körüljárással*




*4.3. ábra. Hengerpalást megjelenítése kitöltött hátlapokkal*

paraméterrel a poligonok végső raszterizálásának az értelmezését adjuk meg. Négy mód van definiálva:

- `GL_POINT`: a poligon szögpontjai, melyek a határoló élek kezdőpontjai, pontokként vannak kirajzolva. A pont kirajzolási állapotok, mint például a `GL_POINT_SIZE` és a `GL_POINT_SMOOTH` hatással vannak a pontok megjelenésére.

- `GL_LINE`: a poligon határoló élei vonalakként vannak megjelenítve. A `glLineStipple` függvénnyel beállított szaggatott vonalminta és a `GL_LINE_WIDTH` és `GL_LINE_SMOOTH` vonal rajzolási állapotok befolyásolják a vonal kirajzolását.

- `GL_FILL`: a poligon belső területe ki van töltve. A `GL_POLYGON_STIPPLE` and `GL_POLYGON_SMOOTH` poligon rajzolási állapotok hatással vannak a poligon raszterizálására.

A mi esetünkben a következő módon használtuk a `glPolygonMode` függvényt a 4.1. programunkban:

```
// A hátlap poligonkénti megjelenítése
glPolygonMode(GL_BACK, GL_LINE);
```

Az előzőekben említésre kerültek pontokra, vonalakra és poligonokra vonatkozó rajzolási állapotok, melyeket a következőkben ismertetünk.

**Pontméret**

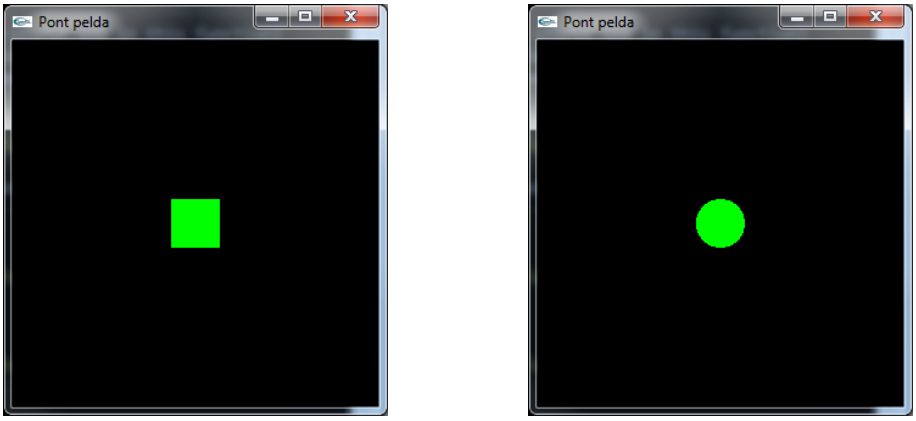
Amikor egy pontot rajzolunk, akkor a pont mérete alapértelmezésben 1 pixel. A `glPointSize(GLfloat size)` függvénnyel tudjuk ezt megváltoztatni, melynek `size` paramétere a pixel közelítő átmérőjét adja meg pixelben. Nem mindegyik méret támogatott, így meg kell győződni arról, hogy az általunk megadott méret rendelkezésre áll-e. A következő kódrészlet segítségével lekérdezhetjük a lehetséges pontméretek minimumát és maximumát, valamint a lépésközt, ami a köztes értékek kiszámítására használható.

```
// a támogatott méretek intervallumának a tárolásra
GLfloat sizes[2];
// a köztes értékek közötti lépésköz tárolására
GLfloat step;

// lehetséges pontméretek minimumának,
// maximumának és a lépésköz lekérdezése
glGetFloatv(GL_POINT_SIZE_RANGE, sizes);
glGetFloatv(GL_POINT_SIZE_GRANULARITY,&step);
```

Az OpenGL specifikáció csak egy pontméret (1.0 pixel) támogatását követeli meg, ezért a különböző megvalósítások között eltérések lehetnek. Amennyiben olyan méretet adunk meg, amely nem támogatott, akkor a hozzá legközelebb lévő értéket fogja használni. Alapértelmezésben a perspektivikus osztás nincs hatással a pontokra. Nem lesznek kisebbek vagy nagyobbak a nézőponttól távolodva vagy közeledve ahhoz. A pontok négyzet alakúak (lásd 4.4. ábrát). Amennyiben kör alakú pontot szeretnénk rajzolni, akkor engedélyeznünk

kell a `glEnable(GL_POINT_SMOOTH)` utasítással a pont simítást[4]. Növelve azok méretét csak nagyobb négyzetet illetve kört kapunk.




*(a) `GL_POINT_SMOOTH` rajzolási állapot engedélyezése nélkül*

*(b) `GL_POINT_SMOOTH` rajzolási állapot engedélyezésével*

*4.4. ábra.* 40-*es méretű pont megjelenítése*

**Vonalvastagság**

Ugyanúgy, ahogy a pontok méretét be lehet állítani, szintén meg lehet adni a különböző méretű vonalvastagságot vonal rajzolásakor a `glLineWidth(GLfloat width)` függvény használatával. A függvény bemenő paraméterével a közelítő értékét adhatjuk meg a beállítani kívánt vonalvastagságnak. Hasonlóan a pontméretének a beállításához, nem támogatott az összes méret. A következő kódrészlet segítségével könnyen lekérdezhetjük a szükséges információkat.

```
// a támogatott méretek intervallumának a tárolásra
GLfloat sizes[2];
// a köztes értékek közötti lépésköz tárolására
GLfloat step;

// lehetséges vonal vastagságok minimumának,
// maximumának és a lépésköz lekérdezése
glGetFloatv(GL_LINE_WIDTH_RANGE, sizes);
glGetFloatv(GL_LINE_WIDTH_GRANULARITY,&step);
```

**Szaggatott vonal**

A vonalvastagságának a beállítása mellett lehetőség van olyan vonalakat létrehozni, amelyeknek szaggatott mintája van, melyet *stipling*-nek neveznek az OpenGL-ben. Ennek a használatához először engedélyeznünk kell ezt a funkciót a `glEnable(GL_LINE_STIPPLE)` függvény meghívásával. Ezek után a `glLineStipple(GLint factor, GLushort pattern)`

[4]Ez a képesség nagyban függ a hardveres megvalósítástól.

függvény segítségével állíthatjuk be az adott vonal szaggatási mintáját, ahol a `pattern` bemenő paraméter egy 16 bit-es `unsigned short`, melynek mindegyik bit-je egy vonalszakaszt reprezentál, ami vagy be van kapcsolva, vagy nem. Mindegyik bit egy pixelnek felel meg alapesetben. A `factor` paraméterrel viszont a minta szélességet adhatjuk meg. Például egy 5-ös `factor` érték beállítása azt jelenti, hogy minden bit 5 egymás utáni pixel-t kapcsol be vagy ki.

A `pattern` paraméter 0-dik (least significant) bit-jét használjuk először a vonal megadására. Ennek az az oka, hogy gyorsabb balra shift-elni az adott mintát a következő maszk érték kinyerésekor.

**Mintával kitöltött poligonok**

Két módszer létezik poligonok mintával való kitöltésére. A szokásos módszer a textúrázás, amikor egy képet húzunk a poligon felületére (lásd a 6. fejezetet). A másik módszer hasonló a szaggatott vonalminta megadásához, kivéve azt, hogy a kitöltési minta tárolására egy megfelelően nagy puffert kell használnunk, amiben egy $32 \times 32$-es bit minta elfér. A bitek olvasását, eltérően a szaggatott vonalmintáktól, a legnagyobb helyi értékű bitekkel kezdjük. A maszkot soronként tároljuk alulról felfele haladva alapesetben. Mindegyik byte-ot balról jobbra olvassuk és egy elég nagy `GLubyte` tömböt kell definiálni ahhoz, hogy 32 darab 4 byte-os sort egyenként el tudjunk tárolni.

Ahhoz, hogy ezt a mintát használni tudjuk, először engedélyeznünk kell a poligon mintával való kitöltését és be kell állítanunk ezt a mintát, mint egy kitöltési mintát. Az adott függvényeket és a hozzájuk tartozó minta definícióját az OpenGL programunk `SetupRC()` függvényében is definiálhatjuk.

```
// A pattern bitmap
GLUbyte pattern =
{ 0x00, 0x00, 0x00, 0x00, ...
  ...                   , 0x00, 0x00, 0x10, 0x00};

// A poligon mintával való kitöltés engedélyezése
glEnable(GL_POLYGON_STIPPLE);

// A poligon kitöltési minta megadása
glPolygonStipple(pattern);
```

Azt tapasztalhatjuk a program futása közben, hogy amennyiben a poligonunkat elforgatjuk, akkor a minta nem fordul a poligonnal együtt. Vagyis a mintát mindig szemből fogjuk látni, bármilyen irányból is nézzünk rá az adott poligonra. Amennyiben azt szeretnénk, hogy a poligonra illesztett kép a poligon felületével együtt mozogjon, akkor a 6. fejezetetben ismertetésre kerülő textúrázást kell használnunk.

**Felosztás és élek**

Habár az OpenGL csak konvex poligonokat tud rajzolni, mégis van lehetőség nem-konvex alakzatok létrehozására, kettő vagy annál több konvex alakzat összeillesztésével. Például a 4.5. ábrán látható alakzat nyilvánvalóan nem konvex, így megsérti az egyszerű poligonokra vonatkozó poligon konstrukciós szabályokat. Az adott alakzatot 4 elkülönülő háromszög[5]

[5] Igazából drótvázas megjelenítés esetén elegendő 3 darab háromszög, amelyek az adott alakzat külső részén

felhasználásával fel lehet építeni, melyek már olyan poligonok, amelyek megfelelnek a szabályoknak.


*(a) Az él flag beállítások nélkül* *(b) Az él flag beállításokkal*

*4.5. ábra. Fogaskerék megvalósítása él flag-ek beállításával és nélküle*

Amikor a poligonok ki vannak töltve, akkor csak az adott alakzat külső élei láthatóak. Amennyiben a `glPolygonMode` függvény segítségével a drótvázas megjelenítést állítjuk be, akkor zavaró lehet az összes apró háromszög megjelenítése egy nagyobb területű felület esetén.

Egy speciális *él flag*-et biztosít az OpenGL a zavaró élek kezelésére. Az él flag beállításával és törlésével a vertex megadásakor jelezzük az OpenGL-nek, hogy mely éleket kell megjeleníteni és melyeket nem. A `glEdgeFlag` függvény egyetlen paramétere állítja az él flag-et `TRUE` vagy `FALSE` értékre. Amikor `TRUE`-ra van állítva, akkor azok a vertexek, amelyeket ezután definiálunk, hozzá fognak tartozni a határvonal szegmenshez.

A következőkben az adott feladat egy általános megoldását adjuk meg, amikor csak egy 2D-s drótvázas „fogaskereket” hozunk létre. Az alapkoncepció hasonló a hengerpalást megvalósításához. Itt is egy kör polárkoordinátás megadásából indulunk ki, amikor is egy külső és egy belső körön haladunk végig és lényegében egy szabályos $n$ szög oldalaira illesztünk háromszögeket úgy, hogy a megfelelő éleket az él flag használatával eltüntetjük. A 4.2. programrészletben a külső háromszögekhez tartozó szögpontok szögeit az `angle`, `Delta_angle` és a `Half_DAngle` szögek határozzák meg, melyeket a 29-ik, 32-ik és 34-ik sorban adunk meg. A 28-ik és 31-ik sorokban lévő él flag beállításokat elhagyva, az adott alakzat a 4.5.(a) ábrán látható módon jelenik meg.

helyezkednek el. Amennyiben kitöltött alakzatot szeretnénk létrehozni, akkor a belső háromszöget is létre kell hoznunk és a drótvázas megjelenítéskor a belső alakzat éleit nem kell megjeleníteni vagyis az él flag-et `FALSE`-ra kell állítani ebben az esetben.

```
1  void fogaskerek(int n, double radius, float x, float y)
2  {
3  // n      – hány oldalú szabályos sokszöggel
4  // közelítünk (n legalább 3)
5  // radius – a sokszög köré írható kör sugara
6  // x, y   – a sokszög köré írható kör
7  // középpontjának koordinátái
8
9    int i;
10 // Az aktuális szögponthoz tartozó szög
11   GLfloat angle;
12 // Két szögpont közötti szögek eltérése
13   GLfloat Delta_Angle;
14 // A szög eltérésének a fele
15   GLfloat Half_DAngle;
16
17   glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
18
19   Delta_Angle = 2.0 * GL_PI / n;
20   Half_DAngle = GL_PI / n;
21
22   if(n < 3)
23     n = 3;
24
25   glBegin(GL_TRIANGLES);
26   for(i = 0, angle = 0.0; i < n; i++, angle += D_Angle)
27   {
28     glEdgeFlag(FALSE);
29     glVertex2f(x + radius * cos(angle),
30     y + radius * sin(angle));
31     glEdgeFlag(TRUE);
32     glVertex2f(x + radius * cos(angle + Delta_Angle),
33     y + radius * sin(angle + Delta_Angle));
34     glVertex2f(x + 2.8*radius * cos(angle + Half_DAngle),
35     y + 2.8*radius * sin(angle + Half_DAngle));
36   }
37   glEnd();
38 }
```

*4.2. kódrészlet. Egy 2D-s fogaskerék modellezése*

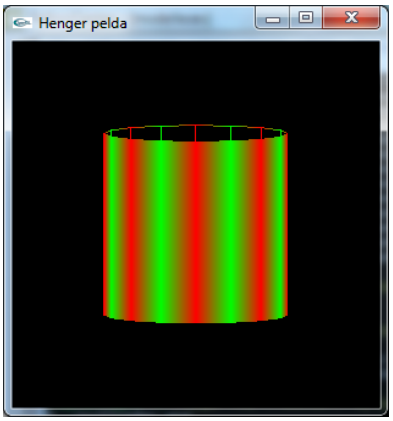
### 4.1.3. Poligon színeinek a beállítása

Eddig a színek használatát csak felületesen érintettük és részletesen csak az 5. fejezetben fogjuk ismertetni. Az OpenGL a színek megadására elkülönített vörös, zöld és kék komponenseket használ, hasonlóan a mai modern PC-k hardveréhez. A szín megadásakor

a színkomponensek telítettségét adjuk meg, és az additív színkeverés elve alapján alakul ki az adott szín[6]. OpenGL-ben a `glColor3f` függvényt használva három darab, 0 és 1 közötti értéket kell megadni a vörös, zöld és kék komponensekre. A `glColor` függvénynek természetesen több változata is létezik, amelyek között van olyan is, amely segítségével például az átlátszóságot szabályozni tudjuk. A függvény használatára láthatunk példát a 4.1. példa program 41-ik és 43-ik sorában.

A `glColor` függvénnyel teljes lapokra vagy alakzatokra és vertexenként is megadhatjuk a szín beállításokat. Az árnyalási modell hatással van a poligonok egyszínű vagy színátmenetes módú kitöltésére. A `glShadeModel(GL_FLAT)` utasítás az egyszínű, a `glShadeModel(GL_SMOOTH)` függvény hívással pedig a színátmenetes kitöltést (lásd 4.6. ábrát) adhatjuk meg.

Általában egyszínű kitöltési módban az adott primitív utolsó vertexénél definiált szín határozza meg az adott alakzat kitöltési színét. Az egyetlen kivétel a `GL_POLYGON` primitív, ahol az első vertex színe határozza meg a poligon színét.




*4.6. ábra. Hengerpalást megjelenítése színátmenetes kitöltéssel*

A színátmenetes kitöltésnek a megvilágításhoz kapcsolódó árnyalásban (lásd 5. fejezetet) is fontos szerepe van, ugyanis így egy folyamatosan változó színt kapunk a háromszögekkel közelített sima felület árnyalásakor adott fénybeállítások mellett, ami a valósághoz megfelelőbb eredményt biztosít.

### 4.1.4. Eldobás

Láthattuk, hogy nyilvánvaló előnyökkel jár az, amikor a takarásban lévő objektumokat nem rajzoljuk ki, még abban az esetben is, ha ez bizonyos plusz költséggel is jár a kirajzolt pixelek $z$ értékeinek és az előzőleg eltárolt fragmens $z$ értékének összehasonlításakor. Noha tudjuk, hogy az adott felület soha nem lesz kirajzolva, akkor miért is adjuk meg? Az *culling/eldobás* az a kifejezés, ami azt a technikát írja le, amellyel azokat a geometriákat írjuk le, amelyekről

[6] Az így megadott szín nem tekinthető végelegesnek, mivel több dolog is (pl. a megvilágítás) befolyásolhatja a képernyőn megjelenő látványt.

tudjuk, hogy soha sem láthatóak. Jelentős teljesítmény javulást érhetünk el azzal, hogy nem küldjük el ezeket a geometriákat az OpenGL meghajtónak és hardvernek.

Az egyik ilyen eldobási technika a *hátsólap-eldobás*, amely elkülöníti a felületek hátterét. A henger példánkban először a henger (lásd 4.1.(a). ábra) hátsó oldalának a belső oldala rajzolódik ki (belül) aztán a külső oldalának a felénk néző poligonjai. A mélységellenőrzésnek köszönhetően a hengerpalást távoli oldalát vagy felülírjuk vagy figyelmen kívül hagyjuk. Mélységellenőrzés nélkül a távoli oldalak poligonjai átütnek/átlátszanak.

Korábban bemutattuk, hogy a körüljárási irány hogyan határozza meg a poligonok elő- és hátlapját. Az objektumok külső oldalának konzisztens megadása nagyon fontos. Ennek a következetes megadásával mondhatjuk meg az OpenGL-nek azt, hogy csak az előlapjait, csak a hátlapjait vagy mind a kettőt renderelje le. Ennek a technikának az alkalmazása különösen fontos tömör objektumok esetén, hiszen ezeknek az alakzatoknak a belső oldalait soha sem látjuk. Természetesen, a mélységellenőrzés is biztosítja az objektum belső oldalainak az „eltakarását”, de a belső oldalak eldobásával drasztikusan tudjuk csökkenteni szükséges mennyiségű feldolgozandó adatot a kép rendereléséhez.

A hátsólap-eldobása alapesetben le van tiltva, ezért ha használni akarjuk ezt a technikát, akkor engedélyeznünk kell a `GL_CULL_FACE` flag-et a `glEnable(GL_CULL_FACE)` utasítással (például a 4.1. program 11-ik sorában). Jól látható az eredeti (4.3. ábra) és a hátsólap-eldobással (4.7. ábra) megjelenített hengerpalástok közötti különbség[7].


*4.7. ábra. Hengerpalást hátsó lapjainak eldobása*

## 4.2. Más primitívek

Az előzőekben a teljesség igénye nélkül bemutattuk, hogy hogyan lehet OpenGL primitívekből bonyolultabb alakzatokat felépíteni. Gyakorlatban a 3D-s grafika egy kicsit több, mint pontok-összekötése számítógép segítségével. A vertexek a 3D-s térben fekszenek és

[7]Elgondolkodtató, hogy ha a hátsólap-eldobás ennyire „kívánatos”, akkor miért van arra szükség, hogy letiltsuk azt. Vannak olyan sík alakzatok (mint például egy papírlap), melyeket mind a két oldalról lehet látni. Továbbá, ha például a hengerpalást műanyagból vagy üvegből készítenénk, akkor az elő- és hátlaphoz tartozó geometriákat is láthatnánk. Átlátszó objektumok létrehozását az 5. fejezetben fogjuk ismertetni.

sík primitívekké állnak össze. A sima görbék és felületek közelítésére is sík poligonokat és árnyalási trükköket használunk. Egy felület annál simábbnak tűnik, minél több poligonnal közelítjük azt.

Az OpenGL további támogatást is biztosít, amelynek segítségével az összetett alakzatok létrehozása könnyebbé válik. A legkönnyebb dolgunk akkor van, ha az OpenGL GLU segéd függvénykönyvtár segítségével állítunk elő gömböket, hengereket, kúpokat és sík korongokat, illetve korongokat lyukkal. Első osztályú támogatást kapunk szabad-formájú felületekhez (Bézier, NURBS), amivel olyan geometriákat is le tudunk írni, amiket egyszerű matematikai képlettel leírható geometria alakzatokkal nem. Végezetül az OpenGL a nagy, rendhagyó konkáv alakzatokat kisebb jobban kezelhető konvex alakzatokra is fel tudja bontani.

### 4.2.1. Beépített felületek

Az OpenGL GLU segéd-függvénykönyvtár, amelyet az OpenGL-lel együtt kapunk meg, több olyan függvényt tartalmaz, amelyek gömböt, hengert és korongot állítanak elő. A henger esetén a felső és alsó sugarakat külön lehet állítani, így az egyik sugár 0-ra való állításával kúpot kapunk. A korong is, a hengerhez hasonlóan, elég rugalmasan paraméterezhető ahhoz, hogy egy lyukas korong (csavaralátét) felületet állítsunk elő vele.

**Kvadratikus felületetek rajzolási állapotok beállítása**

A kvadratikus felületek másodfokú algebrai egyenletekkel leírható felületek, amilyen a gömb, az ellipszis, a kúp, a henger és a paraboloid. Ezekhez a felületekhez ugyanúgy, mint a többi felület esetén, további attribútumokat/tulajdonságokat (pl. normálvektorokat[8], textúra-koordinátákat[9] stb.) rendelhetünk hozzá. Mivel ezeknek a tulajdonságoknak függvényparaméterként történő megadása, egy függvény esetén, igen nagy paraméter listát eredményezne, ezért a kvadratikus alakzatok függvényeinél objektum-orientált modellhez hasonló módszert használ az OpenGL. Vagyis egy kvadratikus objektum létrehozása után, a rendereléshez szükséges attribútumok beállítását további függvények meghívásával végezhetjük el. A következő kódrészlet azt mutatja be, hogy hogyan lehet létrehozni egy üres kvadratikus objektumot és hogyan lehet azt később kitörölni.

```
GLUquadricObj *pObj;
// . . .
// Kvadratikus objektum létrehozása és inicializálása
pObj = gluNewQuadric();
// Renderelési paraméterek beállítása
// Kvadratikus felület rajzolása
// . . .
// Kvadratikus objektum felszabadítása
```

[8] A normálvektoroknak a megvilágításban van fontos szerepük, amit az 5. fejezetben fogunk bővebben ismertetni. Egyelőre elégedjünk meg azzal a definícióval, hogy a normálvektor az adott síkra merőleges egység vektor.

[9] A textúra-koordináták határozzák meg a textúrázásban (lásd a 6. fejezetet), hogy az adott alakzat felületét meghatározó vertexhez a kép melyik pixelét rendelje hozzá, amely már meghatározza, hogy a belső pontok milyen pixel értékekkel lesznek kitöltve később a raszterizálás során.

```
gluDeleteQuadric(pObj);
```

Négy függvény segítségével állíthatjuk be a `GLUQuadricObj` objektum rajzolási állapotait. Az első függvény a rajzolási stílust állítja be:

```
gluQuadricDrawStyle(GLUquadricObj *obj, GLenum drawStyle)
```

Az első pataméter a kvadratikus objektumra mutató pointer. A második paraméter lehetséges értékeit a 4.2. táblázat adja meg.

| **Konstans** | **Leírás** |
|---|---|
| `GLU_FILL` | Solid objektumként jelenik meg. |
| `GLU_LINE` | Drótvázas alakzatként jelenik meg. |
| `GLU_POINT` | Vertex pontok halmazaként jelenik meg. |
| `GLU_SILHOUETTE` | Hasonló a drótvázas megjelenéshez, de a poligonok szomszédos élei nem jelennek meg. |

*4.2. táblázat. Kvadratikus rajzolási stílusok*

A következő függvénnyel azt állíthatjuk be, hogy a kvadratikus objektumokhoz automatikusan generálódjanak-e le a felülethez tartozó normál egységvektorok:

```
gluQuadricNormals(GLUquadricObj *pbj, GLenum normals)
```

A kvadratikus objektumokat normálvektorok nélkül (`GL_NONE`), sima normálokkal (`GL_SMOOTH`), valamint sík normál (`GL_FLAT`) vektorokkal állíthatjuk elő. A legfőbb különbség a sima és sík normálvektorok között az, hogy mindegyik vertexhez egy normálvektor van megadva, amely merőleges a közelítendő felületre, ami így kisimított megjelenést biztosít az adott felületnek. A sík normálvektorok esetén mindegyik normálvektor az adott háromszögre, illetve négyszögre merőleges.

Szintén megadható, hogy a normálvektorok kifele vagy befele mutassanak. Egy megvilágított gömb esetén a normálvektoroknak kifele kell mutatniuk. Amennyiben a gömb belsejét látjuk (például egy boltíves mennyezet esetén), akkor a megvilágítási számításoknál a normálvektoroknak befelé kell mutatniuk. A következő függvény a korábban említett paramétereket állítja be:

```
gluQuadricOrientation(GLUquadricObj *obj, GLenum orientation)
```

Ebben az esetben az `orientation` paraméter vagy a `GLU_OUTSIDE` vagy a `GLU_INSIDE` értékeket veheti fel. Alapértelmezésben a kvadratikus felületek irányítottsága az órajárással ellentétes. A gömbök és a hengerek esetén a külső felület intuitív, a korong esetén a pozitív $z$ tengely felé mutat a külső oldal.

Végül, kérhetjük a textúra-koordináták kiszámolását a következő függvény segítségével:

```
gluQuadricTexture(GLUquadricObj *obj, GLenum textureCoords)
```

melynek második paramétere vagy `GL_TRUE` vagy `GL_FALSE` értékek lehetnek. A legenerált textúra-koordináták egyenletesen helyezkednek el a gömb és a henger felületén. Korong esetén a textúrakép középpontja megegyezik a korong középpontjával.

**Kvadratikus alakzatok rajzolása**

A megfelelő paraméterbeállítás után, mindegyik felület kirajzolása egy egyszerű függvényhívással történik. Például, egy gömb megrajzolásához a következő függvényt kell meghívni:

```
gluSphere(GLUQuadricObj *obj, GLdouble radius, GLint slices,
GLint stacks)
```

Az első `obj` paraméter az előzőleg beállított kvadratikus objektumra mutató pointer. A `radius` bemenő paraméter a gömb sugarát adja meg, amit a `slices` és `stacks` paraméterek követnek. A gömb háromszögsáv (vagy négyszögsáv, attól függően, hogy GLU könyvtár melyik megvalósítását használjuk) gyűrűkből épül fel, amelyek alulról felfele haladva rakódnak egymásra. A `slices` paraméter azt határozza meg, hogy hány háromszög (négyszögek )halmazt használ egy gyűrűn belül. A `stacks` paraméter pedig a gyűrűk számát adja meg. Lényegében a `stacks` paraméter a szélességi és a `slices` paraméter pedig a hosszúsági köreinek a száma.

A henger a pozitív $z$ tengely mentén van összeállítva egymásra pakolt sávokból. A `gluCylinder` függvény paraméterezése hasonló az imént ismertetett `gluSphere` függvényéhez:

```
gluCylinder(GLUquadricObj *obj, GLdouble baseRadius,
GLdouble topRadius, GLdouble height, GLint slices, GLint stacks)
```

A `baseRadius` az origóhoz közelebbi, a `topRadius` pedig a másik oldalhoz tartozó sugarát adják meg a hengernek. Amennyiben az egyik oldal sugarát nullára állítjuk, akkor kúpot kapunk eredményül. A `height` paraméter a henger magasságát adja meg. Az utolsó két paraméter a gömbhöz hasonlóan a szeletek és a gyűrűk számát adják meg.

Az utolsó kvadratikus felszín a korong. A korongot négyesek vagy háromszögsávok gyűrűiként rajzoljuk meg, melyek szeletekre vannak osztva.

```
gluDisk(GLUquadricObj *obj, GLdouble innerRadius,
GLdouble outerRadius, GLint slices, GLint loops)
```

Amennyiben a belső sugár (`innerRadius`) nullával egyenlő, akkor egy tömör korongot kapunk, ellenkező esetben egy lyukas korong lesz az eredmény. A korong az $xy$ síkban van elhelyezve alap esetben.

### 4.2.2. Bézier görbék és felületek

A görbéket és egyeneseket parametrikus egyenletekkel is meg lehet adni, ahol $x$, $y$ és $z$ egy másik változó függvényeként van megadva, ami egy előre definiált intervallum értékeit veheti fel. Például egy részecske időbeli mozgását a következő módon adhatjuk meg:

$$\begin{aligned} x &= f(t), \\ y &= g(t), \\ z &= h(t), \end{aligned} \tag{4.1}$$

ahol $f(t)$, $g(t)$ és $h(t)$ egyedi függvények.

Amikor OpenGL-ben egy görbét definiálunk, akkor szintén parametrikus görbeként adjuk meg azt. A görbe paraméterét $u$-val jelöljük. Felület esetén $u$ és $v$ paraméterek használjuk.

**Kontrollpontok**

A görbék kontrollpontok segítségével adhatunk meg, melyek befolyásolják annak az alakját. A Bézier görbe első és utolsó kontrollpontja valójában része a görbének. A többi kontrollpont mágnesként viselkedik, melyek maguk felé húzzák görbét. A görbe *rangját* a kontrollpontok száma határozza meg. A görbe *foka* eggyel kisebb, mint annak a rangja. A matematikai jelentése ezeknek a fogalmaknak a parametrikus polinom egyenletekre vonatkozik, amelyek pontosan leírják az adott görbét, a rang az együtthatók száma, a fok pedig a legnagyobb kitevője a paraméternek. A kubikus (harmadfokú) görbék a leggyakoribbak.

**Folytonosság**

Amikor két görbét egymás mellé helyezünk, akkor ezek egy végpontban (töréspontban) megegyeznek. A folytonossága ezeknek a görbéknek ebben a töréspontban leírja azt, hogy mennyire sima az átmenet közöttük. A folytonosságnak négy kategóriája van: semmilyen, pozícióbeli (C0), érintőleges (C1) és görbületi (C2). Amikor a két görbének nincs közös pontja, akkor semmilyen folytonosság nem áll fenn közöttük. Pozícióbeli folytonosságot akkor érünk el, ha a két görbe egy közös végpontban találkozik. Amikor a két végpontban a két görbe érintője azonos, akkor beszélünk érintőleges folytonosságról. Végül, a görbületi folytonosság azt jelenti, hogy a két görbületi sugara is megegyezik a töréspontban (ilyen módon még simább az átmenet közöttük).

**Kiértékelők**

Az OpenGL számos olyan függvényt tartalmaz, amelyek megkönnyítik a Bézier görbék és felületek rajzolását. A megrajzolásukhoz megadjuk a kontrollpontokat és az $u$ és $v$ paraméterek tartományait. Ezután, egy megfelelő kiértékelő függvény meghívásával, az OpenGL előállítja azokat a pontokat, amelyek a görbe vagy a felület pontjait alkotják. Először a 2D-s Bézier görbére mutatunk be egy példát, majd kiterjesztjük azt egy 3 dimenziós Bézier felület előállítását szemléltető példává.

**2D-s görbék**

A következőkben lépésről lépésre mutatjuk be, hogy hogyan lehet egy 2D-s Bézier görbét előállítani az OpenGL segítségével.

Az első dolog, amit meg kell tennünk, a görbe kontrollpontjainak a definiálása:

```
//A kontrollpontok száma
GLint nNumPoints = 5;

GLfloat ctrlPoints[5][3]=
  // Végpont
  {{ -3.0f, -3.0f, 0.0f},
  // Kontrollpont
  { 5.0f, 3.0f, 0.0f},
  // Kontrollpont
  { 0.0f, 6.0f, 0.0f},
  // Kontorllpont
  { -0.5f, 6.0f, 0.0f},
```

```
  // Végpont
  { 0.0f, -8.0f, 0.0f }};
```

A renderelés során, például a `RenderScene` függvényben a képernyő törlése után először a glMap1f függvényt hívjuk meg, ami előállítja a görbénk leképezését.

```
glMap1f(
  // Az előállított adat típusa
  GL_MAP1_VERTEX_3,
  // u alsó korlátja
  0.0f,
  // u felső korlátja
  100.0f,
  // A pontok közötti távolság az adatokban
  3,
  // Kontrollpontok száma
  nNumPoints,
  // Kontrollpontokat tartalmazó tömb mutatója
  &ctrlPoints[0][0]);
```

A függvény első paramétere megadja a kiértékelőnek, hogy a vertex koordináta-hármasokat ($x$, $y$ és $z$) állítson elő (`GL_MAP1_VERTEX_3`). Lehetőség van még arra is, hogy textúrakoordinátákat vagy színinformációkat generáljon számunkra.

A következő két paraméter adja meg az a görbe $u$ paraméterének alsó és felső határát. Az alsó érték az első pontot, a felső érték pedig az utolsó pontot határozza meg. Az görbe összes többi pontjának az alsó és felső határ közötti értékek felelnek meg.

A negyedik paramétere a `glMap1f` függvénynek megadja a vertexek közötti lebegőpontos értékek számát a kontrollpontokat tartalmazó tömbben. Mindegyik vertex 3 lebegőpontos számból épül fel, ($x$, $y$ és $z$) ezért ezt az értéket 3-ra állítjuk be. Az utolsó paramétere a függvénynek a kontrollpontokat tartalmazó tömbre mutató pointer, amely definiálja a görbét.

Miután a görbe leképezését megadtuk, engedélyezni kell a kiértékelőnek, hogy használja az adott leképezést. Ez egy állapotváltozón keresztül van nyilvántartva, amit a következő függvényhívással tudunk engedélyezni:

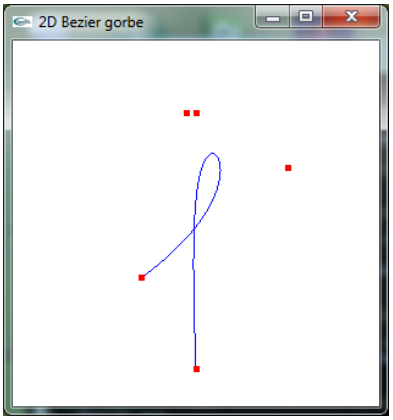
```
// A kiértékelő engedélyezése
glEnable(GL_MAP1_VERTEX_3);
```

A `glEvalCoord1f` függvénynek egyetlen paramétere van, amelyben a parametrikus értéket adjuk meg a görbe mentén. Ez a függvény kiértékeli a görbét erre az értékre és belül meghívja a `glVertex` függvényt a kiszámított vertex koordinátákra. Végig haladva a görbe értelmezési tartományán és a `glEvalCoord1f` függvényt meghívva, egy egyszerű töröttvonallal megrajzolhatjuk az adott görbét.

```
// A pontok összekötése töredezett vonallal
glBegin(GL_LINE_STRIP);
for(i = 0; i <= 100; i++)
{
// A görbe kiértékelése az adott pontban
  glEvalCoord1f((GLfloat) i);
```

```
}
glEnd();
```

A végeredmény a 4.8. ábrán látható, ahol a kontrollpontokat is megjelenítettük piros színnel.




*4.8. ábra. Bézier görbe a kontrollpontokkal együtt megjelenítve*

**Egy görbe kiértékelése**

A fentieket egyszerűbben is meg lehet valósítani az OpenGL-ben. A `glMapGrid` függvény segítségével beállítunk egy rácsot, amely megmondja az OpenGL-nek, hogy hogyan helyezze el egyenletesen az $u$ értelmezési tartományon a rácspontokat (a görbe parametrikus argumentumát). Ezek után a `glEvalMesh` függvény hívással összekötjük a pontokat a megadott primitívek (`GL_LINE` vagy `GL_POINTS`) használatával.

```
// Leképezi a 100 pont rácsát a 0 - 100 intervallumra
glMapGrid1d(100,0.0,100.0);

// Kiértékeli a rácsot és vonalakkal megjeleníti azt
glEvalMesh1(GL_LINE,0,100);
```

Ez a megközelítés hatékony és kompakt, de igazából felületek kiértékelésénél érezhető az előnye.

**Egy 3D-s felület**

Háromdimenziós Bézier görbe létrehozása nagyjából megegyezik a 2D-s Bézier görbe létrehozásával. A pontokat az $u$ értelmezési tartományán kívül a $v$ értelmezési tartományán is definiálni kell. A következőkben egy 3D-s drótvázas Bézier felület előállítását mutatjuk be.

A kontrollpontok megadásához egy $n \times n$-es (jelen esetben $3 \times 3$-as) mátrixot használunk, aminek minden eleme egy 3 elemű ($x$, $y$ és $z$ értékeket) tartalmazó vektor.

```
GLfloat ctrlPoints[3][3][3]=
    {{{ -4.0f, 0.0f, 4.0f},
    // ...
    { 4.0f, 0.0f, -4.0f }}};
```

A `glMap1f` függvény helyett a `glMap2f` függvényt használjuk. Ezzel a függvényhívással adjuk meg a két értelmezési tartomány ($u$ és $v$) mentén a kontrollpontokat.

```
glMap2f(
    // Az előállított adat típusa
    GL_MAP2_VERTEX_3,
    // u alsó korlátja
    0.0f,
    // u felső korlátja
    10.0f,
    // A pontok közötti távolság az adatokban
    3,
    // Dimenzió u irányban (rang)
    3,
    // v alsó korlátja
    0.0f,
    // v felső korlátja
    10.0f,
    // A pontok közötti távolság az adatokban
    9,
    // Dimenzió u irányban (rang)
    3,
    // Kontrollpontokat tartalmazó tömb mutatója
    &ctrlPoints[0][0][0]);
```

Az $u$ alsó és felső korlátját is meg kell adnunk, valamint a pontok közötti távolságot az $u$ értelmezési tartományban. Szintén definiálnunk kell a $v$ értelmezési tartományát. A $v$ értelmezési tartományban a pontok távolsága kilenc ebben az esetben, mivel egy 3 dimenziós kontrollpontokat tartalmazó tömbünk van, az $u$ értelmezési tartomány mindegyik sorában 3 pont 3 értékével ($3 \times 3 = 9$). Ezután megmondjuk, hogy hány pont van definiálva a $v$ irányban mindegyik $u$ felosztás esetén, ami után a kontrollpontokra mutató pointert adjuk meg.

A kétdimenziós kiértékelőt ugyanúgy kell engedélyeznünk, mint az egydimenziós változatát és meghívjuk a `glMapGrid2f` függvényt az $u$ és $v$ irányban lévő felosztások számával:

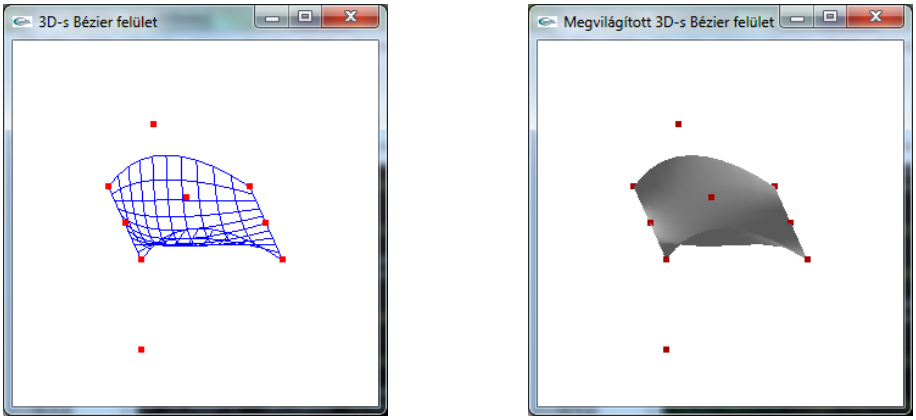
```
// Kiértékelő engedélyezése
glEnable(GL_MAP2_VERTEX_3);

// Magasabb szintű függvény a rács leképezésére
// A rács 10 pontjának a leképezése
// a 0 - 10 tartományra
glMapGrid2f(10, 0.0f, 10.0f, 10, 0.0f, 10.0f);
```

A kiértékelő paraméterezése után meghívhatjuk 2 dimenziós verzióját a `glEvalMesh` függvénynek a felület rács kiértékeléséhez. Itt vonalak használatával értékeljük ki a felületet és az $u$ és $v$ értelmezési tartományt a $[0, 10]$ intervallumra korlátozzuk.

```
// A rács kiértékelése vonalakkal
glEvalMesh2(GL_LINE,0,10,0,10);
```

Az eredményt a 4.9.(a). ábrán láthatjuk. Egy másik hasznos tulajdonsága a kiértékelőknek az, hogy automatikusan számolja ki a felületi normálvektorokat a `glEnable(GL_AUTO_NORMAL)` függvény meghívásával. Ezek után a rács kiértékelésében `GL_LINE` helyett `GL_FILL`-t megadva és egy egyszerű megvilágítást használva (5. fejezet) a 4.9.(b). ábrán látható eredményt kapjuk.




*(a) Drótvázas megjelenítés*

*(b) Megvilágított, kitöltött megjelenítés*

*4.9. ábra. 3D-s Bézier felület*

### 4.2.3. GLUT-os objektumok

A GLUT függvénykönyvtár tartalmaz olyan eljárásokat, amelyek segítségével könnyedén lehet 3D-s geometriai objektumokat létrehozni. Ezeket az objektumokat egyszerű OpenGL renderelő eljárásokkal is meg lehet valósítani. Ezek az eljárások nem generálnak display listákat (lásd 8.1. fejezet) és textúra-koordinátákat (kivéve a teáskanna esetében) az adott objektumok számára, viszont a megvilágításhoz szükséges normálvektorokat előállítják. A következő táblázat (4.3. táblázat) foglalja össze a GLUT-os függvényhívással előállítható objektumokat, melyek paramétereit terjedelmi okokból itt nem részletezzük[10].

[10] Bizonyos objektumok esetén (pl. gömb és kúp) a GLUT-os megvalósítás a korábban ismertetett kvadratikus objektumokat megvalósító függvényeket használja. Ennek következményeként az adott GLUT-os objektumok paraméterlistája nagyon hasonló a kvadratikus objektumokat megvalósító függvényekhez.

| Tömör alakzat | Drótvázas alakzat | 3D-s objektum |
|---|---|---|
| `glutSolidSphere` | `glutWireSphere` | Gömb |
| `glutSolidCube` | `glutWireCube` | Kocka |
| `glutSolidCone` | `glutWireCone` | Kúp |
| `glutSolidTorus` | `glutWireTorus` | Tórusz |
| `glutSolidDodecahedron` | `glutWireDodecahedron` | Dodekaéder |
| `glutSolidOctahedron` | `glutWireOctahedron` | Oktaéder |
| `glutSolidTetrahedron` | `glutWireTetrahedron` | Tetraéder |
| `glutSolidIcosahedron` | `glutWireIcosahedron` | Ikozaéder |
| `glutSolidTeapot` | `glutWireTeapot` | Teáskanna |

*4.3. táblázat. 3D-s GLUT-os objektumok*

# 5. fejezet

# Árnyalás

Amikor egy háromdimenziós világ fotorealisztikus képét állítjuk elő, akkor a modellnek nem csak geometriailag, hanem külső megjelenésének is valósághűnek kell lennie. Ezt gyakran különböző technikák (anyagi tulajdonságok felületekhez való hozzárendelése; különböző fajta fényforrások alkalmazása; textúrák hozzáadása; köd, átlátszóság használata stb.) kombinálásával hajtjuk végre.

## 5.1. Fényforrások

A tárgyakat azért látjuk, mivel fotonok verődnek vissza (vagy bocsátódnak ki) a tárgyak felületéről és a szemünkbe érkezve érzékeljük azokat. Ezek a fotonok vagy egy tárgy felületéről, vagy *fényforrásokból* származhatnak. Ebben a környezetben három fényforrástípust különböztetünk meg: irányított fények, pontfények és reflektorfények. Az 5.3. fejezetben mutatjuk be azt a módszert, ahogyan a különböző típusú fényforrások és a felületek paramétereinek a felhasználásával a visszavert fényt határozzuk meg.

Az irányított fény esetén feltesszük, hogy az egy, az objektumoktól végtelen távoli pontban helyezkedik el a megvilágított objektumoktól. Ilyen fényforrás például a nap. A pont- és reflektorfényeket *pozícionális* fényforrásoknak nevezzük, mivel mind a kettő rendelkezik egy pozícióval a térben.

Mind a három fényforrás esetén megadhatunk intenzitásra vonatkozó paramétereket és szín (RGB) értékeket. A fény tovább bontható ambiens, diffúz és spekuláris intenzitásokra, melyeket később fogunk definiálni. Ezeket a mennyiségeket az 5.1. táblázatban foglaljuk össze, ahol s a *forrás* (angolul *source*) rövidítése.

| Jelölés | Leírás |
|---|---|
| $\mathbf{s}_{amb}$ | Ambiens intenzitás szín |
| $\mathbf{s}_{diff}$ | Diffúz intenzitás szín |
| $\mathbf{s}_{spec}$ | Spekuláris intenzitás szín |
| $\mathbf{s}_{pos}$ | Négy elemű fényforrás pozíció |

*5.1. táblázat. Fényforrásokra vonatkozó paraméterek*

Ez a fajta felosztás fizikailag nem pontos (a valódi fények csak egy egyszerű intenzitással és színnel rendelkeznek), de lehetőséget ad a felhasználónak, hogy a valósidejű grafikai alkalmazásokban nagyobb ellenőrzést gyakoroljon a színtér megjelenésében.

A reflektorfénynek van még néhány paramétere. Először is van egy $\mathbf{s}_{dir}$ irányvektora, amerre a reflektor mutat. Rendelkezik egy $\mathbf{s}_{cut}$ *levágási-szöggel* (cut-off angle) is, ami a reflektor kúp szögének a fele. A kúpon belüli elnyelődés kontrollálására a $\mathbf{s}_{exp}$ *spot-exponent* paramétert használhatjuk, amelyet a fény eloszlásának koncentrálására használhatunk a kúp középpontjában és a középponttól távolodva pedig csökkenthetjük annak a hatását. További paraméterek segítségével befolyásolhatjuk a reflektorfény hatását.

A valóságban a fény hatása a távolság négyzetével arányosan csökken. A valósidejű rendeleléskor egyszerűbb megvalósítani azt a modellt, amikor a fények erőssége nem csökken a távolsággal. Az ilyen fényeket könnyebben lehet kezelni és a hatásukat is gyorsabban lehet kiszámítani. Az irányított fényforrások definíció szerint nem rendelkeznek a távolsághoz kapcsolódó hatással, hiszen végtelen messze vannak a tárgyaktól, bár a pozícionális fényforrásoknál a távolság alapú intenzitás vezérlést is lehet alkalmazni. Például az OpenGL az $\mathbf{s}_c$, $\mathbf{s}_l$ és $\mathbf{s}_q$ paramétereket használja az csillapítás vezérlésére, melyeket az 5.3.4. fejezetben fogunk részletesen ismertetni.

## 5.2. Anyagi tulajdonságok

A valósidejű rendszerekben az anyag ambiens diffúz, spekuláris, fényesség/csillogás és emissziós paraméterekkel adható meg (lásd 5.2 táblázatot). Egy felület színét ezekkel az anyaghoz tartozó paraméterekkel, a fényforrások paramétereivel (melyek megvilágítják a felületet) és egy megvilágítási modellel határozhatjuk meg. Ezt a következő fejezetben ismertetjük.

| Jelölés | Leírás |
|---|---|
| $\mathbf{m}_{amb}$ | Ambiens anyag szín |
| $\mathbf{m}_{diff}$ | Diffúz anyag szín |
| $\mathbf{m}_{spec}$ | Spekuláris anyag szín |
| $\mathbf{m}_{shi}$ | Fényesség paraméter |
| $\mathbf{m}_{emi}$ | Emisszív anyag szín |

*5.2. táblázat. Anyagi konstansok*

## 5.3. Megvilágítás és árnyalás

A *megvilágítás* egy olyan kifejezés, amelyet arra használunk, hogy megadjuk az anyag és fényforrások paramétereivel meghatározott látható szín értékeit. Ahogy látni fogjuk később, a megvilágítás kapcsolatban van a színekkel, textúrákkal és az átlátszósággal is. Ezen elemek vannak egyesítve a képernyőn megjelenő objektumok felületein.

Az *árnyalás* az a folyamat, amely végrehajtja a megvilágítási számításokat és meghatározza azokból a pixelek színeit. Három fő típust különböztetünk meg: flat (sík), Gouraud és Phong. Ezek kapcsolatban vannak azzal, hogy a megvilágítást poligononként, vertexenként vagy pixelenként számítjuk ki. A flat árnyalásnál a szín egy háromszögre van kiszámítva és a háromszög ezzel a színnel van kitöltve. A Gouraud árnyalás esetén a megvilágítás a háromszög mindegyik vertexe esetén meg van határozva és ezeket a színeket interpolálva a háromszög felületén kapjuk a végső eredményt. A Phong árnyalásnál a vertexekben tárolt árnyalási normálvektorokat interpolálva határozzuk meg a pixelenkénti normálvektorokat a háromszögben. Ezeket a normálvektorokat használva számítjuk ki a megvilágítás hatását az adott pixelben.

A flat árnyalást gyors és könnyű megvalósítani. Ugyanakkor nem ad olyan sima eredményt görbe felület esetén, mint a Gouraud árnyalás. Ez előny lehet akkor, ha a felhasználó meg szeretné különböztetni a modellt felépítő primitíveket illetve felületeket. A legtöbb grafikus hardveren a Gouraud árnyalás meg van valósítva a sebessége és a javított minősége miatt. Az a probléma ezzel a technikával, hogy ez az árnyalás nagyban függ a megjelenítendő objektumok részletességétől. Az 5.1. ábrán látható, hogy a kevés poligonnal (nagy méretű háromszögekkel) megvalósított gömb megvilágításakor a csúcsokban számolt színértékek interpolálása miatt nem kapunk olyan sima átmenetes árnyalást, mint az adott gömb nagy számú poligonnal való modellezése esetén.

A Phong árnyalás a felületi normálvektorok interpolálásával és a pixelenkénti megvilágítás kiszámításával kevésbé függ az adott objektum kidolgozottságától. A pixelenkénti megvilágítás kiszámítása bonyolultabb és költségesebb is, ezért korábban ezt a fajta módszert kevésbé használták a grafikus hardverekben.

A Gouraud árnyalással lényegében a Phong árnyaláshoz hasonló eredményt érhetünk el a felület pixelnél kisebb háromszögekre való felosztásával. Ez az algoritmus a gyakorlatban nagyon lassú lehet, de ez a módja annak, hogy a nem programozható grafikus hardveren megvalósított Gouraud árnyalás a Phong árnyalóhoz hasonlóan viselkedjen.

A vertexekben vagy az összes pixelben a megvilágítást a megvilágítási modell segítségével számítjuk ki. A valósidejű grafika esetén ezek a modellek nagyon hasonlóak és mindegyik három fő részre osztható, név szerint *diffúz*, *spekuláris* és *ambiens* komponensekre.

### 5.3.1. A diffúz komponens

A megvilágítási modell ezen része az egyetlen, amely valóban megfelel a fizikai valóságnak és fény és felület kölcsönhatásának.

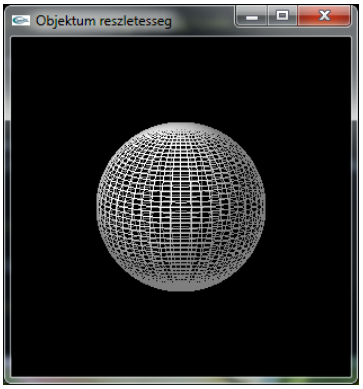
Ez azért van, mivel ez a Lambert törvényen alapul, amely azt mondja ki, hogy az ideális diffúz (teljesen matt és nem csillogó) felületeknél a visszavert fény mértéke az $\mathbf{n}$ felületi normál és az $\mathbf{l}$ fényvektor közötti $\phi$ szög koszinuszától függ (lásd 5.2. ábrát).

A fény vektor a felületi $\mathbf{p}$ pontból a fényforrás felé mutat. A normalizált $\mathbf{n}$ és $\mathbf{l}$ esetén a fény hatása a következő:

$$\mathbf{i}_{diff} = \mathbf{n} \cdot \mathbf{l} = \cos\phi, \tag{5.1}$$

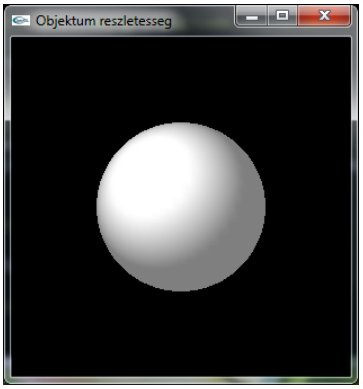
ahol $\mathbf{i}_{diff}$ a szem irányában visszavert fény mértékét megadó fizikai mennyiség. Ezt a diffúz megvilágítási komponensnek nevezzük. Megjegyezzük, hogy a diffúz megvilágítási


(a) Alacsony részletességű drótvázas gömb



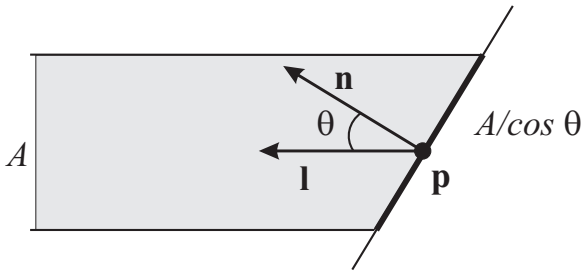

(b) Nagy részletességű drótvázas gömb


(c) Alacsony részletességű kitöltött gömb




(d) Nagy részletességű kitöltött gömb

5.1. ábra. Objektum részletességének hatása Gouraud árnyalás esetén




5.2. ábra. A sugárnyaláb által megvilágított terület nagysága arányosan növekszik az $\mathbf{n}$ felületi normál és az $\mathbf{l}$ fény vektor közötti $\phi$ szög nagyságával. A beeső fény egységnyi területre eső intenzitás mértéke viszont csökken a $\phi$ szög növekedésével.

komponens nullával egyenlő, amennyiben a $\phi > \pi/2$, vagyis a felület a fénnyel ellentétes irányba néz.

Amikor egy foton egy diffúz felülethez ér, akkor hirtelen elnyelődik a felületen. A fotonok és az anyag színétől függően a fotonok vagy teljesen elnyelődnek vagy továbbhaladnak. Azok az érvényes visszaverődési irányok, amelyek a normálvektorokkal $\pi/2$-nél kisebb szöget zárnak be (vagyis a felület felett vannak és nem mennek keresztül azon). A diffúz visszaverődés esetén minden új visszaverődési irány valószínűsége megegyezik. Ez azt jelenti, hogy a megvilágítási egyenlet diffúz komponense független a kamera pozíciójától és irányától. Más szavakkal, a diffúz komponens esetén a megvilágított felület bármely irányból ugyanúgy néz ki.

A fényforrás $\mathbf{s}_{diff}$ és az anyag $\mathbf{m}_{diff}$ diffúz színét használva, az 5.1. egyenletet újra definiálhatjuk:

$$\mathbf{i}_{diff} = \max((\mathbf{n} \cdot \mathbf{l}), 0)\mathbf{m}_{diff} \otimes \mathbf{s}_{diff}, \tag{5.2}$$

ahol $\mathbf{i}_{diff}$ a szín diffúz tagja és a $\otimes$ operátor a komponensenkénti szorzásnak felel meg. Továbbá a $\max((\mathbf{n} \cdot \mathbf{l}), 0)$ taggal azt is figyelembe vesszük, hogy a diffúz megvilágítás nulla, amennyiben az $\mathbf{n}$ és $\mathbf{l}$ közötti szög értéke nagyobb, mint $\pi/2$ radián.

### 5.3.2. A spekuláris komponens

Amíg a diffúz komponens a matt felületek viselkedését modellezi, a spekuláris komponens a felület csillogásáért felelős, ami világos foltként jelenik meg a felületen. Ez a terület a felület görbeségét hangsúlyozza ki, valamint segít a fényforrások irányának és helyének a meghatározásában. A grafikus hardverekben használt modellt a következő egyenlet írja le:

$$\mathbf{i}_{spec} = (\mathbf{r} \cdot \mathbf{v})^{m_{shi}} = (\cos\rho)^{m_{shi}}, \tag{5.3}$$

ahol $\mathbf{v}$ a $\mathbf{p}$ felületi pontból a nézőpont felé mutató vektor és az $\mathbf{r}$ pedig az $\mathbf{l}$ fény vektor $\mathbf{n}$ normálvektorral meghatározott visszaverődése. Ezt nevezzük *Phong megvilágítási egyenletnek* (nem összekeverendő a Phong megvilágítási modellel). A beeső fotonok az $\mathbf{r}$ visszaverődési irányba haladnak tovább. Az 5.3. egyenlet a gyakorlatban azt jelenti, hogy a spekuláris összetevő annál erősebb, minél jobban egybeesik az $\mathbf{r}$ visszaverődési vektor és a $\mathbf{v}$ nézőpont vektor.

Az $\mathbf{l}$ fény vektor az $\mathbf{n}$ normálvektorra nézve az $\mathbf{r}$ vektor irányában verődik vissza. Az $\mathbf{r}$ vektort a következőképpen lehet meghatározni:

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l}, \tag{5.4}$$

ahol feltesszük, hogy az $\mathbf{l}$ és $\mathbf{n}$ normalizáltak és ezért a $\mathbf{r}$ is normalizált. Amennyiben $\mathbf{n} \cdot \mathbf{l} < 0$, akkor a felület nem látható a fényforrásból nézve és a világos terület nem számítódik ki alap esetben. Míg a diffúz komponens nézőpont független, addig a spekuláris tag nézőpont függő. Az 5.3. egyenlet egy népszerű változatát Blinn mutatta be:

$$\mathbf{i}_{spec} = (\mathbf{n} \cdot \mathbf{h})^{m_{shi}} = (\cos\phi)^{m_{shi}}, \tag{5.5}$$

ahol $\mathbf{h}$ az $\mathbf{l}$ és $\mathbf{v}$ között lévő normalizált vektor:

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|}. \tag{5.6}$$

Az 5.5. egyenletre az az ésszerű magyarázat, hogy a $\mathbf{h}$ annak a síknak a normálisa a $\mathbf{p}$ pontban, amely a fényforrásból tökéletesen veri vissza a fényt a nézőpontba. Így az $\mathbf{n} \cdot \mathbf{h}$ tag akkor maximális, ha az $\mathbf{n}$ normális $\mathbf{p}$ pontban egybeesik a $\mathbf{h}$ vektorral. Az $\mathbf{n} \cdot \mathbf{h}$ tényező abban az esetben csökken, amikor az $\mathbf{n}$ és $\mathbf{h}$ között a szög növekszik. Az 5.5. egyenlet előnye az, hogy nem kell kiszámítani az $\mathbf{r}$ visszaverődési vektort.

A kétfajta spekuláris megvilágítás között a következő közelítést írhatjuk fel:

$$(\mathbf{r} \cdot \mathbf{v})^{m_{shi}} \approx (\mathbf{n} \cdot \mathbf{h})^{4m_{shi}}. \tag{5.7}$$

Hasonlóan a diffúz esethez, az anyagi tulajdonságokat és fényforrás paramétereket figyelembe véve a $\mathbf{p}$ pontban, az 5.5. egyenletet a $\mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$ szín vektorral, amely leírja azokat a fényforrásból érkező fotonokat, melyek spekulárisan verődnek vissza egy $\mathbf{m}_{spec}$ anyagi tulajdonságú felületről. Az OpenGL és a Direct3D ennek a formulának a megvalósításait használják:

$$\mathbf{i}_{spec} = \max((\mathbf{n} \cdot \mathbf{h}), 0)^{m_{shi}} \mathbf{m}_{spec} \otimes \mathbf{s}_{spec}, \tag{5.8}$$

ahol szintén figyelembe vesszük azt a tényt hogy, ha $(\mathbf{n} \cdot \mathbf{h}) < 0$ (vagyis a fény a felület alatt van), akkor a spekuláris tag nullával egyenlő. Az $m_{shi}$ a felület csillogásának a mértékét írja le. Az $m_{shi}$ értékének növelésével azt a hatást érjük el, hogy a világos terület nagysága beszűkül.

A Phong megvilágítási modell csak kis mértékben felel meg a fizikai valóságnak. Ezért érdemes másik spekuláris megvilágítási függvényt használni. Schlick adott egy alternatív megközelítést Phong egyenletére, amelyet alap esetben gyorsabban is lehet kiszámítani. Használva az 5.3. egyenlet jelöléseit a Schlick közelítése a következő:

$$\begin{aligned} t &= \cos\rho, \\ \mathbf{i}_{spec} &= \frac{t}{m_{shi} - t m_{shi} + t} \mathbf{m}_{spec} \otimes \mathbf{s}_{spec}. \end{aligned} \tag{5.9}$$

### 5.3.3. Az ambiens komponens

Az egyszerű megvilágítási modellünkben a fények közvetlenül ragyognak a felületeken, de semmi mást nem tesznek, míg a valóságban a fény a fényforrásból kiindulva egy másik felületről visszaverődve is elérheti a tárgyat. A másik felületről érkező fény nem számítható be sem a spekuláris sem pedig a diffúz komponensbe. Az indirekt megvilágítás szimulálására a megvilágítási modellbe belevesszük az ambiens tagot, amely általában csak valamilyen kombinációja az anyagi és fény konstansoknak:

$$\mathbf{i}_{amb} = \mathbf{m}_{amb} \otimes \mathbf{s}_{amb}. \tag{5.10}$$

Ennek a tagnak a modellhez való hozzáadása azt jelenti, hogy egy tárgy valamilyen minimális mennyiségű színnel fog rendelkezni, még akkor is, ha nem közvetlen módon lesz megvilágítva. Ily módon azok a felületek, melyek nem a fény felé néznek nem fognak teljesen feketén megjelenni. A legtöbb API támogat egy globális ambiens értéket. Az OpenGL továbbá támogatja a fényforrásonkénti anbiens értéket, így amikor a fényt kikapcsoljuk, akkor az ambiens összetevő automatikusan el lesz távolítva.

Csak az ambiens tagot használva azon felületek megvilágítására, melyek nem a fény felé fordulnak, azt tapasztaljuk, hogy az eredmény nem elfogadható. Ezeknél a területeknél ugyanaz a szín van megadva és így a három-dimenziós hatás eltűnik. Az egyik megoldás arra, hogy mindegyik objektum meg legyen világítva legalább egy kicsi direkt megvilágítással az, hogy fényeket helyezünk el a színtéren. A másik gyakori technika a *fejlámpa* (headlight) használta, amely egy a nézőponthoz kapcsolt pontfény. Ez a fejlámpa minden felület számára biztosítja a különböző fokú fényességet. Az előzőekben megadott megvilágításkor a spekuláris komponensét kikapcsoljuk, hogy kevésbé zavarjon.

### 5.3.4. A megvilágítási egyenlet

Ebben a fejezetben a teljes megvilágítási egyenletet rakjuk össze lépésről-lépésre. Ez az egyenlet meghatározza a képernyőn megjelenő pixelek színét.

Ez a megvilágítási modell egy lokális megvilágítási modell, amely azt jelenti, hogy a megvilágítás csak a fényforrásokból származó fénytől függ, vagyis más felületről nem érkezik fény. Az előző fejezetekből kiderült, hogy a megvilágítást az ambiens, diffúz és spekuláris komponensek határozzák meg. Valójában az $\mathbf{i}_{tot}$ teljes megvilágítási intenzitás ezeknek a komponenseknek az összege:

$$\mathbf{i}_{tot} = \mathbf{i}_{amb} + \mathbf{i}_{diff} + \mathbf{i}_{spec} \tag{5.11}$$

A valóságban a fény intenzitása fordítottan arányos a fényforrástól mért távolság négyzetével, melyet még nem vettünk figyelembe. Ez a csillapítás csak a pozícionális fényforrásokra igaz és csak a diffúz és spekuláris komponensekre van hatással. A következő formulát gyakran használják a csillapítás távolsággal való vezérlésére:

$$d = \frac{1}{S_c + s_l \left\|\mathbf{s}_{pos} - \mathbf{p}\right\| + s_q \left\|\mathbf{s}_{pos} - \mathbf{p}\right\|^2}, \tag{5.12}$$

ahol $\left\|\mathbf{s}_{pos} - \mathbf{p}\right\|$ az $\mathbf{s}_{pos}$ fényforrás pozíciójától vett távolság a $\mathbf{p}$ pontig, melyet árnyékolunk. $s_c$ a *konstans*, az $s_l$ a lineáris és a $s_q$ a kvadratikus csillapítást kontrollálják. A fizikailag korrekt távolság csillapításhoz az $s_c = 0$, $s_l = 0$ és $s_q = 1$ beállítást kell használni. Az 5.11. egyenletet a következőképpen kell módosítani ahhoz, hogy a távolsággal összefüggő csillapítást figyelembe vegyük:

$$\mathbf{i}_{tot} = \mathbf{i}_{amb} + d(\mathbf{i}_{diff} + \mathbf{i}_{spec}) \tag{5.13}$$

A reflektorfény a színteret különböző módon világítja meg. A $c_{spot}$-tal jelölt szorzótényező ezt a hatást írja le. Amennyiben a pont kívül esik a reflektor kúpján, akkor a $c_{spot} = 0$, ami

azt jelenti, hogy a fénysugarak a reflektorból nem érik el ezt a vertex-et. Ha a kúpon belül van a vertex, akkor a következő formulát használjuk:

$$c_{spot} = \max(-\mathbf{l} \cdot \mathbf{s}_{dir}, 0)^{s_{exp}}, \tag{5.14}$$

ahol $\mathbf{l}$ a fény vektor, $\mathbf{s}_{dir}$ a reflektor iránya és $s_{exp}$ az exponenciális faktor a reflektor középpontjától való halványodását vezérli. Mindegyik vektort normalizáltnak tesszük fel. Ha a fényforrásunk nem reflektorfény, akkor $c_{spot} = 1$. A módosított megvilágítási egyenlet a következőképpen alakul:

$$\mathbf{i}_{tot} = c_{spot}(\mathbf{i}_{amb} + d(\mathbf{i}_{diff} + \mathbf{i}_{spec})). \tag{5.15}$$

Az 5.2. fejezetben már említettük, hogy az anyag rendelkezik egy $\mathbf{m}_{emi}$ emisszív paraméterrel. Ez egy másik ad hoc paraméter, amely azt írja le, hogy a felület mennyi fényt bocsát ki. Megjegyezzük, hogy ez a fény kibocsátás nincs hatással a többi felületre. Ezzel lényegében egy homogén színt adunk a felülethez.

OpenGL-ben, Direct3D-ben és a többi API nagy részében szintén van egy $\mathbf{a}_{glob}$ globális ambiens fényforrás paraméter, amely egy konstans háttérfényt közelít, amely minden irányból körülveszi a tárgyakat. Ezeket a a paramétereket hozzávéve a megvilágítási egyenlethez, kapjuk a következőt:

$$\mathbf{i}_{tot} = \mathbf{a}_{glob} \otimes \mathbf{m}_{amb} + \mathbf{m}_{emi} + c_{spot}(\mathbf{i}_{amb} + d(\mathbf{i}_{diff} + \mathbf{i}_{spec})). \tag{5.16}$$

Ez az egyenlet az egy fényforrás esetére vonatkozó egyenlet. Tegyük fel, hogy $n$ fényforrásunk van és mindegyiket $k$ indexszel azonosítjuk. A megvilágítási egyenletet $n$ fényforrás esetén a következő módon írhatjuk fel:

$$\mathbf{i}_{tot} = \mathbf{a}_{glob} \otimes \mathbf{m}_{amb} + \mathbf{m}_{emi} + \sum_{k=1}^{n} c^k_{spot}(\mathbf{i}^k_{amb} + d^k(\mathbf{i}^k_{diff} + \mathbf{i}^k_{spec})). \tag{5.17}$$

A megvilágítási számítások annál tovább tartanak, minél több fényforrást használunk. Továbbá a fényforrás intenzitás összege 1-nél nagyobb is lehet, ezért az eredmény megvilágítási színt $[0, 1]$ intervallumra korlátozzuk le. Azonban ez a túlcsordulás utáni levágás a színekben eltolást eredményezhet. Ennek elkerülésére néhány rendszer a túlcsorduló színt skálázza a legnagyobb komponenssel. Léteznek sokkal bonyolultabb rendszerek, amelyek korlátozzák azt, hogy hány adott komponens (diffúz, spekuláris stb.) vehet részt a teljes szín kiszámításában. A túlcsordulások gyakran a geometriai részletességet csökkentik, mivel teljes felületen, ahol túlcsordulás volt, ugyanazt a színt kapjuk.

## 5.4. Átlátszóság

Az átlátszóság hatások a valósidejű megjelenítésű rendszerek esetében a végletekig leegyszerűsített és korlátozott. A hatások között például nem érhetőek el a fény elhajlása (fénytörés), a fény csillapodása az átlátszó objektumok vastagsága miatt és a nézeti szögnek köszönhető tükröződés és átviteli/transzmisszió változása.

Az átlátszóság megvalósításához szükség van az átlátszó tárgy színének és a mögötte lévő objektumok színének a keverésére. Amikor egy tárgyat a képernyőn megjelenítünk egy RGB szín és egy $Z$-puffer mélység van hozzákötve mindegyik pixelhez. Egy másik $\alpha$ komponens szintén létrehozható és opcionálisan tárolható. Az alfa az az érték, amely leírja a tárgy átlátszóságának a fokát egy adott pixelben. $\alpha = 1$ azt jelenti, hogy az objektum nem átlátszó és teljes egészében kitölti a pixel területet; $\alpha = 0$ pedig azt jelenti, hogy a pixel egyáltalán nem látszik.

Egy objektum átlátszóvá tételéhez a meglévő színtéren kell megjeleníteni egynél kisebb alfa értékkel. Minden olyan pixel esetén, amelyet az objektum „eltakar” RGB$\alpha$ (RGBA) értékek segítségével lesz megjelenítve. Ezt az értéket összekeverve az eredeti pixelértékkel az **over** operátort használva kapjuk az új pixel értéket:

$$\mathbf{c}_o = \alpha_s \mathbf{c}_s + (1 - \alpha_s)\mathbf{c}_d \qquad [\textbf{over} \text{ operátor}], \tag{5.18}$$

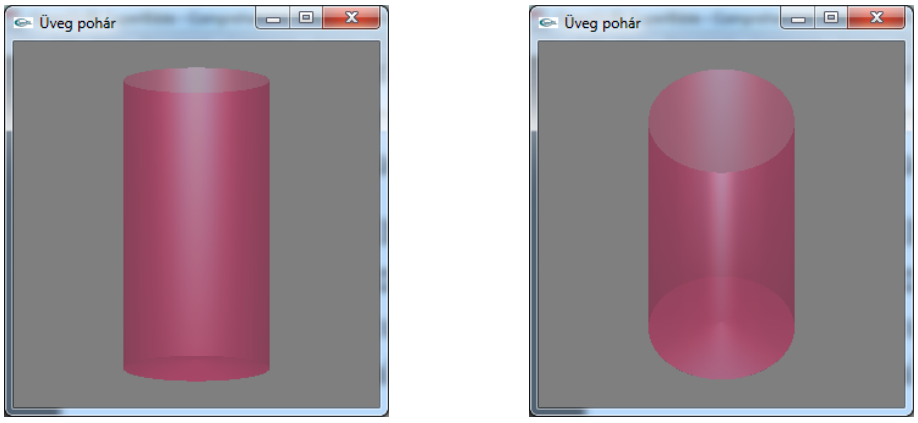
ahol $\mathbf{c}_s$ az átlátszó objektum színe (forrás), $\alpha_s$ a tárgy alfa értéke, $\mathbf{c}_d$ a keveredés előtti (a színpufferben lévő, cél) pixel szín érték. A $\mathbf{c}_o$ az eredmény szín oly módon, hogy az átlátszó objektumot a meglévő színtér **elé** (**over**) helyezzük. A renderelési csővezetékben a $\mathbf{c}_s$ és $\alpha_s$ elküldésével az eredeti $\mathbf{c}_d$ pixel lesz kicserélve $\mathbf{c}_o$ eredményével.

Átlátszó objektumok helyes megjelenítéséhez általában szükségünk van rendezésre. Először a nem átlátszó tárgyakat kell renderelni, aztán az átlátszó objektumokat kell hátulról előre haladva összekeverni a háttérben lévő alakzatok pixel értékeivel. Tetszőleges sorrendben való összekeverés esetén súlyos artifaktumokat kaphatunk, mivel ez a művelet sorrendfüggő vagyis feltételezi, hogy a háttérben lévő tárgyak már a színpufferben vannak. Speciális esetben, amikor két átlátszó tárgy van megjelenítve és mind a kettő alfa értéke 0.5, akkor a keveredésnél nem számít a sorrend. Amennyiben a rendezés nem lehetséges vagy csak részben lett végrehajtva, akkor a legjobb a $Z$-puffer használata, a $z$-mélység írását kikapcsolva az átlátszó objektum esetén. Ily módon az összes átlátszó objektum legalább meg fog jelenni. Más technikák esetén, például a hátsó oldalak eldobásának kikapcsolásával vagy az átlátszó poligonok kétszeri renderelésével és a mélység tesztelést valamint a $Z$-puffer írásának az engedélyezését váltogatva elérhetjük, hogy bizonyos esetekben működni fog, de általában üzembiztosan a rendezéssel működik az átlátszó objektumok helyes megjelenítése.

Az átlátszóságot ki lehet számítani több menetben, két vagy több mélységpuffer használatával. Az első megjelenítési menetben a nem átlátszó felületek $z$-mélység értékeit helyezzük el az első $Z$-pufferben. Ezután az átlátszó objektumokat rendereljük le. A második menetben a mélység tesztet úgy módosítjuk, hogy elfogadjuk azt a felületet, amely az első pufferben lévő $z$-mélység értéknél közelebb van és az átlátszó objektumok közül pedig a legtávolabb van. Így végrehajtva a renderelést a legtávolabbi átlátszó objektum bekerül a színpufferbe a mélység értéke pedig a második $Z$-pufferbe. Ezt a puffert aztán arra használjuk, hogy a következő legközelebbi átlátszó felületet határozzuk meg a következő menetben és így tovább. Jelen pillanatban egyetlen grafikus hardver sem támogatja ezt a dedikált második mélységpuffert, bár a pixel árnyalás (shading) hardveresen támogatott, amit fel lehet használni arra, hogy a $z$-mélység értékeket hasonló stílusban összehasonlítsuk és végrehajtsuk a *mélység hámozás*t (depth peeling). Ez a módszer működik, de meglehetősen lassú is, mivel sok lépésben a pufferek szerepeit felcserélve rajzolunk.

## 5.5. Egy példa megvilágításra és átlátszóságra

Ebben a fejezetben egy olyan példát mutatunk be, amely az előző két fejezetben ismertetett megvilágítással és átlátszósággal kapcsolatos OpenGL függvénykönyvtár lehetőségeit szemlélteti. A példa program az 5.3. ábrán látható üveg poharat modellezi, melynek külső és belső fala külön-külön van megvalósítva. A függvények teljes paraméterezését nem ismertetjük itt. A kódrészletekben az adott helyeken a függvény hívások magyarázatát a megjegyzésekben lehet megtalálni.




*(a) Oldalról* *(b) Felülről*

*5.3. ábra. Üvegpohár*

A poharat alkotó OpenGL primitívek normál egység vektorok meghatározásához az 5.1. kódrészletben használt `calcNormal` függvényt használjuk fel, amelyben először a bemenő 3 pont koordinátáiból meghatároz két vektort. A két vektor keresztszorzatával számítja ki a vektorok által kifeszített síkra merőleges vektort, amit azután egységre normál a `ReduceToUnit` függvény meghívásával.

```
1  // Adott vektor egységnyire való skálázása
2  void ReduceToUnit(float vector[3])
3  {
4    float length;
5
6    // A vektor hosszának a kiszámítása
7    length = (float)sqrt((vector[0]*vector[0]) +
8              (vector[1]*vector[1]) +
9              (vector[2]*vector[2]));
10
11   // Nullával való osztás elkerülése
12   if(length == 0.0f)
13     length = 1.0f;
14
```

```
15    // Elemek elosztása a hosszal
16    vector[0] /= length;
17    vector[1] /= length;
18    vector[2] /= length;
19  }
20
21  // Normál egység vektor kiszámítása három pontból
22  void calcNormal(float v[3][3], float out[3])
23  {
24    float v1[3],v2[3];
25    static const int x = 0;
26    static const int y = 1;
27    static const int z = 2;
28
29    // Két vektor meghatározása a három pontból
30    v1[x] = v[0][x] - v[1][x];
31    v1[y] = v[0][y] - v[1][y];
32    v1[z] = v[0][z] - v[1][z];
33
34    v2[x] = v[1][x] - v[2][x];
35    v2[y] = v[1][y] - v[2][y];
36    v2[z] = v[1][z] - v[2][z];
37
38    // A két vektor keresztszorzatának a kiszámítása
39    out[x] = v1[y]*v2[z] - v1[z]*v2[y];
40    out[y] = v1[z]*v2[x] - v1[x]*v2[z];
41    out[z] = v1[x]*v2[y] - v1[y]*v2[x];
42
43    // Az eredmény egységhosszúra való skálázása
44    ReduceToUnit(out);
45  }
```

*5.1. kódrészlet. Normál egység vektorok meghatározása 3 vertex alapján*

A pohár külső és belső oldalának a megvalósítása hasonlít a már korábban bemutatott hengerpalást megvalósításához. A pohár alja háromszög primitívekből van felépítve. A primitívek vertexeit/szögpontjait úgy állítjuk elő, hogy a `calcNormal` függvény bemenő paraméterének a típusával megegyezzen. A függvény hívásokban a `glNormal` függvény meghívásával állítjuk be a kiszámított normál egység vektorokat az adott primitívek esetén.

```
1   // A pohár külső oldalának a megvalósítása
2   void Uveg_kulso(int n, double radius1,
3                   double radius2, double height)
4   {
5     int i;
6     GLfloat angle;
7     GLfloat x1, x2;
8     GLfloat y1, y2;
```

```
9
10    float normal[3];
11    float vector[3][3];
12    GLfloat x1_next, y1_next;
13    GLfloat x2_next, y2_next;
14
15    if(n < 3)
16      n = 3;
17
18    glBegin(GL_QUADS);
19
20   // A hengerpalást alsó és felső koordinátáinak kiszámítása
21   // normálvektorok kiszámítása
22    for(i = 0, angle = 0.0; i < n;
23        i++, angle += 2.0 * GL_PI / n)
24    {
25      x1 = radius1*sin(angle);
26      y1 = radius1*cos(angle);
27
28      x1_next = radius1*sin(angle + 2.0 * GL_PI / n);
29      y1_next = radius1*cos(angle + 2.0 * GL_PI / n);
30
31      x2 = radius2*sin(angle);
32      y2 = radius2*cos(angle);
33
34    x2_next = radius2*sin(angle + 2.0 * GL_PI / n);
35    y2_next = radius2*cos(angle + 2.0 * GL_PI / n);
36
37
38    vector[0][0] = x1;
39    vector[0][1] = y1;
40    vector[0][2] = -height;
41
42    vector[1][0] = x2;
43    vector[1][1] = y2;
44    vector[1][2] = 0;
45
46    vector[2][0] = x2_next;
47    vector[2][1] = y2_next;
48    vector[2][2] = 0.0;
49
50  //Normálvektorok kiszámítása és beállítása
51    calcNormal(vector, normal);
52    glNormal3fv(normal);
53
54  // Vertexek elhelyezése
55    glVertex3f(x1, y1, -height);
```

```
56    glVertex3f(x2, y2, 0.0);
57    glVertex3f(x2_next, y2_next, 0.0);
58    glVertex3f(x1_next, y1_next, -height);
59    }
60    glEnd();
61
62  //Pohár aljának a meghatározása
63    glBegin(GL_TRIANGLES);
64
65      for(i = 0, angle = 0.0; i < n;
66          i++, angle += 2.0 * GL_PI / n)
67    {
68      x1 = radius1*sin(angle);
69      y1 = radius1*cos(angle);
70
71      x1_next = radius1*sin(angle + 2.0 * GL_PI / n);
72      y1_next = radius1*cos(angle + 2.0 * GL_PI / n);
73
74      vector[0][0] = x1_next;
75      vector[0][1] = y1_next;
76      vector[0][2] = 0.0f;
77
78      vector[1][0] = x1;
79      vector[1][1] = y1;
80      vector[1][2] = 0.0f;
81
82      vector[2][0] = 0.0;
83      vector[2][1] = 0.0;
84      vector[2][2] = -5.0f;
85
86  //Normálvektorok kiszámítása és beállítása
87      calcNormal(vector, normal);
88      glNormal3fv(normal);
89
90      glVertex3f(x1_next, y1_next, 0.0f);
91      glVertex3f(x1, y1, 0.0f);
92      glVertex3f(0.0f, 0.0f, -5.0f);
93    }
94    glEnd();
95  }
```

*5.2. kódrészlet. Az üvegpohár külső falának a megvalósítása*

```
1   // A pohár belső oldalának a megvalósítása
2   void Uveg_belso(int n, double radius1,
3                   double radius2, double height)
4   {
5     int i;
```

```
6     GLfloat angle;
7     GLfloat x1, x2;
8     GLfloat y1, y2;
9
10    float normal[3];
11    float vector[3][3];
12    GLfloat x1_next, y1_next;
13    GLfloat x2_next, y2_next;
14
15    if(n < 3)
16      n = 3;
17
18    glBegin(GL_QUADS);
19   // A hengerpalást alsó és felső koordinátáinak kiszámítása
20   // normálvektorok kiszámítása
21    for(i = 0, angle = 0.0; i < n;
22        i++, angle += 2.0 * GL_PI / n)
23    {
24      x1 = radius1*sin(angle);
25      y1 = radius1*cos(angle);
26
27      x1_next = radius1*sin(angle + 2.0 * GL_PI / n);
28      y1_next = radius1*cos(angle + 2.0 * GL_PI / n);
29
30      x2 = radius2*sin(angle);
31      y2 = radius2*cos(angle);
32
33    x2_next = radius2*sin(angle + 2.0 * GL_PI / n);
34    y2_next = radius2*cos(angle + 2.0 * GL_PI / n);
35
36    vector[0][0] = x1;
37    vector[0][1] = y1;
38    vector[0][2] = -height;
39
40    vector[1][0] = x2;
41    vector[1][1] = y2;
42    vector[1][2] = 0;
43
44    vector[2][0] = x2_next;
45    vector[2][1] = y2_next;
46    vector[2][2] = 0.0;
47
48  //Normálvektorok kiszámitása és beállítása
49    calcNormal(vector, normal);
50
51  //Mivel megegyezik a belső oldal vertexeinek a megadása
52  //a külső oldaléval, ezért a függvény hívása előtt a
```

```
53 //Renderscene fv.-ben glFrontface fv-nyel beállítjuk a
54 //helyes körbejárást  és a normál egységvektor irányát is
55 //korrigáljuk.
56   normal[0] = -1*normal[0];
57   normal[1] = -1*normal[1];
58   normal[2] = -1*normal[2];
59   glNormal3fv(normal);
60
61 //Vertexek elhelyezése
62   glVertex3f(x1, y1, -height);
63   glVertex3f(x2, y2, 0.0);
64   glVertex3f(x2_next, y2_next, 0.0);
65   glVertex3f(x1_next, y1_next, -height);
66   }
67   glEnd();
68
69 //Pohár alja
70   glBegin(GL_TRIANGLES);
71
72     for(i = 0, angle = 0.0; i < n;
73         i++, angle += 2.0 * GL_PI / n)
74   {
75
76     x1 = radius1*sin(angle);
77     y1 = radius1*cos(angle);
78
79     x1_next = radius1*sin(angle + 2.0 * GL_PI / n);
80     y1_next = radius1*cos(angle + 2.0 * GL_PI / n);
81
82     vector[0][0] = x1_next;
83     vector[0][1] = y1_next;
84     vector[0][2] = 0.0f;
85
86     vector[1][0] = x1;
87     vector[1][1] = y1;
88     vector[1][2] = 0.0f;
89
90     vector[2][0] = 0.0;
91     vector[2][1] = 0.0;
92     vector[2][2] = -5.0f;
93
94 //Hasonló módon járunk el mint a pohár oldalánál
95     calcNormal(vector, normal);
96
97     normal[0] = -1*normal[0];
98     normal[1] = -1*normal[1];
99     normal[2] = -1*normal[2];
```

```
100       glNormal3fv(normal);
101
102       glVertex3f(x1_next, y1_next, 0.0f);
103       glVertex3f(x1, y1, 0.0f);
104       glVertex3f(0.0f, 0.0f, -5.0f);
105
106    }
107    glEnd();
108 }
```

*5.3. kódrészlet. Az üvegpohár belső falának a megvalósítása*

OpenGL-ben az átlátszó tárgyakat szintén színkeveréssel állítjuk elő, ahol a forrás szín alfa komponensét használjuk fel az átlátszóság mértékének a beállítására. Ügyelnünk kell arra, hogy a $Z$-puffer használata nem megfelelő látványt állíthat elő, ezért ebben a példában csak olvasást engedélyezünk az átlátszó objektumok renderelésekor. A `glBlendFunc` függvény hívással álltjuk be a megfelelő színkeveredést és természetesen a megfelelő hatás eléréséhez szükség van a keveredés engedélyezésére. Az átlátszó objektumok renderelését az 5.4. kódrészlet mutatja be.

```
1  /* Modellezés */
2  void RenderScene( void )
3  {
4    /* Szín- és mélység-puffer törlése */
5    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
6
7    glEnable( GL_DEPTH_TEST );
8    glFrontFace(GL_CCW);
9    glShadeModel( GL_SMOOTH );
10
11   /* Mátrix állapot mentése és forgatás a tengelyek körül */
12   glPushMatrix();
13   glRotatef( xRot, 1.0f, 0.0f, 0.0f );
14   glRotatef( yRot, 0.0f, 1.0f, 0.0f );
15
16 //Átlátszó objektum beállítások
17   glEnable(GL_BLEND);
18   glEnable(GL_CULL_FACE);
19 //A mélységpuffer csak olvasható
20   glDepthMask(GL_FALSE);
21 //Színkeveredés paramétereinek a beállítása a forrás és cél
22 // színértékekre.
23   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
24   glColor4f(0.7, 0.1, 0.3, 0.35);
25
26
27   glPushMatrix();
28   /* Pohár modellezése */
```

```
29    glRotatef( 90.0f, 1.0f, 0.0f, 0.0f );
30    glTranslatef(0.0f, 0.0f, 80.0f);
31    glFrontFace(GL_CCW);
32    Uveg_kulso(800, 40.0f, 40.0f, 160.0f);
33
34    glFrontFace(GL_CW);
35    Uveg_belso(800, 39.9f, 39.9f, 160.0f);
36    glPopMatrix();
37
38  //A mélységpuffer írható és olvasható
39    glDepthMask(GL_TRUE);
40  //Színkeveredés letiltása
41    glDisable(GL_BLEND);
42
43    /* Elmentett transzformáció visszaállítása */
44    glPopMatrix();
45
46    /* Modellező parancsok végrehajtása */
47    glutSwapBuffers();
48  }
```

*5.4. kódrészlet. Átlátszó objektumok paraméter beállításai a `Renderscene` függvényben*

Az adott példában (lásd 5.5. kódrészletet) a megvilágítási beállításokat a `SetupRC` függvényben hajtjuk végre, amit a `main` függvényben a GLUT eseménykezelő ciklusba lépése előtt hívunk meg. Ez azt jelenti, hogy a fény beállítások statikusak ebben az esetben, nem változik időben. A függvényben definiálva vannak a megfelelő fényforrásokra és anyagokra érvényes paraméterek, amelyeket a megfelelő függvényekkel állíthatunk be. A `glLightModelfv` függvénnyel egy globális ambiens komponenst lehet megadni. A `glLight` függvény meghívásával az adott fényforrásokra (melyek azonosítói `GL_LIGHT0` ... (`GL_MAX_LIGHTS` - 1) lehetnek) következő paramétereket lehet beállítani: `GL_SPOT_EXPONENT`, `GL_SPOT_CUTOFF`, `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION` és `GL_QUADRATIC_ATTENUATION`. Ezek a paraméterek a reflektor fény és a fény hígulásával kapcsolatosak.

Az anyagi tulajdonságokat a `glMaterialfv` függvénnyel lehet beállítani. A mi esetünkben a fény és anyag között létrejövő kölcsönhatást a `glColor` függvénnyel adjuk meg, azonban ehhez engedélyeznünk kell a színkövetést (color tracking). Engedélyeznünk kell a `GL_COLOR_MATERIAL`-t majd meg kell adnunk azt, hogy a poligonok melyik oldalán és milyen komponensekre legyenek érvényesek a `glColor` beállításai. Végezetül a spekuláris anyagi tulajdonságokat állítjuk be a `SetupRC` függvény végén.

```
1
2   /* Kezdeti megjelenítési beállítások */
3   void SetupRC()
4   {
5   //Ambiens fény komponens
6     GLfloat ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
7   //Diffúz fény komponens
```

```
8     GLfloat diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
9   //Spekuláris fény komponens
10    GLfloat specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};
11  //Spekuláris anyagi tulajdonság
12    GLfloat specref[] =  { 1.0f, 1.0f, 1.0f, 1.0f };
13  //Fényforrás pozíciója
14    GLfloat lightPos[] = { 0.0f, 0.0f, 75.0f, 1.0f };
15
16    /* Szürke háttérszín */
17    glClearColor( 0.5f, 0.5f, 0.5f, 1.0f );
18
19  // Megvilágítás engedélyezése
20    glEnable(GL_LIGHTING);
21  //Globális ambiens megvilágítás beállítása
22    glLightModelfv( GL_LIGHT_MODEL_AMBIENT, ambientLight );
23  // 0-ás sorszámú fényforrás beállítása
24  // ambiens komponensének beállítása
25    glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
26  // 0-ás sorszámú fényforrás beállítása
27  // diffúz komponensének beállítása
28    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
29  // 0-ás sorszámú fényforrás, spekuláris komponensének beállítása
30    glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
31  // 0-ás sorszámú fényforrás, pozíciójának beállítása
32    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
33
34    glEnable(GL_LIGHT0);
35
36  // Color tracking engedélyezése
37    glEnable(GL_COLOR_MATERIAL);
38
39  // Anyagi tulajdonságok beállítása a glColor függvény segítségével
40    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
41
42  // Spekuláris anyagi tulajdonságok beállítása
43  // magas csillogás paraméterrel
44    glMaterialfv(GL_FRONT, GL_SPECULAR, specref);
45    glMateriali(GL_FRONT,GL_SHININESS, 20);
46
47    glColor3f(0.8f, 0.8f, 0.8f);
48
49   }
```

*5.5. kódrészlet. Megvilágítási paraméterek beállításai a `SetupRC` függvényben*

## 5.6. Köd

A valósidejű számítógépes grafikában a *köd* egy egyszerű atmoszferikus hatás, amelyet hozzá lehet adni a végső képhez. A ködöt több céllal is lehet használni:

- A külső tér realisztikusabb megjelenítés szintjének a növelése.
- Mivel a köd hatása a nézőponttól távolodva növekszik, ezért ez segít meghatározni, hogy milyen távol találhatóak az objektumok.
- Ha megfelelően használjuk, akkor ez segít a távoli vágósík hatásának az elrejtésében. Ha a köd paramétereit úgy állítottuk be, hogy azok az objektumok, amelyek a távoli sík közelében helyezkednek el és a köd vastagsága miatt nem láthatóak, akkor azok az objektumok, amelyek a nézeti csonka gúlán kívülre kerülnek a távoli síkon keresztül, úgy fognak tűnni, mintha a ködben tűnnének el. A köd nélkül az objektum a kamera mozgatásával a vágósík mögé kerülő része eltűnne, közeledve pedig feltűnne.
- A köd gyakran hardveresen van megvalósítva, így egy elhanyagolható plusz költséggel lehet azt használni.

Jelölje a köd színét $\mathbf{c}_f$, amit a felhasználó állít be, továbbá jelölje a köd együtthatóját $f \in \lfloor 0, 1 \rfloor$, mely csökken a nézőponttól távolodva. Tegyük fel, hogy az árnyalt oldal színe $\mathbf{c}_s$, ekkor a $\mathbf{c}_p$ végső pixel szín értékét a következőképpen határozhatjuk meg:

$$\mathbf{c}_p = f\mathbf{c}_s + (1 - f)\mathbf{c}_f. \tag{5.19}$$

$f$ definíciója nem-intuitív ebben a megadási módban: ahogy $f$ értéke csökken, a köd hatása növekszik. A következőkben az OpenGL-ben és DirectX-ben használatos megadási módot mutatjuk be. A fő előnye ezeknek az, hogy különböző egyenleteket használhatunk a $f$ megadására.

A *lineáris köd* egy köd konstans, ami lineárisan csökken a nézőponttól távolodva. Ezért két felhasználó által megadott értéket használunk annak a megadására, hogy hol kezdődik és hol végződik a köd a néző $z$-tengelye mentén. Ha $z_p$ az a $z$ érték, ahol a köd hatását kell meghatározni, akkor a lineáris köd együtthatót a következőképpen lehet meghatározni:

$$f = \frac{z_{end} - z_p}{z_{end} - z_{start}}. \tag{5.20}$$

Az exponenciális köd és négyzetes exponenciális köd esetén az $f$ ködegyütthatót a következőképpen lehet meghatározni:

$$f = e^{-d_f z_p}, \tag{5.21}$$

$$f = e^{(-d_f z_p)^2}, \tag{5.22}$$

$d_f$ paraméter a köd sűrűségét vezérli. Miután a köd együtthatóját kiszámítottuk, a kapott értéket a $[0, 1]$ intervallumra csonkoljuk és az 5.19. egyenletet alkalmazzuk a köd végső értékének a kiszámításához.

Néha táblázatokat használnak a ködfüggvény hardveres megvalósítása esetén. Minden mélységre egy $f$ ködegyüttható előre kiszámítanak és eltárolnak. Amikor a ködegyüttható egy adott mélység esetén kell meghatározni, akkor kiolvassák a táblázatból (vagy lineáris interpolációval határozzák meg két szomszédos tábla elemből). Bármilyen értéket el lehet helyezni a köd táblázatban, nem csak az iménti egyenletekben megadottakat. Ez teszi lehetővé számunkra azt, hogy érdekes megjelenítési stílusokat használjunk, amelyekben a ködhatás egy meghatározott módon változhat.

A ködfüggvényeket alkalmazni lehet vertex vagy pixel szinten. A vertex szinten való alkalmazása azt jelenti, hogy a köd hatása a megvilágítási egyenlet részeként lesz kiszámítva és a kiszámított szín értéket interpolálja a poligonon keresztül Gouraud árnyalást használva. A pixel-szintű ködöt a pixelenként tárolt mélység értéket használva számítjuk ki. Minden más együttható megegyezik. A pixel-szintű köd jobb eredményt ad.

Az 5.6. OpenGL kódrészlet egy tipikus ködbeállítást mutat be. A ködszámítások engedélyezése után a lineáris ködegyenlethez szükséges beállításokat hajtjuk végre. A `GL_LINEAR` köd egyenlet mellett a `GL_EXP` és `GL_EXP2` egyenleteket is ki lehet választani. Az exponenciális köd egyenleteknél a $d$ sűrűség paramétert a `glFogf(GL_FOG_DENSITY, d)` függvény hívással lehet beállítani.

```
1  //Köd bekapcsolása
2  glEnable(GL_FOG);
3  // A köd színének a beállítása
4  glFogfv(GL_FOG_COLOR, grey);
5  // Milyen messze kezdődik a köd hatása
6  glFogf(GL_FOG_START, 5.0f);
7  // Milyen messze végződik a köd hatása
8  glFogf(GL_FOG_END, 30.0f);
9  // Melyik köd egyenletet használja
10 glFogi(GL_FOG_MODE, GL_LINEAR);
```

*5.6. kódrészlet. A köd beállítása*

A szem síkja és egy fragmens közötti távolságot, ahogy azt korábban láttuk többféleképpen lehet kiszámítani. Néhány megvalósítás esetén (elsősorban NVIDIA hardver) az aktuális fragmens mélységet fogja használni. Más megvalósításkor a vertex-ek távolságát használja és a vertexek közötti értékeket interpolációval határozza meg. Az előzőekben említett megvalósításokat a `glHint` függvénnyel explicit módon lehet a fragmens köd kiszámítására használni (`glHint(GL_FOG_HINT, GL_NICEST)`). Természetesen ez szebb eredményt ad, viszont több számítást igényel. A gyorsabb, kevésbé számítás igényes ködhatás eléréshez a `glHint(GL_FOG_HINT, GL_FASTEST)` paraméterekkel kell a `glHint` függvényt meghívni.

A köd távolságot mi is kiszámíthatjuk és beállíthatjuk manuálisan a `void glFogCoordf(Glfloat fFogDistance)` függvény meghívásával. Ahhoz, hogy a beállított értékeket használja az OpenGL meg kell hívnunk a `glFogi(GL_FOG_COORD_SRC, GL_FOG_COORD)` függvényt. Az OpenGL által meghatározott köd értékek használatának visszatéréséhez a `glFogi(GL_FOG_COORD_SRC, GL_FRAGMENT_DEPTH)` függvényt kell meghívni.

# 6. fejezet

# Textúrázás

A számítógépes grafikában a textúrázás egy olyan eljárás, amely egy felület megjelenését módosítja egy bizonyos kép, függvény vagy adat segítségével. Egy példa lehet erre az, amikor egy téglafal pontos geometriai megvalósítása helyett egy téglafal színes képét illesztjük egy egyszerű poligonra. Amikor a poligont nézzük, akkor a színes kép fog megjelenni a poligon helyén. Hacsak a szemlélő „nem megy közel” a falhoz a geometriai kidolgozottság hiányát nem fogja észrevenni. A megjelenítéskor a képek és felületek ilyen jellegű kombinálásával gyorsulást lehet elérni.

Néhány textúrázott téglafal megjelenése azonban a geometriai hiányosságoktól eltekintve eltérhet a valóságtól. Például a tégla felszíne síknak tűnik, pedig alapesetben egyenetlen. A bump mapping technika alkalmazásával a tégla felületi normálvektorai megváltoztathatóak oly módon, hogy a megjelenítéskor a felület nem lesz tökéletesen sima. Ezek a felületi normálvektorok szintén eltárolhatóak egy textúraképben (lásd 9.2. fejezetet), amelyek az árnyalás kiszámításánál játszanak fontos szerepet.

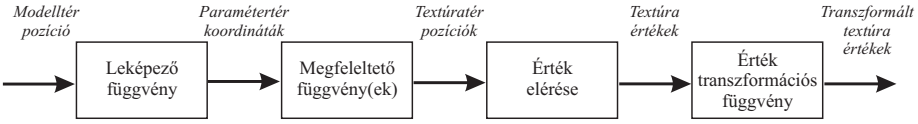
Ez csak két példa volt arra, hogy milyen problémák oldhatóak meg textúrázással. Ebben a fejezetben az alap textúrázási technikák mellett további lehetőségeket is bemutatunk.

## 6.1. Általánosított textúrázás

A textúrázás egy hatékony technika a felületi tulajdonságok modellezésére. Az egyik megközelítési módja a textúrázásnak az, amikor egy színértéket határozunk meg egy poligon vertexe esetén. Ahogy azt az előző fejezetben láttuk a színt a megvilágítási paraméterek valamint az anyagi tulajdonságok és a nézőpont helyének a figyelembe vételével számoljuk ki. Alapesetben az adott színértékeket a felületi pozíciók alapján módosítjuk. A téglafal példa esetén a felület összes pontjában lecseréljük a színt a téglafal textúrakép megfelelő színével. A felületi egyenetlenséget tartalmazó textúrakép esetén pedig a normálvektorok irányát változtatjuk meg adott pozíciókban.

A textúrázás leírható egy általánosított textúra csővezetékkel. A textúrázási eljárás során a felületi pontokhoz tartozó textúra értéket határozzuk meg. Így egy térbeli ponthoz kell megkeresnünk az ennek megfelelő textúratérbeli pozíciót. Ez a hely lehet a világtéren is, de leggyakrabban a modellhez van rögzítve, így a textúrakép követi a modell mozgását. Ezen a térbeli pozíción alkalmazunk egy *leképező függvényt* azért, hogy megkapjuk *paramétertér*

*értékeket*, amelyeket a textúrakép eléréshez fogunk használni. Ezt az eljárást *textúra leképezésnek* nevezzük. Mielőtt ezeket az új értékeket használhatnánk, még egy vagy több *megfeleltető függvényt* használhatunk a paramétertér értékeinek a textúraképtérbe való transzformálásához. Ezeket a textúraképtérbeli értékeket használjuk fel a textúrakép értékeinek eléréséhez, például ezek lehetnek tömb indexek a textúrakép pixeleinek a kinyeréséhez. A kinyert értéket ezután újból transzformálhatjuk az *érték transzformációs függvénnyel*, és végül ezeket az új értékeket használjuk a felület néhány tulajdonságának a módosítására. A 6.1. ábrán látható ez az eljárás részletesen egy textúrakép alkalmazására.




*6.1. ábra. Általánosított textúra csővezeték egy textúrára esetén*

### 6.1.1. A leképező függvény

Ahogy a bevezetésben láttuk, a textúrázáshoz szükségünk van egy textúraképre, amelyben színek vagy az árnyalás során használt optikai/felületi jellemzők vannak eltárolva. A textúraképek általában kétdimenziósak, ahol lebegőpontos $(u, v)$ $(u, v \in [0, 1])$ textúra-koordinátákkal adhatunk meg egy textúratérbeli pozíciót. A textúra-leképezés során ezeket a textúra-koordinátákat rendeljük hozzá a felületi pozíciókhoz, amelyek megadják, hogy egy pontban mivel kell lecserélni a színértéket illetve az adott pozícióban milyen értékkel kell módosítani az árnyalást. A gyakorlatban ezt a leképezést a felületet alkotó poligonok szögpontjaiként határozzuk meg úgy, hogy az adott szögponthoz hozzárendeljük a megfelelő textúra-koordinátákat. Így modellt alkotó primitívekre illesztjük a textúraképből kivágott területeket.

Valósidőben a leképezéseket általában a modellező szakaszban manuálisan adjuk meg és a leképezés eredményét a vertexekben tároljuk. Ez nem minden esetben van így, például az OpenGL `glTexGen` eljárás több különböző automatikus leképező függvényt biztosít, köztük gömbi és sík leképező függvényeket is[1].

Más bemenő adatokat is lehet használni egy leképező függvényben[2]. Bizonyos leképező függvények egyáltalán nem is hasonlítanak a vetítésre. Például a parametrikus görbe felületek definíciójuk alapján rendelkeznek egy $(u, v)$ értékhalmazzal. A textúra-koordinátákat például a nézőpont iránya vagy a felület hőmérséklete alapján is elő lehet állítani.

Nem-interaktív rendereléskor gyakran hívják meg ezeket a leképező függvényeket a renderelési eljárás részeként. Egy leképező függvény lehet, hogy elegendő a teljes modell

[1] A gömbi leképezés a pontokat egy pont körül elhelyezkedő képzeletbeli gömbre vetíti. A henger leképezés az $u$ textúra-koordinátákat a gömbi leképezésekhez hasonlóan számítja ki, a $v$ textúra-koordinátákat a henger tengelye mentén, mint távolságot számítja ki. Ez a leképezés hasznos olyan objektumok esetén, amikor az objektumnak van egy természetes tengelye. A sík leképezés egy irány mentén képez le és a textúrát a teljes felületre alkalmazza.

[2] Például a felületi normálvektort arra lehet használni, hogy kiválasszuk azt a síkot, amelyiket a sík leképezésekkor használunk az adott felületre cube map esetén (lásd 9.1.2. fejezetet).

számára, de gyakran a felhasználónak kell a modellt szétdarabolni és a leképező függvényeket alkalmazni.

### 6.1.2. Megfeleltető függvények

A megfeleltető függvények a paramétertérben lévő értékeket konvertálják textúratérbeli pozíciókra. A megfeleltető függvény megadható egy transzformációs mátrixszal is. Ez a transzformáció eltolhatja, forgathatja, skálázhatja, nyírhatja és ráadásul leképezheti/vetítheti a textúrát egy adott felületre.

Egy másik osztálya a megfeleltető függvényeknek az, amikor azt szabályozzuk, hogy milyen módon alkalmazzuk a textúrát. Tudjuk, hogy egy kép meg fog jelenni a felületen, ahol $(u, v)$-k a $[0, 1)$ intervallumban vannak. De mi történik ezen az intervallumon kívül? A megfeleltető függvény meghatározhatja ezt a viselkedést. Ilyen típusú megfeleltető függvények a következők lehetnek az OpenGL-ben[3]:

- **repeat** esetén (lásd 6.2.a. ábrát) a kép ismétli önmagát a felületen. Algoritmikusan a paraméter egész részét eldobjuk. Ez a függvény akkor hasznos, amikor egy anyag ismételve fedi be a felületet. Gyakran ez az alapbeállítás.

- **mirror** esetén (lásd 6.2.b. ábrát) a kép szintén ismétli önmagát a felületen, de minden egyes ismétléskor tükrözve van. Ez egyfajta folytonosságot kölcsönöz az adott élek mentén a textúrának.

- **clamp to edge** esetén (lásd 6.2.c. ábrát) a $[0, 1)$ intervallumon kívüli értékek esetén a textúrakép első és utolsó sorának vagy oszlopának az ismétlését eredményezi.

- **clamp to border** esetén (lásd 6.2.d. ábrát) a $[0, 1)$ paraméter értékeken kívül a textúra betöltéskor megadott[4] határ színét használja hasonlóan a **clamp to edge** esethez. A határ nem tartozik textúrához.

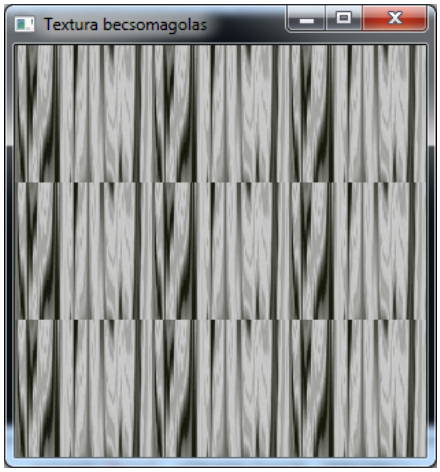
Ezeket a megfeleltető függvényeket mindegyik textúrához különféleképpen lehet hozzárendelni. Például a textúrát ismételhetjük az $u$ tengely mentén és tükrözhetjük a $v$ tengelyen.

A valósidőben alkalmazott utolsó megfeleltető függvény implicit és a kép méretéből van származtatva. Egy textúra $u$ és $v$ értékei alapesetben a $[0, 1)$ intervallumon belül vannak. Az ebben az intervallumban lévő paraméterértékeket megszorozva az adott kép méretével a pixelek pozícióját kapjuk meg a képtérben. Ennek az az előnye, hogy az $(u, v)$ értékek nem függenek a kép méretétől, például a $(0.5, 0.5)$ textúra-koordináta mindig a textúra középpontját adja meg.

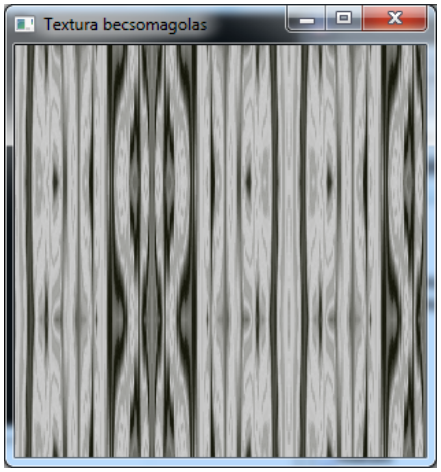
A megfeleltető függvények a paramétertér értékeit használják a textúratér pozícióinak az előállítására. A textúraképek esetén a textúratér pozíciók segítségével nyerjük ki a textúra értékeket a textúraképből.

[3] Ezt a fajta megfeleltető függvényt OpenGL-ben „becsomagolási mód”-nak (angolul wrapping mode) nevezik.

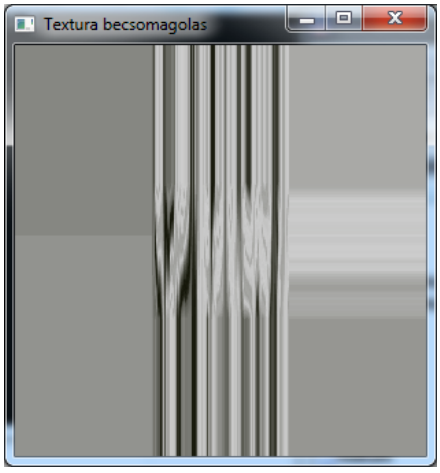
[4] A határszínét a `glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor)` OpenGL függvényhívásával lehet megadni, ahol a `borderColor` RGBA színértéket tartalmaz.



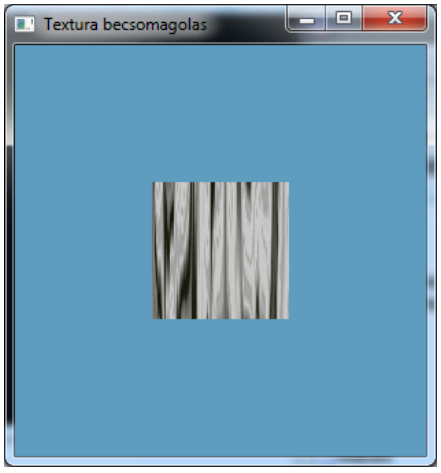

(a) Repeat




(b) Mirror




(c) Clamp to edge




(d) Clamp to border halványkék határszín esetén

6.2. ábra. Textúra becsomagolási módok

A valósidejű grafikai alkalmazások esetén leggyakrabban kétdimenziós képeket használunk, de léteznek más textúra függvények is. Egy tetszőleges háromdimenziós felület textúrázása csak kétdimenziós képpel gyakran nehéz vagy lehetetlen feladat. Az egyik megoldás az, hogy textúra darabokat állítunk elő, ami beborítja az adott felületet. Ezeknek a műveleteknek a végrehajtása összetett felületeken technikailag igen bonyolult.

Egy direkt kiterjesztése a textúraképeknek a háromdimenziós képadat, amit $(u, v, w)$ koordinátákon keresztül érhetünk el[5] a $w$ mélységgel együtt. A háromdimenziós textúra előnye az, hogy az adott modell csúcspontjaihoz közvetlenül a felületi pozíciókat rendelhetjük textúra-koordinátaként és így elkerülhetjük a kétdimenziós textúrázáskor előforduló torzításokat. Ebben az esetben a textúrázás úgy hat, mintha az anyagot reprezentáló háromdimenziós textúrából lenne kifaragva a modell. A háromdimenziós textúraképre jó példa egy orvosi CT-s képsorozat.

### 6.1.3. Textúra értékek

A legegyszerűbb adat, amelyet egy textúra érték kinyeréskor kaphatunk az egy RGB hármas, amelyet a felületi szín kicserélésére vagy módosítására használhatunk fel. Hasonlóan egy szürkeárnyalatos értéket ($[0 - 255]$) is tárolhat a textúrakép. Egy másik típusú adat az RGB$\alpha$. Az $\alpha$ érték alapesetben a szín átlátszóságát fejezi ki, amely befolyásolhatja a pixel végső színértékét. Természetesen léteznek másfajta adattípusok is, mint például a felületi egyenetlenség leképezés esetén, ahol felületi normálvektorokat kapunk vissza.

Miután a textúra értékeket kinyerjük azokat vagy közvetlenül vagy transzformálva használhatjuk fel. Az így kapott értékeket felületi tulajdonságok módosítására alkalmazhatunk. Emlékezzünk arra, hogy a legtöbb valósidejű rendszer esetén Gouraud árnyalást használnak, ami azt jelenti, hogy csak bizonyos értékek vannak interpolálva a felületen. Így csak ezeket az értékeket tudja a textúra módosítani. Alapesetben a megvilágítási egyenlet RGB eredményét módosítjuk, mivel ezt az egyenletet értékeljük ki minden vertexnél és a színt ezután interpoláljuk.

A legtöbb valósidejű rendszer megadhatunk egy módszert a felület színértékének módosítására. Ezeket a módszereket *egyesítő függvényeknek* vagy *textúra keveredés operátoroknak* nevezzük. Egy kép egy felületre való illesztése a következőket foglalja magába:

- **replace** - Egyszerűen az eredeti felületi színt lecseréli a textúra színére. Megjegyezzük, hogy ez eltávolítja az árnyalás során meghatározott értéket.
- **decal** - Hasonló a **replace** egyesítő függvényhez, de amikor a textúrakép tartalmaz egy $\alpha$ értéket, akkor a textüraszín az eredeti felületi színnel, de az eredeti $\alpha$ érték nem módosul.
- **modulate** - Megszorozza a felületi színértékét a textúra színével. Az árnyékolt felület a textúra színével van módosítva, amely egy árnyékolt textúrázott felületet ad.

Ezek a leggyakoribb módszerek egyszerű színes textúra leképezésekre. Azt, amikor a **replace**-t egy megvilágított környezetben textúrázásra használunk, néha izzó textúra

[5] Más rendszerekben $(s, t, q)$ textúra-koordinátákat használnak.

használatnak nevezzük, hiszen a textúra színe mindig ugyanúgy néz ki tekintet nélkül az árnyalásra.

## 6.2. Textúraképek

Textúraképek használata esetén egy kétdimenziós kép ténylegesen egy poligon felületére van illesztve. Az előzőekben a poligon oldaláról tekintettük át az eljárást. Most a kép szemszögéből és annak a felületre való alkalmazását vizsgáljuk meg. A következőkben a textúraképet egyszerűen csak textúrának fogjuk nevezni. Ráadásul, amikor egy pixelre hivatkozunk, akkor egy képernyőrács cellát értünk majd rajta. Lényegében ebben az esetben a pixel egy megjelenített színérték a képernyőn.

A textúra kép mérete gyakran korlátozva van $2^m \times 2^n$ vagy néha $2^m \times 2^m$ méretre, ahol $m$ és $n$ nem negatív egészek. Tegyük fel, hogy van egy $256 \times 256$ pixel méretű képünk, amit egy négyzeten akarunk textúraként használni. Amennyiben a leképezett négyzet mérete nagyjából megegyezik a textúra méretével, a textúra a négyzeten majdnem ugyanúgy néz ki, mint az eredeti kép. De mi történik akkor, amikor a leképezett négyzet tízszer annyi fragmenst fed le, mint a textúrakép (*nagyítás*) vagy ha a leképezett négyzet csak a textúrakép pixeleinek a töredékét fedi le (*kicsinyítés*)? A válasz attól függ, hogy milyen mintavételező és szűrő módszereket használunk ezekben az esetekben.

### 6.2.1. Nagyítás

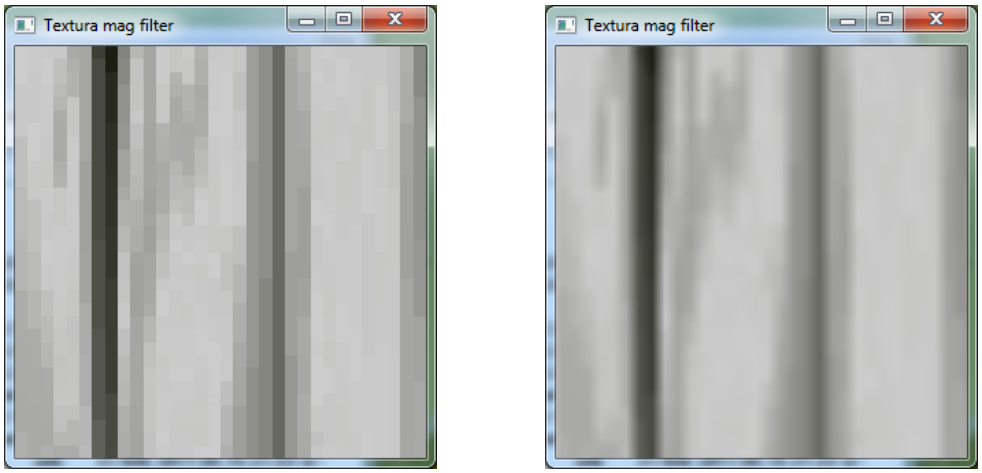
A leggyakrabban használt nagyítás szűrési technika a *legközelebbi szomszéd* (nearest neighbor) és a *bilineáris interpoláció* (néha *lineáris interpolációnak* is nevezik). A legközelebbi szomszéd technika egyik jellemzője az, hogy a texelek különállóan láthatóvá válnak (lásd 6.3.(a) ábrát). Ezt a hatást *pixelesedésnek* nevezzük és azért fordul elő, mert a módszer a nagyítás során a legközelebbi texelt veszi mindegyik pixel középpontjában, ami blokkosodást eredményez. A módszer minősége néha gyenge, viszont pixelenként csak egy texelt kell felhasználnunk.

A bilineáris interpoláció mindegyik pixel esetén négy szomszédos texelt vesz, majd kétdimenzióban lineárisan interpolálja azokat. Az eredmény homályosabb, viszont a legközelebbi szomszéd módszernél tapasztalt élek recésségének nagy része eltűnik (lásd 6.3.(b) ábrát). A bilineáris interpolált $\mathbf{b}$ szín a $(p_u, p_v)$ pozícióban (lásd 6.4. ábrát) a következőképpen számítható ki:

$$\begin{aligned}\mathbf{b}(p_u, p_v) =&(1-u')(1-v')\mathbf{t}(x_l, y_b) + u'(1-v')\mathbf{t}(x_r, y_b) + \\ &(1-u')v'\mathbf{t}(x_l, y_t) + u'v'\mathbf{t}(x_r, y_t),\end{aligned} \tag{6.1}$$

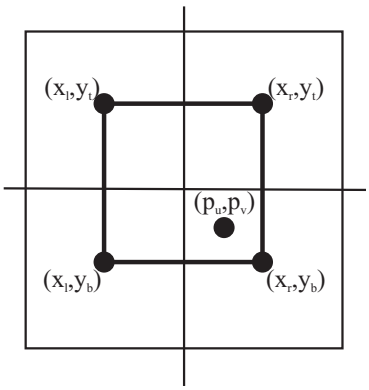
ahol a $\mathbf{t}(x, y)$ a texel színét jelöli a textúrában, $x$ és $y$ egészek.

A szűrőt az alapján választjuk ki, hogy milyen eredményt szeretnénk elérni. Általánosságban elmondható, hogy a bilineáris interpoláció használata az esetek többségében jó eredményt ad.




(a) Legközelebbi szomszéd (b) Bilineáris interpoláció

6.3. ábra. Textúrakép nagyítási technikák




6.4. ábra. Bilineáris interpoláció jelölései, ahol négy texelt veszünk figyelembe

### 6.2.2. Kicsinyítés

Amikor egy textúrát kicsinyítünk, akkor több texel fedhet be egy pixelt. A pixelek helyes értékeinek meghatározásához összegezni kell a textúra értékek hatását, amelyeket az adott pixel lefed. Azonban nehéz pontosan meghatározni a textúra értékek átlagát egy bizonyos pixel pozícióban, mivel nem tudjuk azt, hogy hány textúra értéket kell az átlagszámításkor figyelembe venni.

Ezen korlátok miatt számos különböző módszert használnak valósidőben. Az egyik ilyen módszer a legközelebbi szomszéd használata, ami ugyanúgy határozza meg a megfelelő értéket, mint a nagyítás esetén használt szűrő. Ez a szűrő komoly aliasing problémát okoz. Ha éles szögből nézünk rá egy síkfelületre feszített textúrára, akkor artifaktumok/műtermékek jelenek meg (lásd 6.5.(a) ábrát). A probléma abból ered, hogy csak egyetlen egy texel befolyásolja a pixel értékét adott pozícióban. Ezek az artifaktumok sokkal jobban észrevehetőek, amikor a nézőponthoz viszonyítva a felület mozog, amit *időbeli aliasingnak* nevezünk.

A bilineáris interpolációt szintén használhatunk kicsinyítéskor. Négy texel értékét átlagoljuk ebben az esetben is. Ha viszont egy pixel értékére több, mint négy textúra érték van hatással, akkor a szűrő hibázni fog.

A textúra jelfrekvenciája attól függ, hogy milyen közel helyezkednek el a texelek a képernyőn. A mintavételezési tétel[6] miatt meg kell bizonyosodnunk arról, hogy a textúra jelfrekvenciája nem nagyobb, mint a mintavételezési frekvencia fele. Ehhez azt kell tenni, hogy vagy a mintavételezési frekvenciát kell növelni vagy a textúra frekvenciáját kell csökkenteni. Antialiasing módszerekkel növelhetjük a mintavételezési frekvenciát. Azonban ezek a módszerek csak korlátozott mértékben tudják növelni a mintavételezés frekvenciáját.

A textúra antialiasing algoritmusok alapötlete az, hogy a textúrákat elő-feldolgozzuk és egy olyan adatstruktúrát hozunk létre, amely lehetővé teszi azt, hogy texelek pixelre gyakorolt hatásának a közelítését gyorsan számíthassuk ki. Valósidejű alkalmazásokban ezeknek az algoritmusoknak az a jellegzetessége, hogy egy rögzített mennyiségű időt és erőforrást használnak, ami azt eredményezi, hogy mindegyik textúra esetén egy fix számú mintát veszünk pixelenként.

#### Mipmapping

A legnépszerűbb antialiasing módszer textúrák esetén a *mipmapping*[7] (lásd 6.5.(b) ábrát). A legtöbb mipmap technika legegyszerűbb grafikus hardveren is meg van valósítva.

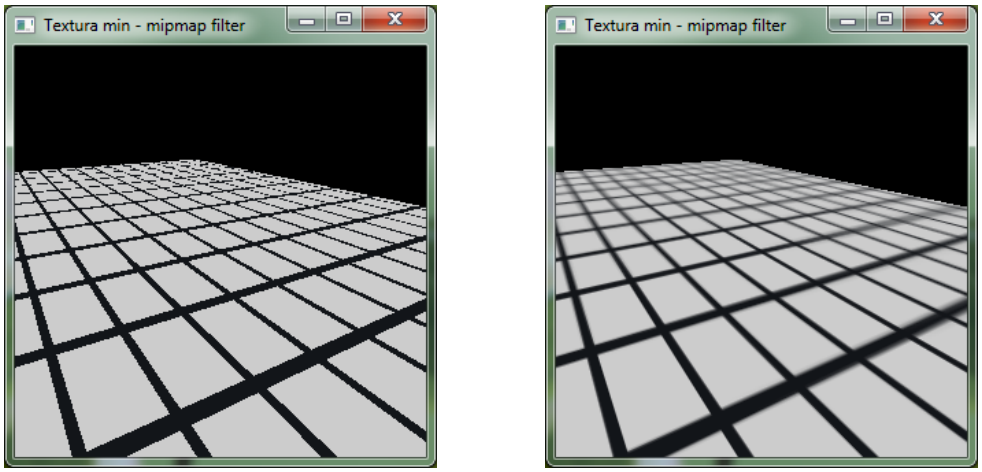
Amikor a mipmap kicsinyítő szűrőt használjuk, akkor az eredeti textúra mellett a textúra kicsinyített változatait is felhasználjuk. A kicsinyített textúrákat úgy állítjuk elő, hogy az eredeti textúrából kiindulva, mindig az előző textúra méretét csökkentjük a negyedére úgy, hogy minden új texelt négy szomszédos texel átlagaként számítunk ki. A csökkentést addig hajtjuk végre addig, amíg a textúrának egyik vagy mind a két dimenziója egy texellel lesz egyenlő. Az így kialakult textúra felbontási szintek mentén egy harmadik tengelyt definiálhatunk, amit $d$-vel jelölünk.

A jó minőségű mipmap textúra létrehozásához szükség van jó szűrésre és gamma korrekcióra. Amennyiben kihagyjuk a gamma korrekciót[8], akkor az átlagos fényessége a

[6] Nyquist-Shannon mintavételezési tétel

[7] A mip a latin *multum in parvo* rövidítése, aminek a jelentése „sok dolog kis helyen”.

[8] A gamma korrekció egy kontraszt értékeket optimalizáló eljárás, ami azt biztosítja, hogy a képek kontrasztja




*(a) Legközelebbi szomszéd* *(b) Mipmap*

*6.5. ábra. Textúrakép kicsinyítési technikák*

mipmap szinteken el fog térni az eredeti textúra fényességétől. Ahogy eltávolodunk az objektumtól és nem a korrigált mipmap textúrákat használjuk az objektum sötétebben fog megjelenni és a kontrasztja és a részletessége is megváltozhat.

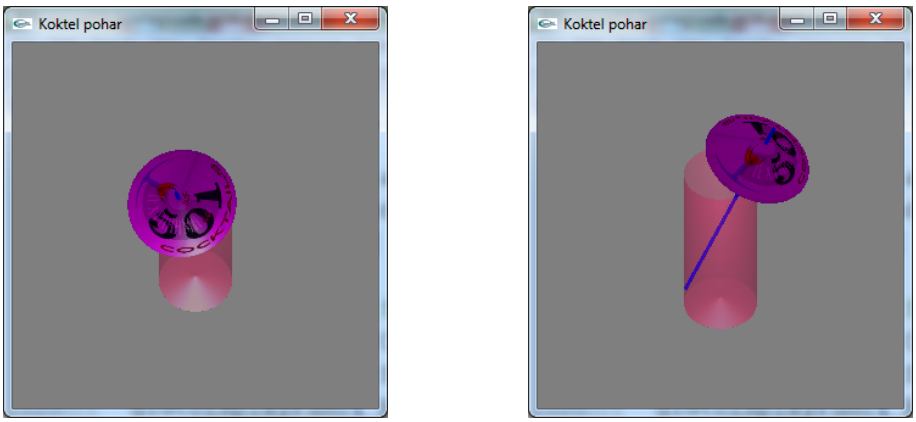
A $d$ koordináta kiszámításakor azt határozzuk meg, hogy hol mintavételezzünk a mipmap piramis tengely mentén. Azt szeretnénk elérni, hogy egy pixel-texel arány legalább $1:1$ legyen azért, hogy a Nyquist arányt elérjük. A cél az, hogy nagyjából meghatározzuk azt, hogy a pixelre mekkora textúra terület van hatással. Két fajta módszert használnak a $d$ kiszámítására (OpenGL-ben $\lambda$-ának nevezik és részletesség szintjének is nevezik.). Az egyik a pixel által formált négyszög hosszabb élét használja a pixel kiterjedésének a megközelítésére. A másik a legnagyobb abszolút értékű ($\frac{\partial u}{\partial x}$, $\frac{\partial v}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial v}{\partial y}$) differenciát használja mértékként. Mindegyik differencia azt határozza meg, hogy mekkora a változás nagysága a textúra-koordinátákban a képernyő adott tengelye mentén. Például a $\frac{\partial u}{\partial x}$ az $u$ érték változásának a nagyságát jelenti egy pixelre nézve az $x$ tengely mentén.

Amikor az $(u, v, d)$ hármast használjuk a mipmap textúra elérésére, akkor a $d$ értéke egy valós szám. Mivel $d$ nem egész, ezért a $d$ textúra szint felett és a szint alatt mintavételezünk. Az így kapott $(u, v)$ pozíciót használjuk egy-egy bilineárisan interpolált minta kinyerésére a két szomszédos textúra szintből. Az eredményt ezután lineáris interpolációval kapjuk meg attól függően, hogy $d$ milyen távolságra van a két szomszédos textúra szinttől. A teljes eljárást trilineáris interpolációnak nevezzük és pixelenként hajtjuk végre.

közel azonos lesz. Ezt a korrekciót korábban a képek CRT monitorokon való megjelenítésekor használták.

## 6.3. Egy OpenGL példa a textúrázásra

Ebben a fejezetben az 5.5. fejezetben található üvegpohár példát egészítjük ki egy textúrázott objektummal (lásd 6.6. ábrát) és ezen mutatjuk be az OpenGL alap textúrázás beállításainak a lehetőségeit.




*(a) Felülről* *(b) Oldalról*

*6.6. ábra. Üvegpohár textúrázott koktél ernyővel*

Nem mutatjuk be külön a `Szivoszal()` függvény törzsét (lásd 6.3. kódrészletet), ami lényegében a 4.1. fejezetben található hengerpalást kódjával egyezik meg (lásd 4.1. kódrészlet 31-54 sorait). Hasonlóképpen nem részletezzük a textúra kép betöltését elvégző `LoadDIBitmap()` (lásd 6.1. kódrészletet), ami egy megadott BMP típusú állomány pixel adatait valamint a képhez hozzátartozó információkat (méret, típus stb.) adja vissza.

A pixel adatok betöltése után egy globális textúra azonosítót foglalunk le a `glGenTextures(GLsizei n, GLuint *textures)` OpenGL függvény meghívásával. Ezt az azonosítót a textúra használatakor a `glBindTexture(GLenum target, GLuint texture)`[9] OpenGL függvény második paramétereként kell megadni. A `glBindTexture` függvényt az adott textúra használata előtt kell meghívni[10]. A függvény első paraméterében a textúra típusát adjuk meg. A függvény meghívásával a textúra és a hozzátartozó paraméter beállítások betöltődnek[11].

Ezek után a textúra nagyítási, kicsinyítési, csomagolási és környezeti paramétereit állítjuk be a `glTexParameteri(GLenum target, GLenum pname, GLint param)` és a `glTexEnvi(GLenum target, GLenum pname, GLint param)` OpenGL függvények segítségével.

[9] Ezzel a technikával egyszerre több textúrát is betölthetünk és gyorsan tudjuk váltogatni azokat szükség szerint.

[10] A `glBindTexture()` függvényt a textúra és a textúra paraméterek beállítása előtt is meg kell hívni.

[11] Amikor már nincs szükség az adott azonosítójú textúrákra, akkor a textúra objektum törléséhez a `glDeleteTextures(GLsizei n, GLuint *textures)` függvényt hívjuk meg.

A textúra megadásához a `glTexImage2D(GLenum target, GLint level, GLint internalformat, GLsizei width, GLsizei height, GLint border,GLenum format, GLenum type, void *data)` függvényt kell meghívni. A függvényben meg kell adni a textúra típusát, a textúra szintjét (amennyiben mipmap textúrázást használunk, akkor ez az érték 0-tól eltérő is lehet), valamint a textúraképpel kapcsolatos információkat.

```
1
2  int main( int argc, char* argv[] )
3  {
4
5      BITMAPINFO  *info;  /* Bitmap információ */
6      GLubyte *bits;      /* Bitmap RGB pixelek */
7
8  ...
9
10     bits = LoadDIBitmap("koktel.bmp", &info);
11     if (bits == (GLubyte *)0)
12         return (0);
13
14     glGenTextures(1, &texture1);
15     glBindTexture(GL_TEXTURE_2D, texture1);
16
17     glTexParameteri(GL_TEXTURE_2D,
18                     GL_TEXTURE_MAG_FILTER, GL_LINEAR);
19     glTexParameteri(GL_TEXTURE_2D,
20                     GL_TEXTURE_MIN_FILTER, GL_LINEAR);
21     glTexParameteri(GL_TEXTURE_2D,
22                     GL_TEXTURE_WRAP_S, GL_REPEAT);
23     glTexParameteri(GL_TEXTURE_2D,
24                     GL_TEXTURE_WRAP_T, GL_REPEAT);
25     glTexEnvi(GL_TEXTURE_ENV,
26               GL_TEXTURE_ENV_MODE, GL_MODULATE);
27
28     glTexImage2D(GL_TEXTURE_2D, 0, 3, info->bmiHeader.biWidth,
29                  info->bmiHeader.biHeight, 0, GL_RGB,
30                  GL_UNSIGNED_BYTE, bits);
31
32     free(info);
33     free(bits);
34
35 ...
36
37 }
```

*6.1. kódrészlet. Textúra beállítások OpenGL-ben*

A textúrázás engedélyezéséhez meg kell hívni a `glEnable(GL_TEXTURE_2D)` függvényt (lásd 6.2. kódrészletet) és ezután az adott objektumhoz hozzákötjük a megfelelő textúrát a

`glBindTexture(GL_TEXTURE_2D, texture1)` függvényhívással, ahol a `texture1` egy globális változó és megegyezik a 6.1. kódrészlet `glBindTexture` utasítás második paraméterével. Miután a textúra hozzárendelése megtörtént, egy-egy megfeleltetéssel hozzákötjük az alakzat vertexeihez a megfelelő $(s,t)$ textúra-koordinátákat a `glTexCoord2f(Glfloat s, GLfloat t)` OpenGL függvény meghívásával. Miután már nem használjuk az adott textúrát, letiltjuk a `glDisable(GL_TEXTURE_2D)` függvényhívással.

```
1
2  void Ernyo(int n, double radius1,
3             double radius2, double height)
4  {
5    int i;
6    GLfloat angle;
7    GLfloat x1, y1;
8    GLfloat x2, y2;
9  // Texturazashoz
10   GLfloat f;
11
12   if(n < 3)
13     n = 3;
14
15 //Texturazas
16   glEnable(GL_TEXTURE_2D);
17   glBindTexture(GL_TEXTURE_2D, texture1);
18
19   glBegin(GL_QUAD_STRIP);
20
21   for(i = 0, angle = 0.0, f = 0.0; i <= n; i++,
22         angle += 2.0*GL_PI/n, f += 1.0/n)
23   {
24     x1 = radius1*sin(angle);
25     y1 = radius1*cos(angle);
26
27     x2 = radius2*sin(angle);
28     y2 = radius2*cos(angle);
29
30 //Textura koordinata
31     glTexCoord2f(f, 1.0);
32     glVertex3f(x1, y1, -height/2.0);
33
34 //Textura koordinata
35     glTexCoord2f(f, 0.0);
36     glVertex3f(x2, y2, height/2.0);
37   }
38   glEnd();
39
40 //Texturazas vege
```

```
41    glDisable(GL_TEXTURE_2D);
42  }
```

*6.2. kódrészlet. Textúrázott ernyő megvalósítása*

Ezek után nem marad más dolgunk, mint az elkészült objektumokat a megfelelő pozícióba transzformáljuk és az objektumokat megvalósító függvényeket a `RenderScene` függvényben (lásd 6.3. kódrészletet) meghívjuk.

```
1  void RenderScene( void )
2  {
3  ...
4    glDisable(GL_CULL_FACE);
5    glPushMatrix();
6    glFrontFace(GL_CCW);
7    glRotatef( 70.0f, 1.0f, 0.0f, 0.0f );
8    glPushMatrix();
9      glTranslatef(0.0f, 0.0f, -70.0f);
10     glColor3f(1.0f, 0.0f, 1.0f);
11     Ernyo(100, 1.0f, 30.0f, 15.0f);
12   glPopMatrix();
13   glTranslatef(0.0f, 0.0f, -16.0f);
14   glColor3f(0.0f, 0.0f, 1.0f);
15   Szivoszal(100, 1.0f, 150.0f);
16   glPopMatrix();
17
18 ...
19 }
```

*6.3. kódrészlet. Módosított RenderScene függvény*

### 6.3.1. További textúrázással kapcsolatos függvények

Ebben az alfejezetben néhány olyan függvényt mutatunk be, amelyek hasznos kiegészítésül szolgálnak.

**Színpuffer használata**

Egy- vagy kétdimenziós textúrákat meg lehet adni a színpufferből származó adatokkal. A színpufferből egy kép beolvasásával és annak új textúraként való felhasználását a következő két függvény segítségével tudjuk megvalósítani:

```
void glCopyTexImage1D(GLenum target, GLint level,
GLenum internalformat, GLint x, GLint y,
GLsizei width, GLint border);

void glCopyTexImage2D(GLenum target, GLint level,
GLenum internalformat, GLint x, GLint y,
GLsizei width, GLsizei height, GLint border);
```

Ezek a függvények hasonlóan működnek, mint a `glTexImage` függvény, de ebben az esetben `x` és `y` a színpufferben adjuk meg azt a pozíciót, ahonnan elkezdjük olvasni a textúra adatot. A forrás puffer a `glReadBuffer` függvénnyel állítjuk be, amely hasonlóan viselkedik a `glReadPixels` függvényhez. Megjegyezzük, hogy `glCopyTexImage3D` függvény nem létezik, mivel egy kétdimenziós színpufferből nem lehet térfogati adatokat kinyerni.

**Textúrák frissítése**

Új textúrák többszöri betöltése a valósidőben teljesítmény problémákat okozhat. Ha egy betöltött textúrára nincs többé szükségünk, akkor azt vagy részben vagy teljes egészében le lehet cserélni. Egy textúra lecserélését gyakran sokkal gyorsabban lehet végrehajtani, mint egy új textúrát megadni közvetlenül a `glTexImage` függvénnyel. Ennek a végrehajtására a `glTexSubImage` függvény különböző változatait használhatjuk:

```
void glTexSubImage1D(GLenum target, GLint level,
GLint xOffset,GLsizei width,
GLenum format, GLenum type, const GLvoid *data);

void glTexSubImage2D(GLenum target, GLint level,
GLint xOffset, GLint yOffset,
GLsizei width, GLsizei height,
GLenum format, GLenum type, const GLvoid *data);

void glTexSubImage3D(GLenum target, GLint level,
GLint xOffset, GLint yOffset, GLint zOffset,
GLsizei width, GLsizei height, GLsizei depth,
GLenum format, GLenum type, const GLvoid *data);
```

A legtöbb paraméter megfelel a `glTexImage` függvény argumentumainak. Az `xOffset`, `yOffset`, és `zOffset` paraméterek azokat az eltolásokat határozzák meg, amelyek meglévő textúra képen cserélünk le a megadott textúra adattal. A `width`, `height` és `depth` értékek a beillesztendő textúra méretét adják meg.

Az utolsó függvényhalmaz megengedi számunkra, hogy kombináljuk a színpufferből való olvasást és egy textúra részének beszúrását vagy lecserélését. Ezek a függvények a `glCopyTexSubImage` különböző változatai:

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

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

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

A glCopyTexSubImage3D függvény esetében a színpuffert felhasználhatjuk egy 3D-s textúra egy kétdimenzós szeletének lecserélésére is.

**Textúra mátrix**

A textúra-koordinátákat transzformálhatjuk egy textúramátrixszal. A textúra koordinátákat lehet eltolni, skálázni és forgatni is. A glMatrixMode függvény segítségével a textúramátrixot a GL_TEXTURE paraméterrel jelölhetjük ki. Ezekután a glRotatef(), glScalef(), glTranslatef() valamint a speciális OpenGL mátrix műveletekkel módosíthatjuk a textúramátrixot. A textúramátrix verem hasonlóan működik, mint a transzformációknál ismertetett más fajta (modellnézeti, projekciós stb.) mátrixok a glPushMatrix() és glPopMatrix() függvények meghívásakor. Viszont a textúramátrix verem mélysége maximálisan csak kettő lehet.

**Mipmap szintek automatikus előállítása**

A mipmap textúrázáskor az eredeti textúrakép kicsinyített változataira is szükség van. A GLU segéd-függvénykönyvtár tartalmaz egy gluScaleImage nevű függvényt, amelynek ismételt meghívásával a szükséges mennyiségű mipmap szintet elő lehet állítani. Létezik ennél egy sokkal kényelmesebb módszer, ami létrehozza a skálázott képeket és a glTexImage függvénynek megfelelően be is állítja azokat.

```
int gluBuild1DMipmaps(GLenum target, GLint internalFormat,
    GLint width,GLenum format,
    GLenum type, const void *data);

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

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

Ezeknek a függvényeknek a használata a glTexImage használatával közel megegyezik, de nem rendelkeznek level paraméterrel, ami megadja a mipmap szinteket és nem nyújtanak semmilyen támogatást a textúra határokhoz. Ráadásul fenntartásokkal kell kezelni ezeknek a függvényeknek a használatát, mivel nem fogunk olyan minőséget kapni, mint egy professzionális képszerkesztő esetén.

Amennyiben előre tudjuk azt, hogy az összes mipmap szintet be fogjuk tölteni, akkor használhatjuk az OpenGL hardveres gyorsítását a mipmap szintek előállításához. Ezt a GL_GENERATE_MIPMAP textúra paraméter GL_TRUE-ra való állításával érhetjük el:

```
glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);
```

Amikor ezt a paramétert beállítjuk, akkor minden glTexImage vagy glTexSubImage függvény hívásakor, amelyek az alaptextúrát (0-ik mipmap szintet) frissítik automatikusan frissíti az alacsonyabb szintű mipmap szinteket is. A grafikus hardver használatával ez a módszer alapjában véve gyorsabb, mint a gluBuildMipmaps függvények használata. Azonban meg kell győződnünk arról, hogy az OpenGL 1.4-es verziójával megegyező vagy annál

későbbi változatát használjuk, mivel ez a függvény eredetileg egy OpenGL kiterjesztésben szerepelt, amely csak az 1.4-es verzióval vált a szabvány részévé.

**Részletesség szintjeinek befolyásolása**

Amikor a mipmapping engedélyezve van, akkor az OpenGL egy formula alapján meghatározza, hogy melyik mipmap szintet kell kiválasztani. Be lehet állítani azt az OpenGL-ben, hogy a kiválasztási feltételt hátrébb (a nagyobb mipmap szintek felé) vagy előrébb (a kisebb mipmap szintek felé) tolja. Ez azt eredményezheti, hogy a teljesítmény javul a kisebb mipmap szintek használatakor, vagy a textúrázott objektum „élessége” növekszik nagyobb mipmap szintek használatakor. A következő példában a nagyobb részletességű szintek felé mozgatjuk el a részletességet, ami azt eredményezi, hogy a textúrák élesebbek lesznek és a textúra feldolgozása kissé hosszabb ideig fog tartani:

```
glTexEnvf(GL_TEXTURE_FILTER_CONTROL, GL_TEXTURE_LOD_BIAS, -1.5);
```

# 7. fejezet

# Ütközés-detektálás

Az *ütközés-detektálás* fontos és alapvető alkotórésze sok számítógépes grafikai és virtuális valóság alkalmazásnak. Az ütközés-detektálás része annak, amit gyakran *ütközés kezelésnek* nevezünk, amit három fő részre lehet felosztani: *ütközés-detektálás*, *ütközés-meghatározás* és *válasz az ütközésre*. Az ütközés-detektálás eredménye egy logikai érték, amely megmondja azt, hogy kettő vagy több tárgy ütközött-e vagy sem. Az ütközés-meghatározás megtalálja az aktuális objektumok metszéspontjait. Az ütközés-válasz meghatározza azt, hogy milyen műveletet kell végrehajtani két tárgy ütközésekor.

Mivel egy színtér több száz objektumot tartalmazhat, ezért egy jó ütközés-detektáló rendszernek szintén meg kell birkóznia ezzel a feladattal. Ha a színtér $n$ mozgó és $m$ statikus objektumot tartalmaz, akkor egy naív megközelítés

$$nm + \left( \begin{array}{c} n \\ 2 \end{array} \right) \tag{7.1}$$

objektum tesztet hajtana végre minden egyes képkocka esetén. A kifejezés első tagja a statikus és dinamikus objektumok tesztjeinek a számát, míg a második tag dinamikus objektumok egymással történő tesztjeinek a számát adja meg. $m$ és $n$ növekedésével az elvégzendő tesztek száma nagy mértékben megnő.

Meg kell említenünk, hogy a teljesítmény kiértékelése nagyon bonyolult az ütközés-detektálás algoritmusok esetében, hiszen az algoritmusok függnek az aktuális ütközési forgatókönyvtől és nincs olyan algoritmus, amely minden esetben a legjobban viselkedik.

## 7.1. Ütközés-detektálás sugarakkal

Ebben a fejezetben egy gyors technikát mutatunk be, ami jól működik bizonyos feltételek esetén. Képzeljük el, hogy egy gépkocsi halad felfelé egy emelkedőn és használni akarunk valamilyen információt az útról (például az utat felépítő primitíveket) azért, hogy a kocsi kerekeit az úton tartsuk az animáció közben. Természetesen ezt végre lehet úgy is hajtani, hogy a kerekek és az utat alkotó összes primitív esetén elvégezzük a tesztet. Ugyanakkor nem minden esetben van szükség ilyen részletes ütközés-detektálásra/meghatározásra. Ehelyett, a mozgó objektumot közelíthetjük egy sugárhalmazzal. A gépkocsi esetében elhelyezhetünk

egy-egy sugarat a négy keréknél. Ez a közelítés jól működik a gyakorlatban mindaddig, amíg feltehetjük, hogy csak a négy kerék van kapcsolatban a környezettel (út). Tegyük fel, hogy a gépkocsi egy síkon áll a kezdetekben és a sugarakat úgy helyezzük el, hogy a kezdőpontjaikat a kerekek és a környezet érintkezési pontjában helyezzük el. A kerekeknél elhelyezett sugarak metszését teszteljük a környezettel. Ha a sugár origója és a környezet közötti távolság nulla, akkor a kerék pontosan a talajon van. Amennyiben a távolság nagyobb, mint nulla, akkor a kerék nem érintkezik a környezettel, negatív érték esetén pedig a kerék behatol a környezetbe. Az alkalmazás használhatja ezt a távolságot az ütközés-válasz kiszámítására - a negatív távolság a gépkocsit felfele mozgatná, míg a pozitív távolság a kocsit lefele mozgatná (hacsak a kocsi nem a levegőben repül egy rövid ideig).

A metszéstesztek felgyorsításához a *hierarchikus ábrázolást* alkalmazhatjuk, melyeket a számítógépes grafikában gyakran használunk. A környezetet BSP fával ábrázolhatjuk. Attól függően, hogy milyen primitíveket használunk a környezetben, különböző sugár-objektum metszési módszerek szükségesek.

Amire szükségünk van az a sugár útjában lévő legközelebbi objektum, emiatt a negatív sugárparaméterhez tartozó metszéseket is vizsgálnunk kell. Annak az elkerülésére, hogy két irányba kelljen keresni, a tesztelő sugár origóját mozgatjuk vissza addig, amíg kívül nem esik az út geometriájának határoló térfogatán és akkor teszteljük a környezettel. A gyakorlatban ez csak azt jelenti, hogy a 0 távolságban kezdődő sugár helyett, negatív távolságnál kezdődik a sugárnyaláb.

## 7.2. BSP fák

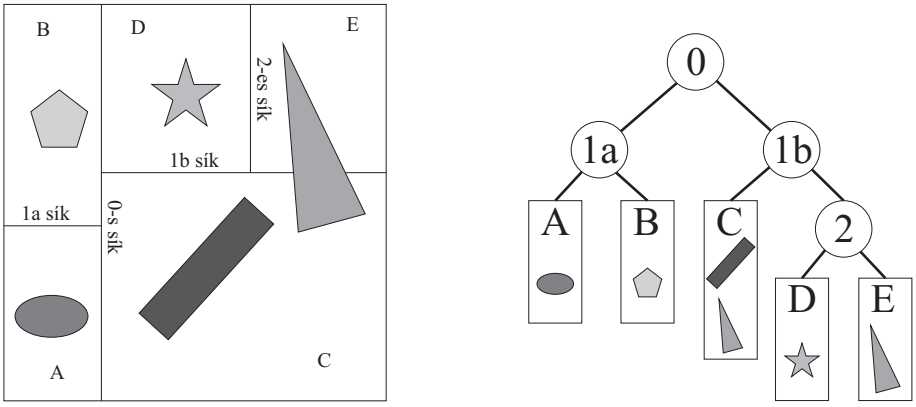
A *bináris térparticionáló fáknak* vagy BSP[1] fáknak két lényegében különböző változata létezik, melyeket tengely-igazított (axis-aligned) és poligon-igazított (poligon-aligned) nevezzük. A fákat a felosztás művelet rekurzív végrehajtásával hozzuk létre. A felosztáskor egy sík[2] segítségével a teret két részre osztjuk, majd a geometriákat ebbe a két részbe rendezzük. Egy érdekes tulajdonsága ezeknek a fáknak az, hogy ha a fákat egy bizonyos módon járjuk be, akkor a geometriai tartalma a fának lerendezhető tetszőleges nézőpontból.

### 7.2.1. Tengely-igazított BSP fák

A tengely-igazított BSP fát a következő módon hozzuk létre. Először a teljes színteret kerítjük be egy *tengely-igazított befoglaló dobozba* (*Axis-Aligned Bounding Box*, röviden AABB). Az alapötlet az, hogy ezt követően rekurzívan felosztjuk ezt a dobozt kisebb dobozokra. A doboz egyik tengelyét kiválasztjuk és egy merőleges síkot állítunk elő, amely kettévágja a teret két dobozra. Néhány esetben rögzítik ezt a felosztó síkot, így pontosan két egyenlő részre osztja fel a dobozt. Máskor engedélyezve van a sík pozíciójának a megváltoztatása. Egy olyan objektum, amelyet a sík elmetsz vagy ezen a szinten van eltárolva vagy mind a két részhalmaz eleme lesz vagy pedig ténylegesen szét van vágva a síkkal két különálló objektumra. Mindegyik részhalmaz ezután egy kisebb dobozban lesz és ez a felosztó-sík

[1] Angolul: Binary Space Partitioning trees
[2] Ez rendszerint a tér egy poligonja.

eljárást ismételjük felosztva mindegyik AABB-t rekurzívan addig, amíg valamilyen feltétel nem tejesül, ami megállítja a folyamatot. Ez a feltétel gyakran egy felhasználó által megadott maximális fa mélység vagy amikor egy dobozban lévő primitívek száma egy felhasználó által definiált küszöbérték alá nem esik. A 7.1. ábra egy példát mutat egy tengelyigazított BSP fára.




*(a) Tér felosztás*

*(b) BSP fa struktúra. Mindegyik levél csomópont egy területet jelöl.*

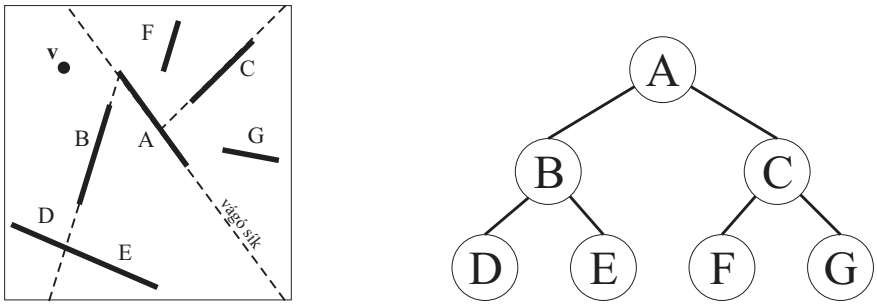
*7.1. ábra. Tengely igazított BSP fa. A térfelosztás a tengely mentén bárhol megengedett, nem csak azok középpontjaiban. A térbeli térfogatok A-tól E-ig vannak felcímkézve.*

Egy stratégia a doboz felosztására a tengelyek ciklikus váltogatása, vagyis a gyökérnél az $x$-tengely, a gyerekeknél az $y$-tengely és az unokák esetén a $z$ tengely mentén vágunk, ezután ezt ismételjük. Egy másik stratégia az, hogy a doboz legnagyobb oldalát keressük meg és e mentén daraboljuk a dobozt. Például, a doboz az $y$-irányban nagyobb, ezután a vágás valamilyen $y = d$ sík mentén fog megtörténni, ahol $d$ egy skalár konstans. Kiegyensúlyozott fához a $d$ értékét kell úgy beállítani, hogy a két tér részbe egyenlő számú primitív kerüljön. Ez egy számítás igényes és nehéz feladat, így gyakran ehelyett a primitívek átlag vagy medián középpontját választjuk.

Az elölről-hátra rendezés egy durva példa arra, hogy hogyan lehet a tengely-igazított BSP fákat használni. Tegyük fel, hogy egy $N$-nel jelölt csomóponton éppen áthaladunk. $N$ a bejárás gyökere. Az $N$ síkját megvizsgáljuk és a fa bejárását rekurzívan folytatjuk a sík azon oldalán, ahol a nézőpont elhelyezkedik. Így csak akkor kezdhetjük el a másik oldal bejárását, amikor a fa fél részét teljesen bejártuk. A közelebbi rész bejárása a fának befejeződhet, amikor egy csomópont doboza teljesen a nézőpont (pontosabban a közelebbi sík) mögött van. Ez nem ad pontos rendezést, mivel az objektumok a fa több csomópontjában is lehetnek. Bár ez ad egy durva rendezést, amely gyakran hasznos. A bejárást a nézőponthoz viszonyított csomópont síkjának a másik oldalán elkezdve az objektumok egy hozzávetőleges rendezését kapjuk hátulról elölre haladva. Ez hasznos az átlátszóság rendezéskor (lásd 5.4. fejezetet). Ez a bejárás hasznos egy sugárnak a színtér geometriához való ellenőrzése esetén. A nézőpont helyzetét egyszerűen a sugár kezdőpontjával kell felcserélni. Egy másik felhasználása a nézeti csonka gúla eltávolítása lehet.

### 7.2.2. Poligon-igazított BSP fák

A másik típusú BSP fa a poligon-igazított forma. Ebben a sémában egy poligont választjuk ki, mint felosztót felező síkként, amely ketté osztja a teret. Ez lesz a fa gyökere. Azt a síkot választjuk ki, amelyiken a poligon fekszik. Arra használjuk ezt a síkot, hogy a színtér maradék poligonjait felosszuk két halmazra. Azokat a poligonokat, amelyeket a felosztó sík elmetsz szétválasztjuk két elkülönülő darabra a metsző vonal mentén. Ezután a felosztó sík mindegyik félsíkjában egy másik poligont választunk felosztóként, amely csak az adott féltérben lévő poligonokat választja szét. Ezt addig folytatjuk rekurzívan, amíg az összes poligon be nem kerül a BSP fába. Egy hatékony poligon-igazított BSP fa előállítása időigényes eljárás és ilyen fákat általában egyszer számítunk ki és aztán az eltárolt változatot újra hasznosítjuk. Egy ilyen típusú BSP fát a 7.2. ábrán láthatunk.




*(a) Tér felosztás* *(b) BSP fa struktúra*

*7.2. ábra. Poligon-igazított BSP fa. A teret először az A poligonnal osztjuk fel. Ezután mindegyik féltér fel lesz osztva B és C poligonokkal. A B poligonnal meghatározott vágó sík a bal alsó sarokban lévő poligont metszi el és feldarabolja azt D és E poligonokra.*

Általában az a legjobb, ha kiegyensúlyozott fát alakítunk ki, azaz egy olyat melynek minden levelének a mélysége ugyanaz vagy legfeljebb csak eggyel tér el a többitől. Egy teljesen kiegyensúlyozatlan fa nem hatékony. Egy példa erre egy olyan fa, ahol mindegyik felosztó poligont úgy választunk ki, hogy a sík egy üres altérre és az összes többi poligonra osztja fel a teret. Sok fajta stratégia létezik a felosztó síkot meghatározó poligon megkeresésére, amely egy jó fát ad vissza. Egy egyszerű stratégia a *legkevésbé-keresztezett feltétel*. Először több lehetséges poligont véletlenszerűen választunk ki. Azt a poligont használjuk, melyet legkevesebb alkalommal metszi el a többi poligon. Egy 1000 poligonból álló teszt színtér esetén empirikus úton bebizonyították, hogy elegendő csak 5 poligont vizsgálni vágási műveletenként ahhoz, hogy jó fát kapjunk eredményül. 5-nél több poligon tesztelésével nem kaptak jobb eredményt, habár ezt a számot valószínűleg növelni kell abban az esetben, ha a színtéren található poligonok száma nagyobb.

Ez a típusú BSP fa rendelkezik néhány hasznos tulajdonsággal. Az egyik az, hogy egy adott nézet esetén a struktúra pontosan bejárható hátulról-előre (vagy elölről-hátulra) haladva. Egy egyszerű pont/sík összehasonlítással lehet meghatározni azt, hogy a kamera a gyökér sík melyik oldalán található. Ettől a síktól távolabb lévő poligonok azután kívül esnek a kamerához közelebbi oldal poligonjaitól. Most a távolabbi oldal halmaza esetén vesszük a

következő szint felosztó síkot és meghatározzuk, hogy a kamera melyik oldalán van. Az a részhalmaz, ahol a kamera megtalálható újból kívül van a közelebbi részhalmazon és a távolabbi oldali részhalmaz távolabb van a kamerától. Ezt rekurzívan folytatva az eljárás létrehoz egy pontos hátulról-előre haladó sorrendet és egy *festő algoritmussal* megjeleníthető a színtér. A festő algoritmus nem használ $Z$-puffert; amennyiben mindegyik objektum ki van rajzolva hátulról előre haladva, mindegyik közelebbi objektumot rárajzoljuk a hátrébb lévő objektumra és így nincs szükség $z$-mélység összehasonlításra.

Például tekintsük a $\mathbf{v}$ nézőpontot a 7.2. ábrán. Figyelmen kívül hagyva a nézeti irányt és a nézeti csonka gúlát, a $\mathbf{v}$ az $A$ vágó sík bal oldalán helyezkedik el. Így a $C$, $F$ és $G$ $B$, $D$ és $E$ mögött vannak. $C$ vágó síkkal összehasonlítva $\mathbf{v}$-t azt kapjuk, hogy $G$ a sík ellentétes oldalán van, így ezt jelenítjük meg elsőként. $B$ sík egy tesztje megadja, hogy $E$-t $D$ előtt kell megjeleníteni. A hátulról előre haladó sorrend ekkor, $G$, $C$, $F$, $A$, $E$, $B$, $D$. Megjegyezzük, hogy ez a sorrend nem garantálja, hogy az egyik objektum közelebb van, mint a másik. Másik felhasználása a poligon-igazított BSP fának az ütközés-detektálása.

## 7.3. Dinamikus ütközés-detektálása BSP fák használatával

Ez az algoritmus meghatározza az ütközéseket a BSP fával leírt geometriával és ütközővel, ami lehet akár egy gömb, egy henger vagy egy objektum konvex burka. Ez szintén alkalmazható dinamikus ütközés-detektálásra. Például ha egy gömb az $n$-edik frame-en lévő $\mathbf{p}_0$ pozícióból az $n+1$e-dik frame-en a $\mathbf{p}_1$ pozícióba mozog, akkor az algoritmus meg tudja mondani, hogy történt-e ütközés a $\mathbf{p}_0$ és $\mathbf{p}_1$ -et összekötő egyenes szakaszon. Az algoritmust olyan kereskedelmi forgalomban kapható játékok használták, ahol a karakter geometriája egy hengerrel volt közelítve.

Az alap BSP fát nagyon hatékonyan lehet vonal darabok tesztelésekor használni. A vonal szegmenst egy pontként lehet ábrázolni, amely $\mathbf{p}_0$-ból $\mathbf{p}_1$-be mozog. Számos metszés lehet, de az első (ha van egyáltalán) adja meg az ütközést a pont és a BSP fában ábrázolt geometria között. Ezt könnyen ki lehet terjeszteni $r$ sugarú gömb kezelésére, ami $\mathbf{p}_0$-ból $\mathbf{p}_1$-be mozog, a pont helyett. A vonal szegmensek és a BSP fa csomópontokban tárolt síkok tesztelése helyett, mindegyik síkot $r$ távolságra mozgatjuk az egység normál mentén. Ezt mindegyik ütközés-kéréskor röptében megcsináljuk, így egy BSP fát bármilyen méretű kör esetén használhatjuk. Feltéve, hogy egy sík $\pi : \mathbf{n}\cdot\mathbf{x}+d = 0$, a kiigazított sík egyenlete $\pi : \mathbf{n}\cdot\mathbf{x}+d\pm+r = 0$, ahol az $r$ előjele attól függ, hogy a sík melyik oldalán folytatjuk a tesztelést egy ütközés keresésében. Feltéve, hogy a karakter a sík pozitív félterében van, azaz $\mathbf{n}\cdot\mathbf{x}+d \geq 0$ ki kell vonnunk az $r$ sugarat a $d$-ből. Megjegyezzük, hogy ekkor a negatív félteret *tömörnek* tekintjük, azaz valami, amit a karakter nem léphet át.

A gömb, egy karaktert nem igazán jól közelíti meg egy játékban. A konvex burka a karaktert alkotó vertexeknek, vagy egy henger, amely körülveszi a karaktert jobb munkát végez. Ahhoz, hogy ezeket a határoló térfogatokat használjuk a $d$ értékét különböző módon kell kiigazítani a sík egyenletében. Egy mozgó konvex burok $S$ vertex halmazának a BSP fával való teszteléséhez a következő skalár értéket kell a síkegyenlet $d$ értékéhez hozzáadni:

$$-\max_{\mathbf{v}_i \in S}(\mathbf{n}\cdot(\mathbf{v}_i - \mathbf{p}_0)). \tag{7.2}$$

A negatív előjel ismét azt tételezi fel, hogy a karakter a síkok pozitív félterében mozog. A $\mathbf{p}_0$ pont tetszőlegesen megválasztott referencia pont. A gömb esetén a gömb középpontját választjuk értelemszerűen. Egy karakter esetén egy lábhoz közeli pont választható vagy talán egy pont a köldöknél. Néha ez a választás egyszerűsíti az egyenleteket (ahogy a gömb középpontja esetén is). A $\mathbf{p}_0$ pont esetén vizsgáljuk az ütközést a kiigazított BSP fában található síkokra. Egy dinamikus kérés esetén, ahol a karakter egyik pontból a másikba mozog egy képkocka alatt, a $\mathbf{p}_0$ pontot a vonal szegmens kezdő pontjaként használjuk. Feltéve, hogy a karakter egy $\mathbf{w}$ vektorral mozdul el egy képkocka alatt a vonal szegmens végpontja $\mathbf{p}_1 = \mathbf{p}_0 + \mathbf{w}$.

A henger talán még hasznosabb, mivel gyorsabban lehet elvégezni a tesztet és elég jól hasonlít egy karakterhez egy játékban, bár a sík egyenletét kiigazító érték származtatása bonyolultabb. Amit általában ebben az algoritmusban csinálni szoktunk az az, hogy határoló a térfogat (gömb, konvex burok és henger ebben az esetben) tesztelését a BSP fával átfogalmazzuk egy $\mathbf{p}_0$ pont tesztelésére a kiigazított BSP fával. Ezután ezt terjesztjük ki egy mozgó objektumra, a $\mathbf{p}_0$ pontot cseréljük ki egy $\mathbf{p}_0$-ból induló és $\mathbf{p}_1$-ben végződő vonal szegmensre.

A 7.3.(a). ábrán láthatóak a henger teszt paraméterei, ahol a $\mathbf{p}_0$ a referencia pont a henger aljának a közepén található. A 7.3.(b). ábra mutatja a henger tesztelését mutatja a $\pi$ síkkal. A 7.3.(c). ábrán a $\pi$ síkot mozgatjuk úgy, hogy a sík éppen csak érinti a hengert. A $\pi$ síkot $\pi'$ síkba mozgatjuk az $e$ távolsággal a 7.3.(d). ábrán látható módon. Így a tesztelést redukáltuk a $\mathbf{p}_0$ pont $\pi'$ síkkal való tesztelésére. Az $e$ értékét röptében számítjuk ki mindegyik síkra és mindegyik képkockára. Gyakorlatban kiszámítjuk a $\mathbf{p}_0$-ból a $\mathbf{t}$-be mutató vektort, ahol az elmozgatott sík érinti a hengert (lásd 7.3.(c). ábrát). Ezután $e$-t a következőképpen számítjuk ki:

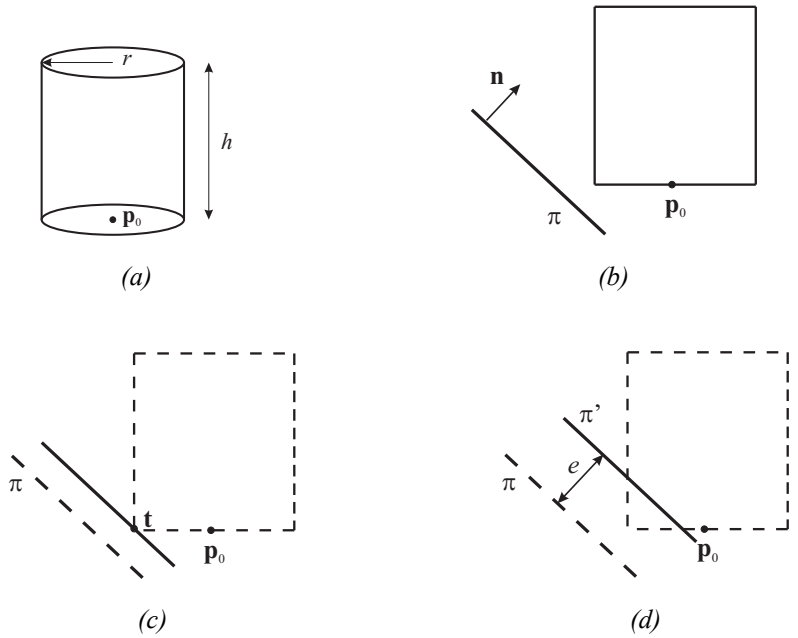
$$e = \left|\mathbf{n} \cdot (\mathbf{t} - \mathbf{p}_0)\right|. \tag{7.3}$$

Ezután már csak $\mathbf{t}$-t kell kiszámítani. A $\mathbf{t}$ $z$-komponense esetén, ha $n_z > 0$, akkor $t_z = p_{0z}$, azaz a $\mathbf{p}_0$ $z$-komponense. Különben $t_z = p_{0z} + h$. Ha $n_x$ és $n_y$ nulla, akkor a henger aljának tetszőleges pontját használhatjuk. Egy választás a henger aljának a középpontja $(t_x, t_y) = (p_x, p_y)$. Különben a henger aljának a szélén lévő pontot választjuk:

$$\begin{aligned} t_x &= \frac{r n_x}{\sqrt{n_x^2 + n_y^2}} + p_x, \\ t_y &= \frac{r n_y}{\sqrt{n_x^2 + n_y^2}} + p_y, \end{aligned} \tag{7.4}$$

ahol a sík normálisát az $xy$-síkra vetítjük le, normalizáljuk és ezután $r$-rel skálázzuk.

Pontatlanság előfordulhat a módszer használata során. Hegyes kiszögellés esetén az ütközést korábban detektálhatjuk. Ennek a problémának a megoldására extra ferde síkokat vezetünk be. Gyakorlatban a külső szögét számítjuk ki a két szomszédos síknak és egy extra síkot veszünk be, ha a szög nagyobb, mint 90°. A ferde síkok természetesen növelik a pontosságot, de nem adnak megoldást az összes problémára.




*7.3. ábra. Henger tesztelése adott π síkkal*

A pszeudokódját ennek az ütközés-detektálásnak a 7.1. kódrészlet mutatja. Ez a BSP fa `N` gyökerével hívjuk meg, melynek a gyerekei az `N.negativechild` és `N.positivechild` és a vonal szakaszt a `p0` és `p1` pontok határozzák meg. Megjegyezzük, hogy az ütközés pontját (ha van ilyen) egy globális `p_impact` változóban kapjuk meg.

```
1  HitCheckBSP(N,v0,v1)
2  returns({TRUE, FALSE});
3  if(not isSolidCell(N)) return FALSE;
4  else if(isSolidCell(N))
5      p_impact = v0;
6    return TRUE;
7  end
8  hit = FALSE;
9  if(clipLineInside(N shift out, v0, v1, &w0, &w1))
10   hit = HitCheckBSP(N.negativechild, w0, w1);
11   if(hit) v1 = p_impact
12 end
13 if(clipLineOutside(N shift out, v0, v1, &w0, &w1))
14   hit = HitCheckBSP(N.positivechild, w0, w1);
15 end
16 return hit;
```

*7.1. kódrészlet. Ütközés-detektálás BSP fával*

Az `isSolidCell` `TRUE` értéket ad vissza, ha elértük a fa levelét és a negatív féltérben

vagyunk. A `clipLineInside` `TRUE` értékkel tér vissza, ha a vonal szakasz (`v0` és `v1`) belül van a csomópont eltolt síkjában, ami a negatív féltérben van. Ez szintén metszi a vonalat a csomópont eltolt síkjával és visszatér az eredmény szakasszal (`w0` és `w1`)-ben. A `clipLineOutside` hasonlóan működik. Megjegyezzük, hogy a `clipLineInside` és a `clipLineOutside` eljárások által visszaadott vonal szakaszok átfedik egymást.

Megmutatták, hogy ez az algoritmus 2.5 és 3.5-szer „drágább” annál az algoritmusnál, amelyik nem használ dinamikusan igazított síkokat. Ez azért van, mert a számításban plusz rezsiköltség van a megfelelő kiigazításban, és azért, mert a BSP fa a ferde síkok miatt nagyobb. Ez a lassulás komolynak tűnhet, de ez az adott környezetben nem is olyan rossz. Az előnye ennek a módszernek az, hogy egy egyszerű BSP fára van szükség az összes karakter és objektum ellenőrzéséhez. Az alternatíva az lehetne, hogy különböző BSP fákban tárolnánk minden különböző sugarú objektum típusokat.

# 8. fejezet

# Térbeli adatstruktúrák

A nagy teljesítmény elérésének gyakori akadálya a valós-idejű alkalmazásokban a geometriai áteresztőképesség. Ezekben az alkalmazásokban a színtér és a modellek sok ezer vertexből épülnek fel, melyekhez normálvektorok, textúra koordináták és más attribútumok kapcsolódnak. Ezt a rengeteg adatot a CPU-nak és a GPU-nak kell feldolgoznia. Ráadásul az adatok másolása grafikus hardver felé szintén szűk keresztmetszet lehet a teljesítményre nézve.

Az OpenGL számos olyan lehetőséget biztosít, amely a gyors geometria áteresztő képességet és az adatok rugalmas és kényelmes kezelését teszi lehetővé a programozó számára.

## 8.1. Display listák

A primitívek kötegeit `glBegin`/`glEnd` függvény párosok segítségével állítjuk össze, melyek egyedi `glVertex` hívásokat tartalmaznak. Ez nagyon rugalmas módja a primitívek összerakásának. Sajnos, amikor a teljesítményt is figyelembe kell venni, akkor ez a lehető legrosszabb módja a geometria továbbítására a GPU felé. Vegyük a következő pszeudokódot, ami egy megvilágított, textúrázott háromszög:

```
glBegin(GL_TRIANGLES);
  glNormal3f(x, y, z);
  glTexCoord2f(s, t);
  glVertex3f(x, y, z);
  glNormal3f(x, y, z);
  glTexCoord2f(s, t);
  glVertex3f(x, y, z);
  glNormal3f(x, y, z);
  glTexCoord2f(s, t);
  glVertex3f(x, y, z);
glEnd();
```

11 függvényhívást kell elvégeznünk egy háromszög létrehozásához. Mindegyik függvény potenciálisan drága ellenőrző kódot tartalmaz az OpenGL meghajtóban, ráadásul 24 különböző négy byte-os paramétert kell a veremre helyezni és természetesen visszaadni a hívó függvénynek. Ez kis munka a CPU-nak. Egy 3D-s színtér létrehozásához 10 000-szer vagy többször ennyi háromszögre van szükség. Könnyű elképzelni, hogy a grafikus hardver

a CPU-ra várakozik, amíg összerakja és továbbítja a geometriai kötegeket. Természetesen van olyan lehetőség, amikor vektor-paraméterű függvényeket használunk, mint például a `glVertex3fv`, amelyek segítségével konszolidálni lehet a köteget, valamint sávokat és legyezőket is használhatunk a redundáns transzformációk és másolások csökkentésére. Az alapmegközelítés azonban abból a teljesítmény igényből ered, hogy szükség van több ezer nagyon kicsi, potenciálisan költséges művelet geometriai kötegekben való elküldésére Ezt a módszert gyakran *közvetlen módú renderelésnek* nevezik.

### 8.1.1. Kötegelt feldolgozás

Az OpenGL, ahogy ezt már korábban említettük, szoftveres felületet biztosít a számítógép grafikus hardveréhez. Azt gondolhatjuk, hogy a meghajtó program az OpenGL a parancsokat „valamilyen" módon speciális hardver utasításokra vagy műveletekre alakítja át és aztán a grafikus kártyára küldi, hogy azokat azonnal végrehajtsa. Ami nagyjából igaz attól eltekintve, hogy ezek a parancsok nem hajtódnak végre egyből. Ehelyett egy lokális pufferben gyűlnek össze, amíg egy határt el nem érnek. Ekkor ezek a parancsok a hardverhez kerülnek/ürítődnek[1]. A fő oka az ilyen típusú elrendezésnek az, hogy a grafikus hardverhez vezető út sok időt vesz igénybe, legalábbis a számítógép idejében kifejezve azt. A puffer küldése a grafikus hardverhez egy aszinkron művelet, ami azt jelenti, hogy a CPU egy másik feladatot kezdhet el és nem kell várnia az elküldött kötegelt renderelési utasítások befejeződéséig. A hardver egy adott parancshalmaz renderelése alatt, amíg CPU azzal van elfoglalva, hogy egy új grafikus képhez tartozó (tipikusan egy animáció következő képkockája) parancsokat dolgoz fel. Ez a fajta *párhuzamosítás* nagyon hatékonyan működik a CPU és a grafikus hardver között.

Három esemény idéz elő ürítést az aktuális renderelő parancsok kötegénél. Az első akkor következik be, amikor a meghajtó program parancs puffere tele van. Ehhez a pufferhez nem férünk hozzá és nem módosíthatjuk annak méretét sem. Akkor is történik ürítés, amikor egy puffer cserét hajtunk végre. Ez a művelet addig nem hajtódik addig, amíg a sorban álló parancsok mindegyike végre nem hajtódik. A puffercsere egy nyilvánvaló jelzés a meghajtó programnak, hogy az adott színtér létrehozása befejeződött és az elküldött parancsok eredményének meg kell jelennie a képernyőn. Amennyiben egyszeres színpuffert használunk, akkor az OpenGL nem szerez tudomást arról, hogy mikor fejeztük be a parancsok küldését és így arról sem tud, hogy mikor kell a kötegelt parancsokat a hardverre küldeni, hogy végrehajtsa azokat. Ahhoz, hogy ezt az eljárást elősegítsük, a `glFlush()` parancsot kell meghívnunk, hogy manuálisan idézzük elő az ürítést.

Néhány OpenGL parancs azonban nem pufferelt későbbi végrehajtás céljából (például `glReadPixels` és `glDrawPixels`). Ezek a függvények közvetlenül érik el a frame puffert és olvassák és írják azt direkt módon. Ezek az utasítások sebesség csökkenést idéznek elő a csővezetéken való áthaladásban, mivel az aktuális sorban álló parancsokat először ki kell üríteni és végre kell hajtani azokat, mielőtt a színpuffert közvetlenül módosítanánk. Erőszakosan kiüríthetjük a parancs puffert és várhatunk arra, hogy a grafikus hardver befejezze az összes renderelési feladatát a `glFinish()` függvény meghívásával. Ezt a függvényt csak nagyon ritkán használjuk a gyakorlatban.

[1] Az eredeti terminológiában angolul ezt *flush*-nak nevezik.

### 8.1.2. Előfeldolgozott kötegek

Az OpenGL parancsok hívásához kapcsolódó tevékenységek igen költségesek. A magas szintű OpenGL parancsok lefordulnak és átalakítódnak alacsony szintű hardver utasításokká. Egy összetett geometria, vagy csak egy nagy mennyiségű vertex adat esetén ez az eljárás több ezerszer végrehajtódik csak azért, hogy egy kép megjelenjen a képernyőn. Természetesen ez az imént említett probléma azonnali renderelési mód esetén.

A geometria vagy másik OpenGL adat gyakran ugyanaz marad képkockáról képkockára. Az egyetlen dolog, ami változik, az a modellnézeti mátrix. A megoldás erre a feleslegesen ismételt többletmunkára az, hogy elmentjük a parancs pufferben található, előre kiszámított adatdarabot, amely valamilyen ismételt renderelési műveletet hajt végre, mint például egy tórusz megrajzolása. Ezt az adatdarabot később bemásolhatjuk egyszerre a parancspufferbe, megtakarítva ezzel a sok függvényhívást és a fordítási munkát, amely az adatot létrehozza.

Az OpenGL megoldást nyújt az előfeldolgozott parancsok létrehozására. Ezt az előfeldolgozott parancslistát *display listának* hívjuk. Ugyanúgy, ahogy az OpenGL primitíveket `glBegin`/`glEnd` utasításokkal határoljuk el, a display listákat `glNewList`/`glEndList` függvény hívásokkal különítjük el egymástól. Egy display listát egy egész értékkel azonosítunk, amit nekünk kell megadni. A következő kód részlet egy tipikus példája a display lista létrehozásának:

```
glNewList(<unsigned integer name>,GL_COMPILE);
// ...
// OpenGL függvényhívások
// ...
glEndList();
```

A `GL_COMPILE` paraméter azt jelzi az OpenGL-nek, hogy csak fordítsa le a listát és még ne hajtsa azt végre (nem fog megjelenni az adott alakzat). Használhatjuk a `GL_COMPILE_AND_EXECUTE` értéket is, ami párhuzamosan felépíti a display listát és végre is hajtja a renderelési utasításokat. Rendszerint a display listákat csak felépítjük a program inicializálási részében és csak a rendereléskor hajtjuk végre azokat.

A display lista azonosító tetszőleges előjel nélküli egész lehet. Azonban, ha ugyanazt az értéket kétszer használjuk, akkor a második display lista felülírja az előzőt. Éppen ezért érdemes egy olyan mechanizmust használni, amely megóv bennünket a már létrehozott display lista felülírásától. Különösen hasznos ez akkor, amikor többen fejlesztenek egy függvénykönyvtárat. A `GLuint glGenLists(GLsizei range)` függvény meghívásával, visszatérési értékként egy egyedi display lista azonosító sorozat első elemét kapjuk vissza. A display lista felszabadítást a `glDeleteLists(GLuint list, GLsizei range)` függvény meghívásával végezhetjük el, amely nem csak a display lista neveket, hanem a listák számára lefoglalt memória területeket is felszabadítja.

Az előre lefordított OpenGL parancsokat tartalmazó listákat a `glCallList(GLuint list)` függvényhívással tudjuk végrehajtani. A display listákat tartalmazó tömböket a `glCallLists(GLsizei n, GLenum type, const GLvoid *lists)` utasítás segítségével tudjuk lefuttatni, ahol az első paraméter a display listák számát adja meg a `lists` nevű tömbben. A függvény második paramétere pedig a tömb adat típusát határozza meg, amely rendszerint `GL_UNSIGNED_BYTE`.

### 8.1.3. Display lista kikötések

Néhány fontos dolgot meg kell említenünk a display listákkal kapcsolatban. Habár a legtöbb megvalósítás esetén egy display lista javít a teljesítményen, mégis annak hatása nagyban függ attól, hogy a gyártók mennyi energiát fektettek a display lista létrehozásának és végrehajtásának optimalizálására. A display listákat tipikusan akkor érdemes használni, amikor a lista állapotváltozásokat tartalmaz (például a fény ki- és bekapcsolása esetén).

Amennyiben a display listák neveit nem a `glGenLists` utasítással hozzuk létre először, akkor lehet, hogy működni fog egyes megvalósítások esetén, de előfordulhat az is, hogy nem.

Néhány parancs használata a display listában nem elfogadható. Például nincs értelme a display listákban a `glReadPixels` függvény hívással a frame puffert betölteni egy memória területre mutató pointerbe . Hasonlóképpen a `glTexImage2D` parancs meghívása közvetlenül a textúra betöltése után eltárolja az eredeti képadatot a display listában. Lényegében a textúra ebben az esetben kétszer akkora memória területen lesz eltárolva. Végül a display lista nem tartalmazhat display lista létrehozást. Azt meg lehet tenni, hogy az egyik display listában meghívjuk a másik display listát, de nem tartalmazhatnak `glNewLists`/`glEndList` függvényhívásokat.

## 8.2. Vertextömbök

A display listákat gyakran használják és egy kényelmes eszköze a feldolgozott OpenGL parancsoknak. Azt is megfontolhatnánk az előző tapasztalatok alapján, hogy a modell vertex adatait előre kiszámítva egy tömbben eltárolhatjuk, megtakarítva ezzel a számítási időt a display listákhoz hasonlóan. Az az egyetlen hátránya ennek a megközelítésnek, hogy végig kell menni a teljes tömbön, és vertexenként kell az adatokat az OpenGL-nek átadni. Ennek az az előnye a display listákkal szemben, hogy a geometria változhat a műveletek során.

Az OpenGL *vertextömbök* használatával mind a két megoldás jó tulajdonságát kihasználhatjuk. Előre kiszámított vagy módosított geometriát lehet nagy mennyiségben, egy időben átvinni a CPU és GPU között. Az alap vertextömbök majdnem olyan gyorsak, mint a display listák, viszont a vertextömbökben tárolt geometriáknak nem kell statikusnak lenniük.

Az OpenGL-es vertextömbök használata négy alaplépést foglal magába:

- Először össze kell rakni a geometriához tartozó adatokat egy vagy több tömbben. Ezt algoritmikusan, illetve egy állományból betöltve is el lehet végezni.
- Meg kell mondani az OpenGL-nek, hogy hol van az adat. Renderelés során az OpenGL a vertex adatot a megadott tömbökből „húzza” be.
- Világosan meg kell mondani, hogy mely tömböket használja az OpenGL. Elkülönített tömbökben tárolhatunk vertexeket, normálvektorokat, színeket és így tovább.
- Végül, végre kell hajtani az OpenGL parancsokat, amelyek a megadott adatok alapján előállítják a megfelelő objektumot/objektumokat.

### 8.2.1. Geometria összeállítása

Az első előfeltétele a vertextömbök használatának az, hogy a modelljeinket tömbökben kell tárolni.

```
// Vertex száma
GLuint VertCount = 100;
// Vertex pointer
GLfloat *pData = NULL;
// Normálvektor pointer
GLfloat *pNormals = NULL;
// ...
// Megfelelő méretű memória terület foglalása
  pData = malloc(sizeof(GLfloat) * VertCount * 3);
  pNormals = malloc(sizeof(GLfloat) * VertCount * 3);
// ...
// Adatok feltöltése
// ...
```

### 8.2.2. Tömbök engedélyezése

A `RenderScene` függvényben engedélyeznünk kell a vertexek és normálvektor tömbök használatát

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
```

A letiltásra a `glDisableClientState(GLenum array)` függvényt használhatjuk. Ezek a függvények a következő konstansokat fogadják el bemeneti paraméterként: `GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_SECONDARY_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, `GL_FOG_COORDINATE_ARRAY`, `GL_TEXURE_COORD_ARRAY` és `GL_EDGE_FLAG_ARRAY`.

Rendszerint egy kérdés szokott felmerülni ezeknek a függvényeknek a bevezetésekor: miért van szükség egy új függvényre, amikor a `glEnable`-t is használhatnánk erre a feladatra? Az OpenGL kliens-szerver modell alapján van megtervezve. A szerver a grafikus hardver és a kliens a gazda CPU és memória. Mivel az engedélyezett/letiltott (enable/disable) állapotot speciálisan a kliens oldali képre alkalmazzuk ezért vezettek be egy új függvényeket.

### 8.2.3. Hol van az adat?

Mielőtt a vertex adatokat használnánk, meg kell mondani, hogy hol tároljuk az adatokat. A következő utasítás erre ad egy példát:

```
glVertexPointer(2, GL_FLOAT, 0, pData);
```

Természetesen a többi vertextömb adattípusokhoz a nekik megfelelő függvényeket kell használni:

```
void glVertexPointer(GLint size, GLenum type,
      GLsizei stride, const void *pointer);
```

```
void glColorPointer(GLint size, GLenum type,
      GLsizei stride, const void *pointer);
void glTexCoordPointer(GLint size, GLenum type,
      GLsizei stride, const void *pointer);
void glSecondaryColorPointer(GLint size, GLenum type,
      GLsizei stride, const void *pointer);
void glNormalPointer(GLenum type,
      GLsizei stride, const void *pData);
void glFogCoordPointer(GLenum type,
      GLsizei stride, const void *pointer);
void glEdgeFlagPointer(GLenum type,
      GLsizei stride, const void *pointer);
```

Ezek a függvények nagyon hasonlóak egymáshoz és majdnem egyformák az argumentumaik is. A normálvektor, köd koordináta és az él flag kivételével mindegyik függvény első paramétere a `size`. Ez a paraméter a koordináta típus elemeinek számát tartalmazza. Például egy vertex kettő $(x, y)$, három $(x, y, z)$ vagy négy $(x, y, z, w)$ komponensből áll.

A `type` paraméter adja meg a tömb adattípusát. Nem mindegyik adattípus használható a vertextömb specifikáció során. A 8.1. táblázat foglalja össze a hét vertextömb függvény lehetséges adattípusait.

A `stride` paraméter byte-ban adja meg két egymást követő tömbelem közötti eltolás mértékét. Amennyiben ez az érték 0-val egyenlő, akkor ez azt jelenti, hogy az elemek a tömbben szorosan egymásután vannak elhelyezve. Végül az utolsó paraméter, az adat tömbre mutató pointer. A multi-textúrák esetén a `glBegin`/`glEnd` függvény páros használatakor az új textúra-koordinátákat a `glMultiTexCoord` függvény hívással lehet mindegyik textúra egységhez elküldeni. Vertextömbök esetén a cél textúra egységet `glClientActiveTexture(GLenum texture)` függvény hívással lehet beállítani a `glTexCoordPointer` számára. A `target` paraméter `GL_TEXTURE0`, `GL_TEXTURE1` ... értékeket veheti fel.

### 8.2.4. Adatok betöltése és rajzolás

Végezetül, készen állunk arra, hogy a vertextömbjeinket használjuk. Lényegében kétféle módon használhatjuk azokat.

Mivel az OpenGL ismeri a vertex adatokat, a következő kóddal kinyerhetjük a vertex adatokat OpenGL-ben.

```
glBegin(GL_POINTS);
for(i = 0; i < VertCount; i++)
  glArrayElement(i);
glEnd();
```

A `glArrayElement` függvény veszi a megfelelő tömb adatokat azokból a tömbökből, amelyek a `glEnableClientState` függvénnyel engedélyezve lettek.
A `glEnableClientState` több függvényhívást helyettesít (például a `glNormal`, `glColor`, `glVertex` és így tovább). Amennyiben egy adott blokkot az elejétől a végig át akarunk

| Parancs | Elemek | Érvényes adattípusok |
|---|---|---|
| glColorPointer | 3, 4 | GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE |
| glEdgeFlagPointer | 1 | nem meghatározott (mindig GLboolean) |
| glFogCoordPointer | 1 | GL_FLOAT, GL_DOUBLE |
| glNormalPointer | 3 | GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE |
| glSecondaryColorPointer | 3 | GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE |
| glTexCoordPointer | 1, 2, 3, 4 | GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE |
| glVertexPointer | 2, 3, 4 | GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE |

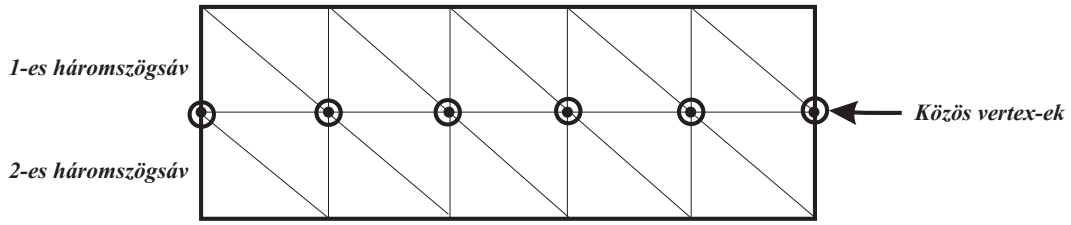
*8.1. táblázat. Vertextömb méretek és adattípusok*

küldeni, akkor a `glDrawArrays(GLenum mode, GLint first, GLint count)` függvényhívással egyszerűen megtehetjük. A `mode` paraméter adja meg, hogy milyen primitívet akarunk előállítani. A `first` argumentum határozza meg, hogy a tömb melyik elemétől kezdve és a `count` paraméter adja meg, hogy hány elemet akarunk a tömbökből kinyerni. Így az előző kódrészletet egy egyszerű `glDrawArrays(GL_POINTS, 0, VertCount)` függvényhívással helyettesíthetjük.

## 8.3. Indexelt vertextömbök

Az indexelt vertextömbök is vertextömbök, csak ebben az esetben vertextömbhöz tartozik egy külön index értékeket tároló tömb, amely megadja azt, hogy melyik vertexeket és milyen sorrendben kell felhasználni az adott objektum felépítésekor. Ez egy kicsit nyakatekertnek tűnhet először, de ezzel a módszerrel memória területet takaríthatunk meg és csökkenthetjük a transzformációs költségeket is. Ideális körülmények között gyorsabb is lehet, mint a display lista.

A kapcsolódó primitívek közös vertexekkel rendelkezhetnek, melyeket nem lehet egyszerűen háromszögsávok, háromszög-legyezők, négyszögsávok használatával megoldani. Például két szomszédos háromszögsáv esetén, akár hagyományos, akár vertextömbbel való renderelési módszert használunk nem létezik olyan módszer, amellyel vertexek halmazát meg lehetne osztani (a 8.1. ábrán a bekarikázott vertexeket kétszer kell megadni).




*8.1. ábra. Két háromszögsáv*

Amennyiben a vertexeket vagy a normálvektorokat újra felhasználjuk a vertextömbökben, akkor csökkenteni tudjuk a memória használatot, valamint egy jó OpenGL megvalósításban a transzformációk számát, illetve a transzformációval töltött időt is jelentősen csökkenteni lehet.

A következő példában egy egyszerű kockát hozunk létre ilyen módon (8.1. példa). Ahelyett, hogy az összes vertexet tartalmazó vertextömböt hoznánk létre, csak az egyedi vertexeket adjuk meg. Ezután egy másik index tömb segítségével adjuk meg a megfelelő geometriát.

```
1 // ...
2 // Tömb, amely a kocka nyolc sarkát tartalmazza
3 static GLfloat sarkok[] =
4 // A kocka előlapja
5   { -25.0f, 25.0f, 25.0f,
```

```
6     25.0f, 25.0f, 25.0f,
7     25.0f, -25.0f, 25.0f,
8     -25.0f, -25.0f, 25.0f,
9   // A kocka hátlapja
10    -25.0f, 25.0f, -25.0f,
11    25.0f, 25.0f, -25.0f,
12    25.0f, -25.0f, -25.0f,
13    -25.0f, -25.0f, -25.0f };
14
15  // Az index tömb, ami a kockát állítja elő
16  static GLubyte indexek[] =
17    // Előlap
18    { 0, 1, 2, 3,
19    //Felső lap
20    4, 5, 1, 0,
21    Alsó lap
22    3, 2, 6, 7,
23    // Hátlap
24    5, 4, 7, 6,
25    // Jobb oldali lap
26    1, 5, 6, 2,
27    // Bal oldali lap
28    4, 0, 3, 7 };
29
30  // ...
31  void RenderScene(void)
32  {
33  // ...
34
35  // A vertextömb engedélyezése és megadása
36  glEnableClientState(GL_VERTEX_ARRAY);
37  glVertexPointer(3, GL_FLOAT, 0, sarkok);
38
39  // Négyszögsávokkal való megjelenítés
40  glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, indexek);
41
42  // ...
43  }
```

*8.1. kódrészlet. Egy kocka megadása indexelt vertextömb segítségével*

A `glDrawElements` függvény nagyon hasonlít a `glDrawArrays` függvényre. Azonban a glDrawElements esetén az index tömböt is meg kell adni, ami meghatározza, hogy milyen sorrendben kell a vertextömböt tekinteni. Másik változata a `glDrawElement` függvénynek a `glDrawRangeElements`, ami két plusz paraméter segítségével megadja, hogy az indexek mely részét kell felhasználni az objektum létrehozásához. További lehetőséget biztosít a `glMultiDrawArrays` függvény, amely segítségével több index tömböt lehet elküldeni egyetlen függvényhívás segítségével. A `glInterleavedArrays` függvény pedig lehetővé

teszi, hogy több tömböt egy összesített tömbben helyezzünk el. Nincs változás a tömbök elérésében vagy azok átküldésében, de a memória szervezése valószínűleg javíthatja a teljesítményt néhány hardveres megvalósítás esetén.

## 8.4. Vertex puffer objektumok

A display listák egy gyors és könnyű módszert adnak az azonnali kódolási módban (a `glBegin`/`glEnd` használatakor). A legrosszabb esetben a display lista egy előre lefordított OpenGL adatot fog tartalmazni, amely készen áll arra, hogy gyorsan a parancs pufferbe másoljuk és elküldjük a grafikus hardverhez. Legjobb esetben viszont egy adott megvalósítás egy display listát a grafikus hardverbe másolhat, lecsökkentve a hardver sávszélességét lényegében nullára. Ez utóbbi felettébb kívánatos, de az elért teljesítményjavulás nagysága eléggé bizonytalan. Továbbá a display listákat létrehozásuk után nem lehet módosítani.

A vertextömbök viszont biztosítják számunkra az összes rugalmasságot, amit szeretnénk. Továbbá az indexelt vertextömbök segítségével csökkenthetjük a vertex adatok mennyiségét, amelyet a hardverhez kell átküldeni, ezáltal csökkentve az elvégzendő transzformációk számát. A dinamikus objektumok, mint például ruha, víz esetén a vertextömbök használata egy kézenfekvő választás lehet.

Az OpenGL még egy lehetőséget biztosít a geometriai áteresztőképesség vezérlésére. A vertextömbök használatakor át lehet küldeni a CPU kliens oldaláról egyedi tömböket a grafikus hardverre. Ezt a tulajdonságot nevezzük *vertex puffer objektumnak*, amely lehetővé teszi azt, hogy a vertextömböket ugyanúgy használjuk és kezeljük, mint a textúra adatok betöltését és kezelését.

### 8.4.1. Vertex puffer objektumok kezelése és használata

Az első dolog, amit meg kell tennünk a vertex puffer objektumok használatához az, hogy vertextömböket kell használnunk. Ezek használatát az előző fejezetekben (lásd 8.2. és 8.3. fejezeteket) már bemutattuk. Ezután puffer objektumokat kell létrehoznunk a `glGenBuffers` függvény meghívásával, amelynek első paramétere a kért objektumoknak a számát adja meg, a második paraméter pedig egy olyan tömb, ami az új puffer objektumok neveivel van feltöltve. A puffereket a `glDeleteBuffers` utasítással lehet felszabadítani. A vertex puffer objektumok össze vannak kötve, hasonlóan a textúra objektumokhoz. A `glBindBuffer` függvény összeköti az aktuális állapotot egy bizonyos puffer objektumhoz. A `target` paraméter határozza meg azt, hogy milyen típusú tömböt fogunk kijelölni. Ez lehet `GL_ARRAY_BUFFER` a vertex adatok esetén (beleértve a normálvektorokat, textúrakoordinátákat stb.) vagy `GL_ELEMENT_ARRAY_BUFFER` index tömbök számára, amelyeket a `glDrawElements` és más index-alapú rendereléskor használunk.

**Puffer objektumok betöltése**

A vertex adatok grafikus hardverre történő másolásakor először hozzá kell csatolni a szóban forgó puffer objektumot és aztán kell meghívni a `glBufferData` függvényt.

```
glBufferData(GLenum target, GLsizeiptr size,
             GLvoid *data, GLenum usage)
```

A `target` paraméter ismét a `GL_ARRAY_BUFFER` vagy `GL_ELEMENT_ARRAY_BUFFER` értékeket kaphatja meg. A `size` adja meg a vertextömb méretét byte-ban. Az utolsó paraméter pedig egy teljesítmény használati utasítás, melynek lehetséges értékeit a 8.2. táblázat foglalja össze.

| Használati utasítás | Leírás |
|---|---|
| `GL_DYNAMIC_DRAW` | A puffer objektumban tárolt adat valószínűleg sokszor fog változni, de a változások között a kirajzolásra többször fogja használni a forrást. Ez a segítség a megvalósítás számára jelzi, hogy az adatot olyan helyen kell tárolni, melynek időnkénti megváltoztatása nem okoz nagy gondot. |
| `GL_STATIC_DRAW` | A puffer objektumban tárolt adat nem valószínű, hogy változni fog és sokszor ez lesz a forrása a rajzolásnak. Ez azt jelzi, hogy a megvalósításnak az adatot olyan helyen kell tárolnia, ami gyorsan olvasható, de nem szükséges gyorsan frissíteni azt. |
| `GL_STREAM_DRAW` | A pufferben tárolt adat gyakran változik és a változások között csak néhányszor lesz szükség a forrásra. Ez a segítség azt jelzi a megvalósításnak, hogy időben változó geometriai (például animált geometria) objektumot tárolunk, amit egyszer használunk és utána meg fog változni rögtön. Elengedhetetlen, hogy az ilyen jellegű adatot olyan helyen tároljuk, amelyet gyorsan lehet frissíteni még a gyorsabb renderelés kárára is. |

*8.2. táblázat. Puffer objektum használati utasítás*

### 8.4.2. Renderelés vertex puffer objektumokkal

Két dologban van eltérés a vertextömb objektumok esetén. Az első az, hogy hozzá kell rendelni az adott vertextömböt a renderelési módhoz mielőtt a vertex mutató függvényt meghívnánk. Másodsorban, az aktuális tömbre mutató pointer ezentúl egy eltolás lesz a vertex puffer objektumban. Például a

```
glVertexPointer(3, GL_FLOAT,0, pVerts);
```

függvényhívás ezentúl a következőképpen fog kinézni:

```
glBindBuffer(GL_ARRAY_BUFFER, bufferObjects[0]);
glVertexPointer(3, GL_FLOAT,0, 0);
```

Ugyanez érvényes a renderelési függvényhívásokra is. Például:

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, bufferObjects[3]);
glDrawElements(GL_TRIANGLES, nNumIndexes, GL_UNSIGNED_SHORT, 0);
```

## 8.5. Poligon technikák

Eddig a pontig azt feltételeztük, hogy a megjelenítendő modell pontosan olyan formátumban áll a rendelkezésünkre, amilyenre éppen szükségünk van és az a megfelelő részletességgel van kidolgozva. A valóságban csak ritkán van ilyen szerencsénk. A modellező programoknak és az adatelőállító eszközöknek saját fura szokásaik és korlátaik vannak, amelyek félreérthetőséget és hibát okoznak az adathalmazban és így a megjelenítés során is.

A poligon ábrázolás általános célja a vizuális pontosság és a sebesség. A „*pontosság*” mindig az adott környezettől függ. Ez jelentheti azt, hogy egy modell adott pontossággal jelenik meg; például egy repülőgépszimulátor esetén, ahol fontos az általános benyomás. A mérnök irányítani és pozicionálni akarja a modelleket valósidőben és minden részletet minden időpillanatban látni akar a számítógépen. Összehasonlítva ezt egy játékkal, ahol ha a képkockák sebessége elég magas, kisebb hibák vagy pontatlanságok az adott képkockán belül megengedettek, hiszen nem biztos, hogy észreveszik azt vagy a következő képkockán már nem lesz látható. A valósidejű munkák esetén mindig fontos ismerni azokat a határokat, ahol a problémákat meg kell oldani, hiszen ezek meghatározzák azokat a technikákat, amelyek alkalmazhatóak azokra.

### 8.5.1. Poligonokra és háromszögekre való felbontás

A *poligon felbontás* az a folyamat, amikor a felületet poligonok halmazára bontjuk fel. Jelen esetben poligon felületek felbontásával fogunk foglalkozni. Sok grafikai alkalmazásprogramozási felület (API) és grafikus hardver háromszögekre van optimalizálva. A háromszögek, mint elemi alkotó részek vesznek részt a renderelés során, belőlük tetszőleges felületek előállíthatóak.

A renderelő lehet, hogy csak konvex poligonokat tud kezelni vagy a felületet kisebb részekre kell vágni azért, hogy az árnyalás vagy a visszatükröződő fény megfelelően jelenjenek meg. Nem grafikai okokból a poligon felbontáskor meg szoktak fogalmazni olyan feltételeket, mint például, hogy egyetlen egy poligon se legyen nagyobb egy előre megadott területnél vagy a háromszögek esetén a háromszögek szögeinek nagyobbaknak kell lenniük egy előre megadott minimum szögnél. A szögekre vonatkozó kikötések nem-grafikai alkalmazások (pl. véges elem analízis) esetén megszokottak, ezek egy felület megjelenését is javítják. A hosszú, vékony háromszögeket jobb elkerülni, mivel különböző árnyalások esetén a vertexek közötti nagy távolságok miatt interpoláláskor a hibák jobban megjelennek.

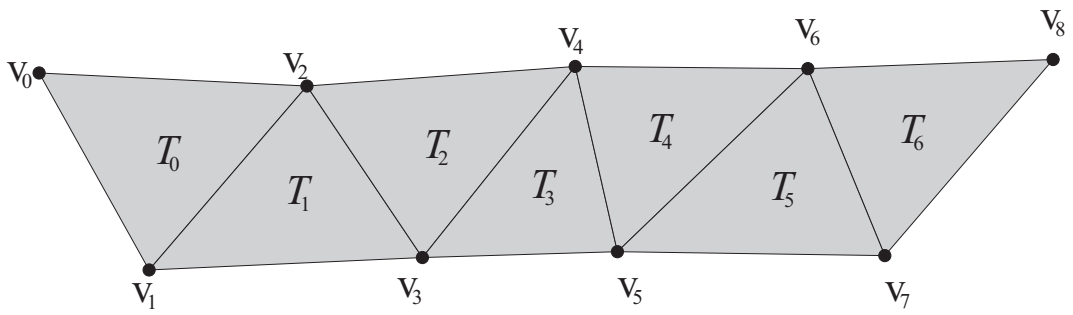
Az első lépés egy felület felbontása esetén az, hogy hogy egy 3D-s poligon esetén meghatározzuk azt, hogy melyik a legjobb vetítés vetítés 2D-re. Erre azért van szükség, hogy a problémát és a probléma megoldásául szolgáló algoritmust leegyszerűsítsük. Az egyik módszer az, amikor eldöntjük, hogy melyik $xyz$ koordinátát töröljük annak érdekében, hogy csak kettő maradjon. Ez egyenértékű azzal, amikor a poligont leképezzük az $xy$, $yz$ és $xz$ síkokra. Az a legjobb sík, amikor az adott poligon esetén a legnagyobb leképezett területet kapjuk. Ez a sík meghatározható úgy is, hogy egyszerűen kidobjuk azt a koordinátát, amely a legnagyobb nagyságú a poligon normálvektorában. Például, ha a poligon normálvektora $(-5, 2, 4)$, akkor az $x$ koordinátát hagyjuk el, mivel a $-5$ a legnagyobb abszolút értékben a vektorban. A poligon normálvektor tesztjével nem mindig lehet meghatározni a legnagyobb

területű vetületet. Az eredmény helyessége függ a normálvektor kiszámítási módjától és attól, hogy a poligon sík-e vagy sem.

### 8.5.2. Háromszögsávok és hálók

A grafikus teljesítmény növelésének egy nagyon gyakori módja az, hogy kevesebb vertexet küldünk át háromszögenként a grafikus csővezetéken. Nyilvánvaló előnyei, hogy kevesebb pontot és normálvektort kell transzformálni, kevesebb vonalvágást kell végrehajtani, kevesebb megvilágításhoz tartozó számítást kell elvégezni és sorolhatnánk még a többi elvégzendő műveletet. Habár egy alkalmazás szűk keresztmetszete lehet a kitöltési sebesség (azaz a másodpercenként kitöltendő pixelek száma).

A háromszögsávok és háromszöglegyezők esetén a közös vertexeket csak egyszer kell elküldeni a grafikus csővezetékbe, hiszen az első három vertex megadása után minden új vertexszel egy új háromszöget hozunk létre (lásd 8.2. ábrát).




*8.2. ábra. Háromszögek sorozata egy háromszögsávként ábrázolható. Megjegyezzük, hogy a háromszögek irányítottsága háromszögenként váltakozik, ennek következtében az első háromszög irányítottsága határozza meg az összes háromszög körbejárását.*

Egy szekvenciális háromszögsávot, rendezett vertexek listájával definiálhatjuk $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n$, ahol a $T_i$ az $i$-ik háromszöget $\triangle \mathbf{v}_i, \mathbf{v}_{i+1}, \mathbf{v}_{i+2}$ jelöli, $0 \leq i < n-2$ esetén. Azért hívjuk szekvenciálisnak, mert a vertexeket a megadott sorrendben küldjük a GPU felé. A definícióból következik, hogy a szekvenciális háromszögsáv $n$ vertexe $n-2$ háromszöget határoz meg. Az $n-2$-t a háromszögsáv hosszának nevezzük.

Ha egy szekvenciális háromszögsáv $m$ háromszöget tartalmaz, akkor az első három vertex alkotja az első háromszöget. Újabb vertexek hozzá vételével kapjuk a maradék $m-1$ háromszöget. Ez azt jelenti, hogy a vertexek $v_a$ átlagos száma egy $m$ hosszú szekvenciális háromszögsáv esetén a következőképpen fejezhető ki:

$$v_a = \frac{3 + (m-1)}{m} = 1 + \frac{2}{m}. \tag{8.1}$$

Könnyen látható, hogy $m \to \infty$ esetén $v_a \to 1$ teljesül. Valós esetben ennek nincs nagy jelentősége. Amennyiben $m = 10$, akkor $v_a = 1.2$. ami azt jelenti, hogy átlagosan 1.2 vertexet küldünk át háromszögenként. Ebből adódik a háromszögsávok jelentősége.

Attól függően, hogy hol van a renderelési csővezetéknek a szűk keresztmetszete, potenciálisan lehetőség van arra, hogy a renderelési idő kétharmadát megspóroljuk szekvenciális háromszögsávok segítségével.[2] A sebességnövekedés a redundáns műveletek, mint például az adatok grafikus hardverre való küldése, megvilágítási számítások elvégzése, vágás, mátrix transzformációk stb. elkerülése miatt következik be.

Ha nem követeljük meg a szigorú szekvenciáját a háromszögeknek, ahogy ezt a szekvenciális háromszögsávok esetén tesszük, akkor hosszabb és ezáltal hatékonyabb sávokat hozhatunk létre. Ezeket a háromszögsávokat *általánosított háromszögsávoknak* nevezzük. Ahhoz, hogy ilyen sávokat tudjunk előállítani szükség van egy fajta vertex gyorsítótárra a grafikus kártyán, amely a transzformált és a megvilágított vertexeket tárolja, ahol a vertexeket el lehet érni és ki lehet cserélni rövid bit kódok küldésével. Így a háromszögsáv előállító a puffer tartalmát teljes mértékben kézben tarthatja, bár néhány esetben ezek a tárak FIFO típusúak, amikor a felhasználónak nem kell kezelni azt. Amikor a vertexek a gyorsítótárba kerülnek, akkor más háromszögek is felhasználhatják azokat elenyésző költséggel.

A szekvenciális háromszögek „általánosításához” a *csere* műveletet kell bevezetnünk, amely megcseréli a két utolsó vertexnek a sorrendjét. Az Iris GL-ben[3], erre külön utasítás létezik. OpenGL-ben és Direct3D-ben viszont a csere parancsot egy vertex újra küldésével valósíthatjuk meg, ami cserénként egy vertexnyi plusz költséget jelent. A csere ilyen módú megvalósítása egy olyan háromszöget hoz létre, melynek nincs területe. Mivel egy új háromszögsáv létrehozásának a költsége két vertex, szemben a csere egy plusz vertexével, még így is jobban járunk, mintha újra kezdenénk a háromszögsávot. Továbbá az aktuális API hívások, melyek a sáv küldésért felelősek, további költségeket jelentenének, így kevesebb API hívás szintén növelheti a teljesítményt. Egy háromszögsáv, amely a $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, csere, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6)$ vertexek átküldésére várakozik, megvalósítható a következőképpen $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_2, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6)$, ahol a csere a $\mathbf{v}_2$ vertex újraküldésével lett megvalósítva (lásd 8.3. ábrát).

Ahogy korábban említettük a háromszög-legyező (lásd 8.4. ábrát) hasonló jó tulajdonsággal rendelkezik, mint a háromszögsáv. A legtöbb alacsony szintű grafikus API támogatja a háromszög-legyezők létrehozását. Megjegyezzük, hogy egy általános konvex poligont könnyen lehet háromszög-legyezővé alakítani és természetesen háromszögsávvá is könnyű alakítani.

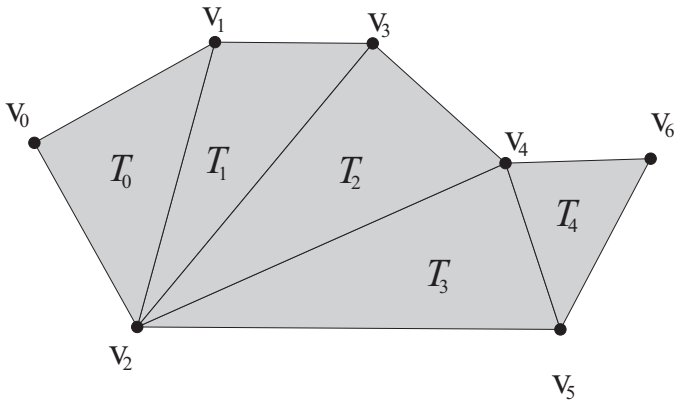
A *középső vertexen* az összes háromszög osztozik. Egy új háromszöget a középső vertex, az előzőleg elküldött vertex és az új vertex segítségével hozzuk létre. Az $n$ vertexből felépülő háromszög-legyezőt a $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$ rendezett vertex listái alkotják, ahol $\mathbf{v}_0$ a középső vertex. Az $i$-ik háromszög $\triangle \mathbf{v}_0, \mathbf{v}_{i+1}, \mathbf{v}_{i+2}$ jelöli, $0 \leq i < n-2$ esetén. A 8.1. képletet alkalmazva, ebben az esetben is $m \to \infty$ esetén, $v_a$ a háromszög-legyezőknél is egy vertexhez tart háromszögenként. Bármelyik háromszög-legyező átalakítható háromszögsávvá (amely sok cserét fog tartalmazni), de ez fordítva nem hajtható végre.

**Sávok előállítása**

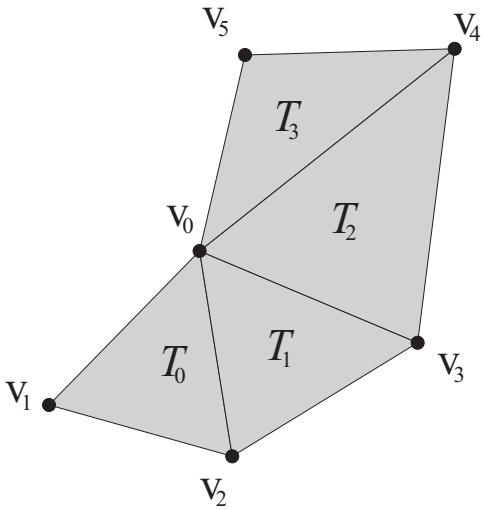
Egy adott általános háromszöghálót érdemes hatékonyan szétbontani háromszögsávokra.

[2] Ez háromszög-legyezők esetén is igaz. Néhány gyártó cég azt javasolja, hogy inkább háromszögsávokat használjunk a háromszög-legyezők helyett is a meghajtó optimalizálása miatt.

[3] Integrated Raster Imaging System Graphics Library, melyet a Silicon Graphics (SGI) fejlesztett ki 2- és 3D-s számítógépes grafika létrehozására az IRIX-alapú IRIS grafikus munkaállomásaikon




*8.3. ábra. A grafikus csővezetékbe a* $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_2, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_6)$ *vertexeket küldjük, amelyek egy háromszögsávot fognak létrehozni. A csereművelet a* $\mathbf{v}_2$ *vertex kétszeri alkalmazásával van megvalósítva.*




*8.4. ábra. Háromszög-legyező. A* $T_0$ *háromszöghöz a* $\mathbf{v}_0$ *(középső vertex),* $\mathbf{v}_2$ *és* $\mathbf{v}_2$ *vertexeket küldi el. A rákövetkező* $T_i$ $(i > 0)$ *háromszög esetén csak a* $v_{i+2}$ *vertexet küldi el.*

Optimális háromszögsávok előállítására bebizonyították, hogy NP-teljes probléma és ezért meg kell elégednünk heurisztikus módszerekkel, amelyek a háromszögsávok számának az alsó határához közelítenek. A következőkben egy mohó stratégiát fogunk bemutatni a szekvenciális háromszögsávok létrehozására.

Mindegyik háromszögsáv létrehozó algoritmus a poligon halmaz szomszédsági adat struktúrájának a létrehozásával kezdődik, ahol mindegyik poligonhoz tartozó él esetén szomszédos poligonra való hivatkozást is el kell tárolni. A poligonok szomszédjainak a számát *foknak* nevezzük, és egész értéke 0 és a poligon vertexeinek a száma között van.

*Euler* síkbeli kapcsolódó gráfokra vonatkozó tétele (8.2. egyenlet) alapján meghatározhatjuk a vertexek átlagos számát, amit a csővezetékbe küldünk.

$$v - e + f - 2g = 2, \tag{8.2}$$

ahol $v$ a vertexek számát, $e$ az élek számát, $f$ a lapok számát és $g$ az objektumban lévő lyukak számát jelöli. Mivel kapcsolódó gráfok esetén minden él mentén két lap helyezkedik el és minden lapnak legalább három éle van, ezért a$2e \geq 3f$ mindig teljesül. Behelyettesítve ezt *Euler tételbe*, egyszerűsítés után azt kapjuk, hogy $f \leq 2v - 4$. Ha mindegyik lap háromszög, akkor $2e = 3f \Rightarrow f = 2v - 4$. Mindent összevetve ez azt jelenti, hogy a háromszögek száma kisebb vagy egyenlő a vertexek számának a kétszeresénél a háromszögekre való felbontáskor. Mivel a háromszögenkénti vertexek száma a háromszögsávban az egyhez tart, minden vertexet (átlagosan) legalább kétszer kell elküldeni a szekvenciális háromszögsáv használatakor.

*Továbbfejlesztett SGI háromszögsáv-képző algoritmus*

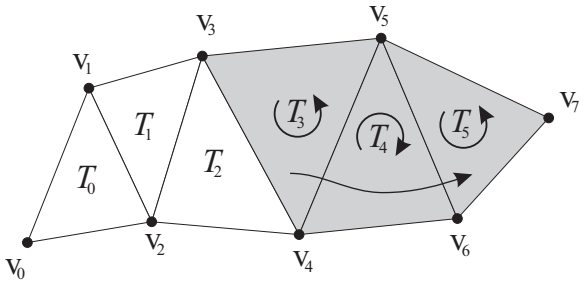
Ez az algoritmus csak olyan modellek esetén működik, amelyek teljesen háromszögekből épülnek fel. A mohó algoritmusok olyan optimalizálási algoritmusok, amelyek a lokális optimumok alapján döntenek (azt választják, amely a legjobb az adott pillanatban). Az SGI algoritmus is egy ilyen mohó algoritmus, ahol mindig azt a kezdő háromszöget választja, amelyiknek a legkisebb a foka (legkevesebb szomszédos oldala van). Néhány módosítással az SGI algoritmus a következőképpen néz ki:

1. Válasszuk ki a kezdő háromszöget.
2. Építsünk 3 különböző háromszögsávot a háromszög minden éle mentén.
3. Terjesszük ki ezeket a háromszögsávokat az háromszögsáv első elemétől az ellenkező irányba.
4. Válasszuk ki a három közül a leghosszabb háromszögsávot és töröljük a többit.
5. Ismételjük meg a folyamatot az 1-es lépéstől addig, amíg az összes háromszög be nem került a sávba.

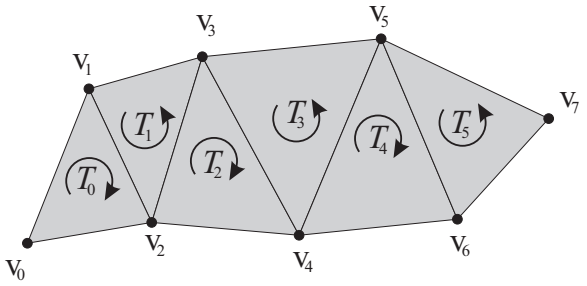
Az első lépésben az algoritmus a legkisebb fokszámú háromszöget választja ki. Amennyiben több ilyen megegyező fokszámú háromszög létezik, akkor az algoritmus a szomszédos háromszögek szomszédjainak a fokszáma alapján dönt. Ha ezek után még mindig nem egyértelmű a háromszög kiválasztása, akkor tetszőlegesen kiválaszt egyet.

Végezetül a sávot a háromszögsáv kezdő és végső háromszögének az irányba terjesztjük ki. Az alapötlet az, hogy az elkülönített háromszögek nem szerepelnek egyetlen egy háromszögsávban sem. Lényegében ezeknek a háromszögeknek a számát minimalizálja az SGI algoritmus. Lineáris idejű algoritmus megvalósítható hash táblák segítségével, amelyek a szomszédsági adatokat tárolják, valamint prioritási sorok segítségével, amelyeket mindegyik új sáv kezdő háromszögének keresésére használunk fel. Többen is bebizonyították, hogy tetszőleges háromszöget választva az algoritmus során ugyan olyan jó eredményt kaphatunk. Így nem biztos, hogy megéri a jó kezdő háromszög kiválasztásával bajlódni. A 2-es lépéstől a 4-es lépésig biztosítva van az, hogy a kezdő háromszöget az aktuális háló leghosszabb sávjában megtaláljuk.

Egy praktikus szempont lehet az, hogy a háromszögek irányítottságát meg kellene őrizni azért, hogy helyes árnyalást és hátsólap eltávolítást kapjunk. Emlékezzünk vissza, hogy a háromszögsáv első háromszöge határozza meg a háromszögek körbejárását (lásd 8.2. ábrát). Egy példát láthatunk arra, hogy mi történik abban az esetben, ha a körbejárás megváltozik a kiterjesztés során, ahogy ez a 8.5. ábrán is látható.




*(a)*



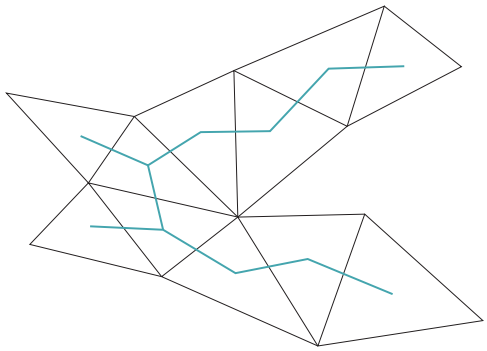

*(b)*

*8.5. ábra. Sáv kiterjesztés. (a) $T_3$ háromszöget (órajárással ellentétes körbejárású) választjuk kezdő háromszögként és a sáv jobbra terjeszkedik $T_4$ és $T_3$ háromszögeket magába foglalva. (b) A sávot balra terjesztettük ki, így növelve annak hosszát. Az eredmény sávban a $T_0\mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ háromszöget választjuk kezdő háromszögként. Emiatt a teljes háromszögsáv körbejárása megváltozik (órajárással megegyező lesz), mivel $T_0$ is ilyen irányítottságú. Ezt a problémát kikerülhetjük úgy, hogy az első vertexet megduplázzuk a sávban, amivel egy üres-területű háromszöget hozunk létre.*

*Duális gráf háromszögsáv-képző algoritmus*

Egy másik stratégia az, amikor a háromszög-háló duális gráfját használjuk. Ekkor a gráf élei a szomszédos lapok középpontjait összekötő élei lesznek (lásd 8.6. ábrát). Ezen élek gráfját *feszítőfának* nevezzük, amelyet azután jó háromszögsávok keresésére használhatunk fel.



*8.6. ábra. Háromszög-háló és annak duális gráfja*

*Pufferbarát háromszögsáv-képző algoritmus*

Tételezzük fel, hogy a háromszöghálókat háromszögsávokkal hoztuk létre. Mivel a grafikus kártyák többségének van vertex puffere (gyorsítótára), ezért a csővezetéken átküldendő sávok sorrendjei különböző vertex puffer teljesítményt fognak mutatni számunkra, mivel a gyorsítók különböző tárméretekkel rendelkeznek. Egy egyszerű előfeldolgozási művelettel javíthatjuk a sorrendet, amely a következőképpen néz ki:

- Vegyünk egy háromszögsávot, melynek a vertexeit a vertex gyorsítótár (FIFO) egy szoftveres szimulációjában helyezzük el.
- A fennmaradó háromszögsávok közül kiválasztjuk azt, amelyik a legjobban használja ki a tár tartalmát.
- Az eljárást addig ismételjük, amíg az összes háromszögsávot fel nem dolgoztuk.

Az NVIDIA NVTriStrip[4] függvénykönyvtár is pontosan ezeket a lépéseket követi.

**Háromszöghálók**

A háromszöghálók vertexek listájából és körvonalak halmazából állnak. Minden vertex pozíció és további adatokat tartalmaz, mint például diffúz és spekuláris színeket, árnyalási normálvektort, textúra-koordinátákat stb. Mindegyik háromszög körvonal egész értékű indexek listájával rendelkezik. Ezek az indexek a vertexekre mutatnak a listában.

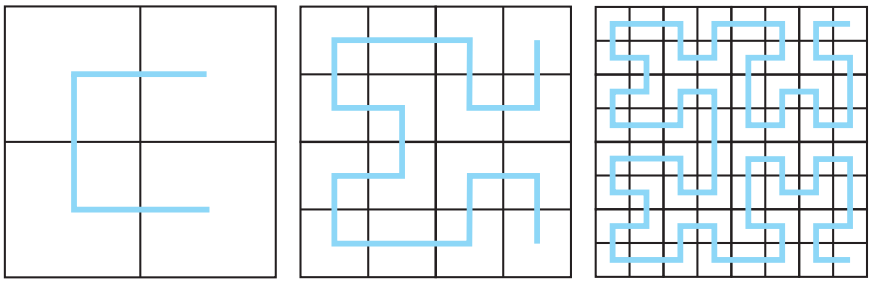
*Általános renderelési szekvenciák létrehozása*

Adott egy indexelt háromszög háló, amely hatékonyan renderelhető egy primitív vertex gyorsítótárral rendelkező grafikus hardver segítségével. A kérdés továbbra is az indexek

[4] `http://developer.nvidia.com/object/nvtristrip_library.html`

megfelelő sorrendjének meghatározása. Ráadásul, ahogy azt az imént már említettük, a különböző hardverek különböző méretű vertex gyorsítótárral rendelkeznek. Az egyik megoldás az, hogy mindegyik méretre más és más módon állítjuk elő az indexek sorrendjét. Természetesen ez nem a legszebb megoldás. Az igazi megoldás egy *univerzális* index sorozat előállítása, amely az összes lehetséges vertex gyorsítótár méret esetén jól viselkedik.

Mielőtt rátérnénk az univerzális algoritmus ismertetésére, szükség van a *terület-kitöltő görbe* fogalmának bevezetésére. A terület-kitöltő görbe egy egyszerű, folytonos görbe, amelyik nem metszi önmagát, kitölt egy olyan négyzetet vagy téglalapot, amely uniform négyzetrácsait csak egyszer érint annak bejárása során. A jó terület-kitöltő görbe jó térbeli összefüggőséggel rendelkezik, amely azt jelenti, hogy amikor bejárjuk azt mindig az előzőleg meglátogatott pontok közelében maradunk. A Hilbert görbe (lásd 8.7. ábrát) egy példa a terület-kitöltő görbére, amely rendelkezi a térbeli összefüggőség tulajdonsággal is.



*8.7. ábra. Első-, másod- és harmadrendű Hilbert görbe*

Ha ilyen terület-kitöltő görbét használunk a háromszögek bejárására a háromszög háló esetén, akkor a vertex gyorsítótárban nagy találati arányra számíthatunk. Egy rekurzív algoritmussal jó index sorozatot lehet létrehozni a háromszöghálókra. Az alap ötlet az, hogy szétvágjuk a hálót megközelítőleg két egyforma méretű hálóra, majd az egyik illetve a másik hálót rendereljük le. Ezt ismételjük rekurzívan minkét hálóra. A háló szétvágását előfeldolgozási lépésként hajtjuk végre kiegyensúlyozott él-vágás algoritmussal[5], amely minimalizálja az él vágások számát. Az algoritmus komplexitása lineáris a hálóban lévő háromszögek számára nézve. Az algoritmus a következőképpen néz ki:

1. Hozzuk létre a háromszögháló duális gráfját.
2. Ezután a kiegyensúlyozott él-vágás algoritmussal eltávolítjuk a minimális élek halmazát a duális gráfban azért, hogy a hálót két különálló, nagyjából megegyező méretű hálóra vágjuk szét.
3. A jó gyorsítótár teljesítmény eléréséhez ajánlott, hogy az utolsó háromszög az első hálóban közel legyen a második háló első háromszögéhez. Ezt elérhetjük azzal, hogy az első háló utolsó háromszöge és a második háló első háromszöge esetén megengedjük, hogy a duális gráfban egy élen osztozzanak, amelyet átvágtunk.

Ezt ismételve rekurzívan, az indexek sorozata előállítható. A renderelési sorrendet a rekurziós folyamat határozza meg.

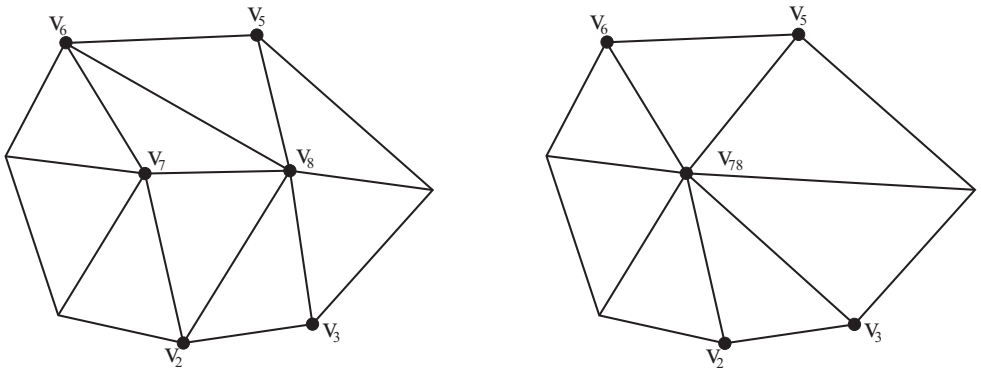
[5] A MeTiS szoftver csomaggal ez elvégezhető `http://www-users.cs.umn.edu/karypis/metis`

### 8.5.3. Háló egyszerűsítés

A *háló egyszerűsítéssel adat csökkentésként* vagy *decimálásként* is találkozhatunk, ami egy részletes modell poligonjainak a számának csökkentését jelenti oly módon, hogy megpróbálja megőrizni annak megjelenését. A valósidejű munka során ez az eljárás a csővezetéken átküldendő vertexek számát csökkenti, amely fontos lehet az adott alkalmazás régebbi számítógépeken való futtatásakor. Továbbá a modellek redundánsak lehetnek egy elfogadható megjelenítés esetén is. Három fajta poligon egyszerűsítési technikát különböztethetünk meg: statikus, dinamikus és nézőpont-függő. A statikus egyszerűsítéskor az alap ötlet az, hogy elkülönített részletességi szinteket[6] hozunk létre még a renderelés előtt és a megjelenítő választ ezek közül. A dinamikus egyszerűsítés egy folytonos spektruma az LOD modellek néhány diszkrét modelljével szemben. Ezért is nevezik azokat folytonos részletességiszint[7] algoritmusoknak. A nézőpont-függő technikákra jó példa az, amikor egy terep renderelésekor a közeli területeket részletesebben, míg a távolabbiakata távolságtól függően kisebb részletességgel jelenítjük meg.

**Dinamikus egyszerűsítés**

Az egyik módszer a poligonok számának csökkentésére az élek összevonása művelet. Ezt a műveletet két vertex egybe olvasztásával lehet elvégezni (lásd 8.8. ábrát). Ez a művelet két háromszöget, három élt és egy vertexet távolít el egy szolid modellből. Az Euler tétel szerint egy 3000 háromszögből álló modellen 1500 él összevonás műveletet alkalmazva nullára redukálja a lapok számát.




(a) (b)

*8.8. ábra. Él összevonás. Az (a) ábrán látható* $\mathbf{v}_6\mathbf{v}_7$ *és* $\mathbf{v}_6\mathbf{v}_8$ *illetve a* $\mathbf{v}_2\mathbf{v}_7$ *és* $\mathbf{v}_2\mathbf{v}_8$ *élek összevonásával a (b) ábrán látható módon eltűnik a* $\mathbf{v}_7\mathbf{v}_8$ *él és a* $\triangle\mathbf{v}_6\mathbf{v}_7\mathbf{v}_8$ *és* $\triangle\mathbf{v}_7\mathbf{v}_2\mathbf{v}_8$ *háromszögek.*

Az élek összevonása művelet visszafordítható. Az él összevonásokat rendezve, az egyszerűsített modellből kiindulva visszaállíthatjuk az eredeti modellt. Ez felhasználható például

[6] Level of Detail, röviden LOD
[7] Continous Level of Detail, röviden CLOD

a modellek hálózaton való továbbításakor, egyfajta tömörítési módszerként alkalmazva azt. Emiatt a tulajdonsága miatt, erre az egyszerűsítésre nézőpont független haladó hálózásként is hivatkoznak rá[8].

A $\mathbf{v}_7$ és $\mathbf{v}_8$ vertexeket összeolvasztottuk (8.8.a. ábra) $\mathbf{v}_{78} = \mathbf{v}_7$ vertexbe (8.8.b. ábra). Ugyanakkor a másik vertexbe való olvasztás is ($\mathbf{v}_{78} = \mathbf{v}_8$) elfogadható lett volna az egyszerűsítés során. Az egyszerűsítési rendszernek csak ez a két lehetősége van a *részhalmaz elhelyezési stratégia* használatakor. Az előnye ennek a stratégiának az, hogy ha korlátozzuk a lehetőségek számát, akkor értelemszerűen kódolhatjuk az aktuálisan végrehajtott választást. Mivel kevesebb esetet kell megvizsgálni, ezért ez a módszer gyorsabb, viszont alacsonyabb minőségű közelítést adhat, mivel kisebb megoldás teret jár be.

Az *optimális elhelyezés* eléréshez több lehetőséget kell megvizsgálni. Ahelyett, hogy az egyik vertexet a másikba olvasztanánk az élen lévő mindkét vertexet egy új pozícióban húzunk össze[9]. Ennek a technikának az előnye az, hogy egy jobb minőségű háló jön így létre. A hátránya az, hogy több műveletet kell végrehajtani és több memóriát is kell felhasználni a nagyobb területen való elhelyezési lehetőségek kiválasztására.

Bizonyos vertex összeolvasztásokat a költségekre való tekintet nélkül el kell kerülni. Ilyen esetek azok, amikor például konkáv alakzatok esetén a vertex összeolvasztás után egy új él a poligonon kívülre kerül és így elmetszi annak a határát. Ezt úgy lehet észlelni, hogy ellenőrizni kell, hogy vajon a szomszédos poligonok normálvektorának az iránya megfordul-e az összeolvasztás következtében.

A következőkben a Garland és Heckbert alap célfüggvényét mutatjuk be. Egy adott vertexhez megadhatjuk azon háromszögek halmazát, amelyeknek eleme ez a vertex és mindegyik háromszöghöz adott a sík egyenlete. A célfüggvény egy mozgó vertexre a síkok és az új pozíció távolságainak négyzet összegei, amely formálisan a következőképpen néz ki:

$$c(\mathbf{v}) = \sum_{i=1}^{m} (\mathbf{n}_i \cdot \mathbf{v} + d_i)^2, \tag{8.3}$$

ahol a $\mathbf{v}$ az új pozíció, az $\mathbf{n}$ az adott sík normálvektora és $d$ az eredeti pozícióhoz viszonyított eltolási értéke.

Ez a célfüggvény többféleképpen módosítható.

1. Képzeljünk el két háromszöget, amelyek egy közös éllel rendelkeznek. Ez az él egy nagyon éles él, amely pl. egy turbina lapát része lehet. A célfüggvény értéke a vertex összeolvasztás esetén ebben az esetben alacsony, mivel az egyik háromszögön csúszó pont nem kerül távol a másik háromszög síkjától. Az egyik módja az ilyen esetek kezelésének az, hogy egy olyan síkot veszünk hozzá az objektumhoz, amely tartalmazza az élet és az él normálvektora a két háromszög normálisának az átlaga. Így azoknál a vertexeknél, amelyek nagyon eltávolodnak ettől az éltől, nagyobb költség függvény értéket fogunk kapni.

2. A költség függvény másik fajta kiterjesztése másfajta felületi tulajdonságok megőrzésére szolgál. Például a modell gyűrődési és határ élei fontosak a megjelenítésben, ezért

[8] View-independent Progressive Meshing, röviden VIPM

[9] Ez történhet úgy, hogy az adott élen például a középpontban vagy azon bárhol helyezzük el az új pozíciót. Esetleg előfordulhat az is, hogy nem az élen találjuk meg az új pozícióját az összeolvasztott vertexeknek.

kisebb mértékben szabad csak azokat módosítani. Érdemes más felületi tulajdonság esetén is megőrizni a pozíciókat, ahol változik az anyag, textúra élek vannak és változik a vertexenkénti színek száma.

3. Leginkább azokat az éleket érdemes összevonni, amelyek a legkisebb észlelhető változást eredményezik. Lindstrom és Turk ötlete az volt, hogy kép-vezérelt függvényt használjunk ilyen esetek kezelésére. Az eredeti modellből különböző nézetből (mondjuk 20), legyártjuk a képeket. Ezután minden potenciális élre kipróbáljuk az összevonásokat az adott modell esetén és előállítjuk a képeket, amelyeket összehasonlítjuk az eredetivel. Azt az élet vonjuk össze, amelyik vizuálisan a legkisebb különbséget adja. Ennek a célfüggvény értékének a kiszámítása igen drága és természetesen nem valósítható meg valósidőben, bár az egyszerűsítési műveletet el lehet végezni előzetesen, amelyet később fel lehet használni.

Komoly problémát jelent az egyszerűsítési algoritmusoknál az, hogy gyakran a textúrák észrevehetően eltérnek az eredeti megjelenésüktől. Ahogy az élek eltűnnek, a felület mögött lévő textúrázási leképezés eltorzulhat.

A poligon csökkentési technikák hasznosak lehetnek, de nem szabad csodaszerként kezelni azokat. Egy tehetséges modellező létrehozhat egy alacsony poligon számú modellt, amely minőségben sokkal jobb, mint az automatikus eljárással előállított. Az egyik oka ennek az, hogy a legtöbb redukáló algoritmus nem tud semmit a vizuálisan fontos elemekről vagy a szimmetriáról. Például a szemek és az orr a legfontosabb része az arcnak. Egy naiv algoritmus elsimítja ezeket a területeket, mivel lényegtelenek.

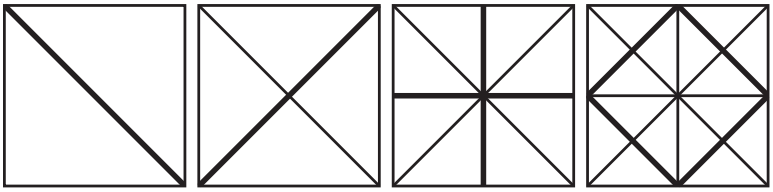
**Nézőpont-függő egyszerűsítés**

A terep az egyik olyan modell típus, amelyik egyedi tulajdonságokkal rendelkezik. Az adatokat rendszerint egyenletes rácson megadott magassági értékekként tárolják el. A nézőpont-függő módszerek általában valamilyen feltételben megadott kritérium határértékéig folytatják az egyszerűsítést. Szín vagy bump map textúrák segítségével lehetőség van kis méretű felületi részleteket ábrázolni. A külső területek esetén alkalmazni lehet azt a technikát, amikor a nézőponthoz közelebb lévő terepet nagyobb részletességgel ábrázoljuk.

Az egyik típusú algoritmus az élösszevonásos, amit az előző fejezetekben tárgyaltunk, kiegészítve egy célfüggvénnyel, ami a nézőpontot is figyelembe veszi. A terepet nem egy egyszerű hálóként kell ilyen esetben kezelni, hanem kisebb részterületekre bontva.

Az algoritmusok egy másik osztálya a magasságmező rácsból származtatott hierarchikus adatstruktúrát használ. Az alapötlet az, hogy egy hierarchikus struktúrát építünk fel az az adatok felhasználásával és kiértékeléskor pedig csak a bonyolultság szintjének megfelelően állítjuk elő a terep felszínt. Általánosan használt hierarchikus struktúra a bináris háromszögfa, amely egy nagy, jobb háromszöggel kezdődik a magasságmező darab sarkaiban lévő vertexekkel. Ez a háromszög felosztható az átfogón lévő középpont és a szemközti sarokpont összekötésével (lásd 8.9. ábrát). Ezt a felosztást addig folytathatjuk, amíg el nem érjük a magasságmező rácsának a részletességét.

Mindegyik háromszöghöz előállítunk egy hibahatárt. Ez a hibahatár fejezi ki azt a maximális mennyiséget, amivel a magasságmező eltérhet a háromszögekkel kialakított síktól. A hibahatár és a háromszög együtt határozzák meg egy torta-alakú szeletét a térnek, amely



*8.9. ábra. Bináris háromszögfa létrehozása. A baloldalon a magasságmező két háromszöggel van közelítve. A következő szinteken, mindegyik háromszög újból ketté van osztva. Az osztásokat a vastag szakaszok jelölik.*

tartalmazza a teljes terepet, amely kapcsolatban áll ezzel a háromszöggel. A futás alatt ezeket a hibahatárokat leképezzük a nézősíkra és kiértékeljük a megjelenítésre kifejtett hatásukat.

Mindegyik nézőpont-függő technika esetén a legjobb az, ha előre kiszámítjuk és eltároljuk a felület textúra térképén a megvilágítás hatását vagy egy elkülönített normál bump mapet használunk magasságmezők felületi normálvektorai számára. Ellenkező esetben a magasságmező részletesség szintjének a változásával, a megvilágítás és az interpolálás változása is határozottan észrevehető lesz.

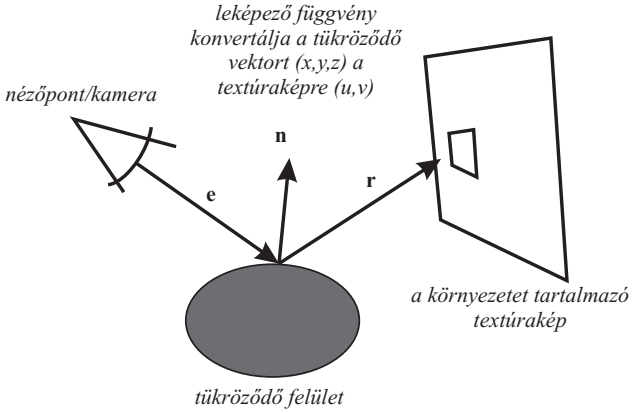
# 9. fejezet

# Realisztikus színtér

Ebben a fejezetben olyan módszereket mutatunk be a teljesség igénye nélkül, amelyek segítségével realisztikusabbá tudjuk tenni a színtereket.

## 9.1. Környezet leképezés

A *környezet leképezés* (Environment Mapping, röviden EM) egy egyszerű, mégis hatékony módszer görbe felületeken való tükröződés megjelenítésére. Mindegyik környezet leképezés módszer egy sugarat indít a nézőpontból a tükröződő objektum egy pontjába. Ez a sugár ezután a pontban lévő normálvektor alapján visszaverődik. Ahelyett, hogy megkeresnénk a legközelebbi felülettel való metszését, ahogy azt a sugárkövetés során tesszük, a környezet leképezés a visszavert fényvektornak az irányát használja a környezetet tartalmazó kép indexének a meghatározására (lásd 9.1. ábrát).




*9.1. ábra. Környezet leképezés. A kamera egy objektum felé néz. Az* $\mathbf{r}$ *visszavert fényvektorát az* $\mathbf{e}$ *és* $\mathbf{n}$ *vektorokból számítjuk ki. A visszavert fényvektor eléri a környezetet tartalmazó textúraképet. Az elérési információt a leképező függvény felhasználásával számítjuk ki, amely az* $(x, y, z)$ *visszatükröződő vektort alakítja át* $(u, v)$ *értékre.*

A környezet leképezés feltételezi, hogy az objektumok és fények, melyek a felületen

tükröződnek, messze vannak és a tükröződő felület önmagát nem tükrözi. Ha ezek a feltételezések igazak, akkor a tükröződő felületet körülvevő környezetet egy kétdimenziós leképezésként kezelhetjük.

A környezet leképezési algoritmus lépései a következőek:

- A környezetet ábrázoló kétdimenziós kép előállítása és betöltése.
- A tükröződő objektum mindegyik pixelére az objektum felületén lévő pozíciókban kiszámítjuk a normál egységvektorokat.
- A visszavert fényvektornak a kiszámítása a nézőpontvektor (nézőpont iránya) és a normál egységvektorból.
- A visszavert fényvektor segítségével meghatározzuk a környezeti térkép egy indexét, ami a környezet színe az objektum adott pontjában.
- A környezeti térképből kinyert texel adatokat használjuk fel az aktuális pixel színezésére.

A leképező függvények a visszavert fényvektort egy vagy több textúrára képezik le. Az egyik ilyen leképező függvény Blinn és Newell módszere.

### 9.1.1. Blinn és Newell módszere

Mindegyik leképezett pixelre kiszámítjuk a visszavert fényvektort és $(\rho, \phi)$ gömbi koordinátákba transzformáljuk azokat. A $\phi \in [0, 2\pi]$-t hosszúsági körnek, $\rho \in [0, \pi]$-t szélességi körnek nevezzük. $(\rho, \phi)$-t a következő összefüggések alapján számítjuk ki:

$$\begin{aligned} \rho &= \arccos(-r_z) \\ \phi &= \arctan(\frac{r_y}{r_x}), \text{ ha } r_x \neq 0, \end{aligned} \tag{9.1}$$

ahol $\mathbf{r} = (r_x, r_y, r_z)$ a normalizált visszavert fényvektor. A nézőponthoz tartozó visszavert fényvektort, hasonlóan számítjuk a fény tükröződési vektoréhoz:

$$\mathbf{r} = \mathbf{e} - 2(\mathbf{n} \cdot \mathbf{e})\mathbf{n}, \tag{9.2}$$

ahol $\mathbf{e}$ a normalizált vektor a felület pozícióban és $\mathbf{n}$ az egység normálvektor az adott pozícióban.

A $(\rho, \phi)$ gömbi koordinátákat a $[0, 1)$ tartományra képezzük le és $(u, v)$ koordinátaként használjuk a környezet textúra eléréséhez, a tükröződő szín előállítására. Mivel a tükröződési vektort transzformáljuk gömbi koordinátákba, így a környezetet tartalmazó textúrakép egy „kiterített” gömb képe. Lényegében a textúra befed egy gömböt, ami körbeveszi a tükröződési pontot. Ezt a leképező függvényt néha szélességi-hosszúsági leképezésnek is hívják, mivel $v$ a szélességi körökkel, $u$ pedig a hosszúsági körökkel egyezik meg.

Annak ellenére, hogy könnyű megvalósítani ezt a módszert, a módszernek van néhány hátránya. Először is $\phi = 0$-ban van egy határ, másodszor a térkép összefut a sarkoknál. A környezet leképezésben használt képnek egyeznie kell a szegélyeknél a függőleges élek mentén és el kell kerülni a torzítási problémákat a felső és alsó élek környezetében.

A valósidejű grafikai alkalmazásban a 9.1. egyenletet használhatjuk az indexek kiszámítására a vertexekben és ezután interpolálhatjuk ezeket a koordinátákat. Hiba fordul elő abban az esetben is, amikor egy háromszög vertexei olyan indexekkel rendelkeznek a környezeti térképen, melyek a sarkokon mennek keresztül.

Ezt a módszert nem alkalmazzák gyakran, mint környezet leképezési technikát. Csak történeti okokból ismertettük és azért, mert a gömbi leképező függvényt általában gyakran használják a textúra leképezésben.

### 9.1.2. Cube map környezet leképezés

A cube map környezeti térképet úgy kapjuk, hogy a kamerát egy kocka középpontjában helyezzük el és levetítjük a környezetet a kocka oldalaira. A gyakorlatban a színteret hatszor rendereljük le úgy, hogy a kamerát a kocka középpontjában helyezzük el. Nagy előnye ennek a módszernek, hogy a környezeti térképet bármely renderelővel könnyen elő lehet állítani valósidőben.

A visszavert fényvektornak az iránya meghatározza, hogy a kocka melyik oldalát használjuk. A visszavert fényvektor abszolút értékben legnagyobb komponense meghatározza, hogy milyen kapcsolatban van az oldallal (pl. $(-3.2, 5.1, -8.4)$ a $-Z$ oldalt jelöli ki). A maradék két komponenst a legnagyobb komponens abszolút értékével elosztva, majd a $[0, 1]$ intervallumra leképezve kapjuk meg a textúra-koordinátákat a kiválasztott lapon.

### 9.1.3. Sphere map környezet leképezés

Ebben az esetben a textúraképet egy tökéletesen tükröződő gömbön megjelenő környezet ortogonális nézetéből állítjuk elő, így ezt a textúrát gömbtérképnek nevezzük. Egyik lehetőség egy ilyen gömbtérkép előállítására az, hogy egy csillogó gömbről készítünk fényképet. Ezt az kör alakú eredmény gömbtérképet néha fényvizsgálatnak is hívják, ahogy a megvilágítás hatását visszaadja a gömb pozíciójában. A gömbtérképet szintetikus színtér esetén vagy sugárkövetéssel vagy pedig a cube map környezeti térképnél használt képek gömbre való vetítésével lehet előállítani.

A gömbtérképnek van egy bázisa. A képet egy $\mathbf{f}$ tengely mentén nézzük a világtérben $\mathbf{u}$ felfele mutató vektorral és feltesszük, hogy a $\mathbf{h}$ vektor vízszintesen jobbra mutat (mindegyik vektor normalizált). Ez egy bázis mátrixot ad:

$$\begin{pmatrix} h_x & h_y & h_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{9.3}$$

A gömbtérkép egy elemének eléréséhez először az $\mathbf{n}$ felületi normált és a szem pozíciójából a vertexbe menő $\mathbf{e}$ vektort transzformáljuk. Ez az $\mathbf{n}'$ és $\mathbf{e}'$ vektorokat állítja elő a

gömbtérkép terében. A visszavert fényvektort a következőképpen állítjuk elő:

$$\mathbf{r} = \mathbf{e}' - 2(\mathbf{n}' \cdot \mathbf{e}')\mathbf{n}', \tag{9.4}$$

ahol $\mathbf{r}$ eredmény vektor a gömbtérkép terében van.

A tükröződő gömb a teljes környezetet mutatja meg, ami a gömb előtt található. Ez mindegyik visszavert irányt leképezi a gömb kétdimenziós képének egy pontjára.

Ha meg akarjuk határozni a tükröződési irányt a gömbtérkép egy adott pontjában, akkor szükségünk van a gömb pontjában a felületi normálvektorra. Fordítsuk meg az eljárást és vegyük a gömbön a pozíciót és vezessük le a felületi normált a gömbön, ami az $(u, v)$ paramétereket határozza meg a textúra adatok eléréséhez.

A gömb normálvektora $(r_x, r_y, r_z)$ a visszavert fényvektor és a szem iránya $(0, 0, 1)$ között fél úton található. Az $\mathbf{n}$ normálvektor egyszerűen felírható a szem és a visszavert fényvektor összegeként, amelyet ezután normalizálunk:

$$\begin{aligned} m &= \sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}, \\ \mathbf{n} &= \left( \frac{r_x}{m}, \frac{r_y}{m}, \frac{r_z + 1}{m} \right). \end{aligned} \tag{9.5}$$

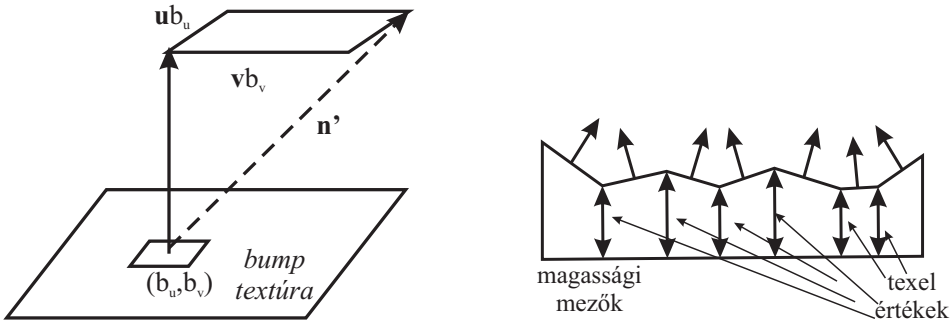
A gömb leképezés egyik hátránya az, hogy a gömbtérképen két pont közötti mozgás nem lineáris. Ráadásul a gömbtérkép csak egyetlen nézőpont irány esetén érvényes. Így ha változik a nézőpont iránya, akkor a leképezést újra végre kell hajtani. Továbbá, mivel a gömbtérkép nem tartalmazza a teljes környezetet, így előfordulhat, hogy képkockáról-képkockára ki kell számolni a környezeti leképezés textrúra-koordinátáit az új nézőpont irányra az alkalmazás szakaszban[1]. Így, amennyiben a nézőpont iránya változik, akkor érdemesebb nézőpont független környezeti leképezést használni.

## 9.2. Felületi egyenetlenség leképezés

A felületi egyenetlenség leképezés (angolul bump mapping) egy olyan technika, amely a felületek megjelenését teszi egyenetlenné. Ez a leképezés olyan tulajdonságot szimulálhat, amit ellenkező esetben sok poligon felhasználásával lehet csak modellezni. Az alap ötlet az, hogy a textúra nem a szín komponenst változtatja meg a megvilágítási egyenletben, hanem a felületi normálvektorokat módosítja, amelyeket egy textúrában tárolunk el. A felület geometria normálja változatlan marad, csupán a megvilágítási egyenletben használt normálvektorokat változtatjuk meg pixelenként.

Az egyik felületi egyenetlenség textúrázási technika esetén $b_u$ és $b_v$ előjeles értékeket tárolnak el egy textúrában. Ez a két érték a normál változásának a mennyiségét tárolja $u$ és $v$ tengelyek mentén. Ezeket a textúra értékeket használjuk a normálisra merőleges két vektor skálázására, mely textúra értékek általában bilineárisan interpoláltak. A két $b_u$ és $b_v$ adja meg, hogy a felület milyen irányba néz a pontban (lásd 9.2.(a). ábrát).

[1] Vizuális artifaktumok jelenhetnek meg, úgy mint a gömbtérkép néhány része megnagyobbodik és a szingularitás is problémát okozhat.




*(a) Az* $\mathbf{n}$ *normálvektort az* $\mathbf{u}$ *és* $\mathbf{v}$ *irányokban* $(b_u, b_v)$ *értékekkel, ami egy* $\mathbf{n}'$ *nem normalizált vektort ad*

*(b) Magassági mezők és azok hatása az árnyalási normálvektorokra*

*9.2. ábra. Felületi egyenetlenség leképezési technikák*

Egy másik módja a felületi egyenetlenség leképezés megvalósítása esetén magassági mezőket használunk a felületi normálvektorok irányainak a módosítására (lásd 9.2.(b). ábrát). Mindegyik szürkeárnyalatos textúraérték egy magasság értéket jelent, ahol egy fehér texel egy magas területet, egy fekete texel pedig alacsony területet jelent. Ez egy gyakori formátum felületi egyenetlenség térkép előállításakor vagy szkennelésekor. A szomszédos oszlopok különbsége adja meg az $u$, valamint a szomszédos sorok különbsége a $v$ meredekségét.

A pixelenkénti felületi egyenetlenség leképezés meggyőző és olcsó módja annak, hogy a geometriai részletesség látszatát növeljük. Az objektumok körvonalai körül azonban a hatás eltűnik. Ezeknél az éleknél a szemlélő azt veszi észre, hogy nincsenek valódi egyenetlenségek, csak sima kontúrok. Egy másik probléma az, hogy a felületi egyenetlenség alkalmazásakor az egyenetlenségek nem vetnek árnyékot a saját felületükön, ami nem felel meg a valóságnak. Természetesen léteznek fejlettebb valósidejű renderelő módszerek, melyek használatával önárnyalási hatást is el lehet érni.

Statikus színterek esetén a megvilágítást előre is ki lehet/kell számítani. Például, ha egy felületen nincs spekuláris megvilágítás és a fények nem mozognak a felülethez viszonyítva, akkor a felületi egyenetlenséghez tartozó árnyalást ki lehet számítani egyszer és az eredményt egy szín textúraként használjuk ezen a felületen. Hasonlóan, ha a felület, fény és kamera mindegyike rögzítve vannak egymáshoz, akkor fényes egyenetlen felületet elegendő egyszer előállítani.

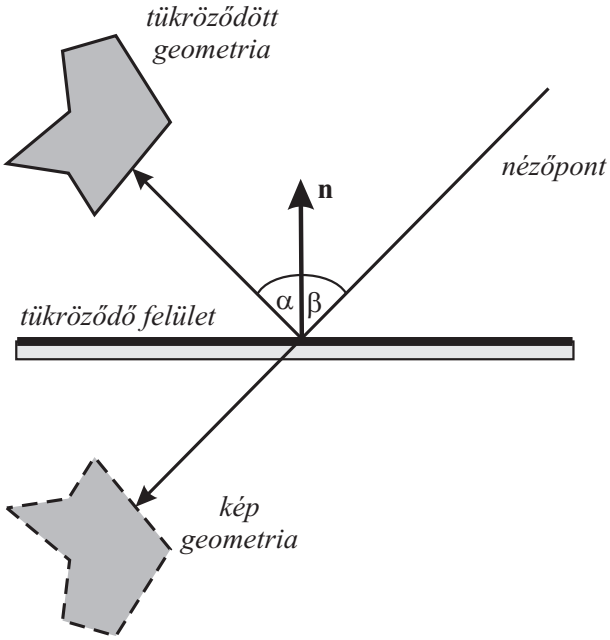
## 9.3. Tükröződések

A tükröződés, fénytörés és árnyék mindegyike a globális megvilágítás hatásaira példa, ahol egy objektum egy másik objektum előállítását befolyásolja a színtéren. Ezek a hatások nagyban növelik az előállított kép valósághűségét, ugyanakkor a nézőnek egy támpontot ad a térbeli kapcsolatok meghatározásában.

### 9.3.1. Sík tükröződés

A sík tükröződést könnyebb megvalósítani és végrehajtani, mint egy általános tükröződést. Ebben az esetben egy sík felületen való tükröződést értünk, mint például egy tükör.

Egy ideális tükröződő felületre érvényes a *tükröződési törvény*, amely szerint a beesési szög megegyezik a visszavert fény kilépési szögével. Ennek a törvénynek köszönhetően az objektum tükrözött képe egyszerűen maga a tükrözött objektum (lásd 9.3. ábrát).




*9.3. ábra. Sík tükröződés, ahol $\alpha$ a visszavert fény kilépési szöge és $\beta$ a fény beesési szöge ($\alpha = \beta$)*

A visszatükrözött sugár követése helyett a beeső fényt követhetjük a tükröződő felületen. Azt a következtetést vonhatjuk le ebből, hogy egy tükröződést előállíthatunk egy objektum másolatának a transzformálásával a tükröződő pozícióba. Ahhoz, hogy helyes eredményt érjünk el a pozíció és az irány figyelembevételével a fényforrásokat is tükrözni kell.

Ha feltesszük, hogy a tükröződő felület $\mathbf{n}$ normálvektora $(0, 1, 0)$ és ez az origón megy keresztül, akkor a mátrix, ami erre a síkra tükröz egy egyszerű tükröző $\mathbf{S}(1, -1, 1)$ skálázó mátrix. Általános esetre az $\mathbf{M}$ tükröződési mátrixot egy $\mathbf{n}$ normálvektort és a tükröződő felület $\mathbf{p}$ pontját felhasználva vezetjük le. Mivel már ismerjük, hogy hogyan kell tükrözni az $y = 0$ síkra nézve, ezért először a síkot az $y = 0$ síkba transzformáljuk, ahol elvégezzük az egyszerű skálázást, majd a végén visszatranszformáljuk a síkot az eredeti pozíciójába. Ezen mátrixok összefűzésével kapjuk meg a $\mathbf{M}$ mátrixot.

Először a síkot eltoljuk a $\mathbf{T}(-\mathbf{p})$ transzformációval úgy, hogy az origón keresztül menjen. Ezután a tükröződő felület $\mathbf{n}$ normálvektorát forgatjuk, hogy párhuzamos legyen az $(0, 1, 0)$ $y$-tengellyel. Ezt a forgatást az $\mathbf{R}(\mathbf{n}, (0, 1, 0)$ (lásd 3.67. egyenletet) felhasználásával hajtjuk végre. Ezeknek a transzformációknak az összefűzésével kapjuk a következő összefüggést:

$$\mathbf{F} = \mathbf{R}(\mathbf{n}, (0,1,0))\mathbf{T}(-\mathbf{p}). \tag{9.6}$$

A tükröződő felület így az $y = 0$ síkhoz lesz igazítva. Ezután az $\mathbf{S}(1,-1,1)$ skálázást hajtjuk végre, majd visszatranszformáljuk az $\mathbf{F}^{-1}$-vel. Így az $\mathbf{M}$-t a következő módon állítjuk össze:

$$\mathbf{M} = \mathbf{F}^{-1}\mathbf{S}(1,-1,1)\mathbf{F}. \tag{9.7}$$

Megjegyezzük, hogy ezt a mátrixot újra kell számolni, ha a pozíció vagy a tükröződő felület irányítottsága megváltozik.

Először a megjelenítendő színtér $\mathbf{M}$-mel transzformált tükröződő objektumait, majd a színtér többi részét rajzoljuk ki a tükröződő felülettel együtt. A tükröződő felületnek részlegesen átlátszónak kell lenni azért, hogy a tükröződés látható legyen. Amennyiben éles szögben nézünk rá az adott színtérre, akkor a tükröződő geometria láthatóvá válhat. A „kilógó ” rész eldobásával[2] ez a probléma megoldható.

Egy másik probléma a hátsólap-eldobás miatt fordul elő. Amennyiben a hátsólap-eldobás be van kapcsolva és egy objektumot skálázunk a tükröződési mátrixszal, akkor a hátsólap-eldobás helyett az előlapok lesznek eldobva. A megoldás az, hogy a hátsólap-eldobásból az előlap-eldobásra váltunk.

### 9.3.2. Fénytörések

Több fizikai törvényt is figyelembe kell vennünk a fénytörés szimulálásakor. Az egyik ilyen tényező a Snell törvény, amely a beérkező és kilépő vektorok kapcsolatát állapítja meg, amikor egyik közegből (mint például levegő) a másikba (például a víz) lép a fény:

$$n_1 \sin(\theta_1) = n_2 \sin(\theta_2), \tag{9.8}$$

ahol az $n_k$ az adott közeg törésmutatója és $\theta_k$ ($k \in \{1,2\}$) a felületi normálishoz viszonyított szög (lásd 9.4. ábrát).

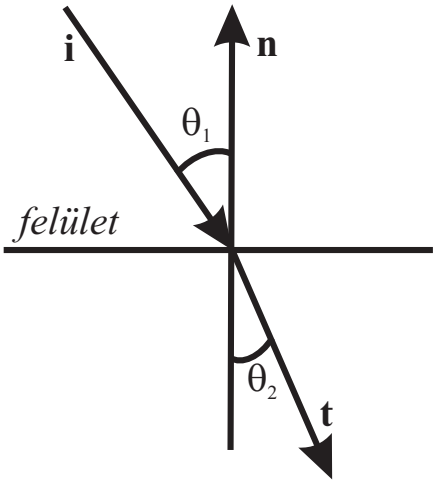
Az $\mathbf{i}$-vel a bejövő vektort és $\mathbf{n}$-nel a felületi normálvektort jelöljük, ahol mindegyik normalizálva van. A normalizált $\mathbf{t}$ fénytörés vektor kiszámítására a következő képletet használhatjuk:

$$\mathbf{t} = r\mathbf{i} + (w - k)\mathbf{n}, \tag{9.9}$$

ahol

$$\begin{aligned} r &= n_1/n_2, \\ w &= -(\mathbf{i}\cdot\mathbf{n})r, \\ k &= \sqrt{1 + (w-r)(w+r)}. \end{aligned} \tag{9.10}$$

[2] A legalkalmasabb ennek a problémának a megoldására a *stencil puffer* használata.




*9.4. ábra. Snell törvénye. A fény az egyik közegből a másikba haladva megtörik az adott közegek törésmutatójától függően. A beesési szög akkor nagyobb vagy egyenlő törési szögnél, ha egy alacsonyabb törésmutatójú közegből magasabb törésmutatójú közeg felé haladunk.*

Ez a kiértékelés eléggé költséges, mivel a fénytörés mértéke a horizont környékén csökken. Kis bejövő szögek esetén a következő közelítést használhatjuk:

$$\mathbf{t} = -c\mathbf{n} + \mathbf{i}, \tag{9.11}$$

$c$ víz szimulálása esetén 1.0 körül van. Ebben az esetben $\mathbf{t}$ vektort normalizálni kell.

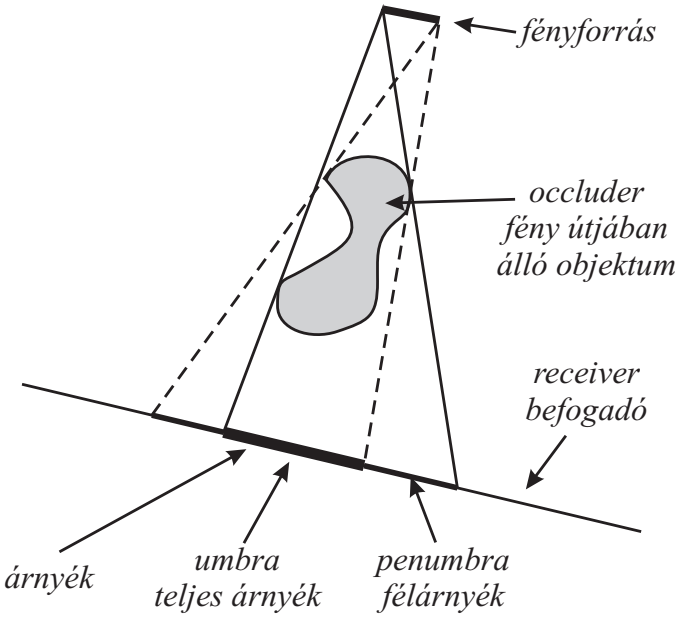
## 9.4. Árnyék síkfelületen

Az árnyékok fontos elemei a valósághű képek előállításánál és a felhasználóknak az objektum elhelyezéséről adnak némi információt. Az árnyékokkal kapcsolatos terminológiákat a 9.5. ábrán láthatjuk, ahol a *fény útjában álló objektum* árnyékot vet a *befogadó* felületre. A pont fényforrás teljesen árnyékolt régiókat állít elő, amelyeket néha *éles árnyékoknak* neveznek. Amennyiben területi vagy térfogati fényforrásokat használnak, akkor sima árnyékok keletkeznek. Ekkor mindegyik árnyéknak van egy *teljesen árnyékolt* területe (umbra) és egy *részlegesen/félig árnyékolt* régiója (penumbra). A sima árnyékokat az árnyék sima éleiről lehet felismerni. Fontos megjegyeznünk, hogy az éles árnyék éleinek egy alul-áteresztő szűrővel való elsimításával nem lehet helyesen előállítani a sima árnyékokat.

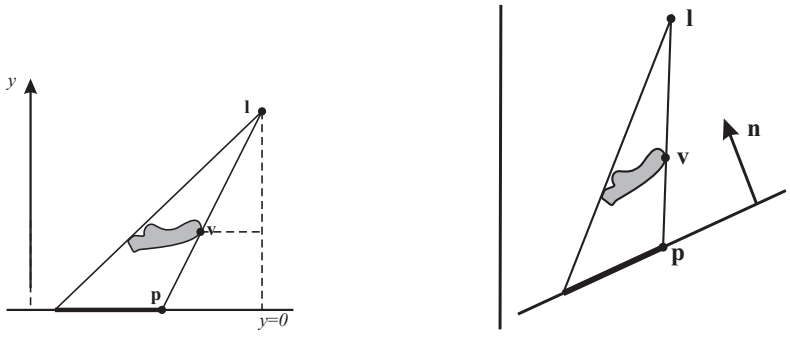
Egy egyszerű esete az árnyalásnak az, amikor az objektumok árnyékai egy sík felületen jelennek meg.

### 9.4.1. Vetített árnyék

Ebben az esetben egy mátrixot hozunk létre, amely az objektum vertexeit vetíti le egy síkra és ezután az árnyék előállításakor a háromdimenziós objektumot másodszor is megjelenítjük. Tételezzük fel a 9.6. ábrán lévő helyzetet.




*9.5. ábra. Árnyékkal kapcsolatos elnevezések*




*(a)* $y = 0$ *síkra vetett árnyék*

*(b)* $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ *síkra vetett árnyék*

*9.6. ábra. Vetített árnyék sík felületen. A fényforrás az* $\mathbf{l}$ *pontban található. A* $\mathbf{v}$ *pontot vetítjük, melynek a képe az adott síkon* $\mathbf{p}$ *pont.*

Az $x$ koordináták vetítésével kezdjük a levezetést. A 9.6.(a). ábrán látható hasonló háromszögek alapján a következő egyenlőségeket írhatjuk fel:

$$\begin{aligned} \frac{p_x - l_x}{v_x - l_x} &= \frac{l_y}{l_y - v_y} \\ &\Longleftrightarrow \\ p_x &= \frac{l_y v_x - l_x v_y}{l_y - v_y} \end{aligned} \tag{9.12}$$

a $z$ koordinátát hasonlóan kapjuk: $p_z = (l_y v_z - l_z v_y)/(l_y - v_y)$, míg az $y$ koordináta 0-val egyenlő. Az összefüggések alapján az $\mathbf{M}$ projekciós mátrixot a következőképpen írhatjuk fel:

$$\mathbf{M} = \begin{pmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{pmatrix}. \tag{9.13}$$

Könnyű belátni, hogy $\mathbf{Mv} = \mathbf{p}$, ami azt jelenti, hogy $\mathbf{M}$ valóban a megfelelő projekciós mátrix.

Általános esetben a sík egyenlete, amelyikre az árnyék vetődni fog $\pi : \mathbf{n} \cdot \mathbf{x} + d = 0$ (lásd 9.6.(b). ábrát). A cél az, hogy előállítsunk egy mátrixot, amely $\mathbf{v}$ pontot vetíti le $\mathbf{p}$ pontba. Az $\mathbf{l}$ pontból induló sugár, amely a $\mathbf{v}$ ponton keresztül megy és elmetszi a $\pi$ síkot. Ebből következik, hogy a $\mathbf{p}$ pontot a következőképpen lehet előállítani:

$$\mathbf{p} = \mathbf{l} - \frac{d + \mathbf{n} \cdot \mathbf{l}}{\mathbf{n} \cdot (\mathbf{v} - \mathbf{l})}(\mathbf{v} - \mathbf{l}). \tag{9.14}$$

Mátrix alakba felírva a 9.14. egyenletet kapjuk, hogy

$$\mathbf{M} = \begin{pmatrix} \mathbf{n} \cdot \mathbf{l} + d - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & \mathbf{n} \cdot \mathbf{l} + d - l_y n_y & -l_y n_z & -l_y d \\ -l_z n_x & -l_z n_y & \mathbf{n} \cdot \mathbf{l} + d - l_z n_z & -l_z d \\ -n_x & -n_y & -n_z & \mathbf{n} \cdot \mathbf{l} \end{pmatrix}. \tag{9.15}$$

Az általános mátrixba behelyettesítve (9.15. egyenlet) az $y = 0$ síkhoz tartozó speciális értékeket ($\mathbf{n} = (0, 1, 0)^T$ és $d = 0$) speciális $\mathbf{M}$ mátrixot (9.13. egyenlet) kapunk.

Az árnyék előállításához egyszerűen ezt a mátrixot kell alkalmaznunk az objektumokra, amelyek árnyékot vetnek a $\pi$ síkra. Az így kapott vetületeket sötét színnel és megvilágítás nélkül kell megjeleníteni. Megjegyezzük, hogy lényegében a fény útjában álló objektumot kétszer rendereljük, először a vetített poligonokat árnyékként, másodszor pedig az eredeti objektumként.

A gyakorlatban szükség van egy olyan módszerre, amely megakadályozza azt, hogy a vetített poligonokat a befogadó felület mögött állítsuk elő. Egy biztonságos módszer lehet

az, amikor a talajt rajzoljuk ki először, aztán a vetített poligonokat kikapcsolt $Z$-puffer ellenőrzéssel és azután az összes többi geometriát. Így a vetített poligonok mindig a talajon jelennek meg, mivel nem történik mélység ellenőrzés.

Hasonló hiba előfordulhat az árnyék képzéskor, amivel a tükröződéskor is találkoztunk. A vetített árnyék a síkon kívül is megjelenhet. A megoldása is hasonló az ott alkalmazott módszerrel, el kell távolítani a kilógó részt.

A hátránya a vetítési modellnek amellett, hogy csak sík felületek esetén működik az, hogy az árnyékot mindegyik képkocka esetén elő kell állítani, még akkor is, ha az árnyék nem változik. Mivel az árnyékok függetlenek a nézőponttól (nem változik az alakjuk a nézőpont változásával)., A gyakorlatban egy jól működő módszer az, amikor az árnyékot egy textúrában állítjuk elő, amit aztán egy textúrázott téglalapként jelenítünk meg. Az árnyéktextúrát csak akkor kell újra kiszámítani, ha az árnyék megváltozik, vagyis amikor a fényforrás vagy a fény útjában álló objektum vagy a befogadó felület mozog.

A mátrixok nem mindig állítják elő a megfelelő eredményt, például, amikor a fényforrás alatta van az objektum legfelső pontjának. Hasonló renderelési hiba fordul elő sík tükröződés esetén, amikor az objektumok a tükröződő felület másik oldalán találhatóak. Az árnyék előállításának az esetében akkor fordul elő hiba, ha egy árnyékot vető objektum a távolabbi oldalán található a befogadó felületnek. Az ilyen módon generált árnyékokat *hamis árnyékoknak* nevezzük. Ennek elkerülésére a befogadó síkot kell használnunk az árnyékoló objektumok levágásához és eldobásához, mielőtt a vetített árnyék poligonjait előállítjuk. Mivel a vágást a vetítés előtt kell végrehajtani, ezért ezt a műveletet még az alkalmazás oldalon végre kell hajtani.

# Irodalomjegyzék


[1] Tomas Akenine-Moller and Eric Haines. *Real-Time Rendering (2nd Edition)*. A K Peters/CRC Press, July 2002.

[2] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, March 2003.

[3] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, Second Edition*. CHARLES RIVER MEDIA, 2004.

[4] Richard S. Wright, Jr. Benjamin Lipchak, and Nicholas Haemel. *OpenGL SUPERBIBLE, Fourth Edition, Comprehensive Tutorial and Reference*. Addison-Wesley Professional, June 2007.