
Production Systems and Information Engineering
Volume 13 (3), pp. 33–54.
https://doi.org/10.32968/psaie.2025.3.4

PER-PIXEL MEGVILÁGÍTÁS MULTI-OBJECT MODELLEK ESETÉN

MILEFF PÉTER
Miskolci Egyetem, Informatikai Intézet
Általános Informatikai Intézeti Tanszék
peter.mileff@uni-miskolc.hu

Absztrakt. A számítógépes grafikában a per-pixel világítás széles körben alkalmazott technika a valósághű fényhatások elérésére, amely a fény kölcsönhatásait pixelenként számítja ki. Ez a megközelítés lehetővé teszi a felületi jellemzők, például a textúrák és a finomabb részletek megjelenítését, amelyek kulcsfontosságúak a magas minőségű vizuális élményhez. A per-pixel világítás továbbfejlesztésére gyakran alkalmazzák a normal map technikát, amely bonyolult felületi részleteket képes szimulálni a geometriai komplexitás növelése nélkül. Azonban a pontos normálvektor-számítások kihívást jelenthetnek olyan modellek esetén, amelyek több objektumból állnak, ahol különböző orientációjú és geometriájú szomszédos felületek osztoznak ugyanazokon a csúcsokon. Jelen cikk egy olyan módszert mutat be, amely helyes normálvektorok számítását teszi lehetővé többobjektumos modellekhez, biztosítva a vizuálisan konzisztens és valósághű fényhatásokat komplex jelenetekben. A javasolt technika olyan gyakori problémákra ad megoldást, mint a normál interpolációs problémák és az abból fakadó fény inkonzisztenciák. Az eredmények bemutatják, hogy a módszer jelentősen javítja a per-pixel világítás vizuális minőségét normal map technikával kombinálva, és robusztusabb megoldást nyújt a valós idejű rendereléshez.

Keywords: *pixelenkénti megvilágítás, normál simítás, multi-object modellek*

1. Bevezetés

A számítógépes grafika napjainkra a modern technológia szerves részévé vált, és számos iparágra gyakorol jelentős hatást a szórakoztatástól és a virtuális valóságtól kezdve a tudományos vizualizáción át egészen az ipari tervezésig. A valósághű és magával ragadó virtuális 3D-környezetek létrehozásának képessége nemcsak forradalmasította a vizuális médiát, hanem jelentősen növelte a lehetőségeket összetett rendszerek szimulálására és elemzésére olyan területeken, mint a mérnöki tudományok, az orvostudo-

mány vagy az oktatás. Ahogy az egyre élethűbb és dinamikusabb vizualizációk iránti igény nő, a 3D-környezetek megjelenítéséhez használt technikáknak folyamatosan fejlődniük kell, hogy megfeleljenek a realizmus és a teljesítmény elvárásainak.

A 3D-vizualizáció egyik legkritikusabb eleme a fény pontos szimulációja. A világítás nem pusztán esztétikai elem; alapvetően meghatározza, hogyan érzékeljük a formákat, textúrákat és térbeli kapcsolatokat egy virtuális jelenetben. Az, ahogyan a fény kölcsönhatásba lép a felületekkel – árnyékokat vet, visszatükröződik a tárgyakról, vagy megtörik az átlátszó anyagokon – jelentősen hozzájárul a vizuális realizmushoz és a jelenet immerszív minőségéhez. Hatékony világítás nélkül még a legaprólékosabban kidolgozott 3D-modellek is laposnak és élettelennek tűnhetnek, ami rontja a néző képességét a vizuális információ pontos értelmezésére [9].

Többobjektumos 3D-modellek esetén, ahol számos felület és anyag osztozik egy közös teren, a világítási számítások bonyolultsága jelentősen megnő. Minden egyes objektum nemcsak különböző fényforrásokból kap megvilágítást, hanem maga is befolyásolja a környezet fényviszonyait árnyékok vetésével, a fény szomszédos objektumokra való visszaverésével és a globális megvilágítási hatásokhoz való hozzájárulással. Ezeknek a kölcsönhatásoknak a pontos modellezése alapvető fontosságú a fotórealisztikus megjelenítés eléréséhez, amely számos területen alkalmazható, a filmes speciális effektektől kezdve az építészeti vizualizáción át a virtuális prototípus-készítésig.

Jelen cikk célja, hogy bemutassa a többobjektumos 3D-modellek világításának számítására és optimalizálására szolgáló technikákat. A komplex világítási helyzetekből adódó kihívások kezelése révén növelhető a 3D-vizualizációk valósághűsége, ezáltal mind művészeti, mind tudományos célokra hasznosabbá téve azokat.

2. Fények a számítógépes vizualizációban

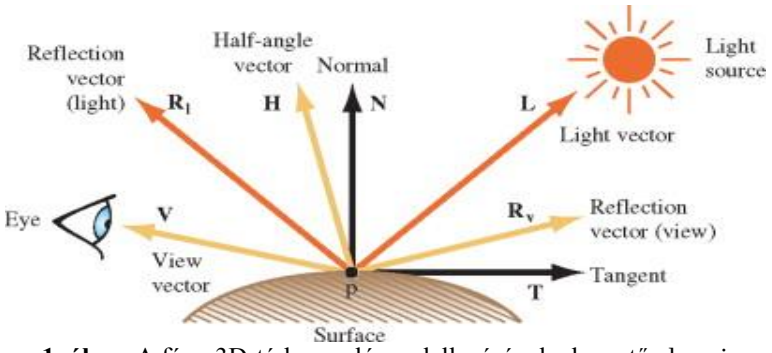
A fények modellezésének igénye már a számítógépes vizualizáció kezdetén megjelent. Kezdetben offline megvalósításra volt lehetőség a korlátozott hardverek és az alacsony számítási teljesítménynek köszönhetően. Később azonban a hardverek fejlődésével párhuzamosan merült fel a fények valós időben való modellezése. Manapság pedig már egyre kifinomultabb megoldások állnak rendelkezésre.

A világítás modellezése alapvető része a számítógépes grafikának, kulcsfontosságú szerepet játszik egy jelenet valósághűségének, hangulatának és térbeli mélységének meghatározásában. Jelentős hatással van az objektumok és környezetek érzékelésére azáltal, hogy szimulálja, hogyan lép kölcsönhatásba a fény a felületekkel. A 3D-grafikában a megvilágítási modellek a fény felületekre gyakorolt hatásait számítják ki, beleértve a spekuláris kiemeléseket, az árnyékokat és a tükröződéseket. Ezek a hatások segítenek létrehozni a mélység illúzióját és fokozzák a jelenetek valósághűségét. Az olyan fejlett technikák, mint a globális illumináció és a sugárkövetés, figyelembe veszik a fény bonyolult kölcsönhatásait, például a szóródást, a törést és a közvetett megvilágítást, tovább javítva ezzel a vizuális minőséget.

2.1. Megvilágítási modellek

A valós világban az objektumok megjelenésére jelentős hatással vannak a fényforrások. Ezek a hatások számítógépes környezetben megvilágítási modellek segítségével szimulálhatók. A megvilágítási modell olyan egyenletek halmazát reprezentálja, amelyek közelítik (modellezik) a fényforrások objektumokra gyakorolt hatását. A számítástechnikában a megvilágítási egyenlet egy olyan matematikai összefüggéshalmaz, amelyet az objektum végső színértékének kiszámítására használnak. Ez az egyenlet segít meghatározni, hogyan lép kölcsönhatásba a fény a jelenet objektumaival a valósághű vizuális hatások elérése érdekében.

A megvilágítási modell tartalmazhatja a fényforrás visszaverődését, elnyelését és áteresztését is. A modell egy objektum felületének egy pontján számítja ki a színt a fényforrások, az objektum pozíciója és felületi jellemzői, valamint esetenként a néző helyzete és a környezet (például más tükröződő objektumok vagy a légköri tulajdonságok) figyelembevételével.



1. ábra. A fény 3D-térben való modellezésének alapvető elemei

A hagyományos megközelítés a valós idejű számítógépes grafikában a fény számítására az, hogy a fényt egy csúcspontban (vertex) a környezeti (ambient), diffúz és spekuláris komponensek összegeként számoljuk ki. A legegyszerűbb formában (amelyet az OpenGL és a Direct3D is használ) a függvény egyszerűen ezeknek a világítási komponenseknek az összege (egy maximális színértékre korlátozva). Ennek megfelelően van egy ambiens tagunk, majd a fényforrásokból érkező diffúz és spekuláris fények összege [15].

$$i_{total} = k_{aia} + \sum (k_{did} + k_{sis})$$

ahol az i_{total} a fény intenzitása (RGB-értékként), amely a globális ambiens komponens intenzitásának, valamint a fényforrásokból származó diffúz és spekuláris komponensek összegének eredménye. Ezt lokális világítási modellnek nevezzük, mivel a csúcspontot csak a fényforrás világítja meg, más objektumról származó fényt nem vesz figyelembe. Az évek során számos megvilágítási modellt fejlesztettek ki különböző célokra. Ezek a modellek az egyszerű közelítő megoldásoktól a

rendkívül kifinomult algoritmusokig terjednek, ahol az egyes megoldások a számítási hatékonyság és a vizuális realizmus között próbálnak egyensúlyozni. A legismertebb modellek közé tartoznak:

Megvilágítási modellek típusai[5] [13]:

1. **Phong-világítási modell:** Az egyik legkorábbi és legszélesebb körben használt világítási modell. A Phong-modell bevezeti a spekuláris kiemelések (fényes pontok), az ambiens fény és a diffúz visszaverődés fogalmát. Egyszerűsíti a fény számítását azáltal, hogy a visszaverődéseket a néző perspektívája alapján számolja ki, így ideális valós idejű rendereléshez.
2. **Blinn–Phong-modell:** A Phong megvilágításra alapuló változat, amely kisebb módosítást alkalmaz a spekuláris visszaverődés számításában. Ennek eredményeként jobb teljesítményt és pontosabb kiemeléseket biztosít szélesebb látószög tartományban.
3. **Lambert-féle diffúz visszaverődés modell:** Ezt a modellt diffúz visszaverődés szimulálására használják, ahol a felület minden irányba egyenlően szórja a fényt. A modell alapja a Lambert-féle koszinusz törvény, amely kimondja, hogy a fény intenzitása arányos a fényforrás és a felületi normál közötti szög koszinuszával.
4. **Fizikai alapú renderelés (PBR):** Az utóbbi években a kutatás a PBR felé mozdult el, ahol a világítást a valós fizikai tulajdonságokhoz közelebb álló módon modellezik. A PBR olyan fogalmakat integrál, mint az energiamegmaradás, a Fresnel-hatások és a mikrofelületi elmélet, így rendkívül élethű világítási szimulációt nyújt, amely ma már alapvető a modern videójátékok és a filmipar számára.
5. **Oren–Nayar-modell:** A Lambert-modell kiterjesztése, amely a durva felületeket szimulálja a mikroárnyékolások által okozott fényszóródás figyelembevételével.
6. **Ward-anizotróp modell:** Ezt a modellt az anizotróp visszaverődések kezelésére tervezték, amelyek olyan felületeken fordulnak elő, mint a szálcsiszolt fém vagy a haj, ahol a visszaverődés irányfüggő módon változik.

Megvilágítási technikák [5]:

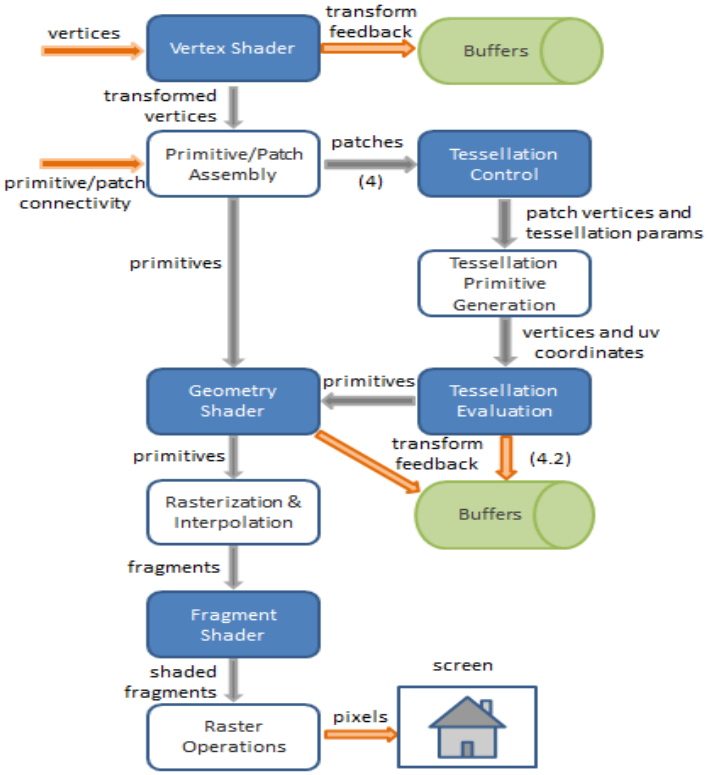
1. **Globális illumináció (GI):** A GI-technikák szimulálják, hogyan lép kölcsönhatásba a fény több felülettel egy jelenetben, lehetővé téve a fény visszaverődését és szóródását. Ez hozzájárul a realisztikusabb környezeti fények és lágy árnyékok létrehozásához. A globális illumináció megvalósításához gyakran használt módszerek közé tartozik a sugárkövetés (ray tracing) és a radiosity.
2. **Sugárkövetés (Ray Tracing):** Ez a technika egyedi fénysugarak útját követi végig, ahogy azok áthaladnak a jeleneten, és kölcsönhatásba lépnek a felületekkel, anyagokkal és fényforrásokkal. Bár számításigényes, a sugárkövetés rendkívül pontos tükröződéseket, fénytöréseket és árnyékokat eredményez.

3. **Radiosity:** A radiosity a diffúz felületek közötti fénycsere modellezésére összpontosít. Kiszámítja, hogyan verődik vissza a fény a felületekről, és ez miként járul hozzá a jelenet teljes megvilágításához. A radiosity különösen hatékony belső terek és építészeti modellek renderelésénél, ahol a közvetett világítás kiemelt szerepet játszik.

2.2. Hogyan illeszkedik a megvilágítás a grafikus csővezetékbe

A grafikai csővezeték (graphics pipeline) egy olyan koncepcionális keretrendszer a számítógépes grafikában, amely definiálja a 2D- vagy 3D-képek előállításának lépéseit egy jelenet leírásából kiindulva. Azért nevezik „csővezetéknek", mert az adatok több, egymást követő feldolgozási szakaszon haladnak át, ahol különböző műveletek történnek: a geometria transzformálása, textúrák és színek alkalmazása, valamint a világítás, árnyalás és a végső kép kiszámítása [2]. A *fix funkciós* grafikus csővezeték és a *programozható grafikus csővezeték* két különböző megközelítést képviselnek arra, hogyan kezeli a grafikus hardver a világítást, árnyalást és egyéb renderelési feladatokat. Míg a fix funkciós pipeline előre meghatározott, korlátozott funkcionalitást kínál, addig a programozható pipeline jóval nagyobb rugalmasságot és kontrollt biztosít a fejlesztők számára. Manapság már szinte kizárólag a programozható grafikus csővezetéket alkalmazzuk a számítógépes vizualizációban. A programozható shaderek (vertex shader, fragment/pixel shader) megjelenésével a modern GPU-k lehetővé teszik a renderelési folyamat minden egyes lépésének testreszabását, beleértve a világítás számítását is.

Az alábbi ábra az OpenGL 4 grafikus csővezetékét mutatja be [8].



2. ábra. The OpenGL 4 csővezeték

A korábbi fix funkciós csővezetékek esetében előre programozott fénymodellek használatára volt lehetőség. A Figure 2 alapján látszik, hogy ma az alkalmazott fény algoritmusának megvalósítása a programozói oldalra tolódott. Ez a rugalmasság sokkal kifinomultabb és élethűbb megvilágítási hatásokat tesz lehetővé. A programozó feladata vertex és fragment árnyalók segítségével az adott fénymodell kidolgozása és programozása [6] [7]. A következő táblázat a két megközelítést hasonlítja össze.

1. táblázat
Fix és programozható csővezeték összehasonlítás

Tulajdonság	Fix funkciós csővezeték	Programozható csővezeték
Megvilágítási modell	Előre definiált modellek (pl. Phong)	Customizable lighting models, including PBR
Fényforrások száma	Limitált (általában max 8)	Customizable lighting models, including PBR
Fény számítás	Vertexenkénti, felületek között interpolálva	Pixelenkénti (részletesebb és realisztikusabb)

Árnyékok és effektek	Limitált vagy nem létező	Valós idejű árnyékok, globális illumináció, egyedi effektek
Vezérlés és rugalmasság	Little to no control over lighting equations	Full control over how lighting is calculated
Felületi részletesség	Alap megvilágítás; limitált lehetőség a finom részletek megjelenítésére	Finom részletek megjelenítése bump mapping, normal mapping, stb segítségével.
Árnyaló használat	Nincs shader, hard kódolt operációk	Egyedi vertex és pixel árnyalók
Teljesítmény	Gyorsabb, de kevésbé realisztikus	Lassabb, de realisztikusabb

2.3. Fény a virtuális térben

A számítógépes grafikában a megvilágítási modellek kulcsszerepet játszanak a valósághű képek előállításában azáltal, hogy szimulálják, hogyan lép kölcsönhatásba a fény a felületekkel. A fény számítására két széles körben alkalmazott technika a vertex-alapú világítás (más néven Gouraud-árnyalás) és a per-pixel világítás (amelyet általában Phong-árnyalással vagy fejlettebb pixel shaderekkel valósítanak meg). Mindkét megközelítésnek megvannak a maga jellegzetességei, erősségei és korlátai, amelyek meghatározzák alkalmazhatóságukat különböző kontextusokban [3].

2.3.1. Vertexalapú megvilágítás

A vertexalapú megvilágítás a 3D-modell csúcspontjain (vertex-ein) számítja ki a fényt, majd ezeket az értékeket interpolálja a poligon felületén. Ez a módszer számítási szempontból rendkívül hatékony, mivel a számításokat csak a modell csúcspontjain kell elvégezni, ami jelentősen csökkenti a szükséges műveletek számát. A vertexalapú világítás hatékonysága különösen alkalmassá teszi valós idejű alkalmazásokhoz, ahol a renderelési sebesség kritikus tényező, például régebbi videójátékoknál vagy alacsony teljesítményű eszközökön futó programoknál. A technika megvalósítása egyszerű és kevésbé igényel számítási kapacitást, ami a számítógépes grafika hajnalán, a korlátozott hardveres képességek idején különösen előnyös volt.

Mindazonáltal a vertexalapú világításnak jelentős korlátai vannak. Mivel a fény interpolációval kerül kiszámításra a csúcspontok között, előfordulhatnak vizuális hibák és a részletek hiánya, különösen alacsony tesszellációjú felületek esetén. Például a spekuláris kiemelések (csillanások) elmosódottnak vagy fakónak tűnhetnek. Emellett a vertexalapú világítás nehezen kezeli a komplex világítási szituációkat, mint például az éles tükröződések, spotfények vagy finom felületi részletek, mivel az interpoláció nem képes pontosan lekövetni a fény intenzitásának vagy színének gyors változásait.

2.3.2. Per-pixel megvilágítás

A per-pixel megvilágítás, más néven Phong-árnyalás, ezzel szemben minden egyes pixelre külön-külön végzi el a világítási számításokat a felületen, lehetővé téve a fény és az anyagtulajdonságok közötti sokkal finomabb kölcsönhatás szimulálását. Ez a módszer sokkal részletesebb és valósághűbb képeket eredményez. A per-pixel világítás elsődleges előnye a magasabb vizuális minőség: jóval pontosabb és részletesebb világítási hatásokat nyújt, mint a vertexalapú módszerek, képes precízen modellezni a spekuláris kiemeléseket, tükröződéseket és más összetett fényjelenségeket, amelyek jelentősen növelik a realizmust [10].

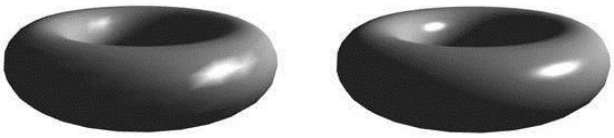
A modern számítógépes grafikában a per-pixel világítást programozható shaderekkel valósítják meg. Ez a megközelítés lehetővé teszi fejlett effektusok, például bump mapping, normal mapping és dinamikus árnyékok létrehozását, amelyek ma már minden modern videójáték, szimuláció és virtuális valóság 3D-renderelésének alapkövei.

Mindazonáltal a per-pixel világítás számítási költsége magasabb. Mivel minden pixelre el kell végezni a világítási számításokat, lényegesen több feldolgozási teljesítményt és memória-sávszélességet igényel, ami korlátozó tényező lehet valós idejű alkalmazásokban, különösen gyengébb hardvereken. Emellett a per-pixel világítás implementálása gyakran bonyolultabb algoritmusokat és shader-programozást igényel, ami növeli a fejlesztési időt és a komplexitást.

2.3.3. A két megközelítés közötti választás

A vertexalapú és a per-pixel modellek közötti választás az adott alkalmazás specifikus követelményeitől, a kívánt vizuális részletességtől és a rendelkezésre álló hardver erőforrásoktól függ. A vertexalapú megvilágítás továbbra is hasznos olyan helyzetekben, ahol az egyszerűség és a sebesség fontosabb a részletes vizuális hatásoknál, például mobiljátékoknál vagy alacsony poligonszámú modelleket használó alkalmazásoknál. Ezzel szemben a per-pixel világítást részesítik előnyben ott, ahol a magas vizuális minőség alapvető, például modern videójátékokban, filmes CGI-jeleneteknél vagy virtuális valóság élményekben.

A vertexalapú világítástól a per-pixel világítás felé történő fejlődés jól tükrözi a számítógépes grafika általános törekvését a fotórealizmus elérésére [15]. Ahogy a hardverek – beleértve a CPU-kat és GPU-kat – teljesítménye nőtt, lehetővé vált a komplexebb és számításigényesebb technikák, például a per-pixel világítás alkalmazása, amelyek jelentős javulást hoznak a vizuális minőségben. Napjainkban sok grafikus motor dinamikusan állítja a részletességet, kombinálva a vertexalapú és a per-pixel világítási technikákat a teljesítmény optimalizálása mellett a magas vizuális minőség megőrzése érdekében.



3. ábra. Vertex vs. per-pixel megvilágítás

Jelen cikk a per-pixel megvilágításra fókuszál a továbbiakban.

3. Per-pixel megvilágítás multi-object modellek esetén

A fények területe és modellezése a számítógépes vizualizáción belül már egy komplexebb környezetet igényel. Általában ilyenkor már készül egy olyan minimális grafikus (esetleg egy játékmotor részeként), amely már valamilyen strukturált formában képes az alábbi funkciókra:

- különböző modelleket betölteni
- azok adatait egy megfelelő belső struktúrában tárolni
- rendelkezik valamilyen, a „virtuális világ”-ot leíró megközelítéssel, struktúrával
- több kamerát kezelni, melyek akár mozgathatók is (pl. FPS kamera)
- az alapvető shaderek kezelésére
- képes a fényeket és azok paramétereit egy belső objektumstruktúrával leírni
- rendelkezik (egyszerű) matematikai könyvtárral
- esetleg egyéb utility részekkel, különböző segédszolgáltatásokkal

Láthatjuk, hogy azért itt már egy jelentősebb munkára van szükség ahhoz, hogy egy olyan tesztkörnyezetet tudjunk kialakítani, ahol akár egy-egy rövidebb kódsor módosításával képesek vagyunk egy új beállítást tesztelni, vizuálisan megtekinteni.

3.1. A fény alkalmazásának folyamata

Természetesen a fények gyakorlati implementációja függ az alkalmazott API-tól, amely jelenleg OpenGL, Vulkan vagy DirectX-et jelent. És akár idevehetjük a szoftveresen megvalósított fények megoldását is. Azonban függetlenül attól, hogy milyen grafikus API-t használunk, a virtuális világ fénybe borítása, számos dolog előkészítését igényli.

Elsősorban szükségünk van a virtuális világot alkotó geometriai információra (épületek, tárgyak, vegetáció stb.), melyeket leggyakrabban egy vagy több valamilyen modellfájlból töltjük be. Bár számos különböző modell-leíró struktúra / formátum (Max, Blend, gITF, Obj, FBX, ASE stb.) létezik, az alapvető geometriát amelyre a fényeknek szüksége van, mindegyik tartalmazza. Amennyiben az alkalmazás túlmutat a programozási környezet sandboxába beégetett világtól, úgy általában valamilyen világleíró struktúra is alkalmazásra kerül, amely már a térben is elhelyezi a modelleket és egyben definiálja azokat a fényforrásokat (paraméterekkel együtt), amelyek a virtuális világ megvilágítását képezik. A geometriai adatok előkészítési fázisának lezárását jelenti az, amikor azok feltöltésre kerülnek a GPU

memóriájába.

A kirajzolás folyamata számos modul/elem meglétét követeli meg, amelyre jelen cikkben nem térünk ki. Fontos azonban kiemelni a shadereket, melyek már a konkrét kirajzolást fogják elvégezni.

Természetesen a fényeket számos módon lehet implementálni. Bizonyos részek (pl. mátrixok) számolhatók a CPU-oldalon, vagy akár a GPU-oldalon is. Jelen cikk ezekre nem tér ki. Amennyiben minden, a működéshez szükséges elem rendelkezésre áll, úgy a kirajzolás folyamata a rendering cikluson belül a következő:

```
Projekciós, nézeti és modelmátrix számítása
      for loop - model.object.size

         light shader aktiválása
           for loop - numberOfLights
              Fény paramétereinek átvitele a GPU árnyékolóba:
                  - position, constant linear, quadratic
                    attenuation, ambient, diffuse, specular, stb
              end for
              Objektum textúrájának beállítása
              Objektum egyéb paramétereinek beállítása
              object i. rajzolása
           light shader vége
      end for
```

3.1. A normál vektorok fontossága

Egy fényforrás felületre gyakorolt vizuális hatása nemcsak a felület és a fényforrás tulajdonságaitól függ, hanem nagymértékben attól is, hogy milyen szögben éri a fény a felületet. Ez a szög alapvető a spekuláris (tükröződő) visszaverődéshez, és befolyásolja a diffúz visszaverődést is. Emiatt néz ki egy megvilágított, görbült felület különbözően a különböző pontokon, még akkor is, ha a felület színe mindenhol azonos. A szög kiszámításához ismernünk kell, hogy a felület milyen irányba néz. Ezt az irányt egy, a felületre merőleges vektor határozza meg. A „merőleges” másik elnevezése a „normális”, és egy adott pontban a felületre merőleges, nem zéró vektort normálvektornak nevezzük.

Fontos szempont tehát, hogy a normálvektorok megfelelően legyenek kiszámítva ahhoz, hogy a fények az elvárásoknak megfelelő vizuális eredményt nyújtsanak. A vektorok általában két úton kerülnek megadásra. Nagyon gyakran a normálvektorok a háttértárolón tárolt modellfájllal együtt kerülnek letárolásra egy fájlban. Ilyenkor az adatok konzisztenciájáért az alkalmazott modellszerkesztő szoftver felelős. Ebben az esetben a grafikus motornak más feladata nincs, minthogy beolvassa ezeket az adatokat is és alkalmazza azokat.

Másik megoldásként a normálvektorokat kiszámolhatjuk a modell betöltése során is saját magunk. Bár ez a megközelítés elengedhetetlen a fejlesztés során, kész játékok esetében inkább a modellel együtt való tárolást választják azért, mert egy-egy nagyobb modell esetén a normálok számítása és transzformálása akár másodpercekben mérhető ideig is eltarthat. Ez pedig jelentős hatással lehet a nyújtott játékélményre.

Alapjában véve két típusú normálvektort szokás megkülönböztetni:

- **Felületi normális:** Ez a vektor egy felület- (általában háromszög) szintű vektor, amely a felületre merőleges egységnyi normálvektor. A vektor irányát a csúcspontok megadásának sorrendje, valamint a koordináta-rendszer jobb- vagy balkezes volta határozza meg. A felületi normál a felület elülső oldala felől kifelé mutat. Egy felülethez egy normálvektor tartozik.
- **Csúcsponti normális:** A csúcsponti normálok (néha pszeudo-normáloknak is nevezik) olyan értékek, amelyeket minden csúcspontnál tárolnak, és amelyeket leggyakrabban a renderelő használ a világítási vagy árnyalási modellek (például Phong-árnyalás) visszaverődéseinek meghatározására.

4. Normal mapping

A számítógépes vizualizációban mindig fontos szerepet töltött be grafikai realisztikusság növelése. Valós idejű szoftverek, játékok esetében ez különösen hangsúlyos, hiszen az offline rendereléshez képest itt valós időben kell a kornak megfelelő grafikai színvonalat elérni. Ennek egyik kézenfekvő megközelítése az, ha növeljük a virtuális világ és a modellek poligonjainak számát. A megközelítés bár helyes és jól alkalmazható, a vertex szám növelése azonban nagy terhet ró (főként a korai) a GPU-k számára: lényegesen több poligon megy át a grafikus csővezetéken, több poligont kell vágni, raszterizálni stb. [12]. Másik megközelítés, amelyet valójában a poligonszám-növeléssel együtt célszerű alkalmazni az, ha nagyobb, finomabb felbontású textúrákat alkalmazunk. Amennyiben azonban a felületeket közelről nézzük, mint amikor egy játékban egy falhoz kerülünk, ez a megközelítés sem jelent drasztikus javulást. Ennek oka, hogy a valóságban a felületek nem simák, számos lyuk, mélyedés vagy kidomborodás található rajtuk, melyeket a fény különböző beesési szögei miatt látunk különböző árnyalatúnak, színűnek. Ezt a két megközelítést kombinálva tehát megtehetnénk azt is akár, hogy a fal minden kis felületi érdességét vertexhálóval modellezzük le, amelyre magasabb minőségi textúrát húzunk, azonban ezt a mai GPU-k még nem képesek kezelni, az eredmény bár látványilag szép, de sebességben lassú lenne. Tipikus példa lehet egy téglafal. Egy ilyen fal meglehetősen durva felületű, és biztosan nem teljesen sima: tartalmaz mélyített cementrészeket és számos kisebb lyukat és repedést. Amennyiben egy ilyen felületet a megszokott megjelenítéssel, effektek nélkül látunk, a fal mélységérzete eltűnik.

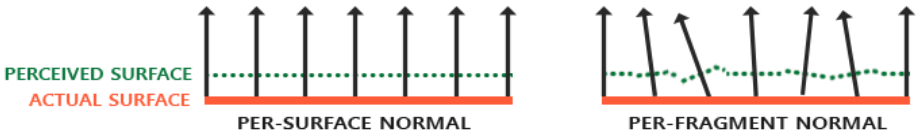
Ebből kifolyólag a felületek realisztikusabbá tételének más alternatív megoldásai is kialakultak. Az ilyen megoldások általában a képtérben dolgoznak és további tex-

túrákat használnak fel arra, hogy egyébként egy sima 2D-poligon számára valamilyen jellegű extra részletességet adjanak. Az irodalomban több megoldás is kialakult (bump mapping, displacement mapping, normal mapping, parallax mapping), melyek sikeresen képesek „mélységet” adni a 2D-képnek. Jelen cikkben a továbbiakban a normal mappingra fókuszálunk, de a bemutatottak a többi módszerre is alkalmazhatók.

4.1. Normal mapping a gyakorlatban

Amikor a klasszikus per-pixel lightingot alkalmazzuk, akkor a fragment árnyalóban pixelszinten került kiszámításra a fény intenzitása. Ehhez vertexszintű normálisokkal dolgozunk. A három vertex által alkotott barycentrikus koordináta-rendszernek megfelelően képesek vagyunk az árnyaló belsejében lineárisan interpolálni a normálisokat, amely során megkapjuk pixelszintű normálisokat. Ezzel szép, egyenletes fényszóródás érhető el az adott felületen, azonban a modell nem lép kölcsönhatásba a felületi anyagtulajdonságokkal, gödrökkel és repedésekkel. Főként szembetűnő ez a téglák közötti mély barázdákon. A felület egyszerűen sima marad.

Az iparnak ezért egy olyan megoldásra volt szüksége, amely képes a fény rendszert informálni a felület mélységeinek részleteiről. Nem elegendő önmagában az, hogy a háromszögön belül interpoláljuk a normálisokat. Olyan megoldásra van szükség, amely valódi pixelszintű normálisokat biztosít. Ezzel a technikával a felület sokkal komplexebbé tehető:



4. ábra. Felületi normális és a fregmentenkénti normális összehasonlítása [4]

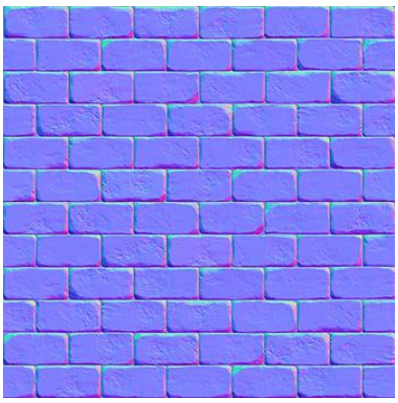
Ezzel a megoldással olyan vizuális hatást érhetünk el, amellyel a felület mélységérzetet kap az egyedi fényvisszaverődéseknek köszönhetően. A technika lényegében egy trükk, amely a néző számára egy sokkal realisztikusabb élményt nyújt. A megközelítést összefoglalóan normal vagy bump mappingnak nevezzük.

A normál mapping megvalósításához tehát szükségünk van a pixelenkénti normálisokra. Egy kézenfekvő megközelítés, ha diffúz textúrához hasonlóan egy kétdimenziós textúrát használunk fel a felületi normálisok perturbációjának tárolására. Mivel a normálvektorok geometriailag értelmezhetők és a textúrák pedig színinformációkat tárolnak, valamilyen úton meg kell feleltetni a vektorokat az egyes színeknek.

A színek *r, g, b* komponensekből tevődnek össze. Ezeket a komponenseket felhasználhatjuk a normálisok x, y, z komponenseinek tárolására is. Mivel a normálvektorok értékkészlete –1 és 1 között van, így első lépésként ezt a [0,1] tartományra kell transzformálnunk.

```
vec3 rgb_normal = normal * 0.5 - 0.5; // transforms from [-1,1]
to [0,1]
```

Ezzel a megközelítéssel lehetővé válik, hogy egy felület normálisait pixelenként tároljuk egy 2D-textúrában. Az alábbi példa a téglafal normal mapját mutatja be:



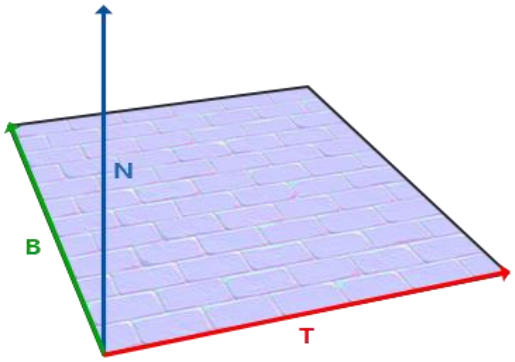
5. ábra. Normális értékek RGB textúrában tárolva

Láthatóan a normal map kékes színezetű. Ez azért van, mert majdnem minden normális kifelé mutat a felületből a pozitív z irányba (0,0,1), amely kékes színt jelent. A színbeli eltérések ott figyelhetők meg, ahol a felület eltér a megszokott simától. Ezeken a pontokon a normálisok eltérnek a pozitív z iránytól, ezek a részek fogják a mélységérzetet adni a néző számára. Figyeljük meg a téglát tetejét, amely majdnem mindenhol zöldes árnyalatot kap. Ez ezért van, mert ezeken a pontokon a normálisok mindinkább a pozitív *y* irányba (0,1,0) mutatnak, amelynek leképzése a zöld szín.

4.2. Tangens tér

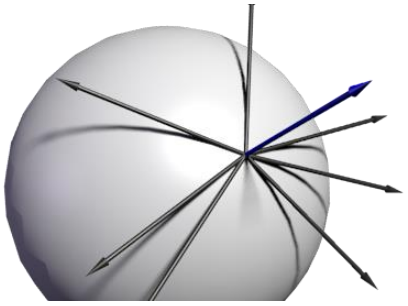
A tangens tér egy olyan tér, amely a háromszög felületéhez képest lokálisan értelmezett ortogonális rendszer. A normálisok ezen referencia koordináta-rendszerben relatívan vannak megadva. Felfoghatjuk úgy, hogy ez a lokális tér a normal map vektorainak tere, ahol minden normális úgy van definiálva, hogy a pozitív z irányba mutasson. A tangens teret definiáló speciális mátrix segítségével a lokális térben értelmezett normálvektorokat a világ vagy kamera térbe az adott felület végleges orientációjának megfelelően.

Ezt a mátrixot TBN-mátrixnak nevezzük, melynek összetevői a *tanges, bitangens* és *normál* vektorok. A vektorok ortogonális rendszert alkotnak, azaz merőlegesek egymásra. A normálvektor a felületből kifelé mutat, a jobbra és előre mutató vektor pedig a *tangens* és *bitangens*. A következő ábra a három vektor kapcsolatát mutatja be:



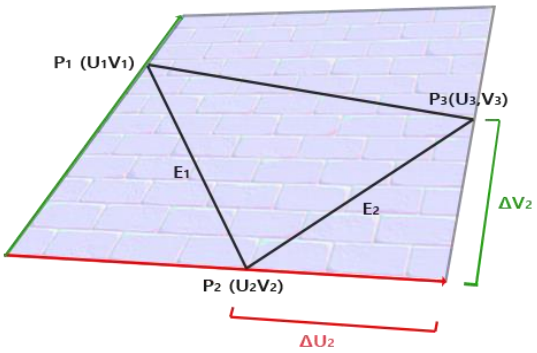
6. ábra. Tangens tér

Míg a normálvektor számítása meglehetősen egyszerű, addig a tangens és bitangens vektorok számítása nem magától értendő. Azt tudjuk, hogy a tangens vektor merőleges a normálvektorra. Azonban számos ilyen definiálható:



7. ábra. Tetszőleges számú tangens vektor definiálható a kékkel jelzett normálvektor mellé

Elméletileg bármely merőleges vektor alkalmazható tangens vektornak, de célszerű konzisztensnek lenni a szomszédos face-ekkel, különben a face-ek közötti szélek csúnyák lesznek az árnyékolás során. Az elfogadott megoldás az, hogy a tangens vektort abba az irányba irányítjuk, amerre a textúrakoordináták is mutatnak. A tangens vektor meghatározása után a bitangens már könnyen kiszámítható. Az alábbi ábra ezt mutatja be:



8. ábra. A tangens vektor orientációja mindig a textúra koordináták irányának felel meg

Az ábrán a háromszög E_2 élének textúrakoordináta-különbségét ΔU_2-vel és ΔV_2-vel jelöltük. Irányuk megegyezik a tangens (T) és bitangens (B) vektorok irányával. Ezek alapján mindkét él (E_1, E_2) felírható a T és B vektorok lineáris kombinációjaként:

$$E_1=\Delta U_1 T + \Delta V_1 B$$
$$E_2=\Delta U_2 T + \Delta V_2 B$$

Ez alapján pedig:

$$(E_{1x},E_{1y},E_{1z}) = \Delta U_1(Tx,Ty,Tz) + \Delta V_1(Bx,By,Bz),$$
$$(E_{2x},E_{2y},E_{2z}) = \Delta U_2(Tx,Ty,Tz) + \Delta V_2(Bx,By,Bz)$$

Az E értékét két vektorpozíció különbségéből számíthatjuk és ΔU és ΔV pedig a textúrakoordináták különbsége. Tehát két ismeretlenünk (T és B) van és két egyenletünk. A feladat tehát megoldható. A problémát felírhatjuk másképpen, mátrixszorzás formájában:

$$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

Az egyenletek ilyen formában való felírása a T és B vektorok megoldását nagyban megkönnyíti. Ha mindkét oldalt megszorozzuk a ΔUΔV mátrix inverzével, akkor a következőhöz jutunk:

$$\begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

Ez lehetővé teszi az egyenletrendszer megoldását T és B-re. Ehhez a textúrakoordináta mátrixinverzének számítása szükséges, a következőképpen: A

végeredményül kapott egyenletrendszer már megadja a T és B vektorok kiszámítását.

A következő mintapélda a gyakorlatban mutatja be a T és B vektorok számítását egy háromszög esetében:

```
// Edges of the triangle : position delta
vec3 deltaPos1 = v1 - v0;
vec3 deltaPos2 = v2 - v0;


// UV delta
vec2 deltaUV1 = uv1 - uv0;
vec2 deltaUV2 = uv2 - uv0;
// tangens és bitangens számítása
float r = 1.0f / (deltaUV1.x * deltaUV2.y - deltaUV1.y *
deltaUV2.x);
vec3 tangent = (deltaPos1 * deltaUV2.y - deltaPos2 * deltaUV1.y) *
r;
vec3 bitangent = (deltaPos2 * deltaUV1.x - deltaPos1 * deltaUV2.x) *
r;
// Normalize results
normalize(tangent);
normalize(bitangent);
```

Mivel egy háromszög mindig sík alakzat, így egy tangens/bitanges párra van szükségünk, azaz háromszög minden vertex-éhez egy a pár fog tartozni. A továbbiakban egy viszonylag komplexebb minta modellen, egy fej modellen [1] mutatjuk be a megvalósított megvilágítási modell eredményét. A megjelenítés motorja C++ nyelven került implementálva, amely az OpenGL/GLSL-t használja a GPU programozáshoz. A jobb demonstrációs eredmények érdekében két fényforrást alkalmazunk.

9. ábra. Fejrenderelés faceszintű normal/tangent/bitangent attribútumokkal. Az eredmény a flat shadinghez hasonló.

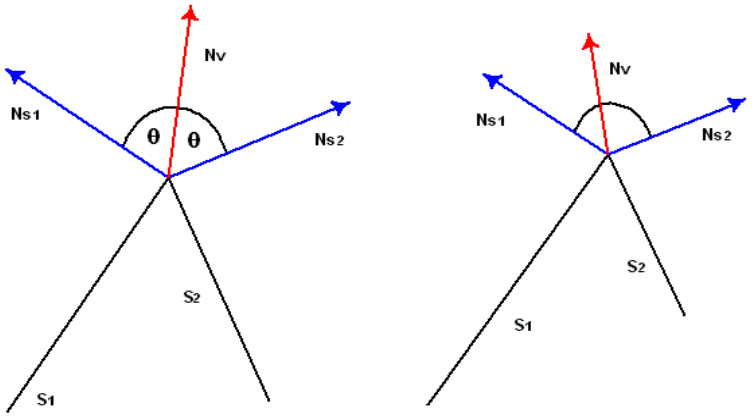
Jól látható, hogy a képen látható eredmény nem felel meg az elvárásoknak, hasonló a flat shadinghez. A modellen számos hiba is felismerhető, melyek azért vannak, mert ennél az esetnél (szándékosan) faceszintű vertex attribútumokat (normálvektor, tangent, bitangent) alkalmaztunk a háromszögek esetében, hogy mutassuk a problémát. Egy ilyen komplexebb modell esetén, amikor a legtöbb megjelenítendő felület görbe, a faceszint normális nem képes kielégítő eredményt adni.

5. Fénykorrekció

5.1. Vertex attribútumok simítása

A 9. ábra alapján bemutatott renderelés eredménye azért nem megfelelő, mert a modellt alkotó egymás melletti háromszögek normálvektorai irányban „távol" vannak egymástól, ezért a fény számítása során szembetűnő fénytörés keletkezik. Ahhoz, hogy a megvilágítás eredménye kielégítő legyen, át kell térnünk a vertexszintű normalvektorok alkalmazására. A vertexszintű normálisok azt jelentik, hogy vertexenként definiálhatjuk a normálvektorok értékeit, melyek a háromszög belsejében értelmezett barycentrikus koordináta-rendszernek köszönhetően interpolálhatók a háromszögön belül a raszterizáció során.

A fény különböző irányultságú felületek közötti töréséből fakadó „hiba" csökkentésére az egyik hatékony megoldás a közös vertexen osztozó háromszögek normálisainak simítása. Ezt a módszert a következő ábra szemlélteti, amely két felületet (S1 és S2) mutat felülnézetből, élben látva. Az S1 és S2 felületek normálvektorai kékkel vannak jelölve. A csúcspont normálvektora pirossal van ábrázolva. A csúcspont normálvektorának az S1 felület normáljával bezárt szöge megegyezik a csúcspont normálvektorának az S2 felület normáljával bezárt szögével. Amikor ezt a két felületet megvilágítjuk és Gouraud-árnyalással rendereljük, az eredmény egy simán árnyalt, lekerekített él a két felület között.
A következő illusztráció két felületet (S1 és S2), azok normálvektorait, valamint a csúcspont normálvektorát mutatja.



10. ábra. A piros nyíl a két felületen (s1, s2) osztozó vertex normálvektorát mutatja

Ha a csúcspont normál az egyik hozzá tartozó felület irányába dől, az a felületen lévő pontok fényintenzitását növeli vagy csökkenti attól függően, hogy mekkora szöget zár be a fényforrással (10. ábra, jobb oldali kép). A csúcspont normális az S1 felület felé dől, így kisebb szöget zár be a fényforrással, mintha a csúcspont normál egyenlő szöget zárna be a felületi normálokkal.

Bár jelen esetben kimondottan a normálvektorokra fókuszáltunk, nem szabad azonban elfeledkeznünk arról, hogy jelen példában a megvilágítási modell mellett normal mapping is alkalmazásra került. A tangens térhez felhasznált *tangens* és *bitangens* vektorok is szintén vertex attribútumként jelennek meg, tehát a simítást a tangens tér vektoraira is egyaránt alkalmazni kell.

11. ábra. A vertex attribútumok simítása bár sokat javít,
de további problémák/törések fedezhetők fel az eredményen

5.2. Modellszintű simítás

A renderelésnél tapasztalt hiba, a különböző törésvonalak megjelenése a modell különböző területein jól mutatja, hogy még további kiegészítésekre van szükség. A probléma abból fakad, hogy a modell nem egy vertex halmazként lett elkészítve, egy egészeként, hanem több különböző részek külön objektumként kerültek megalkotásra.
A gyakorlatban számos oka lehet annak, hogy egy komplexebb modellt így valósítanak meg. Lehet valamilyen valós logikai rendezőelv szerinti felbontás, vagy csupán egyszerűen könnyebb volt így elkészíteni a végső alakzatot.

A korábban bemutatott vertex attribútum smoothing megközelítés bár jól működik, olyan algoritmust kell létrehozni, amely az attribútumok simítása során képes figyelembe venni a különböző objektumokat is. Az algoritmust ki kell terjeszteni az egymással határos objektumokra is, melyek közös vertexeken osztoznak.
Az alábbi mintakód egy ilyen megközelítést mutat be:

```
void ComputeAndSmoothVertexAttributes(t3DModel model) {
     CVector3 vVector1, vVector2, vNormal, vPoly[3];
     pModel.numOfAllVertex = 0;

    for (index = 0; index < model.numOfObjects; index++) {
       t3DObject pObject = model.pObject[index];
       model.numOfAllVertex += pObject.numOfFaces*3;

       CVector3 pTempNormals = new CVector3 [pObject.numOfFaces];
       pObject.pNormals = new CVector3 [pObject.numOfVertices];
       pObject.pTangents = new CVector3 [pObject.numOfVertices];
       pObject.pBitangents  = new CVector3 [pObject.numOfVertices];

       for (i = 0; i < pObject.numOfFaces; i++) {
           vPoly[0] = pObject.pVerts[pObject.pFaces[i].vertIndex[0]];
           vPoly[1] = pObject.pVerts[pObject.pFaces[i].vertIndex[1]];
           vPoly[2] = pObject.pVerts[pObject.pFaces[i].vertIndex[2]];

            // Calculate the face normals
            vVector1 = Vector(vPoly[0], vPoly[2]);
            vVector2 = Vector(vPoly[2], vPoly[1]);
            vNormal = Cross(vVector1, vVector2);
            vNormal = Normalize(vNormal);

            pTempNormals[i] = vNormal;
       }

       CVector3 vSum(0.0, 0.0, 0.0);
       CVector3 vTagentSum(0.0, 0.0, 0.0);
       CVector3 vBiTagentSum(0.0, 0.0, 0.0);
       CVector3 vZero = vSum;

       int shared = 0;

       for (i = 0; i < pObject.numOfVertices; i++) {

           CVector3 vvertex = pObject.pVerts[i];

           for (subindex = 0; subindex < model.numOfObjects; subindex++) {
               t3DObject pModelObject = model.pObject[subindex];

               for (int j = 0; j < pModelObject.numOfFaces; j++) {
                   CVector3 v1 = pModelObject.pVerts[pModelObject.pFaces[j].vertIndex[0]];
                   CVector3 v2 = pModelObject.pVerts[pModelObject.pFaces[j].vertIndex[1]];
                   CVector3 v3 = pModelObject.pVerts[pModelObject.pFaces[j].vertIndex[2]];

                   if (vvertex == v1 || vvertex == v2 || vvertex == v3) {
                     vSum = AddVector(vSum, pModelObject.pFaces[j].normal);
                     vTagentSum = AddVector(vTagentSum, pModelObject.pFaces[j].tangent);
                    vBiTagentSum = AddVector(vBiTagentSum, pModelObject.pFaces[j].bitangent);
                     shared++;
                    }
               }
```

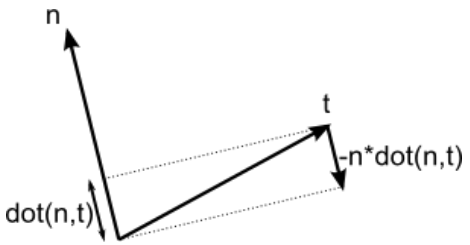
```
            }

            // Get the normal by dividing the sum by the shared
            // Negate the shared so it has the normals pointing out
            pObject.pNormals[i] = DivideVectorByScaler(vSum, shared);
            pObject.pNormals[i] = Normalize(pObject.pNormals[i]);
            pObject.pTangents[i] = DivideVectorByScaler(vTagentSum, shared);
            pObject.pTangents[i] = Normalize(pObject.pTangents[i]);
            pObject.pBitangents[i] = DivideVectorByScaler(vBiTagentSum, shared);
            pObject.pBitangents[i] = Normalize(pObject.pBitangents[i]);

            vSum = vZero;                                   // Reset the sum
            vTagentSum = vZero;                             // Reset the tangent sum
            vBiTagentSum = vZero;                           // Reset the bitangent sum
            shared = 0;                                     // Reset the shared
        }
    }
```

12. ábra. Multi-object modell, objektumok közötti vertex attribútumok simításval

5.3. Ortogonalitás

Szükséges megemlíteni egy utolsó kiegészítést a bemutatott megoldáshoz, ami egy pici többletteljesítmény-igénnyel tovább növelheti a képminőséget. Amikor nagyobb alakzatokkal dolgozunk, egy vertex számos face része is lehet. Azt már tudjuk, hogy amennyiben nem simítjuk, átlagoljuk a tangens vektorokat, úgy az eredmény gyakran nem kielégítő. A simításnak azonban egy olyan mellékhatása lehet, hogy az új TBN-vektorok az átlagolás során elveszthetik a merőlegességüket egymásra, így a képzett TBN-mátrix már nem lesz ortogonális. Bár a vizuális eredményben ez csak kevésbé észrevehető, célszerű azonban ezt az apróságot is figyelembe venni.

A probléma orvosolására a *Gram–Schmidt*-féle matematikai eljárást [14] alkalmazhatjuk, amely segítségével (újra)ortogonalizálhatjuk a TBN-vektorokat, hogy azok merőlegesek legyenek egymásra. Ezt szintén két úton tehetjük meg: a TBN-vektorok számítása, átlagolása után, vagy a vertex árnyalóban. A Gram–Schmidt-féle megoldás a következő:



13. ábra. Gram–Schmidt-féle ortogonalizálási eljárás geometriai szemléltetése

A példában *n* és *t* vektorok majdnem merőlegesek egymásra. Az ortogonalizáláshoz annyit kell tenni, hogy a *t* vektort a *–n* irányba „toljuk" el *n* és *t* skaláris szorzatának mértékével (*dot(n,t)*). Azaz:

```
t = normalize(t - n * dot(n, t));
```

Vertex árnyalóban elvégezve pedig:

```
vec3 T = normalize(vec3( modelMatrix * vec4(tangent, 0.0)));
vec3 N = normalize(vec3(modelMatrix * vec4(normal, 0.0)));
T = normalize(T - dot(T, N) * N);    // Újra-ortogonalizálás vec3 B = cross(T,
N); // Bitangens vektor számításamat3 TBN = mat3(T, B, N)
```

5.4. Koordináta-rendszer sodrása

Szimmetrikus modellek esetén előfordulhat az az eset, amikor az UV-koordináták orientációjának iránya helytelen és a tangens vektor orientációja is rossz lesz. Ennek ellenőrzése nagyon egyszerű. A TBN-vektoroknak egy jobbsodrású koordináta-rendszert kell meghatároznia. Pl. *n* és *t* vektorok vektoriális szorzatának *b*-vel egyező orientációt kell eredményeznie. Ennek matematikai ellenőrzéséhez a skaláris szorzat használható fel, miszerint A és B vektorok akkor rendelkeznek azonos orientációval, ha skaláris szorzatuk nagyobb mint nulla. Tehát dot(A,b) > 0. A TBN-vektorok esetében tehát a *dot(cross(n,t), b)* eredményét kell ellenőrizni:

```
if (dot(cross(n, t), b) < 0.0f){
     t = t * -1.0f;
 }
```

Az ellenőrzést minden vertexre végrehajtva korrigálhatjuk az esetleges rossz UV-orientációkból fakadó hibákat.

6. Összefoglalás

A fények valós idejű modellezése fontos szerepet tölt be a számítógépes vizualizációban. A hardverek rohamos fejlődése az évek során lehetővé tette, hogy számos különböző megoldás szülessen. Míg a korai években csak nagyon korlátozott megvilágítás modellek alkalmazása volt lehetséges, ma a képminőséget akár különböző vizuális effektekkel tovább növelhetjük, egy részletgazdagabb környezetet teremthetünk. Jelen cikk a multi-object modellek per-pixel megvilágításával foglalkozott.

A felületek minőségének élethűbbé tételéhez pedig normal mappingot alkalmaztunk. A végső eredmény elérése több technika együttes alkalmazását igényelte. Fontos szempont, hogy a multi-object modellek esetén a különböző objektumokat mindig egyben vizsgáljuk, a vertex attribútumokat ezeknek megfelelően számoljuk és transzformáljuk. Az objektumok határainál jelentkező vizuális problémák így kiküszöbölhetők. A bemutatott eljárások már komplex modellek esetén is egyaránt alkalmazhatók.

Irodalomjegyzék

[1] 3D Head model. (2024). https://www.zbrushcentral.com

[2] Jason Gregory (2018). Game Engine Architecture. 3rd ed. A K Peters/CRC Press.

[3] Tomas Akenine-Moller, Eric Haines, Naty Hoffman (2018). Real-Time Rendering. 4th ed. A K Peters/CRC Press.

[4] Normal Mapping. (2024). https://learnopengl.com/Advanced-Lighting/Normal-Mapping.

[5] Matt Pharr, Wenzel Jakob, Greg Humphreys (2023). Physically Based Rendering: From Theory to Implementation. 4th ed. The MIT Press.

[6] Wolfgang Engel (2009). ShaderX7: Advanced Rendering Techniques. 1st ed. Charles River Media.

[7] Kyle Halladay(2019). Practical Shader Development: Vertex and Fragment Shaders for Game Developers. 1st ed. Apress.

[8] David Wolff (2018). OpenGL 4 Shading Language Cookbook – Third Edition: Build high-quality, real-time 3D graphics with OpenGL 4.6, GLSL 4.6 and C++17. 3rd ed. Packt Publishing.

[9] Péter Mileff, Judit Dudra (2022). The Past and the Future of Computer Visualization. *Production Systems and Information Engineering*, *Volume* 10, No 1, 16–29.

[10] Eric Lengyel (2019). Foundations of Game Engine Development. *Volume 2: Rendering*, Terathon Software LLC.

[11] Frank Luna (2016). *Introduction to 3D Game Programming with DirectX 12*. Illustrated edition, Mercury Learning and Information.

[12] Miroslav Dimitrijević, Jelena Letić, Ratko Obradovic (2012). Light and shadow in 3D modeling, *Machine Design*, Vol. 5, No. 3, 1821–1259.

[13] Thorn, A. (2020). 3D Lighting and Materials. In: Moving from Unity to Godot. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-5908-5_5

[14] Stephen Andrilli, David Hecker (2010). Elementary Linear Algebra. Fourth edition. https://doi.org/10.1016/B978-0-12-374751-8.00011-1

[15] Tom McReynolds, David Blythe (2005). *Advanced Graphics Programming Using OpenGL*. Morgan Kaufmann.