SZAKDOLGOZAT

Perjéssy Lóránt

Debrecen

2010

Debreceni Egyetem

Informatikai Kar

MOZGÓKÉP ELŐÁLLÍTÁSA SUGÁRKÖVETÉSES KÉPALKOTÁS SEGÍTSÉGÉVEL

Témavezető:
Dr. Schwarcz Tibor
Egyetemi adjunktus

Készítette:
Perjéssy Lóránt
Programtervező informatikus

Debrecen
2010

Tartalomjegyzék

Bevezetés

A számítógépes grafika gyakorlatilag a vizuális megjelenítőeszközök kialakulása óta egy igen népszerű irányzat. Bár a kezdetekben nem volt lehetőség élethű és vizuálisan pontos vagy igazán kellemes képek megjelenítésére, már ekkor is voltak törekvések az emberi szem és tudat által képnek mondható információ megjelenítésére, gondoljunk csak az úgynevezett ASCII-artokra. A valódi (és főképp a valós idejű) képalkotási lehetőségek azonban a számítógépek grafikus gyorsítókártyáinak segítségével érkeztek el, az információ technológia ezen területén azonban mindig van hová fejlődni.

A komputergrafika népszerűségének egyik valószínű oka feltehetőleg az egyik fő jellemzőjével magyarázható: szó szerint látványos. Ennél fogva ez az irányzat könnyedén magával tudja ragadni akár a laikusokat is. A végső soron kialakult kép mögött azonban sokféle matematikai háttér állhat. A különböző algoritmussal ellátott programok viszont nem csak hatékonyságukban különböznek, a kialakult kép megjelenését, realisztikusságát is befolyásolják. Ezen szakdolgozat a képalkotás egyik ilyen módszerével, a sugárkövetéses képalkotással foglalkozik.

A dolgozat első fejezetében ennek a képalkotási módszernek a bemutatását láthatjuk. Megismerjük az eljárás matematikai hátterét, az előnyét és hátrányát más módszerekkel szemben, az algoritmus gyorsításának lehetőségeit, valamint a kép végső minőségét javító technikákat is.

A második fejezetben ennek a technológiának a megvalósítását lehetővé tevő hardver- és szoftvereszközökről lesz szó. A képalkotás animáció szintű alkalmazásával is megismerkedünk az által, hogy megnézzük, a mai számítógépes eszközök milyen lehetőségeket biztosítanak számunkra korábbi társaikhoz képest, és mik azok az algoritmusok, amelyek egyszerűen a számítási igényeik miatt nem férnek (egyelőre) a megvalósíthatóság keretei közé.

A harmadik fejezetben egy ilyen, valósidejű sugárkövetéses algoritmus megvalósítását láthatjuk. Szó esik majd az igénybe vett hardver- és szoftverelemekről, a képalkotást végző program(ok) feladatáról, a nagy számításigényű feladatok gyorsításának módszereiről és az elkészült program futásidejű statisztikáiról.

A negyedik és egyben végső fejezetben ennek a képalkotási módszernek a közeli és

távoli jövőbeli lehetőségeit mérlegeljük.

A szakdolgozat célja egy az erre az eljárásra épülő korszerű program bemutatása és egyben az ebben az irányzatban rejlő lehetőségek ismertetése magyar nyelven, ugyanis a számítógépes grafikával foglalkozó magyar nyomtatott és internetes szakirodalom sajnos még ma is le van maradva angol társához képest, valamint az információ technológia különböző ágazataival foglalkozó dokumentumok a szak folyamatosan fejlődő jellegének köszönhetően soha nem lehetnek eléggé naprakészek.

1. A sugárkövetésről általánosságban

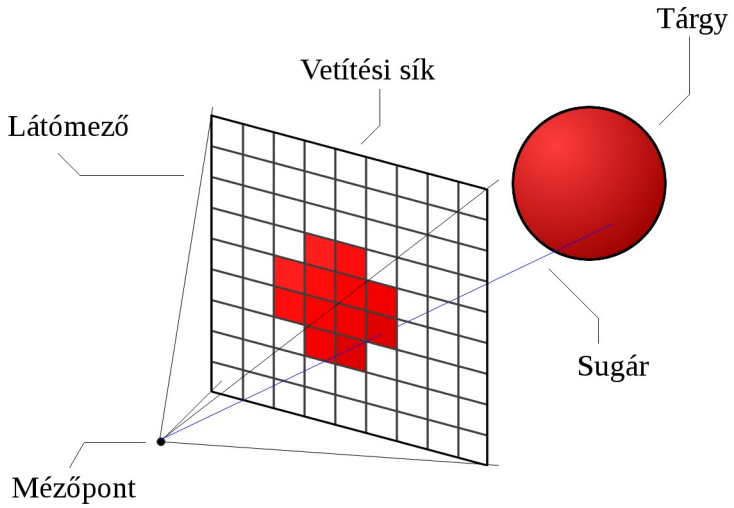
A *sugárkövetéses algoritmus* (*ray tracing*) lényege, amint a nevéből is következtetni lehet rá, a fénysugarak, fotonok útjának lekövetése számítógépes szimulációval. Ez a fajta képalkotás, vagy más néven *renderelés*, igen közel áll a fény természetes fizikai terjedésének működéséhez, attól azonban egy igen egyszerű, de annál fontosabb jellemzőben eltér: az algoritmus során nem a fényt kibocsátó tárgyaktól kiindulva követjük végig a fotonok útvonalát a szemünkig, vagy a kamera érzékelőjéig, hanem fordítva: a modellezett világ szemlélőjének pontjától indítunk sugarakat a látómező minden irányába és ezen sugarakat követve döntjük el, hogy a néző mit lát az adott irányba tekintve.

Ezen módszer kialakulásának története egészen a XX. század közepéig nyúlik vissza. Az 1960-as években már ismert volt az algoritmus, azonban az akkori számítógépek nem tették lehetővé a használatát, így sokáig feledésbe is merült, más, gyorsabb képalkotási módszereknek adva át a teret. Andrew S. Glassner 1986-ban a SIGGRAPH nevezetű évente megrendezésre kerülő komputergrafikai konferencián előadásával azonban ismét behozta a köztudatba az algoritmust, majd a népszerűségnek köszönhetően 1989-ben *An Introduction to Ray Tracing*[1] című könyvében összegyűjtötte és publikálta az addigi, témával kapcsolatos ismereteket. Ezt követően a számítási teljesítmények folyamatos növekedése miatt egyre több területen kezdték el alkalmazni. A 1990-es évek első felében már a játékipar is alkalmazta, természetesen a módszer igencsak leegyszerűsített változatait, amelyek lehetővé tették, hogy az akkori PC-ken valós idejű megjelenítést érjenek el. Legnagyobb hasznát azonban a filmipar látja, mert ez az a terület, ahol a valós időben történő megjelenítést mellőzve is tudják alkalmazni, évről évre látványosabb előrelépések keretében.

1.1. Az algoritmus működése

A módszer, alapjában véve igen egyszerű, tekintve, hogy közel áll a fény általunk ismert működéséhez és ezáltal könnyedén megérthető: a nézőpontból kiindulva egy vetítési sík minden pontján át indítunk egy félegyenest. Ezeknek a félegyeneseknek vesszük a metszéspontjait a modellezett világ összes tárgyával, majd minden sugárnál a nézőponthoz legközelebb eső metszéspontot választjuk ki. Mivel a metszéspontszámítás igen költséges feladat, ezért az algoritmus használatakor általában csak háromszögeket és gömböket

használunk primitív elemként, a komplexebb alakzatokat pedig ezekből építjük fel. A legközelebbi metszéspont megtalálása után megkapjuk az adott irányban látható legközelebbi tárgyat (tehát a keletkező képen a tárgyak takarása is megfelelő lesz).



1. ábra: A képalkotás elemei

Ezt követően az adott pontot árnyalnunk kell. Itt gyakorlatilag nagyon sok módszert alkalmazhatunk, de ha valósághű képalkotásra törekszünk, az árnyalási egyenlet[2] lehető legteljesebb alkalmazására kell törekednünk:

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_{\Omega} L_r(x,\omega_i,\omega_o)\, L_i(x,\omega_i)\, d\omega_i$$

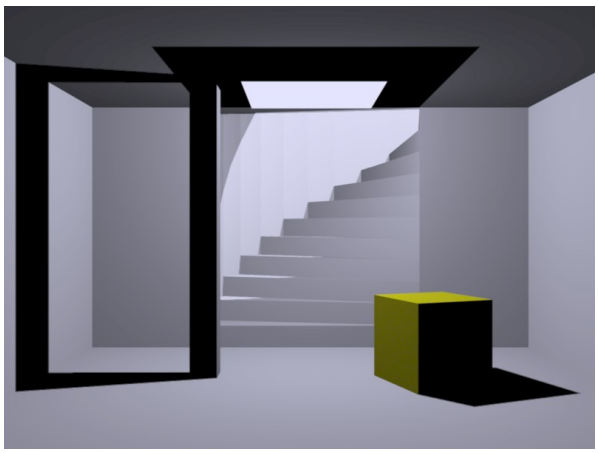
A képletben szereplő $L_o(x,\omega_o)$ adja meg egy x pontból ω_o irányba kilépő fény erejét. Az anyag önmaga által kibocsátott fényét az $L_e(x,\omega_o)$ függvény, az anyag által visszavert fényt az $L_r(x,\omega_i,\omega_o)$ kétirányú visszaverődési eloszlásfüggvény (BRDF), és a felületre érkező fényt az $L_i(x,\omega_i)$ függvény határozza meg. Az ω_i beesési irányt is figyelembe vevő függvényeket természetesen a felületet érő összes irányból ki kellene számolni, ez a módszer azonban gyakorlatilag alkalmazhatatlan lenne.

Ez alapján két nagy csoportra oszthatjuk a sugárkövetéses algoritmusokat:

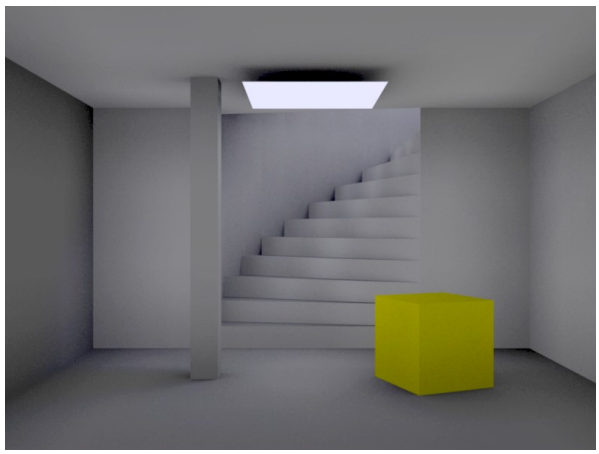
- A *globális illuminációs algoritmusok* (2. ábra) az árnyalási egyenletet a lehető legpontosabban, nagyon kevés egyszerűsítéssel próbálják közelíteni. Ezek az eljárások az integrál alatti területet általában véges sok sugár halmazával próbálják

helyettesíteni, melyeket egy általában a felület adott pontja fölé emelt képzeletbeli félgömb felületének irányába indítanak el, többnyire egyenletes eloszlással. A hangsúly itt a sok sugár alkalmazására helyezendő, mivel csak így lehet megfelelően utánozni a természetben is előforduló szórt visszaverődéseket.

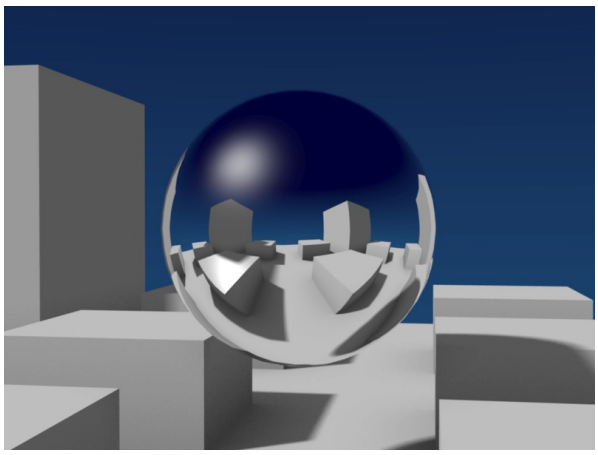
- A *lokális illuminációs algoritmusok* (3. ábra) ezzel szemben megpróbálják leegyszerűsíteni az árnyalási egyenletet, elhagyva belőle bizonyos elemeket. A visszavert fényeknél általában csak tökéletes tükröket (4. ábra) szoktak alkalmazni, ezáltal a tükröződések során elegendő mindössze egyetlen sugár útját követnünk a tükröződés után is. Szintén fontos egyszerűsítésnek lehet alávetni a felületre érkező fényt leíró függvényt azáltal, hogy a szomszédos tárgyakról érkező szórt fényt nem, csak a fényforrásokból érkező közvetlen fényeket vesszük figyelembe. Ezáltal természetesen csökken a realisztikusság, de az algoritmus hatékonysága és ezáltal a sebessége nő.



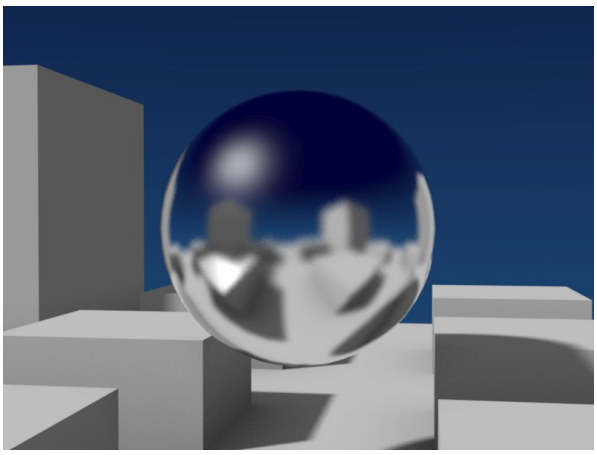
2. ábra: Lokális illumináció



3. ábra: Globális illumináció



4. ábra: Tökéletes tükör



5. ábra: Matt tükör

1.2. Előnyök és hátrányok

A ray tracerek és a másik igen elterjedt képalkotási algoritmus, az *inkrementális képszintézis* (másik nevén *scanline algoritmus*) között nagyon sok eltérés akad, melyek befolyásolják mind az ezeket alkalmazó programok sebességét, mind az alkotott képet.

A két módszer közötti alapvető eltérés a két különböző megközelítésből fakad: míg a sugárkövetésnél azt nézzük meg, hogy az elkészülő képen mely képpontokon át mely tárgyakat láthatjuk, addig az inkrementális képalkotásnál pontosan fordítva járunk el: azt nézzük meg, hogy a világ egyes elemei a képnek mely képpontjait foglalják majd el. Ehhez, háromszögek alkalmazása esetén mindössze az első csúcspont helyzetét kell meghatároznunk, a háromszög többi pontját a szomszédos képpontokon tovább haladva már sokkal gyorsabban meg tudjuk határozni, mert csupán összeadásra és egyenlőtlenség-vizsgálatra van szükség hozzá. A sugárkövetésnél azonban minden sugarat egymástól függetlenül kell elindítanunk, így nem tudjuk előre meghatározni, hogy a korábban már kiszámolt sugár-objektum metszetek alapján mely térbeli elemeket zárhatjuk ki az újabb sugarak indításánál. Ezen felül az algoritmus legszűkebb keresztmetszetét a metszéspontszámítás jelenti, amelyben használatra szorul az osztás, a skaláris- és a vektoriális szorzat is.

Sebesség területén tehát egyértelműen alul marad a sugárkövetéses algoritmus, a realisztikusság területén azonban ez már koránt sincs így. Bár mára már igen fejlett technológiát képviselnek az olyan inkrementális képalkotást alkalmazó programcsomagok,

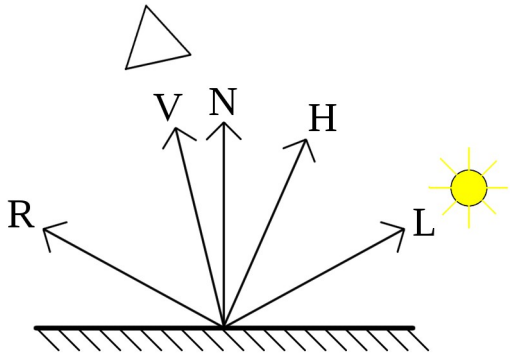
mint a DirectX, vagy az OpenGL, azonban ezek a könyvtárak az eljárás tulajdonságából adódóan idővel valószínűleg zsákutcába jutnak: bizonyos vizuális jelenségeket nem képesek, vagy csak nagyon nehezen tudnak megalkotni. Ilyennek számít a többszörös tükröződés, a kiterjedéssel rendelkező fények esetén keletkező félárnyékok, az indirekt megvilágítás, amelynél az objektumok befolyásolják a szomszédos tárgyaik végső színét, de a globális illumináció bármely más tényezője is ide sorolható. Ezek mind olyan jelenségek, amelyeket csak nagyon nehezen lehet alkalmazni scanline algoritmusokban, így hiába rendelkezünk a megfelelő hardveres felszereltséggel, az ilyen szintű számítógépes grafika létrehozásánál már nem nyernénk jelentős sebességnövekedést, ha a scanline algoritmust használnánk.

1.3. A kép előállítása és minőségének javítására

1.3.1. Árnyalás

A legelső sugárkövetéses algoritmusok a sugarak nézőpontból történő elindítása után mindössze az első metszéspontig követték azokat, ott azonban megálltak. A vetítési sík azon pontját, amelyen egy adott sugár áthaladt, a képernyőn pontosan megfeleltethetjük egy képpontnak. Ahhoz azonban, hogy az adott képpontban láthassunk is valamit, meg kell állapítanunk, hogy a metszéskor kiválasztott tárgy felületének ezen pontja milyen fényintenzitással rendelkezik. Ehhez alkalmazhatunk nem-valósághű árnyalási módszereket is, de ezeket általában technikai illusztrációkhoz, rajzfilmekhez, ritkább esetben videojátékokhoz, vagy bármilyen kísérleti művészeti jellegű alkotásokhoz szokták használni. A realisztikus árnyaláshoz az arra alkalmas kétirányú visszaverődési eloszlásfüggvényt kell választanunk. Használhatjuk a Lambert koszinusz törvényére épülő egyszerű képletet, amely az adott pontból a fényforrás felé mutató egységvektor és a felület normálvektorának skaláris szorzatával, valamint az anyag színével adja meg a felület fényerejét. Ez a módszer bár elég egyszerű, de sík, matt felületek megjelenítésére kiválóan alkalmas és gyakorlatilag erre épül sok másik árnyalási algoritmus is. Görbe, valamint fényes felületekhez már célszerű a Phong-árnyalást, vagy ennek egy módosított változatát, a Blinn-Phong árnyalást használni. Valósidejű képalkotásnál ma ezeket az algoritmusokat szokták leginkább alkalmazni. Állókép renderelésénél használhatunk ennél bonyolultabb árnyalásokat is, olyanokat mint az Oren-Nayar, Ward anizotróp, vagy a komplex, de igencsak valósághű Cook-Torrance. Mivel a

dolgozatban a későbbiekben a Blinn-Phong módszer fog szerepet játszani, ezért most álljon itt ennek az árnyalásnak a szemléltetése:



6. ábra: A Blinn-Phong árnyaláshoz szükséges vektorok

Jelölje a felület adott pontban vett normálvektorát N, a nézőpont felé mutató egységvektort V, a fényforrás irányába mutató egységvektort L, valamint ennek az N szerinti tükörirányát R. Legyen az adott pont végső színe:

$$c = c_a + \sum_k \left(m_d l_d (L_k \cdot N) + m_s l_s (R_k \cdot V)^n \right),$$

ahol c_a az ambiens, mindenhol jelen lévő környezeti fény színét, m_d az anyag diffúz, m_s az anyag spekuláris színét adja meg, l_d és l_s rendre a k-adik fényforrás diffúz és spekuláris színét jelölik, valamint az n pozitív valós szám egy szabadon választott hatvány, amely a felületen megjelenő csúcsfény erősségét befolyásolja (minél nagyobb ez a szám, annál kisebb lesz a csúcsfény területe). Ez a hagyományos Phong árnyalási modell.

A Blinn-Phong árnyalást eredetileg azért prezentálták, hogy elérhető legyen egy, az előbbinél gyorsabb, de ahhoz nagyban hasonlatos modell. Ennek ellenére nem csak hatékonyabb, de jobban meg is közelíti a valós világbeli csúcsfények jellegzetességét, mert a Phong-árnyalással ellentétben a sík felületeken itt a csúcsfények nem kör alakúak lesznek, hanem annál elnyúltabb alakúak[3]. A módosított árnyalás a következő lesz ezáltal:

$$c = c_a + \sum_k \left(m_d l_d (L_k \cdot N) + m_s l_s (N \cdot H)^n \right),$$

ahol $H = \frac{L+V}{|L+V|}$, tehát a tükrözött R vektort egyáltalán nem kell kiszámolnunk.

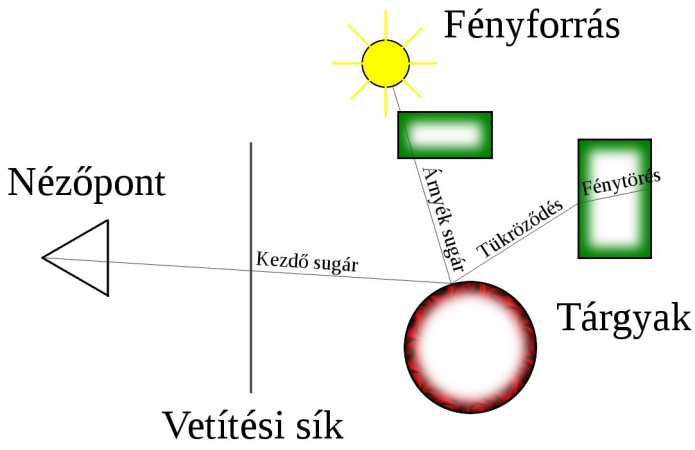
A keletkezett képen így láthatóvá váló objektumoknak van egy közös tulajdonsága, mégpedig, hogy mind egyetlen, homogén színnel rendelkezik. A valóságban ezzel szemben a tárgyaknak igen részletes mintázatai is előfordulnak, ezt nevezzük textúrának. A komputergrafikában ma már általános eljárás az anyagmintázatok használata. Az anyagmintázat megadásának két fő módszere létezik:

- *Előre elkészített textúra*, melynél egy korábban elkészített, vagy a valós világból fényképezéssel létrehozott anyagmintázatot használunk fel újra. Ezek használatához általában UV koordinátákat kell alkalmazni, melyek olyan kétdimenziós vektorok, amelyeket a modellezett világ minden elemének minden csúcspontjához megadunk, és a mintázatot ez alapján illeszti rá a program a felületekre.
- *Procedurális textúra*, melynél minden egyes képponthoz a képponton át látható metszéspont világbeli koordinátája alapján határozunk meg egy algoritmus segítségével egy színt.

Bármilyen textúrázást is használjunk - akár többet is egyszerre -, a minta többféleképpen befolyásolhatja a pont keletkező színét. Módosíthatja annak bármely tényezőjét: ambiens, diffúz, spekuláris szín, spekuláris csúcsfény hatványa, de akár az adott pont normálvektorát is. Bármilyen operátorral összekötve alkalmazhatjuk ezt, például a legáltalánosabb esetben a textúra színének és az anyag színének a szorzatát szokás venni.

1.3.2. Komplex sugarak

Eddig a lépésig azonban csak a közvetlenül látható tárgyak megállapítására volt lehetőségünk. Turner Whitted 1980-ban megjelent publikációjában[4] tovább vitte az algoritmust és három új sugártípussal bővítette azt (7. ábra): árnyékokkal, tükröződésekkel és fénytörésekkel. Ezek segítségével az algoritmussal már elő lehetett állítani vetett árnyékokat, tükröződő felületeket, valamint átlátszó vagy áttetsző objektumokat, amiken át láthatóvá válnak a mögöttük lévő tárgyak, sőt akár a felületekhez törésmutatót rendelve a fénytörés optikai jelenségét is szimulálni lehetett.



7. ábra: A Whitted ray tracer sugártípusai

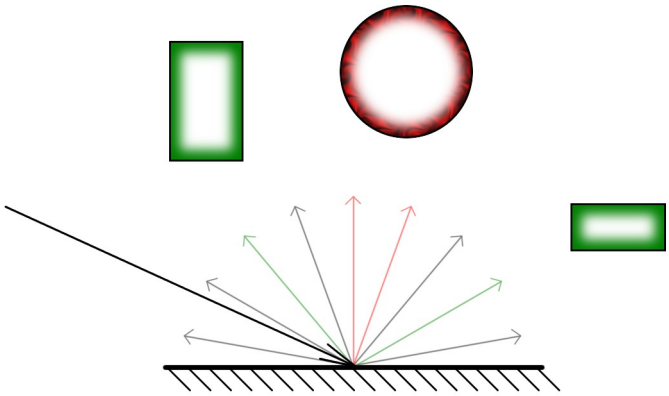
A Whitted ray tracer egy metszéspont megtalálásakor félegyenest indít az összes fényforrás irányába. Ha valamelyik sugár nem éri el közvetlenül a kijelölt fényforrását, mert egy köztes tárgy eltakarja azt, akkor a képpont színének számításakor nem veszi figyelembe az összegben az ezzel a fényforrással számoló tagokat.

Amennyiben az induló sugár által elmetszett objektum tükröződési együtthatója (amely nulla és egy közötti értéket vehet fel, ahol az egy jelenti a tökéletes tükröt) nagyobb, mint nulla, akkor az algoritmus rekurzívan egy újabb sugarat indít, mely a bejövő sugár a normálvektorra történő tükrözésével megkapható. A tükörirányú sugár ezek után az érintett felület tükröződési hányadosától függően fog hozzájárulni a képpont végső színéhez. Az m_r nulla és egy közötti tükröződési együtthatóval ellátott felület adott pontjában $1 - m_r$ mértékben játszik szerepet az anyag konkrét színe és m_r mértékben a tükörsugár által összegyűjtött összes többi fény. A rekurziót általában egy előre megadott lépésszámig szokták folytatni, vagy addig, amíg a következő sugár el nem ér egy bizonyos küszöbérték alatti mértéket, melynek köszönhetően elhanyagolhatóvá válik.

Ha a sugár áttetsző felületet érint, majdnem hasonló módon kell eljárni. A felületnek legyen m_t az áttetszőségét mutató nulla és egy közötti faktor. Ekkor a felület önálló színe $1 - m_t$ mértékben befolyásolja a végső színt, míg a felület mögötti elemek m_t mértékben, amelyek a beérkező sugár egyenes irányú folytatásával érhetőek el. Amennyiben a modellezett felület előtti és mögötti közegek eltérő anyagsűrűséggel rendelkeznek, úgy a

Snellius-Descartes-törvény alapján határozhatjuk meg a sugár új irányát. Színszórást is elérhetünk, ha egy új sugár helyett többet indítunk el a fénytörés után és mindegyikhez különböző színt (hullámhosszú fényt) rendelünk. A sugarak csak a nekik megfelelő fényt fogják fel útjuk során, a fénytörés pontjában pedig ezeknek a sugaraknak a színét kell összegezni.

Amennyiben a globális illumináció alkalmazására is törekszünk, nagyban hozzájárulhatunk az elkészülő kép fotorealisztikus mivoltához. Ezt több módszerrel is megtehetjük.



8. ábra: Indirekt megvilágítás

Az *ambiens okklúzió* (*ambient occlusion*, a környezeti fények takarása) módszerével javíthatunk a mindenhol jelenlévő fény monotonitásán. Ehhez a metszéspont megtalálásakor a felület pontjáról egy félgömb felületének irányába több különböző sugarat kell indítanunk (8. ábra). Ezeket a sugarakat általában a Monte Carlo-módszer segítségével osztják el. Az adott pont ambiens együtthatójának intenzitását az alapján határozzák meg, hogy a sugarak hány százaléka nem talált el egy objektumot sem egy bizonyos (általában kis) távolságon belül.

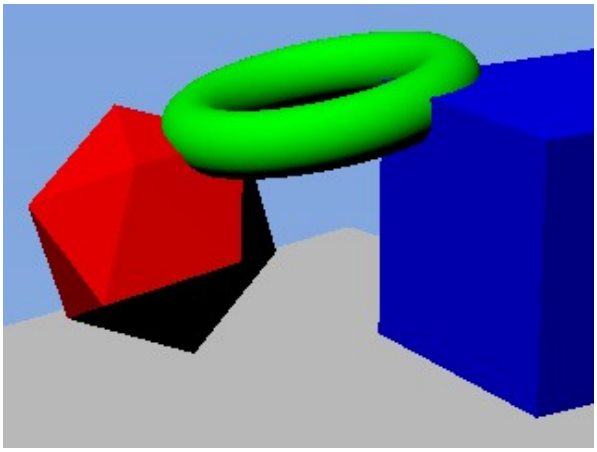
Ezt az eljárást kicsit megváltoztatva a tárgyak egymásra gyakorolt színező hatását is elérhetjük. Ez az irány vezet el a *path trace* módszeréhez, amelynél a felületek diffúz színét a környező tárgyak színe is befolyásolja. Pillanatnyilag ez a legélethűbb képalkotási módszer, mert ez közelíti meg legteljesebben a fény valós természetét, de egyben ez a leglassabb is. Azokat a fénytani hatásokat, amelyeket a path trace alkalamzása során az algoritmus működéséből kifolyólag mindenképpen megkapunk, más módszereknél trükkökkel

alkalmazhatjuk. Ilyen például a globális illuminációt megközelítő *radiozitás* (*radiosity*) módszere, amellyel a szórt fényeket előre kiszámoljuk és eltároljuk a tárgyak csúcspontjai mentén, vagy a *fotontérképek* (*photon mapping*) alkalmazása, melyek a fényforrásokból induló fotonok útját követik le és ezáltal rögzítik a felületekre eső fény erejét.

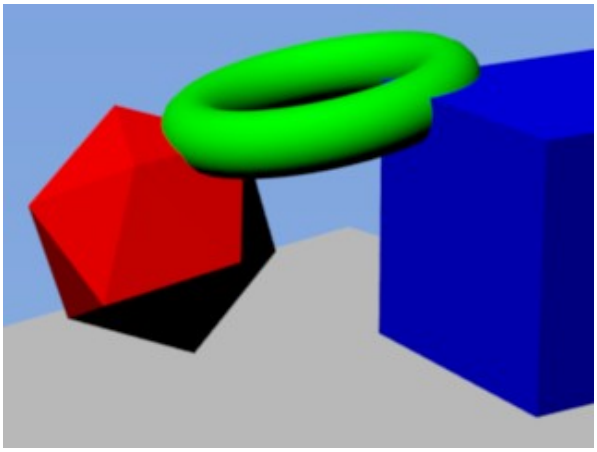
1.3.3. Fotorealisztikus renderelés

Egy komputergrafikával alkotott kép és egy valós világbeli fotó között még rengeteg eltérést találhatunk. Ezek legtöbbje a fényképezőgép lencséinek fizikai mivoltából következik. Ha az élethű képalkotás a célunk, nem árt ezeket az optikai jelenségeket is utánoznunk.

Az első ilyen szembetűnő tényező a tárgyak élén vehető észre. Mivel az elkészült kép egy rasztertáblában tárolódik és a monitoron ennek a táblának az elemei egy az egyben egy-egy képpontnak felelnek meg, ezért a megjelenített képen a tárgyak szélei élesek, durvák és recések (*aliased*) lesznek (9. ábra).



9. ábra: Élsimítás nélkül

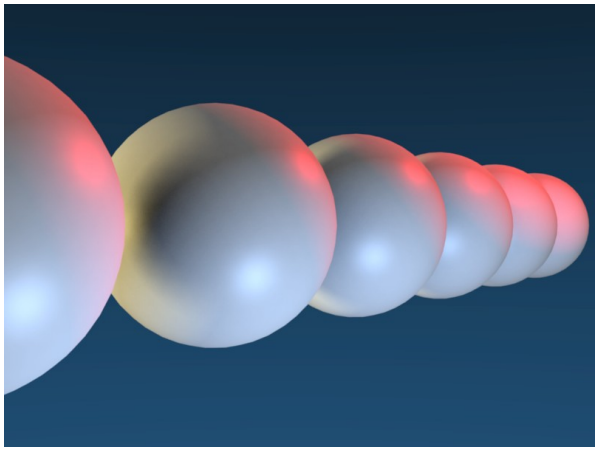


10. ábra: Élsimítással

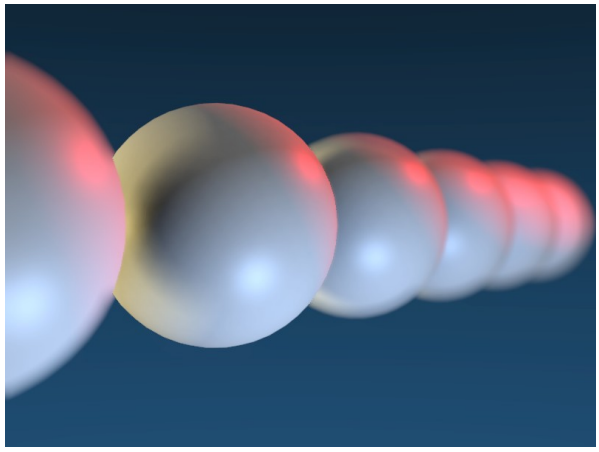
Ennek a problémának a legegyszerűbb orvoslása az elkészülő kép felbontásának növelésével érhető el, ez azonban nem lépheti túl a megjelenítőeszköz felbontóképességét. Jobb hatást érhetünk el úgy, ha képalkotásnál a sugarak indításakor egy képponthoz nem egy sugarat rendelünk hozzá, hanem többet. Ezt tehetjük a képpont egyszerű rács menti továbbosztásával, véletlenszerű eloszlással, vagy egy előre meghatározott eloszlásfüggvénnyel. Az így kapott szubpixeleket átlagolva (súlyozatlanul vagy akár súlyozottan) meghatározhatjuk a fő képpont végső színét. Ennek a módszernek a segítségével

a tárgyaink széle nem lesz annyira szembetűnően éles. Ezt a módszert *élsimításnak* (*antialiasing*) hívják (10. ábra).

A szimulált világgal ellentétben az életben egy kamera használatakor a készülékbe jutó fény nem egyetlen ponton halad át, hanem egy kiterjedéssel rendelkező területen. Ennek hatására azok a tárgyak, amelyek nem esnek a fókuszálás síkjába, homályosabbak lesznek (12. ábra). Ezt a jelenséget nem csak a realisztikus hatás javítása érdekében célszerű felhasználni, de a kép esztétikájának növelése és a jelenet témájának kiemelése érdekében is.



11. ábra: Mélységéletlenség nélkül

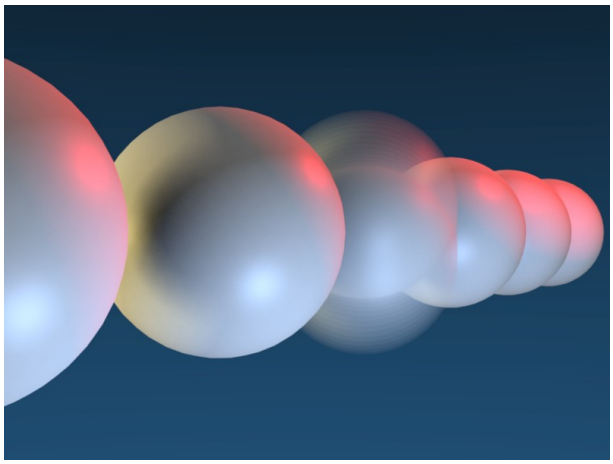


12. ábra: Mélységéletlenséggel

Sugárkövetéses algoritmusnál ezt úgy érhetjük el, hogy a jelenetről több különböző képet készítünk, majd ezeket átlagoljuk. A képek készítése előtt a nézőpontot és a vetítés síkját el kell forgatni a fókuszpont, tehát egyben a jelenet témája körül. Minél nagyobb mértékben változtatjuk meg a képek között a nézőpont helyét és irányát, annál nagyobb lesz a fókuszon kívül eső tárgyak *mélységéletlensége* (ez megfelel a fényképezőgépek blendéjének növelésének). Ennek az eljárásnak *mélységélesség*, angolul *depth of field* a neve. Mivel ez egy elég költséges eljárás, ezért gyakran utólagos képfeldolgozással szokták utánozni a hatását úgy, hogy csak egy képet renderelnek, majd a képpontokra egy Gauss-elmosást alkalmaznak a képpontok z-értéke és a fókusztávolság alapján (természetesen elmosáskor a szomszédos, de térben egymástól nagyon távol elhelyezkedő képpontokat nem mossák össze).

Másik, nagyon hasonló jelenség a hosszú expozíciós időből adódó gyors tárgyak elmosódása. Mivel rendereléskor a jelenet egyetlen, végtelenül rövid pillanatát készítjük el, így a hagyományos képalkotásnál minden tárgy egyenletesen éles, egyik sem kelt

mozgásérzetet ezáltal. Ha a gyorsan mozgó tárgyakat elmosódottnak szeretnénk ábrázolni (13. ábra), akkor nem egyetlen időpillanatot kell leképeznünk, hanem egy időintervallumot. Ezt természetesen csak diszkrét időpillanatok egymás utáni renderelésével érhetjük el, amelyeknek az eredményét átlagolni kell. Mivel az egyes tárgyak a különböző időpillanatokban különböző pozíciókat vehetnek fel, ezért a mozgó tárgyak homályosabbak lesznek a képen az álló tárgyaknál. Természetesen ezt a hatást is el lehet érni utólagos képfeldolgozással, de ugyan úgy, mint a mélységélességnél, itt is pontosabb képet kapunk, ha ténylegesen a tárgyakat (vagy a kamerát) mozgatjuk el, és átlagoljuk az így kapott képeket.



13. ábra: Elmosódás

1.4. Az animálhatóság elérése

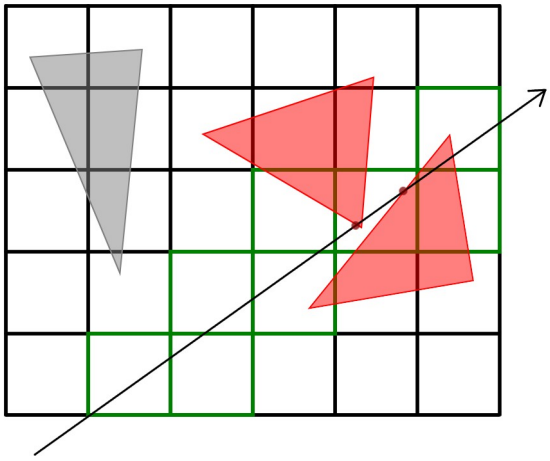
A sugárkövetés azonban már önmagában is jelentős igénybevételnek teszi ki a számítógépet, még ha a fentebb említett technikákat akár mellőzzük is. Ahhoz, hogy a renderelés időigényét csökkentsük, bizonyos előkészítéseket kell végeznünk a megjelenítendő jeleneten. Mivel a metszéspontszámítás a legköltségesebb eleme a sugárkövetésnek és egy naiv sugárkövető rendszer az összes sugár esetén az összes térbeli objektumra nézve kiszámolja azok metszéspontját, ezért mindenféleképpen érdemes a metszéspontok számítását csökkenteni azáltal, hogy az egyértelműen kizárható objektumokkal nem számolunk metszéspontot. Ehhez a modellezett jelenetet fel kell osztani kisebb részekre. Erre a felosztásra két fő irányzatát létezik:

- *Térfelosztás* (*space partitioning*), melynél az objektumteret osztjuk fel úgy, hogy

megadjuk, a tér mely részein mely objektumok foglalnak helyet. Ennek legalapvetőbb formája a *bináris térfelosztású* (*binary space partitioning*) fa használata, amely vágósíkokat helyez az objektumtérre, ezáltal osztva mindig két részre a teret. Ez egy eléggé általános módszer, mivel a vágósíkok bármilyen irányban elhelyezkedhetnek. Éppen ezért többnyire csak inkrementális képalkotásnál szokták alkalmazni. Ennek egy speciális változatát, a *kd-fát*, ahol a vágósíkok merőlegesek a tengelyekre, már a sugárkövetésnél is lehet alkalmazni. Térfelosztású módszernek számít továbbá a *nyolcasfa* (octree) is, amely nem két, hanem nyolc (egyenlő méretű) részre osztja a teret egy lépés során. Ezeket a módszerek mind rekurzívan kell ismételni addig, amíg el nem értünk egy optimális felosztást.

Szintén ide sorolható a sugárkövetést alkalmazó programokban talán leggyakrabban alkalmazott térfelosztású módszer, az *egységrács* (*uniform grid*) is, amely a jelenet köré helyezett, lehető legkisebb méretű téglatestet osztja fel azonos méretű cellákra. A felépített rácsban a következő algoritmussal[5] lehet megtalálni a sugárhoz legközelebb eső metszéspontot:

1. Azonosítani kell a sugár kiindulási helyének celláját (vagy, ha a téglatesten kívül helyezkedik el ez a pont, akkor a téglatestbe történő belépési pont celláját kell megkeresni).
2. Ha ez a cella tartalmaz objektumot, akkor meg kell nézni a sugár ezen tárgyakkal vett metszéspontjait. Ha van ilyen, akkor ezek közül a legközelebbit kell kiválasztani és megtaláltuk a megfelelő metszéspontot.
3. Amennyiben nincs metszéspont, a kétdimenziós képalkotásban használatos *digitális differenciaelemző* (DDA) algoritmus háromdimenziós változatával tovább lépünk a sugár mentén a következő nem üres cellába. Ezt a lépést addig folytatjuk, míg nem találunk egy olyan cellát, amely nem üres. Ha találtunk ilyet, a metszéspontkeresés lépésénél folytatjuk az algoritmust. Ha nem találtunk, mert elértük a befoglaló téglatest határát, akkor nincs metszéspont.



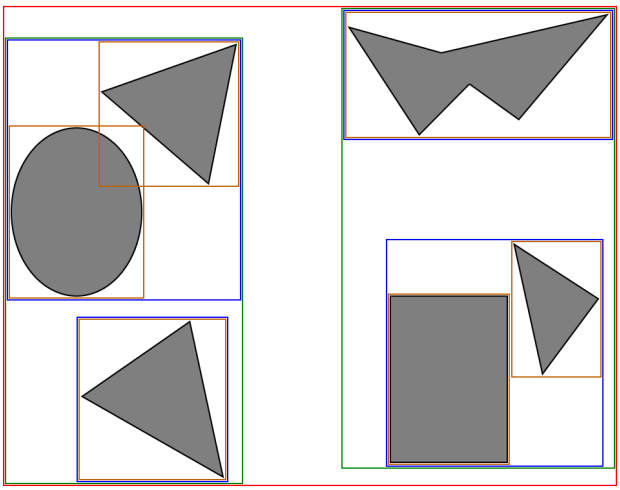
14. ábra: Egységrács bejárása

Az algoritmus így azonban még abban az esetben, ha az aktuális cellában talált objektumot, és annak meg is találta a metszéspontját a sugárral, de az nem ugyan ebben a cellában helyezkedik el, nem ad jó eredményt. Ezért az olyan metszéspontokat, amelyeknek a sugártól vett távolsága nagyobb, mint az aktuális cella maximális távolsága a sugártól, el kell hagyni.

- A *befoglalótestek* (*bounding volumes*) jelentik a modellezett világ felosztásának másik lehetőségét. Itt az előbbiekkel szemben nem a teret osztjuk fel részekre, hanem a jelenetben résztvevő objektumok listáját. Ehhez nem csak az összes tárgy köré kell befoglalótestet (téglatestet vagy gömböt) emelni, hanem az egyes objektumcsoportok, objektumok, vagy objektumokat alkotó elemek köré is. A sugarak indításakor először azt nézzük meg, hogy az egyes befoglalótesteket érintik-e egyáltalán. Amennyiben nem, akkor még a metszéspontok kiszámítása előtt el lehet dobni az objektumot. Szintén gyorsíthatjuk ezt a fajta keresési módot, ha az objektumok befoglalóját a sugár irányának megfelelően az egyik tengely mentén rendezve nézzük végig. Ekkor, ha már találtunk metszéspontot, nem kell az e pont mögött elhelyezkedő objektumokat megvizsgálni, azonban ügyelni kell az egymást metsző befoglalókra, mert ezeknél lehet, hogy egy megtalált metszéspont után egy másik objektum közelebbi metszésponttal rendelkezik.

  A befoglalótesteket *hierarchiába* is rendezhetjük (*bounding volume hierarchy*),

amelyekkel tovább növelhetjük az algoritmus gyorsaságát azáltal, hogy ha egy objektumcsoport közös befoglalóját nem metszi az aktuális sugár, akkor biztos, hogy a benne lévő összes tárgyat egyaránt elhagyhatjuk.



15. ábra: Befoglalótestek hierarchiája

A nem látható tárgyak mellett azonban kiválogathatjuk a kamerának hátat fordító lapokat is. Ezzel a *hátsó lap eldobással* (*back-face culling*) közel a lapok felét kihagyhatjuk a számolásból. Ehhez mindössze a sugár irányvektorának és a lap normálvektorának skaláris szorzatát kell megnéznünk, hogy pozitív-e, mert ha az, akkor a felület nem a kamera felé fordul.

A nézőpont és a lapok irányának kapcsolatán alapuló hátsó lap eldobás analógiájára még egy optimalizálást végezhetünk az árnyalás során: a lapok irányát a fények pozíciójával is össze kell vetni. A saját magukra önárnyékot vető tárgyaknál ugyanis teljesen felesleges a vetett árnyékhoz használatos árnyéksugarat is elindítanunk, mivel a felületek azon része, amelyek önárnyékba borulnak, már úgysem lesznek jobban megvilágítva a vetett árnyékok kiszámolása után sem. Ez gyakorlatilag a fényforrást nézőpontnak véve felel meg egy hátsó lap eldobásnak.

A képalkotást ezeknek a fázisoknak a beépítésével nem csak állóképek esetén gyorsíthatjuk, de ha elérjük a másodpercenkénti körülbelül huszonöt kép alkotásának lehetőségét, akkor már animáció megjelenítésére is használhatjuk a sugárkövetéses algoritmust. Ehhez azonban a megfelelően gyors szoftver mögé megfelelő képességekkel rendelkező hardver is szükségeltetik.

2. Egy korszerű videokártya képességei

Az első sugárkövetéses algoritmusok megvalósításától napjainkig hosszú utat járt be a technológia. Az akkor elkészített programok a legegyszerűbb grafikát is hosszú percek vagy órák alatt produkálták, míg ma ezeket másodpercek alatt el tudja készíteni bármelyik személyi számítógép processzora. Az algoritmusok hatékonysága folyamatosan javult, az őket futtató processzorok képességei és sebességei is folyamatosan nőttek. Ez a tendencia folyamatosan így haladt tovább, egészen a 2000-es évek első feléig, amikor is első körben nyílt meg a lehetőség arra, hogy a sugárkövetéses algoritmusok – a háromdimenziós inkrementális képalkotáshoz hasonlóan, bár azoktól jó pár évvel lemaradva – ne csak a központi feldolgozóegységen futhassanak.

2.1. A kötött funkcionalitású csővezeték

Bár a háromdimenziós gyorsítást támogató külső grafikuskártyák már a kilencvenes években elterjedtek voltak, ezeket az eszközöket a ma is hagyományosnak mondható inkrementális képalkotásra tervezték, sugárkövetésre képtelenek voltak. Ahogy fejlődtek a gyorsítókártyák, úgy léptek fel egyre nagyobb igények velük szemben. A videokártyák azonban csak az előre beléjük égetett műveleteket voltak képesek elvégezni a *kötött funkcionalitású csővezetéken* (*fixed function pipeline*) keresztül. Ezek a műveletek együttes nevükön a T&L (*transform and lighting* – transzformáció és megvilágítás) műveleteket takarják, amelyek magukba foglalták a megjelenítendő alakzatok primitívekre (pontok, vonalak, háromszögek) bontását, transzformálását (az objektumvilág koordinátáiból a megjelenítési ablak koordinátáiba mozgatást és forgatást) és vágását, a csúcspontokon történő fényerő és szín számítását az előre megadott fényforrások alapján, majd a csúcspontok interpolálásával kapott raszterpontok előre megadott módosítását (vágás, színkeverés) és végül megjelenítését. A folyamatos igények miatt azonban egyre gyakrabban merültek fel igények az így megvalósítható Gouraud-árnyalásnál bonyolultabb árnyalási módokra és transzformálási műveletekre, amelyeket a grafikuskártya-gyártók már nem voltak hajlandóak előre belekódolni a vezérlőegységükbe, inkább szabad kezet adtak a programozóknak azáltal, hogy programozhatóvá tették a grafikus feldolgozóegységeket (GPU).

2.2. A programozható grafikus csővezeték

A programokat futtatni képes videokártyák első körben természetesen csak egy az assembly-hez hasonló, alacsony szintű nyelvet értelmeztek, mely ugyan eléggé behatárolta a programok képességeit, mégis nagy lépésnek számított a csupán paraméteresen állítható elődeikhez képest. Nem kellett azonban sokat várni a magasabb szintű, C-hez hasonlatos nyelvek megjelenéséig. Ezeknek a nyelveknek a megjelenésével könnyebbé vált a programozás és átláthatóbbá vált a kód maga. Természetesen a C-hez hasonlatosság nem jelenti azt, hogy ugyan azokat az utasításokat ismerik, a hardverek és ezáltal a grafikus nyelvek utasításkészlete azonban folyamatosan bővül és ez vezetett el oda, hogy korábbi változataikkal szemben mára már ismerik az olyan bonyolultabb programnyelvi elemeket is, mint az elágazások és ciklusok, valamint képesek 32 bites lebegőpontos számokat is használni. Ezek a képességek és az, hogy évről évre nagyobb méretű programokat képesek értelmezni és futtatni a GPU-k, ahhoz vezetett, hogy a korábban tervezett GPU-n futó sugárkövetéses algoritmusok terveivel szemben[6], ma már nem kell több különálló programot írnunk erre az egy feladatra, hiszen egyetlen programmal is képesek lehetünk azt végrehajtani.

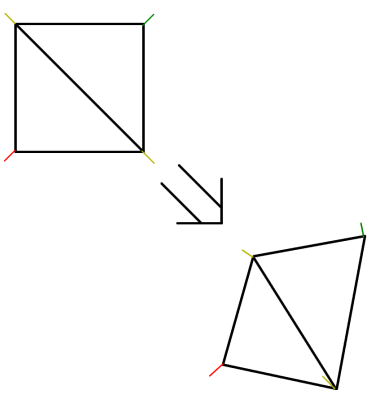
Első gondolatunk talán rögtön az lehetne ezzel kapcsolatban, hogy megkérdőjelezzük: van-e ennek előnye a központi processzoron történő számolással szemben, hiszen hiába fejlődnek olyan gyorsan a videokártyák, ugyan ez elmondható a központi processzorokról is, ráadásul azok előnyben is vannak sebesség területén a grafikus vezérlőkkel szemben. Ennek ellenére egy dologban mégis jobbak a grafikuskártyák: párhuzamos feldolgozásban. Egy mai átlagos videokártya 32-128 shader processzorral rendelkezik, de akadnak már 480 processzorral rendelkezők is. Ezek az egységek felelnek az árnyalóprogramok futásáért. Önmagában egy-egy ilyen egység nem sokat jelent, mert sebesség tekintetében hátrányt szenvednek a CPU-kkal szemben, mivel azonban igen sok van belőlük egy kártyán, együtt igen hatékonyak tudnak lenni akár egy többmagos központi processzorral szemben is és ezáltal az *SIMD* (*simple instruction, multiple data* – egy művelet több adaton történő alkalmazása) képességgel potenciális hardvert jelentenek a sugárkövetéses algoritmusok számára.

Egy videokártyára írt program futása, azonban feladata ellátása érdekében nem annyiból áll, hogy előre lefordítjuk a program forrását, majd egy időpillanatban elindítjuk a kész

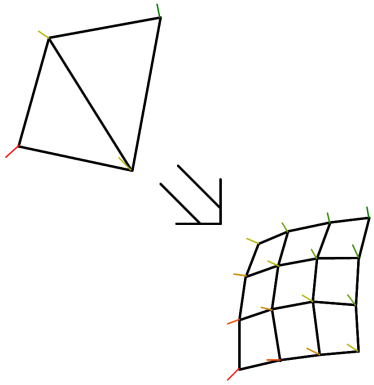
programot és egy másik időpillanatban kapunk tőle egy eredményt. Mivel az inkrementális komputergrafika lehetőségeinek bővítésére tervezték ezeket, így azoknak megfelelő feladatköröket képesek ellátni a grafikuskártyán, valamint a hardverkülönbségek miatt minden alkalommal, amikor használni akarjuk ezeket a programokat, a videokártyára kell bíznunk a forráskód lefordítását. Ezeket a programokat feladatuk szerint tehát egy-egy csoportba sorolhatjuk, amely csoportok mind más és más feladatot képesek ellátni. Ezek a programtípusok a következők lehetnek:

- *csúcspont árnyaló* (*vertex shader*),
- *geometria árnyaló* (*geometry shader*),
- *képpont árnyaló* (*pixel/fragment shader*).

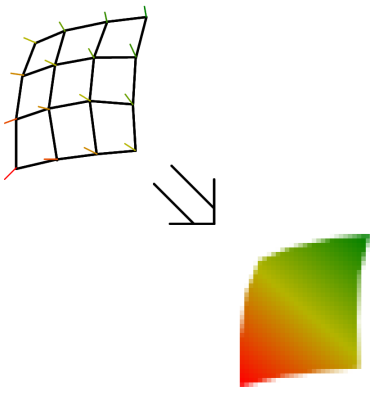
Mindhárom kategória neve egyben utal a feladatára is. Amint egy megjelenítésre szánt alakzatot az összes csúcsának pozíciójával, valamint összes többi attribútumával együtt átküldtünk a CPU-ról a GPU-ra, az éppen aktív csúcspont árnyaló program fut le az összes csúcson, ezáltal módosítva azok helyzetét vagy egyéb jellemzőjét. A viszonylag újnak számító geometriai árnyaló a poligonok primitívekre történő lebontása után fut le a keletkező alakzatokon, így adva lehetőséget arra, hogy a primitíveket tovább bonthassuk, ezáltal növelve a megjelenítendő tárgyak részletességét. A képpont árnyalóval pedig a primitívek raszterizálása után az egyes képpontok színét befolyásolhatjuk.



16. ábra: Csúcspont árnyaló



17. ábra: Geometria árnyaló



18. ábra: Képpont árnyaló

A grafikus kártyák magas szintű nyelven történő programozására három nyelvet különböztethetünk meg:

- GLSL (OpenGL Shading Language): Az OpenGL nyílt, többplatformos 3D-s grafikus könyvtár által nyújtott nyelv.
- HLSL (Hight Level Shader Language): A Microsoft DirectX saját nyelvezete.
- Cg (C for Graphics): Az Nvidia grafikus vezérlők által támogatott nyelv.

Bár ezeket a nyelveket, ahogy azt a nevük is mutatja (*shader language* – árnyaló nyelv), első sorban árnyalásra találták ki, gyakorlatilag igen sokrétű alkalmazásuk lehetséges, akár a grafika területein kívül is. Éppen ezért egyáltalán nem kell különleges dolognak tekinteni, ha a már hangfeldolgozásra és kódtörésre is alkalmas GPU-n sugárkövetést szeretnénk alkalmazni.

A sokrétű használatra alkalmas grafikus hardvert azonban, ha az inkrementális képalkotás helyett más feladatra akarjuk használni, akkor már nem olyan egyértelmű, hogy az általunk kiszemelt feladatot miképpen tudná ellátni az. Éppen ezért az ilyen algoritmusokat át kell alakítani a scanline algoritmus szemléletében olyan utasítássorozatokká, amelyeket a három programtípus (csúcspont-, geometria- vagy képpont árnyaló) közül valamelyik képes ellátni.

2.2. Árnyalók alkalmazása sugárkövetéses képalkotáshoz

Amint azt láthattuk, egy speciális feladatot ellátó programot átültetni a grafikus kártyára nem triviális. Egy sugárkövetéses algoritmus implementálásánál azonban viszonylag szerencsés helyzetben vagyunk, mert a végső soron megjelenítendő képpontokhoz pontosan egy-egy sugarat szeretnénk társítani és, mint azt tudjuk, a képpont árnyaló szintén egy-egy képponton fog lefutni. Ez alapján leszögezhetjük, hogy a sugárkövetés lényegi részét a képpont árnyalónak kell elvégeznie, mivel ez az a programtípus, amely minden egyes pixelt külön tud értelmezni. Ahhoz azonban, hogy egy képpontárnyaló a program ablakának összes pontján lefusson, egy olyan alakzatot kell a képernyőre rajzolni, amely az egész ablakot lefedi. Ehhez elegendő egy téglalapot a képernyő teljes felületére felfeszíteni.

Ahhoz azonban, hogy az egyes képpontokon el tudjunk indítani egy-egy sugarat, tudnunk kell a sugarak indulási helyét és irányát. Ehhez kell használni a csúcspont árnyalót, amellyel elegendő a csúcspontokon kiszámolni a sugarak irányát (a nézőpont helye közös minden sugárnál), majd a grafikus processzor magától interpolálni fogja ezeket az irányokat a csúcspontok között, így kapjuk meg képpontonként a megfelelő irányt.

Mivel a programkönyvtárak csak a csúcspont- és képpont árnyaló jelenlétét igénylik (amennyiben a programozható csővezetéket használjuk egyáltalán) és a geometria árnyalót csak opcionálisnak tekintik, ezért erre a feladatra nem kell programot írni a sugárkövetéshez.

2.3. A valósidejűség biztosítása

A grafikus egységről tehát láthattuk, hogy egyáltalán nem ugyanúgy hajtja végre a programokat, mint a központi processzor. A kártya célspecifikus mivoltából adódóan azonban nem csak a programok végrehajtásának módja és az utasításkészlet különbözik, hanem a hardver felépítése is. Erre legjobb példa a párhuzamos jellegből adódóan az, hogy a grafikus processzorokat rövid számolásokra készítették fel és nem hosszabb kalkulálásokra. Ennek eredménye egy adatfolyam-processzor, amely nem rendelkezik sem veremmel, sem gyorsítótárral. Következtetésképp az algoritmus használata során nem tudunk rekurziót alkalmazni és fa adatszerkezeteket bejárni, vagy csak ezek saját, úgymond szoftveres megvalósításával tudnák elérni ezeket a lehetőségeket. Ahhoz azonban, hogy ténylegesen valósidejű képalkotást érhessünk el az algoritmus alkalmazása során, ezeket a módszereket mellőznünk kell.

Természetesen figyelembe kell venni a grafikus programok utasításkészletét azon szempontból is, hogy mely utasítások a legidőigényesebbek. Ez általában az adatelérést takarja, tehát törekednünk kell a jelenet elemeinek vizsgálatát minimálisra csökkenteni. Emiatt az algoritmus implementálása során a befoglalótestek alkalmazása előnyben részesül az egységráccsal szemben, mert ez utóbbinál nem csak az egyes objektumokat kell időnként elérni, de a sugár követése során rengeteg, akár üres cellát is. A befoglalótesteket azonban nem tudjuk hierarchiába rendezni rekurzió nélkül, tehát a jelenetekben résztvevő tárgyak száma és részletessége is bizonyos határok közé szorul.

A gyorsítótár hiánya is következményeket von maga után. A központi processzorokon futó hagyományos programoknál megszokott módszerekkel szemben, itt némely esetben teljesen más szemléletmódot igényel a programalkotás. A hétköznapi programoknál megtanult optimalizálási módszerek egy része nem érvényes a GPU-programokra. Ilyen például a gyakran használt változók eltárolása. Ez egy meglepő és szokatlan tulajdonság, de a forráskódok írása és optimalizálása során több helyen is igazolta magát. Hiába osztunk ugyan azzal a számmal akár három-négy helyen is, ha a hányadost előre eltároljuk egy lebegőpontos

változóban, azzal sokkal nagyobb terhelésnek tesszük ki a grafikus egységet az adat tárolása miatt, mint azzal, ha minden egyes alkalommal osztást alkalmaznánk szorzás helyett. Vektoroknál ez a vonás még élesebben érzékelhető. Éppen ezért, azok a gyorsítási módszerek, amelyek a CPU-nál már megszokottá váltak, itt lehet éppen, hogy mellőzendőek.

Egy lokális illuminációs Whitted-féle sugárkövető rendszer azonban implementálható, bizonyos megszorítások mellett. A rekurzió hanyagolása miatt nem alkalmazhatunk tükröződést és áttetszőséget egyszerre egy felületre (közvetlen vetett árnyékokat azonban igen, mert ezeknek a sugaraknak elegendő legfeljebb egy lépést megtenniük), valamint iteratív módon kell a többszörös sugarak követését implementálni. Ezt könnyedén megtehetjük úgy, hogy a sugarak által összegyűjtött fényt nem hátulról előrefelé összegezzük, hanem az elsődleges sugár színéhez adjuk hozzá a másodlagosét, harmadlagosét, és sorban a többiét is. Hogy a végső szín hány százalékát teszi ki a következő sugár, azt könnyedén megállapíthatjuk az eddig érintett felületek tükröződési együtthatójának szorzatából[7]. Az iteráció n lépés alatt tehát a következő színt kell, hogy adja:

$$c(n) = \sum_{j=1}^{n} c_j \left(\prod_{i=1}^{j-1} r_i\right) R_j ,$$

ahol c_j a j-edik felületből nyert szín, r_i pedig az i-edik felület tükröződési együtthatója, továbbá $R_j = 1 - r_j$, amennyiben $j < n$, egyébként $R_j = r_n$. Erre a feltételes együtthatóra azért van szükség, mert csak így biztosíthatjuk, hogy az n-edik iteráció során érintett tárgyból érkező fény teljes egészét figyelembe vesszük, tehát úgy tekintjük, mintha az utolsó lépésben egyik tárgy sem lehet már tükörfelület. Az utolsó felületet is vehetnénk tükröződőnek, a benne megjelenítendő tükörkép elhagyásával, de ekkor ez a felület valamivel sötétebbnek tűnne a kelleténél, ezért célszerű ezt a képletet alkalmazni.

Ezeket a feltételeket figyelembe véve, a megfelelő, korszerű grafikus hardveren implementálhatóvá válik egy valósidejű sugárkövetéses renderelő program.

3. Megvalósítás

Az algoritmus CPU-n futó részének implementálása C++ nyelven, míg GPU-n futó felének megalkotása GLSL nyelven történt. Ez utóbbira azért esett a választás, mert a HLSL nyelvvel szemben platformfüggetlen, a Cg nyelvvel szemben pedig hardverfüggetlen. A program jól szemlélteti a korszerű grafikus kártyák képességeit, ugyanakkor épp ezért bizonyos minimális követelményei is adódnak. Hardver szempontjából a videokártyának legalább ATI (AMD) Radeon HD 2xxx kategóriájúnak, vagy Nvidia GeForce 8xxx típusúnak kell lennie, mert ezek támogatták először az OpenGL 3.0-ás, valamint a GLSL 1.30-as verzióját, amelyeket a program is igényel. Az algoritmus implementálása gyakorlatilag ezeknél régebb verziójú API-kkal is lehetséges lenne, de ezek az újabb verziók hozták be a lebegőpontos textúrák és a bitenkénti operátorok alkalmazásának lehetőségét, amelyeket alkalmazva a program hatékonyabb működésre tehet szert. A program továbbá használja még a Simple DirectMedia Layer (SDL) platformfüggetlen és nyílt API-t is, amellyel az ablak- és inputkezelést egyszerűsíthetjük le.

A kész szoftver ezeken, vagy az ezeknél jobb grafikus meghajtókártyákon képes valósidejű sugárkövetéses képalkotást végrehajtani. A programban implementálva lett a Blinn-Phong árnyalás, valamint az elsődleges sugarak követése mellett az elsődleges árnyéksugarak és a bármilyen szintű tükörsugarak követésének képessége, mindez egy háromszögekből *felületmodellezéssel* (*boundary-representation*) felépített tárgyakból álló tetszőleges jelenet keretében. Megvilágításként használhatunk egyaránt napfényt (irányított fényforrás) és lámpafényt (pontszerű fényforrás) is, azonban ezekből a fényforrásokból egy jelenetben összesen egy szerepelhet. Ez a fényforrás lehet állóhelyzetű is, de mozoghat is egy adott pont körül. A centrális vetítéshez használt kamera objektum szintén rendelkezik egy koordinátával, amelyre néz és amely a jelenet tárgyát jelöli ezáltal. Az elhelyezkedése e körül a pont körül lehet állandó, vagy foroghat, esetleg ”imbolyoghat” körülötte, de megadhatjuk a távolságát is a tárgytól, amelyre irányul, valamint a fókusztávolságát is, amellyel a hagyományos, 35mm-es filmeknél használt fókusztávolsággal egyenértékű számmal adhatjuk meg, hogy milyen távol legyen a leképezés síkja a fókuszponttól és ezáltal növelhetjük vagy csökkenthetjük a látószöget. Ezeket a paramétereket egy jelenetleíró fájlban adhatjuk meg, amely egyben megadja a jelenetet felépítő Wavefront Obj típusú modellfájl elérési útját is

(amely pedig egy külső, harmadik fájlra hivatkozik, ami az objektumok anyagát adja meg)[8] [9].

3.1. A központi feldolgozóegység feladata

Bár a központi processzort nem veszi jelentősen igénybe a program, a grafikus programkódot és a jelenetet azonban csak ennek a használatával helyezhetjük a videokártya memóriájába, így ezt a feldolgozóegységet is igénybe kell venni.

A program induláskor a főablak létrehozása előtt betölti a munkakönyvtárában található *raytracer.conf* nevű állományt, amely az ablak futásidejű paramétereit tartalmazza. Ezeket a paramétereket a `CSettings` nevű osztály egy példánya tartalmazza, amely a következő jellemzőket képes betölteni az egyszerű kulcs-érték párokból álló konfigurációs fájlból:

```
defaultScene = <string: a jelenetet leíró fájl helye és neve>
width = <int: ablak szélessége>
height = <int: ablak magassága>
fullscreen = <opcionális, logikai: teljes képernyős ablak legyen-e>
freeze = <opcionális, float: kimerevítendő időpillanat>
maxRayDepth = <int: maximális iterációs lépések száma>
shadow = <logikai: legyenek-e vetett árnyékok>
reflThreshold = <float: a tükröződési küszöbérték, amely alatt el
lehet hagyni a további sugarakat>
farZ = <float: a távoli vágósík távolsága>
```

Amennyiben a programot egy parancssori argumentummal hívjuk meg, úgy ezt a paramétert sztringként felhasználja a betöltendő jelenetet leíró fájl megjelölésére, amennyiben ezt nem adjuk meg, akkor pedig a konfigurációs fájlból deríti ki a jelenetfájl elérési útját.

A program a paraméterek betöltése után létrehoz egy azoknak megfelelő OpenGL renderkontextussal rendelkező `CWindow` osztályú ablakot, amelyben ellenőrzi, hogy teljesülnek-e a program hardverkövetelményei, valamint inicializálja a grafikus könyvtárat kezelő `CRenderer` objektumot, majd belép a főciklusába, amelyben minden ciklusléptés alkalmazásával meghívja a rajzolási eljárást és ellenőrzi, hogy a kilépési feltételek (escape billentyű lenyomása, kilépés szignál érkezése a program felé) teljesülnek-e.

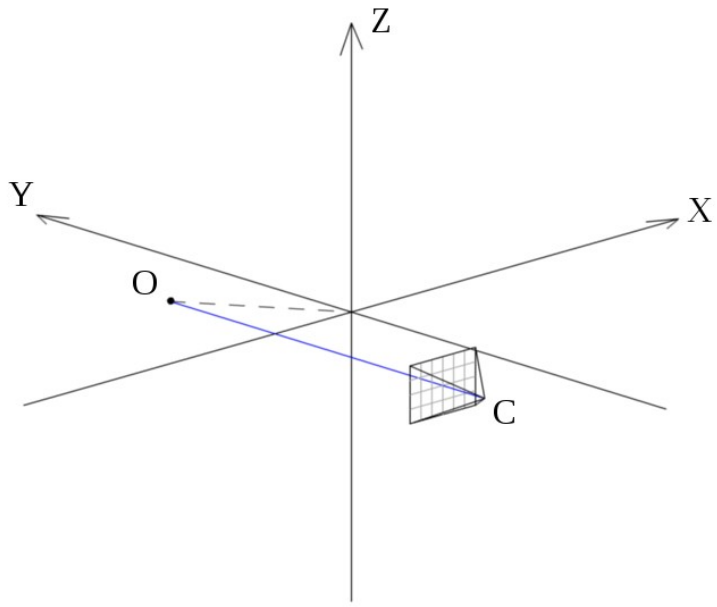
A renderelő osztály létrejöttekor inicializálja az OpenGL környezetet, majd betölti a beállításokban vagy parancssori argumentumban megadott jelenetfájlt egy `CScene` osztály típusú objektumba. A jelenetfájlnak a következő kulcs-érték/értékhalmaz párokat kell tartalmaznia:

```
model = <string: a tárgyakat leíró .obj állomány neve>
focus = <float: a kamera fókusztávolsága>
rotX = const <float: kamera dőlésszöge> |
       wave <float: hullámzás sebessége> <float: dőlésszög> <float:
kilengés mértéke fokban>
rotZ = const <float: kamera forgásszöge> |
       wave <float: hullámzás sebessége> <float: forgássszög>
<float: kilengés mértéke fokban> |
       linear <float: kamera forgási sebessége>
distance = <float: távolság a jelenet tárgyától>
origin = <float x y z: a jelenet tárgyának pozíciója>
light = (lamp | sun) (const | wave)
lightPos = <float x y z: a lámpafény helye/napfény iránya>
# A következő tulajdonságok megadása csak a mozgó (wave)
fényforrások esetén szükséges:
lightSpeed = <float x y z: a mozgás sebessége>
lightAmplitude = <float x y z: a tengelyek menti kilengés mértéke>
```

A kamera tehát az `origin` által meghatározott pontra fog tekinteni. Ezt úgy érhetjük el, hogy a vetítési centrumot első körben erre a pontra helyezzük, a vetítési síkot pedig az Y tengelyre merőlegesen, ettől a ponttól az Y tengely mentén `focus` / 12.0 távolsággal pozitív irányba eltolva. A vetítési síkon felvett ablak magassága két egység objektumtérbeli mértékekkel mérve, szélessége pedig a beállított felbontástól függően $2.0 \cdot ratio$, ahol a *ratio* változó adja meg az ablak vízszintes és függőleges felbontásának arányát (így például 12-es fókusztávolsággal érhetünk el 90°-os függőleges látómezőt). Ezt követően a vetítési centrumot és a vetítési síkot az Y tengely mentén negatív irányban eltoljuk `distance` távolsággal.

A 19. ábrán látható a nézeti modell felállításának első fázisa. Az O pont jelöli az

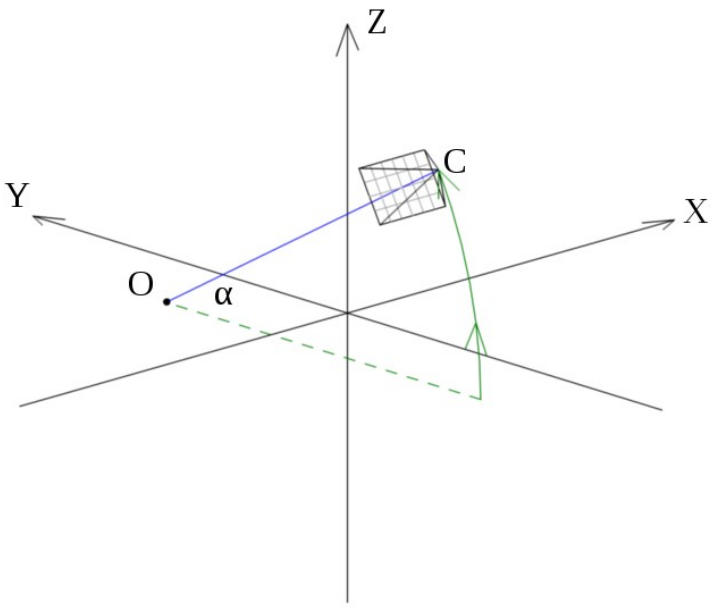
`origin` által megadott nézési célpontot, a C pedig a vetítési centrumot. A két vektor különbségének hossza egyenlő a `distance` paraméter értékével.



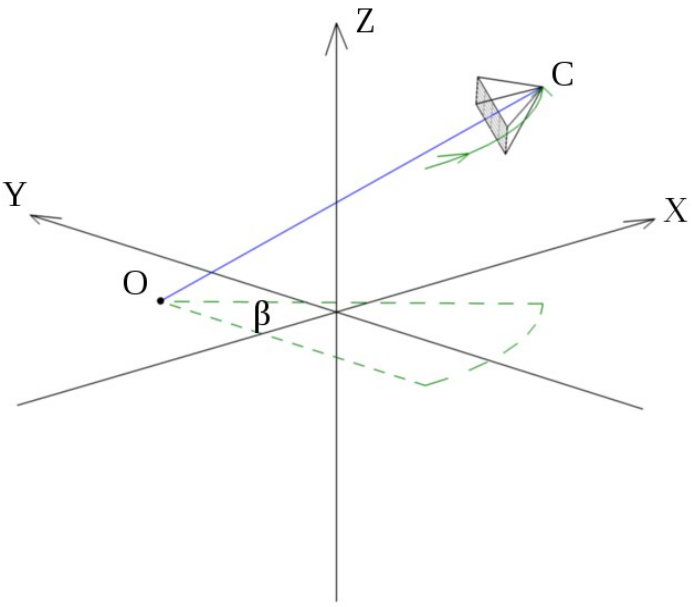
19. ábra: A kamera modell felállítása (1. lépés)

Következő lépésként a vetítési centrumot és a vetítési síkot elforgatjuk az O ponton áthaladó, X tengellyel párhuzamos egyenes körül `rotX` fokkal (a 20. ábrán alfa jelöli). Ezáltal egy bizonyos dőlésszöget adhatunk a kamerának, mely által jobban rá lehet látni a tárgyak tetejére, vagy akár aljára.

Harmadik és egyben utolsó lépésként az O ponton áthaladó, Z tengellyel párhuzamos egyenes körül is elforgatjuk a vetítési centrumot és a vetítési síkot, most azonban `rotZ` fokkal (a 21. ábrán béta jelöli), amellyel a jelenet körüli forgását biztosíthatjuk.



20. ábra: A kamera modell felállítása (2. lépés)



21. ábra: A kamera modell felállítása (3. lépés)

Az állandó pozíciójú fény a `lightPos` paraméter által megadott X, Y, Z koordinátán fog elhelyezkedni, a mozgó fény pedig a következő koordinátán:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \cos(time \cdot lightSpeed_x) \cdot lightAmplitude_x \\ \sin(time \cdot lightSpeed_y) \cdot lightAmplitude_y \\ \cos(time \cdot lightSpeed_z) \cdot lightAmplitude_z \end{pmatrix} + lightPos$$

Mivel a mozgó fény X tengely menti pozíciója az aktuális időtől koszinuszosan, az Y tengely menti pozíciója pedig szinuszosan függ, ezért az azonos X és Y sebességű és amplitúdójú fény egyenletes kör, az azonos sebességű, de eltérő amplitúdójú fény pedig ellipszis alakú mozgást végez (a Z tengely menti vetületet tekintve) a `lightPos` által megadott pont körül.

A program a jelenet után a csúcspont és képpont árnyaló programokat, tehát az algoritmus lényegi részét is betölti. Mielőtt azonban átadná ezeket a videokártyának, az árnyalókban előre elhelyezett speciális makró szövegeket kicseréli a nekik megfelelő értékekre. Ilyen makró formájában kerül rögzítésre az, hogy lesz-e szükségünk tükröződésekre (az iterációk száma nagyobb-e mint egy) és árnyékokra, valamint, hogy a fényforrás milyen típusú. Ezek segítségével az árnyaló program fordítása során további optimalizálást érhetünk el azáltal, hogy a nem kívánt technikákat mellőzzük a programból. Így például a tükrök tiltása esetén az iterációt képező ciklust kihagyva elég egyetlen egy függvényhívást alkalmazni a fő sugár elindításához. Ekkor kerül konstans formájában átadásra az iterációk megengedett száma, a tükröződési küszöbérték és a távoli vágósík elhelyezkedése is. A megfelelő értékek behelyettesítése után az árnyaló lefordítására és videokártya-memóriába feltöltésére kerül sor.

A konstans értékeket tehát már átadtuk a grafikus programnak, a változó értékek átadásához azonban le kell kérdeznünk a grafikus programban elérhető változók helyét, hogy a rajzolás során ezekre a helyekre illeszthessük be az ott elérhető változóknak a megfelelő értékeket. Ez a következő formában történik meg az összes, a csúcspont- és a képpont árnyalóban szereplő változóra egyaránt:

```
GLint valtozoNeve = glGetUniformLocation(program, "valtozoNeve");
```

Ezt követően az inicializálás sikeresnek nevezhető és a program belép a rajzolási fázisba.

3.2. Adatátvitel a videokártya irányába

A videokártya azonban nem fér hozzá sem a központi memóriához, sem a merevlemezhez. Ahhoz, hogy a jelenetet ki tudjuk rajzolni, el kell tárolnunk azt a videomemóriában, amelybe a központi processzorral tölthetjük be az adatokat. Az OpenGL szokásos igénybevétele során többféle módon is küldhetünk tárolásra és később elérésre szánt adatokat a gyorsítókártyára a különböző buffer object-ekkel, például a vertex buffer objecttel is, ezeket azonban nem tudjuk elérni az árnyalókból, mivel értelemszerűen normális esetben nem az árnyalóknak kell ezeket elérniük, hanem ezek megjelenítésekor futnak le rajtuk az árnyalók. Így gyakorlatilag egyetlen lehetőség maradt az adatok átültetésére, mégpedig a textúrák használata. A textúrákat ugyanis el lehet érni az árnyalóprogramokból, így az ezekben elhelyezett adatokat (akár anyagmintázatot írnak le, akár bármi mást) fel tudjuk használni a jelenet kirajzolásához. A tárgyak csúcspontjait, ezekhez rendelt normálvektorait, valamint a csúcsok összekötési sorrendjét meghatározó indexeket tehát a memóriában tárolt egydimenziós tömbből le kell tükrözni a videokártya memóriájába egy-egy szintén egydimenziós textúrába. Mivel a textúrákat elsősorban három vagy négy komponensből álló 24 vagy 32 bites elemekből szokás felépíteni, amelyek, mint pontok az egyes textúrabeli képpontok (texel) R, G és B, azaz vörös, zöld és kék színértékét jelölik, ezért ezeket az elemeket háromdimenziós vektorként alkalmazva egyaránt könnyedén eltárolhatjuk egy-egy lebegőpontos, egydimenziós textúrában a csúcspontok helyét és normálvektorát is. A háromszögeket meghatározó indexeket természetesen elegendő egy egész számok sorozatából álló textúrában eltárolni, ahol az egyes texelek a háromszögek csúcsaihoz rendelt koordinátákat és normálvektorokat tárolják felváltva.

v0.x	v1.x	v2.x	...
v0.y	v1.y	v2.y	...
v0.z	v1.z	v2.z	...

22. ábra: Csúcspontok textúrája

n0.x	n1.x	n2.x	...
n0.y	n1.y	n2.y	...
n0.z	n1.z	n2.z	...

23. ábra: Normálvektorok textúrája

v0	n0	v3	n3	...
v1	n1	v4	n4	...
v2	n2	v5	n5	...

24. ábra: Indexek textúrája

A 22. ábrán látható módon elrendezett indexek tehát úgy határozzák meg a háromszögeket, hogy az első texel három értéke adja az első háromszög három csúcsának

indexét (amelyeket aztán a 23. ábrán szemléltetett csúcspont textúra megfelelő elemeinek kiolvasására használunk fel), a második texel adja ugyan ezen háromszög normálvektorainak indexeit (amiket aztán ugyan így a 24. ábrán látható módon tárolt normálvektor textúra elemeinek elérésére használunk).

A befoglalótesteket hasonló módon tároljuk el a videomemóriában. Egy textúra írja le a befoglaló téglatestek helyét és méretét, két átellenes sarokpont megadásával, egy másik textúra pedig az ezen befoglalótest által körbehatárolt objektum háromszögeinek kezdő- és végindexét, valamint a rajta alkalmazandó anyag indexét alkotja. Az objektumok anyagának tulajdonságait (diffúz szín, spekuláris szín, csúcsfény hatvány értéke, tükröződési együttható) egy hatodik textúrában tároljuk.

b0.x	b0.X	...
b0.y	b0.Y	...
b0.z	b0.Z	...

25. ábra: Befoglalók elhelyezkedését leíró textúra

b0.s	b1.s	...
b0.e	b1.e	...
m0	m1	...

26. ábra: Befoglalók jellemzőinek textúrája

d0.r	s0.r	exp0	...
d0.g	s0.g	ref0	...
d0.b	s0.b		...

27. ábra: Anyagok textúrája

A 25. ábrán szereplő texel-elrendezésből látható, hogy a befoglalótestenként meghatározott objektumonkénti két csúcspontot leíró két texelt hogyan rendezzük el, a 26. ábrán pedig a befoglalók kezdő és végindexét, valamint anyagindexét meghatározó texelek sorrendjét láthatjuk. A 27. ábra szintén a fentebb leírt anyagtulajdonságok textúrában történő elhelyezését mutatja.

Az olyan egyszerű adatok, mint az aktuális időpont, a kamera és a fényforrás állapota azonban nem igénylik textúrák használatát ahhoz, hogy az árnyalóprogramokból megtudhassuk ezek értékét. Ezekhez elegendő, ha az OpenGL által nyújtott CPU és GPU közötti, valamint GPU-n belüli egyszerű adatátviteli lehetőségeket használjuk. Ezt az API három különböző változótípussal oldja meg:

- A *uniform* típusú változók az OpenGL-től a GLSL felé irányuló adatküldést teszik lehetővé. Segítségükkel egész vagy lebegőpontos számokat, esetleg vektorokat küldhetünk át a shader program irányába. Ezek azok a változók, amelyeket inicializáláskor azonosítanunk kellett, hogy a rajzolási cikluson belül értéket adhassunk nekik a `glUniform()` függvénnyel. Ezek az értékek a grafikus programok

futása alatt végig változatlanok, csak a rajzolási ciklus egyes lépéseiben adhatunk nekik különböző értéket. Ezáltal az egész jelenetre érvényes adatok átadására lesznek alkalmasak, tehát az aktuális idő vagy a kamera és a fény helyének közlésére. Ezt a típust egyaránt elérhetjük a csúcspont- és képpont árnyalókból is.

- Az *attribute* változók az egyes primitívek csúcspontjaihoz tartoznak. Minden egyes csúcsponthoz külön értékeket adhatunk meg velük, így a rajzoláskor az ablakra felfeszített téglalap négy sarkában is ezekkel adhatjuk meg, hogy a nézőpontból elindított sugarak ezen csúcsokon át milyen irányba haladnak. Az attribútum típusú változókat értelemszerűen csak a csúcspont árnyalóban érhetjük el.
- A *varying* típusú változók a csúcspont- és a képpont árnyaló közötti egyirányú kommunikációt teszik lehetővé. A csúcspont árnyalóban megadott kimeneti (*out* típusú) varying változókat a grafikus vezérlő interpolálja a raszterizálás során, majd a képpont árnyalónak az új értékeket adja át (*in* típusú varying változóként).

A csúcspont árnyaló tehát uniform változóként megkapja a kamera helyzetét és irányát, majd ezek alapján előállítja a nézőpont és a sugarak objektumtérbeli koordinátáit (mivel az ablakra feszített téglalap kirajzolásakor a Z tengely mentén lefelé néző irányt adtunk meg, amely az OpenGL alapértelmezett nézési iránya, ezért ezt a felülnézeti irányt át kell transzformálnunk oldalirányú nézetre, amely sokkal jobban megfelel a jelenet szemléltetésére). A sugarak irányát ezután interpolálja a csúcspontok között az összes képponton át, amelyet később a képpont árnyaló már fel tud majd használni. Továbbá közölnünk kell még a képpont árnyalóval a fényforrás aktuális pozícióját, valamint a felhasználandó textúrák azonosítóját is egy-egy integer típusú uniformon keresztül.

Az adatok ilyenformán történő megadása után a képalkotás folyamán a csúcspont árnyaló képes lesz létrehozni az induló sugarakat, majd a képpont árnyaló az összes sugárnál meg tudja vizsgálni azok metszéspontjait a jelenetbeli alakzatokkal.

3.3. A grafikus feldolgozóegység feladata

A rajzolási fázis ezen pontjától kezdve az összes sugárról tudjuk, hogy egymástól elkülönülve, párhuzamosan fut le a grafikus vezérlőegységen. Mielőtt azonban a képpont árnyalóval elkezdenénk a sugárkövetést, mindenféleképpen normalizálnunk kell a csúcspont árnyalóból kapott sugár irányvektorát, mert az interpolálás során nem normalizált értéket

kapunk.

A képpont árnyalóban a sugárkövetés fő ciklusa a következő lépésekből áll:

1. Sugár indítása az aktuális pontból az aktuális irányba.
2. Amennyiben nincs találat, az aktuális tükröződési együttható értékével beszorzunk egy előre megadott konstans színt (úgymond az ég színét), majd ezt az értéket hozzáadjuk a végleges színhez és kilépünk a ciklusból.
3. Ha van találat, a kapott pontot árnyaljuk a már korábban bemutatott Blinn-Phong árnyalás szerint.
4. Ha nem vetül önárnyék az adott pontra, akkor sugarat indítunk belőle a fényforrás felé. Amennyiben ez a sugár eltalált egy objektumot a fényforrás eléréséig, akkor feketére állítjuk a felület ezen pontjának színét.
5. Ha elértük a maximális iterációs lépést vagy a megadott tükröződési küszöbérték alá esett az aktuális tükröződési együttható, akkor ez utóbbi együtthatóval beszorozzuk az anyag színét, majd a kapott színt a végső színhez adjuk és kilépünk a ciklusból. Ha a két feltétel egyike sem teljesül, az aktuális tükröződési együtthatót az anyag tükröződési együtthatójának egyből kivont értékével szorozzuk be, majd ezt az egészet szorozzuk össze a felület adott pontján vett színével, amit végül hozzáadunk a végső színhez.
6. Az aktuális tükröződési együtthatóhoz hozzászorozzuk a sugár által elmetszett anyag tükröződési együtthatóját.
7. Az elmetszett felület normálvektorára tükrözzük a sugár irányvektorát, valamint a metszéspontra állítjuk annak helyvektorát.

Ezzel a ciklussal véges sok lépésben megkapunk az összes képpontra egy színt.

Az elindított sugarakat kezelő függvény alapvetően két egymásba ágyazott for ciklusból áll. A külső az összes objektumon halad végig és megnézi, hogy a sugár érinti-e a befoglaló téglatestüket. A téglatestek csúcspontjait a következő GLSL paranccsal deríthetjük ki: `texelFetch(texturaID, texel, 0).rgb`, ahol a `texturaID` a megfelelő textúra azonosítója, a `texel` pedig a textúrából lekérdezendő képpont oszlopszáma. A harmadik paraméter jelen esetben kötelezően nulla, mert a textúrákon belül nem használtunk külön részletességi szinteket. Mint a függvény végén álló komponens szelekcióból látható, a kapott adat mindhárom komponensét fel akarjuk majd használni, így ezt az értéket egy háromelemű

vektorban kell eltárolnunk ezek után.

Azt, hogy egy sugár érint-e egy téglatestet úgy döntjük el, hogy megnézzük, hogy a sugár a téglatest oldalait a sugár kiindulási pontjától a sugáron haladva milyen messze metszi el. Ezeknek a távolság-intervallumoknak – amelyeket a szemközt oldalakon mért értékek alkotnak – vesszük a metszetét, és amennyiben ez szintén egy érvényes intervallum, akkor a sugár érinti a téglatestet.

A belső, az objektumokat alkotó háromszögeken futó ciklus tehát csak azoknál az objektumoknál indul el, amelyeknek a befoglalóját érinti a sugár. Az objektumot alkotó háromszögek kezdő- és végindexét a befoglalók csúcspontjaihoz hasonló módon kérdezzük le. Amennyiben ez a sugár nem elsődleges sugár, megvizsgáljuk, hogy az aktuális háromszög nem az-e, amelyiket az iteráció előző lépésében megtaláltuk, mert ha az, kihagyjuk a vizsgálatból, ezáltal gyorsítjuk is az algoritmust és a lebegőpontos hibákból adódó tükröződési problémákat is kiküszöböljük. Következőnek a háromszög csúcsait és normáljait megadó indexeket kérdezzük le, amelyek alapján a csúcspontok és normálvektorok értékét tartalmazó textúrából megkaphatjuk a háromszög térbeli elhelyezkedését. Alkalmazzuk a hátsó lap eltávolítást, majd amennyiben még mindig metszéspontszámításra esélyes az aktuális háromszög, kiszámoljuk azt. Ha van tényleges metszéspont, és az közelebb esik az összes többi korábbi metszéspontnál, akkor feljegyezzük. Legvégül ezt a feljegyzett metszéspontot adja vissza a függvény.

3.4. Metszéspontszámítás

A sugárkezelés algoritmusa tehát ezáltal kész van, már csak az egyik legkritikusabb pontját, a metszéspontszámítást kell megfelelően implementálni. Erre a feladatra T. Möller és B. Trumbore 2005-ben megjelent publikációjában[10] olvashatunk egy igen hatékony megoldást. A módszer a gyorsaságát abból nyeri, hogy a háromszög síkjának egyenletét nem kell sem kiszámolni, sem tárolni hozzá. Ezzel a módszerrel a következőképpen határozhatjuk meg egy sugár és egy háromszög metszéspontját:

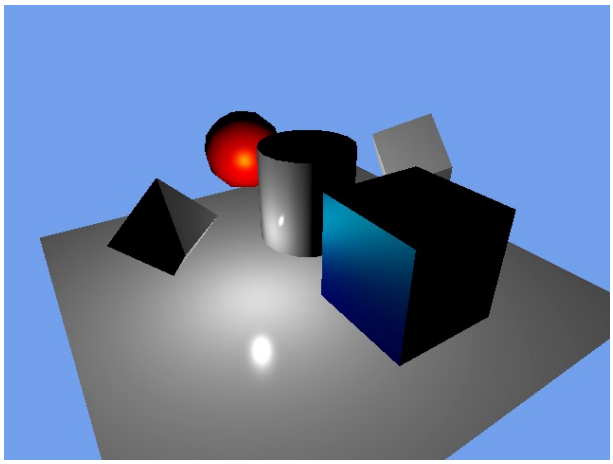
$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix},$$

ahol E_1 és E_2 a háromszög $\vec{AB}$ és $\vec{AC}$ élét alkotó vektor, T a sugár kiindulási pontjának

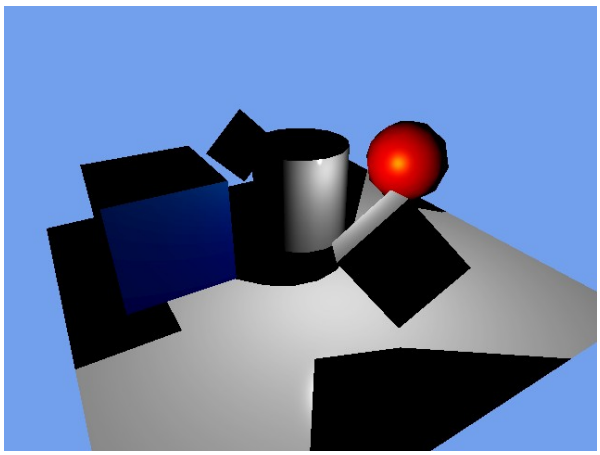
és a háromszög első csúcspontjának különbsége, D pedig a sugár irányvektora. Továbbá $P = D \times E_2$, valamint $Q = T \times E_1$. A kapott számhármasban t a sugár félegyenesén vett távolságot adja meg a kiindulási ponttól a metszéspontig, az u és v értékek pedig a metszéspontot a háromszög baricentrikus koordinátáival adják meg. Ahhoz, hogy eldöntsük, a pont a háromszögön belül van, ennek a két értéknek kell nullánál nagyobbnak lennie, az összegüknek pedig egynél kisebbnek. Mint láthatjuk, ez a módszer mindössze vektorok különbségét, skaláris szorzatát és vektoriális szorzatát tartalmazza, osztást egyáltalán nem, így valóban igen hatékony.

3.5. Eredmények

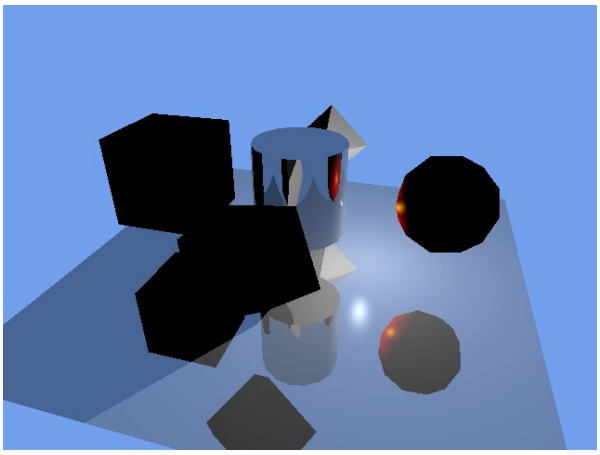
Az implementált algoritmus biztató eredményeket hozott a pár évvel ezelőtti sugárkövetést alkalmazó programokkal szemben. Egy 32 shader processzorral rendelkező Nvidia GeForce 8600GT típusú grafikus vezérlőegységen egy öt-hat egyszerűbb tárgyból álló (kockák, henger, gömb) jelenet az elsődleges sugarak mélységéig egyértelműen képes valós időben futni (28. ábra), árnyékok (29. ábra) vagy egyszeres tükröződések (30. ábra) alkalmazásával pedig 20 képkocka/másodperces sebesség körül képes animálni a jelenetet 640x480-as ablakméret mellett.



28. ábra: Elsődleges sugarak

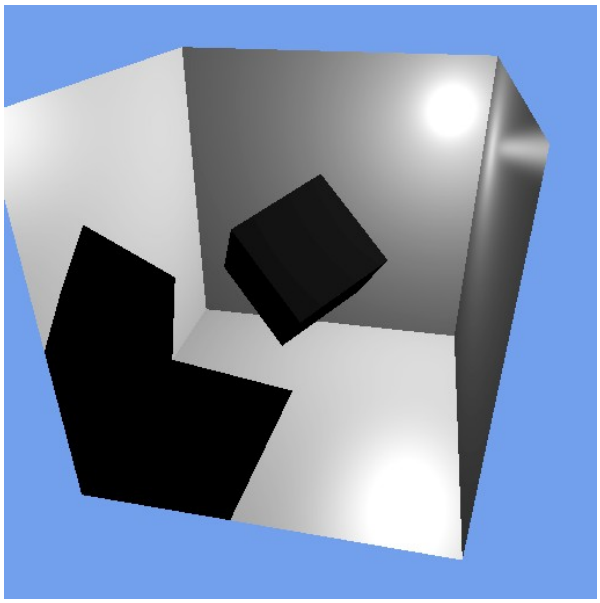


29. ábra: Árnyékok

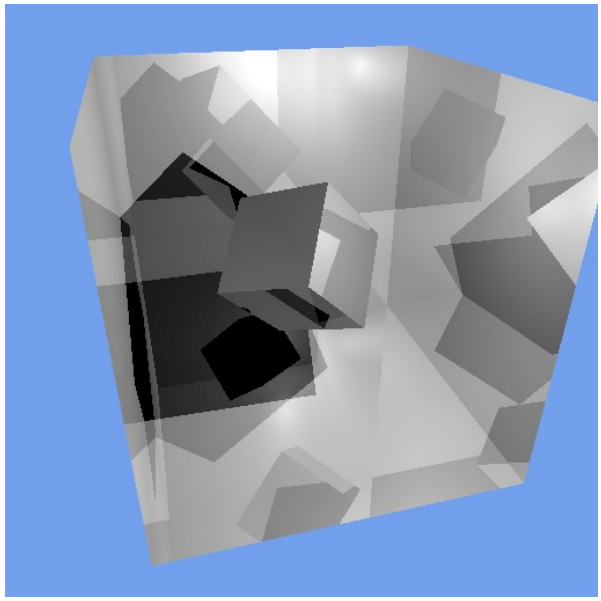


30. ábra: Tükröződés

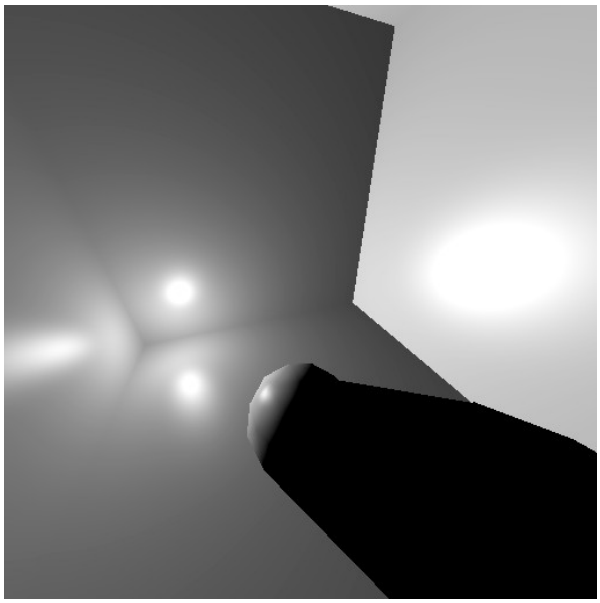
Az ablakméret vagy a jelenetben résztvevő tárgyak/háromszögek csökkentésével határozott mértékben lehet növelni ezt a sebességet. Egy 512x512-es méretű ablakban megjelenő Cornell doboz, egy-két egyszerűbb elemmel már árnyékokkal és/vagy tükröződésekkel (akár többszörössel is) képes ténylegesen valós időben futni.



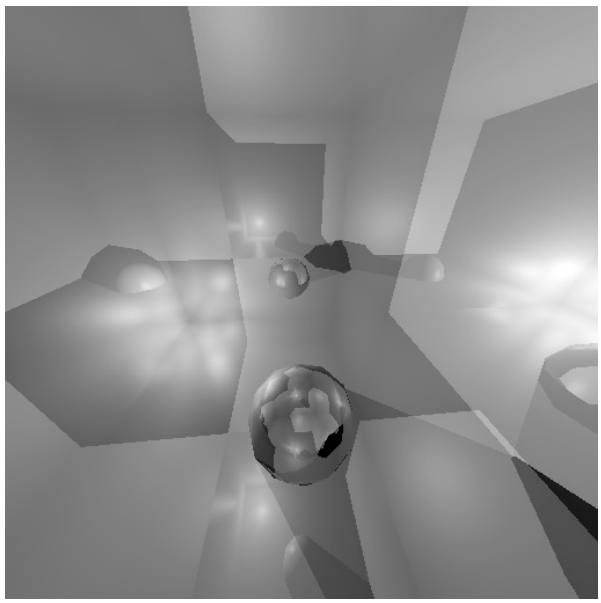
31. ábra: Árnyékok



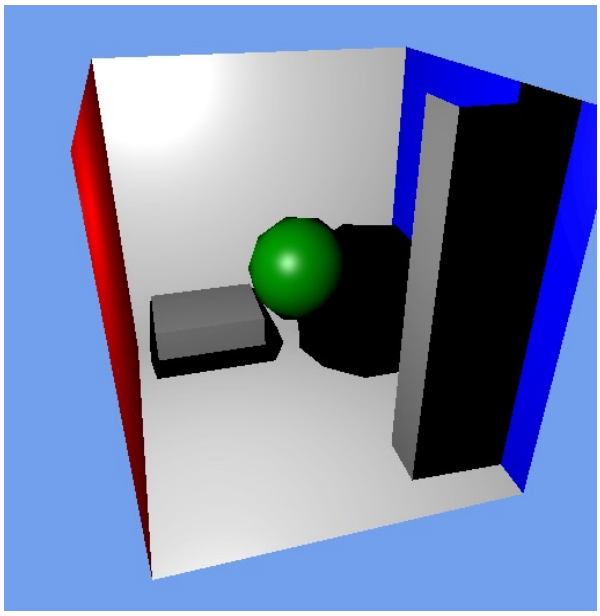
32. ábra: Árnyékok és kétszeres tükröződés



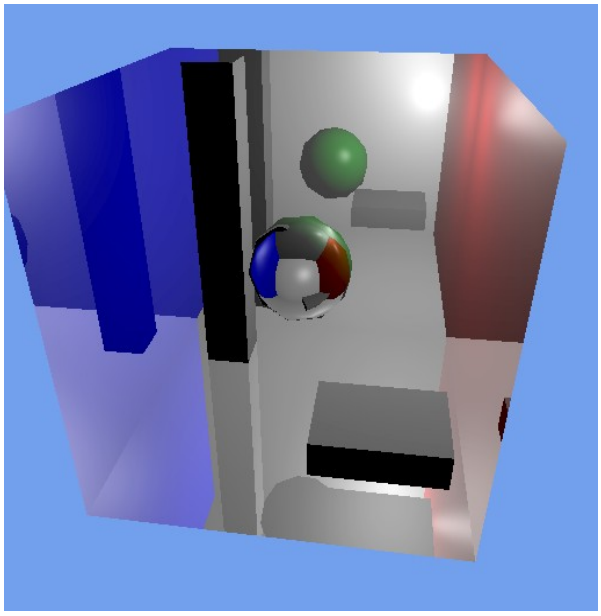
33. ábra: Árnyékok



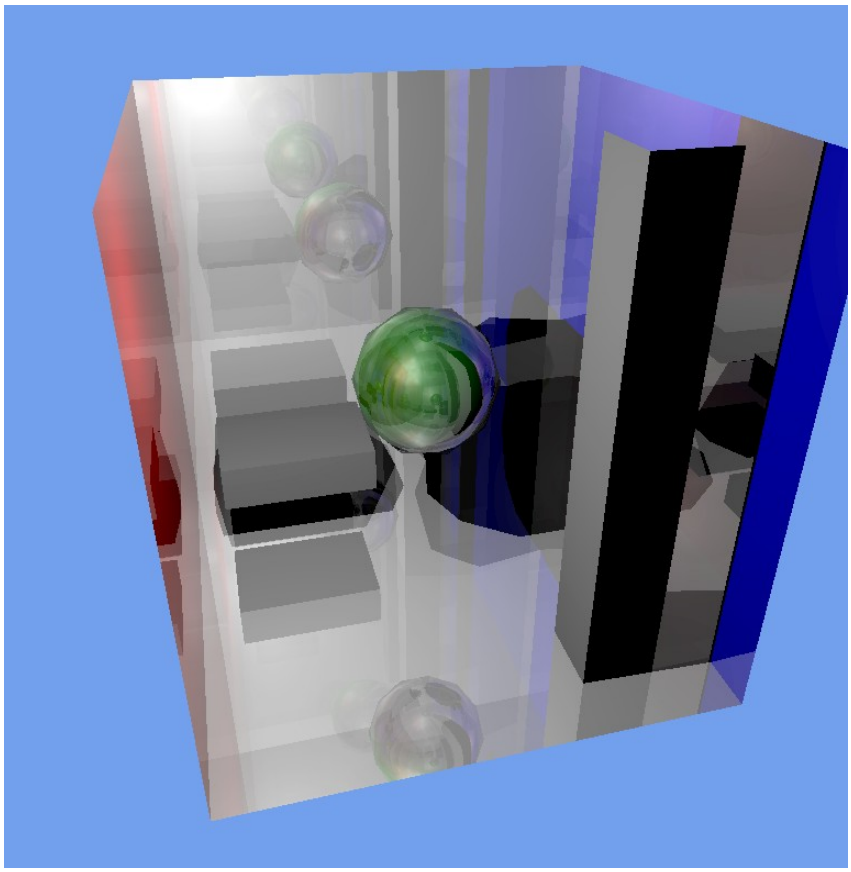
34. ábra: Árnyékok és tükröződés



35. ábra: Árnyékok



36. ábra: Tükröződés



37. ábra: Árnyékok és végtelen tükröződések

Természetesen lehetőségünk van a programot nem valósidejű képalkotásra is használni, ekkor még látványosabb képeket tud készíteni (37. ábra).

4. Lehetőségek

A dolgozatban implementált program természetesen továbbfejleszthető bizonyos irányvonalak mentén. Mivel a befoglalótesteket nem szerveztük hierarchiába, levetővé válhat az objektumok mozgatása, amellyel szintén növelhető a jelenet animációs tartalma. A fényforrások száma is növelhető azzal, hogy a programban ciklusba szervezzük a fények kezelésével foglalkozó részt. Ezek természetesen mind valamilyen fokú sebességcsökkenést vonnak maguk után. Amennyiben azonban tovább növekedik a grafikus kártyákon futtatható programok maximális mérete, lehetséges lenne a szoftverben egy vermet alkalmazva a fák bejárása, amellyel a befoglalótestek hierarchiája is alkalmazhatóvá válna.

A közelmúltat és a jelent figyelembe véve pedig felállíthatóvá válik egy jövőkép, amellyel megsejthetjük, milyen lehetőségek nyílnak meg számunkra a komputergrafika ezen területén a jövőben.

4.1. Közeljövő

A grafikuskártyák képességeinek egyértelmű fejlődéséből kifolyólag pár éven belül valószínűleg lehetségessé válik akár nagyobb, például 1024x768-as felbontásban is valós időben működtetni egy ehhez hasonló programot, akár több fényforrással és mélyebb szintű tükröződésekkel is. Egy heurisztikus élsimító algoritmussal akár a tárgyak durva éleitől is megszabadulhatnak ezek a programok.

Ezekkel a technikákkal megvan az esély arra, hogy a valós idejű sugárkövetéses algoritmusok alkalmazásának népszerűsége ugrásszerűen növekedni kezd. Mivel precízebb képet lehet vele alkotni, mint az inkrementális képszintézissel, ezért ennek nagyon sok szakterületen használt vehetnék, akár az orvostudományban, akár a filmiparban.

4.2. Távolabbi jövő

Mivel a számítógépes játékokat meghajtó motorok igen fejlett szinten alkalmazzák az inkrementális képalkotást, ezért ezen a területen áttörő eredményeket csak az igazán jó minőségű grafikával lehetne elérni. Ehhez pedig a globális illuminációs megvilágítási módszer és a fotorealisztikus technikák alkalmazására lenne szükség. Természetesen ehhez

roppant mértékű számítási kapacitásra lenne szükség, de talán egy napon majd elérik a komputerek ezt a szintet is és akkor képesek lesznek másodpercenként sokszor is az akár teljes mértékben reálisnak tűnő képek alkotására, amivel végső soron akár át is vehetik az inkrementális képszintézis helyét.

Összefoglalás

A szakdolgozat keretében ismertetésre került a Whitted-stílusú sugárkövetéses algoritmus, valamint annak néhány változatának működése, az algoritmus során alkalmazható árnyalási módszerek, köztük a Blinn-Phong árnyalással, valamint megismertük a sugárkövetéses algoritmus előnyeit és hátrányait az inkrementális képalkotással szemben, továbbá azokat a technikákat, amelyekkel a képszintézis által keletkező képet még élethűbbé lehet tenni. Azt is megtudtuk, hogy mik azok a feltételek, amelyeket teljesíteni kell egy ilyen algoritmussal működő program valós idejű futtatásához.

A matematikai háttér mellett megismerkedtünk az informatikai háttérrel, amely lehetővé teszi egy ilyen algoritmus valós idejű programban történő implementálását. Rövid betekintést nyertünk a grafikus hardverek fejlődésébe, a háromdimenziós képalkotást elősegítő szoftveres programcsomagok képalkotási módszerébe, valamint abba, hogy hogyan lehet ezeknek az eszközöknek a képességeit kihasználni egy sugárkövetéses algoritmussal működő program létrehozásához.

Nem csak elméleti, de gyakorlati oldalát is megtapasztalhattuk egy ilyen program létrejöttének, amellyel egyben reprezentálásra került az algoritmus elméleti háttere is, valamint a korszerű hardverek képessége is. Megtudhattuk, hogy milyen módon lehet az algoritmust teljes mértékben a grafikus gyorsítókártyán alkalmazni, valamint, hogy milyen lépéseket kell végezni ahhoz, hogy a lehető leghatékonyabban működjön az. A program elkészítése után annak animált képalkotási teljesítményét ellenőriztük.

A megalkotott program természetesen nem tökéletes, mint ahogy egyik program sem az, éppen ezért megnéztük annak módosítási, javítási lehetőségeit, azokat a módszereket, amelyekkel még hatékonyabbá tehető az. Ezt a folyamatot a jövőre kivetítve pedig becslést tudtunk adni arra, hogy a sugárkövetéses képalkotás területén milyen változásokra számíthatunk a közeli- és távolabbi jövőben egyaránt.

Irodalomjegyzék

[1] Andrew S. Glassner: An Introduction to Ray Tracing. Academic Press, London, 1989.

[2] James T. Kajiya: The Rendering Equation, 1986

[3] Addy Ngan, Frédo Durand, Wojciech Matusik: Experimental Validation of Analytical BRDF Models, 2004

[4] Turner Whitted: An Improved Illumination Model for Shaded Display. Communications of the ACM, Holmdel, New Jersey, 1980.

[5] John Amanatides, Andrew Woo: A Fast Voxel Traversal Algorithm for Ray Tracing, 1987

[6] Timothy J. Purcell, Ian Buck, William R. Mark, Pat Hanrahan: Ray Tracing on Programmable Graphics Hardware, 2005

[7] Martin Christen: Ray Tracing on GPU, 2005

[8] Wavefront Obj, http://local.wasp.uwa.edu.au/~pbourke/dataformats/obj/

[9] Wavefront Material Format, http://local.wasp.uwa.edu.au/~pbourke/dataformats/mtl/

[10] Tomas Möller, Ben Trumbore: Fast, Minimum Storage Ray/Triangle Intersection, 2005

[11] Dr. Szirmay-Kalos László: Számítógépes Grafika. ComputerBooks Kiadó, Budapest, 2003.

[12] Carsten Wächter, Alexander Keller: Instant Ray Tracing: The Bounding Interval Hierarchy, 2006