IX. Erdélyi Tudományos Diákköri Konferencia

Valós idejű, egységesített légkörmegjelenítés

Szerző:

Áfra Attila Tamás
Babeş-Bolyai Tudományegyetem, Kolozsvár
Matematika és Informatika Kar
Informatika szak
2. évfolyam

Témavezető:

dr. Darvay Zsolt, adjunktus
Babeş-Bolyai Tudományegyetem, Kolozsvár
Matematika és Informatika Kar

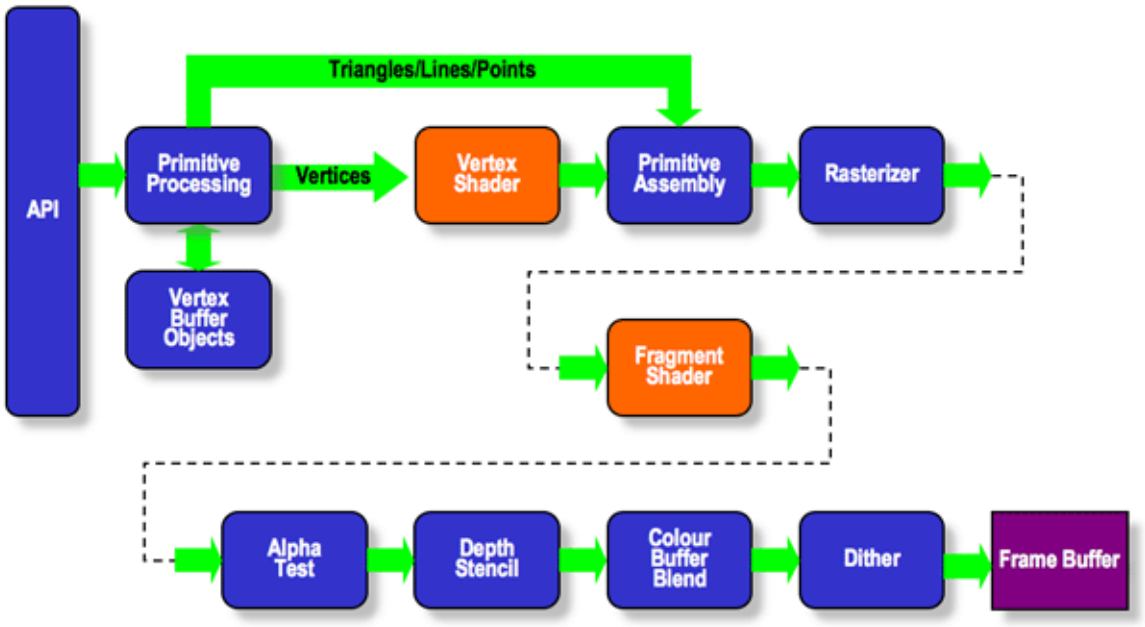
Kolozsvár, 2006. november 25-26.

1. Bevezetés

1.1. Valós idejű 3D-s grafika

Napjainkban a valós idejű számítógépes térbeli megjelenítés mindennapossá és elengedhetetlenné vált. Számos CAD rendszer, játékprogram, interaktív oktatószoftver, és szimulátor kivitelezhetetlen lenne nélküle. A valós idejű megjelenítés elsősorban gyors kell legyen. A minőség, élethűség csak másodlagos fontosságú, mivel a jelenlegi számítógépek teljesítménye nem elég nagy ahhoz hogy a legprecízebb, legkomplexebb algoritmusokat lehessen alkalmazni valós időben. Éppen ezért csak kevésbé szamításigényes módszereket lehet használni, melyeknek optimizálása fontosabb szerepet játszik mint a nem valós idejű megjelenítés esetében.

Az optimális teljesítmény érdekében a grafikai számításokat nem a központi processzoron, a CPU-n (Central Processing Unit) kell lefuttatni, hanem a pontosan erre a célra kifejlesztett 3D-s gyorsítóchipen, a GPU-n (Graphics Processing Unit), ami akár több százszor gyorsabban tudja elvégezni ezeket. Ez annak köszönhető, hogy míg a CPU-k nagy része valójában magonként csak egyetlen szálon tudja végezni a számításokat, addig a GPU-k akár 128-an. A sebességnek viszont ára van: a GPU programozhatósaga messze nem olyan flexibilis mint a CPU-nak.



***1.1. ábra**: OpenGL programozható csővezeték ([2])*

A GPU csővezeték arhitektúrával rendelkezik (1.1. ábra), amely napjainkban már nagy mértékben programozható, de még vannak bizonyos limitációk, amelyek a jövőben valószínűleg szinte teljesen el fognak tűnni.

1.2. Légkörmegjelenítés

A 3D-s grafika segítségével megjeleníthető valós vagy fiktív helyszínek két típusúak lehetnek: belső terek és külső terek. A kettő közül egyértelműen a külső terek megjelenítése a komplexebb, mivel a látótávolság nem pár méter hanem akár több száz kilométer. Ezen kívül van még egy másik nagyon nehezen megoldható probléma: a légköri jelenségek megjelenítése.

A két legfontosabb megjelenítendő dolog: az égbolt és a légperspektíva. A légkörnek köszönhető, hogy nappal az ég nem fekete, nem látszanak a csillagok és az is, hogy a távolban levő tárgyak színe halványabb, az ég színe fele tolódik, a szemlélőtől mért távolságtól függően. Éppen ezért, ez az utóbbi jelenség, a légperspektíva, nagyon fontos szerepet játszik a tárgyak és domborzati formák távolságának becslésében.

Ezen jelenségek megjelenítése komplex számításokat igényel. Az eddig megjelent valós idejű alkalmázasok szinte mindegyike nagyon nagy mértékben leegyszerűsítette a légkörmegjelenítést, elsősorban azért mert csak még egy pár éve, 2001-ben, jelent meg az első programozható csővezetékű GPU, ami nélkül nem lehet megvalósítani ezeknek a számításoknak a hardveres gyorsítását.

(a) Az égbolt naplemente után

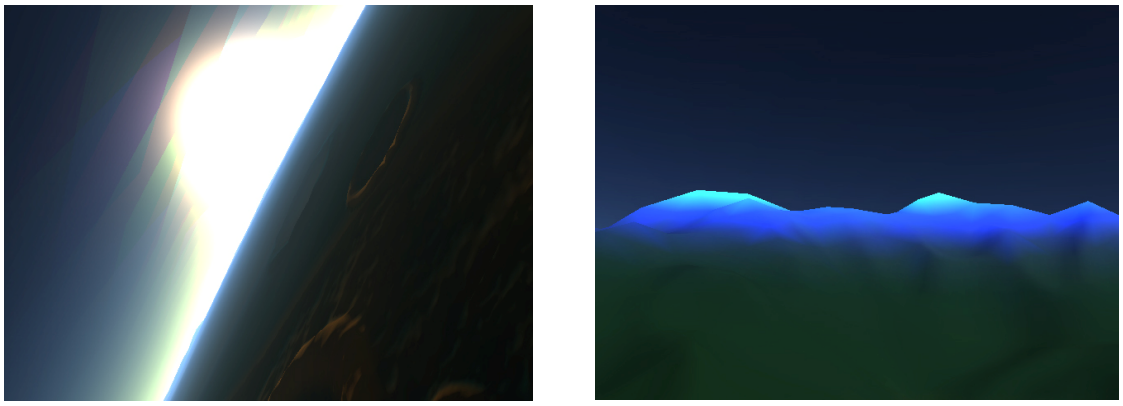
(b) Légperspektíva

***1.2. ábra**: Légköri jelenségek*

Az egyetlen légköri jelenség amely hardveresen implementálva van a nem programozható GPU-kban, az a köd, de a használt algoritmus nem egy fizikai modellre épül, így a végeredmény, bár hasonlít a valódi ködre, nem elég élethű. Ez különösen a szimulátorprogramoknál okoz gondot (pl. repülőgép szimulátor), mivel abban az esetben a légperspektíva igen fontos szerepet játszik.

A mai GPU-k nagyon sokat fejlődtek az utóbbi pár évben és így lehetővé vált fizikailag helyesebb, komplexebb légköri modellek implementációja. Ennek ellenére a programok többsége még mindig a régi ködmodellt alkalmazza és az égbolt megjelenítéséhez előre renderelt, rendkívül élethű képeket vagy akár fényképeket használ. Ez elsősorban azért van így, mert a jelenlegi fizikai modell alapú valós idejű légkörmegjelentítési algoritmusoknak több súlyos hibájuk van és nehézkesen implementálhatóak, különösen bonyolultabb programok esetében.

A dolgozat célja egy olyan új algoritmus kidolgozása, amely mentes az imént említett problámáktól és rendelkezik még egy nagyon fontos tulajdonsággal: az egységesített megjelenítéssel. Ez abban áll, hogy az összes fő légköri jelenséget egyetlen algoritmussal lehet megjelentíteni, akár a légkörön belülről, akár az űrből szemlélve. Ennek köszönhetően sokkal egyszerűbb az implementáció, könnyebben bővíthető és a végeredmény fotorealisztikusabb. Ezt a tradicionális, nem valós idejű légkörmegjelenítési algoritmusok [6] és a *deferred shading* módszer [1] ötvözésével és megfelelő mértékű optimizációval lehet elérni.



(a) Hibás Mie szóródás (b) Hibás légperspektíva

***1.3. ábra**: Hibák az eddigi valós idejű légkörmegjelenítési algoritmusokban ([7])*

2. A Föld légköre

2.1. Bevezetés

Az atmoszférikus modell alapját a fény szóródása képezi. A légkörön keresztül haladó fény (Nap, Hold, csillagok, stb...) a szemünkbe nem direkt módon jut el, hanem a légkörnek köszönhetően számtalanszor szóródik, elnyelődik, megtörik, visszaverődik, spektruma torzul. E miatt nappal az égbolt nem fekete, mivel a légkör tulajdonképpen másodlagos fényforrásként viselkedik.

A fény szóródását a légkörben a levegőmolekulák és az aeroszolok okozzák. E jelenség során a fény csak intenzitásában változik, frekvenciája és hullámhossza megmarad. A szórás mértékét elsősorban a részecskék mérete és a fény hullámhossza befolyásolja.

Egy másik fontos tényező az atmoszféra megjelenítésénél a légkör vastagsága, illetve sűrűsége, amely a tengerszint feletti magasságtól függ. A sűrűség nagy mértékben befolyásolja a végeredményt, mivel minél nagyobb a sűrűség, annál nagyobb a fény szóródásának mértéke. A sűrűség exponenciálisan változik a magasságtól függően, amit a következő képlettel lehet kifejezni:

$$\rho = e^{-\frac{h}{H_0}}, \qquad (2.1)$$

ahol ρ a relatív sűrűség, h a magasság és H_0 egy a légkörre jellemző konstans (a lévegőmolekulák esetében 8 km, az aeroszolok esetében 1.2-3 km)

Mivel a levegőmolekuláknak és az aeroszoloknak különbözik a méretük és sűrűségük azonos magasságban, ezért külön kell kiszámolni az általuk előidézett szóródást.

2.2. Rayleigh szóródás

A levegőmolekulák (melyeknek a mérete jóval kisebb mint a látható fény hullámhossza) által előidézett szóródást Rayleigh szóródásnak nevezzük. Az égbolt kék (ill. napfeltekor és

naplementekor vöröses, narancssárgás) színe ennek a jelenségnek köszönhető. Ez a miatt van, hogy a Rayleigh szórás mértéke egyenes arányos $\frac{1}{\lambda^4}$-el, ahol λ a fény hullámhossza. Egy adott irányba való szóródás mértékét a következő függvénnyel lehet kiszámolni [9]:

$$\beta_R(\theta,\lambda)=\frac{\pi^2\left(n^2-1\right)^2}{2N\lambda^4}\left(\frac{6+3p_n}{6-7p_n}\right)\left(1+\cos^2\theta\right), \qquad (2.2)$$

ahol θ a szóródás szöge, λ a szórt fény hullámhossza, n a levegő törésmutatója (1,0003), N az egységnyi térfogatban lévő molekulák száma ($2{,}545\cdot10^{25}$) és p_n a depolarizálási faktor (0,035).

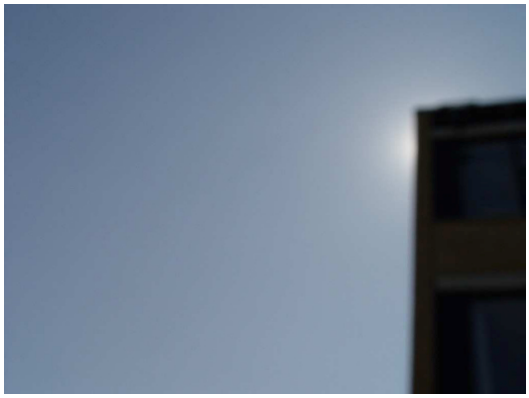
A totális szóródási (total scattering) együtthatót, melynek segítségével ki lehet számolni, hogy egy légkörön áthaladó fénysugár mennyit veszített az intenzitásából, a következő képlet adja meg:

$$\beta_R(\lambda)=\frac{8\pi^3\left(n^2-1\right)^2}{3N\lambda^4}\left(\frac{6+3p_n}{6-7p_n}\right) \qquad (2.3)$$

2.3. Mie szóródás

A Mie szóródási élmélet általánosan alkalmazható bármilyen méretű részecskék esetében, így tulajdonképpen a Rayleigh szóródás a Mie szóródásnak egy speciális esete. Az oka annak, hogy a levegőmolekulák esetében a Rayleigh szóródási képleteket használjuk az, hogy a Mie szóródás kiszámolása jóval bonyolultabb és időigényesebb.

Az aeroszolok mérete megközelítőleg egyenlő vagy nagyobb mint a szórt fény hullámhossza, ezért a Mie elméletet szükséges alkalmazni. A Mie szóródásnak köszönhető például a köd és a Nap körül megfigyelhető világosabb zóna (2.1. ábra).



(a) Világos zóna a Nap körül

(b) Köd

***2.1. ábra**: Mie szóródás által okozott légköri jelenségek*

Egy adott irányba való szóródást a következő képpen lehet kifejezni:

$$\beta_M(\theta,\lambda) = 0{,}434c\frac{4\pi^2}{\lambda^2}0.5F_M(\theta,g), \tag{2.4}$$

ahol c a koncentráció, ami függ a turbiditástól (T):

$$c = (0{,}6544T - 0{,}6510)\cdot 10^{-16} \tag{2.5}$$

és $F_M(\theta,g)$ a Cornette által módosított Henyey-Greenstein függvény:

$$F_M(\theta,g) = \frac{3(1-g^2)}{2(2+g^2)}\frac{(1+\cos^2\theta)}{(1+g^2-2g\cos\theta)^{3/2}}, \tag{2.6}$$

ahol g a szórás aszimmetrikussága, amit következő képlet ad meg:

$$\begin{aligned} g &= \frac{5}{9}u - \left(\frac{4}{3} - \frac{25}{81}u^2\right)x^{-1/3} + x^{1/3}, \\ x &= \frac{5}{9}u + \frac{125}{729}u^3 + \left(\frac{64}{27} - \frac{325}{243}u^2 + \frac{1250}{2187}u^4\right)^{1/2}, \end{aligned} \tag{2.7}$$

ahol ha g = 0, akkor a függvény egyenértékű a Rayleigh szórással. Az u függ az atmoszférikus körülményektől és a hullámhossztól és 0,7 és 0,8 között mozog.

A totális szóródást az alábbi képlet adja meg:

$$\beta_M(\lambda) = 0{,}434c\pi\frac{4\pi^2}{\lambda^2}K, \qquad (2.8)$$

ahol K függ a hullámhossztól és megközelítőleg 0,67.

A képleteket megvizsgálva észrevehetjük, hogy a Mie szórás sokkal kevésbé függ a hullámhossztól mint a Rayleigh szórás. Ennek köszönhető például az, hogy a köd nem színes.

2.4. Optikai mélység

Ha egy fénysugár áthalad az atmoszférán, akkor veszít az intenzitásából, mivel szóródik és részben elnyelődik. Ezt a kombinált hatást nevezzük extinkciónak. A Föld légkörének esetében az elnyelődés oly csekély mértékű, hogy elhanyagolható.

Az extinkciót a következő képlet adja meg:

$$I = I_0 \cdot e^{-t(s,\lambda)}, \qquad (2.9)$$

ahol I a végleges intenzitás, I_0 az eredeti intenzitás és $t(s,\lambda)$ az optikai mélység.

Az optikai mélység az átlátszóság mértékét fejezi ki egy adott szakasz mentén a légkörben és integrálás segítségével lehet kiszámolni:

$$t(s,\lambda) = \int_s \beta(s,\lambda)\rho(s)ds = \beta(s,\lambda)\cdot\int_s \rho(s)ds, \qquad (2.10)$$

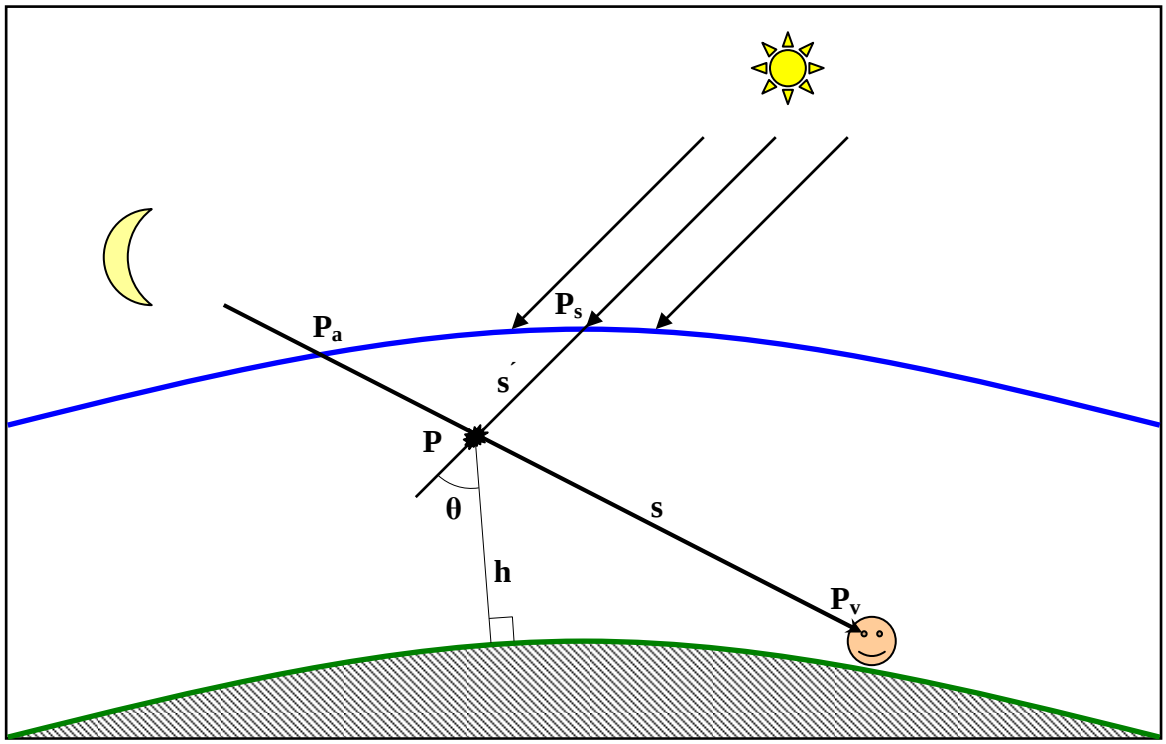
ahol s az illető szakasz.

***2.2. ábra**: Extinkció ([4])*

Mivel nem csak egyetlen szóródási együtthatóval dolgozunk, ezért az optikai mélység számolásakor össze kell adni ezeket, hogy a megfelelő eredményhez juthassunk.

2.5. Egyszeres (elsődleges) szóródási modell

Egy fénysugár számtalanszor szóródik amíg a szemünkbe ér, de mivel egy olyan algoritmusnak a kidolgozása a cél, amely valós időben is alkalmazható, ezért nincs lehetőség többszörös szóródás szimulálására, mivel az nagyon időigényes és hatalmas mennyiségű memóriára van szükség ([5]). Szerencsére a légkör esetében az elsődleges szóródás kiszámolása elegendő a megfelelő minőség eléréséhez, mert már a másodlagos szóródás hatása is alig észlelhető.



***2.3. ábra**: Egyszeres szóródási modell*

Az egyszerűség kedvéért a Napból érkező fénysugarakat tekintsük egymással párhuzamosaknak, mivel a Föld és a Nap közötti távolság hatalmas. Ez nagy mértékben leegyszerűsíti a modellt. Egy adott hullámhosszon a végső intenzitást a következő képpen kaphatjuk meg:

$$
\begin{aligned}
&I_v(\lambda) = I_a(\lambda)\cdot e^{-t(P_v P_a,\lambda)} + \int_{P_v}^{P_a} I_s(\lambda)\cdot R(\theta,s,\lambda)\cdot e^{-t(s,\lambda)-t(s',\lambda)} ds,\\
&R(\theta,s,\lambda) = \beta_R(\theta,\lambda)\rho_R(s) + \beta_M(\theta,\lambda)\rho_M(s),\\
&t(s,\lambda) = \beta_R(\lambda)\cdot\int_s \rho_R(l)dl + \beta_M(\lambda)\cdot\int_s \rho_M(l)dl,
\end{aligned}
\qquad (2.11)
$$

ahol I_v az intenzitás a P_v pontban, I_a az intenzitás a P_a pontban és I_s az intenzitás a P_s pontban.

A P_v a kamerának a poziciójában kell legyen, hogy ha az a légkörön belül helyezkedik el. Ellenkező esetben a légkör határán kell lennie. A P_a a kamerához képest legközelebbi, légkörön belüli, nem átlátszó pont egy adott irányban, amely fényt bocsát ki vagy ver vissza. Ha nincs ilyen pont, akkor a P_a a légkör határán helyezkedik el. A P_s az a pont, ahol a Napból érkező

fénysugarak belépnek a légkörbe, így az mindig a légkör határán helyezkedik el. Mivel a valóságban nem lehet pontosan meghatározni, hogy milyen magasságban van a légkör határa, ezért egy olyan magasságot kell választani, ahol a légsűrűség elhanyagolhatóan kicsi.

2.6. Optimizáció a GPU-ra

A szóródási számításokat az optimális teljesítmény érdekében egy *pixel (fragment) shader*, vagyis egy GPU által lefuttatandó program keretein belül kell implementálni. Ugyanakkor a képletek módosítás nélkül való beépítése nem fog vezetni elfogadható sebességhez, mivel a leegyszerűsített modell is bonyolult az átlagos valós idejű grafikai modellekhez képest.

Ideális esetben a szóródást a teljes fényspektrumon kell kiszámolni, mivel az nagy mértékben függ a hullámhossztól. Valós időben viszont egyelőre erre nincs lehetőség, mivel egyetlen textúrában legfeljebb négy színcsatornát lehet tárolni. Így, a legtöbb grafikai algoritmushoz hasonlóan, három hullámhosszon kell elvégezni a számításokat, a negyedik színcsatorna pedig más fajta információ tárolására fog szolgálni (*deferred shading*). Ez a három hullámhossz a grafikában nagyon gyakran használt három alapszínnek kell megfeleljen, amelyek a vörös (680 nm), zöld (570 nm) és a kék (475 nm). Ennek az egyszerűsítésnek az ellenére a minőség alig észrevehető mértékben csökken.

A legnagyobb probléma viszont a (2.11)-es képlet kiértékelése, mivel a benne levő integrálokat numerikusan kell megoldani, ami nagyon erőforrásigényes. A fő integrált minden pixel esetében ki kell numerikusan számolni, viszont az optikai mélység kiszámolását nagy mértékben optimizálni lehet.

A képletben szereplő optikai mélység tulajdonképpen két optikai mélység összege: az egyik az s' szakasz, a másik pedig az s szakasz mentén levő. Az s' két végpontja minden esetben az aktuális szórási pozició és egy a légkör határán elhelyezkedő pont. Így az optikai mélység e szakasz mentén meghatározható egy a földtől számított magasság és egy zenitszög segítségével, mivel az egyedüli változó, amitől függ egy pontban a légkör sűrűsége (és ami szerint az optikai mélységet is integráljuk), az a magasság. Jelen esetben a zenitszög a szóródási ponton és a Föld (vagy az illető égitest) középpontján áthaladó egyenes és az s' szakasz tartóegyenese által bezárt szög. Így az optikai mélység hullámhossztól független együtthatóját előre kiszámolhatjuk (a

valós idejű szimuláció kezdete előtt) és eltárolhatjuk egy kétdimenziós tömbben [8], jelen esetben egy textúrában. A megfelelő minőség érdekében a textúrának a lehető legnagyobb felbontása (jelen pillanatban ez 4096x4096) és 32 bites precizitású lebegőpontos formátuma kell legyen. Mivel két sűrűségre és így két optikai mélységre van szükség, ezért egy adott paraméterpárnak két értek kell megfeleljen. A köztes paramétereknek megfelelő értékeket (amelyek nincsenek eltárolva) lineáris interpolációval közelíthetjük meg. Mivel a jelenlegi GPU-kba nincs hardveresen beépítve a 32 bites lebegőpontos formátumú textúrák implicit lineáris interpolálása (a 8 bites egész, illetve a 16 bites lebegőpontos formátumok esetében viszont igen), ezért ezt manuálisan, a *shader* programon belül, kell implementálni. Tetszés szerint további optimizációkat is lehet alkalmazni. Például a kétdimenziós tömböt ki lehet cserélni egy egydimenziósra a nagyobb teljesítmény érdekében ([7]). Ez csak minimális mértékben fogja csökkenteni a minőséget, viszont a teljesítmény látványosan nagyobb lesz, elsősorban az egyszerűbb interpoláció miatt. Ez az optimizáció viszont szükségtelenné fog válni, amint megjelennek az első GPU-k, amelyek már hardveresen képesek az ilyet formátumú textúrák interpolálására.

Az *s* szakasz menti optikai mélységet viszonylag könnyen meg lehet kapni. Más algoritmusok a számítás során felhasználjak az előbb említett tömböt, viszont ez bizonyos esetekben súlyos vizuális hibákhoz vezet (*1.2. ábra*), mivel így a számítás nem elég pontos. Ezért más módszert kell alkalmazni. Ha a fő integrál kiszámításához szükséges mintákat a P_a pontból kiindulva a P_v pont fele vesszük, akkor kihasználhatjuk azt, hogy az optikai mélységet mindig a P_aP szakasz mentén kell megkapni és így a két integrált egyszerre, egymással párhuzamosan tudjuk számolni, hiszen a minták közösek lehetnek. Ebben az esetben a két integrált nem lehet azonos típusú Riemann összeg segítségével megkapni (pl. közép és jobb).

A harmadik optikiai mélységet nem szükséges külön kiszámolni, mivel azt az *s* szakasz menti sűrűség integrálása során már megkaptuk, amikor a szóródást a P_v pontban számoltuk ki (vagy abban a pontban, amely a legközelebb áll hozzá).

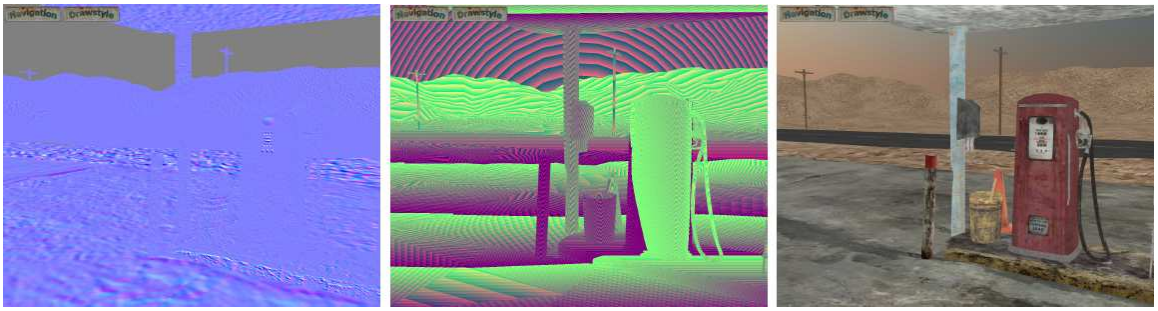
3. Deferred shading

3.1. Megvilágítás a *deferred shading* módszerrel

Egy fényforrás által megvilágított tárgy kirajzolása általában egyetlen lépésben történik. Ha több fényforrás van, akkor annyi lépésben lesz kirajzolva, ahány fényforrás világítja meg a tárgyat. Ez sok fényforrás és komplex geometria esetén súlyos teljesítménycsökkenéshez vezet, mivel rengeteg számítás többször lesz elvégezve fölöslegesen, minden egyes lépés során, mint például a geometriai számítások, textúrázás, fényvisszaverődés, fénytörés, stb....

Erre a problémára ad megoldást a *deferred shading* módszer ([1]). Ahogy a neve is mutatja, a megvilágítás („árnyalás”) késleltetve történik, pontosabban a többi számítástól szétválasztva, egy külön rajzolási fázisban. Az első fázisban ki kell rajzolni minden tárgyat megvilágítás nélkül egy ideiglenes textúrába (vagy többe, ha szükséges), és utána annyiszor kell lefuttatni a megvilágítási fázist, ahány fényforrás van. Így minden szükséges számítás csak egyszer lesz kiszámolva.

A tradicionális módszer esetében egyszerű a számolás, mivel a megvilágításhoz szükséges összes adat rendelkezésre áll. Ezek közül a legfontosabb az aktuális (a képsíkra levetített) pixel (pontosabban fragment) térbeli koordinátái, amelyekhez direkt hozzáférés van a GPU vertex(csúcs)interpoláló egységének köszönhetően. A *deferred shading* esetében viszont az egyedüli adat amihez hozzá lehet férni a második fázisban, az a textúra amibe az első fázis során rajzoltunk. Így nem csak az albedót kell eltárolni az első fázisban, hanem többek között az interpolált pozíciót is.



(a) Normális (b) Pozíció (c) Albedó

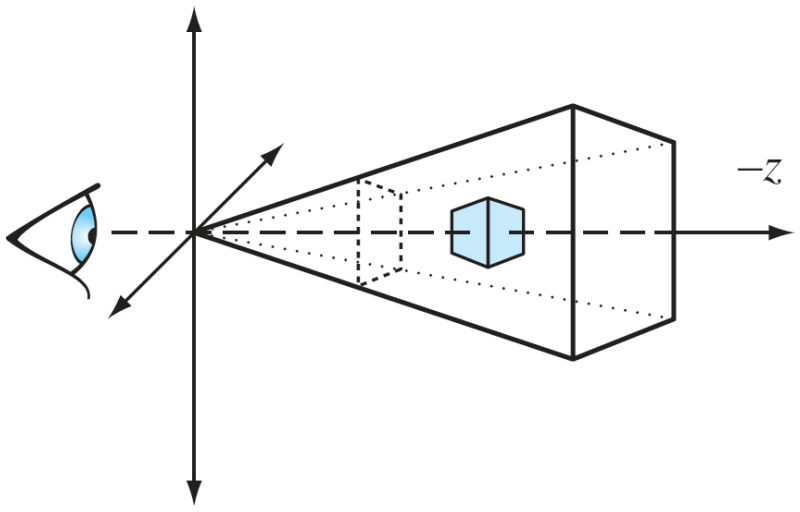
***3.1. ábra**: G-puffer ([1])*

Egy textúra maximum négy csatornával rendelkezhet: vörös (R), zöld (G), kék (B) és a speciális alfa (A) csatorna. Három szükséges az albedó tárolására és még további három, hogy a pozíciót is el lehessen tárolni. Így legalább két ideiglenes textúrára van szükség, ami lényegesen bonyolultabbá teszi az implementációt, több memóriát foglal és jelentős teljesítménycsökkenést von maga után. Az utóbbi annak köszönhető, hogy a szükséges memóriasávszélesség megkétszereződik és ami így akár a szűk keresztmetszetet is képezheti. Ezen ideiglenes textúrákat együttesen G-puffernek nevezzük.

3.2. Pozíciótömörítés

Általános esetben szükség van három koordinátára, hogy egy pontot meg lehessen határozni a térben. A *deferred shading* módszer esetében viszont nem szükséges eltárolni mind a három koordinátát, vagyis lehetőség van tömörítésre ([11]).

A 3D-s grafikában ahhoz, hogy meg lehessen jeleníteni térbeli tárgyakat, szükségünk van egy vetítési módszerre, aminek a segítségével a geometriát le lehet vetíteni a képsíkra. A két legfontosabb vetítési módszer a párhuzamos és a perspektivikus vetítés. A továbbiakban a perspektivikus vetítésről lesz szó, mivel ennek a módszernek a segítségével lehet szimulálni a valódi lencsék működését.



***3.2. ábra**: Perspektivikus vetítés ([3])*

A látómező gúla alakú, ahol a gúla csúcsának koordinátái megegyeznek a virtuális kameráéval. Ezt a gúlát két egymással párhuzamos síkkal metsszük, amelyek merőlegesek a kamera orientációvektorára (OZ tengely). Így kialakul egy csonkagúla, és csak az fog levetítődni a képsíkra, ami ennek a gúlának a belsejében van. A kamerához közelebb lévő sík a képsíkot határozza meg, a másik pedig a látótávolság korlátozására szolgál.

Egy vetítési mátrix segítségével a csonkagúlát egy téglatestté lehet transzformálni (a kisalap változatlan marad), aminek következtében a benne lévő geometria is torzulni fog. A kapott koordináták még mindig a térben lesznek, viszont ha elhagyjuk a Z-t, vagyis a mélységet, akkor az illető pontot ki lehet rajzolni a képernyőre. A transzformáció során figyelembe vettük a perspektivikus torzulást, így ezzel a módszerrel helyes eredményhez juthatunk.

Ha csak a transzformáció utáni Z koordinátát tároljuk el a *deferred shading* első fázisa során, akkor a vetítési mátrix inverzének segítségével vissza tudjuk kapni az eredeti koordinátákat. Az X-et és Y-t nem szükséges ebben az esetben eltárolni, mivel azok egy szorzás segítségével kiszámolhatóak az aktuálisan kirajzolandó pixel (ami a kisalapon helyezkedik el) pozíciójából.

Ezt viszont tovább is lehet optimizálni, kihasználva a GPU által nyújtott speciális funkciókat. Tételezzük fel, hogy a kamera az origóban van. Ekkor tudva az aktuális pixelnek megfelelő normalizált mélységet és irányvektort, ki lehet számolni az eredeti pozíciót. Ha a kamera nem az origóban helyezkedik el, akkor a kapott pozícióhoz hozzá kell adni a kamera pozícióját.

A normalizált mélységet a következő képpen lehet megkapni:

$$z' = \frac{z}{z_f}, \qquad (3.1)$$

ahol z a vetítési transzformáció utáni mélység és z_f a távoli sík mélysége.

Az irányvektor hossza egyenlő kell legyen a távoli sík és a vektor tartóegyenese által meghatározott pont és az origó közti távolsággal. Ha ismerjük a csonkagúla nagyalapja négy csúcsának megfelelő vektorokat, akkor a GPU segítségével a belső pontoknak megfelelő vektorokat megkaphatjuk lineáris interpolációt használva. Tehát az eredeti pont koordinátáit kiszámolhatjuk a következő képlet segítségével:

$$V = z' \cdot D + C\,, \qquad (3.2)$$

ahol *D* az irányvektor és *C* a kamera pozíciója.

Így az ideiglenes textúrában elegendő csak a normalizált mélységet eltárolni, és e miatt elegendő csak egyetlen textúrát alkalmazni. Ugyanakkor ez a módszer lényegesen kevésbe számításigényes mint a mátrixszorzásos, mivel egy vektor és egy mátrix összeszorzása 16 szorzást és 12 összeadást igényel, míg egy vektor skalárral való szorzása csak 4 szorzást.

A csonkagúla nagyalapja négy csúcsának megfelelő vektorokat a következő képpen lehet kiszámolni:

$$D_{tl} = P^{-1} \cdot \begin{pmatrix} -1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \quad D_{tr} = P^{-1} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \quad D_{bl} = P^{-1} \cdot \begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix}, \quad D_{br} = P^{-1} \cdot \begin{pmatrix} 1 \\ -1 \\ 1 \\ 1 \end{pmatrix}, \qquad (3.3)$$

ahol a D_{tl} a bal felső, a D_{tr} a jobb felső, a D_{bl} a bal alsó és a D_{br} a jobb alsó csúcs. A P^{-1} a perspektvikus vetítési mátrix inverze, amely az OpenGL esetében a következő:

$$P^{-1} = \begin{pmatrix} \frac{ar}{c} & 0 & 0 & 0 \\ 0 & \frac{1}{c} & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{z_n - z_f}{2 z_n z_f} & \frac{z_n + z_f}{2 z_n z_f} \end{pmatrix}, \qquad (3.4)$$

$$c = \operatorname{ctg} \frac{\alpha}{2},$$

ahol *ar* a képarány és α a függőleges látószög.

3.3. Légkörmegjelenítés a *deferred shading* módszert használva

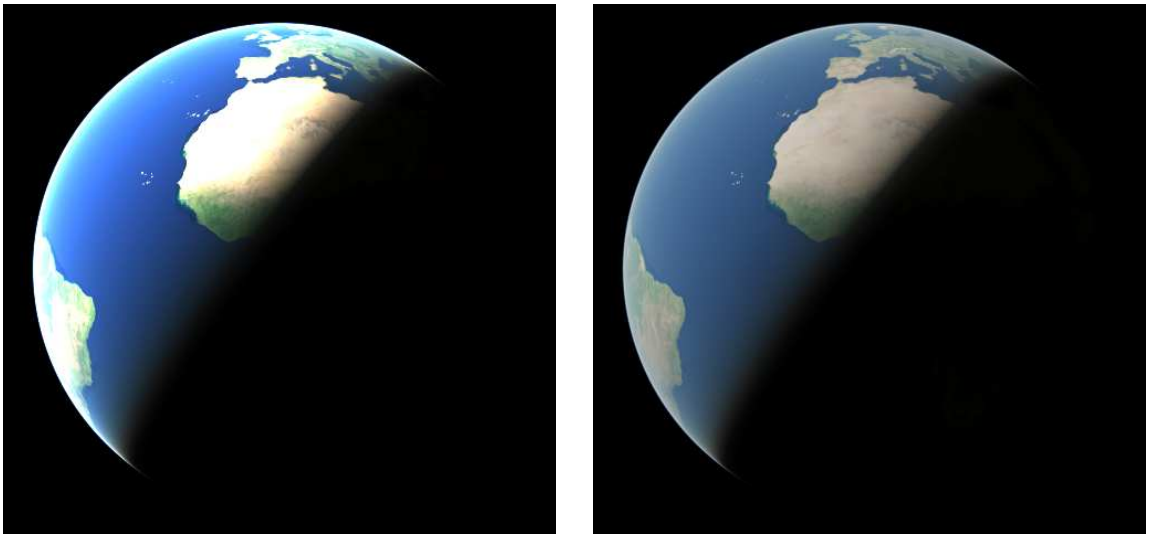
A *deferred shading* módszert megfelő módosítások után fel lehet használni a légkör megjelenítéséhez, a megvilágítás helyett. Így a kapott algoritmus örökölni fogja a *deferred shading* összes előnyét.

A módosításokat a fő shader programban kell elvégezni, amely helyettesíti a megvilágítást a légkörmodellel. Ahhoz, hogy a szórási számításokat (2.11-es képlet) el lehessen végezni, előbb meg kell kapjuk a szükséges paramétereket. Először ki kell számoljuk a P_a és P_v pontok pozicóját. Ezt úgy lehet megvalósítani, hogy metsszük az aktuális pixel irányvektorának tartóegyenesét a légkör határát képező gömbbel. Ha az egyenes nem metszi a gömböt, vagy csak egyetlen pontban metszi, akkor a eredeti pixel értéke változatlan marad. Ez után ellenőriznünk kell a már említett feltételeket, hogy a két pont közül melyek kell elhelyezkedjenek a gömb felszínén és ez után, felhasználva a két metszéspontot, ki kell számolnunk a két pont végleges pozícióját.

A képletben használt I_a intenzitás az első renderelési fázisból származó, eredeti pixelérték (albedó), az I_s a fő fényforrásra (pl. Nap) jellemző, előre megadott intenzitás, az I_v pedig a *shader* által eredményként visszatérített végső pixelérték.

3.4. Magas dinamikai tartományú (HDR) megjelenítés

Általában a számítógépek monitorai legfeljebb 8 biten tudják ábrázolni az egyes színcsatornák intenzitását. E miatt a GPU képpuffere (ahol a megjelenítendő kép van tárolva) is 8 bites precizitású az optimális memóriahasználat érdekében. 8 biten viszont a maximálisan ábrázolható fényerő-kontraszt 255:1, ami általában elég, viszont kültéri megjelenítéshez nagyon kevés, mivel abban az esetben a kontraszt lehet akár 1.000.000:1 is.



(a) *Tone mapping* operátor nélkül (b) *Tone mapping* operátorral

***3.3. ábra**: Tone mapping (képek a demo programból)*

Ezért ebben az esetben magas dinamikai tártományú megjelenítés (HDR) ajánlott, ami könnyedén beépíthető a *deferred shading* megjelenítési módszerbe. A megfelelő precizitás érdekében létrehozunk egy újabb megjelenítési fázist: a *deferred shading* módszerrel kapott képet nem rajzoljuk ki egyből a képernyőre, hanem egy lebegőpontos, legalább 16 bites precizitású textúrába, amit az új fázistban fogunk feldolgozni.

Ahhoz, hogy a megfelelő hatást elérjük, kell alkalmazzunk egy olyan operátort, amely egy magas dinamikai tartományú képet egy alacsony (8 bites) dinamikai tartományú képpé transzformál úgy, hogy a részletek ne vesszenek el és a hatalmas kontraszt érzékelhető legyen. Az ilyeneket *tone mapping* operátoroknak nevezzük és az emberi látás elveire épülnek. Több *tone mapping* operátor létezik, amelyek között vannak olyanok is, amelyek valós időben is alkalmazhatóak. Az egyik legegyszerűbb és legjobb minőséghez vezető ilyen operátor a következő ([10]):

$$L' = \frac{L}{1+L}, \tag{3.5}$$

ahol L az eredeti intenzitás és L' a végleges intenzitás.

Ez az operátor két okból is nagyon jó. Elsősorban azért, mert bármekkora lehet az erdeti fényerő, a végeredmény mindig benne lesz a $[0,1)$ intervallumban, és ezért nagyon könnyen leképezhető a 8 bites pixelformátumra jelentős minőségveszteség nélkül. Az operátor másik nagy

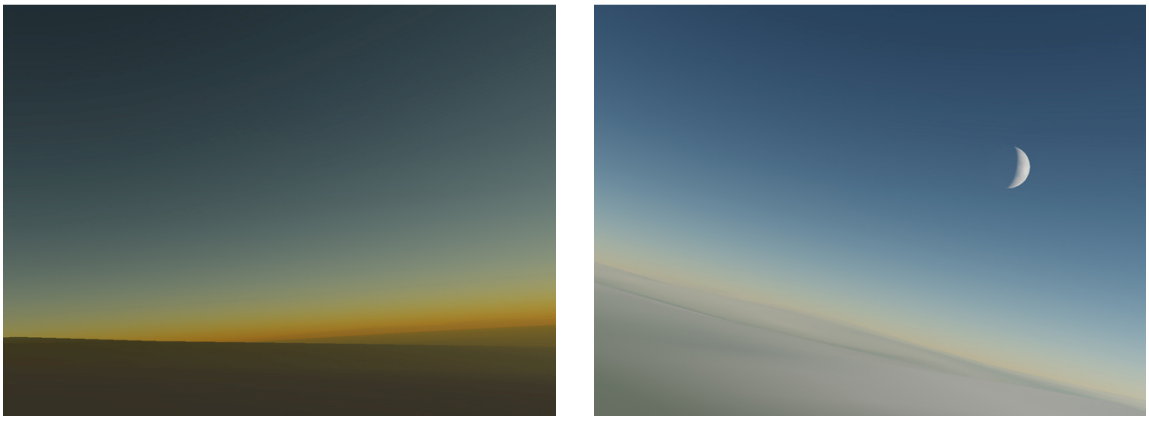
előnye a hatékonyság, mivel a megfelelő számítosokat a GPU is el tudja végezni a nélkül, hogy a megjelenítési sebesség túlságosan csökkenne.

Ugyanakkor ez a fázis tovább bővíthető egyéb optikai jelenségek szimulálásával a nagyobb élethűség érdekében (pl. adaptív exponálás, *bloom* effektus).

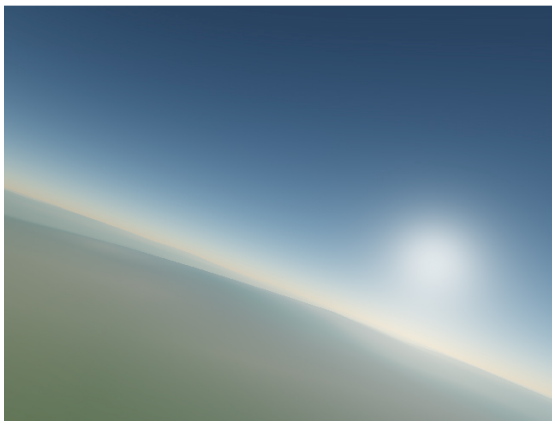
4. Implementáció

Az algoritmust C++ és OpenGL 2.0 segítségével valósítottuk meg. A *shader* programok Shader Model 2.0 és 3.0 standardok szerint lettek megírva. A beépített funkciók mellett az *EXT_framebuffer_object* és *ARB_texture_float* OpenGL extenziók voltak felhasználva. A demo program optimális futtatásához legalább NVIDIA GeForce 6000 vagy ATI Radeon X1000 családba tartozó GPU szükséges és minimum 128 MB videomemóriával kell rendelkezzen.

A program a Földet, a Holdat és a Napot jeleníti meg és a billentyűzet segítségével tetszőleges pozícióból lehet szemlélni azokat. Ugyanakkor lehetőség van a látószög növelésére illetve csökkentésére (zoom). Az egyszerűség és jobb átláthatóság érdekében a Föld felhőzete nincs megjelenítve.



***4.1. ábra**: Képek a demo programból (1)*

***4.2. ábra**: Képek a demo programból (2)*

5. Konklúzió

Egy olyan új valós idejű légkörmegjelenítési algoritmust sikerült kidolgozni, melynek segítségével fotorealisztikusan meg lehet jeleníteni a fő légköri jelenségeket, kihasználva a modern GPU-k által nyújtott lehetőségeket. Az eddigi hasonló algoritmusokkal szemben teljesen általános, könnyen bővíthető, implementálható és a végeredmény minősége megközelíti a nem valós idejű algoritmusokét. A *deferred shading* módszerre épül, így örökli az összes pozitív tulajdonságát, aminek következtében hatékonyan be lehet építeni a már meglévő modern grafikus megjelenítési rendszerekbe. Ugyanakkor lehetőség van a légköri modell kicserélésére, így akár egy a Földétől eltérő tulajdonságokkal rendelkező atmoszférát is lehet szimulálni.

További kutatások során meg lehet vizsgálni, hogy milyen optimizációkat lehet még bevezetni és hogy milyen más légkörmodelleket lehet hatékonyan implementálni.

Könyvészet

[1] Shawn Hargreaves, Mark Harris: *Deferred Shading*, 6800 Leagues Under the Sea, NVIDIA, 2004.

[2] Khronos Group: *OpenGL ES 2.X and the OpenGL ES Shading Language for programmable hardware* (http://www.khronos.org/opengles/2_X/)

[3] Tom McReynolds, David Blythe: *Advanced Graphics Programming Using OpenGL*, Morgan Kaufmann, 2005.

[4] Ralf Stokholm Nielsen: *Real Time Rendering of Atmospheric Scattering Effects for Flight Simulators*, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2003.

[5] Tomoyuki Nishita, Yoshinori Dobashi, Kazufumi Kaneda, Hideo Yamashita: *Display Method of the Sky Color Taking into Account Multiple Scattering*, Pacific Graphics 1996.

[6] Tomoyuki Nishita, Takao Sirai, Katsumi Tadamura, Eihachiro Nakamae: *Display of The Earth Taking into Account Atmospheric Scattering*, SIGGRAPH 1993.

[7] Sean O'Neil: *Accurate Atmospheric Scattering*, GPU Gems 2, Addison-Wesley Professional, 2005.

[8] Sean O'Neil: *Real-Time Atmospheric Scattering* (http://www.gamedev.net/columns/hardcore/atmscattering/)

[9] A. J. Preetham, Peter Shirley, Brian Smits: *A Practical Analytic Model for Daylight*, SIGGRAPH 1999.

[10] Erik Reinhard, Michael Stark, Peter Shirley, James Ferwerda: *Photographic Tone Reproduction for Digital Images*, SIGGRAPH 2002.

[11] Carsten Wenzel: *Real-time Atmospheric Effects in Games*, SIGGRAPH 2006.