Sugárkövetés (1. rész)

A sugárkövető algoritmusok szépsége, ereje a fény viselkedésének szimulációjában gyökerezik. Az olyan effekteket, mint a tükröződés vagy az árnyékok – amelyeket nehézkes scanline algoritmusokkal élethűen modellezni – a sugárkövetés természetes eredményei. Viszonylag könnyű implementálni és impresszív vizuális élményt nyújt. Most induló cikksorozatunk segítségével elkalauzoljuk olvasóinkat a ray tracing és a fotorealisztikus modellezés világába.

A klasszikus sugárkövetés összehasonlítása a hagyományos scanline algoritmusokkal

A sugárkövetés hátránya a viszonylag magas számítási idő. Míg a *scanline* algoritmusok kihasználják a fény koherenciáját, a sugárkövetés minden egyes sugárra külön értelmezi a műveletet, az előző sugár eredményétől függetlenül. Előnye, hogy nem csak poligon alapú felületekkel képes dolgozni, képes a fény és a különböző anyagok viselkedésének valósághű modellezésére.

A sugárkövetés (idegen szóval ***ray tracing)*** – mint arra a neve is utal – alapvetően a fotonok útjának követésén alapul, egy három dimenziós koordinátarendszerben matematikai függvényekkel leírt környezetből a fénytan törvényeinek megfelelően fotorealisztikus képet számol ki.

A mai grafikus ún. ***scanline*** algoritmusokkal dolgoznak. Bár ezeken a ***pixel*** és ***vertex shaderekkel*** felvértezett kártyákon is nagyon élethű effekteket lehet létrehozni (láttam már ***shader language***-ben írt sugárkövetőt is), ezek használata még mindig nem közelíti meg a sugárkövetéssel számolt képek minőségét. Jelen cikk írója szerint a poligon alapú grafikus gyorsítókat a jövőben fel fogják váltani a sugárkövetéssel dolgozó kártyák. Már léteznek ***raytracing*** kártyaprototípusok, bár kereskedelmi forgalomban még nem kaphatóak és hatalmas számításigényük miatt még gyermekcipőben járnak a fejlesztések. Egyelőre be kell érnünk a szoftveres sugárkövetéssel.

Természetesen léteznek Linux alá írt ingyenes sugárkövető programok, mint például a POV-Ray, és akár ezek használatát is ismertethetnénk e hasábokon... mi azonban eltekintünk ettől, ebben a cikksorozatban azt a célt tűzzük ki magunk elé, hogy az olvasókat az algoritmus mélységeibe kalauzoljuk, nem elégszünk meg a felszínes ismeretekkel.

Sajnos a témáról magyar nyelvű irodalom az interneten alig található, ezért igyekszem majd az egyes fogalmak angol megfelelőjét is megadni, így az olvasók könnyebben tudnak majd kapcsolódó publikációkat, írásokat keresni. Ebben a számban a matematikai alapokra is kitérünk, ezek ismerete létfontosságú lesz a továbbiak megértéséhez.

A szem működése

Kezdjük is mindjárt az alapoknál. Hogyan is működik az emberi szem? A szem a fény energiáját a retinában lévő idegelemek segítségével elektromos és kémiai ingerületté alakítja majd ezeket az ingerületeket a tényleges idegrendszer vezető elemei

1. ábra Ezen a képen jól látszanak a sugárkövetés jegyei: tükröződés, fénytörés, árnyékok

(látóidegek) az agy nyakszirti lebenyének meghatározott részéhez vezetik, ahol az elektromos információ bonyolult idegi mechanizmusok révén dekódolódik és képi információvá alakul. Szemünk optikai tulajdonságai révén fordított állású képet készít, amit az agy fordít vissza. A szem optikai része (szemlencse, üvegtest, stb.) a beérkező fényt egy fókuszpontba „tereli össze", ez a pont az ideghártya központi, kiemelt jelentőségű területe, az ún. sárgafolt (***makula***), ez az éleslátás tényleges kialakulásának a helye. A sárgafoltban megközelítőleg 127 millió receptorsejt szolgál a fényinger felvételére és kémiai ingerré alakítására.
Három fajta érzékelő sejtet lehet megkülönböztetni: a vörös, a kék és a zöld színekre érzékenyeket.
E három színből az összes többi előállítható, ezt a színkeverési metódust ***additív*** színkeverésnek nevezzük.
Az additív színkeverés tehát a szemünkben történik, ez a színes televízió, a monitorok és egyéb színes kijelző készülékek működésének alapja.
A szemünkbe egyidejűleg, a szem tehetetlenségét kihasználva gyors egymásutánban érkező, vagy egymáshoz igen közeli apró pontocskák formájában beérkező ***trikromatikus*** (három különböző hullámhosszúságból álló) színingerek additív színkeverés útján egyetlen színné alakulnak.

Egy kis matematika

Most, hogy túlestünk a látás anatómiai részén, kicsit megtornáztatjuk a bal agyféltekét, rátérünk a matematikai alapozásra. Azon olvasóink, akik e matematikai alapokkal teljesen tisztában vannak, akár át is ugorhatják ezt a részt. Az egyszerűség kedvéért legtöbbször vektoriális egyenleteket adunk meg, tehát feltételezzük az elemi vektormatematika ismeretét. Néhány műveletet speciálisan jelölök majd a formulákban, ezek: * = skalár szorzás (két vektor esetén a lentebb ismertetett skalár szorzat), sqrt = négyzetgyök, ^ = hatványozás (programozásban jártasabb olvasóinknak már ismerősek lehetnek e jelölések). A vektorok egy-egy komponensére a nagy betűvel szedett vektor neve utáni kis betűs komponensjelöléssel hivatkozok, azaz például Vx a V vektor X koordinátájára utal.
Egy három dimenziós koordinátarendszer X,Y,Z tengelyekből áll, ahol az egyes pontok helyvektorait a tengelyeknek megfelelő bázisvektorok (i,j,k) valamint adott skalár mennyiségek (X,Y,Z koordináták) szorzataként létrejövő vektorok összegeként definiálhatjuk, azaz P = X*i + Y*j + Z*k.
Három dimenziós koordinátarendszerben egy egyenes térbeli pontjait a következő egyenlettel definiálunk:
P = P0 + V*t

Ahol p0 az egyenesen fekvő adott pont helyvektora, V az egyenes irányvektora, t pedig egy tetszőleges skalár mennyiség. Bármilyen értéket adunk t-nek, a formulából számított P pont mindig az egyenesen fekvő pont lesz.
Az egyenes irányvektora az egyenes irányával párhuzamos vektor. Az irányvektor nagyon könnyen kiszámítható az egyenesen fekvő két pont ismeretében V = P1 - P0
Ha a kapott vektort normalizáljuk, akkor a fenti egyenletben t a P0 ponttól mért távolságot jelenti az egyenes mentén az irányvektor irányában. Egy adott vektor normalizálása egy olyan vektort eredményez, melynek iránya megegyezik az eredeti vektoréval, hossza egységnyi:
Vn = V / |V|
A három dimenziós vektor abszolút értékét (azaz hosszát) Pitagorasz tétele alapján egyszerűen kiszámíthatjuk a következő formula segítségével:
|V| = sqrt(X^2 + Y^2 + Z^2).
Amint azt a későbbiekben látni fogjuk, gyakran lesz szükségünk két vektor által bezárt szög kiszámítására. Ezt az ún. skaláris szorzat adja. Két vektor (A és B) skaláris szorzata egy olyan skalár érték, amely egyenlő a két vektor által bezárt szög koszinuszával.

A*B = cos(alfa) azaz cos(alfa) =
➥(Ax*Bx + Ay*By +
➥Az*Bz)/(|A|*|B|).

A gömb

A sugárkövetés metszéspontszámításokon alapul, futásidejének oroszlánrészét azzal tölti, hogy metszéspontokat keres fénysugarak (szakaszok) valamint a térbeli objektumok között.
A térbeli objektumokat, felületeket egyenletekkel kell definiálnunk ahhoz, hogy metszéspontokat tudjunk számolni. Az egyik legegyszerűbb felület a gömb (angolul ***sphere***).
Próbálkozzunk meg egy egyenes és egy origóban álló gömb metszéspontjának kiszámításával.
Tudjuk, hogy a gömb felülete azon pontok halmaza, amelyek egy adott ponttól (a gömb középpontja) azonos távolságban vannak. Ez a távolság a gömb sugara. Az origóban álló gömb pontjai ki fogják elégíteni a |P| = r egyenletet. Ha mind az egyenes mind

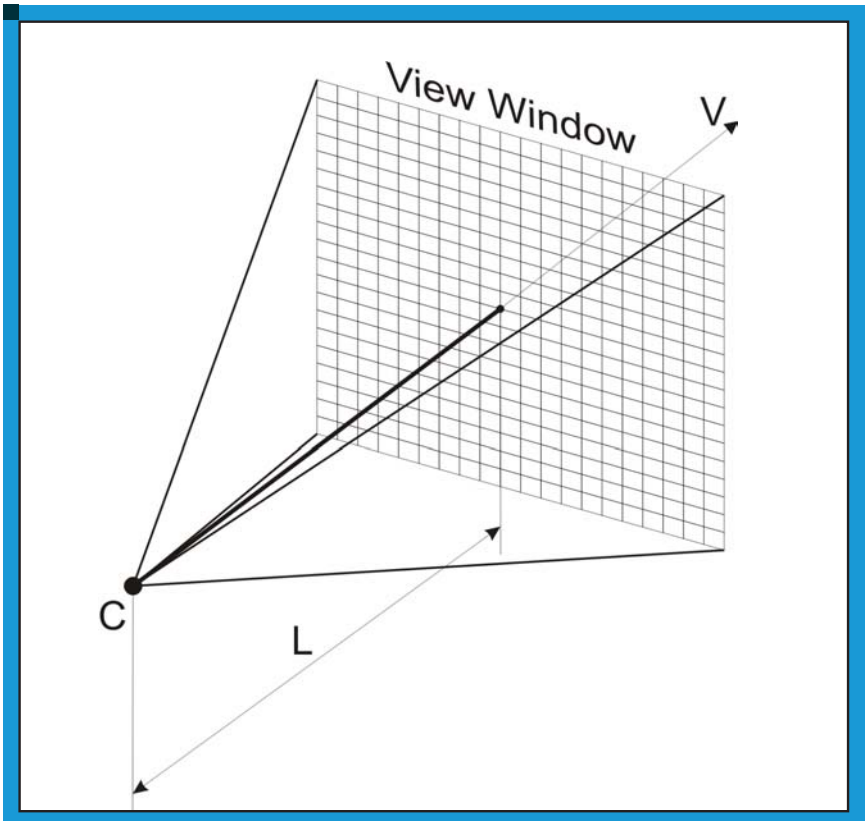
a gömb egyenletét koordinátákkal írjuk fel, a következő egyenletrendszerhez jutunk:

```
X = X0 + Vx*t
Y = Y0 + Vy*t
Z = Z0 + Vz*t
R = sqrt(X^2 + Y^2 + Z^2)
```

Az egyenes X,Y,Z koordinátáit rendre behelyettesítve a gömb egyenletébe majd a kapott egyenletet egyszerűsítve és a másodfokú egyenletek általános alakjára (a*x^2 + b*x + c = 0) hozva, a következő formulához jutunk:

```
(Vx^2 + Vy^2 + Vz^2)*t^2 +
➥(2*(X0*Vx + Y0*Vy + Z0*Vz))*t  +
➥(X0^2 + Y0^2 + Z0^2 – r^2) = 0
```

Ezek alapján a tetszőleges középpontú gömb egyenlete is levezethető, a tetszőleges középpontú gömb egyenlete |P – O| = r azaz r = sqrt((X – Ox)^2 + (Y – Oy)^2 + (Z – Oz)^2), ennek levezetését az olvasóra bízzuk.
A másodfokú egyenlet determinánsát kiszámítva (d = b^2 - 4*a*c) megtudjuk, hogy az egyenes metszi-e a gömböt (d nem negatív, azaz a másodfokú egyenletnek van megoldása) illetve hogy egy vagy 2 pontban metszi (amennyiben egy pontban metszi, ez az egyenes a gömb egyik érintője).
Az másodfokú egyenletek általános megoldóképletét (t1 = (-b + sqrt(b^2 – 4*a*c))/(2*a*b), t2 = (-b - sqrt(b^2 – 4*a*c))/(2*a*b)) felhasználva megkapjuk azon t értékeket, amelyet az egyenes egyenletébe behelyettesítve egy olyan pontot kapunk, amely a gömbfelület ponthalmazának része is egyben, azaz a gömb és egyenes metszéspontját!
Amint láttuk, a gömbfelületet egy másodfokú egyenlettel lehet definiálni, a gömb tehát egy másodfokú (kvadratikus) felület. A későbbiekben több másod és magasabb fokú felületet is ismertetni fogok, elégedjünk meg egyenlőre e felülettel.
Összegezve az eddigieket, most már ki tudjuk számítani fénysugarak és gömbök metszéspontját, azaz azt a pontot, ahol egy adott helyről indított fénysugár eltalál egy gömböt. Hamarosan látni fogjuk, hogy szükségünk lesz a felület metszéspontban számított normálvektorára is. A normál vektor egy olyan vektor, amely merőleges a felületre. Ez gömbök esetében rendkívül egyszerű, a normál vektor a gömb középpontjából a metszéspontba mutat, azaz: N = P – O. Amikor normálisokat számolunk, figyeljünk arra, hogy a normál vektor a felületből mindig kifelé mutasson!



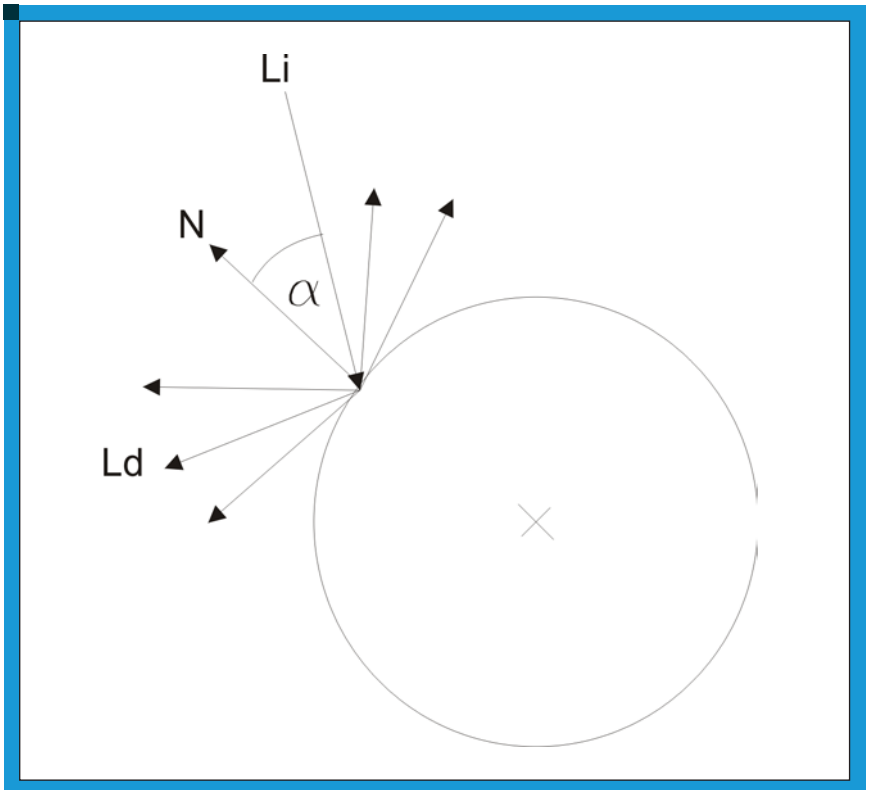
2. ábra A virtuális kamera felépítése és paraméterei

Fényforrások

A klasszikus sugárkövetés pontszerű fényforrásokat használ, a fotonok a tér egy definiált pontjából indulnak útjukra, azaz e fényforrásoknak nincs térbeli kiterjedése A valóságban ilyen fényforrások természetesen nem léteznek, hiszen minden fényt emittáló testnek kell legyen térbeli kiterjedése, ugye? Mi most ettől eltekintünk (későbbiekben még visszatérünk erre a témára), a számítások egyszerűsítése érdekében egyelőre csak pontszerű fényforrásokkal számolunk, azon beül is a legegyszerűbb esettel, irányítatlan, konstans fényforrással dolgozunk. Ez a fényforrás a tér minden irányába egyenlő mértékben bocsát ki fotonokat, a fényerősség pedig nem csökken a fényforrástól való távolság növelésével, tehát konstans a tér bármely pontjában. Az ilyen fényforrásnak mindössze két paramétere van: a kibocsátott fény színét és erősségét definiáló RGB hármas, valamint a fényforrás pozíciója.

Szemünk, a virtuális kamera

Ez mind szép és jó – mondhatja az olvasó – na de hogyan számoljunk ebből két dimenziós képet? A megoldás az emberi szem felépítését utánzó virtuális kamerában rejlik. A kamera pozícióját (a szem analógiájával élve a szemünk fókuszpontját) a tér egy adott pontjába helyezzük el, majd meghatározzuk azt az irányvektort, amerre a kamera néz (nézetvektor, idegen szóval ***view vector).*** *A* kamerapozíciótól adott távolságra pedig elhelyezzük a nézősíkot (***view plane***), amely merőleges a kamera ***view vector***-ára. A nézősík és a kamera pozíciója közti távolságot szemtávolságnak nevezzük, a továbbiakban L-lel jelölve. Ez a paraméter egy tetszőleges, 0-nál nagyobb érték, a perspektívát, kameránk látószögét szabályozhatjuk vele.



3. ábra A diffúz fény nézőpontfüggetlen, nagysága a beérkező fényenergia és a beesési szög függvénye

A nézősík az a sík, ahol a kiszámolt képünk keletkezik, ennek egy részén (***view window*** – nézőablak) fogjuk fel és tároljuk a beérkező fényenergiát. A nézőablak területét kis négyzetekre osztjuk fel oly módon, hogy minden egyes négyzet megfelel egy-egy pixelnek a számított képen, tehát a ***frame bufferünk*** minden egyes pixeléhez hozzárendeljük a nézőablak egy-egy celláját. Ha például egy 640x480 méretű képet akarunk számolni, a nézőablakot 640 oszlopra és 480 sorra osztjuk. A nézőablak méreteit ***FOV***-nak (***Field Of View*** – látómező) is nevezik a sugárkövetés terminológiájában. Az ***FOV*** – az L szemtávolsággal egyetemben – befolyásolja a perspektívát, kameránk látószögét. Az ***FOV*** áldásos hatása, hogy különböző felbontású képek számításakor minden esetben ugyanazt a térrészt fogja be kameránk (gondoljunk bele, ***FOV*** számítások használata nélkül nagyobb felbontású képek esetén a kameránk szélesebb látószöggel rendelkezne).
A sugárkövetés célja tehát nem más, mint a nézőablak minden egyes rácspontjára (azaz a ***bitmap*** minden egyes pixelére) kiszámítani a környezetből beérkező fényenergiát.

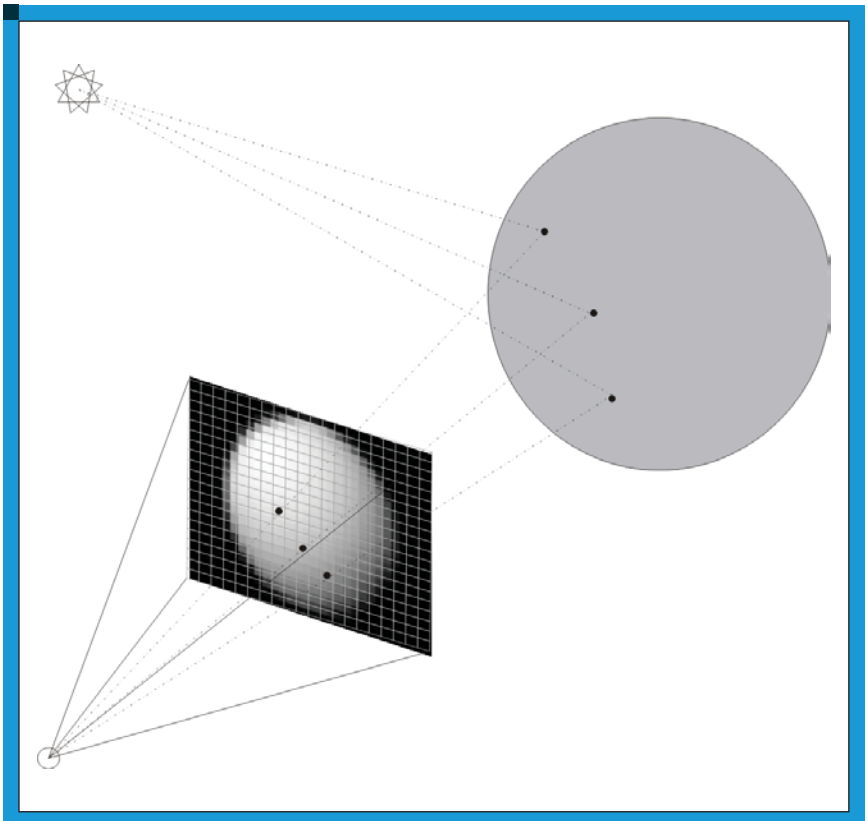
Anyag és fény

Gyorsan összefoglalnám az eddig tanultakat: fotonokat tudunk indítani a tér pontjaiból (fényforrások) és azon fotonokat, amelyek a kamerába nézősíkjába csapódnak be, fel tudjuk fogni, így kirajzolva a számított képet.
Igen ám, de hogy is van ez? Mi történik azokkal a fotonokkal, amelyek a fényforrásból egy objektumba csapódnak? A színes felületek színét az határozza meg, hogy a rájuk vetülő különböző hullámhosszúságú fényekből mennyit vernek vissza, illetve nyelnek el. A tárgyak színét tehát a megvilágítás színe is befolyásolja.
A különböző anyagok különbözőképpen reagálnak a fényre, ezért a modellezendő anyagtól (***material)*** függően kell a fény viselkedését leírni illetve modellezni. Mindjárt be is vezethetjük az árnyalási modell (***shading model)*** fogalmát, amely leírja, hogy egy felület milyen módon reagál a fényre.
Az árnyalási modell egy olyan függvény, amely az annak megfelelő anyagtulajdonságok alapján meghatározza, hogy az anyag felületére érkező fénysugár milyen reakciót idéz elő (a fényenergia mely részét tükrözi, melyiket szórja, melyiket nyeli el).

Több ilyen modell is létezik, egyesek a valóság egyre pontosabb megközelítésére születtek, míg mások csak egyetlen valós anyagtípus viselkedését próbálják meg modellezni. Az árnyalási modellek túlnyomó többsége elhanyagolja a fény hullámtulajdonságaiból adódó jelenségeket, azaz nem lehet velük például egy prizma viselkedését szimulálni. Egy már megírt és működő sugárkövető algoritmust némi hozzáértéssel módosíthatunk úgy, hogy figyelembe vegye a fény hullámtulajdonságait is (a fény spektrumát felpartícionálva több hullámhosszon is ki kell lőni ugyanazt a sugarat, valamint az árnyalási modelleket és a fényutak meghatározását is hullámhosszal paraméterezhetővé kell tenni), ennek megvalósítását azonban az olvasóra bízzuk.
Mivel ez a cikksorozat első része, az eddigiekben minden esetben a legegyszerűbb modellt ismertettük. Nem teszünk kivételt az árnyékolási modellekkel sem.
Lássuk tehát a diffúz, más néven a Lambert modellt. Lambert egyenlete egy tökéletesen diffúz, egyenetlen felületet vesz alapul, amely a tér minden irányába azonos fényerősséggel veri vissza a beérkező fényenergia egy részét. Ez azt jelenti, hogy a fényerősség nézőpontfüggetlen, azaz a tér bármely pontjából tekintünk egy felület adott pontjára, a visszavert fényerő konstans lesz. A diffúz fény függ a felület becsapódási pontban állított normálisától valamint e normális és a beérkező fénysugár által bezárt beesési szög koszinuszától:

Ld = Li*cos(alfa)

ahol Ld az eredményként kapott diffúz fény, Li a beeső fény, alfa pedig a már említett beesési szög. A diffúz árnyalási modellben szereplő együtthatókat a 2. *ábra* szemlélteti.
A diffúz árnyalási modellben egyetlen anyagtulajdonságunk van, ez pedig az anyag diffúz színe, melyet a már említett (és a számítástechnikában leggyakrabban használt) additív színkeverésnek megfelelően egy RGB hármassal definiálunk, 0 és 1.0 valamint a közéjük eső valós számokkal.
Itt jegyezném meg, hogy cikksorozatunkban mindvégig RGB hármasokkal dolgozunk, tehát amikor színre



4. ábra A diffúz árnyalással számolt kép kirajzolódik a bitmapben (a nézőablakban)

hivatkozok valahol, akkor az a szín mindenképpen egy RGB hármassal van definiálva. Ez a szín megadja, hogy az anyag felületére érkező fény mely részét veri vissza a környezetbe diffúz fényként, a következő képlettel számítható mindhárom összetevőre:

Ld = Cd*Li*cos(alfa)

Ez a modell kizárólag matt felületű anyagok modellezésére alkalmas, amelyek nem rendelkeznek tükörszerű fényvisszaverő vagy fényáteresztő tulajdonsággal.

Ray casting

Most már megvan mindenünk ahhoz, hogy egy egyszerű sugárkövetőt leprogramozzunk. Útjukra tudjuk indítani a fénysugarainkat a fényforrásból, ezen sugarak egy része az általunk definiált gömbökbe fog csapódni, s ezekről a diffúz árnyékolási modellnek megfelelő mennyiségű fény fog a kamera síkjára érkezni.
Ez eddig szuper, csakhogy figyelmen kívül hagytuk eddig azt a tényt, hogy a fénysugarak nagy része nem a kamera síkjára fog érkezni, legnagyobb részüket elnyeli a nagy semmi, ugye?

Ahhoz pedig hatalmas mennyiségű fénysugarat kellene kilőnünk és követnünk a fényforrásból, hogy a kamera síkján definiált rács pontjaira elegendő mennyiségű foton csapódjon be. Leszögezhetjük, hogy ez egy nagyon költséges algoritmus. Ennél lényegesen hatékonyabb megoldás lenne, ha a fénysugarakat visszafelé követnénk, azaz a kamerából indulnánk el a fényforrások felé. Ez utóbbi megoldást ***backward***, előbbit ***forward ray tracing*** -nek hívják szakmai berkekben. Mi főleg a ***backward ray tracing***-gel fogunk foglalkozni, de későbbi cikkekben még valószínűleg visszatérünk a ***forward*** megoldáshoz is egy-két szó erejéig.
A diffúz modellel megvalósított visszirányú (***backward***) sugárkövető megvalósítása tehát a következő fő lépésekből áll:

1. Felinicializáljuk a ***scene***-t, azaz definiáljuk a gömbjeinket (középpont, sugár, diffúz szín), a kamerát és a fényforrásokat

2. Kilövünk egy sugarat (egyenest) a kamerából a nézőablakon definiált következő rácsponton (a következő pixelen) keresztül. Ezen az egyenesen 2 pontunk van: a rácspont és a kamerapozíció Ezt a sugarat elsődleges sugárnak (***primary ray***) vagy szemsugárnak nevezzük (***eye ray***), mivel a kamerából lőjük ki, és (mint azt a rekurzív sugárkövetés ismertetésénél látni fogjuk) a fényút végigkövetésében ez az első kilőtt sugár is egyben.

3. A fenti egyenletek segítségével könnyedén meghatározható, hogy ez az egyenes metszi-e a térben elhelyezett gömbjeink valamelyikét. Fontos megemlíteni, hogy a negatív t értékeket figyelmen kívül kell hagynunk, hiszen ezek kamera mögötti metszéspontokat takarnak.

4. Amennyiben igen, kiválasztjuk a legközelebbi metszésponthoz tartozó felületet (hiszen ez takarja a mögötte levőket), s megnézzük, a fény milyen szögben érkezik a metszéspontban állított normálishoz képest.

5. A normális és a metszéspontból a fényforrásba mutató vektor (a fényvektor, azaz L = I – LS) által bezárt szög koszinusza kiszámítható a fent említett skaláris szorzat segítségével. A diffúz formula pedig cos(alfa) és a fényerősség alapján megadja, hogy mennyi fény érkezik a kamera síkjának adott pontjára. Itt figyeljünk arra, hogy egy fényforrás a gömbnek csak az egyik félgömbjét világítja be, tehát ha a skaláris szorzat negatív (a szög nagyobb, mint 90 fok), akkor a pont árnyékban van!

6. A 2-6 lépéseket ismételgetjük egészen addig, amíg a nézőablak minden rácspontját nem érintettük, azaz a kimeneti képünkön minden pixelre ki nem számítottuk az RGB hármast.

Ezeket a lépéseket a következő pszeudokód jellegű C függvény is jól szemlélteti, ennek segítségével az alkotó kedvű olvasók a következő számig már meg is írhatják az első ***Hello World*** sugárkövető programocskát.

Ebben a nagyon leegyszerűsített algoritmusban nem ismertettük a *scene* inicializációt (az olvasóra bízzuk, hisz implementációfüggő), a kamera pozicionálása és irányítása pedig nincs megoldva. Kameránk az origóban helyezkedik el és előrefelé (a Z tengely pozitív irányába) néz (azaz a nézővektora 0,0,1).
A kamera tetszőleges pozicionálásához és irányításához már fel kell használunk a sugártranszformáció módszerét, amelyet a következő számban fogok ismertetni. A számolandó kép méreteit `iWidth` és `iHeight`, az ***FOV*** méreteket `iFOVWidth`, `iFOVHeight` jelöli. Az algoritmus nem számol tükröződéssel, fénytöréssel és árnyékokkal sem.
A fény útját csak az első becsapódásig követjük, a felületek közti fénytani kölcsönhatásokat elhanyagoljuk. Ezt az algoritmust nevezzük ***ray castingnak***, ***Arthur Appel*** mutatta be még 1968-ban. A rekurzív sugárkövetést, már fénytörések, árnyékok, tükröződések modellezésére is alkalmas. A rekurzív sugárkövetés ***Appel*** úr úttörő munkáján alapszik, de ezt már ***Turner Whitted***-nek köszönhetjük, aki 1979-ben publikálta munkájának gyümölcsét. Amint azt a neve is mutatja, a rekurzív sugárkövetés tulajdonképpen rekurzív ***ray casting***, de erről bővebben majd a következő számban olvashattok.
A következő számban tehát ***Whitted úr*** művével fogunk kezdeni, ezután rátérünk a sugártranszformációkra, a textúrázásra, Phong árnyalási modelljére, majd további felületekkel ismerkedünk meg. Egyesek talán kicsit „szájbarágósnak” érezhetik az olvasmányt – nem minden alap nélkül. Az ő megnyugtatásukra azért leírom, hogy ez szándékos, de csak az első számot terveztem ilyenre, a továbbiakban nagyobb léptékkel haladunk majd és igyekszem majd tömörebben fogalmazni. Ettől függetlenül remélem sokak érdeklődését sikerült felkeltenem a téma iránt. Ha valakinek kérdései merültek fel a sugárkövetéssel kapcsolatban, vagy véleményét szeretné kifejezni, az ne hezitáljon, dobjon egy e-mailt a lenti címre, amint időm és energiám engedi, válaszolni fogok.

1. Lista A ray casting implementáció diffúz árnyalással

```
void RenderScene() {
  int iL = 300;  // Ez a fókusztávolság, azaz a nézősík távol
                 // sága a kamerapozíciótól
  Vector  vectNormal, vectEye, vectIntersection;
  RGB rgbSphereDiffuseColor = {1.0, 1.0, 1.0}; // A gömbök
                        //diffúz színe (ez esetben fehér)
  RGB rgbPixelColor;
  for (int iX = 0; iX < iWidth; iX++) {
    for (int iY = 0; iY < iWidth; iY++) {
      vectEye = {((iX - (iWidth/2))*iFOVWidth)/iWidth,
      ➥((iY - (iHeight/2))*iFOVHeight)/iHeight,  iL};
      rgbPixelColor = {0.0, 0.0, 0.0};

      // Minden gömböt megvizsgálunk, van-e metszéspontja
      // a szemsugárral (aSpheres egy
      // globális tömb, amelyben a gömbdefiníciókat tároljuk)
      for (int iSphereIdx = 0; iSphereIdx < giNumOfSpheres;
      ➥iSphereIdx++) {
        // Van metszéspontunk?  Amennyiben van,
        // IntersectSphere() true-val tér vissza, valamint
        // a metszéspontot és az abban adott felületi normá
        // list a mutatóikon keresztül átadott
        // Vektor  struktúrákba írja.
        if (IntersectSphere(vectEye, aSpheres[iSphereIdx],
        ➥&vectNormal, &vectIntersection))  {
          // Minden fényforrás hozzájárulását hozzáadjuk
          // a metszéspontból a kamerába érkező
          // fény színéhez
          for (int iLigtSrcIdx = 0; iLigtSrcIdx <
          ➥iNumOfLightSources; iLigtSrcIdx++) {
            // CalcDiffuse kiszámítja a metszéspontba érkező
            // fényt az  iLigtSrcIdx által
            // indexelt fényforrásból. aLightSrcs egy globális
            // tömb, amelyben a
            // fényforrásainkat tároljuk)
            rgbPixelColor += CalcDiffuse(vectNormal,
            ➥vectIntersection, aLightSrcs[iLigtSrcIdx],
            ➥agbSphereDiffuseColor);
          }
        }
      }
      // PutPixel  a megadott koordinátákon található pixelt
      // a megadott szinűre
      // színezi, azaz kirajzolja a pixelt a kiszámított
      // színnel.
      PutPixel(iX, iY, rgbPixelColor);
    }
  }
}
```

Szendi Ákos
(akos.szendi@gomortel.hu)

27 éves, szabadúszó programozóként tevékenykedik. A Miskolci Egyetem villamosmérnök szakos hallgatója. Kevéske szabadidejében gitározni tanul vagy épp egy jó könyvet tart a kezében.