Debreceni Egyetem
Informatikai Kar

Sugárkövetés megvalósítása a programozható grafikus hardveren

Témavezető:
Dr. Tornai Róbert
Egyetemi Adjunktus

Készítette:
Berkes Balázs
V. Programtervező Matematikus

Debrecen, 2007

Tartalomjegyzék

1. Bevezetés

A számítógépes képalkotás az informatika területének egyik rohamosan fejlődő és igen érdekelt ága. Az ezen a területen elért fejlődéseket az élet sok területén felhasználják. Ilyen területek például az orvostudomány, a számítógépes játékipar, filmipar, építőipar, stb.

A sugárkövetés a számítógépes képalkotás egy olyan ága, amely segítségével a leginkább valóságosabb, úgynevezett fotorealisztikusabb képeket állíthatunk elő. A módszer a fény útját, annak fizikai tulajdonságait figyelembe véve próbálja meg szimulálni. Mivel viszont ez a szimulációs igen műveletigényes folyamat, ezért a sugárkövetés általában nem használható fel valósidejű alkalmazásokban.

Mivel a sugárkövetés az egyik legtökéletesebb képalkotásnak tekinthető, amit számítógépen megvalósíthatunk, ezért számítógépes képalkotás egyik legnagyobb forradalma a sugárkövetés valósidejű megvalósíthatósága lenne. Emiatt a tény miatt igen sok fejlesztési kísérlet történik ezen a területen.

A személyi számítógépeibe építtet grafikus hardverek a képelőállításhoz sokkal egyszerűbb pontosabban kisebb műveletigényű technikákat alkalmaznak, amelyek igen hatékonyak (valósidejűek) viszont az előállított kép minősége távol áll a valóságostól vagy az imént említett sugárkövetéses képalkotástól.

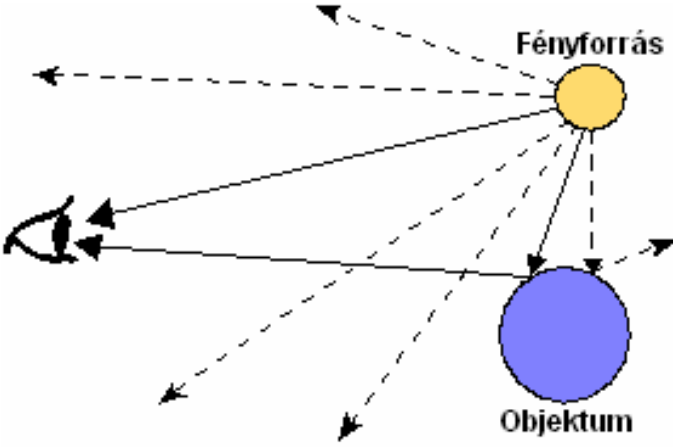
A napjainkban kifejlesztett grafikus hardverek, (bár még mindig az egyszerűbb technikákat alkalmazzák) számítási kapacitása olyan mértékben növekedett, illetve olyan új technológiák lettek bevezetve, hogy az már lehetőséget kínál általános célú műveletek elvégzésére is. Ennek kihasználását az teszi indokolttá, hogy bizonyos feladatok jól illethetőek a grafikus hardver sajátos architektúrájához, illetve esetenként kismértékű gyorsítás is elérhető segítségével.

Dolgozatomban a grafikus hardver ezen előnyeit felhasználva próbálok bevezetést nyújtani, a Sugárkövetés Grafikus Hardveren történő megvalósíthatóságába. A lehetséges megvalósítás magyarázata előtt betekintést nyújtok a sugárkövetés és a programozható grafikus hardver sajátosságaiba, kiemelve a témakör azon jellegzetességeit, amelyeket a megvalósítás során felhasználok.

2. Bevezetés a sugárkövetés fogalmaiba

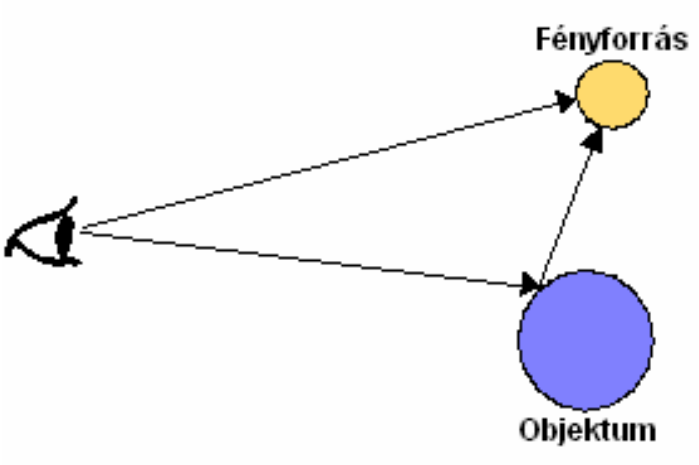
2.1. A számítógéppel történő képalkotás a valóság modellezésével

A való életben a fényt a fényforrások fotonokként árasztják szét a környezetükbe. A fotonok kölcsönhatásba lépnek objektumokkal, amely hatására szétszóródnak mindaddig, amíg a környezet el nem nyeli őket, vagy amíg el nem érnek egy képalkotó eszközt. Ilyen képalkotó eszköz lehet például egy kamera, érzékelő műszer vagy az emberi szem.
A sugárkövetés egy képszintézis technika, amely szimulálja a fény kölcsönhatását a környezetével. Tehát képeket ábrázolhatunk számítógépen, úgy hogy készítünk egy modellt (környezetet), amelyben elhelyezünk egy virtuális érzékelőt, és szimuláljuk a fény viselkedését úgy, hogy nyomon követjük a fénysugarak (fotonok) útját a fényforrástól az érzékelőig.
Ezen szimulációs technika megvalósítása viszont igen költséges, mivel a fényforrás(ok) által szétszórt fénysugaraknak csak igen csekély hányada éri el ténylegesen az érzékelőt.



2. ábra: A fényforrásból induló sugaraknak csak igen csekély hányada éri el az érzékelőt.

Emiatt a tény miatt egy másik szimulációs technikát szokás alkalmazni, amely a fényt fordított irányban az érzékelőtől kiindulva és a fényforrás felé tartva követi. Ez az eljárás a számítógépes képalkotásban nagyon hasznos, mert elkerülhetjük azon fölösleges fénysugarak szimulálását, amelyek sosem érik el az érzékelőt.



2. ábra: Ha úgy tekintjük, hogy a fénysugarak az érzékelőből indulnak ki, akkor csak a képalkotásban ténylegesen résztvevő sugarakkal kell foglalkozni.

Különféle irodalmak különféleképpen nevezik a sugárkövetés ezen két fajtáját. A fényforrásból induló szimulációt előrefelé (forward) haladó, míg az érzékelőből induló szimulációt visszafelé (backward) haladó sugárkövetésnek nevezik. Ezen két különféle elnevezés ellenére a sugárkövetés elnevezést általánosan a fénysugarakat az érzékelőből kiindulva (és a fényforrásba tartva) szimuláló technikát nevezik.

2.2. A Whitted-stílusú sugárkövetés

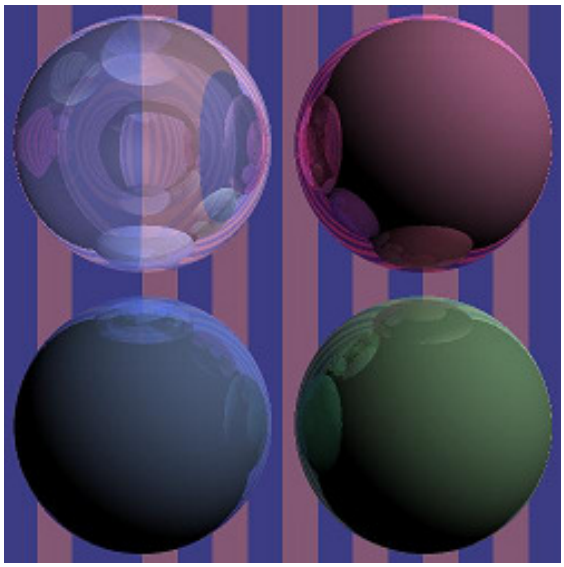
1980-ban Whitted publikált egy sugárkövető algoritmust. A Klasszikus vagy Whitted-stílusú sugárkövetés a képalkotáshoz az ábrázolandó jelenet fontos jellemzőit felhasználva alábbi hatásokat szimulálja:

- közvetlen megvilágítás: Egy objektum nem sugároz magából fényt, azaz ha két objektum közel van egymáshoz, és azok nem tükröződő és nem átlátszó objektumok, valamint egyik sem árnyékolja a másikat, akkor az objektumok színét csak a fényforrás határozza meg.

- árnyékolás: A jelenet objektumai árnyékolhatják egymást.

- tükröződés: Ha egy objektum, tükröződő felületű, akkor azon láthatóak a jelenet más objektumai. Tükröződő objektum lehet például egy tükör.

- átlátszóság: Egy objektum átlátszó, ha látszanak a mögötte lévő objektumok. Átlátszó például egy üveglap, vagy üveggömb.



3. ábra: Sugárkövetéssel történő képelőállítás eredménye tükröződés és átlátszóság megvalósításával.

Az árnyékolás, tükröződés és átlátszóság hatások megvalósítása miatt a Whitted-stílusú sugárkövetést egy globális megvilágítási algoritmus, amely azt jeleni, hogy a kiszámított kölcsönhatás a fény és egy objektum között, függ a jelenet többi objektumától (például tükröződés esetén).

Ezzel ellentétben a lokális megvilágítási algoritmusok a fény és a felület kölcsönhatásának számítása során úgy tekintik, mintha a fénnysugárral kölcsönhatásba lépett objektum lenne az egyetlen a jelenetben, így ezen algoritmussal nem ábrázolhatóak a korábban említett képhatások.

2.3. A Whitted-stílusú sugárkövetés egy egyszerű implementációja

Az alábbiakban a Whitted-stílusú sugárkövetés egy lehetséges implementációját adom meg.

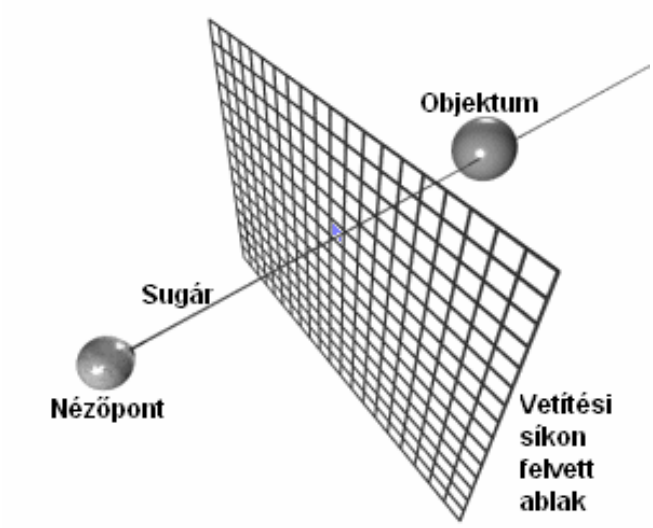
```
Ciklus az előállítandó kép minden (x , y) képpontjára  {
   sugár = ElsődlegesSugárGenerátor( x,y,nézőpontadatai,VetítésiSíkonFelvettAblakAdatai );
   Szín = Sugárkövetés( sugár );
```

```
   SzínBufferbeÍr( x , y , szín );
}

Szín Sugárkövetés( sugár ) {
   Szín = Háttérszín;
   Metszéspont = LegközelebbiSugár-ObjektumMetszéspontKeresés ( sugár );
   Ha létezik metszéspont akkor {
      Szín = ÁrnyékolóSugár (Metszéspont );
      Ha a metszett Objektum Tükröződő akkor
            Szín += Sugárkövetés( Tükrözöttsugár ) * Tükröződéserőssége
      Ha a metszett Objektum Átlátszó akkor
            Szín += Sugárkövetés( Áteresztettsugár ) * Áteresztéserőssége
   }
   return Szín;
}

Szín ÁrnyékolóSugár ( Metszéspont ) {
   Szín = Objektumszín * KörnyezetiFénySzíne;
   Ciklus a jelenet összes fényforrására {
        Sugár = ÁrnyékolóSugárGenerátor (Metszéspont , FényforrásHelyzete );
        ObjektumMetszéspont = LegközelebbiSugár-ObjektumMetszéspontKeresés ( sugár );
         Ha nem létezik metszéspont VAGY Távolság(ObjektumMetszéspont , Metszéspont) <
         Távolság(AktuálisFényforrásHelyzete , Metszéspont) akkor {
             Szín *= MegvilágítottObjektumSzín( AktuálisFényforrás , Metszéspont ,
             MetszéspontbeliNormálVektor);
         }
    }
}
```

Egy sugárkövető algoritmus működése során az első legalapvetőbb lépés azon sugarak meghatározása, amelyek ténylegesen szerepet játszanak az eredménykép előállításában. A számítógépes képalkotás digitális képek előállítására alkalmas, azaz egy kép előre

meghatározott számú képpontból áll (például 256*256 pixel). Mivel minden fénysugár pontosan egy színt határoz meg ezért a nézőpontból az alábbiak alapján indítunk fénysugarakat:

- a jelenet három-dimenziós terében elhelyezzük a nézőpontot
- meghatározunk egy vetítési síkot és azon egy ablakot, amelyet felosztunk annyi részre ahány képpontból az eredménykép áll, azaz az ablak télalap alapú területét felosztó szabályos rács az eredménykép egy-egy pixelének felel megy.
- a nézőpontból kiindulva a vetítési síkon felvett ablak rácspontjain keresztül sugarakat indítunk a jelenet objektumai felé.



4. ábra: Sugarak indítása a nézőpontból a jelenet objektumai felé a vetítési síkon felvett ablak rácspontjain keresztül.

A közvetlenül a nézőpontból indított sugarakat elsődleges sugaraknak nevezik.

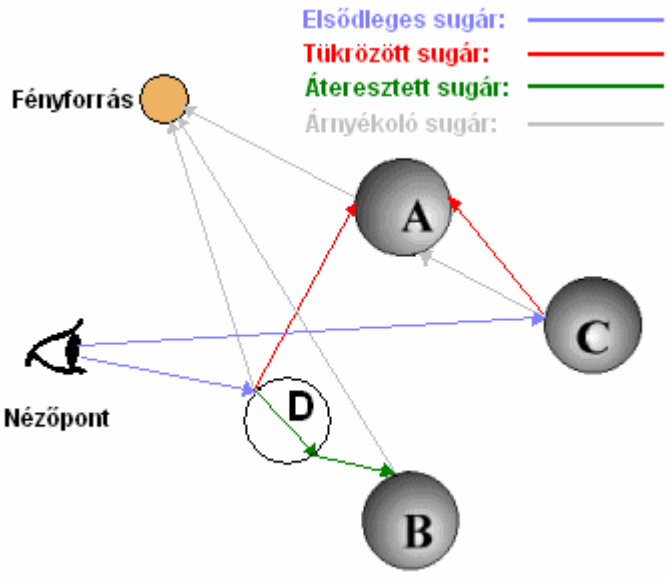
A megvalósítás során az elsődleges sugarakat olyan egyenes szakasznak tekintjük, amelyek a nézőpontból indul, és amelynek irányát az határozza meg, hogy a vetítési síkon felvett ablak mely rácspontján halad át.

A sugárkövető algoritmusok eredményét, azaz az eredménykép pixeleinek színét az elődleges sugarak színeinek meghatározása adja. Egy sugár színét a sugár által metszett és a nézőponthoz legközelebb lévő objektum színe határozza meg. Egy objektum színét egy pontban (abban a pontban ahol egy sugár elmetszi) meghatározza a megvilágítási modell, valamint az, hogy az objektum tükröződő-e, fényáteresztő-e illetve, hogy árnyékolja-e a jelenet egy másik objektuma.

Amikor a fénysugár egy objektummal kölcsönhatásba lép, az objektum anyagtulajdonságától függően a sugár iránya változhat, illetve újabb sugarak keletkezhetnek. Az algoritmus megoldása során ezt úgy tekintjük, mintha a sugár-objektum metszéspontban a sugár több részsugárra osztódna szét. Az így keletkezett részsugarak színei befolyásolják az eredeti sugár színét. Az újonnan keletkezett részsugarak útjuk során (objektumokkal való ütközés esetén) szintén részsugarakra osztódhatnak szét, így egy rekurzív feladathoz jutunk el.

Egy sugár objektum metszéspontban az alábbi új sugarak indulhatnak:

- Árnyékoló sugár: A metszéspontból egy sugarat indítunk a jelenet fényforrásába. Ha a sugár útja során objektummal ütközik, akkor az objektum (amely metszéspontjából az árnyékoló sugarat indítottuk) egy másik objektum által árnyékolt.
- Tükrözött sugár: A metszéspontból induló tükrözött sugár irányát az eredeti sugár és a metszéspontban lévő (objektum) normálvektorból számítandó. A tükrözött sugár színét összeadjuk az objektum aktuális (más sugarak által meghatározott) színéhez. A tükrözött sugár színét az elsődleges sugarakhoz hasonlóan határozzuk meg.
- Áteresztett sugár: A metszéspontból induló áteresztett sugár irányát az eredeti sugár és a metszéspontban lévő (objektum) normálvektorból, valamint az objektum fénytörési jellemzőjéből számítandó. Az objektum színe a tükrözött sugárhoz hasonlóan számítandó.



5. ábra: A képalkotásban résztvevő sugarak.

A valóságban a fénytörés rekurzióját végtelennek tekinthetjük (például két egymással szemben lévő tükör esetén) viszont a számítógépes megoldásnál maximalizálni kell a rekurzió mélységét a végtelen ciklus elkerülése végett.

2.4. A sugárkövető algoritmusok matematikai háttere

Mivel a sugarakat egyeneseknek tekintjük, valamint a jelenetet alkotó objektumok különféle grafikus primitívek lehetnek, ezért a sugárral kölcsönhatásban lévő objektum meghatározása nem más, mint egy egyenes (szakasz) és egy grafikus primitív metszéspontjának meghatározása.
Az egyenes-objektum metszéspontok meghatározása különféle matematikai képletekkel határozható meg.
Egy fénysugarat a nézőpont és a vetítési síkon felvett ablak egy rácspontja által definiált vektor segítségével paraméteres alakban írhatjuk fel.
Ha a nézőpont $C(x_0,y_0,z_0)$ (ahol x_0,y_0,z_0 egy három-dimenziós vektor koordinátái) valamint a vizsgált képpont $P(x_1,y_1,z_1)$, akkor az erre a két pontra illeszkedő egyenes (sugár) bármely (x,y,z) pontja megadható az egyenes paraméteres egyenletével:

$$x = x_0 + t * (x_1 - x_0)$$
$$y = y_0 + t * (y_1 - y_0)$$
$$z = z_0 + t * (z_1 - z_0)$$

Amennyiben

$$x_1 - x_0 = \Delta x$$
$$y_1 - y_0 = \Delta y$$
$$z_1 - z_0 = \Delta z$$

akkor

$$x = x_0 + t * \Delta x$$
$$y = y_0 + t * \Delta y$$
$$z = z_0 + t * \Delta z$$

A t paraméter 0 és 1 közé eső értékei a nézőpont és a vizsgált pixel közé eső fénysugárpontokat definiálják, míg negatív értékek esetén a nézőpont mögött vagyunk, valamint 1-nél nagyobb t értékekre a képsík túlsó oldalára kerülünk.
A fénysugár és az objektumok metszéspontjainak meghatározásához minden egyes, a jelenetet alkotó objektumtípust matematikailag le kell írnunk.

Gömb esetén a metszéspont az alábbi másodfokú egyenlet alapján számítható:

Ha G(a, b, c) a gömb középpontja és r a sugara, valamint (x, y, z) a sugár azon pontja amely metszi a gömböt akkor:

$$(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2$$

Behelyettesítve x–et y-t és z-t, a következő képlet adódik:

$$(\Delta x + \Delta y + \Delta z)^2 t^2 + 2t[\Delta x(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] +$$
$$+(x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 = 0$$

Poligon esetén a metszéspont az alábbi képlet alapján számítható:

Egy poligon síkjának az egyenlete:

$$Ax + By + Cz + D = 0$$

Behelyettesítés után t-re adódik:

$$t = \frac{Ax_0 + By_0 + Cz_0 + D}{A\Delta x + B\Delta y + C\Delta z}$$

Hacsak $A\Delta x + B\Delta y + C\Delta z \neq 0$.

Ugyanazon jeleneten belül geometriai primitíveknek igen sok fajtája fordulhat elő, amelyekkel való metszéspont meghatározás különböző képletek segítségével történhet.
Mivel bármely objektum közelíthető háromszögek segítségével, ezért a metszéspontok meghatározása általánosítható csak egyenes-háromszög metszéspontok számítására. Az objektumok háromszögráccsal való közelítése egy igen elterjedt optimalizációs lépés egy sugárkövető algoritmus megvalósítása során, mert így elegendő csak egyfajta geometriai primitívre felkészítni az eljárást.

2.5. Gyorsítási technikák

A sugárkövetés alapú algoritmusok egyik leginkább műveletigényes (leggyakrabbam végrehajtott) pontja a sugarak kezdőpontjához legközelebb eső objektum metszéspontjának meghatározása áll, ezért a legtöbb gyorsítást ezen a területen érhetünk el.
A legközelebbi metszéspont meghatározása, legdurvább esetben úgy történhet, hogy a vizsgált sugarat elmetsszük a jelenet összes objektumával, majd vesszük közülük azt, amelyik a legkisebb távolságra van a sugár kiindulópontjától. Ennek a megoldásnak a legnagyobb hibája az, hogy több száz illetve több ezer objektumból álló jelenet esetén nagyon sok metszéspontszámítást kell elvégezni, valamint az objektumok zömét a sugár el sem metszi, így nagyon sok fölösleges számítást végzünk el.
A gyorsítás egyik legfontosabb feladata ezért az, hogy eredményesen kiszűrjük azon objektumokat, amelyekkel egy sugár nem lép kölcsönhatásba.
A gyorsítás általában valamilyen térpartícionáló adatstruktúra segítségével történik. Egy térpartícionáló adatstruktúra felosztja a jelenetet kisebb részekre. Ezek után egy sugarat csak azon objektumokkal teszteljük, amely objektumok részei azon térrésznek, amely térrészen a sugár áthalad.
Egy jó gyorsító technika alkalmazása esetén csak a sugár (minél kisebb) környezetében lévő objektumokat vizsgáljuk.
Ilyen térpartícionáló gyorsító technikák például a (hierarchikus) befoglaló testek alkalmazása, BSP fák, kd fák, egyenletes négyzetháló, alkalmazkodó négyzetháló, hierarchikus négyzetháló stb.

3. A grafikus hardver

3.1. A grafikus hardver képelőállítása

Nem a sugárkövetés az egyetlen lehetőség a számítógépes grafikák létrehozására. Valójában a grafikus processzorok, amelyek manapság minden személyi számítógépben megtalálhatóak, teljesen eltérő algoritmust alkalmaznak képalkotáshoz.
A grafikus hardver a jelenetet általában, egyenesek, pontok vagy háromszögek csúcspontjait alkotó vektorok sorozataként kapja meg. Ezen vektorsorozaton kívül még további attribútumok is átadhatóak, amelyek meghatározzák az előállítandó képet. Ilyen attribútumok lehetnek a csúcspontok (elsődleges és másodlagos) színei, normálvektorai és textúra koordinátái. Egy vektort valamint a hozzá kapcsolódó fent említett attribútumokat együttesen vertex-eknek nevezzük.
A vertexeken kívül további adatok is átadhatóak a grafikus hardvernek. Ilyen adatok lehetnek a transzformációs mátrixok, az objektumok anyagjellemzői és az objektumokra ható fényforrások, köd, stb. jellemzői, valamint különféle képelőállítási megszorítások, mint például z-buffer alkalmazása, háttal álló háromszögek figyelmen kívül hagyása stb.
Az így megadott jeleneten a grafikus hardver különféle rögzített sorrendű műveleteket hajt végre, mely műveletek sorozatát grafikus csővezetéknek nevezzük.
A grafikus csővezeték tehát olyan műveletek sorozata, amely egy három-dimenziós jelenetből egy két-dimenziós raszteres képet állít elő.

Egy grafikus csővezeték minden fázisa különféle műveletek sorozatát hajtja vére.

Vertex feldolgozás:
Ez a fázis a következő műveleteket hajthatja végre:

- Vertex transzformáció: A jelenetet alkotó vertexek pozícióit és normálvektorait megszorozza a megadott különféle transzformációs mátrixokkal. A transzformációk lehetnek: eltolás, skálázás, forgatás és középpontos illetve merőleges vetítés. Ezen mátrixszorzások elvégzése után meghatározódnak a vertexek nézőpont-koordinátarendszerbeli pozíciója valamint a három-dimenziós vektorok két-dimenziós vetületeinek koordinátái.

- Megvilágítás: A vertexek normálvektorát, pozícióját, a megadott anyagjellemzőt és a fényforrások jellemzőit (pozíció, szín stb.) felhasználva, a megadott megvilágítási modell szerint beállítja a vertexek másodlagos színét.
- Textúra koordináta generálás: A vertexek textúra koordinátáit megszorozza egy transzformációs mátrixszal. Illetve lehetőség van automatikus textúra koordináta generálásra, amelyet általában a vertexek normálvektorait és a transzformációs mátrixokat felhasználva számít ki.

Vágás:

Ez a fázis hajtja végre a látótérbe való, illetve a felhasználó által megadott vágósíkokkal való vágások elvégzését a hátsó lapok elhagyását továbbá a primitívenkénti attribútumok kiértékelését. A műveletek elvégzése előtt végrehajtódik az úgynevezett primitív összeillesztés, amely során meghatározódik, hogy mely vertexek tartoznak egy grafikus primitívhez. A különféle vágások végrehajtása során újabb vertexek jöhetnek létre.

Raszterizáció:

A raszterizáció meghatározza primitíveket alkotó fragmentumok pozícióját. A fragmentum olyan adatok összessége, amelyből a későbbiek során meghatározódik egy pixel színe az adott pozícióban, tehát a fragmentum képpontokhoz köthető dolog. Egy fragmentum nem csak színértéket tárol, hanem például normál értékeket illetve textúra koordinátákat.

A fázis eredménye:

- A fragmentum helyzete.
- A fragmentumok attribútumainak értékei, amelyeket a vertexek attribútumaiból interpolál.

Fragmentum textúrázás és színezés:

Ezen fázis bemenete az interpolált fragmentum értékek, és ezen értékeket felhasználva adja eredményül a fragmentum végleges színét és a melységét. Például a fragmentum színét (amelyet a csúcsok színeinek interpolációjából kapunk) kombinálja a fragmentumhoz tartozó texel színével. A fázishoz tartozó művelet még a köd által okozott hatások számítása.

Az attribútumok áradása a grafikus csővezeték egyes fázisainak, valamint azok konfigurálása a különféle grafikus API-k segítségével történhet. Ilyen API-k a például a DirectX és az OpenGL.

3.2. A programozható grafikus hardver

A grafikus hardverek korábbi verzióiban a grafikus csővezeték csak konfigurálható volt, ami azt jelenti, hogy a jelenet és néhány további jellemző megadásán kívül csak néhány előre meghatározott képelőállítási mód közül választhattunk. Az ilyen jellegzetességű csővezetéket fix-funkciójú grafikus csővezetéknek nevezzük.
Példaként tekinthetjük a fix-funkciós megvilágítási modellt, amellyel általában csak a kibocsátott, környezeti, szórt és tükröződő megvilágítási hatásokat lehet elérni. Ez a modell széles körben használható, viszont sok más megvilágítási modell is létezik.

A grafikus hardverek technológiájának legújabb fejlesztése, az utóbbi pár évben kialakult programozható funkcionalitás, amely arra törekedik, hogy minél több a fix-funkcióssal megegyező vagy attól eltérő grafikai hatást lehessen megvalósítani a grafikus hardveren.
A grafikus hardverek legújabb fejlesztéseit a grafikus csővezeték vertex és fragmentum feldolgozási szintjének minél rugalmasabbá tételével próbálják meg elérni. Ezen szintek programozhatóvá tétele az egyik legkorszerűbb fejlesztése a mai grafikus processzoroknak.
A jelenlegi grafikus hardverek programozható részei, a vertex feldolgozó egység, amely a vertex árnyaló programokat hajtja végre, valamint a fragmentum feldolgozó egység, amely a fragmentum árnyaló programokat hajtja végre.
A vertex illetve fragmentum feldolgozó egység a központi vezérlő egységhez (CPU) hasonlóan saját speciális alacsony szintű (assembly szerű) programozási nyelvel programozhatóak. A feldolgozó egységekre írt programokat úgy tekintjük, hogy azok grafikus csővezeték megfelelő fázisait hajtják végre. A programokat árnyaló programoknak (shader-eknek) nevezik, és két fajtájuk van:

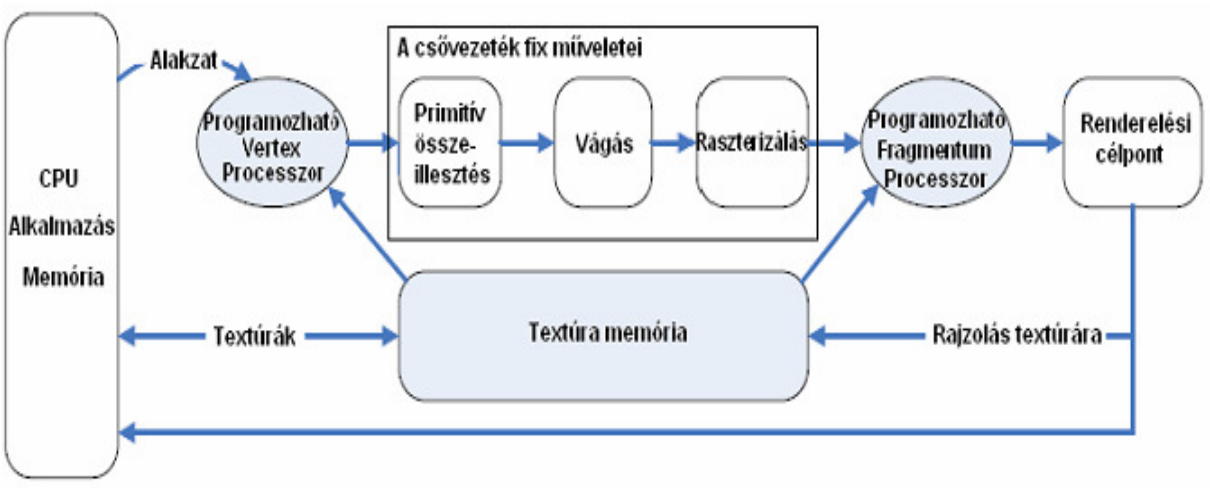
- Vertex árnyaló programok: A vertex árnyaló programokat a grafikus hardver Programozható Vertex Feldogozóegysége hajtja végre. Egy ilyen program a jelenet vertexein végez műveleteket. A műveletek lehetnek vektorok szorzása különféle

transzformációs mátrixokkal, textúra koordináta generálás és a vertexek elsődleges illetve másodlagos színeinek meghatározása tetszőleges megvilágítási algoritmust alkalmazva.

- Fragmentum vagy pixel árnyaló programok: Fragmentum vagy pixel árnyaló programok a grafikus hardver Programozható Fragmentum Feldogozóegysége hajtja végre. Egy ilyen program a jelenet fragmentumain végez műveleteket. Műveletvégzés során változtathatja a fragmentum színét, valamint egyik legfontosabb feladatai közé a texelek (egy adott textúra képpontja) fragmentumokhoz való hozzárendelése.

A vertex illetve fragmentum árnyaló programok alap adattípusai a vektorok melyeknek hossza 4 valamint 4x4-es mátrixok. Ezen típusokra vonatkozó alapértelmezett operátorok hardveresen vannak megvalósítva. Grafikus hardvertől függően további utasítások lehetnek hardveresen gyorsítottak. Ilyen utasítások például a trigonometrikus és logaritmikus utasítások.

A programozható grafikus csővezeték az alábbi ábrával modellezhető:



6. ábra: A programozható grafikus csővezeték

3.3. Alacsony és Magas szintű árnyaló nyelvek

A programozható grafikus hardverre kétféleképpen írhatunk árnyaló programokat:

- alacsony szintű (assembly szerű) nyelvekkel
- magas szintű (a C programozási nyelvhez hasonló) nyelvekkel

Az alacsony szintű nyelvek a grafikus feldolgozóegység utasításkészletével szoros kapcsolatban állnak, ezért előnyei hogy segítségükkel jobban kihasználhatóak egy grafikus hardver adta lehetőségek így optimalizáltabb, esetenként gyorsabb programok készíthetőek.
A különböző gyártók által fejlesztett grafikus feldolgozóegységek viszont különböző utasításkészlettel, regiszterekkel stb. vannak ellátva, ezért az alacsony szintű programozási nyelvei is igen eltérőek egymástól. Ez problémát jelent a programozás során, mivel az egyik gyártó hardverére megírt program nem biztos, hogy működni fog egy másik gyártó hardverén.

A probléma megoldása a magas szintű programozási nyelvek bevezetésével történt, amelyek nyelvek előnye, hogy a programozónak nem feltétlenül szükséges foglalkoznia a speciális hardver közeli dolgokkal és segítségükkel jobban átlátható programok készíthetőek mint az alacsony szintű nyelvekkel.
Az magas szintű árnyaló nyelvből több is létreött:

- HLSL (DirectX Hight-Level Shading Language) amely a Microsoft DirectX része, amelyet az nVidia-val közösen fejlesztett. A HLSL nagyon hasonló C programozási nyelvhez
- GLSL (OpenGL Shading Language) egy C (programozási nyelv) szerű programozási nyelv, amely operációs rendszer független. A legfőbb grafikus hardver gyártók elérhetővé teszik a fejlesztéseiket az OpenGL kiterjesztéseken keresztül.
- Cg (C for graphics) szintén C-szerű magas szintű programozási nyelv, amelyet az nVidia fejlesztett ki a saját grafikus kártyáira. A Cg független a grafikus API-któl valamint a fordító képes generálni árnyaló programot mind a DirectX-nek mind az OpenGL-nek.

A magas szintű programozási nyelveken megírt programokat, a processzor utasításkészletével szoros kapcsolatban álló assembly szintű kóddá kell fordítani, amelyet általában a grafikus hardverek gyártói által fejlesztett meghajtóprogramok végeznek el.
A magas szintű nyelvek hátránya (éppen a gyártófüggetlenség miatt) az, hogy a fordítóprogramok rosszabbul optimalizált alacsony szintű kódot eredményeznek, mintha azt kimondottan az adott hardver sajátosságainak figyelembevételével készítenénk el.

4. A sugárkövetés megvalósítása a programozható grafikus hardveren

4.1. A grafikus hardver általános célú felhasználása

A grafikus hardvert annak egyre rohamosabb fejlődése, valamint a programozható vertex és fragmentum feldolgozó egységek bevezetése óta egyre inkább használják általános célú műveletek elvégzésére. Ilyen általános célú műveletek lehetnek például lineáris algebrai számítások, vagy a most bemutatásra kerülő sugárkövetés megvalósítása.
Mivel viszont a grafikus hardver működése, architektúrája lényegesen eltér a központi vezérlőegységnél megszokottól, ezért az azon történő műveletvégzéshez másfajta módszerek használandók.
Mint azt korábban leírtam a grafikus hardver programozására két lehetőség kínálkozik:

- vertex árnyaló programokkal, amelyeket a programozható vertex feldolgozóegység hajt végre.
- fragmentum árnyaló programokkal, amelyeket a programozható fragmentum feldolgozóegység hajt végre.

Ezen két lehetőség közötti különbség az, hogy a grafikus csővezeték mely részén, milyen adatokkal végeznek műveletet, valamint hogy egy adott árnyaló program mikor és hányszor futhat le.
Eszerint a megközelítés szerint a vertex árnyaló programok egyszerre egy vertexen végeznek műveleteket, tehát egy ilyen árnyaló program a jelenet összes vertexére lefut egyszer. Példaként tekintsünk egy jelenetet, amely egy háromszögből áll. Ebben az esetben a vertex árnyaló program háromszor fut le.
A fragmentum árnyaló programok egy-egy fragmentumon végeznek műveletet. Egy ilyen árnyaló program annyiszor fut le, ahány értékes fragmentumból áll a kép. Azt, hogy hány ilyen van, azt a grafikus csővezeték fragmentum feldolgozási fázisát megelőző műveletek eredményei határozzák meg.
Példaként tekintsünk egy jelenetet, amely egy négyszögből (két háromszög) áll. Tegyük fel, hogy a grafikus csővezeték megelőző műveleteinek elvégzése után a négyzet éppen úgy áll,

hogy megjelenítés során pontosan az eredménykép felét takarná le. Ebben az esetben a fragmentum árnyaló program annyiszor fut le, mint az eredménykép pixeleinek számának fele.

A vertex és fragmentum árnyaló programok másik jellegzetessége az adatokhoz való hozzáférés, illetve hogy milyen adatokat szolgáltathatnak.
A vertex árnyaló programok bemenete egy vertex illetve további attribútumok, melyek a különféle grafikus API-k segítségével adhatóak át. A vertex árnyaló programok kimenetei pedig vertexek, amelyeken különféle műveletek el lettek elvégezve (például transzformációs mátrixokkal való szorzás) és amelyet a grafikus csővezeték további fázisainak továbbít, valamint további attribútumok amelyeket a fragmentum árnyaló programoknak adhat át.
A fragmentum árnyaló programok bemenetei a fragmentumok, amelyeken műveletet kell végezni, attribútumok, melyek a különféle grafikus API-k segítségével adhatóak át, és amelyeket a vertex árnyaló programok adnak át, valamint az egyik legfontosabb bemenetei a különféle textúrák. Egy fragmentum árnyaló programnak átadott textúrának bármely texele elérhető a programon belül. Így a textúrák tekinthetőek úgy, mint véletlen elérésű csak olvasható globális memóriák lennének a program számára. Éppen emiatt a tény miatt a grafikus hardver felhasználását általános célú műveletek elvégzésére a fragmentum árnyaló programok segítségével valósítják meg.
Egy fragmentum árnyaló program kimenete egy vagy több fragmentum, amelyeket a megfelelő renderelési célpont(ok)ra ír. Az hogy egyszerre hány kimenete lehet az árnyaló programnak, hardverfüggő.

4.2. A sugárkövetés programozható grafikus hardveren történő megvalósításának megközelítései

Alapvetően több megközelítés lehetséges a sugárkövetés programozható grafikus hardverrel történő megvalósítása.
Az egyik ilyet Carr prezentálta, aki a grafikus hardvert csak a sugár-háromszög metszés tesztelésére használta és így az algoritmus megvalósításának nagyobb részét a központi

vezérlőegységgel végeztette. Ez a módszer első ránézésre jónak bizonyulhat, viszont ettől jóval több műveletet is elvégeztethetünk a hardverrel.

A másik megközelítést Purcell prezentálta, aki egy teljes sugárkövetőt implementált a grafikus hardveren. Az elsődleges sugárgenerálástól kezdve a gyorsítási struktúra átszelését, metszéspontok meghatározását és az árnyékolást mind a grafikus hardverrel végeztette el. Gyorsító technikaként reguláris térháló adatstruktúrát használt, így ez volt az első térpartícionáló adatstruktúrát alkalmazó sugárkövető algoritmus grafikus hardveren.

A fejezet további részében betekintést nyújtok egy sugárkövető algoritmus megvalósíthatóságába a programozható grafikus hardver lehetőségeit felhasználva. A megvalósítás alapjaként a Purcell által prezentált módszert használom fel.

4.3. A fragmentum árnyaló programok és a sugarak kapcsolata

A sugárkövetés grafikus hardveren történő megvalósítása előtt tekintsük meg, hogy hogyan illeszthető a probléma a grafikus hardver architektúrájához, valamint mik azok a legfontosabb dolgok a grafikus hardver programozásával kapcsolatban, amelyeket figyelembe kell venni a megoldás során.

Mivel a sugárkövetéses algoritmusok működésük során annyi sugarat szimulálnak, ahány képpontból az eredménykép áll, ezért a grafikus hardveren történő megvalósítás során a sugarakkal történő műveletvégzések implementációját fragmentum árnyaló programokkal lehet megvalósítani. Mivel a fragmentum árnyaló programok pixeleken operálnak, ezért a sugarakat fragmentumokként lehet reprezentálni.

Ahhoz, hogy az eredménykép minden egyes pixelére lefusson egy fragmentum árnyaló program, a grafikus API segítségével meg kell adni egy akkora négyzetet, amely az egész képernyőt befedi.
Az OpenGL API-n keresztül ez a következő kódrészlettel tehető meg:

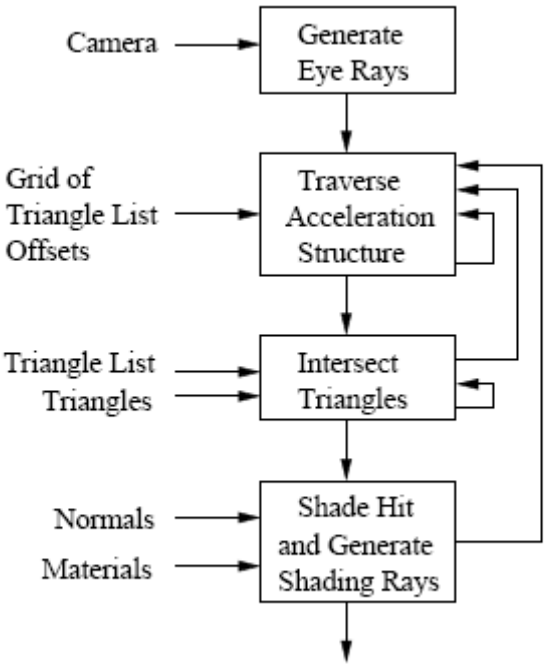
```
glViewport(0 , 0, w, h ) ;
glMatrixMode (GL_PROJECTION ) ;
glLoadIdentity ( ) ;
gluOrtho2D(−1, 1, −1, 1) ;
glMatrixMode (GL_MODELVIEW) ;
glLoadIdentity ( ) ;

glBegin (GL_QUADS ) ;
glTexCoord2f(0 ,0) ; glVertex3f(−1,−1,−0.5f) ;
glTexCoord2f(1 ,0) ; glVertex3f( 1,−1,−0.5f) ;
glTexCoord2f(1 ,1) ; glVertex3f( 1, 1,−0.5f) ;
glTexCoord2f(0 ,1) ; glVertex3f(−1, 1,−0.5f) ;
glEnd ( ) ;
```

Ezen műveletek végrehajtása alapvető fontosságú, mert csak így tudjuk kikényszeríteni, hogy egy tetszőleges méretű kép esetén minden fragmentumra lefusson az árnyaló program.

4.4. A sugárkövető algoritmus felosztása végrehajtási fázisokra

A sugárkövető algoritmus fragmentum árnyaló programokkal való implementálására első átgondolásra az kínálkozik az egyik legegyértelműbb megoldásnak, hogy az egész feladatot valósítsuk meg egy programon belül. Ez elméletileg lehetséges is lenne mivel minden képpont színét egy sugár határozza meg, valamint egy fragmentum árnyaló program az előállítandó kép minden pixelére lefut.
A grafikus hardver programozásának viszont vannak bizonyos korlátoltságai, amelyek miatt a sugárkövetés algoritmusát nem lehetséges egyetlen fragmentum árnyaló programmal megvalósítani. Ilyen korlátoltságok lehetnek például azok, hogy a jelenlegi grafikus hardverek nem támogatják a rekurziót és a legtöbb grafikus hardver esetében egy programon belül maximalizálva van a végrehajtható egyes utasításfajták száma. Másik ilyen korlát a vezérlési szerkezetek támogatása, azaz a legtöbb grafikus kártya még nem támogatja az egyes vezérlési szerkezeteket (például a ciklusok különböző fajtáit).

Ahhoz, hogy a legtöbb grafikus hardveren megvalósítható legyen; a rekurzív sugárkövetést az alábbi végrehajtási fázisokra kell (lehet) szétosztani:



A fázisok mindegyikét fragmentum árnyaló programok valósítják meg. Az egyes fázisok végrehajtását úgy tekinthetjük, hogy az adott fragmentum árnyaló program az előállítandó kép minden pixelére lefut. Így tekinthetjük úgy mintha az egyes fázisok kimenetele, egy vagy több akkora méretű textúra, mint mekkora méretű az előállítani kívánt kép mérete. A fázisok bemeneti különféle konstansok, valamint textúrák lehetnek. A bemenetként kapott textúrák vagy egy megelőző fázis eredményei vagy külsőleg létrehozott textúrák lehetnek.
Mivel az algoritmus végrehajtása során általában vektorokkal végzünk műveletet, ezért a fázisok bemeneteiként illetve eredményeként szolgáló textúrákban is vektorokat (illetve azok koordinátáit) kell tárolnunk. A vektorok koordinátáinak tárolásához három vagy négy komponensű, (pozitív és negatív) lebegőpontos értékeket tartalmazó textúrákat kell használni.

Mint az a folyamatábrán is látszik; egy fázis befejeződése után a vezérlés átadódik vagy ugyanarra a fázisra vagy egy másikra.

A fragmentum árnyaló programok nem tudják azt meghatározni, hogy miután befejezték a működésüket, melyik másik program fusson le.
Erre az egyik kínálkozó megoldás, hogy az egyes különálló programokat sorban egymás után hajtsuk végre, és fragmentumonként vezessünk be segédértékeket, amely azt jelzi, hogy egy fragmentumra éppen melyik program fusson le. Hogy ezt hogyan lehet kivitelezni azt a későbbiekben, az egyes fázisok kifejtésénél mutatom be.
Az egyes fragmentum árnyaló programok egymás utáni végrehajtását a választott grafikus API végzi el. A grafikus API feladata tehát annyi, hogy sorban átadja a fragmentum árnyaló programokat végrehajtásra a grafikus hardvernek, valamint biztosítania kell azt, hogy az egyes fázisok megkapják a megfelelő bemenetüket (attribútumok, textúrák).
Egy fragmentum program csak annyit tehet, hogy a plusz átadott információ alapján eldönti, hogy az adott műveletet el kell-e végezni.

4.5. A jelenet tárolása és a gyorsítási struktúra kiválasztása

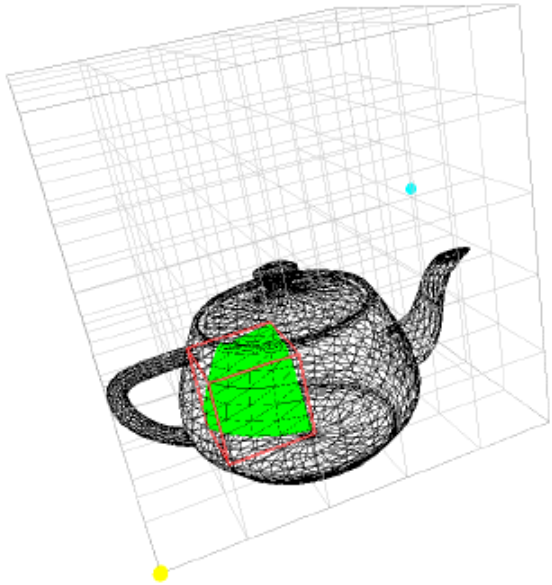
Mielőtt a sugárkövetés megvalósításának egyes fázisait végrehajtó fragmentum árnyaló programok megkezdenék működésüket a jelenetet tárolni kell. A jelenet tárolásába beletartozik a választott gyorsítási struktúra létrehozása is. Mivel a fragmentum árnyaló programok globális (csak olvasható) memóriaként textúrákat használhatnak, ezért a jelenetet illetve a gyorsítási struktúrát textúrákban kell tárolni.

Bár a gyorsítási struktúrák felépítése szerves része a sugárkövető algoritmusoknak a megvalósítását mégsem a grafikus hardver fogja elvégezni. Ennek legfőbb okai, hogy a struktúra felépítését általában elegendő csak egyszer végrehajtani, valamint az árnyaló programok korlátoltságai (például az egy programon belül lévő utasítások számának korlátjai) és a feladat bonyolultsága miatt ezt a feladatot költségesebb lenne a grafikus hardverrel végeztetni.

Így tehát a jelenet tárolását, valamint a gyorsítási struktúra létrehozását a központi vezérlő egység valamint a grafikus API végzi el. A grafikus API feladata csupán annyi, hogy létrehozza a megfelelő méretű és tulajdonságú textúrákat.

4.5.1. A reguláris térháló gyorsítási technika

A reguláris térháló az egyik legkönnyebben megvalósítható térbeli gyorsító technika a jelenlegi grafikus hardvereken, mivel minimális adatelérés szükséges a gyorsítási struktúra átszelése során, valamint az átszelés műveletigénye lineáris. Az egyetlen hátránya, hogy csak statikus objektumok esetében alkalmazható, ami azt jelenti, hogy ha a jelenetben változás történik, akkor az egész gyorsítási struktúrát újra fel kell építeni.
A továbbiakban a jelenet objektumait úgy tekintjük, hogy azok háromszögekkel vannak közelítve, így a jelenetben csak egyfajta geometriai primitív szerepel.
A reguláris térháló megvalósítása során a jelenetet felosztjuk egyenlő méretű voxelekre. Ha egy voxel, tartalmaz egy háromszöget, illetve egy háromszögrészt, akkor az adott voxel kap egy hivatkozást arra a háromszögre. Tehát a voxeleket tekinthetjük úgy mint háromszögek tárolóját.



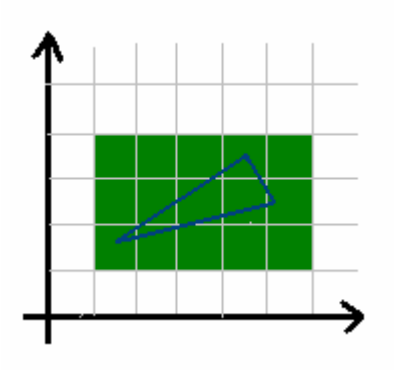
Egy voxel a jelenetet felépítő háromszögek egy részét tartalmazza

A reguláris térháló gyorsító adatstruktúrát az alábbi lépésekkel építhetjük fel:

- A voxelek méretének meghatározása. Esetünkben minden voxel megegyező méretű és így kockának tekinthető.

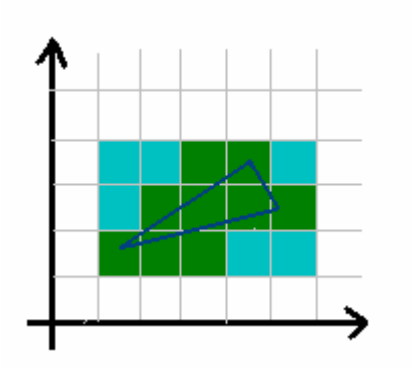
Ezek után a jelenet minden *H* háromszögére a következőket kell elvégezni:

- A *H* háromszöget körülvevő voxelek meghatározása.

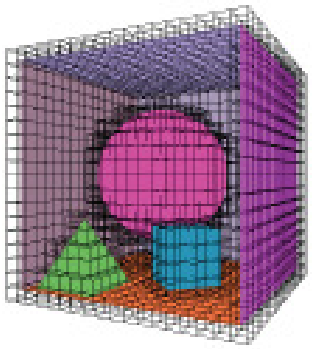


Egy háromszöget határoló voxelek

- Háromszög-kocka metszéspont teszt a határoló voxelek mindegyikével és a háromszöggel.
- Ha a háromszög és egy voxel metszi egymást akkor a voxelhez hozzáadjuk a háromszög referenciáját.



A zöld színnel jelöl voxelek kapnak egy referenciát a háromszögre.



Egy komplett jelenetben szerepet játszó voxelek.

4.5.2. Az adatok tárolása textúrákban

Az előzőekben leírtak alapján a jelenet objektumai valamint a térbeli elválasztó struktúra az alábbi módon tárolhatóak.

Elsőként tároljuk le a jelenet háromszögeinek csúcsainak a koordinátáit, illetve a csúcsokhoz tartozó további adatokat. További adatok a csúcsok normálvektorai, textúra koordinátái, színei továbbá az anyagjellemzők. Következő lépés a gyorsító struktúra tárolása, amely tartalmazza a megfelelő hivatkozásokat az előzőekben létrehozott háromszögadatokra.

A koordináták tárolására legalkalmasabb egy olyan textúra, amely alkalmas négykomponensű (RGBA), lebegőpontos értékek tárolására. Másik jellegű adatok a különféle indexek tárolása. Ilyen index lehet például az egy háromszöghöz tartozó csúcsvektorok indexei, valamint az egy voxel által tartalmazott háromszögek indexei. Az indexek tárolása történhet egész értékek tarolására alkalmas textúrákban. A textúra dimenzióját tekintve érdemes minél kevesebb külön textúrában tárolni a dolgokat, mert a grafikus hardverek erősen limitálják az egy időben hozzáférhető textúrák számát.

4.6. A megvalósítást végző árnyaló programok magyarázata

Elsődleges sugárgenerátor:

Az elsődleges sugár generátor az egyik legegyszerűbb a fázisok közül. Fő feladata meghatározni az elsődleges sugarak kezdőpontját és irányát.

A feladatot megvalósító árnyaló program bemenetként megkapja a nézőpont koordinátáit, valamint a vetítési síkon felvett ablak négy sarkának koordinátáit. Ahhoz, hogy meghatározzuk, hogy a vetítési síkon felvett ablak mely rácspontján halad át a sugár, szükség

van arra, hogy tudjuk azt, hogy a fragmentum árnyaló program mely képpontra fut épen le. Ezt az információt többféleképpen is megtudható. Vagy lekérdezhető a programon belül, vagy a fragmentum árnyaló programnak átadott interpolált textúra koordináta érték segítségével deríthető ki.

Amint megvan az aktuális képpont, akkor abból könnyen ki lehet számítani a vetítési síkon felvett ablak megfelelő pontját, és ha ez az információ is rendelkezésünkre áll, akkor abból az aktuális képponthoz tartózó elsődleges sugár irányát.

A végrehajtás ezen fázisának eredménye tehát a sugár kezdőpontját illetve irányát tartalmazó, négy komponensű (RGBA) textúra. A sugár kezdőpontját tartalmazó textúra az elsődleges sugarak esetén ugyanazokat az értékeket fogja tartalmazni. Ezen adatot mégis le kell tárolni sugaranként, mert a végrehajtás későbbi fázisaiban, a sugarak kezdőpontja változhat (például amikor tükröződik egy sugár).

Mivel egy vektor csak három komponensből áll (x, y, z érték), elegendő lenne csak háromkomponensű textúrák alkalmazása, viszont a negyedik komponens jól használható sugarankénti információ tárolására.

Az egyik ilyen használandó információ annak jelzése, hogy egy sugár éppen milyen állapotban van. A végrehajtást végző fázisok így csak azokkal a sugarakkal foglalkoznak, amelyek számukra fontosak. Tehát ezen a ponton jön elő, az hogy a fázisokat megvalósító fragmentum árnyaló programokat elegendő sorban egymás után végrehajtani. Azt hogy egy fragmentum tekintetében végre kell-e hajtani a műveletet, azt ezen extra értékekkel jelezhetjük.

Ezen a felfogás szerint megkülönböztetjük a sugarakat aszerint, hogy a végrehajtás mely fázisában tart éppen. Eszerint egy sugár állapota lehet *kezdeti, érvénytelen, átszelő, metszésponttesztelő* illetve *árnyékoló* állapotban. Ezen értéket tárolja a továbbiakban a sugár kezdőpontját tartalmazó textúra megfelelő komponense.

Ha ezen fázisnak még átadjuk az egész jelenetet befoglaló legnagyobb téglalap határcsúcsainak koordinátáit, akkor egy egyszerű sík-egyenes metszéspont meghatározással valamint néhány egyszerű vizsgálattal kiszűrhetőek azon sugarak, amelyek biztosan nem játszanak szerepet a képelőállítás során. Ezen sugarakkal a továbbiakban nem akarunk foglalkozni így állapotukat *érvénytelen*re állítjuk.

Az elsődleges sugárgenerátor fázis az *érvénytelen* sugarakon kívüli minden más sugár *kezdeti* állapotban van.

Térpartícionáló adatstruktúra átszelő:

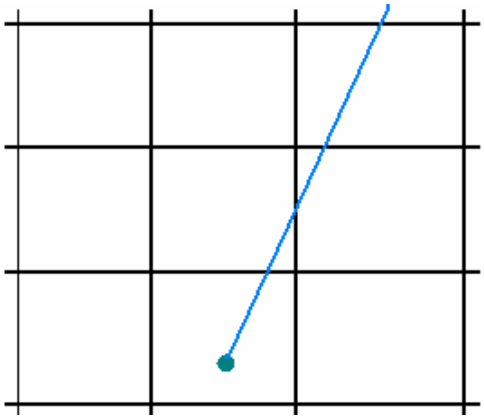
A térpartícionáló adatstruktúra átszelő műveleti fázis felsorolja a sugár által átmetszett voxeleket, a reguláris térháló gyorsítási struktúrában. A reguláris térháló átszelése egy 3 dimenziós vonalrajzoló algoritmusnak tekinthető.

Az átszelő fázis két lépésre osztható. Első lépése az inicializáció, a második lépése pedig végrehajtó rész amely feladata azon voxelek felsorolása, amelyeket az aktuális sugár átszel, mindaddig amíg az aktuális voxel nem tartalmaz háromszöget.

A térpartícionáló adatstruktúra átszelő inicializációs része bemenetként megkapja a sugarak irányát és kezdőpontját tartalmazó textúrát, valamint a rácstérkép adatait tartalmazó textúrát. Eredménye az átszelés következő részét segítő paramétereket tartalmazó textúrák lesznek. Ha az a voxel amelyben az adott sugár kezdőpontja található tartalmaz háromszöget akkor az aktuális sugarat *metszésponttesztelő* állapotba teszi, az összes többi sugarat pedig *átszelő* állapotba.

Az eredmény előállításához előbb bevezetést nyújtok egy három-dimenziós vonalrajzoló alapjaiba amely jól használható esetünkben a voxelek felsorolására. A bevezetéshez használt ábrákat a jobb átláthatóság miatt két-dimenzióban tüntetem fel.

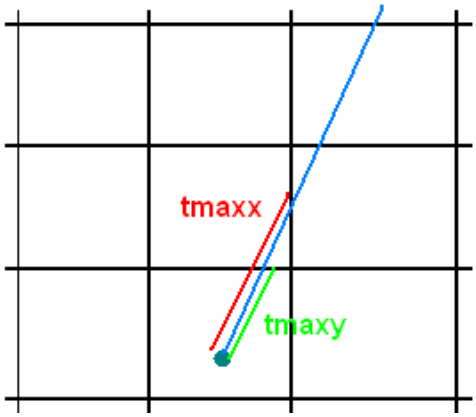
Adottnak tekintjük a sugár kezdőpontját, irányát illetve a voxelek méretét.



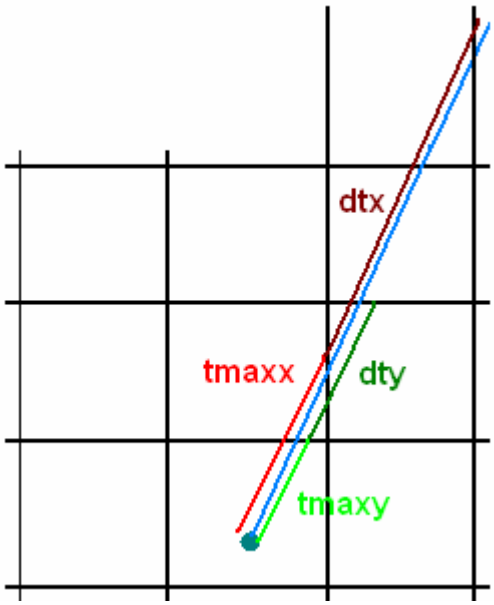
Ekkor a sugár bizonyos pontokban metszi a vízszintes és függőleges egyeneseket.

Első lépésben határozzuk meg a sugár kezdőpontjához az adott irányba legközelebb eső vízszintes és függőleges egyenesek metszéspontját. A metszéspontot határozzuk meg

paraméteres alakban, így legyen *tmaxx* a legközelebbi függőleges egyenes metszéspontját meghatározó paraméter és *tmaxy* pedig a legközelebbi vízszintes egyenesét.



Ezek után határozzuk meg az adott irányba következő vízszintes és függőleges egyenesek metszéspontjának paraméterét, és vonjuk ki ezekből rendre *tmaxx*-et és *tmaxy*-t. A keletkező mennyiséget nevezzük *dtx*-nek és *dty*-nak.



Ezek után azon voxelek felsorolása amelyeket a sugár átszel a következőképpen történik:

1. Az első felsorolandó voxel a sugár kezdőpontját tartalmazó voxel, amit konkrétan úgy kaphatunk meg, hogy a kezdőpont koordinátáit elosztjuk a rácsmérettel, és vesszük az egész részét.
2. Számítsuk ki *tmaxx*-et és *tmaxy*-t, valamint *dtx*-et és *dty*-t
3. Ha *tmaxx* < *tmaxy* akkor a következő voxel metszéspont a sugár egyenesének *tmaxx* paraméterű pontja. *tmax* = *tmax*+*dtx*. Ezt a lépést addig ismételjük, amíg szükséges.

Az imént bemutatott művelet könnyen általánosítható három dimenzióra, ahol is egyenes-egyenes metszéspont meghatározás helyett egyenes-sík metszéspont meghatározások szükségesek.
Tehát a térpartícionáló adatstruktúra átszelő inicializációs részének eredményei a *tmaxx*, *tmaxy* és *tmaxz* értékeket tartalmazó textúra, amely értékekből a mindig az átszelés során a sugár által metszett aktuális voxel indexeit lehet kiszámitani (min{ *tmaxx*, *tmaxy, tmaxz* }). Ezen fázis további kimenetelei a *dtx*, *dty és dtz* értékeket tartalmazó textúra, valamint a sugár kezsőpontját tartalmazó textúra, amely a sugarak állapotának megváltoztatásához kell.

A térpartícionáló adatstruktúra átszelő végrehajtó része fogja ténylegesen felsorolni a sugár által metszett voxeleket. Ezen fázis egyik legegyszerűbb megoldása az lenne, ha a folyamat egy ciklusban addig lépkedne végig a voxeleken, amíg azok nem tartalmaznak háromszöget. Most viszont egy olyan megoldást mutatok be, amely segítségével megkerülhetőek a fragmentum árnyaló programon belüli ciklusok alkalmazása. Ez módszer hasznos lehet akkor, ha az algoritmust olyan grafikus hardveren akarjuk megvalósítani, amely még nem támogatja a ciklusokat.
A fázis bemenetként megkapja a sugarak irányát és kezdőpontját tartalmazó textúrákat, a rácstérkép adatait tartalmazó textúrát valamint a *tmaxx, tmaxy* és *tmaxz* és a *dtx*, *dty és dtz* értékeket tartalmazó textúrákat.
A fragmentum árnyaló program a voxelfelsorolás csak egy lépését kell hogy megvalósítsa.
Ezen fázis a következőket hajtja végre:

- a *tmaxx, tmaxy* és *tmaxz* értékek közül kiválasztja a legkisebbet. Ezen értéket, (mint a sugár egyenesének paraméterét) felhasználva meghatározza a sugár által a sugár középpontjához legközelebb eső, a felderítés során még be nem járt voxel indexeit.
- A kiválasztott min{*tmaxx, tmaxy* és *tmaxz*} értékhez hozzáadja a neki megfelelő a *dtx*, *dty és dtz* értéket. Például ha min{*tmaxx, tmaxy* és *tmaxz*} = *tmaxx*, akkor *tmaxx* = *tmaxx* + *dtx*.
- a rácstérkép adatait tartalmazó textúrában megvizsgálja, hogy az adott voxel tartalmaz-e háromszöget.
- Ha nem tartalmaz akkor a sugár állapotát továbbra is *átszelő* állapotban hagyja.
- Ha az aktuális voxel tartalmaz háromszöget, akkor a sugár állapotát megváltoztatja *metszésponttesztelő* állapotúra.

- Ha a sugár olyan voxel metsz el, amelyből arra lehet következtetni, hogy az már nem háromszöget tartalmazó voxelt átszelni akkor a sugár állapotát *érvénytelen* állapotba teszi.

Tehát ezen fázis eredménye az új *tmaxx, tmaxy* és *tmaxz* értékeket tartalmazó textúra, valamint a (esetenként új állapot értéket tartalmazó) sugár kezdőpontját tartalmazó textúra.

Ezen a ponton megjegyezném, hogy annak a feladatnak az elvégzése, amely az egyes fragmentum árnyaló programok eredményeit a megfelelő textúrákba másolja a grafikus API feladata.

A metszéspont tesztelő:

Ezen fázis csak a *metszésponttesztelő* állapotú sugarakkal foglalkozik. Feladata, hogy megvizsgálja, hogy egy adott sugár egy adott voxel által tartalmazott háromszögek valamelyikét elmetszi-e a voxel határain belül.

Bemenete a sugár irányát és kezdőpontját tartalmazó textúrák, a rácstérkép adatait tartalmazó textúrák, a háromszögek csúcspontjait tartalmazó textúra, a háromszögek csúcsainak indexeit tartalmazó valamint a *tmaxx, tmaxy* és *tmaxz* értékeket tartalmazó textúra.

A fázis a ciklusok elkerülése végett egy lépésben csak egy háromszöggel tesztel, ezért valamely olyan textúrában, amelyet bemenetként is megkap valamint kimenetként is szolgáltat, tárolni kell, hogy az adott voxel hányadik háromszögét teszteltük már le. Legyen ez a kimenet a *tmaxx, tmaxy* és *tmaxz* értékeket tartalmazó textúra, tehát ezen textúra negyedik komponense egy számot fog tartalmazni, amely azt mutatja hogy hányadik voxelen belüli háromszöget vizsgáljuk.

Működésének lépései:

- A bemenetként kapott adatokból az aktuális voxel, valamint a tesztelendő háromszög (csúcspontjainak) meghatározása.
- Háromszög-egyenes metszéspont meghatározása (ha van).

- Ha van metszéspont az adott háromszöggel, valamint a metszéspont a voxel határain belül van, akkor az adott sugarat *árnyékoló* állapotba teszi, valamint az kimenetet megfelelően beállítja.
- Ha nincs metszéspont, vagy a metszéspont a voxel határain kívül van, akkor az adott sugarat *metszésponttesztelő* állapotban hagyja, ha van még tesztelhető háromszög az adott voxelen belül, ha már nincs több háromszög, akkor a sugarat visszaminősíti *átszelő* állapotúvá.

A fázis kimenetelei a sugarak állapotát is tartalmazó sugárkezdőpont tároló textúra, *tmaxx, tmaxy* és *tmaxz* értékeket valamint a tesztelendő háromszög sorszámát tartalmazó textúra, valamint egy új kimenet a metszéspontról kell, hogy adatokat tartalmazzon. Ezen kimenetben tárolni kell a metszéspontot valamint annak háromszögnek az indexét, amelyre a metszéspont vonatkozik.

Az árnyékoló:

Ez a fázis végzi el azon műveleteket amelyek hatására ténylegesen azt mondhatjuk, hogy egy sugárkövető algoritmust valósítottunk meg. A fázis szükséges feladatát a Witted-stílusú sugárkövetést alapul véve határozom meg.

Tehát a fázis feladatai:

- egy adott (háromszöget metsző, vagy *érvénytelen* állapotú) sugár színének meghatározása.
- az eredménykép adott képpontjának színének beállítása az adott sugár jellegének figyelembevételével. Például ha egy sugár nem árnyékoló sugár és érvéntelen állapotú, akkor az eredménykép aktuális képpontjának színét, (amelyet esetleg már egy másik jellegű sugár meghatározott) a választott háttérszínnel kell megszorozni.

Ezen fázis a színek meghatározásához megkülönböztet különféle jellegű sugarakat. Eszerint egy sugár lehet elsődleges, árnyékoló, tükröződött illetve áteresztett.

A árnyékoló fázis műveletei közé tartozik még a különböző jellegű sugarak új kezdőpontjának, illetve irányának, meghatározása.

6. Összefoglalás

A dolgozatomban bemutatásra került sugárkövető algoritmus, programozható grafikus hardveren (GPU) történő megvalósításának csak egy lehetséges változata. Mivel a manapság forgalomban lévő grafikus kártyák képességei igen eltérőek, ezért nehéz olyan optimális konkrét megoldást létrehozni, amely azok mindegyikén működik.

Dolgozatom egy jó betekintést nyújt, a programozható grafikus hardver általános célú felhasználására, valamint egy olyan kiindulópontot biztosít a sugárkövetés megvalósíthatóságához, amelyet követve lehetőség kínálkozik a feladat implementálására a nem legmodernebb grafikus hardvereken is.

Mivel a sugárkövetés, valamint a programozható grafikus hardveren történő általános célú műveletvégzés napjaink egyik rohamosan fejlődő és legtöbb új gyorsítási technikát produkáló tudományága, ezért a bemutatott problémával kapcsolatban is egyre több olyan publikáció lát napvilágot, újabb ötleteket, valamint esetenként gyorsabb megvalósítást biztosít.

Összefoglalva az elért eredményeket, a programozható grafikus hardver (főként az erősen párhuzamosított architektúrája miatt) jól alkalmazható a sugárkövetés számításigényes műveleteinek gyorsítására. Bár a GPU sosem lesz a feladatra specializált hardver, viszont annak rohamos fejlődése a közeljövőben lehetővé teheti a számítógépes grafika egyik nagy előrelépésének megvalósítását, ami nem más, mint a valósidejű sugárkövetés.

Irodalomjegyzék

1. Martin Christen, Marcus Hudritsch, Wolfgang Engel: Ray Tracing on GPU, Diploma thesis, University of Applied Sciences Basel (2005)

2. Timothy J. Purcell, Ian Buck, William R. Mark, Pat Hanrahan: Ray Tracing on Programmable Graphics Hardware, Stanford University

3. Andrew Wood, Brendan McCane, and Scott A. King: Ray Tracing Arbitrary Objects on the GPU, University of Otago, Department of Computer Science.

4. Schwarcz Tibor: BEVEZETÉS A SZÁMÍTÓGÉPI GRAFIKÁBA, Debreceni Egyetem

5. Niels Thrane, Lars Ole Simonsen: A Comparison of Acceleration Structures for GPU Assisted Ray Tracing (2005)

6. Timothy John Purcell: RAY TRACING ON A STREAM PROCESSOR

7. Tor Dokken, Trond R. Hagen, Jon M. Hjelmervik: The GPU as a high performance computational resource

8. Greg Humphreys: Ray Tracing Acceleration Techniques. University of Virginia (2003)

9. By Richard S. Wright Jr., Benjamin Lipchak: OpenGL SuperBible, Third Edition