Debreceni Egyetem
Informatika Kar

Shader nyelvek lehetőségei
Felület részleteinek megjelenítése

Témavezető:
Dr. Tornai Róbert
Egyetemi adjunktus

Készítette:
Mile Tamás
Programtervező Matematikus

Debrecen, 2008

TARTALOMJEGYZÉK

Köszönetnyilvánítás

Szeretnék köszönetet mondani Dr. Tornai Róbertnek, aki elvállalta a diplomamunkám témavezetését, és tanácsaival segítette annak elkészítését.

1. Bevezetés

Számos területen szükség van arra, hogy objektumokat, alakzatokat jelenítsünk meg a számítógéppel, legyen az orvostudomány, építészet, szórakoztatás technika, stb. Ezt az igényt próbálja meg kielégíteni a számítógépes grafika, amely a számítógépek teljesítményének növekedésével együtt folyamatosan fejlődött, és egyre komplexebb vizuális hatások megjelenítésére nyílt lehetőség.

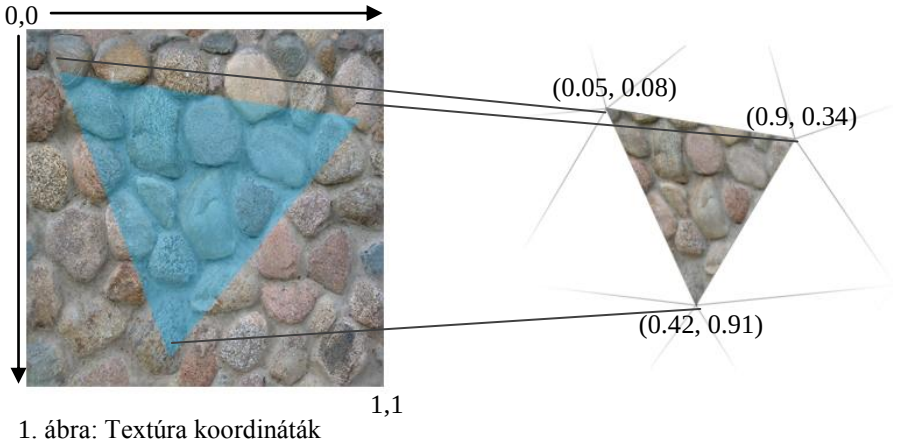
Alapvetően két részre bonthatjuk a számítógépes grafikát:

- 2 dimenziós grafika (2D)
- 3 dimenziós grafika (3D)

2D-s grafikáról akkor beszélünk, amikor az ábrázolandó objektumok 2 dimenziósak (négyzet, kör, stb.) és egy síkban helyezkednek el. Bizonyos esetekben több síkban helyezkednek el egymás mögött, és ha ezt a megjelenítés során próbáljuk érzékeltetni, akkor beszélünk 2.5D-s grafikáról.

3D-s grafika esetén 3 dimenziós objektumokat ábrázolunk (kocka, henger, gúla, stb.). Mivel a jelenlegi számítógépes kijelzők 2 dimenziósak, ezért az objektumokat le kell képezni egy 2 dimenziós vetítési síkra. Ez történhet ortogonálisan – építészet, gépészet – de akár centrálisan is. Ekkor úgy látjuk az objektumot a kijelzőn, mint a valós világban. A megjelenítésig viszont elég sok számítást kell elvégezni, amihez idő kell. Ennek az időnek a nagysága alapján beszélhetünk valós idejű és nem valós idejű megjelenítésről. Nem valós idejű megjelenítésnél a képeket már előre kiszámolják, és ezután összefűzik egy animációba és nincs lehetőség az interaktivitásra. Egy kép kiszámítása akár órákig is eltarthat csúcsteljesítményű számítógépeken. Ilyenek például a 3D-s animációk melyeket speciális szoftverekkel (modellező programok) készítenek. Valós idejű megjelenítésnél a cél az interaktivitás. Azt szeretnék, hogy a kép frissüljön, ahogy megváltoztatjuk a körülményeket, például elforgatjuk az objektumot. Az emberi szem számára a 24 kép/másodperc már folyamatosnak tűnik, bár ennek az elérése nem feltétlen szükséges a valós idejű megjelenítéshez.

A 3D-s grafikához az objektumokat valahogy reprezentálni kell. Erre több lehetőség is van. A legelterjedtebb megoldás hogy a felszínt poligonokkal ábrázoljuk. Ehhez általában háromszögeket használunk és ezekkel a háromszögekkel közelítjük az objektum felszínét. Ezzel egy háromszögrácsot építünk fel, és minél kisebbek a poligonok annál pontosabban közelíti az eredeti objektum alakját. A rács pontjait képezik a háromszögek csúcspontjai, ezeket vertexeknek nevezzük. Ezeket a vertexeket képezzük le a vetítési síkra. Összekötve a pontokat egyenesekkel, megjeleníthetjük az objektumunk alakját 2 dimenziós háromszögekkel. Ezeket a háromszögeket ezután ki kell tölteni. A kijelzők apró képpontokból épülnek fel, más néven pixelekből, és a kitöltés során ezeket a pixeleket színezzük ki. Ez történhet egyetlen színnel is, de akár úgy is, hogy egy előre letárolt mintát használunk fel, úgynevezett textúrákat. A textúrák felhasználásával történő kitöltést nevezzük textúra leképzésnek. A textúra egy adott pontját texelnek nevezzük. Minden texelre hivatkozhatunk koordinátával. Egy textúra bal felső sarka általában a (0,0), míg a jobb alsó sarka az (1,1) koordinátájú texel. Minden vertexhez hozzárendeljük egy-egy textúra koordinátát és ezzel jelezzük, hogy az adott háromszögre a textúra melyik részét kell leképezni.



1. ábra: Textúra koordináták

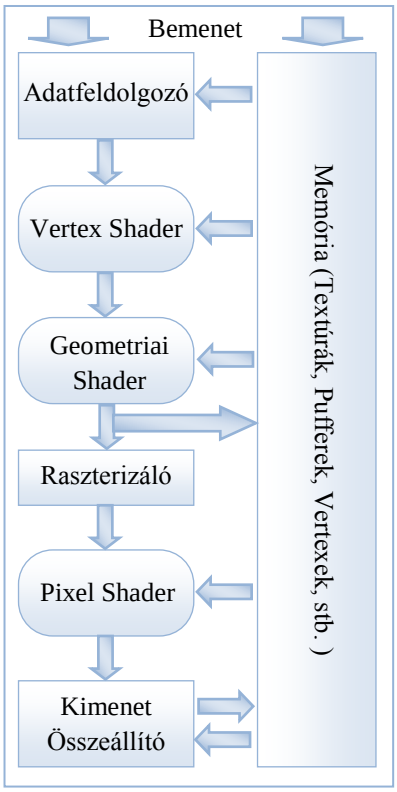
Egy objektumot azért látunk, mert visszaveri a ráeső fény egy részét. A visszavert fény színét már meg tudjuk jelenítni egy diffúz textúra leképzéssel, de az erősségét nem. Mivel a fény tetszőleges szögből érkezhet, ezt nem számolhatjuk ki előre. Vagyis minden egy pixel esetén minden képkockára ki kell számolni a fényerősséget. A vetítések, textúra leképzések, és megvilágítás már akkora számítási teljesítményt igényel, hogy 1996-ban létrejött ez első célhardver (3Dfx, Voodoo 1), melynek a feladata kizárólag a számítógépes grafika megjelenítése. A hardveres gyorsítás lényege hogy egy grafikus processzor (GPU) veszi át számítási terhet a CPU-ról. A GPU és maga a grafikus hardver is a vizuális megjelenítéshez

lett tervezve, ezért ezeket a speciális feladatokat sokkal gyorsabban tudja elvégezni. A hardveres gyorsítónak csak át kell adni a vetítési beállításokat, vertexek adatait, a fényforrások pozícióját és hogy melyik háromszöghöz melyik textúra tartozik és miden szükséges számítást elvégez és a vetítésnek megfelelően megjeleníti az objektumot. Ezt úgynevezett API-n (Application Programming Interface) más néven alkalmazásprogramozási interfészen keresztül tehetjük meg. Jelenleg két különböző API van elterjedve. Az OpenGL és a DirectX. Az OpenGL-t a Silicon Graphics fejlesztette ki 1992-ben, míg a DirectX-et a Microsoft Corporation, és a Windows 95 szerves része volt. A DirectX valójában egy API-gyűjtemény, amelynek csak az egyik része tartozik a 3 dimenziós grafikához.

Kezdetben nem volt lehetőség befolyásolni azt a folyamatot, hogy hogyan végezze el a háromszögek kitöltését a hardver, vagy a hogy hardveresen manipuláljuk a vertexek pozícióját. Alapvető effektek létrehozására volt lehetőség, de ezek eléggé korlátozottak voltak. Az igazi áttörés a DirectX 8.0 megjelenésével jött el. A DirectX 8.0 ugyanis tartalmazta a Shader Model 1.0-át és 1.1-et, amely magában foglalja a Vertex Shader-t és a Pixel Shader-t. Természetesen OpenGL-hez is léteznek a megfelelő kiterjesztések.

2. Shader model

A shader model alapvetően két részre bontható, pixel shader-re és vertex shader-re, de a legújabb hardverekben már a geometriai shader is megjelent. Ezek segítségével utasíthatjuk a GPU-t hogy milyen műveleteket végezzen az adatokkal amíg azok áthaladnak a grafikus futószalagon.



2. ábra: DirectX futószalag

2.1 Vertex shader

A vertex shader írja le a GPU számára, hogy milyen műveleteket kell elvégeznie a vertexeken. A vertex shader minden utasítása, minden vertexen ugyanúgy végrehajtódik. Fontos hogy a vertex shader futása során semmiféle információnk nincs arról, hogy melyik vertexet manipuláljuk vagy, hogy melyik poligonhoz tartozik, és nem érhetjük el más vertexek adatait sem. Minden vertexhez tárolhatunk el

információkat, amelyeket majd később a pixel shader-ben felhasználunk.

2.2 Pixel shader

OpenGL esetén fragment shader-nek nevezik. A pixel shader hasonlóan működik, mint a vertex shader, azzal a különbséggel, hogy itt a poligon kitöltése során érintett pixelek színét és az esetleg hozzá tartozó egyéb értékeket (alfa, Z-érték, stb.) manipulálhatjuk. Itt sincs információnk arról, hogy éppen melyik pixelt manipuláljuk, és nem érhetünk el másik pixelt sem. A vertex shader-től kapott adatokat a hardver interpolálja a vertexek között.

2.3 A shader model evolúciója

Az évek során a shader modell folyamatosan fejlődött. A következő táblázat mutatja az egyes pixel shader verziók tulajdonságait.

Tulajdonság/Verzió	PS 2.0	PS 2.0a	PS 2.0b	PS 3.0	PS 4.0
Textúrák száma	8	korlátlan	8	korlátlan	korlátlan
Textúra műveletek	32	korlátlan	korlátlan	korlátlan	korlátlan
Utasítás helyek	32+64	512	512	≥512	≥65536
Kiterjesztett utasítások	32+64	512	512	65536	korlátlan
Textúra indirekció	4	korlátlan	4	korlátlan	korlátlan
Tároló regiszterek	12	22	32	32	4096
Konstans regiszterek	32	32	32	224	16x4096
Gradiens utasítások	nincs	van	nincs	van	van
Ciklusszámláló regiszterek	nincs	nincs	nincs	van	van
Dinamikus folyamvezérlés	nincs	nincs	nincs	van	van
Bitműveletek	nincs	nincs	nincs	nincs	van

1. táblázat: pixel shader verzió összehasonlítás

A shader model-hez speciális programozási nyelv tartozik (Shading Language). Ez kezdetben a CPU programozáshoz hasonlóan alacsony szintű nyelv volt. Később létrejöttek a különböző magas szintű nyelvek. Az OpenGL esetén GLSH, a DirectX esetén HLSL-ről beszélünk, de az egyik legnagyobb VGA gyártó cég az Nvidia is készített saját nyelvet melyet Cg-nek nevezett el. Ezek a nyelvek a C programozási nyelv szintaxisára épülnek.

A kezdeti verziók nagyon korlátozottak voltak. A 2.0-ás verzió már sok mindenre elég volt, de a DirectX 9.0c-ben megjelenő 3.0-ás verzió volt az első, amivel teljesen szabadon

dolgozhattunk. Itt már nagyságrendekkel bonyolultabb shader-eket írhatunk mint korábbi verziókkal. Lehetőség nyílt szubrutinok írására, valamint ebben a verzióban volt elsőnek elérhető a dinamikus adatfolyam vezérlés (DFC - Dynamic Flow Control). DFC esetén már használhatunk változó kifejezéseket feltételes utasításokban ahol a feltétel kiértékelése után csak a megfelelő utasítások hajtódnak vére. A ciklusokat megszakíthatjuk, illetve írhatunk nem előírt lépésszámú ciklusokat is. Már a 2.0-ás verzióban is használhattunk feltételes utasításokat, de a feltételben csak konstansokat használhattunk.

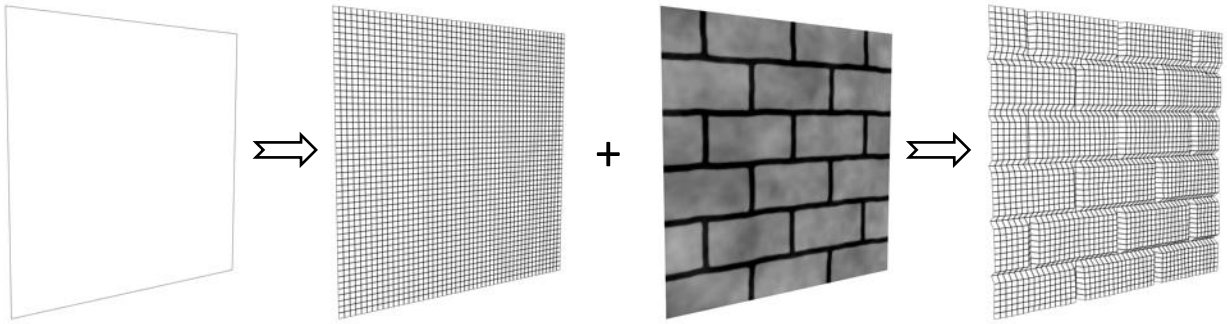
3. Felületek részleteinek megjelenítése

A pixel shader megjelenésével nagy lendületet kapott a számítógépes grafika fejlődése. Az igazán látványos effektek a számítógépes játékokban jelentek meg, és itt volt igazán nagy szükség a valós idejű megjelenítésre. Mint már említettük, az objektumokat poligonokból építjük föl. Minél több poligont használunk annál részletesebb képet tudnunk létrehozni. A poligonszám növelésével viszont egyre több számítási teljesítményt igényel és egyre távolabb kerülünk a valós idejű megjelenítéstől. Egy objektum felületén általában olyan apró kiemelkedések is vannak, aminek a megjelenítéséhez túlságosan sok poligonra lenne szükség. Gondoljunk csak egy fal felszínére. Ha csak egy falat szeretnénk megjeleníteni akkor egy téglatestet kell modelleznünk, amit akár 2*6 poligonnal (minden oldalhoz 2 háromszöget használunk) leírhatunk. Viszont egy fal felület érdes. Számtalan sok kis bemélyedés és kiemelkedés van a felszínén. Ha ezt az egyenetlenséget poligonokkal szeretnénk leírni, akkor több ezer, de akár millió poligonra is szükségünk lehet. Habár a mai videokártyák, rendkívül nagy teljesítményűek, ekkora poligonszám mellett a valós megjelenítés lehetetlen, és az adatok tárolása nagy memória mennyiséget kívánna meg. A másik probléma hogy az objektumokat leíró poligonokat valahogy be kell vinni a számítógépbe és erre gyakran csak az ember képes (léteznek már eszközök, amik képesek letapogatni az objektum felszínét és létrehozni a poligonrácsot teljesen automatikusan). Több ezer poligon betáplálása rendkívül időigényes.

3.1 Displacement mapping

Ennek a problémának a megoldására született meg a displacement mapping technika [1]:

- Letároljuk a kiemelkedések nagyságát. Leggyakoribb megoldás hogy egy képen tároljuk, amit mélység- vagy magasságtérképnek nevezünk. Azok a részek, amik kiemelkednek a felszínről világosabbak, amik pedig bemélyednek, azok sötétebbek. Egy ilyen képet akár egy képszerkesztővel megrajzolhatunk.
- Létrehozzuk az objektumot leíró kis részletességű poligon rácsot.
- A poligonokat úgynevezett mikro poligonokra osztjuk fel.
- A magasságtérképet a ráfeszítjük a kis részletességű poligonrácsra.
- Minden egyes mikro poligonhoz tartozó vertexet eltolunk a felület normálvektorának irányába, a hozzá legközelebbi magasságtérkép értéknek megfelelően.



3. ábra: a displacement mapping folyamata

Ezt a módszert gyakran használják 3 dimenziós animációk készítéshez, a modellező programokban. Alacsony poligonszám mellett alkalmazható valós idejű megjelenítésben is, például egy táj domborzatának kialakításakor is.

3.2 Bump mapping

A bump mapping a felületek érdességének megjelenítésére szolgál. Az technika azon az elven alapul, hogy a felület lévő apró kiemelkedéseit és bemélyedéseit a fényforrás különböző mértékben világítja meg. A felület valódi eltorzítása helyett csak a felületre érkező fényt számoljuk ki. Ez a technika is a magasságtérképet használ. A szomszédos pontok magasságából kiszámítható az adott pontban a felület meredeksége.

3.3 Emboss bump mapping

Bump mapping technika során a pixelek színét változtatjuk. A pixel shader megjelenéséig azonban nem volt lehetőség a pixeleket egyenként manipulálni. A bump mapping-ot azokkal az alapvető textúra műveletekkel kellett megoldani, amit pixel shader-t nem támogató grafikus kártyák nyújtottak. Az emboss pass bump mapping [2] egy olyan technika, amivel bizonyos szintig szimulálhatjuk a felületet érdességét akár pixel shader nélkül is. A bump mapping esetén ki kellene számolni pixelenként a felület meredekségét és a fény beesési szöge alapján meghatározni a fényerősséget. Ez a technika közvetlenül határozza meg a fényerősséget.

- Legyen M_0 a magasságtérkép az eredeti (u,v) textúra koordinátákkal
- Legyen M_1 a magasságtérkép a fény irányában eltolt ($u+\Delta u$, $v+\Delta v$) textúra koordinátákkal. A fényt be kell forgatnunk a poligon tangens terébe. A poligon tangens tere egy olyan 3 dimenziós derékszögű koordináta rendszer, ahol az egyik tengely a poligon normálvektora, a másik egy poligonnal párhuzamos tangensvektor, és a harmadik tengely a binormálvektor. Ez minden vertex esetén más és előre ki van számolva.
- Az M_1-M_0 különbségéből megkapjuk az fényerősséget
- A fényerősséget összeszorozzuk a diffúz texturával.

4. ábra: Emboss bump mapping

Előnyök:

- A legtöbb videokártyán implementálható.
- Gyors
- A magasságtérképet tárolhatjuk a diffúz textúra alfa csatornájában

Hátrányok:

- Mivel nem számoljuk ki a felület tényleges meredekségét, ezért nincs lehetőség spekuláris fény megjelenítésére (az fény, ami a felületről közvetlenül a szembe tükröződik)
- Csak éles átmeneteknél ad jó eredményt (fa repedései, téglafalak élei, stb.).

3.4 Normal mapping

A normal mapping [3] egy olyan bump mapping technika ahol már a felület normálvektorát használjuk fel a számítások során. Viszont ahelyett hogy a mélységi térképből számolnánk ki a normálvektorokat, olyan textúrát készítünk, amely már tartalmazza őket, ezért ehhez a technikához nincs szükség magasságtérképre. A normálvektorokat, a poligonhoz tartozó tangens térben reprezentáljuk. Mivel egy 3 dimenziós vektor megadásához 3 paraméter (x,y,z) szükséges, ezért egy 3 komponensű képen (RGB) tároljuk a komponensek értékeit. Ezt a képet normáltérképnek nevezzük. Viszont egy ilyen képen egy komponens értékes 0 és 255 közé esik. A normalizált normálvektor koordinátái pedig -1 és +1 közi esnek. Ezért alkalmazzuk a következő formulát a vektor koordinátáinak rgb komponensé alakításához:

r = (x+1) / 2 * 255, g = (y+1) / 2 * 255, b = (z+1) / 2 * 255

A (0,0,1) vektor rgb reprezentációja pl. (128,128,255) lesz.

A normal mapping technika implementálásához már pixel shader szükséges. Mindegy egyes pixel esetén normáltérkép alapján kiszámítjuk a fényerősséget. A pixel shader adatokat vár a vertex shader-től. Ezeket az adatokat egy struktúrába rendezve adjuk át. A normal mapping esetén a következő adatokra van szükségünk vertexenként

- textúra koordináta (UV)
- fényforrásba mutató vektor a tangens térben (Lts)

Ezeket az adatokat tehát a vertex shader-ben számoljuk ki. A vertex shader kimeneti struktúráját nevezzük VSOUT-nak. A vertex shader az adatokat főprogramtól kapja. Ezt a továbbiakban INPUT-tal jelöljük a kódban. A vertex shader pseudo kódban:

```
VSOUT.UV := INPUT.textúra_koordináta
tangenstér mátrix feléptése
NTB := { INPUT.normál, INPUT.tangens, INPUT.binormal}
a világítási vektor kiszámítása.
L := INPUT.fényforrás_pozíció - INPUT.vertex_pozíció
L := Normalizál( L )
VSOUT.Lts := NTB * L
```

A pixel shader megkapja a VSOUT struktúrát. A hardver interpolálni fogja ezeket az adatokat a vertexek között. A pixel shader pseudo kódban:

```
az aktuális pixelhez tartozó normálvektor
Nts := TextúraMintavételező( normáltérkép, VSOUT.UV ) * 2 -1
a fényerősség meghatározása
diffúz_fényerő := SkalárisSzorzat( Nts, VSOUT.Lts )
Mivel a skaláris szorzat negatív is lehet, ezért a negatív értékeket levágjuk
diffúz_fényerő := Max( 0, fényerő )
az aktuális pixelhez tartozó UV koordinátájú diffúz textúra texelje
diffúz_szín := TextúraMintavételező( diffúz_textúra, VSOUT.UV )
PSOUT.szín := diffúz_szín * diffúz_fényerő * fény_szín
```

A PSOUT a pixel shader kimeneti struktúrája, ami jelen esetben csak egy szín komponensből áll. A valóságban mindig érkezik valamennyi szórt fény a külvilágból. Ezek az objektum körül lévő többi objektumról verődnek vissza és szóródnak szét. Ennek a fénynek a szimulálásához bevezetjük a környezeti fényt. Ez a felület minden pontjára azonos erősséggel hat, függetlenül a felület fekvésétől. A módosított pixel shader:

```
…
diffúz_szín := TextúraMintavételező( diffúz_textúra, VSOUT.UV )
PSOUT.szín := diffúz_szín * (diffúz_fényerő + környezeti_fényerő ) * fény_szín
```

Az emboss pass bump mapping technikánál nem számolhattunk spekuláris fényt. A normáltérkép segítségével viszont egyszerűen kiszámíthatjuk, mivel minden szükséges információ a rendelkezésünkre áll. A spekuláris fény számításához többféle modell is létezik. Mi most a Blinn-Phong modell szerint fogunk számolni.

$k_{spec} = (\vec{N} \cdot \vec{H})^E$, ahol $\vec{N}$ a felület normálvektora, és $\vec{H} = \vec{L}+\vec{V}$, ahol a $\vec{V}$ vektor a szembe mutató vektor és $\vec{N}$ vektor a fényforrásba mutató vektor. $E \geq 1$ a Phong kitevő.

A spekuláris fény implementálásához tehát szükségünk van a nézeti vektorra. Bővíteni kell a VSOUT struktúrát.

- nézőpontba mutató vektor a tangens térben (Vts)

A vertex shader-t is bővíteni kell.

```
a nézeti vektor kiszámítása.
V := INPUT.néző_pozíció - INPUT.vertex_pozíció
V := Normalizál( V )
VSOUT.Vts := NTB * V
```

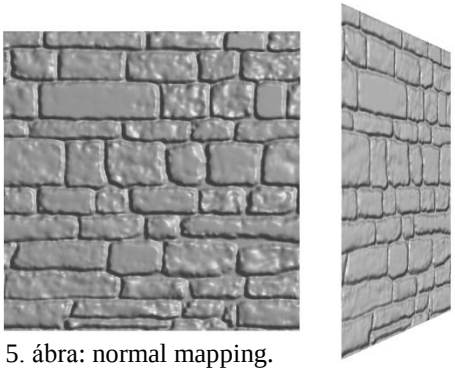
Ezek után már meg kell írni a pixel shader-ben a spekuláris fény kiszámítását

```
…
diffúz_fényerő := Max( 0, fényerő )
spekuláris fényerő kiszámítása
H := VSOUT.Vts + VSOUT.Lts
spekuláris_fényerő := SkalárisSzorzat( Nts, H );
spekuláris_fényerő := Max( 0, spekuláris_fényerő )
az aktuális pixelhez tartozó UV koordinátájú diffúz textúra texelje
diffúz_szín := TextúraMintavételező( diffúz_textúra, VSOUT.UV )
PSOUT.szín := diffúz_szín * ( fényerő + környezeti_fényerő ) * fény_szín
PSOUT.szín := PSOUT.szín + spekuláris_fényerő
```

A normál textúra segítségével nem a csak a szórt és a szembe visszaverődő fényt számolhatjuk ki. Használható tükröződések és fénytörések megjelenítéséhez is. A normal mapping már nem csak az érdesség megjelenítése képes, hanem minden olyan részlet megjelenítésére, amit poligonokkal túlságosan költséges lenne megoldani.

Előnyök:

- Nem korlátozza a megjelenítés minőségit a felület típusa.
- Nem igényel nagy számítási teljesítményt
- Már Shader Modell 2.0-ban is implementálható



5. ábra: normal mapping.

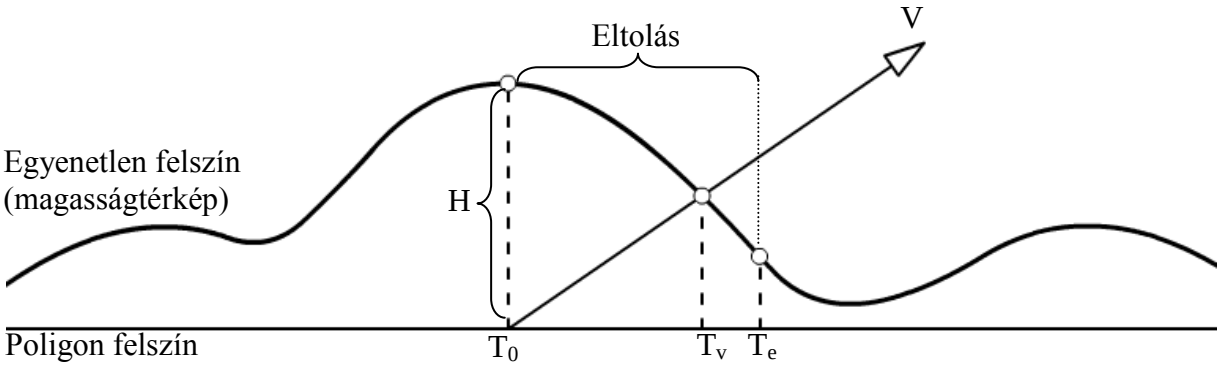
Hátrány:

- Minden modellhez el kell készíteni a normáltérképeket

Ez a technika szinte tökéletes jeleníti meg a felületet, ha az szemben helyezkedik el (5. ábra, bal oldali kép) a szemlélővel. Viszont ha csökkenjük a betekintési szöget láthatóvá válik (5. ábra, jobb oldali kép), hogy mégis csak egy lapos felületet nézünk.

3.5 Paralax mapping

A paralax mapping [4] a normal mapping egy módosított változata. A technika lényege, hogy a magasság és nézőpont függvényében módosítjuk a textúra koordinátákat. Ezzel próbáljuk meg azt a hatást elérni mintha valóban kidomborodna vagy benyomódna a felület. A magasabb területek a nézőponttól elfele, míg az alacsonyabbak a nézőpont irányába toldónak. Minden pixelhez külön eltolási értéket számolunk. Az eltolás iránya a szembe mutató vektor vetülete lesz a poligon síkjára. Valójában nem azt számoljuk ki, hogy az adott textúra texelt hol kellene látni, mert - ahogy a pixel shader leírásában említettük -, mi csak az aktuális pixelt manipulálhatjuk. A feladat tehát az, hogy meghatározzuk, hogy az aktuális pixelnél melyik texelt kellene látni.



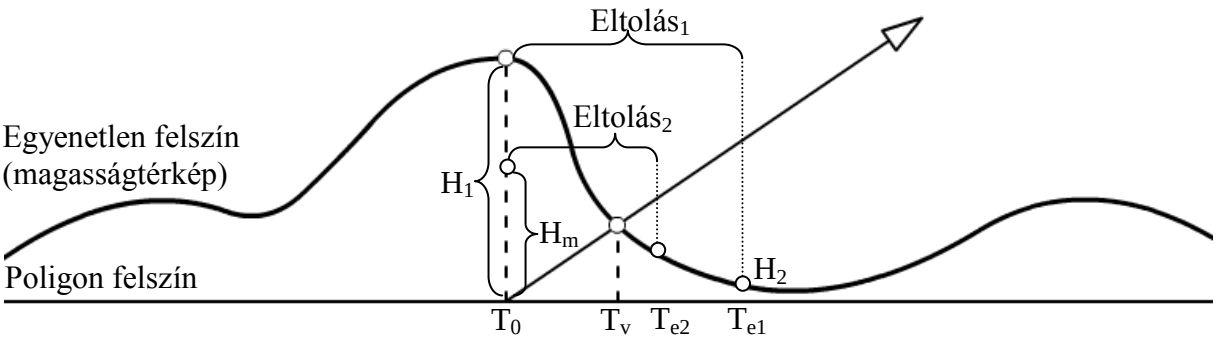
6. ábra: V a nézőpontba mutató vektor, T_0 Az eredeti textúra koordináta, T_v annak a texelnek a koordinátája mit látnunk kellene. H az eredeti textúra koordinátához tartozó magasságtérkép érték. T_e a módosított koordinátához tartozó texel

A normal mapping-hoz nem használtunk magasság térképet, mert csak a fényerősséget számoltuk és ehhez elegendő volt a felület pontjainak normal vektora. Most egy magasság térképre is szükségünk lesz. Az emboss technika esetén a diffúz textúra alfa rétegében tároltuk. Ezt most is megtehetnénk, de célszerű a normáltérkép alfa csatornájában tárolni. Ennek oka, hogy a számítások során a magasság értékre és a normálvektorra egyszerre van szükségünk, és ha az értékek egy képen vannak tárolva, akkor egy művelettel lekérdezhetjük őket. Adott tehát a normáltérkép, amely alfa csatornája tartalmazza a magasság térképet. Mivel a normal mapping módosításáról van szó a vertex shader nem is változik. Csak a pixel shader-t módosítjuk. Mielőtt ezt megtennék, még pár dolgot át kell gondolni. A textúra

koordináták 0 és 1 közé vannak skálázva, mint ahogy a magasságtérkép értékei is. Viszont egy textúra akár egy 2x2 méteres felületet is lefedhet. Ebben az esetben egy 0.5 magasságú kiemelkedés fél méteres lenne. Ezért be kell vezetnünk egy skálázási faktort, amivel visszaalakítjuk a magasságtérkép értékeit reális tartományba. Ha 2x2 méteres példánál maradunk, és a legnagyobb kiemelkedés mondjuk 0.02m, akkor a skálázási faktor 0.02 / 2 = 0.01. A másik dolog amire figyelni kell, hogy milyen magasságértéknél mondjuk azt egy felületről hogy be van nyomódva. Ha csak vesszük a magasságtérképet, akkor egy pont vagy a poligon felszínén van vagy ki van emelkedve mivel a magasságértékek 0 és 1 között vannak. Vezessünk be egy mélységi eltolást. Ezután a negatív magasságérték azt jelenti, hogy az adott rész be van nyomódva, míg a pozitív azt hogy ki van emelkedve.

```
skálázási_faktor := 0.01
mélységi eltolás, a -0.5 a legtöbb esetben ideális
mélységi_eltolás := -0.5
az eltolás iránya megegyezik a szembe mutató vektor x és y koordinátájával mivel a
tangens térben ez éppen a Vts vetülete poligon síkjára.
eltolási_irány := (x,y)
az aktuális pixelhez tartozó magasságérték
H := TextúraMintavételező( normáltérkép, VSOUT.UV )
uv_eltolás := eltolási_irány * ( H + mélységi_eltolás) *  skálázási_faktor
a pixel shader többi része annyiban módosul, hogy VSOUT.UV helyett mindenhol
VSOUT.UV + uv_eltolás fog szerepelni
```

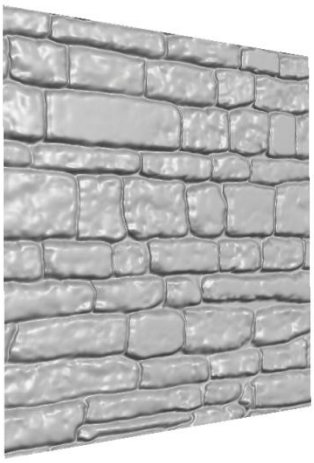
Az eltolás mértékén lehet pontosítani. A probléma ugyanis az, hogy mi csak közelítjük a T_V pontot. Ez a technika meredekebb részeken nagyon pontatlan eredményt ad (7. ábra).

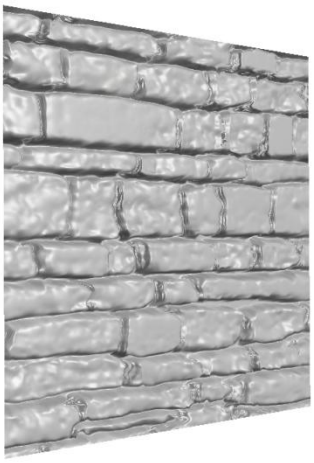


7. ábra: Az eltolás pontosítása. H_m a H_1 és H_2 magasságok átlaga

Vegyük tehát az eredeti és az eltolt textúra koordinátákhoz tartozó magasságértékek átlagát és ezzel a magassággal újraszámoljuk eltolást. Ezt a lépés többször végrehajtva pontosabb eredményt kapunk.

```
…
eltolási_irány := (x,y)
az aktuális pixelhez tartozó magasságérték
H1 := TextúraMintavételező( magasságtérkép, VSOUT.UV )
uv_eltolás := eltolási_irány * ( H + mélységi_eltolás) *  skálázási_faktor
az iterációk száma
k := 3
AMÍG  k<3
        H2 := TextúraMintavételező( magasságtérkép, VSOUT.UV + uv_eltolás)
        H1 := ( H1 + H2 ) / 2
        uv_eltolás := eltolási_irány * ( H1 + mélységi_eltolás) *  skálázási_faktor
VÉGE
…
```





8.ábra: a.) 1 iteráció skálázási faktor = 0.04 b.) 3 iteráció skálázási faktor = 0.04 c.) 3 iteráció skálázási faktor = 0.15

Előny:

- Csak néhány művelettel végzünk többet mint a normal mapping-nál és szebb eredményt kapunk.
- Pixel Shader 2.0-val megvalósítható 2 iterációig.

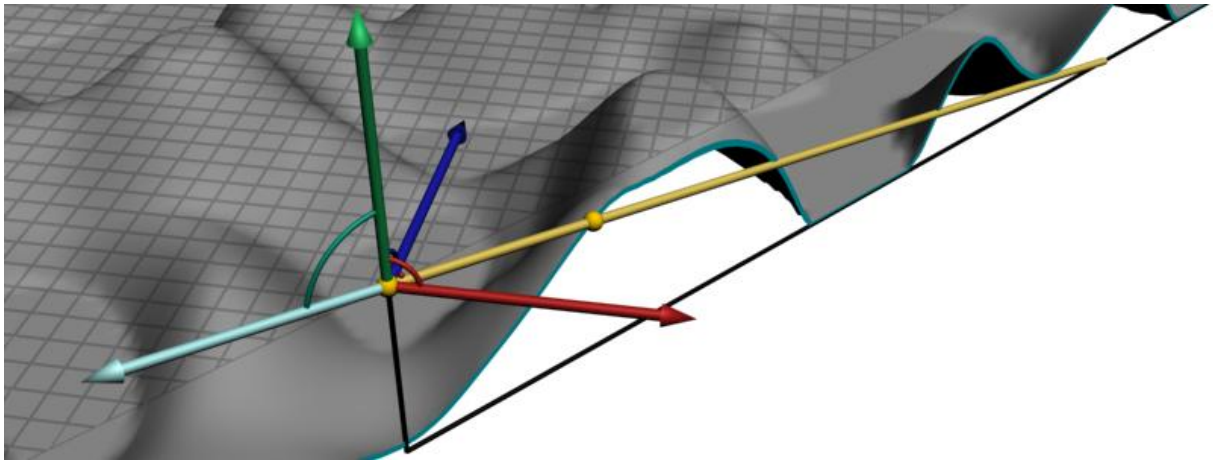
Hátrány:

- Csak egy bizonyos skálázási faktorig működik jól és ez függ a magasság térkép milyenségétől. Éles kiemelkedések esetén könnyen kaphatunk hibás eredményt, mivel nem vesszük figyelembe, hogy az adott pixel nincs-e takarásban (8. ábra c.).

Az emboss pass bump mapping esetén a fényerősséget nem korrekten számoltuk. Ezt orvosolta a normal mapping. A paralax mapping a felületek valódi kidomborodását szimulálja, nem korrektül. Ha teljesen helyesen szeretnénk megjeleníteni a felületek kiemelkedésit, akkor egy új technikára a relief mapping-ra van szükségünk.

3.6 Relief mapping

A relief mapping [5] már nem egyszerű bump mapping technika. Nem csak árnyaljuk a pixeleket és eltolásokat számítunk, hanem gyakorlatilag egyenként kiszámítjuk, hogy az adott pixelnél melyik texel látszik és melyik nem, ezáltal a magasabb kiemelkedések eltakarják a mögöttük lévő pontokat. A látható texel megtalálásához sugárkövetést használunk. Ez azt jelenti, hogy egy adott pontból egy egyenes mentén elindulunk (egy sugarat követünk) és keressük, hogy az egyenes hol metszi a felületet. A magasságtérképet úgy tekintjük, mintha a sima poligon felett vagy alatt helyezkedne el. Mind a két megvalósítás használható, mi az utóbit fogjuk használni. A sugarat amit követni fogunk és ami mentén a metszéspontot keresni fogjuk, a szemből indítjuk az aktuális pixelhez tartozó poligon felszínén lévő pontba. Ezt a pontot, belépési pontnak fogjuk nevezni és P_0-al jelöljük. A metszéspontot jelöljük P_e-vel.



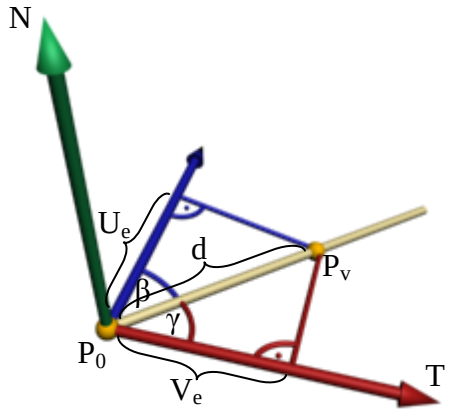
9. ábra: 3D-s metszet a vizsgált részről. A belépési pontból induló egyenes (sárga). A zöld vektor a normálvektor (N), a piros a binormálvektor (B) és a kék a tangensvektor (T). A világoskék vektor a nézőpontba mutat (V).

A szem számára a metszéspont lesz látható az adott pixelnél. Ennek a pontnak a színét, az eddigiekhez hasonlóan a hozzá tartozó diffúz textúra texele, és a fényforrás határozza meg. A diffúz textúra texelének a koordinátája belépési ponthoz tarozó textúra koordináta eltoltja lesz. A tangens és binormálvektorok épen a texturához tartozó koordináta rendszert alkotják. Meg kell meghatározni az (U_e,V_e) eltolást. Tudjuk, hogy (10. ábra)

$$\text{Cos}(\beta)=\frac{U_e}{d}$$

és

$$\text{Cos}(\gamma)=\frac{V_e}{d}$$



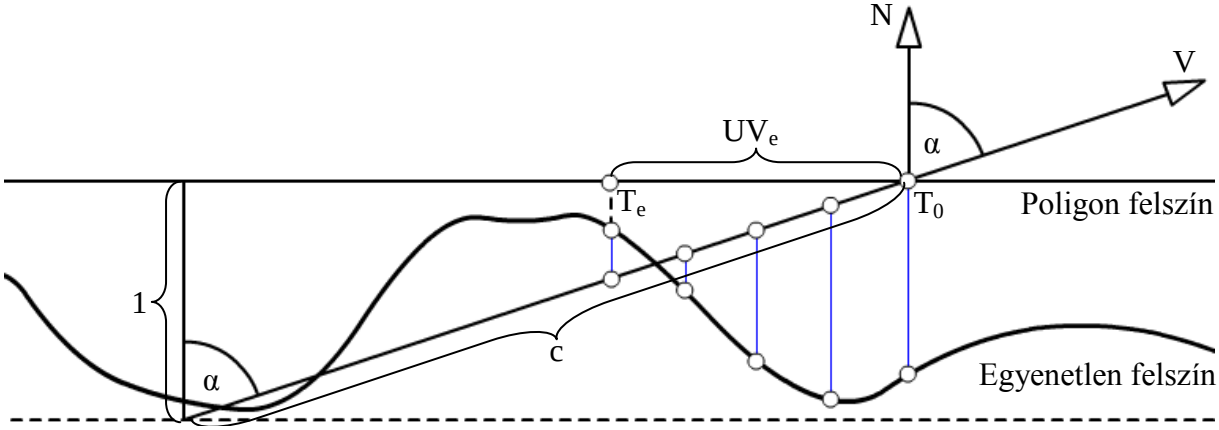
10. ábra

A β és γ értékeire nincs szükségünk hiszen egyből ki tudjuk számolni a Cos(β) és Cos(γ) értékeit. A sugarunk iránya pontosan megegyezik a szemből a belépési pontba mutató vektor irányával ezért a megfelelő skaláris szorzatokkal megkapjuk Cos(β) és Cos(γ) értékeit.

$$\text{Cos}(\beta) = \vec{V} \cdot \vec{T} \quad \rightarrow \quad \vec{V} \cdot \vec{T} = \frac{U_e}{d} \quad \rightarrow \quad (1)\ U_e = d \times (\vec{V} \cdot \vec{T})$$

$$\text{Cod}(\gamma) = \vec{V} \cdot \vec{B} \quad \rightarrow \quad \vec{V} \cdot \vec{B} = \frac{V_e}{d} \quad \rightarrow \quad (2)\ V_e = d \times (\vec{V} \cdot \vec{B})$$

Már csak a d értékére van szükségünk, ami a metszéspont távolsága a belépési ponttól. Ehhez meg kell határoznunk a metszéspontot és itt használjuk a sugárkövetést. Mivel az egyenetlen felszín csak a magasságtérképen létezik, mintavételezéssel kell megkeresnünk a metszéspontot. Egy kezdeti közelítő megoldás megtalálásához a legegyszerűbb módszer a lineáris keresés.



11. ábra: metszet a sugárkövetésről. T_0 a belépési ponthoz tartozó textúra koordináta, T_e az eredményül kapott metszésponthoz tartozó textúra koordináta, UV_e az eltolás mértéke.

3.6.1 Relief mapping lineáris kereséssel

Mivel minden szükséges pont egy síkban helyezkedik el, ezért visszavezetjük a keresést 2 dimenzióba (11. ábra). A szaggatott vonal jelöli a magasságtérkép alját. Ez a poligontól 1 egység távolságra van, mivel a magasságtérkép értékei 0 és 1 között vannak tárolva. A maximális távolság, amin belül szeretnénk megtalálni a metszéspontot c. A lineáris kereséshez osszuk fel ezt a távolságot n egyenlő részre. Így n+1 osztópontot (n_0,…,n_n) kapunk ahol $P_0=n_0$. Induljunk a n_0 pontból és sorban minden osztópontra vizsgáljuk meg, hogy az adott osztópont a magasság térkép fölött vagy alatt helyezkedik el. Ha alatta, akkor megállunk és legyen ez a pont a metszéspont. Ez ugyan nem a pontos megoldás, de az osztópontnak az aktuális és az azt megelőző pont között kell lennie, ha az n elég nagy. Ha az osztópontok száma kicsit, akkor megeshet, hogy átugrunk egy éles kiemelkedésen és hibás eredményt kapunk. Az m. osztóponthoz tartozó magasságot jelöljük h_m-el. Ez nem lesz más, mint (1/n)*m mivel a magasságértékek [0,1] intervallumban vannak és n osztópont esetén, ha lépünk egyet a sugáron, akkor 1/n-et lépünk a magasságtérkép szintjén. Legyen a metszésponthoz tartozó magasságérték h_e. Ekkor

$$d = h_e / Cos(\alpha)$$

Az (1) és (2) képletben d helyére behelyettesítve megkapjuk U_e -t és V_e -t. Mivel a magasság térkép a poligon alatt helyezkedik el, ki kellene vonni 1-ből a magasság térkép értékeit, mivel most az a pont ami magasabban van közelebb lesz a poligonhoz. Ezért ennél a technikánál célszerű a magasságtérképben az 1-ből kivont értékeket tárolni. A magasságtérképet épen ezért nevezhetjük mélységtérképnek is. A paralax mapping-hoz hasonlóan itt is szükségünk lesz egy skálázási faktorra a reális megjelenítéshez és itt is használhatunk mélységi eltolást. Ehhez a technikához a pixel shader-nek a eddigiekhez képest következő adatokra van szüksége:

- nézőpontba mutató vektor (V)
- fényforrásba mutató vektor (L)
- normálvektor (N)
- tangensvektor (T)
- binormálvektor (B)

Az utóbbi háromra azért van szükség, hogy kiszámoljuk a szükséges koszinusz értékeket. Hogy miért nem számoljuk ki már a vertex shader-ben és adjuk egyből az eredményt? Ahogy már említve volt, a pixel shader során a vertexekről kapott adatokat lineárisan interpolálja a hardver. A lineárisan interpolált vektorok skaláris szorzata nem egyezik meg a skaláris szorzatok közötti lineáris interpolációjával, mivel a koszinusz nem lineáris függvény. A vertex shader tehát a következő.

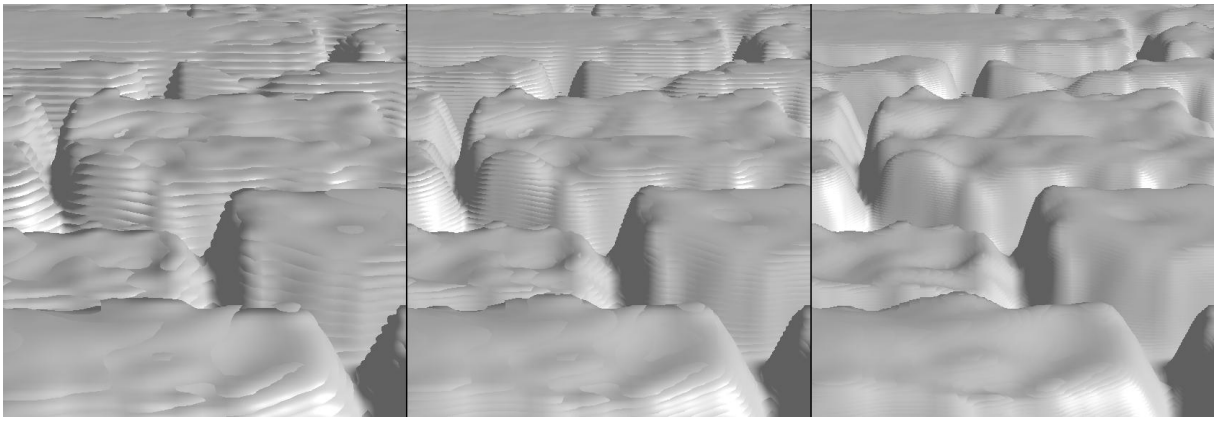
```
VSOUT.UV := INPUT.textúra_koordináta
tangenstér mátrix feléptése
NTB := { INPUT.normál, INPUT.tangens, INPUT.binormál}
a világítási vektor kiszámítása.
L := INPUT.fényforrás_pozíció - INPUT.vertex_pozíció
VSOUT.L := Normalizál( L )
VSOUT.Lts := NTB * L
a nézeti vektor kiszámítása.
V := INPUT.nézőpont_pozíció - INPUT.vertex_pozíció
VSOUT.V := Normalizál( V )
VSOUT.Vts := NTB * V
a további adatok átadása
VSOUT.N := INPUT.normál
VSOUT.T := INPUT.tangens
VSOUT.B := INPUT.binormal
```

A pixel shader

```
a koszinusz alfa, béta, és gamma értéke
cosa := SkalárisSzorzat( -VSOUT.V, VSOUT.N )
cosb := SkalárisSzorzat( VSOUT.V, VSOUT.B )
cosg := SkalárisSzorzat( VSOUT.V, VSOUT.T )
két osztópont közötti magasságkülönbség, ennyivel fogunk előrelépni minden iterációban.
ndh := 1/n
tároljuk az aktuális osztópont magasságát
lépés := 0
```

```
az osztópont távolsága a belépési ponttól d = lépés / cosa * skálázási faktor ,de
az iteráció során nem változó paramétereket kiemeljük
sb := cosb * 1/ cosa * skálázási faktor
st := cosg  * 1/ cosa * skálázási faktor
eltolás = { sb * lépés, st * lépés }
a lineáris keresés
AMÍG lépés < n
        h = TextúraMintavételező( magasságtérkép, VSOUT.UV + eltolás )
        HA lépés < h AKKOR
                lépés = lépés + ndh
                eltolás = { cosa * lépés, cosb * lépés}
        VÉGE
VÉGE
a további lépések megegyeznek a normal mapping-gal csak VSOUT.UV helyett
VSOUT.UV + eltolás kell használni a textúra mintavételezéseknél
```

A paralax mapping-gal ellentétben, ezt a technikát már nem tudjuk shader model 2.0-ban implementálni, mert túlságosan kevés iterációt tudunk végrehajtani. Ezért 3.0-ás árnyalási modellre lesz szükségünk. Ezzel az algoritmussal elég nagy lépésszámra van szükségünk, hogy szép eredményt kapjunk és nagyban függ a pontossága a betekintési szögtől. Minél kisebb a betekintési szög annál hosszabb távolságot kell végignéznünk, és mivel az osztópontok száma állandó, egyre nagyobb lesz a mintavételezési távolság. A osztópontok számának növelése azonban növeli a szükséges számítási teljesítményt és csökkenti a megjelenítés sebességét (12. ábra).

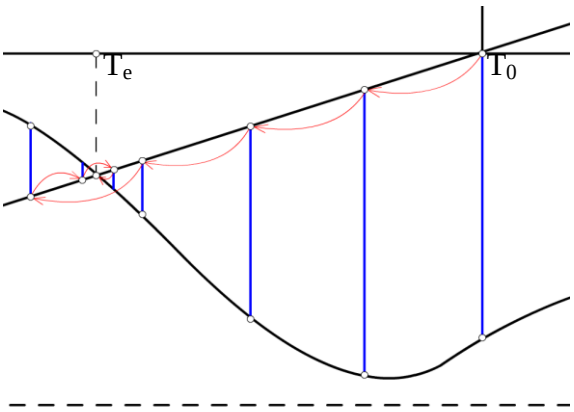


12. ábra: a.) n = 10 b.) n = 20 c.) n = 40

A kép első részén jól látható hogy a kevés lépésszám miatt sok pixelhez ugyanaz a magasságérték tartozik, vagyis a textúra koordinátákat ugyanannyira toljuk el. 40 osztópont esetén már egész szép a kép bár még mindig látható a rétegződés. Növelhetnénk tovább az osztópontok számát, de a megjelenítés sebessége túlságosan le fog csökkenni. Egy olyan megoldásra van szükségünk, amivel úgy kaphatunk pontosabb megoldást, hogy nem kell növelnünk az osztópontok számát.

3.6.2 Relief mapping bináris kereséssel

Használjuk továbbra is a lineáris keresést, és alkalmazzunk bináris keresést utolsó és az azt megelőző osztópont között. Legyen ez a két pont n_v és n_{v-1} és n_m az n_v és n_{v-1} között elhelyezkedő pont. Az n_m azonos távolságra van n_v és n_{v-1}-től. Ha n_m az egyenetlen felület alatt helyezkedik el, akkor legyen $n_v = n_m$, ha pedig felette akkor legyen $n_{v-1} = n_m$. Ezután, ismételjük a lépést a kívánt pontosság eléréséig.

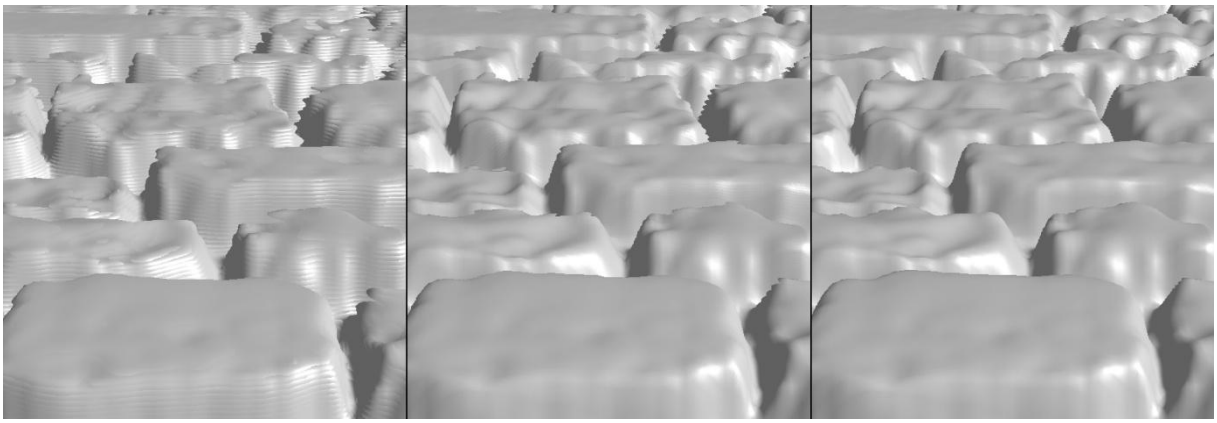


13. ábra: lineáris keresés + bináris keresés

A pixel shader-t a következő pár sorral kell kiegészíteni.

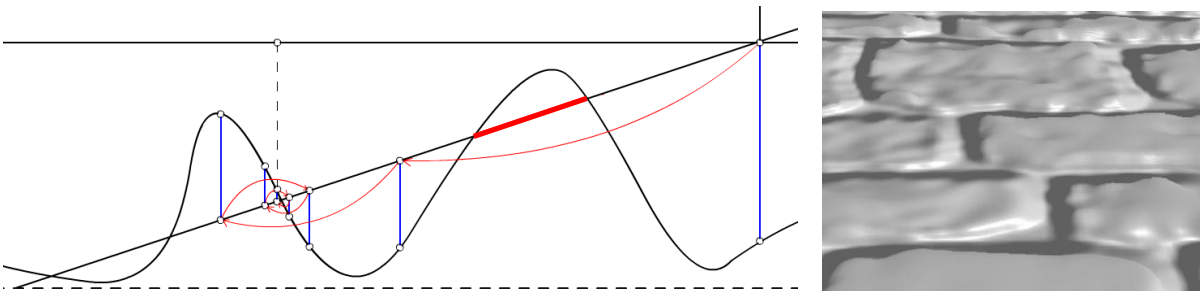
```
lineáris_keresés
…
AMÍG számláló < k
        felére csökkentjük a lépéstávolságot
        ndh := ndh/2
        HA lépés < h AKKOR lépés := lépés + ndh
                ELLENBEN lépés := lépés - ndh
        lépés := lépés + ndh
        eltolás := eltolás = { st * lépés, sb * lépés }
        h = textúra_mintavételező( magasságtérkép, VSOUT.UV + eltolás )
VÉGE
```

…



14. ábra: a.) n = 5, k = 4 b.) n = 10, k = 4 c.) n = 20, k = 4

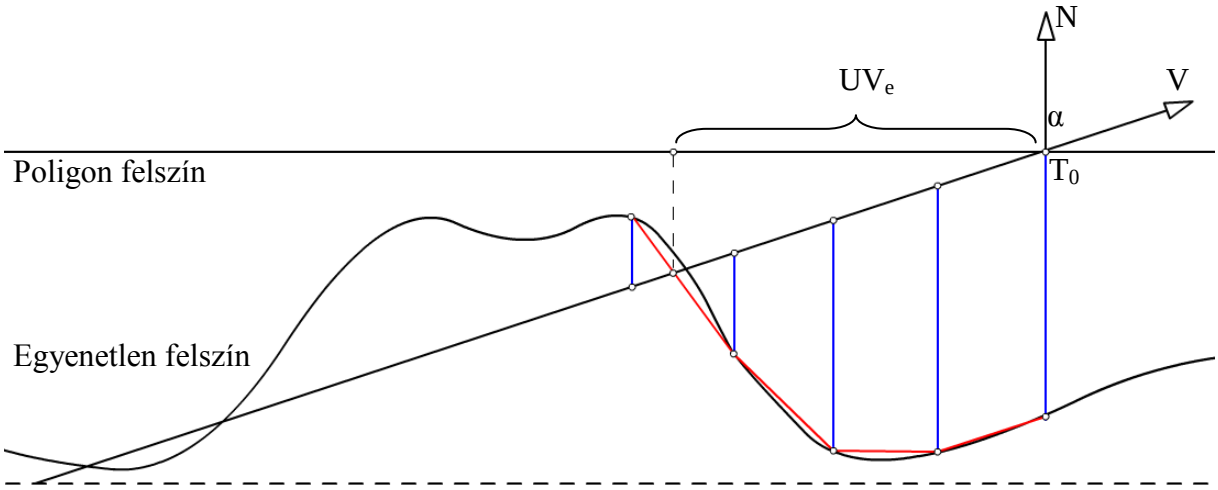
Látható (14. ábra), hogy már 5 osztópont esetén is sokat javult a kép a bináris keresésnek köszönhetően, de még elég sok helyen kaptunk hibás megoldást. 20 osztópont esetén bináris kereséssel, ami összesen 24 iteráció, már sokkal szebb eredményt kaptunk, mint a 40 osztópontos lineáris kereséssel. Felmerül a kérdés, hogy miért nem használunk csak bináris keresést, ha ilyen kis lépésszámmal megtaláljuk a megoldást. Ezt azért nem tehetjük, mert ha nincs behatárolva elég pontosan a metszéspont helyzete, akkor a bináris keresés nagy valószínűséggel át fogja ugrani a helyes megoldást (15. ábra).



15. ábra: bináris keresés a.) A vastag piros vonal jelöli, hogy hol kellett volna megállnia a lineáris keresésnek b.) a hibás végeredmény

A lineáris keresés megoldását pontosíthatjuk egy másik módszerrel is. A lineáris keresés során, a mintavételezés eredményeit egyenes szakaszokkal összekötve, közelíthetjük az eredeti felületet. Ezeket a szakaszokat le tudjuk írni képlettel és meg tudjuk határozni, hogy hol metszi el a sugár őket. Mindössze annyit kell tudnunk, hogy melyik intervallumban vagyunk. Ezt a technikát interval mapping-nak nevezik [6].

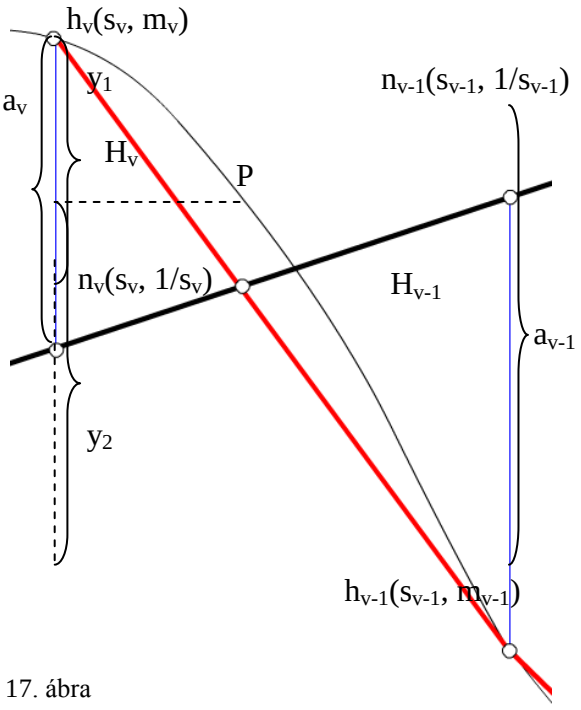
3.6.3 Interval mapping



16. ábra: interval relief mapping

Az intervallum meghatározásához a két hozzá tartozó osztópontra van szükség. A két pont segítéségével fel tudjuk írni ez egyenes egyenletét. Használjuk hát a lineáris keresést a két osztópont megtalálásához. Ez a két osztópont megegyezik a bináris keresés során használt osztópontokkal, azaz n_v és n_{v-1}. Az új megoldásunk a T_e tehát két egyenes metszéspontja lesz, amit pontosan ki tudunk számolni.

Az egyenletek felírásához tekintsük a 17. ábrát. Mivel mindkét egyenes egy síkban van, számolhatunk kettő dimenzióban. Ismerjük az egyenesen elhelyezkedő pontok koordinátáját. Látható, hogy a H_v és H_{v-1} hasonló háromszögek ezért oldalaik arány megegyezik.



17. ábra

$a_v = 1/s_v - m_v, \quad a_{v-1} = m_{v-1} - 1/s_{v-1}$

$r = a_{v-1} / a_v$

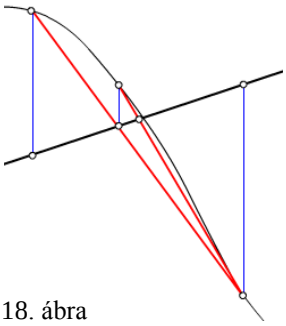
Az y1 és y2 arány is r lesz és tudjuk, hogy $y1+y2=m_{v-1}-m_v$. Felírható az egyenletrendszer.

$$\left.\begin{array}{l} x_2/x_1 = r \\ x_1+x_2=m_{v-1}-m_v \end{array}\right\} \quad x1=(m_{v-1}-m_v)/(1+r) \quad \rightarrow \quad P_x = m_v + x_1$$

$y_2/y_1 = r$

$y_1+y_2=m_{v-1}-m_v$

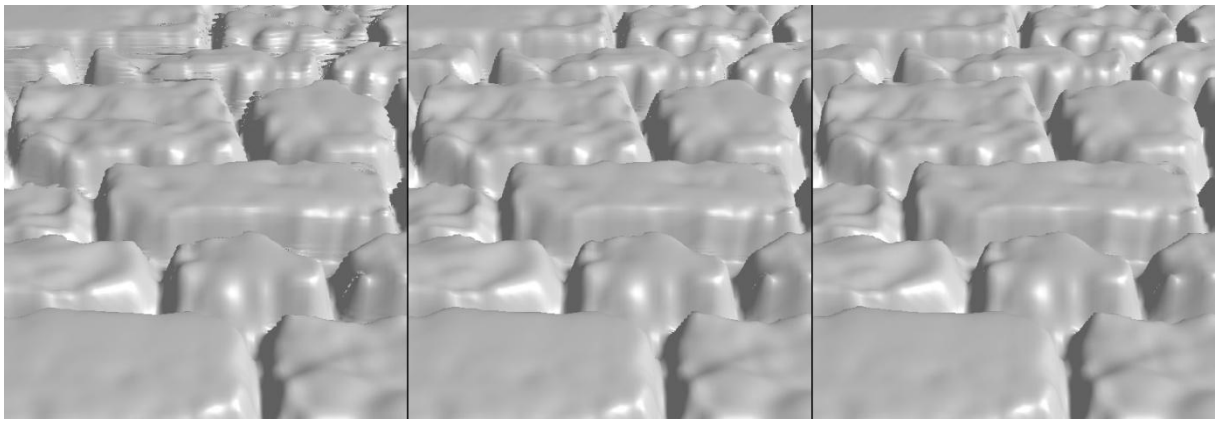
$\Big\}$ $y1=(m_{v-1}-m_v)/(1+r)$ $\rightarrow$ $P_y = m_v + y_1$



18. ábra

Ezzel megkaptuk a P pontot és kiszámolhatjuk hozzá a textúra koordináta eltolását, mivel a hozzá tartozó magasság épen $h_e = P_y$. Természetesen itt is végezhetünk több iterációt (18. ábra). Az módszer implementálásához módosítnunk kell a lineáris keresést.

```
AMÍG lépés < n
    h := TextúraMintavételező( magasságtérkép, VSOUT.UV + eltolás )
    HA lépés < h AKKOR
        lépés_előző := lépés
        h_előző := h
        lépés := lépés + ndh
        eltolás := { cosa * lépés, cosb * lépés}
    VÉGE
VÉGE
intervallum módszer
AMÍG lépés < n
    r := ( h_előző – lépés_előző ) / ( lépés – h )
    lépés := h + (h_előző-h) / (arány+1)
    eltolás := eltolás := { cosa * lépés, cosb * lépés}
    h := TextúraMintavételező( magasságtérkép, IN.UV + eltolás )
VÉGE
```



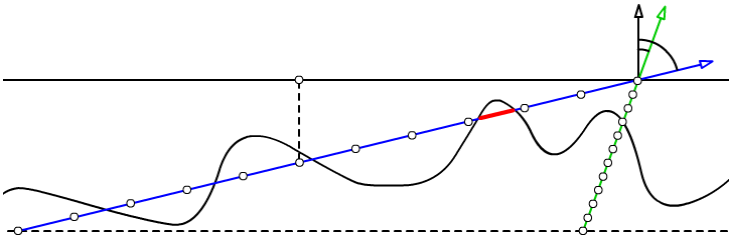
19. ábra: a.) n = 8, k = 2 b.) n = 14, k = 2 c.) n = 20, k = 2

Látható (19. ábra), hogy 2 iterációval már 14 osztópont esetén is szép képet kapunk. Már 8 osztópont is elég lenne, de itt olyan nagyok az ugrások, hogy a lineáris kereséssel kapott megoldás hibás, és ezért rossz intervallumban fogunk számolni.

A bináris keresésnek és az intervallum módszernek is van egy közös gyenge pontja, ami nem más, mint az azt megelőző lineáris keresés. Ha lineáris keresés nem ad elég pontos eredményt, akkor egyik módszer sem tud jó megoldást szolgáltatni. A lineáris keresés pontossága pedig nagyban függ a betekintési szögtől. Vegyük hát figyelembe a betekintési szöget a számítások során.

3.6.4 Nézetfüggő relief mapping

Az egyik lehetőség, hogy a látószög függvényében változtatjuk az osztópontok számát [7]. Látható, hogy nagy szög esetén átugorjuk a helyes megoldást, kis szögnél pedig fölöslegesen sok osztópontot használunk (20. ábra). Kis betekintési szög esetén több, nagy betekintési szög esetén pedig kevesebb osztópontra van szükségünk. Ne feledjük, hogy a lineáris keresés során használt Cos(α)-hoz tartozó α szög nem egyenlő a betekintési szöggel. Az α a felület normálvektora és a nézőpontba mutató vektor által bezárt szög. A betekintési szög lényegében az α ellentettje aza 90°- α. Az osztópontok meghatározásához a következőt kell tennünk.

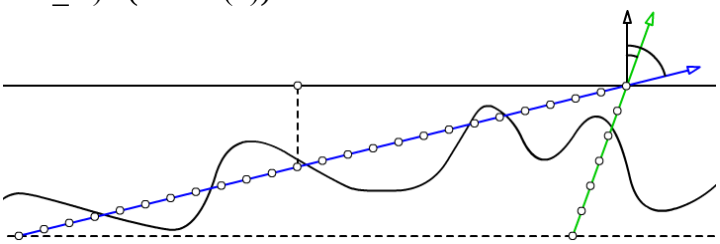


20. ábra: az osztópontok eloszlása kis (kék) és nagy (zöld) betekintési szög esetén

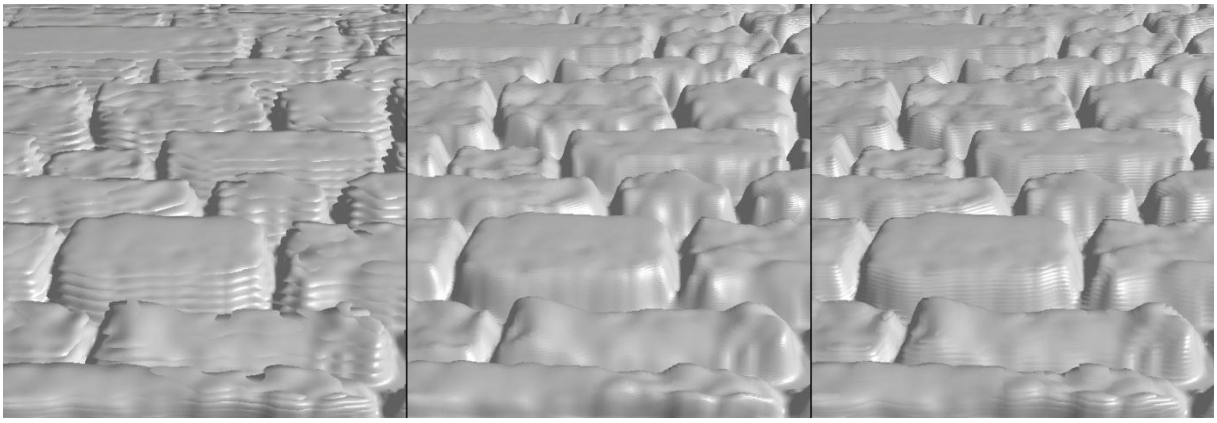
- Definiáljuk a minimális és maximális osztópontok számát. Legyen max_n a maximális és min_n a minimális osztópontok száma.
- Az osztópontok száma a betekintési szög függvényében

$$n = min\_n + (max\_n - min\_n) \cdot (1- Cos(\alpha))$$

Ekkor α=90° esetén épen max_n-t α=0° esetén pedig épen min_n-t kapjuk (21. ábra).



21. ábra: az osztópontok eloszlása kis (kék) és nagy (zöld) betekintési szög esetén



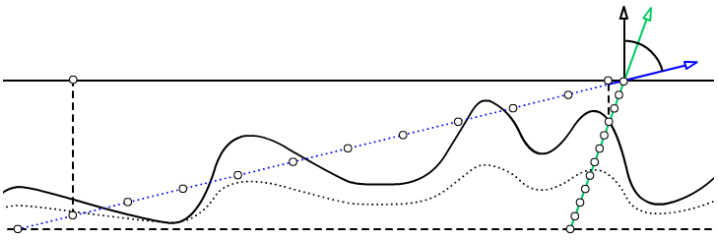
22. ábra: a.) n = 5 b.) n = 25 c.) n = [5,25]

A másik lehetőség a skálázási faktor változtatása a betekintési szög függvényében [7] (23. ábra). Amikor kicsi a betekintési szög csökkentjük a skálázási faktort így nem lesznek nagy kiemelkedések, amit a nagy osztópont távolság miatt átugorhatnánk. Itt ugyanazt a formulát használjuk mint a előző esetben csak most épp ellentétesen.

- Definiáljuk a minimális és maximális skálázási faktort. Legyen max_sf a maximális és min_sf a minimális skálázási faktor.
- Az skálázási faktor a betekintési szög függvényében

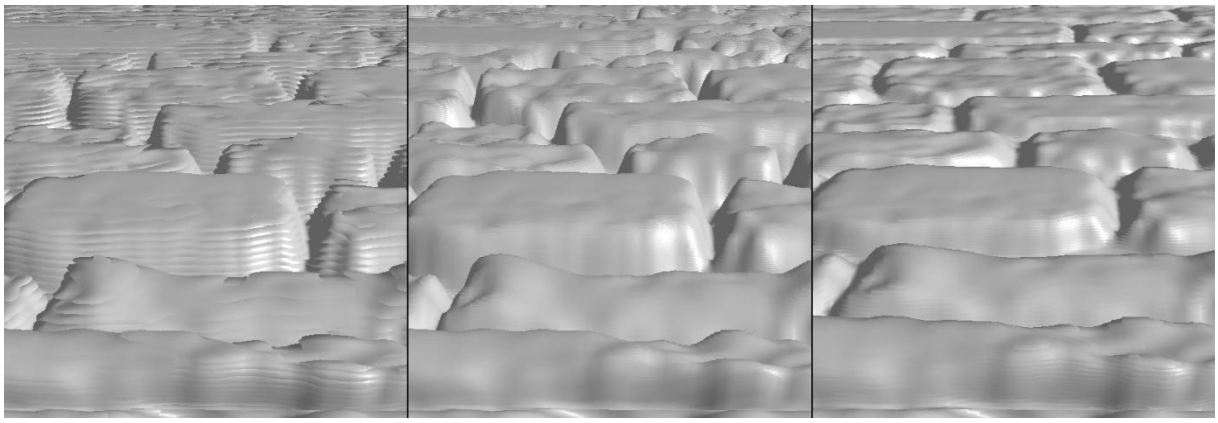
$$\text{skálázási faktor} = \text{min\_fs} + (\text{max\_fs} - \text{min\_fs}) \cdot \text{Cos}(\alpha)$$

Ekkor α=0° esetén épen max_fs-t α=90° esetén pedig épen min_fs-t kapjuk. Az ábra (24. ábra) első képén látható hogy a távolban egyre erősebbek a képhibák, míg változó skálázási faktor esetén a felület ellaposodik, és a képhibák eltűnnek. A koszinusz függvény megfelelő volt arra, hogy 0 és 1 közé transzformálja a betekintési szöget. Viszont szebb képet kaphatunk, ha egy kicsit módosítjuk a görbéjét. A probléma ugyanis az hogy túlságosan hamar elkezd zuhanni a görbe. Számoljuk tehát a skálázási faktort a következő formula segítségével.



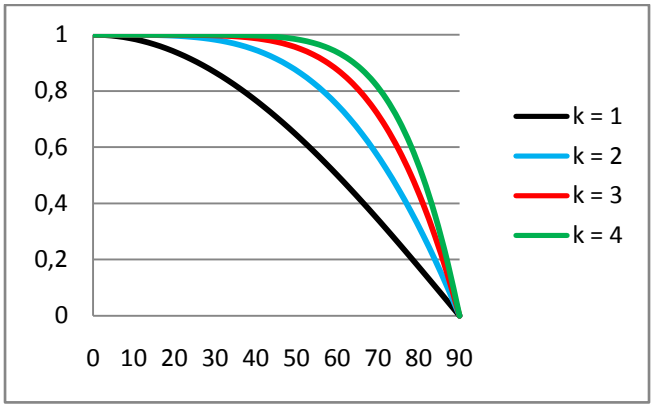
23. ábra: a magasságértékek nagy (kék) és kis (zöld) skálázási faktor esetén

$$\text{skálázási faktor} = 1-(1-\text{Cos}(\alpha))^{k} \qquad \text{(25.ábra)}$$



24. ábra: a.) n = 10 fps = b.) n = 40 fps = c.) n = 15, s.f.= [0,1] fps =

A megfelelő osztópont megválasztásával és valamilyen további pontosító algoritmus segítségével tetszőlegeses pontos eredményt érhetünk el. További optimalizáláshoz használhatjuk a fent leírt két meg megoldás valamelyikét. A relief mapping azonban további lehetőséget is tartogat. Eddig nem foglakoztunk azzal a jelenséggel, hogy az egyenetlensége árnyékor vethetnek a felületre. Erre azért nem volt szükség mert az egyenetlenségek annyira kicsik voltak hogy az általuk vetett árnyékok elhanyagolhatóak volt. Gyakorlatilag csak olyan alacsony szögben beeső fény esetén vettek volna árnyékot ami mer nem is látszódott volna a gyenge fényerősség miatt. A relief mapping azonban már képes olyan magas kiemelkedések korrekt megjelenítésére, aminek az árnyékával már számolnunk kell.

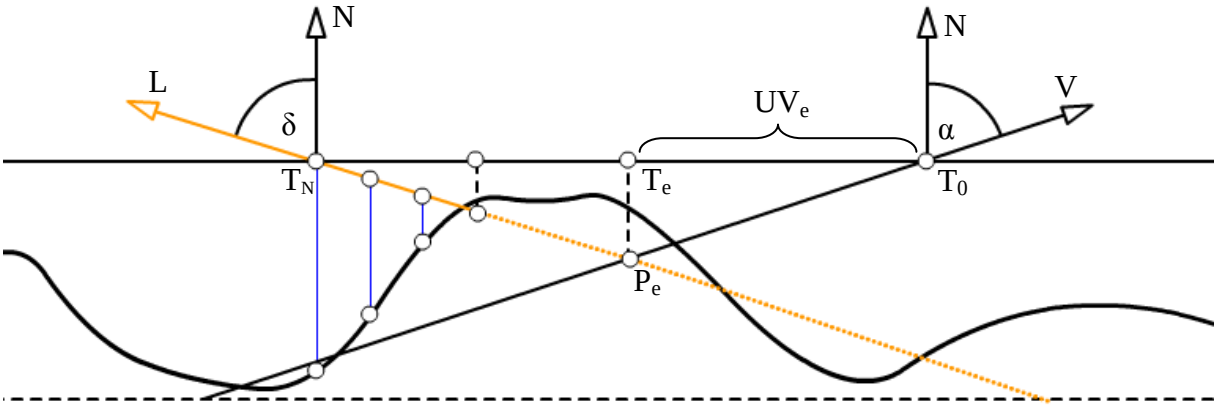


25. ábra: a magasságértékek nagy (kék) és kis (zöld) skálázási faktor esetén

3.6.5 Relief mapping önmagára vetett árnyékkal

A relief mapping képes megfelelően megjeleníteni a felület egyenetlenségeit a sugárkövető algoritmusnak köszönhetően. Ezt az algoritmust azonban nem csak arra használhatjuk, hogy a megvizsgáljuk a szem mely texeleket láthatja. Ha a szem helyet a fényforrásból indítjuk a sugárkövetést, akkor megvizsgálhatjuk, hogy a fényforrás számára mely texelek láthatóak [5,7]. Ha egy pont nem látható a fényforrásból, akkor az azt jelenti, hogy az adott pontot nem

éri el a fény, vagyis árnyékban van. Ezt felhasználva megjeleníthetjük az egyenetlen felület önmagára vetett árnyékát.



26. ábra: L a fényforrásba mutató vektor, T_N a fénysugár belépési pontjához tartozó textúra koordináta

A sugárkövetés elindításához szükségünk van a sugár irányára és a belépési pontra. A sugár irányát meghatározhatjuk a két pont segítségével, amin a sugár áthalad. Ez a két pont pedig nem más, mint a szemből indított sugárkövetés után kapott metszéspont és a fényforrás pozíciója. A fényforrás pozícióját pontosan ismerjük, de a másik pont térbeli koordinátáit még nem ismerjük mivel eddig 2 dimenzióban számoltunk és csak a hozzá tartozó magasságérték érdekelt minket. Viszont könnyen kiszámolhatjuk. Ismerjük szemből indított sugár irányvektorát és vizsgált pont távolságát a belépési ponttól.

$P_e(x,y,z) = P_0 + -\vec{V} \cdot d$, ahol P_0 nézőpontból indított sugár belépési pontja

A fényforrásból indított sugár irányvektora tehát

$\vec{L} = P_e - P_L$, ahol P_L a fényforrás pozíciója

Már csak a fénysugár belépési pontjára van szükségünk, pontosabban a hozzá tartozó textúra koordinátákra. Ezt könnyen meghatározhatjuk, mivel tudjuk, hogy $T_e = T_0+UV_e$ és T_e-t ismerjük.

$T_e = T_N + UV_N \quad \rightarrow \quad T_N = T_e-UV_N$

Már csak meg kell határozni az UV_N értékét. Az UV_N meghatározásához szükségünk van P_e pont távolságára a belépési ponttól.

$d = h_e / \mathrm{Cos}(\delta)$, ahol $\mathrm{Cos}(\delta) = \vec{N} \cdot \vec{L}$

A belépési pontot távolsága ismeretében kiszámíthatjuk UV_N-t az eddigiek szerint.

$$\text{Cos}(\beta) = \vec{N} \cdot \vec{T} \quad \rightarrow \quad \vec{N} \cdot \vec{T} = \frac{U_n}{d} \quad \rightarrow \quad (1)\ U_n = d \times (\vec{N} \cdot \vec{T})$$

$$\text{Cod}(\gamma) = \vec{N} \cdot \vec{B} \quad \rightarrow \quad \vec{N} \cdot \vec{B} = \frac{V_n}{d} \quad \rightarrow \quad (2)\ V_n = d \times (\vec{N} \cdot \vec{B})$$

Most már megvan minden szükséges információnk a sugárkövetés megkezdéséhez. Ha sugárkövetés után kapott megoldás megegyezik az előző sugárkövetés megoldásával, akkor az adott pontot eléri a fény, ha nem egyezik meg, akkor árnyékban van. Természetesen számítási pontatlanságok miatt szükséges egy határérték (S) definiálására, amin belül a két pontot azonosnak tekintjük. Most is különböző számú osztóponttal dolgozhatunk, de érdemes nézeti sugárkövetéshez használt osztópontszámmal dolgozni. Ez azért előnyös, mert így biztos, hogy azonos magasságszinten találjuk meg a metszéspontokat, ha azok egyeznek. Természetesen itt már egy kicsit módosul az árnyalás számítása az előző technikákhoz képest. Nem csak arra kell figyelni, hogy az árnyékolt pixelek sötétebbek legyenek, hanem arra is hogy a visszaverődő fény sem érkezhet egy olyan pontról amit el sem ér a fény.

Módosítanunk kell a vertex shader-t mert a P_e térbeli koordinátájához szükségünk van a belépési pont térbeli koordinátájára pixelenként. A VSOUT tehát a következő adatokat tartalmazza.

- textúra koordináta (UV)
- vertex pozíció (PV)
- fény pozíció (PL)
- nézőpontba mutató vektor (V)
- nézőpontba mutató vektor a tangens térben (Vts)
- fényforrásba mutató vektor a tangens térben (Lts)
- normálvektor (N)
- tangensvektor (T)
- binormálvektor (B)

A vertex shader-t most nem részletezzük, mivel semmilyen újfajta kalkulációt nem tartalmaz. A pixel shader-t viszont ki kell egészíteni és meg kell változtatni árnyalást is.

```
...
a nézeti sugárkövetés metszéspontjának meghatározásáig a shader változatlan
eltároljuk az aktuális osztóponthoz magasságát értékét
```

```
lépésV := lépés
az eltolt textúra koordináta
uv_eltolt := VSOUT.UV + eltolás
a metszés pont térbeli koordinátáinak meghatározás
pozíció := VSOUT.PV + VSOUT.V * 1/ cosa * skálázási_faktor * lépés
a fényforrásba mutató vektor meghatározása
L := PL – pozíció
a koszinusz alfa, béta, és gamma értéke
cosa := SkalárisSzorzat( -L, VSOUT.N )
cosb := SkalárisSzorzat( L, VSOUT.B )
cosg := SkalárisSzorzat( L, VSOUT.T )
két osztópont közötti magasságkülönbség, ennyivel fogunk előrelépni minden iterációban.
ndh := 1/n
az osztópont távolsága a belépési ponttól d = lépés / cosa * skálázási faktor ,de
az iteráció során nem változó paramétereket kiemeljük
sb := cosb * 1/ cosa * skálázási faktor
st := cosg  * 1/ cosa * skálázási faktor
eltolás = { sb * lépés, st * lépés }
a belépési textúra koordináta meghatározása
uv:= uv_eltolt - eltolás
a kezdeti magasságot 0-ra állítjuk mivel a belépési ponttól indulunk
lépés := 0
eltolás = { sb * lépés, st * lépés }
a lineáris keresés
AMÍG lépés < n
        h = TextúraMintavételező( magasságtérkép, VSOUT.UV + eltolás )
        HA lépés < h AKKOR
                lépés = lépés + ndh
                eltolás = { cosa * lépés, cosb * lépés}
        VÉGE
VÉGE
```

fénysugár metszéspontjához tartozó lépés

lépésL := lépés

ellenőrízzük hogy a fény eléri-e a metszéspontot

HA abs(lepesV - lepesN) > S AKKOR árnyék := 0

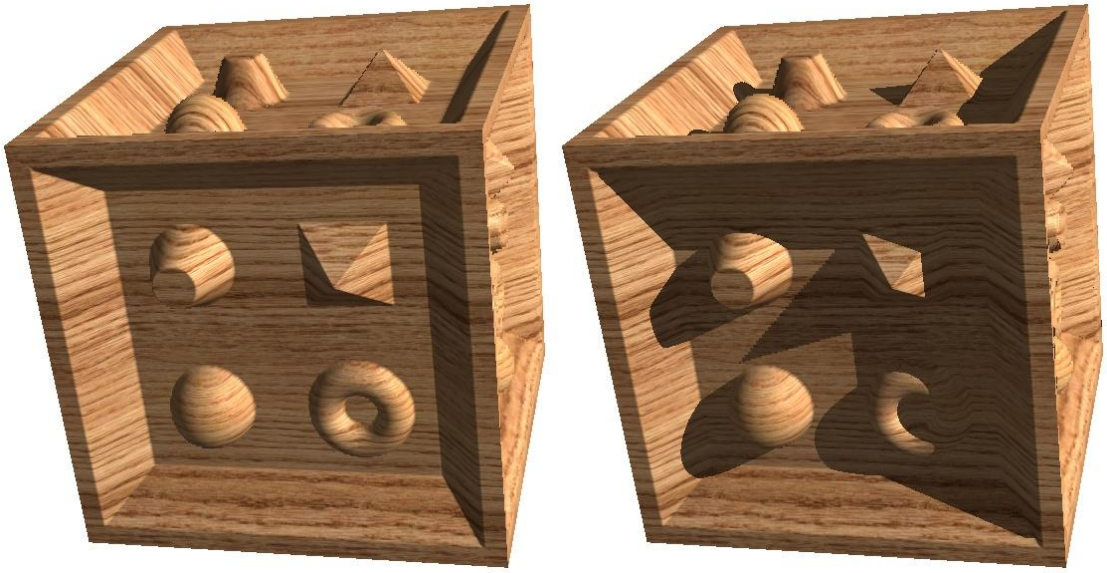
ELLENBEN árnyék := 1

…

a módosított árnyalás

PSOUT.szín := diffúz_szín * (fényerő * árnyék + környezeti_fényerő) * fény_szín

PSOUT.szín := PSOUT.szín + spekuláris_fényerő * árnyék

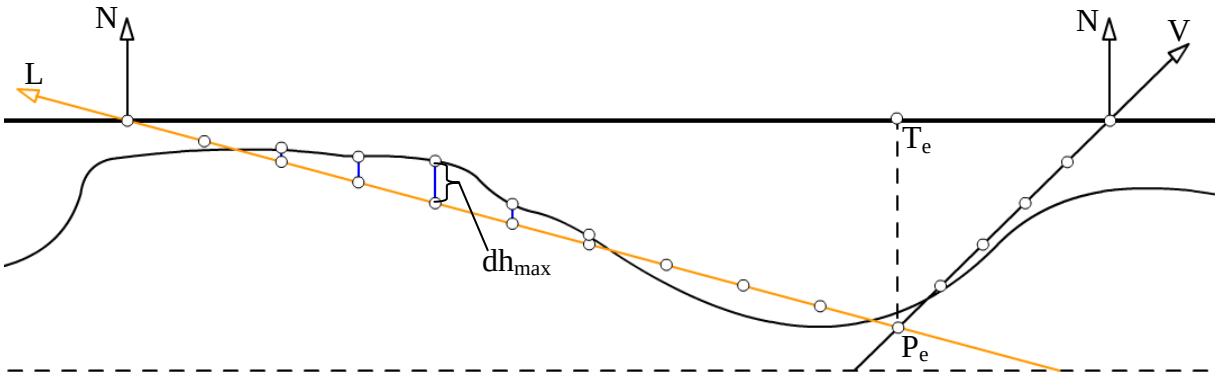


27. ábra: a.) relief mapping árnyék nélkül b.) relief mapping árnyékkal

Látható (27. ábra), hogy az árnyékok szélei nagyon élesek. A valóságban ritkán élesek ennyire az árnyékok. Ez azért van, mert a fényforrásnak mindig van kiterjedése, és míg a fényforrás egyik része nem világítja meg a felületet a másik része lehet, hogy igen, és ilyenkor valamennyi fény eléri a felületet. Minél nagyobb része van kitakarva a fényforrásnak annál gyengébb lesz a fényerő.

3.6.6 Relief mapping puha vetett árnyékkal

A kiterjedéssel rendelkező fényforrások szimulálása meglehetősen sok számítást igényel, ezért egy egyszerűbb modellt fogunk használni. A fényforrás kiterjedését nem vesszük figyelembe, helyette egy csak azzal foglalkozunk, hogy mennyire van kitakarva [7,8]. A módszer [7]-ban megoldáson alapul, de mi már az árnyék kiszámítása során meghatározzuk a fényforrás blokkolásának mértékét, így kevesebb számításra van szükség.



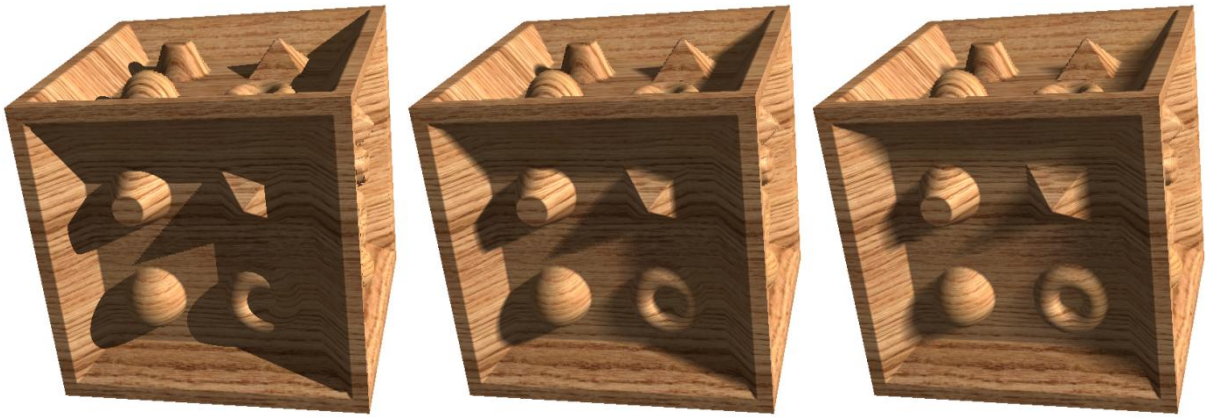
28. ábra: L a fényforrásba mutató vektor, dh_{max} a maximális eltérés az osztópontok magassága és a hozzájuk tartozó magasságértékek között, a lineáris keresés során

Nézzük meg, hogy a fényforrásból indított sugár maximálisan milyen mélyen hatol be az egyenetlen felszínbe. Mivel a sugárkövetési algoritmus során minden ponthoz kiszámoljuk a magasságtérkép értékét, nincs más dolgunk csak megjegyezni a maximális távolságot az adott osztópont magassága és a hozzá tartozó magasságérték között. Ezután definiáljunk egy határértéket, aminél ha kisebb a távolság, akkor

- A határérték legyen M
- Skálázzuk 1 és 0 közé a dh_{max} értékét árnyék = (mdh_{max} / M). Így az átmenet lineáris lesz 0 és 1 között, de az betekintési szögtől függő skálázási faktornál leírtak alapján változtathatunk az átmeneten.

Minél nagyobbnak választjuk meg a M értékét, annál jobban elmosódik az árnyék széle. A dh_{max} meghatározásához módosítani kell a lineáris keresés algoritmusát az árnyék számításánál. Fontos hogy a dh_{max}-ot addig kell keresni, amíg el nem érjük P_e-t. Tehát itt nem az első metszéspontnál álunk meg, hanem amíg el nem érjük P_e-t

```
…
a lineáris keresés,legyen a metszésponthoz tartozó magasság kezdetben -1, ezzel jelezve, hogy ismeretlen
lépésL := -1
AMÍG lépés < n
        h = TextúraMintavételező( magasságtérkép, VSOUT.UV + eltolás )
        ha az egyenetlen felszín alatt vagyunk
        HA lépés < h  ÉS lépésL < LépésV AKKOR
                és még nincs metszéspontunk
                HA lépésL = -1 AKKOR lépésL := lépés
                        ha viszont már van
                        ELLENBEN dhmax = Max( dhmax, lépés - h )
        VÉGE
        lépés = lépés + ndh
        eltolás = { cosa * lépés, cosb * lépés}
VÉGE
ellenőrizzük, hogy a fény eléri-e a metszéspontot
HA Abs( lépésL - lépésN ) > S AKKOR
        ellenőrizzük, hogy érheti-e a felszínt valamennyi fény
        HA dhmax < M AKKOR árnyék := 1 – dhmax / M
                ELLENBEN árnyék := 0
Az árnyalás változatlan marad
…
```



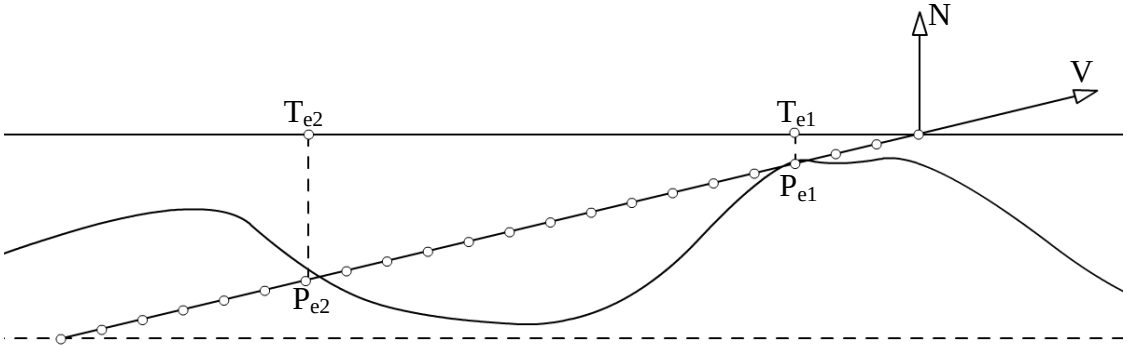
29. ábra: a.) M =0 b.) M = 0.2 c.) M = 0.15

A textúra koordináták tologatásával egy olyan esztétikai hibát hozhatunk létre, ami lerontja a képminőséget. A kiemelkedések széleinél láthatjuk (30. ábra), hogy milyen éles az átmenet a pixelek között. Ha egy textúrát transzformálunk (forgatás, nyújtás), akkor bizonyos pixelekhez több texel is tartozhat és el kell dönteni melyik texelt rajzoljuk ki. Ebben az esetben bizonyos texeleket el fogunk hagyni és képen éles átmenetek lesznek a pixelek között. Ennek elkerülése érdekében a textúrákon szűréseket végez a hardver. Ez azt jelenti, hogy a szomszédos vagy közeli texelek értékeit összemossa és így a pixelek között mindig folyamatos lesz az átmenet. Amikor lekérdezünk egy texelt akkor mi már ezt az átlagolt értéket kapjuk meg. A relief mapping során viszont nem azok köré a texelek köré rajzoljuk ki a texelt ahol az eredetileg volt, hanem eltoljuk, és ezért látunk a képen éles átmenetet. Ezeket az éleket elmoshatjuk, azaz a szomszédos pixelek színeit kombinálva csökkenthetjük az átmenet erősségét. Ezt nevezik élsimításnak.

30. ábra: az éleknél keletkező pixelesedés kiemelve

3.6.7 Relief mapping élsimítással

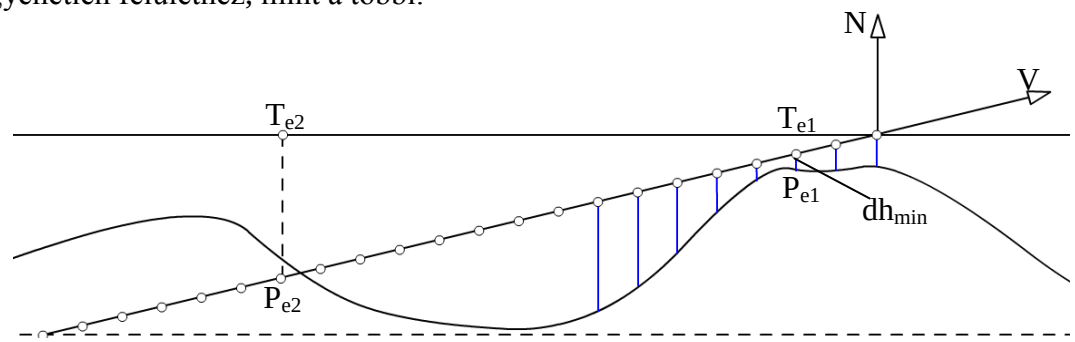
Ahogy már említve volt, a pixel shader nem ad lehetőséget a szomszédos pixelek színeinek eléréséhez. Az élsimítás során tehát azt kell meghatározni, hogy a szomszédos texel milyen színű. Az egyik megoldás (31. ábra), hogy a metszéspont megtalálása után megnézzük, hogy a rákövetkező metszéspont a magasságtérkép felett vagy alatt helyezkedik el. Ha ismét a magasságtérkép felett vagyunk, akkor az azt jelenti, hogy egy élnél találtunk metszéspontot. Ez nagyban függ az osztópontok számától. Ha nagy az osztópontok távolsága, akkor olyan részeket is élnek fogunk venni, ami valójában nem az. Ha azt kaptuk, hogy egy felület élét



31. ábra: az első osztópont pont egy élnél van ezért folytatjuk a keresést a következő osztópontig. A T_{e1} és T_{e2} koordinátákhoz tartozó texelek értékeit átlagoljuk

találtuk el akkor a sugárkövetést folytatnunk kell a következő metszéspont megtalálásához, ugyanis az ehhez a ponthoz tartozó texelt fogjuk kiátlagolni az eredeti metszésponthoz tartozóval. Természetesen nem vehetjük egyszerűen a texel értékét, mert a fényviszonyok, jelentősen befolyásolják az értékeket. Mind a két pontban ki kell számolni az árnyalt pixel színét. Ezzel a megoldással csak az a probléma, hogy egy vékony kiemelkedés esetén is élként kezeli a metszéspontot függetlenül attól, hogy a közepénél találtuk el, mivel a következő lépésben már kilép az egyenetlen felület felé

A másik és egyben jobb lehetőség, hogy nem a metszéspontot vizsgáljuk, hanem a metszéspont keresése során azt figyeljük, hogy az egyes osztópontokban mennyire kerülünk közel a felülethez. Definiáljunk egy határértéket (E), aminél ha közelebb kerülünk, akkor megjegyezzük az osztópont távolságát a magasságértéktől. A felülethez több helyen is közel kerülhetünk ezért azt a pontot jegyezzük meg, ami a legközelebb volt a felülethez az összes közül. Mivel a határérték fix, a távolság 0 és a határérték között lesz. A távolságot 0 és 1 közé skálázzuk, ahogy ezt már többször megtettük a relief mapping során más értékekkel. Ezt a skálázott értéket használjuk fel arra, hogy milyen arányban kell a két texelt figyelembe venni. Minél nagyobb a távolság annál kevésbé fog dominálni az élhez tartozó texel. Természetesen a metszéspont keresés során mindig lesz minimális távolságunk és az is valószínű, hogy ez kisebb lesz, mint határérték mivel a metszéspontot megelőző pont valószínűleg közelebb van az egyenetlen felülethez, mint a többi.



32. ábra: a P_{e1} pont élnél van mivel ő van a legközelebb és a rákövetkező pont már távolabb van

Épen ezért csak akkor mondjuk, hogy él mellett haladtunk el, ha az adott osztópont után következő osztópont ismét távolabb van a magasságtérképtől (32. ábra). Mivel lineáris keresés addig tart, amíg az egyenetlen felszín alá nem érünk, az addig vizsgál pontok, mind a felület felett fognak elhelyezkedi.

- Legyen dh_{min} a legkisebb távolság felülettől a keresés során
- Ha ebben a pontban élt találtunk és $dh_{min} < E$ akkor skálázzuk ezt a távolságot 0 és 1 közé és legyen a texel_arány = dh_{min} / E
- A metszésponthoz tartozó pixel színe tehát texel1+(texel2-texel1)×texel_arány

Az élek detektálására és megszüntetésére még számos megoldás létezik, de ezek úgynevezett utómunkák, amiket már akkor végzünk mikor megkaptuk a kész képet. Ez nem tartozik ehhez a technikához. A 32. ábrán látható az éldetektálás eredménye.

33. ábra: az éldetektálás eredménye, a fehér részek az élek.

A technika implementálásához csak a pixel shader-t kell módosítanunk. Az alap relief mapping pixel shader-ét fogjuk módosítani. A lineáris keresés a puha vetett árnyék technikához hasonlóan változik, csak most minimális távolságot keresünk.

```
…
tároljuk az élhez tartozó magasságot, kezdetben -1, ezzel jelezve, hogy még ismeretlen
lépésE := -1
lineáris keresés
AMÍG lépés < 1
    h = TextúraMintavételező( magasságtérkép, VSOUT.UV + eltolás )
    dh := lépés - h
    ha az egyenetlen felszín alatt vagyunk
    HA dh < 0  AKKOR
        ha ez a leközelebbi távolság
        HA dhmin > dh AKKOR
            megjegyezzük a távolságot
            dhmin := dh
            meghatározzuk a következő ponthoz tartozó távolságot
            eltolás : = { cosa * ( lépés + ndh ), cosb * ( lépés + ndh ) }
```

```
                hköv := TextúraMintavételező( magasságtérkép, VSOUT.UV + eltolás )
                dhköv := hköv – dh
                ha a következő osztópont távolsága nagyobb élt találtunk
                HA dhköv > dhmin ÉS AKKOR lépésE := lépés
        VÉGE
        lépés = lépés + ndh
        eltolás = { cosa * lépés, cosb * lépés}
VÉGE
a metszésponthoz tartozó texel színe
texel_szín2 := ÁrnyaltTexel( VSOUT.UV + eltolás )
ha nem találtunk élt
HA lepesE = 1 AKKOR PSOUT.szín := texel_szín2
        ELLENBEN
                ha az él elég közel van, akkor a texel arány meghatározása
                HA dhmin < E AKKOR
                        arány := dhmin / E
                        eltolás := { cosa * lépésE, cosb * lépésE }
                        texel_szín1 := ÁrnyaltTexel( VSOUT.UV + eltolás );
                        az átmente függvénnyel kiszámoljuk a pixel színét
                        PSOUT.szín := Átmenet( texel_szín1, texel_szín2, arány)
                VÉGE
        VÉGE
```

34. ábra: a.) relief mapping élsimítás nélkül fps = b.) relief mapping élsimítással fps =

Ezzel már minden adott a szép kép eléréséhez, már csak ki kell választani a megfelelő technikákat és optimális paramétereket.

Előnyök

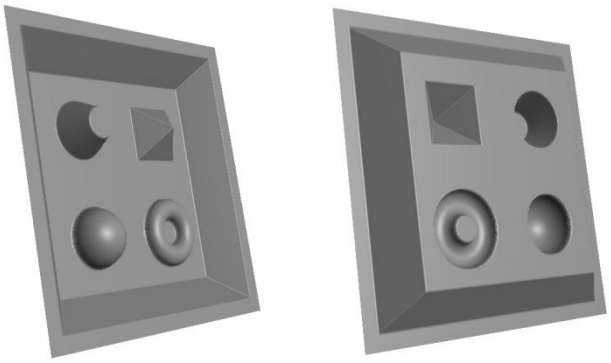
- A felület egyenetlenségei valóban kiemelkednek és eltakarják a nem látható részeket
- Megjeleníthető a spekuláris fény, és az önmagára vetett árnyék
- Kevés poligonszám mellett is létrehozhatunk tetszőleges részletességű felületeket

Hátrány

- Az eddigi technikákhoz képest, nagyságrendekkel nagyobb számítási teljesítményre van szükség
- Minden modellhez el kell készíteni a normáltérképet, amihez általában elkészítik a teljes részletességű nagy poligonszámból álló modellt. Ehhez többek között displacement mapping technikát használnak.

3.6.8 Kétoldalas relief mapping

A számítógépes grafika során azokat a poligonokat, amik háttal vannak nem szoktuk kitölteni, mert általában nem látszódnak. Viszont bizonyos esetekben szükséges a kirajzolásuk. Erre vagy akkor van szükség, amikor átlátszó felületeket ábrázolunk, és ezért a hátoldalas poligon látszódnak rajtuk keresztül vagy akkor, amikor olyan objektumot reprezentálunk, ami olyan vékony anyagból áll, hogy nem érdemes mind a két oldalát lemodellezni, hanem egyszerűen ugyanazt a poligont használjuk fel, mindkét oldalhoz. Ilyen például a papírlap, vagy egy lemezből készült cső. A hátsó lapok kirajzolása egy egyszerű textúra leképzés esetén nem okoz problémát, mivel a hardver megoldja. A relief mapping esetén azonban több probléma is felmerül. Az egyik hogy a felület normálvektorai pont az ellenkező irányba mutatnak. A másik, hogy ami az egyik oldalt kiemelkedés az a másik oldalt benyomódás lesz (35. ábra), vagyis a magasságtérkép inverzét kell venni. Ezeket a módosításokat azonban könnyen elvégezhetjük.



35. ábra: kétoldalas relief mapping. Elülső oldalaz (balra), hátsó oldal (jobbra)

- Meg kell állapítani, hogy a poligon melyik oldalát látjuk. Ha az $\alpha \geq 0$ akkor a poligon elülső oldalát látjuk, ha $\alpha < 0$ akkor pedig a hátsót.
- Ezután az új α értéke $|\alpha|$ lesz
- Ha az elülső oldalt látjuk, minden marad változatlan
- Ha a hátsó oldalt látjuk
    - a magasságértéket kivonjuk 1-ből
    - a felület normálvektorát -1-el megszorozzuk

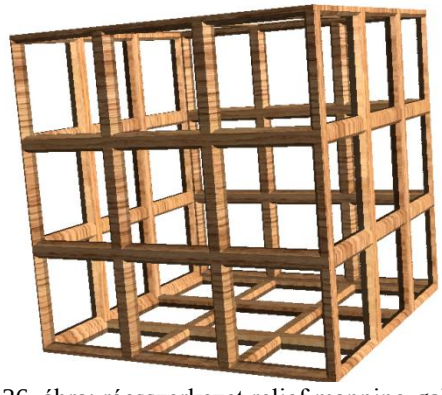
A pixel shader-ben tehát a következő sorokat kell módosítani.

```
a koszinusz alfa, béta, és gamma értéke
cosa := SkalárisSzorzat( -VSOUT.V, VSOUT.N )
előjel := 1
mélységi_eltolás := 0
HA cosa < 0 AKKOR
        cosa := abs(cosa)
        előjel := -1
        mélységi_eltolás := -1;
eltolás = { sb * lépés + mélységi_eltolás, st * lépés + mélységi_eltolás }
…
        HA előjel < 0 AKKOR h := 1 - h
az adott ponthoz tartozó normálvektor tehát
Nts := előjel * TextúraMintavételező( normáltérkép, VSOUT.UV ) * 2 -1
```

Az objektum kirajzolásához először a hátsó lapokat majd ezután a felénk nézőket rajzoljuk ki.

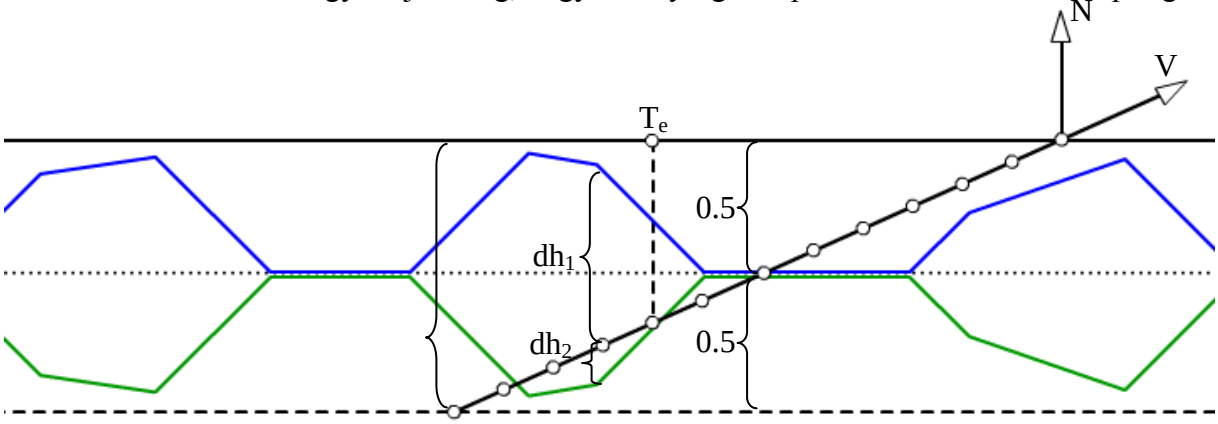
3.6.9 Rácsfelületek megjelenítése kétoldalas relief mapping-gal

Kis módosítással a kétoldalas relief mapping-ot használhatjuk rácsfelületek megjelenítésre. Ezek olyan felületek, amin lyukak vannak és, kiterjedésükhöz képest laposak. Ilyen például egy drótkerítés. Két dolgot kell módosítanunk. Az egyik, hogy a fülét azon részeit ahol lyukak vannak, átlátszóvá kell tenni. Ezt vagy úgy oldjuk



36. ábra: rácsszerkezet relief mapping-gal

meg, hogy egy átlátszósági maszkot definiálunk, amit valamelyik textúra szabad csatornájában eltárolunk, vagy pedig úgy hogy a 0 magasságú pontokat átlátszónak tekintjük. A másik változás, hogy a felület két oldala szimmetrikus, tehát a hátsó oldalnál nem az inverzét kell venni a magasságtérképnek. A lyukak miatt viszont átlátunk a felületen és ezért láthatunk bizonyos részletek a túlsó oldalból is. Ezért mind a két oldalt figyelembe kell venni a számítások során. Ezt úgy oldjuk meg, hogy a mélységtérkép inverzét is betesszük a poligon



37. ábra: a szimmetrikus magasságtérkép. az eredeti magasságtérkép kék, a inverz magasságtérkép zöld.

alá, ez eredeti mélységtérkép alá (37. ábra).

A lineáris keresés akkor áll meg, amikor a két magasságtérkép közé érünk. Ez úgy tudjuk leellenőrizni, hogy az kiszámoljuk az osztópont magassága és a magasságtérkép értéke közötti különbséget mind a két magasságtérképhez. Ha a kettő között vagyunk, akkor a különbségek ellentétes előjelűek lesznek, ellenkező esetben pedig azonosak. Ha tehát $dh_1 \times dh_2 \leq 0$ és $dh_1 \neq dh_2$ akkor megállunk. Azért nem engedjük meg az egyenlőséget, mert akkor mindig megállna a két magasságtérkép találkozásánál. Ebben az esetben dh1 ≤ 0 és dh2 ≥ 0. Ha –dh1 > dh2 akkor a második magasságtérképnél találtuk meg a megoldást, ezért a normálvektor át kell számítani. Mivel a felület szimmetrikus ezért csak a z koordinátát kell megszorozni -1-el. A reálisabb megjelenítés érdekében a mélységi eltolást -0.5-re állítjuk, így az eredeti magasságtérkép a poligon felé, míg az inverze a poligon alatt lesz. Módosítsuk tehát a kétoldalas relief mapping pixel shader-ét.

```
    a koszinusz alfa, béta, és gamma értéke
    cosa := SkalárisSzorzat( -VSOUT.V, VSOUT.N )
    előjel := 1
    mélységi_eltolás := -0.5
    HA cosa < 0 AKKOR előjel := -1
```

```
cosa := abs(cosa)
eltolás := { sb * lépés + mélységi_eltolás, st * lépés + mélységi_eltolás }
a lineáris keresés
AMÍG lépés < 1
        h1 := 0.5 * TextúraMintavételező( magasságtérkép, VSOUT.UV + eltolás )
        h2 := 1 – h1;
        dh1 := step – h1
        dh2 := step – h2
        HA dh1 * dh2 > 0 VAGY dh1 = dh2 AKKOR
                lépés = lépés + ndh
                eltolás = { sb * lépés + mélységi_eltolás, st * lépés + mélységi_eltolás }
        VÉGE
VÉGE
…
az adott ponthot tartozó normálvektor tehát
Nts := előjel * TextúraMintavételező( normáltérkép, VSOUT.UV ) * 2 -1
HA dh2 < –dh1 AKKOR Nts.z := Nts.z * -1
…
```

3.6.10 Kétrétegű relief mapping

Az előző technikához képest annyiban változik ez e módszer, hogy nem ugyanazt a magasságtérképet és normáltérképet használjuk a számítások során, hanem két külön térképet. Ezzel lehetőség van nem csak szimmetrikus felületek megjelenítésére is. A pixel shader csak annyiban változik, hogy figyelni kell mikor melyik magasság- illetve normáltérképről vesszük az adatokat.

38. ábra: nem szimmetrikus alakzat relief mapping-gal

```
…
a lineáris keresés
AMÍG lépés < 1
        h1 := 0.5 * TextúraMintavételező( magasságtérkép1, VSOUT.UV + eltolás )
        h2 := 0.5 * TextúraMintavételező(magasságtérkép 2, VSOUT.UV + eltolás )
```

```
            HA előjel < 0 AKKOR
                ha a hátsó lapot látjuk, akkor a hátsó magasságtérkép kerül felülre
                ht := h1
                h1 := 1 - h2
                h2 := h1
                ellenkező esetben az elülső
                VÉGE ELLENBEN   h2 := 1 – h2
…
HA dh2 > –dh1 AKKOR
        attól függően, hogy a poligon melyik oldalát látjuk változik, hogy melyik
        normáltérkép van felül ezért mindig a megfelelő normáltérképről kell venni az
        értékeket
        HA előjel < 0 AKKOR
            Nts := TextúraMintavételező( normáltérkép1, VSOUT.UV ).xyz * 2 -1
            ELLENBEN
            Nts := TextúraMintavételező( normáltérkép2, VSOUT.UV ).xyz * 2 -1
        Nts.z := előjel * Nts.z * -1
         VÉGE
         ELLENBEN
            HA előjel < 0 AKKOR
            Nts := TextúraMintavételező( normáltérkép2, VSOUT.UV ).xyz * 2 -1
            ELLENBEN
            Nts := TextúraMintavételező( normáltérkép1, VSOUT.UV ).xyz * 2 -1
            Nts.z := előjel  * Nts.z * -1
        VÉGE
…
```

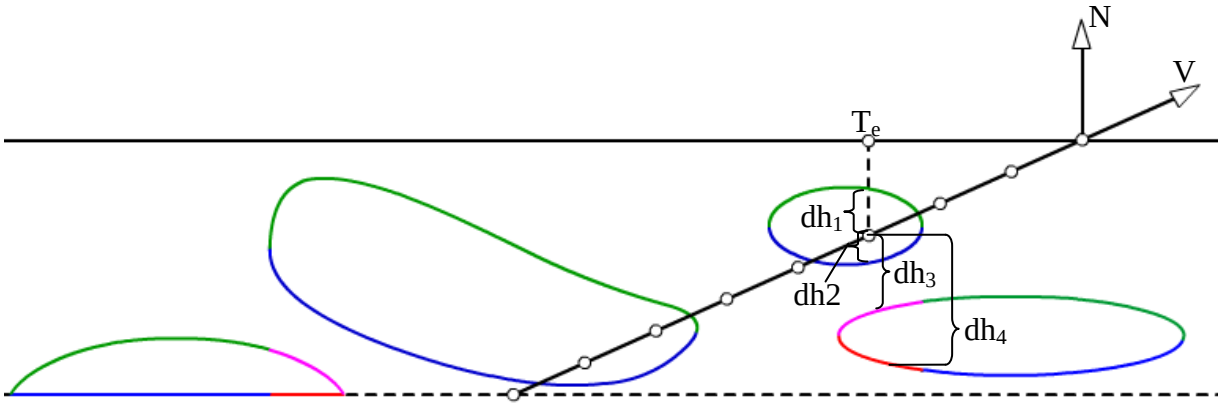
3.6.11 Nem magasságtérkép alapú rétegű relief mapping

A módszer lényege, célja hogy egymást átfedő felületeket tudjunk megjeleníteni [9]. Ilyen felület például egy láncszemekből összefűzött függöny, de tetszőleges alakzatot megjeleníthetünk, amit a 4 réteggel tudunk reprezentálni. A cím egy kicsit megtévesztő, ugyanis itt is magasságtérképeket használunk. A magasságtérkép értékei azt jelölik, hogy a

felület hol kezdődik és hol ér véget. Az 1. réteg tárolja a felület legfelső rétegének magasságát. A 2. réteg azt tárolja, hogy ha haladunk az egyenetlen felszín felől a poligon felszíne felé, milyen magasságban lépünk ki az egyenetlen felületből. A 3. és 4. réteg szintén ezt tárolja (38. ábra). Előfordulhat, hogy bizonyos részeken egy-két rétegre nincs szükség, akkor a magasságértéket 1-re állítjuk. Mivel a 4 réteg tárolásához a kép mind a négy csatornájára szükség van, ezért a normáltérképet külön kell tárolni. Minden magasságréteghez tartozik egy normáltérkép. Mivel a normáltérkép tárolásához 3 csatornára van szükség, ezért három képre lenne szükségünk. Azonban tudjuk, hogy egy egységvektor esetén

$$\sqrt{x^2 + y^2 + z^2} = 1\,, \qquad \text{amiből} \qquad z = \sqrt{1 - (x^2 + y^2)}$$

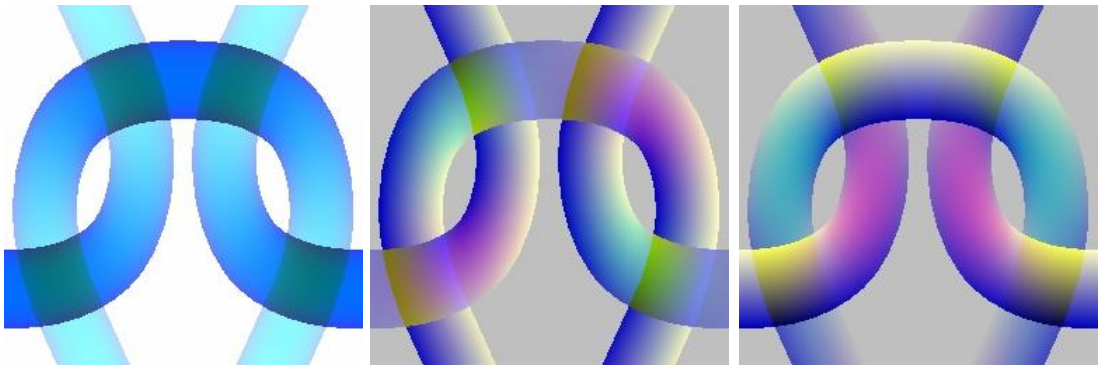
ezért, elegendő csak az x és y koordinátákat eltárolni, és z értékét pedig majd kiszámoljuk. A négyzetre emelés miatt az előjelét nem tudjuk ugyan meghatározni, de tudni fogjuk, hogy a felület melyik részén találtunk metszéspontot és innen tudjuk a normálvektor előjelét is. Ha olyan rétegnél találunk metszéspontot, ami a felület felső határát jelzi, azaz 1. vagy 3. rétegben, akkor a normálvektor a z koordinátája pozitív előjelű, ellenkező esetben pedig negatív előjelű. A metszéspont megtalálásához hasonló módszert alkalmazunk, mint az eddigi 2 rétegű megoldásoknál (39. ábra). A lineáris keresést addig folytatjuk, amíg be nem lépünk



39. ábra: 4 rétegű relief mapping magasságrétegek: 1. réteg zöld, 2. réteg kék, 3. réteg lila, 4. réteg piros

az egyenetlen felszínbe. Ez két esetben történhet meg. Vagy az 1. és 2. réteg között, vagy a 3. és 4. réteg között, mivel a páratlan réteg felső határ a páros pedig alsó határt jelöl. Tehát ha $dh_1 \times dh_2 < 0$ vagy $dh_3 \times dh_4 < 0$ akkor megállunk, mert metszéspontot találtunk. Ezt a két feltételt akár össze is vonhatjuk, mivel ha az első szorzat negatív elsőjelű, akkor a második biztos pozitív és fordítva, ezért ezek szorzata is negatív előjelű lesz. Tehát ha $dh_1 \times dh_2 \times dh_3 \times dh_4 < 0$, akkor megállunk. A metszéspont itt is ahhoz a réteghez fog tartozni, amelyikhez a legközelebb az adott osztópont. Miután meghatároztuk, hogy a metszéspont

melyik réteghez tartozik, vesszük a hozzá tartozó normálvektort és kiszámíthatjuk az árnyalást. Természetesen arról 1. magasságréteghez az 1. normáltérkép tartozik, a 2-hoz a 2. és így tovább.



40. ábra: a.) magasságtérkép b.) normáltérkép x c.) normáltérkép y

Az implementáláshoz a dupla rétegű relief mapping-ot fogjuk módosítani.

```
mélységi_eltolás := -0.5
a lineáris keresés
AMÍG lépés < 1
        mivel egy képen van mind a 4 réteg, egyszerre lekérdezhetjük mind a 4 értéket
        h most egy 4 elemű tömb, és ha hátsó lapot látunk, akkor a magasságrétegek
        inverzével számolunk
        h := TextúraMintavételező( magasságtérkép, VSOUT.UV + eltolás )
        hátsó oldal esetén, a magasságtérkép inverzével számolunk
        HA előjel < 0 AKKOR h := 1 - h
        ahogy a dh is, minden eleméből kivonjuk az aktuális magasságot
        dh := h - step;
        HA dh[1] * dh[2]* dh[3] * dh[4] >= 0 AKKOR
                lépés = lépés + ndh
                eltolás = { sb * lépés + mélységi_eltolás, st * lépés + mélységi_eltolás }
        VÉGE
VÉGE
NtsX, NtsY, NtsZ négy elemű tömökb
NtsX := TextúraMintavételező( normáltérképX, VSOUT.UV + eltolás )
NtsY := TextúraMintavételező( normáltérképY, VSOUT.UV + eltolás )
NtsZ := Négyzetgyök( 1 – ( NtsX* NtsX + NtsY* NtsY ) )
```

```
meghatározzuk melyik normálréteg értékére van szükségünk
ha a metszéspont az 1. és 2. réteg között van
HA dh[1] * dh[2] < dh[3] * dh[4] AKKOR
        ha a metszéspont az 1. rétegen van
        HA –dh[1] < dh[2] AKKOR Nts := { NtsX[1], NtsY[1], NtsZ[1] }
                ha a metszéspont az 2. rétegen van
                ELLENBEN Nts := { NtsX[2], NtsY[2], -1*NtsZ[2] }
        ha a metszéspont az 3. és 4. réteg között van
        VÉGE ELLENBEN
                ha a metszéspont az 3. rétegen van
                HA –dh[3] < dh[4] AKKOR Nts := { NtsX[3], NtsY[3], NtsZ[3] }
                        ha a metszéspont az 4. rétegen van
                        ELLENBEN Nts := { NtsX[4], NtsY[4], -1*NtsZ[4] }
VÉGE
a normálvektor tehát
Nts.z := előjel * Nts.z
```

Az előnyök és hátrányok felsorolásánál nem említettünk két fontos tulajdonságát a relief mapping-nak. A pixel shader során csak azokat a pixeleket tudjuk módosítani, amik az adott poligonhoz tartoznak. Az egyik probléma tehát, hogy a rajzfelületünk korlátozott. Ezért a poligon széleinél a kiemelkedéseket gyakorlatilag elhagyjuk. Amikor a felületet kiemeljük a poligon felé akkor a távolabbi széleknél lesz levágva a kiemelkedés. Amikor benyomjuk a felületet akkor pedig a közelebbi szélek, úgymond eltakarják a felületet. Hogy ezt a levágást elkerüljük, használjuk a 3.7.4-es módszert. A másik amit meg kell említeni, hogy a textúra koordinátákat a hardver mindig visszavezeti a [0,1] intervallumba. A (2.3, -1.6) textúra koordináta megegyezik a (0.3, 0.4) textúra koordinátákkal. Ezért amikor kicsit a betekintési szög, előfordulhat, hogy olyan hosszú a sugarunk, hogy a metszésponthoz nagyon távoli textúra koordináta fog tartozni, amit viszont a hardver visszaalakít. Ennek a következményeként, úgy látjuk minta a poligon alá benézve a végtelenbe tekintenénk. Megtehetjük azt, hogy azokat a texeleket, amiket túlságosan távolról hozunk be, elhagyjuk. Viszont a pixel shader során nem tudjuk, hogy mi az koordináta, aminél már el lehet hagyni a texelt, mivel nincs róla semmilyen információnk. Definiálhatjuk statikusan a pixel shader-ben a határokat - mert mondjuk tudjuk, hogy csak olyan négyszöglapokat fogunk megjeleníteni

aminél csak a 0 és 0.5 közé eső textúra koordináták a megengedettek -, de ezt ritkán alkalmazhatjuk. Egy jelenet megtervezésénél ezeket figyelembe kell venni. Nem szabad megfeledkezni a milyen korlátozások vannak.

4. Összefoglalás

Az érdes vagy egyenetlen felületek megjelenítése csak egy nagyon kicsi része azoknak a technikáknak, amelyekkel növelhetjük a számítógépes grafika minőségét, de nélkülözhetetlen ahhoz. A kezdeti emboss bump mapping-tól eljutottunk a relief mapping-hoz. Minden technikának megvolt az előny és a hátránya is és voltak korlátozások, amit figyelembe kellett venni. A számítógépes grafika kezdete óta, hardverek számítási teljesítménye szabja meg, hogy mely technikák kezdenek elterjedni. A Normal mapping-ra azt mondtuk, hogy gyors. Viszont ezt csak azért tehettük, mert mire megjelent az a hardver, amin implementálható volta, addigra a GPU-k számítási teljesítménye már nagyon nagy volt. 1-2 év múlva a Relief mapping is csak egy általános effekt lesz a valós idejű megjelenítésben. Újabb és újabb megoldások fognak megjelenni.

Az emboss bump mapping-ot ma már nem nagyon használják, mivel elterjedtek azok a hardverek, amelyek támogatják shader model valamelyik verzióját. Ma már egy olyan monitorvezérlő, amely támogatja a shader model 3.0-át, átlagosnak számít. Márpedig, láthattuk, hogy pixel shader 3.0-ban már bármelyik effekt implementálható. A Normal mapping a megjelenése óta még mindig elegendő a jó eredmény elérésre mivel képes annyira pontosan megjeleníteni felületet, hogy az szemlélő csak speciális esetben látja meg a különbséget a nagy poligonszámú modell és a kis poligonszámú normál mapping-os modell között. A paralax mapping minimális számítási többletet igényel, de segítségével nagy mértékben kitolhatjuk azt a határt, ahol már nem tudja megközelíteni a nagy poligonszámú megoldást. A relief mapping önmagában is fejlődni fog és újabb és újabb változatai jelennek meg. Mivel sugárkövetésre épül, szinte korlátlan lehetőségeket rejt. Az alapelve ugyan már régóta ismert a Relief mapping a jelenlegi hardvereknek még komoly kihívást okoz. Itt most nem egy darab objektumra kell gondolni, hanem egy teljes komplex jelenetre ahol akár több szám objektum is lehet. A Relief mapping-ot akkor érdemes választani, ha fontos a felület korrekt megjelenítése, fontos hogy látható legyen a felület kiemelkedése. A kívánt minőség eléréséhez el kell dönteni, hogy melyik módszert válasszuk és hogy milyen paraméterekkel. Például a 4 Rétegű Relief mapping és képes megjeleníteni azokat a felületeket, amit az

egyszerű Relief mapping, de fölösleges számításokat fog végezni. Ne használjunk relief mapping-ot vetett árnyékkal, ha a felület milyensége vagy a fényviszony olyan, hogy nem fog árnyékot vetni a felület. A megjelenítés során használjuk több különböző módszert és optimalizáljuk a shader kódot az igényeinkhez.

Irodalomjegyzék

[1] Karhu, K., (2002). Displacement Mapping

[2] Michael, G., GDC (1999)., Emboss Bump Mapping

[3] Szirmay-Kalos , L., Umenhoffer, T (2006). Displacement Mapping on the GPU - State of the Art

[4] Welsh, T., (2004). Parallax Mapping with Offset Limiting: A PerPixel Approximation of Uneven Surfaces

[5] Policarpo, F., Oliviera, M.M., (2004), Comba J.L.D., Real-Time Relief Mapping on Arbitrary Polygonal Surfaces

[6] Risser, E. & Shah, M. & Pattanaik, S., (2004), Interval Mapping

[7] Tatarchuk, N. (2005) Dynamic Image Image-Space Per Per-Pixel Displacement Mapping with Silhouette Antialiasing via Parallax Occlusion Mapping

[8] Tatarchuk, N., Oa,t C., Sander, P.V., Mitchell, J.L., Wenzel, C., Evans, A., (2006). Advanced Real-Time Rendering in 3D Graphics and Games. SIGGRAPH 2006 Course 26

[9] Policarpo, F & Oliveira, M. M., (2006). Relief Mapping of Non-Height-Field Surface Details

Függelék

A mellékelt DVD tartalmazza az összes leírt technika implementációját HLSH nyelven, valamint a ShaderShower nevű programot, amely segítségével különböző beállításokkal és modellekkel tekinthetjük meg a technikákat.