Debreceni Egyetem
Informatikai Kar

Nem fotorealisztikus megjelenítés OpenGL segítségével

Témavezető:
Dr. Tornai Róbert
egyetemi adjunktus

Készítette:
Sajtos István
programtervező matematikus

Debrecen
2010

Tartalomjegyzék

Köszönetnyilvánítás

Szeretnék köszönetet mondani témavezetőmnek, Dr. Tornai Róbertnek hasznos szakmai tanácsaiért, és páromnak, aki mindenben támogatott és segített a diplomamunka készítése alatt, és aki nélkül ez a munka nem készülhetett volna el.

1. Bevezetés

A jelenlegi grafikus hardverek képességei messze túlmutatnak a húsz évvel ezelőtti jóslatokon. Napjaink számítógépeit a tökéletes valósághűség szimulálására, modellezésére próbálják kihegyezni. Gondoljunk csak a robotikában is alkalmazott mesterséges intelligencia törekvésekre, képfeldolgozó algoritmusokra, digitális beszédfeldolgozásra.

A szimulált valóság előállítása a komputergrafikában is jelenlévő igény, nem is lehet ez másként, hisz ez az igény a fejlesztések fő mozgatórugója. Az utóbbi években azonban egy új irányvonal megerősödése is elkezdődött, a valósidejű, háromdimenziós, nem fotorealisztikus ábrázolásé. Nem fotorealisztikus vagy nem valószerű ábrázolás alatt azokat a megjelenítési technikákat értjük, amelyek klasszikus művészi rajzokhoz, illusztrációkhoz hasonló eredményt szolgáltatnak. Ilyen eredmények lehetnek egy ceruzarajz, festmény, rajzfilmszerű ábrázolás, stb. Látni fogjuk, hogy a lehetőségek száma igen széles, rengeteg jól alkalmazható módszer született már a témában, és jelenleg is kutatások folynak az eddig meg nem valósított ábrázolási formákat illetően. A valósidejű arra utal, hogy ezek a programok interaktívak lehetnek, tehát ezek a megjelenítési technikák jól alkalmazhatóak pl. videojátékokban, CAD rendszerekben.

A nem fotorealisztikus ábrázolást választottam diplomadolgozatom témájául, mert kevésbé kedvelt és ismert, mint a fotorealisztikus ábrázolás, mégis rengeteg lehetőséget rejt, és napjainkban egyre szélesebb körben alkalmazzák.

A dolgozattal kitűzött célom látványos, jól alkalmazható technikák ismertetése mind elméletben mind pedig gyakorlatban, és azok összehasonlítása a fotorealisztikus módszerekkel megvalósítás és teljesítmény szempontjából. Célom továbbá az ismertetett módszerekhez valós idejű per-fragment implementáció készítése (már ahol ez lehetséges), mivel feltevésem az, hogy a mai modern grafikus hardverek kellő támogatást nyújtanak ezek megfelelő futtatásához. Az implementáció egy fps-számlálót is megvalósít, amely leméri, hogy átlagosan hány frame-et produkál a program másodpercenként.

Mivel számos említésre méltó módszer van, ezért a tárgyalt módszerek kiválasztása részben szubjektív véleményen alapult, a fő cél mégis vizuálisan megnyerő módszerek ismertetése. Ezen technikák megismerése jó kiindulópont azoknak, akik nem fotorealisztikus megjelenítéssel kívánnak foglalkozni, de azoknak is hasznos lehet, akik csak betekintést szeretnének nyerni a nem fotorealisztikus ábrázolás világába.

Az implementáció elkészítéséhez az OpenGL API-t választottam, amelyről rövid áttekintést adok az implementáció tárgyalása előtt a 2. fejezetben. A 3. fejezet a megvilágítási számításainkhoz szükséges elméleti alapokat adja meg. A 4. fejezet a fotorealisztikus módszerek megismerésébe vezet be. A 5. fejezet alfejezetekre bontva tartalmazza a nem fotorealisztikus módszerek ismertetését. Minden egyes alfejezet az abban tárgyalt módszer elméleti áttekintésével indul, a fejezetek végén a módszert implementáló shader-ek ismertetése történik. A 6. fejezetben a korábban megismert módszerek teljesítményvizsgálatával foglalkozom. A 7. fejezet a program rövid ismertetését tartalmazza.

2. Az OpenGL és a GLSL

A szabványos interfészek megjelenése előtt a hordozható grafikus alkalmazások fejlesztése gondot okozott. A problémát az jelentette, hogy a gyártók által kínált grafikus hardverek architektúrája és a hozzájuk tartozó grafikus könyvtárak nem voltak szabványosítva.

A problémára megoldást az 1992-ben specifikált OpenGL (Open Graphics Library) jelentett, amely egy hardverfüggetlen grafikus könyvtár. Az OpenGL tulajdonképpen egy ipari szabvány, amelyet a Silicon Graphics más, hardver- és szoftverfejlesztő nagyvállalatokkal együttműködve alkotott meg.

Az OpenGL háromdimenziós alkalmazások fejlesztésére alkalmas API (alkalmazásprogramozói interfész), amely könnyen használható, és széles körben támogatott. Legnagyobb előnye a konkurens DirectX-el szemben a platformfüggetlensége. Az OpenGL fejlesztését az OpenGL ARB (Architecture Review Board)[1] felügyeli, amelyet a Silicon Graphics alapított.

1992 óta számos OpenGL verzió jelent meg. A 2.0-ás verzió megjelenése előtt a gyártók által definiált kiterjesztések szolgáltak egyedüli lehetőségként komolyabb módosítások beépítésére. A programozható hardver megjelenésével azonban mindez kevéssé vált, ezért megalkották az OpenGL Shadig Language-t (GLSL[2]). A GLSL az OpenGL saját programozási nyelve shader programok írásához. A GLSL 1.5-ös verziója a fragment és vertex shader-ek mellé bevezette geometriai shader-eket is.

Jelenleg a 4.0-ás verziónál tart az OpenGL szabvány, amely a GLSL 4.00-ás verzióját tartalmazza.

2.1. A shader-ek jelentősége

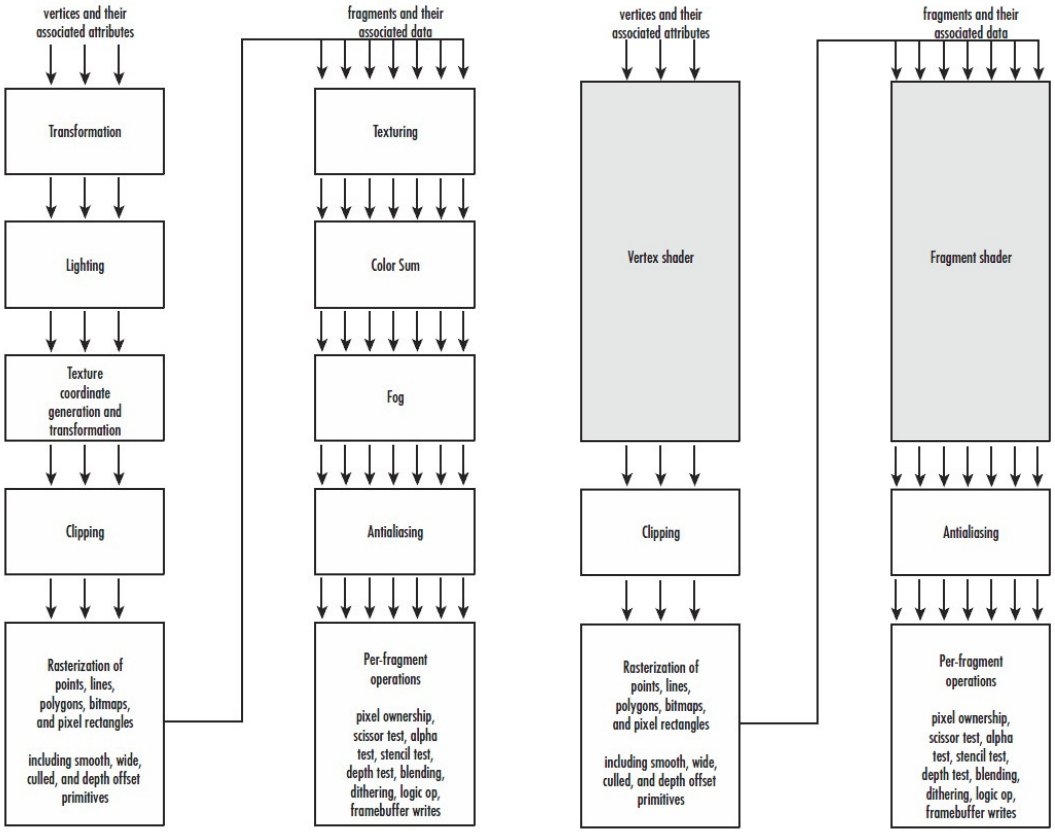
A kezdeti grafikus hardvereket úgy tervezték, hogy ugyanazon kötött számítások végrehajtása mellett nagy teljesítményű, gyors megjelenítést produkáljanak. A végrehajtás menete paraméterezhető volt, de közvetlenül nem volt módosítható. Innen származik a fixed functionality (kötött funkcionalitás) elnevezés is. Ahogy a GPU-k fejlődtek, megjelentek az

[1] Jelenleg is aktív tagjai az AMD, Apple, ARM, Blizzard, IBM, Intel, NVIDIA, S3 Graphics és TransGaming képviselőiből állnak. [1]

[2] Szokták még GLslang-nak is nevezni.

általános célú grafikus processzorok, amelyek tetszőleges utasítássorozattal programozhatóak, így szinte bármilyen elképzelhető számítást végrehajtanak. A shader-ek lehetőséget biztosítanak a fejlesztőknek arra, hogy saját maguk definiáljanak algoritmusokat a megjelenítéshez.

Az OpenGL 2.0 két programozható processzort definiált, egyet a vertex shader-ek, egyet pedig a fragment shader-ek végrehajtására. Az 1. ábra szemlélteti az OpenGL hagyományos és programozható csővezetéke közötti eltéréseket. A shader-ek alkalmazásával teljes mértékben helyettesíthetjük az OpenGL hagyományos feldolgozású csővezetékét, sőt, mi magunk szabhatjuk meg a feldolgozás módját. Az 1.5-ös GLSL egy harmadik, ún. geometriai processzort is bevezetett.



(a) (b)

1. ábra. Shader-ek helye az OpenGL csővezetékében; (a) hagyományos csővezeték, (b) programozható csővezeték. [2]

2.1.1. Vertex shader

Vertex shader-eknek nevezzük azokat a programokat, amelyek a vertex processzoron kerülnek futtatásra. A vertex processzor a beérkező vertexeken és a hozzájuk tartozó adatokon végez műveleteket. A vertex processzor egyszerre csak egy vertexen dolgozik, ezáltal nem helyettesíthetőek vele az egyidejűleg több vetrexre támaszkodó műveletek. A vertex shader-ek megváltoztathatják a vertexek tulajdonságait, mint a szín, pozíció, stb. A vertex shader outputja a raszterizálóhoz kerül [3].

2.1.2. Fragment shader

A fragment processzoron kerül végrehajtásra. A fragment processzor az inputját a raszterizálótól kapja. Maga a fragment processzor a fragmentum értékeken és a hozzájuk tartozó adatokon végez műveleteket. Egy fragment shader-ben a szomszédos fragmentumok elérése nem lehetséges, továbbá egy fragment shader nem tudja megváltoztatni egy fragmentum (x, y) pozícióját. Feladata a fragmentum színének meghatározása [3].

3. Megvilágítási alapok

Ahhoz, hogy megértsük a következő fejezetekben leírt módszereket, tudnunk kell, hogy miként is kezeli az OpenGL a fényeket, anyagokat, illetve hogy hogyan határozhatjuk meg az objektumok színét ezek segítségével.

3.1. Fények, anyagok

A komputergrafikában olyan absztrakciókat használnak a fényforrások megadására, amelyeknek fizikai háttere csupán részleges, de nagyon jól modellezhetőek velük a valós megvilágítási effektusok.

Az OpenGL a fényforrások által kibocsátott fényt az alábbi komponensekre bontja, és ezekkel a komponensekkel jellemezhetőek az anyagok fényvisszaverő tulajdonságai is. Ezen fényforrás- és anyagjellemzők értékei színintenzitásokkal adhatóak meg.

Ambiens komponens: Az ambiens vagy környezeti fény a térrészben minden irányban azonos intenzitással jelen lévő fényt jelenti. Úgy tudjuk legkönnyebben elképzelni a környezeti fény hatását, ha az erősen felhős időben megfigyelhető nappali fényviszonyokra gondolunk.

Diffúz komponens: A diffúz vagy szórt fény mindig egy bizonyos irányból érkezik, és az objektumokkal ütközve azonos mértékben verődik vissza minden irányba. A diffúz visszaverődés a matt felületek jellemzője. Hatása a nézési iránytól független. A diffúz komponens írja le leginkább azt, amit mi az anyag színeként érzékelünk.

Spekuláris komponens: A spekuláris vagy tükrözött fénynek szintén van iránya. A spekuláris fény a tükröződő felületeken megjelenő fényes foltokért felelős. Hatása nagyban függ a nézőpont és a kiindulás helyétől.

Shininess komponens: Csak az anyagtulajdonságoknál adható meg, fényforrásoknál nem. Az anyagokon megjelenő spekuláris fényfoltok mérete és fényessége adható meg vele.

Az OpenGL a fényforrás helye szerint a következő fényforrás típusokat különbözteti meg:

Irány fényforrás: Az irány fényforrásokat úgy tekinthetjük, mintha végtelen távolban helyezkednének el, így a jelenetet elérő fénysugarakat párhuzamosnak feltételezzük. Irány fényforrásnak tekinthető pl. a Nap is.

Pontszerű fényforrás: A jelenethez közel helyezkednek el, az objektumot elérő fénysugarak nem párhuzamosak. Hatását úgy tudjuk elképzelni, ha pl. egy asztali lámpa fényére gondolunk.

3.2. Megvilágítási számítások

A valós világban a színeket mint a tárgyakról visszaverődő fényt érzékeljük. Tehát a felület egy adott pontjában megjelenő szín a fényforrásból kiinduló fény és a felület kölcsönhatásaként áll elő. Ebben a kölcsönhatásban a fényforrás által kibocsátott fény színe, iránya, a felület színe, orientációja és a nézőpont helye vesz részt. A fényforrások és az anyagok színe, tulajdonságai a 3.1. szakaszban ismertetettek szerint adhatók meg, a fényvisszaverődés modellezése vektorok segítségével történik.

Vektorok közötti szögeket fogunk számolni, hogy meghatározzuk, milyen szögben éri a fény a felületet, milyen szögben verődik vissza róla, illetve hogy hogyan orientálódik a felület vagy a fényforrás a nézőponthoz képest. A komputergrafikában két vektor szöge helyett a szög koszinuszát alkalmazzák a megvilágítási számításokban, ezt a skaláris szorzattal, más néven belső szorzattal határozhatjuk meg. Az A és B vektorok belső szorzata a következőképpen számolható:

$$\frac{A \cdot B}{\|A\| * \|B\|} = \cos\alpha,$$

ahol $\|A\|$ az A vektor hossza, α pedig az A és B által bezárt szög. Látható tehát, ha A és B egységnyi hosszúságúak, akkor a belső szorzat pontosan a két vektor szögének koszinuszát adja.

3.3. Per-vertex és per-fragment megközelítés

Ha háromdimenziós alkalmazást fejlesztünk, akkor már az elején célszerű eldöntenünk, hogy a végeredmény szempontjából a teljesítmény vagy a minőség fontosabb. Ha a teljesítmény a fontosabb, akkor érdemes per-vertex implementációt készíteni, ellenkező esetben per-fragment implementáció készítése ajánlott a megvilágítási számításokhoz.

Per-vertex (csúcspontonkénti) számítás: Az intenzitásértékeket a vertexekben számítjuk ki, majd a kiszámított intenzitásértékeket interpoláljuk a vertexek között elhelyezkedő pontokban. Ez lineáris interpolációnak felel meg, ezért gyors, de nem túl minőségi megvalósítást eredményez.

Per-fragment (fragmentumonkénti) számítás: A vertexekben nem számítjuk ki az intenzitásértékeket, csak az ahhoz szükséges vektorokat határozzuk meg, és ezeket interpoláljuk a vertexek között elhelyezkedő fragmentumokra. Az intenzitásokat az interpolált vektorokat felhasználva fragmentumonként számítjuk ki. Mivel a számításokat minden felületi pontban el kell végezni, a módszer lassabb, mint a per-vertex megoldás, viszont jobb minőségű végeredményt szolgáltat.

Habár a mai nagyteljesítményű grafikus hardverek jól megbirkóznak a számításigényesebb feladatokkal is, interaktív 3D-s alkalmazások fejlesztésekor érdemes körültekintően eljárni a per-pixel vagy per-vertex implementáció megválasztásánál.

4. A hagyományos (fotorealisztikus) megjelenítés

Ahhoz, hogy megértsük, hogy a nem fotorealisztikus módszerek mennyiben térnek el fotorealisztikus társaiktól, ismernünk kell a fotorealisztikus árnyalásnál alkalmazott számításokat. Ez a szakasz a Phong-árnyalás vizsgálata során mutatatja be számunkra, hogy miként értelmezhetőek a fényvisszaverődési kölcsönhatások a fotorealisztikus megvilágítási modellekben. Ez a későbbi teljesítményelemzés során is hasznos lesz.

Szükségünk lesz mindenekelőtt egy árnyalási egyenletre, ez fogja meghatározni, hogy milyennek látjuk majd a megvilágított felületet. Ha matt és tükrös felületeket is modellezni szeretnénk, akkor az ambiens, diffúz és spekuláris fényvisszaverődési számítások mindegyikét szerepeltetnünk kell az árnyalási egyenletben. A tisztán ambiens területet azok a pontok alkotják a felületen, amelyeket közvetlenül nem érnek el a fényforrásból érkező fénysugarak, míg a diffúz és spekuláris részeket a fény közvetlenül éri. A diffúz fényvisszaverődés önmagában is képes megadni a felület formáját, a spekuláris visszaverődés a felület anyagának optikai tulajdonságairól ad információt.

Az árnyalási egyenletet általános alakban a következőképpen írhatjuk fel:

$$I = I_a + I_d + I_s$$

Az I_a, I_d és I_s rendre a visszavert ambiens, diffúz és spekuláris intenzitásokat jelölik, ezek összegeként kapjuk meg a végleges színintenzitást, amit I jelöl.

4.1. Phong-árnyalás

Ha valóban realisztikus eredményt kívánunk kapni, akkor fragmentumonként kell elvégeznünk a megvilágítási számításokat, azaz egy per-fragment árnyalási modellt kell alkalmaznunk. A Phong-árnyalással [4, 5, 6] pontosan ilyen módszert kapunk. A Phong árnyalási mód mindössze annyit mond, hogy a normálvektorokat interpolálnunk kell a csúcspontok között, hogy aztán az intenzitásértékeket fragmentumonként számíthassuk ki. A megvilágítási számításokban alkalmazott vektorokat a 2. ábra szemlélteti.

Az ambiens visszaverődés: Mivel azt mondtuk, hogy a környezeti fényt a térben minden irányban azonos intenzitással jelenlevő fénynek tekinthetjük, az ambiens fényvisszaverődést a következőképpen határozhatjuk meg:

$$I_a = k_a * l_a$$

l_a a környezeti fény intenzitása, k_a pedig a felület ambiens fényvisszaverő együtthatója.

A diffúz visszaverődés Lambert modellje: A Phong-árnyalásnál a Lambert által megfigyelt összefüggést használjuk a diffúz fényvisszaverődés meghatározására. Lambert törvénye kimondja, hogy a visszaverődött fény intenzitása a megvilágítási szög koszinuszával arányos. Ennek meghatározása a felületi normális (N) és a vizsgált felületi pontból a fényforrásba mutató vektor (L) alapján a belső szorzattal történik.

$$I_d = k_d * (N \cdot L) * l_d ;$$

ahol k_d a felület diffúz visszaverődési együtthatója, l_d pedig a fényforrás diffúz fényintenzitása.

A spekuláris visszaverődés Phong modellje: Ez egy tapasztalati megfigyelésen alapuló modell. A spekuláris visszaverődés a tükröződő felületeken a pontos visszaverődési irány környezetében figyelhető meg. Minél jobban tükröz a felület, annál kisebb környezetben jelenik meg a spekulárisan tükrözött fény rajta. Azaz egy tökéletesen tükröző felület esetében csakis a tükörirányban lenne jelen a spekuláris visszaverődés. A Phong által javasolt összefüggés alapján a spekuláris visszaverődés intenzitását a következőképpen számíthatjuk ki:

$$I_s = k_s * (V \cdot R)^n * l_s$$

k_s a felület spekuláris visszaverődési együtthatója, V az adott felületi pontból a nézőpontba mutató vektor, R a visszaverődés irányába mutató vektor, n a felület fényességét meghatározó hatvány (exponens), l_s pedig a fényforrás által kibocsátott spekuláris intenzitás.

Az így megkapott I_a, I_d és I_s együtthatókat a korábban általánosan megadott árnyalási egyenletbe behelyettesítve megkapjuk a Phong modell által javasolt árnyalási egyenletet.

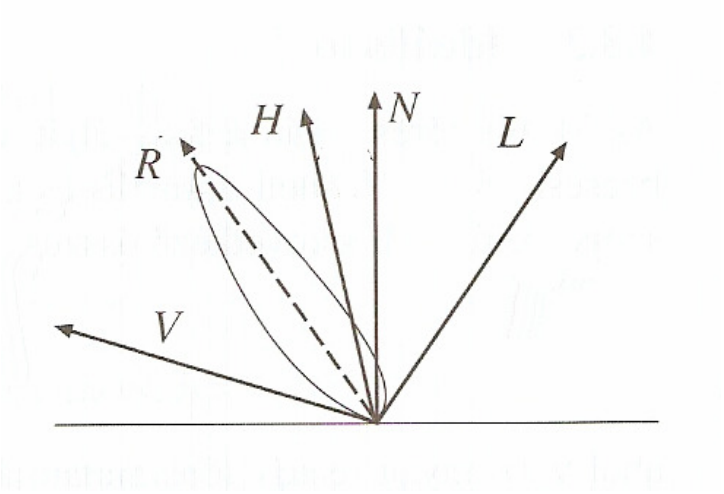
$$I = k_a * l_a + k_d * (N \cdot L) * l_d + k_s * (V \cdot R)^n * l_s$$

A spekuláris visszverődés Phong-Blinn modellje: Hatékonyabb megvalósítást kapunk, ha a Phong modell helyett a Phong-Blinn modellt választjuk, mivel a Phong modellben az *R* vektor számítása költséges. A modell egy *H* vektort, az ún. félszög vektort javasolja a spekuláris számítások elvégzéséhez. *H* az *L* és *V* szögét felező vektor.

$$H = \frac{L+V}{2}$$

A modellel a spekuláris visszaverődés a következőképpen számítható:

$$I_s = k_s * (H \cdot N)^n * l_s$$



2. ábra. A megvilágítási számításoknál alkalmazott vektorok. [4]

4.2. A Phong-árnyalás implementáció

Az implementáció a vertex shader-ben határozza meg a vektorokat. A fényforrás pozícióját az első fényforrásra (*GL_LIGHT0*) beállítottak alapján kezeli. A számítások előtt kamera-koordinátarendszerbe kell transzformálnunk az aktuális vertexet. A számítások végrehajtásánál pontszerű fényforrást feltételezünk.

1. KÓDRÉSZLET. Phong-árnyalás vertex shader

```
// vektorok deklarációi
varying vec3 N;
varying vec3 L;
varying vec3 R;
varying vec3 V;
```

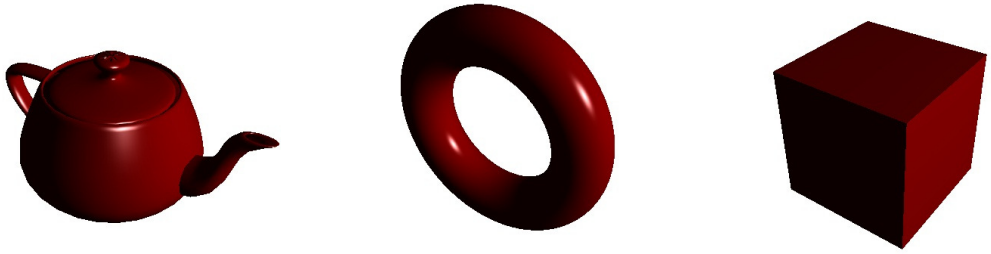
```
void main()
{
      // a vertex kamera-koordinátarendszerbeli helye
      vec3 Position = vec3(gl_ModelViewMatrix * gl_Vertex);

      // vektorok meghatározása
      N = gl_NormalMatrix * gl_Normal;
      L = normalize(vec3(gl_LightSource[0].position)-Position);
      R = reflect(-L, N);
      V = normalize(-Position);

      gl_Position = ftransform();

}
```

A fragment shader a vertex shader által meghatározott vektorokat interpolálva kapja meg, ezeket normalizálni kell, mielőtt felhasználjuk őket. A shader az alkalmazásban beállított anyagtulajdonságokat és az első fényforrásra beállított jellemzőket használja fel a pontbeli fényvisszaverődés meghatározásához. Az implementáció a 3. ábrán látható eredményt adja.

2. KÓDRÉSZLET. Phong-árnyalás fragment shader

```
// vektorok deklarációi
varying vec3 N;
varying vec3 L;
varying vec3 R;
varying vec3 V;

void main()
{
      // a szükséges belső szorzatok értékét tároló változók
      float nDotl;
      float vDotr;

      // az anyag fényvisszaverő jellemzői komponensenként tárolva
      vec3 ka = vec3(gl_FrontMaterial.ambient);
      vec3 kd = vec3(gl_FrontMaterial.diffuse);
      vec3 ks = vec3(gl_FrontMaterial.specular);
      float s  = gl_FrontMaterial.shininess;

      // a fényforrás által kibocsátott fény komponensenként tárolva
      vec3 la = vec3(gl_LightSource[0].ambient);
      vec3 ld = vec3(gl_LightSource[0].diffuse);
      vec3 ls = vec3(gl_LightSource[0].specular);

      // a visszavert intenzitások komponensenkénti
      // tárolására szolgáló változók
      vec3 i, ia, id, is;

      // a szükséges belső szorzatok meghatározása
      nDotl = max(dot(normalize(N), normalize(L)), 0.0);
      vDotr = max(dot(normalize(V), normalize(R)), 0.0);
```

```
        // a fényvisszaverődés komponensenkénti meghatározása
        ia = ka * la;
        id = kd * (nDotl) * ld;
        is = ks * (pow(vDotr, s)) * ls;

        // fragmentum végső intenzitása
        i = ia + id + is;

        gl_FragColor = vec4(i, 1.0);

}
```



3. ábra. A Phong-árnyalás implementációjának eredménye.

5. Nem fotorealisztikus technikák

A komputergrafikában az elsődleges szempont mindig is a valósághű megjelenítés irányába történő fejlődés volt, és valószínűleg nem lesz ez másként a későbbiekben sem. Ez a fejezet betekintést nyújt egy ettől eltérő vizualizáció megértésébe. A nem fotorealisztikus megjelenítési technikák nagyon széles területet fednek le, éppen ezért nagyon szerteágazó kutatási területtel rendelkeznek. Ezen technikák közös célja, hogy a hagyományosan kézzel alkotott művek illúzióját adják. Az alfejezetekben ismertetésre kerülnek olyan nem fotorealisztikus módszerek, amelyeket előszeretettel használnak a videojáték-iparban (lásd 4. ábra), CAD rendszerekben vagy a komputergrafika más területein. Akad közöttük olyan technika is, amely a képzőművészetben már évszázadok óta jelen van.

(a) (b)

4. ábra. Nem fotorealisztikus megjelenítést használó videojátékok;
(a) Borderlands[3], (b) Street Fighter 4[4].

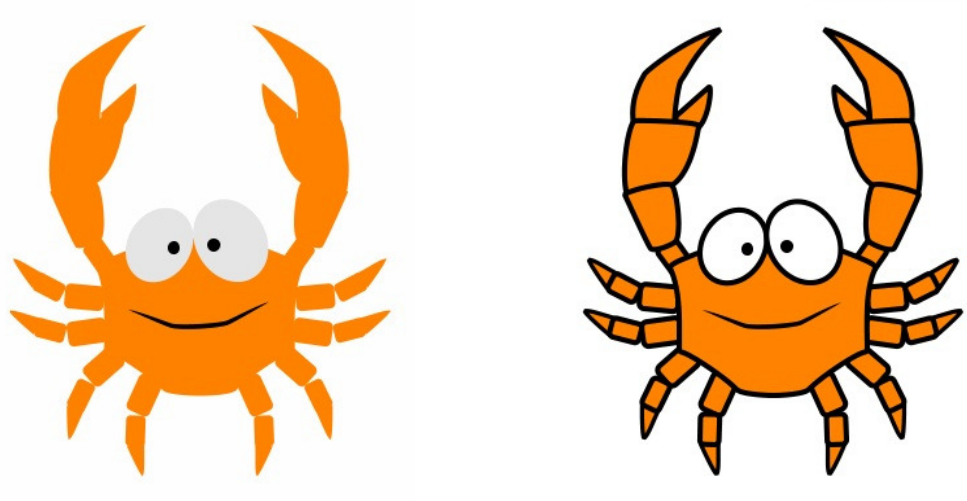
5.1. Kontúrok rajzolása

Amikor nem fotorealisztikus módszereket használunk az ábrázoláshoz a komputergrafikában, a célunk legtöbbször továbbra is a valós világ objektumainak modellezése, csak egy kevésbé valószerű, mondhatni absztrakt formában. Ez az absztrakció azonban oda vezet, hogy gyakran megváltozik a modellezett objektumok színe, formája, ezáltal azok sokkal nehezebben különíthetők el a környező, szintén absztrakt világtól, csökken a felismerhetőség. Sok esetben az absztrakció olyan magas szintre emelkedik, a

[3] http://downloads.2kgames.com/borderlands/website/images/screenshots/hiRes/9.jpg
[4] http://www.thebitbag.com/wp-content/uploads/2008/03/marseille_rolling01_bmp_jpgcopy.jpg

felismerhetőség olyannyira leromlik, hogy szükség van valamire, ami vizuálisan elszeparálja az objektumokat egymástól és az őket körülvevő világtól, segít az objektumok formájának követésében és javítja a felismerhetőséget. A képregényrajzolók, grafikusok, animátorok a kezdetektől fogva alkalmazzák a kontúrokat (körvonalakat) erre a célra. (lásd 5. ábra)

Ahhoz azonban, hogy létre tudjuk hozni a kontúrokat, először meg kell adnunk a kontúr definícióját. Mint ahogyan két különböző grafikus is máshogy rajzolná be egy objektum kontúrját, úgy a számítástechnikában alkalmazott módszerek is eltérő eredményeket mutatnak. Más-más módszerek állnak a rendelkezésünkre attól függően, hogy az objektumtérben (3D) vagy a képtérben (2D) folytatjuk a vizsgálódást, továbbá más-más eredményeket kapunk a felhasznált módszerektől függően is.



(a) (b)

5. ábra. Kontúrok szerepe a felismerhetőségben; (a) figura kontúrok nélkül, (b) figura kontúrokkal.[5]

5.1.1. Kontúrvizsgálatok a képtérben

Képfeldolgozó módszerek használatával könnyedén megkaphatunk bizonyos körvonalakat a már kétdimenzióra leképezett jelenetből. Az eljárások célja a kiindulási képből, azaz az ún. intenzitásképből egy élkép meghatározása, ami viszont csak a feltárt éleket tartalmazza. Bár

[5] http://www.how-to-draw-funny-cartoons.com/how-to-do-an-outline.html

ezen eljárások pontossága megkérdőjelezhető, sok alkalmazásnak megfelel az általuk nyújtott precizitás.

Az emberi szem az intenzitásváltozásokra nagyon érzékeny, ez alapján tudja érzékelni az objektumok határvonalait is. Az objektumok kontúrpontjainak feltárásához azokat a helyeket kell meghatározni a képen, ahol a környező képpontok színintenzitásai gyorsan változnak. Annyit kell csupán tennünk, hogy a háromdimenziós jelenetet leképezzük kétdimenziósra, majd az így kapott képet alávetjük a képfeldolgozásban alkalmazott éldetektáló eljárások egyikének. A módszer OpenGL-ben egyszerűen megvalósítható, mivel a színbuffer tartalma a *glCopyTexImage2D* vagy a *glCopyTexSubImage2D* utasítások segítségével könnyen egy kétdimenziós textúrába másolható, és ezen az éldetektálás végrehajtható. Ez az eljárás azonban nem alkalmazható olyan jelenetekre, amely textúrázott modelleket tartalmaz, mivel az éldetektáló eljárások természetüknél fogva – mivelhogy az egy adott területen jelen lévő intenzitásváltozásokat keresik – sok fölösleges élt is feltárnak. További hiányossága az eljárásnak, hogy az azonos színű objektumokat tartalmazó jelenetek esetén gyakran nem találja meg a szükséges élpontokat. A megoldás a problémára a mélységtérképek (depth map) használata.

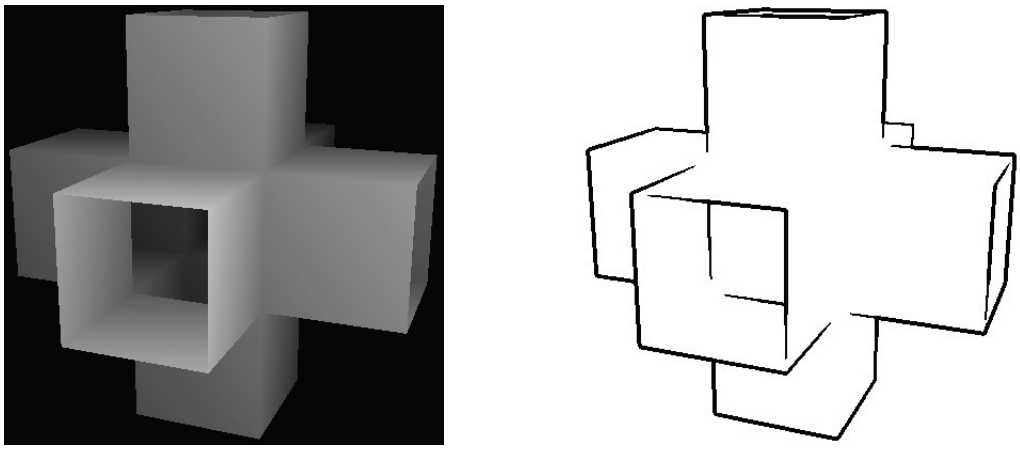
5.1.1.1. Mélységtérképek

Egy mélységtérkép [7] egy olyan kétdimenziós képnek (intenzitásképnek) felel meg, amelyet a háromdimenziós jelenetből úgy képzünk, hogy a képpontok intenzitása az objektumon annak megfelelő felületi pont térbeli mélységét reprezentálja. (lásd 6 ábra) Az így létrehozott mélységtérkép tulajdonképpen egy szürkeskálás intenzitáskép. (A képen megjelenő intenzitások mind a szürke árnyalatai.)

Az összefüggő objektumoknál, objektumrészeknél a mélységtérképen lágy intenzitásváltozás tapasztalható, különálló objektumoknál, objektumrészeknél viszont ez a változás ugrásszerű.

Szerencsére az OpenGL a mélységbufferből is képes adatokat kiolvasni és azokat kétdimenziós textúrába írni az 5.1.1. szakaszban már említett parancsok segítségével. Az így kapott ún. mélységtextúrát aztán már továbbadhatjuk feldolgozásra egy éldetektáló eljárásnak. A mélységtérkép használatával elkerülhetjük a hagyományos képek esetén fellépő hiányosságokat, nevezetesen a homogén felületek által elrejtett éleket, illetve a textúrázott

felületeken feltárt hamis kontúrokat. Azonban ez a módszer sem tár fel tökéletesen minden szükséges élt, elsősorban a gyűrődések megtalálása, illetve az azonos mélységben elhelyezkedő objektumok, felületrészek körvonalainak feltárása jelenthet problémát.



(a) (b)

6. ábra. Éldetektálás mélységtérképen; (a) mélységtérkép,
(b) mélységtérképen meghatározott élek. [7]

Látható a 6. ábrán, hogy a mélységtérképek használatával egészen jó eredményeket kaphatunk, de még mindig rengeteg él marad feltáratlanul. Egy másik hatékony megoldás a normáltérképek (normal map) használata.

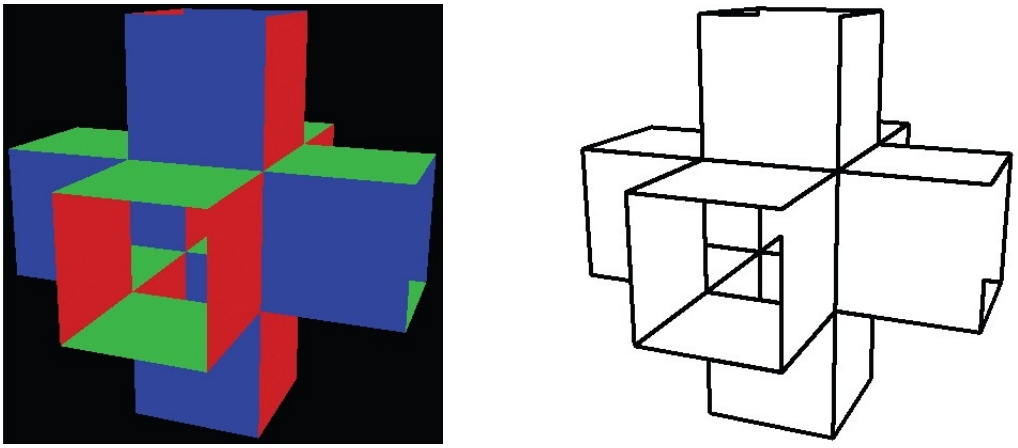
5.1.1.2. Normáltérképek

Ahogy a neve is mutatja, a felületi normálvektorokat használjuk fel a normáltérkép [7] előállításához. A következő műveletek segítségével állítható elő egy normáltérkép:

- Helyezzünk el egy vörös fényforrást az *X* tengelyen, egy zöld fényforrást az *Y* tengelyen és egy kék fényforrást a *Z* tengelyen, ezek nézzenek az objektumra.
- Helyezzünk el az előbbi fényforrásoknak megfelelően ellentétes intenzitású fényforrásokat ugyanazokon a helyeken, de ellentétes irányban.
- Állítsuk az objektum színét fehérre.
- Állítsunk be az objektumra diffúz anyagtulajdonságot.

Egy felületi pont végső színe a diffúz visszaverődésnél ismertetettek szerint a pontbeli normálvektor és a pontból a fényforrásokba mutató vektorok alapján a belső szorzattal

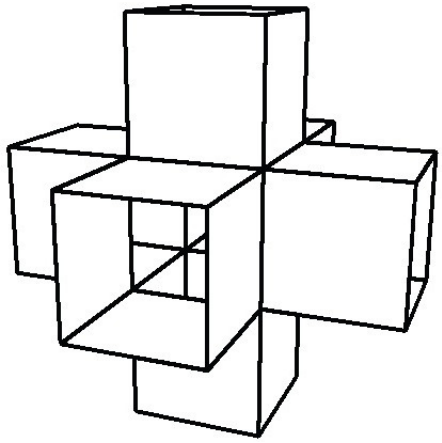
meghatározható. Az így előállított normáltérkép egy olyan intenzitáskép, amely minden pontjában a neki megfeleltethető objektumponthoz tartozó normálvektort reprezentálja. (lásd 7. ábra) Az élek aztán ebből a normáltérképből meghatározhatók, de az így megkapott élkép sem tartalmazza hiánytalanul a szükséges kontúrokat.



(a) (b)

7. ábra. Éldetektálás normáltérképen; (a) normáltérkép,
(b) normáltérképen meghatározott élek. [7]

A mélységtérképekből és a normáltérképekből nyert élek kombinálásával nagyon jó eredményeket érhetünk el. (lásd 8. ábra) A képtérben alkalmazott módszerek előnye, hogy sebességükre a háromdimenziós jelenet részletessége nincs hatással, a teljesítmény elsősorban a felbontástól függ, hiszen az éldetektáló műveleteket egy kétdimenziós intenzitáskép pixelein végezzük.



8. ábra. Mélységtérképen és normáltérképen detektált élek kombinálva. [7]

5.1.1.3. Éldetektálás

Láthattuk, miként lehet előállítani a háromdimenziós jelenetből azokat a kétdimenziós intenzitásképeket, amelyekből meghatározhatóak a szükséges kontúrok. A kontúrok kinyeréséhez azonban egy megfelelő éldetektáló eljárás szükséges. A Sobel éldetektáló [8] eljárás mélységtérképeken is és normáltérképeken is elfogadható pontossággal képes meghatározni a kontúrokat. Az eljárás a konvolúciós maszkolás segítségével detektálja az éleket.

A képfeldolgozásban gyakran alkalmaznak maszkokat az élek detektálására. A maszkokat mint mátrixokat képzelhetjük el, amelyek komponensei valós súlyok lehetnek. Általában 3x3-as maszkokat alkalmaznak, mivel ezek alakjukat tekintve centrálszimmetrikusak, ezáltal jól illeszthetőek a centrális elemet a vizsgált képpontnak megfeleltetve. Továbbá kevésbé zajérzékenyek, mint a kisebb méretű maszkok, valamint a legtöbb esetben megfelelő eredményt szolgáltatnak. A maszkolás lényege, hogy nem az egész képet vizsgáljuk, hanem annak egy kisebb, a maszk méretének megfelelő részletét. A kép összes pontjára sorra ráillesztjük a maszkot, majd ez alapján értelmezünk valamilyen műveletet a maszk elemei és az eredeti kép maszk által lefedett pontjai között, eredményként pedig egy maszkválaszt – az értelmezett művelet alapján meghatározott értéket – kapunk.

A konvolúciós maszkolás lényege, hogy a maszk és a kép között értelmezett műveletként a belső szorzatot választjuk. 3x3-as, sorfolytonosan 9 dimenziós vektorokra leképzett maszkok esetén a konvolúciós maszkolás által adott maszkválasz érték

$$(W, X) = \sum_{i=1}^{9} w_i x_i \ ,$$

ahol w_i a W maszkvektor i-edik eleme, x_i pedig a vizsgált képterület pontjaiból képzett X vektor i-edik eleme.

A maszkokban megválasztott súlyok értéke kritikus a végeredmény szempontjából. Az éldetektálásnál fontos, hogy a homogén intenzitású területeken a maszkválasz 0 legyen, ehhez 0 súlyösszegű maszkot kell használnunk. A Sobel éldetektálásnál alkalmazott 3x3-as maszkokat a 9. ábra mutatja.

-1	-2	-1
0	0	0
1	2	1

(a)

-1	0	1
-2	0	2
-1	0	1

(b)

9. ábra. Sobel éldetektálásnál alkalmazott maszkok; (a) vertikális, (b) horizontális. [8]

A Sobel éldetektáló egy gradiens alapú módszer, azaz az intenzitásváltozás helyét és mértékét a gradiensvektorral határozza meg. A gradiensvektor (érintővektor) a kép első deriváltjaiból képzett

$$G[f(x,y)] = \begin{bmatrix} G_x \\ G_y \end{bmatrix}$$

vektor, ahol

$$G_x = \frac{\partial f}{\partial x} \text{ és } G_y = \frac{\partial f}{\partial y}$$

a gradiensvektor komponensei, az *x* és *y* tengely irányába történő változás mértékét adják meg. A gradiensvektor hossza

$$\left|G[f(x,y)]\right| = \sqrt{G_x^2 + G_y^2} \ .$$

Ha meghatározzuk a gradiens nagyságát (a gradiensvektor hosszát), akkor megkapjuk a változás mértékét. A G_x és G_y közelítésére a 9. ábrán látható maszkokat használjuk fel úgy, hogy az intenzitáskép és a maszk között a konvolúciós műveletet hajtjuk végre.

Az élképet az intenzitásképből megkapjuk, ha minden (*x, y*) pontjában meghatározzuk a gradiens nagyságát. Említettük, hogy a maszkválasz homogén területek vizsgálata esetén sötét színt eredményez, az intenzitásátmeneteknél pedig világosat. Viszont a nem fotorealisztikus módszereknél általában fekete vagy sötét élekre van szükségünk, tehát a megkapott élképet binarizálnunk kell. Előfordulhat, hogy a kapott élek nem megfelelőek az alkalmazási célra, ekkor további utófeldolgozó lépések hajthatóak végre a kapott élképen.

Amennyiben a sötét színű kontúrokat át szeretnénk vinni az élképről az eredeti intenzitásképre, küszöbölést alkalmazhatunk. Ha az élképen egy képpont intenzitása egy adott küszöbérték fölött van, akkor feketére állítjuk a képpont intenzitását, különben az intenzitásképen annak megfelelő képpont színét használjuk.

A Sobel éldetektáló eljárás mind a mélységtérképekre, mind a normáltérképekre végrehajtható, az így kapott éleket egyesíthetjük egy harmadik élképen a pontosabb eredmény érdekében, sok esetben azonban a mélységtérképen történő éldetektálás önmagában is megfelelő eredményt szolgáltat.

5.1.2. Kontúrvizsgálatok az objektumtérben

Ha az objektumteret választjuk vizsgálódásunk helyéül, akkor precízebb, megfelelőbb eredményeket kaphatunk, mintha a képtérben vizsgálódnánk, hiszen geometriai információk alapján vizsgáljuk az élek kontúr voltát. Mivel az éleknek rengeteg típusát különböztethetjük meg az objektumtérben, meg kell adnunk, hogy mit tekintünk kontúr élnek. Az objektumtérben kontúrok meghatározásához elegendő az ún. sziluett élek meghatározása, sőt, a sziluett élek azontúl, hogy megadják az objektum szigorú értelemben vett határolóéleit (körvonalait), bizonyos gyűrődéseket is feltárnak.

Egy élt akkor tekintünk sziluett élnek, ha egy elülső poligonlap egy hátsó poligonlappal kapcsolódik össze az adott élen keresztül. Ez matematikailag a felületi normálvektor és a csúcspontból a nézőpontba mutató vektor segítségével, a belső szorzattal a következőképpen fejezhető ki:

$$N_1 \cdot V * N_2 \cdot V \leq 0,$$

ahol N_1 és N_2 az élben találkozó poligonlapok normálvektorai, V pedig a vizsgált élen elhelyezkedő csúcspontból a nézőpontba mutató vektor. Ha a feltétel teljesül, akkor a vizsgált él sziluett él.

A vizsgálatokhoz tárolnunk kell az éllistát, hozzájuk kapcsolódóan a poligon- és csúcslistát. Az élek vizsgálatát minden egyes jelenet megrajzolása előtt el kell végeznünk, azaz minden egyes képkocka megjelenítése előtt be kell járnunk az éllistát, és minden egyes él esetén meg kell vizsgálnunk, hogy a megadott feltétel teljesül-e. Ez nagy poligonszámú modellek estén nagyon költséges lehet. További korlátozó tényező, hogy a shader-ekben történő megvalósítás a szomszédos vertexek ismerete nélkül nehéz. A CPU-n való megvalósítás nagy poligonszámú modellek esetén szerény eredményeket produkál, hiszen a CPU-nak a háttérben futó folyamatok kiszolgálását is el kell látnia. Interaktív alkalmazásokban történő kontúrmeghatározásra jobban megfelelnek a képtérben végrehajtott műveletek. [7, 9]

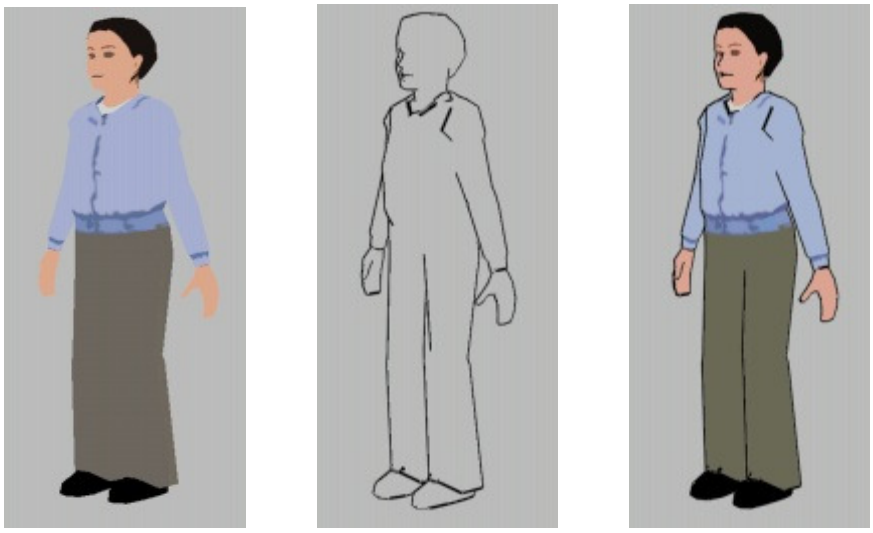
5.1.3. Kontúrrajzolás a standard OpenGL segítségével

Nem szükséges minden esetben bonyolult módszereket implementálni ahhoz, hogy megkapjuk a kívánt kontúrokat. Az objektum sziluett éleit megkaphatjuk néhány egyszerű OpenGL-utasítás segítségével, ehhez azonban kétszer kell megrajzolnunk az objektumot. Első lépésként csak az objektum elülső oldalát rajzoljuk meg a kívánt kitöltési módot és az alapértelmezett mélységbuffer beállításokat használva, második lépésként megrajzoljuk az objektum hátsó lapjait vonalas megjelenítéssel, módosított mélységbuffer beállításokkal. Az eljárás kódja a következő:

3. KÓDRÉSZLET. Sziluett élek rajzolása a standard OpenGL-el

```
// első lépés
glPolygonMode(GL_FRONT,GL_FILL);   // elülső oldal rajzolása kitöltve
glDepthFunc(GL_LESS);              // alap mélységbuffer-beállítás
glCullFace(GL_BACK);               // hátsó lapok eltávolítása
display();                         // rajzolást végző eljárás

// második lépés
glPolygonMode(GL_BACK,GL_LINE);    // hátsó oldalak rajzolása drótvázzal
glDepthFunc(GL_LEQUAL);            // mélységbuffer átállítása
glCullFace(GL_FRONT);              // elülső lapok eltávolítása
glColor3f(0.0, 0.0, 0.0);          // fekete rajzolási szín beállítása
display();                         // rajzolást végző eljárás
```

Az alapértelmezett mélységbuffer-beállítás a *GL_LESS*, ebben az esetben egy objektumpont színe csak akkor kerül rajzolásra a képernyőn, ha az objektumpont mélységértéke kisebb a mélységbufferben szereplő értéknél. A *GL_EQUAL* beállítás használatával előírhatjuk, hogy egy objektumpont színe akkor is felülírja a korábbit, ha az objektumpont mélységértéke megegyezik a bufferben szereplő értékkel. A vonalvastagság opcionálisan megadható a *glLineWidth* parancs segítségével. Az eljárás megfelelő eredményt adhat egyszerű objektumok esetén, ahol gyorsan és egyszerűen akarjuk megkapni a sziluett éleket, illetve nem akarunk további feldolgozó lépéseket végrehajtani az éleken. Animált jelenetek estén az eljárás hátránya, hogy az élek megjelenése és eltűnése hirtelen váltással történik, ez különösen vastagabb vonalbeállítás esetén lehet zavaró, mivel nem folytonos animáció érzetét keltheti. Az eljárás eredménye a 10. ábrán látható. [9]



(a) (b) (c)

10. ábra. Sziluett élek az objektumtérben; (a) modell sziluett élek nélkül, (b) sziluett élek, (c) modell sziluett élekkel. [9]

5.1.4. Kontúrrajzolás-implementáció

Mivel az objektumtérben folytatott kontúrvizsgálatok tipikusan a vertexek, poligonok szomszédsági viszonyain alapulnak, shader-ben történő megvalósításuk nem egyszerű. A képtérben való éldetektálás szintén a szomszédsági viszonyokra épül, hiszen a maszkválasz meghatározásához közvetlen hozzáféréssel kell rendelkeznünk a maszk által lefedett képpontokhoz. Ez azonban egyszerűen megoldható az OpenGL által nyújtott lehetőségekkel.

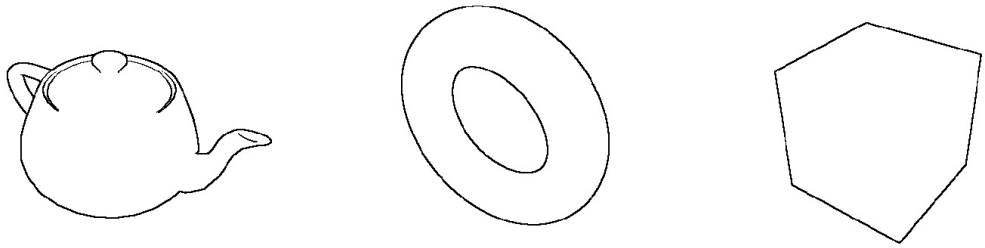
A mélységbuffer tartalmát a *glCopyTexSubImage2D* parancs egy mélységtextúrába olvassa, ezt a textúrát kapja meg a fragment shader. Vertex shader-re nincs szükségünk, a szükséges transzformációs műveleteket a hagyományos csővezeték elvégzi. Normáltérkép használatát az implementáció mellőzi, hiszen az általam használt egyszerű objektumokra a mélységtérképen történő éldetektálás önmagában is megfelelő eredményt ad.

A shader-nek két textúra kerül átadásra, a már említett mélységtextúra, valamint a színbufferből kiolvasott textúra, amely az élek nélkül renderelt jelenetet tartalmazza. Átadásra kerül továbbá a shader-nek egy a maszkeltolásokat (offset) tartalmazó tömb, melynek segítségével a mélységtextúra maszk által fedett területeit elérjük. A shader a 9. ábrán szereplő maszkokkal dolgozik.

A fragmentum végső színintenzitását küszöböléssel határozzuk meg. Az élpontokban a kiszámolt gradiensérték magas, az összes többi pontban alacsony. Ahol az intenzitásérték átlép egy bizonyos küszöböt, ott feketére állítjuk a fragmentum színét, egyébként a színbufferben szereplő színt alkalmazzuk. A végeredményt a 11. ábra mutatja.

4. KÓDRÉSZLET. Sobel éldetektáló fragment shader

```
// a jelenetet ábrázoló textúra
uniform sampler2D colorMap;
// a mélységtextúra
uniform sampler2DShadow shadowMap;
// a maszkoláshoz szükséges eltolásértékeket tartalmazó tömb
uniform vec2 offset[9];

void main(void)
{
      // a maszk által lefedett képpontok intenzitásainak
      // tárolására szolgáló tömb
      vec4 sample[9];

      // horizontális és vertikális irányokba történő változás
      vec4 Gx;
      vec4 Gy;

      // a fragmentum színét tároló változó
      vec3 color;

      // a maszkoláshoz szükséges intenzitásértékek
      // összegyűjtése a sample tömbbe
      for(int i = 0; i < 9; i++){
            sample[i] = shadow2D(shadowMap, gl_TexCoord[1].stp +
                                            vec3(offset[i], 0));
      }

      // horizontális változás
      Gx = (sample[2] + (2.0 * sample[5]) + sample[8]) -
           (sample[0] + (2.0 * sample[3]) + sample[6]);

      // vertikális változás
      Gy = (sample[6] + (2.0 * sample[7]) + sample[8]) -
           (sample[0] + (2.0 * sample[1]) + sample[2]);

      // a gadiens nagyságát (változás mértékét) reprezentáló intenzitás
      color = sqrt((Gx * Gx) + (Gy * Gy)).rgb;

      // a végső fragmentumszín meghatározása küszöböléssel
      if(color.r < 0.07)
            // az objektum (jelenet) színe
            color = vec3(texture2D(colorMap, gl_TexCoord[0].xy).rgb);
      else
            // a kontúr színe (fekete)
            color = vec3(0.0, 0.0, 0.0);

      gl_FragColor = vec4(color, 1.0);
}
```



11. ábra. Kontúrrajzoló implementáció eredménye.

5.2. Rajzfilm árnyalás (Cartoon shading)

Egyszerű, mégis látványos eredményt kapunk, ha a kívánt jelenetet cartoon shading segítségével ábrázoljuk. A cél olyan összhatás elérése, amely a rajzfilmek, rajzok, képregények képi világát idézi. A technika lényege, hogy az objektumok megjelenítéséhez mindössze néhány színt, tónust használunk, így az eredmény kétdimenziós ábra érzetét kelti bennünk. Ezt a fajta megjelenítést a videojáték-ipar is évek óta használja cel-shading néven, általában valamilyen kontúrrajzolási eljárással együtt alkalmazzák a megfelelő eredmény eléréséhez. Nézzük meg, hogyan is működik a módszer.

5.2.1. A rajzfilm árnyalás által alkalmazott színek

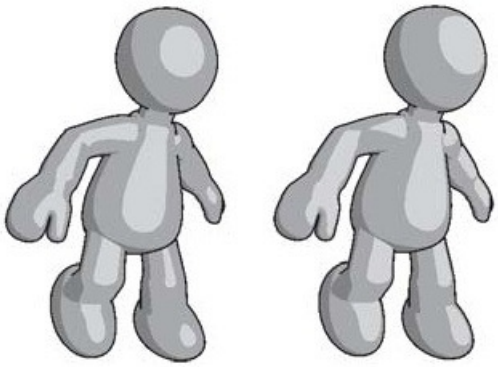
A fotorealisztikus megjelenítésnél egy megvilágított felületen a színintenzitások finoman, egyenletesen változnak. Ez annak köszönhető, hogy ezek a módszerek az árnyalási egyenletben a fény beesési szögét figyelembe véve számítják ki a végső színintenzitást. A cartoon shading viszont előre megadott színintenzitásokat rendel a fragmentumokhoz attól függően, hogy milyen visszaverődési feltételek teljesülnek az adott pontban. A különböző cartoon shading eljárások nagyon eltérőek lehetnek olyan tekintetben, hogy hány színt alkalmaznak egy anyag árnyalásánál. Ahogy a fotorealisztikus módszereknél, úgy a cartoon shading technikánál is meg kell határozni a diffúz és spekuláris visszaverődési számításoknál felhasznált szögeket, de nem ez alapján kerül kiszámításra a végső színintenzitás, csak arra használjuk, hogy válasszunk egyet a már előre definiált intenzitásokból.

Egy lehetséges megközelítése a rajzfilm árnyalásnak a következő:

1. Első lépésben azt kell eldöntenünk, hogy a felület egy adott pontjában van-e diffúz fényvisszaverődés. Ehhez tudnunk kell, hogy a felület egy adott pontjában a diffúz (és spekuláris) visszaverődés csak akkor van jelen, ha a fény beesési szöge kisebb, mint kilencven fok. Ha ez teljesül, akkor diffúz, különben ambiens rajzolási szín lesz kiválasztva. (Több diffúz rajzolási színt is megadhatunk a fény beesési szöge alapján).
2. Második lépésként azt vizsgáljuk, van-e spekuláris visszaverődés az adott pontban. Ehhez meg kell adnunk egy felső küszöböt a Phong vagy Phong-Blinn modell által javasolt vektorok szögére. Ha a szög az adott pontban a küszöbérték alatt van, akkor a spekuláris szín felülírja a ponthoz az első lépésben kiválasztott ambiens vagy diffúz színt.

[10]

Természetesen ez csak egy lehetséges megközelítése a problémának. Gyakran egydimenziós textúrákban tárolják a lehetséges rajzolási színeket, és ebbe indexelnek. A 12. ábra egy rajzfilmárnyalt modellt ábrázol spekuláris régióval és anélkül.



(a) (b)

12. ábra. Rajzfilmárnyalt modellek; (a) spekuláris régióval,
(b) többszörös diffúz régiókkal. [10]

5.2.2. Rajzfilm árnyalás implementáció

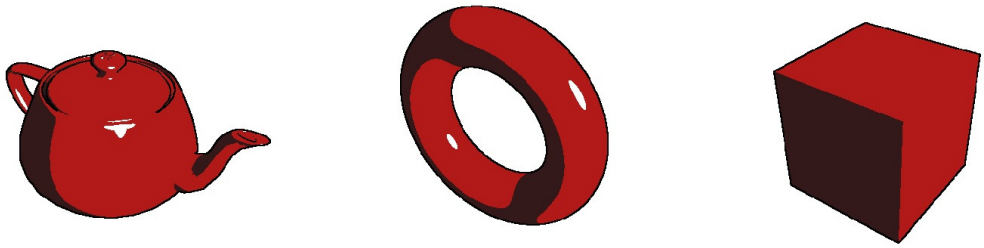
Az itt felvázolt technika három színt használ az objektum ambiens, diffúz és spekuláris megvilágításának szimulálására.

Az implementáció pontszerű fényforrást feltételezve végzi a szükséges számításokat. A megvilágítási számításokhoz használt vektorokat a vertex shader-ben határozzuk meg, amelynek kódja megegyezik a Phong-árnyalásnál bemutatottal.

A fragment shader megkapja a vertex shader-ben meghatározott vektorokat interpolálva, majd meghatározza a diffúz és spekuláris visszaverődésnél alkalmazott belső szorzatokat. Természetesen az interpolált vektorokat felhasználás előtt még normalizálnunk kell. A meghatározott belső szorzatok alapján kiválasztjuk a megfelelő rajzolási színt, majd beállítjuk a fragmentum színét a kiválasztott értékre. A rajzfilm árnyalás eredményét a 13. ábra szemlélteti.

5. KÓDRÉSZLET. Rajzfilm árnyalás fragment shader

```
// vektorok deklarációi
varying vec3 N;
varying vec3 L;
varying vec3 R;
varying vec3 V;

void main()
{
	// a szín tárolására szolgáló változó
	vec4 color;
	// a szükséges belső szorzatok értékeit tároló változók
	float nDotl;
	float vDotr;
	// a szükséges belső szorzatok meghatározása
	nDotl = max(dot(normalize(N), normalize(L)), 0.0);
	vDotr = max(dot(normalize(V), normalize(R)), 0.0);

	// fragmentum színének meghatározása
	// a visszaverődési szögek alapján
	if (nDotl > 0.0)
		color = vec4(0.7, 0.1, 0.1, 1.0); // diffúz szín
	else
		color = vec4(0.2, 0.1, 0.1, 1.0); // ambiens szín

	if (vDotr > 0.97)
		color = vec4(1.0, 1.0, 1.0 ,1.0); // spekuláris szín

	gl_FragColor = color;

}
```



13. ábra. A rajzfilm árnyalás implementációjának eredménye.

5.3. Hatching

A hatching vagy vonalkázás már a középkorban is nagyon népszerű ábrázolási forma volt. A módszer lényege, hogy egymáshoz közel, párhuzamosan elhelyezett vonalak használatával szimulálják az árny- és árnyékhatásokat. A cross-hatching vagy keresztvonalkázás, ahogy a neve is mutatja, bevezeti a metsző vonalakat, ezáltal az árnyalásban és a térfogat érzékeltetésében jobban alkalmazható módszert ad. (lásd 14. ábra)

Mivel az eljárás csak és kizárólag vonalakat használ, a művészek a vonal vastagságát, hosszúságát, a vonalak mennyiségét (sűrűségét), valamint a vonalkázás szögét felhasználva tudnak összetett jeleneteket ábrázolni, fényviszonyokat modellezni.

14. ábra. Klasszikus képzőművészeti hatching-megvalósítás.[6]

[6] http://www-cs.engr.ccny.cuny.edu/~wolberg/projects/engraving/

5.3.1. Hatching értelmezése a komputergrafikában

A hatching komputergrafikai megvalósítása nem könnyű feladat, hiszen nem egyszerű egy háromdimenziós jelenetből olyan vonalas ábrát készíteni, ahol a vonalak a modell formáját követve, a térbeli összefüggőséget megtartva jelennek meg az ábrán. Éppen ezért hatching alatt a számítógépes grafikában legtöbbször egyszerűen vonalas ábrát értünk, és nem teszünk szoros megkötéseket a vonások tulajdonságait (forma, hossz, stb.) illetően. Az így értelmezett definíció elég távol áll a képzőművészeti megközelítéstől, éppen ezért nagyon sokféle módszer született a hatching komputergrafikai implementálásához. Ezek a módszerek, ahogy a 15. ábra is mutatja, teljesen eltérő eredményeket produkálhatnak.

(a) (b)

15. ábra. Különböző hatching megvalósítások; (a) textúrák nélkül [1], (b) textúrákkal [10].

5.3.2. Textúraalapú hatching

A hatching implementálásában a nehézséget a vonalak közötti összefüggés megtartása okozza. Ahogy a képzőművészeti ábrázolásban, úgy a komputergrafikai megvalósításban is teljesülnie kell annak, hogy a vonalak követik a modell formáját. Továbbá figyelnünk kell arra, hogy a vonalak közötti összefüggések megmaradjanak, és hogy – ahol szükséges – az ábrázolt vonalak folyamatosan kapcsolódjanak egymásba.

Ezeket a követelményeket a képtérben megvalósítani lehetetlen, hiszen ott nem állnak rendelkezésre a modell geometriai információi, ezáltal az itt létrehozott vonások nem tudnák követni az objektum alakját.

Egyes módszerek a vonások létrehozását a felület görbültsége alapján pontról pontra végzik el. Azontúl, hogy ezek a módszerek nagyon mesterséges hatást keltenek (nem hasonlítanak a kézzel rajzolt illusztrációkhoz), rettentően számításigényesek, tehát animációban, interaktív alkalmazásokban nem használhatóak.

A legmeggyőzőbb eredményt akkor kapjuk, ha textúrák használatával próbáljuk szimulálni a hatching-et. A textúrák használatával megoldható a szükséges vonalkapcsolatok megtartása, a vonások sűrűségének szabályozása és az alacsony számítási igény fenntartása.

5.3.2.1. A textúrák létrehozása

A textúrák létrehozása sokféleképpen történhet. A legjobb eredményeket a kézzel rajzolt vagy valamely professzionális rajzolóprogrammal készített textúrák használatával kapjuk, de létrehozhatjuk a textúrát algoritmusok alkalmazásával is.

A legtöbb textúraalapú megközelítés a két fő irányú, azaz vízszintes és függőleges, vagy ettől minimálisan eltérő irányú vonásokat alkalmaz a textúrákon, mivel így a legegyszerűbb fenntartani az összefüggőséget. Több textúrát hozunk létre, amelyek különböző tónusoknak felelnek meg. A legvilágosabb tónust reprezentáló textúra rendszerint csak néhány vízszintes vonást tartalmaz, a legsötétebb tónust reprezentáló textúrát sűrű vízszintes és függőleges irányú vonalak keresztezik. Az köztes tónusokat reprezentáló textúrák ezek átmenetei.

5.3.2.1.1. Algoritmikusan generált textúrák

A textúrák automatikus létrehozása leggyakrabban véletlenszerűsítő eljárások alapján történik, azaz a vonásokat véletlenszerűen helyezzük el a textúrát jelentő négyzeten. Ezek a módszerek azonban nagy körültekintést igényelnek, mivel alkalmazásuk a vonások rossz elosztásához vezethet. (lásd 16. ábra)

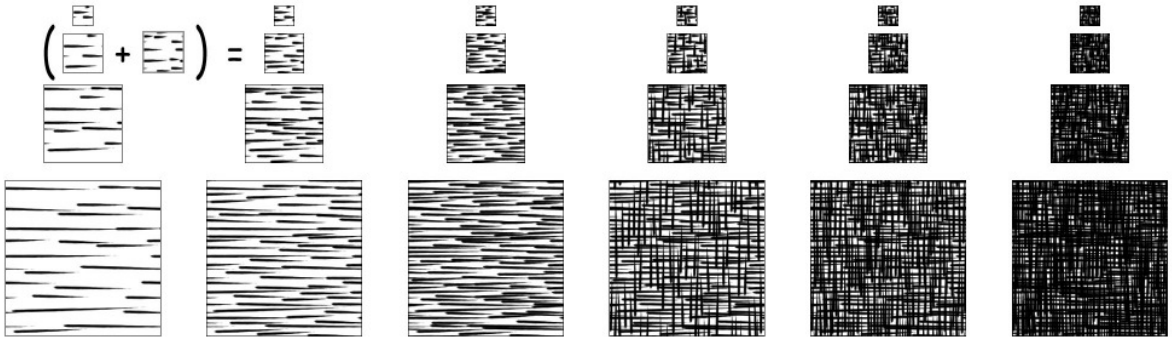
16. ábra. Rosszul illesztett vonások a hatching textúrán. [11]

A megfelelő illeszkedés eléréséhez egyszerre több véletlenszerűen meghatározott helyre kell megpróbálnunk illeszteni a vonást, és a legmegfelelőbb helyre kell elhelyezni. Az elhelyezendő vonások hossza legtöbbször nem egységes, hanem szintén véletlenszerűen kerül meghatározásra.

5.3.2.1.2. Tonal Art Map-ek

Ha a tónusokat reprezentáló textúrákat csak egyetlen méretben hozzuk létre és így alkalmazzuk őket a modelleken, akkor a távolabb lévő modelleken – a kisebb méretű poligonok miatt – a vonások összesűrűsödnek, a modell – az egyébként világos részeken is – sötét tónusba borul.

A tonal art map-ek használata kiküszöböli ezt a problémát. Az alapötlet az, hogy az egyes tónusokat több méretben hozzuk létre, hasonlóan a mip map-ekhez, azzal a különbséggel, hogy a kisebb méretű textúrák a nagyoknak egy kisebb részletét tartalmazzák, nem pedig a nagyok kicsinyített mását. (lásd 17. ábra) A tonal art map-ek használata lehetővé teszi, hogy a modelleken megjelenő vonások sűrűsége a távolságtól függetlenné váljon. [11]



17. ábra. Tonal art map-ek [11]

5.3.2.2. A textúrák alkalmazása

A textúrák és a poligonok között általában nem az egy az egyhez megfeleltetést alkalmazzuk, mivel ez a poligonok határainál nagyon durva átmeneteket eredményezne, akárcsak a konstans árnyalás. A multitextúrázást felhasználva vegyíthetjük a textúrákat a poligonokon. Ehhez végre kell hajtanunk a megvilágítási számításokat a poligon csúcsait

alkotó vertexekre, ez alapján a vertexekhez hozzá kell rendelnünk egy a megfelelő tónust jelképező textúrát, majd a poligon köztes pontjaiban vegyítenünk kell a vertexekhez rendelt textúrákat. A textúrák poligonokhoz, vertexekhez, fragmentumokhoz rendelése, vegyítése sokféleképpen történhet, a legjobb eredményt adó megoldás kiválasztása nagyban függ a használt textúráktól.

5.2.3. Hatching-implementáció

A hatching-implementációnál memóriában létrehozott textúrákat használunk az objektumok textúrázásához, az így létrehozott textúrákat a 18. ábra szemlélteti. A textúrák kiválasztásánál csak a diffúz fényvisszaverődést használjuk fel, így a vertex shader-ben és a fragment shader-ben is kevesebb számítást végzünk. Hogy a fragment shader hozzá tudjon férni a textúrakoordinátákhoz, a vertex shader-ben a textúrakoordinátát a vertexhez kell rendelni.

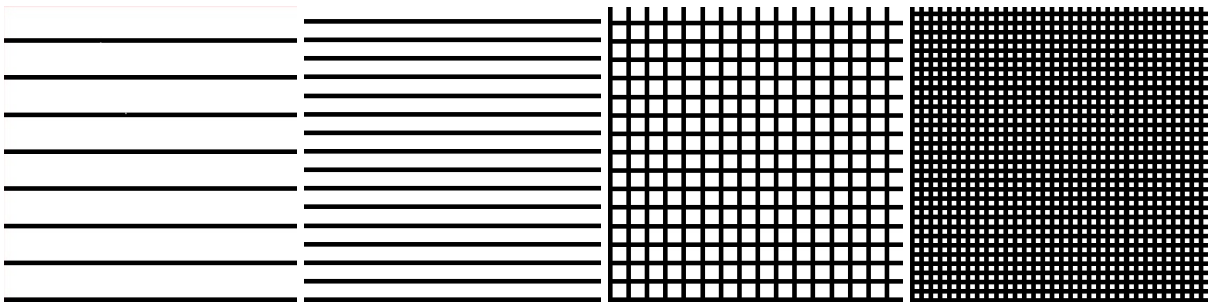
6. KÓDRÉSZLET. Hatching vertex shader

```
// vektorok deklarációi
varying vec3 N;
varying vec3 L;

void main()
{
      // a vertex kamera-koordinátarendszerbeli helye
      vec3 Position = vec3(gl_ModelViewMatrix * gl_Vertex);

      // vektorok meghatározása
      N = gl_NormalMatrix * gl_Normal;
      L = normalize(vec3(gl_LightSource[0].position)-Position);

      // textúrakoordináta beállítása
      gl_TexCoord[0] = gl_MultiTexCoord0;

      gl_Position = ftransform();

}
```

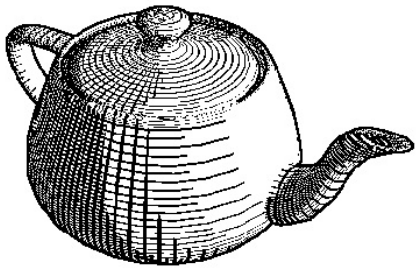
Mivel a megvilágítási számítások a hatching-implementáció esetében is fragmentumonként történnek, az egyes textúrákat nem a vertexekhez rendeljük, hanem a fragmentumra kiszámolt diffúz fényvisszaverődés szerint választunk a fragment shader-nek átadott textúrák közül. A végeredmény a 19. ábrán látható.

7. KÓDRÉSZLET. Hatching fragment shader

```
// vektorok deklarációi
varying vec3 N;
varying vec3 L;

// a különböző tónusokat reprezentáló textúrák
uniform sampler2D sampler0;
uniform sampler2D sampler1;
uniform sampler2D sampler2;
uniform sampler2D sampler3;

void main (void)
{
      // az anyag diffúz fényvisszaverő jellemzője
      vec3 kd = vec3(gl_FrontMaterial.diffuse);

      // a fényforrás diffúz fényvisszaverő jellemzője
      vec3 ld = vec3(gl_LightSource[0].diffuse);

      // a diffúz fényvisszaverődést meghatározó belső szorzat
      float nDotl;

      // a fragmentum színét tároló változó
      vec4 color;

      // a diffúz fényvisszaverődés mértéke
      nDotl = dot(normalize(N), normalize(L));

      // a fargmentum színének meghatározása
      // a diffúz fényvisszaverődés alapján
      if(nDotl > 0.9)
            color = vec4(1.0, 1.0, 1.0, 1.0);
      else if(nDotl > 0.7)
            color = texture2D(sampler0, gl_TexCoord[0].xy);
      else if(nDotl > 0.5)
            color = texture2D(sampler1, gl_TexCoord[0].xy);
      else if(nDotl > 0.2)
            color = texture2D(sampler2, gl_TexCoord[0].xy);
      else
            color = texture2D(sampler3, gl_TexCoord[0].xy);

      gl_FragColor = color;
}
```



18. ábra. A hatching-implementáció által használt textúrák.



19. ábra. A hatching-implementáció eredménye.

5.4. Technikai illusztrációk – Gooch-árnyalás

Az előző fejezetekben olyan módszerekről szóltunk, amelyeknél a művészi információátadást a praktikusság és letisztultság fölé helyeztük. Léteznek olyan illusztrálási formák a hagyományos és a komputergrafikai ábrázolásban, amikor a legapróbb részlet sem maradhat rejtve, ilyenkor nem mindegy, hogy milyen módszert választunk egy adott ábra elkészítéséhez. Különösen igaz ez a technikai illusztrációk esetében, ahol a felismerhető formák, geometriai kapcsolatok, méretek igen kényes megjelenítést igényelnek. Hogy el tudjuk képzelni, mik is azok a technikai illusztrációk, gondoljunk csak a kezelési útmutatókban alkalmazott ábrákra vagy a CAD rendszerekben is alkalmazott megjelenítési módokra. Ezek az alkalmazási területek általában kizárják a fényképes ábrák, fotorealisztikus komputergrafikai megoldások felhasználását, hiszen ezek képtelenek olyan részletességgel visszaadni a geometriai információkat, amilyet az alkalmazási terület megkövetel. Tehát olyan árnyalási modell megadása szükséges a számítógépes alkalmazásokban is, amely a kézzel rajzolt technikai illusztrációk árnyalási modelljét közelíti. A múltban rengeteg kutatás foglalkozott a technikai illusztrálás folyamatának számítógépes automatizálásával.

Az illusztrátorok nagyon különböző technikákat alkalmazhatnak az ábrák készítése során, de még így is megfigyelhetőek bizonyos általánosan alkalmazott fogások, amelyek egyfajta absztrakcióként szolgálhatnak a művelet automatizálásánál.

5.4.1. Gooch-árnyalás

Ahogyan a valós világban, úgy a virtuális térben is a vizuális mélységérzékelés elengedhetetlen kelléke a fény.

A fotorealisztikus megvilágítási modellek segítségével nagyszerűen visszaadható a térbeliség érzése, viszont kevés geometriai információt ad a formák pontos felismeréséhez. A problémát az okozza ezeknél a módszereknél, hogy túl tág az a dinamikai tartomány, amelyet az árnyalásnál felhasználunk, így a görbültségre, kapcsolatokra vonatkozó információk a sötét és világos tartományokban elvesznek, nem követhetőek nyomon. (lásd 20. ábra)

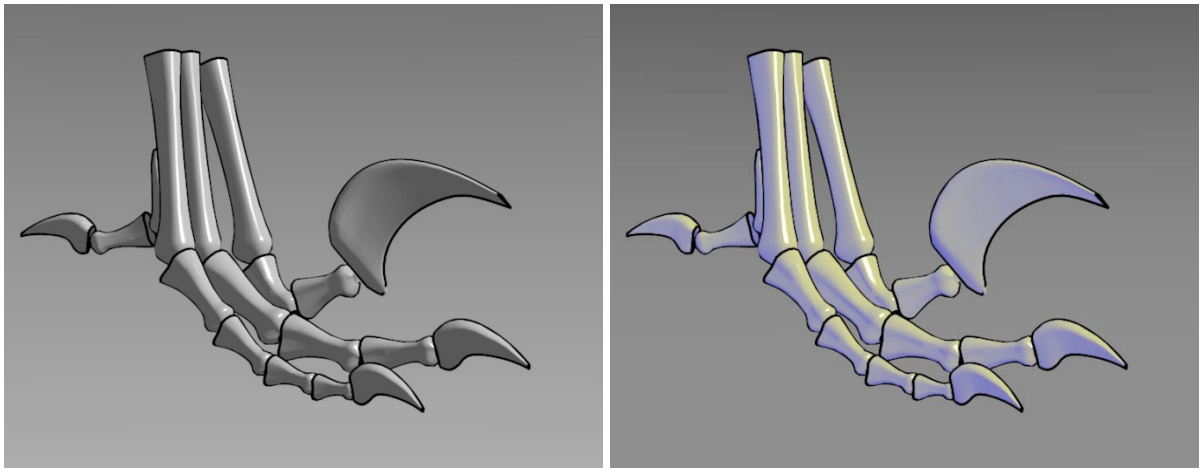
A másik véglet a korábbi fejezetekben is vizsgált nem fotorealisztikus módszerek alkalmazása. Ezek a módszerek a legtöbb esetben túlságosan kevés vizuális információt szolgáltatnak, ez sem kedvez a geometriai felismerhetőségnek. Látható tehát, hogy a technikai illusztrációk 3D-s modellezése igen problémás lenne a hagyományos fotorealisztikus, vagy az előző fejezetekben megismert nem fotorealisztikus módszerekkel, hiszen vagy túl alacsony, vagy túl magas absztrakciós szintet jelentenek a kívánt eredmény eléréséhez.

Az illusztrátorok a technikai ábrák készítésekor számos olyan fogást alkalmaznak, amelyek felfedik előttünk az objektum geometriai kapcsolatait, anélkül, hogy a térbeliség érzése elveszne. A technikai illusztrációk számítógépes készítésére az egyik legmegfelelőbb módszer – a Gooch-árnyalás – még 1998-ból származik. [12, 13]

A módszer a következő szempontokat határozza meg az absztrakció alapjaként, amelyek leginkább a festékszóró pisztollyal és tollal készült ábrákon figyelhetőek meg:

- Az élek rajzolása sötét színnel történik.
- Az objektumok árnyalásakor használt színintenzitások a feketétől és fehértől nagymértékben különböző hideg illetve meleg színek. Ezek adnak információkat a felületi normális irányáról, a felület görbültségéről.
- Egyetlen fényforrást használ, ez fehér színű fényfoltként jelenhet meg a felületen, és segíti a felület formájának követhetőségét.
- Árnyékolás nem szerepel az ábrákon, mert az elfedheti a fontos részeket.

Az így készült illusztrációk matt hatást keltenek, különösen akkor, ha egyáltalán nem használjuk a fényforrás adta spekuláris fényvisszaverődés lehetőségét.



(a) (b)

20. ábra. Karom-modell ábrázolása; (a) Phong-árnyalással, (b) Gooch-árnyalással. [12]

Említettük, hogy a hagyományos módszerek elrejtik a fontos információkat a szélsőségesen sötét illetve világos részekben, ezért az illusztrátorok gyakran nagyobb hangsúlyt fektetnek a színátmenetek használatára a fényerősség-alapú árnyalással szemben. A Gooch-árnyalás technika a színátmeneteket és a fényerőt egyaránt figyelembe veszi az árnyalásnál. Egy hideg-meleg színátmenetet alkalmaz arra alapozva, hogy az emberi észlelés érzékeny a színhőmérsékletekre, így azokat megfelelően alkalmazva a mélységérzékelés visszaadható általuk. A hideg színek (kék, ibolya, zöld) mélyítenek, a meleg színek (vörös, narancs, sárga) emelnek. A legmegfelelőbb hideg-meleg színpárosítás a kék-sárga. A számításhoz felhasznált hideg és meleg komponensek végső értékéhez a felület diffúz színe is hozzájárul egy általunk meghatározott mértékben:

$$k_{hideg} = k_{kék} + \alpha k_d$$

$$k_{meleg} = k_{sárga} + \beta k_d$$

$k_{kék}$ és $k_{sárga}$ az előre megadott kék és sárga színintenzitás, k_d pedig a felület diffúz színkomponense, amit a hideg és meleg színkomponensek rendre az α és β skalárok által megadott mennyiségben vesznek figyelembe.

A felület egy pontjában megjelenő végső színt a következő árnyalási egyenlet adja meg:

$$I = \left(\frac{1+L\cdot N}{2}\right)k_{hideg} + \left(1-\frac{1+L\cdot N}{2}\right)k_{meleg}$$

Látható, hogy ez már egyáltalán nem hasonlít a Phong-módszernél alkalmazott árnyalási egyenlethez. Mivel a belső területek árnyalása közepes intenzitású színekkel történik, az élek

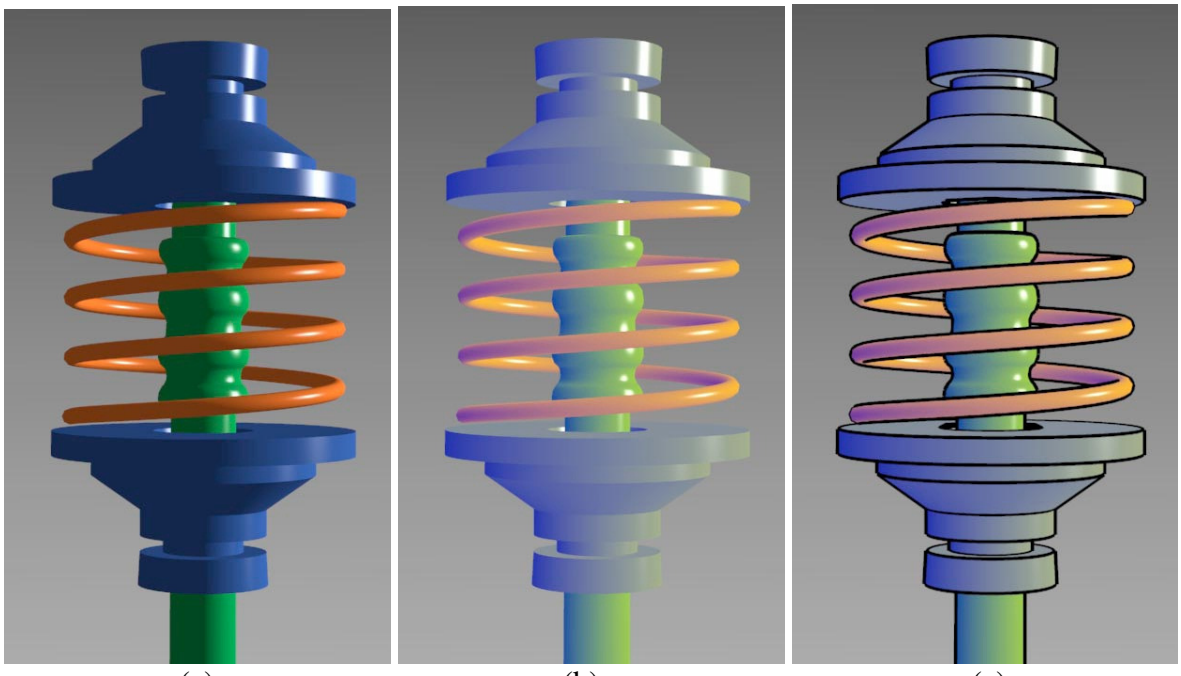
és a spekuláris fényfoltok kitűnnek a maguk sötét, illetve világos árnyalataival. A módszer a fényforrást illetően azt a javaslatot teszi, hogy úgy helyezzük el, hogy a jelenetet valamelyest felülről érje a fény, mivel az emberi érzékelés számára ez természetesebb hatást nyújt.

5.4.1.2. Élek meghatározása

Az élek megjelenítése kritikus a technikai illusztrációk készítésekor, hiszen rengeteg információt hordoznak az anyag formájáról. Mivel a különböző éltípusokat a képtérben nem tudjuk pontosan elkülöníteni, azok meghatározása az objektumtérben javasolt.

5.4.1.3. Spekuláris fényfoltok

A spekuláris visszaverődés megjelenítése szintén fontos a geometriai információk szempontjából. Amennyiben ezt szerepeltetni szeretnénk a jeleneten, az a fotorealisztikus modellek egyikével történhet. A spekuláris fények hatását a technikai illusztrációkon a 21. ábra szemlélteti.



(a) (b) (c)

21. ábra. Alkatrész-modell ábrázolása; (a) Phong-árnyalással,
(b) Gooch-árnyalással élek nélkül, (c) Gooch-árnyalással élekkel. [12]

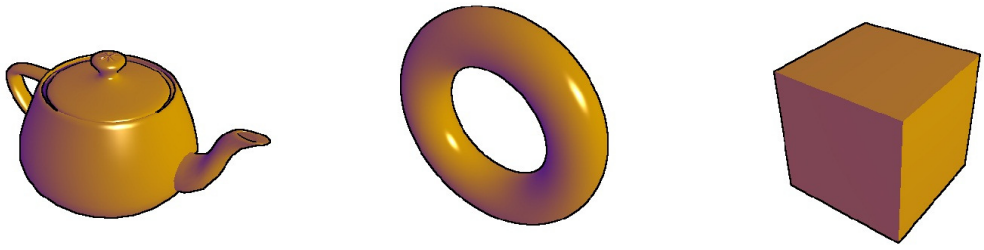
5.4.2. Gooch-árnyalás implementáció

A vertex shader kódja megegyezik a Phong-árnyalásnál bemutatottal. Az árnyalási egyenletben felhasznált súlyok kiválasztása tapasztalati úton történt. Az implementáció a spekuláris visszaverődést is felhasználja a formák kiemelésére. Csakúgy, mint korábban, a shader az alkalmazásban beállított fény- és anyagjellemzőket használja a számításokhoz. Az implementáció eredménye a 22. ábrán látható.

8. KÓDRÉSZLET. Gooch-árnyalás fragment shader

```
// vektorok deklarációi
varying vec3 N;
varying vec3 L;
varying vec3 R;
varying vec3 V;

void main()
{
      // az anyag felhasznált fényvisszaverő jellemzői
      // komponensenként tárolva
      vec3 kd  = vec3(gl_FrontMaterial.diffuse);
      vec3 ks  = vec3(gl_FrontMaterial.specular);
      float s  = gl_FrontMaterial.shininess;

      // a fényforrás által kibocsátott fény spekuláris komponense
      vec3 ls = vec3(gl_LightSource[0].specular);

      // a felhasznált kék és sárga színintenzitások
      vec3 kBlue   = vec3(0.0, 0.0, 0.6);
      vec3 kYellow = vec3(0.6, 0.6, 0.0);

      // a szükséges belső szorzatok értékeit tároló változók
      float nDotl;
      float vDotr;

      // a hideg és meleg színintenzitások tárolására szolgáló változók
      vec3 kCool;
      vec3 kWarm;

      // a diffúz szín figyelembevételét szabályzó súlyok
      float alpha = 0.45;
      float beta  = 0.45;

      float temp;
      // a visszavert intenzitások tárolására szolgáló változók
      vec3 i, is;

      // a felületi színintenzitás meghatározása
      // az árnyalási egyenlet alapján
      nDotl = dot(normalize(N), normalize(L));

      kCool = min(kBlue   + alpha * kd, 1.0);
```

```
        kWarm = min(kYellow + beta  * kd, 1.0);

        temp = (1.0 + nDotl) / 2.0;
        i = temp * kWarm + (1.0 - temp) * kCool;

        // a visszavert spekuláris intenzitás meghatározása
        vDotr = dot(normalize(V), normalize(R));
        is = ks * max((pow(vDotr, s)),0.0) * ls;

        // a spekuláris intenzitás hozzáadása a korábbi "matt" intenzitáshoz
        i = i + is;

        gl_FragColor = vec4(i, 1.0);

}
```



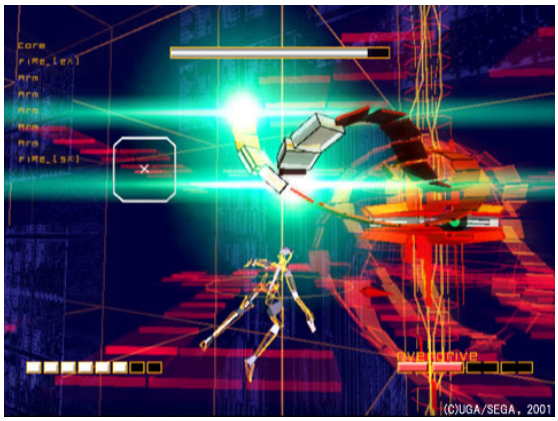
22. ábra. A Gooch-árnyalás implementációjának eredménye.

5.5. Egyéb nem fotorealisztikus technikák

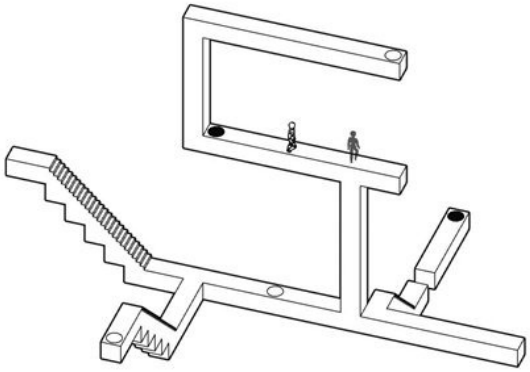
A hagyományos képzőművészet által inspirált nem fotorealisztikus módszerek (vízfesték, stb.) megvalósítása nagyon gyakran a képtérben történik. Számos olyan technika létezik, amelyeknek számítógépes megvalósításában kiváló eredményeket értek el, alkalmazásuk interaktív szoftverekben azonban mégsem terjedt el. Ennek egyik oka a megvalósítás számításigénye, a másik pedig az, hogy alapvetően nem mozgóképes megvalósításra, hanem statikus médiára (papír, festővászon, stb.) tervezték őket, ezért az interaktív alkalmazások, akár videojátékok gyors jelenetváltozásait nem tudják szórakoztató formában közvetíteni.

Az interaktív szoftverek közül a tervezői rendszerek a technikai illusztrációkat, a videojátékok a rajzfilm árnyalást, ezekhez kapcsolódóan pedig még a kontúrrajzolást alkalmazzák. Más nem fotorealisztikus technikák térhódítása az interaktív alkalmazásokban nem túl szerencsés, de azért mindig akad egy-két ötletes próbálkozás a videojáték-iparban. A

23. ábrán olyan videojátékokból láthatunk képeket, amelyek szokatlan képi világukkal próbáltak újítani.



(a)



(b)

(c)

23. ábra. Egyedi grafikájú videojátékok; (a) Rez[7], (b) echochrome[8], (c) MadWorld[9].

[7] http://www.futuregamez.net/ps2games/rez/rez1.jpg
[8] http://static.gamesradar.com/images/mb/GamesRadar/us/Games/E/Echochrome/Bulk%20Viewers/PS3/2008-05-01/UPM101.rev_echo.2--article_image.jpg
[9] http://nintendowiiuk.files.wordpress.com/2008/09/madworld_99.jpg

6. A teljesítményről

Minden interaktív program esetében fontos a teljesítmény. Mivel az implementált árnyalók mind per-fragment végzik a számításokat, a teljesítményellenőrzés méginkább szükséges. Megvizsgáljuk a shader-ekben alkalmazott számításokat, és összevetjük a különböző árnyalók által nyújtott teljesítményt.

6.1. Számítások a shader-ekben

Egy fotorealisztikus (Phong) és négy nem fotorealisztikus (kontúr, rajzfilm, hatching, Gooch) árnyaló került megvalósításra. A kontúrrajzoló shader az egyetlen közöttük, amely a képtérben, és nem az objektumtérben végzi a számításokat. A többi árnyaló az objektumtérben dolgozik, és fragmentumonként végzi a megvilágítási számításokat.

A Phong-, rajzfilm- és Gooch-árnyalóknak két-két belső szorzatot kell kiszámítaniuk a diffúz és spekuláris fényvisszaverődésnek megfelelően, míg a hatching-árnyalónál csak egy belső szorzat kerül kiszámolásra, mivel nem alkalmazza a spekuláris visszaverődést.

A rajzfilm-, Gooch- és hatching-árnyalók alkalmazása után a jelenetre a kontúrárnyaló is végrehajtásra kerül, így ezek alkalmazásánál az éldetektálás költségével is számolnunk kell. Összességében lényegesebb eltérést a számítás sebességében a következő tényezők okozhatnak:

- A Phong-árnyaló esetében nem történik kontúrdetektálás.
- A hatching-árnyaló nem számolja a spekuláris fényvisszaverődést.
- A rajzfilm-árnyaló a számításoknál nem veszi figyelembe (nem alkalmazza) a spekuláris exponenst (hatványkitevőt).
- A teljesítményre hatással lehet még az adatmozgatás (textúrák), valamint az alkalmazott aritmetikai alapműveletek száma.

6.2. Beépített FPS-számláló vs. Fraps

Hogy lássuk, milyen teljesítményt is nyújt a program, implementálásra került egy fps-számláló, amely a képernyő bal alsó sarkában jeleníti meg a másodpercenkénti

képkockaszámot. Az fps-számláló megvalósításához a freeglut által felkínált eszközöket használtam. Az ablak inicializációja óta eltelt idő a *glutGet(GLUT_ELAPSED_TIME)* utasítás segítségével lekérdezhető, így nyilvántartható a két képkocka megjelenítése között eltelt idő. Minden egyes képkocka megjelenítése előtt növeljük eggyel a számlálót, és vizsgáljuk az eltelt időt. Az eltelt időt milliszekundumban kapjuk meg, és minden 1000 milliszekundum után kiírjuk az új fps értéket. Sejthető, hogy a két képkoca megjelenítése között eltelt idő nem pontosan 1 milliszekundum lesz, hanem ennek többszöröse. Ezáltal az egy másodperc elteltét a legutóbbi fps-számváltáshoz képest a

$$\text{legutóbbi\_váltás\_óta\_eltelt\_idő} >= 1000 \text{ milliszekundum}$$

feltétellel vizsgálhatjuk, majd az 1000 milliszekundumra visszaszámolt fps érték a

$$\frac{1000 \text{ milliszekundum}}{\text{legutóbbi\_váltás\_óta\_eltelt\_idő}} * \text{legutóbbi\_váltás\_óta\_mért\_fps}$$

öszefüggéssel megkapható. A képernyőn megjelenő fps értékek várhatóan nem pontosan 1000 milliszekundumonként fog váltani. Az fps-számláló alkalmazásakor persze nem árt figyelembe vennünk, hogy a GPU-n végrehajtott számítások mellett a teljesítmény a CPU terheltségétől is függ.

Hogy lemérjem az fps-számláló pontosságát, a közismert Fraps nevű programot használtam viszonyítási alapként, amelynek sok hasznos funkciója közül csak egyik az fps-számlálás. A Fraps által mért eredmények jól illeszkednek a programba épített számláló által közölt értékekhez. A 24. ábra a mért értékeket mutatja bal felső sarokban a Fraps, bal alsó sarokban a beépített számláló által közölt értékekkel.

24. ábra. Fps-számlálók.

7. A program működése

A program indítása után egy 512x512 képpont méretű ablakot kapunk, amelyben képernyőmenü segítségével választhatunk a megjelenítendő objektumok és a megjelenítés módja között. A menü a jobb egérgomb megnyomásával érhető el. A megjeleníthető objektumok a következők:

- TeaPot (alapértelmezett),
- Torus,
- Cube.

A modellek létrehozása a freeglut megfelelő utasításaival történik. A választható megjelenítési módok:

- Phong (alapértelmezett),
- Cartoon,
- Gooch
- Hatching.

A Phong-árnyalás kivételével mindegyik megjelenítési mód alkalmazza a kontúrokat is.

8. Összefoglalás

A bevezetésben kitűzött célom a komputergrafikában alkalmazott nem fotorealisztikus módszerek ismertetése és a bemutatott módszerekhez valós idejű, fragmentumonként számoló árnyalók implementálása volt. Ennek keretében megvalósításra került a komputergrafikában ma vélhetően leginkább alkalmazott két nem fotorealisztikus megjelenítési mód is. Az egyik a rajzfilm árnyalás, amely ma már a videojátékok jelentős hányadában visszaköszön, a másik a Gooch-árnylás, amelyet elsősorban a háromdimenziós tervezőprogramokban alkalmaznak. Megvalósított megjelenítési mód továbbá a hatching is, amelynek gyakorlati jelentősége – mivel nem terjedt el az alkalmazása – nem túl nagy, a kézzel készített illusztrációk számítógépes reprodukálásában betöltött szerepe azonban említésreméltó. A képfeldolgozási módszerek komputergrafikai alkalmazhatóságát kiaknázva hoztam létre a kontúrokat az említett nem fotorealisztikus árnyalókhoz, hiszen ezek nélkül az árnyalók eredménye csupán részleges lenne. A fotorealisztikus ábrázolás által kínált megjenítést a Phong-árnyaló szemlélteti. Ennek implementálása a különböző árnyalók matematikai modelljeinek, az általuk nyújtott teljesítményeknek és a kapott eredményeknek az összevetését szolgálta.

A teljesítménymérést a megvalósított fps-számláló végzi, amely folyamatosan informálja a felhasználót a program által produkált képkockaszámról. A teljesítményre vonatkozó feltevésem – miszerint a mai grafikus hardverek képesek az árnyalást fragmentumonként, valós időben végrehajtani – helyes volt. Úgy gondolom, hogy a bevezetésben kitűzött célokat sikerült megvalósítanom.

A grafikus hardverek fejlődése a felhasználói igényeknek és a piaci versenynek köszönhetően mind a mai napig gyors ütemben zajlik. A programozható GPU megjelenése vitathatatlanul az egyik legjelentősebb állomása volt ennek a nagy múltra visszatekintő evolúciónak. A programozható, nagyteljesítményű hardverre fejlesztett videojátékok képi világa egyre részletesebb, emiatt gyakran hajlamosak vagyunk megfeledkezni arról, hogy emögött a végletekig optimalizált programkód és a grafikus programozásban alkalmazott trükkök állnak, nem pedig a hardver nyers ereje.

A dolgozatban bemutatott shader-ek fragmentumonként végzik el a megvilágítási számításokat, és úgy tűnik, a modern grafikus hardver ügyesen megbirkózik a feladattal. Azonban a mai kor követelményeinek megfelelő részletgazdag jelenetek fragmentumonkénti előállítása a jelenlegi hardveres támogatással lehetetlen. Mivel a megjelenítők mérete,

felbontása, valamint az ábrázolandó objektumok részletessége a videohardverek teljesítményével párhuzamosan növekszik, úgy gondolom, a jövőben sem lesz megoldható a fragmentumonkénti árnyalás az interaktív alkalmazásokban.

Ábrajegyzék

Irodalomjegyzék

[1] *Randi J. Rost*, „OpenGL® Shading Language, Second Edition”, Addison Wesley Professional, 2006.

[2] *Richard S. Wright, Jr, Benjamin Lipchak, Nicholas Haemel*, „OpenGL superbible : comprehensive tutorial and reference, 4th ed.”, Addison Wesley Professional, 2007.

[3] „The OpenGL Shading Language” http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf

[4] *Szirmay-Kalos László, Antal György, Csonka Ferenc,* „Háromdimenziós grafika, animáció és játékfejlesztés”, ComputerBooks Kft., 2003.

[5] „Diffuse Refelection” http://www.siggraph.org/education/materials/HyperGraph/illumin/diffuse_reflection.htm

[6] „Phong Model for Specular Reflection” http://www.siggraph.org/education/materials/HyperGraph/illumin/specular_highlights/phong_model_specular_reflection.htm

[7] *Aaron Hertzmann*, „Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines”, SIGGRAPH Course Notes, 1999.

[8] *Fazekas Gábor, Hajdu András*, „Képfeldolgozási módszerek”, Debreceni Egyetem, Informatikai Intézet, 2004.

[9] *Jeff Lander*, „Under the Shade of the Rendering Tree”, Game Developer Magazine, vol. 7, no. 2, p17–21, Feb. 2000.

[10] *Wolfgang F. Engel,* „Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks”, Wordware Publishing, 2002.

[11] *Emil Praun, Hugues Hoppe, Matthew Webb, Adam Finkelstein,* „Real-Time Hatching”, SIGGRAPH Proceedings, p581-586, 2001.

[12] *Amy Gooch, Bruce Gooch, Peter Shirley, Elaine Cohen,* „A Non-Photorealistic Lighting Model For Automatic Technical Illustration”, SIGGRAPH Proceedings, p447-452, 1998

[13] *Amy Gooch,* „INTERACTIVE NON-PHOTOREALISTIC TECHNICAL ILLUSTRATION”, Master’s thesis, University of Utah, December 1998.

[14] *Juhász Imre,* „OpenGL”, mobiDIÁK könyvtár, Debreceni Egyetem, 2003.

[15] „GLSL Tutorial”
http://www.lighthouse3d.com/opengl/glsl/